

Synchronisation

INF3173 – Principes des systèmes d'exploitation
Automne 2024

Francis Giraldeau
giraldeau.francis@uqam.ca

Université du Québec à Montréal



Agenda

- Introduction
- Synchronisation en espace utilisateur
- Interblocage
- Synchronisation espace noyau
- Études de cas, exemples, exercices

Introduction

- Les opérations sur une donnée partagée par plusieurs processeurs peut conduire à une condition critique (résultat imprévisible, aléatoire)
- L'état final dépend de l'ordre spécifique des opérations
- Mécanisme de synchronisation permet d'empêcher les conditions critiques
- But : assurer l'accès exclusif (exclusion mutuelle)

Sections critiques (1)

- Soit deux fils d'exécution sur deux processeurs opérant sur la même variable
- Valeur de départ 1, CPU 0 soustrait 1, CPU 1 ajoute 1, résultat attendu 1

	CPU 0		CPU 1		
Temps	Instruction	Registre	Instruction	Registre	Mémoire
0					1
1	load	1			1
2		1	load	1	1
3	sub 1	0		1	1
4		0	add 1	2	1
5	store	0		2	0
6			store	2	2

Opération entrelacées
Résultats possibles:
{ 0, 1, 2 }

	CPU 0		CPU 1		
Temps	Instruction	Registre	Instruction	Registre	Mémoire
0					1
1	load	1			1
2	sub 1	0			1
3	store	0			0
4			load	0	0
5			add 1	1	0
6			store	1	1

Opération groupées
Résultats possibles:
{ 1 }

Sections critiques (2)

- Il faut empêcher l'utilisation simultanée de la variable commune *solde*. Lorsqu'un thread (ou un processus) exécute la séquence d'instructions (1 et 2), l'autre doit attendre jusqu'à ce que le premier ait terminé cette séquence.
- Section critique : suite d'instructions qui opèrent sur un ou plusieurs objets partagés et qui nécessitent une utilisation exclusive des objets partagés.
- Chaque thread (ou processus) a ses propres sections critiques.
- Les sections critiques des différents processus ou threads qui opèrent sur des objets communs doivent s'exécuter en exclusion mutuelle.
- Avant d'entamer l'exécution d'une de ses sections critiques, un thread (ou un processus) doit s'assurer de l'utilisation exclusive des objets partagés manipulés par la section critique

Assurer l'exclusion mutuelle?

- Encadrer chaque section critique par des opérations spéciales qui visent à assurer l'utilisation exclusive des objets partagés.
- Si P1 est dans sa section critique et P2 désire entrer dans sa section critique, alors P2 attend que P1 quitte sa section critique

P1

```
/* début du block protégé */  
critical_section_enter(&lock);  
tmp = x;  
tmp++;  
x = tmp;  
/* fin du block protégé */  
critical_section_leave(&lock);
```

P2

```
/* début du block protégé */  
critical_section_enter(&lock);  
tmp = x;  
tmp--;  
x = tmp;  
/* fin du block protégé */  
critical_section_leave(&lock);
```

Les 4 conditions pour l'exclusion mutuelle

- Deux processus ne peuvent être en même temps dans leurs sections critiques.
- Aucune hypothèse ne doit être faite sur les vitesses relatives des processus et sur le nombre de processeurs.
- Aucun processus suspendu en dehors de sa section critique ne doit bloquer les autres processus.
- Aucun processus ne doit attendre trop longtemps avant d'entrer en section critique (attente bornée).

Spinlock \rightarrow variable $\xrightarrow{\text{Soit}}$ 0, 1
 \hookrightarrow cmpxchg

Semaphor \rightarrow il pourrait aller plus que 10

Supposons qu'il existe une condition critique sur un entier en mémoire, dont la valeur de départ est 10. Deux fils d'exécution tentent d'incrémenter la variable pratiquement en même temps. Sélectionner toutes les valeurs finales possibles.

Waiting for next clap

\rightarrow pas de synchron.

8

9

10

11 ✓

12 ✓

13

En Java, une méthode marquée synchronized est-elle protégée par un verrou récursif?

Chaque obj au java à un verrou

Oui

Non

Associer le type de synchronisation avec sa description

Accès exclusif ✓

B. Mutex

Contient un compteur

C. Spinlock

Semaphore

Scrutation

Select choice

Spinlock

Point de rendez-vous

D. Sémaphore

barrier

Signaler un changement d'état ✓

E. Variable de condition

Que se passe-t-il quand un fil d'exécution tente d'obtenir un mutex alors qu'il est déjà occupé?

Results

You answered:

Une condition critique se produit

La tentative d'accès génère le signal SIGSEGV

Le fil bloque et est réveillé lorsque le verrou est libéré ✓

Le verrou est volé à l'autre fil d'exécution

C'est pas possible

The correct answer was

Le fil bloque et est réveillé lorsque le verrou est libéré

Supposons qu'un fil d'exécution démarre 4 fils d'exécution avec `pthread_create()`. On veut utiliser une barrière dans le fil principal pour s'assurer qu'ils sont tous démarrés. À combien devrait-on initialiser la barrière?

N'importe quel nombre pair fonctionne

4

5 ✓ $\leftarrow f; i/e + 1$

$4 * 2 + 1$

Techniques pour l'exclusion mutuelle

- Support matériel
 - Désactivation des interruptions
 - Instructions atomiques
 - Verrou actif
- Algorithmique
 - Algorithme de Dekkers
 - Algorithme de Peterson
- Support du système d'exploitation
 - Attente passive
 - Communication inter-processus
 - Barrière, Semaphore et Mutex

Masquage des interruptions

```
/* linux/kernel/sched.c */
static int migration_cpu_stop(void *data)
{
    struct migration_arg *arg = data;
    local_irq_disable();
    __migrate_task(...);
    local_irq_enable();
    return 0;
}
```

Arrêt d'un processeur

Prévenir les interruptions
pendant la migration de la tâche

```
/* linux/arch/x86/include/asm/irqflags.h */
static inline void native_irq_disable(void)
{
    asm volatile("cli": : : "memory");
}

static inline void native_irq_enable(void)
{
    asm volatile("sti": : : "memory");
}
```

CLI - Clear Interrupt Flag

STI - Set Interrupt Flag

Problèmes liés au masquage des interruptions

- La désactivation est généralement réservée au système d'exploitation. Un processus en espace utilisateur ne peut pas désactiver les interruptions, sinon il pourrait empêcher l'ordonnanceur de s'exécuter en bloquant les interruptions de l'horloge.
- Si les interruptions ne sont pas réactivées, alors le système ne réagirait plus aux commandes et ne fonctionnerait plus normalement.
- Elle n'assure pas l'exclusion mutuelle, si le système n'est pas monoprocesseur (le masquage des interruptions concerne uniquement le processeur qui a demandé l'interdiction). Les autres processus exécutés par un autre processeur pourront donc accéder aux objets partagés.

Attente active avec alternance

- Utiliser une variable tour qui mémorise le tour du processus qui doit entrer en section critique.
- tour est initialisée à 0.

*→ c'est rapide
mais on gaspie
tmp de proces*

```
Processus P0
while (1)
{   /*attente active*/
    while (tour !=0) ;
    section_critique_P0() ;
    tour = 1 ;

    ....
}
```

```
Processus P1
while (1)
{   /*attente active*/
    while (tour !=1) ;
    section_critique_P1() ;
    tour = 0 ;

    ....
}
```

Attente active avec alternance: problèmes

- Un processus peut être bloqué par un processus qui n'est pas en section critique.
 - P0 lit la valeur de tour qui vaut 0 et entre dans sa section critique. Il est suspendu et P1 est exécuté.
 - P1 teste la valeur de tour qui est toujours égale à 0. Il entre donc dans une boucle en attendant que tour prenne la valeur 1. Il est suspendu et P0 est élu de nouveau.
 - P0 quitte sa section critique, met tour à 1 et entame sa section non critique. Il est suspendu et P1 est exécuté.
 - P1 exécute rapidement sa section critique, **tour** = 0 et sa section non critique. Il teste tour qui vaut 0. Il attend que **tour** prenne la valeur 1.
- Attente active consomme du temps CPU.

Solution de Peterson

- Cette solution se base sur deux fonctions `entrer_region` et `quitter_region`.
- Chaque processus doit, avant d'entrer dans sa section critique appeler la fonction `entrer_region` en lui fournissant en paramètre son numéro de processus.
- Cet appel le fera attendre si nécessaire jusqu'à ce qu'il n'y ait plus de risque.
- A la fin de la section critique, il doit appeler `quitter_region` pour indiquer qu'il quitte sa section critique et pour autoriser l'accès aux autres processus.

