# 重点

| SPS | MPS (Thread) | | Memory | Disk | Disk Cache |
|---|---|---|---|---|---|
| FIFO | Load sharing | FCFS | FIFO | FIFO | |
| SPN | | SNTF | CLOCK | SSTF | |
| SRT | | Preemptive SNTF | LRU | SCAN | LRU |
| HRRN | Gang scheduling | | Optimal | C-SCAN | LFU |
| RR | | | | | |
| FQ | | | | | |
| | | | | | |

## Ep.1 OS Introduction - OS:从入土到入土

知识点:

- ⭐**Objectives** - 目标:
  - **Convenience** - 便利
    - **Program** (Execution APP) - 程序
    - **Interface** (Application ⇆ Hardware) - 接口
    - Resource managemer - 资源管理器
  - **Efficiency** - 效率
    - **Software** (Same as ordinary software) - 软件
    - **Relinquish** & **Resumes** control for CPU - 主动放弃、重新控制CPU
  - **Ability to evolve** - 能升级
- Evolution - 演变
  - Serial Processing - 串行
  - Simple Batch Systems - 批处理
  - Uniprogramming - 单道
  - Multiprogramming - 多道
  - Time-Sharing System - 分时
  - Modern OS - 现代OS
- ⭐**Services** - 服务
  - Program execution
  - IO opeartuibs
  - File systems
  - Communication

名词：

- **System Calls - 系统调用**：系统低层的接口，介于用户模式和内核模式之间的接口。
  **OS provides** programming **interface** to services.
  Program requests a **service from an OS's kernel**.
  May include hardware-related services, creation and execution of new processes, and communication with integral kernel services such as process scheduling.
- Application Programming Interface(API)：感觉就是系统调用的封装orz……
  **High-level** - usually used by programs.
  Rather than direct use of system calls.
  Usually **less detailed** then raw calls.

辨析：

| Kernel Space | User Space |
| --- | --- |
| 内核空间 | 用户空间 |

| Monolithic Kernel | Microkernel |
| --- | --- |
| 操作系统应提供的多数功能都由这个大内核提供。 | 只有一些最基本的功能，<br>其他服务则由运行在用户模式且与其他应用程序类似的进程提供。 |

# Ep.2 Process Management - 进程管理

## 1. Process - 进程

见Process。

## 2. Thread - 线程

## 3. Mutual Exclusion & Synchronization - 互斥和同步

- 运行结果取决于：
  - Activities of other processes - 其他进程的活动
  - The way the OS handles interrupts - OS处理中断的方式
  - Scheduling policies of the OS - OS的调度策略
- 并发的困难：
  - Sharing of global resources.
    全局资源的共享。
  - Difficult for the OS to manage the allocation of resources optimally.
    OS难以对资源进行最优化分配。
  - Difficult to locate programming errors as results are not deterministic and reproducible.
    定位程序设计错误很困难。
- OS关注的问题（必须保证）：

    - Be able to keep track of various processes.
      要能追踪不同的进程。（利用PCB可以实现）
    - Allocate and de-allocate resources for each active process.
      能为每个活动进程分配和释放各种资源。
    - Protect the data and physical resources of each process against interference by other processes.
      保护每个进程的数据和物理资源，避免被其他进程无意干扰。
    - Ensure that the processes and outputs are independent of the processing speed.
      进程的功能、输出结果要与执行速度（指令执行顺序）无关，即“可重现性”。
- **Critical Section - 临界区**
  一段需要被保护的代码，**一个时间内只能有一个进程在**该临界资源的临界区。
- **Semaphore** - 信号量
  A variable that has an integer value upon which only three operations are defined.
- **Monitors** - 管程
  利用**Condition variables**，实现**Synchronization**(同步)。
- **Message Passing** - 信号传递
    - **Synchronization** - 同步
      使得能强制互斥
    - **Communication** - 通信
      使得能交换信息
- 生产消费者:

```
// producer
void producer()
{
  while (true)
  {
    produce();     // 生产商品
    semWait(e);    // 检查是否有空间可装载商品
    semWait(lock); // 检查是否被互斥保护
    /* 临界区开始 */
      append(); // 添加商品到缓冲区
    /* 临界区结束 */
    semSignal(lock); // 解除互斥锁
    semSignal(n);    // 可购买商品+1
  }
}

// consumer
void consumer()
{
  while (true)
  {
    semWait(n);    // 检查是否有剩余商品可购买
    semWait(lock); // 检查是否被互斥保护
    /* 临界区开始 */
      buy(); // 购买商品
    /* 临界区结束 */
    semSignal(lock); // 解除互斥锁
    semSignal(e);    // 可装载商品的空间+1
  }
}
```

