Cours de S.E. – les Processus

A. B. Dragut

Univ. Aix-Marseille II, IUT Aix en Provence

Plan

- Généralités
  - démarrage
  - terminaison
- Mise en œuvre des processus
  - généralités et création de processus
  - signaux
  - identification de processus
  - exemple et discussion
- Création des processus, états
  - partage de l'unité centrale
  - états et relations père-fils
  - race conditions
  - chargement autre programme, paramètres, environnement
- Gestion de la mémoire centrale
  - compilation et édition de liens
  - structure
  - exemple
  - image mémoire
  - tas, pile, mémoire virtuelle
  - lien avec la mémoire physique réelle

Notion de processus

- un processus est un programme en train d'être exécuté par le système.
- les processus ont une structure, un environnement, et au cours de leur existence passent par plusieurs états ; ils ont un identificateur unique : le PID
- les processus coexistent au sein du système, partageant ses ressources.
- on étudie :
  - le démarrage et la terminaison des processus
  - l'environnement et la structure mémoire d'un seul processus
  - les relations entre les processus ainsi que leur gestion

  principalement pour des SE de type Unix.

« Vitrine » des processus – pseudo système de fichiers /proc/<pid>

- visualisation renseignements dynamiques système
- un répertoire dédié par processus :

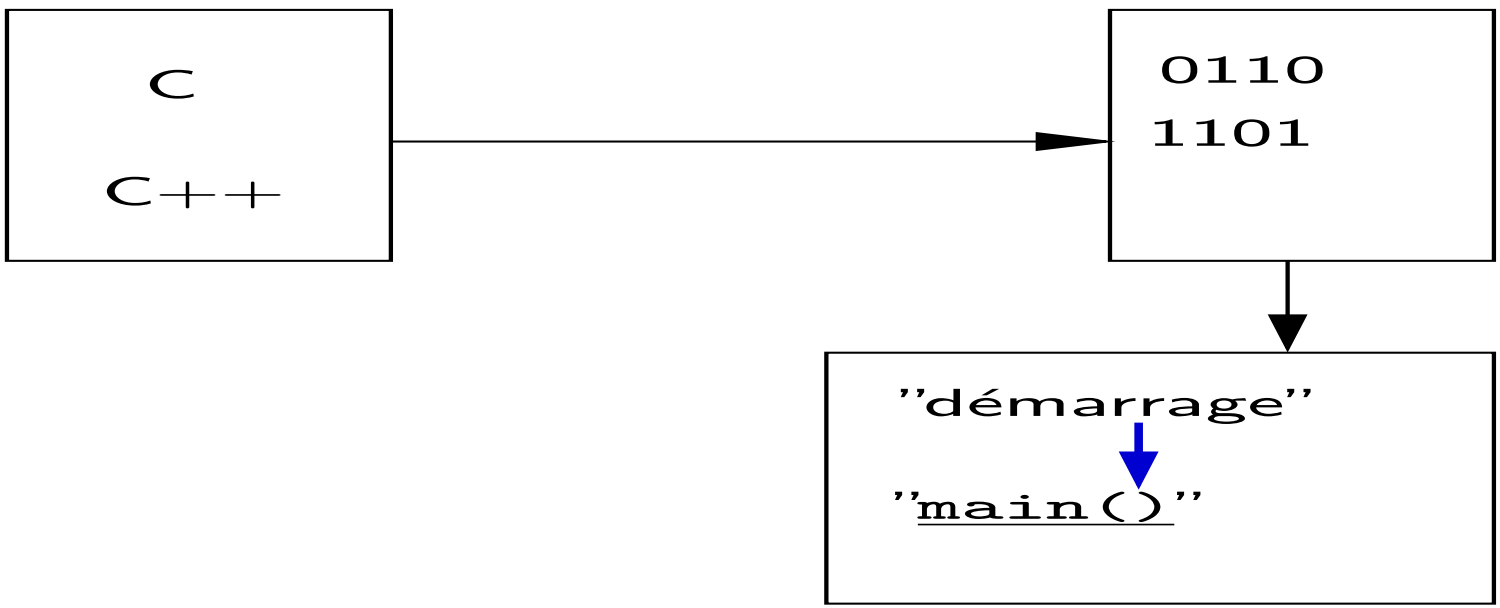
  /proc/<lePidDuProc>

  e.g. pour pid== 1234
  - commande de lancement – **/proc/1234/cmdline**
  - lien vers l'exécutable – **/proc/1234/exe**
  - répertoire courant de travail – **/proc/1234/cwd**
  - sous-répertoire avec les descripteurs de fichiers ouverts –

    /proc/1234/fd/0 −> /dev/pts15
    /proc/1234/fd/1 −> /dev/pts15
    /proc/1234/fd/2 −> /dev/pts15
    /proc/1234/fd/3 −> /tmp/temp.txt
  - renseignements sur la mémoire occupée, et autres statistiques
  - etc.

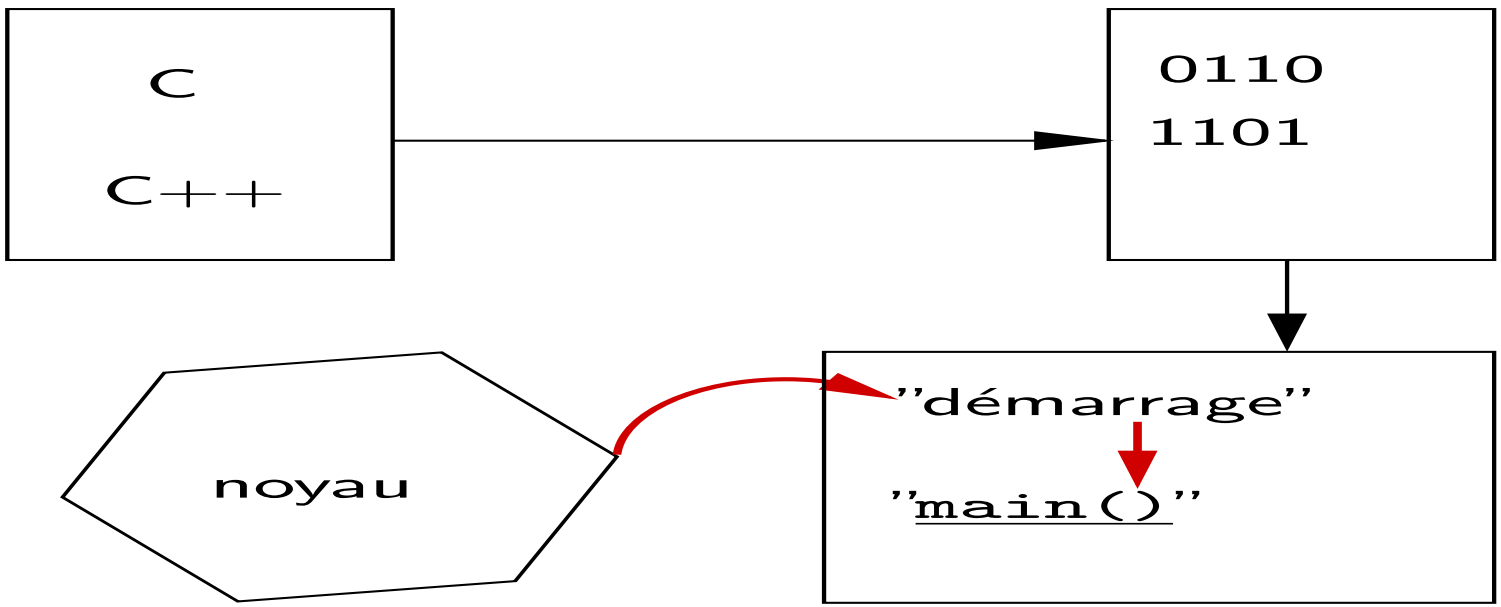
Démarrage d'un processus Unix

- programme écrit en C — commence par la fonction `main()`
- compilation ⟶ code objet relocatable
- édition de liens ⟶ code exécutable
- une procédure de démarrage est également insérée dans le code exécutable par l'éditeur de liens : lors du démarrage du programme par le noyau, celui-ci appelle justement cette procédure de démarrage



Démarrage d'un processus Unix

- rappel : `int main(int argc, char *argv[])` — procédure de démarrage
  - récupère depuis le noyau les arguments de ligne de commande et l'environnement et
  - les met en place pour `main()`
  - appelle `main()`, démarrant ainsi effectivement le programme
- c'est bien entendu le noyau qui démarre le programme (au moyen d'une des fonctions `exec()`



Terminaison d'un processus Unix

Cinq manières :

- Trois normales (volontaires) :
  - retour de la fonction `main()`
  - appel de la fonction `exit()`
  - appel de la fonction `_exit()`
- Deux anormales :
  - appel de la fonction `abort()`
  - réception d'un signal

Fonctions exit() et _exit()

```
#include <stdlib.h>
void exit(int Status);
```

```
#include <unistd.h>
void _exit(int Status);
```

- La fonction `_exit()`, spécifiée par POSIX, revient tout de suite au noyau.
- La fonction `exit()`, spécifiée par ANSI C, fait d'abord un peu le ménage (fermeture des streams de la bibliothèque C I/O standard, etc.) et appelle ensuite `_exit()`.
- La procédure de démarrage (qui avait appelé `main()` au début) fait également un appel à `exit()` si `main()` a fait un retour.

Remarque

La valeur de retour (exit code) est définie soit avec l'appel d'une de ces fonctions, soit avec un return(Status) du `main()`.

