Chapter 1

通用计算机系统：一个或多个 CPU 和若干设备控制器，公共总线提供了共享内存的访问。

引导程序（BootStrap Program）一般位于 ROM 或 EEPROM，它初始化系统的各组件（CPU 寄存器、设备控制器、内存内容），计算机开机后操作系统最终被加载到 RAM 中。

中断：系统发生某个异步/同步事件后，处理器暂停正在执行的程序，转去执行处理该事件程序的过程。外中断（中断）：I/O 中断、时钟中断； 内中断（异常）：系统调用、缺页、断点、程序性异常（如算数溢出等）。现代操作系统是中断（Interrupt Driven）驱动的。陷阱是软件产生的中断。

存取速度：磁带<光盘丨<硬盘<固态硬盘<闪存丨<内(主)存<高速缓存<寄存器 三级、二级和 primary。只有 Primary 是易失。

内核态：模式位（0），能够访问系统所有资源，执行特权指令（I/0、时钟、控制控制寄存器），使用内核栈。

进程同步类型：blocking(synchronous) vs nonblocking (asynchronous)

Process->CPU->core

作业池保存所有进入系统的作业，内存保存多个作业（作业池的子集），选择一个作业并开始执行，该作业可能需要等待另一个任务执行，**多道程序**时 CPU 切换到另一个作业并执行。

Dual-mode 只能访问属于它的存储空间和普通寄存器，使用用户栈

缓冲（在传输数据时临时存储数据）、缓存（将部分数据存储在更快的存储中以提高性能）

保护：控制进程或用户访问，如权限机制；安全：防御内外部攻击，如提权攻击

集群-同构，分布式-异构。

响应时间是用户发出终端命令到系统作出响应的时间间隔。进程数(用户数)为 n，每个进程的运行时间片为 q，则系统的响应时间为 S＝n×q

实时嵌入系统-固定时间约束：处理必须在约束内完成仅当满足约束时，才能正确操作

Chapter 2

系统调用与库函数的区别：API 是一组函数定义，说明了如何获得一个给定的服务；系统调用则是通过软中断向内核发出一个明确的请求。由程序通过高级应用程序接口 （API） 访问，而不是直接使用系统调用。系统调用的实现是在内核完成的，用户态函数是在函数库实现的。系统调用是进程和操作系统内核之间的编程接口。每个系统调用对应一个封装例程（wrapper routine，唯一目的就是发布系统调用）

系统调用传参方法：寄存器、内存的块或表（地址通过寄存器传递）、堆栈。

系统调用的类型：进程控制、文件管理、设备管理、信息维护、通信（消息传递/共享内存）、保护。

Background Services -Run in user context not kernel context. Known as services, subsystems, daemons

e.g.:fork,exit,wait,open,read,write,close,ioctl,getpid,alarm,sleep,pipe,shm.open,mmap,chmod,umask,chown

链接与装载：source code->compile->relocatable object file(能够装入物理内存地址)

,object file->combine through linker-> executable file

操作系统的结构：简单结构（MS-DOS、最初的 UNIX），分层方法(底层（第 0 层）是硬件;最高（N 层）是用户界面) 、微内核，宏/单内核，可加载内核模块，混合系统。比较：通信效率/修改维护

微内核的主要功能是为客户端程序和运行在用户空间中的各种服务提供通信，只有进程通信、内存管理、CPU 调度等最基本功能由微内核实现。特点是便于扩展操作系统，增加新服务不需要修改内核，在用户模式中比在内核模式中安全可靠，但频繁消息传递使性能受损。

VM：illusion that a process has its own processor and (virtual memory)

strace – trace system calls invoked by a process

gdb – source-level debugger

perf – collection of Linux performance tools

tcpdump – collects network packets

Chapter3

进程是正在执行的程序（Unit of Resource ownership & Unit of Dispatching）：包含代码、PC、寄存器、数据段、堆栈、堆。①代码段 ②数据段：存放已初始化全局变量（静态变量和全局变量）③BSS 段：包含程序中未初始化的全局变量④堆⑤栈。

new running waiting(wait for I/O or event) ready terminated

等待->就绪：等待事件发生；就绪->运行：调度，运行->就绪：时间片用完，CPU 被抢占，运行->等待：中断

进程运行时可能发生①发出 I/0 请求②时间片用完③创建子进程④等待中断

PCB： 1.Process state 2.Program counter 3.CPU registers 4.CPU scheduling information 5.Memory-management information 6.Accounting information 7.I/O status information 8.page table or relocation register and limit register 9.file open table

作业队列 = 系统中所有进程的集。就绪队列 = 驻留在主内存中的所有进程集，已准备就绪并等待执行。设备队列 = 等待 I/O 设备的进程集。

长期调度程序（long-term scheduler, job scheduler）从作业池中选择放入就绪队列的进程，其控制着程序的多道性。短期调度程序（short-term scheduler, CPU scheduler）从就绪队列中选择下一个要执行的进程并分配 CPU。中期调度程序：可将进程从内存或 CPU 竞争中移出，之后重新调入内存，并从中断处开始执行，分时系统。

上下文切换：保存正在运行的进程的状态（CPU 寄存器以及内存分配），保存程序计数器和其他寄存器的值，更新 PCB 的信息，把 PCB 移入就绪等队列，选择另一个进程执行并更新 PCB，更新内存管理的数据结构，恢复上下文。

父进程（pid>0，实际上是子进程的 pid），子进程（pid=0），-1 出错。子进程复制父进程的数据段，BSS 段，堆空间，栈空间，文件描述符，但是对于文件描述符关联的内核文件表项（即 **struct file 结构体，描述 VFS 的一个文件对象**），代码段则是采用共享的方式。父子进程资源共享：所有、部分、无；执行：同时、等待中止；地址空间：复制、加载新程序

进程可以通过系统调用 wait()返回状态值到父进程，系统调用 exit()用于终止进程。僵尸进程：当进程已经终止时，其父进程没有调用 wait（因为虽然资源被释放，但是还在进程表的条目中）。

if(pid1<0){ fprintf(stderr, "Fork Failed"); exit(-1); }

else if (pid1==0) { execlp("/bin/ls","ls",NULL); } /* child process */

else { wait(NULL); printf("child Complete"); exit(0); } /*parent process */

进程间通信（IPC）：共享内存，消息传递。管道：普通管道（单向，父子关系，又称匿名管道），命名管道（双向）。内存共享块(通信由用户控制，主要问题是提供机制，允许用户进程在访问共享内存时同步其操作)，系统调用创建(有限缓存-共享数据，但只能使用 BUFFER_SIZE-1 个元素)，一旦内存共享块在两个或更多的进程间建立，这些进程可以借助内存共享块来通信，不再需要内核的协助。信息传递通常包含系统调用，send()和 receive()操作。消息传递可以用作同步机制来处理通信进程间的行动。

Exit()输出的数据由 wait 接收，进程资源由操作系统处理

直接通信-指明 PQsend(P，message)/receive(Q，message)，链接自动建立，链接与一对通信进程完全关联，每对之间正好存在一个链接，链路可以是单向的，但通常是双向的. **间接通信**-通过邮箱（一般以端口实现）每个邮箱都有一个唯一的 ID，仅当进程共享邮箱时，才能进行通信。仅当进程共享公共邮箱时才建立链接，链接可能与许多进程相关联，每对进程可能共享多个通信链接，链路可以是单向的或双向的。允许链接最多与两个进程关联/一次只允许一个进程执行接收操作/允许系统任意选择接收器,发送方将收到接收者的通知。

消息队列：程序先调用 msgget 函数创建、打开消息队列，接着调用 msgsnd 函数，把输入的字符串添加到消息队列中。子进程调用 msgrcv 函数，读取消息队列中的消息并打印输出，最后调用 msgctl 函数，删除系统内核中的消息队列。

远程系统调用-stub-（client-side proxy for the actual procedure on the server）服务器上实际过程的客户端代理，客户端存根定位服务器并封送参数；服务器端存根接收此消息，解压缩编组参数，并在服务器上执行该过程。

