**TOPIC\_01**

操作系统：用于用户和硬件的中间层（intermediary）用于管理与协调计算机硬件

操作系统是什么

{

1. 用于用户和硬件的中间层（intermediary）用于管理与协调计算机硬件

Application programs（应用程序）：定义如何使用系统资源来解决用户的计算问题

1. Resource allocator资源分配器 管理所有资源，决定对资源的有效和公平使用的冲突请求
2. Control program控制程序 控制程序的执行以防止错误和不正确使用计算机

}

操作系统扫盲

{

操作系统是电脑启动（boots up）电脑启动第一个启动的程序（program）

操作系统是直接建立在硬件上的

操作系统的主要目的是想要让硬件的使用更高效

}

The one program running at all times on the computer” is the kernel（内核 ）Everything else is either a system program (ships with the operating system) or an application program

bootstrap program（引导程序）是启动过程（start up）在rom或eeprom里，被认为是固件（firmware），初始化系统所有方面，加载操作系统内核（kernel）并开始执行

（linux的启动过程是init）

内核扫盲

{

kernel is the program that constitutes the central core of the operating system

kernel is the first part of operating system to load into memory during booting

kernel remains in the memory during the entire computer session  
}

Each device controller has a local buffer 没个设备都有一个本地缓冲区

CPU moves data from/to main memory to/from local buffers CPU将数据从主存移动到本地缓冲区

I/O is from the device to local buffer of controller I/O从设备到控制器的本地缓冲区

Device controller informs CPU that it has finished its operation by causing an interrupt 设备控制器通过引起中断来通知CPU它已经完成了它的操作

中断通常通过包含所有服务例程地址的中断向量表（interrupt vector）将控制权转移到中断服务例程interrupt service routine（ISR）

trap(陷入) or exception（异常）是由错误或用户请求产生的软件生成的中断

An operating system is interrupt driven操作系统是中断驱动

Main memory – only large storage media that the CPU can access directly:只有CPU可以直接访问的大型存储介质

{

Random access

Typically volatile典型的挥发性

}

Secondary storage – extension of main memory that provides large nonvolatile(非易失性) storage capacity提供大容量非易失性存储器的主存储器扩展

Solid-state disks \*（固态硬盘）– faster than hard disks, nonvolatile

Caching 缓存 复制信息更快（main memory可以看作second storage的cache）

Device Driver 设备驱动程序，管理I/O，提供控制器和内核之间的统一接口

Storage-Device Hierarchy设备存储层次结构

Uniprogramming （单道程序系统）allows only one program to be present in memory at a time.

Multiprogramming (Batch system)（多道程序系统）

{

What if I/O happens? 

When it has to wait (for I/O for example), OS switches to another job. Which job to select for running if multiple ones present in the system 

One job selected and run via job scheduling（作业调度）

}

Timesharing (multitasking)（分时系统）是一种逻辑扩展，在这种扩展中，CPU频繁地切换作业，用户可以在每个作业运行时与之交互，从而创建交互计算

User mode（用户模态）mode bit=1

Kernel mode （内核模态）内核模式通常为操作系统的最低级别 mode bit=0

若用户模态想要访问内存或硬盘资源，需要通过API

硬件提供的模式位（mode bit）

提供区分系统运行用户代码或内核代码的能力

某些指令被指定为特权指令，只能在内核模式下执行

系统调用将模式更改为内核，从调用返回将其重置为用户

multi-mode operations多模式操作 例子VMM

Program is a passive entity, process is an active entity. 程序是被动实体，过程是主动实体。

Computer-system operation 计算机系统操作：一个或多个CPU、设备控制器通过公共总线连接，提供对共享内存的访问，CPU和设备的并发执行争夺内存周期

**扫盲**

**{**

MS DOS不支持多任务

How does the software trigger an interrupt ?

**Executing a special operation called system call**

**}**

**TOPIC\_02**

**User interface用户界面**

**{**

Batch interface（批处理界面）

GUI（用户图形界面）

{

对用户友好的桌面隐藏（desktop metaphor）界面

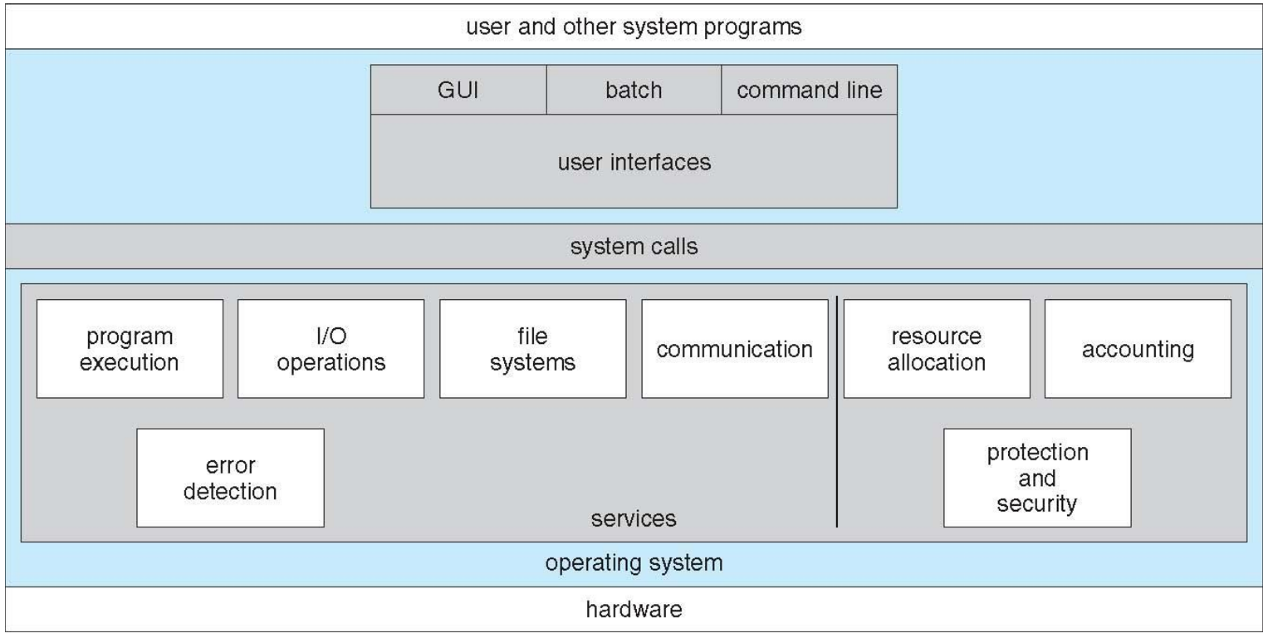
Icon 图标（代表着属性，程序等等）

Folder 文件夹（按下鼠标调用）

}

CLI 命令行界面（允许直接输入命令（有时通过内核程序实现，有时通过系统程序实现））

**}**



**command interpreter 命令解释程序（允许直接输入命令（有时通过内核程序实现，有时通过系统程序实现））**

**{**

有时是内置的（在内核里），有时是程序的名称（windows，unix都这样），如果只是程序的名称，那么添加新功能时就不用修改shell了。

功能：获取并执行用户指定的下一条命令。

如果有多个可选的命令解释程序系统时，我们称之为Shell（外壳）。

**}**

**System call 系统调用：Programming interface to the services provided by the OS（服务编程接口）To access the services of operating system, the interface is provided by system**

**Call**

**{**

**通常用C或c++写**

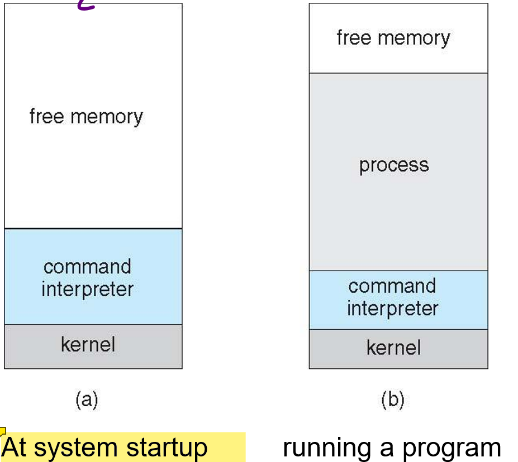
**Application Programming Interface (API) 应用程序接口** （设计程序用，相当于规定了一组函数）（通过这个来访问system call，不直接调用系统调用）

**System-call interface （系统调用接口）**截取api函数的使用，每个系统都有一个相关数字，系统调用接口维护根据这些数字编制索引的表（table），系统调用接口可调用操作系统内核中的所需系统调用，并返回系统调用状态与任何返回值

通常，**系统调入还需要参数**，我们可以通过寄存器的传递参数，参数要是比寄存器多就放把这些参数存在块（block）和表中，并块和表的地址连接到寄存器里。也可以用程序放入或压入（pushed）到堆栈（stack）中，并通过操作系统弹出（poped）

