Part 1 操作系统概述

1.1 Operating System

宏内核 Monolithic Kernel
单任务 uniprogramming
批处理 batch system
多任务 multiprogramming
分时系统 time-sharing
调用函数 invoke a function

1.1.1 历史

●串行处理 Serial
●批处理：吞吐量优先
○监控程序 Monitor
●多任务处理：并发度优先
●分时系统：周转时间优先、响应时间优先

1.1.3 System Call vs API

System calls

- Request OS services
  - Process control: abort, create, terminate process allocate/free memory
  - File management: create, delete, open, close file
  - Device management: read, write, reposition device
  - Information maintenance: get time or date
  - Communications: send/receive message

1.1.4 实时系统

● 硬实时：保证准时完成。
● 软实时：重要任务优先级高而已，未必能准时。

1.2 计算机硬件

特权指~ privileged instruction
轮询 poll
●指令是通过数据总线传输的

1.2.1 中断

●硬中断 Interrupt
由硬件发起、异步的
●软中断 Exception
由软件发起、同步的
○陷阱 Trap: System Call
○错误 Fault
○中止 Abort

1.2.3 硬件级保护

Privileged instructions

- Examples
  - Change mode bit in processor status register
  - Change which memory locations a user program can access
  - Send commands to I/O devices
  - Jump into kernel code

Non-privileged ("safe") instructions

- Examples
  - Load, store
  - Add, subtract, ...
  - Conditional branch, jump to subroutine, ...

Even better approaches

- Problems with the additive base and bounds?
  - Memory sharing between processes? (e.g., how to share one copy of same machine instructions)
  - Memory fragmentation as processes come and go
- Paging and segmentation
  - Translation done in hardware, using a table for each process
  - Table is set up and managed by kernel

●所有 I/O 操作都是特权指令
(3) 内存保护
●Base register & limit register

Memory protection

- Protect
  - Interrupt vector and the interrupt service routines
  - Data access and over-write from other programs
- HW support: two registers for legal address determination
  - Base register – holds the smallest legal physical memory address
  - Limit register – contains the size of the range
- Memory outside the defined range is protected

OS Service	Hardware Support
Protection	Kernel/User Mode Protected Instructions Base & Limit Registers
Interrupts	Interrupt Vectors
System Calls	Trap Instructions
I/O	Interrupts, Memory-Mapping
Synchronization	Atomic Instructions
Virtual Memory	Translation Lookaside Buffers
Scheduling	Timer

Part 2 进程管理

2.1 进程

抢占 preemption
级联的 cascading
约束 constraint 凭借 resort

(1) 用户内存中的进程
栈 stack:从上往下坨，局部变量
堆 heap:从下往上堆，malloc()
data：全局变量 *text*：代码区
程序、数据、栈和属性（即 PCB）的集合被称为**进程映像**。每个进程映像都由进程控制块、用户栈、进程专用地址空间以及和其它进程的共享地址空间组成。

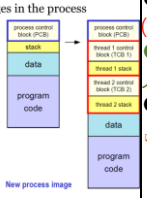
(2) Process Control Block
● Process identifier
● CPU state information
> user-visible, control & status registers
> stack pointers 用户可改
● Process control infomation
·scheduling:状态,优~级,事件
·used memory and I/O, opened files
·pointer to next PCB

2.2 线程与进程间通信

2.2.1 线程

> 独立拥有：Program Counter、寄存器组、栈空间
> 与兄弟线程共享：代码区、数据区、操作系统资源
（1）TCB：见 PCB 红色字

- Multithreading requires changes in the process description model
  - each thread of execution receives its own control block and stack
    - own execution state ("Running", "Blocked", etc.)
    - own copy of CPU registers
    - own execution history (stack)
  - the process keeps a global control block listing resources currently used



（2）U-to-K 模型

2.2.2 Inter-Process Communi~

(1) 分类
●共享内存 shared memory
●消息传递 MsgPass (SysCall)
○ Socket
○ Remote Procedure Call
(2) 信息传递
●阻塞式发送：发送方阻塞到消息被接收方接收完毕。
●非阻塞式发送：发送方后台发送信息,前台继续干别的事情。
●阻塞式接收：等~
●非阻塞式接收：收到之后再调个回调。
(3) RPC
客户端 stub；服务端 skeleton

