Examen Noyau M1 – Master SAR

Décembre 2006

3 heures – Tout document papier autorisé

Barème donné à titre indicatif

L. Arantes, P. Sens, Gaël Thomas

1. SYSTEME DE FICHIERS (7 POINTS)

Nous considérons la partition numéro 0 avec des blocs de taille 1024 octets.

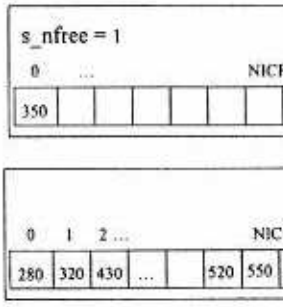
Le programme ci-dessous manipule le fichier *fich1* qui se trouve dans le même répertoire que le programme (répertoire courant). Nous considérons que le fichier *fich1* ne se trouve pas ouvert avant l'exécution du programme. Il a une taille de **2800** octets et est composé des blocs de données **40, 41 et 42**. Le numéro de l'inode du répertoire courant est **66** et du fichier *fich1* est le **33**. L'inode **33** n'est référencée que par le nom de fichier *fich1*.

Observations : O_APPEND permet un positionnement en fin de fichier.

SEEK_SET : seek à partir du début du fichier.

```
1 :   int main (int argc, char* argv []) {
2 :    int fd1, fd2;
3 :    char buffer[2048];
4 :    fd1 = open ("fich1", O_RDONLY);
5:     memset (buffer,'#',2048);
6:     lseek (fd1, 2000, SEEK_SET);
7 :    if (fork() == 0) {
8 :              fd2 = open ("fich1", O_RDWR | O_APPEND ) ;
9 :              write(fd2, buffer, 1200);
10 :             read(fd1, buffer, 700);
11:       close (fd1); close (fd2);
12:       exit (0);
13 :      }
14 :   wait(NULL);
16 :   close (fd1);
17 :   return 0;
18:   }
```


On suppose aussi que le super
partition 0. Sa configuration est su



1.1

Quel sera la configuration du su
supposant qu'il n'y a pas d'autres

1.2.

En ne montrant que les informat
table des descripteurs de fichiers
fils, la table de fichiers ouverts *f*
après l'exécution du **read** de la **li**

Vous préciserez tous les champs
dans les tables `u_ofile[]`, `file`
`f_flags, f_count)` et *inod*
`i_addr, i_dev, i_number)`

Avant l'exécution du programm
trouve déjà dans la table d'inod
fich1 (33).

La table d'inode sur disque comm
entrées par bloc. Les inodes sor
leur numéro d'inode. Le premie
trouve donc à l'offset 0 du bloc d

1.3. (0,5 point)

Quel est le numéro du bloc sur disque qui contient l'inode du répertoire courant ? Et celui du *fich1* ?

La figure 1 montre l'inode du répertoire courant et du ses blocs de données.

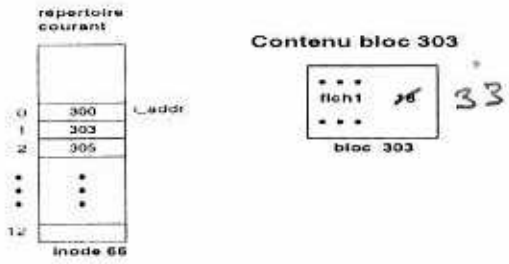


Figure 1

La figure 2 montre la configuration du buffer cache **avant** l'exécution du programme. La fonction d'hachage est égale à **(n°dev + n°bloc) % 5,** n°dev étant 0.

Nous n'indiquons dans la figure que les numéros des blocs. Aucun bloc de la *b_freelist* n'est marqué en écriture différée.

Le buffer cache est constitué de **8** blocs.

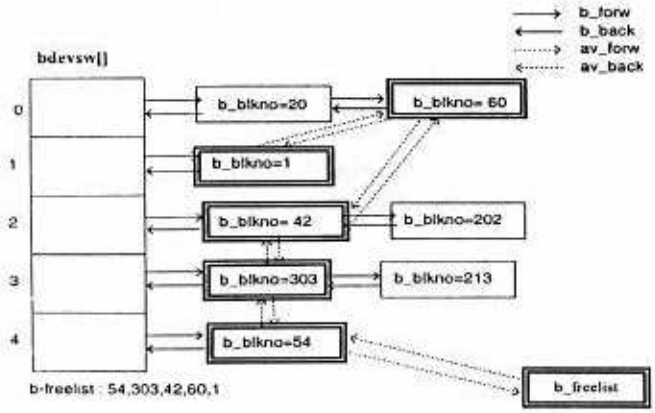


Figure 2

1.4.