Messages can be delivered exactly once rather than at most once

Chapter4

每个线程（资源的调度单位）有独立的线程 ID，程序计数器，寄存器组和堆栈。是进程内一个**执行单元或一个可调度实体**。有执行状态(状态转换)，不运行时保存上下文，有一个执行栈，有一些局部变量的静态存储，可存取所在进程的内存和其他资源，可以创建、撤消另一个线程。同一进程内的所有线程共享进程的地址空间。

资源拥有单元称为进程（或任务），调度的单位称为线程

一个线程程序的线程共享 heap memory 和 global variables，但每个线程都有属于自己的一组 registers 和 stack memory。

Stack for main thread-> idle -> Stack for thread 3,2,1,shared memory->idle->heap

用户线程（user thread）：位于内核之上，管理无需内核支持，用户线程切换不需要内核特权，一对多模型中一个线程发起系统调用而阻塞，则整个进程在等待。内核线程（kernel thread）由操作系统直接管理，一个线程发起系统调用而阻塞，不会影响其他线程。用户线程调度算法可针对应用优化；**时间片分配给线程**，所以多线程的进程获得更多 CPU 时间。

在一个 SMP 机器上：**多个线程**可以同时在不同的处理器上运行

多对一模型：映射多个用户线程到一个内核线程。优点：①不需要操作系统支持②可以调整调度策略以满足应用程序（用户级别）的需求③由于没有系统调用，因此降低了线程操作的开销；缺点：①无法利用多处理器（没有真正的并行性）②一个线程阻塞时整个进程阻塞。

一对一模型：映射每一个用户线程到内核线程中。①每个内核级线程可以在多处理器上并行运行②当一个线程阻塞时，可以调度进程中的其他线程；缺点①线程操作的开销更高②操作系统必须随着线程数量的增加而很好地扩展。

多对多模型：多路复用多个用户级线程到同样数量或数量更少的内核线程上。

双层模型：在多对多模型的基础上允许绑定某个用户线程到一个内核线程。

线程池：速度更快，限制可用线程的数量，将要执行的任务从创建任务的机制中分离出来。

轻量级进程(LWP)是建立在内核之上并由内核支持的用户线程，它是内核线程的高度抽象，每一个轻量级进程都与一个特定的内核线程关联。轻量级进程由 clone()系统调用创建，与父进程是共享进程地址空间和系统资源。与普通进程区别：LWP 只有一个最小的执行上下文和调度程序所需的统计信息。

LWP-轻量级进程（Lightweight process, LWP）：实现多对多模型或二级模型的系统在用户线程和内核线程之间通常设置一种中间数据结构，通常为 LWP。

对于用户线程库，LWP 表现为一种应用程序可以调度用户线程来运行的虚拟处理器。每个 LWP 与内核线程相连，该内核线程被操作系统调度到物理处理器上运行。

案例：win-32 ETHREAD (executive thread block)(thread start address); KTHREAD (kernel thread block)(kernel-space, scheduling and synchronization information,kernel stack); TEB (thread environment block) （user space,save thread id and user stack and storage);

Java-extend thread class and implement the runnable interface;**Linux**-clone() allows a child task to share the address space of the parent task (process)

struct task_struct，它并不保存任务本身的数据，而是指向其他存储这些数

据的数据结构（e.g. 打开文件列表、信号处理信息、虚拟内存等）的指针
concurrent execution- 其实是分时间片执行
程序执行顺序的正确性问题：**同步、互斥，**互斥是两个任务之间不可以同时运行，他们会互相排斥，必须等待一个执行完再执行，而同步也是不能同时运行，但是必须要按照某种次序来运行。
upcall：内核通知应用程序与其有关的特定事件的过程；

Chapter 5

非抢占式（Nonpreemptive）调度：①Switches from running to waiting state. ④Terminates. **抢占式**（Preemptive）调度：②Switches from running to ready state.（被抢占）RR ③Switches from waiting to ready.（主动去抢占）
原则：优先权原则、短进程优先原则、时间片原则
Dispatcher：①切换上下文②切换到用户模式③跳转到用户程序的合适位置
CPU Burst: 对长作业更好，少量长 CPU 执行； I/O Burst：对短作业更好，大量短 CPU 执行
进程调度算法评价准则：CPU utilization – keep the CPU as busy as possible/Throughout – number of processes that complete their execution per time unit **周转时间**（turnaround time）：进程从提交到完成所经历的时间；带权周转时间 W= T(周转时间)/t(CPU 执行时间)；**等待时间：**进程在就绪队列中等待的时间总和；**响应时间：**提出请求到首次被响应的时间。
响应比 R=(waitting time + execution time)/ execution time；吞吐量 Throughput ：单位时间内所完成的进程数，跟进程本身特性和调度算法都有关系；平均周转时间不是吞吐量的倒数
先来先服务（FCFS）：有利于长进程（或 CPU Bound），而不利于短进程（或 I/O Bound），平均等待时间往往很长。响应时间不稳定。
最短作业优先（SJF）：对预计执行时间短的作业（进程）优先分派处理器，是最优的，减少平均周转时间。**最短剩余时间优先（SRTF）**是基于抢占的 SJF 算法。估算下一次 CPU Burst 的长度：$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$（U-predicted value for the next CPU burst., a- 历史记录的权重，t- actual length of nth CPU burst 上一次的时间）。 有利于 I/O 密集型的作业。非抢占式会导致饥饿。不实际。
优先级调度（priority scheduling）：最小的整数 => 最高的优先级。存在无穷阻塞和老化的问题。优先级调度包括最短作业优先都有饥饿的风险，但是高响应比优先：满足短任务优先，**随着长作业等待时间增加，响应比变大，机会增大**，因此不会导致饥饿。老化是增加长期等待的进程优先级。
时间片轮转（RR）：就绪进程按照 FCFS 原则，排成队列，每次调度时将 CPU 分派给队首进程，让其执行一个时间片（time slice）。每个执行完 time slice 后到队尾。
提高进程并发性和响应时间特性，从而提高资源利用率。其不可能导致饥饿现象。RR 的平均周转时间比 SJF 长，但响应时间要短。进程结束后有新的进程产生，挂在该进程前面。
a small unit of CPU time (**time quantum**)-each process gets 1/n of the CPU time in chunks of at most q time units at once. No process waits more than (n-1)q time units.
q must be large with respect to context switch, otherwise overhead is too high
多级队列调度：每个作业归入一个队列，不同队列可有不同的优先级、时间片长度、调度策略等。多级反馈队列算法允许进程在队列中迁移。
foreground (interactive) 前台（交互式）-RR；background (batch) 后台 （批处理）-FCFS
多级反馈队列和 RR 调度算法相似——它们不会先选择短任务。
设置多个就绪队列，分别赋予不同的优先级，队列 1 的优先级最高。每个队列执行时间片的长度也不同，规定优先级越低则时间片越长。新进程进入内存后，先投入队列 1 的末尾，按 FCFS 算法调度；若按队列 1 一个时间片未能执行完，则降低投入到队列 2 的末尾，同样按 FCFS 算法调度；如此下去，降低到最后的队列，则按"时间片轮转"算法调度直到完成。仅当较高优先级的队列为空，才调度较低优先级的队列中的进程执行。若新进入的进程进入了比当前执行进程优先级高的队列，停止并把被抢占的进程投入原队列末尾。为提高系统吞吐量和缩短平均周转时间而照顾**短进程。**为获得较好的 I/O 设备利用率和缩短响应时间而**照顾 I/O 型进程。**I/O 型进程：让其进入最高优先级队列及时响应 IO 交互；I/O 次数不多，I/O 完成后放回一开始的队列以免主次下降。
高响应比优先调度算法 Highest Response Ratio Next(HRRN)既考虑作业的执行时间也考虑作业的等待时间，综合了先来先服务和最短作业优先两种算法，把 CPU 分配给就绪队列中**响应比**最高的进程。
调度标准中的冲突: a.CPU 利用率和响应时间 b.平均周转时间和最大等待时间 c.I/O 设备利用率和 CPU 利用率
线程调度：
Local Scheduling – How the threads library decides which to put onto an available LWP
Global Scheduling – How the kernel decides which kernel thread to run next
turnaround time = termination time – arrival time
wait time= turnaround time – burst time.
Pthread
int pthread_create(pthread_t *restrict tidp,const pthread_attr_t *restrict attr, void *(*start_rtn)(void),void *restrict arg);
当创建线程成功时，函数返回 0，若不为 0 则说明创建线程失败
int pthread_join(pthread_t thread, void **rval_ptr);
返回值: 成功则返回 0, 否则返回错误编号.
参数:thread: 线程 ID;rval_ptr: 指向返回值的指针(返回值也是个指针).
隐私多线程：将线程的创建与管理交给编译器和运行时库来完成，thread-pool/fork-join parallelism/OpenMP/Grand Central Dispatch/TBB

