

Architecture et système d'exploitation

Omar RIFKI

omar.rifki@univ-littoral.fr

Basé sur les cours d'Andrew Tanenbaum
(Vrije Universiteit) et d'Eric Ramat (ULCO).

Laboratoire d'Informatique, Signal et Images de la Côte d'Opale (LISIC)
Université du Littoral - Côte d'Opale (ULCO)



Déroulement du module

- ▶ **Cours:** 6 x 2 heures [1er semestre: 3 cours]
- ▶ **Travaux pratiques:** 14 x 3 heures [1er semestre: 7 séances]

Notation

- ▶ pour chaque séance de TP:
 - ▶ un compte rendu à envoyer sur Moodle à la fin de la séance
 - ⇒ une note
- ▶ **note du module = 50% examen + 50% moyenne des notes TP**

Objectifs

- ▶ étudier les composants essentiels d'un système d'exploitation (SE)
- ▶ comprendre l'interaction entre matériel et logiciel via le SE
- ▶ (TP) développer des fonctions de base d'un SE
- ▶ (TP) écrire du code applicatif basée sur la couche système

Environnements de travail des TPs

- ▶ **prérequis:** développement en C/C++
- ▶ travail uniquement sous Linux
- ▶ XV6: implémentation d'Unix version 6 (noyau)

Plan

- 1 Introduction
- 2 Processeur et assembleur x86
- 3 Unix/Linux
- 4 Processus
- 5 Fichiers

Plan

1 Introduction

2 Processeur et assembleur x86

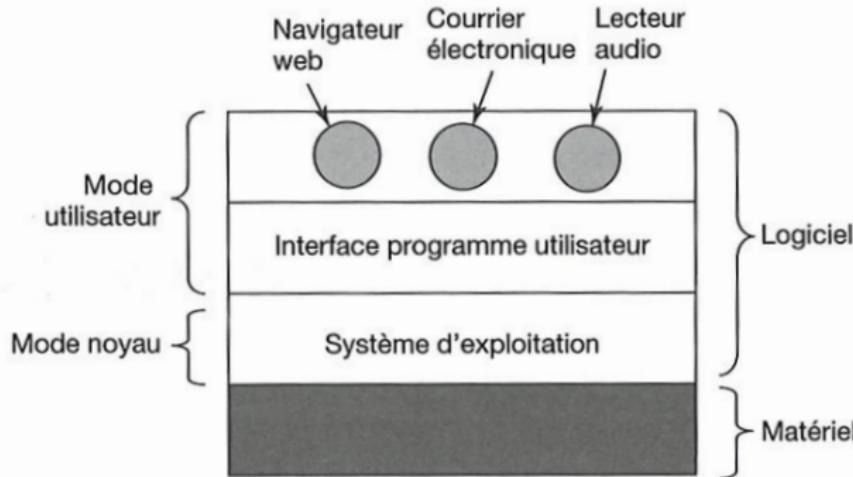
3 Unix/Linux

4 Processus

5 Fichiers

Introduction: Qu'est-ce qu'un SE?

- ▶ couche logiciel entre matériel et applications dont le but est de gérer tous les périphériques et ressources et de fournir aux programmes utilisateurs une interface simplifiée avec le matériel.

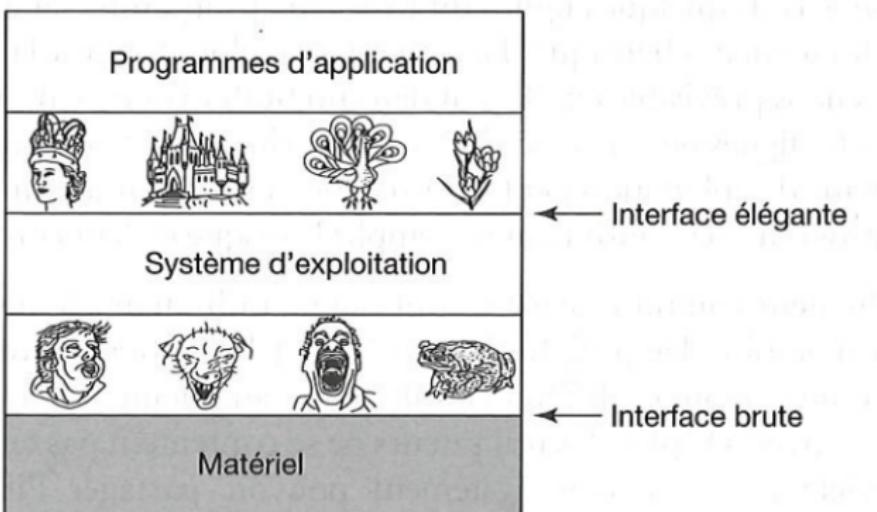


©Tanenbaum

Exemples: Windows 95/98/ME, Windows NT/2000/Vista, UNIX et ses clones (System V, Solaris, FreeBSD..), Linux.

Introduction: Qu'est-ce qu'un SE?

- ▶ couche logiciel entre matériel et applications dont le but est de gérer tous les périphériques et ressources et de **fournir aux programmes utilisateurs une interface simplifiée avec le matériel.**
 - ▶ cache les détails de fonctionnement du matériel
 - ▶ fournit au programmeur des abstractions cohérentes et élégantes



Introduction: Qu'est-ce qu'un SE?

- ▶ couche logiciel entre matériel et applications dont le but est de **gérer tous les périphériques et ressources** et de fournir aux programmes utilisateurs une interface simplifiée avec le matériel
 - ▶ gestionnaire de la mémoire et les entrée-sorties (clavier, écran,...)
 - ▶ **multiplexage** (partage) des ressources en temps et en espace

Introduction: Historique

Quatre générations:

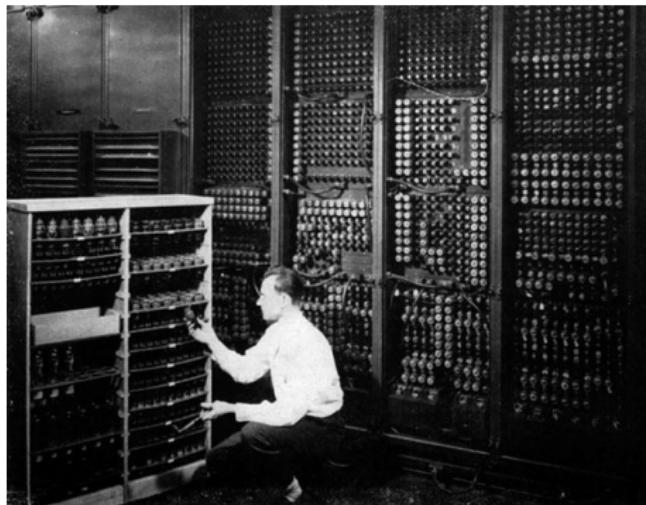
- ① (1945-55) tubes électroniques
- ② (1955-65) transistors et systèmes par lots (*batch systems*)
- ③ (1965-80) circuits intégrés et multiprogrammation
- ④ (1980-présent) ordinateurs personnels

Introduction: Historique

Quatre générations:

① (1945-55) tubes électroniques

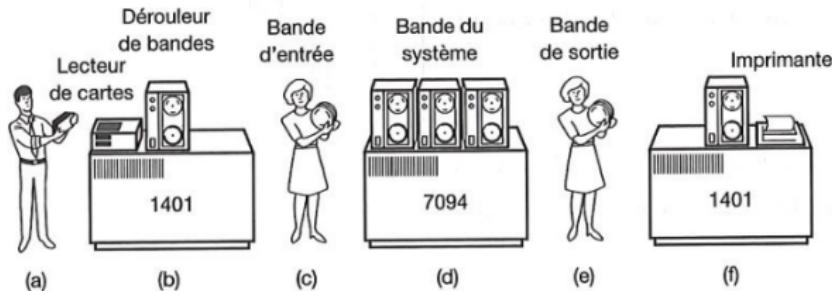
- ▶ language machine uniquement
- ▶ aucun SE



Introduction: Historique

Quatre générations:

- ② (1955-65) transistors et systèmes par lots (*batch systems*)
 - ▶ **language:** FORTRAN et assembleur
 - ▶ **SE:** FMS (Fortran Monitor System) et IBSYS de IBM 7094



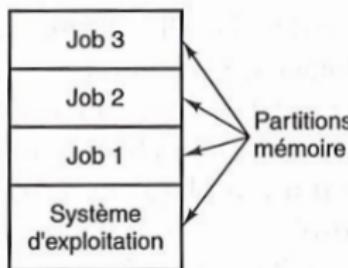
- (a) Les programmeurs apportent leurs cartes au 1401.
- (b) Le 1401 écrit l'ensemble des jobs sur bande.
- (c) L'opérateur apporte la bande au 7094.
- (d) Le 7094 effectue les calculs demandés.
- (e) L'opérateur transfère les résultats sur le 1401.
- (f) Le 1401 imprime les résultats.



Introduction: Historique

Quatre générations:

- ➊ (1965-80) circuits intégrés et multiprogrammation
 - ▶ **SE**: OS/360 de IBM 360
 - ▶ projet **MULTICS (MULTIplexed Information and Computing Service)** entre MIT, Bell labs et General Electric ⇒ naissance de UNIX



multiprogrammation avec trois jobs en mémoire

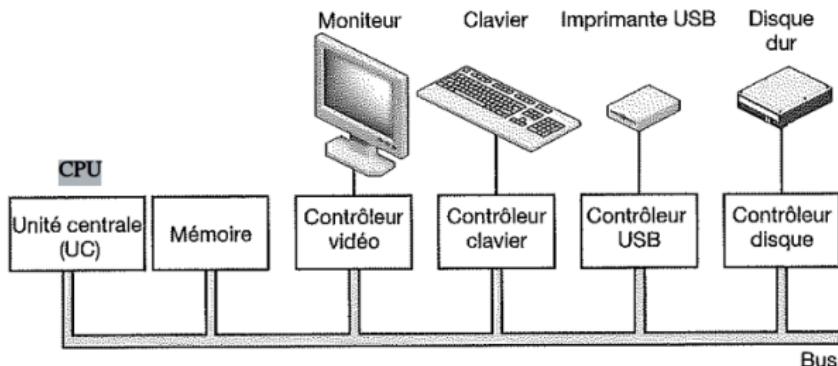
Introduction: Historique

Quatre générations:

- ④ (1980-présent) ordinateurs personnels
 - ▶ en 1974, Intel a sorti le 8080, premier processeur 8 bits généraliste
 - ▶ Gary Kildall a développé un SE **CP/M** pour les ordinateur Intel
 - ▶ IBM a équipé ses PC avec MS-DOS (*Microsoft Disk Operating System*) suite à un contrat avec Bill Gates
 - ▶ Doug Engelbart de Standford a inventé le concept d'**IHM** (interface homme-machine), repris par Xerox PARC ensuite Apple (Steve Jobs)
 - ▶ ...

Introduction: Structure matérielle d'un ordinateur

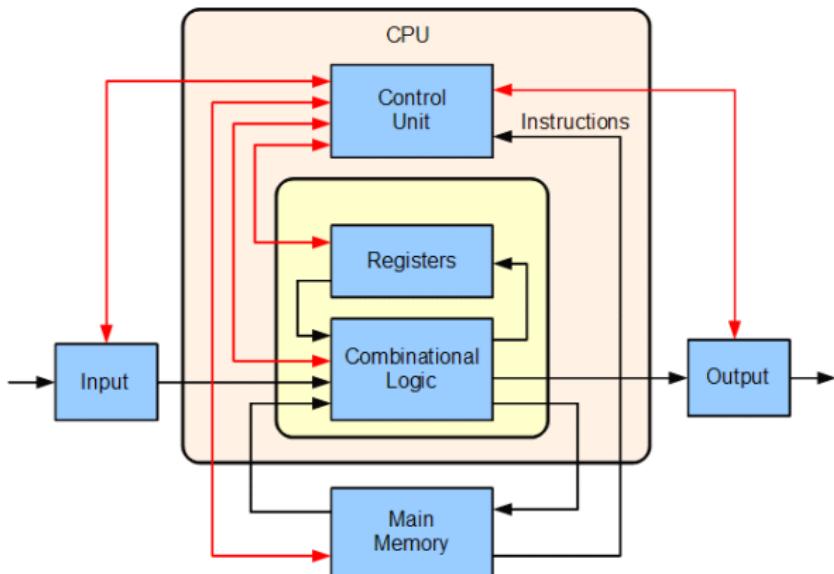
- ▶ **Architecture de Von Neumann:** l'ordinateur est structuré en quatre parties distinctes: **unité arithmétique et logique (UAL), unité de contrôle (UC), la mémoire et les entrée-sorties.**



©Tanenbaum

Introduction: Processeur

CPU: unité centrale de calcul (*central processing unit*).



(CC) Wikipedia



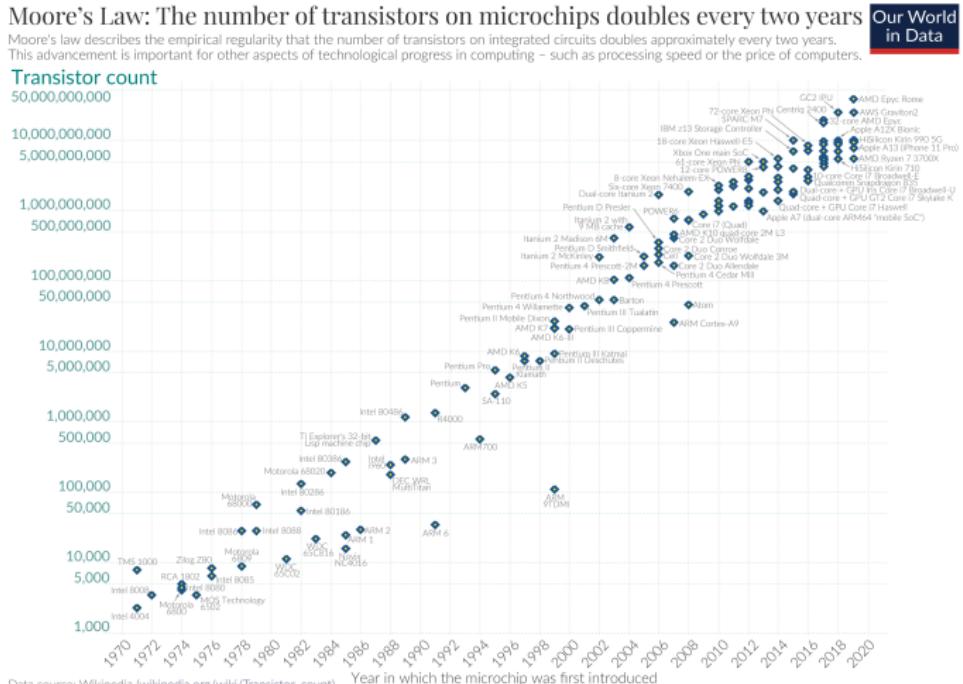
ARM: , Intel: , AMD:



Introduction: Loi du Moore

Doublement du nombre de transistors tous les 18 mois.

⇒ Evolution de la puissance de calcul et de la complexité des ordinateurs



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
OurWorldInData.org – Research and data to make progress against the world's largest problems.

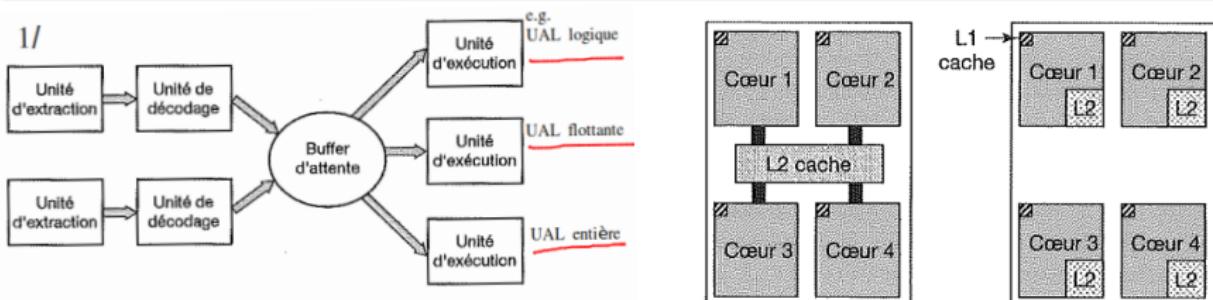
Licensed under CC-BY by the authors Hannah Ritchie and Max Roser.

Introduction: Multithreading et circuits multicoeurs

Montée en puissance

- ▶ le premier processeur Intel (4004): 4 bits; 2250 transistors; 10⁴ nm
- ▶ aujourd'hui, l'un des derniers processeurs AMD (Ryzen 9 3900X): 64 bits; jusqu'à 64 coeurs; environ 10 milliards de transistors; 12 nm

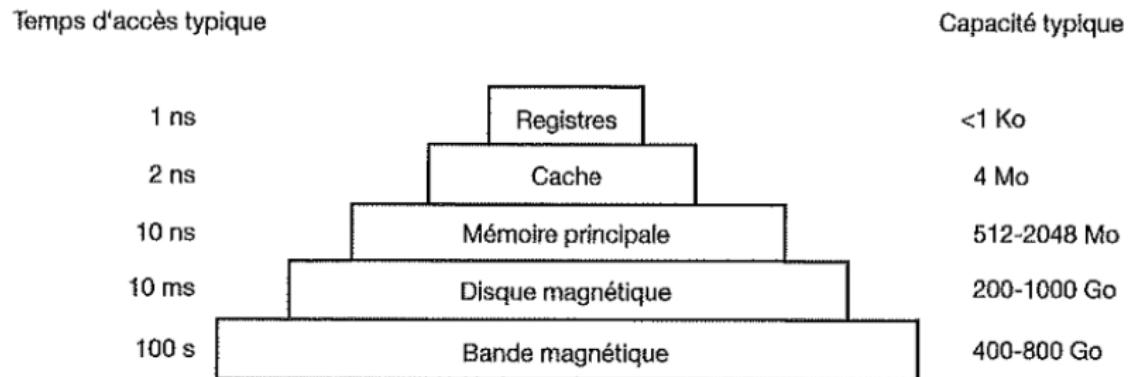
⇒ 1/architecture **superscalaire**, 2/multithreading, 3/circuit **multicoeurs**.



