Noyau UNIX
Travaux Dirigés 2002/2003

02/11/02

Spécialité Système / Réseaux 2002
ARANTES Luciana (luciana.arantes@lip6.fr)
OGEL Frédéric (frederic.ogel@inria.fr)
THOMAS Gael (gael.thomas@lip6.fr)

Table des Matières

TD 1 - Utilisation des sockets

Question 1

Rappelez ce qu'est une socket.

Reponse

Un TSAP (Transport Service Access Point). Abstraction se placant au dessus d'un protocole de transport (TCP ou UDP) et offrant une api de communication (envoi et reception de messages). A noter le mode "raw" permettant la mise en oeuvre de protocoles juste au dessus d'IP (ex ping).

+ eventuellement un dessin...

Question 2

Detaillez les structures de donnees utilisees pour representer les differents types d'addresses.

Reponse

```
Les bibliotheques concernees sont

        #include        <sys/socket.h>
        #include        <netinet/in.h>
        #include        <netdb.h>


*------------------------------------------------*
:                                                :
:       Quel reseau ?   ===>    netent           :
:                               ------           :
*------------------------------------------------*


struct netent
        {
        char *  n_name          ;       /* nom officiel du reseau       */
        char ** n_aliases       ;       /*                              */
        int     n_addrtype      ;       /* type d'adresses du reseau    */
        int     n_net           ;       /* ndu reseau -host byte order */
        }

*------------------------------------------------*
:                                                :
```

```
:       Quelle machine ? ===>   hostent         :
:                               -------         :
*-----------------------------------------------*

        struct  hostent
                {
                char *  h_name          ;/* nom officiel du hote        */
                char ** h_aliases       ;/* alias (liste)               */

                int     h_addrtype      ;/* type d'addresse du hote     */
                int     h_length        ;/* longueur de cette addresse  */
                char ** h_addr_list     ;/* addresses from name server  */
                }

        #define h_addr h_addr_list[0]

*-----------------------------------------------*
:                                               :
:       Quel service ?  ===>    servent         :
:                               -------         :
*-----------------------------------------------*

        struct servent
                {
                char *  s_name;         /* nom officiel du service      */
                char ** s_aliases;      /* alias (liste)                */
                int     s_port;         /* N de port du service        */
                char *  s_proto;        /* protocole a employer         */
                }

*-----------------------------------------------*
:                                               :
:       Quel protocole ? ===>   protoent        :
:                               --------        :
*-----------------------------------------------*

struct protoent
        {
        char *  p_name          ;       /* nom officiel du protocole    */
        char ** p_aliases       ;       /* alias du protocole           */
        int     p_proto         ;       /* ndu protocole               */
        }

*-----------------------------------------------*
:                                               :
:       Qui appeller ?  ===>    sockaddr_in     :
:                               -----------     :
*-----------------------------------------------*


        <netinet.h>     <------->       on est dans le monde Internet
                                        (pour unix ancien style on aurait
                                                sockaddr_un,  etc)

        struct sockaddr_in
                {
                short           sin_family      ;/* type d'adresse      */
                u_short         sin_port        ;/* 'u_' pour 'unsigned'*/
                struct in_addr  sin_addr        ;
```

```
char          sin_zero[8]     ;/*     (bourrage)      */
}
```

Question 3

Rappeler la difference entre mode connecte et non connecte. Quelles sont les etapes d'initialisation du serveur et du client en mode connecte ? en mode non connecte ?

Reponse

- MODE CONNECTE:
  - Pour le serveur et le client: Creer un point de demi connexion: socket [type_reseau, type_protocole, protocole_detaille] ;
  - Etablir la connexion; pour le serveur:
    - Dire "Je suis a l'adresse locale...": bind [socket, adress_locale, longueur_adresse] ;
    - Dire "...et je vous attend !": listen [socket, taille_file_d_attente] ;
    - Dire "... Entrez !": nouveau _socket = accept [socket, adresse, longueur_adresse] ;
  - Etablir la connexion; pour le client: Sonnez a la porte: connect [socket, adresse _destination, longueur _adresse] ;
  - Fermer la connexion (par l'extremite ou on est): close [socket] ;

  ATTENTION:

  Le client utilise un seul socket !

  Le serveur utilise un socket de connexion (socket, bind) ET un socket de conversation (send, recv, write, read) la fonction essentielle de 'accept' est de fabriquer un socket de conversation pour le client qui appelle !

- MODE NON CONNECTE:
  - Pour le serveur et le client: Creer un point de demi connexion: socket [type_reseau, type_protocole, protocole_detaille] ;
  - établir la liaison; pour le serveur: Dire "Je suis a l'adresse locale...": bind [socket, adresse_locale, longueur_adresse] ;
  - Fermer la connexion (par l'extremite ou on est):close [socket] ;

Question 4

Quelles primitives sont utilisees pour l'envoi et la reception de messages ?

Reponse

- MODE CONNECTE: Pour le serveur et le client:

  Utiliser la connexion etablie (apres bind-listen-accept–connect):

  write [socket, tampon, longueur] ; read [socket, tampon, longueur] ;

  send [socket, tampon, longueur, flag] ;

  recv [socket, tampon, longueur, flag] ;

- MODE NON CONNECTE: Pour le serveur et le client: Utiliser la liaison

  sendto [socket, tampon, long_tampon, flags, adr, long_adr];

  recvfrom [socket, tampon, long_tampon, flags, adr, long_adr] ;

Question 5

Ecrire le squelette d'un serveur en mode connecte (et en mode non connecte ?) Ecrire le squelette des clients correspondant.

Reponse

```
MODE CONNECTE:

/* ------------------------------------------------------------------- */
/*              Processus client.                                      */
/* ------------------------------------------------------------------- */

/* ------------------------------------------------------------------- */
/*      include's standards                                            */
/* ------------------------------------------------------------------- */

#include        <stdio.h>

#include        <sys/types.h>
#include        <sys/socket.h>
#include        <netinet/in.h>
#include        <netdb.h>

/* ------------------------------------------------------------------- */

int             main            (int argc, char * argv[])
        {
        int                     s               ;
        int                     len             ;
        struct  sockaddr_in     sa              ;
        struct  hostent *       hp              ;
        char                    buf [BUFSIZ+1]  ;

        char *                  myname          ;
        char *                  host            ;
        int                     PORT_NUMBER     ;
        char *                  requete         ;

/* ------------------------------------------------------------------- */
/*      Traitement de la ligne de commande                             */
/* ------------------------------------------------------------------- */

/* Traitement classique des parametres d'une main. */

        myname = argv[0] ;              /*      nom du programme        */
        if (argc != 4)
                {
                fprintf (stderr, "\nUtilisation: machine  port \"requete\"\n");
                exit(1) ;
                }
        host = argv[1]  ;               /*      machine souhaitee       */
        PORT_NUMBER = atoi(argv[2]) ;           /*      port du serveur         */
        requete = argv[3] ;
/* ------------------------------------------------------------------- */
/*      traduction du nom de machine                                   */
/* ------------------------------------------------------------------- */
        if ((hp=gethostbyname(host))==NULL)
                {
```

```
                fprintf (stderr, "\nMachine inconnue \n") ;
                exit (1) ;
                }

        bcopy((char *)hp->h_addr, (char *)&sa.sin_addr, hp->h_length) ;
        sa.sin_family = hp->h_addrtype ;

        sa.sin_port = htons(PORT_NUMBER);


/* -------------------------------------------------------------------- */
/*      allouer un socket ouvert, se connecter au serveur remote        */
/* -------------------------------------------------------------------- */

        if ((s=socket(hp->h_addrtype,SOCK_STREAM,0))<0)
                {
                perror ("socket")       ;
                exit (1)                ;

                }
        if (connect(s,&sa,sizeof sa)<0)
                {
                perror ("connect")      ;
                exit (1)                ;
                }


/* -------------------------------------------------------------------- */
/*      emettre la question                                             */
/* -------------------------------------------------------------------- */

        if (write(s, requete, strlen(requete)) != strlen(requete))
                {
                fprintf (stderr,"%s: write error\n",myname);
                exit (1) ;
                }

/* -------------------------------------------------------------------- */
/*      ecriture de ce qui est lu en reponse                            */
/* -------------------------------------------------------------------- */

        while ((len=read(s, buf, BUFSIZ))>0)
                {
                write(1,buf,len);
                }

/* -------------------------------------------------------------------- */
/*      fermeture du socket et terminaison du programme                 */
/* -------------------------------------------------------------------- */


        close (s)       ;
        exit(0)         ;

        }

/* -------------------------------------------------------------------- */
/*              Procedure Serveur.                                      */
/* -------------------------------------------------------------------- */
```

```
/* ------------------------------------------------------------------- */
/* Include's standards                                                 */
/* ------------------------------------------------------------------- */

#include        <stdio.h>

#include        <sys/types.h>
#include        <sys/socket.h>
#include        <netinet/in.h>
#include        <netdb.h>

/* ------------------------------------------------------------------- */
/*      Define's pour simplifier l'ecriture.                           */
/* ------------------------------------------------------------------- */

#define BACKLOG         5
#define MAXHOSTNAME     32

/* ------------------------------------------------------------------- */
/*      Traitement des arguments passes a la procedure 'main'.         */
/* ------------------------------------------------------------------- */

int             main            (int argc, char * argv[])
        {
        int                     s, t                    ;
        int                     i,PORT_NUMBER           ;
        struct sockaddr_in      sa, isa                 ;

        struct hostent *        hp                      ;
        char *                  myname                  ;
        char                    localhost[MAXHOSTNAME+1];

        myname = argv[0]        ;       /* nom du programme             */
        PORT_NUMBER = atoi(argv[1]);
/* ------------------------------------------------------------------- */
/*      identifier la machine du serveur (remote)                      */
/* ------------------------------------------------------------------- */

        gethostname(localhost, MAXHOSTNAME);
        if ((hp=gethostbyname(localhost))==NULL)
                {
                fprintf (stderr, "\nMachine inconnue ! \n")             ;
                exit (1)                                                ;
                }

        sa.sin_port = htons(PORT_NUMBER)                                        ;
        bcopy ((char *)hp->h_addr,(char *)&sa.sin_addr,hp->h_length)    ;
        sa.sin_family = hp->h_addrtype                                  ;

/* ------------------------------------------------------------------- */
/*      allouer un socket                                              */
/* ------------------------------------------------------------------- */

        if ((s=socket(hp->h_addrtype,SOCK_STREAM,0))<0)
/*          -                           ancien  (demi)socket            */
                {
                perror ("socket")                                       ;
```

```
                exit (1)                                              ;
                }

/* -------------------------------------------------------------------- */
/*      associer le socket au port du service                          */
/* -------------------------------------------------------------------- */

        if (bind(s,&sa,sizeof sa)<0)
                {
                perror ("bind")                                       ;
                exit (1)                                              ;
                }

/* -------------------------------------------------------------------- */
/*      guetter                                                         */
/* -------------------------------------------------------------------- */

        listen (s,BACKLOG)                                            ;

/* -------------------------------------------------------------------- */
/*      boucle infinie sur accept pour identifier le socket utilise     */
/* -------------------------------------------------------------------- */

        while(1)
                {
                i = sizeof isa                                        ;
                if ((t=accept(s,&isa,&i))<0)
                        {
                        perror ("accept")                             ;
                        exit (1)                                      ;
                        }
                traiter_requete ( t )                                         ;
                close(t)                                              ;

                }
```

Question 6

Quelles sont les limitations de ce modele de base ? Que faire pour les corriger ?

Reponse

Rendre le serveur multi-client : 1) avec pthread_create ou 2) select + IO non bloquante avec fcntln (meme si ca ne regle pas le pb du sleep...)

Gerer des Timeouts a grand coup de SIGALARM, en gerant un echancier des timers expirant (via une file ou un tas).

TD 2 - Utilisation des pthreads

But

Le but de ce TD est de se familiariser avec l'api des *pthreads* à travers la synchronisation de plusieurs processus coopérants.

Prérequis

Vous devez avoir une bonne pratique de la programmation en langage C.

Question 1

Rappeler le modele sous-jacent a l'implementation des *pthreads*.

Quels en sont les avantages/inconvenients ?

Reponse

[d'apres la page de Xavier Leroy...]

- What is the implementation model for LinuxThreads?

  LinuxThreads follows the so-called "one-to-one" model: each thread is actually a separate process in the kernel. The kernel scheduler takes care of scheduling the threads, just like it schedules regular processes. The threads are created with the Linux clone() system call, which is a generalization of fork() allowing the new process to share the memory space, file descriptors, and signal handlers of the parent.

- Advantages of the "one-to-one" model include:

  * minimal overhead on CPU-intensive multiprocessing (with about one thread per processor); * minimal overhead on I/O operations; * a simple and robust implementation (the kernel scheduler does most of the hard work for us).

  The main disadvantage is more expensive context switches on mutex and condition operations, which must go through the kernel. This is mitigated by the fact that context switches in the Linux kernel are pretty efficient.

- Have you considered other implementation models?

  There are basically two other models. The "many-to-one" model relies on a user-level scheduler that context-switches between the threads entirely in user code; viewed from the kernel, there is only one process running. This model is completely out of the question for me, since it does not take advantage of multiprocessors, and require unholy magic to handle blocking I/O operations properly. There are several user-level thread libraries available for Linux, but I found all of them deficient in functionality, performance, and/or robustness.

The "many-to-many" model combines both kernel-level and user-level scheduling: several kernel-level threads run concurrently, each executing a user-level scheduler that selects between user threads. Most commercial Unix systems (Solaris, Digital Unix, IRIX) implement POSIX threads this way. This model combines the advantages of both the "many-to-one" and the "one-to-one" model, and is attractive because it avoids the worst-case behaviors of both models – especially on kernels where context switches are expensive, such as Digital Unix. Unfortunately, it is pretty complex to implement, and requires kernel support which Linux does not provide. Linus Torvalds and other Linux kernel developers have always been pushing the "one-to-one" model in the name of overall simplicity, and are doing a pretty good job of making kernel-level context switches between threads efficient. LinuxThreads is just following the general direction they set.

Question 2

Detaillez les attributs d'un *pthread*.

Reponse

cf man pthread_attr_init et la structure phtread_attr_t. insiste surtout sur la difference entre les threads joignables et les threads detaches : la liberation des ressources ne ce fait pas au meme moment...

Question 3

Implementer un mini serveur : le programme lit une commande, crée un thread detaché qui enregistre la commande dans un fichier de log pendant qu'il execute la commande.

Reponse

par exemple pour un mini-shell: le plus simple est de bloquer sur une lecture de ligne de commande, cf le client du tp1 et ensuite de faire un pthread_create avec la fonction qui ecrit dans le fichier de log et la ligne de commande comme argument, et faire un pthread_detach pour detache le thread.

sinon on peut reprendre le client du TP1 qui au lieu d'envoyer la requete, utilise des threads pour tracer la commande.

```
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex;
FILE *f;

void *ecrit(void *msg) {
pthread_mutex_lock(&mutex);
fprintf(f, (char *)msg);
pthread_mutex_unlock(&mutex);
return(0);
}

int main(int argc, char **argv) {
char buff[4096];
pthread_t tid;
int n;

if(!(f = fopen("log", "a+"))) {
perror("[thread] fopen");
```

```
pthread_exit((void*)1);
}

if(pthread_mutex_init(&mutex, 0) == -1) {
perror("pthread_mutex_init()");
exit(1);
}

for(n=2; n>1; ) {
printf("> ");
fflush(stdout);
switch(n = read(0, buff, 4096)) {
case -1:
perror("read()");
exit(1);
case 0:
case 1: break;
default:
buff[n] = 0;
if(pthread_create(&tid, 0, ecrit, buff) == -1) {
perror("pthread_create");
exit(1);
}
pthread_detach(tid);
system(buff);
}
}

fclose(f); /* où est le pbm ---> fclose avant fprintf */
/* pourquoi mettre un lock est inutile? */
/* pourquoi ça marche quand même */
}

/* gcc -lpthread -o thread-test thread.c */
```

Attention, il faut utiliser un pthread_mutex pour proteger le fichier de log.

Question 4

Reprendre le programme précedent en considerant que le fichier de log est repliqué. On souhaite avoir un *thread* par replicat du fichier et attendre la fin des mises à jour pour effectuer la commande.

Reponse

Idem precedent, sauf qu'on ne fait pas le pthread_detach, mais on fait un pthread_join sur chaque thread, avant d'effectuer la commande.

TD 3 - Introduction au noyau Unix

But

Le but de ce TD est d'introduire les principaux concepts du code système et son application dans le système Unix. Il présente aussi les principales structures de données utilisées par le noyau, structures que nous utiliserons au cours des TD ultérieurs.

Prérequis

Vous devez avoir une bonne pratique du système Unix, tant du point de vue utilisation des commandes du *shell* que du point de vue programmation en langage C.

Question 1

Qu'est-ce que le noyau

Reponse

C'est le fichier /unix

C'est le système d'exploitation, c'est à dire un ensemble de services offerts aux processus. Dans d'autres systèmes, on a système = noyau + utilitaires. Sous Unix, le noyau est dissocié des utilitaires, ce qui permet l'étude "simple" du noyau.

Caractéristiques

Seul programme exécuté en mode S Seul programme résident Chargé en mémoire au moment du boot

Question 2

Quelles sont les différences entre exécution en mode usager et en mode système ? Pourquoi y-a-t-il ces différences ?

Reponse

Différences

mode U a des instructions en moins : notion d'instruction privilégiée exemples : modifications du SR, du masque d'IT, des registres de la MMU, instructions STOP ou RTE du 68020. Adressage U : restreint aux segments du processus. En mode S, l'adressage n'est pas restreint.

Justification

rôle du système : partager les ressources, partage équitable =¿ les processus ne peuvent pas accéder directement aux ressources. Exemple de ressource : le processeur. Un processus ne peut exécuter l'instruction qui empêche les autres processus d'utiliser la ressource.

Accès aux instructions privilégiées

On peut écrire un programme contenant STOP et le compiler. Si on essaye de l'exécuter, le matériel va le détecter et déclenchera une exception. Le système prendra la main et terminera le processus. Le seul moyen pour utiliser les instructions privilégiées est de les écrire dans le noyau.

Question 3

Expliquez les actions effectuées lors d'un appel système. Expliquez le changement de mode d'exécution.

Reponse

Enchaînement des actions

Mode U : mettre le numéro d'appel système dans un registre TRAP 5 (5 = numéro de la trappe dédiée aux appels système) L'instruction TRAP provoque l'exécution par le matériel de : changement de pile U en pile S sauvegarde d'un contexte minimum (PC, SR) passage en mode S déroutement en fonction du numéro de trappe Le déroutement provoque l'exécution d'un code écrit en assembleur sauvegarde des registres du processeur empilement d'un "faux" contexte d'appel de procédure appel d'une procédure en C Le reste est écrit en C aiguillage en fonction du numéro d'appel système code de l'appel système Au retour de l'appel système, le "faux" contexte de procédure est dépilé restauration des registres RTE L'instruction RTE provoque l'exécution par le matériel de : restauration du contexte minimum changement de pile S en pile U retour en mode U

Utilisation des primitives

Le compilateur ne particularise pas les primitives système. Il n'a aucune connaissance de la sémantique de "read" par exemple. Il compile un appel à "read" comme n'importe quel autre appel de fonction. La librairie standard contient "read.o", module compilé contenant sur le 68020 (code HP-UX simplifié) : _read: movq &0x3,%d0 trap &0x5 bcc.b _read1 jmp _read2 _read1: rts _read2: mov.l movq &-1,%d0 rts Ce module est une routine de support de langage évolué.

Question 4

Expliquez ce qui se passe lors d'une interruption par le matériel. Expliquez le mécanisme et la raison du masquage des interruptions.

Reponse

Les IT sont numérotées de 0 (moins prioritaire) a 7 (plus prioritaire) sur le 68020. Exemples possibles : IT disque (5) et IT tty (3)

IT tty, puis IT disque : l'IT tty commence à être traitée. Lorsque survient l'IT disque (plus prioritaire), il y a déroutement pour traitement. Lorsque le traitement de l'IT disque est terminé, le traitement de l'IT tty est repris. IT disque, puis IT tty : l'IT disque est traitée, puis retourne. L'IT tty était ignorée. Au retour, elle est prise en compte.

Masquage d'IT de niveau n =¿ toutes les IT de 0 à n sont ignorées (elles seront prises en compte plus tard, lorsqu'elles seront démasquées).

Justification du masquage

Il y a deux sortes de masquage : - masquage implicite lors d'une IT. - masquage explicite en mode S par modification du masque dans SR.

Masquage implicite : pour éviter la récursion du traitement des IT.

Masquage explicite : pour assurer l'exclusion mutuelle entre du code normal et une routine d'IT.

Question 5

Comment se passe l'initialisation du système au démarrage de l'ordinateur ?

Reponse

Boot

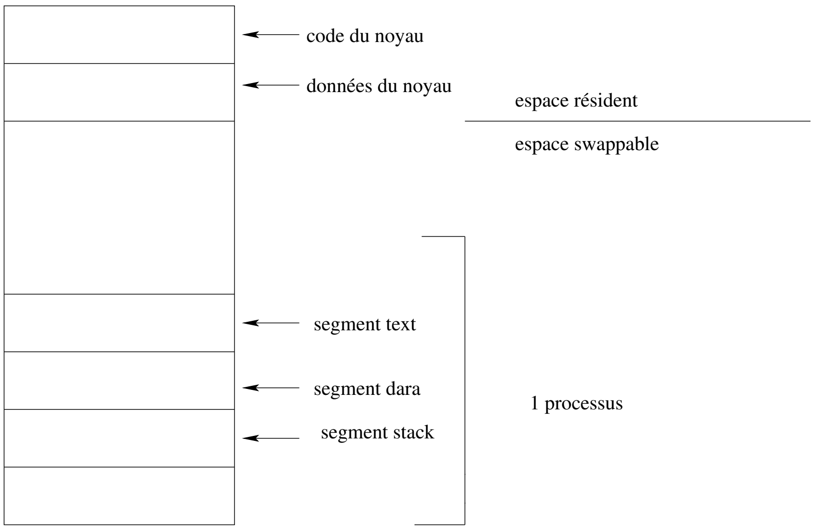
Au boot, le processeur démarre en mode S. - exécution d'un code en Rom, Eprom (indépendant de l'OS) - chargement et exécution du bloc 0 du disque (dépendant de l'OS) - chargement du fichier /unix (nécessite la connaissance des fichiers) - initialisation des structures de données et lancement des processus.

Question 6

Où est le noyau dans la mémoire de l'ordinateur ? Où est le code ? Où sont situées les structures de données qu'utilise le système ?

Où sont situés les processus qui s'exécutent ? Que faut-il pour gérer un processus ? Comment est organisé l'espace mémoire d'un processus ?

Reponse



Remarques

La notion de segment est une notion "logique" sur Unix (et n'a rien à voir avec les segments du 80x86)

Le segment stack est automatiquement agrandi par le noyau en cas de débordement de pile (traitement de l'exception). Le segment data est agrandi explicitement par la primitive système brk ou sbrk (appelée automatiquement par malloc).

Question 7

Justifiez la présence d'une pile d'exécution pour l'usager et d'une pile pour le système.

Reponse

Pile S distincte de la pile U

Preuve par l'absurde : si la pile U est presque pleine, et que le processus fait un appel système, cela conduirait à une exception (débordement de pile) en mode S (ou à l'écrasement des données situées après en mémoire) =¿ crash. Déjà que le noyau doit contrôler les bétises que font les utilisateurs, si en plus il devait se contrôler lui-même !

Une pile S par processus

Le noyau est réentrant. Plusieurs processus peuvent exécuter le noyau en même temps. D'où la nécessité de plusieurs piles, en particulier d'une par processus. Cette pile est inutilisée lorsque le processus est en mode U.

Taille de la pile S

La taille de la pile S peut être déterminée : le noyau n'étant pas récursif, la simple vue du source permet de calculer l'imbrication maximum des fonctions, d'où la place occupée dans la pile. On y ajoute la place occupée par la profondeur maximum des IT 1 + IT 2.. et on obtient une borne supérieure pour la taille de la pile (Mini 6 : 2 Ko)

Question 8

Etudiez la structure `user` et la structure `proc`. Expliquez pourquoi le descripteur de processus est décomposé en deux structures.

Déterminez les différents segments composant un processus. Quelles sont les informations supplémentaires nécessaires pour partager les segments de code entre plusieurs processus ?

Expliquez la présence d'un numéro d'identité (*pid*) pour chaque processus. Comment est-il attribué ?

Où est située la pile système d'un processus ?

Reponse

descripteur de processus = user + proc

"user" n'est en rien lié au concept d'utilisateur. C'est en fait la partie non résidente du descripteur de processus. On a séparé le descripteur en deux pour économiser de la place en mémoire résidente, c'est à dire pour laisser plus de place pour les processus.

Les différents segments sont (en terminologie traditionnelle Unix) : - text : code du processus - data : données, c'est à dire variables globales et tas (malloc) - stack : pile, y compris les variables locales Le segment text est partagé par tous les processus utilisant ce code. Le segment data peut être agrandi par utilisation de la primitive "brk" ou "sbrk" (appelé entre autres par la fonction de librairie malloc) Le segment stack est agrandi automatiquement lors d'un débordement de pile. De plus, il y a le segment appelé "zone u" (voir plus loin).

Le partage du segment text est assuré par une indirection à travers la table text[], référencée par proc[].p_textp. Chaque entrée dans la table text[] représente un segment distinct. Les informations qui s'y trouvent sont : - localisation du segment (adresses mémoire ou disque, taille) - pointeur dans inode[] correspondant au fichier exécutable - nb de processus utilisant le segment - nb de processus en mémoire utilisant le segment La primitive "exec" convertit le nom du fichier en pointeur dans la table inode[], recherche si il y a déjà une entrée dans text[] utilisant cette inode. Si c'est le cas, "exec" ne fait qu'incrémenter le nombre de références.

Le "pid" est utilisé pour référencer un procesus (pour "kill", "ps", etc). Il est atribué séquentiellemet à partir de 0 jusqu'à 30000. Lorsqu'il arrive à la limite, on repart à 0 en sautant les numéros encore en cours d'utilisation. L'indice dans proc[] ne suffit pas, car il est ephémère. Un processus peut se terminer, et un autre peut utiliser l'entrée libérée tout de suite après. Alors qu'il faut très longtemps (au moins un jour à peu près sur un système générant beaucoup de proecssus) pour qu'un "pid" soit réalloué.