Solution de Peterson (2)

```
void entrer_region(int process)
{
    int autre;
    autre = 1 - process; //l'autre processus
    interesse[process] = TRUE; //indiquer qu'on est intéressé
    tour = process; //la course pour entrer se gagne ici
    while (tour == process && interesse[autre] == TRUE);
}
    attente active (busy waiting)
```




Si un autre processus appelle également entrer_region et modifie tour pour qu'il pointe vers lui-même (tour = autre), alors la condition `tour == process` deviendra false pour le processus actuel.

```
void quitter_region (int process )
{
    interesse[process] = FALSE;
}
```

Problème : attente active = consommation du temps CPU

Instructions atomiques

↳ indivisible

- Instruction exécutant plusieurs opérations indivisible
 - Fetch-And-Add  Cette instruction effectue deux opérations atomiques en une seule :
1) Lit la valeur d'une variable. 2) Ajoute une valeur à cette variable.
 - Test-And-Set  Lit la valeur d'une variable et la définit à une nouvelle valeur (souvent true), en 1 seule opération atomique. Utilisation classique : verrouillage binaire (spinlock).
 - Compare-And-Swap  Vérifie si une variable contient une valeur attendue, et si oui, remplace cette valeur par une nouvelle.
- Permet d'implémenter des primitives de synchronisation efficacement

Exemple instruction atomique cmpxchg

lock cmpxchg reg/mem, reg

Réservation
explicite du bus

Nom de
l'instruction

Opérande de
destination

Opérande
source

```
lock cmpxchg [rdi], rcx ; compare la valeur [rdi] avec rcx
                        ; si ([rdi] == rcx)
                        ;   ZF = 1, [rdi] = rcx
                        ; sinon
                        ;   ZF = 0, rax = [rdi]
```

Le bus est verrouillé
entre la comparaison et
l'échange, ce qui garanti
le résultat

Implémentation d'un verrou actif (spin_lock)

```
/* début du block protégé */  
spin_lock(&lock);  
x++;  
/* fin du block protégé */  
spin_unlock(&lock);
```

La boucle se répète tant
que le verrou n'est pas obtenu.

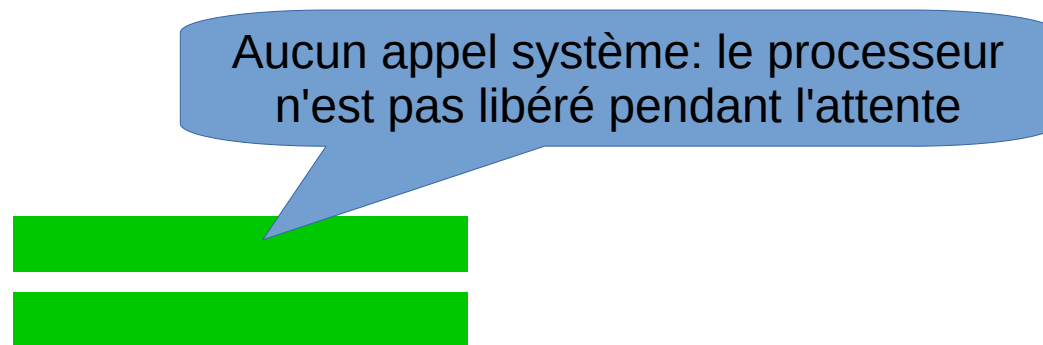
```
spin_lock:                                ; adresse du verrou dans rdi  
    mov rcx, 1                            ; valeur de comparaison  
spin_lock_retry:                          ; remettre a zero rax  
    xor rax, rax                          ; compare la valeur [rdi] avec rcx  
    lock mcpxchg [rdi], rcx               ; si ([rdi] == rax)  
                                           ; ZF = 1, [rdi] = rcx  
                                           ; sinon  
                                           ; ZF = 0, rax = [rdi]  
    jnz spin_lock_retry                  ; on utilise verrou  
    ret                                  dans sec. Critique  
  
spin_unlock:                              ; adresse du verrou dans rdi  
    mov qword [rdi], 0                   ; remettre le verrou à 0  
    ret
```

jump not zero

→ rax = 0

Synchronisation: attente active

- spinlock
 - Rapide: une instruction atomique (ex: cmpxchg)
 - Réservé pour une attente courte et probabilité de contention faible
 - Fréquent dans le noyau (ex: interruption)



Exclusion mutuelle avec SÉ

- Remplacer l'attente active par l'attente passive

*état,
bloqué*

1. • SLEEP() est un appel système qui suspend l'appelant en attendant qu'un autre le réveille.
2. • WAKEUP(process) : est un appel système qui réveille le processus process.

Synchronisation: attente passive

- Mutex implémenté avec **futex()**
 - Rapide si aucune contention: aucun appel système pour prendre le verrou (ex: cmpxchg)
 - Si le verrou est déjà utilisé:
 - futex(FUTEX_WAIT): attente passive (schedule)
 - futex(FUTEX_WAKE): verrou libéré

il met en bloque la tâche

surcoût d'appel sys

surcoût d'appel sys.

il met en bloque la tâche



Verrou occupé: futex wait

verrou active → tous est vert

Mutual Ex

Défis de l'exclusion mutuelle

- Interblocage (*deadlock*)
 - Un cycle d'attente empêche le système d'évoluer.
- Famine (*starvation*)
 - Une tâche n'arrive jamais à s'exécuter parce qu'elle n'est jamais choisie par rapport à une autre tâche, malgré qu'elle soit prête.
- Interblocage actif (*livelock*)
 - Une situation dans laquelle au moins deux processus s'exécutent et dont l'état change sans cesse sans que le travail progresse.
- Inversion des priorités
 - Une tâche moins prioritaire détient un verrou qu'une tâche plus prioritaire a besoin pour s'exécuter.
- Surcoût
 - Quel est l'impact de performance du choix de synchronisation?

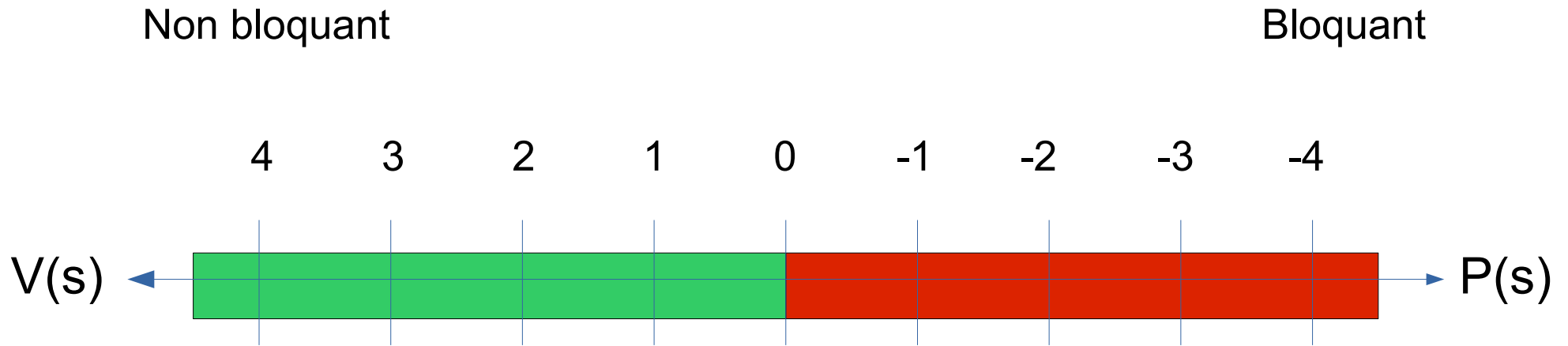
Sémaphores (1)

- Pour contrôler les accès à un objet partagé, E. W. Dijkstra (1965) suggéra l'emploi d'un nouveau type de variables appelées sémaphores.
- Un sémaphore est un compteur entier qui désigne le nombre d'autorisations d'accès disponibles.
- Chaque sémaphore a un nom et une valeur initiale.
- Les sémaphores sont manipulés au moyen des opérations :
 - P() désigné aussi par down ou wait (Proberen)
 - V() désigné aussi par up ou signal (Verhogen)