2.3 进程调度

中程调度
medium-term scheduling
周转时间 Turnaround Time
服务时间 Service Time
总等待时间 Waiting Time
首次应答时间 Response Time
吞吐量 Throughput ←RAID

2.3.1 长中短调度

长：从磁盘进入内存（新建→就绪）
degree of multiprogrami~、合理搭配计算-I/O 密集进程
中：是否挂起 suspend（内存移到磁盘），swap out：滚出内存。
短：下一个给 CPU 的是谁？
决策者 scheduler
执行者 dispatcher：上下文切换、跳转到程序的对应位置

- Dispatch latency – time it takes for the dispatcher to stop one process and start another running:
  - Scheduling time
  - Interrupt re-enabling time
  - Context switch time

Scheduling criteria

- CPU utilization: keep the CPU as busy as possible
- Throughput: # of processes that complete their execution per time unit
- Turnaround time: amount of time to execute a particular process
- Waiting time: amount of time a process has been waiting in the ready queue
- Response time: amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

2.3.2 短程调度的时机

非抢占：阻塞；运行完毕。**抢占**：+时间片用尽；所需资源到达。

2.3.3 短程调度算法

(1) FCFS / FIFO 非抢占
(2) Round Robin 抢占(时间片)
●时间片结束了才会再找下一个，时间片中间来一个进程鸟都不鸟一下。

- Performance
  - q large ⇒ FCFS
  - q small ⇒ Interleaved
  - q must be large with respect to context switch, otherwise overhead is too high (all overhead)
- How do you choose time slice?
  - What if too big? Response time suffers
  - What if infinite? Get back FIFO
  - What if time slice too small? Throughput suffers!

Comparisons between FCFS and RR

- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Simple example: 10 jobs, each takes 100s of CPU time; RR scheduler quantum of 1s; All jobs start at the same time
- Completion Times:

Job #	FIFO	RR
1	100	991
2	200	992
...	...	...
9	900	999
10	1000	1000

  - Both RR and FCFS finish at the same time
  - Average turnaround time is much worse under RR!
  - Bad when all jobs have same length

(3) SPN / SJF 非抢占
●选定了就让它运行完!
●Shortest Path Next
●Shortest Job First
(4) SRTF 抢占(到达)
●一有新进程到，立刻重新选人。老渣男了！
●Shortest Remaining Time F~

- SPN/SRTF are the best you can do at minimizing average turnaround time
  - Provably optimal (SPN among non-preemptive, SRTF among preemptive)
  - Since SRTF is always at least as good as SPN, focus on SRTF
- Comparison of SRTF with FCFS and RR
  - What if all jobs the same length?
    - SRTF becomes the same as FCFS (i.e., FCFS is best can do if all jobs the same length)
  - What if jobs have varying length?
    - SRTF (and RR): short jobs not stuck behind long ones
- Starvation
  - SRTF can lead to starvation if many small jobs!
  - Large jobs never get to run
- Somehow need to predict future
  - How can we do this?
  - Some systems ask the user
    - When you submit a job, have to say how long it will take
    - To stop cheating, system kills job if takes too long
  - But: even non-malicious users have trouble predicting runtime of their jobs
- Bottom line, can't really know how long job will take
  - However, can use SRTF as a yardstick for measuring other policies
  - Optimal, so can't do any better
- SRTF Pros & Cons
  - Optimal (average response time) (+)
  - Hard to predict future (-) and Unfair (-)

(5) Priority 抢占或非抢占
● 低优先级进程饥饿 starve
● 优先级反转 priority inversion:
H 被 M 抢跑了。H/L 共享资源 R、M 不需要 R。L 先把 R 锁了。

- SPN/SJF is an example of priority scheduling
  - In SPN, the shortest job has the highest priority
- Can also assign processes fixed priorities
- Process priority is usually represented as a number
  - Varies whether higher or lower numbers correspond to high priority
- Priority scheduling can be preemptive or non-preemptive
  - If non-preemptive, a new higher-priority process added to ready queue won't take the CPU from a lower-priority running process
  - If preemptive, a new higher-priority process added to ready queue immediately takes the CPU from a lower-priority running process

(6) Multilevel Queue Sche~
(7) M~ Feedback Queue S~
●反馈即：可在队列间移动
●队列间 RR，队列内 FIFO。
●优先级越高的队列，时间片就越短。
●一旦某个进程被系统抢占一次，那么它就会被移到低一个优先级的队列的队尾。
●新进程会直接进入最高优先级队列的队尾。
(8) HRRN 非抢占
●最短相应比 w+s / s

