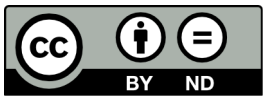
université de BORDEAUX

Programmation système

(INF355)

F. Pellegrini

Université de Bordeaux



Structure d'un ordinateur (1)

- Un ordinateur est une machine programmable de traitement de l'information
- Pour accomplir sa fonction, il doit pouvoir :
  - Acquérir de l'information de l'extérieur
  - Stocker en son sein ces informations
  - Combiner entre elles les informations à sa disposition
  - Restituer ces informations à l'extérieur

Structure d'un ordinateur (2)

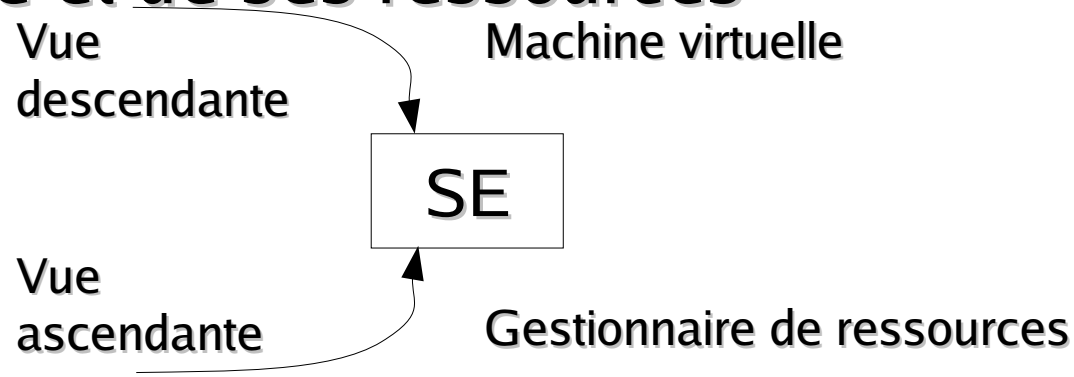
- L'ordinateur doit donc posséder :
  - Une ou plusieurs unités de stockage, pour mémoriser le programme en cours d'exécution ainsi que les données qu'il manipule
  - Une unité de traitement permettant l'exécution des instructions du programme et des calculs sur les données qu'elles spécifient
  - Différents dispositifs « périphériques » servant à interagir avec l'extérieur : clavier, écran, souris, carte graphique, carte réseau, etc.

Système d'exploitation (1)

- Un système d'exploitation est un programme qui, du point de vue du programmeur, ajoute une variété de fonctionnalités en plus de celles déjà offertes par la couche ISA (« *Instruction set architecture* »)
- Services rendus sous la forme :
  - D'appels système
  - D'appels de fonctions de bibliothèques

Système d'exploitation (2)

- Buts d'un système d'exploitation
  - Décharger le programmeur d'une tâche de programmation énorme et fastidieuse et lui permettre de se concentrer sur l'écriture de son application
  - Protéger le matériel et ses usagers des fausses manipulations
  - Offrir une vue simple, uniforme, et cohérente de la machine et de ses ressources



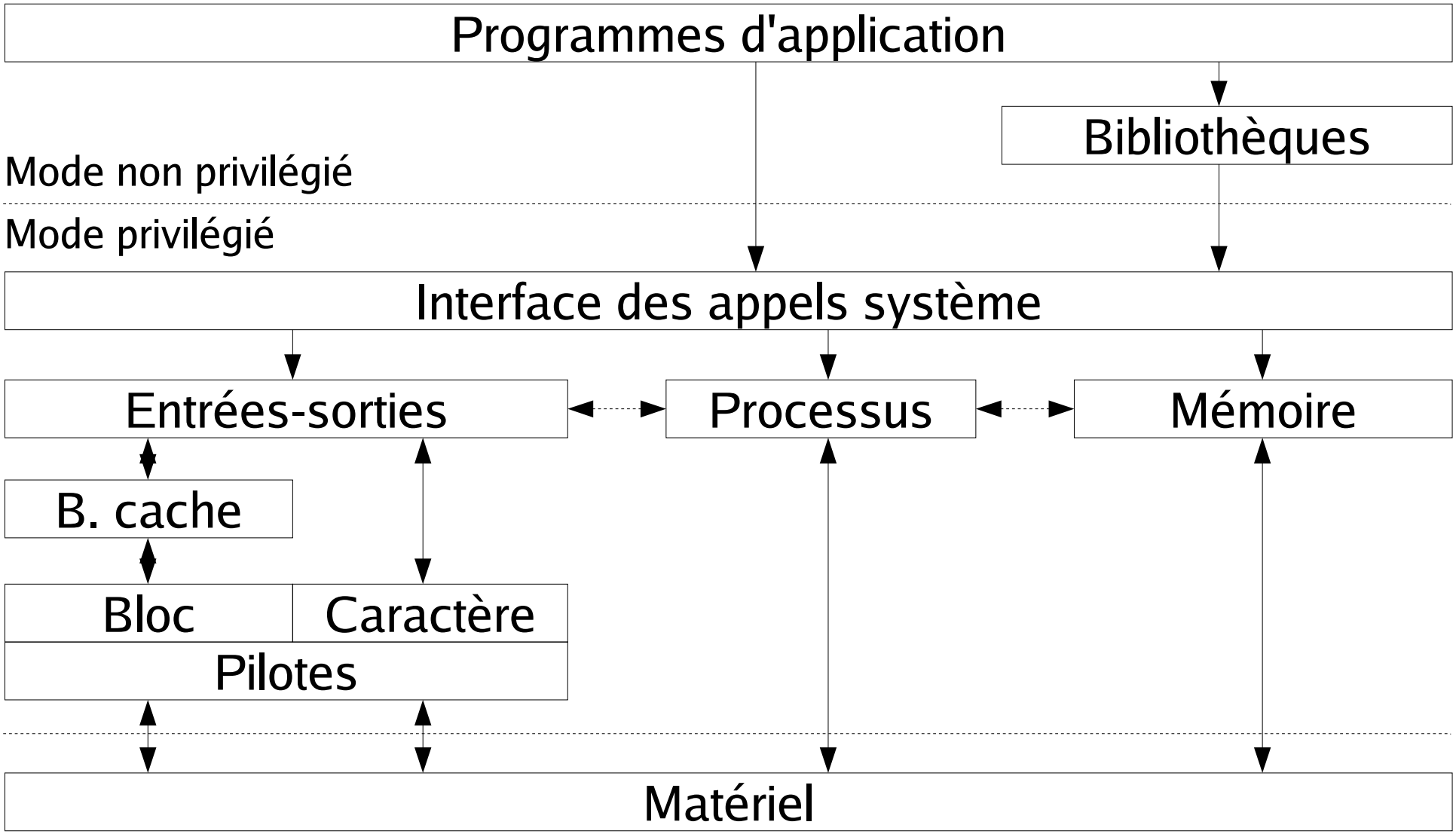
Système d'exploitation (3)

- En tant que machine virtuelle, le système fournit :
  - Une vue uniforme des entrées/sorties
  - Une mémoire virtuelle et partageable
  - La gestion sécurisée des accès
  - La gestion des processus
  - La gestion des communications inter-processus

Système d'exploitation (4)

- En tant que gestionnaire de ressources, le système doit permettre :
  - D'assurer le bon fonctionnement des ressources et le respect des délais
  - L'identification de l'usager d'une ressource
  - Le contrôle des accès aux ressources
  - L'interruption d'une utilisation de ressource
  - La gestion des erreurs
  - L'évitement des conflits

Structure d'un s.e. monolithique



Démarrage de l'ordinateur (1)

- La carte mère, lorsqu'elle est mise sous tension
  - Effectue ses tests internes (intégrés dans son « *chipset* »)
    - Bipe et/ou clignote en cas d'erreur
  - Détecte le processeur et active son initialisation

Démarrage de l'ordinateur (2)

- Le processeur, lorsqu'il est activé
  - Effectue ses tests internes
  - N'active qu'un seul de ses cœurs
  - Positionne le compteur ordinal de ce cœur à une adresse prédéfinie appelée « vecteur de réinitialisation » (« *reset vector* »)
    - Souvent 0xFFFFFFF0
    - Anciennement 0xFFF0, puis 0xFFFF0
  - Démarre l'exécution des instructions à cette adresse

Démarrage de l'ordinateur (3)

- L'adresse correspondant au vecteur de réinitialisation n'est plus en « bout de mémoire » sur les ordinateurs modernes
  - Elle correspond à une adresse dans la RAM
- La mémoire située à l'adresse du vecteur de réinitialisation contient un JMP vers l'adresse réelle de début du BIOS
  - Instruction de branchement rendue accessible à cette adresse par la carte mère, par configuration initiale de la MMU (« *Memory Management Unit* »)

Démarrage de l'ordinateur (4)

- Le BIOS :
  - Initialise un ensemble minimal de périphériques
  - Effectue des tests des principaux composants de la carte mère (« *Power-on self test* », ou POST)
    - Checksum de sa ROM, test carte graphique, etc.
  - Rend compte graphiquement du résultat des tests suivants
    - Mémoire, clavier, périphériques de stockage, etc.

Démarrage de l'ordinateur (5)

- Une fois ces tests effectués, le BIOS :
  - Charge éventuellement des pilotes de périphériques supplémentaires fournis par les *firmwares* (logiciels embarqués) de ces périphériques
  - Construit éventuellement en mémoire des tables de données recensant ces périphériques et les adresses de leur pilotes
    - Norme ACPI (« *Advanced Configuration and Power Interface* »), de 1996 à 2013
    - Norme UEFI (« *Unified Extensible Firmware Interface* »), depuis 2005

Démarrage de l'ordinateur (6)

- Une fois les périphériques identifiés, le BIOS :
  - Cherche sur les périphériques de stockage autorisés un bloc de démarrage de système d'exploitation (« *boot sector* » ou « *master boot record* », MBR)
    - S'il n'y en a pas, il s'arrête en affichant un message d'échec
  - Charge ce bloc dans la mémoire vive
    - Pour le MBR, 440 octets de code et 72 octets de table des partitions, chargés à l'adresse 0x7C00
  - Effectue un JMP au début de ce bloc

Démarrage de l'ordinateur (7)

- Le démarrage de l'ordinateur met en œuvre une chaîne d'actions visant à offrir modularité et flexibilité, en dépit du fait que les composants matériels sont « figés » par conception
  - Initialisation de composants non connus à l'avance au moyen de pilotes de périphériques « *Plug-and-play* » (PnP)
    - Question de la sécurité et de l'authentification
  - « Mystification » des composants respectant les normes les plus anciennes pour s'adapter aux normes plus récentes

Démarrage du système (1)

- Le bloc de démarrage chargé par le BIOS est soit celui d'un système particulier, soit celui d'un logiciel de démarrage multi-systèmes
- Dans ce dernier cas, c'est ce logiciel qui propose un menu permettant de choisir le bloc de démarrage qui sera chargé en mémoire et lancé à la place du logiciel de démarrage

Démarrage du système (2)

- Le bloc de démarrage chargé par le BIOS effectue le chargement du noyau (« *kernel* ») du système d'exploitation et l'exécute
- Le noyau, en premier lieu :
  - Active les pilotes primitifs dont il a besoin pour poursuivre son installation
    - Gestionnaires de bus et de périphériques de stockage
      - En s'appuyant éventuellement sur les tables construites par le BIOS
    - Systèmes de fichiers primitifs

Démarrage du système (3)

- Ensuite, le noyau :
  - Crée les structures de données internes correspondant aux services qu'il doit rendre
    - Table des processus, des CPU, des segments, etc.
  - Crée « à la main » un unique processus
    - Le « processus 0 »
    - Initialise « à la main » les attributs de ce processus
  - Active les différents CPU
  - « Rend la main » au processus 0, comme à la fin d'une interruption « normale »

Démarrage du système (4)

- Le « processus 0 » utilise les services du noyau pour se dupliquer et remplacer le code de son clone par le code d'un processus « normal » chargé à partir du système de fichiers
  - « Init », le « processus 1 »
    - Parfois remplacé par « systemd »
  - Création par « fork - exec », avant de disparaître
    - Permet de modifier et de paramétrer le démarrage du système sans toucher au noyau

Démarrage du système (5)

