M1 - Noyau
Unix – Mécanismes Internes

Sept. 2013

Equipe pédagogique :
ARANTES Luciana (Luciana.Arantes@lip6.fr)
SENS Pierre (Pierre.Sens@lip6.fr)
SOPENA Julien (Julien.Sopena@lip6.fr)
THOMAS Gaël (Gael.Thomas@lip6.fr)

Table des matières

TD 1 : Du saut à la commutation . . . 2
TD 2 : Introduction au noyau Unix . . . 5
TD 3 : La synchronisation des processus . . . 7
Fichier slp.c . . . 9
TD 4 : Les signaux . . . 11
Fichier sig.c . . . 13
TD 5 : La gestion du temps et l'ordonnancement des processus . . . 16
Fichier clock2.c . . . 17
TD 6 : Commutation de processus . . . 21
Fichier swtch.c . . . 23
TD 7 : Création et terminaison de processus . . . 25
Fichier fork.c . . . 26
Fichier exit.c . . . 29
TD 8 : Le buffer cache . . . 30
Fichier bio2.c . . . 31
TD 9 : Répresantation Interne des Fichiers . . . 38
Fichier iget.c . . . 40
Fichier namei.c . . . 43
TD 10 : Structure des fichiers
Traduction d'adresse / Gestion de l'espace libre sur disque . . . 47
Fichier subr.c . . . 49
Fichier alloc.c . . . 52
TME 1 :Construction d'un ordonnanceur dans l'espace utilisateur . . . 55
TME 2 :Implementation et utilisation de primitives de synchronisation . . . 59
TME 3 :Implementation du mécanisme de timeout . . . 65
TME 4 :Implementation des fonctions du Buffer Cache . . . 67

Annexes : 69
Fichier buf.h . . . 70
Fichier callo.h . . . 72
Fichier conf.h . . . 73
Fichier fblk.h . . . 74
Fichier filsys.h . . . 75
Fichier inode.h . . . 76
Fichier mount.h . . . 77
Fichier param.h . . . 78
Fichier proc.h . . . 79
Fichier signal.h . . . 81
Fichier text.h . . . 82
Fichier tty.h . . . 83
Fichier types.h . . . 84
Fichier user.h . . . 85
Fichier var.h . . . 87

TD 1 - Du saut à la commutation

Vous allez étudier le code d'un Unix version 6 à 7 pendant le module Noyau. Vous pouvez trouver les sources qui n'ont pas été reportées dans ce fascicule aux URL suivantes :

— La version 6 en pdf : v6.cuzuco.com/v6.pdf ;
— La version 6 interactive : http ://unix-tree.huihoo.org/V6/usr/sys/ken/index.html ;
— La version 7 interactive : http ://unix-tree.huihoo.org/V7/usr/sys/sys/index.html.

But

Ce TD a pour but de vous introduire la notion de commutation.

Prérequis

Vous devez être familiarisé avec la programmation en C.

Question 1

Que sont les segments de données, de code et de pile ? Comment le CPU y accède ? Décrivez dans le petit programme suivant l'état de la mémoire au point indiqué.

```
int x = 37;

void f() {
int i = 17;

  printf("Hello: %d\n", i);

  i = 45;

  // point d'exécution
}

void main() {
  f();
}
```

Question 2

Les fonctions `setjmp` et `longjmp` sauvegardent leur environement dans une structure opaque `jmp_buf`. L'environment est constitué des registres. Setjmp sauvegarde l'environnement et renvoie 0. Lors

du longjmp, l'exécution reprend au point du setjmp avec un code de retour égal au second paramètre du longjmp.

```
int  setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

Que fait le code suivant ? A quel mécanisme ce code vous fait-il penser ?

```
jmp_buf buf;

void f() {
  if(write(...) == -1)
    longjmp(buf, 1);
}

void g() {
  if(setjmp(buf) == 0) // on vient de sauvegarder l'environment
    f();
  else                 // retour = 1 => on vient du longjmp
    perror("Erreur");
}
```

Question 3

Que fait le programme suivant ? Déroulez l'exécution sur 5 secondes. Qu'est ce que ce programme va afficher ?

```
1   jmp_buf buff, bufg;

2   void f() {
3      int n = 0;

4      while(1) {
5        printf("Execute f: %d\n", n++);
6        sleep(1);
7        if(setjmp(buff) == 0)
8          longjmp(bufg, 1);
9      }
10  }

11  void g() {
12     int m = 1000;

13     while(1) {
14       printf("Execute g: %d\n", m++);
15       sleep(1);
16       if(setjmp(bufg) == 0)
17         longjmp(buff, 1);
18    }
19  }

20   void main() {
21     if(setjmp(bufg) == 0)
22       f();
```

```
23     else
24       g();
25   }
```

Question 4

Modifier le programme pour préserver la pile à chaque commutation.

Bref rappel sur les signaux

Question 5

Ecrivez un programme qui affiche la chaîne de caractère "Bonjour" toutes les secondes. Pour réaliser ce programme, vous utiliserez la fonction `int alarm(int nb_sec)` qui envoie un signal `SIGALRM` toutes les `nb_sec` secondes au processus.

On vous rappel qu'un signal est un message envoyé par un processus ou par le noyau à un processus. Un processus enregistre des comportements associés au signal : le comportement ignorer le signal (`SIG_IGN`), le comportement exécuter le comportement par défaut (`SIG_DFL`) qui est bien souvent d'arreter le processus ou un comportement personnalisé. Dans ce cas, il s'agit d'une fonction du processus. Un comportement personnalisé est souvent appelé gestionnaire ou handler en anglais. Pour associer un gestionnaire à un signal, on peut utiliser la fonction C `signal(int no, void (*handler)(int))`. Elle associe la fonction `handler` au signal `no`. Vous verrez de façon beaucoup plus appronfondie les signaux dans le module POSIX et dans le suite de ce module.

TD 2 - Introduction au noyau Unix

But

Le but de ce TD est d'introduire les principaux concepts du code système et son application dans les systèmes Unix version 6 et version 7. Ceux-ci tournaient sur des machines appelées des PDP-11 qui étaient parmi les premières machines à fournir une visualisation de la mémoire. Ce TD 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

Quelles sont les différences entre exécution en mode *usager* et en mode *système* ? Pourquoi y-a-t-il ces différences ? Comment le mode d'exécution est-il géré par le PDP-11 ?

Question 2

Rappelez ce qu'est ce qu'un appel système.

Question 3

Rappelez ce qu'est une interruption matériel.

Question 4

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

Question 5

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

Question 6

Qu'est ce qu'est la zone U ?

Question 7

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

Question 8

Qu'est-ce que le BIOS ? Comment se passe l'initialisation du système au démarrage de l'ordinateur ?

TD 3 - La synchronisation des processus

But

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

Prérequis

Vous devez connaître la différence entre mode utilisateur et mode système et être famillarisés avec la table des processus.

Question 1

Rappelez le rôle des fonctions `sleep` et `wakeup`.

Question 2

En utilisant les fonctions `sleep` et `wakeup` vues en cours, essayez de programmer une fonction `flock` qui verrouille un fichier en lecture/écriture (donc permet un accès exclusif au fichier). Si le fichier est déjà verrouillé, 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 verrouiller 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).

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 ?

Question 3

Quelle sont les opérations effectuées par les fonctions `spl` et `gpl` ? Quel en est l'effet ?

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

Question 4

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 de la variable `runrun` ?

Question 5

Qu'est-ce que le swapper ? Quel est le rôle des variables `runin` et `runout`.

Question 6

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

Question 7

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

Fichier slp.c

```
1  #include "../sys/param.h"
2  #include "../sys/types.h"
3  #include "../sys/user.h"
4  #include "../sys/proc.h"
5
6
7  char    runin, runout, runrun;
8
9
10 /*
11  * Give up the processor till a wakeup occurs
12  * on chan, at which time the process
13  * enters the scheduling queue at priority pri.
14  * The most important effect of pri is that when
15  * pri<=PZERO a signal cannot disturb the sleep;
16  * if pri>PZERO signals will be processed.
17  * Callers of this routine must be prepared for
18  * premature return, and check that the reason for
19  * sleeping has gone away.
20  */
21
22
23 sleep(chan, pri)
24 caddr_t chan;
25 {
26     register struct proc *rp = u.u_procp;
27     register s, op;
28
29
30     if (pri > PZERO) {
31         if(issig())
32             goto psig;
33         spl(CLINHB);
34         rp->p_wchan = chan;
35         rp->p_stat = SSLEEP;
36         rp->p_pri = pri;
37         spl(NORMAL);
38         if(runin != 0) {
39             runin = 0;
40             wakeup(&runin);
41         }
42         swtch();
43         if(issig())
44             goto psig;
45     } else {
46         spl(CLINHB);
47         rp->p_wchan = chan;
48         rp->p_stat = SSLEEP;
49         rp->p_pri = pri;
50         spl(NORMAL);
51         swtch();
52     }
53     return;
54
```

```
55
56      /*
57       * If priority was low (>PZERO) and
58       * there has been a signal,
59       * execute non-local goto to
60       * the qsav location.
61       */
62   psig:
63      aretu(u.u_qsav);
64  }
65
66
67  /*
68   * Wake up all processes sleeping on chan.
69   */
70  wakeup(chan)
71  register caddr_t chan;
72  {
73      register struct proc *p;
74      register c, i ;
75
76
77      c = chan;
78      p = &proc[0];
79      i = NPROC;
80      do {
81          if(p->p_wchan == c) {
82              setrun(p);
83          }
84          p++;
85      } while(--i);
86  }
87
88
89  /*
90   * Set the process running;
91   * arrange for it to be swapped in if necessary.
92   */
93  setrun(p)
94  register struct proc *p;
95  {
96      p->p_wchan = 0;
97      p->p_stat = SRUN;
98      if(p->p_pri < u.u_procp->p_pri)
99          runrun++;
100     if(runout != 0 && (p->p_flag&SLOAD) == 0) {
101         runout = 0;
102         wakeup(&runout);
103     }
104 }
```

TD 4 - 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, en particulier, les signaux ne peuvent pas être masqués. De plus, la version étudiée ne propose pas le signal SIGCHILD.

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

Quelle est la différence entre un signal ignoré et un signal masqué ?

Question 2

Quels sont les rôles des variables p->p_sig et u.u_signal ? Pourquoi quelle raison u.u_signal est dans la zone swappable ?

Question 3

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

Question 4

Quels sont les rôles des fonctions `kill`, `psignal`, `issig`, `psig`, `fsig`, `sendsig` et `ssig` ?

Question 5

Rappelez les opérations bit-à-bit en C. Que fait la fonction *fsig* (lignes 170-186) ?

Question 6

Expliquez comment se déroule l'émission d'un signal avec `psignal`. Expliquez ce qui se passe lorsque le processus récepteur est en attente d'une ressource système ?

Question 7

Expliquez quand et comment se déroule la réception d'un signal.

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

Comment est réalisé l'appel de la fonction (handler) spécifiée par l'utilisateur ?

Question 8

Expliquez le code de `sendsig()`.

Fichier sig.c

```
1
2  /*
3   * signal system call
4   */
5  ssig()
6  {
7      register a;
8
9      a = u.u_arg[0];
10     if(a<=0 || a>=NSIG || a ==SIGKIL) {
11         u.u_error = EINVAL;
12         return;
13     }
14     u.u_ar0[R0] = u.u_signal[a];
15     u.u_signal[a] = u.u_arg[1];
16     u.u_procp->p_sig &= ~(1<<(a-1));
17 }
18
19 /*
20  * kill system call
21  */
22 kill()
23 {
24     register struct proc *p, *q;
25     register a;
26     int f;
27
28     f = 0;
29     a = u.u_arg[1];
30     q = u.u_procp;
31     for(p = &proc[0]; p < &proc[NPROC]; p++) {
32         if(p->p_stat == NULL)
33             continue;
34         if(a != 0 && p->p_pid != a)
35             continue;
36         if(a == 0 && (p->p_ttyp != q->p_ttyp || p <= &proc[1]))
37             continue;
38         if(u.u_uid != 0 && u.u_uid != p->p_uid)
39             continue;
40         f++;
41         psignal(p, u.u_arg[0]);
42     }
43     if(f == 0)
44         u.u_error = ESRCH;
45 }
46
47 /*
48  * Send the specified signal to
49  * the specified process.
50  */
51 psignal(p, sig)
52 register struct proc *p;
53 register sig;
54 {
```

