Contents

I 计算机系统概述

1. 操作系统的基本概念

操作系统 (Operating System, OS) 是计算机系统中最基本的系统软件。

1.1 特征

1) 并发 (Concurrence): 指两个或多个事件在同一时间间隔内发生

与之相关, 并行是指两个或多个事件在同一时刻发生

2) 共享 (Sharing)

3) 虚拟 (Virtual)

4) 异步 (Asynchronism): 多道程序环境允许多个程序并发执行.

2. 操作系统发展历程

1) 单道批处理系统 (Batch Processing)

2) 多道批处理系统 (Multiprogramming Batch Processing System): 一段时间内内存中同时存在多个进程, 即并发任务.

3) 分时操作系统 (Time Sharing Systems): 分时系统通过频繁地在多个进程间切换来近似实现并行

3. 操作系统运行

3.1 特权模式 (Privileged Mode)

将 CPU 的运行模式划分为用户态 (User Mode) 和内核态 (Kernel Mode). 特权指令 (privileged instruction), 是指不允许用户直接使用的指令. 模式的切换是通过 trap 实现的.

3.2 计时器 (timer)

通过时钟中断的管理, 可以实现进程的切换.

3.3 中断 (Interrupt)

- 中断（Interruption）: 指来自 CPU 执行指令外部的事件, 如信息, 设备的 I/O
- 异常（Exception）: 指来自 CPU 执行指令内部的事件

Figure I.1: 内中断和外中断的联系与区别

3.4 系统调用 (System Call)

用户在程序中调用操作系统所提供的一些子功能, 系统调用可视为特殊的公共子程序.

4. 操作系统结构

1) 分层法
2) 模块化
3) 宏内核
4) 微内核
5) 外核

5. 引导 (bootstrap)

在计算机刚刚启动, 操作系统还未开始运行之前, 需要开机后的第一个程序——引导加载器 (bootstrap loader) 来一步一步地初始化操作系统。对大多数操作系统来说, bootstrap 都会被存储在 ROM 中, 并且需要在一个已知的位置.

BIOS : 基本 I/O 系统

II 进程与线程

1. 进程与线程

1.1 进程

进程 (process) 是 OS 进行资源分配和调度的单位, 以特定形式存在于内存中, 具有一定的封闭性, 是多道技术的重要基础.

1.2 进程的状态与转换

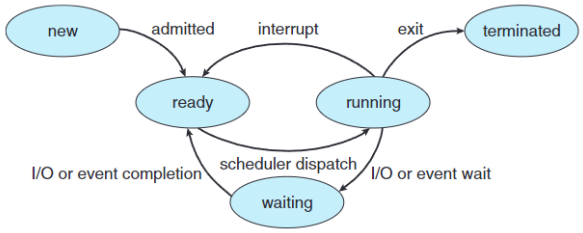

Figure II.1: Diagram of process state

1.3 进程的组成

内核部分: 进程控制块 (Process Control Block, PCB), 系统利用 PCB 来描述进程的基本情况和运行状态.

用户部分:

- Text section: 存储代码
- Data section: 存储代码中的全局变量、静态变量;
- Heap section: 常说的“堆”, 被动态分配的内存;
- Stack section: 常说的“栈”, 存储一些暂时性的数据, 如函数传参、返回值、局部变量等;

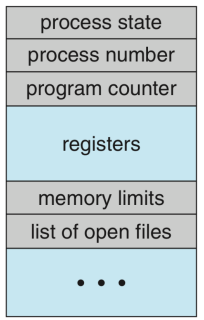

Figure II.2: PCB

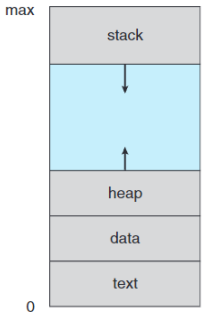

Figure II.3: Layout of a process in memory

1.4 进程控制

1) 进程的创建: fork() 创建进程, 该进程只有进程号与 parent 进程不一样, 同时通过检查返回值 pid 来判断属于 parent 还是 child.

使用 execXX() 覆盖进程的地址空间, 以实现执行其他程序

2) 进程的终止: exit() 终止进程, 同时返回状态值, 值被 parent 进程的 wait() 接收. 若 parent 没有 wait(), 进程变为僵尸进程, 被 systemd wait.

3) 进程的通信: 通过诸如共享存储 (shared memory), 消息传递 (message passing), 文件 / 管道 (pipe) 等的方法通信

1.5 进程切换

上下文切换 (context switch): 切换 CPU 到另一个进程需要保存当前进程状态并恢复另一个进程的状态.

上下文可能包括: CPU 寄存器中的值, 进程状态, 内存的管理信息等.

