340 – TRAITEMENT DES FICH OUVERTS

Descripteur de fich
- Dans un prc
- Désigne un fich ouvert
- Sert à la manipulation
- C'est un entier tout simple (int)

Trois descripteurs standard
- 0 : entrée standard
- 1 : sortie standard
- 2 : sortie standard pour les msg d'err

→ C'est des conventions de l'esp user : le noyau s'en fiche

Utilisation des descripteurs
- Ouverture : creat(2), open(2)…
- Manipulation : read(2), write(2)…

```
#include<unistd.h>
#include<string.h>
#include<fcntl.h>
int main(void) {
  char msg[] = "Hello, World!\n";
  int fd = creat("hello", 0666);
  write(fd, msg, strlen(msg));
  close(fd);
}
```

Questions :
- Pourquoi strlen et pas sizeof ? sizeof retourne la taille d'un pteur dans le cas ou `msg`aurait été un pteur. strlen retourne le nb de caract. sizeof retourne le nb d'octets + le NULL (caché). Parfois, les deux concordent, parfois non.

Organisation interne : 3 niveaux
TD. Tables des descripteurs (une par prc)
- Une entrée par descripteur, propre à chacun des prc du sys
- Pointe sur un fich ouvert (→TFO*)

TFO. Table de fich ouverts (globale)
- Une entrée par demande d'ouverture d'un fich
- Chq open ou create ou autre
- Pointe sur un inode en mém (→TIM*)

TIM. Table des inodes en mém (globale)
- Une entrée par fich (inode) distinct manipulé (ou en cache)
- Synchronisée avec les inodes sur disque

*Qui contient un cpteur.

Info sur les fich ouverts ?
- /proc/sys/fs/inode-nr nb d'inodes en mém
- /proc/sys/fs/file-nr nb d'inodes ouverts distincts

Commandes
- lsof(1) et fuser(1) permet de « voir » ou « chercher » les fich ouverts
- Cherchent/voient aussi les communications réseau et les tubes

/proc/PID
- /proc/PID/fd le fich (ou autre) associé au descripteur
- /proc/PID/fdinfo des info sur le fich ouvert

TIM : Caches des fich
- Table des inodes en mém → cache par fich

Le SE minimise les accès disque : asynchronisme
- En lect (readahead) et en écr (flush)
- Demandes des users ≠
  Lect et écr effectives sur disque
- sync(1), sync(2), fsync(2) : forcer les écr
- /proc/sys/vm/drop_caches : libérer l'esp des caches

Cohérence entre accès concurrents
- Prc peuvent lire et écrire sur le mm fich
- Chacun a la mm vision du contenu (le cache noyau)

Attention : ne pas confondre
- Caches noyau
- Caches applicatifs (prog et biblio)

TFO : Table des fich ouverts
- Une entrée par le open(2) (ou autre) effectué
- Contient le mode d'ouverture (lect, écr…)
- Contient le curseur lect-écr (éventuel) dans le fich

→ Un mm fich peut être manipulé indépendamment par des prc
- lseek(2) permet de déplacer le curseur lect-écr

hello_wr
```
#include<unistd.h>
#include<string.h>
#include<fcntl.h>
int main(void) {
  char msgout[] = "Hello, World!\n", msgin[50];
  int fdout = creat("hello", 0666);
  int fdin = open("hello", O_RDONLY);
  write(fdout, msgout, strlen(msgout));
  ssize_t len = read(fdin, msgin, sizeof(msgout));
  write(1, msgin, len);
  close(fdout); close(fdin); return 0;
}
```
- Un seul fich hello
- Deux ouvertures (distinctes)
- Deux descripteurs

Questions :
- Et si on inverse le read et le write ? Rien. Surtout que si le fich existe déjà, creat va l'écraser.
- Comment s'assurer de la cohérence d'un fich si plsrs prc peuvent écrire en mm temps ? Au programmeur d'en assurer.
- Pourrait-on faire communiquer des prc via un fich commun ouvert ? Oui, mais c'est pénible à synchroniser.

Threads, fork, exec
Pthreads partagent
- Les descripteurs sont associés au prc

→ Donc partagés par les threads

Fork duplique (et partage)
- La table des descripteurs est dupliquée
- Les entrées dans la table des fich ouverts sont partagées
- En particulier le curseur de position (lect/écr)
- Les compteurs de la table des fich ouverts sont incrémentés
- **Question** : Pourquoi ne pas incrémenter les compteurs de la TIM? On n'a pas fait d'open ou de creat.

Exec préserve
- La table des descripteurs est préservée (incluant les entrées et sorties standard)

→ Permet de préserver 0, 1 ou 2

hello_fork.c
```
#include<unistd.h>
#include<fcntl.h>
#include<wait.h>
int main(void) {
  char buf[50];    size_t len = 0;
  int fd = open("bonjour.txt", O_RDONLY);
  pid_t p = fork();
  if (p==0) {
    len += read(fd, buf, 5);
    sleep(2);
    len += read(fd, buf+len, 5);
  } else {
    sleep(1);
    len += read(fd, buf, 5);
    wait(NULL);
  }
  write(1, buf, len); return 0;
}
```
```
$ cat bonjour.txt
Bonjonde!
ur mouldiou!
```

Spécificités Linux
Flag O_CLOEXEC
- O_CLOEXEC flag de open(2) (et autres syscalls)
- Le descripteur sera automatiquement fermé lors d'un execve(2)

→ Évite la fuite de descripteurs ou gaspillage de ress
→ Mais pas portable

Tâches Linux
clone(2) permet de décider quoi partager ou cloner
- CLONE_FILES la table des descripteurs
- CLONE_FS des info liées au sys de fich, dont chdir et umask

Duplication de descripteurs
Descripteurs synonymes
- Deux descripteurs d'un mm prc peuvent pointer une mm entrée dans la table des fich ouverts
- Syscalls dup2(2) (et dup(2))

Quel est l'intérêt ?
- Redéfinir les entrées et sorties standard
- Redirection de fich
- Communication par tube (pour plus tard)

Question : Quelle est la diff entre dupliquer un descripteur et ouvrir deux fois un fich ? Dupliquer : On peut créer un autre descripteur et le faire pointer nous-mm vers la mm entrée (rediriger). Intérêt : Au lieu que le stdout pointe vers un fich spécial du terminal, on le fait pointer vers un fich où on va écrire.

Redirection de la sortie standard
```
#include<unistd.h>
#include<fcntl.h>
#include<stdio.h>

int main(void) {
  int fd = creat("sortie", 0666);
  dup2(fd, 1);
  printf("Hello World!\n");
  return 0;
}
```

Redirection de l'entrée standard
```
#include<unistd.h>
#include<fcntl.h>
#include<stdio.h>

int main(void) {
  int fd = open("hello", O_RDONLY);
  dup2(fd, 0);
  close(fd);
  execlp("lolcat", "lolcal", NULL);
  perror("lolcal");
  return 1;
}
```

Autre partage de descripteurs ou fich
- Un descripteur est un entier, partager 4 n'avance à rien

→ Le noyau doit être impliqué

Via sockets Unix
- unix(7)
- Partage des fich ouverts
- Via messages auxiliaires (SCM_RIGHTS)

Via /proc (Linux)
- /proc/PID/fd : proc(5)
- Partage des inodes en mém
- Mm des fich supprimés
- Mm de communication interprc (tubes, sockets...)

350 – IMPLÉMENTATION DES SYS DE FICH

Types de sys de fich
- Nbux types existent
- Wikipédia en liste et compare une centaine

Nbux, car spécifiques
- À des sys et/ou organisations
  Contrôle de l'évolution, mais aussi syndrome NIH
- À des contraintes physiques des périphériques et des ordi
- À des besoins spécifiques des users

Format de stockage
Spécifie la représentation des données sur disque
- Champs de bits
- Structures de données

Implémentation
- Dans les SE
- Dans les outils annexes (mkfs(8), fsck(8)...)

→ C'est compliqué, surtout si le format n'est pas documenté
→ Besoin maximal de fiabilité : ne pas manger les données !

Découpage en blocs
- Blocs de taille fixe, configurable, ou variable (ça dépend)
- Découpe tout l'esp disque
- Simplifie la gestion : blocs au lieu d'octets
- Tout est des blocs ensuite : données ou gestion

Limites principales
- Nb max d'inodes
- Taille max d'un fich
- Nb d'entrées max par répertoire
- Taille max du volume…

Exemples de limites
Type	Taille fich	Taille volume
FAT32	4 Go	16 Tb
NTFS	16 Eo*	16 Eo
ext4	16 To	1 Eo
btrfs	16 Eo	16 Eo
ZFS	16 Eo	256 Zo†

Ce sont des limites du format : la plupart des implémentations (sys et outils) ne les atteignent pas forcément.
*16 Eo = 2^{64} octets = 18 446 744 073 709 552 000 octets (20 chiffres)
†256 Zo = 2^{78} octets ≈ $3 * 10^{23}$ (24 chiffres)

Détermination des limites
- Contraintes internes au type de sys
  Tailles en octets de valeurs numériques
- Param configurés par l'user
  Lors du formatage: mke2fs(8), mkfs(8)…
- Limites d'implémentations
  Le format peut stoker plus, mais les logiciels ne peuvent pas lire
- Combinaisons directes et indirectes de tout ça

Besoins et fctionnalités
De base
- Stocker les données des fich (gros et petit)
- Stocker les métadonnées (y compris étendues xattr(7))
- Stocker les entrées des répertoires
- Gérer l'esp libre (inodes et blocs)

Plus avancés
- Chiffrement et compression
- Journalisation
- Instantanés (snapshots) et branches
- Déduplication
- Multi-volumes, RAID…
- Somme de contrôle (checksum)
- Correction d'err

Allocation et adressage des fich
Allocation contiguë
- Les blocs de données d'un fich sont contigus
- Ex : ISO 9660 (CDs)
- Naïf : en général, la taille des fich est inconnue et évolue

Allocation chainée
- Un bloc de données connait l'adr du suivant
- Ex : FAT
- Prob : accès direct lent (lseek(2))

Allocation indexée
- Un fich connait la liste de ses blocs de données
- Prob : comment stocker des gros fich ? Difficile avec de gros fich

Allocation indexée Unix
Pteurs vers les blocs de données
- Pteur direct : contient l'adr d'un bloc de données
- Pteur indirect : contient l'adr d'un bloc contenant des pteurs directs
- Pteur indirect double : contient l'adr d'un bloc contenant des pteurs indirects…

Question : Comment déterminer où s'arrêtent les données ? On se fie à sa taille.

