Système d'exploitation: théorie

Romain Deleuze

27 décembre 2019

Table des matières

Chapitre 1

Lecture 1: intro

Lundi 23 Septembre 2019 8 :3

1.1 Structure de l'OS

1. Sert a traduire ce que le CPU est capable de faire face à la demande de l'utilisateur.
2. L'ordinateur est capable de faire 3 choses :
3. Structure : Cpu (registres et compteur d'instructions), mémoire (en binaire), sortie INPUT/OUTPUT (périphériques).
   - Chercher des infos dans la mémoire -> mettre dans un registre
   - Faire des opérations dans un registre (Comparaisons, arithmétiques)
   - Renvoyer le résultat dans la mémoire ou vers INPUT/OUTPUT(sorties lentes)
4. Les instructions et le contenu de la mémoire sont en binaire
5. l'OS ajoute une couche, interface de plus haut niveau entre les composants machines et l'homme ou entre les programmes et les composants

1.2 Assembleur

1. Premier language entre le CPU et les composants
2. Était fait en fonctions de l'architecture des CPU

1.3 Concurrency

1. Les différents programmes doivent se protéger les uns des autres lors de l'exécution (multiutilisateur)
2. Faire attention à la mémoire
3. L'OS fourni cette protection et vérifie les instructions

1.4 Prompt

1. L'autre façon d'intéragir avec l'OS sans GUI
2. Grâce à une série de commandes (ls, dir, cd)

1.5 Fonctions principales

1. Machine étendue
   - Machine virtuelle (exemple : librairie web scraping)
   - Conviviale et facile d'emploi (GUI)
2. Manageur de ressources
   - Gère les ressources, se protège lui-même.
   - Première instruction -> Chercher l'OS en lui-même
   - Partage les ressources de façon cohérente (Empêche la corruption de mémoire entre les programmes)

1.5.1 Machine étendue

1. Opération INPUT/OUPUT
   - Création de drivers/pilotes -> Soulage l'OS (exemple : echo, retour à la ligne)
   - I/O lente (entrée utilisateur -> temps en microsecondes -> milliers d'instructions perdues)
   - Tout est transformé en binaire
   - Fait le lien entre l'hardware et l'OS
2. Manageur de ressources Les ressources sont utilisées par des applications et des processus d'arrière-plan. Il faut les partager :
   - Partage dans le temps : ordonnancement, le CPU va être utiliser à tour de rôle par les différents processus
     - si attente d'avoir fini -> problème : attente des autres processus.
     - Temps dédié en millisecondes.
   - Partage dans l'espace : Gestion de la mémoire -> Mémoire virtuelle avec la RAM et l'HDD Passer de la mémoire d'un processus à l'autre = swap -> problème pour ordinateur temps réel(temps défini pour répondre à un processus). Un système de priorité entre les processus est défini dans la file d'attente

1.6 Classement des OS

1. Mainframe (salle entière)
2. Serveur (+ petit que mainframe)
3. Multiprocesseurs (plusieurs CPU en parallèle)

4. Personnel (Ordinateurs lambdas)
5. De poche (portable)
6. Embarqués
7. Temps réel

1.6.1 Mainframe

1. Machine capable de générer un nombre énorme d'I/O simultanés
2. Nécessite un OS adapté : Unix
3. 3 types de services :
   — Batch -> Programme interactif qui tourne tout seul en arrière-plan (backup, calculs)
   — Transaction processing -> Transaction = suite d'instructions qui n'a de sens que si elle est exécutée complètement (transaction bancaire) -> Incomplète -> Revenir en arrière = rollback
   — Timesharing -> Partage de temps entre les processus

1.6.2 Serveur

1. Moins grand et puissant que les Mainframe
2. Multiutilisateur
3. Partage de ressources -> hardware et/ou software

1.6.3 Multiprocesseurs

1. Séparation en core et multicore
2. Permet aux ordinateurs de devenir de plus en plus petits

1.6.4 Personnel

1. Différents OS vendus avec les PC
2. Sont énormément plus petits que les serveurs

1.6.5 De poche

1. ARM pour les chips plus petites
2. Arriver aux téléphones

1.6.6 Embarqués

Micropuces, Cartes bancaires, IoT

1.6.7 Temps réel

Machine qui réagit dans un temps bas (programme qui gère les appels téléphoniques -> (carte prépayé, peu de temps pour regarder si la personne est abonnée ou non -> time slice given) number portability (garder le numéro après un changement d'opérateur)

