

Système d'exploitation

SYSIR3 - SYSG4

M. Bastreggi (mba)

Haute École Bruxelles Brabant — École Supérieure d'Informatique

Année académique 2020 / 2021

D'après le cours de M.Jaumain

Section synchronisation - IPC

IPC

- sémaphores Dijkstra
- producteur-
consommateur
- section critique & non
- solutions
- section critique :
Sémaphores
- IPC system V

Avancement

- sémaphores Dijkstra
- producteur-consommateur
- section critique & non solutions
- section critique : Sémaphores
- IPC system V

sémaphores

- ▶ Introduits par Dijkstra en 1965.
- ▶ Permettent de réserver des ressources non partageables
- ▶ Appels Système : DOWN et UP

sémaphore

- ▶ Un **compteur** de ressources disponibles partagé
- ▶ Une **file** de processus attendant une ressource
- ▶ Deux **Appels Système DOWN, UP** obtenir/restituer une ressource

demande de ressource indisponible \Rightarrow processus **bloqué**
par le noyau (change d'état)

Système d'exploitation

└ IPC

└ sémaphores Dijkstra

└ sémaphore

- Un **compteur** de ressources disponibles partagé
- Une **file** de processus attendant une ressource
- Deux **Appels Système DOWN, UP** obtenir/restituer une ressource

demande de ressource indisponible => processus **bloqué**
par le noyau (change d'état)

Un processus bloqué n'est pas ordonnancé

UP et DOWN sont des appels système => l'ordonnanceur a la main à la fin de l'appel système

sémaphores : compteur de ressources

Un sémaphore, est souvent représenté par un entier r

- ▶ $r > 0$, r ressources disponibles
- ▶ $r = 0$, aucune ressource disponible

Système d'exploitation

└ IPC

└ sémaphores Dijkstra

└ sémaphores : compteur de ressources

Un sémaphore, est souvent représenté par un entier r

- $r > 0$, r ressources disponibles
- $r = 0$, aucune ressource disponible

il s'agit d'abus de langage, un sémaphore regroupe plusieurs informations : notamment une file de processus.

sémaphores : file de process en attente

Un sémaphore utilise une file de processus en attente de la ressource

En cas de processus en attente d'une ressource \Rightarrow valeurs négatives du compteur

- ▶ $r < 0$, aucune ressource disponible et $|r|$ processus bloqués en attente dans la file

sémaphores : appel système - DOWN

DOWN (r)

- ▶ demande une ressource. Le sémaphore est décrémenté.

```
si ressource—indisponible alors bloquer le process  
et le mettre en file d'attente;  
fsi  
r--;
```

la demande (DOWN) d'une ressource indisponible fait passer à l'état BLOQUÉ jusqu'à la libération (UP) de la ressource par celui qui la détient

sémaphores : appel système - UP

UP (r)

- ▶ restitue une ressource. Le sémaphore est incrémenté. Si il existe des processus bloqués en file d'attente, l'état du premier de la file est remis à PRÊT, celui-ci sort de la file en obtenant la ressource et est de nouveau candidat pour être élu.

```
si file —non—vide alors débloquent un process  
  (ce qui revient à lui attribuer  
   la ressource);  
r++;  
fsi
```

Appels Système

les appels système UP et DOWN sont rendus atomiques en bloquant les interruptions

Down (r)	Up (r)
CLI ; If($r \leq 0$) (id P en file ; état P bloqué) ; $r=r-1$; STI ;	CLI ; If($r < 0$) (sortir premier de file ; état premier prêt) ; $r=r+1$; STI ;

Système d'exploitation

└ IPC

└ sémaphores Dijkstra

└ Appels Système

Appels Système

les appels système UP et DOWN sont rendus atomiques en bloquant les interruptions

Down(r)	Up(r)
CLI;	CLI;
If($r <= 0$) (id P en file; état P bloqué);	If($r < 0$) (sortir premier de file; état premier prêt);
$r=r+1$;	$r=r+1$;
STI;	STI;

bloquer les interruptions est une instruction réservée au code du noyau, on suppose que ce code est sûr.

