Systèmes d'exploitations II

Présentation #5 : Exclusion mutuelle

14/11/2021

Ahmed Benmoussa

Centre universitaire d'Aflou

Rappel: Ordonnancement des processus

- **L’ordonnanceur CPU.**
- **Métriques de performance**: temps de résidence, temps d’attente, temps de réponse, …
- **Stratégie d’ordonnancement**: préemptif et non-préemptifs
- **Algorithme d’ordonnancement**: FIFO, SJF, RR, MLFQ, …

Communication Inter-processus (IPC)

- How one process can pass information to another process?
- Making sure that two processes do not get in each others way.
- Proper sequencing.

- **Programme N°1**

Thread A	Thread B	
x = 1;	x = 2;	**Quelle est la valeur de X?**

La valeur de X peut être 1 ou 2 (non-deterministe)

- **Programme N°2:** initialement y=12

Thread A	Thread B	
x = y+1;	y = y*2;	**Quelle est la valeur de X?**

Rappel: Cours Processus

```
/* Processus père */

int main (){
        int i=4, j=10;
        int p;
        p = fork();
        j += 2;
        if (p == 0){
                i += 3; j += 3;
        }
        else{
                i *= 2; j *= 2;
        }
        printf("\n proc= %d, i=%d, j=%d;
père = %d \n",getpid(), i, j, getppid());
return 0;
}
```

j=12 →

```
/* Processus fils */

int main (){
        int i=4, j=10;
        int p;
        p = fork();
        j += 2;
        if (p == 0){
                i += 3; j += 3;
        }
        else{
                i *= 2; j *= 2;
        }
        printf("\n proc= %d, i=%d, j=%d;
père = %d \n",getpid(), i, j, getppid());
return 0;
}
```

← j=12

Rappel: Cours Processus

```
/* Processus père */

int main (){
        int i=4, j=10;
        int p;
        p = fork();
        j += 2;
        if (p == 0){
                i += 3; j += 3;
        }
        else{
                i *= 2; j *= 2;
        }
        printf("\n proc= %d, i=%d, j=%d;
père = %d \n",getpid(), i, j, getppid());
return 0;
}
```

i=8

j=24

```
/* Processus fils */

int main (){
        int i=4, j=10;
        int p;
        p = fork();
        j += 2;
        if (p == 0){
                i += 3; j += 3;
        }
        else{
                i *= 2; j *= 2;
        }
        printf("\n proc= %d, i=%d, j=%d;
père = %d \n",getpid(), i, j, getppid());
return 0;
}
```

i=7

j=15

Rappel: Cours Processus

```
/* Processus père */

int main (){
        int i=4, j=10;
        int p;
        p = fork();
        j += 2;
        if (p == 0){
                i += 3; j += 3;
        }
        else{
                i *= 2; j *= 2;
        }
        printf("\n proc= %d, i=%d, j=%d;
père = %d \n",getpid(), i, j, getppid());
return 0;
}
```

i=8

j=24

```
/* Processus fils */

int main (){
        int i=4, j=10;
        int p;
        p = fork();
        j += 2;
        if (p == 0){
                i += 3; j += 3;
        }
        else{
                i *= 2; j *= 2;
        }
        printf("\n proc= %d, i=%d, j=%d;
père = %d \n",getpid(), i, j, getppid());
return 0;
}
```

i=7

j=15

Résultat

proc= 18317, i=8, j=24; père = 18310
proc= 18318, i=7, j=15; père = 18317

- **Programme N°3:** Initialement x = 0

Thread A

x = x+1;

Thread B

x = x+2;

Quelle est la valeur de X?

La valeur de **X = 3**

Execution 1

Thread A	**Thread B**
load r1, x	
	load r1, x
add r2, r1, 1	
	add r2, r1, 2
store x, r2	
	store x, r2
final: x == 2	

Execution 2

Thread A	**Thread B**
load r1, x	
	load r1, x
add r2, r1, 1	
	add r2, r1, 2
	store x, r2
store x, r2	
final: x == 1	

Programmation concurrente est difficile

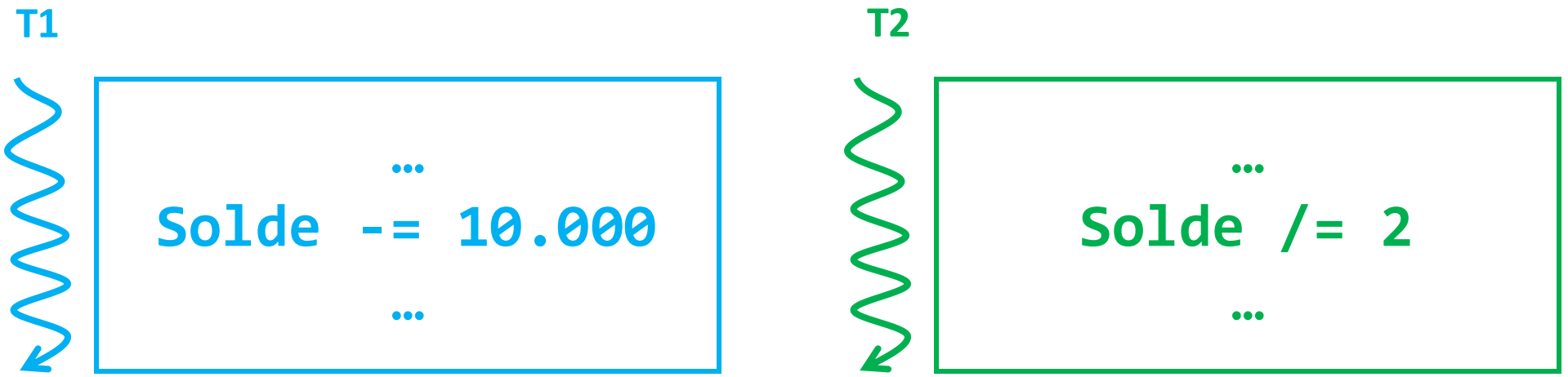
Pourquoi?

- Les programmes concurrent sont **non deterministes**.
  Exécuter deux fois avec les même valeurs, deux differentes réponses.
- Les instructions du programme sont exécutées de facon non atomique l'instruction **`x+=1`** peut etre compilée comme suit:

```
LOAD x
ADD 1
STORE x
```