```
55
56      if((unsigned)sig >= NSIG)
57          return;
58      if(sig)
59          p->p_sig |= 1<<(sig-1);
60      if(p->p_stat == SSLEEP && p->p_pri > PZERO)
61          setrun(p);
62  }
63
64  /*
65   * Returns true if the current
66   * process has a signal to process.
67   * This is asked at least once
68   * each time a process enters the
69   * system.
70   * A signal does not do anything
71   * directly to a process; it sets
72   * a flag that asks the process to
73   * do something to itself.
74   */
75  issig()
76  {
77      register n;
78      register struct proc *p;
79
80      p = u.u_procp;
81      while(p->p_sig) {
82          n = fsig(p);
83          if((u.u_signal[n]&1) == 0)
84              return(n);
85          p->p_sig &= ~(1<<(n-1));
86      }
87      return(0);
88  }
89
90  /*
91   * Perform the action specified by
92   * the current signal.
93   * The usual sequence is:
94   *if(issig())
95   *psig();
96   */
97  psig()
98  {
99      register n, p;
100     register struct proc *rp;
101
102     rp = u.u_procp;
103     n = fsig(rp);
104     if (n==0)
105         return;
106     rp->p_sig &= ~(1<<(n-1));
107     if((p=u.u_signal[n]) != 0) {
108         u.u_error = 0;
109         u.u_signal[n] = 0;
110         sendsig(p, n);
111         return;
112     }
113     switch(n) {
```

```
114
115             case SIGQUIT:
116             case SIGINS:
117             case SIGTRC:
118             case SIGIOT:
119             case SIGEMT:
120             case SIGFPT:
121             case SIGBUS:
122             case SIGSEG:
123             case SIGSYS:
124                 if(core())
125                     n += 0200;
126         }
127         exit(n);
128     }
129
130     /*
131      * find the signal in bit-position
132      * representation in p_sig.
133      */
134     fsig(p)
135     struct proc *p;
136     {
137         register n, i;
138
139         n = p->p_sig;
140         for(i=1; i<NSIG; i++) {
141             if(n & 1)
142                 return(i);
143             n >>= 1;
144         }
145         return(0);
146     }
147
148     /* adapted from the version 6 */
149     sendsig(void *handler, int num) {
150         sp = u.u_ar0[SP] - 2;
151         grow(n);
152         u.u_ar0[SP] = sp;
153         suword(sp,    u.u_ar0[PC]); /* sp[0] = PC  */
154         u.u_ar0[PC] = handler;
155     }
```

TD 5 - 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 Unixet d'étudier un exemple de routine de traitement d'interruptions.

Prérequis

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

Question 1

Récapitulez les différentes notions du temps présentes dans le système.

Question 2

Que sont les *timeouts* ? A quoi servent-ils ? Quelle est la structure de données utilisée pour implémenter ces *timeouts* ?

Question 3

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

Question 4

Expliquez la structure de la routine d'interruption horloge et l'enchaînement des diverses fonctions. Décrivez l'algorithme de la fonction *clock* et le fonction *realtime*.

Question 5

Programmez la fonction *restart*. Elle doit appeler les fonctions dont les timeouts sont arrivés à expiration. Après cela, elle doit décaler les entrées corrrespondantes au timeouts en cours vers le début du vecteur de *callout*.

Question 6

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

Fichier clock2.c

```
1  #include "../sys/param.h"
2  #include "../sys/conf.h"
3  #include "../sys/proc.h"
4  #include "../sys/user.h"
5  #include "../sys/var.h"
6
7  /*
8   * clock is called straight from
9   * real time clock interrupt.
10  * Functions:
11  *       implement callouts
12  *       maintain user/system times
13  *       profile user proc's and kernel
14  *       lightning bolt wakeup.
15  */
16
17
18 clock()
19 {
20      extern int iaflags, idleflag;
21      register struct callo *p1;
22      register int *pc;
23
24      if (v.ve_callout[0].c_func != 0) {
25          p1 = &v.ve_callout[0];
26          while (p1->c_time<=0 && p1->c_func!=0)
27              p1++;
28          p1->c_time--;
29      }
30      if (! idleflag)
31          {
32              if(u.u_procp->p_cpu < 80)
33                  u.u_procp->p_cpu++;
34          }
35
36
37      if (user_mode ()) {
38          u.u_utime++;
39      } else {
40          if (! idleflag)
41              u.u_stime++;
42      }
43
44
45      if ((v.ve_callout[0].c_func!=0 && v.ve_callout[0].c_time<=0) )
46          iaflags |= CALOUT;
47      if (++lbolt >= HZ)
48          iaflags |= WAKEUP;
49 }
50
51
52 realtime()
53 {
54      register struct proc *pp;
```

```
55
56
57      lbolt -= HZ;
58      time++;
59
60
61      /* force a switch every second */
62      runrun++;
63      wakeup(&lbolt);
64      for (pp = &v.ve_proc[0]; pp < proc_end; pp++)
65          if (pp->p_stat) {
66              if (pp->p_time <= 127)
67                  pp->p_time++;
68
69
70              if(pp->p_clktim)
71                  if(--pp->p_clktim == 0)
72                      psignal(pp, SIGCLK);
73              /*
74               * Update CPU Usage info:
75               */
76              pp->p_cpu >>= 1;
77              if (pp->p_pri >= (PUSER-NZERO))
78                  setpri(pp);
79
80
81          }
82      if (runin != 0) {
83          runin = 0;
84          wakeup((caddr_t)&runin);
85      }
86  }
87
88
89
90  /*
91   * timeout is called to arrange that
92   * fun(arg) is called in tim/HZ seconds.
93   * An entry is sorted into the v.ve_callout
94   * structure. The time in each structure
95   * entry is the number of HZ's more
96   * than the previous entry.
97   * in this way, decrementing the
98   * first entry has the effect of
99   * updating all entries.
100  */
101 timeout(fun, arg, tim)
102 int (*fun)();
103 caddr_t arg;
104 int tim;
105 {
106     register struct callo *p1, *p2;
107     int ps;
108     register int t;
109
110
111
112
113     t = tim;
```

```
114         p1 = &v.ve_callout[0];
115         ps = gpl();
116         spl(CLINHB);
117         while(p1->c_func != 0 && p1->c_time <= t) {
118             t -= p1->c_time;
119             p1++;
120         }
121         p1->c_time -= t;
122         p2 = p1;
123         while(p2->c_func != 0)
124             p2++;
125         if(p2 == &v.ve_callout[v.v_callout-2])  {
126             spl(ps);
127             panic("no callout space");
128         }
129         while(p2 >= p1) {
130             (p2+1)->c_time = p2->c_time;
131             (p2+1)->c_func = p2->c_func;
132             (p2+1)->c_arg = p2->c_arg;
133             p2--;
134         }
135         p1->c_time = t;
136         p1->c_func = fun;
137         p1->c_arg = arg;
138         spl(ps);
139 }
140
141
142 restart()
143 {
144         struct callo *p1;
145         struct callo *p2;
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163 }
164
165
166 #define PDELAY  (PZERO-1)
167 delay(ticks)
168 {
169         extern wakeup();
170
171
172         if (ticks<=0)
```

```
173             return;
174         timeout(wakeup, (caddr_t)u.u_procp+1, ticks);
175         sleep((caddr_t)u.u_procp+1, PDELAY);
176     }
177
178     /*
179      * Set user priority.
180      * The rescheduling flag (runrun)
181      * is set if the priority is higher
182      * than the currently running process.
183      */
184     setpri(up)
185     {
186         register *pp, p;
187
188         pp = up;
189         p = (pp->p_cpu & 0377)/16;
190         p =+ PUSER + pp->p_nice;
191         if(p > 127)
192             p = 127;
193         if(p > u.u_procp->p_pri)
194             runrun++;
195         pp->p_pri = p;
196     }
```

TD 6 - Commutation de processus

But

Le but de ce TD est de comprendre comment le noyau gère la commutation de processus, les appels systèmes et les interruptions.

Prérequis

Vous devez connaître la différence entre mode utilisateur et mode système, être familiarisés avec la table des processus et connaître le rôle des variables `runrun`, `runin` et `runout`.

Commutation

Question 1

En sachant que le PDP-11 gère une mémoire segmentée et en vous aidant des figures 1 et figures 2, expliquez comment le PDP-11 gère la mémoire.

Question 2

Quels sont les segments d'un processus ? Quels sont les segments du noyau ? Comment est organisée physiquement la mémoire associée ?

Question 3

Comment le noyau commute ?

Question 4

Expliquez l'algorithme de la fonction *swtch* de commutation dans le noyau.

Entrée et sortie du système

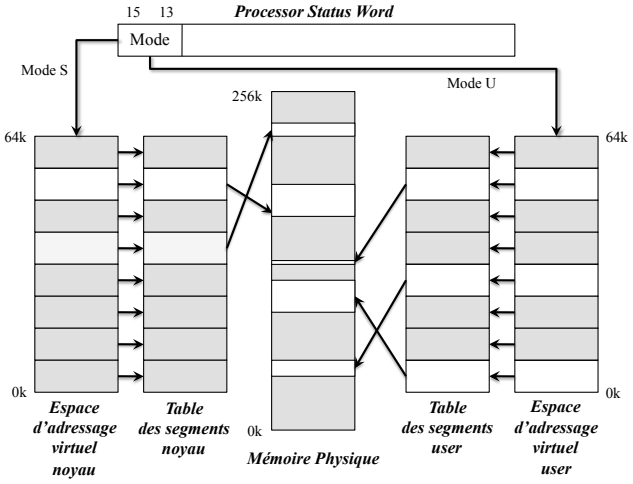

FIGURE 1 – Tables des segmentation. Les zones grisées ne sont pas mappées ou allouées.

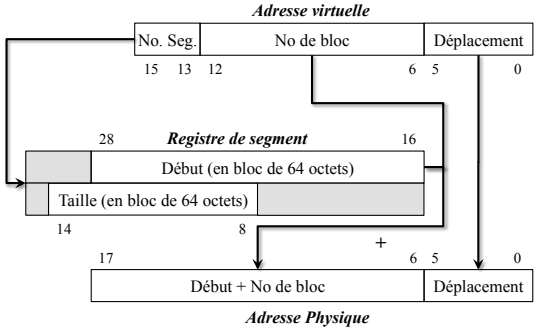

FIGURE 2 – Résolution d'une adresse virtuelle.

Question 5

Que sont une trap, une exception et une interruption ? Décrivez la table des interruptions.

Question 6

Expliquez les actions effectuées lors d'un appel système.

Question 7

Expliquez les actions effectuées lors d'une interruption.

Fichier swtch.c

```
1  /*
2   * This routine is called to reschedule the CPU.
3   * if the calling process is not in RUN state,
4   * arrangements for it to restart must have
5   * been made elsewhere, usually by calling via sleep.
6   */
7  swtch()
8  {
9      static struct proc *p;
10     register i, n;
11     register struct proc *rp;
12
13     if(p == NULL)
14         p = &proc[0];
15     /*
16      * Remember stack of caller
17      */
18     savu(u.u_rsav);
19     /*
20      * Switch to scheduler's stack
21      */
22     retu(proc[0].p_addr);
23
24 loop:
25     runrun = 0;
26     rp = p;
27     p = NULL;
28     n = 128;
29     /*
30      * Search for highest-priority runnable process
31      */
32     i = NPROC;
33     do {
34         rp++;
35         if(rp >= &proc[NPROC])
36             rp = &proc[0];
37         if(rp->p_stat==SRUN && (rp->p_flag&SLOAD)!=0) {
38             if(rp->p_pri < n) {
39                 p = rp;
40                 n = rp->p_pri;
41             }
42         }
43     } while(--i);
44     /*
45      * If no process is runnable, idle.
46      */
47     if(p == NULL) {
48         p = rp;
49         idle();
50         goto loop;
51     }
52     rp = p;
53     /*
54      * Switch to stack of the new process and set up
```

```
55          * his segmentation registers.
56          */
57         retu(rp->p_addr);
58         sureg();
59         /*
60          * If the new process paused because it was
61          * swapped out, set the stack level to the last call
62          * to savu(u_ssav).  This means that the return
63          * which is executed immediately after the call to aretu
64          * actually returns from the last routine which did
65          * the savu.
66          *
67          * You are not expected to understand this.
68          */
69         if(rp->p_flag&SSWAP) {
70             rp->p_flag =& ~SSWAP;
71             aretu(u.u_ssav);
72         }
73         /*
74          * The value returned here has many subtle implications.
75          * See the newproc comments.
76          */
77         return(1);
78     }
```

TD 7 - 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 analysant le code de *newproc* :
— Quelles sont les resources (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 ?

Question 2

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

Terminaison de processus - exit

Question 3

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

Question 4

Donnez l'algorithme du code d' *exit*. Quelle est la signification (l'utilité) du wakeup de la lignes 35 ?

Question 5

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.

Fichier fork.c