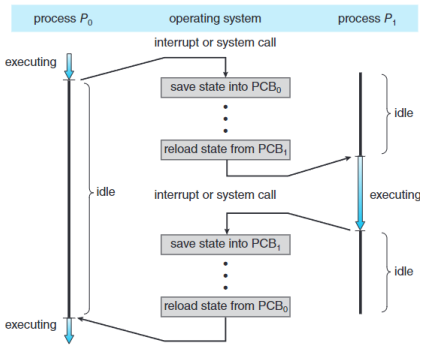

Figure II.4: 进程间的 context switch

1.6 线程和多线程模型

线程是一种轻量级的进程，它在进程的基础上进行划分，是进程内的一个可调度的执行单元，以减小进程 folk 和切换的开销为目的.

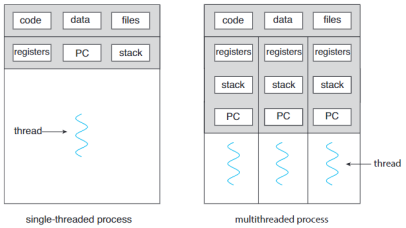

Figure II.5: Single-threaded and multithreaded processes

多线程模型，由于用户级线程和内核级线程连接方式的不同，分为三种.

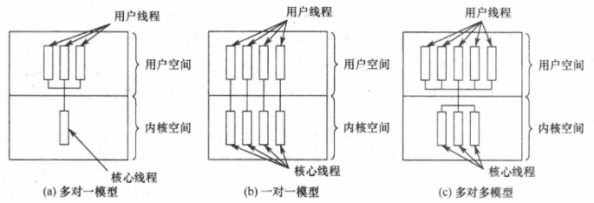

Figure II.6: 多线程模型

2. 进程调度

调度问题 (scheduling). 分为抢占式调度与非抢占式调度.

2.1 进程调度 Queues

- Job queue
  系统中所有进程
- Ready queue
  在 main memory 中的所有进程，准备好等待运行
- Device queues
  等待 I/O 设备的进程

2.2 调度衡量标准

- CPU utilization (CPU 利用率)
  应尽可能使 CPU 有效工作.
- Throughput (吞吐率)
  表示单位时间内 CPU 完成作业的数量.
- Turnaround time (周转时间)
  指从作业提交到作业完成所经历的时间.
- Waiting time (等待时间)
  指进程处于 **ready queue** 的时间之和 (不算 I/O 时的时间)
- Response time (响应时间)
  指从用户提交请求到系统首次产生响应所用的时间.

2.3 调度实现

由 CPU scheduler 选择哪一个就绪态的将要被执行后，由 dispatcher(调度器) 来完成具体的切换工作包括：

1) 在两个进程间进行上下文切换；
2) 切换到用户态；
3) 跳转到用户程序中合适的位置以继续进程执行

而从 dispatcher 停止上一个运行时的进程，完成上下文切换，并启动下一个进程的延时，称为 dispatch latency.

2.4 调度算法

FCFS 是最基本的非抢占式调度方法，就是按照进程先来后到的顺序进行调度.

SJF 选择需要运行时间最短的进程先运行. 能够保证理论上平均等待时间最小；但可能导致运行时间长的进程一直被推迟，从而导致“饥饿”现象. 此外，进程运行时间难以获知.

Priority Scheduling 选择优先级最高的进程先运行. 可以分为抢占式或非抢占式实现. 可使用 priority aging，让等待过久的任务赋予较高的优先级，以此避免饥饿的发生.

RR 每一个进程最多连续执行一个时间片的长度，完成后被插入到 FIFO ready queue 的末尾，并取出 FIFO ready queue 的队首进行执行，是使用分时技术后的 FCFS 调度. 若在时间片内进程完成运行可立即切换到下一个线程.

Multilevel Queue 将 ready queue 分成多个队列，每个队列使用不同的调度算法.

Multilevel Feedback Queue 在 Multilevel Queue 基础上，允许进程在队列间转移，以此实现更灵活更科学的调度

3. 同步与互斥

3.1 Background

共享数据上的并发访问导致的不同步

Race Condition(竞态条件) 多个进程对寄存器中的内容进行并发的访问会发生竞态条件.

3.2 The Critical-Section Problem

仅在 Critical section 中修改 register. 细粒度的 critical section 并发性更好.

```
do{
    Entry section;
    Critical section; // 临界区段
    Exit section;
    Remainder section;
}while(TRUE);
```

Code 1: General structure of a typical process P_j

Solution to Critical-Section Problem:

1) Mutual Exclusion (互斥)：对一个 critical section，仅有一个相关进程可以运行.

2) Progress (空闲让进)：当无进程处于临界区，可允许一个请求进入临界区的进程立即进入自己的临界区

3) Bounded Waiting (有限等待)：对请求进入的进程，保证能在有限时间内进入临界区.

3.3 Peterson's Solution

- 只有两个进程参与
- 两个进程共享两个变量:
  - `int turn` ：指示轮到谁进入临界区
  - `bool flag[2]` ：用于指示进程是否准备好进入临界区

```
while (true) {
    flag[i] = TRUE;
    turn = j;
    while ( flag[j] && turn == j);
    CRITICAL SECTION;
    flag[i] = FALSE;
    REMAINDER SECTION;
}
```

Code 2: The Algorithm for Process P_i

但遇到指令重排会失效，可在 `flag[i]=TRUE` 后增加个内存屏障，保证 flag 都处理完才能进行下面的指令.