Chapter 6

临界资源：硬件打印机、磁带机等，软件的消息缓冲队列、变量、数组、缓冲区等；临界区：访问临界资源的那段代码，shared data(对共享数据的操作可能会导致数据的不一致性，需要机制确保进程的有序执行。)
临界区解决方案必须满足：①互斥（mutual exclusion）②空闲让进（progress）想进就进 ③有限等待（bounded waiting）
Two process solution：
Peterson Solution: 不会产生死锁，也不会产生饥饿现象

P0:
do { flag[0]= true; turn = 1;
while (flag[1] and turn = 1) ;
critical section
flag[0] = false;
remainder section } while (1);
	P1: do {
	flag[1]= true;
	turn = 0;
	while (flag[0] and turn =0) ;
	critical section
	flag[1] = false;
	remainder section } while (1);

n processes：Bakery Algorithm: 在进入其关键部分之前，进程会收到一个数字。最小数字的持有人进入关键部分。编号方案始终以增加枚举顺序生成数字。解决了饥饿问题。
choosing[i]=true，表示进程 i 正在获取它的排队登记号，保证计算 number[i] 值的原子操作；
number[i]是进程 i 的当前排队登记号。如果值为 0，表示进程 i 未参加排队，不想获得该资源。
Synchronization Hardware: ①TestAndSet ②Swap
Special machine instruction result in busy-
Semaphores: 当进程发现信号量不为正时，选择阻塞自己而不是忙等待（让权等待！）。 wait-P signal-V
Bounded-Buffer Problem：假设缓冲池有 n 个缓冲区，mutex=1; empty=n; full=0;

do {
...
produce an item in nextp ...
wait(empty);
wait(mutex);
...
add nextp to buffer ...
signal(mutex);
signal(full);
} while (1);
	wait(full); wait(mutex);
	...
	remove an item from buffer to nextc ...
	signal(mutex);
	signal(empty);
	...
	consume the item in nextc ...
	} while (1);

Reader-Writer Problem: rw_mutex = 1; mutex = 1; read_count = 0;

共享变量 mutex = 1
do {
wait(mutex);
Rritical Section
signal(mutex)
Remainder Section;
} while (1);
	do {
	wait(mutex);
	readcount++;
	if (readcount == 1)
	wait(wrt);
	signal(mutex);
	...
	reading is performed ...
	wait(mutex);
	readcount--;
	if (readcount == 0)
	signal(wrt);
	signal(mutex);
	} while (TRUE);

Dining-Philosophers Problem：chopstick[i]=1

```
do {
wait(chopstick[i]) wait(chopstick[(i+1) % 5])
...  eat ...
signal(chopstick[i]); signal(chopstick[(i+1) % 5]);
...  think ...
} while (1);
```

Semaphore :s1,s2,s3,s4,s5:=0;

P1:{ Input(a) siginal(s1) wait(s2) x=a+b wait(s5) print(x,y,z)}	P2:{ wait(s1) Input(b) siginal(s2) siginal(s3) y=a*b siginal(s4)}	P3:{ wait(s3) input(c) wait(s4) z=y+c-a siginal(s5)}

基于 TestAndSet 的算法：
boolean TestAndSet(boolean &target) {
boolean rv = target;
target = true;
return rv; }
进程共享变量 boolean lock = false
对于第 i 个进程
do {
while (TestAndSet(lock)) ;
	基于 Swap 的算法：进程共享变量 boolean lock = false 和 boolean waiting[n] 对于第 i 个进程
	do {
	key = true;
	while (key == true) Swap(lock,key);
	Critical Section
	lock = false;
	Remainder Section

Critical Section
lock = false;
Remainder Section
}while(1);
分析：满足了互斥，但不满足有限等待条件。假设系统调度不幸使得某个号数的进程每次都如下述情况运行，则这个号数的进程会一直霸占进入临界区的机会，使得其他号数的进程无限等待。
lock = false;
Remainder Section
}while(1);
do {
while (TestAndSet(lock)) ;
Critical Section
	} while(1);
	分析：满足了互斥，但不满足有限等待条件。假设系统调度不幸使得某个号数的进程每次都如下述情况运行，则这个号数的进程会一直霸占进入临界区的机会，使得其他号数的进程无限等待。
	lock = false;
	Remainder Section }
	do {
	key = true;
	while (key == true) Swap(lock,key);
	Critical Section

race condition： 多个进程同时访问和操作共享数据的情况。共享数据的最终值取决于最后完成的进程。故结果取决于访问发生的特定顺序。->必须同步并发进程

进程之间竞争资源:互斥，死锁，饥饿

并发执行->竞争资源->互斥；并发执行->合作->同步

wait(S)≡P(S) ≡ down(S) ： 表示申请一个资源

signal(S)≡V(S) ≡ up(S) ： 表示释放一个资源

一个同步 wait 操作与一个互斥 wait 操作在一起时，同步 wait 操作在互斥 wait 操作前。

typedef struct {int value; struct process *list; } semaphore;

block() – place the process invoking the operation on the appropriate waiting queue.

running → waiting

wakeup() – remove one of processes in the waiting queue and place it in the ready queue.

waiting→ready

wait(semaphore *S) {
S->value--;
if (S->value < 0) {
add this process to S->list;
block();
} }
	signal(semaphore *S) {
	S->value++;
	if (S->value <= 0) {
	remove a process P from S->list;
	wakeup(P);
	}}

semaphores -mutual exclusion、synchronization

Binary semaphore（二值信号量） /Mutex locks

Context-switch

In general, the operating system must save the state of the currently running process and restore the state of the process scheduled to be run next. Saving the state of a process typically includes the values of all the CPU registers in addition to memory allocation. Context switches must also perform many architecture-specific operations, including flushing data and instruction caches

Busy waiting is where a process checks repeatedly for a condition- it is "waiting" for the condition, but it is "busy" checking for it. This will make the process eat CPU (usually).

Chapter 7

死锁必要条件：Mutual exclusion 互斥-某个时间一个资源只有一个进程能使用； hold and wait 占有等待-已经拥有至少一个资源的进程在等待获取被其他进程持有的额外资源 No preemption：不可抢占，只有进程完成任务后自愿进行资源的释放。Circular wait 循环等待

（死锁预防）circular wait：按顺序拥有资源。With A Cycle But No Deadlock-如果每个资源只有一个 instance 则为 deadlock，但如果每个资源不止有一个实例不一定是死锁

系统资源分配图：申请边 Pi→Rj， 分配边 Rj→Pi

1. 对于共享资源不设互斥限制，对于非共享资源保持互斥。

2. 要求进程在开始执行之前请求和分配其所有资源，或者仅允许进程在进程没有资源时请求资源。（资源静态配置方式）

3. 如果请求的被另一个持有，则自己释放所有资源，同时被抢占的资源回到该进程的等待列表中，最后只有所有资源都能获得时重启

4. 资源的有序申请。为每个资源分配一个序号，所有进程必须按照升序选择资源。

（死锁避免）每个进程声明可能需要的每种类型的最大资源数。

动态检查资源分配状态，以确保永远不会出现循环等待条件。状态由 可用的资源、被分配的资源、进程需要的最大值 决定。

每个资源只有一个实例：资源分配图算法：假设 Pi 申请 Rj，只有将申请边变成分配边并且不会导致分配图形成环时，才允许分配。

（死锁避免）**资源有多个实例：银行家算法**

安全算法：

1. work 初始为 Available， Finish[i]=false
2. Find an index i Finish[i]=false Need_i<=Work,若不存在跳到 4
3. work=work+allocation_i Finish[i]=true.回到第二步
4. 如果所有的 i，finish[i]==true，系统处于安全模式。（work 是当前所有可用的资源）