Introduction: Mémoire

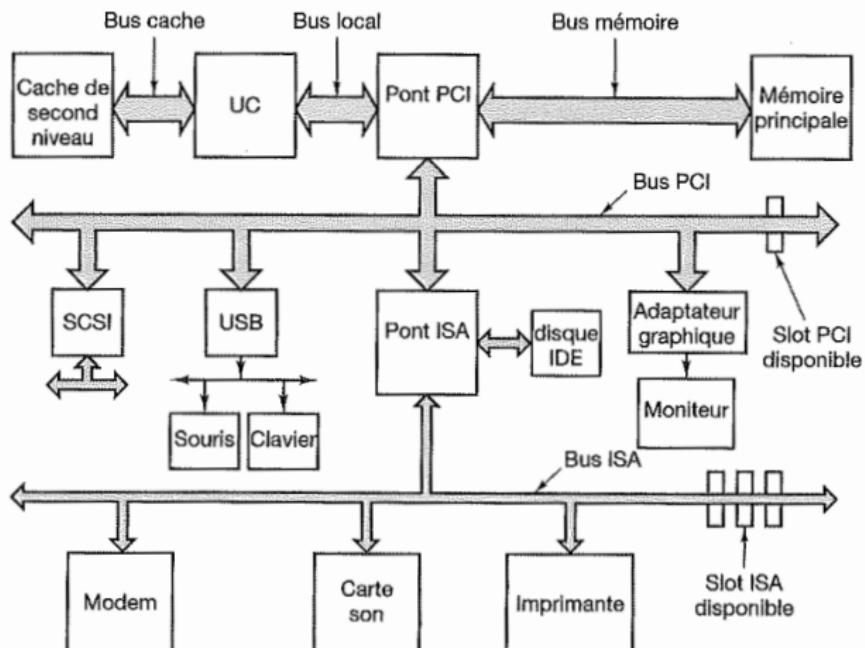
Dans l'idéal, la mémoire devrait être: 1/rapide, 2/disponible en grande quantité et 3/peu onéreuse.

Un découpage hiérarchique classique de la mémoire:



Introduction: Bus

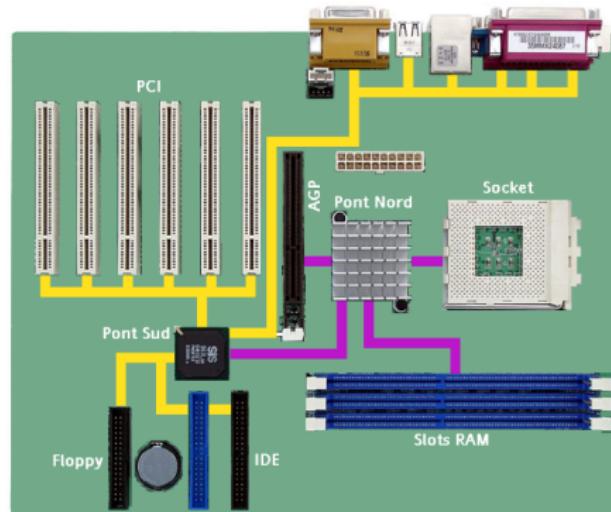
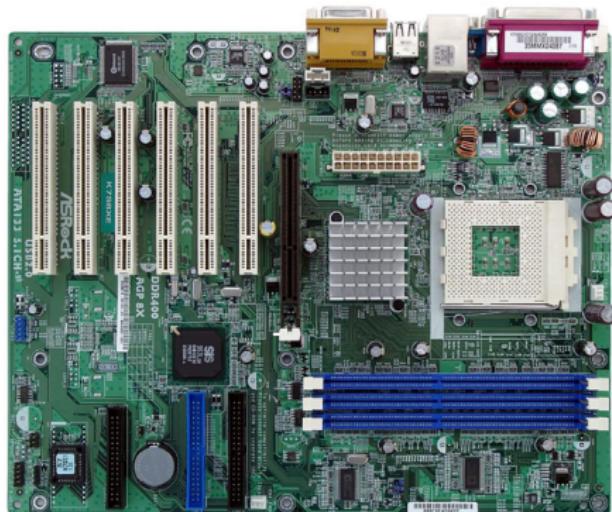
Exemple: structure d'un système Pentium.



©Tanenbaum

Introduction: Carte mère et ses composants

Concrètement ...



Introduction: Quelques mots clés

Quelques mots clés

- ▶ processus, “thread”, concurrence, ordonnancement (*scheduling*) et coordination
- ▶ espace d'adressage
- ▶ protection, isolation, partage et sécurité
- ▶ communication et protocole
- ▶ stockage persistant, transactions, consistance et résilience
- ▶ interfaces aux périphériques

Introduction: Fonctions

Concrètement ...

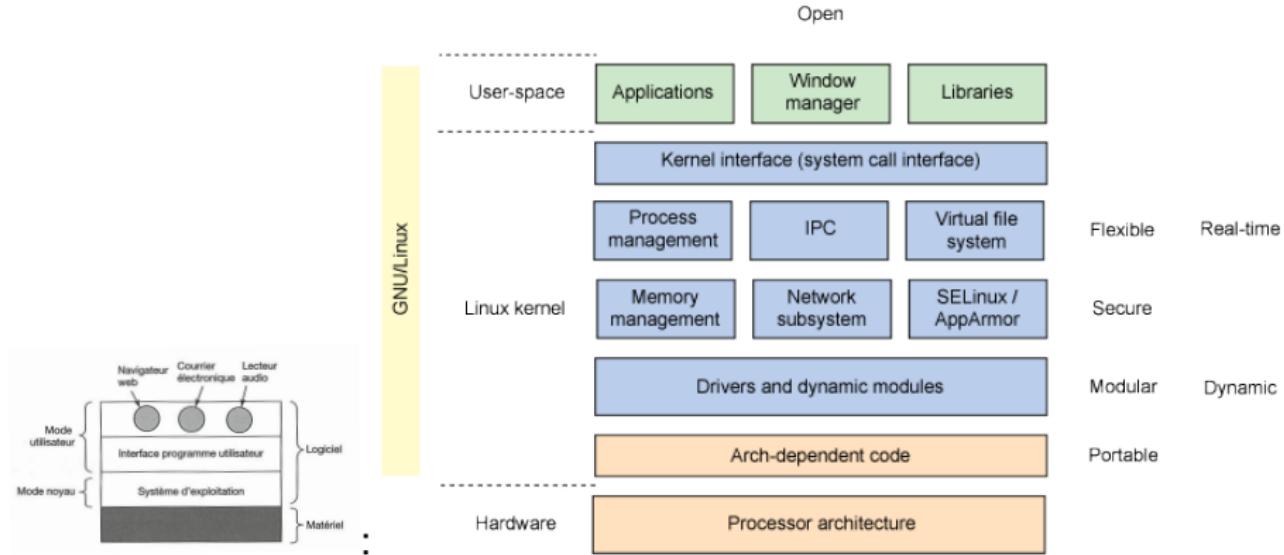
Ce que fait un SE

- ▶ gérer la mémoire et les entrée-sorties (clavier, écran, disque, ...)
- ▶ ordonner les “tâches” sur le ou les processeurs

Et plus encore ?

- ▶ gérer les utilisateurs (pour les systèmes multi-utilisateurs)
- ▶ gérer les fichiers et les systèmes de stockage
- ▶ gérer les communications (réseaux)
- ▶ gérer le multitâche

Introduction: Architecture



©IBM - M. Tim Jones

Appels systèmes

L'interaction entre SE et programmes utilisateurs est définie par un ensemble **d'appels systèmes** fournit par le SE

Thèmes abordés

- ① **processeur et assembleur**
- ② **historique Unix/Linux** et commandes Unix
- ③ **processus:** gestion, section critique, ordonnancement et synchronisation, communication inter-processus
- ④ **système de fichiers**

Processeur et assembleur

On ne peut pas parler de système d'exploitation sans notions de processeur et d'assembleur !

Plan

1 Introduction

2 Processeur et assembleur x86

3 Unix/Linux

4 Processus

5 Fichiers

Objectifs

Objectifs

- ▶ revoir l'assembleur et le langage machine
- ▶ faire le lien entre le langage C, la compilation et l'assembleur
- ▶ comprendre l'implantation d'un programme en mémoire
- ▶ entrevoir la relation avec le système d'exploitation (appels systèmes)

x86

On va s'intéresser dans ce cours uniquement à l'assembleur pour processeur de la famille x86

Des architectures différentes (**x86 (Intel, AMD); PowerPC (IBM); 68x00 (Motorola); SPARC (Sun); ARM; MIPS...**) ont des instructions différentes

Langage Assembleur

Définition

Langage haut niveau

(Ada, C, C++, etc)

Langage d'assemblage (asm)

Mnémoniques associées au langage machine

Langage machine

Binaire en mémoire qui forme un exécutable

```
c := c + 3;  
...
```

Compilation

```
mov eax, [esi]  
add eax, 3  
...
```

Assemblage

```
01010111  
10001110  
...
```

Assembleur est une *représentation du langage machine*

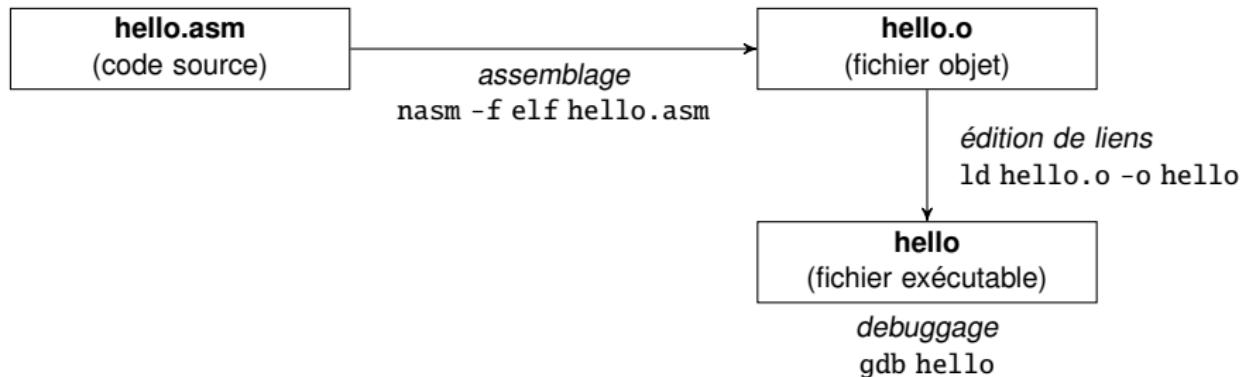
Instruction en language asm: **code op** | opérandes (Ø, valeurs, registres, @mémoires)

Langage Assembleur

Procédure d'assemblage

En 3 phases:

- 1 Saisie du code assembleur avec un éditeur de texte
- 2 Assemblage du code
- 3 Edition des liens



Avantages

- ▶ Accès à toutes les possibilités de la machine
- ▶ Vitesse d'exécution du code
- ▶ Faible taille du code généré
- ▶ Meilleure connaissance du fonctionnement de la machine

Inconvénients

- ▶ Temps de codage plus long
- ▶ Fastidieux
- ▶ Pas de structures évoluées
- ▶ Garde-fous minimaux
- ▶ Absence de portabilité

Langage Assembleur

Processeurs 80x86

x86

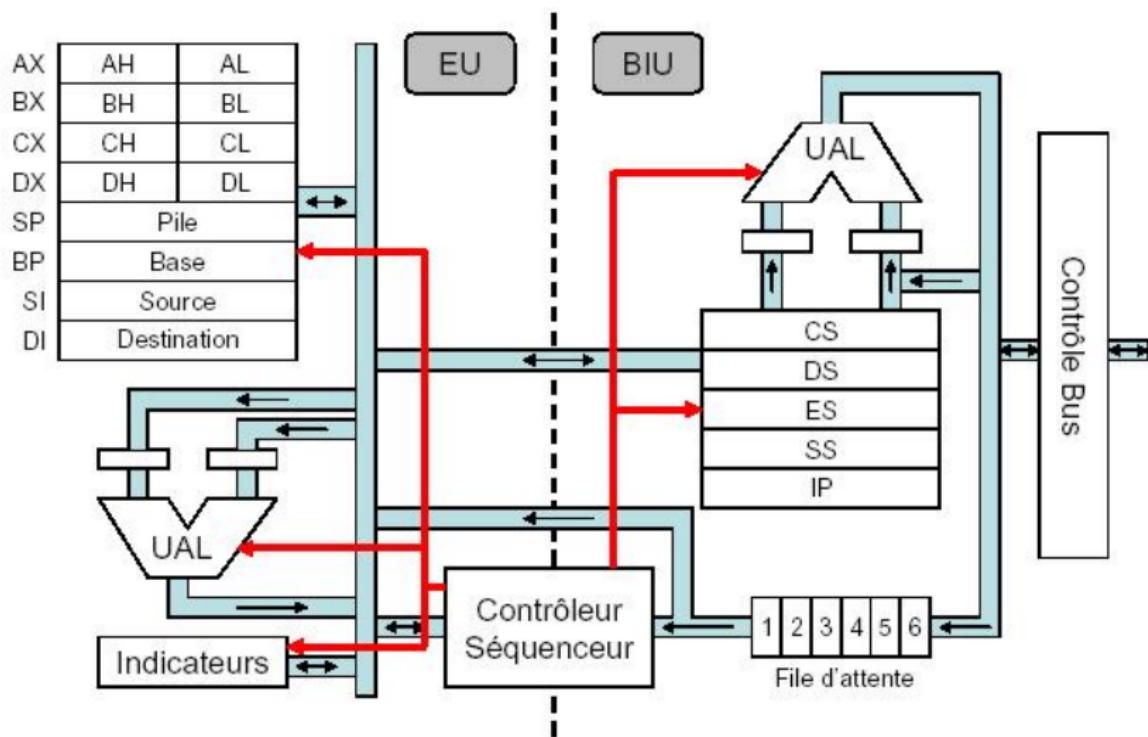
Famille de microprocesseurs compatibles avec le jeu d'instructions de l'Intel 8086

Evolution

- ▶ 1978: **8086**, ensuite **80286, 80386, 80486, Pentium, Celeron, Xeon, Core, etc.** chez Intel; et chez AMD: **K5, K6, Athlon, Duron, Sempron, Opteron, Turion et Haipad**
- ▶ augmentation de la fréquence d'horloge, et de la largeur des bus d'adresses et de données
- ▶ ajout de nouvelles instructions et registres (**exemple Intel: des registres 32 bits à partir de 386 et 64 bits à partir de Xeon**)
- ▶ **Compatibilité ascendante:** Un programme écrit dans le langage machine du 286 peut s'exécuter sur un 386

Langage Assembleur

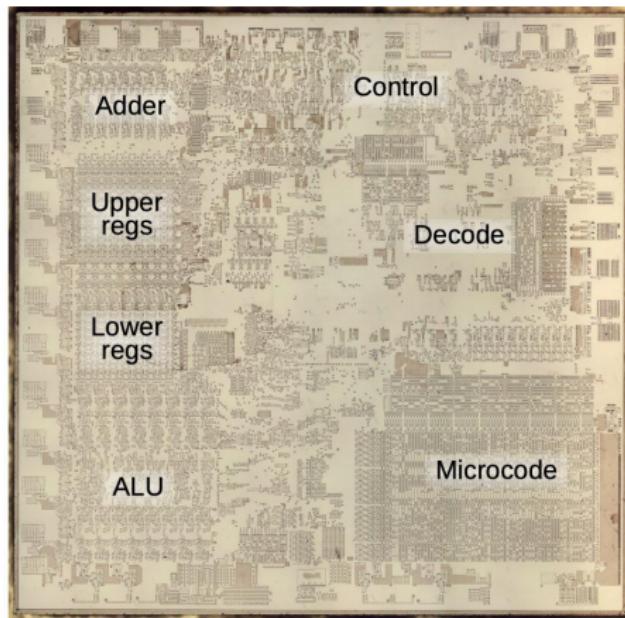
Processeurs 80x86: architecture interne (mode 16 bits)



Langage Assembleur

Processeurs 80x86: architecture interne (mode 16 bits)

Concrètement...



Architecture interne d'un Intel 8086
- <http://www.righto.com>

Langage Assembleur

Processeurs 80x86: architecture interne (mode 16 bits)

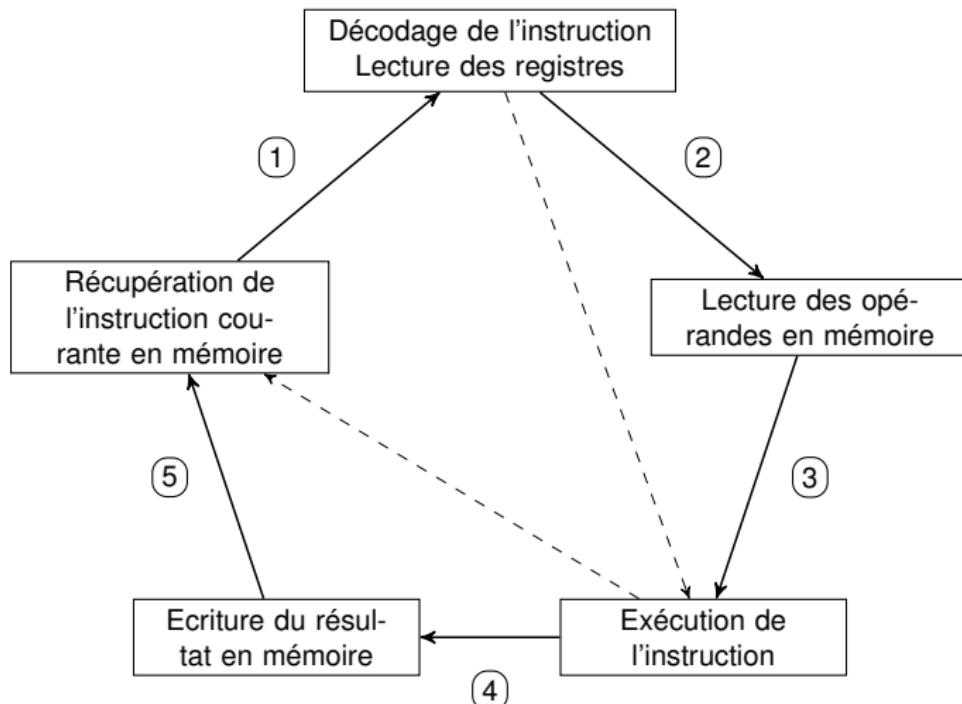
Composants du CPU

- ▶ **Bus:** nappes de fils transportant l'info
(données sur 16 bits, adresses sur 20 bits -*mode réel*-, infos de contrôle)
- ▶ **Registres:** emplacements de stockage à accès rapide (16 bits; pour les registres de données divisables en 2 x 8 bits)
- ▶ **UAL:** unité de calcul (entiers et booléens)
- ▶ **Unité de contrôle (UC):** interprète les instructions
- ▶ **Horloge:** cadence les instructions

NB: en TP, on utilisera un assembleur x86 sous linux, et on exécutera le programme sur des Intel core i5

Langage Assembleur

Processeurs 80x86: Unité de contrôle



Cycle de fonctionnement de l'UC.