- 读者写者（读者优先）：

```
int readCount; // 全局变量 - 读者数目
semaphore x = 1,    // x - 修改readCount的互斥锁
          wsem = 1; // wsem - 写者锁
// Reader
void reader()
{
  while (true)
  {
    // 通知一名读者爷爷来了
    semWait(x); // 对修改readCound操作加互斥锁
      readCount++;
      if (readCount == 1) semWait(wsem); // 存在读者，加上写者锁（相当于第一个进自习室的把灯打开，写者就知道里面有人
    semSignal(x);
    // 开读（非临界区）
    READUNIT();
    // 通知一名读者爷爷走了
    semWait(x); // 读完需要修改readCount,故又要加锁
      readCound--;
      if (readCount == 0) semSignal(wsem); // 没有读者，释放写者锁（相当于最后一个走的关灯）
    segSignal(x);
  }
}
// Writer
void writer()
{
  while (true)
  {
    semWait(wsem);
      /* 写者临界区 开始 */
      WRITEUNIT();
      /* 写者临界区 结束 */
    semSignal(wsem);
  }
}
```

# 4. Deadlock & Starvation - 死锁和饥饿

## (1) Conditions for Deadlock - 死锁发生条件

- **Mutual Exclusion** - 互斥
  Only one process may use a resource at a time.
  一次只有一个进程可以使用一个资源。
- **Hold-and-Wait** - 持有等待
  A process may hold allocated resources while awaiting assignment of other resource.
  当一个进程等待其他进程的资源时，继续占有自己已分配的资源。
- **No Preemption** - 非抢占
  No resource can be forcibly removed from a process holding it.
  不能强行抢占进程已有的资源。
- **Circular Wait** - 循环等待
  A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain.
  存在一个闭合的进程链，每个进程至少占有此链中下一个进程所需的一个资源。（如上图(c)所示）

## (2) 死锁解决三种方式

表 6.1　操作系统中死锁检测、预防和避免方法小结[ISLO80]

| 原　则 | 资源分配策略 | 不同的方案 | 主要优点 | 主要缺点 |
| --- | --- | --- | --- | --- |
| 预防 | 保守；预提交资源 | 一次性请求所有资源 | • 对执行一连串活动的进程非常有效<br>• 不需要抢占 | • 低效<br>• 延迟进程的初始化<br>• 必须知道将来的资源请求 |
| | | 抢占 | • 用于状态易于保存和恢复的资源时非常方便 | • 过于经常地、没必要地抢占 |
| | | 资源排序 | • 通过编译时检测是可以实施的<br>• 既然问题已在系统设计时解决，因此不需要在运行时计算 | • 禁止增加的资源请求 |
| 避免 | 介于检测和预防中间 | 操作以便发现至少一条安全路径 | • 不需要抢占 | • 必须知道将来的资源请求<br>• 进程不能被长时间阻塞 |
| 检测 | 非常自由；只要有可能，请求的资源都允许 | 周期性地调用以便测试死锁 | • 不会延迟进程的初始化<br>• 易于在线处理 | • 固有的抢占丢失 |

- Prevent Deadlock - 防止死锁
  - Requesting all resources at once - 一次性请求所有资源
  - Preemption - 抢占（别人拥有的资源）
  - Resource ordering - 安排资源分配方式
- Avoid Deadlock - 避免死锁
- Detect Deadlock - 检测死锁

## (3) Banker's Algorithm - 银行家算法

- 状态
  - Safe State - 安全状态
  - Unsafe State - 危险状态
- 进程资源相关表
  - Claim matrix $C$ - 需求表
  - Allocation matrix $A$ - 已分配表
  - $C - A$ - 还需要分配资源表
- 资源相关表
  - Resources vector $R$ - 总共资源数
  - Available vector $V$ - 剩余资源数

### (4) Dining Philosophers Problem - 哲学家就餐问题

```
semphore fork[5] = {1},
         room = 4;
void philosopher(int i)
{
  while (true)
  {
    think(); // 哲学家先要沉思
    wait(room); // 申请进入房间
      wait(fork[i]);       // 尝试拿起左边的叉子
      wait(fork[(i+1)%5]); // 尝试拿起右边的叉子
        eat(); // 嗨吃狂吃
      signal(fork[(i+1)%5]); // 放下右边叉子
      signal(fork[i]);       // 放下左边叉子
    signal(room); // 退出这个房间!
  }
}
```

# Ep.3 Memory

## 1. Logical Organization - 逻辑组织

- Memory is organized as **linear**.
  在逻辑组织中，存储空间是被**线性组织**的。
- Programs are written in **modules**.
  程序是由一个个“**模块**”编写的。
  - Modules can be written and compiled independently.
    模块能被单独编写、编译。
    但不能单独执行，必须程序的所有模块一起执行。
  - **Different degrees of protection** given to modules (read-only, execute-only, ...).
    不同的模块能分配**不同的保护权限**。
  - Shard modules.
    模块可以被多个进程共享。
- Segmentation is the tool that most readily satisfies requirements.
  “**分段技术**”是一个容易满足需求的工具，将在之后介绍该内存管理技术。

## 2. Physical Organization - 物理组织

- Moving information between the two levels of memory should be a **system responsibility**.
  在两级存储器间**移动信息的任务**应该**交给系统负责**，
  It cannot leave the programmer with the responsibility to manage memory.
  程序员不负责内存管理。
  - **Memory available** for a **program plus its data** may be **insufficient**.
    对于程序代码和数据来说，**可用的存储空间可能不够。**（因此产生了覆盖技术(Overlay)）
  - **Overlaying** allows various modules to be **assigned the same region of memory** but is time consuming to program.
    **覆盖**允许多个模块分配在同一个内存区域上，但整个程序仍能运行。

