Université de Toulon
Département d'Informatique

Processus: Création, synchronisation, communication

Joseph Razik

dernière mise à jour: 17 septembre 2019

Table des matières

Chapitre 1

Introduction

Ce module est centré sur le processus : la description de son cycle de vie, la synchronisation inter-processus et la communication inter-processus. Mais tout d'abord, qu'est ce qu'un processus ? De manière intuitive et simplifiée un processus est une instanciation d'un programme, c'est-à-dire un programme dont le code contenu dans un fichier est lu puis exécuté.

Or, dans les systèmes actuels, il est rare qu'un seul processus soit exécuté à la fois sur un ordinateur. En général, de nombreux processus sont parallèlement en cours d'exécution. Or chacun de ces processus peut être régi par le même code exécutable ou par une code différent, et être à un stade d'exécution différent et n'ayant qu'une connaissance limitée de l'existence des autres processus. Cette pseudo indépendance issue de la pensée "je suis seul à m'exécuter" pose différents problèmes d'accès à des ressources qu'il faut régler. Il peut également parfois être nécessaire à différents processus de s'échanger des informations et de la faire au bon moment comme par exemple dans un calcul partagé entre différents processus.

Références :

— *Systèmes d'exploitation*, A. Tanenbaum
— *Les systèmes d'exploitation*, de F. Schwaab et B. Wrobel-Dautcourt
— *La programmation sous Unix*, J-M. Rifflet
— *Systèmes informatiques*, http ://sinf1252.info.ucl.ac.be/
— Les pages man ...

Chapitre 2

Synchronisation

D'où vient le besoin de synchronisation ?
Un processus, ou un programme, pour s'exécuter a besoin de plusieurs choses :
— Un CPU pour exécuter les instructions de son code exécutable,
— De la mémoire pour y placer son code que le CPU ira lire,
— De la mémoire pour y stocker des informations temporaires ou de travail et des données,
— Des accès à des périphériques d'entrée/sortie : écran, clavier, lecteur, imprimante, etc.

Tous ces besoins font donc appel à des ressources qui peuvent être plus ou moins partagées entre différents processus. Cette gestion des ressources se fait généralement à travers le système d'exploitation comme par exemple la gestion du CPU par l'ordonnanceur. En effet, deux processus ne peuvent utiliser en même temps le CPU et il faut donc ordonner, ou orchestrer, l'assignation du CPU aux processus. Il y a donc un besoin de synchronisation pour éviter le chaos dans les différents accès aux ressources, qu'elles soient matérielles comme dans l'exemple précédent ou bien logiciel par exemple concernant l'accès au contenu d'une variable en mémoire.

2.1 Exclusion Mutuelle

Pour protéger l'accès aux ressources disponibles pour chaque processus, une technique peut être mise en place : l'Exclusion Mutuelle. Le principe est que quand une ressource est acquise par un processus alors cette ressource n'est plus disponible pour les autres processus qui la demanderaient. Une fois que le processus possesseur de cette ressource la relâche, la rend, un des autres processus demandeurs peut dorénavant l'obtenir.

2.1.1 Les ressources

Ainsi, tout est une question de ressources. Celles-ci peuvent être caractérisées en fonction de leur mode d'accès :
— ressource privée : l'accès est local au processus,
— ressource commune : la ressource est connue de l'ensemble des processus et ils peuvent la posséder,
— ressource partageable : plusieurs processus peuvent la posséder au même instant,
— ressource critique : un seul processus peut la posséder à la fois.

2.1.2 Typologie des processus

Il est également possible de classer les processus selon leurs besoins :
— Les processus indépendants n'utilisent que des ressources locales,
— Les processus concurrents ou parallèles utilisent une ou plusieurs ressources communes. Cette utilisation peut être simultanée ou en exclusion mutuelle. Ces processus ont une progression parallèle, ils peuvent être soit en compétition pour des ressources communes, soit coopérer par des échanges.

Toutefois, la notion de processus indépendant peut être ambiguë et signifie parfois que le processus est concurrent et fonctionne en parallèle d'autres mais n'a pas de dépendance par rapport à d'autres processus.

2.1.3 L'Exclusion Mutuelle

Hypothèse : soit un ensemble de processus concurrents utilisant une ou plusieurs ressources communes non partageables avec pour chaque processus une vitesse d'exécution propre.

L'exclusion mutuelle est un protocole qui se décompose en trois partie : un prologue, un corps d'instructions et l'épilogue. L'exclusion mutuelle sert à protéger l'accès à une ressource critique pendant l'exécution d'une série d'instruction d'un processus particulier : celui qui a obtenu l'exclusion mutuelle. Uniquement ce processus particulier pourra accèder et manipuler la ressource obtenue en exclusion mutuelle. L'ensemble des instructions exécutées en exclusion mutuelle s'appelle la **section critique**.

Ainsi, l'accès à une ressource critique dans une section critique suit le protocole suivant :

— le prologue : régie l'entrée dans la section critique,
— le corps : la section critique
— l'épilogue : régie la sortie de la section critique.

2.1.4 L'objectif

L'objectif maintenant est de proposer un algorithme pour les prologue et épilogue de la section critique afin de réaliser l'exclusion mutuelle avec les contraintes suivantes :

— Un seul processus doit être à la fois en section critique ;
— Si plusieurs processus sont en attente devant la section critique, et si aucun processus n'est en section critique, alors il faut garantir un temps d'attente fini aux processus en attente ;
— Le blocage hors section critique d'un processus ne doit pas empêcher un autre processus d'entrer en section critique ;
— Il n'y a pas de processus privilégié.

2.2 Les modèles pour la réalisation de l'Exclusion Mutuelle

Les différents modèles qui permettent de mettre en place le principe de l'exclusion mutuelle reposent sur l'idée du jeton : celui qui a le jeton est celui qui est autorisé à accéder à la ressource critique. Dans les paragraphes qui suivent nous allons décrire différents mécanismes permettant de réaliser le protocole d'exclusion mutuelle :

— L'attente active,
— Les verrous,
— Les sémaphores.

2.2.1 L'attente active

La méthode de l'attente active est simple à mettre en oeuvre et consiste à tester la valeur d'une variable commune entre les processus. Cette variable s'appelle *flag* (drapeau) et mémorise le droit de passage. Le processus teste la valeur de la variable en boucle jusqu'à l'obtention du droit de passage, du jeton. Les prologue et épilogue d'exclution mutuelle par cette méthode d'attente active est assez simple (cf. algos 2.1 et 2.2).

Pour assurer la cohérence de la valeur de la variable commune, le test et le positionnement de la variable doivent impérativement être effectués dans une séquence d'instructions indivisible dans le temps. Dans le cas contraire, il n'y a plus de garantie d'avoir un seul et unique processus à la fois en section critique.

En effet, sur un ordinateur muni d'un ordonnanceur à temps partagé, le système d'exploitation peut stopper l'exécution d'un processus à n'importe quel endroit de son code et passer la main à un autre processus. Il est donc possible que le premier processus soit stoppé après la lecture de la variable. Or, si celle-ci était à 0, la prochaine instruction qui sera exécutée par ce processus sera de mettre la variable à 1. Mais un deuxième processus obtenant l'unité centrale à ce moment et faisant le même test va lui aussi avoir un droit de passage (la valeur n'a pas encore été changée par le premier processus), il va positionner la variable à 1 et entrer en section critique. Quand le premier processus reprendra la main, il effectuera tout d'abord le positionnement de la variable à 1, sans re-tester car il l'a déjà fait. Ainsi, si le deuxième processus est toujours en section critique, nous aurons deux processus en section critique qui auront des autorisations pour cela.

Algorithme 2.1 : Prologue d'entrée en section critique de l'*attente active*

```
Prologue:
lire P                                          /* on teste la variable commune */
si P = 0 alors
  | P ← 1                                     /* la ressource critique est libre */
sinon
  | Aller à Prologue          /* la ressource critique est prise, on retourne tester */
finsi
```

Algorithme 2.2 : Epilogue de sortie de section critique de l'*attente active*

```
Epilogue :
P ← 0                                  /* il suffit de redonner le droit de passage */
```

Cette méthode présente des inconvénients :

- L'attente active consomme inutilement du temps unité centrale uniquement pour tester une variable par un processus en attente
- La réalisation de l'indivisibilité dans le temps est délicate. Elle peut mettre en oeuvre différents mécanismes, par exemple :
  - Sur une machine mono-processeur : la séquence d'instructions lecture+positionnement sera ininterruptible,
  - Sur une machine multiprocesseur : blocage de la mémoire commune.

  Toutefois, sur la plupart des architectures actuelles, il existe des instructions spéciales garantissant l'indivisibilité. Par exemple, l'instruction *Test And Set* (TAS) qui effectue la lecture, le test, et le positionnement de la variable de manière indivisible dans le temps (une seule instruction processeur).

2.2.1.1 L'algorithme de Peterson

En 1981, Peterson a proposé une méthode qui permet de réaliser l'exclusion mutuelle d'une ressource tout en permettant une alternance entre les acquéreurs (chacun son tour). Cet algorithme de synchronisation de 2 processus repose sur la définition des deux procédures de prologue et d'épilogue ainsi que sur la donnée d'une variable commune (P) et d'un tableau de "réservation" (S).

Avant d'entrer en section critique, chaque processus appelle le prologue `EntrerRegion` avec comme paramètre son propre numéro de processus (0 ou 1). Après cet appel, le processus est mis en attente si besoin jusqu'à ce qu'il puisse entrer. L'attente est une attente active avec test de la variable partagée. Une fois la section critique terminée, le processus exécute l'épilogue `SortirRegion` pour autoriser un autre processus à entrer.

Algorithme 2.3 : Prologue pour l'algorithme de Peterson

```
Entrées : EntrerRegion(Entier processus) /* processus = 0 ou 1 */
S[processus] ← Vrai ;          /* le processus souhaite entrer en section critique */
P ← processus
tant que (P = processus) et (S[1-processus] = Vrai) faire
fintq
```

Algorithme 2.4 : Epilogue pour l'algorithme de Peterson

```
Entrées : SortirRegion(Entier processus) /* processus = 0 ou 1 */
S[processus] ← Faux
```

A vous de simuler l'exécution de cet algorithme avec deux processus (processus 0 et 1) et comprendre

le fonctionnement de celui-ci. Par ailleurs, il est également possible d'utiliser un seul et même algorithme à la fois pour le prologue et pour l'épilogue.

Algorithme 2.5 : Prologue et Epilogue pour l'algorithme de Peterson

Entrées : EntrerSortirRegion(Entier processus) `/* processus = 0 ou 1 */`
S[processus] ← Vrai ; `/* le processus souhaite entrer en section critique */`
P = processus
tant que *(P = processus)* **et** *(S[1-processus] = Vrai)* **faire**
fintq
section_critique()
S[processus] ← Faux

2.2.1.2 Complément sur la fonction TAS

De nombreux ordinateurs prennent en charge l'instruction *Test And Set Lock* qui réalise :
— chargement d'un mot mémoire dans un registre
— rangement d'une valeur non nulle à l'adresse du mot chargé

Les opérations qui consistent à charger un mot mémoire et y placer une valeur sont indivisibles dans le temps. Aucun autre processeur ou processus ne peut accéder au mot mémoire tant que l'instruction n'est pas terminée. Le processeur exécutant l'instruction TSL verrouille le bus mémoire pour interdire aux autres processeus (s'il y en a) d'accéder à la mémoire tant qu'il n'a pas terminé. Dans le cas d'un seul processeur il est possible de simplement désactiver les interruptions (l'instruction devient initerruptible). Mais dans le cas multi-processeur, ce n'est pas possible. Ci-dessous est donné le code assembleur possible pour les prologue et époligue en utilisant l'instruction test and set lock.

```
1   entrer_region:
2   tsl reg, verrou ; Test and Set Lock
3   ; realise en une instruction atomique reg = verrou et verrou = 1
4   cmp reg, 0 ; le verrou etait-il nul en entrant ?
5   jnz entrer_region ; si non, on boucle, attente active
6   ; si oui, on entre en section critique
7   ret
```

Listing 2.1 – Code assembleur du Test And Set

```
1   sortir_region:
2   move verrou, 0 ; met le verrou a 0
3   ret
```

Listing 2.2 – Code assembleur du Test And Set - épilogue

2.2.2 Les verrous

Un verrou est une structure de données composée d'une variable booléenne et d'une file d'attente de processus. La variable booléenne est initialisée à la valeur 0 pour indiquer que la ressource critique est libre. Le passage de la valeur de cette variable à 1 indique évidemment que la ressource critique est occupée.

La différence apparente entre l'attente active et les verrous est l'existence d'une liste d'attente de processus. En effet, un processus qui n'obtient pas le jeton sera mis en file d'attente dans l'état endormi. Il sera réveillé quand il pourra passer.

La manipulation d'un verrou se fait à travers deux primitives, *Verrouiller* et *Déverrouiller*, qui correspondent au prologue et à l'épilogue de l'entrée en section critique qui s'exécutera en exclusion mutuelle.

L'utilisation des verrous est pratique et facile pour assurer l'accès à une ressource critique unique. Cependant ils ne suffisent pas dans le cas où il est possible d'avoir un nombre limité de processus entrés en section critique (plus qu'un). Dans ce cas il faut une autre structure de données : les sémaphores.

Algorithme 2.6 : Prologue pour la méthode des verrous, primitive *Verrouiller*

```
Verrouiller:
lire P
si P = 0
alors
| P ← 1
sinon
| mettre le processus dans la file d'attente et le mettre dans l'état endormi
finsi
```

Algorithme 2.7 : Epilogue pour la méthode des verrous, primitive *Déverrouiller*

```
Déverrouiller:
si file d'attente non vide
alors
| sortir un processus de la file et le réveiller
sinon
| P ← 0
finsi
```

2.2.3 Les sémaphores

Les sémaphores peuvent être considérés comme une extension ou une généralisation des verrous. Ils ont été proposés par Dijkstra en 1967-68. Un sémaphore est défini par deux objets :

— Une variable entière,

— Une file d'attente de processus.

La différence apparente entre les verrous et les sémaphores est le passage d'une variable de type booléen à une variable entière qui peut être négative, positive ou nulle.

L'utilisation des sémaphores repose sur quatre primitives de manipulation qu'il est impératif d'utiliser :

— `création(s, val)` : primitive de création d'un sémaphore de nom `s` et de valeur initiale `val`.

Algorithme 2.8 : Création d'un sémaphore : `creation(s,val)`

```
récupération d'une zone de mémoire
création de la structure de données de nom s
E(s) ← val
pointeur de file d'attente ← nil
```

— `P(s)` : demande d'un droit de passage, d'un jeton (P = prendre [1])

Algorithme 2.9 : Prologue pour les sémaphores, primitive P

```
E(s) ← E(s) − 1 ;      /* prise systématique d'un jeton, mémorise ainsi la demande */
si E(s) < 0 alors
| le processus est placé dans la file d'attente et s'endort (état bloqué)
finsi
```

— `V(s)` : restitution du droit de passage (V = libérer [2])

— `destruction(s)` : primitive de destruction d'un sémaphore de nom `s`.

2.2.3.1 Exemple d'accès à une ressource critique protégée par un sémaphore

Soit une ressource critique protégée par un sémaphore initialisé à 1. Les sémaphores initialisés à 1 sont destinés à réaliser une exclusion mutuelle, ils seront généralement appelés Exmut ou Mutex (Mutual Exclusion).

