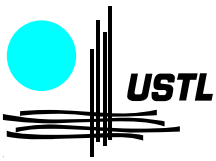


**Licence d'informatique**
**Module de Programmation des systèmes**

# Examen seconde session 2008

Philippe MARQUET

Juin 2008

**Durée : 3 heures.**
**Documents de cours, TD, et TP autorisés.**
**Les réponses seront concises et concrètes.**

## 1 Cette `alarm` vous réveille-t-elle ?

La primitive

```
int sleep(unsigned seconds);
```

suspend l'exécution du processus courant pour `seconds` secondes.

L'objet des exercices suivants est de comprendre et critiquer des implémentations de cette primitive `sleep()` à l'aide d'un minuteur géré par la primitive `alarm()` et de la délivrance du signal `SIGALRM`.

Soit la proposition suivante d'un premier étudiant :

```
static pid_t self;

static void
alrm_handler_1(int sig)
{
    kill(self, SIGCONT);
}

void
sleep_1(unsigned seconds)
{
    struct sigaction sa;

    sa.sa_handler = alrm_handler_1;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigaction(SIGALRM, &sa, (struct sigaction *) 0);

    self = getpid();
    alarm(seconds);

    kill(self, SIGSTP);
}
```

**Exercice 1**
Explicitez l'intention de l'étudiant proposant cette implémentation. Expliquez pourquoi cette implémentation est erronée. □

Un second étudiant propose une autre implémentation :

```
static int it_is_time;

static void
alrm_handler_2(int sig)
{
    it_is_time = TRUE;
}

void
sleep_2(unsigned seconds)
{
    struct sigaction sa;

    sa.sa_handler = alrm_handler_2;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigaction(SIGALRM, &sa, (struct sigaction *) 0);

    it_is_time = FALSE;
    alarm(seconds);

    while (! it-is-time)
        ;
}
```

**Exercice 2**
Explicitez l'intention de l'étudiant proposant cette implémentation. Quel(s) inconvénient(s) voyez-vous à cette implémentation. □

# 2 Lister les fichiers atopiques

Il s'agit de développer une commande `atopique` dont l'objet est de lister les entrées sous le répertoire courant qui sont des liens symboliques référençant des fichiers en dehors de la partition courante. On qualifie ces fichiers d'*atopiques*.

On développe une fonction utilitaire `atopique_file()` préliminaire pour aider à l'implantation de cette commande.

Une partition (ou disque logique, périphérique, *device*) est représentée par une valeur entière de type `dev_t` telle celle utilisée dans les structures `struct stat`.

La partition courante est celle correspondant au répertoire courant. On suppose que la valeur de la variable

```
static dev_t current_device;
```

identifie la partition courante.

**Exercice 3**
Donnez le code d'une fonction

```
int atopique_file(const char *pathname, const struct stat *st);
```

qui, considérant le fichier désigné par les deux paramètres `pathname` (son nom), et `st` (la valeur de type `struct stat` correspondante), teste si cette entrée dans le système de fichier est un fichier atopique, c'est-à-dire :

- est un lien symbolique ;
- ce lien symbolique référence un fichier qui est en dehors de la partition courante.

La fonction `atopique_file()` retourne un booléen : une valeur `TRUE` si l'entrée est un fichier atopique, une valeur `FALSE` sinon. □

Pour compléter le développement de la commande `atopique`, il reste maintenant à définir une fonction

```
void atopique(const char *pathname);
```

qui

- si l'entrée de nom `pathname` est un fichier atopique, affiche le nom cette entrée ;
- dans le cas d'un répertoire, vérifie pour chacune de ses entrées, et ce récursivement, s'il s'agit d'un fichier atopique.

Dans ses grandes lignes, la fonction `main()` de la commande `atopique` consisterait en une initialisation de la variable `current_device` suivie d'un simple appel à `atopique(".")`.

**Exercice 4**
Donnez le code de cette fonction `atopique()`. □

# 3 Word count parallèle totalisant

La commande Unix `wc` compte les lignes, mots, et lettres des fichiers dont les noms lui sont fournis sur la ligne de commande :

```
% wc xam08*.c
      93     227    2031 xam08controuv.c
     154     410    3050 xam08rdv.c
      55     105    1014 xam08sleep.c
      45      86     723 xam08wc-ps.c
      63     126     975 xam08wc-th.c
     410     954    7793 total
```

On remarque que la commande affiche aussi les nombres de lignes, mots et caractères totaux. Dans le cas ou un unique paramètre est fourni à la commande, cette ligne de totaux n'est pas produite :

```
% wc xam08sleep.c
      55     105    1014 xam08sleep.c
```

Nous désirons implémenter une version parallèle de cette commande. Le principe est de déléguer le comptage de différents fichiers à différents fils d'exécution.

