Systèmes et Réseaux (ASR 2) - Notes de cours

Anne Benoit

May 13, 2015

Contents

Fonctionnement du cours

Planning du cours: `http://graal.ens-lyon.fr/~abenoit/asr15/`

2h cours / 2h TD ou TP par semaine; cours le mercredi à 10h15 sauf contre-ordre.

Première semaine: pas de TD, et deux cours la deuxième semaine. Cours mercredi 28 janvier à 10h15, mercredi 4 février à 10h15, et jeudi 5 février à 10h15, puis tous les mercredis.

Evaluation

Contrôle continu: 1 partiel et 1 (ou plusieurs) DM, peut être des devoirs sur table "surprise". Examen à la fin du semestre, note finale: (CC+2NE)/3. Questions de cours/TD.

Résumé du cours

Suite du cours ASR1, on se concentre sur le S et le R (architecture traitée en ASR1).

Pré-requis: notions d'architecture, maîtrise du C.

Plan du cours:

Système:

- Introduction sur les composants d'un OS, notion de processus et de threads
- Synchronisation des processus et deadlocks
- Ordonnancement des processus
- Mémoire et mémoire virtuelle

Réseaux:

- Structure d'un réseau, les différentes couches
- TCP/IP, MAC
- Algorithmes de routage
- Contrôle de congestion

Objectifs: connaître les fondamentaux. On ne détaillera pas le fonctionnement de tous les systèmes et de tous les protocoles réseaux. Présentation des idées clés qui sont à la base de tout système et protocole réseau. Avant de regarder comment ça marche, on étudiera "pourquoi" cela fut inventé, quel problème cela résout, et puis cela fait "quoi"?

Bibliographie

Silberschatz, Galvin et Gagne, "Operating System Concepts" pour la partie systèmes. Les curieux peuvent consulter et expérimenter avec SOS, Simple Operating System, `http://sos.enix.org/`.

James Kurose et Keith Ross, "Analyse structurée des réseaux" pour la partie réseaux.

PARTIE 1: Systèmes

1 Structure des systèmes d'exploitation

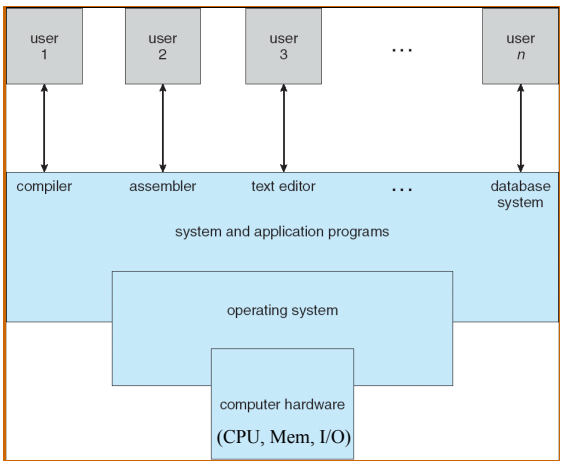
Objectifs: présenter les différents composants d'un OS et les bases de l'organisation d'un système informatique; décrire les services fournis par l'OS aux utilisateurs, processus et autres systèmes; discuter des différentes façons de structurer un OS.
Aperçu:

- OS: intermédiaire entre utilisateur et matériel (hardware).
- But: environnement dans lequel l'utilisateur peut exécuter ses programmes facilement et efficacement.
- OS: logiciel qui gère le matériel, qui doit fournir des mécanismes appropriés. Il cherche à optimiser l'utilisation du matériel.
- Différences entre OS, suivant buts recherchés: facile d'utilisation, performant, combinaison? Buts doivent être identifiés avant de se lancer dans la conception d'un OS.
- L'OS doit être conçu morceau à morceau.

1.1 Organisation et architecture d'un système informatique

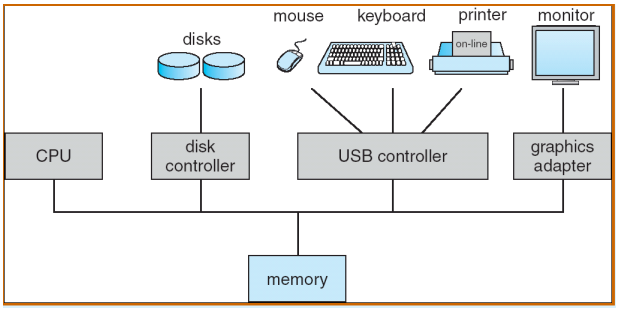
Système informatique: 4 principaux composants:

1. Matériel: CPU, mémoire, périphériques d'entrée-sortie (I/O);
2. OS: contrôle le matériel;
3. Programmes des applications: utilisation des ressources système pour résoudre des problèmes définis par les utilisateurs;
4. Utilisateurs: les personnes.



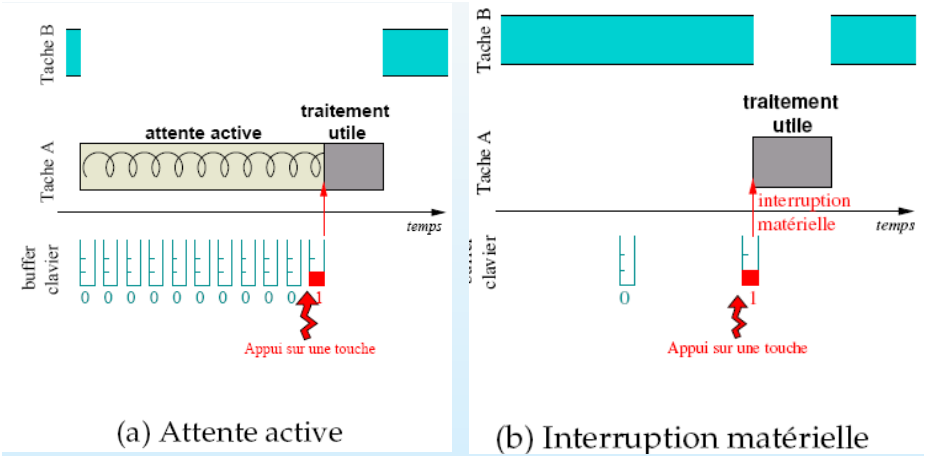
Organisation.

- Un ou plusieurs CPUs, et contrôleurs de périphériques qui sont connectés via un unique bus qui donne accès à une mémoire partagée;
- Execution concurrente, compétition pour cycles mémoires;
- Besoin de synchroniser les accès à la mémoire (contrôleur de mémoire);
- Périphériques I/O et CPU peuvent s'exécuter en parallèle;
- CPU: déplace données de/vers la mémoire vers/de des buffers locaux (registres); opérations load/store;
- I/O entre le périphérique et le contrôleur (ou buffer local); informe le CPU quand il a terminé via une interruption.



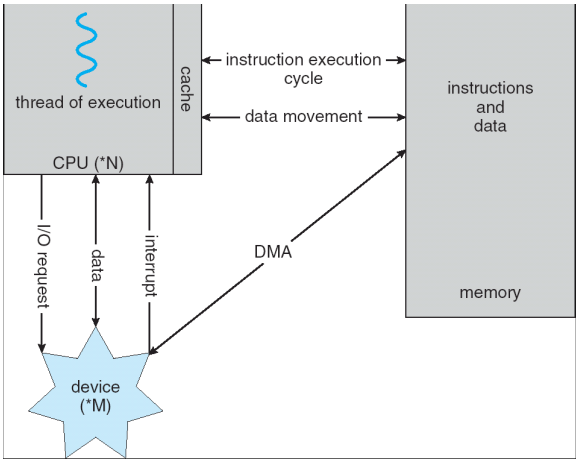
Gestion des interruptions.

- OS basés sur les interruptions, notamment pour la gestion des périphériques;
- Autre type d'interruption: un *trap* est une interruption générée par un logiciel (erreur ou requête utilisateur);
- Interruption: récupérer l'adresse de la routine d'interruption (vecteur d'interruptions, ou sondage pour trouver le périphérique concerné); sauver les registres et le compteur de programme pour pouvoir reprendre l'exécution après l'interruption;
- Interruptions désactivées lorsqu'on traite une interruption (éviter de perdre une interruption);
- Attente active ou interruption matérielle:



Structure des I/O.

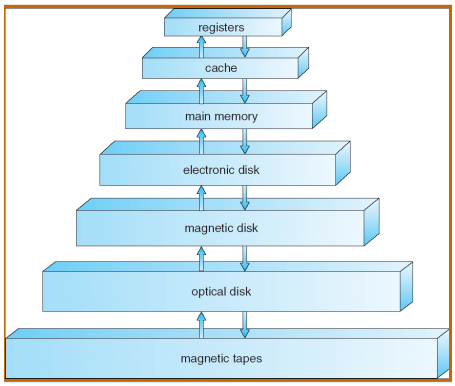
- I/O basées sur les interruptions;
- Un driver de périphérique qui peut parler avec le contrôleur, et qui présente une interface uniforme au reste de l'OS;
- DMA: accès direct mémoire pour les I/O rapides, permet de transmettre les données beaucoup plus rapidement, pas d'intervention du CPU.



Structure du stockage.

- Mémoire: seul média que le CPU peut accéder directement, mais trop petit pour contenir tous les programmes et les données, et volatile;
- Stockage secondaire: extension de la mémoire, non volatile;
- Hiérarchie des périphériques de stockage: plus c'est rapide, plus c'est cher et plus c'est petit!
- L'OS contrôle les mouvements entre mémoire et disque. En dessous, c'est matériel;

- Notion de cache: copier de l'information dans un stockage plus rapide temporairement; politiques de taille de cache et de remplacement des données.



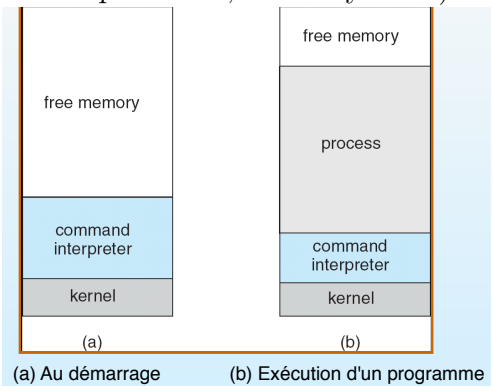
Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	> 16 MB	> 16 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 – 0.5	0.5 – 25	80 – 250	5,000.000
Bandwidth (MB/sec)	20,000 – 100,000	5000 – 10,000	1000 – 5000	20 – 150
Managed by	compiler	hardware	operating system	operating system
Backed by	cache	main memory	disk	CD or tape

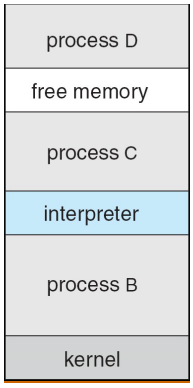
Concepts importants.

- **Multi-programmation:** pour l'efficacité, avoir plusieurs utilisateurs qui se partagent le CPU et les périphériques d'IOs (un utilisateur ne peut pas tout utiliser tout le temps); ordonnancement des tâches: si tâche en cours d'exécution bloquée sur IO, on en choisit une autre.

- **Partage du temps:** extension logique: le CPU passe d'une tâche à l'autre très rapidement, pour avoir une grande interaction; temps de réponse < 1 seconde. Notion de processus, ordonnancement des processus, swapping si tous les processus ne tiennent pas en mémoire, et utilisation de mémoire virtuelle (cf cours mémoire).

Illustration de l'exécution avec MS-DOS (une seule tâche à la fois, le processus écrase une partie du CI, qui sera rechargé sur le disque après exécution du processus), et FreeBSD (plusieurs processus, Berkeley Unix):





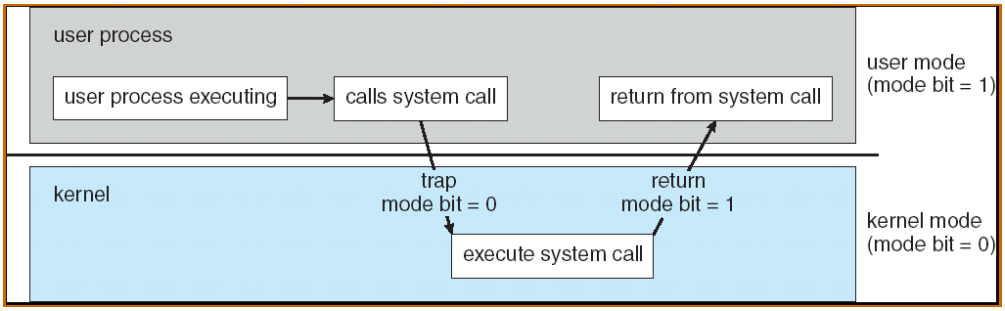
Modes utilisateurs et noyaux.

- Opérations: basées sur les interruptions déjà évoquées;

- Deux modes d'opération pour protéger l'OS: mode utilisateur et mode noyau (kernel);

- Le matériel fournit un bit de mode qui indique si le système tourne en mode U ou K; certaines instructions ne peuvent s'exécuter qu'en mode K (privilégiées);

- Notion d'appels systèmes, qui permet de basculer en mode K;



1.2 Services fournis par l'OS

Gestion des processus. Processus= entité active ($\neq$ programme). Besoin de ressources (CPU, mémoire, IO, fichiers). Un ou plusieurs fils d'exécution (threads), chaque thread a son propre compteur de programme. Multi-programmation: plusieurs processus concurrents.

Responsabilités de l'OS: créer/supprimer processus utilisateur et système; suspendre/reprendre processus; fournir des mécanismes de synchronisation et de communication entre processus, et pour gérer les interblocages.

Cours sur les processus, les threads, la synchronisation, les interblocages, et l'ordonnancement des processus.

Gestion de la mémoire. Il faut que les données/instructions soient en mémoire pour pouvoir exécuter une instruction. Gestion de la mémoire: décider ce qui est en mémoire pour optimiser l'utilisation du CPU et le temps de réponse des utilisateurs.

Responsabilités de l'OS: savoir qui utilise quelles parties de la mémoire; décider quelles données déplacer dans/hors de la mémoire.

Cours mémoire et mémoire virtuelle.

Gestion du stockage et des IOs. Notion de fichier qui donne une vue uniforme et logique de l'information stockée. Fichiers organisés habituellement en répertoires, contrôle d'accès sur les fichiers, et l'OS fournit la possibilité de créer/supprimer les fichiers et répertoires, des primitives pour les manipuler, des techniques de sauvegarde sur stockage stable.

Gestion également des disques secondaires et des opérations associées, tels la gestion des espaces libres, ...

L'OS cherche à cacher les spécificités matérielles de l'utilisateur; sous-système spécifique aux entrées-sorties qui s'occupe de la gestion mémoire des IOs, des interfaces et des drivers.

On n'en discutera pas en détail dans le cours (pas de concepts rigolos).

Services pour l'utilisateur.

- Interface utilisateur $\rightarrow$ CLI: ligne de commande uniquement (par exemple, rm file.txt), plusieurs "shells"; GUI: interface graphique (icônes qui représentent les

fichiers..., utilisation de la souris); Batch: commandes dans un fichier; Systèmes modernes: à la fois CLI et GUI

- Execution de programmes: charger un programme en mémoire, l'exécuter, terminer l'exécution;
- Opérations IO: l'utilisateur ne peut pas directement contrôler un périphérique d'IO, c'est le système qui fournit les IOs;
- Système de fichiers;
- Communications: mémoire partagée ou échange de messages;
- Detection d'erreurs: erreurs matérielles CPU, mémoire, I/O, ou dans le programme utilisateur → s'assurer d'une exécution correcte et consistante; outils de débogage...

Services pour l'efficacité: partage de ressources (plusieurs jobs/utilisateurs).

- Allocation des ressources (cycles CPU, mémoire, fichiers, périphériques IO);
- Comptes (savoir qui utilise quoi en quelle quantité);
- Protection et sécurité (les processus concurrents ne doivent pas interférer entre eux).
  - Protection: mécanisme de contrôle d'accès aux ressources défini par l'OS.
  - Sécurité: permet la défense contre des attaques (internes ou externes, tels des virus, des vols d'identité, ...)
  - Qui peut faire quoi? Notions d'ID d'utilisateur et de groupe associés aux processus et aux fichiers.

1.3 Appels systèmes et programmes systèmes

Interface aux services fournis par l'OS, écrits en langage de haut niveau (C, C++);

API: Application Program Interface. Utilisée par les programmes pour faire des appels systèmes.

Les classiques sont Win32 API (Windows), POSIX API (Unix, Linux, Mac OS X), Java API (Machine virtuelle Java JVM).

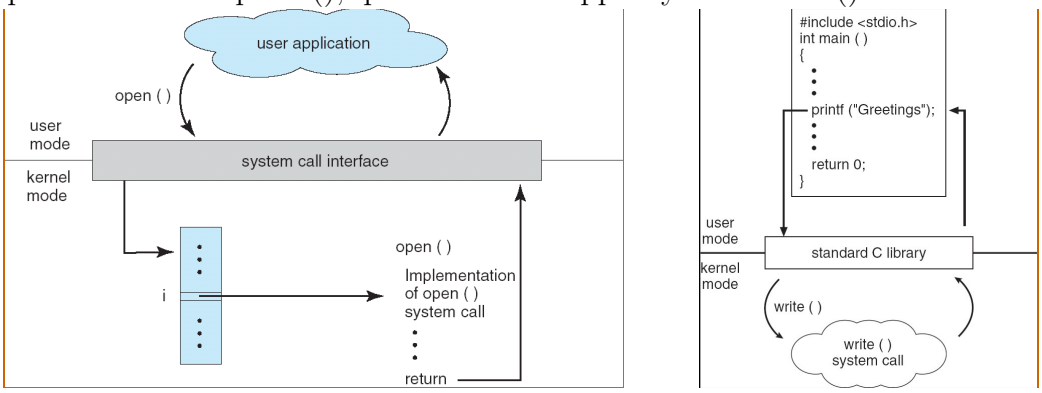
API: portabilité, i.e., un programme qui utilise une API peut tourner sur n'importe quel système qui supporte cette API.

Exemple d'API: lors de la copie d'un fichier, séquence d'appels systèmes dont par exemple l'appel à ReadFile() (Win32 API): description des paramètres à passer, de la valeur de retour.

Implémentation des appels systèmes. Un numéro associé à chaque appel système. Interface d'appels systèmes qui maintient une table indexée par ces numéros.

L'appelant n'a pas besoin de connaître les détails de l'implémentation, mais juste obéir à l'API et comprendre ce que l'OS va faire. TPs: implémentation d'un appel système.

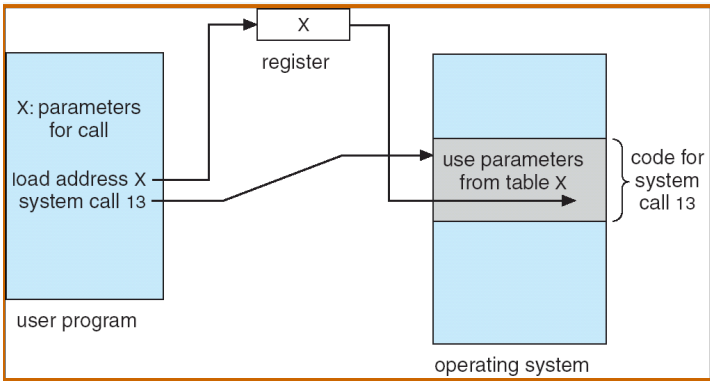
Exemple d'appel système via l'interface ou via une librairie C: le programme C invoque l'appel à la librairie printf(), qui déclenche l'appel système write().



Passage des paramètres: informations autre que le numéro de l'appel système invoqué, comme avec ReadFile. Trois méthodes pour passer les paramètres à l'OS:

1. Les mettre directement dans les registres du CPU;
2. Les stocker dans une table en mémoire, et l'adresse du bloc mémoire est passé en paramètre dans un registre (approche de Linux et Solaris, exemple ci-dessous);
3. Les placer directement sur la pile du programme.

Deux dernières méthodes: pas de limite sur la taille et le nombre des paramètres.



Programmes système. La plupart des utilisateurs voient l'OS comme défini par les programmes systèmes, et non les appels systèmes. Programmes systèmes: environnement commode pour le développement de programmes et leur exécution. Va d'une simple interface d'un appel système à des programmes plus complexes.

Exemple de la gestion de fichiers: programme système pour la copie, qui fait appel à plusieurs appels systèmes (OpenFile, ReadFile, WriteFile, ...), mais aussi information de débogage, éditeurs de texte, commandes pour chercher dans un fichier, compilateurs, assembleurs, charger et exécuter des programmes, communiquer entre processus, envoi de messages, navigateur web, messagerie électronique, ...

1.4 Conception, implémentation et structure d'un OS

La structure interne d'un OS peut varier grandement; nécessaire de bien définir d'abord les buts de l'OS et les spécifications, ce qui va dépendre en particulier du matériel utilisé.

- Buts utilisateurs: OS facile à utiliser, fiable, et rapide;
- Buts système: OS facile à concevoir, à implémenter, à maintenir, mais aussi flexible, sans erreurs, et efficace.

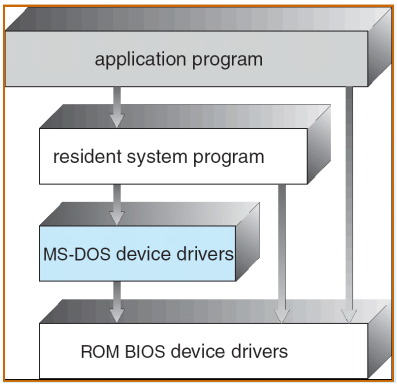
Attention à bien séparer les politiques (qu'est ce qu'on veut faire?) des mécanismes (comment c'est fait?), ce qui permet une plus grande flexibilité.

Exemple d'un timer: c'est un mécanisme pour protéger le CPU, par exemple empêcher une boucle infinie: interruptions à intervalles réguliers (compteur décrémenté par l'OS, interruption à 0). La politique peut être modifiée: décider combien de temps donner à un utilisateur avant de l'interrompre.

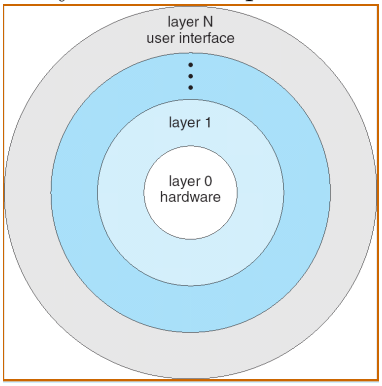
Implémentation. Traditionnellement en langage assembleur, maintenant en langage de haut niveau C, C++: plus rapide, plus compact, plus facile à déboger, et plus portable!

