

# Les processus

Un processus est un programme en cours d'exécution.

Un programme est donc un **élément passif** (un ensemble d'octets sur le disque) tandis qu'un processus est un **élément actif** (un programme en cours d'exécution).

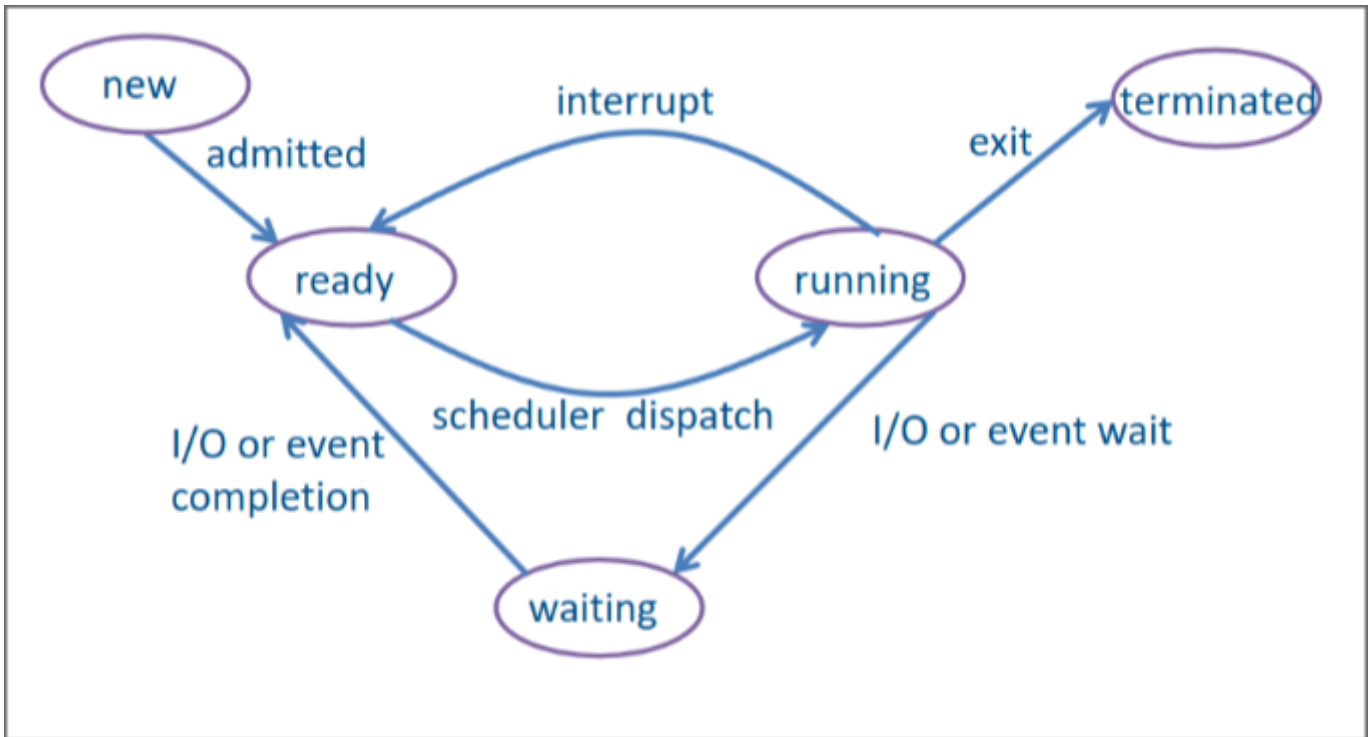
## Que comporte un processus ?

- Le code du programme
- Le program counter (à quel instruction on est dans le programme, qui permet de savoir quelle sera la suivante) et les registres
- La pile (stack) et les données du programme

## Informations concernant le processus

- PID (process ID) qui est l'identifiant du processus
- PPID (parent process id) qui est l'identifiant du processus parent
- Priorité du processus
- Temps CPU : temps consommé au CPU
- Tables des fichiers
- Etat du processus

