CONSERVATOIRE NATIONAL DES ARTS ET METIERS

Durée : 2 heures

METHODES DE PROGRAMMATION SYSTEMES

UE NSY103 – NANCY/METZ

CORRECTION DE L'EXAMEN du 28 JUIN 2012

Année 2011 – 2012, deuxième semestre
Coefficient : 2/3

Correcteur : Emmanuel DESVIGNE

Documents autorisés : NON

Cnam
CONSERVATOIRE NATIONAL
DES ARTS ET METIERS

Exercice n°1 : processus, threads (5 questions pour un total de 7 points)

Question 1.a) [*1 point*] : Un processus est un programme exécuté par un ordinateur, défini par :

- un ensemble d'instructions à exécuter (le code),
- un espace d'adressage (qui lui est propre) en mémoire vive (RAM) pour y stocker ce code, la pile à l'exécution, les variables globales, les constantes, le tas (heap), etc.
- un ensemble de ressources affectées au programme (descripteurs de fichier, socket réseau, etc.).

Sous Unix/Linux, l'ensemble des propriétés d'un processus est stocké dans une structure appelée le « PCB » (Process Control Block).

Un processus peut être en cours de création (initialisation/démarrage), dans l'attente du CPU (prêt), en cours d'exécution (élu), bloqué dans l'attente d'une ressource ou d'un événement (bloqué), terminé et dans l'attente que son processus père vienne lire son code de retour (zombie) ou terminé. Ce qui peut se résumer par le schéma suivant :

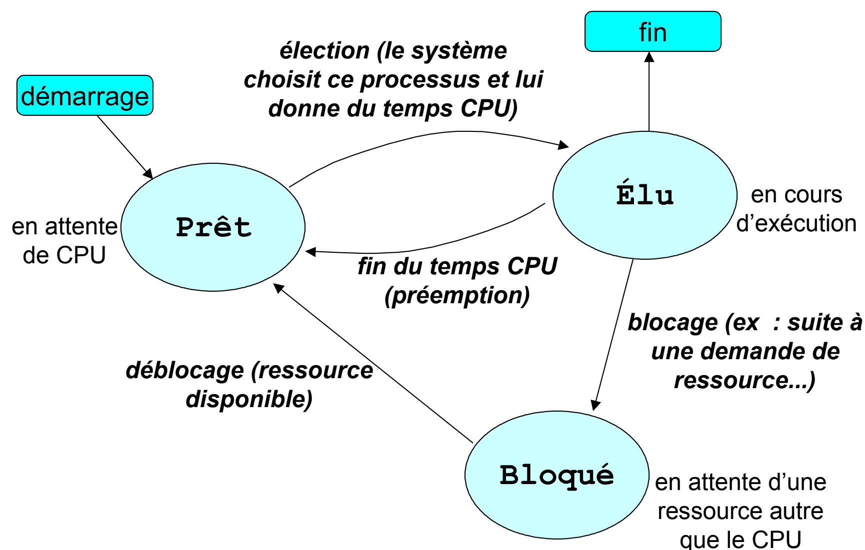


Diagramme d'état d'un processus Unix/Linux

Question 1.b) [*1 point*] : Un processus léger (thread) permet d'effectuer de la multiprogrammation au sein d'un même processus, ce qui assure une implémentation moins coûteuse en CPU :

- 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 plusieurs thread partagent le même espace mémoire allège leur création ;
- la préemption d'un thread et la reprise de l'exécution d'un autre thread au sein d'un même processus est bien moins coûteuse que la commutation de contexte entre deux processus (pas besoin d'isoler leur espace mémoire) ;
- et pour la même raison (espaces mémoire non isolés), les mécanismes de communication inter-thread sont moins coûteux (accès direct aux variables globales du processus par exemple).

Il existe deux types de thread :

- les **threads utilisateurs**, qui sont implémentés dans une bibliothèque de fonctions qui s'exécutent en mode utilisateur. Ils sont utilisés lorsque le système d'exploitation ne connaît pas ce concept. L'ordonnancement des différentes tâches des différents threads est alors assurée par un ordonnanceur spécifique, qui s'exécute dans le processus 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".