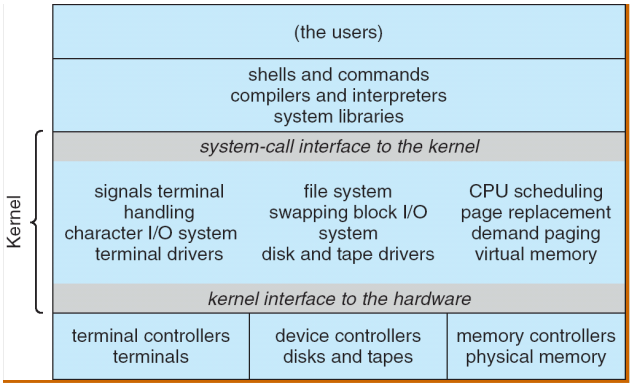
Structure des OS. Résumé des structures classiques utilisées.

1. Structure simple: exemple de MS-DOS, avec le plus de fonctionnalités possible dans le moins d'espace possible. Pas vraiment de séparation entre interface et niveaux de fonctionnalité. Pas de mode utilisateur/noyau, pas de protection.



2. Structure en couches: chaque couche s'appuie sur celle du dessous. Exemple d'UNIX qui a une structure limitée, avec 2 parties distinctes: les programmes systèmes et le noyau. Beaucoup de fonctions à un même niveau.

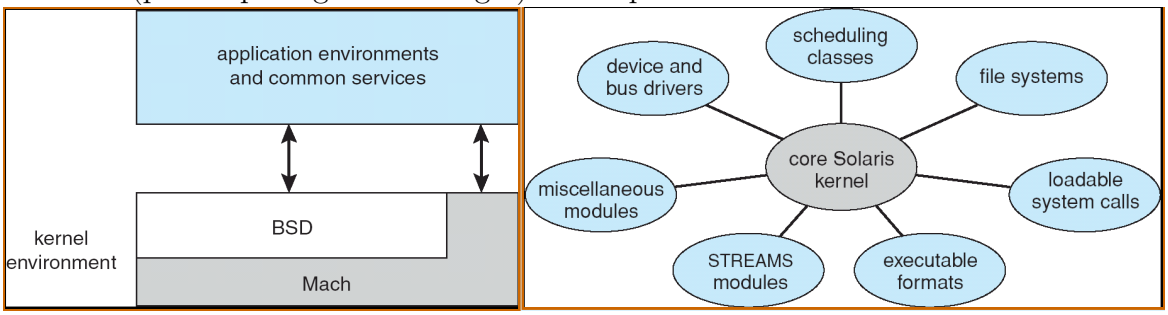




3. Structure avec un micro-noyau: avoir le plus de fonctionnalités possible dans l'espace utilisateur plutôt que dans le noyau. Le noyau gère le minimum sur la gestion des processus et de la mémoire, et permet les communications (échange de message). Avantages: facile à étendre, facile à porter, plus fiable (peu de code tourne en mode noyau), plus sécurisé. Par contre, perte de performance car communications entre espace utilisateur et noyau.

Exemple de Mac OS X: approche hybride (couches et micro-noyau) avec des extensions possibles (modules). Mach microkernel (Carnegie Mellon University) pour gestion mémoire, communication interprocessus. BSD Kernel (Berkeley) pour les pthreads, Posix API.

4. Structure avec des modules noyau: indépendance des composants, qui communiquent via des interfaces connues. Proche du modèle en couches, mais plus flexible et plus efficace (pas de passage de messages). Exemple de Solaris.

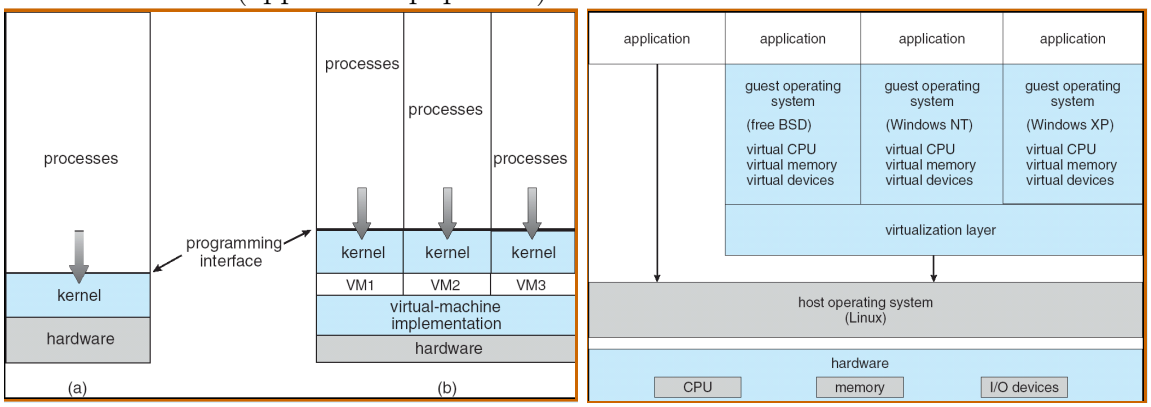


Génération de l'OS et boot. OS: peut tourner sur toute une classe de machines différentes; doit être configuré pour la machine spécifique sur laquelle il est installé. Programme SYSGEN qui retourne l'information matérielle nécessaire.

Booting: démarrer l'ordinateur en chargeant le noyau. *Programme bootstrap:* code stocké dans la ROM qui peut localiser le noyau, le charger en mémoire et l'exécuter.

Machines virtuelles. Une machine virtuelle fournit une interface identique au matériel sous-jacent: matériel et noyau de l'OS considérés comme du matériel. L'OS crée l'illusion de multiples processus s'exécutant chacun sur son propre processeur (avec sa propre mémoire (virtuelle). Les ressources de la machine physique sont partagées pour créer les machines virtuelles.

Exemple (a) sans mémoire virtuelle, et (b) avec mémoire virtuelle. A droite, architecture VMware (application populaire).



Protection totale des ressources du système: les VM sont isolées les unes des autres. Par contre, pas de partage direct des ressources! VM: parfait pour la recherche et le développement des OS, pour ne pas perturber les opérations systèmes de base. Concept de VM difficile à implémenter: doit fournir un dupliqua exact de la machine physique.

1.5 Conclusion

- OS: gère le matériel, fournit un environnement pour que les programmes utilisateurs puissent s'exécuter, fournit un ensemble de services (appels systèmes);
- Programmes: chargés dans mémoire principale (volatile), et autres moyens de stockage; IO pour accéder aux périphériques de stockage;
- Multiprogrammation (plusieurs jobs en mémoire simultanément) et timesharing (partage du temps: le CPU passe très rapidement d'un job à un autre pour être réactif);
- Modes utilisateur/noyau: instructions privilégiées s'exécutent en mode noyau;
- Bien séparer les politiques des détails d'implémentation (mécanismes): les mécanismes doivent être en place et la politique peut être choisie ensuite librement;
- Langage haut-niveau pour faciliter l'implémentation, la maintenance, et la portabilité.

Petites questions d'application du cours.

1. Plusieurs utilisateurs se partagent le système. Problèmes de sécurité? Peut-on garantir la même sécurité que sur une machine dédiée?
2. Que sont les interrupts? Différence entre trap et interrupt? Les traps peuvent-ils être générés intentionnellement par un programme utilisateur?
3. Comment faire pour obtenir un profil statistique du temps passé dans chaque portion du code d'un programme qui s'exécute? Pourquoi de telles statistiques sont importantes?
4. Discuter les avantages et désavantages de l'approche micro-noyau (microkernel).

Et ensuite? On détaillera la notion de processus puis celle de threads.

2 Gestion des processus

Objectifs: introduire la notion de processus = un programme en exécution, et décrire leurs caractéristiques: ordonnancement, création, terminaison, communication.

2.1 Concept de processus

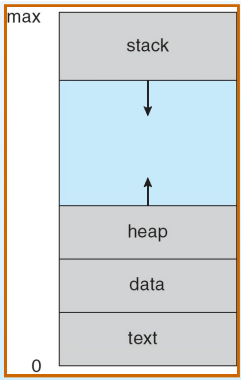
OS: execute des programmes. Programme = passif, c'est un bout de code. Un processus est un programme en exécution; on peut avoir plusieurs processus pour un même programme (par exemple un navigateur Web).

Process = task = job.

Un processus progresse séquentiellement.

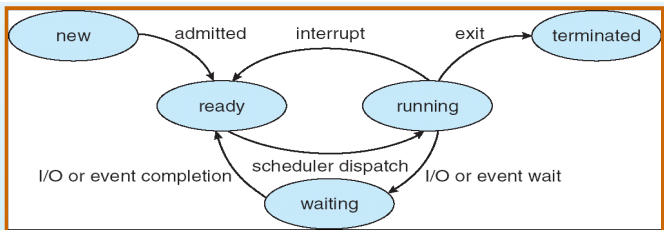
Processus = compteur de programme (PC), pile (stack), section du texte (code du programme), section des données, tas (heap, mémoire allouée dynamiquement).

Structure d'un processus dans la mémoire .



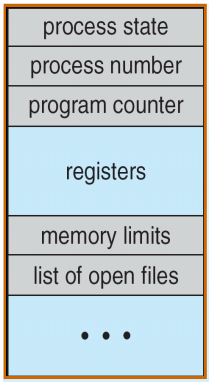
Etat d'un processus. Lors de son exécution, un processus change d'états:

- new: processus créé;
- running: instructions exécutées;
- waiting: en attente d'un événement;
- ready: en attente d'être assigné à un processeur;
- terminated: exécution terminée.



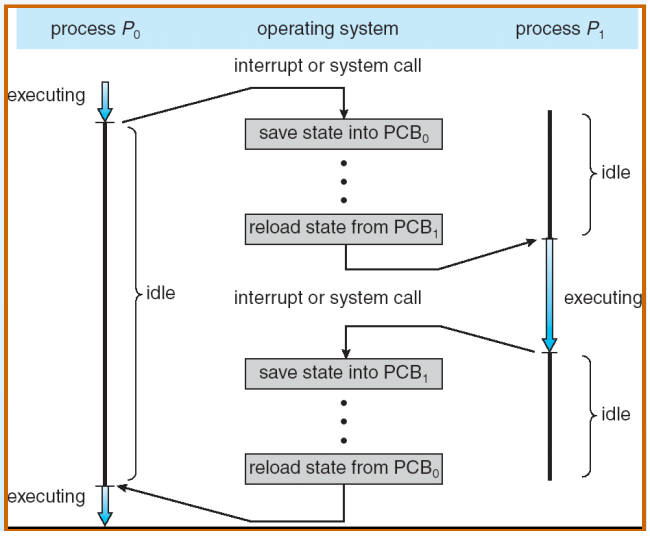
Process Control Block, PCB. A chaque processus, on associe un dépôt avec toute l'information qui le concerne, le PCB:

- état du processus (new, ready, running, ...);
- PC, registres du CPU;
- information pour l'ordonnancement (voir chapitre ordo);
- information pour la gestion de la mémoire (voir chapitre mémoire);
- informations de compte (temps CPU...);
- statut des IOs: fichiers ouverts...

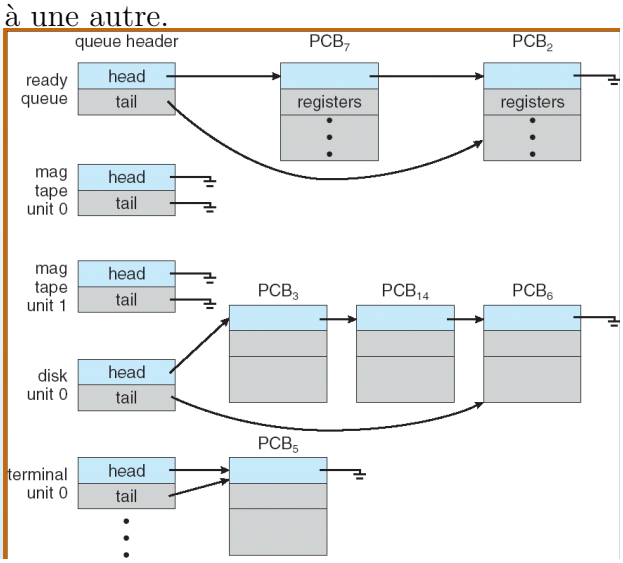


2.2 Ordonnancement des processus

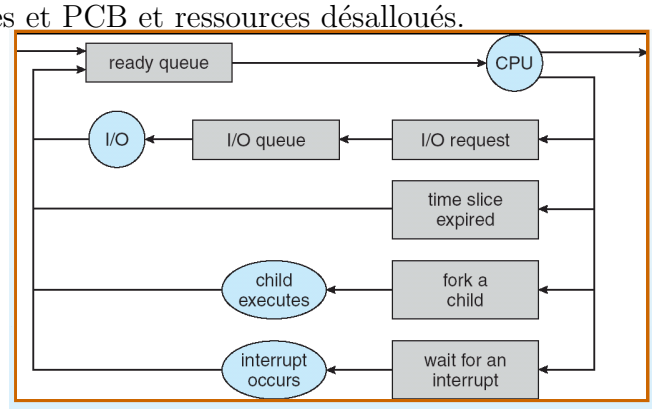
Rappel: multi-programmation pour maximiser l'utilisation du CPU, et partage du temps pour être interactif → le CPU passe d'un processus à un autre, et l'ordonnanceur doit choisir un processus prêt. Le système doit sauvegarder l'état du processus interrompu (son PCB), et charger le PCB du nouveau processus. Pas de travail utile lors d'un changement de contexte, c'est du temps perdu, doit être très rapide. Dépend du matériel: si le CPU a plusieurs ensembles de registres (Sun UltraSPARC), on n'a qu'à changer un pointeur vers l'ensemble utilisé... Il faut aussi faire attention à la mémoire...



Files d'ordonnancement. Une file avec tous les processus du système, la file des processus dans l'état ready, et des files pour les périphériques. Un processus migre d'une file à une autre.



Un processus en cours d'exécution peut être interrompu par une requête d'IO (mise dans la file d'IO correspondante, et ready quand IO terminée), une expiration du timer (directement ready), une création d'un fils (ready quand exécution du fils terminée), une attente d'interruption (ready quand l'interruption a eu lieu). Terminaison: disparaît des files et PCB et ressources désalloués.

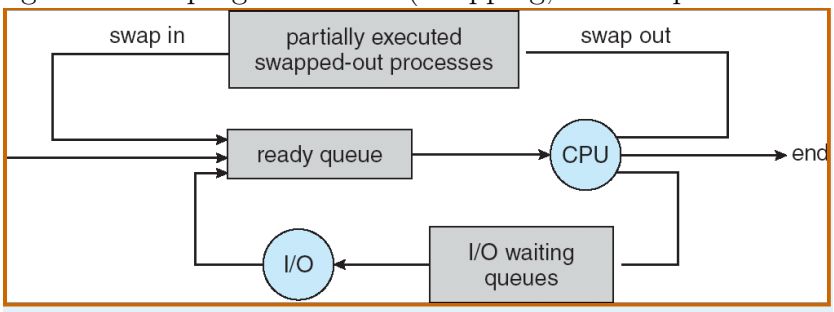


Ordonnanceurs. Ce sont les ordonnanceurs qui décident comment les processus migrent d'une file à une autre.

- Ordonnanceur court-terme (ordonnanceur du CPU): sélectionne le processus qui doit être exécuté, invoqué très fréquemment, doit être rapide!
- Ordonnanceur long-terme (ordonnanceur des tâches): sélectionne les processus à mettre dans la file ready; contrôle le degré de multiprogrammation: nombre de processus en mémoire; nombre constant: invoqué quand un processus termine uniquement, pas besoin d'être rapide.

Classification des processus: gourmands en IO (multiples utilisations du CPU courtes), ou en temps CPU (quelques longues utilisations du CPU). L'ordonnanceur long-terme doit choisir un bon mélange des deux catégories.

Ajout d'un ordonnanceur moyen-terme: l'ordo long-terme est parfois absent ou minimaliste, et il peut être avantageux de supprimer des processus de la mémoire → réduire le degré de multiprogrammation (swapping, voir chapitre mémoire).



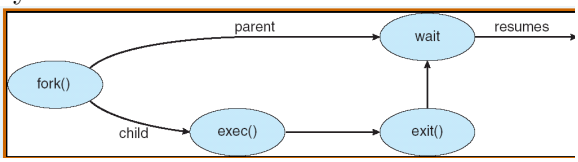
2.3 Opérations sur les processus

Les processus peuvent être concurrents, et peuvent être créés/supprimés dynamiquement. L'OS doit fournir un moyen de créer/ supprimer des processus.

Création. Appel système create-process, un processus parent crée un processus fils, on obtient un arbre des processus. Chaque processus a un unique pid (process id). Plusieurs politiques possibles:

- Partage de ressources (temps CPU, mémoire, fichiers, IO): père et fils peuvent tout se partager, partage d'un sous-ensemble des ressources du père, pas de partage (le fils obtient ses ressources de l'OS);
- Exécution: concurrente, ou bien le père attend que le fils ait terminé son exécution;
- Espace d'adressage: le fils peut être un dupliqua du père (même programme et données), ou être un nouveau programme.

Exemple d'UNIX: l'appel système **fork** crée un nouveau processus; **exec** charge un nouveau programme dans l'espace mémoire du processus. Processus fils: diffère du père par son pid (et ppid, pid du père), et statistiques d'utilisation remises à zéro. Linux: copie à l'écriture: duplication d'une page mémoire lorsqu'un processus en modifie une instance (pour avoir un fork très rapide). Appel système **wait** pour être enlevé de la file ready.



Terminaison. Appel système **exit** pour demander à l'OS de supprimer le processus; ressources désallouées (mémoire, fichiers ouverts, buffers d'IO). Un père peut terminer l'exécution d'un de ses fils avec **abort**, par exemple si le fils utilise trop de ressources, ou sa tâche n'est plus nécessaire. Si le père quitte, ses fils peuvent être tous terminés en cascade (exemple de l'OS VMS), ou bien alors le ppid des fils devient le processus initial init (exemple de Linux).

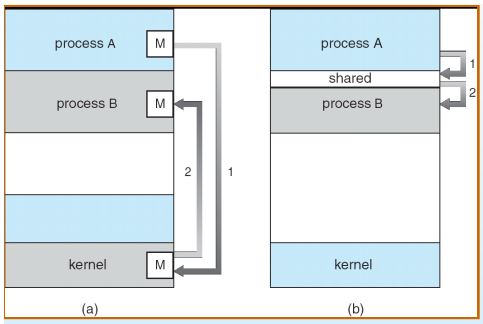
2.4 Processus coopérants

Le contraire des processus indépendants: ils peuvent affecter/ être affecté par l'exécution d'un autre processus.

Avantages:

- Permet l'accès concurrent à un fichier partagé;
- Accélérer les calculs si plusieurs CPU;
- Permet la construction d'un système modulaire;
- Commode même pour un seul utilisateur (éditer, compiler, imprimer).

Deux principaux modèles de communication:



(a) envoi de messages (b) mémoire partagée

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", value); return 0;}
}
```

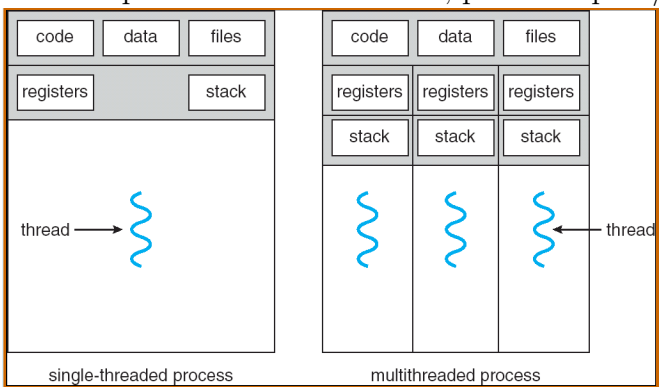
3 Les threads

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.

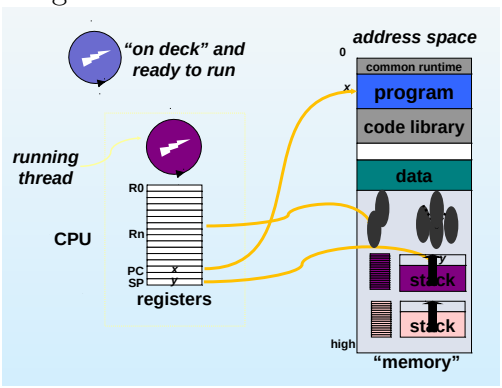
Pour un processus multi-threadé, plusieurs piles/registres, un par thread d'exécution.



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.

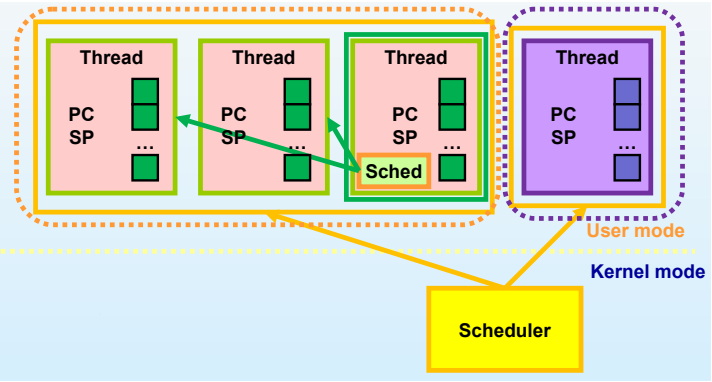
3.2 Threads utilisateurs/noyaux

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.

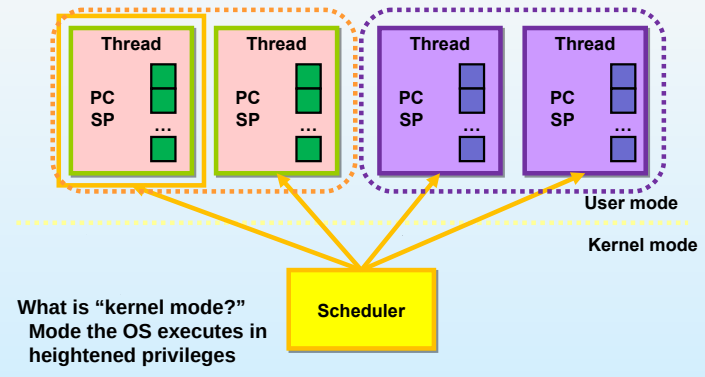
Pas besoin de support particulier du système (utilise n'importe quel Unix); création et changement de contexte rapides (pas d'appels systèmes); définit son propre modèle de thread et ses politiques d'ordonnancement; l'OS ne voit qu'un seul processus et décide quel processus s'exécute.



3.2.2 Threads noyaux

Espace noyau: API $\rightarrow$ 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

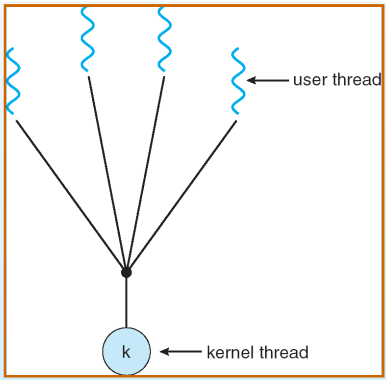
Quels threads vont marcher le mieux? Threads utilisateurs: synchronisation = simple appel de fonction, attractif si peu de contention (sinon, beaucoup de traps pour effectuer des appels systèmes). En plus, si un thread utilisateur se bloque, tout le processus est bloqué, et c'est plus difficile d'utiliser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

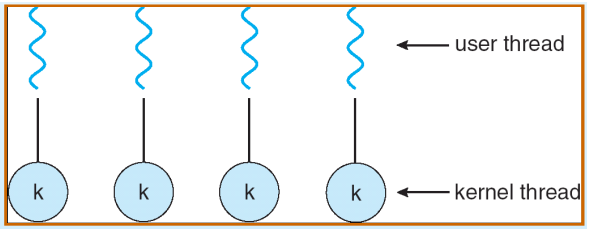
3.2.4 Modèles de multi-threading

Définit les relations entre threads utilisateurs et threads noyaux.

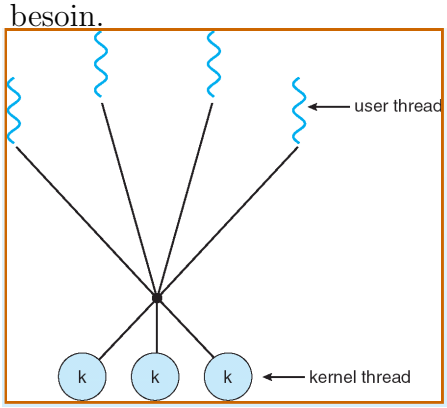
Many-to-one: Gestion des threads dans l'espace utilisateur. Processus entier bloqué si un thread fait un appel système. Un seul thread peut accéder le noyau à la fois.



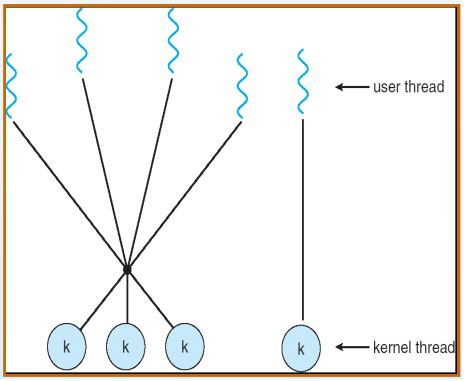
One-to-one: Chaque thread utilisateur a un thread noyau. Permet plus de concurrence, et l'utilisation de multi-processeurs. Création plus lourde: créer thread utilisateur = créer processus noyau.



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).

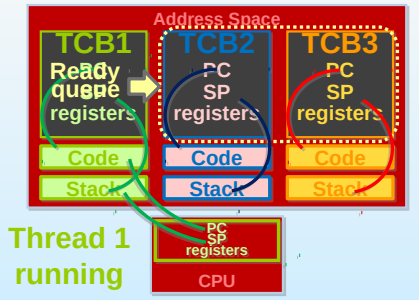
- Etat privé à chaque thread: PC, registres, pile, SP (stack pointer).

- 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.

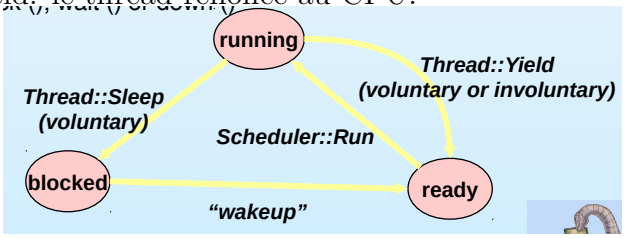
Thread Control Block (TCB): donnée maintenue par l'OS pour décrire chaque thread, contient les données privées d'un thread suspendu (similaire au PCB des processus).

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

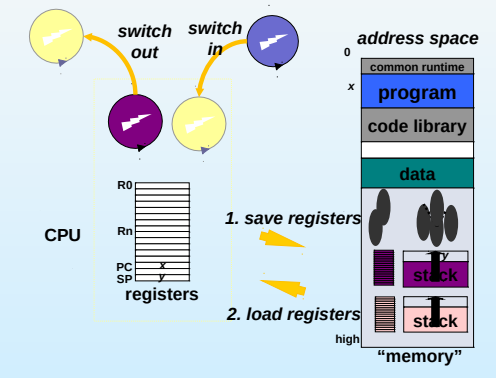


Etats d'un thread. 3 états possibles pour un thread.

Yield: le thread renonce au CPU.



Changement de contexte :



1. Le thread rend le contrôle à l'OS:
   - Evénement volontaire, par exemple un appel à Yield, ou bien appel système, ou en attente de synchronisation;

- Evénement involontaire: interruption matérielle (gestion des interruptions du CPU), l'OS peut forcer un Yield lorsqu'il récupère le contrôle suite à une interruption.
- Rappel: interruptions de timer pour le partage de temps.

2. Choix du prochain thread (par l'OS): on attend si pas de thread prêt, cf chapitre ordo pour les différentes politiques de choix.

3. L'OS sauvegarde l'état du thread courant: dans son TCB! Pas si facile, par exemple pour sauvegarder le PC:

   ```
   100 store PC in TCB
   101 switch to next thread
   ```

   Lorsqu'on restaure le thread, re-execute l'instruction 100! Il faut sauver l'adresse 102...

4. L'OS charge le prochain thread: récupérer son TCB en mémoire, charger les registres et le SP (la pile est déjà en mémoire).

5. L'OS exécute le thread suivant: aller au PC.

Exemple:

```
ChgtContexte (tcb1, tcb2)
   sauver regs dans tcb1
   charger regs de tcb2
   // maintenant, SP pointe vers la pile de tcb2
   aller à tcb2.pc

yield x (x=1,2)
   print "start yield Tx"
   ChgtContexte (tcbx, tcby)
   print "end yield Tx"

Thread x (x=1,2)
   print "start Tx"
   yieldx()
   print "end Tx"
```

On obtient: start T1, start yield T1, start T2, start yield T2, end yield T1, end T1, end yield T2, end T2 (pas de préemption, T1 chargé en premier).

Création d'un nouveau thread, ou "fork" d'un thread:

1. Allouer et initialiser un nouveau TCB;

2. Allouer une nouvelle pile;

3. Faire comme si le thread allait appeler une fonction: PC pointe vers la première instruction, SP pointe vers la nouvelle pile, la pile contient les arguments passés à la fonction;

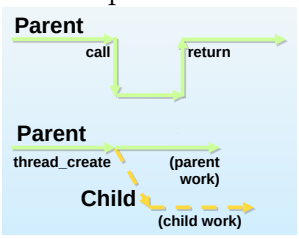
4. Ajouter le thread à la file "ready".

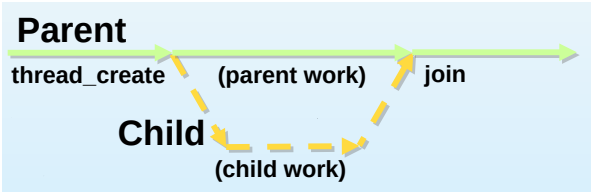
Comment implémenter un join?

```
Child: print "child works"

Parent: create child thread
             print "parent works"
             yield()
             print "parent continues"
```

En sortie, on veut avoir "parent works, child works, parent continues". Peut se produire si child est choisi pour l'exécution après le yield, mais pas de raison! yield = ralentir le CPU, le programme doit fonctionner +- n'importe quel yield! Solutions pour le join dans le chapitre suivant.





3.5 Conclusion

- Thread: flot de contrôle au sein d'un processus.
  - Processus multi-threadé: plusieurs threads dans un même espace d'adressage;
  - Avantages: capacité de réaction, partage des ressources, économie, utilisation d'une architecture multi-processeur.
- Threads utilisateurs (visibles au programmeur, inconnus du noyau) vs threads noyaux (gérés par l'OS): U plus rapides à créer et gérer, pas d'intervention du noyau.
- Les OS gèrent en général les threads, bibliothèques de threads qui fournissent des APIs au programmeur.
- Défis soulevés par la programmation multi-thread.

4 Synchronisation des processus

Exemple: retour au paradigme producteur-consommateur. Dans la solution déjà étudiée, on avait au plus $N-1$ éléments dans le buffer (où N était la taille du buffer). Idée: rajouter un compteur "count" qui indique le nombre d'éléments dans le buffer (pratique pour tests vide/plein, et incrémenter/décrémenter lors de production/consommation).

Problème: s'assurer que deux processus ne vont pas toucher le compteur en même temps! Sinon on peut se retrouver dans un état incohérent.

Implémentation de $count++$:

```
register1 := count;
register1 := register1 + 1;
count := register1;
```

Et pour $count--$:

```
register2 := count;
register2 := register2 - 1;
count := register2;
```

On peut avoir l'exécution suivante, avec initialement $count = 5$:

```
register1 := count;
register1 := register1 + 1;
register2 := count;
register2 := register2 - 1;
count := register1;
count := register2;
```

Et au final on a le compteur à 4 alors que 5 buffers sont remplis!

On ne peut pas (et doit pas) faire d'hypothèses sur l'ordre relatif des exécutions. Seuls comptent: (i) l'ordre d'exécution interne d'un processus; (ii) les relations logiques entre processus (synchronisation).

Le résultat dépend de l'ordre d'exécution entre plusieurs processus, il s'agit d'une **situation de compétition** (race condition).

4.1 Sections critiques et actions atomiques

Objectif: protéger les accès aux variables partagées. Solution: Assurer qu'un ensemble d'opérations est exécuté de manière indivisible (atomique), par exemple les 3 opérations de $count++$.

Section critique: Ensemble d'opérations qui ne doivent pas être exécutées de façon concurrente.

Exclusion mutuelle: Permettre un accès exclusif à un ensemble d'opérations.

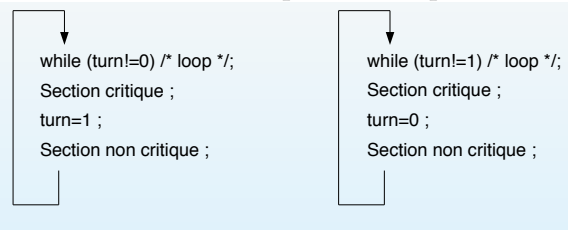
Solution: on veut des opérations d'entrée et de sortie de section critique qui garantissent l'exclusion mutuelle.

```
Processus Pi {
  ...
  Entrée en section critique
       Code section critique
  Sortie de section critique
  ...
}
```

Réalisation d'une section critique:

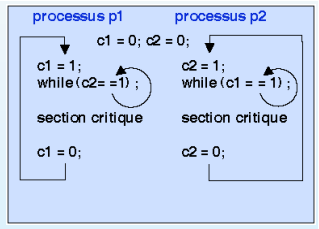
- Attente active: boucle sur un test d'entrée pour entrer en section critique.
  - Très inefficace;
  - Fonctionne s'il n'y a qu'un seul processeur;
  - Utilisé parfois en mode noyau pour de très courtes sections.
- Primitives spéciales atomiques.

Des solutions? 2 processus qui bouclent en attente active:



L'exclusion mutuelle est respectée, mais alternance stricte entre les processus: blocage alors qu'un processus n'est pas en section critique, non désiré!

Deuxième essai avec des drapeaux pour indiquer quand on va rentrer en section critique:



On a maintenant l'exclusion mutuelle et la progression, mais un risque d'interblocage!

Propriétés désirées:

1. Exclusion mutuelle: si P_i est dans sa section critique, aucun autre processus exécute sa section critique;
2. Progression: s'il n'y a aucun processus en section critique et des processus veulent rentrer en section critique, alors la sélection du prochain processus à rentrer en section critique ne peut être repoussée indéfiniment;

3. Attente bornée: lorsqu'un processus P veut rentrer en section critique, il doit y avoir une borne sur le nombre de fois que d'autres processus sont autorisés à rentrer en section critique avant que P puisse rentrer (chaque processus s'exécute à une vitesse non nulle, et pas d'hypothèses sur les vitesses d'exécution).

4.2 Solutions matérielles et logicielles

4.2.1 Solution de Peterson

Solution logicielle, suppose l'atomicité des opérations LOAD et STORE (elles ne peuvent pas être interrompues). Deux processus P_0 et P_1 qui se partagent deux variables *turn* (à qui c'est le tour de rentrer en section critique SC) et *Boolean* $flag[2]$ (vrai si P_i est prêt à rentrer en section critique).

Algorithme pour le processus P_i ($i = 0$ ou 1, et $j = 1 - i$ est l'autre processus):

```
while (true) {
  flag[i] := true;
  turn := j;
  while (flag[j] && turn == j);  // Si l'autre proc veut entrer en SC, il peut
  // SC
  flag[i] := false;
  // code hors section critique
}
```

Preuve:

1. L'exclusion mutuelle est préservée: Si les 2 proc sont en SC, alors $flag[i] = flag[j] = true$. Or, P_i entre en SC seulement si $flag[j] = false$ ou $turn = i$, et $turn$ a une seule valeur → un seul proc peut entrer en SC (par exemple, P_j), l'autre (P_i) se bloque sur le while. On a donc $flag[j] = true$ et $turn = j$, et cela reste vrai tant que P_j est dans sa SC: seul P_j peut mettre $turn$ à i, et mettra $flag[j]$ à $false$ en sortant de SC.

2. La progression et l'attente bornée sont satisfaits: P_i est bloqué seulement si $flag[j] = true$ et $turn = j$. Si P_j n'est pas prêt à rentrer, $flag[j] = false$ et P_i peut entrer. Si $flag[j] = true$, alors $turn = i$ ou j, et l'un des processus va pouvoir rentrer. Si c'était P_j, lorsqu'il sort, il met $flag[j]$ à $false$, ou $turn$ à i si $flag[j]$ devient à nouveau $true$. P_i ne change pas la valeur de $turn$ pendant la boucle while, et donc P_i va entrer en SC (progression) après au plus une entrée de P_j (attente bornée).

4.2.2 Synchronisation matérielle

Pas de *préemption*: pas d'interruption d'un processus s'exécutant en mode noyau, et donc pas de conditions de compétition! Par exemple, désactiver les interruptions sur un uni-processeur.

Par contre on utilise plutôt un noyau préemptif: un processus peut interrompre un processus noyau, c'est plus réactif, mais il faut mettre en oeuvre des solutions au problème de la section critique: besoin d'un verrou.

Machines modernes: instructions matérielles spéciales qui sont atomiques, i.e., on ne peut pas les interrompre. Deux opérations:

1. TestAndSet: teste un mot mémoire et change sa valeur à true

   ```
   boolean TestAndSet (boolean *target) {
     boolean rv := *target;
     *target := true;
     return rv;
   }
   ```

   Utilisation: verrou *lock* partagé, initialement à $false$

   ```
   while (true) {
     while (TestAndSet(&lock)); // boucle vide
     // SC
     lock := false
     // code hors section critique
   }
   ```

2. Swap: échange le contenu de deux mots mémoire

   ```
   void Swap (boolean *a, boolean *b) {
     boolean temp := *a;
     *a := *b;
     *b := temp;
   }
   ```

   Utilisation: verrou *lock* partagé, initialement à $false$, et une variable locale *key* par processus.

   ```
   while (true) {
     key := true;
     while (key := true) { Swap(&lock, &key); }
     // SC
     lock := false;
     // code hors section critique
   }
   ```

Les deux algorithmes satisfont l'exclusion mutuelle, mais pas l'attente bornée...

Algorithme avec attente bornée pour n processus, avec TestAnd Set: variable partagée *lock* et tableau de booléens *waiting*, initialisés à false, et variable locale *key*.

```
while (true) {
  waiting[i] := true;
  key := true;
  while (waiting[i] && key) { key := TestAndSet(&lock); }
```

```
  waiting[i] := false;
  // SC
  j := (i+1) % n;
  while ((j != i) && !waiting[j]) j := (j+1) % n;
  if (j==i) lock := false;
  else waiting[j] := false;
  // code hors section critique
}
```

Preuve que l'algorithme utilisant TestAndSet pour n processus avec attente bornée vérifie les 3 conditions de solution au problème de la section critique:

1. Exclusion mutuelle: P_i entre en SC uniquement si $waiting[i] := false$ ou $key := false$; key passe à $false$ seulement si TestAndSet() est exécuté, et le premier processus à exécuter TestAndSet() trouvera $key = false$, les autres attendent. $waiting[i]$ peut passer à $false$ uniquement si un autre processus quitte sa SC, et un seul $waiting[i]$ est mis à $false$, garantissant ainsi l'exclusion mutuelle.

2. Progression: mêmes arguments que précédemment: lorsqu'un processus sort de SC, il met soit $lock$ à $false$, soit un $waiting[j]$ à $false$, permettant à un autre processus de rentrer en SC.

3. Attente bornée: lors d'une sortie de SC, le tableau est parcouru circulairement dans l'ordre $(i+1, i+2, \ldots, n-1, 0, \ldots, i-1)$, et le premier processus dans cet ordre qui est prêt à rentrer en SC ($waiting[j] = true$) peut entrer. Chaque processus en attente rentrera dans sa SC après au plus $n-1$ tours.

Par contre, le TestAndSet atomique est dur à implémenter sur multiprocesseurs, et ces solutions matérielles sont difficiles à utiliser pour le programmeur ... d'où l'intérêt des sémaphores.

4.2.3 Sémaphores

Outil de synchronisation sans attente active. Sémaphore:

- variable entière S, et deux opérations atomiques:
- opération $wait(S)$ { while $S \leq 0$ { // boucle vide }; $S--$; }
- opération $signal(S)$ { $S++$; }

Deux processus ne peuvent pas modifier simultanément la valeur d'un sémaphore.

Mutex = sémaphore binaire (valeur 0 ou 1): peut être plus simple à implémenter et fournit l'exclusion mutuelle.

Sémaphore S initialisé à 1; $wait(S)$ avant d'entrer en section critique, et $signal(S)$ en sortie de section critique.

Autres utilisations des sémaphores.

- Sémaphore pour contrôler l'accès à une ressource qui a un nombre fini d'instances: initialisée au nombre de ressources; $wait(S)$ pour utiliser une ressource, et $signal(S)$ pour relâcher une ressource. $S = 0$ quand toutes les ressources sont utilisées.
- Sémaphore pour synchroniser deux processus: P_1 execute une instruction S_1, et P_2 a une instruction S_2 qui doit avoir lieu après S_1: initialiser un sémaphore *synch* à 0, puis P_1: S_1; $signal(synch)$; et P_2: $wait(synch)$; S_2;

Implémentation des sémaphores. Doit garantir l'atomicité de $wait()$ et $signal()$: problème de la section critique, où *wait* et *signal* sont placés en SC. Solution avec attente active: *spinlock semaphore*, le processus attend activement le sémaphore. Avantage: pas de changement de contexte, utile si les verrous sont pris pour des durées très courtes. Mais les applications passent beaucoup de temps en SC → ce n'est pas une bonne solution en général.

Solution: modifier les opérations wait et signal, en utilisant une file d'attente des processus bloqués sur un sémaphore. Sémaphore S: deux données: $S.value$, un entier avec la valeur du sémaphore, et $S.list$, une liste de processus.

Deux opérations, implémentées avec deux appels systèmes:

- $block()$ place le processus dans la file d'attente appropriée;
- $wakeup(P)$ défile un processus de la file et le place en état ready.

On peut maintenant écrire le code de wait et signal:

```
wait (semaphore *S) {
  S.value --;
  if (S.value < 0) {
    block(); // met le processus dans S.list
  }
}

signal (semaphore *S) {
  S.value ++;
  if (S.value <= 0) {
    wakeup(P); // enlève un processus P de la liste S.list
  }
}
```

Problèmes possibles avec les sémaphores:

- Utilisation non correcte: signal(mutex) suivi de wait(mutex) (pas d'exclusion mutuelle!); ou wait(mutex) suivi de wait(mutex) (blocage); oubli d'un wait ou d'un signal...
- Interblocages: deux processus ou plus attendent indéfiniment un événement qui peut être causé uniquement par l'un des processus bloqué. Exemple avec deux sémaphores S et Q initialisés à 1:
  P_1 effectue wait(S); wait(Q); ...; signal(S); signal(Q);
  P_2 effectue wait(Q); wait(S); ...; signal(Q); signal(S);

- Famine: un processus peut rester dans la file d'attente de son sémaphore et ne jamais en sortir.

4.3 Problèmes classiques de synchronisation

4.3.1 Producteurs/consommateurs

C'est le problème qu'on a déjà rencontré, avec un buffer de messages de taille fixée N, géré circulairement. Solution avec des sémaphores?

- Sémaphore mutex initialisé à 1 (entrées en SC);
- Sémaphore full initialisé à 0 (nombre de données);
- Sémaphore empty initialisé à N (nombre de cases vides).

Codes symétriques pour le producteur et le consommateur: le producteur produit un item puis wait(empty); wait(mutex); ajoute son item au buffer; signal(mutex); signal(full); et recommence.
Le consommateur effectue wait(full); wait(mutex); retire un item du buffer; signal(mutex); signal(empty); et recommence.

4.3.2 Lecteurs/rédacteurs

Autre problème classique, avec des lecteurs qui ne font que lire une donnée (partagée), et des rédacteurs qui peuvent lire et écrire. On peut avoir plusieurs lectures simultanées, mais un seul rédacteur à la fois. Solution avec des sémaphores?

- Sémaphore mutex initialisé à 1 (entrées en SC);
- Sémaphore wrt initialisé à 1 (nombre de rédacteurs);
- Entier nbLect initialisé à 0 (nombre de lecteurs).

Le rédacteur attend simplement wrt avant de faire son écriture: while(true) { wait(wrt); écrit dans le fichier; signal(wrt)}.

Code du lecteur:

```
while(true) {
  wait(mutex); nbLect ++;
  if (nbLect == 1) wait(wrt);
  signal(mutex);
  // Lecture
  wait(mutex); nbLect --;
  if (nbLect == 0) signal(wrt);
  signal(mutex);
}
```

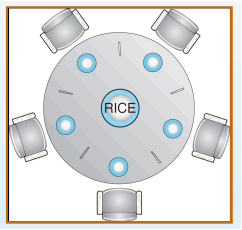
Le premier lecteur se bloque sur wrt si un rédacteur est en SC, et les suivants se bloquent sur mutex. Lors d'un signal(wrt), on peut reprendre l'exécution d'un rédacteur en attente, ou des lecteurs. Le dernier lecteur à sortir de sa lecture fait aussi signal(wrt).

Risque de *famine*: le rédacteur doit attendre que tous les lecteurs aient fini de lire avant de prendre la main, mais le nombre de lecteurs n'est pas borné... Les lecteurs peuvent monopoliser les ressources au détriment des rédacteurs.

Solution plus juste: rajouter un sémaphore rw (read-write) initialisé à 1. Objectif: si un rédacteur est en attente, ne pas autoriser de nouveaux lecteurs à accéder le fichier. Ainsi, le rédacteur commence par wait(rw) et termine par signal(rw). Code du lecteur?

4.3.3 Le dîner des philosophes

Des philosophes sont installés autour d'une table ronde, et un philosophe a besoin de 2 baguettes pour manger. Le philosophe alterne le mode "penser" et le mode "manger".



Solution avec des sémaphores? Tableau de sémaphore baguette[N] (pour N philosophes), initialisés à 1. Le code du philosophe i est le suivant:

```
while(true) {
  wait(baguette[i] );
  wait(baguette[(i+1) mod N]);
  // MANGE
  signal(baguette[i] );
  signal(baguette[(i+1) mod N]);
  // PENSE
}
```

Il se peut que chaque philosophe tienne une baguette, et on se retrouve en interblocage...

Solutions:

- Ne pas avoir plus de 4 philosophes à table;
- Permettre à un philosophe de manger uniquement si les 2 baguettes sont disponibles;
- Solution asymétrique: les philosophes de numéro impair prennent d'abord la baguette $i + 1$.

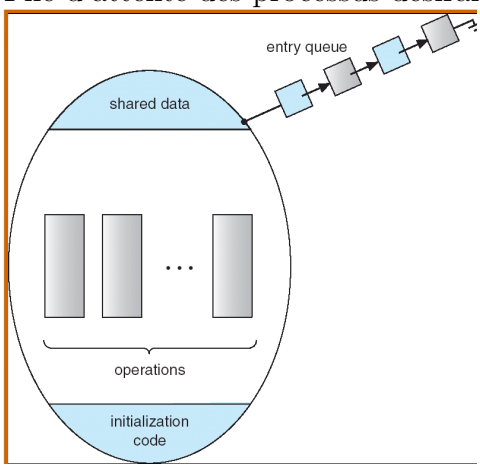
Toute solution satisfaisante doit garantir qu'il n'y ait pas famine, i.e., un des philosophes ne va pas mourir de faim. L'absence d'interblocages ne garantit pas l'absence de famine.

4.4 Les moniteurs

Autre outil de synchronisation: abstraction haut niveau. Dans un moniteur, un seul processus peut être actif à la fois. Procédures définies dans le moniteur.

