**导论 Introduce**

**什么是操作系统?**

计算机中一直运行着的程序就是**内核 (kernel)**

**bootstrap program**位于内核，放在ROM，用于把bootstrap导入内存然后启动(初始化代码)，放在固定位置。启动时最先运行的程序就是 bootstrap program

**MBR** (master boot record)在disk上，一个disk只有MBR, 并且只有一个boot sectors.

**并发concurrent**是同一段时间内发生，并行parallelism是同一时刻

**batch processing批处理系统**，特点是操作系统在处理一个事务的时候不能和用户发生交互，交互性差，但是有最佳的thoughout

**time-sharing（multi-tasking）分时系统**，操作系统轮流为多个用户服务, CPU 频繁切换作业，以便用户可以在每个作业运行时与其进行交互，从而创建交互式计算（交互性。分时系统是multitask的一个具体实现。分时系统不一定支持分时用户.有handheld device,必须分时

**multi-programming 多道程序设计系**统，通过安排作业（编码与数据）使得 CPU 总有一个执行作业， 从而提高 CPU 利用率，可能在内存上保留多个任务的信息。multitask是multiprogramma的一种

**Clustered System**集群系统: 往往使用storage-area network (SAN)来共享存储。Asymmetric clustering:has one machine in hot-standby mode;Symmetric clustering: has multiple nodes running applications, monitoring each other

**Distributed Systems**可以是client-server, peer2peer, clustering system..

**中断 Interrupt**

中断**会造成进程的切换，操作系统是中断驱动的**。中断通过中断向量执行，中断向量中会记录所有服务例程的地址。包括：**外部中断**：分为 I/O 中断和时钟中断；**内部中断** (异常 exception)：由 CPU 产生，包括系统调用、缺页异常、断点指令等等

中断处理会保存程序状态寄存器中的内容，但是子程序调用不会保存

**双模态dual-mode**

**用户态(user mode)**：只能访问为其分配的寄存器和存储空间，只能执行普通指令，用户程序和操作系统以外的服务程序都运行在用户态中，使用用户栈。不能修改它的page table

**内核态(kernel mode)**：可以访问所有的系统资源，执行特权指令，可以直接操作和管理硬件设备.内核程序运行的时候处于内核态，使用内核栈.可以用interrupt（比如system call, page fault）, trap（trap synchronous software interrupt）， exception从用户态切换到内核态

**Theorem** 不可能发生在用户态的事件是：进程切换，因为需要调度处理器和系统资源。而系统调用和中断既可以发生在用户态又可以发生在内核态。关闭中断，设置定时器初始值，内存单元复位，I/O，修改设备状态表都**发生在内核态**；而比如触发trap可以发生在用户态

**操作系统服务**

1.用户接口：操作系统提供的接口有分为命令接口和程序接口两类，命令接口包括命令行 CLI (commad)和图形用户界面 GUI，而程序接口有 system call

2.程序执行：将程序装载到内存中执行

3.I/O 操作：操作文件和 I/O 设备交互

4.进程之间的信息交流

5.错误检测：检测 CPU，内存，I/O 设备出现的错误

6.系统资源的分配

7.用户操作的记录(Accounting)：形成日志

**系统调用 system call**

系统调用是操作系统为用户态提供服务的Programming interface， API

1.系统调用是进程和 OS 内核的程序接口，使用高级语言来编写

2.操作系统的 API 封装了一系列的系统调用，每个系统调用对应了一个封装例程 (wrapper routine)

3.系统调用是在内核完成的，用户态的函数是在编程语言的库函数中实现

4.执行系统调用的几个过程：执行 trap 指令，传递系统调用操作，执行相关操作，返回用户态

**Theorem** 系统调用中**三种参数传递的方式**: 通过寄存器传递参数，通过栈传递，通过内存中的 block 和table 等数据结构存储参数，并在系统调用中把块地址作为参数放在寄存器中（基于栈和block的方法不限制参数长度）

fork，读取时间需要**通过系统调用来完成**

**内核结构**

操作系统分成了若干层次，**最外层是用户接口**，最内层是硬件

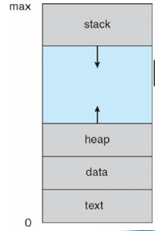
**单内核结构 (宏内核结构 Monolithic Kernels)**：内核的全部代码打包在一个文件中，优点是效率很高， 缺点是维护和修改内核非常困难，容易出 bug。Linux是宏内核，Linux使用了一种全新的内核模块机制，用户可以根据需要，Loadable Kernel Module(动态加载模块) 在不需要重新编译内核的情况下动态地装入insmod/移除内核

**微内核接口(Micro Kernels)**：内核的最基本功能由中央内核实现，其他的功能都委托给独立的进程来实现，这些进程和内核通过一定的接口来通信，这些即内核的模块。更安全，可扩展，分布式计算,可以使得一些用户服务被视为用户程序

**虚拟机**将分层方法发挥到了极致。它将硬件和操作系统内核视为所有硬件

**进程和线程**

**进程(process, job)**

进程就是正在执行中的程序，放在外存的程序就不是进程。进程是计算机工作的基本单位，进程包含了一系列的指令和资源，是一个抽象实体，在执行任务的时候，要分配和释放各类资源。每个进程都有唯一的标识号 pid(process identifier). 进程与程序最大的不同是进程有生命周期

进程包括:

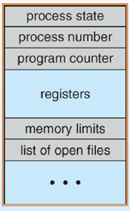
- 程序代码 text section

- 数据区data section，存储**全局变量**

- **堆heap**(构造为完全二叉树，父亲需要都比儿子大或者小)，存放动态分配的变量

- **栈区satck**，存放临时变量

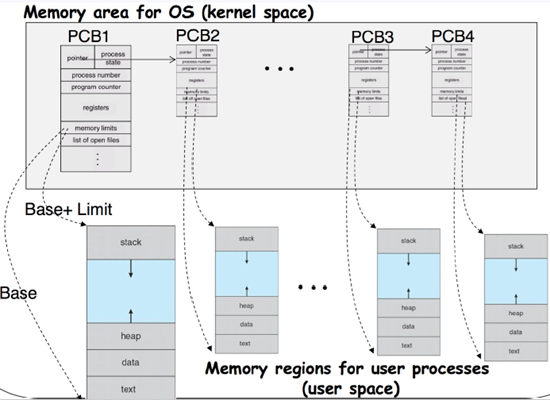
**进程控制块 Process Control Block (PCB)**

PCB位于内核，包含进程的状态(run, etc)，programm counter（下一条指令的地址）， 寄存器信息，调度信息，内存管理信息， 统计信息，I/O状态信息

在Linux 内核中，进程的信息用一个 C 语言的结构体task\_struct 表示(current是其指针)，定义

在头文件<linux/sched.h>中，一系列进程构成一个双向链表。进程的 task\_structure 是进程存

在的唯一标志



**进程的调度 Process Scheduling** （进程调度不需要额外的硬件）

进程的调度需要在ready(**ready queen** in main memory, ready)，running (**job quene**)和 waiting(**device quene**,等待I/O等设备) 三种状态之间切换，进程的调度可以分为

**长程调度(job scheduler)**：选择应将哪些进程放入内存(ready quene),不频繁。决定degree of multiprogramming

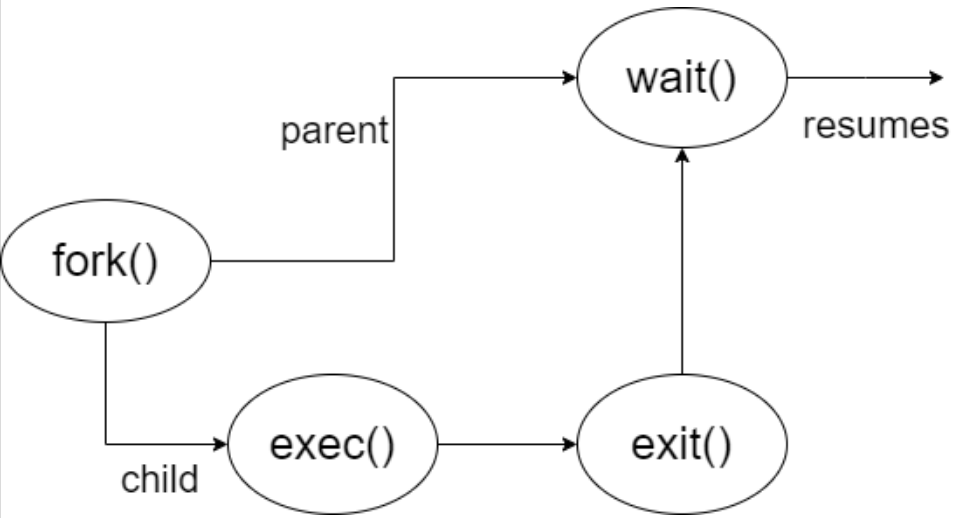
**短程调度(CPU scheduler)**：CPU 可以选择下一个执行的进程，非常频繁，也是调度最关心的

