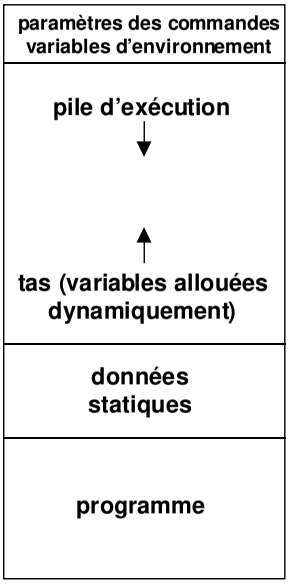
Processus dans Unix

- **Un processus réalise l'exécution d'un programme**
  - commande (du langage de commande)
  - programme d'application
- **Un processus comprend**
  - une mémoire qui lui est propre (mémoire virtuelle)
  - un contexte d'exécution (état instantané)
    - pile (en mémoire)
    - registres du processeur
- **Un processus et identifié par un numéro (*pid*)**
  - La commande *ps* donne la liste des processus en cours d'exécution (voir *man ps*)
  - La fonction *getpid*() indique le numéro du processus qui l'exécute



mémoire d'un processus dans Unix

Vie et mort des processus

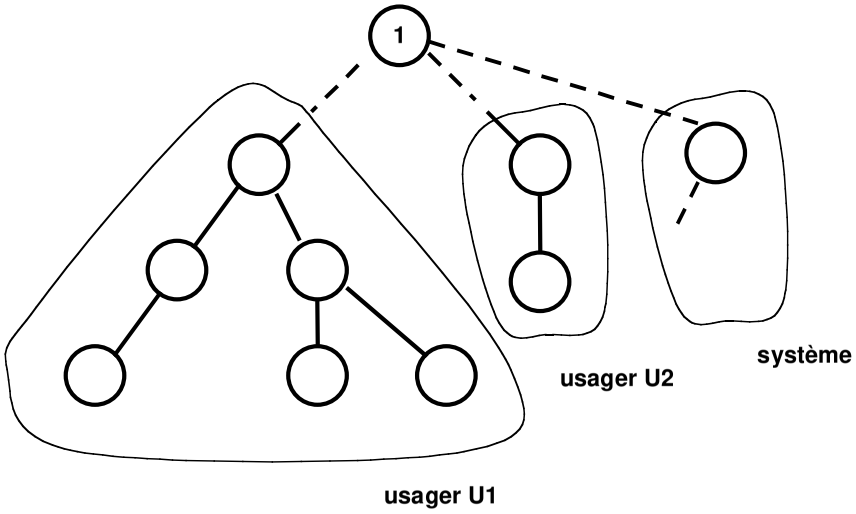
- **Un processus a généralement un début et une fin**
  - Début : création par un autre processus
    - il existe un processus "primitif" créé à l'origine du système
  - Fin
    - auto-destruction (à la fin du programme)
    - dstruction par un autre processus
    - certains processus ne se terminent pas ("démons", réalisant des fonctions du système)
- **Dans Unix**
  - Dans le langage de commande
    - un processus est créé pour l'exécution de chaque commande
    - on peut créér des processus pour exécuter des commandes en (pseudo)-parallèle :
      - `prog1 & prog2 &` créé deux processus pour exécuter `prog1` et `prog2`
  - Au niveau des appels système
    - un processus est créé par une instruction spéciale *fork* (voir plus loin)

Création des processus dans Unix

- L'appel système *fork* permet de créér un processus
- Le processus créé (fils) est un clone (copie conforme) du processus créateur (père)
- Le père et le fils ne se distinguent que par le résultat rendu par *fork*
  - pour le père : le numéro du fils (ou -1 si création impossible)
  - pour le fils : 0

```
if (fork() != 0) {
        /* programme du père */
        ...
} else {
        /* programme du fils */
        /* en général exec (exécution d'un nouveau programme) */
}
```


Hiérarchie de processus dans Unix



- Fonctions utiles
  - `getppid()` : obtenir le numéro du père
  - `getuid()` : obtenir le numéro d'usager (auquel appartient le processus)

Relations entre processus dans Unix (1)

