L3 Informatique - Systèmes d'exploitation

Communications inter-processus

D. Béchet

Denis.Bechet@univ-nantes.fr

Université de Nantes

Faculté des Sciences et Techniques

2, rue de la Houssinière BP 92208

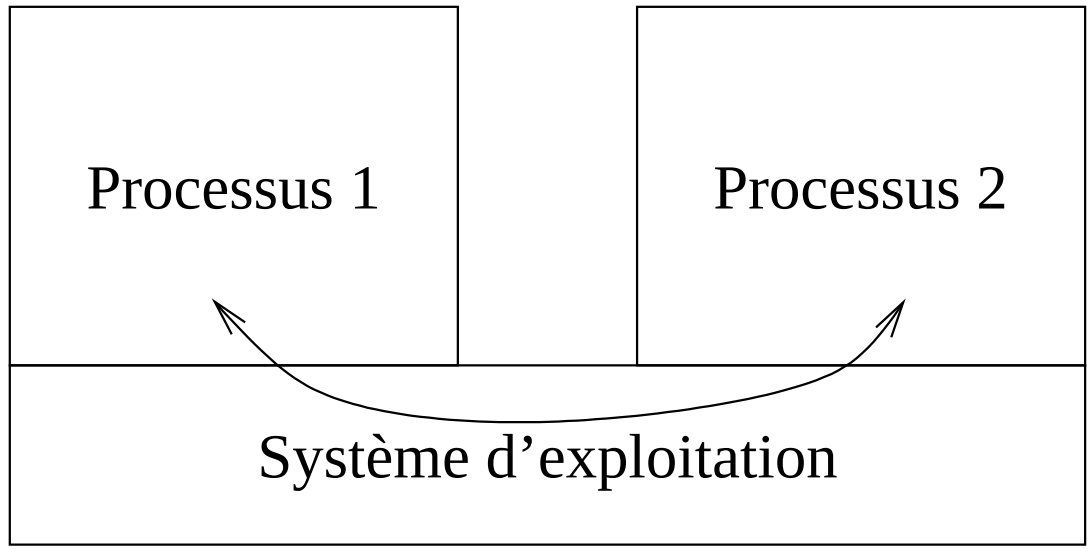
44322 Nantes cedex 3, France

http://www.sciences.univ-nantes.fr/info/perso/permanents/bechet

UNIVERSITÉ DE NANTES

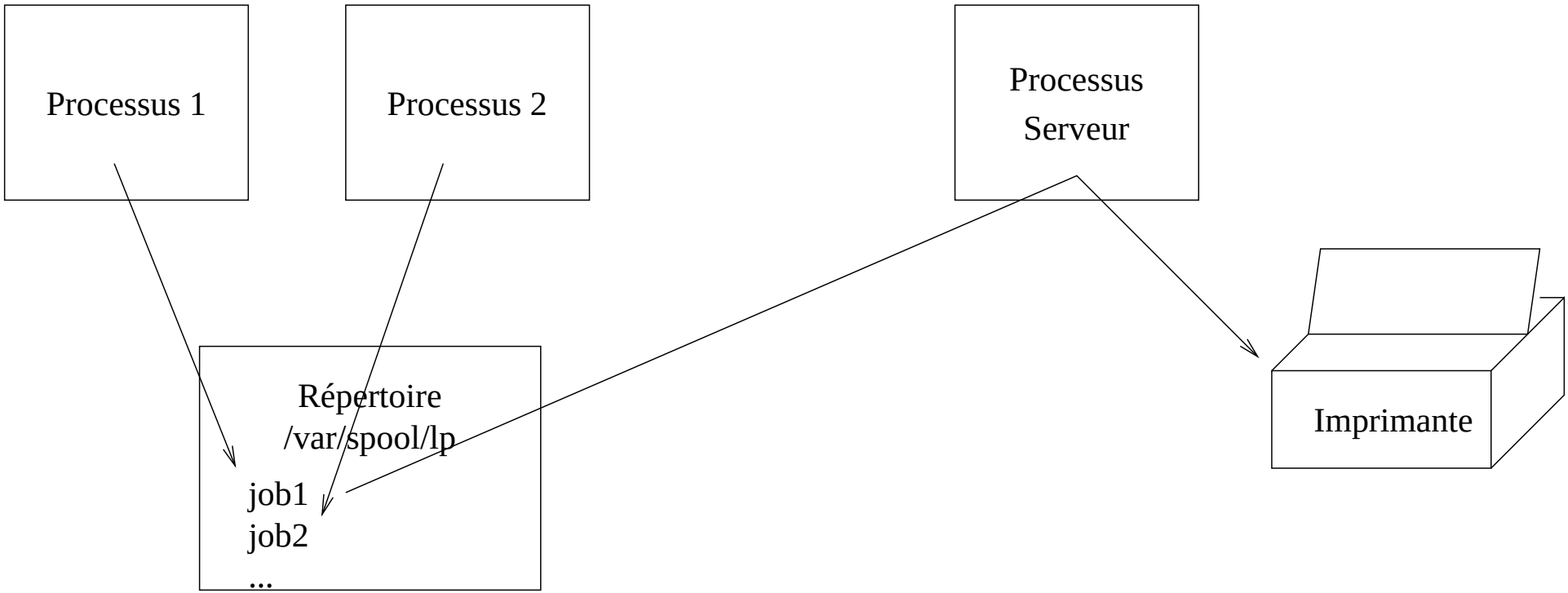
Communication entre processus

- Les processus ont besoin de communiquer entre eux:
  - Client/serveur (par exemple pour l'interface graphique)
  - Arrêt d'un processus, d'un service ou du système
  - Vérrouillage d'une ressource (imprimante)
  - ...



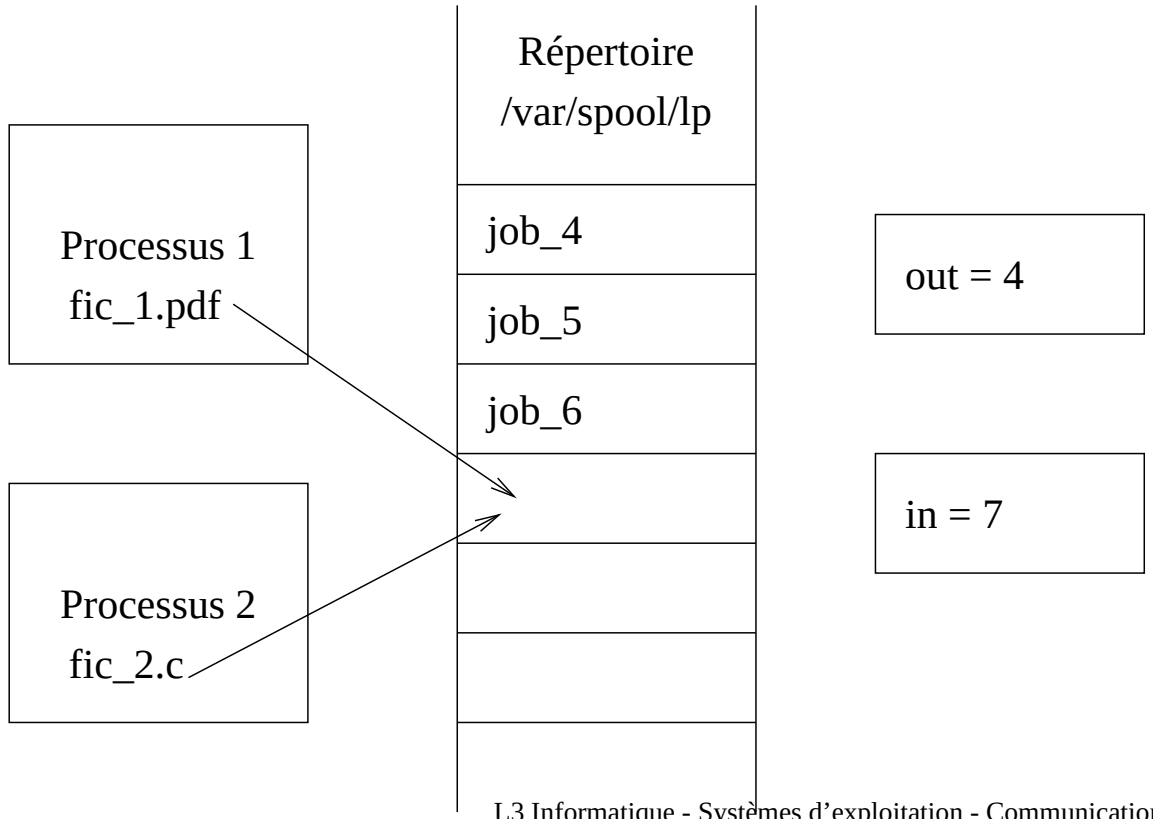
Ressources partagées

- Exemple : *spooling* pour l'imprimante



Situation de compétition

- Deux variables partagées :
  - *out* : donnant le prochain fichier à imprimer
  - *in* : donnant le prochain numéro libre dans le répertoire



Algo 1 de spooling

- Deux variables partagées :
  - *out* : donnant le prochain fichier à imprimer
  - *in* : donnant le prochain numéro libre dans le répertoire

1. Récupérer la variable *in* (valeur x)
2. Copier le fichier à imprimer sur job_x
3. Modifier la variable *in* (valeur $x + 1$)

UNIVERSITÉ DE NANTES

Situation de compétition pour l'algo 1

1. Processus 1 récupère la variable *in* (valeur 7)
2. Processus 1 commence la copie de fic_1.pdf sur job_7
3. Commutation de processus : Processus 2 continue
4. Processus 2 récupère la variable *in* (valeur 7)
5. Processus 2 copie fic_2.c sur job_7
6. Processus 2 modifie la variable *in* (valeur 8)
7. Commutation de processus : Processus 1 continue
8. Processus 1 termine la copie de fic_1.pdf sur job_7
9. Processus 1 modifie la variable *in* (valeur 8)

$\Longrightarrow$ Une impression défectueuse assez souvent

UNIVERSITÉ DE NANTES

Algo 2 de spooling

- Deux variables partagées :
  - *out* : donnant le prochain fichier à imprimer
  - *in* : donnant le prochain numéro libre dans le répertoire

1. Récupérer la variable *in* (valeur x)
2. Modifier la variable *in* (valeur $x + 1$)
3. Copier le fichier à imprimer sur job_x

UNIVERSITÉ DE NANTES

Situation de compétition pour l'algo 2

1. Processus 1 récupère la variable *in* (valeur 7)
2. Commutation de processus : Processus 2 continue
3. Processus 2 récupère la variable *in* (valeur 7)
4. Processus 2 modifie la variable *in* (valeur 8)
5. Processus 2 copie fic_2.c sur job_7
6. Commutation de processus : Processus 1 continue
7. Processus 1 modifie la variable *in* (valeur 8)
8. Processus 1 copie fic_1.pdf sur job_7

$\Longrightarrow$Une impression défectueuse parfois : très difficile à déboguer

Section critique

- Comment éviter les situations de compétition ?
  - *Exclusion mutuelle* : interdire l'accès simultané aux ressources partagées
  - *Section critique* : interdire l'exécution simultanée du code accédant aux ressources partagées
- Conditions de coopération entre processus :
  1. Deux processus ne peuvent pas se trouver en même temps dans leur section critique
  2. Aucune hypothèse ne doit être faite sur la rapidité des processus ou le nombre de CPU
  3. Un processus à l'extérieur de sa section critique ne doit pas bloquer les autres
  4. Aucun processus ne doit attentre indéfiniment pour entrer dans sa section critique

Section critique avec attente active

Nous supposons une mémoire commune aux processus (cas des threads ou de l'espace mémoire en mode noyau)

- Interdire les interruptions matérielles en entrant dans la section critique et les autoriser à la sortie :
  - Mauvaise solution pour un programme utilisateur
  - Ne marche pas si plusieurs CPU
  - Solution utilisée parfois dans le noyau du système d'exploitation pour des accès de courte durée

Section critique avec attente active

- Variable de vérrouillage : vaut 0 si aucun processus dans sa section critique et 1 sinon :
  - Mauvaise solution si l'accès à la variable n'est pas protégé : le problème passe de l'accès à la ressource à l'accès à la variable de vérrouillage !