- Programmer does not know **how much space will be available**.
  程序员**并不知道**实际还**剩多少可用空间**（只知道逻辑地址空间）。

# 3. Memory Management Techniques - 内存管理技术

| Technique | Description | Strengths | Weaknesses |
|---|---|---|---|
| **Fixed Partitioning** | Main memory is divided into a number of static partitions at system generation time. A process may be loaded into a partition of equal or greater size. | Simple to implement; little operating system overhead. | Inefficient use of memory due to internal fragmentation; maximum number of active processes is fixed. |
| **Dynamic Partitioning** | Partitions are created dynamically, so that each process is loaded into a partition of exactly the same size as that process. | No internal fragmentation; more efficient use of main memory. | Inefficient use of processor due to the need for compaction to counter external fragmentation. |
| **Simple Paging** | Main memory is divided into a number of equal-size frames. Each process is divided into a number of equal-size pages of the same length as frames. A process is loaded by loading all of its pages into available, not necessarily contiguous, frames. | No external fragmentation. | A small amount of internal fragmentation. |
| **Simple Segmentation** | Each process is divided into a number of segments. A process is loaded by loading all of its segments into dynamic partitions that need not be contiguous. | No internal fragmentation; improved memory utilization and reduced overhead compared to dynamic partitioning. | External fragmentation. |
| **Virtual Memory Paging** | As with simple paging, except that it is not necessary to load all of the pages of a process. Nonresident pages that are needed are brought in later automatically. | No external fragmentation; higher degree of multiprogramming; large virtual address space. | Overhead of complex memory management. |
| **Virtual Memory Segmentation** | As with simple segmentation, except that it is not necessary to load all of the segments of a process. Nonresident segments that are needed are brought in later automatically. | No internal fragmentation, higher degree of multiprogramming; large virtual address space; protection and sharing support. | Overhead of complex memory management. |

表 7.2　内存管理技术

| 技　术 | 说　明 | 优　势 | 弱　点 |
| --- | --- | --- | --- |
| 固定分区 | 在系统生成阶段，内存被划分成许多静态分区。进程可装入大于等于自身大小的分区中 | 实现简单，只需要极少的操作系统开销 | 由于有内部碎片，对内存的使用不充分；活动进程的最大数量是固定的 |
| 动态分区 | 分区是动态创建的，因而每个进程可装入与自身大小正好相等的分区中 | 没有内部碎片；可以更充分地使用内存 | 由于需要压缩外部碎片，处理器利用率低 |
| 简单分页 | 内存被划分成许多大小相等的页框；每个进程被划分成许多大小与页框相等的页；要装入一个进程，需要把进程包含的所有页都装入内存内不一定连续的某些页框中 | 没有外部碎片 | 有少量的内部碎片 |
| 简单分段 | 每个进程被划分成许多段；要装入一个进程，需要把进程包含的所有段都装入内存内不一定连续的某些动态分区中 | 没有内部碎片；相对于动态分区，提高了内存利用率，减少了开销 | 存在外部碎片 |
| 虚存分页 | 除了不需要装入一个进程的所有页外，与简单分页一样；非驻留页在以后需要时自动调入内存 | 没有外部碎片；支持更多道数的多道程序设计；巨大的虚拟地址空间 | 复杂的内存管理开销 |
| 虚存分段 | 除了不需要装入一个进程的所有段外，与简单分段一样；非驻留段在以后需要时自动调入内存 | 没有内部碎片；支持更多道数的多道程序设计；巨大的虚拟地址空间；支持保护和共享 | 复杂的内存管理开销 |

# 4. Paging - 分页

- **Partition memory** into equal fixed-size chunks that are relatively small.
  内存划分为相对较小的等大小固定块。
- **Process** is also divided into small fixed-size chunks of the same size.
  进程也被划分。
- Pages - 页：进程的块
- Frames - 页框（帧）：可用的内存块
- 地址转换：**页表内页号$n$ + 页框内偏移量$m$**