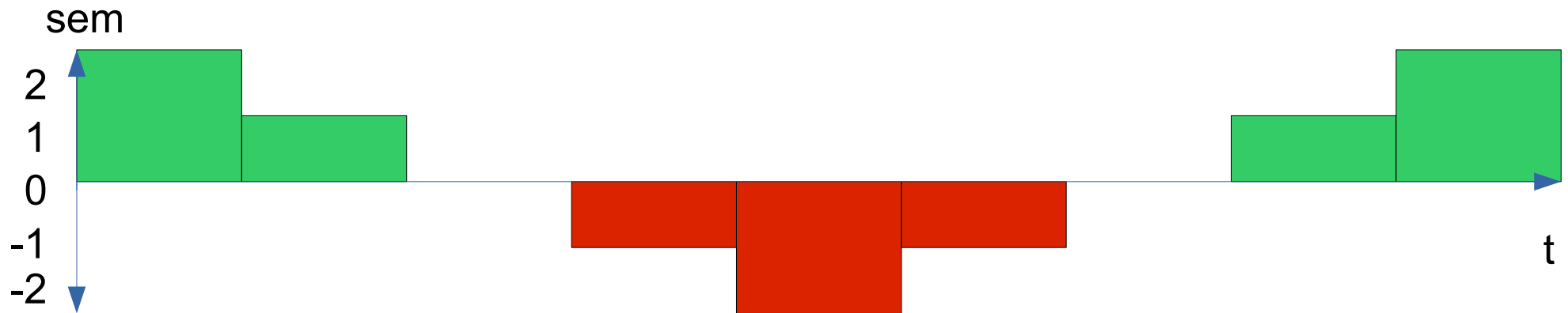
Sémaphores (2)

- L'opération $P(S)$ décrémente la valeur du sémaphore S si cette dernière est supérieure à 0. Sinon le processus appelant est mis en attente du sémaphore.
- L'opération $V(S)$ incrémente la valeur du sémaphore S , si aucun processus n'est bloqué par l'opération $P(S)$. Sinon, l'un d'entre-eux sera choisi et redeviendra prêt.
- Chacune de ces deux opérations doit être implémentée comme une opération indivisible.
- Chaque sémaphore a une file d'attente.
 - File FIFO : sémaphore forte
 - File LIFO : sémaphore faible (famine possible)

Sémaphores (3)



Exemple d'exécution d'une sémaphore



Temps	Processus	Action	sem	Pile			Exec	
0			2					
1	A	P(s)	1				A	
2	B	P(s)	0				A	B
3	C	P(s)	-1	C			A	B
4	D	P(s)	-2	C	D		A	B
5	B	V(s)	-1	D			A	C
6	A	V(s)	0				C	D
7	D	V(s)	1				C	
8	C	V(s)	2					

Exclusion mutuelle au moyen de sémaphores

- Les sémaphores **binaires** permettent d'assurer l'exclusion mutuelle :

```
Semaphore sem = 1;
```

```
Processus P1 :  
{  
    P(sem)  
    Section_critique _de_P1() ;  
    V(sem) ;  
}
```

```
Processus P2 :  
{  
    P(sem)  
    Section_critique_de_P2();  
    V(sem) ;  
}
```

```
P1: P(mutex) → mutex=0  
P1 : entame Section_critique_de_P1();  
P2 : P(mutex) → P2 est bloqué → file de mutex  
P1: poursuit et termine Section_critique_de_P1();  
P1 : V(mutex)→ débloquent P2 → file des processus prêts  
P2: Section_critique_de_P2();  
P2: V(mutex) → mutex=1
```

Implémentation des sémaphores

```
typedef struct semafun {  
    int count;  
    int lock;  
    GQueue *queue;  
} semafun_t;
```

Utilisation d'un verrou actif
pour une courte durée

```
void mysemaphore_wait(sema_t *sem) {  
    spin_lock(&sem->lock);  
    sem->count--;  
    if (sem->count < 0) {  
        /* s'ajouter à la queue */  
        spin_unlock(&sem->lock);  
        /* se mettre en veille */  
    } else {  
        spin_unlock(&sem->lock);  
    }  
}
```

```
void mysemaphore_signal(sema_t *sem) {  
    spin_lock(&sem->lock);  
    sem->count++;  
    if (sem->count <= 0) {  
        /* pop queue */  
        spin_unlock(&sem->lock);  
        /* réveiller le processus */  
    } else {  
        spin_unlock(&sem->lock);  
    }  
}
```

Implémentation des sémaphores binaires

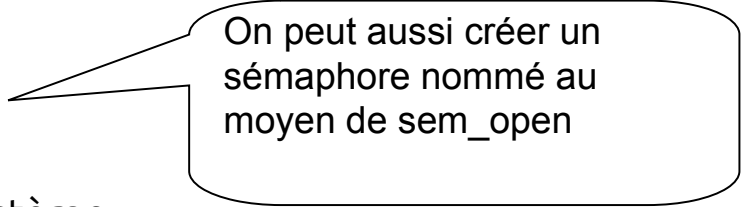
- Utilisation de Test-and-Set-Lock (TSL) et thread_yield()
- On dispose d'une librairie qui gère des fils d'exécution au niveau utilisateur et d'une fonction thread_yield() qui permet à un fil de libérer volontairement le processeur. Le fil est alors déplacé à la fin de la liste des fils prêts, et un autre fil est exécuté.

```
struct mutex_t {  
    int t;  
    // t=0 si non verrouillé  
}  
  
mutex_lock(mutex_t *m) {  
    while (TSL(m->t) != 0) {  
        // verrou occupé  
        // céder le processeur  
        thread_yield();  
    }  
}
```

```
mutex_unlock (mutex_t *m) {  
    m->t = 0;  
}
```

Sémaphores POSIX (1)

- Les sémaphores POSIX sont implantés dans la librairie <semaphore.h>
- Le type sémaphore est désigné par le mot : sem_t.
- L'initialisation d'un sémaphore est réalisée par l'appel système :



On peut aussi créer un sémaphore nommé au moyen de sem_open

int sem_init (sem_t*sp, int pshared, unsigned int count) ;

où

- sp est un pointeur sur le sémaphore à initialiser
 - count est la valeur initiale du sémaphore
 - pshared indique si le sémaphore est local au processus ou non (0 pour local et non null pour partagé).
- La suppression d'un sémaphore :
int sem_destroy(sem_t * sem);

Sémaphores POSIX (2)

- `int sem_wait(sem_t *sp)` : est l'opération P.
- `int sem_post(sem_t *sp)` : est l'opération V.
- `int sem_trywait(sem_t *sp)` : décrémente la valeur du sémaphore `sp`, si sa valeur est supérieure à 0 ; sinon elle retourne une erreur (`sem_wait` non bloquant).
- `int sem_getvalue(sem_t *sem, int *sval)` : retourne dans `sval` la valeur courante du sémaphore.

Exemple de sémaphore pour l'exclusion mutuelle

```
static uint64_t a = 0; // variable globale
int main(int argc, char **argv) {
    int p, i;
    pthread_t t1, t2;
    init_count_sem();
    pthread_create(&t1, NULL, count_passive, &a);
    pthread_create(&t2, NULL, count_passive, &a);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("pid=%d a=%" PRIu64 "\n", getpid(), a);
    return EXIT_SUCCESS;
}
```

Deux fils accèdent
en même temps à la
variable globale « a »

```
void *count_passive(void *arg) {
    volatile uint64_t *var = (uint64_t *) arg;
    volatile uint64_t i, j;
    sem_wait(&sem_lock);
    for (i = 0; i < MAX; i++) {
        *var = *var + 1;
    }
    sem_post(&sem_lock);
    return NULL;
}
```

```
static sem_t sem_lock;
void init_count_sem() {
    sem_init(&sem_lock, 0, 1);
}
```

La sémaphore empêche
deux fils sérialiser
l'exécution des boucles

Différences entre sémaphore et mutex

Sémaphore

- Compteur
- Opérations $P()$, $V()$ par n'importe quel processus
- Synchronisation plus générale