2.4 互斥与同步

同步 Synchronization
互斥 Mutual Exclusion
条件同步 Condition Synchro~
临界区 critical section
竞争条件 race condition
有限等待 bounded waiting
二值信号量 binary semaphore
多值信号量 counting ~
有限并发 bounded concurrency
管程 monitor 原语 primitive
原子操作 atomic operation

2.4.1 同步的概念

● 互斥：数据一致性
● 条件同步：执行顺序

2.4.2 正确互斥四条件

忙则等待、空闲让进、有限等待、让权等待。

2.4.3.1 软件实现互斥

Test then set：无法保证互斥
Set then test：互斥是能互斥了，倒是经常死锁/活锁了。
Dekker：有阶级的 Set then test；两个预定锁，互相谦让，客场让主场，主客场轮流当。

Analysis of Peterson's algorithm

```
boolean P1WantIn = false;  //must be in shared memory
boolean P2WantIn = false;  //must be in shared memory
int turn = 1;  //must be in shared memory
P1:
while (true) {
    P1WantIn = true;
    turn = 2;
    while (P2WantIn && turn == 2) do {nothing};
    C.S.1;
    P1WantIn = false;
    remainder section;
}
P2:
.....
```

? If the order is reversed and a process is preempted after setting the turn variable but before setting the WantIn variable, then both processes can get into the critical section!

2.4.4 信号量

Semaphore structure

- A new data structure called *semaphore*
  - An integer with a nonnegative initial value : s.count
  - An initially empty queue: s.queue

```
struct semaphore {
    int count;
    queueType queue;
};
```

semInit(s) 1
P1: semWait(s) 0
P2: semWait(s) -1 → P2
P3: semWait(s) -2 → P2 → P3
P1: semSignal(s) Strong sem -1 → P3

- Three atomic operations on semaphore
  - Initialize s: semInit(s)
  - Increase s: semSignal(s)
  - Decrease s: semWait(s)
- Semaphores: A special data structure, including an integer variable and a queue.
  - Integer is used for signaling
  - Queue is used to hold processes waiting on the semaphore
- semWait(P) primitive is used to receive signal
- semSignal(V) primitive is used to send signal
- semWait and semSignal cannot be interrupted

● **强信号量**：FIFO

信号量三大用途

● *互斥*：初始化为 1
● *有限并发*：初始化为 n
● *通信* signaling：初始化为 0

一、*会合 Rendezvous*

Guarantee that a1 happens before b2 and b1 happens before a2.

Create two semaphores, named aArrived and bArrived,
- aArrived indicates whether Thread A has arrived at the rendezvous, and bArrived likewise
- and initialize them both to zero.

```
Process A:                  Process B:
statement a1;               statement b1;
semSignal(aArrived);        semSignal(bArrived);
semWait(bArrived);          semWait(aArrived);
statement a2;               statement b2;
```

多人会合在同一点：

- Define shared variables:

```
int n = the number of threads;
int count = 0; //count keeps track of how many processes have arrived
Semaphore mutex = 1; //mutex provides exclusive access to count so that
                     //threads can increment it safely
Semaphore barrier = 0; //barrier is locked (zero) until all threads arrive
```

- Process i:

```
rendezvous i:
semWait(mutex);
count = count + 1;
semSignal(mutex);
if (count == n) semSignal(barrier);
semWait(barrier);
critical point i;
```

```
rendezvous i:
semWait(mutex);
count = count + 1;
semSignal(mutex);
if (count == n) semSignal(barrier);
semWait(barrier);
semSignal(barrier);
critical point i;
```

二、例题

(1) 生产者-消费者问题

Unbounded buffer, 1 producer, 1 consumer
- ✓ `in` modified only by producer and `out` only by consumer
- ✓ no race condition: no need for mutexes, just a while loop

```
item[] b;
int in, out;

void producer()
{
  while (true) {
    item = produce();
    b[in] = item;
    in++;
  }
}
```

```
void consumer()
{
  while (true) {
    while (out == in);
    //忙等待，不能从空缓冲区中取
    item = b[out];
    out++;
    consume(item);
  }
}
```

