Système d'Exploitation (Operating System)

Linux –Distribution **Ubuntu**

TP SE1 pour Licence 2 –Informatique, A.U. 2023-2024

Initiation Rapide & Prise en Main de Linux –Ubuntu

Dpt Informatique. Université de Skikda

TP SE1 pour Licence 2 –Informatique, A.U. 2023-2024

1. Introduction aux Systèmes d'Exploitations (SE)

- Différents SE existent : Unix, Linux, OS, Windows, iOs, Android, Windows Phone, …
- OS (Mac, Apple) et Windows (Microsoft) : SE commerciaux (Copyright)
- Unix et Linux : SE open source (Copyleft) soumis à la licence GNU
- Windows et OS admettent plusieurs versions
- Linux : admet plusieurs distributions (RedHat, **Ubunto**, Debian, Fedora, Mandriva, …)
- Tous les laboratoires de recherche (en informatique), académies scientifique et universités du monde entier utilisent Linux.

2. Installation du Système Linux

- Différents environnements existent : PC, station de travail, serveur, …
- Chaque environnement a sa propre configuration machine : il faut en tenir compte !

- Plusieurs Modes d'installation :
  - Linux seul sur une machine,
  - Sous Machine virtuelle,
  - Au choix au boot (au démarrage) avec un autre SE (ex. Windows)

3. Commandes Linux de Base et Notion de Shell

- Linux dispose de 2 modes :
  - Mode textuel (ligne de commande avec prompt ou invite)
  - Mode graphique
- Le système Unix/Linux fait la différence entre *majuscule* et *minuscule* (ex. Paradigme PAradigme).

3. Commandes Linux de Base et Notion de Shell

- Après installation de Linux, toujours créer son compte utilisateur (répertoire *home*) puis son propre répertoire de développement.
- Ne jamais travailler au ***root*** (administrateur) –répertoire racine.
- Il faut toujours se connecter au compte utilisateur (user/home/~$) créer lors de l'installation (et ouvert en session user).
- Le compte *root* (racine: /$) ne doit servir qu'à l'administration du système.

- **exit** : commande pour fermer la session courante
- **passwd** : pour modifier le mot de passe (au moins 6 caractères)

3.1. man (manuel)

- Commande ***man*** affiche l'aide en consultant les pages de manuel
- Pour obtenir de l'aide sur une commande, tapez :

  man <nom_commande>

- L'aide sur une option : *man* ***a*** *ls* (donne les infos sur l'option ***a*** de **ls**)

3.2. Types de commandes

- Trois types de commandes existent :
  - Pour **répertoires**,
  - Pour **fichiers**,
  - Pour **processus**
- Commandes pour Répertoires (directories)

cd	pwd	ls	mkdir	rmdir

- Commandes pour Fichiers (Files)

touch	cp	mv	cat	echo	nano	rm	diff

- Commandes pour Processus, ex. : **ps** -f

3.3. Commandes pour Répertoires
ls / mkdir / rmdir / cd / pwd

- Ces commandes s'appliquent uniquement sur les répertoires

3.3.1 *ls*

- ***ls*** : permet de lister le contenu d'un répertoire (liste d'un répertoire)
- Beaucoup d'options sont disponibles pour cette commande, parmi elles, certaines sont intéressantes
- **ls** (**sans option**) : liste les fichiers en plusieurs colonnes
- ***ls –l*** : liste les fichiers et répertoires avec les droits d'accès
- **ls –a** : liste tous les fichiers (même ceux commençant par un point)

- **NB**: La touche flèche "haut" du clavier vous permet de retrouver les commandes que vous avez déjà tapé.

3.3.2. mkdir / rmdir

- **mkdir** : (make directory) création d'un répertoire

  mkdir nom_répertoire : permet de créer un répertoire

- **rmdir** : (remove directory) destruction d'un répertoire

NB: rmdir nom_répertoire : permet de supprimer un répertoire vide

- **rm -r** nom_répertoire : permet de supprimer un répertoire non vide

3.3.3. cd (change directory)

- La commande **cd** vous permet de se déplacer dans les répertoires tant que les permissions l'autorisent
- Tapez **cd**
- **cd** <sans paramètre> vous ramène dans votre répertoire personnel (rép. par défaut de l'utilisateur : **~** ou **$HOME**)
- **cd /home** : entrer dans le répertoire /home s'il existe !!
- Tapez **cd ..** (puis **pwd**), que constateriez-vous ? (pour remonter au rép. parent)
- Tapez **cd /** que constateriez-vous ? (pour remonter au rép. racine)