## Etat



- **new** correspond à un programme qui a été sélectionné pour être démarré, ses instructions ont été recopiées en mémoire par l'OS et un nouveau processus y a été attaché mais pas encore exécuté, son contexte d'exécution et ses détails n'ont pas encore été préparés.
- **ready** le processus a été créé et dispose de toutes les ressources pour effectuer ses opérations
- **running** le processus a été choisi par le scheduler pour tourner, il va donc exécuter ses instructions jusqu'à écoulement du temps imparti. Si il a besoin de plus de ressource, il passe dans l'état *waiting*, si il a terminé son exécution il passe en état *terminated* sinon il peut encore passer en *ready* si un processus de plus haute priorité arrive.
- **waiting** le processus est en attente d'un événement (exemple appui d'un bouton ou écoulement d'un certain temps) ou de ressources (exemple lecture de disque). Le processus ne peut rien faire pour l'instant.
- **terminated** une fois que le processus est terminé (ou a été tué), il libère la totalité des ressources qu'il a détenues.

Vous pouvez avoir plus d'information sur ce sujet en [consultant ce site](#).

## Pour exécuter plusieurs processus

Le système alterne très vite entre les différents états pour donner l'illusion que plusieurs processus s'exécutent en même temps.

En somme on garde en mémoire les processus, le **scheduler** va choisir les processus à exécuter; lorsqu'un processus est en attente un autre processus va être sélectionné pour être exécuté. Le but du scheduler est de maximiser l'utilisation du CPU.

# Le scheduler

Le scheduler va sélectionner le processus à exécuter, c'est lui qui va alterner entre les différents états de chaque processus.

Le scheduler utilise un algorithme précis et il doit être le plus rapide possible.

Le scheduler classe les processus selon leur type :

- Processus CPU (calculs)
- Processus E/S (I/O, entrée sortie)

On va toujours vouloir privilégier les processus entrée-sorties, qui sont ceux qui dialoguent avec l'utilisateur et qui vont donner l'illusion que les choses s'exécutent en même temps.

## Changement de contexte

Pour changer de processus on doit pouvoir sauvegarder le contexte (les données) du processus précédent.

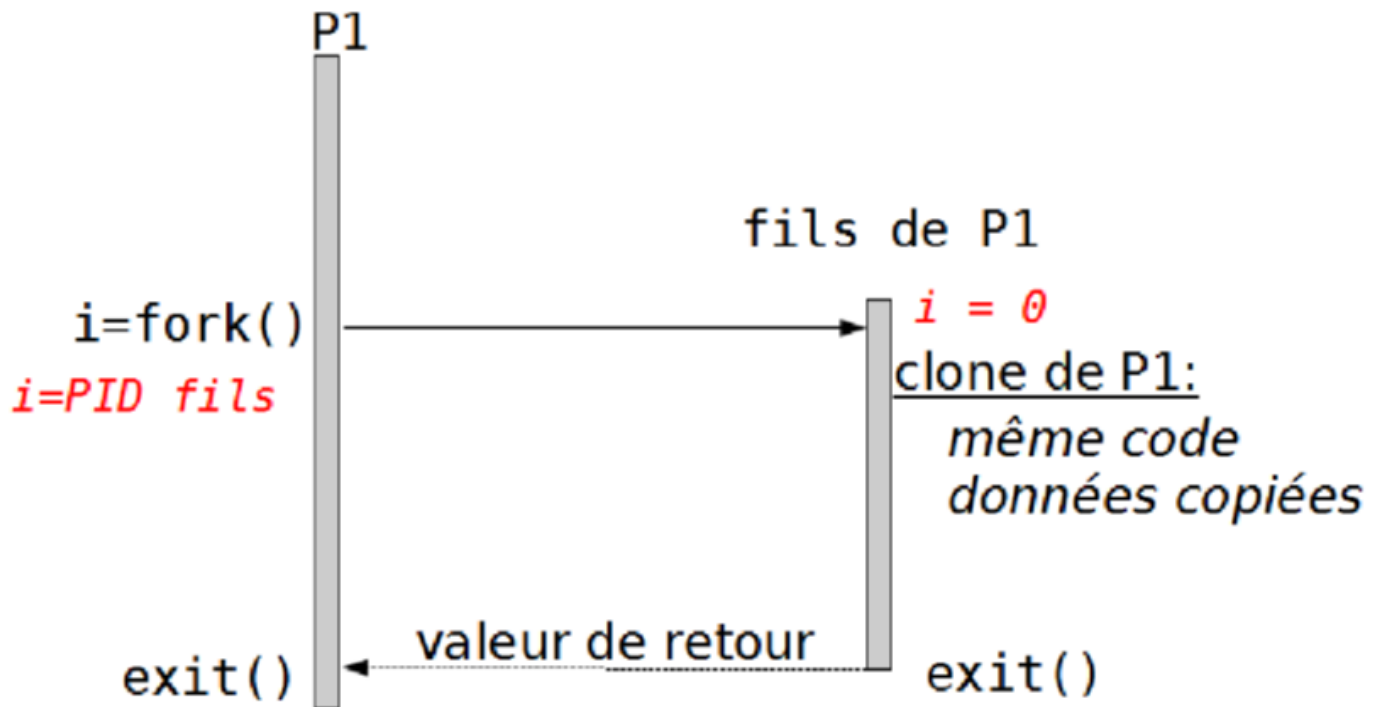
Le système va donc sauvegarder toutes les informations du processus pour pouvoir le redémarrer plus tard.