Quels numéros de bloc sont acc
chaque accès en indiquant que
open, qui demandent chacun
depuis le disque ?

Donnez la nouvelle configuratio
l'exécution du **open de la ligne**
à la figure 2, **seule les** pointeu
rappelle que **les buffers sont re**
tête) et sont insérés du côté back

Nous considérons que toutes les

1.5.

Lors de l'exécution des lignes d
de blocs accédés et les entrée
(numéro de ligne, fonction) ces

Donnez la configuration de la *b*
vous demande d'indiquer les bl
des pointeurs av_forw (de la tèt

1.6

Si on considère que lors du clo
fichier *fich1* sera mise à jour, que
disque ?

1.7

En considérant que le noyau es
père en train d'exécuter le progr
lors de l'appel au **open** de la li
au open de la ligne 8 ? Justifiez

Si un troisième processus ve
concurremment avec les de
concurrence serait-elle gérée ?

2. ETUDES DU CODE DES TUBES (7 POINTS)

Le but de cet exercice est de comprendre l'implémentation des tubes anonymes (pipe). Les tubes permettent la communication entre processus de même famille. Un tube est une file d'attente FIFO implantée au niveau du noyau et accédée en lecture ou écriture à l'aide d'un descripteur. Lorsqu'un producteur écrit des données dans un tube, ces données sont stockées dans la file d'attente. Lorsqu'un consommateur lit des données à partir du tube, ces données sont retirées.

L'appel système pipe(fd) permet de créer un tube p. p est un tableau à deux entrées : p[0] permet de lire les données, p[1] permet d'écrire. Pour lire et écrire on utilise les appels read et write standards.

L'exemple suivant illustre l'utilisation des tubes. Le code des fonctions système associées est donné en annexe.

```
...
int fd[2];

pipe(fd);

if (fork() == 0) {
   /* processus lecteur */
   close(fd[0]);
   /* lecture bloquante de données dans le tube */
   /* les 10 octets lus sont les 10 octets écrits par
le père */
   /* Ces données sont retirées dès qu'elles sont
lues */
   read(fd[1], buf, 10);
   ...
}
else {
  /* processus ecrivain */
  close(fd[1]);
  /* ecriture  de données dans le tube */
  write(fd[0], buf, 10);
 ...
}
```

On vous rappelle que `readi` est la fonction du noyau qui effectue une lecture sur une inode et qu'elle utilise les champs u.u_count et u.u_offset pour savoir à partir combien d'octets doivent être lu

2.1.

Quelle est la structure de donné pipes ? Où sont stockées les donn

2.2.

Quels sont les indices qui indique

2.3

Donnez l'algorithme de readp

2.4.

Donnez l'algorithme de writep

2.5.

Combien de processus peuvent d'écriture) simultanément à un réponse en vous appuyant sur le

2.6.

Est-ce qu'une même donnée (justifier) ?

2.7.

Que ce passe t-il pour les écrivain

2.8.

Que ce passe t-il pour les lecteurs

4. NOYAU PREEMPTIF (3 POINTS)

On considère trois tâches T1, T2 e

T1 est plus prioritaire que T2 et T

On considère aussi un appel syst présente en mémoire, l'utilise p l'inode A. On suppose que sys_e

Initialement T3 est élu.

T1 et T2 sont à l'état bloqué (SLEEP) en attente sur l'événement E.

A son réveil T1 s'exécute pendant 5ms de temps CPU puis fait un appel à `sys_exo` avant de se terminer

A son réveil T2 s'exécute 5ms avant de se terminer.

Soit le scénario suivant :

Au temps t=0, T3 fait un appel à `sys_exo` puis se termine.

A t=5ms, une interruption disque qui dure 2ms fait un wakeup sur l'événement E.

On néglige les temps de commutation et le quantum est de 100 ms.

4.1. (0,5 point)

Faites un diagramme temporel représentant l'exécutant des trois tâches en considérant un noyau **non préemptif**.

A quels instants les tâches se terminent ?

