

SYSTÈME D'EXPLOITATION I

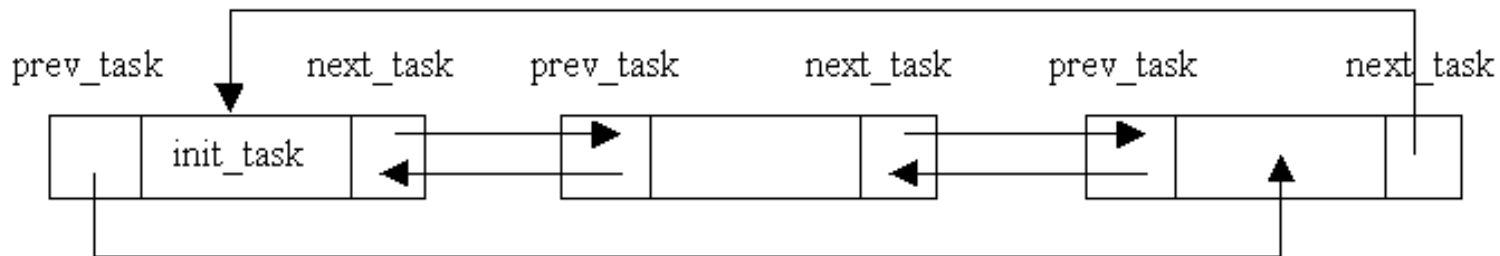
SIF-1015

Contenu du cours 3

- Processus et threads
 - Implémentation
 - Table des processus et files d'attente
 - Concurrency: Exclusion Mutuelle et Synchronisation des processus et des threads
 - Partage de données par exclusion mutuelle
 - Implémentation du problème du Producteur/Consommateur
 - » Synchronisation
 - » Mémoire partagée
 - Implémentation des sémaphores
 - » Pipe
 - LECTURES:
 - Chapitres 3, 4, 6 et 7 (OSC)
 - Chapitres 11 et 13 (Matthew et Stone)
 - Chapitres 12 et 29 (Blaess)
 - Chapitres 4 (Card); 3, 10, 11 (Bovet)
 - Chapitre 12 (Kelley)
 - Chapitres 3, 4 (Mitchell)

Processus et threads (Implémentation)

- Table des processus (process list)
 - Tous les processus (max: 512) dans le système sont contenus dans une liste circulaire doublement chaînée de structures ***task_struct***
 - ***Tête de liste: init_task()*** (processus 0: exécuté quand aucun autre processus n'est ***TASK_RUNNING***, son descripteur permet de repérer le début de la table des processus)
 - ***prev_task***: pointeur vers le processus précédent
 - ***next_task***: pointeur sur le prochain processus

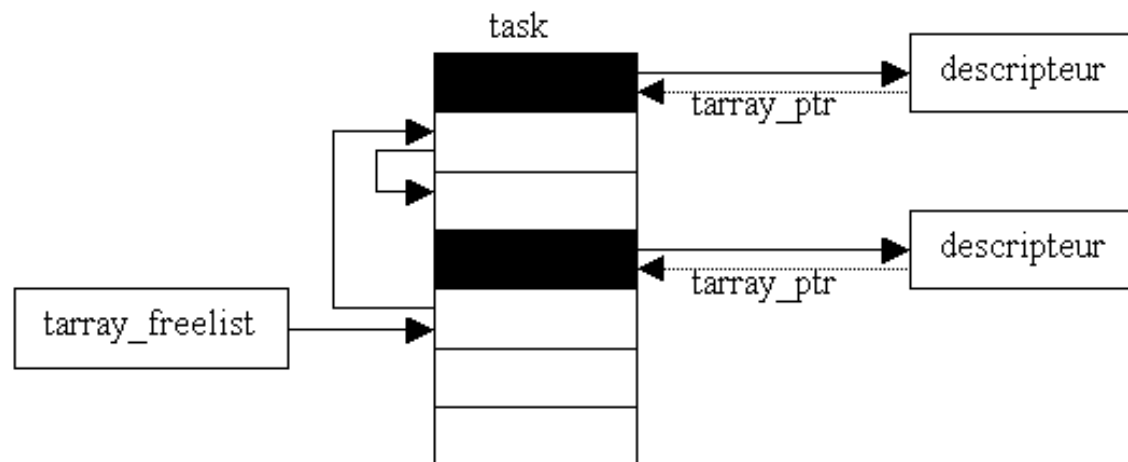


Processus et threads (Implémentation)

- Table des processus
 - Chaque processus dans le système est référencé par un descripteur. Ce descripteur contient les attributs du processus ainsi que les informations permettant sa gestion
 - La structure ***task_struct***, définie dans le fichier ***<linux/sched.h>***, caractérise un processus (voir page 70, Card)
 - Par exemple, l'état d'un processus (champ *state*) peut être exprimé par les constantes:
 - ***TASK_RUNNING***
 - ***TASK_INTERRUPTIBLE***
 - ***TASK_UNINTERRUPTIBLE***
 - ***TASK_ZOMBIE***
 - ***TASK_STOPPED***

Processus et threads (Implémentation)

- Table des processus (organisation)
 - Les descripteurs de processus sont alloués dynamiquement par le noyau par des appels à la fonction ***kmalloc()***
 - Le tableau ***task***, défini dans le fichier source ***kernel/sched.c***, contient des pointeurs sur ces descripteurs. Ce tableau est mis-à-jour à chaque fois qu'un processus est créé ou détruit
 - Pour localiser rapidement un emplacement vide du tableau ***task***, LINUX utilise une liste doublement chaînée (***tarray_freelist***) des emplacements libres du tableau ***task***

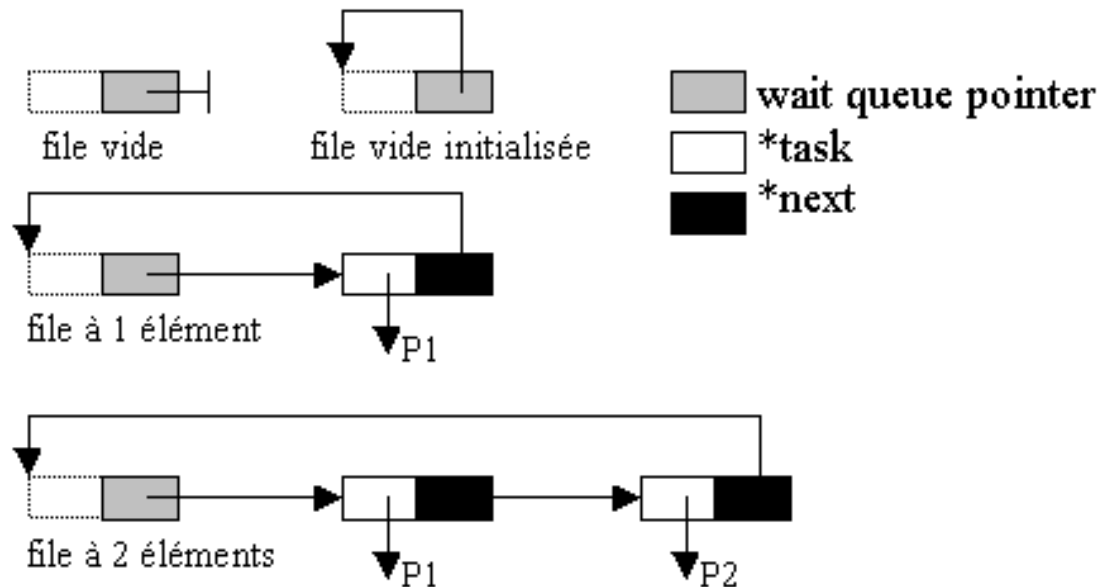


Processus et threads (Implémentation)

- Table des processus (organisation)
 - Les descripteurs de processus prêts ou en cours d'exécution (***TASK_RUNNING***) sont placés dans une liste doublement chaînée (***runqueue***)
 - ***Tête de liste: init_task()***
 - ***nr_running***: variable de ***init_task()*** donnant le nombre de processus prêts à l'exécution
 - ***prev_run***: pointeur sur le dernier processus exécuté
 - ***next_run***: pointeur sur le prochain processus à exécuter

Processus et threads (Implémentation)

- Synchronisation des processus et files d'attente
 - LINUX fournit deux outils permettant aux processus de se synchroniser en mode noyau: les files d'attente (***wait queues***) et les sémaphores (structures définies dans ***<linux/wait.h>***)
 - Une **file d'attente** est une liste chaînée et circulaire de descripteurs. Chaque descripteur (structure ***wait_queue***) contient l'adresse d'un descripteur de processus (***struct task_struct***) ainsi que le pointeur sur l'élément suivant de la file (***struct wait_queue***).



Processus et threads (Implémentation)

- Exclusion mutuelle et Synchronisation des processus et threads
 - La vitesse d'exécution relative des processus est difficile à déterminer d'avance et dépend des actions prises par chaque processus, de la façon que le SE gère les interruptions et les politiques de séquençage du SE:
 - Le partage de ressources globales est dangereux (ex: variables globales accédées par plusieurs processus en lecture/écriture)
 - La gestion de l'allocation des ressources est aussi difficile. (ex: P1 requiert et se voit offrir le contrôle d'un dispositif d'I/O et que P1 est suspendu avant d'avoir utilisé ce dispositif). Le SE ne doit pas simplement verrouiller la ressource et en limiter l'accès à d'autres processus.
 - Il devient difficile de détecter les erreurs de programmation qui sont non déterministes et non reproductibles

Processus et threads (Implémentation)


- Exclusion mutuelle et Synchronisation des processus et threads
 - Un exemple simple de concurrence: Les processus P1 et P2 invoquent la procédure ***echo()*** et accèdent aux variables partagées ***chin*** et ***chout***.

```
void echo()
{
    chin = getchar();
    chout = chin;
    putchar(chout);
}
```

Processus et threads (Implémentation)

- Exclusion mutuelle et Synchronisation des processus et threads
 - Un exemple simple de concurrence: Les processus P1 et P2 invoquent la procédure ***echo()*** et accèdent aux variables partagées ***chin*** et ***chout***.

Process P1

```
.  
chin = getchar();  
.   
chout = chin;  
.   
putchar(chout);  
.   
.
```

Process P2

```
.  
.   
chin = getchar();  
.   
chout = chin;  
.   
putchar(chout);  
.
```

Processus et threads (Implémentation)

- Exclusion mutuelle et Synchronisation des processus et threads

- Interactions

Degree of Awareness	Relationship	Influence that one Process has on the Other	Potential Control Problems
Processes unaware of each other	Competition	<ul style="list-style-type: none"> •Results of one process independent of the action of others •Timing of process may be affected 	<ul style="list-style-type: none"> •Mutual exclusion •Deadlock (renewable resource) •Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	<ul style="list-style-type: none"> •Results of one process may depend on information obtained from others •Timing of process may be affected 	<ul style="list-style-type: none"> •Mutual exclusion •Deadlock (renewable resource) •Starvation •Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	<ul style="list-style-type: none"> •Results of one process may depend on information obtained from others •Timing of process may be affected 	<ul style="list-style-type: none"> •Deadlock (consumable resource) •Starvation

Processus et threads (Implémentation)

- Exclusion mutuelle et Synchronisation des processus et threads
 - Coopération entre processus par partage (exclusion mutuelle)

```
void P(i)
{
    while(TRUE)
    {
        Introduction dans la zone
        // section critique
        Sortie de la zone critique
    }
}

void main()
{
    Démarrage des processus Pi
}
```

Processus et threads (Implémentation)

- Exclusion mutuelle et Synchronisation des processus et threads
 - L'utilisation de l'exclusion mutuelle peut occasionner deux problèmes:
 - L'interblocage (deadlock) survient quand par exemple deux processus P1 et P2 ont besoin de deux ressources R1 et R2 pour accomplir leurs tâches. Si le SE assigne R1 à P2 et R2 à P1. Chaque processus attend donc pour une des ressources (P1 attend R1 et P2 attend R2). P1 et P2 ne libèreront pas R2 et R1 respectivement tant qu'ils n'auront pas acquis l'autre ressource (R1 pour P1 et R2 pour P2) requise pour accomplir leurs tâches.
 - La famine (starvation) survient quand un processus se voit refuser l'accès à une ressource R donnée durant une longue période de temps.