② 有限 buffer，多产多消

```
Producer:
while (true)
{Produce an item;
 semWait(empty);
 semWait(mutex);
 Buffer(in)=item;
 in=(in+1)%n;
 semSignal(mutex);
 semSignal(full);}
```

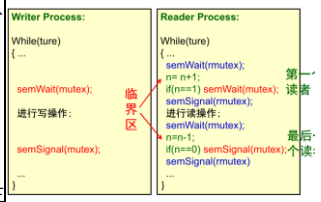
```
Consumer:
while (true)
{
 semWait(full);
 semWait(mutex);
 item=Buffer(out);
 out=(out+1)%n;
 semSignal(mutex);
 semSignal(empty);
 consume the item;}
```

先请求资源，再管 mutex。
先 signal，再 wait。不然会死锁

(2) 读者写者问题

① 读者优先算法

readcount = 0; rmutex = 1; mutex = 1



② 公平情况算法

readcount = 0; rmutex = 1; mutex = 1; *wmutex = 1*

```
Reader process:
while(ture){
  semWait(wmutex);    //检测是否有写者存在，无
  semWait(rmutex);    //申请使用readcount
  readcount++;        //读者数量加1
  if (readcount==1) semWait(mutex); //如果此为第
  semSignal(rmutex);  //释放readcount的使用权，
  semSignal(wmutex); //恢复wmutex
  进行读操作;
  semWait(rmutex);    //申请readcount的使用权，
  readcount--;        //读者数量减1
  if(readcount==0) semSignal(mutex);//如果没有读
  semSignal(rmutex);  //释放readcount的使用权，
}
```

```
Writer process:
while{true}
  {
   semWait(wmutex);          //检测是否
   semWait(mutex);           //申请对数
   进行写操作
   semSignal(mutex);         //释放数据
   semSignal(wmutex);        //恢复wm
  }
```

2.4.5 管程

● 共享、安全、互斥
（1）P 唤醒 Q
● P 等待 Q 执行：Hoare
● Q 等 P 继续执行：MESA
（2）Hoare
● 不可用于通信：condition 链
● 入口等待队列、条件变量队列、紧急等待队列。
（3）管程与生产消费者问题

```
producer put(item) //放数据
  { item nextp;
    if (count==n)          //所有缓冲区为满缓冲区
      cwait(notfull);  //等待空缓冲区
    buffer(in)=nextp;
    in=(in+1)mod n;
    count=count+1;
    if( full队列不为空) csignal(notempty);
                          //唤醒等待满缓冲 区队列中的进程
  }
```

```
Procedure get(item)//取数据
{ item nextc;
  if (count==0) //所有缓冲区为空缓冲区
    cwait(notempty); //等待一个装满数据的缓冲区
  nextc=buffer(out);
  out=(out+1)mod n;
  count=count-1;
  if (empty队列不为空) csignal(notfull)}
                          //唤醒等待空缓冲区队列中的进程
```

```
{in=0; out=0; count=0; }  //管程变量初始化
```

```
Producer:
while(true) {
  生产item;
  pc.put(item);
}
```

```
Consumer:
while( true){
  pc.get(item);
  消费item;
}
```

2.5 死锁与饥饿

死锁检测 Deadlock detection
死锁恢复 Deadlock Recovery
死锁预防 Deadlock Prevention
死锁避免 Deadlock Avoidance

死锁、活锁与饥饿

- 饥饿：当多个进程同时申请某类资源时，OS按照一定的分配策略（调度策略）分配资源，分配策略可能是公平的，能保证请求者在有限时间内获得所需资源。也可能是不公平的，即不能保证等待时间的上界。当等待时间造成明显影响时，发生进程饥饿，以至于进程无法按要求完成时，称为饿死。
- 活锁：在忙等待的情况下发生的饥饿称为活锁。比如两个人相对过一个只供一人通过的窄道，两人都想礼让对方，但由于两人同时进行礼让，有可能两人同时让开，又同时抢道
- 饿死与死锁的差别：
  - 死锁进程等待的是永远不会释放的资源，饿死进程等待的是会释放但不会分配给自己的资源；
  - 死锁一定发生循环等待，饿死则不一定；
  - 死锁一定涉及多个进程，饿死的进程则可能是一个。

