Synchronisation

Processus coopératifs

Définition : peut affecter ou être affecté par autres processus.

Pourquoi processus coopératifs ?

- partage de données
- accélération du calcul
- modularité
- commodité

Exécution concurrentielle et asynchrone $\Rightarrow$ besoin de communication et *synchronisation*

Exemple (1)

Deux processus partagent une variable `n` initialisée à 0.

PROCESSUS P_0

```
...
n = n + 2;
...
```

PROCESSUS P_1

```
...
n = n + 3;
...
```

On désire obtenir `n = 5`.

Exemple (2)

Traduction en langage machine

PROCESSUS P_0

```
...
LOAD A,n
ADD A,2
STORE A,n
...
```

PROCESSUS P_1

```
...
LOAD A,n
ADD A,3
STORE A,n
...
```

Une exécution possible :

t	P	instr	n
1	0	LOAD A,n	0
2	0	ADD A,2	0
3	1	LOAD A,n	0
4	1	ADD A,3	0
5	1	STORE A,n	3
6	0	STORE A,n	2

Résultats possibles : `n` = 2, 3 ou 5 !!!

Problème de la section critique

PROCESSUS $P_i, i = 0, \ldots, n-1$

while (1) {
  entrée dans la section critique
  section critique
  sortie de la section critique
  section restante
}

Dans la section critique un processus peut modifier des variables communes, écrire dans un fichier partagé, etc $\Rightarrow$ si un processus exécute sa section critique, aucun autre processus n'est autorisé d'exécuter la sienne.

Une solution doit satisfaire :

- **Exclusion mutuelle** : si un processus exécute sa section critique aucun autre processus ne peut exécuter sa section critique.
- **Déroulement** : si aucun processus n'exécute sa section critique et si certains processus désirent entrer dans les leurs, alors seulement les processus qui ne se trouvent pas dans leurs section restantes peuvent décider qui rentrera prochainement dans sa section critique. Cette sélection ne peut pas être reportée indéfiniment.
- **Attente limitée** : il doit exister une limite au nombre de fois que l'on permet à d'autres processus à entrer dans leurs sections critiques après qu'un processus a effectué une requête pour entrer dans sa section critique et avant que la requête ne soit accordée.

Algorithme 1 (2 processus)

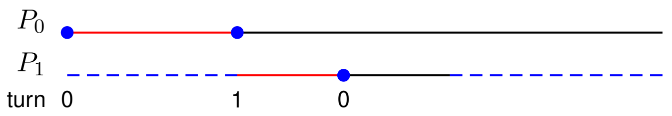
```
int turn = 0 ou 1;
```

PROCESSUS $P_i, i = 0, 1$

```
while(1){
    while(turn != i);
    section critique
    turn = 1 - i;
    section restante
}
```

Exclusion mutuelle : OK

Déroulement : NON



Algorithme 2 (2 processus)

```
int flag[2] = {0, 0};
```

PROCESSUS $P_i, i = 0, 1$

```
while(1){
    flag[i] = 1;
    while(flag[1 - i]);
    section critique
    flag[i]= 0;
    section restante
}
```

Exclusion mutuelle : OK

Déroulement : NON

t_O : P_0 fixe `flag[0]` à 1

t_1 : P_1 fixe `flag[1]` à 1

$\Rightarrow$ interblocage

Algorithme 3 (2 processus)

```
int flag[2] = {0, 0}, turn = 0 ou 1;
```

PROCESSUS $P_i, i = 0, 1$

```
while(1){
   flag[i] = 1;
   turn = 1 - i;
   while(flag[1 - i] && turn != i);
   section critique
   flag[i]= 0;
   section restante
}
```

Exclusion mutuelle, déroulement, attente limitée - OK

Algorithme du boulanger

```
int flag[n] = {0,...,0}, num[n] = {0,...,0}
```

PROCESSUS $P_i, i = 0, \ldots, n-1$

```
while(1){
   flag[i] = 1;
   num[i] = max(num[0],...,num[n-1]) + 1;
   flag[i] = 0;
   for (j = 0; j < n; j++) {
      while (flag[j]);
      while (num[j]!=0 && (num[j],j)<(num[i],i));
   }
   section critique
   num[i]= 0;
   section restante
}
```


Sémaphores

Introduction

Les solutions au problème de la section critique sont difficiles à généraliser pour des problèmes plus complexes.

