概述

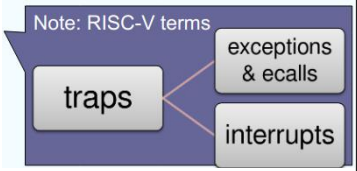
Policy: What will be done? 策略（确定具体做什么事）
Mechanism: How to do it? 机制（定义做事方式）

OS 是用户和硬件之间的中间人。目标：**用户**：容易学容易用，可靠性高，速度快。**系统**：易于设计实现维护，灵活可靠错误少高效。
系统四大部分：硬件、OS、应用程序和用户。
OS 是用户与计算机硬件之间的接口。**命令级**，提供键盘或鼠标命令。程序级：提供**系统调用**。
OS 是计算机系统资源的管理者，资源分配者是一个控制程序，扩充裸机的第一层系统软件。

大型机系统 Mainframe System
批处理系统(单道、多道)、分时系统。(多个作业保存在内存中，CPU 在它们之间进行切换)。
发展： no software->resident monitors->multi programming 多道->multi-tasking(multitasking)。
最高 throughput！

Time Sharing system 分时系统
分时系统是多个用户分时共享，把系统资源分割，每个用户轮流使用一个时间片。
并发:两个或多个事件在同一时间间隔内发生
并行:两个或多个事件在同一时刻发生。
逻辑上，并行是并发的子集

集群系统 Clustered: SMP,DSP
实时系统 Real Time 有软实时 硬实时。用于工业控制、显示设备、医学影像、科学实验
手持 Handheld：PDA cellular telephone 嵌入式.
市场格局 Unix 服务器 Win 桌面 Android 手机
特权指令：用户程序不能直接使用，如 IO 时钟设置 寄存器设置，系统调用不是特权指令
双模：用户态：执行应用程序时。内核态：执行操作系统程序时。
仅内核态：Set value of the system timer, Clear memory, Turn off interrupts, Modify entries in device-status table, Access I/O device，CSR
一般来说，两态转换、保存 PC,PSW 等由硬件实现，额外保存通用寄存器等操作系统实现
Interrupt：Caused by externel events.
Exceptions: Caused by executing instructions.



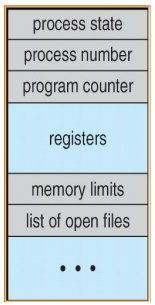
操作系统结构

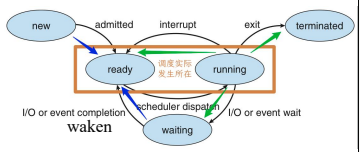
操作系统服务：
用户界面：CLI GUI
程序执行 IO 操作 文件系统操作 通信 错误检测 资源分配 统计 保护和安全
操作系统的用户界面
操作系统接口：命令接口和程序接口(系统调用)；命令接口：CLI GUI；程序接口：系统调用指 OS 提供的服务。
系统调用是进程和 OS 内核间的程序接口。
一般用 C/C++写，大多数由程序提供的叫做 API 应用程序接口(API 不是系统必定提供的)，而不是直接的系统调用。三个常见的 API 是 win32 API，POSIX API 和 JAVA API。Time()是一个系统调用
系统调用有三种传参方式：寄存器传参 参数存在内存的块和表中，将地址通过寄存器传递，linux 和 solaris 用这种参数通过**堆栈**传递。
strace -xf [command]: 查看某命令执行过程中发生的所有系统调用情况
Types of syscalls: Process control, File management, Device management, Information maintenance, Communications, Protections

ELF: .text: code, .rodata: initialized read-only data, .data: initialized data, bss: uninitialized data (未初始化的静态/全局变量在 bss 段)
动态链接： loader 链接 ELF 与 lib，.interp 段
Running binary： sys_execve -> do_execve ->...->load_elf_binary->start_thread ， 从 entry point address 作为用户程序起始地址。
静态连接：_start 地址在 execve 调用后立刻执行；动态链接：先执行 ld.so
操作系统结构：
简单结构
MSDOS，小、简单功能有限的系统，没有划分为模块，接口和功能层次没有很好的划分
原始 UNIX：受到硬件功能限制，原始 UNIX 结构受限，分为两部分：系统程序和内核。
UNIX LINUX 单内核结构(Monolithic)
层次结构：OS 被划分为很多层，最底层 0 层是硬件，最高层是用户接口。通过模块化，选择层，使得每个层使用较低层的功能和服务。
微内核结构 microkernel system
只有最基本的功能直接由微内核实现，其他功能都委托给独立进程。也就是由两大部分组成：微内核和若干服务。**好处**：利于拓展、容易移植到另一种硬件平台设计。更加可靠（内核态运行的代码更少了），更安全。**缺点**：用户空间和内核空间的通信开销很大。Windows NT windows 8 10 mac OS L4
单/宏内核 monolithic kernel
与微内核相反，内核的全部代码，包括子系统都打包到一个文件中。更加简单更加普遍的 OS 体系。**优点**：组件之间直接通讯，开销小；**缺点**：很难避免源代码错误 很难修改和维持；内核越来越大。如 OS/360，VMS Linux
模块 modules
大多数现代操作系统都实现了内核模块。面向对象，内核部件分离，通过已知接口进行沟通，都是可以载入到内核中的。总而言之很像层次结构但是更加灵活。Linux solaris。
混合系统 Hybrid
大多数现代操作系统不是单一模型。Linux 和 solaris 是 monolithic+module。Windows 大部分是 monolithic 加上 microkernel。Mac 是层次 Hybrid;还有一些如 eBPF 支持沙箱

Process 进程

a running program，与 program 最大区别在静态/动态（状态）
进程(用户部分)：PC, 寄存器,数据段(全局 data),栈(临时 data),堆(alloc dynamic)。

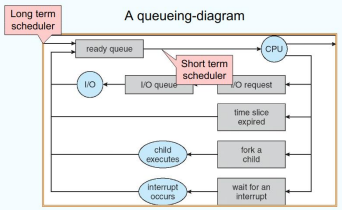




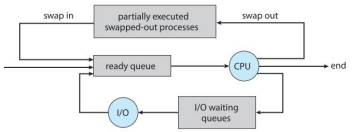
进程状态会因为程序(系统调用),OS(调度),外部(中断)动作而改变状态。单处理器下，最多一个 run. ready 进程构成就绪队列；waiting 进程构成**多种**等待队列(存 PCB)。运行最多 k（核数）最少 0，等待最多 n 最少 0，就绪最多 n-1 最少 0。
进程（内核部分）：PCB：Process state, PC, registers, CPU scheduling information, Memory management information, Accounting information, File management, I/O status information（相当于元数据）
Linux 的 PCB 保存在 struct task_struct 里
Fork: fork() -> exec*() [replace memory space] -> parent calls wait() for children to terminate**(child fork() = 0)**
wait(): block until any child terminates. waitpid(): block until a specific child completes
Signal: signal() system call allows a process to specify what action to do on a signal
Zombies&Orphans: Zombie die until its parent has called wait() or its parent dies. Orphan is "adopted" by the process with pid 1.

进程调度

调度队列有：作业 job 队列 就绪队列 设备队列 进程在这些队列内移动。下图是队列图，新进程开始就处于就绪队列



Long-term/Job：选择应该被带入就绪队列的进程，调度频率低，控制多道程序设计的程度(内存中进程数)，unix win 不使用
Short-term/CPU：从接下来要执行的进程中选择并分配 CPU，频率很高。
Midterm：能将进程从内存或 CPU 竞争中溢出，降低多道程序设计的程度。进程可以被换出，之后再换入。



进程可以分为 **IO 型**和 **CPU 型**(CPU-bound)
上下文切换：CPU 切换进程时，必须保存当前状态再载入新状态。进程的上下文储存在 PCB 中。overhead 过程，不能进行其他事务。

进程操作

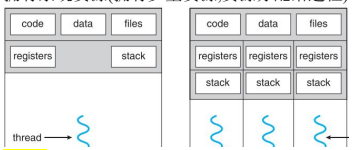
父子进程资源共享模式：共享全部资源/部分/不共享；**执行模式**：并发执行/父进程等待子进程结束再执行；**地址空间**：子拷贝父/子进程装入另一个新程序。
进程终止：父进程终止时，不同 OS 对子进程不同：不允许子进程继续运行/级联终止/继承到其他父进程上(init, PID=1)。
合作进程：(**独立进程**：运行期间不会受到其他进程影响)进程合作优点：信息共享,运算速度提高,模块化,方便。生产消费问题的两种缓冲：无限缓冲 unbounded-buffer 生产者可以无限生产;有限缓冲，缓冲满后生产者要等待

进程通信 IPC

Shared memory: shm_open(): creates a shn segment; mmap(): memory-map a file pointer to the shared memory object; Reading and writing to shared memory is done by using the pointer returned by mmap().
Message passing: basic: send(Q, msg), recv(Q, msg),每个 link 对应一组 pair. 实现通信 link：通过 mailbox(ports)接受信息（新的操作：create mailbox, destroy mailbox）
Synchronization: blocking - synchronous , non-blocking - asynchronous
Signals, Pipes: Ordinary pipes – unidirectional, parent-child relationship (fd[0] is the read end; fd[1] is the write end); Named pipes: bidirectional
RPC remote procedure calls

Thread 线程

资源拥有单位：process
调度单位：thread
OS 将它们分别处理，调度单元被称为线程或轻量进程 LWP(lightweight process),资源拥有单元被称为进程任务。线程就是进程内一个执行单元或可调度的实体。重型/传统线程=单线程的进程
线程能力：有状态转移,不运行时保存上下文,有一个执行栈,有局部变量的静态存储,可以存取所在线程的资源,可以创建撤销其他线程,不拥有系统资源(拥有少量资源,资源分配给进程)



动机:线程共享 code section，data section(heap)，signals(没有 Mem, File 等资源边界限制)。**优点**：创建新线程耗时少,context switch 开销小(no cache flush),线程间通讯可以通过 shared memory;responsiveness, scalability **缺点**：weak isolation; one thread fails, process fails
用户级线程：不依赖于 OS 核心(内核不了解用户线程的存在)，利用线程库提供创建、同步调度和管理线程的函数来控制用户线程。一个线程发起系统调用而阻塞，则整个进程在等待。
内核级线程：依赖于 OS 核心，由内核的内部需求进行创建和撤销 ，用来执行一个指定的函数。一个线程发起系统调用而阻塞，不会影响其他线程。时间片分配给线程，所以多线程的进程获得更 多 CPU 时间。

多线程模型：

多对一：将多个用户级线程映射到一个内核线程，由线程库在用户空间进行调度。green thread 优点：无需 OS 支持，可以调整(tune)调度策略满足需求，线程操作开销很低。缺点：无法利用多处理器，不是真并行，一个线程阻塞时整个进程也阻塞
一对一：一个用户到一个内核，每个内核线程 OS 完成独立调度， win NT/XP/2000 linux Solaris 9 later。优点：每个内核线程可以并行跑在多处理器上，一个线程阻塞，进程的其他线程可以被调度。缺点：线程操作开销大,OS 对线程数的增多处理必须很好
多对多：多用户到多内核，允许 OS 创建足够多的内核线程，Solaris prior v9 win NT/2000 with ThreadFiber。
两极模型：多对多的变种，一部分多对多，但是有一个线程是绑定到一个内核上。IRIX HP-UX Tru64 Solaris 8 earlier
线程调用 fork：两种情况：仅复制线程、复制整个进程的所有线程（Linux 为第一种）

