Cours avancé de S.E.

Andreea Dragut & Vincent Risch

Univ. Aix-Marseille

2012

Plan général

1. Gestion de la mémoire
   - Allocation (statique, dynamique)
   - Gestion manuelle, automatique

   $\Rightarrow$ Architecture système

   $\Rightarrow$ Gestion système

   $\Rightarrow$ Intéraction processus (IPC)
2. Gestion du temps
   - Temps partagé, ordonnanceur
   - Algorithmes
   - Cadre d'illustration

Plan de ce cours – Mémoire – Allocation, gestion

- Architecture mémoire système
  - Structuration
  - Segments
  - Pages
- Gestion système
  - Mémoire virtuelle
  - Mémoire réelle
  - Tables de correspondance
  - Gestion mémoire processus utilisateur
  - Appels systéme, suivi après `exec()`
- Intéraction processus (IPC)
  - Mappage mémoire E/S
  - Mémoire partagée
  - Mémoire partagée et IPC
  - Synchronisation et exclusion mutuelle

Architecture mémoire système

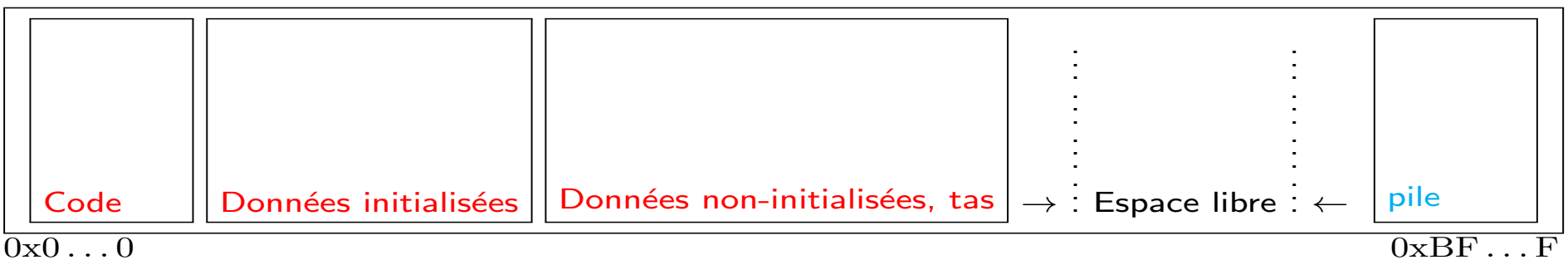
- En général –
  - mémoire « vue depuis le noyau »
  - mémoire « vue depuis le processus utilisateur »
- spécificités :
  - séparation,
  - droits d'opération
  - zones accessibles
  - consistance
  - sécurité

Structuration mémoire – vue depuis le noyau

- Descripteur de processus
  - Mapping mémoire
  - Descripteurs fichiers ouverts
  - Répertoire courant
  - Pointeur pile noyau
- Pile noyau
  - petite par défaut
  - croît dans des cas extrês d'appels imbriqués d'interruptions/exceptions
- Table processus
  - Table associative de descripteurs de processus, indexée par les PID
  - Arbre doublement chaîné (liens vers les enfants et vers les parents)

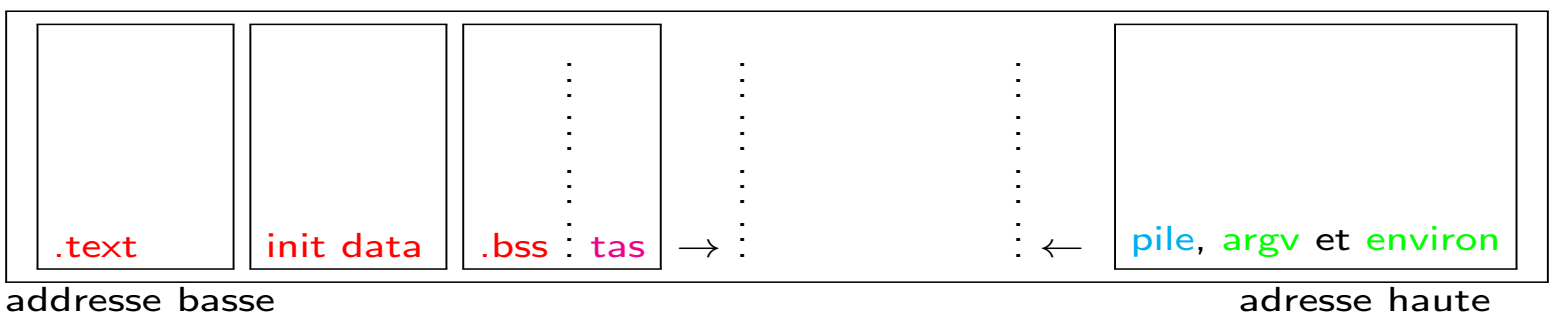
Structuration mémoire – vue depuis le processus

- Allouée et initialisée lors du chargement et exécution d'un processus
- Les accés mémoire en mode utilisateur sont restrictionnés à cet espace d'adressage



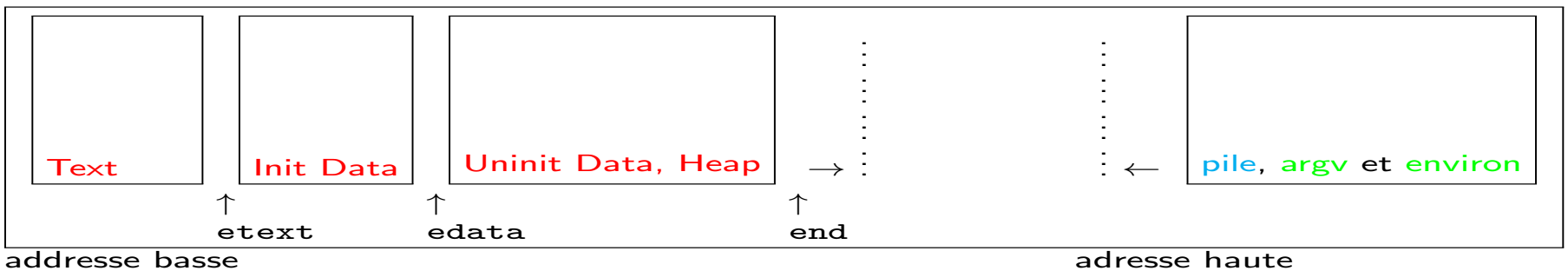
Segments mémoire processus

- Code (ou bien text)
  - Linux – format ELF pour fichiers objet (`.o` et exécutables)
- Données initialisées
  - zones uniquement en lecture `.rodata`
  - variables globales initialisées (et C `static`)
- Autres données
  - variables globales non-initialisées – mises à zéro au démarrage du processus (segment `.bss`)
  - Tas — allocation dynamique (`malloc()`)
- Pile
  - "frames" pour les appels de fonction
  - arguments, variables locales (C `automatic`)