Sémaphores : outil de synchronisation proposé par Edsger Dijkstra (1965).

- un compteur entier
- une opération P (*proberen* - tester)
- une opération V (*verhogen* - incrémenter)

Définition classique

```
class Semaphore
{  int cpt;

   Semaphore(int n) {cpt = n;}

   P()
   {  while (cpt <= 0); // on suppose que le dernier test
      cpt--;            // et la -- sont une opération atomique
   }

   V()
   { cpt++; } //on suppose que ++ est une opération atomique
}
```


Section critique - solution

```
Semaphore mutex(1);
```

PROCESSUS $P_i, i = 0, \ldots, n-1$

```
while(1){
   mutex.P();
   section critique
   mutex.V();
   section restante
}
```


Exemple

Processus P_i execute la section S_i, $i = 0, 1$. On désire que S_1 ne soit exécutée qu'après la fin de S_0.

Solution

```
Semaphore sync(0);
```

PROCESSUS P_0	PROCESSUS P_1
...	...
S_0	sync.P();
sync.V();	S_1
...	...

Sémaphores - implémentation

Principal désavantage des solutions proposées : *attente active*

```
while(condition);
```

Attente active = gaspillage du temps CPU

Solution : implémentation de sémaphores sous forme d'appels système

- le processus en attente s'endormit.
- le SE le réveille quand il peut continuer

Sémaphores - implémentation

```
class Semaphore
{  int cpt;
   liste l;
   Semaphore(int n) {cpt = n; l =();}
   P()
   {  if(cpt <= 0)
      {  p = processus appelant;
         endormir(p);
         l.ajouter(p);
      }
      else cpt--;
   }
   V()
   {  cpt++;
      if (cpt > 0 && !l.vide())
      {  p = l.retirer();
         reveiller(p);
         cpt--;
      }
   }
}
```


Attention aux interblocages!

```
Semaphore a(1), b(1);
```

PROCESSUS P_0	PROCESSUS P_1
...	...
a.P();	b.P();
b.P();	a.P();
...	...
b.V();	a.V();
a.V();	b.V();
...	...

Pb1 (Producteur - Consommateur)

```
buf[N];
```

PRODUCTEUR

```
in = 0;
while (1)
{  produire next;
   buf[in] = next;
   in = (in + 1) % N;
}
```

CONSOMMATEUR

```
out = 0;
while (1)
{  next = buf[out];
   out = (out + 1) % N;
   consommer next;
}
```


Solution de Pb1

```
buf[N];
Semaphore empty(N), full(0);
```

PRODUCTEUR

```
in = 0;
while (1)
{  produire next;
   empty.P();
   buf[in] = next;
   in = (in + 1) % N;
   full.V();
}
```

CONSOMMATEUR

```
out = 0;
while (1)
{  full.P();
   next = buf[out];
   out = (out + 1) % N;
   empty.V();
   consommer next;
}
```

Pb2 (Lecteurs - Rédacteurs)

```
objet partagé X
```

LECTEUR
```
while (1)
{  ...
   lecture de X
   ...
}
```

RÉDACTEUR
```
while (1)
{  ...
   modification de X
   ...
}
```

- plusieurs lectures simultanées sont possibles
- rédacteurs - accès exclusive

Version 1 : priorité aux lecteurs

Version 2 : priorité aux rédacteurs

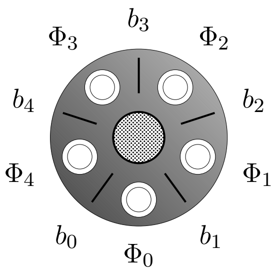
Solution de Pb2

```
objet partagé X
Semaphore wrt(1), mutex(1);
int readcnt = 0;
```

LECTEUR
```
while (1)
{  ...
   mutex.P();
   readcnt++;
   if (readcnt=1) wrt.P();
   mutex.V();
   lecture de X
   mutex.P();
   readcnt--;
   if (readcnt=0) wrt.V();
   mutex.V();
   ...
}
```

RÉDACTEUR
```
while (1)
{  ...
   wrt.P();
   modification de X
   wrt.V();
   ...
}
```


Pb 3 (Le dîner des philosophes)



PHILOSOPHE $\Phi_i, i = 0, \ldots, 4$
```
while(1)
{  penser();
   prendre b_i et b_((i+1) mod 5)
   manger();
   lâcher b_i et b_((i+1) mod 5)
}
```


Pb 3 : tentative de solution

```
Semaphore b[5](1);
```

PHILOSOPHE $\Phi_i, i = 0, \ldots, 4$
```
while(1)
{  penser();
   b[i].P();
   b[(i + 1) % 5].P();
   manger();
   b[i].V();
   b[(i + 1) % 5].V();
}
```

Les philosophes peuvent mourir de faim !!!