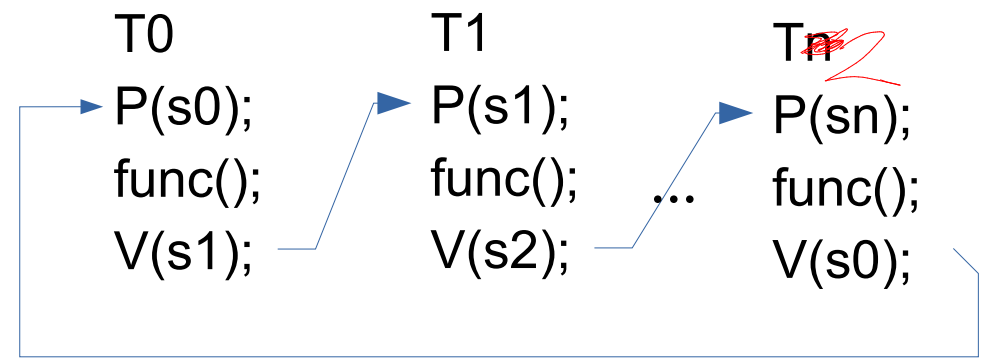
Mutex

- Binaire
- Spécifique à l'exclusion mutuelle

Exemple de course à relais

$S_0 = 1$ $S_1 = 0$ $S_2 = 0$

```
// chain.c
void *do_run(void *arg) {
    int i;
    args_t *args = (args_t *) arg;
    for (i=0; i < 2; i++) {
        sem_wait(args->curr);
        printf("tour id=%d\n", args->id);
        hog();
        sem_post(args->next);
    }
}
```



Calcul à tour de rôle

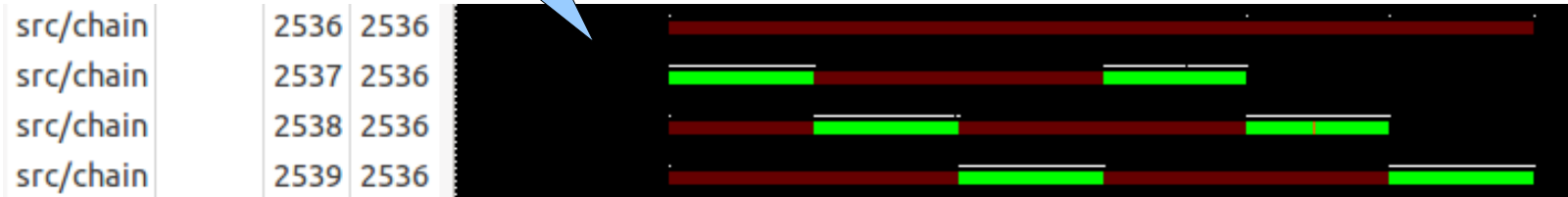
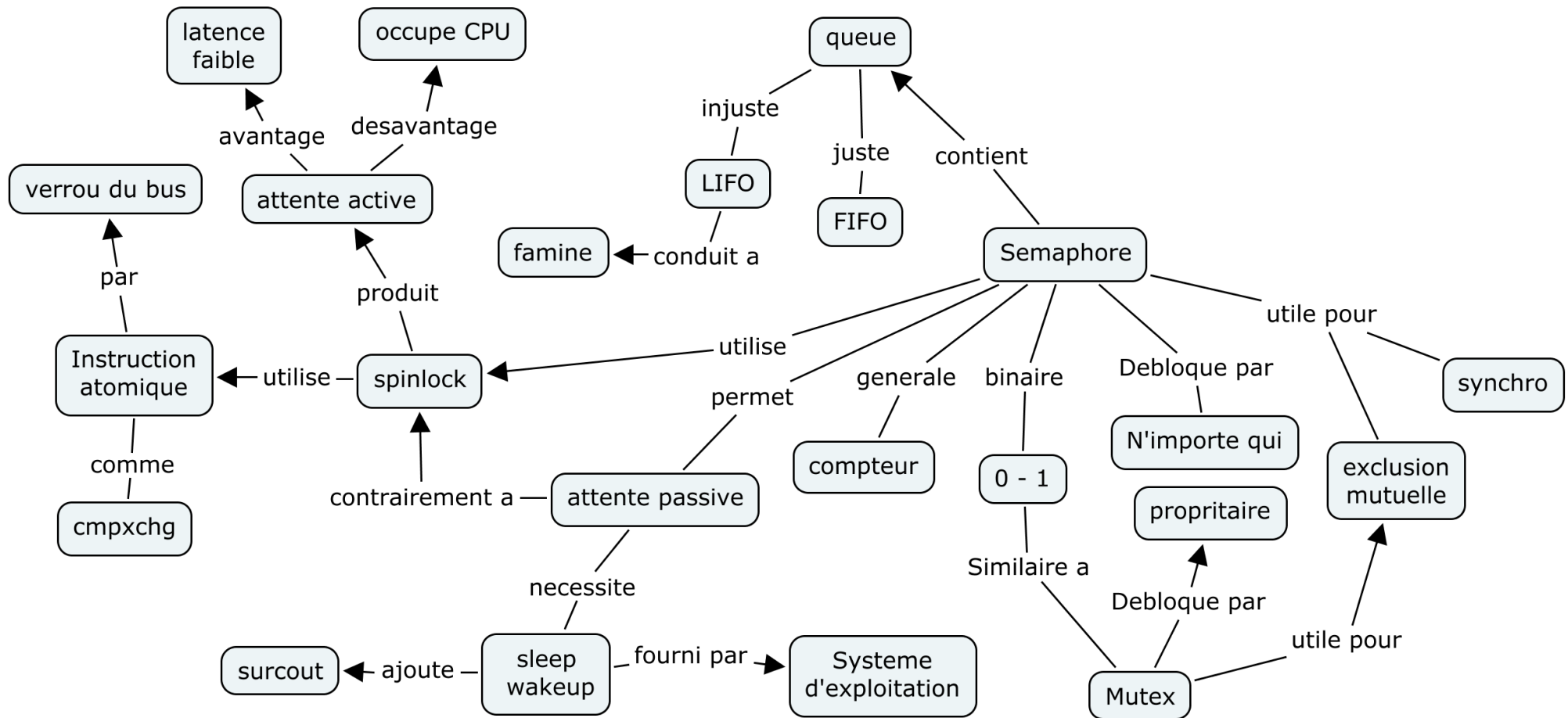