Savoir qui va où

- Le système — renseigner sur les extrémités des segments
- Variables externes `etext`, `edata`, `end`
- Devoir du cours : faites `man 3 end` pour mieux comprendre



Appel système `brk()`

- Le « program break »– fin du segment de données (noninitialisés et tas)
- `brk()` – augmente ou diminue tout simplement la taille du segment de données (noninitialisés et tas)

```
#include <unistd.h>

int brk(void *adresse);

void *sbrk(intptr_t increment);
```

- donc modifient la variable externe end
- `sbrk()` ajoute un déplacement et renvoie la valeur précédente du « program break »
- Ne plus utiliser ces appels – obsolètes pour l'utilisateur
- Utiliser `malloc()` (ou resp. `new`), etc.

Et la pile ?

- pour la pile – pas de variable externe
- par contre – limitations
- « resource limits » — limitations gérées par le système
  - taille max de la pile – `RLIMIT_STACK`
  - nombre max de processus pour un user id donnée — `RLIMIT_NPROC`
  - etc.
- Devoir du cours : faites `man getrlimit` pour mieux comprendre

Illustration

- sur la pile on met entre autres les variables locales auto(matic) des fonctions (et non pas celles static — car celles-ci auront une seule copie partagée par toutes les instances des fonctions, et se retrouvent dans le .data, etc.)
- sur le tas on fait de la place avec l'allocation dynamique

```
int maFonction() {
  int * p;     // le pointeur p est mis sur la pile
  p = new int;// l'int pointe par p est mis sur le tas
  ...
  delete p;    // l'int pointe par p disparait du tas
               // mais p lui-meme vit encore sur la pile
  return 0;
} // maintenant p disparait lui aussi de la pile
```

Comment fonctionne la pile ?

- à chaque appel de fonction : de la place pour les paramètres, l'adresse de retour, et puis pour les variables locales
- si la fonction en appelle une autre, la pile est augmentée, on empile les données pour la nouvelle fonction appelée, ...
- lors du retour de la fonction, on dépile ce qui avait été empilé pour la fonction en question, pour restaurer la pile exactement comme elle était avant.
- Pourquoi il ne faut pas renvoyer l'adresse d'une variable locale au retour d'une fonction (une référence C++ non plus) $\longleftarrow$ puisqu'elle disparaît au retour de la fonction.
- appellé auto(matic) : le type de stockage (storage class) automatiquement géré par le système : sans intervention explicite du programmeur.

Pour voir

- Pour la pile : utilisant *strace*, au fur et à mesure qu'un programme s'exécute : les appels à `new` sont traduits en appels à `brk()`.
- en utilisant le répertoire /proc/<PID> pour un PID d'un processus affichage des adresses virtuelles pour tous les segments de text, données, etc.

Pour voir

Compiler, et ensuite *strace*'er le programme avec

strace -e trace=brk nomProg

```
int main() {
  maFonction();// l'operateur new s'appuie sur le brk
  pause();//pr avoir le temps de regarder les adresses
  return 0;
}
```

on voit le brk du new avec l'adresse brk(0x522000) — regarder ensuite dans /proc/15616/maps (avec *15616* son PID), on retrouve (extrait commenté) les adresses début-fin et protections, y compris la *0x522*

```
000000400000-000000401000 r-x   --- instructions -- read, execute
000000500000-000000501000 rw-   --- donnees      -- read, write
000000501000-000000522000 rw-   --- tas (-> brk) -- read, write
7ffffffe7000-7fffffff000 rw-    --- pile         -- read, write
```

l'appel brk() rendant la fin de la nouvelle zone — le *0x522* – addresses virtuelles.

Comparaison

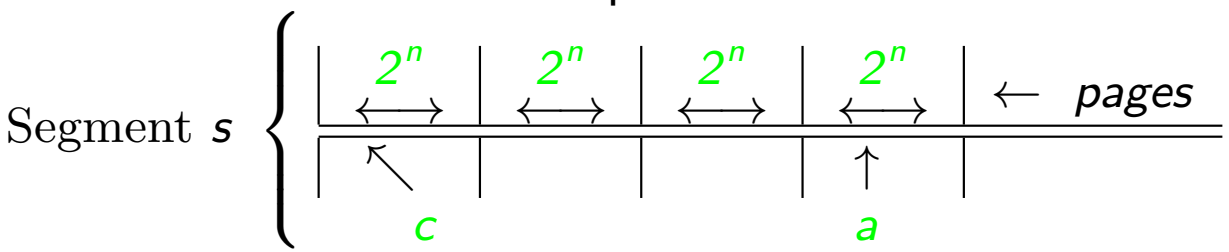
- Faire tourner deux autres variantes du programme
  - une variante avec les appels à `new` et `delete` mis en commentaire — on verra alors que
    - le tas disparaît de l'affichage de maps,
    - l'appel à `brk()` disparaît de l'affichage de *strace*
  - une autre variante qui fait des appels récursifs : `maFonction()` s'appelle elle-même en décroissant un paramètre d'entrée — on voit alors comme la taille du segment de pile augmente vers le bas, i.e. la première adresse virtuelle est plus petite que dans la variante sans beaucoup d'appels.
- appel : Ces adresses dites virtuelles ont un sens **uniquement** pour le processus en cours, pour lequel elles sont définies.

Pagination

- Les processus utilisent de la mémoire virtuelle
  - Un grande plage d'adresses mémoire
  - Simplification de la gestion de la mémoire au niveau du processus
  - Traduction automatique en adresses réelle par le matériel (CPU, MMU)
- Méchanisme de pagination
  - Unité d'allocation et de protection – la page
  - Taille fixe (2^n), e.g. 4KB, 2MB
  - Le noyau mappe les pages physiques sur les pages virtuelles, différemment pour chaque processus
- Méchanisme clé pour assurer la séparation logique entre processus
  - Pour un processus donné, rend invisible la mémoire du noyau et celles des autres processus
  - Protection et séparation efficace au niveau de l'adressage

Comment interpréter une adresse virtuelle ?