La pile système d'un processus est placée dans la zone u, après la structure user.

La zone u est à une adresse fixe (40000 dans les sources étudiés). C'est une adresse virtuelle fixe. L'adresse physique est différente pour chaque zone u (la preuve : il y a plusieurs zones u à un instant donné dans la mémoire), mais le système utilise l'unité de gestion mémoire de telle manière que la zone u soit toujours vue à la même adresse. Ceci permet : - de ne pas changer les adresses stockées dans la pile système - de référencer le processus courant par u.u_procp

La pile système ne peut pas grandir. Sa taille est fixée.

Question 9

Qu'est-ce que le BIOS ? Quand Linux utilise-t-il le BIOS ? Pourquoi ne l'utilise-t-il pas après ? (On ne s'intéressera ici qu'à des architecture Intel IA32.)

(Après avoir répondu, aller sur *http://www.acl.lanl.gov/linuxbios/* pour savoir pourquoi cette question n'aura peut-être bientôt plus aucun sens :-).)

Reponse

Le BIOS (Basic Input Output System) est un ensemble de routines qui permettent à l'ordinateur de fonctionner lors du démarrage. C'est lui qui est le premier programme exécuté. Il a pour tâche de charger un système d'exploitation.

Linux utilise le BIOS seulement au démarrage, pour lire le fichier qui contient le noyau sur le disque dur. Ensuite, il utilisera ses propres driver pour accéder aux périphériques. MS/DOS, par comparaison, utilise intensivement le BIOS tout au long de l'utilisation de la machine.

Linux fonctionne en mode protégé et le BIOS en mode réel, d'où difficultés (s'il fallait passer en mode réel à chaque accès disque par exemple, ce serait lent ; notons que des DOS-extender font ça, qui sont des petits programmes qui permettent de faire tourner des programmes en mode protégé sous DOS ; par exemple DJGPP qui est le portage de GCC sous DOS a un DOS-extender). Certaines mauvaises langues prétendent même que les routines du BIOS sont lentes et pleines de bugs :-).

Question 10

En utilisant les fonctions `sleep` et `wakeup` vues en cours, essayez de programmer une fonction `flock` qui vérouille un fichier en lecture/écriture (donc permet un accès exclusif au fichier). Si le fichier est déjà vérouillé, la fonction devra mettre le processus demandeur en attente. Écrire une fonction `frelease` qui libère le verrou, et réveille les éventuels processus qui attendraient de vérouiller ce fichier.

On utilisera le champ `i_flag` de l'inode correspondant au fichier, en particulier les valeurs `ILOCK` et `IWANT`. On s'endormira sur l'adresse de l'inode, avec le paramètre `PINOD`.

Dans `frelease`, on réveillera tous les processus endormis sur l'inode (`i_flag` aura alors le bit `IWANT` positionné à 1).

À votre avis, cette solution est-elle utilisable telle quelle dans le noyau ?

Que se passe-t-il si un processus ouvre le fichier et y accède sans avoir préalablement appelé `flock` ? Pensez-vous que ce soit un réel problème ?

À votre avis, cette fonction est-elle équitable ?

Reponse

Pour le code : à peu près comme plock / prele sauf qu'on passe la structure `struct file` à la place de l'inode, et donc il faut gérer ça dans le code.

Oui, c'est utilisable tel quel.

S'il fait read ou write avant *flock*, ca va causer de problème si d'autres ont deja ouvert le fichier.Ca casse completement la semantique d'un lock, puisque celui qui fait *flock* croit qu'il est le seul a avoir acces au fichier, ce qui n'est vrai que si tout le monde utilise flock. C'est pas vraiment un problème. C'est au programmeur d'application de faire attention et de mettre de lui-meme des locks là où il en a besoin.

Pour le source, c'est vraiment tres ressemblant a *plock/prele*.

Equitablité:

Sous linux, j'ai fait le test suivant: un programme fait : open lock sleep unlock sur un fichier.

Je le lance 1 fois, il va jusqu'au sleep ensuite j'en lance 2 copies le 1er finit, donc un des 2 autres va etre reveille. Je relance le 1er, donc encore 2 attendent. l'autre quitte, donc un des 2 qui attendent est pris. Et en fait, il y en a qui ne sera jamais pris ! Pour mieux voir, ca ressemble a peu pres a ca :

P1	P2	P3
lock	wait	wait
wait	lock	wait
lock	wait	wait
wait	lock	wait

P3 est toujours en wait.

Il semblerait qu'il n'y ait pas de prise en compte des dates de demande des lock dans linux, et qu'il prenne en fait le dernier qui a fait une demande, comme s'il gerait une pile des demandes de lock...

TD 4 - La synchronisation des processus

But

Le but de ce TD est de comprendre l'implémentation des fonctions de synchronisation `sleep`, `wakeup`, `spl` et `gpl`

Prérequis

Vous devez, au cours des TD précédents, vous être familiarisés avec l'utilisation de ces fonctions de synchronisation.

Vous devez avoir manipulé les primitives de gestion de l'horloge.

Vous devez également savoir ce qu'est un niveau d'interruption pour un processeur.

Question 1

Donnez des exemples d'utilisation des fonctions de synchronisation `sleep` et `wakeup`.

Qu'est-ce qu'un événement. Qu'est-ce que cela représente ?

Reponse

Exemples d'utilisation de sleep et wakeup

voir le module "pipe.c"

Il ne faut jamais faire sleep dans une routine d'IT : le processus qu'on endort a 99 chances sur 100 de ne pas être le bon !

Evénement

C'est essentiellement une convention entre celui qui s'endort et celui qui réveille. Ca pourrait presque être un entier compris entre 1 et n avec un nom symbolique. Le seul problème est que (pour le buffer cache par exemple) on doit pouvoir distinguer entre deux buffers, alors que le nombre de buffers est paramétrable. Il devient alors difficile d'utiliser des simples entiers. Un identifiant unique est l'adresse de la ressource symbolisant la raison de l'attente.

Note : dans pipe, on s'endort soit sur ip, soit sur ip+1, soit sur ip+2. Ce sont des adresses à l'interieur de l'inode, permettant de raffiner l'événement.

En cas de compétition pour la ressource, il faut mettre une boucle autour du sleep. C'est le cas dans "pipe", mais pas dans le driver d'imprimante car on est seul à accéder l'imprimante.

Question 2

En quelles circonstances faut-il utiliser en plus `spl` ? Donnez des exemples.

Quelle est l'opération effectuée par la fonction `spl` ? Quel en est l'effet ?

Reponse

Circonstances

Quand il y a une routine d'IT susceptible d'accéder à la ressource critique. Voir par exemple "iowait" dans le buffer cache ainsi que le driver d'imprimante.

Operation effectuée par spl

Masquage d'IT. Effet : interdiction momentanée de la routine d'IT.

Implementation

On voit dans le code comment cela est implementé : changement des bits 8, 9 et 10 du registre d'état du 68020, c'est à dire changement du masque d'IT.

Question 3

Lorsque vous appelez `sleep`, quand s'effectue la commutation de processus ? Même question lorsque vous appelez `wakeup`.

Dans quelles autres circonstances intervient la commutation de processus ? Quel est le rôle des variables `runin`, `runout` et `runrun` ?

Sleep

Commutation immédiate (appel de swtch). Il y a abandon volontaire du processeur.

Wakeup

Le (ou les) processus est mis dans l'état "prêt à tourner". La commutation arrivera plus tard. Le processus faisant wakeup n'abandonne pas le processeur.

Quand se fait la commutation ?

La commutation se fait : - soit par abandon volontaire du processeur (sleep ou exit) (pause et wait appellent sleep). La commutation se fait immédiatement. - soit par abandon involontaire du processeur (fin de quantum, swap) Dans ce cas, la commutation est faite dans la phase de retour de mode Système en mode Usager (soit en fin d'appel système, soit en fin de traitement d'IT). Important : la routine d'IT horloge ne commute pas les processus : elle ne fait que signaler qu'il faudrait faire une commutation. La commutation se fait au retour en mode Usager (qui ne correspond pas forcément au retour de l'IT si on était déjà dans une IT ou dans un appel système).

Swapper

C'est le processus 0, il est spécial à deux titres : - c'est le seul processus créé artificiellement (de toutes pièces) et pas par fork ou sa version interne. - c'est le seul processus système (plus vrai sur System

V), c'est à dire le seul processus n'ayant pas de code, données, et pile utilisateur. Il passe son temps dans une fonction du noyau (sched). Son rôle : évincer les processus, ou les ramener en mémoire. On l'appelle souvent "scheduler", mais c'est en fait le "swapper".

runrun

Signale, lors d'un retour en mode usager, la nécessité de commuter. L'algorithme du bout d'assembleur qui retourne en mode usager est : si runrun # 0 alors swtch () fin si Exemples : wakeup met runrun à 1, l'IT d'horloge met runrun à 1 toutes les secondes.

runout, runin

Le swapper (processus 0) peut s'endormir pour deux raisons : - aucun processus swappé n'est à ramener en mémoire, c'est à dire qu'aucun processus swappé n'est dans l'état "prêt à tourner". - plus de place en mémoire et aucun processus n'attend en swap depuis plus de 3 secondes (2 sur System V). Il n'y a donc pas de place "facile" à récupérer.

Le swapper signale l'une de ces deux conditions en mettant : - runout = 1 (aucun processus n'est à ramener en mémoire) - runin = 1 (plus de place en mémoire) et en s'endormant (sleep) sur cette ressource.

Lorsque, quelque part, on détecte un remède et que le swapper attend justement sur ce problème, on met la variable correspondante à 0 et on réveille le swapper avec l'adresse de cette variable.

Exemples : - setrun met runout à 0 si le processus à réveiller est swappe. - sleep met runin à 0 car on peut alors libérer de la place facilement.

Question 4

Que signifie l'argument `pri` de `sleep` ? Quand joue-t-il ?

Reponse

Argument de sleep

La priorité est rangée dans la structure proc. Différentes valeurs de priorité (Version 6) :

```
Moins prioritaire
        |
    ... | priorités "normales" des processus
    100 | PUSER
     90 | PSLEP
     40 | PWAIT
      1 | PPIPE
      0 | PZERO
    -50 | PRIBIO
    -90 | PINOD
   -100 | PSWP
        |
        v
 Plus prioritaire
```

Rôle de la priorité

Cette priorité est utilisée lors de la commutation de processus suivant le réveil de ce processus. En cas de réveil de deux processus, le plus prioritaire est mis sur le processeur. Exemple : un processus attendant une fin d'entrée-sortie disque (PRIBIO dans iowait) est plus prioritaire lors du réveil qu'un lecteur attendant qu'un écrivain remplisse le tube (PPIPE dans readp).

Question 5

Ecrivez l'algorithme des fonctions `sleep` et `wakeup`.

Reponse

```
Algorithme de SLEEP
-------------------

    debut section critique
si priorité suffisamment basse pour accepter les signaux
    alors
détecter les signaux
initialiser proc
fin section critique
réveiller le swapper s'il est en manque de place mémoire
commuter
détecter les signaux
    sinon
initialiser proc
fin section critique
réveiller le swapper s'il est en manque de place mémoire
commuter
fin si
    fin section critique

Algorithme de WAKEUP
-------------------

    appeler setrun pour tous les processus endormis sur l'adresse

Algorithme de SETRUN
--------------------

    mettre le processus à "prêt à tourner"
    si priorité plus forte que le processus courant
alors demander une commutation lors du retour en mode usager
    fin si
    si processus swappé et swapper attend qu'un ps puisse revenir en mémoire
alors réveiller le swapper
    fin si

Recalcul des priorités, fin de quantum

    Le recalcul des priorités est effectué toutes les secondes pour
    tous les processus endormis pour des causes "non-système", c'est à dire
    lorsque leur priorité est supérieure à PUSER.
    Le recalcul des priorités est une fonction des paramètres (struct proc)
    - nice (valeur de base specifiée par l'utilisateur)
    - cpu (temps d'occupation du processeur)

    Le paramètre time (temps de résidence en mémoire ou sur disque) est
    utilisé par le swapper

algorithme du SWAPPER
--------------------
debut_boucle:
    début de section critique
pour tous les ps prêsts à tourner, swappes et dont temps sur disque > n
```

```
    faire
p1 := ce processus
n := le temps de résidence de ce ps sur disque
fin pour
/* p1 = le processus swappé qui attend depuis le plus longtemps */
si il n’y a pas de processus à ramener en mémoire
    alors
runout := 1
attendre (runout)
revenir à "debut_boucle"
fin si
    fin section critique

    /* p1 est le processus à ramener en mémoire */
    a := taille du processus
    si le ps a une entrée dans "text" mais si le texte est swappé
alors a += taille du segment texte
    fin si
    allouer de la mémoire
    si allocation réussie
alors aller à "ramener"
    fin si

    /* pas assez de mémoire pour ramener le processus */
    debut section critique
/* on va éjecter les processus endormis */
pour tous les processus
    faire
si ps non système, en mémoire, non verrouillés et en sleep
    alors aller à "ejecter"
fin si
fin pour
/* aucun processus endormi à éjecter */
si le processus n’attend pas sur disque depuis plus de 3 secondes
    alors
runin := 1
attendre (runin)
revenir à "debut_boucle"
fin si
/* on va chercher à éjecter un processus qui est prêt à tourner */
pour tous les ps en mémoire
    faire
si ps non système, en mémoire, non verrouillés, "prêt à tourner"
   ou "endormi", et temps de résidence en mémoire > n
    alors
p1 := ce processus
n := le temps de résidence de ce ps en mémoire
fin si
fin pour
si aucun processus n’est en mémoire depuis plus de 2 secondes
    alors
runin := 1
attendre (runin)
revenir à "debut_boucle"
fin si
/* p1 = le processus qui est en mémoire depuis le plus longtemps */

ejecter:
    fin section critique
```

```
    marquer le processus "swappe" dans proc
    swapper ce processus
    revenir à "debut_boucle"

ramener:
    si le processus a une entrée dans "text"
alors
    si x_ ccount = 0
alors ramener le texte en mémoire
    fin si
    ccount++
    fin si
    ramener le reste du processus en mémoire (segments data et stack)
    libérer la place dans l'espace de swap
    marquer le processus "en mémoire" dans proc
    temps de résidence du processus en mémoire (p_time) := 0
    revenir à "debut_boucle"
```

Fichier slp.c

```
#include "../sys/param.h"
#include "../sys/types.h"
#include "../sys/user.h"
#include "../sys/proc.h"

char    runin, runout, runrun, curpri;

/*
 * Give up the processor till a wakeup occurs
 * on chan, at which time the process
 * enters the scheduling queue at priority pri.
 * The most important effect of pri is that when
 * pri<=PZERO a signal cannot disturb the sleep;
 * if pri>PZERO signals will be processed.
 * Callers of this routine must be prepared for
 * premature return, and check that the reason for
 * sleeping has gone away.
 */

sleep(chan, pri)
caddr_t chan;
{
        register struct proc *rp = u.u_procp;
        register s;

        s = gpl();
        spl(CLINHB);
        if (pri > PZERO) {
                if(issig())
                        goto psig;
                rp->p_wchan = chan;
                rp->p_stat = SSLEEP;
                rp->p_pri = pri;
                spl(NORMAL);
                if(runin != 0) {
                        runin = 0;
                        wakeup(&runin);
                }
                swtch();
                if(issig())
                        goto psig;
        } else {
                rp->p_wchan = chan;
                rp->p_stat = SSLEEP;
                rp->p_pri = pri;
                spl(NORMAL);
                swtch();
        }
        spl(s);
        return;

        /*
         * If priority was low (>PZERO) and
         * there has been a signal,
```

```
         * execute non-local goto to
         * the qsav location.
         */
psig:
        spl(s);
        aretu(u.u_qsav);
}

/*
 * Wake up all processes sleeping on chan.
 */
wakeup(chan)
register caddr_t chan;
{
        register struct proc *p;
        register c, i ;

        c = chan;
        p = &proc[0];
        i = NPROC;
        do {
                if(p->p_wchan == c) {
                        setrun(p);
                }
                p++;
        } while(--i);
}

/*
 * Set the process running;
 * arrange for it to be swapped in if necessary.
 */
setrun(p)
register struct proc *p;
{
        p->p_wchan = 0;
        p->p_stat = SRUN;
        if(p->p_pri < curpri)
                runrun++;
        if(runout != 0 && (p->p_flag&SLOAD) == 0) {
                runout = 0;
                wakeup(&runout);
        }
}
```

Fichier misc.s

```
        .globl  _spl, _gpl
*
* Priority routines
*

_spl    move.l  4(sp),d1        d1 = interrupt level
        asl.w   #8,d1           d1 = d1 << 8
        move.w  sr,d0           d0 = status register
        and.w   #0xf8ff,d0      d0 = d0 without interrupt level
        or.w    d1,d0           d0 = appropriate interrupt level
        move.w  d0,sr
        rts

_gpl    clr.l   d0              d0 = 0L
        move.w  sr,d0           d0 = status register
        and.w   #0x700,d0       d0 = interrupt level (bits 10-8)
        asr.w   #8,d0           d0 = interrupt level (bits 2-0)
        rts
```

TD 5 - Les signaux

But

Le but de ce TD est de comprendre l'implémentation des signaux. L'implémentation étudiée ne prend pas en compte la gestion des signaux suivant la norme POSIX

Prérequis

Vous devez avoir utilisé les primitives système `kill` et `signal`.

Vous devez en outre avoir compris les mécanismes de synchronisation à base de `sleep` et `wakeup`.

Question 1

Donnez des exemples en C d'utilisation des signaux. Récapitulez les actions pouvant être effectuées à la réception d'un signal. Rappelez les mécanismes apportés par la version POSIX des signaux

Reponse

Pour l'exemple, voir TP1

- Préparation du traitement du signal

  int (*signal (signo, fonction)) ()
  int signo ;
  int (*fonction)() ;

  signo correspond au numéro de signal à traiter;
  fonction correspond a:

  – un pointeur sur une fonction à appeler lors de l'arrivée du signal
  – SIG_DFL (valeur numérique = 0) action par défaut
  – SIG_IGN (valeur numérique = 1) ignorer le signal

  l'appel à *signal* renvoie l'adresse de l'ancienne fonction.

- Envoi du signal

  int kill (pid, signo)
  int pid, signo ;

kill envoie le signal spécifié par *signo* au processus *pid.* si $signo = 0$ le signal n'est pas envoyé mais les test de validité sont faits. si $pid = 0$ le signal est envoyé à tous les processus du "même groupe",c'est à dire ayant le même terminal.

Question 2

Récapitulez les différents signaux et les circonstances dans lesquelles ils sont émis.

Reponse

Récapitulation des actions : fonction : fonction appelée lors de la réception d'un signal SIG_DFL : pour la plupart des signaux, terminaison du processus SIG_IGN : ignorer le signal

SIGHUP	1	hangup
SIGINT	2	interrupt ([BREAK])
SIGQUIT	3	quit ([CTRL][slash])
SIGILL	4	illegal instruction (not reset when caught)
SIGTRAP	5	trace trap (not reset when caught)
SIGIOT	6	IOT instruction
SIGEMT	7	EMT instruction
SIGFPE	8	floating point exception
SIGKILL	9	kill (cannot be caught or ignored)
SIGBUS	10	bus error
SIGSEGV	11	segmentation violation
SIGSYS	12	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal from kill
SIGUSR1	16	user defined signal 1
SIGUSR2	17	user defined signal 2
SIGCLD	18	death of a child
SIGPWR	19	power-fail restart

liste et actions par défaut : voir annexe

Attention : les numéros de signaux varient d'une version d'Unix à l'autre. Ces valeurs numériques ne sont données qu'à titre d'exemple, pour bien comprendre que SIGHUP, par exemple, n'est qu'un "#define". En programmation, il faut toujours utiliser SIGHUP et jamais 1.

Question 3

Quelles sont les différences entre les signaux sous Unix et les interruptions matérielles ?

Reponse

Les signaux sont un mécanisme logiciel, implémenté par le système d'exploitation.

Les interruptions matérielles sont un mécanisme matériel, implémenté en interne par le processeur : il faut une ou plusieurs lignes physiques d'interruption des périphériques vers le processeur (i.e. dépot du bit dans p_sig) et un mécanisme de déroutement du processeur via un "vecteur d'interruptions" (i.e.

appel de la fonction dans psig()). Lorsque l'IT arrive, le processeur finit généralement l'instruction en cours. Lorsqu'un signal arrive, Unix attend de remettre le processus destinataire sur le processeur.

Par rapport aux interruptions matérielles, les signaux d'Unix manquent :

- de niveaux d'interruptions (voir TD introduction code système)
- du masquage automatique des interruptions de niveau inférieur ou égal
- d'un dispositif de recherche et identification du circuit en interruption

Par rapport aux interruptions matérielles, les signaux d'Unix ajoutent :

- mécanisme d'action par défaut
- pas de distinction entre interruptions synchrones (TRAP, ILL, FPE...) et interruptions asynchrones (QUIT, ALARM, INT, USR1, USR2...),
- distinction qui est faite en matériel (voir TD introduction code système)
  Cette dualité "implementé par le logiciel"/"implémenté par le matériel" se retrouve dans "buffer cache"/"cache matériel".

Question 4

Quels sont les contrôles effectués par la primitive système `kill` ?

Expliquez comment se déroule l'émission d'un signal.

Expliquez ce qui se passe lorsque le processus récepteur est en attente d'une ressource système ? Quels sont les différents cas à envisager ?

Reponse

```
algorithmes

kill (numéro de processus, numéro de signal)

    si numéro de signal hors limites
alors
    erreur = Invalid argument
    retour
    fin si
    si signal = SIGKILL et processus = 1
alors
    erreur = Invalid argument
    retour
    fin si
    pour toutes les entrées de la table proc
si entrée non utilisée
    alors
passer à l'entrée suivante
fin si
si l'entrée ne correspond pas au processus demandé
    alors
passer à l'entrée suivante
fin si
```

```
si pas de processus spécifié mais ce n'est pas le même terminal
de contrôle ou ce sont les processus 0 ou 1
    alors
passer à l'entrée suivante
fin si
si pas super utilisateur ou pas le même user-id
    alors
passer à l'entrée suivante
fin si
psignal
si processus > 0
    fin de parcours de la table
    fin pour
    si pas d'entrée trouvée
alors
    erreur = No such process

psignal

    si numéro de signal hors limite
alors
    retour
    fin si
    mettre à 1 le bit correspondant au numéro de signal dans p_sig
    si processus endormi sur un événement non système (PINOD...)
alors
    remettre le processus en exécution
    fin si

Si le processus récepteur est en attente d'un ressource système,
il n'est pas réveillé immédiatement, la détection du signal ne se fera
qu'au retour du mode système vers le mode usager.
```

Question 5

Expliquez comment se déroule la réception d'un signal. Comment est réalisé l'appel de la fonction spécifiée par l'utilisateur ?

Que se passe-t-il en cas de réception de plusieurs signaux ?

Reponse

```
algorithm ISSIG
---------------

issig (tester s'il y a un signal en attente)

    tant qu'un signal n'est pas traité
faire
    trouver le plus petit numéro de signal non traité
    si signal = SIGCLD (mort d'un fils)
alors
    si ce signal est ignoré
alors
    pour toutes les entrées de proc
libérer l'entrée de proc des fils à SZOMB
    sinon si ce signal est intercepté
alors
```

```
    retourner le numéro de signal (vrai)
    fin si
    sinon si le signal n'est pas ignoré ou en mode trace
alors
    retourner le numéro de signal (vrai)
    fin si
    mettre à 0 le bit correspondant à ce signal
    fin tant que
    retourner 0 (faux)

Algorithme PSIG
---------------

psig (exécute l'action convenable pour ce signal)

    si mode trace
alors
    stop
    si numéro de signal = 0
alors
    retour
    effacer le bit correspondant au signal
    si ce signal est traité
alors
    si signal ignoré
alors
    retour
    fin si
    erreur = 0
    si signal n'est pas SIGILL ou SIGTRAP
alors
    effacer l'action spécifiée
    fin si
    créer un nouveau contexte sur la pile usager
    modifier le contexte sauvegardé avec l'adresse de l'action (*)
    retour
    fin si
    si signal = SIGQUIT, SIGILL, SIGTRAP,...
alors
    vider la mémoire
    fin si
    sortir


(*) En détail

n = u.u_ar0 [SP] - 4  récupère l'adresse du sommet de pile
et demande 1 mot de plus
n = nouveau sommet de pile
grow (n) étend, si nécessaire la pile pour
inclure cette adresse
suword (n, u.uar0[PC]) copie la valeur sauvegardée de
compteur programme "en dessous
u.u_ar0[SP] = n mémorise le nouveau sommet de pile
u.u_ar0[PC] = p nouvelle adresse = adresse du handler
return

Après ce return, le retour au mode usager provoque, en fait,
le retour au contexte sauvegarde dans u.uar0, c'est à dire
```

```
au handler du signal ; quand celui-ci se termine on
retournera terminer le code qui a été interrompu et dont
l'adresse figure sur la pile usager.
```

Question 6

Quels sont les inconvénients des signaux ?

Proposez un schéma permettant de corriger ces inconvénients.

Reponse

Limites de l'implémentation décrite (l'originale, utilisée jusque dans System V) :

- l'action spécifiée est remise à 0 après avoir été utilisée : si le programme de traitement n'a pas le temps de remettre l'action de traitement, un deuxième signal identique sera "oublié"
- le traitement des signaux reçus pendant des appels système est insuffisant
- il n'est pas possible de masquer un signal pendant les débuts du traitement
- pas de gestion des signaux à l'intérieur des primitives système ("read")

=¿ les signaux ne sont pas un mécanisme "fiable" de communication inter-processus.

Ceci est apporté par les signaux sous BSD4.3 qui n'ont rien à voir avec ce qui est décrit ici. Ces signaux modélisent une machine "virtuelle", et la différence avec les IT matérielles, notamment en ce qui concerne le masquage des signaux, est beaucoup plus réduite. Voir "The Design and Implementation of the 4.3 BSD Unix O.S.", ainsi que l'examen de juin 1992.