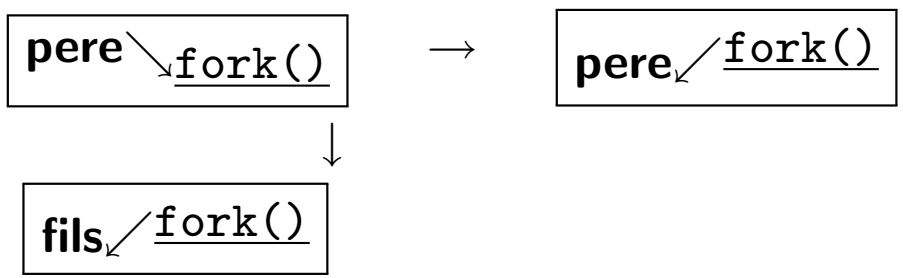
SE

- chaque processus est identifié par son PID, de type `pid_t`
- le noyau maintient beaucoup de renseignements sur chaque processus
- certains de ces renseignements sont dans l'espace mémoire du système (et ne swappent pas)
- d'autres sont dans l'espace du processus
- il existe enfin un pseudo-SGF nommé proc, permettant l'accès à ces renseignements : proc, monté d'habitude dans /proc est ainsi une interface vers les structures de données du noyau concernant chaque processus (on en a vu l'usage pour voir les adresses des segments en mémoire vive)
- on a vu que l'appel système `exec()` permet le chargement en mémoire d'un programme
- en fait, cet appel système agit sur un processus déjà existant, l'écrasant complètement
- il faut donc pouvoir créer de nouveaux processus tout frais
- ceci se fait avec l'appel système `fork()`.

fork()

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork();
```

- cet appel système, depuis un processus nommé père,
  - fait apparaître un nouveau processus dans le système, nommé fils
  - revient (return) deux fois : une fois dans le père, une fois dans le fils
  - peut échouer, par exemple
    - si la table de processus est pleine
    - si le quota de processus pour cet utilisateur a été atteint
- le nouveau processus est pratiquement une copie reproduisant la plupart des caractéristiques du père



`fork()`

- Valeur de retour : le nouveau PID est alloué au fils, et renvoyé par `fork()` uniquement dans le père
- le fils se voit alloué comme PPID le PID du père (le noyau assigne à tous les processus un PID et également le PID de leur père)
- la table de correspondance entre les pages virtuelles et les pages physiques est entièrement dupliquée : la mémoire virtuelle du fils est structurée de manière identique à celle du père ;
- en fait dans la table des pages du nouveau processus, les pages physiques RW sont marquées copy-on-write, i.e. dès qu'on essaiera d'écrire dedans, les pages seront réellement dupliquées (autrement non), la table mise à jour, et après on écrira.
- l'état du processeur est également sauvegardé, car au moment du `fork()`, le père et le fils en sont au même point

`fork()` — dédoublement

- retour de `fork()` : deux processus identiques (au PID et quelques autres aspects près)
- l'exécution continue dans chacun indépendamment de l'autre, à partir du point de retour de `fork()`

```
#include <unistd.h>
#include <iostream>
int main() {
  int a = 0; // un seul processus, une variable
  fork(); //au retour il y a deux processus identiques
  a = 1;      // chacun met 1 dans _sa_ variable a
  std :: cout << a; // chacun l'affiche: on voit 11
  return 0; // chacun finit
}
```

`fork()` — dédoublement

- la seule différence « visible » au moment du retour de `fork()` : la valeur de retour, 0 dans le fils, et le PID du fils dans le père
- les PIDs sont positifs $\Longrightarrow$ différencier le code du père du code du fils

```
pid_t pidFils;
if((pidFils = fork())) { // alors on est dans le pere
...
}
else { // sinon on est dans le fils
...
}
```

Remarque

Attention aux endroits d'où on appelle `fork()` *: le moins d'expressions entourantes possible, pas dans des <<, etc. On ne contrôle pas l'ordre des opérations autour de l'appel, et on ne sait alors pas quand on dédouble effectivement le processus.*

Utilisations principales de `fork()`

- pour qu'un processus se dédouble afin que plusieurs sections de son programme soient exécutées « en même temps ». ceci arrive souvent pour ce qu'on appelle serveurs :
  - le parent attend une demande de service,
  - à réception de la demande, il se dédouble, et
  - laisse le fils traiter la demande, tandis que
  - le parent revient dans l'attente d'une nouvelle demande
- pour qu'un processus exécute un programme (totalement) différent. ceci est, par exemple, le travail des *shell*s :
  - on écrit une *commande* dans le shell
  - le shell récupère toute la ligne, l'analyse et la transforme selon ses règles
  - se dédouble et
  - fait `exec()` dans le fils avec le programme demandé dans la ligne de commande

Lors du dédoublement ? — les fichiers

- la table des descripteurs de fichiers : c'est comme si `dup()` avait été appelé pour chaque descripteur ouvert du processus père
- mais la table des fichiers ouverts est partagée par les deux processus (copy-on-write)
- $\Longrightarrow$ tant qu'aucun des deux (père ou fils) ne fait d'`open()`, les entrées de la table des descripteurs de fichiers pointent vraiment vers les mêmes entrées de la table des fichiers ouverts
- ! ! père et fils, font avancer le même pointeur de fichier, donc peuvent tranquillement écrire l'un à la suite de l'autre, sans écraser mutuellement ce qu'ils écrivent (ex. : s'ils écrivent sur la sortie standard, laquelle est redirigée dans un fichier)
- par contre, ce que lit l'un l'autre ne lit plus, il lit à la suite
- il est en général recommandable soit
  - de synchroniser le père et le fils, s'ils utilisent les mêmes descripteurs
  - de les faire utiliser des descripteurs différents (chacun fermant ceux inutilisés)

Rappel :

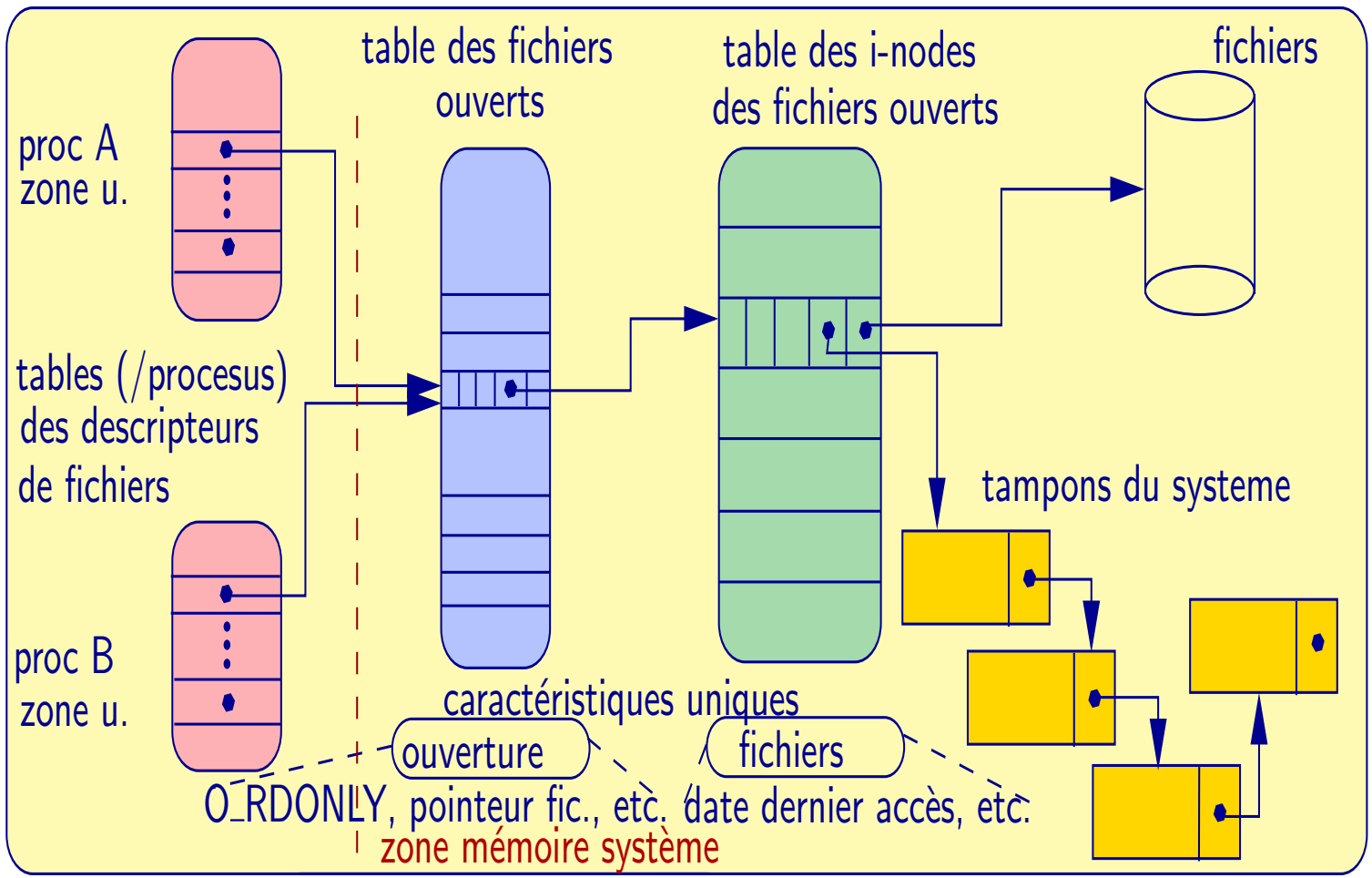


table des fichiers ouverts
table des i-nodes des fichiers ouverts
fichiers
proc A zone u.
tables (/procesus) des descripteurs de fichiers
proc B zone u.
tampons du systeme
caractéristiques uniques
ouverture
fichiers
O_RDONLY, pointeur fic., etc.
date dernier accès, etc.
zone mémoire système

Lors du dédoublement ? — les signaux, les ressources

- le masque des signaux et leurs dispositions sont également dupliqués,
- mais les signaux pendants ne le restent que pour le père ; le fils n'en a plus.

- le fils hérite des limites de ressources du père
- il pourra les changer, mais uniquement en les diminuant davantage
- par contre, les compteurs (temps, nombre de défauts de page, etc.) sont bien entendu réinitialisés

getpid() et getppid()

« qui suis-je ? », « d'où viens-je ? », fork() ne peut pas répondre au fils

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);  // PID du processus appelant
pid_t getppid(void); // PID du pere
```

chaque processus a également un user ID, un group ID, un effective user ID, → getuid(), etc.