3.4 Synchronization Hardware

在临界区关中断，出了后再打开.

或使用一些 atomic 指令.

```
bool lock=false;

bool TestAndSet (bool *target){
    bool rv = *target;
    *target = TRUE;
    return rv:
}

while (true) {
    while(TestAndSet(&lock))
        ; /* do nothing */
    critical section;
    lock = FALSE;
    remainder section;
}
```

Code 3: TestAndSet

```
bool lock=false;

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

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

Code 4: Swap

3.5 *Solution with Mutex Locks*

```
void acquire(){
    while(!available)
        ; /* busy wait */
    available=false;
}

void release(){
    available=true;
}
```

Code 5: acquire and release

上述是自旋锁 (spin lock), 为 mutex lock 的一种实现, 其会存在忙等待 (busy waiting).

3.6 *Semaphores(信号量)*

```
struct Semaphore{
    int val;
    Semaphore(int _val)val(_val){}
};

void wait(Semaphore S){
    while(S.val<=0);
    S.val--;
}

void signal(Semaphore S){
    S.val++;
}
```

Code 6: Semaphores with busy waiting

```
struct Semaphore{
    int val;
    struct process *L;
    Semaphore(int _val)val(_val){}
};

void wait(Semaphore S){
    S.val--;
    if(S.val<0){
        add this process to S.L;
        block(S.L);
    }
}

void signal(Semaphore S){
    S.val++;
    if(S.val<=0){
        remove a process P from S.L;
        wakeup(P);
    }
}
```

Code 7: Semaphores without busy waiting

signal 与 wait 都是 atomic 的.

```
Semaphore S(1); // initialized to 1
wait(S);
Critical Section;
signal(S);
```

Code 8: Semaphore Usage

3.7 *Monitors(管程)*

管程定义了一个数据结构和能为并发进程所执行 (在该数据结构上) 的一组操作, 这组操作能同步进程和改变管程中的数据. 管程中一次只能有一个进程处于活动状态.

条件变量 (Condition Variables) 将进程阻塞原因定义为条件变量. 对条件变量只能进行两种操作,

- x.wait：当 x 对应的条件不满足时, 正在调用管程的进程调用 x.wait 将自己插入 x 条件的等待队列, 并释放管程. 此时其他进程可以使用该管程.
- x.signal：x 对应的条件发生了变化, 则调用 x.signal, 唤醒一个因 x 条件而阻塞的进程.

3.8 同步问题例子

缓存有界 缓存区存在上界, 使用 empty 与 full 两个信号量控制缓存区, 缓存区是否需要再一个信号保护依据题目决定.

例如, 缓存区至多缓存 N 个物品, A 生成物品放入缓存区, B 从缓存区中取出消耗物品

```
Semaphore empty=N, full=0, mutex=1;
```

```
A{
    P(empty);
    P(mutex);
    product;
    V(mutex);
    V(full);
}
```

```
B{
    P(full);
    P(mutex);
    consume;
    V(mutex);
    V(empty);
}
```

同类并发 设定 count 计数, 并在为 0 时作特殊处理. 记得 count 需要保护.

例如, 有 A, B 两种进程需要一项资源 s, 但多个 A 或多个 B 可以同时使用.

```
Semaphore s=1, sa=1, sb=1;
int counta=0, countb=0;
```

```
A{
    P(sa);
    if(counta==0)P(s);
    counta++;
    V(sa);
    run A;
    P(sa);
    counta--;
    if(counta==0)V(s);
    V(sa);
}
```

```
B{
    P(sb);
    if(countb==0)P(s);
    countb++;
    V(sb);
    run A;
    P(sb);
    countb--;
    if(countb==0)V(s);
    V(sb);
}
```

依赖关系 为所依赖的资源设定信号量.

例如, $T_1 : A, C$, $T_2 : B$, 运行顺序为 $A \to B \to C$.

