

I403A Systèmes d'exploitation

Séance 2

Processus et thread

Sébastien Combéfis

2018–2019



Ce(tte) œuvre est mise à disposition selon les termes de la Licence Creative Commons Attribution – Pas d'Utilisation Commerciale – Pas de Modification 4.0 International.

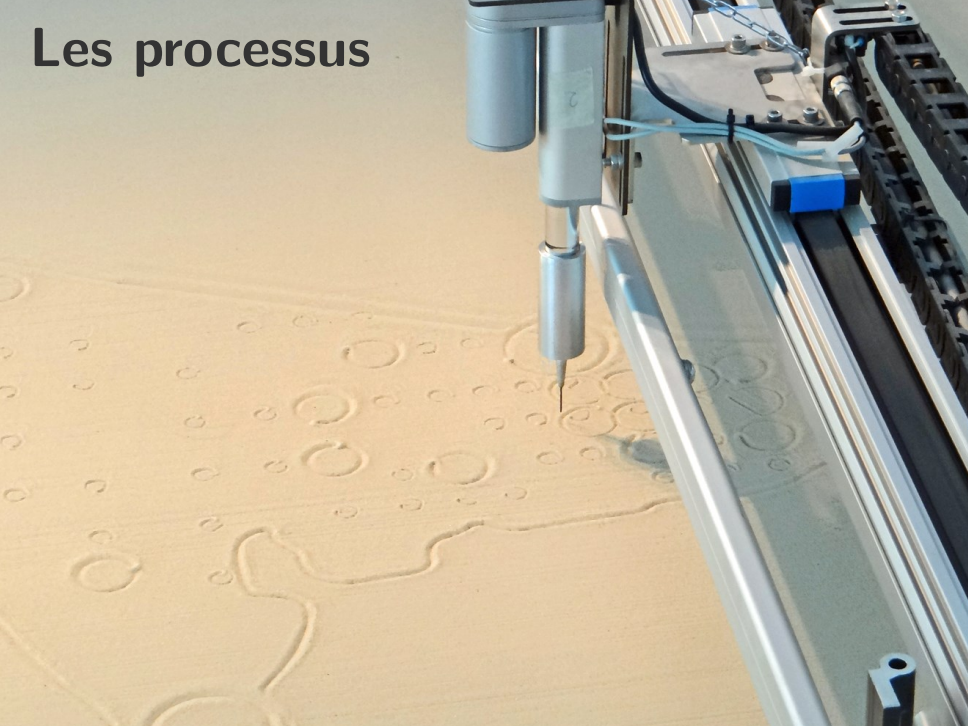
Rappels

- Définition de la notion de **système d'exploitation**
 - Système informatique et origines des OS
 - Caractéristiques d'un OS
 - Fonctions et opérations faites par l'OS
- **Services et structure** des OS
 - Mode noyau et mode utilisateur
 - Fonctionnement des appels systèmes
 - Différentes structures possibles d'un OS et exemples réels

Objectifs

- Décrire et comprendre les processus
 - Création, exécution et terminaison
 - États d'un processus
 - Structure en mémoire
- Décrire et comprendre les threads
 - Processus VS thread
 - Types de threads
 - Modèles de multi-threading

Les processus



Processus (1)

- Un système d'exploitation peut exécuter **plusieurs programmes**
 - Partage des ressources : CPU, mémoire, périphérique E/S...
 - Compartimentation des programmes
- Un **processus** est un programme en cours d'exécution
 - **Programme** : passif et sur le disque
 - **Processus** : actif et en mémoire
- Plusieurs **types** de processus

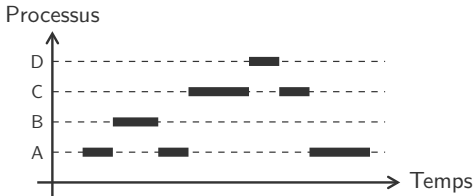
Processus systèmes et processus utilisateurs

Multiprogrammation (1)

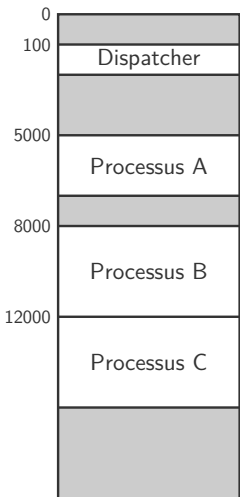
- Plusieurs processus s'exécutent **de manière concurrente**

Le CPU est multiplexé entre les différents processus

- Deux formes de **parallélisme**
 - **Un seul processeur** : pseudo-parallélisme ou concurrence
 - **Multiprocesseur** : (véritable) parallélisme



Multiprogrammation (2)



Exemple d'exécution :

01 5000

02 5001

03 5002

04 5003

05 5004 Requête E/S

06 100

07 101

08 102

09 103

10 8001

11 8002

12 8003 Timeout

13 100

14 101

15 102

16 103

17 5005

18 5006

19 5007

Processus (2)

- Différents types d'**activité CPU**
 - Système en **batch** : exécution de jobs
 - Système en **temps partagé** : programme utilisateur ou tâche
- **Plusieurs programmes** même avec un seul utilisateur