1. P, du néerlandais de Proberen : tester
2. V, du néerlandais Verhogen : incrémenter

Algorithme 2.10 : Epilogue pour les sémaphores, primitive V

$E(s) \leftarrow E(s) + 1$; /* on rend le jeton */
si $E(s) \leq 0$; /* il y a des processus en attente */
alors
| on réveille un processus de la file d'attente (il redevient actif)
finsi

Algorithme 2.11 : Destruction d'un sémaphore

vérification qu'il ne reste aucun processus en attente et en section critique
libération de la mémoire

Supposons le déroulement suivant :
— création du sémaphore Mutex initialisé à 1,
— création d'un processus P_1,
— création d'un processus P_2,
— exécution des algorithmes des deux processus P_1 et P_2.

Supposons également que dans leur algorithme propre, les deux processus font appel à l'exclusion mutuelle pour accéder à la même ressource critique. Leur algorithme ressemblerait à ceci :

P_1	P_2
*	*
*	*
P(Mutex)	P(Mutex)
*	*
section critique	section critique
*	*
V(Mutex)	V(Mutex)
*	*
*	*

TABLE 2.1 – Algorithme des deux processus

Les parties entre P et V étant la section critique. Une trace d'exécution possible est présentée plus en détail dans le tableau 2.2.

Remarque : Les algorithmes d'entrée et de sortie d'une section critique doivent impérativement vérifier les conditions suivantes :
— Si aucun processus ne se trouve en section critique, alors il ne doit pas y avoir de processus bloqué derrière le verrou ou le sémaphore d'exclusion mutuelle ;
— Aucun processus ne doit pouvoir entrer en section critique sans exécuter la primitive P ;
— Aucun processus ne doit sortir de section critique sans exécuter la primitive V ;

2.2.3.2 Propriétés des sémaphores

— On ne peut pas initialiser un sémaphore avec une valeur négative, mais la valeur courante de $E(s)$ peut devenir négative selon le nombre d'exécution de la primitive P ;
— $E(s) = E_0(s)$ - nombre d'exécution de $P(s)$ + nombre d'exécution de $V(s)$, avec $E_0(s)$ la valeur initiale du sémaphore et $E(s)$ sa valeur courante ;
— Si $E(s) > 0$, $E(s)$ représente le nombre de processus pouvant passer ;
— Si $E(s) < 0$, $E(s)$ représente le nombre de processus en attente ;
— Si $E(s) = 0$, aucun processus n'attend et aucun processus ne peut passer.

2.2.4 Quelques limites

— Il est interdit de manipuler un sémaphore ou un verrou autrement qu'avec les primitives dédiées ;

P_1		P_2	
zzz		**P**(**Mutex**)	$E(Mutex) \leftarrow 0$
⋮		*	P_2 peut entrer en
⋮		*	section critique
zzz		*	
P_1 prend la main		*zzz*	
P(**Mutex**)	$E(Mutex) \leftarrow -1$	P_2 reprend la main	car P_1 est bloqué
zzz	donc P_1 se bloque	*	
⋮		*	
zzz		**V**(**Mutex**)	$E(Mutex) \leftarrow 0$
réveil de P_1		*	
zzz		*	
P_1 prend la main		*zzz*	
		⋮	
V(**Mutex**)	$E(Mutex) \leftarrow 1$	⋮	
		⋮	

TABLE 2.2 – Exemple d'exécution, initialisation de E(Mutex) à 1, on suppose que P2 commence.

- Quand un processus est tué en section critique, il faut remettre le système en état de fonctionnement, par exemple exécuter V(s) avant de mourir. Dans le cas contraire, il est possible d'avoir un blocage définitif des processus en attente ;
- L'écriture et la vérification des algorithmes utilisant des verrous ou des sémaphores n'est une chose facile : les primitives sont dispersées dans le texte et sont de bas niveau.
- La stratégie de gestion de la file d'attente est importante car sinon il y a le risque de *famine*, i.e. qu'un processus est indéfiniment en attente d'une ressource. Il faut alors faire attention à ce que :
  - La mise en file et la sortie de file respectent les mêmes stratégies, par exemple FIFO ou gestion par priorité ;
  - La gestion de la file ne doit pas permettre à un sous-ensemble de processus de bloquer indéfiniment un autre sous-ensemble de processus.

2.3 Synchronisation entre processus

La synchronisation ne se limite pas uniquement à l'accès concurrentiel à des ressources critiques qu'il faut protéger. Il peut aussi parfois être nécessaire à des processus de communiquer entre eux simplement par exemple pour s'attendre. Cette forme de synchronisation afin que des processus puissent coopérer peut être réalisée à l'aide de l'exclusion mutuelle. Nous allons maintenant détailler d'autres mécanismes de synchronisation plus complexes que l'élémentaire exclusion mutuelle.

L'objectif est donc de créer un mécanisme qui soit indépendant de la vitesse d'exécution propre des processus mais qui permette aussi à un processus actif p :

- D'en bloquer un autre ou de se bloquer lui-même ;
- D'activer un autre processus q en lui transmettant éventuellement une information. Cependant, le processus q peut se trouver aussi bien dans l'état bloqué que déjà dans l'état actif. Deux cas se présentent alors :
  - Le signal d'activation n'est pas mémorisé, il est donc perdu si q ne l'attend pas ;
  - Le signal d'activation est mémorisé, et dans ce cas q ne se bloquera pas lorsqu'il se mettra en attente de ce signal.

Bien que toutes ces possibilités soient dirigées vers un processus récepteur q, ce récepteur peut être explicitement désigné ou non. Nous obtenons ainsi les mécanismes à :

- Actions directes : le processus est désigné par son nom (pid) ou bien agit directement sur lui-même ;

— Actions indirectes : le processus ne manipule que des noms de variables accessibles par d'autres processus.

Nous allons développer le cas des actions indirectes. Le mécanisme des actions indirectes met en jeu un ou plusieurs objets intermédiaires connus des processus. Ces objets ne leur sont manipulables que par des opérations indivisibles spécifiques. Il existe deux types de synchronisation :

— La synchronisation par événements,
— La synchronisation par sémaphores.

2.3.1 Par événements

Dans ce cas de synchronisation, un processus se bloque lorsque l'événement qu'il attend n'est pas encore arrivé. Le déclenchement d'un événement débloque alors le processus ou tous les processus qui l'attendent. En général, le processus activité par l'occurrence d'un événement exécute une fonction explicitement attachée à l'arrivée de cet événement. Ce fonctionnement est similaire au traitement des interruptions matérielles.

Il y a deux façons de répondre à l'émission d'un événement selon qu'il est mémorisé ou non :

— Événements non mémorisés : un événement émis quand aucun processus ne l'attend est perdu.
— Évènements mémorisés : les événements émis sont mémorisés mais en général on ne mémorise qu'une seule activation pour un processus déjà actif.

Trois primitives sont définies pour la synchronisation par événements :

— bloquer un processus en attente d'un événement,
— associer une action à un événement,
— envoyer un événement à un processus.

2.3.2 Par sémaphores

La synchronisation par sémaphores introduit un type particulier de sémaphores : le sémaphore privé.

Un sémaphore s est dit privé au processus p si seul le processus p peut exécuter la primitive P(s). Les autres processus ne peuvent agir sur s que par la primitive V(s). Avec ce mécanisme, un signal d'activation est envoyé par la primitive V, et ce signal est attendu par la primitive P. Ce mécanisme est un mécanisme à mémoire car le sémaphore posséde une valeur entière pour mémoriser les *jetons*. Une ou plusieurs opération V(s) effectuées avant les primitives P(s) du processus p seront alors prises en compte le bon nombre de fois. Un sémaphore privé au processus p est créé par p et initialisé à 0. De manière générale, un sémaphore privé est attaché à un processus alors qu'un processus d'exclusion mutuelle est attaché à une ressource.

Exemple :

P_1	P_2	P_3
zzz	**création(s,0)**	*zzz*
···	**P(s)**	*zzz*
···	blocage	reprise de l'exécution
···	*zzz*	*
···	*zzz*	**V(s)**
···	réveil	*
···	*zzz*	*
···	reprise de l'exécution	*zzz*
···	*	···
zzz	*	···
reprise de l'exécution	*zzz*	···
V(s)	*zzz*	···
*	mémorisation du réveil	···
*	*zzz*	···
zzz	reprise de l'exécution	···
···	**P(s)**	···
···	pas de blocage	···
zzz	*	*zzz*

Le premier P(s) provoque un blocage du processus P2 car le sémaphore est initialisé à 0. Si P2 est bloqué sur une primitive P(s), quand le signal de déblocage V(s) arrive, il est réveillé. Si P2 n'est pas bloqué quand le signal de déblocage est émis, celui-ci est mémorisé. Quand P2 exécutera P(s), il ne sera pas bloqué, il y a anticipation du déblocage. Il est possible de mémoriser un nombre quelconque d'activations quand le processus auquel elles sont destinées est actif.

2.3.3 Les rendez-vous

Les rendez-vous permettent de synchroniser deux processus afin d'être certain qu'ils soient au même instant à un endroit précis de leur code respectif. Un moyen simple de réaliser un rendez-vous consiste à utiliser deux sémaphores privés.

P_1	P_2
*	*
*	*
P(s_1)	V(s_1)
V(s_2)	P(s_2)
*	*
*	*

TABLE 2.3 – Algorithme d'un rendez-vous entre deux processus

Au niveau des deux séquences respectives P-V et V-P, il y a attente forcée des deux processus jusqu'à l'arrivée au point de rendez-vous.

De plus il est possible d'insérer une section critique mais uniquement dans le processus réalisant la séquence ordonnée P-V.

P_1	P_2
*	*zzz*
P(s_1)	*zzz*
blocage	reprise de l'exécution
zzz	*
...	**V(s_1)**
réveil	**P(s_2)**
reprise de l'exécution	blocage
*	*zzz*
*	...
*	*zzz*
V(s_2)	réveil
*	...
zzz	reprise de l'exécution
...	*

2.4 Interblocage

2.4.1 description et analyse du problème

Un interblocage (ou *deadlock*) est une situation dans laquelle un ensemble de processus se retrouvent mutuellement bloqués en attente de ressources sans possibilité de pouvoir sortir de cette situation.

Un cas d'interblocage se pose avec la configuration simple de deux processus concurrents et de deux ressources critiques. Lorsque chaque processus possède une des deux ressources et qu'ils veulent obtenir la seconde, ils se bloquent mutuellement et indéfiniment (voir exemple tableau 2.4). D'une manière générale, un interblocage est susceptible de se produire dès que plusieurs processus convoitent plusieurs ressources critiques.

P_1	P_2
*	*
demande la ressource R_2	demande la ressource R_1
ressource R_2 attribuée	ressource R_1 attribuée
*	*
*	*
demande la ressource R_1	demande la ressource R_2
blocage	blocage
Les deux processus se bloquent mutuellement Ils ne peuvent le savoir ni se débloquer d'eux-mêmes	

TABLE 2.4 – Exemple d'interblocage à deux processus

Il est important de remarquer qu'un processus bloqué ne peut pas savoir qu'il est bloqué. En effet, pour connaître son état un processus doit être actif.

Voici un deuxième exemple d'un interblocage sur une ressource partageable : la mémoire centrale. L'accès à la mémoire centrale est protégée en exclusion mutuelle mais pour l'utilisation même de la mémoire, les processus ne bloquent pas tout le contenu de celle-ci. Ils ne réservent que la quantité dont ils ont besoin. Ainsi la mémoire est une ressource partageable car plusieurs processus peuvent "posséder" une partie de la mémoire.

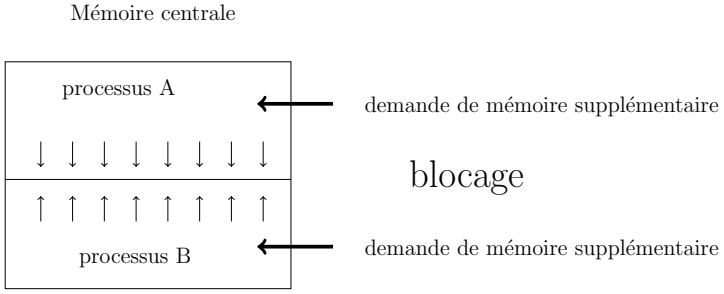
A priori, l'accès à la mémoire étant protégé, nous pouvons penser qu'il n'y a pas de risque d'interblocage. Supposons que nous avons deux processus A et B qui se partagent la mémoire. Supposons que ces processus ont un besoin croissant de mémoire.

Au bout d'un moment, toute la mémoire est partagée entre ces deux processus. Puis :

— Le processus A demande une extension de mémoire : il est mis en attente ;

— Le processus B demande une extension de mémoire : il est mis en attente.

Ainsi les deux processus sont bloqués indéfiniment.



L'interblocage est donc un phénomène qui peut se produire assez facilement dans un système. Cependant, il existe différentes façon de prendre en compte cette éventualité :

— Soit on ignore complètement les possibilités d'interblocages. Cette politique est celle le plus souvent mise en place dans les systèmes d'exploitation courants car le prix à payer pour éviter les interblocages est élevé ;

— Soit on les détecte et on y remédie : on tente alors de détecter une situation d'inteblocage et en général on traite cet interblocage en éliminant les processus bloqués ;

— Soit on les évite en allouant avec précaution les ressources, de façon dynamique. Un système peut suspendre le processus qui veut demander une ressource si cela conduit à un interblocage. Il lui redonnera la ressource lorsqu'il n'y aura plus de risque ;

— Soit on les prévient en empêchant qu'il y ait des ressources en exclusion mutuelle, ou qu'un processus possédant déjà des ressources n'en demande des autres, ou qu'une ressource prise par un processus ne puisse être libérée que par ce processus, ou qu'il y ait une attente circulaire.

2.4.2 Graphe d'allocation des ressources

Le graphe d'allocation des ressources permet de rendre compte de l'état actuel du système du point de vue des ressources acquises et demandées par les processus. Ce graphe est composé de deux types de

noeuds et d'un ensemble d'arcs orientés :

— Les processus sont représentés par des cercles ;
— Les ressources sont représentées par des rectangles, chaque rectangle contient autant de points que d'exemplaires de la ressource dans le système ;
— Des arcs orientés d'une ressource vers un processus pour indiquer qu'un exemplaire de la ressource est alloué au processus ;
— Des arcs orientés d'un processus vers une ressource pour indiquer qu'un processus demande un exemplaire de la ressource (bloqué en attente de celle-ci).

Exemple : Soient 3 processus A, B et C qui ont besoin de 3 ressources R, S et T selon l'ordre donné dans le tableau suivant :

A	B	C
Demande R	Demande S	Demande T
Demande S	Demande T	Demande R
Libère R	Libère S	Libère T
Libère S	Libère T	Libère R

TABLE 2.5 – Algorithme des trois processus A, B, C

Si les processus sont exécutés de façon séquentielle : A entièrement puis B entièrement puis C, il n'y a pas d'interblocage. Mais si nous supposons maintenant qu'une seule instruction de chaque processus est exécutée tour à tour (A, B, C, A, B, C ...), alors il y a interblocage à la sixième instruction exécutée.