**}**

**MS-DOS是单任务，Loads program into memory, overwriting all but the kernel（就算是命令解释程序也会被进程覆盖），系统启动时invoke shell，离开时reload shell**

****

**FreeBSD是多任务**

System Programs 系统程序：为程序的开发和执行提供了环境

Simple structure – MS-DOS (在最小的空间提供最大的功能，接口和功能级别（levels of functionality）没有很好的分开)

More complex -- UNIX （受硬件的控制，早期unix有两个可分开的部分组成）

Layered – an abstraction

Microkernel –Mach管他是啥，记住

**内核**

**{**

包括系统调用接口下方和物理硬件上方的所有内容

提供文件系统、CPU调度、内存管理和其他操作系统功能；一个级别的大量功能  
**}**

Layered Approach 分层方法：底层（第0层）是硬件；最高层（第N层）是用户界面。在模块化的情况下，选择的层使得每个层都只使用较低层的功能（操作）和服务（碗状图）

Microkernel System Structure 微内核：Moves as much from the kernel into user space（从内核移动到用户空间）（mach是一个例子）MAC OS X kernel使用消息传递（message passing）在用户模块之间进行通信，损害：Performance overhead of user space to kernel space communication有开销

Modules 模块 loadable kernel modules（可加载内核模块，也叫内核扩展）和layer相似但更灵活

Linux和Solaris内核位于内核地址空间中，因此是单片的（monolithic）

System boot系统引导

{

Small piece of code – bootstrap loader, stored in ROM or EEPROM locates the kernel, loads it into memory, and starts it

通用引导加载程序GRUB允许从多个磁盘、版本、内核选项中选择内核

}

问题盲点

{

OS X -> hybrid kernel，layered

uniprocessing systems 单处理系统 只允许同一时间执行一个进程

}

**TOPIC\_03 进程**

（Batch system – jobs 批处理系统-作业

Time-shared systems（分时系统）—用户程序或任务（task））这是一个概念混淆问题，虽然在ppt也提出但估计不考

**进程-正在执行（be in execution）的程序；进程执行必须按顺序（sequential）进行**

**{**text section（文本段，即程序代码）

Program counter – location of instruction to next execute（老师说重要）

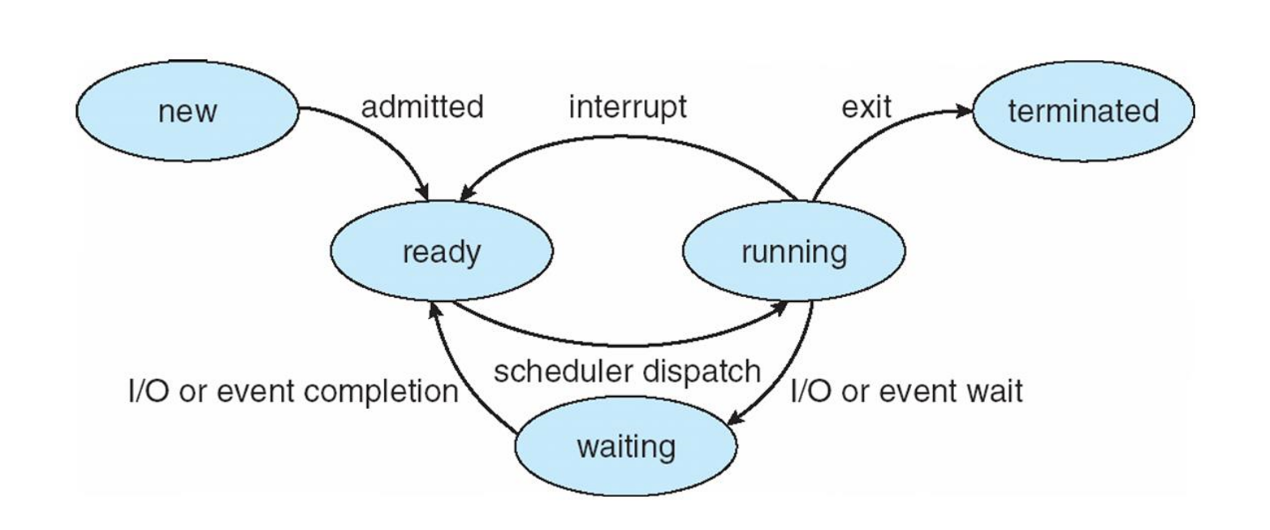
Stack（变量，返回值，参数），存的是短暂的数据

Heap

Data section（数据段，里面有全局变量）  
**}**

Program is passive entity stored on disk (executable file), process is active程序是被动的可执行文件，而进程是活动实体

state（进程）包括下面五种状态



**Process Control Block (PCB)（进程控制块）也叫task control block**

**{**

State

Process counter

CPU registers

CPU scheduling information(调度信息)（包含优先级）

Memory-management information

Accounting information

I/O status information

扫盲：PCB里有code，stack，data，但是没有Bootstrap program

**}**

Threads线程

Process Scheduling进程调度

**scheduling queues of processes**

**{**

Job queue – set of all processes in the system （进程进入系统后直接进入，这里面有所有系统进程）

Ready queue

Device queues（设备队列） – set of processes waiting for an I/O device比如打印机键盘等

**}**

Short-term scheduler（短期调度程序） (or CPU scheduler)

Long-term scheduler (or job scheduler) 选择将那些进程加入就绪队列，controls the degree of multiprogramming控制多道程序，很少被调用

I/O-bound process – spends more time doing I/O than computations, many short CPU bursts CPU-bound process – spends more time doing computations; few very long CPU bursts

Medium-term scheduler can be added if degree of multiple programming needs to decrease 移出内存，放磁盘上，需要时再调用它，我们称之为swapping，目的是降低多道程序

**当CPU切换到另一个进程时， 系统必须保存旧进程的状态，并通过上下文切换（context switch）为新进程加载保存的状态  
{**

什么时候需要上下文切换 重要！

**Multitasking** – When the CPU needs to switch processes in and out of memory, so that more than one process can be running.

**Kernel/User Switch** – When switching between user mode to kernel mode, it may be used (but isn’t always necessary).

**Interrupts** – When the CPU is interrupted to return data from a disk read.

**}**

**Process Creation非常重要！**

**{**

process identifier (pid) 进程标识符

concurrently 并发的

fork()（经常考） system call creates new process

exec() system call used after a fork() to replace the process’ memory space with a new program

父进程用abort()来终止子进程的运行

cascading termination级链终止：如果一个进程终止，那么他的子进程也必须终止

If no parent waiting (did not invoke wait()) process is a zombie 僵尸进程，子进程死了，但死的不彻底（僵尸进程会占用一个pid，但系统可以调用的pid值有限，一旦僵尸进程多了，就不能产生新进程了，所以调用wait函数可以有效地消除僵尸进程）

If parent terminated without invoking wait , process is an orphan 孤儿进程 ，子进程还没死，父进程就先死了（孤儿进程出现会被定期出现的init带走并杀死，所以影响不大）

**}**

Interprocess Communication 进程间通信

**Cooperating processes need inter process communication （进程间通信）(IPC)**

**{**

Shared memory 共享内存（常规访问，无需内核，较难实现，由用户进程而不是操作系统控制。）

Message passing消息传递(调用较少数量的信息，会使用到send(message) 和receive(message)，是消息系统和进程间的相互通讯) Establish a communication link between them创建连接

提供IPC的原因（出选择）

{  
Information sharing

Computation speedup

Modularity

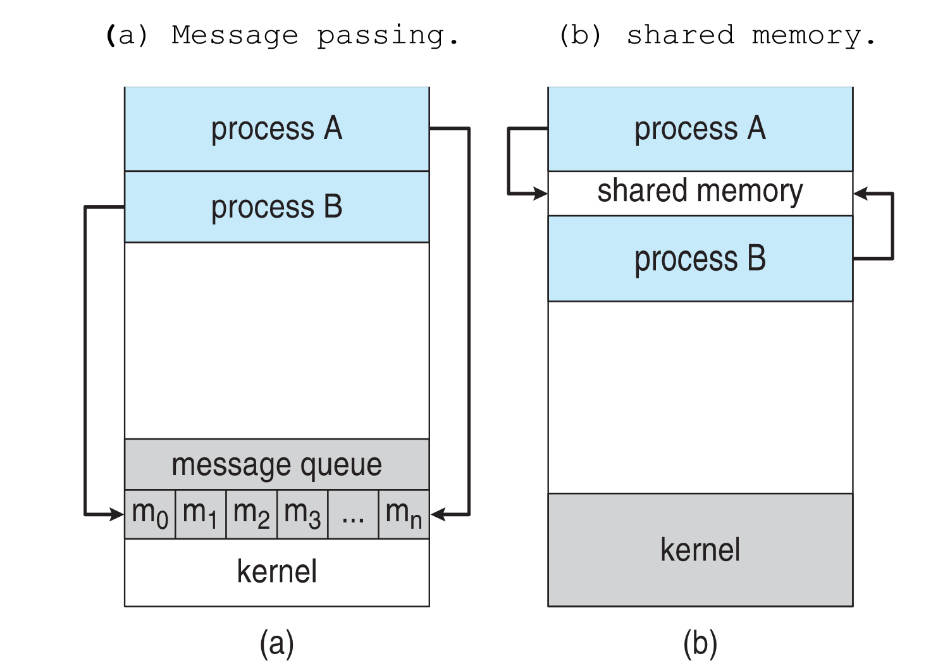
Convenience

}

**}**

无边界缓冲区（unbounded-buffer）对缓冲区的大小没有实际限制

有界缓冲区（bounded-buffer）假设有一个固定的缓冲区大小

****

Direct Communication 直接通信 自动建link，link两边是确定的两个进程，用send，receive表示想其他进程发消息从其他进程收消息，发送者是一定要指定发送谁的，接收者可以指定，也可以接受任意发送者的message，可单向（unidirectional）可双向