Ensuite le scheduler va sélectionner un autre processus et en charger les informations/contexte pour le démarrer.

Il va ainsi faire cela tout le temps pour alterner entre tous les processus en attente, prêts et en cours pour maximiser l'utilisation du CPU et donner l'illusion que tout fonctionne en même temps.

## Création d'un processus (fork)

# Lors du fork() ...



Pour créer un processus on utilise l'appel système *fork*. Le processus créé par un fork est appelé le processus *fils*, et le processus qui a créé le *fils* est appelé le *père*.

Le processus *fils* est un clone de son *père*, toutes les données du premier sont recopiées dans le fils.

La fonction `fork()` en C va retourner un entier :

- `-1` si une erreur est survenue (comme souvent en C, une valeur négative veut dire qu'une merde s'est passée)
- `0` pour le processus fils
- Le PID du fils pour le processus père

## Exemples en C

### Exemple simple

Voici un autre exemple :

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int main (void)
```

```

{
    /* Variable pour stocker la réponse du fork */
    pid_t pid;

    /* Fork et mise du résultat dans la variable */
    pid = fork();

    /* Si le pid est 0, alors c'est le fils qui lit l'info */
    if (pid == 0) {
        printf("Je suis le processus fils\n");

        /* Si le pid est autre chose, alors c'est le père qui lit l'info */
    } else {
        printf("Je suis le processus père et mon fils est le : %d\n", pid);
    }

    /* Fin des deux processus */
    return EXIT_SUCCESS;
}

```

Va retourner quelque chose comme :

```

Je suis le processus père et mon fils est le : 243328
Je suis le processus fils

```

## Exemple plus complexe

```

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int main (void) {
    /* La valeur de i par défaut est 5 */
    int i=5;
    pid_t ret;

    /* Ce code sera exécuté uniquement sur le père */
    printf("Avant le fork() ... \n");

    /* La valeur de retour sera 0 sur le processus fils, et le pid du fils sur le processus père

```

```

*/
ret = fork();

/* Le code à partir d'ici sera exécuté sur les deux processus */
printf("Après le fork() ... \n");

/* Sur le processus fils, i sera multiplié par 5 */
if(ret == 0) {
    i*=5;

/* Sur le processus père, i sera additionné de 5 */
} else {
    i+=5;
}

/* Le code ici sera exécuté sur les deux processus */
printf("La valeur de i est: %d\n", i);

/* On retourne la valeur de succès d'exécution ce qui va tuer les deux processus */
return EXIT_SUCCESS;
}

```

Va retourner :