- Exemple : Créer un répertoire avec : **mkdir SE1G523**

  puis **cd SE1G523**

  puis **cd ..** ou **cd /**

  puis **cd** (rép. par défaut user **~** ou **$home**) puis **rmdir SE1G523**

3.3.4 pwd

- **pwd** : permet de savoir dans quel répertoire vous êtes ! (affiche le rép. courant)

Exercice

Réalisez les traitement suivants :

1. Donner la commande qui permet d'afficher votre répertoire de travail courant.
2. Créer les répertoire tplinux et tp à la racine de votre répertoire personnel.
3. Créer les répertoires tp1 sous tplinux ET tp2 sous tp.
4. Supprimer les répertoires tplinux et tp;
5. Créer de nouveau en une seule commande ces quatre répertoires.
6. Placez-vous dans le répertoire tplinux.
7. Vérifier que vous êtes bien dans ce répertoire.
8. Placer vous dans le répertoire tp1.
9. Déplacer vous du répertoire tp1 au répertoire tp.
10. Afficher le contenu de ce répertoire. quelle est la taille des fichiers? Le propriétaire? La date de création?

3.4. Commandes pour Fichiers

touch / mv / rm / cat /diff / echo / cp / nano

- Ces commandes ne concernent que les fichiers.

3.4.1. Commandes pour Fichiers : touch /cp / mv / cat / nano

- **touch :** permet de créer un fichier

  touch nom_fichier

- **cp** : permet de copier un ou plusieurs *fichiers* d'un emplacement (répertoire) vers un autre répertoire (dont il faut préciser le chemin d'accès : " path")

  cp nom_fichier /home/perso

- **mv** : permet de déplacer un ou plusieurs *fichiers* d'un répertoire vers un autre

  mv nom_fichier /home/perso

- **cat >nom_fichier**: remplir un fichier (Entrée → ctrl +c)
- **cat nom _fichier**: afficher le contenu d'un fichier

- **cat –n nom fichier 1 nom fichier 2 >nomfichier 3 : fusionner deux fichier**
- **nano nom_fichier**: étendre le contenu d'un fichier (ctrl+x) → o → entrée)

Remarque
mv: permet aussi de renommer un fichier

3.4.6. Commandes pour Fichiers : rm

- **rm** : permet de détruire (supprimer) un ou plusieurs *fichiers* (liens logiques) ou *liens physiques* du répertoire courant (dans certains cas il faut préciser le chemin “ path”)

NB : Un fichier supprimé sous Linux ne peut pas être récupéré

- **rm** <nom_fichier> : efface un fichier
- **rm** * : efface tous les fichiers du répertoire
- **rm** –i <fichier1> <fichier2> <fichier3> : –i demande confirmation de l’effacement de chaque fichier

3.4.5. Commandes pour Fichiers : diff / echo

- **diff** : affichage des différences entre deux fichiers
- Cette commande affiche les différences entre deux fichiers.
- Cela est parfois utilisé par des utilitaires stockant des modifications incrémentales dans des sources ou textes sous leur contrôle, permettant:
  - de retrouver des versions antérieures ou
  - le travail à plusieurs
- La commande **echo** : permet l'affichage d'un msg sur l'écran
- Elle écrit sur la ligne courante les arguments qui lui sont passés :

echo arg1 arg2 …

Exemple : **echo "Bonjour !"**

Remarque : echo "Bonjour !" > nom fichier : permet d'afficher le message dans un fichier

Exercice

Réalisez les traitement suivants :

1. Créer dans votre répertoire TPLINUX un fichier monfichier1 contenant le texte ”c’est la séance de TPSE1”.
2. Créer dans votre répertoire TPLINUX un fichier monfichier2 contenant le texte ”c’est le deuxième exercice ”.
3. Afficher le contenu du fichier monfichier 1
4. Concaténer (fusionner) le fichier monfichier1 et monfichier2 dans un fichier monfichier3.Afficher le résultat.
5. Copier les deux fichier monfichier1et monfichier2 dans le dossier TP;
6. Renommer les deux fichier monfichier1 et monfichier2 du dossier TP monfichier3 et mon fichier4.
7. Que fait la commande echo ”créer”.