Chapitre 2

Histoire

Lecture 2: Histoire des OS

Lundi 30 Septembre 2019 8 :3

2.1 Concepts incontournables

1. Les processus
2. L'espace addressable
3. Le système de fichier -> Organiser le stockage
4. Les appels système -> Mode utilisateur/système

2.1.1 Processus

1. Plusieurs programmes peuvent tournés sur la même machine en même temps en leur allouant des ressources du processeur.
2. 2 fois le même programme = 2 processus différents.
3. Daemon = programme en tâche de fond sous manière séquentiel et non pas parallèle (sauf dans le cas d'architecture multiprocesseur)

2.1.2 L'espace addressable

1. Les programmes ont besoin de mémoire pour s'exécuter
2. Elle est limitée -> utilisation du disque en tant que mémoire virtuel (plus de mémoire que ce qu'il y en a)
3. Grande partie du programme dans la RAM et le reste dans le disque
   - si la mémoire RAM est remplie -> envoi de la mémoire la plus vieille dans le disque
   - Le swap (fait par l'OS) -> prend du temps (machine à temps réel ne peut pas swap)
   - besoin d'une table pour savoir où se trouve la mémoire dans le disque lié à la mémoire RAM

2.1.3 Système de fichiers

1. La racine du disque -> root = /
2. APFS = système d'Apple
3. ext2/3/4 = Linux
4. NTFS, ReFS = Windows

2.1.4 Appels système

1. Permet au système le contrôle d'accès aux ressources
2. Mode de fonctionnement des processeurs récents :
   — Mode noyau = accès total (permissions proche de l'OS)
   — Mode utilisateur = accès restreint
   — Superviseur = permissions de l'OS
3. Programmes : mode utilisateur
   — Sécurité
   — Opération à risque -> Appel à l'OS
4. Undo = appel système grâce à un registre

2.2 Structure des OS

1. Différentes manières de conception d'un OS
   — Monolithiques
   — En couches
   — A micro-noyau
   — Machine virtuelles

2.2.1 Monolithiques

1. Les plus répandus
2. Collection de procédure appelant des routines systèmes
3. L'ensemble de ces routines forme un exécutable le noyau = kernel (d'où les appels systèmes pour passer en mode noyau)

fonctionnement

— Envoi des paramètres (passer un paramètre à une fonction) -> print("texte") = print(paramètre)
— Appel au noyau -> Envoi des paramètres vers la **stack qui contiendra l'adresse**
— Analyse les paramètres
— Sélection de la routine
— Exécution de la routine
— Retour en mode utilisateur

2.2.2 En couches

1. Le système est construit en couches ayant chacune une fonctionne propre :
   — Allocation du processeur aux différents processus (multiprogramming)
   — Gestion de la mémoire
   — Communication processus-console d'un opérateur (multiusers) -> chaque utilisateur a sa console
   — Gestion des I/O
   — Programme des utilisateurs
   — Opérateur

2.2.3 Micro-noyau

1. Noyaux monolithiques de plus en plus volumineux
2. Appel système courant ou rare dans le noyau
3. Solution :
   — Noyau : Contient les quelques routines courantes
   — Routines rares : dans des programmes systèmes -> réduit le Nb de lignes dans le noyau

2.2.4 Machines virtuelles

1. Plusieurs machines contenues dans une seule machine
2. IBM 360 -> Un moniteur de machine virtuelle avec un ensemble de machines virtuelles

2.3 Historique de systèmes

Nouvelles intégrations -> nouvelles générations

2.3.1 1ère génération

1. Pas d'OS
2. 1956 : I/O system
3. 1er coprocesseur mathématique
4. Début des supers ordinateurs
5. Mémoire à torre de ferrite
6. Concepteur = programmeur = utilisateur
7. Langage machine

2.3.2 2ème génération

1. 1959 : IBM 7090 : Transistor -> moins de mécanique = moins de panne
2. 5 fois plus rapide
3. Plus fiable
4. Personnel dédié à chaque tâche
5. Premier ordinateurs personnels
6. Premiers OS
7. Programmation à niveau plus élevé (Fortran)
8. Toujours des périphériques en retard (mémoire perforée) -> télétype

