Fiche d'INF104

Les machines actuelles stockent les instructions du programme et ses données dans le même espace (Von Neumann).

1 Fork et processus

Processus sous UNIX, 3 zones mémoires :
— code exécutable (text segment)
— pile de données (stack segment)
— données statiques (data segment)

Lors d'un appel à `fork`, `stack`, `data` et le pointeur sur `text` sont copiés pour le nouveau processus.

Attention à la cascade de `fork` si on ne fait pas attention au processus dans lequel on l'appelle.

2 Ordonnancement

2.1 Politiques avec file

Dans ces politiques, on applique l'algorithme pour l'attribution d'un seul quantum et on le remet dans la file.
— *First Come First Served* : file classique
— *Shortest Job First* : file de priorité sur la durée d'exécution
— Priorité définie par l'utilisateur (avec vieillissement pour éviter la famine).
— *PAPS* : file de priorité élu par priorité puis par ordre d'arrivée.

2.2 Politiques de type tourniquet

Dans ces politiques, on reprend les précédentes, mais en planifiant à l'avance les quantums pour que tous les processus aient un quantum prévu (attente bornée). Elles permettent d'éviter la famine.
— *Round Robin* : découpe en quantum de temps et les distribue de manière uniforme aux processus.
— *Round Robin within priority* : découpe en quantum de temps distribués en fonction des priorités.

2.3 Implémentations

Pour UNIX : *Round robin with priority* avec priorité dynamique (dépend du type d'interruption).

3 Sémaphore

3.1 Principe

Objectif : contrôler l'exécution de deux acteurs indépendants.

C'est un objet muni d'un *compteur* qui indique le nombre de jetons disponible ou en attente et d'une *file d'attente* de processus.

Opération (métaphore du panier) :
— **P** : prendre un jeton
— **V** : rendre un jeton

P bloque tant qu'il n'y a pas de jeton.

Attention situation d'interblocage et safety (variables partagées, etc)

3.1.1 Section critique

Sémaphore initialisé à 1, jeton pris au début de la section critique et relâché à la fin. Un seul accès possible à la fois.

3.1.2 Rendez-vous

`x` et `y` initialisés à 0.

```
PROCESSUS 1          PROCESSUS 2
P (x)                V (x)
V (y)                P (y)
```

3.1.3 Producteur consommateur

Syncronisation entre une source et un client. Exemple d'utilisation : les pipes. On trouve généralement une version améliorée de ce principe avec plusieurs cases, en initialisant le sémaphore du producteur à N et du consommateur à 0. Note : c'est un rendez-vous avec une opération au milieu.

```
PRODUCTEUR           CONSOMMATEUR
P (x)                P (y)
ecrire donnees       read data
V (y)                V (x)
```

3.1.4 Barrière

`x` initialisé à 1, `barriere` à 0.

```
P (x)
if (nb < N-1)
    nb++
else for (i=1 to N)
    V (barriere)
    nb--
V (x)

P (barriere)
```

3.1.5 Lecteur-écrivain

`o,x,y` initialisé à 1.
`LECTEUR`

```
P (o)
V (o)

P (x)
    if nb_lecteur = 0
        P (y)
    nb_lecteur++
V (x)

lecture
```

```
P (x)
    if nb_lecteur = 1
        V (y)
    nb_lecteur--
V (x)
```

ECRIVAIN

```
P (o)
P (y)

ecriture

V (o)
V (y)
```

3.1.6 Dangers à vérifier

- Famine (attente indéfinie)
- Interblocage
- Taille des sections critiques

3.2 Solutions logicielles

Dans la suite, $i = 0$ et $j = 1$ désignent les deux processus.

