# LO03 Notes I

## Partie 1

Traduit par 22124765 UTSEUS,SHU

# Table des matières

# 1 操作系统介绍

操作系统是用户与物理机器之间的接口，它能够使计算机的功能得以使用。它提供了一个称为“虚拟机”的基础，在此基础上构建程序和实用工具。操作系统的目的是允许开发应用程序，而无需担心硬件操作和管理的细节。

例如，我们可以通过一个简单的系统调用“read”来读取文件，而无需关心它位于哪种存储介质上（硬盘、U 盘等）。

## 1.1 操作系统的主要功能

操作系统的主要功能包括：

1. 程序的加载与启动
2. 处理器、内存和外设的管理
3. 进程（正在运行的程序）和文件系统的管理
4. 错误检测

## 1.2 操作系统的的结构

操作系统采用分层结构：每一层都使用下层的功能。通常将其划分为 5 层：

1. 最底层是内核，它是硬件与软件之间的接口，负责管理 CPU、中断、进程间的通信以及同步。内核完全驻留在内存中。
   — 内核是操作系统的核心部分，负责管理系统资源和控制程序执行。
2. 第二层是内存管理器，负责在进程之间分配内存。
   — 内存管理器确保每个进程获得所需的内存资源，同时防止内存泄漏和非法访问。
3. 第三层是输入/输出管理模块，负责管理所有外设（键盘、显示器、磁盘、打印机等）。
   — 输入/输出管理模块处理所有与外设相关的操作，包括数据传输和设备控制。
4. 第四层是文件管理器，负责磁盘空间的管理、文件操作，同时确保数据的完整性和文件的安全性。
   — 文件管理器负责文件的创建、删除、读写和保护，确保数据的安全和完整。
5. 第五层是资源分配模块，负责合理使用资源：
   — 它创建新进程并为其分配优先级。
   — 它允许每个进程在合理的时间内获取所需的资源。
   — 它允许互斥访问非共享资源，并避免死锁的发生。

### 1.3 进程(processus)

进程是一个正在执行的程序。我们区分两种类型的进程:

1. 系统进程，它们代表系统完成某项任务（例如管理打印机）。
2. 用户进程。

进程执行它们的代码，并通过系统调用（如 open、read、write 等）向内核请求服务。一个进程可以创建一个或多个子进程，这些子进程又可以创建子进程，从而形成一个树状结构。

## 2 内存的结构与管理

内存是:
— 能够记录、保存和重现信息的设备。
— 我们根据以下标准对存储器进行分类：特性（容量、带宽等）和访问类型（顺序、直接等）。
— 字（Word）：在内存中可寻址的信息单位（字节的组合）。
— 地址（Address）：引用内存中一个元素（字）的数值。

### 2.1 内存的种类

1. 非易失性存储器：ROM（只读存储器），也称为固定存储器
— 其内容是固定的（或几乎固定）
— 内容被永久保存
2. 易失性存储器：RAM（随机访问存储器），也称为动态存储器
— 其内容是可修改的
— 在断电时会丢失信息
— “随机”在这里的意思是“无限制访问”（而不是随机的）

### 2.2 ROM

外部来看，只读存储器类似于一个没有输入线和写命令的存储块。该系统可以限制为 k 条地址线、n 条输出线和一个电路选择信号。

例子：编写 a 和 b 四个组合的八个函数的真值表是很容易的。这样我们得到 $8 \times 4 = 32$ 个可能的值。这些值可以存储在一个有 5 条地址线的 32 位 ROM 中，并且有一个输出。三个位用于识别函数 $f_i$，剩下的两个地址线对应于变量 a 和 b。

