IPC

Questions principales

Expliquez le mécanisme du producteur-consommateur. Détaillez-en le principe, le code, les appels système liés. On va créer deux sémaphores distinct : un pour bloquer/liberer la production et un pour bloquer/liberer la consommation. Ceux-ci seront des quantités de "ressource disponible".

```
Ensemble des ressources disponible comme étant un tableau, entièrement vide.
-----------------------------------------------------------
|       |       |       |       |       |       |       |
|       |       |       |       |       |       |       |
-----------------------------------------------------------
```

Le sémaphore de la production possède n nombre de chose à produire. Il va donc prendre une ressource `down(prod)`ce qui va rendre le producteur bloqué si le nombre de case n'est plus positif et attendre qu'un consommateur va libèrer une case.

De même, le lecteur va lire et si la quantité de produit, commencant à 0, est nulle, il va attendre producteur ajoutera une ressouce.

Le producteur va réguler l'arrivage de case remplie du lecteur et le consommateur va réguler l'arrivage de case vide du producteur.

```c
#define QUANTITE 5

typedef char produit;

int main()
{
    int tete = 0;
    int queue = 0;
    tab produit[QUANTITE];
    /*
    *
    * VARIABLES A PARTAGER
    */
    int s = semget();
    semctl(s, ...);

    if (fork() == 0)
    {
        while(1)
```

```
        {
            down(conso);
            // cool j'ai une ressource
            up(libre);
        }

        exit(EXIT_SUCCESS);
    }

    while(1)
    {

    }

    exit(EXIT_SUCCESS);
}
```

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. Une section critique est une partie de code où un process accède à une ressource partagée, ce qui peut entrainer des accès concurrent. On a donc besoin d'une exclusien mutuelle pour être sur qu'un process ne s'approprie pas une ressource déjà utilisée par un autre process. Il existe différentes techniques, toutes ayant leur défauts et avantages, pour réaliser cette exclusion.

Il faut ces 4 conditions pour assurer une exclusion mutuelle :

- Deux process ne peuvent pas être en section critique en même temps.
- Le nombre de CPU n'a pas d'importance.
- Aucun process ne peut bloquer un autre process s'il n'est pas dans sa section critique.
- Aucun process ne doit attendre indéfiniment pour entrer dans sa région critique.

Exclusions mutuelles avec attente active

- Désactiver les interruptions : C'est une des solutions les plus simples à réaliser. Cepandant, elle n'est absolument pas recommandée pour les process utilisateurs car un process pourrait ne pas réactiver les interruptions. De plus sur les machines multi-processeurs, désactiver les interruptions n'affecte qu'un seul CPU.
- Variables partagées : le principe est d'avoir une variable en mémoire partagée, avec une valeur initialement à 0. Quand un process entre dans

sa région critique, il teste la variable, si elle est à 0, il la set à 1 et entre sa région critique. Un autre process aura juste à attendre que cette variable redevienne 0. Le problème de cette technique est qu'une interruption peut se produire entre le test et la modification de la variable. Imaginons qu'un process test la variable, cette dernière vaut 0, une interruption survient et un autre process prend la main, test également cette variable, la met à 1 et entre dans sa région critique. Quand le premier process à la main, il met aussi la variable à 1 et entre dans sa région critique en même temps que le deuxième process.

- Alternance (question suivante) : Cette solution est plus sécurisée que la solution avec les variables partagées, cependant elle possède trop d'attente active.

"'C // Process A :

```
while(TRUE)
{
    while(turn != 0);
    critical_region();
    turn = 1;
    noncritical_region();
}

// Process B :

while(TRUE)
{
    while(turn != 1);
    critical_region();
    turn = 0;
    noncritical_region();
}
```

"'

Ici, la variable turn garde en mémoire partagée "à qui est le tour" d'entrer dans sa section critique. Dans l'exemple, si turn vaut 0, c'est au tour de A, si turn vaut 1, c'est au tour de B. Si le process A test turn et que turn vaut 0, il entre dans sa région critique. Le process B quant à lui, si turn vaut 0, il sera mis en attente active (boucle) jusqu'à ce qu'il perde la main. On a donc une perte de temps inutile. Quand le process A quitte sa section critique, il met la variable d'alternance à 1 pour permettre au process B d'entrer dans sa section critique. Le gros inconvénient de ce système est donc l'attente active d'un process, l'effet est encore plus présent si la section critique d'un des deux process est beaucoup plus longue que l'autre.

- BTS (question suivante) : BTS (Bit Test and Set) est une instruction atomique pour tester et modifier une variable. Cette instruction va bloquer le bus mémoire pour empêcher tout process d'accéder à la variable pendant qu'elle est testée/modifiée. Bloquer la mémoire est différent de désactiver les interruptions : en effet, sur une machine multi-process, bloquer les interruptions bloque seulement un CPU. La seule manière est donc de bloquer l'accès à la mémoire. Cette solution possède toujours le problème d'attente active.

