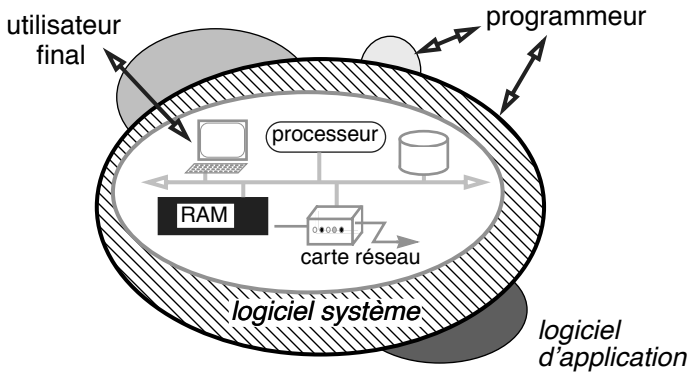
Troisième partie

Processus

Le SX fournit une interface d'accès aux ressources matérielles pour un ensemble de traitements indépendants (processus)



⇒ gérer la progression des processus suivant la disponibilité des ressources
≙ *ordonnancement*

Contenu de cette partie

Processus

- représentation selon différents points de vue : utilisateur, programmeur, SX
- interfaces système
- mise en œuvre par le SX
- politiques d'ordonnancement
- *aperçu : architecture d'une application parallèle*

Plan

Notion de processus

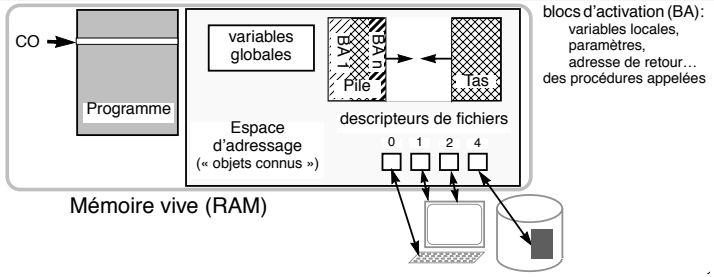
processus $\triangleq$ activité d'exécution d'un programme par un processeur

Un processus n'est pas un programme

- Analogie :
  - *livre* $\sim$ programme (statique) ;
  - (activité de) *lecture* d'un livre $\sim$ processus (dynamique)
- 2 exécutions d'un même programme = 2 processus différents (chaque processus travaille sur ses propres données)
  Exemple : traitement de texte

Point de vue du programmeur

processus $\triangleq$ activité d'exécution d'un programme par un processeur



- Etat de la machine = état du processeur (registres) + état de la mémoire (données)
- Exécution d'une instruction (action) $\leftrightarrow$ changement d'état de la machine (une instruction est définie par la relation qu'elle établit entre états initiaux et états finaux)
- Exécution d'un programme = exécution d'une suite

Point de vue SX

processus = utilisateur de ressources

- le SX doit assurer la disponibilité des ressources demandées par les processus
- l'utilisateur/programmeur doit disposer d'une interface pour demander/restituer les ressources nécessaires à l'exécution de ses processus

$\rightarrow$ l'activité d'un processus est abstraite (réduite) aux opérations d'allocation/restitution de ressources $\rightarrow$ pour le SX :

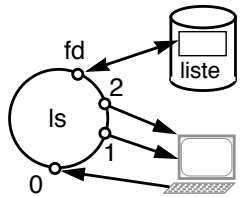
état d'un processus = état (demandeur/utilisateur) par rapport à l'allocation des différentes ressources

Modèle fourni par le SX

Contrôle des ressources utilisées par un processus : environnement d'exécution

- processeur et mémoire vive sont gérés entièrement par le SX
  $\rightarrow$ transparent pour l'utilisateur/le programmeur
- l'utilisation des ressources périphériques est souvent demandée explicitement, via une interface spécifique

  Remarque
  Unix propose une interface unifiée (flots (fichiers)) pour les échanges de données entre un processus et son environnement



- les variables d'environnement fournissent un moyen général et souple pour contrôler et échanger les informations et les données relatives aux ressources disponibles lors de l'exécution :
  - identifiant de l'utilisateur courant, de la machine,
  - protocole utilisé par le terminal,
  - chemins d'accès aux bibliothèques et exécutables…