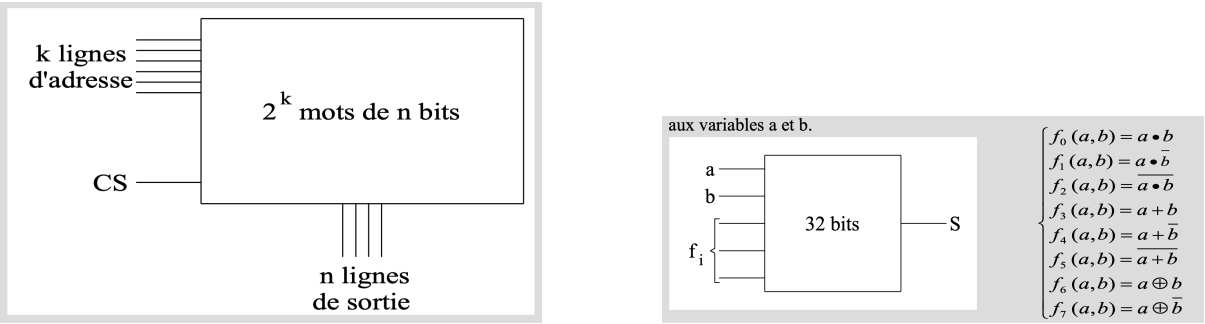



FIGURE 1 – ROM

## 2.3 RAM

DRAM：动态随机存取存储器 - 动态：需要周期性地刷新信息（电容器）

— 成本较低

SRAM：静态随机存取存储器

— 静态：不需要刷新 - 比 DRAM 快得多

— 但是成本高得多

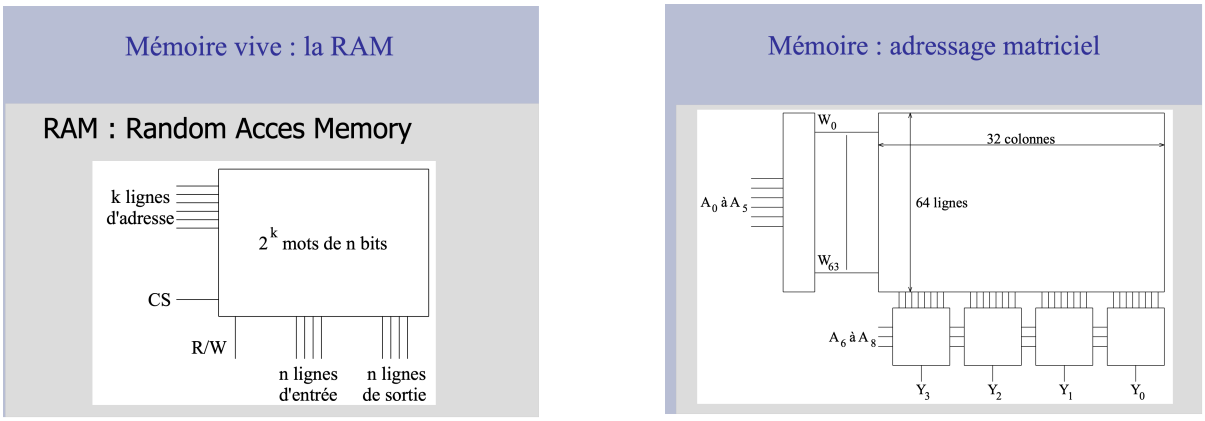



FIGURE 2 – RAM

**缓存存储器（SRAM,mémoire cache）**

— 处理器需要持续的数据吞吐量来读取指令和数据，以避免无所事事地等待。

— 问题：存储这些指令和数据的中央存储器太慢，无法保证这种吞吐量。

— 主意：在中央存储器和处理器之间使用一个非常快速的中间存储器（缓存存储器）。

部分地管理中央单元和中央存储器之间的速度差异。

— **Unité centrale（CPU）**：中央处理单元，它是计算机的主要处理部件。

— **Bus de données**：数据总线，用于在不同组件之间传输数据。

— **Bus d'adresse**：地址总线，用于指定数据传输的目标地址。

— **Unité de gestion mémoire（MMU）**：内存管理单元，负责管理虚拟地址和物理地址之间的映射。

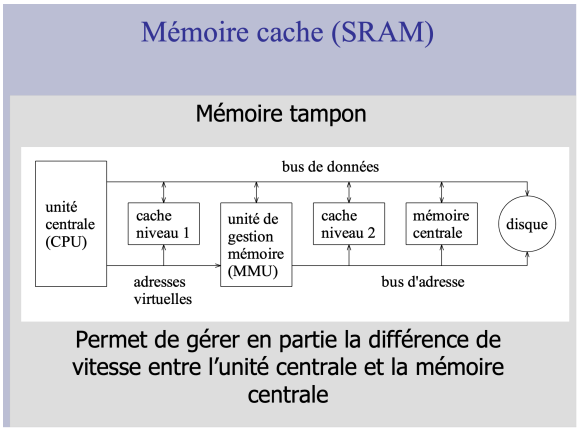



FIGURE 3 – 缓存存储器架构图

— **Mémoire tampon（Cache niveau 1 et niveau 2）**：缓冲存储器，分为一级缓存和二级缓存，用于存储 CPU 频繁访问的数据和指令，以减少访问主存储器的次数。
— **Mémoire centrale**：中央存储器，是计算机的主存储器，用于存储大量的数据和程序。
— **Disque**：磁盘，用于长期存储数据。

## 2.4 寄存器（Registre）

— 寄存器集成在 CPU 中
— 寄存器是一个存储与指令相关的信息的字：对于 64 位 Windows 机器是 64 位
— 在 CPU 中数量不多
— 非常快速（CPU 的速度）

# 3 虚拟内存

— 原理：我们将物理内存划分为大小固定的块或帧。
— 每个进程的内存区域（逻辑地址）被划分为与块大小相同的页面。
— 处理器拥有一个专用寄存器，其中包含一个对应表的物理地址。

每个进程的块表（对应表）指示哪些块在内存中。它包含进程虚拟空间中每个块的条目（内存中的位、块开始的物理地址等）。

— 块表的地址是进程上下文的一部分，需要在上下文切换时保存或恢复。
— 需要：
  — 一个快速转换虚拟地址到物理地址的机制。
  — 一个块替换策略。

虚拟内存和物理内存都为虚拟内存和物理内存分别结构化为页和帧。页的大小是固定的，等于一个帧的大小。它在 512 字节到 8KB 之间变化。例如，在一个物理内存为 32KB 的机器上运行一个 64KB 的程序（使用 4KB 的页面）且每条指令为一个字节。

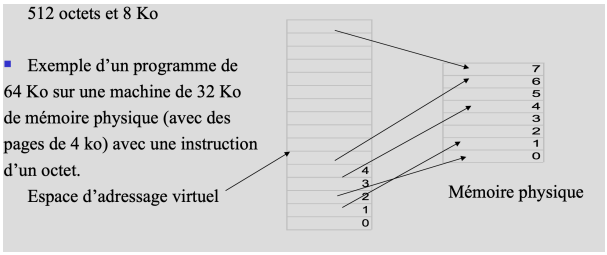



FIGURE 4 – Example

## 3.1 虚拟地址转换

执行过程中由指令引用的虚拟地址必须转换为物理地址。页面与帧之间的对应关系存储在称为**页表**的表中。该表的条目数等于虚拟页面数。在进程执行期间，进程的页表必须位于中央存储器中。