```
1
2  fork ()
3  {
4      register proc *p1; *p2;
5
6      p1 = u.u_procp;
7
8      for (p2 = &proc[0]; p2 < &proc[NPROC}; p2++)
9          if (p2->p_stat == NULL)
10             goto found;
11     u_u_error = EAGAIN;
12     goto out;
13
14  found:
15     if (newproc()) {
16         u.u_ar0[R0] = 0;
17         u.u_cstime [0] = 0;
18         u.u_cstime [1] = 0;
19         u.u_stime = 0;
20         u.u_cutime [0] = 0;
21         u.u_cutime [1] = 0;
22         u.u_utime  = 0;
23         return;
24     }
25
26     u.u_aro[R0] = p2->p_pid;
27  out:
28     u.u_ar0[R7] = +2;
29 }
30
31
32 /* Create a new process (the internal version of fork )
33    The new process returns 1 in the new process.
34    The essential fact is that the new process is created in such
35    a way that it appears to have started executing in the
36    same call to newproc as the parent,but in fact the code runs is that of switch.
37    The subtle implication of the returned value of switch is that this is
38    the value that newproc's caller in the new process sees.*/
39
40 newproc ()
41 int a1, a2;
42 struct proc *p,*up;
43 register struct proc *rpp;
44 register *rip, n;
45 {
46     p = NULL;
47
48     /* First, just locate a slot for a process and copy the useful
49        info from this process int it. The panic 'cannot happen' because fork has
50        already checked for the existance of a slot. */
51
52  retry:
53     mpid++;
54     if (mpid < 0) {
```

```
55          mpid =0;
56          goto retry;
57      }
58      for (rpp = &proc[0]; rpp < &proc [NPROC]; rpp++) {
59          if (rpp ->p_stat == NULL && p== NULL)
60              p = rpp;
61          if (rpp->p_pid == mpid)
62              goto retry;
63      }
64
65      if ((rpp = p)== NULL )
66          panic ("no procs");
67
68      /* make proc entry for new proc */
69
70      rip = u.u_proc;
71      up =rip;
72
73      rpp->p_stat = SRUN;
74      rpp->p_flag = SLOAD;
75      rpp->p_uid = rip->p_uid;
76      rpp->p_ttyp = rip ->p_ttyp;
77      rpp->p_nice= rip ->p_nice;
78      rpp->p_textp = rip ->p_textp;
79      rpp->p_pid = mpid;
80      rpp->p_ppid = rip ->p_pid;
81      rpp ->p_time = 0;
82
83      /* make duplicate entries where needed */
84
85      for (rip = &u.u_ofile[0]; rip <  &u.u_ofile[NOFILE];)
86          if ((rpp = *rip++) !=NULL)
87              rpp->f_count ++;
88
89      if ((rpp = up->p_textp) != NULL) {
90          rpp ->x_count++;
91          rpp ->x_ccount ++;
92      }
93
94      u.u_cdir ->i_count++;
95
96      /* Partially simulate the environment of the new process so that
97         when it is actaully  created (by copying) it will look right */
98
99      savu (u.u_rsav);
100     rpp =p;
101     u.u_procp = rpp;
102     rip = up;
103     n = rip -> p_size;
104     a1 = rip ->p_addr;
105     rpp -> p_size =n;
106     a2 =malloc (coremap,n);
107
108     /* if there is not enough memory for the new process,
109        swap ou the current process to generate the copy */
110
111     if (a2 == NULL) {
112         rip ->p_stat = SIDL;
113         rpp-> p_addr = a1;
```

```
114             savu (u.u_ssav);
115             xswap (rpp,0,0);
116             rpp-> p_flag = SSWAP;
117             rip-> p_stat = SRUN;
118         }
119         else {
120             /* there is memory, so just copy */
121 
122             rpp-> p_addr = a2;
123             while (n--)
124                 copyseg (a1++, a2++);
125         }
126 
127         u.u_procp = rip;
128         return (0);
129     }
```

Fichier exit.c

```
1  exit()
2  {
3      register int *a, b;
4      register struct proc *p;
5
6      u.u_procp->p_flags &= ~STRC;
7      for (a = &u.u_signal[0]; a < &u.u_signal[NSIG];)
8          *a++ = 1;
9      for (a = &u.u_ofile[0]; a < &u.u_ofile[NOFILE]; a++)
10         if (b = *a) {
11             *a = NULL;
12             closef(b);
13         }
14
15
16     iput(u.u_cdir);
17
18
19     b = malloc(swapmap, 1);
20     if (b == NULL)
21         panic("out of swap");
22
23     p = getblk(swapdev,b);
24     bcopy(&u, p->b_addr, 256);
25     bwrite(p);
26
27
28     a = u.u_procp;
29     mfree(coremap, a->p_size, a->p_addr);
30     a->p_stat = SZOMB;
31
32 loop:
33     for(p = &proc[0]; p < &proc[NPROC]; p++)
34         if (a->p_ppid == p->p_pid) {
35             wakeup(p);
36             for(p = &proc[0]; p < &proc[NPROC]; p++)
37             if (a->p_pid == p->p_ppid) {
38                     p->p_ppid = 1;
39                     if (p->p_stat == SSTOP)
40                         setrun(p);
41                 }
42             swtch();
43             /* no return */
44         }
45
46
47     a->p_ppid = 1;
48     goto loop;
49 }
```

TD 8 - 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 ?

Question 2

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

Question 3

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

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.

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*.

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`.

Question 7

Question 8

Que signifie l'état `B_DONE` ? Expliquez comment est effectué le contrôle de la fin de l'entrée / sortie. Décrivez le fonctionnement de la fonction iodone().

Question 9

Décrivez l'algorithme de `getblk`.

Question 10

Complétez le corps de la fonction *brelse*, qui libère un tampon quand le noyau a fini de l'utiliser. La fonction doit réveiller les processus qui se sont endormis parce que le tampon était occupé et ceux qui se sont endormis parce qu'aucun tampon ne restait dans la liste de tampons libres. La fonction doit alors placer le tampon à la fin de la liste de tampons libres, à moins qu'une erreur d'entrée-sortie ne se soit produite. N'oubliez pas que la liste de blocs libres est une ressource critique et doit être accédée de façon exclusive (masquage/démasquage d'interruption).

Question 11

Complétez le corps de la fonction *bread* qui effectue la lecture d'un bloc disque. Cette fonction doit utiliser la fonction *getblk* pour rechercher le block dans le buffer cache. S'il y est, le système le lui retourne immédiatement sans le lire physiquement du disque. Sinon, *bread* doit appeler la fonction du périphérique disque qui lance la lecture d'un bloc. Dans ce cas, la fonction devra endormir le processus qui l'a appelée, qui sera réveillé par l'interruption disque.

Expliquez maintenant le code de la fonction *breada*, qui lit deux blocs, dont le deuxième de façon asynchrone. Quel est le but d'offrir une telle fonction ?

Question 12

Complétez le corps de la fonction *bwrite* qui effectue l'écriture d'un bloc disque. La fonction indique au périphérique du disque qu'il y a un tampon dont le contenu doit être enregistré sur le disque. Si l'écriture est synchrone, le processus appelant s'endort en attendant la fin de l'écriture. Puis il libère le bloc quand il est réveillé. Si l'écriture est asynchrone, la fonction lance l'écriture mais n'attend pas sa fin.

Expliquez le code de la fonction *bdwrite* et *bawrite*.

Fichier bio2.c

```
1  #include "../sys/buf.h"
2  #include "../sys/param.h"
3  #include "../sys/types.h"
4
5
6  /*
7   * The following several routines allocate and free
8   * buffers with various side effects.  In general the
9   * arguments to an allocate routine are a device and
10  * a block number, and the value is a pointer to
11  * to the buffer header; the buffer is marked "busy"
12  * so that no one else can touch it.  If the block was
13  * already in core, no I/O need be done; if it is
14  * already busy, the process waits until it becomes free.
15  * The following routines allocate a buffer:
16  *      getblk
17  *      bread
18  *      breada
19  * Eventually the buffer must be released, possibly with the
20  * side effect of writing it out, by using one of
21  *      bwrite
22  *      bdwrite
23  *      bawrite
24  *      brelse
25  */
26
27
28 /*
29  * Unlink a buffer from the available list and mark it busy.
30  * (internal interface)
31  */
32 notavail(bp)
33 {
34     register s;
35
36
37     s = gpl();
38     spl(BDINHB);
39     bp->av_back->av_forw = bp->av_forw;
40     bp->av_forw->av_back = bp->av_back;
41     bp->b_flags |= B_BUSY;
42     bfreelist.b_bcount--;
43     spl(s);
44 }
45
46
47 /*
48  * Read in (if necessary) the block and return a buffer pointer.
49  */
50 struct buf *
51 bread(dev, blkno)
52     dev_t dev;
53 daddr_t blkno;
54 {
```

```
55      register struct buf *bp;
56
57
58
59
60
61
62
63
64
65
66  }
67
68
69  /*
70   * Read in the block, like bread, but also start I/O on the
71   * read-ahead block (which is not allocated to the caller)
72   */
73  struct buf *
74  breada(dev, blkno, rablkno)
75      dev_t dev;
76  daddr_t blkno, rablkno;
77  {
78      register struct buf *bp, *rabp;
79
80
81      bp = NULL;
82      if (!incore(dev, blkno)) {
83          bp = getblk(dev, blkno);
84          if ((bp->b_flags&B_DONE) == 0) {
85              bp->b_flags |= B_READ;
86              bp->b_bcount = BSIZE;
87              (*bdevsw[bmajor(dev)].d_strategy)(bp);
88          }
89      }
90      if (rablkno && bfreelist.b_bcount>1 && !incore(dev, rablkno)) {
91          rabp = getblk(dev, rablkno);
92          if (rabp->b_flags & B_DONE)
93              brelse(rabp);
94          else {
95              rabp->b_flags |= B_READ|B_ASYNC;
96              rabp->b_bcount = BSIZE;
97              (*bdevsw[bmajor(dev)].d_strategy)(rabp);
98          }
99      }
100     if (bp == NULL)
101         return(bread(dev, blkno));
102     iowait(bp);
103     return(bp);
104 }
105
106
107 /*
108  * Write the buffer, waiting for completion.
109  * Then release the buffer.
110  */
111 bwrite(bp)
112 register struct buf *bp;
113 {
```

```
114     register flag;
115
116
117
118
119
120
121
122
123
124
125
126 }
127
128
129 /*
130  * Release the buffer, marking it so that if it is grabbed
131  * for another purpose it will be written out before being
132  * given up (e.g. when writing a partial block where it is
133  * assumed that another write for the same block will soon follow).
134  * This can't be done for magtape, since writes must be done
135  * in the same order as requested.
136  */
137 bdwrite(bp)
138 register struct buf *bp;
139 {
140     bp->b_flags |= B_DELWRI | B_DONE;
141     bp->b_resid = 0;
142     brelse(bp);
143 }
144
145
146 /*
147  * Release the buffer, start I/O on it, but don't wait for completion.
148  */
149 bawrite(bp)
150 register struct buf *bp;
151 {
152     bp->b_flags |= B_ASYNC;
153     bwrite(bp);
154 }
155
156
157 /*
158  * release the buffer, with no I/O implied.
159  */
160 brelse(bp)
161 register struct buf *bp;
162 {
163
164
165
166
167
168
169
170
171
172
```

```
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191 }
192
193
194 /*
195  * See if the block is associated with some buffer
196  * (mainly to avoid getting hung up on a wait in breada)
197  */
198 incore(dev, blkno)
199 register dev_t dev;
200 daddr_t blkno;
201 {
202     register struct buf *bp;
203     register struct buf *dp;
204
205
206     dp = bhash(dev, blkno);
207     for (bp=dp->b_forw; bp != dp; bp = bp->b_forw)
208         if (bp->b_blkno==blkno && bp->b_dev==dev)
209             return(1);
210     return(0);
211 }
212
213
214 /*
215  * Assign a buffer for the given block.  If the appropriate
216  * block is already associated, return it; otherwise search
217  * for the oldest non-busy buffer and reassign it.
218  */
219 struct buf *
220 getblk(dev, blkno)
221     register dev_t dev;
222 daddr_t blkno;
223 {
224     register struct buf *bp;
225     register struct buf *dp;
226
227
228  loop:
229     spl(NORMAL);
230     dp = bhash(dev, blkno);
231     if (dp == NULL)
```

```
232             panic("devtab");
233         for (bp=dp->b_forw; bp != dp; bp = bp->b_forw) {
234             if (bp->b_blkno!=blkno || bp->b_dev!=dev)
235                 continue;
236             spl(BDINHB);
237             if (bp->b_flags&B_BUSY) {
238                 bp->b_flags |= B_WANTED;
239                 sleep((caddr_t)bp, PRIBIO+1);
240                 goto loop;
241             }
242             spl(NORMAL);
243             notavail(bp);
244             return(bp);
245         }
246         spl(BDINHB);
247         if (bfreelist.av_forw == &bfreelist) {
248             bfreelist.b_flags |= B_WANTED;
249             sleep((caddr_t)&bfreelist, PRIBIO+1);
250             goto loop;
251         }
252         spl(NORMAL);
253         bp = bfreelist.av_forw;
254         notavail(bp);
255         if (bp->b_flags & B_DELWRI) {
256             bp->b_flags |= B_ASYNC;
257             bwrite(bp);
258             goto loop;
259         }
260         bp->b_flags = B_BUSY;
261         bp->b_back->b_forw = bp->b_forw;
262         bp->b_forw->b_back = bp->b_back;
263         bp->b_forw = dp->b_forw;
264         bp->b_back = dp;
265         dp->b_forw->b_back = bp;
266         dp->b_forw = bp;
267         bp->b_dev = dev;
268         bp->b_blkno = blkno;
269         return(bp);
270 }
271
272
273 /*
274  * Wait for I/O completion on the buffer; return errors
275  * to the user.
276  */
277 iowait(bp)
278 register struct buf *bp;
279 {
280
281
282         spl(BDINHB);
283         while ((bp->b_flags&B_DONE)==0)
284             sleep((caddr_t)bp, PRIBIO);
285         spl(NORMAL);
286         geterror(bp);
287 }
288
289
290 iodone(bp)
```