Processus et threads (Implémentation)

- Exclusion mutuelle et Synchronisation des processus et threads
 - Exemple d'interblocage (deadlock):
 - Un ensemble de processus en attente (blocked) chacun retenant une ressource et attendant d'acquérir une ressource retenue par un autre processus de cet ensemble
 - Un système avec deux disques
 - P_1 et P_2 ont chacun un disque et chacun requiert l'autre disque
 - semaphores A et B , initialisés à 1

P_1
wait (A);
wait (B);

P_2
wait(B)
wait(A)

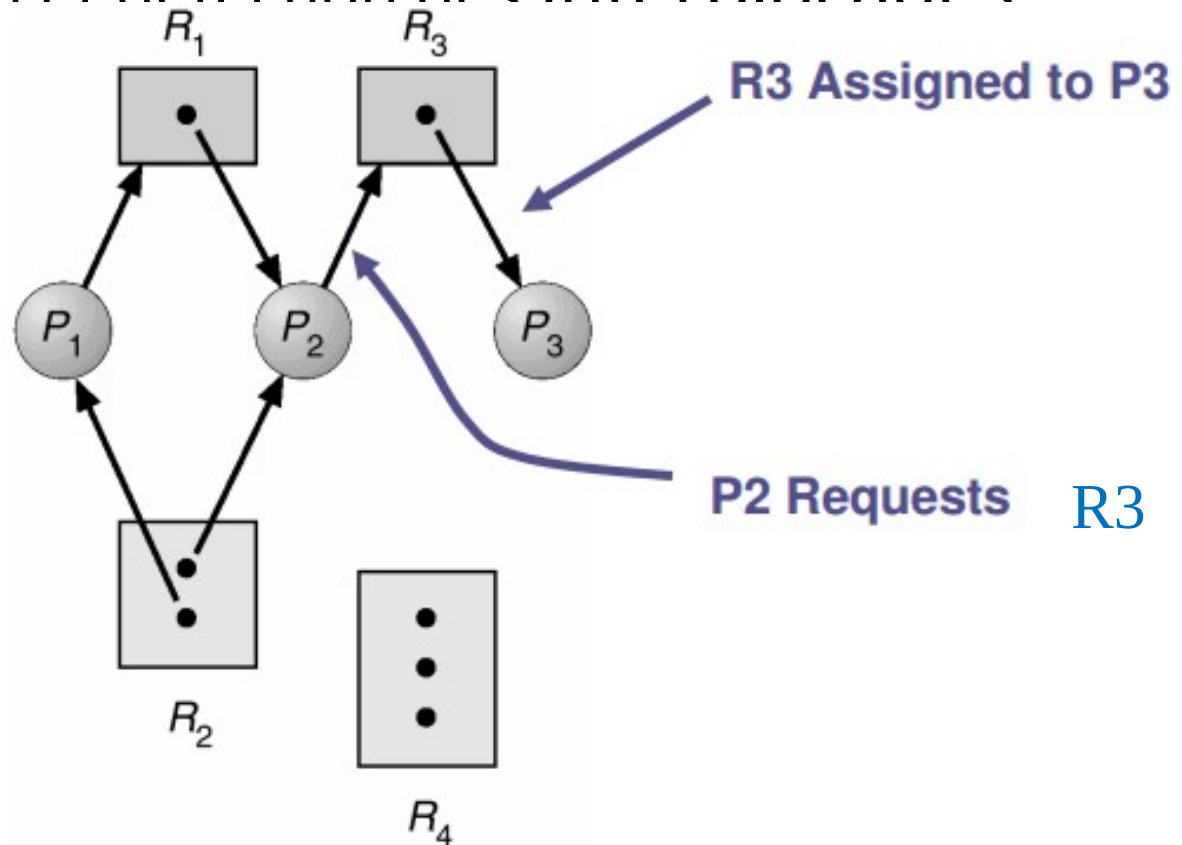
Processus et threads (Implémentation)

- Exclusion mutuelle et Synchronisation des processus et threads
 - Conditions nécessaires pour avoir un interblocage (deadlock):
 - Exclusion mutuelle: Une ou plusieurs ressources sont retenues par un processus en mode exclusif (aucun autre processus ne peut accéder à ces ressources).
 - Retenir/Attendre (Hold/Wait): Un processus retient une ressource et attend d'acquérir une ressource retenue par un autre processus.
 - Pas de préemption (No Preemption): Une ressource sera libérée que de façon volontaire par le processus qui en fait l'utilisation.
 - Attente circulaire (Circular Wait): Le processus P1 attend le processus P2 qui attend le processus P3 qui attend le processus P1
 - Voir le petit vidéo sur youtube décrivant de façon ludique les 4 conditions nécessaires à l'interblocage:
<https://www.youtube.com/watch?v=myomEBjnIDw>

Processus et threads (Implémentation)

- Exclusion mutuelle et Synchronisation des processus et threads
 - Modélisation et détection des interblocages (deadlock)

Graphe d'allocation de ressources



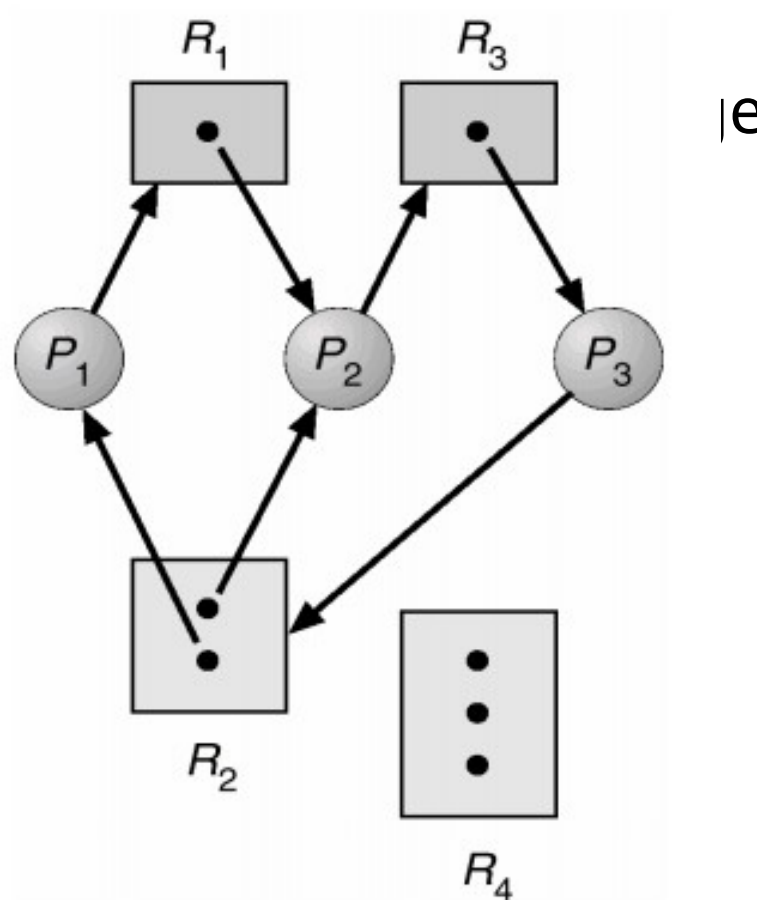
Processus et threads (Implémentation)

- Exclusion mutuelle et Synchronisation des processus et threads
 - Modélisation et détection des interblocages (deadlock):
 - SI le graphe est sans aucun cycle ALORS
 - Pas de processus en interblocage
 - SI un graphe comporte au moins un cycle ALORS
 - SI une ressource possède plusieurs instances ALORS
 - » Il peut survenir un interblocage
 - SI une ressource possède qu'une seule instance ALORS
 - » Un interblocage survient

Processus et threads (Implémentation)

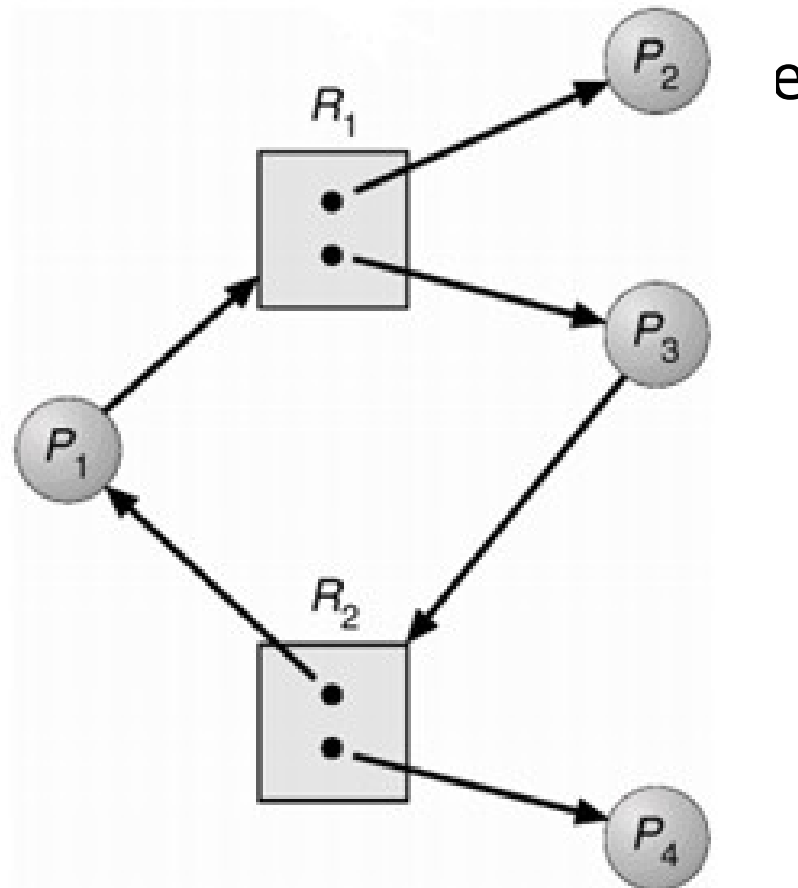
- Exclusion mutuelle et Synchronisation des processus et threads
 - Modélisation et détection des interblocages (deadlock):

- Exemple c



Processus et threads (Implémentation)

- Exclusion mutuelle et Synchronisation des processus et threads
 - Modélisation et détection des interblocages (deadlock).
 - Exemple



Processus et threads (Implémentation)

- Exclusion mutuelle et Synchronisation des processus et threads
 - Stratégies de gestion des interblocages
 - Ignorer les interblocages (Algo. de l'autruche): Plusieurs OS utilisent cette stratégies
 - S'assurer qu'un interblocage ne survienne jamais:
 - Prévention: Voir à ce que les 4 conditions d'occurrence d'un interblocage ne surviennent pas en même temps.
 - Évitement (avoidance): Permettre l'occurrence de ces conditions en simultané mais en calculant les RAG pour détecter les cycles possibles et alors empêcher les opérations dangereuses.
 - Permettre les interblocages
 - Détection: Savoir qu'un interblocage est arrivé.
 - Récupération des ressources.