2.5.1 死锁四条件

● 资源互斥、持有等待、不可抢占
● 循环等待

2.5.2 数学建模

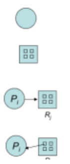
(1) 联合进程图示

Joint Progress Diagram

(2) 资源分配图

Resource Allocation Graph

- Process
- Resource Type with 4 instances
- P_i requests instance of R_j
- P_i is holding an instance of R_j



2.5.3 检测并恢复

● 所有资源、当前可用资源

● 总资源请求、当前已分配资源、还要多少资源

●检测时机：某进程开始等待、定时检测

一、资源分配图化简

● 先看 只有入边 的进程

● 只要这些进程赶紧走掉就能释放资源了

2.5.4 死锁预防

●静态

- Remove one of the design or scheduling conditions?
- remove "mutual exclusion"?
  - → not possible: must always be supported by the OS; 但可增加资源总数
- remove "hold & wait"?
  - require that a process gets all its resources at one time
  - → inefficient and impractical: defeats interleaving, creates long waits, cannot predict all resource needs
- remove "no preemption" = allow preemption?
  - require that a process releases and requests again →ok
- remove "circular wait"?
  - ex: impose an ordering of resources →inefficient, again

死锁的四条件
(1) 资源互斥
(2) 持有等待
(3) 不可抢占
(4) 循环等待

2.5.5 死锁避免

●动态

(1) 介入时机：有新资源请求

(2) 银行家算法：死锁检测扩展

小结：银行家算法中的数据结构

- 资源向量 Resource：是一个含有m个元素，其中的每一个元素代表一类资源全部数目。
- 可利用资源向量Available：是一个含有m个元素，其中的每一个元素代表一类可利用的资源数目，其初值是系统中所配置的该类全部可用资源数目。如果Available[j]=k，表示系统中现有R_j类资源k个。
- 最大需求矩阵Claim：是一个含有n×m的矩阵，它定义了系统中n个进程中的每个进程对m类资源的最大需求。如果Claim(i,j)=k，表示进程需要R_j类资源的最大数目为k。
- 分配矩阵Allocated：是一个含有n×m的矩阵，它定义了系统中每一类资源当前已分配给每一进程的资源数。如果Allocation(i,j)=k，表示进程i当前已分得R_j类资源k个。
- 需求矩阵Request：是一个含有n×m的矩阵，用以表示每一个进程尚需的各类资源数。如果Need(i,j)=k，表示进程i还需要R_j类资源k个，方能完成其任务。

Request(i,j)= Claim(i,j)-Allocated(i,j)

Part 3 内存管理

3.1 实内存管理

●存储管理：掉电后信息丢失

●设备管理：掉电后信息仍在

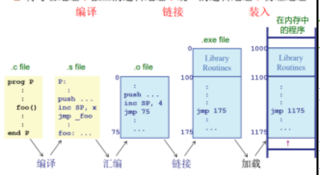
●主存的作用：CPU 取指令与数据，CPU 与外设通讯。

●时间局部性：循环

●空间局部性：顺序执行

3.1.2 程序的装入

- 符号名地址→独立的逻辑地址→统一的逻辑地址→物理地址（编译 链接 装入）



●绝对装入：仅适用于单道

●静态重定位：装入已是 PA

●动态重定位：装入仍是 LA

3.1.3 内存分配方案

Fixed Partitioning：分书架、可以 equal-size 也可以 unequal。涉及分区表、进程表。

Dynamic ~：堆箱子。涉及分配分区表、空闲分区表(链)

First-Fit：每次都从全局开头找

Next-~：从上次分配位置开始找

Best-Fit：恰恰好

Worst-Fit：找最大空格、土豪

- 从搜索的空闲分区速度及主存利用率来看，最先适应分配算法、下次适应分配算法和最佳适应算法比最坏适应算法性能好。
- 如果空闲区按从小到大排列，则最先适应分配算法等于最佳适应分配算法。反之，如果空闲区按从大到小排列，则最先适应用分配算法等于最坏适应分配算法。
- 空闲区按从小到大排列时，最先适应分配算法能尽可能使用低地址区，从而，在高地址空间有较多较大的空闲区来容纳大的作业。下次适应分配算法会使存储器空间得到均衡使用。
- 最佳适应分配算法的主存利用率最好，因为，它把刚好或最接近申请要求的空闲区分给作业；但是它可能会导致空闲区分割下来的部分很小。在处理某种作业序列时，最坏适应分配算法可能性能最佳，因为，它选择最大空闲区，使得分配后剩余下来的空闲区不会太小，仍能用于再分配。
- 由于最先适应算法简单、快速，在实际的操作系统中用得较多；其次是最佳适应算法和下次适应算法

