Systèmes et Réseaux (ASR 2) - Notes de cours
Cours 3

Anne Benoit

February 5, 2015

1 Structure des systèmes d'exploitation

2 Gestion des processus

2.1 Concept de processus

2.2 Ordonnancement des processus

2.3 Opérations sur les processus

2.4 Processus coopérants

2.4.1 Mémoire partagée

Exemple du paradigme producteur-consommateur: le processus producteur produit de l'information, consommée par le processus consommateur. Deux variantes, à buffer non borné (le producteur peut toujours produire), ou avec un taille de buffer fixée.

Buffer circulaire de taille fixée: item buffer[N]; int in (emplacement où le producteur peut placer un élément); int out (emplacement où le consommateur peut se servir).

Au départ, in=0 et out=0, buffer vide.

Producteur:
```
while (true) {
  // Produit un élément "item"
  while (((in+1) mod N) == out) {} // Pas de place, ne rien faire
  buffer[in] = item; // insérer l'élément
  in = (in + 1) mod N;
}
```

Consommateur:
```
while (true) {
  while (in == out) {} // Rien à consommer
  item = buffer[out]; // récupérer l'élément
  out = (out + 1) mod N;
  // Consommer l'élément "item"
}
```

Au plus $N - 1$ éléments dans le buffer.

2.4.2 Communication inter-processus

Autre technique de communication: échange de messages entre processus. Au moins 2 opérations disponibles: send(message) et receive(message). Etablissement d'un *lien de communication* entre deux processus, puis échange de messages possible.

Plusieurs variantes:

- Communication directe: les processus se nomment de façon explicite: send(P,msg), receive(Q,msg). Liens établis automatiquement; un lien pour chaque paire de processus communicants; liens uni-directionnels ou bi-directionnels.
- Communication indirecte: utilisation de boites aux lettres (bal), ou ports; chaque bal a un id unique. Liens établis uniquement si les processus se partagent une bal; liens associés à plusieurs processus; possibilité d'avoir plusieurs liens entre 2 mêmes processus; uni- ou bi-directionnel.
- Com. indirecte: si bal A partagée entre 3 processus, P_1 envoie et P_2, P_3 reçoivent, qui obtient le message? Dépend de la solution choisie: lien associé à au plus 2 processus; un seul processus à la fois peut faire receive; choix arbitraire du receveur.
- Passage de message bloquant (synchrone: l'envoyeur ne fait rien jusqu'à ce que le msg soit reçu, idem pour la réception) ou non-bloquant (asynchrone: envoi puis continue son activité, réceptionne soit un msg valide soit null puis continue).
- File de messages associée à un lien de communication: (i) capacité 0 = pas de buffer, l'envoyeur doit attendre le récepteur (rendez-vous); (ii) capacité bornée (buffering explicite): l'envoyeur doit attendre si le lien est plein; (iii) capacité non bornée (buffering automatique): l'envoyeur n'attend jamais.
- Message: taille fixée ou variable.

2.4.3 Communication dans les systèmes client-serveur

Un serveur et des clients qui veulent accéder ce serveur: possibilité d'utiliser la mémoire partagée et les échanges de message. Autres stratégies de communication, que l'on ne détaillera pas ici:

- Les sockets (adresse IP + numéro de port), pour accéder à un serveur web, ftp;
- Appels de procédure distants (RPC, Remote Procedure Call) et RMI de Java (Remote Method Invocation, invoquer une méthode sur un objet distant).

2.5 Conclusion

- Processus = programme en exécution; différents états possibles (new, ready, running, waiting, terminated) et PCB (représentation du processus par l'OS);
- Files d'attentes: ready (en attente du CPU) et IO; ordonnanceur long-terme (choix des processus ready) vs ordo court-terme (sélection d'un processus à exécuter);

- Un processus doit pouvoir créer un processus fils qui s'exécute en concurrence (de préférence);
- Processus coopératifs: communiquent au moyen de mémoire partagée ou d'échanges de messages.

Exercice: Que va-t-il s'afficher?

```
int value = 5;
int main() {
  pid_t pid; pid = fork();
  if (pid == 0) {value += 15; return 0;}
  else if (pid > 0) {wait(NULL); printf("PERE: value = %d\n", value); return 0;}
}
```

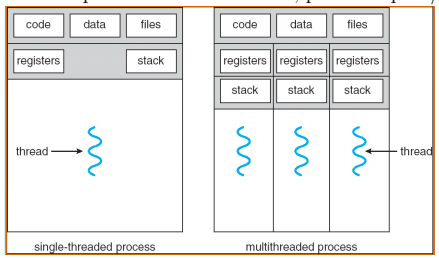
3 Les threads

3.1 Threads vs processus

On a vu dans la section précédente: processus = programme en exécution.

