SYS2 NOTES

Mathieu 'xar' Nativel

June 14, 2019

Contents

1 Coucou

Voilà mes notes de sys2, il y a probablement masse de fautes, et des passages pas très claires désolé. C'est aussi malheureusement possible que je me sois trompé des fois, si vous trouvez une erreur ou un truc suspect, hésitez pas à m'envoyer un message. Bisous et bonne révisions.

2 18/02 Introduction : Réseaux

2.1 Réseaux

Le réseau c'est chiant: N réseau, M applications nécessitent N*M implémentations ... :(Finalement même problème qu'avec la compilation. Solution : IR. (Intermediate Representation)

2.1.1 Modèle OSI

Modèle OSI: découper en plusieurs couche protocolaires les liaisons réseaux. Chaque couche 'n' n'a besoin que de la couche 'n-1'.

2.1.2 TCP/IP Protocoles de l'internet

Dans une trame TCP il y a deux champs flags & seq. Seq = numéro de séquence du paquet. Et le flag sert à la machine réceptrice pour pouvoir acknowledge la réception du paquet. Aussi flag de synchronisation.

On renvoit B + 1 pour acknowledge B.

SOCKET API

```
int fd = socket(int domain, int type, int protocol);
```

Voir man pages: socket, connect, bind, listen, accept, send, recv, sendto, recvfrom.

1. App Implem en userland.
2. Transport Exemples : **TCP DP SCTP** Permet de résoudre différents problèmes: on envoie un paquet à une machine, où la machine doit elle le forward? Serveur HTTP ou BROWSER ? **Permet aussi de mettre les messages reçus dans l'ordre.** Implémenté dans le **kernel**.

3. Réseau (Network) Protocole connu **IP** Permet l'utilisation des routeurs. Implémenté dans le **kernel**.

4. Lien (Link) Permet de transférer des trames à un destinataire. On utilise habituellement ETHERNET. La couche de link et la couche physique forment les driver et la carte réseau. (NIC)

5. Physique (Physical) En pratique non implémenté côté OS mais implémenté dans la couche réseau.

2.1.3 Trame Ethernet : couche de lien

Pour résoudre les problèmes de collision, on ne peut parler que pendant un certain temps. Ce temps est déterminé par la MTU. Si quelqu'un est déjà en train de parler on se tait pendant un tps aléatoire. A la base on avait des **HUBS** qui envoyait les trames à chaque machine, chaque machine devant ensuite les traiter pour savoir si ça les concerne ou pas. (Addr mac dest). Pas viable, du coup on utilise des **SWITCH**, qui limite les collisions grâce à une table d'apprentissage reliant les machines et leurs ports. (d'après les premiers échanges) Problème: Comment relier plusieurs réseaux, par exemple avec des MTU différentes?? On utilise donc des **ROUTEURS** permettant de relier plusieurs réseaux différents entre eux.

1. addr mac source / addr mac dest

2. type ex: IPV4

3. addr ip source / addr ip dest

4. info protocoles

5. port src / port dest

6. size / checksum

2.1.4 Routeurs

Machine 1 -> **Switch** -> **Routeur A** -> **Routeur B** -> **Switch** -> Machine 2 Pour indiquer sur quel réseau envoyé: exemple: 192.168.7.0/24 –> Le 24 me dit que les 24 premiers bits sont la descriptions de mon réseau. (Ici 192.168.7: 3 octets = 24 bits). Les routeurs gèrent la fragmentation de paquets (découpage des trames si elles sont trop grosses comparés à la MTU). Le routeur peut découper la trame et préciser qu'ils va mtn envoyer des

fragments. De sorte que la machine puisse récupérer la trame en entier en alignant les **segments**. La route utilisé pour relier deux machines peut être différente pour deux paquets différents : soucis d'ordre d'arrivée des paquets.

3 25/02 Process vs thread

Le but du jeu est d'éxécuter du code sur la machine. L'abstraction est le process: instanciation en mémoire d'un programme. Un process a plusieurs états et est identifiable par un pid. Un process lorsqu'il est schedulé s'exécute pendant **quelques ms**. Le syscall utilisé pour créer un nouvel address-space est **exec**.

3.1 Points communs & différences

L'idée du thread est d'avoir un équivalent de process mais en pouvant partager le même AS. En gros avoir plusieurs tâches à l'intérieur de ma tâche initiale. **Un process avec plusieurs fils d'éxécution.**

3.1.1 Composantes d'un thread

Registres et stack, le reste est partagé avec les autres process.