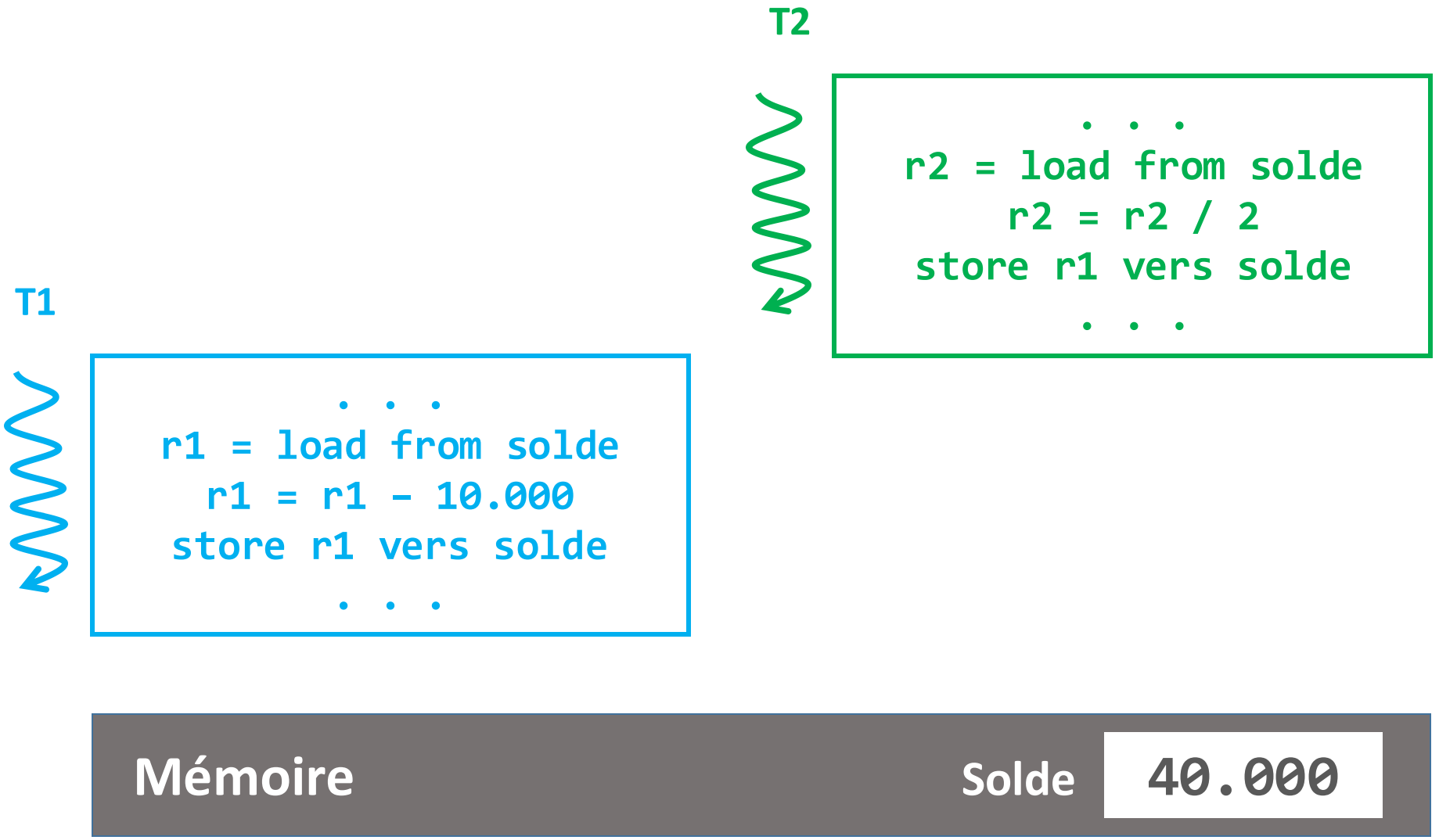
Gestion de compte bancaire

- Supposant deux Threads modifiant le solde d'un compte bancaire

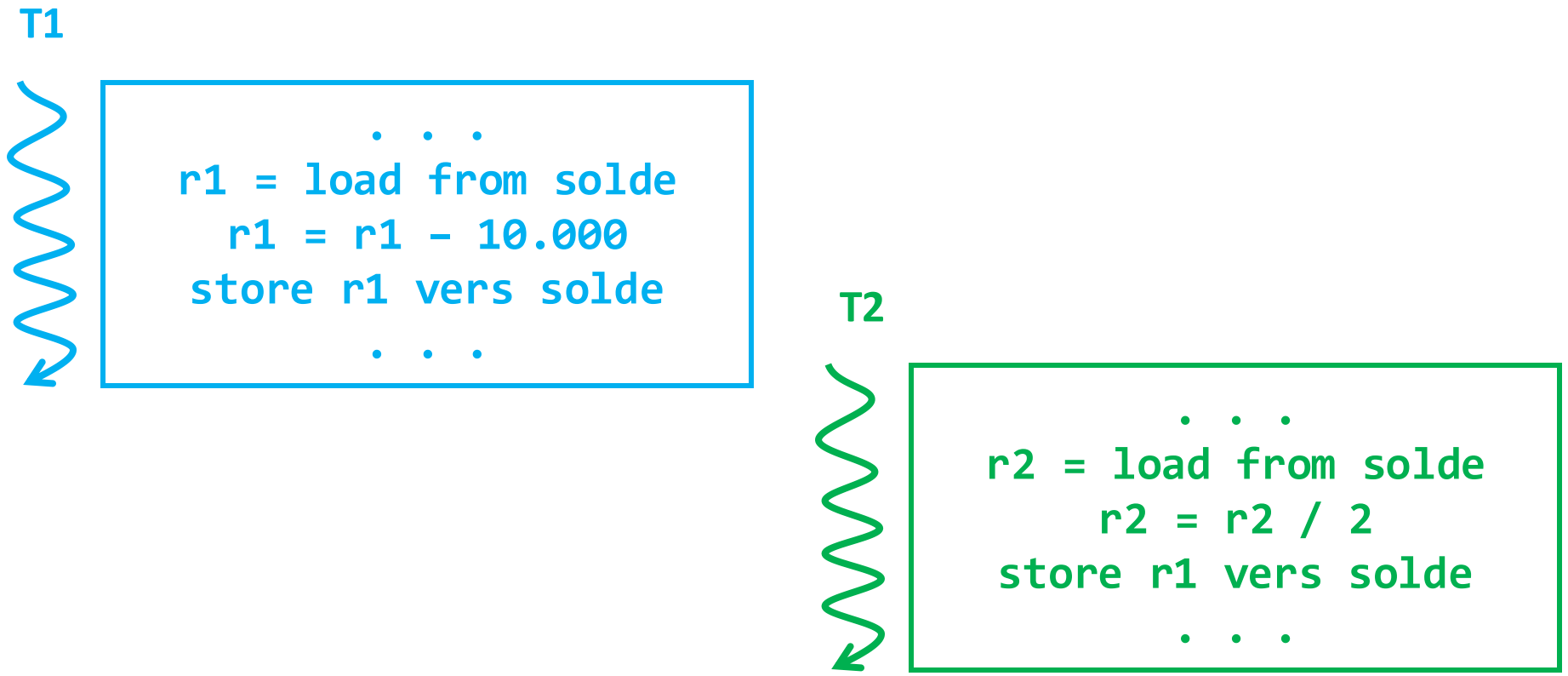


- Que ce passe t-il quand les deux Threads sont en execution?

