Systèmes d'exploitation et Programmation système (RS)

Lucas Nussbaum <lucas.nussbaum@loria.fr>

Supports de cours, TD et TP largement basés sur ceux de Martin Quinson <martin.quinson@loria.fr>

Telecom Nancy – 2ième année

2016-2017

À propos de ce document

Document diffusé selon les termes de la licence

Licence Creative Commons version 3.0 France (ou ultérieure)

Attribution ; Partage dans les mêmes conditions

http://creativecommons.org/licenses/by-sa/3.0/fr/

Remerciements

- Sacha Krakoviack pour son cours et Bryant et O'Hallaron pour leur livre
- Les (autres) emprunts sont indiqués dans le corps du document

Aspects techniques

- Document LaTeX (classe latex-beamer), compilé avec latex-make
- Schémas : Beaucoup de xfig, un peu de inkscape

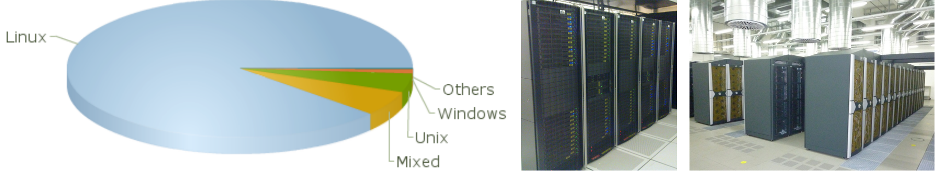
Site du cours : http://members.loria.fr/lnussbaum/RS/

- TD/TP, exams et projets (sources disponibles aux enseignants sur demande)

À propos de moi. . .

Lucas Nussbaum

- Formation : ingénieur ENSIMAG (2005), Doctorat (2008)
- Depuis 2009 :
  - Enseignant-chercheur (Maître de conférences) à l'univ. de Lorraine
    - Principalement en licence professionnelle ASRALL
      (Administration de Systèmes, Réseaux et Applications à base de Logiciel Libre)
  - Chercheur dans l'équipe MADYNES du LORIA
- Recherche : Systèmes distribués, calcul à haute performance, Cloud



- Contributeur au logiciel libre
  Debian (Project Leader 2013-2015, *Quality Assurance*), Ruby
- Plus d'infos :
  - `http://members.loria.fr/lnussbaum/` (`Lucas.Nussbaum@loria.fr`)

Organisation pratique du module

Module en deux parties

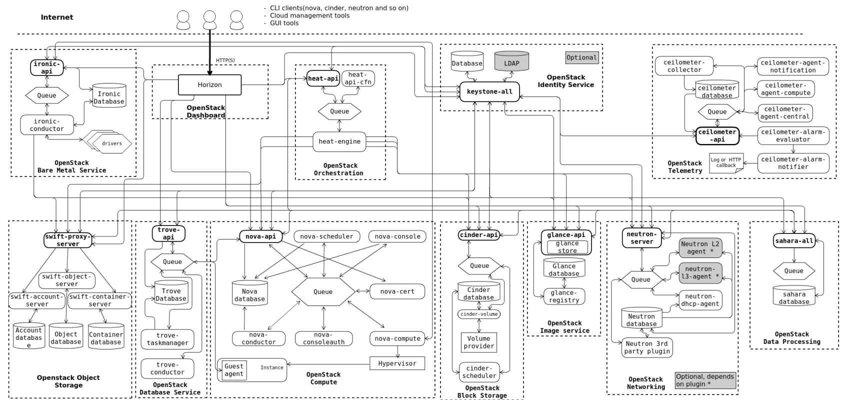
- Partie système (intervenant en cours : Lucas Nussbaum)
  - 5 cours, 3 TD, 3 TP
  - Examen sur table (mi octobre)
    - Documents interdits sauf un A4 R/V **manuscrit**
  - un projet (pour novembre)
    - Binômes et Git obligatoires
    - Le sujet arrive bientôt...
- Partie réseaux (intervenant en cours : Isabelle Chrisment)

Implication

- Manipulation : programmez ! Expérimentez !
- Questions bienvenues : pendant/après le cours, par mail, etc.

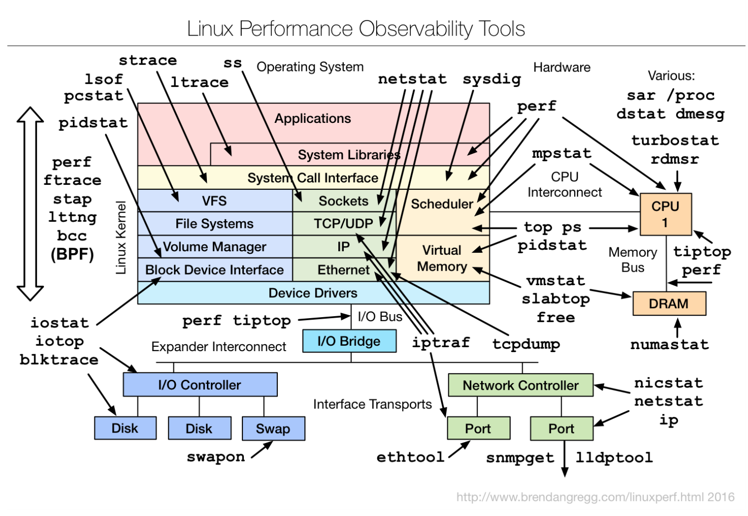
Pourquoi un cours de système ?

- Quatre concepts fondamentaux de l'Informatique (G. Dowek) : Information, **Machine**, Algorithme, Langage
- Les architectures et infrastructures modernes sont complexes
  - Wikipedia : 1145 serveurs, 30 900 CPUs
  - OVH : 250 000 serveurs
  - OpenStack (pile logicielle permettant de créer un *Cloud* privé)



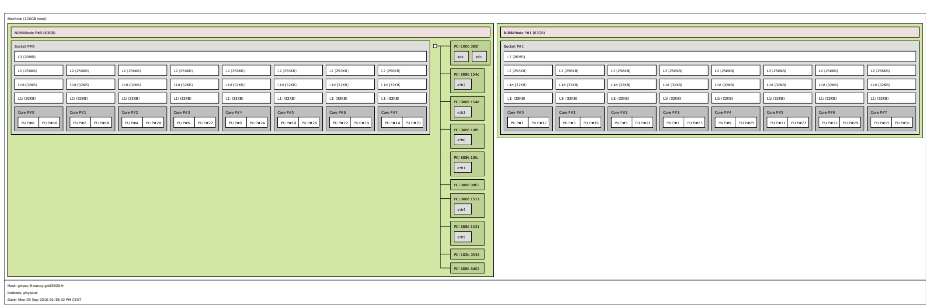
Pourquoi un cours de système ? (2)

- Noyau Linux (et outils d'observation)



Pourquoi un cours de système ? (3)

- Système dual-socket Intel récent (2x Intel E5-2630v3, 8 cœurs/CPU)



Pourquoi un cours de système ? (4)

Comment les utiliser efficacement ? Comment les concevoir ?
$\leadsto$ Performances, sécurité, résilience, efficacité énergétique

Métiers (à la sortie de TELECOM Nancy) :

- **IT Operations / Administration système et réseaux**
  $\leadsto$ Assurer le maintien en conditions opérationnelles d'une infrastructure (suivi des incidents, montées de version, etc.)
- Mais mouvement vers le modèle **DevOps** ($\approx$Google Site Reliability Engineers)
  - Suppression des silos *software development* vs *operations*
  - Infrastructure as Code
  - Itérations rapides, automatisation, tests automatiques

  Compétences nécessaires : développement logiciel, compréhension profonde des systèmes (combinaison très recherchée sur le marché du travail)
- **Systèmes embarqués** : domotique, industrie automobile, *appliances* diverses
- **Sécurité informatique** (souvent lié à des aspects système/réseau)
- Même comme pur développeur, savoir ce qui se passe sous le capot est utile !

Objectif du module

Utiliser efficacement le système d'exploitation

Contenu et objectifs du module

- Grandes lignes du fonctionnement d'un système d'exploitation (OS)
  Focus sur UNIX (et Linux) par défaut, mais généralisations
- Concepts clés des OS : processus, fichier, édition de liens, synchronisation
- Utilisation des interfaces système : programmation pratique, interface POSIX
- Programmation système (et non programmation interne *du* système)
  Plutôt du point de vue de l'utilisateur (conception d'OS en RSA)

Motivations

- OS = systèmes complexes les plus courants ; Concepts et abstractions claires
- Impossible de faire un programme efficace sans comprendre l'OS
- Comprendre ceci aide à comprendre les abstractions supérieures

Prérequis : Pratique du langage C et du shell UNIX

Bibliographie succincte (pour cette partie)

Livres

- Bryant, O'Hallaron : Computer Systems, A Programmer's Perspective.

Autres cours disponibles sur Internet

- **Introduction aux systèmes et aux réseaux (S. Krakowiak, Grenoble)**
  Source de nombreux transparents présentés ici.
  `http://sardes.inrialpes.fr/~krakowia/Enseignement/L3/SR-L3.html/`
- **Programmation des systèmes** (Ph. Marquet, Lille)
  `http://www.lifl.fr/~marquet/cnl/pds/`
- Operating Systems and System Programming (B. Pfaff, Stanford)
  `http://cs140.stanford.edu/`

Sites d'information

- `http://systeme.developpez.com/cours/`
  Index de cours et tutoriels sur les systèmes

URL du cours : `http://members.loria.fr/lnussbaum/RS/`

Plan de cette partie du module :

Systèmes d'exploitation et programmation système

1. Introduction
   Système d'exploitation : interface du matériel et gestionnaire des ressources.

2. Processus
   Processus et programme ; Utilisation des processus UNIX et Réalisation ; Signaux.

3. Fichiers et entrées/sorties
   Fichiers et systèmes de fichiers ; Utilisation ; Réalisation.

4. Exécution des programmes
   Schémas d'exécution : interprétation (shell) et compilation (liaison et bibliothèques)

5. Synchronisation entre processus
   Problèmes classiques (compétition, interblocage, famine) ; Schémas classiques.

6. Programmation concurrente
   Qu'est ce qu'un thread ; Modèles d'implémentation ; POSIX threads.

Premier chapitre

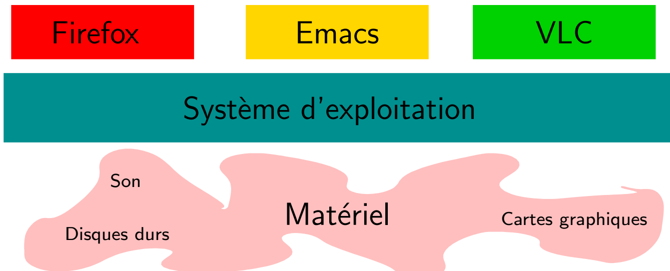
Introduction

- Qu'est ce qu'un système d'exploitation ?
- Pourquoi UNIX ?
- Interfaces du système d'exploitation
- Protection des ressources
- Conclusion

Qu'est ce qu'un système d'exploitation

Logiciel entre les applications et le matériel

- Offre une interface unifiée aux applications
- Gère (et protège) les ressources

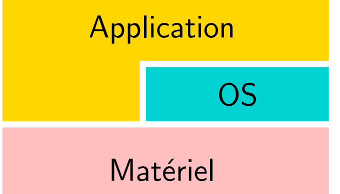


Évolution des systèmes d'exploitation : Étape 0

À l'origine, pas de système d'exploitation

Une machine, un utilisateur, un logiciel

- Tout au plus une bibliothèque de services standards
  Exemples : premières machines, systèmes embarqués (voitures, ascenseurs)
- Complexité conceptuelle réduite ...
- ... mais complexité technique accrue : on refait tout à chaque fois ($\Rightarrow$ cher)



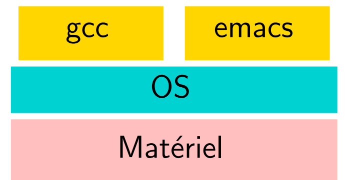
- Surtout, cela pose un problème d'efficacité :
  Si le programme est bloqué (disque, réseau, humain), la machine est inutilisée

Évolution des systèmes d'exploitation : Étape 1

Maximiser le temps d'utilisation du matériel

Une machine, plusieurs logiciels

- Problème fondamental de la précédente solution :
  Si le programme est bloqué (disque, réseau, humain), la machine est inutilisée
- Solution : **plusieurs processus**. Si l'un est bloqué, on exécute les autres.

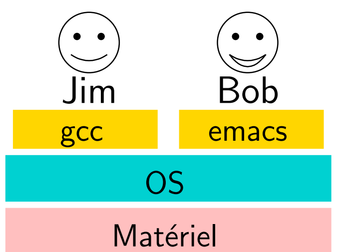


- Mais que se passe-t-il si un processus :
  - Fait une boucle infinie
  - Écrit dans la mémoire des autres processus
- Le système doit donc fournir une certaine protection :
  $\Rightarrow$ interposition entre les applications et le matériel

Évolution des systèmes d'exploitation : Étape 2

Une machine, plusieurs logiciels, plusieurs utilisateurs

- La solution précédente est chère (en 1970) : il faut une machine par utilisateur.
- Solution : **partage de la machine entre utilisateurs**
  (les utilisateurs tapent moins vite que l'ordinateur ne compile)

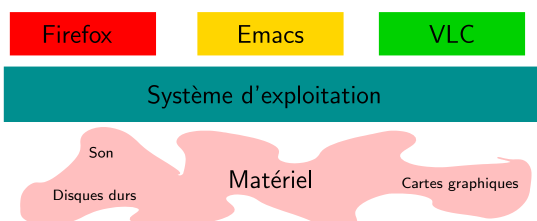


- Mais que se passe-t-il si les utilisateurs sont :
  - Trop gourmands ou trop nombreux
  - Mal intentionnés
- Le système doit protéger et gérer les ressources

Rôle de l'OS : intermédiaire matériel ↔ applications

Deux fonctions complémentaires

- Adaptation d'interface : offre une interface plus pratique que le matériel
  - Dissimule les détails de mise en œuvre (abstraction)
  - Dissimule les limitations physiques (taille mémoire) et le partage des ressources
- Gestion des ressources (mémoire, processeur, disque, réseau, affichage, etc.)
  - Alloue les ressources aux applications qui le demandent
  - Partage les ressources entre les applications
  - Protège les applications les unes des autres ; empêche l'usage abusif de ressources



Fonctions du système d'exploitation

	Organe physique	Entité logique
Gestion de l'activité		
Déroulement de l'exécution (processus)	Processeur	Processus
Événements matériels		
Gestion de l'information		
Accès dynamique (exécution, mémoire)	Mémoire principale	Mémoire virtuelle
Conservation permanente		
Partage	Disque	Fichier
Gestion des communications		
Interface avec utilisateur	Écran clavier, souris	Fenêtre
Impression et périphériques	Imprimante	
Réseau	Réseau	
Protection		
de l'information	Tous	Toutes
des ressources		

Premier chapitre

Introduction

- Qu'est ce qu'un système d'exploitation ?
- Pourquoi UNIX ?
- Interfaces du système d'exploitation
- Protection des ressources
- Conclusion

UNIX, POSIX et les autres

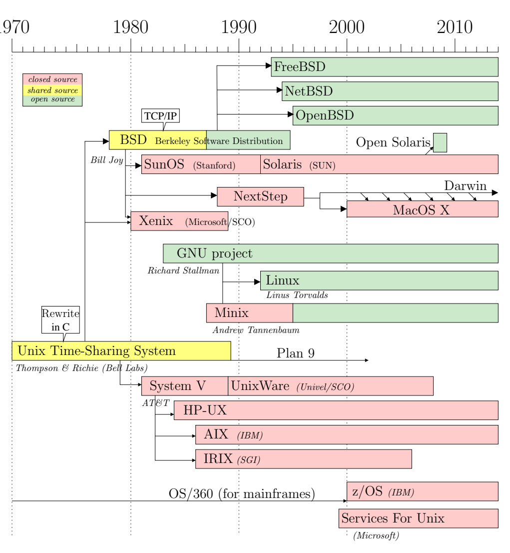
Qu'est ce qu'UNIX ? Pourquoi étudier UNIX ?

- Famille de systèmes d'exploitation, nombreuses implémentations
- Impact historique primordial, souvent copié
- Concepts de base simple, interface *historique* simple
- Des versions sont *libres*, tradition de code ouvert

Et pourquoi pas Windows ?

- En temps limité, il faut bien choisir ; je connais mieux les systèmes UNIX
- Même s'il y a de grosses différences, les concepts sont comparables
- D'autres cours à l'ESIAL utilisent Windows
  occasion d'étudier ce système ; le cours de RS permet d'aller plus vite
- Système plus *transparent* que Windows : plus facile de comprendre ce qui se passe sous le capot

Historique



1965 MULTICS : projet de système ambitieux (Bell Labs)

1969 Bell Labs se retire de MULTICS, Début de UNICS

1970 Unix projet Bell Labs officiel

1973 Réécriture en C
Distribution source aux universités
Commercialisation par AT&T

80-90 Unix War : BSD vs. System V

90-10 Effort de normalisation :

- Posix1(88) processus, signaux, tubes
- Posix1.b(93) semaphores, shared memory
- Posix1.c (95) threads
- Posix2 (92) interpreteur
- Posix :2001, Posix :2004, Posix :2008 Mises à jour

Premier chapitre

Introduction

- Qu'est ce qu'un système d'exploitation ?
- Pourquoi UNIX ?
- Interfaces du système d'exploitation
- Protection des ressources
- Conclusion

Notion d'interface

Un service est caractérisé par son interface

- L'interface est l'ensemble des fonctions accessibles aux utilisateurs du service
- Chaque fonction est définie par :
  - son format : la description de son mode d'utilisation (syntaxe)
  - sa spécification : la description de son effet (sémantique)
- Ces descriptions doivent être :
  précises, complètes (y compris les cas d'erreur), non ambigües.

Principe de base : séparation entre interface et mode de réalisation

Avantages :

- Facilite la portabilité
  - Transport d'une application utilisant le service sur une autre réalisation
  - Passage d'un utilisateur sur une autre réalisation du système
- Les réalisations sont interchangeables facilement

Dans POSIX

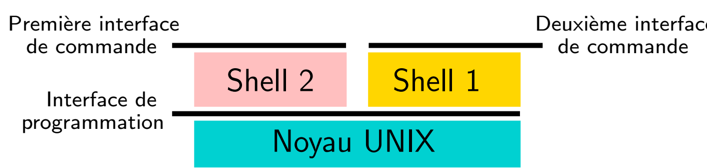
- C'est l'interface qui est normalisée, pas l'implémentation

Interfaces d'un système d'exploitation

En général, deux interfaces

- Interface de programmation (**A**pplication **P**rogramming **I**nterface)
  - Utilisable à partir des programmes s'exécutant sous le système
  - Composée d'un ensemble d'appels systèmes (procédures spéciales)
- Interface de commande ou interface de l'utilisateur
  - Utilisable par un usager humain, sous forme textuelle ou graphique
  - Composée d'un ensemble de commandes
    - Textuelles (exemple en UNIX : `rm *.o`)
    - Graphiques (exemple : déplacer l'icone d'un fichier vers la corbeille)

Exemple sous UNIX :



Exemple d'usage des interfaces d'UNIX

- **Interface de programmation**
  Ci-contre : programme copiant un fichier dans un autre
  (`read` et `write` : appels système)

```
while (nb_lus = read(fich1, buf, BUFFSIZE )) {
  if ((nb_lus == -1) && (errno != EINTR)) {
    break; /* erreur */
  } else if (nb_lus > 0) {
    pos_buff = buf;
    a_ecrire = nb_lus;
    while (nb_ecrits =
            write(fich2,pos_buff,a_ecrire)) {
      if ((nb_ecrits == -1) && (errno != EINTR))
        break;  /* erreur */
      else if (a_ecrire == 0)
        break;  /* fini */
      else if (nb_ecrits > 0) {
        pos_buff += nb_ecrits;
        a_ecrire -= nb_ecrits;
} } } }
```

- **Interface de commande**
  Commande shell :

```
$ cp fich1 fich2
```

  recopie fich1 dans fich2

- **Documentation**
  `man 1 <nom>` : documentation des commandes (par défaut)
  `man 2 <nom>` : documentation des appels système
  `man 3 <nom>` : documentation des fonctions

  d'autres sections, mais plus rares. Voir `man 7 man` ;)

Découpage en couches du système

Niveau utilisateur

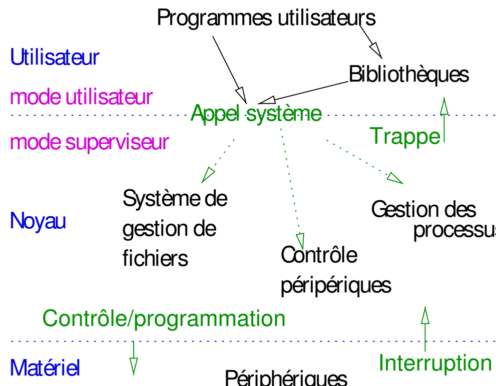
- Applications utilisateur
- Logiciels de base
- Bibliothèque système
- Appels de fonction

Niveau noyau

- Gestion des processus
- Système de fichier
- Gestion de la mémoire

Niveau matériel

- Firmware, contrôleur, SOC



OS = bibliothèque système + noyau

(d'où le nom GNU/Linux)

- Ce cours : bibliothèque système
- RSA : le noyau

(c) Philippe Marquet, CC-BY-NC-SA.