```
Semaphore A=0,B=0;
```

```
T1{
    run A;
    V(A);
    P(B);
    run C;
}
```

```
T2{
    P(A);
    run B;
    V(B);
}
```

4. Deadlocks

4.1 概念

Deadlocks 指多个进程因竞争资源而造成的互相等待. 但同时满足以下四个条件时, 死锁产生.

1) Mutual exclusion(互斥性): 一个 resource 只能同时被一个 process 使用.
2) Hold and wait: 进程手里至少有一个 resource, 且等待其他进程的 resource.
3) No preemption: 不能抢占
4) Circular wait: 循环等待.

4.2 预防

防止死锁的发生只需破坏死锁产生的 4 个必要条件之一即可.

1) Mutual Exclusion: 难以打破.
2) Hold and Wait: 让一个线程/进程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
3) No Preemption: 也不太行
4) Circular Wait: 通过给资源编号, 规定进程/线程只能按特定的顺序申请资源 (但难以做到)

4.3 避免

资源分配图 是一种有两类节点的有向图, 我们用圆节点 T_i 表示进程/线程, 用方节点 R_j 表示资源, 方节点中的实心点表示一个资源类别的一个实例.

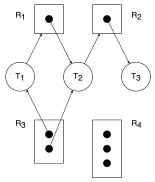

Figure II.7: 资源分配图

银行家算法 就是需要动态地检测某个资源申请是否会导致系统进入不安全状态, 如果会导致系统进入不安全状态, 则等待资源足够再分配.

数据结构:

- $Available$(可利用资源向量) M
- Max(最大需求矩阵) $N \times M$
- $Allocation$(分配矩阵) $N \times M$
- $Need$(需求矩阵) $N \times M$

$$Need = Max - Allocation$$

Algorithm 1 银行家算法

Require: $Request_i$ M, 表示进程 P_i 请求资源的数量
if $Request_i \leq Need_i$ and $Request_i \leq Available_i$ **then**
  系统尝试分配资源:

$$Available- = Request$$
$$Allocation_i+ = Request$$
$$Need_i- = Request$$

  运行安全性算法检查分配后是否处于安全状态
  if 安全 **then**
    完成分配
  else
    恢复分配, 并让 P_i 继续等待
  end if
else
  无法或无需分配
end if

Algorithm 2 安全性算法

初始化安全序列, $Work = Available$, 表示当前状态的剩余资源量
while 寻找 i, 有 i 不在安全序列并且 $Need_i \leq Work$ **do**
  把 i 加入安全序列
  $Work+ = Allocation_i$
end while
若安全序列有所有进程, 则系统安全, 否则不安全.

4.4 检测

面向单实例资源检测 反正就是资源状态图

4.5 解除

1) 都别活, 杀死所有死锁中的进程/线程
2) 一个一个杀, 杀到没有死锁
3) 留活命, 但是需要回滚部分进程, 强行抢占占有的资源

III 内存管理

1. 内存基本管理

1.1 基本原理和要求

内存管理的主要功能有:

1) 内存空间的分配与回收
2) 地址转换
3) 内存空间的扩充
4) 内存共享
5) 存储保护

程序的链接与装入 需要编译, 连接, 装入.

动态装载 (dynamic loading): 一个例程以可重定位装载格式 (relocatable load format) 存储在磁盘上, 被调用时, 就动态地被装载到内存中.

逻辑地址与物理地址 区分物理地址 (physical address) 和虚拟地址 (virtual address), 后者也叫逻辑地址 (logical address).

内存管理单元 (memory management unit, MMU) 实现从虚拟地址到物理地址的映射的硬件. TBL 也属于 MMU 的一部分.

内存保护 base 和 limit 两个上下限寄存器来实现框定进程的内存空间, 始于 base 寄存器中存储的地址, 终于 base + limit 对应的地址. 内存保护也是通过 MMU 实现的.

1.2 Contiguous Memory Allocation(连续内存分配)

固定划分 (fixed partition) 内存空间划分为若干固定大小的区域, 每个分区只分配一个进程. 可能导致内部碎片 (internal fragmentation), 即分配量大于进程实际需求量.

可变划分 (variable partition) 只要是空闲且足够大的连续内存区域都可以被分配. 长时间运行后导致外部碎片 (external fragmentation), 即存在大量较小的, 难以利用的 holes.

可以使用 First Fit, Best Fit, Worst Fit 等分配策略以减少外部碎片.

1.3 Paging

进程中的块称为页或页面 (Page), 内存中的块称为页框或页帧 (Page Frame). 外存也以同样的单位进行划分, 直接称为块或盘块 (Block). 进程在执行申请内存时, 即要为每个 Page 分配内存中可用 Frame, 这就产生了 Page 和 Frame 的一一对应。

地址结构 前一部分为页号 (page number)p, 后一部分为页内偏移量 (page offset)d. 对于 page size 为 4KB 的页, page offset 需要有 $\log_2 4096 = 12$ 位。

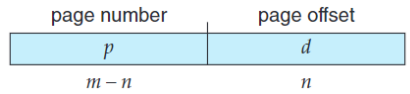

Figure III.1: address

页表 (page table) 存储逻辑的页到物理的帧的映射关系. 从虚拟地址到物理地址的映射, 实际上就是在页表中查询虚拟地址中的 page number, 将其换为 frame number, 再直接拼接 offset 就行了.