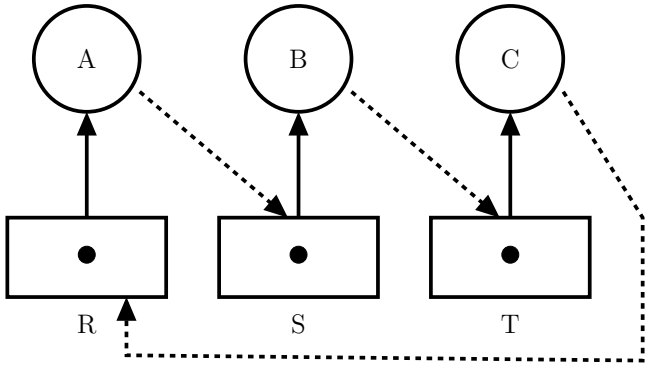


FIGURE 2.1 – Interblocage circulaire

2.4.3 Réduction du graphe d'allocation des ressources

Un graphe d'allocation des ressources peut vite devenir difficile à lire si beaucoup de processus et de ressources sont en jeu. Il est alors possible de réduire la complexité du graphe. Ce graphe réduit, comme le graphe initial, peut être utilisé afin déterminer s'il existe ou non un interblocage. Pour réduire un graphe d'allocation des ressources, il faut vérifier les arcs entre les ressources et les processus et appliquer les règles suivantes :

— Si une ressource ne possède que des arcs qui sortent, on les efface (il n'y a pas de demandes) ;
— Si un processus ne possède que des arcs entrant, on les efface (il possède toutes les ressources dont il a besoin) ;
— Si une ressource a des arcs sortant mais que pour chaque requête vers cette ressource il existe une ressource de ce type disponible, alors on peut effacer ces flèches.

Cette troisième règle peut être difficile à comprendre mais elle repose sur le fait qu'il faut distinguer ce qu'est un blocage temporaire de quelques processus, de ce qu'est un interblocage, c'est-à-dire un blocage total de tous les processus mis en jeu. Elle repose également sur le fait que les demandes sont des demandes statiques, c'est-à-dire que ce sont toutes les demandes qu'effectueront les processus et pas uniquement celles effectuées à un instant donné. Ainsi si un processus possède toutes les ressources dont il a besoin

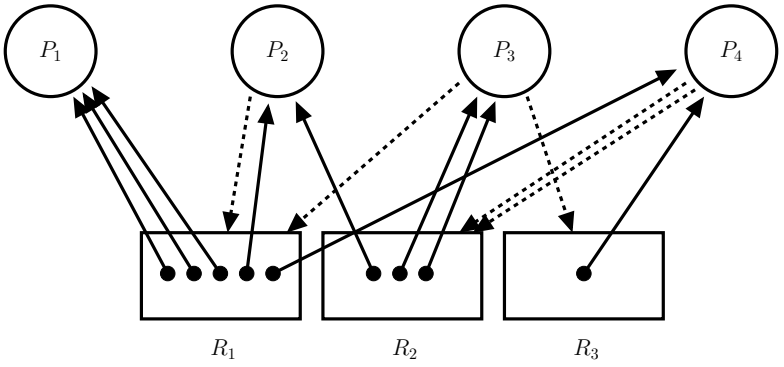
(règle 2), les ressources qu'il possède seront libres tôt ou tard avec certitude. C'est pourquoi la règle 3 réintroduit ces ressources dans le cycle d'analyse du graphe.

Par exemple, soient 4 processus P_1, P_2, P_3, P_4 et 3 ressources R_1, R_2 et R_3. Le tableau 2.6 montre l'allocation courante des ressources et le nombre maximum de ressources nécessaires pour l'exécution des processus. Il ne reste aucune ressource disponible au moment de la construction du graphe.

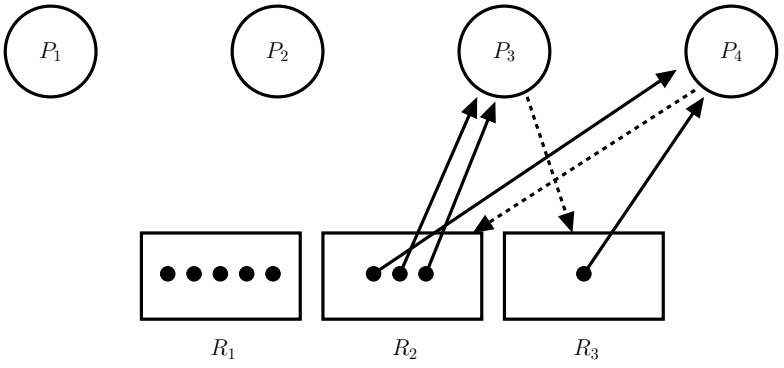
	allocations			demandes		
Processus	R_1	R_2	R_3	R_1	R_2	R_3
P_1	3	0	0	0	0	0
P_2	1	1	0	1	0	0
P_3	0	2	0	1	0	1
P_4	1	0	1	0	2	0

TABLE 2.6 – Allocation courante et besoins des 4 processus

Nous obtenons alors le graphe suivant :



Nous allons entamer sa réduction : P_1 ne possède que des flèches qui pointent vers lui et donc par la règle 2, on efface ces flèches et les 3 ressources R_1 sont alors libérées. Ainsi la requête de P_2 en ressource R_1 peut être satisfaite et dans ce cas la flèche de requête devient une flèche d'acquisition. P_2 ne possède alors lui aussi plus que des flèches pointant vers lui et nous pouvons donc les effacer (ou appliquer la règle 3). La requête de P_3 en ressource R_1 peut alors être satisfaite ainsi qu'une des deux requêtes de P_4 en ressource R_2. Comme R_1 ne possède que des flèches sortantes, par la règle 1, on peut les effacer. Le graphe final réduit est le suivant :



2.4.4 Détection

Dans le cas de la détection, on ne cherche pas à empêcher les interblocage mais uniquement détecter si le système est dans une situation d'interblocage et dans ce cas d'y remédier. Afin de détecter une telle

situation, il est possible de construire dynamiquement le graphe d'allocation des ressources. Deux cas de détection d'interblocage se présentent :

— Si les ressources sont à exemplaire unique et si le graphe contient au moins un cycle, alors il existe un interblocage ;
— Si les ressources sont à exemplaires multiples et si le graphe contient au moins un cycle terminal (aucun arc ne permet de le quitter), alors il existe un interblocage ;

Le système peut effectuer cette phase de détection d'interblocage (et donc de construction du graphe) :

— Soit à chaque nouvelle demande de ressource (coûteux en cpu),
— Soit périodiquement ou lorsque l'utilisation du processeur est inférieure à un seuil (la détection dans ce cas peut être tardive) ;

2.4.5 Guérison

Lorsque le système détecte un interblocage, il doit faire en sorte de le supprimer avec le moins d'effet de bord pour les autres processus. Il existe dans ce cas aussi plusieurs stratégies possibles :

— Retirer temporairement une ressource à un processus pour l'attribuer à un autre,
— Restaurer un état antérieur et éviter de tomber de nouveau dans une situation d'interblocage,
— Supprimer un ou plusieurs processus (version souvent privilégiée par les systèmes).

2.4.6 Prévention

Dans le cas de la prévention, l'objectif est d'empêcher un interblocage en intervenant juste avant l'allocation de ressource à un processus. Ainsi, dès qu'un processus demande une nouvelle ressource, le système détermine si l'attribution de cette ressource va conduire à un interblocage. Si l'attribution ne pose pas de problème, le système lui donne effectivement, sinon, le système étudiera de nouveau la demande quand la situation aura changé (ressources rendues par d'autres processus). Le système détermine qu'une situation est sûre (dit aussi "état sûr"), si tous les processus peuvent terminer leur exécution. Il est bien sûr possible que les processus soient de temps à autre en attente mais dans tous les cas ils pourront terminer leur exécution. Si le système ne peut pas garantir que les processus pourraient terminer leur exécution alors l'état est qualifié de "non sûr".

Afin de déterminer si un état est sûr ou non sûr, il est possible d'utiliser le graphe d'allocation de ressources. Cependant, une fois le graphe établi et/ou réduit, il faut appliquer un algorithme de détection de cycle, ce qui n'est pas forcément ni évident ni rapide. En 1965, Dijkstra a proposé un algorithme d'ordonnancement appelé l'*Algorithme du banquier* qui permet de déterminer si un état est sûr ou non. Pour cela, l'algorithme défini et utilise des données qu'il est possible de déduire du graphe :

— Une matrice C des allocations courantes de ressources (d'ordre $n \times m$). L'élément $C(i, j)$ de la matrice désigne le nombre de ressources de type j que possède le processus i ;
— Une matrice R des demandes de ressources (d'ordre $n \times m$). L'élément $R(i, j)$ de la matrice désigne le nombre de ressources de type j qu'il manque au processus i pour pouvoir poursuivre son exécution ;
— Un vecteur A (d'ordre m) dont l'élément $A(j)$ correspond au nombre de ressources de type j actuellement disponibles ;
— Un vecteur E (d'ordre m) dont l'élément $E(j)$ correspond au nombre total de ressources de type j existant dans le système.

L'algorithme est donné à l'algorithme 2.12.

Bien que cet algorithme permette d'éviter les interblocages, il est rarement utilisé car il est difficile de connaître à l'avance les besoins en ressources des processus.

2.5 Généralisation des sémaphores

Les primitives P et V introduites par Dijkstra ont été définies dans le cadre d'une gestion de sémaphores élémentaires qui ne permettent de prendre ou de rendre qu'un seul jeton à la fois. Cependant, plusieurs extensions de la notion de sémaphore permettent de généraliser ceux-ci.

— Une extension pour prendre ou rendre k jetons,
— Une extension pour opérer de façon indivisible sur un ensemble de sémaphores.

Algorithme 2.12 : Algorithme du banquier

`Début` :
Trouver un processus P_i non marqué dont la rangée i de R est inférieure à A ; `/* i.e. il existe suffisamment de ressources pour que` P_i `puisse s'exécuter */`
si *existe un tel processus* **alors**
  ajouter la rangée i de C à A et marquer le processus ; `/* le processus pouvant s'exécuter, il rendra à un moment les ressources qu'il possède */`
sinon
  état est non sûr et il y a interblocage. Fin de l'algorithme
finsi
si *tous les processus sont marqués* **alors**
  état sûr et l'algorithme est fini
sinon
  Aller à Début
finsi

2.5.1 Primitives P et V à k jetons

Cette extension des sémaphores permet de prendre ou rendre k jetons. Les nouvelles primitives sont $P(s,k)$ et $V(s,k)$, k étant un entier positif. Toutefois, cette extension elle-même peut être implantée de deux manières différentes selon qu'il soit autorisé ou non de faire une allocation partielle des ressources ou non pour la primitive P. Voici les deux cas possibles :

— Décrémenter E(s) de façon à conserver $E(s) \geq 0$, en mémorisant le cas échéant la fraction de demande non satisfaite. La file est gérée en FIFO.

Algorithme 2.13 : Primitive $P(s,k)$ pour sémaphore généralisé à allocation partielle

W $\leftarrow E(s) - k$
si $W < 0$ **alors**
  $j \leftarrow |W|$, $E(s) \leftarrow 0$, mettre le processus en queue de file d'attente avec sa demande non satisfaite j
sinon
  $E(s) \leftarrow W$
finsi

Algorithme $V(s,k)$?

— Ne décrémenter E(s) que si $E(s) \geq k$, sinon mémoriser la demande non satisfaite. La file d'attente peut être gérée en FIFO (algo 2.14) ou par priorité (2.15) mais avec un risque de famine.

Algorithme 2.14 : Primitive $P(s,k)$ pour sémaphore généralisé à allocation complète (FIFO)

si $E(s) \geq k$ **alors**
  $E(s) \leftarrow E(s) - k$
sinon
  bloquer le processus en queue de la file
finsi

Algorithme $V(s,k)$?

2.5.2 Tableaux de sémaphores

On généralise les primitives P et V à un ensemble de sémaphores regroupés dans un tableau. La primitive P(tableau_sémaphore) permet d'appliquer globalement la primitive P sur chaque élément du tableau. Ainsi, il n'y a pas d'interblocage car soit on possède toutes les ressources, soit on en possède aucune.

Algorithme 2.15 : Primitive $P(s,k)$ pour sémaphore généralisé à allocation complète (priorités)

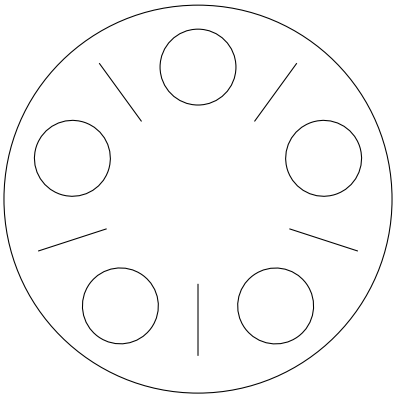
si $E(s) \geq k$ **alors**
| $E(s) \leftarrow E(s) - k$
sinon
| bloquer le processus au rang k de la file
finsi

Toutefois dans le système Unix/Linux, un objet sémaphore hybride est défini, croisement entre un sémaphore simple, un tableau de sémaphores et les sémaphores à k jetons.

2.6 Allocation globale des ressources

Le problème des Philosophes aux spaghettis : comment combiner l'emploi des sémaphores d'exclusion mutuelle et des sémaphores privés pour résoudre un problème d'interblocage. Bien que le nom original de ce problème soit « Les Philosophes aux spaghettis », pour une analogie plus parlante nous allons transposer ce problème en « Les Philosophes aux nouilles chinoises ».

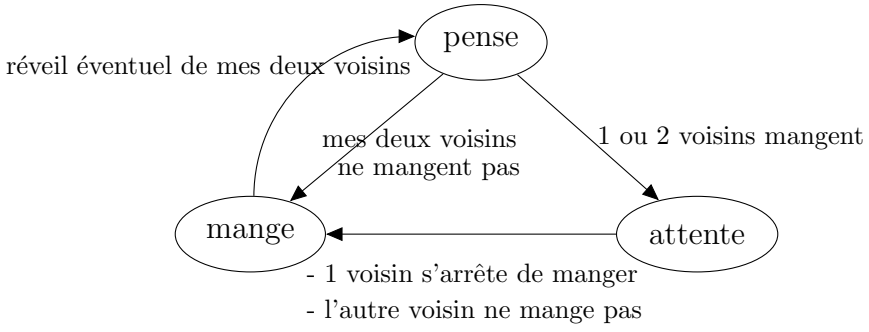
Enoncé : Soient cinq philosophes réunis autour d'une table pour philosopher et manger.



Au moment du repas un problème pratique à résoudre se pose. En effet, le repas est composé de nouilles qui se mangent avec deux baguettes. Or la table n'est dressée qu'avec une baguette par couvert. Les philosophes décident d'adopter le rituel suivant :

- Chaque philosophe prend place sur une chaise,
- Chaque philosophe qui mange utilise la baguette à sa gauche et à sa droite,
- A tout instant, chaque philosophe se trouve dans l'un de états suivants :
  - Il mange avec deux baguettes,
  - Il attend la baguette de droite, ou de gauche ou les deux,
  - Il pense, sans utiliser de baguette.

Nous obtenons alors le graphe de changement d'état suivant :



Au départ, tous les philosophes pensent et tous les philosophes cessent de manger au bout d'un temps fini.

Le problème est de définir les prologue et épilogue afin que les philosophes ne rencontrent pas d'interblocage.

Formalisation du problème :

— Chaque philosophe est un processus,
— Tous les processus utilisent le même algorithme,
— Les états des processus sont les suivant :
  — $C(i) = 0$, le philosophe i pense,
  — $C(i) = 1$, le philosophe i veut manger mais manque de baguette(s),
  — $C(i) = 2$, le philosophe i mange.
— Transitions possibles pour l'allocation des baguettes sont :
  — $C(i) = 0 \rightarrow C(i) = 2$, si $C(i+1) \neq 2$ **et** $C(i-1) \neq 2$
  — $C(i) = 0 \rightarrow C(i) = 1$, sinon.