Exemple

```
#include <sys/types.h>
#include <unistd.h>
#include <iostream>
int main() {
  if(pid_t pidFils=fork()){//alors on est dans le pere
    std :: cout << "Pere " << ::getpid()
                << " et fils " << pidFils << "\n";
  }  else { // sinon on est dans le fils
      std :: cout << "Fils " << ::getpid()
                  <<" de pere "<<::getppid()<< "\n";
  } // atention on ne sait pas qui passe en premier
  return 0;
}   // la logique doit etre independante de cet ordre
```

```
Fils 9557 de pere 9556
Pere 9556 et fils 9557
```

ou bien

```
Pere 9556 et fils 9557
Fils 9557 de pere 9556
```

Discussion et questions

- sous Unix (et famille), la seule manière normale de créer des processus est avec `fork()`. Durant l'amorçage du système (boot), le noyau crée le processus *init*, qui est le père de tous les processus et a le PID *1*.
- Le PID *0* désigne les pseudo-processus du noyau.
- tous les processsus ont un vrai père (sauf *init*, qui est de PPID 0).
- un processus peut changer son UID comme il veut, s'il est super-utilisateur — ainsi on finit par faire le login et avoir un shell en son nom (le tout partant de *init* de UID root)
- la commande *ps* permet de voir les processus.
- Questions
  - les étapes — états durant la vie d'un processus ?
  - que se passe-t-il quand un deux, père ou fils, termine ? que fait l'autre ?
  - que se passe-t-il quand on se dédouble avec `fork()` dans une boucle ?

Contexte d'un processus

- plusieurs processus « indépendants », chacun le CPU « pour soi »
- comment gérer ?
- notion fondamentale – contexte d'un processus
  - l'´**tat du CPU** qui l'exécute :
    - registres accessibles au programme
    - registres de contrôle (e.g. l'instruction courante $\longrightarrow$ compteur ordinal)
    - mémoire accessible (registre-segment, dispositif de pagination)
  - **programme** exécuté par le processus
  - **données** en mémoire sur lesquelles il agit
  - **ressources** physiques et logiques que le SE lui attache
  - données **administratives**
    - identification de l'utilisateur
    - consommation de ressources

Commutation de contexte

- système multiprocessus – le CPU exécute tour-à-tour des processus
- commutation de contexte – transition de l'attribution du CPU d'un processus à l'autre
- étapes de la commutation de contexte :
  - suspension de l'exécution du processus P
  - sauvegarde contexte P
  - nouveau processus Q – rétablissement contexte Q (ayant été sauvegardé auparavant)
  - mise en exécution Q
- opérations nécessaires à la commutation de contexte :
  - lectures/écritures sur disque (si swap)
  - mise-à-jour structures noyau
- coûteux – plus intéressant – threads

Ordonnancement – mission et ingrédients

- besoin de décider du « nouveau processus » – élection
- notion de temps
  - temps général du système – périphérique **horloge temps réel**
  - temps « vu par un processus »– succession des tranches de temps (jiffy) – CPU $\longrightarrow$ processus – 100ms
- étapes du déroulement (cycle) :
  - CPU en train d'exécuter un processus P
  - horloge – interruption du CPU (ou bien P libère le CPU)
  - comptabilisation temps CPU utilisé par P
  - sauvegarde contexte P
  - ordonnanceur –
    - « rangement » de P (dans ses structures – « file »)
    - élection du nouveau processus Q (recevant le jiffy suivant)
  - restauration contexte Q
  - CPU travaille maintenant sur Q
  - horloge – nouvelle interruption du CPU (ou bien Q libère le CPU)
  - comptabilisation temps CPU utilisé par Q
  - etc.

Libération du CPU par un processus

- fin du processus
- attente entrée-sortie
- réception certains signaux (e.g. suspension)
- appel fonction spéciale `sched_yield()` par processus lui-même :

```
#include <sched.h>
int sched_yield(void);
```

  - libération CPU (sans bloquer)
  - « rangement » par l'ordonnanceur pour attendre un nouveau tour
  - élection autre processus qui prendra le CPU –
  - si pas de tel autre processus – continuation du même
  - renvoie 0 si succès, -1 sinon (avec errno).

Ressources consommées par un processus

- temps, mémoire, nombres de pages, etc
- structure spécifique

```
struct rusage {
    struct timeval ru_utime; /* user time used */
    struct timeval ru_stime; /* system time used */
    long   ru_maxrss;        /* maximum resident set size */
    long   ru_ixrss;         /* integral shared memory size */
    long   ru_idrss;         /* integral unshared data size */
    long   ru_isrss;         /* integral unshared stack size */
    long   ru_minflt;        /* page reclaims */
    long   ru_majflt;        /* page faults */
    long   ru_nswap;         /* swaps */
    long   ru_inblock;       /* block input operations */
    long   ru_oublock;       /* block output operations */
    long   ru_msgsnd;        /* messages sent */
    long   ru_msgrcv;        /* messages received */
    long   ru_nsignals;      /* signals received */
    long   ru_nvcsw;         /* voluntary context switches */
    long   ru_nivcsw;        /* involuntary context switches */
};
```

Ordonnancement – critères

- L'ordonnanceur décide qui prend la main (le CPU).
- But à long terme – nombre de processus en mémoire
- But à **cours terme** – élection du processus suivant à prendre le CPU
  1. Minimiser le temps d'exécution des processus
  2. Minimiser la variance du temps moyen d'exécution
  3. Maximiser le nombre de processus complétés (throughput)
     - minimiser le coût de gestion (commutation contexte, etc)
     - optimiser l'usage des ressources système
  4. Assurer l'équitabilité du partage du CPU — compromis avec 1

Ordonnancement – techniques

- FIFO — premier venu, premier servi
- Tourniquet (round robin) – chacun un petit bout – queue FIFO **circulaire** avec interruption (i.e. quantum de temps alloué)
- Shortest Job First (SJF) – le processus avec le moins de travail restant – impossible de prédire
- Shortest Remaining Time First (SRTF) — SJF préemptif
- Multilevel Feedback Queue – approximation de SRTF

Ordonnancement par Multilevel Feedback Queue

- le passé « prédit » l'avenir : si processus fait beaucoup d'E/S, « alors » continuera ainsi
- favoriser les processus ayant été les moins gourmands en CPU
- **comportement adaptatif**
- Problème de SRTF : processus de calcul – possible ne plus avoir de CPU
- Multilevel Feedback Queue
  - plusieurs queues, de priorités différents,
  - chacune – tourniquet, tant qu'elle ne se vide pas
  - une fois vide, on passe à la suivante (moins prioritaire)
  - chacune – son quantum de temps, augmentant exponentiellement d'une queue à la suivante moins prioritaire
  - les priorités des processus (i.e. queue dans laquelle ils sont placés) changent dans le temps — moins fréquemment que 1/quantum.
  - les processus avec beaucoup E/S « flottent », les gourmands « tombent au fond ».

États d'un processus

- états Unix (et famille), visibles avec la commande *ps*
  - R — Running ou Runnable — en cours d'exécution ou prêt
  - T — sTopped — arrêté (par un signal, ou parce qu'il est en train d'être suivi « à la trace »— being traced — `ptrace()` — observation et/ou déboguage).
  - S — interruptible Sleep — en attente pour un événement — `sleep()`, `pause()`
  - D — uninterruptible sleep — si on l'interrompait, de graves problèmes de cohérence des données pourraient s'en suivre, donc le noyau ne le permet pas — d'habitude attente d'entrées-sorties
  - Z — defunct, ou Zombie — fils terminé mais dont la fin n'a pas encore été acquiescée par le père

R→`SIGSTOP`→T→`SIGCONT`→R ; S→`SIGSTOP`→T→`SIGCONT`→S ;
R→`pause()`→S→`SIG`nal dérouté→R,...

- le diagramme d'états complet d'un SE comprend aussi les interruptions pour le partage équitable du temps, les défauts de page, l'envoi sur le swap, etc.

États et relations père-fils

- si le père termine avant le fils, celui-ci devient automatiquement le fils de *init*
- si le fils termine avant le père — alors ce dernier doit prendre connaissance de son sort (ou bien avoir annoncé qu'il n'en était pas intéressé). tant que cela n'est pas fait, le fils est zombie :
  - toutes ses ressources sont libérées par le noyau,
  - seulement son status (code de sortie (si défini) plus no. du signal éventuel l'ayant tué) est gardé par le noyau,
  - il reste une entrée dans la table des processus, le marquant en état Z
- Trop de zombies $\longrightarrow$ des entrées occupées dans la table des processus, et bloquage.

États et relations père-fils – zombies

- libérer les entrées de la table des processus : dès que le père récupère le status des fils
- le père récupère le status du fils → appel système de la famille `wait()`.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *statusFils);
```

- attente changement processus d'un fils quelconque
- récuperation renseignements dans *statusFils*
- renvoie le pid du fils si succès, -1 sinon, avec errno :
  - `ECHILD` – pas de fils à attendre,
  - `EINTR` – interruption par un signal

Attente plus précise

- contrôle plus fin – spécifier le fils à attendre

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pidFils, int *statusFils,
              int   optionsWait);
```

- spécification du fils à attendre
  - le fils de pid $==$ *pidFils*, si *pidFils* > 0,
  - un fils quelconque, si *pidFils*$== -1$ (comme `wait()`),
  - le fils de meme GID, si *pidFils*$== 0$,
  - le fils de GID $== |$*pidFils*$|$ si *pidFils*< -1
- récupération renseignements dans *statusFils*
- si *optionsWait* $==$ `WNOHANG`, alors pas d'attente (i.e. retour immédiat), et renvoie 0 si aucun des fils spécifiés (ou le fils spécifié) n'a/n'ont changé d'état
- renvoie le pid si succès, -1 sinon (avec errno)

Status du fils

- analyse du status récupére à l'aide de macros :
- `WIFEXITED(`*statusFils*`)` – vrai si le fils a terminé suite à un appel à `exit()` ou à `_exit()` ou si return du `main()`, et alors on peut apprendre le code de sortie avec la macro `WEXITSTATUS(`*statusFils*`)`
- `WIFSIGNALED(`*statusFils*`)` – vrai si un signal a terminé le fils, et alors on peut savoir lequel avec la macro `WTERMSIG(`*statusFils*`)`
- `WIFSTOPPED(`*statusFils*`)` – vrai si un signal a arrêté le processus, et alors on peut savoir lequel avec la macro `WSTOPSIG(`*statusFils*`)`
- etc.