- Ensuite, le processus « Init », comme tout processus « normal » utilisant les services du noyau :
  - Crée en cascade tous les autres processus nécessaires au fonctionnement du système
    - Processus « démons » (« *daemon* »)
    - Leur nom est souvent terminé par la lettre « d » : « initd », « inetd », etc.
  - En tant qu'ancêtre commun à tous les processus, joue un rôle spécifique lorsqu'ils terminent

Démarrage du système (6)

- Le démarrage du système met en œuvre une chaîne d'actions visant à offrir modularité et flexibilité, en dépit du fait que le noyau est compilé
  - S'appuie sur les pilotes de périphériques connus du BIOS pour s'abstraire des spécificités matérielles
  - Démarrage du système de façon modulaire, sous forme de « démons » lancés par un unique processus initial, « init », lui-même facilement modifiable et paramétrable
    - « /etc/init.d/... »

Processus

- Un processus est une instance d'un programme en train de s'exécuter au sein d'un système d'exploitation
- Il est caractérisé, au sein du système, par un certain nombre d'attributs :
  - Pid, ppid, uid, gid, ... et autres [que nous reverrons]
  - Espace d'adressage propre
    - Constitué de « segments » : code, données initialisées constantes, données initialisées variables, données non initialisées, tas, pile

Espace d'adressage (1)

- La plupart des couches ISA considèrent la mémoire comme un espace linéaire et continu commençant de l'adresse 0 à l'adresse 2^{32}-1 ou 2^{64}-1
  - En pratique, on n'utilise pas plus de 44 fils d'adresses (adressage de 16 TéraMots)

Espace d'adressage (2)

- Pour héberger simultanément plusieurs processus (en plus du système), il faut découpler l'adressage physique de l'adressage vu par chacun de ces processus
- On utilise pour cela, souvent de façon combinée, deux dispositifs :
  - Mémoire virtuelle
  - Segmentation

Segmentation (1)

- La segmentation consiste à avoir autant d'espaces d'adressage que de zones mémoire d'usages différents
  - Adresses partent à partir de 0 pour chaque segment
  - Un « descripteur de segment » indique l'adresse de début du segment dans l'espace d'adressage virtuel
    - L'adresse physique est calculée à la volée en ajoutant le déplacement au sein du segment à l'adresse de début du segment

Segmentation (2)

- Un descripteur de segment est associé à chaque segment
  - Taille à l'octet près
  - Droits d'accès
    - Politique « w ^ x »
      - En fait, c'est plutôt un NAND
    - Pour économiser des descripteurs de segments, les données initialisées constantes sont parfois placées dans les segments de code
- Permettent de détecter les accès invalides
  - « *Segmentation fault* »

Segmentation (3)

- La distinction entre segments globaux et segments privés permet de partager facilement des zones mémoire entre processus :
  - Code et données du noyau
  - Code des bibliothèques
  - Zones de mémoire partagées entre processus
  - « *Mapping* » des fichiers en mémoire

Modes d'exécution (1)

- Les micro-architectures modernes implémentent nativement des mécanismes matériels permettant de distinguer entre plusieurs modes d'exécution
  - Mode non privilégié : accès restreint à la mémoire, interdiction d'exécuter les instructions d'entrées-sorties
  - Mode privilégié : accès à tout l'espace d'adressage et à toutes les instructions
    - Il peut en exister plusieurs : « *rings* » concentriques, par exemple hébergeant les pilotes de périphériques

Modes d'exécution (2)

- Les appels système s'exécutent en mode privilégié, pour pouvoir accéder à l'ensemble des ressources de la machine
- Les programmes d'application s'exécutent en mode non privilégié
  - Ne peuvent accéder au matériel sans passer par les routines de contrôle d'accès du système
- Le passage du mode non privilégié au mode privilégié ne peut se faire que de façon strictement contrôlée (« *traps* » et interruptions)

Interruptions (1)

- Les interruptions sont des événements qui, une fois reçus par le processeur, conduisent à l'exécution d'une routine de traitement adaptée
  - L'exécution du programme en cours est suspendue pour exécuter la routine de traitement
  - Analogue à un appel de sous-programme, mais de façon asynchrone

Interruptions (2)

- Il existe plusieurs types d'interruptions, identifiées par leur numéro
  - Numéro d'IRQ (« *Interrupt ReQuest* »)
- Ceci permet d'effectuer un « pré-tri » en appelant la « bonne » routine de traitement selon le périphérique qui a émis l'interruption
  - Configuration manuelle sur les anciens systèmes
  - Auto-configuration en mode « PnP »

Interruptions (3)

- Les interruptions peuvent être :
  - Asynchrones : interruptions « matérielles » reçues par le processeur par activation de certaines de ses lignes de contrôle
    - Gestion des périphériques
  - Synchrones : interruptions générées par le processeur lui-même :
    - Par exécution d'une instruction spécifique (« trap »)
      - Exemple : l'instruction `INT` de l'architecture x86
      - Sert à mettre en œuvre les appels système
    - Sur erreur logicielle (erreur d'accès mémoire, de calcul ...)
      - Sert à mettre en œuvre les exceptions

Interruptions (4)

- Lorsque le processeur accepte d'exécuter une interruption :
  - Il sauvegarde dans la pile l'adresse de la prochaine instruction à exécuter dans le cadre du déroulement normal
  - Il se sert du numéro de l'interruption pour indexer une table contenant les adresses des différentes routines de traitement (« vecteur d'interruptions »)
    - Accessible en lecture seule en mode non privilégié
  - Il se déroute à cette adresse
    - Passage en mode privilégié si le processeur en dispose

Interruptions (5)

- Dans les systèmes récents, le processeur met en place une pile dédiée pour traiter les appels afin d'éviter :
  - De mettre en danger le système si la pile utilisateur est prête à déborder
  - Qu'un pilote mal conçu détruise la pile du processus en cours d'appel système
  - Que l'utilisateur puisse récupérer des informations sensibles en analysant sa pile au retour d'appel

Appels système (1)

- Les arguments sont empilés dans la pile utilisateur
  - Tout comme un appel de fonction classique
- Un « trap » est effectué
  - Appel « INT / syscall » au lieu de « call »
  - ...Ou toute autre méthode selon le système
- Un appel système est donc coûteux !
  - Mécanisme prenant plusieurs centaines de cycles
  - Pas toujours optimal en termes de performance !

Appels système (2)

- Les appels système sont documentés en section 2 du manuel, ou en section 3p pour les appels POSIX
  - Exemple : « man 2 write »

- Norme POSIX : voir diapositives 22 à 24 du cours de Marc Zeitoun

Appels système et erreurs

- La gestion des erreurs est essentielle à la robustesse d'un programme
  - Elle est critique pour les programmes utilisant les interfaces d'accès au réseau, à cause de l'asynchronisme et des appels-système bloquants

- Voir diapositives 25 à 27 du cours de Marc Zeitoun

Architecture des services du système

- Conçus comme des ensembles homogènes
- Principes architecturaux identiques :
  - « Descripteurs » privés permettant d'identifier les ressources
  - Pointent vers des descripteurs globaux permettant d'assurer la cohérence de la ressource au niveau du système :
    - Gestion de l'atomicité / exclusion mutuelle
    - Gestion des droits d'accès

Atomicité

- Principe garantissant qu'un appel système à une ressource ne sera pas interrompu par un autre appel système sur la même ressource
  - Même si l'appel système s'effectue en plusieurs fois, avec endormissement du processus
  - L'appel a l'air de s'être déroulé de façon atomique
    - « a-tomos » : qui ne peut être coupé
- Exemple : deux processus qui écrivent en même temps dans un *pipe*
  - Celui qui a commencé terminera

Interface de gestion de fichiers

- Offre une vue sur le système de fichier
  - Pas accès aux structures internes de chaque système de fichier
  - Vue abstraite en termes d'actions élémentaires
    - Enchaînement d'appels grâce aux descripteurs

- Voir pages 28 à 63 des diapositives de Marc Zeitoun

La « bufferisation » des E/S (1)

- Les entrées-sorties représentent la majeure partie des appels système
- Les appels système individuels coûtent cher
- Il faut les « factoriser » : un même appel système doit regrouper plusieurs entrées/sorties demandées par l'utilisateur
  - Cette factorisation doit avoir lieu dans l'espace de l'utilisateur, avant l'appel système proprement dit

La « bufferisation » des E/S (2)

- À chaque descripteur de « haut niveau » doit correspondre une zone de stockage temporaire
  - Zone tampon (« *buffer* »)
- Les données en écriture sont stockées dans la zone avant leur utilisation
  - En lecture : lecture d'un bloc entier, puis consommation caractère par caractère
  - En écriture : accumulation des caractères puis écriture d'un bloc entier

La « bufferisation » des E/S (3)

- Les descripteurs de « haut niveau » sont les structures FILE
  - Pointeur de flot / flux (« *stream* »)
  - Voir « /usr/include/stdio.h »
    - Et ses descendants : « /usr/include/libio.h », etc.
  - Contiennent (entre autres) :
    - Des pointeurs vers le début des tampons en lecture et en écriture, vers la position courante qui leur correspond, et la taille des données qu'ils contiennent
    - Le descripteur de fichier associé

La « bufferisation » des E/S (4)

- Utilisation de fonctions spécifiques :
  - fopen(), fread(), fseek(), fprintf(), fscanf(), etc.
- Création d'un flot à partir d'un descripteur :
  - fdopen()
- Des flots standards correspondent aux descripteurs standards de bas niveau
  - stdin : descripteur 0
  - stdout : descripteur 1
  - stderr : descripteur 2

Types de « bufferisation » (1)

- Question importante de mise en œuvre :
  - À quel moment vidanger un tampon d'écriture ?
- Réponse :
  - Idéalement : le plus tard possible, pour économiser le plus d'appels système possible
  - En pratique : ça dépend des cas !
    - Sous-question : est-ce que l'utilisateur a vraiment besoin des données en question tout de suite ?

Types de « bufferisation » (2)

- Trois types de « bufferisation » :
  - « Bufferisation » bloc : lorsque le flot de sortie correspond à un fichier
  - « Bufferisation » ligne : lorsque le flot de sortie correspond à un terminal
    - La vidange est déclenchée par le caractère « \n »
  - Pas de « bufferisation »
- Demande explicite de vidange : fflush()

Types de « bufferisation » (3)

buffer1.c

```
#include <stdio.h>
main ()
{
  printf ("Je passe par A...");
  sleep  (2);
  printf (" puis par B\n");
  sleep  (2);
  printf ("Je passe par C aussi...");
  sleep  (2);
  *((int *) NULL) = 42;                  /* Ouh le vilain SEGFAULT ! */
  printf (" puis par D\n");
  return (0);
}
```

Types de « bufferisation » (4)

buffer2.c

```
#include <stdio.h>
main ()
{
  printf ("Je passe par A...");
  fprintf (stderr, "Je passe par A tout pareil...");
  sleep  (2);
  printf (" puis par B\n");
  fprintf (stderr, " puis par B tout pareil\n");
  sleep  (2);
  printf ("Je passe par C aussi...");
  fprintf (stderr, "Je passe par C aussi tout pareil...");
  sleep  (2);
  *((int *) NULL) = 42;                  /* Ouh le vilain SEGFAULT ! */
  printf (" puis par D\n");
  fprintf (stderr, " puis par D tout pareil\n");
  return (0);
}
```

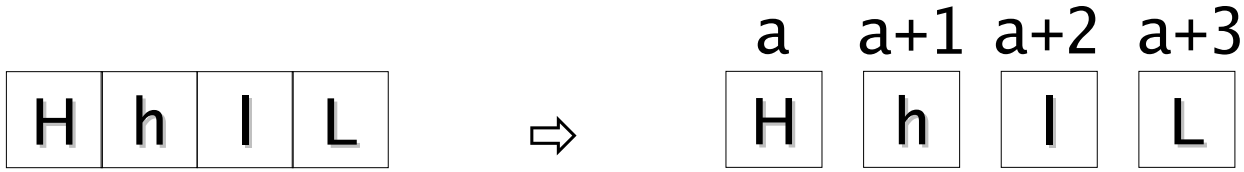
- À tester avec « > » et/ou « 2> », puis fflush()

Types de « bufferisation » (5)

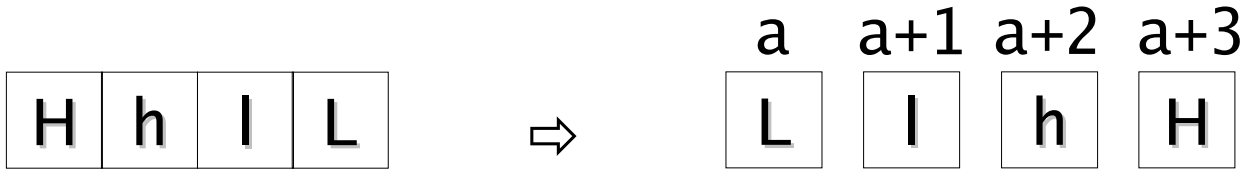
- On ne débogue pas « au printf » mais « au fprintf (stderr) » !