3.1.2 Création d'un thread

Allocation d'une stack. Allocation d'un state (états des registres). La construction de thread dépend totalement de l'OS mais aussi de l'architecture. Sur linux: clone. (On fait pas tellement la diff entre thread & process!) $\text{thread}_{\text{create}}(\text{fct})$

3.1.3 Implém original des threads

On change nos flow d'éxécution avec des SIGNALS. On utilise **sigalarm** pour envoyer ce signal tout les x temps. Implém en userland. Ces implém sont encore utilisés (par exemple node.js utilise ça) de plus le context switch est beaucoup + rapide que l'implém qui suit.

1. Avantages Pas de concurrence d'accès au données. Pendant une commande bloquante hors syscall, je peux attendre en faisant autre chose.
2. Désavantages Je fais un syscall dans un seul thread et tout le process est bloqué. Solution je fais des syscall 'non-bloquants', et on réimplante un scheduler au dessus du premier. (pour les threads)

3.1.4 Faire tourner des threads d'un mm process sur 2 CPU

Sur linux la solution consiste à considérer n'importe quels entités schedulable comme une tâche. (**task**) En terme d'implémentation, on a une structure task (avec accès à l'AS) il suffit que pour les threads d'avoir un ptr qui pointe sur le même AS. Ainsi au point de vue kernel on ne fait presque pas de différence entre thread & process.

3.2 Communiquer entre différents programmes

3.2.1 Pipe

Il y a différents 'cousins' de pipe: mkfifo (syscall) permet de créer un named pipe.

1. Désavantage du pipe On essaie de ne faire que de la lecture et écriture, on a des ressources qui ne sont pas partagés: La heap, les signaux ... la mémoire. Les fds ne sont pas partagés.

3.2.2 Désavantage du fork

Fork duplique l'AS, ce qui n'est pas viable si on a envie de continuer à utiliser les mêmes données ! Pour ce faire on peut utiliser les signaux, ou les pipes. (Les pipes sont bufferisés par kernel?) Si on veut décider de parler avec un autre process, il faut avoir créé le pipe avant le fork!

3.2.3 Utiliser une socket

Socket UNIX, socket local pas lié à un réseau. Ou socket INET. Point de rencontre: un fichier. Fichier (sys)socket. J'ai des permissions car c'est un fichier. Plus pratique qu'un socket INET. Grâce aux sockets UNIX, on peut donner des privilèges ou envoyer des fd. Envoyer des fd c'est cool notamment lorsque le process veut avoir accès à un fichier où il n'a pas les droits.

1. Exemples

   (a) Depuis 4-5 Xorg tourne en userland Comment avoir accès aux fichiers de settings? Xorg va se connecter à (systemd)-logind (ce qui s'occupe de nous logger), logind va voir si l'utilisateur est bien rattaché au siège (tty physique lié à cette écran). Et ensuite va lui donner l'accès aux fds de settings.

(b) Mon browser J'ai pas envie que mon browser ai le droit d'ouvrir tous les fichiers sur ma machine! Mais uniquement ceux que je lui ai demandé d'ouvrir –> **FileDialog**

3.3 Synchroniser deux parties d'AS

SHM (/dev/shm) l'idée c'est de partager de la mémoire. Le point de rencontre entre différents process est toujours le **filesystem**. Dans shm on a des fichiers qui ne sont pas stockés sur le disque et qui ont un nom figé. On peut donc échanger sur une zone mémoire. Pas top car lorsqu'on a les droits on peut écrire dans l'espace mémoire d'un autre process?

4 01/03 Scheduling

Il faut que temps du choix du programme qui tourne ne prenne pas bcp de tps relativements aux programmes.

4.1 Task

- Deadline
- Temps d'exec
- Dépendances

4.2 Batch system

Historiquement ce qui était utilisé avant: On a une queue (un panier) et quand quelqu'un veut s'exec il s'insère dans la queue. Critères de mesures:

4.2.1 Turaround time

Tps que l'utilisateur met à récup sa tâche.

4.2.2 Throughput (débit)

Le nombre de taches qu'on sort sur une durée X.

4.2.3 2 types d'algo

- FCFS: First Come First Serve
- SJF: Shortest Job First

4.3 Time Sharing

Plusieurs tâches "en mm temps". La complexité d'un algo de scheduling est en O(1)! (temps constants tjrs!) Essayer de faire un algo équitable pour que toutes les tâches puissent tourner au moins une fois et avoir autant de CPU dont il a besoin. Métrique: - Response time: temps entre le new et le first running.