Interface programmatique (API) de gestion des processus

Créer un processus

Windows

```
BOOL CreateProcess (
 LPCTSTR lpApplicationName,    // programme exécutable
 LPTSTR lpCommandLine,         // ...ou ligne de commande
 LPSECURITY_ATTRIBUTES lpProcessAttributes
 LPSECURITY_ATTRIBUTES lpThreadAttributes
 BOOL bInheritHandles,         // indicateurs d'héritage
 DWORD dwCreationFlags,        // priorité, nouvelle fenêtre...
 LPVOID lpEnvironment,         // → environnement
 LPCTSTR lpCurrentDirectory,
 LPSTARTUPINFO lpStartupInfo, // fenêtre, redirections
 LPPROCESS_INFORMATION lpProcessInformation // résultat
);

typedef struct _PROCESS_INFORMATION {
  HANDLE hProcess;
  HANDLE hThread;
  DWORD dwProcessId;
  DWORD dwThreadId;
} PROCESS_INFORMATION, * LPPROCESS_INFORMATION;
```

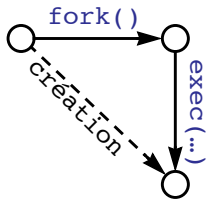
API processus

Créer un processus

Unix

Décomposition en 2 primitives

- Création d'un processus fils : `fork()`
  - Hérite de l'environnement construit par le processus père
  - Exécute le même programme que le père
- Commutation de programme : `exec(...)`
  Le fils charge un nouveau programme à exécuter



Exemple

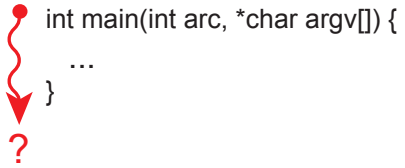
```
... /* code exécuté par le père (seul) */
if (fork()==0) {
     /* code exécuté par le fils */
     exec("prog_fils",...);
} else {
     /* code exécuté par le père */
}
```

API processus

Terminer un processus

- *Windows* : `VOID ExitProcess(UINT uExitCode) //code de retour`
- *Unix* : `VOID exit(int ret) // code de retour`

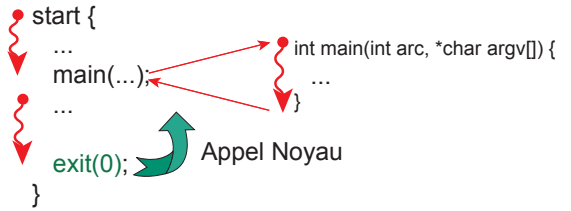
Comment le SX garantit-il la terminaison «propre» des processus ?



API processus

Réponse : interposition (enveloppe)

appel d'une procédure enveloppant le programme principal
et se terminant systématiquement par un appel à `exit(...)`



Autres opérations

- lister les informations de gestion d'un processus :
  ressources utilisées, identifiant, programme, utilisateur...
- suspendre/reprendre un processus :
  masquées dans d'autres primitives : `wait()`, `sleep()`, `read()`...

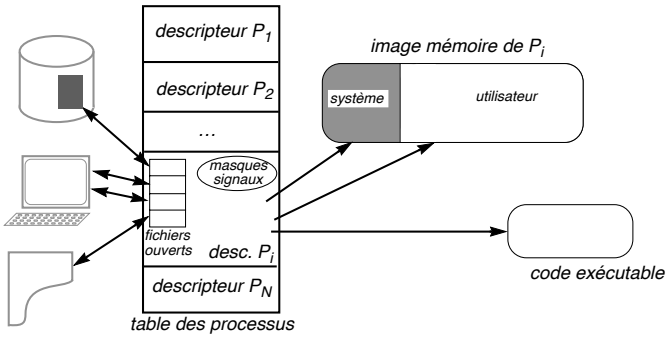
Plan

Représentation des processus

Point de vue SX

processus = utilisateur de ressources
→ état d'un processus = état d'allocation des ressources

- pour chaque processus, les informations d'allocation des ressources (obtenues/demandées) sont conservées dans un descripteur de processus (ou PCB : Process Control Block)
- la table des processus regroupe les différents descripteurs de processus