Section critique avec attente active

- Altérnance de l'accès : une variable `tour` indique quel processus peut entrer dans la section critique :

```
/* Processus 1 */
while(TRUE) {
  while(tour != 1);
  section_critique();
  tour = 2;
  suite();
}
```

```
/* Processus 2 */
while(TRUE) {
  while(tour != 2);
  section_critique();
  tour = 1;
  suite();
}
```

  - Mauvaise solution si les processus sont en nombre quelconque ou fortement occupés à l'extérieur de la section critique

Section critique avec attente active

- Solution de T. Dekker (1965) : plusieurs variables de vérrouillage : solution valide mais complexe à mettre en oeuvre et, pour cette raison, jamais utilisée
- Solution de Peterson (1981), ici pour 2 processus :

```
int tour;
int intéressé[2];
section_critique(int i) { /* numéro du processus */
  int autre = 1 - i;
  intéressé[i] = TRUE;
  tour = i;
  while(tour == i && intéressé[autre] == TRUE);
  ... /* la section critique */
  intéressé[i] =  FALSE;
}
```

  - Demande de connaître le nombre maximum de processus intéressés et d'assigner un numéro à chacun

Section critique avec attente active

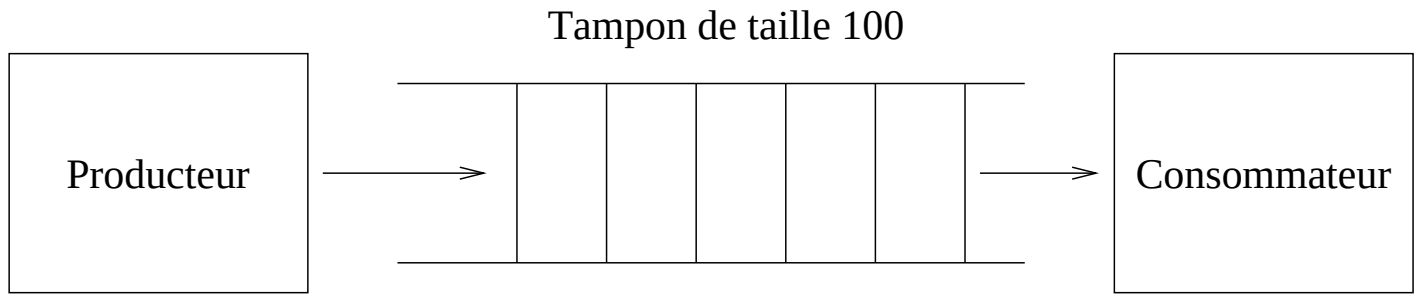
- Instruction Test and Set Lock (TSL) : teste la case mémoire `drapeau` et la met à 1 en une opération assembleur indivisible

```
section_critique:
    tsl registre, drapeau
    cmp registe, #0
    jnz section_critique
    ... /* la section critique */
    mov drapeau, #0
```

- Sur les processeurs de type Pentium, on utilise le préfixe d'instruction `LOCK` qui verrouille le bus pendant toute l'exécution de l'instruction

Sleep et Wakeup

- Les solutions de Peterson ou utilisant l'instruction TSL sont correctes mais utilisent une boucle pour l'attente $\Longrightarrow$mauvaise utilisation du CPU
- `sleep()` : endort le processus courant
- `wakeup(pid)` : réveille le processus `pid`
- Utilisable dans le cas du problème des producteurs-consommateurs :



Sleep et Wakeup

- (Mauvaise) solution avec une situation de compétition :

```
producteur() {
  while(TRUE) {
    produire_un_élément();
    if (taille_tampon == 100) sleep();
    ajouter_l_élément();
    taille_tampon += 1;
    if (taille_tampon == 1) wakeup(consommateur);
  }
}

consommateur() {
  while(TRUE) {
    if (taille_tampon == 0) sleep();
    enlever_un_élément();
    taille_tampon -= 1;
    if (taille_tampon = 99) wakeup(producteur);
    consommer_l_élément();
  }
}
```

Sleep et Wakeup

- (Mauvaise) solution avec une situation de compétition :
  1. `taille_tampon` est nul
  2. Le consommateur teste `taille_tampon == 0`
  3. Commutation de processus
  4. Le producteur ajoute un élément, incrémente `taille_tampon` et appelle `réveiller(consommateur)` : le consommateur n'est pas encore endormi $\Longrightarrow$rien ne se passe
  5. Commutation de processus
  6. Le consommateur s'endort
  7. Commutation de processus
  8. Le producteur remplit le tampon et s'endort

UNIVERSITÉ DE NANTES

Sémaphores

- Solution de E. Dijkstra (1965) : utilise des *sémaphores* :
  - `down(sémaphore)` : si le sémaphore a une valeur > 0, l'opération le décrémente et continue ; si le sémaphore = 0, le processus est endormi
  - `up(sémaphore)` : incrémente le sémaphore ; si au moins un processus a été endormi par un `down(sémaphore)`, réveille un des processus pour qu'il finisse l'opération `down`

Ces opérations doivent être atomiques et utilisent une file d'attente des processus endormis sur un sémaphore particulier $\Longrightarrow$gérée, en général, par le système d'exploitation

Sémaphores

- Mise en oeuvre sous UNIX: librairie IPC sur les sémaphores
  - `semget()` pour créer un nouveau sémaphore (ou récupérer une référence sur un sémaphore déjà existant à l'aide d'une clé)
  - `semop()` pour effectuer des opérations sur un ensemble de sémaphores ("down" ou "up")
- Utilisable sur les processus (pas que sur les threads)

Solution du problème des P/C

Il faudrait coder les up et down avec des sémaphores IPC :

```
sémaphore mutex = 1;
sémaphore emplacements_vides = 100;
sémaphore emplacements_occupés = 0;

producteur() {
  while(TRUE) {
    produire_un_élément();
    down(emplacements_vides);
    down(mutex);
    ajouter_l_élément();
    up(mutex);
    up(emplacements_occupés);

  }
}
```

```
consommateur() {
  while(TRUE) {

    down(emplacements_occupés);
    down(mutex);
    enlever_un_élément();
    up(mutex);
    up(emplacements_vides);
    consommer_l_élément();
```

Solution du problème des P/C

Mauvaise solution :

```
sémaphore mutex = 1;
sémaphore emplacements_vides = 100;
sémaphore emplacements_occupés = 0;

producteur() {                          consommateur() {
  while(TRUE) {                           while(TRUE) {
    produire_un_élément();
    down(emplacements_vides);               down(mutex);
    down(mutex);                            down(emplacements_occupés);
    ajouter_l_élément();                    enlever_un_élément();
    up(mutex);                              up(emplacements_vides);
    up(emplacements_occupés);               up(mutex);
                                            consommer_l_élément();
  }
}
```

Moniteurs

- Hoare (1974) and Brinch Hansen (1975) ont proposé une méthode de protection de haut niveau : les collections de fonctions synchronisées appelées moniteurs
- Constructions spécifiques à un langage
- Seul un processus peut exécuter une fonction du moniteur à la fois

```
monitor exemple
  integer i;
  procedure producteur(x)
     ...
  end;
  procedure consommateur(x)
     ...
  end;
end moniteur;
```

UNIVERSITÉ DE NANTES

Moniteurs Java

- Les moniteurs sont la base de la synchronisation en Java
- Un verrou par classe (méthodes de classe)
- Un verrou par objet (méthodes d'instance et construction `synchronized`)
- Synchronistation des méthodes avec le modifieur de méthodes `synchronized`
- Synchronistation d'un bloc d'instructions avec la construction `synchronized`

UNIVERSITÉ DE NANTES

Moniteurs Java : verrou sur une classe

```
public class VariableProtégée {
  static int variable_globale;
  public static synchronized void plus() {
      variable_globale++;
  };
  public static synchronized int val() {
      return variable_globale;
  };
}
...
// Pas de situation de concurrence
// Les opérations plus() et
// val() sont atomiques
VariableProtégée.plus();
int v = VariableProtégée.val();
System.out.println(v);
```

Moniteurs Java : verrou sur une classe

```
public class Classe {
   ...
   public static synchronized type1 méthode1(...) {
      ...
   }
   public static synchronized type2 méthode2(...) {
      ...
   }
   ...
}
```

- Seul un thread peut appeler les méthodes `méthode1`, `méthode2`,... à la fois
- Les méthodes peuvent s'appeler les unes les autres dans un thread

Moniteurs Java : verrou sur une classe

```
class Variable1 {
    static int variable_globale;
    public static synchronized void plus() {
        System.out.println("Début de Variable1.plus() : "+val());
        variable_globale++;
        try { Thread.sleep(1000); } catch (InterruptedException e) {};
        System.out.println("Fin de Variable1.plus() : "+val());
    };
    public static synchronized int val() { return variable_globale; };
}
public class Clone1 extends Thread {
  public void run() {
    Variable1.plus();
    int v = Variable1.val();
    System.out.println("Fin de Clone1.run() : " + v);
  };
  public static void main(String argv[]) {
    new Clone1().start();
    new Clone1().start();
  };
}
```

Moniteurs Java : verrou sur une classe

Exécution de `java Clone1` :

```
Début de Variable1.plus() : 0
Fin de Variable1.plus() : 1
Fin de Clone1.run() : 1
Début de Variable1.plus() : 1
Fin de Variable1.plus() : 2
Fin de Clone1.run() : 2
```

Moniteurs Java : verrou sur un objet

```
public class VariableProtégée {
  int variable;
  public synchronized void plus() {
      variable++;
  };
  public synchronized int val() {
      return variable;
  };
}
...
// Pas de situation de concurrence
// Les méthodes plus() et
// val() sont atomiques sur un objet
VariableProtégée obj = new VariableProtégée();
obj.plus();
int v = obj.val();
System.out.println(v);
```

