1. 概述 (Overview)

Introduction:

计算机系统分为四个部分：Hardware, OS, System/Application Programs, Users. 包括两类接口：**命令级接口**提供键盘或鼠标命令；**程序级接口**提供系统调用。

OS 服务:

- 对于**User**，OS 要容易学容易用，可靠性高，速度快。因此提供以下服务：User Interface(CLI, GUI), Program Execution, I/O Operations, File System Manipulation, Communication, Error Detection.
- 对于**System**，OS 要易于设计实现维护，灵活可靠，错误少，高效，因此提供以下服务：Resource Allocation, Accounting, Protection and Security.

操作系统发展历程:

单道批处理 Batch Processing System

- Jobs 存储在内存/外存中，而内存中始终有一个 Job 在运行
- 操作系统负责在内存中 Job 结束时加载下一个 Job 到内存中运行
- 我们将加载到内存中运行的程序称为**进程 (Process)**，这是一个动态的概念

多道批处理 Multiprogramming Batch System

- 在批处理系统基础上，当正在运行的 Job 发生 I/O 操作时，操作系统负责让 CPU 转而运行另一个 Job
- 多道批处理系统通过这种方式避免了批处理系统等待用户输入等 I/O 事件的 CPU 时间浪费
- 在宏观上，多道批处理系统是**并发 (Concurrency)** 的；在微观上，它仍然是顺序串行的

分时系统 Time-Sharing System

- 将时间划分为很小的时间片，操作系统负责安排各个 Job / 各个用户轮流运行
- 由于时间频率很高，远高于人类的反应速度，因此操作系统可以快速在多个进程之间切换运行，让用户误以为这些进程是同时运行的
- 分时系统属于**多任务的 (Multitasking)**，但已经不是批处理 (Batch) 的了；它相比多道批处理系统拥有了更好的**交互性**

集群 Clustered

一组互联主机构成的统一计算机资源。对称集群：多个节点跑程序，互相监视；非对称：一台计算机处于热备用模式。集群用于高性能计算。

实时系统 Real-Time System

- 若某个动作必须在规定时间内完成，则为**实时系统**，否则系统失败
- 若能够接受偶尔违反时间规定且不会引起永久性损害，则为**软实时系统**
- 及时性和可靠性高，交互性不如分时系统

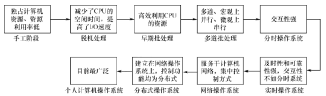


图 1.1 操作系统的发展历程

系统调用:

大部分用户都通过打包后的 **API** 来间接调用 OS 提供的编程接口服务 System Calls。三个最普遍的 API 为 Win32 API (Windows), POSIX API (Unix-like), Java API (Java 虚拟机)。

当 CPU 处于用户态时，只能执行非特权指令，而操作系统内核运行在内核态。虽然我们可以在用户态调用相关的 Syscall 指令，但是执行系统调用时会发生特权级的切换，因此它属于**特权相关指令 (privilege-sensitive)**，**不属于特权指令**。

可以通过命令 strace -xf [command] 查看某命令执行过程中发生的所有系统调用。

仅内核态可做的：set value of the system timer, Clear Memory, Turn off interrupts, Modify entries in device-status table, Access I/O devices.

看上去像但不是特权的 (in kernel mode): read the clock, issue a trap instruction, switch from user to kernel mode.

按照功能，System Calls 可以分为以下几类：Process Control, File Management, Device Management, Information Maintenance (e.g. time, date), Communications, Protection

系统调用的三种传参方式：寄存器传参（但有时需要的参数大于可用的寄存器数）；参数存在内存中 Block 或 Table 中，将对应地址通过寄存器传参（Linux 和 Solaris 使用的方法）；堆栈传参 (User Push, OS Pop).

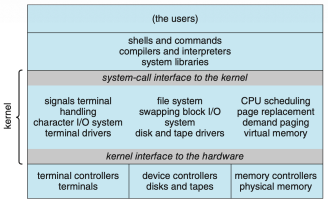
系统程序可分为：

- **File Manipulation:** cp, rm, mkdir, ls
- **Status Information:** top, ps, df
- **File Modification:** editors like vi, emacs
- **Programming Language Support:** gcc, javac, gdb
- **Program Loading and Execution:** load, exec
- **Communications:** ssh, ftp, mail

OS 结构:

单内核 Monolithic Structure

最原始的 UNIX OS 只包含 System Programs 和 Kernel 两个层次，只在一定程度上分层，所有内核功能都放入一个单一的、静态的二进制文件中，运行在单一地址空间内。



优点：结构简单，系统接口调用开销小，内核内通信快。

分层法 Layered Approach

操作系统被划分为多个层次，每个层次都**只能使用**它的下一层提供的 functions 和 services，这是一个单向依赖。

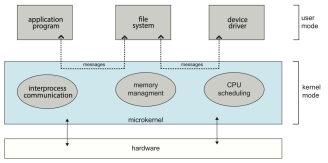
其中，最底层 layer 0 是硬件层，最高层 layer N 是用户接口层。

优点：简化系统设计与调试，易于扩充维护。

缺点：依赖关系固定不灵活，执行功能要自上而下穿越多层，开销大。

微内核 Microkernel

本来放在内核里的功能尽量移动到用户态 (User Space) 中运行，不同模块之间通过 Message Passing 进行交流。



优点：可拓展、可移植、可靠性高、安全性高

缺点：微内核频繁在用户态和内核态切换，开销大，性能低。

模块化 Modules

大多数现代 OS 使用的方式。采用面向对象设计，将各个核心功能都做成独立模块。不同模块之间不使用 Message Passing，而是直接调用彼此的接口函数。

分层法也是一种特殊的模块化设计。而**可加载内核模块 loadable kernel modules, LKM**在实现上更加优秀，相比于分层法，它允许任何模块都能直接调用其它模块的接口。在启动时，内核的核心模块会被加载；而运行时需要用到其它模块功能时才会加载对应模块。

混合系统 Hybrid Systems

- Linux = Monolithic + Modules.
- Windows = Monolithic + Microkernel (+ Modules).

系统启动: Bootstrap Program → Bootloader (MBR) → Kernel → Init.

Bootstrap Program 和 Bootloader 都**不属于 OS 内核**的一部分。Bootstrap Program 存在于 ROM 中，负责初始化硬件并加载 Bootloader。Bootloader 存在于磁盘的 MBR 中，负责加载 OS 内核。

2. 进程 (Process)

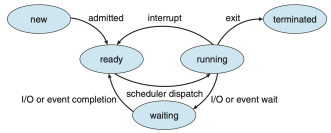
概念: 进程是程序的动态执行实体，每个进程都有自己独立的内存空间包含用户部分和内核部分：

用户部分：代码区 (text section)、数据区、堆、栈

内核部分：进程控制块 Process Control Block PCB 包括 Process State, Program Counter, CPU Registers, CPU Scheduling Info, Memory Management Info, Accounting Info, I/O Status Info.

当 CPU 需要在不同进程间切换时，它保存上一个进程的 PCB，并加载下一个进程的 PCB，即**Context Switch**。两个进程的切换由 Dispatcher 完成，切换延迟称为**Dispatch Latency**。

状态:



running -> waiting 是主动的；waiting -> ready 是被动的。

1. 有几个核就最多有几个进程处于 running 状态
2. ready queue 最多有 n-1 个进程，最少有 0
3. waiting queue 最多有 n 个进程，最少有 0

进程操作:

Creation: 父进程创建子进程时，关于资源是否共享有三种策略：

- **父与子共享所有资源:** 比如打开的文件、内存空间均共享
- **子只共享父进程资源的子集:** 比如只继承打开的标准输入输出
- **父与子完全不共享资源:** 子进程有自己的独立资源空间

Linux API **fork()** 采用第一种策略。执行 fork() 后，父进程中返回子进程的 PID，子进程中返回 0。

- **Orphan:** 父进程先结束，还在运行的子进程即为孤儿进程，被 PID=1 收养
- **Zombie:** 子进程先结束，且父进程未调用 wait() 回收资源，子进程变为僵尸进程
- **Daemon:** 连续调用多次 fork()，脱离终端控制，在后台运行的守护进程

Termination: 进程执行完后，调用 **exit()** 告知 OS 释放资源，此时父进程可调用 **wait()** 获取返回值。

除此之外，父进程还可以用 **abort()** 强制终止子进程，常见原因有：子进程超出了分配的资源；子进程任务不再需要；父进程自己终止，顺带终止子进程。

进程通信 IPC:

消息传递 Message Passing

- 通过 send/receive 消息通信，系统调用实现
- 适用于分布式系统或安全隔离环境；交换少量数据

共享内存 Shared Memory

- 进程间共享一块内存区域，建立共享区域后无需内核帮助
- 比消息传递快，但需要同步机制防止冲突
- 一个进程内的线程之间天然共享内存；不同进程之间需要通过特殊系统调用实现

管道 Pipe

- 一种特殊共享文件，只要非满即可写，非空即可读
- 半双工通信，单向数据流

除此之外还有 **Sockets**（双方不在同一台机器上）、**信号 Signals**（异步通知）、**Remote Procedure Calls**（远程过程调用）等。

3. 线程 (Thread)

特点: 资源分配单位是进程，**调度单位是线程**(unit of dispatch)

共享: Code, Data, Heap, Files. **独享:** Regs, Stack, PC (+Thread ID).

优点: 创建新线程耗时少；context switch 开销小 (**economy**)；线程间通讯可使用 shared memory；**responsiveness**；**scalability**；**resource sharing**

缺点: weak isolation；one thread fails, process fails

User-Level Threads (ULT) 有关线程管理的所有工作都在用户态完成，内核意识不到线程的存在。

- 用户应用程序可以使用**线程库**设计成多线程程序，如 POSIX Pthreads, Windows API, Java 等
- 使用 ULT 的系统，调度仍然以进程为单位，内核只知道有一个进程在运行
- **优点:** 线程切换不涉及内核，开销小；进程根据自身需求设计调度算法；实现与操作系统无关，易移植
- **缺点:** 无法利用多处理器；一个线程阻塞则整个进程阻塞

Kernel-Level Threads (KLT) 有关线程管理的所有工作都在内核态完成

- OS 为每个内核级线程设置线程控制块 TCB，其调度以线程为单位
- **优点:** 多处理器可并行执行多个线程；一个线程阻塞，其他线程可继续运行
- **缺点:** 线程切换需内核介入，开销大；线程调度受限于内核，灵活性差

多线程模型:

- **多对一:** 相当于 ULT，阻塞则全阻。当一个 ULT 要访问内核时，才将其映射到 KLT 上
- **一对一:** 相当于 KLT，并发好，开销大。现代 OS 多用此法
- **多对多:** n 个 ULT 映射到 m 个 KLT 上，$n \geq m$

Fork() & Exec():

在一个多线程进程中，如果一个线程调用 fork()，是会复制所有线程还是只复制调用者呢？一些 UNIX 系统为两者都提供了对应版本的 fork()，具体使用哪种视开发者的目的而选择。(Linux 为只复制调用者)

例如 pthreads 中，fork() 的标准行为如下：

Only the thread that invokes fork() is duplicated, and the child process inherits the parent's address space but contains only that single calling thread.