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

两种线程取消：Asynchronous：立即终止目标线程;**Deferred**：目标线程不断检查自己是否该终止。**信号处理**：信号由特定事件产生，发送给进程，被发送后需要被处理。发送信号到所应用的线程/每个线程/某些线程/规定特定线程接收信号。**线程池**：用现有线程处理请求比等待创建新线程快；限制了可用线程的数量。**线程特有数据**：允许线程自己保存数据拷贝，在无法控制创建线程时很有用，如线程池。

调度程序激活 Scheduler Activations
多对多和两级模型需要通信来维护分配给应用适当数量的线程；SA 提供 upcalls，一种从内核到线程库的通信机制；这种通信保证了应用可以维持正确数量的线程。Win xp linux
WIN XP 实现一对一模型，但是通过 fiber 库也支持多对多。每个线程包括：ID 寄存器集 用户栈和内核栈 私有数据存储区。后面三个都是线程的上下文。主要数据结构 ETHREAD 执行线程块 KTHREAD 内核线程块 TEB 线程执行环境块 后者在用户空间 前两者内核空间。
Linux 把线程叫做 tasks（不区分线程和进程），除了 fork，额外提供了 **clone** 系统调用完成线程创建，它允许子任务和父任务共享地址空间

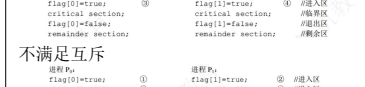
CPU Scheduling（软件算法）

non-preemptive(running 主动出)，preemptive
分派程序(dispatcher)：上下文切换 切换到用户模式 跳转到对应位置来重启程序。花费的时间叫分派延迟(dispatcher latency)。
调度算法的选择准则和评价：
面向用户：**turnaround** (进 ready queue 到完成所用时间 带权周转时间=周转时间/CPU 执行时间) **response**(进 ready queue 到第一次输出) **waiting**(在 ready queue 中等待时间) 。面向系统：**throughput**（单位时间完成的任务数）**CPU utilization（利用率）**。调度算法自身：易于实现 开销较小。
调度算法
First-Come, First-Served (FCFS) ：根据 ready queue 先后分配，非抢占，最简单，利于长进程，不利于短进程，利于 CPU 型不利于 IO 型。
Shortest-Job-First (SJF) ：对预计执行时间短的作业优先分派 CPU。**Shortest Remaining Time First (SRTF)**实时抢占。SJF 能给出最小平均等待时间，但是实际不知道下一个 CPU 脉冲 burst 时长-> 预测：Exponential averaging of previously observed burst durations：
新预测=(1-alpha)第 n 次时长+alpha 原预测
Priority：总是把 CPU 分配给就绪中最高优先级（静/动态）的进程。**一般数字越小优先级越高**。SJF 是以下一次 CPU 的脉冲长度作为优先数的优先级调度特例。优先级调度也可以是抢占/非抢占的。问题：**starvation**，低优先级的进程可能永远无法执行->**aging** 逐渐增加在系统中等待时间长的进程的优先级
Round Robin(RR)：通过时间片轮转提高并发性和响应时间，提高资源利用率，抢占式。
RR 算法性能依赖于时间片大小，如果 q 很大，就和 FCFS 一样了**(时间片未到程序也可以结束)**；同时 q 也要足够大来保证上下文切换，否则开销过大。时间片长度的影响因素：响应时间一定时，就绪进程越多，时间片越小；应当使用户输入通常能在一个时间片内完成，否则相应 平均周转和平均带权周转都会延长。
一般来说 RR 比 SJF 有更高的平均周转，无 starvation, response time 更好。时间片固定时，用户越多响应时间越长。
当时间片太长时（IO），会自动调度
Multi-Level Queue: 将就绪队列根据性质或类型的不同分为多个独立队列区别对待，综合调度。不同队列可能有不同的调度算法。例如系统进程、用户交互、批处理等这样的队列分法(Real time, System, Interactive,Batch processes)。一般，分成前台 foreground(交互式 interactive)和后台(批处理)，前台一般 RR，后台 FCFS。
多级队列在队列间的调度算法有：固定优先级，即先前台后后台，有饥饿；给定时间片，如 80%执行前台的 RR，20%执行后台的 FCFS。
Multilevel Feedback Queue：是 RR 和优先级算法的综合，**允许进程在不同的就绪队列切换**，等待时间长的进程会进入到高优先级队列中。
优点：提高吞吐量降低平均周转而照顾断进程；为 IO 设备利用率和降低响应时间而照顾 IO 进程型；**IO bound 给予高优先级，CPU bound 低优先级**
***Multi-Processor, Real-time, Thread**

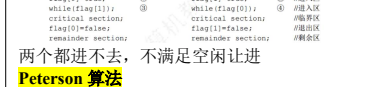
Process synchronization

Race condition:A race condition is a situation in which a memory location is accessed concurrently, and at least one access is a write.
The Critical-Section Problem
每个进程都有一个临界区，其中共享数据被访问，需要保证只有**一个**进程进入临界区。临界区问题的解决必须满足三个要求：**互斥(mutual exclusion)；空闲让进(progress)；有限等待(bounded wait)**。让权等待不是必须的。

不满足空闲让进，对面不进自己也不能进



不满足互斥



两个都进不去，不满足空闲让进
Peterson 算法
只用于两个进程的情况，并且假设 L/S 是原子操作，是一种软件解决方法
管程
专门使用一个进程来处理 CS 资源，所有相关访问由其调度

The two processes share two variables:

- int turn;
- boolean flag[2]

The variable turn indicates whose turn it is to enter the critical section.

The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!

Process P_i

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

Meets all three requirements; solves the critical-section problem for two processes.

现代 OS 中不适用(若编译器交换 flag 和 turn 两条指令（并行）则寄)

硬件同步方法

单处理器：在临界区禁止中断。

多处理器：**Memory barriers**(an instruction forcing any change in memory to be propagated (made visible) to all other processors，强制 SC)

Hardware instruction

抽象出两个硬件实现的原子操作：赋值和交换

```
boolean test_and_set(boolean *target) {
    boolean rv = *target;
    *target = true;

    return rv;
}
```

```
while(1) {
    key = TRUE;
    while (key == TRUE)
        Swap(&lock,&key);
    critical section
    lock = false;
    remainder section
}
```

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

Test_and_set 存在 busy waiting 的情况。

test-and-set 的共享变量是 lock 初始 false，swap 也是一样，但是多了局部变量 key 不是共享的。

硬件方法优点：进程数随意，简单，支持多个临界区；

缺点：可能饥饿（图片解决），可能死锁，会引起忙等待（无法让权等待）

```
/* critical section */
j = (i + 1) % n;
while ((j != i) && !waiting[j])
   j = (j + 1) % n;
if (j == i)
   lock = 0;
else
   waiting[j] = false;
/* remainder section */
```

mutex 互斥锁

acquire(), release()，等价于 binary semaphore

semaphores

分为 counting 和 binary(mutex lock)，两个操作 wait(P) 和 signal(V)。为了避免忙等待：

```
wait (S){
        value--;
        if (value < 0) {
                // add this process to waiting queue
                block(); }
    }
Signal (S){
        value++;
        if (value <= 0) {
                // remove a process P from the waiting queue
                wakeup(P); }
    }
```

具有忙等的信号量值非负，但是这种实现可以为负：**正数表示空闲资源数，负数的绝对值代表等待该信号量的进程数**。连续的 wait 顺序是需要注意的，但是连续的 signal 无所谓。同步 wait 和互斥 wait 相邻时，要先同步 wait。优点：简单、表达能力强；缺点：不够安全，会死锁，实现复杂，会 starvation

实现同步：见右图 A->B

优先级倒置(priority inversion)：当优先级较低的进程持有较高优先级进程所需的锁定时的调度问题。解决方法：**priority inheritance:** temporary assign the highest priority of waiting process to the process holding the lock.

```
semaphore S = 0;

P0() {
    /* Section A */
    signal(S);
}

P1() {
    wait(S);
    /* Section B */
}
```

Bounded-Buffer Problem 有限缓冲区 生产者-消费者问题

```
void produce() {
    /* something */
    ++n;
    /* something */
}

void consume() {
    /* something */
    --n;
    /* something */
}
```

```
producer(){
    while (true)  {
            //   produce an item
        wait (empty);
        wait (mutex);
            //  add the item to the  buffer
        signal (mutex);
        signal (full);
    }
}

consumer(){
    while (true)  {
        wait (full);
        wait (mutex);
            //  remove an item from buffer
        signal (mutex);
        signal (empty);
            //  consume the item
    }
}
```

设置 empty+full==n, mutex 保护 buffer

Readers-Writers Problem

读写互斥，其中所有的读共享一把锁（writer 直接 wait rw_mutex 即可）

```
reader() {
    wait(mutex);                //  ┐
    read_count++;               //  ├ obtain `mutex` to increase
    if (read_count == 1) {      //  │ `read_count`
        wait(rw_mutex);         //  ┘
    }
    signal(mutex);              //  ┐
                                //  ├─ readers share `rw_mutex` to
    /* critical section */      //  │  read critical resource
                                //  ┘
    wait(mutex);                //  ┐
    read_count--;               //  ├ obtain `mutex` to decrease
    if (read_count == 0) {      //  │ `read_count`
        signal(rw_mutex);       //  ┘
    }
    signal(mutex);              //  ┘
}
```

Dining-Philosophers Problem 哲学家进餐

N 个哲学家坐在圆桌，每个哲学家和邻居共享一根筷子；哲学家吃饭要用身边的两只筷子一起吃：

```
While (true) {
    wait ( chopstick[i] );
    wait ( chopStick[ (i + 1) % 5] );

        // eat

    signal ( chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );

        // think

}
```

邻居不允许同时吃饭；哲学家只会思考或者吃饭。共享数据：数据集/一碗米饭；共享变量 chopstick[5]初始为 1

这个解决方案可以保证没有 2 个哲学家同时使用 1 个筷子，但是很显然会导致死锁，如果 5 个哲学家同时拿起左手筷子，就死锁了。

一些其他的可能解决：最多只允许 4 个哲学家同时拿/临界区内必须拿起两根筷子/使用非对称的解决方法：奇数先拿左手，偶数先拿右手。

Deadlock 死锁

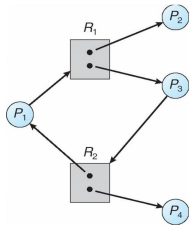
死锁指多个进程因竞争共享资源而造成的一种僵局，若无外力作用，这些进程都将永远不能再向前推进。进程按以下顺序使用资源：申请 使用 释放，申请和释放为系统调用。

四个必要条件：

Mutual exclusion：一次只能有一个进程使用资源;**hold and wait**：一个进程必须至少占有一个被等待资源并且等待另一个被占用资源; **no preemption**：资源不能被抢占，只能在进程使用完成后释放;**circular wait**：进程间循环等待资源