Pb 3 : Solution 1

Permettre au maximum à 4 philosophes d'être assis à la table

```
Semaphore b[5](1), s(4);
```

PHILOSOPHE $\Phi_i, i = 0, \ldots, 4$

```
while(1)
{  penser();
   s.P();
   b[i].P();
   b[(i + 1) % 5].P();
   manger();
   b[i].V();
   b[(i + 1) % 5].V();
   s.V();
}
```


Pb 3 : Solution 2

Solution asymétrique

```
Semaphore b[5](1);
```

PHILOSOPHE $\Phi_i, i = 0, \ldots, 4$

```
while(1)
{  penser();
   if (i % 2)
   {  b[i].P();
      b[(i + 1) % 5].P();
   }
   else
   {  b[(i + 1) % 5].P();
      b[i].P();
   }
   manger();
   b[i].V();
   b[(i + 1) % 5].V();
}
```


Les IPC

Inroduction

- IPC = **I**nter **P**rocess **C**ommunication
- apparues en 1983 dans Unix System V
- permettent communication et synchronisation rapides et efficaces
- ne sont pas intégrées dans le système de fichiers
- regroupent trois mécanismes
  - files de messages
  - segments de mémoire partagée
  - sémaphores

Fonctions de gestion

IPC	création/accès	contrôle	opérations
messages	`msgget`	`msgctl`	`msgsnd`, `msgrcv`
mémoire	`shmget`	`shmctl`	`shmat`, `shmdt`
sémaphores	`semget`	`semctl`	`semop`

Gestion de clefs

Pour créer ou accéder une IPC, il faut posséder une clef.

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(const char *pathname, int proj_id);
```

Cette fonction crée une clef unique à partir d'un nom de fichier (typiquement on utilise le nom d'exécutable qui crée l'IPC) et un numéro de projet.

Droits d'accès des IPC : le même principe de fonctionnement que les droits d'accès des fichiers.

Files de messages

- File de messages $\sim$ "boîte à lettres"
- Message = type + contenu
- Les processus autorisés peuvent :
  - déposer des messages
  - lire des messages dans l'ordre d'arrivée selon le type

Création/acces

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

Exemple :

```
/* création */
cle = ftok(argv[0], 1);
msqid = msgget(cle, 0644|IPC_CREAT|IPC_EXCL);

/* accès */
cle = ftok("toto", 1);
msqid = msgget(cle, 0);
```

Envoie/réception de messages (1)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msqid, struct msgbuf *msgp,
   size_t msgsz, int msgflg);
ssize_t msgrcv(int msqid, struct msgbuf *msgp,
   size_t msgsz, long msgtyp, int msgflg);
```

Exemple :

```
typedef struct
{  long type; /* champs obligatoire */
   char nom[30];
   int age;
   double poids;
} monmsg
```


Envoie/réception de messages (2)

Exemple (suite) :

```
monmsg m;

/* envoyer un message */
m.type = 3;
strcpy(m.nom, "toto"); m.age=666; m.poids = 8.35e7;
msgsnd(msqid, &m, sizeof(monmsg), 0);

/* lire le 1er message de type 20 */
msgrcv(msqid, &m, sizeof(monmsg), 20, 0);

/* lire le 1er message */
msgrcv(msqid, &m, sizeof(monmsg), 0, 0);

/* lire le 1er message qui n'est pas de type 20 */
msgrcv(msqid, &m, sizeof(monmsg), 20, MSG_EXCEPT);

/* lire le 1er message de min type ≤ 20 */
msgrcv(msqid, &m, sizeof(monmsg), -20, 0);
```


Sémaphores IPC

Plus puissants que les sémaphores “classiques” de Dijkstra

- opérations atomiques sur *ensembles* de sémaphores
- opérations `P(n)` et `V(n)`
- permettent d'organiser “rendez-vous”

