Circuits intégrés
Gestion des processus

Sommaire

1. La loi de Moore
2. Microcontrôleurs
3. Systèmes sur puce
4. Gestion des processus
   - L'ordonnanceur
   - Programmation concurrente

Sommaire

1. La loi de Moore
2. Microcontrôleurs
3. Systèmes sur puce
4. Gestion des processus
   - L'ordonnanceur
   - Programmation concurrente

La loi de Moore

- La miniaturisation progressive des circuits électroniques a joué un rôle très important dans l'évolution de l'informatique.
- Depuis les années 1970 la *loi de Moore*[1] qui énonce que le nombre de transistors dans les micro-processeurs double tous les 2 ans, s'est avérée incroyablement juste.
- 1971 : le processeur 4004 d'Intel contient 2250 transistors.
  2016 : le Core i7 Broadwell-R contient environ 3,2 milliards de transistors, voir ici.
- Conséquences :
  - augmentation de la puissance de calcul des ordinateurs ;
  - baisse des coûts ;
  - baisse de la consommation énergétique ;
  - possibilité d'intégrer sur une même puce tous les composants d'un ordinateur.

1. Gordon Moore est cofondateur de la société Intel (États-Unis), un des leaders mondiaux des fabricants de semi-conducteurs (avec Samsung (Corée du Sud), TSMC (Taiwan). Actuellement, le seul fabricant franco-italien (ST-microelectronics, n'est plus dans le top-10)).

- Composants essentiels d'un ordinateur :
  - microprocesseur ;
  - mémoire (vive, morte) ;
  - interfaces d'entrées/sorties ;
  - alimentation.
- Actuellement, les *microcontrôleurs* (μC) et les systèmes sur puce [2] sont de tels systèmes, qui contiennent tous les composants d'un ordinateur sur un seul circuit intégré.

2. En anglais on parle de SoC, System on Chip.

Sommaire

1 La loi de Moore

2 Microcontrôleurs

3 Systèmes sur puce

4 Gestion des processus
- L'ordonnanceur
- Programmation concurrente

Microcontrôleurs

- Les caractéristiques physiques des μC sont en dessous de celles d'un ordinateur «classique» :
  - fréquence de quelques MHz (au lieu de qqs GHz) ;
  - mémoire vive de qqs Ko (au lieu de qqs Go) ;
  - consommation électrique de qqs centaines de mWh (contre quelques centaines de Wh pour ordinateur de bureau).
- Les μC sont utilisés dans des *systèmes embarqués* :
  - automobile (régulateur de vitesse p. ex.) ;
  - avionique (commandes de vol ?) ;
  - objets connectés (p.ex. montres) ;
  - robotique, domotique,. . .

- typiquement, on retrouve les μC dans des systèmes de régulation et ils utilisent des périphériques auxiliaires :
  - capteurs externes (ex. : capteur de vitesse pour un régulateur de vitesse) ;
  - convertisseurs A/N (Analogique/Numérique) ;
  - timer (déclenche une action à intervalles de temps réguliers. Ex. : mesure de vitesse) ;
  - actionneurs (ex. : vanne d'injection électronique pour un régulateur de vitesse).
- On ne retrouve pas de Système d'Exploitation (S.E.) sur ces μC et ils fonctionnent en *temps réel*.
- L'architecture de Harvard ($\neq$ de celle de von Neumann) qui caractérise les μC repose sur *deux types de mémoire* : une pour les données et une pour le programme, et deux bus à caractéristiques $\neq$, adaptées aux différents types de mémoire.

- Les μC sont aussi caractérisés par des CPU RISC (Reduced Instruction Set Computer ou Ordinateur à jeu d'instructions réduit), ce qui simplifie l'architecture globale.

Ex. 1 Rechercher la documentation technique des μC PIC 18F de la société Microchip. En particulier, trouver :

- le type et la capacité maximale de la mémoire programme ;
- la fréquence maximale d'horloge ;
- le type et la capacité maximale de la mémoire données ;
- le nombre maximum de ports d'entrées/sorties.