页表中的每个条目由多个字段组成，特别是：

— 内存中的存在位
— 引用位（R）
— 修改位（M）
— 对应页面的帧编号
— 在磁盘上的位置

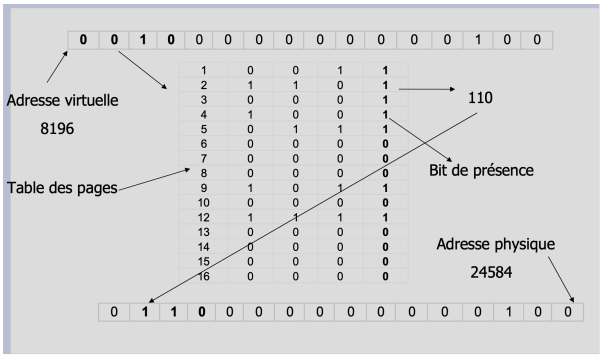



FIGURE 5 – 例：虚拟地址转换 -16bits

### 3.1.1 内存管理单元（MMU）

地址转换由处理器的一个硬件组件 MMU（内存管理单元）执行。

MMU 通过查询页表来验证请求的虚拟地址是否对应于物理内存地址。如果是这种情况，MMU 将在内存总线上传输实际地址，否则将发生页面错误。页面错误会引发一个中断，其作用是将缺失的页面从磁盘加载到内存中。

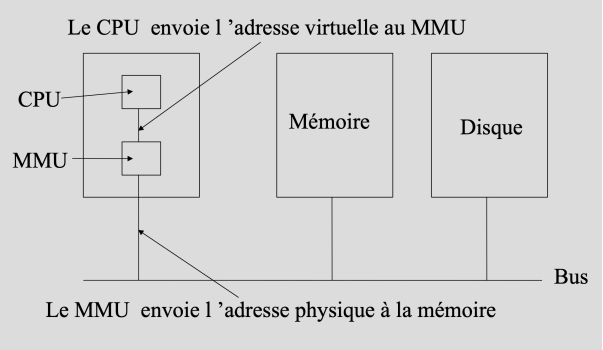



FIGURE 6 – 例：MMU

## 3.2 页面置换算法

在页面错误之后，操作系统必须从磁盘中将缺失的页面调入内存。如果内存中没有空闲的页框，则必须从内存中移除一个页面以替换为所需页面。如果要移除的页面自加载到内存以来已被修改，则必须将其写回磁盘。

页面置换算法记录对页面的访问引用。选择要移除的页面取决于过去的引用。存在不同的算法：

1. 贝拉迪最优算法（l'algorithme optimal de Belady）
2. 最近未使用算法（NRU）（La non récemment utilisé）
3. 先进先出算法（FIFO）（l'algorithme FIFO）
4. 时钟算法（l'algorithme de l'horloge）
5. 最近最少使用算法（LRU）（La moins récemment utilisé）

### 3.2.1 最优页面置换(Remplacement de pages optimal)

— 包括移除将来尽可能晚被引用的页面。
— 这种策略难以实施，因为很难预测程序未来的引用。
— 它被用作其他策略的参考，因为它最小化了页面错误数量。
— 访问次数：20，页面错误次数：9（45%）

### 3.2.2 先进先出(Remplacement de pages FIFO)

— 它使用 FIFO（先进先出）队列来存储内存中的页面。当发生缺页错误时，它会移除最早进入队列的页面，也就是队列头部的页面。

| 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **7** | 7 | 7 | **2** | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | **7** | 7 | 7 |
| | **0** | 0 | 0 | 0 | 0 | 0 | 4 | **4** | 4 | 4 | **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | | **1** | 1 | 1 | **3** | 3 | 3 | 3 | 3 | 3 | 3 | 3 | **1** | 1 | 1 | 1 | 1 | 1 | 1 |

TABLE 1 – 最优页面置换的页面引用序列和页面错误

— 这种策略并不是很优，因为它并没有真正基于页面的使用情况。
— 算法的异常现象：在某些情况下，增加帧数可能会导致缺页次数增加，而不是减少。


FIGURE 7 – Example——FIFO

### 3.2.3 最近最少使用(Remplacement de pages LRU)

— 移除最近最少使用的页面
— 该算法基于这样一个事实：如果一个页面刚刚被使用过，那么它在不久的将来很可能会被再次使用。


FIGURE 8 – Example-LRU

**如何实现 LRU 算法：**

— 需要为内存中的每一页记住最后一次引用的日期。
— 为每一页关联一个 n 位的寄存器来模拟页面的老化。
— 在每一步，如果页面存在，就将寄存器中最高位且不同于 1 的位设置为 1。
— 当需要移除一个页面时，选择寄存器值最小的页面。

**解释：** LRU 算法是一种常用的页面置换算法，用于决定在内存满时应该替换掉哪一页。算法的核心思想是：如果一个页面很长时间没有被访问过，那么在不久的将来它被访问的可能性也较小，因此可以被替换掉。

实现 LRU 算法的一种方法是使用一个称为“时钟”的寄存器。每个页面都有一个与之关联的时钟寄存器，该寄存器有 n 位。当页面被访问时，对应的寄存器中的所有位都会翻转（即 0 变 1，1 变 0）。然后，将寄存器中最高位的位（最老的位）设置为 0，以模拟页面的老化过程。