Adressage des fich – Exercice
Ex : adressage indexé de ext2/3
Dans la table des inodes
- Il y a 15 pteurs de blocs :
  - 12 sont des pteurs directs
  - 1 est indirect
  - 1 est indirect double
  - 1 est indirect triple
- Un bloc fait 4ko (défaut typique, mais configurable)
- Un pteur de bloc est représenté sur 32 bits (4o)

Questions : Quelles sont les tailles max
- D'un fich si tous ses blocs sont pleins ? 16Tb.
- D'un fich si la taille est codée sur 32 bits ?
- D'un volume si tous les blocs sont utilisés ?

Allocations modernes
Extents
- extent = suite de blocs contigus
- On stocke 2 nbs plutôt que tous les blocs de la suite
- ext4, btrfs, ntfs…
- Prob : cf. allocation contiguë

Arbres B (B-tree)
- Struct de données arborescente équilibrée
- Adaptée aux sys de fich et base de données
- btrfs, zfs, ntfs, etc.
- Prob: nbreux détails algorithmiques

Journalisation
Prob : corruption
- Panne lors d'une écr

→ Données partiellement/mal écrites
→ Incohérences données et métadonnées

Sol : écrire en deux temps
- On écrit les données dans un journal
- Quand le journal est écrit, on recopie dans le disque

→ Prob de coût : écr plus chère (copie intermédiaire)

Principe de la journalisation
Si panne pendant l'écr dans journal
- On jette les données du journal (tant pis!)
- Les données du disque sont vieilles, mais cohérentes

Si panne pendant l'écr du disque
- Le journal est complet et cohérent
- On termine l'écr depuis le journal

Détails de la journalisation
- Nbux détails spécifiques
- Configurations poss

Question : Journaliser seulement les métadonnées est-il un bon compromis ? Oui, car c'est la partie la plus importante.

Copie sur écr (copy-on-write)
Principe
- Au lieu de modifier quelque chose
- On en fait une copie modifiée
- On utilise la copie au lieu de l'original
- On libère l'original

→ Plus efficace que la journalisation classique

Généralisation de l'approche
- Utilisé par ZFS et btrfs
- Instantanés : on peut garder d'anciennes versions
- Branches : des versions peuvent évoluer indépendamment

400 – COMMUNICATION INTER-PRC

Chq prc
- Est autonome
- Vit isolé dans son propre esp mém

Trop restrictif
- Besoin de collab, comm et de coop
- Prc diff opèrent ens vers un mm obj

Communication interprc
- IPC (interprocess communication)
- Mécanismes du SE
- Parfois offerts par biblio et démons (esp user)

Formes de coopération
Coopération interne
- App conçue à la base multithreads et multiprc
- Ex : navigateurs modernes

→ Objectifs : performance, asynchronisme, isolation…

Coopération protocolaire
- Communiquer avec des app qui respectent un protocole
- Ex : la plupart des app réseaux

→ Attention à être robuste

Coopération des données
- App manipule des données, produite (ou non) par d'autres app
- Ex : ouvrir/enregistrer, tubes shell, etc.

Philosophie Unix
1978
- Make each prog do one thing well. To do a new job, build afresh rather than complicate old prog by adding new « features ».
- Expect the output of every prog to become the input to another, as yet unknown, prog. Don't clutter output with extraneous info. Avoid stringently columnar or binary input formats. Don't insist on interactive input.
- Design and build software, even operating sys, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.
- Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.

1994
- Write prog that do one thing and do it well
- Write prog to work together
- Write prog to handle text streams, because that is a universal interface.

Mise en garde
Vaste domaine
- La communication entre app informatiques est un domaine bien large

Deux modèles de communication
1. Données échangées entre prc (message passing)
- Chacun les traite à sa façon
- Un seul prc a les données à la fois

2. Données partagées entre prc (shared memory)
- Données communes
- Accès et modif non exclusives

Questions : à quel modèle correspond
- Tube shell « | » : 1
- 2 threads et une struc de données dans le tas : 2
- Fich dans ~/Documents : 2
- Base de données : 2

Exemples d'IPC chez POSIX
- Fich
- Terminaison de prc : exit(2) et wait(2)
- Signaux : signal(7); y compris kill(1) (410)
- Tubes (pire) : pipe(7); y compris « | » et tubes nommés fifo(7) (420)
- Sockets UNIX : unix(7). (430)
- Sockets classiques (TCP, UDP…): socket(7), tcp(7), udp(7)
- Mém partagée: shm_overview(7)

→ Plus quelques autres
→ Plus tous ceux spécifiques à d'autres sys
→ Plus tous ceux « applicatifs » basés sur ces primitives sys RPC (remote proedure call), bus logiciels…

Système de fichiers
Les fich ne sont pas exclusifs à un prc
- Les prc peuvent utiliser les fich pour communiquer entre eux
- L'accès et la protection sont connus

Exemples
- Fich de données et autres doc (explicites à l'user)
- Fich temp pour passer des données (compilation…)
- Fich spéciaux pour initier un autre type de comm (tubes nommés, etc.)
- Spool (impression, cron job, etc.)
- Fich projeté en mém (mmap(2))

Socket réseau
Communication distante
- Obj primaire : faire communiquer des app sur des ordi distincts
- Peut aussi être utilisé pour des app sur une mm mach
- Majoritairement pour du clients-serveur

Avantages
- Le SE peut optimiser l'efficacité du traitement
- Ex : client/serveur X (X(7))

Bus logiciels
Support de comm de haut niveau
- Majoritairement applicatif (en esp user)
- Permet de faire communiquer des app via des objets partagés et des envois de msg
- Souvent pour le réseau (ex. CORBA)

Exemple : D-BUS
- Utilisé dans les bureaux graphiques Unix modernes
- Un démon + clients exposent&utilisent services&objets
- Biblio utilisables par les prog
- Démon et biblio utilisent des primitives sys pour faire le travail de comm, de sync et de protection

Gestion de la communication
Il faut un protocole de comm
- Moyens de comm (données, struc de données) que les prc peuvent échanger et accéder
- Primitives d'accès et de protection
- Mécanismes de sync
- Attention aux prob standards interblocage, famine…

Qui s'en charge ? Le SE
- Fournit des syscall pour IPC
- ± riches et complexes
- Règles spécifiques au cas par cas

→ Impose un protocole mais garantit certaines propriétés

Qui s'en charge ? Le programmeur
- Utilise les primitives sys pour implémenter ses propres protocoles et app
- Prend en cpt les limites et caractéristiques des IPC utilisées
- Est libre d'implémenter tout ce qu'il veut par-dessus

→ les IPC sys sont des outils pour bâtir sa sol de coop

Qui s'en charge ? Biblio et langages
- Abstraient les IPC sys
- Offrent des fctionnalités et protocoles clé en main
- Ex : requête https en JavaScript, JEE, D-Bus…

→ Il est normal de les utiliser quand c'est adapté

410 – SIGNAUX

Forme d'interruption logicielle
- Analogie avec les interruptions matérielles
- Permet d'expédier à un prc une info urgente

Comportement asynchrone
- Un signal est envoyé
- Il sera reçu et traité au moment opportun

Sémantique des signaux
Liste des signaux
- Les signaux sont catalogués
- La liste est fixée
- Chacun est documenté

→ signal(7)

Un gestionnaire de signaux par prc
- Chq prog gère toutefois les signaux comme il veut
- La sémantique doit être documentée dans le prog (man)
- En particulier si elle diverge du catalogue

Exemples de signaux
SIGINT
- Ctrl C génère ce signal dont le comportement par défaut est d'arrêter le prc

SIGSEV
- Une err de seg provoque l'expédition de ce signal au prc fautif

kill
- Cmd kill(1). Envoie signal SIGTERM par défaut.
- Syscall kill(2)
- **Question** : kill est souvent une cmd interne du shell, pourquoi ? Raison technique : le shell est déjà en train de s'exécuter : si on veut arrêter un prc, on ne veut pas lancer un nouveau prc pour aller l'arrêter (bug poss).

Actions poss pour un signal
Pour chq catégorie de signal, un prc peut
- Accepter le comportement par défaut
  En général, arrêt du prc
- Ignorer le signal (pas tous)
- Gérer le signal (pas tous)

Gestion des permissions
- Seuls les prc d'un mm user peuvent s'envoyer des signaux
  Et root (RTFM pour les détails)
- Pas de kill sur le prc du voisin

Quelques signaux

Signal	Valeur	Action	Description
SIGHUP	1	T	Le terminal se ferme
SIGINT	2	T	Ctrl C au clavier
SIGKILL	9	TD	Terminer le prc
SIGSEGV	11	M	Err de seg
SIGCHLD	-	I	Term d'un enfant

- Action par défaut : T = terminer, D = défaut obligatoire (on ne peut pas le gérer, ni l'ignorer (intraitable), M = image mém, I = ignorer

Gestion classique des signaux en deux étapes
Écrire la fct gérante (en C classique)
- Signature simple void foo(int sig) (pour sa_handler)
- Ou complète void bar(int sig, siginfo_t* info, void* uctx) (pour sa_sigaction)

Associer fct et signal
- sigaction(2)
- Struc struct sigaction un peu pénible
- sigemptyset(3) et cie

Extra
- pause(2) suspend l'exécution jusqu'à un signal
- strsignal(3) et psignal(3) pour le texte des signaux

sigaction.c
```
#include<stdlib.h>
#include<stdio.h>
#include<unistd.h>
#include<signal.h>
#include<string.h>
void gere(int sig) {
  printf("Reçu %d: %s\n", sig, strsignal(sig));
  exit(1);
}
int main(void) {
  struct sigaction action;
  sigemptyset(&action.sa_mask);
  action.sa_flags = 0;
  action.sa_handler = gere;
  sigaction(SIGINT, &action, NULL);
  sigaction(SIGTERM, &action, NULL);
  pause();
}
```

Info supplémentaires
- Pour ignorer un signal, mettre SIG_ING dans sa_handler
- Pour l'action par défaut, mettre SIG_DFL dans sa_handler
- Les signaux d'une mm catégorie ne sont pas empilés
  Une rafale d'un mm signal peut activer une seule invocation de la fct gérante

Bloquer (masquer) les signaux
- Retarde la gestion de signaux jusqu'au déblocage
- sigprocmask(2) pour manipuler le masque de signaux bloqués
- sa_mask de sigaction(2) permet de masquer des signaux automatiquement pendant l'exec de la gérante

Voir les signaux
- /proc/PID/status montre l'état des signaux « Sig* »
- sigpending(2) voir les signaux en attente

Threads, fork et exec
pthreads
- Partagent : les gérantes, les signaux ignorés
- Copie : les signaux bloqués (masque des signaux)
- Fctionnalités fines existent pthread_kill(3), pthread_sigmask(3)
- Certains signaux en attente peuvent être partagés ou pas

fork
- Hérite : les gérantes, les signaux ignorés et bloqués
- Vide : les signaux en attente

execve
- Préserve : les signaux ignorés, bloqués et en attente
- Vide : les gérantes