Indirect Communication 间接通信 通过一个抽象的对象(例：邮箱)，进程只有在共享邮箱时才能进行通信，用send，receive，表示向邮箱发东西，从邮箱收东西

Synchronization 同步

Blocking（阻塞）就被认为是同步的

Nonblocking（非阻塞）就被认为是非同步（asynchronous）

Blocking send阻塞发送 发送进程阻塞，直到消息由接收进程或邮箱接收

Blocking receive 接收进程阻塞，直到消息可用

Non-blocking send 发送进程发送消息，并恢复操作

Non-blocking receive 就单纯的收到了

如果是阻塞发送对应阻塞接收，就会引起一个rendezvous（交会）

三种队列实现方法

{

1. Zero capacity – no messages are queued on a link. Sender must wait for receiver (rendezvous) 队列中不能有消息，所以是发送者堵塞

2. Bounded capacity – finite length of n messages Sender must wait if link full

3. Unbounded capacity – infinite length Sender never waits

}

**Communication of Client-Server Systems客户端-服务器通信**

**{**

**Sockets** 套接字：通信的端点，由IP地址和端口号（port）组成（例：161.25.19.8:1625 1625为port ），成对出现，通讯的每对进程也需要一对套接字。127.0.0.1 (loopback)，这是一个特殊的地址，他引用自身

Socket in java： Connection-oriented (TCP)（面向连接） and Connectionless (UDP)（无连接）（UDP用*MulticastSocket（datagramsocket*的子类*）*）

**Remote Procedure Calls（RPC）**远程过程调用 通过客户端提供的存根（stub），调用客户端进程，每个单独远程过程都有存根，客户端存根用来定位和封装（marshal）参数，以便通过网络运输。服务器端接收并解开它（Microsoft Interface Definition Language (MIDL)（windows上的存根代码））通过External Data Representation (XDR)（外部数据表示）的方式解决大小端存储方式的问题 （操作系统会有一个matchmaker（月老)来绑定两端端口）

**Pipes**管道

**{**

Ordinary pipes普通管道 单向的，只能由创建进程外部访问，一般用于父子进程（fd【1】是写入端，fd【0】是读取端）在windows里被叫做anonymous pipes（匿名管道）

Name pipes 是bidirectional（双向的）不需要父子关系，还可以被多个进程使用

**}**

**}**

扫盲

The entry of all the PCBs of the current processes is in Process Table 进程表

}

**TOPIC\_04 线程**

Multicore Programming 多核编程

Parallelism 并行 一个系统可以同时执行多个任务

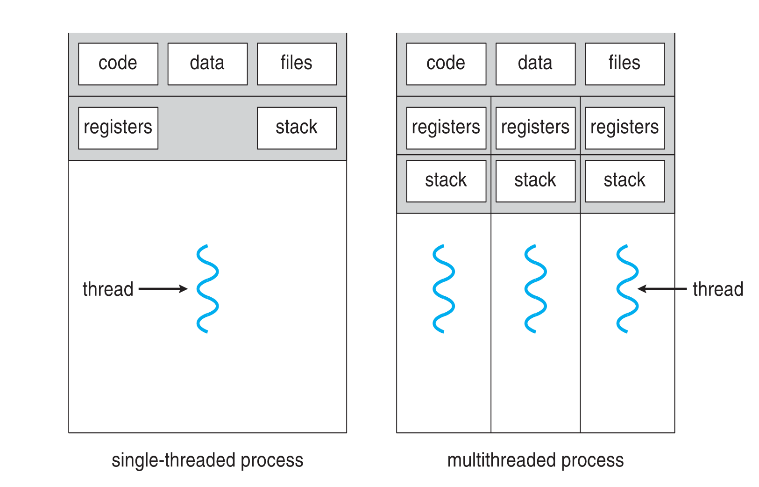
Data parallelism 将相同数据的子集分布在多个核心上，每个核心上的操作相同same

operation on each

Task parallelism 在核心上分布线程，每个线程执行唯一的操作unique operation

Concurrency 并发 支持多个任务，且每个任务都能得到进展（单处理器是并发运行，无法并行）

CPU有核心和硬件线程（hardware threads）



User Threads 用户线程 位于内核之上，无需内核支持

Kernel Threads 内核线程

Amdahl’s Law：注意Serial portion of an application has disproportionate effect on performance gained by adding additional cores

**Multithreading Models 多线程模型**

**{**

Many-to-One 缺点：Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time多线程不能再多核中，因为一个内核只能执行一个线程（并未增加并发性） 例子Solaris Green Threads和 GNU Portable Threads

One-to-One （很好的提供了并发性，但开发人员要谨慎地限制数量）例子 Windows Linux Solaris 9 and later

Many-to-Many （可以变种为双层模型（tow-level-model），双层模型允许用户线程绑定到内核线程） 例子：Solaris prior to version 9  
**}**

**Implicit Threading 隐式线程 随着线程数量的增加，程序的正确性（program correctness）在显式线程中变得更加困难，由编译器和运行时库而不是程序员创建和管理线程**

**{**

Thread Pools线程池 建一个pool里面预先放好一些线程

OpenMP ：是一组编译指令（compiler directives）和API（C，C++等），Identifies parallel regions（识别并行区域）

**}**

如果fork以后立刻调用exec，那么只需复制进程中调用fork的那一个线程，若不调用就复制进程的所有线程

Signals（信号） are used in UNIX systems，which are delivered to a process（UNIX系统中使用信号通知进程发生了特定事件。）

**signal handler 信号处理器**

**{**

default handler缺省处理程序（每个信号都有，由内核进行）Every signal has default handler that kernel runs when handling signal handler

User-defined signal handler用户定义信号处理程序（可以改写缺省）override default

**}**

Thread Cancellation 线程撤销 在线程完成之前就终止它

我们将要取消的线程叫做目标线程（target thread）

**撤销目标线程的两种情况**

**{**  
Asynchronous cancellation 异步撤销 立即撤销

Deferred cancellation 延迟撤销 不断检查是否应撤销 cancellation depends on thread state（实际上，取消取决于进程状态，只有当进程处于取消点（cancellation point）时才取消，否则会被一直挂起。达到取消点后，清理处理程序（Then cleanup handler is invoked）就会被启用）

（linux通过信号量来撤销线程）

**}**

Thread-local storage (TLS)线程本地存储 ：允许每个线程拥有自己的数据副本（当您无法控制线程创建过程时很有用）

Pending挂起 就是等着的意思

Scheduler Activations 调度器激活 用户线程库与内核线程的一种通讯方案

lightweight process (LWP) 轻量级进程 用户线程和内核线程的中间结构 用于many-to-many及其变形，每个lwp与一个内核线程相连，对于用户线程来说，它相当于一个虚拟处理器

调度器激活提供了upcall handler（回调处理程序）（会调是由内核发送到线程库）

Clone（）允许子进程分享父进程的地址空间

**TOPIC\_05 进程调度**

CPU burst（CPU执行） followed by I/O burst（I/O执行）

Nonpreemptive 非抢占的

Preemptive 抢占的

Running’to ready，waiting to running都可是抢占的的（running to terminate和running to waiting是不可抢占的）

Dispatcher 调度程序 将CPU的控制权交给短期调动程序

Dispatch latency 调度延迟：终止一个运行到启动另外一个运行所用时间

Turnaround time – amount of time to execute a particular process

Waiting time – amount of time a process has been waiting in the ready queue

**调度算法**

**{**

**FCFS** 先进先出，没有优先级 缺点：产生Convoy effect（护航效果），即一个长进程一直占用CPU，很多短进程只能等着

**Shortest-Job-First (SJF)** 谁短谁先来（可以是抢占的也可以是非抢占的，若是抢占的就遵从最短剩余距离优先（shortest-remaining-time-first）） 这里有算的方法，我赌它不考，我就不写了

**Priority Scheduling**优先级调度 （可以是抢占的或非抢占的）（会产生Starvation（饥饿）或indefinite blocking（无穷堵塞）的问题）—>解决方法：aging(老化)：逐渐增加等待时间长的进程的优先级，还有另外一个比较难的就是预测下一个CPU burst