- **Faire attendre un processus**
  - `sleep(n)` : se bloquer pendant `n` secondes
  - `pause`: se bloquerjusqu'à la réception d'un signal par un autre processus
- **Envoyer un signal à un autre processus**
  - Un signal est un événement envoyé à un processus par un autre processus. Vis à vis du processus destinataire, il est asynchrone (non lié aux étapes de son exécution)
  - Il existe un certain nombre de signaux prédéfinis correspondant à différentes actions requises de la part du destinataire
  - `kill (pid, sig)` : envoie le signal `sig` au processus `pid`
    - l'envoi d'un signal cause souvent (pas toujours) la terminaison du processus récepteur (d'où le nom)
    - un signal ne peut en principe être envoyé qu'aux processus du même usager (même *uid*)
    - un processus peut se protéger contre l'envoi d'un signal
    - un processus peut définir une séquence de traitement (traitant, ou *handler*) pour chaque signal qu'il peut recevoi.

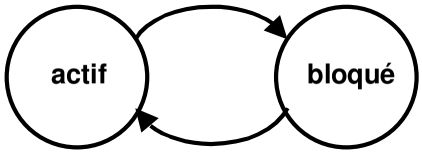
Relations entre processus dans Unix (2)

- **Synchronisation entre un processus père et ses fils**
  - Le fils termine son exécution par `exit(statut)`. `statut` est un code de fin (0 si normal, sinon code indiquant une erreur)
  - Un processus père peut attendre la fin de l'exécution d'un fils par `wait (*stat_loc)`. la variable (facultative) `stat_loc` recueille le statut. `wait` renvoie le *pid* du fils.
  - Exemple : le père attend la fin d'un fils spécifié :

```
while (childpid != wait(&status))
  ;
```

États d'un processus

États logiques

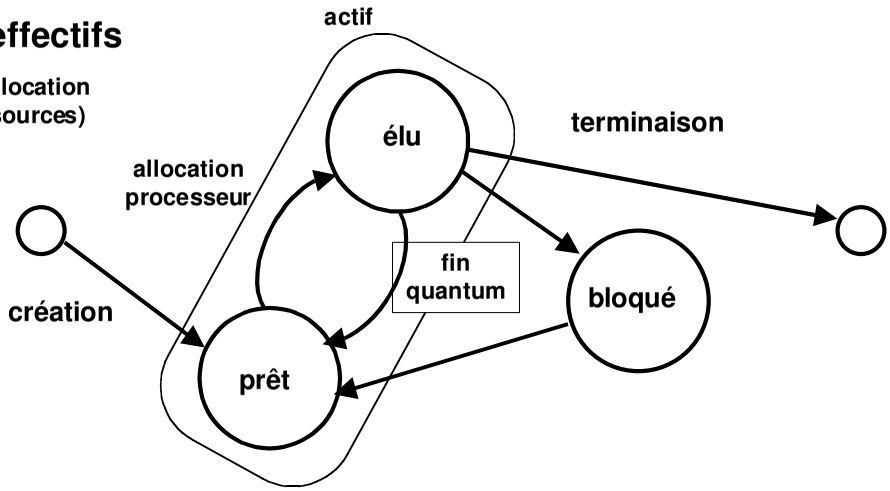


causes de blocage :

entrée-sortie
attente d'un signal
(entre autres wait, pause, sleep,...)

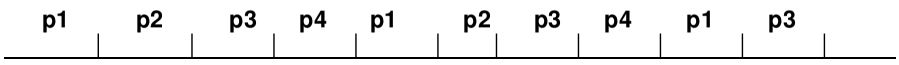
États effectifs

(avec allocation de ressources)



Raisonner sur les processus parallèles (important !)

- Dans un système d'exploitation, le processeur est multiplexé entre les processus prêts (quantum d'exécution)



- Les instants de commutation sont indépendants de la logique interne du déroulement de chaque processus
  - s'il y avait suffisamment de processeurs, il n'y aurait pas de partage (pas de commutation)
- Pour le raisonnement logique sur les processus, il ne faut faire aucune hypothèse sur l'ordre relatif des exécutions (ou, ce qui revient au même, sur les vitesses relatives). Seuls comptent
  - l'ordre d'exécution interne à chaque processus
  - les relations logiques entre les processus (synchronisation)

Le problème de l'exclusion mutuelle

- **Opérations sur un compte bancaire**
  - les processus p1 et p2 sont lancés depuis deux agences différentes

processus p1

```
1. courant = lire_compte (1867A)
2. nouveau = courant + 1000
3. ecrire_compte (1867A, nouveau)
```

processus p2

```
1. courant = lire_compte (1867A)
2. nouveau = courant + 3000
3. ecrire_compte (1867A, nouveau)
```