Premier chapitre

Introduction

- Qu'est ce qu'un système d'exploitation ?
- Pourquoi UNIX ?
- Interfaces du système d'exploitation
- Protection des ressources
- Conclusion

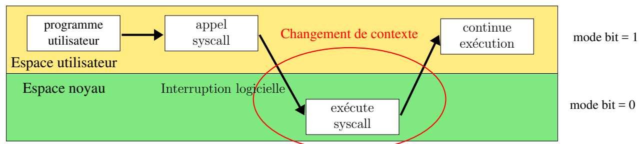
Protection des ressources (rapide introduction – cf. RSA)

Méthodes d'isolation des programmes (et utilisateurs) dangereux

- Préemption : ne donner aux applications que ce qu'on peut leur reprendre
- Interposition : pas d'accès direct, vérification de la validité de chaque accès
- Mode privilégié : certaines instructions machines réservées à l'OS

Exemples de ressources protégées

- Processeur : préemption
  - Interruptions (matérielles) à intervalles réguliers $\rightarrow$ contrôle à l'OS
  - (l'OS choisit le programme devant s'exécuter ensuite)
- Mémoire : interposition (validité de tout accès mémoire vérifiée au préalable)
- Exécution : mode privilégié (espace noyau) ou normal (espace utilisateur)
  - Le CPU bloque certaines instructions assembleur (E/S) en mode utilisateur



Limites de la protection

Les systèmes réels ont des failles

Les systèmes ne protègent pas de tout

- `while true ; do mkdir toto; cd toto; done` (en shell)
- `while(1) { fork(); }` (en C)
- `while(1) { char *a=malloc(512); *a='1'; }` (en C)

Réponse classique de l'OS : gel (voire pire)
On suppose que les utilisateurs ne sont pas mal intentionnés (erreur ?)

> *Unix was not designed to stop people from doing stupid things, because that would also stop them from doing clever things.*
>
> – Doug Gwyn

Deux types de solutions

Technique : mise en place de quotas

Sociale : "éduquer" les utilisateurs trop gourmands

Résumé du premier chapitre

- Qu'est ce qu'un système d'exploitation ?
  - Un intermédiaire entre les applications et le matériel
- Rôles et fonctions du système d'exploitation
  - Offrir une interface du matériel unifiée et plus adapté
  - Assurer la gestion et la protection des ressources
- Les différentes interfaces d'un système d'exploitation
  - Interface de programmation (API)
  - Interface de commande (shell, environnement graphique)
- Techniques classiques de protection des ressources
  - Préemption
  - Interposition
  - Mode d'exécution (protégé ou non)

Deuxième chapitre

Processus

- Introduction
- Utilisation des processus UNIX
  - Mémoire virtuelle, environnement
  - Création des processus dans Unix
  - Quelques interactions entre processus dans Unix
- Réalisation des processus UNIX
- Communication par signaux
  - Principe et utilité
  - Terminaux, sessions et groupe en Unix
  - Exemples d'utilisation des signaux
- Conclusion

Qu'est ce qu'un processus ?

Définition formelle :

- Entité dynamique représentant l'exécution d'un programme sur un processeur

Du point de vue du système d'exploitation :

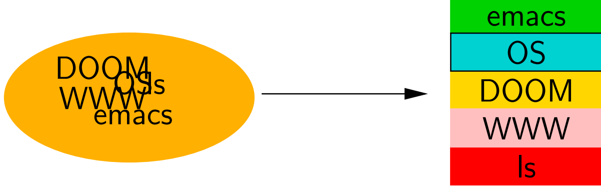
- Espace d'adressage (mémoire, contient données + code)
- État interne (compteur d'exécution, fichiers ouverts, *etc.*)

Exemples de processus :

- L'exécution d'un programme
- La copie d'un fichier sur disque
- La transmission d'une séquence de données sur un réseau

Utilité des processus : simplicité

- L'ordinateur a des activités différentes
- Comment les faire cohabiter simplement ?
  - En plaçant chaque activité dans un processus isolé
    L'OS s'occupe de chacun de la même façon, chacun ne s'occupe que de l'OS



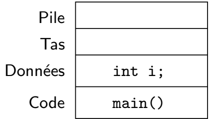
- La décomposition est une réponse classique à la complexité

Ne pas confondre processus et programme

- Programme :
  Code + données (**passif**)

```
int i;
int main() {
  printf("Salut\n");
}
```

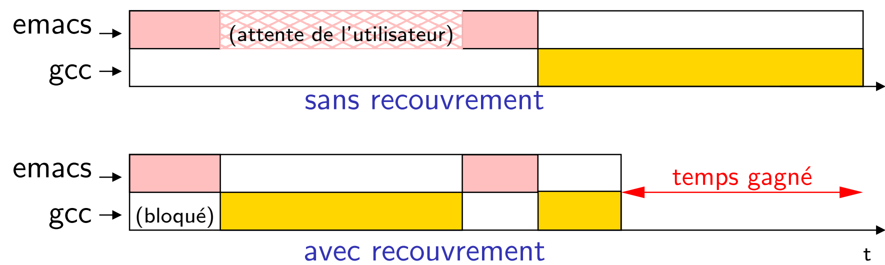
- Processus :
  Programme **en cours d'exécution**



- Vous pouvez utiliser le même programme que moi, mais ça ne sera pas le même processus que moi
- Même différence qu'entre classe d'objet et instance d'objet

Utilité des processus : efficacité

- Les communications bloquent les processus
  (communication au sens large : réseau, disque ; utilisateur, autre programme)
  ⇒ **recouvrement des calculs et des communications**



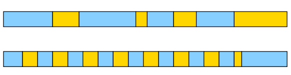
- Parallélisme sur les machines multi-processeurs

Parallélisme et pseudo-parallélisme

Que faire quand deux processus sont prêts à s'exécuter ?

- Si deux processeurs, tout va bien.
- Si non, FCFS ? Mauvaise interactivité !

- Pseudo-parallélisme = chacun son tour
- Autre exécution pseudo-parallèle



Le pseudo-parallélisme

- fonctionne grâce aux interruptions matérielles régulières rendant contrôle à OS
- permet également de recouvrir calcul et communications

  On y reviendra.

Relations entre processus

Compétition

- Plusieurs processus veulent accéder à une ressource exclusive (*i.e.* ne pouvant être utilisée que par un seul à la fois) :
  - Processeur (cas du pseudo-parallélisme)
  - Imprimante, carte son
- Une solution possible parmi d'autres :
  FCFS : premier arrivé, premier servi (les suivants attendent leur tour)

Coopération

- Plusieurs processus collaborent à une tâche commune
- Souvent, ils doivent se synchroniser :
  - p1 produit un fichier, p2 imprime le fichier
  - p1 met à jour un fichier, p2 consulte le fichier
- La synchronisation se ramène à :
  p2 doit attendre que p1 ait franchi un certain point de son exécution

Faire attendre un processus

Primordial pour les interactions entre processus

Attente active

Processus 1

```
while (ressource occupée)
   { };
ressource occupée = true;
...
```

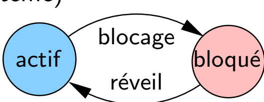
Processus 2

```
ressource occupée = true;
utiliser ressource;
ressource occupée = false;
...
```

- Gaspillage de ressource si pseudo-parallélisme
- Problème d'atomicité (*race condition* – on y reviendra)

Blocage du processus

- Définition d'un nouvel état de processus : bloqué
(exécution suspendue ; réveil explicite par un autre processus ou par le système)




```
  ...
sleep(5); /* se bloquer pour 5 secondes */
  ...
```

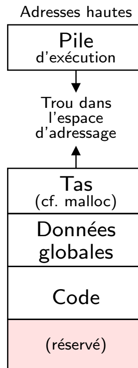
Deuxième chapitre

Processus

- Introduction
- Utilisation des processus UNIX
  - Mémoire virtuelle, environnement
  - Création des processus dans Unix
  - Quelques interactions entre processus dans Unix
- Réalisation des processus UNIX
- Communication par signaux
  - Principe et utilité
  - Terminaux, sessions et groupe en Unix
  - Exemples d'utilisation des signaux
- Conclusion

Processus UNIX

- Processus = exécution d'un programme
  - Commande (du langage de commande)
  - Application
- Un processus comprend :
  - Une mémoire qui lui est propre (mémoire virtuelle)
  - **Contexte** d'exécution (pile, registres du processeur)
- Les processus sont identifiés par leur **pid**
  - Commande `ps` : liste des processus
  - Commande `top` : montre l'activité du processeur
  - Primitive `getpid()` : renvoie le pid du processus courant



Environnement d'un processus

- Ensemble de variables accessibles par le processus
  (sorte de configuration)
- Principaux avantages :
  - L'utilisateur n'a pas à redéfinir son contexte pour chaque programme
    Nom de l'utilisateur, de la machine, terminal par défaut, . . .
  - Permet de configurer certains éléments
    Chemin de recherche des programmes (PATH), shell utilisé, . . .
- Certaines sont prédéfinies par le système (et modifiables par l'utilisateur)
- L'utilisateur peut créer ses propres variables d'environnement
- Interface (dépend du shell) :

Commande tcsh	Commande bash	Action
`setenv`	`printenv`	affiche toutes les variables définies
`setenv VAR <valeur>`	`export VAR=<valeur>`	attribue la valeur à la variable
`echo $VAR`	`echo $VAR`	affiche le contenu de la variable

- Exemple : `export DISPLAY=blaise.loria.fr:0.0` défini le terminal utilisé
- L'interface de programmation sera vue en TD/TP

Vie et mort des processus

Tout processus a un début et une fin

- Début : création par un autre processus
  - `init` est le processus originel : pid=1 (`launchd` sous mac)
    Créé par le noyau au démarrage, il lance les autres processus système
- Fin
  - Auto-destruction (à la fin du programme) (par exit)
  - Destruction par un autre processus (par kill)
  - Destruction par l'OS (en cas de violation de protection et autres)
  - Certains processus ne se terminent pas avant l'arrêt de la machine
    - Nommés ≪démons≫ (disk and execution monitor → daemon)
    - Réalisent des fonctions du système (login utilisateurs, impression, serveur web)

Création de processus dans UNIX

- Dans le langage de commande :
  - Chaque commande est exécutée dans un processus séparé
  - On peut créer des processus en (pseudo-)parallèle :

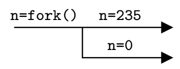
```
$ prog1 & prog2 & # crée deux processus pour exécuter prog1 et prog2
$ prog1 & prog1 & # lance deux instances de prog1
```

- Par l'API : clonage avec l'appel système fork (*cf.* transparent suivant)

Création des processus dans Unix (1/3)

Appel système `pid_t fork()`

- Effet : clone le processus appelant
- Le processus créé (fils) est une copie conforme du processus créateur (père)
  Copies conformes comme une bactérie qui se coupe en deux
- Ils se reconnaissent par la valeur de retour de `fork()` :
  - Pour le père : le `pid` du fils (ou –1 si erreur)
  - Pour le fils : 0



Exemple :

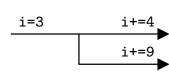
```
if (fork() != 0) {
   printf("je suis le père, mon PID est %d\n", getpid());
} else {
   printf("je suis le fils, mon PID est %d; mon père est %d\n",
          getpid(), getppid());
   /* en général exec(), (exécution d'un nouveau programme) */
}
```

Création des processus dans Unix (2/3)

Duplication du processus père $\Rightarrow$ duplication de l'espace d'adressage

```
int i=3;
if (fork() != 0) {
     printf("je suis le père, mon PID est %d\n", getpid());
     i += 4;
} else {
     printf("je suis le fils, mon PID est %d; mon père est %d\n",
             getpid(), getppid());
     i += 9;
}
printf("pour %d, i = %d\n", getpid(), i);
```

```
je suis le fils, mon PID est 10271; mon père est 10270
pour 10271, i = 12
je suis le père, mon PID est 10270
pour 10270, i = 7
```

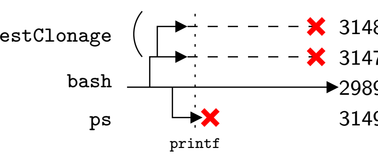


Création des processus dans Unix (3/3)

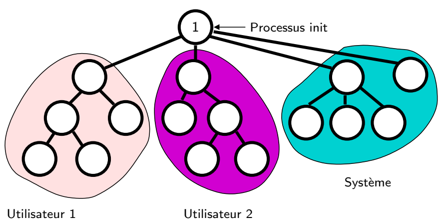
testClonage.c

```
int main() {
  if (fork() != 0) {
    printf("je suis le père, mon PID est %d\n", getpid());
    sleep(10) /* blocage pendant 10  secondes */
    exit(0);
 } else {
    printf("je suis le fils, mon PID est %d\n", getpid());
    sleep(10) /* blocage pendant 10  secondes */
    exit(0);
  }
}
```

```
$ gcc -o testClonage testClonage.c
$ ./testClonage & ps
je suis le fils, mon PID est 3148
je suis le père, mon PID est 3147
[2] 3147
 PID TTY          TIME CMD
2989 pts/0 00:00:00 bash
3147 pts/0 00:00:00 testClonage
3148 pts/0 00:00:00 testClonage
3149 pts/0 00:00:00 ps
$
```



Hiérarchie de processus Unix



- Quelques appels systèmes utiles :
  - getpid() : obtenir le numéro du processus
  - getppid() : obtenir le numéro du père
  - getuid() : obtenir le numéro d'usager (auquel appartient le processus)

Quelques interactions entre processus (1/2)

Envoyer un signal à un autre processus

- En langage de commande, `kill <pid>` tue `pid` (plus de détails plus tard)

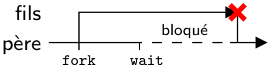
Faire attendre un processus

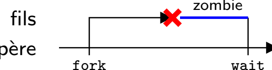
- `sleep(n)` : se bloquer pendant n secondes
- `pause()` : se bloquer jusqu'à la réception d'un signal (cf. plus tard)

Quelques interactions entre processus (2/2)

Synchronisation entre un processus père et ses fils

- Fin d'un processus : `exit(etat)`
  etat est un code de fin (convention : 0 si ok, code d'erreur sinon – *cf.* `errno`)
- Le père attend la fin de l'un des fils : `pid_t wait(int *ptr_etat)`
  retour : pid du fils qui a terminé ; code de fin stocké dans `ptr_etat`
- Attendre la fin du fils `pid` :
  `pid_t waitpid(pid_t pid, int *ptr_etat, int options)`
- Processus zombie : terminé, mais le père n'a pas appelé wait().
  Il ne peut plus s'exécuter, mais consomme encore des ressources. À éviter.





Faire attendre un processus

Fonction sleep()

- Bloque le processus courant pour le nombre de secondes indiqué
- `unsigned int sleep(unsigned int seconds);`
- `usleep()` et `nanosleep()` offrent meilleures résolutions (micro, nanoseconde) mais interfaces plus compliquées et pas portables

Exercice : la fonction somnole (à compléter)

Écrire une fonction qui affiche à chaque seconde le temps restant à dormir :

Exemple d'affichage :

Déjà dormi 1 secondes sur 3
Déjà dormi 2 secondes sur 3
Déjà dormi 3 secondes sur 3

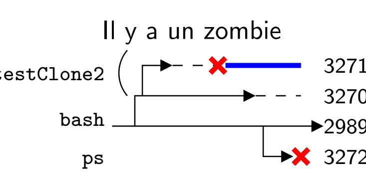
```
void somnole(unsigned int secondes) {
  int i;
  for (i=0; i<secondes; i++) {
   printf("Déjà dormi %d secondes sur %d\n",
         i, secondes);
   sleep(1);
  }
}
```

Exemple de synchronisation entre père et fils

testClone2.c

```
int main() {
  if (fork() != 0) {
    printf("je suis le père, mon PID est %d\n", getpid());
    while (1) ; /* boucle sans fin sans attendre le fils */
  } else {
    printf("je suis le fils, mon PID est %d\n", getpid());
    sleep(2) /* blocage pendant 2 secondes */
    printf("fin du fils\n");
    exit(0);
} }
```

```
$ gcc -o testClone2 testClone2.c
$ ./testClone2
je suis le fils, mon PID est 3271
je suis le père, mon PID est 3270
fin du fils
->l'utilisateur tape <ctrl-Z> (suspendre)
Suspended
$ ps
 PID TTY         TIME CMD
2989 pts/0 00:00:00 bash
3270 pts/0 00:00:03 testClone2
3271 pts/0 00:00:00 testClone2 <defunct>
3272 pts/0 00:00:00 ps
$
```



Autre exemple de synchronisation père fils

testClone3.c

```
#include <sys/types.h>
#include <sys/wait.h>
int main() {
 if (fork() != 0) {
  int statut; pid_t fils;
  printf("Le père (%d) attend.\n", getpid());
  fils = wait(&statut);
  if (WIFEXITED(statut)) {
   printf("%d : fils %d terminé (code %d)\n",
        getpid(), fils, WEXITSTATUS(statut));
  };
  exit(0);
```

(suite de testClone3.c)

```
  } else { /* correspond au if (fork() != 0)*/
    printf("je suis le fils, PID=%d\n",
           getpid());
    sleep(2) /* blocage pendant 2 secondes */
    printf("fin du fils\n");
    exit(1);
} }
```

```
$ ./testClone3
je suis le fils, PID=3312
Le père (3311) attend
fin du fils
3311: fils 3312 terminé (code 1)
$ ps
 PID TTY          TIME CMD
2989 pts/0 00:00:00 bash
3313 pts/0 00:00:00 ps
$
```

Il n'y a pas de zombie

Exécution d'un programme spécifié sous UNIX

Appels systèmes exec

- Pour faire exécuter un nouveau programme par un processus
- Souvent utilisé immédiatement après la création d'un processus : `fork+exec` = lancement d'un programme dans un nouveau processus
- Effet : remplace la mémoire virtuelle du processus par le programme
- Plusieurs variantes existent selon le mode de passage des paramètres (tableau, liste, passage de variables d'environnement)
- C'est aussi une primitive du langage de commande (même effet)

Exemple :

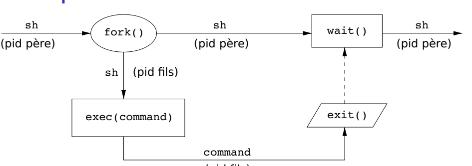
```
main() {
  if (fork() == 0) {

     code=execl("/bin/ls", "ls", "-a", 0); /* le fils  exécute : /bin/ls -a  .. */
     if (code != 0) { ... } /* Problème dans l'appel système; cf. valeur de errno */
  } else {
     wait(NULL); /* le père attend la fin du fils */
  }
  exit(0);
}
```

L'exemple du shell

Exécution d'une commande en premier plan

```
$ commande
```

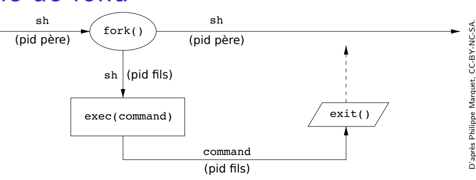
- 4 syscalls : `fork`, `exec`, `exit`, `wait`



Exécution d'une commande en tâche de fond

```
$ commande &
```

- Le shell ne fait pas `wait()`
- Il n'est plus bloqué



D'après Philippe Marquet, CC-BY-NC-SA.

Résumé du début du deuxième chapitre

- Utilité des processus
  - Simplicité (séparation entre les activités)
  - Sécurité (séparation entre les activités)
  - Efficacité (quand l'un est bloqué, passe à autre chose)
- Interface UNIX
  - Création : fork()
    - résultat=0 $\rightarrow$ je suis le fils
    - résultat>0 $\rightarrow$ je suis le père (résultat=pid fils)
    - résultat<0 $\rightarrow$ erreur
  - Attendre un fils : deux façons
    - wait() : n'importe quel fils
    - waitpid() : un fils en particulier
  - Processus zombie : un fils terminé dont le père n'a pas fait wait()
  - Bloquer le processus courant : deux façons
    - Jusqu'au prochain signal : pause()
    - Pendant 32 secondes : sleep(32)
  - Appel système `exec()` : remplace l'image du process actuel par le prog spécifié

Deuxième chapitre

Processus

- Introduction
- Utilisation des processus UNIX
  - Mémoire virtuelle, environnement
  - Création des processus dans Unix
  - Quelques interactions entre processus dans Unix
- Réalisation des processus UNIX
- Communication par signaux
  - Principe et utilité
  - Terminaux, sessions et groupe en Unix
  - Exemples d'utilisation des signaux
- Conclusion

Réalisation des processus

Processus = mémoire virtuelle + flot d'exécution

L'OS fournit ces deux ressources en allouant les ressources physiques

Objectif maintenant :

En savoir assez sur le fonctionnement de l'OS pour utiliser les processus

- À propos de mémoire
  - Organisation interne de la mémoire virtuelle d'un processus Unix

- À propos de processeur
  - Pseudo-parallélisme : allocation successive aux processus par tranches de temps

Objectifs repoussés à plus tard :

- Les autres ressources : disque (chapitre 3), réseau (seconde moitié du module)
- Détails de conception *sous le capot* (module RSA)

Allocation du processeur aux processus

Pseudo-parallélisme

Principe

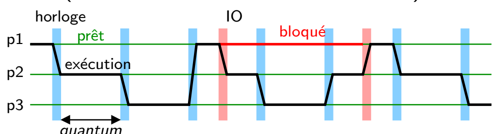
- Allocation successive aux processus par tranches de temps fixées (multiplexage du processeur par préemption)

Avantages

- Partage équitable du processeur entre processus (gestion + protection)
- Recouvrement calcul et communications (ou interactions)

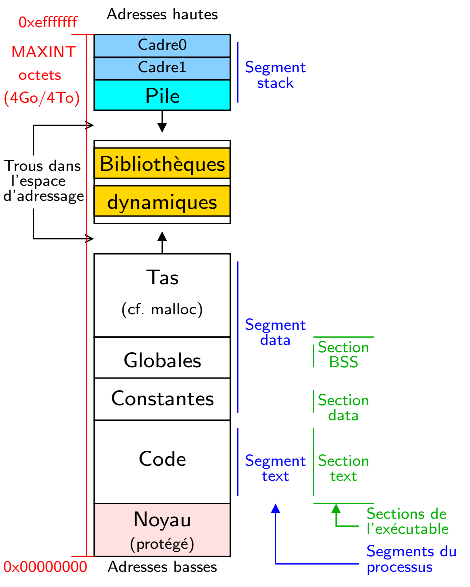
Fonctionnement

- Interruptions matérielles (top d'horloge, I/O, ...) rendent le contrôle à l'OS
- L'OS ordonnance les processus (choisit le prochain à bénéficier de la ressource)
- Il réalise la commutation de processus pour passer le contrôle à l'heureux élu
- (Comment ? Vous verrez en RSA !)