3.2.1 Algorithme de Dekker

L'idée est d'introduire un ordre pour distinguer les deux processus.

```
etat[i] = 1

while etat[j] = 1
    if tour = j
        etat[i] = 0
        while tour = j {}

        etat[i] = 1;
tour = j
etat[i] = 0
```

3.2.2 Algorithme de Peterson

On manipule une variable par processus pour l'accès.

```
etat[i] = 1
tour = j

// Faire passer j s'il veut y aller
while (etat[j] = 1)
   && (tour == j) {}

// section critique

etat[i] = 0
```

3.3 Pipe

Modèle producteur-consommateur, deux utilisations en console :

- `programme-A | programme-B`, envoie la sortie de `programme-A` vers `programme-B`
- `programme > fichier`, écrit la sortie de programme vers le fichier `fichier`

Tout ce qui devait être écrit dans la sortie console ne s'affiche plus et est écrit vers la destination. [**partiel 2014, Q3, processus et fichier**]

4 Signaux

Pour ignorer des signaux, on utilise `signal` avec le numéro du signal et l'argument `SIG_IGN`. Sinon, `signal (sig, &fonction)` avec `fonction` une fonction de type `void fonction (int)` ;

On utilise `kill` pour envoyer des signaux :

- `SIGKILL` (9)
- `SIGALRM` (14)
- `SIGUSR1` (16)
- `SIGUSR2` (17)

5 Gestion des fichiers

5.1 Généralités

Les informations sur les fichiers sont stockées dans un bloc nommée *i-list*. Chaque fichier est constitué de plusieurs blocs. Sous système POSIX (UNIX, Linux, ...), même les périphériques sont exposés comme des fichiers.

5.2 Alignement mémoire

Les éléments des structures en mémoire sont alignées sur des multiples de 2, 4, 8, ...

```
struct noalign {
    char c;      // 1
    double d;    // 8
    int i;       // 4
    char c3[3]; // 3
}
```

Ici, le système rajoute 7 octets de rembourrage entre c et d, et un octet après $c3$, ce qui fait une structure de 24 octets. En mettant c en dessous de $c3$, on corrige le problème, et la structure n'en fait plus que 16 octets.

6 Partitions et mémoire

6.1 Politiques

On combine généralement ces politiques avec un ramasse-miette qui défragmente les partitions.

- *First fit* : on utilise la première partition libre de taille suffisante.
- *Best fit* : on utilise la partition de taille la plus proche.
- *Worst fit* : on utilise la partition de taille la plus grande.

Knuth : Il y a toujours un tiers de blocs libres par rapport aux blocs occupés avec ces politiques.

6.2 Mémoire virtuelle et pagination

Les adresses sont constitués de deux parties : l'*adresse physique* de la page et le *déplacement* dans cette page (qui doit aller jusqu'à la taille de la page). Pour les processus, il y a également une numérotation des *pages logiques*, que le système associe ensuite à des pages physiques. Cela permet d'isoler les processus entre eux et abstraire la mémoire en virtualisant l'espace d'adressage.

Méthode d'accès : on transforme l'adresse de logique à physique, puis on récupère la donnée dans la page physique.

6.3 Allocation des pages

Pour les exercies sur les allocations, on fait un tableau page virtuelle à gauche, temps discret en haut, et page physique dans les cases.

6.3.1 Stratégies

- *Least recently used* : on place les plus anciennement utilisées sur la mémoire lente.
- *Least frequently used* : on place les moins utilisées sur la mémoire lente.
- *First in first out* : on place les dernières créées sur la mémoire lente.

7 Mémo fonctions

7.1 Processus

```
int fork ()
```

Crée un nouveau processus. Retourne son pid dans le processus père, 0 dans le fils, ou -1 en cas d'erreur.

```
int wait (int pid)
```

Attend la fin du processus de pid `pid`. Retourne 0 si pas d'erreur, ou le numéro d'erreur.

```
int execv (char* filename, char* argv
    [])
```

Existe sous plusieurs formes. Remplace le code à exécuter par le processus et la mémoire par celui du fichier, en remplissant le `argv` du `main` du programme cible. Ne retourne qu'en cas d'erreur *avant* le lancement.