Traitement de texte, Web browser, etc.
- **Système embarqué** avec un seul utilisateur sans multitâches

Un seul programme et activité interne de gestion du système

L'abstraction processus

- Un processus est une **abstraction**

Les logiciels exécutables sont organisés en processus séquentiels

- Chaque processus possède son propre **processeur virtuel**

Il a l'impression d'être le seul à être exécuté sur un processeur

- Pas d'hypothèses de **temporisation**

Un processus peut être arrêté à tout moment par l'OS

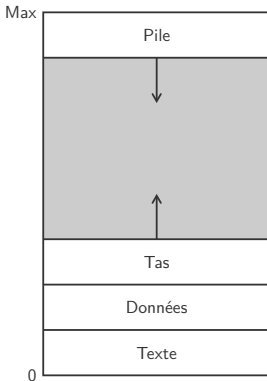
- Plusieurs processus peuvent être lancés d'un même **programme**

Un programme est une entité passive (fichier exécutable)

Structure en mémoire (1)

- Un processus est bien plus que juste le **code du programme**

Il s'agit de la section texte du processus



Structure en mémoire (2)

- Plusieurs **zones en mémoire** pour chaque processus
 - **Zone texte** : code du programme
 - **Pile** : données temporaires (paramètres des fonctions, adresse de retour, variables locales...)
 - **Tas** : mémoire dynamiquement allouée lors de l'exécution
 - **Zone de données** : variables globales
- **Autres données** associées à un processus
 - Compteur ordinal (*Program counter*, PC)
 - Registres du CPU
 - Ressources attribuées : E/S, fichiers, zone mémoire...

Commandes ps

- Statut des processus obtenus avec la commande ps

Possibilité de filtrer les types de processus désirés

```
combefis@MacBook-Pro-de-Sebastien-3 ~ -- zsh -- 114x27
# combefis at MacBook-Pro-de-Sebastien-3.local in ~ [12:05:06]
→ ps -a
  PID TTY          TIME CMD
 40149 ttys000    0:00.14 login -pf combefis /bin/zsh
 40151 ttys000    0:00.38 -zsh
 40425 ttys000    0:00.00 ps -a
13570  ttys001    0:00.01 login -pf combefis /bin/zsh
13571  ttys001    0:00.15 -zsh

# combefis at MacBook-Pro-de-Sebastien-3.local in ~ [12:05:08]
→ ps -aef
  UID    PID  PPID  C  STIME  TTY          TIME CMD
    0      1      0  0  4sep17 ??        11:13.40 /sbin/launchd
    0     11      1  0  4sep17 ??        1:19.00 /usr/libexec/UserEventAgent (System)
    0     12      1  0  4sep17 ??        0:08.31 /usr/libexec/kextd
    0     13      1  0  4sep17 ??        0:32.94 /usr/libexec/taskgated -s
    0     14      1  0  4sep17 ??        1:07.46 /usr/sbin/notifd
    0     15      1  0  4sep17 ??        1:24.33 /usr/sbin/securityd -i -s off
    0     16      1  0  4sep17 ??        0:31.24 /System/Library/CoreServices/powerd.bundle/powerd
    0     17      1  0  4sep17 ??        1:44.41 /usr/libexec/configd
    0     18      1  0  4sep17 ??        0:39.48 /usr/sbin/distnoted daemon
    0     20      1  0  4sep17 ??        0:19.90 /usr/libexec/diskarbitrationd
    0     21      1  0  4sep17 ??        1:55.77 /usr/libexec/opendirectoryd
    0     24      1  0  4sep17 ??        0:02.17 /usr/libexec/warmd
  213    25      1  0  4sep17 ??        0:01.14 /System/Library/PrivateFrameworks/MobileDevice.framework/Versions
/A/Resources/usbmuxd -launchd
    0     26      1  0  4sep17 ??        0:00.87 /usr/libexec/stackshot -t
```