一种进程分类的方式是将进程根据主要耗时的操作类型来分，即分为 **I/O bound**(花大量时间在I/O上，而不是计算上) **CPU bound**

**上下文切换Context Switch**：进程切换时，保留上一个进程状态，同时载入的进程状态（比如一些相关寄存器，stack，memory，但是不会影响全局变量），刷新指令和数据，在 PCB中有专门存储上下文信息的地方。

**进程的创建与 fork 系统调用**

父子的resource的关系可以是全部共享，部分共享，不共享



**fork**被调用之后系统先给新的进程分配资源，然后把原来的进程的**数据**复制到新的进程中（两份数据彼此独立），函数的返回值是一个pid，调用 fork 之后两个进程的除了返回值pid 不同以外没有任何不同，其中父进程fork函数返回的是子进程的 pid，子进程返回的是 0。父进程等子进程运行完再继续运行

fork 系统调用可能只会复制当前的调用它的线程(考试)，也可能会复制进程中所有的线程

fork-> sys\_fork -> do fork 是真实的流程

**进程的替换与 exec 系统调用**

系统调用 exec 是以新的进程去代替原来的进程，但进程的 pid 保持不变。因此，exec 系统调用并没有创建新的进程，只是替换了原来进程上下文的内容。原进程的代码段，数据段，堆栈段被新的进程所代替。exec 系统调用可能会替换掉整个进程，但是不会只替换掉调用它的子线程里的内容。

**进程的终止与 wait**

进程的终止分为：正常结束，异常结束和外界干预，可以用 exit(0) 来退出进程

系统调用 wait()：通过wait 系统调用可以让父进程等待子进程结束之后再执行，wait 调用会返回进程的状态信息和结束进程的 pid；父进程一旦调用了 wait 就立即阻塞自己，由 wait 函数分析是否当前进程的某个子进程已经退出了；如果找到了退出的子进程就会将其销毁，如果没有找到已经退出的子进程，wait 就会一直阻塞父进程

**Cascade termination** ：父进程terminates导致子进程全部abort

**进程间通信**

进程根据是否可以和其他进程进行通信和交互分为Independent独立进程和Cooperating合作进程。而进程间的通信又分为直接通信(link自动建立，有且仅有一个link)和间接通信（使用相同mailbox的时候建立link，一个link可能连接多个进程，每个pair进程会分享多个link）。

**共享内存模型shared memory**(共享的内存来进行数据的交互)和消息模型message passing(消息队列的数据结构来缓存进程之间的通信内容)是两种常见的**interprocess Communication (IPC)**

**常见的通信机制有**：信号 signal，共享内存，共享文件，数据库，管道 pipe（把⼀个进程的标准输出与另⼀个进程的标准输⼊连接起来），消息 message 和套接字 socket，Linux中还有文件锁和 POSIX 线程，互斥锁 mutex 等等。

**阻塞的消息传递(blocking, synchronous**)就是发送方进程需要阻塞直到消息被接受，而接收方需要阻塞到有消息收到，

**非阻塞(non-blocking, asynchronous)**当发送方和接受方都是阻塞的时候，就可以认为二者达到了一个 rendezvous。发送方可以一直发，接收方收到valid message or null

**Communication in Client-Server Systems**：Sockets，remote Procedure Calls, Remote Method Invocation (Java)

**线程**

进程是操作系统中的资源的拥有单位和调度单位，拥有虚拟的地址空间，控制了一些资源， 有状态，优先级和调度。而调度的单位则称为线程 (thread)，线程是进程中的一个执行单元或者可以调度的实体。多线程和单线程的优劣要看具体情况

特点：1.有执行状态，不运行的时候会保存上下文 2.每个进程有一个**独立栈**，有局部变量的静态存储区 3.线程可以取用所在进程的资源，但是不拥有系统资源 4.可以创建、撤销另一个进程 5.一个进程中的多个线程可以并发执行，相比于进程系统的开销小，切换快

**线程和进程的比较**：

1. 进程通常是独立的，而线程则是进程的子集（同一process的不同thread**分享**code, data, files;或者说，分享heap，global变量，data section，但是**不不分**享寄存器值和stack） 2. 进程比线程携带更多的状态信息、 而进程中的多个线程共享进程状态以及内存和其他资源3.进程有**独立的地址空间**，而线程共享进程的虚拟地址空间4.进程只能通过系统提供的进程间**通信机**制进行交互5.同一进程中线程间的上下文切换通常比进程间的上下文切换**快**

**多线程模型**

用户级线程（User Threads）：不依赖于 OS 内核，利用线程库函数来控制用户线程。用户不了解用户线程的存在，线程切换不需要内核的特权。一个线程发起系统调用而阻塞的时候，整个进程会等待。即使习题不支持多线程，也能进行模拟

内核级线程（Kernel Threads）：依赖于 OS 内核，在内核的内部需求进行创建和撤销，线程的切换和上下文信息维护都由内核完成

**lightweight process (LWP机制)** 用户线程和内核线程之间的虚拟处理器，进程可以在LWP上安排用户线程运行;每个 LWP 都附属于内核线程

**multithreaded programming**好处Responsiveness, Resource Sharing, Economy, Scalability

而多线程模型可以分为如下：

**多对一模型（Many-to-One）**：多个用户级线程对应于一个内核线程，通过用户级的运行时库来实现，OS 感觉不到用户线程的存在。优点有不需要 OS 的支持，可以调整调度策略来满足应用程序 (用户级) 的需求；缺点是当一个线程阻塞的时候整个进程都会阻塞

**一对一模型（One-to-One）**：一个用户级线程对应于一个内核线程，线程的创建、同步和调度需要 OS 内核来进行，比如 Windows XP 和 Linux 都是一对一模型。优点有多个进程可以并行，当一个线程阻塞的时候其他的线程依然可以执行；缺点是线程操作的开销更高，OS 内核必须有更好的 scale

**多对多模型（Many-to-Many）**：多个用户线程对应多个内核线程，方便，比如 Windows NT/2000

此外还有二级模型，就是多对多 + 一对一的缝合版

**Thread Pool**: 在pool中可以创造很多thread，并且使得thread在pool中等待，优势：

1. 使用现有线程处理请求通常比创建新线程稍微快一些。

2. 允许将应用程序中的线程数量限制在池的大小范围内。

3. 将要执行的任务与创建任务的机制分离，可以采用不同的策略来运行任务。例如，任务可以被安排定期运行。

**Thread Cancellation**

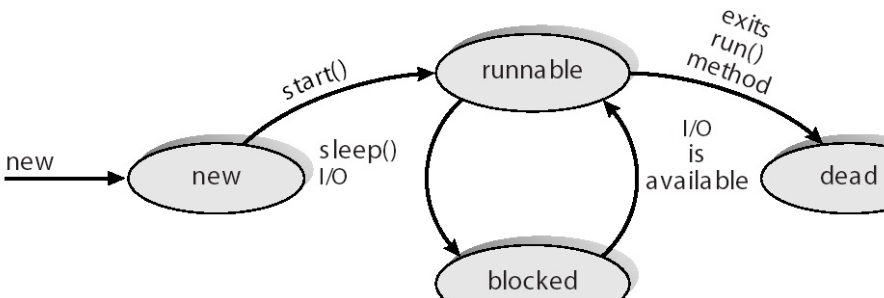
Asynchronous cancellation 立即终止目标线程

Deferred cancellation 允许目标线程周期性地检查是否应该取消

**Linux thread**

clone() allows a child task to share the address space of the parent task (process)

**Java Threads**



**CPU 调度**

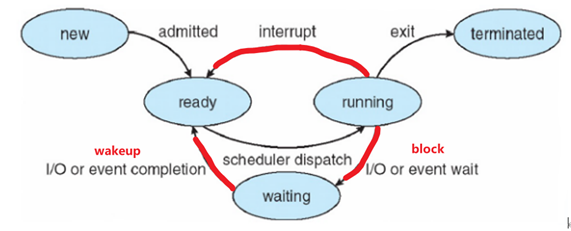
**基本概念**

- Maximum CPU utilization obtained with multiprogramming

- C**PU–I/O Burst Cycle** – 程序运行需要 CPU运行和I/O等待. CPU burst再I/O burst。调度器不会关心进程在CPU busrt还是I/O

- CPU burst distribution：A burst is an instance of breaking

**调度发生的时间节点：**running to waiting（非抢占式）,running to ready state（preemptive）, waiting to ready（preemptive）, terminates（nonpreemptive）



自己主动需要某些资源，比如IO和文件是**waiting状态**(在后面就是block阻塞态)，而如果是因为 CPU 等外部因素，比如时间片轮转和优先级，会被调度成**ready状态**。而terminated则是因为发生意外导致了进程终止或者进程自己正常运行之后结束。