当需要替换页面时，算法会选择寄存器值最小的页面进行替换，因为这个值最小的页面表示它已经被最长时间没有被访问过了。

### 3.2.4 时钟算法或第二次机会算法(L'algorithme de l'horloge ou de la seconde chance)

— 该算法是 LRU 算法的一种近似。
— 页面被存储在一个环形列表中，形状像一个时钟。
— 一个指针指向最老的页面。
— 当页面错误发生时，从指针指向的页面开始逐个检查页面。
— 遇到的第一个引用位为 0 的页面将被替换。新添加页面的 R 位设置为 1。
— 如果检查的页面的 R 位不为 0，则将其设置为 0。

**解释：** 时钟算法是一种页面置换算法，它试图模拟 LRU 算法的行为，但实现起来更简单。算法使用一个环形列表来跟踪内存中的页面，每个页面都有一个引用位（R 位）。当页面被访问时，其 R 位被设置为 1。当需要替换页面时，算法从环形列表的某个起点开始检查每个页面的 R 位。如果找到一个 R 位为 0 的页面，它将被替换。如果 R 位为 1，则将其重置为 0，并继续检查下一个页面，直到找到 R 位为 0 的页面为止。


FIGURE 9 – Example- 时钟算法

### 3.2.5 Cas linux et windows

— 页面窃贼（执行页面置换算法的守护进程）定期唤醒，以查看内存中空闲页框的数量是否至少等于一个“最小”阈值。
— Windows 10 分配高达 17408 MB，即 17.4 GB 的虚拟内存。
— Windows 为每个进程分配至少 32 MB 的内存。Windows 的页面管理与 Linux 相当。

# 4 进程(processus)

进程可以相互同步和通信。

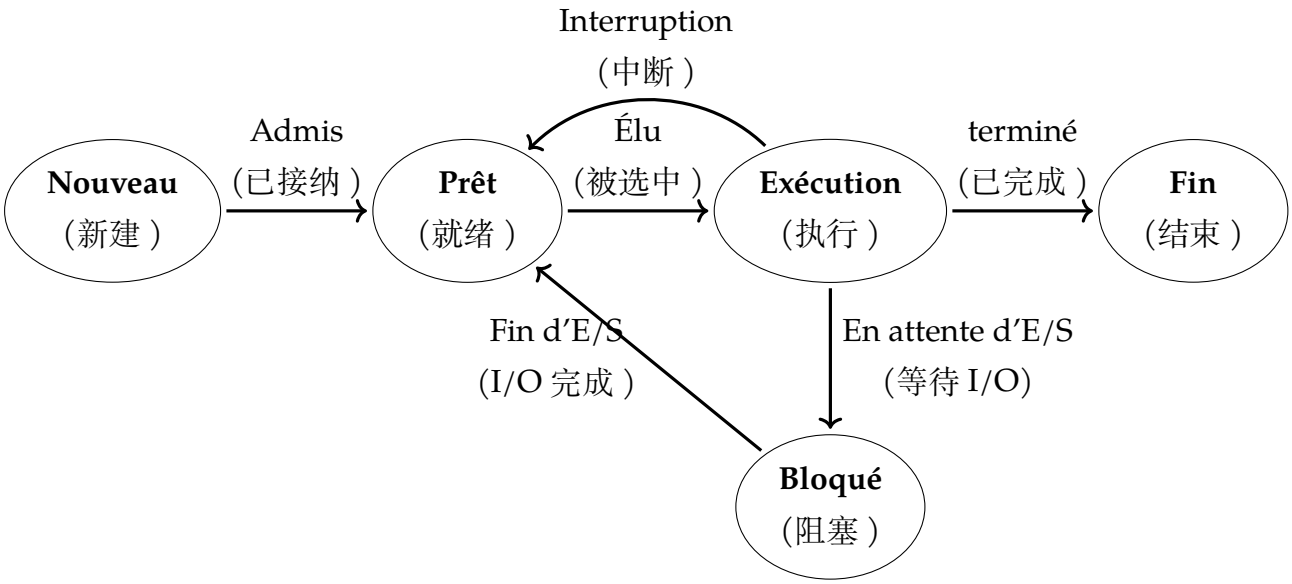
进程必须共享计算机的公共资源（中央处理单元、内存、外设、文件等)。

内核负责进程的正常运行，必须确保一个进程未经授权不会访问其内存或磁盘上的数据。

内核必须在进程之间分配执行时间，并根据所谓的“调度”原则决定下一个要运行在中央处理单元上的进程。

当一个进程执行时，它会改变状态。它可能处于以下三种状态之一：

1. **就绪（选中)**：正在执行中（在中央处理单元 UC 中执行）
2. **就绪**：等待处理器（它拥有所有资源，除了 UC）
3. **阻塞**：等待某个事件（缺少一个或多个资源以继续）




## 4.1 进程同步

— 在多进程操作系统中，多个进程以伪并行或并行方式执行，并共享对象（内存、打印机等)。

— 无特别预防措施的对象共享可能导致不可预测的结果。最终状态取决于进程的调度顺序。

— 当一个进程修改共享对象时，其他进程在该进程完成修改之前，既不应读取也不应修改它。

— 换句话说，对对象的访问必须通过互斥进行。

### 4.1.1 进程同步的问题示例

考虑银行的两个客户 A 和 B 共享同一个账户。账户余额为 1000 美元。假设在同一时间，两个客户各自发起一个操作。客户 A 要求提取 1000 美元，而客户 B 想要存入 100 美元。执行客户请求的两个进程共享“余额”变量。

```
1        如果（余额 >= 1000 美元）
2            余额 = 余额 - 1000 美元
3        否则出错
```