但一个线程使用 exec() 系统调用，则整个进程都会被替换。

线程池 Thread Pool: 当提交给池的任务能够**异步执行**时，使用线程池效果最佳

1. 在启动时创建一定数量的线程，并将它们置于一个池中，这些线程会处于空闲状态并等待被分配任务
2. 当服务器接收到请求时，它不会直接创建新线程，而是将请求提交给线程池后继续等待其他请求
3. 如果池中有空闲的线程，该线程会被唤醒并立即处理请求；若无线程可用，则该任务会排队等候，直到有线程释放出来
4. 一旦某个线程完成了它的工作，它会返回到池中，等待之后的工作分配

优点: 比等待创建新线程更快；限制了在任何时刻存在的线程数量；将要执行的任务与创建任务的机制分离开来，使我们能够采用不同的策略来运行任务，比如可以安排任务在延迟一段时间后执行或定期执行

4. CPU 调度 (Scheduling)

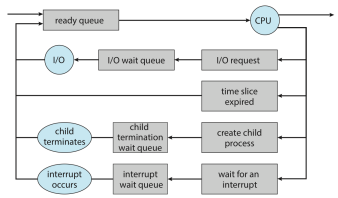
概念:

Multiprogramming 环境下，进程的个数通常大于 CPU 的个数。CPU 调度就是 OS 关于哪个 ready 进程可以运行（使用 CPU）以及运行多久的决定。

分派程序 Dispatcher 用于将 CPU 的控制交给短程调度所选择的进程，功能有：上下文切换，切换到用户模式，跳转到用户程序合适位置。

为了进行进程的调度，OS 维护了三条相关的 Scheduling Queues:

- **Job Queue:** 系统中所有进程的队列
- **Ready Queue:** 所有处于 ready 状态的进程队列
- **Device Queue:** 也称 waiting queues，等待某个 I/O 设备的进程队列



Scheduler 是决定哪个程序该执行的程序：

- **Long-term / Job:** 决定哪些进程可以进入内存，放入 ready queue
  - 控制系统中同时有多少个进程在竞争 CPU，如果太多则会造成频繁的切换，导致效率下降；如果太少则 CPU 可能空闲，浪费资源
  - 通常会让 I/O-bound Process 和 CPU-bound Process 混合进入系统
- **Short-term / CPU:** 从 ready queue 中决定下一个由 CPU 执行的进程
- **Medium-term:** 决定哪些进程可以进入 ready queue
  - 某些暂时不能运行的进程可能因为内存不足而被换出到磁盘上，等内存充足时再换入
  - 目的是提高内存利用率和系统吞吐量，降低多道程序度（内存中进程数）

评价:

非抢占式 non-preemption

1. 即便有优先级更高的任务进入就绪队列，也不会打断当前 CPU 正在执行的进程，直到该进程运行结束或某些事件发生。

2. 调度发生时机为 Running => Terminated 和 Running => Waiting。
3. 优点是实现简单，系统开销小。

抢占式 preemptive

1. 若有优先级更高的任务进入就绪队列，则允许调度程序根据某种原则去暂停当前正在执行的进程，将 CPU 分配给这个更重要的进程。
2. 调度可以发生在任何时候。
3. 抢占式调度提高了系统吞吐率和响应效率，但使得调度控制更复杂。

对于调度算法自身，要求易于实现，开销较小。对于用户：

- **CPU Utilization**: CPU 使用率
  - 等于 CPU 工作时间 / 总时间
- **Throughput**: 吞吐量，每个时间单元内完成的进程数量
- **Turnaround Time**: 周转时间，从进程创立到进程完成的时间，是等待进入内存、在 ready queue 中等待、在 CPU 上执行、I/O 执行等时间的总和
  - 等于进程的完成时间减去到达时间
- **Waiting Time**: 等待时间，在 ready queue 中（或在 Ready 状态下）等待所花的时间之和
  - 等于进程的周转时间减去 CPU Burst Time 和 I/O Burst Time
- **Response Time**: 响应时间，交互系统从进程创立到第一次产生响应的时间
  - 等于进程第一次获得 CPU 的时间减去到达时间

算法：

First-Come, First-Served (FCFS) 非抢占式

利于长进程，不利于短进程；利于 CPU-Bound，不利于 I/O-Bound。

Shortest-Job-First (SJF) 抢占式 & 非抢占式

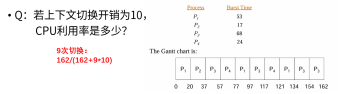
- 从 Ready Queue 中选择估计下一次运行时间最短的进程运行。
- 对于**非抢占式系统**，「估计下一次运行时间」指该进程所需的总运行时间 (shortest-next-CPU-burst)；
- 对于**抢占式系统**，「估计下一次运行时间」指该进程的剩余运行时间 (shortest-remaining-time-first, SRTF)。
- 给定一个 Processes 集合，SJF 可以确保具有最佳的 Minimum Average Waiting Time。
- 缺点是对长进程不利，会导致 Starvation；完全未考虑进程优先级；预测不准确。

可用 Exponential Averaging 方法预测下一次 CPU Burst 时间：

$$\tau_{n+1} = \alpha \cdot t_n + (1-\alpha) \cdot \tau_n$$

Round-Robin (RR) 抢占式

- 定义了一个较小的固定时间单元时间片（time slice / quantum），保证任何一个进程都不会连续运行超过一个时间片的时间（除非这是 Ready Queue 的唯一进程）。
- 如果唯一的进程时间片耗尽的同时，另一个进程进入 Ready Queue，则下一个运行的进程一定是新进入的进程。除了这种情况外，则具体看题目要求处理进入顺序，默认新进程优先。
- 相比 SJF 算法而言，RR 算法的**平均等待时间**更长，但是**响应时间**更短。
- Quantum 的选择对 RR 算法的性能影响很大。
  - 如果时间片足够大，以至于所有进程都能在一个时间片内执行完毕，则 RR 算法退化为 FCFS 算法；
  - 如果时间片非常小，则 CPU 将在进程间频繁切换，真正用于运行用户进程的时间将减小。



除此之外，响应时间固定时，就绪进程越多，时间片越小；时间片固定时，用户越多，响应时间越长；要求用户输入能在一个时间片内完成，否则平均周转和平均带权周转时间会变大。

Priority Scheduling 抢占式 & 非抢占式

- 为每个进程设定优先级，每次从就绪队列中选择优先级最高的进程
- 优先级可以是**静态的**：如进程类型、对资源的要求、用户要求
- 优先级也可以是**动态的**：如优先级随等待时间增加而提高
- 通过动态优先级的方式可以避免 Starvation，aging 逐渐增加等待时间长的进程的优先级

Multi-Level Queue

- 在系统中设置多个 Ready Queue，将不同类型或性质的进程固定分配到不同的就绪队列，系统可根据不同用户进程的需求为每个队列选择不同调度策略。
- 例如对于一个 B/S 程序，前台队列可以使用 **RR** 调度保证 response，后台队列可以使用 **FCFS**。当然，不同队列之间也应该有调度，例如：
  - 方法一：Fixed Priority Sheduling，仅优先级高的前台队列为空时才能够执行优先级低的后台队列
  - 方法二：Time Slice，两个队列都有一个确定的 CPU 时间，当时间到了则转向另一个队列，一般 **80%** 时间用于前台，**20%** 时间用于后台。

Multi-Level Feedback Queue

- 融合前几种算法的优点，通过动态调整进程优先级和时间大小，可以兼顾多方面的系统目标。
- 每个队列有不同的优先级和时间片大小，优先级高的队列时间片小，优先级低的队列时间片大。
- 新进程进入最高优先级队列，若在时间片内未完成，则被降级到下一级队列。
- 通常，若某进程占用 CPU 时间较长，则更容易被移动至优先级更低的队列中；相对的，I/O Bound 或交互式的进程则更容易留在优先级更高的队列中

5. 同步 (Synchronization)

两个信号产生竞争，其竞争情况影响最终结果，称为竞态条件 (Race Condition)。

Critical Section 临界区：

```
while (true) {
  entry section;    // 进入区 检查进程是否可以进入临界区
  critical section; // 临界区 访问临界资源
  exit section;     // 退出区 清除正在访问临界区的标志
  remainder section;// 剩余区 其它代码
}
```

Mutual Exclusion 互斥：

互斥也被称为间接制约关系，当一个进程进入 Critical Section 使用临界资源时，另一个进程必须等待。

1. **空闲让进 Progress**：临界区空闲，则允许请求进入的进程立即进入
2. **忙则等待 Mutex**：已有进程进入临界区时，其它试图进入的进程必须等待
3. **有限等待 Bounded Waiting**：保证等待时间有限
4. **让权等待**：当进程不能进入临界区时，应立即释放处理器，防止忙等待

前三个要求必须实现，第四个尽可能实现。

软件实现方法

单标志法使用一个公用整型变量表示允许进入临界区的进程编号。

Process 0

```
while (true)
{
  while (turn!=0);
  critical section;
  turn = 1;
  remainder section;
}
```

Process 1

```
while (true)
{
  while (turn!=1);
  critical section;
  turn = 0;
  remainder section;
}
```

如果进程 P_0 离开循环，那么变量 $turn$ 永远不会变为 1，导致 P_1 永远无法进入临界区，违反了**空闲让进**条件。

双标志先检查法使用布尔变量 flag[0] 和 flag[1] 分别表示两个进程是否想进入临界区。

Process 0

```
while (true)
{
  while (flag[1]);
  flag[0] = true;
  critical section;
  flag[0] = false;
  remainder section;
}
```

Process 1

```
while (true)
{
  while (flag[0]);
  flag[1] = true;
  critical section;
  flag[1] = false;
  remainder section;
}
```

对 flag 的设置不是原子性的，所以存在以下情况：P_0 和 P_1 同时访问临界资源，在 entry section 对 flag 检查都通过，导致双方同时进入临界区，违背**忙则等待**原则。

此时若将检查放在设置后面，虽然不会出现双方同时进入的情况，但是会出现双方同时对自己的 flag 进行了设置，导致检查都不通过的**饥饿**现象。

Peterson 算法结合了以上算法的思想，利用 flag[] 解决互斥访问问题，利用 turn 解决饥饿问题，其核心思想是，若双方都想要进入临界区，则将进入的机会“谦让”给对方：

Process 0

```
while (true)
{
  flag[0] = true;
  turn = 1;
  while (flag[1]
    && turn==1);
  critical section;
  flag[0] = false;
  remainder section;
}
```