- quantum : ≈10ms (≈ millions d'instructions à 1Ghz)
- temps de commutation : ≈0,5ms
- `yield()` rend la main volontairement

Structure de la mémoire virtuelle d'un processus



- Pile : pour la récursivité
  - Cadres de fonction (*frame*)
  - Arguments des fonctions
  - Adresse de retour
  - Variables locales (non statique)
- Bibliothèques dynamiques
  - Code chargé ... dynamiquement
  - Intercalé dans l'espace d'adresses
- Données
  - Tas : malloc()
  - globales et `static` (modifiables)
  - Variables constantes
- `exec` lit l'exécutable et initialise la mémoire correspondante
- Noyau : infos sur le processus
  "Process Control Block"
  (pid, autorisations, fichiers ouverts, ...)
  (parfois au dessus de la pile)

Deuxième chapitre

Processus

- Introduction
- Utilisation des processus UNIX
  - Mémoire virtuelle, environnement
  - Création des processus dans Unix
  - Quelques interactions entre processus dans Unix
- Réalisation des processus UNIX
- Communication par signaux
  - Principe et utilité
  - Terminaux, sessions et groupe en Unix
  - Exemples d'utilisation des signaux
- Conclusion

Communication inter-processus (IPC) dans Unix

Aspect central de la programmation système

Moyens de communication entre processus sous Unix

- Signaux : suite de cette séance
- Fichiers et tubes (*pipes*, FIFOs) : partie 3.
- Files de messages : pas étudié [cette année]
- Mémoire partagée et sémaphores : partie 5.
- Sockets (dans les réseaux, mais aussi en local) : fin du semestre

Signaux

Définition : événement asynchrone

- Émis par l'OS ou un processus
- Destiné à un (ou plusieurs) processus

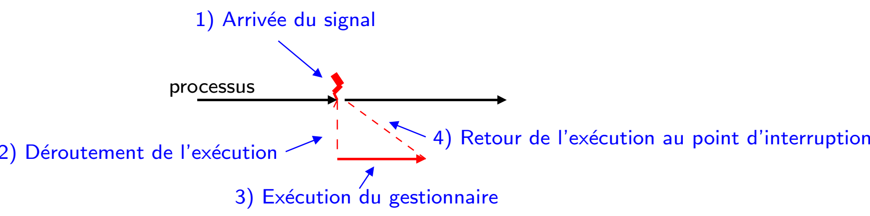
Intérêts et limites

- Simplifient le contrôle d'un ensemble de processus (comme le shell)
- Pratiques pour traiter des événements liés au temps
- Mécanisme de bas niveau à manipuler avec précaution
  (risque de perte de signaux en particulier)

Comparaison avec les interruptions matérielles :

- Analogie : la réception déclenche l'exécution d'un gestionnaire (*handler*)
- Différences : interruption reçue par processeur ; signal reçu par processus
  Certains signaux traduisent la réception d'une interruption (on y revient)

Fonctionnement des signaux



Remarques (on va détailler)

- On ne peut évidement *signaler* que ses propres processus (même `uid`)
- Différents signaux, identifiés par un nom symbolique (et un entier)
- Gestionnaire par défaut pour chacun
- Gestionnaire vide $\Rightarrow$ ignoré
- On peut changer le gestionnaire (sauf exceptions)
- On peut bloquer un signal : mise en attente, délivré qu'après déblocage
- Limites aux traitements possibles dans le gestionnaire (ex : pas de signal())

Quelques exemples de signaux

Nom symbolique	Cause/signification	Par défaut
SIGINT	frappe du caractère <CTRL-C>	terminaison
SIGTSTP	frappe du caractère <CTRL-Z>	suspension
SIGSTOP	blocage d'un processus (*)	suspension
SIGCONT	continuation d'un processus stoppé	reprise
SIGTERM	demande de terminaison	terminaison
SIGKILL	terminaison immédiate (*)	terminaison
SIGSEGV	erreur de segmentation (violation de protection mémoire)	terminaison +*core dump*
SIGALRM	top d'horloge (réglée avec `alarm`)	teminaison
SIGCHLD	terminaison d'un fils	ignoré
SIGUSR1	pas utilisés par le système	terminaison
SIGUSR2	(disponibles pour l'utilisateur)	terminaison

- KILL et STOP : ni bloquables ni ignorables ; gestionnaire non modifiable.
- Valeurs numériques associées (ex : SIGKILL=9), mais pas portable
- Voir man 7 signal pour d'autres signaux (section 7 du man : conventions)

core dump : copie image mémoire sur disque (premières mémoires : toriques → core ; dump=vidanger)

États d'un signal

Signal **pendant** (*pending*)

- Arrivé au destinataire, mais pas encore traité

Signal **traité**

- Le gestionnaire a commencé (et peut-être même fini)

Pendant, mais pas traité ? Est-ce possible ?

- Il est bloqué, càd retardé : il sera délivré lorsque débloqué
- Lors de l'exécution du gestionnaire d'un signal, ce signal est bloqué

Attention : au plus un signal pendant de chaque type

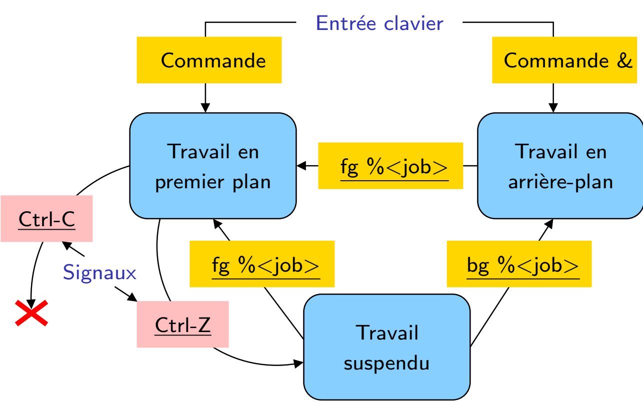
- L'information est codée sur un seul bit
- S'il arrive un autre signal du même type, le second est perdu

Deuxième chapitre

Processus

- Introduction
- Utilisation des processus UNIX
  - Mémoire virtuelle, environnement
  - Création des processus dans Unix
  - Quelques interactions entre processus dans Unix
- Réalisation des processus UNIX
- Communication par signaux
  - Principe et utilité
  - Terminaux, sessions et groupe en Unix
  - Exemples d'utilisation des signaux
- Conclusion

États d'un travail



- Travail (*job*) = (groupe de) processus lancé par une commande au shell
- Seul le travail en premier plan peut recevoir des signaux du clavier
- Les autres sont manipulés par des commandes

Terminaux, sessions et groupes en Unix

- Concept important pour comprendre les signaux sous Unix (détaillé plus tard)

- Une session est associée à un terminal, donc au login d'un utilisateur par shell
  Le processus de ce shell est le leader de la session.
- Plusieurs groupes de processus par session. On dit plusieurs travaux (*jobs*)

- Au plus un travail interactif (avant-plan, *foreground*)
  Interagissent avec l'utilisateur via le terminal, seuls à pouvoir lire le terminal
- Plusieurs travaux en arrière plan (*background*)
  Lancés avec & ; Exécution en travail de fond

- Signaux SIGINT (frappe de <CTRL-C>) et SIGTSTP (frappe de <CTRL-Z>)
  sont passés au groupe interactif et non aux groupes d'arrière-plan

Exemple avec les sessions et groupes Unix

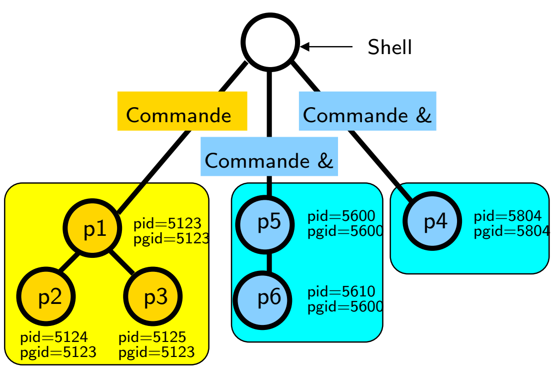
loop.c

```
int main() {
   printf"processus %d, groupe %d\n", getpid(), getpgrp());
   while(1) ;
}
```

```
$ loop & loop & ps
processus 10468, groupe 10468
[1] 10468
processus 10469, groupe 10469
[2] 10469
  PID TTY          TIME CMD
 5691 pts/0 00:00:00 bash
10468 pts/0 00:00:00 loop
10469 pts/0 00:00:00 loop
10470 pts/0 00:00:00 ps
$ fg %1
loop
[frappe de control-Z]
Suspended
$ jobs
[1] + Suspended         loop
[2] - Running           loop
```

```
$ bg %1
[1] loop &
$ fg %2
loop
[frappe de control-C]
$ ps
  PID TTY          TIME CMD
 5691 pts/0 00:00:00 bash
10468 pts/0 00:02:53 loop
10474 pts/0 00:00:00 ps
$ [frappe de control-C]
$ ps
  PID TTY          TIME CMD
 5691 pts/0 00:00:00 bash
10468 pts/0 00:02:57 loop
10475 pts/0 00:00:00 ps
$
```

Exemple de travaux



- Signaux CTRL-C et CTRL-Z adressés à **tous** les processus du groupe jaune
- Commandes shell `fg`, `bg` et `stop` pour travaux bleus

Deuxième chapitre

Processus

- Introduction
- Utilisation des processus UNIX
  - Mémoire virtuelle, environnement
  - Création des processus dans Unix
  - Quelques interactions entre processus dans Unix
- Réalisation des processus UNIX
- Communication par signaux
  - Principe et utilité
  - Terminaux, sessions et groupe en Unix
  - Exemples d'utilisation des signaux
- Conclusion

Envoyer un signal à un autre processus

Interfaces

- Langage de commande :

```
kill -NOM victime
```

- Appel système :

```
#include <signal.h>

int kill(pid_t victime, int sig);
```

Sémantique : à qui est envoyé le signal ?

- Si $victime > 0$, au processus tel que $\texttt{pid} = victime$
- Si $victime = 0$, à tous les processus du même groupe (`pgid`) que l'émetteur
- Si $victime = -1$:
  - Si super-utilisateur, à tous les processus sauf système et émetteur
  - Si non, à tous les processus dont l'utilisateur est propriétaire
- Si $victime < -1$, aux processus tels que $\texttt{pgid} = |victime|$ (tout le groupe)

Redéfinir le gestionnaire associé à un signal (POSIX)

Structure à utiliser pour décrire un gestionnaire

```
struct sigaction {
  void (*sa_handler)(int);                          /* gestionnaire, interface simple */
  void (* sa_sigaction) (int, siginfo_t *, void *);    /* gestionnaire, interface complète */
  sigset_t sa_mask;                            /* signaux à bloquer pendant le traitement */
  int sa_flags;                                                           /* options */
}
```

- Gestionnaires particuliers : SIG_DFL : action par défaut ; SIG_IGN : ignorer signal
- Deux types de gestionnaires
  - `sa_handler()` connait le numéro du signal
  - `sa_sigaction()` a plus d'infos

Primitive à utiliser pour installer un nouveau gestionnaire

```
#include <signal.h>
int sigaction(int sig, const struct sigaction *newaction, struct sigaction *oldaction);
```

- Voir `man sigaction` pour les détails

Autre interface existante : ANSI C

- Peut-être un peu plus simple, mais bien moins puissante et pas thread-safe

Exemple 1 : traitement d'une interruption du clavier

- Par défaut, Ctrl-C tue le processus ; pour survivre : il suffit de redéfinir le gestionnaire de SIGINT

test-int.c

```
#include <signal.h>
void handler(int sig) {                    /* nouveau gestionnaire */
   printf("signal SIGINT reçu !\n");
   exit(0);
}
int main() {
   struct sigaction nvt,old;
   memset(&nvt, 0, sizeof(nvt));
   nvt.sa_handler = &handler;
   sigaction(SIGINT, &nvt, &old);     /*installe le gestionnaire*/
   pause ();                                /* attend un signal */
   printf("Ceci n'est jamais affiché.\n");;
}
```

```
$ ./test-int
[frappe de CTRL-C]
signal SIGINT reçu !
$
```

Exercice : modifier ce programme pour qu'il continue après un CTRL-C

Note : il doit rester interruptible, *i.e.* on doit pouvoir le tuer d'un CTRL-C de plus

```
#include ¡signal.h¿

void handler(int sig) {
  printf(”signal SIGINT reçu !\n”);
}
```

```
int main() {
  struct sigaction nvt,old;
  nvt.sa_handler = handler;   sigaction(SIGINT, &nvt, &old);
  pause ();
  nvt.sa_handler = SIG_DFL; sigaction(SIGINT, &nvt, &old);
  while (1);  }
```

Exemple 2 : temporisation

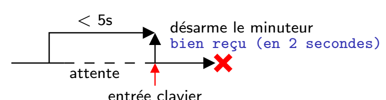
`unsigned int alarm(unsigned int nb_sec)`

- `nb_sec` >0 : demande l'envoi de SIGALRM après environ `nb_sec` secondes
- `nb_sec` $=0$: annulation de toute demande précédente
- Retour : nombre de secondes restantes sur l'alarme précédente
- Attention, `sleep()` réalisé avec `alarm()` $\Rightarrow$ mélange dangereux

```c
#include <signal.h>
void handler(int sig) {
  printf("trop tard !\n");
  exit(1);
}
int main() {
  struct sigaction nvt,old;
  int reponse,restant;
  memset(&nvt, 0, sizeof(nvt));
  nvt.sa_handler = handler;
  sigaction(SIGALRM, &nvt, &old);
```

```c
  printf("Entrez un nombre avant 5 sec:");
  alarm(5);
  scanf("%d", &reponse);
  restant = alarm(0);
  printf("bien reçu (en %d secondes)\n",
         5 - restant);
  exit (0);
}
```

5s
trop tard !
attente entrée



Exemple 3 : synchronisation père-fils

- Fin ou suspension d'un processus ⇒ SIGCHLD automatique à son père
- Traitement par défaut : ignorer ce signal

- Application : `wait()` pour éviter les zombies mangeurs de ressources
  C'est ce que fait le processus `init` (celui dont le pid est 1)

```
#include <signal.h>
void handler(int sig) {                         /* nouveau gestionnaire */
  pid_t pid;
  int statut;
  pid = waitpid(-1, &statut, 0);   /* attend un fils quelconque */
  return;
}
int main() {
  struct sigaction nvt,old;
  memset(&nvt, 0, sizeof(nvt));
  nvt.sa_handler = &handler;
  sigaction(SIGCHLD, &nvt, &old);   /* installe le gestionnaire */
  ... <création d'un certain nombre de fils> ...
  exit (0);
}
```

Résumé de la fin du deuxième chapitre

- Quelques définitions
  - pgid : groupe de processus ; un par processus lancé par shell (et tous ses fils)
  - Travail d'avant-plan, d'arrière-plan : lancé avec ou sans &
- Communication par signaux
  - Envoyer un signal $<NOM>$ à $<victime>$:
    - Langage commande : kill -NOM victime
    - API : kill(pid_t victime, int sig)
      $victime > 0$: numéro du pid visé
      $victime = 0$: tous les process de ce groupe
      $victime = -1$: tous les process accessibles
      $victime < -1$: tous les process du groupe $abs(victime)$
  - Changer le gestionnaire d'un signal : sigaction (en POSIX)
  - Exemples de signaux
    - SIGINT, SIGSTOP : Interaction avec le travail de premier plan (ctrl-c, ctrl-z)
    - SIGTERM, SIGKILL : demande de terminaison, fin brutale
    - SIGALARM : temporisation
    - SIGCHLD : relations père-fils

Troisième chapitre

Fichiers et entrées/sorties

- Fichiers et systèmes de fichiers
  - Introduction
  - Désignation des fichiers
- Interface d'utilisation des fichiers
  - Interface en langage de commande
  - Interface de programmation
  - Flots d'entrée/sortie et tubes
- Réalisation des fichiers
  - Implantation des fichiers
  - Manipulation des répertoires
  - Protection des fichiers
- Conclusion

Fichiers

Définitions

- Fichier : un ensemble d'informations groupées pour conservation et utilisation
- Système de gestion de fichiers (SGF) : partie de l'OS conservant les fichiers et permettant d'y accéder

Fonctions d'un SGF

- Conservation permanente des fichiers (*ie*, après la fin des processus, sur disque)
- Organisation logique et désignation des fichiers
- Partage et protection des fichiers
- Réalisation des fonctions d'accès aux fichiers

Unix et les fichiers

Les fichiers jouent un rôle central dans Unix

- Support des données
- Support des programmes exécutables

- Communication avec l'utilisateur : fichiers de config, `stdin`, `stdout`
- Communication entre processus : sockets, tubes, etc.

- Interface du noyau : `/proc`
- Interface avec le matériel : périphériques dans `/dev`

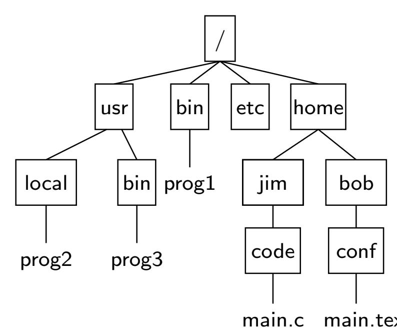
Désignation des fichiers

Désignation symbolique (nommage) : Organisation hiérarchique

- Noeuds intermédiaires : répertoires (*directory* – ce sont aussi des fichiers)
- Noeuds terminaux : fichiers simples
- Nom absolu d'un fichier : le chemin d'accès depuis la racine

Exemples de chemins absolus :

```
/
/bin
/usr/local/bin/prog
/home/bob/conf/main.tex
/home/jim/code/main.c
```



Raccourcis pour simplifier la désignation

Noms relatifs au répertoire courant

- Depuis `/home/bob`, `conf/main.tex` = `/home/bob/conf/main.tex`

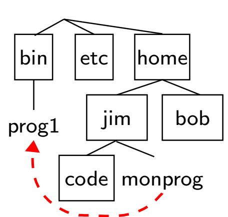
Abréviations

- Répertoire père : depuis `/home/bob`, `../jim/code` = `/home/jim/code`
- Répertoire courant : depuis `/bin`, `./prog1` = `/bin/prog1`
- Depuis n'importe où, ~bob/ = /home/bob/ et ~/ = /home/<moi>/

Liens symboliques

Depuis `/home/jim`

- Création du lien : `ln -s` *cible nom_du_lien*
  Exemple : `ln -s /bin/prog1 monprog`
- `/home/jim/prog1` désigne `/bin/prog1`
- Si la cible est supprimée, le lien devient invalide



Liens durs : `ln cible nom`

- Comme un lien symbolique, mais copies indifférenciables (ok après `rm cible`)
- Interdit pour les répertoires (cohérence de l'arbre)

Règles de recherche des exécutables

- Taper le chemin complet des exécutable (/bin/ls) est lourd
- ⇒ on tape le nom sans le chemin et le shell cherche
- Variable environnement PATH : liste de répertoires à examiner successivement
  `/usr/local/bin:/usr/local/sbin:/sbin:/usr/sbin:/bin:/usr/bin:/usr/bin/X11`
- Commande `which` indique quelle version est utilisée

Exercice : Comment exécuter un script nommé `gcc` dans le répertoire courant ?