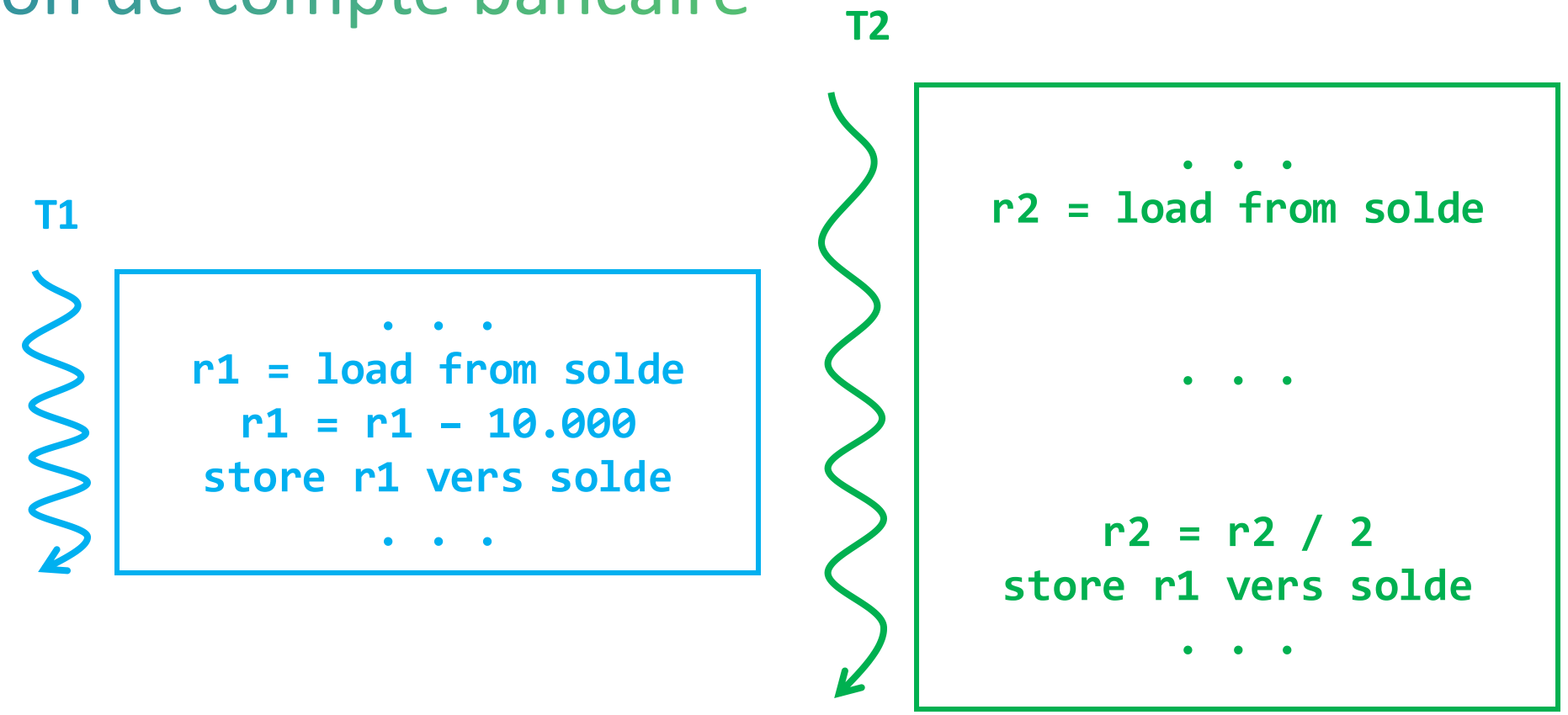
Gestion de compte bancaire



Gestion de compte bancaire



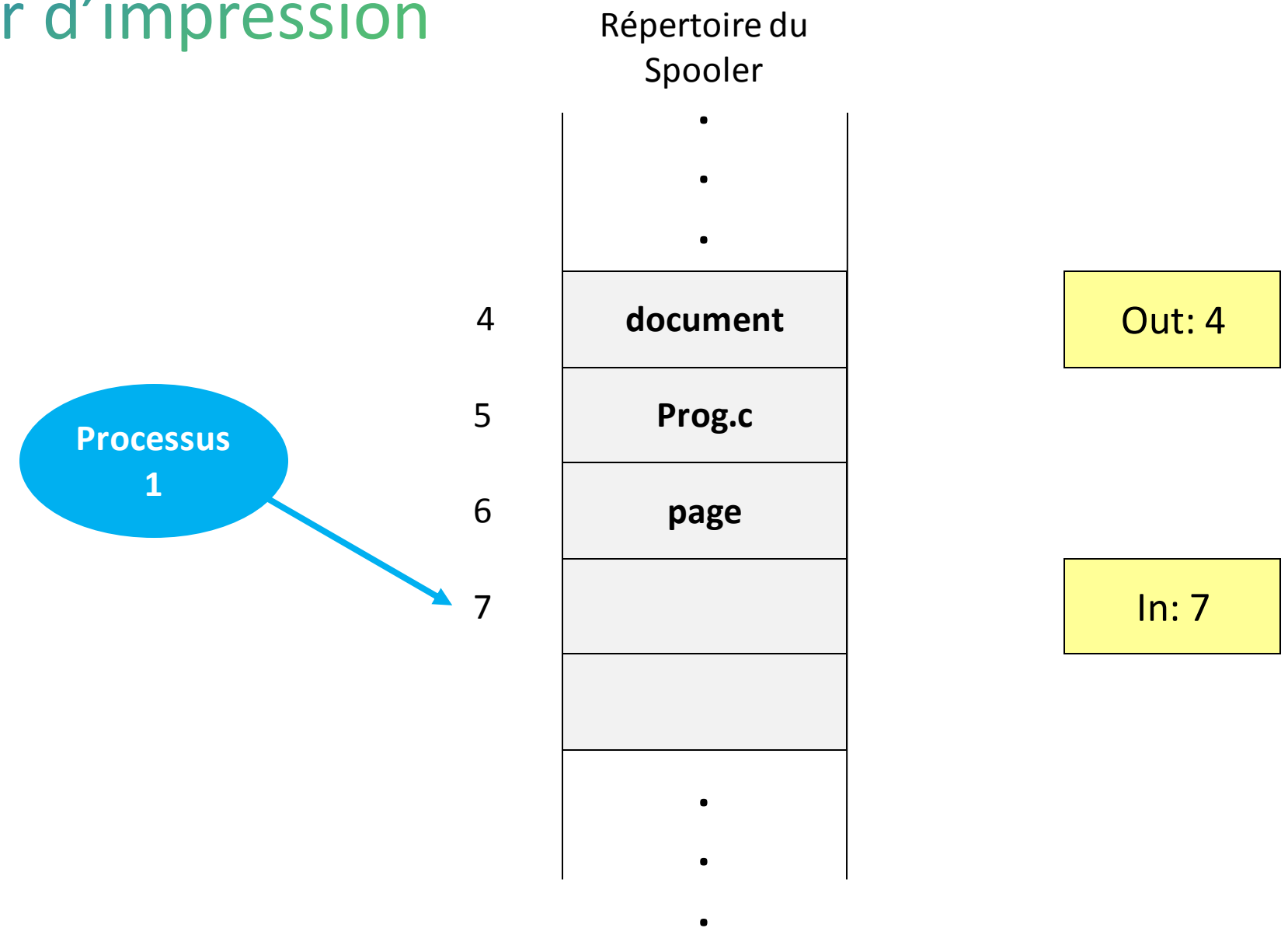
Gestion de compte bancaire



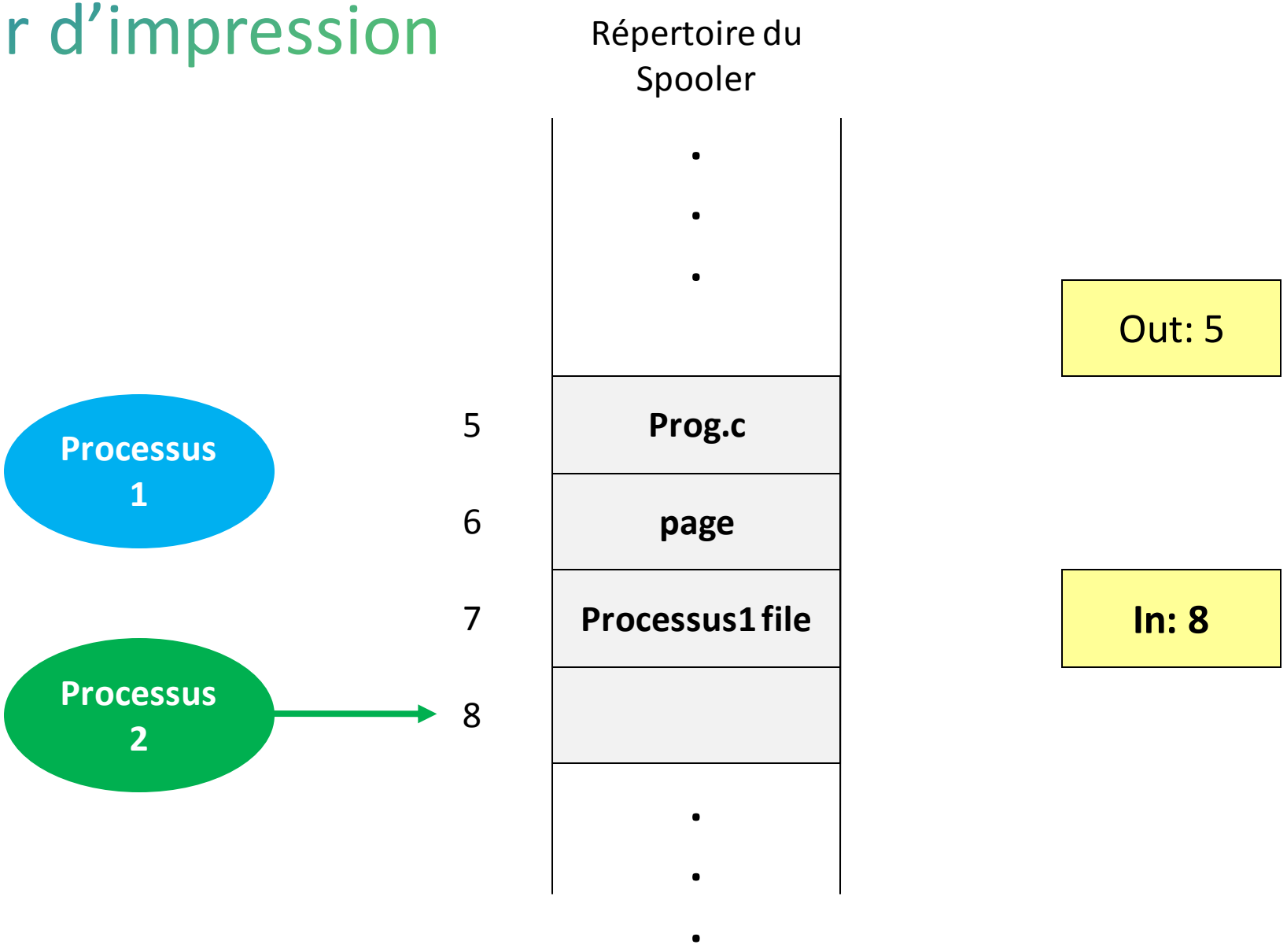
FAUX!