Deux approches d'une telle implémentation parallèle peuvent être imaginées : une approche à base de processus et une version à base de processus légers.

**Exercice 5**
Supposant disponible la commande Unix `wc`, donnez une implémentation d'une commande `wc-ps` qui crée autant de processus qu'il y a de fichiers sur la ligne de commande, chacun des processus prenant en charge le comptage d'un fichier. On ne se soucie pas d'afficher la dernière ligne donnant les totaux ; on ne se soucie pas non plus de l'ordre d'affichage des lignes du résultat. □

On désire modifier cette commande `wc-ps` pour afficher, en fin d'exécution, la ligne fournissant les totaux.

Pour ce faire, on propose de récupérer les affichages réalisés par les processus `wc` sur leur sortie standard. On fera donc en sorte que cette sortie standard soit redirigée vers un tube.

On suppose définis le type `struct wc_s` et la fonction `pr_wc_line()` suivants :

```
struct wc_s {
    unsigned wc_w;              /* word */
    unsigned wc_l;              /* line */
```

```
    unsigned wc_c;              /* char */
};

void pr_wc_line(const char *str, const struct wc_s *pwc);
```

Cette fonction affiche sur la sortie standard une ligne de résultat comme le fait la commande `wc` pour chacun des fichiers ou pour l'éventuelle ligne des totaux. La valeur `str` est celle de la chaîne de caractères, nom de fichier ou `"total"`, affichée avec le résultat alors que la valeur pointée par `pwc` contient ce résultat.

On dispose aussi de la fonction

```
int parse_wc(int fd, struct wc_s *pwc, char *name);
```

qui

- lit sur le descripteur de fichier `fd` une ligne de résultat telle que celle affichée par la commande `wc` ;
- analyse cette ligne ;
- met à jour la structure pointée par `pwc` et la chaîne de caractères pointée par `name` avec les informations issues de cette ligne.

Si, et seulement si, la fin de fichier ou une erreur est retournée par la lecture sur le descripteur `fd`, cette fonction `parse_wc()` retourne une valeur nulle.

Dans le détail, cette nouvelle implémentation de `wc-ps` assurant l'affichage de la ligne de totaux est donc une commande :

- créant d'un tube ;
- faisant en sorte que les processus `wc` fils créés produisent leur affichage dans ce tube ;
- utilisant la fonction `parse_wc()` pour analyser les résultats lus depuis ce tube ;
- affichant ces résultats au fur et à mesure ;
- cumulant ces résultats dans une structure `struct wc_s` ;
- affichant finalement la ligne des totaux.

**Exercice 6**
Donnez le code de cette commande `wc-ps` créant le tube et les processus `wc` dont la sortie standard sera redirigée vers ce tube. □

**Exercice 7**
Expliquez comment le processus principal de cette commande `wc-ps` peut identifier la terminaison des processus fils créés. □

**Exercice 8**
Donnez le code de cette commande `wc-ps` récupérant les résultats des processus fils et assurant les affichages de ces résultats et du total. □

On traite maintenant d'une implémentation parallèle de `wc` à l'aide de processus légers. L'idée est de créer un nombre `n_thread` d'au plus `MAX_THREADS` processus légers. Cette valeur `MAX_THREADS` sera en général inférieure au nombre de fichiers à traiter ; chaque processus léger devra donc traiter plusieurs fichiers. Une structure de donnée globale

```
struct wc_job_s {
    char *wc_filename;
    struct wc_s wc_result;
    int wc_done;              /* wc_result est disponible */
};
static wc_job_s wc_jobs[MAX_JOBS];
```

mémorise les travaux à réaliser et les résultats des travaux réalisés. Le champ `wc_done` indique si oui ou non le travail a déjà été effectué.

Le principe est que chacun des `n_thread` processus légers consulte cette structure pour identifier un travail à réaliser, compte le nombre de caractères, mots, lignes d'un fichier et range le

résultat dans la structure, et ce tant qu'il reste des travaux non pris en charge. Le cœur de la fonction d'un processus léger est réalisé à l'aide de la fonction

```
void wc_file(const char *filename, struct wc_s *pwc);
```

qui retourne dans la structure pointée par `pwc` le nombre de mots, lignes et caractères du fichier `filename` (on ne demande pas de donner le code de cette fonction `wc_file()`).