```

Avant le fork() ...
Après le fork() ...
La valeur de i est: 10
Après le fork() ...
La valeur de i est: 25

```

## Fin d'un processus

Un processus se termine quand il n'y a plus aucune instruction à exécuter ou lorsque l'appel système `exit(int)` est appelé (cette fonction permet de renvoyer une valeur entière au processus père).

### wait et waitpid

Un processus père peut attendre la mort de son fils à l'aide des fonctions `wait()` et `waitpid()` et peut ainsi récupérer l'entier retourné par le `exit(int)` du fils.

La fonction `wait()` va simplement attendre la mort d'un fils (peu importe lequel) tandis que la méthode `waitpid()` va attendre la mort d'un processus fils déterminé.

Les fonctions `wait` et `waitpid` retourne le pid du fils, il faut donc passer le pointeur d'une variable en argument pour récupérer les valeurs. Voici un exemple :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>

int main(void) {
    char chaine[100+1];
    int compteur = 0;
    pid_t pid_fils;

    /* On crée un nouveau processus */
    switch (fork()){
        /* Si le résultat est -1 c'est qu'il y a eu un problème */
        case -1:
            printf("Le processus n'a pas été créé.");
            exit(-1);

        /* Si on est le processus fils, on demande d'entrer une chaine de caractères */
        case 0:
            printf("Entrez une chaine de caractères : ");
            scanf("%100[^\n]%%c", chaine);

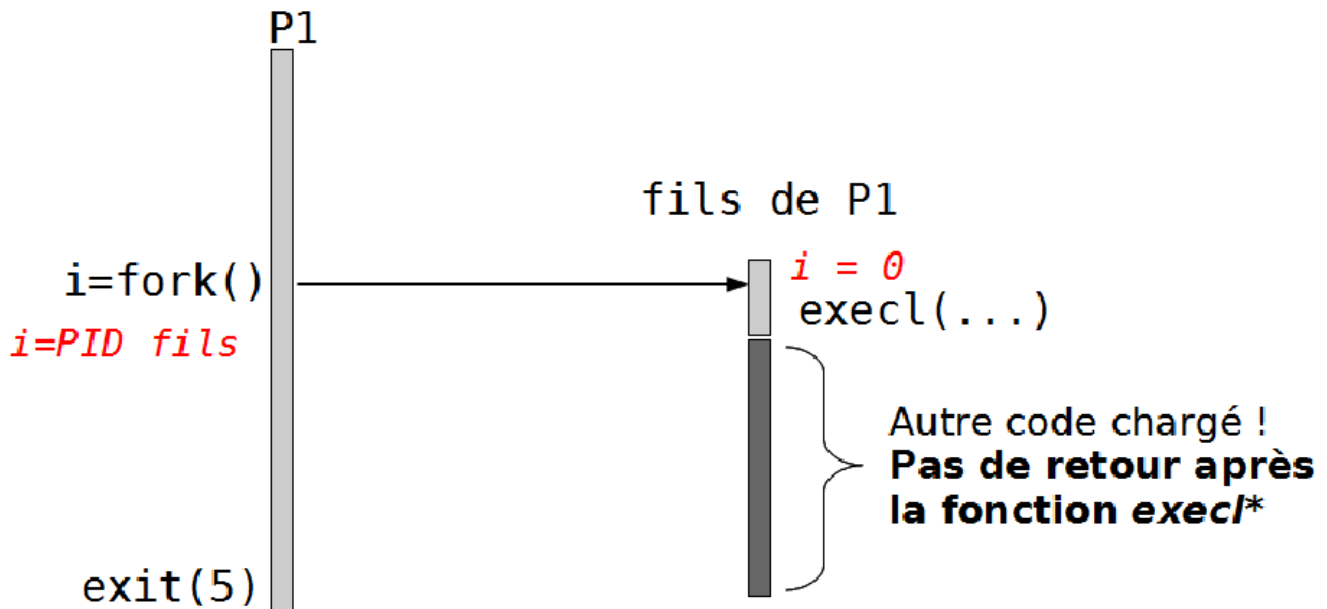
            /* On retourne la longueur de la dite chaine en exit */
            exit(strlen(chaine));

        /* Si on est le processus père, on attends la mort du fils et on récupère la sortie du
        exit dans une variable */
        default:
            /* On stocke le retour du exit dans une variable ainsi que le PID du fils */
            pid_fils = wait(&compteur);
            /* On extrait la longueur de la chaine depuis la sortie du wait avec WEXITSTATUS */
            printf("Enfant %d est mort. Compteur = %d", pid_fils, WEXITSTATUS(compteur));
    }
}
```

```
return EXIT_SUCCESS;
}
```

## execl

`execl` permet d'avoir de charger un autre dans le processus, une fois cette fonction `execl` exécuté le code du processus remplacé est perdu.