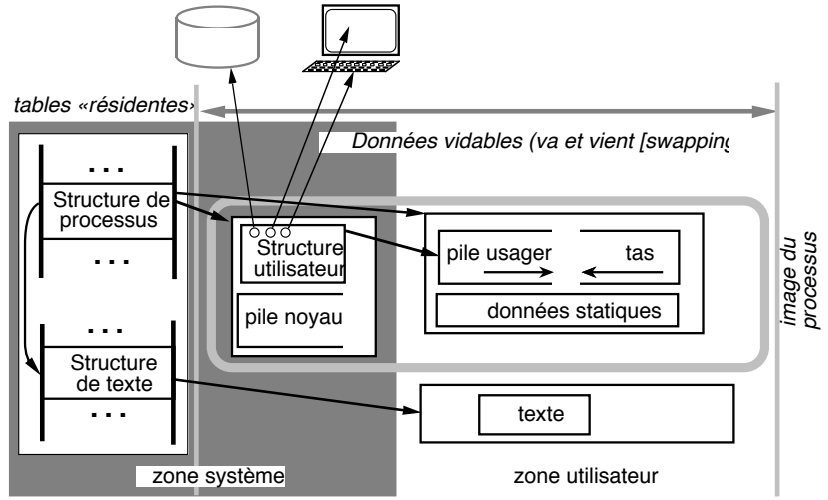


Descripteur de processus

- identifiant du processus
- copie du contexte processeur (à la dernière commutation) :
  - mot d'état programme (CO,CC...),
  - registres (généraux,adresses piles et segments...)
- informations de gestion des ressources
  - UC : état du processus (prêt, bloqué...), priorité... mémoire : adresse de la table des pages (zones allouées, droits d'accès...)
  - E/S : périphériques accessibles (descripteurs de fichiers ouverts : droits d'accès, tampons...)
  - statistiques d'utilisation (utilisé pour les algorithmes d'allocation des ressources)
- liens vers les processus créateur (père)/créés(fils)

Le contexte du processus comprend le descripteur de processus et les structures qu'il référence (tables, piles...)

Exemple (Unix)



Descripteur de processus = Structure utilisateur + Structure de processus

Mise en œuvre des opérations sur les processus

Création

- Création du descripteur de processus initialisé
  - à partir des paramètres d'appel pour Windows
  - à partir du descripteur de processus père pour Unix
- Le processus est prêt ou suspendu

Destruction/Terminaison

- Libération des ressources utilisées par le processus
- Libération du descripteur de processus

Plan

Ordonnancement des processus

But : gérer l'allocation des ressources aux processus

Situation

Certaines ressources (UC, imprimante...)
n'admettent qu'un nombre limité d'utilisateurs simultanés

→ une file est associée à chaque ressource, qui
contient les (descripteurs des) processus en attente de la ressource
Par la suite, pour être concis, « mettre un processus en attente » sera utilisé pour : « intégrer le descripteur d'un processus à une file d'attente ».

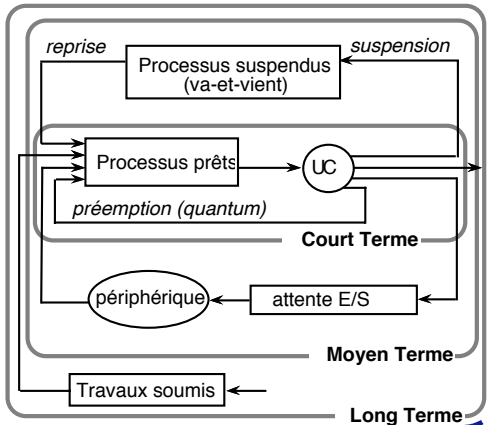
→ lorsque la ressource est disponible,
le choix du prochain processus auquel allouer la ressource dépend
- de la nature de la ressource
- de critères définissant la politique d'allocation appliquée
  - *priorités* : privilégier certains processus pour l'accès à la ressource
  - *équité* : garantir un accès à tout processus
  - *contraintes de temps* : garantir une borne sur le temps d'attente
  - ...