- While performance depends on the exact sequence of process requests and sizes, statistical conclusions can be reached:
- Best-fit placement
  - paradoxically, the worst performer! it quickly litters memory with small fragments and requires compaction frequently
- First-fit placement
  - the best and fastest
- Next-fit placement
  - the runner-up: slightly worse than first-fit, because it spreads fragmentation more evenly (whereas first-fit has a tendency to preserve big blocks at the end of memory)

Buddy System：二分与融合

求兄弟：找到对应的位，翻一下。

3.1.4 内存不够怎么办

节流：overlay (固定区、覆盖区)

开源：compact、swap

swap 文件内类似动态分区。

3.2 虚拟内存

resident set 驻留集

Translation Lookaside Buffer

●页号从 0 开始

- 连续模式：给一个进程分配一个连续的内存空间
  - 单一连续分配：只适用于单道程序系统
  - 固定分区：分区大小固定
  - 动态分区：分区大小依程序大小变化
  - 伙伴系统：固定分区与动态分区的综合
- 非连续模式：给一个进程分配多个分散的内存空间
  - 分页：存储单元固定
  - 分段：存储单元依模块大小变化
  - 段页式：分段与分页的综合

3.2.1 内存分页

●小页面：减少页内碎片，提高内存利用率；但进程页表变长、且会降低页面换进换出的效率。

●LA 转 PA

●缺页中断的处理

- 发起缺页中断：MMU 查询页表失败，即目标内存页已被 swap 出内存
- OS 将进程置为 Blocked
- OS 发起 I/O 请求，将页面从磁盘中 swap 入内存
- 在 I/O 操作在后台运行的时候，CPU 将让给下一个进程
- I/O 完成，硬件发起 I/O 中断，OS 返回
  - 修改页表
  - 将上述进程重新置为 Ready

●哈希页表：把「进程号+页号」拿去哈希

3.2.2 内存分段

●分段在编译时由编译程序完成。

3.2.3 段页式

段号S	段内页号P	页内地址W

页偏移量W

段内地址d（段偏移量）

3.2.4 Replacement Policy

	Local Replacement	Global Replacement
Fixed Allocation	每个进程分配固定页框数的内存空间，在整个运行期间不再改变	
Variable Allocation	• 若运行中频繁缺页中断，则再增加若干页框，直至缺页率减少到适当值为止。 • 主导权在驻留集管理 • 关键是确定驻留集大小和变化的算法，即工作集策略	• 主导权在置换算法 • 先为每个进程分配一定量的页框，OS 有一个空闲页框队列，某进程缺页时从空闲队列取页框装页，当空闲队列为空后，OS 再调出某一进程的页置换

3.2.5 开算

注意，性能曲线中 OPT 无论如何都无法与其它相交，它是界。

●最优置换策略 OPTimal

淘汰掉在未来最晚用掉的。

●最近最久不使用 LRU

> 约定最顶上的一个最安全。

> 只要一有内存访问，就要把访问的页面提前。

> 可行的最优策略，但开销大

●免死金牌 Clock/NRU

>新页面起手两条命、刚被访问的页面变成两条命

>**蓝色**是收回“免死金牌”的机制：被蓝色指到的页面，扣一条命；扣到没命的就寄、被换出、蓝标给下一个人——先杀人，再移动。

>初始置页的时候，蓝标是卡在第一格不变的。**别做快，记得扣命！**

若加多一个 M 位，则扫第一遍找 A/M 均为 0 的、第二遍扫 A=0 的、并按照 NRU 扣金牌，必要时重复上述两步。

●FIFO

Part 5 I/O Management

●Block devices 块设备

可寻址、块式存储、互独访问

e.g. 磁盘、磁带

●Character/Stream devices

字符流、不可寻址。键鼠/网络

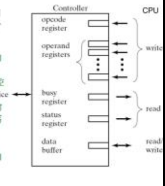
Basic Functions of I/O Subsystem

- Presents a logical or abstract view of communication and storage devices to the users and to other high-level subsystems by hiding the details of the physical devices;
  - 隐藏物理设备的差异性
- Facilitates efficient use of communication and storage devices;
  - 尽可能地使CPU与输入输出设备并行
- Supports the convenient sharing of communication and storage devices.
  - 共享设备的使用：设备保护、分配和调度