**Round Robin (RR)** 加入time quantum（时间量，也叫时间片（slice））这个算法是抢占的（Good for response time），80% of CPU bursts should be shorter than q

**Multilevel Queue** 多级队列制度 就绪队列被分成foreground (interactive) 前台进程（交互进程）和background (batch)后台进程（批处理进程）。两个队列可以定优先级，只有高优先级的队列空了，低优先级队列才能运行，同时两个队列可以分别定自己队列的算法。

**Multilevel Feedback Queue** 多级反馈队列制度 就相当于不只分为两个队列，队列间的进程是流动的，不是固定的，详情见书

**}**

Thread Scheduling 线程调度

多对一和多对多会调度用户级线程以便可以在LWP上运行，这个方案叫process-contention scope (PCS)（进程竞争范围）

系统采用system-contention scope (SCS)（系统竞争范围）来决定那个内核级线程调度到一个处理器上（系统中所有线程的竞争）

**Multiple-Processor Scheduling多处理器调度方法**

**{**

我们主要关注多处理器中的Homogeneous processors（同构处理器）

Asymmetric multiprocessing 非对称多处理 让一个处理器处理调度等各种杂事，其他处理器全部执行用户代码（only one processor accesses the system data structures, alleviating the need for data sharing）

Symmetric multiprocessing (SMP)（对称多处理）每个处理器自我调度

**}**

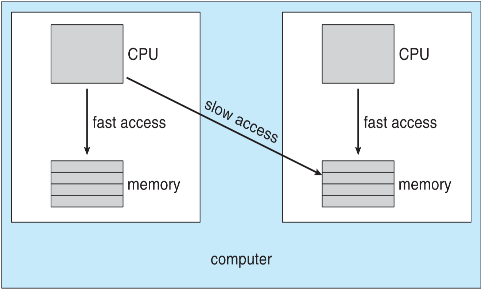
**Processor affinity（处理器亲和性）让某个进程不要一个处理器离开而去别的处理器，否则开销很大**

**{** soft affinity 软亲和性 保持进程运行在单个处理器上，但这个进程也可以迁移到别的处理器

hard affinity 硬亲和性 允许进程运行在某个处理器子集(processor sets)上

**}**

NUMA（非同一内存访问）架构 ：其中一个CPU访问内存的某些部分会比其他部分更快



NUMA与CPU调度

**Load balancing（负载平衡） keep workload evenly distributed平均分，**

**{**

Push migration（推迁移） – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs 定期检查负载，超载的处理器中的进程推走到不忙的处理器上

Pull migration（拉迁移） – idle processors pulls waiting task from busy processor 处理器空闲了，就从别的繁忙的处理器上拉正在等待的进程（推拉迁移一般并行实现）

**}**

Multicore Processors 多核处理器 在一个芯片上放多个处理器核

Real-Time CPU Scheduling实时CPU调度

Soft real-time systems软实时系统 不保证调用关键实时进程，但起码关键实时进程优先级会高一些

Hard real-time systems硬实时系统 严格一些，有截止期限，在截止期限后完成就相当于没有完成。

**影响性能的两种延迟**

**{**

Interrupt latency（中断延迟）–time from arrival of interrupt to start of routine that services interrupt 收到中断后到中断开始的时间

Dispatch latency（调度延迟）–time for schedule to take current process off CPU and switch to another 停止一个进程到启动新的进程所需时间

调度延迟的冲突阶段

{  
 抢占以在内核运行的进程

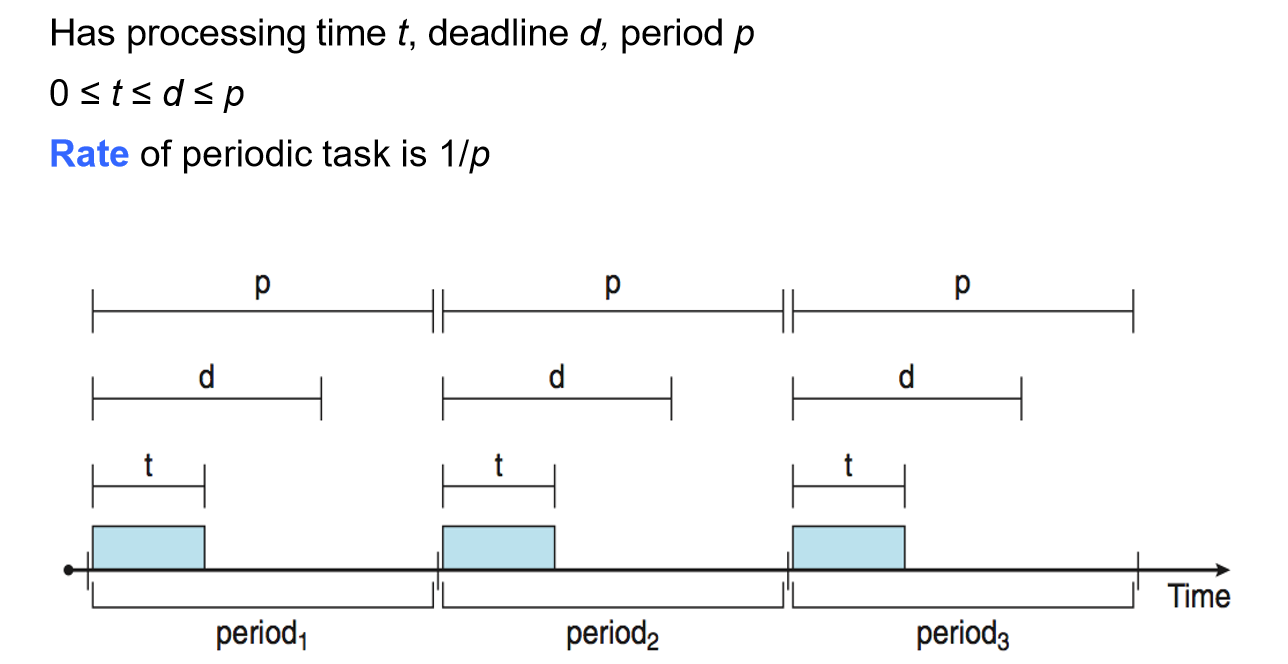
低优先级进程释放高优先级进程所需的资源

}

**}**

Priority-based Scheduling 优先权调度 我们的优先级只能满足软实时进度，我们现在想满足硬实时制度

进程是周期性的（periodic）



周期性任务调度的各种算法

{

Rate-monotonic Scheduling 单调速率调度 抢占的静态优先级 周期小的优先级高

Earliest Deadline First Scheduling (EDF)最早截止期限优先调度 截止期限越早优先级越高（优先级动态变化且是可以抢占的）

}

Linux Scheduling in Version 2.6.23 +

{

Linux系统基于调度类，每个类都有特定的优先级，，内核程序最最高优先级的调度类中

Linux标准内核实现两个调度类，采用CFS的默认调度和实时调度

Completely Fair Scheduler (CFS)完全公平调度程序，根据友好值（nice value）为每个任务分配一定比例的CPU时间, nice value（友好值）在-20—19之间，越小优先级越高

}

CFS采用target latency（目标延迟）–任务应至少运行一次的时间间隔，如果所说的活动任务数增加，则目标延迟可能会增加

CFS调度器在变量vruntime中维护每个任务的虚拟运行时间virtual run time，虚拟运行时间和基于任务优先级的衰减（decay）因子有关，优先级越低，衰减率越高

正常默认优先级产生虚拟运行时间=实际运行时间

**TOPIC\_06 Synchronization同步**

Concurrent 同时发生的

Inconsistency 不一致（数据同时发生同时发生的访问会引发数据不一致）

Race condition 竞争状况

Critical Section Problem 临界区问题

entry section

exit section

remainder section 剩余区

**临界区问题三个满足要求**

**{**

Mutual Exclusion 互斥

Progress 进步 如果临界区现在是空着的，且有进程想进入临界区，那么就会在非剩余区执行的进程中选一个进入临界区

Bounded Waiting 有限等待

**}**

**Peterson’s Solution**

**{**

分享两个数据项

int turn;

Boolean flag[2]