资源分配图(resource-allocation graph)：

点集：进程（圆）资源类型（方，点表示实例）。进程 Pi 到资源 Rj 的请求边(request edge)表示进程 Pi 申请资源 Rj 的一个实例；资源 Rj 到进程 Pi 的分配边(assignment edge)表示 Rj 的一个实例已经分配给了进程 Pi。



如果分配图无环->没有进程死锁，如果有环，那么可能死锁。如果(环内)每个资源恰好只有一个实例，有环则必死锁。（多个实例有环不一定死锁）

死锁处理

保证系统不进入死锁：prevention avoidence；允许进入死锁但是需要恢复：detection recovery。U L W 三个系统都忽略问题假装没有死锁，是鸵鸟方法。

死锁预防 Prevention

通过限制请求的方式来预防死锁，本质上就是破坏上面的四个必要条件。

Mutual exclusion：非共享资源必须互斥，共享资源不需要互斥。天生难以破坏。

Hold and wait：保证一个进程**申请一个资源时不能占有其他资源->**一旦申请资源就一次性获取所有资源，如果没法一次性获取所有资源就释放已经申请到的资源，是资源静态预分配的方法；缺点：低资源利用率、可能饥饿。

No preemption：允许进程/线程强行抢占另外一个进程/线程持有的资源，进程需要获取到原有的资源和申请的新资源后才能运行。缺点：难以实现，同样可能饥饿

Circular wait：给资源设置显式序号，请求必须按照资源序号递增的方式进行，破坏循环等待条件。缺点：效率低，可能被小号“阻塞”

死锁避免 Avoidance

前面的方法虽然避免了死锁，但是降低了吞吐率，我们可以通过获取一些额外的事先信息(prior information)从而**动态检查**避免死锁。

安全状态：

安全状态->没有死锁；不安全状态->可能有死锁；避免->保证系统永远不进入非安全状态。

资源分配图，single instance 死锁避免算法：

引入一种虚线需求边（claim edge）Pi->Rj 表示进程 Pi 在未来可能请求资源 Rj。当进程真正请求资源时，用请求边覆盖掉需求边。当资源被分配给进程后，用分配边来覆盖掉请求边，当资源被释放后，分配边恢复为需求边。系统必须事先得到所有需求边。

算法：假设进程 Pi 申请资源 Rj。只有在需求边 Pi ->Rj 变成分配边 Rj->Pi 而**不会导致资源分配图形成环**时，才允许申请，变为请求边。用该算法循环检测，如果没有环存在，那么资源分配会使系统继续安全状，否则就会不安全，Pi 就要等待。

Banker，多实体资源类型避免算法(没人用)：

要求：每个进程实现说明最大需求；进程请求资源时可能会等待；进程拿到资源后必须在有限时间内释放它们。

数据结构：N 进程数，M 资源类型的种类数；Available[M]表示可用资源数，Max[N][M]表示每个进程最大需求，Allocation[N][M]表示每个进程已分配，Need[N][M] = Max - Allocation 表示未来分配

安全状态检测算法：O(M * N * N)

贪心算法确定一条顺次分配路径

资源请求算法：

假设请求 request 已经发生，如果产生的资源分配状态是安全的，那么进程 Pi 可以分配到资源；否则进程 Pi 必须等待 Request[i]并且恢复到原有的资源分配状态。

死锁检测

允许系统进入死锁状态的话，那么系统就需要提供检测算法和恢复算法。

等待图，wait-for graph 单实体资源类型检测算法：O（n^2)

等待图是资源分配图的变形，节点都是进程，Pi->Pj 表示 Pi 在等待 Pj 释放 Pi 所需的资源。当且仅当等待图中有一个环，系统死锁

多实体资源类型检测算法：O(M * N * N)

类似银行家安全状态检测算法，但是此时 need 变成了 request

Avoidance 和 Detection 两个部分的区别在于，前者是在资源分配前预先判断是否可能有一条一定安全的路径，因为给定的是进程请求资源的 MAX 估计，实际上并不一定会达到这个状态，不安全也不一定会死锁；而后者给定的是 REQUEST，即当前已经同时收到了这么多资源请求，如果不能全部分配那就是死锁。

死锁检测算法的应用

检测算法的调用时刻及频率取决于：死锁发生频率以及死锁发生时受影响的进程数。如果经常发生死锁，那么就要经常调用检测。如果在不确定的时间调用检测算法，资源图可能有很多环，通常不能确定哪些造成了死锁

死锁恢复

检测到死锁后：通知管理员 系统自己恢复。打破死锁的方法：抢占资源 进程终止。

进程终止

两种方法来恢复死锁：终止所有死锁进程、一次终止一个进程直到不死锁。许多因素都影响终止进程的选择：优先级 进程已经计算了多久，还要多久完成 进程使用了哪些类型的资源 进程还需要多少资源 多少进程需要被终止 进程是交互的还是批处理的

抢占资源

选择一个牺牲品 victim：要代价最小化

回滚：回退到安全状态，但是很难，一般需要完全终止进程重新执行

饥饿问题：保证不会总是回滚同一个进程。常见方法是为代价因素加上回滚次数。

Main Memory 主存（易失）

逻辑地址/虚地址/相对地址：由 CPU 生成，首地址为 0，逻辑地址无法在内存中读取信息。

物理地址/实地址/绝对地址：内存中储存单元的地址，可以直接寻址。

物理地址中的逻辑地址空间是通过一对基址寄存器和界限地址寄存器控制的：

Dynamic relocation using a relocation register [base, base+limit)

地址绑定的三种情况：

编译时间：如果编译时就知道进程在内存中的地址，那么就可以生成绝对代码 absolute code。

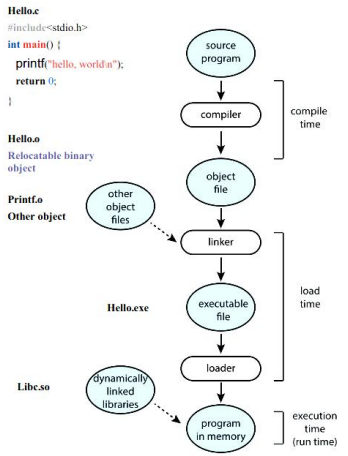
装载时间：编译时不知道在哪，那么编译器生成可重定位代码 relocatable code。

执行时间：如果进程在执行时可以移动到另一个内存段，需要硬件支持也就是 base and limit **目前绝大多数都是采用这种。**

Memory-Management Unit (MMU)

就是将虚拟地址映射到物理地址的硬件设备。在 MMU 中，base 寄存器叫做重定位寄存器，用户进程送到内存前，都要加上重定位寄存器的值。PA=relocation reg+LA。用户程序只能处理 LA，永远看不到真的 PA。

Dynamic Loading（动态加载）



进程大小会受到物理内存大小的限制，为了有更好的空间使用率，采用动态加载。一个子程序只有在调用时才被加载，所有子程序都可以重定位(relocatable load format)的形式存在磁盘上，需要的时候装入内存中。OS 不需要特别支持，是程序设计做的事。当需要大量的代码来处理一些不常发生的事时很有用，如错误处理。

Dynamic Linking 动态链接

在运行时动态链接共享库(shared library)

Swapping（交换技术）

进程可以暂时从内存中交换到备份存储 backing store 上，当需要再次执行时再调回。需要动态重定位 dynamic relocate

备份存储(backing storage)：是快速硬盘，而可以容纳所有用户的所有内存映像，并为这些内存映像提供直接访问，如 Linux 交换区 windows 的交换文件 pagefile.sys

Roll out roll in：如果有一个更高优先级的进程需要服务，内存交换出低优先级的进程以便装入和执行高优先级进程，高执行完后低再交换回内存继续执行。

交换时间的主要部分是转移时间 transfer time。总转移时间与所交换的内存大小成正比。系统维护一个就绪的可立即运行的进程队列，并在磁盘上有内存映像。

***可分为进程级(std)和页级**

Contiguous Allocation（连续分配）

内存通常分为两个区域：一个驻留 resident 操作系统，一个用于用户进程，由于中断向量一般位于低内存，所以 OS 也放在低内存。重定位寄存器用于保护各个用户进程以及 OS 的代码和数据不被修改。Base 是 PA 的最小值；limit 包含了 LA 的范围，每个 LA 不能超过 Limit。MMU 地址映射是动态的。

Multiple-partition allocation：分区式管理将内存划分为多个连续区域叫做分区，每个分区放一个进程。有**固定分区**和**动态分区**两种。

固定分区：

类似分页，将内存分成多个固定大小的分区，没有外部碎片，但有内部碎片

动态分区：

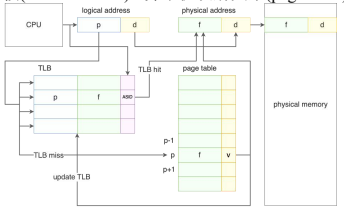
动态划分内存，在程序装入内存时切出一个连续的区域 hole 分配给进程，分区大小恰好符合需要。操作系统需要维护一个表，记录哪些内存可用哪些已用。从一组可用的 hole 选择一个空闲 hole 的常用算法 first best（最多碎片）worst-fit 三种。First 和 best 在时间和空间利用率都比 worst 好。*还有一个 next-fit 是每次都从上次查找结束的位置开始找

碎片 fragmentation

外碎片指所有的总可用内存可以满足请求，但是并不连续。外碎片可以通过紧凑压缩 compaction 拼接来减少。动态并且在执行时间完成的重定向可以进行紧凑操作，但是拼接的开销很大。内碎片是固定分区的问题，可以通过多种大小固定分区缓解。

分页存储管理

分页允许进程的 PA 空间非连续；将物理内存分为固定大小的块，叫做帧 frame/物理块/页框，将逻辑内存也分为同样大小的块叫做页 page，Linux Win(x86)是 4KB。OS 需要跟踪所有空闲帧(free-frame list)，并维护映射关系(page table)



页表的实现

页表放在内存中，PTBR(page-table base reg)指向页表(如 RiscV 中 satp) PRLR page-table length register 说明页表长度。**页表是每个进程不同的，而 TLB 是全局共享的**。TLB 维护了 ASID addresspace identifier，用来唯一地标识进程，为进程提供空间保护，否则每次切换根页表需要 flush TLB。*Associative memory:一种支持并行搜索的内存，如果虚页号与其中键匹配上，则直接返回物理帧。

Effective Access Time 有效访问时间 EAT

effective memory-access time

$$= \underbrace{\text{hit ratio} \times \text{memory-access}}_{\text{TLB hit}} + \underbrace{(1-\text{hit ratio}) \times 2 \times \text{memory-access}}_{\text{TLB miss}}$$

$$= p_{hit} \times t + (1 - p_{hit}) \times 2t$$

$$= (2 - p_{hit})t$$

*可能还要加上访问 TLB 时间

保护 protection

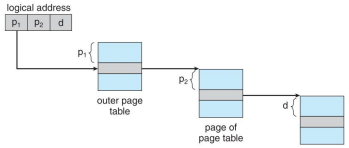
内存保护通过与每个帧关联的保护位实现，如 Valid bit 存在页表中的每一个条目上。

Shared code 共享代码：

如果代码是可重入即只读代码 reentrant code 或纯代码 pure code，可以共享，共享代码在各个进程中的逻辑地址空间相同。然后每个进程再花较小的空间保存私有代码和数据即可。