Moniteurs Java : verrou sur un objet

```
public class Classe {
   ...
   public synchronized type1 méthode1(...) {
      ...
   }
   public synchronized type2 méthode2(...) {
      ...
   }
   ...
}
```

- Seul un thread peut appeler les méthodes `méthode1`, `méthode2`,... d'un objet particulier à la fois
- Les méthodes peuvent s'appeler les unes les autres dans un thread
- Les méthodes peuvent s'appeler les unes les autres si cela ne concerne pas le même objet

Moniteurs Java : verrou sur deux objets

```
class Variable2 {
    int variable;
    public synchronized void plus() {
        System.out.println("Début de Variable2.plus() : "+val());
        variable++;
        try { Thread.sleep(1000); } catch (InterruptedException e) {};
        System.out.println("Fin de Variable2.plus() : "+val());
    };
    public synchronized int val() { return variable; };
}
public class Clone2 extends Thread {
  Variable2 obj;
  public Clone2(Variable2 o) { obj = o; };
  public void run() {
    obj.plus(); int v = obj.val();
    System.out.println("Fin de Clone2.run() : " + v);
  };
  public static void main(String argv[]) {
    new Clone2(new Variable2()).start();
    new Clone2(new Variable2()).start();
  };
}
```

Moniteurs Java : verrou sur deux objets

Exécution de `java Clone2` :

```
Début de Variable2.plus() : 0
Début de Variable2.plus() : 0
Fin de Variable2.plus() : 1
Fin de Clone2.run() : 1
Fin de Variable2.plus() : 1
Fin de Clone2.run() : 1
```

Moniteurs Java : verrou sur un objet

```
class Variable3 {
    int variable;
    public synchronized void plus() {
        System.out.println("Début de Variable3.plus() : "+val());
        variable++;
        try { Thread.sleep(1000); } catch (InterruptedException e) {};
        System.out.println("Fin de Variable3.plus() : "+val());
    };
    public synchronized int val() { return variable; };
}
public class Clone3 extends Thread {
  Variable3 obj;
  public Clone3(Variable3 o) { obj = o; };
  public void run() {
    obj.plus(); int v = obj.val();
    System.out.println("Fin de Clone3.run() : " + v);
  };
  public static void main(String argv[]) {
    Variable3 obj = new Variable3();
    new Clone3(obj).start(); new Clone3(obj).start();
  };
}
```

Moniteurs Java : verrou sur un objet

Exécution de `java Clone3` :

```
Début de Variable3.plus() : 0
Fin de Variable3.plus() : 1
Fin de Clone3.run() : 1
Début de Variable3.plus() : 1
Fin de Variable3.plus() : 2
Fin de Clone3.run() : 2
```

Moniteurs Java : verrouillage externe

```
public class VariableProtégée {
  public int variable;
}
...
// Pour incrémenter variable :
synchronized(obj) {
      obj.variable++;
};
// Pour récupérer variable :
int v;
synchronized(obj) {
      v = obj.variable;
};
```

Sleep et wakeup avec Java : wait et notify

- `obj.wait()` : endort le thread courant sur l'objet `obj`. Le thread doit posséder un verrou `synchronized` sur `obj`. L'appel libère le verrou sur `obj`
- `obj.notify()` : réveille un des threads endormis sur l'objet `obj`. Le thread doit posséder un verrou `synchronized` sur `obj`. L'appel libère le verrou sur `obj`. La fin de la méthode correspond à une nouvelle demande de verrouillage sur `obj`
- Variantes:
  - `obj.wait(long timeout)` : comme `obj.wait()` mais réveille le thread au bout de `timeout` millisecondes s'il dort encore
  - `obj.notifyAll()` : réveille tous les threads endormis sur l'objet `obj` au lieu d'un seul

Wait et notify : exemple

```
public class Clone4 extends Thread {
  static Object verrou = new Object();
  public void run() {
    while (true) {
      System.out.println(getName());
      synchronized(verrou) {
        try { verrou.wait(); } catch (InterruptedException e) {};
      }
    }
  };
  public static void main(String argv[]) {
    new Clone4().start();
    new Clone4().start();
    new Clone4().start();
    while(true) {
      synchronized(verrou) { verrou.notify(); }
      try { sleep(1000);  } catch (InterruptedException e) {};
    }
  };
}
```

Wait et notify : exemple

Exécution de `java Clone4` :

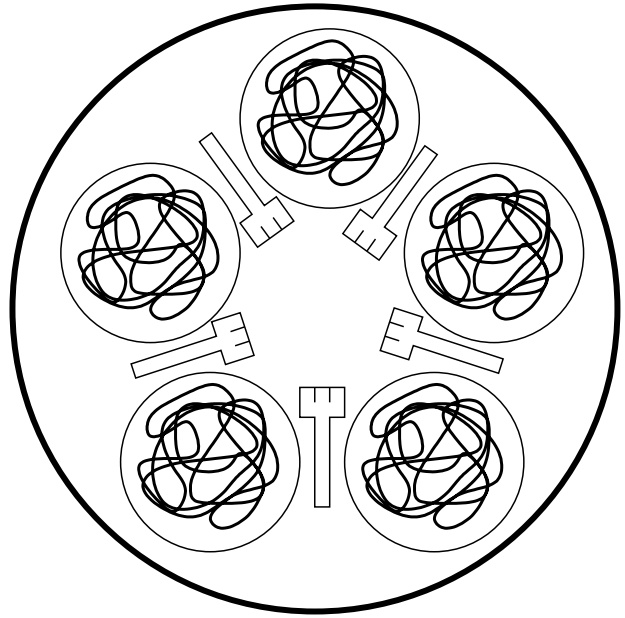
```
Thread-0
Thread-2
Thread-1
Thread-0
Thread-2
Thread-1
Thread-0
Thread-2
Thread-1
Thread-0
...
```

Problèmes classiques de synchronisation

- Problème du diners des philosophes : problème lié à l'accès à plusieurs ressources
- Problème des lecteurs/écrivains : accès simultané à une ressource par plusieurs lecteurs ou bien par un seul écrivain

Problème du diners des philosophes

- 5 philosophes, 5 plats de spaghettis et 5 fourchettes
- Les philosophes réfléchissent ou mangent
- Pour manger, il faut la fourchette à sa gauche et la fouchette à sa droite
- Les philosophes ne doivent pas mourrir de faim



Mauvaise solution pour le diner

```
philosophe() {
  while(true) {
    réfléchir();
    prendre_la_fourchette_à_sa_gauche();
    prendre_la_fourchette_à_sa_droite();
    manger();
    replacer_la_fourchette_à_sa_gauche();
    replacer_la_fourchette_à_sa_droite();
}
```

Situation de disette pour tous les philosophes:

1. Ils décident de manger au même moment
2. Ils commencent par récupérer tous la fourchette se trouvant à leur gauche
3. Ils sont bloqués en essayant de récupérer la fourchette à droite (il n'y a plus de fourchette sur la table)

UNIVERSITÉ DE NANTES

Problème des lecteurs et des écrivains

- Processus lecteurs : les processus qui veulent uniquement lire une ressource
- Processus écrivains : les processus qui veulent modifier cette ressource
- Conditions d'utilisation de la ressource
  1. Seul un écrivain peut modifier la ressource à la fois
  2. Pendant qu'un écrivain modifie la ressource, les lecteurs ne sont pas autorisés à la lire

Problème courant avec les bases de données (fichier `/etc/passwd` par exemple)

Solution possible pour les L/E

Ici, priorité des lecteurs sur les écrivains :

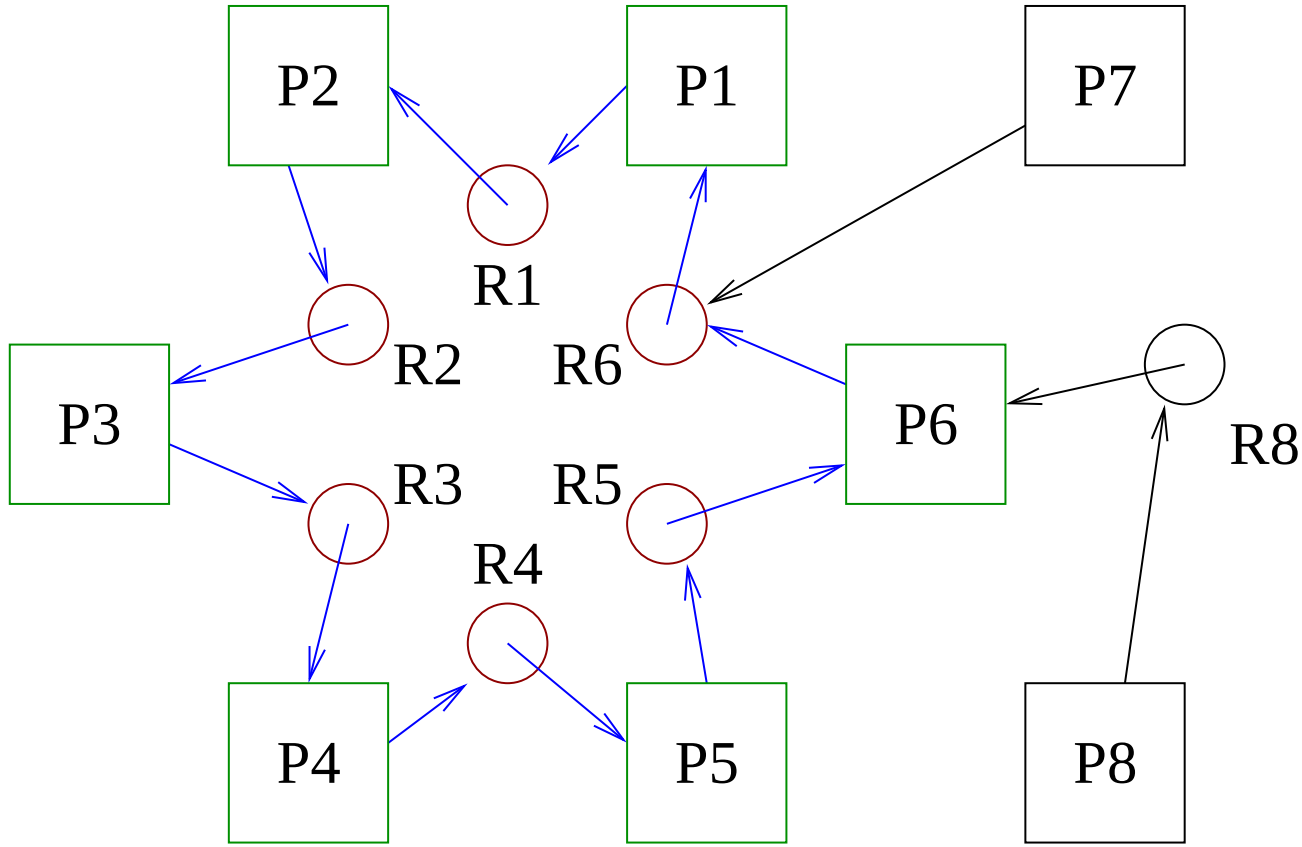
```
sémaphore mutex; /* = 1 : accès à nl */
sémaphore bd;    /* = 1 : contrôle d'accès à la base */
int nl = 0;      /* nombre de lecteurs */

lecteur() {
  while(true) {
    down(mutex)
    nl += 1;
    if (nl == 1) down(bd);
    up(mutex);
    lire_la_base();
    down(mutex)
    nl -= 1;
    if (nl == 0) up(bd);
    up(mutex);
    utiliser_les_données();
  }
}
```

```
ecrivain() {
  while(true) {
    préparer_les_données();
    down(bd);
    modifier_la_base();
    up(bd)
  }
}
```