Langage Assembleur

Processeurs 80x86: Registres (mode 64 bits)

Registres présents sur une architecture 80x86, 64 bits

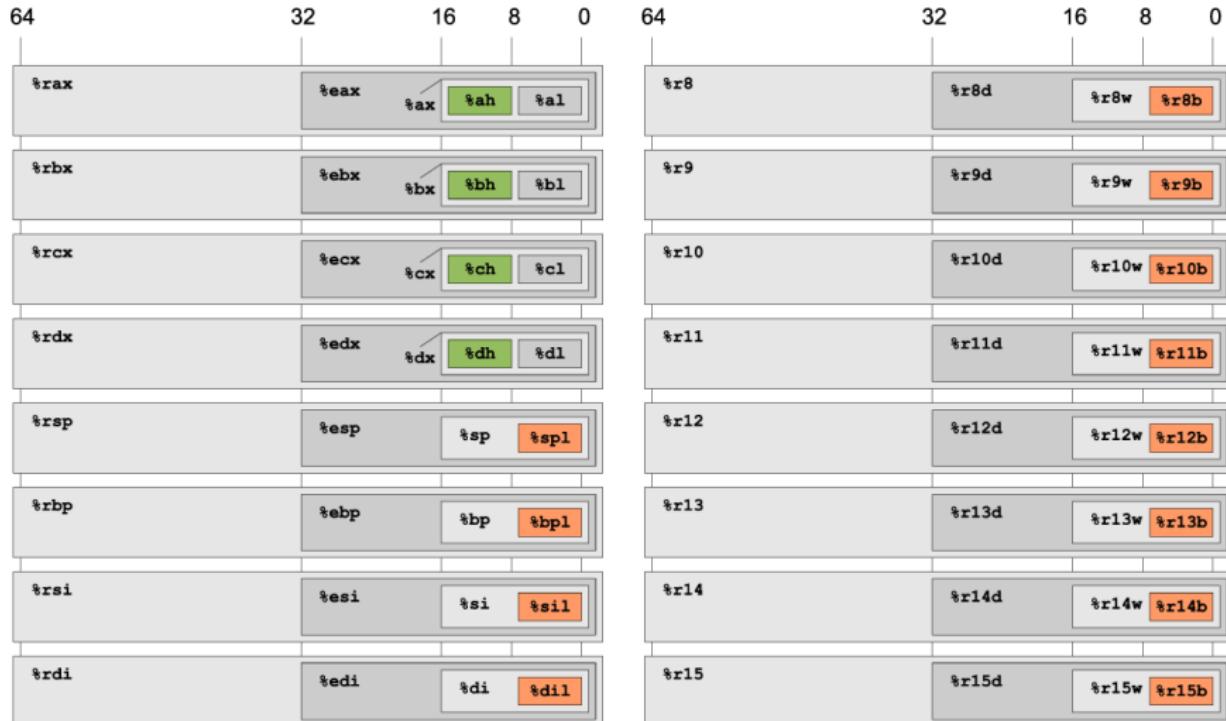
- ① **RAX, RBX, RCX, RDX, R8 à R15** (registres généraux de 64 bits)
- ② **xS (CS, DS, SS, ES)**: Registres de segments (en mode 32 ou 16 bits)
- ③ **EFLAGS** (registre d'état et de contrôle)
- ④ **RIP** (Instruction Pointer ou EIP en 32 bits ou IP en 16 bits)
(pointe sur la prochaine instruction à exécuter)
- ⑤ **RSP et RBP** (registres de pile)
- ⑥ **RSI et RDI** (utilisé pour les transferts en mémoire)

Registres de segment

En mode 32 bits ou 16 bits, nécessité d'utiliser les registres de segment **xS**

Langage Assembleur

Processeurs 80x86: Registres (mode 64 bits)



Registres d'un processeur 80x86 - 64 bits -

Langage Assembleur

Processeurs 80x86: Registres

Utilité des registres

- ▶ stocker des opérandes lors d'opérations logiques ou arithmétiques
- ▶ stocker des opérandes pour des calculs d'adresses
- ▶ stocker des pointeurs (adresses mémoire)
- ▶ certaines instructions imposent les registres à utiliser

Utilisations particulières

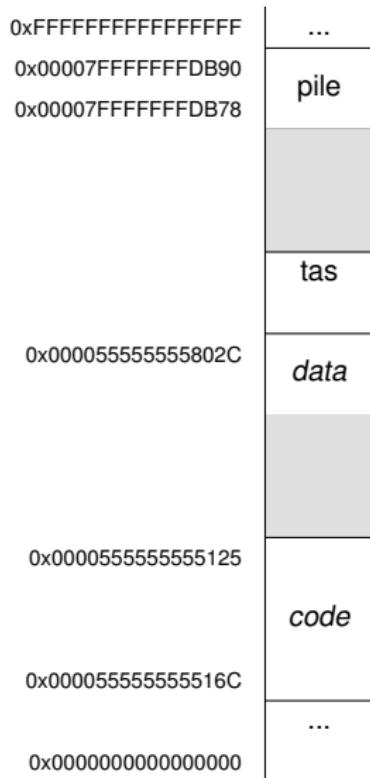
- ▶ **RAX** : accumulateur ou valeur de retour de fonction
- ▶ **RBX** : pointeur vers les données
- ▶ **RCX** : compteur de boucles
- ▶ **RDX** : accumulateur aux. et pointeur pour les entrées/sorties
- ▶ **RSI** : pointeur source pour la manipulation de caractères
- ▶ **RDI** : pointeur destination pour la manipulation de caractères
- ▶ **RSP** : pointeur de début de pile (sommet)
- ▶ **RBP** : pointeur de bas de pile

La pile

- ▶ La pile est une structure (et une zone mémoire) qui sert à stocker temporairement des informations
- ▶ Utilisée lors d'appel de procédures (passage de paramètres)
- ▶ Fonctionne en mode LIFO (*Last In First Out*)

Langage Assembleur

La pile



- ▶ la pile “grandit” vers le bas
- ▶ la tas “grandit” vers le haut
- ▶ **RBP** = adresse du “fond” de la pile
- ▶ **RSP** = adresse du sommet de la pile (dernier élément empilé)

Langage Assembleur

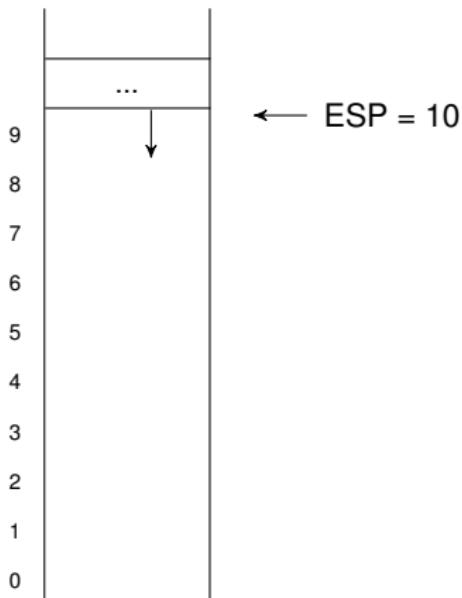
La pile

- ▶ **push arg** : empiler *arg* (reg., case mémoire, valeur)
- ▶ **pop arg** : dépiler dans *arg* (reg., case mémoire)

- ▶ empilement de mots (16bits) ou double-mots (32bits)
- ▶ c'est l'argument qui détermine la taille
- ▶ parfois nécessaire d'indiquer la taille

Langage Assembleur

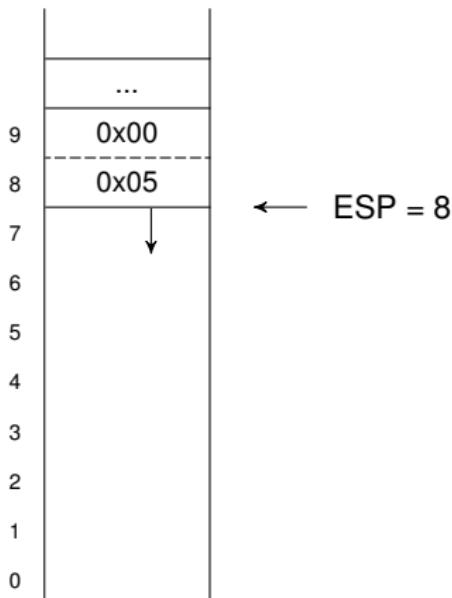
La pile



```
mov ax, 5  
mov ebx, 0x34  
push ax  
push ebx  
pop ecx  
pop [edx]
```

Langage Assembleur

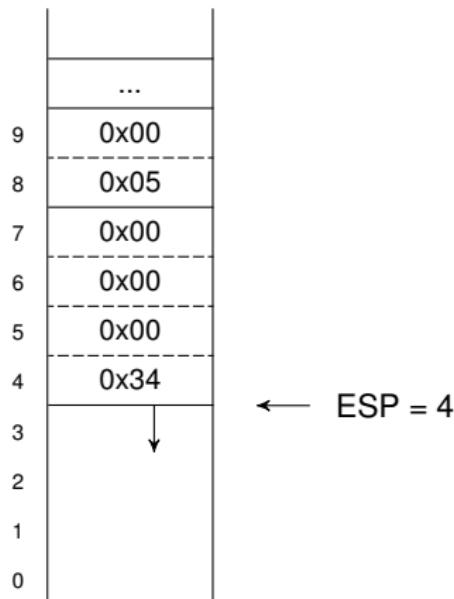
La pile



```
mov ax, 5  
mov ebx, 0x34  
push ax  
push ebx  
pop ecx  
pop [edx]
```

Langage Assembleur

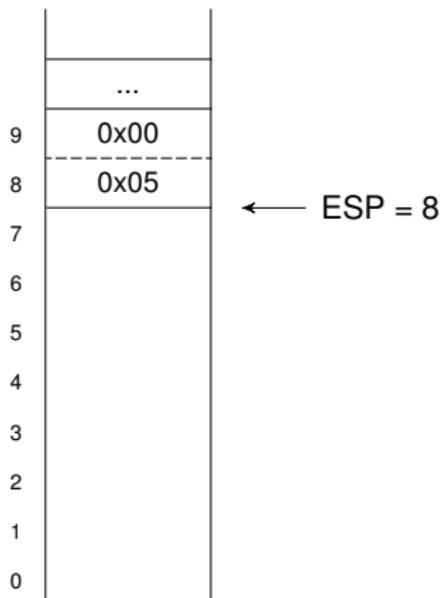
La pile



```
mov ax, 5  
mov ebx, 0x34  
push ax  
push ebx  
pop ecx  
pop [edx]
```

Langage Assembleur

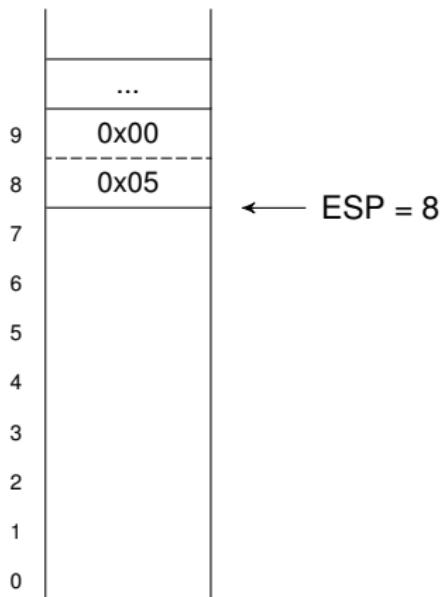
La pile



```
mov ax, 5  
mov ebx, 0x34  
push ax  
push ebx  
pop ecx ; ecx ← 0x34  
pop [edx]
```

Langage Assembleur

La pile

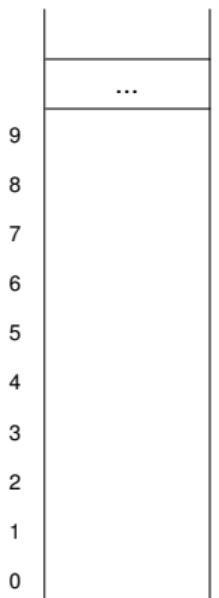


```
mov ax, 5  
mov ebx, 0x34  
push ax  
push ebx  
pop ecx ; ecx ← 0x34  
pop [edx]
```

→ combien de cases dépilerées?

Langage Assembleur

La pile



```
mov ax, 5
mov ebx, 0x34
push ax
push ebx
pop ecx    ; ecx ← 0x34
pop word [edx]; [edx] ← 0x05
```

Langage Assembleur

Les procédures

Une procédure

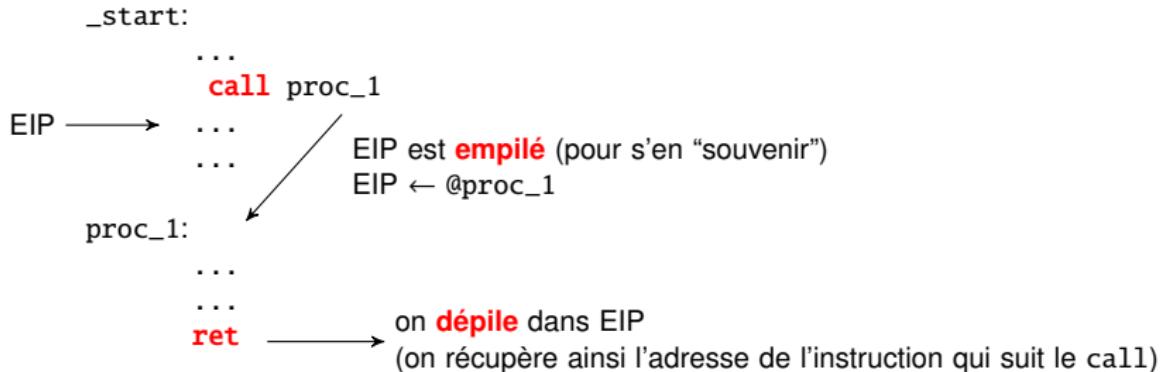
- ▶ est une suite d'instructions
- ▶ possède un **début** et une **fin**
- ▶ avec **début** signalé par un *label*
- ▶ et **fin** signalée par l'instruction **ret**
- ▶ peut être appelé avec l'instruction **call <label>**

Attention

- ▶ il faut que la pile soit dans le même état à la fin de la procédure qu'avant son appel
- ▶ EIP doit être en sommet de pile (instruction **ret**)

Langage Assembleur

Les procédures



- ▶ L'instruction `ret` s'attend à trouver `EIP` en sommet de pile !
- ▶ Une procédure doit rendre la pile et les registres dans l'état où elle les a trouvé

Langage Assembleur

Les procédures: Passage de paramètres

- Comment passer des paramètres à une procédure ?

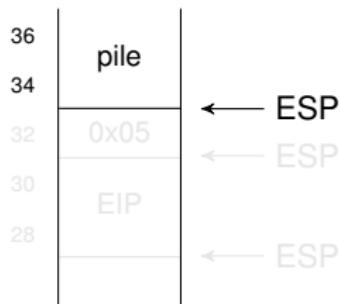
Langage Assembleur

Les procédures: Passage de paramètres

- ▶ Les paramètres d'une procédure peuvent être passés par **la pile**
- ▶ Les paramètres sont empilés avant l'instruction **call**
- ▶ Comme en C, si le paramètre doit être modifié par la procédure, l'adresse de la donnée doit être passée, pas sa valeur

Langage Assembleur

Les procédures: Passage de paramètres



```
_start:  
    ...  
    push word 5  
    call maproc  
    ...  
maproc :  
    ...  
ret
```

Dans la procédure maproc,
comment récupérer le paramètre 5 ?

Langage Assembleur

Les procédures: Passage de paramètres



_start:

...
push word 5
call maproc

...

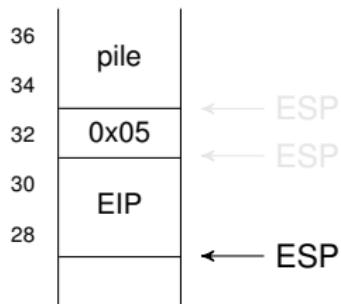
maproc :
...

ret

Dans la procédure maproc,
comment récupérer le paramètre 5 ?

Langage Assembleur

Les procédures: Passage de paramètres

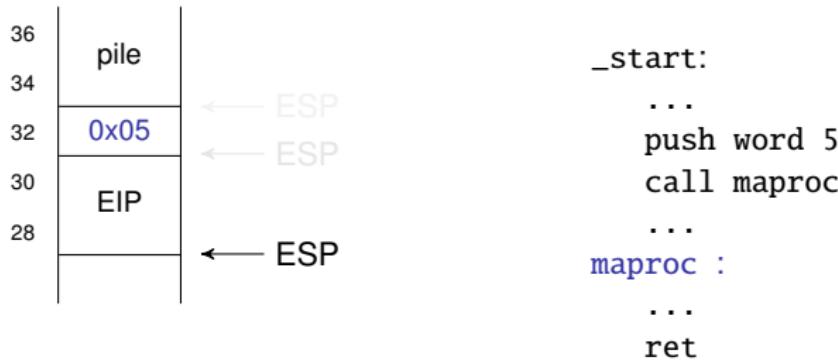


```
_start:  
    ...  
    push word 5  
    call maproc  
    ...  
maproc :  
    ...  
    ret
```

Dans la procédure maproc,
comment récupérer le paramètre 5 ?

Langage Assembleur

Les procédures: Passage de paramètres



Dans la procédure `maproc`,
comment récupérer le paramètre 5 ?

Langage Assembleur

Les procédures: Passage de paramètres

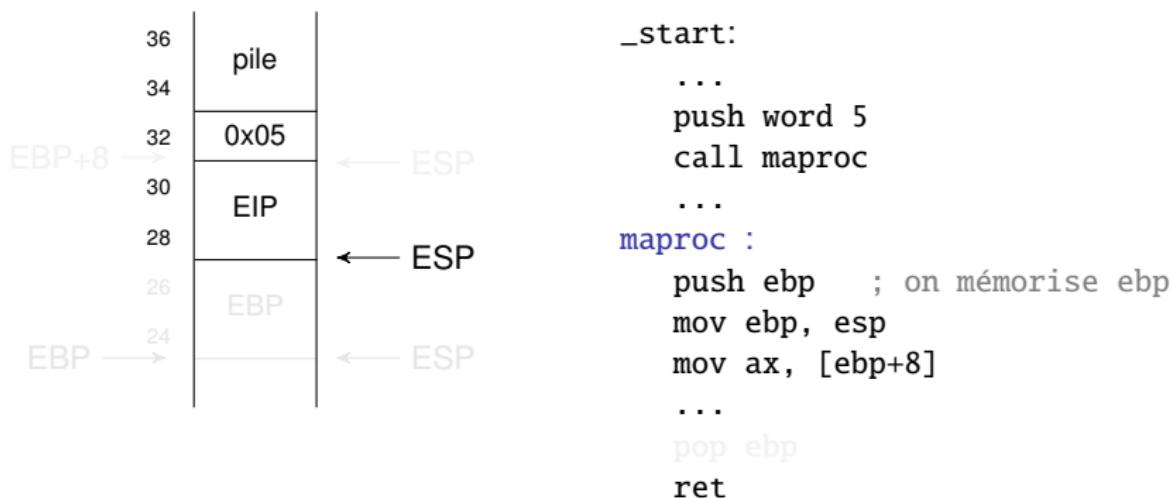
- ▶ On ne peut lire les paramètres qu'au moyen du registre **ebp** (pointeur de base de la pile)

Comment?