Le « boutisme » (1)

- Pour stocker un mot machine en mémoire, il faut savoir dans quel ordre stocker les octets du mot dans les cases consécutives de la mémoire :
  - Octet de poids fort en premier (« *big endian* »)



  - Octet de poids faible en premier (« *little endian* »)



Le « boutisme » (2)

- La même question se pose lorsqu'on écrit une donnée sous forme binaire dans un fichier
  - Sous forme caractère, cela ne pose pas de problème, mais la densité d'information est faible
    - Ne pas confondre « write() » et « printf() » !
- Fonctions normalisées de conversion :
  - htons(), ntohs(), htonl(), ntohl()
    - « *Host to Network* » et « *Network to Host* »
  - C'est le « *little endian* » qui a gagné sur Internet

Le « boutisme » (3)

ecriture.c

```
#include <stdio.h>
#include <fcntl.h>
main ()
{
  int    fd;
  FILE * fp;
  int    i = 42;

  fd = open ("/tmp/brol1.txt", O_WRONLY | O_CREAT | O_TRUNC, 0600);
  write (fd, &i, sizeof (i));
  close (fd);
  fp = fopen ("/tmp/brol2.txt", "w");
  fprintf (fp, "%d", i);
  fclose (fp);
  return (0);
}
```

La « plomberie » des descripteurs (1)

- La « plomberie » des descripteurs repose sur deux appels système : « close() » et « dup() »
  - Conception modulaire analogue au couple « fork() / exec() » pour les processus
  - Fusionnés dans l'appel « dup2() »
- Permet la redirection d'un flot en remplaçant un descripteur par un autre
- Voir pages 58 à 62 des diapositives de Marc Zeitoun

La « plomberie » des descripteurs (2)

dup.c

```
#include <stdio.h>
#include <fcntl.h>
main ()
{
  int    fd;

  printf ("Et hop, sur le terminal !\n");
  fd = open ("/tmp/brol.txt", O_WRONLY | O_CREAT | O_TRUNC, 0600);
  dup2 (fd, 1);
  printf ("Et hop, dans le fichier !\n");
  write (fd, "Et hop, dans le fichier aussi !\n", 32);
  return (0);
}
```

- À tester avec « > /tmp/brol2.txt »

Droits des processus (1)

- Tout processus lancé au nom d'un utilisateur bénéficie des droits d'accès de cet utilisateur aux ressources du système
  - À son lancement, le processus mémorise le UID (« *User ID* ») et GID (« *Group ID* ») de l'utilisateur l'ayant lancé
    - Par héritage des UID et GID de son processus père
    - Commande « newgrp » qui permet d'empiler un nouvel interpréteur avec un autre identifiant de groupe

Droits des processus (2)

- Dans certains cas, on peut vouloir donner au processus les droits du créateur du programme exécutable
- Exemple de la commande « su » :
  - Un utilisateur doit lancer un programme qui lui donne les droit d'un autre utilisateur
  - Seul le super-utilisateur a le droit de changer d'identité, donc un programme de changement d'identité doit s'exécuter au nom du super-utilisateur

Droits des processus (3)

- Le système maintient donc deux UID et GID :
  - Réels : ceux de l'utilisateur ayant lancé le processus
  - Effectifs : ceux de l'utilisateur ayant donné ses droits sur l'exécutable
- Un processus peut modifier ses droits au moyen d'appels système
  - « setuid() », « setgid() », « seteuid() », « setegid() »
  - Permet de perdre des privilèges, pas d'en gagner !

Droits des processus (4)

- L'attribution de droits effectifs est réalisée au moyen du « bit s » des droits de fichiers
  - Souvent donné à des programmes au nom du super-utilisateur
    - Ou à un utilisateur maintenant des fichiers collectifs : utilisateur fictif gérant une base de données, etc.
      - Moins utilisé avec l'arrivée des services réseau
  - Risque important pour la sécurité du système
    - Le « bit s » peut être interdit sur certains systèmes de fichiers (paramètre « nosuid » au montage)

Vie et mort des processus (1)

- Sous Unix, un processus ne peut être créé... que par un autre processus !
  - Problème de l'œuf et de la poule, résolu au démarrage du système par la création « à la main » du « processus 0 »

Vie et mort des processus (2)

- Tout processus est identifié par un numéro unique, le PID (« *Process ID* »)
  - Il ne peut y avoir deux processus vivants de même numéro
  - Récupérable par « getpid() »
- Tout processus a un unique parent, qui l'a créé
  - Récupérable par « getppid() » (« *Parent PID* »)
  - Un parent peut créer de multiples fils au cours de son histoire

Vie et mort des processus (3)

- Deux primitives élémentaires :
  - « fork() » : un processus demande à se dupliquer
  - « exec() » : un processus demande à remplacer ses segments par ceux d'un exécutable présent dans le système de fichiers
- À la différence des appels uniques de type « spawn() » d'autres systèmes
  - Un appel unique offre bien moins de fonctionnalités
  - Exemple de la « philosophie d'Unix »

fork (1)

- Appelé par le processus lui-même
- En cas de succès, le système crée un nouveau processus « fils », clone du processus « père »
  - Copie de la mémoire du père : segments, pile, ...
    - Mais pas partage en écriture !
    - Optimisation système : partage en lecture des pages des segments et copie en écriture (« *copy-on-write* »)
  - Copie des descripteurs des ressources systèmes
    - Descripteurs des fichiers ouverts, etc.

fork (2)

- Un seul processus appelle « fork() », et deux processus en reviennent !
  - Comment les distinguer ?
- Valeur de retour de fork() :
  - 0 : je suis le fils
  - >0 : je suis le père, et la valeur est le PID du fils
    - Un père n'a pas d'autre moyen de le connaître
  - -1 : erreur

fork (3)

- Qui « revient » en premier ?
  - On n'a aucun moyen de le savoir !
    - Ce n'est pas spécifié par la norme
  - C'est à l'ordonnanceur de gérer les processus
    - Comportement variable d'un système à l'autre
  - Ce n'est en fait pas une question pertinente...
    - Si cela a une importance, alors mise en place de mécanismes de synchronisation explicite au sein des programmes

fork (4)

fork.c

```
#include <stdio.h>
#include <unistd.h>
main ()
{
  pid_t  pid;

  pid = fork ();
  if (pid == 0)
    printf ("Le fils parle\n");
  else if (pid > 0)
    printf ("Le pere parle\n");
  return (0);
}
```

- À tester avec « while true ; do ./fork ; done »

fork (5)

- Les liens de parenté des processus forment un arbre (généalogique)
  - Visible avec la commande « pstree »
- Père et fils diffèrent sur leurs :
  - PID, PPID
  - Statistiques temporelles, verrous, signaux en attente, ... [nous le reverrons]
- La commande « ps -l » fournit nombre de ces informations

Terminaison des processus (1)

- Un processus se termine normalement en :
  - Appelant « exit() » ou « _exit() », à tout moment
  - Effectuant un « return() » dans la fonction « main() »
- La fonction « exit() » :
  - Appelle les fonctions enregistrées au moyen de la fonction « atexit() »
  - Vidange les tampons des flots ouverts
  - Continue comme _exit()

Terminaison des processus (2)

- L'appel système « _exit() » :
  - Ferme les descripteurs ouverts
  - Transfère au « processus 1 » la parenté des fils du processus et envoie un signal SIGCHLD au processus père du processus
  - Termine le processus
- Un processus peut se terminer anormalement de façon asynchrone, sur réception d'un signal
  - Déclenche l'appel à _exit()

wait (1)

- L'appel système wait() permet à un père de connaître le code de retour (statut) de l'un de ses fils décédés

  pid_t wait (int * statut)

- Il existe en fait plusieurs fonctions :

  wait(), waitpid (), waitid(), wait3(), wait4()

  - Permettent de spécifier un PID particulier ou non
  - Bloquantes ou non

wait (2)

- Plusieurs macros permettent d'extraire des informations du statut du fils :
  - WEXITSTATUS(statut) : code de retour du processus, tel que renvoyé par _exit() ou return()
  - WIFEXITED(statut) : vrai si le processus s'est terminé normalement
  - WIFSIGNALED(statut) : vrai si le processus a été terminé par un signal
  - WTERMSIG(statut) : numéro du signal de fin
  - ...

wait (3)

- Le caractère bloquant de wait() permet de mettre en œuvre une synchronisation entre le père et le fils
  - Le père s'endort tant qu'un de ses fils n'a pas terminé
  - Exemple des interpréteurs de commandes :
    - Faire un « & » demande moins de travail que d'attendre la terminaison du fils pour redonner la main à l'interpréteur

Processus zombies (1)

- Un processus, lorsqu'il décède, n'est pas directement supprimé de la table des processus du système
  - Il faut permettre à son père de récupérer son code de retour, sur la base de son PID
  - Cette information ne peut être stockée chez le père
    - Problème de taille des structures de stockage
    - On ne sait quel ancêtre la lira effectivement

Processus zombies (2)

- Tant qu'un wait() n'a pas ciblé le processus, il reste dans l'état « zombie » ou « défunt »
- Risque d'accumulation de zombies dans la table des processus si le père ne s'en occupe pas
  - Le nombre de processus qu'un père peut créer est limité, pour ne pas engorger la table des processus
  - Le « processus 1 » sert de « fossoyeur » de tous les processus dont il hérite

Processus zombies (3)

zombie.c

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
main ()
{
  pid_t     pid;
  int       i;

  for (i = 0; i < 60; i ++) {
    if ((pid = fork ()) <= 0) /* Crée des fils          */
      return (pid);           /* Qui terminent de suite */
  }
  sleep (15);
  do {
    wait (NULL);
    printf ("RIP !\n");
  } while (errno != ECHILD);

  return (0);
}
```

États des processus

- Les processus, une fois créés, prennent successivement de nombreux états :
  - Principaux : actif (en mode utilisateur), en attente du processeur, en attente d'une autre ressource, zombie
  - Secondaires : actif (en mode noyau), « swappé » sur disque
- Diagramme d'états : voir page 73 des diapositives de Marc Zeitoun

exec (1)

- Remplace le contenu d'un processus par un nouveau programme exécutable
  - En fait, ensemble de six fonctions :

    exec{l,v}{Ø,p,e} (...)

  - Toutes construites au-dessus de l'appel système execve()

exec (2)

- Le processus conserve ses :
  - PID, PPID
  - UID et GID réels
  - Répertoire courant, signaux en attente
  - Descripteurs de fichiers ouverts
    - Sauf si drapeau « close-on-exec » du fichier positionné au préalable au moyen de « fcntl() »
  - ...

exec (3)

- Le processus voit se modifier ses :
  - Segments mémoire
  - UID et GID effectifs, si le fichier exécutable possède le « bit s »
  - ...

exec (4)

- On ne doit jamais retourner d'un appel système exec()
  - Sinon, c'est que l'appel système à échoué !
  - Voir la cause de l'erreur avec errno

Environnement et `main` (1)

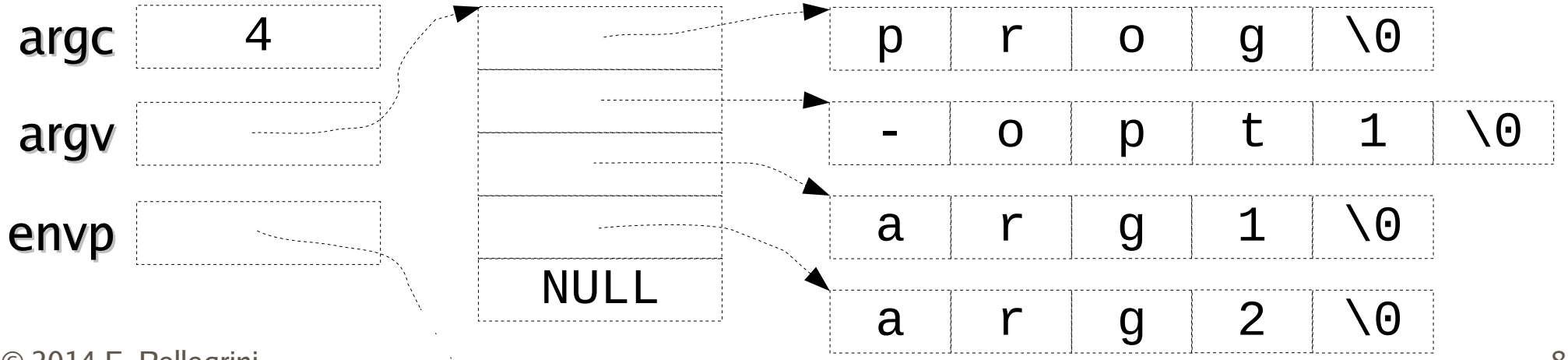
- La fonction `main` retourne un `int`
  - Code de retour du programme
    - Pourra être lu par le père ou l'ancêtre
- La fonction main possède trois arguments
  - `argc`, nombre de paramètres passés à la commande, y compris son propre nom
  - `argv`, tableau des arguments de la commande
  - `envp`, tableau des variables d'environnement héritées du processus père

Environnement et `main` (2)

- Format complet de la fonction `main`