exemples de problèmes résolus par les sémaphores

Les sémaphores de Dijkstra vont permettre de résoudre certains problèmes de synchronisation de processus comme :

- ▶ le producteur-consommateur (mécanisme implémentant les **pipe**)
- ▶ la section/région critique

Avancement

- sémaphores Dijkstra
- **producteur-consommateur**
- section critique & non solutions
- section critique : Sémaphores
- IPC system V

producteur-consommateur

producteur-consommateur

- ▶ un processus (producteur) produit des données qu'un autre (consommateur) utilise
 - ▶ les deux processus tournent en parallèle
 - ▶ le volume de données à échanger est illimité
 - ▶ on utilise un tableau en mémoire partagée de **taille fixe** pour l'échange de données (bounded buffer), FIFO circulaire
 - ▶ on veut réaliser un échange fiable
- une synchronisation est nécessaire

producteur-consommateur

- ▶ il faut éviter d'écraser des données non encore lues (produire trop vite)
- ▶ il faut éviter la lecture de données inexistantes (consommer trop vite)

producteur-consommateur

Problème du tampon délimité (bounded buffer)

- ▶ Deux processus communiquent via une mémoire partagée.
- ▶ Le producteur utilise des cases vides et doit être bloqué si il n'y a plus de cases vides (non encore lues) le tampon est plein.
- ▶ Le consommateur utilise des cases pleines et doit être bloqué si aucune case n'est pleine le tampon est vide.

producteur-consommateur

solution avec sémaphores

- ▶ deux sémaphores pour deux ressources :
 - cases pleines (consommateur)
 - cases vides (producteur)
- ▶ une table partagée :
 - tableau circulaire de N cases.

Deux indices de parcours **tête et queue** serviront aux deux processus pour parcourir le tableau.

À ce stade les indices sont des variables locales aux processus

Système d'exploitation

└ IPC

└ producteur-consommateur

└ producteur-consommateur

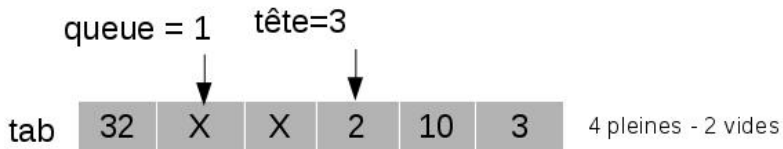
solution avec sémaphores

- deux sémaphores pour deux ressources :
 - cases pleines (consommateur)
 - cases vides (producteur)
- une table partagée :
 - tableau circulaire de N cases.

Deux indices de parcours **tête** et **queue** serviront aux deux processus pour parcourir le tableau.
À ce stade les indices sont des variables locales aux processus

- Le producteur a besoin d'une ressource **case vide**
- Le consommateur a besoin d'une ressource **case pleine**

producteur-consommateur



```
tab[queue] = info ; // produire  
queue = queue+1 % 6 ;
```

```
info = tab[tête] ; // consommer  
tête = tête+1 % 6 ;
```

producteur-consommateur

Initialisations : les N cases de tab sont vides

- ▶ sem vides = N
- ▶ sem pleines = 0
- ▶ initialisation de tête et queue ? tête = queue = $N-1$ (par exemple)

le sémaphore **pleines** vaut 0 et bloque le consommateur au début

Système d'exploitation

└ IPC

└ producteur-consommateur

└ producteur-consommateur

Initialisations : les N cases de tab sont vides

- sem vides = N
- sem pleines = 0
- initialisation de tête et queue ? tête = queue = $N-1$ (par exemple)

le sémaphore **pleines** vaut 0 et bloque le consommateur au début

producteur : **down (vides)** ; ... **up (pleines)**
consommateur : **down (pleines)** ; ... **up (vides)**

producteur-consommateur

Producteur()	Consommateur()
<pre>while (TRUE) { Down(vides); tab[queue]=info; queue=(queue+1) mod N; Up(pleines); }</pre>	<pre>while (TRUE) { Down(pleines); info=tab[tête]; tête=(tête+1) mod N; Up(vides); }</pre>