Ce problème est très difficile a déboguer

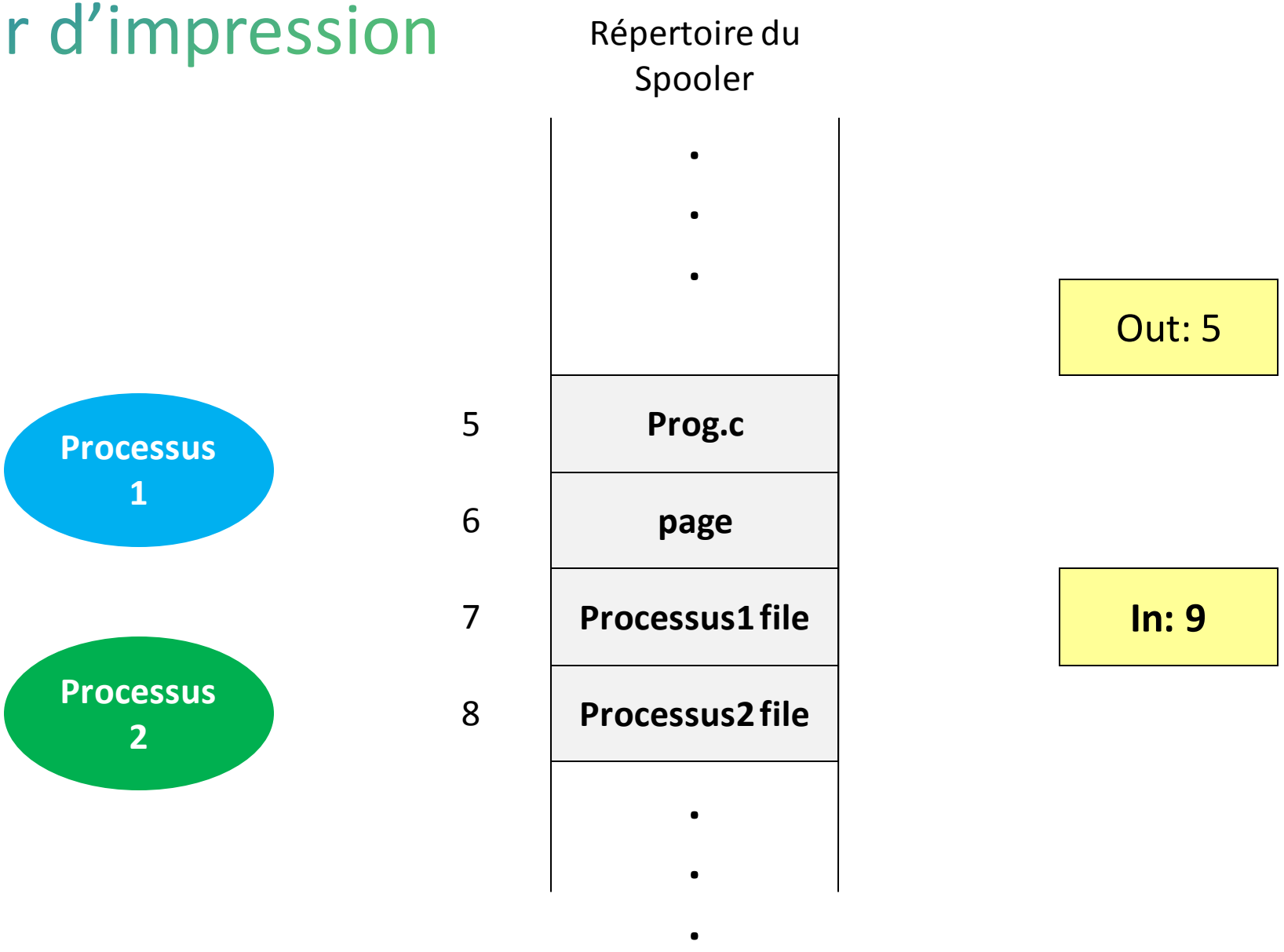
Spooler d'impression



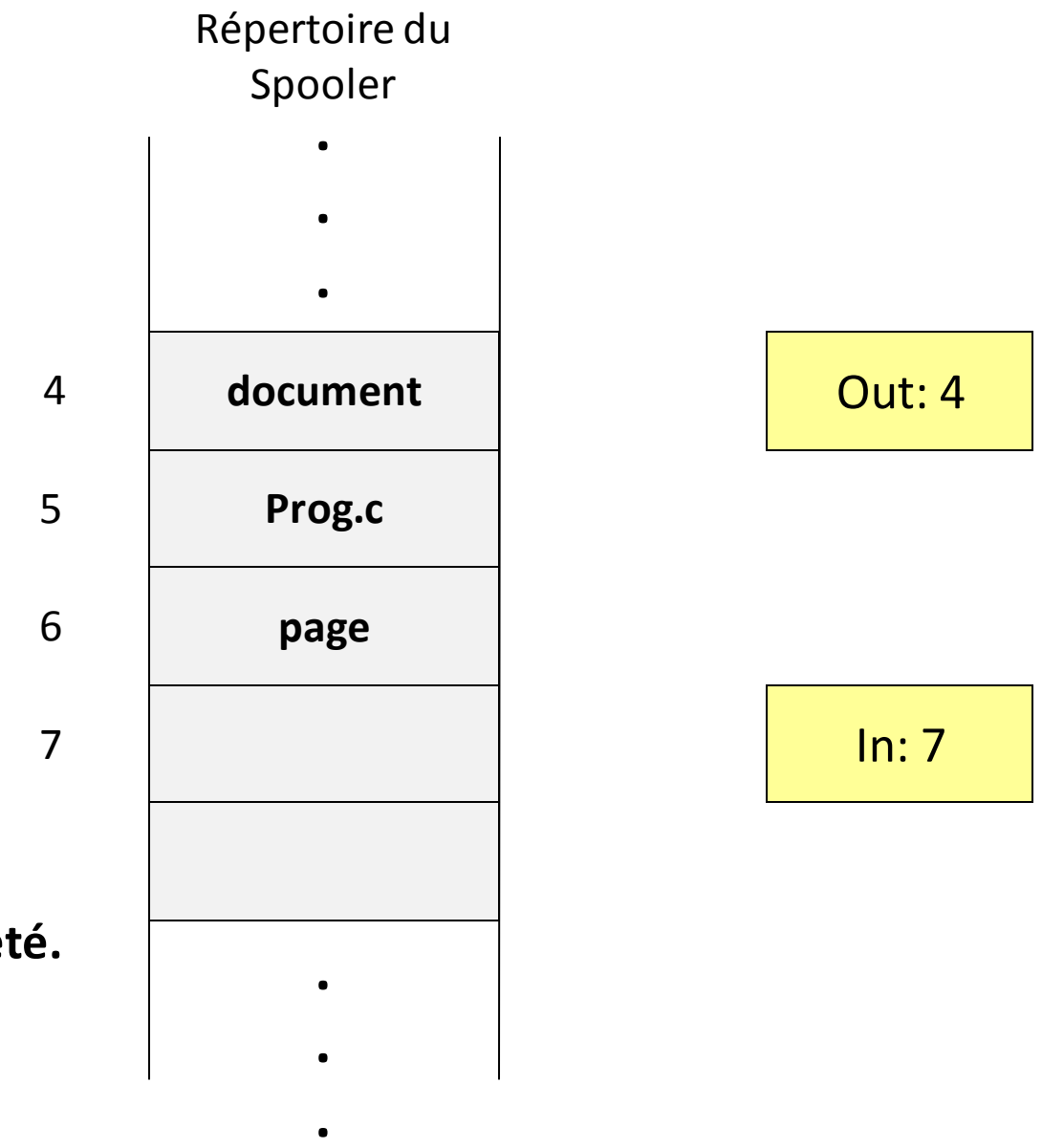
Spooler d'impression



Spooler d'impression



Problème

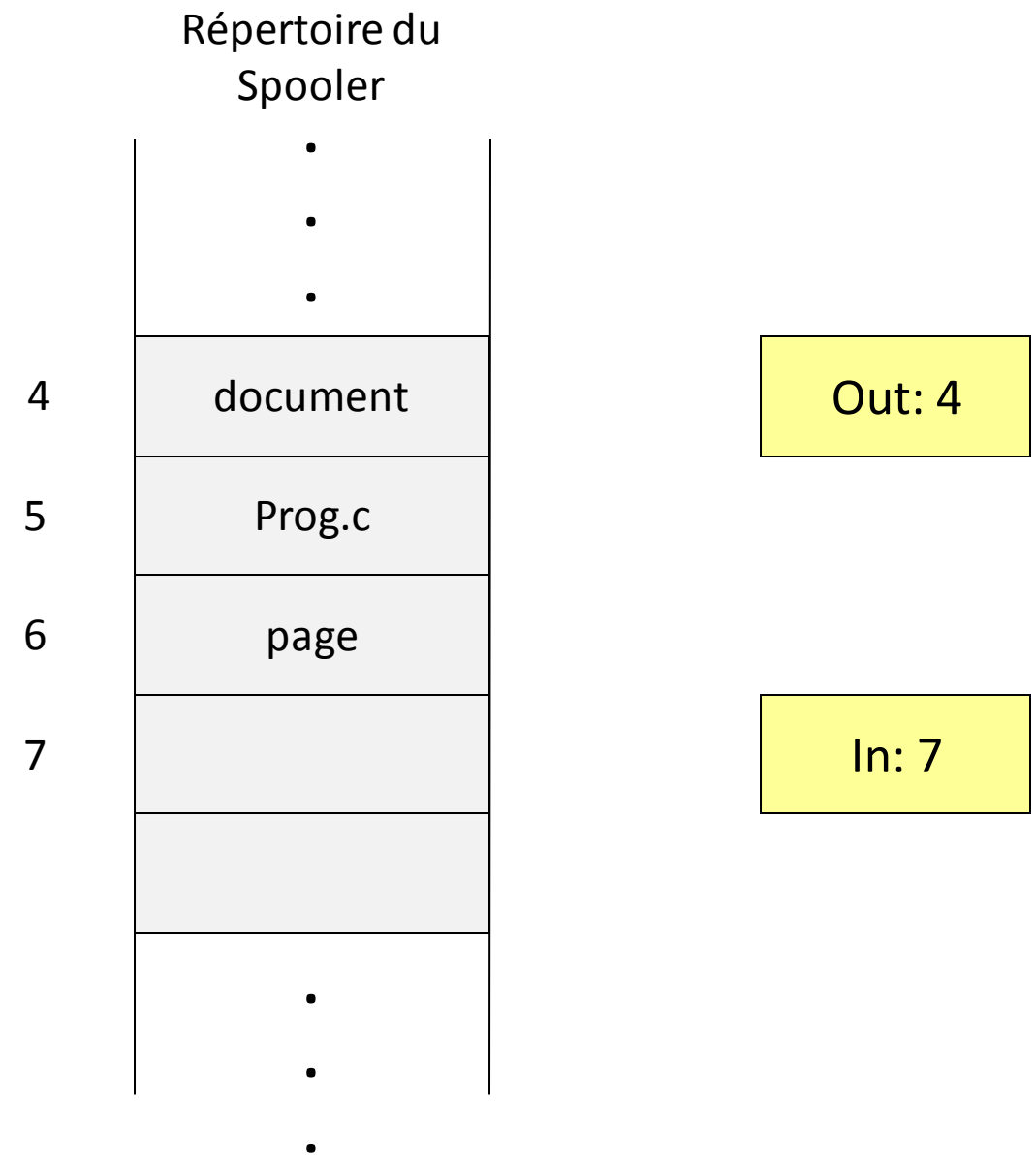


A ce moment le Processus 1 est arrêté.

