I42 — Réseaux et Système
Partie « système d'exploitation »

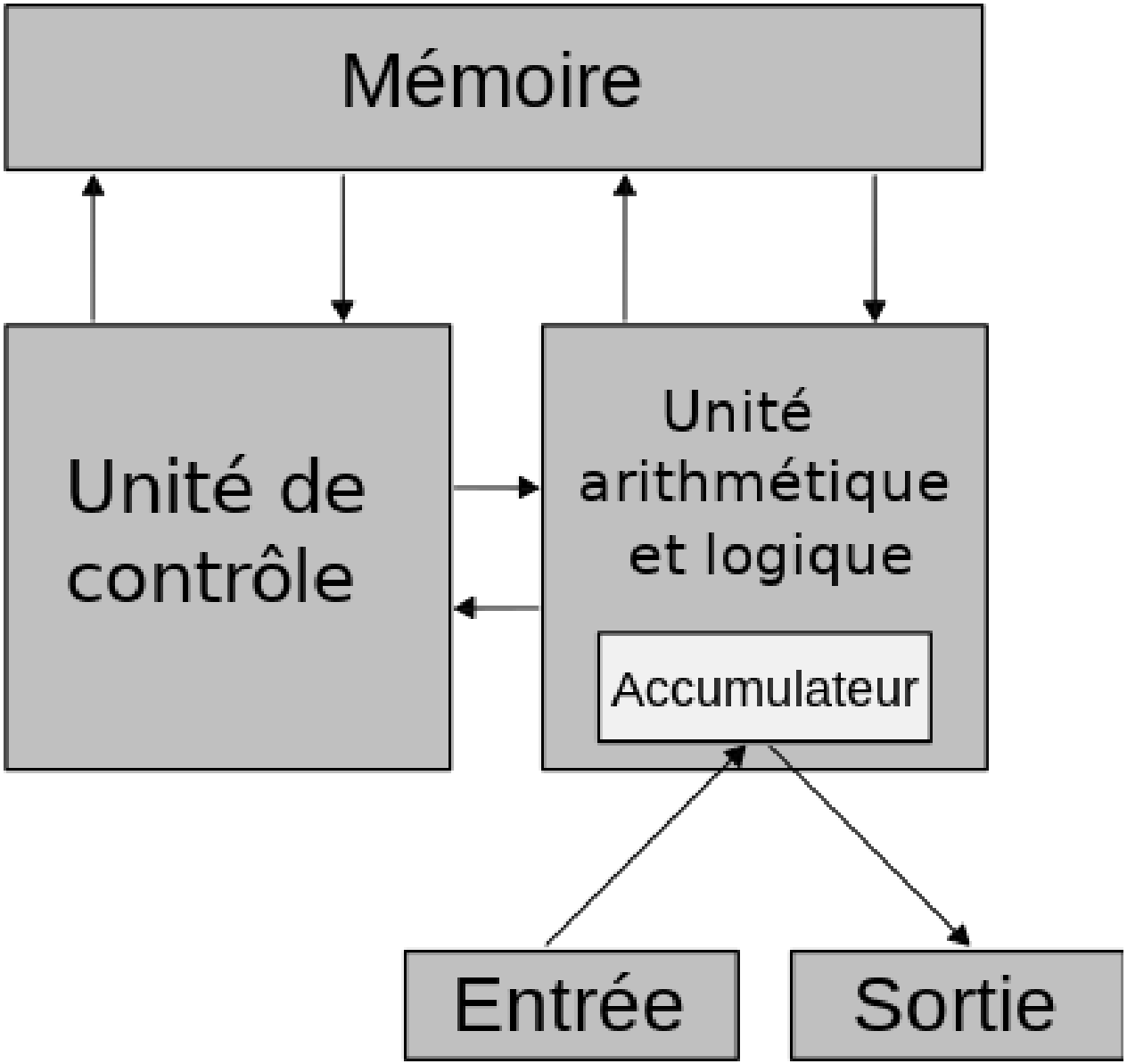
J. Razik

Dpt. Informatique — UTLN

2020–2021

Plan

Synchronisation


Mémoire
Unité de contrôle
Unité arithmétique et logique
Accumulateur
Entrée
Sortie

Exclusion Mutuelle

Typologie des ressources

Différents types de ressources :

- Ressource privée,

Typologie des ressources

Différents types de ressources :

- Ressource privée,
- Ressource commune,

Typologie des ressources

Différents types de ressources :

- Ressource privée,
- Ressource commune,
- Ressource partageable,

Typologie des ressources

Différents types de ressources :

- Ressource privée,
- Ressource commune,
- Ressource partageable,
- Ressource critique.

Typologie des processus

Différents types de processus :

- Processus indépendants,

Typologie des processus

Différents types de processus :

- Processus indépendants,
- Processus concurrents ou parallèles.

L'Exclusion Mutuelle

- Prologue,

L'Exclusion Mutuelle

- Prologue,
- Section critique,

L'Exclusion Mutuelle

- Prologue,
- Section critique,
- Épilogue.

L'Exclusion Mutuelle

- Prologue,
- Section critique,
- Épilogue.

L'Exclusion Mutuelle

- Prologue,
- Section critique,
- Épilogue.

Objectif :

- Proposer des algorithmes pour les prologue et épilogue réalisant les contraintes suivantes :

L'Exclusion Mutuelle

- Prologue,
- Section critique,
- Épilogue.

Objectif :

- Proposer des algorithmes pour les prologue et épilogue réalisant les contraintes suivantes :
  - Un seul processus est à la fois en section critique ;

L'Exclusion Mutuelle

- Prologue,
- Section critique,
- Épilogue.

Objectif :

- Proposer des algorithmes pour les prologue et épilogue réalisant les contraintes suivantes :
  - Un seul processus est à la fois en section critique ;
  - Si des processus sont en attente devant la section critique et qu'il n'y a aucun processus en section critique, alors il faut garantir un temps d'attente fini ;

L'Exclusion Mutuelle

- Prologue,
- Section critique,
- Épilogue.

Objectif :

- Proposer des algorithmes pour les prologue et épilogue réalisant les contraintes suivantes :
  - Un seul processus est à la fois en section critique ;
  - Si des processus sont en attente devant la section critique et qu'il n'y a aucun processus en section critique, alors il faut garantir un temps d'attente fini ;
  - Le blocage d'un processus hors section critique ne doit pas empêcher un autre processus d'entrer en section critique ;

L'Exclusion Mutuelle

- Prologue,
- Section critique,
- Épilogue.

Objectif :

- Proposer des algorithmes pour les prologue et épilogue réalisant les contraintes suivantes :
  - Un seul processus est à la fois en section critique ;
  - Si des processus sont en attente devant la section critique et qu'il n'y a aucun processus en section critique, alors il faut garantir un temps d'attente fini ;
  - Le blocage d'un processus hors section critique ne doit pas empêcher un autre processus d'entrer en section critique ;
  - Il n'y a pas de processus privilégié.

Modèles pour la réalisation de l'Exclusion Mutuelle

- L'attente active,

Modèles pour la réalisation de l'Exclusion Mutuelle

- L'attente active,
- Les verrous,

Modèles pour la réalisation de l'Exclusion Mutuelle

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

L'attente active

Principe :

- Tester une variable (*flag*) commune entre les processus,

L'attente active

Principe :

- Tester une variable (*flag*) commune entre les processus,
- Mémorise le droit de passage,

L'attente active

Principe :

- Tester une variable (*flag*) commune entre les processus,
- Mémorise le droit de passage,
- Le processus teste la valeur de variable en boucle jusqu'à obtention du droit de passage.

L'attente active

Principe :

- Tester une variable (*flag*) commune entre les processus,
- Mémorise le droit de passage,
- Le processus teste la valeur de variable en boucle jusqu'à obtention du droit de passage.

L'attente active

Principe :

- Tester une variable (*flag*) commune entre les processus,
- Mémorise le droit de passage,
- Le processus teste la valeur de variable en boucle jusqu'à obtention du droit de passage.