Cette stratégie met en oeuvre une allocation globale des baguettes. Le tableau des états C est partagé entre les différents philosophes : ils peuvent en consulter le contenu et aussi le mettre à jour. Aussi, pour que les philosophes qui utilisent ce tableau dans une section critique puisse accéder à une valeur cohérente de ce tableau, il est nécessaire de le protéger par un sémaphore d'exclution mutuelle : `S(Mutex)`.

Une première façon de définir les algorithmes des prologue et épilogue pourrait être donnée par les algorithmes suivants (algo. 2.16 et 2.17) :

Algorithme 2.16 : Prologue (pour le philosophe i noté φ_i)

```
P(Mutex) /* l'accès à la variable d'états des processus se fait en exclusion
   mutuelle */
si C(φ_{i-1}) ≠ 2 et C(φ_{i+1}) ≠ 2 /* les voisins ne mangent pas */
alors
 | C(φ_i) ← 2 /* mange */
sinon
 | C(φ_i) ← 1 /* attente */
finsi
V(Mutex)
```

Algorithme 2.17 : Epilogue (pour le philosophe i noté φ_i)

```
P(Mutex) /* l'accès à la variable d'états des processus se fait en exclusion
   mutuelle */
C(φ_i) ← 0 /* pense */
si C(φ_{i-1}) = 1 et C(φ_{i-2}) ≠ 2
alors
 | C(φ_{i-1}) ← 2 /* mange */
finsi
si C(φ_{i+1}) = 1 et C(φ_{i+2}) ≠ 2
alors
 | C(φ_{i+1}) ← 2 /* mange */
finsi
V(Mutex)
```

Mais cette implantation présente un problème. En effet, les philosophes qui ne peuvent avoir de baguette(s) mettent leur variable d'état à *attente* mais ils n'attendent, ils ne se bloquent pas. De plus, quand un philosophe termine de manger, il ne réveille pas explicitement un autre philosophe, il lui change son état sans l'avertir. Il faut alors introduire la notion de sémaphore privé pour gérer les blocages et réveils des philosophes.

Ainsi, chaque philosophe posséde un sémaphore privé que nous noterons pour plus de convenance sous

forme de tableau `S(`φ_i`)`, qui sera associé à l'état *mange*. Ce sémaphore, comme la plupart des sémaphores privés, agit sur le processus lui-même et non sur une ressource (par exemple la variable d'état C). Nous obtenons alors les algorithmes de prologue et d'épilogue 2.18 et 2.19.

Algorithme 2.18 : Prologue (pour le philosophe i noté φ_i)

P(Mutex) `/* l'accès à la variable d'états des processus se fait en exclusion mutuelle */`
si $C(\varphi_{i-1}) \neq$ *mange* **et** $C(\varphi_{i+1}) \neq$ *mange*
alors
  C(φ_i) $\leftarrow$ 2 `/* mange */`
  V(S(φ_i)) `/* on peut manger donc on se donne notre propre jeton */`
sinon
  C(φ_i) $\leftarrow$ 1 `/* attente */`
finsi
V(Mutex)
P(S(φ_i)) `/* on se bloque si en attente sinon on passe car on a déjà notre jeton */`

Algorithme 2.19 : Epilogue (pour le philosophe i noté φ_i)

P(Mutex) `/* l'accès à la variable d'états des processus se fait en exclusion mutuelle */`
C(φ_i) $\leftarrow$ 0 `/* pense */`
si $C(\varphi_{i-1}) = 1$ **et** $C(\varphi_{i-2}) \neq 2$
alors
  C(φ_{i-1}) $\leftarrow$ 2 `/* mange */`
  V(S(φ_{i-1})) `/* réveil de` φ_{i-1} `*/`
finsi
si $C(\varphi_{i+1}) = 1$ **et** $C(\varphi_{i+2}) \neq 2$
alors
  C(φ_{i+1}) $\leftarrow$ 2 `/* mange */`
  V(S(φ_{i+1})) `/* réveil de` φ_{i+1} `*/`
finsi
V(Mutex)

Les différentes variables sont intialisées ainsi :
— C : le tableau des états, tous intialisés à $C(i) = 0$: les philosophes pensent,
— $S(\varphi_i)$: les sémaphores privés de synchronisation, initialisés à 0,
— $Mutex$: le sémaphore d'exclusion mutuelle de protection de C, initialisé à 1.

Algorithme 2.20 : Corps d'instructions de chaque philosophe

penser
demander les baguettes `/* appel de prologue */`
manger
libérer les baguettes `/* appel d'épilogue */`
recommencer

Il faut faire très attention à l'utilisation d'une primitive P à l'intérieur d'une section critique. Car si un processus est entré en section critique protégée par une exclusion mutuelle, aucun autre processus ne pourra entrer dans une section critique quelconque protégée par la même exclusion mutuelle. Et donc peut être qu'aucun autre processus ne pourra débloquer ce processus si celui-ci est bloqué sur la primitive P.

Comme les processus bloqués ne peuvent se réveiller eux-mêmes, c'est bien les autres processus qui vont les réveiller par l'intermédiaire de la primitive V.

Ainsi, dans les deux algorithmes utilisant les sémaphores privés, pour éviter le blocage des processus avant d'avoir relâcher l'exclution mutuelle sur la variable C, on utilise la propriété de mémorisation des sémaphores pour se donner d'avoir le droit de passage.

2.7 Les moniteurs

Les moniteurs sont une structure qui a été proposée par Hoare en 1972 afin de remédier au principal inconvénient des sémaphores : le manque de lisibilité du code. En effet, les primitives P et V sont dispersées dans le texte et il est donc difficile de s'assurer qu'il n'y aura pas d'interblocage et que les programmes se termineront de façon sûre.

2.7.1 Composition

Un moniteur est composé :
— De variables de synchronisation appelées *varaiables_conditions*,
— De ressources partagées,
— De procédures d'accès aux ressources :
  — Externes : primitives utilisées par les processus (prologue/épilogue),
  — Internes : des procédures réservées au moniteur pendant l'exécution des prologues et épilogues).

Comme pour les sémaphores, les valeurs des variables internes ne sont pas accessibles aux processus mais uniquement au procédure du moniteur.
De plus, les moniteurs sont communs à tous les processus donc leur accès se fera en exclusion mutuelle.

2.7.2 Procédures internes : primitives de synchronisation

Les moniteurs proposent deux procédures qui sont appelées dans les primitives externes d'accès aux ressources :
— **WAIT** :
  — met en attente le processus qui fait WAIT sur la *variable_condition* (équivalent du P des sémaphores),
  — libère le moniteur (très important, bloque le processus, pas le moniteur),
— **SIGNAL** :
  — réveil et active immédiatement un processus en attente sur la *variable_condition* s'il en existe un,
  — suspend le processus qui vient d'exécuter SIGNAL si un processus en attente vient d'être réveillé.
  Attention : SIGNAL ne mémorise pas la demande de réveil et donc celui-ci est perdu si aucun processus n'attendait (grosse différence avec le V des sémaphores).

Pour utiliser ces procédures, la convention suivante a été prise :

variable_condition.WAIT

variable_condition.SIGNAL

2.7.3 Procédures externes

L'appel à un moniteur par un processus se fait également selon la syntaxe objet suivante :

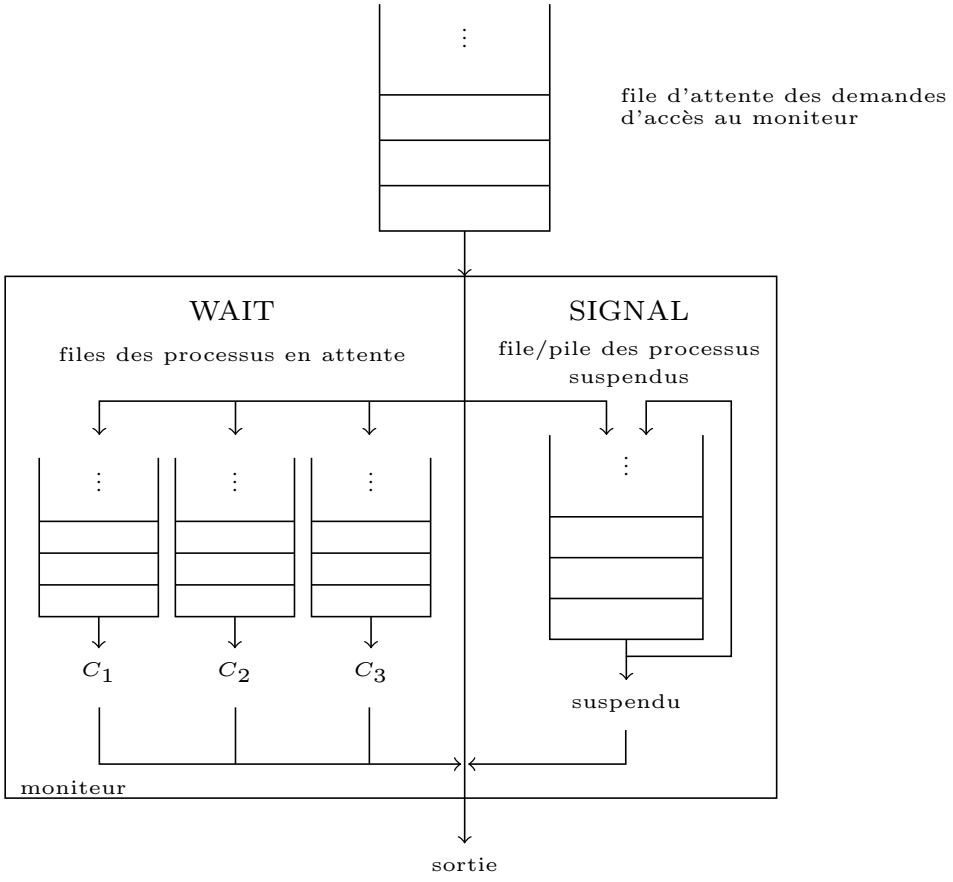
nom_moniteur.nom_procédure(arguments)

Ces procédures correspondent aux procédures de prologue et d'épilogue pour accéder à des ressources à l'aide d'un moniteur.

2.7.4 Modélisation à l'aide de files d'attente

Ce modèle est composé de :
— Une file pour l'accès au moniteur,
— n files d'attente pour les processus, une file par *variable_condition* C_i,

— Une file ou un pile pour les processus suspendus par SIGNAL.



Remarques :

— Un processus en attente sur une condition libère l'exclusion mutuelle de l'accès au moniteur,
— Le processus suspendu après l'exécution de SIGNAL ne libère pas l'exclusion mutuelle dans le moniteur car :
  — Soit un processus est en attente de cette condition et reprend son exécution (et reprend donc l'exécution dans une procédure du moniteur et doit donc le posséder),
  — Soit c'est le même processus qui poursuit son exécution si aucun processus n'a été réveillé.
  Dans tous les cas, aucun nouveau processus ne rentre dans le moniteur. La file des processus suspendus est "prioritaire" par rapport aux processus de la file d'entrée au moniteur.
— Un processus suspendu après SIGNAL sera débloqué lorsqu'il sera en tête de file des processus suspendus et que le processus actif dans le moniteur quitte le moniteur ou bien se met en attente sur une *variable_condition*.

Au cours de son exécution et de l'utilisation du moniteur, un processus peut ainsi se retrouver dans les états suivants :
— normal : le processus est seul actif dans le moniteur,
— attente pour entrer dans le moniteur (exclution mutuelle pour l'accès),
— attente dans la file d'une *variable_condition* (après la commande WAIT),
— suspendu après exécution de la commande SIGNAL si un processus a été réveillé.

2.7.4.1 Exemple de gestion d'une ressource unique

Tout d'abord on défini la variable moniteur avec ses composantes : son corps et les deux primitives externes *Acquérir* et *Libérer*.

Algorithme 2.21 : ressource_unique : moniteur

Données :
libre : booléen
libération : condition
début
  libre ← vrai `/* corps du moniteur */`
fin

Procédure `Acquérir`

début
  si *non libre* **alors**
    libération.WAIT `/* bloque le processus et libére le moniteur pour éviter l'interblocage */`
  finsi
  libre ← faux
fin

Procédure `Libérer`

début
  libre ← vrai
  libération.SIGNAL
fin

Pour utiliser le moniteur de cette ressource critique unique, un processus fera selon le shéma suivant :

Les moniteurs sont des objets de plus haut niveau que les sémaphores, qui sont eux-mêmes de plus haut niveau que les verrous.

Voici un exemple d'exécution de deux processus utilisant un moniteur pour l'accès à une ressource critique. Soient deux processus P_1 et P_2 qui s'exécutent en parallèle et sont concurrents pour la ressource unique protégée par le moniteur. Au départ ils sont également concurrents pour l'entrée dans le moniteur. On supposera que P_1 entre le premier dans le moniteur pour la procédure *Acquérir*.

Algorithme 2.24 : Algorithme d'un processus utilisant un moniteur

```
...
ressource_unique.Acquérir
...
section critique
...
ressource_unique.Libérer
...
```

P_1		P_2
ressource_unique.Acquérir		ressource_unique.Acquérir
entrée moniteur		... blocage ...
libre ← faux		*zzz*
sortie moniteur	→ réveil de P_2 →	entrée moniteur
*		libération.WAIT
section critique		libérer le moniteur
avec la ressource		... attente ...
*		*zzz*
ressource_unique.Libérer		*zzz*
entrée moniteur		*zzz*
libre ← vrai		*zzz*
libération.SIGNAL	→ réveil immédiat →	reprend le moniteur
suspendu		libre ← faux
sortie moniteur	← réactivation ←	sortie moniteur
*		*
*		section critique
*		avec la ressource
*		*
*		ressource_unique.Libérer
*		entrée moniteur
*		libre ← vrai
*		libération.SIGNAL
*		sortie du moniteur
*		*

Chapitre 3

Communication

La communication et la synchronisation sont deux domaines fortement liés. En fait, la synchronisation est une forme élémentaire de communication où est échangé un jeton.

La plupart des mécanismes de communication passent par l'intermédiaire de variables communes afin de s'échanger des informations. Ainsi, un protocole de communication classique se compose de l'enchaînement d'actions suivant :

— Synchronisation nécessaire aux échanges,
— Section critique pour l'accès aux données communes,
— Emission ou réception des valeurs échangées.

Nous allons maintenant voir plusieurs modèles de communication plus ou moins simples.

3.1 Modèle du producteur-consommateur simple

Le modèle du producteur-consommateur met en oeuvre une communication entre deux processus basée sur l'existence et la valeur de deux ressources duales. Par exemple, le nombre de cases pleines et le nombre de cases vides d'un même buffer, tampon d'échange.

La partie synchronisation du protocole est basée sur les sémaphores attachés aux deux ressources et la section critique est limitée à l'opération P sur ces sémaphores.

Nous allons présenter ci-après trois exemples d'implémentation de ce protocole.

3.1.1 Producteur-consommateur de ressources élémentaires

Soient deux processus : un producteur et un consommateur. Le producteur produit des bonbons, le consommateur mange les bonbons. La ressource critique n'est pas directement le nombre de bonbons mais surtout le nombre de papier de bonbons qui est limité à N.

Il faut donc introduire deux sémaphores pour protéger ces deux ressources :

— Le sémaphore *papier*, initialisé à N,
— Le sémaphore *bonbon*, initialisé à 0.