★ 面向用户的评价标准

**周转时间 Turn-around Time**：进程从arrival到完成所经历的时间

**响应时间 Response time**：从进程提交到首次被响应所需要的时间

**等待时间 Waiting Time**：进程在ready quene中等待的时间之和 (一个进程可以有多段等待时间相加。就等于 turnaround time – burst time)

★ 面向系统的性能准则

**吞吐量 Throughput**：系统在单位时间内完成的进程数

**CPU 利用率**：CPU 工作时间占总时间的比重，我们需要让 CPU 尽可能地忙碌

**调度算法**

**FCFS 算法**：有利于长进程而不利于短进程，对 CPU 型进程(short burst)友好

**SJF 算法**：预计执行时间(CPU burst)最短的进程优先，但事实上很难实现，因为运行时间不好估计。它的抢占式调度版本Preemptive SJF也叫 Shortest-Remaining-Time-First (SRTF)，如果新来的比当前执行的花的时间少就抢占。 SJF is Optimal in average waiting time

**Priority 调度**：进优先级高的进程先执行。静态优先级会在进程一提交就确定优先级，动态优先级则会对优先级随情况调整

**饥饿 (Starvation)**：(发生在SJF, Priority中)优先级低的进程长时间得不到 CPU 的使用权的情况。优先级高的进程霸占 CPU，解决的方法是老化 (Aging)——采用动态优先级策略，让进程的优先级随着时间的推移而提高。

**RR 算法 (Roung Robin分时调度)**：基于分时系统，先将系统中的进程按照 FCFS 排成队列，每次调度将 CPU 分配给队列顶端的进程执行一个时间片Quantum，在时间片结束之后会发生中断，将该进程切换到就绪状态并放到队列末尾(注意：1. 一个时间片结束后会放到队列末尾，所以如果优先级相同，会切换到另一个进程 2.抢占式调度立刻发生，中断的时间片直接排道队列最后 3.一个进程运行时间不足一个时间片的时候，时间片会提前结束)，然后进行上下文切换，执行下一个进程。时间片够大的时候就是FCFS。对于CPU bound，时间片越长越好

**高响应比优先调度算法**Highest Response Ratio Next(HRRN)，满足短任务优先而且不会饥饿。每次调度时，先计算R，选R最高的调度

响应比R = (等待时间+ 要求执行时间) / 要求执行时间

**多级队列调度 Multilevel Queue**：将就绪队列也分为若干个子队列，不同的队列可能有不同的优先级、时间片长度和调度策略。允许进程在不同队列之间迁移

**Asymmetric multiprocessing** 只有一个处理器访问系统数据结构，减轻了数据共享的需要；其他处理器只执行用户代码；**Symmetric multiprocessing(SMP)** 每个处理器都是自我调度的。多个处理器可能会访问和更新一个共享的数据结构。

**同步 Sychronized**

**基本定义**

**竞争条件Race Condition**：多个进程都需要访问和修改一块共享数据区，此时数据的状态取决于最后一块访问这一区域的过程。竞争条件面临的三个主要问题：1.互斥 Mutual Exclusion 多个进程不能同时得到一个资源2.死锁 Dead Lock 进程之间互不相让，导致永远得不到资源(多个信号量交叉等待)3.饥饿 Starvation 资源分配不公平导致进程长时间得不到资源

例子：初始值为1，一个++，一个--，最后值可能为0，1，2

**临界问题 Critical-section**

临界资源 Critical Resource，一次只允许一个进程访问的资源(mutex=1)，比如消息缓冲队列；

临界区 Critical Section，进程中访问临界资源的代码片段(可以进行进程调度)。 代码中体现为P和V包着的部分

同步问题的 solution 必须满足以下几个条件：1. 互斥 Mutual Exclusion，一个进程在访问临界资源时其他进程不能访问

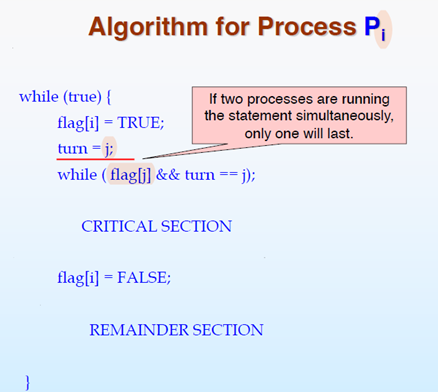
2.空间让进 Progress，如果临界资源没有进程在访问，并且有进程想要进入临界区域，就应该允许其进入

3.有限等待 Bounded Waiting，进程需要在等待有限时间之后可以访问资源，即不能产生死锁

用specail machine instruction解决会存在busy waiting

**Peterson’s Solution**

假设有原子操作LOAD和SOTRE, 然后可以定义proccess Pi为下图。理论可行但是事实上不可行，因为现实中不存在原子操作



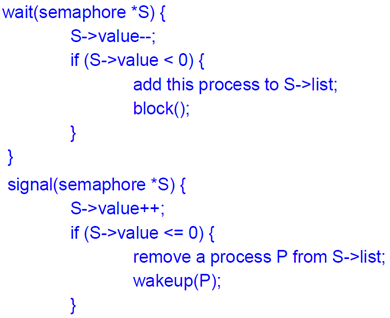
**同步硬件 Synchronization Hardware**

对于uniprocessor(只有一个CPU)，只要disable中断;其余可以通过在硬件中定义原子操作

**spin locks** 原地等待方式解决资源冲突的，一个线程获取了一个自旋锁后，另外一个线程期望获取该自旋锁，获取不到只能够原地busy wait,多用于multiprocessor systems

**信号量 Semaphores**

信号量是一种数据类型，只有**wait(--)v**等待资源和**signal(++) p**释放资源 2种**原子操作**，信号量可以分为整型信号量、记录型信号量, 二值信号量(只有0，1，不是说只能被2个线程访问)。信号量的两种**原子操作**（不能中断，一旦开始只能执行完）：block() 是将进程放置到等待队列上面，即从 running 切换到 waiting， 而 **wakeup()** 则是讲 waiting 转换成 ready 状态。信号量可以解决所有mutual exclusive



list 指向等待队列的下一个进程，记录型信号量 >0 表示资源的个数，<0表示在等待队列中的进程个数(0的时候没有等待)，不会busy waiting

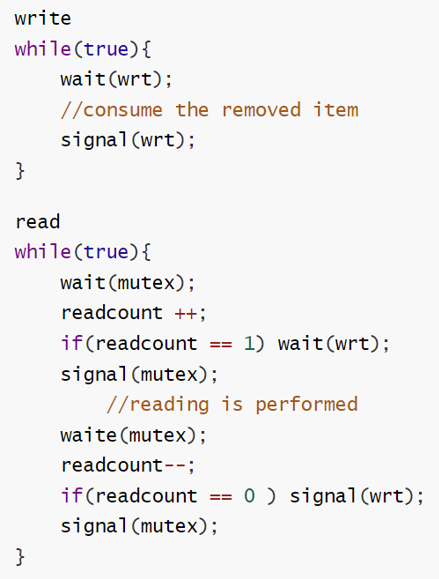
**同步的三个经典问题**

mutex=n 允许n个进程同时访问一个资源

**Bounded-Buffer Problem**：生产者写入缓冲区，消费者读取缓冲区。假设缓冲区有 N 个，需要设置三个信号量 (排它锁)mutex=1, (两个临界信号量) **full=0, empty=N**。V队列可以反，P队列不能反

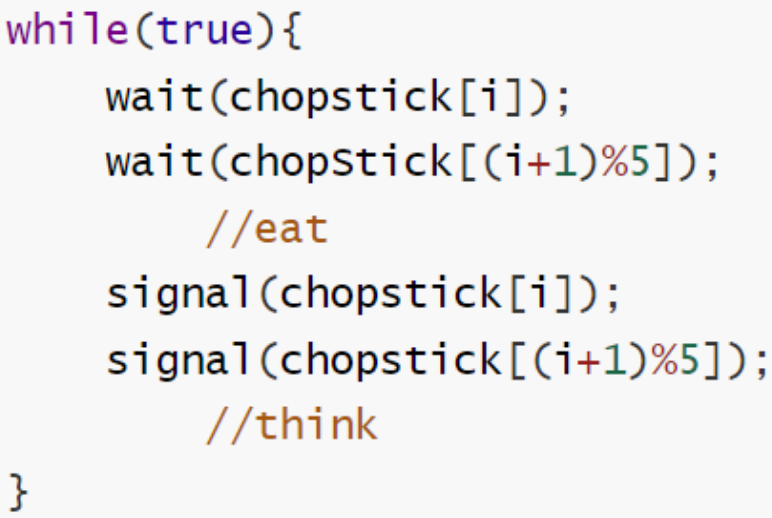


**Readers and Writers Problem**允许多个读者一起读， 但是写者只能有且仅有一个。并且读者和写者不能一起。两个信号量 mutex=1, wrt=1。一个计数器readcount=0



bounded-wait writer会变成一个饥饿问题

**Dining-Philosophers Problem**：一群哲学家围成一圈吃饭，每两个哲学家之间有一根筷子，一个人吃饭需要拿起左右两边的筷子，哲学家在吃饭和思考之间切换状态。五个信号量 chopstick[5]=1(均为1)



如果都是左撇子（都选择eat，并且先左边），会导致死锁问题。可以再加个count=4或者搞个mutex=1(放在刚开始的两个wait的两侧)，还可以奇偶来先左or右来避免死锁