- Mémoire divisée en pages de taille 2^n
- Un segment de mémoire s occupe plusieurs pages
- Soit c l'adresse de début de s
- Soit une adresse virtuelle a qui référence un endroit *dans le segment s*



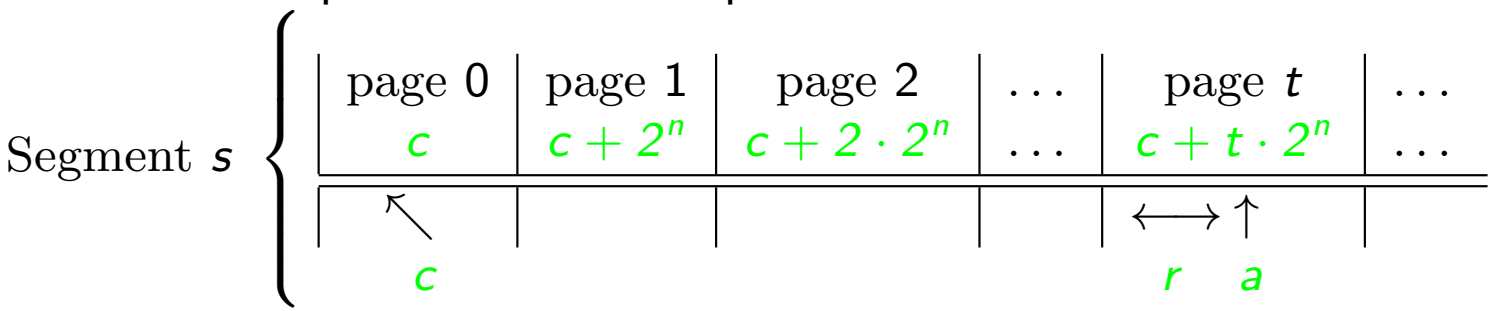
- sur quelle page t du segment s se trouve l'adresse a ?

$$\Longrightarrow t = \frac{a - c}{2^n}$$

- et plus précisément dans la page ?

Comment interpréter une adresse virtuelle ? (suite)

- Donc : a « tombe » dans la page $t = \dfrac{a - c}{2^n}$
- Mais t doit être un entier
- Et $a - c$ n'est pas forcément une puissance de 2



- $a = c + t \cdot 2^n + r$

$\Longrightarrow$ r = le reste de la division de $a - c$ par 2^n

Comment interpréter une adresse virtuelle ? (suite)

- Donc : $a = c + t \cdot 2^n + r$
- Comment simplifier ces calculs, sachant qu'on utilise des bits ?
  - aligner les tailles – puissances de 2
  - séparer les termes $\Longrightarrow$ l'addition devient « concaténation »
- Exemple :
  - une page a une taille de $4096 = 2^{12}$ donc $n = 12$
  - les adresses sont sur 32 bits
  - un segment contient $128 = 2^7$ pages, et commence à l'adresse $c = 0xAB180000 = \underbrace{10101011000110\ldots0}_{13\text{ bits}}$
  - $13 + 7 + 12 = 32$
  - alors l'adresse $a = 0xAB183005 \longrightarrow$ adresse $r = 5$ dans la page $t = 3$: $a = 0xAB183005 = \underbrace{1010101100011}_{c \to 13\text{ bits}}\ \underbrace{0000011}_{t \to 7\text{ bits}}\ \underbrace{000000000101}_{r \to 12\text{ bits}}$

Correspondance mémoire virtuelle — mémoire réelle

- Pour aller vite – matériel – CPU MMU/TLB – *translation lookaside buffer*
- Mais taille limitée

$\Longrightarrow$ logiciel – tables de correspondance, gestion *swap* sur disque

- Une instruction machine accès mémoire —
  - d'abord MMU/TLB (matériel) – si ok, on continue ; sinon, *fault*

  $\rightarrow$ interruption noyau (logiciel) – si erreur du programme, `SEGFAUL` ; sinon, traitement (disque, mise à jour tables, etc), et redémarrage instruction

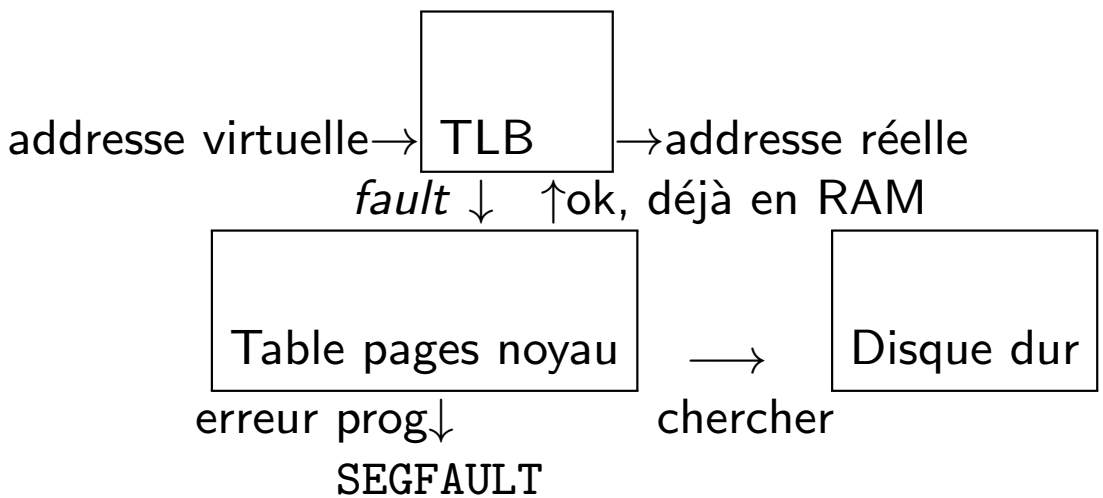


Table des pages

- Une entrée dans la table
  - Adresse physique
  - Drapeaux – Valide/Sale/Accédée
  - Noyau – R/W/X
  - Utilisateur – R/W/X
- Mappage pages physiques, par exemple `mem_map_t` de Linux
  - `counter` — combien d'utilisateurs mappent une page physique
  - `age` — temps t pour des heuristiques de swapping, comme l'algo de Belady
  - `map_nr` — numéro de la page physique
- Zone pour l'allocation et libération des pages physiques