Algorithme 9 : 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
```

L'attente active

Principe :

- Tester une variable (*flag*) commune entre les processus,
- Mémorise le droit de passage,
- Le processus teste la valeur de variable en boucle jusqu'à obtention du droit de passage.

Algorithme 11 : 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 12 : Épilogue de sortie de section critique de l'*attente active*

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

L'attente active

Problème : assurer la cohérence de la valeur de la variable

L'attente active

Inconvénients de cette méthode

- Consommation inutile de temps CPU juste pour tester une variable,

L'attente active

Inconvénients de cette méthode

- Consommation inutile de temps CPU juste pour tester une variable,
- Réalisation de l'indivisibilité dans le temps délicate,

L'attente active

Inconvénients de cette méthode

- Consommation inutile de temps CPU juste pour tester une variable,
- Réalisation de l'indivisibilité dans le temps délicate,
  - Sur une machine monoprocesseur : instruction spéciale *Test-And-Set*,

L'attente active

Inconvénients de cette méthode

- Consommation inutile de temps CPU juste pour tester une variable,
- Réalisation de l'indivisibilité dans le temps délicate,
  - Sur une machine monoprocesseur : instruction spéciale *Test-And-Set*,
  - Sur une machine multiprocesseur : blocage de la mémoire centrale.

L'algorithme le Peterson (1981)

- Permet de réaliser l'exclusion mutuelle d'une ressource en permettant une alternance entre les acquéreurs (chacun son tour),
- Synchronise 2 processus avec une seule variable P.

L'algorithme le Peterson (1981)

- Permet de réaliser l'exclusion mutuelle d'une ressource en permettant une alternance entre les acquéreurs (chacun son tour),
- Synchronise 2 processus avec une seule variable P.

Algorithme 15 : 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 16 : Épilogue pour l'algorithme de Peterson

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

Les verrous

- Structure composée de
  - Une variable booléenne commune entre les processus,
  - Une file d'attente de processus.
- Valeur de la variable à 0 = ressource critique libre,
- Un processus n'obtenant pas le jeton $\rightarrow$ file d'attente endormi,
- Un processus en attente sera réveillé quand il pourra passer.

Les verrous

- Structure composée de
  - Une variable booléenne commune entre les processus,
  - Une file d'attente de processus.
- Valeur de la variable à 0 = ressource critique libre,
- Un processus n'obtenant pas le jeton $\rightarrow$ file d'attente endormi,
- Un processus en attente sera réveillé quand il pourra passer.

Algorithme 19 : 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
```

Les verrous

- Structure composée de
  - Une variable booléenne commune entre les processus,
  - Une file d'attente de processus.
- Valeur de la variable à 0 = ressource critique libre,
- Un processus n'obtenant pas le jeton $\rightarrow$ file d'attente endormi,
- Un processus en attente sera réveillé quand il pourra passer.