Fichier sig.c

```
/*
 * the signal system call.
 */
ssig(signo, fun)
int signo, fun;
{
        register pid;
        register struct proc *p;

        if ((unsigned)--signo >= NSIG-1 || signo == SIGKILL-1) {
                u.u_error = EINVAL;
                return;
        }
        u.u_rval1 = u.u_signal[signo];
        u.u_signal[signo] = fun;
        u.u_procp->p_sig &= ~(1L<<signo);
        if (signo == SIGCLD-1) {
                pid = u.u_procp->p_pid;
                for (p = &proc[1]; p < &proc[NPROC]; p++) {
                        if (pid == p->p_ppid && p->p_stat == SZOMB)
                                psignal(u.u_procp, SIGCLD);
                }
        }
}

/*
 * the kill system call.
 */
kill(pid, signo)
int pid;
int signo;
{
        register struct proc *p, *q;
        int f;

        if (signo < 0 || signo > NSIG) {
                u.u_error = EINVAL;
                return;
        }
        /* Prevent proc 1 (init) from being SIGKILLed */
        if (signo == SIGKILL && pid == 1) {
                u.u_error = EINVAL;
                return;
        }
        f = 0;
        q = u.u_procp;
        for(p = &proc[0]; p < &proc[NPROC]; p++) {
                if (p->p_stat == NULL)
                        continue;
                if (pid > 0 && p->p_pid != pid)
                        continue;
                if (pid==0 && (p->p_ttyp!=q->p_ttyp||p<=&proc[1]))
                        continue;
                if (u.u_uid != 0 && u.u_uid != p->p_uid)
```

```
                        continue;
                f++;
                psignal(p, signo);
                if (pid > 0)
                        break;
        }
        if (f == 0)
                u.u_error = ESRCH;
}

/*
 * Send the specified signal to
 * all processes with 'tp' as its
 * controlling teletype.
 * Called by tty.c for quits and
 * interrupts
 */
signal(tp, sig)
struct tty *tp;
int sig;
{
        register struct proc *p;

        for(p = &proc[0]; p < &proc[NPROC]; p++)
                if(p->p_ttyp == tp)
                        psignal(p, sig);
}

/*
 * Send the specified signal to
 * the specified process.
 */
psignal(p, sig)
register struct proc *p;
register int sig;
{
        sig--;
        if (sig < 0 || sig >= NSIG)
                return;
        p->p_sig |= 1L<<sig;
        if (p->p_stat == SSLEEP && p->p_pri > PZERO)
                setrun(p);
}

/*
 * Returns true if the current
 * process has a signal to process.
 * This is asked at least once
 * each time a process enters the
 * system.
 * A signal does not do anything
 * directly to a process; it sets
 * a flag that asks the process to
 * do something to itself.
 */
issig()
{
        register n;
        register struct proc *p, *q;
```

```
        p = u.u_procp;
        while(p->p_sig) {
                n = fsig(p);
                if (n == SIGCLD) {
                        if (u.u_signal[SIGCLD-1]&01) {
                                for (q = &proc[1]; q < &proc[NPROC]; q++)
                                        if (p->p_pid == q->p_ppid &&
                                                      q->p_stat == SZOMB)
                                                freeproc(q, 0);
                        } else if (u.u_signal[SIGCLD-1])
                                return(n);
                } else if ((u.u_signal[n-1]&1) == 0 || (p->p_flag&STRC))
                        return(n);
                p->p_sig &= ~(1L<<(n-1));
        }
        return(0);
}

/*
 * Perform the action specified by
 * the current signal.
 * The usual sequence is:
 *      if (issig())
 *              psig();
 */
psig()
{
        register int n, p;
        register struct proc *rp;

        rp = u.u_procp;
        if (rp->p_flag&STRC)
                stop();
        n = fsig (rp) ;
        if (n==0)
                return;
        rp->p_sig &= ~(1L<<(n-1));
        if ((p=u.u_signal[n-1]) != 0) {
                if (p & 1)
                        return;
                u.u_error = 0;
                if (n != SIGILL && n != SIGTRAP)
                        u.u_signal[n-1] = 0;
                n = u.u_ar0 [SP] - 4 ;
                grow(n);
                suword(n, u.u_ar0[PC]);
                u.u_ar0[SP] = n;
                u.u_ar0[PC] = p;
                return;
        }
        switch(n) {

        case SIGQUIT:
        case SIGILL:
        case SIGTRAP:
        case SIGIOT:
        case SIGEMT:
        case SIGFPE:
```

```
        case SIGBUS:
        case SIGSEGV:
        case SIGSYS:
                if (core())
                        n += 0200;
        }
        exit(n);
}

/*
 * find the signal in bit-position
 * representation in p_sig.
 */
fsig(p)
struct proc *p;
{
        register i;
        register n;

        n = p->p_sig;
        for(i=1; i<=NSIG; i++) {
                if (n & 1L)
                        return(i);
                n >>= 1;
        }
        return(0);
}
```

TD 6 - La gestion du temps et l'ordonnancement des processus

Le but de ce TD est de comprendre l'implémentation des fonctions de gestion du temps dans un système comme Unix, d'étudier un exemple de routine de traitement d'interruptions ansi que l'ordonnancement des processus.

Prérequis

Vous devez avoir manipulé les primitives de gestion de l'horloge et ordonnancement des processus.

Gestion du temps

Question 1

Rappelez l'utilisation de la fonction `alarm`. Ecrivez un programme composé de deux processus. Le processus père génère un processus fils qui doit exécuter une fonction `traite` toutes les secondes. Au bout d'une minute, le processus père le prévient qu'il doit s'arrêter. Lorsque le processus fils s'arrête effectivement, le processus père se termine.

Question 2

Récapitulez les différentes notions du temps présentes dans le système.

Reponse

Note On appelle "tick" un top d'horloge. Il y a généralement HZ ticks par seconde.

Temps global - horloge du système

C'est le temps en secondes écoulées depuis le 1 janvier 1970 à 00h00 (GMT). La traduction en temps compréhensible par un être humain est effectuée par des routines de la librairie. Encore appele temps "wall clock" par analogie avec les pendules murales.

Temps système et usager d'un processus

C'est le temps CPU utilisé par un processus et ses fils, mesuré en ticks depuis sa création. Le temps d'un processus fils est accumulé lorsque le père enregistre (par "wait") la terminaison.

Alarme

L'appel système "alarm" déclenche l'envoi d'un signal au bout d'un certain temps exprimé en secondes. Ce temps est placé dans la structure proc (champ p_clktim) du processus.

Timeout

Mécanisme mis à la disposition des drivers. Voir Question 6.

Scheduling et swapping

Il s'agit de p_cpu, p_time.

Lbolt

La routine d'IT horloge réveille toutes les secondes tous les processus endormis sur la variable lbolt. Cela ne correspond pas à une ressource spécifique, c'est un service rendu par l'horloge aux drivers.

Profiling

Mesure des temps d'exécution des différentes fonctions des programmes usager. Utilisable par l'option "-p" de "cc", visualisation des résutats par "prof".

Question 3

Que sont les *timeouts* ? A quoi servent-ils ? Expliquez l'utilisation des deux routines `timeout` et `delay`.

Reponse

Timeout

Mécanisme analogue à "alarm" pour le mode système. C'est un service rendu par le noyau (l'IT horloge) aux drivers. Différences avec "alarm" en mode usager : - pas de terminaison implicite du processus, pas de signal envoyé - une fonction est appelée - l'unité de temps est plus petite (plus de precision) : c'est le tick

Utilisation

Un driver veut compléter l'initialisation d'un circuit après 100 ms (c'est à dire 5 ticks). La fonction complete_init() doit être appelée dans 5 ticks. timeout (complete_init, 0, 5) ; Delay est réservée aux usages plus "lourds" nécessitant une commutation de processus. Elle endort le processus pendant n ticks. delay (n)

Question 4

Expliquez la structure de la routine d'interruptions et l'enchaînement des diverses fonctions. Vérifiez que les notions récapitulées à la question 1 sont bien présentes.

Reponse

```
Organisation et enchaînement

    Commence par une partie en assembleur qui va successivement appeler
    les fonctions. L'algorithme en est (toutes les 1/HZ secondes) :
clock ()
```

```
si il y a des timeouts expirés
    alors restart ()
fin si
si une seconde s'est ecoulée
    alors realtime ()
fin si



Algorithme de CLOCK
--------------------

    s'il y a au moins un timeout dans le tableau
alors mettre à jour le temps du premier non écoulé
    fin si
    si le processeur était actif lorsque l'IT s'est déclenchée
alors actualiser cpu
    fin si
    si le processeur était en mode usager lorsque l'IT s'est déclenchée
alors
    actualiser utime (temps CPU usager du processus courant)
    si profiling actif
alors actualiser le profiling
    fin si
sinon
    si le processeur était actif lorsque l'IT s'est déclenchée
        alors actualiser stime (temps CPU système du processus courant)
    fin si
    fin si
    si il y a des timeouts expirés
alors mettre le bit CALOUT dans iaflags
    fin si
    si une seconde au moins s'est écoulée
alors mettre le bit WAKEUP dans iaflags
    fin si


Algorithme de REALTIME
----------------------
    actualiser lbolt
    actaliser l'horloge globale du système
    "tout" était utilisée dans la V6 pour sleep (remplace par alarm, pause)
    runrun := 1, c'est à dire demander une commutation au retour en mode usager
    réveiller tous les processus en attente sur lbolt
    pour toutes les entrées occupées de proc
faire
    si p_time <= 127 alors p_time++  fin si
    si alarme demandée
alors
    p_clktim--
    si p_clktim = 0
alors envoyer le signal SIGCLK
    fin si
    fin si
    actualiser p_cpu
    si priorité "utilisateur"
alors recalculer la priorité en fonction de cpu et nice
    fin si
    fin pour
```

```
    si le swapper attend que de la mémoire puisse se libérer
alors réveiller le swapper
    fin si


Algorithme de RESTART
--------------------

    pour tous les timeouts arrivés à expiration
faire lancer la fonction
    fin pour
    début section critique
tasser le tableau
    fin section critique
```

Question 5

Quelle est la structure de données utilisée pour implémenter ces *timeouts* ?

Reponse

Un tableau contenant (entre autre) les temps par rapport au timeout précédent. Lorsqu'un nouveau timeout est demandé, il est inséré à la bonne place dans le tableau en recalculant le temps du suivant. Pourquoi cette structure de donnée complexe ? Il vaut mieux prendre du temps lors de l'appel à timeout et optimiser la recherche lors de l'IT qui doit être la plus rapide possible, ainsi que l'actualisation de tous les temps puisqu'il n'y en a qu'un seul à actualiser : le premier.

Question 6

Ecrivez l'algorithme de la routine `timeout`. Expliquez brièvement comment fonctionne `delay`.

Reponse

début section critique
  parcourir le tableau des timeouts pour trouver la place où insérer notre nouveau timeout
  décrémenter le temps de l'entrée suivante
  chercher la dernière entrée du tableau
  décaler le tableau (partie située après la place d'insertion)
  actualiser l'entrée nouvellement libérée
fin section critique

Ordonnancement des processus

Question 7

Expliquez l'algorithme de la fontion *setpri*. Quand est-ce que cette fonction est appelée?

Reponse

La fonction *realtime* appèle la fonction *setpri*.

```
Algorithme SETPRI
-----------------
```

```
- Process priority are calculated according to the formula:
priority = min{127, (time used + PUSER + p_nice}

where :
        . time used = accumulated central porcessor time (usually since process was lasted
swapped in), measured in cloclk tickets divided by 16;

        . PUSER =100;

        . p_nice is a parameter used to bias the process priority. It is normaly positive
and hence reduce the process's effective precedence since in UNIX the lower the prioirity,
the higher the precedence.

- The rescheduling flag (runrun) is set if the process, whose priority has just been
recalculated, has less precedence thean the current process;
```

Question 8

Expliquez l'algorithme de la fontion *swtch*. Quand est-ce que cette fonction est appelée?

Reponse

```
"Swtch" is called by "trap", "sleep", "expand", "exit", "stop", and "xalloc".

Algorithme swtch
----------------

This procedure is executed in three phases which in general involve three different
processes: The first and the third are called the "retiring" and the "arising" processes
respectively. Process 0 is always the intermediate process (it may be the "retiring" or
the "arising" process).

- The static variable "p", defines a starting point for searching through the "proc"
arrayto locate the next process to be activated;(obs.: being static,  "p" is preserved between
calls). if "p" is null, its value us set to the beginning of the "proc" array;

-A call to "savu" saves the stack pointer and the environment pointer for the current
process (r5 and r6) in "u.u_rsav";

- "retu" resets r5 and r6 and the kernel adressregister 6 to adress the "scheduler's"
(process 0) data segment;

Obs: The combination of successive calls on "savu" and "retu" constitutes a so-called
"coroutine jump".

- Process 0 then executes:

   . The set of process is searched to find the process whose state is "SRUN" and which
is loaded and for which "p_pri" is a maximum (the smallest value). There are two nested
loops, from which there is no exit till a runnable process can be found. The function idle
() is only disturbed by a clock interrupt; The flag "runrun" is reset to indicate that a
higher priority process than the "retiring" process is ready to run.

  . the priority of the "arising" process is saved in "curpri";
```

. "retu" is called which causes a corotine jmp to the "arising" process, which becomes the current process.

- Phase Three begins:

. "sureg" copies into user mode segment registers, the values apprpriate for the current process. They have been stored before in "u_uisa" and "u_uisd";

. if the new process paused beacuse it was swapped out, set the stack level to the last call to savu (u_save);

. The "return" follows values set by the stack pointer and the environment pointer. These (just before the return) have values equal to those when the most recent "savu(u.u_rsav) was performed.

Fichier clock.c

```
#include "../sys/param.h"
#include "../sys/conf.h"
#include "../sys/proc.h"
#include "../sys/user.h"
#include "../sys/var.h"

/*
 * clock is called straight from
 * real time clock interrupt.
 * Functions:
 *      implement callouts
 *      maintain user/system times
 *      profile user proc's and kernel
 *      lightning bolt wakeup.
 */

clock()
{
        extern int iaflags, idleflag;
        register struct callo *p1;
        register int *pc;

        if (v.ve_callout[0].c_func != 0) {
                p1 = &v.ve_callout[0];
                while (p1->c_time<=0 && p1->c_func!=0)
                        p1++;
                p1->c_time--;
        }
        if (! idleflag)
        {
                if(u.u_procp->p_cpu < 80)
                        u.u_procp->p_cpu++;
        }

        if (user_mode ()) {
                u.u_utime++;
                if (u.u_prof.pr_scale)
                    addupc (u.u_ar0[PC], &u.u_prof, 1);
        } else {
                if (! idleflag)
                        u.u_stime++;
        }

        if ((v.ve_callout[0].c_func!=0 && v.ve_callout[0].c_time<=0) )
                iaflags |= 1<<15-CALOUT;
        if (++lbolt >= HZ)
                iaflags |= 1<<15-WAKEUP;
}

realtime()
{
        register struct proc *pp;

        lbolt -= HZ;
```

```
        time++;

        /* force a switch every second */
        runrun++;
        wakeup(&lbolt);
        for (pp = &v.ve_proc[0]; pp < proc_end; pp++)
                if (pp->p_stat) {
                        if (pp->p_time <= 127)
                                pp->p_time++;

                        if(pp->p_clktim)
                                if(--pp->p_clktim == 0)
                                        psignal(pp, SIGCLK);
                        /*
                         * Update CPU Usage info:
                         */
                        pp->p_cpu >>= 1;
                        if (pp->p_pri >= (PUSER-NZERO))
                                setpri(pp);

                }
        if (runin != 0) {
                runin = 0;
                wakeup((caddr_t)&runin);
        }
}


/*
 * timeout is called to arrange that
 * fun(arg) is called in tim/HZ seconds.
 * An entry is sorted into the v.ve_callout
 * structure. The time in each structure
 * entry is the number of HZ's more
 * than the previous entry.
 * in this way, decrementing the
 * first entry has the effect of
 * updating all entries.
 */
timeout(fun, arg, tim)
int (*fun)();
caddr_t arg;
int tim;
{
        register struct callo *p1, *p2;
        int ps;
        register int t;


        t = tim;
        p1 = &v.ve_callout[0];
        ps = gpl();
        spl(CLINHB);
        while(p1->c_func != 0 && p1->c_time <= t) {
                t -= p1->c_time;
                p1++;
        }
        p1->c_time -= t;
        p2 = p1;
```

```
        while(p2->c_func != 0)
                p2++;
        if(p2 == &v.ve_callout[v.v_callout-2])  {
                spl(ps);
                panic("no callout space");
        }
        while(p2 >= p1) {
                (p2+1)->c_time = p2->c_time;
                (p2+1)->c_func = p2->c_func;
                (p2+1)->c_arg = p2->c_arg;
                p2--;
        }
        p1->c_time = t;
        p1->c_func = fun;
        p1->c_arg = arg;
        spl(ps);
}

restart()
{
        register struct callo *p1;
        register struct callo *p2;
        int ps;

        p1 = &v.ve_callout[0];
        while (p1->c_func!=0 && p1->c_time<=0) {
                (*p1->c_func)(p1->c_arg);
                p1++;
        }
        p2 = &v.ve_callout[0];
        ps = gpl();
        spl(CLINHB);
        while (p2->c_func = p1->c_func) {
                p2->c_time = p1->c_time;
                p2->c_arg = p1->c_arg;
                p1++;
                p2++;
        }
        spl(ps);
}

#define PDELAY  (PZERO-1)
delay(ticks)
{
        extern wakeup();

        if (ticks<=0)
                return;
        timeout(wakeup, (caddr_t)u.u_procp+1, ticks);
        sleep((caddr_t)u.u_procp+1, PDELAY);
}
```

Fichier swtch.c

```
/* set user priority.
 * The rescheduling flag (runrun)
 * is set if the priority is higher
 * than the currently random process
 */

setpri (up)
{register *pp,p;
 pp=up;
 p = (pp->p_cpu & 0377)/16
 p += PUSER + pp->p_nice;
 if (p > 127)
   p =127;
 if (p < curpri)
   runrun++;
 pp->p_pri = p;
}

/*
 * This routine is called to reschedule the CPU.
 * if the calling process is not in RUN state,
 * arrangements for it to restart must have
 * been made elsewhere, usually by calling via sleep.
 */
swtch()
{
        static struct proc *p;
        register int i, n;
        register struct proc *rp;

        if (p == NULL)
                p = &proc[0];
        /*
         * Remember stack of caller
         */
        savu(u.u_rsav);
        /*
         * Switch to scheduler's stack
         */
        retu(proc[0].p_addr);

loop:
        runrun = 0;
        rp = p;
        p = NULL;
        n = 128;
        /*
         * Search for highest-priority runnable process
         */
        i = NPROC;
        do {
                rp++;
                if(rp >= &proc[NPROC])
                        rp = &proc[0];
```

```
                if(rp->p_stat==SRUN && (rp->p_flag&SLOAD)!=0) {
                        if(rp->p_pri < n) {
                            p = rp;
                            n = rp->p_pri;
                        }
                }
        } while(--i);
        /*
         * If no process is runnable, idle.
         */
        if(p == NULL) {
                p = rp;
                idle();
                goto loop;
        }
        rp = p;
        curpri = n;
        /* Switch to stack of the new process and set up
         * his segmentation registers.
         */
        retu(rp->p_addr);
        sureg();

        /* The value returned here has many subtle implications.
         * See the newproc comments.
         */
        return(1);
}
```

TD 7 - Noyau préemptif

But

Le but de ce TD est d'étudier les noyaux préemptifs.

Un noyau préemptif permet d'interrompre le traitement à n'importe quel moment dans le noyau pour élire un processus plus prioritaire.

Question 1

Pourquoi les noyaux préemptifs sont ils plus propices pour gérer le temps-réel ?

Reponse

Ils permettent de traiter plus rapidement les événements tel que les interruptions

Question 2

Dans un noyau non préemptif, supposons qu'une tâche plus prioritaire est réveillée alors que l'on est en train d'exécuter un appel système. A quel moment la tâche plus prioritaire va être effectivement élue ?

Reponse

La tache est élue uniquement lorsqu'on rebascule en mode utilisateur (c'est à dire, soit à la fin de l'appel système, ou si l'appel appelle sleep).

Question 3

On souhaite rendre préemptif un noyau Unix non-préemptif (comme celui des TDs précédents). Pour cela, dès qu'une tâche plus prioritaire est réveillée, elle doit s'exécuter. Modifiez l'algorithme de wakeup (rappelé en annexe) en conséquence.

Reponse

```
wakeup(chan)
register caddr_t chan;
{
        register struct proc *p;
        register c, i ;

        c = chan;
```

```
        p = &proc[0];
        i = NPROC;
        do {
                if(p->p_wchan == c) {
                        setrun(p);
                }
                p++;
        } while(--i);
if (runrun)
  swtch ();
}


/*
 * Set the process running;
 * arrange for it to be swapped in if necessary.
 */
setrun(p)
register struct proc *p;
{
        p->p_wchan = 0;
        p->p_stat = SRUN;
        if(p->p_pri < curpri)
                runrun++;
        if(runout != 0 && (p->p_flag&SLOAD) == 0) {
                runout = 0;
                wakeup(&runout);
        }

 }
```

Question 4

Le noyau étant interruptible, il faut protéger les variables du noyau contre des accès concurrents. On souhaite donc mettre en place un système de verrouillage interne au noyau. On entoure la manipulation de toutes les structures internes de la façon suivante :

lock(&s);

// Manipulation de la structure interne s

unlock(&s);

La primitive lock permet de verrouiller la structure globale s. Une fois verrouillée, si un autre processus entre dans le noyau pour modifier s, il sera bloqué jusqu'à ce que la ressource soit débloquée par le unlock correspondant.

Pour gérer les verrous, on ajoute la table globale *Tlock* ainsi définie:

#define MAXLOCK 100

struct t_lock char*adr; Tlock[MAXLOCK];

Programmez les primitives *lock* et *unlock*.

Reponse

```
extern void sleep(void *, int);
extern void wakeup(void *);
extern void panic(const char *, ...);

#ifndef PUSER
#define PUSER 0
#endif

#define LOCKED 0x1
#define PTLOCK PUSER+2
#define LOCK PUSER+1
#define MAXLOCK 128

/* définition d'une structure de lock */
struct t_lock {
char *addr;
};
struct t_lock TLock[MAXLOCK];
int ftlock;

void lock_table() {
while(ftlock & LOCKED)
sleep(&ftlock, PTLOCK);
ftlock |= LOCKED;
}

void unlock_table() {
ftlock &= ~LOCKED;
wakeup(&ftlock);
}

/* le lock */
int lock(char *addr) {
int n = 0;

/* aquisition de la table de lock */
lock_table();

/* recherche si on est quelque part */
loop:
             n=0;
while((n < MAXLOCK ) && (TLock[n].addr != addr))
n++;

            n=0;
while((n < MAXLOCK ) && (TLock[n].addr != NULL))
n++;


if(n == MAXLOCK)
/* panic("Lock Table Full!!!"); */
{/* endormir le processus sur une variable*/
    unlock_table ();
    sleep (&libre);
    goto loop;
```

```
}

if(TLock[n].addr!=NULL) {
  /* attendre la libération de l'adresse */
unlock_table();
sleep(TLock[n], LOCK);
lock_table();
goto loop;
} else {
        /* entrée libre */
                        TLock[n].addr = addr;
                        unlock_table();
                        return(0);
}
      }

/* le unlock */
int unlock(char *addr) {
unsigned int i;

lock_table();
for(i=0; i<MAXLOCK; i++)
if(TLock[i].addr == addr) {
                     wakeup(TLock[i]);
                     TLock[i].addr = NULL;
     wakeup (libre);
                     unlock_table();
                     return(0);
}
unlock_table();
}
```

Question 5

Quel problème peut se passer si une tâche T1 est bloquée par une tâche T2 moins prioritaire ? En quelques mots, comment les noyaux préemptifs résolvent ce problème ?

Reponse

Il s'agit du problème classique de l'inversion de priorité : la tâche plus prioritaire est tributaire de la tâche moins prioritaire : elle est obligée d'attendre que cette dernière libère la ressource. C'est comme si temporairement la priorité de la tâche T1 etait également à celle de la tâche T2.

Pour résoudre cela, il faut implémenter un mécanisme d'heritage de priorité (c'est le cas dans la gestion des verrous des pthreads) : T2 hérite de la priorité de T1 le temps que T2 libère la ressource. Dès que la ressource est libéré l'ancienne priorité de T2 est restituée.

Fichier preempt.c

```
char    runin, runout, runrun, curpri;

/*
 * Wake up all processes sleeping on chan.
 */
wakeup(chan)
register caddr_t chan;
{
        register struct proc *p;
        register c, i ;

        c = chan;
        p = &proc[0];
        i = NPROC;
        do {
                if(p->p_wchan == c) {
                        setrun(p);
                }
                p++;
        } while(--i);
}

/*
 * Set the process running;
 * arrange for it to be swapped in if necessary.
 */
setrun(p)
register struct proc *p;
{
        p->p_wchan = 0;
        p->p_stat = SRUN;
        if(p->p_pri < curpri)
                runrun++;
        if(runout != 0 && (p->p_flag&SLOAD) == 0) {
                runout = 0;
                wakeup(&runout);
        }
}
```

TD 8 - CRÉATION ET TERMINAISON DE PROCESSUS

Création de processus - fork

L'appel système *fork* appele la fonction *newproc* pour créer un process fils.

Question 1

En analisant le code de *newproc*:

- Quelles sont les ressources (structures) qui seront partagées par le père et le fils? Quels compteurs de référence seront alors modifiés?
- Comment le manque de mémoire est-il géré ?

Reponse

Les strucutures que seront partagées (lignes 33 à 38): fichiers ouverts (entrées dans la table file), segment de code, répertoire courant.

Les compteurs: f_count (fichiers), x_count et x_ccount (segment de code), i_count (inode du répertoire courant).

S'il n'y a pas de mémoire suffisante pour créer le fils, faire "swap out" du processus courant, le fils (lignes 47 à 53). "The new copy will have to be made on disk".