Les algorithmes respectifs de ces deux processus sont assez simples :

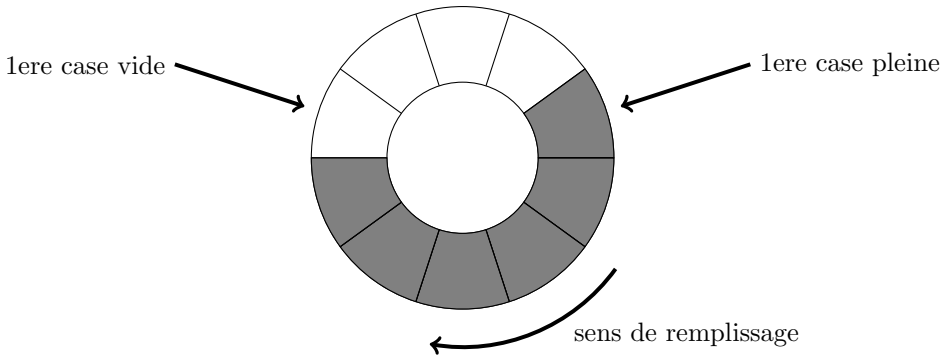
Producteur	Consommateur
P(papier) fabrique un bonbon V(bonbon)	P(bonbon) mange un bonbon V(papier)

Dans ce cas, il n'y a pas d'interblocage possible car les deux processus ne sont jamais concurrents sur la même ressource et l'exclusion mutuelle est garantie.

3.1.2 Producteur-consommateur de messages dans un tampon circulaire

Dans cet exemple, nous supposons que le tampon est de forme circulaire. Aussi, nous devons introduire deux autres ressources pour gérer le tampon : un pointeur vers la première case vide et un pointeur vers la première case pleine. Au départ les deux pointeurs ont la même valeur : 0.

A la place des bonbons et papier nous avons cette fois comme ressource de base le nombre de cases vides et le nombre de cases pleines, sachant que les deux sont liées (duales) par le fait que le nombre de cases vides + le nombre de cases pleines = N. Nous avons donc deux sémaphores : *case_vide* et *case_pleine*, initialisés respectivement à N et 0.



Les algorithmes respectifs de ces deux processus sont les suivants :

Producteur	Consommateur
P(case_vide)	P(case_pleine)
range le message dans T[pointeur_vide]	lit le message de T[pointeur_plein]
pointeur_vide ++	pointeur_plein ++
V(case_pleine)	V(case_vide)

Dans ce cas aussi il n'y a pas d'interblocage car les deux processus ne sont toujours pas concurrents sur les mêmes ressources manipulées. Comme il n'y a qu'un seul producteur, il est le seul à manipuler la variable *pointeur_vide* et il n'est donc pas utile de la protéger par un sémaphore d'exclusion mutuelle. Il en est de même pour le consommateur avec la variable *pointeur_plein*. Quand il n'y a plus de cases vides, le producteur se bloque, et s'il n'y a plus de cases pleines, le consommateur se bloque.

3.1.3 Producteur-consommateur géré par un moniteur

Dans cet exemple, la synchronisation et la section critique sont assurées par l'utilisation d'un moniteur. Pour cela, prenons le moniteur de nom *moniteur*, il faut alors définir ses variables internes, les variables conditions et les deux primitives externes d'accès au moniteur.

Algorithme 3.1 : moniteur

Données :
occupé : entier
non_plein, non_vide : condition `/* Les variables conditions */`
début
  occupé ← 0 `/* corps du moniteur */`
fin

Procédure `Déposer(`*message M*`)`

début
  si *occupé = N* **alors**
    non_plein.WAIT
  finsi
  ranger M dans le tampon
  occupé ← occupé + 1
  non_vide.SIGNAL
fin

Procédure `Prélever(`*message M*`)`

début
  si *occupé = 0* **alors**
    non_vide.WAIT
  finsi
  prendre M du tampon
  occupé ← occupé - 1
  non_plein.SIGNAL
fin

Les algorithmes des processus utilisant le moniteur est assez simple :

Producteur	Consommateur
produire un message M moniteur.Déposer(M)	moniteur.Prélever(M) consommer le message M

3.2 Modèle du producteur-consommateur multiple

Soient N producteurs et M consommateurs concurrents. Il faut cette fois-ci s'assurer de l'accès en exclusion mutuelle aux variables à la fois entre producteurs et respectivement entre consommateurs. En effet, dans ce cas de figure, plusieurs producteurs (respectivement consommateurs) peuvent vouloir accéder à la même case.

L'exclusion mutuelle portera donc sur les variables partagées, c'est-à-dire les pointeurs de la première case vide et de la première case pleine. Nous obtenons alors les algorithmes suivants :

Producteur	Consommateur

P(case_vide)
  P(Mutex_pointeur_vide)
  ranger le message dans T[pointeur_vide]
  pointeur_vide++
  V(Mutex_pointeur_vide)
V(case_pleine)
	P(case_pleine)
	  P(Mutex_pointeur_plein)
	  prendre le message de T[pointeur_plein]
	  pointeur_plein++
	  V(Mutex_pointeur_plein)
	V(case_vide)

Dans les deux cas, producteur ou consommateur, l'ordre des primitives $P()$ est primordiale. En effet, dans l'ordre P(case_vide) P(Mutex_pointeur_vide), les cases libres peuvent être réservées par les différents producteurs mais ils seront ensuite concurrents pour l'exécution de la section critique. Les producteurs traiteront l'un après l'autre le pointeur et donc une case différente.

Dans l'ordre inverse, P(Mutex) puis P(case_vide), un seul producteur rentre à la fois dans la section critique et là se bloque éventuellement sur une demande de case vide. Les autres producteurs ne peuvent pas réserver de case vides, ils sont bloqués avant.

En revanche, en général, l'ordre des instructions V n'est pas important car ces primitives ne sont pas bloquantes. Soit on prévient d'abord qu'un message est prêt, soit on relâche d'abord le pointeur.

3.3 Communication par boîte aux lettres

La boîte aux lettres est une variable commune à tous les processus qui vont communiquer. Elle est privé au propriétaire. Le principe de fonctionnement est très similaire au fonctionnement des sémaphores privés :

— Seul le propriétaire a le droit de la lire,
— Tous les autres processus peuvent y écrire.

Les files de messages réalisent des boîtes aux lettres partagées avec des droits d'accès distincts entre les différents processus utilisateurs.

3.4 Sémaphores à messages

On complète les primitives de synchronisation P et V par l'émission et la réception de messages. Ainsi, un sémaphore à message est représenté par :

— Un entier,
— Une file d'attente de processus,
— Une file de messages.

les primitives ont alors la forme suivante :

$P(s,\ message\_recu)$: Si le sémaphore est passant, alors le premier message de la file est prélevé et est transmis au processus dans la variable *message_recu*.

$V(s,\ message\_emis)$: Le message contenu dans la variable *message_emis* est ajouté à la file de message.

3.5 Processus séquentiels communiquants

Le principe du modèle CSP (Communicating Sequential Process) proposé par Hoare en 1977 est de réaliser :

— La synchronisation,
— La communication,
— L'exclusion mutuelle.

Avec uniquement le mécanisme du rendez-vous. Plusieurs langage mettent en oeuvre ce principe comme OCCAM et ADA.

Chapitre 4

Vie d'un processus, synchronisation, communication sous Unix

Dans ce chapitre, nous allons nous concentrer sur les mécanismes offerts par Unix pour la synchronisation et la communication entre processus. Une grande partie des notions précédemment vues concernant ces deux domaines sont directements implantées sous Unix dont principalement les sémaphores. Toutefois, Unix offre encore d'autres outils intéressants comme les tubes, les zones de mémoire partagée ou encore les files de messages.

Cependant, ces outils ne sont utiles que si nous sommes dans le cadre de processus multiples, or comment peut-on créer des processus sous Unix ? La première partie de ce chapitre répond à cette question en abordant les deux principaux aspects de la vie d'un processus : la création et la destruction.

4.1 Vie des processus

Avant d'exister, un processus doit d'abord résoudre une étape liée à l'existence de deux identités : l'utilisateur actuel qui veut exécuter le processus et le propriétaire du fichier contenant le programme à exécuter. Les deux ne sont pas forcément identiques. En effet, il suffit d'observer le répertoire `/bin/` : le propriétaire des exécutables présents est l'utilisateur `root` mais, heureusement, n'importe quel utilisateur peut tout de même les exécuter.

Ainsi, un processus doit conserver l'information liée à l'utilisateur ayant lancé le processus mais aussi des attributs relatifs au propriétaire du fichier avec ses droits. Les attributs d'`uid` et de `gid` (user id et group id) jouent ce rôle.

De plus, au court de son exécution, le processus peut être amené à changer d'identité et, par exemple, d'endosser temporairement le rôle du propriétaire. Lors de l'exécution du programme, le système distingue alors l'`uid` de l'utilisateur en séparant l'`ruid` et le `rgid` qui représente l'id réel de l'utilisateur, de l'utilisateur effectif `euid` et `egid` s'il y a endossement temporaire de privilèges.

Cette élévation de privilèges n'est possible que si l'attribut spécial `suid` (*set-uid bit*) est activé pour ce fichier (ou son groupe) par son propriétaire. Plus précisément :

- Le bit `setuid` autorise l'exécution du fichier en prenant les droits du propriétaire du fichier,
- Le bit `setguid` autorise l'exécution du fichier en prenant les droits du groupe du propriétaire du fichier.

Il est possible de connaître si un programme possède le `setuid` bit en affichant les privilèges de ce programme dans le système de fichiers : la lettre `s` apparaît dans les champs `x` adéquats.

Dans le système Unix, un processus possède de nombreux autres attributs liés à sa gestion par le système. Dans le cadre de notre analyse de création et destruction d'un processus, nous allons simplement retenir ceux-ci :

- Un identifiant unique : le `pid`,
- Une image mémoire (*core image*) qui contient toutes les zones mémoires privées du processus : texte exécutable, données, piles ...
- Une liste des fichiers ouverts,
- Les privilèges liés à l'utilisateur.

4.1.1 Création dynamique de processus : fork()

La commande `fork()` est un mécanisme qui permet de dupliquer totalement l'image mémoire du processus qui exécute cette commande pour créer un nouveau processus que l'on peut voir comme un clone de celui-ci. Le processus exécutant la commande `fork` est appelé *processus père* et le nouveau processus créé est appelé *processus fils*. Par le biais de cette commande, le processus fils va hériter de tous les attributs du processus père :

— Les privilèges,
— Les fichiers ouverts,
— Le texte exécutable,
— les valeurs des variables.

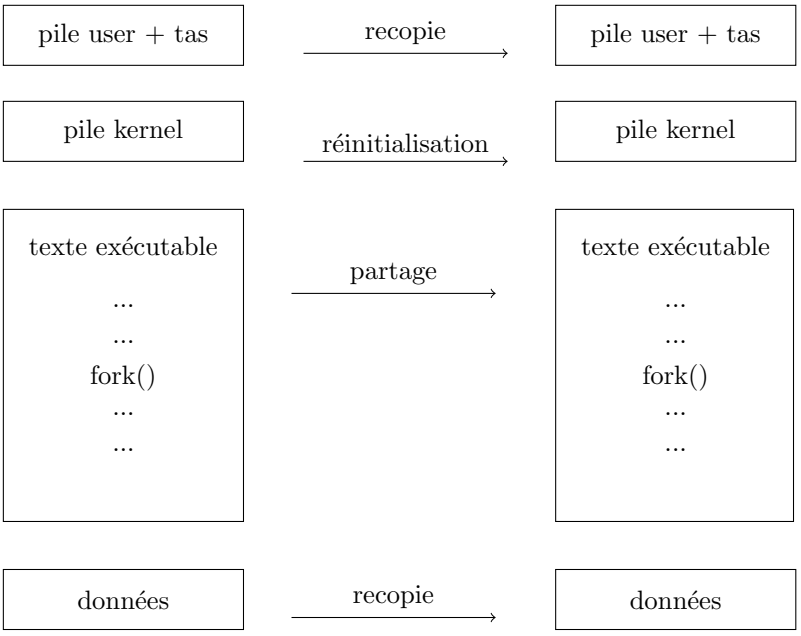
Comment alors distinguer le processus père du processus fils s'ils sont identiques et exécutent exactement le même code ? En fait, la commande fork fournit une valeur de retour différente selon que le processus est le processus père ou le processus fils.

— Si le processus est le processus père, il reçoit le pid du processus fils créé.
— Si le processus est le processus fils, `fork` retourne la valeur `0`.

Aussi, la commande `fork()` est souvent utilisée conjointement avec un test conditionnel sur le retour de cette fonction comme le montre l'exemple suivant :

```
...
/* instructions exécutées par le seul processus existant (le père) */
...
if ((pid = fork()) == 0)
    { /* instructions exécutées uniquement par le fils */ }
else
    { /* instructions exécutées uniquement par le père */ }
...
/* instructions exécutées par les deux processus */
...
```

Le compteur ordinal du processus fils est initialisé à l'adresse suivant l'instruction `fork()`. Le texte exécutable n'est pas dupliqué mais partagé. Il est d'ailleurs partagé entre tous les processus exécutant le même programme. Le schéma suivant illustre de manière simple ce mécanisme de duplication.



Nous pouvons remarquer que dans l'algorithme de l'instruction `fork` (cf. algo 4.1) la procédure `procdup()` possède deux instructions de retour avec des valeurs différentes mais non conditionnel. En fait, quand le père recopie le son contexte d'exécution dans celui du son fils, il met à jour le compteur ordinal du fils et aussi du père. Par ce biais, il indique que la prochaine instruction du fils à exécuter sera le premier return alors que la prochaine instruction du père est l'instruction suivant ce return. Ainsi le père n'effectuera jamais ce return et continuera les instructions suivantes nécessaires à la création du fils.

Par ailleurs, une fois que le père a mis l'état du fils dans le mode `SRUN`, celui-ci est ordonnancable. Toutefois, le processus père étant en mode kernel, il n'est pas préemptible. Le fils ne pourra vraiment obtenir l'unité centrale que lorsque le processus père quittera le mode kernel, c'est-à-dire à la fin de l'exécution de `fork()`.

Rappel : l'exécution d'un processus est séparée en deux modes complémentaires mais séparés : le mode utilisateur (*user*) et le mode noyau (*kernel*). Le mode utilisateur concerne l'exécution d'instructions appartenant à l'espace mémoire du processus, le mode noyau correspond à l'exécution d'instructions externes (librairies chargées) qui doit être protégée et gérée par le système d'exploitation.

4.1.2 Terminaison de processus : exit()

Le moyen propre et normal pour un processus de terminer son exécution est d'utiliser la commande `exit()`. Cette commande permet deux choses : terminer le processus et envoyer une message à son processus père. Cette valeur est donnée en paramètre de l'appel à `exit` et permet donc de donner un indication sur la raison de la terminaison de celui-ci.

L'utilisation classique est donc celle-ci : `exit(status)`, avec généralement la valeur 0 quand la raison de la terminaison est normale.

Il est important et obligatoire qu'à tout moment un processus possède un processus père. Dans le cas contraire on ne pourrait plus y accéder par les liens de chaînage et les structures de données ne seraient plus cohérentes. La seule exception est le processus **init** de valeur 1.

Pour résumer, la fonction `exit()` libère et efface tout du processus **sauf** la structure **proc**.