- Solution 1 : export PATH=". :$PATH" ; gcc <bla>
- Solution 2 : ./gcc <bla>

Utilisations courantes des fichiers

- Unix : fichiers = suite d'octets sans structure interprétée par utilisateur
- Windows : différencie fichiers textes (où \n est modifié) des fichiers binaires

Programmes exécutables

- Commandes du système ou programmes créés par un utilisateur
- Exemple : `gcc -o test test.c ; ./test`
  Produit programme exécutable dans fichier `test` ; exécute le programme test
- Question : pourquoi `./test` ? (car test est un binaire classique : `if test $n -le 0;`)

Fichiers de données

- Documents, images, programmes sources, etc.
- Convention : le suffixe indique la nature du contenu
  Exemples : .c (programme C), .o (binaire translatable, cf. plus loin), .h (entête C), .gif (un format d'images), .pdf (Portable Document Format), *etc.*
  Remarque : ne pas faire une confiance aveugle à l'extension (*cf.* `man file`)

Fichiers temporaires servant pour la communication

- Ne pas oublier de les supprimer après usage
- On peut aussi utiliser des tubes (*cf.* plus loin)

Troisième chapitre

Fichiers et entrées/sorties

- Fichiers et systèmes de fichiers
  - Introduction
  - Désignation des fichiers
- Interface d'utilisation des fichiers
  - Interface en langage de commande
  - Interface de programmation
  - Flots d'entrée/sortie et tubes
- Réalisation des fichiers
  - Implantation des fichiers
  - Manipulation des répertoires
  - Protection des fichiers
- Conclusion

Utilisation des fichiers dans le langage de commande

Créer un fichier

- Souvent créés par applications (éditeur, compilateur, etc), pas par shell
- On peut néanmoins créer explicitement un fichier (cf plus loin, avec flots)

Quelques commandes utiles

créer un répertoire `mkdir <nom du répertoire>` (initialement vide)

détruire un fichier `rm <nom du fichier>` (option `-i` : interactif)

détruire un répertoire `rmdir <rep>` s'il est vide ; `rm -r <rep>` sinon

chemin absolu du répertoire courant `pwd` (*print working directory*)

contenu du répertoire courant `ls` (*list* – `ls -l` plus complet)

Expansion des noms de fichiers (*globbing*)

- * désigne n'importe quelle chaîne de caractères :
  `rm *.o` : détruit tous les fichiers dont le nom finit par .o
  `ls *.c` : donne la liste de tous les fichiers dont le nom finit par .c

Interface de programmation POSIX des fichiers

Interface système

- Dans l'interface de programmation, un fichier est représenté par descripteur
  Les descripteurs sont de (petits) entiers (indice dans des tables du noyau)
- Il faut **ouvrir** un fichier avant usage :
  `fd = open("/home/toto/fich", O_RDONLY, 0);`
  - Ici, ouverture en lecture seule (écriture interdite)
  - Autres modes : O_RDWRITE, O_WRONLY
  - fd stocke le descripteur alloué par le système (ou −1 si erreur)
  - Fichier créé s'il n'existe pas (*cf.* aussi creat(2))
  - `man 2 open` pour plus de détails
- Il faut **fermer** les fichiers après usage :
  `close (fd)`
  - Descripteur `fd` plus utilisable ; le système peut réallouer ce numéro

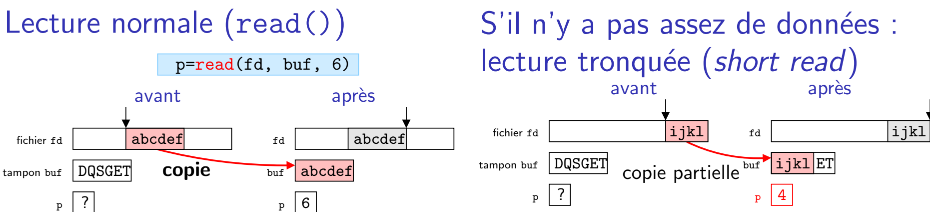
Interface standard

- Il existe une autre interface, plus simple (on y revient)

Interface de programmation POSIX : `read()`

L'objet fichier

- Chaque fichier a un pointeur courant : tête de lecture/écriture logique
- Les lectures et écritures le déplacent implicitement
- On peut le déplacer explicitement (avec `lseek()` – voir plus loin)

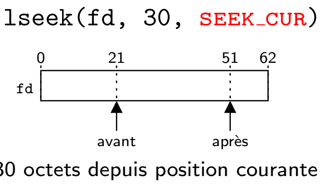


Interface de programmation POSIX : `lseek()`

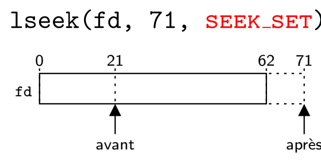
Objectif

- Permet de déplacer le pointeur de fichier explicitement

Exemples



+30 octets depuis position courante

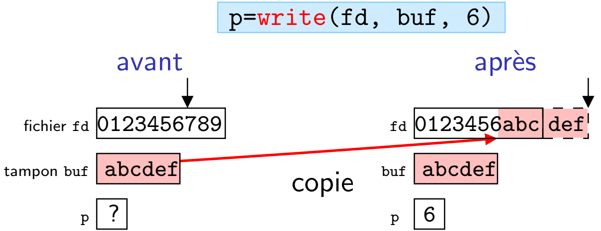


place le pointeur à la position 71

- Le pointeur peut être placé après la fin du fichier

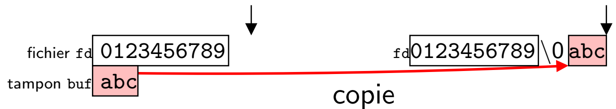
Interface de programmation POSIX : `write()`

- Écriture dans les conditions normales :



  Le fichier a été rallongé

- Si un `lseek()` a déplacé le pointeur après la fin, remplissage avec des zéros



- Possibilité d'écriture tronquée (*short-write*) si le disque est plein, ou si descripteur est un tube ou une socket (*cf.* plus loin)

Différentes interfaces d'usage des fichiers

Interface POSIX système

- Appels systèmes `open`, `read`, `write`, `lseek` et `close`.
- Utilisation délicate : lecture/écriture tronquée, traitement d'erreur, *etc.*
- Meilleures performances après réglages précis

Bibliothèque C standard

- Fonctions `fopen`, `fscanf`, `fprintf` et `fclose`.
- Plus haut niveau (utilisation des formats d'affichage)
- Meilleures performances sans effort (POSIX : perfs au tournevis)
- Un syscall $\approx$ 1000 instructions $\Rightarrow$ écritures dans un tampon pour les grouper

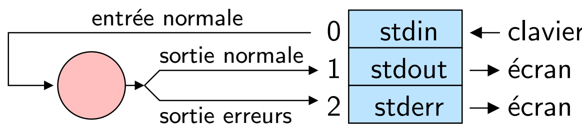
```
#include <stdio.h>

FILE *fd = fopen("mon_fichier","w");
if (fd != NULL) {
  fprintf(fd,"Résultat %d\n", entier);
  fprintf(fd,"un autre %f\n", reel);
  fclose(fd);
}
```

```
#include <stdio.h>

FILE *fd = fopen("mon_fichier","w");
if (fd != NULL) {
  char toto[50];
  fscanf(fd,"%s%d",toto, &entier);
  fscanf(fd,"%s%s%f",toto,toto, &reel);
  fclose(fd);
}
```

Fichiers et flots d'entrée sortie

Sous Unix, tout est un fichier

- Les périphériques sont représentés par des fichiers dans `/dev`
- Tout processus est créé avec des flots d'entrée/sortie standard :
  - stdin : entrée standard (connecté au terminal)
  - stdout : sortie standard (connecté à l'écran)
  - stderr : sortie d'erreur (connecté à l'écran)
- Ces flots sont associés aux descripteurs 0, 1 et 2
- Ils peuvent être fermés ou redirigés vers des ≪vrais≫ fichiers



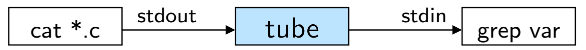
Flots et interface de commande

Rediriger les flots en langage de commande : < et >

```
cat fich # écrit le contenu de fich sur la sortie standard (l'affiche à l'écran)
cat fich > fich1 # copie fich dans fich1 (qui est créé s'il n'existe pas)
./prog1 < entrée # exécute prog en écrivant le contenu de entrée sur son entrée standard
```

Tubes (*pipes*) : moyen de communication inter-processus

- Branche la sortie standard d'un processus sur l'entrée standard d'un autre
- Exemple : `cat *.c | grep var`
  - crée deux processus : `cat *.c` et `grep var`
  - connecte la sortie du premier à l'entrée du second à travers un tube



Exercice : que fait la commande suivante ?

`cat f1 f2 f3 | grep toto | wc -l > resultat`

Compte les occurences de toto dans les trois fichiers, et écrit le décompte dans le fichier resultat

Interface de programmation des tubes

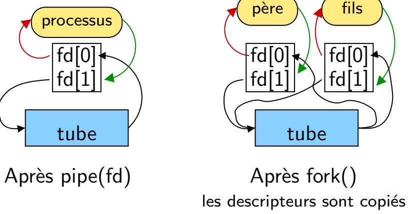
Appel système pipe()

- Crée un tube : deux descripteurs (l'entrée et la sortie du tube)

```
int fd[2];
pipe(fd);
```

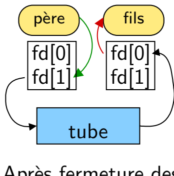
- Ce qui est écrit sur l'entrée (`fd[1]`) est disponible sur la sortie (`fd[0]`)
  par défaut, cela permet de se parler à soi-même :)
  Mnemotechnique : On lit sur 0 (stdin), et on écrit sur 1 (stdout)

- Application classique : communication père–fils



Après pipe(fd)

Après fork()
les descripteurs sont copiés



Après fermeture des descripteurs inutiles

Programmation d'un tube père/fils

```
#define BUFSIZE 10
int main(void) {
  char bufin[BUFSIZE] = "empty";
  char bufout[ ] = "hello";
  pid_t childpid;
  int fd[2];
  pipe(fd);
  if (fork() > 0) { /* père */
    close(fd[0]);
    write(fd[1], bufout, strlen(bufout)+1);
  } else {          /* fils */
    close(fd[1]);
    read(fd[0], bufin, BUFSIZE);
  }
  printf("[%d]: bufin = %.*s, bufout = %s\n",
         getpid(), BUFSIZE, bufin, bufout);
  return 0;
}
```



```
$ ./parentwritepipe
[29196]: bufin=empty, bufout=hello
[29197]: bufin=hello, bufout=hello
$
```

Exercice : modifier le programme pour tenir compte des lectures/écritures tronquées

```
int todo = strlen(bufout)+1, done;
char *p=bufout;
while (todo) {
  done = write(fd[1],p,todo);
  todo -= done; p += done;    }
```

```
int done=0; bufin[0]='a';
while (bufin[done]) {
  done += read(fd[0],
               bufin+done,
               BUFSIZE-done);   }
```

Capacité d'un tube

`write(fd, &BUFF, TAILLE)` : écriture d'au plus TAILLE caractères

- S'il n'y a plus de lecteur :
  - Écriture inutile : on ne peut pas ajouter de lecteurs après la création du tube
  - Signal SIGPIPE envoyé à l'écrivain (mortel par défaut)
- Sinon si le tube n'est pas plein : Écriture atomique
- Sinon : Écrivain bloqué tant que tous les caractères n'ont pas été écrits (tant qu'un lecteur n'a pas consommé certains caractères)

`read(fd, &BUFF, TAILLE)` : lecture d'au plus TAILLE caractères

- Si le tube contient n caractères : Lecture de $min(n, TAILLE)$ caractères.
- Sinon s'il n'il a plus d'écrivains : Fin de fichier ; `read` renvoie 0.
- Sinon : Lecteur bloqué jusqu'à ce que le tube ne soit plus vide (jusqu'à ce qu'un écrivain ait produit suffisamment de caractères)

Parenthèse : Opérations non-bloquantes

- Drapeau `O_NONBLOCK` ⇒ read/write ne bloquent plus jamais :
- Retournent -1 (et errno=`EAGAIN`) si elles auraient dû bloquer
- `fnctl(fd, F_SETFL, fcntl(fd[1], F_GETFL) | O_NONBLOCK);`

Tubes nommés (FIFOs)

- Problème des tubes : seulement entre descendants car passage par clonage
- Solution : tubes nommés (même principe, mais nom de fichier symbolique)

```
#include <sys.types.h>
#include <sys/stat.h>
int mkfifo(char *nom, mode_t mode);
```

  - renvoie 0 si OK, -1 si erreur
  - mode est numérique (on y revient)

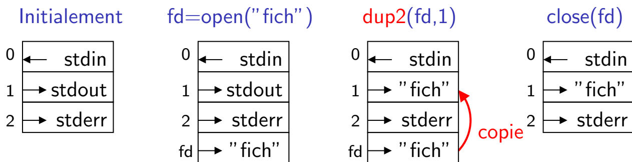
- Après création, un processus l'ouvre en lecture, l'autre en écriture
- Il faut connecter les deux extrémités avant usage (bloquant sinon)

- la *commande* `mkfifo`(1) existe aussi

Copie de descripteurs : dup() et dup2()

`ls > fich`

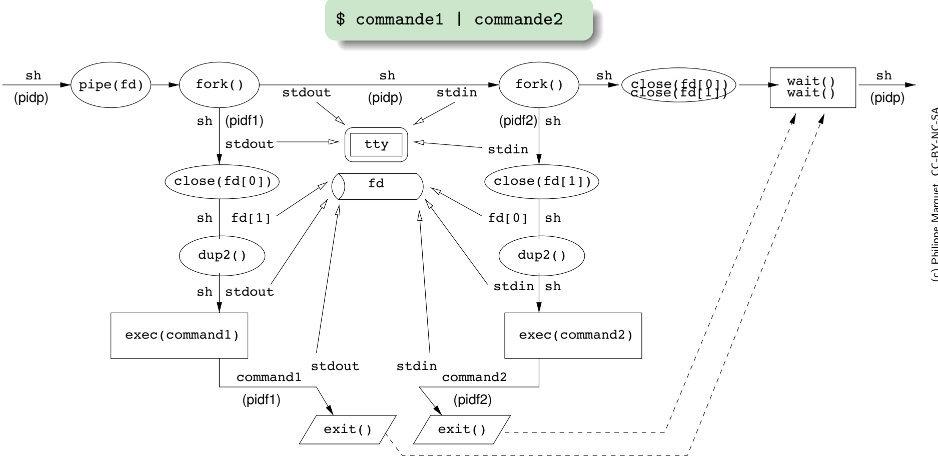
- Cette commande inscrit dans `fich` ce que `ls` aurait écrit.
- Comment ça marche ?
  On utilise les appels systèmes `dup()` et `dup2()` pour copier un descripteur :



- Appel système dup(fd) duplique `fd` dans le premier descripteur disponible
- Appel système dup2(fd1, fd2) recopie le descripteur `fd1` dans `fd2`
  on dit "dup to", duplicate to somewhere

L’exemple du shell

Création d’un tube entre deux commandes



(c) Philippe Marquet, CC-BY-NC-SA.

- (Un peu touffu, mais pas si compliqué à la réflexion :)
- Si on oublie des `close()`, les lecteurs ne s’arrêtent pas (reste des écrivains)

Projection mémoire

Motivation

- Adresser le fichier dans l'espace d'adressage virtuel
  Pratique pour sérialiser des données complexes de mémoire vers disque
- Le fichier n'est pas lu/écrit au chargement, mais à la demande

Réalisation

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

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
```

- `addr` : où le mettre, quand on sait. NULL très souvent
- `prot` : protection (PROT_EXEC, PROT_READ, PROT_WRITE, PROT_NONE)
- `flags` : visibilité (MAP_SHARED entre processus ; MAP_PRIVATE copy-on-write, etc)
- `fd`, offset : Quel fichier, quelle partie projeter.
- Retourne l'adresse (virtuelle) du début du block

Troisième chapitre

Fichiers et entrées/sorties

- Fichiers et systèmes de fichiers
  - Introduction
  - Désignation des fichiers
- Interface d'utilisation des fichiers
  - Interface en langage de commande
  - Interface de programmation
  - Flots d'entrée/sortie et tubes
- Réalisation des fichiers
  - Implantation des fichiers
  - Manipulation des répertoires
  - Protection des fichiers
- Conclusion

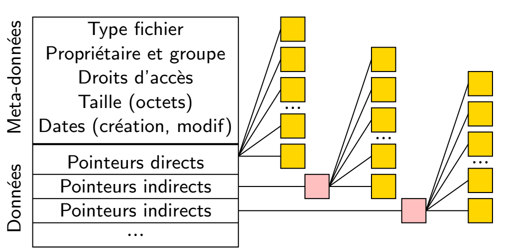
Représentation physique et logique d'un fichier

Représentation physique d'un fichier

- Le disque est découpé en *clusters* (taille classique : 4ko)
- Les fichiers sont écrits dans un ou plusieurs *clusters*
- Un seul fichier par *cluster*

Structure de données pour la gestion interne des fichiers

- Chaque fichier sur disque est décrit par un **inode**
  Structure d'un inode :



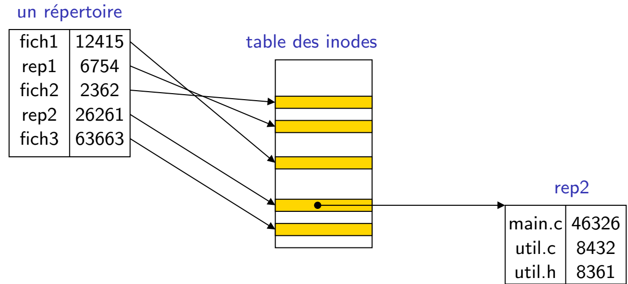
- Pointeurs et tables d'indirection contiennent adresses de *clusters*
- ≈ 10 blocs adressés directement
  ⇒ accès efficace petits fichiers (les plus nombreux)
- `stat`(2) : accès à ces info

Noms symboliques et inodes

- Les inodes sont rangés dans une table au début de la partition
- On peut accéder aux fichiers en donnant leur inode (*cf.* `ls -i` pour le trouver)
- Les humains préfèrent les noms symboliques aux numéros d'identification

⇒ notion de répertoire, sous forme d'arborescence (chainage avec racine)

- Les répertoires sont stockés sous forme de fichiers ≪normaux≫
- Chaque entrée d'un répertoire associe nom symbolique ↔ numéro d'inode



Manipulation de répertoires

Appels systèmes opendir, readdir et closedir

- Équivalent de open, read et close pour les répertoires

EXEMPLE : Implémentation de `ls -i .`

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>

int main() {
  DIR *dirp;
  struct dirent *dp;

  dirp = opendir(".");
  while ((dp = readdir(dirp)) != NULL) {
    printf ("%s\t%d\n", dp->d_name, dp->d_ino);
  }
  closedir(dirp);
}
```

- On pourrait compléter cette implémentation avec stat(2)

Protection des fichiers : généralités

Définition (générale) de la sécurité

- confidentialité : informations accessibles aux seuls usagers autorisés
- intégrité : pas de modifications indésirées
- contrôle d'accès : qui a le droit de faire quoi
- authentification : garantie qu'un usager est bien celui qu'il prétend être

Comment assurer la sécurité

- Définition d'un ensemble de règles (politiques de sécurité) spécifiant la sécurité d'une organisation ou d'une installation informatique
- Mise en place de mécanismes de protection pour faire respecter ces règles

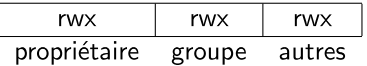
Règles d'éthique

- Protéger ses informations confidentielles (comme les projets et TP notés !)
- Ne pas tenter de contourner les mécanismes de protection (c'est la loi)
- Règles de bon usage avant tout :
  La possibilité technique de lire un fichier ne donne pas le droit de le faire

Protection des fichiers sous Unix

Sécurité des fichiers dans Unix

- Trois types d'opérations sur les fichiers : lire (r), écrire (w), exécuter (x)
- Trois classes d'utilisateurs vis à vis d'un fichier :
  propriétaire du fichier ; membres de son groupe ; les autres



  Granularité plus fine avec les Access Control List (pas étudiés ici)
- Pour les répertoires, `r` = `ls`, `w` = créer des éléments et `x` = `cd`.
- `ls -l` pour consulter les droits ; `chmod` pour les modifier

Mécanisme de délégation

- Problème : programme dont l'exécution nécessite des droits que n'ont pas les usagers potentiels (exemple : gestionnaire d'impression, d'affichage)
- Solution (setuid ou setgid) : ce programme s'exécute toujours sous l'identité du propriétaire du fichier ; identité utilisateur momentanément modifiée
  identité réelle (celle de départ) vs identité effective (celle après setuid)

Résumé du troisième chapitre

- Désignation des fichiers
  - Chemin relatif *vs.* chemin absolu :
  - $PATH :
- Utilisation des fichiers
  - Interface de bas niveau : open, read, write, close
    Problèmes : I/O tronquée, perfs par manque de tampon
  - Interface standard : fopen, fprintf, fscanf, fclose
  - Trois descripteurs par défaut :
  - Rediriger les flots en shell : $<$, $>$ et $|$
  - Tubes, tubes nommés et redirection en C : pipe(), mkfifo(), dup() et dup2()
- Réalisation et manipulation des fichiers et répertoires
  - Notion d'inode :
  - Manipulation des répertoires : opendir, readdir, closedir
- Quelques notions et rappels de protection des fichiers

Quatrième chapitre

Exécution des programmes

- Schémas d'exécution
- Interprétation
  - Principe de fonctionnement d'un interpréteur
  - Exemple d'interpréteur : le shell
- Compilation et édition de liens
  - Principes de la compilation
  - Liaison, éditeur de liens et définitions multiples en C
  - Bibliothèques statiques et dynamiques
- Conclusion

Schémas d'exécution d'un programme

- Programme (impératif) : suite d'instructions, d'actions successives
- Exécution : faire réaliser ces actions par une machine dans un état initial pour l'amener à un état final

- Langage [de la] machine : suite de 0 et de 1, difficile pour humains
- Langues humaines ambiguës et complexes, inaccessibles aux machines

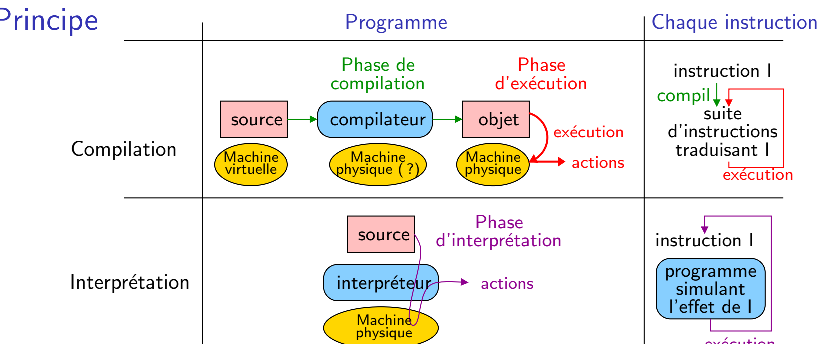
$\Rightarrow$ les humains utilisent des langages informatiques traduits en langage machine