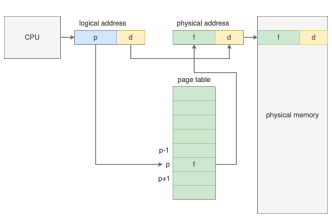

Figure III.2: 地址转化

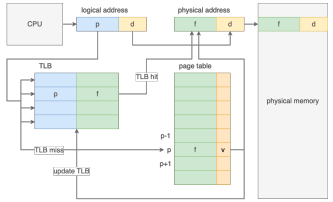

Figure III.3: 地址转换 with tlb

硬件支持 页表应当作为一个进程的元信息被维护. 被放在内存中, 页表基址寄存器 (page-table base register, PTBR) 维护一个指向页表的指针来. 每个进程有一个 PTBR.

页表缓存 (translation look-aside buffer, TLB): 加速页表的维护, 页号和帧号以键值对的形式存储在 TLB 中.

Associative Lookup $= \epsilon$ time unit. Assume memory cycle time is t ms. Hit ratio $= \alpha$

有效内存访问时间 (effective access time, EAT):

$$\begin{aligned} EAT &= (t+\epsilon)\alpha + (2t+\epsilon)(1-\alpha) \\ &= (2-\alpha)t + \epsilon \end{aligned}$$

共享页 (shared page) 多个页可以对应同一个帧. 提高代码重用率.

1.4 页表设计改进

分层分页 (hierarchical paging) 典型的就是二级页表 (two-level page table). linux 最高可到七级.

哈希页表 (hashed page table) 维护了一张哈希表，以页号的哈希为索引，维护了一个链表，每一个链表项包含页号、帧号、和链表 next 指针，以此来实现页号到帧号的映射.

反式页表 (inverted page table) 以物理地址为索引维护映射关系. 使用特殊硬件 content-addressible memory (CAM), 通过数据查找相应地址, 加速.

1.5 Swapping

进程可以临时从存储器交换到后备存储器，然后被带回存储器以继续执行。

2. 虚拟内存管理

仅仅一部分在 memory 的程序需要运行. 所以逻辑内存可以大于物理内存. 甚至可以把虚地址扩展到和外存一样大.

2.1 Demand Paging(请求式调页)

只把被需要的页载入内存

Valid-Invalid Bit 每个 page table entry (PTE) 会有一个 bit 标识所指页是否 valid.

Page Fault 当所要访问的页面不在内存中时，便产生一个 Page Fault.

1) 对地址, 检查一张 PCB 里的内部表:
   若无效引用 $\rightarrow$ abort
   若只是并没有被分配, 继续
2) 从可用帧列表里拿出 frame 用来写入
   如果可用帧列表为空，则进行页置换
3) 开始从后备存储读取内容，并写入 frame
4) 完成读写后，更新内部表和页表等元信息
5) 重新执行引起 page fault 的 instruction

Performance of Demand Paging Page Fault Rate: $0 \le p \le 1$

$$\begin{aligned} EAT =& (1-p) \times \text{memory access} \\ &+ p \begin{pmatrix} & \text{page fault overhead} \\ + & \text{swap page out} \\ + & \text{swap page in} \\ + & \text{restart overhead} \end{pmatrix} \end{aligned}$$

2.2 Copy-on-Write

fork 后, 只有当子进程需要发生内容修改的时候，才真正去复制父进程的内容

2.3 Allocation of Frames

维护可用帧列表 (free-frame list)，在 demand paging 系统里记录当前哪些帧是空闲的。

对于单个 process 分配的 frames 数量，存在一个较严格的分配上下界

1) 不能大于 free-frame 总量
2) 不能小于 process "执行每一条指令所需要涉及的 frames" 的最大值

早期分配算法 (frame-allocation algorithm) 主要有这么两种:

1) Equal allocation: 每一个进程被分配的 frame 总量都相同
2) proportional allocation: 按进程的大小来分配

2.4 Page Replacement

用一个修改位 (dirty bit 或 modified bit) 来记录页是否被修改过.

被替换的帧称为牺牲帧 (victim frame)

Optimal 选之后再也不会被用到的或下一次用到的时间最晚的页作为 victim frame. 理论上最优.

FIFO 选择正在使用中的、最早进入内存的 frame 作为 victim frame

LRU 选择最久没被访问过的 frame 作为 victim. 但维护 "最久没被访问过" 这个信息并不容易, 可以使用计数器或链表序列维护.

LRU Approx 使用一些方式近似 LRU

1) Reference-Bits: 在初始化时被置 0；被使用时置 1. 替换 0 的 frame.
2) Second-Chance Algorithm: 循环地遍历 frames, 检测 reference bit, 若 1 则置 0, 若 0 替换.
3) Enhanced Second-Chance Algorithm / NRU: 二元组维护 (dirty, reference), 循环地遍历四次 frame, 依次寻找 (0,0), (0,1), (1,0), (1,1) 的替换.

置换范围 replacement 分为 local 和 global

- local replacement 只发生在属于当前进程的帧中
- global replacement 的 scope 是所有帧

2.5 Thrashing(颠簸)

频繁的 paging 活动, 几乎所有 frames 都正在被使用.

Priority 用 priority replacement algorithm 来解决