Turn决定哪个进程进入临界区

flag[i] = true代表着进程i已经在ready状态了

**}**

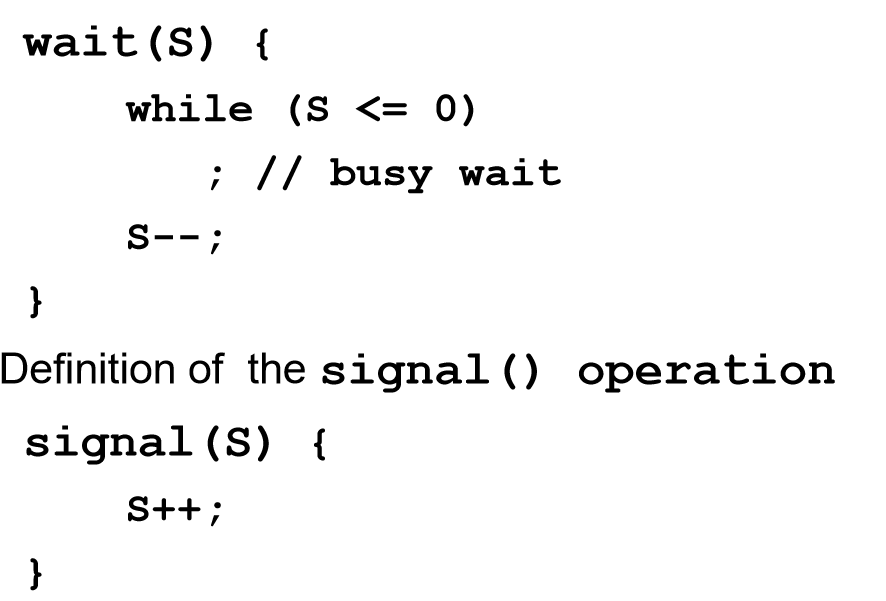
**Synchronization Hardware 硬件同步**（peterson是软件方面的solution，这个是硬件方面的）基于locking（加锁）为前提，单处理器（Uniprocessors）可以简单的解决这个问题,它禁用了中断，没有抢占，效率还高。在多处理器（multiprocessor）是不行的，因此，系统原子（atomic）的交换两个字作为不可中断（non-interruptible）的硬件指令

**Mutex Locks 互斥锁** Protect a critical section by first acquire() a lock then release() （函数中有布尔值available，为true时代表互斥锁可用，为false代表有其他进程正在用互斥锁，需要一直等待）（这两个指令必须是原子的）the lock

对这两个函数必须原子的调用。但互斥锁会出现忙等待（busy waiting）问题，这也叫自旋锁（spinlock），等待的进程会一直调用acquire函数，浪费CPU的周期。

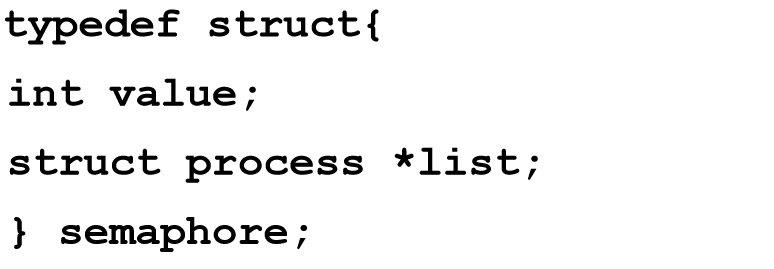
**Semaphore 信号量** 与互斥锁相同，也是一个同步工具

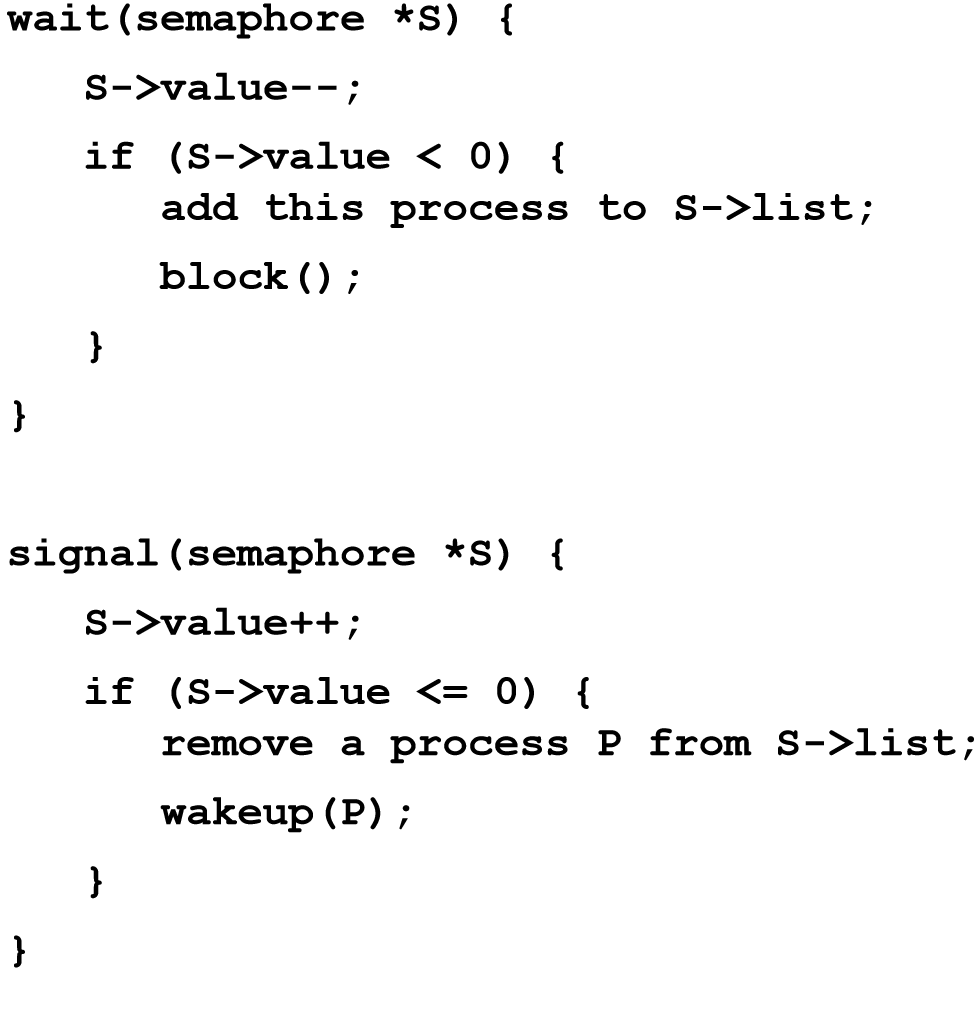
一个信号量S是一个整型变量 ，通常调用wait（）（也叫P）和signal（）（也叫V）



信号量分为Counting semaphore（计数）（可以控制访问有多个实例的资源）和Binary semaphore（二进制（只有0，1）），wait和signal都放在临界区里,这样也会busy waiting

我们可以不让信号量busy waiting而让他堵塞（block），将他放在信号量相关的等待队列里（使用block（）），等再其他的调用signal时，用wakeup（）把他从堵塞中捞回来





Suspended 暂停的

Deadlock死锁 几个进程都等着，而等着的条件是其中一个等着的进程接下来要执行的事情，这样就永远卡住了

Starvation饥饿 一直在队列里，一直不被被运行（indefinite blocking）

Priority Inversion 优先级反转

解决优先级反转问题可采用priority-inheritance protocol 优先继承协议：低优先级A有高优先级C的锁，C本来在等，但一个介于两个优先级之间的B来抢占A了，为了不让B抢，就在A持有C的所的阶段，把C的优先级暂时的继承不让B抢占。等锁放手了再变回原来的优先级

相关小问题

{

Bounded Buffer Problem 有界缓冲问题：在最开始将mutex初始化为1，full初始化为0（代表满的缓冲区数量），empty初始化为n（n个缓冲区）（代表空的缓冲区数）在生产者消费者问题中，生产者一直在生产满的缓冲区，消费者把满的缓冲区消费为空的缓冲区

Readers-Writers Problem读者—写者问题：读者只读，写者又能写又能读。

我们想让很多读者都能同时读，但作者在的时候，我们不希望其他读者或作者在读或写，否则会引起麻烦

我们初始rw\_mutex=1（rw代表读写Mutual exclusion，顾名思义，被读者和写者进程用用）， mutex=1（确保更新read\_count时的互斥），read\_count=0（有多少个读者读）