Système d'exploitation

└ IPC

└ producteur-consommateur

└ producteur-consommateur

Producteur()	Consommateur()
<pre>while (TRUE) { Down(vides); tab[queue]=info; queue=(queue+1) mod N; Up(pleines); }</pre>	<pre>while (TRUE) { Down(pleines); info=tab[tête]; tête=(tête+1) mod N; Up(vides); }</pre>

Le producteur et le consommateur accèdent la table en modification à des indices différents (pas d'accès concurrent)

garanti par les sémaphores de Dijkstra :

si tête = queue => (vides =0 ou pleines =0) => producteur ou consommateur bloqué

producteur-consommateur

Imaginer la transmission de 8 caractères avec un buffer de taille 3

Cela peut-il fonctionner avec un buffer de taille 1 ?

Avancement

- sémaphores Dijkstra
- producteur-consommateur
- section critique & non solutions
- section critique : Sémaphores
- IPC system V

concurrency d'accès

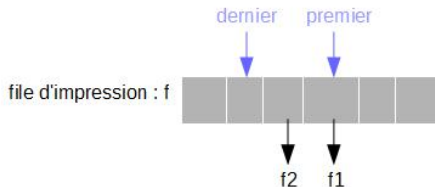
Un problème peut se poser lié à la concurrence d'accès lors du partage de ressources en mise à jour (RAM, fichier, BD).

concurrency d'accès

illustration du problème

P1 et P2 déposent un fichier pour l'impression.

Une table f et des indices de remplissage de la table tous partagés, mémorisent les dépôts faits



concurrency d'accès

code de P1	code de P2
<pre>dernier = dernier -1 ; f[dernier] = f1 ;</pre>	<pre>dernier = dernier -1 ; ———> interruption et ordonnancement de P1 f[dernier] = f2</pre>

une interruption survient avant la fin de la mise à jour par P2 la main est donnée à P1 ...

=> les changements faits par P1 à ce stade seront écrasés par P2 (perte de la demande d'impression de P1)

concurrency d'accès

- ▶ Ce schéma risque de se produire lorsque plusieurs processus accèdent en mise à jour à des données partagées
 - lire
 - modifier
 - écrire
- ▶ réservation de la dernière place disponible sur l'avion
- ▶ compteur ou données partagées en mémoire ou sur disque

il faut y prendre garde et éviter les mises à jour concurrentes

Système d'exploitation

└ IPC

└ section critique & non solutions

└ concurrence d'accès

concurrence d'accès

- Ce schéma risque de se produire lorsque plusieurs processus accèdent en mise à jour à des données partagées
 - lire
 - modifier
 - écrire
- réservation de la dernière place disponible sur l'avion
- compteur ou données partagées en mémoire ou sur disque

il faut y prendre garde et éviter les mises à jour concurrentes

une opération de mise à jour doit se terminer avant qu'une deuxième ne commence

le code de Mise à Jour doit être exécuté de manière exclusive

section critique

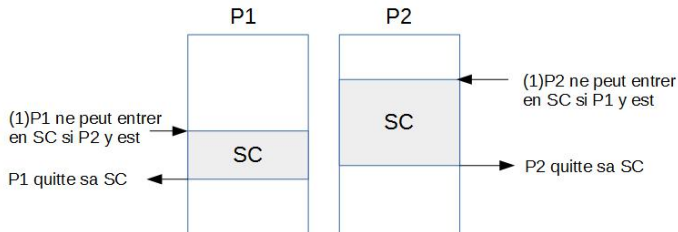
On nomme **Section critique** la partie de code d'un processus accédant en mise à jour à une ressource partagée.

- ▶ éviter que plus d'un processus se trouve en même temps dans sa section critique.
- ▶ dès qu'un processus rentre dans sa section critique, on empêche les autres de rentrer dans la leur.