- On initialise **ebp** avec la valeur de **esp** (pointeur début de la pile)
(en ayant pris soin de sauvegarder *avant* la valeur de **ebp**)

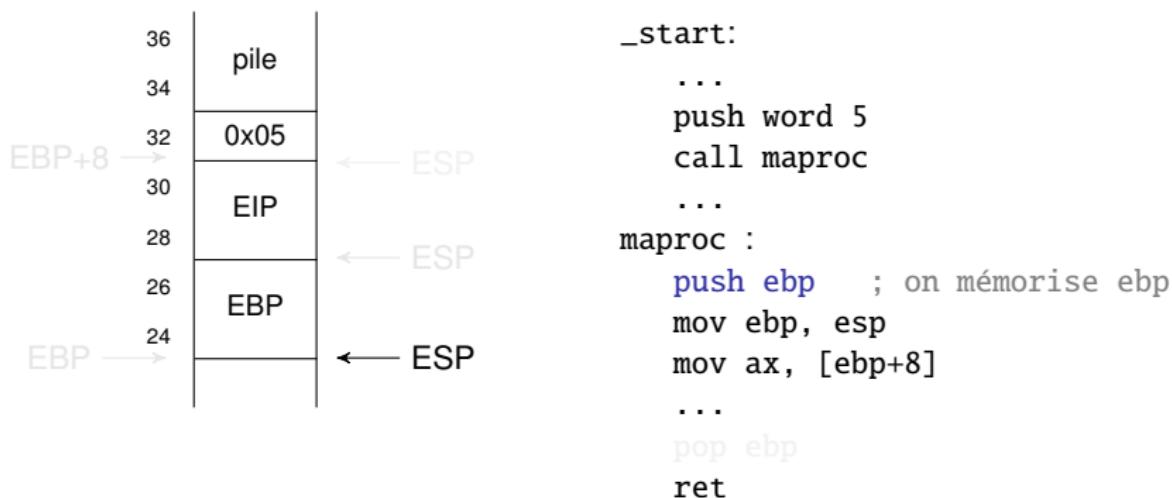
Langage Assembleur

Les procédures: Passage de paramètres



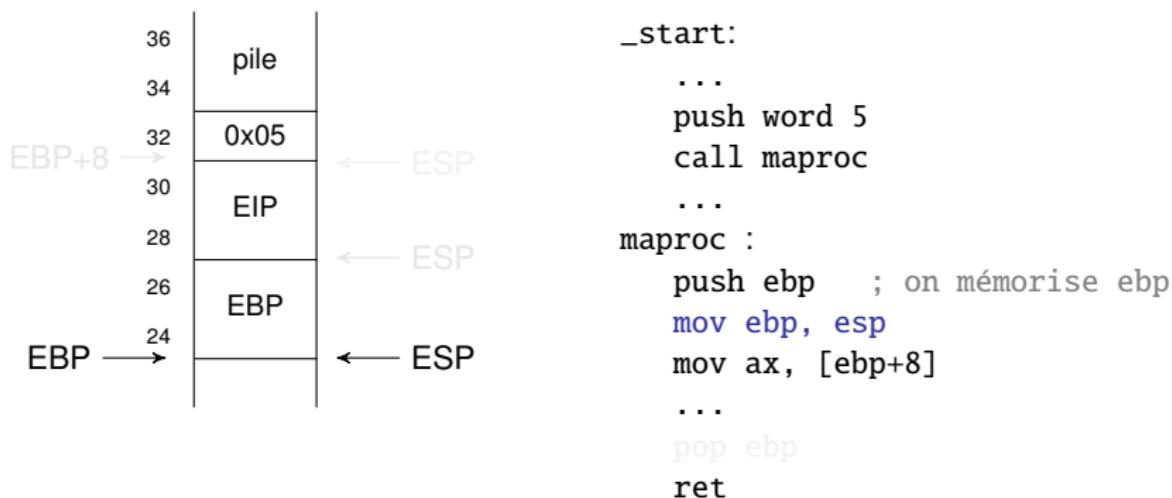
Langage Assembleur

Les procédures: Passage de paramètres



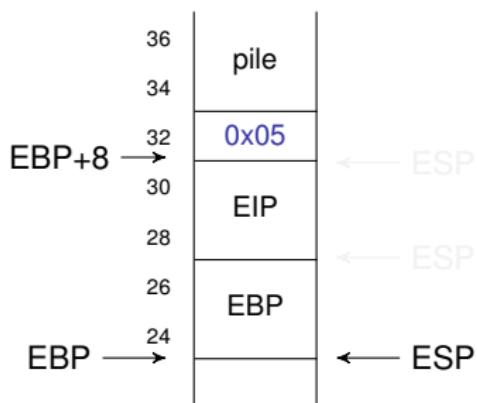
Langage Assembleur

Les procédures: Passage de paramètres



Langage Assembleur

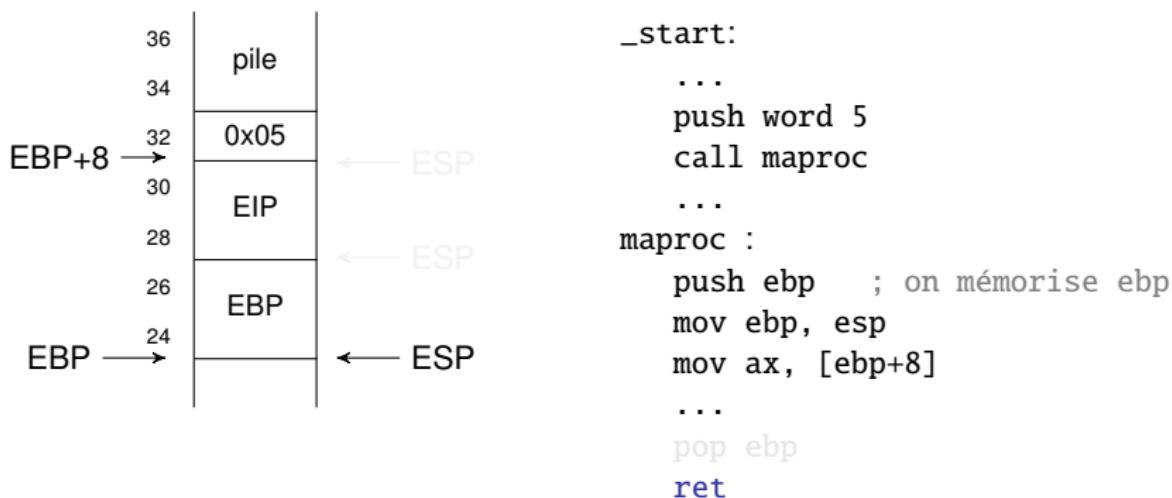
Les procédures: Passage de paramètres



```
_start:  
...  
push word 5  
call maproc  
...  
maproc :  
    push ebp ; on mémorise ebp  
    mov ebp, esp  
    mov ax, [ebp+8]  
...  
    pop ebp  
ret
```

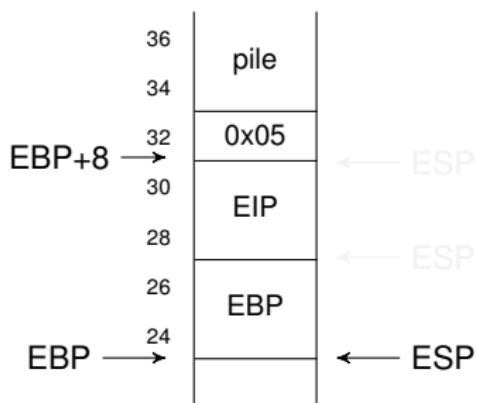
Langage Assembleur

Les procédures: Passage de paramètres



Langage Assembleur

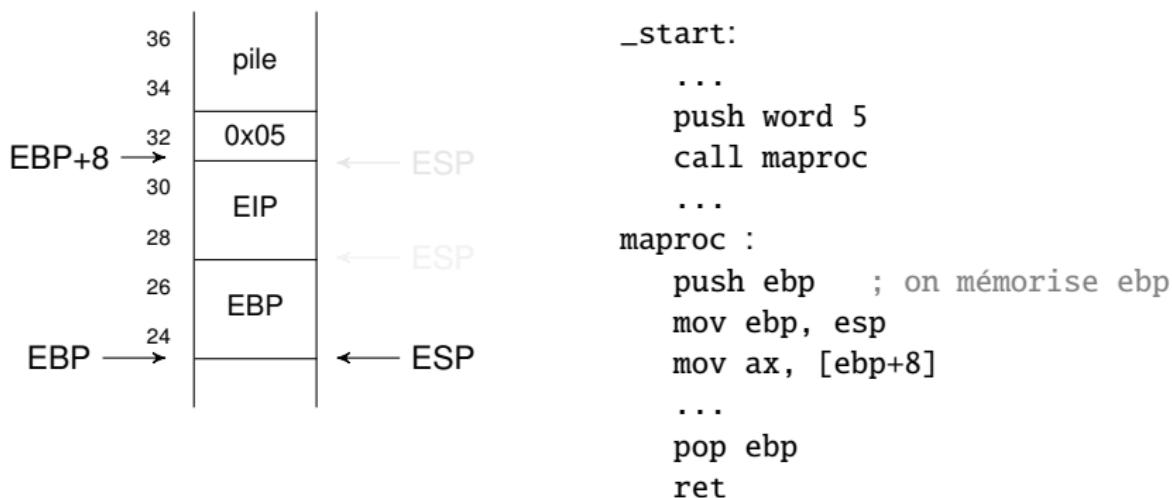
Les procédures: Passage de paramètres



```
_start:  
    ...  
    push word 5  
    call maproc  
    ...  
maproc :  
    push ebp      ; on mémorise ebp  
    mov ebp, esp  
    mov ax, [ebp+8]  
    ...  
    pop ebp  
    ret
```

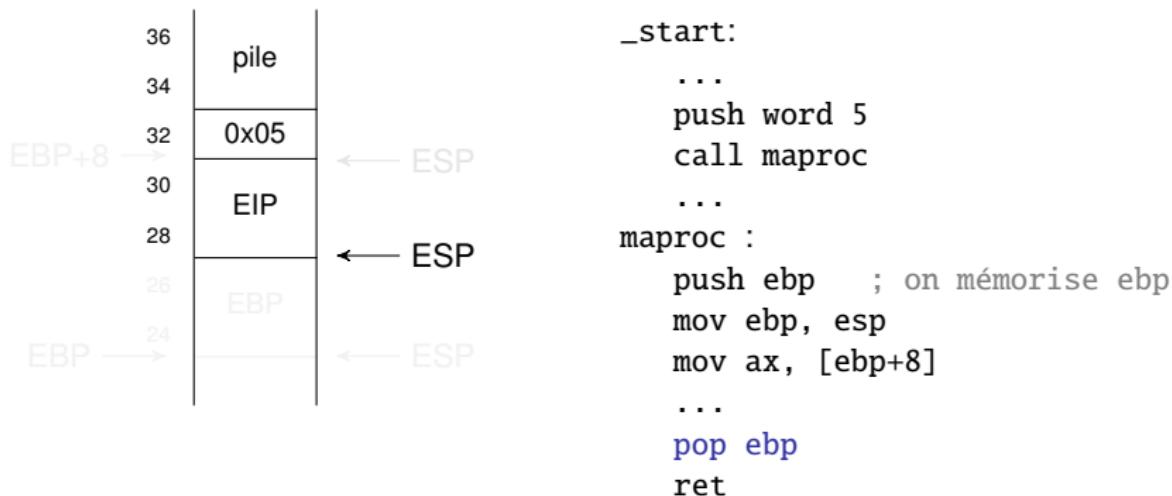
Langage Assembleur

Les procédures: Passage de paramètres



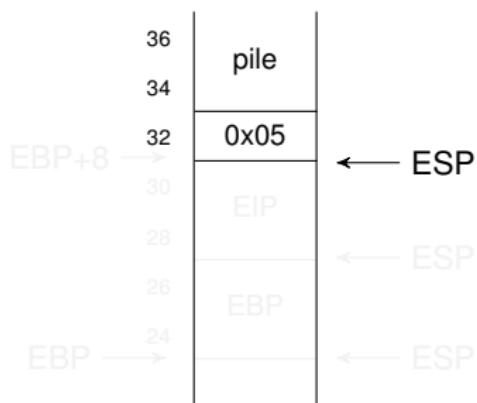
Langage Assembleur

Les procédures: Passage de paramètres



Langage Assembleur

Les procédures: Passage de paramètres



`_start:`

...

`push word 5`

`call maproc`

...

`maproc :`

`push ebp ; on mémorise ebp`

`mov ebp, esp`

`mov ax, [ebp+8]`

...

`pop ebp`

`ret`

Langage Assembleur

gdb: debugger sous Linux

gdb = outil indispensable de debuggage !

Quelques commandes utiles

- ▶ Lancement : **gdb <nom prg>**
- ▶ **break <label>** : définition d'un point d'arrêt (parmi les étiquettes du code asm du prog.)
- ▶ **info break** : liste des points d'arrêt
- ▶ **run** : exécution du programme jusqu'au premier point d'arrêt
- ▶ **c (continue)** : exécution jusqu'au prochain point d'arrêt

Langage Assembleur

gdb: debugger sous Linux

- ▶ **info registers** : valeur des registres du CPU
- ▶ **info variables** : liste des variables avec leur adresse
- ▶ **x <@ var>** : valeur contenue à l'adresse de la variable
- ▶ **set disassembly-flavor intel** : indique que l'on veut voir le code assembleur selon le type Intel (x86)
- ▶ **disassemble <label>** : affiche le code assembleur à partir de l'étiquette jusqu'à la suivante

Plan

1 Introduction

2 Processeur et assembleur x86

3 Unix/Linux

4 Processus

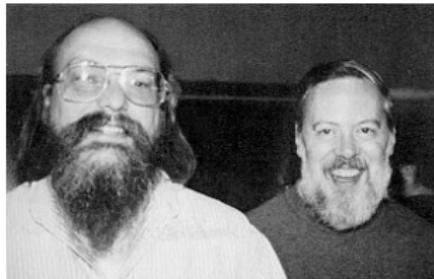
5 Fichiers

Historique d'Unix et Linux

- ▶ Historique: d'Unics de Ken Tompson & Dennis Ritchie en 1969 à GNU/Linux avec son noyau 5.19.9 (septembre 2022)
 - ▶ Les différents Unix et “The Open Group”
-
- ▶ Concept de logiciel libre
 - ▶ GNU et FSF
 - ▶ Open Source
-
- ▶ GNU/Linux
 - ▶ Les distributions de GNU/Linux

Historique d'Unix et Linux

Quelques personnages



Ken Tompson
Dennis Ritchie
Les pères d'Unix

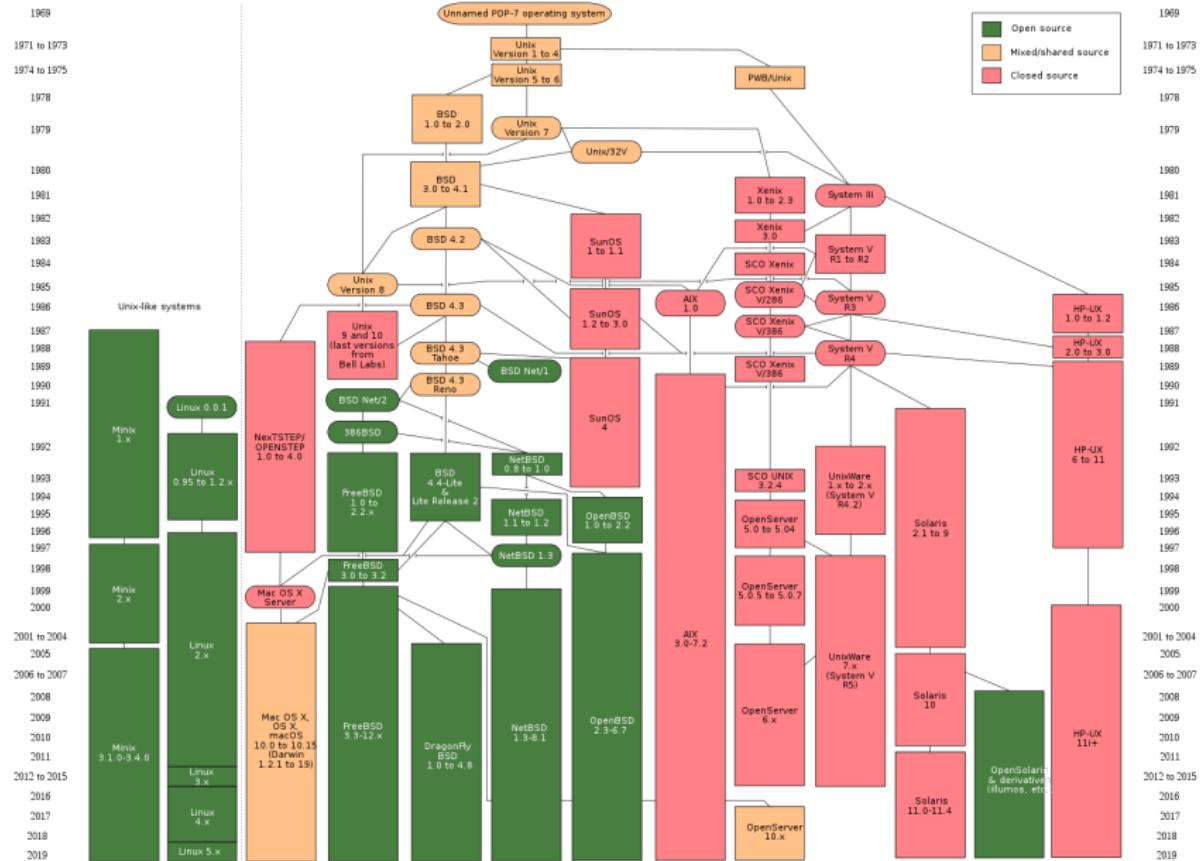


Richard Stallman
Le père de GNU



Linus Torvalds
Le père de Linux

Historique d'Unix et Linux



Unix propriétaires

- ▶ AIX d'IBM (version 1.0 en 1986 → version 7.1 en 2010) sur architectures PowerPC et IA-64
- ▶ HP-UX d'HP (version 1.0 en 1984 → version 11.31 en 2007) sur architecture PA-RISC de Motorola (68000) et Itanium d'Intel
- ▶ UnixWare de SCO (version 1.0 en 1991 → version 7.1.4 en 2008) sur architectures x86 (propriétaire de 1993 à 1995 : Novell)
- ▶ IRIX de SGI (version 3.0 en 1986 → version 6.5 en 1998 et 6.5.22 en 2007 - fin du produit)
- ▶ Solaris de SUN (version 1.0 en 1991 → version 11 en 2010) ; aujourd'hui, c'est OpenSolaris

Historique d'Unix et Linux

Unix libres

- ▶ la famille BSD (*Berkeley Software Distribution*) : à partir de la 4.3
 - ▶ NetBSD (version 0.8 en 1993 → version 5.1 depuis 2010) sur 15 architectures différentes ; utilisé pour les systèmes embarqués
 - ▶ OpenBSD : *fork* de NetBSD en 1995 ; centré sur la notion de liberté, de qualité de la doc et sur la sécurité
 - ▶ FreeBSD (version 1 en 1993 → version 8.2 depuis 2011) ; une distribution pour tout et en toute liberté ; la plus utilisée !
- ▶ Linux