Classification des problèmes

1. Situation de compétition : lorsque au moins deux processus accèdent en même temps à une ressource partagée $\Longrightarrow$mauvaise synchonisation
2. Interblocage : lorsque deux ou plus de deux processus se bloquent mutuellement l'accès aux ressources $\Longrightarrow$blocage du système. Voir l'exemple des philosophes
3. Famine : au moins un processus n'accède jamais à la ressource qu'il convoite $\Longrightarrow$absence d'équité (exemple des écrivains si un pool de lecteurs ne relache jamais la ressource dans l'exemple précédent)

Interblocages



Un cycle alterné ressources/processus

Prévention des interblocages

1. Modélisation des processus : démontrer l'absence d'interblocage
   - Réseaux de Petri
   - Diagrammes temporelles
2. Prévention : empécher la création des cycles
   - Ordre strict sur les sémaphores
   - Opération down() unique en parallèle sur plusieurs sémaphores
3. Détection : supprimer un interblocage
   - Suppression de processus
4. Politique de l'autruche !!!
   - Utilisé souvent les SE,... pour la gestion de l'espace mémoire et l'espace du disque

UNIVERSITÉ DE NANTES

Les réseaux de Petri

- Un réseau de Petri (RdP) est un graphe biparti comportant des places et des transitions
  - Une place est représentée par un cercle
  - Une transition est représentée par un trait
  - Les places et les transitions sont reliés par des arcs orientés
  - Un arc relie soit une place à une transition, soit une transition à une place
  - Le nombre de places et de transitions est fini, non nul

Les réseaux de Petri

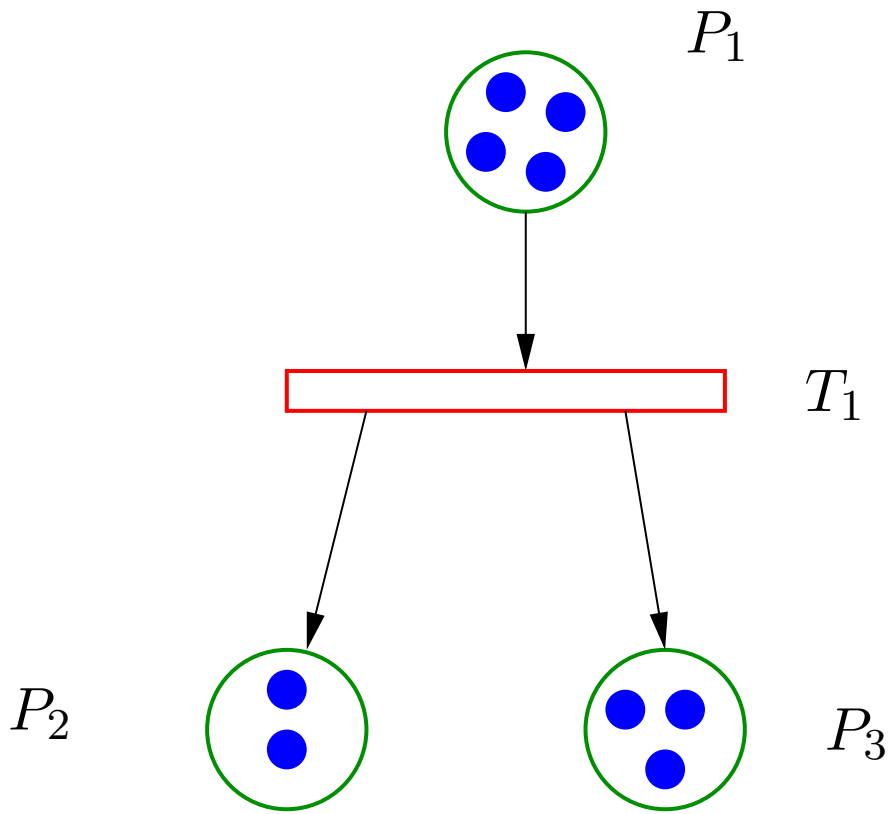
- Chaque place peut contenir un nombre entier (positif ou nul) de jetons ou marques
- Le nombre de marques de la place i sera noté $M(P_i)$ ou M_i.
- Le franchissement d'une transition ne peut s'effectuer que si chacune des places en amont de cette transition est validée ou franchissable, c'est à dire contient au moins une marque

UNIVERSITÉ DE NANTES

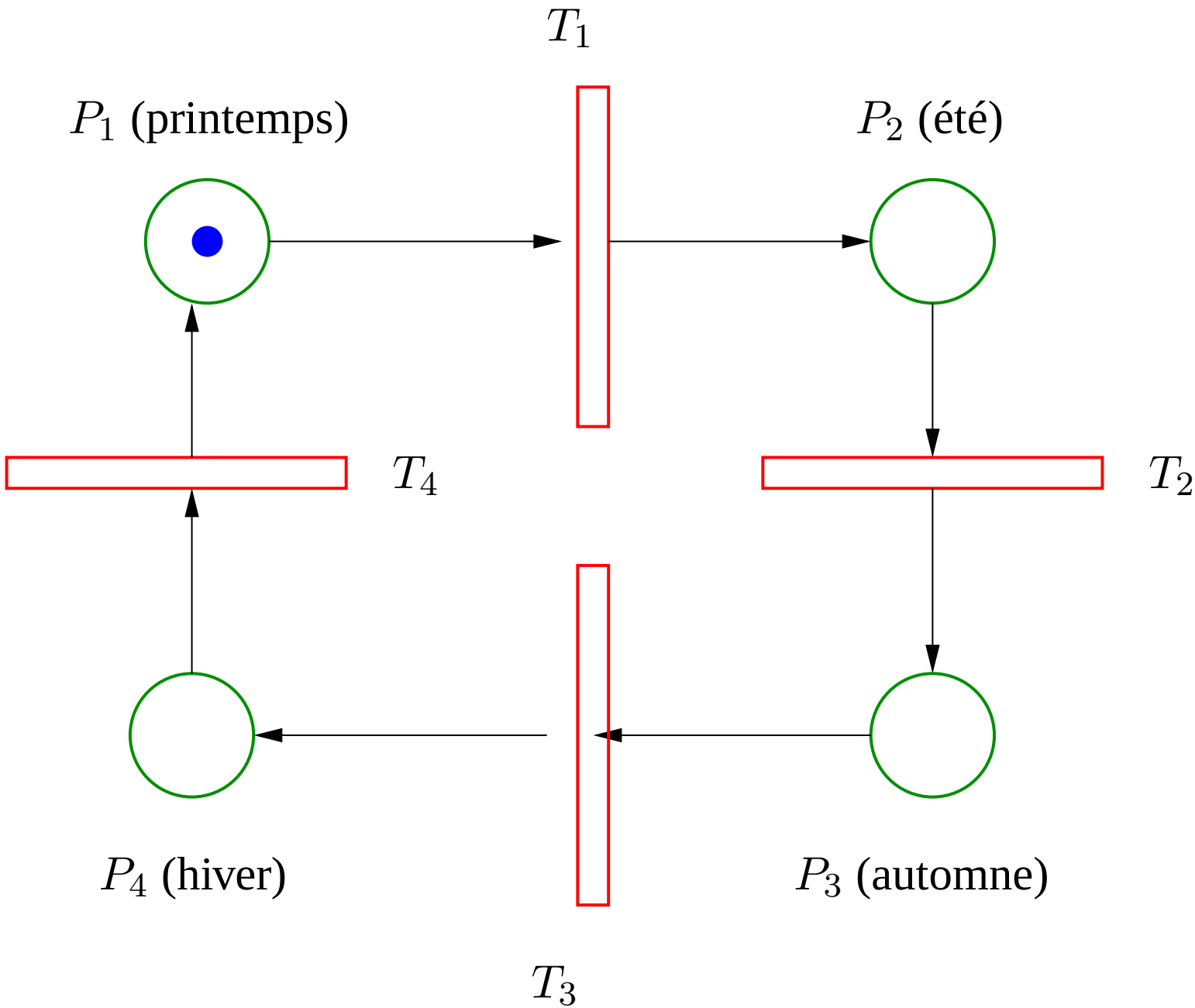
Les réseaux de Petri

- Une transition sans place amont est toujours validée : c'est une transition source
- Le franchissement (ou tir) d'une transition T consiste à retirer une marque dans chacune des places amont et à ajouter une marque dans chacune des places aval de la transition T.
- Parmi plusieurs transitions franchissables, une seule est franchie à la fois.
- Le franchissement d'une transition est considéré comme une opération atomique (indivisible).

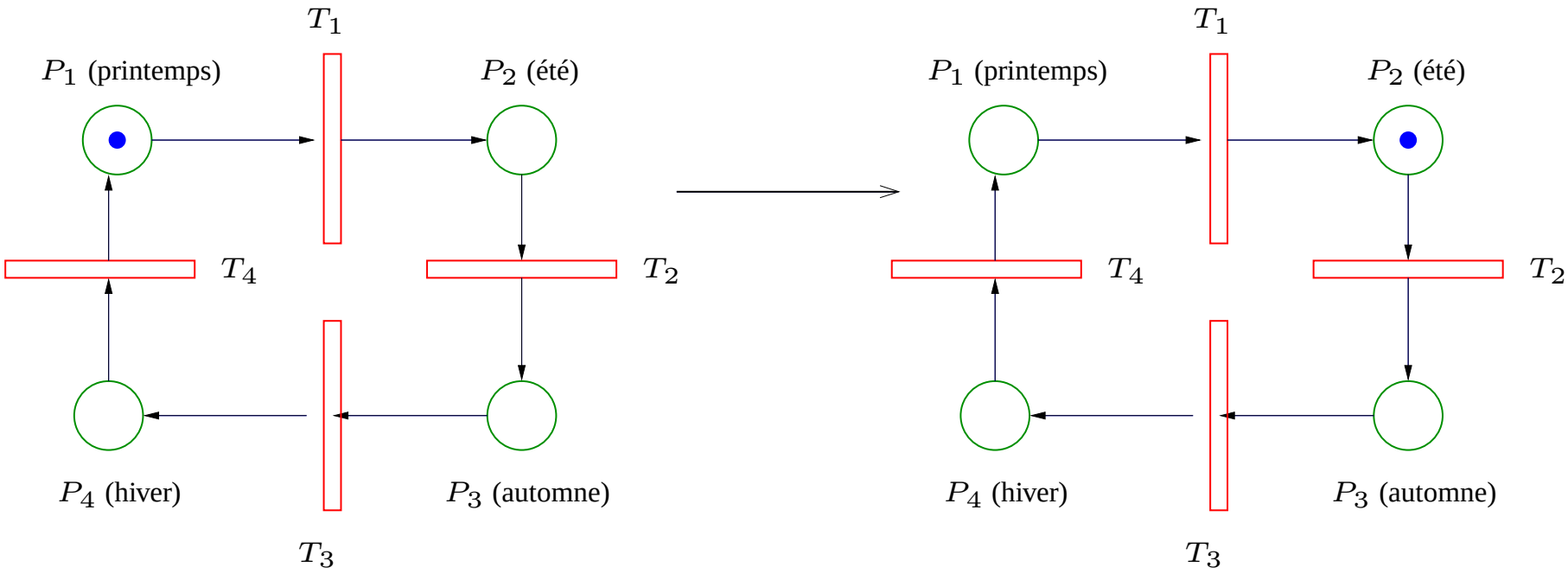
marquage = (4,2,3)