Chaque processus accède de manière exclusive à la mise à jour.

section critique

deux processus ne peuvent se trouver en même temps dans leurs **Sections Critiques**



blocage des interruptions

non solution 1

- ▶ Interdire toute interruption pendant la section critique
 - ▶ CLI - STI
 - CLI;
 - sectionCritique();
 - STI;
 - ▶ Pour architectures mono processeur
 - ▶ Que se passe-t'il si le processus contient une boucle infinie si les interruptions sont bloquées?
- (-) Ne convient pas aux processus utilisateur ...

variable de verrouillage

non solution 2

Soit $v=0$, une **variable partagée** qui vaut 0 si personne n'est en section critique et 1 sinon.

code de P1	code de P2
<pre>while (TRUE) (while (v==1); v=1; SectionCritique1(); v=0; SectionNonCritique();)</pre>	<pre>while (TRUE) (while (v==1); v=1; SectionCritique2(); v=0; SectionNonCritique();)</pre>

(-) défaut : provoquons une interruption avant $v=1$

Système d'exploitation

└ IPC

└ section critique & non solutions

└ variable de verrouillage

non solution 2

Soit $v=0$, une **variable partagée** qui vaut 0 si personne n'est en section critique et 1 sinon.

code de P1	code de P2
while (TRUE) { while (v==1); v=1; SectionCritique1(); v=0; SectionNonCritique(); }	while (TRUE) { while (v==1); v=1; SectionCritique2(); v=0; SectionNonCritique(); }

(-) défaut : provoquons une interruption avant $v=1$

La variable v est elle même une ressource partagée accessible en mise à jour.

Son accès devrait être exclusif. Ce n'est pas le cas ici. Il faudrait rendre les instructions **while (v==1); v=1;** non interruptibles

Nous verrons plus tard comment partager des variables

Instruction TSL

non solution 3

TSL est une instruction atomique qui permet de lire et écrire en mémoire

- ▶ en une seule instruction (bloque l'accès au bus de mémoire à tous les CPU's)
- ▶ Trouvez le défaut. (Si un processus est prioritaire)

TSL suite

TSL se comporte comme BTS dans l'architecture X86

```
while (TSL (varlock) == 1);  
    ...                ; Section Critique  
varlock = 0;
```

TSL se comporte comme BTS dans l'architecture X86

```
while (TSL (varlock) == 1);  
...                               ; Section Critique  
varlock = 0;
```

- (-) cas d'un processus prioritaire ? **priorités inversées**

alternance

non solution 4

Soit $v=0$, une variable partagée qui vaut 0 si P1 peut s'exécuter et 1 si P2 peut s'exécuter.

code de P1	code de P2
<pre>while (TRUE) (while (v==1); SectionCritique1(); v=1; SectionNonCritique();)</pre>	<pre>while (TRUE) (while (v==0); SectionCritique2(); v=0; SectionNonCritique();)</pre>

- ▶ (-) ordre imposé
- ▶ (-) attente forcée

Avancement

- sémaphores Dijkstra
- producteur-consommateur
- section critique & non solutions
- **section critique : Sémaphores**
- IPC system V

sémaphores

Les sémaphores vont permettre de réaliser une section critique et résoudre le problème de concurrence.

La région de code est la ressource unique que deux processus ne peuvent accéder en même temps. Nous représentons cela par un sémaphore initialisé à 1

```
sem r = 1
```

```
DOWN(r)  
SC  
UP(r)
```

Tous les processus concurrents auront un code semblable.
Accès exclusif à la section critique.

Système d'exploitation

└ IPC

└ section critique : Sémaphores

└ sémaphores

sémaphores

Les sémaphores vont permettre de réaliser une section critique et résoudre le problème de concurrence. La région de code est la ressource unique que deux processus ne peuvent accéder en même temps. Nous représentons cela par un sémaphore initialisé à 1

sem r = 1

DOWN(r)
SC
UP(r)

Tous les processus concurrents auront un code semblable.
Accès exclusif à la section critique.

Les sémaphores évitent l'attente active, un processus bloqué n'est pas élu.