引申出 First variation（第一读者问题）和Second variation（第二读者问题）

}

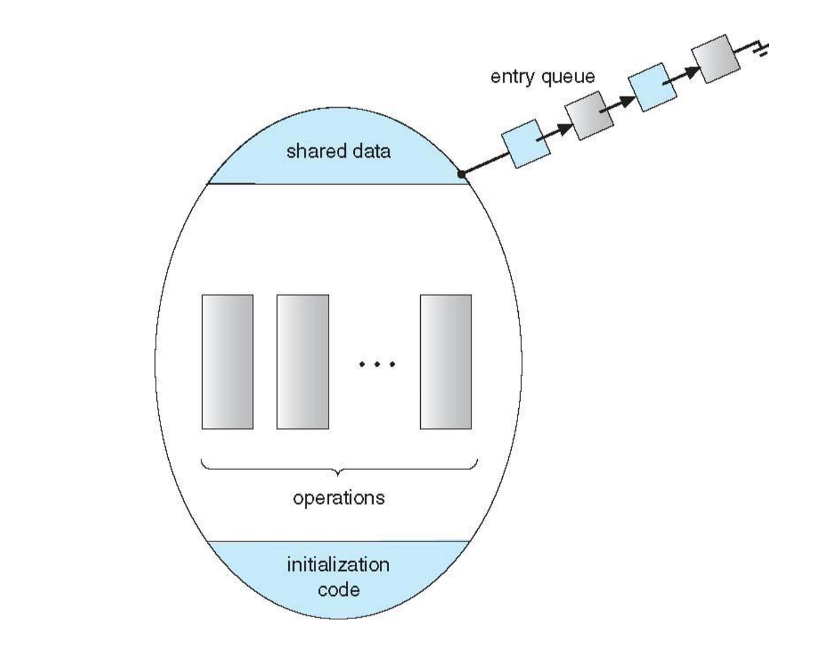
**为了防止信号量所出现的时序等种种问题，研究人员开发了一种高级语言工具——Monitor（管程）**

**{**

管程类型（type）属于Abstract data type（ADT）（抽象数据类型）

管程的局部变量只能由局部函数访问

管程确保每次只有一个进程在管程内处于活动状态



If several processes queued on condition x, and x.signal() executed, which should be resumed? FCFS frequently not adequate conditional-wait（有条件等待）construct of the form x.wait(c) Where c is priority number Process with lowest number (highest priority) is scheduled next

**}**

**TOPIC\_07 死锁Dead lock**

**{**Mutual exclusion 互斥

Hold and wait 占有并等待

No preemption 非抢占

Circular wait 循环等待**}**四个同时成立，便能引起死锁

**Resource-Allocation Graph 资源分配图**

**{**

request edge 申请边 进程P指向资源R

assignment edge 分配边 资源R指向进程P

instance 实例 一旦成环，只有一个实例就一定死锁，多个实例只能说可能死锁

**}**

**Deadlock prevention 死锁预防**

**{**

**互斥**：这个条件必须成立，没法做出改变。

**持有和等待**：**1.**我是一个资源（我有100个A），有人找我要我的资源,但只要1个，我强行把我100个资源全部给他。

**2.**我是一个进程，我只能在没有任何资源的时候才能申请资源，我有一些资源，如果我想再申请一点，我必须要先把自己的资源全部扔掉。

（两种都会降低资源利用率，引起饥饿，）

**非抢占**：我是一个进程，我申请了多个资源，但并不是全部资源都能立刻给我，那在我得到我想要的资源之前，我本来所拥有的资源都可被抢占

**循环等待** 给每个进程排序，让每个进程以递增（increasing order）的形式请求资源

**}**

Deadlock avoidance 死锁避免：使系统有先验信息（a priori information），要求每个进程说明每个类型需要资源的最大数量（maximum number），通过避免死锁算法保证永远不会有死锁地产生

**The deadlock-avoidance algorithm避免死锁算法（ensure that there can never be a circular-wait condition）**

**{**

Safe State 安全状态 如果系统能让按顺序为进程分配资源，且不构成死锁，那他就达到了安全状态。即，只有存在一个安全序列（safe sequence），系统才处于安全状态，安全状态一定不死锁，不安全状态有可能死锁。 在这一点来说，死锁避免就是让系统一直处于安全状态。（这个不是算法，是下面算法想要达到的目的）

**Resource-Allocation Graph Scheme 资源分配图算法（只适用于每种资源类型只有一个的情况）**

**{**

Claim edge 需求边（虚线）相当于未来可能成为申请边的边

request edge 申请边

Pi请求资源Rj只有在将请求边缘转换为分配边缘不会导致在资源分配图中形成循环的情况下，才能授予请求

**}**

**Banker’s Algorithm 银行家算法 每个进程需优先声明所需最大资源，如果资源不够必须等待，资源满足给予资源后要在有限时间内归还**

**{**

Safety Algorithm 安全算法**（回去多看几遍）（复杂度O(m\*n²)）**

Resource-Request Algorithm 资源请求算法

**}**

**}**

**Deadlock Detection死锁检测**

**{**

**对于每个资源有一个实例**

**{**

wait-for graph 等待图 忽略掉资源，只连进程（P1->R2->P3==P1->P3）。定期调用在图中搜索循环的算法(复杂度为n的2次方，n为顶点)。如果有循环，则存在死锁

**}**

**对于每个资源有多个实例**

**{**

Detection Algorithm 死锁检测算法（复杂度O(m\*n2)）

**}**

**}**

**Recovery from Deadlock死锁恢复**

**{**

**Process Termination进程终止**

**{**

Abort all deadlocked processes终止所有进程

Abort one process at a time until the deadlock cycle is eliminated一个一个中止，直到消除死循环

**}**

**Resource Preemption 资源抢占**

**{**

三个问题

Selecting a victim – minimize cost先择牺牲者来最小化代价

Rollback – return to some safe state, restart process for that state采用回滚 回滚到安全状态 ，让这个被抢占资源的进程重新回到队列等待分配

Starvation 确保一个进程只能被有限次的牺牲

**}**

**}**

**盲点**

**{**

For Mutual exclusion to prevail in the system at least one resource must be held in a non sharable mode 互斥要求至少有一个资源处于非共享模式（存在资源一次只能被一个进程使用）

**}**

**TOPIC\_08 内存管理 Memory Management**

Background：Main memory can take many cycles, causing a stall（暂停） Cache（高速缓存） sits between dis and CPU registers只有主存储器和寄存器可以直接被CPU访问,访问寄存器无需一个CPU时钟周期，但访问主存会需要多个CPU时钟周期，在这种情况下，如果没有足够的数据支持，**系统暂停（stall）**，对于频繁访问内存的CPU来说，这样是绝对不行的，于是我们在CPU芯片上加了**高速缓存Cache**

逻辑内存地址中有 **base （基地址寄存器）**and **limit registers（界限地址寄存器 ）**，他们指定了范围的大小

**Address Binding 地址绑定** a mapping from one address space to another

在磁盘上准备调入内存一边执行的队列叫输入队列input queue

第一个逻辑地址在0000（物理地址并不在）

源代码地址通常是符号（比如count），编译器把这些符号地址绑定到**可重定位的地址**（bind to **relocatable addresses**），链接程序或加载程序（Linker or loader）把这些可重定位地址绑定到**绝对地址（absolute addresses）**

Compile time编译时 absolute code （绝对代码）

Load time加载时 relocatable code（可重定位代码 ）

Execution time（书里叫runtime time）执行时 如果进程在执行期间可以从一个内存段移动到另一个内存段，则绑定将延迟到运行时

**地址**

**{**

Logical address – generated by the CPU; also referred to as virtual address （Logical address space为所有逻辑地址的集合）

Physical address – address seen by the memory unit

两个地址在编译和加载时绑定方式都相同，但在执行时有区别

**}**

Memory-Management Unit (MMU)内存管理单元：在execution time将虚拟地址到物理地址的运行时映射（map）

Base register now called relocation register（重定位寄存器）

MS-DOS用四个重定位寄存器

Dynamic loading动态加载 能够减少内存的使用 操作系统可以通过library来实现

Dynamic Linking动态链接：到执行时间再创建的链接 OS要检查所需的某个程序是否在某个进程的内存空间中，如果不在地址空间中，请添加到地址空间动态链接对于库特别有用 系统也称为共享库（share libraries）

Static linking 静态链接：系统库和程序代码由加载程序（loader）组合到二进制程序映像中

Stub 存根 一小段代码 Stub用例程的地址替换自身，并执行例程

Shared libraries共享库 操作系统检查例程是否在进程的内存地址中，不在地址空间就添加到地址空间

Swapping 交换：A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution

进程的总物理内存空间可以超过物理内存（因为有后被存储器backing store）能增加系统多道程序制度

Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images（大到足以容纳所有用户所有内存映像副本的快速磁盘；必须提供对这些内存映像的直接访问）

Roll out（swap out）移出去（不需要你了，先换走低优先级的）

roll in（swap in）换进来

如果下一个要放到CPU上的进程不在内存中，则需要调出一个进程并调出目标进程，不太好的是，这样所造成的上下文切换时间会比较长

交换时间的主要部分是传输时间；总传输时间与交换的内存量成正比

系统维护准备运行的进程的就绪队列（ready queue），这些进程在磁盘上有内存映像

request\_memory() and release\_memory() 通知操作系统内存需求变化情况

在交换上的其他限制

{

Pending I/O待处理I/O 该进程正在等待不能换出 解决方法：double buffering（双缓冲），但会增加开销

现代操作系统在空闲内存低于某个阈值（threshold），才启用交换（Swapping normally disabled. Started if more than threshold amount of memory allocated. Disabled again once memory demand reduced below threshold）