Process 1

```
while (true)
{
  flag[1] = true;
  turn = 0;
  while (flag[0]
    && turn==0);
  critical section;
  flag[1] = false;
  remainder section;
}
```

Peterson 算法很好地遵循了**空闲让进、忙则等待、有限等待**三个原则，但仍未实现**让权等待**。

硬件实现方法

屏蔽中断

CPU 只会在发生中断时切换进程，因此屏蔽中断能够保证当前运行的进程让临界区的代码顺利执行完。

```
...
关中断
临界区
开中断
...
```

缺点是限制了 CPU 交替执行程序的能力，效率低；只能在单处理器系统中使用；将开关中断权限给用户不明智。

TestAndSet 是一个原子操作：

```
bool TestAndSet(bool *lock) {
    bool old = *lock; // 存放 lock 旧值
    *lock = true;     // 无论之前是否加锁，都将 lock 设置为 true
    return old;       // 返回旧值
}
```

通常用 true 表示资源正被占用，用 false 表示空闲：

```
while TestAndSet(&lock); // 加锁并检查
critical section;
lock = false;            // 解锁
remainder section;
```

TS 指针相当于保证了检查和设置这两个操作是原子性的。

暂时无法进入临界区的进程会占用 CPU 循环执行 TS 指令，因此没有实现**让权等待**原则。

Swap 也是原子操作，用来交换两个字节的内容：

```
void Swap(bool *a, bool *b) {
    bool temp = *a;
    *a = *b;
    *b = temp;
}
```

此时每个临界资源都有一个共享布尔变量 lock，初值为 false；每个进程都有一个局部布尔变量 key，初值为 true。

其原理和 TS 指令实现互斥区别不大，过程描述如下：

```
bool key = true;
while (key != false) Swap(&lock, &key);
critical section;
lock = false;
remainder section;
```

硬件实现方法的优缺点：

- **优点**
  1. 简单，容易验证正确性
  2. 适用于任意数量的进程，支持多处理器
  3. 支持系统中有多个临界区
- **缺点**
  1. 等待进入临界区的进程会占用 CPU 执行 while 循环，没能实现**让权等待**
  2. 从等待进程中随机选择一个进程进入临界区，有的进程可能一直选不上，从而导致饥饿现象（**有限等待**原则）

Mutex Lock 互斥锁

互斥锁是解决临界区最简单的软件工具。一个进程在进入临界区时要调用 acquire()，来获得锁；在退出临界区时要调用 release() 来释放锁。

```
acquire() {
    while (!available);  // 忙等待
    available = false;   // 获得锁
}
release() {
    available = true;    // 释放锁
}
// 都是原子操作，通过硬件机制实现
```

上面描述的互斥锁也称自旋锁 Spin Lock，它的缺点是需要忙等待，当有一个进程在临界区时，任何其它进程在进入临界区前都需要连续循环调用 acquire()，这种连续循环显然浪费了 CPU 周期。因此，互斥锁通常用于多处理器系统，一个线程在一个处理器上旋转，而不影响其它线程的执行。

自旋锁的优点是进程在等待锁期间没有发生上下文切换，并且等待的代价不高。不需要忙等待的互斥锁称为 Sleep Lock。

Semaphore 信号量

信号量是用来解决互斥和同步问题的功能更强的机制，它只能被两个标准原语 wait() 和 signal() 访问，也可简写为 P() 和 V()。

整型信号量被定义为一个用于表示资源数量的整型量 S。值大于等于 0 即可的称为**计数信号量**，只能取 0 和 1 的称为二**进制信号量**，后者可用于实现互斥锁。

相比于普通整型变量，整型信号量只有三种操作：初始化、P 操作、V 操作：

```
wait(S) {          // 相当于进入区
  while (S <= 0); // 若资源数不够，忙等待
  S = S - 1;       // 若资源数够，则占用一个资源
}
signal(S) {        // 相当于退出区
  S = S + 1;       // 释放一个资源
}
```

整型信号量的使用方式基本与互斥锁相似，因此它也有忙等待的缺点。

记录型信号量是一种不存在忙等现象的进程同步机制，它除了需要一个用于表示资源数量的整型变量 value 外，还增加了一个进程链表 L，用于链接所有等待该资源的进程：

```
typedef struct {
    int value;
    struct process *L;
} semaphore;
```

此时负数的绝对值代表了正在等待该信号量的进程数。

对应的 P 操作和 V 操作描述如下：

```
wait(semaphore S) {
  S.value--;
  if (S.value < 0) {
      add this process to S.L;
      block(S.L);   // running -> waiting
  }
}

signal(semaphore S) {
  S.value++;
  if (S.value <= 0) {
      // 仍有进程在等待该资源
      remove a process P from S.L;
      wakeup(P);    // waiting -> ready
  }
}
```

此时，为了使多个进程能够互斥地访问某个临界资源，需要为该资源设置一个互斥信号量 S，其初值为 1，对应可用资源数为 1，然后将各个进程访问该资源的临界区过程置于 P(S) 和 V(S) 之间：

```
semaphore S = 1;
Pi() {
    ...
    P(S);
    critical section;
    V(S);
    ...
}
```

优先级倒置 Priority Inversion:

优先级较低的进程 PL 持有优先级较高进程 PH 所需的锁，导致 PH 无法优先执行。

解决方法是**优先级继承 Priority Inheritance**，临时提升 PL 的优先级至 PH 的优先级，迅速完成其工作，待 PL 释放锁后再恢复原优先级。

经典问题：

Bounded-Buffer Problem, 生产者消费者问题：

```
semaphore mutex = 1;  // 临界区互斥信号量
semaphore empty = n;  // 空闲缓冲区
semaphore full  = 0;  // 缓冲区初始化为空

producer() {
    while (1) {
        produce a item;
        wait(empty);   // 获取空缓冲区单元
        wait(mutex);   // 进入临界区
        add the item to the buffer;
        signal(mutex); // 退出临界区
        signal(full);  // 满缓冲区数 +1
    }
}

consumer() {
    while (1) {
        wait(full);    // 获取满缓冲区单元
        wait(mutex);   // 进入临界区
        remove an item from buffer;
        signal(mutex); // 退出临界区
        signal(empty); // 空缓冲区数 +1
        consume the removed item;
    }
}
```

Readers-Writers Problem, 读者写者问题：

```
int count = 0;      // 整型信号量
semaphore mutex = 1;
semaphore rw    = 1;

writer() {
  while (1) {
    P(rw);          // 互斥访问共享文件
    write sth...
    V(rw);          // 释放共享文件
  }
}

reader() {
  while (1) {
    P(mutex);       // 互斥访问 count 变量
    if (count == 0) // 当第一个 Reader 读共享文件时
      P(rw);        // 阻止 Writer 访问共享文件
    count++;
    V(mutex);       // 释放 count
    read sth...
    P(mutex)        // 互斥访问 count 变量
    count--;
    if (count == 0) // 当最后一个 Reader 离开
      V(rw);        // 允许写进程写
    V(mutex);       // 释放 count
  }
}
```

Dining-Philosophers Problem, 哲学家进餐问题：

```
semaphore chopstick[5] = {1,1,1,1,1};

// Process i
Pi() {
  do {
    P(chopstick[i]);       // 取左边筷子
    P(chopstick[(i+1)%5]); // 取右边筷子
    Dining...
    V(chopstick[i]);       // 放回左边筷子
    V(chopstick[(i+1)%5]); // 放回右边筷子
  } while(1);
}
```

上述贪心算法虽然浅显易懂，但是当所有哲学家同时想要进食，并且同时取到左边筷子时，此时想要取右边筷子就会进程阻塞，发生**死锁**。为此，可以使用一些限制条件：

1. 至多允许 4 名哲学家同时进食，以保证至少有一名哲学家能拿到左右两边的筷子
2. 要求奇数号哲学家先拿左边筷子，偶数号哲学家先拿右边筷子，以保证相邻哲学家都想进食时，

只有一名哲学家可以取到筷子，而另一名阻塞等待

3. 仅当哲学家两边筷子都可用时，才允许他拿起筷子

对于方案 3，仅需定义一个访问 chopstick[] 资源的互斥信号量 mutex 即可：

```
semaphore chopstick[5] = {1,1,1,1,1};
semaphore mutex = 1;        // 临界区互斥信号量

// Process i
Pi() {
  do {
    P(mutex);
    P(chopstick[i]);        // 取左边筷子
    P(chopstick[(i+1)%5]); // 取右边筷子
    V(mutex);
    Dining...
    V(chopstick[i]);        // 放回左边筷子
    V(chopstick[(i+1)%5]); // 放回右边筷子
  } while(1);
}
```

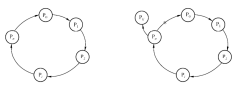
6. 死锁 (Deadlock)

概念： **死锁**指多个进程因竞争资源而造成的一种僵局，各个进程互相等待对方手里的资源而阻塞。它与**饥饿**有如下区别：

1. 发生饥饿的进程可以只有一个；而死锁是因循环等待对方手里的资源导致的，因此发生死锁的进程必然大于等于两个
2. 发生饥饿的进程可能处于 Ready 状态（长期得不到 CPU），也可能处于 Waiting 状态（长期得不到 I/O 设备）；而发生死锁的进程只能处于 Waiting 状态

四个必要条件： （必须同时满足）

- **互斥条件 Mutex：** 进程要求对所分配的资源进行排他性使用，即一段时间内某资源只能为一个进程占用，此时其它请求该资源的进程只能等待
- **不可剥夺条件 No Preemption：** 进程所获得资源使用完前，不能被其它进程夺走，只能主动释放
  - 因此对可剥夺资源（如 CPU 和 Memory）竞争不产生死锁
- **请求并保持条件 Hold & Wait：** 进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占用，此时请求进程被阻塞，但自己已获得的资源保持不变
- **循环等待条件 Circular Wait：** 存在资源循环等待链，链中每个进程已获得的资源同时被链中下一个进程所请求
  - 例：P_0 等待 R_1（被 P_1 占用），P_1 等待 R_2（被 P_2 占用），...，P_n 等待 R_0（被 P_0 占用）



有环不一定就死锁，例如上右图，如果此时 P_K 将资源释放，P_n 就能获得该资源，循环等待就会被打破。因此，我们说循环等待只是死锁的**必要条件**；但若系统中每类资源都只有一个资源，则资源分配图中含圈就变成了系统出现死锁的**充分必要条件**。

处理策略对比：

实现方案	资源分配策略
死锁预防	设置限制条件， 破坏产生死锁的 4 个必要条件
死锁避免	资源动态分配过程中， 寻找可能的安全允许顺序， 防止系统进入不安全状态
死锁检测	允许死锁，定期检查死锁是否已经发生， 通过剥夺等措施解除死锁