Processus et threads (Implémentation)

- Exclusion mutuelle et synchronisation des processus et threads
 - Les **sémaphores** permettent principalement de résoudre deux types de problèmes:
 - L'**exclusion mutuelle**: permet d'empêcher deux processus d'accéder à une même ressource en même temps (ex: modification du contenu d'un fichier)
 - Le problème des **producteurs/consommateurs**: permet la coopération de deux processus, un qui produit des informations et l'autre qui les consomme de façon synchronisée
 - Ces problèmes peuvent être résolus par des sémaphores binaires, mais le compteur du sémaphore peut prendre d'autres valeurs positives, qui représentent le nombre d'unités d'une même ressource disponibles, le compteur prend alors la valeur du nombre d'unités accessibles en même temps. Un compteur négatif représente en valeur absolue le nombre de processus en attente de ressource

Processus et threads (Implémentation)

- Exclusion mutuelle et synchronisation des processus et threads
 - Un sémaphore comprend une valeur entière (un compteur) et deux opérations:
 - ***P*** (du hollandais ***proberen***, tester, DOWN, wait, attendre_ressource): cette opération est utilisée quand un processus veut entrer dans une section critique d'un programme
 - ***V*** (du hollandais ***verhogen***, incrémenter, UP, signal, liberer_ressource): cette opération est utilisée quand un processus quitte une section critique d'un programme
 - Utilisation typique:

P(s)

section critique

V(s)

Processus et threads (Implémentation)

- Exclusion mutuelle et synchronisation des processus et threads

P(S) :

si $S > 0$ alors

$S--$

sinon

bloquer le processus en le plaçant dans une file associée à S

V(S) :

$S++$;

si la file associée à S n'est pas vide alors

débloquer (au moins) un processus de la file

Processus et threads (Implémentation)

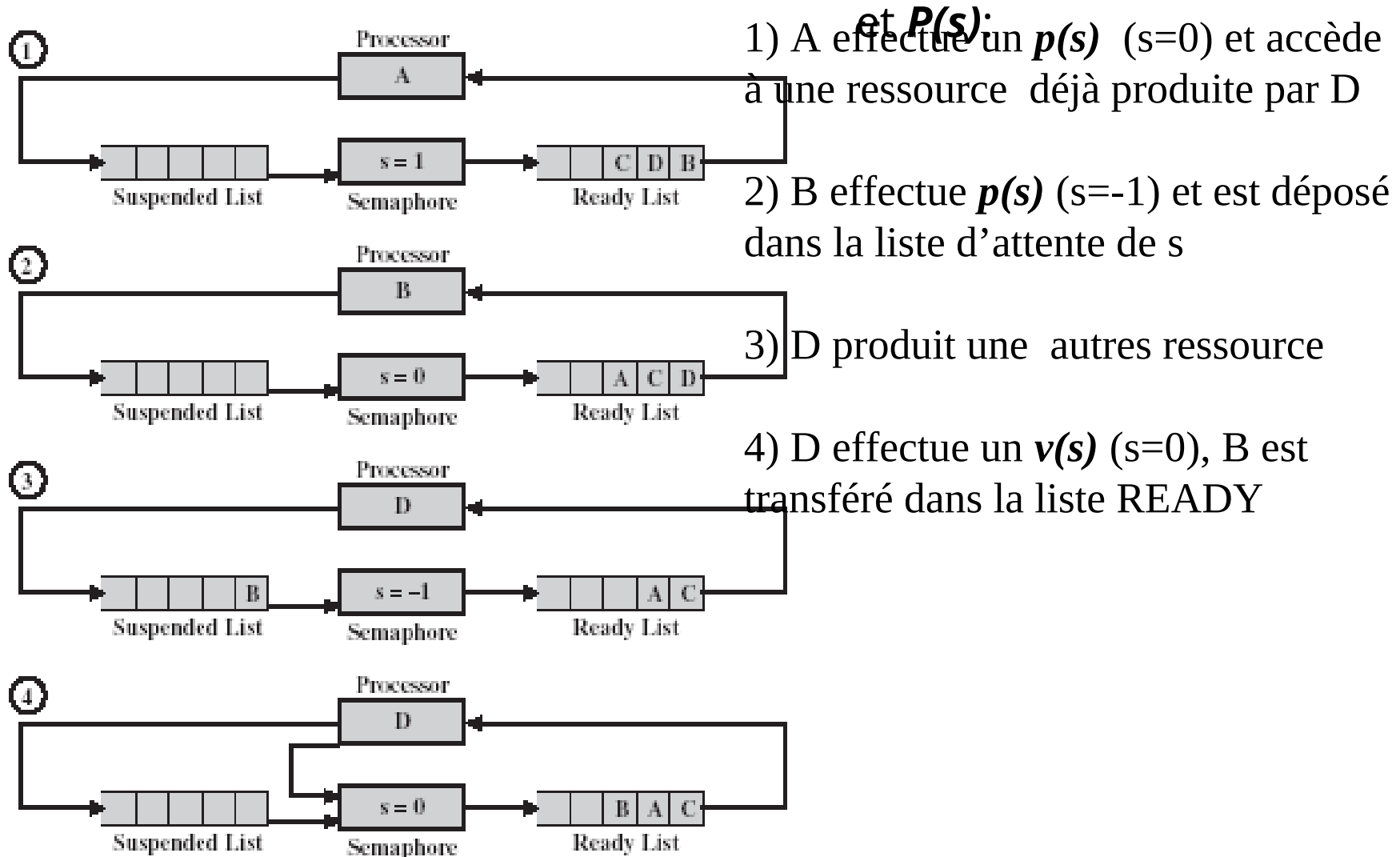
- Exclusion mutuelle et synchronisation des processus et threads
 - Fonctionnement de l'opération **$P(s)$** :
 - Le sémaphore **s** est décrémenter de 1 (**$s--$**)
 - SI compteur du sémaphore **$s > 0$** ALORS
 - Le processus P peut se servir de la ressource requise
 - SI compteur du sémaphore **$s \leq 0$** ALORS
 - Le processus P est endormi jusqu'à ce qu'une ressource soit disponible
 - Le processus P est déposé dans une file d'attente associée à **s**

Processus et threads (Implémentation)

- Exclusion mutuelle et synchronisation des processus et threads
 - Fonctionnement de l'opération **V(s)**:
 - Le sémaphore **s** est incrémenter de 1 (**s++**)
 - SI compteur du sémaphore **s** ≤ 0 ALORS
 - Un processus P est extrait de la file d'attente de **s**
 - P est inséré dans la liste de processus READY
 - SI aucun processus en attente sur **s** ALORS
 - Incrément du compteur du sémaphore **s**
 - **s** est positif représente le nombre de ressources disponible et **s** négatif le nombre en valeur absolue de processus en attente d'une ressource
 - Les opérations **P** et **V** doivent être réalisées de manière atomique (sans interruption) donc en mode noyau (kernel)

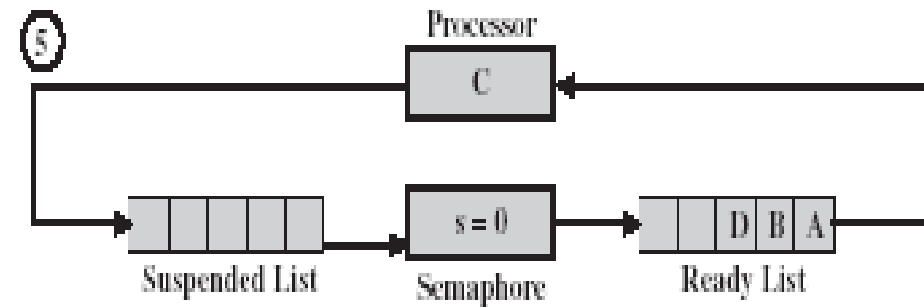
Processus et threads (Implémentation)

- Exclusion mutuelle et synchronisation des processus et threads



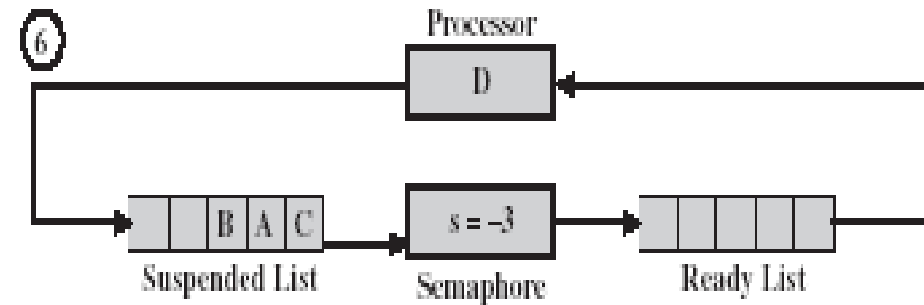
Processus et threads (Implémentation)

- Exclusion mutuelle et synchronisation des processus et threads
 - Fonctionnement des opérations $V(s)$ et $P(s)$:

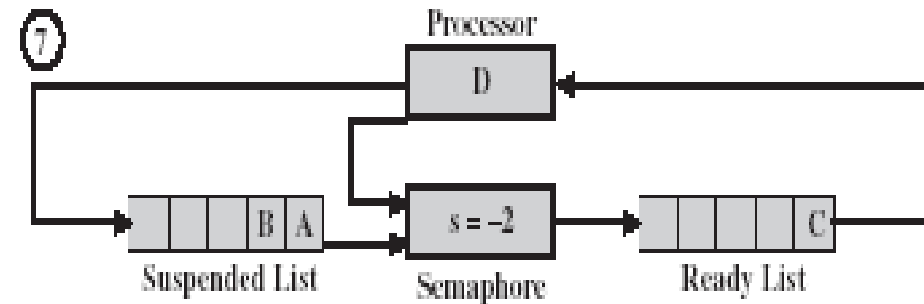


D retourne dans la liste READY, effectue un $p(s)$ ($s=-1$) et est déposé dans la liste d'attente de s

A et B effectuent $p(s)$ ($s=-3$) et sont déposés dans la liste d'attente de s



D produit une ressource et effectue $v(s)$ ($s=-2$), C est transféré dans la liste READY



Processus et threads (Implémentation)

- Exclusion mutuelle et synchronisation des processus et threads
 - Exclusion mutuelle utilisant les sémaphores:
n = 3 // nombre de processus
s = 1 // une ressource disponible, une donnée partagée