Commandes top

- Affichage d'informations structurées sur les processus avec top

Possibilité de classement selon différents critères

```
combefis — top — top — top — 114x27
Processes: 193 total, 2 running, 1 stuck, 190 sleeping, 1061 threads
Load Avg: 1.47, 1.46, 1.47 CPU usage: 4.62% user, 5.6% sys, 90.30% idle
SharedLibs: 15M resident, 13M data, 0B linkedit.
MemRegions: 80491 total, 2825M resident, 68M private, 683M shared.
PhysMem: 5654M used (1205M wired), 1119M unused.
VM: 495G vsize, 1070M framework vsize, 109061(0) swaptins, 182594(0) swappouts.
Networks: packets: 13794993/146 in, 9167708/2838M out. Disks: 4846034/1016 read, 7267880/608G written.
12:10:21

PID COMMAND NCPU TIME #TH #WQ #PORT #REGS MEM RPRVT PURG CMPRS VPRVT VSIZE PGRP PPID
40536 mdworker 0.0 00:00.03 8 4 62 69 1940K 1084K 0B 0B 56M 2420M 40536 223
40532 top 12.2 00:04.78 1/1 0 23 40 2404K 2176K 0B 0B 45M 2403M 40532 40151
40498 mdworker 0.0 00:00.21 5 1 55 73 11M 9976K 0B 0B 68M 2424M 40498 223
40497 mdworker 0.0 00:00.24 5 1 55 77 12M 11M 0B 0B 64M 2428M 40497 223
40495 mdworker 0.0 00:00.07 5 1 55 63 2408K 1416K 0B 0B 55M 2418M 40495 223
40485 QuickLookUIH 0.0 00:00.14 3 1 120 93 4724K 2460K 0B 0B 46M 2455M 40485 223
40483 QuickLookSat 0.0 00:00.19 2 0 46 64 3280K 2248K 0B 0B 53M 2415M 40483 1
40482 quicklookd 0.0 00:00.18 4 0 87 83 4672K 3280K 0B 0B 64M 2949M 40482 223
40469 cupsd 0.0 00:00.07 3 0 46 48 2544K 2128K 0B 0B 64M 2423M 40469 1
40468 printtool 0.0 00:00.03 2 1 37 42 1364K 928K 0B 0B 45M 2411M 40468 223
40459 com.apple.Co 0.0 00:00.01 2 1 30 47 1048K 464K 0B 0B 37M 2404M 40459 1
40456 ScopedBookma 0.0 00:00.03 2 1 40 50 1732K 1268K 0B 0B 45M 2435M 40456 223
40455 CVMCompiler 0.0 00:01.54 2 1 31 119 35M 35M 0B 0B 67M 2473M 40455 223
40452 com.apple.hi 0.0 00:00.01 2 0 34 46 1084K 492K 0B 0B 45M 2411M 40452 1
40451 CVMCompiler 0.0 00:00.45 2 1 31 76 19M 18M 0B 0B 58M 2443M 40451 223
40450 com.apple.BK 0.0 00:00.03 2 1 48 60 1940K 980K 0B 0B 45M 2435M 40450 1
40449 ubd 0.0 00:00.03 6 0 66 66 2976K 1852K 0B 0B 53M 2421M 40449 223
40448 librariand 0.0 00:00.04 2 0 46 49 2240K 1160K 0B 0B 55M 2444M 40448 223
```

- L'OS doit **gérer** en permanence les processus exécutés

Sur certains systèmes, ils sont tous toujours actifs

- Plusieurs **opérations** faites par l'OS
 - Création et suppression de processus
 - Allocation du CPU (*scheduling*), attribution de ressources
 - Synchronisation et communication
 - Gestion des interblocages (*deadlock*)

Création d'un processus

- Un processus peut être **créé** à différents moments
 - Initialisation du système
 - Appel système de création de processus
 - Requête de l'utilisateur
 - Initialisation d'un travail en traitement par lots
- Un processus possède un **identifiant** (*Process Identifier*, PID)
Identifiant unique pour effectuer des opérations sur ce processus
- Un processus **démon** (*daemon*) est exécuté en arrière plan

Unix : l'appel système fork

- Création d'un nouveau processus avec l'appel système fork

- Le nouveau processus est une copie du processus appelant

Création d'un processus fils, copie exacte du processus parent

- Valeur de retour

- En cas de succès, le PID du fils dans le parent et 0 dans le fils
- En cas d'échec, -1 dans le parent

```
#include <unistd.h>

pid_t fork(void);
```

Copie d'un processus (1)

Processus parent



```
int main()
{
    pid_t pid = fork();


    if (pid == 0)
    {
        printf ("Fils.\n");
    }
    else
    {
        printf ("Parent.\n");
    }

    return 0;
}
```

- Exécution de l'appel système fork qui duplique le processus

Copie d'un processus (2)

Processus parent




```
int main()
{
    pid_t pid = fork();

    if (pid == 0)
    {
        printf ("Fils.\n");
    }
    else
    {
        printf ("Parent.\n");
    }

    return 0;
}
```

Processus fils



```
int main()
{
    pid_t pid = fork();

    if (pid == 0)
    {
        printf ("Fils.\n");
    }
    else
    {
        printf ("Parent.\n");
    }

    return 0;
}
```

- Les deux processus sont identiques, mais ont un **PC différent**

Exemple de fork

```
int main()
{
    pid_t pid = fork();

    if (pid < 0) // Erreur
    {
        printf ("Une erreur s'est produite.\n");
    }
    else if (pid == 0) // Dans le fils
    {
        printf ("Je suis dans le fils.\n");

        execlp ("/bin/echo", "echo", "Hello World!", NULL);
    }
    else // Dans le parent
    {
        printf ("Je suis dans le parent.\n");

        wait (NULL);
        printf ("Mon fils a fini.\n");
    }

    return 0;
}
```

Windows : l'appel système CreateProcess

- L'appel système CreateProcess crée un **nouveau processus**
- Il va aussi **charger un programme** dans le processus créé

```
#include <WinBase.h>
// #include <ProcessThreadapi.h> (Windows 8)

BOOL WINAPI CreateProcess(
    _In_opt_      LPCTSTR lpApplicationName,
    _Inout_opt_  LPTSTR lpCommandLine,
    _In_opt_     LPSECURITY_ATTRIBUTES lpProcessAttributes,
    _In_opt_     LPSECURITY_ATTRIBUTES lpThreadAttributes,
    _In_         BOOL bInheritHandles,
    _In_         DWORD dwCreationFlags,
    _In_opt_     LPVOID lpEnvironment,
    _In_opt_     LPCTSTR lpCurrentDirectory,
    _In_         LPSTARTUPINFO lpStartupInfo,
    _Out_        LPPROCESS_INFORMATION lpProcessInformation
);
```

