UNIVERSITE A.MIRA-BEJAIA
FACULTE DES SCIENCES EXACTES
DEPARTEMENT INFORMATIQUE

SUPPORT DE COURS
SYSTEMES D'EXPLOITATION
GESTION DE PROCESSUS

I. Rappel sur les systèmes d'exploitation

1. Définitions

Un système d'exploitation, est l'ensemble des programmes qui masquent toutes les opérations relatives au bon fonctionnement de l'ordinateur.

Un ordinateur se compose de :
- un ou plusieurs processeurs
- une ou plusieurs mémoires
- des interfaces de connexion aux périphériques
- une horloge pour la synchronisation
- un logiciel de base (ROM) pour interpréter et exécuter les commandes système de base.

2. Couches d'un ordinateur :

- Machine : circuits, micro-programmes, assembleur
- Système : noyau, gestionnaire de processus, de mémoire, système de gestion de fichiers, gestionnaires de périphériques, logiciels utilitaires (compilateurs, sécurité etc.)
- Applications : Traitement de texte, applications bancaires etc.

3. Historiquement :

- Système « porte fermée » avec un operateur humain ou un moniteur d'enchainement des programmes utilisateurs
- Système à traitement par lots : les entrées sorties (E/S) sont pris en charge par des processeurs auxiliaires et sont organisés en travails (jobs)
- Multiprogrammation et temps partagé : le module gestionnaire de processus et de mémoire permettent d'avoir plusieurs processus présents an mémoire qui s'exécutent en suivant un algorithme d'ordonnancement choisi.
- Système réseaux ou réparti (distribué) : le système réseau permet de relier plusieurs PCs pour des objectifs de communications, alors que le système reparti gère plusieurs PCs pour le bénéfice d'une même application (un utilisateur).

4. Modes d'exploitation

- Ordinateur individuel (PC)
- Centre de calcul (System VAX, UNIX)
- Système bancaire
- Système temps réels
- Système réparti

5. Caractéristiques communes aux SE :

- Partage des ressources : le SE assure la gestion des différents unités de la machine (CPU, MC, MA, périphériques,…) et permet l'accès aux données communes. En univers repartis, il s'occupe de la gestion de voies de communication, du control de flux et évite les congestions dans les réseaux.

- Virtualisation de la machine : un langage de commande permet à l'utilisateur de formuler ses requêtes. Les commandes sont exécutés directement par les SE (interpréteur de commandes : Shell).
-Appels système : pour créer et détruire des processus ou des fichiers, pour réaliser des entrées sorties, pour configurer la stratégie de sécurité de la machine, etc ; ils sont généralement dénommés API : Application Programming Interface.
- Programme/Processeur/Processus :
Programme : une suite statique d'informations
Processeur : unité physique qui exécute le programme
Processus : une action ou une séquence d'actions qui s'exécute pour réaliser une tache. C'est le plan d'exécution d'un programme.

- Composition d'un SE : C'est la vision d'un SE come un ensemble de couches de logiciels superposé. Chaque couche utilise les couches inferieures et fournit un service aux couches supérieures.
* Approche modulaire : noyau, gestionnaire de processus, gestionnaire de la mémoire, systèmes de gestion des fichiers, gestionnaire des Entrées/Sorties, gestionnaire du réseau et de la communication internet, interpréteur de commandes, etc.
* Approche micro-noyaux : certaines de ces fonctions sont confiées à des serveurs indépendants

- Gestion de l'information : les informations sont gérés par le SE en objets ; il gère : leur implantation dans la mémoire (adressage) leur désignation (unique) leur protection (droits d'accès), leur partage, leur destruction, leur migration, les communications et leur réutilisation !

II. Gestion de processus :
1. Définitions : un processus est une entité dynamique crée à un instant donné et détruit en général au bout d'un temps fini.

C'est un programme en cours d'exécution, il englobe les instructions (code), les données, les registres, les fichiers, la pile, et une table de symbole, ainsi qu'un ou plusieurs threads d'exécutions.

Exemples : UNIX est un système multiprocessus à thread unique. Windows est un système multiprocessus multithreads. En général :

- un processus à multi-threads est appelé tache (task)
- un processus « lourd » est une tache avec un seul thread
- un processus « léger » est un thread
- un job (travail) est un processus en traitement par lots.