Sommaire

1 La loi de Moore

2 Microcontrôleurs

3 Systèmes sur puce

4 Gestion des processus
- L'ordonnanceur
- Programmation concurrente

Systèmes sur puce

- Contrairement aux μC, les systèmes sur puces (ou SoC) rassemblent sur un même circuit intégré tous les composants d'une carte mère d'ordinateur.
- Ils sont dotés d'un S.E.
- Leur puissance de calcul est comparable à celle d'un ordinateur : processeurs multicœurs cadencés à plusieurs GHz, présence de processeurs dédiés (graphique p.ex.), capacité mémoire en Go, tout cela sur quelques cm^2.
- leur architecture est celle de von Neumann, ils ont généralement un jeu d'instruction RISC.
- souvent ces SoC sont dotés de *deux bus de communication* :
  - un bus haute performance pour les échanges entre le CPU et les différents composants mémoire ;
  - un bus plus lent pour les communications avec les périphériques : GPS, capteurs, USB, modem, ethernet, wifi, bluetooth, audio, HDMI,...

- Les SoC sont présents sur les smartphones, tablettes et de plus en plus de périphériques embarqués (voitures, robots).
- Un inconvénient des SoC est qu'ils sont (pratiquement) non réparables et non extensibles.

Ex. 2 les téléphones/tablettes Apple utilisent le SoC A13 bionic. Déterminer, pour ce SoC :

- la précision de gravure, le nombre de transistors ;
- le type du jeu d'instructions (RISC ?), le nombre de cœurs, leur fréquence ;
- la présence d'un GPU, son nombre de cœurs ;
- la présence d'un processeur dédié à l'I.A., son nombre de cœurs ;
- la capacité de la RAM.

Sommaire

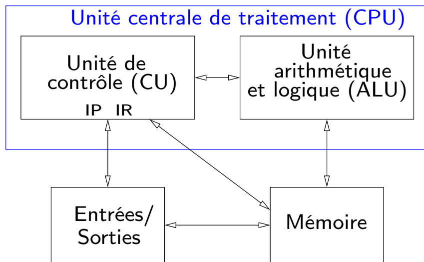
1. La loi de Moore
2. Microcontrôleurs
3. Systèmes sur puce
4. Gestion des processus
   - L'ordonnanceur
   - Programmation concurrente

Gestion des processus

- Rq : dans ce paragraphe (encore plus que dans les précédents, on ne fait qu' effleurer un sujet très vaste...)
- L'utilisation quotidienne que nous avons d'un ordinateur révèle qu'il semble capable (en fait son S.E.) d'exécuter plusieurs programmes en même temps :
  - saisir du texte dans un éditeur ;
  - télécharger/lire son courriel dans un client de messagerie ;
  - naviguer sur le web/télécharger un fichier ;
  - écouter de la musique/regarder une vidéo.
  - ...
- On parle d'*exécution concurrente* des programmes et de *S.E. multitâches*.
- L'*ordonnanceur* est la composante du S.E. qui gère cet aspect.

L'ordonnanceur

- Rappel 1 : un programme est un fichier exécutable écrit en langage machine.
- L'exécution d'un programme consiste en :
  - son chargement en RAM à une adresse donnée ;
  - la copie de cette adresse dans le registre IP du C.U. (Control Unit, unité de contrôle) du CPU
- Rappel 2 : architecture de von Neumann :



- Sans autre mécanisme un tel système devrait séquentiellement exécuter *entièrement* chaque programme et 2 programmes ne pourraient s'exécuter en même temps.
- Le mécanisme qui va permettre cette exécution concurrente est celui des *interruptions*.
- Une interruption est un signal envoyé au processeur et généré :
  - soit par un matériel (horloge, carte réseau,. . .) ;
  - soit par un programme (qui a la capacité de l'interrompre).
- Le *gestionnaire d'interruptions* est un programme qui, fonctionne schématiquement comme suit :
  - le processeur s'interrompt (à la fin de l'instruction courante), par exemple à cause d'une interruption d'horloge ;
  - le gestionnaire d'interruptions est chargé : il sauvegarde en mémoire le contexte du programme qui s'exécutait ;
  - il décide à quel programme il va rendre la main, par exemple un navigateur web ;
  - il charge son contexte en mémoire ;
  - il rend la main au CPU et au programme du navigateur.