Question 2

Donnez l'algorithme de *fork* et *newproc.*

Reponse

```
"fork ()"

Lines
  5-9  "fork" makes a search fo a slot in the "proc array, which is referenced by "p2".
/* chercher une entrée libre; p2 fait référence à cette entrée */.
```

```
  12-20   for the new process, "fork" returns 0 and initialises various
  accounting parameters.
  /* pour le nouveau processus "fork" renvoi 0 et initialise plusieurs parameters */

  21      for the parent process, "fork" returns the value of the child's
process identification.
/* le processus renvoi pour le père, le pid du fils */




newproc ()

lines

  8-11   "mpid" is an integer which is stepped through the value 0 to 32767.
  Since the cycle  may eventually repeat, a check is made that the
  number is not still in use. If so new value is tried.
  /* mpid est une valeur entre 1 est 32767 (positif). Un test est fait
  pour s'assurer que le numéro du pid n'est pas utilisé */

 13-20   A search is made through the "proc" array for a null "proc" structure
 (indicated by "p_stat" having a null value);
 /* entrée libre -> p_stat = NULL */

 22-23   At this point, the address of the new entry in the "proc" array is
 stored as both "p" and "rpp", and the address of "proc" entry for the
 current process is stored both as "up and "rip";

 /* rpp -> adress de la nouvelle entrée dans proc (fils) */
 /* rip -> current process (père) */

24-32    The attributes of the new process are stored in the new "proc" entry.
 Many of these area are copied from the current process;
 /* les atributs du nouveau processus sont sauvegardés dans la nouvelle
         entrée. La plupart des attribus sont copiés du père */


33-34    The new process inherits the open files of its parent. Increment the
      reference count of each of these;
      /* le nouveau processus hérite les fichiers ouverts du père. Le compteur
      f_count est incrémenté */

35-36 If there is a separate text segment, increment the associated reference
      counts. Notice that "rip", "rpp" are used for temporary reference here;
      /* problème de casting */
  /*  S'il y a un segment de code séparé, incrmenter le compteur associé */

38: increment the reference count for the parent's current directory;
    /* incrémenter le compteur de reférence concernant le répertoire courant */

39: save the current values of the environment and stack pointers in u.u_rsav.

40 - 42 : restore the values of "rip" and "rpp". Temporarily change the value
         of u.u_proc from the value appropriate to the current process to
  the value appropriate to the new process.
```

```
46: try to find an area in main memory in which to create the new
     data segment;

47: if there is no suitable area in main memory, the new copy will have to
     be made on disk.
     /* S'il n'y a pas de mémoire, le processus fils sera swappé. Son segment
     de donnée pointe ver a1 (le segment de donnée) du père. Le swap va se
     charger de le copier sur disque */

48: Mark the current process as "SIDL" to head off temporarily any further
    attemp to swap it out.

50: save the current values of the environment and stack pointers in "u.u_sav".

51: call "xswap" to copy the data segment into the disk swap area.
    Because the second parameter is zero, the main meory area will not be released;
    /* appeler xxswap pour copier directement sur le swap */

52: Marked the new process as "swapped out";

53: return the new process to its normal state.

55: there was room in main memory, so store the adress of the new "proc" entry
     and copy the data segment a block at a time.
     /* il y a de la place en mémoire : copier le segment de données du père vers le
fils */

60: restore the current process's "per process data area" to its previous state.

61: return with a value of zero.



Obs: la table u_ofile est copiée lorsqu'on fait le "copyseg" de la lgine 58.
     S'il n'y a pas de mémoire, le fils est créé sur le swap (a2== NULL; rpp->p_addr
pointe ver le segment de donnée du père (a1) et on fait xswap du fils xswap (rpp,0,0).
```

Terminaison de processus - exit

Question 3

- Qu'est ce qu'un processus zombi ? Quand un processus n'est plus zombi ?

Reponse

Tous processus se terminant passe dans “l'ètat zombi” dans lequel il reste aussi longtemps que son père n'a pas pris connaissance de sa terminaison.

Question 4

- Donnez l'algorithme du code d' *exit*.

Reponse

```
lignes
```

```
6:      reset the tracing flag

7-8:   set all of the values in the array "u.u_signal" to one so that no
future execution of "issig" will ever be followed by execution of
"psig".
/* ignorer les signaux */

9-13:    call "closef" to close all files which the process has open. For the
 most part, "closing" simply involves decrementing a reference count.

14:     reduce the reference count for the current directory.
       /* fermer les fichier ouverts; décrémenter  le compteur de référence du répertoire
courant */

15-17:   a place is needed to store "per process" information until the process
       can look at it. A block in a swap area of the disk is a convenient place;

18:    find a suitable buffer;

19:    copy the lower half of the "u" struct into the buffer area;

22 :    enter the core space occupied by the process into the free list.

23:     set the process state to "zombie"?

26-37 : The remaining code searches the "proc" array to find the parent process
       and to wake it up, to make any children "wards of the state", and,
       if they have "stopped" for tracing, to release them. Finally the
       code includes (for this process) a last call to "swtch".
```

Question 5

Quelle est la signification (l'utilité) du wakeup de la lignes 28 ?

Quelle est l'utilité des lignes 15 à 20 ?

Question 6

Les primitives *exit* et *wait* sont très liées. Donnez le code interne de la primitive *wait()* (sans prendre en compte les statistiques d'utilisation) en vous inspirant du code de exit.

Reponse

```
1:wait ()
2:{
3: register f, *bp;
4: register struct proc *p;

5: f=0;
6: loop:
7: for (p = &proc[0]; p < &proc [NPROC]; p++)
```

```
8:   if (p->p_ppid == u.u_procp -> p_pid) {
9:      f++;

10:    if (p->p_stat == SZOMB) {
11:        u.u_ar0[R0] = p->p_pid;
12:         bp = bread (swapdev, f =p->p_addr);
13:        mfree (swapmap, 1, f);
14:        p->p_stat = NULL;
15:        p->p_id = 0;
16:        p->p_ppid =0;
17:        p->p_sig =0;
18:        p_ttyp =0;
19:        p->p_flag =0;
20        u.u_ar0[R1]= p->u_arg[0];
21:        brelse (bp);
22:        return;
23:    }

24:    if (p->p_stat == SSTOP) {
25:       if (( p->p_flag & SWTED) == 0) {
26:          p->p_flag |=SWTED;
27:  u.u_ar0[R0] = p->p_pid;
28:  u.u_ar0[R1]= (p->p_sig >> 8) | 0177;
29:  return;
30:          }
31:    p->p_flag = & ~(STRC | SWTED);
32:    setrun (p);
33:    }
34:  }
35:  if (f)
36     {
37      sleep (u.u_procp, PWAIT);
38      goto looop;
39   }
40   u.u_error = ECHILD;
```

Lignes

7: search the whole "proc" array looking for child process;

10: if the child is a "zombie:

11: - save the child's process identifying number, to report back to the parent;

12: - read the record back from the disk swap area, and release the swap space;

14-19 - reinitialise the "proc" array entry.
/* libérer entrée du fils */

20 - save the u_arg[0] value also to report back tp the parent;
/* variable passée comme parmètre dans le exit */

21 - release the buffer area;

24 Is the child in a "stopped" -> TRACE

35 if one or more children were found, but none were "zombies" or
"stopped", sleep and then look again.

```
/* j'ai trouvé au moins un fils */
```

Fichier fork.c

```
fork ()
{
    register proc *p1; *p2;

    p1 = u.u_procp;

    for (p2 = &proc[0]; p2 < &proc[NPROC}; p2++)
       if (p2->p_stat == NULL)
            goto found;
     u_u_error = EAGAIN;
     goto out;

    found:
      if (newproc()) {
        u.u_ar0[R0] = 0;
        u.u_cstime [0] = 0;
        u.u_cstime [1] = 0;
        u.u_stime = 0;
        u.u_cutime [0] = 0;
        u.u_cutime [1] = 0;
        u.u_utime  = 0;
        return;
      }

          u.u_aro[R0] = p2->p_pid;
 out:
   u.u_ar0[R7] = +2;
 }


/* Create a new process (the internal version of fork )
   It returns 1 in the new process.
   The essential fact is that the new process is created in such
   a way that it appears to have started executing in the
   same call to newproc as the parent,but in fact the code runs is that of switch.
   The subtle implication of the returned value of switch is that this is
   the value that newproc's caller in the new process sees.*/

newproc ()
    int a1, a2;
    struct proc *p,*up;
    register struct proc *rpp;
    register *rip, n;

    p = NULL;

     /* First, just locate a slot for a process and copy the useful
info from this process int it. The panic 'cannot happen' because fork has
already checked for the existance of a slot. */

    retry:
    mpid++;
    if (mpid < 0) {
```

```
        mpid =0;
       goto retry;
    }
    for (rpp = &proc[0]; rpp < &proc [NPROC]; rpp++) {
      if (rpp ->p_stat == NULL && p== NULL)
        p = rpp;
      if (rpp->p_pid == mpid)
        goto retry;
    }

  if ((rpp = p)== NULL )
      panic ("no procs");

      /* make proc entry for new proc */

      rip = u.u_proc;
      up =rip;

      rpp->p_stat = SRUN;
      rpp->p_flag = SLOAD;
      rpp->p_uid = rip->p_uid;
      rpp->p_ttyp = rip ->p_ttyp;
      rpp->p_nice= rip ->p_nice;
      rpp->p_textp = rip ->p_textp;
      rpp->p_pid = mpid;
      rpp->p_ppid = rip ->p_pid;
      rpp ->p_time = 0;

   /* make duplicate entries where needed */

      for (rip = &u.u_ofile[0]; rip <  &u.u_ofile[NOFILE];)
         if ((rpp = *rip++) !=NULL)
           rpp->f_count ++;

       if ((rpp = up->p_textp) != NULL) {
              rpp ->x_count++;
               rpp ->x_ccount ++;
       }

      u.u_cdir ->i_count++;

       /* Partially simulate the environment of the new process so that
 when it is actaully  created (by copying) it will look right */

    savu (u.u_rsav);
    rpp =p;
    u.u_procp = rpp;
    rip = up;
    n = rip -> p_size;
    a1 = rip ->p_addr;
    rpp -> p_size =n;
    a2 =malloc (coremap,n);

  /* if there is not enough memory for the new process,
     swap ou the current process to generate the copy */

      if (a2 == NULL) {
         rip ->p_stat = SIDL;
         rpp-> p_addr = a1;
```

```
        savu (u.u_sav);
        xswap (rpp,0,0);
        rpp-> p_flag = SSWAP;
        rip-> p_stat = SRUN;
     }
     else {
  /* there is memory, so just copy */

     rpp-> p_addr = a2;
     while (n--)
        copyseg (a1++, a2++);
    }

     u.u_procp = rip;
     return (0);
}
```

Fichier exit.c

```
exit()
{
register int *a, b;
register struct proc *p;

     u.u_procp->p_flags &= ~STRC;
     for (a = &u.u_signal[0]; a < &u.u_signal[NSIG];)
         *a++ = 1;
     for (a = &u.u_ofile[0]; a < &u.u_ofile[NOFILE]; a++)
          if (b = *a) {
              *a = NULL;
              closef(b);
          }

       iput(u.u_cdir);

       b = malloc(swapmap, 1);
       if (b == NULL)
           panic("out of swap");

      p = getblk(swapdev,b);
      bcopy(&u, p->b_addr, 256);
      bwrite(p);

      a = u.u_procp;
      mfree(coremap, a->p-size, a->p_addr);
      a->p_stat = SZOMB;

loop:
       for(p = &proc[0]; p < &proc[NPROC]; p++)
          if (a->p_ppid == p->p_pid) {
              wakeup(p);
 for(p = &proc[0]; p < &proc[NPROC]; p++)
     if (a->p_pid == p->p_ppid) {
                     p->p_ppid = 1;
                     if (p->p_stat == SSTOP)
                        setrun(p);
      }
     swtch();
     /* no return */
 }

   a->p_ppid = 1;
   goto loop;
}
```

TD 9 - Répresantation Interne des Fichiers

But

Le but de ce td est d'étudier comment les fichiers et répertoires sont répresentés et accédés dans le système UNIX.

Question 1

Comment la répresantation interne d'un fichier est faite dans le système Unix? Et celle d'un répertoire?

Reponse

La répresantation interne d'un fichier est donnée par un *i_node*, qui contient une description de la disposition sur le disque d'un fichier et d'autres information telles que le propriétaire, les permission d'accès et le dates d'accès.

Les répertoires sont des fichiers qui informe au système de fichiers sa structure hiérarchique. Un répértoire est un fichier dont les donnés sont une suite d'éléments qui comprennent chacun un numéro de i_node et le nom d'un fichier contenu dans ce répertoire; Chaque répertoire contient aussi les i_nodes du répértoire . (répértoire courant) et .. (répértoire père) [Bach].

Question 2

Pourquoi y-a-t-il une table `inode` en mémoire ?

Reponse

table inode[] : c'est un "cache" des inodes sur le disque. Les inodes utilisés à un instant donné sont chargés en mémoire dans cette table.

Question 3

Etudiez les chaînages entre la table des *i_node* et la table *des fichiers*. Etudiez aussi le chaînage entre ces deux tables et la table *de descripteurs de fichier utilisateur*.

Expliquer le rôle des champs `i_count` et `f_count`.

Deux processus peuvent-ils ouvrir un même fichier ? Un même fichier peut-il avoir deux ou plusieurs noms?

Reponse

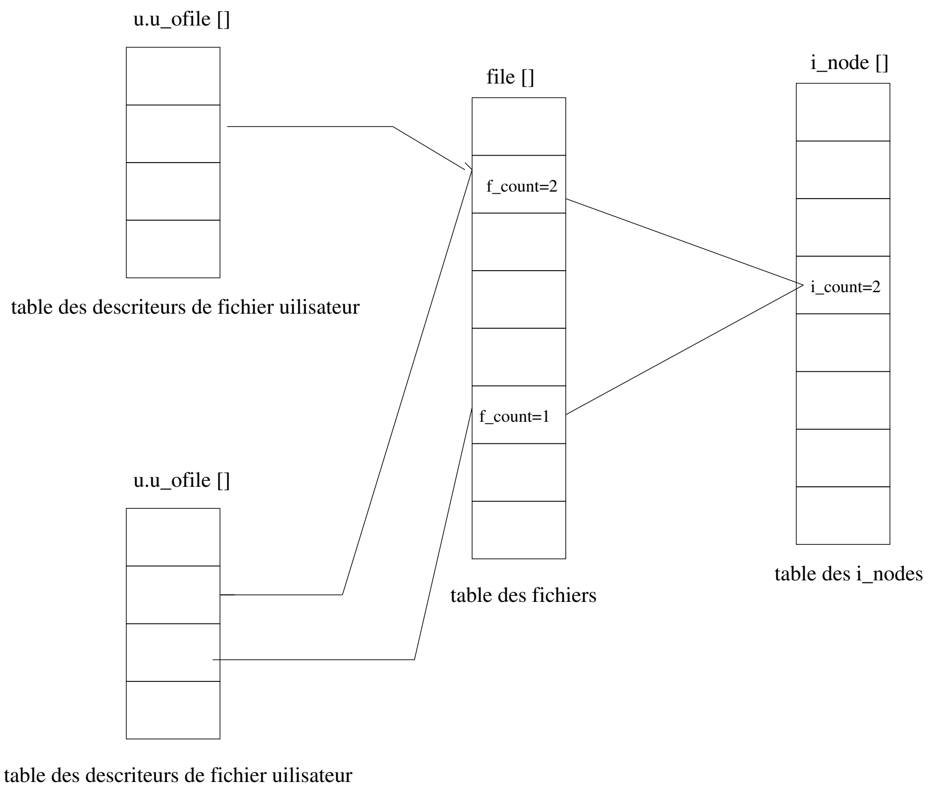
Le noyau contient deux autres strucutures de données, la *table des fichiers* et la *table des descripteurs de fichier utilisateur*. La table des fichiers est une strucutre globale du notau, mais la table de descipteurs est allouée pas processus. Quand un processus ouvre ou crée un fichier, le noyau alloue un elément dans chaque table, correspondant à l'inode du fichier. La table des fichiers préserve le déplacement octet dans le fichier d'ou la prochaine opération de lecture ou écriture débutera, ainsi que les droits d'accès permis au processus qui a permis au processus qui a fait l'a ouvert (open). Le noyau retourne un descripteur de fichier sur les appels système *open* et *create*, qui est un indice dans la table des descripteurs de fichier utilisateur [Bach].

i_count : nb d'utilisation de l'inode (nb de fois où le fichier est ouvert, nb d'entrées dans text [] utilisant cette inode (0 ou 1)).

f_count: nombre de fois qu'une entrée de la la *table des fichiers* et référencé par les processus (entrée dans la *table des descripteurs de fichier utilisateur*).

Deux processus peuvent ouvrir un même fichier. L'ordre dans lequel se font les écritures est alors très important..

Tout fichier a un *i_node*, mais peut avoir plusieurs noms, chaque nom étant en rélation avec le i_node. Chaque nom est appelé un lien.



Divers scenarii :

- ouvertures séparées

  processus A
  fd = open ("f", O_WRONLY)
  write ("aaa", 3)
  write ("c", 1)

  Processus B

fd = open ("f", O_WRONLY)
write ("bb", 2)

Plusieurs résultats possibles dans "f" : bbac ou aaac

- deux entrées séparées dans file[] (deux pointeurs de fichier) i_count = 2, f_count = 1 (première entrée), f_count = 1 (deuxième)

- ouverture héritée par fork

  fd = open ("f", O_WRONLY)
  write ("aaa", 3)
  if (fork () == 0)
  write ("bb", 2)
  else write ("c", 1)

  Plusieurs résultats possibles dans "f" : aaacbb ou aaabbc

  -une seule entrée dans file[] (un seul pointeur de fichier partagé) i_count = 1, f_count = 2

;

Question 4

Quelles sont les informations qu'une strucutre *i_node* contient?

Reponse

Les *i_nodes* disque sont constitués des champs suivants:

- Identificateur du propiétaire du fichier;
- Type du fichier.
- permissions d'accès au fichier.
- dates d'accès au fichier (dernière modification et dernier accès)
- nombre de liens au fichier
- La table des adresses disque des données du fichier.
- La taille du fichier

La copie en mémoire de l'inode contient le champs suivants en plus:

- état de l'inode indiquant si
  - l'inode est vérrouillé ;
  - un processus attend que l'inode soit déverrouillé;
  - la représentation mémoire de l'inode différe de la copie disque au résultat d'un changement de dans l'inode;
  - la représentation mémoire de l'inode différe de la copie disque au résultat d'un changement de données;
  - le fichier est un point de mentage;
- le numéro du périphérique logique du système de fichiers qui contient le fichier; le numéro de l'inod (les inodes sont rangés dans un tableau linéaire sur le disque). Le noyau identifie le numéro d'un inode par sa position dans ce tableau.
- Pointeurs sur d'autre inodes mémoire.

- Un compte référence, indiquant le nombre d'instances du fichier qui sont actives.

Question 5

Décrivez l'algorithme de la fonction *iget*, qui rend la référence à un *i_node* dont le numéro est connu.

Reponse

"iget" est appelée par "main", "unlink", "ialloc" et "namei" avec 2 paramètres qui identifie de manière unique un fichier: un device e le numéro de l'inode du ficher sur le device. "iget" renvoi la réference à l'entrée de l'inode dans la tabel de i'node en mémoire.

```
while (non efféctue)
 Pour tous le entrées de la table de inode en mémoire
Si (inode présent dans le table de inodes en mémoire)
     Si (inode verrouillé)
       {sleep (événement: inode est déverrouillé);
continue;
   }

incrémenter le compte référence inode;
verrouiler le inode;
return (pointeur vers l'inode dans la table de inode);

Noter quelle est la prochaine entrée libre de la table;

Si la table de inode est pleine (pas d'inode libres) afficher message et sortir;
incrémenter le compte référence inode;

incrémenter le compte référence inode;
verrouiler le inode;
indiquer qu'il n'y a pas de référence à autre inode;

Lire le block disque qui contient le inode.  /*bread */

Si (erreur lecture)
{
   libérer le buffer cache;
   libérer le l'entrée de la table de inode;   /*iput*/
     return (erreur);
}

Chercher le inode dans le block; *
liberer le buffer cache;

return (pointeur vers l'inode dans la table de inode);

}

* Assumption that "inode" information begins in block #2 and the convention that valid
"inode" begins at one (not zero).
```

Question 6

Décrivez l'algorithme de la fonction *iput*, utilisée pour libérer un *i_node*.

```
 Si il y a une seule référence à l'inode {
    verrouiller l'inode;
    Si compte de lien de l'inode == 0
      {
        libérer les blocs disque du fichier (algo itrunc);
positionner le type du fichier à 0;
libérer l'inode (algo ifree);
     }
     mettre à jour les information de l'inode disque concernant la date du dernier
accès, si nécessaire;

     }

     décrémenter le compe référence de l'inode;
     libérer le verrou sur l'inode;
```

Question 7

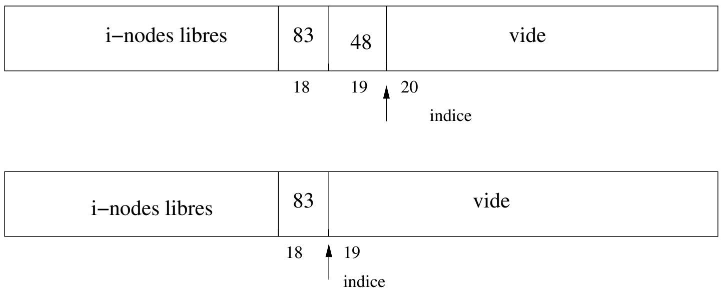
Comment le noyau affecte un *i_node* disque à un fichier nouvellement créé? Comment le noyau gère les inodes libres dans le superbloc?

Reponse

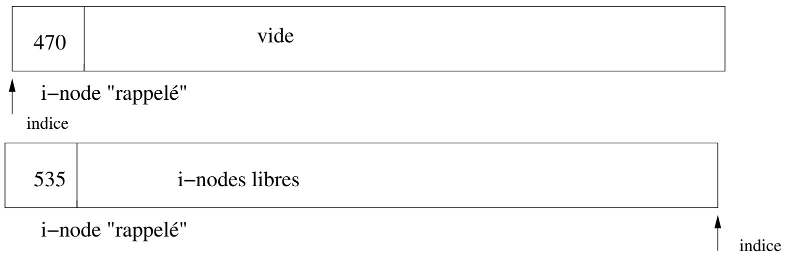
L'algorithme *ialloc* affecte un i-node disque à un fichier nouvellement créé. Le système de fichiers contient une liste linéaire d'inodes. Un inode est libre si son champ *text* vaut 0. Pour améliorer les performances, le superbloc du système contient un tableau qui cache des numéros d'inodes libres du système de fichier. L'algorithme *ifree* libère un inode.

```
ialloc ()   [Bach]
{
     while (non effectué)
     {
  if (superblock verrouillé)
          {
    sleep (événement: superbloc devient libre);
    continue; /*blouce while */
  }
        if (liste d'inodes dans le superbloc est vide)
{
  verrouiler le superblock;
  accéder à l'inode "rappelé" pour rechercher des i-nodes libres;
  rechercher sur le disque des i-nodes libres jusqu'à remplir le
  superbloc, ou jusqu'à ce qu'il n'y ait plus d'inodes libres
  (algorithmes bread et brelse);

  déverrouiler le superbloc;
  wakeup (evénement: le superbloc est devenu libre);
  if (aucun i-node sur le disque n'est libre)
    return (pas d'inode);
  initialiser l'inode "rappelé" pour la prochaine recherche sur le disque;
       }

       /* il y a des i-nodes dans las liste des i-nodes du superbloc */
```

```
        accéder à un numéro d'inode de la liste des inodes du superbloc;

        accéder à l'inode (algorithme iget);
        if (inode n'est pas libre après tout)
       {
          écrire l'inode sur le disque;
 libérer l'inode (algorithme iput)
 continue;
        }

        /*inode est libre */
        initialiser l'inode;
        écrire l'inode sur le disque;
        décrémenter le compte d'inodes libres dans le systèmes de fichiers;
        return (i-node)
}



ifree ()

{
incrémenter le compte d'inodes libres dans le système de ficiers;
if (le superblco est vérrouillé)
  return;
if (liste d'inodes pleine)
  {
           if (numéro de l'inode est plus petit que le numéro de
    l'inode "rappelé" )
      i-node "rappelé" = numéro de l'inode en paramètre;
   }
   else
     ranger le numéro de l'inode dans la liste des inodes;
   return;
}
```



Attribution d'un inode libre depuis le milieu de la liste



Attribution d'un inode libre – Liste du superbloc vide

Question 8

A quoi sert la fonction *namei*? Décrivez son algorithme.

Reponse

L'algorithme *namei* fait la conversion d'un chemin d'accès en un i-node.