- Sémaphores de Dijkstra : Dijkstra a introduit un nouveau type de variable : le sémaphore. Un sémaphore est une variable qui compte le nombre de ressources disponibles si le sémaphore est positif, et le nombre de process demandant la ressource si le sémaphore est négatif. Il existe deux opérations sur les sémaphores : down et up.

  - down : test si le sémaphore est plus petit ou égal à 0, si c'est le cas, le process est mis à l'état bloqué, sinon il décrémente le sémaphore et entre dans sa section critique. Evidemment, tester et modifier la valeur du sémaphore est une opération atomique (voir BTS).
  - up : incrémente la valeur du sémaphore, si un ou plusieurs etaient à l'état bloqué après un down, un d'entre eux est choisi par l'ordonnanceur (si l'ordonnanceur garde une file, alors c'est le premier de la file qui est choisi).

Les sémaphores permettent de résoudre le problème du producteur/consommateur. Down et up sont implémenté comme des appels système. Désactiver les interruptions dans ce cas là est raisonnable car il ne s'agit que de quelques microsecondes pour tester et modifier la variable si nécessaires. Voir question producteur / consommateur pour plus d'informations.

Les sémaphores suppriment donc l'attente active, car un process passe à l'état bloqué si son down ne lui permet pas de continuer.

Expliquez la réalisation d'une section critique via "BTS", "alternance" et via "sémaphores de Dijkstra". Détaillez les appels système Down et up. Comparez ces trois approches. Voir question ci-dessus.

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. `sem : Map<clé, identifiant>; tab_set : Map<identifiant, set_sema>`

key :

- `IPC_PRIVATE`
- une clé manuelle(risque qu'elle soit déjà prise)

nsems :

- Le nombre de sémaphores du set

semflg :

- `IPC_CREAT`
- `IPC_EXCL`

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.

`semget(IPC_PRIVATE, 3, IPC_CREAT | 0666);` 9 bits = 0666 = 110110110b

semctl

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

Il est possible de spécifié l'indice d'un seul sémaphore de ce set via `semnum`.

cmd :

- `SET_VAL` : initialise la valeur arg.val d'un semaphore du set
- `IPC_STAT` : statistique
- `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

```
#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 :

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

`nsops`== nombre d'actions `sops` à effectuer puisqu'il s'agit d'un tableau.

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 :

```
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 ? Les structures utilisées pour les sockets :

- `sockaddr` comprend les information sur le socket lui-même.

```
struct sockaddr
{
    unsigned short sa_family; // AF_INET, AF_UNIX, AF_NS, AF_IMPLINK
    char sa_data[14];         // adresse
}

struct sockaddr_in
{
    short int sin_family;  // Comme sa_family de sockaddr
    unsigned short int sin_port; // Le port (Network Byte Order)
    struct in_addr sin_addr; // L'adresse (Network bye Order)
    unsigned char sin_zero[8]; // Pas utilisé, mettre à NULL
}
```

```
struct in_addr
{
    unsigned long s_addr; // L'adresse (Network Byte Order)
}
```

Les fonctions pour manipuler les adresses IP :

```
int inet_aton(const char * str, struct in_addr * adrr);
in_addr_t inet_addr(const char * str);
char * inet_ntoa(struct in_addr inaddr);
```

- inet_aton convertit une chaîne (en notation pointé) en une adresse en NBO. L'adresse sera stocké dans la structure in_addr. Retourne 1 si la chaîne était valide, 0 sinon.
- inet_addr convertit une chaîne (notation pointé) en entier (NBO). Retourne l'adresse si la chaîne est valide.
- inet_ntoa convertit une adresse en network byte order en une chaîne en notation pointée. (inverse de inet_addr et inet_aton)

Exemple :

```
struct sockaddr_in dest;
dest.sin_addr.s_addr = inet_addr("68.178.157.132");
```

Les sockets permettent la communication entre deux process qui se trouvent sur deux machines différentes, dans cette commnication, il y aura un serveur et un ou plusieurs client. C'est toujours le client qui demande la connexion au serveur. Les sockets établissent un canal de communication entre deux process de la même manière qu'un pipe. Ils ne font que retransmettre les données. Ils sont identifiés à l'aide d'un entier (d'un descripteur).

Le serveur et le client doivent chacun avoir un socket. Le socket du serveur permet d'écouter l'arrivée d'un client. Le socket d uclient permet de s'identifier. Si le client désire envoyer de l'information au serveur, il les envoie via l'appel système `write` en donnant comme premier paramètre le handle du serveur. Le serveur lui utilisera l'appel système `read` avec comme handle le socket du client. Pour fermer un socket, on le ferme avec `close`.