```
monitor monitor-name {
  // déclaration de variables partagées
  procedure P1 (...) {...}
  ...
  initialisation (...) {...}
}
```

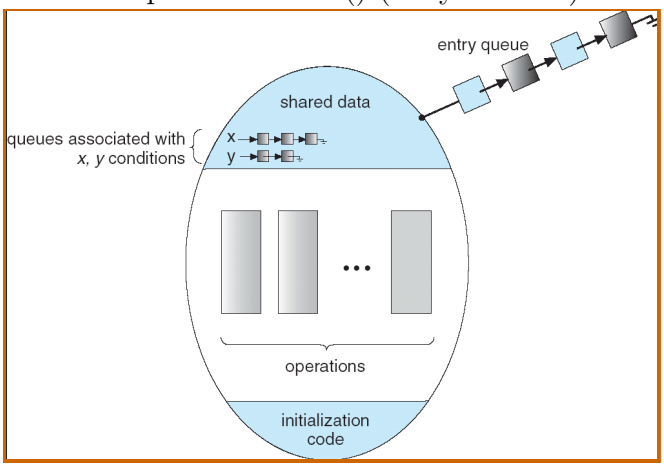
File d'attente des processus désirant entrer dans le moniteur:



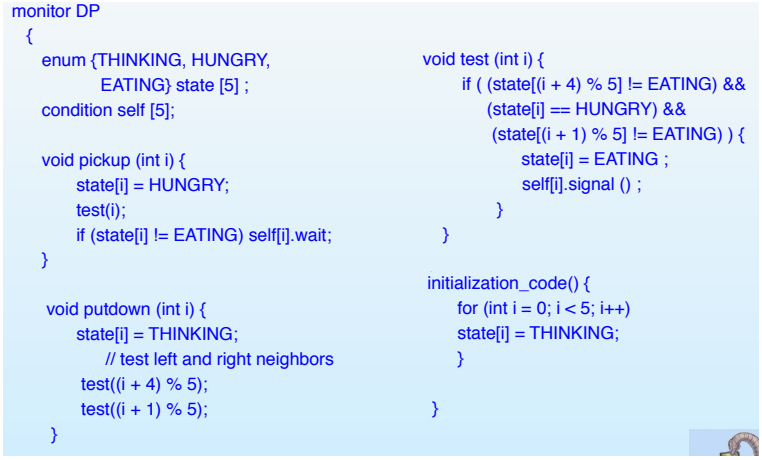
En plus, possibilité d'utiliser des variables de condition:

```
condition x,y;
```

Deux opérations: x.wait() suspend le processus, et x.signal() reprend l'exécution d'un des processus bloqués sur x.wait() (s'il y en a un).



Et voici le code pour le dîner des philosophes (le philosophe i appelle dp.pickup(i), il mange, puis il appelle dp.putdown(i).



Implémentation des moniteurs. On peut facilement implémenter les moniteurs à l'aide de sémaphores: un sémaphore mutex initialisé à 1, et un sémaphore next initialisé à 0. On utilise aussi une variable partagée int next-count = 0.

Les procédures sont remplacées par:

```
wait (mutex);
  // code de la procédure
if (next-count > 0) signal(next);
else signal(mutex);
```

L'exclusion mutuelle est garantie, et next-count indique le nombre de processus qui veulent rentrer dans le moniteur suite à un x.signal.

Pour chaque variable de condition x, on a un sémaphore x-sem initialisé à 0, et int x-count = 0 (nombre de processus bloqués sur x). On a donc:

```
x.wait:
     x-count ++;
     if (next-count > 0) signal (next);
     else signal (mutex);
     wait(x-sem);
     x-count --;

x.signal:
     if (x-count >0) {
        next-count ++;
        signal(x-sem);
        wait(next);
        next-count --;
     }
```

4.5 Conclusion

- Processus séquentiels coopérants partageant des données:
  - Nécessité de fournir un mécanisme d'exclusion mutuelle;

  - Sections critiques: un seul processus/thread à la fois;
  - Plusieurs algorithmes pour résoudre le problème;

- Problème de l'attente active: utiliser les sémaphores. Permet de résoudre les problèmes classiques, utilisés pour tester presque tous les nouveaux schémas de synchronisation;
- Erreurs faciles: solutions de haut-niveau comme les moniteurs et leurs variables de condition;
- Chaque OS fournit un support pour la synchronisation. Par exemple, Linux fournit des sémaphores et spinlocks, et les interruptions sont désactivées pour de petites sections critiques. Pthreads fournit des verrous mutex, des variables de condition et des verrous lecture/écriture, et une extension (POSIX SEM) contient les sémaphores.

Pour tester ses connaissances

1. Proposer une implémentation de l'exclusion mutuelle avec temps d'attente borné à l'aide de l'instruction Swap().
2. Montrer comment implémenter un sémaphore à l'aide d'un moniteur.