**死锁 DeadLock**

**基本概念**

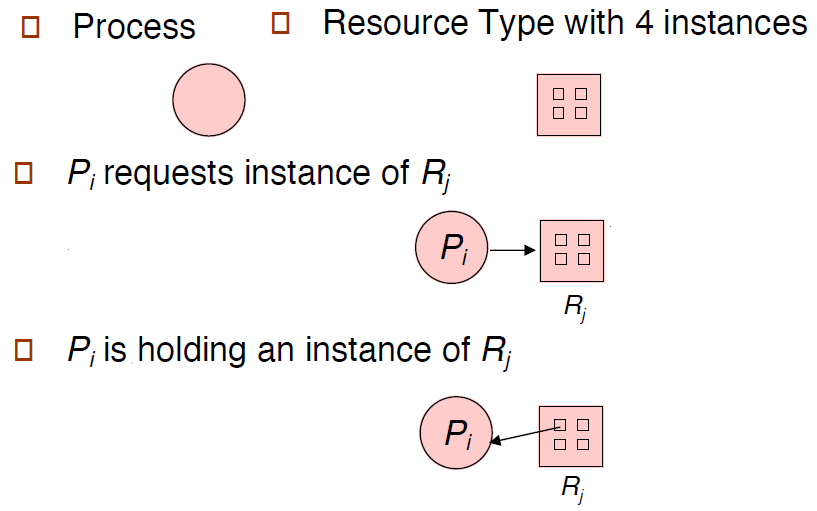
多个进程因为共享资源而造成互相等待，进程将永远阻塞不能继续执行，四个必要条件：

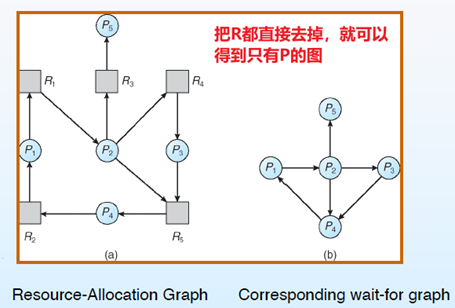
1. 互斥 Mutual Exclusion：一个资源只能被一个进程使用2.占有并等待 hold and wait 3.资源不可抢占/剥夺 No preemption 4.循环等待 Circular Wait：一系列的进程都占有了前一个进程需要的资源，并在等待后一个进程释放资源(每个进程按照资源id号从小到大的顺序请求资源就能避免)。N个进程，M个资源，每个进程可以分配W个资源



**资源分配图 Resource Allocation Graph**

**P** 表示进程，**R** 表示资源，一个资源节点可以有多个实例。请求边request edge P指向R，表示进程请求这个资源，分配边R指向 P，表示资源被进程占用。资源分配图的变形就是等待图。资源分配图中有环就可能存在死锁：1.没有环的肯定没有死锁 2.有环并且only one instance per resource type，产生死锁 3.有环但是several instances per resource type，就不会产生死锁





**死锁的处理**

(never进入死锁)在程序的开发阶段预防和避免死锁，（死锁recover）在程序运行的时候进行死锁的检测和解除，**（鸵鸟方法）假装死锁从**来没发生过被几乎所有操作系统使用

**预防死锁prevention**从产生死锁的四个必要条件出发（显然这要的要求是比较苛刻的，也难以达到）：1.互斥：给不能共享的资源加锁2.占有并等待：确保一个进程请求一个资源的时候不占有其他任何资源3.资源不能被抢占 4.循环等待：资源必须有序地申请，找到需要的数量从小到大的顺序，可以破坏循环等待

**死锁避免Avoidance**

如果存在一种调度方式使得进程可以申请到所有所需资源并完成运行，那么就安全状态，而这个进程序列也叫做安全序列safe sqeuence

safe state不会产生死锁；但unsafe不一定会死锁

**资源分配图算法**(用于避免单资源实例)

用虚线来标识需求边(claim edge是可能需要，当真的需要资源的时候会转换为需求边)，在一定条件下可以转换成请求边(请求边释放了就又会重新变为claim edge)，如果进程 P 需要资源 R，就作一条 P 指向 R 的需求边，只有当这条边变成分配边 R-P 而不会使资源分配图形成环的时候才允许这次申请。

**Banker’s Algorithm** （用于避免多实例资源死锁），没有真实os会用

假设有 n 个进程和 m 种资源：

**available[m]=k** 表示第 m 种资源有 k 个实例 (m)

**max[i,j]=k** 表示第 i 个进程需要资源 j 的 k 个实例 (n\*m)

**allocation[i,j]=k** 表示第 i 个进程已经被分配了资源 j 的 k 个实例 (n\*m)

**need[i,j]=max[i,j]-allocation[i,j]** 表示第 i 个进程还需要资源 j 的 k 个实例

**Algorithm 1 判断多资源实例是否安全**

1: 令 work=available，finish[i]=false for 1 ≤ i ≤ n

2: 找到一个下标 i 使得 finish[i]=false 并且 need[i] ≤ work[i] 即每种资源的需求都不超过可分配的数量，如果没有符合条件的就进入第 4 步

3: work=work+allocation[i],finish[i]=true 即第i 个进程在分配到资源之后成功执行，并且释放了所有占有的资源

4: 如果所有的 finish[i] 都变成了 true，则系统处于安全状态，否则就是不安全的

**Algorithm 2 资源分配算法**

1: 令request[i,j]=k 表示进程i 需要资源j 的k 个实例，必须要保证 request[i] 不超过need[i] 和available[i]

否则就出现错误了

2: 假装将第 i 个进程所需要的资源都分配给了它，即进行了如下操作：

available-= request[i]

allocation[i] += request[i]

need[i] -= request[i]

3: 调用安全算法来检测进行了上面的操作之后系统是否处于安全状态，如果是就将资源分配给 Pi，否则 Pi

必须进入等待状态（如果不需要等待就是可以立即分配，不用交换序列顺序）

**死锁的恢复**

**进程中止** Process Termination：杀掉所有或者一个个杀直到恢复

资源抢占 Resource Preemption： Rollback回到**安全状态**（Starvation 相同的进程可能会一直被选中作为牺牲者，包括回滚次数在成本因素中）

**主存**

**基本概念**

程序一定要从磁盘装在入内存之后才能运行，内存和寄存器是 CPU 可以直接访问的存储部分，而 cache 位于内存和寄存器之间

**内存保护**

每一个进程在内存中都应当有一块连续的内存空间，而单个进程应当只能访问自己的内存空间，而不能访问其他进程的内存空间。 base到base+limit的范围内才是安全的，这两个特殊的寄存器只能由内核特权指令修改。通过MMU具体实现，如果不在范围内就会异常，中断程序

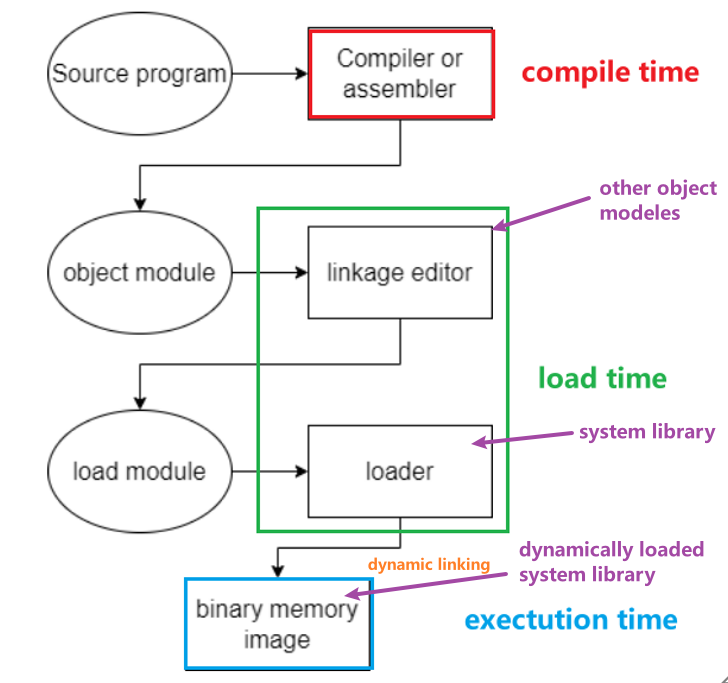
**地址绑定 Binding**

用户的源代码需要经过一系列步骤才能变成可执行程序并装入内存运行,分三个阶段：编译时间(compile time)，装载时间(load time)和执行时间(execution time,绑定结束)。而内存也分三种：符号地址(symbolic addresses)，可重定位地址(relocatable addresses)和绝对地址(absolute addresses)。

**compile time**，compiler 会将代码中的 symbol 转为可重定位地址；而如果在 compile time 就知道了进程最终会被安置在何处，那么在 compile time 就将 symbol 转为绝对地址(如果起始地址发生改变，就需要重新编译)