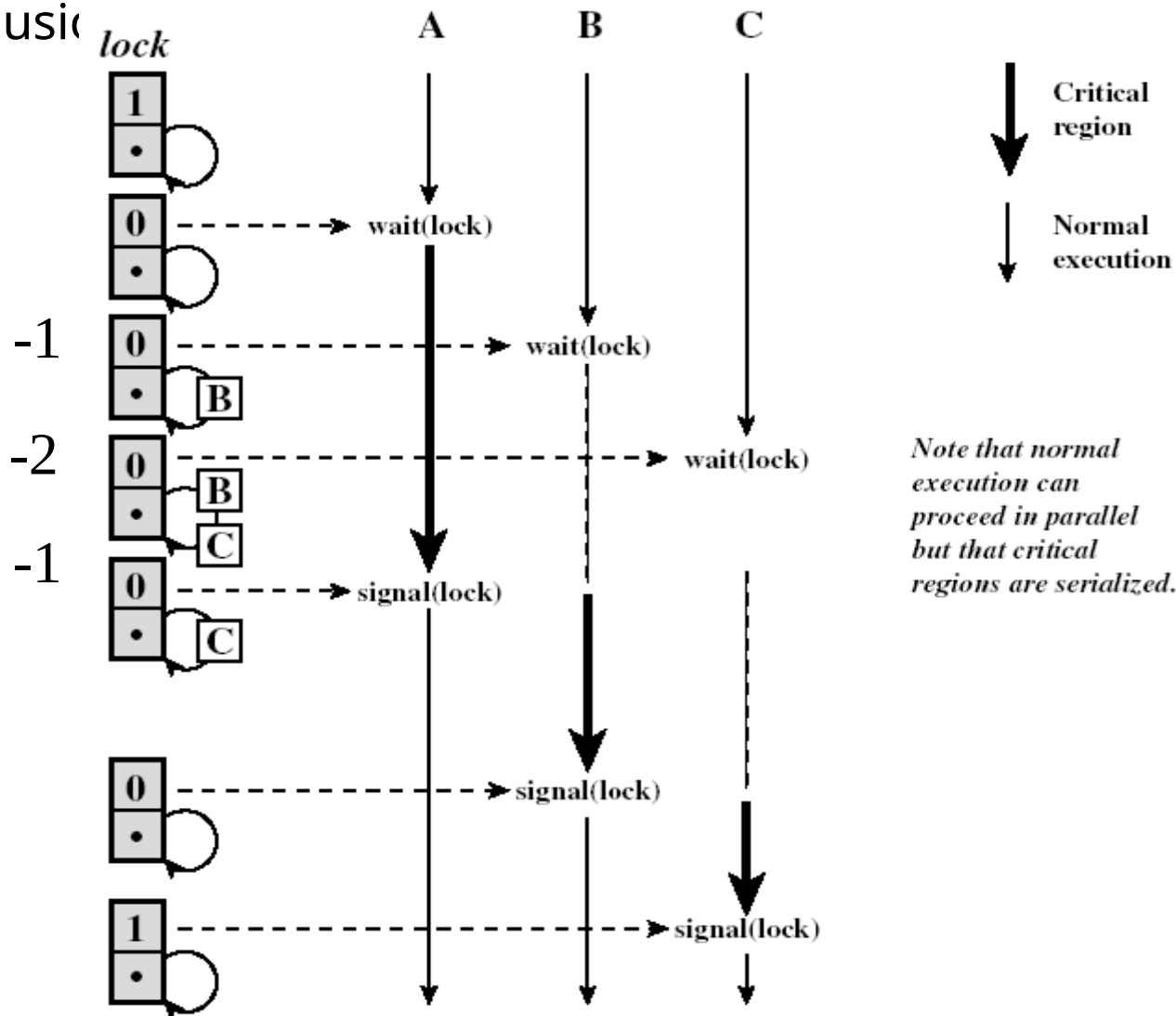
```
void P(i)
{
    while(TRUE)
    {
        p(s)
        // section critique
        v(s)
    }
}
```

```
void main()
{
    Démarrage de A B et C // processus Pi
}
```

Processus et threads (Implémentation)

- Exclusion mutuelle et synchronisation des processus et threads

– Exclusivité



Processus et threads (Implémentation)

- Exclusion mutuelle et synchronisation des processus et threads
 - Implémentation des sémaphores (processus)
 - Définitions des fonctions de sémaphore (appels système):
 - La fonction ***semget()*** permet la création d'un nouveau sémaphore ou d'obtenir la clé d'un sémaphore existant
 - Appel système
- int semget(key_t cle, int num_sems, int drapeaux_sem);***
- » ***cle***: un numéro, identificateur de région mémoire (entier) utilisé par des processus indépendants pour accéder un même sémaphore
 - » ***num_sems***: nombre de sémaphores demandés (généralement 1)
 - » ***drapeaux_sem***: drapeaux similaires à ceux utilisés par la fonction ***open()***. Les neuf bits inférieurs sont les permissions du sémaphore, comme celles des fichiers (RWX/UGO). Ces bits peuvent être combinés par un opérateur OU (|) avec la constante **IPC_CREAT** pour créer un nouveau sémaphore (ignorer si existant). **IPC_CREAT** et **IPC_EXCL** sont combinés pour forcer la création d'un nouveau et unique sémaphore (retourne -1 en cas d'erreur)
- ***semget()*** retourne un nombre positif (identificateur de sémaphore(s)) ou -1 en cas d'échec

Processus et threads (Implémentation)

- Exclusion mutuelle et synchronisation des processus et threads
 - Implémentation des sémaphores (processus)
 - Définitions des fonctions de sémaphore:
 - La fonction ***semop()*** permet de modifier la valeur d'un sémaphore
 - Appel système
 - iut semop(int id_sem, struc sembuf *sem_ops, size_t num_sem_ops);***
 - » ***id_sem***: identificateur de sémaphore tel que retourné par ***semget()***
 - » ***sem_ops***: pointeur sur un tableau de structures ***sembuf***
 - struct sembuf{***
 - short num_sem; // no. du sémaphore 0 si un seul sémaphore***
 - short sem_op; // valeur utilisée pour modifier le sémaphore***
 - // -1 pour l'opération P et +1 pour l'opération V***
 - short sem_flg; // généralement fixé à SEM_UNDO***
 - }***
 - » ***num_sem_ops***: nombre de structures ***semop*** (nombre de sémaphores)

Processus et threads (Implémentation)

- Exclusion mutuelle et synchronisation des processus et threads
 - Implémentation des sémaphores (processus)
 - Définitions des fonctions de sémaphore:
 - La fonction ***semctl()*** permet de contrôler directement les informations relatives à un sémaphore
 - Appel système
iut semctl(int id_sem, int num_sem, int command, ...);
 - » ***id_sem***: identificateur de sémaphore
 - » ***num_sem***: numéro du sémaphore (généralement 0)
 - » ***command***: action à accomplir sur le sémaphore. **SETVAL** permet d'initialiser un sémaphore avec la valeur ***val*** de l'union ***semun***. **IPC_RMID** sert à détruire un identificateur de sémaphore quand il n'est plus utilisé.
 - » Le quatrième paramètre représente si il est présent une ***union semun*** devant posséder les attributs:
union semun{
int val;
struct semid_ds *buf;
unsigned short *array;***}***

Processus et threads (Implémentation)

- Exclusion mutuelle et synchronisation des processus et threads
 - Implémentation de l'exclusion mutuelle par sémaphores
 - Programme ***sem1.c*** pouvant être invoqué plusieurs fois, un paramètre optionnel donné en ligne de commande permettra de définir si le programme doit ou non créer ou détruire le sémaphore binaire qui limitera l'accès à une section critique de programme.
 - L'invocation paramétrée permettra l'affichage de paires de X
 - L'invocation sans paramètre permettra l'affichage de paires de O

Processus et threads (Implémentation)

- Programme ***sem1.c***: fichier inclus ***semun.h***

```
#if defined(__GNU_LIBRARY__) && !defined(_SEM_SEMUN_UNDEFINED)
    /* union semun is defined by including <sys/sem.h> */
#else
    /* according to X/OPEN we have to define it ourselves */
    union semun {
        int val; /* value for SETVAL */
        struct semid_ds *buf; /* buffer for IPC_STAT, IPC_SET */
        unsigned short int *array; /* array for GETALL, SETALL */
        struct seminfo *__buf; /* buffer for IPC_INFO */
    };
#endif
```

Processus et threads (Implémentation)

- Programme ***sem1.c***: Section des ***include*** et prototypes de fonctions

```
/* After the #includes, the function prototypes and the global variable, we come to the
main function. There the semaphore is created with a call to semget, which returns the
semaphore ID. If the program is the first to be called (i.e. it's called with a parameter
and argc > 1), a call is made to set_semvalue to initialize the semaphore and op_char is
set to X. */

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

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

#include "semun.h"

static int set_semvalue(void);
static void del_semvalue(void);
static int semaphore_p(void);
static int semaphore_v(void);

static int sem_id;
```

Processus et threads (Implémentation)

- Programme ***sem1.c***: Création et initialisation du sémaphore, initialisation de ***op_char*** selon le nombre de paramètres passés en ligne de commande

```
int main(int argc, char *argv[])
{
    int i;
    int pause_time;
    char op_char = 'O';

    srand((unsigned int)getpid());

    sem_id = semget((key_t)1234, 1, 0666 | IPC_CREAT);

    if (argc > 1) {
        if (!set_semvalue()) {
            fprintf(stderr, "Failed to initialize semaphore\n");
            exit(EXIT_FAILURE);
        }
        op_char = 'X';
        sleep(2);
    }
}
```

Processus et threads (Implémentation)

- Programme ***sem1.c***:

```
/* Then we have a loop which enters and leaves the critical section ten times.
There, we first make a call to semaphore_p which sets the semaphore to wait, as
this program is about to enter the critical section. */

for(i = 0; i < 10; i++) {

    if (!semaphore_p()) exit(EXIT_FAILURE);
    printf("%c", op_char);fflush(stdout);
    pause_time = rand() % 3;
    sleep(pause_time);
    printf("%c", op_char);fflush(stdout);

/* After the critical section, we call semaphore_v, setting the semaphore available,
before going through the for loop again after a random wait. After the loop, the call
to del_semvalue is made to clean up the code. */

    if (!semaphore_v()) exit(EXIT_FAILURE);

    pause_time = rand() % 2;
    sleep(pause_time);
}

printf("\n%d - finished\n", getpid());

if (argc > 1) {
    sleep(10);
    del_semvalue();
}

exit(EXIT_SUCCESS);
}
```

Processus et threads (Implémentation)

- Fonctions utilitaires de ***sem1.c: set_semvalue()***

```
/* The function set_semvalue initializes the semaphore using the SETVAL command in a  
semctl call. We need to do this before we can use the semaphore. */  
  
static int set_semvalue(void)  
{  
    union semun sem_union;  
  
    sem_union.val = 1;  
    if (semctl(sem_id, 0, SETVAL, sem_union) == -1) return(0);  
    return(1);  
}
```

Processus et threads (Implémentation)