Le noyau intervient via les deux Appels Système DOWN et UP.

généralisation du problème du producteur-consommateur

Les codes étudiés plus haut ne posent pas le problème de concurrence en effet :

- ▶ les variables queue et tête ne sont pas partagées
- ▶ pas d'accès concurrent aux cases du tableau (synchronisation mise en oeuvre)

Dans le cas de **plusieurs consommateurs/producteurs** nous devons partager les indices tête/queue

- ▶ **la mise à jour doit être protégée** par une **section critique**

producteur-consommateurS

plusieurs consommateurs avec section critique

Consommateur()

```
While (TRUE) {  
  Down(pleines);  
  Down(mutex);  
  info=tab[tête];  
  tête=tête+1 mod N;  
  Up(mutex);  
  Up(vides);  
}
```

producteurS-consommateur

plusieurs producteurs avec section critique

```
Down(vides);  
Down(mutex);  
...  
  
Up(mutex);  
Up(pleines);
```

producteurS-consommateurS

Attention à l'ordre des Appels Système down dans le cas de partage de mutex entre producteur et consommateur

ordre des DOWN!!!

```
Down(mutex);  
Down(pleines ou vides);  
...
```

interblocage possible!!

Questions ?



Avancement

- sémaphores Dijkstra
- producteur-consommateur
- section critique & non solutions
- section critique : Sémaphores
- IPC system V

IPC - system V

Interprocess Communication :
Librairie à trois volets :

- ▶ **sémaphores**
- ▶ **mémoire partagée**
- ▶ files de **messages**

structures

Caractéristiques des IPC

- ▶ Créées dynamiquement
- ▶ Sont **persistantes**
- ▶ Nécessitent une suppression explicite (IPC_RMID), cette dernière n'étant **pas toujours immédiate**
- ▶ Partage **sans ancêtre commun créateur**

Suppression (IPC_RMID) autorisée aux **créateur**,
propriétaire, **root**

Système d'exploitation

└ IPC

└ IPC system V

└ structures

Caractéristiques des IPC

- Créées dynamiquement
- Sont **persistantes**
- Nécessitent une suppression explicite (IPC_RMID), cette dernière n'étant **pas toujours immédiate**
- Partage **sans ancêtre commun créateur**

Suppression (IPC_RMID) autorisée aux **créateur, propriétaire, root**

la commande **ipcs** montre les ressources IPC accessibles du système

identification

- ▶ **identifiant** unique attribué par le noyau
- ▶ peut être obtenu via une **clé**

obtenir l'identifiant

Obtenir l'identifiant à partir d'une clé :

```
int semget (key_t key, int nsems, int semflg);  
int shmget(key_t key, size_t size, int shmflg);  
int msgget(key_t key, int msgflg);
```

- ▶ key - clé connue par plusieurs process ou IPC_PRIVATE
- ▶ xxxflg - IPC_CREAT | IPC_EXCL - droits d'accès rw-rw-rw- (9 bits)

valeur de retour : identifiant de l'ensemble

size : (shm...) arrondi supérieur taille page

Système d'exploitation

└ IPC

└ IPC system V

└ obtenir l'identifiant

obtenir l'identifiant

Obtenir l'identifiant à partir d'une clé :

```
int shmget(key_t key, int nshare, int shmflg);  
int shmat(key_t key, void *shmadr, int shmflg);  
int shmctl(key_t key, int cmd, int shmflg);
```

- key - clé connue par plusieurs process ou IPC_PRIVATE
- shmflg - IPC_CREAT | IPC_EXCL - droits d'accès rw-rw-rw- (9 bits)

valeur de retour : identifiant de l'ensemble

size : (shm...) arrondi supérieur taille page

Il est possible que la même clé soit choisie par un autre programme "non en rapport" ce qui crée des interférences.



Pas de garantie d'accès exclusif à ce type de ressource (il suffit d'utiliser la même clé ou l'identifiant) => restreindre les droits

IPC_PRIVATE

- ▶ si key vaut IPC_PRIVATE une nouvelle ressource est créée d'office
- ▶ aucun risque d'utilisation d'une ressource existante
- ▶ l'identifiant IPC peut être communiqué aux autres processus par différents moyens

fork, pipe, ...

partage

Pour utiliser un ensemble de sémaphores il faut en connaître l'identifiant.