- Each I/O device consists of two parts:
  - the controller or module, containing most of the electronics
  - the device proper, such as a disk drive
- The job of the controller is:
  - to control the I/O and
  - handle bus access for it

- Why I/O controllers? Why not connecting the devices directly to the bus?
  - wide variety of peripherals with various operation methods: don't want to incorporate heterogeneous logic into CPU
  - controllers offer a more unified hardware command interface
  - data transfer rate slower or faster than memory or CPU
  - different data and word lengths
  - multiplexing: one module serving several devices (ex: SCSI)
- Functions of an I/O device controller
  - interface to CPU and memory via system bus
  - interface to one or more peripherals by custom data links

- 接口寄存器（可寻址的，即系统为其分配了地址），可分成：
  - 数据接口寄存器（输入时，由控制器写CPU读，输出相反）
  - 地址接口寄存器（CPU写，控制器读）
  - 控制寄存器（控制信息是CPU向控制器发的命令，由CPU写，控制器读）
  - 状态接口寄存器（控制器向CPU反映的设备状态信息，由控制器写，CPU读）



- I/O地址空间：I/O功能的地址集合称为I/O空间，包括：
  - 所有设备寄存器
  - 内存中用以映射设备的数据缓冲区
- I/O地址空间在系统中配置的三种模式
  - 独立于内存的I/O地址空间模式：I/O地址空间与内存地址空间相互独立，对I/O地址空间的访问需要用特殊的I/O指令。
  - 映射到内存空间的I/O地址空间模式：I/O寄存器映射成内存的一部分，可以用内存访问语句来读写I/O设备寄存器。Memory-mapped
  - 混合模式 → buffer: 内映，控制寄存器: 独编

- Programmed I/O (or "polling" or "busy waiting"): 可编程I/O
  - the CPU must repeatedly poll the device to check if the I/O request completed
- Interrupt-driven I/O: 中断I/O
  - the CPU can switch to other tasks and is (frequently) interrupted by the I/O device
- Direct Memory Access (DMA): 直接存储访问
  - the CPU is involved only at the start and the end of the whole transfer; it delegates control to the I/O controller that accesses memory directly without bothering the CPU

- 缓冲的引入
  - 缓和CPU与I/O设备之间速度不匹配的矛盾
  - 减少对CPU的中断频率，放宽对CPU对中断响应时间的限制。
  - 提高CPU和I/O设备之间的并行性
- 缓冲的实现方法
  - 采用专用的硬件缓冲器（如Cache，I/O设备和控制器的专用缓冲区）
  - 在内存中划出一个具有n个存储单元的缓冲区，称"软件缓冲"，也称"内存缓冲区"。
- 缓冲技术实现的有效性前提
  - 快方的快速阵发性(包括突发性与间歇性)的快，而不是持续性的快（例:各种I/O缓冲区）。
  - 访问局部性，即局部的数据被连续多次访问（例:Cache）。

调度算法	为解决什么问题而引入	优点	缺点
FIFO 先进先出	直观的解决方案	简单、公平	未对寻道进行优化，所以平均寻道时间较长
SSTF 最短服务时间优先	为了解决FIFO算法平均寻道时间过长的问题	比FIFO的平均寻道时间要短	并非最优，而且会导致“饥饿”现象
SCAN 扫描（电梯）	为了解决SSTF导致的“饥饿”现象	兼顾了较小的寻道时间和防止饥饿	偏向于两端（最内、最外）的磁道
C-SCAN 循环扫描	为了解决SCAN的偏好	兼顾了较小的寻道时间和防止饥饿	磁臂黏着（磁臂长期停留在某处不动的情况）

Part 6 File

●组块因子 F：一个组块里面有多少条记录。

● File Logical Structure

指文件内部的结构安排

- 字节流
- Record Sequence
  - Pile
  - Sequential File
  - Indexed File
  - Indexed Sequential File
- Tree

6.1 File System Implementation

●盘上信息：启动扇区、已用未用空间管理、目录结构、文件

●内存中信息：两张 FCB、"DCB"、挂载分区信息

> 进程的 FCB：文件描述符(指针)