# 5. Virtual Memory - 虚拟内存

虚拟内存管理格式（地址）：

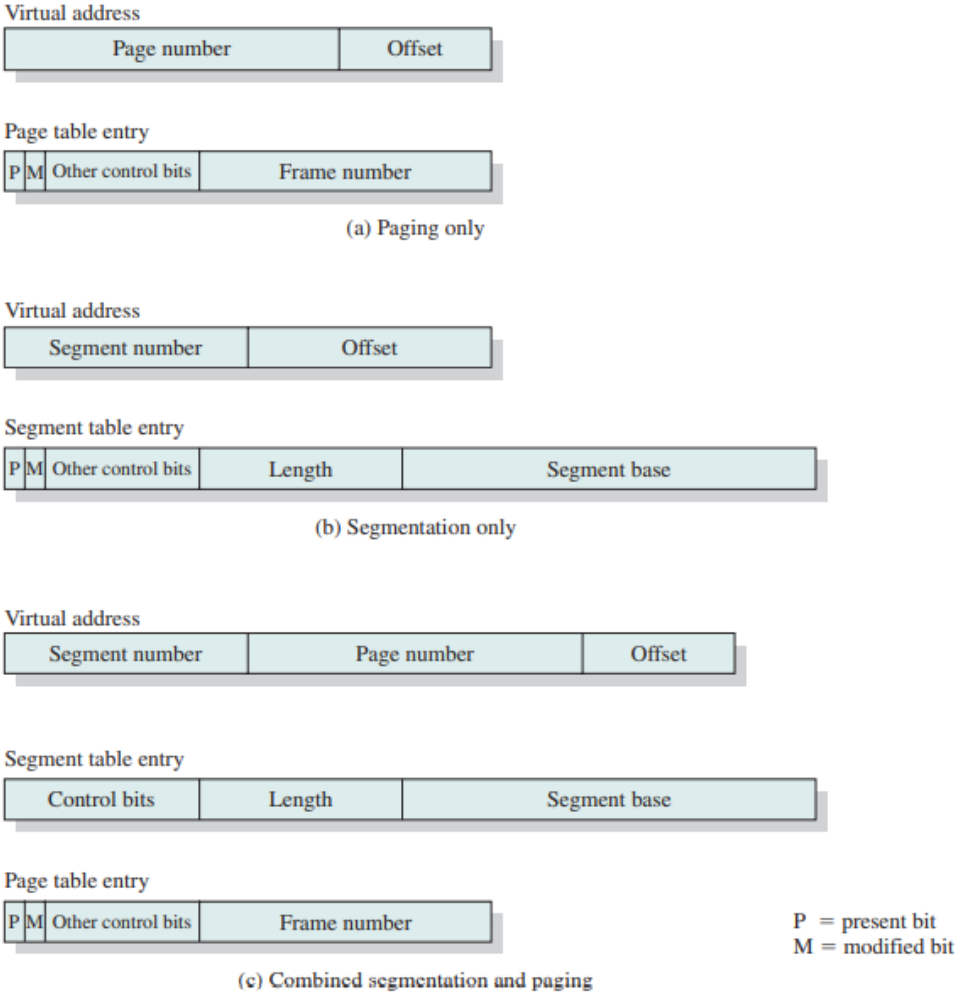


**Figure 8.1 Typical Memory Management Formats**

- Hardware and Control Structures - 硬件
  - Two-Level Hierarchical Page Table - 两级页表