}

Contiguous Memory Allocation连续内存分配

**主内存**

**{**

常驻操作系统，通常用中断向量保存在低内存中

然后将用户进程保存在高内存中

**}**

将内存分为多个固定大小的分区，每个分区只包含一个进程，因此，多道程序的程度受限于分区数

kernel code being transient（暂时的）and kernel changing size

Multiple-partition allocation多分区方法

Variable-partition可变分区

Hole 可用内存块 各种大小的空穴分散在整个内存中

**Dynamic Storage-Allocation Problem动态存储分配问题**

**{**

First-fit首次适应 一旦找到足够大的hole就停止

50-percent rule 50%规则：1/3的外部碎片不可用

Best-fit最优适应 分配最小的足够大的hole

Worst-fit 直接分配最大的hole

**}**

**Fragmentation 碎片**

**{**

External Fragmentation外部碎片

Internal Fragmentation内部碎片  
**}**

通过compaction（紧缩）来解决外部碎片问题（只有重定位是动态时且重定位在execution time才可使用）

**Segmentation 分段 支持用户视图的内存管理方案**

**{**

逻辑地址由有序对（two tuple）组成 < segment-number, offset>

Segment table段表段表的每个图条目都有段基地址base和段界限limit offset超过limit就会trap in os

**}**

分段和分页都可以允许进程的物理空间是非连续的

Paging 分页（用来避免外部碎片）Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available(进程的物理地址空间可以是非连续的；只要进程可用，就为其分配物理内存) , 避免了不同大小的内存块将物理内存划分为固定大小的称为帧的块的问题

Frame 帧（backing store也被分成各个帧）

page table 页表 包含物理内存中每个页的基地址（OS为每一个进程都提供一个页表）

由CPU生成的每个地址分为Page number (p)（页码）和Page offset (d)（页偏移）

页码用做页表的索引

Page-table base register (PTBR) 指向页表基地址 需要两次内存访问

Page-table length register (PTLR) 指示页表长度 需要两次内存访问

缓解延迟（两次访问）问题的方法之一：Associative memory or translation look-aside buffers (TLBs) 转换表缓冲区：一个关联的高速内存，一些TLBs存储了address-space identifiers (ASIDs)（地址空间标识符）来唯一标识每个进程并为进程提供地址空间的保护，如果TLb不支持ASID那么每次上下文切换都要flush（刷新）

在TLB里Some entries can be wired down（有线连接）for permanent fast access

Associative Memory关联内存

Effective Access Time（EAT）有效内存访问时间 在ppt上好好看

Memory Protection内存保护 implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed

内存保护可通过Valid-invalid 有效位与无效位（不在地址空间）或使用page-table length register (PTLR)的方式

Shared Pages 共享页

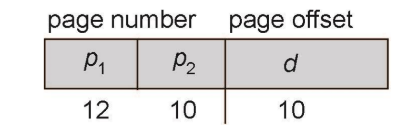
read-only (reentrant) code 可重入代码=pure code 纯代码 可以共享

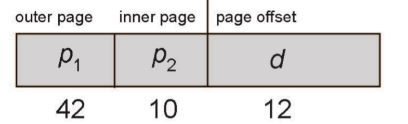
Structure of the Page Table页表结构

{

Hierarchical Paging分层分页

Two-Level Page-Table Scheme双层分页

32位

64位（不太适当大于32位还是哈希页表好一些）

也叫forward-mapped page table（前向映射页表）（由外向内）

Hashed Page Tables（哈希页表）

出现指针 （虚拟页码，映射的页码，指向链表下一个元素的指针）

clustered page tables（群集页表）是一个类似哈希页表的变体，对于（sparse）稀疏地址空间非常有用

Inverted Page Table 倒置页表：减少存储每个页表所需的内存，但增加在发生页引用时搜索表所需的时间（加入了PID值）

因为倒置页表按照物理地址来排序，但查找是根据虚拟地址，所以非常浪费时间，用哈希表可将搜索限制为一页或最多几页的表条目，通过一个虚拟地址到共享物理地址的映射来实现共享内存

}

**扫盲  
{**

**没有priority swapping这个名词（至少书和ppt没见过）**

**}**

**TOPIC\_09 虚拟内存管理**

Virtual memory 虚拟内存：用户逻辑内存与物理内存的分离（Virtual memory allows execution of a process that may not be completely in memory）

Virtual address space 虚拟地址空间：进程如何存储在内存中的一个逻辑视图

Demand Paging 请求调页 加载load time时进入内存或需要时才把页放入内存

Lazy swapper 惰性交换器 除非需要某个页面，否则不把页交换到内存中（他不是交换器，交换器要换整个程序，但他只换页面）

处理页面交换程序的是pager（调页程序）

需要的硬件支持

{

有效位 页既合法又在内存中（memory resident 内存驻留）

无效位 页无效（不在逻辑地址空间中）或有效但当前在磁盘上，对标记无效位的页面访问会产生page fault（缺页错误）

}

start process with no pages in memory的情况下，可以Pure demand paging（纯粹按需调页），即只有出现中断时才调页，一开始的也不在内存中，导致不断发生页错误

Programs tend to have locality of reference 访问局部性/局部引用：使得请求调页有合理的性能（一条指令会调多个页，从而引发多个也错误，这是个巨大的隐患，但通过实践，我们发现具有访问局部性，这种情况基本不会发生）

locality of reference 局部引用

{

时间局部性 被引用一次的储存器位置，在接下来的时间会经常 被引用

空间局部性 被引用一次的储存器位置，在接下来的时间，他旁 边的储存器位置也会被引用。  
}

请求调页的硬件支持

{

Page table with valid / invalid bit 有有效位和无效位的页表

Secondary memory (swap device with swap space) 带有交换空间的外存

Instruction restart 指令重启  
}

EAT = (1 – p) x memory access + p x (page fault overhead + swap page out + swap page in )

Copy-on-Write （COW）写时复制：allows both parent and child processes to initially share the same pages in memory，把修改的页面复制，这样就不用全部复制，节省空间

free pages are allocated from a pool（页面池） of zero-fill-on-demand pages（按需填0（目的是清除以前的内容））

pool应始终有空闲帧

vfork函数不支持写时复制，因为vfork没有拷贝空间，子进程用的就是父进程的空间

Page Replacement页面置换

多道程度增加会引起过度分配（over-allocation）

通过修改page fault服务例程以包含页面替换防止内存过度分配

使用modify（dirty）位减少页面传输的开销（overhead）

页面替换完成了逻辑内存和物理内存之间的分离-可以在较小的物理内存上提供较大的虚拟内存

Victim frame 牺牲帧

Frame-allocation algorithm帧页面算法 决定How many frames to give each process Which frames to replace

**Page-replacement algorithm 页面置换算法 达到最低的缺页次数，获得最多的页面错误率** reference string 引用串

**{**

**First-In-First-Out (FIFO) Algorithm** 在这里，不是说帧越多，缺页的次数就越少，这个莫名奇妙的现象被称为Belady’s Anomaly

**Optimal Algorithm（OPT/MIN）最优页面置换**：需要预测对比下一个最近使用，替换离下一次引用时间最远的

**Least Recently Used (LRU) Algorithm**最近最少使用算法 使用过去的历史信息，置换最长时间没有用过的页

**LRU Approximation Algorithms近似LRU的算法**

**{**

**额外引用位算法Reference bit** 引用位 每一页关联一点，初始值=0，被引用就置1

当页被引用时，位设置为1，不用，位设置为0。例：01000000比00100000更最近使用 ，11111111则代表这一页每周期最少使用一次

**Second-chance algorithm 第二次机会算法**是一种FIFO置换算法，引用值为0，直接置换，为1，就给他一次机会，不置换，同时把1置为0。具体看ppt

**Enhanced Second-Chance Algorithm增强型第二次机会算法**

增加修改位（0，1）最近没有引用但修改过

**}**

**Counting Algorithms计数算法**

**{**

Lease Frequently Used (LFU) Algorithm: 最不常使用页置换算法

Most Frequently Used (MFU) Algorithm: 最常使用页置换算法

**}**

**}**

**Allocation of Frames 帧分配**

**{**

**fixed allocation 固定分配**

Proportional allocation 比例分配

**priority allocation 优先级分配**

选择要替换的帧（从优先级较低的进程中选择要替换的帧）

**}**

Global replacement 全局置换 一个进程从所有进程集中选一个替换帧，执行时间变化大，但是throughput量也大，比较常见

Local replacement 局部置换 每个进程只从自己的进程分配的帧中选择，性能一致，但可能造成利用率不足

Non-Uniform Memory Access （NUMA）非均匀内存访问：访问的速度各不相同

最佳性能来自于“接近”调度线程的CPU分配内存

Solaris通过创建 lgroups（延迟组）解决

Thrashing 系统抖动：一个进程不停的在交换。 如果进程没有足够的页面，那么页错误率会非常高（多道程序程度如果过高会产生抖动）

Locality model 局部模型 随着进程执行，从一个局部移动到另一个局部。一个程序通常由多个不同的的可能重叠的局部组成。（Σsize of locality > total memory size便会产生抖动）