Deadlock Prevention 死锁预防： 破坏四个必要条件之一。

1. **破坏互斥条件**
   - 若将只能互斥使用的资源改造为允许共享使用，则系统不会进入死锁状态
   - 但很多临界资源只能互斥使用，因此该方法不太可行
2. **破坏不可剥夺条件**
   - 当一个已经保持了某些不可剥夺资源的进程，请求新的资源而得不到满足时，它必须释放已经保持的所有资源
   - 这意味着，进程已占有的资源会被暂时释放，或者称被剥夺了
   - 该方法实现复杂，且可能会导致前一阶段工作失效
3. **破坏请求并保持条件**
   - 要求进程在请求资源时不能持有不可剥夺资源，具体有以下两种实现方式：
     (a) 要求进程在能够获取所需的所有资源前不能运行。进程运行期间不会再申请资源，从而破坏”请求”；在等待获取期间，进程不占有任何资源，从而破坏”保持”
     (b) 允许进程获得运行所需的部分资源后就运行，运行过程中逐步释放，只有全部释放后才能请求新的资源
   - 方法一实现简单，但会造成资源浪费，也会导致饥饿现象；相对来讲，方法二更加优秀
4. **破坏循环等待条件**
   - 采用顺序资源分配法，给系统中各类资源编号，只允许进程按照编号递增的顺序请求资源
   - 一个进程只有在已经保持小编号资源时，才能申请更大编号资源，因此已持有大编号资源的进程不能再逆向申请小编号资源，因此不会产生循环等待条件
   - 缺点在于，编号时只能考虑大多数资源使用这些资源的顺序，实际使用顺序可能与编号不一致，从而导致编程困难和资源浪费

Deadlock Avoidance 死锁避免： 同样属于事先预防策略。对于每次资源申请操作，只有在不产生死锁的情况下，系统才会为其分配资源。

当系统能够按照某种顺序为每个进程分配其所需的资源，并且不会导致死锁，此时该顺序就被称为**安全序列**（可能有多个）。若系统能找到这么一个安全序列，则此时为 Safe State；若不能，则称系统处于 Unsafe State。

银行家算法是最著名的死锁避免算法。该算法将操作系统视为银行家，管理的资源视为资金，进程请求资源相当于向银行家贷款。进程运行前需要声明对各种资源的最大需求量，其数量不超过系统的资源总量。

```
// 当进程 P_i 发出资源请求 Request[i]:
if Request[i] > Need[i]:
    error;                  // 所需资源超出宣布的最大值
if Request[i] > Available:
    P_i Wait;               // 尚无足够资源，等待

// 全部条件满足，试探性分配资源并更新数据结构
Available       -= Request[i];
Allocation[i]   += Request[i];
Need[i]         -= Request[i];

// 执行 Safety Algorithm，若通过才正式分配资源给 P_i
if Safety():
    // 分配后为 Safe State，正式分配
    grant request;
else:
    // 回滚
    Available      += Request[i]
    Allocation[i] -= Request[i]
    Need[i]        += Request[i]
    P_i Wait;
```

```
bool Safety() {
    Work = Available;
    Finish[n] = {0};    // 初始化为 0
    while exists p such that
            Finish[p] == 0              // 该进程还未运行结束
            && Need[p] <= Work:         // 该进程所需资源小于可用资源
        Work = Work + Allocation[p]; // 假定该进程释放资源
        Finish[p] = 1;                  // 假定该进程运行结束

    if all(Finish == 1):
        return true;    // Safe State
    else:
        return false;   // Unsafe State
}
```

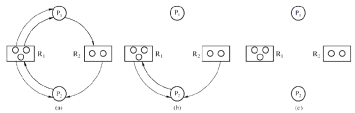
实际上，各个进程进入向量 Finish 的顺序即为一个可行的安全序列，若运行结束后安全序列中包含了所有进程（即所有均为 1），此时系统就处于安全状态；否则为不安全状态。

银行家算法虽然理论上很完美，但现实操作系统几乎不采用（包括 Windows、Linux、Unix）。现代操作系统通常采用**死锁忽略**（即假设死锁不会发生）或简单的**死锁预防**，极少采用复杂的死锁避免机制

- 它要求进程运行前必须**声明所需资源最大数量**，这在实际中很难预测
- 算法开销较大
- 资源的种类和数量在现代系统中是动态变化的

Deadlock Detection 死锁检测：

在资源分配图中，我们用圆圈表示一个进程，用框表示一类资源，框内每个圆表示该类资源中的一个资源。从进程到资源的有向边称为请求边，表示该进程申请一个单位的该类资源；从资源到进程的有向边称为分配边，表示该类资源已有一个资源分配给了该进程。



我们可以通过判断资源分配图是否**不可完全简化**，来检测当前系统状态是否为死锁状态：

1. 找到所有既不阻塞也不孤立的进程释放，消去该进程的所有有向边
2. 第一步中释放的资源可能可以唤醒某些因等待这些资源而被阻塞的进程，重复步骤一进行简化

假定每个资源类型只有一个实例，那么将该算法化为等待图进行环判断，时间复杂度为 $O(n^2)$，其中 n 为进程数。

对于多实例的资源类型，使用类似银行家算法的方式，时间复杂度为 $O(m*n^2)$，其中 m 为资源类型数，n 为进程数。

一旦检测出死锁，就可以通过某些措施来解除死锁：

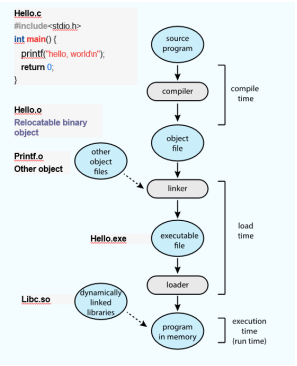
1. **资源剥夺法：** 挂起某些死锁进程，并抢占它的资源，将这些资源分配给其它死锁进程。简化后的资源分配图中，还有边相连的那些进程就是死锁进程。
   - 选择 victim 时，一般按照最小成本确定抢占顺序
   - 被抢占的进程需要回滚到安全状态，但是实现很难，通常直接终止进程重新执行
   - 保证该进程资源不会一致被抢占，为其成本加上回滚次数
2. **撤销进程法：** 强制撤销部分甚至全部死锁进程，并剥夺这些进程的资源。该方法实现简单，但可能会付出较大的代价。
3. **进程回退法：** 让一个或多个死锁进程回到足以回避死锁的状态，进程回退时释放资源。该方法需要系统保持进程的历史信息，并设置还原点。

7. 内存管理 (Main Memory)

主要功能：

1. **内存空间的分配与回收**：由操作系统负责内存空间的分配与管理，记录内存的空闲空间、内存的分配情况，并回收已结束进程所占用的内存空间。
2. **地址转换**：程序的逻辑地址与内存中的物理地址需要转换。
3. **内存空间扩充**：利用虚拟存储技术从逻辑上扩充内存。
4. **内存共享**：允许多个进程访问内存的同一部分，并且对内存共享区域进行受控访问。
5. **存储保护**：保证各个进程在各自的存储区域运行，互不干扰。

Link & Load： 一个用户源程序要变为在内存中执行的程序，通常需要以下几个步骤：



当我们写好一个程序时，程序中变量、函数、指令等都只是个“符号名”，没有真实的物理地址。

为了让这些符号名绑定到真正的内存地址上，我们有如下三种地址绑定的时机：

1. **编译时 (Compile Time)：** 绝对装入
   - 程序被加载到的地址编译时已确定，可以直接产生绝对地址代码
   - 仅适用于单道程序环境，要求程序逻辑地址和物理地址完全相同
   - 程序起始地址改变则需重新编译
2. **装入时 (Load Time)：** 静态重定位
   - 程序中指令和数据的地址都是相对于起始地址的逻辑地址
   - 装入过程中将目标程序中的相对地址修改为实际的绝对地址
   - 加载完成后，程序的内存位置即固定，不能再移动
3. **执行时 (Execution Time)：** 动态重定位
   - 将相对地址的计算推迟到执行时，需要 base & limit register 支持
   - 装入内存后的所有地址均为相对地址
   - 程序运行期间可在内存中移动，可分配到不连续的存储区
   - 程序运行时只需装入部分代码即可运行，运行期间可动态申请分配内存

逻辑地址和物理地址在 Compile-time 和 Load-time 是等价的，在 Execution-time 不等价。

在执行链接时，根据链接的时机不同，我们也可分出以下三种链接方式：

静态链接 Static Linking

- 运行之前就将程序及所需的库函数链接成一个完整的模块
- 将几个目标模块装配成一个装入模块时，需要将每个模块的外部调用符号转换为相对地址
- 由于编译后的单个目标模块的相对地址都是从自身 0 开始，静态链接后需要修改这些相对地址

装入时 Dynamic Linking at Load Time

- 在装入内存时，边装入边链接
- 优点是便于修改和更新，便于实现对目标模块的共享

运行时 Dynamic Linking at Execution Time

- 当程序执行时需要某动态模块时，才对它进行链接
- 优点是加快模块装入过程，节省内存空间
- 可执行文件体积更小，且当库更新时不需要重新链接可执行库

加载重定位寄存器和界地址寄存器必须使用特权指令，即只有操作系统内核可以加载。

动态加载 Dynamic Loading：

进程的大小会受到物理内存的限制，为了获得更高的内存空间利用率，例程仅在被调用时才会被加载；所有的例程都以可重定位的加载格式被保存在磁盘中。对于错误处理代码等不常用但大量的代码，该方式可以节省大量内存空间。

不需要 OS 特殊支持，由开发者负责实现。

交换技术 Swapping：

进程（或一部分进程）可从内存临时被交换（swap）到后备存储（backing store）中，之后可再次回到内存并继续执行。

在 high level，我们并不知道进程是驻留在内存中还是后备存储中，需要使用时，自动将其调入内存。交换时间主要为 Transfer Time，与所交换的内存大小成正比。

存储管理方式：

在操作系统由单道向多道发展时，存储管理方式的变化为**单一连续分配 => 固定分区分配 => 动态分区分配**。为了更好提高内存的利用率，最后又从连续分配方式发展到**离散分配方式**——页式存储管理。

Contiguous Allocation： 为一个用户程序分配一个连续的内存空间。内存被划分为系统区和用户区，**系统区位于低地址**，而用户区位于高地址。

1. **单一连续分配 Single Contiguous Allocation**
   - 用户区内存中有且仅有一个用户程序，系统区供操作系统使用
   - 实现简单、无外部碎片、不需要进行内存保护
   - 有内部碎片，且内存利用率极低
2. **固定分区分配 Multiple-partition Allocation**
   - 是最简单的多道程序存储管理方式
   - 将用户区划分为若干固定大小的分区(hole)，每个分区只装入一个作业
   - 分区的大小可以不等，通过分区使用表进行维护
   - 不存在外部碎片，但程序小于分区大小时产生**内部碎片**
3. **动态分区分配 Dynamic Partition Allocation**
   - 亦称可变分区分配
   - 分区的大小和数量是可变的
   - 通过一定算法使得分配的分区大小正好适合进程的需要
   - 随着时间的推移，动态分区分配方式难免会产生外部碎片