- ▶ par un ultérieur appel à `semget` (même valeur de clé (`!= IPC_PRIVATE`), pas de `IPC_CREAT|IPC_EXCL`).
- ▶ par clonage de la variable `id` du père.
- ▶ `id` comme paramètre de la commande `exec`.
- ▶ communication entre process (pipe, mémoire partagée ..) ...

Système d'exploitation

└ IPC

└ IPC system V

└ partage

partage

Pour utiliser un ensemble de sémaphores il faut en connaître l'identifiant.

- par un ultérieur appel à `semget` (même valeur de clé (`!= IPC_PRIVATE`), pas de `IPC_CREAT|IPC_EXCL`).
- par clonage de la variable `id` du père.
- `id` comme paramètre de la commande `exec`.
- communication entre process (pipe, mémoire partagée ...) ...

- KEY - plusieurs appels à `semget`
- un appel à `semget` (père) et clonage (`fork`)
- ...(`execlp (...)`)

IPC - sem

sem = sémaphores

Cette implémentation est plus riche que l'implémentation imaginée par Dijkstra

- ▶ ensembles de sémaphores et opérations atomiques sur l'ensemble
- ▶ Appel Système ZERO - débloquent plusieurs process en même temps (rendez-vous de process)

IPC - sem operations

Sémaphores ensembles

- ▶ Les sémaphores IPC permettent de gérer **un ensemble** de sémaphores et donc de ressources.
- ▶ Les opérations sont définies individuellement pour chaque sémaphore de l'ensemble.
- ▶ Le noyau garantit l'**atomicité de l'ensemble d'opérations**.
- ▶ C'est du "tout ou rien".

IPC - sem operations

Les trois opérations sur un sémaphore IPC System V

- ▶ DOWN
- ▶ UP
- ▶ ZERO // attente de zéro

Les trois opérations sur un sémaphore IPC System V

- DOWN
- UP
- ZERO // attente de zéro

- DOWN bloque un processus tant que le compteur de ressources est $<$ au nombre de ressources demandé. Le compteur de ressources est décrémenté en conséquence.
- UP restitue des ressources, le compteur de ressources est incrémenté en conséquence
- ZERO ou attente de zero bloque un ou plusieurs processus tant que le compteur de ressources est $>$ à 0. N'altère pas le compteur de ressources.

Les processus en attente de 0 sont libérés dès que le compteur bascule à 0 via un down.

structures

Une structure anonyme décrit chaque sémaphore de l'ensemble :

```
unsigned short semval;    /* valeur du sémaphore positive ou nulle */  
                        /* Compteur de ressources */  
unsigned short semzcnt;  /* # En attente du zéro */  
unsigned short semncnt;  /* # En attente de ressource */  
pid_t          sempid;   /* dernier processus agissant */
```

Deux types d'attente et deux files distinctes

- ▶ attente de ressource ($\text{semval} > 0$)
- ▶ attente que semval soit nul (rendez-vous de processus)

semop - appels système

DOWN - ZERO - UP

```
int semop(int semid,  
    struct sembuf *sops, // tableau de sembuf  
    unsigned nsops);    : ul  
// structure sembuf :  
    unsigned short sem_num; /* Numéro du sémaphore 0.. */  
    short          sem_op; /* Opération sur le sémaphore :  
                           <0 (down), 0 (zero), >0 (up) */  
    short          sem_flg; /* IPC_NOWAIT SEM_UNDO */
```

opérations effectuées dans l'ordre et **atomiquement**
renvoie -1 en cas d'échec. Dans ce cas, aucune opération
n'est effectuée.

semop - NOWAIT - UNDO

- ▶ en présence de IPC_NOWAIT, en cas d'indisponibilité, errno vaut EAGAIN
- ▶ l'option SEM_UNDO annule l'opération si le processus se termine (à utiliser à bon escient)

semctl - initialiser, libérer, contrôler

valeur initiale, libération et autres

```
int semctl (semid, semno, cmd, ...);  
int semctl (semid, semno, cmd, us );
```