4.1.3 Recouvrement de processus : exec()

La famille des fonctions `exec()` (`execl()`, `execlp()`, `execle()`, `execv()`, `exevp()`, `execvpe()`) effectue ce que l'on appelle un recouvrement de l'image mémoire du processus exécutant `exec()` par le texte exécutable issu d'un fichier tout en réinitialisant la zone de données. A la différence de `fork`, il n'y a pas de création de processus. Le code programme exécutant l'instruction `exec()` est simplement "effacé" et "remplacé" par un autre. En général, on ne revient plus jamais au processus à partir duquel a été exécuté l'instruction `exec()`, le compteur ordinal est réinitialisé au début du programme appelé. Le cas de retour au programme appelant est en cas d'impossibilité d'exécuter `exec()`.

L'instruction `exec()` est généralement utilisée conjointement avec l'instruction `fork` : tout d'abord il y a création d'un nouveau processus avec `fork` puis exécution d'un autre programme dans le processus fils avec `exec()`.

Les fonctions `exec()` prennent comme paramètre le nom du fichier à charger (et donc exécuter) ainsi qu'un ensemble d'arguments pour ce nouveau programme. Par exemple :
`execl(nom_fichier, nom_fichier, arg1, ..., NULL);`. Nous pouvons remarquer qu'il est indiqué deux fois `nom_fichier`. Le premier paramètre défini en fait le nom (avec chemin absolu) du fichier à charger contenant le programme. Le second correspond en fait à `argv[0]` comme dans les shell ou en C, qui contient le nom du programme exécuté.

La différence entre les différentes instructions `exec()` est la suivante :

- Les fonctions du type `execl()` prennent les paramètres sous forme de liste. Cette liste étant de longueur indéterminée (`exec()` ne peut pas connaître à l'avance le nombre de paramètre du programme qu'il chargera) un marqueur de fin de liste est utilisé : NULL.
- Les fonctions du type `execv()` prennent les paramètres sous forme de tableau (exactement comme `argv[]`). Pour la même raison que pour les listes, la taille du tableau ne peut être connue. Le marqueur NULL est donc également utilisé comme dernier élément du tableau.

Algorithme 4.1 : fork()

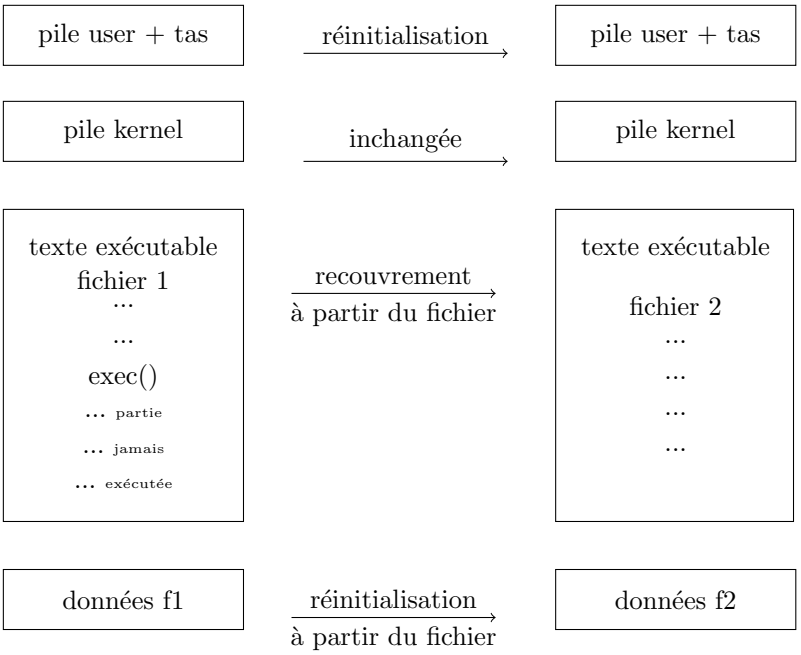
```
début
    appel de swapexpand /* réserve de la place sur le swap pour la pile et le données du
       fils qui va être créé */
    compter le nombre de processus de l'utilisateur
    récupérer une entrée libre dans la table proc /* via la liste libre */
    test anti-panique si l'entrée est bien libre
    si plus d'entrée libre alors
        erreur tablefull
    finsi
    si (nombre processus < limite user) ou (dernière entrée et processus root) alors
        sortir l'entrée proc de la liste libre
        initialiser l'état du fils à SIDL
        appel à la fonction newproc()
        début
            chercher un PID pour le fils
            chaîner le fils sur les listes (père, frère, terminal)
            hériter (masques et flags pour le traitement des signaux, priorité, UID et GID, job control (^Z,
            fg, bg, ...), taille des zones de données)
            RAZ des horloges du fils (user, système, ...)
            incrémenter le nombre de fichiers ouverts dans la table des fichiers du système (dont les
            tubes/pipe)
            incrémenter le nombre d'utilisateur du répertoire courant
            positionner l'état du père à SKEEP /* le processus est conservé en mémoire centrale,
               non swappable */
            appel à la fonction procdup()
            début
                duplication des espaces du père pour le fils (espace virtuel, zones privées)
                récupérer une entrée U_area (table user) pour le fils
                copier la zone U_area du père vers celle du fils
                création et recopie de la pile kernel du père vers le fils
                copie du context (savectxt()) du père vers le fils avec mise à jour du compteur ordinal
                et des registres d'espace mémoire pour le fils
                retourner 1                                    ← compteur ordinal du fils
                RAZ des statistiques de la mémoire virtuelle du fils   ← compteur ordinal du père
                recopie des données et des piles user du père vers le fils
                retourner 0
            fin
            si fils (valeur de retour de procdup() = 1 alors
                déverrouiller les piles
                retourner 1
            finsi
            si père (valeur de retour de procdup() = 0 alors
                mettre l'état du fils à SRUN
                mettre le fils dans la runqueue pour qu'il puisse être activé
                mettre le fils swappable
                mettre le père swappable (not SKEEP)
                retourner 0
            finsi
        fin
        si fils (valeur de retour de newproc() = 1) alors
            initialiser l'horloge départ
            initialiser le nombre de ticks
            retourner 0
        finsi
        si père (valeur de retour de newproc() = 0) alors
            retourner PID
        finsi
    finsi
fin
```

Algorithme 4.2 : exit()

Données : status : une valeur de retour sur la raison de la terminaison
début
marquer le processus comme insensible aux signaux
RAZ de tous les timers
mettre le processus dans l'état **SZOMB**
fermer les fichiers ouverts par le processus (et aussi les tubes)
décrémenter les compteurs de la table des fichiers ouverts du système
décrémenter le nombre d'utilisateur du répertoire courant
libérer le terminal attaché au processus
libérer la mémoire virtuelle, la mémoire physique, la zone **U_area** et la pile kernel
sortir le processus de la file des processus prêts et le mettre dans la file des Zombies
pour *tous les fils de ce processus* **faire**
faire adopter ce fils par init (processus numéro 1)
finpour
stocker la valeur de status dans la structure **proc** du processus
envoyer le signal **SIGCHLD** au père `/* qui sera réveillé en cas d'attente */`
effectuer un switch() `/* changement de contexte mais partiel, pas de sauvegarde du contexte actuel, on lance un autre processus */`
fin

— Les fonctions du type `exec*p()` recherchent le nom du programme donné en paramètre relativement à la variable `PATH`. Par contre, si un caractère `/` apparaît dans le nom du fichier, ce nom est considéré comme un chemin absolu et donc il n'y aura pas de recherche dans le `PATH`.
— Les fonction du type `exec*e()` prennent en plus des arguments de type `argv` un ensemble de variables d'environnement sous forme de liste (`le`) ou de tableau (`ve`).

Le schéma donné dans la partie `fork` est modifié comme suivant :



L'algorithme 4.3 décrit brièvement les principales étapes de la fonction `exec()` dont deux qui sont importantes à noter :
— La gestion des droits d'exécution,
— La sauvegarde des paramètres d'`exec()`.

En effet, selon le positionnement du *sticky bit* sur le programme à charger, l'utilisateur effectif et réel peuvent être modifiés ce qui implique potentiellement des droits différents. Dans les deux cas, les droits réels sont comparés aux droits nécessaires associés au fichier à exécuter.

La deuxième étape vient simplement du fait que les données du processus courant vont être remplacées par celles du nouveau processus. Elles seront donc en quelques sortes effacées. Ainsi, le nouveau processus ne pourra accéder aux paramètres de la commande `exec()`. Le moyen utilisé pour contrer ce problème consiste à simplement mettre ces valeurs sur le swap et de lié cette zone du swap avec le nouveau processus. Une fois le nouveau processus prêt à pouvoir être exécuté, les paramètres sauvegardés sur le swap sont de nouveau positionnés dans le nouvel espace mémoire correspondant à ces données (`argc`, `argv` et environnement).

4.2 Synchronisation

4.2.1 Synchronisation père-fils : wait()

Nous avons vu que la fonction `exit` permet à un processus de se terminer et d'envoyer une valeur à son processus père afin de l'informer de la raison de sa terminaison. En fait, en même temps qu'il se termine, le processus fils envoie également un signal particulier au processus père afin de le prévenir qu'il se termine. Cet envoie de signal est en fait une forme de synchronisation que le père peut attendre via la fonction `wait()`.

En effet, en exécutant la commande `wait`, le processus père se met explicitement en attente de la terminaison d'un de ses processus fils. En retour, la fonction `wait` retourne le `pid` du fils qui s'est terminé mais également la valeur de status de celui-ci.

Cependant, comme il n'est pas possible à une fonction de retourner deux valeurs, la valeur de status est retournée à travers l'argument de `wait`. L'utilisation est la suivante : `pid = wait(&status)`.

Toutefois, la valeur qui sera placée dans `status` ne sera pas directement la valeur de status qu'a envoyé le processus fils, seuls les 8 bits de poids faibles correspondent à celui-ci. Pour récupérer cette valeur, il suffit d'utiliser la macro `WEXITSTATUS(status)`.

Il faut noter que sur une attente via `wait`, dès la terminaison d'un processus fils, quelqu'il soit, le processus père sera réveillé.

Par ailleurs, comme le montre l'algorithme 4.4, on ne sort de la fonction `wait()` que par une instruction **retourne(PID du fils)** ou **retourne(ECHILD)** (indiquant une erreur = -1)

4.2.2 Synchronisation par évènement : kill, pause, signal

La synchronisation qui existe entre un processus père et un processus fils par rapport à la terminaison de celui-ci est en fait un cas particulier d'un mécanisme de synchronisation offert par Unix : la synchronisation directe par évènement.

Ce mécanisme est très similaire au mécanisme des interruptions et on appelle parfois ce mécanisme "interruption logicielle". Les bases de ce mécanisme sont la sensibilité d'un processus à l'arrivée de certains événements et d'effectuer le cas échéant une série d'instructions spécifiques à cet événement. Le processus sensible ne doit pas forcément être endormi et attendre cet événement. De manière générale, l'événement peut arriver n'importe quand et ce système de gestion est dit asynchrone.

Les systèmes fournit principalement les trois instructions de base suivantes :

— `kill` : envoie un signal (événement) particulier à un processus particulier désigné par son pid (ou un groupe de processus si la valeur est négative).

— `signal` : associe une fonction à la réception d'un signal,

— `pause` : met en pause le processus en attente d'un événement quelconque.

Les signaux ou événements qu'il est possible d'envoyer sont prédéfinis par le système et sont ordonnés par priorité (`/usr/include/bits/signum.h` sous Linux). Il existe une trentaine de signaux dont 2 sont laissés à l'utilisateur et n'ont pas de sens particulier pour le système. Ces signaux peuvent être désignés soit par leur numéro soit par leur nom. Les valeurs de signaux pouvant varier d'un système à un autre, il est préférable d'utiliser les noms. Voici un sous-ensemble des signaux définis sous unix :

```
#define SIGHUP          1       /* Hangup (POSIX).  */
#define SIGINT          2       /* Interrupt (ANSI).  */
```

Algorithme 4.3 : exec()

```
Données : nom du fichier à exécuter, argv : tableau des paramètres
début
    /* mise-à-jour des UID et GID selon les droits du fichier/propriétaire/utilisateur  */
    si le bit X est positionné alors
        recopie des valeurs du processus effectuant l'exec
    sinon
        si le bit S est positionné alors
            endossement des valeurs liées au fichier
        finsi
    finsi
    trouver le fichier et lire de son magic number en entête
    /* le magic number permet de distinguer le type de fichier: exécutable ou shell script
       */
    /* sauvegarde de ce qui va être écrasé */
    allocation en swap et recopie des arguments d'exec et l'environnement dedans
    appel à getxfile()
    début
        si fichier déjà ouvert en écriture alors
            retourner ETXTBUSY
        finsi
        incrémenter le compteur de référence au fichier
        positionner le flag pagination_à_la_demande
        si taille du programme > taille disponible dans le swap alors
            retourner Erreur
        finsi
        appel à xalloc()
        début
            allocation sur le swap pour les piles et les données du nouveau programme
            création et remplacement des tables auxiliaires (tables des pages) du processus
            si le code exécutable est déjà utilisé par un autre processus alors
                ajouter le processus courant à la liste des processus partageant ce texte exécutable
            finsi
            mise-à-jour de la table auxiliaire pour le code exécutable à partir du swap ou de la mémoire des
            autres processus
            retourner
        fin
        retourner
    fin
    mise-à-jour du sommet de la pile user et recopie des paramètres d'exec
    sauvegarde de argv et argc dans la structure save-state
    appel à setregs()
    début
        mettre tous les signaux à default
        positionner le niveau de privilège à 3 (mode user)
        fermer les fichiers
    fin
    rendre les buffers
    relâcher l'i-node
    retourner /* pas au programme appelant mais au nouveau programme */
fin
```

Algorithme 4.4 : wait()

Résultat :
- Retourne le pid du processus qui vient de se terminer
- La paramètre permet de récupérer le status de terminaison du fils suite à l'appel de la fonction **exit()**

début
  `Boucle` :
  pour *chaque fils* **faire**
    si *état du fils* = ***SZOMB*** **alors**
      status = status du fils conservé dans la structure **proc** du fils zombie
      appel à la fonction **freeproc**
      début
        `/* libération de ce qu'il restait du fils */`
        enlever le processus fils de la liste des **UID**
        enlever le processus fils de la liste des **PID**
        enlever le processus fils de la liste des groupes
        mise-à-jour des statistiques sur les ressources du père
        RAZ de la table **proc** du fils
        ajouter l'entrée de la table **proc** dans la liste des entrées libres
      fin
      retourner ***PID*** *du fils*
    finsi
  finpour
  si *aucun fils* **alors**
    retourner ***ECHILD***
  finsi
  appel à la fonction **sleep()** avec une priorité sensible aux signaux `/* PWAIT = 158 */`
  Aller à Boucle
fin

```
#define SIGQUIT         3       /* Quit (POSIX).  */
#define SIGABRT         6       /* Abort (ANSI).  */
#define SIGKILL         9       /* Kill, unblockable (POSIX).  */
#define SIGUSR1         10      /* User-defined signal 1 (POSIX).  */
#define SIGSEGV         11      /* Segmentation violation (ANSI).  */
#define SIGUSR2         12      /* User-defined signal 2 (POSIX).  */
#define SIGPIPE         13      /* Broken pipe (POSIX).  */
#define SIGALRM         14      /* Alarm clock (POSIX).  */
#define SIGTERM         15      /* Termination (ANSI).  */
#define SIGCLD          SIGCHLD /* Same as SIGCHLD (System V).  */
#define SIGCHLD         17      /* Child status has changed (POSIX).  */
#define SIGCONT         18      /* Continue (POSIX).  */
#define SIGSTOP         19      /* Stop, unblockable (POSIX).  */
#define SIGPWR          30      /* Power failure restart (System V).  */
```

La gestion des signaux peut varier d'un système Unix à un autre par rapport à leur mémorisation. En effet, sous Unix System V, l'association d'une fonction à un signal ne fonctionne qu'une fois. Avant d'exécuter réellement la fonction de traitement associée (le *handler*), le système supprime l'association. Ainsi, il est recommandé de réassocier le signal et la fonction avec la fonction `signal` dès le début de celle-ci.

