CONSERVATOIRE NATIONAL DES ARTS ET METIERS

Durée : 2 heures

METHODES DE PROGRAMMATION SYSTEMES

UE NSY103

CORRECTION DE L'EXAMEN
Année 2008 – 2009, deuxième semestre
Coefficient : 2/3

Documents autorisés : NON

Cnam
CONSERVATOIRE NATIONAL
DES ARTS ET METIERS

Exercice n°1 : thread *(3 questions pour un total de 6 points)*

Question 1.a) [*1,5 point*] : Le concept de thread permet d'effectuer de la multiprogrammation (plusieurs programmes se partagent le temps CPU), au même titre que les processus. Seulement, les processus souffrent de trois défauts majeurs :

- la création d'un nouveau processus (avec la création d'un nouveau PCB, la mise en œuvre de segments de mémoire isolés, etc.) est coûteuse, même si le « *copy on write* » permet d'alléger ce mécanisme ;
- le fait que deux processus aient deux espaces mémoire isolés rend la commutation de contexte coûteuse ;
- et pour la même raison (espaces mémoire isolés), les mécanismes de communication inter-processus sont eux aussi coûteux (recopie d'information par exemple).

Or, parfois, les programmeurs ont besoin de faire de la multiprogrammation, sans qu'il soit nécessaire (bien au contraire) que les deux tâches aient des espaces mémoire isolés. Les threads (ou processus légers) ont été inventés dans ce but : proposer un mécanisme qui permet de créer plusieurs tâches au sein d'un même processus. Ces tâches partagent le même code et les mêmes données (tas, variables globales, constantes, etc.). Seule la pile à l'exécution et l'état des registres sont spécifiques à chaque thread.

Question 1.b) [*1,5 point*] : Il existe deux types de thread : les threads utilisateur et les threads noyau.

Les threads utilisateurs sont implémentés dans une bibliothèque de fonctions qui s'exécutent en mode utilisateur. Le système d'exploitation (et en particulier l'ordonnanceur) n'a pas connaissance de la notion de thread. L'ordonnancement des différentes tâches des différents threads est assurée par un ordonanceur spécifique, qui s'exécute dans le processus en mode utilisateur (sans privilège système).

Les threads noyaux sont implémentés dans le système d'exploitation. L'ordonnanceur du système d'exploitation doit connaître la notion de thread, et doit gérer lui-même l'ordonnancement des tâches des différents threads au sein d'un processus.

Avantages des threads utilisateurs :

- ils permettent, dans les anciens systèmes d'exploitation, de proposer aux programmeurs le mécanisme de thread même quand le système d'exploitation ne proposait pas ce concept ;
- souvent, la bibliothèque de fonctions qui assure l'ordonancement des différents threads au sein d'un processus ne nécessite pas un coûteux passage en mode "privilégié".

Leur inconvénient : si un thread effectue un appel système bloquant, ce sont tous les threads du processus qui se retrouvent bloqués.

Question 1.c) [3 *points*] : Tout d'abord, le programmeur utilise les mutex `Mut1` et `Mut2` pour protéger l'accès à des variables locales. Or, ces variables sont stockées dans la pile à l'exécution. Sachant que chaque thread possède sa propre pile à l'exécution, ces variables ne sont pas partagées. D'ailleurs, le compilateur fournira une erreur " variables `compteur_commun1` et `compteur_commun2` inconues dans la fonction `second_thread()` ". La solution : ressortir les déclarations de ces deux variables dans le corps du programme, en dehors de toute fonction, afin que ces variables soient globales. Elles seront alors connues de toutes les fonctions, et leurs valeurs sera partagée entre les deux threads (d'où l'intérêt de les protéger par des mutex).

Ensuite, un des threads positionne un verrou `Mut1` avant de verrouiller `Mut2`. Inversement, le second thread verrouille `Mut2` avant `Mut1`. Il peut donc y avoir interblocage, ou **étreinte mortelle** entre les deux threads (s'il y a préemption d'un tread entre deux verrouillages, et que l'autre thread positionne ses deux verrous). Dans le cas présent, comme il n'y a que deux threads et deux ressources à protéger, la solution est simple : il suffit de toujours verrouiller les mutex dans le même ordre (par exemple en intervertissant les lignes " `pthread_mutex_lock(&Mut2);` " et " `pthread_mutex_lock(&Mut1);` " dans la fonction `second_thread()`.