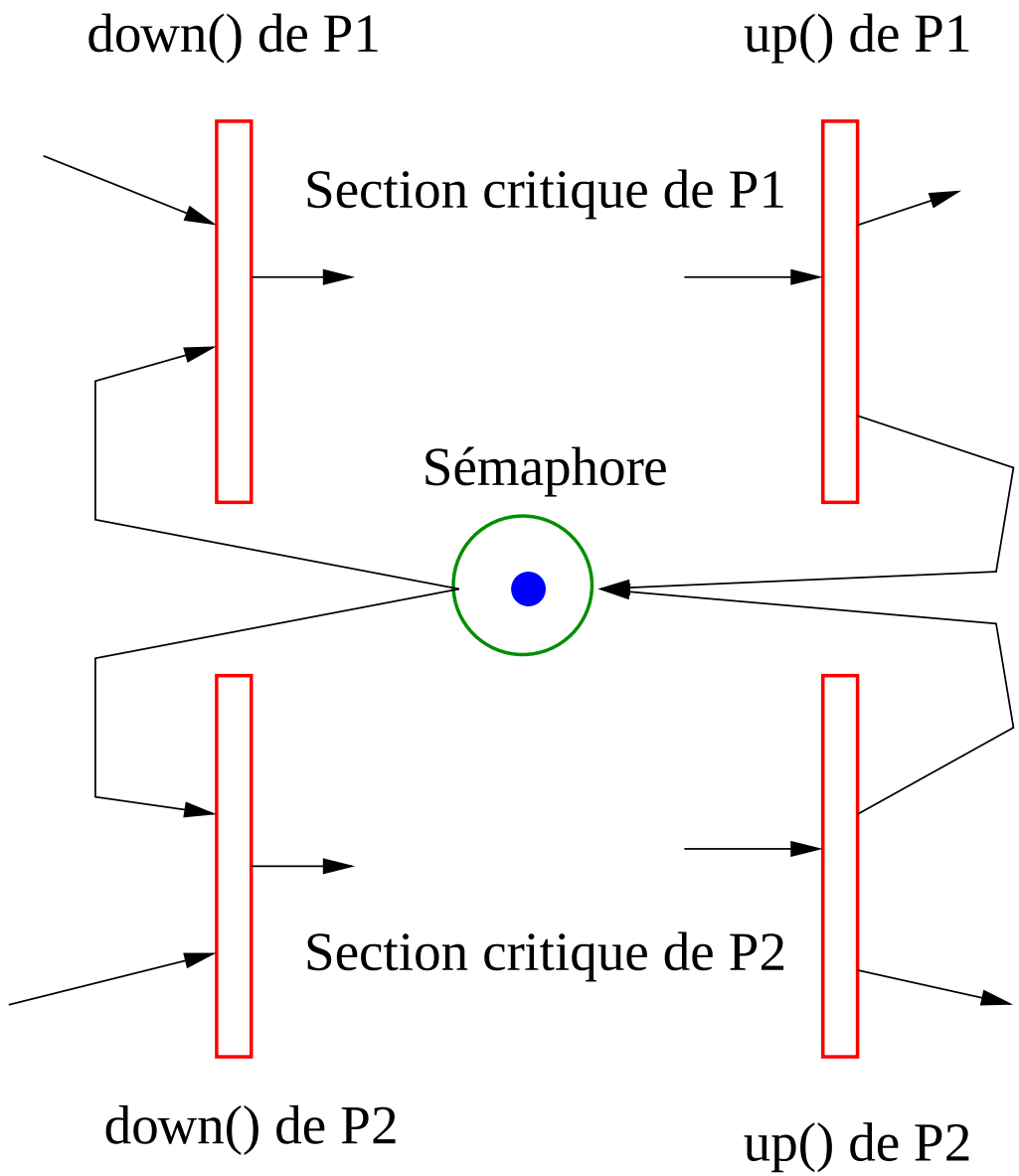
RdP



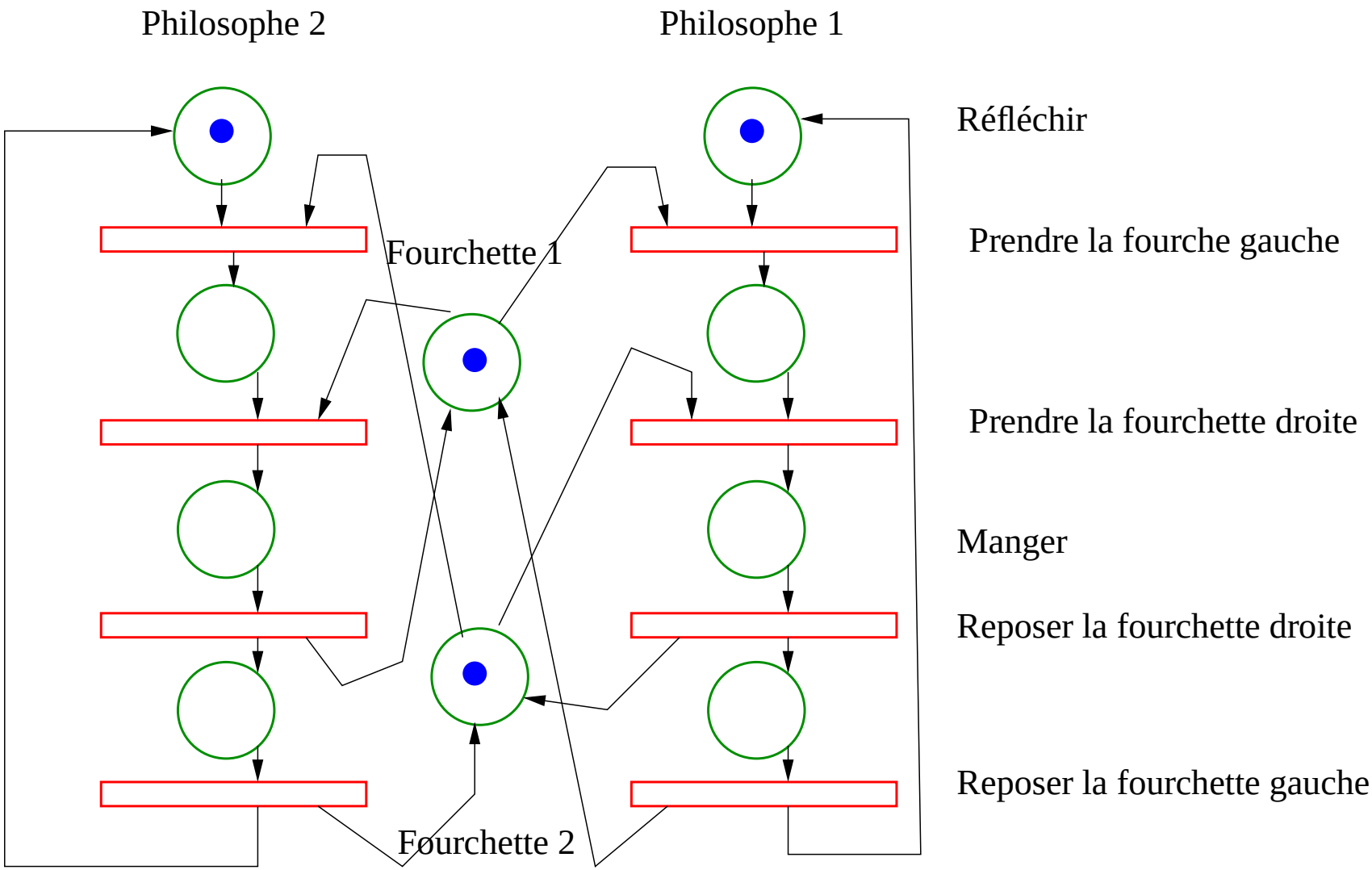
Transition


T_1
P_1 (printemps)
P_2 (été)
T_4
T_2
P_4 (hiver)
P_3 (automne)
T_3
T_1
P_1 (printemps)
P_2 (été)
T_4
T_2
P_4 (hiver)
P_3 (automne)
T_3