- Une stratégie possible de choix du programme à exécuter est d'utiliser une *file d'attente* : évite de monopoliser le CPU par un seul programme.
- Vocabulaire à connaître :
  - **exécutable** : fichier binaire qui contient des instructions machines ;
  - **processus** : programme en cours d'exécution. Chaque processus a un numéro unique pour le S.E. : son PID (Process Identificator)
  - **thread** : suite d'instructions démarrée par un processus. Notion propre à la programmation concurrente. Un processus peut lancer plusieurs threads.
- Exemple : un navigateur peut lancer deux threads qui s'exécutent simultanément : un thread qui dessine la page web, un autre qui télécharge un fichier.
- Rq : on voit là qu'un navigateur moderne est un gros programme, ayant des fonctionnalités semblables à celles d'un S.E.

- Un processus peut être interrompu autrement que par les interruptions d'horloge : typiquement un processus qui attend des données (interactions avec un utilisateur via des périphériques comme le clavier/la souris) sera interrompu par le S.E. et mis *en attente*.
- Lorsqu'une interruption clavier parviendra au S.E. indiquant des données concernant l'exécution de ce processus, il sera mis dans l'état *prêt*.
- États possibles d'un processus :
  - **nouveau** : juste après son lancement ;
  - **prêt** : dans la file d'attente des processus pouvant être exécutés ;
  - **en exécution** : en train de s'exécuter ;
  - **en attente** : interrompu, en attente p.ex. d'une entrée ;
  - **terminé** : en train d'être terminé, desallocation des ressources qu'il utilisait.
- Les états nouveau et terminé sont éphémères. Normalement l'état varie entre prêt, en exécution et en attente.

- Sous UNIX/Linux, la commande `ps` (processus state) permet d'obtenir des renseignements sur les processus connus du S.E.
- La commande `kill` permet de *tuer* un processus (en lui envoyant un signal d'interruption). Pour un processus lancé en console, cette commande a le même effet que la séquence Control-C tapée au clavier. Le processus peut intercepter le signal envoyé et proposer d'éventuelles sauvegardes à l'utilisateur. L'option -9 de `kill` arrête immédiatement le processus, sans lui laisser cette possibilité.

Ex. 3

1. À l'aide de la commande `man ps` (qui affiche la page de manuel du système décrivant la commande `ps`), déterminer les options à utiliser pour :
   - afficher tous les processus (et pas seulement ceux de l'utilisateur qui a lancé la commande) ;
   - afficher le nom de l'utilisateur qui a lancé chaque processus ;
   - afficher aussi les processus qui n'ont pas été lancés depuis un terminal.
2. Lancer votre navigateur en console en arrière plan : `firefox &` Rechercher son pid avec la commande `top`. Noter l'état du processus, le % de CPU qu'il utilise. tuer ce processus avec la commande `kill`.

Programmation concurrente

- La gestion des processus par l'ordonnanceur est ce qui permet au S.E. d'être multitâches. Elle optimise aussi l'utilisation des ressources de la machine (exemples : mise en pause des processus qui sont en attente d'entrées, mise en veille du CPU (en abaissant sa fréquence d'horloge) pour économiser de l'énergie).
- Mais ce fonctionnement pose aussi le problème de l'utilisation par plusieurs processus de ressources partagées.
- Exemple :

```
1  import time
2  from os import getpid
3
4  pid = str(getpid())
5  with open("test.txt", "w") as fichier:
6      for i in range(1000):
7          fichier.write(pid + " : " + str(i) + "\n")
8          fichier.flush()
9          time.sleep(0.01)
```