4.2. (1 point)

Reprendre la question précédente avec un noyau **préemptif sans héritage de priorité.**

4.3. (1 point)

Reprendre la question précédente avec un noyau **préemptif avec héritage de priorité**.

4.4. (Question de cours) (0,5 point)

Dans quel cas un noyau (préemptif ou non) doit masquer les interruptions ?

5. PRE-CHARGEMENT DE BLOCS (3 P

Le but de cet exercice est de c
qui charge en avance les 5 pr
d'entrées/sorties du système. C
cache mais ne sera pas utilisé. C
automatiquement par iget lorsq
signature de la fonction :

```
void iprefetch(struct
```

Pour simplifier l'exercice, on n
de manière asynchrone et on n
Dans le cas où un fichier fait n
sur que le nombre de blocs du
la taille d'un bloc.

5.

Donnez le code la fonction iprefet

ANNEXE A

```
/*
 * Max allowable buffering per pipe.
 */
#define PIPSIZ  4096

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

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

        ip = ialloc(rootdev);
        if(ip == NULL)
                return;
        rfp = falloc();
        if(rfp == NULL) {
                iput(ip);
                return;
        }
        r = u.u_filex;          /* Save Read File index/descriptor */
        wfp = falloc();
        if(wfp == NULL) {

                rfp->f_count = 0;
                u.u_ofile[r] = NULL;
                iput(ip);
                return;
        }
        wfp->f_flag = FWRITE|FPIPE;
        wfp->f_inode = ip;
        rfp->f_flag = FREAD|FPIPE;
        rfp->f_inode = ip;
        ip->i_count = 2;
        ip->i_flag = IACC|IUPD|ICHG;
        ip->i_mode = IFREG;
}

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

        register struct inode *ip;
        ip = fp->f_inode;

loop:
```



```
        plock(ip);

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

                /*
                 * If there are not bo
                 * writer active, retu
                 * satisfying read.
                 */

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

        /*
         * Read and return
         */

        u.u_offset = fp->f_offset;
        readi(ip);
        fp->f_offset = u.u_offset;
        /*
         * If reader has caught up wi
         * offset and size to 0.
         */

        if(fp->f_offset == ip->i_size)
                fp->f_offset = 0;
                ip->i_size = 0;
                if(ip->i_mode&IWRITE)
                        ip->i_mode &=
                        wakeup((caddr_
                }
        }
        prele(ip);
}

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

        register struct inode *ip;
        register int c;


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

loop:
```

```
        *
        * If all done, return.
        */

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

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

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

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

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

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


ANNEXE B : FONCTIONS ET STRUCTU

```
daddr_t bmap(struct inode *ip, daddr_t

struct buf *getblk(dev_t dev, daddr_t b

plock(struct inode *ip;   /* lock an in
prele(struct inode *ip);  /* unlock an

struct  inode
{
        short   i_flag;     /* inode fl
        short   i_count;    /* referenc
        dev_t   i_dev;      /* device w
        ino_t   i_number;   /* i number
        unsigned i_mode;    /* inode mo
        short   i_nlink;    /* director
        short   i_uid;      /* owner */
        short   i_gid;      /* group of
        long    i_size;     /* file siz
        union {
           daddr_t i_a[NADDR]; /* devic
           daddr_t i_rd;       /* devic
           }i_unl;
        time_t  i_atime;    /* time las
        time_t  i_mtime;    /* time las
        time_t  i_ctime;    /* last tim
        u_short i_shlockc;  /* count of
        u_short i_exlockc;  /* count of
        struct filsys *i_fs;   /* file
}

struct buf
{
        int     b_flags;        /* buff
        struct  buf *b_forw;    /* prev
        struct  buf *b_back;    /* next
        struct  buf *av_forw;   /* prev
        struct  buf *av_back;   /* next
        int     b_dev;          /* majo
        int     b_count;        /* tran
        union {
            caddr_t b_un_addr;
            struct filsys *b_un_filsys;
            struct dinode *b_un_dino;
            daddr_t *b_un_daddr;
        } b_un;
        int     *b_xmem;        /* tran
        int     b_base;         /* page
        int     b_size;         /* numb
        daddr_t b_blkno;        /* bloc
        char    b_error;        /* retu
        int     b_resid;        /* byte
        int     b_pri;          /* Prio
};

#define BSIZE   512             /* size
```