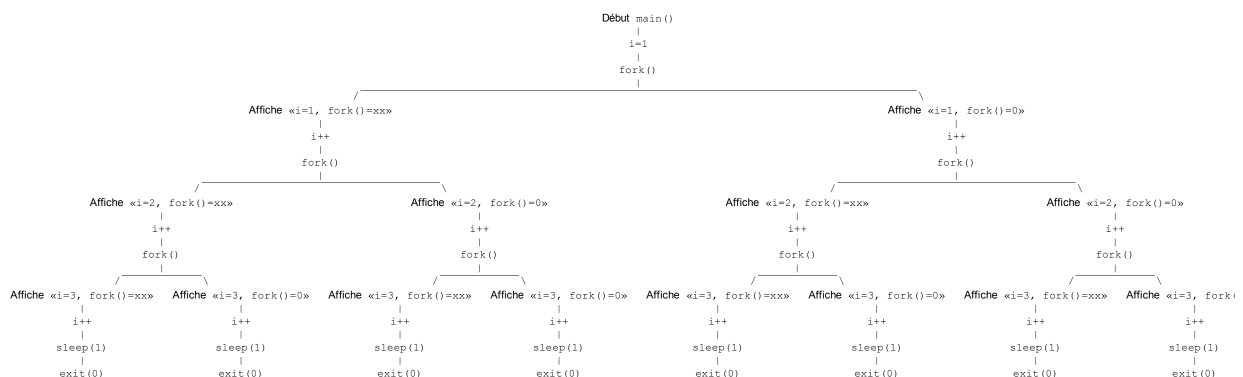
Leur inconvénient : si un thread effectue un appel système bloquant, ce sont tous les threads du processus qui se retrouvent bloqués.

Question 1.c) [*1 point*] : La composante du système d'exploitation qui gère l'affectation du CPU (gestion des états des processus, élection, etc.) s'appelle l'**ordonnanceur** (*scheduler* en anglais). Parmi les différents algorithmes utilisés pour programmer des ordonnanceurs, citons :

- l'algorithme du tourniquet (round robin) : pour cet algorithme, le temps est découpé tranches (on parle de « quantum de temps »). 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 bloquante entre temps). Une file des processus permet de lancer les processus les uns à la suite des autres (premier arrivé, premier servi) pour un temps donné.
- l'algorithme du premier arrivé, premier servi : il consiste à mettre en place une simple file d'attente, et à lancer les processus les uns à la suite des autres, dans leur ordre d'arrivée. Il n'y a pas de préemption : chaque processus s'exécute jusqu'à ce qu'il soit terminé, ou jusqu'à ce qu'il fasse appel à une primitive système bloquante.

[***Note du correcteur :*** *tout autre algorithme sera accepté, du moment que son principe est correctement énoncé*]

Question 1.d) [*2 points*] : Le programme initialise la variable `i` à 1 et lance un `fork()`. Chacun des deux processus résultant de cette opération (le père et le fils) vont avoir une variable locale `i` qui leur est propre, et dont la valeur est incrémentée. Puis, un nouveau `fork()` est exécuté par ces deux processus, ce qui fait qu'il y aura 4 processus. Le même mécanisme est répété, nous amenant ainsi à 8 processus. Le déroulement des opérations peut se schématiser ainsi :



On constate alors :

- que la ligne « `i=3, fork()=xxx` » (avec `xxx` un entier nul ou non nul) sera affichée 8 fois ;
- que la ligne « `i=xxx, fork()=0` » (avec `xxx` valant 1, 2, ou 3) sera affichée 7 fois ;
- et qu'au final, il s'affichera 14 fois une ligne du style : « `i=xxx, fork()=yyy` ».