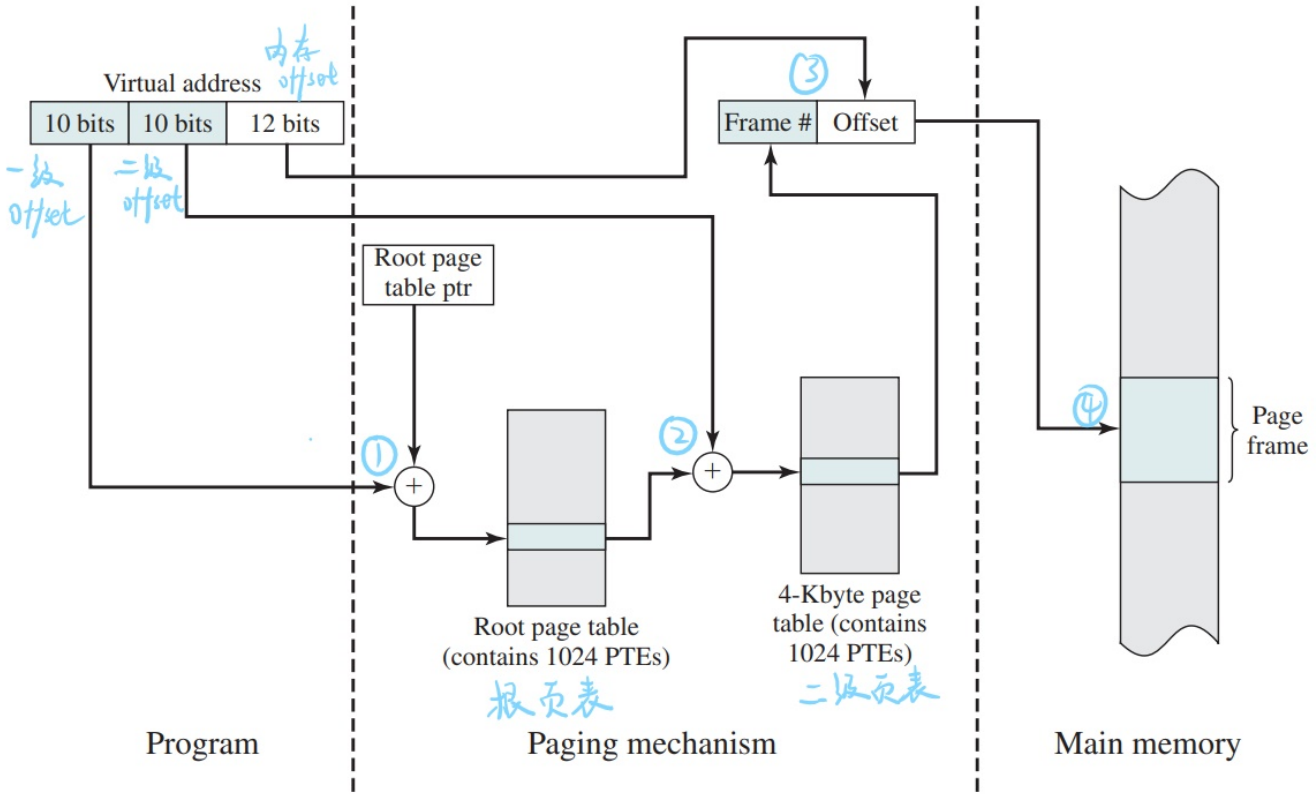


**Figure 8.4 Address Translation in a Two-Level Paging System**

- Inverted Page Table - 倒排页表

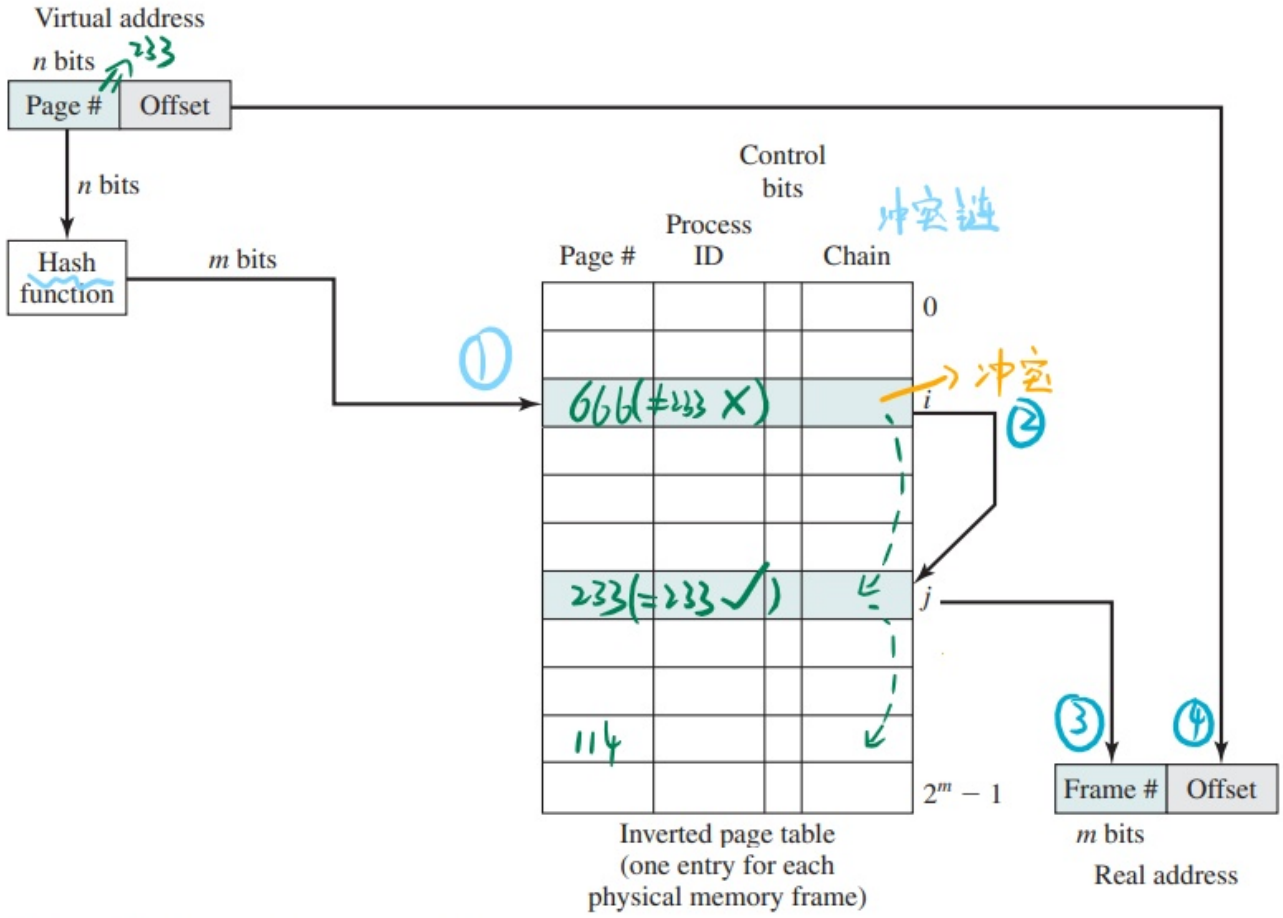


**Figure 8.5 Inverted Page Table Structure**

- Translation Lookaside Buffer(TLB) - 转换检测缓冲区（快表）

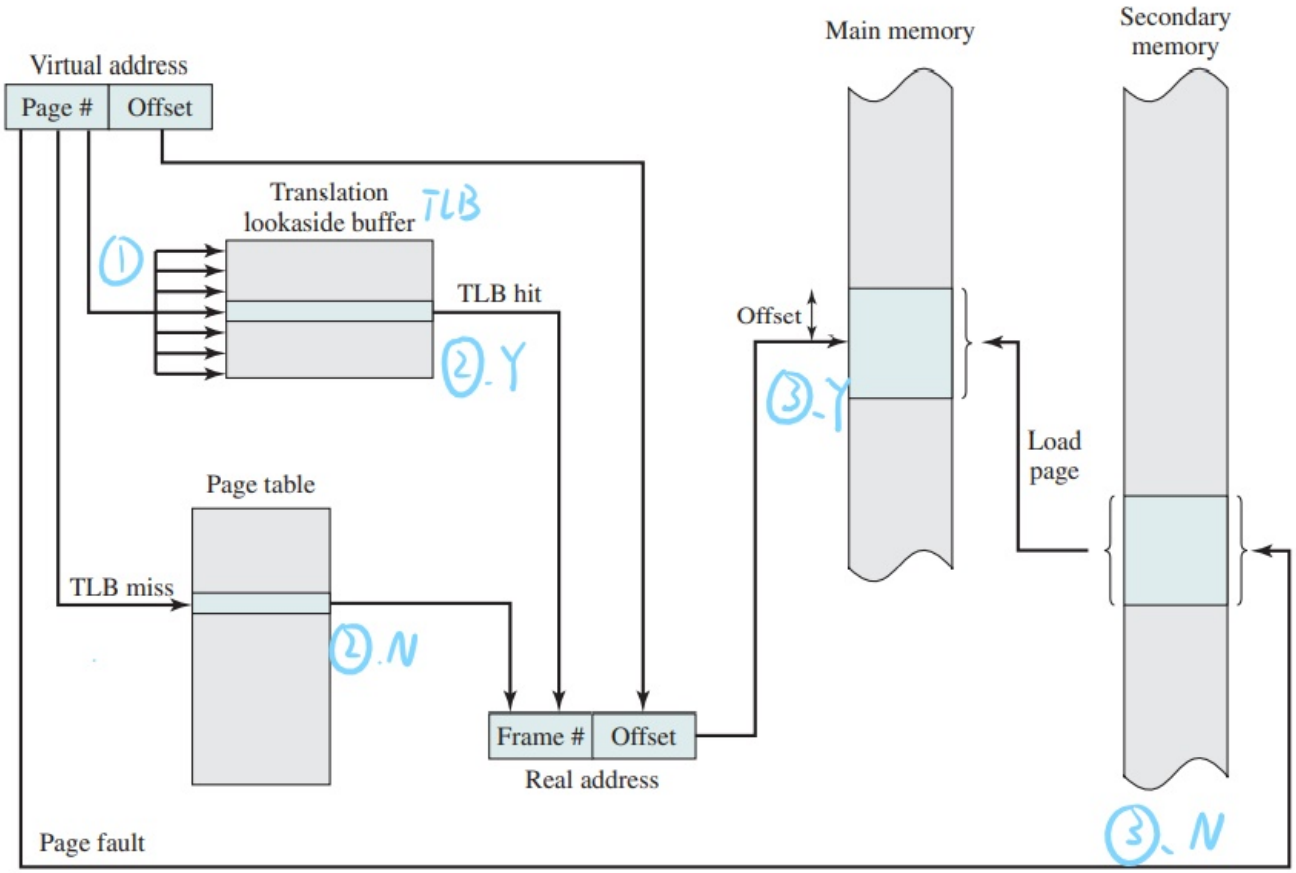


**Figure 8.6** Use of a Translation Lookaside Buffer

- Operating System Software - 软件

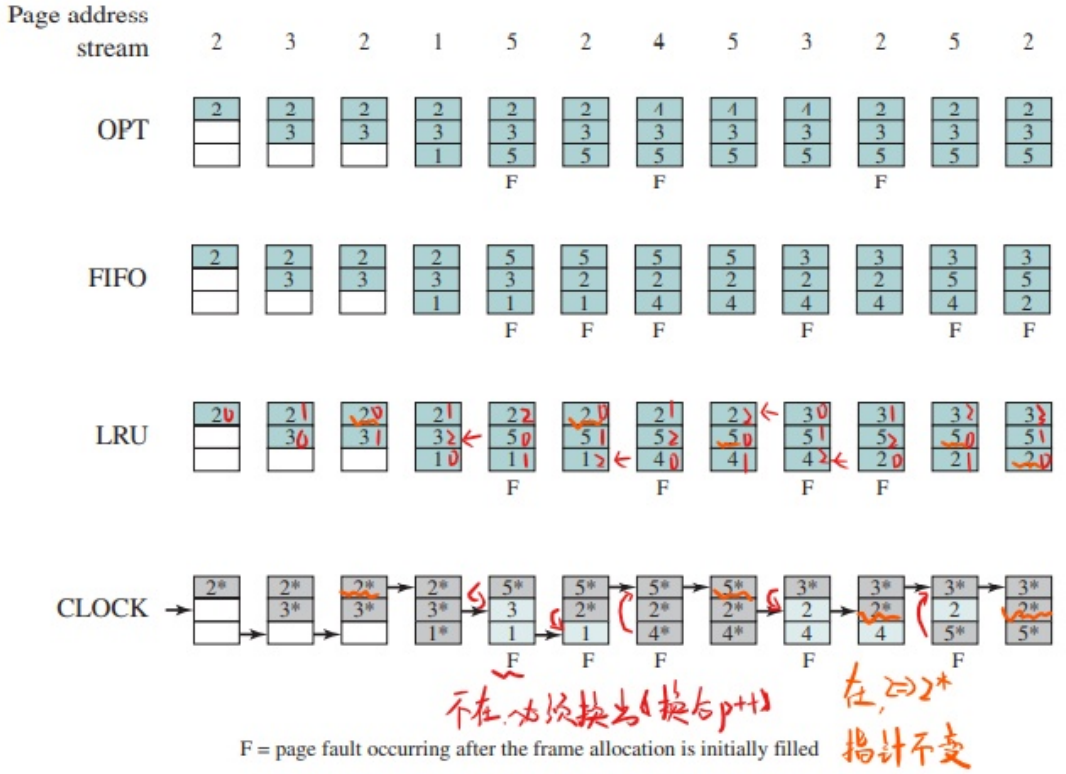


**Figure 8.14** Behavior of Four Page Replacement Algorithms

- Optimal Policy - 最优算法(理想模型)
- First-in-first-out (FIFO) - 先进先出算法
- Least Recently Used (LRU) - 最近最少使用

    - Clock Policy - 时钟策略
- Working Set Strategy - 工作集策略
  管理进程的驻留集（页的容量）

# Ep.4 Scheduling - 调度

## 2. Multiprocessor Scheduling

- Processor Shceduling - 进程调度：直接采用FCFS
- Thread Scheduling - 线程调度：
    - Static scheduling - 静态调度
        - Load Sharing - 负载分配
          Processes are not assigned to a particular processor.