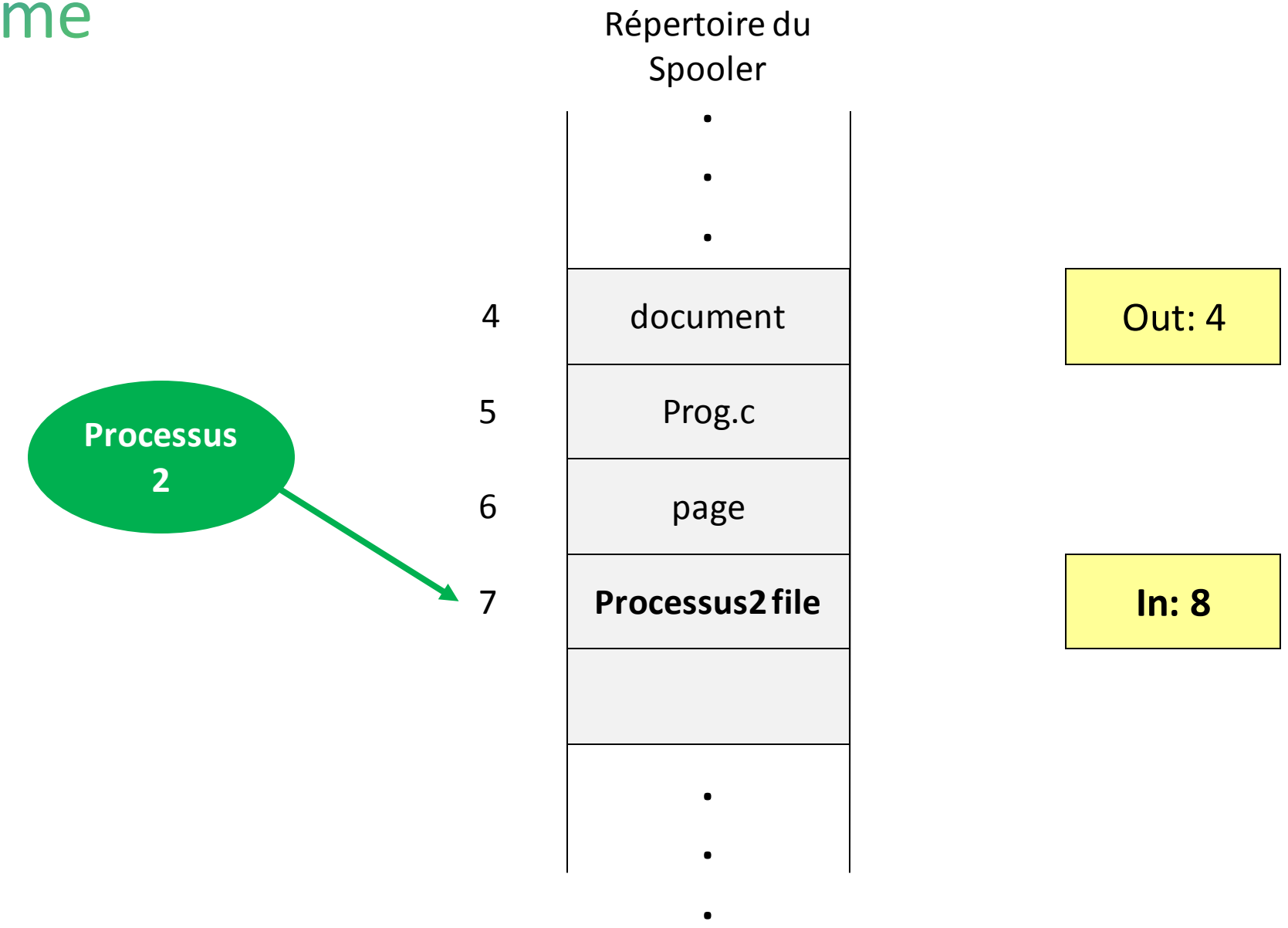
Problème

Le processus 2 est en exécution.


```
next_free_slot = 7
```

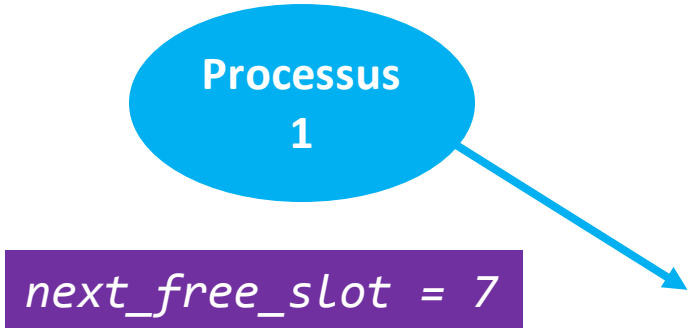


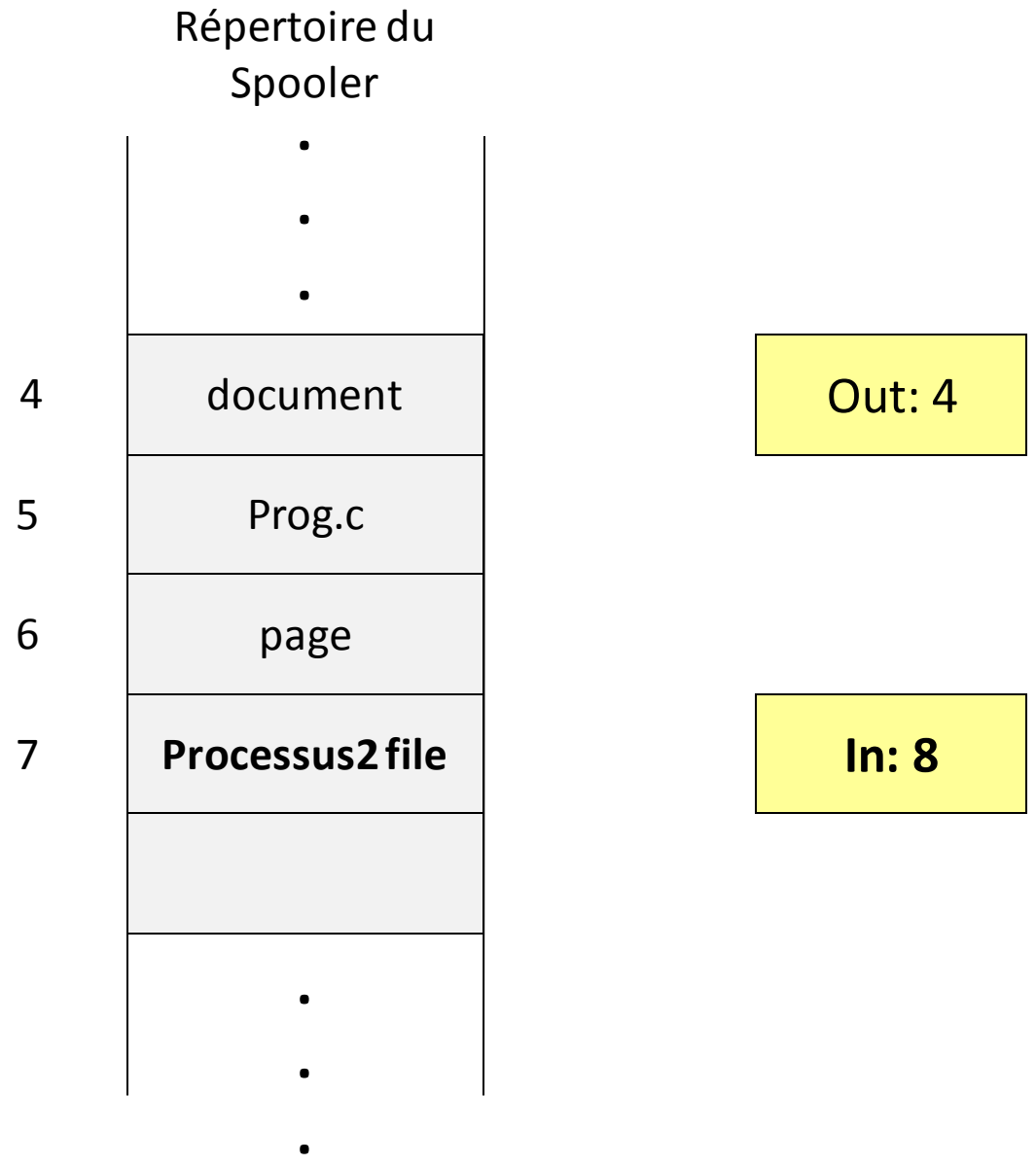
Problème


Répertoire du Spooler
Processus 2