Interruption des syscalls
- Un signal reçu et géré peut interrompre certains syscalls
- Syscalls dits « interruptibles »
- Détail dans le man de chacun des syscalls (ou signal(7))
- Prc dans l'état POSIX « S » selon ps

Qu'est-ce qui se passe alors
1. Prc dans syscall
2. Signal attrapé; gérante invoquée; gérante terminée (return)
3. Syscall terminé de force (interrompu)
   - Retourne EINTR (si pas commencé)
     - Ou autre val si travail partiellement réalisé
   - Sauf si SA_RESTART dans sa_flag de sigaction(2)
     - Mais pas pour tous les syscalls
   - RTFM pour les détails

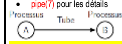
Synchronisme
L'approche asynchrone de sigaction(2) a des défauts
POSIX
- sigwaitinfo(2), sigtimedwait(2), sigsuspend(2), sigwait(3)
- Attend des signaux
- Note : Bloquer les signaux avant avec sigprocmask(2) ou autre

Linux
- signalfd(2)
- Crée un descripteur de fich spécial
- Permet de gérer les signaux comme des évén (« tout est fich »)

→ Attendre un signal avec poll(2), select(2)…

420 – TUBES

- Canal de comm unidirectionnel avec deux bouts
- Les octets écrits au bout en écr (write(2))
- Sont lisibles dans l'ordre au bout en lect (read(2))
- Flot d'octets (stream) : pas de concept de msg
- Les octets lus sont consommés
- pipe(7) pour les détails

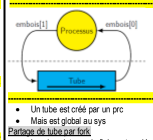


Tube et prc
Descripteurs de fich
- Pour un prc, un bout de tube est un descripteur
- Chq extrémité se manipule comme fich ouvert read(2), write(2), close(2), dup2(2), poll(2), etc.
- Mais pas lseek(2) (err ESPIPE)

Niveau noyau
- Esp mém du SE
- L'esp et son accès sont gérés par le SE
- Capacité limitée (64ko défaut actuel sous Linux)
- Mais ce n'est pas un prob
- Libéré automatiquement quand plus utilisé

Deux sortes de tubes
Tubes simples (majoritairement utilisés)
- Création : syscall pipe(2)
- « Retourne » deux descripteurs de fich
- int fds[2]; pipe(fds);
- fds[0] le bout en lect
- fds[1] le bout en écr
- Astuce mnémotech : 0=stdin 1=stdout

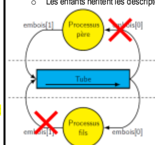
Tubes nommés
- Création : mkfifo(1) et mkfifo(3)

Tubes simples


Communication par tube
- Un tube est créé par un prc
- Mais est global au sys

Partage de tube par fork
- Les descripteurs de fich sont copiés
- Les bouts de tubes sont partagés

Communications
- Entre parent et enfant
  - Le parent crée le tube
  - L'enfant hérite les descripteurs
- Entre deux enfants
  - Le parent crée le tube
  - Les enfants héritent les descripteurs



Synchronisation
Lecture
- S'il y a des données dans le tube
  - read lit le max d'octets
- Si le tube est vide
  - Si un écrivain existe : read bloque
  - Si aucun écrivain : read retourne 0 (fin de tube)

Écriture
- S'il y a aucun lecteur
  - Signal SIGPIPE envoyé (par défaut, termine le prc)
- S'il y a un lecteur (ou plus)
  - Si assez de place : write écrit tous les octets
  - Si le tube est plein (ou presque) : write bloque

Contrôle de flux
Lecteur qui va trop vite
- Bloqué jusqu'à ce qu'un écrivain écrive
- Ou plus données ni d'écrivain (read retourne 0)

Écrivain qui va trop vite
- Bloqué jusqu'à ce qu'un lecteur consomme
- Ou que plus de lecteurs (SIGPIPE)

Questions
- Pourquoi ce n'est pas symétrique ? (0 vs. SIGPIPE) Les 2 bouts du tube n'ont pas la mm fct. 0 : normal : il n'y a plus de données à lire, on arrête. SIGPIPE : prob : si un prc essait de transmettre qqchose et qu'il n'y a rien, c'est qu'il y a eu un prob.
- Comment gérer SIGPIPE ? On le traite comme on veut.

Extra
Opérations atomiques
- PIPE_BUF (512 min, 4096 chez Linux)
- write écrits PIPE_BUF octets (ou moins) atomiquement
- Atomiquement = écrit d'un coup sans que d'autres écr concurrentes s'entrelacent

Entrées-sorties non bloquantes
- Flag O_NONBLOCK poss (via fnctl(2))
- Les règles de sync et d'atomicité changent
- RTFM

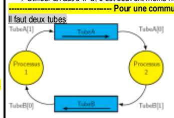
Danger : interblocage
Comment se bloquer tout seul ?
```
#include<unistd.h>
int main(void) {
  int embois[2];
  char buf;
  pipe(embois);
  read(embois[0], &buf, 1);
}
```
Question : Comment se bloquer à deux ? On crée un pipe et on va lire (read) : Le read va être bloquant, car il va détecter qu'il y a l'embois[1] et un descripteur qui pointe vers l'écrivain : il se dit qu'il y a possibilité de recevoir de l'info à un moment donné et il va attendre.

Bonnes pratiques
Un seul lecteur et un seul écrivain
- « Tjrs par deux ils vont, ni plus, ni moins » – Yoda

Fermer les bouts inutiles
- Laisser des bouts trainer cause des prob de sync
- Souvent : lecteur bloqué, car un bout d'écrivain reste quelque part

Plsrs écrivains et/ou lecteurs ?
- C'est techment poss, mais :
- Bien comprendre les règles de sync et d'atomicité
- Les clients doivent être coopératifs
- Msg de taille fixe aide bcp

→ Utiliser un autre IPC, c'est souvent moins risqué

Pour une communication bidirectionnelle
Il faut deux tubes


- Des sys offrent des tubes bidirectionnels (pas portable !)
- Utiliser un autre IPC, c'est souvent plus simple

Linux et pseudo fich tubes
- Ouvrir un pseudo fich tube de /proc/PID/fd est poss
- Le mode d'ouverture indique quel bout du tube on obtient

→ Permet d'ajouter des lecteurs et des écrivains
→ Mm si c'est souvent pas une bonne idée

Tubes nommés
Limites des tubes simples
- Via héritage des prc
- En créant le tube d'avance

→ Communication entre prc indépendants difficile

Principe des tubes nommés
- Les tubes nommés ne sont pas hérités, mais désignés
- Donc plus besoin d'hériter des descripteurs
- Ni de créer le tube d'avance

Caractéristiques
- Exactement comme un tube simple
- Mais : ouverture d'un tube par un nom
- Plus : gestion des droits
- Plus : mécanisme de rendez-vous entre prc

Fich spécial « tube »
- Crée avec mkfifo(1) et mkfifo(3) (et mknod(2))
- L'inode (via chemins) désigne le tube
- Les droits du fich sont les droits d'accès au tube
- Ouvrir (open(2)) le fich c'est accéder au tube
- Le fich est et reste vide
  - Le tube est entièrement en mém
  - Le fich n'est qu'une astuce pour désigner

Rendez-vous
- open(2) bloque jusqu'à avoir un lecteur et un écrivain
- Le tube se comporte ensuite comme un tube simple

→ Mm sync, mm atomicité

Substitution de prc
Chez bash(2) : <(CMD)
Principe
- Exécute CMD dans un prc indépendant
- Où la sortie standard de CMD est redirigée dans un tube
- Substitue l'arg <(CMD) par le chemin du tube
- Quelqu'un qui ouvrira ce chemin sera connecté au tube

Deux implémentations
- Pseudo fich tubes (proc(5)) si dispo
- Tube nommé sinon

430 – SOCKETS

Communication par sockets
- POSIX sockets alias BSD sockets alias Berkeley sockets
- Pour la communication réseau entre prc socket(7)
- API offerte par le SE

Socket ?
- Point de communication abstrait
- Boîte d'émission et de réception

→ Un socket est un descripteur de fich

Ceci n'est pas un cours de réseau
- On fait juste communiquer des prc
- On implémente des protocoles de communication
- Et on expose des abstractions et services aux prc

→ C'est la responsabilité du SE

API des sockets
API commune
- Diff et nbreux protocoles
- Diff types de communication
- Y compris propriétaires ou désuets

API générale
- Abstractions et syscalls communs
- Mais détails spécifiques à Chq protocole
- Et à Chq variante Unix
- API complexe avec défauts de conception historiques : berk!

Autre sockets
- « Socket » devenu un terme générique
- Autres langages et sys ont leur propre API de sockets
- API souvent proche (concepts et vocabulaire), parfois meilleure

Types de communication
- 3 dimensions principales
- Nbreuses variations spécifiques

Granularité
- Flux d'octets (stream)
- Messages (datagram, packet)

Connectivité
- Connecté et bidirectionnel : modèle client-serveur
- Non connecté : modèle pair à pair

Fiabilité (réseau principalement)
- Fiable : service garanti, obligation de résultat
  Risques de sacrifices : moins de débit et plus de latence
- Non fiable : service au mieux, obligation de moyen
  Risques de pertes de données, modif du contenu, pertes de l'ordre, duplications

Petite sélection de syscalls
- socket(2), socketpair(2) création de sockets
- bind(2), listen(2), accept(2) coté serveur
- connect(2) coté client
- write(2), send(2), sendto(2), sendmsg(2) émission
- read(2), recv(2), recvfrom(2), recvmsg(2) reception
- close(2), shutdown(2) fermeture
- getsockopt(2), getsockopt(2), ioctl(2) configuration
- getsockname(2), getpeername(2) identification
- lseek(2) bien évidement interdits (err ESPIPE)

Création de socket
socket(int domain, int type, int protocol)
Domaine = famille de protocoles
- AF_INET pour IPv4 (ip(7)) ou AF_INET6 pour IPv6 (ipv6(7))
- AF_UNIX (ou AF_LOCAL) pour socket Unix
- plus de 20 chez Linux, AF = address family

Type = sémantique de la communication
- SOCK_STREAM : flux d'octets, connecté, fiable
  Ex : TCP chez IP (tcp(7)). Analogie: téléphone
- SOCK_DGRAM : msgs, non connecté, non fiable
  Ex : UDP chez IP (udp(7)). Analogie: courrier postal
- SOCK_SEQPACKET: msgs, connecté, fiable

Protocole
- Protocole particulier si plus d'un pour un domaine et un type
- 0 = protocole par défaut

Socket du domaine Unix
- AF_UNIX (ou AF_LOCAL). Voir unix(7)
- SOCK_STREAM, SOCK_DGRAM ou SOCK_SEQPACKET

Ressemblances avec les tubes
- Communication efficace via la mém
- Zones de mém gérées par le SE
- Prc lisent/écrivent dans des descripteurs
- Anonymes ou nommés
- Sync
  - Lect si vide: bloquée ou 0 si aucun écrivain
  - Écrivain SIGPIPE si aucun lecteur ou bloqué si plein

Différence avec les tubes
- Utilise l'API des sockets POSIX
- Bidirectionnel
- Connecté ou non connecté
- Flux d'octets ou messages

Adr de socket
- Désignation d'un socket existant ou potentiel
- Struct C semi-opaques, fragiles et contraignantes (berk!)
- Détails spécifiques à chq domaine
  Ex. chez IP: adr IP + num de port

Structures d'adr
- struct sockaddr: structure abstraite
  - Utilisée dans les signatures des syscalls
- struct sockaddr_XXX: une version spécifique à Chq domaine
  - Utilisées pour allouer et accéder aux champs
  - struct sockaddr_in6 pour IPv6
  - struct sockaddr_un pour les sockets Unix
- struct sockaddr_storage structure assez grande pour stocker n'importe quelle structure spécifique

→ On caste allègrement entre des pteurs de ces types (berk!)