外部碎片问题可以通过周期性的紧凑技术来克服，即对内存碎片进行整理。

动态分区分配方式使用**空闲分区链**来维护分区。当进程要进入主存时，需要按照一定的分配算法从空闲分区链中选出一个分区。

按照分区检索方式，可分为顺序分配方法和索引分配方法：

- **顺序分配方法：** First-Fit > Next-Fit ≈ Best-Fit > Worst-Fit
- **索引分配方法：** 快速适应算法，伙伴系统，哈希算法等

Segment 分段管理：

- **Segment-table base register, STBR** 指向 Segment Table 的物理起始地址
- **Segment-table length register, STLR** 表示一个程序的分段数量

有外部碎片，无内部碎片

Paging 分页管理：

分页思想将物理内存空间划分为若干固定大小（通常为 4KB）的分区，称为 **Frame**，将逻辑内存空间划分为同样固定大小的分区，称为 **Page**。

分页管理**不会产生外部碎片**，而内部碎片也只会在系统为进程申请的最后一个分区中出现，每个进程平均只会产生半个块大小的内部碎片。

- **PTBR** 指向页表，切换页表只需要改变该寄存器即可
- **PTLR** 表示页表长度，这些信息存放在 PCB 中，切换线程时也需要切换该寄存器

每次访问 VA 都需要两次内存访问，一次查页表，一次查数据/指令。为了避免查页表，引入了转换表缓冲区 **TLB**。

TLB Reach 指通过 TLB 能够访问到的内存量，等于 $TLBSize * PageSize$。增加 TLB Reach 能够显著提高系统性能。

部分 TLB 维护了 ASID (Address Space Identifier)，用来唯一标识进程，为进程提供了空间保护，否则每次进程切换都需要清空 TLB。

Structure of Page Table:

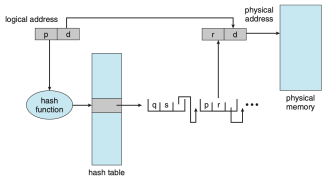
分级页表

略

哈希页表 Hashed Page Table

将虚页号的哈希值存到哈希表中，**哈希表的每一项都是链表，链着哈希值相同的页号**，链表中元素包含三个字段：虚拟页编号、映射页帧的值、指向链表中下一个元素的指针。

查表时匹配虚拟页标号，将第二个字段用于和 offset 组成实际物理地址：

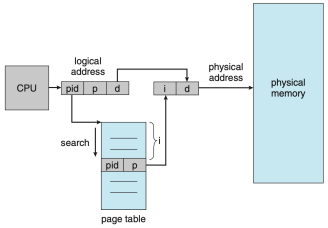


反向页表 Inverted Page Table

整个物理内存使用一张页表，每个 physical frame 对应页表中的一个表项，表项中记录该 frame 中存放的虚拟页号和该页所属的进程标识符 PID。

典型的**拿时间换空间**，为了缓解查找慢，通常使用一个哈希表将搜索限制在一个或少数页表项上。

另外，当共享内存时不能使用该方法，因为一个物理页无法对应多个虚拟地址。



8. 虚拟内存 (Virtual Memory)

Demand Paging 按需调页：

通过 Demand Paging，操作系统就好像为用户提供了一个比实际内存容量大得多的存储器，这就是 Virtual Memory。

相比于基本分页系统，请求分页系统需要知道每个 Page 是否已调入内存；若未调入，还需要知道该页在外存中的存放地址。为了实现页面置换功能，操作系统还需要一些额外信息和算法来决定换出哪个页面，以及被换出的页面是否被修改过。

因此，一个 Page Entry 至少包含如下几个字段：

页号	物理块号	状态位 P	访问字段 A	修改位 M	外存地址

请求分页系统中的页表项

在请求分页系统中，每当要访问的页面不在内存中时，便会产生一个 Page Fault，其处理流程为：

1. 检查 PCB 判断访问是否合法。如果访问地址不属于该进程，则为非法访问，终止进程
2. 如果访问合法，但 Page 不在内存，产生 Page Fault 并准备调入
3. 通过一定算法选择一个空闲 Frame
4. 从外存中将所需页面读入该 Frame
5. 更新 Page Table Entry 和 PCB
6. 重新执行刚刚因 Page Fault 而中断的指令

一条指令在执行期间可能产生多次缺页中断，Page Fault Rate = 1 也不代表所有 Page 都是 Page Fault。

Copy-on-Write 写时复制：

fork() 时子进程共享父进程的 Page，只复制父进程的页表；当子进程或父进程要向某一 Page 写入时，MMU 检测到该页是只读的 COW 页，则会先创建该共享页的 Copy，然后仅写入该拷贝页，并修改对应的页表项。

Frame Allocation 帧分配：

OS 通常使用链表维护所有空闲的帧。为了将有限的物理 Frames 分配给所有进程，可采用以下几种算法：

1. **平均分配算法**，将系统中所有可供分配的物理块平均分配给各个进程。
2. **按比例分配算法**，根据进程的大小按比例分配物理块。
3. **优先权分配算法**，为重要和紧迫的进程分配较多的物理块。通常采取的方法是将所有可分配的物理块分成两部分：一部分按比例分配给各个进程；一部分则根据优先权分配。

Prepaging 预调页

在 pure demand paging 中，为了减少冷启动时大量的 Page Fault，会在刚开始时预先分配一些页。

假设 s 个页被预调到内存中，其中 a 比例个页被用到。问题在于节省的 s*a 个 Page Fault 成本是否大于预调 s*(1-a) 个未被使用页的成本。

如果 a 接近于 0，则调页失败；如果 a 接近于 1，则预调页成功。

Page Replacement 页面置换算法：

当 Free-Frame List 为空，但用户仍然需要 Frame 来 Page In 时，就需要进行页面置换。

1. **Global Allocation**：允许一个进程从所有帧中选择一个帧进行替换，无论该帧是否已分配给其它进程
2. **Local Allocation**：仅允许一个进程从分配给它的帧中选择一个帧进行替换

FIFO

实现简单，但没有利用局部性原理，性能较差。

除此之外，FIFO 算法还有可能因为系统为该进程分配的物理块增多而出现 Page Fault 次数不减反增的异常现象，这被称为 **Belady 异常**。

LRU

LRU 算法优先替换最近最长时间未使用的页面，该算法为每个 Page 维护一个上次访问的时间戳，并淘汰现有页面中值最大的 Page。

LRU 获取多长时间没引用有两种方法：

1. **计数器**：每个页表项都有一个 Counter，每次被引用，就把时钟信息复制到 Counter；置换时，选择时间最小的页
2. **栈实现**：维护一个双向链表页码栈，引用页面时将其移动到栈顶，需要改变 6 个指针；替换时直接替换栈底部的页

根据局部性原理，LRU 算法的性能较好，但实现起来需要寄存器和栈的硬件支持，硬件开销大。

Additional-Reference Bits

通过定期记录引用位来获得额外的页信息。每个页保留一个字节，每一个时钟间隔，时钟产生中断，OS 把每个页的引用位移动到该字节的最高位，其他位右移一位。这样，每个页都有一个 8 位的引用历史。

全 0 说明该页在最近 8 个时钟周期内没有使用过；全 1 说明每个周期内都至少用过一次。值越大说明越最近使用过，因此选择值最小的页替换。

CLOCK

CLOCK 算法及其变体尝试用更小的开销来接近 LRU 算法的性能，它也被称为 **Second-Chance 置换算法**。

最简单的 CLOCK 置换算法将内存中的 Page 链接成循环队列，当 Page 首次装入或被访问时，其 Reference Bit 置 1。当需要替换一个 Page 时，一个替换指针会顺序检查循环队列中的 Page，如果遇到 Ref = 0 的 Page，则将其替换；如果遇到 Ref = 1 的 Page，则将其置 0，给予第二次驻留内存的机会。

Enhanced CLOCK 算法 使用访问位 A 和修改位 M 一起作为替换的依据，根据二者组合，总共有如下四种类型的 Page：

- **(A=0, M=0)**：最近未被访问，且未被修改，是最佳的淘汰页
- **(A=0, M=1)**：最近未被访问，但已被修改，是次佳的淘汰页
- **(A=1, M=0)**：最近已被访问，但未被修改，可能再次访问
- **(A=1, M=1)**：最近已被访问，且已被修改，可能再次访问

此时，替换指针扫描总共分为三步：

1. 从指针当前位置开始扫描一轮，寻找 **(A=0, M=0)** 的 Page 并选中。第一次扫描期间不改变 A 位
   - 第一次扫描期间不改变 A 位
   - 有可能找不到符合要求的 Page
2. 若第一步失败，开始第二轮扫描，寻找 **(A=0, M=1)** 的 Page 并选中；扫描期间，将所有扫描到的 Page 的 A 位置 0
   - 此时也有可能失败，即存在所有 Page 的 A 均为 1 的情况
3. 若第二步也失败，则重复 *1* 和 *2*，此时一定能找到被淘汰的 Page

Optimal

OPT 置换算法选择淘汰**以后永不使用或最长时间内不会再被访问**的 Page，保证获得最低的缺页率。然后，进程中哪个页面是未来最长时间内不会被访问的是无法预测的，因此该算法无法实现，但可利用该算法去评价其它算法。

在题目中，替换引用序列中下一次该页号出现位置最远的页即可。

LRU 算法是最接近 OPT 算法的置换算法。

Counting-Based Algorithms

为每个页保存一个用于记录引用次数的计数器，具体有两种方案：

1. **LFU (Least-Frequently Used)**：选择引用次数最少的页面进行置换，并且会定期右移计数器，防止老页面长期驻留内存
2. **MFU (Most-Frequently Used)**：选择引用次数最多的页面进行置换，假设最近被引用的页面在未来也会被频繁引用

两种都很少使用，LFU 需要频繁更新计数器，MFU 则违背了局部性原理。

Thrashing 抖动：

一个进程在频繁地进行 Page 换出换进，这一现象称为 **Thrashing**。

抖动会导致 CPU 利用率很低，然后 OS 认为 CPU 太闲了，增加多道程序程度，结果进一步引发抖动。

系统发生 Thrashing 的根本原因是分配给每个进程的物理块太少，不能满足进程正常运行的基本要求，导致每个进程运行时频繁缺页。

1. 为了限制抖动，我们可以要求 Frame 替换算法不能对其它进程的 Frames 进行替换
   - 或者使用 Priority Replacement Algorithm，只允许高优先级的进程替换低优先级进程的 Frame
2. 进一步，为了从根本上预防抖动，我们要为每个进程分配足够多的物理块，以满足该进程的 Locality