Convertir les langages informatiques en langage machine

- Compilation : conversion à priori, génération d'un fichier *binaire exécutable* depuis un fichier *source* par un **compilateur**
- Interprétation : programme auxiliaire (**interpréteur**) traduit au fur et à mesure interpréteur $\approx$ simulateur de machine virtuelle ayant un autre langage machine
- Mixte : compilation pour une machine intermédiaire
  $+$ interprétation par une *machine virtuelle*

Compilation et interprétation

Principe



Exemples

- Compilation : C, C++, Fortran, Ada, assembleur
- Interprétation : shell Unix, Tcltk, PostScript, langage machine
- Autre : Lisp (les deux), Java/Scala (machine virtuelle), Python (interprétation ou machine virtuelle), Ocaml (interprétation, compilation ou machine virtuelle)
- On peut faire de la compilation croisée voire pire

Compilation et interprétation : comparaison

Compilation

☺ Efficacité

- Génère du code machine natif
- Ce code peut être optimisé

☹ Mise au point

- Lien erreur $\leftrightarrow$ source complexe

☹ Cycle de développement

- cycle complet à chaque modification : compilation, édition de liens, exécution

Interprétation

☹ Efficacité

- 10 à 100 fois plus lent
- appel de sous-programmes
- pas de gain sur les boucles

☺ Mise au point

- Lien instruction $\leftrightarrow$ exécution trivial
- Trace et observation simples

☺ Cycle de développement

- cycle très court : modifier et réexécuter

- L'interprétation regagne de l'intérêt avec la puissance des machines modernes
- On peut parfois (eclipse/Visual) recharger à chaud du code compilé

Schéma mixte d'exécution

Objectifs

- Combiner les avantages de la compilation et de l'interprétation
- Gagner en portabilité sans trop perdre en performances

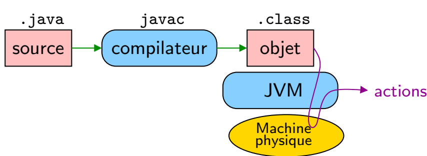
Principe

- Définir une machine virtuelle (qui n'existe donc pas)
- Écrire un simulateur de cette machine virtuelle
- Écrire un compilateur du langage source vers le langage machine virtuel

Exemple : Java inventé pour être (**le** langage d'internet)

- Programmes (applets) téléchargés doivent fonctionner sur toutes les machines

- Phase de compilation et objets indépendents de machine physique
- Portage sur une nouvelle machine : écrire une JVM pour elle



Quatrième chapitre

Exécution des programmes

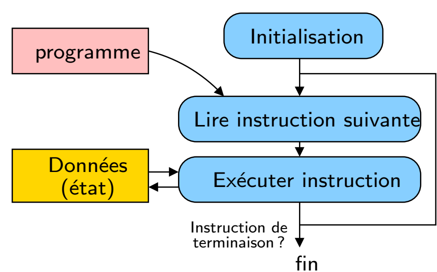
- Schémas d'exécution
- Interprétation
  - Principe de fonctionnement d'un interpréteur
  - Exemple d'interpréteur : le shell
- Compilation et édition de liens
  - Principes de la compilation
  - Liaison, éditeur de liens et définitions multiples en C
  - Bibliothèques statiques et dynamiques
- Conclusion

Principe de fonctionnement d'un interpréteur

Définition de la machine virtuelle

- Éléments du ≪pseudo-processeur≫ (analogie avec processeur physique)
  - pseudo-registres : zones de mémoire réservées
  - pseudo-instructions : réalisées par une bibliothèque de programmes

- Structures d'exécution
  - allocation des variables
  - pile d'exécution

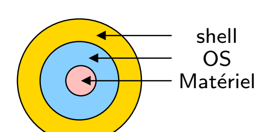


Cycle d'interprétation

- Analogie avec cycle processeur
- Pseudo-compteur ordinal (PC)

Exemple d'interpréteur : le shell

Définition



- Programme interface entre OS et utilisateur interactif
  Interpréteur le langage commande (→ appels systèmes)
- Mot à mot, shell=coquille, car ça ≪englobe≫ l'OS
- Il existe plusieurs shells (sh, csh, tcsh, bash, zsh, *etc.*)

Fonctions principales d'un shell

- Interpréteur commandes des usagers ; accès aux fonctions de l'OS
  - Création et exécution de processus
  - Accès aux fichiers et entrées / sorties
  - Utilisation d'autres outils (éditeur, compilateur, navigateur, *etc.*)
- Gestionnaire de tâches : travaux en mode interactif ou en tâche de fond
- Personnaliser l'environnement de travail : variables d'environnement
- *Scripting :* programmation en langage de commande (if, while, *etc.*)

Quatrième chapitre

Exécution des programmes

- Schémas d'exécution
- Interprétation
  - Principe de fonctionnement d'un interpréteur
  - Exemple d'interpréteur : le shell
- Compilation et édition de liens
  - Principes de la compilation
  - Liaison, éditeur de liens et définitions multiples en C
  - Bibliothèques statiques et dynamiques
- Conclusion

Composition de programmes

Les programmes ne s'exécutent jamais «seuls»

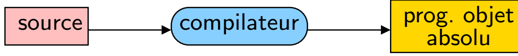
- Applications modernes = assemblage de modules (conception et évolution ☺)
- Applications modernes utilisent des «bibliothèques» de fonctions
- *Cf.* programmation objet et prolongement «par composants» (en PAR, 3A)

Problème : la liaison

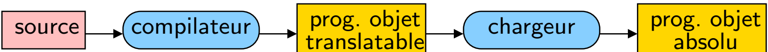
- Un programme P appelle une procédure `proc` définie dans Q
- Appel de routine en assembleur : `jump` à une adresse donnée
- Question : quelle adresse associer à `proc` dans le code machine de P ?
- Réponse : on sait pas tant que le code machine de Q est inconnu (pourquoi ?)
- Solution : Dans P, liaison entre `proc` et son adresse faite après compilation
  Édition de liens

Cycle de vie d'un programme compilé

- Compilation en programme objet absolu
  - Adresses (des routines) en mémoire fixées

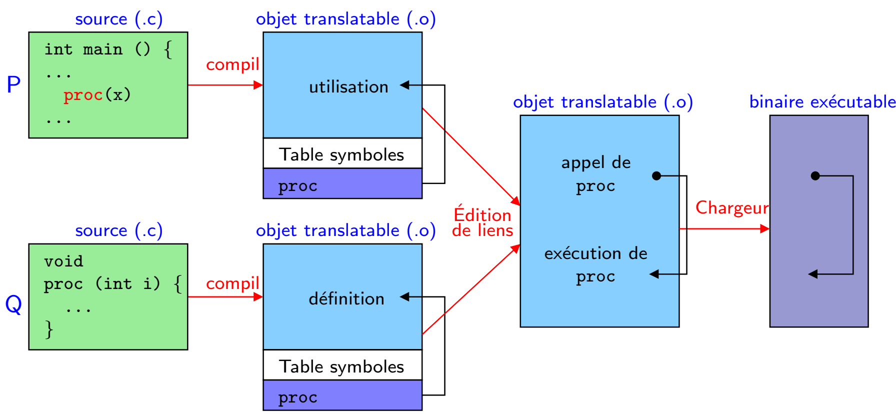


- Compilation en programme objet translatable
  - Adresses définies à une translation près
  - ☺ On peut donc regrouper plusieurs objets translatables dans un seul gros
  - ☹ Ce n'est pas exécutable (le CPU ne translate pas)
  - ⇒ chargeur convertit code translatable (combinable) en code absolu (exécutable)



- Exemples
  - `gcc -c prog.c` ⇒ `prog.o` = programme objet translatable
  - `gcc -o prog prog.c` ⇒ `prog` programme objet absolu (compilo + chargeur)

Édition de liens



- L'éditeur de lien et le chargeur sont souvent regroupé dans le même utilitaire : `ld`

Exemples de commandes de compilation

- La commande gcc invoque compilateur, éditeur de liens et chargeur

`gcc prog.c`	→	`a.out`	= binaire exécutable
`gcc -o prog prog.c`	→	`prog`	= binaire exécutable
`gcc -c prog.c`	→	`prog.o`	= binaire translatable

- Si un programme est composé `prog1.c` et `prog2.c`, alors :
  `gcc prog1.c prog2.c` → binaire exécutable complet dans `a.out`
  `gcc -c prog1.c prog2.c` → `prog1.o` et `prog2.o` = 2 objets translatables
  `gcc -o prog prog.o prog1.o` → `prog` depuis objets translatables

Quelques problèmes de l'édition de liens

Théorie relativement simple...

- Définition manquante

```
...
proc(x);
...
/* pas de définition de proc */
```

```
$ gcc -o prog prog1.c prog2.c prog3.c
/tmp/cckdgmLh.o(.text+0x19):
In function ‘main’:
undefined reference to ‘proc’
collect2: ld returned 1 exit status
$
```

- Définitions multiples

```
...
proc(x);
...
int proc; /* déf. comme entier */
...
void proc(int x) {
... /* déf. comme fonction */
}
```

```
$ gcc -o prog prog1.c prog2.c prog3.c
$
```

- Pas de message d'erreur !
- Que se passe-t-il ?

... pratique souvent déroutante

Définitions multiples en C

Deux catégories de symboles externes

- Symboles «externes» : visible de l'éditeur de liens (hors de toute fonction)
- Symbole fort :
  - définition de procédures
  - variables globales initialisée
- Symbole faible :
  - déclaration de procédures en avance (sans le corps de fonction)
  - variables globales non-initialisée

Règles de l'éditeur de liens pour les définitions multiples

- deux symboles forts : interdit ! (erreur détectée)
- un symbole fort et plusieurs faibles : le fort est retenu
- plusieurs symboles faibles : choix aléatoire

Pièges des définitions multiples

Module A	Module B	
`int proc(int i) {`	`int proc(int i) {`	Deux symboles forts : interdit
`int x;` `int p1(int i) {`	`int x;` `int p2(int i) {`	Deux symboles faibles désignent le même objet

`int x; /* 4o */`
`int y; /* 4o */`
`int p1(int i) {`
	`double x; /* 8o */`
	`int p2(int i) {`
		Deux symboles faibles. Modifier B.x écrase A.y ! !

`int x=3;`
`int y;`
`int p1(int i) {`
	`double x;`
	`int p2(int i) {`
		A.x fort ; B.x faible ; Modifier B.x écrase encore A.y ! !

- Contrôle de types strict entre modules pas obligatoire en C

⇒ Déclaration explicite des références externes nécessaire

Exercice : que se passe-t-il dans l'exemple précédent (deux pages avant) ?

int proc = symbole faible ; void proc(int x) = symbole fort, c'est lui qui est choisi (mais comportement étrange quand on change int proc)

Déclarations des variables externes

- Règle 1 : utiliser des entête (.h) pour unifier les déclarations
- Règle 2 : éviter les variables globales autant que possible !
  - déclarer `static` toute variable hors des fonctions (visibilité = fichier courant)
- Règle 3 : bon usage des globales (si nécessaire)
  - chaque globale déclarée fortement dans un module M
  - `extern` dans tous les autres (avec son type)
  - Seulement consultation depuis l'extérieur, modification seulement depuis M
    Fonctions dans M pour la modifier depuis ailleurs, au besoin (*setters*)
- Règle 4 : bon usage des fonctions
  - déclaration explicite des fonctions externes (avec signature)
  - mot clé `extern` optionnel : la signature (ou prototype) suffit

Fichier un.c

```c
int x;
extern double y;
static int z;
int f(int i) {... }
static int g(int i) {... }
/* x: écriture
   y: lecture
   z: écriture
   f: accessible
   g: accessible */
```

Fichier deux.c

```c
extern int x;
double y;




/* x: lecture
   y: écriture
   z: non-référençable
   f: invisible
   g: non-référençable */
```

Fichier trois.c

```c
extern double y;
int f(int i);




/* x: invisible
   y: lecture
   z: non-référençable
   f: accessible
   g: non-référençable */
```

Exercices : (à compléter)

Module A

```
int f(int i) { ... }
```

Module B

```
static double f=4.5;
```

Que se passe-t-il ?

Pas de problème
f.B static ⇒ invisible d'ailleurs

Module A

```
void p2(void);
int main() {
  p2();
  return 0;
}
```

Module B

```
#include <stdio.h>

char main;

void p2() {
  printf("0x%x\n",
         main);
}
```

Ca marche (et affiche 0x55).
Link sans soucis : main.B est faible, il est écrasé
Execution : ref(main.B) = def(main.A), et 0x55 est l'adresse du main() à gauche

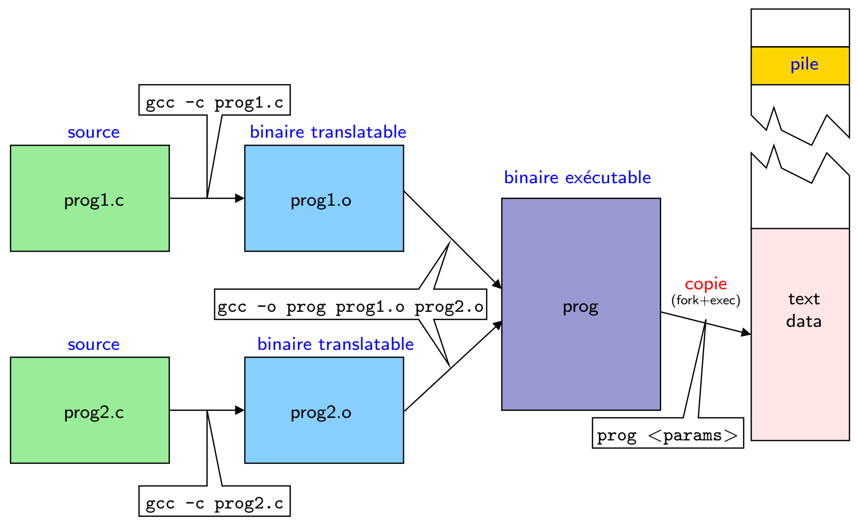
Exercice : pourquoi chaque programme doit contenir une fonction main() ?

car par convention, exec() passe le controle à une fonction de ce nom

Exercice : que se passe-t-il quand main() fait return ou exit()

main : la fonction qui a invoqué main() reprend le contrôle. Elle appelle _exit()
exit : _exit() aussi ; libc reprend le contrôle, termine le processus et prévient l'OS

Cycle de vie d'un programme : résumé



Quatrième chapitre

Exécution des programmes

- Schémas d'exécution
- Interprétation
  - Principe de fonctionnement d'un interpréteur
  - Exemple d'interpréteur : le shell
- Compilation et édition de liens
  - Principes de la compilation
  - Liaison, éditeur de liens et définitions multiples en C
  - Bibliothèques statiques et dynamiques
- Conclusion

Bibliothèques

- Bibliothèque = recueil de fonctions utilisées dans divers programmes
- Ne pas réinventer la roue ; réutilisation du code
- Exemples : `atoi`, `printf`, `malloc`, `random`, `strcmp`, `sin`, `cos`, `floor`, *etc.*

Divers moyens de rendre ce service :

- Réalisation par le compilateur :
  - Compilateur Pascal remplace appels fonctions standards par code correspondant
  - ☹ Changer le compilateur pour changer ce code
- Approche par binaire translatable :
  - Le code de toutes les fonctions standards placé dans un `/usr/lib/libc.o`
  - ☹ Plusieurs Mo à chaque fois, même si inutilisé
- Approche par collections de binaires translatables :
  - Code chaque fonction dans un binaire translatable différent
  - ☹ `gcc toto.c /usr/lib/printf.o /usr/lib/scanf.o ...` (un peu lourd)
- Approche par archives de binaires translatables (**bibliothèques statiques**) :
  - Concaténation de tous les .o dans un fichier, et l'éditeur de lien choisi

Bibliothèques statiques

- `/usr/lib/libc.a` : bibliothèque standard du langage C (printf, strcpy, *etc.*)
- `/usr/lib/libm.a` : bibliothèque mathématique (cos, sin, floor, *etc.*)

Utilisation :

- Passer bibliothèques utilisées à l'éditeur de lien (libc.a ajoutée par défaut)
  `gcc -o prog prog.c /usr/lib/libm.a`
- L'éditeur de liens ne copie que les «fonctions» effectivement utilisées
- Notation abrégée : -l<nom> équivalent à /usr/lib/lib<nom>.a
  Exemple : `gcc -o prog prog.c -lsocket` (inclut /usr/lib/libsocket.a)
- Chemin de recherche : Variable `LIBPATH`, ajout avec `gcc -L<rep>`

Format et création

- Archives de binaires translatables
- Manipulées par `ar`(1)
  `ar rcs libamoi.a fonct1.o fonct2.o fonct3.o`
- Remarque : historiquement, `tar` est une version particulière de `ar`

Dépendances entre bibliothèques statiques

- Difficulté lorsqu'une bibliothèque utilise une autre bibliothèque
- L'ordre des -ltoto lors de l'appel à l'éditeur de liens devient important car :
  - Éditeur de liens ne copie que les objets nécessaires, pas toute l'archive
  - Il ne parcourt ses arguments qu'une seule fois
- Règle : déclaration de fonction doit être placée après son premier usage

Exercice : quelle(s) commande(s) fonctionne(nt)/échoue(nt) ? Pourquoi ?

prog.c utilise machin() de libmachin.a, et machin() utilise truc() de libtruc.a

- `gcc -o prog prog.c -ltruc -lmachin` : référence à truc() indéfinie
- `gcc -o prog prog.c -lmachin -ltruc` : fonctionne

Exercice : que faire en cas de dépendances circulaires ?

faire figurer une bibliothèque deux fois pour casser le cycle

Bibliothèques dynamiques

- Défauts des bibliothèques statiques :
  - Code dupliqué dans chaque processus les utilisant
  - Liaison à la compilation (nouvelle version de bibliothèque ⇒ recompilation)
- Solution : Bibliothèques dynamiques :
  Partage du code entre applications et chargement dynamique
- Exemples : Windows : **DLL** (dynamically linkable library) ; Unix : **.so** (shared object)
- Bibliothèques partagées à plus d'un titre :
  sur disque (un seul .so) et en mémoire (physique, du moins)
- Chargement dynamique :
  - ☹ Impose une édition de lien au lancement (`ldd`(1) montre les dépendances)
  - ☺ Mise à jour des bibliothèques simplifiée (mais attention au DLHell)
    Versionner les bibliothèques (et même les symboles) n'est pas trivial
  - ☺ Mécanisme de base pour les greffons (*plugins*)
  - ☺ Même possible d'exécuter du code au chargement/déchargement !

```
void chargement(void) __attribute__((constructor)) {...}
void alafin(void) __attribute__((destructor)) {...}
```

Bibliothèques dynamiques

Construction

- `gcc -shared -fPIC -o libpart.so fonct1.c fonct2.c fonct3.c`
- `-fPIC` (compil.) : demande code repositionnable (Position Independent Code)
- `-shared` (éd. liens) : demande objet dynamique

Outils (voir les pages de man)

- ar : construire des archives/bibliothèques
- strings : voir les chaines affichables
- strip : supprimer la table des symboles
- nm : lister les symboles
- size : taille de chaque section d'un fichier elf
- readelf : afficher la structure complète et tous les entêtes d'un elf (nm+size)
- objdump : tout montrer (même le code desassemblé)
- ldd : les bibliothèques dynamiques dont l'objet dépend, et où elles sont prises

Chargement dynamique de greffons sous linux

Greffon *(plugin)* :

- Bout de code chargeable dans une application pour l'améliorer

→ Choisir l'implémentation d'une fonctionnalité donnée (affichage, encodage)

→ Ajouter des fonctionnalités (mais bien plus dur, nécessite API dynamique)

Compiler du code utilisant des greffons :

- `gcc -rdynamic -ldl` *autres options*

Exercice : Pourquoi ce -ldl ?