Sockaddr du domaine Unix
```
struct sockaddr_un {
  sa_family_t sun_family;      /* AF_UNIX */
  char        sun_path[108];   /* Chemin */
};
```
Attention, sun_path a une taille max (berk!) non portable (reberk!)

Fich spécial socket
- Utilisé pour « nommer » les socket*
- Type « s » selon ls -l
- Créé par bind(2)
- Supprimé par unlink(2)
- open(2) échoue (ENXIO)

*Linux offre aussi des sockets avec des noms « abstraits » indépendants du sys de fich (non portable).

Envoyer et recevoir
Envoyer
- write(int fd, const void *buf, size_t len)
- send(int fd, const void *buf, size_t len, int flags)
- sendto(int fd, const void *buf, size_t len, int flags, const struct sockaddr *addr, socklen_t addrlen)
- sendto = send + addr = write + flags + addr
- sendmsg(int fd, const struct msghdr *msg, int flags)

Recevoir
- read(int fd, void *buf, size_t len)
- recv(int fd, void *buf, size_t len, int flags)
- recvfrom(int fd, void *buf, size_t len, int flags, struct sockaddr *addr, socklen_t *addrlen)
- recvfrom = recv + addr = read + flags + addr
- recvmsg(int fd, struct msghdr *msg, int flags)

Note: ne pas préciser addr si connecté.

Mode connecté
Serveur
- bind(int fd, const struct sockaddr *ad, socklen_t adlen)
  Expose une « adr » publique
- listen(int fd, int backlog)
- Prépare un serveur à recevoir des clients
- backlog est soumis au culte du cargo (berk!). SOMAXCONN est bien.
- accept(int fd, struct sockaddr *ad, socklen_t *adlen)
- Récupère ou attend le prochain client
  - Retourne un nouveau socket, connecté directement au client
  - On a donc un socket d'écoute + un socket par client connecté

Client
- connect(int fd, const struct sockaddr *ad, socklen_t adlen)
  - Se connecte à un serveur spécifique
  - Retourne 0 si réussi, fd est maintenant connecté

  → read et write fonctionnent !

Exemple de client socket unix
```
#include "machine.h"
int main(int argc, char **argv)
{
  int sock = socket(AF_UNIX, SOCK_STREAM, 0);

  struct sockaddr_un addr;
  addr.sun_family = AF_UNIX;
  strncpy(addr.sun_path, "sock", sizeof(addr.sun_path)-1);

  int res = connect(sock, (struct sockaddr*)&addr, sizeof(addr));
  if(res==-1) { perror("connect"); exit(1); }

  write(sock, "Hello", 6);

  char buf[6];
  read(sock, buf, 6);
  printf("reçu: %s\n", buf);

  close(sock);
  return 0;
}
```

Exemple de serveur socket unix
```
#include "machine.h"
int main(int argc, char **argv)
{
  int sock = socket(AF_UNIX, SOCK_STREAM, 0);

  struct sockaddr_un addr;
  addr.sun_family = AF_UNIX;
  strncpy(addr.sun_path, "sock", sizeof(addr.sun_path)-1);

  int res = bind(sock, (struct sockaddr *) &addr, sizeof(addr));
  if (res == -1) { perror("bind"); exit(1); }
  listen(sock, SOMAXCONN);

  int cli = accept(sock, NULL, NULL);
  write(cli, "World", 6);
  char buf[6];
  read(cli, buf, 6);
  printf("reçu: %s\n", buf);
  close(cli);

  close(sock);
  unlink("sock");
  return 0;
}
```

Modèles populaires de serveurs
Un client après l'autre
- Boucle principale de accept(1)
- Traite Chq client entièrement, et dans l'ordre
- Prob
  - Traitements courts seulement
  - Un client peut bloquer les autres

Multiplexage
- Une liste de clients connectés
- Boucle principale avec un select(2) ou poll(2)
  - surveille le socket d'écoute + chacun des clients connectés
  - socket d'écoute bouge = on accepte un nouveau client
  - socket d'un client bouge = on traite sa demande
- Prob : messages courts seulement, pas adapté aux cas compliqués

Multithread
- Un thread principal écoute
- On lance un nouveau thread par client (ou pool de threads)
- Prob : programmation multithread

Multiprc
- Un prc principal écoute
- Un sous-prc (fork(2)) par client (ou pool de prc)
- Prob : lourd et isolation des clients
- Avantage : robuste et isolation des clients

Données auxiliaires
- Données spécifiques supp aux msg
- Alias « messages de contrôle » (cmsg)
- Contenu sémantique et spécifique :
  Contenu ont du sens pour le SE
- Mais ce qui est poss est spécifique à chq domaine
- recvmsg(2) et sendmsg(2) pour les utiliser
- cmsg(3) pour y accéder
- API horrible (berk!)

Descripteurs de fich auxiliaires
- Utilisable dans les sockets du domaine Unix
- SCM_RIGHTS passe des fich ouverts
- L'émetteur attache des descripteurs de fich
- Le sys crée des descripteurs dans le prc récepteur
- C'est pas forcément les mm num de descripteur
- Mais c'est les mm fich ouverts

Paire de sockets
- socketpair(2) crée deux sockets connectés
- Ressemble fortement à pipe(2)
- Mais bidirectionnel !
- Msg poss (pas slm flux d'octets) !

500 – SYNCHRONISATION

Systèmes concurrents
- Des éléments logiciels (voire matériels)
- Sont capables de s'exécuter en « mm temps »
- Indépendamment du moment ou de l'ordre de leur exec
- Sans tout briser

Questions
- Ça veut dire quoi en « mm temps » ? Prochains diapo.
- Quel rapport avec les SE ? Prochains diapo.

En mm temps ?
Concurrence
Un élément logiciel s'exécute avant que les autres finissent
L'ordre des exec de chacun est variable
- Changements de contexte et politiques d'ordonnancements
- Interruptions matérielles
- Signaux logiciels
- Programmation événielle
- Invocation de sous-programmes (en tirant vraiment l'élastique)

Parallélisme
Exec physiquement au mm moment
- Architectures multiprc et multicoeurs
- Voire sys distribués
  Mais rendu là on a d'autres difficultés en plus

Exemple: incrément
- Suspects: Deux threads t1 et t2 d'un mm prc
- Victime: Une variable globale i partagée
- Code du crime, exécuté par les deux threads:
  i++;

Où est le prob?
```
; assembleur x86    ; pep8        ; Français
movq i(%rip), %eax  ; LDA  1,d    ; Charge i dans A
addq $1, %eax       ; ADDA 1,i    ; Incrémente A
movq %eax, i(%rip)  ; STA  i,d    ; Sauve A dans i
```

On a pas de prob !
Thread 1	Thread 2	i	A(t1)	A(t2)
		0	?	?
LDA i,d		0	0	?
ADDA 1,i		0	1	?
STA i,d		1	1	?
	LDA i,d	1	1	1
	ADDA 1,i	1	1	2
	STA i,d	2	1	2

On a un prob !
Thread 1	Thread 2	i	A(t1)	A(t2)
		0	?	?
LDA i,d		0	0	?
ADDA 1,i		0	1	?
	LDA i,d	0	1	0
	ADDA 1,i	0	1	1
	STA i,d	1	1	1
STA i,d		1	1	1

Instructions machines
Concurrence vs parallélisme
- Monoprc
  Prob quand changement de contexte au mauvais moment
- Multiprc
  Probabilité de prob bien plus grande

→ Débogage difficile : syndrome « chez moi ça marche »

Architectures matérielles
i++ peut être une seule instruction « addq $1, i(%rip) »
- Monoprc: Plus de vraiment de prob
  - Changement de contexte avant ou après l'instruction
- Multiprc: Tjrs prob de concurrence
  - 1 instruction, mais Plsrs cycles
  - RAM partagée
  - Caches et cohérence mém

Quel rapport avec les SE ?
Traitement d'évén logiciels et matériels
- Des évèn fondamentalement imprévisibles
- Interruptions matérielles
- Syscalls de prc (si vrai parallélisme)

Performance des SE
Exploitation des possibilités de concurrence et parallélisme
- Traitements parallèles internes : threads sys
- Préemption sys : les noyaux modernes sont préemptifs
- Une approche « un seul prc en sys à la fois » fonctionne, mais est très limitante côté performance

Centre de service
- Offre de mécanismes de sync pour les prc

Classification des programmes concurrents
Disjointe
- Pas d'interaction entre entités logicielles

→ Facile mais ça n'arrive pas souvent

Compétitive
- Des ress partagées existent
- On veut s'assurer de leur disponibilité et cohérence

→ C'est un travail pour le SE

Coopérative
- Des éléments logiciels coopèrent
- La concurrence fait partie du programme
- C'est des modèles de programmation spécifiques

→ Le SE offre des services de sync
→ Mais il y a aussi des ress à gérer

Situation de compétition (race condition)
- Situation où le résultat est diff
- Dépendamment du moment ou de l'ordre d'exec

→ C'est souvent problématique

Résultats diff
- Tous pas forcément corrects

→ Bogue, y compris de sécurité

Ordre et moment
- Pas connus ou pas contrôlables
- Car ordonnancement, latence matérielle, évén externes…
- Donc situations difficiles à reproduire et à tester

→ Indéboguable (Heisenbogues)

Problématiques de concurrence partout

- Retrait bancaire

```
if(montant < solde) {
    solde =- montant;
    return montant;
} else {
    return 0; // Solde insuffisant
}
```

- Suppression d'un maillon d'une liste doublement chainée

```
if (current->next != NULL)
  current->next->prev = current->prev;
if (current->prev != NULL)
  current->prev->next = current->next;
```

Question : Trouver des scénarios problématiques ?