- ▶ cmd - IPC_RMID(supprimer l'ensemble), GETALL, SETALL, SETVAL
- ▶ us - union semnum à définir par nos soins

la suppression d'un sémaphore est immédiate

semctl - contrôle

```
union semun {  
    int val; /* valeur pour SETVAL */  
    struct semid_ds *buf; /* tampon pour IPC_STAT, IPC_SET */  
    unsigned short *array; /* table pour GETALL, SETALL */  
    struct seminfo *__buf; /* tampon pour IPC_INFO  
                           Spécificité Linux */  
};
```

exemple down

```
int main (void) {  
    int semid;  
    struct sembuf sb_down[] = {{0,-1,0}};  
  
    semid = semget (120, 1, IPC_CREAT | 0666);  
    printf ("valeur_avant: %d\n", semctl(semid,0,GETVAL));  
    semop (semid, sb_down, 1); // bloque si 0 ressources  
    printf ("valeur_après: %d\n", semctl(semid,0,GETVAL));  
}
```

Voir exemples down, up, zero, alt_A et alt_B

attacher - détacher

shm = shared memory (mémoire partagée)

Une mémoire partagée par plusieurs processus est visible via l'espace d'adressage de chacun d'entre eux. Il faut l'**attacher** à cet espace un peu comme un monte un système de fichiers.

attacher - détacher

```
void *shmat(int shmid, const void *shmaddr, int shmflg);  
int shmdt (const void *shmaddr);
```

attacher/détacher un segment de mémoire partagée à/de son espace d'adressage

Un **compteur d'utilisation** est incrémenté ou décrémenté par ces Appels Système

Système d'exploitation

└ IPC

└ IPC system V

└ attacher - détacher

attacher - détacher

```
void shmatt(int shmid, const void *shmaddr, int shmlg);  
int shmatt(const void *shmaddr);
```

attacher/détacher un segment de mémoire partagée à/de son espace d'adressage

Un **compteur d'utilisation** est incrémenté ou décrémenté par ces Appels Système

Le noyau choisit d'attacher une mémoire partagée à une adresse

frontière de cadre de page : une adresse de la forme

0xXXXXXX000

multiple de 4096

attachement - fork, exec, exit

- ▶ fork - le fils hérite des segments attachés (compteur++)
- ▶ exec, exit - les segments sont détachés (pas détruits).

libérer

```
int shmctl(int shmid, int cmd, struct shmids *buf);
```

- ▶ opérations de contrôle
- ▶ cmd=IPC_RMID marqué pour destruction (postposée au dernier detach)

Exemple - attacher

```
static char * c;  
int main() {  
    int shm;  
    struct shmid_ds ds;  
    shm = shmget(123,1,0666|IPC_CREAT);  
    shmctl (shm, IPC_STAT, &ds);  
    printf (" taille _allouée_%d\n", ds.shm_segsz);  
    c=shmat(shm,0,SHM_RDONLY); // SHM_EXEC - 0 (R/W)  
    printf ("adresse_:_%p\n", c);  
    while (*c != 0) write (1,c,1);  
    shmdt(c);  
    exit (0);  
}
```

Exemple - RMID

```
int main()
{
    int shm;
    char t [2];
    shm = shmget(123,1,0777|IPC_CREAT);
    c=shmat(shm,0,0); // (R/W)
    printf ("adresse_:_%p\n", c);
    while (read(0,t,2) == 2 ) *c = t[0];
    *c=0;
    shmctl(shm,IPC_RMID,0);
    shmdt(c);
    exit (0);
}
```

Système d'exploitation

└ IPC

└ IPC system V

└ Exemple - RMID

Exemple - RMID

```
int main()
{
    int shm;
    char t [2];
    shm = shmget(123,1,0777|IPC_CREAT);
    c=shmctl(shm,0,0); // (R/W)
    printf ("%s\n",t);
    while (read(0,t,2) == 2 ) sc = t[0];
    sc=0;
    shmctl(shm,IPC_RMID,0);
    shmctl(c);
    exit (0);
}
```

- Une mémoire partagée est une ressource persistante.
- Elle n'est supprimée qu'après une demande explicite et son détachement (implicite - explicite) par tous les process.
- Un détachement **implicite** a lieu lors d'un exit ou d'un exec