Algorithme 21 : 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 22 : Épilogue 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
```

Les sémaphores

- Structure
  - Une variable entière,
  - Une file d'attente de processus.
- Des fonctions de manipulation,
  - Création avec valeur initiale,
  - Demande de jeton P,
  - Restitution de jeton V,
  - Destruction.

Les sémaphores

Algorithme 23 : Création d'un sémaphore : `creation(s,val)`

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

Algorithme 24 : Destruction d'un sémaphore

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

Les sémaphores

Algorithme 25 : Prologue pour les sémaphores, primitive P

$E(s) \leftarrow E(s) - 1$; `/* 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

Algorithme 26 : Épilogue 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

Les sémaphores

Propriétés

- On ne peut initialiser un sémaphore avec une valeur négative,

Les sémaphores

Propriétés

- On ne peut initialiser un sémaphore avec une valeur négative,
  - Mais la valeur courante peut devenir négative.

Les sémaphores

Propriétés

- On ne peut initialiser un sémaphore avec une valeur négative,
  - Mais la valeur courante peut devenir négative.
- $E(s) = E_0(s)$ - 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 ;

Les sémaphores

Propriétés

- On ne peut initialiser un sémaphore avec une valeur négative,
  - Mais la valeur courante peut devenir négative.
- $E(s) = E_0(s)$ - 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 ;

Les sémaphores

Propriétés

- On ne peut initialiser un sémaphore avec une valeur négative,
  - Mais la valeur courante peut devenir négative.
- $E(s) = E_0(s)$ - 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 ;

Les sémaphores

Propriétés

- On ne peut initialiser un sémaphore avec une valeur négative,
  - Mais la valeur courante peut devenir négative.
- $E(s) = E_0(s)$ - 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.

Les sémaphores

Propriétés

- On ne peut initialiser un sémaphore avec une valeur négative,
  - Mais la valeur courante peut devenir négative.
- $E(s) = E_0(s)$ - 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.

Les sémaphores

Propriétés

- On ne peut initialiser un sémaphore avec une valeur négative,
  - Mais la valeur courante peut devenir négative.
- $E(s) = E_0(s)$ - 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.

Points importants

- Il est **interdit** de manipuler un sémaphore (ou un verrou) autrement qu'avec les primitives dédiées ;

Les sémaphores

Propriétés

- On ne peut initialiser un sémaphore avec une valeur négative,
  - Mais la valeur courante peut devenir négative.
- $E(s) = E_0(s)$ - 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.

Points importants

- Il est **interdit** de manipuler un sémaphore (ou un verrou) autrement qu'avec les primitives dédiées ;
- Un processus **doit** exécuter P avant d'entrer en section critique ;

Les sémaphores

Propriétés

- On ne peut initialiser un sémaphore avec une valeur négative,
  - Mais la valeur courante peut devenir négative.
- $E(s) = E_0(s)$ - 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.

Points importants

- Il est **interdit** de manipuler un sémaphore (ou un verrou) autrement qu'avec les primitives dédiées ;
- Un processus **doit** exécuter P avant d'entrer en section critique ;
- Un processus **doit** exécuter V en sortant de section critique ;

Les sémaphores

Propriétés

- On ne peut initialiser un sémaphore avec une valeur négative,
  - Mais la valeur courante peut devenir négative.
- $E(s) = E_0(s)$ - 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.

Points importants

- Il est **interdit** de manipuler un sémaphore (ou un verrou) autrement qu'avec les primitives dédiées ;
- Un processus **doit** exécuter P avant d'entrer en section critique ;
- Un processus **doit** exécuter V en sortant de section critique ;
- Si aucun processus n'est en section critique, alors il ne doit pas y avoir de processus bloqué par le sémaphore.

Les sémaphores

Exemple d'exécution

Les sémaphores

Points auxquels il faut prêter attention

- 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 pas 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.

Synchronisation entre processus

- Mécanisme d'exclusion mutuelle
  - Simple et efficace pour protéger une ressource,
  - Peut servir à la synchronisation entre processus.
- D'autres mécanismes plus complexes à définir pour la synchronisation.

Synchronisation entre processus

Nouveau mécanisme qui permet :

- De bloquer un autre processus ou lui-même ;
- D'activer un autre processus q en lui transmettant éventuellement une information.
  - Le signal d'activation n'est pas mémorisé ;
  - Le signal d'activation est mémorisé.

Ces mécanismes peuvent être à :

- Actions directes : le processus est désigné par son nom (*pid*) ou agit directement sur lui-même ;
- Actions indirectes : le processus manipule des noms de variables accessibles à d'autres processus.
  - Utilise des objets intermédiaires communs ;
  - Manipulation de ces objets par des fonctions spéciales dédiées ;
  - Deux types de synchronisation.
    - ★ Synchronisation par événements ;
    - ★ Synchronisation par sémaphore.

Synchronisation par évènements

- Fonctionnement
  - Le processus se bloque lorsque l'événement qu'il attend n'est pas encore arrivé ;
  - Le déclenchement d'un événement débloque le ou les processus qui l'attendent ;
  - Le processus activé exécute une fonction spécialement attaché.
- Deux façons de répondre à l'arrivée d'un événement
  - L'événement n'est mémorisé ;
    - ★ Un événement émis sans être attendu est perdu.
  - L'événement est mémorisé ;
    - ★ En général mémorisation d'une seule émission.
- 3 primitives de synchronisation
  - Blocage d'un processus sur l'attente d'un événement ;
  - Association d'une action à un événement ;
  - Envoi d'un événement à un processus.

Synchronisation par sémaphores

- Nouveau type de sémaphore : le sémaphore privé
  - Seul le propriétaire de s fait $P(s)$,
  - Les autres processus font $V(s)$.
- Mécanisme à mémoire grâce à $E(s)$ entier
  - Comptabilise le nombre exact d'événement (jetons),
  - Même si non encore attendu.
- Sémaphore initialisé à 0.
- Distinction des deux types de sémaphore
  - Sémaphore d'exclusion mutuelle : attaché à une ressource ;
  - Sémaphore privé : attaché à un processus.

Sémaphore privé

Exemple d'exécution

Les rendez-vous

- Permettent de synchroniser deux processus
  - Au même instant ;
  - À un endroit précis de leur code.
- Réalisation à partir de sémaphores privés.

Interblocage

Phénomène d'interblocage (*deadlock*)

Phénomène d'interblocage

- Phénomène fréquent,
- Différentes façons de le prendre en compte
  - On ignore le problème,
  - On le détecte et y remédie,
  - On les évite,
  - On les prévient.

Graphe d'allocation des ressources

- Rend compte de l'état actuel du système en ressources acquises et demandées ;
- Composé de deux noeuds et arcs
  - Les processus : des cercles ;
  - Les ressources : des rectangles contenant un point par exemplaire ;
  - Des arcs orientés ressource-processus : ressource allouée ;
  - Des arcs orientés processus-ressource : nouvelle demande d'une ressource.

Graphe d'allocation des ressources

Exemple

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

Réduction du graphe d'allocation des ressources

Règles de réduction

- Si une ressource ne possède que des arcs sortant, on les efface ;
- Si un processus ne possède que des arcs entrant, on les efface ;
- Si une ressource a des arcs sortant mais que chaque requête entrante peut être servie, on efface les flèches.

Exemple avec 4 processus et 3 ressources :

	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 – Allocation courante et besoins des 4 processus

Réduction du graphe d'allocation des ressources

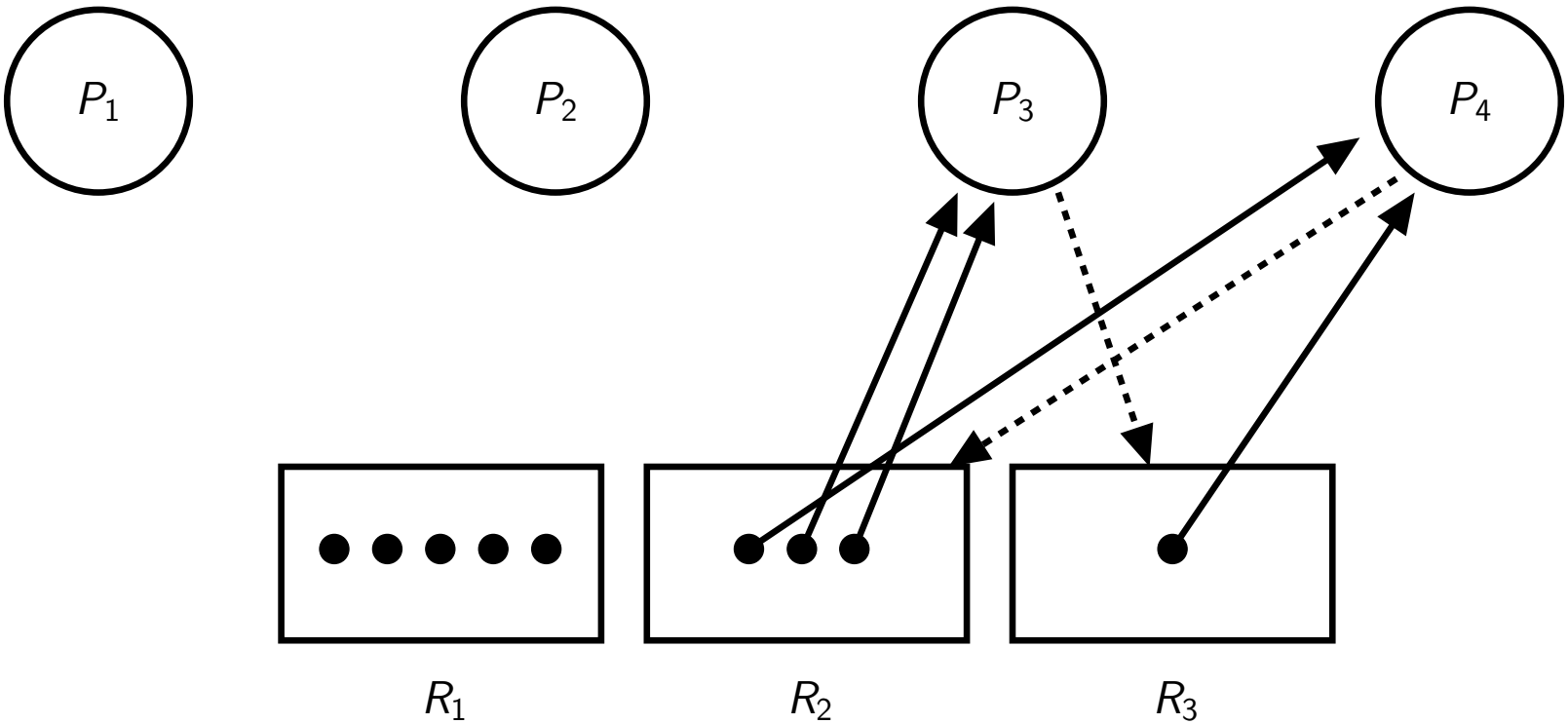
Exemple

	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

Réduction du graphe d'allocation des ressources

Exemple

	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



Interblocage

Détection

- Ne cherche pas à empêcher les interblocages ;
- Détecte les situations d'interblocage
  - Construction dynamique du graphe d'allocation des ressources ;
  - Ressources à exemplaire unique et graphe avec au moins un cycle ;
  - Ressources à exemplaires multiples et graphe avec au moins un cycle terminal.
- Quand effectuer cette détection ?
  - À chaque nouvelle demande de ressource ;
  - Périodiquement ;
  - Utilisation du processeur sous un seuil.

Interblocage

Guérison

- En cas de détection, suppression avec le moins d'effet de bord
  - Retirer temporairement une ressource ;
  - Restauration d'un état antérieur ;
  - Suppression de processus.

Interblocage

Prévention

Objectif : empêcher un interblocage avant l'allocation d'une ressource
À chaque demande de ressource, analyse de la situation

- Situation *sûre* si tous les processus peuvent terminer leur exécution ;
- Situation *non sûre* sinon.

Détermination de la situation

- Analyse du graphe, réduction, détection de cycle ;
- Autre méthode : *l'Algorithme du banquier* (Dijkstra - 1965)
  - Algorithme d'ordonnancement ;
  - Permet de déterminer si un état est *sûr* ou *non sûr* ;
  - Formalisation des données sous cette forme ;
    - ★ Matrice C des allocations courantes, $C(i,j)$ = nb. de ressources j possédé par i ;
    - ★ Matrice R des demandes, $R(i,j)$ = nb. de ressources j qu'il manque à i ;
    - ★ Vecteur A, $A(j)$ = nb. de ressource j actuellement disponibles ;
    - ★ Vecteur E, $E(j)$ = nb. total de ressource j dans le système.

Algorithme du banquier

Algorithme 27 : 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

Algorithme du banquier

Exemple

	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

Généralisation des sémaphores

Généralisation des sémaphores

Primitives P et V à k jetons

- Nouvelles primitives : $P(s,k)$ et $V(s,k)$;
- Deux implantations possibles ;
  - Allocation partielle autorisée ;

Algorithme 28 : 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

  - Allocation complète uniquement.

Algorithme 29 : 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

Généralisation des sémaphores

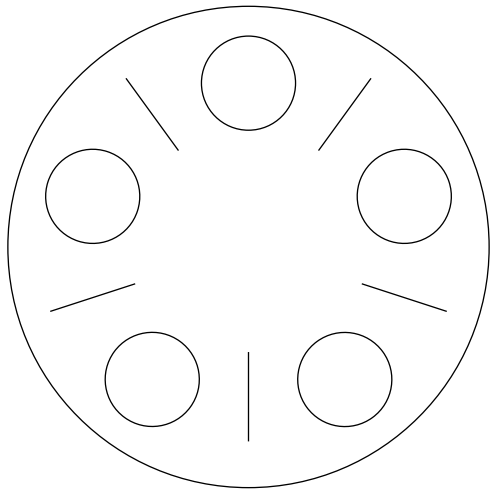
Tableau de sémaphore

- 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 ;
- Pas d'interblocage : soit on possède toutes les ressources, soit on en possède aucune ;
- Dans le système Unix/Linux, un objet sémaphore hybride est défini, croisement entre
  - Un sémaphore simple,
  - Un tableau de sémaphores,
  - Les sémaphores à k jetons.

Allocation globale des ressources

Les Philosophes aux spaghettis