2.3.3 3ème génération

Le circuit intégré -> Toujours plus petit et plus puissant (Jack Kilby et Robett Noyce)

2.3.4 Multiutilisateurs

1. Compatible time sharing system -> multiutilisateur -> 3 en même temps

2.3.5 MULTIX

Ancêtre d'UNIX

2.4 UNIX

UNplexed Information and Computing Service -> codé en C

— Ken Thompson
— Dennis Retchie
— Brian Kerrigan

1. 1970 : 1ère version
2. 1973 : Langage C -> Distribution libre
3. 1975 : Version 6 -> Base commune
4. 1978 : BSD -> Berkley university
5. 1984 : System III, IV et V
6. 1991 : Linux

2.4.1 POSIX

Normalisation en 1990 -> Norme POSIX = couche de base avec les mêmes paramètres et la même base. Passage d'un programme d'un ordi à l'autre. = Portable Operating System Interface

2.5 Micros

1. Permet de passer aux ordinateurs personnels
2. 1971 Premier microprocesseur = Intel 4004
3. 1975 Premier ordinateur = Altair -> ne sert à rien
4. 1976 Premier Apple (Bojniak)
5. IBM PC : Aout 1981 -> Architecture ouverte -> Appels à des gens spécialisés pour chaque composant et utilisation de MS-DOS
6. Premier programme -> Lotus (tableur pour la gestion)
7. Bill Gates crée son propre OS avec Tim Paterson -> MS-DOS
8. Compétition entre MacOS et MS-DOS
9. 1992 : Abandon d'IBM dû à la compétition
10. Retro/Reverse Engenieering = décompilation des programmes

2.6 Windows

1. Invention de la souris par Kerox, Bill Gates y a cru et à lancer Windows par la suite
2. 1995 : Triomphe de Windows sur MacOS

Versions :

— 3 branches principales -> Windows 1 à 3.11 -> NT2000 -> 95, 98, Me

— Fusion XP -> CE -> RT

Chapitre 3

Processus et threads

Lecture 3: Processus et threads

Lundi 14 Octobre 2019 8 :30

3.1 Processus

Le premier processus lancé est l'OS Chaque programme lancé est un processus

3.1.1 Les processus

1. Programme = ligne de code, il a des variables propres à lui, des appels aux fonctions, son contexte
2. L'OS va donner le contexte au programme et toutes ses variables qui vont avec
3. Processus = code en exécution

3.1.2 Types de processus

1. Les processus qui tournent en background
2. Les programmes interactifs

3.1.3 Création des processus

1. Initialisation système
2. Processus parent
3. Requête
4. Batch -> processus qui a peu d'interaction avec l'utilisateur et se lance à un moment précis (ex : backup)

Sous Unix/Linux

1. Unix : base processus
2. Processus 0
3. Processus 1 : init
4. /etc/iinitab -> iinitab = table des processus à lancer
5. dameon = processus qui attendent un event (ex : inetd -> gestion des ports internet)
6. fork -> programme qui lance un autre programme (copie de lui-même)
7. gettty = programme utilisateur (prêt à recevoir un login)
8. process login -> A la connection d'un utilisateur -> va voir dans le fichier passwd pour le mdp -> ok -> lance un shell (BASH)

Sous Windows

1. Une fonction WIN32 -> CreateProcess se charge à la fois de la création du nouveau processus et de sa personalisation
2. Des paramètres lui sont fournis
3. Démarre les programmes mentionnés

3.1.4 La fin d'un processus

1. Normal exit -> voluntary
2. Error exit -> voluntary
3. Fatal error -> unvoluntary
4. Killed by another process -> unvoluntary (commande : kill -> kill -9) -> signal qui est envoyé au processus (kill, siguser...)

exemple : division par 0 -> impossible -> détecter par l'OS -> arrêt du programme

Processus normal exit

1. Fin de programme
2. Intervention utilisateur
3. Linux : appel exit
4. WIndows : ExitProcess
5. Libération des ressources
6. PCB effacé (Process control block)

3.1.5 État des processus

QUESTION EXAMEN

1. Lorsque un programme est réveillé il se retrouve -> prêt
2. Passé de l'état prêt à Élu -> élection (dépend de l'ordre d'importance du programme)
3. Élu -> le programme à la main sur le CPU
4. CPU veut donner du temps aux autres programmes -> suspend le programme qui avait la main
5. exemple : Le process demande un nom à l'utilisateur -> le process est bloqué -> le CPU le libère
6. si autre process est prêt -> le CPU l'élu -> le process à la main sur le CPU
7. lors de l'input de l'utilisateur -> process débloqué -> repassage à l'état prêt

