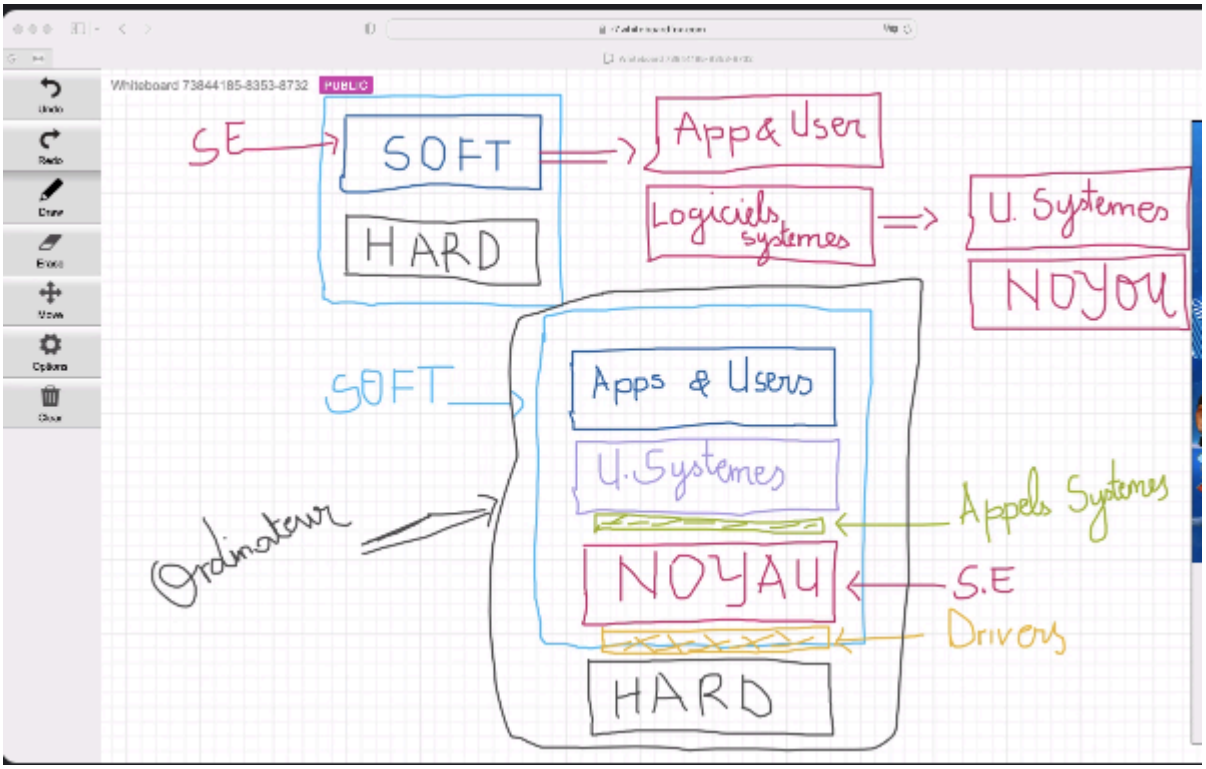
Le but de notre étude est de savoir comment ces processus se comportent au niveau du noyau.
Le processus est une instance de programme en cours d'exécution.
Cette instance est lancée soit par un utilisateur soit par le noyau.
Processus = Programme en cours d'exécution + registres en mémoire + état du processeur
L'espace d'adressage d'un processus est privé.