7.2 Sémaphores

```
sem_t*  sem_open (const char* name,
    int options)
```

Déclare une nouvelle sémaphore de nom `name` (au format "/nom"). `options` vaut `O_CREAT` ou `O_EXCL`, seul le 1er est utile ici.

```
int sem_wait (sem_t* sem)
```

Équivalent de l'opération **P**, prend un jeton ou bloque. Retourne 0 si aucune erreur, -1 sinon.

```
int sem_post (sem_t* sem)
```

Équivalent de l'opération **V**, rend un jeton et débloque un processus. Retourne 0 si aucune erreur, -1 sinon.

7.3 Signaux

```
int kill (int pid, int signal)
```

Envoie le signal `signal` au processus `pid` Retourne 0 si aucune erreur, -1 sinon.

```
func signal (int sig, func f)
```

Associe *pour un appel* le signal `sig` à la fonction `f`. La valeur de retour n'a pas d'importance, et il faut bien donner *l'adresse de la fonction.* (`&mafonction`).

```
void alarm (int seconds)
```

Supprime la dernière alarme, et si `seconds` $\neq 0$ rajoute une alarme qui lancera le signal `SIGALRM` dans `seconds` secondes. Retourne le nombre de secondes avant le déclenchement de l'alarme précédente.

```
int sigsetjmp (jmp_buf env, int
    sigmask)
```

Enregistre l'état du programme dans le contexte `env`. Généralement `mask` $= 0$, mais il faut réappeler `signal`. `env` doit être accessible, le déclarer en global (à l'extérieur des fonctions) pour l'utiliser avec des signaux. Attention valeur retour : retourne 0 ou -1 selon si pas d'erreur, sauf si l'on revient à cette instruction avec un saut. Dans le dernier cas, retourne l'entier passé en paramètre de `siglongjmp`, ce qui permet de savoir si on a sauté et d'où.

```
int siglongjmp (jmp_buf env, int val)
```

Saute à l'instruction `sigsetjmp` sauvegardé dans `env` en lui faisant retourner `val`.

7.4 Fichiers

```
int open (char* filename, int mode)
int fopen (char* filename, char* mode)
```

`open` ouvre le fichier `filename` avec le mode `mode`. `mode` peut valoir `O_RDONLY`, `O_WRONLY`, `O_RDWR`, mais on peut utiliser des flags plus complexes comme `O_CREAT`, `O_EXCL`, `O_APPEND`...Pour `fopen`, `mode` prend "r", "w", "a" et leur version multimode avec un +.

```
int read (int fichier, char* buffer,
    unsigned size)
```

Lit `size` caractères du fichier *ouvert* `fichier` et les écrit dans le tampon mémoire `buffer`. `buffer` *doit* être initialisé à la bonne taille. Renvoie une valeur strictement positive si pas d'erreur.

```
int write (int fichier, char* buffer,
    unsigned size)
```

Écrit `size` caractères du fichier *ouvert* `fichier` en les récupérant du tampon mémoire `buffer`. Renvoie une valeur strictement positive si pas d'erreur.

```
int lseek (int fichier, int offset,
    int mode)
```

Déplace le curseur sur le fichier `fichier` de `offset` caractères ou bytes selon le mode. `mode` peut prendre les valeurs `SEEK_CUR` (position actuelle), `SEEK_SET` (début), Retourne la position du pointeur sur le fichier, ou -1 en cas d'erreur.

```
int lockf (int fichier, int mode, int
    size)
```

`mode` peut valoir `F_LOCK` ou `F_ULOCK`. La synchronisation est assurée par le système, mais pas obligatoire.

```
int pipe (int pipefd[2])
```

Crée un pipe et remplie le tableau avec deux descripteurs de fichiers. `pipefd[0]` est réservé à la lecture, et `pipefd[1]` à l'écriture. Retourne 0 ou -1 si une erreur apparait.