为此，我们引入进程**工作集 Working Set**的概念。工作集是指在某段时间间隔内，进程要访问的 Page 集合。一般来讲，工作集 W 可通过时间 t 和工作集窗口尺寸 Δ 来确定：

Working Set 工作集：

工作集是对 Locality 的近似表示，反映了进程在接下来的一段时间内可能频繁访问的 Page 集合，因此该进程被分配的物理块个数不能小于工作集大小，否则会在运行过程中频繁缺页。

- WSS_i：进程 i 在窗口 Δ 内实际访问的**不同页面 (Unique Pages)** 的数量，即工作集大小。这就是该进程当前必须占用的物理帧数。
- D：Total Demand = $\sum WSS_i$，即所有进程工作集大小之和。
- m：Total Memory Size，系统中可用的物理帧总数。
- 如果 $D > m$，则系统就会发生抖动现象。

工作集的估计

为了近似估计工作集，我们可以设置 1 bit 引用位和一个定时器：

1. **硬件支持**：每个页表项有一个引用位。CPU 访问该页时，硬件自动把该位置 1。
2. **定时采样**：
   - 假设 $\Delta = 10000$ 次引用。
   - 设置定时器，每隔 5000 次引用触发一次中断。
3. **中断处理**：
   - 当定时器响了，OS 醒来，把当前的硬件引用位的值，拷贝到内存里的变量中（作为历史记录），然后**把硬件引用位清零**。
4. **判断逻辑**：
   - 如果发生缺页，OS 检查过去几次中断记录下来的位。
   - 只要在最近几个周期里（代表 Δ 时间段，上例为两个周期），某个页的位曾经是 1，它就在工作集里。
   - 如果所有记录位都是 0，说明它很久没用了，可以被踢出工作集。

Page-Fault Frequency 页面错误频率：

WS 模型能用于 pre-paging，但是控制抖动的灵活性上不如 PFF。

我们为期望的页错误率设定上下界，若页错误率超出上界，则为进程分配更多的帧；若页错误率低于下界，则要移除一些该进程拥有的帧。

如果页错误率太高，且无可用帧，则选择一些进程 Swap 到后备存储中。

Allocate Kernel Memory 分配内核内存：

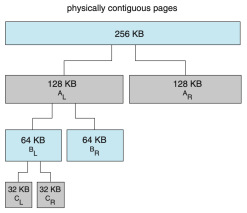
内核可能会请求不同大小的数据结构内存，有时大小可能不足单个页。因此，内核必须谨慎使用内存，并尝试最小化由于碎片造成的浪费。这十分重要，因为许多操作系统不对内核代码或数据进行分页系统处理。

Buddy System 伙伴系统

由物理连续页组成的固定大小段中分配内存。使用 2 的幂次方分配器 (power-of-2 allocator) 从这个段中分配内存，该分配器以 2 的幂次方大小（4 KB、8 KB、16 KB 等等）为单位满足请求。对于更大内存的请求，它将向上取整到下一个最高的 2 的幂次方。

优点：可以通过合并快速形成更大的段

缺点：内部碎片



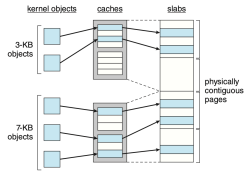
Slab Allocation 板块分配

预先了解到 Kernel 中常见数据结构（称为 Object）的大小，并预先准备好对应粒度的小内存块，注册到每类 Object 的 Cache 中。

当一个 Object 需要内存时，首先查询对应 Cache 中是否有空闲的内存块，若有则直接分配；若没有，则从 Buddy System 中申请一页或多页内存，划分成多个该类 Object 大小的内存块，放入 Cache 中备用。

优点：避免内部碎片引起的内存浪费；内存请求可以快速满足

缺点：需要维护多个 Cache，增加了管理开销



是为了解决 **Inefficiency and internal fragmentation when allocating many small, fixed-size kernel objects.**

9. 文件系统 (File System)

File Basic：

除了文件本身数据外，大多数操作系统都会保存如下与文件相关的属性：

- **Name**：unique，以容易读取的形式保存
- **Type**：被支持不同类型的文件系统所使用
- **Creater**：文件创建者的 ID
- **Owner**：文件当前所有者的 ID
- **Location**：指向设备上文件的指针
- **Size**：文件当前大小，也可包含文件允许的最大值
- **Protection**：文件访问控制相关信息
- **Time**：包含文件创建、上次修改、上次访问的时间，用于保护和跟踪文件的使用

系统会为打开的文件维护一个 Open-File Table，所谓打开就是系统在检索到指定文件的 Dentry 后，将该目录项从外存复制到内存的 Open-File Table 中，并将该表项的文件描述符返回给用户。当用户再次对该文件发出操作请求时，可通过文件描述符在表中查找到文件信息，从而节省检索开销。

多进程系统使用进程表和系统表两级

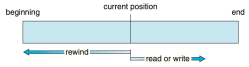
- **System-wide Table**：包含与进程无关的信息，如 Location、Time、Size。
- **Per-process Table**：包含进程对文件的使用信息，如读/写指针，Access Rights，以及指向系统表中对应条目的指针。
- 当一个进程执行 open 时，会为其进程表中增加一个条目，并指向系统表中对应该文件的条目。
- 系统表会为每个文件维护一个 Open Count，以记录多少进程打开了该文件；当 Open Count 为 0 时，即可从系统表中删除该条目。

文件名只有一开始检索有用，之后用的都是文件描述符来访问 Open-File Table 中的表项。实际上，打开文件表中也不会保存文件名。

Access Methods 访问方法：

Sequential Access 顺序访问

最简单最常用，文本编辑器使用的方式。No read after last write，即不能读还没有写入任何数据的位置，于 direct access 区分。



Direct Access 直接访问

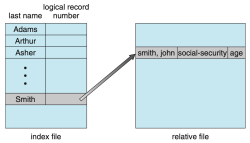
文件由固定长度的 record 组成，允许直接读写文件中的任何位置，适用于数据库系统等。

sequential access	implementation for direct access
reset	cp = 0;
read_next	read cp; cp = cp + 1;
write_next	write cp; cp = cp + 1;

单看**吞吐量**，顺序读写性能更好；看响应速度，则随机读写性能才是关键。

Indexed Block Access 索引顺序访问

顺序访问和直接访问的结合。文件中数据按顺序存储，但同时维护一个索引用于快速定位特定记录。



Directory:

为了便于文件管理，我们引入 File Control Block 存放控制文件所需的各种信息。文件一一与 FCB 对应，FCB 的有序结合称为 Directory。

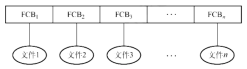
Windows 中，一个 FCB 就是一个 Dentry，因此 FCB 中会保存文件名；而在 Linux 中，FCB 是 inode，而 Directory 会保存 <filename, FCB> 的映射关系，因此 FCB 中不保存文件名。

但总的来说，FCB 中包含如下信息：

- **基本信息：** 文件物理位置、文件物理结构、文件逻辑结构等
- **存取控制信息：** 各类用户的存取权限
- **使用信息：** 文件建立时间、上次修改时间等

目录操作： 搜索文件；创建文件；删除文件；遍历目录（list）；重命名文件；遍历文件系统（traverse）。

Single-Level Directory



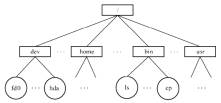
当建立新文件时，必须检索所有目录项，以确保没有“重名”的情况，因此单级目录结构存在查找速度慢、文件不允许重名、不便于文件共享等缺点。

Two-Level Directory

分为 Master File Directory 和 User File Directory 两级：

提高了检索的速度，解决了多用户之间的文件重名问题，但是缺乏灵活性，没有分组能力。

Tree-Structured Directory

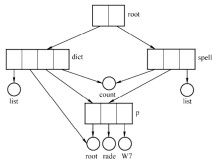


树形目录结构可以很方便地对文件进行分类，层次结构清晰，对文件的管理和保护更有效。

但是，在树形目录中查找一个文件，需要按路径逐级访问中间节点，也会影响查询速度。目前大多数操作系统都采用了该结构。

In order to solve name collision, the file system normally adopts tree-like directory structures

Acyclic-Graph Directory



树形目录结构虽然实现了文件分类，但不便于实现文件共享。为此，我们在树形基础上增加了一些指向同一节点的有向边，使得整个目录成为一个**有向无环图**。这种结构允许目录共享子目录或文件，同一文件或子目录可以出现在两个或多个目录中。

1. **软链接**，又称符号链接，是一个指向文件的指针，类似于快捷方式。
   - 删除被软链接指向的那个文件，并不会一起删除软链接文件，但软链接会失效
   - 从本质上看，软链接是特殊的独立文件
   - 对应命令 `ln -s <target> <linkname>`
2. **硬链接**，复制链接文件目录项中所有的元信息，存到目标目录中，此时文件平等属于两个目录
   - 文件元信息更新时保证一致性
   - 删除被硬链接指向的那个文件，不会删除硬链接文件，文件内容依然存在，只有当所有指向该文件的硬链接都被删除后，文件内容才会被删除（reference count=0）
   - 从本质上看，硬链接是目录表项
   - 对应命令 `ln <target> <linkname>`

General Graph Directory

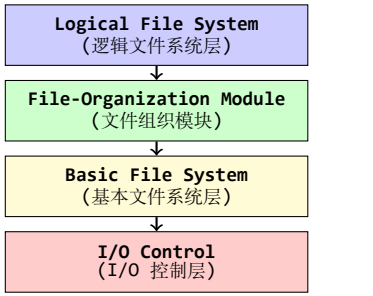
如果允许目录结构存在环（cycles）的话，那么原来的无环图结构就进一步泛化为通用图（general graph）结构。

在各种操作时，通过算法来避免出现问题。

File System: 需要解决两个问题：

- 定义用户接口，包括文件属性、文件操作、目录结构等
- 定义算法和数据结构，以将逻辑文件系统映射到外存

一个好的文件系统应具有如下层次结构：



逻辑文件系统 Logical File System（最上层）

- 管理元数据（Metadata）：存储和管理关于文件结构的信息，但不包含文件的实际内容
- 检查用户是否有权限打开文件、解析文件路径等

文件组织模块 File-Organization Module

- **地址映射（Logical to Physical Mapping）**：将逻辑块号转换为物理地址（柱面、磁头、扇区）
- **空闲空间管理（Free-Space Management）**：维护未被分配的磁盘块清单

基本文件系统 Basic File System

- 发送**抽象指令**：如”读取第 N 块数据”，不涉及具体硬件细节
- **缓冲区与缓存管理**：
  - Buffer（缓冲区）：内存和硬盘间数据传输的临时存储
  - Caches（缓存）：存储频繁使用的文件系统元数据