```
int
main (
int             argc,   /* Nombre d'arguments y compris le nom      */
char *          argv[], /* Tableau des chaînes des arguments        */
char *          envp[]) /* Tableau des chaînes des variables d'env. */
{
  ...
}
```

```
% prog -opt1 arg1 arg2
```



Environnement et main (3)

- Exemple d'usage des paramètres de main

environnement.c

```
int
main (
int             argc,   /* Nombre d'arguments y compris le nom      */
char *          argv[], /* Tableau des chaînes des arguments        */
char *          envp[]) /* Tableau des chaînes des variables d'env. */
{
  int       i;

  for (i = 0; i < argc; i ++)
    printf ("Argument %d : \"%s\"\n", i, argv[i]);

  for (i = 0; envp[i] != NULL; i ++)
    printf ("Environnement %d : \"%s\"\n", i, envp[i]);

  return (0);
}
```

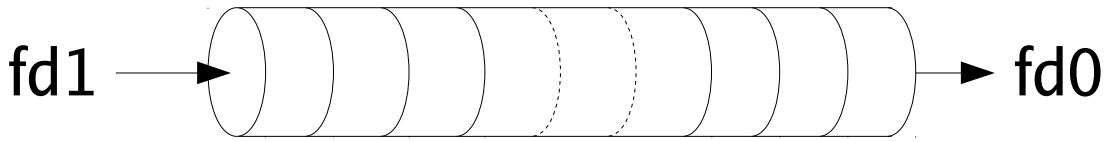
Environnement et `main` (4)

- Les petits programmes se servent rarement des variables d'environnement
  - On ne spécifie pas `envp` dans la définition de `main`
  - Pas de problème car c'est le dernier argument, stocké au fond de la pile d'appel
- Format usuel de la fonction `main`

```
int
main (
int             argc,     /* Nombre d'arguments y compris le nom  */
char *          argv[])   /* Tableau des chaînes des arguments    */
{
  ...
}
```

Tubes de communication (1)

- Les tubes de communication anonymes (« *pipes* ») sont des objets système permettant l'échange de données entre processus ayant un ancêtre commun
- Un *pipe* est un objet système constitué :
  - D'un descripteur d'écriture
  - D'un descripteur de lecture
  - D'un tampon intermédiaire de capacité inconnue mais finie



Tubes de communication (2)

- Un processus crée un tube anonyme au moyen de l'appel système :

  int pipe (int fd[2])

- Cet appel système crée l'objet *pipe* et donne accès à deux descripteurs :
  - fd[0] : lecture à partir du tube
    - 0 = « stdin » = lecture / read
  - fd[1] : écriture dans le tube
    - 1 = « stdout » = écriture / write

Utilisation des tubes (1)

- Pour qu'un tube anonyme soit partagé entre deux processus, il faut qu'il ait été créé par un ancêtre commun
- Chacun des deux processus ferme le descripteur dont il n'a pas besoin
  - Économie de ressources système
  - Surtout : évite les interblocages !

Utilisation des tubes (2)

pipe.c

```
int
main ()
{
  int       fd[2];
  int       pid;
  int       c;

  pipe (fd);
  if ((pid = fork ()) == 0) {  /* Le fils est l'écrivain   */
    close (fd[0]);             /* Donc il ferme la lecture */
    write (fd[1], "A", 1);
  }
  else if (pid > 0) {          /* Le pere est le lecteur   */
    close (fd[1]);             /* Donc il ferme l'ecriture */
    read (fd[0], &c, 1);
    printf ("Le pere a recu \"%c\".\n", c);
  }

  return (0);
}
```

Sémantique des tubes (1)

- Unix cherche à offrir une sémantique unique aux différents types de descripteurs
- Cependant, les tubes ne sont pas des fichiers comme les autres :
  - Modèle de programmation « producteur / consommateur »
    - Processus travaillant de façon partiellement asynchrone
  - Taille de tampon finie (et petite !)
    - Pas nécessairement de correspondance entre la taille des données à lire / écrire et la capacité restante du *pipe*

Sémantique des tubes (2)

- Sémantique de read() :
  - S'il existe des octets dans le tube, read(..,x) renvoie au plus x octets
    - Différence avec les fichiers classiques
  - S'il n'y a plus d'octets dans le tube et plus d'écrivain potentiel, read() renvoie 0
    - Analogue à une fin de fichier classique
  - S'il n'y a plus d'octets dans le tube mais au moins un écrivain potentiel, read() bloque et endort le processus jusqu'à retrouver l'un des cas ci-dessus
    - Ne nécessite pas de prise en charge particulière

Sémantique des tubes (3)

- Sémantique de write() :
  - S'il reste au moins un lecteur potentiel, l'appel système write (...,x) ne retournera qu'une fois qu'il aura effectivement écrit x octets dans le tube
    - Analogue à l'écriture dans un fichier classique
    - Peut conduire à l'endormissement du processus s'il restait moins de x octets libres dans le tube
    - Ne nécessite pas de prise en charge particulière
    - Respect du principe d'atomicité

Sémantique des tubes (4)

- S'il n'existe plus aucun lecteur potentiel, l'appel système write (...,x) retournera une erreur mais, avant cela, un signal SIGPIPE sera envoyé au processus
  - Écrire dans un tube sans lecteur est une anomalie qui sort du cadre du schéma de communication normal
  - Puisque plus personne n'est intéressé à ce que le processus a à dire, autant qu'il termine
    - Comportement par défaut à la réception d'un signal SIGPIPE
    - Exemple d'un « ls | more » où l'utilisateur termine le « more »
  - Simplifie l'écriture des programmes
    - Gère le cas le plus courant
    - Sinon, il faut empêcher le signal SIGPIPE de tuer le processus

Tubes nommés (1)

- Les tubes nommés (« *named pipes* ») servent à la communication entre des processus qui n'ont pas d'ancêtre commun
  - Ancêtre local des *sockets* réseau
- Un tube nommé est créé au moyen de l'appel :

  mkfifo (char * path, mode_t mode)
- Doit avoir été ouvert par au moins un lecteur et un écrivain pour réaliser une communication
  - Par défaut, l'ouverture du tube nommé bloque jusqu'à ce que l'autre extrémité ait été ouverte

Tubes nommés (2)

mkfifo.c

```
int
main ()
{
  int       fd;
  int       pid;
  int       c;

  mkfifo ("/tmp/brol.fifo", 0600);
  if ((pid = fork ()) == 0) {  /* Le fils est l'écrivain */
    fd = open ("/tmp/brol.fifo", O_WRONLY);
    write (fd, "A", 1);
  }
  else if (pid > 0) {          /* Le pere est le lecteur */
    fd = open ("/tmp/brol.fifo", O_RDONLY);
    read (fd, &c, 1);
    printf ("Le pere a recu \"%c\".\n", c);
  }

  return (0);
}
```

Signaux (1)

- De nombreux événements se produisent de façon asynchrone et demandent un traitement adéquat
  - Arrivée d'un bloc disque, d'une frappe clavier, ...
- Au niveau du matériel, ces événements donnent lieu à des interruptions
- Lorsqu'il reçoit une interruption, le processeur se déroute et exécute une routine de traitement adaptée
  - Passage de l'univers logiciel à l'univers matériel

Signaux (2)

- Lorsque l'événement concerne un processus et non le système en tant que tel, il faut déléguer son traitement au processus lui-même
- Un signal est la représentation d'un événement logiciel destiné à être traité par un processus
  - Comme pour les interruptions, il existe plusieurs types de signaux, identifiés par des numéros
  - On leur donne également des noms d'usage :
    - SIGTERM, SIGKILL, SIGSEGV, SIGPIPE, etc.
    - Voir « /usr/include/signal.h » et « /usr/include/bits/signum.h »

Signaux (3)

- Le service de gestion des signaux (la « signalerie ») contient des appels systèmes permettant :
  - De déclarer une routine de traitement pour un signal de numéro donné
    - Routine fournie par l'utilisateur
      - SIGKILL et SIGSTOP ne pourront voir leur comportement modifié par l'utilisateur
    - Traitement par défaut : ignorer, tuer le processus
      - Avec génération d'un fichier *core* ou pas
      - On ne « tue » pas un processus, on lui demande de se suicider

Signaux (4)

- De masquer ou d'accepter certains types de signaux
  - SIGKILL et SIGSTOP ne seront jamais masqués ni ignorés
- D'envoyer des signaux
  - À un autre processus ou à soi-même
  - Envoi limité aux processus que l'on possède

- Voir pages 133 à 150 des diapositives de Marc Zeitoun

Les tâches légères (1)

- Les processus Unix sont des objets « lourds »
  - Liés à de nombreux autres objets du système :
    - Table des fichiers ouverts
    - Table des descripteurs de segments mémoire
    - Liste des processus fils
    - Etc.
- Leur création et leur destruction sont coûteuses

Les tâches légères (2)

- Le partage d'informations entre processus est très coûteux
  - Les tubes, tubes nommés, etc., nécessitent une double recopie d'informations entre l'espace utilisateur et l'espace noyau
    - Le mécanisme de partage de segments mémoire (appels System V « shm* ») ne permet pas de gérer efficacement les pointeurs
- L'utilisation du parallélisme ne peut se faire qu'à « gros grain »
  - Nécessité d'un mécanisme à « grain fin »

Les tâches légères (3)

- Les « *threads* » (fils d'exécution) sont des tâches légères s'exécutant au sein du même espace d'adressage
  - Un processus Unix est un conteneur hébergeant le *thread* associée à la fonction main()

Les tâches légères (4)

- Les *threads* partagent tout, à part :
  - Leur identifiant
    - Unique au sein du processus
  - Leur état d'exécution
    - Registres, pile, compteur ordinal, ordonnancement, ...
  - Leur ensemble de signaux en attente et bloqués
  - Leur valeur d'errno
    - Ce n'est en fait pas une variable globale !
- Voir pages 85 à 94 des diapositives de Marc Zeitoun

Parallélisation avec les *threads* (1)

- On crée un *thread* avec « thread_create() »
  - Réalise un « *spawn* » alors que fork() duplique

threads.c (version 1)

```
#include <stdio.h>
#include <pthread.h>
void * f (void * ptr)
{
  printf ("Je suis le thread de rang %d\n", (int) ptr);
  return (NULL);
}
main ()
{
  pthread_t t[4];

  for (int i = 0; i < 4; i ++)
    pthread_create (t + i, NULL, f, (void *) i);
  printf ("Je suis le maitre\n");
  return (0);
}
```

Parallélisation avec les *threads* (2)

- Le code de main() s'achève sur un exit()
- Le code des autres *threads* s'achève sur un « pthread_exit() »
- Synchronisation par « pthread_join() »

threads.c (version 2)

```
main ()
{
  pthread_t t[4];

  for (int i = 0; i < 4; i ++)
    pthread_create (t + i, NULL, f, (void *) i);
  printf ("Je suis le maitre\n");
  for (i = 0; i < 4; i ++)
    pthread_join (t[i], NULL);
  return (0);
}
```

Compilation et *threads* (1)

- Les *threads* peuvent tirer parti de la mémoire partagée pour se synchroniser

spinlock.c (version 1)