La compilation et l'exécution du programme en laissant l'appel à la fonction donne le résultat attendu :

```
$ gcc question1.c -o question1
$ ./question1
i=1, fork()=2501
i=1, fork()=0
i=2, fork()=2502
i=2, fork()=2503
i=3, fork()=2504
i=3, fork()=2505
i=2, fork()=0
i=3, fork()=0
i=3, fork()=2506
i=3, fork()=0
i=3, fork()=0
i=2, fork()=0
i=3, fork()=2507
i=3, fork()=0
$
```

Question 1.e) [*2 points*] :

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

// La variable globale nb_fils contiendra le nombre de fils en
// cours d'exécution.
int     nb_fils = 0;

int main()
{
    int     i,j;
    for (i=1; i <= 3; i++)
    {
        j = fork();
        if (j == 0)
        {
            printf("Fils au PID %d, je fais un sleep(1)\n", getpid());
            sleep(1);
            printf("Je suis un fils au PID %d, je me termine\n", getpid());
            exit(0);
        }
        else
        {
            // Je suis le père, je note que j'ai un nouveau fils
            nb_fils++;
        }
    }
    // On récupère les codes de fin des fils pour éviter les zombies
    while (nb_fils > 0)
    {
        wait(NULL);
        nb_fils--;
        printf("Mort d'un fils\n");
    }
    printf("Le père a détruit tous les zombies. Fin du programme.\n");
    exit(0);
}
```

[***Note du correcteur :*** *la destruction des zombies est obligatoire pour avoir 2 points*]

Exercice n°2 : annuaire téléphonique *(3 questions pour un total de 7 points)*

Question 2.a) ***[1 point]*** Dans un système multiprogrammé, il faut absolument qu'un processus ait l'assurance d'être le seul à accéder à un fichier lors d'une écriture. Sinon, un autre processus lisant en même temps le fichier pourrait obtenir des données inconsistantes. Ou deux processus qui écriraient en même temps pourraient interférer l'un l'autre. Par contre, rien n'empêche à plusieurs processus de lire en même temps un même fichier.

Question 2.b) ***[2,5 points]*** Trois solutions peuvent être mises en œuvres pour la gestion de l'annuaire :

- une solution non centralisée avec les sémaphores (algorithme des **lecteurs rédacteur**). Avantage de cette solution : code efficace. Inconvénient : si plusieurs programmeurs développent diverses composantes de l'application, tous doivent respecter les mêmes règles (obligation d'utiliser les sémaphores comme convenu) ;
- une solution centralisée sous forme de **moniteur** : les ordres de lectures ou d'écritures sont envoyés à un processus unique, via une « API » proposée par le moniteur. Les mécanismes choisis afin de communiquer entre les processus (tubes, files de messages, etc.) assurent leur synchronisation ;
- une solution **client-serveur** utilisant les couches réseau. Ressemble au modèle du moniteur, excepté que clients et serveurs peuvent être potentiellement sur deux machines distinctes. La solution ne semble pas adaptée ici, les appels systèmes aux fonctions réseau seraient coûteux, alors que les processus sont sur une même machine.

Question 2.c) ***[3,5 points]*** La solution à cet exercice correspond à la résolution du problème lecteurs-rédacteur. On utilise une variable globale `nb_lect` qui contient le nombre de lecteurs à un instant T. Il faut ensuite utiliser deux sémaphores :

- un sémaphore `sem_nb_lect` qui protège l'accès à la variable globale `nb_lect`,
- et un sémaphore `sem_exclu` qui permet à un rédacteur ou à l'ensemble de tous les lecteurs de s'assurer l'exclusivité de l'accès au fichier.

Ces éléments sont initialisés au début du programme avec le code suivant :

```
int   nb_lect = 0 ;
Init(sem_nb_lect, 1);
Init(sem_exclu, 1);
```

Les fonctions « `ajouter_une_identite()` » et « `effacer_une_entree()` » sont des fonctions ayant toutes deux un rôle de rédacteur. Leur squelette est identique :