La fonction prends en paramètre, deux choses :

- Le **chemin vers le programme**
- Les **arguments du programme** ce qui commence par le chemin du programme (une deuxième fois) et qui termine par un NULL

Voici un exemple d'execl :

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int main(void) {
    /* On crée un nouveau processus avec fork() */
    switch(fork()) {
        /* Si fork retourne -1 c'est qu'il y a eu un problème */
```



```

    case -1: printf("Erreur fork()\n");
             exit(-1);

    /* Si fork retourne 0 c'est que c'est le processus fils, on va donc exécuter la commande
ls avec execl */
    case 0: printf("Je suis le fils\n");
            /* Execl va lancer la commande "ls -l" */
            /* Le premier paramètre est le chemin vers le programme */
            /* Le deuxième paramètre est le chemin vers le programme qui va être passé en
argument */
            /* Le troisième paramètre est le flag "-l" qui sera passé en argument */
            /* Le NULL termine la liste des arguments */
            if(execl("/run/current-system/sw/bin/ls", "/run/current-system/sw/bin/ls", "-l",
NULL)) {

                /* Si le execl retourne -1, c'est qu'il y a eu une merde */
                printf("Erreur execl()\n");
                exit(-2);
            }
            printf("Ce code ne sera jamais exécuté car il est après le execl");

    /* Pour le processus père, on va simplement attendre que le fils ai terminé */
    default: wait(NULL);
            printf("Mon fils a terminé\n");
}

return EXIT_SUCCESS;

/* Le switch n'a pas besoin de break car dans tous les cas, cela se fini par un exit, il ne
peut donc rien y avoir après */
}