Schéma de concept



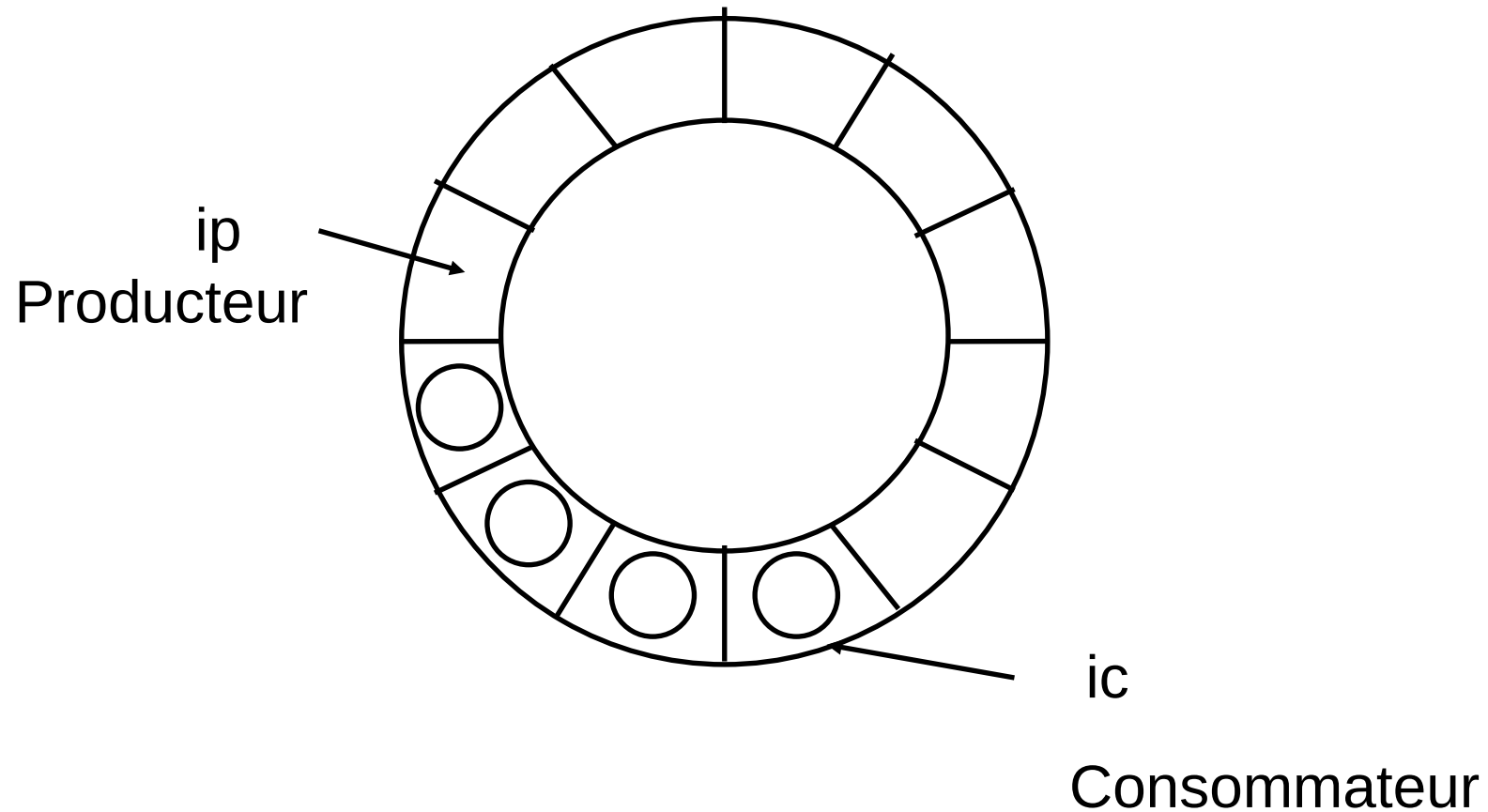
Problème du producteur/consommateur

- Deux processus partagent une mémoire tampon de taille fixe N. L'un d'entre eux, le producteur, dépose des informations dans le tampon, et l'autre, le consommateur, les retire.
- Le tampon est géré comme une file circulaire ayant deux pointeurs (un pour les dépôts et l'autre pour les retraits).

```
int ip = 0;                                     Producteur
while(1) {
    S'il y a, au moins, une entrée libre dans le tampon
    alors produire(tampon, ip); ip = modulo(ip+1, N);
    sinon attendre jusqu'à ce qu'une entrée se libère.
}
```

```
int ic = 0;                                     Consommateur
while(1) {
    S'il y a, au moins, une entrée occupée dans le tampon
    alors consommer(tampon, ic); ic = modulo(ic+1, N);
    sinon attendre jusqu'à ce qu'une entrée devienne occupée
}
```

Problème du producteur/consommateur (2)



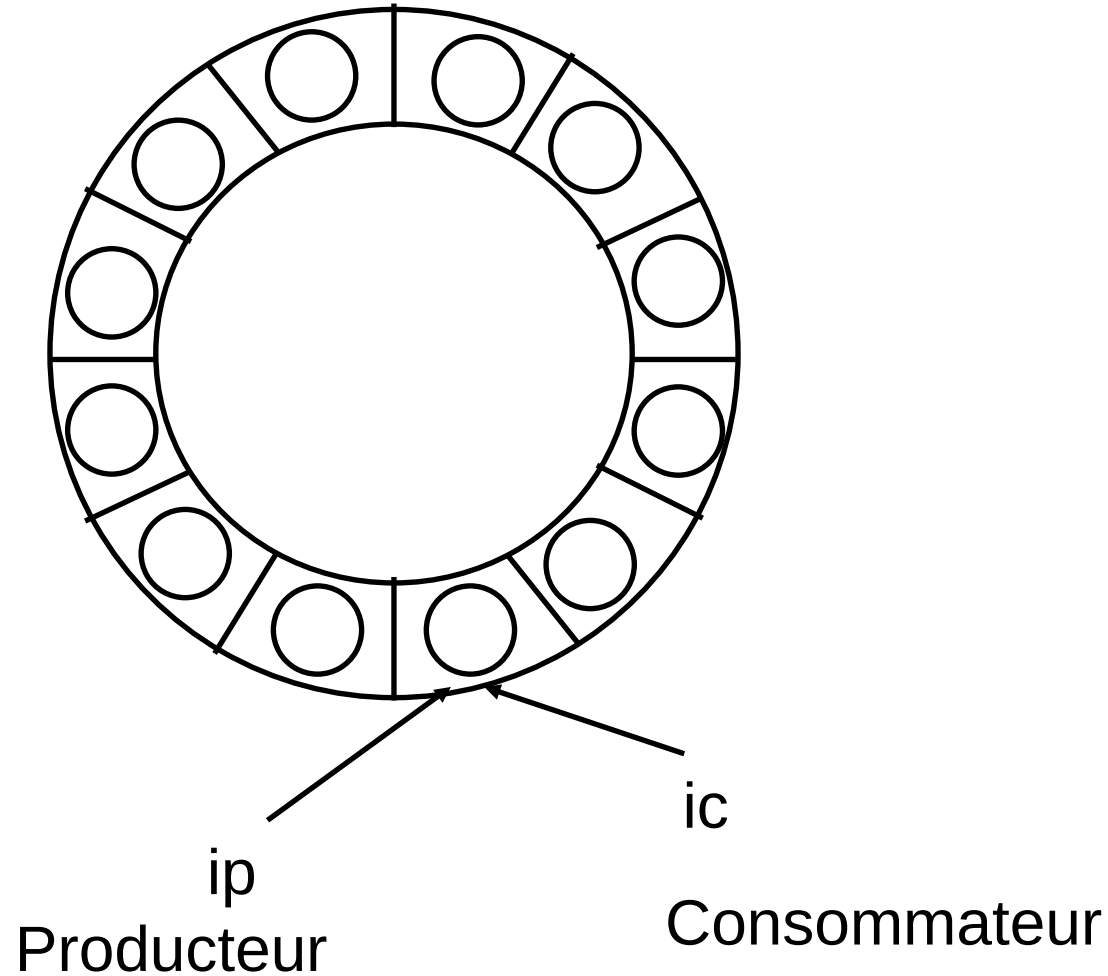
Problème du producteur/consommateur (3)



tampon plein

occupe = N;

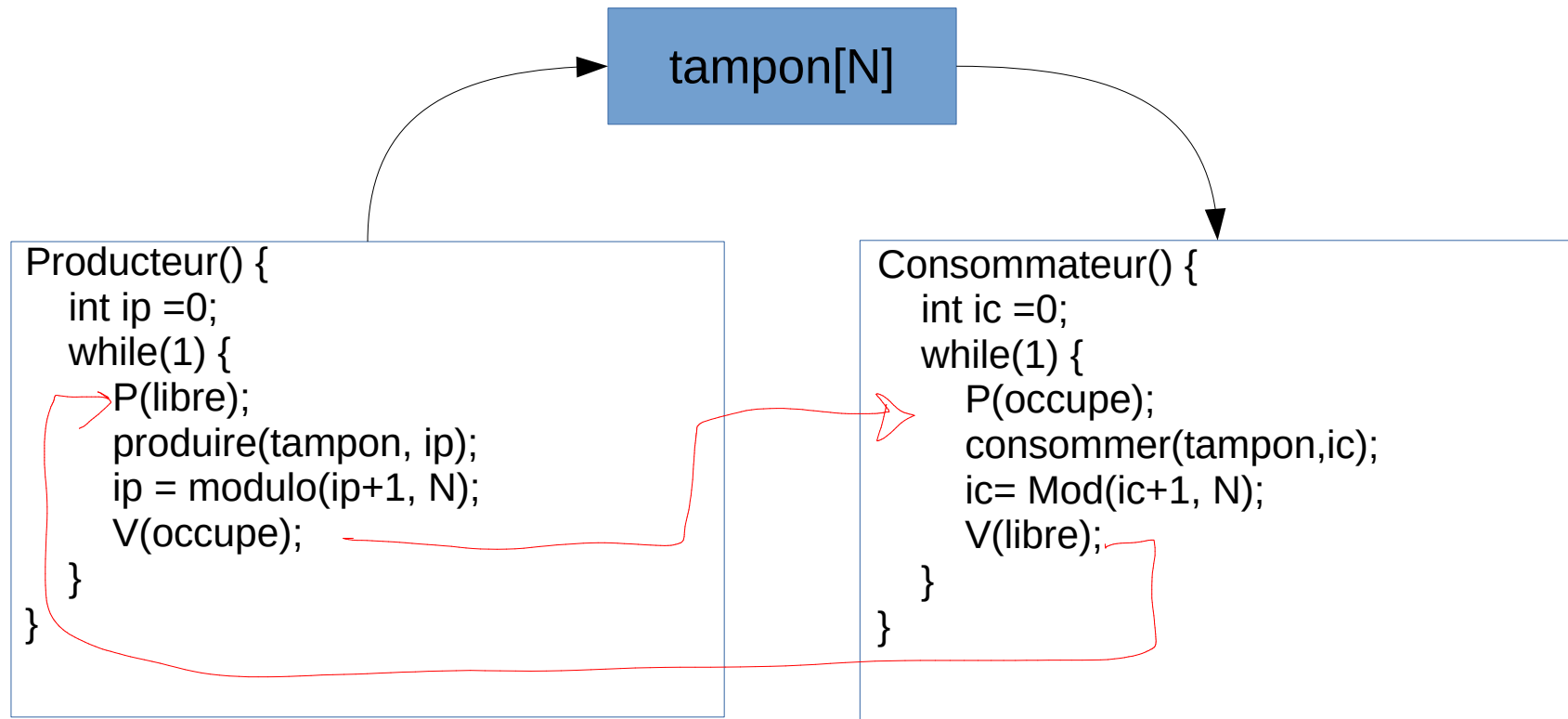
libre = 0



Problème du producteur/consommateur (4)