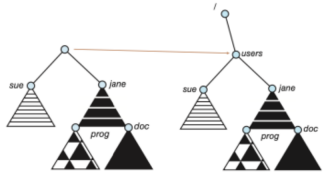
I/O 控制层 I/O Control（最底层）

- **设备驱动程序（Device Drivers）**：将 OS 通用指令翻译成特定硬件能执行的电子信号
- **中断处理程序（Interrupt Handlers）**：处理硬件完成 I/O 操作后发送的中断信号
- 直接控制具体的 I/O 设备（硬盘、SSD 等）

Mount 文件系统挂载：

挂载将一个存储设备中的文件系统根目录，挂载到另一个文件系统的某个目录下（mount point）。

文件系统在访问前必须挂载。



左图时未安装的卷，右图将其挂载到 users 上。

传统 Linux 设备命名中，/dev/sda 代表第一个 SCSI 硬盘，/dev/sda1 代表该硬盘的第一个分区。/dev/hda 代表 IDE0 主盘；/dev/hdb 代表 IDE0 从盘；/dev/hdc 代表 IDE1 主盘；/dev/hdd 代表 IDE1 主盘。

/dev/hd0s2 是 Solaris/BSD 风格。

文件共享和权限保护：

多用户系统的文件共享很有用，需要一定保护机制实现。在分布式系统，文件通过网络访问；网络文件系统 NFS 时常见的分布式文件共享方法。

文件权限分为：读、写、执行、追加（append）、删除、List

访问控制列表 ACL 有三种用户类型：Owner、Group、Public。UNIX 中一个类型有 rwx 三种权限，因此共需要 9 bits 来表示权限。

FS Structures:

File System on Disk 要求持久化存储。磁盘被划分为多个分区（卷），每个分区中有一个独立的文件系统：

- **Boot Control Block:** 引导块（Windows 的分区引导记录，Unix 的 Boot Block）
  - 通常是卷中第一个块，包含 OS 启动所需的信息
- **Volume Control Block:** 卷控制块。在 Unix/Linux 中称为 **Superblock（超级块）**
  - 记录分区的总数量、块大小、空闲块数量和指针、空闲的 FCB 数量和指针等信息
- **FCB（File Control Block):** 在 Unix/Linux 中对应 **inode**。它包含文件的所有元数据（权限、所有者、时间戳、大小、数据块指针），唯独**不包含文件名**（文件名在目录里）

File System in Memory 内存中信息用于管理文件系统并通过缓存提高性能，在 mount 时加载：

- **Mount Table:** 记录挂载点信息
- **Directory Structure Cache (dentry cache):** 保存最近访问过的目录信息，加速路径查找，避免频繁读盘
- **Open-file Tables（打开文件表）:**
  - **System-wide（系统级）:** 整个 OS 只有一张。记录打开文件的 inode 信息和引用计数
  - **Per-process（进程级）:** 每个进程 PCB 中有一个。记录当前读写指针（Offset）和指向系统表的指针
- **Buffers:** 用于缓冲 disk block 的内容，读写文件都会经过 buffer，从而统一 CPU 和 I/O 设备之间的速度差异

一个**open()**操作，需要经过：

1. 用 filename 在 FS 中的目录结构查找该文件；找到后，其 FCB 被复制进 System-wide Open-file Table
2. 在当前进程的 Per-process Open-file Table 中创建一个新条目，包含读写指针和指向 System-wide Table 中该文件条目的指针，同时该 entry 的 reference count 加 1
3. 之后，对文件的所有操作依赖该指针，该指针被称为 file descriptor 或 file handle(in Windows)

VFS:

虚拟文件系统提供了一个面向对象的抽象层接口 API，上层应用可以只使用 write() 系统调用，而不用关心其底层是 ext4、NTFS 还是 NFS。

- **Superblock object:** 代表一个已挂载的文件系统，对应磁盘分区的超级块
- **Inode object:** 代表一个具体文件，对应（但不是）FCB
  - 只有文件被访问时，才在内存中创建 Inode Object
- **Dentry object (Directory Entry):** 代表路径中的一项（如‘/home/user‘ 中的‘home‘、‘user‘ 都是 dentry）
  - Dentry 是内存对象，在磁盘上没有对应的数据结构
  - 包含指向关联 Inode Object 的指针，也包含指向父目录和子目录的指针
- **File object:** 代表一个**被进程打开的文件**。它包含当前的读写位置（offset）
  - 因为不同进程可以打开同一个文件且读写位置不同，所以 File Object 属于进程，而 Inode Object 属于文件系统唯一

VFS 只存在于内存中，它在系统启动时建立，在系统关闭时消亡

Blocks Allocation: 磁盘将 blocks 分配给文件有多种不同的算法，算法也决定了文件系统的性能和空间利用率。

连续分配 Contiguous Allocation

占用磁盘上一组连续的块，类似数组。只需要记录 Start Block 和 Length。Fisrt-Fit 和 Best-Fit 表现差不多，但是 First-Fit 时间快很多。

优点： 读写速度快，支持顺序和随机访问。

缺点： 存在外部碎片，文件很难动态增长（需要 realloc）。

变体：Extent-Based Allocation

Extent 是一组连续块的集合。每个文件由多个 Extent 组成，每个 Extent 记录 Start Block 和 Length，长度可以不等。Extent 之间可以不连续，由指针链接，从而解决了文件大小无法增长的问题。

链接分配 Linked Allocation

每个文件都是一组 Blocks 组成的链表，blocks 不必是连续的。目录中需要记录文件的 Start Block，每个 Block 包含一个指向下一个 Block 的指针。

一个指针长 4B，因此用户实际可用空间小于一个块大小。

优点： 无外部碎片，文件可动态增长。

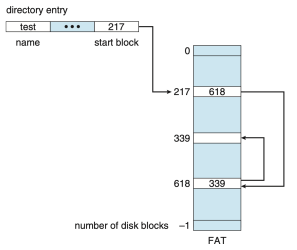
缺点： 只能顺序访问，指针开销大，可靠性差（指针损坏导致文件丢失）。

也可以将多个连续块合并为一个簇（cluster），链表链接簇，从而减少指针开销。

变体：FAT 文件系统

File Allocation Table 将所有指针提取出来放在内存的一张表中，这样在内存中查表就能实现较快的随机访问。

例如一个文件 test 占据了块 217, 618, 339，对应的 FAT 表项为：



索引分配 Indexed Allocation

每个文件有一个索引块（Index Block），里面存了所有数据块的地址数组。

优点： 支持直接随机访问，无外部碎片，文件可动态增长

缺点： 索引块本身有开销。如果文件很小，一个索引块浪费空间；如果文件很大，一个索引块存不下指针

变体：多级索引（Multi-level Indexing）

为了支持大文件，索引块可以设计成多级的。考虑块大小 4KB，块地址 4B：

- **1-Indirect:** 索引块直接存数据块指针，存 4KB/4B = 1K 个索引地址，因此最大文件大小为 1K*4KB=4MB
- **2-Indirect:** 索引块存一级索引块指针，一级索引块存数据块指针，最大文件为 1K * 1K * 4KB = 4GB
- **3-Indirect:** 索引块存二级索引块指针，二级索引块存一级索引块指针，一级索引块存数据块指针
- **4-Indirect:** 以此类推

In an i-node based file system implementation, the i-node typically stores 12 direct block pointers, one 1-indirect block pointer, one 2-indirect block pointer, and one 3-indirect block pointer. Suppose block size of 2^{10} bytes and each pointer takes up 4-byte. What is the maximum file size that can be supported in the file system (approximately)?

$12*2^{10}+2^8*2^{10}+2^{16}*2^{10}+2^{24}*2^{10} \approx 16\text{GB}$

Free Space Management: OS 通过如下三种方式来进行 Free Space 的管理：

位图 Bit Vector / Map

1 bit 表示 1 个块的状态。0 表示空闲，1 表示已分配。查找空闲块时，从位图中顺序扫描 0 位。

优点： 实现简单，容易找到连续空闲块

缺点： 对于大磁盘，位图本身很大，需要占用较多内存

链表 Linked List

将所有空闲块链接成一个链表，链表头指针保存在超级块中。每个空闲块包含一个指向下一个空闲块的指针。

优点： 实现简单，节省空间

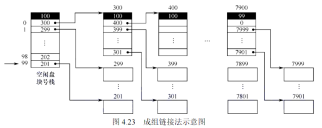
缺点： 查找空闲块时需要遍历链表，速度较慢；难以找到连续空间

成组链接 Grouping

在链表的基础上进行改进。超级块中保存一个指向空闲块链表头的指针，以及一个计数器，表示链表中空闲块的数量。

在第一个空闲块里存下 n 个空闲块的地址，每组的最后一个块又记录下一组的空闲盘块总数和空闲盘块号，构成一条链。

目前 UNIX 采用的方式，适合大型计算机。



计数 Counting

也称**空闲表法**；保存连续空闲块中，第一个空闲块地址以及连续空闲块数量 n。

Page Buffer 页缓冲：

通过虚拟内存技术，以页为单位进行缓冲，而非面向 FS 的 Block 为单位。

使用虚拟内存相关接口通常比使用文件系统相关接口更快。

Recovery 恢复：

一致性检查：将目录结构与磁盘数据块进行对比，并且纠正不一致的地方。

用系统程序将磁盘数据都被分到另一个设备，然后从该设备恢复。

日志文件系统 Log-Structured File System

将所有修改都作为 transaction 记录在日志中，而不是修改文件本身。定期将日志中的修改应用到文件系统中。

当系统崩溃时，可以通过日志回放来恢复文件系统到一致状态。

10. 磁盘调度 Mass-Storage

现代计算机的二级存储基本选用 Hard Disk Drives, HDDs 或 Non-Valatile Memory, NVM, 在这里我们主要关注 HDD 的结构与相关访问时间量化。

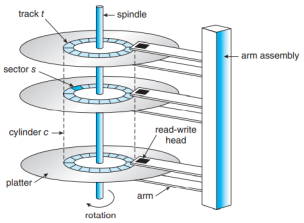


Figure 11.1 HDD moving-head disk mechanism.