4
document
5
Prog.c
6
page
7
Processus2 file
Out: 4
In: 8

Problème

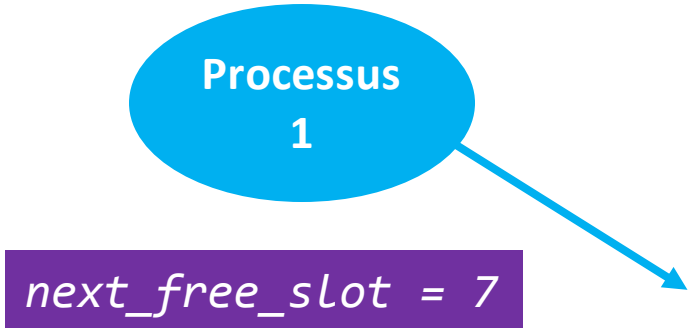
le Processus 1 est en exécution.

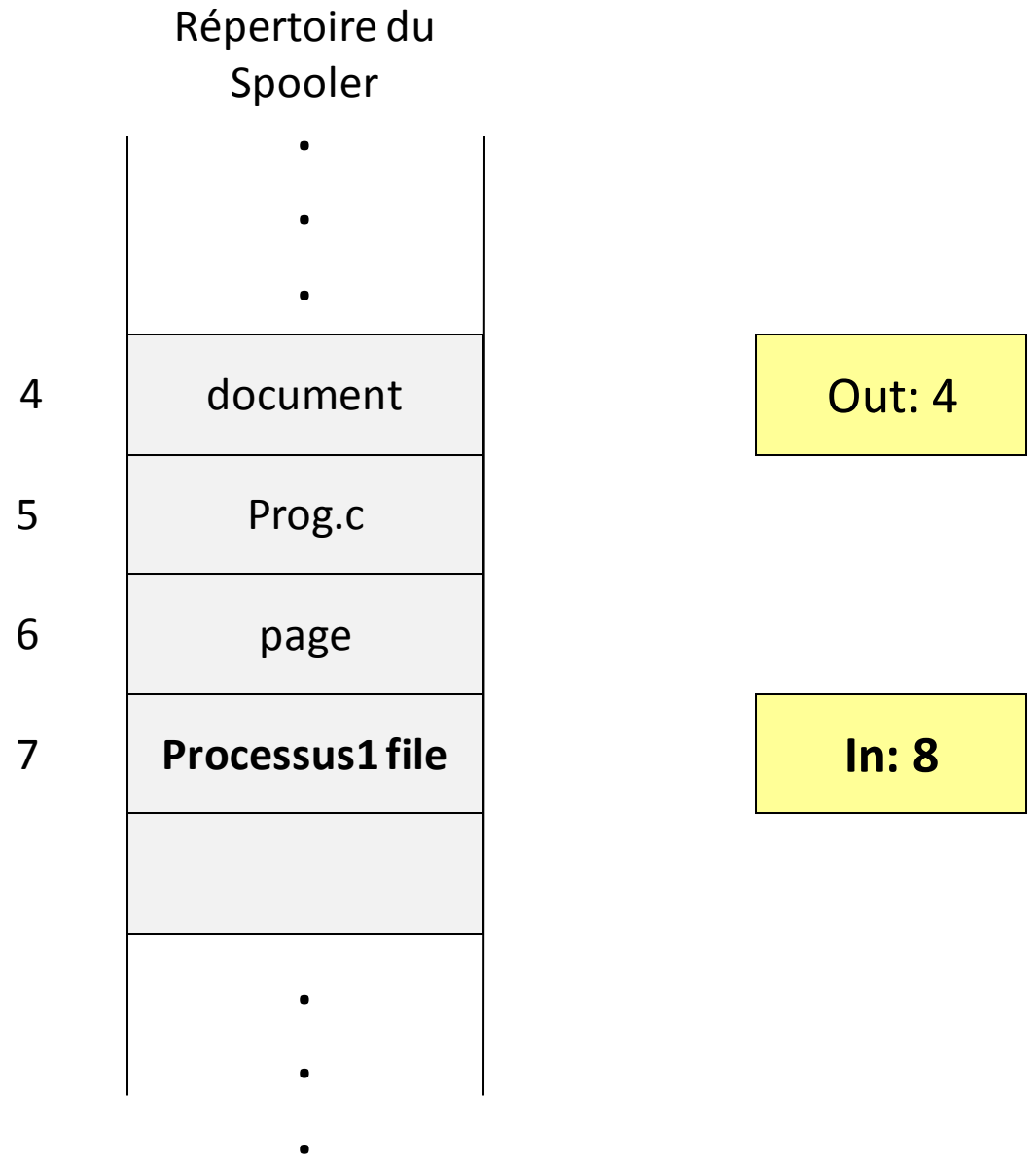




Problème

le Processus 1 est en exécution.