而一般在**load time**，可重定位地址会转为绝对地址，当进程起始地址发生改变时，我们只需要重新装载即可。

如果进程在 **execution time** 时，允许被移动，那么可能从重定位地址会转为绝对地址,这一步就需要延迟到执行时间来执行。绝大多数操作系统都使用这种方案。



**逻辑地址(虚拟地址)和物理地址**

逻辑地址，由 CPU 生成(一般obj,exe中文件地址都是逻辑地址)，首地址作为0，而其他的指令的地址都用相对于首地址的偏移量来决定,逻辑地址不能直接用来读内存中的信息。物理地址，可以用来直接寻址。均为页号+偏移组成一对基地址寄存器和限长寄存器定义了逻辑地址空间，其中基地址寄存器存放地址的首地址，而限长寄存器存储整个地址空间的长度

**内存管理单元 MMU**

MMU 将虚拟地址映射成为物理地址，在 MMU 中，重定位寄存器(dynamic relocation using a relocation register)中的值被加到每一个由用户进程生成的地址中。MMU 还负责内存访问的保护。TBL 也属于 MMU 的一部分

**动态装载dynamic loading**

如果程序还没有被调用，那么它会以可重定位装载格式(relocatable load format)存储在磁盘上；当它被调用时，就动态地被装载到内存中。即，例程只有在需要的时候才被载入内存。对于大量但不经常需要访问的代码片段（例如错误处理代码），这种方式可以节省大量的内存空间。动态装载不需要OS支持，而是由开发者来负责实现

**动态链接和共享库**

能被动态链接的库就被称为动态链接库(dynamically linked libraries, DDLs)，由于它们可以被多个进程共享，所以也被称为共享库(shared libraries)。动态链接需要OS

**交换技术 Swapping Swap Space**

一个进程可以暂时被移出内存并保存在后备存储器中，之后再装入内存执行。后备存储器是一个可以存储所有内存映像的磁盘，可以提供对这些内存映像的直接访问，在 Linux 中表现为交换区，而在 Windows 的系统中是一个 pagefile.sys 文件。

swap主要是transfer 的过程耗时。系统会维护一个准备队列来存放准备运行的进程，其内存映像都存储在磁盘上面。系统在正常运行的时候是不会去交换的，只有当可用内存数量低于一个阈值的时候才会启动自动交换，在可用内存增加到一定数量的时候停止换出。

**Lazy swapper**： 直到一个page要被用到才真正换到memory中。最快的swap space是raw partition

**连续分配 Contiguous Allocation**

主存分为两个单独的分区，常驻操作系统 (Resident operating system) 保存在具有中断向量的低内存中，用户进程被保存在高内存中,浮动寄存器 relocation register被用来保护用户进程，将进程和进程之间隔离开，并防止进程更改OS中的数据和代码，其中：

**Base register**保存最小的物理地址

**Limit register**包含了逻辑地址的区间，每个逻辑地址必须要处于这个区间中

**MMU** 是动态映射逻辑地址

**分区式管理**:将内存划分成若干个连续的分区，每个分区只能存放一个进程，分区的方式有：固定分区就是将内存分成若干块固定长度的区域，区域和区域之间存在着未使用的内存空间；动态分区需要动态地划分内存，在程序装入内存的时候把连续的一块内存区域分配给该进程，并且大小正好适合该进程的需要，存在下面几种动态分配算法（注意这类题1.**一块内存只要空间够可以分给不止一个进程** 2.每次分配后刷新剩余大小）：

**First-Fit** 分配第一个发现的可用的足够大的内存分区给进程

**Best-Fit** 分配所有可用块中最小的并且足够大的内存分区给进程

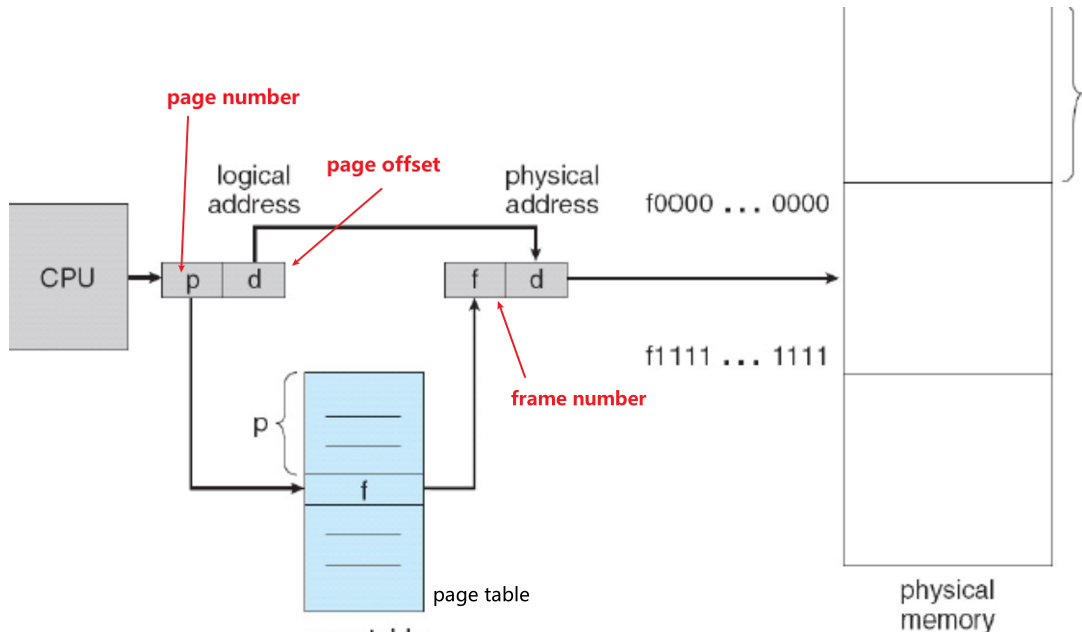
**Worst-Fit** 分配最大的一个分区给进程。在速度和内存利用上都比前两种要差

很明显，best-fit 和 worst-fit 算法需要遍历所有内存之后才可以做出决定

一个进程内部的空闲的内存（固定分区的时候出现），称为**内部碎片 (inner frag- mentation)**，相邻的两个内存空间之间空闲的内存段叫做**外部碎片 (outer fragmentation)**，碎片的存在会减少可运行的进程的个数。可以通过拼接和压紧compaction来进行管理，但是消灭内存碎片开销很大。Multiple-partition可以减少外部碎片

**页式存储管理**

可以使一个进程内部的逻辑地址所处的物理空间可以不连续，只要在内存有空的地方就可以给进程分配物理内存。物理内存被分为若干个固定大小的帧 (Frame)，而逻辑地址也被分成了若干个大小相同的页 (Page)，一般来说页和帧的大小相同，都是 4KB，运行一个含有 n 页的程序需要找到内存中 n 个空闲的帧。



逻辑地址结构由页号 p 和页内偏移量 d 组成，对于逻辑地址空间大小为 2^m，页的大小是 2^n B的情况，页号有 **m-n** 位，偏移量有 **n** 位 （单级页表的page table大小与程序有多少page无关，entry数量\*每个entry大小）

**页表 Page Table**

页表位于内存中，记录了页面在内存中对应的物理块号，每一个页表项由页号和物理内存的块号组成。进程执行的时候，通过查找该表，把逻辑地址空间通过页表的转换变成了物理地址空间。Page table 中还会有vaild位表示是否有效。每个进程都有Page Table，但是TLB只有唯一一个

同时，页表也有**页表寄存器 PTR** 来存储页表的(base)起始地址 F 和(length)页表长度 M，两个寄存器分别叫做 **PTBR** 和**PTLR**。页表的访问速率如果太慢就会导致程序运行的效率降低，同时，页表如果太大，就会导致内存的利用率降低。

**快表 TLB**

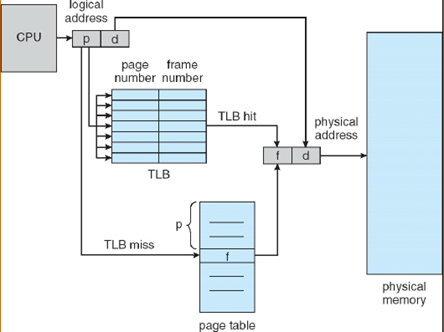
TLB，translation look-aside buffers，相当于页表的cache，存放了一些当前访问的若干项页表项，支持并行的快速访问，相对应的内存中的页表也叫做慢表

1. CPU 给出程序的逻辑地址之后，先去查询 TLB 2. 若查到了对应的物理块号，就加上偏移量之后直接去内存访问对应的物理块 3.若没有在TLB 中查到就在页表中继续查询对应的物理块号，然后将查到的物理块号存储到 TLB

TLB 的访问命中率α，假设快表 TLB 的访问需要的时间是x，内存访问时间(memory reference)是 y，则有效访问时间 (EAT) 的计算公式是：

EAT = (x + y)α + (x + 2y)(1 − α)

TLB 中命中的时候只需要直接访问内存即可，耗时 x+y;而失配的时候要多访问一次内存去查页表，因为需要的时间是 x+2y 。没有TLB的时候**EAT=2y**