- Deux prc parallèles font fork(2)
  - Attribuer correctement un PID diff
  - Ne pas corrompre la table des prc
- Deux threads parallèles font malloc(3)
  - Attribuer correctement une zone mém distincte
  - Ne pas corrompre les struc internes du tas
- Deux prc lisent écrivent en mm temps dans un tube
  - Attribuer des octets diff (sans en perdre)
  - Ne pas corrompre les struc internes du tube
- Résoudre un chemin (path_resol(7))
  - Alors qu'un prc renomme ou déplace des répertoires
- Prob théoriques classiques de sync
  - Diner des philosophes
  - Producteurs et consommateurs (file bornée)
  - Coiffeur endormi (file d'attente)
  - Écrivains et lecteurs (accès concurrents en lect ou écr)

Besoin d'ordre et de discipline !

- Éviter les accès simultanés qui rendraient le sys incohérent
- Garantir une certaine équité
- Maintenir la performance
- Éviter que le sys ne se bloque

510 – SECTION CRITIQUE

Prob de concurrence

- Les threads (et prc)
- Le SE lui-mm
- Sont fortement concurrents, voire parallèles

→ Comment gérer cette concurrence correctement de façon programmative ?

Objectifs

- Contrôler les situations de compétition
- Prévenir la corruption de ress partagées
- Indépendamment du type de ress
- Rester efficace

Section crituque

Sc = Zone de code

- Zone de code = morceau de prog
- Attention, pas forcément contiguë

Sc = Zone d'exclusivité

- Exécuté que par un seul thread* max à la fois
- Qui manipule une ress potentiellement partagée

→ On protège une ress en contraignant l'exec du code qui manipule cette ress

*Sans perte de généralité, on utilise « threads », mais ça s'applique pareil aux prc monothread, tâches noyau ou toute autre entité logicielle en cours d'exec.

Un thread en section critique

- N'est pas nécessairement actif à 100%
- Il peut faire des syscalls bloquants (et devenir bloqué)
- Il peut être préempté (et devenir prêt)

Sc en bref

- Tant qu'un thread n'est pas sorti
- Aucun autre thread ne peut y rentrer

Les 4 règles des section critique

1) Au max, un seul thread à la fois en sc
2) Pas de supposition sur la vitesse ou le nb de threads
3) Un thread hors sc ne bloque pas les autres
4) Pas d'attente infinie pour entrer en sc (famine)

Sol qui marche pas

```
long i; // ressource partagée
int flag; // booléen protegant la ressource

void inc(void) {
  while (flag) { } // on attend l'absence du flag
  flag = 1;          // on verrouille
  i++;               // on manipule
  flag = 0;          // on dévérouille
}
```

- Toute « stratégie » de ce type est à bannir !

Question

- Lesquelles des 4 règles sont violées ? 1)
- Trouvez un scénario (ordonnancement) où ça ne fonctionne pas

Exclusion mutuelle stricte à tours

```
long i; // ressource partagée

extern int nb; // nombre de threads
int tour;      // à qui c'est le tour?

void inc(int t) { // on est le thread 't' (de 0 à nb-1)
  while (t != tour) {}  // on attend notre tour
  i++;                  // on manipule
  tour = (tour+1) % nb; // on passe au suivant
}
```

Question : Lesquelles des 4 règles sont violées ? 1)

Sol de Peterson (1981)

- Pour deux threads seulement

```
long i; // ressource partagée

int flag[2]; // qui est interessé ?
int tour;    // à qui le tour (si les deux en veulent) ?

void inc(int k) { // k c'est moi, !k c'est l'autre
  flag[k] = 1; // on veut entrer
  tour = !k;   // on est poli
  while (flag[!k] && tour == !k) { } // attente
  i++;         // on manipule
  flag[k] = 0; // on n'en veut plus
}
```

- Se généralise à un nb quelconque de threads.

Barrière mémoire

Instruction (ou indication) destinée:

- Aux prc
  Force les écr et lect du bon côté de la barrière
- Au compilateurs C
  Prévient les optimisations qui changent la sémantique
- Coût non négligeable

Exemples

- Instruction mfence en x86
- Pas de mot clé C standard
- atomic_thread_fence de C11 stdatomic.h
- Extension C de gcc : __atomic_thread_fence
  (et le plus ancien __sync_synchronize)

Volatilité

- Mot clé C volatile
- Déclare une donnée comme mutable par qqchose d'autre (comme un autre thread)

Pour le compilateur seulement

- Informe qu'une modif indépendante est poss
- Et qu'il doit éviter des optimisations
- Par défaut, le compilateur n'est pas conservateur !
- Les opt de compilation changent le comportement du code

Quand le compilateur voit

« int x = 0; while(x == 0) {} »

- Si x n'est pas volatile
  On peut juste implémenter une boucle infinie
- Si x est volatile
  On doit tester x à Chq tour, « au cas où… »

Bonne ou mauvaise chose ?

- La présence de volatile dans du code est souvent douteuse
- Son utilisation ne permet pas magiquement de résoudre les prob de concurrence
- Son coût est non négligeable
- volatile considered harmful

Peterson + volatile + barrières mém

```
#include <stdatomic.h>

long i; // ressource partagée

volatile int flag[2]; // qui est interessé ?
volatile int tour;    // à qui le tour ?

void inc(int k) { // k c'est moi, !k c'est l'autre
  flag[k] = 1; // on veut entrer
  tour = !k;   // on est poli
  atomic_thread_fence(memory_order_seq_cst);//barrière
  while (flag[!k] && tour == !k) { } // attente
  atomic_thread_fence(memory_order_seq_cst);//rebarrière
  i++;          // on manipule
  atomic_thread_fence(memory_order_seq_cst);//rerebarrière
  flag[k] = 0; // on n'en veut plus
}
```

Le matériel à la rescousse

Idée : empêcher le changement de contexte

- Masquer les interruptions matérielles au niveau du prc (dont l'horloge)
- Verrouiller le bus (en multiprc)

Prob

- Grain grossier
- Couteux
- Seul le noyau peut faire ça
  (ok pour lui, mais pas pour les prc)

Instructions machine atomiques

- C11 _Atomic et stdatomic.h, extension gcc ou assembleur
- Permet des manipulations atomiques :
  Indivisible pour l'observateur

Exemple : incrément atomique

- lock add (x86)
- __atomic_add_fetch (gcc)
- ++ sur un type _Atomic (C11)

Mise en oeuvre matérielle

- Accès mém exclusif de la donnée le temps de l'exec
- Ex : verrouillage des lignes de cache mém

Limites

- Coût non nul
- Slm certaines instructions et val simples

Incrément atomique

Extension gcc

```
long i; // Ressource partagée
void inc(void) {
  // On manipule sans verrous
  // Directement fonction built-in gcc
  __atomic_add_fetch(&i, 1, __ATOMIC_RELAXED);
}
```

C11 avec _Atomic

```
_Atomic long i; // Ressource partagée
void inc(void) {
  i++; // on manipule
}
```

En vrai ?

Le compilateur compile vers des instr mach atomiques

Il peut ajouter aussi des barrières

Sections critiques plus grosses ?

On implémente un verrou atomique

- Instructions « test and set », « compare and exchange »…
- x86: xchg, lock cmpxchg…
- gcc: __atomic_test_and_set, __atomic_compare_exchange…
- C11: atomic_flag_test_and_set, atomic_compare_exchange…

```
#include <stdatomic.h>

long i; // ressource partagée
atomic_flag flag; // booléen protegant la ress

void inc(void) {
  while(atomic_flag_test_and_set(&flag)) {}
  i++; // on manipule
  atomic_flag_clear(&flag);
}
```

- C'est une version fctionnelle de la « sol qui marche pas »

Attente active (*spinlock*)

while (...) { }, infinite : le temps que les choses se libère.

- Quand ça fonctionne, ça reste inefficace

→ Ça gaspille du temps prc à activement rien faire

Questions

Voici deux autres propositions :

- while (...) { sched_yield(); } †
- while (...) { sleep(1); }
- Pourquoi c'est pas vrm bcp mieux ? Gourmant. Boucle infinie.
- Y a-t-il des cas où c'est mm pire que la proposition initiale ? Non.

†En gros sched_yield(2) force un appel à l'ordonnanceur pour éventuellement donner le prc à un autre thread.

Limites des proposions à date

Objectifs

- Contrôler les situations de compétition
- Prévenir la corruption de ress partagées
- Indépendamment du type de ress
- Rester efficace

Limites

- Approches purement algorithmiques limitées
- Instructions machine spécifiques peu portables
- Bricolage bas niveau
- Potentiellement inefficace (spinlock)

Sol : Un nouveau niveau d'indirection

Vraie sol : demander au SE de nous lorsque les ress sont dispo (520)

- Langages, biblio et SE à la rescousse
- Ils fournissent des services et des modèles de sync
- Que les développeurs peuvent utiliser

520 – OUTILS DE SYNCHRONISATION

Concurrence et sc

On y arrive à la main, mais

- C'est compliqué
- C'est bas niveau
- C'est douteux d'un point de vue performance (attente active)

Peut-on faire mieux ?

- Le SE est ± capable de se débrouiller
- Qu'en est-il des prc et threads ?

Sol : Un nouveau niveau d'indirection

- SE, biblio et langages
- Ils fournissent des outils

→ Services et des modèles de sync clé en main

- Les développeurs peuvent les utiliser

Outils spécialisés

- La programmation concurrente reste complexe
- Ces outils ne suppriment pas les difficultés fondamentales
- Au mieux, ils les transforment et les déplacent…

Besoin de performance

- Rappel : les syscalls coutent cher
- Une partie des mécanismes est faite en esp user
  → Langages et biblio
- Une autre partie en mode noyau, par le SE
  → ordonnancement et états d'exec des prc

Concrètement

- Implémentés avec les tech primitives précédentes
- Garantissent l'efficacité et la fiabilité

→ Bienvenue dans la programmation concurrente moderne !

Éliminer l'attente active

Le SE gère le cycle de vie des threads

Sol à l'attente active

- Un thread veut entrer en sc déjà occupée
  On le bloque (passage de l'état actif à bloqué)
  On appelle l'ordonnanceur
- Un thread sort d'une sc
  Un autre thread était en attente ?
  On le réveille (passage à l'état prêt)
  Et on appelle l'ordonnanceur
- Le tout de façon performante !

Question : Quels sont les cas où l'attente active est préférable à un changement de contexte ? Lorsque le changement de contexte coûte plus cher : mieux vaut attendre.