Sous Unix BSD, il n'y a pas de défaut car il a mémorisation des évènements dans un masque.

Le schéma suivant présente l'architecture et le fonctionnement d'une gestion des signaux :

4.2.2.1 Le signal SIGCHLD

La gestion par défaut du signal de terminaison d'un fils par le père est de l'ignorer, de ne rien faire. Le processus père est réactivé par le signal et si celui-ci ne fait rien avec, le processus retourne à son état précédent.

	Processus 1	Processus 2		
	...	`signal(Evt, Fct)`	← (1)	association de la fonction
	...	...		à l'événement
	...	...		
	...	...		
	...	`pause()`	← (2)	attente d'un événement
	...	...		
(3) réveil de *P2* →	`kill(P2, Evt)`			
	...	Fonction Fct()	← (4)	exécution de la fonction
	...	...		
	...	...		
	...	return	← (5)	retour à l'instruction
	...			suivant `pause`

Habituellement, pour terminer proprement un processus fils qui vient de se terminer, le père doit effectuer un appel à la fonction `wait()`. Une façon de procéder est de faire un appel à cette fonction dans la fonction de capture du signal `SIGCHLD`.

Cependant, depuis la norme POSIX.1-2001, le fait d'ignorer explicitement le signal de terminaison d'un fils a pour effet pour le noyau de finir la terminaison du processus sans que le processus père n'ai à faire `wait()` et donc sans que le processus fils ne devienne zombie.

4.2.2.2 Les signaux de terminaison

Il existe 5 signaux de terminaison de processus : `SIGTERM`, `SIGINT`, `SIGQUIT`, `SIGKILL` et `SIGHUP`. Le rôle final de chacun de ces signaux est de terminer le processus qui le reçoit mais la façon de gérer cette réception diffère légèrement :

- `SIGTERM` : permet de demander poliment et gentiment au processus de se terminer. Ce signal peut être bloqué ou capturé pour effectuer une action que le programmeur a prévu pour ce cas. Habituellement, la fonction de capture sert à effacer des fichiers temporaires ou autres objets créés par le processus qu'il n'est pas utile de laisser exister après la fin du processus.
- `SIGINT` : Ce signal de terminaison est très similaire au signal `SIGTERM`. Celui-ci peut être généré par la combinaison de touches *Ctrl+C*. Ce signal peut être capturé ou ignoré mais la différence avec `SIGTERM` est que même dans ces cas le processus se terminera.
- `SIGQUIT` : Avec ce signal, le processus va se terminer et générer un fichier *core dump* qui correspond à une sauvegarde de l'espace mémoire et du contexte du processus au moment de l'arrivée du signal. L'objectif est orienté vers la débuggage. Ce signal peut aussi être émis par l'utilisateur via les touches *Ctrl+*.
- `SIGKILL` : Le processus va se terminer immédiatement. Ce signal ne peut être capturé et ne peut être ignoré. Ainsi, il n'est pas possible d'effectuer de fonctions de nettoyage ni de core dump. Ce signal est la dernière extrémité pour terminer un processus.
- `SIGHUP` : Ce signal est émis quand la connexion est perdu avec le terminal. Ceci n'arrive plus souvent mais apparaissait quelques années en arrière avec les connections lentes. L'action par défaut à la réception de ce signal est la terminaison du processus. Actuellement, ce signal est plus souvent capturé pour indiquer au processus de devenir un démon (se détache du terminal) ou bien de relire la configuration du processus démon.

4.3 Communication inter-processus

Dans le système Unix standard, les processus sont isolés les uns des autres. Ils ne peuvent donc communiquer que par des primitives du noyau et les noms des objets partagés sont transmis par filiation en héritage. Deux processus issus de filiation différente ne peuvent communiquer qu'à travers le système de fichiers.

Nous allons aborder quelques outils à notre disposition sous Unix pour permettre la communication et/ou la synchronisation entre processus, de même filiation ou non. Certains outils passent par le système de fichiers, d'autre passe par une zone de la mémoire gérée par le noyau du système.

4.3.1 Les tubes (pipe)

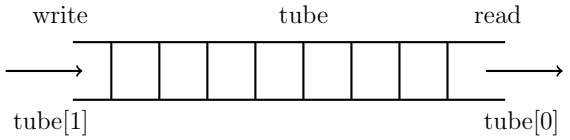
4.3.1.1 Description

Un tube est zone mémoire gérée selon le modèle du producteur-consommateur avec les propriétés suivantes :

— L'accès se fait en exclusion mutuelle,
— Il faut **toujours au moins** 1 lecteur et 1 écrivain,
— Les opérations de lecture, respectivement d'écriture, sont bloquantes lorsque le tube est vide, respectivement plein.

La fonction Unix `pipe(int[2])` permet de créer un tel tube. Ce tube est perçu par le système comme un fichier conventionnel mais la différence avec un fichier conventionnel. L'ouverture d'un fichier en lecture et/ou écriture avec la commande `open()` retourne un seul descripteur qui sera utilisé pour les entrées/sorties. Dans le cas d'un tube, nous avons deux descripteurs distincts : un pour l'écriture et un pour la lecture. De plus, il n'est pas possible d'utiliser la commande `open()`, il faut obligatoirement utiliser la fonction `pipe()`.

Cette commande prend en argument un tableau de taille 2 pour contenir deux descripteurs de fichier (un descripteur de fichier est en fait un `int`). Par exemple, suite à la commande `pipe(tube)`, `tube[0]` contient le descripteur pour la lecture et `tube[1]` contient le descripteur pour l'écriture dans le tube.



4.3.1.2 Fonctionnement

Un tube s'utilise également comme un fichier ordinaire à l'aide des fonctions `read()` et `write()` : `read(tube[0],...)` et `write(tube[1],...)`. L'écriture et la lecture avec ces fonctions se réalisent en exclusion mutuelle.

La fermeture d'un des descripteurs du tube se fait à l'aide de la fonction `close(desc)`. Il faut noter qu'il n'est pas possible de fermer les deux descripteurs à la fois, il faut fermer successivement les deux descripteurs avec `close()`.

La fonction de contrôle `fnctl()` est également définie pour les tubes afin de modifier ou de consulter les propriétés de ceux-ci (taille, occupation, état bloquant etc).

Illustrons l'utilisation classique d'un tube pour deux processus issus d'une même filiation :

```
int t[2];
pipe(t)    //crée un tube, t[0] et t[1] contiennent les deux descripteurs
if (fork()) {
    // P1 processus père, écrivain
    close(t[0]);
    ...
} else {
    // P2 processus fils, lecteur
    close(t[1])
```

```
    ...
}
```

Une des règles d'utilisation des tubes est de fermer les descripteurs inutiles. Ainsi, si P1 est le processus écrivain, on ferme son descripteur de lecture : `close(t[1])`. On fait de même pour P2 mais sur l'autre descripteurs, celui d'écriture : `close(t[0])`.

Or, les contraintes de manipulation et d'utilisation d'un tube imposent qu'il n'est possible de faire `open()` sur un tube et il faut toujours au moins un lecteur et un écrivain pour qu'un tube existe.

Qu'en est-il dans cette situation ?

Dans cette situation, la communication père-fils à travers le tube fonctionne. En effet, par l'exécution de `fork()`, le tube devient à la fois ouvert par le processus père et par le processus fils. Ainsi, chacun des deux descripteurs du tube sont ouverts deux fois (une fois par le père, une fois par le fils). Si après la création du processus fils nous comptons le nombre de lecteurs et d'écrivains, nous somme passé de $(1, 1)$ avant `fork()` à $(2, 2)$ après `fork()`.

Avec les opérations `close()` respectives du père et du fils juste après l'exécution de `fork()`, le compteur du nombre de lecteurs et d'écrivains revient à $(1, 1)$. Ainsi le tube continue de satisfaire les conditions nécessaires pour qu'il reste actif et ne soit pas cassé/fermé.

4.3.1.3 Cas du tube cassé

Le tube peut être cassé dans deux situations :
— Il n'y a plus d'écrivain,
— Il n'y a plus de lecteur.
La gestion de ces deux situations par le système est différente.

Plus d'écrivain : Dans la situation de l'exemple précédent, admettons que le processus fils soit bloqué en lecture (lecture avec `read()` sur un tube vide). Que se passe-t-il si le père (le producteur) meurt ? En plus du fait que le processus fils va être adopté par le processus `init 1`, le compteur du nombre de lecteurs et d'écrivains passe à $(1, 0)$. Il n'y a donc plus de processus écrivain et le tube est cassé. Normalement, le processus fils devrait rester bloqué indéfiniment car le tube sera indéfiniment vide. Mais le comportement de la fonction `read()` est différente pour un tube. Les valeurs de retour de `read()` sont :
— Le nombre d'octets lus (> 0),
— La valeur 0 pour indiquer une fin de fichier atteinte,
— -1 si il y a eu une erreur.
Le fait que l'opération `read()` soit faite sur un tube cassé ne vas pas être considérée par `read` comme une erreur mais comme une fin de fichier. Donc `read()` se débloque et retourne 0. Le processus est ainsi débloqué. S'il y avait des données dans le tube, elles ne sont pas perdues. C'est pour cela aussi qu'il est important de tester la valeur de retour pour savoir pourquoi l'opération `read()` s'est débloquée (données arrivées ou tube cassé).

Plus de lecteur : Dans la situation inverse, le processus fils meurt et le processus veut écrire dans le tube. Le compteur de descripteurs ouvert passe de $(1, 1)$ à $(0, 1)$. N'ayant plus de lecteur, le tube se casse. Le comportement de la fonction `write()` sur un tube cassé est un peu différent de celui de `read()`. Pour rappel, les valeurs de retour de `write()` sont :
— Le nombre d'octets écrits (≥ 0),
— -1 en cas d'erreur.
Dans le cas de l'écriture sur un tube cassé est considéré comme un cas d'erreur et est donc non bloquant (retour de la valeur -1). Par ailleurs, `write()` envoie en plus le signal `SIGPIPE` au processus qui a voulu écrire dans un tube cassé (à lui même). Le traitement par défaut associé à la réception du signal `SIGPIPE` est la terminaison du processus, généralement accompagné du message *broken pipe*.

Exemple un peu plus complet de l'utilisation d'un tube :

```
int tube[2];
pipe(tube);
if (fork()==0){
```

```
    /*fils lecteur*/
    close(tube[1]);
    if (read(tube[0], ...)){
        ...
    }
    close(tube[0]);
    exit(0);
}
else{
    /* pere ecrivain */
    close(tube[0]);
    if(write(tube[1], ...)){
        ...
    }
    close(tube[1]);
    wait(NULL);
    exit();
}
```

4.3.1.4 Tube et processus de filiation différente

Si deux processus veulent communiquer à travers un tube mais qu'ils ne sont pas liés par une même filiation que le créateur du tube, ils ne peuvent plus simplement utiliser une variable contenant les deux descripteurs. En effet, n'étant pas issus d'une même filiation il n'y a pas de copie des variables entre les processus.

Unix met à notre disposition une solution qui utilise le système de gestion de fichier. La fonction `mkfifo()` va créer sur le disque un vrai fichier portant un nom et ayant un type particulier : tube (pipe). Le fichier ainsi créé est un vrai fichier présent sur le disque et donc accessible via son nom par n'importe quel processus ayant les droits pour cela.

L'affichage des droits par la commande `ls` montre le type particulier de ce fichier : `prw-rw-r--`.

Un processus voulant utiliser un tel tube doit donc connaître le nom absolu de ce fichier et doit posséder les droits pour l'ouvrir soit en lecture, soit en écriture avec la fonction `open()`, comme pour un fichier classique. L'utilisation se fait toujours avec les fonctions `read()` et `write()`.

4.3.2 La fonction `dup2`

Les deux méthodes de communication présentées juste avant sont des méthodes de communication explicites : le programmeur qui va utiliser ce moyen écrit son programme en conséquence (implantation de `pipe()` ou connaissance du nom d'un fichier `FIFO`).

Cependant, il arrive qu'un utilisateur veuille faire communiquer deux processus entre eux sans que le programmeur n'est explicitement prévu cette communication. C'est le cas par exemple lorsqu'on veut rediriger la sortie d'un programme vers l'entrée d'un autre programme : les 2 processus sont indépendants et nous n'avons pas accès aux sources ou pas intérêt à les modifier pour intégrer un tube ou donner le nom d'un fichier FIFO pour une utilisation peut être unique.

Heureusement, pour réussir à faire ces redirections il existe comme solution la commande `dup2()`.

Prenons l'exemple des commandes shell `ls` et `wc`. Celles-ci utilisent la sortie standard pour afficher leurs résultats et l'entrée standard pour lire les données à traiter dans le cas de `wc`. Ainsi, dans leur code source, ils effectuent des opérations (`write()`/`read()`) sur les descripteurs des sortie et entrée standard.

L'idée est de duper les programmes en les laissant écrire sur leurs descripteurs usuels tout en changeant silencieusement le vrai descripteur qui sera utilisé.

Pour rappel, les descripteurs associés aux entrées/sorties sont fixes et sont les suivants :

— Descripteur d'entrée standard : 0 (`STDIN`)
— Descripteur de sortie standard : 1 (`STDOUT`)
— Descripteur d'erreur standard : 2 (`STDERR`)

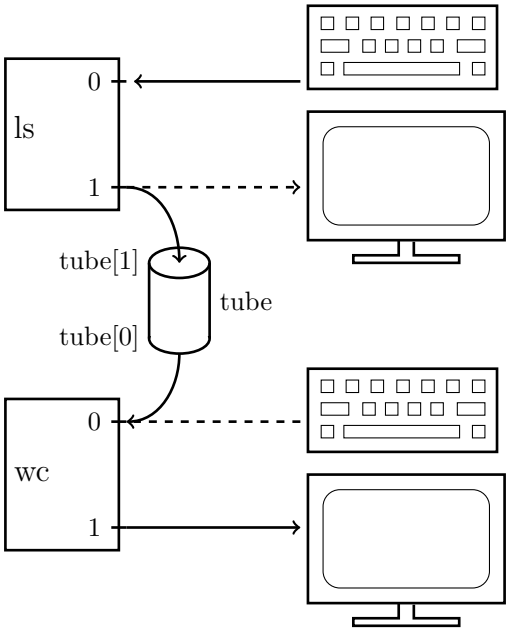
La commande `dup2(int descTo, int descFrom)` permet de créer une sorte d'alias entre un descripteur existant et un autre et au final les deux descripteurs seront équivalents. Cette fonction prend deux paramètres :

— descFrom : le descripteur qui sera dupé,

— descTo : le descripteur vers lequel sera redirigé le descripteur descFrom.

Ainsi, par exemple après l'exécution de `dup2(tube[0], 0)`, le descripteur de l'entrée standard (le clavier) sera redirigé vers la sortie d'un tube. Dès que des caractères (marqués par une fin de ligne) seront dans le tube, le processus attendant sur l'entrée standard va les considérer comme ayant été tapés au clavier.

De même pour le descripteur de sortie, après `dup2(tube[1], 1)`, tout ce que le programme d'origine va afficher à l'écran sera redirigé vers l'entrée du tube.