- La solution du problème au moyen des sémaphores nécessite deux sémaphores.
- Le premier, nommé *occupe*, compte le nombre d'emplacements occupés. Il est initialisé à 0.
- Il sert à bloquer/débloquer le consommateur ($P(\text{occupe})$ et $V(\text{occupe})$).
- Le second, nommé *libre*, compte le nombre d'emplacements libres. Il est initialisé à N (N étant la taille du tampon).
- Il sert à bloquer/débloquer le producteur ($P(\text{libre})$ et $V(\text{libre})$).

Problème du producteur/consommateur (5)



Problème du producteur/consommateur (6)

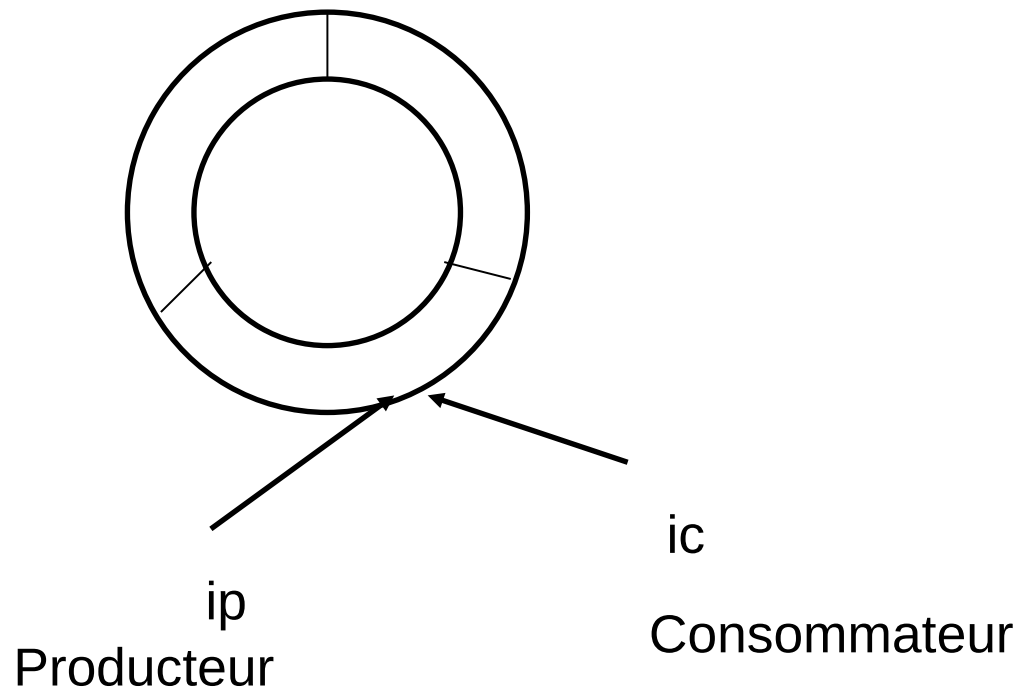
occupe = 0;

libre = 3



Consommateur bloqué

Producteur peut produire jusqu'à 3



Une production

Problème du producteur/consommateur (7)

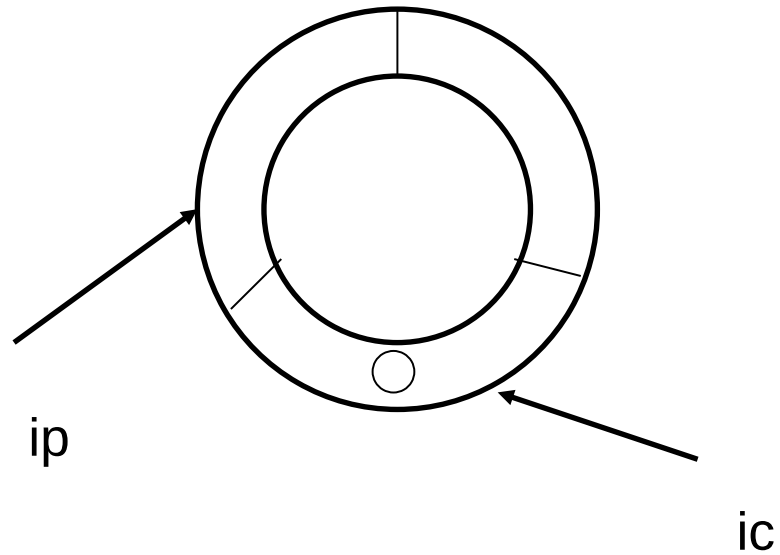
occupe = 1;

libre = 2



Consommateur peut consommer 1

Producteur peut produire 2



Une production

Problème du producteur/consommateur (8)

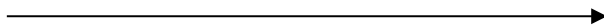
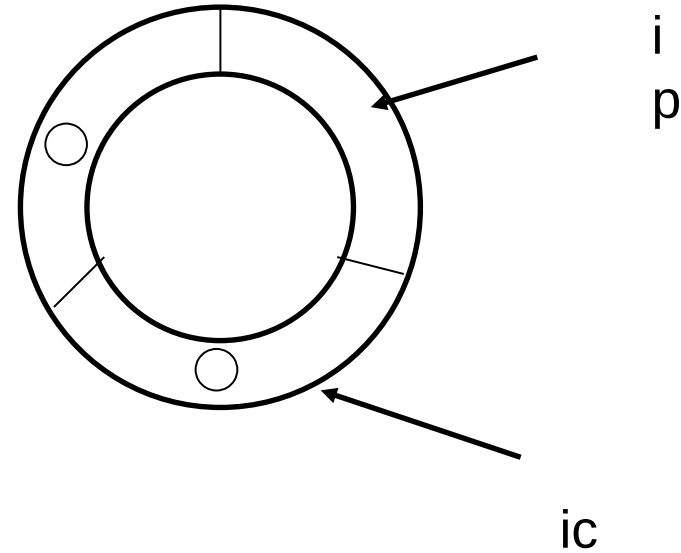
occupe = 2;

libre = 1



Consommateur peut consommer 2

Producteur peut produire 1



Une production

Problème du producteur/consommateur (9)

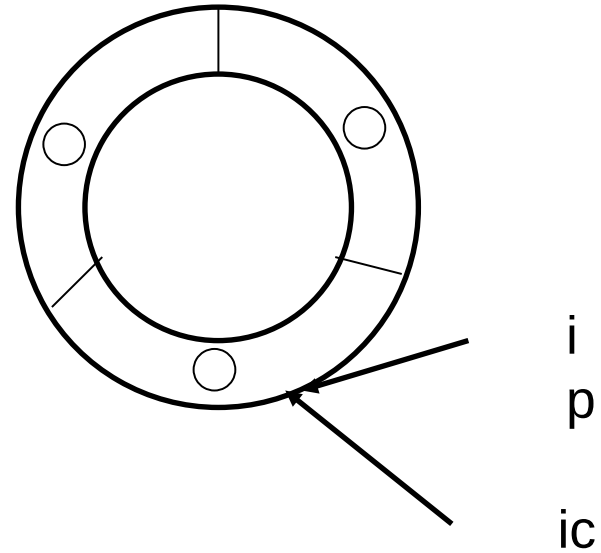
occupe = 3;

libre = 0



Consommateur peut
consommer 3

Producteur bloqué



Une consommation

Problème du producteur/consommateur (10)