TLB Reach =TLB Size \* Page Size

提升Page Size可以降低page fault, 会导致更多的碎片，可以通过提供多种不同的Page Size来解决

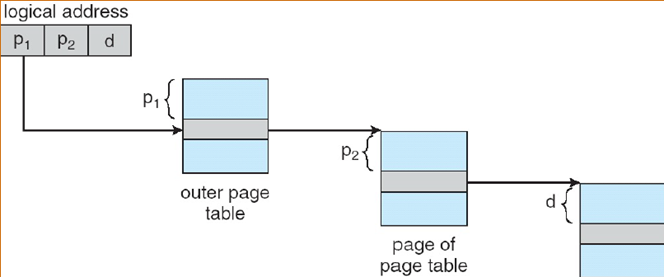
**共享页**

多个页可以对应同一个帧，共享页可以用来提高代码重用率，共享库通常用共享页来实现

**页表结构**

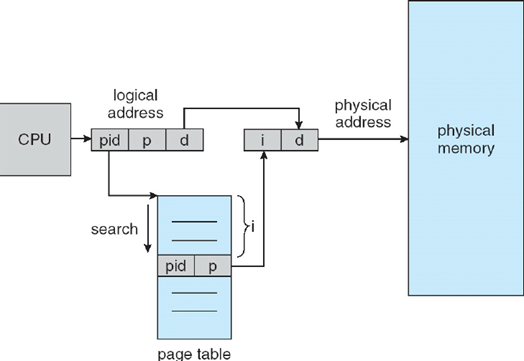
这些方法减少了存储页表所需要的内存空间，但是加大了内存的访问时间

**分级页表Hierarchical Page Table**，将页号继续拆分成页号 + 偏移量，用新的页号作为原本页号的索引，层层查找页表。只有第一层是满的，之后的几层只有有需要的时候才有（按需分配）



**哈希页表 Hashed Page Table**，将逻辑地址用哈希函数映射到页表中, 搜索速度较慢，因此将页表项放入哈希表中。TLB 可以用来加速哈希表的引用。

**反向页表Inverted Page Table**，页表中通过 pid 和逻辑地址共同查询物理地址号。整个系统中只有一张表。整个操作系统只维护一张反转页表，由物理地址做索引，从0开始排，然后对比对表项中的逻辑地址是否相同再找到物理地址。由于不需要每个进程都存储一张页表，只**存储物理地址数量**个表项, indexed by frame number，每条记录的内容是哪个page属于这个frame，所以相对来说节省了内存空间。但是无法支持共享页。



**段式管理 Segmentation**

可以将一个进程分成若干段，包括代码段、数据段、栈区、本地变量区等等，每个段中从 0 开始编地址， 并分配一段连续的地址空间，逻辑地址分为段号和段内偏移量。并且每个进程都有一张表示逻辑地址和物理地址映射关系的段表，其中每个段表项对应进程中的某一段，表中的项记录了段号、段长和本段在内存中的起始地址。每个段长度不一样，共享内存的时候分段更方便

映射方法和页表一致，就是，先按照segment的号去找到对于的基地址

基地址＋偏移量， 如果偏移量超出length就是invalid

**虚拟内存 Virtual Memory**

**基本概念**

虚拟内存的基本思路就是将一个程序正在运行的部分放入物理内存中，而不运行的部分则放在磁盘中，使得虚拟内存的空间要比物理内存大。虚拟内存的管理分为请求调页 (Demand Paging) 和请求段式管理(Demand segmentation)两种。程序有较好的局部性locality的时候虚拟内存能够更好地发挥作用

**按需调页 Demand Paging**

只在需要的时候将进程的页调入内存中运行，减少 I/O 和内存的使用，程序的响应更快，支持更多用户同时使用。但是二分法搜索不适合这种环境

**页表的结构**

有效位 (valid bit) 为1则在memory中，否则page fault，在请求调页系统中，每个页表项的页号对应：

- 物理块号

- 状态位 P，表示该页是否已经调入内存中，供程序访问页的时候参考

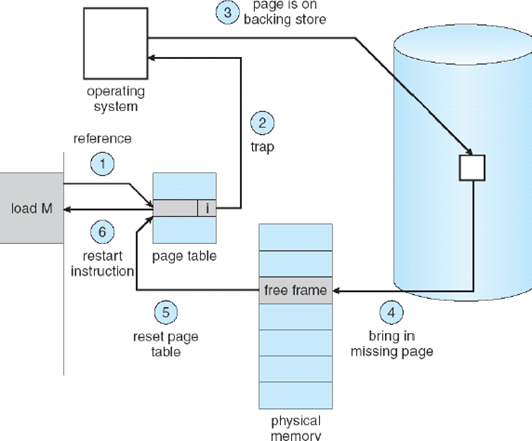
- 访问字段 A，用于记录本也在一段时间内被访问的次数，或者已经有多久没有被访问

- 修改位 R/W，表示页在调入内存后是否被修改过

- 外存地址，指出该页在外存中对应的存放地址

**Page Fault**

程序第一次访问页表的中的某一项的时候肯定会导致缺页，因为这个时候这一页还没有调入内存中， valid 位是无效，此时操作系统会把页调入内存中并修改表项，然后重新运行刚才的程序 (因为没有访问到对应的页)。有n个不同的页面，引用字符串长为p, 则缺页次数不会少于n，不会多于p



缺页率p

EAT=(1 – p)\*physical-memory-access+p\* (page-fault-overhead+swap-page-out+swap-page-in+restart-overhead)

Page fault和程序结构也有关

**进程创建**

虚拟内存允许进程创建时**写时拷贝(COW copy-on-write)**，允许父进程和子进程一开始的时候共享一些页，如果有进程需要改变页的内容，才会 copy 一份这个页供两个进程分别使用

**页的置换 Page Replacement**

无空闲的物理帧frame的时候就需要进行页的置换，将一部分没有用的页换出，把有需要的页调入内存中。页的置换的目标是尽可能减小置换之后缺页的概率

将页面换出时需要将原本物理帧中存访的页调出时，我们可以使用1bit的dirty位来标识这一页是否被修改过，如果修改过就需要写入磁盘中

别忘了最开始的3次缺页！

**FIFO先进先出算法**，缺点是会引发 Belady’s Anomaly(更多的帧导致更多的缺页)

**OPT 最佳页面置换算法**：选择未来不再使用的或者当前最不需要用的页替换，使用的时候需要往后搜索引用串，观察最晚被再次引用到的现有页

**LRU 最近最少使用算法**：将距离上次使用最久的帧换出，记录页面使用时间的硬件开销太大，实现方法有：1. 计数器，每个页有一个counter，每次页被引用就把时钟拷贝到计数器中 2.stack，用一个双向的链表来维护一个页号的栈，被引用的时候就移动到栈顶，优势在于确定替换页面的时候不需要搜索

**LRU Approximation 算法**

★ **NRU**，使用一个reference bit，初始化为 0，使用过之后就变成 1，优先替换引用位是 0 的页，很明显这个算法非常局限

★ Second chance，也叫**clock算法**，如果被替换的页的引用位是 1 就将引用位变成0，然后保留这一页，按时钟顺序往下找，直到找到一个引用位是 0 的进行替换，替换后指针往下走一格停下，也就是说在引用位全是 1 的时候可能会转一圈

**其他**

**帧分配**

固定分配：**平均分配算法 Equal allocation**每个进程分到等量的帧，按比例分配 Proportional allocation 就是按照进程的大小来分配相应的物理帧

**优先级分配**：按照优先级比例分配。如果一个进程发生了缺页，可以置换自己的帧，也可以置换出一个优先级更低的进程的帧。置换的策略有**全局置换**就是可以置换所有可以置换的帧，问题是不能预测page fault rate 而**局部置换**只能置换该进程拥有的帧，问题是 low throughput: 因为不能用其他进程的空闲帧。

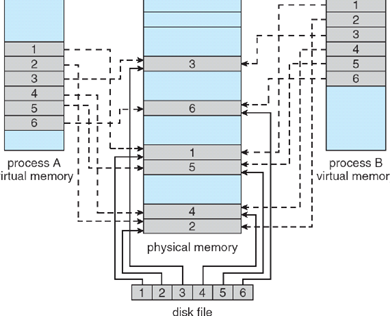
**Thrashing**

如果有一个进程**没有足够的页数**，会导致缺页率非常高，降低 CPU 的使用效率，抖动指的就是一个进程频繁换进换出的行为，当计算机的两个局部性失效的时候就容易出现抖动的现象。解决的措施就是分配更多的memory， 以及降低 degree of multiprogramming。（表现为，用于交换空间的磁盘利用率非常高,**CPU 和其他设备的利用率很低（long-term调度会放入更多的进程**），说明分配给每个进程的物理内存空间不够）

**working set**:基于局部性假设，将每一个进程在一个时间窗口内用到过的 frame建模为一个进程的 working set，如果所有进程的 working set 的大小之和大于可用 frame 的数量，那么就可能会出现 thrashing。此时操作系统可能就会选择挂起某个 process，以降低 degree of multiprogramming。working set 的思路可以用于实现 pre-paging