```
monitor semaphore {
  int value = 0; condition c;
  semaphore-signal() { value++; c.signal(); }
  semaphore-wait() { if (value == 0) then c.wait();
                                  value --; }
}
```

3. Quelle est la différence entre le signal() d'un sémaphore et celui d'un moniteur?

5 Les interblocages

Objectifs: décrire les interblocages, qui empêchent un ensemble de processus concurrent de compléter leur tâche. Présenter des méthodes pour prévenir ou éviter les interblocages.

5.1 Le problème des interblocages

Ensemble de processus concurrents: chacun tient une ressource et attend une ressource tenue par un autre processus de l'ensemble.

Exemple 1: système avec 2 disques, P_1 et P_2 tiennent chacun un disque et attendent le deuxième pour poursuivre leur exécution.

Exemple 2: deux sémaphores A et B initialisés à 1, P_1 fait wait(A); wait(B); et P_2 fait wait(B); wait(A);

Exemple 3: un pont à sens-unique avec deux voitures qui se sont engagées simultanément. Pour résoudre l'interblocage, il faut qu'une voiture recule, peut obliger d'autres voitures à reculer... Possibilité de famine.

Modèle.

- Un ensemble de types de ressources $\{R_1, R_2, \ldots, R_m\}$ (peuvent être des cycles CPU, de l'espace mémoire, des fichiers, des périphériques IO, ...);
- Chaque ressource de type R_i a W_i instances;
- Un ensemble de processus $\{P_1, P_2, \ldots, P_n\}$;
- Chaque processus utilise les ressources en effectuant une requête de ressource (attend si la requête ne peut pas être satisfaite immédiatement), une utilisation, puis le relâchement de la ressource.

Caractérisation des interblocages. On a un interblocage si les 4 conditions suivantes sont vérifiées simultanément:

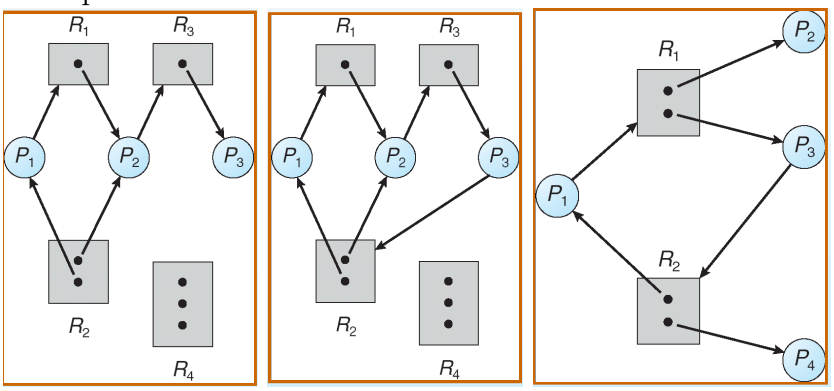
1. Exclusion mutuelle: seulement un processus à la fois peut utiliser une ressource;
2. Tient-et-attend: un processus qui tient au moins une ressource est en attente d'autres ressources;
3. Pas de préemption: une ressource ne peut être relâchée qu'intentionnellement par le processus qui la tient, une fois qu'il a terminé;
4. Attente circulaire: il y a un ensemble $\{P_1, \ldots, P_k\}$ de processus en attente, tels que P_1 attend une ressource tenue par P_2, ..., P_i attend une ressource tenue par P_{i+1}, ... et P_k attend une ressource tenue par P_1.

L'attente circulaire implique le tient-et-attend, les 4 conditions ne sont pas entièrement indépendantes.

Graphe de l'allocation des ressources. Les sommets sont constitués de deux types de noeuds: les processus du système $(P_1, \ldots, P_n)$, et les types de ressources $(R_1, \ldots, R_m)$. Les sommets R_j peuvent être sous-divisés en plusieurs instances de ressources.

Il y a deux types d'arêtes: une arête de **requête** $P_i \to R_j$ si P_i attend la ressource R_j, et une arête d'**assignement** $R_j \to P_i$ si P_i tient la ressource R_j (ou une instance de la ressource).

Exemples:



(a) sans interblocage; (b) avec interblocage: il y a un circuit; (c) avec un circuit mais sans interblocage.

- Si le graphe n'a pas de circuits, il n'y a pas d'interblocages.
- Si le graphe a un circuit, avec une seule instance par type de ressource il y a interblocage, et il y a possibilité d'interblocage s'il y a plusieurs instances par type de ressource.

Petit exercice: On considère un système avec 3 processus et 2 ressources: P_2 demande la ressource A, qui est détenue par P_1. De plus, les trois processus demandent la ressource B.

1. Dessiner le graphe d'allocation des ressources correspondant.
2. Commenter l'état du système si la ressource B est donnée au processus P_1, P_2 ou P_3.

5.2 Méthodes pour gérer les interblocages

Différentes catégories de méthodes:

- S'assurer que le système n'entrera jamais en interblocage: prévenir ou éviter les interblocages;
- Autoriser l'entrée en situation d'interblocage, les détecter et récupérer;
- Ignorer le problème et prétendre qu'il n'y a jamais d'interblocages; utilisé par la plupart des OS, incluant UNIX et Windows!

Souvent, on peut combiner les méthodes (pour différents problèmes d'allocation de ressources du système).

5.2.1 Prévenir des interblocages

Contrainte sur la façon dont les requêtes peuvent être effectuées: s'assurer qu'au moins une des 4 conditions n'est pas vérifiée.

1. Exclusion mutuelle: on peut s'en passer pour les ressources qui peuvent être partagées, mais requis pour les ressources non partageables.

2. Tient-et-attend: il faut s'assurer que quand un processus demande une ressource, il n'en tient pas d'autres: (a) imposer que le processus demande et obtiennent toutes les ressources dont il a besoin avant de commencer son exécution, ou (b) ne permettre de demander une ressource uniquement lorsque le processus n'en tient pas.
   Avec (a), des appels systèmes avant l'exécution pour demander les requêtes, avant tout autre action du processus. Exemple: copier données d'un DVD vers un fichier sur disque, trier le fichier, puis imprimer: il faut réserver le DVD, le disque, et l'imprimante avant de commencer.
   Avec (b), on peut tenir des ressources puis les relâcher avant d'en redemander de nouvelles; pour l'exemple, le processus prend le DVD et le disque, puis le disque et l'imprimante.
   Problème de faible utilisation des ressources, et risque de famine.

3. Pas de préemption: garantir la préemption: si un processus tient des ressources et en demande une autre qui n'est pas disponible, toutes les ressources qu'il tenait lui sont retirées, et ces ressources sont rajoutées à la liste des ressources qu'il attend. Processus redémarré lorsqu'il peut avoir toutes les ressources dont il a besoin. Il faut pouvoir sauvegarder/restaurer facilement l'état d'une ressource.

4. Attente circulaire: Imposer un ordre total sur les types de ressources, et imposer une demande des ressources dans l'ordre croissant. $F(R_i)$: numéro associé à la ressource. Interblocage: P_i tient R_{i-1} et attend R_i, tenu par P_{i+1}. Du coup, on a $F(R_{i-1}) < F(R_i)$, et pour le cycle de k processus, on obtient $F(R_1) < F(R_2) < \cdots < F(R_k) < F(R_1)$, ce qui est absurde. Il faut s'assurer que l'ordre est respecté: témoin qui enregistre l'ordre et envoie des messages d'avertissement.

5.2.2 Eviter les interblocages

Besoin d'information additionnelle, sur la façon dont les ressources vont être demandées.

Modèle (simple et utile): chaque processus déclare le nombre maximum de ressources de chaque type dont il peut avoir besoin. Algorithme: examine l'état d'allocation des ressources (nombre de ressources disponibles et allouées, et demandes max), pour s'assurer qu'il n'y aura jamais de condition d'attente circulaire.

Etat sûr: il existe une séquence $< P_1, \ldots, P_n >$ de tous les processus du système, telle que pour chaque P_i, les ressources que P_i peut encore demander sont soit disponibles immédiatement, soit tenues par les P_j avec $j < i$.

Ainsi, si P_i ne peut pas s'exécuter immédiatement, il peut attendre que tous les P_j, $j < i$, aient terminé, et alors il peut récupérer les ressources dont il a besoin, s'exécuter, et relâcher ses ressources, permettant ainsi à P_{i+1} de s'exécuter.

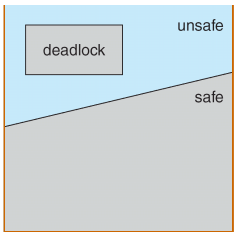
Exemple avec 12 instances d'une ressource.

	Besoin max	Ressources utilisées
P_1	10	5
P_2	4	2
P_3	9	2

L'état est sûr, avec la séquence $< P_2, P_1, P_3 >$.

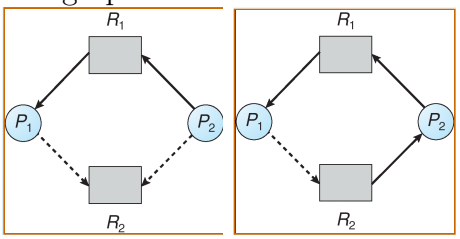
Par contre, si on donne une ressource de plus à P_3, alors on ne peut faire que P_2, puis on a seulement 4 ressources de disponible et on ne peut pas avancer. L'état n'est pas sûr.

Si l'état est sûr, il n'y a pas interblocage. Possibilité d'interblocage si l'état n'est pas sûr. Eviter les interblocages = s'assurer qu'on n'entrera jamais dans un état non sûr.



Avec une seule instance par type de ressource. Utilisation d'un graphe d'allocation des ressources. Arête de *demande* $P_i \to R_j$ qui indique qu'un processus P_i peut avoir besoin de R_j durant son exécution, représentée en pointillés. Devient une arête de *requête* lorsque le processus demande effectivement la ressource. Arête de requête transformée en arête d'*assignement* lorsque la ressource est allouée au processus (arête $R_j \to P_i$). Lorsque la ressource est relâchée, on retourne à l'état avec une arête de demande.

Chaque processus doit donc demander à priori les ressources dont il peut avoir besoin. Exemple où P_2 demande R_2, mais on ne peut pas lui allouer R_2 sinon il y a un circuit dans le graphe.



Algorithme: si P_i demande R_j, la ressource ne lui est allouée que si convertir l'arête de requête en arrête d'assignement ne crée pas de circuit dans le graphe d'allocation des ressources.

Algorithme de recherche de circuit? DFS qui colorie les sommets en blanc, gris, noir. Si on retombe sur un sommet gris (arc arrière), il y a un circuit (cf cours Algo2). Recherche en $O(|V|+|E|)$, où $|V|$ est le nombre de sommets et $|E|$ est le nombre d'arêtes.

Avec plusieurs instances par type de ressource: l'algorithme du banquier. Soit n le nombre de processus et m le nombre de types de ressources. On maintient les structures suivantes:

- *Available* est un vecteur de longueur m: si $Available[j] = k$, il y a k instances de R_j de disponible;
- Max est une matrice $n \times m$: si $Max[i, j] = k$, P_i peut demander au plus k instances de R_j à la fois durant son exécution;
- *Allocation* est une matrice $n \times m$: si $Allocation[i, j] = k$, P_i détient actuellement k instances de R_j;
- $Need$ est une matrice $n \times m$: $Need = Max - Allocation$, i.e., si $Need[i, j] = k$, P_i peut demander au plus k instances supplémentaires de R_j pour compléter sa tâche.

Algorithme de sûreté: détermine si un état est sûr. Utilise les vecteurs $Work$ et $Finish$, de longueur m et n. Initialement, $Work = Available$, et $Finish[i] = false$ pour $1 \leq i \leq n$.
Algo: Tant qu'il existe un processus i tel que $Finish[i] = false$ et $Need_i \leq Work$,
$\{Work := Work + Allocation_i$; $Finish[i] := true$; $\}$
Si $Finish[i] = true$ pour $1 \leq i \leq n$, alors l'état est sûr.

Algorithme du banquier: prend en entrée un vecteur de requêtes du processus P_i, $Request_i$. Si $Request_i[j] = k$, alors P_i demande k instances de R_j.
Algo:
si $Request_i > Need_i$, erreur car le processus a demandé plus que sa demande maximum;
si $Request_i > Available_i$, P_i doit attendre car il n'y a pas assez de ressources disponibles;
sinon, effectuer l'allocation des ressources à P_i:
$Available := Available - Request_i$;
$Allocation_i := Allocation_i + Request_i$;
$Need_i := Need_i - Request_i$;
si l'état obtenu est sûr, les ressources sont allouées à P_i;
sinon, P_i doit attendre et on retourne à l'état précédent.

Exemple avec $n = 5$ processus, $m = 3$ types de ressources, A (10 instances), B (5 instances), et C (7 instances).

On a l'état suivant:

	Allocation	Max
P_1	0 1 0	7 5 3
P_2	2 0 0	3 2 2
P_3	3 0 2	9 0 2
P_4	2 1 1	2 2 2
P_5	0 0 2	4 3 3

Initialement, $Available = (3\ 3\ 2)$.

1. Donner la matrice $Need$.
2. L'état est-il sûr?

Par la suite, si une requête peut être satisfaite, on considèrera qu'elle l'a été dans les questions qui suivent, sinon la requête est suspendue.

3. P_2 effectue la requête (1 0 2). Peut-elle être satisfaite immédiatement?

4. P_5 effectue la requête (3 3 0). Peut-elle être satisfaite immédiatement?

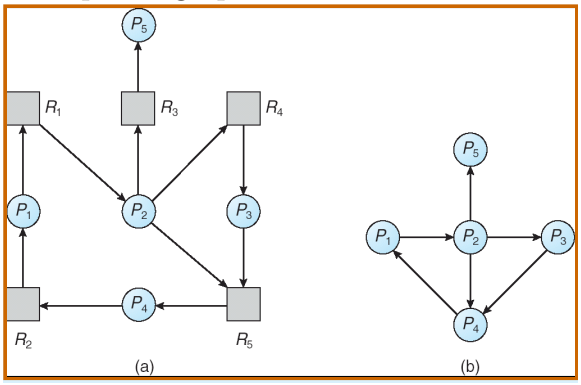
5. P_1 effectue la requête (0 2 0). Peut-elle être satisfaite immédiatement?

5.2.3 Détecter les interblocages

Idée: permettre de rentrer dans une situation d'interblocage: algorithme pour détecter un interblocage et méthode pour récupérer.

Avec une seule instance par type de ressources. Variante du graphe d'allocation des ressources: graphe d'attente. Les noeuds sont les processus, et on a un arc $P_i \to P_j$ si P_i attend que P_j relâche une ressource ($P_i \to R_q$ et $R_q \to P_j$).

Exemple de graphe d'allocation des ressources (a) et de graphe d'attente (b):



On invoque périodiquement un algorithme qui cherche un circuit dans le graphe. S'il y a un circuit, il y a un interblocage.

Avec plusieurs instances. Algorithme similaire à celui du banquier, avec le vecteur *Available* et la matrice *Allocation*. En plus, matrice $n \times m$ *Request*, qui indique la requête courante de chaque processus.

Algorithme de détection: utilise les vecteurs $Work$ et $Finish$, initialisés à $Work = Available$, et pour $1 \leq i \leq n$, si $Allocation_i \neq 0$, alors $Finish[i] = false$, sinon $Finish[i] = true$.

Tant qu'il existe un processus i tel que $Finish[i] = false$ et $Request_i \leq Work$,

$\{Work := Work + Allocation_i; Finish[i] := true; \}$

S'il existe P_i ($1 \leq i \leq n$) tel que $Finish[i] = false$, alors il y a un interblocage. Chaque processus tel que $Finish[i] = false$ est en interblocage.

Exemple avec $n = 5$ processus, $m = 3$ types de ressources, A (7 instances), B (2 instances), et C (6 instances).

On a l'état suivant:

	Allocation	*Request*
P_1	0 1 0	0 0 0
P_2	2 0 0	2 0 1
P_3	3 0 3	0 0 0
P_4	2 1 1	1 0 0
P_5	0 0 2	0 0 2

Initialement, *Available* = (0 0 0).

1. Est-ce qu'il y a un interblocage?
2. Et si P_3 demande une instance supplémentaire de la ressource C?

Usage de l'algorithme de détection. Quand, et à quelle fréquence invoquer l'algorithme de détection? Dépend de la fréquence à laquelle on attend des interblocages, et du nombre de processus qui vont devoir reprendre leur exécution à un point antérieur (un processus par circuit disjoint dans le graphe).

5.2.4 Récupérer d'un interblocage

Solutions: terminer tous les processus en interblocage, ou un processus à la fois jusqu'à ce que l'interblocage soit éliminé. Dans quel ordre terminer les processus?

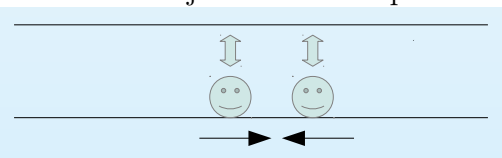
- priorité du processus;
- temps passé à calculer et temps restant jusqu'à complétion;
- ressources utilisées par le processus et besoin en ressources jusqu'à complétion;
- nombre de processus qu'on va devoir terminer;
- processus interactif ou batch.

On veut choisir une victime afin de minimiser le coût. Rollback: on retourne à un état sûr, et on redémarre le processus à partir de cet état. Problème de famine: le même processus peut être toujours choisi comme victime, rajouter le nombre de terminaisons dans le facteur coût.

5.3 Conclusion

- Interblocage: deux processus (ou plus) attendent indéfiniment un événement qui peut être produit uniquement par un bloqué. Approches:
  - Prévenir ou éviter les interblocages: il n'y en a jamais;
  - Permettre de détecter les interblocages et de s'en sortir;
  - Ignorer le problème.
- 4 conditions nécessaires: exclusion mutuelle, tient-et-attend, pas de préemption, attente circulaire.
  - Prévenir: empêcher une de ces conditions;
  - Eviter: information à priori sur l'utilisation des ressources;
  - Détecter: déterminer s'il y a interblocage;
  - Récupérer: terminer des processus ou préempter des ressources.
- Aucune approche n'est suffisante par elle-même.

Et les "livelocks"? Comme un interblocage classique (deadlock), mais les processus changent constamment d'état sans jamais progresser. Exemple du couloir étroit, deux personnes l'une en face de l'autre vont de gauche à droite pour chercher le passage mais se retrouvent toujours dans l'impossibilité de passer.



Petit problème d'application de l'algorithme du banquier. On considère le système suivant, avec 5 processus et 4 ressources, avec plusieurs instances de chaque ressource (3 14 12 12):

	Allocation	*Max*
P_1	0 0 1 2	0 0 1 2
P_2	1 0 0 0	2 7 5 0
P_3	1 3 5 4	2 3 5 6
P_4	0 6 3 2	0 6 5 2
P_5	0 0 1 4	0 6 5 6

1. Donner la matrice $Need$ et le vecteur $Available$.
2. L'état est-il sûr?

Par la suite, si une requête peut être satisfaite, on considèrera qu'elle l'a été dans les questions qui suivent, sinon la requête est suspendue.

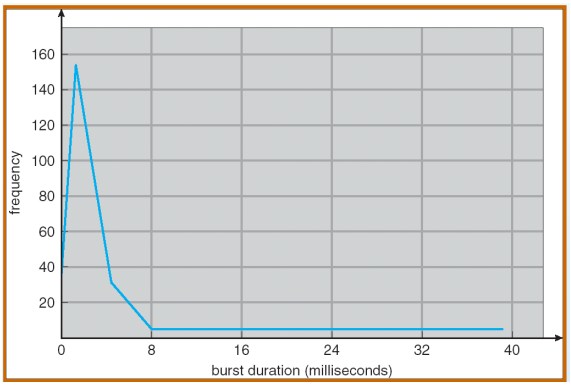
3. P_2 effectue la requête (0 4 2 0). Peut-elle être satisfaite immédiatement?
4. P_3 effectue la requête (1 0 0 1). Peut-elle être satisfaite immédiatement?
5. P_2 effectue la requête (1 0 0 0). Peut-elle être satisfaite immédiatement?

6 Ordonnancement des processus

Concepts de base, critères et algorithmes d'ordonnancement, ordonnancement multi-processeur ou temps réel, exemples d'OS, évaluation des algorithmes.

6.1 Concepts de base

Multiprogrammation pour bien exploiter le CPU: l'execution d'un processus consiste en des cycles d'utilisation CPU et d'attente sur I/O. On parle de *CPU burst*.



Loi exponentielle / hyper-exponentielle: beaucoup de bursts courts, et très peu de bursts longs.

Ordonnanceur: choisit un processus parmi ceux en mémoire qui sont prêts à être exécutés, et lui donne le CPU. Décisions à prendre lorsqu'un processus:

1. passe de l'état running à l'état waiting;
2. passe de l'état running à l'état ready;
3. passe de l'état waiting à l'état ready;
4. se termine.

Ordonnancement coopératif, non-préemptif s'il n'y a que 1 et 4. Sinon, ordonnancement préemptif.

Dispatcher: module qui donne le CPU au processus choisi par l'ordonnanceur court-terme, ce qui implique de changer de contexte, passer en mode utilisateur, et aller au bon endroit dans le programme utilisateur pour redémarrer ce programme.

Doit être très rapide: invoqué à chaque changement de processus. Latence: temps mis pour arrêter un processus et en charger un nouveau.

6.2 Critères et algorithmes d'ordonnancement

6.2.1 Critères

Utilisation CPU – Maintenir le CPU le plus utilisé possible.

Débit – (*throughput*) C'est le nombre de processus qui terminent leur exécution par unité de temps (1 par heure, ou 10 par seconde?).

Temps de retournement – (*turnaround time*) Temps nécessaire pour exécuter un processus (depuis la soumission jusqu'à la terminaison).

Temps d'attente – (*waiting time*) Temps qu'un processus passe dans la file *ready*.

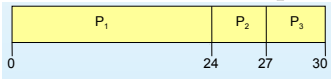
Temps de réponse – (*response time*) Temps entre la soumission d'une requête et la première réponse (et non la sortie finale).

On cherche à maximiser l'utilisation du CPU et le débit, et à minimiser les temps de retournement, attente et réponse. On peut chercher à optimiser la moyenne ou bien le minimum/maximum.

6.2.2 Ordonnancement First-Come, First-Served (FCFS)

Processus	Temps de *burst*
P_1	24
P_2	3
P_3	3

Ordre d'arrivée des processus: P_1, P_2, P_3. L'exécution est:

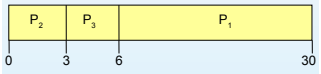


Temps d'attente:

P_1	0
P_2	24
P_3	27

ce qui fait un temps d'attente moyen de $\frac{0+24+27}{3} = 17$.

Si l'ordre d'arrivée est P_2, P_3, P_1, on obtient



Temps d'attente:

P_1	6
P_2	0
P_3	3

ce qui fait un temps d'attente moyen de $\frac{6+0+3}{3} = 3$, bien mieux que l'exécution précédente!

Exemple avec n processus gourmands en I/O, et un gourmand en CPU: si le gourmand en CPU s'exécute en premier, les autres doivent attendre longtemps avant de pouvoir s'exécuter. On parle de l'effet de convoi (*convoy effect*).

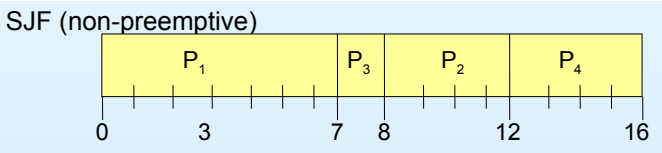
6.2.3 Ordonnancement Shortest-Job-First (SJF)

A chaque processus, on associe la longueur de son prochain burst CPU, et on choisit le processus avec le temps le plus court. Deux versions:

1. **Non préemptif**. Une fois le CPU donné à un processus, on ne peut pas le préempter avant qu'il ait terminé son CPU burst.

Exemple:

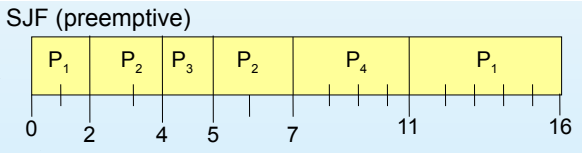
Processus	Temps d'arrivée	Temps de *burst*
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4



Temps d'attente moyen: $\frac{0+6+3+7}{4} = 4$.

2. **Préemptif**. Si un nouveau processus arrive avec un temps d'exécution plus court que le temps restant au processus en cours d'exécution, on lui donne le CPU. *Shortest-Remaining-Time-First (SRTF).*

Avec le même exemple, on obtient



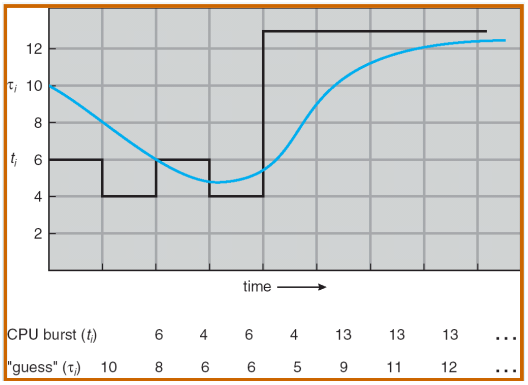