Exemple - affclavier avec fils

```
static char * c;
void detach (int s){ shmtdt(c);};
int main() {
    int shm;
    char t [2];
    struct shmid_ds ds;
    shm = shmget(IPC_PRIVATE,1,0666|IPC_CREAT);
    shmctl (shm, IPC_STAT, &ds);
    printf (" taille _allouée_%d\n", ds.shm_segsz);
    c=shmat(shm,0,0);
    printf ("adresse_:_%p\n", c);
    if (fork()==0) { for(;;) write(1,c,1); exit (0);
    // le fils a hérité de l'id(shm) et du segment de mémoire
    }
    for(;;) { read(0,t,2); *c = t[0]; }
    exit (0);
}
```

Exemples

producteur-consommateur voir prodS.c consS.c alternance
voir alt_A.c, alt_B.c

Questions ?



sémaphores ESI

comment écrire une librairie qui implémente les sémaphores de Dijkstra ?

- ▶ `int ouvrirsem` : crée un sémaphore
- ▶ `int supsem` : détruit le sémaphore
- ▶ `int initsem` : initialise le sémaphore au nombre de ressources disponibles
- ▶ `void down` : demande une ressource
- ▶ `void up` : libère une ressource

ouvrirsem , supsem

```
int ouvrirsem(key_t cle)
{
    int n;
    if ((n=semget(cle,1,0666|IPC_CREAT))==-1)
    {
        perror ("semget");
        exit(-1);
    }
    return(n);
}

int supsem(int sem)
{
    if (semctl(sem,1,IPC_RMID,0)!=0)
    {
        perror ("semctl");
        exit(1);
    }
    return(0);
}
```

initsem

```
int initsem(int sem, int n)
{
    if (semctl(sem,0,SETVAL,n)==-1)
    {
        perror ("semctl");
        exit (1);
    }
    return(0);
}
```

down , up





```
int opsem(int sem, int i)
{
    int n; struct sembuf op[1];
    op[0].sem_num = 0;
    op[0].sem_op = i;
    op[0].sem_flg = 0;
    if ((n=semop(sem,op,1))==-1)
    {
        perror ("semop");
        exit (1);
    }
    return(n);
}

void down(int sem) { opsem(sem,-1); }

void up(int sem) { opsem(sem,+1); }
```

Exemple

```
#include "semaphor.h"
traitement(int sem, char s[])
{ int x=0; char buff[100];
  do { printf("[%d- ]s:je veux lire\n",getpid(),s);
      down(sem); printf("[%s]:en S.C.\n",s); x=read(0,buff,100);
      buff[--x]=0; printf("[%s]:%s\n",s,buff);
      printf("[%s]: quitte la S.C.\n",s); up(sem);
  } while (strcmp(buff,"quit")!=0);
}
main ()
{ int r; int sem;
  sem=ouvrirsem(123);
  initsem(sem,1); // je n'ai qu'un clavier
  if ((r=fork())==0) { traitement(sem,"fils"); exit(0);}
  traitement(sem,"pere");
  wait(0); supsem(sem); exit(0);
}
```

-  Modern Operating Systems Fourth edition - Andrew Tanenbaum, Herbert Bos - Pearson Education
-  Advanced Programming in the UNIX Environment Third Edition - W.Richard Stevens, Stephen A. Rago - Addison Wesley (2014)
-  Programmation Système en C sous Linux 2ième édition - Christophe Blaess - Eyrolles (2005)
-  Intel 64 and IA-32 Architectures Software Developer's Manual - December (2011) (pour toutes les images du chapitre mémoire)

remerciements

merci à P.Bettens et M.Codutti pour la mise en page

Crédits

Ces slides sont le support pour la présentation orale des activités d'apprentissage **SYSIR3** et **SYSG4** à HE2B-ÉSI

Crédits

La distribution opensuse
du système d'exploitation **GNU Linux**.

LaTeX/Beamer comme système d'édition.

GNU make, rubber, pdfnup, ... pour les petites tâches.

Images et icônes

deviantart, flickr, The Noun Project 