Le Multi-Processing (Multi-tâches)

- Multi-(processing/traitement/Utilisateur/Users/tâches/tasks/ processus/threads …)
- Unix/Linux est un SE multi-tâche multi-utilisateurs
- Un utilisateur peut **lancer plusieurs tâches (programmes, processus ou commandes) en parallèle et même en concurrence**

- Pour comprendre le concept de multitâche sous Linux, lancer la commande :

ps *–eaf* *ou bien* *ps -ef*

Le Multi-Processing (Multi-tâches)

On devrait obtenir des infos sous la forme :

UID	PID	PPID	C	STIME	TITY	TIME	CMD
root	1	0	0	08:27	?	00:00:04	Init [5]

•Maintenant, lancer la commande : ***gedit puis ps -eaf (sur 2 fenêtres différentes)***
•Chaque tâche (processus ou Cmd) appartient à quelqu'un (utilisateur) identifié par son **UID**
•Chaque processus est identifié par son numéro, c'est son **PID**
•Chaque processus est le fils d'un autre processus qui est identifié (le processus père) par son **PPID**
•Le reste des infos sont des ***infos de contrôle qui servent au scheduler (ordonnanceur de tâches)***
•Pour n'avoir que les processus qui concernent l'utilisateur actuel, lancer la commande suivante ***: ps -eaf | grep login_utilisateur***

Commandes sur les Processus

-Lister vos processus à l'aide de la commande :

ps –f –u usager où usager est votre identifiant login

–Notez le **PID de celui qui contient la chaine *gedit***

-Tapez la commande ***kill*** **N°PID**

–Notez le **PID de celui qui contient la chaine *bash***

•Tapez la commande: **kill -9 N°PID**

–où **N°PID est le numéro du processus que vous avez relevé.**

–Que s'est-il passé ?

•Tapez

–la commande **top**

Commandes pour processus : ps

- **ps :** fournit la liste des processus actifs (en cours d'exécution)dans le shell actuel

- **ps -f –u user :**Affiche tous les processus associés à un utilisateur particulier

- **ps –ef :** affiche tous les processus en cours d'exécution

- **ps –aux :** Liste de tous les processus du **système**

ps –fg root : Afficher tous les processus associés à un groupe d'utilisateurs particulier
NB: Ici, vous pouvez remplacer UserGroupName par le nom du groupe d'utilisateurs dont vous souhaitez répertorier les processus associés. Par exemple, nous l'avons remplacé par «root» dans notre cas.

- **NB : la commande top : permet de connaître les processus gourmands en puissance de calcul.**

- Ce qui nous intéresse c'est la première colonne PID, qui est le numéro des processus. C'est ce numéro qui est utilisé par **kill**

Commandes pour processus : kill

kill : cette commande permet de détruire (tuer, supprimer) un processus

•Exemple : si le processus cible est le numéro (PID) **546**

•**kill 546**

Tente de détruire le processus 546

•**kill -9 546**

Force la destruction du processus 546

Éditeur de texte sous Linux

Comment écrire son programme ?

•Comme l'écriture de programmes et de scripts passe par l'utilisation d'un éditeur de texte, on se familiarisera avec l'éditeur *gedit*

Programmation en C sous Linux-Ubuntu

Compiler un programme C

- Le Compilateur **GCC = GNU (OpenSource Project) C Compiler**
- C'est le **Compilateur C/C++ pour Linux**
- **gcc s'exécute à partir de l'interface en ligne de commande.**

Ecrire et Compiler mon premier programme C

Ecriture du programme

•À partir du shell (terminal) taper la commande: gedit hello.c

•Ecrire le programme suivant:

```
#include <stdio.h>
int main() {
printf ("Hello word ! \n");
return 0;
}
```

• Compiler et exécuter le programme

- pour Compiler et exécuter ce programme on tape la commande:
gcc hello.c

- Puis la commande : ./a.out

```
[sysexploi@local]$ gcc hello.c
[sysexploi@local]$ ./a.out
Hello word !
```

Quelques Options Utiles

-o filename : permet de changer le nom du fichier de sortie (output).

```
[sysexploi@local]$ gcc hello.c -o Test
[sysexploi@local]$ ./Test
Hello word !
```

Exercice

1. En utilisant l'éditeur gedit, écrire un programme C qui calcule la surface d'un rectangle.
2. Compiler et exécuter le programme.