Temps d'attente moyen: $\frac{9+1+0+2}{4} = 3$.

SJF est optimal (dans sa version préemptive, ou avec tous les temps d'arrivée à 0) pour minimiser le temps d'attente moyen.
Si l'on fait passer un processus court devant un plus long, cela diminue le temps d'attente du court plus que l'augmentation du temps d'attente du long. Ainsi, le temps d'attente moyen diminue.
Exemple non préemptif: P_1 arrive au temps 0 et dure 10, P_2 arrive au temps 2 et dure 2. SJF exécute d'abord P_1 puis P_2, ce qui donne un temps d'attente de $8/2 = 4$, alors que l'exécution P_2 puis P_1 donne un temps d'attente de $4/2 = 2$.

Problèmes de SJF: il faut connaître la longueur du prochain burst CPU! S'il n'est pas donné par l'utilisateur (limite sur le temps d'exécution), il faut prédire cette longueur. Utilisation du moyennage exponentiel (*exponential averaging*):

- α est tel que $0 \leq \alpha \leq 1$;
- t_n est la longueur du burst n;
- τ_{n+1} est la valeur prédite du prochain burst $n+1$, et $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$.

Exemple avec $\alpha = \frac{1}{2}$ et $\tau_0 = 10$.



Si $\alpha = 0$, l'histoire récente ne compte pas (i.e., $\tau_{n+1} = \tau_n$), alors que si $\alpha = 1$, seul le dernier CPU burst compte (i.e., $\tau_{n+1} = t_n$). On peut étendre la formule:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \cdots + (1-\alpha)^j \alpha t_{n-j} + \cdots + (1-\alpha)^{n+1}\tau_0.$$

α et $1-\alpha$ sont plus petits que 1, et donc les termes ont de moins en moins de poids.

6.2.4 Ordonnancement avec priorité

Associer une priorité à chaque processus, et donner le CPU au processus le plus prioritaire (petit entier = haute priorité). Version préemptive ou non.

SJF: algorithme avec priorité, où la priorité est l'estimation de la longueur du prochain burst CPU.

Problème de famine: un processus de faible priorité peut ne jamais s'exécuter.

Solution: *aging* (vieillissement): augmenter la priorité d'un processus avec le temps.

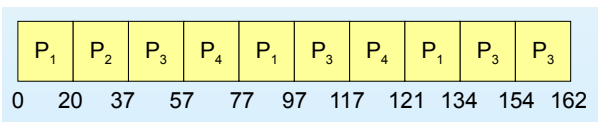
6.2.5 Ordonnancement Round-Robin (RR)

Chaque processus obtient le CPU pour une courte durée (*time quantum*), usuellement entre 10 et 100 millisecondes. Ensuite, le processus est préempté et remis dans la file ready.

Ainsi, s'il y a n processus ready et le time quantum est q, chaque processus obtient $1/n$-ème du temps CPU en tranches d'au plus q à la fois. Ainsi, chaque processus n'attend jamais plus de $(n-1)q$ unités de temps.

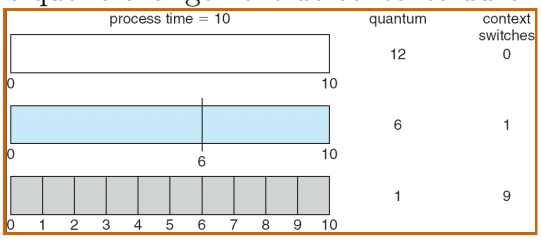
Exemple avec $q = 20$ et 4 processus:

Processus	Temps de *burst*
P_1	53
P_2	17
P_3	68
P_4	24



Le temps de turnaround est souvent plus élevé que celui de SJF, mais meilleur temps de réponse.

Si q est très grand, on se retrouve avec du FCFS. Si q est petit, on a un trop grand surcoût lié aux changements de contexte. Typiquement, q entre 10 et 100 milli-secondes, alors que le changement de contexte dure moins de 10 micro-secondes.

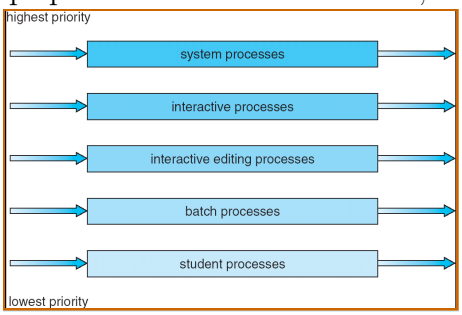


Le temps de turnaround moyen est plus petit si la plupart des processus terminent leur CPU burst durant un q; bon résultats si 80% des CPU burst sont plus petits que q.

6.2.6 Files à plusieurs niveaux

Partager la file ready en plusieurs files distinctes, et chaque file a son propre algorithme d'ordonnancement. Exemple: tâches interactives ordonnancées en RR, et tâches batch en FCFS.

Ordonnancement entre les files: à priorité fixée (on sert d'abord les tâches de la file la plus prioritaire; risque de famine) ou avec un partage de temps (par exemple, 80% du temps pour les tâches interactives, et 20% pour les batch).



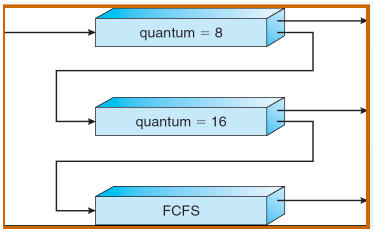
6.2.7 File avec retour d'information

Permet de déplacer un processus d'une file à une autre, ce qui permet d'implémenter le vieillissement.

Ordonnanceur défini par

- le nombre de files;
- l'algorithme d'ordonnancement de chaque file;
- méthode pour décider quand promouvoir un processus;
- méthode pour décider quand rétrograder un processus;
- méthode pour déterminer dans quelle file faire rentrer un nouveau processus.

Exemple avec 3 files, deux avec RR ($q = 8$ ou $q = 16$) et une avec FCFS:



Entrée dans la file avec $q = 8$, s'il ne termine pas en 8 millisecondes, il passe dans la file avec $q = 16$. S'il ne termine pas avec les 16 millisecondes de plus, il passe dans la file FCFS.

6.3 Ordonnancement multi-processeur ou temps réel

L'ordonnancement est plus compliqué en multi-processeur, il faut équilibrer la charge entre les processeurs. Besoin de synchronisation entre processeurs.

Temps réel: il faut terminer des tâches critiques avant des dates fixées d'avance (*hard*), ou alors donner une plus grande priorité aux tâches critiques (*soft*).

6.4 Exemples d'OS

Solaris 2. Trois catégories:

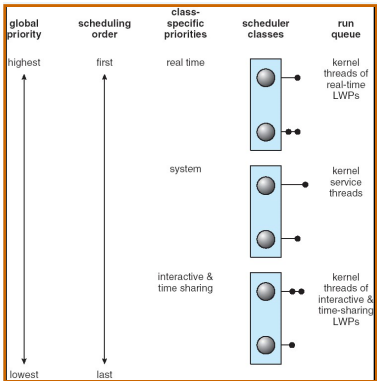


Table de dispatch pour les threads interactifs et à partage de temps. Les processus de forte priorité ont un plus petit q, et une nouvelle priorité plus basse à expiration de q (3ème colonne). Après une I/O, plus grande priorité (4ème colonne).

	priority	time quantum	time quantum expired	return from sleep
(low)	0	200	0	50
	5	200	0	50
	10	160	0	51
	15	160	5	51
	20	120	10	52
	25	120	15	52
	30	80	20	53
	35	80	25	54
	40	40	30	55
	45	40	35	56
	50	40	40	58
	55	40	45	58
(high)	59	20	49	59

Windows XP. Basé sur la priorité, algo préemptif et RR. Listes parcourues par ordre de priorité, l'ordonnanceur cherche le thread de plus haute priorité qui est ready.

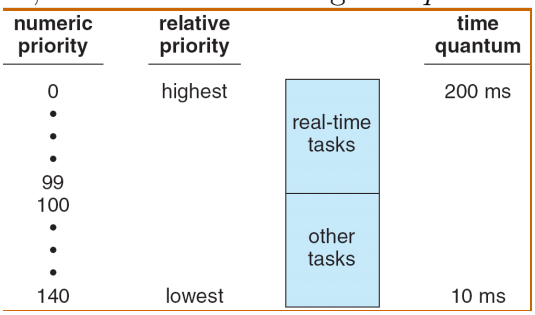
Priorités: classe temps-réel, entre 16 et 31, et les autres ont des priorités entre 1 et 15. Comme avec Solaris, priorité plus basse à expiration d'un time quantum, et priorité augmentée au retour d'une I/O. (Priorité 0 réservée pour la gestion mémoire).

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

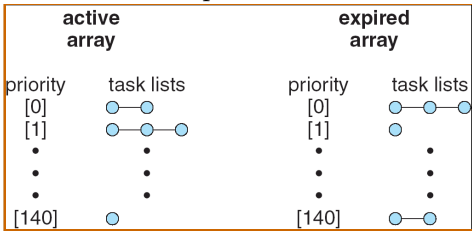
Linux. Deux classes de priorité, pour le partage de temps (mise à jour des priorités comme précédemment) et le temps réel (priorités statiques).

Ordonnancement basé sur des crédits: le processus avec le plus de crédits est ordonnancé; on enlève des crédits lors d'une interruption du timer, et si le crédit atteint 0, on choisit un autre processus. Lorsque tous les processus ont un crédit de 0, on redonne des crédits (basé sur la priorité et l'historique).

Contrairement à précédemment, les petits numéros correspondent aux grandes priorités, et on leur donne un grand q:



On a une liste de tâches par priorité, et à la fin de la tranche de temps, la tâche va dans le tableau *expired*. Une fois le tableau *active* vide, on les échange.



6.5 Evaluation des algorithmes

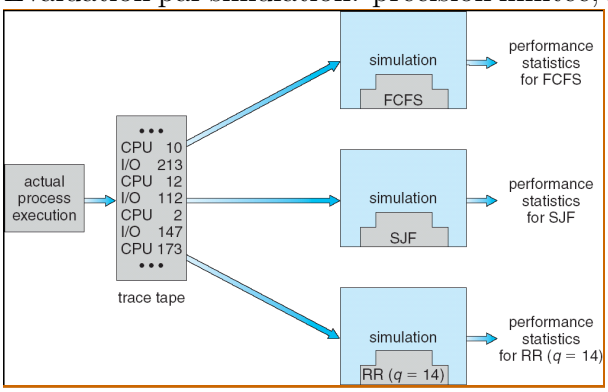
Différents critères, évoqués précédemment: utilisation CPU, temps de réponse, débit, ...

Modélisation déterministe: pour un workload donné, comparer la performance des algorithmes (comme vu sur exemples).

Modèles de files d'attente: distribution des burst CPU et I/O, et distribution des temps d'arrivée. Utilisation de la formule de Little: si la file a une longueur n, le temps

d'attente moyen est W, et le taux moyen d'arrivée est λ, alors $n = \lambda \times W$. Exemple avec λ= 3 processus/sec: pendant une attente de $W = 1$ sec, 3 processus arrivent. A l'état stationnaire, il y a autant d'entrées que de sorties, d'où $n = \lambda \times W$.

Evaluation par simulation: précision limitée, mais plus facile qu'une vraie implémentation.



6.6 Conclusion

- Ordonnancement: choisir un processus prêt et lui donner du temps CPU.
- FCFS: algo simple mais des processus courts peuvent attendre longtemps. SJF: optimal mais difficile de prédire la longueur d'un burst. Cas spécial d'algos avec priorités. Attention à éviter la famine (faire vieillir les processus).
- RR: algo préemptif pour systèmes interactifs.
- Algos à plusieurs niveaux de files d'attentes, combinant par exemple RR et FCFS.
- Méthodes pour évaluer ces algorithmes.

7 Gestion de la mémoire et de la mémoire virtuelle

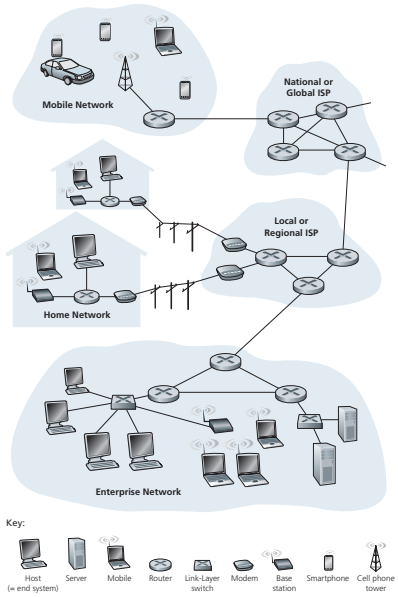
Se référer aux transparents du cours.

PARTIE 2: Réseaux

8 Architecture des réseaux de communication

Réseaux: Internet, téléphonie, ... Tous organisés de la même façon (en couches). On s'intéresse à Internet. Concepts similaires dans les autres réseaux, avec une terminologie différente.

Approche concrète:



Approche fonctionnelle: accès à des applications distribuées, échange de données: Navigation Web, Mail, Messagerie instantanée, Téléphonie, Serveur ftp, Jeux, Pair-à-pair, ...

Les différents types de réseau .

$\leq 1m$	circuit imprimé	Multiprocesseur
	ordinateur	
$10m$	salle	LAN (Local Area Network): réseau local
$\vdots$	bâtiment	
$1km$	campus	
$10km$	ville	MAN (Metropolitan): relie des LANs
$100km$	région	WAN (Wide): relie des MANs et des LANs
$1000km$	continent	
$10000km$	terre entière	Internet

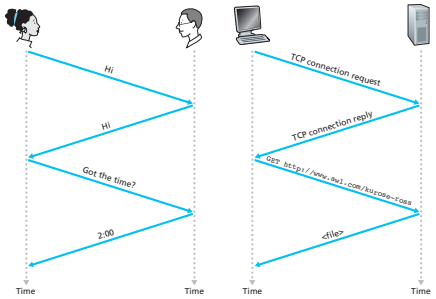
Deux types de service:

- Fiable, orienté, avec connexion
- Non fiable, sans connexion

Pas de service garantissant la durée d'un transfert.

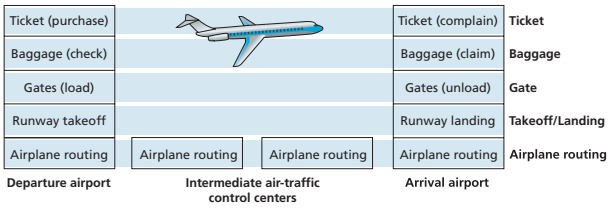
Protocole. Ce sont les protocoles qui gouvernent toutes les communications d'Internet.

Analogie avec le contact humain: demander l'heure, poser une question. Protocoles pour gouverner toutes les communications d'Internet.



8.1 Modèle en couches d'Internet

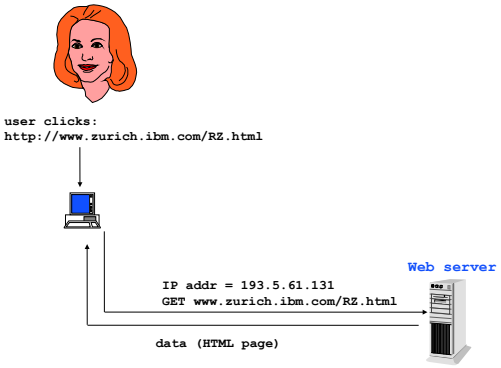
Exemple du réseau aérien:



Les couches d'un réseau TCP/IP:

5	*Application*	Communication
4	*Transport*	
3	*Réseau*	Interconnection
2	*Liaison*	
1	*Physique*	

• Couche 5-appli: aide les personnes et les machines à communiquer. Protocoles= ensemble de règles et de messages (Ex HTTP). Cas le plus simple avec 2 ordis:



• Couche 4-transport: Aide la couche application, avec une interface de programmation (évite à l'utilisateur de répéter les mêmes tâches). Deux protocoles principaux dans TCP/IP:

- UDP (téléphonie, visioconf...). Info transportée = message; non fiable (un message peut être perdu); pas de garantie de séquence.

- TCP: fiable: si une donnée est perdue, on la retransmet. Garantie de séquence. En plus, contrôle de flux/de congestion. Mais unité d'information = octet, données éventuellement groupées différemment à la destination qu'à la source.

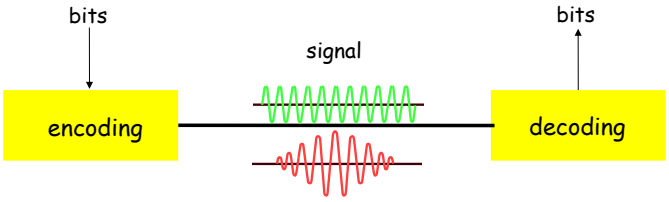
Une application peut utiliser TCP ou UDP selon ses besoins.
Interface de programmation: *socket API*.

• Couche 3-réseau: Fournit la connectivité:
- Utilisation de "routeurs": systèmes intermédiaires non visibles pour l'utilisateur.
- Communication par paquets: l'information est coupée en morceaux (1500B), et des adresses IP sont utilisées pour communiquer.

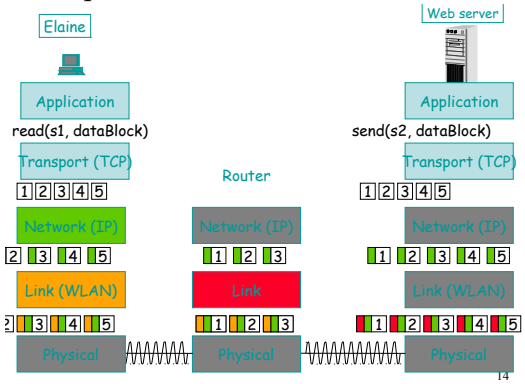
Coeur du réseau (liens entre routeurs): communication par paquets ou commutation de circuits, voir Section 8.2.

• Couche 2-liaison: Adresses MAC en plus des adresses IP (Ethernet), et protocoles ALOHA, CSMA, ...

• Couche 1-physique: Encodage des bits en signal physique (signaux électromagnétiques).

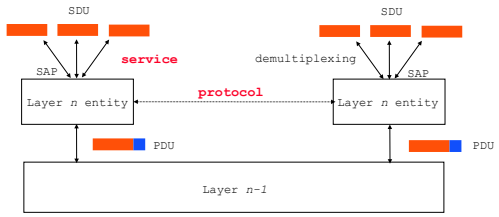


Exemple d'utilisation des couches:



Protocoles et autres définitions. Couche n:
• communique avec les autres entités de couche n en utilisant des PDUs (*Protocol Data Unit*);
• utilise les services de la couche $n-1$ et offre des services à la couche $n+1$: SDUs (*Service Data Unit*).

SAP: *Service Access Point*: interface entre deux couches.
Protocole: règles entre entités de même couche.



Fonctions des couches.

- Détection d'erreurs (améliorer la fiabilité)
- Contrôle de flux (éviter la saturation par PDUs)
- Segmentation puis réassemblage (découpe en paquets)
- Multiplexage (partage d'une même connexion de niveau inférieur)
- Etablissement de connection

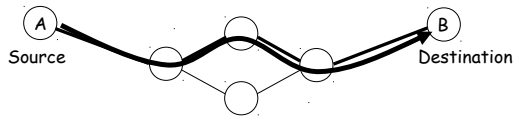
Bilan sur les couches (et PDUs correspondant):
5-appli (Message): executer des applications (HTTP, SMTP, FTP).
4-transport (Paquet/segment): UDP et TCP.
3-réseau (Datagramme): IP, protocoles de routage.
2-liaison (Trame): Ethernet: transmet le paquet de noeud en noeud (routeurs). On véhicule des trames entières
1-physique (Bit): Véhicule les bits de chaque trame, dépend du support physique (fil en cuivre à paire torsadée, cable coaxial, fibre optique, ...)

8.2 Techniques de base: commutation de circuits vs communication par paquets

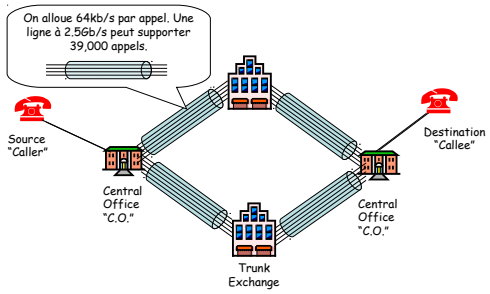
A relié à B par un réseau.

8.2.1 Commutation de circuits

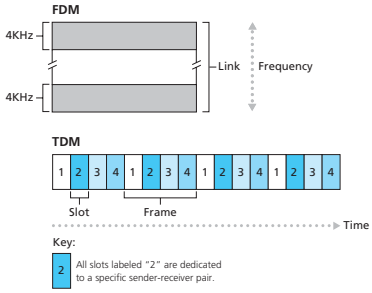
Si A veut appeler B, il faut établir un circuit, communiquer, puis fermer le circuit.



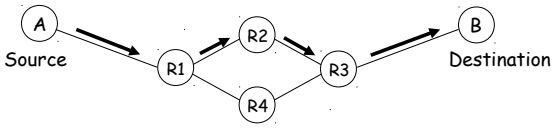
Réseau téléphonique commuté: 1 câble (physique ou virtuel) entre A et B, le débit est garanti de bout en bout.



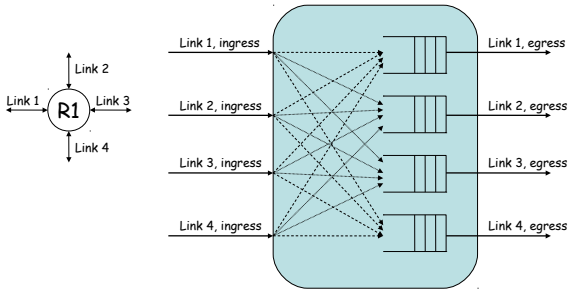
Multiplexage de fréquence vs multiplexage temporel.



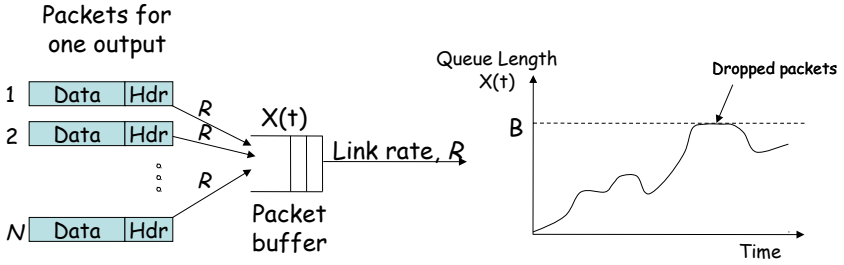
8.2.2 Communication par paquets



Pas de connexion, tables de routage pour aiguiller les paquets. Chaque paquet est indépendant: différents paquets peuvent prendre des routes différentes. Plusieurs paquets peuvent arriver en même temps et vouloir sortir sur le même lien: besoin de buffers au niveau des routeurs.



Multiplexage statistique: le buffer absorbe des rafales temporaires, le lien de sortie n'est pas obligé d'opérer au débit NR. Mais le buffer ayant une taille finie B, des pertes sont possibles.

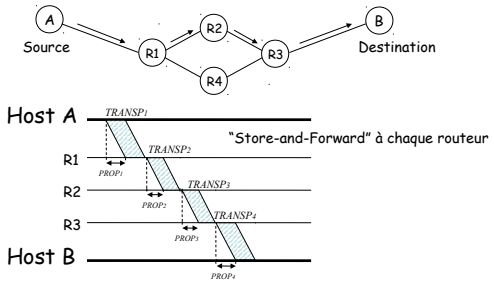


Si A et B envoient avec un débit max x, le taux de A+B est $c < 2x$, d'où un gain de $2x/c$.

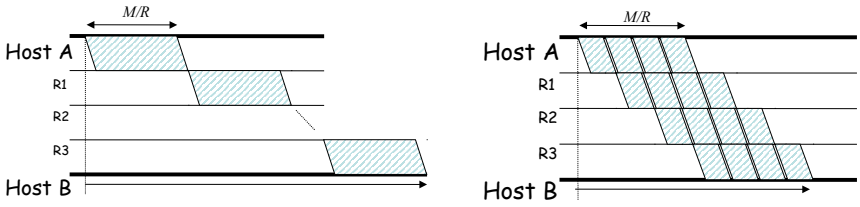
Internet emploie la communication par paquets: utilisation efficace de liens coûteux, et résistance aux pannes (de liens ou de routeurs).

Performance de la communication par paquets. On note:
- M la taille d'un paquet, en bits;
- L la longueur d'un lien, en mètres;
- R le débit, en bits/secondes (b/s);
- D le délai de propagation, i.e., le temps de traversée d'un bit sur un lien de longueur L: $D = L/c$ (noté $PROP$ sur les figs);
- $TRANS$ le temps de transmission, i.e., le temps pour transmettre un paquet de taille M: $TRANS = M/R$ (noté $TRANSP$ sur les figs);
- La latence est le temps entre l'émission du premier bit et la réception du dernier bit. Sur un lien, $latence = D + TRANS$.

Ci dessous, la latence de bout en bout avec 3 routeurs pour aller de A à B est $\sum_i (L_i/c + M/R_i)$.



Pourquoi ne pas envoyer tout le message en un seul paquet?



Décomposer un message en paquets permet d'effectuer des envois en parallèle sur tous les liens, de réduire la latence de bout en bout, et d'éviter qu'un message s'accapare un lien.

1. Calculer le temps de transmission dans les deux scénarios: message non découpé, ou bien message est découpé en k paquets de taille $S = \frac{M}{k}$, et chaque routeur attend d'avoir terminé la transmission d'un paquet avant d'envoyer le suivant (et liens identiques).

   Dans le premier cas, le temps est $n(D + M/R)$.

   Par paquets: Temps pour envoyer le premier paquet: $n\left(D + \frac{S}{R}\right)$.

   Temps pour envoyer les $k-1$ paquets restant: $(k-1)\left(D + \frac{S}{R}\right)$,

   soit un temps total de $T(S) = (n - 1 + \frac{M}{S})\left(D + \frac{S}{R}\right)$.

2. Quelle est la taille optimale des paquets?

   On remarque que $T(S) = \frac{n-1}{R}S + MD\frac{1}{S} + constante$. Pour minimiser cette fonction, qui est de la forme $aS + b/S + c$, on utilise le théorème de la chèvre: un enclos de surface constante ($aS \times b/S = ab$) a un périmètre ($aS + b/S$) de taille minimum lorsque c'est un carré: $aS = b/S$, soit $S = \sqrt{b/a}$. (Vous pouvez aussi dériver la fonction pour vous en convaincre).

   On obtient finalement $S_{opt} = \sqrt{\frac{MDR}{n-1}}$.

   Pour $T(S_{opt})$, on remarque que $aS = b/S$, et la constante vaut $(n-1)D + \frac{M}{R}$, d'où

   $$T(S_{opt}) = 2\sqrt{(n-1)D\,\frac{M}{R}} + (n-1)D + \frac{M}{R} = \left(\sqrt{(n-1)D} + \sqrt{\frac{M}{R}}\right)^2,$$

   et pour des grands messages, le temps est de l'ordre de $\frac{M}{R}$: peu importe n, le nombre de hops!

3. Quels sont les avantages et inconvénients de cette méthode par rapport à l'envoi en un seul paquet?

   Le temps de transmission est meilleur, et en plus en cas d'erreur on n'a qu'un seul paquet à retransmettre. Par contre, il faut rajouter des coûts pour les entêtes de chaque paquet.

Délais liés aux files d'attente. Le routeur R_i doit en fait attendre un délai Q_i avant de pouvoir transmettre le paquet. La latence réelle est $\sum_{i=1}^{n}(L_i/c + M/R_i + Q_i)$.

On note a: taux d'arrivée (paquets/sec), M: taille des paquets (bits), R: taux de transfert (b/s).

Si $aM/R > 1$, la file d'attente croît vers l'infini. Courbe du temps d'attente moyen: 0 quand aM/R vaut 0, et croissance exponentiel en s'approchant de 1.

Capacité limité de la file d'attente → pertes. S'il y a n paquets dans la file, l'attente est $(n-1)M/R$.

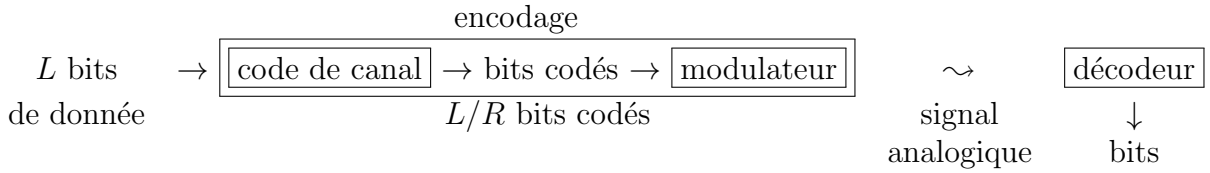
Bilan: circuits vs paquets .

Réseau → comm. circuits → FDM
→ TDM
→ comm. paquets → circuit virtuel
→ datagramme

Circuit virtuel: l'état est maintenu dans les routeurs
(lien1, circuit C1 ⇒ lien 2, circuit C2).

Datagramme: adresse sur l'enveloppe. Avec ou sans connexion.

8.3 La couche 1-physique



Débit binaire (bit rate). C'est le nombre de bits transmis par unité de temps (en bit/sec), on le note b. Loi de Shannon-Hartley:

$$b = B \log_2 \left(1 + \frac{S}{N}\right),$$

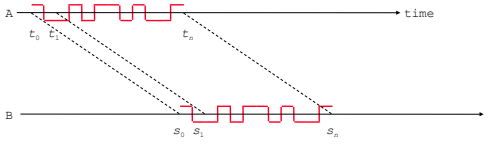
où B est la bande passante (en Hz), et S/N le ratio signal-bruit en dB.

Exemples de débits binaires: modem: 2.4 kb/s à 56 kb/s; ADSL 124 kb/s à 10 Mb/s; Ethernet 10 Mb/s, 100Mb/s, 1Gb/s.

Question: Combien de temps faut-il pour transmettre 1MB à 10 kb/s? La taille du message est $M = 8.10^6$ bits, et $b = R = 10^4$ bits/sec, soit un temps $M/b = 800$ secondes.

Délai de propagation. Temps pour que le début du signal voyage de A vers B.

$D = L/c$, où L est la longueur du lien et c la vitesse du signal (vitesse de la lumière).



$(D = s_0 - t_0)$

Cuivre: $c = 2.3 \times 10^8 m/s$; verre: $c = 2 \times 10^8 m/s$.

Il faut compter environ 5μ sec / km (tour de la terre en 0.2 sec).

Débit (throughput). Nombre de bits **utiles** par unité de temps.

Différent du débit binaire: surcoût du protocole, temps d'attente. Par exemple, pour envoyer un message de 1 octet avec UDP, on crée un paquet Ethernet de 53 octets, et si le débit binaire $b = 64kb/s$, on ne peut obtenir qu'un débit de $1.2kb/s$!

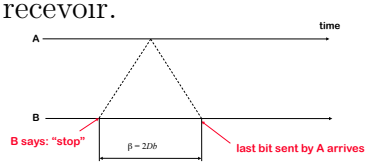
Protocole Stop-and-Go. Possibilité de pertes de paquets: on peut demander des accusés de réception pour s'assurer que les paquets ont été bien reçus.

Protocole Stop-and-Go: si A envoie à B un message M, B envoie un accusé de réception (de taille M') lorsqu'il a reçu M (au temps $D + M/b$), et A attend d'avoir reçu cet accusé avant d'envoyer le message suivant.

Le débit est défini comme le nombre de bits utiles divisé par le temps de cycle, soit
débit $= \frac{M}{M/b+2D+M'/b} = \frac{b}{1+M'/M+2Db/M}$

M'/M est le surcoût lié aux accusés de réception (acks).

$\beta = 2Db$ est le **produit bande-passante délai**, c'est le nombre de bits reçus entre le moment où A dit qu'il ne veut plus rien recevoir, et le moment où il arrête effectivement de recevoir.



9 La couche 2-liaison

Protocoles usuels: Ethernet, PPP, Frame Relay, IEEE 802.11.

Services fournis:

- Tramage: ajout d'une entête aux paquets (adresse physique du noeud destinataire, différente de l'adresse IP de la couche réseau).
- Accès à la liaison: couche MAC (Medium Access Control).
- Transfert fiable: acquittements/retransmissions. Facultatif, sur les liens à fort taux d'erreur.
- Contrôle de flux.
- Détection/correction d'erreurs.
- Exploitation duplex (transmission et réception simultanées).