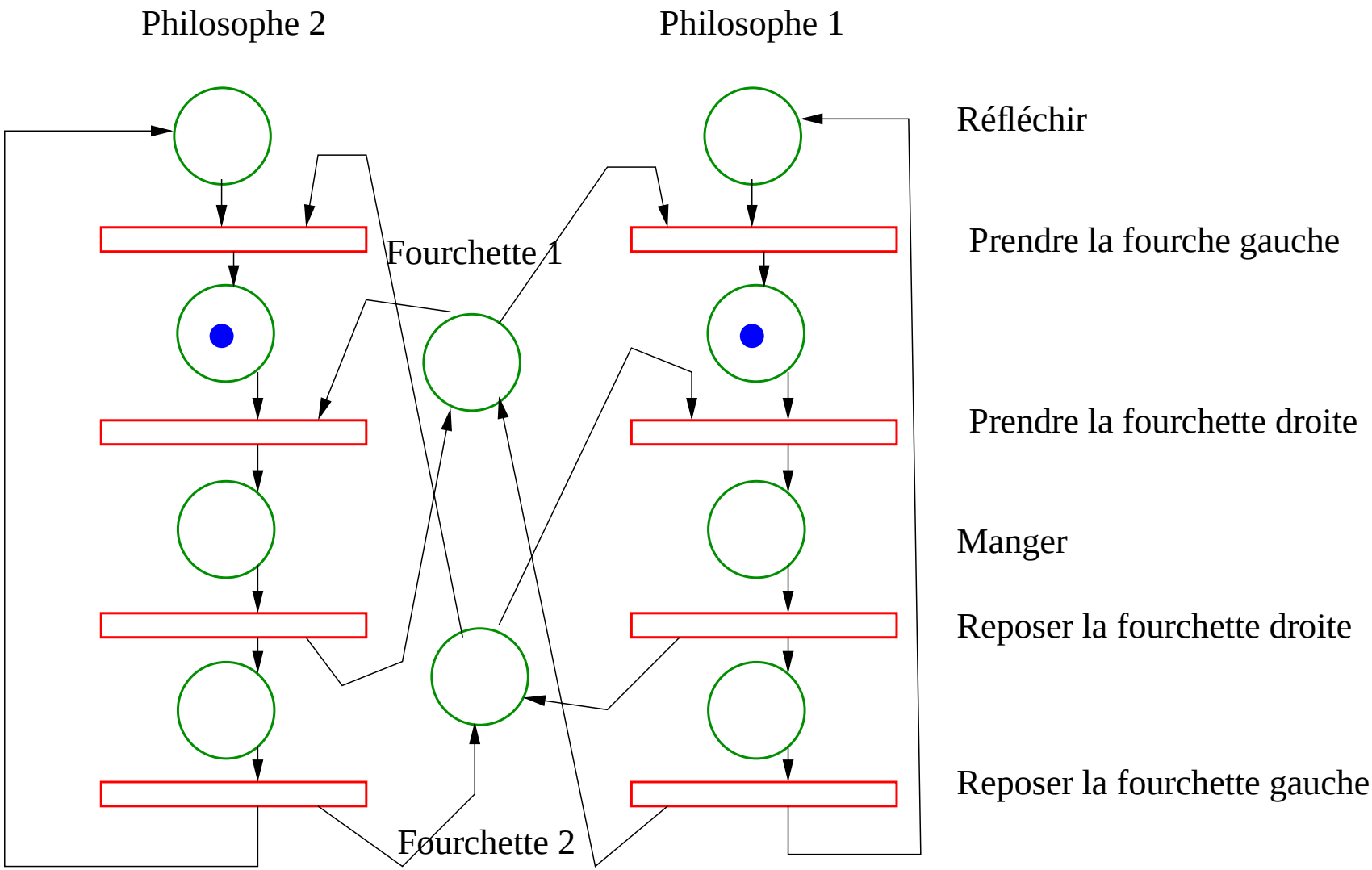
Modélisation d'un sémaphore



Exemple : les philosophes



Interblocage des les philosophes



Interblocage des les philosophes

- Démonstration de la correction d'un algorithme = preuve mathématique de non-accessibilité des configurations correspondant aux interblocages
- Démonstration de la non-correction d'un algorithme = preuve mathématique de l'accessibilité d'une configuration correspondant à un interblocage

Envoie de message

- Communication par message (et boîte aux lettres)
  - Pas besoin de sémaphore (sauf si une boîte aux lettres est partagée par plusieurs processus)
  - Demande de connaître l'adresse du destinataire
  - Base de la programmation par événements (interface graphique, USB,...)
  - Base de la communication entre ordinateurs (ethernet, réseaux locaux, internet (TCP/IP), sockets, architecture client/serveur)
- `send()` : pour envoyer un message à un destinataire
- `receive()` : pour recevoir/récupérer un message depuis une source (ou un ensemble de sources)

UNIVERSITÉ DE NANTES

Systèmes de messagerie

Exemples:

- Fonctions IPC : messagerie par files d'attente de messages utilisables par plusieurs processus
- Interface `BlockingQueue` : pour créer des files d'attentes synchronisées utilisables par les threads d'une même tache Java
- Tube (pipe) : Un canal de communication (unidirectionnel) entre processus : le " | " des shells
- Socket : Un canal de communication (bidirectionnel) entre processus à travers un réseau
- ...

UNIVERSITÉ DE NANTES

Fonctions IPC

- `int msgget(key_t key, int flg)` : permet de créer une nouvelle file de message ou de récupérer l'identificateur d'une file déjà existante à partir d'une clé (`flg` est souvent `IPC_CREAT` + les droits d'accès)
- `msgctl(...)` pour modifier ou supprimer une file
- Commande Linux `ipcs -q` : info sur les files
- Commande Linux `ipcrm -Q clé` : pour supprimer la file de clé `clé`

UNIVERSITÉ DE NANTES

Fonctions IPC

- `int msgsnd(int id, struct msgbuf * msgp, size_t sz, int flg)` pour envoyer un message constitué d'un type (un entier long positif) et d'une suite d'octets de longueur `sz` sur une file
- `ssize msgrcv(int id, struct msgbuf * msgp, size_t sz, long typ, int flg)` pour récupérer le premier message d'un type donné (ou de tous les types si `typ` = 0)
- Structure des messages:

```
struct msgbuf {
  long  mtype;    // type de message ( >0 )
  char  mtext[1]; // contenu du message
};
```

Un producteur de messages

```
#include ...
struct mymsgbuf {
    long  mtype;       /* type du message ( > 0 ) */
    char  mtext[100];  /* contenu du message      */
} message;
#define CLE 100012

int main(int argc, char* argv[]) {
    int i;
    int id = msgget( CLE, IPC_CREAT | 0777);
    if (id == -1) { perror("Création de la file : ");  exit(-1); }
    for(i = 1; i < argc ; i++) {
        printf("[%d] envoie du message %d : <%s>\n",
                                          getpid(), i, argv[i]);
        message.mtype = i;
        strncpy(message.mtext, argv[i], 99);
        msgsnd(id, (struct msgbuf *) &message, 100, 0);
        printf("[%d] fin de l'envoie du message %d : <%s>\n",
                                          getpid(), i, argv[i]);
        sleep(2);
    }
}
```

Un consommateur de messages de type 1

```
#include ...
struct mymsgbuf {
    long  mtype;       /* type du message ( > 0 ) */
    char  mtext[100];  /* contenu du message      */
} message;
#define CLE 100012

int main(int argc, char* argv[]) {
    int i;
    int id = msgget( CLE, IPC_CREAT | 0777);
    if (id == -1) { perror("Création de la file : ");  exit(-1); }
    printf("[%d] réception des messages de type 1\n", getpid());
    while(1) {
        int i = msgrcv(id, (struct msgbuf *) &message, 100, 1, 0);
        if (i == -1)
            perror("Erreur de msgrcv :");
        else {
            printf("[%d] réception du message %d : <%s>\n",
                  getpid(), message.mtype, message.mtext);
        }
    }
}
```

Un consommateur de tous les messages

```
#include ...
struct mymsgbuf {
    long  mtype;       /* type du message ( > 0 ) */
    char  mtext[100];  /* contenu du message      */
} message;
#define CLE 100012

int main(int argc, char* argv[]) {
    int i;
    int id = msgget( CLE, IPC_CREAT | 0777);
    if (id == -1) { perror("Création de la file : ");  exit(-1); }
    printf("[%d] réception de tous les messages\n", getpid());
    while(1) {
        int i = msgrcv(id, (struct msgbuf *) &message, 100, 0, 0);
        if (i == -1)
            perror("Erreur de msgrcv :");
        else {
            printf("[%d] : réception du message %d : <%s>\n",
                        getpid(), message.mtype, message.mtext);
        }
    }
}
```

Exemple d’exécution

```
> prod_msg Un message
[26589] envoie du message 1 : <Un>
[26589] fin de l’envoie du message 1 : <Un>
[26589] envoie du message 2 : <message>
[26589] fin de l’envoie du message 2 : <message>
> conso_1_msg
[26605] réception des messages de type 1
[26605] réception du message 1 : <Un>
^C
> ipcs -q
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x000186ac 163840     bechet     777        100          1
> conso_all_msg
[26662] réception de tous les messages
[26662] : réception du message 2 : <message>
^C
> ipcrm -Q 100012
> ipcs -q
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
>
```

Fonctions IPC

- `key_t ftok(char * pathname, int projet)` : pour créer une clé relativement unique à partir du nom d'un fichier et des 8 premier bits de `projet`

```
key_t clé = ftok("/tmp/fichier", 3);
int id = msgget(clé, IPC_CREAT | 0777);
```

à la place de :

```
#define CLE 100012
int id = msgget(CLE, IPC_CREAT | 0777);
```

Partage de mémoire

- Communication par partage de mémoire
- fonctions IPC :
  - `shmat()` : attacher une zone partagée à la mémoire du processus courant
  - `shmdt()` : détacher cette zone de la mémoire du processus courant
- Projection de fichiers ou d'objets en mémoire : pour permettre à un ou plusieurs processus de "voir" directement un fichier en mémoire
  - `mmap()` : attacher un fichier à la mémoire
  - `munmap()` : détacher ce fichier de la mémoire
  - `shm_open()`, `shm_unlink()` : créer/supprimer des objets mémoires (plutôt que des fichiers)

Fonctions IPC

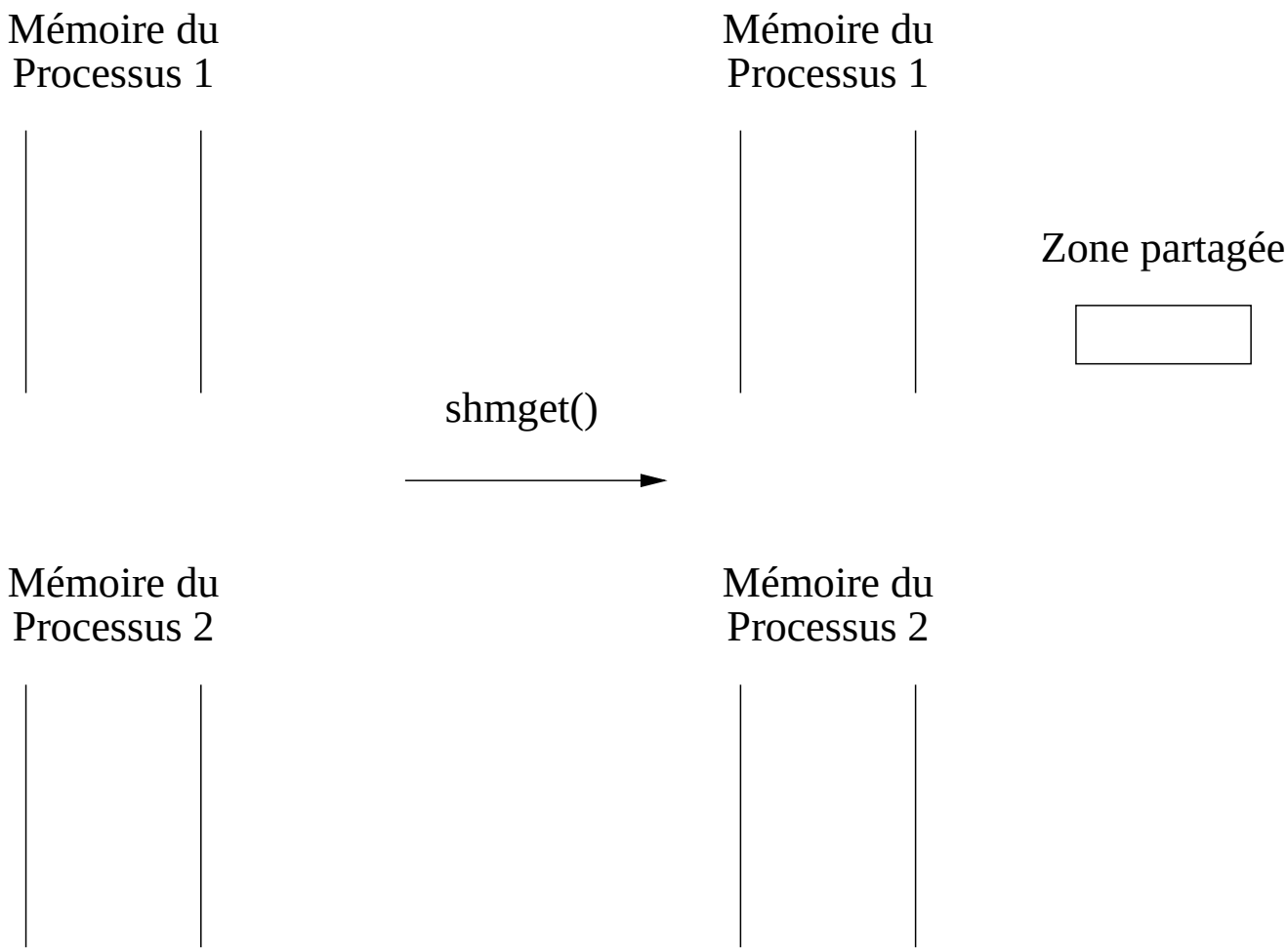
- `int shmget(key_t key, int size, int flg)` permet de créer une nouvelle zone de mémoire partagée ou de récupérer l'identificateur d'une zone déjà existante à partir d'une clé (`flg` est souvent `IPC_CREAT` + les droits d'accès)
- `shmctl(...)` pour modifier ou supprimer une zone
- Commande Linux `ipcs -m` : info sur les zones de mémoire partagée
- Commande Linux `ipcrm -M clé` : pour supprimer la zone de clé `clé`