- Waiting time: temps passé en ready

1. Différents algos:
   - FIFO: comme son nom l'indique first in first out.
   - Round Robin: on définti une quantité de temps "quantum" pendant lequel mon programme est autorisé à rester en running.
   - Fair: Distinguer les programmes interactifs (browser, shell, init ...) des programmes non interactifs (calcul, compilateurs ...) Garder un quantum + long pour les taches CPU bound et un quantum + court pour les taches IO bound.
2. Bottleneck possible
   - CPU (Non interactive)
   - Memory
   - IO (Interactive)
3. Deux types de taches importantes: CPU Bound ou IO BOund.

5 04/03 CPU et synchronisation

5.1 Caches: (CPU)

-TLB -L1(Instructions, Data): Pour Fetch il y a un cache L1 instruction et pour memory un cache L1 data. -L2 -L3, L4 ...

1. Schéma Chaque cache doit être proches **physiquement** avec leur core respectifs. Core > `L2` < Core

   ---

   L3

   Core > `L2` < Core

5.2 Rappels pipeline CPU

Pipeline: FETCH DECODE EXECUTE MEMORY WRITE-BACK

5.3 Problèmes avec le multi-tasking

Lorsque j'ai une tâche qui tourne elle remplit avec ses données les caches, et donc lorsqu'on passe à la tâche d'après nos caches sont useless. En théorie ils faudraient les flush et les re remplir à chaque quantum ... C'est dommage! Du coup il faudrait augmenter le quantum de tps histoire que les taches puissent tirer un maximum avantage des cache après les avoir chargés.

5.4 IO bound/ CPU bound

IO bound: Quantum + petit mais + fréquent! CPU bound: Quantum + long mais - fréquent!

5.5 Queues de scheduling

L'idée est de faire plusieurs queues avec des priorités différentes (allant de 0 à N par priorité croissante). On essaie de vider Q0 avant QN. Ready —> Run = P– Run –> Read = P- Waiting –> Read = P+

5.5.1 Windows

Lors de la création du process on donne son 'type' dans le syscall pour savoir si il est graphique console etc ou pas. Ce qui permet de donner une priorité statique vraiment cool!

5.5.2 Linux

- Ticket Scheduling: on distribue des tickets en fonction des priorités: + t'es prioritaire et + tu as des tickets et chaque fois que tu es schedulé tu utilises tous tes tickets. ça ressemble à un style de fair-round-robin, mais on ne prend pas assez en compte le waiting time d'où le problème pour les IO bound.
- CFS: Completly Fair Scheduler

6 18/03 FileSystem

6.1 Abstraction storage

6.1.1 Fichiers

Il faut être capable de partager des choses, et de les nommer!

6.1.2 Hiérarchie de noms

6.2 Block device

Sur un disque on ne peut pas arriver et écrire au nème bits. On écrit dans des BLOCKS. La lecture et écriture se fait donc sur des blocs. Blocs analogues à des 'buffers' de taille fixe. (Exemples de taille: 2048, 512, 4K)

6.3 Implémentation

Pour réduire le tps d'accés au fichier il nous faut une structure de données adéquate. Dans la plupart des cas ce qui est utilisé est le **btree**.

6.3.1 Inode sous ext

- Metadata (type permissiosn etc)
- Identifié par un nb
- Data:
  - array direct
  - array indirect
  - array double indirect
  - array triple indirect

6.3.2 Directory

Finalement c'est un fichier comme les autres. Le type dans les métadata vont changer. Data: Tableau de **dentry**: {$\text{inode}_{\text{nbr}}$, len, name[]}

6.3.3 Superblock

Méga important sinon ton fs est fucked up! Malheureusement c'est aussi souvent parmi les premiers blocs.

6.4 ISO

Il n'y a pas d'écriture, donc beaucoup moins de contraintes. Pathtable qui marche bien (on est readonly)

6.5 FAT

- Root->$\text{block}_{\text{nbr}}$
- FAT: File Access Table

6.6 SD

Il y a un firmware qui traine dans les cartes SD. DOnc (CPU + RAM + stockage) pour trouver les blocs libres défectueux etc ...

6.7 FileSystem modernes

zfs, btrf Utilise un MerkleTree (permet le versionning + snapshotting)

7 22/03 FileSystem 2

7.1 Mount

Syscall permettant de 'mount' un disque et de l'exposer dans un dossier. (Exposé un nouveau fs). **vfs** Virtual file system: t'exposes ton fs de manière virtuelle histoire de l'avoir de manère unifié.