```
291 register buf *bp
292 {
293     bp->b_flags |= B_DONE;
294     if((bp->b_flags&B_ASYNC)!=0)
295         brelse(bp);
296     else
297         wakeup((caddr_t)bp);
298 }
```

TD 9 - Répresantation Interne des Fichiers

But

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

Question 1

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

Question 2

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

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 ?

Question 4

Quelles sont les informationts qu'une strucutre *i_ node* contient ?

Question 5

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

Question 6

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

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 ?

Question 8

A quoi sert la fonction *namei* ? Donnez son pseudocode.

Fichier iget.c

```
1  /* Look up an inode by device, inumber.
2     A pointer to a locked inode structure is returned
3     It does not include the mounting of volumes*/
4
5
6  iget (dev,ino)
7  {
8      register struct inode *p;
9      register *ip2;
10     int *ip1;
11
12
13  loop:
14     ip = NULL;
15     for (p = &inode[0]; p <  &inode[NINODE]; p++){
16         if (dev == p->i_dev && ino == p->i_number ) {
17             if ((p->i_flag & ILOCK) !=0) {
18                 p->i_flag =| IWANT;
19                 sleep (p,PINOD);
20                 goto loop;
21             }
22
23
24             p->i_count++;
25             p->i_flag |= ILOCK;
26             return (p);
27         }
28         if (ip== NULL && p->i_count  ==0)
29             ip =p;
30     }
31
32     if ((p = ip) == NULL) {
33         printf ("inode table overflow \n");
34         u.u_error = ENFILE;
35         return (NULL);
36     }
37
38
39     p->i_dev = dev;
40     p->i_number = ino;
41     p->i_flag = ILOCK;
42     p->i_count ++;
43     p->i_lastr =-1;
44     ip = bread (dev, ldiv (ino+31,16) );
45
46
47     /* check I/O errors */
48     if (ip->b_flags & B_ERROR) {
49         brelse (ip);
50         iput(p);
51         return (NULL);
52     }
53
54
```

```
55      ip1 = ip->b_addr + 32*lrem(ino+31,16) ;
56      ip2 = &p->i_mode ;
57
58      while (ip2 < &p->i_addr[8])
59          *ip2++ = *ip1++;
60      blrese (ip);
61      return (p);
62  }
63
64
65
66
67  /* Decrement reference count of an inode structure.
68     On the last reference, write the inode out and
69     if necessary, truncate and deallocate the file */
70
71  iput (p)
72  struct inode *p
73  {
74      register *rp;
75      rp =p;
76
77
78      if (rp ->i_count == 1) {
79          rp ->i_flag |= ILOCK;
80          if (rp ->i_nlink == 0) {
81              itrunc (rp);
82              rp->i_mode =0;
83              ifree (rp->i_dev, rp->i_number );
84
85
86          }
87          iupdate (rp, time);
88          prele(rp);
89
90
91      }
92      rp->i_count --;
93
94  }
95
96
97  ifree (dev,ino)
98  /* free the specified inode on the specified device */
99
100
101 iupdate (p,tm)
102 int *p, int *tm;{
103
104
105     /* check accessed and update flags on an inode
106        structure. If either is on, update the inode
107        with the corresponding dates set to the argument
108        tm */
109 }
110
111
112
113
```

```
114 itrunc (ip)
115 int ip;
116 {
117     /* free all disk blocks associated with the specified
118        inode structure. The blocks of the file are removed
119        in reverse order.
120     */
121 }
122
123
124 Pour la fonction prele(ip), voir le fichier pipe.c */
```

Fichier namei.c

```
1  /*
2   * convert a pathname into a pointer to an inode.
3   * Note that the inode is locked
4   * func = function called to get next char of name
5   *        &uchar if name is in user space
6   * schar is name isin system space
7   *
8   * flag = 0, if name is sought
9   *  1 if name is to be created
10  *  2 if name is to be deleted
11  */
12
13
14
15
16 namei(func, flag)
17 int (* func) ();
18 {
19     register struct inode *dp;
20     register c;
21     register char *cp;
22     int eo, *bp;
23
24
25     /* if name starts with '/', start from root;
26        otherwise start from current dir; */
27
28
29     dp = u.u_cdir;
30     if (( c= (*func ()) == '/')
31         dp = rootdir;
32
33         iget (dp->i_dev, dp->i_number );
34         while (c == '/' )
35             c = (*func) () ;
36         if ( c == '\0' && flag !=0) {
37             u.u_error = ENOENT;
38             goto out;
39         }
40
41
42     cloop:
43         /* Here dp contains pointer to last component matched */
44
45
46         if (u.u_error)
47             goto out;
48
49
50         if (c == '\0')
51             return (dp);
52
53
54         /* if there is another component, dp must be a
```

```
55              directory and must have x permission */
56
57
58          if (( dp ->i_mode & IFMT) != IFDIR ) {
59              u.u_error = ENOTDIR;
60              goto out ;
61          }
62
63
64          if (access (dp, IEXEC ))
65              goto out;
66
67
68          /* gather up name into users' dir buffer */
69
70
71          cp = &u.u_dbuf[0];
72
73          while ( c!= '/' && c != '\0' && u.u_error == 0 )  {
74              if (cp < &u.u_dbuf [DIRSIZ] )
75                  *cp++ =c;
76              c= (*func) () ;
77
78
79          }
80
81
82          while (cp < &u.u_dbuf[DIRSIZ])
83              *cp++ = '\0';
84
85
86          while (c == '/')
87              c= (*func) ();
88
89
90          if (u.u_error)
91              goto out;
92
93
94          /* set up to search a directory */
95          u.u_offset [1] = 0;
96          u.u_offset[0] = 0 ;
97          u.u_segflg=1;
98          eo = 0;
99          u.u_count = ldiv (dp->i_size, DIRSIZ +2 );
100         bp= NULL;
101
102
103     eloop:
104         /* if at the end of the directory, the search failed.
105            Report what is appropriate as per flag */
106         if (u.u_count == 0) {
107             if (bp != NULL)
108                 brelse (bp);
109
110             if (flag == 1 && c == '/0' ) {
111                 if (access (dp, IWRITE)
112                     goto out;
113
```

```
114                     u_u_pdir = dp;
115                     if (eo)
116                         u.u_offset [1] = eo - DIRSIZE -2;
117                     else
118                         dp ->i_flag =| IUPD;
119                     return (NULL);
120                     }
121                 u.u_error = ENOENT;
122                 goto out;
123             }
124
125
126             /* if offset is on a block-boundary, read the next
127                directory block.  Release previous if it exsts */
128
129
130             if ((u.u_offset [1] & 0777 == 0){
131                     if (bp != NULL )
132                         brelse (bp);
133                     bp = bread (dp-> i_dev, bmap (dp, ldiv (u.u_offset [1], 512 )));
134                 }
135
136
137                 /* Note first empty directory slot in eo
138                    for possible creat. String compare the directory entry and
139                    the current component.  If they do not match, go back to sloop */
140
141
142                 bcopy (bp->b_addr + (u.u_offset[1] & 0777), &u.u_dent,
143                         (DIRSIZE +2)/2 );
144
145
146                 u.u_offset[1] =+ DIRSIZ +2;
147
148
149                 u.u_count --;
150
151
152                 f (u.u_dent.u_ino == 0) {
153                     if (eo == 0)
154                         eo = u.u_offset [1];
155                     goto eloop;
156                 }
157
158
159                 for (cp = &u.u_dbuf[0]; cp < &u.u_dbuf[DIRSIZ]; cp++)
160                     if (*cp != cp[u.u_dent.u_name - u.u_dbuf] )
161                         goto eloop;
162
163
164                 /* here a component matched in a directory.
165                    if there is more pathname, go back to eloop,
166                    otherwise return */
167
168
169                 if (bp != NULL)
170                     brelse (bp);
171                 if (flag == 2 && c == '\0') {
172                     if (access (dp,IWRITE))
```

```
173                         goto out;
174                     return (dp);
175                 }
176 
177 
178                 bp = dp -> i_dev;
179                 iput (dp);
180                 dp = iget (bp,u.u_dent.u_ino);
181 
182 
183                 if (dp == NULL)
184                     return (NULL);
185                 goto cloop;
186 
187 
188             out:
189                 iput (dp);
190                 return (NULL);
191                 }
```

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.

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 ?

Question 3

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 ?

Question 4

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

Question 5

Expliquez la gestion de l'espace libre.

Question 6

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

A quels moments sont-elles appelées ? Donnez l'algorithme des fonctions `alloc` et `free`.

Question 7

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

Fichier subr.c

```
1  #include "../sys/inode.h"
2  #include "../sys/buf.h"
3  #include "../sys/types.h"
4
5
6  /*
7   * Bmap defines the structure of file system storage
8   * by returning the physical block number on a device given the
9   * inode and the logical block number in a file.
10  * When convenient, it also leaves the physical
11  * block number of the next block of the file in rablock
12  * for use in read-ahead.
13  */
14
15
16 daddr_t
17 bmap(ip, bn, rwflg)
18     struct inode *ip;
19 daddr_t bn;
20 int rwflg;
21 {
22     register i;
23     struct buf *bp, *nbp;
24     int j, sh;
25     daddr_t nb, *bap;
26     dev_t dev;
27
28
29     if(bn < 0) {
30         u.u_error = EFBIG;
31         return((daddr_t)0);
32     }
33     dev = ip->i_dev;
34     rablock = 0;
35     /*
36      * blocks 0..NADDR-4 are direct blocks
37      */
38     if(bn < NADDR-3) {
39         i = bn;
40         nb = ip->i_addr[i];
41         if(nb == 0) {
42             if(rwflg==B_READ || (bp = alloc(dev))==NULL)
43                 return((daddr_t)-1);
44             nb = bp->b_blkno;
45             bdwrite(bp);
46             ip->i_addr[i] = nb;
47             ip->i_flag |= IUPD|ICHG;
48         }
49         if(i < NADDR-4)
50             rablock = ip->i_addr[i+1];
51         return(nb);
52     }
53     /*
54      * addresses NADDR-3, NADDR-2, and NADDR-1
```

```
55          * have single, double, triple indirect blocks.
56          * the first step is to determine
57          * how many levels of indirection.
58          */
59         sh = 0;
60         nb = 1;
61         bn -= NADDR-3;
62         for(j=3; j>0; j--) {
63             sh += NSHIFT;
64             nb <<= NSHIFT;
65             if(bn < nb)
66                 break;
67             bn -= nb;
68         }
69         if(j == 0) {
70             u.u_error = EFBIG;
71             return((daddr_t)0);
72         }
73         /*
74          * fetch the address from the inode
75          */
76         nb = ip->i_addr[NADDR-j];
77         if(nb == 0) {
78             if(rwflg==B_READ || (bp = alloc(dev))==NULL)
79                 return((daddr_t)-1);
80             nb = bp->b_blkno;
81             bdwrite(bp);
82             ip->i_addr[NADDR-j] = nb;
83             ip->i_flag |= IUPD|ICHG;
84         }
85         /*
86          * fetch through the indirect blocks
87          */
88         for(; j<=3; j++) {
89             bp = bread(dev, nb);
90             if(bp->b_flags & B_ERROR) {
91                 brelse(bp);
92                 return((daddr_t)0);
93             }
94             bap = bp->b_daddr;
95             sh -= NSHIFT;
96             i = (bn>>sh) & NMASK;
97             nb = bap[i];
98             if(nb == 0) {
99                 if(rwflg==B_READ || (nbp = alloc(dev))==NULL) {
100                    brelse(bp);
101                    return((daddr_t)-1);
102                }
103                nb = nbp->b_blkno;
104                bdwrite(nbp);
105                bap[i] = nb;
106                bdwrite(bp);
107            } else
108                brelse(bp);
109        }
110        /*
111         * calculate read-ahead.
112         */
113        if(i < NINDIR-1)
```

```
114             rablock = bap[i+1];
115         return(nb);
116 }
```

Fichier alloc.c