<u>Exercice n°2</u> : *ordonnancement temps-réel (3 questions pour un total de 7 points)*

Question 1.a) [1,5 *point*] :

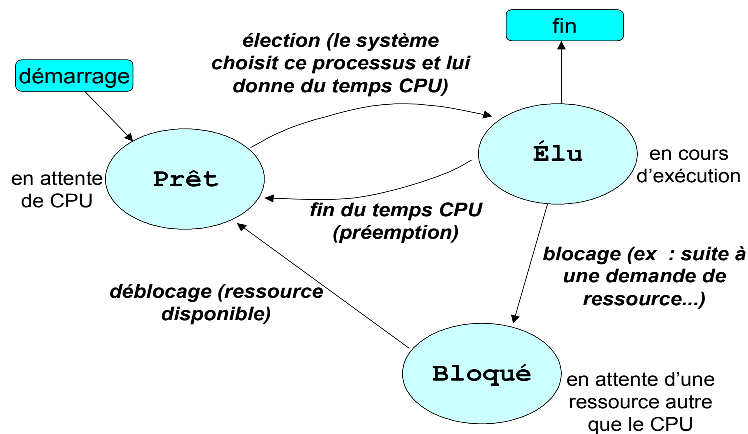


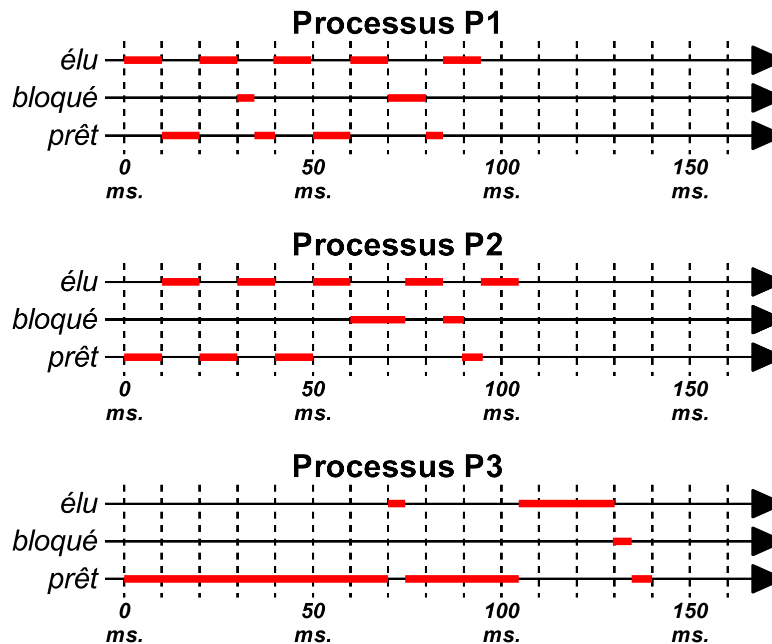
Diagramme d'état d'un processus Unix/Linux

Question 1.b) [1,5 *point*] :

L'algorithme du tourniquet (round robin) sous Linux utilisé pour la classe d'ordonnancement « `SCHED_RR` » élit le processus (ou un des processus) ayant la priorité **la plus élevée**. Lorsqu'un processus est élu, il s'exécute **durant un quantum de temps**, avant d'être préempté (sauf s'il a lancé une fonction système bloquant entre temps, ou si un processus de priorité plus élevé est élu).

Une file d'attente des processus de même priorité permet de lancer les processus les uns à la suite des autres pour un temps donné.

Question 1.c) [*4 points*] :



Temps d'exécution de P1 = 95 ms.

Temps d'exécution de P2 = 105 ms.

Temps d'exécution de P3 = 140 ms.

<u>Exercice n°3</u> : *sculpteur (une question pour un total de 7 points)*

Question 3.a) [7 *points*] :

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <signal.h>
#include <wait.h>
#include <unistd.h>

extern int action_chimique();
extern int action_laser();
extern int action_mecanique();

int pid_fils[3];
int tubes[3][2];

/* Fonction appelée à la réception d'un signal SIGUSR1 par */
/* le processus chimique */
void    reveille_fct_chimique(int p)
{
      /* on ré-arme */
      signal(SIGUSR1, reveille_fct_chimique);
      switch(action_chimique())
      {
            case -1 :
                  /* le traitement suivant sera le laser */
                  kill(pid_fils[1], SIGUSR1);
                  break;
            case 0 :
                  /* fin de la pièce, on passe la main au père */
                  kill(getppid(), SIGUSR1);
                  break;
            case 1 :
                  /* le traitement suivant sera mécanique */
                  kill(pid_fils[2], SIGUSR1);
                  break;
      }
}

void fils_chimique()
{
      /* Récupère les PID de la fraterie */
      /* Tout d'abord, on ferme le pipe en écriture */
      close(tubes[0][1]);
      /* et on lit le tableau */
      read(tubes[0][0], pid_fils, sizeof(pid_fils));
      close(tubes[0][0]);
      /* on arme la fonction qui sera appelée à la réception de SIGUSR1 */
      signal(SIGUSR1, reveille_fct_chimique);
      /* Et on attend les signaux jusqu'à la venue de SIGUSR1 */
      while (1)
            pause();
}