- **À noter...**
  - les variables `courant` et `nouveau` sont locales à chaque processus (il y en a donc deux exemplaires distincts et indépendants)
  - les deux processus se déroulent en parallèle. L'exécution des opérations peut être entrelacée dans un ordre quelconque, à condition de respecter l'ordre local pour chacun des processus
- **Exemples d'exécution**
  - exécution 1 : p2.1 ; p2.2 ; p2.3 ; p1.1 ; p1.2 ; p1.3
  - exécution 2 : p1.1 ; p1.2 ; p2.1 ; p2.2 ; p2.3 ; p1.3
  - quels sont les réultats ? que peut-on en conclure ?

Sections critiques et actions atomiques

- **Comment éviter les problèmes d'accès concurrent aux variables partagées ?**
- **Assurer que l'ensemble des opérations (consultation + mise à jour) est exécutée de manière indivisible (atomique)**
  - pas d'interférences possibles de la part d'autres opérations exécutées en parallèle

processus p1

A1

```
1. courant = lire_compte (1867A)
2. nouveau = courant + 1000
3. ecrire_compte (1867A, nouveau)
```

processus p2

A2

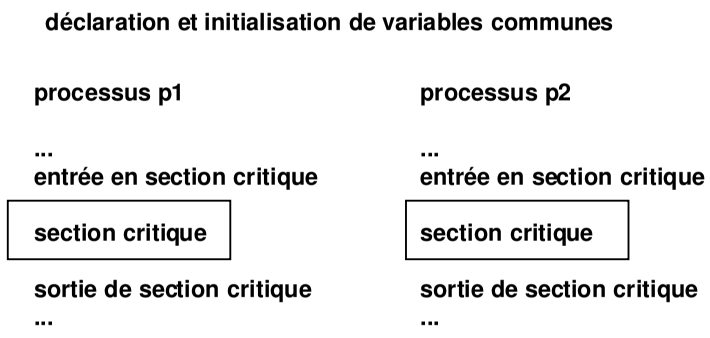
```
1. courant = lire_compte (1867A)
2. nouveau = courant + 3000
3. ecrire_compte (1867A, nouveau)
```

Si A1 et A2 sont atomiques, le résultat de l'exécution parallèle de A1 et A2 ne peut être que celui de A1 ; A2 ou de A2 ; A1, à l'exclusion de tout autre.

- **On dit aussi que la séquence d'actions 1; 2; 3 (dans p1 et dans p2) est une section critique : elle doit être exécutée en exclusion mutuelle (un seul processus au plus peut être dans sa section critique à un instant donné) .**

Réalisation d'une section critique (1)

- **Schéma général**



- **Les opérations "entrée en section critique", "sortie de section critique" doivent garantir l'exclusion mutuelle**

Réalisation d'une section critique (2)

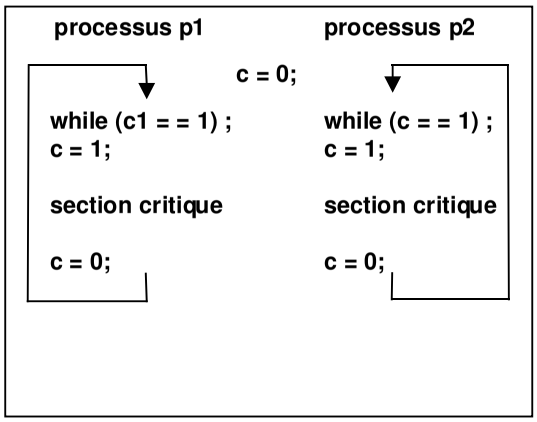
- **Il existe plusieurs modes de réalisation d'une section critique**
  - par attente active
    - très inefficace s'il y a un seul processeur
    - utilisé pour des séquences brèves en multiprocesseur
  - en utilisant des primitives spéciales, (fournies par le système) elles-mêmes atomiques
    - Exemple 1 : comment assurer qu'un seul système Netscape est actif