```
1  #include "../sys/filsys.h"
2  #include "../sys/fblk.h"
3  #include "../sys/buf.h"
4  #include "../sys/inode.h"
5
6
7  typedef struct fblk *FBLKP;
8
9
10 /*
11  * alloc will obtain the next available
12  * free disk block from the free list of
13  * the specified device.
14  * The super block has up to NICFREE remembered
15  * free blocks; the last of these is read to
16  * obtain NICFREE more . . .
17  */
18 struct buf *
19 alloc(dev)
20     dev_t dev;
21 {
22     daddr_t bno;
23     register struct filsys *fp;
24     register struct buf *bp;
25
26
27     fp = getfs(dev);
28     while(fp->s_flock)
29         sleep((caddr_t)&fp->s_flock, PINOD);
30     do {
31         if(fp->s_nfree <= 0)
32             goto nospace;
33         if (fp->s_nfree > NICFREE) {
34             prdev("Bad free count", dev);
35             goto nospace;
36         }
37         bno = fp->s_free[--fp->s_nfree];
38         if(bno == 0)
39             goto nospace;
40     } while (badblock(fp, bno, dev));
41     if(fp->s_nfree <= 0) {
42         fp->s_flock++;
43         bp = bread(dev, bno);
44         if ((bp->b_flags&B_ERROR) == 0) {
45             fp->s_nfree = ((FBLKP)(bp->b_addr))->df_nfree;
46             bcopy((caddr_t)((FBLKP)(bp->b_addr))->df_free,
47                         (caddr_t)fp->s_free, sizeof(fp->s_free));
48         }
49         brelse(bp);
50         fp->s_flock = 0;
51         wakeup((caddr_t)&fp->s_flock);
52         if (fp->s_nfree <=0)
53             goto nospace;
54     }
```

```
55      if(fp->s_nfree <= 0 || fp->s_nfree > NICFREE) {
56          prdev("Bad free count", dev);
57          goto nospace;
58      }
59
60
61      bp = getblk(dev, bno);
62      clrbuf(bp);
63      if(fp->s_tfree) fp->s_tfree--;
64      fp->s_fmod = 1;
65      return(bp);
66
67
68   nospace:
69      fp->s_nfree = 0;
70      fp->s_tfree = 0;
71      prdev("no space", dev);
72      u.u_error = ENOSPC;
73      return(NULL);
74  }
75
76
77  /*
78   * place the specified disk block
79   * back on the free list of the
80   * specified device.
81   */
82
83
84  free(dev, bno)
85  dev_t dev;
86  daddr_t bno;
87  {
88      register struct filsys *fp;
89      register struct buf *bp;
90
91
92      fp = getfs(dev);
93      while(fp->s_flock)
94          sleep((caddr_t)&fp->s_flock, PINOD);
95      if (badblock(fp, bno, dev))
96          return;
97      if(fp->s_nfree <= 0) {
98          fp->s_nfree = 1;
99          fp->s_free[0] = 0;
100     }
101     if(fp->s_nfree >= NICFREE) {
102         fp->s_flock++;
103         bp = getblk(dev, bno);
104         ((FBLKP)(bp->b_addr))->df_nfree = fp->s_nfree;
105         bcopy((caddr_t)fp->s_free,
106                     (caddr_t)((FBLKP)(bp->b_addr))->df_free,
107                     sizeof(fp->s_free));
108         fp->s_nfree = 0;
109         bwrite(bp);
110         fp->s_flock = 0;
111         wakeup((caddr_t)&fp->s_flock);
112     }
113     fp->s_free[fp->s_nfree++] = bno;
```

```
114         fp->s_tfree++;
115         fp->s_fmod = 1;
116 }
117 
118 
119 /*
120  * Check that a block number is in the
121  * range between the I list and the size
122  * of the device.
123  * This is used mainly to check that a
124  * garbage file system has not been mounted.
125  *
126  * bad block on dev x/y -- not in range
127  */
128 
129 
130 badblock(fp, bn, dev)
131 register struct filsys *fp;
132 daddr_t bn;
133 dev_t dev;
134 {
135 
136 
137         if (bn < fp->s_isize || bn >= fp->s_fsize) {
138                 prdev("bad block", dev);
139                 return(1);
140         }
141         return(0);
142 }
```

TME 1 - Construction d'un ordonnanceur dans l'espace utilisateur

But

Le but de ce TP et de programmer une bibliothèque d'ordonnancement dans l'espace utilisateur.

Prérequis

Sujet

Dans ce TP, vous allez construire une bibliothèque d'ordonnancement, c'est à dire une bibliothèque capable d'exécuter plusieurs processus en temps partagé. Cette bibliothèque doit offrir les fonctions suivantes :

- `void init_sched()` : Initialise la bibliothèque.
- `void new_proc(void (*f)(int), int arg)` : enregistre un nouveau processus à exécuter. Ce processus devra être démarré en exécutant `f(arg)`.
- `void start_sched()` : cette fonction doit débuter l'ordonnancement.

Question 1

La première étape de ce TP est de définir la table des processus. Chaque entrée dans le table est définie par une structure `Tproc`. Quels sont les champs nécessaires dans cette structure. Une variable globale `elu` indiquera en permanence quel est l'index du processus actuellement élu dans cette table.

Indication

Quand vous allez commuter un processus, vous aurez besoin de sauvegarder ses registres (son environnement) et sa pile. N'oubliez pas qu'une case de la table ne contient pas forcément un processus.

Question 2

La fonction init_sched doit calculer le haut de la pile d'exécution pour pouvoir la sauvegarder lors d'une commutation. Proposez une méthode pour calculer le haut de la pile. Pour quelle raison la fonction init_sched doit être définie comme une macro C ? Cette macro placera dans une variable global appelée `char *top_stack` cette adresse. Écrivez et testez cette macro.

Question 3

Les fonctions `setjmp` et `longjmp` ne sont pas suffisantes pour faire de l'ordonnancement. En effet, ces fonctions ne sauvegardant pas la pile, les processus partageraient la pile, ce qui conduirait à des erreurs. On vous demande donc d'écrire deux fonctions `mysetjmp` et `mylongjmp` qui, en plus de sauvegarder/restaurer le contexte, sauvegardent/restaurent la pile d'exécution de l'appelant jusqu'au `main`.

La fonction `int mysetjmp(int idx)` prend en paramètre un index dans la table des processus. Elle sauvegarde le contexte du processus appelant à cet index, c'est-à-dire qu'elle sauvegarde la pile du processus appelant et utilise `setjmp` pour sauvegarder les registres. Si le retour du `setjmp` est zéro, c'est que le processus vient d'être sauvegardé. Dans ce cas, la fonction setjmp renvoie zéro. Si le retour du `setjmp` est différent de zéro, c'est que le processus vient d'être restauré suite à un longjmp. Dans ce cas, la fonction `mysetjmp` restaure la pile du processus nouvellement élu (indiqué par la variable globale `elu`) et renvoie la valeur 1.

La fonction `void mylongjmp(int idx)` prend en paramètre un index dans la table des processus. Elle positionne le nouveau processus élu à `idx` et restaure les registres de celui-ci. La restauration de la pile aura lieu dans `mysetjmp`.

Indication

La fonction `mysetjmp` doit effectuer, dans l'ordre :
- Le calcul de la taille de la pile. Pour faire ce calcul, vous utiliserez la variable `top_stack`. Attention, la pile des pentiums se déplace vers le bas...
- L'allocation dans la variable `stack` de la structure `Tproc` du processus `idx` de la place nécessaire pour sauvegarder la pile. Cette partie peut être évitée en définissant la variable `stack` comme un tableau de 64536 octets [1].
- La Copie de la pile courante dans la structure `Tproc`.
- La sauvegarde du contexte dans la sous structure `jmp_buf` de la structure `Tproc`.
- La restauration du processus entrant dans le cas où le `setjmp` renvoie vrai.

La fonction `mylongjmp` doit effectuer, dans l'ordre :
- L'affection de la variable `elu`
- La restauration des registres du processus.

Attention, dans la fonction `mysetjmp`, avant la restauration de la pile du processus entrant, toutes les variables locales et tous les paramètres de la fonction deviennent invalides puisqu'ils sont placés dans la pile.

Question 4

Testez vos fonctions `mysetjmp` et `mylongjmp`. avec les exemples vus en TD.

Question 5

Définissez deux fonctions `f` et `g` qui vous serviront à tester votre ordonnanceur.

Indication

Ces fonctions doivent se terminer et durer suffisamment longtemps pour pouvoir observer l'exécution. Il vous est est donc conseillé de faire des boucles suffisamment longues et de faire des affichages réguliers.

1. En effet, dans le TP, votre pile fera entre 100 et 2000 octets. Dans le cas général, cette solution risque de vous faire des débordements de pile.

Question 6

Programmez la fonction `new_proc`. Enregistrer vos fonctions `f` et `g` avec `new_proc`.

Indication

Cette fonction doit trouver une entrée libre dans la table des processus, noter cette entrée comme utilisée et sauvegarder le contexte. Si le retour de `mysetjmp` est égal à 0, c'est qu'on vient d'enregistrer le contexte, il faut donc sortir de la fonction. Sinon, c'est que le processus vient d'être mis sur le processeur pour la première fois, il faut donc appeler `f(arg)`.

Question 7

Écrivez une fonction `int election()` qui choisit un processus à élire.

Indication

Le but de ce TP n'est pas encore d'étudier différents algorithmes d'élection. Vous pouvez donc choisir un algorithme simple. Vous choisirez comme nouveau processus le suivant de `elu` dans la table des processus. N'oubliez pas de faire un parcourt circulaire. Pensez aussi que les entrées de la table de contiennent pas toutes des processus valides.

Question 8

Définissez une fonction `void commut(int no)` qui fait juste un affichage et commencez à écrire la fonction `start_sched()`. Cette fonction devra mettre en place un gestionnaire pour le signal `SIGALRM` et amorcer l'alarme. Avant de passer à la suite, assurez vous que votre alarme est appelée régulièrement, i.e. que la fonction `commut` fait régulièrement un affichage.

Question 9

Finissez la fonction `void start_sched()`. Elle devra choisir un processus prêt et restaurer son contexte. Pour le moment, la fonction `commut()` ne fait rien, donc le processus choisi s'exécutera de bout en bout. Il ne vous est pas encore demandé de gérer la terminaison de votre application.

Question 10

Écrivez la fonction `commut`.

Indication

La fonction `commut` doit choisir un nouveau processus à mettre sur le processeur, sauvegarder celui qui s'y trouve et placer le nouveau dessus.

Question 11

Tester votre programme avec plusieurs instances de `f` et `g`.

Question 12

Dans cette question, on vous demande de vous occuper de la terminaison de la bibliothèque et des processus. Lorsqu'un processus est terminé, vous devez noter sa case comme vide, et lorsqu'il n'y a plus de processus à élire, vous devez revenir au `main` et quitter proprement votre application.

Indication

Demandez à votre chargé de TP...

Question 13

Que se passe-t-il si vous appelez `new_proc` pendant que vous recevez un signal `SIGALRM` ? Proposez une solution.

TME 2 - Implementation et utilisation de primitives de synchronisation

But

Le but de ce TP est d'implémenter les primitives *sleep* et *wakeup* dans la libsched. Le mode d'emploi de la libsched est donné en annexe.

Sujet

On souhaite ajouter dans la libsched les deux fonctions suivantes :

- *int tsleep(int pri, void *obs)* : fonction qui endort le thread appelant sur l'obstacle *obs* avec la priorité *pri*.
- *int twakeup(void *obs)* : fonction qui réveille tous les threads endormi sur l'obstable *obs*.
  Observation : Contrairement au code de *wakeup*, si une thread plus prioritaire est reveillée, il faudra explicitement appeler l'ordonnanceur (fonction *commut ()*) à la fin de la fonction *twakeup*.

Pour réliser ce TME, vous devez copier le fichier *TME_ sleep_ wakeup.tgz* du répertoir : */Vrac/noyau/* et créer l'environment de test avec la commande :

```
$tar -xzvf  TME_sleep_wakeup.tgz
```

Les répertoires créés sont :
- *src* : fichiers source de la libsched
- *include* : fichiers include de la libsched
- *obj* : fichiers objet de la libsched
- *lib* : bibliothèques
- *demo* : fichiers de tests et fichier makefile correspondants

Le seul fichier source de la *libsched* disponible pour ce TME est le *src/synch.c* où se trouvent les squelettes de la fonction *tsleep* et de la fonction *twakeup*.

Le fichier *Makefile* dans la racine crée la *libsched* tandis que les fichiers *demo/make_ sleep_ wakeup* et *demo/make_ mesg* doivent être utilisés pour créer les exécutables de test *test_ sleep_ wakeup* et *test_ mesg* respectivement.

Exercice 1

- Complétez les fonctions *tsleep* et *twakeup* qui se trouvent dans le fichier *src/synch.c*. Pour créer une nouvelle bibliothèque *libsched.a*, utilisez le fichier *Makfile* de la racine.

- Testez les fonctions *tsleep* et *twakeup* en utilisant le programme *test_ sleep_ wakeup.c* qui se trouve dans le repertoire *demo*. Utilisez pour cela le fichier *make_ sleep_ wakeup* qui se trouve sous le même répertoire.

Pour générer l'exécutable :

```
$make -f make_sleep_wakeup
```

Exercice 2

Maintenant on veut utiliser les fonctions *tsleep* et *twakeup* pour réaliser une synchronisation de type Producteur / Consommateur. Pour cela on va implanter un gestion de file de messages dont la structure est la suivante (définie *demo/mesg.h*) :

```
/* Un message */

typedef struct t_msg {
        int exp;
        void *data;
} t_msg;

/* Une file de message */

#define MAXMSG 8     /* Nombre maximum de messages dans une file */
typedef struct t_fmsg {
        t_msg file[MAXMSG]; /* Les messages */
        int nb_msg;                     /* Nombre de messages */
        unsigned int deposer; /* indice pour déposer */
        unsigned int retirer; /* indice pour retirer */
        void *file_pleine;      /* Condition d'attente file pleine */
        void *file_vide;                /* Condition d'attente file vide */
} t_fmsg;
```

Ecrivez le corps des fonctions *DeposerFile* et *RetirerFile* qui se trouvent dans le fichier *mesg.c*. Pour générer l'exécutable utiliser le fichier *make_ mesg*.