États et relations père-fils – zombies

- le status d'un zombie doit être récupéré par son père (celui qui s'est dupliqué, ou l'adopteur *init*, qui prend toujours soin de récupérer le status)
  - l'entrée du fils de la table des processus est supprimée et
  - le fils disparaît enfin totalement du système.
- « En même temps » qu'il attend ses fils, le père fait aussi d'autres choses, mais il ne dispose que d'un seul flux de contrôle du CPU (laissons les multi-threads pour le moment de côté)
- comment ? le noyau envoie le signal `SIGCHLD` au père, ignoré par défaut.

États et relations père-fils — signalisation

- le signal `SIGCHLD` est envoyé au père lorsque
  - le fils termine
  - le fils est stoppé
- par défaut, ce signal est donc ignoré, et le père doit se souvenir tout seul d'appeler une des fonctions de la famille `wait()` pour attendre et récupérer le status du fils
- le père peut dérouter ce signal vers un traitant (qui fait éventuellement ce travail de récupération du status)

États et relations père-fils — signalisation

- Problème avec plusieurs fils : les signaux ne s'empilent pas, donc si n fils terminent « en même temps », il n'y a pas de garantie de recevoir n signaux `SIGCHLD`, et alors il y a des fils non-`wait()`és.
- Comment faire une boucle tant qu'il y a eu un fils terminé – récupérer, avec `waitpid()` et `WNOHANG` (pour ne pas rester bloqué dans le traitant de `SIGCHLD`) ?
- sauvegarde et restauration d'errno, pour ne pas interférer avec le reste du code (le `SIGCHLD` peut s'être glissé juste après un appel système et avant qu'on ait le temps d'examiner errno)

États et relations père-fils — signalisation et zombies

- Père désintéressé, mais pas de zombies créés.
- Comment annoncer le noyau que le père n'est pas intéressé par le status du fils, à l'aide d'un drapeau dans l'action de `sigaction()` : la valeur `SA_NOCLDWAIT` mise dans le champ .sa_flags (OU binaire avec d'autres valeurs désirées) lors d'une configuration pour le signal `SIGCHLD`.
- alors il n'y a plus de zombies de créés,
- mais même dans ce cas-ci, le signal `SIGCHLD` est toujours envoyé.
- De plus, `wait()` bloque jusqu'à la fin de tous les fils, et sort en erreur `ECHILD`.

Remarque

On a toutes ces possibilités car on peut ne pas s'intéresser au status des fils, mais vouloir savoir toutefois au moins qu'ils ont fini, sinon quand « exactement » ils ont fini.

États et relations père-fils — signalisation et `wait()`

- rappel : l'arrivée d'un signal non-ignoré interromp certains appels système
- et la famille `wait()` en fait partie
  - on ne sait pas quand et si le fils va finir,
  - vrai pour `waitpid()` également, si appelée sans `WNOHANG`.
- signal différent de `SIGCHLD` $\Longrightarrow$ interruption comme moyen de sortir de l'attente
- si signal `SIGCHLD` $\Longrightarrow$ pas d'interruption, pour le père pour faire et finir son `wait()` en bonnes conditions,
- problème s'il n'y a pas de `wait()` dans le traitant du signal `SIGCHLD`

États et relations père-fils — signalisation et `wait`

- plusieurs cas : le signal `SIGCHLD`
  - est configuré pour l'action par défaut (`SIG_DFL`) : `wait()` n'est en fait pas interrompue par l'arrivée de `SIGCHLD`
  - est dérouté vers un traitant particulier :
    - si `SIGCHLD` est bien légitime, annonçant un changement de status d'un fils (et envoyé par le noyau), alors `wait()` est interrompue quand même,
    - Linux redémarre automatiquement le `wait()`, lequel sort avec le PID du fils, donc pas d'erreur d'interruption
    - sur d'autres systèmes il faut mettre le drapeau `SA_RESTART` dans le champ .sa_flags, lors du déroutement de `SIGCHLD` avec `sigaction()`.
    - sinon, i.e. si jamais le `SIGCHLD` a été envoyé sans raison (i.e. par un processus qui fait le malin, et non pas par le noyau), alors `wait()` sort en erreur `EINTR`.

Remarque

Attention, en général, `SIG_IGN` *n'est pas la même chose que* `SIG_DFL` *pour* `SIGCHLD`*.*

Plusieurs `fork()` et race conditions

Plusieurs `fork()`

- le tout dépend de l'endroit :
  - si seulement le père fait *n* dédoublements, alors il y aura autant de fils et pas d'autres.
  - si les fils se mettent aussi à faire des `fork()`, alors il faut faire très attention dans quelles conditions — nombre exponentiel

Race conditions

- des conflits ou des interblocages peuvent survenir si dans la logique du programme on s'attend à ce que un des deux (père ou fils) passe avant l'autre
- il faut toujours concevoir les programmes pour qu'ils fonctionnent correctement quel que soit
  - l'ordre de reprise de la main : père d'abord ou fils d'abord
  - le nombre d'instructions de l'un s'exécutant entre deux instructions quelconques de l'autre

Démons

- utilité : services essentiels et « continus » exemples :
  - `syslogd` – logs pour tout programme (console, fichier)
  - `cron` – lancement de tâches régulier préprogramme (*crontab*)
  - `inetd` – gestions requêtes connexions réseau
- d'habitude
  - restent en vie $\longrightarrow$ autant que le noyau,
  - relisent fichier configuration $\Longleftarrow$ signaux
- fils de *init*
- pas de terminal controllant
- Répertoire courant de travail — */*, */var/spool/...*, etc.

Création de démons

- rendre fils de init (e.g. appel `fork()`, père `exit`)
- déconexion de toute console (session propre – `setsid()`)
- changement répertoire courant – `chdir("/")` – car le répertoire de lancement $\longrightarrow$ pourrait être démonté dans le futur
- reconfig masque création fichiers – `umask(0)`
- fermeture descripteurs fichiers non désirés, etc.
- fonction `daemon()` fournie $\longrightarrow$ initialisation correcte

Fonction daemon()

```
#include <unistd.h>

int daemon(int nochdir, int noclose);
// detachement du terminal controllant
// execution dans le background
// nochdir == 0 <=> changement rep courant /
// noclose == 0 <=> redir std{in,out,err} /dev/null
```

- fait un fork() – si réussite, père – _exit(0) ;
- fils – 0 si succès, sinon -1 et mêmes erreurs que fork() et que setsid().

Fonction `chroot()`

- en général – *cd /../* – « sur place »(i.e. toujours au /)
- protection – garantir qu'un processus (et sa descendence) ne sortira pas d'une sous-arborescence désirée – le / à voir par ce processus est à un endroit donné autre que le vrai / (hérité par `fork()`)
- exemple – CGI des sites web, daemons http, programmes moins sûrs, etc.

```
#include <unistd.h>
int chroot(const char *Chemin);
```

- renvoie 0 si succès, -1 sinon, (avec errno), mais ATTENTION
  - ne change pas le répertoire courant $\Rightarrow$ appeler `chdir()` également
  - seulement un processus privilégié peut appeler `chroot()`
  - fermer tous les descripteurs ouverts ailleurs
  - faire précharger libraries, etc. si possible, avant
  - root peut toujours sortir d'une « prison `chroot()` »

Famille des fonctions exec()

```
#include <unistd.h>
int execve(const char *nomFic, char *const argv [],
           char *const envp[]); // appel systeme
```

- si *nomFic* contient un '/', alors il est utilisé comme un chemin. sinon, la variable d'environnment `PATH` est utilisée pour chercher un exécutable *nomFic*
  - soit binaire, (compilé et linkédité)
  - soit texte à interpréter, commençant par `#! cheminInterpreteur [arg]`
- les *argv* et *envp* sont passés au programme (si par exemple C ou C++, la fontion `main()` les récupère). ils finissent obligatoirement par un pointeur nul
- en C la convention est que *argv[0]* est le nom du programme. C'est au programme contenant l'appel à `execve()` de le préparer explicitement.
- *envp* est l'environnement : des chaînes de la forme <NOMVARIABLE>=<valeur>, comme par exemple `TERM=xterm`.

Famille des fonctions `exec()`

- si succès, alors
  - la fonction `execve()` ne revient pas
  - l'image du processus est complètement remplacée par l'image du nouveau processus à démarrer depuis le programme *nomFic*
- sinon, différentes erreurs la font renvoyer -1 et positionner errno
  - pas de fichier trouvé
  - pas de permissions d'accès sur le chemin
  - pas de permission d'exécuter le fichier
  - trop de liens symboliques (d'habitude une boucle)
  - le nom du fichier est trop long
  - il y a trop d'arguments
  - le système de fichiers est monté en `noexec` (par exemple la ligne correspondante du /etc/fstab contient ce drapeau)
  - etc.

Exemple

```
#include <unistd.h>
#include <string.h>
#include <iostream>
int main() {
  char *a[]={"txX","1","2",0};char *e[]={"BLAH=bli",0};
  if(-1 == execve("txX",a,e)){ std::cout<<"Oops\n"; }
  return 1;
} // execute avec le dir de txX dans le PATH
```

```
#include <iostream>// fichier txX.cxx-->executable txX
int main(int argc, char *argv[], char *envp[]) {
  if(argc < 3) { std::cout<<"deux␣args\n";return 1; }
  std::cout << "recu␣"<<argv[1]<<"␣"<<argv[2]<< "\n";
  if(envp && envp[0]) std::cout << envp[0] << "\n";
  return 0;
} // donc dans txX.cxx compile et linkedite en txX
```

Famille des fonctions exec()

```
#include <unistd.h>
extern char **environ; // transmet l'environnement
//ce ne sont que des front-ends pour execve
//les deux suivantes, sans PATH (nom et chemin complet)
int execl(const char *cheminFic, const char *arg, ...);
int execle(const char *cheminFic, const char *arg, ...,
           char * const envp[]);//pas besoin de environ
// et de plus, la liste des arg finit avec 0
int execlp(const char *nomFic, const char *arg, ...);
// idem pour arg, mais avec recherche dans PATH

int execv(const char *cheminFic, char *const argv[]);
int execvp(const char *nomFic, char *const argv[]);
// comme execve, sans l'environnement
```

Discussion `fork()` et `exec()`

Programme
instruction
instruction
`fork()`
instruction
instruction
instruction
instruction
instruction
instruction
instruction
instruction
...