- Fonctions utilitaires de ***sem1.c: del_semaphore()***

```
/* The del_semaphore function has almost the same form, except the call to semctl uses
the command IPC_RMID to remove the semaphore's ID. */

static void del_semaphore(void)
{
    union semun sem_union;

    if (semctl(sem_id, 0, IPC_RMID, sem_union) == -1)
        fprintf(stderr, "Failed to delete semaphore\n");
}
```

Processus et threads (Implémentation)

- Fonctions utilitaires de ***sem1.c: semaphore_p()***

```
/* semaphore_p changes the semaphore by -1 (waiting). */
static int semaphore_p(void)
{
    struct sembuf sem_b;

    sem_b.sem_num = 0;
    sem_b.sem_op = -1; /* P() */
    sem_b.sem_flg = SEM_UNDO;
    if (semop(sem_id, &sem_b, 1) == -1) {
        fprintf(stderr, "semaphore_p failed\n");
        return(0);
    }
    return(1);
}
```


Processus et threads (Implémentation)

- Fonctions utilitaires de ***sem1.c: semaphore_v()***

```
/* semaphore_v is similar except for setting the sem_op part of the sembuf structure to 1,
so that the semaphore becomes available. */
```

```
static int semaphore_v(void)
{
    struct sembuf sem_b;

    sem_b.sem_num = 0;
    sem_b.sem_op = 1; /* V() */
    sem_b.sem_flg = SEM_UNDO;
    if (semop(sem_id, &sem_b, 1) == -1) {
        fprintf(stderr, "semaphore_v failed\n");
        return(0);
    }
    return(1);
}
```

Processus et threads (Implémentation)

- Résultat de l'exécution de ***sem1.c***:

```
$ sem1 1 &  
[1] 1082  
$ sem1  
OOXXOOXXOOXXOOXXOOXXOOXXOOXXOOXXOOXXOOXXOOXX  
1083 - terminé  
1082 - terminé  
$
```

Processus et threads (Implémentation)

- Synchronisation des threads
 - Implémentation du problème des producteurs/consommateurs
 - Création d'un thread producteur
 - Appel système ***pthread_create()***
 - Création d'un thread consommateur
 - Appel système ***pthread_create()***
 - Établir la structure partagée entre le producteur et le consommateur
 - Établir le dispositif de synchronisation par sémaphore

Processus et threads (Implémentation)

- Synchronisation des threads
 - Notion de partage de la mémoire entre threads
 - Un ensemble de threads s'exécutent toujours dans le contexte plus général d'un processus
 - Chaque thread fonctionne avec son contexte spécifique: son ***tid***, sa pile, son pointeur de pile, pointeur de programme, ses registres d'état et ses registres d'usage général
 - Chaque thread partage le reste du contexte associé au processus avec les autres threads: l'espace d'adresse-virtuelle de l'utilisateur qui comprend:
 - Code du programme
 - Données R/W
 - Le heap (pour la mémoire allouée par des fonctions ***malloc()***)
 - Le code des bibliothèques partagées
 - Les threads partagent aussi le même ensemble de fichiers ouverts et les gestionnaires de signaux

Processus et threads (Implémentation)

- Synchronisation des threads
 - Notion de partage de la mémoire entre threads (Site csapp, sharing.

```
#include "csapp.h"
#define N 2
void *thread(void *vargp);

char **ptr; /* global variable */

int main()
{
    int i;
    pthread_t tid;
    char *msgs[N] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < N; i++)
        Pthread_create(&tid, NULL, thread, (void *)i);
    Pthread_exit(NULL);
}

void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int cnt = 0;
    printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
}
```

Processus et threads (Implémentation)

Var. globale: 1 instance (ptr [data]), partagée

Var. locale automatique: 1 instance (msgs)

```
char **ptr; /* global */

int main()
{
    int i;
    pthread_t tid;
    char *msgs[N] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    Pthread_exit(NULL);
}
```

Var. locale automatique: 2 instances (myid dans p0 [pile du thread p0], myid dans p1 [pile du thread p1])

```
/* thread routine */
void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int cnt = 0;

    printf("[%d]: %s (cnt=%d)\n",
        myid, ptr[myid], ++cnt);
}
```

Var. locale statique: 1 instance (cnt [data]) partagée

Processus et threads (Implémentation)

- Threads mal synchronisés (Site csapp, badcnt.c)

```
#include "csapp.h"

#define NITERS 100000000
void *count(void *arg);

/* shared counter variable */
unsigned int cnt = 0;

int main()
{
    pthread_t tid1, tid2;

    Pthread_create(&tid1, NULL, count, NULL);
    Pthread_create(&tid2, NULL, count, NULL);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    if (cnt != (unsigned)NITERS*2)
        printf("BOOM! cnt=%d\n", cnt);
    else
        printf("OK cnt=%d\n", cnt);
    exit(0);
}

/* thread routine */
void *count(void *arg)
{
    int i;
    for (i = 0; i < NITERS; i++)
        cnt++;
    return NULL;
}
```

➤ gcc -c csapp.c

➤ gcc -o badcnt badcnt.c csapp.o -lpthread

```
linux> badcnt
BOOM! ctr=198841183
```

```
linux> badcnt
BOOM! ctr=198261801
```

```
linux> badcnt
BOOM! ctr=198269672
```

**ctr devrait être
égal à 200,000,000.
Quel est le problème?!**

Processus et threads (Implémentation)

- Code assembleur de la boucle de comptage

Code C

```
for (i=0; i<NITERS; i++)  
    ctr++;
```

Code asm correspondant du thread i (gcc -O0 -fforce-mem)

eax <- ctr Load ctr (L_i)
edx <- eax + 1 Update ctr (U_i)
ctr <- edx Store ctr (S_i)

Head (H_i)

Tail (T_i)

```
.L9:  
    movl -4(%ebp),%eax  
    cmpl $99999999,%eax  
    jle .L12  
    jmp .L10  
-----  
.L12:  
    movl ctr,%eax          # Load  
    leal 1(%eax),%edx      # Update  
    movl %edx,ctr          # Store  
-----  
.L11:  
    movl -4(%ebp),%eax  
    leal 1(%eax),%edx  
    movl %edx,-4(%ebp)  
    jmp .L9  
-----  
.L10:
```


Processus et threads (Implémentation)

- Synchronisation par les sémaphores
 - Solution: Opération P et V de Dijkstra sur des *sémaphores*.
 - *sémaphore*: variable entière de synchronisation
 - P(s): [while (s == 0) wait(); s - - ;]
 - V(s): [s++ ;]

Processus et threads (Implémentation)

- Synchronisation par les sémaphores

- Sémaphore POSIX (voir csapp.c)

```
/* initialize semaphore sem to value */
/* pshared=0 if thread, pshared=1 if process */
void Sem_init(sem_t *sem, int pshared, unsigned int value)
{
    if (sem_init(sem, pshared, value) < 0)
        unix_error("Sem_init");
}
/* P operation on semaphore sem */
void P(sem_t *sem) {
    if (sem_wait(sem))
        unix_error("P");
}

/* V operation on semaphore sem */
void V(sem_t *sem) {
    if (sem_post(sem))
        unix_error("V");
}
```

Processus et threads (Implémentation)

- Synchronisation par les sémaphores
 - Sémaphore POSIX (Autre fonction importante)
- **`int sem_destroy(sem_t sem);`**
 - Permet de détruire le sémaphore `sem`

Processus et threads (Implémentation)

- Synchronisation par les sémaphores

```
/* goodcnt.c - properly sync'd
counter program */
#include "csapp.h"
#define NITERS 10000000

unsigned int cnt; /* counter */
sem_t sem;        /* semaphore */

int main() {
    pthread_t tid1, tid2;

    Sem_init(&sem, 0, 1);

    /* create 2 threads and wait */
    ...

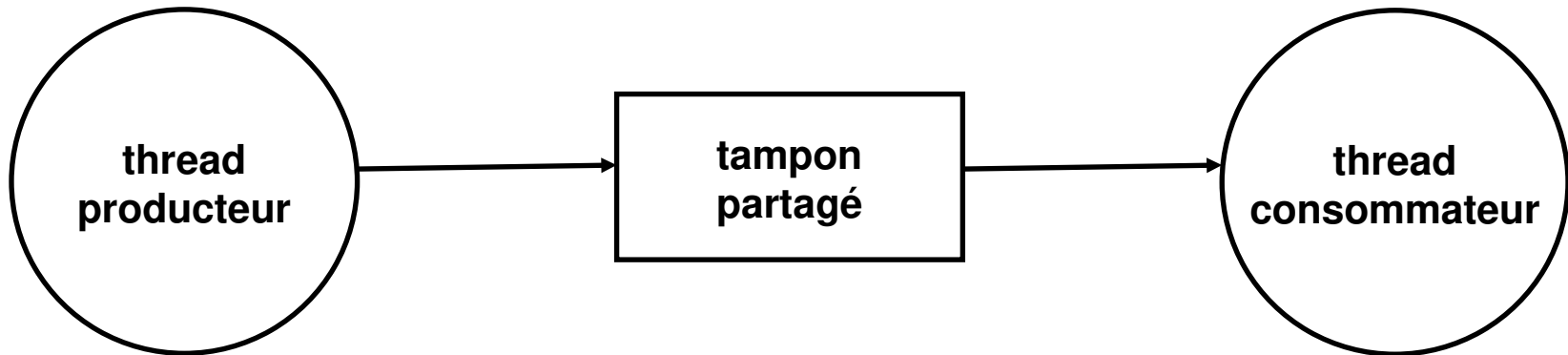
    if (cnt != (unsigned)NITERS*2)
        printf("BOOM! cnt=%d\n", cnt);
    else
        printf("OK cnt=%d\n", cnt);
    exit(0);
}
```

```
/* thread routine */
void *count(void *arg)
{
    int i;

    for (i=0; i<NITERS; i++) {
        P(&sem);
        cnt++;
        V(&sem);
    }
    return NULL;
}
```

Processus et threads (Implémentation)

- Synchronisation par les sémaphores
(Producteur/Consommateur)



Processus et threads (Implémentation)

- Synchronisation par les sémaphores (Producteur/Consommateur):
 - Producteur attend pour un emplacement vide, insert un objet dans le tampon et signal la présence de l'objet au consommateur
 - Consommateur attend pour un objet, l'enlève du tampon et signal au producteur qu'un espace est vide
- Exemples
 - Traitement multimédia:
 - Producteur génère les frames vidéo MPEG, le consommateur affiche les frames
 - GUI Event-driven
 - Producteur détecte des clicks de souris, des déplacements de souris, et touches enfoncées au clavier et insère ces événements dans un tampon approprié
 - Consommateur retire ces événements et rafraîchit l'écran

Processus et threads (Implémentation)

- Synchronisation par les sémaphores (Producteur/Consommateur):

```
/* buf1.c - producer-consumer
on 1-element buffer */
#include "csapp.h"

#define NITERS 5

void *producer(void *arg);
void *consumer(void *arg);

struct {
    int buf; /* shared var */
    sem_t full; /* sems */
    sem_t empty;
} shared;
```

```
int main() {
    pthread_t tid_producer;
    pthread_t tid_consumer;

    /* initialize the semaphores */
    Sem_init(&shared.empty, 0, 1);
    Sem_init(&shared.full, 0, 0);

    /* create threads and wait */
    Pthread_create(&tid_producer, NULL,
                  producer, NULL);
    Pthread_create(&tid_consumer, NULL,
                  consumer, NULL);
    Pthread_join(tid_producer, NULL);
    Pthread_join(tid_consumer, NULL);

    exit(0);
}
```