```
$ make -f make_mesg
$ ./mesg
```

Libsched : Mode d'emploi

La bibliothèque d'ordonnancement libsched permet de tester des algorirthmes d'ordonnancement de fonctions utilisateur. L'utilisateur a l'illusion que ses fonctions s'exécutent en parallèle.

Grâce à libsched, on peut définir et paramètrer de nouveaux algorithmes d'ordonnancement.

Deux fichiers sont fournis par la bibliothèque libsched.a et le fichier d'inclusion sched.h

Fonctions de la librairie

```
#include <sched.h>
int CreateProc(function_t func, void *arg, int duration);
```

Cette fonction permet de créer une nouvelle fonction (que l'on appelle processus léger) qui pourra s'exécuter en parallèle. Le paramètre func est le nom de la fonction, arg est un pointeur vers les arguments de la fonction et duration est la durée estimée de la fonction. Par défaut le paramètre duration n'est pas utilisé mais il peut être utile pour des algorithmes d'ordonnnancement du type SJF (Shortest Job First).

CreateProc retoune l'identifant du processus léger créé (pid).

```
void SchedParam(int type, int quantum, int (*felect)(void));
```

Cette fonction permet de régler les paramètres de l'ordonnanceur. type indique le type d'ordonnancement. 3 types sont possibles (définis dans sched.h) :

- BACTH indique un ordonnancement sans temps partagé de type FIFO. Dans ce cas, les paramètres quantum et felect sont ignorés.
- PREMPT indique un ordonnancement préemptif de type "tourniquet". C'est l'ordonnancement par défaut. Dans ce cas, le paramètre quantum fixe en seconde la valeur du quantum de temps.
- NEW indique une nouvelle stratégie d'ordonnancement (définie par l'utilisateur). Dans ce cas, le paramètre quantum fixe en seconde la valeur du quantum de temps. Si quantum est égal à 0, l'ordonnancement devient non préemptif (sans temps partagé). Le paramètre felect est le nom de la fonction d'élection qui sera appelée automatiquement par la librairie avec une période de "quantum" secondes (si quantum est différent de 0).

La fonction d'élection felect doit avoir la forme suivant :

```
int Mon_election(void) {
/* Choix du nouveau processus élu */
    return elu;
}
```

La fonction d'élection choisit le nouveau processus élu (à l'état RUN) en fonction des informations regroupées dans la table Tproc définie dans sched.h :

```
// Types d'ordonnancement

#define NEW       1    // Nouvelle strategie definie par l'utilisateur
#define PREMPT    2    // Ordonnancement preemptif
#define BATCH     3    // Ordonnancement de type Batch

#define MAXPROC   15   // Nombre maximum de processus
```

```
#define MINPRIO   0
#define MAXUSERPRIO 50 // Priorité maximum d'un thread utilisateur
#define MAXPRIO   100  // Priorité maximum


// Etat d'un thread
#define RUN 1
#define IDLE 2
#define ZOMB 3
#define SLEEP 4

typedef void (*function_t)();

/* Table des descripteurs de processus */


struct proc {
  int p_flag;                   // Etat de la tache
  int p_pri;                   // Priorité
  int p_usrpri;                // Priorité en mode utilisateur
  int p_pid;                    // Pid
  sigjmp_buf p_env;             // Contexte matériel (registres)
  function_t p_func;            // Le code
  void *p_arg;                  // Les arguments
  unsigned long p_stack_size;             // La taille de la pile d'exécution
  char *p_stack_svg;            // La pile
  struct timeval p_end_time;    // date de fin
  struct timeval p_start_time;  // date de création
  struct timeval p_realstart_time;  // date de lancement
  double p_ncpu;                // "cpu" consommés
  double p_duration;            // temps estimé de la tâche
  void *p_ptr;                  // Pointeur pour données additionnelles
};

struct proc Tproc[MAXPROC];
```

Une fois le processus choisi, felect doit retourner l'indice dans Tproc du processus élu.

```
int GetElecProc(void);
```

Fonction qui retourne l'indice dans Tproc du processus élu.

```
void sched(int printmode);
```

Cette fonction lance l'ordonnanceur. L'ordonnancement effective des processus ne commence qu'à partir de l'appel à cette fonction. Par défaut l'ordonnanceur exécute un algorithme similaire à Unix à base de priorité dynamique. Le paramètre printmode permet de lancer l'ordonnanceur en mode " verbeux ". Si printmode est différent de 0, l'ordonnanceur affichera à chaque commutation la liste des tâches prêtes. Cette fonction se termine lorsqu'il n'existe plus de tâche à l'état prêt (RUN).

```
void commut(int s);
```

Cette fonction réalise une commutation de tâche. Elle appelle la fonction d'élection pour choisir la tâche prête de plus haute priorité et change le contexte pour exécuter la tâche élue. Le paramêtre s peut prendre une valeur quelconque (cela n'a pas d'influence sur la commutation).

```
void PrintStat(void);
```

Cette fonction affiche les statistiques sur les tâches exécutées (temps réel d'exécution, temps processeur consommé, temps d'attente).

Exemple L'exemple suivant illustre l'utilisation des primitives

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <malloc.h>
#include <sched.h>

// Fonction utilisateur

void MonProc(int *pid) {
    long i;

    for (i=0;i<8E7;i++)
        if (i%(long)4E6 == 0)
            printf("%d - %ld\n",*pid, i);

    printf("############ FIN PROC %d\n\n", *pid );
}

// Exemple de primitive d'election definie par l'utilisateur
// Remarques : les primitives d'election sont appelées directement
// depuis la librairie. Elles ne sont appelées que si au
// moins un processus est à l'etat pret (RUN)
// Ces primitives manipulent la table globale des processus
// définie dans sched.

// Election aléatoire

int RandomElect(void) {
    int i;

    printf("RANDOM Election !\n");

    do {
        i = (int) ((float)MAXPROC*rand()/(RAND_MAX+1.0));
    } while (Tproc[i].p_flag != RUN);
    return i;
}




int main (int argc, char *argv[]) {
    int i;
```

```
    int *j;

    // Créer 3 processus

    for (i = 0; i < 3; i++) {
        j = (int *) malloc(sizeof(int));
        *j= i;
        CreateProc((function_t)MonProc,(void *)j, 0);
    }

    // Exemples de changement de paramètres

    // Définir une nouvelle primitive d'election avec un quantum de 2 secondes
    SchedParam(NEW, 2, RandomElect);

    // Redéfinir le quantum par defaut
    SchedParam(PREMPT, 2, NULL);

    // Passer en mode batch
    SchedParam(BATCH, 0, NULL);

    // Lancer l'ordonnanceur en mode non "verbeux"
    sched(0);

    // Imprimer les statistiques
    PrintStat();

    return EXIT_SUCCESS;
}
```

TME 3 - Implementation du mécanisme de timeout

But

Le but de ce TME est d'implanter le mécanisme de *timeout* comme nous avons vu en TD (Gestion de Temps).

Sujet

Nous souhaitons ajouter à la libsched la fonction **timeout** qui permet de spécifier que la fonction *fun* doit être exécutée après *tim* ticks d'horloge avec l'argument *arg* :

timeout (void (*fun) ( ), void *arg, int tim).

Comme nous avons vu en TD, la fonction qui traite l'interruption horloge contrôle le vecteur de fonctions (vecteur *callout*) en exécutant chaque fonction au bon moment (voir fonctions *clock* et *restart* du TD).

Vous devez récupérer le fichier *TME_ callout.tgz* dans */Vrac/noyau/* et créer l'environment de test avec la commande :

```
$tar -xzvf  TME_callout.tgz
```

Les répertoires créés sont les mêmes que ceux du TME précédent (*tsleep/twakeup*). Cependant cette fois-ci, le seul fichier source de la *libsched* disponible pour le TME est *src/callout.c* où se trouvent les squelettes de la fonction *timeout* et de la fonction *restart*.

La structure *struct callo* est défini dans le fichier *include/callo.h* et la variable *v* où se trouve le vecteur de *callout* est déclarée dans le fichier *include/var.h*

Exercice 1

Vous devez programmer les fonctions **timeout** et **restart** afin d'offrir un mécanisme de timeout aux programmeurs. Pour cela, vous allez compléter le corps des fonctions *timeout* et *restart* que se trouvent dans le fichier *src/callout.c*. Pour générer la bibliothèque *libsched*, vous devez utilisez le fichier *Makefile* que se trouve sous la racine. Le fichier *demo/test1_ callo.c* permet de tester le mécanisme de *timeout*. Utilisez le fichier *makecallo* pour générer l'exécutable :

```
$make -f makecallo test1_callo
```

L'exécutable *test1_ callo* doit générer la sortie suivante :

```
8 TICKS: func1 appelée par main
12 TICKS: func3 appelée par main
14 TICKS: func2 appelée par main
17 TICKS: func3 appelée par func2
18 TICKS: func4 appelée par func1
```

Observations : Si nécessaire, vous pouvez utiliser les macros MASK_ALRM et UNMASK_ALRM pour simuler le masquage/démasquage d'interruption (voir fichier *include/callo.h*).

Exercice 2

Maintenant nous voulons que la fonction *timeout* renvoie une valeur entière (*id*), qui identifiera l'appel du timeout en question :

id = timeout (void (*fun) ( ), void *arg, int tim).

Pour cela, le champ *int c_ cid* a été ajouté à la *struct callo* afin sauvegarder l'identifiant du *timeout* (voir fichier *include/callo.h*).

En vous inspirant de l'algorithme utilisé par *timeout*, programmez la fonction *untimeout(ident)*, qui enlève de la table de *callout* l'entrée correspondante au *timeout* dont l'identifiant est *ident*. La fonction renvoie -1, si elle n'a pas trouvé l'identifiant. Sinon elle renvoie 0.

Vous devez utilisez les fichier it demo/test2_callo.c pour tester la fonction *untimeout*.

Utilisez le fichier *makecallo* pour générer l'exécutable :

```
$make -f makecallo test2_callo
```

L'exécutable *test2_ callo* doit générer la sortie suivante :

```
8 TICKS: func1 appelée par main
12 TICKS: func3 appelée par main
18 TICKS: func4 appelée par func1
```

TME 4 - Implementation des fonctions du Buffer Cache

But

Le but de ce TME est d'implanter des fonctions du buffer cache.

Sujet

Nous souhaitons ajouter à la *libsched* un mécanisme de buffer cache.

Vous devez récupérer le fichier *TME_ buffer_ cache.tgz* dans */Vrac/noyau* et créer l'environment de test avec la commande :

> tar -xzvf TME_buffer_cache.tgz

Les répertoires créés sont les mêmes que ceux des TMEs précédents (*tme_ sleeep_ wakeup* et *tme_ callout*). Cependant cette fois-ci, le seul fichier source de la *libsched* disponible pour le TME est **src/bio.c** où se trouve les fonctions du buffer cache à completer. La structure *struct buf* est définie dans le fichier *include/buf.h* ainsi que les flags utilisés par le buffer cache.

Exercice 1

Vous devez programmer les fonctions *getblk* et *brelse* du buffer cache. Pour générer la bibliothèque *libsched*, vous devez utilisez le fichier *Makefile* qui se trouve sous la racine.