检查 need,检查 available，计算：

Available = Available – Requesti，Allocationi = Allocationi + Requesti，Needi = Needi – Requesti

If safe Þ the resources are allocated to Pi

If unsafe Þ Pi must wait, and the old resource-allocation state is restored

死锁检测：wait for graph，结点是 process

死锁恢复：打破死锁：进程中止（打破一个环或所有环中元素）/抢占资源（找一个替罪羊/回退）

死锁避免因追踪当前资源分配增加了运行成本，但比死锁预防允许更多并发，死锁避免可以增加系统吞吐量。

死锁检测假设资源分配方面和进程请求资源的过程中不存在循环等待。

Chapter 8 & Chapter 9

物理内存 frame，逻辑内存 page，块大小相同

逻辑地址：程序在编译后，每个目标模块都从 0 号单元开始编址，该相对地址称为逻辑地址；**线性地址**：逻辑地址加上相应段的基地址，若没有启用分页机制，那么线性地址直接就是物理地址，如果启用了分页机制，则需要再变换一次产生物理地址。

address binding 三种阶段发生：编译时生成绝对代码，加载时生成重定位代码（relocatable),执行时绑定。

MMU：用户进程中的逻辑地址被加上了 relocation register 重定位寄存器。

SWAP：可以将进程暂时从内存中交换到后备存储，然后重新放入内存以继续执行，时间的主要部分是转移时间;总传输时间与交换的内存量成正比。当空闲内存低于某一个阈值时，启用交换换出，当空闲内存增加一定数量时，停止换出。

源程序→编译→链接（形成逻辑地址）→加载（静态重定位）→执行（动态重定位）

静态重定位：作业装入内存时，必须一次性分配给要求的全部内存空间，一旦进入内存，整个运行期间就不能在内存中移动。对重定位的存储管理方式，整个系统只需一个重定位寄存器。

分区存储管理：固定分区（fixed partition）可以采用静态重定位，不会产生外部碎片（external fragmentation），会产生内部碎片；可变分区（variable partition），不会产生内部碎片（internal fragmentation），会产生外部碎片。在装程序时切分内存，需要保存已分配的分区和空闲分区的信息

碎片-内部-一个分区中的 unused，外部-Total memory space exists to satisfy a request, but memory is not contiguous ->compaction/defragmentation（Compaction is possible only if relocation is dynamic and is done at execution time）

分段式存储管理：在编译用户程序时，编译器会根据程序自动构造段，有利于程序的动态链接；会产生外部碎片，不会产生内部碎片；段中可重入代码可以共享。段的基地址寄存器+段长度寄存器。

分页式存储管理：会产生内部碎片，不会产生外部碎片，但是每个进程平均只产生半个块大小的内部碎片；页表中引入有效位（valid bit）和脏位（dirty bit）Linux(global directory/middle directory/page table/offset)

分段：CPU(logical address)->segmentation unit(linear address)->paging unit(physical address)->physical memory

支持动态内存分配的要求：连续内存分配:当没有足够的空间给程序去扩大它已分配的内存空间时，将要求重新分配整个程序；纯段式分配：当没有足够的空间给段去扩大它的已分配内存空间时，将要求重新分配整个段；纯页式分配：在没有要求程序地址空间再分配的方案下，新页增加的分配是可能的。段/页分配允许共享代码，连续不行。

TLB 有效内存访问时间EAT = p * (t + ma) + (1 − p)(2ma)

缺页处理：1.Operating system looks at another table to decide:(1)Invalid reference→abort (2)Just not in memory 2.Get empty frame 3.Swap page into frame 4.Reset tables 5.Set validation bit = v 6. Restart the instruction that caused the page fault 非法则舍弃，合法找到空的物理帧，交换页，重置页表，validation bit=v，重做指令。

Page Fault Rate = p. EAT = (1 − p) * ma + p * (缺页错误时间)

计算：32-bit 意思是一个地址的大小，4K page size 表示 page offset 为 12bit, 所以 page number 为 20 bit

页表项:

物理块号，状态位 P（用于指示该页是否已调入内存，供程序访问时参考），访问字段 A（用于记录本页在一段时间内被访问的次数，或最近已有多长时间未被访问），修改位 M（R/W：表示该页在调入内存后是否被修改过。），外存地址。

操作系统内核(4GB)->环境变量->参数->对战->用户区（空闲）->数据->代码（0)

Copy-on-Write（进程创建） allows both parent and child processes to initially share the same pages in memory until either process modifies. More efficient process creation. 硬件要求：内存访问需要检查是否该页表是写保护，若是，trap，os 处理。

页面替换算法：查找所需页面在磁盘上的位置；查找空闲帧，如果没，使用页面替换算法选择受害者 帧，将受害页写入磁盘；相应地更改页面和帧表。将所需的页面放入（新）帧;更新页面和帧表；重新启动进程

目标：lowest page-fault rate

First-In-First-Out Algorithm：先进先出 **Belady's Anomaly**

Optimal Page Replacement OPT/MIN：“在离当前最远位置上出现的”页被置换。

Least Recently Used： 选择内存中最久没有引用的页面被置换，左边第一个最远

LRU-Approximation Page Replacement: 参考位：页表项的 A 位，初始 0，When page is referenced bit set to 1，代替是 0 的

Additional Reference Bits：Use right-shift history byte for each page, current reference bit shift into the left-most bit. choose the page with lowest number.

Second-Chance Algorithm：FIFO + reference bit. Circular Queue. 如果要替换的页面（按时钟顺序）具有参考位 =1，则：设置参考位 0

Enhanced Second-Chance Algorithm：使用有序对(引用位,修改位)，淘汰的顺序为(0,0) (0,1) (1,0) (1,1)

Counting-Base Page Replacement：（保留对每页引用次数的计数器），LFU，MFU

Page Buffering Algorithm（页面缓冲算法）：通过被置换页面的缓冲，有机会找回刚被置换的页面——空闲页面和已修改页面，仍停留在内存中一段时间

帧分配：固定（equal/proportional），priority(高优先级进程缺页时：1.替换他的帧 2. 替换低优先级进程的帧)

Global replacement: 流程从所有帧集中选择替换帧;一个进程可以从另一个进程获取帧 **Local replacement:** 每个进程仅从其自己的分配帧集中选择

组合方式：固定分配-局部置换，可变分配-局部/全局置换

Thrashing - a process is busy swapping pages in and out

一个进程无法获得足够多的页，频繁缺页导致:低 CPU 利用率->误导 OS 认为提高多任务的程度->内存中主流更多进程->每个进程拥有的页帧数更少->恶性循环

多道程序程度越大，CPU 利用率先升后减。

根本原因：sun(size of locality)> total memory size

内存映射文件 I/O 允许通过将磁盘块映射到内存中的页面将文件 I/O 视为常规内存访问

Buddy System: Allocates memory from fixed-size segment consisting of physically contiguous pages.

Slab Allocator: 每个数据结构有自己单独的 cache,一个 cache 包含多个 slab,slab 是一个或多个物理上连续的页。Slab is one or more physically contiguous pages. Cache consists of one or more slabs. Single cache for each unique kernel data structure. When cache created, filled with objects marked as free. When structures stored, objects marked as used. If slab is full of used objects, next object allocated from empty slab. If no empty slabs, new slab allocated. Benefits include no fragmentation, fast memory request satisfaction.