Résumé du fonctionnement d'un ordinateur

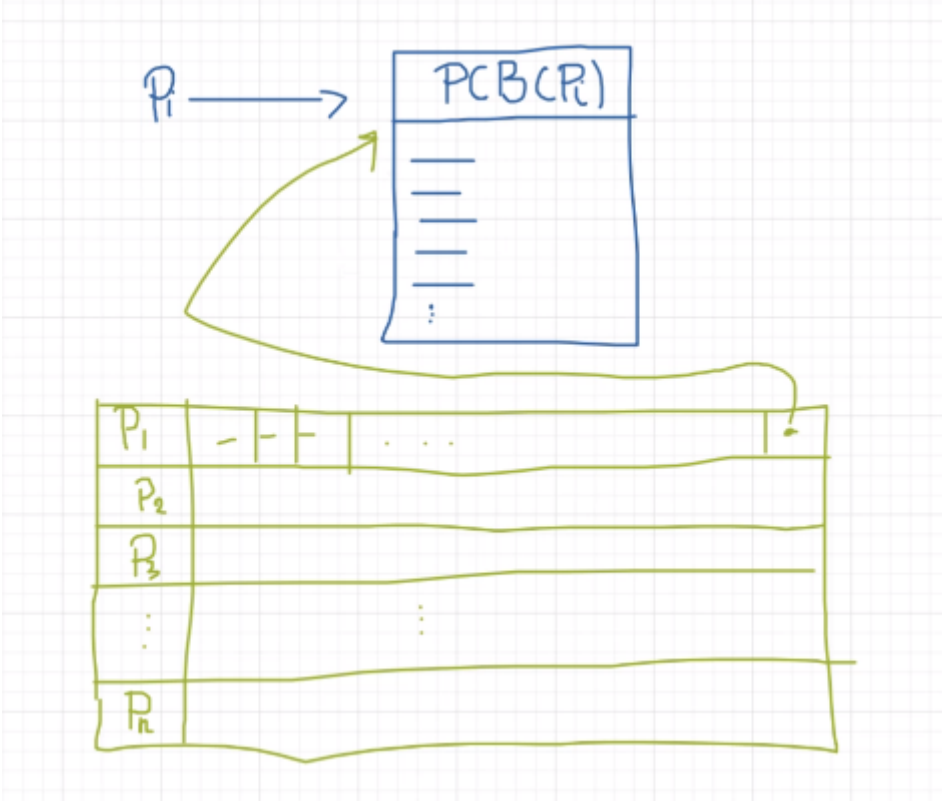
PCB = Process Control Block

- structuré de données créée lors de la création du processus (dynamiquement alloué par le système)
- Il est au processus ce que le CNI est au citoyen
- Le compteur ordinal est une variable qui contient l'adresse de la prochaine instruction d'un programme.
- Contexte mémoire = code et informations sur les variables

Table des processus:

- Est une structure de données
- commune à tous les processus
- chaque entrée contient des informations sur un processus
- contient un pointeur pointant vers le PCB
- contient les segments de données et de code d'un processus

- Stockée dans l’espace mémoire du noyau (autrement dit, hautement protégé, aucun processus ne peut y arriver)



Relation (injective) entre la table des processus et le PCB

Constitution d’un processus:

- Segment de code: contient les instructions du processus, il est en lecture seule
- Segment de données: Contient les données du programme (variables déclarées initialisées/non initialisées)
- Pile: Structure de données. Elle est “inversée” et positionnée “en haut” de l’espace d’adressage comme ça en grandissant elle ne va pas “déborder”. Elle est inversée parce que sinon la pile va croître et atteindre l’espace d’adressage d’un autre processus. La pile croît sur le “tas” (structure de données arborescente). Lorsque la pile occupe tout l'espace, dans ce cas on fera du swapping. Elle contient les fonctions.
- TAS: Contient les données produites lors de l’exécution d’un programme

  L’espace mémoire d’un processus n’est pas contigu, sinon il y aurait beaucoup de fragmentation de la mémoire (ce qu’on essaie d'éviter).

  Espace mémoire est divisé en des parties appelées pages; ces dernières sont adressées, c’est pour cela qu’on parle d’espace d’adressage quand on parle

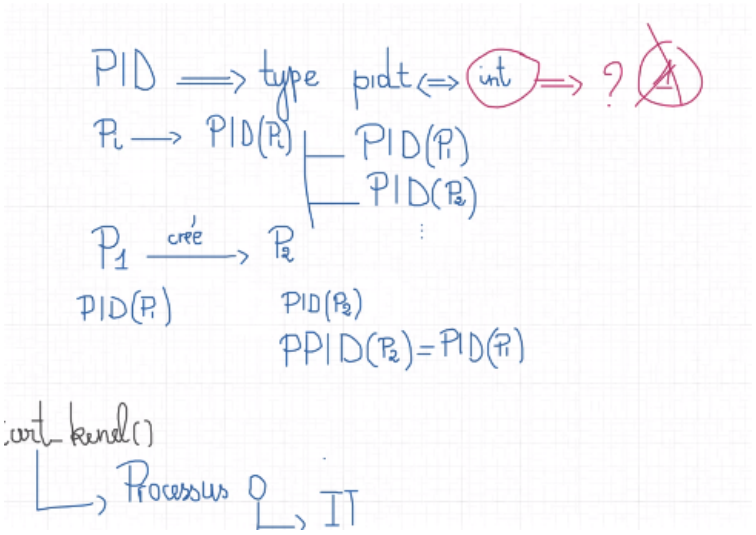
d'espace mémoire. Évidemment ce n'est pas tout l'espace mémoire d'un processus qui peut-être adressé.