        - Gang Scheduling - 组调度
          **A set of related thread** scheduled to run on **a set of processors** at the same time, on a one-to-one basis.
        - Dedicated processor assignment - 专用处理器分配
          Provides implicit scheduling defined by the assignment of threads to processors.
    - Dynamic scheduling - 动态调度
      **The number of threads** in a process can be **altered** during the course of execution.
- 具体算法
    - Earliest Deadline First(EDF) - 最早截止时间优先
    - Rate Monotonic Scheduling (RMS) - 速率单调调度

# Ep.5 I/O - 输入输出设备

## 1. I/O

There may be differences of several orders of magnitude between the **data transfer rates**.
传输速率差几个数量级。

- Evolution
    1. **Processor directly controls** a peripheral(外围的) device.
       **处理器直接控制**外围设备。
    2. A **controller** or **I/O module** is added.
       增加了**控制器**或**I/O模块**。
    3. Same configuration as step 2, but now **interrupts** are employed.
       与阶段2的配置相同，但采用了**中断**方式。
    4. The I/O module is given direct control of memory via **DMA**.
       I/O模块通过**DMA(直接存储访问器)**直接控制存储器。
    5. The I/O module is enhanced to become a **separate processor**, with a **specialized instruction set** tailored for(专门定制的) I/O.
       I/O模块有一个**单独的处理器**，有专门为I/O设计的**特殊指令集**。
       不光有内存，还有一些特殊的指令集，此时的I/O Module称为“**I/O通道**”(**I/O Channel**)。

6. The I/O module has a **local memory** of its own and is, in fact, **a computer** in its own right.
   I/O模块有自己的**局部存储器**，事实上其本身就是**一台计算机。**
   此时的I/O Module称为“**I/O处理器**”(**I/O Processor**)。

- Buffer - 缓冲区
  - Singal Buffer - 单缓冲
  - Double Buffer - 双缓冲(**buffer swapping, 缓冲交换**)
    Use two system buffer instead of one.
    用两个系统缓冲区来代替一个。
    A process can **transfer data** to or from **one buffer** while the operating system **empties or fills the other buffer**.
    在一个进程向一个缓冲区中传送数据（或取数据）的同时，OS也正在清空（或填充）另一个缓冲区。故使得操作效率变高。
  - Circular Buffer - 循环缓冲