2. Etats d'un processus : nouveau (création), prêt (files d'attentes), actif (processeur) bloqué (file d'attente associé à la ressource) terminé (détruit)
D'autres états existent : permuté/swappé (Mémoire centrale ou disque dur), zombie (le père est détruit) suspendu (le SE le stoppe), etc.

3. Identification d'un processus : les informations concernant le processus se trouvent dans une structure de donnée le BCP (ou PCB en anglais) le bloc de control du processus. On y trouve : PID, état, informations sur la zone mémoire occupé, comptabilité (les différents temps et dates), les informations sur les entrés sorties, PID du père, PID des fils, terminal utilisé, UID, GID, priorité, droits d'accès etc.

4. Ressources : entités nécessaires à un processus pour évoluer. On distingue :

Ressource locale / commune
Ressource commune à n points d'accès
Ressource critique n=1
Ressource libre/occupée (free/busy)
Processus indépendants ou parallèles : utilisent des ressources communes ou indépendantes
Processus coopératifs ou compétitifs : utilisent des ressources critiques ou dépendantes

Exemple : un fichier en mode lecture a n points d'accès (commune), un fichier en mode écriture a n=1 (critique).

5. Operations sur les processus : au lancement du système il n'existe qu'un seul processus (initiateur) qui est l'ancêtre de tous les autres processus. Puis le processus dispatcher (distributeur) est lancé qui lance le scheduler (ordonnanceur).
Les processus utilisateurs peuvent être lancé par des commandes (Shell) et eux-mêmes peuvent créer d'autres processus. Le processus créateur est appelé le père et les processus crées des fils. Les processus sont donc structurés en arborescence.

**Création* : la création se fait par un appel système qui définit l'identité (PID) et initialise le vecteur d'état (PCB). On définit récursivement la descendance d'un processus P :
i) un processus Q crée par P appartient à la descendance de P
ii) tout processus crée par Q appartient à la descendance de P
iii) il n'existe aucune autre façon de créer la descendance de P
Un processus fils partage les ressources du père et peut utiliser d'autres ressources du système. Un processus père peut continuer de s'exécuter en parallèle avec ses fils ou se bloquer en attente de leur terminaison.

*Destruction : pour détruire un processus un appel systemes est utilisé par le processus père ou par le SE quand:
- la dernière instruction est exécuté (exit, return, etc.)
- le père ou le SE interrompt le processus pour mauvais fonctionnement (abort)
Quand un processus père se termine normalement soit ses fils ont déjà terminés soit ils se terminent avec lui (en cascade)

6. Files d'attente des processus : les processus prêt ou bloqués sont gérés en file d'attente :
- une seule file d'attente des processus prêts : L'entête de la file contient un pointeur ver les premier et le dernier PCB et chaque PCB contient un pointeur vers le suivant.
- diagramme de files d'attente : Plusieurs files d'attente pour les processus prêts et pour chaque ressource occupée.
- files d'attente multi-niveaux à retour (MFQ : multi-level feedback queues) : chaque niveau de priorité des processus est associé une file d'attente avec un algorithme d'ordonnancement. Le feedback permet de déplacer un processus vers une priorité supérieure, s'il a trop attendu ou une priorité inferieure s'il a trop duré.

7. Ordonnancement (scheduling) :
Critères du choix d'un algorithme : temps d'attente, de restitution, de réponse, capacité de traitement, débit etc.

Scheduleur : processus système qui élit un processus parmi ceux en attente (dans la file) pour lui allouer le processeur. L'activation du processus est effectuée par un autre processus système : dispatcheur (distributeur)

Modes d'ordonnancement : long-terme (usines), moyen terme (banques), court terme (PC)

Algorithmes d'ordonnancement sans préemption : un processus reste actif jusqu'à terminaison.
FIFO (First In First Out/premier arrivé premier servi): La date de création définit l'ordre d'exécution.
SJF (Shortest Job First /le plus court d'abord) : Le temps d'exécution définit l'ordre. A chaque processus est alloué un temps d'exécution probable lors de sa création en se basant sur la teille du programme, son sens, les E/S requises et l'environnement de son traitement. Mais le temps d'exécution « estimé » reste difficile à prédire.
Algorithme à priorité (Priority): Le processus avec la plus haute priorité d'abord !
Le scheduler assigne à chaque processus créé un niveau de priorité selon certains critères :

- critères internes : limite de temps, besoin en espace mémoire, nombre de fichiers, etc.
- critères externes : importance du processus (processus système, serveur, utilisateur, etc)

Algorithmes d'ordonnancement avec préemption : un processus en cours d'exécution peut être arrêté et remis dans la file d'attente avant qu'il n'ait terminé son traitement.

