# TD n° 17 : Ordonnancement

Dans ce TD, on se propose de modéliser le partage par plusieurs processus du temps de calcul sur un processeur, dans une version simplifiée de l'algorithme de ROUND-ROBIN, dit aussi algorithme du *tourniquet*.

## 1 Processus

On peut considérer grossièrement qu'un processus est un programme en train de s'exécuter, même si la réalité est plus compliquée. Pour simplifier, on pourra imaginer qu'il correspond au lancement d'un exécutable, comme par exemple : `ls`, `emacs`, `firefox`, etc. Un processus est identifié de manière unique par un entier : son *identifiant* de processus ou PID (pour **P***rocess* **ID***entifier*).

On introduit le type structuré suivant :

C

```c
struct processus {
  char *exec;  // nom de l'exécutable
  int pid;     // identifiant du processus
};

typedef struct processus processus;
```

On suppose qu'on dispose d'une fonction `lance_processus` qui prend en argument un nom d'exécutable et qui se charge de lancer un processus, en lui attribuant un PID unique et en renvoyant une structure processus où les deux champs `exec` et `pid` ont été initialisés. On suppose que l'on dispose également d'une fonction `est_fini` qui permet de savoir si l'exécution du processus est terminée ou non, ainsi que d'une fonction `arrete` qui met fin à un processus (que celui-ci soit terminé ou non) et qui libère la mémoire associée à ce processus `p`. Une implémentation fictive qui permet de tester les fonctions est proposée sur le site du cours.

C

```c
processus *lance_processus(char *exec);
bool est_fini(processus *p);
void arrete(processus *p)
```

## 2 Processeur et ordonnanceur

L'entité d'un système qui s'occupe de l'ordre dans lequel les processus sont exécutés s'appelle un *ordonnanceur*. On modélise la file d'attente des processus en cours par une liste chaînée circulaire (voir Fig. 1) :

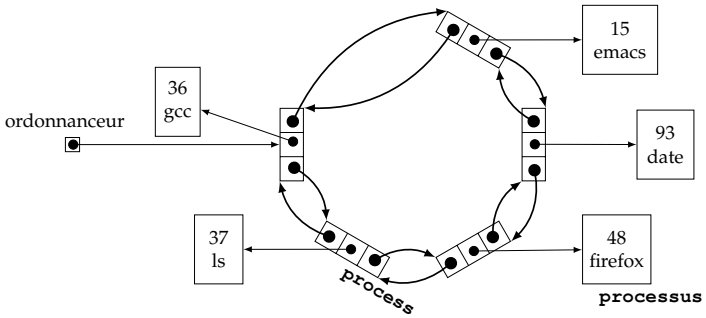


FIGURE 1 – File d'attente des processus. Le processus `gcc` de PID 36 est le premier dans la file, suivi du processus `ls`. Le dernier processus de la file, de nom `emacs`, pointe vers le tout premier, de nom `gcc`.

✔

Dans une liste circulaire doublement chaînée, la première cellule possède une référence vers la dernière cellule (c'est sa cellule précédente) et la dernière cellule possède une référence vers la première (c'est sa cellule suivante).

On propose la structure C suivante :

C

```c
struct process {
    processus *actif;
    struct process *suivant;
    struct process *precedent;
};

typedef struct process process;
```

Chaque processus est référencé par un maillon de la liste. Ce maillon contient en plus une référence vers le maillon précédent et une référence vers le maillon suivant.

On modélise un ordonnanceur par un pointeur sur un maillon de la liste, de type `process*`. Pour pouvoir modifier ce pointeur, en particulier lorsque l'ordonnanceur avance dans la file des processus, il est nécessaire de le passer par adresse aux fonctions, qui prendront donc en argument un pointeur sur un pointeur vers un maillon, de type `process**`.

1. Écrire une fonction `void ps(process **ordonnanceur)` qui affiche la liste des processus (avec leur identifiant et leur nom). Par exemple, dans la situation de la figure 1, un appel à `ps` provoque l'affichage :

```
PID   CMD
36    gcc
37    ls
48    firefox
93    date
15    emacs
```

Remarquons qu'ici il est inutile de passer un pointeur sur l'ordonnanceur (qui est lui-même un pointeur), puisque l'on n'a pas besoin de le modifier.

2. Écrire une fonction `void ajoute_process(process **ordonnanceur, processus *p)` qui ajoute un nouveau processus `*p` dans la file d'attente d'un ordonnanceur `*ordonnanceur`. Ce nouveau processus doit être ajouté à la fin de la file pour ne pas doubler les processus existants. Attention à bien gérer le cas où aucun processus n'était présent avant l'ajout, ce qui nécessite de modifier `*ordonnanceur` et qui justifie son passage via un pointeur. Par exemple, dans la situation de la figure 1, un appel à `ajoute(&ordonnanceur, lance_processus("jupyter"))` mène à la situation de la figure 2.

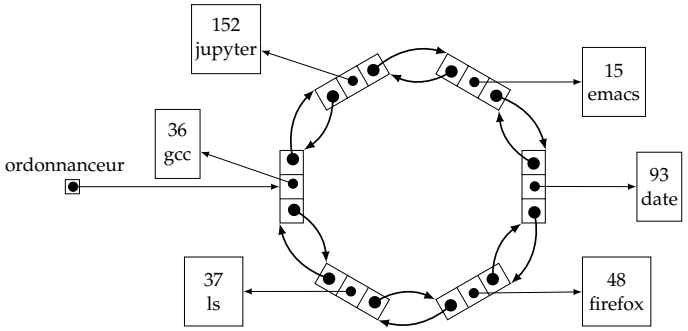


FIGURE 2 – Ajout du processus `jupyter` de PID 152 à la fin de la file.