Ce principe fonctionne car après exécution d'une commande `exec()`, les descripteurs d'entrée/sortie 0, 1 et 2 sont conservés. Ils ne sont ni réinitialisés ni créés. Ils sont directement repris du processus *hôte* qui fait `exec()`, et ceci même si elles ont été redirigées.

4.3.3 Les mécanismes IPC : Inter Process Communication

Les IPC sont des mécanismes du noyau Unix pour étendre les possibilités de communication et de synchronisation des processus. Trois mécanismes ont été proposés :

— Les sémaphores,

— Les zones de mémoires partagées,

— Les files de messages.

Les IPC sont des ressources indépendantes des processus et possèdent des attributs tels que :

— Un nom,

— Des droits d'accès.

Par ailleurs, ces objets étant indépendants des processus, ils sont rémanents, c'est-à-dire qu'ils peuvent survivre à la terminaison du processus. En effet, si le processus ne détruit pas explicitement les objets IPC qu'il a créés, ceux-ci continuent d'exister dans le système.

Pour manipuler les objets IPC, il est nécessaire d'utiliser des primitives noyau prédéfinies et qu'on peut regrouper par famille :

— Accès ou création : `*get()` – `semget(), shmget(), msgget()` ;

— Contrôle (dont destruction) : `*ctl()` – `semctl(), shmctl(), msgctl()` ;

— Utilisation : `semop(), shmat(), shmdt(), msgsnd(), msgrcv()` ;

4.3.3.1 Identification / Unicité

Les objets IPC étant définis dans un contexte inter-processus global, ces objets peuvent être manipulés aussi bien par des processus d'une même filiation que par des processus indépendants. Cet avantage des IPC amène aussi plusieurs contraintes : comment garantir que les processus manipule bien le bon objet IPC ? Comment garantir que l'objet n'est bien créé qu'une seule fois ?

— Pour résoudre la désignation unique des objets IPC, un processus va créer un tel objet en le désignant par une clé. L'ensemble des processus qui veulent partager cet objet doit en connaître la clé et l'utiliser. Cette clé peut être soit implicite, soit définie par convention, soit se trouver dans un fichier.

— Pour résoudre le problème de l'existence unique de l'objet il y a plusieurs possibilités.
  — Soit on force le processus créateur (il est choisi par le programmeur),
  — Soit tous les processus essaient de le créer et seul le premier réussira. Le retour de la fonction de création permettra de distinguer le créateur effectif des autres.

Une fois l'objet créé, le système fournit un identifiant unique qui servira à sa manipulation. De plus, comme ces objets sont rémanents, il faut explicitement s'assurer de la destruction des objets qui ne sont plus utilisés. Il existe dans le système la commande shell `icps` qui permet de lister l'ensemble des objets IPC existant actuellement dans le système. Leur destruction pourra se faire *via* la commande `ipcrm`.

4.3.3.2 Création

Le mode de création peut être différent selon que la communication et/ou la synchronisation a lieu entre des processus d'une même filiation ou non.

Dans le cas de processus d'un même filiation, le système d'exploitation permet de le laisser seul gérer la désignation de l'objet IPC et de ne pas avoir à l'indiquer nous-mêmes. Par le biais des recopies des données par la commande `fork()`, l'identifiant unique de l'objet sera partagé entre les différents processus. Le mot clé particulier à utiliser est `IPC_PRIVATE` comme dans l'exemple suivant :

```
id=*get(IPC_PRIVATE, arguments, droits)
```

Le processus père va créer un objet IPC qui est indiqué comme étant privé au processus. Ainsi le système va fournir un identifiant unique (`id`) pour objet IPC créé en retour de la fonction `*get()`.

Dans le cas de processus de filiation différente, on ne peut pas utiliser le mot clé `IPC_PRIVATE`. Il faut alors explicitement donner la clé de l'objet, la même pour tous les processus. La création se fera comme dans les exemples suivants :

```
id=*get(cle, argument, IPC_CREAT|droits)
id=*get(cle, argument, IPC_CREAT|IPC_EXCL|droits)
```

Le mot clé `IPC_CREAT` indique la demande de création de l'objet s'il n'existe pas et sinon récupère juste l'identifiant d'un objet déjà existant. Si on ne met pas ce mot clé et que l'objet n'existe pas, il ne le créera pas.

Le mot clé `IPC_EXCL` (généralement en conjonction de `IPC_CREAT`) permet de s'assurer de la création effective, c'est-à-dire que si l'objet existe déjà, non seulement il ne sera pas recréé mais la fonction retournera une erreur car elle n'aura pas pu créer l'objet. Ceci permet d'assurer qu'il n'y a pas dans le système un même objet rémanent qui n'a pas été effacé pour une raison quelconque.

4.3.3.3 Génération de la clé

Habituellement, la clé n'est pas explicitement donnée aux fonctions `*get()`, on utilise une fonction intermédiaire qui va générer une clé : la fonction `ftok()`.

Cette fonction permet de créer une clé à partir de deux éléments :

1. Un nom de chemin dans le système de fichier, par exemple le nom du fichier source ;
2. Un nom de ressource (par exemple un sémaphore), identifiée par un caractère fixé par l'utilisateur.

Le point (1) permet de s'assurer que la clé n'est pas utilisée par d'autres programmes, le point (2) permet de distinguer différentes ressources entre elles pour ce processus. Exemple :

```
semid = semget(ftok("/home/exo.c", 'A'), ...);
```

Toutefois, il peut arriver des collisions dans la clé générée.

4.3.3.4 Destruction

La destruction explicite d'un objet IPC par un processus se fait à travers la commande contrôle `*ctl()` qui dans l'exemple suivant :

```
*ctl(id, IPC_RMID, arguments)
```

Il faut indiquer l'identifiant unique de l'objet (obtenu par `*get()`), et utiliser le mot clé particulier `IPC_RMID`.

Dans le cas d'un filiation, le père doit attendre que les fils soient terminés avant d'effectuer la demande de destruction de l'objet.

Dans le cas général, on ne peut connaître l'ordre de terminaison des processus. Mais le premier processus qui effectuera cette commande détruira l'objet IPC, même si les autres processus n'avaient pas fini avec lui (ils obtiennent un message d'erreur). La seule exception à ce comportement est pour les zones de mémoire partagées.

4.3.3.5 Les Sémaphores

Les objets IPC sémaphores correspondent sémaphores généralisés : des tableaux de sémaphores à k jetons. Ils possèdent les propriétés suivantes :

- Les opérations P et V sur un ensemble de sémaphores sont indivisibles,
- Décrémentation/Incrémentation de n jetons en une seule primitive P ou V,
- La primitive P peut être rendue non bloquante,
- Un mécanisme d'*undo* permet de mémoriser les actions faites et de défaire celles-ci en cas de problème (destruction accidentelle du processus). Ceci permet de conserver une valeur cohérente du sémaphore.

Nous allons décrire plus en détails l'utilisation des primitives définies pour les sémaphores.

La création La création d'un sémaphore passe par l'utilisation de la fonction `semget()` avec comme paramètre la clé d'identification (ou `IPC_PRIVATE`), le nombre de sémaphores du tableau, les droits et drapeaux particuliers. Exemple de création d'un tableau de 1 sémaphore :

```
semid = semget(IPC_PRIVATE, 1, IPC_CREAT|0644)
```

Les droits peuvent aussi être spécifiés par les constantes `O_RDONLY`, `O_WRONLY` et `O_RDWR`. Ne pas oublier le drapeau particulier `IPC_EXCL`.

Les opérations La manipulation des sémaphores passe principalement par la fonction `semop()`. Elle permet de réaliser les primitives P et V, et prend trois paramètres :

- L'identifiant du sémaphore (obtenu par `semget()`),
- Un pointeur sur un tableau de structures définissant les opérations à réaliser,
- Le nombre de structures dans le tableau précédent.

Les structures pour les opérations sont les suivantes :

```
struct sembuf {
       unsigned short sem_num;  /* Numéro du sémaphore        */
       short          sem_op;   /* Opération sur le sémaphore */
       short          sem_flg;  /* Options pour l'opération   */
}
```

Les options possibles pour `sem_flg` sont `IPC_NOWAIT` afin de rendre non bloquante l'opération, et `SEM_UNDO` pour indiquer l'opération sera annulée lorsque le processus se terminera.

Exemple de la primitive P() :

```
P(int id) {
    struct sembuf sb;
    sb.sem_num = 0;
    sb.sem_op = -1;
    sb.sem_flg = 0;

    semop(id, &sb, 1);
}
```

Les contrôles Les opérations de contrôle sur les sémaphores sont assez variées et sont importantes. Elles servent à l'initialisation, la consultation et la destruction des sémaphores. La fonction `semctl()` prend quatre paramètres :

— L'identifiant unique du sémaphore,
— L'indice du premier sémaphore concerné,
— La commande de contrôle à effectuer,
— Les paramètres pour la commande de contrôle.

Les commandes principales sont :

— `SETVAL` : initialise la valeur d'un sémaphore,
— `SETALL` : initialise tous les sémaphores par un tableau de valeurs,
— `GETVAL` : récupère la valeur d'un sémaphore,
— `GETALL` : récupère la valeur de tous les sémaphores dans un tableau,
— `GETPID` : récupère le PID du dernier processus à avoir effectuer `semop()`,
— `IPC_STAT` : informations diverses sur le tableau de sémaphores,
— `IPC_RMID` : destruction immédiate.

Quand le paramètre de commande nécessite un argument, le type de celui-ci (le quatrième de `semctl()`) est une union `semun` :

```
union semun {
    int              val;    /* Valeur pour SETVAL */
    struct semid_ds *buf;    /* Tampon pour IPC_STAT, IPC_SET */
    unsigned short  *array;  /* Tableau pour GETALL, SETALL */
    struct seminfo  *__buf;  /* Tampon pour IPC_INFO (spécifique à Linux) */
    };
```

Par exemple, pour initialiser tous les sémaphores d'un tableau de 3 sémaphores, il faut faire :

```
sem_array[0] = 0;
sem_array[1] = 4;
sem_array[2] = 1;
semctl(id, 0, SETALL, sem_array);
```

Pour initialiser juste le 4eme élément du tableau à 10 il faut faire :

```
semctl(id, 3, SETVAL, 10);
```

4.3.3.6 Les Zones de Mémoire partagée (Shared Memory)

Les zones de mémoire partagée sont un moyen efficace de partager des informations entre un ensemble de processus. Une vision simpliste est que ces zones sont une variable partagée entre les processus : chacun peut la modifier et la lire, et dès qu'une modification est effectuée, tous les processus lisant ensuite la valeur auront la nouvelle valeur. Par contre, il est du ressort du programmeur de mettre en place les modèles et protocoles de communication/synchronisation pour manipuler cette zone partagée.

La création La création d'une zone de mémoire partagée se fait par la commande `shmget()`, et prend comme paramètre :

— La clé d'identification,
— La taille en octets de la zone,
— Les drapeaux et les droits.

Les opérations Deux opérations sont possibles sur les zones de mémoire partagée : l'attachement et le détachement à l'espace mémoire virtuel du processus.

La fonction `shmat()` permet d'attacher la zone. Elle prend trois paramètres et retourne l'adresse où est attachée cette zone. Les trois paramètres sont les suivants :

- L'identifiant unique de la zone manipulée,
- Une adresse mémoire :
  - Si la valeur est à 0 (cas habituel et le plus sûr), le système détermine lui-même l'adresse appropriée.
  - Si la valeur est non nulle, le système essaie d'attacher la zone au plus près de l'adresse indiquée.
- Des flags restreignant les droits d'accès.

La fonction `shmdt()` permet de détacher la zone de l'espace du processus, sans la détruire. L'unique paramètre de la fonction est l'adresse à détacher (la variable préalablement attachée).

Le Contrôle Contrairement aux sémaphores, beaucoup moins d'opérations de contrôle sont disponibles pour les zones de mémoire partagée. La fonction `shmctl()` prend au plus trois paramètres :

- L'identifiant de la zone de mémoire partagée,
- La commande de contrôle à effectuer (`IPC_RMID`, `IPC_STAT`, ...)
- Un pointeur sur les arguments de la commande, si besoin.

Il faut noter toutefois que la destruction de l'objet zone de mémoire partagée ne sera effective qu'une fois que tous les processus manipulant cet objet l'auront détaché de leur espace mémoire.

4.3.3.7 Les files de messages

Le dernier type d'objet IPC défini est une sorte de généralisation des tubes. En effet, les files de messages permettent à des processus de s'échanger des informations typées selon le modèle du producteur-consommateur. Les messages sont gérés en mode FIFO, comme les tubes.

Les messages eux-mêmes sont définis comme un couple type-données. La gestion et la signification du type est uniquement dépendante de l'utilisateur. Les données sont le contenu du message.

La création La création d'une file de message ne prend que deux paramètres : la clé d'identification et les droits et drapeaux. La fonction appelée est `msgget()`.

Les opérations Comme pour le cas des zones de mémoire partagée, deux opérations sont définies pour les files de messages : `msgsnd()` pour l'émission d'un message et `msgrcv()` pour la réception/le prélèvement d'un message.

La fonction `msgsnd()` prend 4 paramètres :

- L'identifiant de la file de message,
- Un pointeur vers une structure contenant le type et le contenu du message,
- la longueur effective du message,
- Des drapeaux pour la gestion des files.

La définition de la structure d'un message est laissée à l'utilisateur mais elle doit suivre le schéma général suivant :

```
 msgbuf {
        long mtype;       /* message type, must be > 0 */
        char mtext[1];    /* message data */
};
```

Le seul élément obligatoire dans la structure d'un message pour qu'il soit compatible est qu'il commence par un type `long` correspondant au type (qui doit être > 0). Le reste de la structure peut être quasiment quelconque : entier, chaîne de caractères, réels, structures utilisateur, ou toute combinaison de ces types. La limitation est que la définition de la structure d'un message inclue directement les données qui sont manipulées : pas de pointeurs mais directement les valeurs.

En effet, cela n'a pas de sens d'envoyer des pointeurs dans le message car ceux-ci sont des adresses de l'espace virtuel du processus émetteur et ne correspondent pour aucune raison à l'espace virtuel du

récepteur. Ceci s'illustre bien avec les chaînes de caractères, on n'envoie le pointeur de la chaîne mais bien l'ensemble des caractères qui la compose :

```
/* on défini la structure du message */
    struc msgtext {
        long m_type;
        char m_text[N];
    } message;
/* on prépare le message */
    message.m_type = 3;
    strcpy(message.m_text, "Le message");
/* on envoie le message */
   msgsnd(msgid, &message, strlen(message.m_text), 0);
```

La fonction `msgrcv()` permet de prélever des messages de la file. Cette fonction admet 5 paramètres :
- L'identifiant de la file de message,
- Un pointeur vers une variable pour stocker le message reçu,
- Le nombre maximum de caractères pour le contenu du message à recevoir,
- Le type du message à prélever,
- Des drapeaux caractérisant la gestion de la file comme `IPC_NOWAIT` pour rendre la réception non bloquante.

La façon dont les messages sont prélevés dépend de la valeur du type passé en paramètre de `msgrcv()` (la gestion est FIFO) :
- Si le type est 0 : le premier message dans la file est prélevé, quelque soit sont type réel.
- Si le type est > 0 : le premier message dans la file de **même** type est prélevé.
- Si le type est < 0 : le premier message de plus petit type $\leq |type|$ est prélevé.

Le contrôle Comme pour le cas des zones de mémoire partagée, les opérations de contrôle principales se résument à la destruction et la consultation de l'état de l'objet avec les commandes `IPC_RMID` et `IPC_STAT` :

```
msgctl(msgid, cmd, argument);
```