RR (Tourniquet/ Round Robin) : le temps du processus est partagé en tranches égales appelés quantum (10 à 100 ms) et chaque processus de la file a droit à son quantum d'exécution. L'algorithme nécessite une file d'attente circulaire.

SRTF (Shortest Remained Time First/temps d'exécution restant d'abord) : A chaque fois qu'un nouveau processus est introduit dans le pool des processus prêts, le scheduler compare son temps d'exécution avec le temps d'exécution restant du processus actif, s'il est inferieur alors le processus actif est interrompu.

MFQ : les processus changent de files d'attente selon leur temps d'exécution.

Exemples :

- Le DEC10 avait 3 niveaux avec des quantums différents : 0.02, 0.25 et 2s

Au début, un processus est enfilé dans la première file, s'il n'a pas terminé alors il descend d'un niveau. Si un processus est interrompu durant son quantum, alors à son retour il monte d'un niveau. Le processeur est toujours alloué au 1er processus de la file du 1er niveau.

- LINUX a trois files avec 3 algorithmes : SCHED_OTHER : pour tous les processus (priorité 0) avec des niveaux de priorité dynamiques (incrémenté à chaque quantum de temps), SCHED_FIFO et SCHED_RR pour les processus système avec des niveaux de priorité(1 à 99)

Exercice : Soit un OS qui utilise 3 niveaux de priorité (ordre croissant), et des processus qui disposent de niveau fixe. Une file de processus est associé à chaque niveau et géré par un tourniquet avec un quantum de temps de 0.5s. Une file n'est activée que si les files de niveau supérieur sont vides. Donner l'ordre d'exécution des tâches suivantes :

Tâches	Temps d'exécution	Date d'arrivée	Priorité
T1	7	0	2
T2	4	0	3
T3	6	1	1
T4	1	1	2
T5	2	1	3
T6	4	2	1
T7	1	2	2

8. Commutation de contexte : consiste à sauvegarder l'état d'un processus (dans sa pile) et à restaurer l'état d'un autre processus de façon à ce que des processus multiples puissent partager les ressources d'un seul processeur. Une commutation de contexte peut être plus ou moins coûteuse en temps processeur suivant le système d'exploitation et l'architecture matérielle utilisés. Le contexte sauvegardé doit au minimum inclure une partie de l'état du processeur (registres généraux, registres d'états, etc.) ainsi que, pour certains systèmes, les données nécessaires au système d'exploitation pour gérer ce processus.

9. Communication inter-processus (IPC) : des outils systèmes existent pour permettre à des processus de communiquer ou se synchroniser.

Sémaphores : une structure de donnée composés d'un compteur (de tickets !) et d'une file d'attente (de processus qui attendent un ticket) dont l'accès exclusif est géré par le SE. On associe à cette structure deux primitives : P(S) qui permet de récupérer un ticket s'il existe sinon le processus est enfilé, et V(S) qui permet de rendre un ticket et de réveiller éventuellement un processus bloqué en attente d'un ticket.

Messages : une file d'attente de messages, de taille finie, de type chaine de caractère qui relie deux processus. Le processus émetteur crée un message avec la primitive : msgget() et envoie le message avec la primitive : msgsnd(). Le processus récepteur retire le message avec la primitive msgrcv() et c'et le SE qui assure l'accès exclusif à la file d'attente. La lecture d'un message est destructrice.

Mémoire partagée : l'outil permet de mettre en commun un segment mémoire entre plusieurs processus (ce qui est normalement interdit) et la synchronisation est à réaliser par le programmeur. Un processus alloue un segment de mémoire partagée en utilisant shmget (« SHared Memory GET », puis utilise read write pour y accéder

Tubes : ce sont des primitives qui permettent de faire communiquer deux processus sur une même machine ce qui crée un espace mémoire partagée accessible via les appels systèmes read et write. Les tubes sont unidirectionnels. La primitive qui permet de créer un tube nommé est pipe(). Les SE disposent de commandes pour la création de tube sans noms (pour Linux : **|**)

10. Primitives : Les primitives nécessaires pour la gestion de processus en C (sous Linux) et Python (obligatoirement sous Linux pour certaines).