Certains services sont identiques à ceux fournis par la couche transport, mais ne sont pas de "bout en bout".

Chaque noeud a un adaptateur réseau avec une adresse MAC unique ($48 = 6 \times 8$ bits, noté en hexa par exemple 08:A1:21:71:0D:E4).

9.1 Détection et correction d'erreurs

BER: Bit Error Rate: erreurs liées à la couche physique lors de la transmission des bits. Le BER vaut $\frac{1}{4}10^{-10}$ sur un cable, et peut monter à 10^{-4} pour le Wifi. On veut que la couche MAC délivre des paquets sans erreurs: supprimer les paquets avec erreurs.

Utilisation de CRC: Cyclic Redundancy Checksum (32 bits pour Ethernet).

Simple test de parité: permet de détecter une erreur avec un bit de parité:

$01001001|1 \rightsquigarrow 11001001|1 \rightarrow$ Trame fautive!

Parité à deux dimensions: permet de corriger une erreur. Parité par ligne et par colonne:

1100|1
1001|0
1101: erreur sur la première ligne et la première colonne!

CRC. Générateur G, constitué de $r+1$ bits, connu de l'envoyeur et du récepteur.

Pour envoyer un message D de d bits, on y rajoute r bits R à la fin, obtenant $< D + R > = D.2^r$ XOR R (on décale D de r cases à gauche).

On choisit R pour que $< D + R >$ soit divisible par G:
$D2^r$ XOR $R = nG$; $D2^r = nG$ XOR R ; $\Rightarrow R =$ reste$\left(\frac{D2^r}{G}\right)$.

9.2 Partage de liens dans la couche MAC

Cable / Médium partagé:
• Décodage joint: compliqué!
• Exclusion mutuelle: une seule communication à la fois.

La couche MAC fournit l'exclusion mutuelle! Si un noeud envoie R bits/sec, avec M noeuds en parallèle, on obtient en moyenne R/M bits/sec. Protocole simple (peu coûteux à implémenter) et complètement distribué.

9.2.1 Aloha

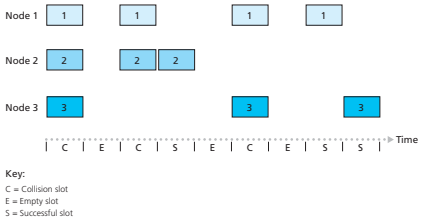
Utilise des accusés de réception et des timers.

Collision: 2 transmissions de paquets se recouvrent. Dans ce cas, les paquets sont perdus et il faut retransmettre.

Version **Pure Aloha**:
- Envoi du paquet
- Attente ACK ou time-out
- Si pas d'ACK (time-out), attendre temps aléatoire et retransmettre.

9.2.2 Slotted Aloha

Même principe mais on synchronise les envois en découpant le temps. C'est plus efficace que pure aloha (environ 2 fois plus efficace), mais besoin de synchronisation!



9.2.3 CSMA

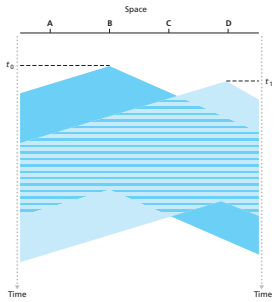
Amélioration par rapport à Aloha: écouter avant de parler!

CSMA = Carrier Sense Multiple Access.

Version **CSMA**:
- Attendre que le canal soit libre
- Envoi du paquet
- Attente ACK ou time-out
- Si time-out, attendre temps aléatoire et recommencer.

Améliore Aloha en évitant certaines collisions, mais le risque de collisions existe toujours (délai de propagation). Besoin de support matériel pour "écouter".

Exemple de collision entre B et D:



Peu de collisions si le temps de transmission est beaucoup plus grand que le temps de propagation: produit bande-passante délai << taille des trames.

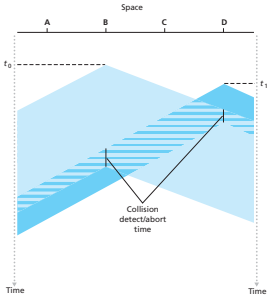
9.2.4 CSMA/CD = Ethernet

CSMA + Collision Detection: détecte les collisions. On remplace donc les ACK par des CD.

Version **CSMA-CD**:
- Attendre que le canal soit libre
- Envoi du paquet et écouter
- Attente (fin de transmission) ou (collision détectée)
- Si collision, envoi de 32 bits de jam puis arrêter la transmission, attendre temps aléatoire et recommencer.
- Sinon, attente délai intertrame (9.6 μ secondes).

Pas besoin d'ACK car l'absence de collisions signifie que la trame a pu être transmise.

Délai intertrame: permet d'éviter les temps "aveugles" alors que les adaptateurs filtrent du bruit.



Meilleure performance que Aloha ou CSMA.

Recul exponentiel (*exponential backoff*).

$k = \min(NbTentatives, 10)$ = le nombre de collisions successives, majoré par 10.

$r = random(0, 2^k - 1) \times SlotTime$, où $SlotTime = 512\times$ le temps de transmission d'un bit (soit $51,2\mu sec$ à 10 Mb/s).

Premier essai: $k = 1$, et $r = 0$ ou $r = SlotTime$.

Deuxième essai (si le premier a échoué): $k = 2$, et $r = 0, 1, 2$ ou $3 \times SlotTime$.

...

Après 10 tentatives ou plus, $r \in [0, 1, \ldots, 1023] \times SlotTime$.

Détecter les collisions. On veut s'assurer que toutes les collisions sont détectées. Exemple où les collisions ne sont pas détectées si le message envoyé est très court.

Si on envoie α bits avec un taux b, il faut $\frac{\alpha}{b} > 2D$. Autrement dit, les messages doivent être de taille supérieure à $\beta = 2Db$ (on retrouve le produit bande-passante délai!).

Ethernet. On fixe $\beta = 512bits = 64B$ (produit bande-passante délai + marge de sécurité). Toutes les trames doivent être de taille au moins β. Ainsi,
• toutes les collisions sont détectées par la source lors d'une transmission;
• les trames en collision sont plus petites que β.

Le diamètre du réseau doit être plus petit si on augmente la bande passante. Ainsi, 10 Mb/s $\rightsquigarrow$ 2km, 100 Mb/s $\rightsquigarrow$ 200m, et pour Ethernet 1Gb/s, on n'utilise pas CSMA/CD.

Ethernet, c'est donc CSMA/CD + exponential backoff. Pas besoin d'ACK, les collisions sont détectées. Délai entre l'envoi de 2 trames de 9.6μsec (filtrer les bruits). Les trames ont une taille entre 64 B (pour CSMA/CD) et 1500 B + header (pour la taille des buffers).

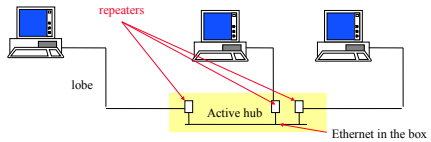
Les adresses MAC source et destination sont dans la trame. Les trames sont lues par toutes les machines connectées. Trame conservée ssi je suis le destinataire.

9.3 MAC pour l'interconnection de réseaux

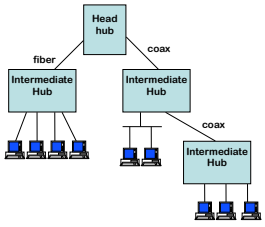
Comment construire un réseau sans routeurs.

1. Câble coaxial. Réseau de taille 200m à 100 Mb/s.

2. Hub actif et répéteurs. Pour résoudre les problèmes de câble, utilisation de hub actifs. Les répéteurs "répètent" les bits (couche physique). Au plus 4 répéteurs sur un chemin.

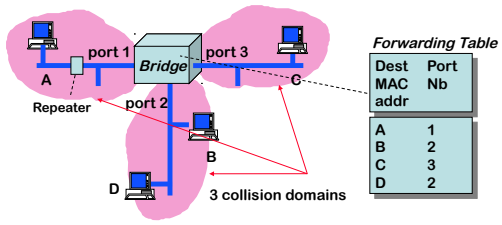


3. Réseau sous forme d'arbre de hubs. Un seul domaine de collision.

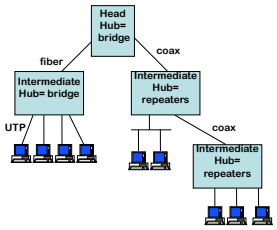


4. Les ponts. • Permet de séparer les domaines de collision.
• Transfert des trames suivant l'adresse MAC.

On peut avoir un paquet de B à D, et en parallèle un paquet de A à C (pas de collision).



5. Ponts et répéteurs combinés. Réseau sous forme d'arbres de hubs et de ponts.



6. Utilisation de câble UTP point-à-point. Permet uniquement une liaison point-à-point, moins cher que le coaxial et plus facile d'utilisation. Avec deux paires de câble, on obtient le Full duplex Ethernet (utilisation dans les deux directions). Plus besoin de CSMA/CD! Seuls restent les formats des trames et les adresses!

7. Protocole d'arbre couvrant. Permet de déployer des ponts sur n'importe quelle topologie. Réseau Ethernet: une centaine de machines.

Conclusion. Couche MAC en Wifi: partage des ondes. Avec câbles: CSMA/CD = Ethernet pour le partage. Maintenant, *Full duplex Ethernet*: liaisons point-à-point et ponts.

5			5	
4			4	IP
3	trame Eth		3	$\longleftrightarrow$ 3
2	$\longleftrightarrow$	2 $\longleftrightarrow$	2	2
1	1	1	1	1
	répéteur	pont		routeur

Architecture vs produits: un *switch* est le nom d'un produit qui est en fait un pont (nom de l'architecture). Le nom du produit dit comment l'objet est implémenté (nom de marketing).

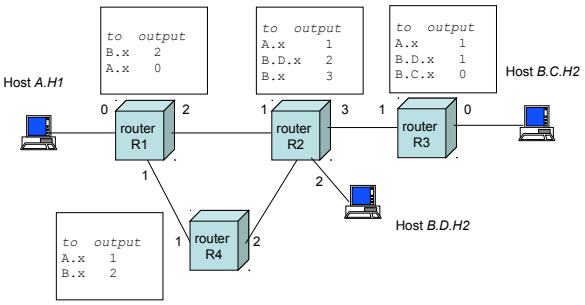
10 La couche 3-réseau

Les principes d'IP, mais nous verrons les algorithmes de routage en section 11.

10.1 Principes

Pourquoi une couche réseau?
• La solution Ethernet ne passe pas à l'échelle. .
• Solution = couche réseau sans connection. IP: Internet Protocol.
• Chaque hôte a une adresse IP.
• Les routeurs retransmettent les paquets suivant les adresses de destination.
• Chaque paquet contient une adresse IP, et les routeurs transmettent les paquets suivant le **plus long préfixe**.

Exemple:



Adresses IP uniques = adresse d'une interface. Pour communiquer, il faut connaître les adresses IP.

Routeurs: interconnectent des sous-réseaux = ensemble d'hôtes avec le même préfixe. Pas de routeur à l'intérieur d'un sous-réseau.

Hôte: transmet le paquet directement via le LAN, ou le donne à un routeur si c'est pour un autre sous-réseau.

10.2 Adresses IP

IPv4: adresses sur 32 bits, notées en notation décimale à points, par exemple 192.78.32.2.

Une partie préfixe et une partie hôte: adresse IP = préfixe:hote. Le préfixe permet d'identifier le sous-réseau, il est utilisé pour le routage. Utilisation de masque de sous-réseau pour identifier le préfixe.

Plusieurs notations:
décimale: 234
binaire: b1110 1010
hexadécimale: xEA

Passer du binaire à l'hexa est facile: 1 chiffre hexa correspond à 4 bits binaire: xE = b1110, xA=b1010.

Passer du binaire au décimale: calculette! b1110 1010 = 128+64+32+8+2 = 234.

Cas particuliers: xF=b1111=15; xFF=b1111 1111 = 255.

Adresses IP sur 32 bits = 4 groupes de 8 bits. Exemples (à compléter):

Décimal	Hexa	Binaire
128.191.151.1	x80 BF 97 01	b0100 0000 1011 1111 1001 0111 0000 0001
129.192.152.2	?	?

Préfixe de sous-réseau: Notation masque ou préfixe.

Masque: on donne une adresse et un masque, par exemple

• Ex1: 128.178.156.13 masque 255.255.255.0

Le masque est la représentation décimale d'une chaîne constituée de 1 sur le préfixe et 0 ailleurs. Si on fait un & sur les bits, on obtient le préfixe. Sur l'exemple, le préfixe est 128.178.156.0.

• Ex2: 129.132.119.77 masque 255.255.255.192

Quel est le préfixe? Combien d'hôtes peuvent être sur le sous-réseau?

Préfixe 129.132.119.64: 26 bits de sous-réseau et 6 bits pour l'hôte. (77=b0100 1101, 192=b1100 0000). Il peut donc y avoir 64 adresses sur ce sous-réseau. Les adresses 0...0 et 1...1 sont réservées, donc au plus 62 hôtes.

Notation préfixe:

• Ex1: 128.178.156.13/24: on indique le nombre de bits qui constituent le préfixe (24 dans ce cas). Les bits en plus sont ignorés: 128.178.156.13/24 = 128.178.156.22/24.

• Ecrire 129.132.119.77 masque 255.255.255.192 en notation préfixe. C'est 129.132.119.77/26, ou 129.132.119.64/26.

• Est ce que les préfixes suivant sont différents?

201.10.0.0/28, 201.10.0.16/28, 201.10.0.32/28, 201.10.0.48/28

Combien d'adresses IP peuvent être allouées à chaque sous-réseau?

4 préfixes différents car ils diffèrent par des bits autre que les 4 derniers. Il reste 16 adresses possibles, moins les 2 réservées, soit 14 hôtes possibles par sous-réseau.

Le préfixe peut lui-même être structuré en sous-préfixes.

Délégation d'adresses. Deux fournisseurs d'accès à internet, ISP1 (62.125/16) et ISP2 (195.44/14). Un client, qui possède les adresses 62.125.44.50/24 chez ISP1, désire passer chez ISP2. Quel impact cela a-t-il?

Si le client souhaite garder les mêmes adresses IP, les adresses de ISP2 ne sont plus contigües, et les tables de routage doivent représenter ISP2 avec deux entrées, 195.44/14 et 62.125.44.50/24.

Adresses IP particulières. Adresse destination 255.255.255.255: broadcast limité, non transmis via les routeurs.

prefixe.1...1: broadcast sur le sous-réseau identifié par le préfixe.

prefixe.0...0: vieille notation pour broadcast.

10/8, 172.16/12, 192.168/16: réservé pour usage interne (intranets).

10.3 Transfert de paquets IP

C'est l'algorithme au coeur de l'architecture TCP/IP: décide ce qu'un système doit faire d'un paquet. Les règles sont simples.

A chaque routeur, on a

- une table des interfaces physiques:

IP	Masque	Numéro d'interface
A	SM	

- une table de routage:

IP dest.	Masque	Next hop
destAR	SM	
default		

destAdr = adresse destinataire dans le paquet IP.

destAR = adresse dans la table de routage.

Les cas sont testés dans cet ordre, les uns après les autres. On s'arrête dès qu'un des cas a réussi.

Cas 1: pour chaque entrée de la table de routage, si (destAdr=destAR), alors envoyer au next hop (il existe un chemin direct vers l'hôte).

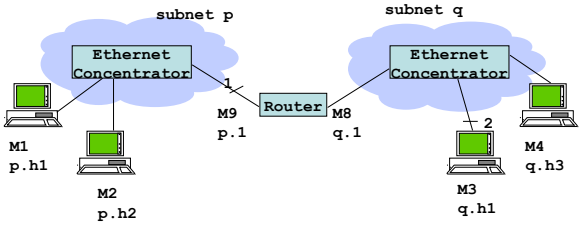
Cas 2: pour chaque entrée de la table des interfaces, si (A & SM = destAdr & SM), envoi direct du paquet sur l'interface (destAdr est sur un réseau directement connecté au routeur).

Cas 3: pour chaque entrée de la table de routage + masque SM, choix du plus long préfixe tel que (destAR & SM = destAdr & SM), et envoi au next hop (il existe une route réseau vers destAdr, on prend celle de plus long préfixe).

Cas 4: pour chaque entrée, si (destAR=default), envoi au next hop (on prend la route par défaut s'il y en a une).

Cas 5: Renvoi message ICMP à la source: "destination non atteignable".

Exemple avec deux sous-réseaux. Mx = adresse MAC.



$M1$ envoie un paquet à $M3$: quelles soit les adresses MAC et IP aux points 1 et 2? Et si $M4$ envoie un paquet à $M3$?

$M1 \rightarrow M3$: En 1, srcIP=p.h1, destIP=q.h1, srcMAC=M1, destMAC=M9.
En 2, adresses IP identiques, mais srcMAC=M8 et destMAC=M3.
Pour $M4 \rightarrow M3$, pas de passage par le routeur. On ne voit le paquet qu'en 2, et on a donc srcIP=q.h3, destIP=q.h1, srcMAC=M4, destMAC=M3.

10.4 Protocole ARP

M1/ip1 veut envoyer un paquet à ip2. Il y a 4 machines sur le sous-réseau (M1/ip1, M2/ip2, M3/ip3, M4/ip4, et le routeur M5/ip5).

1. Envoi d'une requête ARP en broadcast sur le sous-réseau, avec targetIP=ip2. Le routeur ne transmet dans la requête. Dans le paquet, il y a ip1 et M1.
2. M2 s'est reconnu, et envoie donc ARP-reply avec ip2 et M2 à l'originaire de la requête (ip1/M1).
3. M1 envoie le paquet IP à M2.

Table ARP maintenue à chaque hôte, avec les correspondances IP/MAC. Après l'étape 1, tous les hôtes connaissent la correspondance ip1/M1.

Exemple de la section précédente: que doit faire le routeur lorsqu'il reçoit un paquet de M1 pour M3 pour la première fois? Il connait l'adresse IP du destinataire, q.h1, et effectue donc une requête ARP sur le sous-réseau q pour q.h1.

10.5 Entête IP

Cf feuilles distribuées en cours, et IPv4 vs IPv6.

10.6 Conclusion

IP: couche réseau sans connection.
Adresses IP sur 32 bits.
Une adresse IP par interface.
Routeurs: bon passage à l'échelle car ils peuvent agréger les routes.
Hôtes sur Internet: échangent des paquets avec des adresses IP.

11 Algorithmes de routage

11.1 Introduction: les différents types d'algorithmes de routage

IP: tables de routage maintenues au niveau des hôtes et des routeurs, tables utilisées par l'algorithme de transfert de paquets IP.

Routage: méthode de contrôle qui maintient les tables de routage automatiquement au niveau des routeurs. Au niveau des hôtes, utilisation de règles par défaut, et de ICMP.

Différence avec les ponts au niveau du LAN en couche 2? Comment étaient maintenues les informations de routage? En couche 2, les ponts apprennent des paquets qu'ils observent, et utilisent du broadcast initialement.

Différence entre routage et transfert de paquets. Transfert de paquets IP: fait en temps réel pour chaque paquet.

Routage: calculer les tables de routages, uniquement entre routeurs. Pas forcément temps réel: peut mettre jusqu'à 2 minutes! Le but est de minimiser une métrique, comme par exemple le nombre de hops, la capacité des liens, le coût (métriques statiques), ou encore la charge des liens et le délai courant (métriques dynamiques qui dépendent de l'état du réseau).

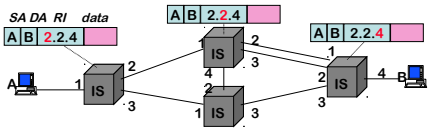
Méthodes de routage de base.

1. Configuration statique: pour jouer uniquement (tous petits réseaux).

2. Inondation: chaque paquet est dupliqué sur chaque lien sortant (noter l'id du paquet pour éviter les boucles). La destination peut recevoir plusieurs fois le même paquet. C'est un algorithme simple (pas besoin de tables de routage), robuste (résiste aux pannes de routeurs ou de liens), et optimal en terme de plus court chemin. Par contre, très coûteux et génère beaucoup de traffic inutile. Méthode utilisée en partie par certaines méthodes de routage (AODV, OLSR).

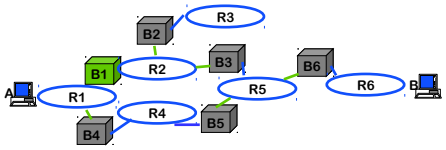
3. Routage par la source: la route est écrite par la source dans l'entête du paquet: liste des numéros de port qu'il faut emprunter. Le routeur lit le prochain hop et déplace le pointeur. Les routes sont découvertes par inondation, puis une route indiquée dans le paquet.

   Exemple: routes qui peuvent être utilisées entre A et B?



4. Anneaux de jetons: inventés dans les années 1980 comme une alternative à Ethernet. Chaque anneau correspond à un domaine de collision, interconnectés par des ponts. A génère un paquet en broadcast-toute-route, tous les ponts écoutent sur tous les anneaux auxquels ils sont connectés, et lorsqu'ils voient un paquet broadcast-toute-route, ils le retransmettent vers tous les autres anneaux, sauf ceux déjà visités par le paquet (on garde cette liste dans l'entête du paquet), ce qui crée 5 paquets différents

sur l'exemple: A-R1-B1-R2-B2-R3, A-R1-B1-R2-B3-R5-B6-R6, A-R1-B1-R2-B3-R5-B5-R4, A-R1-B4-R4-B5-R5-B3-R2-B2-R3, A-R1-B4-R4-B5-R5-B6-R6. Seuls les paquets 2 et 5 atteignent la destination B, A reçoit deux ACK qui prennent la route inverse, et peut choisir une de ces routes.



Classification des algorithmes de routage.

- Routage interne (interior) vs externe (exterior): Deux types de méthodes, suivant que ce soit dans un même domaine, ou entre domaines.
- Algorithme statique (les routes changent lentement, intervention humaine possible) vs dynamique (changement des chemins de routage avec les modifications de topologie ou de charge sur les liens, réactif aux modifications, mais plus sujet aux problèmes tels les boucles, l'oscillation entre routes).
- Algorithme sensible à la charge (coût des liens qui change dynamiquement, représentant le niveau de congestion du lien) vs non sensible à la charge (c'est le cas des algorithmes de routage actuels RIP, OSPF, BGP: le coût d'un lien ne représente pas explicitement le niveau de congestion courant).

Routage à états de liens (*link state*). Algorithme de routage global qui suppose une connaissance de l'état global du système: si on connait tous les coûts des liens, on peut calculer tous les plus courts chemins vers toutes les destinations.

Fonctionnement: chaque noeud diffuse des paquets avec l'état de ses liens vers tous les autres noeuds (broadcast), et ainsi chaque noeud dispose d'une connaissance globale du réseau. Chaque noeud utilise alors l'algorithme de Dijkstra pour calculer les plus courts chemins vers tous les autres noeuds.

Utilisé dans des protocoles de routage interne, tels OSPF, PNNI (ATM).

Routage par vecteur de distance (*distance vector*). Autre grande famille d'algorithmes de routage, basés sur Bellman-Ford. Les routeurs ne connaissent que leur état local (estimation vers les voisins).

Utilisé dans des protocoles internes (RIP, IGRP).

Variante avec vecteur de chemin, utilisé en protocole externe (BGP): maintient l'information sur les chemins, l'optimisation globale et la politique de routage.

11.2 Routage par vecteur de distance

Algorithme qui calcule les plus courts chemins vers toutes les destinations de façon totalement distribuée, en utilisant uniquement les distances entre soi-même et toutes les destinations.

Utilisation de Bellman-Ford distribué (pas facile de distribuer Dijkstra). Coût $A(i,j)$ du lien (i,j), et le but est de calculer le plus court chemin PCC pour tout couple de sommet. $A(i,j) > 0$, et $A(i,j) = +\infty$ si i et j ne sont pas connectés.

Version centralisée, BFC. $p^k(i)$ est le coût de PCC de i vers j en au plus k hops, où j est fixé.

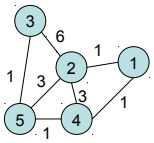
BFC (pour j fixé):
Initialement, $p^0(j) = 0$, et $p^0(i) = +\infty$ pour $i \neq j$.
Tant que $p^k \neq p^{k-1}$,

$p^k(i) = \min_{\ell \neq i}\{A(i,\ell) + p^{k-1}(\ell)\}$ pour $i \neq j$,

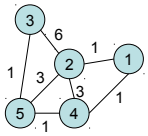
et $p^k(j) = 0$.

Théorème: si le réseau est connexe, l'algorithme s'arrête au plus tard pour $k = n$, et alors $p^n(i) = p(i)$ est le coût d'un PCC pour tout i. On définit le prédécesseur de i sur le chemin par $pred(i) = \text{argmin}_{\ell \neq i}\{A(i,\ell) + p(\ell)\}$.