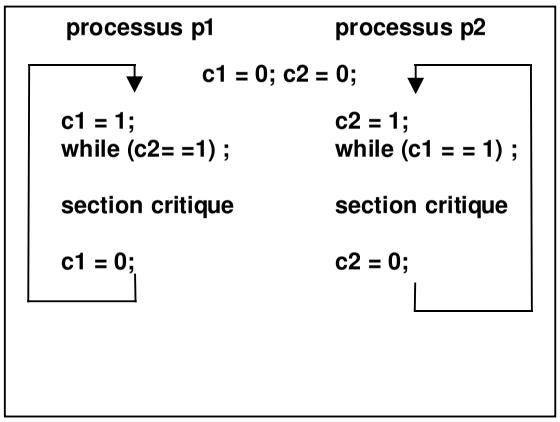
```
/* lancer une session Netscape */
if ((lock_descr = creat("~/.netscape/lock", 0)) == -1) (
        /* afficher message d'erreur */
        ...
} else {
        /* lancer navigateur */
}
...
/* terminer session */
close (lock_descr); unlink ("~/.netscape/lock");
```

    - Exemple 2 : les verrous (détails plus loin)
  - il reste à garantir que les primiives sont elles-même atomiques
    - mécanismes internes au système (masquage des interruptions, Test&Set, etc.)

Réalisation d'une section critique par attente active (1)

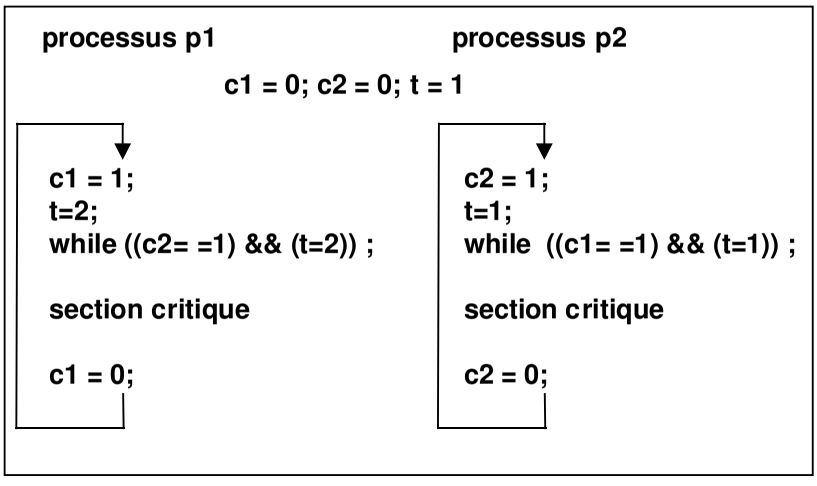
- Réaliser l'exclusion mutuelle par attente active est plus difficile qu'il n'y paraît …
- Exemples de "fausses solutions", pour 2 processus





Réalisation d'une section critique par attente active (2)

- Une solution correcte pour l'exclusion mutuelle par attente active pour 2 processus (Peterson, 1981)



Opérations de verrouillage

- **Les opérations de verrouillage sont une manière de réaliser l'exclusion mutuelle, pour une opération particulière (l'accès à un fichier)**
- **Deux opérations**
  - **v-excl (f) : verrouille le fichier f avec accès exclusif**
  - **dev (f) : déverrouille le fichier f**
  - **Remarque : ces noms sont symboliques, la réalisation en Unix est donnée plus loin**
- **Propriétés garanties**
  - **les opérations v-excl et dev sont atomiques (réalisées par appel système)**
  - **un fichier f verrouillé en accès exclusif par un processus ne peut pas être verrouillé par un autre processus**
  - **un processus qui tente de verrouiller un fichier déjà verrouillé en accès exclusif est bloqué (mis en attente du verrou)**
  - **l'opération de déverrouillage réveille un processus en attente du verrou (et un seul)**

```
...                                          ...
v-excl (f)                                   v-excl (f)
accès au fichier f (section critique)        accès au fichier f (section critique)
dev (f)                                      dev (f)
...                                          ...
```


Résumé de la séance 2

- **Les processus dans Unix** (sera appliqué en TP)
  - **création et destruction**
  - **relations entre processus**
  - **quelques commandes et primitives utiles**
  - **réalisation concrète : états, organisation de la mémoire**
- **Exclusion mutuelle et opérations atomiques**
  - **nécessité de l'exclusion mutuelle**
  - **notions de section critique et d'opération atomique**
  - **premiers exemples de mise en œuvre**