L'application *demo/test1_ buff1.c* permet de tester les deux fonctions. Utilisez le fichier *makebuff* pour générer l'exécutable :

> make -f makebuff test1_buff

L'application *test1_ buff1* crée les threads **UTIL1** et **UTIL2**. Les deux threads appellent la fonction *getblk* pour le bloc 0 du device 0. Cependant, comme la thread **UTIL2** fait une attente active au début de son code, la thread **UTIL1** appelera la fonction *buf=getblk (0,0)* en premier. Lorsque **UTIL2** appelera la même fonction, elle sera bloquée. Elle ne sera réveillée lorsque la thread **UTIL1** libèrera le bloc (fonction *brelse(buf)*). L'exécutable *test1_ buff* doit générer la sortie suivante :

```
UTIL1 : Demande de [0, 0] ([dev, blkno])...
UTIL1 : [0, 0] obtenu
UTIL1 : Debut du traitement...
UTIL2 : Demande de [0, 0] ([dev, blkno])...
```

```
SLEEP
UTIL1 : Appel à brelse...
WAKEUP
###################### fin UTIL1
UTIL2 : [0, 0] obtenu
UTIL2 : Debut du traitement...
UTIL2 : Fin du traitement...
UTIL2 : Appel à brelse...
###################### fin UTIL2
```

Observations :
- Pour endormir ou réveiller une thread, utilisez les fonctions *tsleep* et *twakeup* respectivement.
- Si nécessaire, vous pouvez utiliser les fonctions *int splbio()* et *spl(int s)* pour simuler le masquage/démasquage d'interruption de l'interrruptio, disque. Les constantes NORMAL et BDINHB ont été définies.

Exercice 2

Maintenant vous devez compléter le corps des fonctions **bread** et **bwrite**. Pour tester ces fonctions, vous devez utiliser l'application *demo/test2_ buff.c.* en utilisant le fichier *makebuff* pour générer l'exécutable :

```
$make -f makebuff test2_buff
```

Dans cette application, après avoir obtenu le bloc 0 du device 0 (*buf=getblk(0,0)*), la thread **UTIL1** écrit le message ***UTIL1 MSG***" dans ce buffer en utilisant la fonction *bwrite (buf)*. Lorsque la thread **UTIL2** essaie de lire ce même bloc (*buf=bread (0,0)*), elle se bloquera parce que le bloc se trouve dans l'état B_BUSY, pris par la thread **UTIL1**. A la fin de l'écriture du bloc, la thread **UTIL1**, endormie en attente de la fin E/S, sera réveillée par la fonction *iodone* et libérera le bloc en appellant la fonction *brelse(buf)*). Cette fonction réveillera alors la thread **UTIL2**. A la fin, la thread **UTIL2** libère elle aussi le bloc en appelant *brelse (buf).*

Observations : Les entrées-sorties sur disque de la *libsched* sont simulées. Dans cette exemple, l'écriture dure 50 ticks horloges.

L'exécutable *test2_ buff* doit générer la sortie suivante :

```
UTIL1 : Demande de [0, 0] ([dev, blkno])...
UTIL1 : [0, 0] obtenu
UTIL1 : Debut du traitement...
UTIL2 : Demande de [0, 0] ([dev, blkno])...
SLEEP
UTIL1 : Fin du traitement...
UTIL1 : Appel à bwrite !B_ASYNC...
Strategy : accés au device 0
Strategy : Fin e/s dans 50 ticks...
SLEEP
WAKEUP
###################### fin UTIL1
UTIL2: [0, 0] obtenu. Contenu: ***UTIL1 MSG***
UTIL2 : Debut du traitement...
UTIL2 : Fin du traitement...
UTIL2 : Appel à brelse...
###################### fin UTIL2
```

Annexes des Travaux Dirigés UNIX

Définition des types et structures de données du noyau

Fichier buf.h

```
1  /*
2   * Each buffer in the pool is usually doubly linked into 2 lists:
3   * the device with which it is currently associated (always)
4   * and also on a list of blocks available for allocation
5   * for other use (usually).
6   * The latter list is kept in last-used order, and the two
7   * lists are doubly linked to make it easy to remove a buffer
8   * from one list when it was found by looking through the other.
9   * A buffer is on the available list, and is liable
10  * to be reassigned to another disk block, if and only
11  * if it is not marked BUSY.  When a buffer is busy, the
12  * available-list pointers can be used for other purposes.
13  * Most drivers use the forward ptr as a link in their I/O active queue.
14  * A buffer header contains all the information required to perform I/O.
15  */
16 struct buf
17 {
18     int     b_flags;        /* buffer flags */
19     struct  buf *b_forw;    /* previous buf on b_list */
20     struct  buf *b_back;    /* next buf on b_list */
21     struct  buf *av_forw;   /* previous buf on av_list */
22     struct  buf *av_back;   /* next buf on av_list */
23     int     b_dev;          /* major+minor device name */
24     int     b_count;        /* transfer count */
25     union {
26         caddr_t b_un_addr;          /* low order core address */
27         struct filsys *b_un_filsys; /* superblocks */
28         struct dinode *b_un_dino;   /* ilist */
29         daddr_t *b_un_daddr;        /* indirect block */
30     } b_un;
31     int     *b_xmem;        /* transfer memory address */
32     int     b_base;         /* page number for physical i/o */
33     int     b_size;         /* number of pages for physical i/o */
34     daddr_t b_blkno;        /* block number on device */
35     char    b_error;        /* returned after I/O */
36     int     b_resid;        /* bytes not transfered */
37     int     b_pri;          /* Priority */
38 };
39 #define b_addr  b_un.b_un_addr
40 #define b_filsys b_un.b_un_filsys
41 #define b_dino  b_un.b_un_dino
42 #define b_daddr b_un.b_un_daddr
43
44
45 /*
46  * These flags are kept in b_flags.
47  */
48 #define B_WRITE 0x0000  /* non-read pseudo-flag */
49 #define B_READ  0x0001  /* read when I/O occurs */
50 #define B_DONE  0x0002  /* transaction finished */
51 #define B_ERROR 0x0004  /* transaction aborted */
52 #define B_BUSY  0x0008  /* not on av_forw/back list */
53 #define B_WANTED 0x0010 /* issue wakeup when BUSY goes off */
54 #define B_ASYNC 0x0020  /* don't wait for I/O completion */
```

```
55 #define B_PHYS   0x0040  /* wait I/O completion : physical I/O */
56 #define B_DELWRI 0x0080 /* don't write till block leaves avail list */
57
58
59 extern struct buf buf[];                    /* The buffer pool itself */
60 extern struct buf bfreelist;                /* head of available list */
61 extern char buffers[][BSIZE];
62
63
64 /*
65  * Fast access to buffers in cache by hashing.
66  */
67
68
69 #define bhash(d,b)       ((struct buf *)&hbuf[((int)d+(int)b)&v.v_hmask])
70
71
72 struct hbuf
73 {
74     int      b_flags;
75     struct   buf *b_forw;
76     struct   buf *b_back;
77 };
78
79
80 extern struct hbuf hbuf[];
```

Fichier callo.h

```
1  /*
2   * The callout structure is for a routine arranging
3   * to be called by the clock interrupt
4   * (clock.c) with a specified argument,
5   * in a specified amount of time.
6   */
7
8
9  struct  callo
10 {
11     int     c_time;         /* incremental time */
12     caddr_t c_arg;          /* argument to routine */
13     int     (*c_func)();    /* routine */
14 };
```

Fichier conf.h

```
1  /*
2   * Used to dissect integer device code
3   * into major (driver designation) and
4   * minor (driver parameter) parts.
5   */
6  struct
7  {
8      char    d_major;
9      char    d_minor;
10 };
11
12
13 /*
14  * Declaration of device
15  * switch. Each entry (row) is
16  * the only link between the
17  * main unix code and the driver.
18  * The initialization of the
19  * device switches is in the
20  * file config.c.
21  * Character device switch.
22  */
23 struct  cdevsw
24 {
25     int     (*d_open)();
26     int     (*d_close)();
27     int     (*d_read)();
28     int     (*d_write)();
29     int     (*d_xint)();
30     int     (*d_ioctl)();
31 } cdevsw[];
32
33
34 /*
35  * Block device switch.
36  */
37 struct  bdevsw
38 {
39     int     (*d_open)();
40     int     (*d_close)();
41     int     (*d_strategy)();
42 } bdevsw[];
```

Fichier fblk.h

```
1 struct  fblk
2 {
3     int      df_nfree;
4     daddr_t  df_free[NICFREE];
5 };
```

Fichier filsys.h

```
1  /*
2   * Structure of the super-block
3   */
4  struct  filsys
5  {
6      unsigned short s_isize; /* size in blocks of i-list */
7      daddr_t s_fsize;        /* size in blocks of entire volume */
8      short   s_nfree;        /* number of addresses in s_free */
9      daddr_t s_free[NICFREE];/* free block list */
10     short   s_ninode;       /* number of i-nodes in s_inode */
11     ino_t   s_inode[NICINOD];/* free i-node list */
12     char    s_flock;        /* lock during free list manipulation */
13     char    s_ilock;        /* lock during i-list manipulation */
14     char    s_fmod;         /* super block modified flag */
15     char    s_ronly;        /* mounted read-only flag */
16     time_t  s_time;         /* last super block update */
17     daddr_t s_tfree;        /* total free blocks*/
18     ino_t   s_tinode;       /* total free inodes */
19     short   s_m;            /* interleave factor */
20     short   s_n;            /* " " */
21     char    s_fname[6];     /* file system name */
22     char    s_fpack[6];     /* file system pack name */
23
24
25     /* stuff for inode hashing */
26     ino_t   s_lasti;        /* start place for circular search */
27     ino_t   s_nbehind;      /* est # free inodes before s_last */
28 };
29
30
31 #define NICFREE 50
32 #define NICINOD 100
```

Fichier inode.h

```
1  #define NADDR 13
2
3  struct inode
4  {
5      char            i_flag;
6      char            i_count;  /* reference count */
7      dev_t           i_dev;    /* device where inode resides */
8      ino_t           i_number; /* i number, 1-to-1 with device address */
9      unsigned short  i_mode;
10     short           i_nlink;  /* directory entries */
11     short           i_uid;    /* owner */
12     short           i_gid;    /* group of owner */
13     off_t           i_size;   /* size of file */
14     struct {
15         daddr_t     i_addr[NADDR]; /* if normal file/directory */
16         daddr_t     i_lastr;       /* last logical block read (for read-ahead) */
17     };
18 };
19
20
21 extern struct inode inode[];/* The inode table itself */
22
23 /* flags */
24 #define ILOCK  01    /* inode is locked */
25 #define IUPD   02    /* file has been modified */
26 #define IACC   04    /* inode access time to be updated */
27 #define IMOUNT 010   /* inode is mounted on */
28 #define IWANT  020   /* some process waiting on lock */
29 #define ITEXT  040   /* inode is pure text prototype */
30 #define ICHG0  100   /* inode has been changed */
31
32 /* modes */
33 #define IFMT    0170000  /* type of file */
34 #define IFDIR   0040000  /* directory */
35 #define IFCHR   0020000  /* character special */
36 #define IFBLK   0060000  /* block special */
37 #define IFREG   0100000  /* regular */
38 #define IFMPC   0030000  /* multiplexed char special */
39 #define IFMPB   0070000  /* multiplexed block special */
40 #define ISUID   04000    /* set user id on execution */
41 #define ISGID   02000    /* set group id on execution */
42 #define ISVTX   01000    /* save swapped text even after use */
43 #define IREAD   0400     /* read, write, execute permissions */
44 #define IWRITE  0200
45 #define IEXEC   0100
```

Fichier mount.h

```
1  /*
2   * Mount structure.
3   * One allocated on every mount.
4   */
5  struct  mount
6  {
7      int       m_flags;          /* status */
8      dev_t     m_dev;            /* device mounted */
9      struct inode *m_inodp;      /* pointer to mounted on inode */
10     struct buf *m_bufp;         /* buffer for super block */
11     struct inode *m_mount;      /* pointer to mount root inode */
12 } mount[NMOUNT];
13
14
15 #define MFREE   0
16 #define MINUSE  1
17 #define MINTER  2
```

Fichier param.h

```
1  /*
2   * fundamental constants
3   * cannot be changed
4   */
5
6
7  #define CBSIZE   12     /* number of info char in a clist block */
8  #define CROUND   15     /* sizeof(int *) + CBSIZE - 1 */
9  #define SROUND   7      /* CBSIZE>>1 */
10
11
12 /*
13  * processor priority levels
14  */
15 #define CLINHB   7      /* clock inhibit level */
16 #define BDINHB   6      /* block device inhibit level */
17 #define CDINHB   5      /* character device inhibit level */
18 #define CALOUT   4      /* clock callout processing level */
19 #define CDINTR   3      /* character device interrupt level */
20 #define WAKEUP   2      /* clock wakeup processing level */
21 #define SWITCH   1      /* switch processing level */
22 #define NORMAL   0      /* normal processing level */
```