- Ligne 4 on récupère le pid du processus correspondant au programme Python quand il sera exécuté (et on le transforme en chaîne de caractères).
- Ligne 5 un fichier nommé `test.txt` est ouvert en écriture (créé s'il n'existait pas, recréé vide sinon).
- Ligne 8, on force l'écriture de la ligne 7 à être effectivement faite dans le fichier en mémoire (sur le disque).
- La ligne 9 impose au processus de se mettre en pause pendant un certain temps (en s) passé en argument.

Ex. 4

1. Saisir ce code source. Ouvrir une console et lancer ce programme avec l'interpréteur `python3`. Observer le contenu du fichier `test.txt`.
2. À l'aide de la commande :
   `python3 GetPID.py & python3 GetPID.py & python3 GetPID.py &`
   créer trois processus associés à l'exécution du même programme écrivant dans le fichier `test.txt`. Observer et interpréter à nouveau son contenu. Les numéros de pid écrits sont-ils dans un ordre reproductible si on répète cette commande ? Pourquoi en est-il ainsi ?

- Un accès concurrent à une même ressource peut être problématique et conduire au phénomène d'*interblocage* lorsque celle-ci est en *accès exclusif*.
- L'exemple suivant illustre ce phénomène avec l'accès à la carte son et suppose que celle-ci est un périphérique en accès exclusif.
- Un programme `enregistrer_son` permet d'enregistrer, à l'aide d'un micro, un son qui est envoyé sur la sortie standard. Il doit donc accéder à la carte son, qu'il acquiert au moment de son exécution. Les données peuvent être récupérées à l'aide d'une commande système de redirection (>) dans un fichier dont on choisit le nom, par exemple `test.son` :

```
enregistrer_son > test.son
```

- Un autre programme, `jouer_son`, permet de jouer le son qu'il lit sur l'entrée standard. Ce programme acquiert aussi la carte son au moment de son exécution. Pour lui faire jouer les données lues dans un fichier, on peut utiliser la commande système de «tube» (|), qui redirige la sortie d'une commande sur l'entrée d'une autre. Ici on peut utiliser la commande `cat file` (copy at terminal) qui envoie sur la sortie standard le contenu du fichier dont le nom est `file`. On redirige la sortie de la commande `cat` sur l'entrée de `jouer_son` :

```
cat test.son | jouer_son
```

- On peut vérifier (voir plus loin) que ces deux commandes fonctionnent comme souhaité. Par contre que se passe-t-il si on souhaite directement jouer l'enregistement avec la commande :

```
enregister_son | jouer_son
```

- On observe qu 'il y a un *interblocage*. L'explication est que le programme `enregistrer_son` acquiert la carte son en accès exclusif. Il envoie ses données sur la sortie standard. Le programme `jouer_son` ne peut démarrer, car la carte son n'est pas accessible. De plus, `enregistrer_son`, qui envoie ses données sur la sortie standard, devient lui aussi bloqué car il remplit la zone mémoire du buffer de la sortie standard, sur laquelle il ne peut plus écrire.

Ex. 5

1. Télécharger l'archive compressée `Son.tar.gz` sur le site du cours. Extraire le contenu cette archive dans un dossier de votre choix. les fichiers `enregistrer_son` et `jouer_son` sont les commandes d'enregistrement et d'écoute d'un son qui vont être utilisés. Vérifier qu'ils sont exécutables à l'aide de la commande `ls`.