↙ Duplication processus ⇒

`fork()` → 0, FILS (copie du père ! !)
instruction
instruction
`exec(`*file*`,`*arguments*`)`
⟹ *CHARGEMENT NOUVELLE IMAGE*
`main()` du nouveau programme
nouvelle instruction
nouvelle instruction
...
...
fin nouveau programme

Discussion `fork()` et `exec()`

- Que fait `execv(`*file*`,`*argv*`)` ?
  - remplace programme en cours d'exécution $\longrightarrow$ programme *file*
- Échec possible $\Longleftarrow$ *argv* ?
  - Oui : *argv* mauvais pointeur, ou plus de « `MAXARGV` »args (constante système)
  - Non $\Longleftarrow$ contenu pointé aberrant : $\longrightarrow$ programme *file* analyse
- Héritage ?
  - signaux : `SIG_DFL` $\longrightarrow$ ok
  - signaux :`SIG_IGN` $\longrightarrow$ ok (Solaris, fonct. syst.)
  - signaux : traitant particulier $\longrightarrow$ `SIG_DFL`
  - descripteurs fichiers : oui, sauf mention (flag `FD_CLOEXEC`)

Étapes système `exec()`

- Recherche fichier $\longrightarrow$ chemin $\Longrightarrow$ erreurs accès possibles
- Confirmation exécutable (permissions), bon format (ELF, interpréteur etc.)
- Chargement code, données
- Mise à jour table des pages $\Longrightarrow$ échec possible (plus de mémoire noyau...)
- Zone user (u-dot) $\longrightarrow$ modifications valeurs
  - signaux (en attente, pointeurs traitants $\longrightarrow$ effacés, etc.)
  - descripteurs fichiers – fermés si mention `FD_CLOEXEC`

Comparaison `fork()` et `exec()`

`fork()`	`exec()`
modifie PID	ne modifie pas le PID
modifie PPID	ne modifie pas le PPID
garde « tout »en état	change « tout »
garde descripteurs fichiers	garde descripteurs fichiers sauf mention
revient DEUX fois, vals $\neq$	ne revient pas si succès
duplique un proc existant	charge un nouveau programme
augmente le nombre de proc du sys	n'augmente pas le nombre de procs du sys

Compilation et édition de liens

- la compilation $\Longrightarrow$
  - une suite d'instructions machine — directement compréhensibles par le CPU.
  - Les structures de type if - else ou while sont reproduites avec des sauts conditionnels.
  - le CPU exécute normalement les instructions une à la suite de l'autre,
  - les suit à l'aide du registre compteur ordinal (Program Counter),
  - les branchements modifient sa valeur.
  - n'est pas suffisante pour intégrer les appels à des fonctions de bibliothèques, déjà écrites et compilées.
- l'édition de liens combine plusieurs fichiers résultats de la compilation $\longrightarrow$ un fichier prêt à être chargé en mémoire vive et exécuté par le CPU

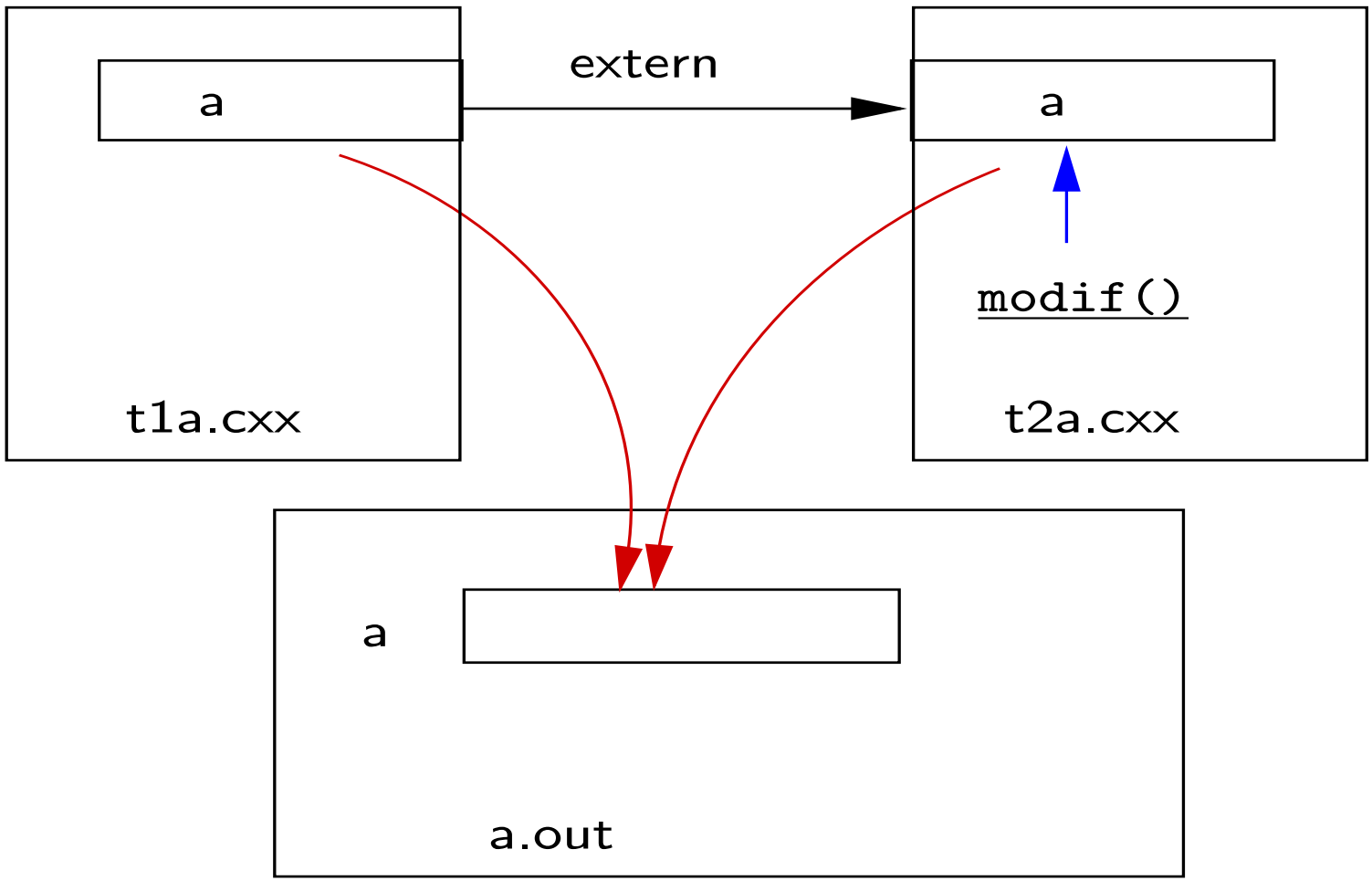
Exemple — rappel *static* et *extern*

```
#include <iostream>
int a = 0;    //definie dans ce fichier, visible partout
void modif();//declaration; sa definition est ailleurs
int main() {
  std::cout << a << "␣"; modif();
  std::cout << a << std::endl; return 0;
} // fichier t1a.cxx
```

```
extern int a;//i.e. vient de l'exterieur de ce fichier
void modif() { // definition de la fonction ici
  a = 1;
} // fichier t2a.cxx, sans main()
```

- compilation *g++ -c t1a.cxx ; g++ -c t2a.cxx*
- édition de liens *g++ t1a.o t2a.o*
- exécution de *a.out* → affiche 0 1.

Exemple — utilisation d'*extern*



Exemple — rappel *static* et *extern*

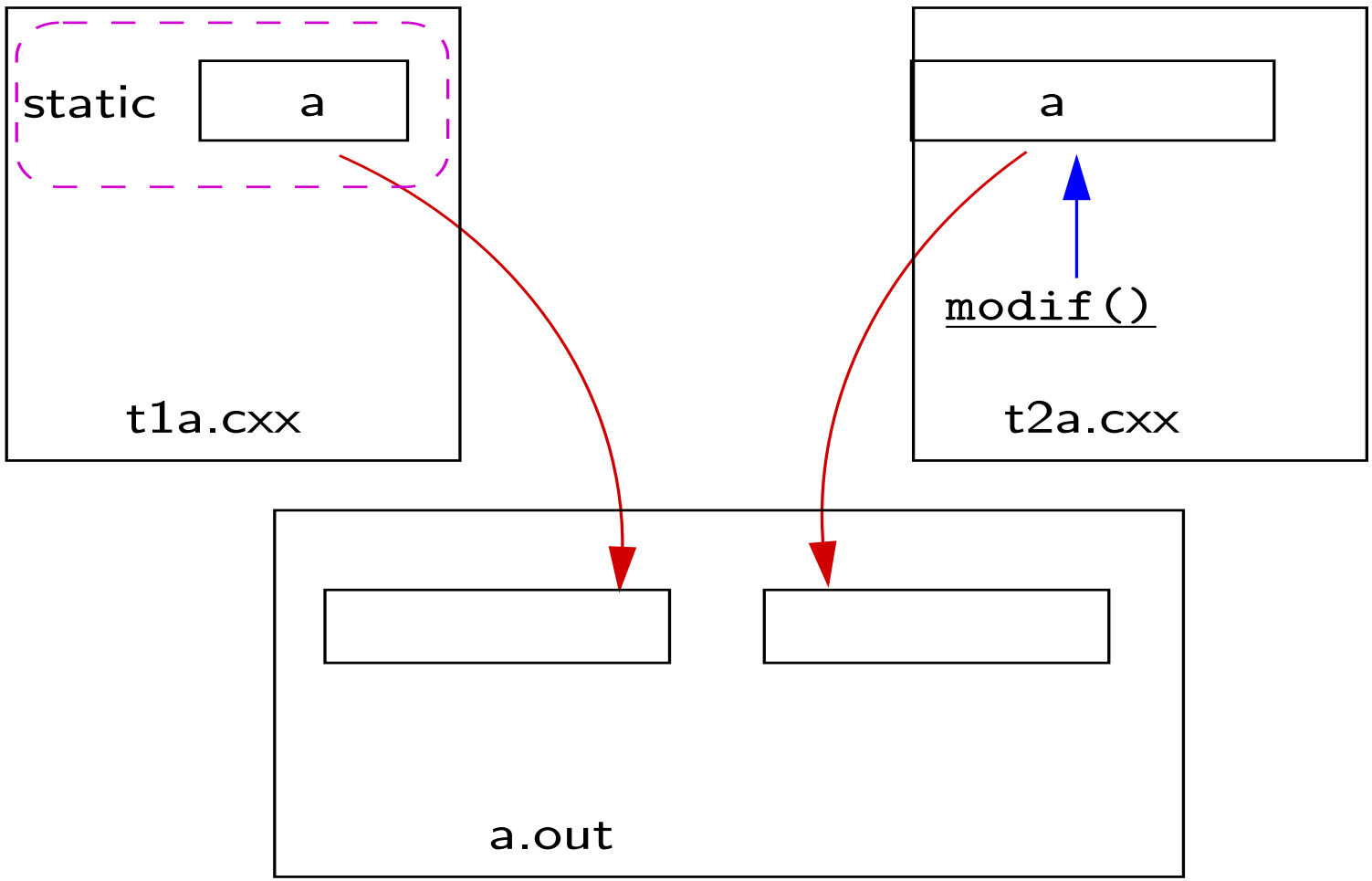
```
#include <iostream>
static int a = 0;// visible uniquement dans ce fichier
void modif();//declaration; sa definition est ailleurs
int main() {
  std::cout << a << "␣"; modif();
  std::cout << a << std::endl; return 0;
} // fichier t1a.cxx
```

```
int a;//visible partout (sauf la ou masquee par static)
void modif() {
  a = 1;
} // fichier t2a.cxx, sans main()
```