Mutex (ou verrou, lock), de mutual exclusion

- Concept général de verrouillage de sc
- Mais détails spécifiques en fonction du contexte (SE, biblio, langage de programmation)

Opérations générales

- Verrouiller : ça entre ou ça attend
- Déverrouiller : ça débloque les autres
- Tenter : ça entre ou ça échoue

Variations

- Actif (spinlock) ou bloquante (passage à l'état bloqué)
- Rapide (un booléen), récursif (un compteur), avec détection d'err

Mutex pthread

- Fourni de base chez pthreads(7)
- pthread_mutex_lock(3), pthread_mutex_unlock(3), pthread_mutex_trylock(3)…
- Limités aux threads d'un mm prc
- RTFM pour les détails

```
#include<pthread.h>

long i; // Ressource partagée
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;

void inc(void) {
  pthread_mutex_lock(&mut); // on verouille
  i++; // on manipule
  pthread_mutex_unlock(&mut); // on déverouille
}
```

Sémaphore

- Concept historique spécifique (Dijkstra, 1962)
- C'est un cpteur de ress
- Le cpteur bloque s'il n'y a plus de ress
- Sémantique atomique garantie (détail d'implémentation)
- Pas d'attente active et pas de famine
- Cas particulier: sémaphore binaire (deux val poss), ressemble au mutex d'un point de vue de l'implémentation

Sémaphore : pseudo-code

Sémaphore { entier val; liste_prc L; };

```
Demander(Sémaphore S)              Libérer(Sémaphore S)
S.val -= 1;                        S.val += 1;
si S.val < 0 alors                 si S.val <= 0 alors
  | ajouter demandeur à S.L;         | enlever un processus de S.L;
  | le passer à bloqué;              | le passer à prêt;
fin                                fin
```

Remarque de vocabulaire

- Demander = Entrer = Down = P = proberen = tester
- Libérer = Sortir = Up = V = verhogen = incrémenter

Sémaphores Unix

- Ress gérées globalement au niveau du sys
- Persistant jusqu'à l'arrêt du sys ou une libération explicite
- Partageables entre threads, prc et users
- Sémaphores POSIX : sem_overview(7)
- Sémaphores System V : semget(2), semop(2)

```
#include <semaphore.h>
long i; // Ressource partagée
static sem_t sem;
void inc_init(void) { // initialisation
  sem_init(&sem, 0, 1); // initialise à 1
}
void inc(void) {
  sem_wait(&sem); // verrouillage
  i++;            // on manipule
  sem_post(&sem); // déverouillage
}
```

Sémaphore vs Mutex

Sémaphore

- Cpteur de ress : atomique, efficace et équitable
- Ceux qui incrémentent sont pas forcément ceux qui décrémentent

Mutex sys

- Délimite une sc qui protège une ress partagée
- Le thread qui déverrouille est celui qui a fait le verrouillage initial
- Info utile pour le SE

Avantages des mutex sys

- Déverrouillage des mutex d'un thread qui termine
- Inversion de priorité poss
- Vérification d'err poss:
  - Un thread déverrouille un mutex sans l'avoir verrouillé
  - Situation d'interblocage

Autres outils et tech de synchronisation

- Variable de condition (file d'attente + service de réveil)
- Moniteur (sous-prog + mutex implicite + variables de condition)
- Barrière
- Verrou lect-écr
- RCU (read-copy-update) : tech sans verrouillage
- Struc de données parallèles clé en main
- Opérations atomiques (C11)…

Ce sont des outils et abstractions de programmation

Futex (fast userspace mutex)

- Bloque un prc jusqu'à un réveil explicite
- Bas niveau et délicat

→ Err classique : on bloque un prc
  Pile au moment où la condition du blocage disparait

- Sert aux biblio pour implémenter les autres mécanismes
  → Mutex pthread et autre
- futex(2) sous Linux

Exemple tentative futex

```
#define _GNU_SOURCE
#include <sys/syscall.h>
#include <unistd.h>
#include <linux/futex.h>
#include "myatomic.h"
long i; // ressource partagée
int flag; // 0=libre; 1=occupé
void inc(void) {
  while(xchg(&flag, 1) == 1) {
    // on s'endort si occupé
    syscall(SYS_futex, &flag, FUTEX_WAIT, 1, 0, 0, 0);
  }

  i++; // on manipule

  xchg(&flag, 0);
  // on réveille un endormi, s'il y en a
  syscall(SYS_futex, &flag, FUTEX_WAKE, 1, 0, 0, 0);
}
```

Exemple futex mieux

```
#define _GNU_SOURCE
#include <sys/syscall.h>
#include <unistd.h>
#include <linux/futex.h>
#include "myatomic.h"
long i; // ressource partagée
int flag; // 0=libre; 1=occupé; 2=endormi
void inc(void) {
  int c = cmpxchg(&flag, 0, 1);
  if(c != 0) {
    if (c != 2) c = xchg(&flag, 2);
    while (c != 0) {
      syscall(SYS_futex, &flag, FUTEX_WAIT, 2, 0, 0, 0);
      c = xchg(&flag, 2);
    }
  }
  i++; // on manipule
  if(xchg(&flag, 0) == 2)
    syscall(SYS_futex, &flag, FUTEX_WAKE, 1, 0, 0, 0);
}
```

530 – INTERBLOCAGE

Alias

- Deadlock, Verrou fatal, Étreinte fatale, Embrasse mortelle

Principe

- Plsrs prc* sont bloqués entre eux et ne peuvent progresser

*Sans perte de généralité on utilise « prc », mais ça s'applique pareil aux threads, tâches noyau ou toute autre entité logicielle en cours d'exec.

Ressources

Quelques ress (au sens large)

- Imprimante, CPU, Sémaphore, Sc (mutex)

Ress et interblocages

- Les interblocages découlent de l'allocation des ress

Évén liés aux ress

- Demander, Utiliser, Libérer la ress

Si un prc demande une ress déjà prise

- Err, Attente, Attente temporisée

Interblocage

Définition plus formelle

- Un ens de prc sont en interblocage si Chq prc dans cet ens est en attente d'un évén que slm un autre prc de ce mm ens peut déclencher — Tanenbaum
- L'évén peut-être est la libération d'une ress

En cas d'interblocage un prc ne peut

- Ni continuer son exec
  Car il est bloqué
- Ni débloquer un autre prc
  En libérant une ress
  Car il est bloqué

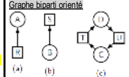
Caractérisation d'un interblocage

Les 4 conditions nécess et suffisantes de l'interblocage

- Exclusion mutuelle
  La ress est soit dispo, soit assignée
- Détention multiple (hold and wait)
  Un prc qui détient une ress peut en demander d'autres
- Pas de réquisition
  Une ress détenue par un prc doit être liberée par lui
- Attente circulaire
  Il doit y avoir un cycle dans les attentes d'évén

Description des allocations

Graphe biparti orienté



- (a) Ress R assignée au prc A
- (b) Prc B demande et attend S
- (c) C et D sont en interblocage

Gestion des interblocages

Quatre stratégies

- Ignorer le prob
- Détecter et résoudre
- Prévenir le prob
- Éviter dynamiquement

Ignorer le problème

Principe

- Prétendre que le prob n'existe pas

Raisonnable si

- Interblocages rares
- Autres solutions trop coûteuses/restrictives

Avantage

- Facile à comprendre, à mettre en œuvre

Détecter et résoudre

Principe

- Vérifier si les prc sont en interblocage et les débloquer

Comment savoir s'il y a interblocage

- Modélisation et analyses de graphe
- Tester

Comment débloquer (sans tout casser)

- Échec du verrouillage
- Retirer de force une ress
- Restauration d'un état antérieur (rollback)
- Éliminer un prc

Prévenir le prob

Principe

- Éliminer une condition de l'interblocage

Exemples

- Spooling : seul un prc a la ress
- Ress toutes demandées d'un coup
- Permettre la préemption
- Ordonner les ress (donc les demandes)

→ Aucun n'est nécessairement faisable

Éviter dynamiquement

Principe

- Forcer l'ordonnanceur à faire le bon choix
- C'est poss via des info supp

Idée

- Un état est sûr s'il existe une séquence d'allocations qui permet aux prc d'aller jusqu'au bout
- Exécuter une allocation que si l'état qui en résulte est sûr

Résolutions en pratique

Pas de solution ultime

- Le coût et l'efficacité d'une tech dépendent fondamentalement de la nature des ress

En pratique,

- Les SE actuels ignorent le prob pour les users
- Seuls les SE critiques prennent éventuellement en cpt ce genre de prob

Problème cousin - Livelock

Jeu de mots sur deadlock

- Les prc ne sont pas bloqués
- Mais ne progressent pas non plus

→ Le CPU est utilisé seulement pour retenter

- Transformer un deadlock en livelock n'est pas un progrès

Problème cousin - Famine

Synonymes

- Starvation, Privation de ress

Définition

- Un gr de prc partagent une ress
- Sans interblocage
- Certains prc n'obtiennent jamais la ress

→ Prob de l'attente infinie

Problème cousin - inversion de priorité

Scénario explicatif

- 3 prc de priorité stricte P1 > P2 > P3
- P3 arrive, s'exécute, demande et acquiert une ress R
- P2 arrive et s'exécute (préempte P3)
- P1 arrive, s'exécute (préempte P2), demande la ress R… et passe à bloqué
- P2 s'exécute alors à nouveau

Prob

- P2 passe devant tout le monde
- P1 termine dernier, c'était pourtant le plus prioritaire

Solutions

- Revoir la conception: est-ce normal que P3 puisse bloquer P1 ?
- Renforcement de la priorité

Renforcement de la priorité

Augmentation temporaire de priorité

- P3 devrait passer avant P2
- Jusqu'à sortir de sa sc et libérer son mutex
- Pour pouvoir débloquer P1 le plus tôt poss

Nécessite la coopération du SE

- Mutex avec priorité statique (PTHREAD_PRIO_PROTECT)
  Le prc qui détient le mutex (P3) gagne en priorité
- Héritage de priorité (PTHREAD_PRIO_INHERIT)
  Le prc qui attend un mutex (P1) transfert automatiquement sa propre priorité au détenteur du mutex (P3)
- Renforcement aléatoire (ex Windows)
  Comme P3 détient un mutex, il va peut-être gagner un petit boost pour peut-être sortir de sa sc

Dîner des philosophes (Dijkstra)

Données

- 5 philosophes. Chacun pense ou mange (temps inconnu)
- 5 fourchettes
- 5 plats de spaghettis
- 2 fourchettes sont nécessaires pour manger

Prob

Comment faire tourner le sys sans bogue ?