États particulier

1. Initialisation
2. Terminé
3. Zombie -> toujours des programmes fils -> ne sait pas ce qu'il fait là
4. Swappé -> process qui est envoyé vers la swap pour travailler
5. Préempté -> processus qui arrête de travailler pour donner du temps de process aux autres
6. Utilisateur
7. Noyau -> fonction root

préemption

1. Process qui passe la main à un autre dépendant de sa priorité
2. Transition de l'état Élu à l'état prêt -> permet aux programmes de tourner en "même temps" -> Distribution du CPU aux différents processus

3.1.6 Les processus et les threads

1. Première partie de la mémoire lue (données initialisées) -> section .text
2. Variables non initialisées chargées -> section .data
3. Les variables de travail situées dans les fonctions -> dans la stack

Initialisation d'une fonction dans la **stack**

1. Premières adresses -> adresse de retour
2. Placement des paramètres de la fonction par dessus dans la stack
3. Paramètres locals par dessus dans la stack
4. Libère les paramètres envoyés dans la stack après le return

Allocation dynamique de la mémoire -> dans le HEAP

exemple : allocation d'un buffer d'un certain nombre de bytes

fonction main

fonction main avec (argc -> compteur d'arguments | argv -> arguments passés à la fonction) Variable **ENV** -> contient toutes les variables utilisateur

3.2 threads

Toute les données initialisés et non initialisés (section .data et .text) sont gardées entre les programmes car elles sont les mêmes pour chaque copie du programme

3.2.1 Exemple

Programme pour compter le nombre de places

```
nbplaces = 10
thread1 -> nbplaces - 5
thread2 -> nb places - 3
thread3 -> nbplaces - 4
```

1. nbplaces peut donc être négatif dû à une utilisation en parallèle -> utilisation du **LOCK** pour la variable.
2. La variable ne pourra pas être touchée par une autre thread pendant le travail d'une autre
3. **INTERLOCK** -> blockage des variables dont une thread a besoin par une autre thread -> process bloqué

3.2.2 Comparaison processus - thread

Sans threads

1. 1.5ms recopie intégrale du processus
2. Communication par le **noyau -> plus lent, nécessite une programmation spécifique**

Avec threads

1. 0.5ms création par recopie de certains éléments
2. Communication par la **mémoire commune -> plus rapide, moins d'effort de programmation**

Chapitre 4

Communication inter-processus

4.1 Notion de processus

Un processus est un programme en cours d 'exécution auquel est associé un environnement processeur (CO, PSW, RSP, registres généraux) et un environnement mémoire appelés contexte du processus.

4.2 Notion de ressources

Une ressource logique (variable) ou une ressource matérielle (processeur, périphérique) caractérisée par :

1. un état : libre/occupée
2. point d'accès (qui peut y toucher) -> un seul processus peut toucher à la variable

4.2.1 Allocation des ressources

Trois étapes :

1. Allocation
2. Utilisation
3. Restitution

4.2.2 interblocage

2 programmes ont besoin de 2 ressources chacun
Une ressource est bloquée par un des 2 programmes -> interblocage -> freeze du système

4.2.3 Famine

Attente infinie -> on ne sait pas quand elle fini

4.2.4 Conditions nécessaires à un interblocage

1. Besoin de ressources critiques (qui ne peuvent pas être partagées et qui ont un seul point d'accès)
2. Occupation et attente : Processus qui occupe au moins une ressource attend d'acquérir une ressource d'un autre processus
3. Pas de réquisition : Les ressources sont libérées sur une seule volonté des processus les détenant
4. Attente circulaire : Au moins un processus P1 attend une ressource détenue par un autre processus P2, P2 attend lui-même un ressource détenue P1

4.2.5 L'exclusion mutuelle

Les processus disposent d'un espace d'adressage, inacessible par les autres processus Pour communiquer entre eux les processus utilisent des systèmes de communication tels que les **pipes**

4.2.6 Section critique

Partie d'un programme dont l'exécution se peut s'entrelacer avec d'autres programmes
protection de la variable utilisée par le premier programme pour éviter qu'une thread l'utilise -> **LOCK**