Processus lourd: un seul thread. Contenu de l'espace d'adressage? Texte (code du programme), données, fichiers, pile (variables locales). Processus défini également par son compteur de programme et le contenu des registres.

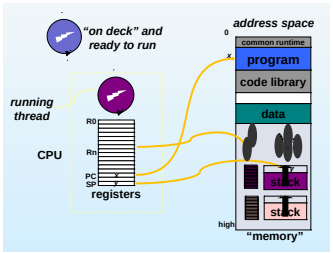
Pour un processus multi-threadé, plusieurs piles/registres, un par thread d'exécution.



Processus constitué:

- d'un ou plusieurs threads (séquence d'instructions qui s'exécutent, entité active);
- dans un même espace d'adressage, privé (données utilisées par le processus, entité passive sur laquelle les threads agissent).

Programme avec 2 threads:



Threads: partagent un même espace d'adressage, moins lourd à gérer (notamment pour effectuer un changement de contexte).

Utilité des threads? Ne pas perdre de temps à attendre une ressource lente. Exemples:

- Système de fenêtres: un thread par fenêtre.
- Serveur web ou BD: un thread par requête.
- Le noyau de l'OS: un thread attend le clavier, un autre la souris, ... Presque tout est lent, sauf le CPU!

Avantages des threads:

- Une même application peut effectuer plusieurs tâches similaires (serveur web);
- Temps de réponse: un programme peut continuer son exécution même si une partie est bloquée;
- Partage de ressources: mémoire et ressources d'un processus partagés, plusieurs threads dans le même espace d'adressage;
- Economie: Allocation mémoire/ressources coûteux. Entre 20 et 100 fois plus lent de créer un processus qu'un thread. Changement de contexte 5 fois plus rapide avec un thread;
- Permet l'utilisation d'architectures multi-processeur.

3.2 Threads utilisateurs/noyaux

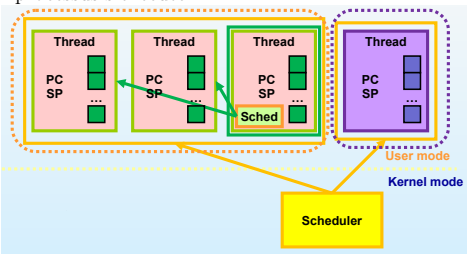
Bibliothèques de threads: API pour créer/gérer les threads.

3.2.1 Threads utilisateurs

Bibliothèques au niveau utilisateur: pas de support noyau, invocation d'une fonction $\neq$ appel système.

Pas besoin de support particulier du système (utilise n'importe quel Unix); création et changement de contexte rapides (pas d'appels systèmes); définit son propre modèle de

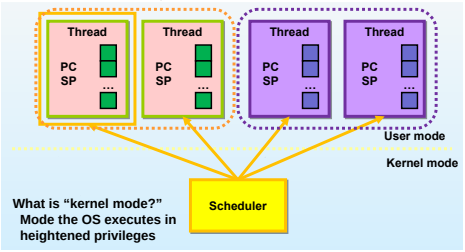
thread et ses politiques d'ordonnancement; l'OS ne voit qu'un seul processus et décide quel processus s'exécute.



3.2.2 Threads noyaux

Espace noyau: API → appel système, code et structures de données présents dans le noyau.

C'est l'OS qui définit le modèle de threads et l'ordonnancement des threads (décide quel thread s'exécute).



3.2.3 Compromis

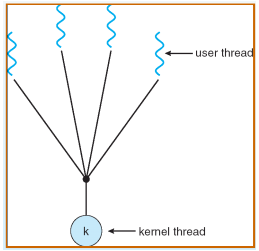
Quels threads vont marcher le mieux? Threads utilisateurs: synchronisation = simple appel de fonction, attractif si peu de contention (sinon, beaucoup de traps pour effectuer des appels systèmes). En plus, si un thread utilisateur se bloque, tout le processus est bloqué, et c'est plus difficile d'utiliser efficacement plusieurs CPUs. On veut des threads noyaux pour chaque CPU.

Comparaison thread utilisateur/ thread noyau/ processus, temps en microsondes pour un fork: (34, 948, 11300). Appel de procédure: 7 μs; Trap: 19 μs.

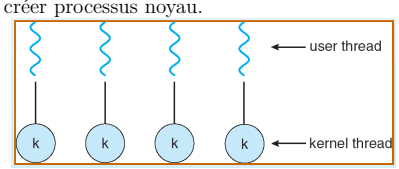
3.2.4 Modèles de multi-threading

Définit les relations entre threads utilisateurs et threads noyaux.