Le PID est un entier typé (type pid_t) et codé comme un int, il représente l'identifiant d'un processus.
PID = Process ID
PPID = Parent PID
Quelques conventions de nommage:

- PID(P1) → Correspond au PID de P1
- PPID(P2) = PID(P1) → P2 processus parent de P1

Le principe de la filiation veut qu'un processus est toujours créé à partir d'un autre.
Lorsque le système démarre, start_kernel (fonction du noyau Linux) est lancé (par impulsion électrique), elle va créer le processus 0. Ce processus va:

- Premier processus de la table des processus
- Initialise les structures de données pour le noyau
- Créer un thread (processus léger) appelé init, puis il dort jusqu'à ce qu'il soit réveillé. Il exécute init() pour devenir le véritable processus init.
-



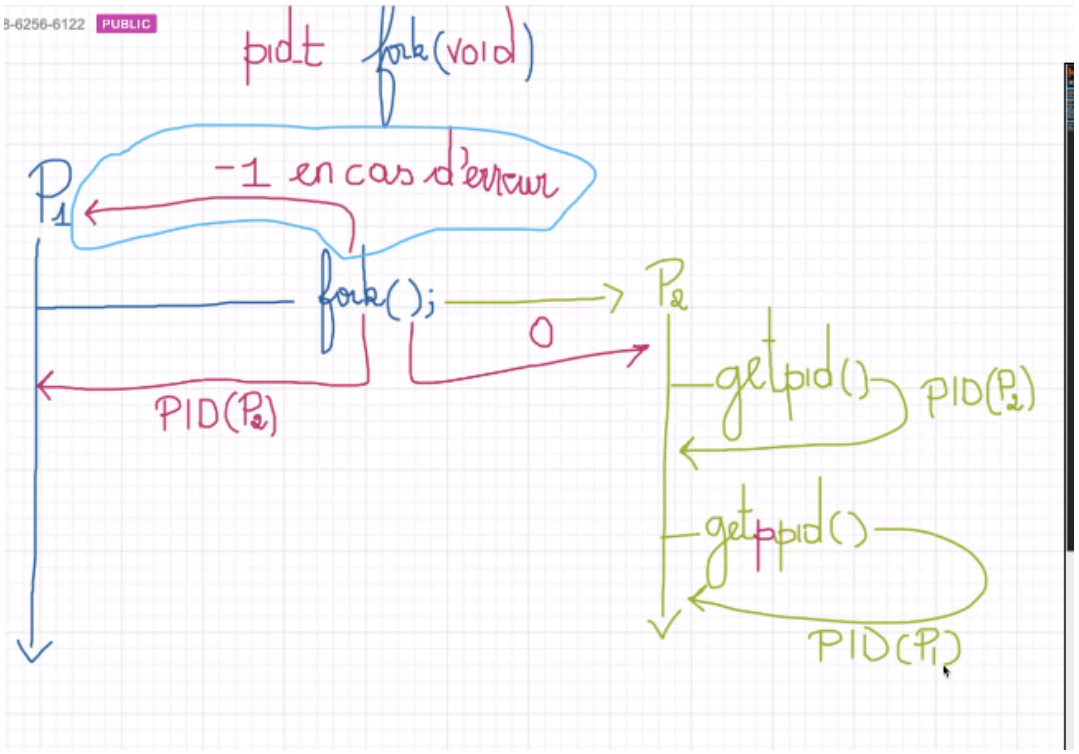
Une session de processus, ensemble de groupes de processus caractérisés d'une certaine manière.

On peut avoir des processus utilisateurs et des processus noyaux (ou systèmes).

Pour toute action que l'on veut faire, il y a un appel système. Et donc pour créer un processus, on utilise l'appel système fork().
Le processus père fait une demande de création de processus et c'est au noyau d'apprécier cette demande.