La vie des processus : terminologie (1/2)

Niveaux d'ordonnancement

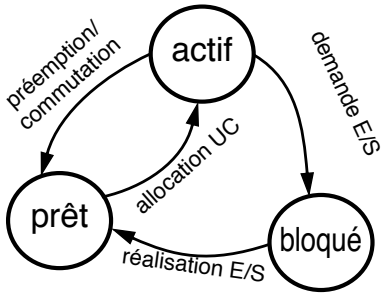
Définis selon la fréquence des décisions d'allocation :

- ***à long terme*** (régulation) :
  choix des programmes à lancer
  (cf traitement par lots)
- ***à moyen terme*** (synchronisation) :
  choix des processus prêts
  (coordination, régulation par
  va-et-vient, attente d'E/S...)
- ***à court terme*** (exécution) :
  choix du processus actif
  (allocation du processeur (UC))



La vie des processus : terminologie (2/2)

Etats d'un processus (pour l'ordonnancement)



Allocation du processeur : politiques d'ordonnancement court terme

Caractéristiques des algorithmes

- priorité associée aux processus, ou non
  - pas de priorité → service équitable (tout processus finit par être servi) et gestion simple
  - priorités
    - contrôle fin de l'allocation
    - risque de famine pour les processus non prioritaires
      remède : faire croître la priorité avec le temps passé (ancienneté)
- réquisition (préemption), ou non
  - pas de réquisition : simple, pas d'hypothèses sur le matériel
  - réquisition
    → possibilité de garantir des temps de réponse
    → nécessaire aux systèmes interactifs et/ou temps-réel

Remarque : pas d'algorithme convenable pour tous les critères...

Premier arrivé, premier servi (politique FIFO)

Principe : servir les requêtes par ordre chronologique d'arrivée

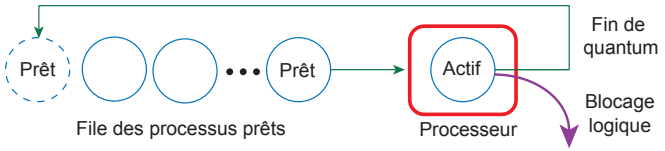
- simple
- sans priorités (équitable)
- sans réquisition
  → forte variance du temps de service : le temps de service d'un processus dépend du comportement des processus qui le précèdent
  → inadapté au temps partagé

Tourniquet (Round-Robin)

Principe

quantum $\triangleq$ durée maxi. d'activité continue pour le processus actif (élu)

- les processus prêts sont rangés dans une file
- le processeur est alloué au processus en tête de file
- le processeur est réquisitionné (préempté)
  au profit du processus suivant dans la file
  - soit en fin de quantum ;
  - soit sur blocage du processus actif (attente d'une synchronisation, d'une E/S...)



- ≈ FIFO, avec réquisition → adapté au temps partagé
- Paramètre essentiel : valeur du quantum (usuel : 10-100 ms)

Allocation à deux niveaux

Principe

- Les processus prêts sont divisés en catégories (processus interactifs, tâches de fond...)
- Chaque catégorie a sa politique d'allocation (FIFO, tourniquet...)
- Une politique d'allocation est définie entre catégories (FIFO, tourniquet, priorités...)

Exemples

- Files multiniveaux
  - gestion interne aux catégories : FIFO
  - ordonnancement entre catégories : priorités statiques
- FSS (Fair Share Scheduling)
  - Chaque catégorie reçoit un nombre de quantums
  - Le nombre de quantums alloués peut être ajusté dynamiquement
- Tourniquet multiniveaux
  - gestion interne aux catégories : tourniquet
  - ordonnancement entre catégories : priorités statiques
  - adaptatif : un processus peut changer de niveau selon qu'il épuise son quantum ou non

Ordonnancement temps-réel (1/2)

Objectif *(temps réel « dur »)* : garantir aux traitements le respect de contraintes de dates « précises » de terminaison (échéances)

Modèle : Un processus ordonnancé est vu comme exécutant une série de tranches de calcul successives (*tâches*)).
Les tâches exécutées par un processus sont caractérisées par

- leur date d'arrivée
- leur (pire) temps d'exécution
- leur échéance (date de fin d'exécution au plus tard)

→ *contrôle d'admission* : l'ordonnanceur peut rejeter une tâche a priori, si son exécution compromet le respect des contraintes de date

Cas courant : tâches périodiques

Les tâches exécutées par chaque processus arrivent à intervalles réguliers.
Elles peuvent alors être caractérisées par

- leur période p (ou leur fréquence : $1/p$)
- leur (pire) temps d'exécution t (avec $t \leq p$)