Processus et threads (Implémentation)

- Synchronisation par les sémaphores (Producteur/Consommateur):

Initialement: empty = 1, full = 0.

```
/* producer thread */
void *producer(void *arg) {
    int i, item;

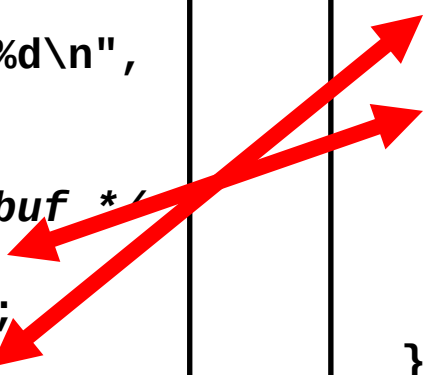
    for (i=0; i<NITERS; i++) {
        /* produce item */
        item = i;
        printf("produced %d\n",
            item);

        /* write item to buf */
        P(&shared.empty);
        shared.buf = item;
        V(&shared.full);
    }
    return NULL;
}
```

```
/* consumer thread */
void *consumer(void *arg) {
    int i, item;

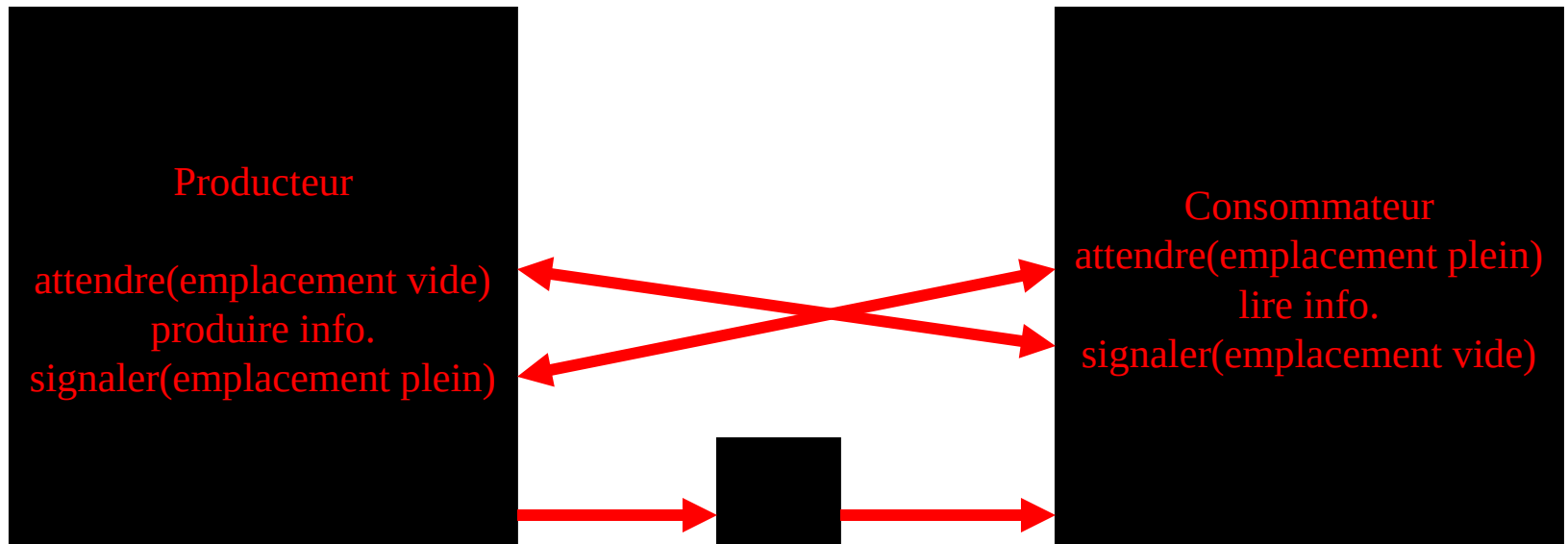
    for (i=0; i<NITERS; i++) {
        /* read item from buf */
        P(&shared.full);
        item = shared.buf;
        V(&shared.empty);

        /* consume item */
        printf("consumed %d\n",
            item);
    }
    return NULL;
}
```



Processus et threads (Implémentation)

- Synchronisation des threads
 - Implémentation du problème des producteurs/consommateurs synchronisé

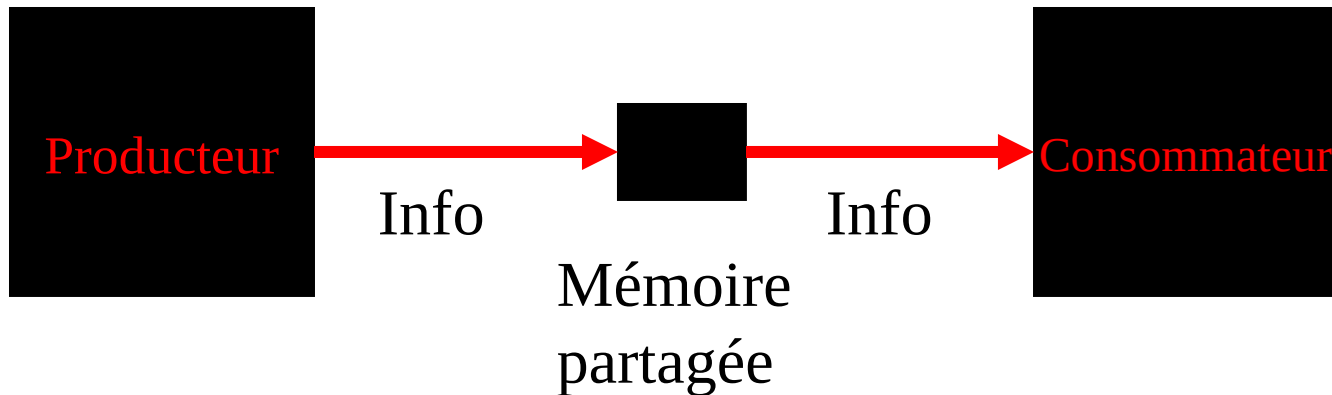


Processus et threads (Implémentation)

- Synchronisation des processus
 - Implémentation du problème des producteurs/consommateurs
 - Création d'un processus producteur
 - Appel système ***fork()***
 - Création d'un processus consommateur
 - Appel système ***fork()***
 - Établir la structure partagée entre le producteur et le consommateur
 - Création d'espaces mémoire partagés avec un appel système ***shmget()***
 - Accès aux espaces mémoire partagés avec un appel système ***shmat()***
 - Détachement à une zone de mémoire partagée par un appel système ***shmdt()***
 - Établir le dispositif de synchronisation par sémaphore
 - Création d'un pipe par l'appel ***pipe()***
 - Opération ***attendre_ressource()*** par un appel système ***read()*** sur la pipe
 - Opération ***liberer_ressource()*** par un appel système ***write()*** sur la pipe

Processus et threads (Implémentation)

- Synchronisation des processus
 - Implémentation du problème des producteurs/consommateurs



Processus et threads (Implémentation)

- Synchronisation des processus
 - Implémentation du problème des producteurs/consommateurs non synchronisé

Ce programme permet d'échanger une séquence de nombres en un producteur et un consommateur de façon non-synchronisée

```
_____/
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

→ #define SHMKEY 75

int shmid; /* identificateur de memoire partagee (global) */

/* _____
Programme principal.
_____ */
void main (void)
{
    int *pt_n; /* pointeur sur un entier partagé entre 2 processus */

    void producteur(void), consommateur(void);

→ if((shmid=shmget(SHMKEY,sizeof(int),IPC_CREAT|0600))==1)
    printf("Erreur avec le shmget dans MAIN \n ");

→ if((pt_n=(int *)shmat(shmid,0,0))==NULL)
    printf("Erreur avec shmat dans MAIN \n");

    *pt_n = 0; /* initialisation du contenu pointer par pt_n */

→ if(fork()==0)
    producteur();
    else
    consommateur();

    printf("Valeur du contenu pointe par pt_n dans MAIN : %d \n",*pt_n);

→ if(shmdt((char *)pt_n) != 0)
    printf("Erreur avec shmdt dans MAIN \n ");

} /* main */
```

Processus et threads (Implémentation)

- Synchronisation des processus
 - Implémentation du problème des producteurs/consommateurs non synchronisé (exemple: ex7PRODCONS.c)
 - **Mémoire partagée:**
 - ***shmget()***: permet de créer un segment de mémoire partagée
 - ***shmat()***: permet d'attacher un pointeur sur un segment de mémoire partagée (retourne une adresse sur la segment de mémoire partagée)
 - ***shmdt()***: permet de détacher un pointeur d'un segment de mémoire partagée

Processus et threads (Implémentation)

- Synchronisation des processus
 - Implémentation du problème des producteurs/consommateurs non synchronisés

```
/* producteur: produit une sequence de 20 nombres de 1 à 20 */
void producteur(void)
{
    int i;
    int *pt_n_prod;

    /* Fait pointer pt_n_prod sur le meme espace memoire partage */
    if((pt_n_prod=(int *)shmat(shmid,0,0))==NULL)
        printf("Erreur avec shmat dans PRODUCTEUR \n");

    for(i=1;i<=20;i++)
        (*pt_n_prod)++;

    if(shmdt((char *)pt_n_prod) != 0)
        printf("Erreur avec shmdt dans PRODUCTEUR \n ");

}

/* consommateur: consomme 20 nombres, les imprime et se termine */
void consommateur(void)
{
    int i;
    int *pt_n_cons;

    /* Fait pointer pt_n_cons sur le meme espace memoire partage */
    if((pt_n_cons=(int *)shmat(shmid,0,0))==NULL)
        printf("Erreur avec shmat dans CONSOMMATEUR \n");

    for(i=1;i<=20;i++)
        printf("Valeur pointe par pt_n_cons: %d \n ",*pt_n_cons);

    if(shmdt((char *)pt_n_cons) != 0)
        printf("Erreur avec shmdt dans CONSOMMATEUR \n ");

}
```