分级页表 Hierarchical page table

逻辑地址空间很大，导致页表要**连续**的大空间，因此要将页表划分的**连续空间变小**。因此可以采用多级页表，增加部分开销（有些节点实际不一定扩展，可能减小），减小连续块大小：



*RV39 三级页表 9 + 9 + 9 + (12)

某计算机采用二级页表的分页存储管理方式，按字节编址，页大小为2^{10}字节，页表项大小为2字节，逻辑地址结构为：

页目录号	页号	页内偏移量

逻辑地址空间大小为2^{16}页，则表示整个逻辑地址空间的页目录表中包含表项的个数至少是

A. 64　B. 128　C. 256　D. 512

一页可存放2^9个页表项，则至少需要$2^{16}/2^9=2^7$=128 个一级页表项

哈希页表

超过 32 位 LA 地址空间时，一般采用哈希页表，将虚页号存到哈希表里，每一项都是链表，链着哈希值相同的页号，查表时用虚页号与链表中的每个元素进行比较从而查物理表号

clustered page table: 每一项对应多个，适用 sparse address spaces, where memory references are non-contiguous and scattered

反向页表

对于每个 physical frame 有一个条目。每个条目包含映射到该 frame 的虚拟页的**虚地址及拥有该页的进程 PID**。因此整个系统只有一个页表，对每个物理内存的帧也只有一条相应的条目。拿时间换空间，需要为页表条目中添加一个地址空间标识符 ASID。

分段 Segmentation

分页无法避免的是用户视角的内存和物理内存的分离。分段管理支持用户视角的内存管理方案，LA 空间是由一组段组成的，每个段都有其名称和长度，地址指定了段名称和段内偏移。因此 LA 通过有序对<segment-number, offset>构成。段表将用户定义的二维地址映射成一维，每一个条目包含 base 和 limit。STBR(segment table base reg)指向内存中段表的位置，STLR(segment table length reg)表示进程的段数，s-number < STLR。同样有 valid 位，还有读写执行的权限设置，也可以进行 code share。内存分配本质是动态存储分配问题。

***segment with paging: 每个段维护自己的页表，这样又会出现内部碎片了**

Virtual Memory 虚存

用户的逻辑存储和物理存储分开；LA 空间可以大于 PA 空间；允许 PA 空间被多个进程共享。

局部性原理：时间：指令的一次执行和下次执行 数据的一次访问和下次访问都集中在一个较短时期内；**空间**：当前指令和邻近的指令 当前数据和邻近的数据都集中在一个小区域内。虚存具有请求调入功能和置换功能，仅把进程的一部分装入内存便可运行进程，能从逻辑上对内存容量进行扩充的

按需调页 Demand Paging

在需要时才调入相应的页。lazy swapper：除非需要页面，否则不进行任何页面置换。

页错误 Page fault

非法地址访问和不在主存或无效的页都会 page fault。其中判断到底是不是非法需要到 PCB 另一个页表中判断。

*Page fault rate 等于 1 不代表 every page is a page fault(reference)

更完整的页表项 请求分页中

虚拟页号 物理帧号 状态位 P(存在位 页是否已调入内存) 访问字段 A(记录页面访问次数) 修改位 R/W(调入内存后是否被修改过) 外存地址(用来调页)

Effective memory-access time 有效访问时间

EAT(Effective Access Time) = (1 - p) * (memory access time) +
p * (page fault overhead + swap page out time + swap page in time)

下一次 mem-access 算到重启后的去了

为了计算 EAT，必须知道需要花多少时间处理 page fault，page fault 会引起以下动作的产生：

1.陷入 trap 到 OS
2.保存用户 reg 和进程状态
3.确定中断是否为 page fault
4.检查页引用是否合法并确定所在磁盘位置
5.从磁盘读页到内存的空闲帧(包含磁盘队列中的等待 磁盘的寻寻道 旋转延迟 磁盘的传输延迟)
6.在等待过程中的 CPU 调度
7.IO 中断
8.保存其他用户寄存器和进程状态（如果进行了 6）
9.确定中断是否来自磁盘
10.修正页表和其他相关表，所需页已在内存中
11.等待 CPU 再次分配给本进程
12.恢复用户寄存器、进程状态和新页表，重启

其中的三个主要 page fault 时间是缺页中断服务时间 缺页读入时间和重启时间

写时复制 copy-on-write

允许父子进程共享同一页面，在某个进程要修改共享页时，它才会拷贝一份该页面进行写。COW 加快了进程创建速度。当确定一个页采用 COW 时，zero fill on demand 按需填零页（zeroed-out pages）需要在分配前填 0. **Win linux solaris 都用了 COW**

页面置换

寻找一些内存中没有使用的页换出去。内存的过度分配 over-allocation 会导致 page fault 调页后发现所有页都在使用。

使用 dirty/modify 位来减少页传输的开销，只有脏页才需要写回硬盘。

基本页面置换过程：

1.查找所需页在磁盘上的位置。2.查找空闲帧，如果有直接使用；如果没有就用置换算法选择一个 victim，并将 victim 的内容写回磁盘(dirty)，改变页表和帧表。3.将所需页读入新的空闲帧，改变页表和帧表。4.重启用户进程。

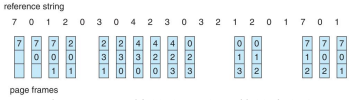
页面置换算法

引用序列叫做引用串 reference string。

注意两个事实：给定页大小，只需要关心页码，不用管完整地址；紧跟页 p 后面对页 p 的引用不会引起页错误。

FIFO 先进先出算法

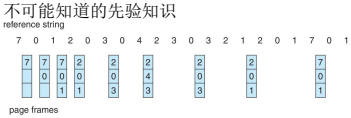
最简单的页面置换算法。必须置换一页时，选择最旧的。不需要记录时间，只需要 FIFO 队列来管理页即可。15 次缺页



FIFO 会出现可用帧越多，错误数越大的问题，这种结果叫 Belady's Anomaly

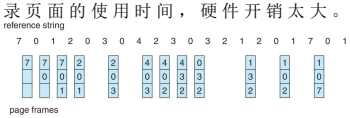
Optimal Page Replacement OPT

OPT 时所有算法中页错误率最低，且绝对没有 Belady 异常。置换最长时间不会使用的页,需要不可能知道的先验知识



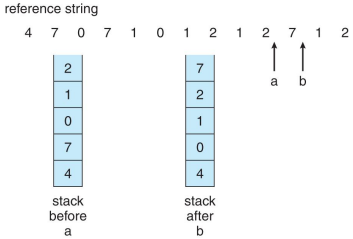
Least Recently Used LRU 最近最久使用

LRU 选择内存中最久没有引用的页面，考虑的是局部性原理，性能最接近 OPT，但是需要记录页面的使用时间，硬件开销太大。



计数器 counter：每一个页表条目都有一个 counter，每次被引用，就把时钟信息复制到 counter。当置换时，置换时间最小的页

栈实现：维护一个页码栈，栈由双向链表实现。引用页面时将该页面移动到顶部，需要改变 6 个指针。替换时直接替换栈底部就是 LRU 页。



LRU Approximation LRU 近似

很少有计算机有足够的硬件支持真正的 LRU，因此许多系统为页表中的每项关联一个引用位 reference bit，初始化为 0，当引用一个页时，对应页面的引用位设为 1，替换时替换掉引用位为 0 的(存在的话)

Additional reference bits 附加引用位算法

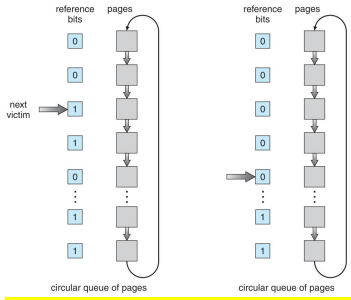
对上面 1 个引用位的补充，变成 k 个引用位，每个定时时钟中断更新，选择最小的。

reference = (ref << k-1) | (reference >> 1)

Second chance 二次机会/clock 算法 NRU

基本算法是 FIFO，选择页时，检查引用位，如果为 0 直接置换。如果为 1，给该页第二次机会，将 reference 置为 0，选择下一个 FIFO 页。

一种实现二次机会算法的方法是采用循环队列，用一个指针表示下一次要置换哪一页。当需要一个帧时，指针向前移动知道找到一个引用位 0 的页，在其向前移动的过程中，它会清除引用位。最坏情况下所有帧都会被给二次机会，他就会清除所有引用位之后再选择页进行置换，此时二次机会=FIFO。



Enhanced Second chance 改进 clock 增强二次机会

通过将引用位和脏位作为有序对来考虑，可以改进二次机会算法。两个位有四种可能：

(0,0)无引用无修改，置换的最佳页
(0,1)无引用有修改，置换前需要写回脏页
(1,0)有引用无修改，很可能会继续用
(1,1)有引用有修改，很可能会继续用且置换前须要写回脏页

淘汰次序(0,0)-> (0,1)-> (1,0)-> (1,1)

当页面需要被置换时，使用时钟算法，置换(0,0)的页，在进行置换前可能要多次搜索循环队列。改进的点子在于给未引用但是修改了的页更高优先级，降低了 IO 数。（Macintosh 使用）

Counting 基于计数的置换算法

每个页保存一个用于记录引用次数的计数器：

Least frequently used LFU：置换计数最小的。但是有问题：一个页可能一开始狂用，但是后来不用了，他的计数可能很大，但是不会被替换。解决方法是定期右移次数寄存器。

Most frequently used MFU：置换计数最大的，因为最小次数的页可能刚调进来，还没来得及用。

这两种很没用，实现开销大，很难近似 OPT。

Page Buffering 页面缓冲

通过被置换页面的缓冲，有机会找回刚被置换的页。

被置换页面的选择和处理：用 FIFO 选择置换页，把被置换的页面放到两个链表之一。即：如果页面无修改，将其归入空闲页链表，否则归入已修改页面链表。

需要调入新页面时，将新页面内容读入空闲页面链表的第一项所指的页面，然后将其删除。

帧分配 allocation of frames

每个进程都需要最小数目的页。

Fixed allocation

平均分配算法 Equal allocation

每个如果有 100 个帧 5 个进程，每一个进程获得 20 个帧。

按比例分配 Proportional allocation

根据进程的大小按比例分配。

优先级分配 Priority allocation

同样按比例分配，但是用优先级进行比例分配。

全局置换 global allocation

允许一个进程从所有帧中选择一个帧进行替换，不管该帧是否已分配给其他进程。

效率不稳定

局部置换 local allocation

每个进程只能从自己的分配帧中进行置换。

内存利用率低

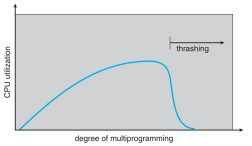
固定分配局部置换 可变分配全局置换 可变分配局部置换

Reclaiming Pages free pages 数量降到一定值后提前选择一些 page replaced 掉。

Major/Minor Page Fault Major:访问的页不在内存中；Minor:访问的页在内存中（shared library；某页被 reclaimed 了但还没实际换出）