## 2. Disk

- Disk performance - 磁盘表现

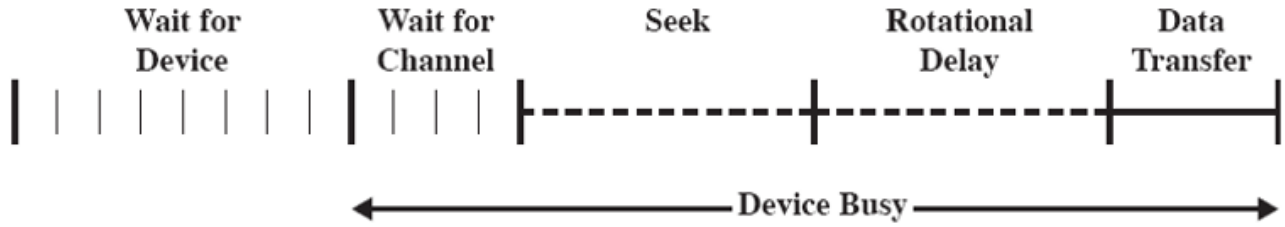


**Figure 11.6 Timing of a Disk I/O Transfer**

  - Wait for Device - 等待设备
  - Wait for Channel - 等待通道
  - Seek - 寻道：寻找数据所在位置，将磁头固定到磁道。
  - Rotational Delay - 旋转延迟：将磁头旋转到适当扇区。
  - Data Transfer - 数据传输
- 磁盘调度策略
  - First-In, First-Out(FIFO) - 先进先出
  - Shortest Service Time First(SSTF) - 最短服务时间优先
  - SCAN - 扫描算法
  - C-SCAN - 循环扫描算法
- Disk Cache - 磁盘缓存
  Disk cache is a buffer in main memory for disk sectors(扇区).
  磁盘高速缓存是内存中为磁盘扇区设置的一个缓冲区。
  Contains a copy of some of the sectors on the disk.
  包含有磁盘中某些扇区的副本
- Cache置换算法
  - Least Recently Used(LRU) - 最近最少使用
  - Least Frequently Used(LFU) - 最不常使用

- Frequency-Based Replacement - 基于频率的置换算法
  - FIFO版本：分为新区(New Section)和老区(Old Section)，新区中被命中计数器不变，老区被命中计数器++（如下图所示）
    存在问题：之前加到队列里，但一直未被再次访问变老，可能后面一段时间才会被局部性原理反复访问，但已经没有时间让其计数器++了！
  - 三区版本：分为新区(New Section)、中间区(Midele Section)和老区(Old Section)，区别是只有老区的会被置换。
    因此有足够的时间让新区和中间区的计数器++。

- RAID

表 11.4 RAID 级别

| 类别 | 级别 | 说明 | 磁盘请求 | 数据可用性 | 大 I/O 数据量传送能力 | 小 I/O 请求率 |
|---|---|---|---|---|---|---|
| 条带化 | 0 | 非冗余 | $N$ | 低于单个磁盘 | 很高 | 读和写都很高 |
| 镜像 | 1 | 被镜像 | $2N$ | 高于 RAID2、3、4 或 5；低于 RAID6 | 读时高于单个磁盘；写时与单个磁盘相近 | 读时最快为单个磁盘的两倍；写时与单个磁盘相近 |
| 并行访问 | 2 | 通过汉明码实现冗余 | $N+m$ | 明显高于单个磁盘；与 RAID3、4 或 5 可比 | 所有列出方案中最高的 | 约为单个磁盘的两倍 |
| | 3 | 交错位奇偶校验 | $N+1$ | 明显高于单个磁盘；与 RAID2、4 或 5 可比 | 所有列出方案中最高的 | 约为单个磁盘的两倍 |
| 独立访问 | 4 | 交错块奇偶校验 | $N+1$ | 明显高于单个磁盘；与 RAID2、3 或 5 可比 | 读时与 RAID0 相近；写时明显慢于单个磁盘 | 读时与 RAID0 相似；写时明显慢于单个磁盘 |
| | 5 | 交错块分布奇偶校验 | $N+1$ | 明显高于单个磁盘；与 RAID2、3 或 4 可比较 | 读时与 RAID0 相近；写时慢于单个磁盘 | 读时与 RAID0 相似；写时通常慢于单个磁盘 |
| | 6 | 交错块双重分布奇偶校验 | $N+2$ | 所有列出方案中最高的 | 读时与 RAID0 相近；写时慢于 RAID5 | 读时与 RAID0 相近；写时明显慢于 RAID5 |

$N$ 表示数据磁盘数量；$m$ 与 $\log N$ 成比例。

# 3. File

## (1) 文件分类

- Field - 域（字段）
  - **Basic element** of data, contains a single value.
    数据的**最基本单位**，包含单个的值。
  - Fixed or variable length.
    有定长或变长。
- Record - 记录（元组）
  - **Collection of related fields** that can be treated as a unit by some application program.
    **相关字段的独立集合**，被一些应用程序作为一个整体。
  - Fixed or variable length.
    定长或变长。
- File - 文件
  - **Collection of similar records**, treated as a single entity.
    **相关记录的集合**，被当作单独的实体。
  - May be **referenced by name**.
    通过**文件名**来区分。

    - **Access control** restrictions usually apply at the file level.
      **访问控制**一般就在文件这一层级应用。
- Database - 数据库
    - **Collection of related data.**
      **相关数据的集合**。
    - **Relationships** among elements of data are explicit.
      数据之间的**关系**很明确清楚。
    - Designed for use by a number of **different applications**.
      被设计用于让**多个应用程序**使用。
    - Consists of one or more **types of files**.
      **多个文件**的集合。

## (2) File Organization and Access - 文件组织和访问

- Pile
- Sequential file
- Indexed sequential file
- Indexed file
- Hashed

## (3) Record Blocking - 记录组块

Records are the logical unit of access of a structured file.
记录是访问结构化文件的逻辑单元。
Blocks are the unit of I/O with secondary storage.
块是与辅存进行I/O操作的基本单位。
For I/O to be performed, records must be organized as blocks.
记录必须组织为块，才能执行I/O。

有三种组块方式：

- Fixed-Length Blocking - 定长组块
- Variable-Length Spanned Blocking - 变长跨越式组块
- Variable-Length Unspanned Blocking - 变长非跨越式组块