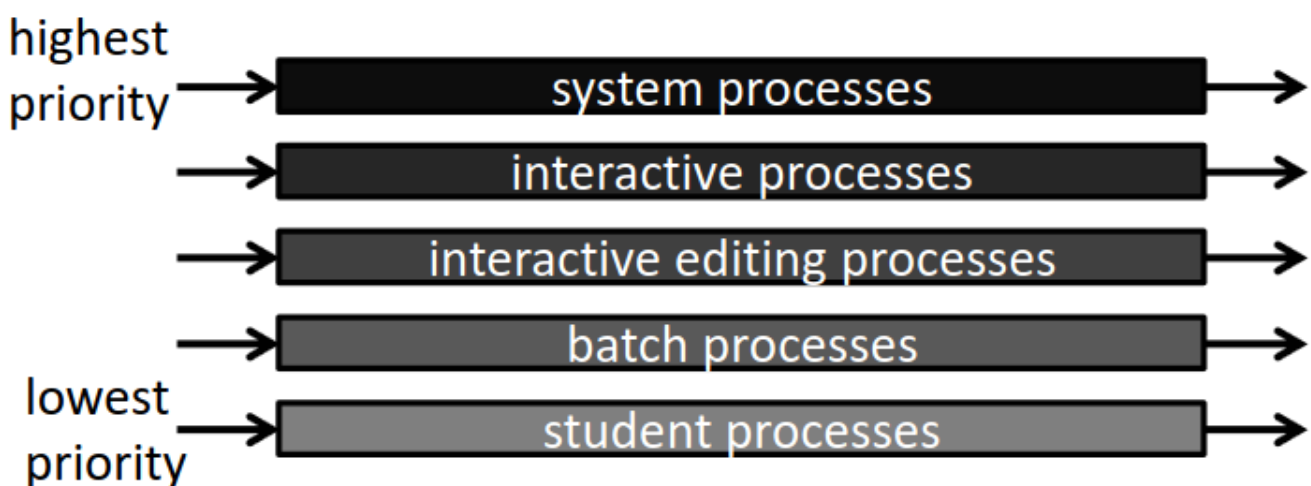
```

# Choix des processus

Le **scheduler** du système d'exploitation doit sélectionner les processus à démarrer pour maximiser l'utilisation du CPU (généralement entre 40% et 90%) pour avoir un débit (le nombre de processus terminés par unité de temps) important (si les processus sont trop long, le débit sera faible).

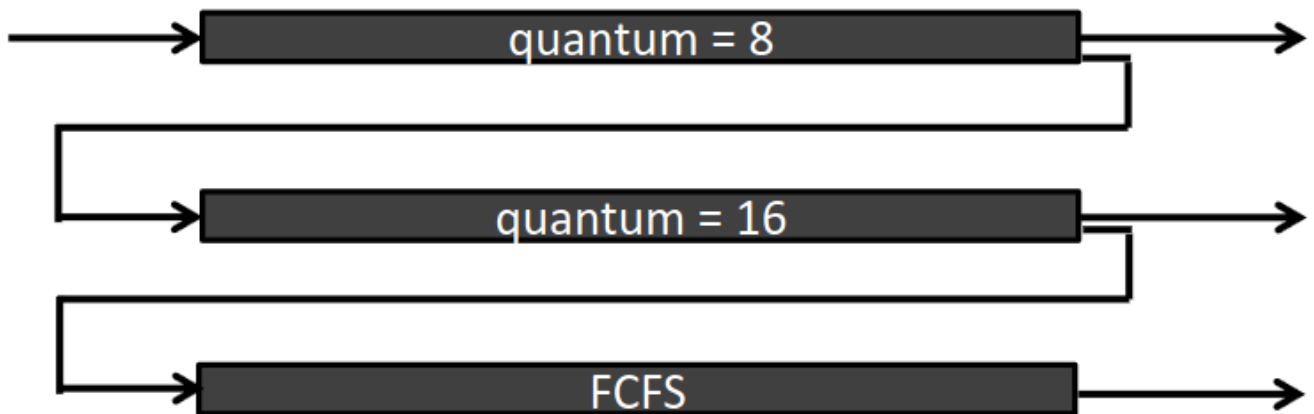
## Algorithmes

- **FCFS (First-Come, First-Served)**, la file ici est une FIFO (first in, first out), c'est l'algorithme le plus simple à implémenter mais il peut être très long, car si le premier processus est long, il ralentit tous les processus qui suivent
- **SJF (Shortest-Job First Scheduling)**, est une amélioration du précédent, il ordonne les processus selon leur durée, ainsi les processus les plus rapides viennent au début et les plus lents à la fin. Cet algorithme est seulement possible si on sait à l'avance la durée du processus, mais aujourd'hui c'est rarement le cas.
- **Priorité**, on tient compte de la priorité d'un processus, ainsi les processus avec la priorité la plus élevée (nombre le plus petit) sont exécutés avant.
  - Cet algorithme peut être préemptif ce qui signifie qu'un processus qui tourne (running) peut être mis sur pause (en état ready) si un processus de plus haute priorité arrive.
  - Cependant cela peut mener à de la famine car les si il y a continuellement des processus de plus haute priorité qui arrive.
    - Ce problème peut être résolu en combinant l'age et la priorité (ainsi les processus ayant attendu trop longtemps passe avant)
- **Round-Robin Scheduling (Tourniquet)**, les processus sont servi dans l'ordre d'arrivée et chaque processus reçoit le CPU pour un temps déterminé (appelé quantum), ainsi on va alterner entre chaque processus avec un temps donné (c'est donc un algorithme préemptif)
  - Si le quantum est trop grand, l'utilisateur·ice aura l'impression que le système lag car rien ne pourra être fait tant que le processus en cours est n'a pas fini son quantum
  - Si le quantum est trop petit, alors on va perdre en efficacité du CPU car beaucoup de l'énergie de calcul sera mise dans le fait d'échanger tous les processus tout le temps.
- **Multilevel Queue Scheduling**, qui s'agit d'avoir de files différentes suivant la nature du processus, une priorité et un mécanisme de scheduling propre est attaché à chaque file, il est ainsi possible d'avoir FCFS et Round-Robin sur des files différentes.



- **Multilevel Feedback Queue Scheduling**, les files sont plus dynamique (un processus n'appartient pas à une file et migrent d'une file à l'autre), chaque file a des caractéristiques précises (quantum, algorithme scheduling, etc).

- Par exemple on peut dire qu'un processus va commencer dans une RR de quantum 8, si il n'a pas fini à la fin de son quantum il passe dans une autre file de priorité moins élevée avec un quantum de 16 et si il n'a toujours pas fini il passe dans une priorité encore moins élevée en FCFS.



## Choix de l'algorithme

Il n'y a pas un seul bon algorithme car chaque algorithme sert à remplir un but précis.

On peut évaluer ces algorithmes selon une certaine utilisation en utilisant des modèles mathématiques, des simulations, des implémentations et des tests.

## Quel algorithme utilisé dans l'OS ?

Sous Windows, c'est un système à 32 niveaux de priorités (préemptif).

Linux en revanche utilise un autre système de scheduling appelé CFS, vous pouvez en apprendre plus dans [cette vidéo](#).

---

Revision #9

Created 27 September 2023 11:44:33 by SnowCode

Updated 2 January 2024 08:42:24 by SnowCode