- Sans corruption, Sans famine, Sans interblocage
- Efficacement

600 – GESTION DE LA MÉMOIRE

Rôles du SE

Répartir (allouer) la mém

- Pour les prc
- Pour lui-mm

→ Efficacement et sans gaspiller

Contrôler et protéger

- Isoler la mém des prc

→ Chq prc a l'impression d'être seul

Offrir des services

- Allocation dynamique
- Mém partagée
- Configuration de politiques
- brk(2), mmap(2)

Objectifs du chapitre

Comprendre

- Comment la mém est gérée par le SE (et le matériel)
- Les possibilités offertes par la gestion moderne de la mém
- Les algo liés à la gestion de la mém (utilisables dans d'autres contextes)

Mémoire

La bonne mém

- est rapide, grande, bon marché, non volatile, mais elle n'existe pas (encore)

En attendant : hiérarchie de mém

- Caches prc, RAM, Disques

Nos hypothèses

- Mém = grand tableau d'octets
- Accès matériel efficace (CPU ↔ RAM)
- Les prc vont et viennent

Dans la vraie vie ?

- Prc modernes (caches, pagination avancée, etc.)
- Parallélisme et des architectures multiprc et multicoeurs
- Architectures hétérogènes (NUMA, non uniform memory access)
- /proc/meminfo, /proc/vmstat

Problème à résoudre

- Adr explicites dans le code machine d'exécutables
  Comment lancer Plsrs prc ?
- Agrandissement de la mém allouée à un prc
  Que faire si on n'a pas la place autour ?
- Clone de prc avec fork(2)
  Mais que faire avec les pteurs existants ?
- Comment partager de la mém entre prc ?
- Comment avoir des modes d'accès (écr,exec) spécifiques ?

Question : Pourquoi on voudrait partager la mém entre prc ? Pour des raisons d'optimisation et de performance : on ne veut pas réécrire, revoir, recopier des choses qui sont déjà immuable.

Solution : un nouveau niveau d'indirection

- Ne plus permettre aux prc de pointer directement la mém
- Les adr utilisées par les prc (pteurs, opérandes des instructions machine…) ne sont pas des adr absolues en RAM

→ On convertit les adr logiques (des prc)
  En adr physiques (en RAM)

```
#include "machine.h"
int main(void) {
  pid_t p = fork();
  printf("pid: %d ; adr: %p ; val: %d\n", getpid(), &p, p);
  pause();
  return 0;
}
```

Mémoire physique (ou réelle)

Mém physique (ou réelle)

- La RAM (un grand tableau d'octets)
- free(1), /dev/mem, /proc/meminfo…

Adr physique (ou réelle)

- Un numéro d'octet dans la RAM

Esp d'adressage physique (ou réel)

- L'ensemble des adr physiques poss
- En général la taille du bus d'adr
- x86-64: actuellement ≈ 48bits (256To)

Mémoire logique (ou virtuelle)

Adr logique (ou virtuelle)

- Les adr utilisables par le logiciel
- Pteurs, registres, opérandes, etc.
- Attention: adr « linéaire » est parfois (abusivement) utilisé

Esp d'adressage logique (ou virtuel)

- L'ensemble des adr logiques poss
- En général la taille d'un pteur ou d'un registre
- x86-64: actuellement ≈ 48bits (voire 57bits)
  Les pteurs sont pourtant 64bits (économie!)

Mém logique

- La mém associée au logiciel qui s'exécute (prc)
- /proc/PID/mem, pmap(1) et colonne VSZ de ps(1)

Unité de gestion mém (MMU)

- MMU = memory management unit
- Composante matérielle, sur le microprc
- Traduit automatiquement et efficacement
  Adr logiques → adr physiques
- Les opérandes et pteurs sont en adr logique
- Ce qui circule sur le bus d'adr est en adr physique

→ C'est transparent pour le logiciel

Bonus

- Les param de traduction sont configurables (en mode noyau)
- MMU s'occupe aussi de vérifier la légalité des accès mém
  Faute CPU si accès à une adr mém logique non valide (selon les param configurés)

Matériel

- Accès direct du matériel (DMA) reste en adr physiques

SE et prc

Chq prc

- A des param de traduction mém spécifiques
- C'est sa « vue » personnelle de sa mém
- Son esp d'adressage logique est automatiquement (MMU) associé à des morceaux de mém physique (ou à des fautes CPU)

Changements de contexte

- Le SE reconfigure le prc
- Et param la traduction à celle du prc actif

Changements de contexte, en pratique

- Quelques registres à maj (voire un seul, CR3 chez x86)
- Coût non négligeable sur les CPU modernes

Méthodes de traduction

- Chq architecture matérielle est diffe et spécifique
- Plsrs possibilités en fonction des microprc
- Certains microprc offrent ou combinent plsrs approches

→ Nbux détails tech

Base et limite (historique)

- Tech historique, très simple et très limitée
- Deux registres spéciaux privilégiés : base et limite
- Traduction : physique = logique + base
- Vérification : logique < limite

Fctionnalités

- Chq prc a son bloc de mém qui part de l'adr 0 pour lui
- Le SE peut redimensionner ou déplacer les blocs de façon relativement transparente

Limites

- Mém relativement contiguë (sinon gaspillage)
- Pas de partage mém entre prc
- Pas de droits fins

Segmentation (historique)

- Tech historique, complexe, et limitée
- Le CPU permet d'avoir Plsrs segments paramétrés indépendamment
- Un segment ≈ un bloc base-limite + droits spécifiques
- Résout les limites du base-limite
- N'existe plus en x86-64
  Sauf via deux registres spéciaux FS et GS

Pagination

- La solution à tous les prob ?

610 – PAGINATION

Principe

- Découper toute la mém physique
  En page physique (ou cadres, frame, page frame) de taille fixe
  sysconf(_SC_PAGESIZE) donne la taille des pages du sys (4ko par exemple)
- Découper tout l'esp d'adressage des prc
  En pages logiques (ou page virtuelle) de mm taille
- Associer efficacement (CPU) les pages logiques aux physiques

Le gagnant actuel

- Offert par la plupart des prc
- Utilisé par la plupart des SE
- Assez couteux et complexe côté prc
- Simple, souple et puissant côté SE

Pagination pour le MMU

Adr logique décomposée

- Numéro de page logique
- Adr dans la page (décalage ou offset)
- Exemple: page de 4ko, 48 bits d'adr logique = 36 bits (numéro de page logique) + 12 bits (décalage (2^12=4k))

Traduction avec une table

- Associer numéro de page logique → numéro de page physique
- Les associations sont stockées dans la table des pages

Exemple très naïf

Pages

- Taille: 4 octets
- Taille du décalage (en bits): 2 bits : 00, 01, 10, 11

Physique

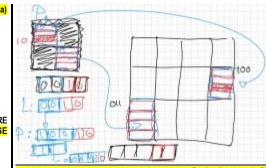
- Adr: 5 bits : # pages = 3 bits, offset = 2 bits
- Esp d'adressage (octets): 8 pages * 4 octets = 32 octets
- Nb de pages: 2^3 bits = 8 pages

Logique

- Adr: 4 bits
- Esp d'adressage (octets): 2^4 bits = 16 octets
- Nb de pages: 2^2 bits = 4 pages

Suite

- Page: 4 octets.. Adr physique: 5 bits. Adr logique: 4 bits.



Pagination pour le SE

- Une table des pages par prc
- Le SE
  - Configure et maintient Chq table des pages
  - Positionne la table du prc actif lors des changements de contextes

Chez Linux

- /proc/PID/pagemap (tableau binaire) pour Chq page logique, quelle page physique est associée (+ info supp)
- /proc/kpagecount (tableau binaire) pour Chq page physique, cb de pages logiques y sont associées

En vrai

- La mém des prc est bcp plus riche que ce que le prc offre
- La table des pages du MMU est trop bas niveau, trop spécifique et trop limitée
- Des struct de données additionnelles sont nécessaires
- Le sys gère des zones virtuelles regroupant Plsrs pages: les lignes de pmap(1)

Table des pages

Où est la table ?

- Registres ? Non, la table est trop grande !
- Un gros bloc en mém ? Où est ce bloc ?

Solution habituelle

- Registre privilégié pour l'adr de la table (CR3 chez x86)
- Tables d'indirection en RAM

Questions

- L'adr dans CR3 est-elle logique ou physique ? C'est une question d'implémentation : elle peut être les 2. Logique : pour des raisons de conformité et de cohérence. Physique : pour des raisons de performance.
- Un prc peut-il modifier la valeur du registre CR3 ? Non, slm l'OS.
- Un prc peut-il modifier la table des pages ? Non, slm l'OS.

Avantage de la pagination

Souplesse maximale

- Permet de mettre diff morceaux de mém
- Permet d'utiliser tout l'esp d'adressage (ou presque)
- Indépendant du nb et de l'utilisation des morceaux
- Possibilité d'avoir des droits fins (lect, écr, exec)
- Possibilité de partager des pages physiques entre prc
  - Pas forcément avec la mm page logique
  - Pas forcément avec les mms droits

Question : Une page physique peut-elle être associée à Plsrs pages logiques diff ? Oui, ex. fork, ce que les librairies partagent et font…

Trop couteux en espace !

- Une seule table d'indirection ne passe pas à l'échelle

Exemple

- 48 bits d'adressage logique
  - 36 bits de numéro de page logique
  - 12 bits de décalage
- 8 octets par entrée de la table
  - Adr de base de la page physique
  - Métadonnées (droits de la page, etc.)
  - Bits réservés
- Taille de la table des pages ?
  - 2^36 * 8o = 549 755 813 888 = 512Go
  - 512Go par table =
  - 512Go par prc (oups!)

Pagination multi-niveaux

- Découper l'adr logique en Plsrs morceaux
- L'adr d'une table + un morceau donne un champ dans la table
- Chq champ d'une table indique
  - Soit l'adr de la table suivante à consulter
  - Soit qu'il n'y a pas de table suivante: faute CPU

Multi-niveau : pourquoi on y gagne ?

- L'esp d'adressage des prc est plein de vide
- Ne remplir que les tables intermédiaires nécessaires

→ Élagage de l'arbre des tables de pages

Dans la vraie vie

Bcp de détails tech (et historiques)

Plsrs schémas poss. et configurables

- x86-64: souvent 48bits d'adressage sur 4 niveaux (mode long 4k)
- 57bits d'adressage sur 5 niveaux chez de récents prc Intel

Taille des pages variable

- Plsrs tailles et schémas peuvent cohabiter en mm temps
- x86-64: 4ko, 2Mo, 1Go

Autres fctionnalités

- Peut se combiner avec la segmentation (x86)
  - Adr logique → adr linéaire → adr physique
- Métadonnées supp

Trop couteux en temps !

