Processus

Thèmes abordés

- Notions de base sur la mise en œuvre et la gestion des processus
- Ordonnancement des processus
- Présentation de l'API processus Unix

1 Questions

1. Comment les interruptions rendent-elles possible la multiprogrammation ?
2. Sur un système biprocesseur, est-il concevable qu'un processus donné puisse s'exécuter sur l'un puis l'autre des deux processeurs alternativement ? Justifiez votre réponse en expliquant le mécanisme de commutation de processus.

2 Ordonnancement des processus

4 processus, P1, P2, P3 et P4, s'exécutent sur un unique processeur. Leurs temps d'exécution ont été mesurés. Pendant ces mesures, seuls des calculs sont effectués par les processus (pas d'E/S). Pour chaque processus, on définit les couples (date d'activation, durée d'exécution) suivants : P1(0,4), P2(1,2), P3(3,4), P4(5,2). On note TR, le temps de réponse, défini par : TR = date de fin - date d'activation.

Question. *Construire le chronogramme représentant l'exécution des différents processus suivant un mécanisme de tourniquet avec un quantum de temps égal à 2. Déterminer le temps de réponse de chaque processus ainsi que le temps de réponse moyen. Que constatez-vous ?*

A chaque processus est attribuée une priorité (forte priorité = petite valeur). La gestion de l'unité centrale s'effectue maintenant selon les priorités avec préemption. Pour chaque processus, les priorités sont les suivantes : 4 à P1, 1 à P2, 3 à P3 et 2 à P4. Ainsi, P2 et P4 qui ont des petites durées d'exécution possèdent les priorités les plus élevées et sont donc avantagées.

Question. *Construire le chronogramme représentant l'exécution des différents processus en utilisant un ordonnancement selon des priorités avec préemption. Déterminer le temps de réponse de chaque processus. Que constatez-vous ?*

Supposons que P3 et P4 accèdent à une ressource partagée en exclusion mutuelle après l'exécution d'une unité de temps hors section critique et qu'ils restent ensuite en section critique jusqu'à la fin de leur exécution. L'accès à la section critique est réalisée à l'aide de sémaphores.

Question. *Dans quel état se situe un processus en attente d'une ressource partagée ? Que se passe-t-il lorsqu'il obtient cette ressource ?*

Question. *Le système utilise à nouveau un ordonnancement à base de priorités. Construire le chronogramme représentant l'exécution des différents processus. Déterminer le temps de réponse de chaque processus.*

3 API processus Unix

3.1 Création de processus

```
pid_t fork();
```

`fork()` permet la création d'un nouveau processus, un processus fils, qui s'exécute de façon concurrente avec le processus qui le crée, appelé processus père. Du processus père, le processus fils hérite :

- du même code,
- d'une copie de la zone de données du père,
- de l'environnement,
- de la priorité,
- des descripteur de fichiers ouverts,
- du traitement des signaux.

La valeur de retour de `fork()` permet de distinguer le processus fils du processus père :

- 0 : C'est le processus fils qui s'exécute.
- >0 : C'est le processus père qui s'exécute. Le numéro est l'identifiant (`pid_t` : PID) du processus fils créé.
- -1 : La création a échoué.

On considère le programme suivant :

```c
#include <stdio.h>   // printf
#include <unistd.h>  // fork
#include <stdlib.h>  // EXIT_SUCCESS

int main(int argc, char *argv[]) {
    fork(); printf("fork 1\n");
    fork(); printf("fork 2\n");
    fork(); printf("fork 2\n");

    return EXIT_SUCCESS;
}
```

Question. *Répondre aux questions suivantes en expliquant l'exécution du programme.*

1. *Combien de processus seront-ils engendrés par ce programme ?*
2. *Combien d'occurrences de chaque type de messages* **fork i** *seront affichées ?*
3. *Quel est l'ordre d'apparition des différents types de messages ?*

3.2 Terminaison / Synchronisation père-fils

3.2.1 Terminaison de processus

```
void exit(int status);
```

`exit(n)` met fin au processus courant avec le code de retour `status`. Par convention, le code de retour d'un comportement correct est 0. Il est d'ailleurs préférable d'utiliser les constantes `EXIT_SUCCESS` ou `EXIT_FAILURE` disponibles dans `stdlib.h` pour des raisons de lisibilité de code.

3.2.2 Synchronisation père-fils

```
#include <sys/wait.h>   // voir man -S 2 wait
pid_t wait(int *status);
```

`wait()` provoque la suspension du processus appelant jusqu'à l'occurrence d'un signal émis vers le processus appelant ou jusqu'à ce que l'un de ses processus fils se termine et retourne :

- -1 : si aucun fils n'existe ou si l'appelant reçoit un signal ;
- >0 : numéro d'identifiant du processus fils qui vient de terminer.

Le valeur de `status` permet de connaître la façon dont le processus fils s'est terminé :

- si le processus fils s'est terminé via la primitive `exit()`, le deuxième octet de la valeur de `status` est le code de retour indiqué à `exit()` ;
- si le processus fils s'est terminé sur la réception d'un signal, la valeur de `status` est égale au numéro du signal reçu par le processus fils ;
- l'utilisation des macros `WIFEXITED`, `WIFSIGNALED`, `WEXITSTATUS`, `WTERMSIG`, ... permet de manipuler simplement la valeur de `status`.

Exemple d'appel de la primitive `wait()` :

```
int status;
if ( wait(&status) != -1 ) {
    if (WIFEXITED(status)) {
        printf("Le processus fils s'est terminé avec le code %i\n",
   WEXITSTATUS(status));
    }
}
```

3.2.3 Exercice

Écrire un programme qui :

- crée un processus fils ;
- dans le processus :
  - affiche la valeur de retour du `fork`, l'identifiant du processus et l'identifiant du processus père ;
  - attend 4 secondes ;
  - se termine avec le code de retour 2.
- dans le processus père :
  - affiche la valeur de retour du `fork`, l'identifiant du processus et l'identifiant du processus fils créé ;
  - se met en attente de la terminaison de son fils ;
  - affiche l'identifiant et le code de retour du processus fils qui vient de terminer.

Primitives utiles :

- `pid_t getpid()` et `pid_t getppid()` fournissent respectivement le numéro d'identifiant du processus appelant et du processus parent. `pid_t` s'apparente à `int` ou `long`.
- `int sleep(int n)` met en pause l'exécution du processus appelant pendant une durée de `n` secondes.

Question. *Que se passe-t-il si le processus fils se termine avant que le processus père ne se bloque sur la primitive `wait()` ?*