Unix mixte

- ▶ Mac OS X basé sur FreeBSD et NetBSD, sur le micro-noyau Mach et une API de haut niveau non libre

Historique d'Unix et Linux

Normalisation

- ▶ The Open Group : consortium de normalisation neutre vis à vis d'une technologie ou une entreprise
 - ▶ possède la marque déposée UNIX®
 - ▶ publie la norme "Single UNIX Specification" (SUS - IEEE 1003.1-2008), remplaçant de POSIX (IEEE 1003)
 - ▶ depuis 2018, SUS et **POSIX** fusionne en "Open Group Base Specification"
- ▶ en 1999, le format SVR4's **Executable and Linkable Format (ELF)** supporté par plusieurs constructeurs est défini pour définir le format des fichiers exécutables
- ▶ en 2015, **Filesystem Hierarchy Standard (FHS)** définit la structure arborescente des répertoires d'une distributions Linux

XV6

- ▶ implémentation de la version 6 de l'Unix de Dennis Ritchie's et Ken Thompson's Unix Version 6 (v6)
- ▶ proche de structure et du style de la version 6
- ▶ développé en ANSI C pour les architectures à base de multi-processeurs x86

Avantages

- ▶ XV6 est un OS léger
- ▶ conçu par le MIT pour un contexte d'éducation
- ▶ possède une structure “facilement” compréhensible
- ▶ XV6 peut s'exécuter dans un environnement émulé comme QEMU

Plan

1 Introduction

2 Processeur et assembleur x86

3 Unix/Linux

4 Processus

5 Fichiers

Processus

Définition

Définition

- ▶ **programme:** suite statique d'instructions
- ▶ **processeur:** entité qui exécute les instructions
- ▶ **processus:** activité, séquence d'instructions en exécution par un processeur afin de réaliser une tâche déterminée ⇒
programme en cours d'exécution avec les valeurs du compteur ordinal, des registres, des variables et les données

Programme et processus

- ▶ Un programme qui s'exécute deux fois donne lieu à deux processus

Processus

Définition

Définition

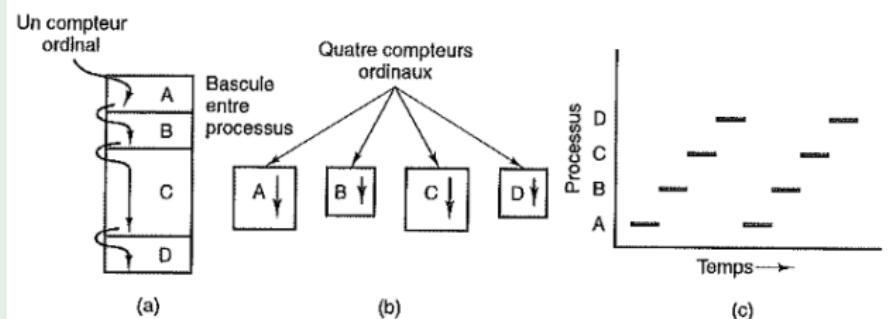
- ▶ **programme:** suite statique d'instructions
 - ▶ **processeur:** entité qui exécute les instructions
 - ▶ **processus:** activité, séquence d'instructions en exécution par un processeur afin de réaliser une tâche déterminée ⇒
programme en cours d'exécution avec les valeurs du compteur ordinal, des registres, des variables et les données
-
- ▶ les éléments affectés par l'exécution d'un programme sont:
 - ▶ les registres du CPU
 - ▶ le contenu de la mémoire (code, données, pile, tas, ...)
 - ▶ l'état des entrées-sorties : table des descripteurs de fichiers, ...

Processus

Définition - modèle de processus

Pseudo-parallélisme

Le processeur bascule constamment d'un processus à un autre. Ce basculement rapide est appelé **multiprogrammation**.



©Tanenbaum

(a) multiprogrammation de quatre programmes, (b) modèle conceptuel de quatre processus séquentiels indépendants, (c) un seul programme est actif à un instant donnée

Processus

Pourquoi utiliser des processus?

Trois grands principes:

Concurrence

- ▶ le système possède plusieurs processus “job” concurrents qui veulent accéder aux ressources (processeur, mémoire, entrées-sorties, ...)
- ▶ le système d’exploitation partage les ressources entre eux

Diviser pour régner (*Divide-and-conquer*)

- ▶ principe général: décomposer un problème initial de grande taille en un ensemble de problèmes de plus petite taille, les résoudre, combiner les solutions
- ▶ Unix: “que des petits programmes simples”

Isoler

Isoler chaque “job” les uns des autres et définir les interactions

Processus

Création d'un processus

Evénements conduisant à la création d'un processus:

- ① initialisation du système
- ② exécution d'un appel système de création de processus par un processus en cours
- ③ requête utilisateur sollicitant la création d'un nouveau processus
- ④ lancement d'un travail en traitement par lots

Processus

Création d'un processus

Concrètement...

Unix

- ▶ un seul appel système: **fork()**. Il crée un clone du processus appelant ⇒ les processus père et fils ont la même image mémoire et les mêmes fichiers ouverts
- ▶ le fils généralement exécute un appel de type **execve()** pour exécuter un autre programme et modifier son image mémoire
- ▶ le premier processus **init** (ou `/sbin/init`) est dans l'image d'amorçage (lancé par le noyau)

Windows

- ▶ un seul appel Win32: **CreateProcess()**. Il prend en charge la création du processus et le chargement du programme

Processus

Fin d'un processus

Événements conduisant à la fin d'un processus:

- ① arrêt normal (volontaire)
- ② arrêt pour erreur (volontaire)
- ③ arrêt pour erreur fatale (involontaire)
- ④ le processus est arrêté par un autre processus (involontaire)

Processus

Fin d'un processus

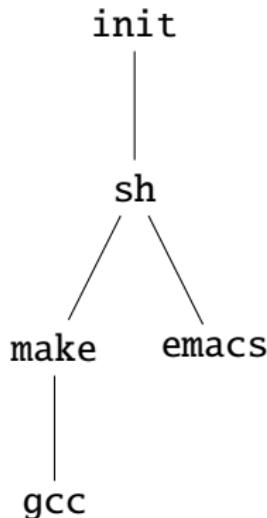
Evénements conduisant à la fin d'un processus:

- ① arrêt normal (volontaire): appel système **exit()** sous Unix, et **ExitProcess()** sous Windows
- ② arrêt pour erreur (volontaire)
- ③ arrêt pour erreur fatale (involontaire). Exemples: référence mémoire inexistance, division par zéro
- ④ le processus est arrêté par un autre processus (involontaire): appel système **kill()** sous Unix, et **TerminateProcess()** sous Windows

Processus

Hiéarchie des processus

- ▶ pas de hiéarchie sous Windows. Tous les processus sont égaux.
- ▶ une hiéarchie arborescente suivant les liens de parenté, avec en racine le processus init. Exemple:

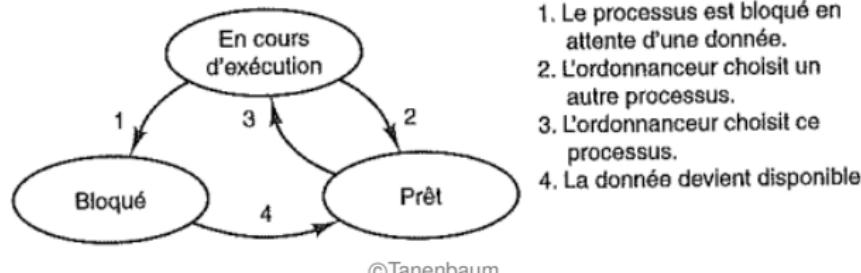


Processus

États des processus

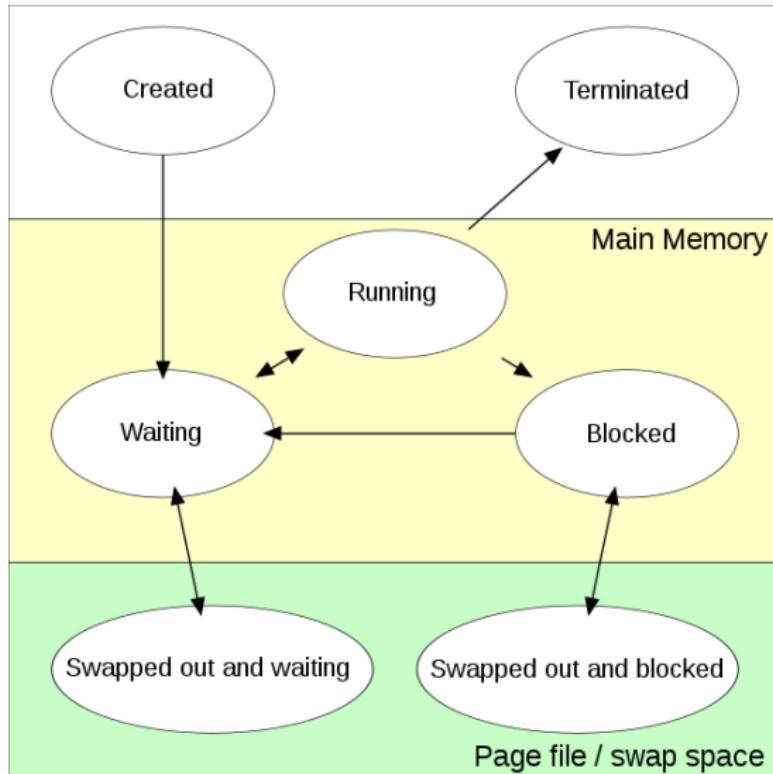
Hormis les états **nouveau (new)** et **terminé (terminated)**, les processus peuvent prendre 3 états:

- ▶ **en cours d'exécution (running)**: utilise le CPU à cet instant (le compteur ordinal est avancé)
- ▶ **prêt (ready)**: exécutable, temporairement arrêté pour laisser un autre processus
- ▶ **bloqué (waiting)**: ne peut pas s'exécuter tant qu'un événement externe (une entrée clavier, le résultat d'une requête disque, un message, ...) ne se produit pas



Processus

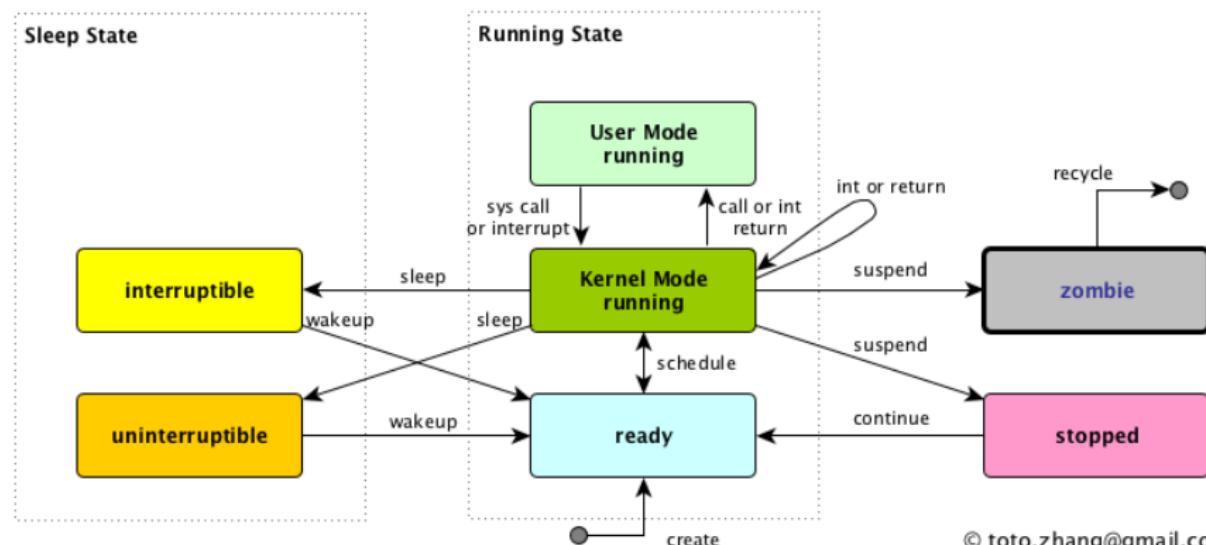
États des processus



Processus

États des processus (Unix-like)

État **bloqué (waiting)** peu être interruptible ou pas: peut être réveillé par un signal (interruptible) ou attend le retour du matériel (non interruptible)



© toto.zhang@gmail.com

Processus

Implémentation des processus

Le SE maintient une table de processus. Chaque entrée représente un processus, et elle est appelée **bloc de contrôle de processus**.

Bloc de contrôle de processus (Process control block - PCB)

Le contexte d'un processus est défini par le **Process control block**:

- ▶ l'identifiant du processus (PID), l'identifiant du processus parent (PPID) et l'identifiant de l'utilisateur du processus (UID)
- ▶ l'état (new, ready, running, waiting, terminated, ...)
- ▶ les valeurs des registres correspondant au processus
- ▶ le compteur ordinal du processus (adresse de la prochaine instruction) et le pointeur de pile
- ▶ l'espace d'adressage du processus
- ▶ la liste des descripteurs de fichiers
- ▶ les informations d'ordonnancement

Processus

Implémentation des processus

Le SE maintient une table de processus. Chaque entrée représente un processus, et elle est appelée **bloc de contrôle de processus**.

Bloc de contrôle de processus (Process control block - PCB)

Gestion du processus	Gestion de la mémoire	Gestion de fichier
Registres	Pointeur vers un segment de texte	Répertoire racine
Compteur ordinal	Pointeur vers un segment de données	Répertoire de travail
Mot d'état du programme	Pointeur vers un segment de la pile	Descripteurs de fichiers
Pointeur de la pile		ID utilisateur
État du processus		ID du groupe
Priorité		
Paramètres d'ordonnancement		
ID du processus		
Processus parent		
Groupe du processus		
Signaux		
Heure de début du processus		
Temps de traitement utilisé		
Temps de traitement du fils		
Heure de la prochaine alerte		

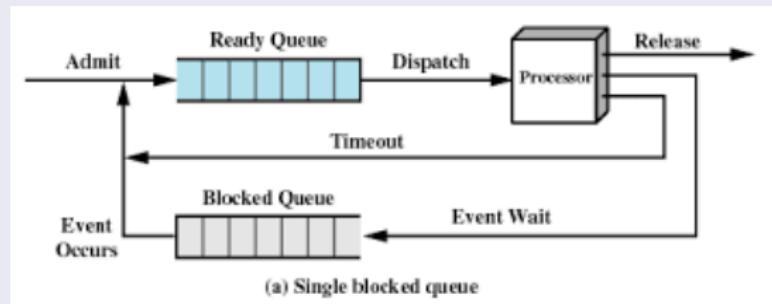
Processus

Implémentation des processus

Le SE maintient également des files d'attente de processus appelée **job queues**. Chacune des entrées des files est un PCB liée à la table des processus.

Job queues

Les PCB sont stockés dans des files d'attente en fonction de leur état (Ready/Waiting)

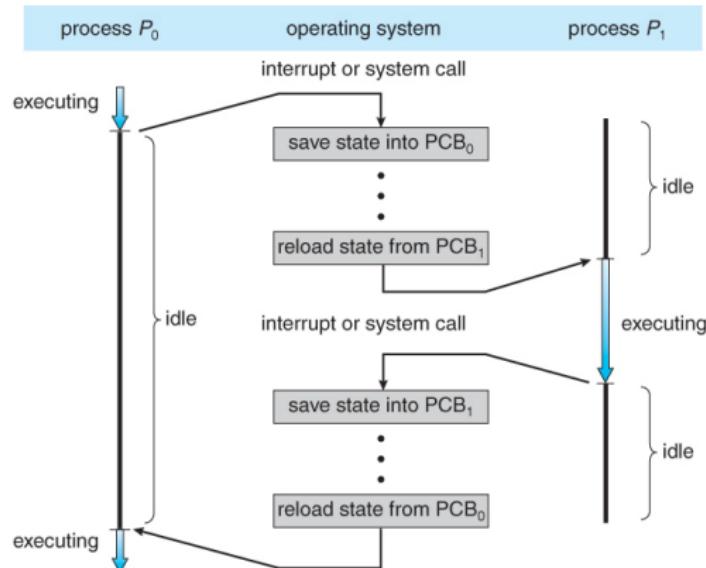


Processus

Implémentation des processus

Changement de contexte

Lorsqu'un processus p est arrêté \Rightarrow copie des informations dans le PCB de p et chargement du PCB du nouveau processus dans les registres



Processus

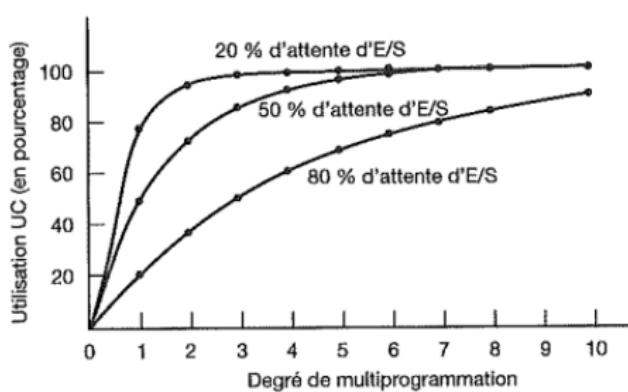
Modélisation de la multiprogrammation

Question. Combien de processus un CPU simultanément?

Modèle probabiliste

Si $0 \leq p \leq 1$ est la fraction de temps un processus passe à attendre la fin d'une E/S \Rightarrow **Taux d'utilisation du CPU pour n processus** = $1 - p^n$

Exemples pour $p = 20\%$, $p = 50\%$, et $p = 80\%$:



Processus

Ordonnancement

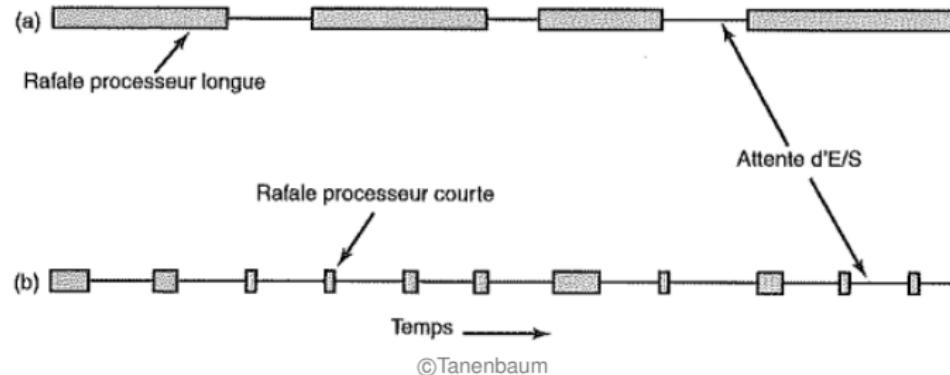
Définition

- ▶ lorsqu'un processus est dans l'état Ready, il peut être exécuté par le CPU (passage en état Running)
- ▶ si plusieurs processus sont dans cet état, le SE doit faire un choix
- ▶ ce mécanisme se nomme l'ordonnancement
- ▶ l'algorithme employé se nomme algorithme d'ordonnancement
- ▶ il est réalisé par l'ordonnanceur (scheduler) qui est une partie du SE

Processus - Ordonnancement

Comportement des processus

(a) processus de traitement et (b) procesus d'E/S:



Processus - Ordonnancement

Quand ordonner?