L'algorithme:

```
- u.u_cdir defines the ``inode'' of a process's current directory.
``iget'' is called to:
  . wait until such time as the ``inode'' corresponding to ``dp'' is no longer locked;
  . increment the reference count;
  . lock the ``inode'';

-  Any attempt to replace or delete the current working directory is
    bounced immediately;

- The label ``cloop'' marks the beginning of a program loop. Each cycle analyses
  a component of the pathname.

- The end of the pathname has been reached successfully. Return the current value of
dp.

cloop:

- "search" permission for directory is coded in the same way as "execute" permissopn
for other files.

- Copy the name into a more accessible location before attempting to match
   it with a directory entry.

- u.u_count is set to the number of entries in the directory;

- The label ``eloop'' marks the beginning of a program loop. Each cycle of
  the loop handles a single directory entry;

- If the directory has been searched without matching the supplied pathname component,
then there must be an error unless:

  . this is the last component of the pathname (c== \0)
  . the file is to be created (flag == 1)
  . the user program has "write permission" for the directory.

- Record the ``inode'' address for the directory for the new file in u.u_pdir;

- when appropriate, read a new block from the directory file (use of bread);

- copy  the eight words of the directory entry into the arry u.u_dent.

- The eloop cycle is terminated by one of:
. return (NULL);
. goto out;
      . a successful match so that the branch to ``eloop'' is not taken;

- if the name is to be deleted (flag ==2), if the pathname has been completed,
   and if the user has ``write'' access to the directory, then return
   a pointer to the directory ``inode''
```

```
- Save the device identity temporarily and call ``iput'' to unlock ``dp'', to decrement
  the reference count on ``dp'' and to perform any consequent processing;

- Revalidate ``dp'' to point to the ``inode'' for the next level file.
```

Fichier iget.c

```
/* Look up an inode by device, inumber.
A pointer to a locked inode structure is returned
It does not include the mounting of volumes*/

iget (dev,ino)
{
  register struct inode *p;
  register *ip2;
  int *ip1;

 loop:
  ip = NULL;
  for (p = &inode[0]; p <  &inode[NINODE]; p++){
    if (dev == p->i_dev && ino == p->i_number ) {
      if ((p->i_flag & ILOCK) !=0) {
           p->i_flag =| IWANT;
           sleep (p,PINOD);
          goto loop;
      }

    p->i_count++;
    p->i_flag |= ILOCK;
    return (p);
  }
    if (ip== NULL && p->i_count  ==0)
      ip =p;
  }

  if ((p = ip) == NULL) {
      printf ("inode table overflow \n");
      u.u_error = ENFILE;
      return (NULL);
   }

    p->i_dev = dev;
    p->i_number = ino;
    p->i_flag = ILOCK;
    p->i_count ++;
    p->i_lastr =-1;
    ip = bread (dev, ldiv (ino+31,16) );

    /* check I/O errors */
   if (ip->b_flags & B_ERROR) {
      brelse (ip);
      iput(p);
      return (NULL);
    }

   ip1 = ip->b_addr + 32*lrem(ino+31,16) ;
   ip2 = &p->i_mode ;

    while (ip2 < &p->i_addr[8])
      *ip2++ = *ip1++;
    blrese (ip);
```

```
    return (p);
   }


/* Decrement reference count of an inode structure. On the last reference,
   write the inode out and if necessary, truncate and deallocate the file */

iput (p)
struct inode *p
{
  register *rp;
  rp =p;

  if (rp ->i_count == 1) {
    rp ->i_flag |= ILOCK;
    if (rp ->i_nlink == 0) {
      itrunc (rp);
      rp->i_mode =0;
      ifree (rp->i_dev, rp->i_number );

    }
    iupdate (rp, time);
    prele(rp);
    rp->i_flag =0;
    rp->i_number =0;
  }
  rp->i_count --;
  prele(rp);
}

 ifree (dev,ino)
   /* free the specified inode on the specified device */

iupdate (p,tm)
     int *p, int *tm;{

  /* check accessed and update flags on an inode structure.
   If either is on, update the inode with the corresponding dates
set to the argument tm */
}


itrunc (ip)
     int ip;
{
  /* free all disk blocks associated with the specified inode structure.
  The blocks of the file are removed in reverse order.
  */
}
```

Fichier namei.c

```
/*
 * convert a pathname into a pointer to an inode. Note that the inode is locked
 * func = function called to get next char of name
 *        &uchar if name is in user space
 * schar is name isin system space
 *
 * flag = 0, if name is sought
 *  1 if name is to be created
 *  2 if name is to be deleted
 */


namei(func, flag)
     int (* func) ();
{
  register struct inode *dp;
  register c;
  register char *cp;
  int eo, *bp;

  /* if name starts with '/', start from root;
     otherwise start from current dir; */

  dp = u.u_cdir;
  if (( c= (*func ()) == '/')
      dp = rootdir;

      iget (dp->i_dev, dp->i_number );
      while (c == '/' )
         c = (*func) () ;
      if ( c == '\0' && flag !=0) {
u.u_error = ENOENT;
goto out;
      }

  cloop:
      /* Here dp contains pointer to last component matched */

      if (u.u_error)
          goto out;

      if (c == '\0')
          return (dp);

      /* if there is another component, dp must be a directory and must
 have x permission */

      if (( dp ->i_mode & IFMT) != IFDIR ) {
u.u_error = ENOTDIR;
goto out ;
      }

      if (access (dp, IEXEC ))
         goto out;
```

```
     /* gather up name into users' dir buffer */

     cp = &u.u_dbuf[0];

     while ( c!= '/' && c != '\0' && u.u_error == 0 )  {
       if (cp < &u.u_dbuf [DIRSIZ] )
             *cp++ =c;
        c= (*func) () ;

     }

     while (cp < &u.u_dbuf[DIRSIZ])
       *cp++ = '\0';

     while (c == '/')
        c= (*func) ();

     if (u.u_error)
        goto out;

     /* set up to search a directory */
     u.u_offset [1] = 0;
     u.u_offset[0] = 0 ;
     u.u_segflg=1;
     eo = 0;
     u.u_count = ldiv (dp->i_size, DIRSIZ +2 );
     bp= NULL;

eloop:
  /* if at the end of the directory, the search failed.
   Report what is appropriate as per flag */
     if (u.u_count == 0) {
if (bp != NULL)
  brelse (bp);

if (flag == 1 && c == '/0' ) {
  if (access (dp, IWRITE)
      goto out;

      u_u_pdir = dp;
      if (eo)
         u.u_offset [1] = eo - DIRSIZE -2;
      else
             dp ->i_flag =| IUPD;
      return (NULL);
 }
  u.u_error = ENOENT;
  goto out;
   }

/* if offset is on a block-boundary, read the next directory block.
   Release previous if it exsts */

if ((u.u_offset [1] & 0777 == 0){
  if (bp != NULL )
    brelse (bp);
  bp = bread (dp-> i_dev, bmap (dp, ldiv (u.u_offset [1], 512 )));
}
```

```
     /* Note first empty directory slot in eo for possible creat.
String compare the directory entry and the current component.
If they do not match, go back to sloop */

     bcopy (bp->b_addr + (u.u_offset[1] & 0777), &u.u_dent,
    (DIRSIZE +2)/2 );

     u.u_offset[1] =+ DIRSIZ +2;

     u.u_count --;

     f (u.u_dent.u_ino == 0) {
       if (eo == 0)
 eo = u.u_offset [1];
       goto eloop;
     }

     for (cp = &u.u_dbuf[0]; cp < &u.u_dbuf[DIRSIZ]; cp++)
     if (*cp != cp[u.u_dent.u_name - u.u_dbuf] )
      goto eloop;

     /* here a component matched in a directory.
if there is more pathname, go back to eloop, otherwise return */

     if (bp != NULL)
             brelse (bp);
     if (flag == 2 && c == '\0') {
                    if (access (dp,IWRITE))
       goto out;
   return (dp);
     }

    bp = dp -> i_dev;
    iput (dp);
    dp = iget (bp,u.u_dent.u_ino);

    if (dp == NULL)
return (NULL);
    goto cloop;

    out:
    iput (dp);
    return (NULL);
      }
```

TD 10 - Structure des fichiers
Traduction d'adresse / Gestion de l'espace libre sur disque

But

Le but de ce TD est l'étude de la structure physique des fichiers sur disque : traduction d'adresse nécessaire entre l'adresse logique d'un bloc fourni par l'utilisateur et l'adresse physique sur disque et gestion de l'espace libre (allocation et libération de blocs).

Prérequis

Vous devez avoir utilisé et compris le fonctionnement des appels système `read`, `write` et `lseek`.

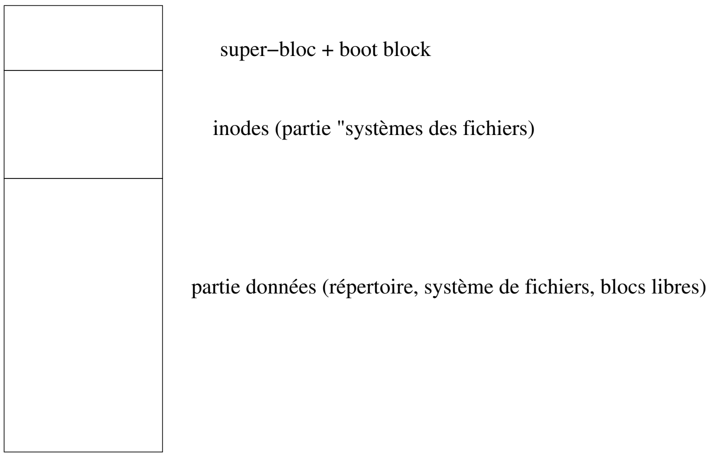
Vous devez avoir assimilé le fonctionnement des entrées/sorties sur Unix et le rôle des *inodes*.

Vous devez connaître la structure générale des disques et des volumes dans le système Unix, ainsi que la méthode d'accès aux données via le *buffer cache*.

Question 1

Rappelez l'implémentation physique des fichiers sous Unix. Rappelez ce que sont les *inodes*.

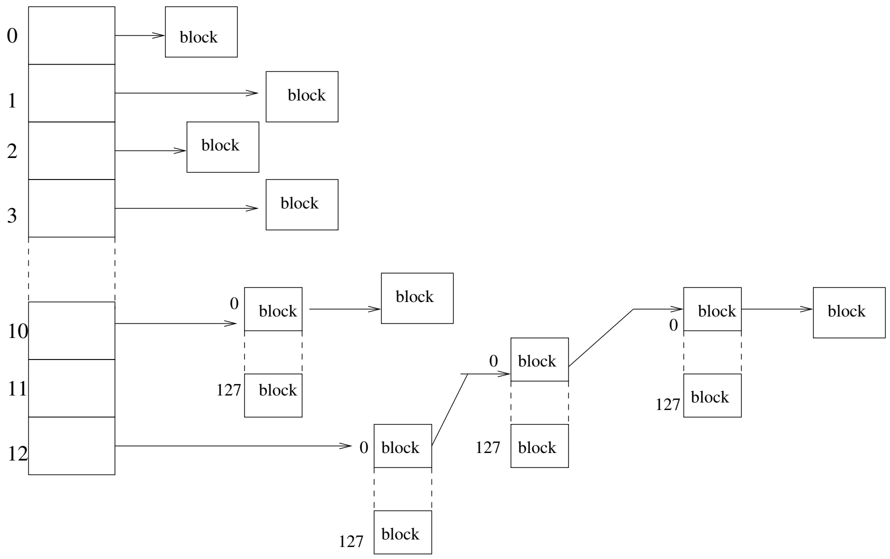
Reponse



En fait, ceci est un "volume" (ou partition, ou système de fichier, ou file-system, ou disque virtuel, ou ... etc).

Un disque réel contient plusieurs volumes avec éventuellement une zone de swap. Un volume contient une arborescence. L'opération consistant à accrocher un volume dans une arborescence existante est le "montage" de volume (utilitaires mount et umount).

Implémentation des fichiers:



Question 2

Quelle est la taille maximum d'un fichier ?

Combien le noyau doit-il faire d'entrées / sorties au minimum et au maximum pour lire dans le fichier ?

Reponse

Taille maximum

512 * (10 + 128 + 128*128 + 128*128*128) octets = 1056837 Ko = 1 Go

Sur des systèmes plus modernes (System V)

1 Ko * (10 + 256 + 256*256 + 256*256*256) = 16843018 Ko = 16 Go mais pb = taille des adresses sur la machine (ex: 23 bits limitent à 4 Go)

Nb d'entrées sorties

Maximum : inode + ind1 + ind2 + ind3 + donnée = 5 blocs Minimum : inode en mémoire, bloc direct déjà en mémoire = 0 blocs

Question 3

Qu'est-ce que le *read-ahead* ? Quels sont les avantages ? Comment est-il détecté ?

Reponse

Read-ahead

Détection par le noyau de lecture séquentielle, sans intervention de l'utilisateur. Avantage : accélération des accès au disque. Désavantage : peut provoquer des accès superflus

Algorithme (dans read) lbn := numéro de bloc logique à lire
bmap => bn := numéro bloc physique ; rablock := no bloc physique suivant
si dernier bloc lu (logique) + 1 = lbn
alors lecture anticipée de rablock
fin si
dernier bloc lu (logique) := lbn

Question 4

Quel est le rôle de la fonction `bmap` ? Expliquer son intérêt. Où se situe-t-elle dans le noyau par rapport à l'appel système ?

Reponse

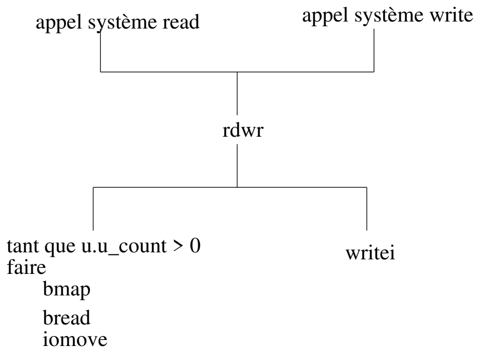
Role:

Traduction d'adresse logique (numéro de bloc dans le fichier) en adresse physique (numéro de bloc sur le disque). Seule fonction connaissant la structure des fichiers, avec la fonction correspondant à unlink (libération de tous les blocs du fichier).

Il existe plusieurs niveau de traduction d'adresse

utilisateur : adresse = (no octet, descripteur de fichier)
read : adresse = (no octet, inode)
adresse = (no bloc ds le fichier, no octet dans le bloc, inode)
bmap : adresse = (no bloc sur le disque, no octet dans le bloc, inode)
buffer : adresse = (no bloc sur le disque, dev)
driver : adresse = (no piste, secteur, face, etc.
)

Situation :



Question 5

Expliquez l'enchaînement des actions lorsqu'un utilisateur utilise les primitives `lseek` et `read`.

```
lseek

    modifier f_offset

read

    u.u_offset := f_offset
    u.u_count := nb de caractères
```

```
    u.u_base := adresse fournie par l'utilisateur
    lbn := numéro de bloc correspondant à u.u_offset

Formation de trous

    #include <fcntl.h>

    main ()
    {
int fd ;

fd = open ("areuh", O_WRONLY | O_CREAT | O_TRUNC, 0666) ;
lseek (fd, 200000000, 0) ;
write (fd, "A", 1) ;
    }

    $ ll areuh
    -rw-r--r--   1 pda      users    200000001 janv 16 22:40 areuh
    $ du areuh (sur BSD)
    48 areuh (blocs d'indirection)

    read renvoie un bloc de 0.
```

Question 6

Ecrivez l'algorithme de bmap.

```
 si bloc trop grand
alors erreur EFBIG (valeur renvoyée = 0)
    fin si

    rablock := 0

    si bloc direct (0 .. NADDR-4)
alors
    nb := numéro bloc physique
    si bloc non nalloue
alors
    si lecture ou pas possible d'allouer un buffer
alors erreur (valeur renvoyée = -1)
    fin si
    nb := numéro bloc alloue par alloc ()
    marquer le buffer "modifie"
    mémoriser nb dans l'inode
    marquer l'inode "modifiée"
    fin si
    si bloc suivant facilement accessible (0 .. NADDR-5)
alors
    rablock := numéro physique bloc suivant
    fin si
    renvoyer numéro bloc physique
    fin si

    déterminer le niveau d'indirection
    (j := 4 - niveau ; nb := 128^(4-j) ; bn := 0 ... 128^(4-j) ; sh := 7x(4-j))

    si trop de niveaux d'indirection
```

```
alors erreur EFBIG (valeur renvoyée = 0)
    fin si

    nb := numéro du premier bloc d'indirection
    si bloc non alloue
alors
    si lecture ou pas possible d'allouer un buffer
alors erreur (valeur renvoyée = -1)
    fin si
    nb := numéro bloc alloue par alloc ()
    marquer le buffer "modifié"
    mémoriser nb dans l'inode
    marquer l'inode "modifiée"
    fin si
    pour tous les blocs d'indirection nécessaires
faire
    lire le bloc d'indirection nb
    si erreur de lecture
alors
    libérer le buffer
    renvoyer 0
    fin si
    bap := adresse du buffer en mémoire (contenant les 128 nos de bloc)
    i := offset dans le bloc
    nb := adresse du i-ème bloc
    si i-ème bloc non alloué
alors
    si lecture ou pas possible d'allouer un buffer
alors
    libérer le buffer dans lequel on a lu le bloc
    erreur (valeur renvoyée = -1)
    fin si
    nb := numéro bloc alloue par alloc ()
    marquer le buffer "modifié"
    mémoriser nb dans le bloc d'indirection
    marquer le bloc d'indirection "modifiée"
sinon libérer le buffer précédent
    fin si
    fin pour
    calculer rablock si possible
```

Question 7

Quel est le rôle des fonctions `alloc` et `free` ?

A quels moments sont-elles appelées ?

Reponse

- Role:

Sans commentaire particulier : alloc alloue un bloc sur le disque et free désalloue un bloc et le remet dans la liste des blocs libres.

- Appel:

alloc : appelée par bmap (si, si, seulement par bmap) free : appelée par itrunc (destruction/troncature de fichier)

Question 8

Expliquez la gestion de l'espace libre.

Donnez un algorithme pour `alloc` et `free`.

Reponse

- Principe:

Le superbloc contient la tête de la liste des blocs libres. Tête de la liste signifie :
- s_tfree : nombre total de blocs libres sur le disque
- s_free : tableau de NICFREE (ex: 100) numéros de blocs libres
- s_nfree : nombre de blocs dans s_free

Note : s_tfree n'est utilisé que pour indiquer le nombre de blocs libres pour "df". Cette information n'est donc pas indispensable au bon fonctionnement du système, mais est plutôt là comme facilité offerte aux utilisateurs.

- Scenarii :

Allocation d'un bloc et superbloc (tête de la liste) plein

```
    avant : s_nfree = 100 ; s_free = |x|x|x|...|x|x|x|
    après : s_nfree =  99 ; s_free = |x|x|x|...|x|x|-|

    Allocation d'un bloc et superbloc (tête de la liste) presque vide

\noindent
    avant : s_nfree =   2 ; s_free =|x|x|-|...|-|-|-|
    après : s_nfree =   1 ; s_free =|x|-|-|...|-|-|-|

    Allocation d'un bloc et superbloc (tête de la liste) vide

\noindent
   avant : s_nfree =   1 ; s_free = |x|-|-|...|-|-|-|
   lecture du bloc suivant de la liste et allocation de ce bloc
   après : s_nfree = 100 ; s_free = |x|x|x|...|x|x|x|
```

```
      alloc

          si superbloc verrouillé
      alors attendre qu'il soit déverrouillé
          fin si
          repeter
      si superbloc (tête de la liste) incohérent
          alors erreur
      fin si
      lire un numéro de bloc dans le superbloc (tête de la liste)
      si numéro de bloc incohérent
```

```
    alors erreur
fin si
    jusqu'à ce que bloc valide
    si superbloc (tête de la liste) épuisé
alors
    verrouiller le superbloc
    lire le bloc suivant de la liste des blocs libres
    recopier ce bloc dans le superbloc (tête de la liste)
    déverrouiller le superbloc
    si bloc incohérent
alors erreur
    fin si
    fin si
    si superbloc (tête de la liste) incohérent
alors erreur
    fin si
    allouer un buffer
    mettre à zéro le contenu du buffer
    décrémenter le nombre de blocs libres
    marquer le superbloc modifié

free

    si superbloc verrouillé
alors attendre qu'il soit déverrouillé
    fin si
    si numéro de bloc incohérent
alors sortir
    fin si
    si liste des blocs libres vide
alors la réinitialiser
    fin si
    si superbloc plein
alors
    verrouiller le superbloc
    localiser le buffer correpondant au bloc libéré
    le remplir avec le contenu du superbloc (tête de la liste)
    remettre à 0 le superbloc (tête de la liste)
    écrire le bloc sur le disque
    déverrouiller le superbloc
    fin si
    ajouter le bloc au superbloc (tête de la liste)
    marquer le superbloc modifié
```

Question 9

Expliquez la provenance du champ `s_flock` du *super-block*.

- Champ s_flock:

Utile seulement si le superbloc est en mémoire (c'est à dire si le volume est monté). Sert à protéger (section critique) une ressource critique (superbloc).

Fichier subr.c

```
#include "../sys/inode.h"
#include "../sys/buf.h"
#include "../sys/types.h"

/*
 * Bmap defines the structure of file system storage
 * by returning the physical block number on a device given the
 * inode and the logical block number in a file.
 * When convenient, it also leaves the physical
 * block number of the next block of the file in rablock
 * for use in read-ahead.
 */

daddr_t
bmap(ip, bn, rwflg)
struct inode *ip;
daddr_t bn;
int rwflg;
{
        register i;
        struct buf *bp, *nbp;
        int j, sh;
        daddr_t nb, *bap;
        dev_t dev;

        if(bn < 0) {
                u.u_error = EFBIG;
                return((daddr_t)0);
        }
        dev = ip->i_dev;
        rablock = 0;
        /*
         * blocks 0..NADDR-4 are direct blocks
         */
        if(bn < NADDR-3) {
                i = bn;
                nb = ip->i_addr[i];
                if(nb == 0) {
                        if(rwflg==B_READ || (bp = alloc(dev))==NULL)
                                return((daddr_t)-1);
                        nb = bp->b_blkno;
                        bdwrite(bp);
                        ip->i_addr[i] = nb;
                        ip->i_flag |= IUPD|ICHG;
                }
                if(i < NADDR-4)
                        rablock = ip->i_addr[i+1];
                return(nb);
        }
        /*
         * addresses NADDR-3, NADDR-2, and NADDR-1
         * have single, double, triple indirect blocks.
         * the first step is to determine
         * how many levels of indirection.
```

```
         */
        sh = 0;
        nb = 1;
        bn -= NADDR-3;
        for(j=3; j>0; j--) {
                sh += NSHIFT;
                nb <<= NSHIFT;
                if(bn < nb)
                        break;
                bn -= nb;
        }
        if(j == 0) {
                u.u_error = EFBIG;
                return((daddr_t)0);
        }
        /*
         * fetch the address from the inode
         */
        nb = ip->i_addr[NADDR-j];
        if(nb == 0) {
                if(rwflg==B_READ || (bp = alloc(dev))==NULL)
                        return((daddr_t)-1);
                nb = bp->b_blkno;
                bdwrite(bp);
                ip->i_addr[NADDR-j] = nb;
                ip->i_flag |= IUPD|ICHG;
        }
        /*
         * fetch through the indirect blocks
         */
        for(; j<=3; j++) {
                bp = bread(dev, nb);
                if(bp->b_flags & B_ERROR) {
                        brelse(bp);
                        return((daddr_t)0);
                }
                bap = bp->b_daddr;
                sh -= NSHIFT;
                i = (bn>>sh) & NMASK;
                nb = bap[i];
                if(nb == 0) {
                        if(rwflg==B_READ || (nbp = alloc(dev))==NULL) {
                                brelse(bp);
                                return((daddr_t)-1);
                        }
                        nb = nbp->b_blkno;
                        bdwrite(nbp);
                        bap[i] = nb;
                        bdwrite(bp);
                } else
                        brelse(bp);
        }
        /*
         * calculate read-ahead.
         */
        if(i < NINDIR-1)
                rablock = bap[i+1];
        return(nb);
}
```

Fichier alloc.c

```
#include "../sys/filsys.h"
#include "../sys/fblk.h"
#include "../sys/buf.h"
#include "../sys/inode.h"

typedef struct fblk *FBLKP;

/*
 * alloc will obtain the next available
 * free disk block from the free list of
 * the specified device.
 * The super block has up to NICFREE remembered
 * free blocks; the last of these is read to
 * obtain NICFREE more . . .
 */
struct buf *
alloc(dev)
dev_t dev;
{
        daddr_t bno;
        register struct filsys *fp;
        register struct buf *bp;

        fp = getfs(dev);
        while(fp->s_flock)
                sleep((caddr_t)&fp->s_flock, PINOD);
        do {
                if(fp->s_nfree <= 0)
                        goto nospace;
                if (fp->s_nfree > NICFREE) {
                        prdev("Bad free count", dev);
                        goto nospace;
                }
                bno = fp->s_free[--fp->s_nfree];
                if(bno == 0)
                        goto nospace;
        } while (badblock(fp, bno, dev));
        if(fp->s_nfree <= 0) {
                fp->s_flock++;
                bp = bread(dev, bno);
                if ((bp->b_flags&B_ERROR) == 0) {
                        fp->s_nfree = ((FBLKP)(bp->b_addr))->df_nfree;
                        bcopy((caddr_t)((FBLKP)(bp->b_addr))->df_free,
                            (caddr_t)fp->s_free, sizeof(fp->s_free));
                }
                brelse(bp);
                fp->s_flock = 0;
                wakeup((caddr_t)&fp->s_flock);
                if (fp->s_nfree <=0)
                        goto nospace;
        }
        if(fp->s_nfree <= 0 || fp->s_nfree > NICFREE) {
                prdev("Bad free count", dev);
                goto nospace;
```

```
	}

	bp = getblk(dev, bno);
	clrbuf(bp);
	if(fp->s_tfree) fp->s_tfree--;
	fp->s_fmod = 1;
	return(bp);

nospace:
	fp->s_nfree = 0;
	fp->s_tfree = 0;
	prdev("no space", dev);
	u.u_error = ENOSPC;
	return(NULL);
}

/*
 * place the specified disk block
 * back on the free list of the
 * specified device.
 */

free(dev, bno)
dev_t dev;
daddr_t bno;
{
	register struct filsys *fp;
	register struct buf *bp;

	fp = getfs(dev);
	while(fp->s_flock)
		sleep((caddr_t)&fp->s_flock, PINOD);
	if (badblock(fp, bno, dev))
		return;
	if(fp->s_nfree <= 0) {
		fp->s_nfree = 1;
		fp->s_free[0] = 0;
	}
	if(fp->s_nfree >= NICFREE) {
		fp->s_flock++;
		bp = getblk(dev, bno);
		((FBLKP)(bp->b_addr))->df_nfree = fp->s_nfree;
		bcopy((caddr_t)fp->s_free,
			(caddr_t)((FBLKP)(bp->b_addr))->df_free,
			sizeof(fp->s_free));
		fp->s_nfree = 0;
		bwrite(bp);
		fp->s_flock = 0;
		wakeup((caddr_t)&fp->s_flock);
	}
	fp->s_free[fp->s_nfree++] = bno;
	fp->s_tfree++;
	fp->s_fmod = 1;
}

/*
 * Check that a block number is in the
 * range between the I list and the size
 * of the device.
```

```
 * This is used mainly to check that a
 * garbage file system has not been mounted.
 *
 * bad block on dev x/y -- not in range
 */

badblock(fp, bn, dev)
register struct filsys *fp;
daddr_t bn;
dev_t dev;
{

        if (bn < fp->s_isize || bn >= fp->s_fsize) {
                prdev("bad block", dev);
                return(1);
        }
        return(0);
}
```

TD 11 - Le buffer cache

But

Le but de ce TD est l'étude du fonctionnement du *buffer cache* et l'optimisation de l'accès aux blocs.

Prérequis

Vous devez avoir compris le rôle de la fonction `bmap` et sa place dans le noyau.

Vous devez en outre connaître les différences entre *drivers* en mode *bloc* et en mode *caractère*.

Question 1

Quelles sont les différences entre les caches d'entrées / sorties sous Unix et les caches des processeurs ?

Reponse

- cache du processeur
  - fonctionnalité : accélérer les accès à la mémoire centrale
  - niveau : matériel. Le cache est géré par le processeur, le programmeur ne le voit pas (sauf pour quelques opérations de gestion / cohérence)
  - situation : le cache du processeur est de la mémoire directement sur la puce ou de la mémoire rapide connectée à un circuit "contrôleur" de cache.
- cache d'entrée/sortie
  - fonctionnalité : accélérer les accès au disque
  - niveau : logiciel. Le cache est géré par le programmeur système, le
  - programmeur en mode usager ne le voit pas (sauf avec sync, crash etc.) situation : le cache d'entrée/sortie est en mémoire centrale.