Comment combiner des sémaphores mutex et privés pour résoudre un problème d'interblocage.



Protocole des philosophes :

- Chaque philosophe utilise pour manger **la** baguette à sa gauche et à sa droite ;
- À tout instant, un philosophe se trouve dans l'un des états suivants :
  - Il mange avec deux baguettes ;
  - Il attend la baguette de gauche, de droite ou les deux ;
  - Il pense sans utiliser de baguette.

Les Philosophes aux spaghettis

Graphe de changement d'état

Les Philosophes aux spaghettis

formalisation

- Chaque philosophe est un processus,
- Tous les processus utilisent le même algorithme,
- Les états des processus sont les suivants :
  - $C(i) = 0$, 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** $C(i-1) \neq 2$

- Cette stratégie utilise une allocation globale des baguettes,
- Le tableau des états C est partagé (consultation et mise à jour).

Les Philosophes aux spaghettis

formalisation

- Chaque philosophe est un processus,
- Tous les processus utilisent le même algorithme,
- Les états des processus sont les suivants :
  - $C(i) = 0$, 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 utilise une allocation globale des baguettes,
- Le tableau des états C est partagé (consultation et mise à jour).

Les Philosophes aux spaghettis

Prologue

Algorithme 30 : 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 $C(\varphi_{i-1}) \neq 2$ ***et*** $C(\varphi_{i+1}) \neq 2$ /* les voisins ne mangent pas */
alors
  $C(\varphi_i) \leftarrow 2$ /* mange */
sinon
  $C(\varphi_i) \leftarrow 1$ /* attente */
finsi
V(Mutex)

Les Philosophes aux spaghettis

Épilogue

Algorithme 31 : Épilogue (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 */
finsi
si $C(\varphi_{i+1}) = 1$ ***et*** $C(\varphi_{i+2}) \neq 2$
alors
  C(φ_{i+1}) $\leftarrow$ 2 /* mange */
finsi
V(Mutex)

Les Philosophes aux spaghettis

Prologue corrigé

Algorithme 32 : 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
  $C(\varphi_i) \leftarrow 2$ /* mange */
  V(S(φ_i)) /* on peut manger donc on se donne notre propre jeton */
sinon
  $C(\varphi_i) \leftarrow 1$ /* attente */
finsi
V(Mutex)
P(S(φ_i)) /* on se bloque si en attente sinon on passe car on a déjà notre jeton */

Les Philosophes aux spaghettis

Épilogue corrigé

Algorithme 33 : Épilogue (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 Philosophes aux spaghettis

Initialisation :

- C : tableau des états, tous initialisés à $C(i) = 0$,
- $S(\varphi_i)$: sémaphores privés de synchronisation, initialisés à 0,
- *Mutex* : sémaphore mutex de protection de C, initialisé à 1.

Algorithme 34 : Corps d'instructions de chaque philosophe

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

Les moniteurs

Les moniteurs (1972 Hoare)

- Objectif
  - Améliorer la lisibilité du code intégrant de la synchronisation,
  - Limiter les risques d'interblocage.
- Composition
  - Variables de synchronisation : *variables_conditions* ;
  - Ressources partagées ;
  - Procédures d'accès aux ressources
    - ★ Externes : primitives utilisées par les processus (prologue/épilogue) ;
    - ★ Internes : procédures internes réservées au moniteur.
  - Variables internes réservées au moniteur (comme $E(s)$).
- Les moniteurs sont communs à tous les processus.

Les moniteurs

Procédures internes

Deux procédures de synchronisation appelées par les primitives externes

- **WAIT** :
  - Met en attente le processus qui fait WAIT sur la *variable_condition*,
  - Libère le moniteur (très important, bloque le processus, pas le moniteur),
- **SIGNAL** :
  - Réveille 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.

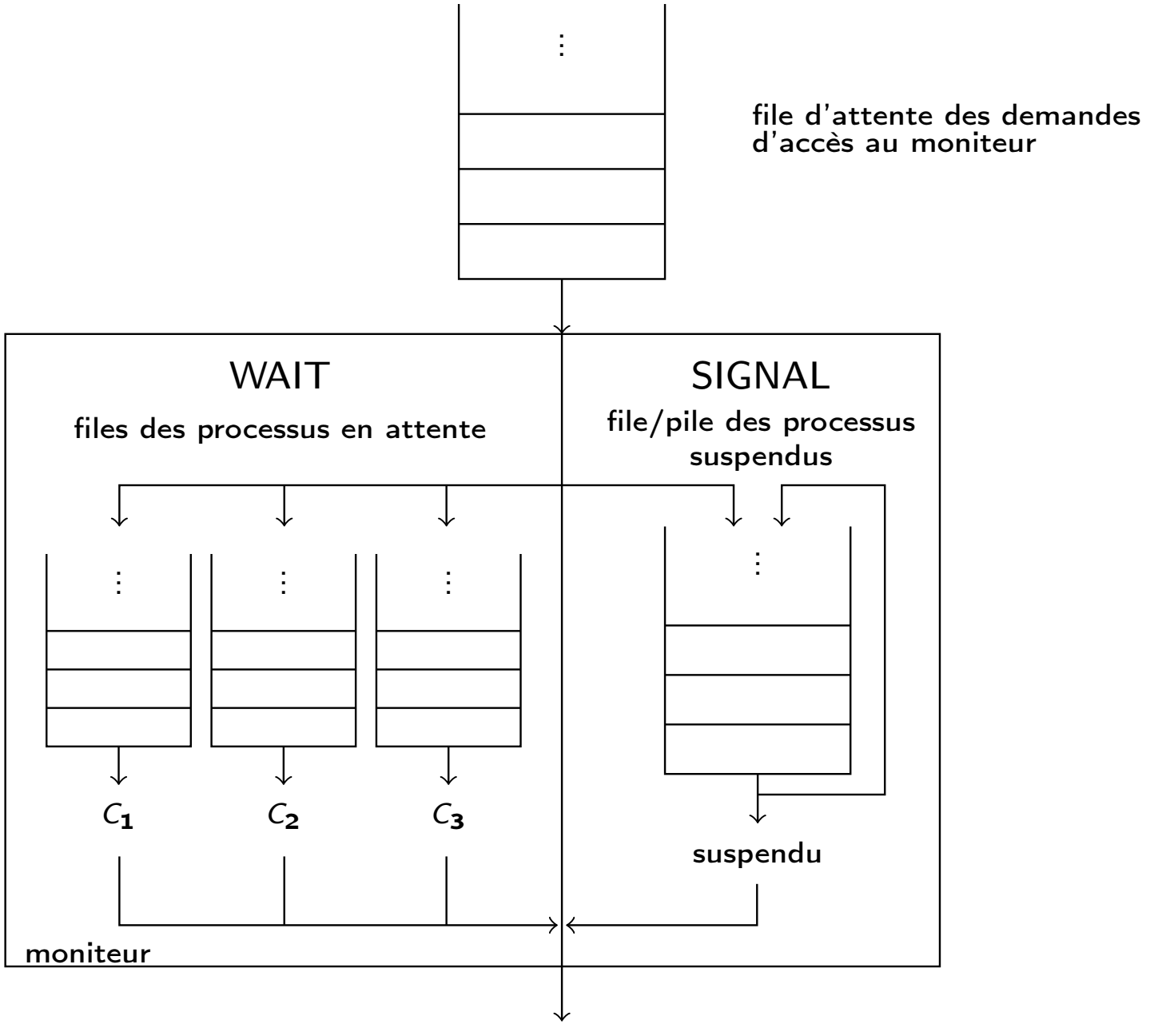
Convention d'utilisation : notation pointée

- variable_condition.WAIT
- variable_condition.SIGNAL

Les moniteurs

- Procédures externes
  - Correspondent aux prologues et épilogues
  - Utilisation de la notation pointée
    - ★ `moniteur.procédure(arguments)`
- Modélisation à l'aide de files d'attente
  - Une file pour l'accès au moniteur ;
  - n files pour les processus en attente, une par par *variable_condition* C_i ;
  - Une file ou un pile pour les processus suspendus par SIGNAL.

Les moniteurs



Les moniteurs

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 ;
  - Soit c'est le même processus qui poursuit son exécution si aucun processus n'a été réveillé.
- 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
  - Quand il sera en tête de file des processus suspendus ;
  - Et quand le processus actif dans le moniteur quitte le moniteur ou bien se met en attente sur une *variable_condition*.

Les moniteurs

Exemple : protection d'une ressource critique

Algorithme 35 : ressource_unique : moniteur (initialisation du 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

Algorithme 38 : Algorithme d'un processus utilisant un moniteur

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

Les moniteurs

Exemple : protection d'une ressource critique

Communication

Modèle du producteur-consommateur simple

Producteur-consommateur de ressources élémentaires

- 2 processus :
  - 1 producteur
    - ★ Produit des bonbons.
  - 1 consommateur
    - ★ Mange des bonbons.
  - Ressource critique : nombre de papier de bonbons
    - ★ Limité à N.

Modèle du producteur-consommateur simple

Producteur-consommateur de ressources élémentaires

- 2 processus :
  - 1 producteur
    - ★ Produit des bonbons.
  - 1 consommateur
    - ★ Mange des bonbons.
  - Ressource critique : nombre de papier de bonbons
    - ★ Limité à N.
- Objets de synchronisation

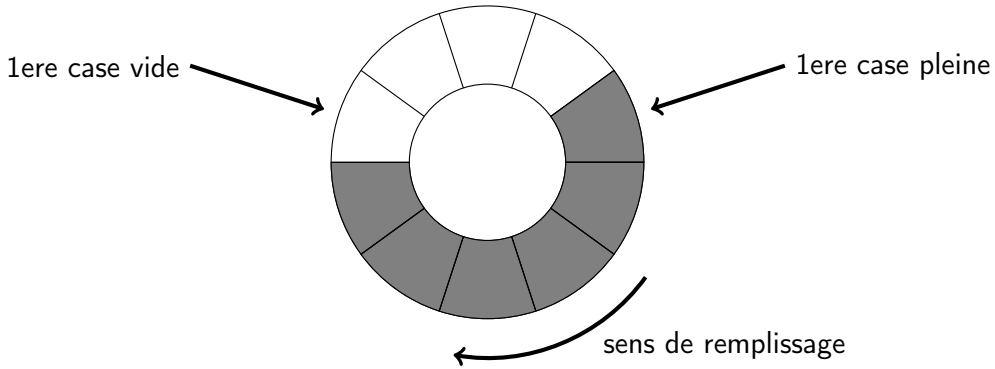
Modèle du producteur-consommateur simple

Producteur-consommateur de ressources élémentaires

- 2 processus :
  - 1 producteur
    - ⋆ Produit des bonbons.
  - 1 consommateur
    - ⋆ Mange des bonbons.
  - Ressource critique : nombre de papier de bonbons
    - ⋆ Limité à N.
- Objets de synchronisation
- Algorithmes des deux processus

Modèle du producteur-consommateur simple

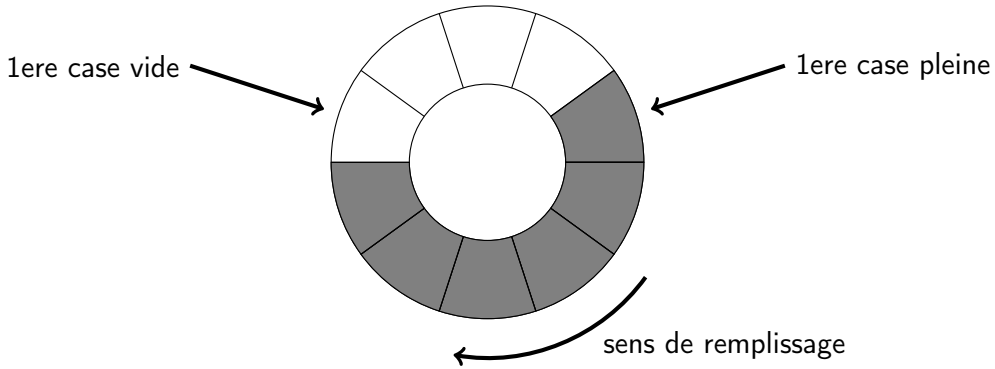
Producteur-consommateur de messages dans un tampon circulaire



- Ressources :
  - Nombre de cases vides ;
  - Nombre de cases pleines.
- Propriété : nombre de cases vides + nombre de cases pleines = N

Modèle du producteur-consommateur simple

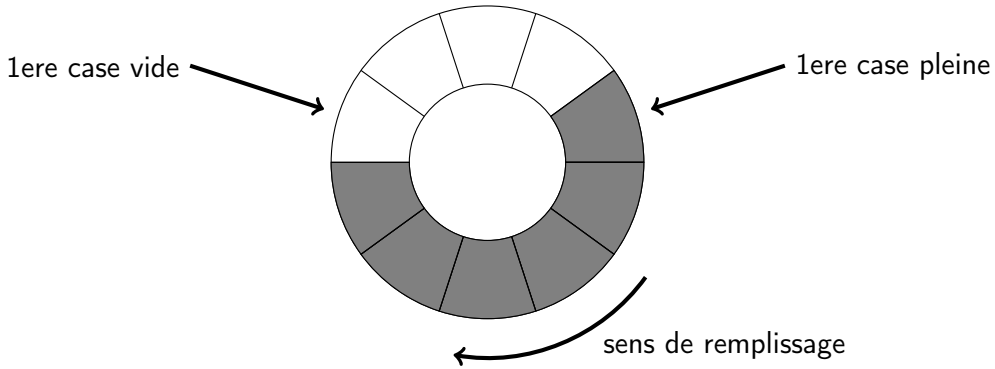
Producteur-consommateur de messages dans un tampon circulaire



- Ressources :
  - Nombre de cases vides ;
  - Nombre de cases pleines.
- Propriété : nombre de cases vides + nombre de cases pleines = N
- Objets de synchronisation

Modèle du producteur-consommateur simple

Producteur-consommateur de messages dans un tampon circulaire



- Ressources :
  - Nombre de cases vides ;
  - Nombre de cases pleines.
- Propriété : nombre de cases vides + nombre de cases pleines = N
- Objets de synchronisation
- Algorithmes

Modèle du producteur-consommateur simple

Producteur-consommateur géré par un moniteur

- Définition des variables du moniteur

Modèle du producteur-consommateur simple

Producteur-consommateur géré par un moniteur

- Définition des variables du moniteur

Algorithme 39 : moniteur

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

Modèle du producteur-consommateur simple

Producteur-consommateur géré par un moniteur

- Définition des variables du moniteur
- Primitive externe (prologue)

Modèle du producteur-consommateur simple

Producteur-consommateur géré par un moniteur

- Définition des variables du moniteur
- Primitive externe (prologue)

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

Modèle du producteur-consommateur simple

Producteur-consommateur géré par un moniteur

- Définition des variables du moniteur
- Primitive externe (prologue)
- Primitive externe (épilogue)

Modèle du producteur-consommateur simple

Producteur-consommateur géré par un moniteur

- Définition des variables du moniteur
- Primitive externe (prologue)
- Primitive externe (épilogue)

Procédure `Prélever(`*message M*`)`

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

Modèle du producteur-consommateur simple

Producteur-consommateur géré par un moniteur

- Définition des variables du moniteur
- Primitive externe (prologue)
- Primitive externe (épilogue)
- Algorithme

Modèle du producteur-consommateur simple

Producteur-consommateur géré par un moniteur

- Définition des variables du moniteur
- Primitive externe (prologue)
- Primitive externe (épilogue)
- Algorithme

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

Modèle du producteur-consommateur multiple

- N producteurs ;
- M consommateurs ;
- Nécessité d'accès en exclusion mutuelle aux variables communes

Modèle du producteur-consommateur multiple

- N producteurs ;
- M consommateurs ;
- Nécessité d'accès en exclusion mutuelle aux variables communes
- Algorithmes

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)

Autres modèles

- Communication par boîte aux lettres ;
  - Seul le propriétaire peut lire ;
  - Tous les autres processus peuvent écrire.

Autres modèles

- Communication par boîte aux lettres ;
  - Seul le propriétaire peut lire ;
  - Tous les autres processus peuvent écrire.
- Sémaphores à messages ;
  - Structure :
    - ★ Un entier ;
    - ★ Une file d'attente de processus ;
    - ★ Une file de messages.
  - Primitives étendues :
    - ★ *P(s, message_reçu)* ;
    - ★ *V(s , message_émis)*.

Vie d'un processus, synchronisation, communication sous Unix

Les processus

Attributs

Les processus

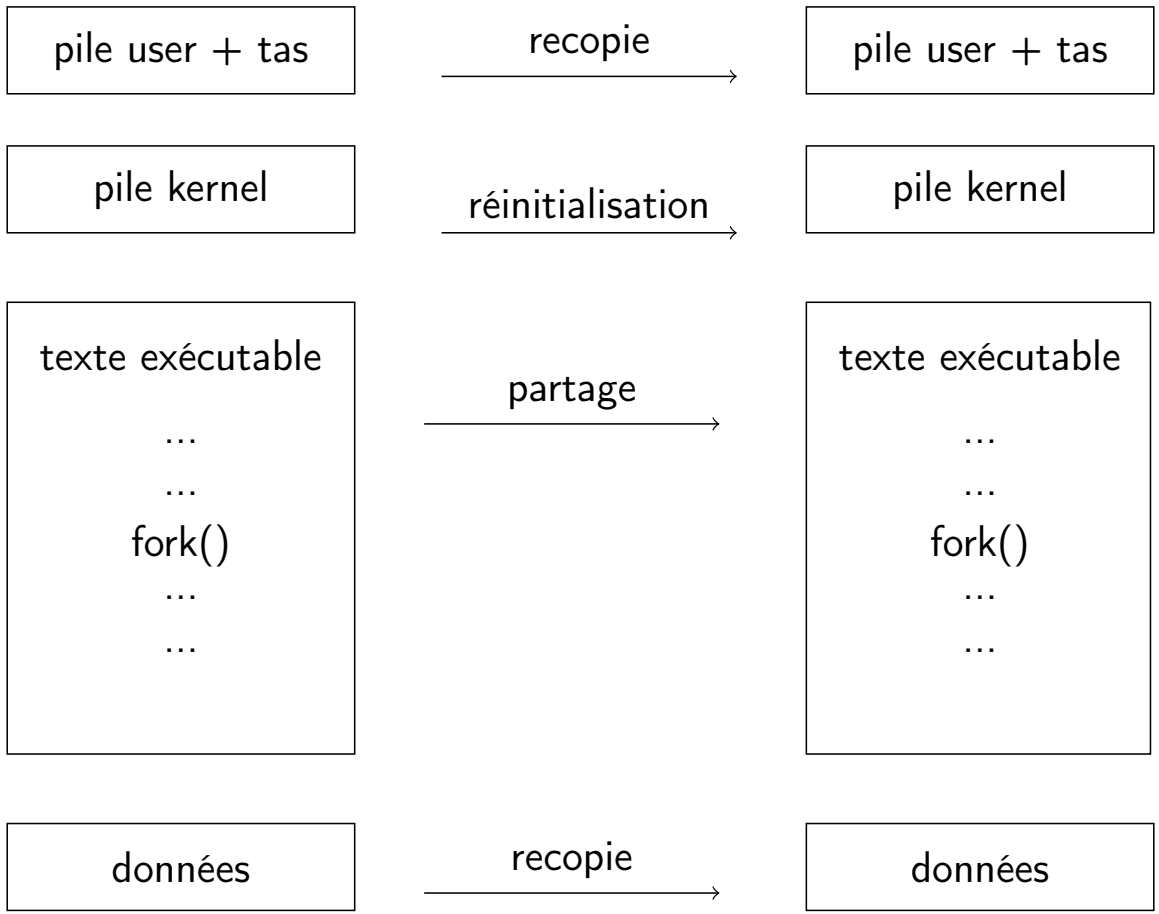
Attributs

- Identité
  - Utilisateur ;
  - Propriétaire ;
    - ★ Droits ;
    - ★ Sticky bit
  - User id : uid ;
    - ★ ruid ;
    - ★ euid.
  - Group id : gid ;
- Un identifiant : pid
- Une image mémoire ;
- Liste des fichiers ouverts ;
- Privilèges liés à l'utilisateur ;
- Variables liées à l'ordonnancement.

Création dynamique de processus : fork()

- Création d'un processus sous Unix/Linux par clonage ;
- Utilisation de la fonction `fork()` ;
- Création d'une relation le filiation
  - Le processus créateur : *processus père* ;
  - Le processus créé : *processus fils*.
- Héritage des attributs
  - Les privilèges ;
  - Les fichiers ouverts ;
  - Le texte exécutable ;
  - Les valeurs des variables.

Création dynamique de processus : fork()



Création dynamique de processus : fork()

- 2 processus identiques, comment les distinguer ?

Création dynamique de processus : fork()

- 2 processus identiques, comment les distinguer ?
- Valeur de retour de *fork()*
  - Processus père : `pid` du processus fils créé ;
  - Processus fils : 0.

Création dynamique de processus : fork()

- 2 processus identiques, comment les distinguer ?
- Valeur de retour de *fork()*
  - Processus père : `pid` du processus fils créé ;
  - Processus fils : 0.
- Exemple d'utilisation classique de *fork()*