Terminaison d'un processus

- Un processus peut être **arrêté** de différentes manières
 - Normal volontaire ou volontaire suite à une erreur
 - Involontaire suite à une erreur fatale
 - Par un autre processus
- **Terminaison volontaire** signalée par un appel système
 - `exit` sous Unix et `ExitProcess` sous Windows
 - Code de retour renvoyé au processus parent récupéré avec `wait`
- **Arrêt d'un autre processus** par un appel système
 - `kill` sous Unix et `TerminateProcess` sous Windows*

Terminaison d'un fils

- Un processus parent peut **terminer l'exécution de son fils**
 - Le fils a dépassé la consommation d'une certaine ressource
 - La tâche assignée au fils n'est plus nécessaire
 - Le parent se quitte et le fils ne peut plus exister

- **Terminaison en cascade** de processus

Tous les fils sont terminés les uns après les autres par l'OS

- Processus **zombie** reste en mémoire dans la table des processus

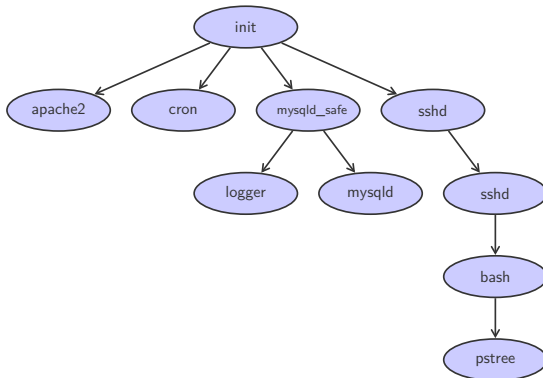
Garder code de retour en attendant que parent appelle `wait`

Hiérarchie des processus

- Un nouveau processus est créé par un **processus parent**

Sous Windows, un parent peut déshériter de ses enfants

- Sous Unix, les processus forment un **groupe de processus**



Processus orphelin

- Processus zombie peut devenir **orphelin**

Le parent n'a pas appelé `wait` et s'est terminé

- Unix assigne les processus orphelins comme **fils de `init`**

Appel régulier de `wait` pour libérer les orphelins

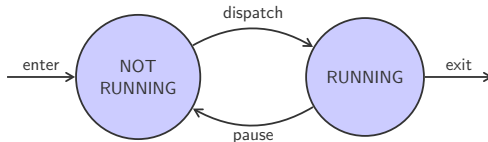
État d'un processus

- Un processus possède un **état** qui change en cours d'exécution
- Le processeur n'exécute qu'**un seul processus** à la fois

Dans le cas d'un simple processeur avec un seul cœur évidemment

- Modèle de base à **deux états**

RUNNING, NOT_RUNNING



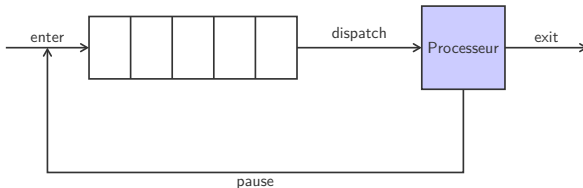
Modèle à deux états

- Processus simplement gérés avec une **file d'attente** (FIFO)

Processus exécutés l'un après l'autre, dans l'ordre d'arrivée

- Un processus se termine car il a **fini ou qu'il a été avorté**

Un processus peut être retiré du CPU pour diverses raisons



Modèle à cinq états (1)

- Cinq états qu'on retrouve dans la plupart des OS

Souvent détaillés plus finement dans les principaux OS modernes

NEW En cours de création

RUNNING Instructions en cours d'exécution

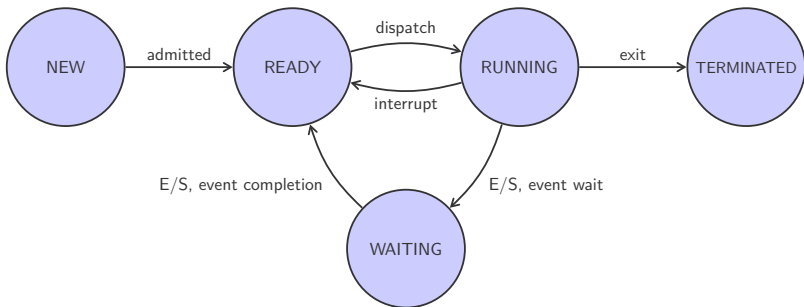
WAITING En attente d'un évènement (E/S, signal...)