颠簸 抖动 Thrashing

频繁的页调度行为叫做颠簸，会导致：CPU 利用率低、OS 认为多道程序程度需要增加、其他进程进入到系统中



颠簸就等价于一个进程不断换入换出页

按需调页能成的原因是局部性原理，进程从一个局部性移动到另一个，局部性可能重叠。为什么颠簸会发生，因为局部大小大于总内存大小，不能将全部经常用的页放到内存中。

工作集合模型 Working set model

WS 工作集：最近 Delta 个页的引用
Delta=工作集窗口=固定数目的页引用
WSS_i 进程 Pi 的工作集大小=在最近的 delta 内总的页面引用次数
如果 delta 太小，不能包含整个局部；delta 太大，可能包含过多局部；delta 无穷工作集为进程执行所接触到的所有页的集合。
D=WSS 求和=帧的总需求量，m=帧总可用量。
如果 D>m 就会发生颠簸，暂停一个进程。

跟踪工作集合模型

OS 跟踪每个进程的 WS，并为进程分配大于其 WS 的帧数，如果还有空闲帧，那么可以启动另一个进程。如果所有 WS 的和的增加超过了可用帧的总数，那么 OS 会暂停一个进程，该进程的页面被换出，且其帧可以被分配给其他进程挂起的进程可以在以后重启。这样的策略防止了颠簸，提高了多道程序的程度，优化了 CPU 使用率。

WS 窗口是一栋窗口，每次引用时，会增加新引用，最老引用会丢失。如果一个页在 WS 窗口内被引用过，那么他就处于 WS 中。

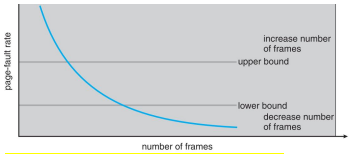
通过固定定时中断和引用可以模拟 WS 模型。假设 delta=10000 个引用，且每 5000 个会出现中断。当中断出现，先复制再清除所有页的引用位。出现 page fault 后，可以检查当前引用位和位于内存内的两个位，确定在过去的 10000 到 15000 个引用之间该页是否被引用过。如果使用过，至少有一个位为 1.如果没有使用过，3 个位全是 0。只要有一个 1，那么可以认为处于 WS 中。这种安排并不完全准确，因为并不知道在 5000 个引用的何处出现了引用。通过增加历史位的位数和终端频率可以降低不确定性，但是开销也会变大。

页错误频率 Page fault frequency schema

WS 模型能用于预先调页，但是控制颠簸不是很灵活，更直接的方法是 PFF。

可以为所期望的页错误设置一个上限和下限，如果页错误率超过上限，那么分配更多的帧，如果低于下限，那么可以从进程中移走帧。

*OOM killer



Memory-Mapped Files 内存映射文件

使用虚存技术将文件 IO 作为普通文件访问。开始的文件访问用按需请求调度，会出现页错误。这样，一页大小的部分文件从文件系统中读入物理页，以后的文件访问就可以按照通常的内存访问来处理，这样就可以用内存操作文件，而非 read write 等系统调用，简化了文件访问和使用。多个进城可以允许将同一文件映射到各自的虚存中，达到数据共享的目的。

Allocating Kernel Memory 内核内存分配

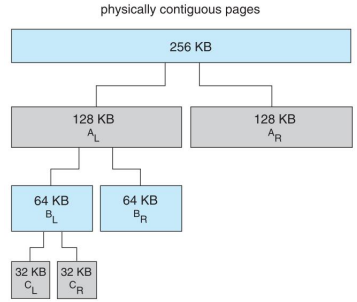
与对待用户内存不同：内核内存从空闲内存池中获取，两个原因：1.内核需要为不同大小的数据结构分配内存。2.一些内核内存需要连续。

Buddy 系统

从物理上连续的大小固定的段上进行分配。

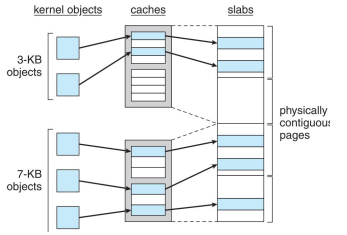
内存分配按 2 得幂的大小来分配：

请求大小必须是 2 的幂；如果不是，那么调整到下一个更大的 2 的幂；当需要比可用的更小的分配时，当前块分成两个下一个较低幂的段。继续这一过程直到适当大小的块可用。Buddy 系统的优点是可以通过合并快速形成更大的段。明显缺点是由于调整到下一个 2 的幂容易产生内碎片。



Slab 分配

为了解决 Buddy 碎片损失的问题，slab 是由一个或多个物理上连续的页组成的。Cache 包含一个或者多个 slab。每个内核数据结构都有一个 cache。每个 cache 都含有内核数据结构的对象实例。当 cache 被创建时，起初包括若干标记为空闲的对象。对象的数量和 slab 大小有关，12KB 的 slab(包含三个连续的页)可以存储 6 个 2KB 的对象。当需要内核数据结构的对象时，可以直接从 cache 上取，并将该对象标记为使用 used。Slab 首先从部分空闲的 slab 中分配，如果没有则从全空的 slab 进行分配。如果没有，从物理连续页上分配新的 slab，把他赋给一个 cache，再从 slab 分配空间。当于预先把内存分成了不同大小的对象，然后根据要放的东西的大小选择合适的空间。**Slab 优点：没有碎片引起的内存浪费；内存请求可以快速满足。**



预调页 prepaging

为了减少冷启动时大量的页错误。

同时将所有需要的页一起调入内存，但是如果预调页没有被用到，那么 IO 就被浪费了。

假设 s 页被预调到内存，其中 a 部分被用到了。问题在于节省的 s*a 个页错误的成本是大于还是小于其他 s*(1-a)不必要的预调页开销。如果 a 接近于 0，调页失败，a 接近 1，调页成功。

页大小

页大小的考虑因素：碎片ě、页表大小ê、IO 开销ê、局部性ě、**page fault** 数量ê、**TLB** 大小及效率ê

TLB 范围 TLB reach

TLB 范围指通过 TLB 可以访问到的内存量。

TLB Reach=TLB size * Page Size。

理想情况，每个进程的 WS 应该位于 TLB 中，否则就会有不通过 TLB 调页导致的大量 IO

增大页大小来缓解 TLB 压力，但可能会导致不需要大页表的进程带来的内碎片；提供多种页大小的支持，那么 TLB 无法硬件化，性能降低。

IO 互锁

IO 互锁指页面必定有时被锁在内存中。

必须锁住用于从设备复制文件的页，以便通过页面置换驱逐。

File System Interface 文件系统接口

文件是存储某种介质上的（如磁盘、光盘、SSD 等）并具 有文件名的一组相关信息的集合

文件属性

名称 标识符(唯一标识该文件的数字) 类型 位置 大小 保护 时间日期 用户标识

所有的文件信息都保存在目录结构中，而目录结构保存在外存上。

文件操作

文件是 ADT 抽象数据类型：创建 写 读 文件内重定位 删除 截短(truncate) Open(Fi)在硬盘上寻找目录结构并且移动到内存中 Close(Fi)将内存中的目录结构移动到磁盘中。

打开文件

每个打开文件都有以下信息：文件指针：跟踪上次读写位置作为当前文件位置指针

文件打开计数器 file-open count：跟踪文件打开和关闭的数量，在最后关闭时，计数器为 0，系统可以移除该条目。

文件磁盘位置 disk location of file：用于定位磁盘上文件位置的信息

访问权限：访问模式信息

锁机制：mandatory lock:根据目前锁请求与占有情况拒绝 access；advisory lock:进程查看锁情况来决定访问策略

文件内部结构 File Structure

None 字/字节的序列流文件结构, Unix

Simple record structure 记录文件结构：lines, fixed length, variable length，如数据库

Complex Structures ： formatted document, relocatable load file

可以通过插入适当的控制字符，用第一种方法模拟最后两个。这些模式由 OS 和程序所决定。

文件类型：

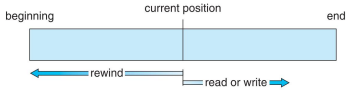
Mac OS: creator attribute

Unix: Magic number in head

访问方法

Sequential access 顺序访问

文件信息按顺序，一个记录接着一个记录处理。访问模式最常会能够用，编辑器和编译器用这种方式。读操作读取文件下一文件部分，并自动前移文件指针，跟踪 IO 位置。写操作向文件尾部增加内容，相应文件指针到新文件结尾。顺序访问基于文件的磁带模型，也适用于随机访问设备。可以重新设置指针到开始位置或者向前向后跳过记录。No read after last write。



Direct access 直接访问/相对访问/随机访问

文件由固定长度的逻辑记录组成，允许程序按任意顺序进行快速读写，直接访问是基于文件的磁盘模型。文件可作为块或记录的编号序列。读写顺序没有限制。可以立即访问大量信息，DB 常用。往往用指向 blocks 的 index 实现。

文件操作必须经过修改从而能将块号作为参数，有读 n 操作，而不是读下一个；写 n 操作；定位到 n；要实现读 n 只需要定位 n 再读下一个即可。注意 n 是相对块号，相对于文件开始的索引号。

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

Indexed block access 索引顺序访问访问

目录结构

目录是包含所有文件信息节点的集合。目录结构和文件在磁盘上。

磁盘结构

磁盘可以装多种文件系统，分区或片 minidisk slice。

目录操作

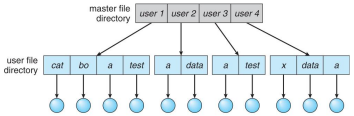
搜索文件 创建文件 删除文件 遍历 list 目录重命名文件 遍历 traverse 文件系统

单级目录

所有文件包含在同一目录中，一个文件系统提供给所有用户。由于所有文件在同一级，不能有重名，此外存在着分组问题

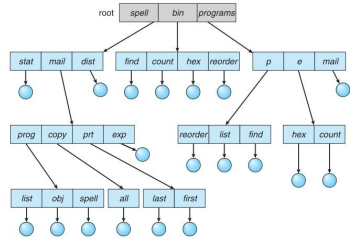
二级目录

为每个用户创建独立目录。每个用户都有自己的用户文件目录 user file directory UFD。不同用户可以有同名文件，搜索效率高，但是没有分组能力。



树形目录

将二级目录拓展即可。搜索高效 有分组能力。主流操作系统使用



无环图目录 Acyclic graph

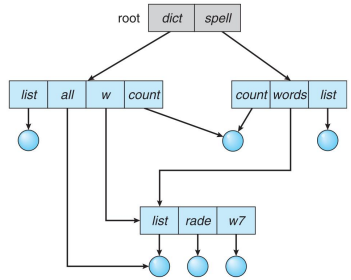
树形结构禁止共享文件和目录。无环图允许目录含有共享子目录和文件。

UNIX 采用链接的方式，软链接是一个路径指针，直接解析；而硬链接是一个目录条目。（删除源文件，软链接失效，硬链接仍然有效）

别名问题（aliasing）：一个文件可以有多个绝对路径名。不同文件名可能表示同一文件

dangling pointer 问题：删除一个文件后指向该文件的其他链接成为 dangling。

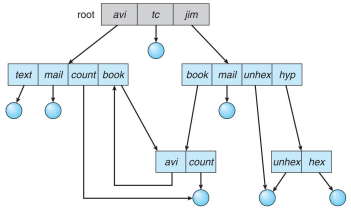
在 inode 中保留一个引用计数。**通过禁止对目录的多重引用，可以维护无环图结构（硬链接）**