Solution

```
#include <stdio.h>
int main ()
{
float S, Larg, Long ;
printf (" donnez la largeur du rectangle : ");
scanf ( " %f ", &Larg);
printf (" donnez la longeur du rectangle : ");
scanf ( "  %f ", &Long);
S = Larg * Long ;
printf ("  la surface du rectangle  est : %f\n" ,S);
return 0;
}
```

Initiation Rapide & threads

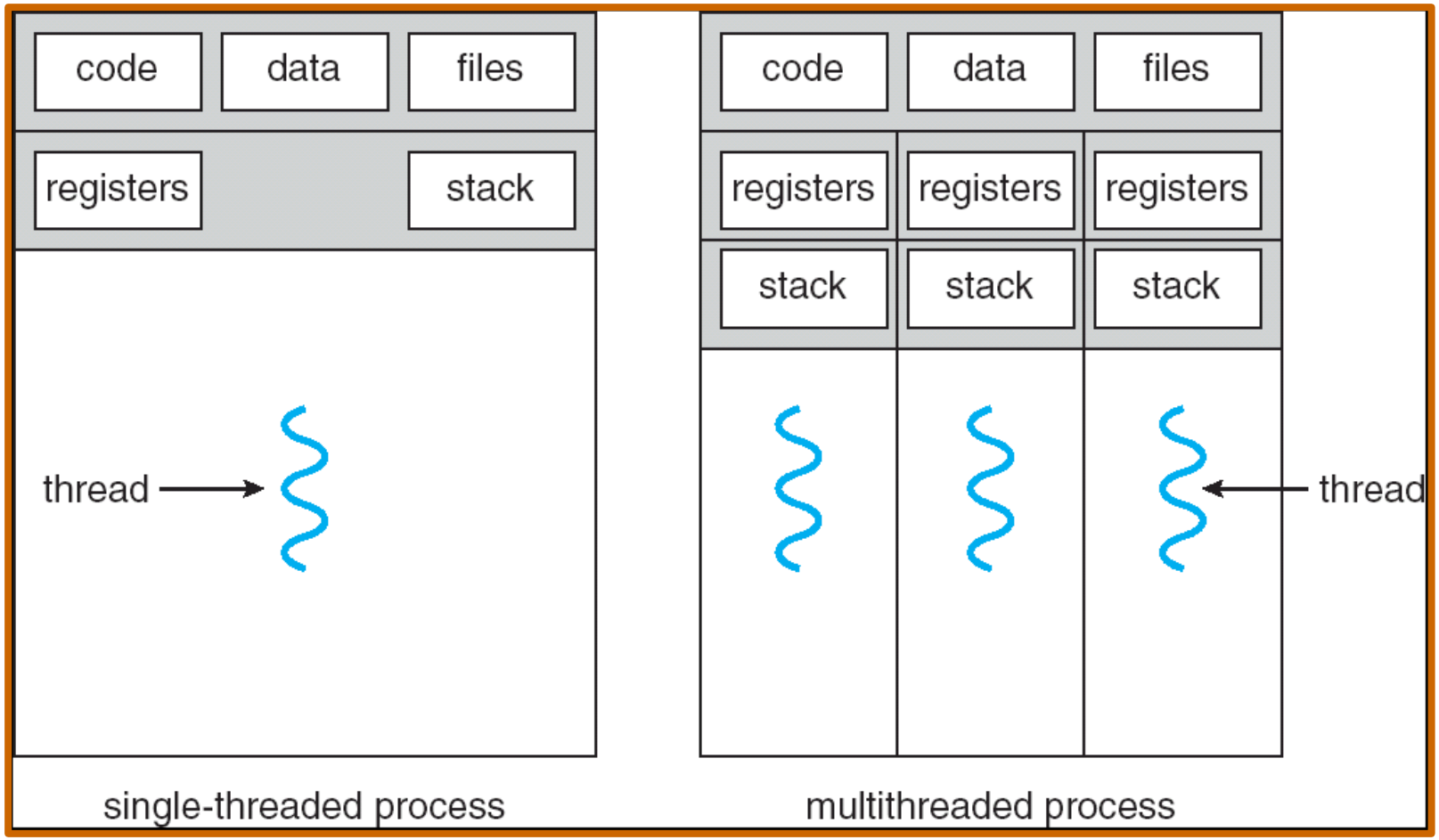
Fil (fil d'exécution)= Flots = threads = lightweight processes = Portion de code (fonction)