```
#include <stdio.h>
#include <pthread.h>
int verrou = 0;
void * f (void * ptr)
{
  sleep (1);
  verrou = 1;
  return (NULL);
}
main ()
{
  pthread_t t;
  pthread_create (&t, NULL, f, NULL);
  while (verrou == 0) ;
  printf ("J'ai fini\n");
}
```

- Bogue avec « gcc -O2 »

Compilation et *threads* (2)

- Les compilateurs optimisent le code séquentiel qui leur est fourni
  - Réordonnancement d'instructions indépendantes, factorisation des accès mémoire, etc.
- La factorisation des accès mémoire modifie la sémantique du code
- Nécessité d'informer le compilateur du risque d'accès concurrents / asynchrones
  - Mot-clé « volatile » de la norme C89

spinlock.c (version 2)

```
volatile int verrou = 0;
```

Exclusion mutuelle (1)

- Lorsque plusieurs *threads* accèdent en écriture à la même zone de données, il y a conflit

mutex1.c

```
static int s = 0;
void * f (void * ptr)
{
  for (int i = 0; i < 100000; i ++)
    s = s + 1;
}
main ()
{
  pthread_t t[4];

  /* Ici : pthread_create() et _join() de 4 threads "f" */

  printf ("Au total, j'en compte %d\n", s);
  return (0);
}
```

Exclusion mutuelle (2)

- Pour éviter les conflits, il faut mettre en place une exclusion mutuelle au niveau des sections critiques où les accès simultanés sont interdits

mutex2.c

```
static pthread_mutex_t m;
void * f (void * ptr)
{
  for (int i = 0; i < 100000; i ++) {
    pthread_mutex_lock (&m);
    s = s + 1;
    pthread_mutex_unlock (&m);
  }
}
main ()
{
  pthread_mutex_init (&m, NULL);
  /* Ici : pthread_create() et _join() de 4 threads "f" */
  pthread_mutex_destroy (&m);
}
```

Exclusion mutuelle (3)

- Seul le *thread* qui a verrouillé un *mutex* peut le déverrouiller
  - Sinon, il serait facile d'« éjecter » un *thread* d'une section critique pour y entrer à son tour...
- Les *mutex* sérialisent l'exécution du code
  - Perte de performance pouvant être importante lorsque le nombre de *threads* augmente
  - Mieux vaut travailler le plus possible sur des copies privées des variables et n'effectuer la mise à jour globale qu'à la fin

Barrière de synchronisation (1)

- Lors de l'exécution d'un code parallèle, on peut vouloir que tous les *threads* aient fini d'exécuter une partie du code avant d'exécuter la suite
  - Par exemple : remplissage parallèle d'un tableau avant d'entamer le traitement collectif de ce tableau
- Cette fonctionnalité est offerte par les barrières de synchronisation

Barrière de synchronisation (2)

- La mise en œuvre d'une barrière de synchronisation nécessite :
  - Une structure de donnée dédiée, pour :
    - Savoir combien de *threads* on doit attendre avant de les laisser passer
    - Compter le nombre de *threads* déjà arrivées à la barrière
    - Savoir lesquelles, pour les réveiller une fois le nombre atteint
  - Une fonction dédiée, que les *threads* appellent pour s'enregistrer auprès de la barrière
    - Plus celles pour gérer la création et la destruction de la barrière

Barrière de synchronisation (3)

- La taille de la barrière ne peut être supérieure au nombre de *threads* lancés, mais peut être inférieure

barriere.c

```
static pthread_barrier_t b;
void * f (void * ptr)
{
  printf ("Avant : %d\n", (int) ptr);
  pthread_barrier_wait (&b);
  printf ("Apres : %d\n", (int) ptr);
}
main ()
{
  pthread_t t[4];
  pthread_barrier_init (&b, NULL, 4);
  /* Ici : pthread_create() et _join() de 4 threads "f" */
  pthread_barrier_destroy (&b);
}
```

Variable de condition (1)

- Pour mettre en œuvre un mécanisme de type « producteur-consommateur », il faut pouvoir :
  - Endormir les *threads* consommatrices lorsqu'il n'y a plus de données à consommer et les réveiller lorsque des données sont de nouveau disponibles
  - Endormir les *threads* productrices lorsqu'il n'y a plus de place pour les stocker les réveiller lorsque des places sont de nouveau disponibles
- Dualité :
  - Des causes d'endormissement
  - Des « ressources » : les données et les places

Variable de condition (2)

- Les *mutex* ne permettent pas de réveiller des *threads* lorsqu'un événement est généré par un autre *thread*
- Nécessité d'un « dortoir » dans lequel les *threads* s'enregistrent pour être réveillés sur commande

Variable de condition (3)

- Les variables de condition sont des structures de données adossées à un *mutex*
- Elles permettent à un *thread* :
  - De s'endormir en attente d'un réveil
    - « pthread_cond_wait() » : attend d'être réveillé
    - « pthread_cond_timedwait() » : attend un délai maximum d'être réveillé
  - De réveiller au moins une ou bien tous les *threads* enregistrés auprès de la variable
    - « pthread_cond_signal () » : réveille au moins un *thread*
    - « pthread_cond_broadcast() » : réveille tout le monde

Variable de condition (4)

- Comme plusieurs *threads* sont réveillés, la condition pour laquelle un *thread* a été réveillé peut donc ne plus être valide et doit donc être testée à nouveau par chaque *thread* réveillé
- C'est pour cela que :
  - La variable de condition est adossée à un mutex
  - Le mutex doit être verrouillé avant de s'endormir
- Ainsi, un *thread*, en se réveillant, a la main sur la ressource qu'il doit tester, sans risque de conflit d'accès

Sémaphore (1)

- La gestion d'un mécanisme de type « producteur / consommateur » peut être conceptualisée de la façon suivante :
  - Les processus qui veulent une ressource s'adressent au « guichet » d'un « entrepôt »
  - S'il y a une ressource disponible, elle disparait des comptes de l'entrepôt
    - Le processus peut la prendre dans l'entrepôt
  - S'il n'y a pas de ressource disponible, les processus patientent au guichet
    - Attente passive, pour économiser des ressources

Sémaphore (2)

- Un sémaphore est une structure de données articulée autour d'un compteur
  - Modélise le nombre de ressources disponibles
  - Initialisé à une valeur positive ou nulle
- Les sémaphores sont gérés par les appels :
  - « sem_init() » : création et initialisation d'un sémaphore
  - « sem_destroy() » : détruit un sémaphore
  - « sem_wait() » : demande une ressource
  - « sem_post() » : rajoute une ressource

Sémaphore (3)

- Lorsqu'un *thread* demande une ressource :
  - Si le compteur n'est pas à zéro, il est décrémenté
  - S'il est a zéro, le *thread* demandeur est endormi
- Lorsqu'un *thread* apporte une ressource :
  - Le compteur est incrémenté
  - S'il était à zéro, les *threads* endormis sont réveillés
    - L'un d'eux pourra la consommer
- Un sémaphore encapsule une variable de condition et un verrou sur le compteur

Sémaphore (4)

- Mise en place d'un mécanisme de type « producteur-consommateur » avec :
  - Un sémaphore comptant les biens et un autre pour les emplacements libres
  - Un verrou pour accéder au stock

prod-cons.c (partie 1)

```
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>

int             numero = 0;  /* Numero de la prochaine pièce  */
int             niveau = 0;  /* Niveau de stock               */
int             stock[7];    /* Zone d'entrepot du stock      */
pthread_mutex_t verrou;      /* Verrou de l'entrepot du stock */
sem_t           pieces;      /* Nombre de pieces en stock     */
sem_t           libres;      /* Nombre de places libres       */
```

Sémaphore (5)

- Le producteur attend une place libre avant de produire puis réveille les consommateurs

prod-cons.c (partie 2)

```
void
producteur (void * r)
{
  while (1) {
    int i, n;
    for (i = rand () % 3; i >= 0; i --) {
      sem_wait (&libres);  /* Attend une place libre */
      pthread_mutex_lock (&verrou);
      stock[niveau ++] = n = rang ++;
      printf ("%d: donne %d, il y en a   %d\n", (int) r, n, niveau);
      pthread_mutex_unlock (&verrou);
      sem_post (&pieces);  /* Reveille ceux qui veulent des pieces */
    }
    sleep (1);
  }
}
```

Sémaphore (6)

- Le consommateur attend une pièce avant de la prendre réveille les producteurs

prod-cons.c (partie 3)

```
void
consommateur (void * r)
{
  while (1) {
    int i, n;
    for (i = rand () % 3; i >= 0; i --) {
      sem_wait (&pieces);
      pthread_mutex_lock (&verrou);
      n = stock[-- niveau];
      printf ("%d: prend %d, il en reste %d\n", (int) r, n, niveau);
      pthread_mutex_unlock (&verrou);
      sem_post (&libres);
    }
    sleep (1);
  }
}
```

Sémaphore (7)

- Avant de lancer les *threads*, il faut initialiser les structures avec les valeurs adéquates

prod-cons.c (partie 4)

```
int
main ()
{
  int       i;
  pthread_t t[5];             /* 5+1 threads */

  pthread_mutex_init (&verrou, NULL);
  sem_init (&libres, 0, 7);   /* 7 libres */
  sem_init (&pieces, 0, 0);   /* 0 pieces */

  for (i = 0; i < 3; i ++)    /* 3 producteurs */
    pthread_create (t + i, NULL, producteur,   (void *) i);
  for ( ; i < 5; i ++)        /* 2+1 consommateurs */
    pthread_create (t + i, NULL, consommateur, (void *) i);
  consommateur ((void *) i);
}
```

Sémaphore (8)

- La structure de sémaphore peut être partagée entre plusieurs processus
  - Doit alors être placée dans un segment de mémoire partagée

Communication inter-processus (1)

- Les tubes de communication ne permettent la communication qu'entre deux processus
  - Nécessité de mécanismes plus souples
- Les mécanismes de synchronisation (verrous, sémaphores) sont des primitives rudimentaires pour réguler l'accès des processus à des ressources partagées
  - Nécessité de mécanismes plus évolués

Communication inter-processus (2)

- Unix System V offre plusieurs mécanismes permettant la communication inter-processus (IPC, pour « *Inter-Process Communication* ») entre processus multiples au sein d'une machine donnée :
  - Mémoire partagée (« *shared memory* »)
  - Files de messages (« *message queues* »)
  - Sémaphores
    - Conçus pour fonctionner en mémoire partagée

Communication inter-processus (3)

- Philosophie homogène relative aux mécanismes d'IPC :
  - Acquisition : méthode « *get() »
    - Analogue à la création d'un descripteur de fichier
  - Utilisation : dépend du type de mécanisme IPC
  - Gestion : méthode « *ctl() »
- L'acquisition nécessite le recours à un espace de nommage dédié
  - Identification univoque par les processus / *threads*
  - « Clé » numérique

Mémoire partagée

- Acquisition : « shmget() »
  - Numéro de clé ou IPC_PRIVATE
- Ancrage / détachement de l'espace d'adressage du processus : « shmat() » et « shmdt() »
- Gestion : « shmctl() »
  - Destruction du segment après détachement de tous les processus l'ayant attaché (IPC_RMID)
  - Verrouillage en mémoire physique

Files de messages

- Acquisition : « msgget() »
  - Numéro de clé ou IPC_PRIVATE
- Envoi / réception de messages : « msgsnd() » et « msgrcv() »
  - Envoi de messages « typés »
  - Réception bloquante ou non, selon le type de message ou non
- Gestion : « msgctl() »
  - Destruction immédiate de la file de messages (IPC_RMID)

OpenMP (1)

- Les mécanismes système de gestion des tâches légères sont très rudimentaires
  - Longs à mettre en œuvre
    - Nombreuses lignes à coder
  - Nombreux risques d'erreurs
    - Y compris pour des opérations simples
- Nécessité d'un mécanisme :
  - Plus simple à mettre en œuvre
    - Utilisable par les programmeurs d'applications
  - Éventuellement plus efficace

OpenMP (2)

- OpenMP est une interface de programmation (API) normalisée pour la programmation parallèle en mémoire partagée
  - Portable
  - Interfaces pour : C, C++, Fortran
  - Portée par l'*Architecture Review Board*
    - AMD, Intel, Cray, IBM, NEC, Microsoft, Oracle, ...

OpenMP (3)

- Interface basée sur :
  - Des annotations de code :

    « #pragma omp *directive* »
  - Des fonctions : « omp_*fonction*() »
- Permet une parallélisation plus ou moins intrusive en termes de modification de code
  - Plus on en dit, plus on a de performances
- Ne permet pas de créer ses propres outils de synchronisation
  - Pas de lien explicite avec les Pthreads

OpenMP : parallélisation (1)

- La directive « parallel » met en œuvre l'exécution parallèle d'une section de code
  - Ne s'applique qu'à l'instruction suivante

omp_par1.c