Working Set 指在某段时间间隔内，进程要访问的页面集合。工作集 W 可由时间 t 和工作集窗口大小 Δ 来确定。工作集反映了进程在接下来的一段时间内很有可能会频繁访问的页面集合, 若 working set 的大小之和大于可用 frame 的数量, 就会出现 thrashing.

2.6 内存映射文件 (Memory-Mapped Files)

将磁盘文件的全部或部分内容与进程虚拟地址空间的某个区域建立映射关系，便可以直接访问被映射的文件，而不必执行文件 I/O 操作，也无须对文件内容进行缓存处理。

2.7 Allocating Kernel Memory

Often allocated from a free-memory pool

Buddy 系统 用来分配物理连续的内存，它由 power-of-2 allocator 实现

当 kernel 需要 n KB 的内存时候，Buddy system 会分配一块 $2^{\lceil \log_2 n \rceil}$ KB 的空间.

其也可以通过 coalesce 相邻的空闲块来形成更大的内存块.

Slab 分配 预先了解到 kernel 内的常见数据结构 (被称为各种 object) 的大小，并预先准备好对应粒度的小内存块，注册到每类 object 的 cache 里. 当一个 object 需要使用内存时，就查询对应的 cache 里是否有空闲的内存块，如果有就分配给它，如果没有就向 Buddy system 申请.

IV 文件系统

1. File-System 基础

1.1 File Concept

一个文件是存储在二级介质上的，具名的一系列相关信息集合，无论是用户还是程序，都需要通过文件来与二级介质进行信息交换。

File Attributes 不同的文件系统下，文件可能有不同的属性，但通常有以下几个:

- name：文件名；
- identifier：用于唯一标识一个文件；
- type：用于标识类型；
- location
- size：文件大小，也可能包含文件被允许的最大大小；
- protection：访问控制信息，决定哪些用户具有对应的读/写/执行权限等；
- timestamp：保存创建时间/上次修改时间/上次使用时间等，这些信息可用来做一些安全保护和使用监控；
- user indentification：创建者/上次修改者/上次访问者等，这些信息可用来做一些安全保护和使用监控；

这些信息也被称为文件的元数据 (meta data)。

File Operations 一些基本的文件操作:

- create：分为两步，一是在文件系统中为文件分配一块空间，二是在目录中创建对应的条目；
- open / close：打开文件后会得到文件的句柄 (handle)，其它对特定文件的操作一般都需要通过这个句柄来完成

  通常来说，文件被打开后需要由用户来负责关闭；

  打开后的文件会被加入到一个打开文件表 (open-file table) 中，这个表中保存了所有打开的文件的信息，包括文件的句柄、文件的位置、文件的访问权限等；

  文件可能被多方用户 (进/线程) 打开，而只有所有用户都关闭文件后才应当释放文件在打开文件表中的条目，所以维护一个 open-file count 用于记录当前文件被打开的次数；
- read / write：维护一个 current-file-position pointer 表示当前操作的位置，在对应位置上做读写操作；
- repositioning within a file：将 current-file-position pointer 的位置重新定位到给定值，也被叫做 seek；
- delete：在 directory 中找到对应条目并删除该条目，如果此时对应的文件没有其它硬链接，则需要释放其空间；
- truncate：清空文件内容，但保留文件属性；
- locking；

Open File Locking 不同的文件操作对应着不同的权限, 通过访问控制列表 (access control list, ACL) 来维护用户们对文件所具有的权限. 精简化后为访问权限位 (access permission bits) 的方式来实现权限控制.

比如在 linux 中有 10 个字符控制权限. 第 1 个字符表示一些类型, 比如 "-" 表示原始文件, "d" 表示一个目录. 后 9 个字符将权限被分为三组，分别代表文件所有者 (owner)、文件所属组 (group)、其他人 (other) 的读 (r)、写 (w)、执行 (x) 权限。

File Types 主要分为数据 (data) 和程序 (program) 两大类. UNIX 系统会在文件开头，使用一串 magic number 来标识文件的类型.

File Struction 文件结构指的是文件数据存储的形式, 常见的有:

- 无结构：流式的存储所有的 words/bytes
- 简单记录结构 (simple record structure)：将文件以 record 为单位存储.
- 复杂结构 (complex structures)；

1.2 Access Methods

- 顺序访问 (sequential access)
- 直接访问 (direct access)/相对访问 (relative access)/随机访问 (random access)
- 索引顺序访问 (indexed sequential-access): 先通过索引表查询位置，然后去访问

2. Directory

目录本质上是一个特殊的文件 (Linux 中). 目录的结构表示的是目录下文件的组织方式.

- Single-Level Directory: 所有的文件都被铺在根目录下
- Two-Level Directory: 主文件目录 (master file directory, MFD) 下为每个用户分配一个用户文件目录 (user file directory, UFD)，每个用户的目录下再存放该用户的文件
- 树形目录 (tree-structured directories): 将目录视为一种特殊文件, 允许用户在目录下自由地创建目录进行分组, 总体文件结构成为一种树形结构

  文件的路径 (path)，分为绝对路径 (absolute path) 和相对路径 (relative path) 两种