普通图目录 General graph

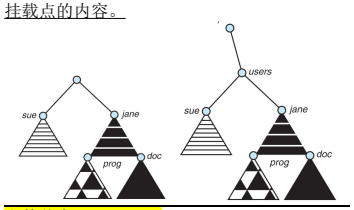
采用这种目录必须确保没有环，仅允许向文件链接，不允许目录，或者允许环，但使用 gabage collection 回收空间。

每次加入链接都要执行环检测算法。



文件系统挂载 mount

文件系统在访问前必须挂载，左图是未安装的卷，右图的 users 为挂载点，挂载会暂时覆盖挂载点的内容。



文件共享 file sharing

多用户系统的文件共享很有用。文件共享需要通过一定的保护机制实现：在分布式系统，文件通过网络访问；网络文件系统 NFS 是常见的分布式文件共享方法。NFS 是 UNIX 文件共享协议 CIFS 是 WIN 的协议。

保护 Protection

访问类型：读 写 执行 追加（append）删除 列表清单（list）

访问控制列表 access-control list ACL

拥有者 owner access

组 group access

其他 public access

在 UNIX 里，一个类型有 rwx 三个权限，所以一个文件需要 3*3=9 位说明文件访问权限。

File System Implementatio

文件系统：是操作系统中以文件方式管理计算机软件资源的软件和被管理的文件和数据结构（如目录和索引表等）的集合。文件系统储存在二级存储中，磁盘。

write() & fsync():

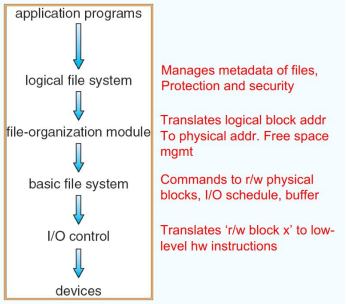
write()为在未来某个时间将数据写回到二级存储中，具体时间由文件系统决定；

fsync()强制将 dirty data 写回 disk。

(soft link can cross volume, hard link can't)

分层设计的文件结构

逻辑文件系统通过 FCB（打开文件）来维护文件结构。



文 件 系 统 实 现

Disk structures

- Boot control block (per volume)
- Volume control block per volume (superblock in Unix)
- Directory structure per file system
- Per-file FCB (inode in Unix)

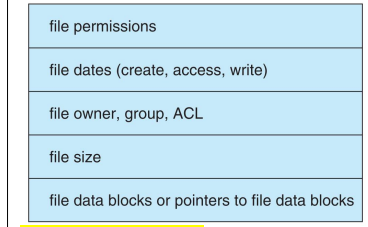
In-memory structures (see fig)

- *In-memory mount table* about each mounted volume
- *Directory cache* for recently accessed directories
- System-wide open-file table
- Per-process open-file table

On-Disk FS structure

如何启动硬盘中 OS，硬盘中包括的 block 总数、空闲 block 的数量和位置，目录结构，文件个体等

FCB，磁盘上元信息，文件打开时复制到内存



In-Memory FS structure

帮助文件系统管理和一些缓存操作

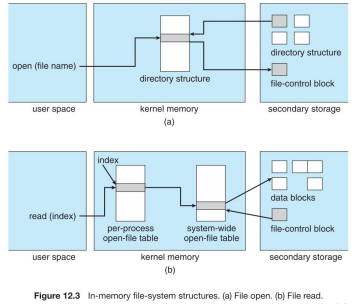


Figure 12.3 In-memory file-system structures. (a) File open. (b) File read.

System-Wide Open-File Table：记录所有被 load 到内存中的 FCB inode；

Per-Process Open-File Table：指向上表的项（包含当前在文件中位置、文件访问模式等）。

虚拟文件系统 VFS

VFS 提供面向对象的方法实现文件系统,不是 on-disk.允许将相同的系统调用接口（API）用于不同类型的文件系统。(Write syscall -> vfs_write)

属于 file-organization

superblock object（文件系统的控制）,inode object（文件控制块）,dentry object（目录条目）,file object（打开的文件）

目录实现

线性列表 linear list：使用储存文件名和数据块指针的线性表。

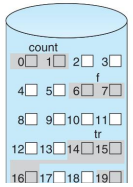
哈希表：线性表与哈西结构，哈希表根据文件名得到一个值返回一个指向线性表中元素的指针。

分配方法 Allocation Method

连续分配 Contiguous Allocation

每个文件在磁盘上占有一组连续的块。优点：访问很容易，只需要起始块位置和块长度就可以读取，支持 random access。但是浪费空间，存在动态存储分配问题。First 和 best 表现差不多，first 时间快很多。存在外碎片问题，此外**文件大小不可增长。**

```
block_size = 512 # Bytes
Q = LA / block_size
R = LA % block_size
```



directory

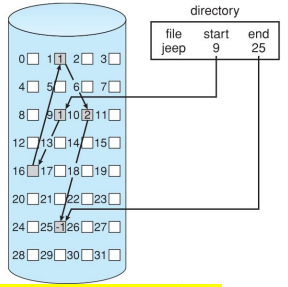
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

变种：基于长度的系统，利于 Veritas FS 采用。解决了文件大小无法增长的问题，增加了另一个叫做 extent 的连续空间给空间不够的文件，然后与原文件块之间有个指针。一个文件可以有多个 extent，有内部碎片问题。

Linked Allocation 链接分配

解决了连续分配的所有问题。每个文件都是磁盘块的链表。访问起来只需要一个起始地址。没有空间管理问题，不会浪费空间，但是不支持 random access。

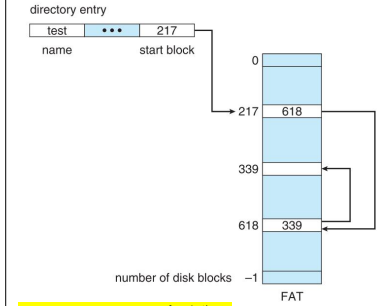
```
valid_block_size = 512 - 1 # Bytes
Q = LA / valid_block_size
R = (LA % valid_block_size) + 1
```



FAT File allocation table 文件系统

磁盘空间分配用于 MS-DOS 和 OS/2。FAT32 引导区记录被扩展为包括重要数据结构的备份，根目录为一个普通的簇链，其目录项可以放在文件区任何地方。原本的链接分配有问题，指针在每个块中都会占空间，可靠性也不高，任何指针丢失都会导致文件其余部分丢失。FAT 采用单独的磁盘区保存链接。

计算机系统启动时，首先执行的是 BIOS 引导程序，完成自检， 并加载主引导记录和分区表，然后执行主引导记录，由它引导 激活分区引导记录，再执行分区引导记录，加载操作系统，最 后执行操作系统，配置系统。



Indexed Allocation 索引分配

索引分配把所有指针放在一起，通过索引块解决这个问题。每个文件都有索引块，是一个磁盘块地址的数组。当首次写入第 i 块时，先从空闲空间管理器获得一块，再将其地址写到索引块中的第 i 个条目。对于小文件，大部分索引块被浪费。如果索引块太小，可以多层索引、然后互相连接。访问需要索引表，支持 randomaccess，动态访问没有外碎片，但是有更大的索引开销。

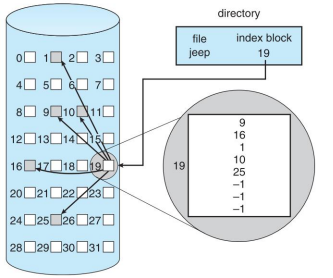
得到地址后可以直接寻址。

链接索引:

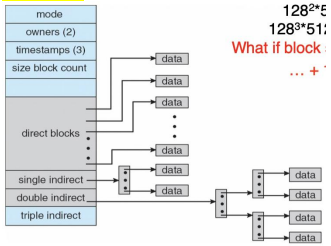
```
block_size = 512 # Bytes
normal_block_size = block_size # Bytes
ptr_valid_block_size = block_size - 1 # Bytes

Q1 = LA / (normal_block_size * ptr_valid_block_size) # the i
R1 = LA % (normal_block_size * ptr_valid_block_size)

Q2 = R1 / normal_block_size # the data block
R2 = R1 % normal_block_size # the offset in the data block
```

二级索引:

```
block_size = 512 # Bytes

Q1 = LA / (block_size ** 2) # the inner index block
R1 = LA % (block_size ** 2)

Q2 = R1 / block_size # the data block
R2 = R1 % block_size # the offset in the data block
```



Inode 结构



索引计算

连续、链接组织的最大文件大小可以大到整个磁盘文件分区。

考虑每块大小 4KB，块地址 4B。

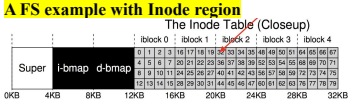
一级索引：一个索引块可以存 4KB/4B=1K 个索引地址，每个索引地址直接引到文件块，所以最大 1K*4KB=4MB。

二级索引：一个索引块可以再继续连接到索引块，因此有 1K*1K*4KB=4GB 的最大文件。

采用 Linux 分配方案，Linux 中共有 15 个指针在 inode 中，前面 12 个直接指向文件块，因此有 48KB 可以直接访问，其他三个指针指向间接块，第一个间接块指针是指向以及间接块，第二个是二级间接块，第三个是三级间接块。因此最大文件的大小为：

12*4KB+1K*4KB+1K*1K*4KB+1K*1K*1K*4KB=48KB+4MB+4GM+4TB

A FS example with Inode region



空闲空间管理

位图 bit vector：空闲置 0 占有置 1，块数计算：(number of bits per word)*(number of 0-value words)+offset of first 1bit

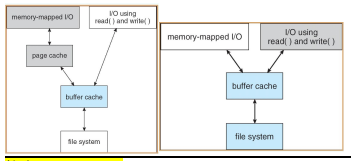
位向量所需空间的计算：disk size/block size 便于查找连续文件。

链表：将所有的空闲块链接起来，将指向第一个空闲块的指针保存在磁盘的特殊位置并且还存在内存中。但是 IO 效率很低，因为需要遍历。对空闲链表的改进是将 n 个空闲块的地址存到第一个空闲块中。这样可以快速找到大量空闲块的地址。

计数：不记录 n 个空闲块的地址，而是记录第一各空闲块和紧跟着的空闲块的数量 n。

页面缓冲 page buffer

将文件数据作为页而不是磁盘块缓冲到虚存。unified cache(统一缓存)