```
...
/* 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 */
...
```

Création dynamique de processus : fork()

L'algorithme de fork

Terminaison de processus : exit()

- Terminaison propre et normale d'un processus : `exit()` ;
  - Termine le processus ;
  - Envoie un message au processus *père*.
- Utilisation classique
  - `exit(status)`
  - Valeur 0 : terminaison normale ;
  - Valeur $\neq 0$: terminaison sur erreur.
- Attention :
  - À tout moment un processus doit posséder un processus père ;
  - Sauf le processus `init`.

Terminaison de processus : exit()

L'algorithme de exit()

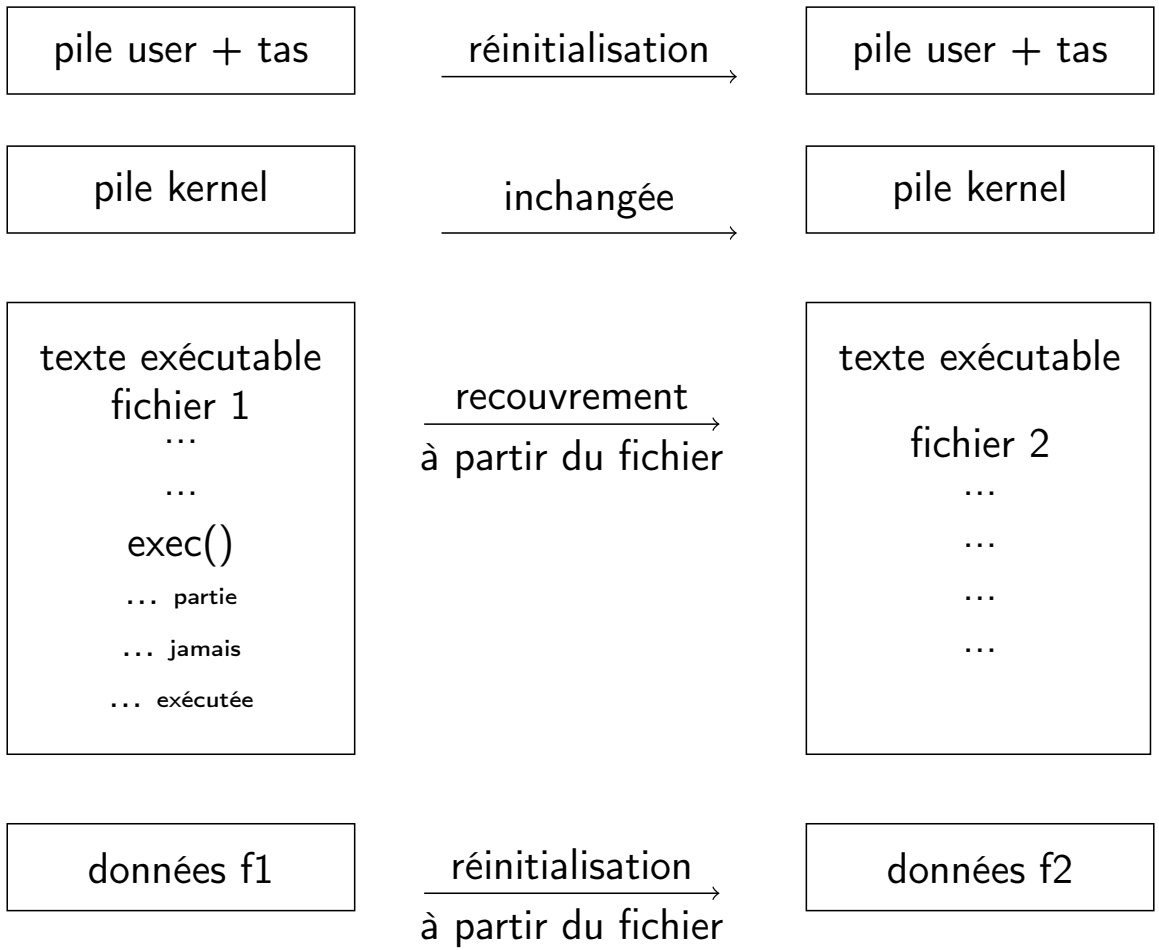
Algorithme 42 : exit()

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

Recouvrement de processus : exec()

- Une famille de fonction :
  - `execl()`, `execlp()`, `execle()`, `execv()`, `execvp()`, `execvpe()`
- Effectue un recouvrement de l'image mémoire d'un processus
  - Remplace de *contenu* d'un processus par un autre ;
  - Code exécutable effacé et remplacé par un autre ;
  - Données réinitialisées ;
  - Le *nouveau* programme reprend à son début ;
  - Impossible de revenir en arrière.
- Généralement utilisé conjointement à `fork()`
  - D'abord appel à `fork()` ;
  - Puis exécution d'un nouveau programme dans le processus fils avec `exec()`.

Recouvrement de processus : exec()



Recouvrement de processus : exec()

- Paramètres :
  - Nom du fichier à charger (à exécuter) ;
  - Arguments au programme ;
  - Exemple :
    - `execl(nom_fichier, nom_fichier, arg1, ..., NULL)` ;
  - Lien avec `argv[]` de `main()`.
- Différentes façons de passer les paramètres
  - `execl()` : paramètres sous forme de liste terminée par `NULL` ;
  - `execv()` : paramètres sous forme d'un tableau dont le dernier élément est `NULL` ;
  - `exec*p()` : si le fichier cible ne commence pas par / alors recherche dans le `PATH` ;
  - `exec*e()` : en plus des arguments, spécifie des variables d'environnement.

Recouvrement de processus : exec()

L'algorithme d'exec

Synchronisation père-fils : wait()

- Rappels sur `exit()` :
  - Envoie une valeur au processus père ;
  - Envoie un signal au processus père.
- Le processus père peut attendre ce signal
  - En utilisant la fonction `wait()` ;
  - Attente explicite de la terminaison d'un fils ;
  - Utilisation :
    - ★ `pid = wait(&status)` ;
    - ★ `pid` : pid du fils terminé ;
    - ★ `status` : « presque » la valeur de retour
    - ★ La valeur de retour est sur l'octet de poids faible
    - ★ Récupération par la macro `WEXITSTATUS(status)`.
- Si attente sur `wait()`, alors réveil dès qu'un fils meurt (n'importe lequel) ;
- Retour de `wait()` :
  - `pid` si un fils est terminé ;
  - -1 et signal `ECHILD` si pas de fils.

Synchronisation père-fils : wait()

L'algorithme de wait

Algorithme 43 : 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

Synchronisation par événement : kill, pause, signal

- Mécanisme général de synchronisation directe par événement ;
- Synchronisation asynchrone :
  - L'événement peut arriver n'importe quand ;
  - Le processus sensible ne doit pas forcément être endormi et en attente.
- 3 instructions de base :
  - `kill` : envoie un signal particulier à un processus particulier ;
  - `signal` : associe une fonction à la réception d'un signal ;
  - `pause` : met en pause le processus en attente d'un signal.
- Les signaux :
  - Liste définie par le système (`/usr/include/bits/signum.h`) ;
  - Trentaine de signaux définis, réservés ;
  - Deux signaux laissés à l'utilisateur ;
  - Désignés par leur valeur ou leur nom.

Synchronisation par événement : kill, pause, signal

Exemple de signaux