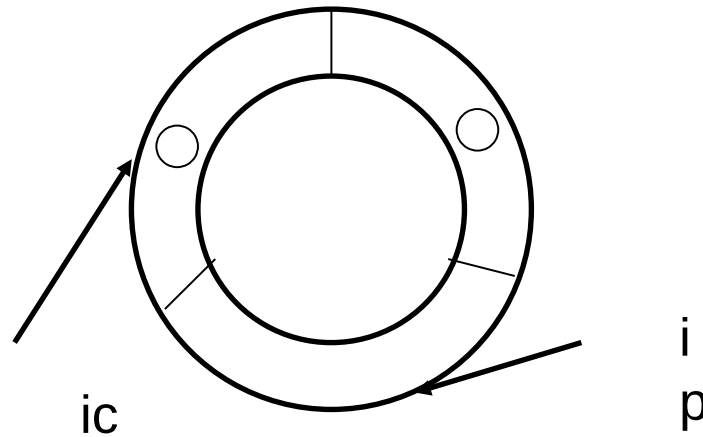
occupe = 2;

libre = 1



Consommateur peut consommer 2

Producteur peut produire 1



Une production

Problème du producteur/consommateur (11)

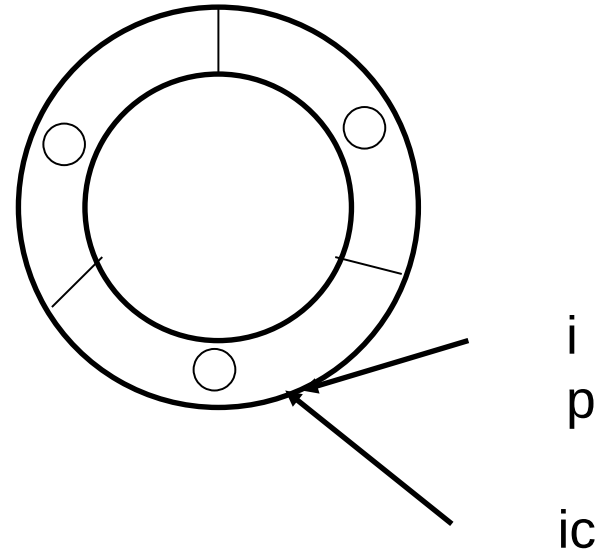
occupe = 3;

libre = 0



Consommateur peut
consommer 3

Producteur bloqué



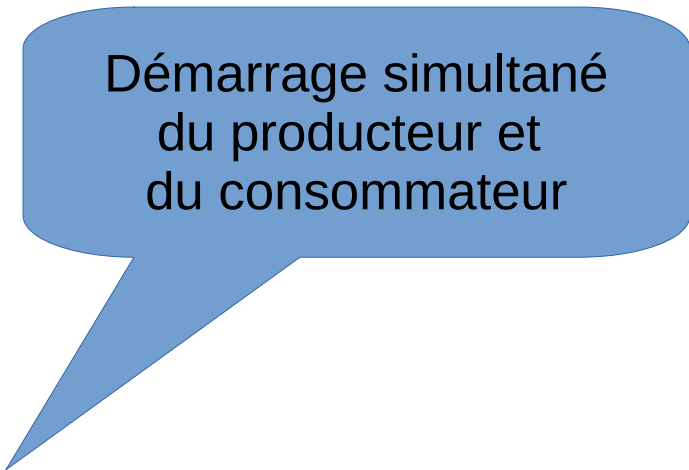
Problème du producteur/consommateur (12)

```
//programme prodcons1.c
#define MAX 6
#define BUF_SIZE 3

typedef struct args {
    sem_t sem_free;
    sem_t sem_busy;
} args_t;

static int buf[BUF_SIZE];

int main(int argc, char **argv) {
    int p, i;
    pthread_t t1, t2;
    args_t args;
    sem_init(&args.sem_free, 0, BUF_SIZE);
    sem_init(&args.sem_busy, 0, 0);
    pthread_create(&t1, NULL, prod, &args);
    pthread_create(&t2, NULL, cons, &args);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("exit\n");
    return EXIT_SUCCESS;
}
```



Démarrage simultané
du producteur et
du consommateur

Exemple du fil producteur

```
void *prod(void *arg) {  
    int ip = 0, nbprod = 0, obj = 0;  
    args_t *args = (args_t *) arg;  
    while(nbprod < MAX) {  
        sem_wait(&args->sem_free);  
        buf[ip] = obj;  
        sem_post(&args->sem_busy);  
        printf("prod: buf[%d]=%d\n", ip, obj);  
        obj++;  
        nbprod++;  
        ip = (ip + 1) % BUF_SIZE;  
    }  
    return NULL;  
}
```

Attendre un
espace libre

Indiquer qu'un
objet est disponible

Exemple du fil consommateur

```
void *cons(void *arg) {  
    int ic = 0, nbcons = 0, obj = 0;  
    args_t *args = (args_t *) arg;  
    while(nbcons < MAX) {  
        sleep(1);  
        sem_wait(&args->sem_busy);  
        obj = buf[ic];  
        sem_post(&args->sem_free);  
        printf("cons: buf[%d]=%d\n", ic, obj);  
        nbcons++;  
        ic = (ic + 1) % BUF_SIZE;  
    }  
    return NULL;  
}
```

Attendre qu'un objet
soit disponible

Indiquer qu'un
espace a été libéré

Exemple d'exécution de prodcons1 fine BUF_SIZE 3

```
$ ./src/prodcons2
prod: buf[0]=1001
prod: buf[1]=1002
prod: buf[2]=1003
cons: buf[0]=1001
prod: buf[0]=1004
cons: buf[1]=1002
prod: buf[1]=1005
cons: buf[2]=1003
prod: buf[2]=1006
cons: buf[0]=1004
cons: buf[1]=1005
cons: buf[2]=1006
exit
```

Le fil prod bloque quand le tampon est plein

Alternance entre prod et cons

Le fil cons traite tous les objets jusqu'à ce que le tampon soit vide.

Exercice 3 :

Généraliser à plusieurs producteurs et plusieurs consommateurs avec un seul tampon.

Problème n producteurs / m consommateurs

```
void *cons(void *arg) {  
    int nbcons = 0, obj = 0;  
    args_t *args = (args_t *) arg;  
    while(nbcons < MAX) {  
        sleep(1); // simuler un lecteur lent  
        sem_wait(&args->sem_busy);  
        sem_wait(&args->mutex);  
        obj = buf[args->ic];  
        printf("cons: buf[%d]=%d\n", args->ic, obj);  
        args->ic = (args->ic + 1) % BUF_SIZE;  
        sem_post(&args->mutex);  
        sem_post(&args->sem_free);  
        nbcons++;  
    }  
    return NULL;  
}
```

Variables ip et ic partagées,
besoin de mutex
lors de la modification

Problème des philosophes

- Cinq philosophes sont assis autour d'une table. Sur la table, il y a alternativement 5 plats de spaghettis et 5 fourchettes.
- Un philosophe passe son temps à manger et à penser.
- Pour manger son plat de spaghettis, un philosophe a besoin de deux fourchettes qui sont de part et d'autre de son plat.
- Si tous les philosophes prennent en même temps, chacun une fourchette, aucun d'entre eux ne pourra prendre l'autre fourchette (situation d'interblocage).
- Pour éviter cette situation, un philosophe ne prend jamais une seule fourchette.
- Les fourchettes sont les objets partagés. L'accès et l'utilisation d'une fourchette doivent se faire en exclusion mutuelle. On utilisera le sémaphore mutex pour réaliser l'exclusion mutuelle.

```
pthread_mutex_t lock;  
pthread_mutex_init(&lock, NULL);  
pthread_mutex_lock(&lock);
```

```
pthread_mutex_lock(&lock);
```

