IPC

Questions principales

Expliquez le mécanisme du producteur-consommateur. Détaillez-en le principe, le code, les appels système liés.

Expliquez la réalisation d'une section critique via "variable partagée", "blocage des interruptions" et via "sémaphores de Dijkstra". Détaillez les appels système Down et Up. Comparrez ces trois approches.

Expliquez la réalisation d'une section critique via "BTS", "alternance" et via "sémaphores de Deijkstra". Détaillez les appels système Down et up. Comparez ces trois approches.

semget(), semctl(), semop() : Quelle est l'utilité ? Quels sont les arguments ? Quelle est la valeur de retour ? Etablissez le lien entre ces appels système et ceux vus en théorie (up() et down())

semget

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

L'appel système `semget()` retourne l'identifieur du set de sémaphore associé à la clé passée en paramètre (`key_t key`). Un nouveau set de `nsems` est créé si la clé a la valeur `IPC_PRIVATE` ou si aucun set de sémaphore n'est associé avec la clé et `IPC_CREAT` est spécifié dans `semflg`.

Si `semflg` spécifie `IPC_CREAT` et `IPC_EXCL` et que le set de sémaphore existe, `semget()` échoue et met `EEXIST` dans `errno`.

Note : `IPC_PRIVATE` n'est pas un flag mais une clé (`key_t`). Si cette valeur spéciale est utilisée, l'appel système ignore tout sauf les 9 derniers bits significatifs de `semflg` et crée un nouveau set de sémaphore. Ceci permet de s'assurer qu'un autre processus n'utilisera pas la même clé.

semctl

```
#include <sys/types.h>
#include <sys/ipc.h>
```

```c
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, ...);
```

`semctl()` exécute l'opération spécifiée par `cmd` sur le set de sémaphore identifié par `semid`, ou sur le `semnume` élément de ce set.

- `IPC_SET` : initialise les valeurs des sémaphores du set.
- `IPC_RMID` : supprime immédiatement le set de sémaphore, réveillant tous les processus bloqués par un appel `semop` sur le set.

semop

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, unsigned nsops);
```

`semop()` effectue des opérations sur les sémaphores sélectionnés du set de sémaphore identifié par `semid`. Chacun des `nsops` éléments du tableau pointé par `sops` définit une opération à effectuer sur un unique sémaphore. Les éléments de cette structure sont de type `struct sembuf`, qui contient :

```c
unsigned short sem_num;  /* semaphore number */
short          sem_op;   /* semaphore operation */
short          sem_flg;  /* operation flags */
```

L'ensemble des opérations contenues dans `sops` est exécutée dans l'ordre du tableau et sont atomiques, ce qui implique que toutes les opérations sont exécutées comme une unité, ou aucune. Cet appel est donc différent d'appeler 2 fois `down()` ou `up()`, ceux-ci ne travaillant que sur le premier sémaphore du set, et n'étant pas atomiques.

ls | wc -l : Comment l'OS parvient à exécuter cette ligne de commande ? Expliquez en détail le mécanisme sous-jacent. De façon très détaillée, expliquez comment il parvient à synchroniser wc et ls afin que wc ait toujours des données à lire.

Lors de l'exécution de la commande, le shell va trouver le token "|" signifiant un "pipe" entre deux commandes.

Au départ :

- ls écrit son résultat dans la sortie standard (1, stdout).
- wc lit dans l'entrée standard (0, stdin).

Il faut donc créer un pipe à l'aide de la fonction du même nom qui prend un tableau à deux entrées d'entier : `C int p[2]; pipe(p);` Lorsque le processus père va créer ses deux fils par le biais du fork(), chacun va hériter de la table des handle du père. Le fork s'occupant du `ls` modifie sa sortie standard pour qu'elle écrive dans le pipe.

```
/* ls */
close(p[0]);    // Fermeture de la lecture
dup2(p[1], 1);

/* wc -l */
close(p[1]);    // Fermeture de l'écriture
dup2(p[0], 0);  // Entrée standard = la lecture dans le pipe
```

Après cela, par le biais du principe de "producteur consommateur", le 2e fils peut lire dans le pipe jusqu'à obtention du `E.O.F.` que le `ls` aura envoyé quand il n'aura plus rien à écrire.

Ainsi, exécuter les logiciels via la commande `exec` ne remplacera pas la table des handle mais seulement text data et stack.

```
execl("/usr/bin/ls", "ls", NULL);

execl("/usr/bin/wc", "wc", "-l", NULL);
```

Quels appels système permettent de gérer les signaux ? Détaillez en les paramètres et le fonctionnement. Quelles sont les limites et les défauts de ces signaux ? Quel est le rôle de la table des interruptions, de la table des processus, de l'ordonnanceur dans ces cas ? Il faut utiliser le prototype :

```
sigaction(SIGNAL, nouveau_trigger, ancien_trigger);
```

Qui retourne 0 en cas de succès ou -1 en cas d'erreur.

Pour assigner une procédure à un signal :

```
struct sigaction action_handler;
action.handler.sa_handler = fct;
sigaction(SIGNAL, action_handler, NULL);

// Méthode deprecated
signal(INTERRUP, void (* fct)(int));
// Qui retournera 'SIG_ERR' en cas d'erreur, ou l'ancienne action en cas de
// réussite.
```

Pour supprimer la fonction liée un signal

```
struct sigaction action_handler;
action.handler.sa_handler = SIG_IGN;
sigaction(SIGNAL, &action_handler, NULL);

// Méthode deprecated
signal(INTERRUP, SIG_IGN);
```

Pour rétablir le comportement par défaut, il faut lier `SIG\_DFL`.

La table des interruptions lie chaque interruption à une fonction et un bit d'activation. Lors d'une interruption, le tableau est scanné **quand le process devient élu** et le premier bit à 1 trouvé lance la fonction associée. Il est donc impossible de lancer plusieurs fois la même interruption. (32 entrées dans la table = 32 bits, un par interruption, et 32 pointeurs de fonctions, un par interruption).

Quand l'ordonnanceur donne la main :

- il exécute en priorité le tableau d'interruptions.
- puis remet les bits, qui étaient à 1, à 0.

- reprend le processus où il s'était interrompu.

Si nous lançons plusieurs fois la même interruption (spam Ctrl^c), elle ne sera faite qu'une fois puisque le bit se mettra une fois à 1.

Si un signal arrive avant un autre, il ne sera pas spécialement effectué en priorité.

Certaines interruptions ne peuvent être changées :

- SIGKILL
- SIGSTOP

2 interruptions sont spécialement créées pour "l'utilisateur" car elles ne possèdent pas de signification.

- SIGUSR1
- SIGUSR2

Pour "lancer" un signal manuellement, il faut utiliser la commande

```
kill(PID, INTERRUP);
```

Un process peut faire une "attente active" (ou volontaire) d'un signal à l'aide de `int pause(void)`, évitant le sempiternel `while(1)`.

L'appel système `int alarm(unsigned sec)` permet de mettre un minuteur qui lance un `SIGALRM` après sec secondes.

socket(), bind(), listen(), accept(), connect() : Quelle est l'utilité ? Quels sont les arguments ? Quelle est la valeur de retour ?

Utilisation de l'appel système pipe et situations d'interblocages : expliquez en vous basant sur des exemples de code comment une telle situation peut être obtenue. Détaillez et faites le lien avec les appels système Up et Down.

Questions secondaires

(socket) Quelle est l'utilité de la fonction htons() ? Comment faire parvenir un message à un processus qui s'exécute sur un autre ordinateur ?

(socket) Comment écrire une application client / serveur où plusieurs clients peuvent être connectés au serveur en même temps ?