- 无环图目录 (acyclic-graph directories): 在树形目录的基础上，允许目录之间存在链接关系，链接分为软链接 (soft link) 和硬链接 (hard link) 两种

  软链接又称符号链接 (symbolic link)，是一个指向文件的指针

  硬链接是复制链接文件目录项的所有元信息，存到目标目录中，此时文件平等地属于两个目录
- 通用图目录 (general-graph directories): 允许目录之间存在环，在各种操作时，通过算法来避免出现问题.

3. File System

文件系统 (file system, FS) 是操作系统中，以文件的方式管理计算机软件资源的软件，以及被管理的文件和数据结构的集合.

3.1 挂载 (mount)

是指将一个文件系统的根目录挂载到另一个文件系统的某个目录 (被称为 mount point). 只有被挂载了，一个文件系统才能被访问.

3.2 文件系统分层设计

文件系统被分为若干层，向下与 device 交互，向上接受 application programs 的请求.

1) I/O control

   向下控制 I/O devices，向上提供 I/O 功能；
2) Basic file system

   向下一层发射 "抽象" 的, 由下一层转化为设备直接支持的指令的, 操作指令；

   与 I/O 调度有关；

   管理内存缓冲区 (memory buffer) 和缓存 (caches)；
3) File-organization module

   以 basic file system 提供的功能为基础；

   能够实现 file 的 logical block 到 physical address 的映射；4

   同时，file-organization module 也囊括了 free-space manager；
4) Logical file system

   存储一些文件系统的结构信息，不包括实际的文件内容信息；

   具体来说，logical file system 会维护 directory 的信息，为之后的 file-organization module 提供一些信息，例如符号文件名；

   FCB 会维护被打开的文件的一些具体信息；

4. File System Implementation

4.1 硬盘数据结构

On-Disk 的数据结构维护 1. 如何启动硬盘中的 OS，2. 硬盘中包括的 block 总数，3. 空闲 block 的数量和位置，4. 目录结构以及文件个体等，下面介绍几个主要的数据结构:

- 操作系统被保存在引导控制块 (boot control block) 中，一般 boot control block 是操作系统所在的 volume 的第一个 block。
- 卷控制块 (volume control block) 维护了 volume 的具体信息，例如 volume 的 blocks 数量、空闲 block 的数量与指针、空闲 PCB 的数量与指针等
- 目录结构 (directory structure) 用来组织 files，同时也维护了 files 的元信息
- 文件控制块 (file control block, FCB) 维护了被打开的文件的具体信息。

4.2 内存数据结构

在 main memory 中维护，用于帮助文件系统管理和一些缓存操作

- 已被挂载的 volume 会被记录在 mount table 中
- Directory cache: 为了提高文件系统的性能
- System-wide open-file table: 记录这个系统中所有进程打开的文件
- Per-process open-file table: 记录每个进程打开的文件
- Buffers: 在内存中，用于缓冲 disk block 的内容

4.3 目录的实现

linear list based 线性检索法通过线性表来存储目录信息，每个目录项包含 file name 和指向 FCB/Inode 的指针，查找时需要遍历查找。

hash table based 哈希表法通过哈希表来存储目录信息，每个目录项包含 file name 和指向 FCB/Inode 的指针，可以直接通过 hash function 进行 random access。

4.4 块分配与块组织

连续分配 (contiguous) 指的是每个文件占用一段连续的 block

$$\text{Logic Address} = \text{block size} * Q + R$$

- Block to be accessed = Q+ start address
- Displacement into block = R

链接分配 (linked) 每个 block 作为一个链节，维护存储信息以外还需要维护指向下一个 block 的指针。此时，FCB 中只需要记录起始地址即可。

$$\text{Logic Address} = (\text{block size} - 1)Q + R$$

(block size − 1) because of pointer

- Block to be accessed is the Qth block in the linked chain of blocks representing the file
- Displacement into block = $R + 1$

索引分配 (indexed) 索引方法将所有指针统一维护到 index block 中。每个文件有自己的 index block，顺序存放着指向文件的所有 block 的指针。

$$\text{Logic Address} = \text{block size}Q + R$$

- Q = displacement into index table
- R = displacement into block

计算时要考虑索引系统所能支持的最大块数量与块地址能支持的最大地址空间大小，取最小的那个作为答案.

4.5 空闲空间管理

- 位图 (bitmap): 用对应 bit 的 0 或 1 来标记对应的 block 是否空闲
- 链表: 用链表将空闲的 block 连起来。
- 分组: 是基于链表方法的改进, 将 n 个空闲块的地址存放在第 0 个空闲块中, 以此类推.
- 计数方法: 维护每个连续内存段的起始地址和额外长度。

4.6 典型文件系统

ext, FAT, NTFS

4.7 虚拟文件系统