Exemple: écrire $p^k(i)$, $pred(i)$, et dessiner les PCC depuis $j = 1$.



Impact des conditions initiales. Est-ce-que l'algorithme converge si l'on change les conditions initiales?



k\i	1	2	3	4	5
0	0	0	0	0	0
1					
2					
3					
4					

k\i	1	2	3	4	5
0	0	6	1	1	0
1					
2					
3					
4					

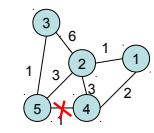
Théorème: l'algorithme converge en un nombre fini d'étapes vers les valeurs correctes pour toute condition initiale telle que $p^0(j) = 0$ et pour chaque noeud i connecté à j.

Version distribuée 1ère version BFD1. Chaque noeud i maintient une estimation $q(i)$ de la distance $p(i)$ de i à 1; $q(1) = 0$ tout le temps et les autres conditions initiales sont quelconques.

De temps en temps, i envoie sa valeur de $q(i)$ à tous ses voisins.

Lorsque i reçoit un $q(j_0)$ depuis un voisin j_0, il met à jour $q(j_0)$ et

$$q(i) = \min_{j \text{ voisin de } i}\{A(i,j) + q(j)\}. \qquad (1)$$



Q: give a possible scenario after link 4-5 breaks

A possible run of algorithm BFD1:

i	1	2	3	4	5
	0	∞	∞	∞	∞
1 -> 2	0	1	∞	∞	∞
2 -> 5	0	1	∞	∞	4
2 -> 3	0	1	7	∞	4
5 -> 4	0	1	7	5	4
2 -> 4	0	1	7	4	4
1 -> 4	0	1	7	2	4
4 -> 5	0	1	7	2	3
5 -> 2	0	1	7	2	3
5 -> 3	0	1	4	2	3
link breaks	4 does as if received ∞ from 5 5 does as if received ∞ from 4 and continue computations from there				
	0	1	4	2	4
5 -> 3	0	1	5	2	4

Si le temps d'envoi d'un message est borné par T, l'algorithme converge au même résultat que la version centralisée en temps au plus nT (si le réseau est connexe et s'il n'y a pas de pannes).

Limite de cet algorithme: chaque noeud doit se souvenir des estimations déjà reçues de tous ses voisins, même si ce ne sont pas les meilleurs. Problème en pratique si l'on doit calculer les plus courts chemins vers toutes les destinations (vecteur de distance = distances vers tous les noeuds du graphe).

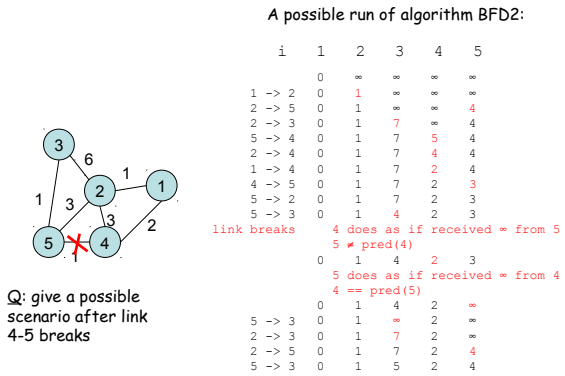
Solution: remplacer l'équation (1) par $q(i) = \min(A(i, j_0) + q(j_0), q(i))$: on choisit le nouveau chemin s'il est plus court que l'ancien. Est-ce une solution convenable?

Version distribuée 2ème version BFD2. Version alternative qui permet de ne se souvenir que du meilleur voisin ($pred(i)$): l'équation (1) est remplacée par:

(2) Si $j_0 = pred(i)$, alors $q(i) = A(i, j_0) + q(j_0)$.

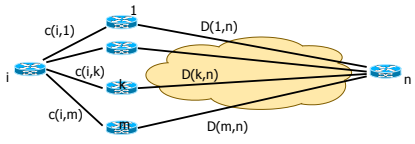
Sinon, si $A(i, j_0) + q(j_0) < q(i)$ alors $q(i) = A(i, j_0) + q(j_0)$ et $pred(i) = j_0$.

Différence avec BFD1 si $j_0 = pred(i)$, car alors on repart de la valeur $A(i, j_0) + q(j_0)$ au lieu de faire le minimum sur tous les voisins. Cette valeur est toujours supérieure à la valeur obtenue par l'équation (1). Après un échange avec les voisins, ceci est réparé et on peut retrouver la valeur de l'équation (1).



En pratique...

- Le noeud i calcule le plus court chemin et next hop pour chaque préfixe de réseau n qu'il connaît.
- Initialement, $D(i, n) = 0$ si i est directement connectée à n, $+\infty$ sinon.
- Le noeud i reçoit du voisin k sa dernière valeur $D(k, n)$ pour tout n (c'est le **vecteur de distance**), et le noeud i calcule les meilleures estimations avec BFD2.
- L'algorithme converge si le réseau est stable. Mécanisme de "hello" pour reprendre les calculs après des modifications: si un voisin k disparaît, timeout car i ne reçoit plus de messages "hello" de k. Equivalent à un message $D(k, n) = +\infty$ pour tout n.



Exemples: voir transparents.

11.3 Protocoles de routage

11.3.1 RIP/RIPv2

Protocole utilisant les vecteurs de distance. La métrique est le nombre de hops. Taille du réseau limitée à 15: $\infty = 16$. Heuristique de l'horizon coupé. Réseau destination identifié par son adresse IP (masques dans RIPv2). Paquets UDP. Broadcast toutes les 30 secondes ou lors de la détection d'une modification, et si une route n'est pas annoncée pendant 3 minutes, timeout (le coût devient ∞).

11.3.2 IGRP (Interior Gateway Routing Protocol)

Protocole propriétaire de CISCO, avec une métrique qui estime le délai global. Plusieurs routes de coût identique sont maintenues afin d'équilibrer la charge. Pas de limite de 15, mais le nombre de routeurs est inclus dans les messages. Boadcast toutes les 90 secondes.

11.4 Routage dépendant de la charge

Le plus court chemin ne produit pas toujours un routage qui maximise le flot total.

Solution: prendre comme coût le délai: si la charge est élevée sur un lien, alors le coût est plus élevé et le lien sera moins utilisé.

Cependant, l'ajout d'un nouveau lien peut diminuer le débit total (paradoxe de Braess): le routage pour avoir les délais les plus courts n'est pas non plus un optimum global.

Routage optimal. Changer l'objectif du routage: minimiser le délai total sujet à des contraintes de flots. La solution optimale dépend de tous les flots. Algorithme distribué proche de l'algorithme de contrôle de congestion dans TCP [BertsekasGallager92].

11.5 Conclusion

L'algorithme à vecteur de distance est intelligent: il est totalement distribué, peu d'information doit être stockée, et c'est simple. Ainsi, il est largement déployé.

Cependant, la convergence est relativement lente, il n'est donc pas adapté à des réseaux grands et complexes. Dans ce cas, on utilise plutôt des protocoles à états de liens.

12 La couche 4-transport

Couche qui assure un transport de bout en bout (de type logique), sans se soucier des routeurs intermédiaires (qui ne gèrent que les couches 1-2-3). Socket: interface de connection entre l'application et la couche transport (cf TDs).

12.1 Gestion des erreurs

Où gérer les erreurs? Toute couche de niveau supérieur à 2 doit offrir un service sans erreurs à la couche du dessus: les paquets sont livrés sans erreurs ou supprimés.

Exemple de la couche MAC: comment savoir si une trame Ethernet a des erreurs? Que faire d'une trame fautive? Et en Wifi?

Les erreurs sont donc (en général) transformées en pertes de paquets. Autres causes de pertes de paquets: débordement de buffers dans les ponts et routeurs, TTL qui expire (oscillation de routes des algorithmes de routage), ... Il faut réparer ces pertes de paquets.

Deux stratégies possibles:

- De bout en bout (*end-to-end*): A envoie ses paquets à B, et B redemande les paquets manquants. Permet de garder la simplicité: les routeurs ne voient pas forcément passer tous les paquets de A vers B (routes différentes), et c'est coûteux de conserver tous les paquets en mémoire au cas où il faudrait les retransmettre.

- A chaque saut (*hop-by-hop*): Chaque routeur fait ce travail (et demande les paquets manquants). Egalement des avantages: si le taux de pertes de paquets est constant, $p \in [0, 1]$, alors un canal avec un débit R a une capacité réelle de $R(1-p)$ paquets/secondes.

  BER: *bit error rate*. Si les erreurs de bits sont indépendantes, alors le taux de pertes de paquets, PLR (*packet loss rate*) est $PLR = 1 - (1 - BER)^L$, où L est la longueur du paquet (en bits).

Avec k liens qui ont un taux de pertes de p, chaque saut à une capacité de $R(1-p)$, soit $C_{hh} = R(1-p)$.

Si l'on considère une correction de bout en bout, le taux d'erreurs sur les paquets est $p' = 1 - (1-p)^k$ (un paquet arrive si et seulement si il ne s'est perdu sur aucun lien), d'où une capacité $C_{ee} = R(1-p)^k$ au lieu de $R(1-p)$.

Exemple:

k	PLR	C_{ee}	C_{hh}
10	0.05	$0.6R$	$0.95R$
10	0.0001	$0.9990R$	$0.9999R$

Ainsi, si le PLR est élevé, la correction de bout en bout n'est pas acceptable. Comment faire pour concilier les deux?

Faire à chaque saut sur les liens à fort taux d'erreur (Wifi mais pas Ethernet): récupérer des erreurs au niveau de la couche MAC, afin d'offrir un faible taux d'erreurs aux couches supérieures. Et en plus, récupération de bout en bout au niveau des hôtes.

12.2 Protocoles ARQ

Comment réparer les pertes de paquets? A-t-on déjà vu un tel protocole?

Protocole stop-and-go: rajouter éventuellement des numéros de séquence aux paquets, l'hôte destinataire envoie un ACK pour chaque paquet. Si pas d'ACK au bout d'un certain temps, l'hôte source retransmet le paquet.

Stop-and-go est un exemple de protocole de retransmission de paquets, c'est la méthode utilisée dans Internet pour réparer les pertes de paquets. On parle de protocoles ARQ (*Automatic Repeat reQuest*). TCP est un protocole ARQ: récupère les paquets perdus et transmet les paquets dans l'ordre.

Quelle est la limite de Stop-and-go? Si le produit bande-passante délai n'est pas très petit, alors le débit est faible, car on perd du temps en attendant un ACK. Exemple: $t = 8\mu sec/paquet$, temps d'aller-retour de 30 ms $\rightarrow$ 1 paquet toutes les 30.008 ms. Le taux d'utilisation du lien est $U = 0.008/30.008 = 0.00027$. Ainsi, sur un lien à 1 Gb/s, on n'utilise que 267 kb/s.

Solution: pipeliner les envois en permettant plusieurs envois avant de recevoir les ACKs. Nécessite un grand buffer au niveau du destinataire, mais on contrôle cette taille de buffer en fixant le nombre d'envois simultanés, on parle de *fenêtre coulissante* (qui a une taille fixée).

Voir exemples de protocoles ARQ:

1. Fenêtre coulissante. Paquets numérotés, taille de fenêtre W (jamais plus de W paquets sans ACK côté destinataire). Fenêtre offerte: paquets envoyés mais non ACK, ou qui peuvent être envoyés. Fenêtre utilisable: paquets que l'on peut envoyer pour la première fois. La fenêtre se déplace vers la droite (coulisse) lorsque l'ACK pour le premier paquet de la fenêtre est reçu.

   S'il n'y a pas de pertes, débit égal au débit du lien si la taille de la fenêtre est telle que $W \geq \beta/L$, où β est le produit bande-passante délai, et L la taille des paquets. Taille de fenêtre min (en termes de bits) égale au produit bande-passante délai.

2. Répétition sélective. ACK positifs qui indiquent les paquets reçus. Détection des pertes au niveau de la source: timeout si pas d'ACK reçu. Lors d'une perte, on retransmet le paquet qui était perdu.

3. Go-back-N. ACK positifs et cumulatifs: si ACK i, alors tous les paquets de numéro inférieur à i ont été correctement reçus. Détection des pertes au niveau de la source par timeout, et retransmission en partant du dernier paquet pour lequel on avait reçu un ACK. Moins efficace que la répétition sélective, mais plus simple, et le buffer destination n'a besoin que d'une taille 1.

4. Go-back-N avec ACKs négatifs. ACKs positifs cumulatifs, et ACKs négatifs: paquets jusqu'à i ok, mais une perte après cela. Détection de perte par timeout ou par réception d'un ACK négatif. Perte: Go-back-N (retransmettre depuis le denier paquet avec ACK).

Où sont utilisés les protocoles ARQ?

- A chaque saut au niveau de la couche MAC: Répétition sélective sur les modems, et stop-and-go en Wifi.
- De bout en bout: couche transport et TCP: variante de répétition sélective avec un peu de go-back-N; couche application: DNS utilise stop-and-go.

Alternatives à ARQ? On peut utiliser des codes pour réparer les paquets. Si la donnée est constituée de n paquets, rajouter k paquets redondants, de façon à ce que l'on puisse reconstruire la donnée à partir de n paquets parmi les $k + n$ paquets.

Avantages: pas de retransmission (intéressant si délai important), et c'est bien pour le multicast car des destinations différentes peuvent perdre des paquets différents. Par contre, moins bon débit car on ajoute de la redondance même si ce n'est pas nécessaire. On peut combiner FEC avec ARQ: codage si on pert un paquet.

12.3 Contrôle de flot

Différences de performance entre machines: si A envoie ses données trop rapidement à B, B ne peut pas les consommer assez rapidement. On peut alors avoir des pertes de paquets si le buffer de B déborde.

Données en provenance de la couche IP $\rightarrow$ buffer (place libre et données TCP) $\rightarrow$ appli

But: empêcher les débordements de buffer du récepteur. Différent du contrôle de congestion (au niveau IP) qui cherche à éviter les pertes de paquets dans le réseau.

Deux techniques principales (voir transparents):

1. Contrôle de flot avec pression retour: le récepteur envoie des messages STOP (pause) ou GO. Après réception d'un STOP, la source arrête de transmettre pendant x msec. C'est très facile à implémenter. Marche bien si le produit bande-passante délai est petit. Utilisé dans les protocoles X-ON / X-OFF et entre les ponts d'un LAN.

2. Utilisation de fenêtre coulissante: le récepteur n'envoie un ACK des paquets que lorsqu'ils ont été consommés par l'application, lorsqu'il a la place d'en recevoir des nouveaux. Permet d'éviter les débordements de buffers, mais par contre la source peut être amenée à renvoyer des paquets sans ACK alors qu'ils avaient déjà été reçus correctement.

3. Méthode des crédits. Le récepteur indique combien de données il est prêt à recevoir. Les crédits (ou fenêtre annoncée) sont envoyés en même temps que les ACKs, qui sont cumulatifs. Ils font bouger la frontière droite de la fenêtre (les ACKs bougent la frontière gauche): ACK n et crédit k: le récepteur a assez de place pour les paquets $n + 1$ à $n + k$. A priori, $n + k$ ne doit pas décroître (des paquets ont pu être envoyés avant que le crédit soit reçu). La source est bloquée si le crédit est 0 (ou égal au nombre de paquets sans ACKs).

   Les crédits sont directement liés au nombre de places disponibles dans le buffer. En cas de perte, le crédit est toujours égal au nombre de places dans le buffer moins le nombre de paquets reçus dans l'ordre.

12.4 La couche transport: UDP vs TCP

Couches physique + liaison + réseau: transportent les paquets de bout en bout. La couche transport, uniquement au niveau des hôtes, rend les services réseaux disponibles aux applications. Deux protocoles de transport: UDP (User Datagram Protocol) qui est uniquement une interface de programmation pour accéder aux services réseau, et TCP (Transmission Control Protocol), qui gère les erreurs et le contrôle de flot.

12.4.1 Le protocole UDP

Utile pour les applications qui n'ont pas besoin de retransmission. Exemples:

- Applications de téléphonie: c'est le délai qui est important, et la retransmission de paquets perdus fait augmenter le délai!
- Application qui n'envoie qu'un seul message (DNS par exemple pour la résolution de noms): plutôt que de payer le surcoût de TCP (plusieurs paquets avant d'envoyer les données), utilisation de stop-and-go au niveau couche application.
- Multicast, non supporté par TCP.

UDP utilise des numéros de port: l'entête contient les numéros de port source et destination, les adresses IP, la longueur du paquet, et un checksum. Envoi du message entier dans un paquet UDP, jusqu'à 8K. Pas de garantie de séquence sur la réception des messages, et des messages peuvent se perdre. MTU: Maximum Transmission Unit. Si un message est trop gros, il est fragmenté au niveau de la couche IP.

Bibliothèque de sockets, liées aux numéros de ports. Pas de connexion: une fois la socket créée, le serveur peut recevoir des messages sur son port, et le client envoie des messages.

12.4.2 Le protocole TCP

Garantit que les données sont délivrées dans l'ordre, sans pertes, sauf si la connection est coupée. Marche pour toute application qui envoie des données, mais pas avec le multicast. En plus, contrôle de flot et contrôle de congestion (pas détaillé ici).

Protocole avec connection: les 2 parties doivent se synchroniser (synchroniser les numéros de séquence) avant de commencer à échanger des données. ARQ pour les pertes, et crédits pour le contrôle de flot. La connection est fermée lorsque la communication est terminée. Bibliothèque de sockets avec connection.

Utilisation de numéros de ports comme UDP. Entête TCP + Données TCP = segment TCP (= données IP), auquel la couche IP rajoute un entête. Les octets à envoyer sont stockés dans un buffer en attendant que TCP décide de créer un segment. MSS: maximum segment size, 536 octets par défaut (paquets IP de taille 576). Numéros de séquence basés sur le numéro d'octet, et non sur le numéro du paquet. Les segments ne sont jamais fragmentés au niveau de la source.

Protocole ARQ: hybride entre go-back-N et répétition sélective: fenêtre coulissante, ACKs cumulatifs, détection de pertes par timeout au niveau de la source. En plus, retransmission rapide et ACKs sélectifs.

- Opération de base (voir transparent): communication bi-directionnelle. Notation:

  firstByte:lastByte+1(SegmentDataLength) ack AckNb+1 win OfferedWindowSize

  Ligne 8: expiration du timer, retransmission. Le récepteur a gardé les paquets ultérieurs, et peut donc faire à réception un ACK de 10001. Numéro de séquence initial choisi aléatoirement entre 0 et $2^{32} - 1$. Les timers sont réinitialisés après timeout (implémentation TCP SACK).

- Retransmission rapide (voir transparent): mécanisme pour découvrir les pertes avant le timeout (souvent trop grand): si 3 duplicatas d'un ACK sont reçus avant le timeout, alors retransmettre. Quel ACK est envoyé sur la figure?

- ACKs sélectifs (SACKs): mécanisme plus adapté en cas de multiples pertes (retransmission rapide bien pour pertes isolées), qui envoie des ACKs précisant exactement les octets reçus (ACKs positifs d'un interval d'octets). La source peut détecter un trou et retransmettre les octets manquants.

La source peut envoyer un octet même avec un crédit de 0, et tester ainsi la fenêtre du récepteur, afin d'éviter les interblocages en cas de perte de crédit.

Mécanismes additionnels.

- Quand envoyer un ACK? Peut être envoyé immédiatement, ou attendre d'avoir des données à envoyer ou de recevoir plus de segments (ACKs cumulatifs). Retarder l'envoi des ACKs permet de diminuer les surcoûts et le nombre de paquets IP, mais retarde les temps de réaction.

  Algorithme: retarder d'au plus 0.5 sec l'envoi d'un ACK. Segments complets: au moins un ACK par segment. Envoi d'un ACK même si le segment reçu n'est pas dans l'ordre (ACK du dernier octet reçu dans l'ordre +1).

- Algorithme de Nagle: grouper des petites données en segments, afin d'éviter de transmettre plein de petits segments. Par contre, rajoute du délai! Cet algorithme décide quand créer un segment et le transmettre via la couche IP. Principe: accepte un seul segment plus petit que MSS sans ACKs.

  Algorithme: lors de nouvelle donnée qui arrive de la couche supérieure ou à réception d'un ACK, si un segment plein est prêt, l'envoyer. Sinon, s'il n'y a pas de données sans ACK, alors envoyer un segment. Sinon, démarrer un timer et attendre; à expiration du timer, créer un segment et l'envoyer.

  L'algorithme peut être désactivé par l'application.

  Exemple (voir transparent) avec des données qui arrivent une par une...

- Silly Window Syndrome avoidance. Ce syndrome SWS apparaît quand le récepteur est lent ou occupé: si la source a beaucoup de données à envoyer, elle se retrouve à l'envoyer en plein de petits paquets si les fenêtres annoncées sont petites, ce qui est une perte de ressources (exemple à gauche).

  Solution: empêcher le récepteur d'envoyer des ACK avec des petites fenêtres.

ReceiveBuffer: contient octets entre PlusGrandLu et ProchainAttendu, puis la fenêtre offerte, puis une réserve. Une nouvelle annonce est faite uniquement quand la réserve est de taille supérieure à min(MSS, 1/2 ReceiveBuffer). Voir exemple sur les transparents, où l'on voit aussi un envoi suite à une expiration du timer pour vérifier que le récepteur ne veut pas de données.

- SWS avoidance au niveau de la source = Nagle: Nagle + SWS avoidance veulent tous les deux éviter l'envoi de petits paquets (côté source ou côté récepteur).

12.5 Conclusion

TCP: fournit un service fiable au programmeur. Complexe et complexe à utiliser, mais puissant: marche bien avec tout type d'applications. Implémente aussi le contrôle de congestion, mais nous n'aurons pas le temps d'en parler cette année!