恢复 Revoverry

一致性检查：将目录结构数据与磁盘数据块比较，并且纠正发现的不一致。

用系统程序将磁盘数据备份到另一个设备。然后从该设备恢复。

日志结构的文件系统

日志文件系统记录文件系统的更新为事务。事务会被写到日志里。事务一旦写入日志就是已经 commit 了，否则文件系统还没更新。

Mass storage system 大容量存储

磁盘的 0 扇区是最外面的第一个磁道的第一个扇区，逻辑块最小传出单位 512B

磁盘调度

寻道时间 Seek time ：磁头移动到包含目标扇区的柱面的时间。

旋转延迟 rorational latency：旋转到目标扇区的时间（1/rpm*60,average latency=1/2*latency）。

传输时间 transfer time：数据传输时间

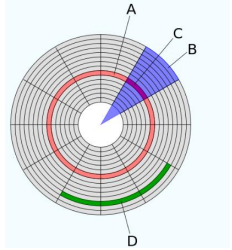
bandwidth = data / time

***A. Track (磁道)：一圈**

***B. Geometrical sector：扇形**

***C. Track sector (扇区)：一块**

***D. Cluster (簇) ： 连 续 扇 区**



调度算法

FCFS 先来先服务：算法公平，但不是最快。

SSTF 最短寻道时间优先：处理靠近当前磁头位置的请求，本质上和 SJF 一样，**低平均响应时间，高吞吐量；响应时间方差较大，可能 starvation**

SCAN：从磁盘一端到另一端，对所有路上经过的柱面进行服务。到达另一端时改变移动方向，继续处理，也叫做电梯算法。**高吞吐量；响应时间方差低（更均匀地响应）；平均响应时间低；但是刚走的地方等待较久**

C-SCAN 磁头从一端移动到另一端，到了另一端就马上返回到磁盘开始，返回路径中不服务。**更均匀的等待时间**

LOOK：磁头从一端到另一端，到达另一端最远的服务就不继续走了，开始折返服务。

C-LOOK：磁头从一端到另一端，到达另一端最远服务就立即返回到磁盘开始的第一个服务，返回路径不服务。

LOOK C-LOOK 对于高负荷 IO 磁盘表现更好，本质上就是对 SCAN 和 C-SCAN 的优化，但是表现依赖于请求类型和数量；磁盘请求又依赖于文件分配策略；磁盘调度算法应该模块化，可以随时更换自由选择。

SSTF 或者 LOOK 都是很棒的默认算法。

***NVM 通常会直接连到系统总线(system bus)，并且擦写寿命有限**

磁盘管理

低级格式化/物理格式化：将磁盘划分为扇区才能进行读写。**逻辑格式化**：创建文件系统。

要是用一个磁盘保存文件，OS 需要这么几步：首先分区，然后逻辑格式化，也就是创建文件系统；为了提升效率然后将块集中到一起成为簇 cluster。一般 bootstrap 存在 ROM 里。

系统启动顺序：ROM 中的代码 (simple bootstrap)->boot block(full bootstrap) 即 boot loader 如 Grub LILO->整个 OS 内核

RAID

0：无冗余 1：镜像 0+1/1+0：条带后镜像/镜像后条带 2：纠错码 3：按 bit 对每个盘进行奇偶校验，结果放在 1 个盘 4：与 3 类似，按块条带化 Striping 5：校验值分散到各个盘 6：P+Q 冗余，差错纠正码，2 个错误

三级存储 Tertiary storage device

Low cost is the defining characteristic of tertiary storage. Generally, tertiary storage is built using removable media Common examples of removable media are floppy disks and CD-ROMs; other types are available

Swap space 三级存储 Tertiary storage device

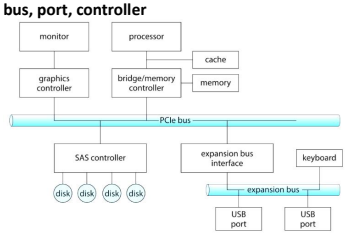
虚存使用硬盘空间作为主存。两种形式：普通文件系统：win 都是 pagefile.sys 独立硬盘分区 linux solaris 都是 swap 分区。还有一种方法：**创建在 raw 的磁盘分区上，这种速度最快。**

性能：

Sustained bandwidth 传输的平均速率 字节/时间。Effective bandwidth IO 时间下的平均速率。前者是数据真正流动时的速率，后者是驱动器能够提供的能力，一般驱动器带宽指前者。

IO system

bus, port, controller



IO 方式

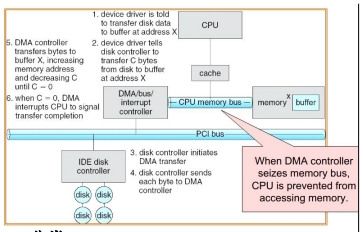
轮询 polling 硬等待

中断 CPU 硬件有一条中断请求线 IRL，IO 设备触发，需要 IO 时就申请中断。

Some Maskable, some not

DMA direct memory access

对于需要进行大量 IO 的设备，为了避免程序控制 IO 即 PIO，将一部分任务下放给了 DMA 控制器，在 DMA 开始传输时，主机向内存中写入 DMA 命令块。然后 CPU 在写入后继续干别的，DMA 去自己操作内存总线，然后就可以向内存进行传输。



IO 分类：block I/O(read, write, seek); character I/O (stream, keyboard, clock); memory-mapped file access; network sockets

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

IO 应用接口

实现统一的 IO 接口，设备驱动提供了 API 来操控 IO 设备(Linux：最底层为 **ioctl**)

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

块设备和字符设备

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: blocking 和 non-blocking

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

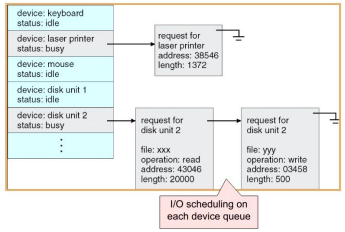
非阻塞 IO：IO 调用立刻返回尽可能多的数据。用户接口就是，接收鼠标键盘输入，还要在屏幕上输出，放视频也是，从磁盘读帧然后显示。

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**

IO 子系统

Device-status Table（device-control table）



Caching - fast memory holding copy of data
Always just a copy,Key to performance
Spooling - hold output for a device
If device can serve only one request at a time

i.e., Printing
Device reservation - provides exclusive access to a device
System calls for allocation and deallocation
Watch out for deadlock

VFS 的 dentry cache 与 inode cache

为了加速对经常使用的目录的访问， VFS 文件系统维护着一个目录项的缓存。为了加快文件的查找速度 VFS 文件系统维护一个 inode 节点的缓存以加速对所有装配的文件系统的访问。用 hash 表将缓存对象组织起来。

File 对象

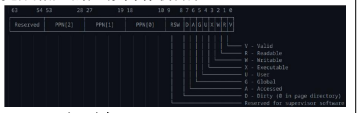
文件对象 file 表示进程已打开的文件，只有当文件被打开时才在内存中建立 file 对象的内容。该对象由相应的 open() 系统调用创建，由 close() 系统调用销毁。

实验

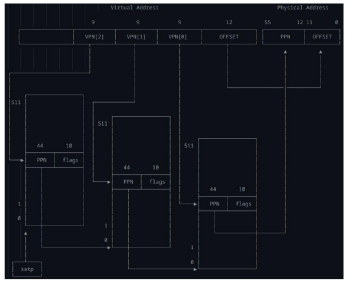
RISC-V 架构中，虚拟内存的页表实现的入口由 satp 寄存器来控制。 satp 寄存器的最高 4 位 MODE 是模式位，用于选择虚拟地址页表的架构，本次实验选用的是 SV39 Mode：接下来的 16 位称作 ASID，用于多进程的地址空间隔离；最后的 44 位 PPN 是物理页号，用于指向页表的物理地址。

当系统启动时，首先会初始化一个临时的根页表，对引导程序与内核程序的内存空间进行虚拟地址到物理地址的映射。之后，通过修改 satp 寄存器触发该根页表的加载，使得内核开始运行在虚拟地址空间中。之后，内核会初始化完成一个全局页表，对整个物理空间进行映射，并同样通过修改 satp 寄存器来加载全局页表。这一切完成后，操作系统就真正完全运行在虚拟地址空间中了。

在本次实验中，临时根页表我们直接使用一个 1GiB 的页映射，进行等值映射和线性映射，并完成相应的虚拟内存初始化。



RV39 三级页表



由于内核态是运行在物理地址上的，所以在进行页表映射时，需要将所有的虚拟地址映射回到物理地址上去建立索引；而在初始化这个全局页表时，由于前面已经启用临时页表运行在虚拟空间中，又要注意将页表项转为虚拟地址进行处理

在 Linux 系统中，vm_area_struct 是虚拟内存管理的基本单元，vm_area_struct 保存了有关连续虚拟内存区域（简称 vma ）的信息。通过其中保存的 vma 内存映射信息，可以很容易地找到数据段对应的具体磁盘数据信息，从而通过操作系统的内存管理模块，进行相应的内存调度。

在一个启用了虚拟内存的系统上，由于各种原因导致程序无法正常访问对应数据页（如访问当前未由内存管理单元（MMU）映射到虚拟内存的页面，或访问权限不足），就会由计算机硬件引发缺页异常。

一般处理缺页异常时，操作系统将根据前面提到的 vma 模块映射信息，尝试建立新的映射关系到虚拟内存;但如果在非法访问内存的情况下，即发现触发 page fault 的虚拟内存地址 Bad Address 不在当前进程的 vm_area_struct 链表中所定义的允许访问的虚拟内存地址范围内，或访问位置的权限条件不满足时，缺页异常处理将终止该程序的继续运行。

- 通过 trap_handler 捕获异常
- 寻找导致产生了异常的地址对应的 VMA
  - 如果当前访问的虚拟地址在 VMA 中没有记录（即不合法地址），则运行出错（本实验不涉及）
  - 如果当前访问的虚拟地址在 VMA 中存在记录，则需要判断产生异常的原因：
    - 如果是匿名区域，那么开辟一页空内存，然后把这一页映射到产生异常的 task 的页表中
    - 如果不是，则访问的页是存在数据的（如代码），需要从相应位置读取内容并载入，然后将对应页映射到页表中
- 返回到产生了该缺页异常的那条指令，并继续执行程序

copy-on-write 策略的基本思想是，当父进程和子进程共享同一块物理内存时，只有当其中一个进程试图写入时，才会复制一份数据。因此，实现该策略的核心是对数据页的操作进行感知。

这里实验文档中已经给出了简单的方法，对每个页面新增一个 refcnt 变量，用于记录当前页面被引用的次数。当发生 fork 时，只需要将 refcnt 加一，并将对应待复制的共享数据页的权限设置为只读 。

这样当出现写入操作时，就会触发 SCAUSE_EXC_STORE_OR_AMO_PAGE_FAULT 异常，此时再进行真正的复制操作。

习题

Suppose a system had 12 resources, 3 processes P0, P1 and P2.

	max	current	need
P0	10	5	5
P1	4	2	2
P2	9	3	6

Currently there are two resources available. This system is in an unsafe state as process P1 could complete, thereby freeing a total of four resources. But we cannot guarantee that processes P0 and P2 can complete. However, it is possible that a process may release resources before requesting any further. For example, process P2 could release a resource, thereby increasing the total number of resources to five. This allows process P0 to complete,which would free a total of nine resources, thereby allowing process P2 to complete as well.

A system has 3 concurrent processes, each of which requires 4 items of resource R. What is the minimum number of resource R in order to avoid the deadlock. Answer: 10

The system design the structure File Control Block (FCB) to manage the files. Commonly, File control block is created on disk when the open system call is invoked.