客户 A 的请求

```
1        余额 = 余额 + 100 美元;
```

客户 B 的请求

两个请求可能都被满足，余额可能变为 1100 美元。

### 4.1.2 关键资源与区段 :定义

必须防止对共享变量“余额”的同时使用。当一个进程执行指令序列时，另一个进程必须等待，直到第一个进程完成该序列。

- **关键资源(ressource critique**：一次只有一个进程可以访问它。
- **互斥(exclusion mutuelle)**：两个想要访问关键资源的进程处于互斥状态：要么是一个，要么是另一个，但不能同时访问。

关键区段：定义

- **关键区段(section critique)**：一组操作共享对象的指令，并且需要对共享对象的独占使用。
- 每个进程都有自己的关键区段。
- 在执行关键区段之前，进程必须确保对关键区段所操作对象的独占使用。

## 4.2 管理互斥：信号量（Semaphore）

信号量是一个整数计数器，表示可用访问权限的数量。每个信号量都有一个名称和一个初始值。信号量通过以下操作进行操作：

1. P(S)：如果信号量 S 的值大于 0，则减少其值。否则，调用该操作的进程将被挂起。
2. V(S)：增加信号量 S 的值，如果没有进程因 P(S) 操作而被阻塞。否则，将选择其中一个进程并使其变为就绪。

```
1        Semaphore S=1
2        Processus P0
```

```
        Tant que pas fini (当还未结束)
        {
        P(S)
        section_critique_P0
        V(S)
        }
        P(S):S=S-1 if S>0
        sinon attente

        Processus P1
        Tant que pas fini (当还未结束)
        {
        P(S)
        section_critique_P1
        V(S)
        }
        V(S):S=S+1
```

信号量可以保证互斥

### 4.2.1 Linux 中的信号量

信号量在 “semaphore.h” 库中实现。类型由关键字 `sem_t` 指定。

1. `Sem_Init`：用于初始化一个信号量。
2. `Sem_destroy`：用于销毁一个信号量。
3. `Sem_Wait`：等同于操作 P。
4. `Sem_post`：等同于操作 V。
5. `Sem_getvalue`：返回信号量的当前值。

## 4.3 哲学家问题

五位哲学家围坐在一张桌子旁。桌子上交替摆放着 5 个碗、5 双筷子和一个装米饭的盘子。每位哲学家花费时间吃饭和思考。为了吃米饭，一位哲学家需要两双筷子，这两双筷子分别位于他的碗的两侧。

所有哲学家共享有限的资源，即筷子和米饭。哲学家们在吃饭和思考之间交替进行，这意味着他们需要频繁地请求和释放资源。每个哲学家需要两双筷子才能吃饭。筷子是关键资源，任何时候都只能由一位哲学家使用。如果一个哲学家拿起了一双筷子，其他哲学家就不能使用它。如果所有哲学家都拿起了他们左边的筷子，等待右边的筷

子，那么所有哲学家都会无限期地等待，导致死锁。如果哲学家们的行为没有得到适当控制，某些哲学家可能会永远得不到筷子，导致饥饿。需要一种机制来确保哲学家们能够公平地访问资源，避免死锁和饥饿。

### 4.3.1 解决办法

— 筷子是共享对象。对一双筷子的访问和使用必须互斥进行。
— 可以通过增加一个包含 5 个信号量的数组（每双筷子一个信号量）来避免问题。信号量初始化为 1。
— 当哲学家 i 无法拿到一双筷子时，他会等待 P(S[i])。
— 当哲学家吃完饭时，他会检查他的邻居是否在等待。如果是，他会唤醒可以吃饭的邻居，通过调用操作 V。
— 如果所有哲学家同时拿起一双筷子，每个人都将无法拿起另一双筷子（资源争用情况）。
— 为了避免这种情况，哲学家不会只拿一双筷子。
— 为此，使用一个名为“盘子”的信号量来管理可以同时到达盘子的哲学家数量。

```
1         Penser ( ); /section non critique/
2         P(plat) /début section critique/
3         P(S[i])
4         prendre_baguette (i) ; P(S[(i+1)%5])
5         prendre_baguette ((i+1)%5) ; manger ( );
6         poser_baguette (i) ;
7         V(S[i])
8         poser_baguette ((i+1)%5) ;
9         V(S[(i+1)%5])
10        V(plat) / fin section critique /
```

Pseudo Code

## 4.4 图书馆问题

一家图书馆只有五本某本书。这些副本经常从读者那里借出，从作者那里借入，并从原件开始由图书管理员进行修改。这个问题模拟了对数据库的访问。一组进程不断请求访问数据库，要么为了写入，要么为了读取信息。为了确保数据库中数据的一致性，如果一个进程正在修改数据库（以写模式访问数据库），则必须禁止所有其他进程访问（无论是读取还是写入）。相反，多个进程可以同时访问数据库，以读取模式。读者代表请求以写模式访问数据库的进程。借阅者代表请求以读取模式访问数据库的进程。

```
1     Algorithme de l'écrivain :
2         Tant que vrai faire
```

```
            Modifier l'original
            Modifier les 5 exemplaires
        Fin Tant que

    Algorithme d'un lecteur :
        Tant que vrai faire
            Consulter un des 5 exemplaires
            Sortir;
        Fin Tant que
```

Pseudo Code

## 4.5 使用信号量同步进程

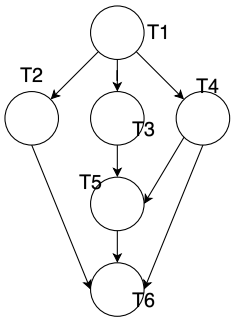
### 4.5.1 一种应用

信号量的另一个应用是同步那些合作以实现一个任务的进程，该任务需要按照一个明确定义的顺序进行。

— 进程 P0
  — ...
  — P(S) 然后呢（Puis-je)?
  — ...
— 进程 P1
  — ...
  — V(S) 去吧（Vas-y）
  — ...