```
#define SIGHUP          1       /* Hangup (POSIX).  */
#define SIGINT          2       /* Interrupt (ANSI).  */
#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 (PO
#define SIGCONT         18      /* Continue (POSIX).  */
#define SIGSTOP         19      /* Stop, unblockable (POSIX).
#define SIGPWR          30      /* Power failure restart (Syste
```

Le signal SIGCHLD

- Gestion par défaut : ignorer le signal, ne rien faire ;
- Lié à la fonction `wait()` ;
  - Faire appel à `wait()` dans la fonction associée au signal `SIGCHLD`.
- Comportement par défaut modifié (POSIX.1-2001)
  - Ignorer explicitement le signal de terminaison ;
  - `SIGCHLD` traité par le noyaux ;
  - Plus besoin de faire `wait()` pour le processus père.

Les signaux de terminaison

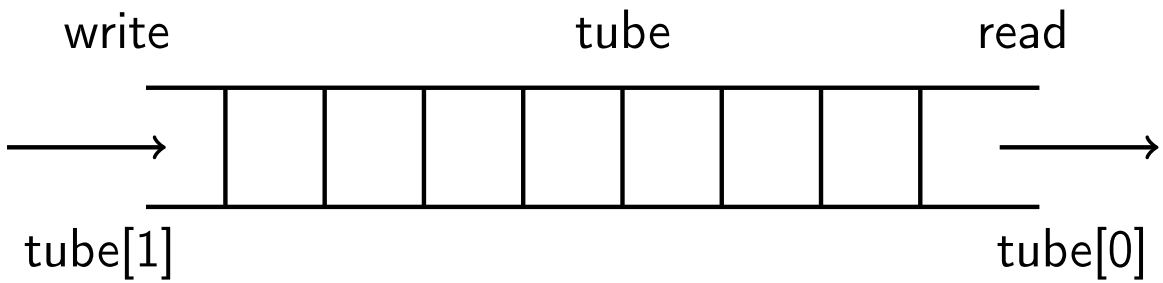
5 signaux de terminaison

- `SIGTERM` : demande polie de terminaison. Peut être bloqué, capturé ;
- `SIGINT` : similaire à `SIGTERM`, peut être généré par *Ctrl-C*, peut être ignoré ou capturé mais même dans ces le processus se terminera ;
- `SIGQUIT` : terminaison du programme et génération d'un `core dump`, orienté débuggage, peut être généré par *Ctrl-* ;
- `SIGKILL` : terminaison immédiate du processus, ne peut être ignoré ou capturé, pas de *core dump* ;
- `SIGHUP` : initialement émis quand une connexion est perdu avec le terminal avec terminaison du processus. Actuellement capturé pour des demandes de passage en démon ou relecture de la configuration.

Communication inter-processus

Les tubes

- Zone de mémoire gérée en producteur-consommateur
  - Accès en exclusion mutuelle ;
  - Il faut **toujours au moins** 1 lecteur **et** 1 écrivain ;
  - Lecture bloquante sur tube vide ;
  - Écriture bloquante sur tube plein ;
- Création par la fonction `pipe(int[2])`
  - Ouverture simultanée de deux descripteurs : un en lecture (0), un en écriture (1) ;
  - Impossible d'utiliser `open()` sur un tube
- Exemple : `int tube[2]; pipe(tube);`



Communication inter-processus

Les tubes

- Utilisation comme un fichier
  - Utilisation de `read()` et `write()`
  - `read(tube[0], ...)` ou `write(tube[1], ...)`
- Fermeture d'un descripteur avec `close()` ;
- Contrôle via la fonction `fnctl()` :
  - Modification ou consultation des propriétés ;
  - Taille, occupation, état bloquant, etc.
- Exemple d'utilisation classique