```
#include <omp.h>
int
main ()
{
#pragma omp parallel
  printf ("Bonjour\n");
  printf ("Au revoir\n");
  return (0);
}
```

- À compiler avec « -fopenmp »

OpenMP : parallélisation (2)

- Le nombre de *threads* utilisés correspond au nombre d'unités de traitement disponibles à l'exécution
  - Peut être borné avec la variable d'environnement « OMP_THREAD_LIMIT »
  - Peut être spécifié avec la variable d'environnement « OMP_NUM_THREADS »
    - « OMP_NUM_THREADS=3 *programme* »

OpenMP : parallélisation (3)

- Le nombre de *threads* utilisés peut être spécifié à chaque occurrence de parallélisation au moyen de la directive « num_threads (x) »
  - Expression évaluée dynamiquement à l'exécution

omp_par2.c

```
#include <omp.h>
int
main (
int    argc,
char * argv[])
{
#pragma omp parallel num_threads (atoi (argv[1]))
  printf ("Bonjour\n");
  printf ("Au revoir\n");
  return (0);
}
```

OpenMP : parallélisation (4)

- Le compilateur met en œuvre une barrière de synchronisation à la fin de la section parallèle

omp_num.c (partie 1)

```
#include <omp.h>
main ()
{
#pragma omp parallel
  f (omp_get_thread_num ());
  return (0);
}
```

- Équivaut à :

```
{
  pthread_t      tid[P];
  for (int i = 1; i < P; i ++)
    pthread_create (tid+i, NULL, f, i);
  f (0);
  for (int i = 1; i < P; i ++)
    pthread_join (tid[i], NULL);
}
```

OpenMP : parallélisation (5)

- La fonction « omp_get_thread_num() » permet de connaître le rang du *thread* courant parmi les « omp_get_num_threads() »

omp_num.c (partie 2)

```
#define DS(n,p,i) (((n) + ((p) - 1 - (i))) / (p))
#define DF(n,p,i) ((i) * ((n) / (p)) + (((i) > ((n) % (p))) \
                  ? ((n) % (p)) : (i)))
#define N 1000
int tableau[N];                  /* Donnees à traiter */
void f (int r)
{
  int p = omp_get_num_threads ();
  int d = DF (N, p, r);
  int f = d + DS (N, p, r);
  printf ("Rang %d travaille de %d a %d\n", r, d, f - 1);
  for (int i = d; i < f; i ++)
    tableau[i] = i;
}
```

OpenMP : parallélisation (6)

- On peut déclarer des sections séquentielles dans une section parallèle
  - Directive « single » : l'une des tâches le fait
  - Directive « master » : la tâche 0 le fait

omp_par3.c

```
int
main ()
{
#pragma omp parallel num_threads (4)
  {
    printf ("Bonjour : %d\n", omp_get_thread_num ());
#pragma omp single
    printf ("Milieu : %d\n", omp_get_thread_num ());
    printf ("Au revoir : %d\n", omp_get_thread_num ());
  }
  return (0);
}
```

OpenMP : parallélisation (7)

- On peut déclarer une barrière de synchronisation au moyen de la directive « barrier »
  - Tous les *threads* doivent avoir atteint la barrière pour qu'ils puissent la traverser

omp_par4.c

```
int
main ()
{
#pragma omp parallel num_threads (4)
  {
    printf ("Bonjour : %d\n", omp_get_thread_num ());
#pragma omp barrier
    printf ("Au revoir : %d\n", omp_get_thread_num ());
  }
  return (0);
}
```

OpenMP : variables (1)

- Des clauses permettent de spécifier le mode de partage des variables
  - « #pragma ... private (...) » : variables privées
    - Propres à chaque *thread*
  - « #pragma ... shared (...) » : variables partagées

omp_var1.c

```
{
  int i = -1;
#pragma omp parallel num_threads (4) private (i)
  {
    i = omp_get_thread_num ();
    printf ("Pour moi, i=%d\n", i);
  }
  printf ("Au final : i=%d\n", i);
  return (0);
}
```

OpenMP : variables (2)

- « #pragma ... firstprivate (...) » : variables privées
  - La valeur de la variable est recopiée à l'initialisation du *thread*
- « #pragma ... lastprivate (...) » : variables privées
  - Valeur affectée par le *thread* exécutant la dernière affectation du programme séquentiel

OpenMP : variables (3)

- Il est possible de spécifier la portée par défaut des variables au sein des *threads*
  - « #pragma ... default (none | shared | private | firstprivate) »

université de BORDEAUX

OpenMP : variables (4)

- À la fin d'une section parallèle, il est possible d'effectuer une réduction sur certaines variables au moyen de la directive « reduction »
  - Appliquée à la copie globale et aux copies locales

omp_var2.c

```
#include <stdio.h>
#include <omp.h>

int
main ()
{
  int f = -1;
#pragma omp parallel default (shared) reduction (*:f) num_threads (4)
  f = 1 + omp_get_thread_num ();

  printf ("f = %d\n", f);
  return (0);
}
```

OpenMP : variables (5)

- On peut définir des variables persistantes dont la valeur est préservée entre sections parallèles au moyen de la directive « threadprivate (...) »
  - Variables déclarées comme « static »

omp_var3.c

```
{
  static int i = -1;
#pragma omp threadprivate (i)
#pragma omp parallel num_threads (4)
  {
    i = omp_get_thread_num ();
    printf ("Pour moi, i=%d\n", i);
  }
  printf ("On passe ici\n");
#pragma omp parallel num_threads (4)
  printf ("Pour moi, i=%d\n", i);
  return (0);
```

OpenMP : boucles (1)

- La directive « for » permet de distribuer automatiquement les indices de boucle au sein d'une section parallèle
  - Les directives peuvent être combinées

omp_for1.c

```
#include <omp.h>
main ()
{
  int tableau[1000];
#pragma omp parallel for
  for (int i = 0; i < 1000; i ++)
    tableau[i] = omp_get_thread_num ();
  for (i = 0; i < 1000; i ++)
    printf ("%d", tableau[i]);
  printf ("\n");
  return (0);
}
```

université de BORDEAUX

OpenMP : boucles (2)

- L'ordonnancement des itérations peut être :
  - Statique : directive « schedule (static) », par défaut
  - Dynamique : directive « schedule (dynamic) »
- Possibilité de spécifier une taille de bloc

omp_for2.c

```
#include <omp.h>
main ()
{
  int i, tableau[1000];
#pragma omp parallel for schedule (dynamic)  /* ou (dynamic,10) */
  for (i = 0; i < 1000; i ++)
    tableau[i] = omp_get_thread_num ();
  for (i = 0; i < 1000; i ++)
    printf ("%d", tableau[i]);
  printf ("\n");
  return (0);
}
```

OpenMP : boucles (3)

- Lors de boucles imbriquées, il faut également s'occuper des indices des boucles internes
  - Au pire, en « private »

omp_for3.c

```
{
  int i, j;
  int t[18][18];
#pragma omp parallel for num_threads (5) private (j)
  for (i = 0; i < 18; i ++)
    for (j = 0; j < 18; j ++)
      t[i][j] = omp_get_thread_num ();
  for (i = 0; i < 18; i ++) {
    for (j = 0; j < 18; j ++) {
      printf ("%d ", t[i][j]);
    }
    printf ("\n");
  }
  return (0);
}
```

OpenMP : boucles (4)

- Les itérations de boucles imbriquées peuvent être réparties équitablement au moyen de la directive « collapse (...) »
  - Permet de spécifier le niveau de fusion des indices

omp_for4.c

```
{
  int i, j;
  int t[18][18];
#pragma omp parallel for num_threads (5) collapse (2)
  for (i = 0; i < 18; i ++) {
    for (j = 0; j < 18; j ++)
      t[i][j] = omp_get_thread_num ();
  }

  ...
}
```

OpenMP : exclusion mutuelle (1)

- L'exclusion mutuelle peut être mise en place au moyen de la directive « critical »

omp_critical.c

```
{
  int i = 0;
#pragma omp parallel num_threads (4) shared (i)
  {
    int j;
#pragma omp critical
    {             /* Sérialisation des accès à la section critique */
      j = i;
      j ++;
      sleep (1);
      i = j;
    }
    printf ("Pour moi, i=%d\n", i);
  }
  printf ("A la fin, i=%d\n", i);
  return (0);
}
```

OpenMP : exclusion mutuelle (2)

- Par défaut, tous les *threads* OpenMP utilisent le même *mutex* pour mettre en œuvre leurs sections critiques
- Il est possible de définir des verrous différents en les nommant dans la directive « critical »
  - « #pragma omp critical *nom* »

OpenMP : exclusion mutuelle (3)

- Ce n'est pas parce qu'une instruction a l'air atomique qu'elle est atomique

omp_atomic1.c

```
int
main ()
{
  int i, j, s = 0;
#pragma omp parallel for collapse (2) shared (s)
  for (i = 0; i < 1000; i ++) {
    for (j = 0; j < 1000; j ++)
      s ++;
  }
  printf ("A la fin, s=%d\n", s);
  return (0);
}
```

université de BORDEAUX

OpenMP : exclusion mutuelle (4)

- L'exclusion mutuelle sur une opération possiblement atomique peut être mise en place au moyen de la directive « atomic »
  - Remplacée par un « critical » s'il n'existe pas d'opération atomique correspondante

omp_atomic2.c

```
{
  int i, j, s = 0;
#pragma omp parallel for collapse (2) shared (s)
  for (i = 0; i < 1000; i ++) {
    for (j = 0; j < 1000; j ++) {
#pragma omp atomic
      s ++;
    }
  }
  printf ("A la fin, s=%d\n", s);
  return (0);
}
```

OpenMP : exclusion mutuelle (5)

- Mettre en place une opération atomique est bien moins coûteux qu'une section critique, mais plus coûteux qu'un accès purement local

omp_atomic3.c

```
  int i, j, s = 0;
#pragma omp parallel shared (s)
  {
    int r = 0;
#pragma omp for collapse (2) schedule (dynamic)
    for (i = 0; i < 1000; i ++) {
     for (j = 0; j < 1000; j ++)
        r ++;
    }
    printf ("Chez %d, r=%d\n", omp_get_thread_num (), r);
#pragma omp atomic
    s += r;
  }
  printf ("A la fin, s=%d\n", s);
```

OpenMP : exclusion mutuelle (6)

- Les directives « critical » et « atomic » associent des verrous à des sections de code
- OpenMP permet également de définir et de positionner des verrous sur des données
  - Type « omp_lock_t »
  - Primitives « omp_init_lock() », « omp_destroy_lock() », « omp_set_lock() », « omp_unset_lock() », « omp_test_lock() »
  - Fonctionnement analogue à celui de pthread_mutex_t

Pile (1)

- Presque tous les langages de programmation incluent le concept de procédure disposant de paramètres d'appel et de variables locales
  - Ces variables peuvent être accédées pendant l'exécution de la procédure mais pas depuis la procédure appelante
  - Elles ne peuvent résider à une adresse absolue en mémoire, car cela empêcherait la réentrance
- Nécessité de créer dynamiquement des instances de ces variables lors des appels de procédures et de les supprimer à la fin

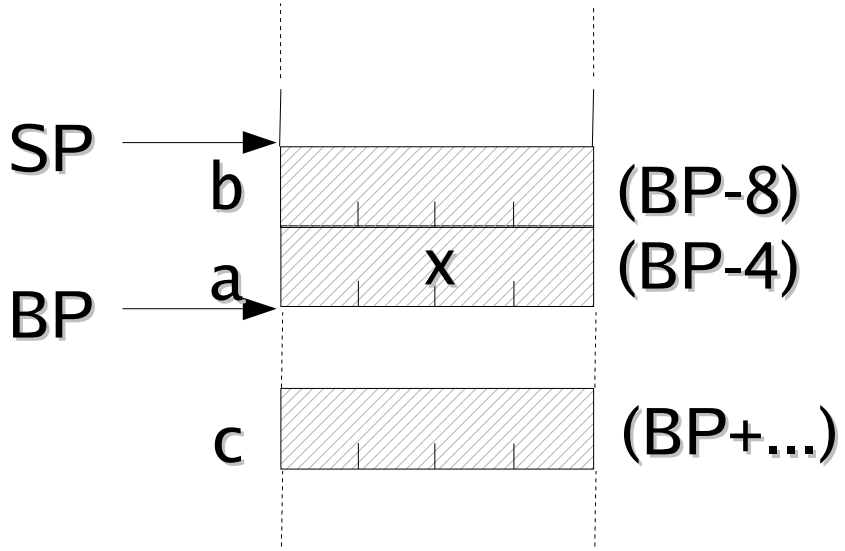
Pile (2)

- Une pile est une zone de la mémoire que l'on n'accède jamais de façon absolue mais toujours relativement à un registre
- Gérée en fait au moyen de deux registres
  - Un registre de base (« *Base Pointer* », ou BP) pointe sur le début de la zone mémoire allouée pour les variables locales de la procédure courante
  - Un registre de sommet de pile (« *Stack Pointer* », ou SP) pointe sur le dernier mot mémoire alloué

Pile (3)

- Les paramètres et les variables locales à la procédure courante sont référencées par rapport à la valeur courante de BP
- La zone de données référencée par BP et limitée par SP est appelée « contexte courant »