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int family, int type, int protocol);
```

- family : on utilise AF_INET ou PF_INET dans le cadre du cours (IPv4 protocols).

- type : Le type de socket : SOCK_STREAM dans le cadre du cours.
- protocol : Protocole utilisé, on utilise 0 dans le cadre du cours (protocole par défaut).

La première chose à faire pour créer une communication grâce aux sockets est d'appelé la fonction `socket`. Cette fonction va créer le canal de communication et retourne un descripteur ou -1 si il y a erreur.

```
int connect(int sockfd, struct sockaddr * serv_addr, int addrlen);
```

La fonction connect est utilisé par le client pour se connecter à un serveur.

- sockfd : le descripteur retourné par la fonction socket.
- serv_addr : l'adresse de la structure sockaddr qui contient l'adresse IP et port. Ceci est l'adresse du serveur.
- addrlen : sizeof(struct sockaddr).
- Retourne 0 si ok, -1 sinon.

```
int bind(int sockfd, const struct sockaddr * addr, socklen_t addrlen);
```

La fonction `bind` est utilisé par le serveur pour assigner une adresse à un socket. `bind` assigne l'adresse spécifiée par la structure `sockaddr` au socket spécifié par le descripteur `sockfd` qui est le descripteur retourné par la fonction `socket`

```
int listen(int sockfd, int backlog);
```

La fonction listen est utilisée par un processus serveur pour marqure le socket comme socket passif, c'est à dire que le socket sera utilisé pour accepter toute connexion entrante en utilisant la fonction `accept()`. `listen()` ne permet pas encore d'accepter les connexions, cette fonction met simplement le socket en mode "écoute".

- sockfd : le descripteur du socket retourné par `socket()`
- backlog : Le nombre maximum de connexions en attentes pour le socket. Si un client essaye de se connecter quand la file d'attente est complète, il se peut qu'il reçoive une erreur.
- Retourne 0 si ok, -1 sinon.

```
int accept(int sockfd, struct sockaddr * addr, socklen_t * addrlen);
```

La fonction `accept` est utilisé par un serveur pour retourner la prochaine connexion d'un client. Cette fonction attend une connexion, le processus est donc en attente d'une connexion. Cette fonction retourne, si elle réussit, le descripteur du socket du client ayant effectué la connexion sur le serveur.

- sockfd : le descripteur retourné par la fonction `socket()`
- addr : contient l'adresse IP et le port du `client`
- addrlen : sizeof(struct sockaddr)

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. Soient deux processus p1 et p2 et deux tubes t1 et t2.

- p1 va lire dans t1, mais t1 est vide donc p1 attend.
- p2 (arrive dans le game) va lire dans t2 mais t2 est vide donc p2 attend.

Qui va remplir t1 et t2 ?

- p1 est le processus qui va remplir t2, mais il est bloqué
- p2 est le processus qui va remplir t1, mais il est bloqué

=> aucun des deux processus ne va pouvoir ECRIRE car aucun des deux ne peut terminer de LIRE car il n'y a pas de `E.O.F.` dans le pipe car aucun ne va ECRIRE ... `OUROBORS`

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 ? Tous les ordinateurs ne stock pas les bytes dans le même ordre (little endian, big endian).

Pour permettre aux machines de communiquer entre elles, on utilise une convention nommé Network Byte Order. Toutes les adresses et port contenus dans les structures doivent être en NBO.

```
unsigned short htons(unsigned short hostshort);
```

La fonction `htons` convertit une adresse (short) en network byte order (short). D'autres dérivées sont disponibles pour les différents type d'entier :

- htonl : même chose que htons mais pour des long
- ntohs : inverse de htons
- ntohl : inverse de htonl

(socket) Comment écrire une application client / serveur où plusieurs clients peuvent être connectés au serveur en même temps ? Si on veut que le serveur puisse gérer plusieurs connexions, il faut faire, après la fonction `listen()`, une boucle infinie avec la fonction accept, et un fork pour chaque connexion.

```
listen(sockfd, 5);
while(1)
{
    newsockfd = accept(sockfd, (struct sockaddr *) &addr, sizeof(struct
    sockaddr);

    if (newsockfd < 0)
    {
        perror("Error on accept");
        exit(1);
    }

    pid = fork();
    if (pid < 0)
    {
        perror("Error on fork");
        exit(1);
    }
    else if (pid == 0)
    {
        close(sockfd);
        /* Processus client */
        // Do something
        exit(0);
    }
    else
    {
        /* On ferme le socket client dans le père,
         * on en a plus besoin car c'est le fils qui gère
         * le client
         */
        close(newsockfd);
    }
}
```