READY Prêt et en attente du processeur

TERMINATED Exécution terminée

Modèle à cinq états (2)

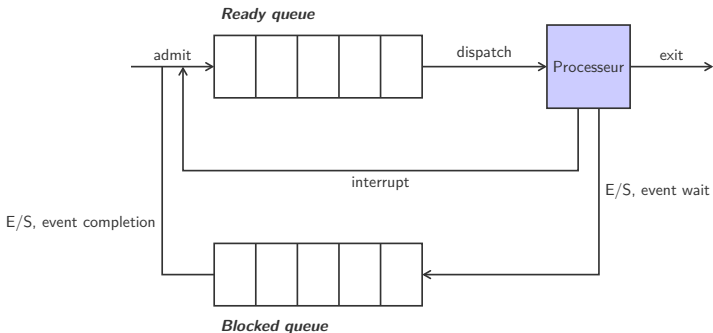
- Plusieurs **transitions** possibles lors d'évènements particuliers
 - Alternance entre READY et RUNNING pour utilisation du CPU
 - Passage par l'état WAITING lors d'une demande E/S, signal...



Modèle à cinq états (3)

- Une nouvelle file d'attente gère les **processus bloqués**
- L'OS parcourt cette file lorsqu'un **évènement se produit**

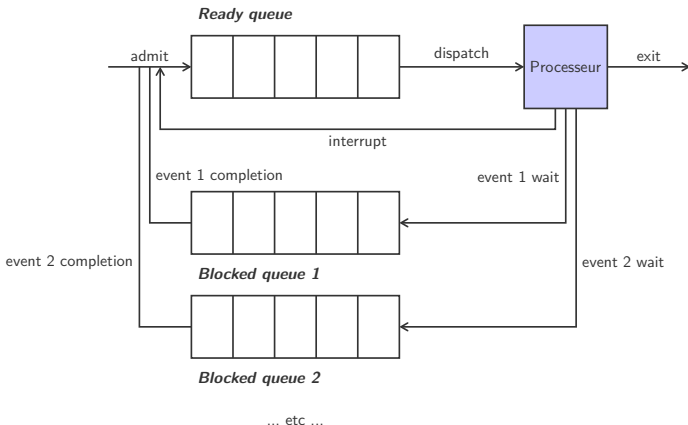
File de type FIFO et en fonction de l'occurrence d'un évènement



Multiples files d'évènements

- Lourdeur lorsque l'OS doit scanner la Blocked Queue

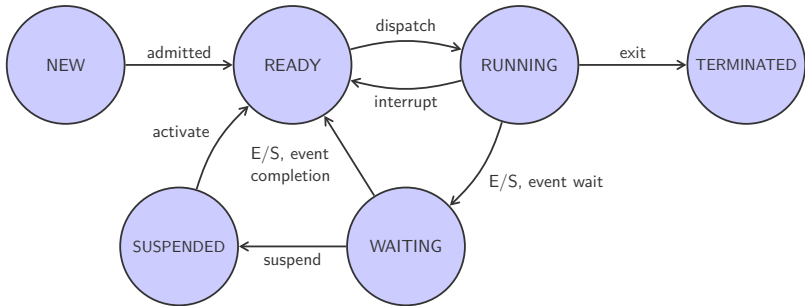
Une file par (type d') évènement



Suspension de processus (1)

- Possibilité de **suspendre un processus** temporairement

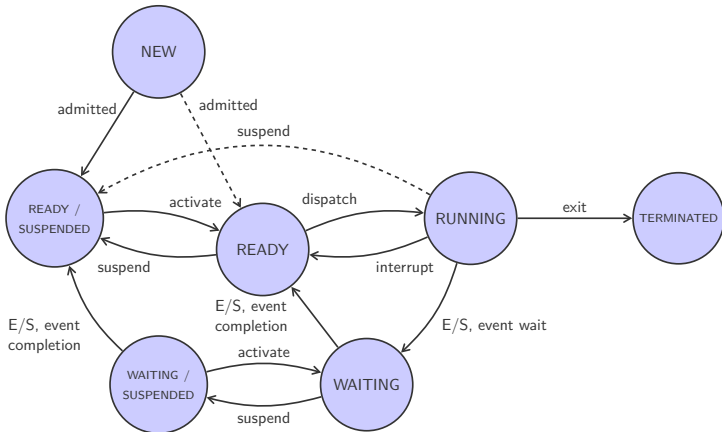
Déplacement depuis la mémoire sur le disque (swapping)



Suspension de processus (2)

- Libérer un processus qui était **en attente** n'est pas idéal

Il faut de nouveaux états pour une meilleure gestion



États des processus sous Linux

- Dans le fichier d'entête `linux/sched.h`

```
/*
 * Task state bitmask. NOTE! These bits are also encoded in
 * fs/proc/array.c: get_task_state().
 *
 * We have two separate sets of flags: task->state is about runnability,
 * while task->exit_state are about the task exiting. Confusing, but this way
 * modifying one set can't modify the other one by mistake.
 */
#define TASK_RUNNING          0
#define TASK_INTERRUPTIBLE    1
#define TASK_UNINTERRUPTIBLE  2
#define __TASK_STOPPED        4
#define __TASK_TRACED         8
/* in tsk->exit_state */
#define EXIT_DEAD             16
#define EXIT_ZOMBIE           32
#define EXIT_TRACE             (EXIT_ZOMBIE | EXIT_DEAD)
/* in tsk->state again */
#define TASK_DEAD              64
#define TASK_WAKEKILL          128
#define TASK_WAKING            256
#define TASK_PARKED            512
#define TASK_NOLOAD            1024
#define TASK_NEW               2048
#define TASK_STATE_MAX        4096
```