Création de processus : La primitive fork() permet à un processus de se dupliquer – et donc créer un nouveau processus – par la fonction : `int fork(void).`
Cette fonction permet la création dynamique d'un nouveau processus qui s'exécute de façon concurrente avec le processus qui l'a créé et qui est une copie conforme.
Cette fonction retourne (par le pid_t contenu dans le header file `<sys/types.h>`) :
• -1 en cas d'échec,
• 0 dans le processus fils,
• Le PID du processus fils dans le père.

Python : **os.fork()** nécéssite **import os**

Récupération des caractéristiques : Les caractéristiques générales d'un processus :
• Identifiant du processus.
• Identifiant du père du processus.
• Répertoire de travail du processus.

• Le propriétaire réel.
• Le propriétaire effectif.
• Le groupe propriétaire réel.
• Le groupe propriétaire effectif.
Il faudrait utiliser ce qui suit (consulter le manuel man pour plus d'informations) :

```
#include <unistd.h>
#include <sys/types.h>
int getpid(void); /* pid courant*/
int getppid(void); /* pid du père */
int chdir(const char *chemin); /* changer le répertoire de travail*/
char * getcwd(char * buf, unsigned long taille); /* récupérer le
chemin absolu du répertoire de travail courant taille=256.*/
int getuid(void); /* Id du propriétaire réel */
int getgid(void); / ID du groupe du propriétaire réel */
int geteuid(void); /* propriétaire effectif */
int getegid(void); /* groupe du propriétaire effectif */
```

Exemple en Python :

```
import os
import time     #from time import sleep

pid=os.fork()
if pid==0:
  print('je suis le fils', os.getpid())
  time.sleep(0.1)              #sleep(0.1)
  print('le pid de mon pere', os.getppid())
else:
  print('je suis le pere', os.getpid())
```

Les primitives Exit() et Wait() : La fonction **exit** met fin au processus qui l'a émis, avec un code de retour **status** :

```
#include <stdlib.h>
void exit (int status)
```

Si le processus à des fils lorsque **exit** est appelé, ils deviennent des « zombies », le pid de leur processus père est changé en 1, qui est l'identifiant du processus init. Le processus 1 et le processus 0 sont chargés de l'ordonnancement des processus. Le père du processus qui effectue un **exit** reçoit son code retour à travers un appel à **wait :** `int wait (int * terminaison).`
On ajoutera les deux fichiers en-tête suivants : `<sys/types.h>` et `<sys/wait.h>.`
Un processus exécutant l'appel système **wait** est endormi jusqu'à la terminaison d'un de ses fils. Lorsque cela se produit, le père est réveillé et **wait** renvoie le PID du fils qui vient de mourir.

Python : sys.exit(status) nécéssite import sys; os.wait() nécéssite import os

Exemple

```
import os
import sys
pid = os.fork()
if pid == 0:
    print(" Processus fils affiche les nombres de 0 à 5")
    for i in range(0, 5):
        print("processus fils  %d"%(i))
    print("Process fils %d termine" %os.getpid())
```

```
    sys.exit()

else:
    print("Attendre")
    pro= os.wait()
    print("Processus fils  %d a fini" % (pro[0]))
    print("Processus pere %d termine" % (os.getpid()))
```

Les primitives de la famille exec() :
Lorsqu'un processus exécute un appel exec, il charge un autre programme exécutable en conservant le même environnement système. La partie de code qui suit l'appel d'une primitive de la famille exec ne s'exécute pas, car le processus où elle se trouve est remplacé par un autre.
Les deux fonctions de base sont **execl et execv.**

```
int execl (char *path, char *arg0, char *arg1 ... );
int execv (char *path, char * argv[ ] );
```

Dans le cas de **execl**, les arguments de la commande à lancer sont fournis sous la forme d'une liste terminée par un pointeur nul :
• **path** est une chaîne de caractères donnant le chemin absolu du nouveau programme à substituer et à exécuter.
• **arg0, arg1, …, argn** sont les arguments du programme.
• Le premier argument, **arg0**, reprend en fait le nom du programme.