- **Un thread est une subdivision d`un processus**
  - Un fil de contrôle dans un processus
- **Les différents threads d 'un processus partagent l'espace adressable et les ressources d'un processus**
  - lorsqu'un thread modifie une variable (non locale), tous les autres threads voient la modification
  - un fichier ouvert par un thread est accessible aux autres threads (du même processus)

Exemple

- **Le processus MS-Word implique plusieurs threads:**
  - Interaction avec le clavier
  - Rangement de caractères sur la page
  - Sauvegarde régulière du travail fait
  - Contrôle orthographe
  - Etc.
- **Ces threads partagent tout le même document**

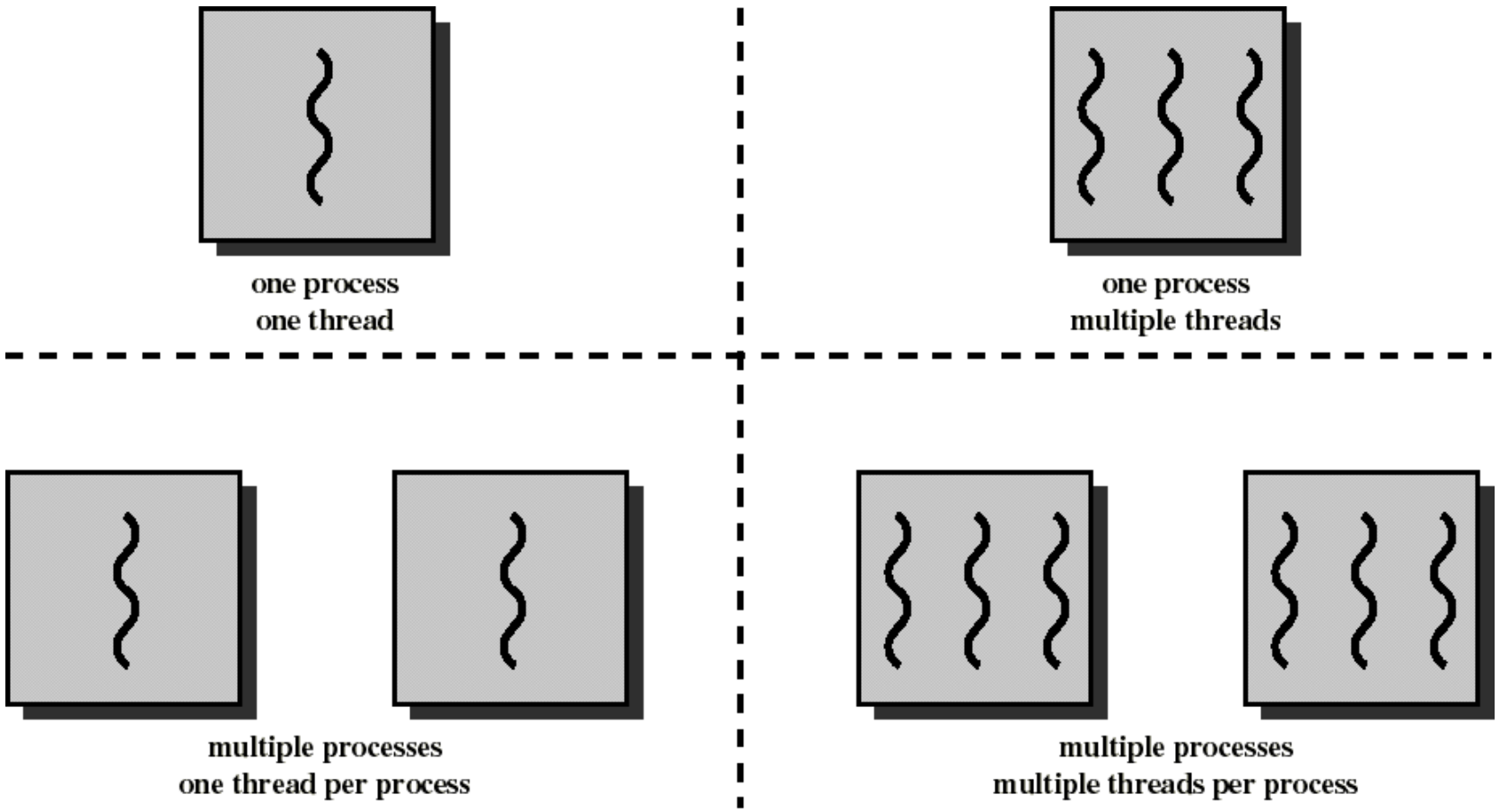
Processus à un thread et à plusieurs threads