```
int ajouter_une_identite(bla bla bla)
/* ou int effacer_une_entree (bla bla bla) */
{
      int   code_retour;
      P(sem_exclu);
      /* portion de code d'accès au fichier en écriture */
      V(sem_exclu);
      return(code_retour);
}
```

Les fonctions « `annuaire_inverse()` » et « `quel_est_le_numero()` » ont exactement le même squelette, à savoir celui d'un lecteur :

```
int annuaire_inverse(bla bla bla)
/* ou int quel_est_le_numero(bla bla bla) */
{
      int   code_retour;
```

```
        /* je me garantis l'exclusivité de la variable nb_lect : */
        P(sem_nb_lect);
        nb_lect++;
        if (nb_lect == 1) /* je suis le premier lecteur */
        {
              /* je donne l'exclusion au groupe des lecteurs */
              P(sem_exclu);
        }
        /* je n'ai plus besoin d'accéder à nb_lect : */
        V(sem_nb_lect);

        /* ici, le coeur de la fonction annuaire_inverse() */
        /* ou de la fonction quel_est_le_numero()... bla bla bla ...  */

        /* je me garantis l'exclusivité de la variable nb_lect : */
        P(sem_nb_lect);
        nb_lect--;
        if (nb_lect == 0) /* je suis le dernier lecteur */
        {
              /* je libère l'exclusion au groupe des lecteurs */
              V(sem_exclu);
        }
        /* je n'ai plus besoin d'accéder à nb_lect : */
        V(sem_nb_lect);
        return(code_retour);
  }
```

<u>Exercice n°3</u> : mémoire, segmentation, pagination *(3 questions pour un total de 8 pts)*

Question 3.a) [*2 points*] : La **pagination** permet la mise en œuvre de la mémoire virtuelle.

Elle est gérée par le **MMU** (Memory Management Unit), ou Unité de Gestion de la Mémoire.

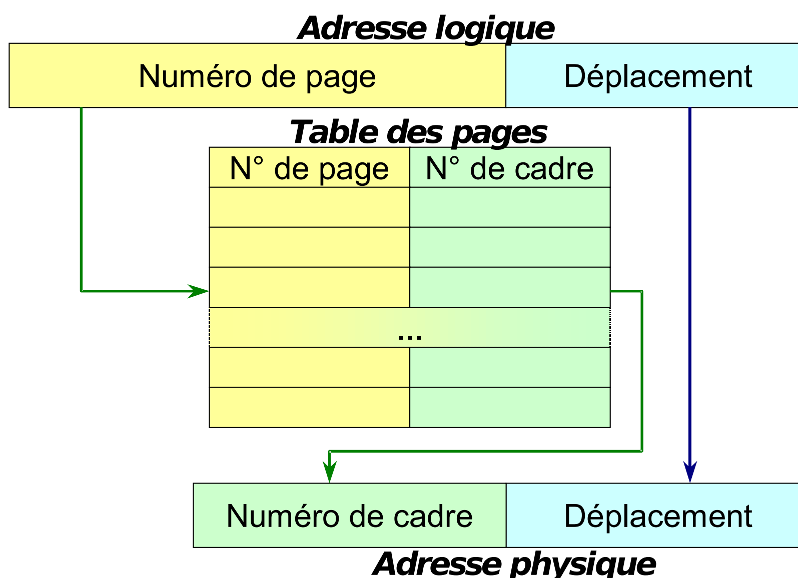
Lorsqu'il n'y a plus de place en mémoire centrale, une case est choisie pour être *swapée* (copiée sur disque pour libérer de la place en RAM). Or, cette case a pu déjà être *swapée* par le passé. Dans ce cas, il est tout à fait intéressant de savoir si le contenu de la case a été modifié en mémoire centrale depuis son dernier chargement.

Ainsi, si le contenu n'a pas été modifié (cas où les accès par les processus ont été réalisés en lecture seule), et si la zone de swap qui contenait cette case n'a pas été ré-attribuée pour stocker une autre case, il est inutile de recopier son contenu sur disque.

C'est pour cette raison que la table des pages contient aussi un « bit M », dit bit de *Modification* (ou « bit D » pour *Dirty* en Anglais). Ce bit vaut 1 si la page est nouvellement allouée (vu qu'elle n'a jamais été swapée par le passé), ou lorsque le contenu de la case a été modifié depuis son dernier chargement en provenance de la zone de swap. Et il vaut 0 après un chargement du cadre depuis un bloc de swap, et reste à 0 tant que le contenu du cadre n'est pas modifié en RAM.