Exemple :

```
void main()
{
execl("/bin/ls", "ls", "-l", NULL);
printf("Erreur lors de l'appel à ls \n");
}
```

Dans le cas de **execv**, les arguments de la commande sont sous la forme d'un vecteur de pointeurs (de type argv [ ]) dont chaque élément pointe sur un argument, le vecteur étant terminé par le pointeur NULL :

Exemple :

```
#include <stdio.h>
#define NMAX 5
void main () {
char *argv [NMAX];
argv [0] = "ls";
argv [1] = "-l";
argv [2] = NULL;
execv ("/bin/ls", argv);
printf("Erreur lors de l'appel à ls \n");
```

Python : La classe multiprocessing permet la création et la gestion de processus dans une application multi-processus.

```
from multiprocessing import Process
p = Process(target = nom_de_la_fonction, args = [argument1, argument2, ...])
p.start()
p.join()          #le processus père peut se bloquer en attente du fils
```

III. Gestion de threads

1. Définitions :

1. Un thread est un flux de séquence unique dans un processus. Étant donné que les threads possèdent certaines des propriétés des processus, ils sont parfois appelés processus à poids légers (light weight process).

2. Un thread est défini par un compteur ordinal et une pile, et sont activés au sein d'un même processus. Ils peuvent être statiques ou dynamiques
- statique : le nombre de threads est fixé à la programmation
- dynamique : la création des threads se fera selon le besoin du programmeur

Les threads communiquent via des variables partagées et cela nécessite des outils de synchronisation : sémaphore, verrou, moniteur etc.

3. Les threads sont un moyen d'améliorer une application grâce au parallélisme. Par exemple, dans un navigateur, plusieurs onglets peuvent être des threads différents. MS Word utilise plusieurs threads, un thread pour formater le texte, un autre thread pour traiter les entrées, etc.
Les threads fonctionnent plus rapidement que les processus pour les raisons suivantes :
1) La création de threads est beaucoup plus rapide.
2) La commutation de contexte entre les threads est beaucoup plus rapide.
3) Les fils peuvent être terminés facilement
4) La communication entre les threads est plus rapide.

Inconvénient : il n'existe pas de protection inter-threads !

2. Threads utilisateur et noyau

La majorité des systèmes permettent le multithreading. Ils sont crées soit au niveau utilisateur, soit au niveau noyau.

Les threads utilisateur sont supportés au-dessus du noyau et sont implantés par une bibliothèque de threads au niveau utilisateur (par exemple pthread sous Linux ou thread dans Solaris). Ils sont portables sur différentes plate-formes. Ils sont gérés par une application où le blocage du thread peut bloquer le processus complet. Le changement de contexte est rapide.

Les threads noyau sont directement supportés par le noyau du système d'exploitation. Le système d'exploitation se charge de leur gestion et le changement de contexte est lent.

Les threads combinés sont implantés par le système d'exploitation (utilisateur et système). Les threads utilisateur sont associés à des threads système. La plupart des tâches de gestion s'effectuent sous le mode utilisateur.

3. Les états d'un thread :

Create : le thread est crée mais pas encore activé
Ready : le thread est prêt
Usr : le thread est activé en mode utilisateur
Kernel : le thread est activé en mode noyau
Wait : le thread est bloqué en attente d'une ressource
Zombie : le thread est mort mais n'est pas encore éliminé
Dead : le thread n'éxiste plus !

4. Contexte d'un thread :

Les threads ne sont pas indépendants les uns des autres contrairement aux processus. En conséquence, les threads partagent avec d'autres threads leur section de code, leur section de données et les ressources du système d'exploitation telles que les fichiers ouverts et les signaux. Mais, comme les processus, un thread a son propre compteur de programme (PC), un ensemble de registres, un espace mémoire pour les données dynamiques appelé TAS (ou heap en anglais) et un espace de pile.

5. Ordonnancement des threads:
Chaque thread peut dépendre d'une politique d'ordonnancement différente. On peut créer des threads plus prioritaires et donc décider de l'ordre d'exécution.
Il existe un ordonnanceur par cœur (core) dans une machine et le thread est affecté automatiquement à un core lors de sa création et donc bénéficie de la politique d'ordonnancement de celui-ci. Le noyau ne permet pas la migration de threads entre cores.
L'ordonnanceur propose une API qui décrit des opérations de transition d'état des threads.
L'implémentation de ces opérations, qui définissent la politique d'ordonnancement choisi, peut être effectuée et modifiée par le programmeur.
Les algorithmes d'ordonnancement possibles sont les même que pour les processus.