→ on peut pas par défaut
de reprendre un verrou

```
pthread_mutex_unlock(&lock);
```

mutex récursif \equiv java

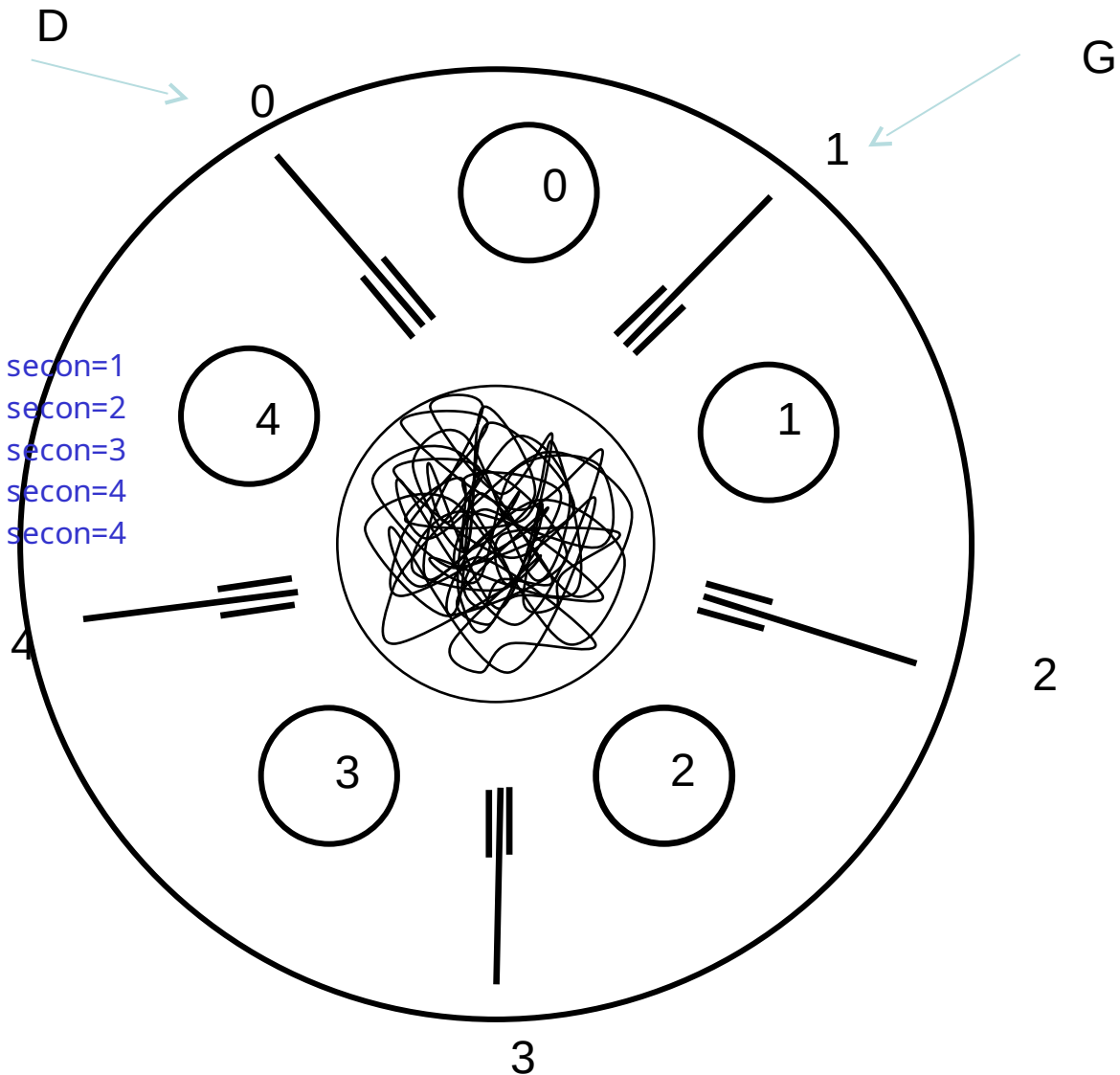
instruction
tous les apple sys : → est implémenté
par futex

Problème des philosophes (2)

philo \rightarrow threads

Baguette \rightarrow verrou

philo 0 gauche=0 droit=1 prem=0 secon=1
philo 1 gauche=1 droit=2 prem=1 secon=2
philo 2 gauche=2 droit=3 prem=2 secon=3
philo 3 gauche=3 droit=4 prem=3 secon=4
philo 4 gauche=4 droit=0 prem=0 secon=4



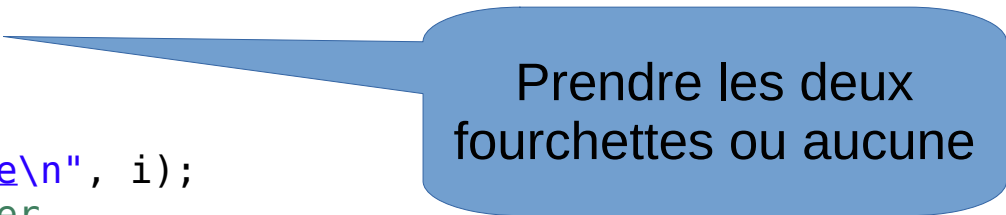
Implémentation du problème des philosophes

```
#define N 5           // nombre de philosophes
#define G (i+1)%N     // fourchette gauche du philosophe i
#define D i           // fourchette droite du philosophe i
#define libre 1
#define occupe 0
int fourch[N] = { libre, libre, libre, libre, libre };
sem_t mutex;

int main() {
    int i, NumPhi[N] = { 0, 1, 2, 3, 4 };
    pthread_t th[N];
    sem_init(&mutex, 0, 1);
    // création des N philosophes
    for (i = 0; i < N; i++)
        pthread_create(&th[i], NULL, philosophe, &NumPhi[i]);
    // attendre la fin des threads
    for (i = 0; i < N; i++)
        pthread_join(th[i], NULL);
    printf("fin des threads \n");
    return 0;
}
```

Implémentation du problème des philosophes

```
void * philosophe(void * num) {
    int i = *(int *) num, nb = 2;
    while (nb) {
        sleep(1);           // penser
        sem_wait(&mutex);    // essayer de prendre les fourchettes pour manger
        if (fourch[G] == libre && fourch[D] == libre) {
            fourch[G] = occupe;
            fourch[D] = occupe;
            sem_post(&mutex);
            nb--;
            printf("philosophe[%d] mange\n", i);
            sleep(1);        // manger
            printf("philosophe[%d] a fini de manger\n", i);
            sem_wait(&mutex); // libérer les fourchettes
            fourch[G] = libre;
            fourch[D] = libre;
            sem_post(&mutex);
        } else
            sem_post(&mutex);
    }
}
```



Prendre les deux
fourchettes ou aucune

Exécution de philosophe.c

```
$ ./src/philosophe
philosophe[4] mange
philosophe[2] mange
philosophe[4] a fini de manger
philosophe[2] a fini de manger
philosophe[1] mange
philosophe[4] mange
philosophe[1] a fini de manger
philosophe[4] a fini de manger
philosophe[3] mange
philosophe[1] mange
philosophe[3] a fini de manger
philosophe[1] a fini de manger
philosophe[0] mange
philosophe[3] mange
philosophe[0] a fini de manger
philosophe[3] a fini de manger
philosophe[2] mange
philosophe[0] mange
philosophe[2] a fini de manger
philosophe[0] a fini de manger
fin des threads
```

Famine des philosophes

- La solution précédente résout le problème d'interblocage. Mais, un philosophe peut mourir de faim, car il ne parvient jamais à obtenir les fourchettes nécessaires pour manger (problème de famine et d'équité).
- Pour éviter ce problème, il faut garantir que si un processus demande d'entrer en section critique, il obtient satisfaction au bout d'un temps fini.
- Dans le cas des philosophes, le problème de famine peut être évité, en ajoutant N sémaphores (un sémaphore pour chaque philosophe). Les sémaphores sont initialisés à 0.
- Lorsqu'un philosophe i ne parvient pas à prendre les fourchettes, il se met en attente ($P(S[i])$).
- Lorsqu'un philosophe termine de manger, il vérifie si ses voisins sont en attente. Si c'est le cas, il réveille les voisins qui peuvent manger en appelant l'opération V .
- On distingue trois états pour les philosophes : penser, manger et faim

Solution à la famine des philosophes

```
void * philosophe(void * num) {
    int i = *(int *) num, nb = 2;
    while (nb) {
        /* penser */
        sem_wait(&mutex);
        phiState[i] = THINKING;
        sem_post(&mutex);
        sleep(1);

        /* essayer de manger */
        sem_wait(&mutex);
        phiState[i] = HUNGRY;
        test(i);
        sem_post(&mutex);

        /* attendre son tour */
        sem_wait(&semPhil[i]);

        /* à nous de manger */
        printf("philosophe[%d] mange\n", i);
        sleep(1);
        printf("philosophe[%d] a fini\n", i);
    }
}
```

```
// vérifie si le philosophe i peut manger
void test(int i) {
    if ((phiState[i] == HUNGRY) &&
        (phiState[G(i)] != EATING) &&
        (phiState[D(i)] != EATING)) {
        phiState[i] = EATING;
        sem_post(&semPhil[i]);
    }
}
```

```
/* laisser manger ses voisins */
sem_wait(&mutex);
phiState[i] = THINKING;
test(G(i));
test(D(i));
sem_post(&mutex);
nb--;
}
```