De manière plus précise, le début du programme comporte les initialisations suivantes :

```
#define MAX_THREADS    ...

static int n_threads;
static int n_jobs;

int
main(int argc, char *argv[])
{
    int i;
    n_threads = min(MAX_THREADS, argc-1);
    n_jobs = argc-1;

    for (i=0 ; i<n_jobs ; i++) {
        wc_jobs[i].wc_filename = argv[i+1];
        wc_jobs[i].wc_done = FALSE;
    }
    ...
```

Les exercices suivants traitent quelques points de la suite de ce programme.

**Exercice 9**
Un processus léger se doit d'accéder en lecture à la structure `wc_jobs` pour identifier que tous les travaux ont été traités et en écriture pour modifier le résultat d'un travail qu'il a pris en charge.

Donnez un moyen pour protéger les accès concurrents entre les processus légers à cette structure `wc_jobs`. □

**Exercice 10**
Donnez le code de la fonction principale des processus légers. □

**Exercice 11**
Expliquez comment le fils d'excution principal peut détecter la terminaison de tous les travaux et donnez le code de la fonction `main()` qui

- crée les processus légers ;
- attend leur terminaison.

Il n'est pas demandé de donner le code final affichant les résultats. □

# 4 Nouveau rendez-vous avec des tubes

Le principe d'un rendez-vous entre $N$ processus est d'attendre que le $N^e$ processus arrive au rendez-vous pour laisser continuer l'ensemble des $N$ processus.

Dans le cadre de cet examen, on propose une implémentation de rendez-vous entre processus à l'aide de tubes. Cette implémentation est basée sur la structure et le principe suivant :

- un processus *leader* est élu parmi les processus désirant se synchroniser par rendez-vous ;
- deux tubes sont utilisés : un tube entre le processus leader et les processus standard, un tube entre les processus standard et le processus leader ;
- la synchronisation est assurée par le fait que la lecture dans un tube vide est bloquante ;

– quand un processus standard arrive au point de rendez-vous, il en informe le leader écrivant un caractère (mettons `'x'`) dans le tube ; il se met alors en attente du rendez-vous en lisant un caractère depuis le second tube ;
– quand le leader arrive au point de rendez-vous, il lit autant de fois le caractère `'x'` depuis le tube qu'il y a de processus standard (c'est-à-dire $N-1$) ; il libère alors ces processus en écrivant autant de caractères dans le second tube.

On utilisera la structure

```
struct rdv_s {
    pid_t rdv_leader;           /* le processus leader */
    int rdv_pipe_std2ldr[2];    /* tube standard vers leader */
    int rdv_pipe_ldr2std[2];    /* tube leader vers standard */
};
```

pour implémenter notre bibliothèque de gestion de rendez-vous.

Trois fonctions forment l'interface de cette bibliothèque : la première fonction

```
void rdv_create(struct rdv_s *rdv);
```

assure une première initialisation d'un mécanisme de rendez-vous en mettant à jour la structure pointée par `rdv`. Cette initialisation consiste uniquement en la création des tubes.

La seconde fonction

```
void rdv_leader(struct rdv_s *rdv);
```

termine l'initilisation de la structure en identifiant le processus courant comme le leader.

La troisième fonction peut alors être utilisée :

```
void rdv(struct rdv_s *rdv, unsigned int n_process);
```

est appelée par `n_process` processus (un certain nombre de processus standard et le processus leader) et assure une synchronisation par rendez-vous.

L'utilisation de la bibliothèque nécessite l'initialisation de la structure `struct rdv_s` par un processus père. Ce processus père va créer des processus fils. Parmi ces fils, un processus désigné deviendra le leader par un appel à `rdv_leader()`. Ensuite, des ensembles de fils, dont le leader est systématiquement membre, pourront se synchoniser par rendez-vous via des appels à `rdv()`.

**Exercice 12**
Donnez le code de la fontion `rdv_create()` en charge de l'initialisation de la structure via la création des tubes. □

**Exercice 13**
Donnez-le code de la function `rdv()`. En particulier, distinguez bien le rôle du processus leader et le rôle des processus standard. □

On désire étendre l'interface de notre petite bibliothèque de rendez-vous par une fonction de destruction. La destruction consiste uniquement à libérer les ressources systèmes associées aux tubes ouverts.

**Exercice 14**
Expliquez pourquoi une fonction de terminaison `rdv_destroy()` appelée par l'ensemble des processus fils, leader et standard, et faisant de simples appels à `close()` sur l'ensemble des descripteurs de fichiers ne libère pas les ressources systèmes associées aux tubes.

Expliquez comment corriger ce problème. □



Fin du sujet