Which kind of swap space is fastest?
A raw partition

2. **文件 F 由 200 条记录组成，记录从 1 开始编号，用户打开文件后，欲将内存中的一条记录插入文件 F 中，作为其第 30 条记录，请回答下列问题，并说明理由。**

(1) 若文件系统为顺序分配方式，每个存储块存放一条记录，文件 F 的存储区域前后均有足够空闲的存储空间，则要完成上还操作最少要访问多少存储块？ F 的文件控制区内容会有哪些改变？

(2) 若文件系统为链接分配方式，每个存储块存放的一条记录和一个链接指针，则要完成上还操作最少要 访问多少存储块？若每个存储块大小为 1KB，其中 4 个字节存放指针，则该系统支撑文件的最大长度是多少？

【答案】(1) 因为要最少访问，所以选择将前 29 块前移一个存储块单元，然后将要写入的记录写入到当前的第 30 条的位置上。由于前移都要先访问原存储块将数据读出，再访问目标存储块将数据写入，所以最少需要访问 29*2+1＝59 块存储块

F 的文件区的文件长度加 1，起始块号减 1

(2) 采用链接方式则需要顺序访问前 29 块存储块，然后将新纪录的存储块插入链中即可，把新的块存入磁盘要 1 次访存，然后修改第 29 块的链接地址存回磁盘又一次访存。一共就是 29+1+1＝31 次。

4 个字节的指针的地址范围为 2^{32}。

所以此系统支撑文件的最大长度为2^{32}×（1KB-4B）＝4088GB

Q：一个文件系统中有一个 20MB 大文件和一个 20KB 小文件，当分别采用连续、链接、链接索引、二级索引和 LINUX 分配方案时，每块大小为 4096B，每块地址用 4B 表示。问：
（1）各文件系统管理的最大文件是多少？
（2）每种方案对大、小两文件各需要多少专用块来记录文件的物理地址(说明各块的用途)？
（3）如需要读大文件前面第 5.5KB 的信息和后面第（16M＋5.5KB）的信息，则每个方案各需要多少次盘 I/O 操作?

A:（1）
连续分配：理论上是不受限制，可大到整个磁盘文件区。
隐式链接：由于块的地址为 4 字节，所以能表示的最多块数为 232=4G，而每个盘块中存放文件大小为 4092 字节。链接分配可管理的最大文件为：4G×4092B=16368GB
链接索引：由于块的地址为 4 字节，所以最多的链接索引块数为 232=4G，而每个索引块有 1023 个文件块地址的指针，盘块大小为 4KB。假设最多有 n 个索引块，则 1023×n+n=232，算出 n=222，链接索引分配可管理的最大文件为：4M10234KB=16368GB
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 个索引表目。
二级索引可管理的最大文件容量为 4KB×1K×1K＝4GB。
LINUX 混合分配：LINUX 的直接地址指针有 12 个，还有一个一级索引，一个二级索引，一个三级索引。因此可管理的最大文件为 48KB＋4MB+4GB＋4TB。
（2）
连续分配：对大小两个文件都只需在文件控制块 FCB 中设二项，一是首块物理块块号，另一是文件总块数，不需专用块来记录文件的物理地址。
隐式链接：对大小两个文件都只需在文件控制块 FCB 中设二项，一是首块物理块块号，另一是末块物理块块号；同时在文件的每个物理块中设置存放下一个块号的指针。
一级索引：对 20KB 小文件只有 5 个物理块大小，所以只需一块专用物理块来作索引块，用来保存文件的各个物理块地址。对于 20MB 大文件共有 5K 个物理块，由于链接索引的每个索引块只能保存（1K－1）个文件物理块地址（另有一个表目存放下一个索引块指针），所以它需要 6 块专用物理块来作链接索引块，用于保存文件各个的物理地址。
二级索引：对大小文件都固定要用二级索引，对 20KB 小文件，用一个物理块作第一级索引，用另一块作二级索引，共用二块专用物理块作索引块，对于 20MB 大文件，用一块作第一级索引，用 5 块作第二级索引，共用六块专用物理块作索引块。
LINUX 的混合分配：对 20KB 小文件只需在文件控制块 FCB 的 i_addr[15]中使用前 5 个表目存放文件的物理块号，不需专用物理块。对 20MB 大文件，FCB 的 i_addr[15]中使用前 12 个表目存放大文件前 12 块物理块块号（48K），用一级索引块一块保存大文件接着的 1K 块块号（4M），剩下还有不到 16M，还要用二级索引存大文件以后的块号，二级索引使用第一级索引 1 块，第二级索引 4 块（因为 4KB×1K×4=16 M）。总共也需要 6 块专用物理块来存放文件物理地址。
(3)
连续分配：为读大文件前面和后面信息都需先计算信息在文件中相对块数，前面信息相对逻辑块号为 5.5K / 4K=1（从 0 开始编号），后面信息相对逻辑块号为（16M＋5.5K）/4K=4097。再计算物理块号＝文件首块号＋相对逻辑块号，最后化一次盘 I/O 操作读出该块信息。
链接分配：为读大文件前面 5.5KB 的信息，只需先读一次文件头块得到信息所在块的块号，再读一次第 1 号逻辑块得到所需信息，共 2 次。而读大文件 16MB＋5.5KB 处的信息，逻辑块号为（16M＋5.5K）/4092=4107，要先把该信息所在块前面块顺序读出，共化费 4107 次盘 I / O 操作，才能得到信息所在块的块号，最后化一次 I/O 操作读出该块信息。所以总共需要 4108 次盘 I / O 才能读取（16MB+5.5KB）处信息。
链接索引：为读大文件前面 5.5KB 处的信息，只需先读一次第一个索引块得到信息所在块的块号，再读一次第 1 号逻辑块得到所需信息，共化费 2 次盘 I / O 操作。为读大文件后面 16MB+5.5KB 处的信息，(16MB+5.5KB)/(4KB×1023)=4,需要先化 5 次盘 I / O 操作依次读出各索引块，才能得到信息所在块的块号，再化一次盘 I/O 操作读出该块信息。共化费 6 次盘 I / O 操作。

二级索引：为读大文件前面和后面信息的操作相同，首先进行一次盘 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处信息都只需要 3 次盘 I / O 操作。

LINUX 混合分配：为读大文件前面 5.5KB 处信息，先根据它的相对逻辑块号，在内存文件控制块 FCB 的 i_addr 第二个表目中读取信息所在块块号，而只化费一次盘 I / O 操作即可读出该块信息。为读大文件后在（16MB＋5.5KB）信息，先根据它的相对逻辑块号判断要读的信息是在二级索引管理范围内，先根据 i_addr 内容化一次盘 I / O 操作读出第一级索引块，再计算信息所在块的索引块号在第一级索引块的表目号为（4097-12-1024）/ 1024＝2，根据第一级索引块第 3 个表目内容再化费一次盘 I / O 操作，读出第二级索引块，就可以得到信息所在块块号，最后化一次盘 I / O 读出信息所在盘块，这样总共需要 3 次盘 I / O 操作才能读出文件后面的信息。

DOS (Disk Operating System) single-user, single-tasking
OS/2: single-user operating system, though it supports multitasking.
Windows XP: single-user operating system
Linux: multi-user, time-sharing environments.
Micro-kernels allow some system services to be implemented just as user programs
管道可以进程间通信，重定向一般不能直接做到
Single-contiguous-allocation：同一时间只能有一个进程在用户区域中运行
高响应比优先调度：综合考虑

3. 高响应比优先调度算法

高响应比优先调度算法主要用于作业调度，是对 FCFS 调度算法和 SJF 调度算法的一种综合平衡，同时考虑了每个作业的等待时间和估计的运行时间。在每次进行作业调度时，先计算后备作业队列中每个作业的响应比，从中选出响应比最高的作业投入运行。

响应比的变化规律可描述为

$$响应比_R = \frac{等待时间+要求服务时间}{要求服务时间}$$

根据公式可知：①作业的等待时间相同时，要求服务时间越短，响应比越高，有利于短作业，因而类似于 SJF。②要求服务时间相同时，作业的响应比由其等待时间决定，等待时间越长，其响应比越高，因而类似于 FCFS。③对于长作业，作业的响应比可以随等待时间的增加而提高，当其等待时间足够长时，也可获得 CPU，克服了“饥饿”现象。

FAT12/16	早期 Windows/DOS	简单，兼容性好 (老系统)	容量小，安全性差，基本淘汰
FAT32	Windows 9x, U 盘、移动硬盘等	兼容性好	单个文件大小有限制 (4GB)，安全性较差
exFAT	闪存设备 (U 盘、SD 卡等)	突破 FAT32 文件大小限制	兼容性不如 FAT32
NTFS	Windows NT 系列 (Windows XP 及以后版本)	安全性高，支持大容量磁盘，功能丰富 (加密、ACL、日志等)	兼容性不如 FAT 系列，在非 Windows 系统上支持可能有限
ReFS	Windows Server 2012 及更高版本	高可用性，数据完整性，可伸缩性	兼容性较差，主要用于服务器环境
ext2	早期 Linux	简单高效	不支持日志功能
ext3	Linux	增加了日志功能，提高了可靠性	相对较老，性能不如 ext4
ext4	Linux (目前最常用)	性能好，支持更大的文件和卷	不支持 inode 动态分配，大量小文件场景可能效率不高
XFS	Linux 服务器	高性能，擅长处理大文件和高并发访问	
Btrfs	Linux	快照、校验和、压缩等高级特性	相对较新，稳定性还在不断改进中
HFS+	macOS 早期版本		相对较老，性能不如 APFS
APFS	macOS High Sierra 及更高版本	针对 SSD 优化，性能高，安全性好，支持快照等	兼容性不如 HFS+
ISO 9660	光盘 (CD-ROM)	广泛兼容性 (光盘)	功能有限
UDF	DVD、蓝光光盘	支持大容量存储	

barber

```
semaphore barber; // init to 0
semaphore mutex; // init to 1
semaphore service; // semaphore on which barber is sleeping, init = 0
int numFull; // number of waiting customers.
barber(){
    // sleep
    while (1){
        wait (service);
        wait (mutex);
        numFull --;
        signal (barber);
        signal(mutex);
        // provide hair cut service here.
        // ...
    }
}
customer(){
    wait (mutex);
    if (numFull > K){ signal (mutex); leave(); }
    else {
        signal (service);
        numFull ++;
        signal (mutex);
        wait(barber); // enjoy hair cut but wait if barber busy.
        //...
    }
}
```

There is only one MBR (master boot record) on a disk drive, and only one boot sector
An advantage of implementing threads in user space is that they don’t incur the overhead of having the OS schedule their execution
A schedule does not care process in I/O or CPU burst.
Pure seg helps sharing
present bit not necessary in demanding page
常见 RISCV 中断：

Reserved
Supervisor software interrupt
Reserved
Machine software interrupt

Reserved
Supervisor timer interrupt
Reserved
Machine timer interrupt

Reserved
Supervisor external interrupt
Reserved
Machine external interrupt

Reserved
Counter-overflow interrupt
Reserved
Designated for platform use

Instruction address misaligned
Instruction access fault
Illegal instruction
Breakpoint
Load address misaligned
Load access fault
Store/AMO address misaligned
Store/AMO access fault
Environment call from U-mode
Environment call from S-mode
Reserved
Environment call from M-mode
Instruction page fault
Load page fault
Reserved
Store/AMO page fault
Reserved
Software check
Hardware error
Reserved
Designated for custom use
Reserved
Designated for custom use
Reserved