- compilation *g++ -c t1a.cxx ; g++ -c t2a.cxx*
- édition de liens *g++ t1a.o t2a.o*
- exécution de *a.out* → affiche 0 0 → deux variables distinctes, avec le même nom.

Exemple — utilisation de *static*

- deux variables distinctes, avec le même nom.



Exemple — rappel *static* et *extern*

- les deux autres combinaisons provoquent des erreurs d'éditions de liens :
  - *static* et *extern* en même temps dans les deux fichiers :

    référence de *a* dans t2a.cxx non-définie

    car l'*extern* dans `t2a.cxx` dit que la définition est ailleurs, or elle est nulle-part, puisque le *static* de `t1a.cxx` la rend invisible dans d'autres fichiers.
  - aucun des deux, dans aucun des fichiers :

    définition multiple de *a*,

    puisque lorsqu'ils sont mis ensemble, les deux `.o` ont chacun une définition de *a* globale, visible partout, ce qui est ambigu, donc interdit.
- dans la table de symboles (crée par le compilateur pour l'éditeur de liens), il y a un attribut qui dit si le symbole est local au fichier objet, ou bien global.

Structure mémoire

- après compilation
  - fichiers objet relocatables — contenant des instructions machine et des données, mais pas prêts pour être exécutés

```
debut:
   ADD   $r0, $r1, $r2
   STORE $r0, valInt
...
   BNZ debut
...
```

    il doivent subir l'édition de liens → les références symboliques deviennent références avec adresses mémoires.
  - par exemple, ici `valInt` et `debut` – symboles.
  - éditions de liens – valeurs absolues (adresses concrètes)
  - idem pour les fichiers objet relocatable partagé, lesquels peuvent également être linkés au moment de l'exécution (dynamiquement – **shared libraries**).

```
allegro % ldd `which bash`
        linux-gate.so.1 =>  (0xffffe000)
        libtermcap.so.2 => /lib/libtermcap.so.2 (0xb7f29000)
        libdl.so.2 => /lib/libdl.so.2 (0xb7f25000)
        libc.so.6 => /lib/tls/libc.so.6 (0xb7df7000)
        ...
```

Structure mémoire – suite

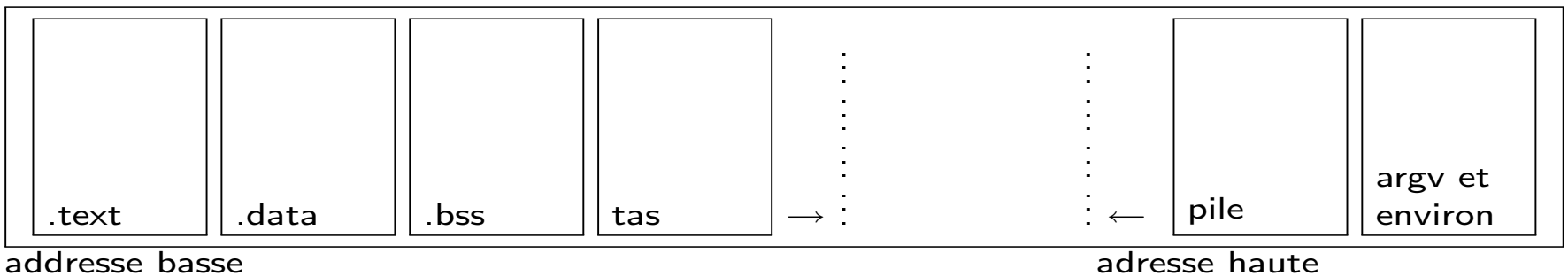
- après l'éditions de liens
  - fichiers exécutables — pouvant être chargés en mémoire par le système et lancés (on en obtient alors une image mémoire processus).
- le stockage en mémoire de masse – respecter différentes normes (propriétaires ou publiques), $\longrightarrow$ la structure exacte de ces fichiers et la manière de les charger effectivement en mémoire vive.
- la gestion des relocatables partagés est légèrement plus compliquée — ne sera pas couverte ici.

Structure mémoire – suite

- Le stockage en mémoire vive des exécutables n'est pas une copie exacte du fichier stocké en mémoire de masse
  - est usuellement composée de segments.
  - un segment $\longrightarrow$ des emplacements mémoire d'adresses consécutives.
  - les segments sont placés en mémoire à différents endroits (espacés)
  - les adresses mémoire utilisées $\longrightarrow$ virtuelles : du point de vue du processus elles ont une valeur bien définie (pour les branchements, où le stockage de données, etc.), dans son espace-mémoire, mais elles sont traduites en adresses physiques réelles par la Memory Management Unit (MMU) — composante matérielle (pour aller le plus vite) — selon un algorithme complexe.
  - référencer, au cours de l'exécution, une adresse hors de tout segment, $\Longrightarrow$ le signal `SIGSEGV`.

Image mémoire

- noyau charge segments → renseignements des tables d'en-tête de programmes (une par segment) → dans fichier exécutable
- le noyau met en place la pile (stack) et le tas (heap)
- Arrangement typique pour l'utilisation de la mémoire : la pile croît du haut vers le bas (les appels de fonctions et leurs variables locales), et le tas croît du bas vers le haut (l'allocation dynamique).



Remarque

Avec ce mécanisme, le processus a l'impression d'être tout seul en mémoire, qu'il a toute rien que pour lui. Cette illusion → mémoire virtuelle.

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 l'allocation mémoire ?

- l'appel système `brk()` qui augmente ou diminue tout simplement la taille du segment de données
- chaque bibliothèque a son allocateur de mémoire.
  - en C on `malloc()` (et `calloc()` et `realloc()`) et `free()`
  - En C++ : `new`, `delete` (et `delete []`).
  - maintient des listes chaînées lui permettant de noter les zones occupées et les zones libres
  - gère l'espace imparti par `brk()` — essayant d'optimiser différents critères (e.g. la minimisation de la fragmentation)
  - $\exists$ allocateurs de mémoire privés (et en C++ les templates des conteneurs standard sont prévus pour en fournir un).