2. En utilisant un micro, vérifier que la commande `./enregistrer_son > test.son` crée un fichier `test.son` (de données de son brutes) de quelques secondes (interrompre le programme au bout de quelques secondes avec Ctrl-C). On pourra vérifier que ces données brutes sont en effet envoyées sur la sortie standard si on ne redirige pas la sortie standard de la commande `enregistrer_son`.
3. À l'aide d'audacity, ouvrir les fichier `test.son` : Fichier->Importer->Données brutes, puis modifier la fréquence d'échantillonnage à 8 kHz. Vérifier/observer la forme d'onde caractéristique du signal de parole.
4. Vérifier qu'on peut jouer le son du fichier `test.son` à l'aide de la commande `jouer_son`, en redirigeant le contenu du fichier `test.son` sur l'entrée de la commande `jouer_son` : `./jouer_son < test.son`

5. Tester maintenant la redirection de l'enregistrement réalisé par `enregistrer_son` sur l'entrée de `jouer_son` :

   ```
   ./enregistrer_son | jouer_son
   ```

   Qu'observez-vous ? Expliquer ce qui est observé. Quel est le nom de ce phénomène ?

- Il est possible d'illustrer les problèmes d'interblocage dans le cadre de la programmation *multithread*, à l'aide du module `threading` de la bilbiothèque standard Python.
- Comme on l'a dit, un *thread* est un «sous-processus», c'est à dire une suite d'instructions démarrées par le programme lui-même et s'exécutant de façon concurrente avec lui.
- On donne page suivante un exemple d'utilisation de threads :

```python
import threading
import time

def hello(n):
    for i in range(5):
        print("Je suis le thread ", n, " et ma valeur est ", i)
        time.sleep(0.001)
    print("------------Fin du Thread ", n)

for n in range(4):
    t = threading.Thread(target=hello, args=[n])
    t.start()
```

- À la ligne 11 le programme crée 4 threads (un à chaque passage dans la boucle) à l'aide de la méthode `Thread()` qui reçoit deux arguments : le nom de la fonction qui sera exécutée par ce thread et les arguments de cette fonction.
- Chaque thread lancé va exécuter la fonction `hello(n)` qui aura reçu en argument le numéro du thread lancé. Cette fonction se charge d'exécuter une boucle affichant à chaque passage le numéro du thread (`n`) en cours d'exécution et la valeur (de 0 à 4) du compteur `i` de boucle.
- La ligne 7 impose à chaque thread de se mettre en pause pendant un bref instant. Cette ligne est à adapter en fonction des capacités de la machine sur laquelle le programme est testé, de façon à faire apparaître l'exécution concurrente (entrelacée) des différents threads.
- Si on lance ce programme, on remarque que les affichages des threads se mélangent. Cela illustre le fait que l'ordonnanceur permet un fonctionnement «multithread», donnant l'illusion que tous les threads s'exécutent en même temps.

- De surcroît, on observe que l'ordre d'exécution des threads n'est pas à 100 % reproductible : bien que créés dans l'ordre de leur numéro (du thread 0 au thread 3), les threads peuvent ensuite être exécutés et se terminer dans un ordre différent. La raison est que l'ordonnanceur attribue à chaque thread un temps d'exécution selon un algorithme qui dépend de nombreux paramètres : l'ensemble des processus du système d'exploitation, la priorité attribuée à chacun d'eux, etc. La seule chose sûre est que chaque thread sera progressivement exécuté, de façon concurrente avec les autres threads et processus.

Ex. 6 Tester le programme précédent et vérifier que plusieurs exécutions ne produisent pas le même affichage. Il est possible que vous ayez à régler le temps de mise en pause pour bien observer le multithreading.

- Le programme ci-dessous illustre le partage d'une ressource par plusieurs threads.

```
1   import threading
2   COMPTEUR = 0
3
4   def incrc():
5       global COMPTEUR
6       for c in range(100000):
7           v = COMPTEUR
8           COMPTEUR = v + 1
9
10  th = []
11  for n in range(4):
12      t = threading.Thread(target = incrc, args=[])
13      t.start()
14      th.append(t)
15
16  for t in th:
17      t.join()
18  print("valeur finale", COMPTEUR)
```

- Comme dans le programme précédent, on y lance 4 threads (en plus de celui du programme principal).
- Chaque thread se charge d'incrémenter une variable globale `COMPTEUR` en lui ajoutant 1 lors de 100000 passages dans une boucle pour.
- À la ligne 17, pour chacun des thread `t`, la méthode `t.join()` se charge d'attendre que le thread soit terminé. Donc l'affichage de `COMPTEUR` fait ligne 18 est réalisé quand tous les threads sont terminés.