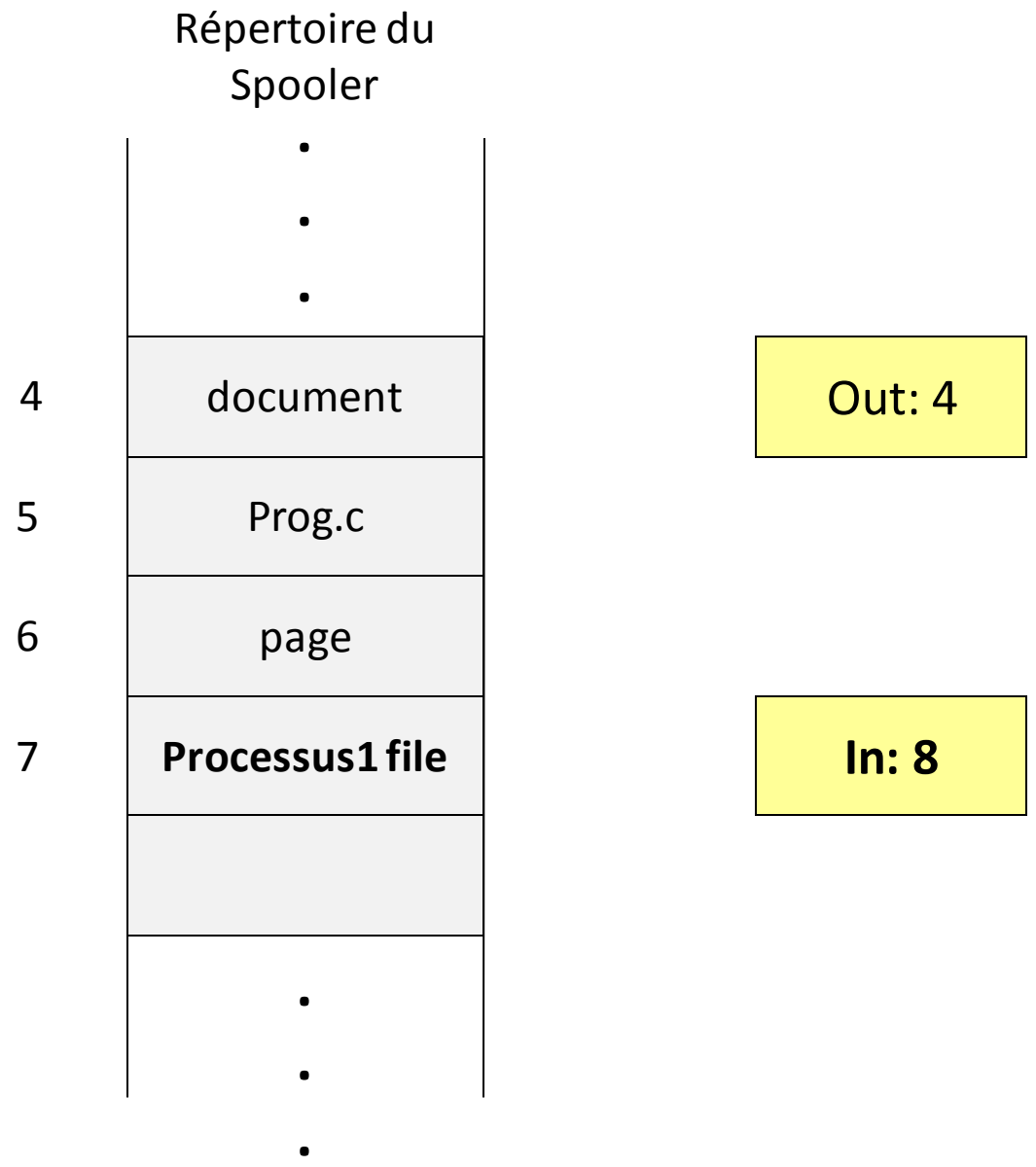


Problème

le Processus2 est remis en exécution.

Processus 2

Son fichier ne sera jamais imprimé



Compétition pour l'accès aux ressources (Race condition)

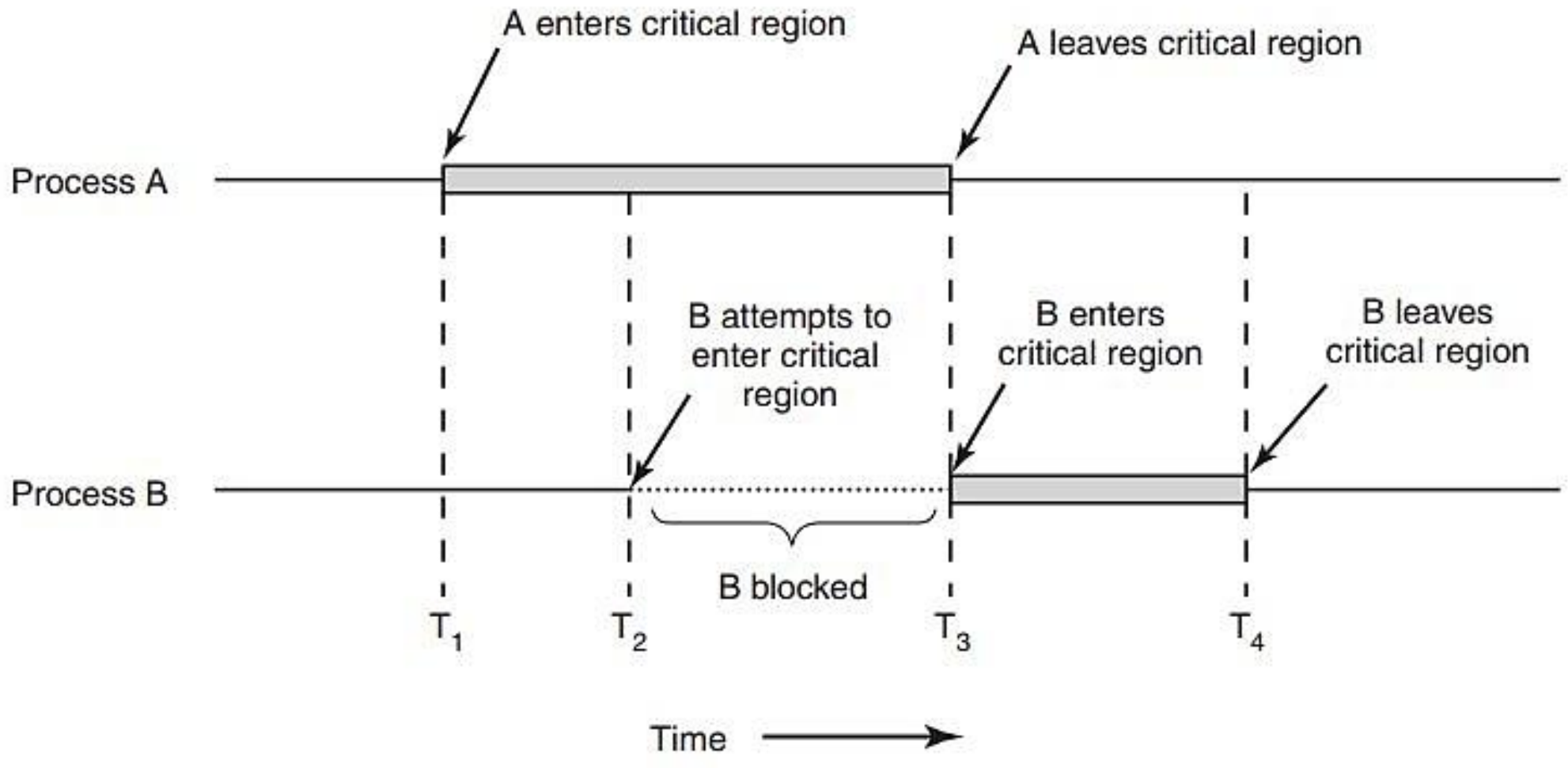
- Les processus peuvent partagé des ressources: espace mémoire, fichiers partagé, …ect.
- Les processus (ou Threads) sont en course, et le résultat de l'exécution du programme depend de qui gagne cette course.

Comment éviter cette compétition d’accès aux ressources?

Région critique (Critical region)

- Région critique ou section critique.
- S’assurer qu’un seul processus accède (lecture/écriture) à la ressource partagée.
- La partie du programme qui utilise une ressource partagée est appelée **section critique** ou **région critique**.

Exemple


A enters critical region
A leaves critical region
Process A
B attempts to enter critical region
B enters critical region
B leaves critical region
Process B
B blocked
T1
T2
T3
T4
Time

Conception d’une solution efficace

- Un et un seul processus doit être à l’interieur d’une région critique.
- Les processus qui ne sont pas à l’interieur de leur région critique ne doivent pas bloquer les autres processus.
- Aucun processus ne doit attendre en infinité pour entrer dans sa région critique.