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) ⟵ 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
7ffffffe7000-7fffffffff000 rw-   --- pile         -- read, write
```

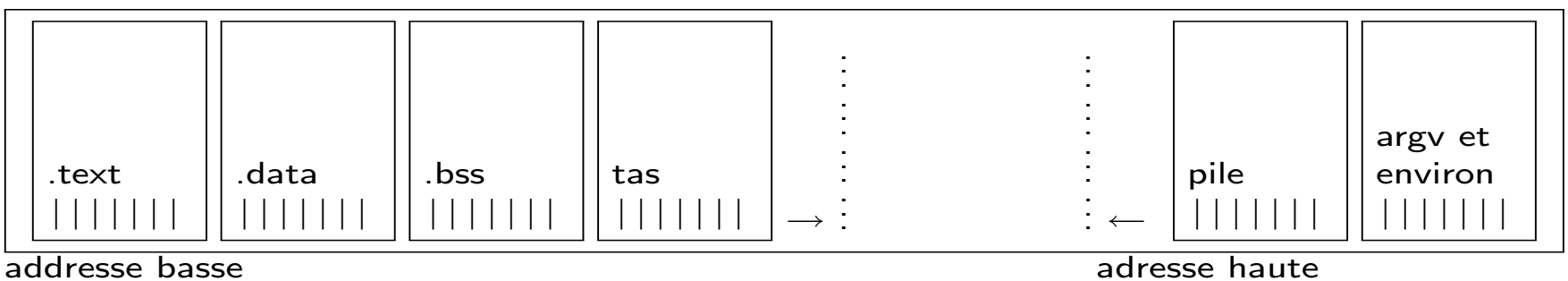
l'appel brk() rendant la fin de la nouvelle zone — le *0x522* – addresses virtuelles.

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.

Mémoire virtuelle

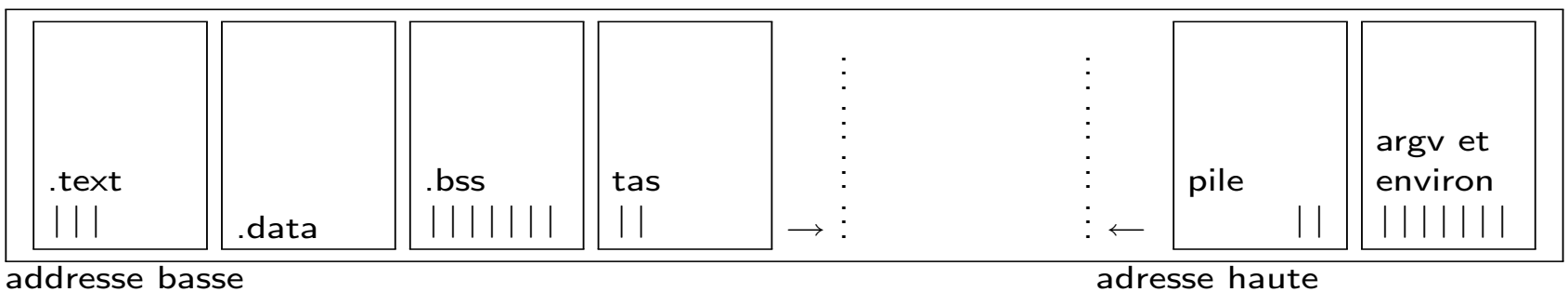
- Faire coexister plusieurs processus en mémoire :
- borne sup de la mémoire telle qu'elle est vue par un processus (mémoire virtuelle) $\longrightarrow$ bus d'adresses du processeur et le système.
- la mémoire physique vive réelle : un espace linéaire, divisé en unités d'allocation nommées pages, de petite taille (par exemple 4K)
- la mémoire virtuelle est elle aussi divisée en pages de la même taille
- chaque segment du programme se voit allouer un nombre de pages virtuelles (donc la taille du segment est un multiple de celle de la page).



- une correspondance entre les pages virtuelles et les pages réelles en mémoire vive est établie au fur et à mesure, et peut changer au cours du temps. gérée par le système.

Mémoire virtuelle, mémoire vive réelle, mémoire de masse

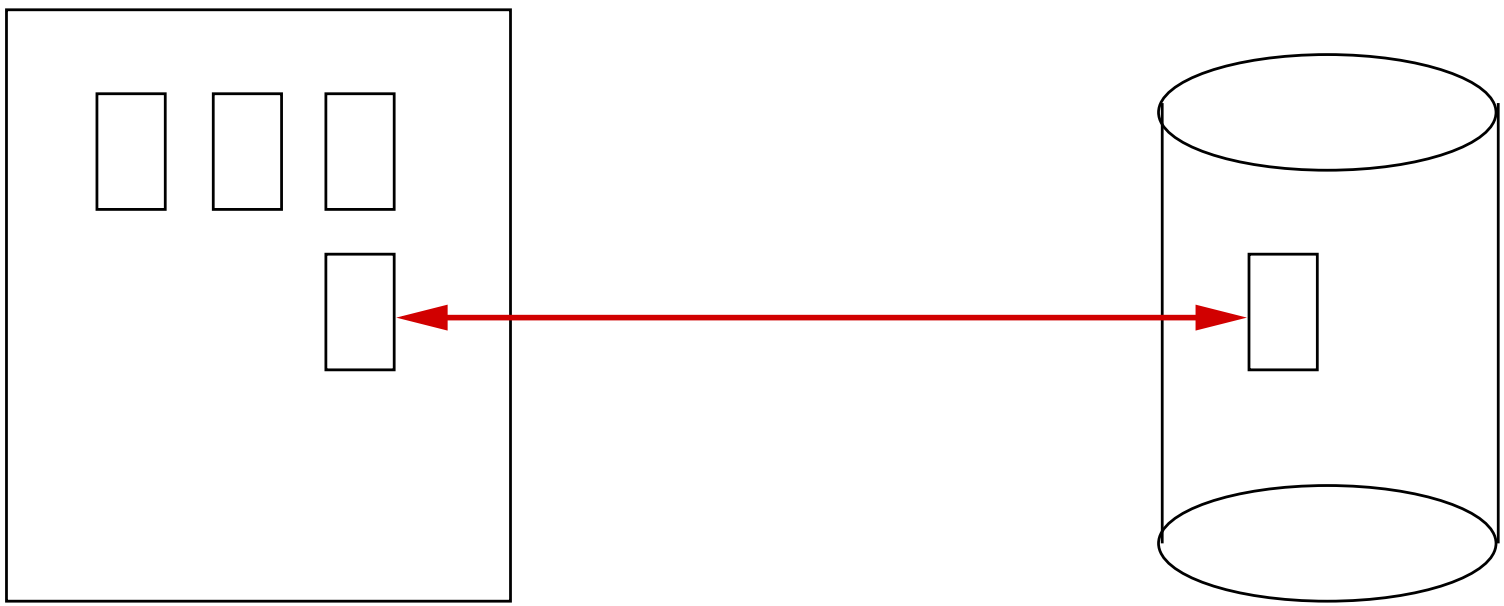
- Un processus n'a pas besoin que tous ses segments soient dans la mémoire vive – principe de localité – on travaille de proche en proche
- le noyau charge en mémoire vive seulement une partie des pages – dès qu'elles sont référencées (i.e. on utilise une adresse tombant dedans)
- au départ par exemple seulement la .bss et une partie du .text, et la pile et le tas.



- la MMU est informée de ces correspondances, et fait la traduction à chaque pas pour le CPU, tant qu'on reste dans les pages présentes en mémoire vive.

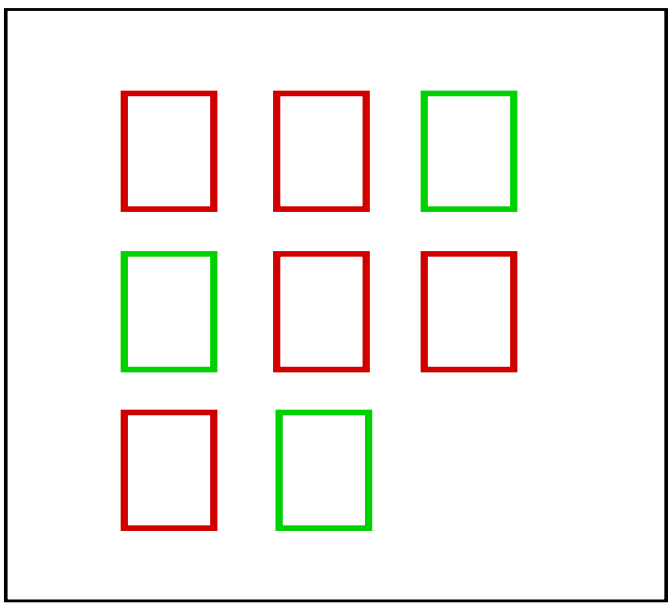
Défauts de page

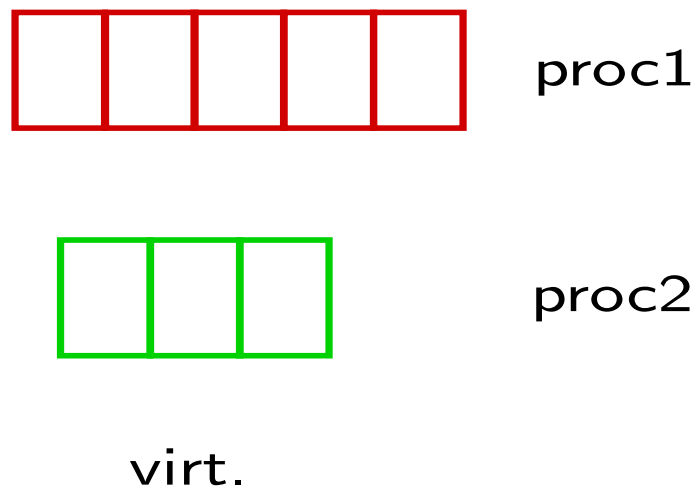
- lorsque le CPU référence une page absente de la mémoire vive, il y a défaut de page ; la MMU fait que le CPU fasse une exception — appelle le noyau pour le traiteur de défauts de page.
  - si c'est la pile ou le tas, le traiteur du noyau alloue simplement encore une page
  - sinon, le traiteur du noyau trouve la page sur disque (dans le swap), et tente de la charger en mémoire vive
  - s'il n'y a plus de place, alors le noyau doit d'abord en faire (envoi sur disque d'une autre page à garder, ou écrasement)



Entrelacement

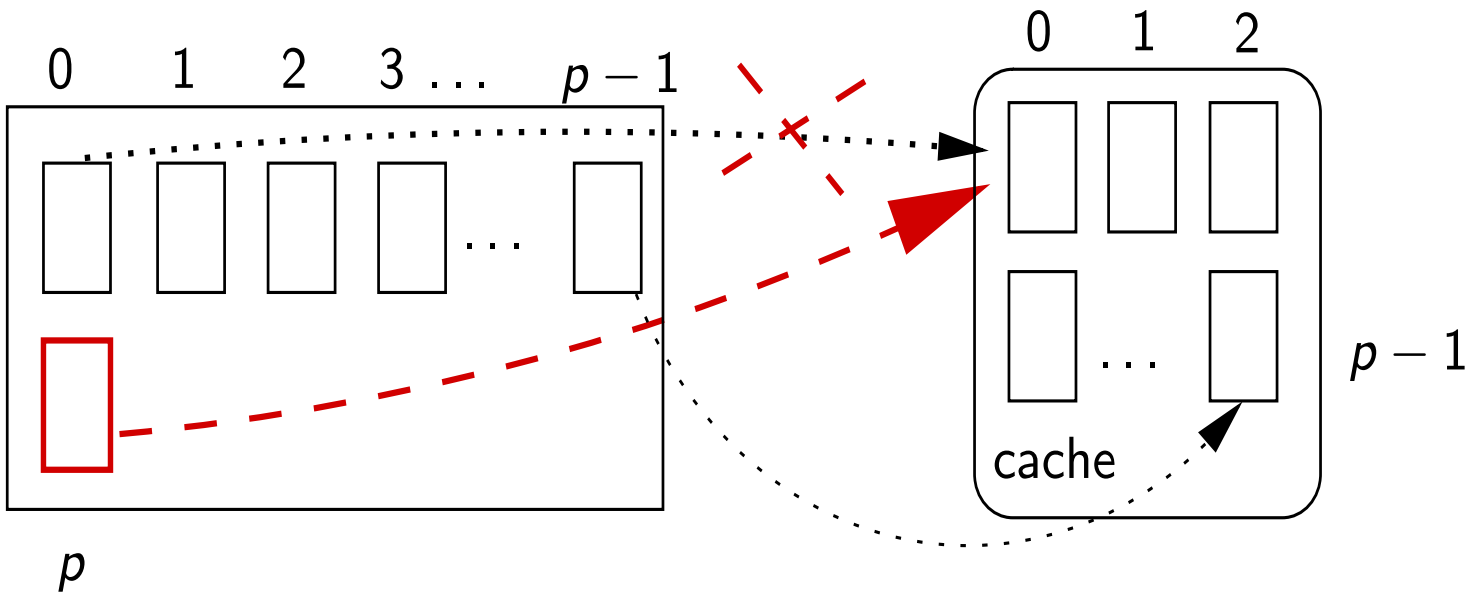
- Les pages sont entrelacées entre les processus, mais seulement le noyau est au courant.
- rappel :
  - pour le processus, les pages virtuelles d'un segment sont contigües
  - ce sont les adresses réelles mémoire vive de ces pages qui peuvent être n'importe où il y a eu de la place au moment de leur chargement





Quelques remarques

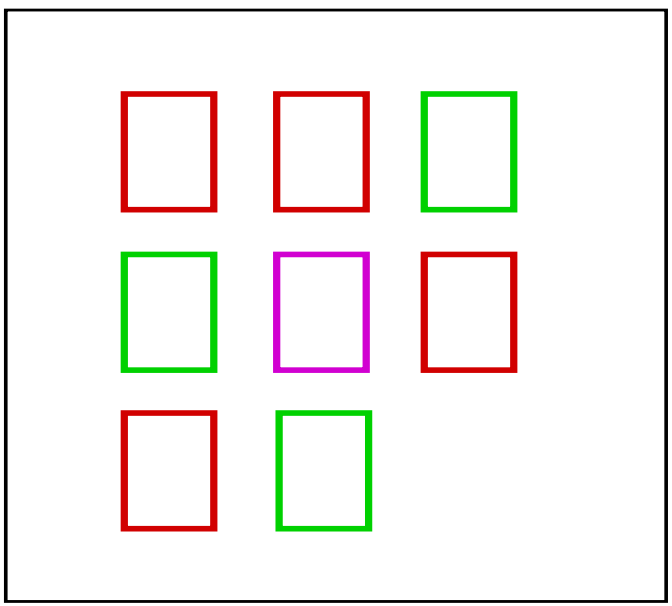
- Allocation pages physiques – les faire correspondre aux pages virtuelles, en optimisant différents critères ;
  - Cache processeur — garde une copie des pages physiques dernièrement utilisées, écrase ces copies quand on lui dépasse la capacité
  - schémas déterministes $\longrightarrow$ le cache, de capacité p pages, contient en position i une page d'adresse-mémoire vive $k \cdot p + i$

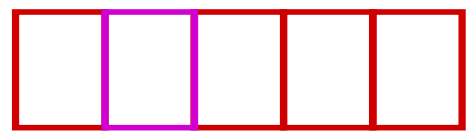


  - par exemple, les pages 0, 1, . . ., $p-1$ sont dans le cache. dès qu'on veut la page p, on jette du cache la page 0. il faut minimiser ces défauts de cache processeur en choisissant attentivement les nouvelles pages physiques fraîchement allouées

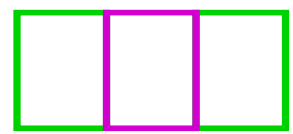
Quelques remarques

- Si plusieurs processus partagent certaines pages physiques, alors il faut également les choisir attentivement et les garder en mémoire, pour ne pas se faire pénaliser par beaucoup de défauts de page.





proc1

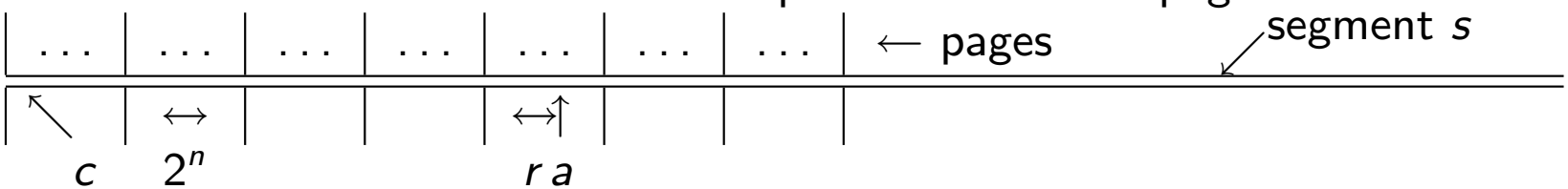


proc2

virt.

Mémoire virtuelle et mémoire vive réelle

- calcul correspondance mémoire virtuelle $\leftrightarrow$ mémoire physique vive
- une adresse virtuelle a (utilisée par exemple dans les instructions machine dans .text), référence un endroit dans le segment s du programme, contenant des pages de taille 2^n
- supposons que s commence à l'adresse virtuelle c.
- données d'entrée : a, c, n, s
- alors $t = [(a - c)/2^n]$ nous dit sur quelle page t de s tombe cette adresse a,
- le reste r de cette division donne le déplacement dans la page



- trouver l'adresse réelle mémoire-vive *p(t,s)* de la page t en question
- adresse réelle mémoire-vive correspondant à a est la somme $p(t, s) + r$