Économie des ressources et augmentation de performance

Gestion mémoire « fainéante »

- Motivation : amélioration performances allocation de mémoire
  - Pagination à la demande – attendre le moment où le processus accède à une adresse de la page avant de l'allouer et de la mapper
  - Permet la surréservation – plus économique
- Motivation : amélioration performances création processus
  - Copy-on-write (copie lors de l'écriture) – lors du clonage d'un processus, ne pas dupliquer sa mémoire tout de suite, mais marquer ces pages comme étant à copier lors du prochain accès en écriture
  - Très important pour Unix :
    - le clonage est la seule manière de création de processus
    - procesuss fils – souvent éphémères – remplacés par l'exécution d'un nouveau programme, avec `exec()`

Caches logiciels

- Cache tampon pour des périphériques de type bloc, cache de pages pour les données des fichiers
- Cache *swap* pour gérér les pages qui sont propres dans le *swap* (sur disque)

Gestion mémoire processus utilisateur

Allocation mémoire

- Apparaît à chaque niveau du système
  - affecte fondamentalement la performance — hautement optimisée
- Liste des zones libres
  - liste chaînée des zones libres de la mémoire libre
  - pointeur adresse zone libre suivante
  - taille zone allouée immédiatement la précédant

Gestion mémoire processus utilisateur

Allocation mémoire

- Buddy System (les « copains »)
  - blocs – divisés e.g. en deux (et chacun encore en deux, etc.) pour trouver la zone nécessaire à l'allocation (pages contigües – amélioration utilisation TLB et RAM)
  - lors d'une libération – fusion uniquement avec l'autre bloc partenaire (le « copain »), s'il est libre
  - efficace, rapide , mais fragmentation interne
  - donc on peut combiner avec free list, slab allocation, etc.

Exemple : A :64K, B :128K, C :64K, D :128K

État						
Libre	1024K					
Allouer A	A	64	128	256		512
Allouer B	A	64	B	256		512
Allouer C	A	C	B	256		512
Allouer D	A	C	B	D	128	512
Libérer C	A	64	B	D	128	512
Libérer A	128		B	D	128	512
Libérer B	256			D	128	512
Libérer D	1024K					

Libération mémoire après appel execve()

```
#include <unistd.h>

int execve(const char *nomFic,
           char *const argv[],
           char *const envp[]);
```

- Rappel arguments : chemin absolu, arguments, environnement (variables shell)
- Rappel : si succès, l'appel ne revient pas
- Écrase tous les segments de mémoire (texte, données initialisées, données non-initialisées et tas, pile) du processus avec ceux du programme qu'on vient de charger
- Garde le PID et le PPID
- Garde les descripteurs de fichiers ouverts, sauf si on a mis `FD_CLOEXEC` avec fcntl()
- Si le fichier avec le programme à charger a un bit `SUID` (ou `SGID`), met le effective `UID` (ou `GID`) du processus au possesseur du fichier
- Renvoie -1 en erreur

Mémoire virtuelle et E/S

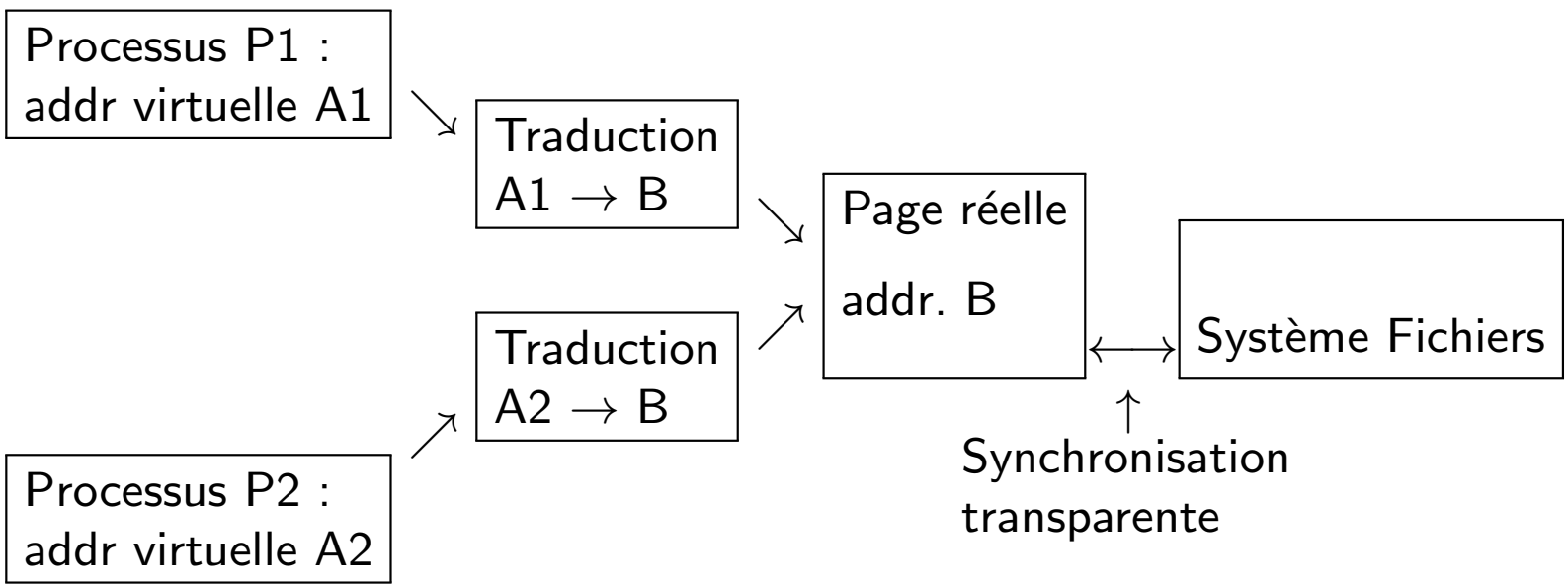
Pages mémoire virtuelle

- Mappage adresses virtuelles – adresses réelles
  - Configuration MMU pour correspondance des pages (*page translation*)
  - Permet le changement dynamique de taille des segments de mémoire virtuelle
  - Offre un méchanisme de protéction des pages mémoire
- Implémente un méchanisme copy-on-write (e.g. pour `fork()`

Mémoire virtuelle, E/S, partage

Entrées/Sorties vers la mémoire

- Mapper les opérations d'Entrée/Sortie vers des simple accès mémoire Load/Store
- Facilite le partage de pages mémoire
  - Utiliser schéma de noms comme pour les fichiers pour identifier des régions de mémoire
  - Même appel système pour implémenter les allocations de mémoire partagée ou bien privée
- Ainsi plusieurs processus peuvent e.g. lire depuis un même fichier en lisant en fait depuis la mémoire.