Fonctions IPC

- `void * shmat(int id, const void * addr, int flg)` pour placer une zone à l'adresse `addr`. Si `addr` = NULL, le SE choisit une région libre de la mémoire du processus. Dans tous les cas, la valeur retournée est l'adresse de la région en mémoire
- `int shmdt(const void * addr)` détache la zone se trouvant à l'adresse `addr`
- Création/destruction de processus :
  - `fork()` : les régions sont partagées par le père et le fils
  - `execve()` et `exit()`: les régions sont détachées (mais pas supprimées)

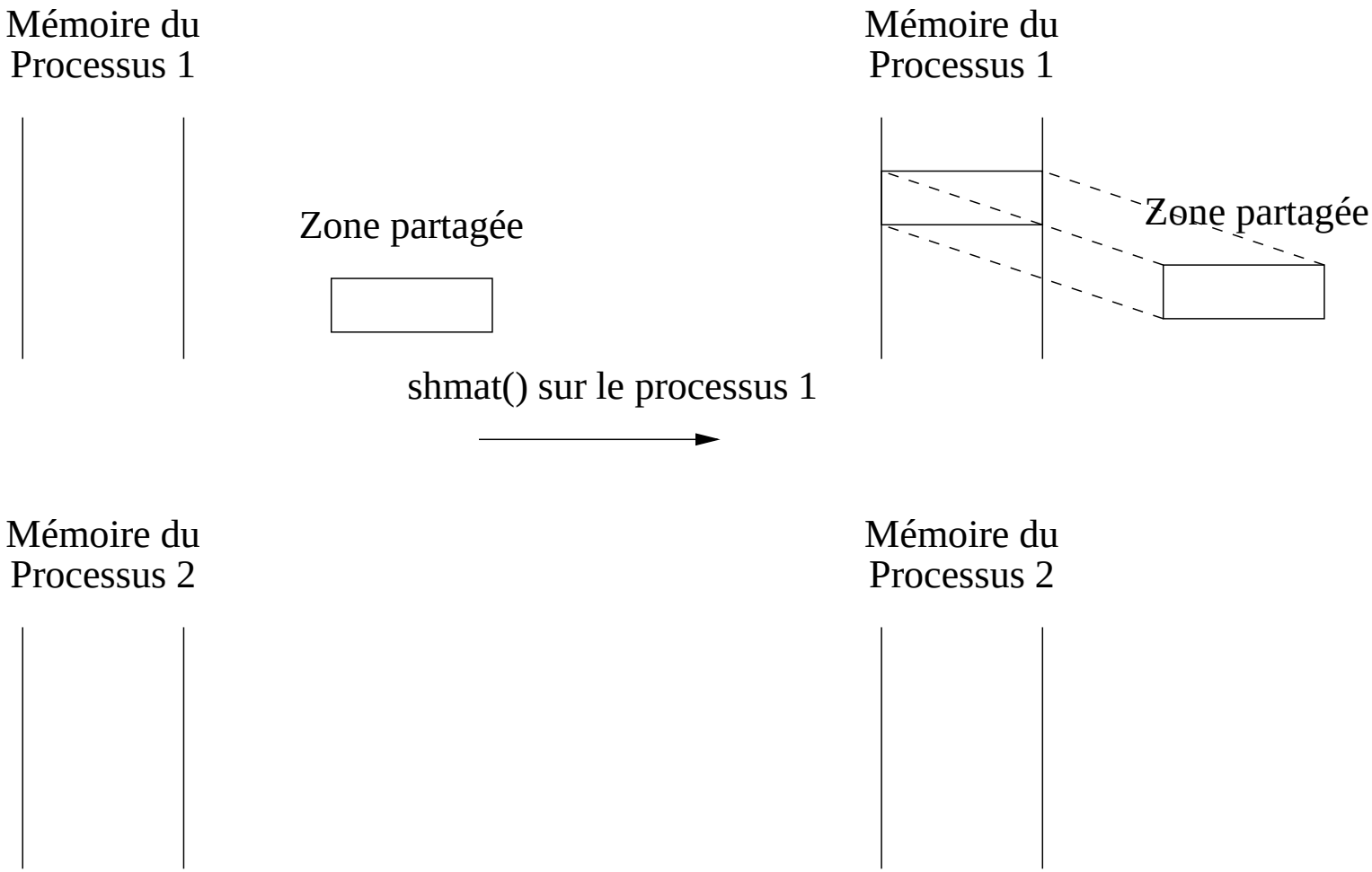
Fonctions IPC

- Création d'une zone de mémoire partagée



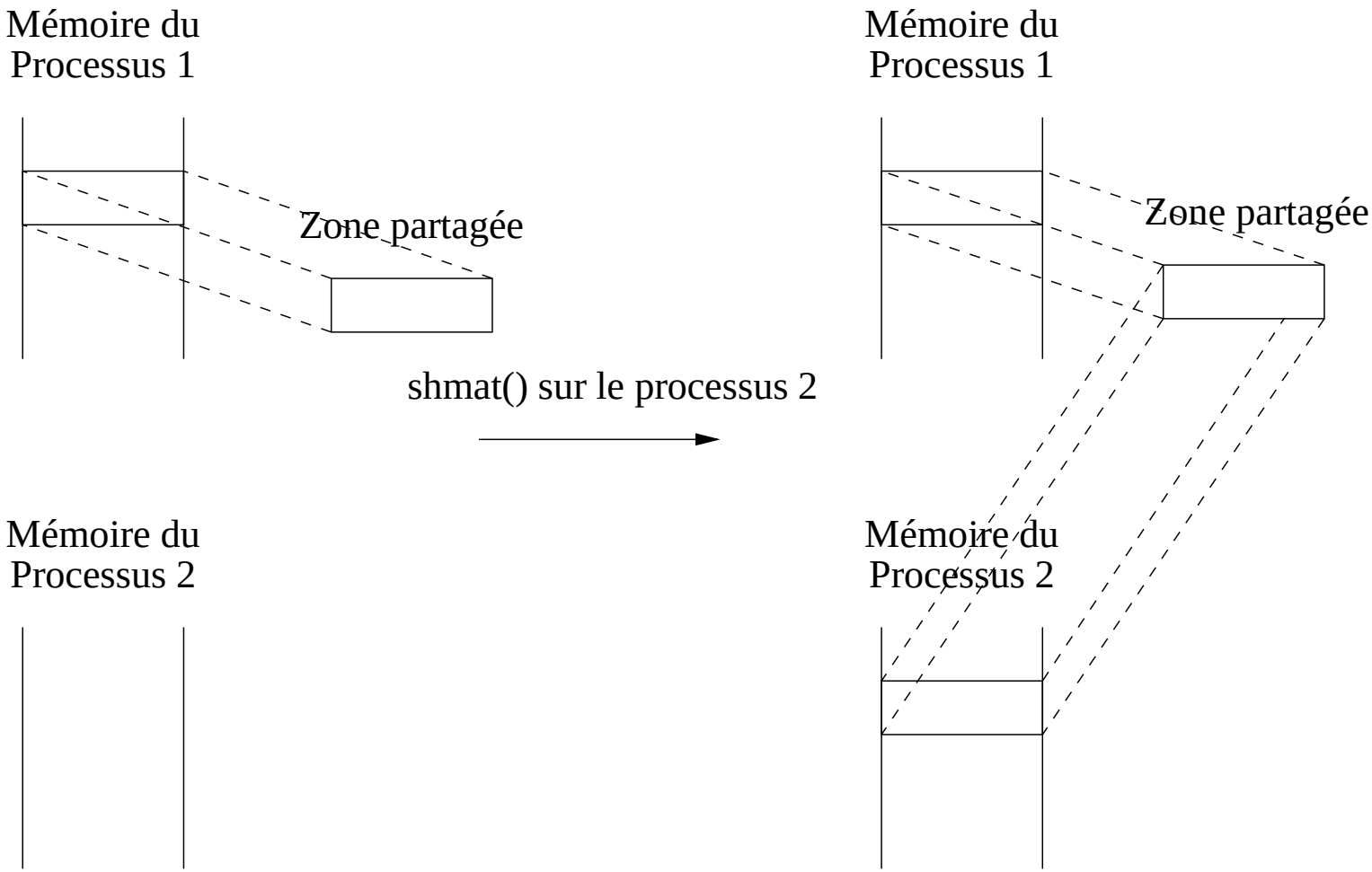
Fonctions IPC

- Attachement de la zone de mémoire partagée au processus 1



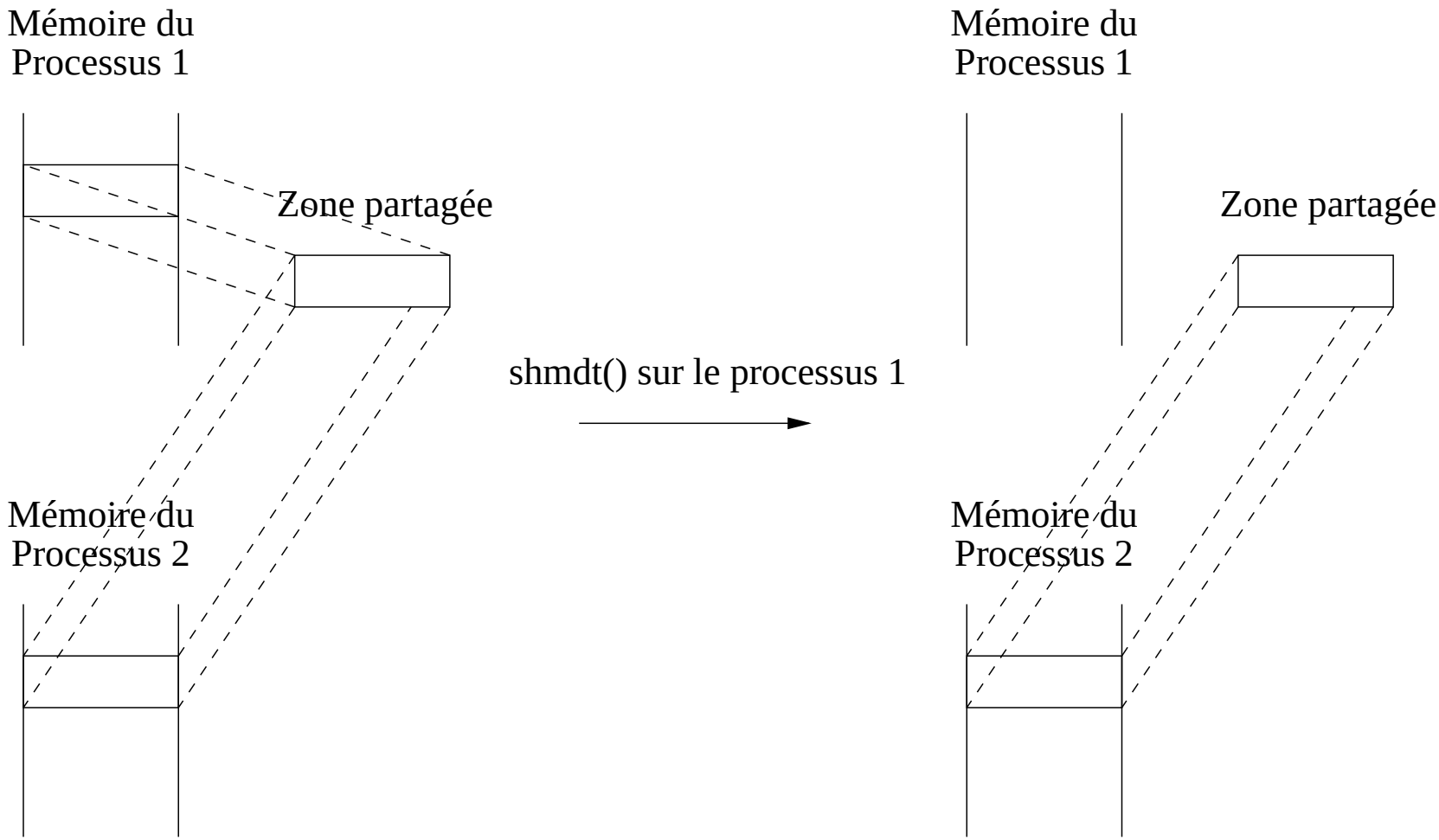
Fonctions IPC

- Attachement de la zone de mémoire partagée au processus 2



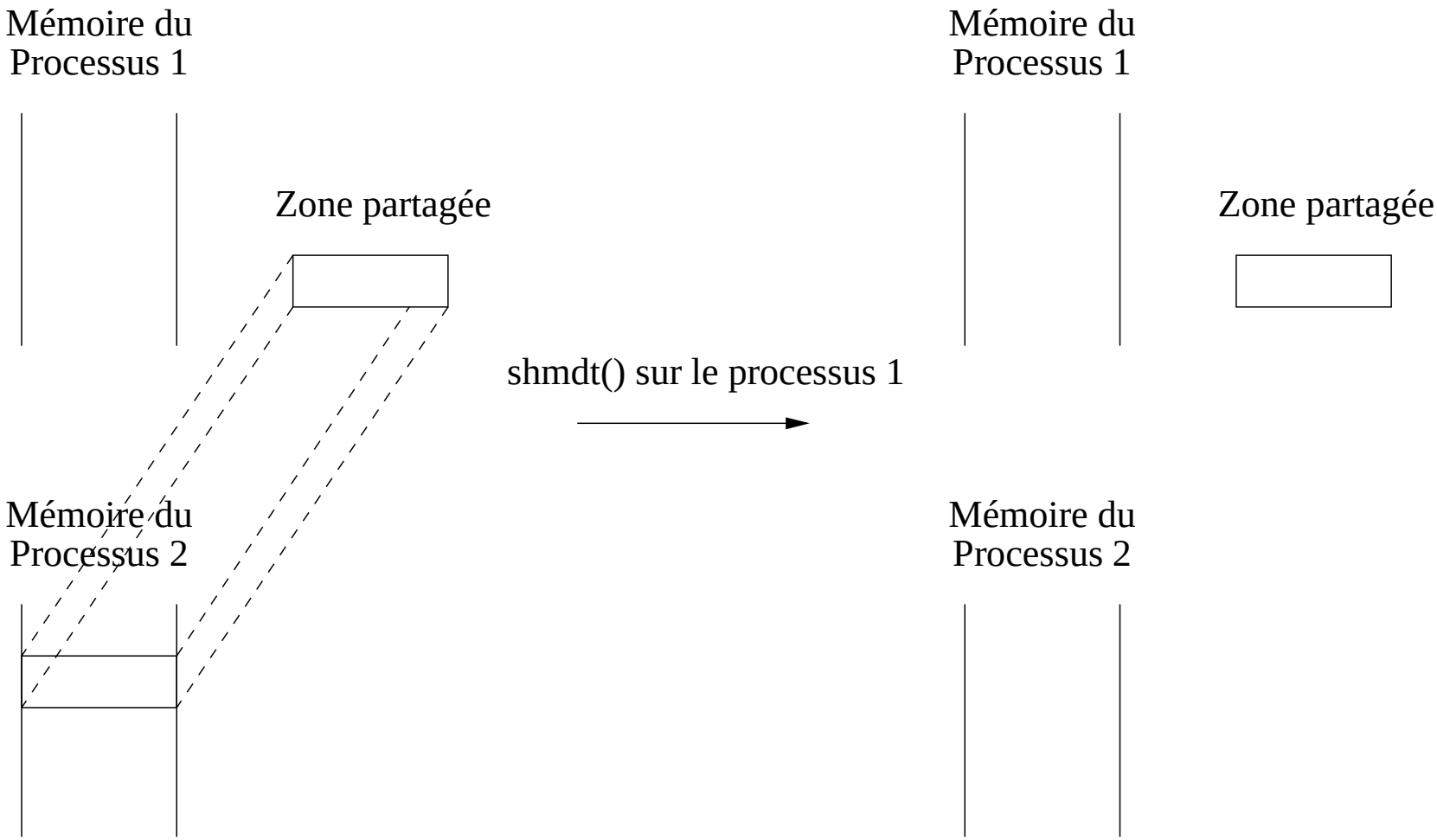
Fonctions IPC

- Détachement de la zone de mémoire partagée du processus 1



Fonctions IPC

- Détachement de la zone de mémoire partagée du processus 2



Un écrivain (situation de compétition)

```
#include ...
#define CLE 100012

struct mybuf {
    int   nbmes;         /* nombre de messages */
    char  mes[50][100]; /* les messages       */
};

int main(int argc, char* argv[]) {
    int i;
    struct mybuf * m;
    int id = shmget( CLE, 100000, IPC_CREAT | 0777);
    if (id == -1) { perror("Création de la mémoire partagée : ");  exit(-
    m = (struct mybuf *) shmat(id, NULL, 0);
    m->nbmes = argc;
    for(i = 0; i < argc ; i++) {
        strncpy(m->mes[i], argv[i], 99);
    }
}
```

Un lecteur (situation de compétition)

```
#include ...
#define CLE 100012

struct mybuf {
    int   nbmes;         /* nombre de messages */
    char  mes[50][100]; /* les messages       */
};

int main(int argc, char* argv[]) {
    int i;
    struct mybuf * m;
    int id = shmget( CLE, 100000, IPC_CREAT | 0777);
    if (id == -1) { perror("Création de la mémoire partagée : ");  exit(-
    m = (struct mybuf *) shmat(id, NULL, SHM_RDONLY); /* lecture seule */
    while (1) {
        for(i = 0; i < m->nbmes ; i++)
            printf("Lecteur [%i] = <%s>\n", i, m->mes[i]);
        sleep(10);
    }
}
```

UNIVERSITÉ DE NANTES

Exemple d'exécution

```
> ecrivain_shm Message_1
> ipcs -m