- ① lorsqu'un nouveau processus est créé
- ② lorsqu'un processus se termine
- ③ lorsqu'un processus bloque sur une E/S
- ④ lors d'une interruption d'E/S se produit
- ⑤ lors des interruptions périodiques d'horloge. Par exemple à chaque k^e interruption (ordonnancement préemptif)

Processus - Ordonnancement

Quand ordonner?

Préemptif / non préemptif

- ▶ un algorithme d'ordonnancement est dit **non préemptif (ou coopératif)** si les tâches sont exécutées dans leur totalité (ou jusqu'à blocage) avant l'exécution d'une autre tâche

- ▶ un algorithme d'ordonnancement est dit **préemptif** lorsque l'ordonnanceur peut interrompre chaque tâche après un délai déterminé (durant une interruption d'horloge)

Processus - Ordonnancement

Quand ordonner?

Impacts de la préemption sur l'ordonnanceur

- ▶ chaque type de système (préemptif ou non) va conduire à différents algorithmes d'ordonnancement
- ▶ chaque type de système (préemptif ou non) est adapté à un type d'environnement
 - ▶ **non préemptif:** traitement par lot
 - ▶ **préemptif:** système interactifs, serveurs ...

Processus - Ordonnancement

Objectifs

Objectifs du scheduler

- ▶ **équité:** attribuer à chaque processus un temps CPU équitable
- ▶ **équilibre:** faire en sorte que toutes les parties du système soient utilisées
- ▶ **capacité de traitement:** maximiser la nombre de “jobs” effectué à l’heure
- ▶ **minimiser les temps d’attente:** délai d’attente entre la soumission du “job” et son achèvement
- ▶ **minimiser les temps de réponse:** temps de production de la première sortie à l’utilisateur en cas d’activité interactive

Processus - Ordonnancement

Critères de comparaison

Critères de comparaison entre algorithmes d'ordonnancement

- ▶ **Délai de rotation (turnaround time):** durée nécessaire pour l'exécution d'un processus (temps d'attente compris)
- ▶ **Temps d'attente:** durée totale qu'un processus est dans l'état Waiting
- ▶ **Débit:** nombre de processus finis par unité de temps
- ▶ **Utilisation du CPU:** pourcentage du temps où le CPU est occupé
- ▶ ...

Processus - Ordonnancement

First Come, First Served - FCFS

Premier arrivé, premier servi (First Come, First Served - FCFS)

- ▶ Algorithme non préemptif
- ▶ File d'attente FIFO pour les processus prêts
- ▶ Facile à comprendre et à programmer
- ▶ Intrinsèquement équitable pour des processus équivalents (même durée, même priorité)

Processus - Ordonnancement

First Come, First Served - FCFS

First Come, First Served - FCFS

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	1	2
B	2	5
C	5	1
D	3	0
E	4	4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Processus - Ordonnancement

First Come, First Served - FCFS

First Come, First Served - FCFS

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	1	2
B	2	5
C	5	1
D	3	0
E	4	4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	D	-	-	-	-	-	-	-	-	-	-	-	-

Processus - Ordonnancement

First Come, First Served - FCFS

First Come, First Served - FCFS

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	1	2
B	2	5
C	5	1
D	3	0
E	4	4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	D	C	C	C	C	C	-	-	-	-	-	-	-

Processus - Ordonnancement

First Come, First Served - FCFS

First Come, First Served - FCFS

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	1	2
B	2	5
C	5	1
D	3	0
E	4	4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	D	C	C	C	C	C	A	-	-	-	-	-	-

Processus - Ordonnancement

First Come, First Served - FCFS

First Come, First Served - FCFS

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	1	2
B	2	5
C	5	1
D	3	0
E	4	4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	D	C	C	C	C	C	A	E	E	E	E	-	-

Processus - Ordonnancement

First Come, First Served - FCFS

First Come, First Served - FCFS

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	1	2
B	2	5
C	5	1
D	3	0
E	4	4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	D	C	C	C	C	C	A	E	E	E	E	B	B

Processus - Ordonnancement

First Come, First Served - FCFS

First Come, First Served - FCFS

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	1	2
B	2	5
C	5	1
D	3	0
E	4	4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	D	C	C	C	C	C	A	E	E	E	E	B	B

$$\text{Temps d'attente moyen} = \frac{1}{N} \sum_i^N Tf_i - (t_i + d_i) = \frac{1}{5}(0 + 2 + 6 + 5 + 8) = 4.2$$

Processus - Ordonnancement

First Come, First Served - FCFS

First Come, First Served - FCFS

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	1	2
B	2	5
C	5	1
D	3	0
E	4	4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	D	C	C	C	C	C	A	E	E	E	E	B	B

$$\text{Temps d'attente moyen} = \frac{1}{N} \sum_i^N Tf_i - (t_i + d_i) = \frac{1}{5}(0 + 2 + 6 + 5 + 8) = 4.2$$

$$\text{Temps de rotation moyen} = \frac{1}{N} \sum_i^N Tf_i - t_i = \frac{1}{5}(3 + 7 + 7 + 9 + 10) = 7.2$$

Processus - Ordonnancement

First Come, First Served - FCFS

Inconvénients

- ▶ Méthode peu performante car temps d'attente assez élevé
- ▶ Les processus courts qui se trouvent à la fin de la file d'attente doivent attendre la fin du processus long qui les précédent

Processus - Ordonnancement

Short Job First - SJF

Le plus court d'abord (Short Job First - SJF)

- ▶ Algorithme non préemptif
- ▶ Choisir la tâche la plus courte parmi les tâches prêtes
- ▶ En cas d'égalité, appliquer l'algorithme FCFS

Avantages

- ▶ Bien adapté au traitement par lots de processus
- ▶ Temps maximaux d'exécution connus ou fixés par les utilisateurs
- ▶ Offre un meilleur temps moyen d'attente

Processus - Ordonnancement

Short Job First - SJF

Short Job First - SJF

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	1	2
B	2	5
C	5	1
D	3	0
E	4	4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Processus - Ordonnancement

Short Job First - SJF

Short Job First - SJF

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	1	2
B	2	5
C	5	1
D	3	0
E	4	4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	D	-	-	-	-	-	-	-	-	-	-	-	-

Processus - Ordonnancement

Short Job First - SJF

Short Job First - SJF

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	1	2
B	2	5
C	5	1
D	3	0
E	4	4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	D	A	-	-	-	-	-	-	-	-	-	-	-

Processus - Ordonnancement

Short Job First - SJF

Short Job First - SJF

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	1	2
B	2	5
C	5	1
D	3	0
E	4	4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	D	A	E	E	E	E	-	-	-	-	-	-	-

Processus - Ordonnancement

Short Job First - SJF

Short Job First - SJF

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	1	2
B	2	5
C	5	1
D	3	0
E	4	4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	D	A	E	E	E	E	B	B	-	-	-	-	-

Processus - Ordonnancement

Short Job First - SJF

Short Job First - SJF

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	1	2
B	2	5
C	5	1
D	3	0
E	4	4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	D	A	E	E	E	B	B	C	C	C	C	C	C

Processus - Ordonnancement

Short Job First - SJF

Short Job First - SJF

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	1	2
B	2	5
C	5	1
D	3	0
E	4	4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	D	A	E	E	E	E	B	B	C	C	C	C	C

$$\text{Temps d'attente moyen} = \frac{1}{N} \sum_i^N Tf_i - (t_i + d_i) = \frac{1}{5}(0 + 9 + 1 + 0 + 3) = 2.6$$

Processus - Ordonnancement

Short Job First - SJF

Short Job First - SJF

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	1	2
B	2	5
C	5	1
D	3	0
E	4	4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	D	A	E	E	E	E	B	B	C	C	C	C	C

$$\text{Temps d'attente moyen} = \frac{1}{N} \sum_i^N Tf_i - (t_i + d_i) = \frac{1}{5}(0 + 9 + 1 + 0 + 3) = 2.6$$

$$\text{Temps de rotation moyen} = \frac{1}{N} \sum_i^N Tf_i - t_i = \frac{1}{5}(3 + 14 + 2 + 4 + 5) = 5.6$$

Processus - Ordonnancement

Shortest Remaining Time First - SRTF

Le plus petit temps restant (Shortest Remaining Time First - SRTF)

- ▶ Algorithme préemptif
- ▶ Choisir la tâche que le temps d'exécution restant est le plus court
- ▶ Si plus courte que celle en cours, alors suspension de la tâche en cours et démarrage de la nouvelle tâche
- ▶ C'est la forme préemptive de l'algorithme SJF

Processus - Ordonnancement

Shortest Remaining Time First - SRTF

Shortest Remaining Time First - SRTF

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	1	2
B	2	5
C	5	1
D	3	0
E	4	4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Processus - Ordonnancement

Shortest Remaining Time First - SRTF

Shortest Remaining Time First - SRTF

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	1	2
B	2	5
C	5	1
D	3	0
E	4	4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	D	-	-	-	-	-	-	-	-	-	-	-	-

Processus - Ordonnancement

Shortest Remaining Time First - SRTF

Shortest Remaining Time First - SRTF

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	1	2
B	2	5
C	5	1
D	3	0
E	4	4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	D	A	-	-	-	-	-	-	-	-	-	-	-

Processus - Ordonnancement

Shortest Remaining Time First - SRTF

Shortest Remaining Time First - SRTF

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	1	2
B	2	5
C	5	1
D	3	0
E	4	4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	D	A	E	-	-	-	-	-	-	-	-	-	-

Processus - Ordonnancement

Shortest Remaining Time First - SRTF

Shortest Remaining Time First - SRTF

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	1	2
B	2	5
C	5	1
D	3	0
E	4	4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	D	A	E	B	B	-	-	-	-	-	-	-	-

Processus - Ordonnancement

Shortest Remaining Time First - SRTF

Shortest Remaining Time First - SRTF

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	1	2
B	2	5
C	5	1
D	3	0
E	4	4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	D	A	E	B	B	E	E	E	-	-	-	-	-

Processus - Ordonnancement

Shortest Remaining Time First - SRTF

Shortest Remaining Time First - SRTF

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	1	2
B	2	5
C	5	1
D	3	0
E	4	4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	D	A	E	B	B	E	E	E	C	C	C	C	C

Processus - Ordonnancement

Shortest Remaining Time First - SRTF

Shortest Remaining Time First - SRTF

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	1	2
B	2	5
C	5	1
D	3	0
E	4	4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	D	A	E	B	B	E	E	E	C	C	C	C	C

$$\text{Temps d'attente moyen} = \frac{1}{N} \sum_i^N Tf_i - (t_i + d_i) = \frac{1}{5}(0 + 9 + 1 + 2 + 0) = 2.4$$

Processus - Ordonnancement

Shortest Remaining Time First - SRTF

Shortest Remaining Time First - SRTF

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	1	2
B	2	5
C	5	1
D	3	0
E	4	4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	D	A	E	B	B	E	E	E	C	C	C	C	C

$$\text{Temps d'attente moyen} = \frac{1}{N} \sum_i^N Tf_i - (t_i + d_i) = \frac{1}{5}(0 + 9 + 1 + 2 + 0) = 2.4$$

$$\text{Temps de rotation moyen} = \frac{1}{N} \sum_i^N Tf_i - t_i = \frac{1}{5}(3 + 14 + 2 + 6 + 2) = 5.4$$

Processus - Ordonnancement

Round Robin - RR

Tourniquet ou circulaire (Round Robin - RR)

- ▶ Algorithme préemptif
- ▶ Algorithme ancien, simple, fiable et très utilisé
- ▶ Mémorisation des tâches en état Ready dans une file du type FIFO

Principe

- ▶ Traitement du processus en tête de file, pendant un **quantum de temps**
- ▶ Si processus bloqué ou terminé avant la fin de son quantum, alors traitement du processus en tête de file
- ▶ Si processus pas terminé au bout du quantum, alors exécution suspendue, insertion en queue de file et traitement du suivant
- ▶ insertion en queue de file des nouveaux processus et des processus en état Waiting

Processus - Ordonnancement

Round Robin - RR

Round Robin - RR

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	1	2
B	2	5
C	5	1
D	3	0
E	4	4

Quantum = 2 unités de temps

Processus Ordonnancement

Round Robin - RR

D

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Processus Ordonnancement

Round Robin - RR



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	-	-	-	-	-	-	-	-	-	-	-	-	-

Processus Ordonnancement

Round Robin - RR



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	C	C	-	-	-	-	-	-	-	-	-	-	-

Processus Ordonnancement

Round Robin - RR



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	C	C	A	-	-	-	-	-	-	-	-	-	-

Processus Ordonnancement

Round Robin - RR



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	C	C	A	D	-	-	-	-	-	-	-	-	-

Processus Ordonnancement

Round Robin - RR



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	C	C	A	D	E	E	-	-	-	-	-	-	-

Processus Ordonnancement

Round Robin - RR



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	C	C	A	D	E	E	C	C	-	-	-	-	-

Processus Ordonnancement

Round Robin - RR



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	C	C	A	D	E	E	C	C	B	B	-	-	-

Processus Ordonnancement

Round Robin - RR

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	-
D	D	C	C	A	D	E	E	C	C	B	B	E	E	-	

Processus Ordonnancement

Round Robin - RR

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	C	C	A	D	E	E	C	C	B	B	E	E	C

Processus Ordonnancement

Round Robin - RR

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	C	C	A	D	E	E	C	C	B	B	E	E	C

$$\text{Temps d'attente moyen} = \frac{1}{N} \sum_i^N Tf_i - (t_i + d_i) = \frac{1}{5}(3 + 9 + 2 + 6 + 5) = 5$$

Processus Ordonnancement

Round Robin - RR

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D	D	C	C	A	D	E	E	C	C	B	B	E	E	C

$$\text{Temps d'attente moyen} = \frac{1}{N} \sum_i^N Tf_i - (t_i + d_i) = \frac{1}{5}(3 + 9 + 2 + 6 + 5) = 5$$

$$\text{Temps de rotation moyen} = \frac{1}{N} \sum_i^N Tf_i - t_i = \frac{1}{5}(6 + 14 + 3 + 10 + 7) = 8$$

Processus - Ordonnancement

Ordonnancement par priorités

Ordonnancement par priorités

- ▶ Certains processus peuvent être plus urgents que d'autres
- ▶ Choix du processus à élire dépend des priorités des processus prêts
- ▶ Regroupement des processus de même priorité dans une file du type FIFO
- ▶ Autant de files que de niveaux de priorité
- ▶ Choix du processus le plus prioritaire en tête de file
- ▶ En général, utilisation de l'algorithme RR pour les processus de même priorité

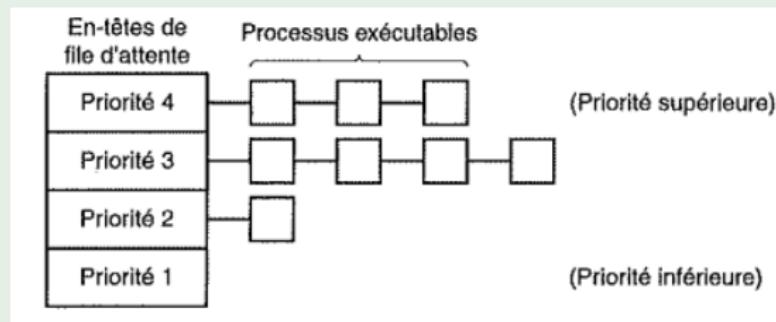
Inconvénient

CPU monopolisé par les processus de priorité élevée ⇒ priorités dynamiques (réduire au fil du temps et de l'exécution)

Processus - Ordonnancement

Ordonnancement par priorités

Ordonnancement par priorités



Inconvénient

CPU monopolisé par les processus de priorité élevée \Rightarrow priorités dynamiques (réduire au fil du temps et de l'exécution)

Exemple: définir une priorité de $1/f$ avec f est la fraction du dernier quatum utilisé par le processus

Processus - Ordonnancement

Ordonnancement par loterie

Ordonnancement par loterie (tirage au sort)

- ▶ Algorithme préemptif ou non
- ▶ Basé sur un tirage aléatoire
- ▶ Un nombre de tickets est attribué à chaque processus (proportionnel à son priorité ou à sa durée, par exemple)
- ▶ Un numéro de ticket est tiré au hasard par quantum et détermine le processus à traiter

Processus - Ordonnancement

Exercice

First Come, First Served - FCFS

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	5	6
B	3	0
C	2	0
D	1	2
E	5	3

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Processus - Ordonnancement

Exercice

First Come, First Served - FCFS

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	5	6
B	3	0
C	2	0
D	1	2
E	5	3

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
B	B	B	C	C	D	E	E	E	E	E	A	A	A	A	A

Processus - Ordonnancement

Exercice

Short Job First - SJF

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	5	6
B	3	0
C	2	0
D	1	2
E	5	3

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Processus - Ordonnancement

Exercice

Short Job First - SJF

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	5	6
B	3	0
C	2	0
D	1	2
E	5	3

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
C	C	D	B	B	B	E	E	E	E	E	A	A	A	A	A

Processus - Ordonnancement

Exercice

Shortest Remaining Time First - SRTF

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	5	6
B	3	0
C	2	0
D	1	2
E	5	3

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Processus - Ordonnancement

Exercice

Shortest Remaining Time First - SRTF

Processus	Durée (d_i)	Date d'arrivée (t_i)
A	5	6
B	3	0
C	2	0
D	1	2
E	5	3

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
C	C	D	B	B	B	E	E	E	E	E	A	A	A	A	A

Communication Inter-processus

Problématiques

Processus non forcément isolés:

- ▶ **Attente d'exécution d'un autre processus:** un processus produit des résultats qui doivent être traités par un autre
- ▶ **Inter-blocage (deadlock):** “le processus P1 utilise la ressource R2 et le processus P2 la ressource R1 or R1 attend le retour de P1 et R2 celui de P2”
- ▶ **Condition de concurrence:** situation où 2 ou plusieurs processus accèdent (en lecture et/ou en écriture) à des données partagées
- ▶ **Envoi de message:** un processus envoie un message à un processus en attente de ce message
- ▶ La politique d'ordonnancement influe sur les résultats

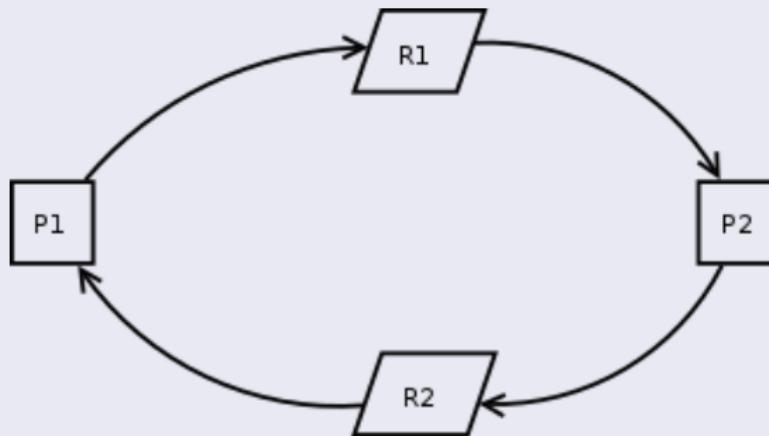
⇒ ceci nécessite une synchronisation entre les processus pour garder leurs états cohérents.