Ex. 7 Saisir le code source de ce programme et le tester. Quelle valeur de compteur observe-t'on ? Est-elle reproductible d'une exécution à l'autre ?

- L'explication de l'observation précédente est la suivante : on pourrait s'attendre, chaque thread réalisant 100000 itérations, que la variable `COMPTEUR` atteigne à la fin du programme la valeur 400000. Ce n'est pas ce qui est observé.
- En effet, si un thread est interrompu juste après avoir exécuté la ligne 7, la commutation de contexte mémorise sa valeur de `v`, qu'il considère comme la valeur de `COMPTEUR`. Ce thread reprendra la main quelques instants plus tard. À cet instant, bien que `COMPTEUR` aura été augmenté par les autres threads, ce thread, retrouvant son contexte avec la même valeur de `v`, affectera à `COMPTEUR` la valeur qu'il aurait affecté si il n'avait pas été interrompu, c'est à dire une valeur inférieure à la valeur que les autres threads auront entre temps donné à cette variable. Cela explique que la valeur affichée soit parfois inférieure à 400000.

- On voit qu'il n'aurait pas fallu que le thread soit interrompu au moment où il exécute les lignes 7 et 8, c'est à dire sur une section du code qu'on appelle une *section critique*.
- Une solution est de disposer d'un *verrou* que l'on acquiert au moment où il ne faut pas être interrompu *par un autre thread essayant lui aussi d'exécuter le code de cette même section*. De cette façon, deux threads ne peuvent accéder simultanément à l'incrémentation de `COMPTEUR`.
- Le programme ci-dessous montre comment utiliser un verrou dans ce but.

```python
import threading
verrou = threading.Lock()
COMPTEUR = 0

def incrc():
    global COMPTEUR
    for c in range(100000):
        verrou.acquire()
        v = COMPTEUR
        COMPTEUR = v + 1
        verrou.release()

th = []
for n in range(4):
    t = threading.Thread(target = incrc, args=[])
    t.start()
    th.append(t)

for t in th:
    t.join()
print("valeur finale", COMPTEUR)
```

- Il faut remarquer que la solution précédente est liée au fait que les threads manipulent le *même* verrou.
- Dans les programmes qui manipulent plusieurs verrous, le phénomène d'interblocage peut se produire, comme c'est illustré dans le programme page suivante.
- Le probléme vient de ce que les fonctions `f1` et `f2` cherchent à acquérir les verrous 1 et 2 dans un ordre différent. On peut alors se retrouver dans une situation où :
  - `f1` a acquis le verrou 1 et est interrompue pour que `f2` s'exécute.
  - `f2` acquiert le verrou 2 et ne peut acquérir le verrou 1, déjà pris, donc elle est bloquée.
  - Même si `f1` reprend son exécution elle est bloquée lorsqu'elle veut acquérir le verrou 2 pris par `f2`.

Ex. 8 Saisir ce code source et verifier le phénomène d'interblocage.

```python
import threading

verrou1 = threading.Lock()
verrou2 = threading.Lock()

def f1():
    verrou1.acquire()
    print("Section critique 1.1")
    verrou2.acquire()
    print("Section critique 1.2")
    verrou2.release()
    verrou1.release()

def f2():
    verrou2.acquire()
    print("Section critique 2.1")
    verrou1.acquire()
    print("Section critique 2.2")
    verrou1.release()
    verrou2.release()

th=[]
t1 = threading.Thread(target=f1, args=[])
t2 = threading.Thread(target=f2, args=[])
t1.start()
th.append(t1)
t2.start()
th.append(t2)

for t in th:
    t.join()
print("fin")
```