Appel système mmap()

```
#include <sys/mman.h>

void *mmap(void *adresseStart,  size_t nbrOctets,
           int   prot,          int    flags,
           int   descrFic,      off_t  offset);
```

- Alloue *nbrOctets* octets dans la mémoire virtuelle du processus, commençant à `adresseStart` si elle n'est pas nulle, ou bien n'importe où autrement
- Mappe dans cette région de mémoire une région depuis le fichier `descrFic`, à partir de la position `offset`
- `adresseStart` doit être un multiple de la taille d'une page mémoire (mais d'habitude elle est nulle)
- Valeur de retour
  - Adresse de début de la zone mémoire ainsi allouée si succès
  - `MAP_FAILED` si échec (correspondant à `(void *)-1`)

Appel système mmap()

```
#include <sys/mman.h>

void *mmap(void *adresseStart, size_t nbrOctets,
           int   prot,         int    drapeaux,
           int   descrFic,     off_t  offset);
```

- Paramètre `prot` – la protection – actions possibles avec le contenu
  - `PROT_EXEC` contenu peut être exécute en tant que instructions
  - `PROT_READ` lecture
  - `PROT_WRITE` écriture
  - OU binaire parmi ces trois
  - `PROT_NONE` aucun accès
- Paramètre `drapeaux` – une valeur parmi `MAP_PRIVATE` et `MAP_SHARED`
  - `MAP_PRIVATE` – copy-on-write, donc page privée (écriture dedans n'affectera pas le fichier `descrFic` et ne sera pas visible pour aucun autre processus)
  - `MAP_SHARED` – page partagée – modifications visible pour les autres processus

  mais à laquelle on peut mettre en OU binaire
  - `MAP_ANONYMOUS` – pas de mappage de fichier (donc `descrFic` sera ignoré) ; méchanisme soujacent pour augmenter/diminuer les segments (pile, tas – `malloc()`, etc.)
  - d'autres valeurs – regardez le `man mmap`

Mémoire partagée — Intéraction processus (IPC) – discussion initiale

- Question : Comment faire pour que des processus se mettent d'accord pour partager la même région de mémoire ?
  - Partager — facile : `mmap()` avec `MAP_SHARED`
  - Se mettre d'accord pour la même — difficile
- Solution : utiliser un nom de fichier comme point de rendez-vous
- Inconvénient : gaspillage espace disque
  - Utiliser alors `MAP_ANONYMOUS` en OU binaire
    mais alors on perd l'association mémoire-nom de fichier, qui nous fournissait le méchanisme de rendez-vous
- Comment faire alors ?
- Utilisation d'autres appels système, dédiés – `shmget()`, `shmat()`, etc.

Mémoire partagée — Intéraction processus (IPC) — marche à suivre

- création — `shmget()` — prend une clé IPC et rend un identificateur (∼ descripteur de fichiers) — utilisé pour les opérations
- attachement au processus — `shmat()` — prend un identificateur et rend l'adresse
- besoin de synchroniser l'accès — deux sémaphores
- à la fin — détachement du processus — `shmdt()`
- suppression avec `shmctl()` et `IPC_RMID`

Sémaphores

- **contrôle de l'accès à des ressources partagées**
  - objets ayant une valeur entière
  - deux opérations atomiques
    - $P()$ — réservation — si sémaphore (ressource) dispo (i.e. valeur positive), décrémentation

$$P(s) : [\textbf{tantque}(s == 0)\{\textbf{attendre}\}; s \leftarrow s - 1]$$

    - $V()$ — libération

$$V(s) : [s \leftarrow s + 1]$$

  - l'attente peut se faire passivement : processus endormi, et réveillé quand s n'est plus 0.
- **gérer l'alternance de l'accès à des ressources partagées**
  - plusieurs sémaphores

Sémaphores style System V

- identification — clé IPC (∼ noms des fichiers)
- ensembles de sémaphores obtenus simultanément – `semget()` — prend une clé IPC et rend un identificateur (∼ descripteur de fichiers) — utilisé pour les opérations
- initialisation, manipulation — `semctl()`
- le noyau maintient des infos — les UID du créateur et du possesseur (∼ fichiers)
- $P()$ et $V()$ — `semop()` — conditions à remplir simultanément sur tous les membres
- persistance — même après la fin des processus — besoin de les supprimer explicitement — uniquement par le créateur ou possesseur ou superutilisateur — `semctl()`

Exemple utilisation deux sémaphores – mémoire partagée

- but : père écrira, fils lira, à tours de rôles
- mais système — élection processus non-déterministe
- nécessaire :
  1. exclusion mutuelle (une action à la fois)
  2. ordre – écriture d'abord, lecture ensuite, cycle
- point 1 $\Longrightarrow$ un sémaphore S_1
- point 2 $\Longrightarrow$ un autre sémaphore S_2
- cycle :
  1. au début – zone mémoire vide – $(S_1 = 1, S_2 = 0)$
  2. quand père commence à écrire – $S_1.P() \Longrightarrow (S_1 = 0, S_2 = 0)$
  3. fin écriture, lecture permise – $S_2.V() \Longrightarrow (S_1 = 0, S_2 = 1)$
  4. quand fils commence à lire – $S_2.P() \Longrightarrow (S_1 = 0, S_2 = 0)$
  5. fin lecture, nouvelle écriture permise – $S_1.V() \Longrightarrow (S_1 = 1, S_2 = 0)$

Création sémaphores System V avec clé

Si c'est le premier processsus qui en a besoin
qui crée le sémaphore de clé *Key*

```
1  Si (sémaphore de clé Key n'existe pas) alors
2     demander au S.E. de créer le sémaphore S
3     demander au S.E. de l'initialiser
   sinon
4     récupérer le sémaphore S auprès du S.E.
   fin si
5  P(S)
6   // instructions critiques
   V(S)
```