6. Modes d'exploitation :
La majorité des SE actuels sont multi-threading, i.e. le processus lance plusieurs threads à la fois pour améliorer le parallélisme (pseudo) surtout dans un environnement multi-cores. Plusieurs modes d'exploitation sont possibles suivant le SE :

- Modèle répartiteur/ouvrier : ou maitre esclave, le SE crée un thread principal (avec une plus grande priorité) qui coordonne un certains nombre de threads de même priorité pour effectuer les calculs.
- Modèle équipe : Comme dans Linux, les threads appartenant au même processus ont la même priorité (hérité du processus).
- Modèle pipeline : plus rare, ou la sortie de l'un est l'entrée de l'autre. Un ordre d'activation des threads est prédéfini par le SE.

7. Primitives : Les opérations concernant les threads sont notamment la création, la terminaison, la suspension et la relance.

Primitive de creation:
```
int pthread_create(pthread_t * thID,
                   const pthread_attr_t * attr,
                   void * (*thFonc)(void *),
                   void *arg);
```

thID: pointeur vers une valeur entière non signée qui renvoie l'identifiant du thread créé.
attr : pointeur vers une structure utilisée pour définir les attributs de thread (NULL par défaut).
thFonc : pointeur vers un sous-programme exécuté par le thread. Le type de retour et le type de paramètre de la fonction doivent être de type void *. La fonction a un seul attribut mais si plusieurs valeurs doivent être transmises à la fonction, une structure doit être utilisée.
arg : pointeur vers void qui contient les arguments de la fonction du thread (possibilité de passer des paramètres de type scalaire ou pointeur)
Le TID (thread identifier) est récupéré grâce à pthread_self() ! (en Python : nomthread.ident)

Exemple en Python:
```
import threading
def afficher(n):
    print('thread secondaire', n)
```

th1=threading.Thread(target=afficher, args=("A",)).start() #th1=Thread(target=afficher, args=(“A”,))
th2=threading.Thread(target=afficher, args=("B",)).start()

ou bien

```
from threading import Thread
def afficher(n):
        tid=threading.get_native_id()
        print(n,tid)

th1=Thread(target=afficher, args=("A",))
th2=Thread(target=afficher, args=("B",))
th1.start()
th2.start()
```

Primitive de terminaison: **ret** est la valeur retourné, qui peut être récupéré par un autre thread en exécutant pthread_join.

```
pthread_exit(void *ret)
```

Python : sys.exit() necessite import sys

Primitive de blocage (d’attente): La primitive join est bloquante. Le thread principal attend la terminaison des threads fils. La variable retour contient l’état de terminaison du thread (NULL par défaut et éventuellement un paramètre (message).

```
pthread_join(pthread_t thID, void **retour).
```

Exemple Python:

```
from random import random
from time import sleep
from threading import *
import sys

def task():
    print(current_thread().ident)
    while True:
        valeur = random()
        print(f'.{valeur}')
        sleep(valeur)
        if valeur > 0.9:
            print('Thread secondaire termine')
            sys.exit()

thread = Thread(target=task)
thread.start()
thread.join()
print('Thread principal termine')
print("Main Thread Identificateur", current_thread().ident)
print("Thread fils Identificateur", thread.ident)
```

Primitive de comparaison: compare si deux threads sont identiques ou non. Si les deux threads sont égaux, la fonction renvoie une valeur non nulle sinon nulle.

```
int pthread_equal(pthread_t t1,  pthread_t t2);
```

Primitive d'annulation : utilisé pour envoyer une demande d'annulation à un thread
```
int pthread_cancel(pthread_t th);
```

En python il n'existe pas d'annulation de thread sauf à travers les événements !

Primitive de détachement: utilisé pour détacher un thread. Un thread détaché ne nécessite pas qu'un thread se joigne à la fin. Les ressources du thread sont automatiquement libérées après l'arrêt si le thread est détaché.
```
int pthread_detach(pthread_t th);
```

En python, un thread détaché est appelé daemon et il suffit de mettre à vrai la variable daemon lors de la création (thread.daemon = True)

Primitive de céder le processeur : permet à un thread non-important de laisser passer un thread important !
`pthread_yield(void)` ou (shed_yield())

En python, cette fonction n'existe pas on utilise à la place sleep() !