7.2 c++

```
struct A {
   int f();
   virtual int f1();
   virtual ~A();

   int a;
   };
struct B : A {
   int b;
   virtual int f1();
   virtual ~B();
   };
```

```
du coup en mémoire:
struct B {
    vtable *vptr;
    int a;
    int b;
    };
A *a = new B();
a->f1();
((B*)a)->f1();
a->vptr[1](a);
A a;
A.f();
A::f(&a);
a.f1();
```

En vrai l'implém se fait en c (le kernel est écrit totalement en C et pas en C++).

```
struct A_operations {
int (*f)();
};
struct B {
struct A_operations B__operations;
};
```

7.3 *dev*

ls -l *dev* montre des 'fichiers' en rapport avec des devices. Deux types de devices : c ou b. (Character ~ Terminaux ou Block ~ Disques à l'origine).

7.4 IOCTL

En UNIX tout est fichier, problème nos devices aussi du coup, il faut bien avoir des opérations propres aux devices, et ne pas rajouter 3223923 millions de syscalls. La solution c'est le syscall ioctl, qui permet de faire des opérations spécifiques sur des fd. Syscall ioctl fait à la base pour les tty pour faire tous ce qui est déplacement de curseur, couleurs, baud rate etc ...

8 25/03 Multithread

8.1 SIMD

Single Instruction Multiple Data Notre processeur fournit pour certaines opération la possibilité de travailler sur plusieurs données en même temps. Exemple: Addition de vecteurs. Instructions sous x86: SSE & AVX.

8.1.1 Implem

Pour un vecteur à N coordonées. On donne à k thread, une slice de V. (N/k) Cette slice est **contigue**. Sinon, on pourrait penser à donner chaque thread **i**: les index 0 + i .. Sauf que la granularité du cache est la **cacheline** et du coup pb lors de l'écriture: Quel est la bonne valeur de la cacheline? Solution: les thread dialoguent entre eux et se synchronisent si ils écrivent sur la même cacheline. Pas viable du tout du coup. (Complexité presque la même que en single thread). C'est le problème de **false sharing**.

8.2 Problème de synchronisation

Voir le false sharing sur le paragraphe d'avant.

8.2.1 Liste chainées

Si l'on supprime un élém d'une liste alors qu'un autre thread s'apprête à le lire, celui ci risque de déréf de la mémoire qui n'est plus allouée. **Section critique** moment lorsqu'on manipule la liste: suppression ou insertion. On va protéger ou **locker** ces sections critiques.

1. Implém: On pose un marqueur dans la liste pour expliquer qu'on est en train de la manipuler. Si quelqu'un d'autre veut la manipuler en même temps il se bloque lorsqu'il voit le marqueur. On ne **lock pas de données mais toujours du code**. Il faut absolument que le test de comparaison/set du marqueur soit une instruction primitive. **Atomic primitives**: -> Test and set (TAS) -> Compare and swap (CAS) On appelle ça un **spinlock** lorsqu'il ne passe pas la main. Si il passe la main (sleep) c'est un **mutex**.

   Pseudo implem

   ```
   for (i;; ):
        if (v == 1):
           v = 0 && break
   ```

```
else:
   sleep(0)
```

8.2.2 Race condition

stat(file) pour vérifier la taille et ensuite open et lol c'est la bonne taille. Mais la taille a pu changé entre temps. Il faut donc utiliser fstat pour préserver le fd sous la main et des info correctes.

8.2.3 Lock

Quand je lock il faut que je fasse attention à toujours libérer ce lock. (analogie avec l'allocation mémoire). Sinon problème de **deadlock** on lock à l'infini et mon programme est bloqué à tout jamais. **livelock** analogie avec deux personnes qui se croisent et essaie de laisser l'autre passer sans que ça marche. Inversion de priorité nécessaire si t2 moins prioritaire que t1 empêche t1 de tourner.

8.3 Types de locks

8.3.1 Spinlock

Lock, le programme ne fait que attendre qu'il puisse accéder à la donnée.

8.3.2 Mutex

Spinlock mais le programme wait en attendant et donne la main aux autres programmes. Force le scheduling! Le mutex est sémantiquement associé à une donnée mais en vrai c'est le code que l'on protège. Dans le cas du spinlock et du mutex on veut protéger une section critique et s'assurer qu'une seule personne puisse y accéder à la fois.

8.3.3 Semaphore

J'ai un compteur, chaque fois que quelqu'un met dans la queue on incrémente, si quelqu'un consomme on décrémente. En gros c'est un mutex où on a rajouté un compteur. On met dans une queue et garanti qu'on sera délock dans l'ordre de la queue.

1. Implém: Mutex + compteur + queue En pratique le mutex est un semaphore avec une val max de 1.

8.3.4 Recursiv lock

Pire idée de la terre : Fonction récursive qui lock au début et unlock à la fin ... C'est dommage elle se deadlock toute seule. On peut faire une solution à partir de sémaphore mais c'est pas ouf ... Vaut mieux faire une fonction autour de la fonction récursive qui lock & unlock.