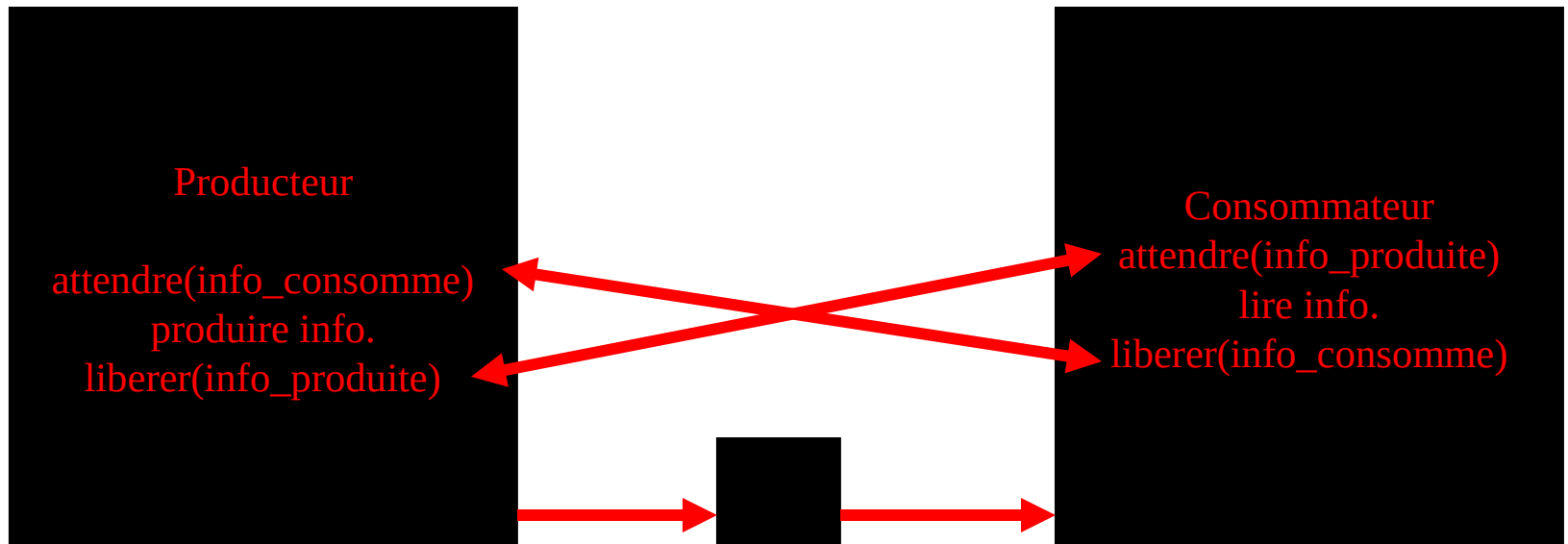
Processus et threads (Implémentation)

- Synchronisation des processus
 - Implémentation du problème des producteurs/consommateurs non synchronisé

```
helios> ex7PRODCONS
Valeur pointe par pt_n_cons: 0
Valeur pointe par pt_n_cons: 20
Valeur pointe par pt_n_cons: 20
Valeur pointe par pt_n_cons: 20
Valeur pointe par pt_n_cons: 20
Valeur pointe par pt_n_cons: 20
Valeur pointe par pt_n_cons: 20
Valeur pointe par pt_n_cons: 20
Valeur pointe par pt_n_cons: 20
Valeur pointe par pt_n_cons: 20
Valeur pointe par pt_n_cons: 20
Valeur pointe par pt_n_cons: 20
Valeur pointe par pt_n_cons: 20
Valeur pointe par pt_n_cons: 20
Valeur du contenu pointe par pt_n dans MAIN : 20
Valeur pointe par pt_n_cons: 20
Valeur pointe par pt_n_cons: 20
Valeur pointe par pt_n_cons: 20
Valeur pointe par pt_n_cons: 20
Valeur pointe par pt_n_cons: 20
Valeur du contenu pointe par pt_n dans MAIN : 20
helios>
```

Processus et threads (Implémentation)

- Synchronisation des processus
 - Implémentation du problème des producteurs/consommateurs synchronisé

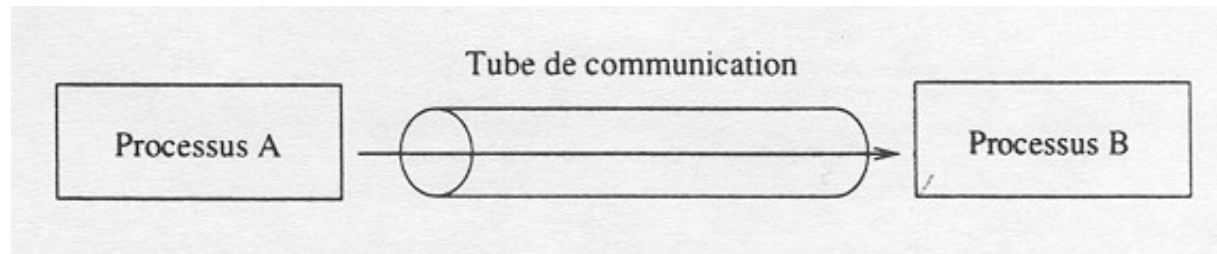


Processus et threads (Implémentation)

- Synchronisation des processus
 - Implémentation du problème des producteurs/consommateurs synchronisé
 - Sémaphores:
 - **info_consomme**: permet de vérifier si une information est consommée
 - » **info_consomme == 1** => espace mémoire commun vide, information non disponible
 - » **info_consomme == 0** => espace mémoire commun plein, information disponible
 - **info_produite**: permet de vérifier si une information est produite
 - » **info_produite == 1** => espace mémoire commun plein, information disponible
 - » **info_produite == 0** => espace mémoire commun vide, information non disponible
 - Opérations:
 - **attendre()**: permet de prendre possession d'une ressource
 - » **info_consomme**: ressource est un espace vide
 - » **info_produite**: ressource est une information disponible
 - **liberer()**: permet de libérer une ressource
 - » **info_consomme**: ressource est un espace vide
 - » **info_produite**: ressource est une information disponible

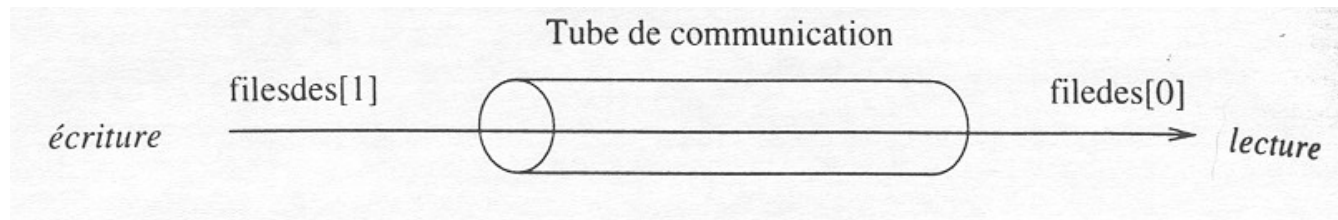
Processus et threads (Implémentation)

- Synchronisation des processus
 - Implémentation du problème des producteurs/consommateurs synchronisé
 - Sémaphores implémentés avec des tubes (pipes)
 - Les tubes représentent un mécanisme de communication entre processus. La transmission des données entre processus s'effectue à travers un canal de communication: les données écrites à une extrémité sont lues à l'autre extrémité



Processus et threads (Implémentation)

- Synchronisation des processus
 - Implémentation du problème des producteurs/consommateurs synchronisé
 - Sémaphores implémentés avec des tubes (pipes)
 - Tubes anonymes: Créer par un appel système ***pipe()***
 - » ***int pipe(int filedes[2]);***
 - » ***filedes[0]***: descripteur de lecture du tube
 - » ***filedes[1]***: descripteur d'écriture du tube



Processus et threads (Implémentation)

- Synchronisation des processus
 - Implémentation du problème des producteurs/consommateurs synchronisé
 - Sémaphores implémentés avec des tubes (pipes)
 - Un tube correspond alors à un sémaphore
 - Chaque écriture dans un tube (**filedes[1]**) correspond à la libération d'une ressource (**liberer()**)
 - Un tube dans lequel nous écrivons n caractères correspond à n ressources disponibles
 - Chaque lecture dans un tube (**filedes[0]**) correspond à l'attente d'une ressource (**attendre()**)
 - Un tube dans lequel nous faisons une lecture correspond à la prise de contrôle d'une ressource
 - Quand un tube est vide, une lecture du tube (**attendre()**) sera alors bloquée

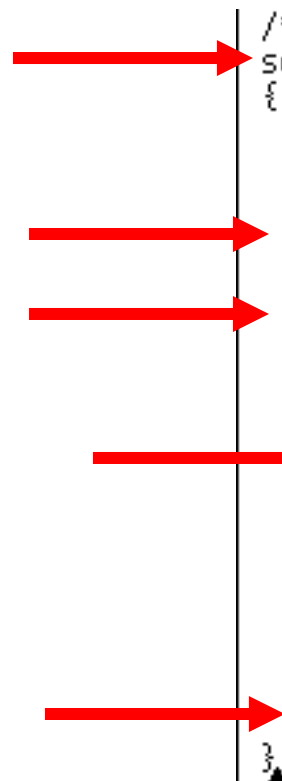
Processus et threads (Implémentation)

- Synchronisation des processus
 - Implémentation du problème des producteurs/consommateurs synchronisé
 - Sémaphores implémentés avec des tubes (pipes)

```
/* creer_sema: creation d'un semaphore */
semaphore creer_sema(int vi)
{
    semaphore sema;
    int i;

    sema=calloc(2,sizeof(int)); /* un ptr sur 2 entiers */
    pipe(sema);

    if(vi >= 0)
    {
        for(i=1; i<=vi; i++)
            liberer_ressource(sema);
    }
    else
    {
        printf("Valeur de semaphore negative non permise \n ");
        exit(1);
    }
    return(sema);
}
```

A vertical line with five red arrows pointing to the following lines of code: the opening brace of the function, the calloc line, the pipe line, the for loop, and the return statement.

Processus et threads (Implémentation)

- Synchronisation des processus
 - Implémentation du problème des producteurs/consommateurs synchronisé
 - Sémaphores implémentés avec des tubes (pipes)

```
/* attendre_ressource: primitive attendre, bloque si sem == 0 (pipe vide) */
void attendre_ressource(semaphore s)
{
    int junk;

    if(read(s[0], &junk, 1) <= 0) /* Lecture dans la pipe */
    {
        printf("Erreur avec attendre_ressource() \n ");
        exit(1);
    }
}

/* liberer_ressource: primitive liberer, sem++ (écriture dans la pipe) */
void liberer_ressource(semaphore s)
{
    if(write(s[1], "x", 1) <= 0) /* Écriture dans la pipe */
    {
        printf("Erreur avec liberer_ressource() \n ");
        exit(1);
    }
}
```

Processus et threads (Implémentation)

- Synchronisation des processus
 - Implémentation du problème des producteurs/consommateurs synchronisés

```
/*
Programme : ex8PRODCONS.C
Langage   : C sur UNIX
Date      : Septembre 2001

Auteur    : F. Meunier

Ce programme permet d'échanger une sequence de nombres entre un producteur et un
consommateur de facon synchronise
*/

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHMKEY 75

typedef int *semaphore;

semaphore creer_sema(int vi); /* valeur initiale du sem >= 0 */
void liberer_ressource(semaphore s);
void attendre_ressource(semaphore s);

int shmid; /* identificateur de memoire partagee (global) */
semaphore info_consomme, info_produite;
```

Processus et threads (Implémentation)