Création sémaphores System V avec clé – séquences

Si c'est le premier processsus qui en a besoin
qui crée le sémaphore de clé *Key*

```
1  Si (sémaphore de clé Key n'existe pas) alors
2     demander au S.E. de créer le sémaphore S
3     demander au S.E. de l'initialiser
   sinon
4     récupérer le sémaphore S auprès du S.E.
   fin si
5  P(S)
6   // instructions critiques
   V(S)
```

P	P'
1	
2	
3	
5	
	1'
	4'
	5'

OK

Création sémaphores System V avec clé – séquences

Si c'est le premier processsus qui en a besoin
qui crée le sémaphore de clé *Key*

```
1  Si (sémaphore de clé Key n'existe pas) alors
2     demander au S.E. de créer le sémaphore S
3     demander au S.E. de l'initialiser
   sinon
4     récupérer le sémaphore S auprès du S.E.
   fin si
5  P(S)
6   // instructions critiques
   V(S)
```

P	P'
1	
2	
3	
5	
	1'
	4'
	5'

OK

P	P'
1	
2	
3	
	1'
	4'
	5'
5	

OK

Création sémaphores System V avec clé – séquences

Si c'est le premier processsus qui en a besoin
qui crée le sémaphore de clé *Key*

1 Si (sémaphore de clé *Key* n'existe pas) alors
2 demander au S.E. de créer le sémaphore S
3 demander au S.E. de l'initialiser
sinon
4 récupérer le sémaphore S auprès du S.E.
fin si
5 P(S)
6 // instructions critiques
V(S)

P	P'
1	
2	
3	
5	
	1'
	4'
	5'

OK

P	P'
1	
2	
3	
	1'
	4'
	5'
5	

OK

P	P'
1	
2	
3	
	1'
	4'
5	
	5'

OK

Création sémaphores System V avec clé – séquences

Si c'est le premier processsus qui en a besoin
qui crée le sémaphore de clé *Key*

```
1  Si (sémaphore de clé Key n'existe pas) alors
2     demander au S.E. de créer le sémaphore S
3     demander au S.E. de l'initialiser
   sinon
4     récupérer le sémaphore S auprès du S.E.
   fin si
5  P(S)
6   // instructions critiques
   V(S)
```

P	P'
1	
•	1'
•	•
•	•

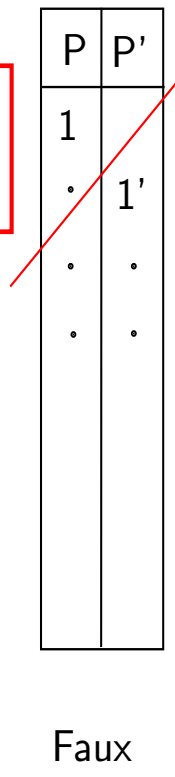
Faux

Création sémaphores System V avec clé – séquences

Si c'est le premier processsus qui en a besoin
qui crée le sémaphore de clé *Key*

doit être atomique

```
1  Si (sémaphore de clé Key n'existe pas) alors
2     demander au S.E. de créer le sémaphore S
3     demander au S.E. de l'initialiser
   sinon
4     récupérer le sémaphore S auprès du S.E.
   fin si
5  P(S)
6   // instructions critiques
   V(S)
```

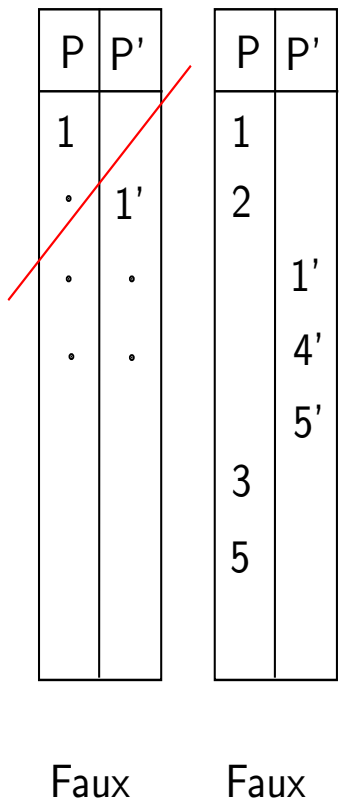


Faux

Création sémaphores System V avec clé – séquences

Si c'est le premier processsus qui en a besoin
qui crée le sémaphore de clé *Key*

1 Si (sémaphore de clé *Key* n'existe pas) alors
2 demander au S.E. de créer le sémaphore S
3 demander au S.E. de l'initialiser
sinon
4 récupérer le sémaphore S auprès du S.E.
fin si
5 P(S)
6 // instructions critiques
V(S)

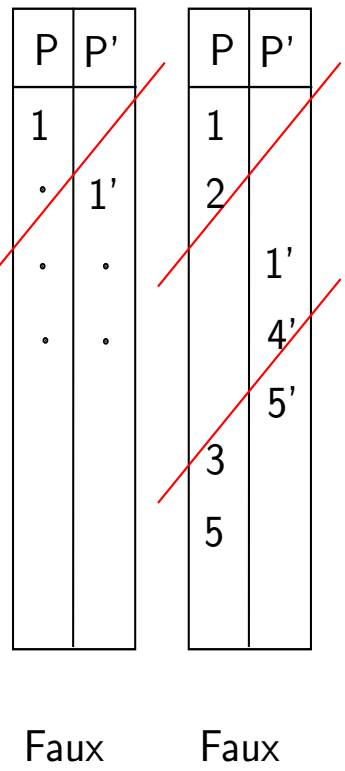


Création sémaphores System V avec clé – séquences

Si c'est le premier processsus qui en a besoin
qui crée le sémaphore de clé *Key*

doit être atomique

```
1  Si (sémaphore de clé Key n'existe pas) alors
2     demander au S.E. de créer le sémaphore S
3     demander au S.E. de l'initialiser
   sinon
4     récupérer le sémaphore S auprès du S.E.
   fin si
5  P(S)
6   // instructions critiques
   V(S)
```



