

Ordonnancement des processus

Motivations : - Équité
- Gain en temps

Objectif : équilibre

Critères pour un "bon" ordonnancement :

- taux occupation CPU
- nb de processus exécutés en un temps donné
- temps pour exécuter un processus
- temps d'attente d'un processus dans la file prêt
- temps de réponse

Ordonneur

Definition

L'ordonneur est un algorithme qui élit un processus. Ce processus a le privilège d'accéder au CPU.

2 types d'ordonnisseurs :

- non préemptif : sélectionne un processus qui s'exécute jusqu'à ce qu'il bloque ou qu'il libère le processeur
- préemptif : sélectionne un processus et l'exécute durant un laps de temps donné. Si le processus est encore actif après ce délai, il est suspendu et l'ordonneur élit un nouveau processus.

4

Différents ordonnanceurs :

- non préemptifs : - FIFO first in first out (1^e soumis, 1^e exécuté)
- SJF shortest job first (le + court d'abord)
- L JF longest job first (— long —)

- préemptifs : - SRTN shortest remaining time next (au plus court reste)
- Round Robin : algorithme du tournoi
(notion de Quantum)

$$T=1s : P_1$$

$$T=2s : P_2$$

$$T=5s : P_3$$

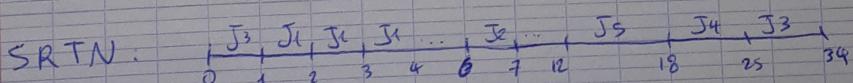
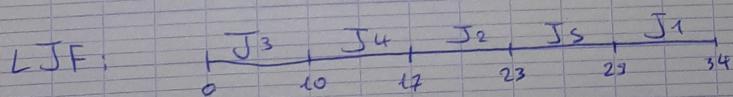
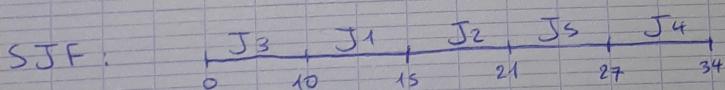
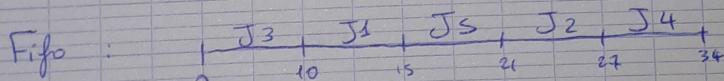


- priorité : un algo round robin profile de priorité

Exemple :

On considère 5 processus :

Jobs	Durée	Soumission	Priorité
J ₁	5s	t = 1s	4
J ₂	6s	t = 3s	5
J ₃	10s	t = 0s	1
J ₄	7s	t = 3s	4
J ₅	6s	t = 2s	5

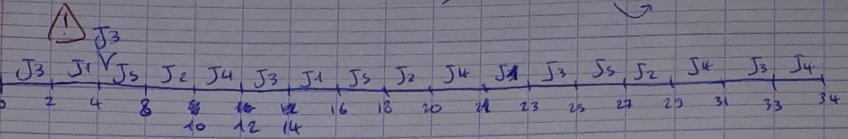


1 4 5 5 4

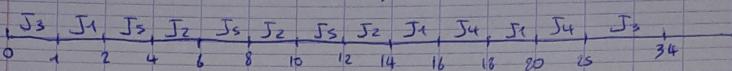
l'ordre

J₃ J₄ J₅ J₂ J₄

Round Robin : (Quantum 2s)



Priorité (Quantum 2s)



Programme principale et arguments:

→ programmation système : langage C

→ fonction principale : main()

int main (int argc, char * argv [])

argc : contient le nombre d'arguments passés

en paramètre par l'interpréteur de

commande au programme appelé

argv : contient les arguments au format txt

ex: \$./prog arg1 arg2 arg3 428 toto

argc = 6

argv [0] = " ./prog "

—— [1] = " arg1 "

:

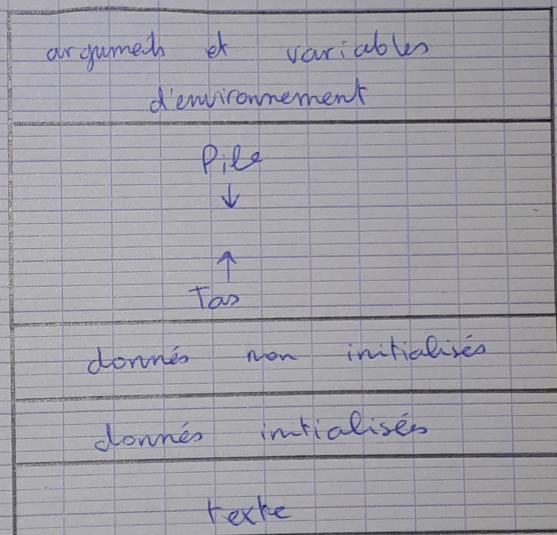
—— [5] = " toto "

Schéma mémoire d'un processus

la mémoire d'un processus est constituée :

- d'un segment texte. Ce sont les instructions exécutées par le processeur. Le segment est généralement en lecture seule et peut être partagée

- Segment de données initialisés (ex : int max = 55)
- Segment de données non initialisés (ex : int tab[100])
les blocs sont alors initialisés à 0 ou NULL
- Pile : sert à la sauvegarde d'informations en particulier lors de l'appel de fonction (et au retour) et au stockage de données
- Tas (pour l'allocation dynamique)
ex : int * t; : t est dans les données non init
 $t = \text{malloc}(10 * \text{sizeof}(\text{int}))$; : t indique où on peut accéder à l'adresse du le tas



la commande size fournit la taille des segments

\$ size /bin/ls

txt	data	bss	others	dec	hex
24576	4096	0	7884	36556	8000

Primitives mémoire :