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     statu
0x000186ac 196611     bechet    777        100000     0
>
> lecteur_shm &
Lecteur [0] = <ecrivain_shm>
Lecteur [1] = <Message_1>
[1] 515
> ecrivain_shm Message_2
> Lecteur [0] = <ecrivain_shm>
Lecteur [1] = <Message_2>
ipcs -m
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     statu
0x000186ac 196611     bechet    777        100000     1
> Lecteur [0] = <ecrivain_shm>
Lecteur [1] = <Message_2>
kill %1
[1]    Terminated                    lecteur_shm
```

Projection de fichier

- `void * mmap(void *addr, size_t l, int prot , int flags, int fd, off_t depl);` pour placer une zone (`l` octets commençant à `depl`) du fichier ouvert `fd`. `prot` indique la protection : `PROT_EXEC|PROT_READ|PROT_WRITE` ou `PROT_NONE`. `flags` indique le type de partage : `MAP_SHARED` ou `MAP_PRIVATE`. `addr` indique l'endroit où doit être placé la projection ou bien `NULL`. Dans tous les cas, la fonction retourne l'adresse de la projection. `depl` doit être un multiple de la taille des pages mémoires (4k octets en général)
- `int munmap(void *addr, size_t l)` pour enlever une projection

Projection de fichier

- Taille du fichier : la taille du fichier n'est pas changé par les projections $\Longrightarrow$une écriture en dehors du fichier génère une *erreur de bus*.
- `ftruncate()` permet de changer la taille des fichiers et des objets
- `fork()` partage la projection entre le père et le fils
- `execve() ou exit()` enlève les projections

Un écrivain (situation de compétition)

```
#include ...

int main(int argc, char* argv[]) {
    int fd = open( "essai.txt", O_RDWR );
    if (fd == -1) { perror("Ouverture : ");  exit(-1); }
    char * m = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    if (m == NULL) { perror("Projection : ");  exit(-1); }
    close(fd);
    strncpy(m, argv[1], 4096);
}
```

Un lecteur (situation de compétition)

```
#include ...

int main(int argc, char* argv[]) {
    int fd = open( "essai.txt", O_RDWR );
    if (fd == -1) { perror("Ouverture : ");  exit(-1); }
    char * m = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    if (m == NULL) { perror("Projection : ");  exit(-1); }
    close(fd);
    while(1) {
        printf("Le message : %s\n", m);
        sleep(3);
    }
}
```