首次适应和最佳适应在执行时间和空间利用方面都好于最差适应。首次适应要更快一些。

TLB Reach - The amount of memory accessible from the TLB= (TLB Size) X (Page Size)

I/O interlock 有时必须将页面锁定到内存中，比如读取设备文件的页绝对不能被替换。

Windows XP--Processes are assigned working set minimum and working set maximum;clustering. Clustering brings in pages surrounding the faulting page

solaris-Maintains a list of free pages to assign faulting processes;Paging is performed by pageout process

Chapter 10 ~12

文件属性：Name;Identifie;Type;Location;Size;Protection;Time, date, and user identification;

Open(Fi) – search the directory structure on disk for entry Fi, and move the content of entry to memory **RAID**-独立冗余磁盘阵列

每个打开的文件具有如下信息：

File pointer: 文件指针对操作文件的每个进程是唯一的,File-open count

Disk location of the file: cache of data access information,Access rights: **per-process** access mode information

访问方法： Sequential Access; Direct Access; Indexed Sequential-Acess

NFS is standard UNIX **client-server file sharing protocol**

CIFS is standard Windows protocol

Directory 需要包含所有文件的信息，组织时注意：**efficiency/naming（不同用户可以对不同文件有相同命名）/grouping（逻辑上组织文件利用属性）**

硬链接相当于一个指针，指向文件的索引节点，系统不会增加 inode 节点。而符号链接类似于 Windows 的快捷方式，是不同的文件，数据块中存放原文件的路径。建立符号链接时引用计数直接复制，建立硬链接时，引用计数加 1。

逻辑文件系统：管理元数据（指文件系统的所有结构数据），管理目录结构，通过 FCB 来维护文件结构。

文件组织模块：知道文件及其逻辑块和物理块，空闲空间管理器。

基本文件系统：向合适的设备驱动程序发送一般命令就可对磁盘上的物理块进行读写。

打开文件的过程：系统调用 open 将文件名传递到逻辑系统，搜索系统范围内的打开文件表以确定某文件是否已被其他进程所使用。如果是，就在单个进程的打开文件表中创建一项，并指向现有系统范围的打开文件表。如果否，根据给定的文件名搜索目录结构，部分目录结构通常缓存在内存中以加快目录操作找到文件后，其 FCB 会复制到内存的整个系统开放文件表中。

连续分配(Contiguous allocation): 所需寻道时间最短，可随机访问，但是浪费空间，且文件不能增长。访问第 n 条记录需要访问 1 次磁盘。

链接分配(Linked allocation): 每个目录条目都有文件首个磁盘块的指针，没有空间浪费，但只能顺序访问文件且可能丢失指针。访问第 n 条记录需要访问 n 次磁盘。变种 FAT

索引分配(Indexed allocation): 将所有的指针放在索引块中，索引块的第 i 个条目指向文件的第 i 个块。目录包含索引块的地址。

链接索引(Linked scheme)，索引块最后一个指针指向下一个索引块

多级索引(Multi-level index)，以二级索引为例，一级索引块中每一个指针对应的二级索引块。m 级索引需要访问 m+1 次磁盘。

一级：1 个索引，因为最后一个不能用，要指向下一个 index table。LA/(512*511)Q 获取第几个 index

组合方案：UNIX 索引块的前几个指针存在 iNode（索引结点）中，前 12 个指针指向直接块，接下来 3 个指针分别指向一级、二级、三级索引块。

分层设计：逻辑文件系统管理元数据（FCB）->文件组织模块（空闲空间管理、文件及其逻辑、物理块）->基本文件系统（读写物理块）->IO 控制（设备驱动程序和中断处理程序）

位向量(bit map)：块空闲为 1，已分配为 0。按字查找第一个空闲块号公式为 (每个字的位数)*(字数)+第一个值为 1 的偏移。盘块分配时，盘块号 b=n*(row-1)+col，盘块回收时 row=(b-1)/n+1, col=(b-1)%n+1

position time = seek time + rotational latency；Disk bandwidth=byte/time

磁盘调度算法：①FCFS 比较公平②SSTF（最短寻道时间优先）：处理距离当前磁头位置最短寻道时间的请求，性能比 FCFS 好③SCAN：从磁臂的一端移动到另一端，到达另一端后反转④C-SCAN：到达另一端后直接返回，不处理回程请求⑤LOOK：只移动到一个方向上的最远请求为止

低级格式化(Low-level formatting, physical formatting): 将磁盘分成扇区以便控

制器读写，每个扇区的数据结构通常由头部、数据区域和尾部组成，头部和尾部通常包括了扇区号、纠错代码。**逻辑格式化(Logical formatting)**:建立文件系统的根目录，将初始文件系统的数据结构存储到磁盘上。

NFS-unix file-system interface->VFS->NFS service layer， External Data Representation (XDR) protocol，A remote directory is mounted over a local file system directory

磁盘 I/O 按块完成，文件系统 I/O 按簇完成。

BootStrap 程序的作用是从磁盘上调入完整的引导程序（存储在磁盘固定位置上的启动块），具有启动分区的磁盘称为启动磁盘。引导分区 (boot partition)包含操作系统和设备驱动程序，**win 将引导代码存在磁盘的第一个扇区，称为主引导记录(MBR)**，引导首先运行 ROM 内存中的代码，其指示系统从 MBR 中读取引导代码，MBR 包含一个分区表和指示引导的分区的标志，当系统找到引导分区时，读取分区的第一个扇区(boot sector)，并继续余下的引导过程。

On-disk file system: Boot control block contains info needed by system to boot OS from that volume,Volume（卷） control block contains volume details,Directory structure organizes the files,Per-file File Control Block

in-memory file system:

An in-memory partition table(分区表),An in-memory directory structure（目录结构）,The system-wide open-file table （系统打开文件表）,The per-process open-file table （进程打开文件表）

VFS:文件系统接口，包括 open，read，write，close 和文件描述符->VFS 接口,网络上唯一标识一个文件的机制 vnode.定义了 4 种主要的对象类型：索引节点对象（表示一个单独的文件），文件对象（表示一个打开的文件），超级块对象（表示整个文件系统），目录条目对象（表示一个单独的目录条目）

交换空间 = 虚拟内存使用磁盘空间作为主内存的扩展

可以通过政策文件系统（window）、单独的磁盘分区实现（Linux 的 SWAP）

RAID – multiple disk drives provides reliability via redundancy. Increases the mean time to failure.Frequently combined with NVRAM(Non-volatile RAM) to improve write performance. RAID0 无冗余，1-拷贝，2-内存中的汉明码 3、4-磁盘中的奇偶验码（3-位 4-块），5-P 分布在所有磁盘上，6-冗余纠错码防止多个磁盘出错 (RAID 1+0) 先镜像再条带，(RAID 0+1)先条带再镜像，0-性能（带状），1-可靠性（镜像）

storage:traditional-volume,pooled->storage pool

Chapter 13

设备驱动程序为 I/O 子系统提供了统一接口。

Direct I/O instructions：使用特殊 I/O 指令针对 I/O 端口地址传输一个字节或字。I/O 指令触发总线线路，选择适当设备，并将位移入或移出设备寄存器。

Memory-mapped I/O：设备控制寄存器被映射到处理器的地址空间，处理器执行 I/O 请求时通过标准数据传输指令读写映射到物理内存的设备控制器。

I/O 端口通常由 4 个寄存器组成：①data-in register②data-out register③status register④control register

轮询（Polling）：①主机重复读取忙位直到清零（忙等待）②主机设置命令寄存器的写位，并写出一个字节到数据输出寄存器③主机设置命令就绪位④当控制器注意到命令就绪位时设置忙位⑤控制器读取命令寄存器，并看到写命令，则从输出寄存区中读取一个字节并准备向设备执行 I/O 操作⑥控制器清楚命令就绪位，清楚状态寄存器的故障位表示 I/O 成功，清除忙位表示完成。

设备控制器通过 IRL 发送信号引起中断，CPU 捕获中断并分派到中断处理程序，中断处理程序通过处理设备来清除中断。

非屏蔽中断（nonmaskable interrupt）：保留用于诸如不可恢复的内存错误等事件。**可屏蔽中断：**由 CPU 在执行关键的不可中断的指令序列前加以屏蔽。

中断向量：包含专门的中断处理程序的内存地址。**中断优先级：**能够使 CPU 延迟处理低优先级中断而不屏蔽所有中断，这也可以让高优先级中断抢占低优先级中断处理。（0-19 不可屏蔽，32-127IRQ 外部中断，0x80）

DMA：主机将 DMA 命令块写到内存，该块包含传输源地址的指针、目标地址的指针、传输字节数，CPU 将这个命令块的地址写到 DMA 控制器，DMA 控制器继续操作内存总线，将地址放到总线，在没有主 CPU 的帮助下执行传输。当 DMA 传输完成后，控制器发送中断信号给处理器检查是否出错。

Block devices（块设备） include disk drives，**Character devices （字符设备）** include keyboards, mice, serial ports

内核与 I/O 有关的服务：I/O scheduling、buffering、caching、spooling（假脱机）、device reservation、and error handling.