Mono-flot

Multi-flots

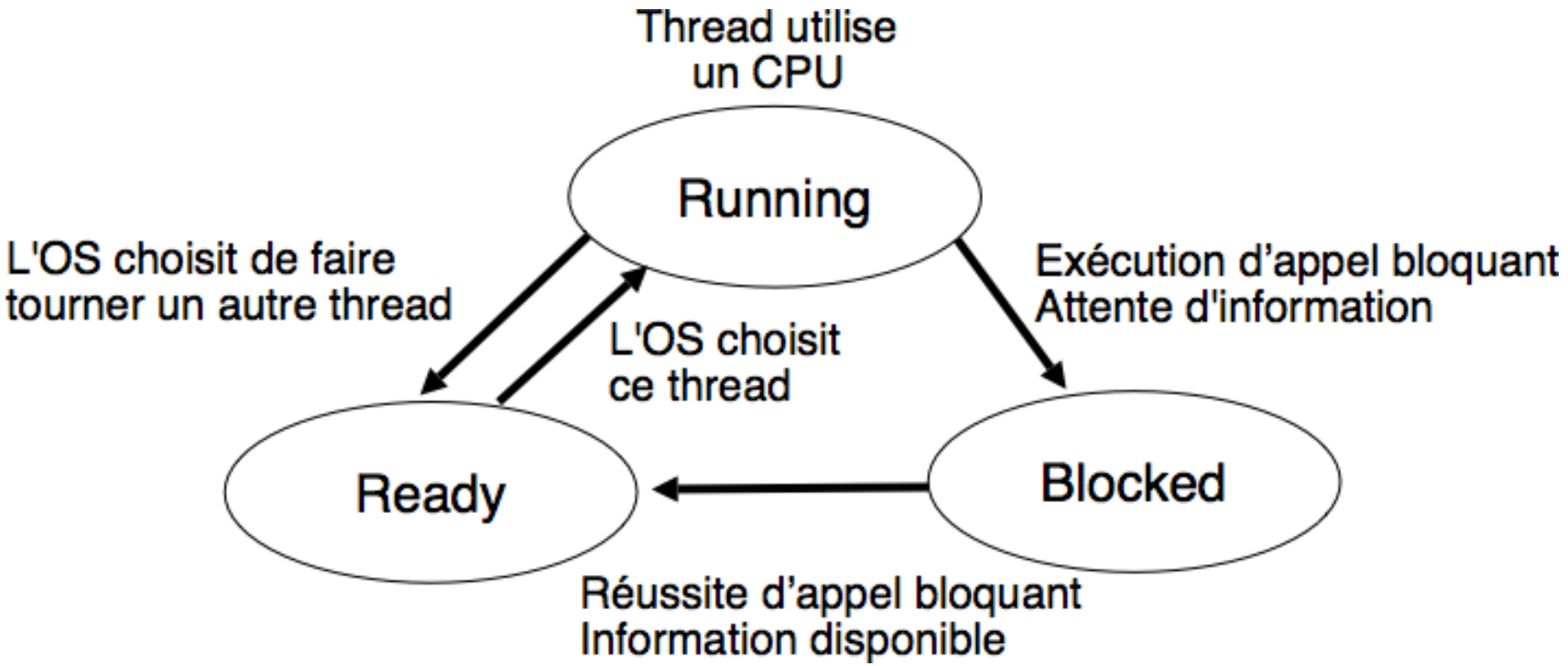
Threads et processus



Thread

- **Possède un état d'exécution (prêt, bloqué…)**
- **Possède sa pile et un espace privé pour variables locales**
- **A accès à l'espace adressable, fichiers et ressources du processus auquel il appartient**
  - En commun avec les autres threads du même processus