Pour charger une bibliothèque nommée dl (dynamic library)

Interface de programmation sous UNIX modernes (Linux et Mac OSX)

```
#include <dlfcn.h>
void *dlopen(const char *filename, int flag);                     /* charge un greffon */
void *dlsym(void *handle, const char *symbole); /* retourne pointeur vers ≪symbole≫ */
int   dlclose(void *handle);                                       /* ferme un greffon */
char *dlerror(void);            /* affiche la cause de la dernière erreur rencontrée */
```

Exemple de greffon sous linux

chargeur.c

```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <dlfcn.h>
int main(int argc, char * argv[]) {
  char path[256], *erreur = NULL;
  int (*mon_main)();
  void *module;
  /* Charge le module */
  module = dlopen(argv[1], RTLD_NOW);
  if (!module) {
    fprintf (stderr, "%s\n", dlerror());
    exit(1);
  }
  /* Retrouve la fonction d'entrée */
  mon_main = dlsym(module, "mon_main");
  erreur = dlerror();
  if (erreur != NULL) {
    perror(erreur);
    exit(1);
  }
  /* Appelle cette fonction */
  (*mon_main)();
  /* Ferme tout */
  dlclose(module);
  return 0;
}
```

module_un.c

```
int mon_main() {
  printf("Je suis le module UN.\n");
}
```

module_deux.c

```
int mon_main() {
  printf("Je suis le module DEUX.\n");
}
```

```
$ gcc -shared -fPIC -o un.so module_un.c
$ gcc -shared -fPIC -o deux.so module_deux.c
$ gcc -o chargeur chargeur.c -rdynamic -ldl
$ ./chargeur ./un.so
Je suis le module UN.
$ ./chargeur ./deux.so
Je suis le module DEUX.
$ ldd chargeur
  libdl.so.2 => /lib/i686/cmov/libdl.so.2 (0xb7f
  libc.so.6 => /lib/i686/cmov/libc.so.6 (0xb7e70
  /lib/ld-linux.so.2 (0xb7fd1000)
$ nm chargeur | grep mon_main
$
```

Bibliothèques dynamiques et portabilité

Chaque système a sa propre interface

- Solaris, Linux, MacOSX et BSD : dlopen (celle qu'on a vu, par SUN)
- HP-UX : shl_load
- Win{16,32,64} : LoadLibrary
- BeOS : load_add_on

libtool offre une interface portable

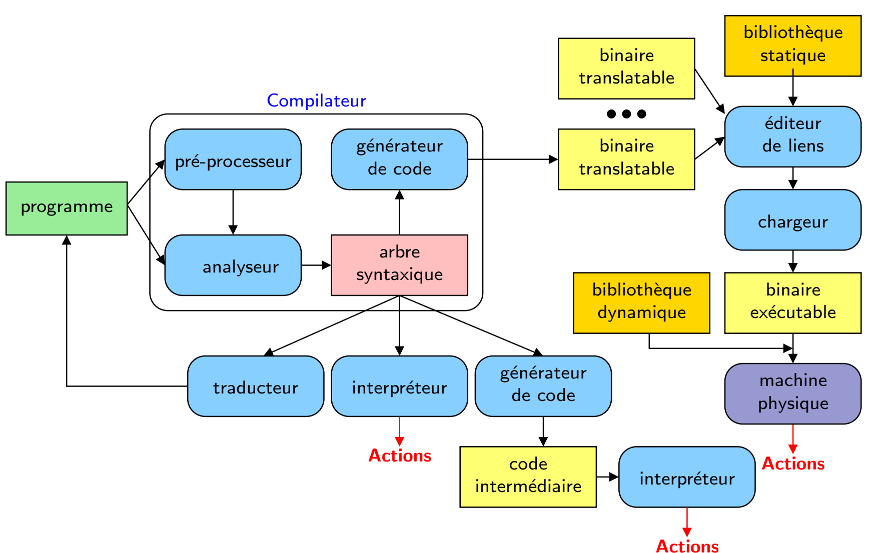
- nommé libltdl (LibTool Dynamic Loading library)
- Il permet également de compiler une bibliothèque dynamique de façon portable
- Associé à autoconf (usage peu aisé ; compilation depuis UNIX principalement)

Interface (fonctions de base : similaires à dlopen)

```
#include <ltdl.h>
lt_dlhandle lt_dlopen(const char *filename);
lt_ptr_t lt_dlsym(lt_dlhandle handle, const char *name);
int lt_dlclose(lt_dlhandle handle);
const char *lt_dlerror(void);
int lt_dlinit(void);
int lt_dlexit(void);
```

.

Schéma général d'exécution de programmes



Résumé du quatrième chapitre

Schémas d'exécution. Problème :

- Interprétation : peu efficace
- Compilation : mise au point, cycle complet à chaque modif
- Autres approches : machine virtuelle (Java)

Bibliothèques et liaison. Pourquoi :

- Éditeur de lien : assembler les "modules"
- Définitions multiples en C
- Bibliothèques statiques
  - Pourquoi : collection de fonctions
  - Comment : archive ar
  - Avantages et défaut : simple, mais duplique le code dans les binaires
- Bibliothèques dynamiques
  - Pourquoi : partage du code entre applications
  - Comment : edition de liens au lancement

Cinquième chapitre

Synchronisation entre processus

- Introduction : notions de base
  - Condition de compétition et exclusion mutuelle
  - Exclusion mutuelle par verrouillage de fichiers
  - Notion d'interblocage
  - Exclusion mutuelle par attente active
  - Problèmes de synchronisation (résumé)
- Sémaphores et schémas de synchronisation
  - Sémaphores
  - Exclusion mutuelle
  - Cohorte
  - Rendez-vous
  - Producteurs / Consommateurs
  - Lecteurs / Rédacteurs
- Autres outils de synchronisation
  - Moniteurs
  - Synchronisations POSIX
  - compare-and-swap
- Conclusion

Problème de l'exclusion mutuelle

Exemple : deux banques modifient un compte en même temps

Agence Nancy

```
1. courant = get_account(1867A)
2. nouveau = courant + 10
3. update_account (1867A, nouveau)
```

Agence Karlsruhe

```
1. aktuelles = get_account(1867A)
2. neue = aktuelles - 10
3. update_account(1867A, neue)
```

- variables partagées + exécutions parallèles entremêlées ⇒ différents résultats :
- (0 ; ? ; ?) N1(0 ; 0 ; ?) N2(0 ;10 ; ?) N3(10 ;10 ; ?) K1(10 ;10 ;10)K2(10 ;10 ;0) K3(0 ;10 ; 0) → compte inchangé
- (0 ; ? ; ?)N1 (0 ;0 ; ?) K1 (0 ;0 ;0) N2(0 ;10 ;0) K2(0 ;10 ;-10) N3(10 ;10 ;-10) K3(-10 ;10 ;-10) → compte −= 10
- (0 ; ? ; ?)K1 (0 ; ? ;0) N1 (0 ;0 ;0) K2(0 ;0 ;-10) N2(0 ;10 ;-10) K3(-10 ;10 ;-10)N3(10 ;10 ;-10) → compte += 10

C'est une **condition de compétition** (*race condition*)

- Solution : opérations **atomiques** ; pas d'exécutions entremêlées
- Cette opération est une **section critique** à exécuter en **exclusion mutuelle**

Réalisation d'une section critique

Schéma général

Processus 1

```
...
entrée en section critique
    section critique
sortie de section critique
...
```

Processus 2

```
...
entrée en section critique
    section critique
sortie de section critique
...
```

- Exclusion mutuelle garantie par les opérations `(entrée en section critique)` et `(sortie de section critique)`

Réalisation

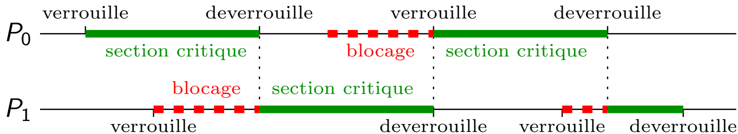
- Attente active : processus à l'entrée section critique boucle un test d'entrée
  - **Inefficace** (sur mono-processeur)
  - Parfois utilisé dans conditions praticulière dans le noyau
- Primitives spéciales : fournies par le système
  - Primitives générales : sémaphores, mutex (on y revient)
  - Mécanismes spécifiques : comme verrouillage de fichiers (idem)
  - Les primitives doivent être atomiques...

Exclusion mutuelle par verrouillage de fichiers

- Verrouiller une ressource garanti un accès exclusif à la ressource
- Unix permet de verrouiller les fichiers
- Appels systèmes de verrouillage/déverrouillage (*noms symboliques*) :
  - `verrouille(fich)` : verrouille `fich` avec accès exclusif
  - `déverrouille(fich)` : déverrouille `fich`
- Propriétés :
  - Opérations atomiques (assuré par l'OS)
  - Verrouillage par au plus un processus
  - Tentative de verrouillage d'un fichier verrouillé ⇒ blocage du processus
  - Deverrouillage ⇒ réveille d'un processus en attente du verrou (et un seul)

```
...
verrouille(fich);
accès à fich (section critique);
deverrouille(fich);
...
```

```
...
verrouille(fich);
accès à fich (section critique);
deverrouille(fich);
...
```



Verrouillage partagé

- Sémantique sur les fichiers : écritures exclusives, lectures concurrentes
- Nouvelle opération : ver-part (verrouillage partagé)

Déjà réalisé	Nouveau verrouille possible ?	Nouveau ver-part possible ?
verrouille(f)	non	non
ver-part(f)	non	oui

- Tentative de verrouillage d'un fichier verrouillé $\Rightarrow$ blocage du processus
- Déverrouillage $\Rightarrow$ réveille d'un processus en attente du verrou (et un seul)

Verrouillage de fichier sous Unix

Opérations disponibles

- Deux types de verrous : exclusifs ou partagés
- Deux modes de verrouillage :
  - Impératif (*mandatory*) : **bloque les accès** incompatibles avec verrous présents (mode décrit précédemment – pas POSIX)
  - Consultatif (*advisory*) : ne **bloque que la pose** de verrous incompatibles
  - Tous les verrous d'un fichier sont de même mode
- Deux primitives :
  - fcntl : générale et complexe
  - lockf : enveloppe plus simple, mais mode exclusif seulement (pas partagé)
- Possibilité de verrouiller fichier entier ou une partie

Interface du verrouillage de fichier sous Unix

```
#include <unistd.h>
int lockf(int fd, int commande, off_t taille);
```

- `fd` : descripteur du fichier à verrouiller
- `commande` : mode de verrouillage
  - F_ULOCK : déverrouiller
  - F_LOCK : verrouillage exclusif
  - F_TLOCK : verrouillage exclusif avec test (ne bloque jamais, retourne une erreur)
  - F_TEST : test seulement
- `taille` : spécifie la partie à verrouiller
  - $= 0$: fichier complet
  - > 0 : `taille` octets après position courante
  - < 0 : `taille` octets avant position courante
- Verrouillage consultatif et exclusif seulement
  Utiliser fcntl pour verrouillage impératif et/ou partagé
- Retour : 0 si succès, -1 sinon (*cf.* `errno`)

Exemple de verrouillage de fichiers Unix

testlock.c

```
#include <unistd.h>
int main (void) {
  int fd;
  fd = open("toto", O_RDWR);  /* doit exister */
  while (1) {
    if (lockf(fd, F_TLOCK, 0) == 0) {
      printf("%d: verrouille le fichier\n",
             getpid());
      sleep(5);
      if (lockf(fd, F_ULOCK, 0) == 0)
        printf("%d: déverrouille le fichier\n",
               getpid());
      return;
    } else {
      printf("%d: déjà verrouillé...\n",
             getpid());
      sleep (2);
    }
  }
}
```

```
$ testlock & testlock &
15545: verrouille le fichier
[4] 15545
15546: déjà verrouillé...
[5] 15546
$ 15546: déjà verrouillé...
15546: déjà verrouillé...
15545: déverrouille le fichier
15546: verrouille le fichier
15546: déverrouille le fichier
$
```

Exercice : que se passe-t-il si on remplace F_TLOCK par F_LOCK ?

15546 bloque jusqu'à ce que 15545 déverrouille le fichier
(et n'affiche rien entre temps)

L'exemple des banques revisité

Rappel du problème

Agence Nancy

```
1. courant = get_account(1867A)
2. nouveau = courant + 10
3. update_account (1867A, nouveau)
```

Agence Karlsruhe

```
1. aktuelles = get_account(1867A)
2. neue = aktuelles - 10
3. update_account(1867A, neue)
```

Implémentation avec le verrouillage de fichier UNIX

Agence Nancy

```
   int fd = open("fichier partagé", O_RDWR);
   lockf(fd, T_LOCK, 0);
1. courant = get_account(1867A)
2. nouveau = courant + 10
3. update_account (1867A, nouveau)
   lockf(fd, T_ULOCK, 0);
```

Agence Karlsruhe

```
   int fd = open("fichier partagé", O_RDWR);
   lockf(fd, T_LOCK, 0);
1. aktuelles = get_account(1867A)
2. neue = aktuelles - 10
3. update_account(1867A, neue)
   lockf(fd, T_ULOCK, 0);
```

Remarques

- Fichier partagé entre deux banques difficile à réaliser
- Autres solutions techniques pour les systèmes distribués ; l'idée reste la même

Notion d'interblocage

Utilisation simultanée de plusieurs verrous ⇒ problème potentiel

Situation

- Deux processus verrouillent deux fichiers

Processus 1

```
...
verrouille (f1) /* 1A */
accès à f1
...
verrouille (f2) /* 1B */
accès à f1 et f2
deverrouille (f2)
deverrouille (f1)
```

Processus 2

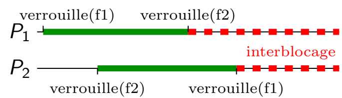
```
...
verrouille (f2) /* 2A */
accès à f2
...
verrouille (f1) /* 2B */
accès à f1 et f2
deverrouille (f2)
deverrouille (f1)
```

Déroulement

Exécution (pseudo-)parallèle

- Première possibilité :
  1a ; 1b ; 2a ; 2b
- Seconde possibilité :
  2a ; 2b ; 1a ; 1b
- Troisième possibilité :
  1a ; 2a ; 1b ; 2b

Exécution de 1a ;2a ;1b ;2b



- P1 et P2 sont bloqués *ad vitam eternam* :
  - P1 attend le `deverrouille(f2)` de P2
  - P2 attend le `deverrouille(f1)` de P1
- C'est un **interblocage** (*deadlock*)

Situation d'interblocage

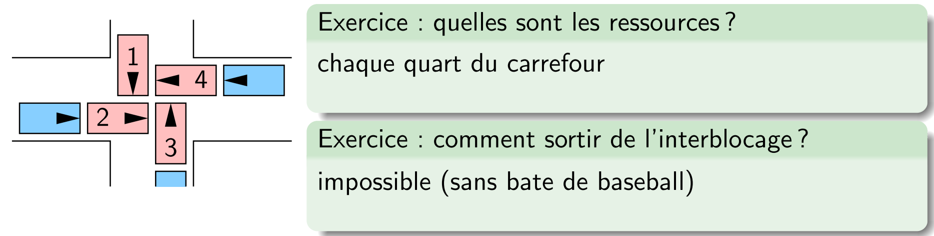
Définition

- Plusieurs processus bloqués dans l'attente d'une action de l'un des autres
- Impossible de sortir d'un interblocage sans intervention extérieure

Conditions d'apparitions

- Plusieurs processus en compétition pour les mêmes ressources
- Cycle dans la chaîne des attentes

Exemple : carrefour lyonnais un vendredi à 18h



Situation réelle d'interblocage

Comment prévenir l'interblocage ?

Solution 1 : réservation globale

- Demandes en bloc de toutes les ressources nécessaires
- Inconvénient : réduit les possibilités de parallélisme
- Analogie du carrefour : mettre des feux tricolores (et les respecter)

Solution 2 : requêtes ordonnées

- Tous les processus demandent les ressources dans le même ordre
- Interblocage alors impossible
- Analogie du carrefour : construire un rond-point

```
verrouille (f1)
verrouille (f2)
accès à f1 et f2
deverrouille (f2)
deverrouille (f1)
```

```
verrouille (f1)
verrouille (f2)
accès à f1 et f2
deverrouille (f2)
deverrouille (f1)
```

Solution 3 : modification de l'algorithme

- Modifier code utilisateur pour rendre impossible l'interblocage
- Analogie du carrefour : construire un pont au dessus du carrefour

Sortir de l'interblocage (quand on a pas su prévenir)

Impossible sans perdre quelque chose

Possibilités de guérison

- Faire revenir un processus en arrière
  - Nécessite d'avoir un point de reprise (*checkpoint*)
  - Travail depuis dernier point de reprise perdu
- Tuer l'un des processus pour casser le cycle

Conclusion

- Prévention et guérison sont tous les deux coûteux
  - Prévention : l'application doit faire attention
  - Guérison : détection + pertes dues au retour en arrière
- La ≪meilleure≫ solution dépend de la situation

Section critique par attente active

Principe (rappel)

- On demande tant qu'on a pas obtenu

Défaut

- Manque d'efficacité car gaspillage de ressources

Avantage

- Implémentable sans primitive de l'OS ni du matériel

$\Rightarrow$ utilisé dans certaines parties de l'OS, par exemple

Attention, ce n'est pas si simple à faire

Réalisation d'une section critique par attente active

- Hypothèses : atomicité d'accès et de modifications de variables
  V ne change pas au milieu de V==1 ni de V=1 (raisonnable sur monoprocesseur)

Solution FAUSSE numéro 1

```
while (tour != MOI);   /* attente active */
SC();                 /* section critique */
tour = 1-MOI;         /* soyons fair-play */
```

Demande alternance stricte :
Entrer deux fois de suite dans SC()
impossible (même si seul processeur)
C'est une **famine**

Solution FAUSSE numéro 2

```
while(flag[1-MOI]);      /*attendre autre*/
flag[MOI] = VRAI;        /*annonce entrer*/
SC();
flag[MOI] = FAUX;       /*débloque autre*/
```

Pas d'exclusion mutuelle (**compétition**) :
P0 teste flag[1] et trouve faux
P1 teste flag[0] et trouve faux
Les deux entrent dans SC()

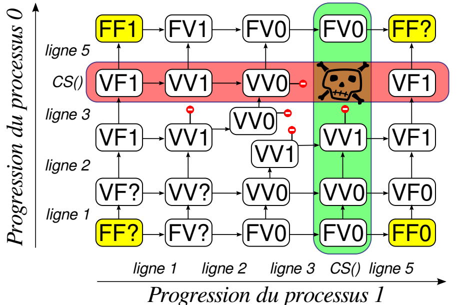
Solution FAUSSE numéro 3

```
flag[MOI]=VRAI;    /*annonce entrer avant*/
while (flag[1-MOI]);  /*attendre l'autre*/
SC();
flag[MOI] = FAUX;       /*débloque autre*/
```

Possibilité d'**interblocage** :
P0 : flag[0] $\leftarrow$ VRAI
P1 : flag[1] $\leftarrow$ VRAI
Les deux entrent dans leur boucle

Algorithme correct d'attente active

```
1. flag[MOI] = VRAI;                              /* Annonce être intéressé */
2. tour = 1-MOI;                        /* mais on laisse la priorité à l'autre */
3. while ((flag[1-MOI]==VRAI)  /* on entre si l'autre ne veut pas entrer */
          && (tour == 1-MOI));       /* ou s'il nous a laissé la priorité */
4. CS();
5. flag[MOI] = FAUX;                                           /* release */
```



GL Peterson. *A New Solution to Lamport's Concurrent Programming Problem*. 1983. (lire l'article)

http://en.wikipedia.org/wiki/Peterson's_algorithm

- Ceci est un diagramme de transition (rarement aussi régulier)

Problèmes de synchronisation (résumé)

- Condition de **compétition** *(race condition)*
  - Définition : le résultat change avec l'ordre des instructions
  - Difficile à corriger car difficile à reproduire (ordre ≪aléatoire≫)
  - Également type de problème de sécurité :
    - Un programme crée un fichier temporaire, le remplit puis utilise le contenu
    - L'attaquant crée le fichier avant le programme pour contrôler le contenu
- **Interblocage** *(deadlock)*
  - Définition : un groupe de processus bloqués en attente mutuelle
  - Évitement parfois difficile (correction de l'algorithme)
  - Détection assez simple, mais pas de guérison sans perte
- **Famine** *(starvation)*
  - Définition : un processus attend indéfiniment une ressource pourtant libre
  - Servir équitablement les processus demandeurs

Cinquième chapitre

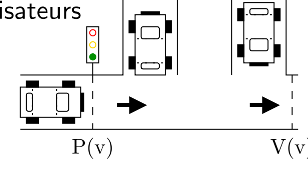
Synchronisation entre processus

- Introduction : notions de base
  - Condition de compétition et exclusion mutuelle
  - Exclusion mutuelle par verrouillage de fichiers
  - Notion d'interblocage
  - Exclusion mutuelle par attente active
  - Problèmes de synchronisation (résumé)
- Sémaphores et schémas de synchronisation
  - Sémaphores
  - Exclusion mutuelle
  - Cohorte
  - Rendez-vous
  - Producteurs / Consommateurs
  - Lecteurs / Rédacteurs
- Autres outils de synchronisation
  - Moniteurs
  - Synchronisations POSIX
  - compare-and-swap
- Conclusion

Sémaphore : outil de base pour la synchronisation

Cohorte : généralisation de l'exclusion mutuelle

- Ressource partagée par au plus N utilisateurs
- Exemples :
  - Parking payant
  - Serveur informatique



Sémaphore

- Objet composé :
  - D'une variable : valeur de la sémaphore (nombre de places restantes)
  - D'une file d'attente : liste des processus bloqués sur la sémaphore
- Primitives associées :
  - Initialisation (avec une valeur positive ou nulle)
  - Prise (P, Probeer, down, wait) = demande d'autorisation (≪puis-je ?≫)
    Si *valeur* = 0, blocage du processus ; Si non, *valeur* = *valeur* − 1
  - Validation (V, Verhoog, up, signal) = fin d'utilisation (≪vas-y≫)
    Si *valeur* = 0 et processus bloqué, déblocage d'un processus ;
    Si non, *valeur* = *valeur* + 1

E.W. Dijkstra. *Solution of a Problem in Concurrent Programming Control*. 1965. (lire l'article)

Schémas de synchronisation

Situations usuelles se retrouvant lors de coopérations inter-processus

- Exclusion mutuelle : ressource accessible par une seule entitée à la fois
  - Compte bancaire ; Carte son
- Problème de cohorte : ressource partagée par au plus N utilisateurs
  - Un parking souterrain peut accueillir 500 voitures (pas une de plus)
  - Un serveur doom peut accueillir 2000 joueurs
- Rendez-vous : des processus collaborant doivent s'attendre mutuellement
  - Roméo et Juliette ne peuvent se prendre la main que s'ils se rencontrent
  - Le GIGN doit entrer en même temps par le toit, la porte et la fenêtre
  - Processus devant échanger des informations entre les étapes de l'algorithme
- Producteurs/Consommateurs : un processus doit attendre la fin d'un autre
  - Une Formule 1 ne repart que quand tous les mécaniciens ont le bras levé
  - Réception de données sur le réseau *puis* traitement
- Lecteurs/Rédacteurs : notion d'accès exclusif entre *catégories* d'utilisateurs
  - Sur une section de voie unique, tous les trains doivent rouler dans le même sens
  - Un fichier pouvant être lu par plusieurs, si personne ne le modifie
  - Tâches de maintenance (défragmentation) quand pas de tâches interactives

Comment résoudre ces problèmes avec les sémaphores ?