虚拟文件系统 (virtual FS, VFS) 有两个主要功能：

1) 封装并透明化具体文件操作，同一接口可能有不同的具体实现，通过这种方式来支持不同的文件系统；
2) 反过来为文件系统提供一个唯一标识文件的方式，VFS 基于一个名为 vnode 的 file-representation structure 的东西来表示文件，其中包含了在整个文件网络中标识文件的数字指示符；

5. 文件系统的性能与安全

- 日志系统: 日志结构的文件系统 (log-structured FS) 是一种文件系统的实现方式，它将所有的修改操作都作为一个事务 (transaction) 记录在一个日志中，而不是直接修改文件本身。
- 恢复: 备份数据，并时常检查数据的一致性和完整性，如果发现问题则尝试利用备份数据进行恢复。

V I/O 管理

1. I/O 管理概述

1.1 概念

- 总线 (bus)：连接各个硬件组成的“抽象”内部通信线路，更侧重于如何规范化地传输数据，是硬件与协议的统一；
- 端口 (port)：设备与总线的连接点；
- 控制器 (controller)：控制设备的硬件组成；
  集成在设备上或单独在电路板上；
  通常包括处理器、私有内存、微代码、总线控制器等；

1.2 I/O 访问方式

轮询 (polling) 指的是 CPU 不断向设备控制器查询设备状态，直到设备就绪，然后进行数据传输。

中断 (interrupt) 计算机向设备发出请求以后，将当前进程调度走，等到设备处理完成后会向 CPU 发送中断，此时计算机再对结果做处理。

DMA(direct memory access) 允许内存和 I/O 设备之间直接交互，不经过 CPU，这样可以减少 CPU 的负担，提高 I/O 性能。

2. 应用程序 I/O 接口

不同设备可能在这些方面有区别：

- 数据传输模式 (data transfer mode)：逐个字节传输或以块为单位传输
- 访问方法 (access method)：需要顺序访问或可以随机访问
- 传输方法 (transfer method)：
  同步的，需要按预计的响应时间进行传输并和系统的其他方面相协调
  阻塞式：一直等待直到 I/O 完成；
  非阻塞式：返回尽可能多的数据，不管是否完成；
  异步的，响应时间不需要规则或者可预测，不需要与其他计算机事件协调
- 共享 (sharing)：
  可共享：可以被多个进程或线程并发使用
  独占的：不能被共享
- 设备速度 (device speed)
- I/O 方向 (I/O direction)：R-，-W，RW

3. 存储

3.1 硬盘

硬盘 (hard disk, HD) 是常见的二级存储，其结构按照从小到大分为：扇区 (sectors)、磁道 (tracks)、柱面 (cylinders)，侧面的磁臂 (disk arm) 会以整体移动上面的所有读写磁头 (r/w heads).

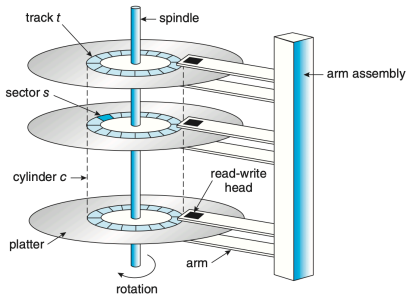

Figure V.1: Moving-head Disk Machanism

从硬盘上读写内容的过程如下:

1) 磁头移动到指定的柱面；
2) 磁头移动到指定的磁道；
3) 磁盘旋转到扇区位于磁头下方；
4) 读写扇区内容；

disk 的平均 I/O 操作时间为:

$$\text{Average I/O time} = \underbrace{\text{average seek time} + \text{rotational latency}}_{\text{average access time}} + \underbrace{\frac{\text{data to transfer}}{\text{transfer rate}}}_{\text{transfer time}} + \text{controller overhead}$$

- 寻道时间 (seek time)：磁头移动到指定柱面的时间
- 旋转时延 (rotational latency)：目标扇区旋转到磁头下方的时间
  一般以 round per minute(rpm) 表示，容易得到，平均旋转时延为 $\frac{1}{2}\frac{1}{rpm}60$s
- 传输时间 (transfer time)：数据在 disk 和 memory 之间传输的时间, 一般就是磁头扫过数据区域的时间.

调度 加速 access time.

disk bandwidth = 传输数据量 / 请求开始到传输完成的时间间隔

logical block address(LBA), logical block 是数据传输的最小单元

常见算法:

- FCFS
- SSTF (shortest seek time first) 选择距离当前磁头最近的请求
- SCAN 算法下磁头在碰到 LBA 边界前只会单向移动，而在移动过程中处理能够处理的请求。
- LOOK 就是不走到底，而是走到最靠近边界的请求对应的 LBA 就提前掉头的 SCAN.
- C-SCAN 即 Circular SCAN，其磁头移动是始终单向的，当磁头达到 LBA 的边界时，径直返回到另一端，回程中不响应任何请求.
- C-LOOK 是在处理完最靠近边界的请求后就直接返回的 C-SCAN.

根据不同的应用场景选择不同的算法, 通常，SSTF 是比较常见的默认选择；而当 I/O 较为频繁的时候，一般使用 LOOK 或者 C-LOOK.

3.2 存储介质管理

- 分区 (partitioning)：将设备的存储空间做划分，每一个都被视为一个单独的存储空间 (即一个 logical disk)。分区信息会以固定的格式被写入存储设备的固定位置。
- 卷创建与卷管理 (volume creating & management)：卷 (volume) 是包含一个文件系统 (file system) 的存储空间，划定文件系统所覆盖的范围。如果直接在一个分区里安装文件系统，那其实这一步已经被隐式地完成；但如果使用比如 RAID 技术，就需要显示地做这一步。
- 逻辑格式化 (logical formatting)：在卷上创建和初始化文件系统。

同时，如果当前分区包含 OS 镜像，则需要对应初始化引导块 (boot sector)。

RAID 0：无冗余 1：镜像 2：纠错码 3：奇偶校验 1 个盘 4：按块条带化 Striping 5：校验盘分散到各个盘 6：P+Q 冗余，差错纠正码