→ calculs d'ordonnancement simplifiés

Ordonnancement temps-réel (2/2)

Stratégies de base

Ingrédients courants : priorités + préemption

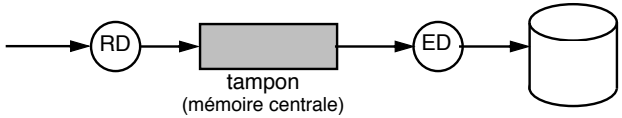
- *ordonnancement à taux monotone* (RMS : Rate Monotonic Scheduling)
  - tâches périodiques
  - priorité fixe : ordre inverse des périodes
  - simple, prévisible (contrôle d'admission, impact d'une surcharge)
- *échéance la plus proche* (EDF : Earliest Deadline First)
  - priorité dynamique : échéance la plus proche
  - optimal
  - complexe (priorités dynamiques) : calcul des priorités, contrôle d'admission, impact d'une surcharge

Plan

Démarche de conception d'une application parallèle

1. définir des *activités élémentaires*, exécutées en parallèle
2. composer (*coordonner*) ces activités élémentaires

Exemple 1 : enregistrement d'un flux de données reçu à haut débit



La réception et l'enregistrement des données doivent être parallèles
→ 2 activités : RD (réception données) et ED (écriture données), qui doivent

- être aussi autonomes que possible
- échanger des données

→ utilisation d'un tampon en mémoire centrale

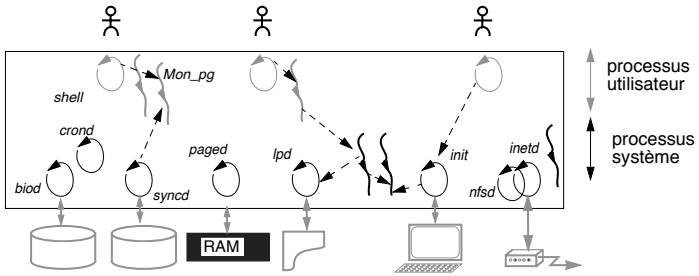
ED et RD doivent se coordonner pour accéder au tampon

- ED doit attendre RD si le tampon est vide
- RD doit attendre ED si le tampon est plein

⇒ primitives pour communiquer (tampons...) et synchroniser (attente...)

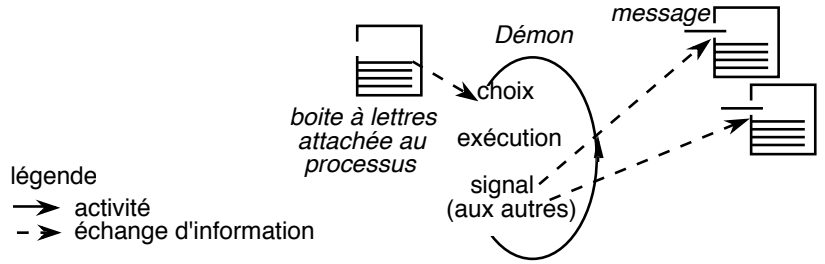
Exemple 2 : structuration « classique » de l'activité d'un SX

Structuration type



- un processus par activité utilisateur
- un processus (démon) par périphérique
- un ou plusieurs processus (démons) pour la gestion mémoire
- les services système peuvent être réalisés par des processus ou bien comme des procédures

Structure de base de l'activité d'un SX : démons (processus cycliques)



Caractère permanent → { simple(conception) ; efficacité de mise en œuvre }

→ utilisé dans les couches basses (SX)

Mécanismes d'interaction pour les processus

Communication

- Synchrone : lecture/écriture vers des flots/fichiers
  → prochain cours + TD (Unix)
- Couplée : tampons
  → tubes (pipes) + TD (Unix/Shell)
- Asynchrone :
  - communication par événements, schéma s'abonner/publier
    → service d'interruptions logicielles (signaux UNIX → TD)
  - E/S non bloquantes (→ TD Unix)
  - mémoire partagée (→ Cours + TD mémoire virtuelle)

Synchronisation (attente contrôlée par le programmeur)

- couplage induit par l'accès aux tubes
- schéma fork/join : possibilité pour un processus père d'attendre la terminaison d'un fils (→ TD)