6.1.1 Directory Implementation

●Linear list

●Hash table：基于链表的改进

6.1.2 Record blocking

结构化文件的组块、I/O 基本单位

- Fixed blocking
- Variable blocking
  - unspanned
  - spanned

6.1.3 File Block Allocation

	连续	链式	索引	
是否预分配	需要	可能	可能	
分区大小是固定还是可变	可变	固定块	固定块	可变
分区大小	大	小	小	中
分配频率	一次	低到高	高	低
分配需要的时间	中	长	短	中
文件分配表的大小	一个表项	一个表项	大	中

●Contiguous Allocation

●显式 Chained:文件名,起始,长度

●隐式(Implicit) Chained：FAT

●Indexed Allocation

把 block 表也存进 block 里

○改 1：var-length portion

○改 2：Multilevel Indexing(inode

6.1.4 Free-space Management

把所有空闲空间连一起看作一个大文件。●Bitmap ●Linked list

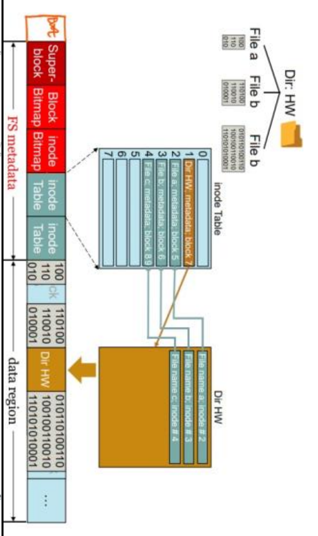
●Grouping:记 index 的 block 成链

●Counting:类似 IndexAlloc 的改 2

6.2 inode

●文件名不存在文件的 inode 中，而是存在目录的 inode 中。

●12 个一级指针（直接指针）



Word Bank

locality 局部性 integrity 完整性

hierarchy 等级制度 LRU,cache,TLB

peripheral device 外设 exploit 开发

simultaneously 同时地

confidentiality 机密性

chronological 按时间顺序的

timeliness 及时性 temporal loca~

rewind 倒带 propensity 倾向

dangling pointer 悬挂

acyclic-graph 无环图

transient 短暂的 malicious 恶意的

analogous 相似的

explicit 显式的、明确的

implicit 隐式的 concatenate 连接

detach 分离；派遣 spatial local~

integrated 综合的、融合的

Redundant Array of Independent Disks 冗余磁盘阵列

cognitive ability 认知能力

propensity 倾向;习性

macroscopic 宏观的；肉眼可见的

hypothetical 假设的 sufficient 足够

dedicate 致力；献身

duplicate *n./adj.* 完全一样的

provably 可证明地 nudge 推动

inheritance n. 继承物

Lab 1 系统软件启动过程

Bootloader 进入保护模式；Bootloader 加载 ELF 格式的 OS 的过程；函数调用堆栈跟踪函数；中断初始化和处理

Lab 2 物理内存管理

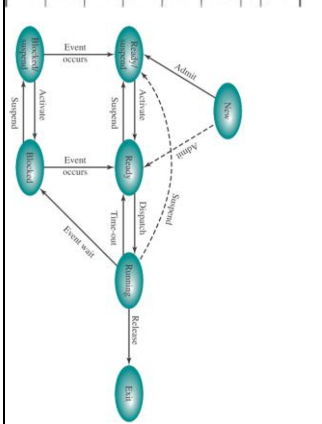
实现 first-fit 连续物理内存分配算法；寻找虚拟地址对应的页表项释放某虚地址所在的页并取消对应二级页表项的映射

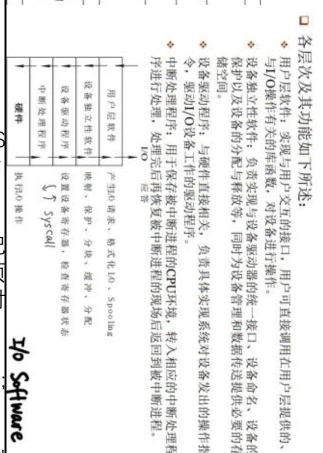
Lab 3 虚拟内存管理

给未被映射的地址映射上物理页；基于 FIFO 的页面替换算法

Lab 5 用户进程管理

加载应用程序并执行；父进程复制自己的内存空间给子进程





by @Jonbgua, https://icourse.club/user/6436