- Synchronisation des processus
 - Implémentation du problème des producteurs/consommateurs synchronisé


```
/*  
Programme principal.  
*/  
void main (void)  
{  
    int *pt_n; /* pointeur sur un entier partagé entre 2 processus */  
    void producteur(void), consommateur(void);  
    if((shmid=shmget(SHMKEY,sizeof(int),0600|IPC_CREAT))==-1)  
        printf("Erreur avec le shmget dans MAIN \n");  
    if((pt_n=(int *)shmat(shmid,0,0))==NULL)  
        printf("Erreur avec shmat dans MAIN \n");  
    *pt_n = 0; /* initialisation du contenu pointer par pt_n */  
    info_produite=creer_sema(0);  
    info_consomme=creer_sema(1); /* permet la production du premier nombre */  
    if(fork()==0)  
        producteur();  
    else  
        consommateur();  
    printf("Valeur du contenu pointe par pt_n dans MAIN : %d \n",*pt_n);  
    if(shmdt((char *)pt_n) != 0)  
        printf("Erreur avec shmdt dans MAIN \n");  
} /* main */
```


Processus et threads (Implémentation)


- Synchronisation des processus
 - Implémentation du problème des producteurs/consommateurs synchronisé

```
/* producteur: produit une sequence de 20 nombres de 1 à 20 */
void producteur(void)
{
    int i;
    int *pt_n_prod;

    /* Fait pointer pt_n_prod sur le meme espace memoire partage */
    if((pt_n_prod=(int *)shmat(shmid,0,0))==NULL)
        printf("Erreur avec shmat dans PRODUCTEUR \n");

    for(i=1;i<=20;i++)
    {

        attendre_ressource(info_consomme);
        (*pt_n_prod)++;
        liberer_ressource(info_produite);
    }

    if(shmdt((char *)pt_n_prod) != 0)
        printf("Erreur avec shmdt dans PRODUCTEUR \n ");

    }

```

Processus et threads (Implémentation)

- Synchronisation des processus
 - Implémentation du problème des producteurs/consommateurs synchronisé

```
/* consommateur: consomme 20 nombres, les imprime et se termine */
void consommateur(void)
{
    int i;
    int *pt_n_cons;

    /* Fait pointer pt_n_cons sur le meme espace memoire partage */
    if((pt_n_cons=(int *)shmat(shmid,0,0))==NULL)
        printf("Erreur avec shmat dans CONSOMMATEUR \n");

    for(i=1;i<=20;i++)
    {
        attendre_ressource(info_produite);
        printf("Valeur pointee par pt_n_cons: %d \n ",*pt_n_cons);
        liberer_ressource(info_consomme);
    }

    if(shmdt((char *)pt_n_cons) != 0)
        printf("Erreur avec shmdt dans CONSOMMATEUR \n ");
}
```

Processus et threads (Implémentation)

- Synchronisation des processus
 - Implémentation du problème des producteurs/consommateurs synchronisé

```
helios> ex8PRODCONS
Valeur pointe par pt_n_cons: 1
Valeur pointe par pt_n_cons: 2
Valeur pointe par pt_n_cons: 3
Valeur pointe par pt_n_cons: 4
Valeur pointe par pt_n_cons: 5
Valeur pointe par pt_n_cons: 6
Valeur pointe par pt_n_cons: 7
Valeur pointe par pt_n_cons: 8
Valeur pointe par pt_n_cons: 9
Valeur pointe par pt_n_cons: 10
Valeur pointe par pt_n_cons: 11
Valeur pointe par pt_n_cons: 12
Valeur pointe par pt_n_cons: 13
Valeur pointe par pt_n_cons: 14
Valeur pointe par pt_n_cons: 15
Valeur pointe par pt_n_cons: 16
Valeur pointe par pt_n_cons: 17
Valeur pointe par pt_n_cons: 18
Valeur pointe par pt_n_cons: 19
Valeur pointe par pt_n_cons: 20
Valeur du contenu pointe par pt_n dans MAIN : 20
Valeur du contenu pointe par pt_n dans MAIN : 20
helios>.
```

Processus et threads (Implémentation)

- Synchronisation et exclusion mutuelle des processus (thread)
 - MUTEX

Without Mutex	With Mutex
<pre>int counter=0; /* Function C */ void functionC() { counter++ }</pre>	<pre>/* Note scope of variable and mutex are the same */ pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER; int counter=0; /* Function C */ void functionC() { pthread_mutex_lock(&mutex1); counter++ pthread_mutex_unlock(&mutex1); }</pre>

Processus et threads (Implémentation)

- Synchronisation et exclusion mutuelle des processus (thread)
 - MUTEX (*pthread_mutex_init()*)

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex,
    const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

DESCRIPTION

The *pthread_mutex_init()* function initialises the mutex referenced by *mutex* with attributes specified by *attr*. If *attr* is NULL, the default mutex attributes are used; the effect is the same as passing the address of a default mutex attributes object. Upon successful initialisation, the state of the mutex becomes initialised and unlocked.

Attempting to initialise an already initialised mutex results in undefined behaviour.

The *pthread_mutex_destroy()* function destroys the mutex object referenced by *mutex*; the mutex object becomes, in effect, uninitialised. An implementation may cause *pthread_mutex_destroy()* to set the object referenced by *mutex* to an invalid value. A destroyed mutex object can be re-initialised using *pthread_mutex_init()*; the results of otherwise referencing the object after it has been destroyed are undefined.

It is safe to destroy an initialised mutex that is unlocked. Attempting to destroy a locked mutex results in undefined behaviour.

In cases where default mutex attributes are appropriate, the macro PTHREAD_MUTEX_INITIALIZER can be used to initialise mutexes that are statically allocated. The effect is equivalent to dynamic initialisation by a call to *pthread_mutex_init()* with parameter *attr* specified as NULL, except that no error checks are performed.

RETURN VALUE

If successful, the *pthread_mutex_init()* and *pthread_mutex_destroy()* functions return zero. Otherwise, an error number is returned to indicate the error. The [EBUSY] and [EINVAL] error checks, if implemented, act as if they were performed immediately at the beginning of processing for the function and cause an error return prior to modifying the state of the mutex specified by *mutex*.

Processus et threads (Implémentation)

- Synchronisation et exclusion mutuelle des processus (thread)
 - MUTEX (***pthread_mutex_lock()***, ***pthread_mutex_unlock()***)

```
#include <pthread.h>
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex);  
int pthread_mutex_trylock(pthread_mutex_t *mutex);  
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

DESCRIPTION

The mutex object referenced by *mutex* is locked by calling *pthread_mutex_lock()*. If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object referenced by *mutex* in the locked state with the calling thread as its owner.

If the mutex type is `PTHREAD_MUTEX_NORMAL`, deadlock detection is not provided. Attempting to relock the mutex causes deadlock. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, undefined behaviour results.

If the mutex type is `PTHREAD_MUTEX_ERRORCHECK`, then error checking is provided. If a thread attempts to relock a mutex that it has already locked, an error will be returned. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error will be returned.

If the mutex type is `PTHREAD_MUTEX_RECURSIVE`, then the mutex maintains the concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock count is set to one. Every time a thread relocks this mutex, the lock count is incremented by one. Each time the thread unlocks the mutex, the lock count is decremented by one. When the lock count reaches zero, the mutex becomes available for other threads to acquire. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error will be returned.

If the mutex type is `PTHREAD_MUTEX_DEFAULT`, attempting to recursively lock the mutex results in undefined behaviour. Attempting to unlock the mutex if it was not locked by the calling thread results in undefined behaviour. Attempting to unlock the mutex if it is not locked results in undefined behaviour.

The function *pthread_mutex_trylock()* is identical to *pthread_mutex_lock()* except that if the mutex object referenced by *mutex* is currently locked (by any thread, including the current thread), the call returns immediately.

Processus et threads (Implémentation)

- Synchronisation et exclusion mutuelle des processus (thread)
 - MUTEX (***pthread_mutex_lock()***, ***pthread_mutex_unlock()***)

The *pthread_mutex_unlock()* function releases the mutex object referenced by *mutex*. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by *mutex* when *pthread_mutex_unlock()* is called, resulting in the mutex becoming available, the scheduling policy is used to determine which thread shall acquire the mutex. (In the case of PTHREAD_MUTEX_RECURSIVE mutexes, the mutex becomes available when the count reaches zero and the calling thread no longer has any locks on this mutex).

If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the thread resumes waiting for the mutex as if it was not interrupted.

RETURN VALUE

If successful, the *pthread_mutex_lock()* and *pthread_mutex_unlock()* functions return zero. Otherwise, an error number is returned to indicate the error.

The function *pthread_mutex_trylock()* returns zero if a lock on the mutex object referenced by *mutex* is acquired. Otherwise, an error number is returned to indicate the error.

Processus et threads (Implémentation)

- Synchronisation et exclusion mutuelle des processus
 - MUTEX: utilisation (fonctions: *pthread_mutex_lock()* et *pthread_mutex_unlock()*)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *functionC();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int counter = 0;

main()
{
    int rc1, rc2;
    pthread_t thread1, thread2;

    /* Create independent threads each of which will execute functionC */

    if( (rc1=pthread_create( &thread1, NULL, &functionC, NULL)) )
    {
        printf("Thread creation failed: %d\n", rc1);
    }

    if( (rc2=pthread_create( &thread2, NULL, &functionC, NULL)) )
    {
        printf("Thread creation failed: %d\n", rc2);
    }

    /* Wait till threads are complete before main continues. Unless we
    /* wait we run the risk of executing an exit which will terminate
    /* the process and all threads before the threads have completed. */

    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);

    exit(0);
}

void *functionC()
{
    pthread_mutex_lock( &mutex1 );
    counter++;
    printf("Counter value: %d\n",counter);
    pthread_mutex_unlock( &mutex1 );
}
```


Processus et threads (Implémentation)

- Synchronisation et exclusion mutuelle des processus
 - MUTEX

```
#include <stdio.h>
#include <pthread.h>

#define NTHREADS 10
void *thread_function(void *);
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int counter = 0;

main()
{
    pthread_t thread_id[NTHREADS];
    int i, j;

    for(i=0; i < NTHREADS; i++)
    {
        pthread_create( &thread_id[i], NULL, thread_function, NULL );
    }

    for(j=0; j < NTHREADS; j++)
    {
        pthread_join( thread_id[j], NULL);
    }

    /* Now that all threads are complete I can print the final result.      */
    /* Without the join I could be printing a value before all the threads */
    /* have been completed.                                                */

    printf("Final counter value: %d\n", counter);
}

void *thread_function(void *dummyPtr)
{
    printf("Thread number %ld\n", pthread_self());
    pthread_mutex_lock( &mutex1 );
    counter++;
    pthread_mutex_unlock( &mutex1 );
}
```

Retourne le # du thread
appellant

Processus et threads (Implémentation)

- Synchronisation et exclusion mutuelle des processus

– MUTEX (essentiel)

```
void *function1()
{
    ...
    pthread_mutex_lock(&lock1);           - Execution step 1
    pthread_mutex_lock(&lock2);           - Execution step 3 DEADLOCK!!!
    ...
    ...
    pthread_mutex_lock(&lock2);
    pthread_mutex_lock(&lock1);
    ...
}

void *function2()
{
    ...
    pthread_mutex_lock(&lock2);           - Execution step 2
    pthread_mutex_lock(&lock1);
    ...
    ...
    pthread_mutex_lock(&lock1);
    pthread_mutex_lock(&lock2);
    ...
}

main()
{
    ...
    pthread_create(&thread1, NULL, function1, NULL);
    pthread_create(&thread2, NULL, function1, NULL); // function2
    ...
}
```

- Il faut donc faire attention à l'ordre d'application des MUTEX