Table des processus

- L'OS maintient une **table des processus** en mémoire

Chaque processus possède une entrée dans cette table

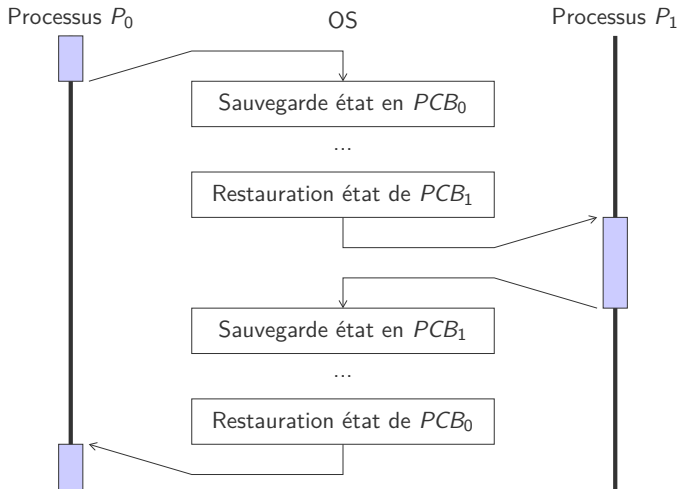
État du processus
Numéro du processus
Compteur ordinal
Registres
Limites en mémoire
Liste des fichiers ouverts
...

Les entrées de la table sont des **blocs de contrôle de processus** (PCB)

Bloc de contrôle de processus

- Stocke les **informations spécifiques** à chaque processus
 - État, identification et compteur ordinal
 - Valeurs des registres du CPU (pour sauvegarde)
 - Allocation de mémoire avec limites pour protection
 - Liste des fichiers ouverts
- On trouve également des informations pour **divers algorithmes**
 - Ordonnancement de processus : priorité, files...
 - Comptabilité : temps CPU utilisé, limites temporelles...
 - Liste des périphériques E/S alloués...

Alternance entre processus



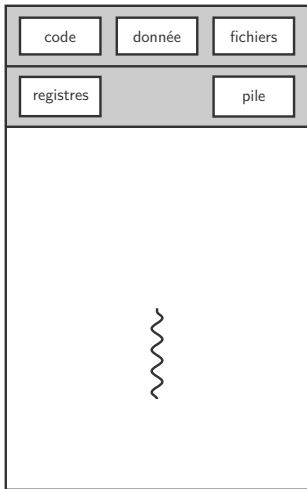
Les threads



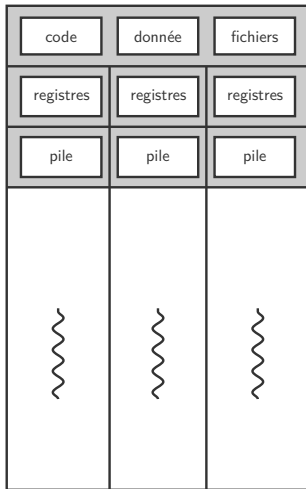
Thread

- Un **thread** est un processus léger
 - « *Un processus dans un processus* »
- Un thread possède une série d'**informations** propres à lui
 - Identifiant, compteur de programme, registres, pile*
- **Partage de ressources** entre threads du même processus
 - Zone code et données, fichiers ouverts et signaux*

Processus mono- et multi-threads



Processus mono-thread



Processus multi-thread

Utilisation des threads

- Distinction travail **avant-plan/arrière-plan**, asynchrone

Éditeur de texte avec sauvegarde régulière

- **Vitesse** d'exécution

Traiter un lot de données, lire le suivant

- **Structure modulaire** de programme

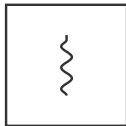
Découpe d'un programme en fonctionnalités

- Fonction à exécuter **plusieurs fois** en parallèle

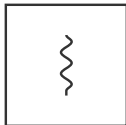
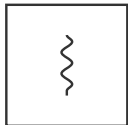
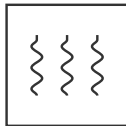
Serveur web par exemple

Threads et processus

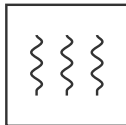
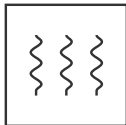
Un processus mono-thread



Un processus multi-thread



Plusieurs processus mono-thread



Plusieurs processus multi-thread

Processus multi-thread

- Une application est implémentée comme **un processus**
 - Par exemple, un web browser prend la forme d'un processus
 - Un processus est composé de **threads de contrôle**
 - Affiche des images et reçoit des informations du réseau
- Peut exploiter les CPU **multi-cœurs**

Calcul intensif, un thread d'exécution par cœur
- Les threads évitent de devoir créer un **nouveau processus**