Fichier proc.h

```
1  /*
2   * One structure allocated per active process.
3   * It contains all data needed
4   * about the process while the
5   * process may be swapped out.
6   * Other per process data (user.h)
7   * is swapped with the process.
8   */
9
10
11 struct  proc
12 {
13     short    p_addr;     /* address of swappable image */
14     short    p_size;     /* size of swappable image (in blocks) */
15     int      p_flag;     /* process flags */
16     char     p_stat;     /* process state */
17     char     p_pri;      /* priority, negative is high */
18     char     p_nice;     /* nice for scheduling */
19     long     p_sig;      /* signal number sent to this process */
20     short    p_uid;      /* real user id, used to direct tty signals */
21     short    p_suid;     /* set (effective) user id */
22     short    p_time;     /* resident time for scheduling */
23     int      p_cpu;      /* cpu usage  for scheduling */
24     short    *p_ttyp;    /* controlling tty */
25     short    p_pid;      /* unique process id */
26     short    p_ppid;     /* process id of parent */
27     caddr_t p_wchan;    /* event process is awaiting */
28     struct text *p_textp;  /* pointer to text structure */
29     short    p_tsize;    /* size of text */
30     short    p_ssize;    /* size of stack */
31     short    p_clktim;   /* time to alarm clock signal */
32 } proc[NPROC];
33
34
35 /* stat codes */
36 #define SSLEEP  1    /* awaiting an event */
37 #define SWAIT   2    /* (abandoned state) */
38 #define SRUN    3    /* running */
39 #define SIDL    4    /* intermediate state in process creation */
40 #define SZOMB   5    /* intermediate state in process termination */
41 #define SSTOP   6    /* process being traced */
42 #define SXBRK   7    /* process being xswapped */
43 #define SXSTK   8    /* process being xswapped */
44 #define SXTXT   9    /* process being xswapped */
45
46
47 /* flag codes */
48 #define SLOAD   0x0001      /* process in memory */
49 #define SSYS    0x0002      /* scheduling process */
50 #define SLOCK   0x0004      /* process locked in memory */
51 #define SSWAP   0x0008      /* process is being swapped out */
52 #define STRC    0x0010      /* process is being traced */
53 #define SWTED   0x0020      /* another tracing flag */
54 #define SMOVE   0x0040      /* process moved */
```

```
55 #define SULOCK   0x0080      /* user settable lock in core */
56 #define STEXT    0x0100      /* text pointer is valid */
57 #define SSPART   0x0200      /* process is partially swapped out */
```

Fichier signal.h

```
1  #define SIGHUP   1        /* hangup */
2  #define SIGINT   2        /* interrupt (rubout) */
3  #define SIGQUIT  3        /* quit (ASCII FS) */
4  #define SIGILL   4        /* illegal instruction (not reset when caught)*/
5  #define SIGTRAP  5        /* trace trap (not reset when caught) */
6  #define SIGIOT   6        /* IOT instruction */
7  #define SIGEMT   7        /* EMT instruction */
8  #define SIGFPE   8        /* floating point exception */
9  #define SIGKILL  9        /* kill (cannot be caught or ignored) */
10 #define SIGBUS   10       /* bus error */
11 #define SIGSEGV  11       /* segmentation violation */
12 #define SIGSYS   12       /* bad argument to system call */
13 #define SIGPIPE  13       /* write on a pipe with no one to read it */
14 #define SIGALRM  14       /* alarm clock */
15 #define SIGTERM  15       /* software termination signal from kill */
16 #define SIGUSR1  16       /* user defined signal 1 */
17 #define SIGUSR2  17       /* user defined signal 2 */
18 #define SIGCLD   18       /* death of a child */
19 #define SIGPWR   19       /* power-fail restart */
20
21
22 #define NSIG     19
23
24
25 #define SIG_DFL (int (*)())0
26 #define SIG_IGN (int (*)())1
```

Fichier text.h

```
1  /*
2   * Text structure.
3   * One allocated per pure
4   * procedure on swap device.
5   * Manipulated by text.c
6   */
7  struct text
8  {
9      daddr_t x_daddr;                /* disk address of segment */
10     caddr_t x_caddr;                /* core address, if loaded */
11     long    x_size;                 /* size (*64) */
12     struct inode *x_iptr;           /* inode of prototype */
13     char    x_count;                /* reference count */
14     char    x_ccount;               /* number of loaded references */
15 } text[NTEXT];
```

Fichier tty.h

```
1  /*
2   * A clist structure is the head
3   * of a linked list queue of characters.
4   * The characters are stored in 4-word
5   * blocks containing a link and 6 characters.
6   * The routines getc and putc (prim.c)
7   * manipulate these structures.
8   */
9  struct clist
10 {
11     int     c_cc;           /* character count */
12     char    *c_cf;          /* pointer to first character */
13     char    *c_cl;          /* pointer to last character */
14 };
15
16
17 struct cblock {
18     struct cblock *c_next;
19     char    c_info[CBSIZE];
20 };
21
22
23 struct  cblock  *cfreelis;
24
25
26 #define CBSIZE  12              /* nombre de caracteres par blocs */
27 #define CROUND  15      (sizeof(*int)+CBSIZE-1)
28 #define SROUND  7       (CBSIZE>>1)
29
30
31
32
33 /* Internal state bits */
34 #define CARR_ON 0x0001          /* Software copy of carrier present */
35 #define WOPEN   0x0002          /* Waiting for open to complete */
36 #define ISOPEN  0x0004          /* Device is open */
37 #define OPEN    0X0004
38 #define READING 0x0010          /* Input in progress */
39 #define WRITING 0x0020          /* Output in progress */
40 #define TTSTOP  0x0040          /* <^s><^q> processing */
41 #define TTSTART 0x0080          /* <^s><^q> processing */
42 #define TIMEOUT 0x0100          /* Delay timeout in progress */
43 #define ASLEEP  0x0200          /* Wakeup when output done */
44 #define XCLUDE  0x0400          /* exclusive use flag, against open */
45 #define HUPCLS  0x0800          /* Hangup after last close */
46 #define ATTENT  0x1000          /* Attention character received */
47 #define TBLOCK  0x2000          /* Tandem queue blocked */
48 #define CNTLQ   0x8000          /* interpret t_un as clist */
```

Fichier types.h

```
1 typedef long           daddr_t  /* disk address */
2 typedef char *         caddr_t  /* core address */
3 typedef int            dev_t    /* device code  */
4 typedef unsigned short ino_t    /* inode number */
```

Fichier user.h

```
1  /*
2   * The user structure.
3   * One allocated per process.
4   * Contains all per process data
5   * that doesn't need to be referenced
6   * while the process is swapped.
7   * The user block is USIZE blocs
8   * long; resides at virtual kernel
9   * location 0xc000; contains the system
10  * stack per user; is cross referenced
11  * with the proc structure for the
12  * same process.
13  */
14 struct user
15 {
16     int     u_rsav[2];      /* saved env. for process switching */
17     int     u_ssav[2];      /* saved env. for swapping */
18     int     u_qsav[2];      /* saved env. for signaling */
19
20     struct proc *u_procp;   /* pointer to proc structure */
21
22     char    u_error;        /* return error code */
23     char    u_intflg;       /* catch intr from sys */
24     int     *u_ar0;         /* address of users saved R0 */
25     int     u_arg[20];      /* arguments to current system call */
26     int     *u_ap;          /* pointer to arglist */
27
28     struct file *u_ofile[NOFILE]; /* pointers to open file */
29
30     int     u_signal[NSIG];       /* disposition of signals */
31
32     int     u_uid;          /* effective user id */
33     int     u_gid;          /* effective group id */
34     int     u_ruid;         /* real user id */
35     int     u_rgid;         /* real group id */
36
37     int     u_uisa[16];     /* prototype of segmentation addresses */
38     int     u_uisd[16];     /* prototype of segmentation descriptors */
39
40     int     u_tsize;        /* text size (in blocs) */
41     int     u_dsize;        /* data size (in blocs) */
42     int     u_ssize;        /* stack size (in blocs) */
43     int     u_csize;        /* amount of stack in use (in blocs) */
44
45     long    u_utime;        /* this process user time */
46     long    u_stime;        /* this process system time */
47     long    u_cutime;       /* sum of childs' utimes */
48     long    u_cstime;       /* sum of childs' stimes */
49
50     int     u_segflg;       /* flag for i/o user or kernel */
51     char    *u_base;        /* base address for IO */
52     int     u_count;        /* bytes remaining for IO */
53     long    u_offset;       /* offset in file for IO */
54     struct inode *u_cdir;   /* pointer to inode of current dir */
```

```
55      char    u_dbuf[DIRSIZ]; /* current pathname component */
56      char    *u_dirp;        /* current pointer to inode */
57      struct  {               /* current directory entry */
58          int     u_ino;
59          char    u_name[DIRSIZ];
60      } u_dent;
61      struct inode *u_pdir;   /* inode of parent directory of dirp */
62                                      structures of open files */
63
64      int     u_stack[1]      /* kernel stack per user
65                               * extends from u + USIZE
66                               * backward not to reach here
67                               */
68  } u;
69
70
71  /* u_error codes */
72
73  #define EPERM   1       /* Not super-user                       */
74  #define ENOENT  2       /* No such file or directory            */
75  #define ESRCH   3       /* No such process                      */
76  #define EINTR   4       /* interrupted system call              */
77  #define EIO     5       /* I/O error                            */
78  #define ENXIO   6       /* No such device or address            */
79  #define E2BIG   7       /* Arg list too long                    */
80  #define ENOEXEC 8       /* Exec format error                    */
81  #define EBADF   9       /* Bad file number                      */
82  #define ECHILD  10      /* No children                          */
83  #define EAGAIN  11      /* No more processes                    */
84  #define ENOMEM  12      /* Not enough core                      */
85  #define EACCES  13      /* Permission denied                    */
86  #define EFAULT  14      /* Bad address                          */
87  #define ENOTBLK 15      /* Block device required                */
88  #define EBUSY   16      /* Mount device busy                    */
89  #define EEXIST  17      /* File exists                          */
90  #define EXDEV   18      /* Cross-device link                    */
91  #define ENODEV  19      /* No such device                       */
92  #define ENOTDIR 20      /* Not a directory                      */
93  #define EISDIR  21      /* Is a directory                       */
94  #define EINVAL  22      /* Invalid argument                     */
95  #define ENFILE  23      /* File table overflow                  */
96  #define EMFILE  24      /* Too many open files                  */
97  #define ENOTTY  25      /* Not a typewriter                     */
98  #define ETXTBSY 26      /* Text file busy                       */
99  #define EFBIG   27      /* File too large                       */
100 #define ENOSPC  28      /* No space left on device              */
101 #define ESPIPE  29      /* Illegal seek                         */
102 #define EROFS   30      /* Read only file system                */
103 #define EMLINK  31      /* Too many links                       */
104 #define EPIPE   32      /* Broken pipe                          */
```

Fichier var.h

```
1  /*
2   * The following is used by machdep.c
3   */
4  struct  var     {
5      int      v_uprocs;       /* max # of user's process */
6      int      v_timezone;     /* timezone */
7      int      v_cargs;        /* max # of bytes given to exec */
8      int      v_cspeed;       /* default asynchronous line speed */
9      long     v_fill[20];     /* rfu */
10     int      v_proc;         /* proc table */
11     struct proc *ve_proc;
12     int      vs_proc;
13     int      v_clist;        /* cblock list */
14     struct cblock *ve_clist;
15     int      vs_clist;
16     int      v_mount;        /* mount table */
17     struct mount *ve_mount;
18     int      vs_mount;
19     int      v_inode;        /* inode table */
20     struct inode *ve_inode;
21     int      vs_inode;
22     int      v_file;         /* file table */
23     struct file *ve_file;
24     int      vs_file;
25     int      v_cmap;         /* core map */
26     struct map *ve_cmap;
27     int      vs_cmap;
28     int      v_smap;         /* swap map */
29     struct map *ve_smap;
30     int      vs_smap;
31     int      v_callout;      /* callout table */
32     struct callo *ve_callout;
33     int      vs_callout;
34     int      v_text;         /* text segment table */
35     struct text *ve_text;
36     int      vs_text;
37     int      v_buf;          /* data buffers */
38     struct buf *ve_buf;
39     int      vs_buf;
40     /* beginning of internal buffers */
41     int      v_Buf;          /* data buffers */
42     struct Buffer_Data *ve_Buf;
43     int      vs_Buf;
44     int      v_io;           /* space for io_info buf */
45     long     *ve_io;
46     int      vs_io;
47     int      v_hbuf;         /* structures for data buffers hashing */
48     struct hbuf *ve_hbuf;
49     int      vs_hbuf;
50     int      v_hino;         /* structures for inode hashing */
51     struct inode **ve_hino;
52     int      vs_hino;
53     int      v_hproc;        /* hash proc lists */
54     struct proc **ve_hproc;
```

```
55      int       vs_hproc;
56      int       v_zero;
57      int       *ve_zero;
58      int       vs_zero;
59  } v;
60
61
62  struct proc *proc_end;  /* last logical proc of proc table */
63  long bufbase;
```