- on peut simplifier : en binaire, $a = \underline{|\overline{c}|t|r|}$, avec r écrit sur n bits, et $c = \underline{|\overline{c}|00...0|}$.
- et alors on peut directement parler de la page virtuelle $v = \underline{|\overline{c}|t|} = a \,\&\, (2^n - 1)$

Exemple

- suppons que le CPU référence l'adresse virtuelle *a=0x4abc3921* , les pages étant de *4K*=2^{12}, i.e. *n=12*.
- on écrit *a=0x*|4*abc*3|921|. ($2^{12} = (2^4)^3$ et un chiffre hexa fait quatre bits)
- alors on sait « tout de suite » que la page virtuelle où tombe cette adresse est la page *v=0x4abc3*.
- on a également « tout de suite » le déplacement *r=0x921* (les *n=12* derniers bits)
- supposons que cette page est déjà en mémoire vive
- alors elle est dans une table du processus,
- la MMU s'empresse de la rechercher dedans,
- la MMU trouve ainsi tout de suite l'adresse *p(v)* de la page physique,
- et calcule vite $p(v) + r$ et le CPU est contente (et le noyau n'est pas dérangé)

Correspondance mémoire virtuelle — mémoire réelle

- chaque processus a une table pour toutes ses pages virtuelles
- cette table est co-gérée par la MMU et par le noyau
- dans chaque entrée dans cette table, il y a
  - des drapeaux spéciaux (P - présente ou pas en mémoire vive, D - sale ou pas, RW – écrivable ou en lecture seulement, etc.) et,
  - l'adresse réelle *p(v)*

page virtuelle	D	P	RW	Adresse réelle
0	0	1	1	0xD3E ($\in$RAM)
1	0	0	0	0xabcd (/disque)
...				

Correspondance mémoire virtuelle — mémoire réelle

- Pour aller plus vite, dès que les correspondances sont établies, on mémorise *p(v)* dans une table associative $v \mapsto p(v)$ d'accès très rapide — la TLB (translation look-aside buffer), dans la MMU-même (matériel)

page et segment	adresse
v	*p(v)*
v'	*p(v')*
v''	*p(v'')*
...	...

- en parallèle
  - recalcul $v \longrightarrow p(v)$
  - accès TLB
- gain de temps
- effectué par le matériel — la MMU
- un échec déclenche le défaut de page mentionné – exception et saut au traitant de défauts de page du noyau
- celui-ci décide s'il fait éventuellement de la place, et puis charge ou simplement alloue, ou bien s'il envoie un `SIGSEGV` au processus.

Allocation de mémoire physique

- chaque processus a droit à un nombre maximal de pages
- le mécanisme gérant l'allocation de pages est le dérobeur de pages : quand il n'y a plus de place, il en dérobe
  - à un autre processus trop gourmand
  - à un autre processus bloqué
  - au même processus
- il doit faire le choix de la page à récupérer

Pages récupérables « facilement »

- pages read-only — ont toujours une image disque, donc du coup on pourrait les écraser — seul ennui : si on y accède souvent ?...
- pages non modifiées depuis leur chargement (bit D(irty) nul) — idem
- ce bit D est positionné à chaque écriture dans la page

Remarque

La MMU a bien du travail pour garder ces tables à jour...

Pages récupérables « avec un petit effort »

- si le bit D est positionné, la page est sale, son contenu ne doit pas être écrasé (pile, ou tas, par exemple)
- alors il faut la sauvegarder sur disque avant de récupérer son espace :
  - recherche d'un espace disque dans la zone de swap
  - lancer le transfert RAM→disque
  - mettre à jour la table pour le segment et la page virtuelle en question (bit P maintenant à zéro, l'adresse réelle maintenant celle sur disque, etc.)
  - attendre la fin effective du transfert
  - récupérer enfin l'espace mémoire vive

Vieillissement des pages

- pour garder tout de même les pages utilisées plus récemment, Linux associe deux bits d'âge pour chaque page virtuelle — quatre passes
- dans la table des pages, il y a un bit A pour l'accès (lecture ou écriture), mis à un par la MMU à chaque accès
- toutes les 50 ms, un processus de Linux passe en revue toutes les pages virtuelles de chaque processus
- algorithme :
  - si le bit A est à 1 alors
    - Linux remet les bits d'âge de la page à 00 et le bit A à 0
  - sinon
    - Linux incrémente de 1 les bits d'âge de la page à 00
    - si les bits d'âge de la page sont 11, alors
    - la page virtuelle est trop vieille
    - la page physique correspondante est récupèrée par le dérobeur (comme on vient d'expliquer)

Pages mémoire en RAM

- possible de demander au noyau de garder pages en RAM
- applications temps réel, haute sécurité

```
#include <sys/mman.h>
int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
int mlockall(int flags);
int munlockall(void);
```

- processus – si privilégié – pas de limite, sinon, `RLIMIT_MEMLOCK`