Effectuer la même tâche peut se faire avec un thread

Avantages du multi-threading

- Augmentation de la **réponse**

Même si un thread bloque, le reste de l'application peut continuer

- **Partage des ressources** du processus

Les différents threads sont dans l'espace d'adressage du processus

- **Économie** lors de la création et du changement de contexte

Plus rapide et moins couteux de créer des threads

- **Montée en charge** sur des architectures multi-processeurs

Les threads peuvent être exécutés sur différents processeurs

Programmation multi-cœur

- Nouvelle façon de penser pour le **programmeur**

Penser son programme pour être parallélisable

- Parallélisme de **données** VS parallélisme de **tâches**

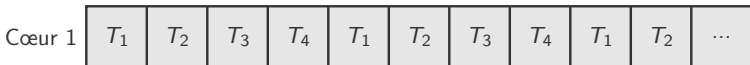
Répartition de données ou exécution de plusieurs tâches

- **Parallélisme** VS **concurrence**

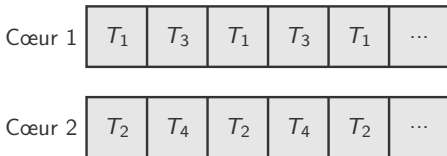
Parallélisme réel ou simulé avec illusion

Concurrence VS parallélisme

■ Architecture **mono-processeur**



■ Architecture **multi-processeur**



Loi d'Amdahl

- **Gain de performance** en améliorant les performances

En ajoutant des unités de calcul au système

- **Formule d'Amdahl**

- S la proportion d'activité à exécuter en série
- N le nombre de cœurs

- **Borne supérieure** pour le gain de performance

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}} \quad \left(\text{on notera que } \lim_{N \rightarrow \infty} = \frac{1}{S} \right)$$

Types de thread

- Le **support pour les threads** peut être fourni à deux niveaux
 - Thread niveau **utilisateur** (ULT)
Gestion intégralement faite par l'application
 - Thread niveau **noyau** (KLT)
Le noyau est multi-threads, ils sont gérés par l'OS
- **Liens** entre les threads utilisateurs et les threads noyaux
Différents modèles de multi-threading

Thread utilisateur

- **Gestion par l'application**, le noyau ne voit pas les threads

Librairies dédiées, par défaut un thread par processus

- **Avantages**

- Changement de contexte plus rapide, en mode utilisateur
- Ordonnancement géré par l'application : flexible
- Indépendants de l'OS

- **Inconvénients**

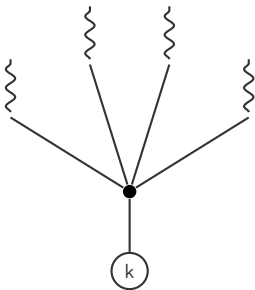
- Un appel système d'un thread bloque tout le processus
- Pas d'utilisation du multiprocesseur

Thread noyau

- Le **noyau est multi-thread**, et l'OS fournit une API
 - Ordonnancement directement géré par le noyau
 - Affectation du CPU fait sur base des threads
- **Avantages**
 - Un thread bloqué ne bloque pas tout le processus
 - Différents threads peuvent être répartis sur plusieurs cœurs
 - Les routines du noyau peuvent être multi-thread
- **Inconvénients**
 - Le changement de contexte nécessite de passer par le noyau

Plusieurs-vers-un

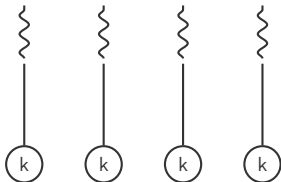
- Plusieurs threads utilisateurs liés au **seul thread noyau**
- La gestion des threads est faite **par l'application**
 - L'application bloque si un seul thread bloque
 - Pas d'exécution parallèle sur multi-cœur



Exemples :

- Solaris Green Threads
- GNU Portable Threads

- Chaque thread utilisateur est lié à **un thread noyau**
- La gestion des threads est faite **par le noyau**
 - Créer un thread devient couteux car il faut un thread noyau
 - Limitation du nombre de threads comme sur Linux et Windows



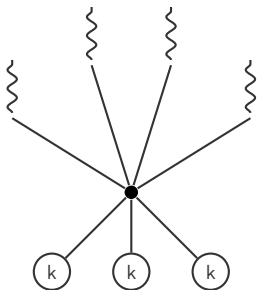
Exemples :

- Windows NT/2000/XP
- OS/2
- Linux
- Solaris 9 et suivants

Plusieurs-vers-plusieurs

- **Multiplexage des threads** utilisateurs sur les threads noyaux
- Permet une meilleure **concurrency**

Plusieurs threads en même temps, moins de blocage



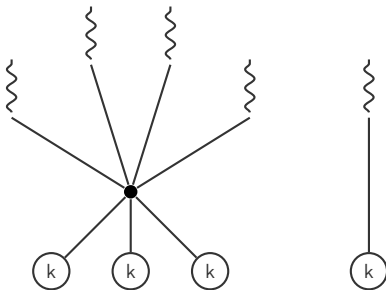
Exemples :

- Solaris avant version 9
- Windows NT/2000 (avec *ThreadFiber*)