→ void * malloc (size_t s) : alloue une zone mémoire de la taille spécifiée l'adresse de dépôt est renournée. le contenu est arbitraire

ex: `int * p;` $\boxed{P} \rightarrow \text{----}$
`p = (int *) malloc(5 * sizeof(int));` $\boxed{P} \rightarrow \boxed{\text{|||||}}$

\rightarrow `void * calloc (int n, size_t s)`

ex: `int * p;` $\boxed{P} \rightarrow \text{----}$
`p = (int *) calloc (5, sizeof(int));` $\boxed{P} \rightarrow \boxed{\text{|||||}}$

\rightarrow `void * realloc (void * ptr, size_t s)`

ex: `p = realloc (p, 2 * sizeof(int))`
 $\boxed{P} \rightarrow \boxed{\text{|||||}}$

\rightarrow `void free (void * ptr)`

ex: `free (p);` $\boxed{P} \rightarrow \boxed{\text{----}} \Rightarrow \boxed{P} \rightarrow \text{----}$

Variables d'environnement

- `char * getenv (const char * name)`
- `int putenv (const char * str)`
- `int setenv (const char * name, const char * value, int value)`
- `void unsetenv (const char * name)`

variables:

HOME	rep. perso
LOGNAME	login
PATH	préfixe par executable
TERM	type de terminal

[S]

cours 4]

Les limitations sur les ressources :

- int getrlimit (int ressource, struct rlimit * rptr)
- int setrlimit (int ressource, const struct rlimit * rptr)
- struct rlimit {
 - * rlim_cur rlim.cur; /* limite courante */
 - rlim_max rlim_max; /* limite max */

Ressources

- RLIMIT_CORE : taille max du fichier core généré en cas d'erreur fatale
- RLIMIT_CPU : maximum du temps CPU en secondes
- RLIMIT_DATA : taille maximale du segment de données (initialisées ou non initialisées)
- RLIMIT_FSIZE : taille max des fichiers créés
- RLIMIT_NOFILE : nombre max de fichiers ouverts
- RLIMIT_NPROC : nb max de processus enfants
- RLIMIT_STACK : taille max de la pile.

Ex: afficher le nb max de fichiers ouverts pour le processus courant et fixer le nb max de processus enfants à 2 :

```
struct rlimit {  
    rlimit RLIMIT_NOFILE;  
    rlimit RLIMIT_NPROC
```

3

```
main () {
```

```
    getrlimit (0,  
    int main (int argc, char * argv [3]) {  
        struct rlimit p;  
        getrlimit (RLIMIT_NOFILE, &p)
```

```
printf("courant: %d\n", max : %d", p.rlim.cur,  
      p.rlim.max);  
p.rlim.cur = 2;  
p.rlim.max = 2;  
set_rlimit(RLIMIT_NPROC, &p);  
exit(0);
```

3

identifier un processus :

pid_t getpid();	→ id processus
pid_t getppid();	→ id _____ père
uid_t getuid();	→ id de l'utilisateur
uid_t geteuid();	→ _____ effectif
gid_t getgid();	→ id du groupe
gid_t getegid();	→ _____ effectif

création de processus :

```
pid_t fork(void);
```

crée un nouveau processus, la valeur du retour est :

- 0 dans le processus fils
- le pid du processus ds le processus père

Exemple : écrire un programme qui crée 5 processus fils → le processus père affiche les pid des processus créés

→ chaque processus fils affichera son pid

```

#include <unistd.h>
int main (int argc, char * argv [3]) {
    pid_t p;
    for (int i=0; i<s; i++) {
        if (p=fork ()) ==0) {
            printf (">d\n", getpid ());
            exit (0);
        }
    }
    printf
    exit (0);
}

int main (int argc, char * argv [ ]) {
    pid_t p; int i;
    for (i=0; i<s; i++) {
        if ((p=fork ()) ==0) {
            printf ("fils : %d\n", getpid ());
            exit (0);
        }
        printf ("pere %d\n", p);
    }
    exit (0)
}

```

Terminaison de processus :

void exit (int status);

effectue un "ménage" et quitte le processus avec le status status.

void _exit (int status);

quitte le processus avec pour status status

int atexit (* func (void)); spécifie les opérations à réaliser lors de la terminaison du processus

pid_t wait(int * statloc);

attend la fin d'un processus fils et récupère sa valeur de retour (dans la var statloc)
retourne le pid du fils terminé

pid_t waitpid(pid_t pid, int * statloc, int options);

attend la fin du processus fils pid, et récupère sa valeur de retour avec statloc. Les options servent éventuellement à avoir un appel non bloquant.

Exemple: écrire un programme qui crée 10 processus fils qui afficheront leurs identifiants. Le processus père affichera les id des processus fils dans l'ordre dans lequel ils se terminent :

```
int main (int argc, int argv [3]) {
    int i; pid_t p; atexit int s;
    for (i=0; i<10; i++) {
        if ((p=fork()) == 0) {
            printf ("fils : %i\n", getpid());
            exit(0);
        }
    }
    while (i<10) {
        p=wait (atexit);
        printf ("père : %i", statloc : %i\n", p, s);
        i++;
    }
    exit(0);
}
```