- Accéder à la mém coute trop d'accès mém

Exemple

- 4 niveaux : 4 tables + RAM finale
- Un accès mém logique = 5 accès mém physique
  - Chercher dans 4 tables + la donnée finale
  - Plus 5 additions, 4 vérifications des droits, etc.
- Les performances sont divisées par 5
- Et ceci pour Chq accès à la mém

TLB et caches prc

TLB (translation lookaside buffer)

- Cache les dernières traductions logiques → physiques
- Cas idéal fréquent : 0 accès mém pour traduire
- Cas pas idéal rare: faire toutes les indirections nécessaires

Caches CPU

- Cache le contenu de la RAM
- Évite l'accès à la RAM complètement

620 – MÉMOIRE VIRTUELLE

Aller plus loin ?

- Offrir à Chq prc une mém plus grande que celle dispo
- Utiliser le disque comme mém supp
- Un prc n'a pas forcément besoin d'être entièrement en mém principale

→ De façon transparente pour les prc

Partitions et fich d'échanges

- Alias: swap
- Fich ou partition dédiés
- Utilisée comme mém supp
- Accès lent, donc à utiliser correctement

Mémoire virtuelle sans pagination (historique)

- Alias: swapping de prc

Quand la mém est faible

- Trouver un prc pas souvent actif
- Copier toute sa mém sur disque (swap out)
- Puis libérer la mém du prc

→ La mém n'est plus faible !

Quand on doit continuer l'exec du prc

- On recharge le prc en mém (swap in)
- Quitte à swap out un autre prc pour faire de la place

Mémoire virtuelle et pagination

Paging (swapping de page)

- Une page virtuelle peut être soit en mém physique, sur le disque (en swap) ou invalide
- Si RAM est pleine: on sauve
  - On descend des pages physiques vers le disque (page out)
- Si accès à une page virtuelle qui est sur le disque: on charge
  - On monte une page physique depuis le disque (page in)

Avantages

- Granularité bcp plus fine que le swapping de prc
- Chargement et déchargement de morceaux de prc au besoin

Mémoire résidente vs. Mémoire virtuelle

Mém virtuelle

- Les pages virtuelles de l'esp mém utilisable d'un prc
- Code + données + pile + tas + bibliothèques + …

→ Ce qui apparait avec pmap(1)

Mém résidente

- Les pages d'un prc physiquement en RAM
- Transparent pour les prc, géré par le noyau
- Habituellement, une page physique est comptée une fois mm si associée à plsrs pages logiques

Question

Qu'est-ce qui peut être plus grand que la taille de la RAM ?

- La taille de la mém virtuelle d'un prc : Oui.
- La taille de la mém résidente d'un prc : Non.
- La somme des tailles de la mém résidente des prc : Oui.

Mise en oeuvre

Côté MMU : on ne change rien

- Pas besoin de changer de prc
- Tout se fait côté SE

Pagination cotée MMU (rappel)

- La table des pages (MMU) indique seulement
  - Si une page logique existe
  - Et si oui : où (quelle page physique) et avec quels droits
- Une faute CPU est lancée
  - Si le CPU accède à une page logique absente
  - Si le CPU accède à une page logique avec les mauvais droits

Côté SE

Migration d'une page sur disque

- Quand le SE migre une page
- Il marque que la page est en swap (et où)
- Ça ne rentre pas dans la table des pages
- Le sys a ses propres struct de données
- Il met à jour la table des pages pour invalider la page logique

→ Coût: copie sur disque et maj de la table des pages

Accès du prc à une page virtuelle en RAM

- MMU traduit correctement adr logique en adr physique
- Le CPU travaille normalement (rien de spécial)

→ Surcoût: 0

Accès du prc à une page virtuelle invalide

- MMU lève une faute CPU (faute de page)
- Le SE
  - Attrape l'interruption matérielle
  - Détermine que la page virtuelle est invalide
  - Envoie SIGSEV au prc
- Le prc est terminé (ou gère le signal)

→ Surcoût: une vérification en plus

Accès du prc à une page virtuelle en swap

- MMU lève une faute CPU (faute de page)

Le SE

- Attrape l'interruption matérielle
- Détermine que la page virtuelle est en fait en swap
- Lance le chargement dans une page physique
- (et éventuellement la migration d'une autre page si pas de place…)
- Passe le prc à bloqué (et appelle l'ordonnanceur)

Quand le chargement est fini, le SE

- Met à jour la table des pages
- Passe le prc à prêt (et appelle l'ordonnanceur)

Lorsqu'élu par l'ordonnanceur, le prc

- Recommence l'instruction fautive
- Qui réussit (cette fois)

Défaut de page

Défaut majeur de page

- L'adr virtuelle est valide
- Mais la page n'est pas en mém : elle est sur disque
- Il faut faire des entrées-sorties pour la récupérer
- Métrique %F de time(1)

→ le sys charge la page en mém (couteux)

Défaut mineur de page

- L'adr virtuelle est valide
- Or page physique est en mém (cache ou chance)
- Mais n'est pas associée dans la table des pages
- Métrique %R (recoverable) de time(1)

→ le sys met juste à jour la table des pages (peu couteux)

Algo de remplacement

Prob bien étudié et bien généralisable (swap, cache…)

Données

- Un grand nb de pages virtuelles
- Une séquence de demandes de pages virtuelles
- Un nb limité de pages physiques

Objectif

- Trouver à chq demande quelle page physique utiliser
- Déterminer quelle page migrer quand la mém est pleine
- Minimiser le nb de défauts de pages (et de migration)

Idées de base : quelles pages migrer ?

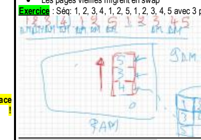
- Idéal : Les pages non utilisées dans un futur proche
- Approximation : Les pages non utilisées récemment
- Approximation pire : Les pages anciennement alouées

Algo naïf : file d'attente (FIFO)

Principe

- Les pages vieilles migrent en swap

Exercice : Séq: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 avec 3 pages physiques



Anomalie de Belady (curiosité théorique)

- Avec plus de pages physiques (plus de RAM)
- Le nb de fautes ne décroit pas forcément
- Dans certaines situations, le nb de fautes peut augmenter

Algo de l'horloge (ou de la seconde chance)

Comme FIFO, mais un bit indique s'il y a eu utilisation

- Un bit marque les pages utilisées (MMU) : 1 lorsque déjà sur la RAM
- Parcours circulaire des pages candidates, débute à ↑
- Si son bit = 1, on le passe à 0 sinon on migre la page en swap
- On tourne la flèche lorsque -1 ou migration

Exercice : Séq : 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 avec 4 pages physiques

621 – MÉMOIRE VIRTUELLE AVANCÉE

Mémoire virtuelle

Aller plus loin ?

- Allouer, initialiser, charger, copier la mém efficacement
- Offrir des services aux prc

Optimisation

- Associer pages logiques et physiques paresseusement
- Mise à zéro paresseuse de la mém
- Partager les pages à outrance
- Charger les fichiers paresseusement

→ De façon transparente pour les prc

Services aux prc

- Allocation de mém
- Projection de fichiers en mém (mmap)
- Communication par mém partagée
- Configuration de politiques (et d'heuristiques)

Zones mémoires virtuelles des prc

- Alias: région mém virtuelle, virtual memory area, ou mapping

→ Les détails au fur et à mesure

Concept du SE

- Ignoré et inconnu du processeur (et MMU)
- Existe pour des raisons de gestion (et d'implémentation)
- Permet de mieux organiser l'espace mém des prc

Regroupe des pages virtuelles

- En morceaux cohérents
- Correspondent aux lignes de pmap(1) et de /proc/PID/maps

→ Ça permet de pas forcement gérer chaque page à part

Autre mém consommée des prc

- Table des pages (celle utilisée par le MMU)
- Struct de gestion (table des prc, des descripteurs, etc.)

→ Géré à l'interne par le SE

Mémoire résidente vs. virtuelle (le retour)

- Swap + allocation paresseuse + chargement paresseux

Mém virtuelle

- Les pages virtuelles dans l'espace d'adressage
- Colonne VSZ de ps(1)
- Colonne VIRT de top(1)
- VmSize, etc. de /proc/PID/status
- Colonne Size de pmap(1)

Mém résidente

- Les pages physiques réellement en RAM
- Colonne RSS et %MEM de ps(1)
- Colonne RES et %MEM de top(1)
- VmRSS, etc. de /proc/PID/status (Linux)
- Colonne Rss de pmap(1)
- Note: rss = resident set size

Copie sur écr (COW, copy-on-write)

- Faire la copie paresseuse de pages mém
- Exemple d'utilité : rendre fork(2) très efficace

Stratégie

- Lors d'une demande de copie de page
- On copie rien, on utilise juste deux fois la mm page physique
- On ne fait une copie de la page seulement au premier accès en écr

Mise en oeuvre COW

Lors d'une copie, on met a jour la table des pages

- La nouvelle page logique pointe la page physique originale
- On enlève les droits en écr de l'ancienne page logique et de la nouvelle page logique

→ Cout: maj de la table des pages

Lors d'un accès en lect à la page logique

- Tout se passe normalement

→ Cout: 0

Lors d'un accès en écr à la page logique

- Le MMU lève une faute CPU
- Le sys attrape l'interruption, puis
- Copie la page physique dans une nouvelle page physique
- Associe la page logique à la nouvelle page physique
- Positionne les droits en écr
- Redonne la main au prc
  Qui recommence l'instruction (et réussit cette fois)
- **Pas besoin de passer à bloqué**: c'est un défaut de page mineur

→ Cout: copie d'une seule page et maj de la table des pages

Questions

- Pourquoi ne pas passer à bloqué ? La page existe déjà, voir ci-haut.
- Comment distinguer un COW d'une vraie page en lect seule ? C'est juste des « collants » qu'on va ajouter dans notre table de pages : si c'est des COW, on va juste l'indiquer.

Zone privée vs. partagée

Zone partagée (shared)

- Diff prc utilisent les mms pages partagées
- Si la zone est écrivable, les modif sont vues par tous
- Une zone partagée peut être utilisée par un seul prc

Zone privée (private)

- Diff prc utilisent des pages privées personnelles et des pages partagées communes (en lect seule)
- Quand une page privée est écrite: les modif sont vues que par le prc
- Quand une page commune est écrite : copie sur écr

Question: Qu'est-ce que ça change pour les zones en lect seule ? Rien du tout : que ce soit partagé ou pas, c'est que le petit « s » qu'on va voir.

Partage des pages sous Linux

- pmap(1) et /prog/PID/maps affiche s pour les zones partagées et - (ou p) pour les zones privées
- Colonne SHR (shared) de top(1): somme des taille des pages partagées
- Colonne Pss (proportional set size) de pmap(1): chaque page divisée par le nb d'users