Working-Set Model 工作集模型，是局部的类似，▲最好能包含刚好一个局部

Encompass 包含

Page-Fault Frequency (PFF) 页面错误频率If actual rate too low, process loses frame；If actual rate too high, process gains frame

Memory-Mapped Files 内存映射文件 通过虚拟内存的形式，将I/O映射到硬盘里，这种情况下，其中一个进程修改器数据，其他进程也能够看得到

进程可以通过mmap（）系统调用显式请求内存映射文件，将文件映射到内核地址空间

进程仍然执行read（）和write（）

内存映射文件可用于共享内存（尽管再次通过单独的系统调用）

**Allocating Kernel Memory分配内核内存：通常从空闲池分配内存（一些内核内存是连续的，因此不受分页的控制，内核请求的内存大小不一，所以要保守的申请以防止内部碎片太多）**

**{**

Buddy System伙伴系统算法：采用power-of-2 allocator（2次幂分配器）分配内存，当所需的分配比可用的要小时，当前块将分裂为下一个2次方的两个伙伴，缺点是会引起碎片，优点是能快速合并（coalesce）

Slab Allocator

Slab是一个或多个物理上连续的页面

cache由一个或多个slab组成

每个内核数据结构都有一个cache

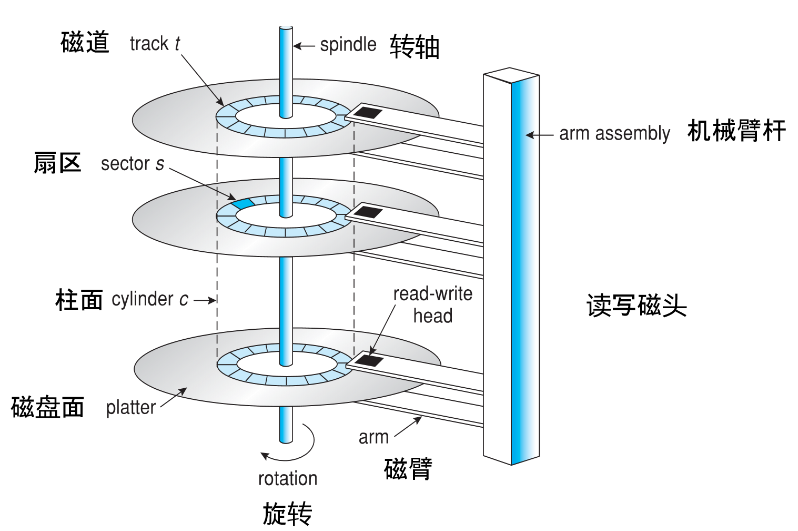
Slab里的对象最开始都是空free的，需要时就分配，其中被用到的对象就从free变成used，如果这个slab满了，就去找别的slab如果都满了，就分配一个新的slab，优点是无碎片且能够快速的满足请求。

**}**

**TOPIC\_10 大容量存储结构（书的12章）**

Magnetic disks 磁盘(提供了大量的secondary storage)（可移动）通过I/O总线（**bus**）连接到计算机的驱动器

Transfer rate 传输速率：驱动器和计算机之间的数据流的速率



**Positioning time (random-access time)定位时间（随机访问时间）**

**{**

1. Seek time寻道时间 磁臂找磁道的时间
2. Rotation latency旋转延迟 读写磁头转到想要的sector的时间

**}**

Head crash 磁头碰撞 相当于磁盘被磁头磨到了然后就可能坏

Host controller（计算机上位于总线末端的控制器）通过busses（总线）和disk controller （磁盘驱动器内置的）相交流

Disk Structure

logical blocks逻辑块 （磁盘驱动器可以看作为一维的逻辑块数组）是最小的传输单元通常为512字节

Low-level formatting 低级格式化（创造一维逻辑块数组按顺序映射到sector）（有的磁盘可以通过低级格式化选择不同的逻辑块大小）

Sector 0 最外侧cylinder

顺序：track磁道内扇区->cylinder柱面内磁道->platter从外到内的柱面顺序

Constant angular velocity 恒定角速度

Constant linear velocity 恒定线速度

**Host-attached storage 主机连接存储 通过与I/O总线通信的I/O端口访问（计算机通过一些I/O端口访问磁盘）**

**{**

**Small Computer System Interface（SCSI）（小型计算机系统接口）**本质上是个bus，一根电缆上最多有16个设备

SCSI initiator requests operation and SCSI targets perform tasks Each target can have up to 8 logical units (disks attached to device controller)（书里没有介绍，把这句话背下来）

**FC 光纤** 一个高速串行结构，也是一个大的交换结构，具有24位地址空间，是storage area networks (SANs)（存储区域网络）的基础

**}**

**Network-Attached Storage（NAS） 网络连接存储 （通过网络而不是通过本地连接提供的存储）**

**{**

NFS（Network File System）（在UNIX里）即网络文件系统 and CIFS (Common Internet File System)（在Windows里）are common protocols（我觉得了解就行）

iSCSI协议使用IP网络来承载SCSI协议

远程连接到设备（块）

**}**

**Storage Area Network（SAN）（存储区域网络）这是一个专用网络**

**{**

多个主机（host）存储到多个阵列

Storage made available via LUN Masking （逻辑单元数掩码）from specific arrays to specific servers通过LUN由具体队列到具体服务器

**}**

**Disk Scheduling，目的：最小化寻道时间，以至于能有一个尽可能快的带宽**

**{**

Seek time≈seek distance

Disk bandwidth 带宽：传输的字节总数，除以第一次服务请求到最后一次传输完成之间的总时间

FCFS先到先得

Shortest Seek Time First（SSTF）最短寻道时间优先（可能导致饥饿）

SCAN 不撞南墙不回头算法，官方叫他为电梯算法（elevator algorithm）（与电梯情况不同的是，有两个鬼一直在一楼和最后一楼按按钮以至于即使磁盘两端没有请求也要到两端再回头）系统会先定最开始走是inward方向还是outward方向

C-SAN 假设电梯最开始的方向是向上走，到达顶层后，电梯失控，直接掉到一楼，再从新开始上，然后上到顶楼在失控，掉到一楼，这样一次一次的循环扫描。

与SAN和C-SAN对应的，有look和c-look这两个算法其实是正经的电梯（没有鬼），但因为莫名其妙的原因，只有san是电梯算法（注意：在网上可能很多人把scan当look看，即不强行经过第一层和最后一层，在这里以老师讲的为准）

通常我们会选择SSTF，但负载较重时建议选择（c-）scan因为会缓解饥饿

**}**

Disk Management 磁盘管理

原始（Raw）磁盘访问OS管不到的APP

Low-level formatting, or physical formatting（低级格式化或物理格式化）在最开始的时候把磁盘分成扇区

sector 通常有512位，每个里面都有

{

header information

plus data

plus error correction code（ECC）纠错代码

}

使用磁盘保存文件前，需要想要在磁盘上记录数据结构，这需两个步骤

{

Partition分区，相当于形成各个cylinder

Logical formatting 逻辑格式化 创建文件系统（为了提高效率，大多数文件系统将块分组到簇（cluster）（簇就是一个更大的块）中，减少I/O随机访问，增加顺序访问。所以磁盘按块完成，但文件系统按簇完成）  
}

Bootstrap 自举（存储在ROM（只读存储器）中） 打开电源或重启时需要Bootstrap loader program（初始自举程序）（存储在引导分区的引导块中（固定位置））

sector sparing 扇区备用 可以处理坏块

Swap-Space交换空间—虚拟内存使用磁盘空间作为主内存的扩展

请硬记

{

4.3BSD 程序刚开始就分配交换空间,且保留文本段和数据段

Kernel uses swap maps to track swap-space use（书里没找到，硬记）

Solaris 2 allocates swap space only when a dirty page is forced out of physical memory（脏页被强制从物理内存中取出）, not when the virtual memory page is first created

}

RAID – redundant array of independent disks 独立磁盘冗余阵列（小成本，高回报）

把同样的东西放在不同磁盘的不同地方（这些磁盘都很便宜），这样的话一个磁盘损坏影响不大

mean time to failure 单个磁盘的平均故障时间得到increase（100个磁盘故障可能性肯定比一个磁盘故障可能性高）

Mean time to repair 平均修复时间

Mean time to data loss 平均数据丢失时间

RAID经常与NVRAM结合以提高写入性能

RAID Scheme（RAID方案）通过存储冗余数据来提高存储系统的性能和可靠性

快照(Snapshot)是在一组更改发生之前（即在某个时间点）文件系统的视图