**内存映射文件 Memory-Mapped Files**

允许文件读写被作为一个routine memory access 来进行，可以将一个磁盘块映射到内存中的一页，一个文件一开始用按需调页的方式读入，文件的子序列的读写被作为普通的内存访问来进行，允许多个进程将同一个文件映射到内存中。简化了文件调用，不再使用read(), write() 这些system call



**内核内存分配 Allocating Kernel Memory**

内核的内存**分配和用户的不同**，通常用一个空闲的内存池，内核需要多种不同大小的内存(减少碎片)，一些内核内存必须要是连续的。因此不适用paging的方法

**Buddy System**：使用一系列物理地址连续的定长段来分配内存，并且内存的分配以2 的幂为单位。

**Slab Allocator**：Slab 分配的大概思路是预先了解到 kernel 内的常见数据结构（被称为各种 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 里。当一个 object 需要使用内存时，就查询对应的 cache 里是否有空闲的内存块，如果有就分配给它，如果没有就向 Buddy system 申请。相当于预先把内存分成不同大小，然后根据要放的东西的大小选择合适的空间来放。

**预调页 prepaging**

在被引用之前预先调如一定的页，可以减少一开始会产生的大量缺页，但是如果预调入的页没有被引用的话就会导致内存和 I/O 的浪费

**I/O interlock**

被I/O用到的帧必须在memory中

**文件系统 File System**

**文件系统接口**

文件是连续的逻辑地址空间。文件信息一般保存在directory structure里面（disk），文件在被访问之前需要先被打开, 文件访问被user access rights和file attributes所保护。文件结构分None：sequence of words, bytes；Simple record structure；按照行或者定长、变长的段来存储文件；Complex Structures：格式化文档Formatted document、可重定位加载文件Relocatable load file。权限通过访问控制列表(access control list, ACL)，但是构建 ACL 性价比低，而且原先固定长的表项可能不定长了。Unix 和 Linux 系统采用了访问权限位来实现权限控制：读(r)、写(w)、执行(x)

**文件的访问方式**

**Open(F\_i)** search the directory structure on disk for entry Fi, and move the content of entry to memory 一旦打开就返回一个**openfile table**(存储文件信息）中entry的指针实际上是reading file control information（注意，不会直接读内容）。如果多个进程共享同一个文件，则只有1个表项包含F的属性，其余为指针；不同进程关于表项内容不同；当所有进程关闭时，才会删除表项

**Close(F\_i)** move the content of entry Fi in memory to directory structure on disk

**打开文件需要**：**文件指针File pointer**，指向上次的文件读写的位置；**文件打开次数的计数器File-open count**，记录文件被打开的次数，当最后一个进程关闭文件的时候就要从一张文件记录表open-file table中删除这个文件名；文件的**磁盘位置Disk location of the file**和**访问权限Access rights**

**文件目录 Directory Structure**

目录是一种特殊的文件

- 单级目录

- 树形目录，需要额外的一位bit来记录file(0), directory(1)。采用树状的文件结构，可以防止文件名冲突 name collision

- 二级目录，主文件目录(master file directory, MFD)下为每个用户分配一个用户文件目录(user file directory, UFD) ，每个用户的目录下再存放该用户的文件

- 无环文件目录，允许了子文件和子文件夹共享，在树形目录的基础上，允许目录之间存在链接关系，链接分为软链接(soft link，快捷方式)和硬链接(hard link；复制链接文件目录项的所有元，只有当没有硬链接指向源文件的时候才能删除源文件。如果用in命令报错，则可能是因为目标文件和原始文件在不同文件系统中导致的)

**文件系统的实现**

**文件系统的层次结构**

文件系统位于二级存储中，即被存储在**磁盘**中。文件系统**挂载(mount)**是指将一个文件系统的根目录挂载到另一个文件系统的某个目录（被称为**mount** point）。只有被挂载了，一个文件系统才能被访问

文件系统都是从根目录开始的，Linux的最上层是根目录**/**，mount + [-t fstype] [-o options] device dirname

文件系统分层如下：

- 逻辑文件系统：管理各类文件的元数据，即文件系统的所有数据结构而没有实际的文件数据，根据给定的符号文件名来管理目录结构，逻辑文件系统通过文件控制块来维护文件的结构

- 文件组织模块：把逻辑块转为物理块

- 基本文件系统：对磁盘上的物理块进行读写

- I/O 控制：由设备驱动程序和中断处理程序组成，实现内存和磁盘之间的 I/O

**Superblock**:是文件系统的第一个块，存放文件系统的基本信息

**常见的文件系统**

- FAT File-allocation table 是 MS-DOS 系统使用的文件系统

- NFS 是网络文件系统， 是一种分布式的文件共享方法

- VFS Virtual File Systems 虚拟文件系统是物理文件系统和服务之间的接口，给用户和程序提供了统一的API.定义一个network-wide unique structure称为vnode. 用struct file结构， 支持multiple file systems

- ext2, ext3都是linux的

- NTFS文件系统是window10的

**可以管理空闲磁盘块的数据结构**： FAT， 位图， 空闲磁盘块链（索引节点不行）

**文件控制块 FCB**

文件控制块 (File Control Block) ，存储文件的各类元数据，包括文件的**权限**、时间信息、大小、所有者或者**数据块的指针**，打开一个文件的系统调用 open 实际上就是把对应的 FCB 调入内存中，内存中的 FCB 不包含文件名的信息

**文件目录的实现**

文件目录的检索可以采用线性检索法、哈希表法和索引等等，**线性检索法**实现简单而耗时较长，**哈希表**减少了搜索时间但是可能会引发冲突collisions和overflow。

**文件磁盘分配**

硬盘中的文件被视为 units of blocks

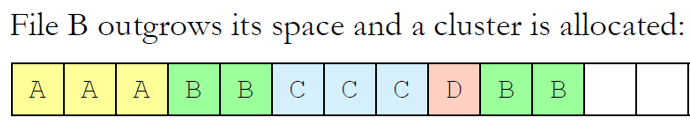
**连续分配Contiguous allocation**：每个文件分配一系列连续的磁盘块，简单，并且支持随机访问但是浪费空间，文件size不能增长，不适合多程序环境

block\_size = a B, 起始地址z

block numer = logical address//a + z

offset = logical address%a

**Extent-based** 是一种改进的连续分配方式(不属于连续分配)，分配磁盘块在extents(范围)中, 一个文件可能由一个或者多个extent。空间不够时会扩展长度



**链接分配 Linked allocation** ：每个 block 作为一个link，维护存储信息以外还需要维护指向下一个 block 的指针。简单，FCB 中只需要记录起始地址即可。不支持随机直接访问，可靠性较低

block\_size = a B, link=b B

block numer = logical address//(a-b)

offset = logical address%(a-b) + b

最大文件长度 (a-b)\*2^(b\*8) B

**索引分配Indexed allocation （Linux）**：使用索引块专门存放所有的指针，每个文件使用一个索引块，包含所有该文件所使用的块的地址，可以分为一级索引和二级索引等形式，二级索引就是用一级索引块存储二级索引的信息，索引块组成的链表被称为链接索引。好处是可以调整大小，调整块顺序不需要I/O,坏处是随机访问比较慢(访问每一级索引块算一次访问block)

文件系统有2^m个文件块，则每个索引块占m bit, 如果FCB的索引表区有f bit, 那么可支持的**最大文件长度**为 **块数\*每个文件块大小 = f/b \* 每个文件块大小**

block number和offset算法和连续分配一样

**二级最大文件长度 块数\*块数\*每个文件块大小**

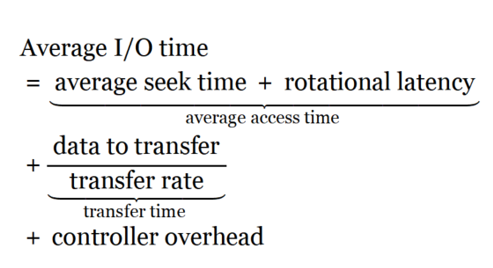
**提升文件访问速度**：提前读,延迟写,为文件分配连续的簇,采用磁盘高速缓存

**Linux 系统的文件系统**一般采用混合分配的机制，也就是以某种分配方式为主体，多种分配方式并存

**大容量存储系统 Mass-Storage Systems**

**磁盘 Disk**

磁盘是二级存储，常见的三级tertiary有CD-ROM, DVD, Tapes (注意下列公式是单个数据)

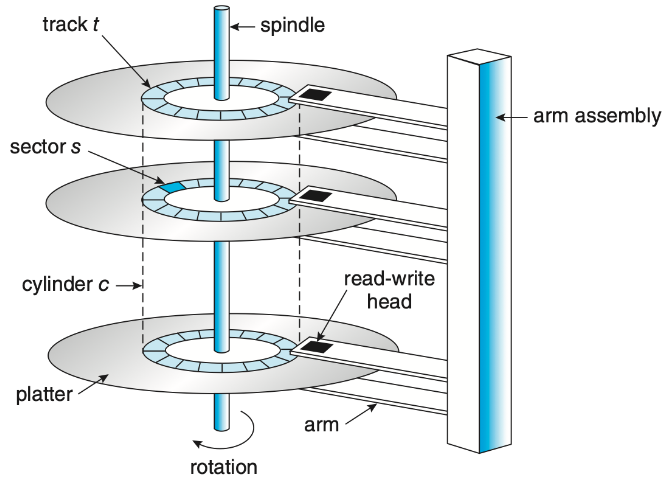


转速为 v 转/分

Rotational latency = 30/v

有时，如果有c个扇区，还要加上v/(60\*c)