Opérations

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops,
   unsigned nsops);

struct sembuf
{  short sem_num; /* numéro de sémaphore 0..nsops-1*/
   short sem_op; /*cette valeur est ajoutée au compteur du sémaphore*/
      /* sem_op < 0 - opération P(n) */
      /* sem_op > 0 - opération V(n) */
      /* sem_op = 0 - attente de valeur 0 du sémaphore (rdv) */
   short sem_flg; /* IPC_NOWAIT, SEM_UNDO */
}
```

Exemple (Sémaphores de Dijkstra)

```
int sem_create(key_t key, int init_val)
{  int semid;
   semid = semget(key, 1, 0660|IPC_CREATE|IPC_EXCL);
   semctl(semid, SETVAL, init_val);
   return semid;
}

void P(int semid)
{  struct sembuf buf;
   buf.sem_num = 0; buf.sem_op = -1; buf.sem_flg = 0;
   semop(semid, &buf, 1);
}

void V(int semid)
{  struct sembuf buf;
   buf.sem_num = 0; buf.sem_op = 1; buf.sem_flg = 0;
   semop(semid, &buf, 1);
}
```


Exemple (rendez-vous)

```
semid = sem_create(cle, N);
/* départ de N processus */
...fork()...
............
/* point de rendez-vous */
P(semid);
buf.sem_num = 0;
buf.sem_op = 0;
buf.sem_flg = 0;
semop(semid, &buf, 1);
/* rendez-vous OK */
V(semid);
```


Exemple (philosophes)

```
baguettes = semget(IPC_PRIVATE, 5,
   0660|IPC_CREAT|IPC_EXCL);
for (i = 0; i < 5; i++)
   semctl(baguettes, i, SETVAL, 1);
...
/* forker 5 philosophes */
...
/* philosophe i a faim */
struct sembuf buf[2];

buf[0].sem_num = i; buf[1].sem_num = (i + 1) % 5;
buf[0].sem_op = buf[1].sem_op = -1;
buf[0].sem_flg = buf[1].sem_flg = 0;
/* prendre les deux baguettes simultanément */
semop(baguettes, buf, 2);
manger();
...
```


Mémoire partagée IPC

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

/* création/accès */
int shmget(key_t key, int size, int shmflg);

/* attacher dans son espace d'adressage */
void *shmat(int shmid, const void *shmaddr,
   int shmflg);

/* détacher */
int shmdt(const void *shmaddr);
```

Exemple (1)

```
int cmp(const void *a, const void *b)
{  sleep(1);
   return *(int *)a - *(int *)b;
}

void print_tab(int *tab)
{  int i;
   for (i = 0; i < N; i++) printf("%4d", tab[i]);
   putchar('\n');
}

int main(void)
{  int shmid, i, *tab;
   pid_t pid;

   shmid = shmget(IPC_PRIVATE, N * sizeof(int),
      0600|IPC_CREAT|IPC_EXCL);
   tab = shmat(shmid, NULL, 0);
```


Exemple (2)

```
   srand(getpid());
   for (i = 0; i < N; i++) tab[i] = rand() % N;
   print_tab(tab);
   pid = fork();
   qsort(tab, N, sizeof(int), cmp);
   if (pid == 0) { shmdt(tab); exit(0); }
   print_tab(tab);
   shmdt(tab);
   shmctl(shmid, IPC_RMID, NULL);
   return 0;
}
```

Exécution

```
[stefan@ecrins examples]$ shmtest
   4   1   7   4   9   7   1   3   6   0
   0   0   1   1   0   4   4   4   7   9
```


Suppression d'une IPC

```
<ipc>ctl(id, IPC_RMID, NULL);
```

```
<ipc> ∈ {msg, sem, shm}
```

Attention : Une IPC ne se supprime pas automatiquement!

Les commandes `ipcs` et `ipcrm`

```
[stefan@ecrins examples]$ ipcs
------ Shared Memory Segments --------
key        shmid  owner  perms bytes     nattch    statu
0x00000000 131073 stefan 600   393216    2         dest

------ Semaphore Arrays --------
key        semid  owner  perms nsems     status
0x00000000 163843 stefan 666   6

------ Message Queues --------
key        msqid  owner  perms used-bytes messages

[stefan@ecrins examples]$ ipcrm -s 163843
[stefan@ecrins examples]$ ipcs -s
------ Semaphore Arrays --------
key        semid  owner  perms nsems     status
```