/* Fonction appelée à la réception d'un signal SIGUSR1 par */
/* le processus laser */
void    reveille_fct_laser(int p)
{
      /* on ré-arme */
```

```
	signal(SIGUSR1, reveille_fct_laser);
	switch(action_laser())
	{
		case -1 :
			/* le traitement suivant sera chimique */
			kill(pid_fils[0], SIGUSR1);
			break;
		case 0 :
			/* fin de la pièce, on passe la main au père */
			kill(getppid(), SIGUSR1);
			break;
		case 1 :
			/* le traitement suivant sera mécanique */
			kill(pid_fils[2], SIGUSR1);
			break;
	}
}

void fils_laser()
{
	/* Récupère les PID de la fraterie */
	/* Tout d'abord, on ferme le pipe en écriture */
	close(tubes[1][1]);
	/* et on lit le tableau */
	read(tubes[1][0], pid_fils, sizeof(pid_fils));
	close(tubes[1][0]);
	/* on arme la fonction qui sera appelée à la réception de SIGUSR1 */
	signal(SIGUSR1, reveille_fct_laser);
	/* Et on attend les signaux jusqu'à la venue de SIGUSR1 */
	while (1)
		pause();
}

/* Fonction appelée à la réception d'un signal SIGUSR1 par */
/* le processus mécanique */
void	reveille_fct_mecanique(int p)
{
	/* on ré-arme */
	signal(SIGUSR1, reveille_fct_mecanique);
	switch(action_mecanique())
	{
		case -1 :
			/* le traitement suivant sera chimique */
			kill(pid_fils[0], SIGUSR1);
			break;
		case 0 :
			/* fin de la pièce, on passe la main au père */
			kill(getppid(), SIGUSR1);
			break;
		case 1 :
			/* le traitement suivant sera laser */
			kill(pid_fils[1], SIGUSR1);
			break;
	}
}

void fils_mecanique()
{
	/* Récupère les PID de la fraterie */
	/* Tout d'abord, on ferme le pipe en écriture */
	close(tubes[2][1]);
	/* et on lit le tableau */
	read(tubes[2][0], pid_fils, sizeof(pid_fils));
```

```
	close(tubes[2][0]);
	/* on arme la fonction qui sera appelée à la réception de SIGUSR1 */
	signal(SIGUSR1, reveille_fct_mecanique);
	/* Et on attend les signaux jusqu'à la venue de SIGUSR1 */
	while (1)
		pause();
}

void fin_programme()
{
	int i, ret;
	/* Fonction appelée par la réception d'un signal SIGUSR1 par le père */
	/* On suppose que la pièce est terminée ; donc, on tue les 3 fils, */
	/* et on récupère le code statut à l'aide de wait() */
	/* pour éviter les zombies. */
	for (i = 0 ; i < 3 ; ++i)
	{
		kill(pid_fils[i], SIGKILL);
		wait(&ret);
	}
	exit(0);
}

int main()
{
	int i;
	/* Initialisation des 3 pipes */
	for (i = 0 ; i < 3 ; ++i)
	{
		if (pipe(tubes[i]) < 0)
		{
			fprintf(stderr, "Erreur avec pipe()\n");
			exit(-1);
		}
	}
	/* On crée les 3 fils */
	for (i = 0 ; i < 3 ; ++i)
	{
		pid_fils[i] = fork();
		if (pid_fils[i] < 0)
		{
			fprintf(stderr, "Erreur de fork(); !!!\n");
			exit(-1);
		}
		if (pid_fils[i] > 0)
		{
			/* code des fils... */
			switch (i)
			{
				case 0 : /* fils chimique */
					fils_chimique();
				case 1 : /* fils laser */
					fils_laser();
				case 2 : /* fils mécanique */
					fils_mecanique();
			}
		}
	}
	/* Le père arme la fct qui tuera les fils une fois la pièce réalisée */
	signal(SIGUSR1, fin_programme);
	/* Pour ne pas s'embêter, on envoie aux trois fils le PID des 3 fils */
	/* en envoyant le tableau pid_fils en entier */
	for (i = 0 ; i < 3 ; ++i)
```

```
	{
		/* on ferme le pipe en lecture */
		close(tubes[i][0]);
		write(tubes[i][1], pid_fils, sizeof(pid_fils));
		close(tubes[i][1]);
	}
	/* On lance le traitement chimique */
	kill(pid_fils[0], SIGUSR1);
	/* Et on attend les signaux jusqu'à la venue de SIGUSR1 */
	while (1)
		pause();
	return(0);
}
```