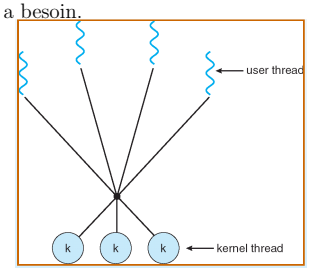
Many-to-one: Gestion des threads dans l'espace utilisateur. Processus entier bloqué si un thread fait un appel système. Un seul thread peut accéder le noyau à la fois.



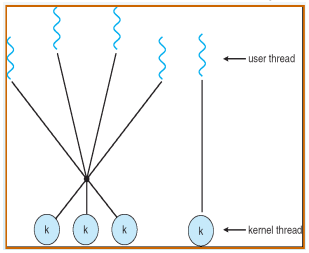
One-to-one: Chaque thread utilisateur a un thread noyau. Permet plus de concurrence, et l'utilisation de multi-processeurs. Création plus lourde: créer thread utilisateur = créer processus noyau.



Many-to-many: Plusieurs threads noyau, l'OS peut créer le nombre de threads dont il a besoin.



Modèle à deux niveaux: Comme many-to-many, mais on peut aussi attacher un thread utilisateur à un thread noyau.



3.2.5 Bibliothèques de threads

Trois bibliothèques principales:

- POSIX Pthreads: noyau/utilisateur, utilisé par les systèmes UNIX (Linux, Mac OS X, ...), permet création et synchronisation des threads;
- Win32 threads;
- Java threads (utilisateur).

3.3 Problèmes spécifiques aux threads

Appels systèmes fork et exec. Est-ce que fork() duplique uniquement le thread qui l'appelle, ou tous les threads? Deux versions de fork! Par contre, exec() remplace toujours tout le processus (tous les threads).

Annulation de threads. On peut terminer l'exécution d'un thread avant qu'il ait terminé son exécution. Deux approches:

- Annulation asynchrone, qui termine le thread immédiatement;
- Annulation différée: le thread regarde périodiquement s'il doit être annulé. Permet une meilleure gestion des ressources (il faut prendre garder aux ressources allouées au thread).

Gestion des signaux. Systèmes UNIX: signaux pour notifier un processus d'un événement. Gestionnaire de signaux: traite les signaux. Signaux synchrones (accès mémoire illégaux, division par 0), délivrés au processus qui a effectué l'opération. Signaux asynchrones, générés par événement externe (Control-C).

Plusieurs options: délivrer le signal au thread concerné, à tous les threads du processus, à certains threads du processus, ou bien à un thread qui gère tous les signaux. Synchrones: au thread. Asynchrones: à tous les threads?

Pools de threads. Créer un ensemble de threads qui attendent du travail, comme par exemple pour un serveur Web. Avantages: plus rapide d'utiliser un thread existant que d'en créer un nouveau; limite sur le nombre de threads de l'application.

Données spécifiques aux threads. Les données du processus sont partagées par tous les threads. Données spécifiques: chaque thread peut avoir sa propre copie de certaines données (supporté par la plupart des bibliothèques de threads).

Activations de l'ordonnanceur. Communication entre le noyau et la bibliothèque de threads: modèles many-to-many et deux-niveaux doivent maintenir le bon nombre de threads noyaux. Upcalls: mécanisme de communication.

Coopération entre threads. Si on suppose que chaque thread a son propre CPU (et avance à son propre rythme), 2 threads qui ont les sorties ABC et 123 respectivement. Que peut-on avoir en sortie? Ordre quelconque 123ABC, 1A2B3C, ABC123, mais ordre local respecté. Si dépendances entre événements, résultats différents suivant l'ordre!

3.4 Implémentation des threads

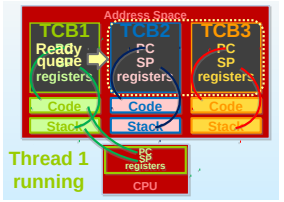
Les threads peuvent accéder des données partagées (cours synchronisation), mais aussi ils se partagent le matériel (CPU et mémoire).

- Etat privé à chaque thread: PC, registres, pile, SP (stack pointer).
- Etat partagé: code, tas, variables globales.

Threads qui ne sont pas en cours d'exécution? Exécution en pause, prêt à être exécuté. Comme pour les processus, il faut sauver l'état privé d'un thread suspendu. On laisse la pile en mémoire où elle se trouve, et on sauve les registres en mémoire. Il faut recharger les registres pour reprendre l'exécution.

Thread Control Block (TCB): donnée maintenue par l'OS pour décrire chaque thread, contient les données privées d'un thread suspendu (similaire au PCB des processus).

Les ordonnanceurs référencent ces TCBs (file ready...).



Etats d'un thread. 3 états possibles pour un thread.
Yield: le thread renonce au CPU.