Avec ce mécanisme, si ce cadre est élu pour être à nouveau stocké sur le disque, si le bloc de swap n'a pas été réutilisé et que le bit M vaut 0, il n'y aura pas besoin de recopier le cadre sur le disque (les contenus étant identiques).

Question 3.b) [*1 point*] :



Traduction d'adresse (pagination)

Question 3.c) [*5 points*] : L'algorithme LRU (Least Recently Used/la moins récemment utilisée) élit comme page celle qui n'a pas été référencée (accès en lecture ou en écriture) depuis le plus longtemps.

L'algorithme de la seconde chance est une optimisation de l'algorithme LRU. Principe : chaque page a un « bit de référence » initialisé à 0 quand la page est chargée en RAM. Pour chercher quelle est la page qui sera élue pour être mise sur disque, on recherche en premier lieu celle qui n'a pas été référencée depuis le plus longtemps (algorithme LRU).

La première fois qu'une page est choisie par cet algorithme, on ne fait rien (on positionne juste son bit de référence à 1). Si une référence est faite à cette page, le bit est remis à 0. Ça n'est que lorsque l'algorithme LRU désignera cette page avec le bit de référence à 1 qu'elle sera élue pour être stockée sur disque.

Déroulement de l'algorithme LRU :

Page demandée	***3***	***4***	***5***	***6***	***4***	***7***	***4***	***0***	***6***	***7***	***3***	***7***	***6***	***5***	***6***	***4***	***5***	***3***	***4***	***6***	***5***	***4***
Case 0	3	3	**3**	6	6	6	**6**	0	0	**0**	3	3	**3**	5	5	5	5	5	**5**	6	6	6
Case 1		4	4	4	4	4	4	4	**4**	7	7	7	7	7	**7**	4	4	4	4	4	4	4
Case 2			5	5	**5**	7	7	**7**	6	6	6	6	6	6	6	6	**6**	3	3	**3**	5	5
Défaut de page	O	O	O	O	N	O	N	O	O	O	O	N	N	O	N	O	N	O	N	O	O	N

A la fin de tous les accès, la table des pages est alors :

page	bit V	case
0	0	/
1	0	/
2	0	/
3	0	/
4	1	1
5	1	2
6	1	0
7	0	/

Déroulement de l'algorithme de la seconde chance (les pages soulignées sont celles dont le bit de la seconde chance est à 1, celle en gras+souligné sont celles qui sont élues pour être déplacées sur disque) **:**

Page demandée	***3***	***4***	***5***	***6***	***4***	***7***	***4***	***0***	***6***	***7***	***3***	***7***	***6***	***5***	***6***	***4***	***5***	***3***	***4***	***6***	***5***	***4***
Case 0	3	3	**3**	6	6	6	**6**	0	0	**0**	3	3	**3**	5	5	5	5	5	**5**	6	6	6
Case 1		4	4	4	4	4	4	4	**4**	7	7	7	7	7	**7**	4	4	4	4	4	4	4
Case 2			5	5	**5**	7	7	**7**	6	6	6	6	6	6	6	6	**6**	3	3	**3**	5	5
Défaut de page	O	O	O	O	N	O	N	O	O	O	O	N	N	O	N	O	N	O	N	O	O	N

A la fin de tous les accès, la table des pages est alors :

page	bit V	case
0	0	/
1	0	/
2	0	/
3	0	/
4	1	1
5	1	2
6	1	0
7	0	/

Observation : l'algorithme de la seconde chance ne change rien dans le cas présent. Son efficacité ne peut être mesurée que sur le long terme, ou dans des cas particuliers.