扇区(sectors)、磁道(tracks)、柱面(cylinders)，侧面的磁臂(disk arm)



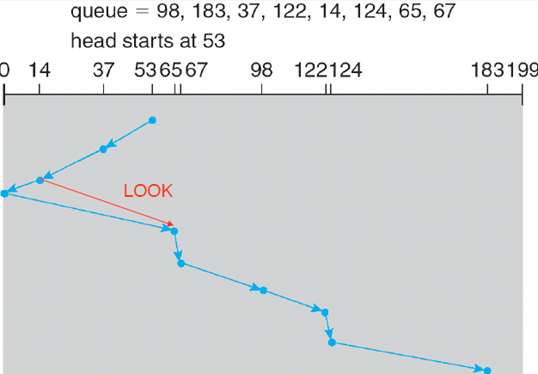
**磁盘的调度算法**

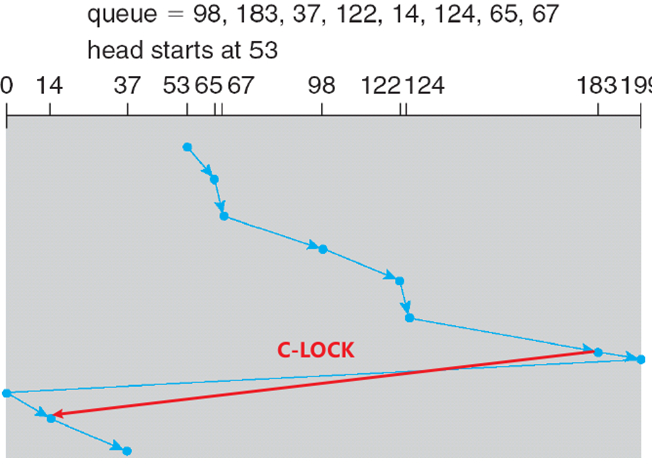
disk bandwidth = 传输数据量 / 请求开始到传输完成的时间间。一般用SSTF, 当I/O频繁则用LO0K或者CLO0K

**FCFS**，有利于**flash memory**(不需要寻道和物理延时)

**SSTF**：下一个找最近的（并不最优的）：低平均响应时间，高吞吐量；响应时间方差较大，存在饥饿，计算seek时间要额外开销

**SCAN** (elevator algorithm)：先往一个方向扫描过去，然后再反向扫描，扫描的方向取决于开始之前一次的方向，并且要先到达边界再转向：高吞吐量，响应时间方差低，平均响应时间低；由于单向移动，如果磁头刚刚经过的地方需要很久才响应。LO0K算法时对其的优化



**C-SCAN**循环扫描算法：返程也要计入距离，但是返程不做读写处理

**RAID 结构**

Redundant Arrays of Inexpensive (independent) Disks(冗余廉价磁盘阵列)，是将多个物理硬盘组合成一个逻辑磁盘，用来提供更好的存储性能和数据备份技术，RAID 提高了磁盘fail 所需的平均时间，通过冗余（性能）来提升可靠性

**磁盘的条带化**是指使用一组磁盘作为一个存储单元，RAID 机制提高了存储系统性能的表现（读写速度提升），并且通过Mirroring 或者 shadowing 来保持每个磁盘的备份

RAID0：没有冗余的条带化，提升n倍性能（n是容器个数），如果如果bit坏了都是灾难性的

RAID1：有镜像磁盘，保证安全性，提升2倍性能

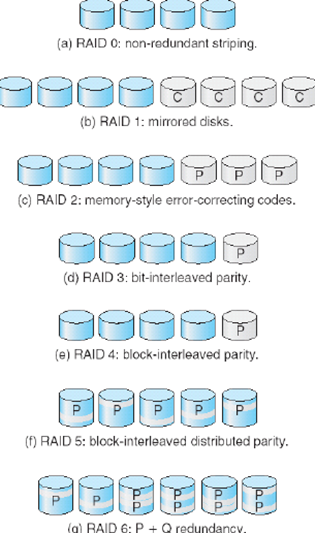
RAID2：有 Memory-style 的纠错码(ECC校验，汉明码)，只能允许单磁盘错误，并且校验码计算特别麻烦

RAID3：bit-interleaved parity 奇偶校验。只用一个盘，它的每个行的数值来存之前每行的序列的就校验码。比RAID2快并且占位小，但也只允许一个disk错

RAID4：按块进行条带化。前几个RAID都是按照bit来条带化的。RAID4方便文件读写

RAID5：RAID4基础上把奇偶校验均匀分布在每个磁盘上

RAID6: P+Q 冗余，纠错码



通常情况下会有**少量热备盘**没有被分配，当其他硬盘失效的时候就会自动替换并在上面重建数据。

**I/O 系统**

I/O被视为特殊的文件，三种 I/O 方式的区别和特点：

**轮询（pooling**是操作系统与 I/O 设备交互的最简单方式。通常设备会用某种方式来标识设备是否可用或空闲，轮询指的是 CPU 不断向设备控制器查询设备状态，直到设备就绪，然后进行数据传输。这种方式的缺点是需要进行 **busy wait**，在设备比较缓慢的情况下会降低 CPU 的利用率。对硬件设备要求最低）非常耗时，因为访问的时候设备可能还在busy，**中断**可以做到并行（中断的优点是 CPU 不需要进行 busy wait，在设备处理时间远大于上下文交换的情况下可以提高 CPU 的利用率。），**DMA** 不需要CPU就可以随机访问内存，因此效率高，耗时短（DMA 通常由独立的硬件设备 DMA controller 来实现）。注意DMA同时有两个口插在总线上,是disk control用的最多的方法。DMA时CPU不能访问内存

**提高 I/O 性能的办法**: 减少上下文切换、数据的 copy; 减少中断，使用更大的数据传输或者是更优秀的控制;使用 DMA; 平衡 CPU，内存，I/O 设备和总线的吞吐量

**buffering**用于应对I/O和CPU,设备等的一切速度差异和不同数据块大小

**caching** 缓冲与高速缓存的差别是缓冲只是保留数据仅有的一个现存拷贝，而高速缓存只是提供了一个驻留在其他地方的数据的一个高速拷贝。当内核接收到 I/O 请求时，内核首先检查高速缓存以确定相应文件的内容是否在内存中。如果是，物理磁盘 I/O 就可以避免或延迟。

**spooling**(Simultaneous Peripheral Operation On Line)假脱机技术：用来保存设备输出的缓冲，这些设备如打印机不能接收交叉的数据流。Virtual devices由spooling实现,可以把打印机这样的独享设备改为共享

**device driver**为I/O提供了统一的设备访问接口, 多个相同终端可以只使用一个共享的驱动

**块设备**：包括硬盘，一般有读写seek的命令，对其进行raw原始IO或者文件系统访问。内存映射文件访问也OK

**字符设备**：键盘鼠标串口，命令是get put。库函数提供具有缓冲和编辑功能的按行访问

**Synchronous I/O**：包括blocking IO和non-blocking I/O

阻塞IO:进程挂起直到IO完成，很容易使用和理解，但是不能满足某些需求

非阻塞IO：IO调用立刻返回尽可能多的数据。用户接口就是，接收鼠标键盘输入，还要在屏幕上输出，放视频也是，从磁盘读帧然后显示。

**Asynchronous I/O**：IO与进程同时运行。

非阻塞和异步的区别：非阻塞的read会马上返回，虽然可能读取的数据没有达到要求的，或者就没读到。异步read一定要完整执行完

**习题**

🍔文件系统用位图法表示磁盘空间的分配情况，位图存于磁盘的32～127号块中，每个盘块占1024个字节，盘块和块内字节均从0开始编号。假设要释放的盘块号为409612，则位图中修改的位所在的盘块号和块内字节序号分别是82，1(块内自己序号从0开始)

🍔On some computer, the clock interrupt handler needs 2 msec (including context switch overhead) per clocktick to execute, and the clock runs at 75 Hz. What fraction of the CPU time is devoted to the clock?

75\*2 = 150ms. This is 15% of the CPU time

🍔 假设一个三级页表的地址分为如下格式

10bit 8bit 6 bit 8 bit

如果要访问256K的进程，page table要至少多大(假设每个页表项大小为n B)？

一个page table=2^8=256 byte

1,2,3级页表各有2^10,2^8,2^6个页表项

而256KB/256B=1024 pages， 需要3级页表2^(10-6)=16张，所以1，2级各需要1张

大小为1\*(2^10)\*n+1\*(2^8)\*n+16\*(2^6)\*n