Question 2

Quel est le rôle du *buffer cache* ? Rappelez l'organisation du *buffer cache*. Comment les descripteurs sont-ils chaînés entre eux ?

Reponse

Rôle du buffer cache : optimiser les accès disques Ne pas confondre buffer cache (mécanisme logiciel) et cache du processeur (mécanisme matériel) De plus, le descripteur "buf" joue le rôle de descripteur d'entrée/sortie

Les données sont pointées par l'union "b_un". Un buffer peut référencer :

- un bloc normal, c'est à dire BSIZE octets de données
- un superbloc
- un bloc de la partie "inodes"
- un bloc indirect

Les données sont stockées dans le tableau buffers [][BSIZE]

Les descripteurs sont chaînés dans deux types de listes :

- la liste des buffers libres (av_...)
- la liste des buffers associées à un périphérique

Ces deux listes correspondent à des méthodes d'accès :

- recherche d'un buffer libre pour lecture d'un bloc sur le disque
- recherche d'un bloc disque dans le buffer cache

En fait, les deuxièmes listes sont plus compliquées, et c'est des listes hashées en fonction du périphérique et du numéro de bloc (voir la fonction de hash "bhash")

0rganisation du buffer cache

Un bloc ne peut pas être dans plusieurs buffers

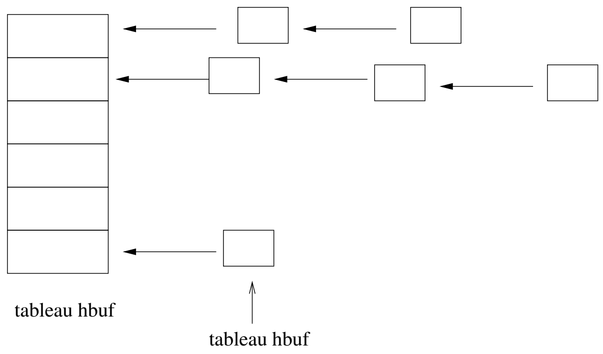
Trois entités :

- structure buf : le descripteur d'entrée sortie
- les données (référencées par l'union b_un de la structure buf)
- tableau hbuf

*Chaînage b_* : Le chaînage "b" sert à retrouver un buffer connaissant le device et le numéro de bloc.

Le tableau hbuf sert à hasher les accès aux buffers. La fonction de hash est : bits de poids faible de (device + bloc) Intérêt de la fonction de hash : répartir la charge quelque soit les différences d'activité sur les disques.

Chaînages av_ : Le chaînage "av" référence les buffers disponibles.
L'ordre dans ce chaînage est important puisque le premier (le plus près de bfreelist) est celui qui est



alloué en premier.
Tout buffer est libre (sauf lorsqu'il est dans un état instable, en cours de manipulation ou d'entrée/sortie par exemple).
Lorsqu'un bloc est utilisé, le buffer correspondant est ôté de la bfreelist, puis remis en dernière position.
gestion LRU

Question 3

Qu'est-ce qu'un *device number* ? Quelle est la fonction qui traite l'entrée / sortie physique ?

Reponse

device number = (majeur, mineur)
majeur = numéro de driver
mineur = numéro de l'unite gerée par le driver

Les versions BSD mettent dans le mineur des informations supplémentaires et dépendantes du driver. Exemple : un bit dans le mineur des unités gérées par le driver de cartouches magnétiques indique s'il faut rembobiner la bande en fin d'utilisation.

Entrée / sortie physique

c'est la fonction "strategy" du driver (mode bloc).

Question 4

Que représentent les états `B_BUSY` et `B_DELWRI` ?

Expliquez pourquoi et quand un buffer ne se trouve plus dans la liste des buffers libres. Expliquez comment il y est remis. Commentez l'intérêt de l'opération.

Reponse

- B_BUSY

  Un bloc est en cours d'entrée/sortie, ou quelqu'un utilise le buffer. Le buffer correspondant n'est donc plus dans la liste des buffers libres. Par contre, il est dans la liste "b", et ne doit pas être consulté. Il est donc marqué "BUSY", et ceux qui veulent y accéder seront mis en attente.

- B_DELWRI

  La primitive système "write", par exemple, marque le buffer "DELWRI", c'est à dire que le bloc a été modifié, mais pas encore recopié sur le disque. Si on veut utiliser le buffer pour un autre bloc, il faudra d'abord recopier le bloc sur le disque.

Retrait de la liste des buffers libres

Un buffer est retiré de la liste des buffers libres lorsqu'il est utilisé (bread/breada). Pourquoi : pour le replacer ensuite en fin de liste des buffers libres, ce qui assure une gestion LRU du cache.

Question 5

Expliquez ce qui se passe lorsqu'un processus demande une lecture et que :

1. le bloc est déjà dans un buffer,
2. le bloc n'est pas dans un buffer, et la *free-list* commence par un buffer non modifié, et
3. le bloc n'est pas dans un buffer, et la *free-list* commence par un buffer modifié.

Expliquez les avantages et inconvénients de l'utilisation du *buffer cache*.

Reponse

- bloc déjà dans un buffer

  Le buffer est trouve en utilisant le chaînage "b"
- bloc n'est pas dans un buffer, la bfreelist commence par un buffer non modifié

  Le buffer n'est pas trouvé en utilisant le chaînage "b". On prend le premier buffer par le chaînage "av", il n'est pas modifié, on l'enlève de la bfreelist, et on fait la lecture.
- bloc n'est pas dans un buffer, la bfreelist commence par un buffer modifié

  Le buffer n'est pas trouvé en utilisant le chaînage "b". On prend le premier buffer par le chaînage "av", il est modifié, on l'enlève de la bfreelist, on recopie le bloc et on recommence la recherche du bloc dans les buffers.

Avantage du cache : réduction des entrées/sorties

Désavantage : sécurité (si crash avant recopie des blocs sur le disque). Pour réduire ce probleme, on utilise un programme qui appelle "sync" toutes les x secondes. On peut aussi déclencher "sync" ou "fsync" manuellement.

Question 6

Quel est le rôle du flag `B_WANTED` ? Expliquez ce qui se passe lorsqu'un processus essaye de lire des données dans un fichier alors qu'une entrée / sortie est déjà en cours sur ce même bloc du même fichier ?

Commentez l'utilisation des deux routines `sleep` et `wakeup`.

Reponse

Multiprogrammation

Lorsqu'un buffer est BUSY (c'est à dire retiré de la bfreelist), il est utilisé par un processus. Il est toujours accessible par le chaînage "b", donc se pose un problème de partage de ressource.

Le flag B_WANTED est mis par un processus qui a détecté que le buffer est occupé, mais qui en a besoin. Il s'endort en mettant B_WANTED de telle sorte que celui qui l'utilisait le réveille à la fin.

sleep/wakeup sleep endort un processus avec deux paramètres :

- événement (identifie la ressource)
- priorité (la signification sera vue au TD 20)

wakeup réveille tous les processus ayant fait un sleep sur l'événement passé en paramètre. à tout sleep doit correspondre au moins un waekup

sleep/wakeup dans getblk/brelse

deux raisons : bp : ce buffer est occupé
bfreelist : il n'y a plus de buffer disponible dans la bfreelist

Question 7

Expliquez comment est effectué le contrôle de la fin de l'entrée / sortie.

Reponse

Fonction iowait

Le driver met le flag B_DONE à 1 et fait un wakeup sur l'événement. Tout le iowait est en section critique, pour pas que le test soit fait, que l'IT arrive, et que le processus attende alors que l'IT arrive.

Le masque d'IT du processeur est mis à BDINHB, il ne reste pas dans sleep.

Question 8

Rédigez l'algorithme de `getblk`.

Expliquez les différences entre les fonctions `bread`, `breada`, `bwrite` et `bdwrite`. Rédigez l'algorithme de ces fonctions.

Reponse

```
getblk

    debut:
démasquer les IT
chercher le bloc dans la liste "b" correspondant au bloc demandé
masquer les IT
si bloc trouvé
    alors
si buffer BUSY (retiré du chaînage "av")
    alors
le marquer "demandé"
attendre
recommencer à debut
    sinon
le retirer du chaînage "av" et le marquer BUSY
démasquer les IT
retour
fin si
    sinon
si bfreelist vide
    alors
marquer la bfreelist "demandée"
attendre
```

```
recommencer à debut
    sinon
si buffer modifié
    alors
lancer la recopie du buffer
recommencer à debut
    sinon
retirer le premier disponible de la bfreeist
le mettre dans la liste "b"
retour
fin si
fin si
fin si

bread

    getblk
    si entrée/sortie déjà faite et finie
alors retour
    fin si
    lancer l'entrée/sortie
    attendre la fin
    retour

breada

    si le bloc n'est pas dans un buffer
alors
    getblk
    si entrée/sortie pas encore faite ou pas encore finie
alors lancer l'entrée/sortie
    fin si
    fin si
    si read-ahead et au moins 1 buf dans la freelist et bloc pas dans un buffer
alors
    getblk
    si entrée/sortie déjà faite et finie
alors remettre le buffer dans la bfreelist
sinon lancer l'entrée/sortie
    fin si
    fin si
    si bloc n'était pas dans un buffer
alors
    bread
    retour
    fin si
    attendre la fin d'entrée/sortie

bwrite

    lancer l'entrée/sortie
    si entrée/sortie synchrone
alors
    attendre la fin d'entrée/sortie
    remettre le buffer dans la bfreelist
    fin si

bdwrite
```

```
    marquer le buffer "modifié" (DELWRI)
    remettre le buffer dans la bfreelist

brelse

    si buffer demandé (B_WANTED)
alors réveiller tous les demandeurs
    fin si
    si bfreelist demandée (B_WANTED)
alors réveiller tous les demandeurs
    fin si
    section critique
insérer le buffer en queue de la bfreelist
actualiser le flags du buffer
    fin section critique

iowait

    section critique
tant que l'entrée/sortie n'est pas terminée
    attendre
fin tant que
    fin section critique
```

Fichier bio.c

```
#include "../sys/buf.h"
#include "../sys/param.h"
#include "../sys/types.h"

/*
 * The following several routines allocate and free
 * buffers with various side effects.  In general the
 * arguments to an allocate routine are a device and
 * a block number, and the value is a pointer to
 * to the buffer header; the buffer is marked "busy"
 * so that no one else can touch it.  If the block was
 * already in core, no I/O need be done; if it is
 * already busy, the process waits until it becomes free.
 * The following routines allocate a buffer:
 *      getblk
 *      bread
 *      breada
 * Eventually the buffer must be released, possibly with the
 * side effect of writing it out, by using one of
 *      bwrite
 *      bdwrite
 *      bawrite
 *      brelse
 */

/*
 * Unlink a buffer from the available list and mark it busy.
 * (internal interface)
 */
notavail(bp)
{
        register s;

        s = gpl();
        spl(BDINHB);
        bp->av_back->av_forw = bp->av_forw;
        bp->av_forw->av_back = bp->av_back;
        bp->b_flags |= B_BUSY;
        bfreelist.b_bcount--;
        spl(s);
}

/*
 * Read in (if necessary) the block and return a buffer pointer.
 */
struct buf *
bread(dev, blkno)
dev_t dev;
daddr_t blkno;
{
        register struct buf *bp;

        bp = getblk(dev, blkno);
        if (bp->b_flags&B_DONE)
```

```
                return(bp);
        bp->b_flags |= B_READ;
        bp->b_bcount = BSIZE;
        (*bdevsw[bmajor(dev)].d_strategy)(bp);
        iowait(bp);
        return(bp);
}

/*
 * Read in the block, like bread, but also start I/O on the
 * read-ahead block (which is not allocated to the caller)
 */
struct buf *
breada(dev, blkno, rablkno)
dev_t dev;
daddr_t blkno, rablkno;
{
        register struct buf *bp, *rabp;

        bp = NULL;
        if (!incore(dev, blkno)) {
                bp = getblk(dev, blkno);
                if ((bp->b_flags&B_DONE) == 0) {
                        bp->b_flags |= B_READ;
                        bp->b_bcount = BSIZE;
                        (*bdevsw[bmajor(dev)].d_strategy)(bp);
                }
        }
        if (rablkno && bfreelist.b_bcount>1 && !incore(dev, rablkno)) {
                rabp = getblk(dev, rablkno);
                if (rabp->b_flags & B_DONE)
                        brelse(rabp);
                else {
                        rabp->b_flags |= B_READ|B_ASYNC;
                        rabp->b_bcount = BSIZE;
                        (*bdevsw[bmajor(dev)].d_strategy)(rabp);
                }
        }
        if (bp == NULL)
                return(bread(dev, blkno));
        iowait(bp);
        return(bp);
}

/*
 * Write the buffer, waiting for completion.
 * Then release the buffer.
 */
bwrite(bp)
register struct buf *bp;
{
        register flag;

        flag = bp->b_flags;
        bp->b_flags &= ~(B_READ | B_DONE | B_ERROR | B_DELWRI);
        bp->b_count = BSIZE ;
        (*bdevsw[bmajor(bp->b_dev)].d_strategy)(bp);
        if ((flag&B_ASYNC) == 0) {
                iowait(bp);
```

```
                brelse(bp);
        } else if ((flag & B_DELWRI) == 0)
                geterror(bp);
}

/*
 * Release the buffer, marking it so that if it is grabbed
 * for another purpose it will be written out before being
 * given up (e.g. when writing a partial block where it is
 * assumed that another write for the same block will soon follow).
 * This can't be done for magtape, since writes must be done
 * in the same order as requested.
 */
bdwrite(bp)
register struct buf *bp;
{
        bp->b_flags |= B_DELWRI | B_DONE;
        bp->b_resid = 0;
        brelse(bp);
}

/*
 * Release the buffer, start I/O on it, but don't wait for completion.
 */
bawrite(bp)
register struct buf *bp;
{
        bp->b_flags |= B_ASYNC;
        bwrite(bp);
}

/*
 * release the buffer, with no I/O implied.
 */
brelse(bp)
register struct buf *bp;
{
        register struct buf **backp;
        register s;

        if (bp->b_flags&B_WANTED)
                wakeup((caddr_t)bp);
        if (bfreelist.b_flags&B_WANTED) {
                bfreelist.b_flags &= ~B_WANTED;
                wakeup((caddr_t)&bfreelist);
        }
        if (bp->b_flags&B_ERROR) {
                bp->b_flags &= ~(B_ERROR|B_DELWRI);
                bp->b_error = 0;
        }


        s = gpl();
        spl(BDINHB);
        backp = &bfreelist.av_back;
        (*backp)->av_forw = bp;
        bp->av_back = *backp;
        *backp = bp;
        bp->av_forw = &bfreelist;
```

```
        bp->b_flags &= ~(B_WANTED|B_BUSY|B_ASYNC);
        bfreelist.b_bcount++;
        spl(s);
}

/*
 * See if the block is associated with some buffer
 * (mainly to avoid getting hung up on a wait in breada)
 */
incore(dev, blkno)
register dev_t dev;
daddr_t blkno;
{
        register struct buf *bp;
        register struct buf *dp;

        dp = bhash(dev, blkno);
        for (bp=dp->b_forw; bp != dp; bp = bp->b_forw)
                if (bp->b_blkno==blkno && bp->b_dev==dev)
                        return(1);
        return(0);
}

/*
 * Assign a buffer for the given block.  If the appropriate
 * block is already associated, return it; otherwise search
 * for the oldest non-busy buffer and reassign it.
 */
struct buf *
getblk(dev, blkno)
register dev_t dev;
daddr_t blkno;
{
        register struct buf *bp;
        register struct buf *dp;

    loop:
        spl(NORMAL);
        dp = bhash(dev, blkno);
        if (dp == NULL)
                panic("devtab");
        for (bp=dp->b_forw; bp != dp; bp = bp->b_forw) {
                if (bp->b_blkno!=blkno || bp->b_dev!=dev)
                        continue;
                spl(BDINHB);
                if (bp->b_flags&B_BUSY) {
                        bp->b_flags |= B_WANTED;
                        sleep((caddr_t)bp, PRIBIO+1);
                        goto loop;
                }
                spl(NORMAL);
                notavail(bp);
                return(bp);
        }
        spl(BDINHB);
        if (bfreelist.av_forw == &bfreelist) {
                bfreelist.b_flags |= B_WANTED;
                sleep((caddr_t)&bfreelist, PRIBIO+1);
                goto loop;
```

```
        }
        spl(NORMAL);
        bp = bfreelist.av_forw;
        notavail(bp);
        if (bp->b_flags & B_DELWRI) {
                bp->b_flags |= B_ASYNC;
                bwrite(bp);
                goto loop;
        }
        bp->b_flags = B_BUSY;
        bp->b_back->b_forw = bp->b_forw;
        bp->b_forw->b_back = bp->b_back;
        bp->b_forw = dp->b_forw;
        bp->b_back = dp;
        dp->b_forw->b_back = bp;
        dp->b_forw = bp;
        bp->b_dev = dev;
        bp->b_blkno = blkno;
        return(bp);
}

/*
 * Wait for I/O completion on the buffer; return errors
 * to the user.
 */
iowait(bp)
register struct buf *bp;
{

        spl(BDINHB);
        while ((bp->b_flags&B_DONE)==0)
                sleep((caddr_t)bp, PRIBIO);
        spl(NORMAL);
        geterror(bp);
}
```

TD 12 - Les pipes

But

Le but de ce TD est de comprendre l'implémentation des pipes, et d'étudier sur un exemple la synchronisation entre processus à l'intérieur du système.

Prérequis

Vous devez avoir utilisé les primitives système `pipe`, `dup` et `fork`.

Vous devez en outre avoir compris les mécanismes de synchronisation à base de `sleep` et `wakeup`.

Question 1

Rappelez l'utilisation des pipes en C. Réalisez le programme suivant composé de deux processsus. Le premier lit des données sur l'entrée standard et les passe par un *pipe* au deuxième qui les affiche sur sa sortie standard.

Question 2

Quelle est la structure de données utilisée par le noyau pour gérer les pipes ?

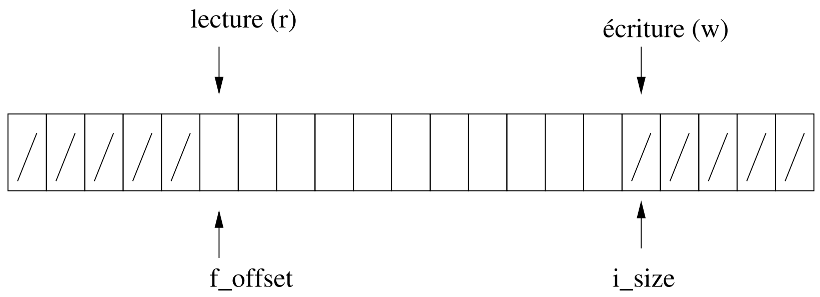
Où sont stockées les données ? Où sont les pointeurs de lecture / écriture ?

Reponse

Structure de données

C'est une gestion FIFO par utilisation d'un fichier.

- Avantages
  - grâce au buffer cache, il est probable que les données seront souvent déjà en mémoire
  - pas de gestion particulière du pipe : réutilisation du code des fichier
  - "swap automatique" des pipes quand il ne sont pas utilisés longtemps
  - pas de mémoire occupée, réservée statiquement : pas de limitation quant au nombre de pipes.
- Inconvénients
  - plus lent qu'une gestion par un tampon direct en mémoire.



Avantage : réutilisation (presque) directe des routines d'accès au fichier

Note:

La taille du fichier est limitée à PIPSIZ (8 blocs directs), pour éviter de ralentir l'accès aux données. La limitation de la taille n'est pas un aspect négatif du pipe.

Question 3

Quels sont les principes de gestion du tampon de stockage de données ?

Combien de processus peuvent accéder simultanément au pipe ?

Comment les divers processus se synchronisent-ils ? Synthétiser par un schéma. Classez les différents types de synchronisation.

Reponse

Principes de gestion

L'écrivain accède au pipe par i_size. S'il arrive au bout du pipe, il s'endort car il doit écrire tout ce qu'a demandé l'utilisateur dans le pipe. Le lecteur accède au pipe par f_offset. Si le pipe est vide, il s'endort, car il doit lire au moins un caractère. Si le pipe n'est pas vide, il lit tout ce qu'il peut. S'il rattrape l'écrivain, il remet les deux compteurs (f_offset et i_size) à 0, et sort, car il n'est pas obligé de lire tout ce qu'a demandé l'utilisateur.

Nombre de processus

Pas de limitation. Il peut y avoir n lecteurs et p écrivains. Il peut y avoir plusieurs processus lors d'un héritage de fichier (par fork), ou duplication explicite de descripteur (par dup).

Synchronisation

De plus, le pipe est une ressource critique qu'il faut protéger. Il y a donc exclusion mutuelle sur l'accès au pipe.

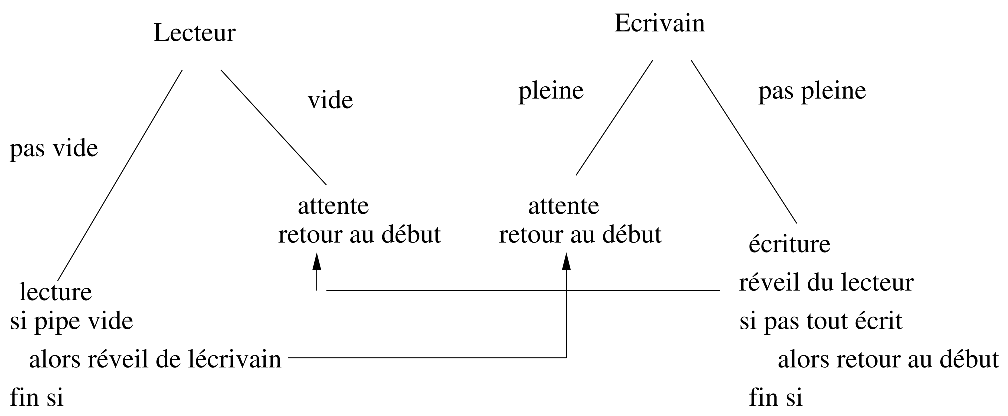
Question 4

Où est situé le module de gestion des pipes dans le noyau ? Quand les fonctions sont-elles appelées ?

Reponse

Situation

Schéma de principe :




```
 -------------------------
| programmes usager       |
|-------------------------|
| couche "appels système" |
|           ex: read      |
|             ------------|
|            | pipe       |
|-------------------------|
| accès aux fichiers      |
|           ex: readi     |
|           ex: bmap      |
|-------------------------|
| gestion du buffer cache |
|           ex: bread     |
|-------------------------|
| drivers bloc            |
|-------------------------|
| matériel                |
 -------------------------
```

Le module de gestion de pipes n'est rien de plus qu'une couche de synchronisation par dessus la couche d'accès aux fichiers. Il y a réutilisation intégrale du code des fichiers, ce qui permet de garder "simple" pipe.c.

Appel des fonctions

La fonction "pipe" est directement le code de la primitive système. Les fonctions "readp" et "writep" sont appelées par les primitives système "read" ou "write" lorsqu'elles accèdent à un pipe (information stockée dans "file"). Normalement, "read" appelle directement "readi" et "write" appelle directement "writei".

Question 5

Ecrivez l'algorithme des fonctions `pipe`, `readp` et `writep`.

Reponse

```
pipe

    allouer un inode dans le disque contenant "/"
    allouer une entrée dans "file" et dans "u_ofile"
    r := indice dans u_ofile alloue par l'appel de falloc
```

```
    allouer une entrée dans "file" et dans "u_ofile"
    recopier les indices "u_ofile" dans le tableau de l'usager
    initialiser les entrées dans "file"
    initialiser l'entrée allouée dans l'inode

readp

    verrouiller le pipe (adresse = ip)
    si le pipe est vide
alors
    déverrouiller l'inode
    si il n'y a que des lecteurs (une seule entrée dans "file")
alors sortir (EOF)
    fin si
    marquer le pipe "demandé en lecture"
    attendre (adresse = ip+2, c'est à dire le 2ème octet de l'inode)
    revenir au debut
    fin si

    u.u_offset = f_offset
    lecture standard dans un fichier (readi)
    si on a rattrapé l'écrivain
alors
    f_offset := i_size := 0
    si le pipe est marqué "demandé en écriture"
alors réveiller l'écrivain (adresse = ip+1)
    fin si
    fin si
    déverrouiller le pipe

writep

    c := nombre de caractères à ecrire
    verrouiller le pipe (adresse = ip)
    si il n'y a que des écrivains (une seule entrée dans "file")
alors
    envoyer un signal (c'est une erreur de la part de l'usager)
    sortir
    fin si
    si le pipe est plein
alors
    déverrouiller l'inode
    marquer le pipe "demandé en ecriture"
    attendre (adresse = ip+1)
    revenir au début
    fin si

    u.u_offset := i_size (écriture à la fin du fichier)
    u.u_count := ce qu'on peut écrire (min (c, limite du pipe))
    écriture standard dans un fichier (writei)
    déverrouiller le pipe
    si le pipe est marqué "demandé en lecture"
alors réveiller le lecteur (adresse = ip+2)
    fin si
    revenir au début
```

Question 6

Quel est le gain procuré par les pipes ? Connaissez-vous d'autres mécanismes de communication entre processus ?

Avantages

FIFO + multi-tâches => d'importants volumes de données peuvent circuler sans prendre beaucoup de place Accès simple en shell => outil très puissant etc.

Autres mécanismes de communication

Version 7 : signaux (très rudimentaires) System III : pipes nommés (résoud le problème de l'ascendance commune) System V : mémoire partagée (très rapide, mais sans mécanisme de synchro) queues de messages (un peu double emploi avec les pipes) sémaphores (très complexes à utiliser, mais grandes possibilités) BSD : sockets (généralisation des pipes, intégration des réseaux)

Les sockets sont très largement repandues, même sur les machines System V grâce à leur possibilité d'accès au réseau.