### 4.5.2 另一种应用

实现一个优先级图，使用 3 个并行进程和信号量来帮助。



# 5 互锁(interlocage)

执行一个进程需要一组资源（内存空间、磁盘空间、文件、外设等），这些资源由操作系统分配给进程。如果某些进程持有资源并且请求其他已被分配的资源，可能会出现问题。

**示例：**

一个进程 P1 持有资源 R1，并等待另一个进程 P2 正在使用的资源 R2。

— 就出现了互锁情况（P1 等待 R2，P2 等待 R1）。这两个进程将无限期地等待。

— 进程 P2 持有资源 R2，并等待资源 R1。

**什么是死锁？**

— 如果每个进程都在等待另一个属于该集合的进程释放资源，则一组进程处于死锁状态。

— 由于所有进程都在等待，没有进程能够执行并释放其他进程所需的资源。它们将无限期地等待。

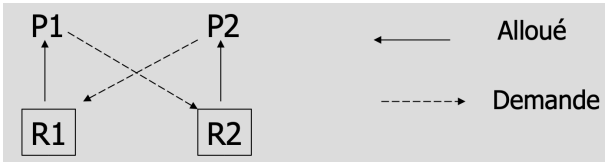



Figure 10 – Example

## 5.1 检测

### 5.1.1 资源分配图

为了检测死锁，系统动态构建了一个显示资源分配的系统资源分配图，该图表明资源的分配和需求。

— 在每次资源请求（由 CPU 耗时结束引起）后修改图时

— 定期或当进程使用量低于某个阈值时（检测可能较慢）

**资源分配图**是一个由两种类型的节点和一组弧组成的二分图：

— **进程**用**圆形**表示

— **资源**用**矩形**表示。每个矩形包含与资源实例数量相等的点

— 从**资源指向进程的有向弧**表示该资源已分配给进程

— 从**进程指向资源的有向弧**表示该进程因等待该资源而被阻塞

此图显示了每个进程所持有的资源以及其请求的资源。

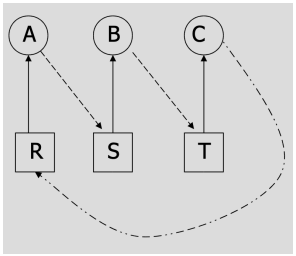
检测通过 reduisant le graphe 来实现检测。

**例：**有三个进程 A、B 和 C，它们使用三个资源 R、S 和 T，如下表所示：

| A | B | C |
|---|---|---|
| Demande R | Demand S | Demand T |
| Demande S | Demande T | Demande R |
| Libère R | Libère S | Libère T |
| Libère S | Libere T | Libère R |

如果进程按顺序 A、B、C 依次执行，则不会出现死锁。
假设进程的执行由一个循环调度器管理，将会出现死锁的情况。
**例：** 如果这些指令以以下顺序执行：

1. A Demande R
2. B Demande S
3. C Demande T
4. A Demande S
5. B Demande T
6. C Demande R




### 5.1.2 资源分配图的简化(Réduction du graphe d'allocation)

在简化资源分配图时，必须检查与每个进程和每个资源相关的箭头：
— 如果某个资源只有指向外部的箭头（没有请求），则删除这些箭头
— 如果某个进程只有指向它的箭头，则删除这些箭头
— 对于每个请求箭头，如果箭头指向的资源块中有可用资源，则应分配该资源

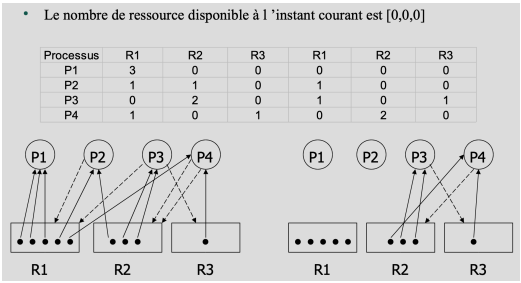



FIGURE 11 – Réduction du graphe d'allocation : Exemple

## 5.2 恢复(La reprise)

当系统检测到死锁时，它必须消除它，这通常通过执行以下操作之一来实现：

1. 暂时从一个进程中收回一个资源以分配给另一个进程。
2. 恢复到之前的状态（回滚）并避免再次陷入相同情况。
3. 终止一个或多个进程。

## 5.3 避免

当一个进程请求资源时，系统必须确定资源分配是否安全：
— 如果存在一种资源分配序列，允许所有进程正常结束，则分配资源。
— 如果不存在这样的序列，则不分配资源。

## 5.4 死锁检测：银行家算法

判断一个状态是安全还是不安全的：**银行家算法**

设有 n 个进程 P1, P2, ..., Pn 和 m 种类型的资源 R1, R2, ..., Rm 存在于系统中。检测死锁的算法使用以下矩阵和向量：

— 矩阵“Alloc”表示当前分配的资源，大小为(n x m)。元素 Alloc[i,j] 表示进程 Pi 持有的资源类型 Rj 的资源数量。
— 矩阵“Req”表示资源需求，大小为(n x m)。元素 Req[i,j] 表示进程 Pi 需要的资源类型 Rj 的资源数量以继续执行。
— 向量 A 表示可用资源，大小为 m。元素 A[j] 表示系统中未分配的资源类型 Rj 的资源数量。
— 向量 E 表示系统中存在的资源类型 Rj 的总资源数量。