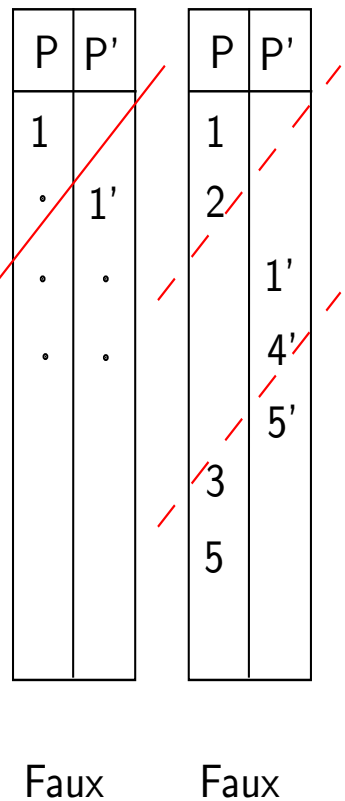
Création sémaphores System V avec clé – séquences

Si c'est le premier processsus qui en a besoin

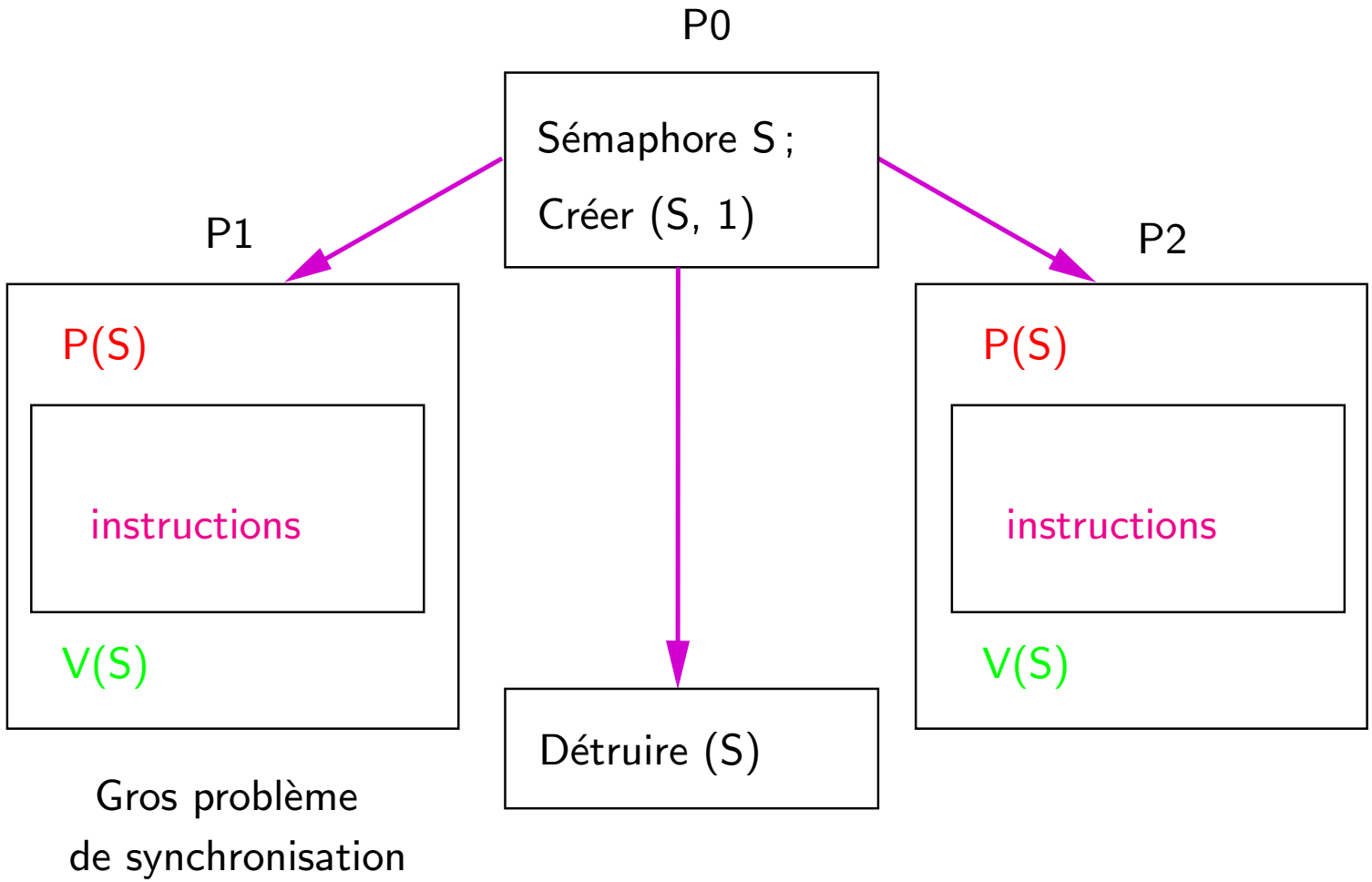
qui crée le sémaphore de clé *Key*

Linux/Unix : ATOMIQUE seulement ceci

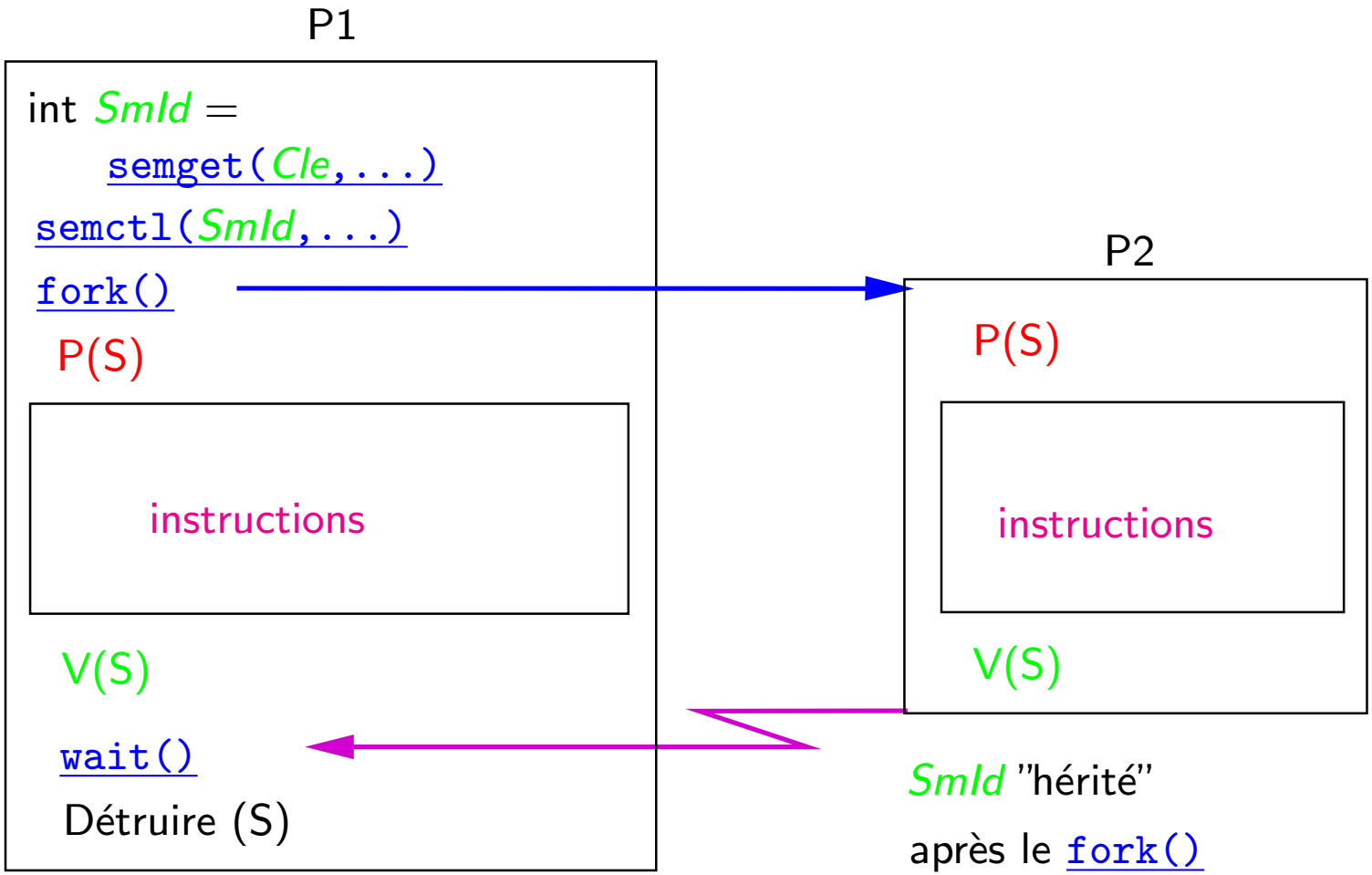
```
1  Si (sémaphore de clé Key n'existe pas) alors
2     demander au S.E. de créer le sémaphore S
3     demander au S.E. de l'initialiser
   sinon
4     récupérer le sémaphore S auprès du S.E.
   fin si
5  P(S)
6   // instructions critiques
   V(S)
```



Sémaphores


P0
Sémaphore S ;
Créer (S, 1)
P1
P(S)
instructions
V(S)
P2
P(S)
instructions
V(S)
Détruire (S)
Gros problème
de synchronisation

Sémaphores System V



Exemple utilisation deux sémaphores – mémoire partagée – suite

- but : père écrira (copiant un fichier dans la zone mémoire), et fils lira la zone mémoire, à tours de rôles
- comment faire précisément ?
- père — boucle (arrêt en fin de fichier) :
  - attend aux sémaphores pour $(1, 0)$, les rend à $(0, 0) \rightarrow S_1.P()$
  - se met à écrire dans le segment, finissant avec '\0'
  - met les sémaphores à $(0, 1) \rightarrow S_2.V()$
- fils — boucle (arrêt sur lecture d'un '.' seul) :
  - attend aux sémaphores pour $(0, 1)$, les rend à $(0, 0) \rightarrow S_2.P()$
  - se met à lire depuis le segment, jusqu'au '\0'
  - met les sémaphores à $(1, 0) \rightarrow S_1.V()$

Schéma du programme – sans tests d'erreur – préparatifs

```
...//include necssrs sys/{types,wait,ipc,...}.h, etc.
int feu = semget(IPC_PRIVATE,1,0700);
semctl(feu,0,SETVAL,1);
semctl(feu,1,SETVAL,0);
int office = shmget(IPC_PRIVATE, maxSize, 0700);
char *guichet = static_cast<char*>(shmat(office, 0, 0));