Modèle à deux niveaux

- **Combinaison** du plusieurs-vers-plusieurs et un-vers-un

IRIX, HP-UX, Tru64 UNIX, Solaris 8 et avant



Librairie de thread

- Une **librairie de thread** fournit une API

Création et gestion de threads

- Deux **approches d'implémentation**

- Intégralement dans l'espace utilisateur, sans support kernel

Utilisation par invocation de fonctions

- Librairie dans l'espace noyau

Utilisation par appels systèmes

- Trois **librairies courantes** principales

POSIX Pthreads, Windows et Java

Stratégie de création

■ Threading **asynchrone**

- Le parent crée un thread
- Il continue ensuite son exécution concurrente avec son fils
- Souvent peu de partage de données entre ces threads

■ Threading **synchrone**

- Le parent crée des threads
- Il attend que ses fils se terminent avec de reprendre
- Stratégie dite de type *fork-join*
- Potentiellement plus de partage de données entre threads

POSIX Pthreads

- **Spécification POSIX** définissant une API pour des threads
IEEE 1003.1c donnant une définition, par une implémentation
- Implémentation disponible sur les **systèmes UNIX**-type
Linux, macOS, Solaris, et via 3rd-party sur Windows
- **Fichier d'entête** `pthread.h`
API pour création et synchronisation de threads

Exemple : POSIX Pthreads

```
int result;

int main()
{
    pthread_t tid;
    pthread_attr_t attr;

    pthread_attr_init (&attr);
    pthread_create (&tid, &attr, worker, NULL);
    pthread_join (tid, NULL);

    printf ("Mon fils a fini avec %d\n", result);

    return 0;
}

void *worker (void *param)
{
    result = 42;
    pthread_exit (0);
}
```

Attendre la fin de plusieurs threads

- Il suffit de construire un **tableau d'identifiants** à attendre

On attend ensuite tout le monde avec une boucle

```
#define NTHREADS 10

// ...

pthread_t workers[NTHREADS];

int i;
for (i = 0; i < NTHREADS; i++)
{
    pthread_join (workers[i], NULL);
}
```

Threads Windows

```
DWORD Result;  
  
int main()  
{  
    DWORD ThreadId;  
    HANDLE ThreadHandle;  
  
    ThreadHandle = CreateThread (NULL, 0, Worker, NULL, 0, &ThreadId);  
    if (ThreadHandle != NULL)  
    {  
        WaitForSingleObject (ThreadHandle, INFINITE);  
        CloseHandle (ThreadHandle);  
  
        printf ("Mon fils a fini avec %d\n", result);  
    }  
  
    return 0;  
}  
  
DWORD WINAPI Worker (LPVOID Param)  
{  
    Result = 42;  
    return 0;  
}
```

Threading implicite

- Ne pas laisser le développeur gérer **explicitement les threads**

Déplacement de la gestion vers les compilateurs et bibliothèques

- Trois manières principales d'avoir du **threading implicite**
 - Thread pool initialisé au démarrage du processus
 - Directives de compilateur OpenMP identifie régions parallèles
 - Grand Central Dispatch (GCD) chez Apple

- Gestion également possible par des **bibliothèques** d'un langage

Par exemple le package `java.util.concurrent` de Java

Problèmes liés au thread (1)

- Sémantique des appels systèmes `fork()` et `exec()`

Copier tous les thread lors d'un fork ou celui qui a appelé fork ?

- Gestion des `signaux` à envoyer à quels threads ?

Thread concerné, tous les threads, certains threads, un unique thread qui gère tous les signaux du processus

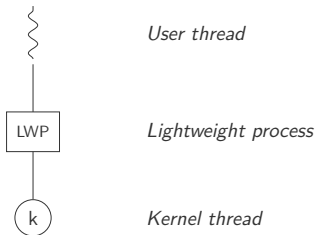
- Plusieurs manières d'`annuler` un thread

- Annulation asynchrone par un thread tueur
- Annulation retardée par vérification régulière d'un flag

Problèmes liés au thread (2)

- Besoin d'avoir un **thread-local storage** (TLS)
 - Partage des données du processus, mais besoin copies locales
 - Attention à différencier variables locales et TLS
- **Communication** nécessaire entre kernel et librairie de threads

Processeur virtuel intermédiaire sur lequel scheduler user thread



Thread sur Windows

- Les **threads Windows** implémentent le modèle un-vers-un
- Principaux **composants** d'un thread

Identifiant, registres, user/kernel stack, stockage privé

- **Structure de données** d'un thread

- ETHREAD : executive thread block

Processus parent, adresse de la routine du thread

- KTHREAD : kernel thread block

Ordonnancement, synchronisation

- TEB : thread environment block

Identificateur, stockage local (accès en user mode)

Thread sur Linux

- Sous Linux, on parle plutôt de tâche

Pas vraiment de distinction entre processus et thread

- Création d'un thread avec l'appel système `clone`

Le processus enfant partage l'espace d'adresses du parent

- Partage de plusieurs informations

- `CLONE_FS` : système de fichiers
- `CLONE_VM` : espace d'adresses
- `CLONE_SIGHAND` : gestionnaire de signaux
- `CLONE_FILES` : fichiers ouverts

Crédits

- <https://www.flickr.com/photos/dalbera/15766751411>
- <https://www.flickr.com/photos/carbonnyc/5241459773>