```
void
f (
int     c)
{
  int   a;
  int   b;
...
  f (a + 1);
...
}
```

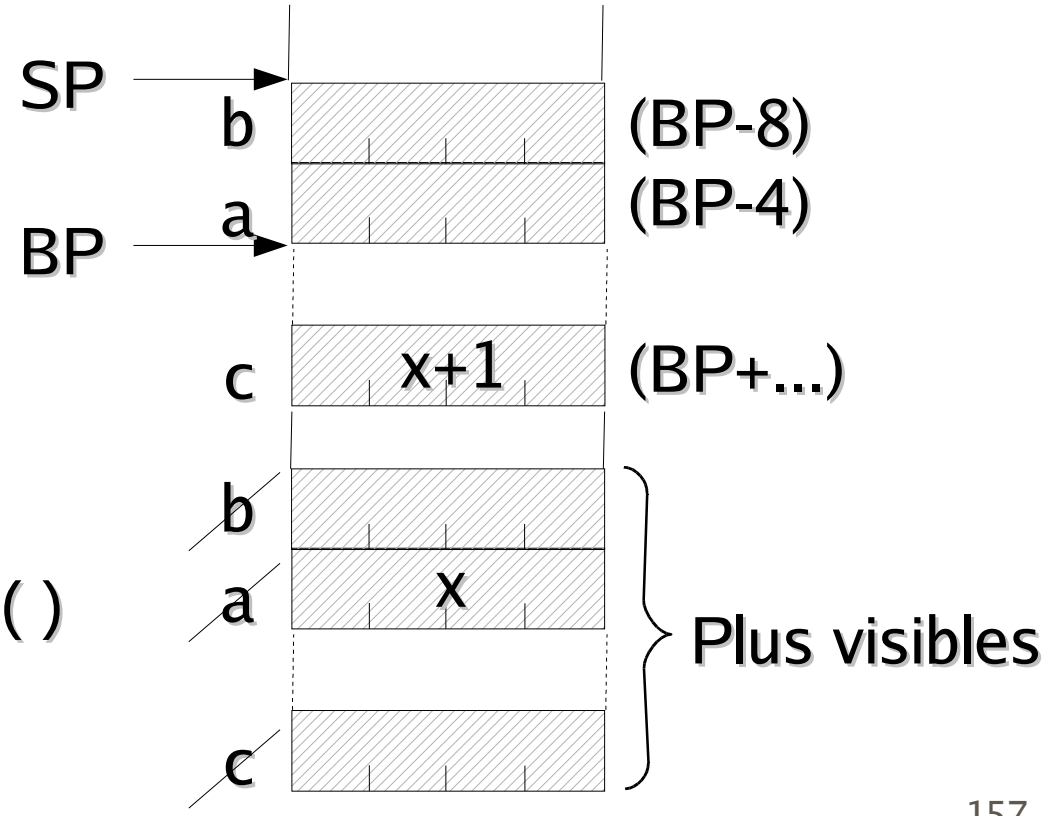


Pile (4)

- Lors d'un appel de procédure, un nouveau contexte courant se crée au sommet de la pile
  - Comment gérer le retour au contexte appelant ?

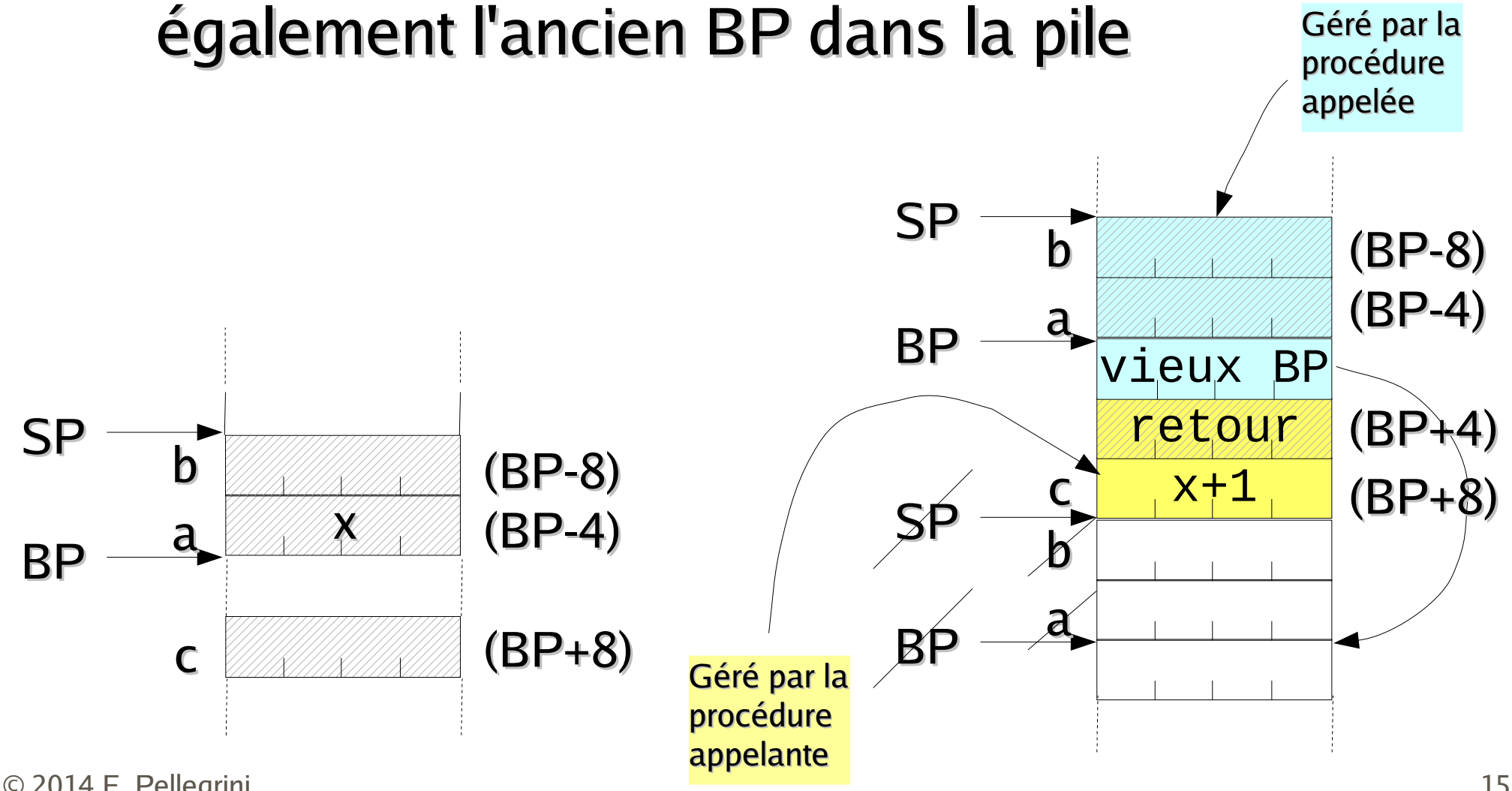
```
void
f (
int     c)
{
  int   a;
  int   b;
...
  f (a + 1);
...
}
```

Appel de `f()`



Pile (5)

- Lors d'un appel de procédure, on sauve également l'ancien BP dans la pile



Pile (6)

- Séquence d'appel d'une procédure, partie gérée par la procédure appelante :
  - Empilage des paramètres, dans l'ordre inverse de celui dans lequel ils sont listés dans la procédure
    - Autorise les fonctions à nombre d'arguments variables : le premier paramètre est le plus proche de BP, les autres sont dessous !
  - Appel de la procédure
    - Sauvegarde automatiquement l'adresse de retour dans la pile

Pile (7)

- Séquence d'appel d'une procédure, partie gérée par la procédure appelée (début de procédure) :
  - Empilage de l'ancien BP dans la pile
  - Copie de la valeur de SP dans celle de BP
    - Le nouveau contexte est basé à la position courante de SP
    - Le premier paramètre est accessible à l'adresse de BP plus la taille de deux adresses entières (l'ancien BP et l'adresse de retour), donc (BP+8)
  - Soustraction à SP de la taille des variables locales
    - Réserve l'espace en cas d'appels ultérieurs

Pile (8)

- Séquence de retour d'une procédure, partie gérée par la procédure appelée (fin de procédure) :
  - Remise dans SP de la valeur de BP
    - Libère la zone des variables locales à la procédure
  - Dépile BP
    - BP pointe de nouveau sur le contexte appelant
  - Appelle l'instruction de retour
    - Dépile la valeur de retour située dans la pile

Pile (9)

- Séquence de retour d'une procédure, partie gérée par la procédure appelante :
  - Incrémentation de SP de la taille de tous les paramètres empilés avant l'appel à la procédure
  - Retour complet à l'état antérieur

Sauts distants (1)

- Les « goto » ne permettent que d'effectuer des sauts locaux à une fonction
  - Le compilateur, ayant la visibilité de l'ensemble de la fonction, peut adapter l'état de la pile afin que les sauts s'effectuent « à niveau de pile constant » entre source et destination du branchement
- On peut avoir besoin d'effectuer des sauts non locaux
  - Remontée rapide d'un arbre récursif
  - Traitement des exceptions

Sauts distants (2)

- Pour effectuer un saut « non local », il faut savoir (et avoir !) l'état de la pile de là où on compte atterrir
  - Impossible dans le cas de fonctions que l'on n'a jamais exécutées
    - À quel niveau de récursion faudrait-il revenir dans une fonction récursive ?
- On ne peut donc effectuer de saut distant que vers un endroit par lequel on est « déjà passé »
  - On peut « mémoriser » l'état de la pile à ce moment

Sauts distants (3)

- On peut en théorie effectuer un saut distant vers un contexte de pile plus profond
  - Mais cela obligerait à sauvegarder l'intégralité de la pile, car son contenu peut changer entretemps
  - Très (trop!) coûteux !
- Est-ce vraiment utile ?
  - Cela transformerait le logiciel en un « plat de spaghetti » impossible à déboguer !
  - On n'a vraiment besoin que de sauts « remontants »
    - Dans ce cas, pas besoin de sauver la pile !

Sauts distants (4)

- Les sauts distants se mettent en œuvre au moyen de deux appels système couplés :
  - On mémorise l'état d'exécution courant d'un *thread* au moyen de l'appel système « setjmp() »
  - On restaure l'état d'exécution courant d'un *thread* au moyen de l'appel système « longjmp() »
    - Ramène au dernier état mémorisé avec setjmp()

Sauts distants (5)

- Pour savoir si l'on revient d'un setjmp()
  - Mécanisme de désambiguïsation analogue à fork()

Sauts distants (6)

- L'utilisation des sauts distants est très délicate et requiert une conception logicielle rigoureuse !
- Comme on ne « remonte » pas normalement des contextes de pile, aucune action de « nettoyage » ne peut être entreprise :
  - Quid des zones mémoire allouées et pointées par des variables locales ?
  - Quid des descripteurs de fichiers ouverts ?

Sauts distants (7)

- Tous les effets de bords créés au sein de l'arborescence d'appel doivent être mémorisés au sein de variables persistantes à cette arborescence
  - Accessibles au niveau du contexte le plus haut
- La sémantique de « setjmp / longjmp » ne dit rien sur la restauration des masques de signaux
  - Utiliser « sigsetjmp / siglongjmp » pour les gérer

Bibliothèques (1)

- Lorsqu'on fournit un ensemble de fonctions rendant un ensemble cohérent de services, il serait préférable de grouper les fichiers objets associés sous la forme d'un unique fichier
  - Facilite la manipulation et la mise à jour
- Ce service est rendu par les fichiers de bibliothèque
  - Fichiers servant à archiver des fichiers objet
  - Utilisables par l'éditeur de liens

Bibliothèques (2)

- Deux types de bibliothèques
  - Bibliothèques statiques
    - Format en « `lib*.a` » (Unix) ou « `*.lib` » (DOS)
    - Liées à l'exécutable lors de la compilation
      - Augmentent (parfois grandement) la taille des exécutables
      - On n'a plus besoin que de l'exécutable proprement dit
  - Bibliothèques dynamiques
    - Format en « `lib*.so` » (Unix, « *shared object* ») ou « `*.dll` » (Windows, « *dynamic loadable library* »)
    - Liées à l'exécutable lors de l'exécution
      - Permettent la mise à jour indépendante des bibliothèques
      - Problème si pas présentes (variable « `LD_LIBRARY_PATH` »)

Ar (1)

- La commande Ar (« archive ») est un outil de gestion de fichiers d'archives
  - Un fichier archive est un fichier contenant d'autres fichiers
  - Peut archiver tout type de fichier
- Bien qu'il existe d'autres outils similaires et plus efficaces tels que Tar, Ar perdure car son format est reconnu par les éditeurs de liens
  - Définition de fichiers de bibliothèques servant à stocker des fichiers objets

Ar (2)

- La commande Ar permet :
  - D'ajouter des fichiers à une archive
  - De remplacer des fichiers contenus dans une archive
    - Commande « `ru` » pour remplacer par les plus récents
  - De supprimer des fichiers contenus dans une archive
  - De modifier l'ordre des fichiers de l'archive
    - L'ordre des fichiers dans l'archive est important !