Fichier pipe.c

```
#include "../sys/param.h"
#include "../sys/conf.h"
#include "../sys/file.h"
#include "../sys/inode.h"
#include "../sys/user.h"

/*
 * Max allowable buffering per pipe.
 * This is also the max size of the
 * file created to implement the pipe.
 * If this size is bigger than 4096,
 * pipes will be implemented in LARG
 * files, which is probably not good.
 */
#define PIPSIZ  4096

/*
 * The sys-pipe entry.
 * Allocate an inode on the root device.
 * Allocate 2 file structures.
 * Put it all together with flags.
 */
pipe(fildes)
int *fildes;    /* 2 word array to return rw file descriptors in */
{
        extern struct file *falloc();
        extern struct inode *ialloc();
        extern iput();
        extern suword();

        register struct inode *ip;
        register struct file *rfp;  /* Read File Pointer */
        register struct file *wfp;  /* Write File Pointer */
        int r;                /* saved read file index/descriptor */

        ip = ialloc(rootdev);
        if(ip == NULL)
                return;
        rfp = falloc();
        if(rfp == NULL) {
                iput(ip);
                return;
        }
        r = u.u_filex;        /* Save Read File index/descriptor */
        wfp = falloc();
        if(wfp == NULL) {
                rfp->f_count = 0;
                u.u_ofile[r] = NULL;
                iput(ip);
                return;
        }
        suword(fildes++, r);      /* return Read file descriptor */
        suword(fildes, u.u_filex);/* return Write file descriptor */
        wfp->f_flag = FWRITE|FPIPE;
```

```
        wfp->f_inode = ip;
        rfp->f_flag = FREAD|FPIPE;
        rfp->f_inode = ip;
        ip->i_count = 2;
        ip->i_flag = IACC|IUPD|ICHG;
        ip->i_mode = IFREG;
}

/*
 * Read call directed to a pipe.
 */
readp(fp)
register struct file *fp;
{
        extern plock();
        extern prele();
        extern readi();
        extern sleep();
        extern wakeup();

        register struct inode *ip;


        ip = fp->f_inode;

loop:
        /*
         * Very conservative locking.
         */

        plock(ip);

        /*
         * If nothing in the pipe, wait.
         */
        if (ip->i_size==0) {

                /*
                 * If there are not both reader and
                 * writer active, return without
                 * satisfying read.
                 */

                prele(ip);
                if(ip->i_count < 2)
                        return;
                ip->i_mode |= IREAD;
                sleep((caddr_t)ip+2, PPIPE);
                goto loop;
        }

        /*
         * Read and return
         */

        u.u_offset = fp->f_offset;
        readi(ip);
        fp->f_offset = u.u_offset;
        /*
```

```
	 * If reader has caught up with writer, reset
	 * offset and size to 0.
	 */

	if(fp->f_offset == ip->i_size) {
		fp->f_offset = 0;
		ip->i_size = 0;
		if(ip->i_mode&IWRITE) {
			ip->i_mode &= ~IWRITE;
			wakeup((caddr_t)ip+1);
		}
	}
	prele(ip);
}

/*
 * Write call directed to a pipe.
 */
writep(fp)
register struct file *fp;
{
	extern int min();
	extern plock();
	extern prele();
	extern psignal();
	extern sleep();
	extern wakeup();
	extern writei();

	register struct inode *ip;
	register int c;


	ip = fp->f_inode;
	c = u.u_count;

loop:

	/*
	 * If all done, return.
	 */

	plock(ip);
	if(c == 0) {
		prele(ip);
		u.u_count = 0;
		return;
	}

	/*
	 * If there are not both read and
	 * write sides of the pipe active,
	 * return error and signal too.
	 */

	if(ip->i_count < 2) {
		prele(ip);
		u.u_error = EPIPE;
		psignal(u.u_procp, SIGPIPE);
```

```
                return;
        }

        /*
         * If the pipe is full,
         * wait for reads to deplete
         * and truncate it.
         */

        if(ip->i_size >= PIPSIZ) {
                ip->i_mode |= IWRITE;
                prele(ip);
                sleep((caddr_t)ip+1, PPIPE);
                goto loop;
        }

        /*
         * Write what is possible and
         * loop back.
         */

        u.u_offset = ip->i_size;
        u.u_count = min ((unsigned)c, (unsigned)PIPSIZ-u.u_offset);
        c -= u.u_count;
        writei(ip);
        prele(ip);
        if(ip->i_mode&IREAD) {
                ip->i_mode &= ~IREAD;
                wakeup((caddr_t)ip+2);
        }
        goto loop;
}

/*
 * Lock a pipe.
 * If its already locked,
 * set the WANT bit and sleep.
 */
plock(ip)
register struct inode *ip;
{
        extern sleep();

        while(ip->i_flag&ILOCK) {
                ip->i_flag |= IWANT;
                sleep(ip, PINOD);
        }
        ip->i_flag |= ILOCK;
}

/*
 * Unlock a pipe.
 * If WANT bit is on,
 * wakeup.
 * This routine is also used
 * to unlock inodes in general.
 */
prele(ip)
register struct inode *ip;
```

```
{
        extern wakeup();

        ip->i_flag &= ~ILOCK;
        if(ip->i_flag&IWANT) {
                ip->i_flag &= ~IWANT;
                wakeup((caddr_t)ip);
        }
}
```

TD 13 - Montage des volumes

But

Le but de ce TD est l'étude du montage de volume, notion permettant aux utilisateurs du système de s'affranchir de la notion de disque physique.

Prérequis

Vous devez avoir assimilé ce qu'est un volume.

Question 1

Quelle est la différence entre *volume* et *disque physique* ?

Rappelez les différentes zones composant un volume, leur rôle, leur structure.

Comment l'administrateur du système prépare-t-il un volume pour utilisation ?

Reponse

- Notion de volume

  Différentes appellation équivalentes à "volume" : partition, disque virtuel, file system. Un volume est une organisation logique d'une partie de disque, il contient une arborescence complète. Plusieurs volumes peuvent coexister sur un même disque physique. C'est le driver qui fait la distinction suivant le mineur associé au volume. Les partitions du disque sont soit cablées dans le driver, soit mises dans une partie du disque (le "label") ne faisant pas partie des volumes.

- Avantages et inconvénients du partitionnement

  Avantage : quand un disque se crashe (crash logique), tout le disque n'est pas endommagé. Desavantage : moins grande souplesse, car il peut ne plus y avoir de place sur un volume et plein de place sur les autres.

  Un utilisateur ne perçoit pas fréquemment la notion de volume (Unix fait tout pour la cacher). Cependant, "ln" est interdit entre deux volumes, "df" liste la place disponible volume par volume, etc. De même, lorsqu'un volume est plein, la notion est explicite !

- Organisation d'un volume

  Un volume contient un boot block (bloc 0), un super block (bloc 1), une zone d'inodes et la zone des données. Le tout forme une arborescence avec une racine. L'inode de la racine est toujours l'inode numéro 2. Les inodes 0 et 1 sont inutilisées pour des raisons historiques (utilisées autrefois comme tête de la liste des blocs libres par exemple).

- Commandes d'administration

  mkfs crée un volume, c'est à dire crée un super-block, une zone d'inodes ne contenant que le répertoire racine, et une zone de données ou presque tous les blocs sont dans la free-list (presque, car le répertoire racine consomme un bloc). Il existe des commandes (suivant le constructeur) pour le formattage physique des disques.

  fsck vérifie un volume. Il procède à un certain nombre de vérifications telles que, par exemple, un bloc n'appartienne pas à deux fichiers en même temps, qu'il appartienne soit à un fichier, soit à la free-list, que tous les inodes occupés sont bien adressés par une entrée de directory, etc.

- Le superbloc est toujours en mémoire

  Le superbloc réside dans un buffer. Ce buffer n'est pas soumis à la gestion LRU du buffer cache car il est retiré de la bfreelist. Le superbloc est lu en mémoire au montage du volume.

Question 2

Qu'est-ce que le montage de volume ? A quoi cela sert-il ? En quelles circonstances l'utilisateur d'un système Unix prend-il conscience de la notion de volume ?

Reponse

Montage et démontage de volumes

Au boot, un seul volume est accessible, c'est le volume qui constitue le disque système. Sa racine devient la racine de l'aborescence. Le montage consiste à accrocher la racine d'un volume quelque part dans l'aborescence existante. La racine de ce volume devient alors un répertoire quelconque. Ceci permet d'abstraire l'utilisateur de la notion de disque, de frontière, etc. Il ne voit plus qu'une arborescence sans s'occuper des passages d'un disque à l'autre.

Le démontage est l'opération inverse. C'est utile lors de l'extinction, ou lors d'une vérification de ce volume par exemple.

Un volume = système de fichier = une arborescence avec une racine Montage = accrocher la racine d'un volume à la place d'un noeud (point de montage, forcément un répertoire) dans une arborescence existante On peut monter des volumes sur des volumes eux-mêmes déjà montés (c'est à dire qu'on n'est pas obligé de monter des volumes sous la racine).

Utilité : le montage permet de s'affranchir de la notion de "volume". La vision présentée à l'utilisateur est une vision abstraite d'une arborescence pure. Il n'y a pas de "SYS$..." (VMS) ou de "A:" (DOS) ou de "309" (VM/CMS) pour identifier un disque physique. Même le Macintosh présente la notion de "disques séparés". Unix homogénéïse tout en une seule arborescence.

Vision des volumes par l'utilisateur : Quand il n'y a plus de place sur un volume (alors qu'il peut rester de la place libre sur d'autres) =¿ commande "df", la seule à montrer les volumes à l'utilisateur Il y a aussi la commande "mount" sans arguments, mais c'est plus limité (indique seulement les volumes montés, accessible normalement seulement par l'administrateur).

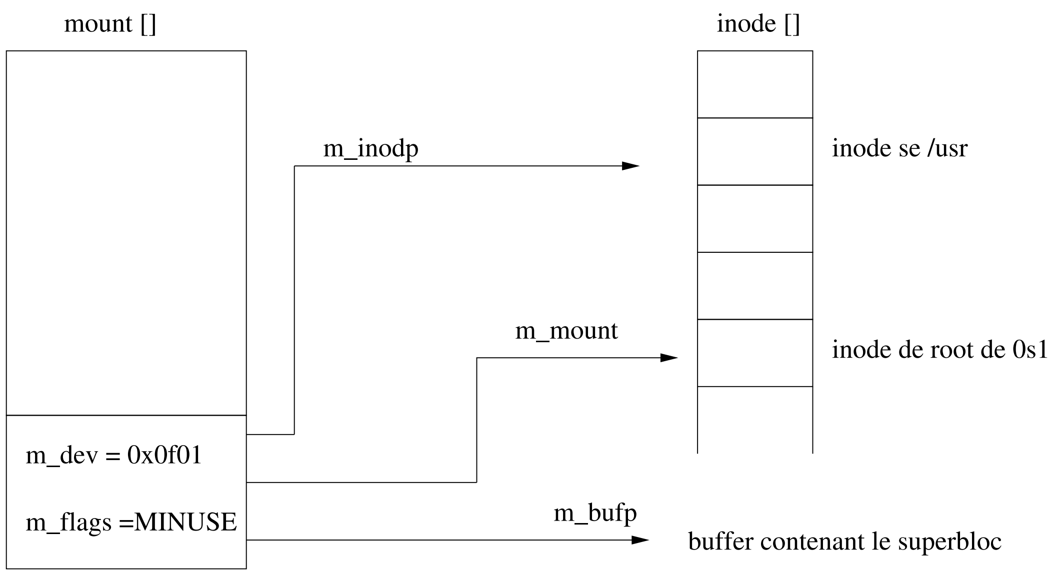
Attention : Le montage rend invisible tout ce qui était dans le répertoire de montage avant. Ce sera à nouveau visible lors du démontage.

Question 3

Expliquez l'organisation de la table `mount`.

Reponse

Exemple : mount /dev/dsk/0s1 /usr
avec /dev/dsk/0s1 => majeur = 15, mineur = 1 (=> dev = 0x0f01)



Il faut noter que le buffer contenant le superbloc est toujours en mémoire. Pourquoi ? C'est une donnée cruciale utilisée en permanence (ex : allocation et désallocation de bloc disque, c'est fait dès qu'il y a un read ou un write à faire). Comment ? Le buffer est retiré de la liste "av_", ce qui fait qu'il n'est plus pris en compte par l'algorithme LRU du buffer-cache, le buffer n'est jamais réutilisé pour d'autres données. En revanche, il est toujours dans la liste "b_", sinon il y aurait risque d'incohérence : par exemple, si j'ouvre /dev/dsk/0s1 en écriture et que je fais un write sur le superbloc, il faut que le superbloc puisse être accédé par les voies normales (chaînage "b_").

Question 4

Expliquez les opérations effectuées par les primitives `mount` et `umount`.

Expliquez en particulier les interactions avec le driver du disque correspondant au volume.

Détaillez les problèmes de partage des inodes.

Reponse

```
    mount

si uid utilisateur != 0
    alors erreur
fin si
chercher l'inode correspondant au périphérique et la ramener en mémoire
    si elle n'y est pas déjà
vérifier le périphérique
chercher l'inode correspondant au chemin et la ramener en mémoire
    si elle n'y est pas déjà
vérifier le point de montage
interdire le montage si inode déjà utilisée (ex: "ls" en cours sur
    le point de montage, un utilisateur a fait "cd" dedans)
interdire le montage à la place de la racine
chercher une entrée vide dans la table de montage
marquer l'entrée comme "état intermédiaire"
ouvrir le driver (fonction "open") du disque
lire le superbloc (le buffer est enlevé de "av_" par geteblk)
```

```
marquer l'entrée comme "occupée"
initialiser le superbloc en mémoire
lire l'inode de la racine du volume
marquer cette inode comme "point de montage" (pour namei, Q5)
libérer les inodes

    umount

si uid utilisateur != 0
    alors erreur
fin si
chercher l'inode correspondant au périphérique et la ramener en mémoire
    si elle n'y est pas déjà
vérifier le périphérique
chercher le périphérique dans la table "mount"
provoquer un "sync" (fonction update)
si le volume est utilisé (répertoire courants, fichiers en exécution,
    fichiers ouverts), interdire le démontage
fermer le driver (fonction "close") du disque
marquer les buffers utilisant ce volume comme "non valides" (binval)
remettre le buffer contenant autrefois le superbloc dans la liste "av_"
```

Interactions avec le driver

mount => open + appel à strategy (read) pour superbloc et inode racine umount => close

Partage des inodes

verrouillé lors d'un "iget" appelée par "namei" déverrouillé lors du "iput"

Question 5

En quelles circonstances le noyau doit-il gérer la notion de montage ?

Reponse

Notion de montage dans le noyau

La notion de montage intervient bien sur dans "mount" et "umount".

Le noyau dans prendre en compte la notion particulière de montage et de changement de volume lors d'une descente dans l'arborescence par namei, ou lors d'une remontée par "..". C'est à dire lors d'une traversée d'une frontière de volume, ou encore lors d'une traversée d'un point de montage. Dans tous les cas, c'est "namei" qui isole la notion de montage du reste du noyau.

Question 6

Modifiez les fonction *iget* en prennant en compte le montage des volumes.

Reponse

Fichier iget-mount.c

```
#
#include "../param.h"
#include "../systm.h"
#include "../user.h"
#include "../inode.h"
#include "../filsys.h"
#include "../conf.h"
#include "../buf.h"

/*
 * Look up an inode by device,inumber.
 * If it is in core (in the inode structure),
 * honor the locking protocol.
 * If it is not in core, read it in from the
 * specified device.
 * If the inode is mounted on, perform
 * the indicated indirection.
 * In all cases, a pointer to a locked
 * inode structure is returned.
 *
 * printf warning: no inodes -- if the inode
 * structure is full
 * panic: no imt -- if the mounted file
 * system is not in the mount table.
 * "cannot happen"
 */
iget(dev, ino)
{
register struct inode *p;
register *ip2;
int *ip1;
register struct mount *ip;

loop:
ip = NULL;
for(p = &inode[0]; p < &inode[NINODE]; p++) {
if(dev==p->i_dev && ino==p->i_number) {
if((p->i_flag&ILOCK) != 0) {
p->i_flag =| IWANT;
sleep(p, PINOD);
goto loop;
}
if((p->i_flag&IMOUNT) != 0) {
for(ip = &mount[0]; ip < &mount[NMOUNT]; ip++)
if(ip->m_inodp == p) {
dev = ip->m_dev;
ino = ROOTINO;
goto loop;
}
panic("no imt");
}
p->i_count++;
p->i_flag =| ILOCK;
return(p);
```

```
}
if(ip==NULL && p->i_count==0)
ip = p;
}
if((p=ip) == NULL) {
printf("Inode table overflow\n");
u.u_error = ENFILE;
return(NULL);
}
p->i_dev = dev;
p->i_number = ino;
p->i_flag = ILOCK;
p->i_count++;
p->i_lastr = -1;
ip = bread(dev, ldiv(ino+31,16));
/*
 * Check I/O errors
 */
if (ip->b_flags&B_ERROR) {
brelse(ip);
iput(p);
return(NULL);
}
ip1 = ip->b_addr + 32*lrem(ino+31, 16);
ip2 = &p->i_mode;
while(ip2 < &p->i_addr[8])
*ip2++ = *ip1++;
brelse(ip);
return(p);
}
```

Question 7

Proposez un schéma pour gérer un système de fichiers distribué (c'est à dire des volumes montés à partir d'autres ordinateurs accessibles par un réseau).

Reponse

NFS

Une table contenant des pointeurs sur des fonctions mettant

en oeuvre la politique du système de fichier (c'est le File System Switch d'ATT, ou le Virtual File System de Sun). Exemple de fonctions : mount/umount, open/close, read/write, stat

Fichier sys3.c

```
extern int uchar();

/*
 * the mount system call.
 */
smount(fspec, freg, ronly)
char *fspec, *freg;
int ronly;
{
        dev_t dev;
        register struct inode *ip;
        register struct mount *mp;
        struct mount *smp;
        register struct filsys *fp;
        struct inode *bip = NULL;

        if(!suser())
                return;

        u.u_dirp = fspec;
        ip = namei(uchar, 0);
        if(ip == NULL)
                return;
        if((ip->i_mode&IFMT) != IFBLK)
                u.u_error = ENOTBLK;
        dev = (dev_t)ip->i_rdev;
        if(bmajor(dev) >= bdevcnt)
                u.u_error = ENXIO;
        if (u.u_error)
                goto out;
        bip = ip;

        u.u_dirp = (caddr_t)freg;
        ip = namei(uchar, 0);
        if(ip == NULL) {
                iput(bip);
                return;
        }
        if ((ip->i_mode&IFMT) != IFDIR) {
                u.u_error = ENOTDIR;
                goto out;
        }
        if (ip->i_count != 1)
                goto out;
        if (ip->i_number == ROOTINO)
                goto out;

        smp = NULL;
        for(mp = &mount[0]; mp < &mount[NMOUNT]; mp++) {
                if(mp->m_flags != MFREE) {
                        if (dev == mp->m_dev)
                                goto out;
                } else
                if(smp == NULL)
```

```
                        smp = mp;
        }
        mp = smp;
        if(mp == NULL)
                goto out;
        mp->m_flags = MINTER;
        mp->m_dev = dev;
        (*bdevsw[bmajor(dev)].d_open)(minor(dev), !ronly);
        if(u.u_error)
                goto out1;

        mp->m_bufp = geteblk();
        fp = mp->m_bufp->b_filsys;
        u.u_offset = SUPERBOFF;
        u.u_count = sizeof(struct filsys);
        u.u_base = (caddr_t)fp;
        u.u_segflg = 1;
        readi(bip);
        if (u.u_error) {
                brelse(mp->m_bufp);
                goto out1;
        }

        mp->m_inodp = ip;
        mp->m_flags = MINUSE;
        fp->s_ilock = 0;
        fp->s_flock = 0;
        fp->s_ninode = 0;
        fp->s_inode[0] = 0;
        fp->s_ronly = ronly & 1;
        if (mp->m_mount = iget(mp->m_dev, ROOTINO))
                prele(mp->m_mount);
        else {
                brelse(mp->m_bufp);
                goto out1;
        }
        ip->i_flag |= IMOUNT;
        iput(bip);
        prele(ip);
        return;

out1:
        mp->m_flags = MFREE;
out:
        if (bip != NULL)
                iput(bip);
        if (u.u_error == 0)
                u.u_error = EBUSY;
        iput(ip);
}

/*
 * the umount system call.
 */
sumount(fspec)
char *fspec;
{
        dev_t dev;
        register struct inode *ip;
```

```
	register struct mount *mp;

	if(!suser())
		return;
	u.u_dirp = (caddr_t)fspec;
	dev = getmdev();
	if(u.u_error)
		return;
	for(mp = &mount[0]; mp < &mount[NMOUNT]; mp++)
		if(mp->m_flags == MINUSE && dev == mp->m_dev)
			goto found;
	u.u_error = EINVAL;
	return;

found:
	xumount(dev);	/* remove unused sticky files from text table */
	update();
	if (mp->m_mount) {
		plock(mp->m_mount);
		iput(mp->m_mount);
		mp->m_mount = NULL;
	}
	for(ip = &inode[0]; ip < &inode[NINODE]; ip++)
		if(ip->i_number != 0 && dev == ip->i_dev) {
			u.u_error = EBUSY;
			return;
		}
	(*bdevsw[bmajor(dev)].d_close)(minor(dev), 0);
	binval(dev);
	ip = mp->m_inodp;
	ip->i_flag &= ~IMOUNT;
	plock(ip);
	iput(ip);
	brelse(mp->m_bufp);
	mp->m_bufp = NULL;
	mp->m_flags = MFREE;
}

/*
 * Check that the user's argument is a reasonable
 * thing on which to mount, and return the device number if so.
 */
dev_t
getmdev()
{
	dev_t dev;
	register struct inode *ip;

	ip = namei(uchar, 0);
	if(ip == NULL)
		return(NODEV);
	if((ip->i_mode&IFMT) != IFBLK)
		u.u_error = ENOTBLK;
	dev = (dev_t)ip->i_rdev;
	if(bmajor(dev) >= bdevcnt)
		u.u_error = ENXIO;
	iput(ip);
	return(dev);
}
```

TD 14 - Le swap

But

Le but de ce TD est de comprendre le fonctionnement du *swap* sous Unix à travers l'exemple d'une version simple.

Prérequis

Vous devez avoir assimilé les routines de synchronisation.

Le système decrit ici ne possède pas de gestion de mémoire virtuelle (paginée) mais uniquement un système de swap (sauvegarde de segments entiers - de taille variable - et non de pages de taille fixe).

Question 1

Quelles sont les caractéristiques du processus 0 ($pid = 0$) ?

Reponse

Le processus 0 est le seul processus à ne pas être créé par fork ou sa version intene (newproc). Les entrées dans la table proc[] ainsi que la structure u sont créées "manuellement" lors du démarrage du système. Ce processus n'a pas de segment text, data ou stack.

Après création du processus 1, le noyau exécute dans le contexte du processus 0 la fonction sched (boucle infinie) : il n'y a donc pas de partie utilisateur, et le processus 0 passe son existence en mode système. Il n'y a pas besoin de segments text, data et stack puisque ce processus passe sa vie dans la boucle de sched() et ne revient jamais en mode utilisateur

Question 2

Quelles sont les conditions qui provoquent la mise en attente du processus 0 ?

Expliquez le rôle des variables `runin` et `runout`.

Quels sont les critères qui provoquent l'éviction (*swap out*) d'un processus de la mémoire, le rappel (*swap in*) d'un processus en mémoire ?

Reponse

Le processus 0 est mis en attente :

- s'il n'y a pas de processus prêt à tourner à ramener de la zone de swap vers la mémoire centrale. Cette situation correspond à runout != 0 On cherche à ramener le processus swappé, prêt à tourner présent sur le swap depuis le plus long temps.
- s'il n'y a pas de processus pouvant être évincé sur le swap. Cette situation correspond à runin != 0 On considère qu'un processus ne peut être swappé "facilement" dans deux cas :
  - il n'y a pas de processus en mémoire et arrêté et le processus à ramener n'a pas attendu très longtemps (moins de 3 secondes) ;
  - si le processus à ramener a attendu plus de 3 secondes, pas de processus présent depuis plus de deux secondes en mémoire.

Question 3

Décrivez l' algorithme de `sched`.

Reponse

```
debut_boucle:
    début de section critique
pour tous les ps prêts à tourner, swappés et dont temps sur disque > n
    faire
p1 := ce processus
n := le temps de résidence de ce ps sur disque
fin pour
/* p1 = le processus swappé qui attend depuis le plus longtemps */
si il n'y a pas de processus à ramener en mémoire
    alors
runout := 1
attendre (runout)
revenir à "debut_boucle"
fin si
    fin section critique

    /* p1 est le processus à ramener en mémoire */
    a := taille du processus
    si le ps a une entrée dans "text" mais si le texte est swappé
alors a += taille du segment texte
    fin si
    allouer de la mémoire
    si allocation réussie
alors aller à "ramener"
    fin si

    /* pas assez de mémoire pour ramener le processus */
    début section critique
/* on va éjecter les processus endormis */
pour tous les processus
    faire
si ps non système, en mémoire, non verrouillés et en sleep
    alors aller à "ejecter"
fin si
fin pour
/* aucun processus endormi à éjecter */
si le processus n'attend pas sur disque depuis plus de 3 secondes
    alors
runin := 1
attendre (runin)
```

```
revenir à "debut_boucle"
fin si
/* on va chercher à éjecter un processus qui est prêt à tourner */
pour tous les ps en mémoire
    faire
si ps non système, en mémoire, non verrouillés, "prêt à tourner"
   ou "endormi", et temps de résidence en mémoire > n
    alors
p1 := ce processus
n := le temps de résidence de ce ps en mémoire
fin si
fin pour
si aucun processus n'est en mémoire depuis plus de 2 secondes
    alors
runin := 1
attendre (runin)
revenir à "debut_boucle"
fin si
/* p1 = le processus qui est en mémoire depuis le plus longtemps */


ejecter:
    fin section critique
    marquer le processus "swappé" dans proc
    swapper ce processus
    revenir à "debut_boucle"

ramener:
    si le processus a une entrée dans "text"
alors
    si x_ ccount = 0
alors ramener le texte en mémoire
    fin si
    ccount++
    fin si
    ramener le reste du processus en mémoire (segments data et stack)
    libérer la place dans l'espace de swap
    marquer le processus "en mémoire" dans proc
    temps de résidence du processus en mémoire (p_time) := 0
    revenir à "debut_boucle"
```

Question 4

Comment est réalisée l'entrée/sortie correspondant au swap ? Quelle est la structure de donnée utilisée ?

Où est situé l'espace de swap ?

Expliquez la procédure `swap`.

Reponse

swap gère les transferts entre la mémoire et la zone de swap

Les paramètres sont :

- numéro du bloc de swap à partir duquel on trouve ce que l'on veut ramener ou évincer (text ou data+stack)

- adresse en mémoire de la zone à transférer
- taille de l'E/S
- flag pour strategy (lecture/écriture)

Le swap est réalisé comme une entrée/sortie normale :
- au travers d'une structure buf (swbuf) qui contient les paramètres. En particulier, la structure swbuf pointe sur la zone mémoire ou le segment doit être copié (ou lu) et non vers un buffer comme dans le filesystem.