Cette demande échoue si

- Plus assez de mémoire
- Limite de processus fils autorisés



Lors du fork() le segment de code n'est pas dupliqué.
En programmation classique (mono thread): 1 programme → 1 processus → 1 thread != Programmation parallèle (multithread)

Lors d'un appel fork() le processus fils reçoit automatiquement une copie identique des variables du père.
Toute instruction placée après le fork() sera exécutée par le fils.

Attention, la mémoire copiée signifie aussi que la mémoire Buffer est copiée. Un \n permet d'aller à la ligne mais aussi de vider le buffer. Donc si le buffer n'est pas vidé avant le fork, le fils recevra son contenu. C'est donc une faille système. On peut utiliser fflush() (avec stdin et stdout) pour vider le buffer.

NB: Toutefois, certains attributs de la mémoire ne sont pas copiés:

- Le PID du processus père et celui du grand-père,
- Le temps d'exécution du processus fils (donc par défaut = 0)
- Les verrous du processus père
- La priorité du processus père.

Deux types de terminaisons de processus: normale et anormale.

Elle est normale lorsque le programmeur l'avait prévu ou suite à certaines conditions prédéfinies qui ont été rencontrées.

Elle est anormale lorsqu'elle n'est pas prévue, et il y'a toujours intervention de signal dans ce cas.

CTRL+C envoie un signal SIGINT pour tuer un processus
L'appel système kill

A la mort d'un processus, les ressources utilisées par ce dernier sont libérées.
Ses processus fils deviennent des processus orphelins, et ils seront adoptés par INIT (conséquemment, leurs PPID = 1)
Le processus mort envoie un signal SIGCHILD à son père. Si ce dernier ignore ce signal, le processus mort devient un processus ZOMBIE. Par défaut le père ignore les SIGCHILD sauf si expressément conçu pour ne pas le faire.

exit(x) est un appel système masqué (un frontal). En réalité ce dernier appelle _exit qui envoie un signal SIGCHILD au père + un status entre [0; 255]. Par défaut le status est 0 ou 1 (en fonction du système). -1 correspond à une terminaison anormale, et pour le reste des status, c'est laissé à l'appréciation du programmeur.

Attente de terminaison de processus:

Un processus zombie est un processus dont le PCB, son PID et son entrée dans la table des processus sont encore présents.
Wait force le père à attendre la terminaison de son premier fils. Lorsque ce fils meurt, SIGCHILD + status sont envoyés au père en lui faisant connaître l'identité du fils qui vient de terminer. Grâce à celà, on évite les processus fils zombies.

Si le père fait un wait après la fin d'un processus, ce processus fils sera zombie jusqu'à ce que le principal atteigne les wait.

Si le père fait un wait alors qu'il n'y a aucun processus, il se produit une erreur et rest retourné -1.

wait() retourne le PID du fils qui est mort.

Si la méthode wait() reçoit NULL, alors on ne pourra récupérer les infos sur la mort du processus.

MACROS: permettent à un processus d'analyser la mémoire d'un fils et déterminer par exemple si la terminaison du processus est normale ou anormale. Les macros s'utilisent en général par paires.

- CAS TERMINAISON NORMALE:
  Le macro WIFEXITED est un macro qui permet de savoir si c'est une terminaison normale (valeur de retour booléenne)

  exit() indique une terminaison normale. Son argument permettra au développeur de savoir précisément la raison de la terminaison du processus, et pour faire en faire un traitement adéquat.

  WEXITSTATUS retourne la valeur qui a été passée à la fonction exit() appelée à la fin du processus fils.

- CAS TERMINAISON ANORMALE
  WIFSIGNALED permet de savoir si c'est une terminaison anormale afin qu'on puisse récupérer le status
  WTERMSIG retournera le numéro du signal ayant causé la terminaison du processus.

  WIFSTOPPED permet de savoir si le processus a changé d'état, càd s'il a été stoppé pendant son exécution.
  WSTOPSIG retournera le numéro de signal causant le stop des processus.

  Le processus principal crée un fils dont le seul rôle est de créer un petit fils, puis il meurt. Le petit fils devient donc orphelin et sera adopté par INIT. INIT crée une boucle pour gérer les wait() sur les processus et par conséquent le petit fils qui aurait été zombie à sa terminaison sera nettoyé. L'avantage d'un processus orphelin est qu'il ne devienne jamais zombie.

  Certaines tâches ne doivent se faire que juste avant la mort d'un processus. pour cela on utilise la fonction atexit().