```
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])
    ...
}
```

- Règle importante : toujours fermer les descripteurs inutiles ;

Communication inter-processus

Les tubes cassés

- Deux cas de cassure du tube :
  - Plus d'écrivain ;
  - Ou plus de lecteur.
- Gestion différente de la situation

Communication inter-processus

Les tubes cassés

- Deux cas de cassure du tube :
  - Plus d'écrivain ;
  - Ou plus de lecteur.
- Gestion différente de la situation
  - Plus d'écrivain

Communication inter-processus

Les tubes cassés

- Deux cas de cassure du tube :
  - Plus d'écrivain ;
  - Ou plus de lecteur.
- Gestion différente de la situation
  - Plus d'écrivain
  - Plus de lecteur

Communication inter-processus

Tubes et processus de filiation différente

- Descripteurs non partagés (pas de fork) ;
- Obligation de passer par le système de fichier ;
  - Création d'un vrai fichier, avec un nom ;
  - Ouverture par son nom ;
  - Droits particuliers : `prw-rw-r-` ;
- Utilisation
  - Création avec `mkfifo()` ;
  - Ouverture du fichier soit en lecture soit en écriture ;
  - Utilisation des fonctions `read()`, `write()` et `close()`.

Communication inter-processus

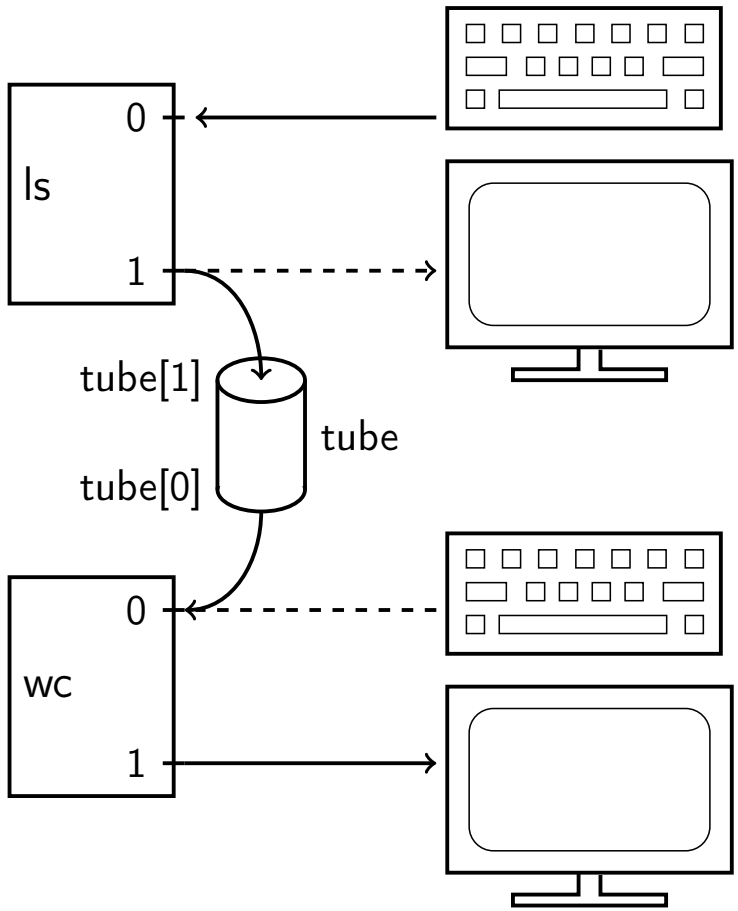
La fonction dup2

- Méthode précédente : communication explicite
  - Les programmes connaissent l'existence du tube
    - ★ Soit héritage des descripteurs par filiation ;
    - ★ Soit connaissance du nom du fichier d'échange.
- Comment faire pour deux programmes inconnus ?
  - Sources non disponibles pour modifier leur code ;
  - Modification trop lourde à mettre place.
  - Exemple : réaliser l'équivalent de « `ls | wc` »
- Solution : `dup2(int descTo, int descFrom)`
  - `descFrom` : le descripteur qui sera dupé (qui sera redirigé) ;
  - `descTo` : le descripteur vers lequel sera redirigé `descFrom`.

Communication inter-processus

La fonction dup2

- `dup2(tube[0], 0);`
- `dup2(tube[1], 1).`



- Par `exec()`, les descripteurs 0, 1 et 2 sont conservés (ni fermés, ni réinitialisés).

Communication inter-processus

Les IPC : Inter Process Communication

- Outils de synchronisation et de communication
  - Les sémaphores ;
  - Les zones de mémoire partagées ;
  - Les files de message.
- Objets indépendants des processus
  - Rémanents : survivent au processus créateur ;
  - Possèdent un identifiant ;
  - Possèdent des droits d'accès.
- Famille de fonctions
  - Accès ou création : `*get()` – `semget()`, `shmget()`, `msgget()` ;
  - Contrôle (dont destruction) : `*ctl()` – `semctl()`, `shmctl()`, `msgctl()` ;
  - Utilisation : `semop()`, `shmat()`, `shmdt()`, `msgsnd()`, `msgrcv()` ;

Communication inter-processus

Les IPC

- Objets globaux au système, il faut assurer
  - Désignation unique et correcte ;
    - ⋆ Utilisation d'une clé
  - Création unique.
    - ⋆ Soit le processus créateur est fixé par le programmeur ;
    - ⋆ Soit tous les processus essaient de créer l'objet et seul le premier réussit.
- La fonction de création retourne un identifiant local pour les fonctions de manipulation ;
- Possibilité de surveillance par le shell
  - Pour la consultation des objets existants : `ipcs` ;
  - Pour la destruction des objets : `ipcrm`.

Communication inter-processus

Les IPC – la création

- Processus de même filiation (*via* `fork()`)
  - Génération gérée par le système d'exploitation ;
  - Valeur spéciale de clé : `IPC_PRIVATE`
  - Exemple : `id = semget(IPC_PRIVATE, arguments, droits)`
- Processus de filiation différente
  - Génération de la clé à partir de deux données
    - ★ Un nom de chemin/fichier ;
    - ★ Un identificateur de ressource (un caractère) ;
    - ★ Utilisation de la fonction spéciale `ftok()`.
- Arguments spécifiques au type d'objet
  - Nombre de sémaphores, taille d'une zone de mémoire etc.
- Droits
  - Droits d'accès classique *rwx* en numérique (0644) ;
  - Demande de création en cas d'inexistence : `IPC_CREAT` ;
  - Demande d'erreur si l'objet est déjà créé : `IPC_EXCL`.

Communication inter-processus

Les IPC : les sémaphores

- Modèle implanté
  - Tableau de sémaphores à k jetons ;
  - P et V indivisible sur un ensemble de sémaphores ;
  - P et V sont à k jetons ;
  - P peut être rendu non bloquant ;
  - Mécanisme *d'undo* possible pour défaire des actions.
- Création : `semid = semget(IPC_PRIVATE, 1, IPC_CREAT|0644)`
- Manipulation : `semop(semid, *sembuf, options)`

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

Communication inter-processus

Les IPC : les sémaphores

- Exemple : une primitive P

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

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

Communication inter-processus

Les IPC : les sémaphores

- Fonction de contrôle : `semctl()`
- Paramètres
  - Identifiant ;
  - Indice du premier sémaphore concerné ;
  - Commande à effectuer ;
    - ★ `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.
  - Paramètres pour la commande
    - ★ Énumération `union semun`.

Communication inter-processus

Les IPC : les sémaphores