```
% gcc -c brol.c -o brol.o
% gcc -c brol_io.c -o brol_io.o
% gcc -c brol_check.c -o brol_check.o
% gcc -c brol_compute.c -o brol_compute.o
% ar ruv libbrol.a brol.o brol_io.o brol_check.o brol_compute.o
```

Ar (3)

- Par défaut, lors de l'édition de liens, les fichiers de bibliothèques sont parcourus linéairement
  - Si deux fichiers contiennent le même symbole, c'est le premier fichier rencontré qui sera inclus
  - Si, lors de l'édition de liens, le fichier objet membre de bibliothèque dont on a besoin a lui-même besoin d'un symbole présent dans un fichier objet déjà vu mais pas inclus, l'édition de liens échouera
- Même problème avec l'ordre dans lequel on demande à l'éditeur de liens de consulter les fichiers bibliothèques

Ranlib

- La commande Ranlib ajoute dans le fichier d'archive un fichier d'index listant les symboles définis dans les fichiers objets de l'archive
  - Permet d'éviter le problème de visibilité induit par le parcours linéaire des fichiers de l'archive

```
% ranlib libbrol.a
```

makefile

```
RANLIB=ranlib
...
libbrol.a TAB     : brol.o trol.o chmol.o
TAB       TAB       $(AR) r $(@) $(?)
TAB       TAB       -$(RANLIB) $(@)
```

Nm (1)

- La commande Nm sert à lister les symboles présents dans les fichiers objets ou les bibliothèques
  - Permet de vérifier si une donnée ou un symbole existe bien dans le fichier objet ou de bibliothèque
    - Aide au diagnostic si la compilation échoue à l'édition de liens en indiquant un symbole non résolu
    - Permet de déterminer quelle bibliothèque utiliser lors de l'édition de liens
  - Permet de vérifier si l'interface d'une bibliothèque correspond bien à sa documentation
    - Liste des fonctions définies

Nm (2)

- Les symboles contenus dans les fichiers objets sont identifiés par une lettre code :
  - `T` : Fonction définie (symbole du segment « text »)
    - Le code de la fonction est disponible dans le fichier
  - `t` : Fonction statique définie
    - Le symbole n'est visible, et donc la fonction n'est directement appelable, qu'à partir du fichier lui-même
  - `U` : Fonction ou variable non définie
    - L'éditeur de liens devra trouver une définition de cette fonction dans un autre fichier objet ou il y aura échec

Nm (3)

- B : Donnée définie dans la zone des données non initialisées (symbole du segment « BSS »)
  - La place est réservée, mais le contenu de la zone n'est pas stocké dans l'exécutable
- D : Donnée définie dans la zone des données initialisées (symbole du segment « data »)
  - Le contenu initial de la zone sera stocké dans l'exécutable
- R : Donnée définie dans la zone des données initialisées et en lecture seule
- b, d, r : Versions statiques des précédents
  - Le symbole n'est pas visible en dehors du fichier objet

Nm (4)

```
% nm common_integer.o
         U __ctype_b_loc
         U fprintf
0000014a T intAscn
00000000 T intLoad
00000190 T intPerm
0000023f T intRandInit
00000121 T intSave
00000267 T intSort1asc1
00000287 t intSort1asc1_2
000002b0 T intSort2asc1
000002d0 T intSort2asc2
000002f0 t intSort2asc2_2
         U _IO_getc
         U qsort
00000000 b randflag.4042
         U random
         U srandom
         U ungetc
```

Nm (5)

```
% nm libcommon.a
common.o:
00000000 T clockGet
         U fprintf
         U fwrite
         U getrusage
00000065 T usagePrint
common_error.o:
00000050 T errorPrint
000000b6 T errorPrintW
00000000 T errorProg
00000000 d errorProgName
         U fflush
         U fprintf
         U fputc
         U stderr
         U strncpy
         U vfprintf
common_integer.o:
...
```

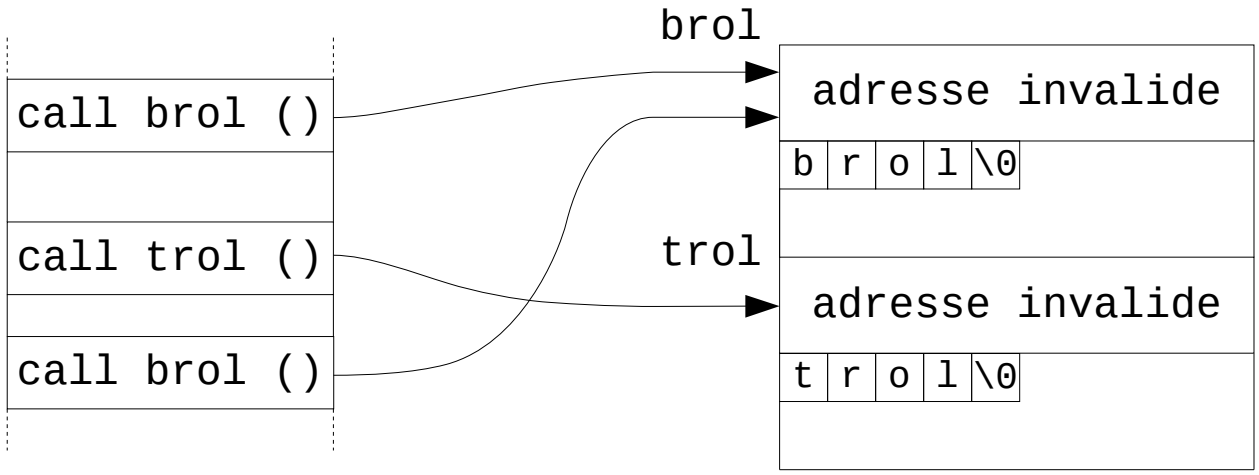
Nm (6)

- Sur certaines implémentations, l'option « -s » permet de voir le contenu du fichier d'index
  - Uniquement les symboles non statiques

```
% nm -s libcommon.a
Archive index:
clockGet in common.o
usagePrint in common.o
intRandInit in common_integer.o
intSort1asc1 in common_integer.o
intSort2asc1 in common_integer.o
intSort2asc2 in common_integer.o
memAllocGroup in common_memory.o
memReallocGroup in common_memory.o
common.o:
00000000 T clockGet
         U fprintf
...
```

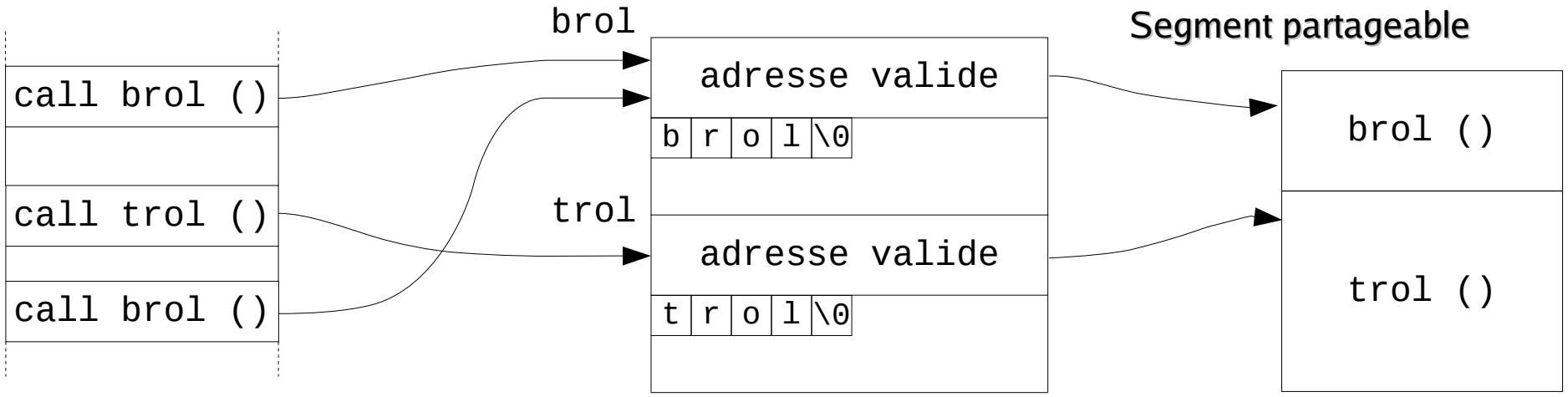
Édition dynamique de liens (1)

- Lorsque l'éditeur de liens trouve la définition d'un symbole dans une bibliothèque dynamique
  - Il ajoute au programme un fichier objet spécial, issu de la bibliothèque, appelé « module d'importation »
  - Le module contient les adresses (invalides) des branchements ainsi que le nom de la fonction



Édition dynamique de liens (2)

- Lors d'un appel à une fonction dynamique, il y a erreur de segmentation et traitement
  - Le contenu de la bibliothèque dynamique est chargé dans un nouveau segment partageable si elle n'était pas déjà présente dans le système
  - Les adresses modifiées pointent dans ce segment



Édition dynamique de liens (3)

- Dans d'autres systèmes, il n'y a pas d'indirections sur les adresses des routines
  - Les adresses des instructions de branchements sont stockées dans la table de relocation et associées au nom de la bibliothèque dynamique à charger
  - Lors du chargement du programme, toutes les bibliothèques dynamiques nécessaires sont chargées en mémoire dans des segments partagés si elles ne l'étaient pas déjà, et les adresses des branchements sont modifiées en conséquence

Édition dynamique de liens (4)

- Certains systèmes permettent aussi le chargement dynamique de bibliothèques à la demande lors de l'exécution du programme
  - Analogue au chargement dynamique des modules dans le noyau
  - Demande explicite par le programmeur
    - Cas des systèmes de type Unix : `dlopen()`, `dlsym()`, `dlclose()`
  - Respecte le paradigme de l'éditeur de liens
    - Pas de symbole non résolu à l'édition de liens
    - Manipulation des symboles dynamiques par pointeurs