Exclusion mutuelle

- Mécanisme permettant l'accès exclusif à une ressource commune (ou ressource partagée).
- Deux type de ressources: physique ou logique.
- Physique: processeur; mémoire; périphériques, …ect.
- Logique: programmes; fichiers; …ect.
- Un seul processus, à la fois, peut utiliser la ressource.
- Une ressource critique doit être utilisée en exclusion mutuelle.

1ère approche: Désactivation des interruptions

- Sur un système mono processeur, une solution simple consiste a désactiver les interruption quand un processus accède a une région critique.
- Donner le pourvoir au processus n'est pas une bonne solution.
- N'est pas efficace sur un système multi-processeurs.

- **Résultat: Nécessité d'une solution plus efficace.**

2ème approche: Variables de blocage ou verrous

- Lock variables
- Solution logicielle.
- Variable à valeur binaire (0 et 1)
- Si la valeur = 0, aucun processus n'est dans sa région critique.
- Si valeur = 1, un processus est entré en section critique.

Verrous: mode de fonctionnement

- Si un processus veut entrer en section critique, il vérifie tout d'abord la valeur du verrous:
- Si verrous = 0 , le processus met verrous = 1 et entre en section critique.
- Sinon, le processus attent jusqu'à que la valeur du verrous = 0

Verrous: Problème

- Possède le même problème vu avec le spooler de l'imprimante.
- Dans un temps t processus1 vérifie que la valeur = 0 a ce moment un autre processus2 entre en exécution.
- Processus2 met verrous = 1 et entre en section critique.
- Quand le premier processus est remis en état d'exécution, lui aussi met le verrous = 1 et entre en section critique.
- Les deux processus sont en section critique!

3ème approche: Alternation strict

- Une variable commune est utilisée, appelée **variable de verrouillage**.
- Un processus attend une valeur pour entrer dans sa région critique.
- Une fois sorti de sa section critique, le processus remodifie la valeur de la variable de verrouillage.

3ème approche: Alternation strict

```
while (TRUE) {
while (turn != 0) /* loop */ ;
      critical region( );
      turn = 1;
      noncritical region( );
}
```

```
while (TRUE) {
      while (turn != 1);
      critical region( );
      turn = 0;
      noncritical region( );
}
```

Alternation strict: problème

- Problème: Un processus qui est hors sa région critique peut bloquer un autre processus.

3ème approche: Alternation strict

Processus 1

```
while (TRUE) {
while (turn != 0) /* loop */ ;
    critical region( );
    turn = 1;
    noncritical region( );
}
```

Processus 2

```
while (TRUE) {
    while (turn != 1);
    critical region( );
    turn = 0;
    noncritical region( );
}
```

turn = 0

3ème approche: Alternation strict

Processus 1

```
while (TRUE) {
while (turn != 0) /* loop */ ;
      critical region( );
      turn = 1;
      noncritical region( );
}
```

Processus 2

```
while (TRUE) {
      while (turn != 1);
      critical region( );
      turn = 0;
      noncritical region( );
}
```

turn = 1

3ème approche: Alternation strict

Processus 1

```
while (TRUE) {
while (turn != 0) /* loop */ ;
      critical region( );
      turn = 1;
      noncritical region( );
}
```

Processus 2

```
while (TRUE) {
      while (turn != 1);
      critical region( );
      turn = 0;
      noncritical region( );
}
```

turn = 1

3ème approche: Alternation strict

Processus 1

```
while (TRUE) {
while (turn != 0) /* loop */ ;
      critical region( );
      turn = 1;
      noncritical region( );
}
```

Bloqué par Processus2

Processus 2

```
while (TRUE) {
      while (turn != 1);
      critical region( );
      turn = 0;
      noncritical region( );
}
```

turn = 1

Solution de Peterson

```
#define FALSE 0
#define TRUE 1
#define N 2                                                    /* number of processes */
int turn;                                                      /* whose turn is it? */
int interested[N];                                             /* all values initially 0 (FALSE) */
       void enter_region(int process);                         /* process is 0 or 1 */
       {
         int other;                                            /* number of the other process */
         other = 1 – process;                                  /* the opposite of process */
         interested[process] = TRUE;                           /* show that you are interested */
         turn = process;                                       /* set flag */
         while (turn == process && interested[other] == TRUE) /* null statement */ ;
       }
       void leave_region(int process)                                /* process: who is leaving */
       {
         interested[process] = FALSE;                  /* indicate departure from critical region */
       }
```

Solution de Peterson: problème

- Les deux processus appellent *enter_region* simultanément.
- Turn = 1
- Processus 0 accède à la région critique, Processus 1 attend processus 0

Instruction TSL

- Test and Set Lock
- Solution hardware.
- Le CPU utilisant l'instruction TSL verrue le bus mémoire.
- Aucun autres CPU ne peut accéder.
- Utilise une variable partagée *lock*.
- Copier la la valeur de lock sur un registre, puis lock = 1

Instruction TSL

```
enter_region:
TSL REGISTER,LOCK          | copy lock to register and set lock to 1
CMP REGISTER,#0            | was lock zero?
JNE enter_region           | if it was not zero, lock was set, so loop
RET                        | return to caller; critical region entered

leave region:
MOVE LOCK,#0               | store a 0 in lock
RET                        | return to caller
```

x86: Instruction XCHG

```
enter_region:
        MOVE REGISTER,#1        | put a 1 in the register
        XCHG REGISTER,LOCK      | swap the contents of the register and lock variable
        CMP REGISTER,#0         | was lock zero?
        JNE enter region        | if it was non zero, lock was set, so loop
        RET                     | return to caller; critical region entered

leave region:
MOVE LOCK,#0                    | store a 0 in lock
RET                             | return to caller
```

- La solution de Peterson et la méthode TSL sont des solutions correctes mais requièrent une **attente active** (*busy waiting*).
- Le Processus qui attend est en boucle constante.
- Consomme du temps CPU.

- Ce qui conduit à un autre problème

Problème de la priorité inversée

- Supposant deux processus H et L
- Processus H a une priorité forte et processus L priorité faible.
- L est dans sa section critique.
- H vient et occupe le CPU. Pour entrer dans sa section critique il doit attendre que L sorte.
- L attent que H libère le CPU pour sortir de sa section critique

- **Priority inversion problem**

Sleep and Wake up

- Bloquer un processus afin d'éviter le gaspillage du temps CPU.
- Utilisation de *sleep* et *wakeup*.
- *Sleep* est un appel système qui bloque celui qui l'a utilisé.
- *Wakeup* est un appel système utilisé pour réveiller un processus.

Problème Producteur-Consommateur

- Deux processus partage un buffer de taille fixe.
- Le processus *Producteur* insère des information au niveau du buffer.
- Le processus *Consommateur* extrait des informations du buffer.
- Peut être généralisé pour m *Producteurs* et n *consommateurs*.

Que ce passe t-il si le buffer est plain?

Problème Producteur-Consommateur: suite

- Producteur en mode *Sleep,* sera réveillé quand un espace se libère.
- Idem pour consommateur, quand le buffer est vide se met en veille. Se réveille quand une information est disponible.
- Nécessite l'utilisation d'une variable partagée.

Code Producteur

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */
void producer(void)
{
        int item;
        while (TRUE) {                                  /* repeat forever */
        item = produce_item();                          /* generate next item */
        if (count == N) sleep();                        /* if buffer is full, go to sleep */
        inser t item(item);                             /* put item in buffer */
        count = count + 1;                              /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);               /* was buffer empty? */
        }
}
```

Code Consommateur

```
void consumer(void)
{
        int item;
        while (TRUE) {                          /* repeat forever */
        if (count == 0) sleep();                /* if buffer is empty, got to sleep */
        item = remove_item();                   /* take item out of buffer */
        count = count - 1;                      /* decrement count of items in buffer */
        if (count == N - 1) wakeup(producer); /* was buffer full? */
        consume item(item);                     /* print item */
        }
}
```

Lost *wakeup* problem

- Producteur met une information, count = 1
- Envoie un wakeup au consommateur.
- Le signal est perdu.
- Le consommateur revérifie count = 0, se remet en ‘veille’ (s’endorme).
- Le producteur continue a remplir le buffer.
- **Both will sleep forever.**

Conclusion

- Communication Inter-processus.
- Race condition.
- Section Critique.
- Exclusion mutuelle.
- Solutions hardware et software.