Exclusion mutuelle par sémaphore

Très simple

Initialisation

```
sem=semaphore(1)
```

Agence Nancy

```
P(sem)
courant = get_account(1867A)
nouveau = courant + 1000
update_account (1867A, nouveau)
V(sem)
```

Agence Karlsruhe

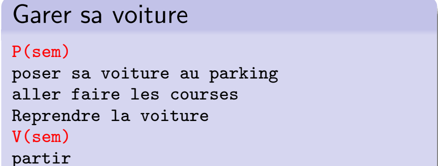
```
P(sem)
aktuelles = get_account(1867A)
neue = aktuelles - 1000
update_account(1867A, neue)
V(sem)
```

Cohortes et sémaphores

Sémaphore inventée pour cela

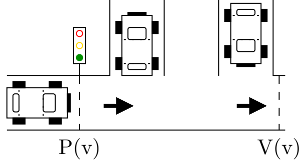
Initialisation

```
sem=semaphore(3)    /* nombre de places */
```



Garer sa voiture

```
P(sem)
poser sa voiture au parking
aller faire les courses
Reprendre la voiture
V(sem)
partir
```



Exercice : quelle est la valeur de *sem.v* ici ?

1, et ça sera 0 quand cette voiture sera rentrée

Rendez-vous et sémaphores

- Envoi de signal
  - Un processus indique quelque chose à un autre (disponibilité donnée)

Initialisation

```
top=semaphore(0)
```

Processus 1

```
...
calcul(info)
V(top)
...
```

Processus 2

```
...
P(top)    /*Bloque en attente*/
utilisation(info)
...
```

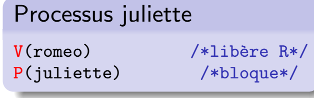
  - top = sémaphore privée (initialisée à 0)
    utilisée pour synchro *avec quelqu'un*, pas sur une ressource
- Rendez-vous entre deux processus
  - Les processus s'attendent mutuellement

Initialisation

```
roméo=semaphore(0)
juliette=semaphore(0)
```

Processus romeo

```
P(romeo)          /*se bloque*/
V(juliette)       /*libère J*/
```



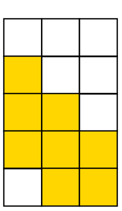
- Rendez-vous entre trois processus et plus
  - On parle de **barrière de synchronisation**
  - La solution précédente est généralisable, mais un peu lourde
  - Souvent une primitive du système

Producteurs et consomateurs

Contrôle de flux entre producteur(s) et consommateur(s) par tampon

Principe

- Situation initiale : tampon vide (pas de lecture possible)
- Après une production :
- Une autre production :
- Encore une production : (plus de production possible)
- Une consommation : (production de nouveau possible)



Réalisation

Initialisation

```
placeDispo=semaphore(N)
infoPrete=semaphore(0)
```

Producteur

```
répéter
  calcul(info)
  P(placeDispo)
  déposer(info)
  V(infoPrete)
```

Consommateur

```
répéter
  P(infoPrete)
  extraire(info)
  V(placeDispo)
  utiliser(info)
```

- Le tampon doit être circulaire pour traiter données dans l'ordre de production
- Attention aux compétitions entre producteurs (idem entre consommateurs)

Lecteurs et rédacteurs

Contrôle d'accès exclusif entre *catégories* d'utilisateurs

- Accès simultané de plusieurs lecteurs
- Accès exclusif d'un seul rédacteur, également exclusif de tout lecteur
- Schéma assez classique (fichier sur disque, zone mémoire, *etc.*)

Première solution

Initialisation

```
lecteur=semaphore(1)
rédacteur=semaphore(1)
NbLect=0
```

Lecteur

```
P(lecteur)
  NbLect = NbLect + 1
  si NbLect == 1 alors
    P(rédacteur)
V(lecteur)
lectures()
P(lecteur)
  NbLect = NbLect - 1
  si NbLect == 0 alors
    V(rédacteur)
V(lecteur)
```

Rédacteur

```
P(rédacteur)
  lecturesEtEcritures()
V(rédacteur)
```

- Problème : famine potentielle des rédacteurs

Lecteurs et rédacteurs sans famine

Initialisation

```
lecteur=semaphore(1)
rédacteur=semaphore(1)
fifo=semaphore(1)
NbLect=0
```

Lecteur

```
P(fifo)
  P(lecteur)
    NbLect = NbLect + 1
    si NbLect == 1 alors
      P(rédacteur)
  V(lecteur)
V(fifo)
lectures()
P(lecteur)
  NbLect = NbLect - 1
  si NbLect == 0 alors
    V(rédacteur)
V(lecteur)
```

Rédacteur

```
P(fifo)
  P(rédacteur)
V(fifo)
  lecturesEtEcritures()
V(rédacteur)
```

Exercice : pourquoi cette nouvelle sémaphore empêche la famine des rédacteurs ?

car tout le monde est bloqué dessus en arrivant
avant, les lecteurs pouvaient doubler les redacteurs car ils ne P(redacteur) pas

Exercice : montrer que les lecteurs peuvent encore partager l'accès

L'accès concurrent est placé apres le V(fifo), rien n'a changé à ce niveau

Cinquième chapitre

Synchronisation entre processus

- Introduction : notions de base
  - Condition de compétition et exclusion mutuelle
  - Exclusion mutuelle par verrouillage de fichiers
  - Notion d'interblocage
  - Exclusion mutuelle par attente active
  - Problèmes de synchronisation (résumé)
- Sémaphores et schémas de synchronisation
  - Sémaphores
  - Exclusion mutuelle
  - Cohorte
  - Rendez-vous
  - Producteurs / Consommateurs
  - Lecteurs / Rédacteurs
- Autres outils de synchronisation
  - Moniteurs
  - Synchronisations POSIX
  - compare-and-swap
- Conclusion

Moniteur

Problème des sémaphores

- **Tous les processus doivent les utiliser correctement**
- Mauvais comportement d'un seul ⇒ problème pour l'ensemble
  - Oubli d'un P(mutex) : CS pas respectée
  - Oubli d'un V(mutex) : deadlock (Deni de Service – DoS)
- Causes possibles :
  - Erreur de programmation
  - Programme malveillant

Solution : le moniteur (`synchronized` en Java)

- Sorte d'objet contenant :
  - Des variables partagées (privées)
  - Une sémaphore protectrice
  - Des méthodes d'accès
- Le compilateur ajoute les appels à la sémaphore pour protéger les méthodes
- Erreur impossible car usage seulement à travers l'API protégée

C.A.R. Hoare. *Monitors: An Operating System Structuring Concept*. 1974. (lire l'article)

L’exemple des banques avec un moniteur Java

```
public class compteBanquaire {
  private int balance;
  compteBanquaire() {
     balance = 0;
   }

  void synchronized modifie(int montant) {
    balance = balance + montant;
  }
}
```

La méthode est invoquable par un thread au plus (en exclusion mutuelle)

La complexité est laissée au compilateur

Synchroniser conjointement des méthodes Java

```
public class compteBanquaire {
 private int balance;
 compteBanquaire() { balance = 0; }

 void synchronized ajoute(int montant)
                        throws Exception {
  if (montant<0) {
    throw new Exception("Montant négatif");
  } else {
    balance = balance + montant;
  }
 }


 void synchronized retire(int montant)
                        throws Exception {
  if (montant<0) {
    throw new Exception("Montant négatif");
  } else if (balance - montant < 0) {
    throw new Exception("Solde insuffisant");
  } else {
    balance = balance - montant;
 }
}
```

Invocations sérialisées au sein de l'objet

- Entre threads
- Entre les méthodes du même objet
- Pas entre les instances
  (invocations parallèles sur $\neq$ objets)

Synchronisation explicite en Java

Parfois, on veut contrôler finement la synchronisation

- synchroniser un *morceau* de la méthode, pas toute la méthode
- deux groupes de méthodes en exclusion mutuelle par groupe, mais pas globale

```
class Tutu {
 synchronized void addObj(Object o){}
}class Toto {
 Tutu myTutu;
 synchronized void addName(String n){
   nameCount++;
   myTutu.addObj(name);
} }
```

- Moniteur de Toto acquis dans `addName`
- Invocation de `Tutu.addObj`
  $\Rightarrow$ Moniteur Tutu verrouillé
  $\rightsquigarrow$ deadlock toto $\leftrightarrow$ tutu possible

```
class Tutu {
 synchronized void addObj(Object o){}
}class Toto {
 Tutu myTutu;
 void addName(String n){
   synchronized(this) {
     nameCount++;
   } myTutu.addObj(name);
} }
```

Verrouillage d'une partie de la méthode :

- Il faut spécifier l'objet verrouillé (ici : `this`)
- Verrou relaché avant `Tutu.addObj`
  $\Rightarrow$ plus de deadlock possible
  (attention à d'éventuelles compétitions)

Les verrous Java sont réentrants : Reprendre le même verrou n'est pas bloquant

Moniteurs et conditions

Moniteurs ne permettent pas d'attendre un événement

- Comparable au P() sur sémaphore à 0 (cf. place libérée sur le parking)
- Nouveau type de variable : `condition x;` (sorte de file d'attente)
- Primitives associées :
  - `x.wait()` : Entre dans la file d'attente associée
  - `x.notify()` : Libère un processus en attente (ou noop si personne n'attend)
- En Java, chaque objet a une condition implicite associée

Exemple : lecteur/écrivain

```
class Channel {
  private int contenu;
  private boolean plein;
  Channel() { plein = false; }
  synchronized void enqueue(int val) {
    while (plein) {
      try {
        wait();
      } catch (InterruptedException e){}
    }
    contenu = val; plein = true;
    notifyAll();
  }
```

```
  synchronized int dequeue() {
    int recu;

    while (!plein) {
      try {
        wait();
      } catch (InterruptedException e){}
    }
    recu = contenu;
    full = false;
    notifyAll();
    return recu;
  }
}
```

Synchronisations POSIX

Les sémaphores

- Font partie de System V, mais également de POSIX

Les verrous : mutex (mutual exclusion)

- Comme une sémaphore de capacité 1 (un processus prend le verrou)
- Sémantique plus simple / pauvre
- Standard = interface ; multiples implémentations (Linux : futex ; BSD : spinlock)
- Mutex réentrants : impossible de faire un deadlock avec soi-même

Les variables de condition

- Pour envoyer un événement aux gens qui l'attendent (ça les débloque)
- `signal()` débloque un seul processus ; `broadcast()` débloque tout le monde.
- Par rapport aux sémaphores : message perdu si personne n'est encore bloqué
- Usage assez complexe : il faut protéger chaque variable par un mutex

On y revient dans le prochain chapitre

compare-and-swap

Autre problème des sémaphores, moniteurs and co

- C'est bloquant : seul un algorithme peut agir en même temps
- Si on cherche non-bloquant, il faut des opérations atomiques

Compare-And-Swap (instructions binaires CAS et CAS2)

- Opération atomique : tentative de modification d'une variable partagée
- Si la valeur partagée est ce que je pense, je change la variable
  Si non, je suis informé de la nouvelle valeur partagée

```
int CAS(int*adresse, int *ancienne_val, int nouvelle_val) {
  if (*adresse == *ancienne_val) {
    *adresse = nouvelle_val;
    return TRUE;
  } else {
    *ancienne_val = *adresse;
    return FALSE;
  }   }
```

- Brique de base pour implémenter les sémaphores dans l'OS
- Recherche (actuelle) : structures partagées « Lock-free » et « Wait-free »

John D. Valois. *Lock-Free Linked Lists Using Compare-and-Swap.* 1995. (lire l'article)

Résumé du cinquième chapitre

Problèmes à éviter lors des synchronisations (et définitions)

- Interblocage : groupe de processus en attente mutuelle
- Compétition : le résultat dépend de l'ordre d'exécution
- Famine : un processus n'obtient pas la ressource car les autres l'en empêchent

Les sémaphores

- Principe : distributeur de jetons
- Opérations :
  - initialisation : mettre des jetons dans la boite
  - P : prendre un jeton (ou bloquer s'il y en a plus)
  - V : rendre un jeton pris

Schémas de synchronisation classiques

- Exclusion mutuelle : Ressource utilisée par au plus un utilisateur
- Cohorte : Ressource partagée entre au plus N utilisateurs
- Rendez-vous : un processus attend un autre, ou attente mutuelle
- Producteurs/consommateurs : utilisation **après** création
- Lecteurs/rédacteurs : concurrence entre **catégories** d'utilisateurs

Résumé du cinquième chapitre (suite)

Moniteurs

- Principe : Objet auto-protégé par sémaphore
- Avantage : Pas d'erreur de manipulation possible
- Usage en Java : synchronized

Variable de condition

- Principe : Comme une sémaphore privée pour la communication entre threads
- Avantage : Permet le passage de relai entre threads
- Usage en Java : `wait()/notifyAll()`

CAS (Compare-And-Swap)

- Principe : Tentative de modif. si personne n'a modifié depuis dernière lecture
- Avantages : Implementer les sémaphores ; algo lock-free

Conclusion générale

Il n'y a pas de magie noire dans l'ordinateur

- Même un OS est finalement assez simple, avec des idées très générales
- Concepts : processus, mémoire, système de fichiers, pseudo-parallélisme
- Idées : interposition, préemption, tout-est-fichier, bibliothèque de fonctions
- Synchro : compétition, interblocage, famine. Sémaphore, mutex, moniteur ...

Indispensable pour comprendre l'ordinateur

- 1% d'entre vous vont coder dans l'OS, 5% vont coder si bas niveau
- Mais ces connaissances restent indispensables pour comprendre la machine
- Programmer efficace en Java demande de comprendre le tas et les threads
- Les problèmes de synchro restent assez similaires dans les couches hautes

Ce que nous n'avons pas vu

- Le chapitre sur les threads, du moins pas dans de bonnes conditions
- L'implémentation de l'OS (cf. RSA) et des machines virtuelles
- High Perf Computing : Message-passing, cache-aware, out-of-core, parallélisme
- Recherche en OS : Virtualisation, synchro wait-free, spec formelle, (exokernels)

Sixième chapitre

Programmer avec les threads

- Introduction aux threads
  - Comparaison entre threads et processus
  - Avantages et inconvénients des threads
  - Design d'applications multi-threadées
- Modèles de threads
  - Threads utilisateur ou noyau, modèles M:1, 1:1 et M:N
  - Études de cas
- Programmer avec les Pthreads
  - Gestion des threads
  - Mutexes, sémaphores POSIX et variables de condition
- Conclusion

Qu'est ce qu'un thread

Définition d'un thread

- Techniquement : un flot d'exécution pouvant être ordonnancé par l'OS
- En pratique : une procédure s'exécutant indépendemment du main
- Mot français : fil d'exécution ou processus léger (*lightweight process*)

Programme *multi-threadé*

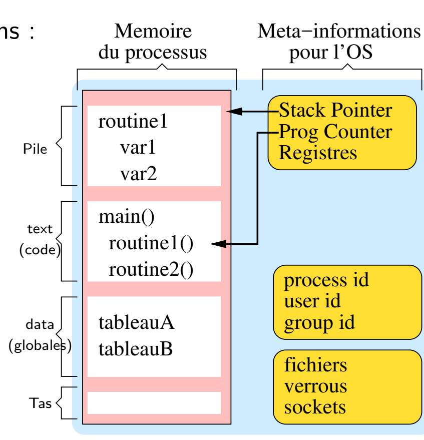
- Un programme contenant diverses fonctions et procédures
- Ces procédures peuvent être exécutées simultanément et/ou indépendamment

Comparaison de processus et threads (1/2)

Processus «classique» (ou «lourd»)

Processus UNIX contient diverses informations :

- PID, PGID, UID, GID
- L'environnement
- Répertoire courant
- Pointeur d'instruction (PC)
- Registres
- Pile
- Tas
- Descripteurs de fichier
- Gestionnaires de signaux
- Bibliothèques partagées
- Outils d'IPC (tubes, sémaphores, shmem, etc)

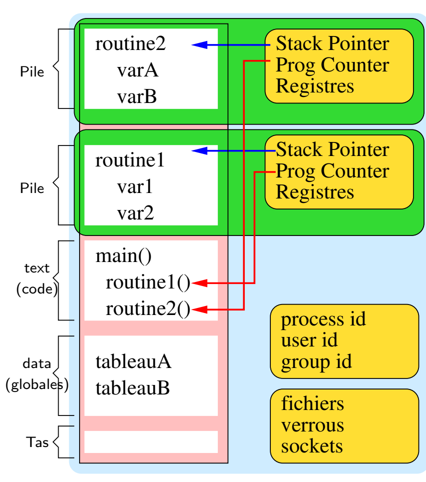


Processus classique

Comparaison de processus et threads (2/2)

Processus légers

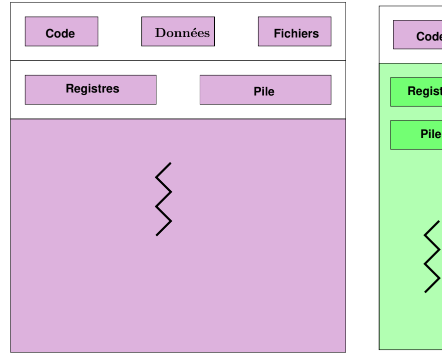
- Plusieurs threads coexistent dans le même processus ≪lourd≫
- Ils sont ordonnançables séparément
- Informations spécifiques
  - Pile
  - Registres
  - Priorité (d'ordonnancement)
  - Données spécifiques
  - Liste des signaux bloqués
- Le reste est partagé
  - Si un thread ferme un fichier, il est fermé pour tous
  - Si un thread fait `exit()`, tout s'arrête
  - Globales et pointeurs vers le tas : variables partagées (à synchroniser)



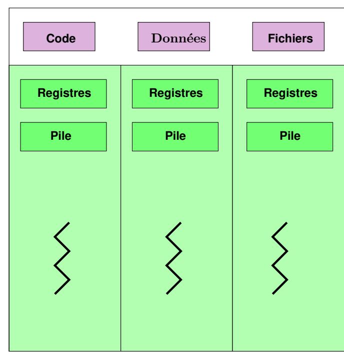
Processus legers

Processus vs. thread en résumé

- Processus : environnement d'exécution pour les threads au sein de l'OS
  État vis-à-vis de l'OS (fichiers ouverts) et de la machine (mémoire)
- Processus = Thread + Espace d'adressage



Processus mono–thread



Processus multi–threads

L'espace d'adressage est *passif* ; le thread est *actif*

Pourquoi processus dits «légers» ?

Les threads contiennent moins de choses

- Partagent la plupart des ressources du processus lourd
- Ne dupliquent que l'indispensable
- ⇒ 2 threads dans 1 processus consomment moins de mémoire que 2 processus

Les threads sont plus rapides à créer

Machine	fork()			threads		
	real	user	sys	real	user	sys
Celeron 2GHz	4.479s	0.364s	3.756s	1.606s	0.380s	0.388s
AMD64 2.5GHz (4CPU)	7.006s	0.936s	6.244s	0.903s	0.300s	0.640s

Les communications inter-threads sont rapides

- Bus PCI (donc, carte réseau) : 128 Mb/s
- Copie de mémoire (Tubes ; Passage message sur SMP) : 0,3 Gb/s
- Mémoire vers CPU (Pthreads) : 4 Gb/s

Usage des threads

Pourquoi utiliser les threads

- Objectif principal : gagner du temps (threads moins gourmands que processus)

Quand utiliser les threads

- Pour un recouvrement calcul/communication
- Pour avoir différentes tâches de priorité différentes
  ordonnancement temps réel ≪mou≫
- Pour gérer des événements asynchrones
  Tâches indépendantes activées par des événements de fréquence irrégulière
  Exemple : Serveur web peut répondre à plusieurs requêtes en parallèle
- Pour tirer profit des systèmes SMP ou CMP (multi-cœurs)

Quand (pourquoi) ne pas utiliser les threads

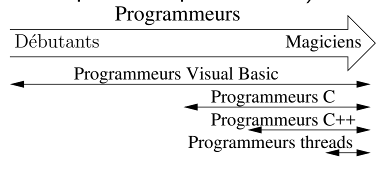
Problèmes du partage de la mémoire

- Risque de corruption mémoire (risque de compétition)
- Besoin de synchronisation (risque d'interblocage)

$\Rightarrow$ Communication inter-threads rapide mais dangereuse

- Segfault d'un thread $\rightarrow$ mort de tous
- Casse l'abstraction en modules indépendants
- Extrêmement difficile à debugger (dépendances temporelles ; pas d'outils)

Programmer avec les threads,
c'est enlever les gardes-fous de l'OS
pour gagner du temps



(Why Threads are a Bad Idea, USENIX96)

Obtenir de bonnes performances est très difficile

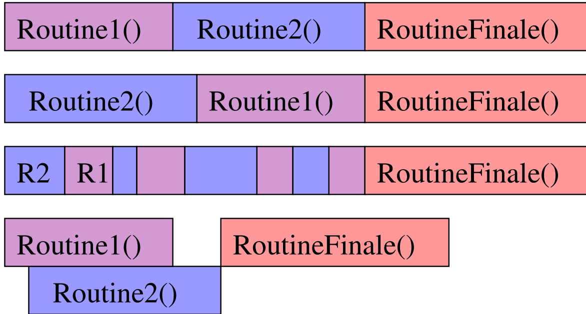
- Verrouillage simple (moniteurs) amène peu de concurrence
- Verrouillage fin augmente la complexité (concurrence pas facilement meilleure)

Design d'applications multi-threadées

Applications candidates

- Applis organisées en tâches indépendantes s'exécutant indépendamment
- Toutes routines pouvant s'exécuter en (pseudo-)parallèle sont candidates
- Interface interactive ($\leadsto$ latence indésirable) avec des calculs ou du réseau