Généralisation de ce problème

Se présente dans :

1. Base de données
2. Le partage des ressources
3. Les automates
4. Le hardware
5. Le développement

4.2.7 Solution matérielle

Masquage des interruptions

4.2.8 Solution algorithmique

Attente active : Toutes les Xms check de la ressource libre ou non = Polling

4.3 Solution fournie par le système

Un sémaphore S peut être vu comme un distributeur de jetons

1. L'opération INIT(S, VAL) fixe le nombre de jetons initial
2. L'opération P(S) attribue un jeton au processus appelant si il en reste sinon -> bloque le processus
3. L'opération V(S) restitue un jeton et débloque un processus de S.L si il en existe un

Jeton = **mutex** -> mutuellement exclusif un seul point d'accès
Avant de lire et de s'allouer une place -> blocage -> Au déblocage du jeton distribue un nouveau jeton

4.3.1 Allocation de ressources

Accès a un ensemble de n ressources critiques
Si ensemble de processus -> accès a un ensemble de n ressources critiques

Allocaton - utilisation - restitution

1. Allouer une ressource = prendre un jeton Res
2. Rendre une ressource = rendre un jeton Res

Plan d'exécution

1. Sémaphore Res initialisé à N (1 jeton par ressource)
2. Allocation : P(Res)
3. Utilisation Ressource
4. Restitution V(Res)

4.3.2 Lecteur-rédacteur

1. Contenu du fichier doit rester cohérent : **Pas d'écriture simultanée**
2. Lectures doivent être cohérentes : **Pas de lecture en même temps que les écritures**

4.3.3 Producteur-consommateur

Exemple : Gestion FIFO -> buffer réseau TCP)

1. L'un produit dans les cases et un autre lit
2. Un producteur ne doit pas produire si le tampon est plein
3. Un consommateur ne doit pas faire de retrait si le tampon est vide
4. Producteur et consommateur ne doivent jamais travailler dans une même case

4.3.4 Résumé

1. Les exécutions de processus ne sont pas indépendantes : les processus peuvent vouloir communiquer et accéder de manière concurrente à des ressources
2. Les sémaphores S est un outil système de synchronisation assimilable à un distributeur de jeton et manipulable par seulement trois opérations atomique P(S), V(S) et Init(S)
3. Il existe plusieurs schémas typiques de synchronisation à partir desquels sont élaborés des outils de communication entre processus
   l'exclusion mutuelle
   les lecteurs/rédacteurs
   les producteurs/consommateurs

4.4 Les signaux

Information atomique envoyée à un processus ou à un programme
Processus peut réagir de différentes manières à un signal
Outils permettant la gestion de processus par l'utilisateur, le système ou autre processus
3 différentes réactions d'un processus face à un signal :

1. Il est dérouté vers une fonction spécifique
2. Le signal est ignoré
3. Peut provoquer l'arrêt du procesus (avec on sans génération d'un fichier core dump)

4.4.1 Différents signaux

1. SIGINT -> provoqué par le caractère associé à intr sur le clavier de l'utilisateur
2. SIGQUIT -> provoqué par le caractère quit associé au clavier de l'utilisateur
3. SIGILL -> Instruction illégale
4. SIGFPE -> Erreur arithmétique (ex : division par zéro)

5. SIGKILL -> Signal de terminaison, son gestionnaire ne peut pas être remplacé
6. SIGSEGV -> Violation mémoire
7. SIGCHLD -> terminaison d'un fils

Lecture 5: Interprocessus

Lundi 04 Novembre 2019 8 :30

4.5 Les tuyaux/pipes

Utilisé surtout dans les systèmes UNIX -> permet à un groupe de processus de communiquer avec un autre groupe de processus
Un coté lit et et un coté écrit -> chaque coté = descripteur de fichier ouvert soit en lecture ou en écriture
Le coté de lecture ne peut rien faire tant que l'autre n'a rien écrit -> permet la synchronisation
exemple : Lecture des fichiers dans un répertoire = ps -ef | more
"more" ne s'exécutera qu'à la fin de la commande précédente, dans ce cas-ci "ps -ef"