SPOOL（Simultaneous Peripheral Operation On Line）：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。打印机虽然是独享设备，通过 SPOOL 技术，可以将它改造为一台可供多个用户共享的设备。

I/O 系统层次：用户程序→系统调用处理程序→设备驱动程序→中断处理程序

后门系统调用 ioctl 能使应用程序访问由设备驱动程序实现的一切功能

application->kernel->device-driver->device-controller->device

非阻塞 read 调用会马上返回，其所读取的数据可以等于或少于所要求的，或为零;

拷贝语义：操作系统保证要写入磁盘的数据就是 write（）系统调用发生时的版本。一个简单方法就是操作系统在 write 系统调用返回前将应用程序缓冲区复制到内核缓冲区中。

读取文件：确定保存文件的设备，转换名字到设备的表示法，把数据从磁盘读到缓冲区中，通知请求进程数据现在是有效的，把控制权返回给进程

STREAM-用户级进程与设备的全双工通信通道 user process->stream head->write queue->write queue->driver end->device->driver end->read quque->read queue->stream head->user process。每个模块包含一个读取队列和一个写入队列。

其他

File access is protected by user access rights and file attributes

open-read file control informantion block from outer storage to memory

生产者消费者-wait 操作不能逆转，signal 可以

```
shared data
#define BUFFER_SIZE 10
typedef struct {
        . . .
} item;
item
buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
item nextProduced; // local variable
while (1) {
        /* produce an item in nextProduced */
        while (((in + 1) % BUFFER_SIZE) == out);

        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
}
        /* consume the item in nextConsumed */
}
```

```
item nextConsumed; // local variable
while (1) {
        while (in == out);
// buffer is empty
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        /* consume the item in nextConsumed */
}
```

Busy waiting is where a process checks repeatedly for a condition- it is "waiting" for the condition, but it is "busy" checking for it. This will make the process eat CPU (usually).

打开文件系统调用

1. the directory structure is searched for the given file name
2. if found, the FCB is copied into a system-wide open-file table in memory
3. an entry is made in the per-process open-file table
4. returns a pointer to the entry in the per-process open-file table ： file descriptor in UNIX, file handle in Windows

关闭文件系统调用

1. the per-process table entry is removed
2. the system-wide table entry's open count is decremented：if the count hits zero, all updated info is copied back to the disk, and the system-wide table entry is removed

工作集：Working-Set Size WSSi (working set of Process Pi) = total number of pages referenced in the most recent D (varies in time)

D = S WSSi º total demand frames ； if D > m Þ thrashing

policy: if D > m, then suspend one of the processes (and swap it out completely)

实现：approximate with interval timer + a reference bit

Example: D = 10,000 references，timer interrupts about every 5000 references，keep in memory 2 bits for each page，whenever a timer interrupts, copy and sets the values of all reference bits to 0，if one of the bits in memory = 1 Þ page in working set

实验部分

Crtl 的快捷键

<Backspace>或<Ctrl-H>删除前一个字符
<Ctrl-U>删除当前行
<Ctrl-C>终止现在的命令，终止一个前台进程
<Ctrl-Z>挂起一个前台进程
<Ctrl+D>退出当前的 shell，eof，必须从登陆 shell 退出，必须关闭所有的 shell
<Ctrl-K>删除一行光标后字符
<Ctrl-P>上一次执行的命令，扫描过的不会再次出现

Shell 命令搜索路径

Shell 搜索的目录名字都保存在 shell 变量 PATH（在 TC shell 中是 path）中。

变量 PATH 中的目录名用符号分开。在 bash 中 ":"

变量 PATH 保存在~/.profile 或者~/.login 中（~:主目录）

Shell 元字符

"	引用多个字符,允许替换	"$file".bak
‘	引用多个字符	‘$100,000’
$	一行的结束/显示变量的值	$PATH
&	让一个命令在后台执行	command &
( )	在子 shell 中执行命令	(cmd1;cmd2)
{ }	在当前 shell 中执行命令	{cmd1;cmd2}
*	匹配 0 个或者多个字符	chap*.ps
?	匹配单个字符	lab.?
[ ]	插入通配符	[a-s],[1,5-9]
^	一行的开始/否定符号	[^3-8]
`	替换命令	PS1=`cmd`
|	创建命令间的管道	cmd1|cmd2
;	分割顺序执行的命令	cmd1;cmd2
<	重定向命令的输入	cmd<file
>	重定向命令的输出	cmd>file
\	转义字符/允许在下一行中继续 shell 命令	
!	启动历史记录列表中的命令和当前命令 !!,!4	
%	TC shell 的提示符，或者指定一个任务号时作为起始字符 %或者%3	

常用命令

whatis：得到任何 LINUX 命令的更短的描述

whoami：显示用户名 leafior

which：当某个工具或程序有多个副本时，用 which 来识别哪个副本在运行

who：显示现在正在使用系统的用户的信息

w：比 who 更加详细地列出系统上用户的信息

hostname：显示登录上的主机的名字 Ubuntu

uname：显示操作系统的信息 Linux

PATH=~/bin:$PATH:. 搜索路径中增加~/bin 和.目录

cal [[month] year]如 cal 4 2011

alias [name[=string]…] 为 name 命令建立别名 string。如 alias more=’pg’

unalias 删除别名 alias ll=’ ls －C’

uptime 显示系统运行时间命令

su [-][-c <command>] [username]

-c< cmd >执行完指定的指令后，即恢复原身份。

-改变身份时，也同时变更工作目录，及 HOME、SHELL、PATH 变量等

Username 指定要变更的用户名，默认为 root

文件

目录文件 d,字符设备文件 c,块设备文件 b 普通文件（文件名不超过 255）

字符设备文件和块设备文件：

fd0 (for floppy drive 0)
hda (for harddisk a)
lp0 (for line printer 0)
tty(for teletype terminal)

管道(FIFO)文件,链接文件,socket 文件

/根目录：包含了所有的目录和文件。

/bin：也称二进制目录，包含了那些供系统管理员和普通用户使用的重要的 Linux 命令的可执行文件。目录/usr/bin 下存放了大部分的用户命令。

/boot ：包括 Linux 内核的二进制映像。内核文件名是 vmlinux 加上版本和发布信息。

/dev：包含所有 linux 系统中使用的外部设备。但是这里并不是放的外部设备的驱动程序。

/etc：存放了系统管理时要用到的各种配置文件和子目录。网络配置文件，文件系统，x 系统配置文件，设备配置信息，设置用户信息等都在这个目录下。

/sbin：系统管理员的系统管理程序。

/home：/home/leafior

/lib：几乎所有的应用程序都会用到这个目录下的系统动态连接共享库。

/mnt ：这个目录主要用来临时装载文件系统 mount

/opt：该目录用来安附加软件包

/proc ：进程和系统得信息，可以在这个目录下获取系统信息。这些信息是在内存中，由系统自己产生的。

/root ：根（root）用户的主目录

/sbin，/usr/sbin，/usr/root/sbin：存放了系统管理的工具、应用软件和通用的 root 用户权限的命令

/tmp：用来存放不同程序执行时产生的临时文件

/usr：存放了可以在不同主机间共享的只读数据。

/lost+found：存放所有和其他目录没有关联的文件，这些文件可以用 Linux 工具 fsck 查找得到。

/var：存放易变数据。/var/spool/mail 存放收到的电子邮件，/var/log 存放系统的日志，/var/ftp

mount [-t fstype] [-o options] /dev/xxyN dirname

如：mount –t vfat /dev/hda1 /mnt/c

类型 设备文件 挂在目录

挂接 U 盘：mount –t vfat /dev/sda1 /mnt/usb

正则表达式

支持工具：awk, ed, egrep, grep, sed, vi

x|y|z	x 或 y 或 z
/L..e/	Love, Live, Lose, …
^x	以 x 开始的 string
x$	以 x 结束的 string
*	*
(xy)+或\(xy\)+	xy,xyxy,xyxyxy, …
xy?	x, xy
xy*	x, xy, xyy, xyyy
xy+	xy, xyy, xyyy, ..
[], [^]	/[Hh]ello/ /[^A-KM-Z]ove/ Love
{n}	匹配 n 次
{n,}	匹配 n 或 n+次
{n,m}	>=n, <=m