Communication Inter-processus

Problématiques

Processus non forcément isolés:

- ▶ **Inter-blocage (deadlock):** “le processus P1 utilise la ressource R2 et le processus P2 la ressource R1 or R1 attend le retour de P1 et R2 celui de P2”



Communication Inter-processus

Problématiques

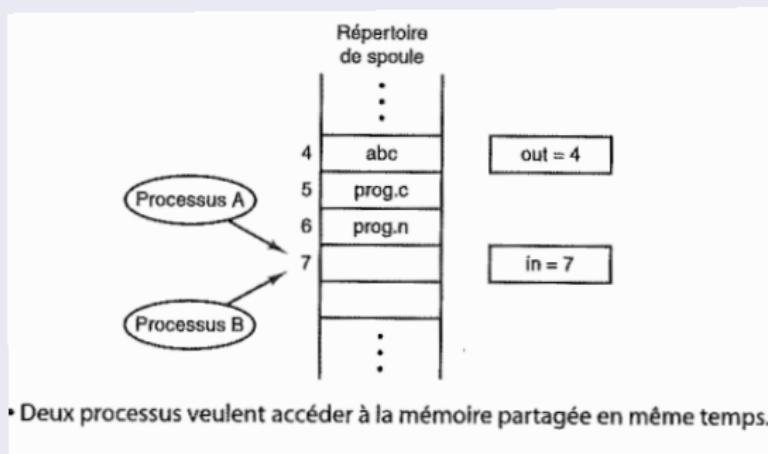
Processus non forcément isolés:

- ▶ **Condition de concurrence:** situation où 2 ou plusieurs processus accèdent (en lecture et/ou en écriture) à des données partagées

Exemple: spool d'impression géré par des variables:

out : prochain fichier à imprimer

in : prochaine entrée libre du spool



Communication Inter-processus

Définitions

Exclusion mutuelle

Méthode qui permet de s'assurer que si un processus utilise **une ressource partagée**, les autres processus seront **exclus** de cette activité

Ressource critique

Un élément logiciel ou matériel utilisé par le système d'exploitation

- ▶ une zone mémoire
- ▶ un fichier
- ▶ un périphérique
- ▶ ...

Section critique

- ▶ portion de code qui manipulent des ressources critiques
- ▶ exécutée en exclusion mutuelle (un seul processus à la fois)

Communication Inter-processus

Sections critiques

Règles d'une section critique

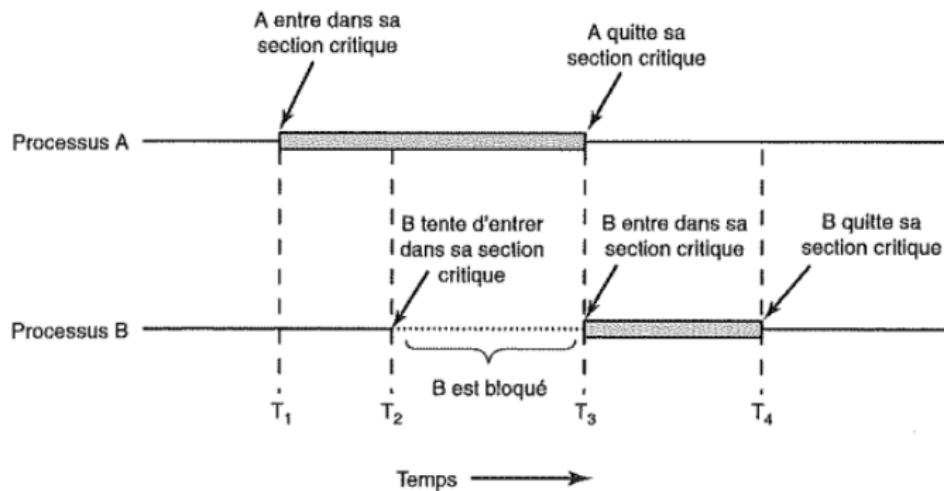
Pour éviter les incohérences des conditions de concurrence, quatre conditions doivent être réunies:

- ① Deux processus ne peuvent être **simultanément** dans une même section critique
- ② Aucune supposition quand au temps d'exécution d'un processus et au nombre de processeurs
- ③ Aucun processus s'exécutant à l'extérieur de sa section critique ne doit bloquer d'autres processus
- ④ Aucun processus ne doit attendre indéfiniment pour entrer dans sa section critique

Communication Inter-processus

Sections critiques

Exclusion mutuelle à l'aide des sections critiques:



Communication Inter-processus

Mécanismes d'exclusion mutuelle

Solution 1: désactivation des interruptions

Suspendre toutes les interruptions (la préemption) dès qu'un processus rentre dans une section critique et jusqu'à sa sortie

Problèmes

- ▶ le processus dans la section critique bloque tous les autres processus
- ▶ les drivers de périphériques sont aussi bloqués
- ▶ si l'architetture dispose de plusieurs processeurs, alors il faut bloquer tous les processeurs
- ▶ beaucoup de pouvoir donné pour un processus utilisateur!

Communication Inter-processus

Mécanismes d'exclusion mutuelle

Solution 2 : variable de verrou

- ▶ définition d'une variable booléenne (0 ou 1)
- ▶ si la variable est égale à 0, le procesus peut entrer dans sa section critique
- ▶ si la variable est égale à 1, le procesus attend
- ▶ en entrant dans la section, le procesus bascule la variable à 1
- ▶ en sortant, il bascule la variable à 0

```
tant que (verrou = 1) attendre  
verrou = 1  
execution de la section critique  
verrou = 0
```

Communication Inter-processus

Mécanismes d'exclusion mutuelle

Solution 2 : variable de verrou

En assembleur (entrée en région critique)

attente:

```
LEA AL, [verrou]
```

```
CMP AL, 0x01
```

```
JE attente
```

```
MOV [verrou], 0x01
```

Problème

- ▶ l'affectation de la variable de verrouillage est critique
- ▶ si 2 processus consultent la variable avant que l'un d'eux affecte la variable à vrai
- ▶ les 2 processus peuvent se trouver simultanément dans leurs sections critiques

Communication Inter-processus

Mécanismes d'exclusion mutuelle

Solution 3: alternance stricte

- ▶ définition d'une variable, qui indique quel processus est autorisé à s'exécuter:
 - ▶ si la variable est égale à 0, le processus 0 peut s'exécuter
 - ▶ si la variable est égale à 1, le processus 1 peut s'exécuter
 - ▶ en sortant de sa section critique, le processus change la variable

Exemple de 2 processus

la variable turn indique quel processus peut s'exécuter, avec processus 0 en (a) et processus 1 en (b):

```
while (TRUE) {
    while (turn != 0) /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

```

while (TRUE) {
    while (turn != 1) /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}

```

(b)

Communication Inter-processus

Mécanismes d'exclusion mutuelle

Solution 3: alternance stricte

Exemple de 2 processus

```
while (TRUE) {
    while (turn != 0) /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1) /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Problème

- ▶ lorsque l'un des deux processus est beaucoup plus lent que l'autre, la règle 3 (des sections critiques) n'est pas satisfaite

Communication Inter-processus

Mécanismes d'exclusion mutuelle

Solution 4: Dekker, 1965

- ▶ une variable *Tour* indique quel processus a le droit de faire une demande
- ▶ chaque processus possède une variable de demande d'entrée dans la section critique
- ▶ si un processus P fait une demande, on vérifie si un autre processus est dans la section critique
- ▶ si c'est cas, on attend que ce soit le tour de P et on renouvelle la demande
- ▶ à la fin de la section critique, on donne la main à un autre processus

Problème

Difficile à généraliser

Communication Inter-processus

Mécanismes d'exclusion mutuelle

Algorithme de Dekker à 2 processus P0 et P1

```
► //Initialisation globale
Veut_entrer[0] = FAUX, Veut_entrer[1] = FAUX, Tour = 0
► //Entree dans la section critique du processus P0
Veut_entrer[0] = VRAI
tant que (Veut_entrer[1] est VRAI)
si Tour = 1
    Veut_entrer[0] = FAUX
    tant que (Tour = 1) attendre
    Veut_entrer[0] = VRAI
► //Section critique du processus P0
► //Sortie de la section critique du processus P0
Veut_entrer[0] = FAUX
Tour = 1
```

Communication Inter-processus

Mécanismes d'exclusion mutuelle

Solution 5: Peterson, 1981

même principe que l'algorithme de Dekker, mais en plus simple.

Algorithme de Peterson à 2 processus P0 et P1

- ▶ //Initialisation globale
Veut_entrer[0] = FAUX, Veut_entrer[1] = FAUX, Tour = 0
- ▶ //Entree dans la section critique du processus P0
Veut_entrer[0] = VRAI
Tour = 0
tant que (Veut_entrer[1] = VRAI et Tour = 0) attendre
- ▶ //Section critique du processus P0
- ▶ //Sortie de la section critique du processus P0
Veut_entrer[0] = FAUX

Communication Inter-processus

Mécanismes d'exclusion mutuelle

Solution 5: Peterson, 1981

P0	P1
1. Ecrire (VRAI, veut_entrer[0]);	1'. Ecrire (VRAI, veut_entrer[1]);
2. Ecrire (0, tour);	2'. Ecrire (1, tour);
3. a0 = Lire (veut_entrer[1]);	3'. a1 = Lire (veut_entrer[0]);
4. Si ($\neg a0$) aller a 7;	4'. Si ($\neg a1$) aller a 7';
5. t0 = Lire(tour);	5'. t1 = Lire(tour);
6. Si ($t0 == 0$) aller a 3;	6'. Si ($t1 == 1$) aller a 3';
7. Section_critique();	7'. Section_critique();
8. Ecrire (FAUX, veut_entrer[0]);	8'. Ecrire (FAUX, veut_entrer[1]);

Cas 1

- ▶ P0 exécute 3 avant que P1 n'exécute 1'
- ▶ les valeurs obtenues par les deux processus sont alors :
 $a0 = FAUX$, $t0 = 0$, $a1 = VRAI$ et $t1 = 1$
- ▶ dans ce cas, on vérifie que P0 entre en section critique, P1 reste bloqué

Communication Inter-processus

Mécanismes d'exclusion mutuelle

Solution 5: Peterson, 1981

P0	P1
1. Ecrire (VRAI, veut_entrer[0]);	1'. Ecrire (VRAI, veut_entrer[1]);
2. Ecrire (0, tour);	2'. Ecrire (1, tour);
3. a0 = Lire (veut_entrer[1]);	3'. a1 = Lire (veut_entrer[0]);
4. Si (\neg a0) aller a 7;	4'. Si (\neg a1) aller a 7';
5. t0 = Lire(tour);	5'. t1 = Lire(tour);
6. Si ($t0 == 0$) aller a 3;	6'. Si ($t1 == 1$) aller a 3';
7. Section_critique();	7'. Section_critique();
8. Ecrire (FAUX, veut_entrer[0]);	8'. Ecrire (FAUX, veut_entrer[1]);

Cas 2

- ▶ P1 exécute 1' avant que P0 n'exécute 3 et 2' après que P0 ait exécuté 5, alors : a0 = VRAI, t0 = 0, a1 = VRAI et t1 = 1
- ▶ aucun des deux processus ne rentre dans la section critique
- ▶ immédiatement après le processus P0 obtient: a0 = VRAI et t0 = 1
- ▶ ainsi, P0 entre en section critique et P1 bloqué

Communication Inter-processus

Mécanismes d'exclusion mutuelle

Solution 5: Peterson, 1981

P0	P1
1. Ecrire (VRAI, veut_entrer[0]);	1'. Ecrire (VRAI, veut_entrer[1]);
2. Ecrire (0, tour);	2'. Ecrire (1, tour);
3. a0 = Lire (veut_entrer[1]);	3'. a1 = Lire (veut_entrer[0]);
4. Si (!a0) aller a 7;	4'. Si (!a1) aller a 7';
5. t0 = Lire(tour);	5'. t1 = Lire(tour);
6. Si (t0 == 0) aller a 3;	6'. Si (t1 == 1) aller a 3';
7. Section_critique();	7'. Section_critique();
8. Ecrire (FAUX, veut_entrer[0]);	8'. Ecrire (FAUX, veut_entrer[1]);

Cas 3

- ▶ P1 exécute 1' avant que P0 n'exécute 3, et 2' entre 2 et 5
- ▶ on a dans ce cas: a0 = VRAI, t0 = 1, a1 = VRAI et t1 = 1
- ▶ ainsi P0 entre en section critique et P1 reste bloqué

Communication Inter-processus

Mécanismes d'exclusion mutuelle

Solution 5: Peterson, 1981

P0	P1
1. Ecrire (VRAI, veut_entrer[0]);	1'. Ecrire (VRAI, veut_entrer[1]);
2. Ecrire (0, tour);	2'. Ecrire (1, tour);
3. a0 = Lire (veut_entrer[1]);	3'. a1 = Lire (veut_entrer[0]);
4. Si (!a0) aller a 7;	4'. Si (!a1) aller a 7';
5. t0 = Lire(tour);	5'. t1 = Lire(tour);
6. Si (t0 == 0) aller a 3;	6'. Si (t1 == 1) aller a 3';
7. Section_critique();	7'. Section_critique();
8. Ecrire (FAUX, veut_entrer[0]);	8'. Ecrire (FAUX, veut_entrer[1]);

Cas 4

- ▶ P1 exécute 1' et 2' avant que P0 n'exécute 2
- ▶ donc: a0 = VRAI, t0 = 0, a1 = VRAI et t1 = 0
- ▶ P1 entre en section critique et P0 reste bloqué

Communication Inter-processus

Mécanismes d'exclusion mutuelle

Solution 5: Peterson, 1981

Implémentation en language C:

```
#define FALSE 0
#define TRUE 1
#define N    2           /* nombre de processus */
int turn;                /* à qui le tour ? */
int interested[N];        /* toutes valeurs initialement
                           0 (FALSE) */

void enter_region(int process); /* le processus est 0 ou 1 */
{
    int other;           /* nombre des autres processus */
    other = 1 - process; /* l'opposé du processus */
    interested[process] = TRUE; /* montre que vous êtes intéressé */
    turn = process;       /* définit indicateur */
    while (turn == process && interested[other] == TRUE)
        /* instruction null */ ;
}

void leave_region(int process) /* processus : qui quitte */
{
    interested[process] = FALSE; /* indique départ de la
                                 section critique */
}
```

Communication Inter-processus

Mécanismes d'exclusion mutuelle

Solution 5: Peterson, 1981

Exemple:

P0



P1



V[0]



V[1]



Tour



Communication Inter-processus

Mécanismes d'exclusion mutuelle

Généralisation à N processus - Solution 5: Peterson, 1981

- ▶ //Initialisation globale
 - in_stage: tableau de N entiers initialises à -1
 - last_process: tableau de N entiers initialises à 0
- ▶ //Entrée dans la section critique du processus Pi
 - Pour j de 0 à N
 - in_stage[i] = j; last_process[j] = i
 - Pour k de 0 à N
 - Si k != i
 - Tant que in_stage[k] >= in_stage[i] ET
 - last_process[j] = i
 - attendre
 - ▶ //Section critique du processus Pi
 - ▶ //Sortie de la section critique du processus Pi
 - in_stage[i] = -1

Communication Inter-processus

Mécanismes d'exclusion mutuelle

Solution 6: instruction TSL (*Test and Set Lock*)

- ▶ rendre atomique le test et l'affection et de la variable Verrou
- ▶ nécessite l'aide du matériel

Instruction TSL

- ▶ TSL RX, LOCK
- ▶ lit le contenu du mot mémoire LOCK, l'affecte au registre RX, puis stocke une valeur différente de 0 à l'adresse mémoire de LOCK
- ▶ se fait au niveau du processeur:
 - ▶ TSL ne peut pas être interrompu (puisque c'est une seule instruction)
 - ▶ le CPU vérifie le bus mémoire pour interdire aux autres processeurs d'accéder à la mémoire et de modifier le verrou
 - ▶ nécessite des bus mémoire spécifiques

Communication Inter-processus

Mécanismes d'exclusion mutuelle

Entrer et sortir de la section critique à l'aide de l'instruction TSL

```
enter_region:  
    TSL REGISTER,LOCK  | copie lock dans le registre et la définit à 1  
    CMP REGISTER,#0    | lock était-elle à 0 ?  
    JNE enter_region   | si elle n'était pas à 0, boucle  
    RET | retourne à l'appelant ; entre en section critique  
  
leave_region:  
    MOVE LOCK,#0        | stocke un 0 dans lock  
    RET| retourne à l'appelant
```

- ▶ l'instruction TSL mets le verrou à 1 si c'est possible (s'il n'est pas verrouillé par un autre processus, sa valeur est à 0)
- ▶ la valeur du LOCK est copié dans un registre
- ▶ en sortant de sa section critique, le processus met le verrou à 0

Communication Inter-processus

Mécanismes d'exclusion mutuelle

Solution 6: instruction TSL

Les Intel x86 implémentent l'**instruction XCHG**, qui permet de façon atomique les contenus de deux emplacements (par exemple un registre et un mot mémoire).

```
enter_region:  
    MOVE REGISTER,#1    | mettre 1 dans le registre  
    XCHG REGISTER,LOCK   | échanger les contenus du registre et de lock  
    CMP REGISTER,#0      | lock était-elle à 0 ?  
    JNE enter_region     | si elle n'était pas à 0, boucle  
    RET | retourne à l'appelant ; entre en section critique  
  
leave_region:  
    MOVE LOCK,#0         | stocke un 0 dans lock  
    RET| retourne à l'appelant
```

Communication Inter-processus

Mécanismes d'exclusion mutuelle

Problème

Les solutions vues jusqu'à présent (TSL, XCHG, Peterson, Dekker) sont bonnes, mais ils ont un inconvénient de faire appel à **l'attente active**.

Examinons maintenant ...

Synchronisation de processus: Sommeil et activation

- ▶ mécanisme qui vise à bloquer l'exécution de certains processus si les ressources nécessaires à leur exécution ne sont pas disponibles
- ▶ une des plus simples primitives: le paire sleep et wakeup.