1. 查找一个在“Req”中“Pi”范围内没有标记的进程“Pi”，如果存在则状态是安全的。
2. 如果不存在这样的进程，则状态是不安全的。算法终止。
3. 如果存在这样的进程，将“Alloc”中的“Pi”范围添加到“A”，标记该进程。
4. 如果所有进程都被标记，则状态是安全的，算法终止；否则，返回步骤 1。

# 6 进程调度(L'ordonnancement des processus)

## 6.1 介绍

计算机的任何软件都可以看作一组进程，这些进程的执行由一个特定的进程管理：调度器（法语 l'Ordonnanceur，英文中的 scheduler）。

**调度器**：
— 决定进程的执行顺序和时间。
— 管理处理器在不同进程之间的分配。
进程的执行是 CPU 计算和等待事件（如 I/O、信号量等）的交替序列。

调度器负责上下文切换（context switch）：例如，当当前进程阻塞或挂起时，处理器将被分配给另一个进程。

## 6.2 调度目标和方式

**调度器的目标**包括：

1. 最大化同时执行的进程数量。
2. 最小化每个进程的等待时间。
3. 最大化处理器的使用时间。
4. 避免饥饿问题（无限等待）。
5. 优先考虑优先级更高的进程。
6. 最小化上下文切换次数（进程切换）。

**调度性能标准**包括：

1. 进程的驻留时间：进程从到达至退出的时间。
2. 进程的等待时间：进程处于就绪状态的总时间。
3. 响应时间：从进程进入系统到开始被处理器处理的时间。
4. 处理能力：单位时间内处理器处理的进程数量。处理器使用率或进程使用率。

**调度策略选择进程执行的标准**包括：

1. 先到先服务：最早到达的进程首先被服务。
2. 最近最少使用（LRU）：最久未使用的进程优先。
3. 更高优先级：基于静态或动态优先级的进程。
4. 最短预计执行时间。

**进程分配给选定进程的时间**：
— 从分配到终止或自愿释放（非抢占式调度）。
— 分配时间可以是固定的或可变的（抢占式调度）。
**如何调用调度器**：

1. 当前一个进程阻塞或让出处理器时。
2. 当分配给进程的时间用完时。
3. 当更高优先级的进程到达时。

## 6.3 非抢占式调度算法（Ordonnancement non préemptif /sans réquisition）

**基本概念**：调度系统根据一组规则选择要执行的进程：

先到先服务（Premier arrivé, premier servi,FCFS，First-Come First-Served）：最先到达的进程首先被服务。

最短作业优先（Plus court d'abord,SPF，Shortest Process First）：优先执行预计执行时间最短的进程。

最高优先级优先(Plus prioritaire d'abord)（基于静态或动态优先级的进程）。(**PPT里面没具体提及**)

进程可以一直占用处理器直到它完成、阻塞（等待某个事件）或主动让出处理器。没有处理器的抢占。

### 6.3.1 FCFS

| Processus | Exécution (s) | Arrivée (s) |
|---|---|---|
| A | 3 | 0 |
| B | 6 | 1 |
| C | 4 | 4 |
| D | 2 | 6 |
| E | 1 | 7 |

TABLE 2 – Processus, Exécution, et Arrivée

执行顺序为：AAABBBBBBCCCCDDE

Temps de séjour moyen 平均驻留时间（平均每个进程在系统中停留时间）: 7.6 s

Temps moyen d'attente 平均等待时间(平均每个进程在就绪队列中等待时间）:4.4s

Nombre de changement de contexte 上下文切换次数 : 5

Unités de temps par processus 每个进程的时间单位 : 3.2 s

FCFS 算法是一种公平性较高的调度算法，它确保了先到达的进程能够先获得服务，从而减少了进程的等待时间。然而，这种算法可能导致短作业饥饿，特别是当长作业很多时，短作业可能需要等待较长时间才能执行。(Temps moyen d'attente élevé si de longs processus sont exécutés en premier)

### 6.3.2 SPF

AAABBBBBBEDDCCCC

Temps de séjour moyen : 6.4 s

Temps moyen d'attente : 3.2 s

| Processus | Exécution (s) | Arrivée (s) |
|---|---|---|
| A | 3 | 0 |
| B | 6 | 1 |
| C | 4 | 4 |
| D | 2 | 6 |
| E | 1 | 7 |

TABLE 3 – Tableau des processus

Nombre de changement de contexte : 5

最短平均等待时间(Meilleur temps moyen d'attente)

## 6.4 抢占式调度算法（Ordonnancement préemptif/avec réquisition）

为了确保没有进程运行时间过长，计算机有一个电子时钟，它会周期性地产生中断。

在每次时钟中断时，操作系统接管并决定：

— 当前进程应继续执行，还是应暂停以便让其他进程运行。

— 如果决定暂停进程以便其他进程运行，它必须首先保存当前进程的状态，然后将即将运行的进程的数据加载到寄存器中（上下文切换，commutation de contexte）。