权限

对于目录：

r :列出目录的内容 w：建立，删除

x :允许用户搜索这个目录，如果你没有对目录的执行特权，那么就不能使用 ls －l 命令来列出目录下的内容或者是使用 cd 命令来把该目录变成当前目录。

Chmod u=rwx courses

文件访问权限 = 默认的访问权限－mask

默认访问权限: 执行文件为 777 文本文件为 666

Umask 013 对于一个新建的可执行文件 764

进程

Shell 执行二进制文件：

1.Shell 使用 fork 创建子进程；2.子进程执行 exec，用命令对应的可执行文件覆盖自身；3.命令执行，bash 等待命令结束。

Shell 执行脚本文件：

创建一个子 shell 并让子 shell 依次执行脚本中命令，执行与从键盘输入的命令采用相同的方式。子 shell 为每一个要执行的命令创建一个子进程。子 shell 执行脚本文件中的命令时，父 shell 等待子 shell 结束。子 shell 遇到脚本文件的 EOF 终止。子 shell 终止，父 shell 结束等待状态，开始重新执行。

命令组： 命令组中的所有命令都在一个进程中执行（在当前 shell 的子 shell 中）

$ (date;echo Hello,World!)

fg[%jobid] 后台→前台

bg 挂起进程→后台，参数同 fg（后台→前台）

jobs 显示所有挂起/停止的和后台进程的作业号

suspend 可以挂起当前 shell 进程

;顺序执行 &并发执行 |重定向

重定向

command < input-file > output-file

command > output-file < input-file

>换成>>则追加文件，否则替换

sdin— 0 stout — 1 sderr — 2

2> &1：使文件描述符 2 为文件描述符 1 的拷贝，导致错误信息送往和该命令输出相同的地方

$ cat lab1 lab2 lab3 2>&1 1>output

标准出错先设置成显示器，标准输出才改为 output

IPC

Linux 实现**进程间通信**(IPC Inter Process Communication)方法有：System V IPC 机制（ 信号量， 消息队列，共享内存）；管道（pipe）、命名管道；套接字（socket）；信号(signal)

概要

最为重要的决策之一是采用 GPL（GNU General Public License）。设

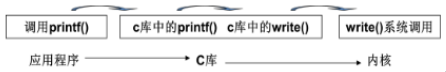
计 Linux 三原则（实用|有限目标|简单设计）
内核版本的序号：major(主版本号).minor(次版本号).patchlevel（对当前版本的修订次数）
Linux 系统结构(计算机硬件|linux 内核|shell|应用层|用户)
Linux 是一个单内核，Linux 内核运行在单独的内核地址空间.Linux 吸取了微内核的精华：其引以为豪的是模块化设计、抢占式内核、支持内核线程以及动态装载和卸载内核模块。
GNU C Library （glibc）提供了连接内核的系统调用接口
Linux 内核源程序安装在/usr/src/linux
Linux 系统引导过程使用内核镜像，/boot 目录下文件名称如：vmlinuz-2.6.15.5：普通内核镜像：zImage（Image compressed with gzip），大小不能超过 512k. 大内核镜像：bzImage（big Image compressed with gzip），包含了大部分系统核心组件：系统初始化、进程调度、内核管理模块

Linux 线程
Linux 2.6 内核支持 clone()系统调用创建线程
pthread_create()：创建线程函数；
pthread_exit()：主动退出线程；
pthread_join()：用于将当前线程挂起来等待线程的结束。这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数返回时，被等待线程的资源就被收回。
pthread_cancel()：终止另一个线程的执行。

重建内核
make clean　删除大多数的编译生成文件，　但是会保留内核的配置文件.config，　还有足够的编译支持来建立扩展模块
make mrproper　删除所有的编译生成文件，　还有内核配置文件，　再加上各种备份文件
make distclean　mrproper 删除的文件，　加上编辑备份文件和一些补丁文件。

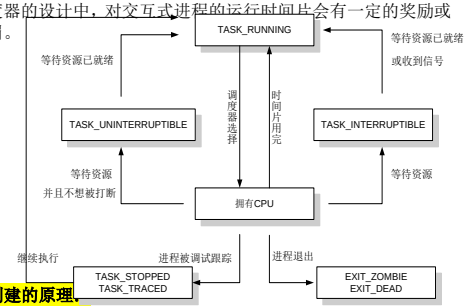
```
# cp /boot/config-`uname -r` .config
# make menuconfig
# make –j4
# make modules_install
# make install
Sudo mkinitramfs -o /boot/initrd.img-2.6.36
Sudo update-initramfs -c -k 2.6.36
Sudo update-grub2 //自动修改系统引导配置，产生grub.cfg启动文件
```

系统调用



系统调用是用户进程进入内核的接口层，它本身并非内核函数，但它是由内核函数（服务例程）实现的。
运行模式（mode）
Linux 使用了其中的两个：特权级 0 和特权级 3 ，即内核模式(kernel mode)和用户模式(user mode）
上下文（context）。三个部分：
用户级上下文：正文、数据、用户栈以及共享存储区；
寄存器上下文：通用寄存器、程序寄存器（IP）、处理机状态寄存器（EFLAGS）、栈指针（ESP）；
系统级上下文：进程控制块 task_struct、内存管理信息(mm_struct、vm_area_struct、 pgd、pmd、pte 等)、核心栈等。
程序执行系统调用步骤：
1、程序调用libc库的封装函数。
2、调用软中断 int 0x80 进入内核。
3、在内核中首先执行system_call函数，接着根据系统调用号在系统调用表中查找到对应的系统调用服务例程 。
4、执行该服务例程。
5、执行完毕后，转入ret_from_sys_call例程，从系统调用返回

进程状态
TASK_RUNNING：正在运行的进程即系统的当前进程或准备运行的进程即在 Running 队列中的进程。只有处于该状态的进程才实际参与进程调度。
TASK_INTERRUPTIBLE：处于等待资源状态中的进程，当等待的资源有效时被唤醒，也可以被其他进程或内核用信号、中断唤醒后进入就绪状态。
TASK_UNINTERRUPTIBLE：处于等待资源状态中的进程，当等待的资源有效时被唤醒，不可以被其它进程或内核通过信号、中断唤醒。
TASK_STOPPED：进程被暂停，一般当进程收到下列信号之一时进入这个状态：SIGSTOP，SIGTSTP，SIGTTIN 或者 SIGTTOU。通过其它进程的信号才能唤醒。
TASK_TRACED：进程被跟踪，一般在调试的时候用到。
EXIT_ZOMBIE：正在终止的进程，等待父进程调用 wait4()或者 waitpid()回收信息。是进程结束运行前的一个过度状态（僵死状态）。虽然此时已经释放了内存、文件等资源，但是在内核中仍然保留一些这个进程的数据结构（比如 task_struct）等待父进程回收。
EXIT_DEAD：进程消亡前的最后一个状态，父进程已经调用了 wait4()或者 waitpid()。
TASK_NONINTERACTIVE：表明这个进程不是一个交互式进程，在调度器的设计中，对交互式进程的运行时间片会有一定的奖励或者惩罚。



进程创建的原理：
系统创建的第一个进程是 init 进程。系统中所有的进程都是由当前进程使用系统调用 fork()创建的。子进程被创建后继承了父进程的资源。子进程共享父进程的虚存空间。
用 exec 系列函数执行真正的任务。
Fork() 函数进程创建的过程：
1) 为新进程分配 task_struct 内存空间;
2) 把父进程 task_struct 拷贝到子进程的 task_struct;
3) 为新进程在其虚拟内存建立内核堆栈;
4) 对子进程 task_struct 中部分进行初始化设置
5) 把父进程的有关信息拷贝给子进程，建立共享关系;
6) 把子进程的 counter 设为父进程 counter 值的一半;
7) 把子进程加入到可运行队列中;
8) 结束 do_fork()函数返回 PID 值.
Linux 把**线程**和进程一视同仁，每个线程拥有唯一属于自己的 task_struct 结构。不过线程本身拥有的资源少，共享进程的资源，如共享地址空间、文件系统资源、文件描述符和信号处理程序。内核线程是通过系统调用 clone()来实现的