Dans l'exemple, routine1() et routine2() sont candidates



Code réentrant et threads-safeness

Code réentrant

- Définition : Peut être appelé récursivement ou depuis plusieurs ≪endroits≫
- Ne pas maintenir d'état entre les appels
  - Contre-exemple : `strtok`, `rand` (`strtok_r` est réentrante)
- Ne pas renvoyer un pointeur vers une statique
  - Contre-exemple : `ctime` (`ctime_r` est réentrante)

Code thread-safe

- Définition : Fonctionne même si utilisé de manière concurente
- Si le code n'est pas réentrant, il faut le protéger par verrous

Problème des dépendances

- Votre code est thread-safe, mais vos bibliothèques le sont-elles ?
- La libc est réentrante (sous linux modernes)
  - Exemple de errno : chaque thread a maintenant son propre errno
- Pour le reste, il faut vérifier (voire, supposer qu'il y a un problème)
- En cas de problème, il faut protéger les appels grâce à un verrou explicite

Patterns classiques avec les threads

Maitre/esclaves

- Un thread centralise le travail à faire et le distribue aux esclaves
- Pool d'esclaves statique ou dynamique
- Exemple : serveur web

Pipeline

- La tâche est découpée en diverses étapes successives
  Chaque thread réalise une étape et passe son résultat au suivant
- Exemple : traitement multimédia (streaming)

Peer to peer

- Comme un maitre/esclaves, mais le maitre participe au travail

Sixième chapitre

Programmer avec les threads

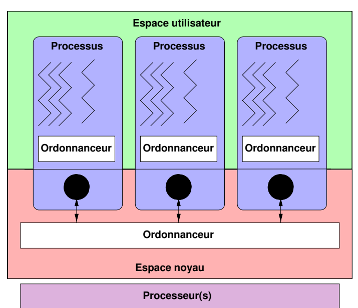
- Introduction aux threads
  - Comparaison entre threads et processus
  - Avantages et inconvénients des threads
  - Design d'applications multi-threadées
- Modèles de threads
  - Threads utilisateur ou noyau, modèles M:1, 1:1 et M:N
  - Études de cas
- Programmer avec les Pthreads
  - Gestion des threads
  - Mutexes, sémaphores POSIX et variables de condition
- Conclusion

Implantation des threads : Modèle M:1

Tous les threads d'un processus mappés sur un seul thread noyau

Service implémenté dans une bibliothèque spécifique

- Gestion des threads au niveau utilisateur
- Avantages :
  - Pas besoin de modifier le noyau
  - Rapides pour créer un thread, et pour changer de contexte
  - Portables entre OS
- Inconvénients :
  - Ordonnancement limité à celui d'un processus système
    ⇒ ≪Erreurs≫ d'ordonnancement
  - Appel bloquant dans un thread
    ⇒ blocage de tous
  - Pas de parallélisme
    (dommage pour les SMP)



(D'après F. Silber)

⇒ Efficace, mais pas de concurrence

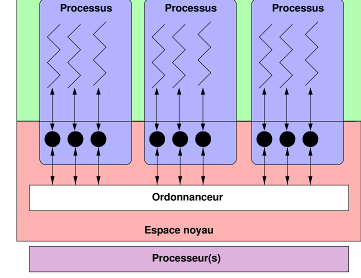
- Exemples : Fibres Windows, Java Green Threads, GNU pth (*portable thread*)

Modèle 1:1

Chaque thread utilisateur mappé sur un thread noyau

Modification du noyau pour un support aux threads

- Gestion des threads au niveau système (LightWeight Process – LWP)
- Avantages
  - Appel bloquant dans un thread
    ⇒ exécution d'un autre
  - Parallélisme possible
  - Ordonnancement du noyau plus approprié
- Inconvénients
  - Besoin d'appels systèmes spécifiques
    (`clone()` pour la création sous linux)
  - Commutation réalisée par le noyau
    ⇒ changements de contexte (coûteux)



(D'après F. Silber)

⇒ Concurrence importante, mais moins efficace
(nombre total de threads souvent borné)

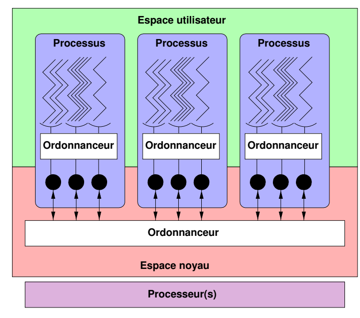
- Exemples : Windows 95/98/NT/2000, OS/2, Linux, Solaris 9+

Modèle M:N

M threads utilisateur mappés sur N threads noyau ($M \geq N \geq 1$)

Services utilisateur basés sur des services noyau

- Coopération entre l'ordonnanceur noyau et un ordonnanceur utilisateur
- Avantages : le meilleur des deux mondes
  - Threads noyaux plus basiques et efficaces
  - Threads utilisateurs plus flexibles
  - Moins de changement de contexte
  - Pas d'erreur d'ordonnancement
- Inconvénients :
  - Extrêmement difficile à implémenter
  - Un peu ≪usine à gaz≫ :
    - modèle théorique plus pur
    - implémentations complexes
    - efficacité parfois discutable
- Concurrence importante, bonne efficacité (?)
- Exemples : Solaris avant v9, IRIX, HP-UX



(D'après F. Silber)

Études de cas (1/3)

Pthreads

- POSIX threads : norme standardisé par IEEE POSIX 1003.1c (1995)
- Ensemble de types et fonctions C dont la sémantique est spécifiée
- Seulement une API : implémentation sous tous les UNIX (et même Windows)

Threads Solaris

- Modèle M:N
- Threads intermédiaires (LWP)

Threads Windows 2000

- Modèle 1:1 pour les WinThreads
- Modèle M:1 pour les fibres (l'OS ne voit pas les fibres)

Threads Java

- Extension de la classe `Thread` ; Implantation de l'interface `Runnable`
- Implémentation dépend de la JVM utilisée

Études de cas (2/3)

Threads sous linux

- Threads mappé sur des *tâches* (tasks)
- Appel système `clone()` pour duppliquer une tâche (fork() implémenté avec ça)

Plusieurs bibliothèques implémentant le standard Pthreads

- Depuis 1996 : LinuxThreads
  - Modèle 1:1, par Xavier Leroy (INRIA, créateur d'Ocaml)
  - Pas complètement POSIX (gestion des signaux, synchronisation)
  - $\approx$ 1000 threads au maximum
- Next Generation POSIX Threads
  - Modèle M:N, par IBM ; abandonné
  - $\leadsto$ NPTL avançait plus vite
  - $\leadsto$ M:N induisait une complexité trop importante
- Maintenant Native POSIX Thread Library
  - Modèle 1:1, par Ulrich Drepper (acteur principal libc, chez Red Hat)
  - Totalement conforme à POSIX
  - Bénéficie des fonctionnalités du noyau Linux 2.6 (ordonnancement $O(1)$)
  - Création de 100 000 threads en 2 secondes (contre 15 minutes sans)

Études de cas (3/3)

Possibilités de quelques systèmes d'exploitations

	1 seul espace d'adressage	plusieurs espaces d'adressage
1 seul thread par espace d'adressage	MS/DOS Macintosh (OS9)	Unix traditionnels
plusieurs threads par espace d'adressage	Systèmes embarqués	Win/NT, Linux, Solaris HP-UP, OS/2, Mach, VMS

Sixième chapitre

Programmer avec les threads

- Introduction aux threads
  - Comparaison entre threads et processus
  - Avantages et inconvénients des threads
  - Design d'applications multi-threadées
- Modèles de threads
  - Threads utilisateur ou noyau, modèles M:1, 1:1 et M:N
  - Études de cas
- Programmer avec les Pthreads
  - Gestion des threads
  - Mutexes, sémaphores POSIX et variables de condition
- Conclusion

Généralités sur l'interface Pthreads

Trois grandes catégories de fonctions / types de données

Gestion des threads

- Créer des threads, les arrêter, contrôler leurs attributs, etc.

Synchronisation par mutex (**mut**ual **ex**clusion)

- Créer, détruire verrouiller, déverrouiller des mutex ; Contrôler leurs attributs

Variables de condition :

- Communications entre threads partageant un mutex
- Les créer, les détruire, attendre dessus, les signaler ; Contrôler leurs attributs

Préfixe d'identificateur	Groupe
`pthread_`	Threads eux-mêmes et diverses fonctions
`pthread_attr_`	Attributs des threads
`pthread_mutex_`	Mutexes
`pthread_cond_`	`Variables de condition`
...	...

L'interface Pthreads en pratique

- Types de données sont structures opaques, fonctions d'accès spécifiques
- L'interface compte 60 fonctions, nous ne verrons pas tout
- Il faut charger le fichier d'entête `pthread.h` dans les sources
- Il faut spécifier l'option `-pthread` à gcc

thread-ex1.c

```
#include <pthread.h>
void *hello( void *arg ) {
  int *id = (int*)arg;
  printf("%d: hello world \n", *id);
  pthread_exit(NULL);
}
int main (int argc, char *argv[ ]) {
  pthread_t thread[3];
  int id[3]={1,2,3};
  int i;

  for (i=0;i<3;i++) {
    printf("Crée thread %d\n",i);
    pthread_create(&thread[i], NULL,
                   hello, (void *)&id[i]);
  }
  pthread_exit(NULL);
}
```

```
$ gcc -pthread -o thread-ex1 thread-ex1.c
$ ./thread-exemple1
Crée thread 1
Crée thread 2
Crée thread 3
1 : hello world
2 : hello world
3 : hello world
$
```

Identification et création des threads

Identifier les threads

- **pthread_t** : équivalent du pid_t (c'est une structure opaque)
- pthread_t **pthread_self** () : identité du thread courant
- int **pthread_equal** (pthread_t, pthread_t) : test d'égalité

Créer de nouveaux threads

- Le programme est démarré avec un seul thread, les autres sont créés à la main
- int **pthread_create**(identité, attributs, fonction, argument) ;
  - identité : [pthread_t*] identifieur unique du nouveau thread (*rempli par l'appel*)
  - attributs : Pour modifier les attributs du thread (NULL → attributs par défaut)
  - fonction : [void *(*)(void *)] Fonction C à exécuter (le ≪main du thread≫)
  - argument : argument à passer à la fonction. Doit être transtypé en void*
  - retour : 0 si succès, errno en cas de problème

Exercice : Comment savoir dans quel ordre vont démarrer les threads créés ?

On peut pas, ca dépend des fois (gare aux races conditions)

Exercice : Comment passer deux arguments à la fonction ?

en définissant une structure

Exemple FAUX de passage d'arguments

thread-faux1.c

```
#include <pthread.h>
void *hello( void *id ) {
  printf("%d: hello world \n", (char *) id);
  pthread_exit(NULL);
}

int main (int argc, char *argv[ ]) {
  pthread_t th[3];
  int i;

  for (i=0;i<3;i++) {
    printf("Crée thread %d\n",i);
    pthread_create(&th[i], NULL, hello, (void *)&i);
  }

  pthread_exit(NULL);
}
```

.

Exercice : Quel est le problème ?

On transforme i en globale inter-thread, et sa valeur peut donc être changée dans le thread lanceur avant le démarrage du thread utilisateur. C'est une belle condition de compétition.

Exemple de passage d’arguments complexes

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 8

typedef struct data {
   int thread_id,sum;
   char *msg;
} data_t;

data_t data_array[NUM_THREADS];

void *PrintHello(void *arg) {
  data_t *mine = (data_t *)arg;

  sleep(1);
  printf("Thread %d: %s  Sum=%d\n",
         mine->thread_id,
         mine->msg,
         mine->sum);
  pthread_exit(NULL);
}
```

```c
int main(int argc, char *argv[ ]) {
  pthread_t threads[NUM_THREADS];
  int rc, t, sum=0;

  char *messages[NUM_THREADS] = {
    "EN: Hello World!", "FR: Bonjour, le monde!",
    "SP: Hola al mundo", "RU: Zdravstvytye, mir!",
    "DE: Guten Tag, Welt!", "Klingon: Nuq neH!",
    "JP: Sekai e konnichiwa!",
    "Latin: Orbis, te saluto!" };

  for(t=0;t<NUM_THREADS;t++) {
    sum = sum + t;
    thread_data_array[t].thread_id = t;
    thread_data_array[t].sum = sum;
    thread_data_array[t].message = messages[t];
    printf("Creating thread %d\n", t);
    rc = pthread_create(&threads[t], NULL,
                        PrintHello,
                        (void*)&thread_data_array[t]);
    if (rc) {
      printf("ERROR; _create() returned %d\n", rc);
      exit(-1);
    }
  }
  pthread_exit(NULL);
}
```

Terminaison des threads

Causes de terminaison des threads

- Le thread termine sa fonction initiale
- Le thread appelle la routine `pthread_exit()`
- Le thread est tué par un autre thread appelant `pthread_cancel()`
- Tout le processus se termine à cause d'un `exit`, `exec` ou `return` du main()

Terminer le thread courant

- void **pthread_exit**(void *retval) ;
  - retval : valeur de retour du thread (optionnel)
- Pour récupérer code de retour, un autre thread doit utiliser `pthread_join()`

Attendre la fin d'un thread

Joindre un thread

- int **pthread_join** (pthread_t, void **)
- Le thread A joint le thread B : A bloque jusqu'à la fin de B
- Utile pour la synchronisation (rendez-vous)
- Second argument reçois un pointeur vers la valeur retour du `pthread_exit()`
- Similaire au `wait()` après `fork()`
  (mais pas besoin d'être le créateur pour joindre un thread)

Détacher un thread

- int **pthread_detach** (pthread_t)
- Détacher un thread revient à dire que personne ne le joindra à sa fin
- Le système libère ses ressources au plus vite
- Évite les threads zombies quand on ne veut ni synchro ni code retour
- On ne peut pas ré-attacher un thread détaché

Attributs des threads

C'est le second argument du create

Les fonctions associées

- Création et destruction d'un objet à passer en argument à _create :
  - int **pthread_attr_init** (pthread_attr_t *)
  - int **pthread_attr_destroy** (pthread_attr_t *)
- Lecture : int **pthread_attr_getX** (const pthread_attr_t *, **T** *)
- Mise à jour : int **pthread_attr_setX** (pthread_attr_t *, **T**)

Attributs existants

- detachstate (int) : PTHREAD_CREATE_[JOINABLE|DETACHED]
- schedpolicy (int)
  - SCHED_OTHER : Ordonnancement classique
  - SCHED_RR : Round-Robin (chacun son tour)
  - SCHED_FIFO : Temps réel
- schedparam (int) : priorité d'ordonnancement
- stackaddr (void*) et stacksize (size_t) pour régler la pile
- inheritsched, scope

Exemple de gestion des threads

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 3

void *travail(void *null) {
 int i;
 double result=0.0;
 for (i=0; i<1000000; i++)
   result = result + (double)random();
 printf("Resultat = %e\n",result);
 pthread_exit(NULL);
}

int main(int argc, char *argv[ ]) {
 pthread_t thread[NUM_THREADS];
 pthread_attr_t attr;
 int rc, t;
 /* Initialise et modifie l'attribut  */
 pthread_attr_init(&attr);
 pthread_attr_setdetachstate(&attr,
                PTHREAD_CREATE_JOINABLE);
```

```
 for(t=0;t<NUM_THREADS;t++) {
  printf("Creating thread %d\n", t);
  if ((rc = pthread_create(&thread[t], &attr,
                         travail, NULL)))  {
   printf("ERREUR de _create() : %d\n", rc);
   exit(-1);
 }}
 /* Libère l'attribut */
 pthread_attr_destroy(&attr);
 /* Attend les autres */
 for(t=0;t<NUM_THREADS;t++) {
  if ((rc = pthread_join(thread[t],
     /*pas de retour attendu*/NULL))) {
   printf("ERREUR de _join() : %d\n", rc);
   exit(-1);
  }
  printf("Rejoint thread %d. (ret=%d)\n",
         t, status);
 }
 pthread_exit(NULL);
}
```

Sixième chapitre

Programmer avec les threads

- Introduction aux threads
  - Comparaison entre threads et processus
  - Avantages et inconvénients des threads
  - Design d'applications multi-threadées
- Modèles de threads
  - Threads utilisateur ou noyau, modèles M:1, 1:1 et M:N
  - Études de cas
- Programmer avec les Pthreads
  - Gestion des threads
  - Mutexes, sémaphores POSIX et variables de condition
- Conclusion

Les mutex Pthreads

Interface de gestion

- Type de donnée : **pthread_mutex_t**
- Création :
  - statique : pthread_mutex_t mutex = **PTHREAD_MUTEX_INITIALIZER**;
  - dynamique : **pthread_mutex_init**(pthread_mutex_t *, pthread_mutexattr_t *);
    Premier argument : adresse de (pointeur vers) la variable à initialiser
- Destruction : **pthread_mutex_destroy**(mutex) (doit être déverrouillé)
- (pas d'attributs POSIX, mais il y en a dans certaines implémentations)

Interface d'usage

- **pthread_mutex_lock**(mutex) Bloque jusqu'à obtention du verrou
- **pthread_mutex_trylock**(mutex) Obtient le verrou ou renvoie EBUSY
- **pthread_mutex_unlock**(mutex) Libère le verrou

Usage des mutex

Les mutex sont un «gentlemen's agreement»

Thread 1

```
Lock
A = 2
Unlock
```

Thread 2

```
Lock
A = A+1
Unlock
```

Thread 3

```
A = A*B
```

- Thread 3 crée une condition de compétition même si les autres sont disciplinés (cf. les verrous consultatifs sur les fichiers)
- Pas d'équivalent des verrous impératifs sur la mémoire
- Pas de moniteurs dans PThreads (durs à implémenter en C)

Sémaphores POSIX

- On parle ici de l'interface POSIX, pas de celle IPC Système V
- #include <semaphore.h>
- Type : **sem_t**

Interface de gestion

- int **sem_init**(sem_t *sem, int pshared, unsigned int valeur)
  pshared != 0 ⇒ sémaphore partagée entre plusieurs processus (pas sous linux)
- int **sem_destroy**(sem_t * sem) (personne ne doit être bloqué dessus)

Interface d'usage

- int **sem_wait**(sem_t * sem) réalise un P()
- int **sem_post**(sem_t * sem) réalise un V()
- int **sem_trywait**(sem_t * sem) P() ou renvoie EAGAIN pour éviter blocage
- int **sem_getvalue**(sem_t * sem, int * sval)

Variables de condition

Interface de gestion

- Type de donnée : **pthread_cond_t**
- Création :
  - statique : pthread_cond_t cond = **PTHREAD_COND_INITIALIZER** ;
  - dynamique : int **pthread_cond_init**(pthread_cond_t *, pthread_condattr_t *)
- Destruction : int **pthread_cond_destroy**(pthread_cond_t *)
- Il n'y a pas d'attributs POSIX (ni linux) pour les conditions

Rappel du principe des conditions

- Cf. conditions Java (`wait()` et `notifyAll()`)
- Autre mécanisme de synchronisation.
  - mutex : solution pratique pour l'exclusion mutuelle
  - sémaphore : solution pratique pour les cohortes
  - condition : solution pratique pour les rendez-vous
  - compare-and-swap : solution pour la synchronisation non-bloquante

Exemple d'usage des conditions

Thread principal

(1) Déclare et initialise une globale requérant une synchronisation (ex : `compteur`)

(1) Déclare et initialise une variable de condition et un mutex associé

(1) Crée deux threads (A et B) pour faire le travail

Thread A

(2) Travaille jusqu'au rendez-vous (point où B doit remplir une condition, *compteur* > 0 par ex)

(2) Verrouille mutex et consulte variable

(2) Attend la condition de B
ça libère le mutex (pour B) et gèle A

(4) Au signal, réveil
mutex automagiquement repris

(4) Déverrouille explicitement

Thread B

(2) Fait des choses

(3) Verrouille le mutex

(3) Modifie la globale attendue par A

(3) Si la condition est atteinte, signale A

(3) Déverrouille le mutex

Thread principal : (5) Rejoint les deux threads puis continue

Les conditions POSIX

Interface d'usage

- pthread_cond_signal(pthread_cond_t *) Débloque un éventuel thread bloqué
- pthread_cond_wait(pthread_cond_t *, pthread_mutex_t *) Bloque tant que la condition n'est pas réalisée
- pthread_cond_timedwait(pthread_cond_t*, pthread_mutex_t*,struct timespec*) Bloque au plus tard jusqu'à la date spécifiée
- pthread_cond_broadcast(pthread_cond_t *) Réveille tous les bloqués (sont encore bloqués par mutex)

Pièges des conditions

- Risque d'interblocage ou de pas de blocage si protocole du mutex pas suivi
- Différence sémantique avec sémaphore :
  - Si sem==0, alors V(sem) $\Rightarrow$ $sem := 1$
  - Signaler une condition que personne n'attend : noop (info perdue)

Résumé du sixième chapitre

- Définition de threads (vs. processus) : proc=espace d'adressage+meta-données OS ; thread=fil d'exécution
- Avantages et inconvénients : Ca va plus vite, mais c'est plus dangereux
- Schémas d'implémentations :
  - Modèle M:1 : M threads par processus (portable et rapide, pas parallèle)
  - Modèle 1:1 : Threads gérés par l'OS directement (plus dur à implémenter, plus efficace en pratique)
  - Modèle M:N : Gestion conjointe (modèle théorique meilleur, difficile à faire efficacement)
- Les fonctions principales de l'interface POSIX :
  - `pthread_create` : crée un nouveau thread
  - `pthread_exit` : termine le thread courant
  - `pthread_join` : attend la fin du thread et récupère son errcode
  - `pthread_detach` : indique que personne ne rejoindra ce thread
  - `mutex_{lock,unlock,*}`: usage des mutex
  - `sem_{wait,post,*}`: usage des sémaphore
  - `cond_{wait,signal,*}`: usage des conditions