处理器在执行每个进程时，仅在几十或几百毫秒后就会将处理器交给另一个进程。

处理器分配给进程的时间称为**量子(Quantum)**。这种进程间的切换应该很快，即明显低于量子时间。

处理器在某一时刻只执行一个进程。但在一秒钟内，处理器可以执行多个进程，并给人一种并行执行（伪并行 ,pseudo parallélisme）的印象。

量子值的选择至关重要：

— 量子值太小会导致进程间通信过于频繁。

— 量子值太大会增加交互模式下短命令的响应时间。

— 量子值通常是几毫秒，在 Unix 的早期版本中为 1 秒。

### 6.4.1 SRT(Shortest Remaining Time)

— 当一个进程到达时，调度器将其预计执行时间与当前正在执行的进程的执行时间进行比较（SRT 算法的预占式版本）。

— 如果这个时间更短，则立即执行新进程。

— 在执行时间相同的情况下，选择最近被执行过的进程。

执行顺序：AAABCCCCEDDBBBBB

| Processus | Exécution (s) | Arrivée (s) |
|---|---|---|
| A | 3 | 0 |
| B | 6 | 1 |
| C | 4 | 4 |
| D | 2 | 6 |
| E | 1 | 7 |

Table 4 – Tableau des processus

平均驻留时间：5.8 秒
平均等待时间：2.6 秒
上下文切换次数：6 次

## 6.5 循环调度（轮转法）

## Ordonnancement réaliste : Ordonnancement circulaire (Tourniquet)

— 这是一个古老、简单且公平的算法。
— 它将等待执行的进程列表存储在一个先进先出（FIFO）队列中。

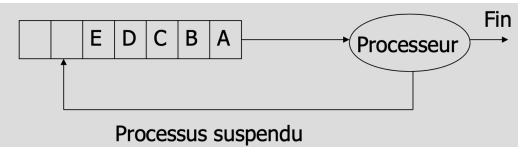



Figure 12

### 6.5.1 例子

| Processus | Exécution (s) | Arrivée (s) |
|---|---|---|
| A | 3 | 0 |
| B | 6 | 1 |
| C | 4 | 4 |
| D | 2 | 6 |
| E | 1 | 7 |

Table 5 – Tableau des processus

ABABACBDCEBDCBCB

Temps de séjour moyen : 8.0 s
Temps moyen d ′attente : 4.8 s
Nombre de changement de contexte : 16
Quantum = 1 s et temps de commutation = 0

## 6.6 带多个队列的优先级调度

调度器为每个进程分配一个优先级（静态或动态）:
— 相同优先级的进程存储在一个先进先出（FIFO）队列中。
— 存在多个队列，数量与优先级数量相同。
— 每个队列中的进程按照轮转（Round Robin）原则进行调度。

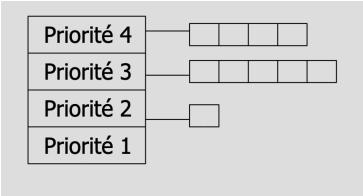



FIGURE 13

**练习**

Un système utilise une file d'exécution.La file est gérée par tourniquet avec une valeur du quantum égale à 1. La priorité initiale de chaque processus (noté de T1 à T5) est donnée dans le tableau ci-après.

Le système est également doté d'une pagination suivant l'algorithme FIFO. Au cours de son exécution une tâche accède à une des pages numérotées de 1 à 5 (cf tableau). Le système alloue en permanence un espace fixe de trois cases en mémoire. A chaque fois qu'un processus est lancé sa priorité est incrémentée du nombre de défaut de pages qu'il a effectué.

| 进程 | ti (量子) | Ti (量子) | 初始优先级 | 页面 |
|---|---|---|---|---|
| T1 | 3 | 0 | 5 | 1,2 |
| T2 | 2 | $\epsilon$ | 6 | 2,3 |
| T3 | 1 | 1 | 7 | 3,4 |
| T4 | 3 | $1+2\epsilon$ | 8 | 4,5 |
| T5 | 2 | $1+\epsilon$ | 9 | 5,6 |

TABLE 6 – 进程调度参数表

## 6.7 公平性问题

公平性问题：

— 低优先级进程的执行可能会被更高优先级的进程到达而持续推迟。

— 可以通过以下方式避免优先级反转：

  — 低优先级的进程占用了高优先级进程执行所需的资源。

  — 低优先级的进程不能执行，因为有持续运行的高优先级进程。

  — 如果一个进程等待资源时间过长，它的优先级会暂时提高。这可以避免饥饿和优先级反转的问题。

## 6.8 Linux 案例

调度器使用多个队列，每个队列都与一个优先级相关联：

— 准备就绪的进程根据其优先级被分配到队列中。

— 以用户模式运行的进程的优先级为正数或零，而以超级用户模式（或管理员模式）运行的进程的优先级为负数（数值越大）。

— 调度器使用轮转（Round Robin）原则。

— 选定的进程最多执行一个量子（大约 100 毫秒）。如果它在量子结束前没有完成或没有阻塞，它将被挂起。

— 挂起的进程被插入到其队列中。

— `nice` 命令允许给进程分配一个更低的优先级（大于 0）。

— `renice` 命令允许在执行过程中更改进程的优先级。

— 进程的优先级每秒钟定期重新计算，考虑到 CPU 使用时间和页面错误数。