Communication Inter-processus

Mécanismes d'exclusion mutuelle

Problème du producteur - consommateur

- ▶ producteur/consommateur (ou lecteur/écrivain): deux processus partage un espace de taille fixe. L'un le remplit et l'autre le vide.
- ▶ possible de généraliser pour m producteur et n consommateur
- ▶ **problèmes:** l'espace est plein car le consommateur n'est pas assez rapide ou l'espace est vide car producteur n'est pas assez rapide

Communication Inter-processus

Mécanismes d'exclusion mutuelle et de synchronisation

Problème du producteur - consommateur

```
#define N 100                                /* nombre de connecteurs dans
                                                le tampon */
int count = 0;                                /* nombre d'éléments dans le
                                                tampon */

void producer(void) {
    int item;
    while (TRUE) {
        item = produce_item( );
        if (count == N) sleep( );
        insert_item(item);
        count = count + 1;
        if (count == 1) wakeup(consumer);
    }
}
void consumer(void) {
    int item;
    while (TRUE) {
        if (count == 0) sleep( );
        item = remove_item( );
        count = count - 1;
        if (count == N - 1) wakeup(producer); /* le tampon était plein ? */
        consume_item(item);
    }
}
```

Communication Inter-processus

Mécanismes d'exclusion mutuelle et de synchronisation

Solution : Sémaphore

- ▶ introduit par E. W. Dijkstra en 1965
- ▶ décompte le nombre de wakeup enregistrés pour usage ultérieur
- ▶ ce type de variable de comptage est appelé **sémaphore**
- ▶ Dijkstra a proposé 2 opérations: **down** et **up** (des généralisation de sleep et wakeup):
 - ▶ opération **down** sur un sémaphore s : si $s > 0$ décrémente s (utiliser un wakeup stocké), sinon ($s = 0$) met le processus en sommeil sans que **down** termine
 - ▶ opération **up** sur un sémaphore s : si un ou plusieurs processus sont en sommeil sur le sémaphore, réveiller l'un d'eux, sinon incrémenter le s

Les activités de vérification, modification et mise en sommeil (ou réveil) de **down** et **up** sont atomique.

Communication Inter-processus

Mécanismes d'exclusion mutuelle et de synchronisation

Problème du producteur - consommateur et les sémaphores

utilisation de 3 sémaphores: full, empty, et mutex.

mutex est un **sémaphore binaire** (prend que 2 valeurs et est initialisé à 1). Il est utilisé pour l'exclusion mutuelle.

```
#define N 100          /* nombre d'emplacements dans le tampon */
typedef int semaphore; /* les sémaphores sont un type de variable int
                        spécial */
semaphore mutex = 1;   /* contrôle l'accès à la section critique */
semaphore empty = N;  /* compte les emplacements vides dans le tampon */
semaphore full = 0;   /* compte les emplacements pleins */
void producer(void)
{
    int item;
    while (TRUE) {           /* TRUE est la constante 1 */
        item = produce_item( ); /* génère quelque chose à placer
                                dans le tampon */
        down(&empty);         /* décrémente le décompte des
                                emplacements vides */
        down(&mutex);         /* entre en section critique */
        insert_item(item);    /* place un nouvel élément dans le
                                tampon */
        up(&mutex);           /* quitte la section critique */
        up(&full);            /* incrémente le décompte des
                                emplacements pleins */
    }
}
```

Communication Inter-processus

Mécanismes d'exclusion mutuelle et de synchronisation

Problème du producteur - consommateur et les sémaphores

utilisation 3 sémaphores: full, empty, et mutex.

mutex est un **sémaphore binaire** (prend que 2 valeurs et est initialisé à 1). Il est utilisé pour l'exclusion mutuelle.

```
void consumer(void)
{
    int item;
    while (TRUE) {           /* boucle sans fin */
        down(&full);         /* décrémente le décompte des
                               /* emplacements pleins */
        down(&mutex);         /* entre en section critique */
        item = remove_item(); /* prélève un élément dans le tampon */
        up(&mutex);          /* quitte la section critique */
        up(&empty);           /* incrémente le décompte des
                               /* emplacements vides */
        consume_item(item);   /* fait quelque chose avec l'élément */
    }
}
```

Communication Inter-processus

Mécanismes d'exclusion mutuelle et de synchronisation

Solution : Sémaphore

- ▶ originellement dans le papier de Dijkstra:
 - ▶ opérateur **down**: opérateur **P** (Proberen - test ou Wait), "Puis-je?"
 - ▶ opérateur **up**: opérateur **V** (Verhogen - incrément ou Signal), "Vas-y"



Edsger Dijkstra

Communication Inter-processus

Mécanismes d'exclusion mutuelle et de synchronisation

Les mutex

- ▶ version simplifiée des sémaphores
- ▶ prend en charge l'exclusion mutuelle (accès concurrent à des ressources partagées)
- ▶ variable qui prend 2 états: déverrouillé (0) et verrouillé (1)
- ▶ le processus qui a besoin d'accéder à sa section critique invoque **mutex_lock**: si le mutex est à 0, l'appel réussit, sinon le processus est bloqué
- ▶ le processus ayant terminé de sa section critique invoque **mutex_unlock**

std::mutex du C++

Classe qui propose les deux primitives lock et unlock.

Communication Inter-processus

Mécanismes d'exclusion mutuelle et de synchronisation

une implémentation des mutex avec TSL

```
mutex_lock:
    TSL REGISTER,MUTEX | copie mutex pour enregistrer et définir mutex à 1
    CMP REGISTER,#0    | mutex était-il à 0 ?
    JZE ok             | si oui, il n'était pas verrouillé, donc retourne
    CALL thread_yield  | mutex est occupé ; planifie un autre thread
    JMP mutex_lock     | réessaye plus tard
ok:
    RET                | retourne à l'appelant ; entrée en section critique

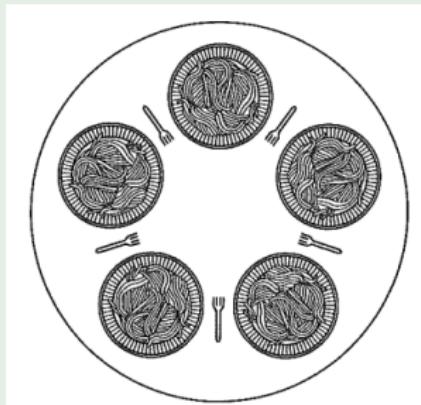
mutex_unlock:
    MOVE MUXTEX,#0     | stocké un 0 dans mutex
    RET                | retourne à l'appelant
```

Communication Inter-processus

Problème du dîner des philosophes

Problème du dîner des philosophes

- ▶ N philosophes autour d'une table
- ▶ N baguettes / fourchettes
- ▶ un philosophe pense puis mange et recommence à penser, puis mange, ...
- ▶ pour manger, il faut 2 baguettes / fourchettes



Communication Inter-processus

Problème du dîner des philosophes

Mauvaise idée !

```
#define N 5
void philosopher(int i) /* nombre de philosophes */
{ /* i: numéro du philosophe,
   <retour listing> de 0 à 4 */
    while (TRUE) {
        penser(); /* le philosophe pense */
        prendre_fourchette(i); /* il prend sa fourchette gauche*/
        prendre_fourchette((i+1) % N); /* prend sa fourchette droite.
                                         <retour listing> % est le modulo */
        manger(); /* il mange des spaghetti */
        poser_fourchette(i); /* repose la fourchette gauche
                               <retour listing> sur la table */
        poser_fourchette((i+1) % N); /* repose la fourchette droite
                                         <retour listing> sur la table */
    }
}
```

- ▶ s'ils décident de manger au même moment
- ▶ ils commencent tous par récupérer la baguette se trouvant à leur gauche, par exemple
- ▶ ils sont alors **bloqués** en essayant de récupérer la baguette à droite

Communication Inter-processus

Problème du dîner des philosophes

Mauvaise idée !

```
#define N 5
void philosopher(int i)                                /* nombre de philosophes */
                                                       /* i: numéro du philosophe,
                                                       <retour listing> de 0 à 4 */

{
    while (TRUE) {
        penser( );                                     /* le philosophe pense */
        prendre_fourchette(i);                         /* il prend sa fourchette gauche*/
        prendre_fourchette((i+1) % N); /* prend sa fourchette droite.
                                         <retour listing> % est le modulo */
        manger( );                                    /* il mange des spaghetti */
        poser_fourchette(i);                          /* repose la fourchette gauche
                                                       <retour listing> sur la table */
        poser_fourchette((i+1) % N); /* repose la fourchette droite
                                         <retour listing> sur la table */
    }
}
```

Interblocage

Si tous les philosophes "prennent" la baguette de gauche alors tout le monde est bloqué

Communication Inter-processus

Problème du dîner des philosophes

Mauvaise idée 2

- ▶ après qu'un philosophus a pris sa fourchette gauche, le code détermine si la fourchette droite est disponible
 - ▶ si ce n'est pas le cas, le philosophe dépose la fourchette gauche, attend pendant un certain moment puis recommence le processus
- ⇒ une situation de famine

Famine

la situation où les programmes continuent de s'exécuter mais ne progresse jamais, se nomme **privation de ressources (ou famine)**.
les types de famines:

- ▶ famine "avérée" : un processus/thread est indéfiniment bloqué quelque soit l'exécution
- ▶ famine "probabiliste" : il y a une probabilité non nulle qu'un processus/thread reste bloqué longtemps

Communication Inter-processus

Problème du dîner des philosophes

Solution suivant les sémaphores

```
#define N 5                                /* nombre de philosophes */
#define GAUCHE (i+N-1)%N                     /* numéro du voisin de gauche de i */
#define DROITE (i+1)%N                       /* numéro du voisin de droite de i */
#define PENSE 0                               /* le philosophe pense */
#define FAIM 1                                /* le philosophe essaie de prendre
                                             <retour listing> les fourchettes */
#define MANGE 2                               /* le philosophe mange */
typedef int semaphore;                      /* les sémaphores sont des int spéciaux*/
int state[N];                             /* tableau pour suivre les états
                                             <retour listing> des philosophes */
semaphore mutex = 1;                      /* exclusion mutuelle pour les sections
                                             <retour listing> critiques */
semaphore s[N];                           /* un sémaphore par philosophe */
void philosophe(int i)                    /* i: numéro du philosophe, de 0 à N -1*/
{
    while (TRUE) {                         /* boucle sans fin */
        penser();                          /* le philosophe pense */
        prendre_fourchettes(i);           /* prend deux fourchettes ou bloque */
        manger();                          /* mange des spaghetti */
        poser_fourchettes(i);             /* repose les deux fourchettes sur
                                             <retour listing> la table */
    }
}
void prendre_fourchettes(int i)/* i: numéro du philosophe, de 0 à N -1*/
{
    down(&mutex);                        /* entre en section critique */
    state[i] = FAIM;                     /* enregistre le fait que le philosophe
                                             <retour listing> a faim */
    test(i);                            /* tente de prendre deux fourchettes */
```

Communication Inter-processus

Problème du dîner des philosophes

Solution suivant les sémaphores

```
    up(&mutex);           /* quitte la section critique */
    down(&s[i]);        /* bloque s'il n'a pas pu prendre
                           <retour listing> les fourchettes */
}
void poser_fourchettes(i) /* i: numéro du philosophe, de 0 à N -1*/
{
    down(&mutex);      /* entre en section critique */
    state[i] = PENSE;
    test(GAUCHE);
    test(DROITE);
    up(&mutex);        /* quitte la section critique */
}
void test(i)             /* i: numéro du philosophe, de 0 à N -1*/
{
    if (state[i] == FAIM && state[GAUCHE] != MANGE && state[DROITE]
        <retour listing> != MANGE) {
        state[i] = MANGE;
        up(&s[i]);
    }
}
```

La solution utilise un tableau de sémaphores, et un sémaphore binaire.

Communication Inter-processus

Inter-blocage (un autre exemple)

Problème du producteur - consommateur et les sémaphores
si les down(&mutex) et down(&empty) sont inversé ?

```
#define N 100          /* nombre d'emplacements dans le tampon */
typedef int semaphore; /* les sémaphores sont un type de variable int
spécial */
semaphore mutex = 1;   /* contrôle l'accès à la section critique */
semaphore empty = N;  /* compte les emplacements vides dans le tampon */
semaphore full = 0;   /* compte les emplacements pleins */
void producer(void)
{
    int item;
    while (TRUE) {           /* TRUE est la constante 1 */
        item = produce_item(); /* génère quelque chose à placer
dans le tampon */
        down(&empty);        /* décrémente le décompte des
emplacements vides */
        down(&mutex);        /* entre en section critique */
        insert_item(item);   /* place un nouvel élément dans le
tampon */
        up(&mutex);          /* quitte la section critique */
        up(&full);           /* incrémente le décompte des
emplacements pleins */
    }
}
```

Il faut être attentif dans l'emploi des sémaphore pour éviter ces situations inprévisibles

Communication Inter-processus

Moniteur

Moniteur

- ▶ approche pour la synchronisation **de haut niveau** des processus mise au point par P. B. Hansen et C. A. R. Hoare en 1974
- ▶ est une collection de procédures, variables, et de structures de données regroupé dans un module spécial (on peut les appeler sans y accéder à l'implémentation)

```
monitor exemple
    integer i;
    condition c;
    procedure producteur( );
    .
    .
    end;
    procedure consommateur( );
    .
    .
end monitor;
```

- ▶ à chaque instant, un seul processeur peut être actif dans un moniteur

Communication Inter-processus

Moniteur

Moniteur

- ▶ les compilateurs gèrent les moniteurs différemment. Ils implémentent des mutex et des sémaphores binaires pour l'exclusion mutuelle (mais ce n'est pas suffisant)
- ▶ les moniteurs introduisent des **variables de condition**: prennent en charge deux opérations wait et signal

Moniteur pour le problème du producteur/consommateur

```
monitor ProdMonitor
    int compteur = 0
    condition plein, vide
    retirer()
    déposer()
    ...
end monitor
```

Communication Inter-processus

Moniteur

Producteur

```
producteur(){  
    while (VRAI){  
        produire_objet()  
        ProdMonitor.deposer()  
    }  
}
```

consommateur

```
consommateur(){  
    while (VRAI){  
        ProdMonitor.retirer()  
        consommer_objet()  
    }  
}
```

Communication Inter-processus

Moniteur

deposer

```
deposer(){  
    si (compteur = N) wait(plein)  
    deposer_objet()  
    compteur = compteur + 1  
    si (compteur == 1) signal(vide)  
}
```

retirer

```
retirer(){  
    si (compteur = 0) wait(vide)  
    retirer_objet()  
    compteur = compteur - 1  
    if (compteur = N - 1) signal(plein)  
}
```

Communication Inter-processus

Echange de messages

Echange de messages

- ▶ emploie deux primitives: **send**, **receive** (des appels système)
- ▶ `send(destination, &message)`
- ▶ `receive(source, &message)`
- ▶ pour se prémunir des pertes de message, le récepteur envoie un **accusé de réception**. Si l'émetteur ne l'a pas reçu il retransmet le message
- ▶ un autre soucis est l'authentification des processus

Communication Inter-processus

Echange de messages

Echange de messages pour le problème du producteur/consommateur

Un maximum de N messages (= nombre d'emplacement dans la mémoire partagée). Côté consommateur:

```
#define N 100          /* nombre d'emplacements dans le
                        <retour listing> tampon */

void consumer(void)
{
    int item, i;
    message m;
    for (i = 0; i < N; i++) send(producer, &m); /* envoie N messages
                                                <retour listing> vides */
    while (TRUE) {
        receive(producer, &m);      /* récupère un message contenant
                                       <retour listing> un élément */
        item = extract_item(&m);    /* extrait l'élément du message */
        send(producer, &m);         /* retourne une réponse vide */
        consume_item(item);        /* utilise l'élément */
    }
}
```

Communication Inter-processus

Echange de messages

Echange de messages pour le problème du producteur/consommateur

Un maximum de N messages (= nombre d'emplacement dans la mémoire partagée). Côté producteur:

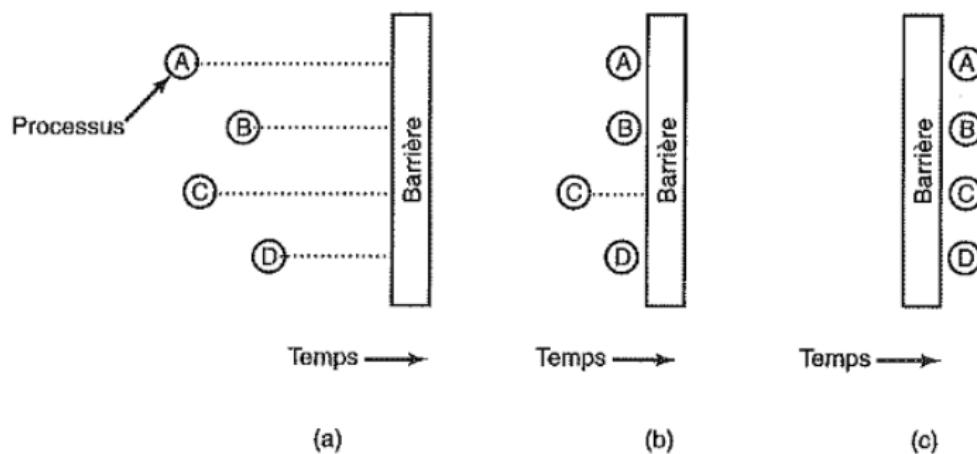
```
void producer(void)
{
    int item;
    message m; /* tampon des messages */
    while (TRUE) {
        item = produce_item(); /* génère quelque chose à placer dans
                               <retour listing> le tampon */
        receive(consumer, &m); /* attend l'arrivée d'un message
                               <retour listing> vide */

        build_message(&m, item);/* construit un message à envoyer */
        send(consumer, &m);    /* envoie l'élément au consommateur */
    }
}
```

Communication Inter-processus

Les barrières

processus approchant la barrière, une fois le dernier arrivé, tous les processus peuvent passer.



Plan

1 Introduction

2 Processeur et assembleur x86

3 Unix/Linux

4 Processus

5 Fichiers

Références

- ▶ Systèmes d'exploitation:
 - ▶ Tanenbaum, Andrew S., et al. Systèmes d'exploitation. Edition 2. Pearson Education, 2008.
- ▶ Assembleur:
 - ▶ Langage Assembleur PC - Paul Carter (2004). Traduction de Sébastien Le Ray, lien pdf ici.
- ▶ Unix/Linux:
 - ▶ Ritchie, D. M., & Thompson, K. (1974). The UNIX time-sharing system. Communications of the ACM, 17(7), 365-375.
 - ▶ Documentaire "AT&T Archives: The UNIX Operating System", lien ici.

Licence

Architecture et système d'exploitation

Under the "GNU Free Documentation License", Version 1.2 or any later.

Copyright ©2020-2021 - LISIC/ULCO
eric.ramat@univ-littoral.fr

Modified by Omar Rifki 2022-2023 - LISIC/ULCO
omar.rifki@univ-littoral.fr