Attributs d'un thread : Les attributs d'un thread sont définis dans `pthread_attr_t` et contient : l'adresse de départ et la taille de sa pile, la politique d'ordonnancement associée, sa priorité, son attachement ou son détachement. Ces attributs (xxx) peuvent être modifiés par les primitives `pthread_attr_init()`, `pthread_attr_getxxx()` , `pthread_attr_setxxx()`.

Il est possible de connaître la priorité ou la politique d'ordonnancement d'un thread avec :
```
pthread_attr_getprio(pthread_attr_t *attr,int prio);
pthread_attr_getsched(pthread_attr_t *attr,int politique);
pthread_getprio(pthread_t *unepthread, int prio);
pthread_getsched(pthread_t *unepthread, int politique);
```

Modifier l'ordonnancement : Les différentes politiques possibles sont : SCHED_FIFO, SCHED_RR , SCHED_OTHER
```
pthread_attr_setsched(pthread_attr_t *attr, int politique);
```

Modifier la priorité
```
pthread_attr_setprio(pthread_attr_t *attr, int prio);
```
La priorité varie dans un intervalle défini par la politique :
PRI_OTHER_MIN <= prio <= PRI_OTHER_MAX
PRI_FIFO_MIN <= prio <= PRI_FIFO_MAX
PRI_RR_MIN <= prio <= PRI_RR_MAX

Pour afficher les attributs threading.Thread en python:
- name : obtenez ou définissez le nom du thread.
- ident : Accédez à l'identifiant de thread unique attribué par l'interpréteur Python.
- native_id : accédez à l'identifiant de thread natif unique attribué par le système d'exploitation.
- démon : obtient ou définit si le thread est un thread démon ou non.
- is_alive() : vérifie si le thread est en cours d'exécution.

Exemple :

```
from threading import Thread
thread = Thread()
print(thread.name)
print(thread.daemon)
print(thread.ident)
print(thread.native_id)
thread.start()
print(thread.ident)
print(thread.native_id)
```

Un verrou d'interpréteur global (Global Interpreter Lock GIL) est utilisé pour implémenter les threads Python, ce qui signifie que la priorité d'un thread ne peut pas être contrôlée.

8. Communication entre threads :
- Variables globales : Le programme principal et le thread qu'il a créé partagent trois zones de la mémoire : le **segment Code** qui comprend l'ensemble des instructions qui composent le programme, le **segment de données** qui comprend toutes les données statiques , initialisées ou non et enfin le TAS Autant le programme principal que ses threads peuvent accéder à n'importe quelle information se trouvant en mémoire dans ces zones. Par contre, le programme principal et le thread qu'il vient de créer ont chacun leur propre contexte et leur propre pile. Mais cela peut aboutir à des incohérences et donc nécessiter une synchronisation.
- Primitive pthread-join/pthread-exit : le exit permet au thread de passer un message au processus qui exécute le join et donc de récupérer le message
-Primitive pthread-create : Elle permet le passage d'un seul argument de la fonction affecté au thread crée. Si la fonction nécessite plusieurs arguments alors il faut utiliser une structure de données (un enregistrement !)

9. Avantages et inconvénients

- Les programmes utilisant des threads sont plus rapides que des programmes classiques, en particulier sur les machines comportant plusieurs processeurs à cause de la commutation de contexte.
- La communication entre processus est couteuse et nécessite souvent l'utilisation de mécanisme fourni par le système. Alors que le partage de certaines ressources entre threads permet une communication plus efficace entre les différents threads d'un processus car les threads partagent une partie de l'état du processus.
- La programmation utilisant des threads est plus difficile, et l'accès à certaines ressources partagées doit être contrôlé par des outils de synchronisation (sémaphores par exemple), en sachant que l'utilisation de la synchronisation peut aboutir à des situations d'interblocage.
- La complexité des programmes utilisant des threads est plus grande que celle des programmes constitués de plusieurs processus plus simples. Cette complexité accrue, lorsqu'elle est mal gérée lors de la phase de conception ou de mise en œuvre d'un programme, peut conduire à de multiples problèmes.