Thread


Thread utilise
un CPU
Running
L'OS choisit de faire
tourner un autre thread
L'OS choisit
ce thread
Exécution d'appel bloquant
Attente d'information
Ready
Blocked
Réussite d'appel bloquant
Information disponible

Pourquoi les Threads

- **Réactivité: un processus peut être subdivisé en plusieurs threads, p.ex. l'un dédié à l'interaction avec les usagers, l'autre dédié à traiter des données**
  - L'un peut exécuter tant que l'autre est bloqué

- **Utilisation de multiprocesseurs: les threads peuvent exécuter en parallèle sur des UCT différentes**

La commutation entre threads est moins dispendieuse que la commutation entre processus

- **Un processus possède mémoire, fichiers, autres ressources**
- **Changer d`un processus à un autre implique sauvegarder et rétablir l'état de tout ça**
- **Changer d'un thread à un autre *dans le même proc* est bien plus simple, implique sauvegarder les registres de l 'UCT, la pile, et peu d 'autres choses**

La communication aussi est moins dispendieuse entre threads qu'entre processus

- **Étant donné que les threads partagent leur mémoire,**
  - la communication entre threads dans un même processus est plus efficace que la communication entre processus

La création est moins dispendieuse

- **La création et terminaison de nouveaux threads dans un proc existant est aussi moins dispendieuse que la création d'un processus**

Pourquoi les Threads(Résumé)

Les threads sont un moyen populaire d'améliorer l'application par le parallélisme.(permettent de dérouler plusieurs suites d'instructions, en PARALLELE, à l'intérieur d'un même processus. Un thread exécutera donc une fonction).

Les threads fonctionnent plus rapidement que les processus pour les raisons suivantes :

1) La création de threads est beaucoup plus rapide.

2) Le passage d'un contexte à l'autre est beaucoup plus rapide.

3) Les threads peuvent être terminés facilement

4) La communication entre thread est plus rapide

Pthreads

- **Une norme POSIX (IEEE 1003.1c) d'un API pour la création et synchronisation de thread**
- **Commun dans les systèmes d'exploitation (OS) UNIX (Solaris, Linux, Mac OS X)**
- **Fonctions typiques:**
  - pthread_create (&threadid,&attr,start_routine,arg)
  - pthread_exit (status)
  - pthread_join (threadid,status)
  - pthread_attr_init (&attr)

Lancer des thread en C

Créer des threads(1/2)

Pour créer des threads, on utilise la bibliothèque pthread dont l'en-tête est pthread.h.

Créer des threads(2/2)

Pour créer un thread :

• on définit une valeur pthread_t t qui va contenir les données du thread

• on appelle pthread_create(&t, NULL, f, arg); où f est une fonction de prototype void *f(void *arg); et arg est une valeur de type void.

Remarque:

- Pour lancer plusieurs threads, il suffit de faire plusieurs appels à pthread_create.

Attente de terminaison d'un thread

Pour attendre qu'un thread t termine on appelle pthread_join(t, NULL)

Si on écrit :

```c
pthread_t t1, t2;
pthread_create(&t1, NULL, f, NULL);
pthread_join(t1, NULL);
pthread_create(&t2, NULL, f, NULL);
pthread_join(t2, NULL);
```

Les deux threads ne vont pas s'exécuter en même temps car on attend avec le `join` que le premier thread termine pour lancer le second.

Il faut donc créer les threads puis les attendre :

```c
pthread_t t1, t2;
pthread_create(&t1, NULL, f, NULL);
pthread_create(&t2, NULL, f, NULL);
pthread_join(t1, NULL);
pthread_join(t2, NULL);
```

Terminaison d'un thread

pthread_exit (status); Le paramètre sera le code de retour du thread (on peut mettre NULL si on ne veut pas de code de retour).

Attributs de thread

Modifier les attributs des threads revient à remplir la structure des attributs de threads qui est du type pthread_attr_t, puis à la passer en tant que deuxième argument à pthread_create().

pthread_attr_init() initialise la structure d'attributs de thread (2ème paramètre de pthread_create) et la remplit avec les valeurs par défaut pour tous les attributs.