```
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) */
};
```

- Exemple : initialisation des sémaphores d'un tableau de 3 sémaphores

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

- Exemple : initialisation du 4e élément du tableau à 10

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

Communication inter-processus

Les IPC : les zones de mémoire partagées (`Shared Memory`)

- Zone de mémoire de taille donnée à la création ;
- Zone non structurée : tableau d'octets ;
- Consultable et modifiable ;
- Modifications visibles par tous les processus ;
- Création
  - Clé d'identification ;
  - Taille en octets de la zone ;
  - Droits et options ;
  - Fonction `shmget()` ;
- Manipulations
  - shmat() : attache la zone dans l'espace mémoire local ;
  - shmdt() : détache la zone le l'espace mémoire local.
- Contrôle : `shmctl()`
  - Commandes : `IPC_RMID`, `IPC_STAT`, etc ;
  - Destruction effective au dernier détachement.

Communication inter-processus

Les IPC : les files de messages

- Modèle producteur-consommateur ;
- Files de messages gérées FIFO comme les tubes ;
- Structure d'un message
  - Un type : entier long (numéro de file, > 0) ;
  - Une donnée non structurée : le contenu du message.
- Création
  - `msgget(clé, droits)`
- Manipulation
  - Envoi de message : `msgsnd()` ;
  - Réception d'un message : `msgrcv()` ;
- Contrôle : `msgctl()`
  - Commandes : `IPC_RMID`, `IPC_STAT`, etc
- Point important
  - On ne peut transmettre que des valeurs, pas des pointeurs.