Exec()
在 Linux 系统中，使程序执行的唯一方法是使用系统调用 exec()。exec 函数族把当前进程映像替换成新的程序文件，而且该程序通常 main 函数开始执行。
其中只有 execve 是真正意义上的系统调用，其它都是在此基础上经过包装的库函数。exec 函数族的作用是根据指定的文件名找到可执行文件，并用它来取代调用进程的内容，就是在调用进程内部执行一个可执行文件。

fork（）函数
fork:创建一个新子进程
#include <sys/types.h>#include <unistd.h>pid_t fork(void);
返回值：调用一次，返回两次。子进程的返回值是 0，父进程的返回值则是子进程的进程 ID,出错为-1
if ((pid=fork()) < 0) {　　/* error handling */}
else if (pid == 0) {　　/* child */}
else {　　/* parent */};
用 fork 函数创建子进程后，子进程往往要调用一种 exec 函数以执行另一个程序。当进程调用一种 exec 函数时，该进程完全由新程序代换，而新程序则从其 main 函数开始执行。调用 exec 并不创建新进程，所以前后的进程 PID 并未改变。exec 只是用另一个新程序替换了当前进程的正文、数据、堆和栈段。
子进程调用 getppid 以获得其父进程的进程 ID
子进程和父进程共享很多资源，除了打开文件之外，很多父进程的其他性质也由子进程继承：如实际用户 ID、实际组 ID、有效用户 ID、有效组 ID 等。父、子进程之间的区别包括 fork 的返回值，进程 ID，不同的父进程 ID，父进程设置的锁，子进程不继承等。

进程调度
Linux 系统采用抢占调度方式。无论内核态还是用户态。
分时技术，对于优先级相同进程进程采用时间片轮转法。
根据进程的优先级对它们进行分类。进程的优先级是动态的。
Linux 2.6 的进程设置 140 个**优先级**。实时进程优先级为 0-99，普通进程优先级 100-139 的数。0 为最高优先权，139 为最低优先权。优先级数值越大，优先级越低，分配的时间片越少
实时进程的 static_prio 不参与优先级 prio 的计算
unsigned long policy: 进程调度策略
调度对象是可运行队列，每个处理器有一个可运行队列
普通进程的**权值**就是它的 counter 的值（处置 21），而实时进程的权值是它的 rt_priority 的值加 1000
PID 是 32 位的无符号整数，它被顺序编号，最大值为 32768。
内核堆栈：thread_info 代替了原先 task_struct 的位置，跟内核堆栈放在一块，thread_info 中放置一个指向 task_struct 的指针

存储管理
一个进程的用户地址空间主要由 mm_struct 结构和 vm_area_structs 结构来描述. mm_struct 结构它对进程整个用户空间进行描述, vm_area_structs 结构对用户空间中各个区间(简称虚存区)进行描述. mm_struct 结构首地址在 task_struct 成员项 mm 中：struct mm_struct *mm：vm_area_struct 结构是虚存空间中一个连续的区域，在这个区域中的信息具有相同的操作和访问特性
fork()是通过拷贝或共享父进程的用户空间来实现的，即内核调用 copy_mm()函数，为新进程建立所有页表和 mm_struct 结构

do_page_fault()
页面异常的处理程序两个参数：
一个是指针,指向异常发生时寄存器值存放的地址。
另一个错误码,由三位二进制信息组成：
　第 0 位——访问的物理页帧是否存在；
　第 1 位——写错误还是读错误或执行错误；
　第 2 位——程序运行在核心态还是用户态。
do_page_fault( )函数定义在 arch/i386/mm/fault.c 文件中
Linux 内核利用守护进程 **kswapd** 定期地检查系统内的空闲页面数是否小于预定义的极限
Buddy 算法是把内存中的所有页帧按照 2^n 划分，其中 n=0～10。划分后形成了大小不等的存储块，称为页帧块，简称页块。数组 free_area[]来管理各个空闲页块组，申请空间的函数为 alloc_pages( )；释放函数为 free_pages( )；
slab 分配器：为经常使用的小对象建立缓冲,小对象的申请与释放都通过 slab 分配器来管理。slab 分配器再与伙伴系统打交道。基于伙伴系统的 slab 分配器

VFSVFS 仅存在于内存
超级块对象 superblock :存储已安装文件系统的信息，通常对应磁盘文件系统的文件系统超级块或控制块。
索引节点对象 inode object ：存储某个文件的信息。通常对应磁盘文件系统的文件控制块
目录项对象 dentry object ：dentry 对象主要是描述一个目录项，是路径的组成部分。
文件对象 file object：存储一个打开文件和一个进程的关联信息。只要文件一直打开，这个对象就一直存在与内存
File 对象
文件对象 file 表示进程已打开的文件，只有当文件被打开时才在内存中建立 file 对象的内容。
该对象由相应的 open()系统调用创建，由 close()系统调用销毁。
添加系统调用：
system_call()函数实现了系统调用中断处理程序：
1.它首先把系统调用号和该异常处理程序用到的所有 CPU 寄存 器保存到相应的栈中， SAVE_ALL
2.把当前进程 task_struct（thread_info）结构的地址存放在 ebx 中
3.对用户态进程传递来的系统调用号进行有效性检查。若调 用号大于或等于 NR_syscalls，系统调用处理程序终止。（sys_call_table）
4.若系统调用号无效，函数就把-ENOSYS 值存放在栈中 eax 寄 存器所在的单元，再跳到 ret_from_sys_call()
5.根据 eax 中所包含的系统调用号调用对应的特定服务例程
实验修改的主要有 3 处地方：

```
for (p = &init_task; (p = next_task(p)) != &init_task;)
    //遍历进程
```

在大部分公司发行的 Linux 系统中，默认的 shell 是：bash
哪个命令可以用来查看 Linux kernel 版本信息 ： uname
在创建 Linux 分区时，至少要创建的两个分区是：Swap/root
哪一个只是在内核运行时存在的 ： proc
Linux 的内核受严格保护，与进程的用户态代码几乎隔绝。若想从用户态进入内核态，可以通过 ： int 0x80

题目：
1.各种分配方案的文件系统可管理的最大文件
连续分配：不受限制，可大到整个磁盘文件区。
链接分配：同上。
二级索引：由于盘块大小为 4KB，每个地址用 4B 表示，一个盘块可存 1K（4096/4=1024）个索引表目，二级索引可管理的最大文件容量为 4KB×1K×1K＝4GB，
2.每种分配方案对 20MB 大文件和 20KB 小文件各需要多少专用块来记录文件的物理地址?
连续分配：文件控制块 FCB 中设二项，一是首块物理块块号，另一是文件总块数
链接分配：在每块存文件的物理块中设置存贮下一块块号的指针。
二级索引：对大小文件都固定要用二级索引，用一块作第一级索引，用另一块作二级索引
3.为读大文件前面第 5.5KB 和后面（16M＋5.5KB）信息需要多少次盘 I/O 操作?
连续分配：相对逻辑块号为 5.5K / 4K=1 计算物理块号＝文件首块号＋相对逻辑块号。1 次
链接分配：该信息所在块前面块顺序读出，共化费 4097 次盘 I / O 操作得到块号最后化一次 I/O 操作读出该块信息。
二级索引：进行一次盘 I / O 读第一级索引块，然后根据它的相对逻辑块号计算应该读第二级索引的那块，第一级索引块表目号=相对逻辑块号 / 1K，对文件前面信息 1 / 1K＝0，对文件后面信息 4097 / 1K＝4，第二次根据第一级索引块的相应表目内容又化一次盘 I / O 读第二级索引块，得到信息所在块块号，再化一次盘 I / O 读出信息所在盘块，这样读取大文件前面或后面信息都只需要 3 次盘 I / O 操作。