Exemple de programmation en C avec Threads

(Exemple1: un seul thread)

```
/* onethread.c */

#include <stdio.h>

#include <pthread.h>

void *ThreadFun()
{

    printf("Hello Je suis un Thread \n");
}

int main()
{
   pthread_t t1;

   pthread_create(&t1, NULL, ThreadFun, NULL);

   pthread_join(t1, NULL);

   return 0;}
```

Explication du code

Dans main() nous déclarons une variable appelée t1, qui est de type pthread_t, qui est un entier utilisé pour identifier le Thread dans le système.

Après avoir déclaré t1, nous appelons la fonction pthread_create() pour créer un thread. pthread_create() prend 4 arguments.

- **Le premier argument est un pointeur vers le t1qui est défini par cette fonction.**
- **Le deuxième argument spécifie les attributs (exe, la taille de la pile, la politique d'ordonnancement, etc.). Si la valeur est NULL, alors les attributs par défaut sont utilisés.**
- **Le troisième argument est le nom de la fonction à exécuter pour le Thread à créer.**
- **Le quatrième argument est utilisé pour passer des arguments à la fonction, ThreadFun.**
- **La fonction pthread_join() fait attendre la fin d'un thread.**

Compilation et exécution d'un programme c avec threads

pour compiler ce programme il faut établir un lien avec la bibliothèque des threads :

Pour compiler : gcc –pthread onethread.c

pour executer: ./a.out

Exemple2: deux (2) threads

```
/* twothread.c */

#include <stdio.h>
#include <pthread.h>
void *fun1(void *arg)
{
printf("Hello Je suis le Thread 1\n");

}
void *fun2(void *arg)
{
    printf("Hello Je suis le Thread 2\n");

}
int main()
{
    pthread_t t1;
    pthread_t  t2;
    pthread_create(&t1, NULL, fun1, NULL);
    pthread_create(&t2, NULL, fun2, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;}
```

Thread: problème de partage de données(1/2)

-Un thread est un sous-ensemble d'un processus, partageant son espace mémoire et ses variables.

- De ce fait, les coûts associés suite à son lancement sont réduits, donc il est plus rapide.

- En plus de cela, à chaque thread est associé des unités propres à lui: comme sa pile (pour gérer les instructions à exécuter par le thread), le masque de signaux (les signaux que le thread doit répondre), sa priorité d'exécution (dans la file d'attente), des données privées, etc.

- Il faut ajouter à cela, les threads peuvent être exécutés en parallèle par un système multitâche.

- Cependant, le partage de la mémoire et les variables du processus entraînent un certain nombre de problèmes lorsqu'il y a un accès partagé à une ressource, par exemple.

- Deux threads voulant effectuer des réservations de places d'avion (ressource: nombre de places disponibles dans un avion).

-On protège l'accès à cette ressource dès qu'un thread est en train de réaliser une écriture (réaliser une réservation) sur cette ressource.

Thread: problème de partage de données(2/2)

En bref,

- Les différents threads d'une application peuvent accéder à toutes les données pour lesquelles ils possèdent une référence. Cela inclut des données partagées par ces différents threads : les différents threads peuvent lire et modifier ces données en parallèle et potentiellement en concurrence. Ces mises à jour peuvent laisser les données dans un état incohérent .
- Lors de la parallélisation de traitements, il est fréquent de devoir gérer des accès concurrents à certaines données: il est nécessaire de synchroniser les threads.

Exemple 3: deux (2) threads partageant une variable (1/2)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int VAL = 0; /* variable glabale partagée */


void* Thread1 (void* arg)
{
 int i;
            for(i=0;i<100;i++ ){
                    VAL=VAL+1;
           printf("thread 1: %d \n", VAL);
          }

          pthread_exit(NULL);
}


void* Thread2 (void*arg)
{
           int i;
            for(i=0;i<100;i++ ){

                    VAL=VAL+1;
               printf("thread 2: %d \n", VAL);
          }

          pthread_exit(NULL); /* Fin du thread */
}
```

Exemple 3: deux (2) threads partageant une variable (2/2)

```
int main (void)
{

/* Déclaration de variable de type thread */
     pthread_t t1;
     pthread_t t2;

/* Création et lancement des threads 1 et 2 */
          pthread_create (&t1, NULL,Thread1, (void*)NULL);
          pthread_create (&t2, NULL, Thread2, (void*)NULL);

/* Attendre la fin des threads pour terminer le  main */
     pthread_join (t1, NULL);
     pthread_join (t2, NULL);

/* Fin Normale du programme */
     return 0;
}
```