sembuf vide[2] ={{0,-1,0},{1,0,0}};// attente pr ecrire
sembuf ecrit[2]={{0,0,0},{1,1,0}};//rendre ok pr lire
sembuf plein[2]={{0,0,0},{1,-1,0}};// attente pr lire
sembuf lu[2]   ={{0,1,0},{1,0,0}};//rendre ok pr ecrire
int p = fork();
```

- rappel – deux sémaphores $\Longrightarrow$`sembuf` deux éléments : chacun trois valeurs
  - indice du sémaphore (0 ou 1),
  - opération dessus ($P() \rightarrow -1$, $V() \rightarrow 1$, ou rien $\rightarrow 0$)
  - drapeaux (ici rien – zéro)
- i.e : *vide* $\rightarrow S_1.P()$, *ecrit* $\rightarrow S_2.V()$, *plein* $\rightarrow S_2.P()$, *lu* $\rightarrow S_1.V()$

Schéma du programme – sans tests d'erreur – père

```
if(p) {
  ifstream inputStream ("./toto");
  for(string line; getline(inputStream,line);) {
    semop(feu, vide, 2); // debut zone critique
    const int dernier = (line.size() < maxSize ?
                         line.size() : maxSize - 1);
    for(unsigned int i = 0; i < dernier; ++i)
        memcpy(guichet+i, line . c_str() + i, 1);
    guichet[dernier]='\0';//memcpy lent pour demo
    semop(feu, ecrit, 2); //fin zone critique
  }
  wait(0); semctl(feu,0,IPC_RMID,0);
  shmctl(office, IPC_RMID, 0);
  inputStream.close();
}
```

Schéma du programme – sans tests d'erreur – fils

```
 else {
   for(bool qCont=true;
       qCont; qCont=(guichet[0] != '.')) {
     semop(feu, plein, 2); // debut zone critique
     for(unsigned int i=0;
         guichet[i] && i<maxSize; ++i) {
       cout << guichet[i] << flush;// lente pour demo
     }// on peut inserer des sleep() et
      // enlever les semphrs.
     cout << "\n";
     semop(feu, lu, 2); // fin zone critique
    }
}
```