Le swap est une zone située sur disque à partir du bloc swplo. La mémoire est allouée dans cette zone par un algorithme "first fit". La fonction malloc (système) réserve l'espace en mémoire centrale ou dans la zone de swap (même chose symétriquement avec mfree).

```
Algorithme :

    début de section critique
    tant que swbuf est occupé
marquer swbuf comme demandé
attendre
    initialiser swbuf :
numéro de périphérique,
buffer occupé
numéro de bloc sur disque
adresse ou segment en mémoire
taille de ce qui doit être transféré
    exécuter l'entrée/sortie (strategy)
    début de section critique
    attendre la fin de l'entrée/sortie
    réveiller les processus en attente de swbuf
    fin de section critique
    remettre à zéro l'état de swbuf
```

Question 5

La procédure `xswap` gère le cas de l'éviction en présence de segments *text* partagés. Quelles sont les actions effectuées par `xswap` ?

Reponse

```
Algorithme :

    début de section critique
    tant que swbuf est occupé
marquer swbuf comme demandé
attendre
    initialiser swbuf :
numéro de périphérique,
buffer occupé
numéro de bloc sur disque
adresse ou segment en mémoire
taille de ce qui doit être transféré
    exécuter l'entrée/sortie (strategy)
    début de section critique
    attendre la fin de l'entrée/sortie
    réveiller les processus en attente de swbuf
```

```
fin de section critique
remettre à zéro l'état de swbuf
```

Fichier sched.c

```
/*
 * The main loop of the scheduling (swapping) process.
 * The basic idea is:
 *  see if anyone wants to be swapped in;
 *  swap out processes until there is room;
 *  swap him in;
 *  repeat.
 * The runout flag is set whenever someone is swapped out.
 * Sched sleeps on it awaiting work.
 *
 * Sched sleeps on runin whenever it cannot find enough
 * memory (by swapping out or otherwise) to fit the
 * selected swapped process.  It is awakened when the
 * memory situation changes and in any case once per second.
 */

sched()
{
        register struct proc *rp, *p;
        register struct text *xp;
        register int a, n;

        /*
         * find user to swap in;
         * of users ready, select one out longest
         */

loop:
        spl(CLINHB);
        n = -1;
        for (rp = &proc[0]; rp < &proc[NPROC]; rp++)
        if (rp->p_stat==SRUN && (rp->p_flag&SLOAD) == 0 &&
            rp->p_time > n) {
                p = rp;
                n = rp->p_time;
        }
        /*
         * If there is no one there, wait.
         */
        if (n == -1) {
                runout++;
                sleep((caddr_t)&runout, PSWP);
                goto loop;
        }
        spl(NORMAL);

        /*
         * See if there is memory for that process;
         */

        rp = p;
        a = rp->p_size;
        if((xp=rp->p_textp) != NULL)
                if(xp->x_ccount == 0)
```

```
                a += xp->x_size;
        if((a=malloc(coremap, a)) != NULL)
                goto found2;

        /*
         * none found.
         * look around for easy memory.
         * Select the largest of those sleeping
         * at bad priority; if none, select the oldest.
         */

        spl(CLINHB);
        for (rp = &proc[0]; rp < &proc[NPROC]; rp++)
        if ((rp->p_flag&(SSYS|SLOCK|SLOAD))==SLOAD &&
            (rp->p_stat==SSLEEP || rp->p_stat==SSTOP))
                goto found1;

        if(n < 3)
                goto sloop;
        n = -1;
        for (rp = &proc[0]; rp < &proc[NPROC]; rp++)
        if ((rp->p_flag&(SSYS|SLOCK|SLOAD))==SLOAD &&
            (rp->p_stat==SRUN || rp->p_stat==SSLEEP) &&
             rp->p_time > n) {
                p = rp;
                n = rp->p_time;
        }
        if(n < 2)
                goto sloop;
        rp = p;

        /*
         * swap user out
         */
found1:
        spl(NORMAL);
        rp->p_flag &= ~SLOAD;
        xswap(rp, 1, 0);
        goto loop;

        /*
         * swap user in
         */
found2:
        rp=p;
        if((xp=rp->p_textp) != NULL) {
                if(xp->x_ccount == 0) {
                    if(swap(xp->x_daddr, a, xp->x_size, B_READ))
                        goto swaper;
                    xp->x_caddr = a;
                    a += xp->x_size;
                }
                xp->x_ccount++;
        }
        if(swap(rp->p_addr, a, rp->p_size, B_READ))
                goto swaper;
        mfree(swapmap, (rp->p_size+7)/8, rp->p_addr);
        rp->p_addr = a;
        rp->p_flag |= SLOAD;
```

```
        rp->p_time = 0;
        goto loop;

sloop:
        runin++;
        sleep((caddr_t)&runin, PSWP);
        goto loop;

swaper:
        panic("swap error");
}

/*
 * swap IO header.
 */
struct  buf     swbuf;

/*
 * swap I/O
 */
swap(blkno, coreaddr, count, rdflg)
caddr_t coreaddr;
{
        register int c;
        register lkflg;

        spl(CLINHB);
        while (swbuf.b_flags&B_BUSY) {
                swbuf.b_flags |= B_WANTED;
                sleep((caddr_t)&swbuf, PSWP);
        }
        swbuf.b_dev = swapdev;
        swbuf.b_flags = B_BUSY | B_PHYS | rdflg;
        swbuf.b_blkno = swplo+blkno;
        swbuf.b_addr = coreaddr;
        swbuf.b_count = count;
        (*bdevsw[bmajor(swapdev)].d_strategy)(&swbuf);
        spl(CLINHB);
        while((swbuf.b_flags&B_DONE)==0)
                sleep((caddr_t)&swbuf, PSWP);
        if (swbuf.b_flags&B_WANTED)
                wakeup((caddr_t)&swbuf);
        spl(NORMAL);
        swbuf.b_flags &= ~(B_BUSY|B_WANTED|B_PHYS);
        return (swbuf.b_flags & B_ERROR);
}


/*
 * Swap out process p.
 * The ff flag causes its core to be freed--
 * it may be off when called to create an image for a
 * child process in newproc.
 * Os is the old size of the data area of the process,
 * and is supplied during core expansion swaps.
 *
 * panic: out of swap space
 * panic: swap error -- IO error
 */
```

```
xswap(p, ff, os)
int *p;
{
        register *rp, a;

        rp = p;
        if(os == 0)
                os = rp->p_size;
        a = malloc(swapmap, (rp->p_size+7)/8);
        if(a == NULL)
                panic("out of swap space");
        xccdec(rp->p_textp);
        rp->p_flag =| SLOCK;
        if(swap(a, rp->p_addr, os, 0))
                panic("swap error");
        if(ff)
                mfree(coremap, os, rp->p_addr);
        rp->p_addr = a;
        rp->p_flag =& ~(SLOAD|SLOCK);
        rp->p_time = 0;
        if(runout) {
                runout = 0;
                wakeup(&runout);
        }
}
```

TD 15 - Gestion de la Mémoire Virtuelle dans un système unix

But

Le but de ce TD est de comprendre la gestion de la mémoire virtuelle d'un système unix. Il est basé sur l'implantation de la mémoire virtuelle du système **BSD 4.4**.

Les questions/réponses sont basées sur le livre "Conception et Implémentation du système 4.4 BSD", McKusick et. all., Addison Wesley.

Question 1

1.1 Quelles sont les structures utilisées par le noyau pour gérer l'espace d'adressage d'un processus ? Quelles informations chacune de ces structures contient ?

Reponse

* Voir figure 1

- *vm_space* : structure qui englobe toutes les structures de données qui décrivent l'espace d'adressage d'un processus, c'est à dire, l'état de la mémoire virtuelle d'un processus. Elle encapsule les structures de données dépendantes et indépendantes de la machine ainsi que des statistiques;
- *vm_map* : structure de données de plus haut niveau qui décrit l'espace d'adressage virtuel de la machine.
- *vm_pmap* : (physical mapping) structure de données, dépendante de la machine, qui contient les informations relatives au matériel de gestion mémoire. Il gère la traduction des adresses virtuelles en physiques.
- *vm_map_entry* : structure qui décrit une plage d'adresses virtuelles contigües possédant les mêmes attributs de protection et d'héritage (région). Chaque *vm_map_entry* pointe sur une chaîne de structures *vm_object*. A la fin de la chaîne se trouve l'objet initialement mappé, qui représente une source de donnée comme un fichier. Entre cet objet et la structure *vm_map_entry* se trouvent un ou plusieurs objets ombre transitoire;
- *objet* : structure qui décrit une source de données pour une plage d'adresse (ex.: fichier, page anonyme). Chaque structure *vm_object* contient une liste chaînée de structures *vm_page*,), ainsi qu'un pointeur vers la structure *pager_struct* (paginateur), qui contient des informations sur le chargement ou l'éviction d'une page de l'objet.
- *objet ombre* : objet spécial qui représente une copie modifiées de données;

- *vm_page* : structure de données de plus bas niveau qui représente la mémoire physique utilisée par le système de mémoire virtuelle (cache en mémoire physique). Il existe une structure *vm_page* pour chaque page de mémoire physique géré par le système de mémoire virtuelle. Une page peut être une ensemble de plusieurs pages matérielles contigües. La structure contient également le statues de la page (modifié, référencée) et des liens vers différents files de pages.

1.2 Expliquez le concept de *région* d'un espace d'adressage, associée à la structure *vm_map_entry*.

Reponse

Le contenu de l'espace d'adressage est défini par une liste de structures *vm_map_entry*, décrivant chacune une région de l'espace d'adressage virtuel, située entre une adresse de début et une adresse de fin.

Une entrée *vm_map_entry* peut mappé des sources de données (objets) comme des fichiers ou des portions de swap anonymes. Autrement dit, une région décrit une plage mémoire dont toutes les adresses sont traitées de la même façon (ex. code d'un programme est une région en lecture seulement). Ainsi une entrée *vm_map_entry* contient le mode de protection associé à la région. Elle contient aussi un pointeur sur l'objet qui fournit les données initiales et qui peut être enchaîné avec d'autres objets lorsque l'objet original est modifié.

Une entrée *vm_map_entry* contient aussi un décalage qui indique l'endroit où commence le mapping de l'objet.

1.3 Quels sont les types d'objets mémoire gérés par le système ?

Reponse

- *Objets nommés* représentent des fichiers. Ils peuvent également représenter des périphériques matériels capable de fournir de la mémoire mapée, comme par exemple les *frame buffers* ;
- *Objets anonymes* représentent des zones de mémoire remplies de zéros lors de leur première utilisation. Les pages modifiées seront stockées dans la zone de swap;
- *Objets ombres* contiennent de copies privées de pages qui ont été modifiées ; ils sont abandonnés lorsqu'ils ne sont plus référencés.
- *Objets copies* contiennent d'anciennes page qui ont été modifiées après avoir été mappées de manière privée. ils sont abandonnés lorsque le mapping privé est abandonné.

1.4 Quelles sont les informations sauvegardées dans un objet ?

Reponse

Un objet contient les informations suivantes :

- Une liste des pages de l'objet qui résident en mémoire principale. Une page peut être mappée dans plusieurs espaces d'adressage, mais elle n'appartient toujours qu'à un seul objet.
- Le nombre de structures *vm_map_entry* ou d'autres objets qui référencent l'objet ;
- La taille du fichier ou de la zone anonyme que l'objet décrit ;
- Le nombre de pages mémoires résidentes que l'objet détient;
- Un pointeur vers le paginateur de l'objet .

Question 2

2.1 Quelles sont les quatres zones de mémoire allouées à un processus lors de sa création ? Où le noyau trouve les informations associées à ces zones pour pouvoir les créer ?

Reponse

Les besoins initiaux en mémoire virtuelle sont décrits dans l'en-tête de l'exécutable. Ces boisons incluent les 4 zones de mémoire à allouer : code, pile, données initialisées et données non initialisées.

2.2 Faites un schéma pour montrer la topologie initiale de l'espace d'adressage d'un processus en utilisant les structures décrites dans la question précédente.

Reponse

* Voir **figure 5.7, page 126** du livre.

Commentaires sur la figure: Les deux premières entrées *vm_map_entry* pointent sur le même objet, l'exécutable : le code réside au début du fichier et la zone de données initialisées se trouve après la fin du code. L'entrée concernant le code décrit une région en lecture tandis que celle concernant les données initialisées décrit un région en *copie sur l'écriture*. Le champ contenant le décalage reflète la valeur de départ différente pour le code et les données initialisées.

Les troisièmes et quatrième entrées décrivent respectivement les zones de données non initialisées et la pile. Ces deux zones sont représentées par des *objets anonymes*.

Question 3

3.1 Expliquez le concept de *copie sur écriture*.

Reponse

Plutôt que de copier chaque page du processus père lors d'un *fork*, le père et le fils utilisent des références pointant sur les mêmes pages physiques. Les tables de pages sont modifiées (bit *copie sur écriture* affecté) pour empêcher toute modification d'une page partagée soit par le père soit par le fils. Ainsi, un défaut de protection surviendra lorsqu'un des processus tentera de modifier une page partagée. Le noyau alors fait une copie de la page en modifiant sa protection pour qu'elle puisse être modifiée. La copie page de la page est associée à un objet *ombre*. Si celui-ci n'existe pas encore, il doit être créé et enchaîné avec l'objet original. Avec cette approche, seules les pages modifiées seront dupliquées, améliorant significativement les performance du *fork*.

3.2 Ajouter dans l'algorithme de *fork* la gestion de mémoire virtuelle.

Reponse

- réserver de l 'espace d'adressage virtuelle pour le fils;
- allouer une structure *proc* pour le fils et la remplir;
- Copier le group de processus du père (UID et GID), ses descripteurs de fichier, ses limites et ses actions de signaux ;
- allouer la *zone u* et l'initialiser en y copiant la zone courante ;
- allouer un structure *vmspace* ;
- dupliquer l'espace d'adressage, en créant des copies des structures *vm_map_entry* du père, et en les marquant copie sur écriture si nécéssaire (ex. :pile, données);

- Faire de sorte que le processus fils retourne 0 au père.

3.3 Quelles sont les différentes actions prises par le noyau lorsqu'il analyse les régions comprennant l'espace d'adressage du père : région en lecture (ex. code), mapping privé (ex. pile), mapping partagé (ex. mémoire partagée)?

Reponse

La commande *fork* parcourt la liste de structure *vm_map_entry* du père en créant des entrées correspondantes dans le fils. Chaque entrée doit être analysée avant qu'une action est prise :

- Si l'entrée mappe une région en lecture (code), le fils peut la référencer ;
- Si l'entrée mappe une région privée (pile, données), le fils doit créer un mapping en *copie sur l'écriture* de la région. Le père convertit aussi son mapping de la région en *copie sur écriture*. Par conséquent, si l'un des deux processus essaie par la suite d'écrire dans la région, il créera un objet *ombre* pour contenir les pages modifiées;
- Si l'entrée mappe une région de partage, une carte de partage (structure *vm_map*) est créée pour référencer l'objet partagé. Les deux entrées *vm_map_entry* référencent désormais cette carte partagée.

3.4 Nous considérons que la pile d'un processus occupe 4 pages. Ces pages se trouvent chargées en mémoire. Après avoir créé un processus fils (*fork*), ce processus écrit sur une variable locale qui se trouve dans la page 0 du segment de la pile. De son côte, le processus fils écrit sur une variable locale qui se trouve dans la page 2 du segment de la pile. Représentez dans un schéma les structures *vm_map_entry* correspondant à la pile des processus père et fils.

Reponse

* Voir figure 3

3.5 Comment l'espace d'adressage d'un processus est remplacé par un nouveau espace d'adressage lors de l'exécution de la commande *exec* ?

Reponse

Pour exécuter un *exec*, le système doit allouer des ressources pour le nouveau contenu de l'espace d'adressage virtuel, mettre en place le mapping de l'espace d'adressage pour qu'il référence la nouvelle image, et libérer les ressources utilisées par l'espace d'adressage existant.

La première étape est de réserver des ressources mémoire pour l'image du nouvel exécutable. Il ne faut réserver de la place que pour ses données et sa pile. Une fois que la réservation est faite, l'espace d'adressage et les ressources de mémoire du processus courant sont libérés comme s'il se terminait.

Ayant déjà une *zone u* et une pile noyau, le noyau alloue alors une nouvelle structure *vmspace* au processus et crée une liste avec les 4 structure *vm_map_entry* : code, segment de données initialisées, données non initialisées et pile.

Le reste du travail comprend la copie des arguments et de l'environnement au sommet de la pile. Les registres prennent de nouvelles valeurs : le pointeur d'instruction point sur le début du code à exécuter et le pointeur de la pile pointe sur le vecteur d'arguments.

3.6 Quelle est la différence entre la commande *vfork* et *fork* ?

Reponse

Si le processus père et fils ne se exécuteront pas simultanément (ex : shell où le père se suspend pour attendre la fin du fils), une seule copie de l'espace d'adressage du processus est suffisante. La commande *vfork* a été créée pour ce type de situation.

Lors de la commande *vfork*, le noyau passe l'espace d'adressage du père au fils et suspend le père. Cependant, le processus fils n'a qu'allouer de nouvelles structure *proc* et la *zone u*, puisqu'il reçoit tout le reste de son père. Lorsque le processus fils a exécuté la commande *exec*, le père récupère ensuite son espace d'adressage, au lieu que le fils ne l'abandonne comme dans un *exec* courant. Avec *vfork* les entrées *vm_map_entry* du père n'ont pas été dupliquées et les tables de pages n'ont pas été marquées en *copie sur écriture*. La commande *vfork* reste alors plus efficace que *fork* .

Question 4

Lorsqu'un processus tente d'accéder à une partie de son espace d'adressage qui n'est pas en mémoire, un défaut de page se produit. Décrivez le traitement réalisé par le gestionnaire de défaut de page (*vm_fault*) lors d'un défaut de page.

Reponse

Le gestionnaire de défauts de page du noyau reçoit l'adresse virtuelle qui a provoqué le défaut. Il est traité en quatre étapes :

- trouver la structure *vmspace* de processus fautif ; à partir de cette structure trouver la tête de la liste des *vm_map_entry* ;
- parcourir la liste des *vm_map_entry* ; pour chaque entrée, vérifier si l'adresse fautive est contenue dans la plage d'adresses. Si le noyau atteint la fin de la liste sans trouver de région valide, l'adresse ne fait pas partie de l'espace d'adressage du processus : envoyer au processus un signal de violation de segment ;
- une fois que la bonne entrée *vm_map_entry* a été trouvée, convertir cette adresse en un décalage à l'intérieur de l'objet : *object_offset = fault_adress - vm_map_entry-> start_address + vm_map_entry-> object_offset* (soustraire l'adresse de départ pour obtenir le décalage dans la région mappée par *vm_map_entry* ; ajouter le décalage de l'objet pour obtenir le décalage absoulu de la page à l'intérieur de l'objet);
- présenter le décalage absolu à l'objet, qui crée alors une structure *vm_page* et demande à son paginateur de la remplir. L'objet retourne ensuit un pointeur sur la structure *vm_page*, mappée à l'endroit correspondant au défaut dans l'espace d'adressage du processus.

Question 5

5.1 Qu'est que c'est un paginateur ?

Reponse

Un paginateur est chargé de fournir les données nécessaires au remplissage d'une page ainsi que l'espace d'adressage nécessaire à son stockage lorsque l'objet auquel il est associé a été modifié. Il fournit le mécanisme grâce auquel les données sont transférées entre la mémoire physique et la mémoire secondaire.

A chaque objet est associée une structure *page_struct* qui répresent un instance du type de paginateur, chargé de fournir le contenu des pages de l'objet. Une structure *page_struct* contient des pointeurs vers de routines spécifiques au type d'objet, permettentant de lire et d'écrire de données, ainsi qu'un pointeur vers le lieu de stockage (ex. un morceau du swap ou un fichier).

Une structure *page_struct* et les données associées sont appelées *instance de paginateur*.

Lors d'un défaut de page, le gestionnaire de défauts alloue une structure *vm_page* associé à l'objet en question et convertit l'adresse de défaut en un décalage à l'intérieur de l'objet. Le cadre de page est passé à l'instance du paginateur pour qu'il la remplisse. L'instance du paginateur est également chargée de sauvegarder le contenu d'une page "sale" lorsque le démon de pagination décide de la déplacer vers la mémoire secondaire.

5.2 Comment une instance d'un paginateur est associé à un objet ?

Reponse

Une instance de paginateur est crée au même moment que l'objet, c'est-à-dire, lors du mapping d'un fichier, d'un périphérique, ou d'une zone de mémoire anonyme dans l'espace d'adressage d'un processus. L'instance existe jusque l'objet soit libéré.

5.3 Décrivez les trois différents types de paginateurs existants en spécifiant le type d'objet qu'ils gèrent.

Reponse

- *paginateur vnode* : gère des objets qui mappent des fichiers. Lorsqu'un fichier est mappé, soit explicitement avec *mmap*, soit implicitement avec *exec*, la routine d'allocation du paginateur de vnode est appelée.
- *paginateur périphérique* : gère des objets représentant des périphériques mappés en mémoire. Ceux-ci fournissent une interface ressemblant à un morceau de mémoire. Par exemple, le *frame buffer* représente une place d'adresses mémoire dont chaque mot correspond à un pixel. La mémoire du périphérique est donc mappé dans l'espace du processus.
- *paginateur de swap* : il se réfère au paginateur utilisé par des objets qui mappent la mémoire anonyme. Le rôle du paginateur de swap est de gérer l'espace de swap, c'est-à-dire, où ranger les pages "sales" (modifiées) et comment les retrouver lorsqu'on en a besoin.

5.4 Quelles sont les deux fonctions associées à un paginateur responsables pour obtenir/sauvegarder une ou plusieurs page ? Qui les appelle ?

Reponse

Pgo_getpages(): sert a obtenir une ou plusieurs pages de données fournies par l'instance du paginateur. Cette routine n'est appelée que par le gestionnaire de défauts de pages pour remplir des pages de manière synchrone.

Pgo_putpages() : elle est appelée pour sauvegarder une ou plusieurs page de donées, soit par le démon de pagination, soit par la fonction *msync* (qui vide sur disque le cache de la mémoire principal).

5.5 Comment un paginateur intervient, lorsque le démon de pagination décide de libérer une page ?

Reponse

L'instance du paginateur est chargée de sauvegarder le contenu d'un page "sale" lorsque le système décide de la déplacer vers la mémoire secondaire, cela dit, lorsque le démon de pagination décide qu'une page donnée est devenu inutile. A ce moment, le démon de pagination demande à l'objet qui possède la page de la libérer. L'objet passe alors la page et son décalage à l'*instance du paginateur* pour qu'il la sauvegarde en disque. Le paginateur doit alors trouver un endroit approprié sur le swap pour la sauvegarder, effectuant les E/S nécessaires. Puis il signale à l'objet que la page peut être libérée. L'*instance du paginateur* prévient après le démon que la page a été libérée afin qu'il puisse placer la structure *vm_page* correpondante dans la liste de *pages libres*.

Question 6

6.1 Le noyau divise les pages en mémoire en listes pour pouvoir mettre en place sa gestion de remplacement de page. Quelles sont les listes gérées par le noyau?

Reponse

Le noyau divise la mémoire principale en quatre listes :

- *verrouillée* : les pages verrouillées ne peuvent pas être paginées. Ces pages sont généralement en train d'être utilisées par le noyau, ou ont été verrouillées avec la fonction *mlock*;
- *active* : les pages actives sont en cours d'utilisation par une ou plusieurs régions de mémoire virtuelle. Sa pagination, entraînera au défaut de page;
- *inactive* : les pages inactives ont un contenu valide, mais elles ne font plus partie d'aucune région active. Si le système a besoin de mémoire, le démon de pagination peut essayer de déplacer des pages actives vers la liste de pages inactives;
- *libre* : le contenu des pages libres n'est plus valable. Elles servent à satisfaire les requêtes d'allocation d'une nouvelle page.

6.2 Quel est le rôle du démon de pagination ? Quand est-il crée ?

Reponse

Les remplacement de pages sont effectuées par le démon de pagination. Lorsqu'il récupère des pages qui ont été modifiées, il est chargé de les écrire dans la zone de swap (en appelant le paginateur associé à l'objet). Il s'exécute donc en tant que processus séparé.

Le demon de pagination est créé avec *fork* durant le démarrage du système. Il reste en mode noyau après le *fork*. (il boucle dans la fonction *vm_pageout()*). Le but du démon de pagination est de conserver au moins 5 % de mémoire dans la liste libre. Lorsqu'une opération qui utilise des pages entraîne le franchissement de cette limite, le démon est réveillé.

6.3 Décrivez l'algorithme du démon de pagination ainsi que sa politique de remplacement de page en utilisant les listes décrites ci-déssus.

Reponse

Lorsqu'une opération qui utilise une page entraîne la chute des 5% de pages dans la liste de *pages libres*, le démon est réveillé. Il commence par vérifier si certains processus peuvent être swappés. S'il en trouve suffisamment pour que la liste libre remonte au-delà de la limite, il les swappe et se rendort. Si cela ne suffit pas, il doit parcourir les listes :

* Voir figure 2

- Lors d'un défaut de page, le noyau alloue une page de la liste de *libre*. Une fois que le paginateur asoccié a remplit la page, elle est placée dans la liste de pages *active* (figure 2 -1).
- Le démon déplace quelques pages du début de la liste *active* vers la fin de la liste *inactive* en mettant à 0 le bit de référence (figure 2 - 2). Autrement dit, les page déplacées vers la liste *inactive* sont celles qui ont été référencées le moins récemment ;
- Il examine quelques pages du début de la liste *inactive*. Les pages dont le bit de référence est affecté sont déplacées vers la fin de la liste de pages *active* (figure 2 - 3) ; Si la page est dans la liste *inactive* est qu'elle n'a pas été modifiée ("propre") ni référencée (bit référence égal à 0), elle est déplacée vers la fin de la liste de *libre* (figure2 - 4). Par contre, si la page a été modifiée, elle doit d'abord être sauvegardée en disque;
- Si la page est référencée lorsqu'elle est dans la liste *libre*, elle est déplace vers la fin de la liste *active* (figure2 - 5).

Annexes des Travaux Dirigés UNIX

Définition des types et
structures de données du noyau