4.5.1 Problèmes

Exemple : Plusieurs processus qui lit et un qui écrit : L1, L2 et E1
La lecture d'une même donnée n'est pas possible dans ce cas-ci car si L1 la lit, la donnée disparait pour les autres.
Besoin du même nombre de pipes de chaque côté.
Communication seulement **unidirectionnelle** et ont besoin d'un **lien de parenté** (Même utilisateur)

4.5.2 Tuyaux nommés

2 pipes qui n'ont pas de lien de parenté permettent de connecter -> orienté hardware (toujours sur la même machine)
exemple : echo coucou > fifo (envoi de la commande "echo coucou" au process fifo -> rien pour lire "coucou")
cat fifo -> echo coucou > fifo (lecteur alloué -> affichage du message "coucou")

4.5.3 Sockets

D'un applicatif à un autre
Communication **bi-directionnelle** et permettant de fonctionner sur **la même machine**
Pour communiquer, l'utilisation d'un socket sur l'une des machines peut être utilisé pour la communication -> Port du socket doit être plus grand que 1023(Well known sockets)
Aucun lien de parenté et de pouvoir communiquer vice-versa.

Chapitre 5

Ordonnancement

5.1 Problème

Quand arrêter un processus ?

5.2 Objectif

Maximiser le taux d'occupation du CPU
Minimiser le temps de réponse des CPU

5.3 Les critères

Temps d'attente -> somme de tout le temps passé en file prêt
Temps de réponse
2 mécanismes :

1. Mécanismes non préemptifs : Qui n'interrompt pas le fonctionnement d'un processus : seul l'attente sur une entrée sortie et la fin d'un processus peuvent provoquer l'appel de l'ordonnanceur
2. Les mécanismes préemptifs : Qui peuvent interrompre le fonctionnement d'un processus, ce qui suppose la mise en place d'un timer

Débit = Throughput : nombre de processus qui s'exécutent complètement dans l'unité de temps (à maximiser).
Temps de rotation = turnaround : le temps pris par le processus de son arrivée à sa fin (à minimiser).
Temps d'attente : attente dans la file prêt (somme de tout le temps passé en file prêt) (à minimiser).
Temps de réponse (pour les systèmes interactifs) : le temps entre une demande et la réponse (à minimiser).

5.4 Ordonnanceur et répartiteur

L'ordonnanceur : Crée la fil d'attente de processus selon des critères bien spécifiques (politique d'ordonnancement) -> envoi des PCB dans la file d'attente(Process control block)
Répartiteur : Donne les différents processus aux différents CPU ou coeur d'un même CPU
Préemption d'un processus : Fin d'un processus renvoyé à l'ordonnanceur pour être remis dans la file d'attente

5.5 Politiques d'ordonnancement

Premier arrivé, premier servi (FIFO)
Plus court d'abord (priorité aux programmes qui prennent le moins de temps)
Par priorité constantes (tout le temps la même priorité)
Par tourniquet (round robin)
Par files de priorités constantes multiniveaux avec ou sans extinction de priorité

5.5.1 Premier arrivé, premier servi

Exemple : Trois processus -> P1(24UT), P2(3UT) et P3(3UT)
P1 n'ayant pas de temps d'attente -> s'exécute
P2 attend la fin de P1 -> 24UT et s'exécute
P3 attend donc 27 UT avant de s'exécuter
Si P1 était mis en dernier le temps d'attente est modifié drastiquement
Le système n'est donc pas prévisible

5.5.2 FSJ : Le plus court d'abord

Sélection du processus nécessitant le moins de temps d'exécution
Méthode non préemptive :

1. exemple : P1(7UT), P2(4UT), P3(1UT) et P4(4UT)
2. Arrivé de P1 en premier -> s'exécute pendant que P2, P3 et P4 arrivent
3. P3 étant le plus court -> s'exécute
4. P2 s'exécute ensuite et P4 par après
5. T = (3 + 6 + 7) / 4 = 4

Méthode préemptive :

1. P1 -> s'exécute
2. P2 arrive pendant l'exécution de P1 -> P2 plus court que P1 -> P2 prend le processeur avant que P1 n'ai fini
3. P3 arrive pendant l'exécution de P2 -> P3 plus court que P2 -> P3 s'exécute et se termine
4. P2 ayant déjà une partie de son exécution faite il est donc plus court que P4 -> P2 se termine

5. P4 étant plus court que le reste de l'exécution de P1 -> Fin de P4 et de P1 par la suite