3. Écrire une fonction `void delete(process *maillon)` qui supprime un maillon de la file des processus et qui arrête le processus correspondant. On suppose ici que ce maillon n'est pas celui pointé par l'ordonnanceur (et donc qu'il y a moins un autre maillon). Il est important de bien comprendre pourquoi, avec cette hypothèse, il est suffisant d'avoir un pointeur vers ce maillon pour être capable de le supprimer. Quelle mémoire est-il également nécessaire de libérer ?
4. Écrire une fonction `void delete_current(process **ordonnanceur)` qui supprime le maillon pointé par l'ordonnanceur. On suppose que ce maillon existe et donc qu'il y a au moins un processus dans la file (mais pas nécessairement plusieurs). Il faut également bien comprendre pourquoi ici il est nécessaire d'avoir un pointeur sur l'ordonnanceur et pas simplement un pointeur sur le maillon en cours.

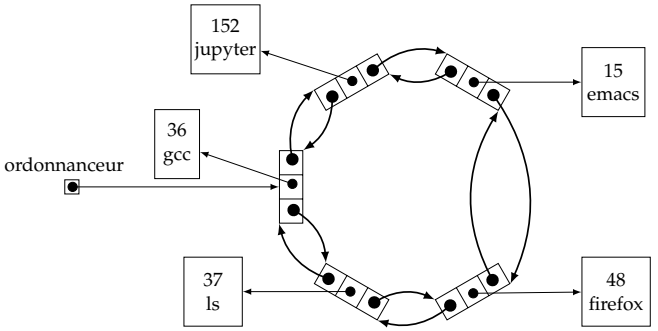


FIGURE 3 – Suppression du processus `date` de PID 93.

5. Écrire une fonction `void kill(process **ordonnanceur, int pid)` qui arrête le processus d'identifiant `pid`, s'il existe, en n'oubliant pas de le retirer de la liste des processus. On rappelle que le `pid` d'un processus est unique à un instant donné. La fonction n'a pas d'effet s'il n'existe pas de processus avec le bon identifiant. Par exemple, Dans la situation de la figure 2, un appel à `kill(&ordonnanceur, 93)` mène à la situation de la figure 3.
6. Écrire une fonction de prototype `void killall(process **ordonnanceur, char* exec)` qui arrête tous les processus de nom `exec`, s'il en existe. On pourra utiliser la fonction définie dans l'en-tête `<string.h>` de prototype `int strcmp(const char *s1, const char *s2)` qui renvoie un entier strictement négatif si la première chaîne est strictement inférieure à la deuxième, nul si les deux chaînes sont égales et strictement positif sinon.

## 3 Algorithme Round Robin

Le principe du partage du temps de calcul est le suivant : le processeur dispose d'une liste de processus auxquels il accorde, à tour de rôle et dans l'ordre, un temps de calcul fixé. Cette unité de temps allouée successivement à chaque processus est appelée *quantum* de temps. La file d'attente est gérée comme une file circulaire. L'ordonnanceur parcourt cette file et alloue un temps processeur à chacun des processus pour un intervalle de temps de l'ordre d'un quantum au maximum. Si un processus a terminé son travail dans le temps imparti, il disparaît de la liste des processus existants.

On suppose que l'on dispose d'une fonction `cpu_quantum` qui alloue un quantum de temps processeur à un processus passé en paramètre.

C
```c
void cpu_quantum(processus *p);
```

7. Écrire une fonction `void round_robin(process **ordonnanceur)` qui simule l'algorithme de ROUND-ROBIN exposé ci-dessus, c'est-à-dire qui prend en paramètre un pointeur sur un ordonnanceur et qui alloue successivement et dans l'ordre un quantum de temps à chaque processus de la file d'attente, tant qu'il y a des processus dans la file. Un processus qui a terminé doit être arrêté et sorti de la file. Par exemple, la première étape de cet algorithme va allouer au processus `gcc` un quantum de temps (via un appel à `cpu_quantum`). Supposons que ce processus n'est pas terminé à la fin de ce quantum de temps, alors, on se retrouve dans la situation décrite à la figure 4. Si le processus `gcc` avait terminé dans le temps imparti, il aurait été supprimé.

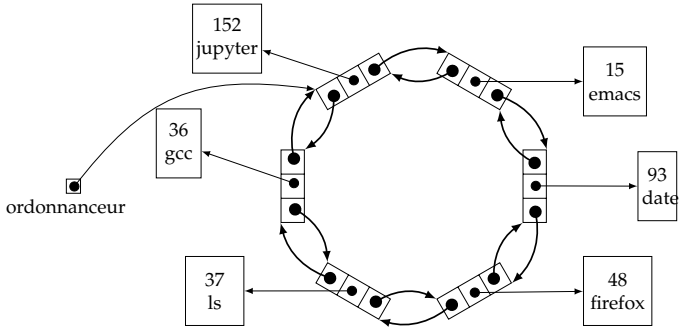


FIGURE 4 – Déplacement du processus courant après allocation d'un quantum de temps au premier processus, en supposant que ce processus n'est pas encore terminé.

☞ **Remarque 1**

*Le système du tourniquet prend son nom du jeu de parcs pour enfants. L'image est que chaque processus est assis sur le tourniquet et, chacun à son tour, ne fait que passer devant le processeur pendant un laps de temps fini.*