其中 0 扇区是最外面的第一个磁道的第一个扇区

评价：

Access Time = Seek Time + Rotation Time + Transfer Time

Bandwidth = $\frac{\text{字节数}}{\text{Access Time}}$

- **Seek Time (寻道时间)**：读写头移动到目标磁道所需时间
  - 取决于当前磁头位置和目标磁道位置的距离
- **Rotation Time (旋转延迟)**：磁盘旋转使读写头到达目标扇区所需时间
  - 平均旋转延迟 = $\frac{1}{2\times RPM/60}$ 秒，即平均转过半圈
  - 最小为 0（磁头已在目标扇区），最大为一圈时间
- **Transfer Time (传输时间)**：从磁盘读出数据或写入数据所需时间
  - 计算方法：读写头转过的全部扇区时间
  - Transfer Time = $\frac{\text{字节数}}{\text{传输速率}}$ 或 $\frac{\text{扇区数}}{\text{每圈扇区数}}\times\frac{60}{RPM}$

其中 Seek Time + Rotation Time 称为 **Positioning Time (定位时间)**。

磁盘调度算法：

- **FCFS**：平等，不会无限期推迟；慢
- **SSTF**：最短寻道优先。平均响应时间短，吞吐量高；饥饿风险
- **SCAN**：电梯算法，来回扫描。平均响应时间短，吞吐量高，方差低；对于刚刚才访问过的位置，可能要经过很久才能再次访问
- **C-SCAN**：单向扫描。相比于 SCAN，提供更均匀的等待时间
- **C-LOOK**：C-SCAN 优化，不走到物理尽头，只到最远请求。

相对来讲 SSTF 较好；LOOK 和 C-LOOK 对于高负荷 IO 磁盘表现更好。

另外，由于 NVM（如 Solid-State Disks, SSD）并不使用物理读写头结构，并且支持随机访问，因此 NVM 的调度算法通常选用 FCFS。

磁盘管理：

1. **物理格式化**：在磁盘上划分磁道和扇区
2. **逻辑格式化**：在磁盘上建立文件系统

RAID：

关键指标：

- **MTTF (Mean Time To Failure)**：平均无故障时间，从参照点至出现故障经过时间的量度
- **MTTR (Mean Time To Repair)**：平均修复时间，故障修复所需的平均时间
- **MTBF (Mean Time Between Failures)**：平均故障间隔时间
  - $MTBF = MTTF + MTTR$
- **Dependability (可靠性)**：从参照点至出现故障经过时间的量度，通常通过 MTTF 作为量度
- **Availability (可用性)**：系统正常工作时间占总时间的比例
  - $Availability = \frac{MTTF}{MTTF+MTTR} = \frac{MTTF}{MTBF}$

RAID Levels

1. **RAID 0**：条带化，无冗余。故障容限 0，数据盘 8，校验盘 0。仅把数据分散到多个磁盘上
2. **RAID 1**：镜像，1:1 备份。故障容限 1，数据盘 8，校验盘 8。每一个盘对应一个校验盘，即镜像，空间利用率约 50%
3. **RAID 2**：内存式 ECC。故障容限 1，数据盘 8，校验盘 4。Hamming 编码方式，$\log_2 n+1$ 个校验盘
4. **RAID 3**：位交叉奇偶校验。故障容限 1，数据盘 8，校验盘 1。所有磁盘同时工作，不适合随机访问
5. **RAID 4**：块交叉奇偶校验。故障容限 1，数据盘 8，校验盘 1。块级条带化，校验盘成为瓶颈
6. **RAID 5**：分布式奇偶校验。故障容限 1，数据盘 8，校验盘 1。校验块分布于各个数据盘之中，性能最优
7. **RAID 6**：P+Q 双校验。故障容限 2，数据盘 8，校验盘 2。二次校验块技术，差错纠正码，需要多一倍校验盘

11. I/O System

Device Driver 提供了一种统一的面向 I/O 子系统的设备访问接口。

I/O 方式：

- 轮询
- 中断驱动
- DMA：在 DMA 开始传输时，主机向内存写入 DMA 命令块，CPU 在写入后就可以去干别的。DMA 只做大量的、以 Blocks 为单位的数据传输

I/O 分类：Block I/O(read, write, seek), Character I/O(stream, keyboard, clock), Memory-Mapped File Access, Network Sockets.

I/O 应用接口：

实现统一 IO 接口，设备驱动提供 API，如 ioctl。具体来说，不同设备在如下方面存在差异：

1. **设备速度**：快/中/慢
2. **I/O Direction**：读写，如 disk；只读，如 CD-ROM；只写，如 graphics controller
3. **数据传输模式**：逐字节传输，命令是 get, put，如 terminal；块传输，命令是 read, write, seek，如 disk
4. **访问方法**：顺序访问，如 modem；随机访问，如 CD-ROM
5. **传输方法**：同步，还分为阻塞和非阻塞（立刻返回尽可能多的数据，不管是否完成）；异步，如网络 IO
6. **共享**：可共享/独占

实验

文件描述符 stdin=0, stdout=1, stderr=2。

习题

线程上下文切换与进程上下文切换相比，主要节省的开销是？

不需要切换地址空间。

另外，还不用切换 Page Table 和 TLB。

The context-switch overhead depends on the following factors Except:

- A. The complexity of the OS and PCB
- **B. Number of Processes in Run Queue**
- C. Multiple sets of registers per CPU
- D. Multiple contexts loaded at once

以下哪种 IPC 方式不需要进入内核态？

A. 消息队列；**B. 共享内存**；C. 管道；D. 信号(signal)

答案选 B。虽然共享内存的创建与销毁需要内核态，但进程间通过共享内存进行通信时，并不需要进入内核态。

For the long-term scheduler, which statement is NOT true?

- A. It decides which process can enter the ready queue.
- **B. When it detects the CPU utilization is low, it will accept more processes into the ready queue.**
- C. It helps balance the parallelism and the performance.
- D. It can access the whole job queue.

假设文件初始为空，最终 test.txt 文件内容是什么？

```
fd = open("test.txt", ...);

write(fd, "hello", 5);

pid_t pid = fork();

if (pid == 0) {
    lseek(fd, 0, SEEK_SET);
    write(fd, "child", 5);
    lseek(fd, 0, SEEK_SET);
    read(fd, buf, 10);
    buf[10] = '\0';
    close(fd);
} esle if (pid > 0) {
    sleep(1);
    lseek(fd, 0, SEEK_SET);
    read(fd, buf, 10);
    buf[10] = '\0';
    write(fd, "parent", 6);
    close(fd);
}
```

fork() 之后，文件描述符 fd 被子进程复制一份，但是它们都指向了同一个打开文件表项，因此子进程和父进程共享同一个文件偏移量。

而 read 和 write 也是共享文件偏移量的，因此最后的"parent" 写入时是从偏移量 5 开始写入的。

read 会在读到 EOF 时停止，不会因为没有读满 10B 而报错或阻塞。因此程序中的两次 read 都读取 5B 的数据"child"，并且将偏移量设置为 5。最终 test.txt 文件内容为"childparent"。

Why is Deferred Cancellation generally the preferred and safest mode for most multithreaded applications?

- A. It requires less overhead because the kernel does not have to track the thread's state
- B. It eliminates the need for any synchronization primitives like mutexes or semaphores.
- **C. It guarantees that the thread will only terminate when it reaches a cancellation point (e.g., a known system call), allowing resources to be safely released**
- D. It is the only mode that allows the thread to continue executing its cleanup handlers after termination.

What is the defining characteristic of Asymmetric Multiprocessing (AMP) scheduling?

- A. Each processor has its own private operating system kernel and a completely separate memory space.
- B. All processors share a common memory space and a single Ready Queue, with the scheduler running on every core.
- C. Processors are divided into groups based on their execution speed, with faster cores handling system tasks and slower cores handling user tasks.
- **D. The operating system kernel runs entirely on one designated master processor, while all other processors are slaves that execute user code.**

下列选项中，降低进程优先级的合理时机是？

- **A. 进程的时间片用完**
- B. 进程刚完成 I/O，进入就绪队列
- C. 进程长期处于就绪队列中
- D. 进程从就绪态转为运行态

下列选项中，满足短任务优先且不会发生饥饿现象的调度算法是？

- A. FCFS
- **B. 高响应比优先**
- C. RR
- D. 非抢占式 SJF

The critical section of a concurrent process is ?

答案是 a segment of code。CS 是访问临界资源的代码，而不是资源本身。

另外，.data 段中保存的是已初始化全局变量和静态变量；.bss 段中保存的是未初始化的全局变量和静态变量，二者都是可写的。

Critical section can be enforced with ageneral semaphore whose initial value is greater than 1.

答案是 False。信号量大于 1 意味着允许多个进程进入临界区。

Page Fault is issued by ?

答案是 MMU。OS 是处理而不是发出。

Considering a system, which uses virtual memory. At what point can address binding be done?

只能选 **execution time**。

In a 32-bit Linux, how much physical memory is needed for a process with three pages of virtual memory(for example, 1 code, 1 data, 1 stack)

32-bit（10-10-12），进程虚拟地址空间大小为 4GB，一个 PT 可以映射 4MB，一个 PD 可以映射 4GB，一个 Page 为 4KB。

这里我们认为 code & data 和 Stack 分别映射在低位和高位虚拟地址空间，因此：

需要一个 PD，两个 PT（分别对应低位的 code & data 和高位的 Stack），以及三个对应 Page，总共为 6。

For a 32-bit Linux, which of the following statements about virtual memory is correct?

- A. Kernel can directly access all 4G virtual memory.
- B. Kernel space uses 1G virtual memory, while all the other user processes share 3G virtual memory.
- **C. Kernel space uses 1G virtual memory, while each of the user processes has its own 3G virtual memory.**
- D. All user processes share the 4G virtual memory.

总体上说，请求分页 (demand-paging) 是个很好的虚拟内存管理策略。但是，有些程序设计技术并不适合于这种环境。例如？

- A. 堆栈
- B. 线性搜索
- C. 矢量运算
- **D. 二分法搜索**

Implementing LRU precisely in an OS is expensive, so practical implementations often use an approximation called ?

NRU (Not Recently Used)。

File access is protected by ?

- A. both user access rights and user priority
- **B. both user access rights and file attributes**
- C. both user priority and file attributes
- D. both file attributes and user password

The Linux Ext2/Ext3/Ext4 disk file systems use ? for file allocation.

- A. contiguous allocation
- B. linked allocation
- **C. indexed allocation**
- D. none-of-the-above

To mount a usb device located at /dev/sdb to /mnt/usb, suppose the file format is fat32. Which of the following shell scripts is correct?

- **A. mount -t vfat /dev/sdb /mnt/usb**
- mount -t fat32 /dev/sdb /mnt/usb
  - 没有 fat32 这个文件系统类型

Operating system for which the NTFS file system was developed ?

- A. Linux
  - ext4, xfs
- B. DOS
  - FAT16, FAT32
- **C. Windows 10**
- D. Unix
  - UFS

Which of the following storage device does not belong to the tertiary storage structure?

Hard Disks 是 secondary storage，而三级存储 teritiary storage 通常包括：CD-ROM, DVD, Tapes 等。

Which kind of swap space is fastest?

- A. A swap file on FAT
- B. A swap file on ext3
- C. A partition with sophisticated file systemfunctions
- **D. A raw partition**

CPU 调度算法？没有在 Linux 的 2.6 以后版中使用

SJF

在 ext2 文件系统中，下面的说法，？是错误的

- A. 每个文件都有一个 inode 节点
- B. 目录文件有 inode 节点
- C. 磁盘设备有 inode 节点
- **D. 打印机设备文件没有 inode 节点**