Operating System Review

胡祥龙 (Xianglong Hu)

07/29/2018

Contents

1 2.3 Interprocess Communication

1.1 race conditions

121.

1.2 critical regions

121.

1.3 disable interrupts

too much privilege, cannot be applied to multicore processors.

1.4 lock variables

race is still there.

1.5 busy waiting

the lock is called the spin lock, 但是 Process A 会在 Process B 不在 critical region 的时候等待，导致浪费了时间。

```
while (TRUE) {
while (turn != 0) /*loop*/ ;
critical region();
turn = 1;
noncritical region();
}
while (TRUE) {
while (turn != 1) /*loop*/ ;
critical region();
turn = 0;
noncritical region();
}
```

1.6 Peterson's solution

这个不错，但是似乎只能用于两个 Processes?

```
#define FALSE 0
#define TRUE 1
#define N 2 /*number of processes*/
```

```
int turn; /*whose turn is it?*/
int interested[N]; /*all values initially 0 (FALSE)*/
void enter region(int process); /*process is 0 or 1*/
{
int other; /*number of the other process*/
other = 1 - process; /*the opposite of process*/
interested[process] = TRUE; /*show that you are interested*/
tur n = process; /*set flag*/
while (turn == process && interested[other] == TRUE) /*null statement*/ ;
}
void leave region(int process) /*process: who is leaving*/
{
interested[process] = FALSE; /*indicate departure from critical
region*/
}
```

1.7 TSL(Test and Set Lock) and XCHG

是从硬件的层面解决这个问题。在修改的时候硬件可以把 memory bus 给锁死，从而使得 race problem 消失。这两个操作在 *mutex* 和 *semaphore* 里有运用。

1.8 busy waiting 的问题: priority inversion

128，简单来说就是 L 进入 critical region 后被 interrupt 了，H 无法进入 critical region，从而形成了僵持。

1.9 The Producer-Consumer Problem

128. 同样是由于 race condition 的问题，由于对 count 的访问没有限制的缘故。

> The essence of the problem here is that a wakeup sent to a process that is not (yet) sleeping is lost. If it were not lost, everything would work. A quick fix is to modify the rules to add a wakeup waiting bit to the picture.

```
define N 100 /*number of slots in the buffer*/
int count = 0; /*number of subsections in the buffer*/
void producer(void)
{
int subsection;
while (TRUE) { /*repeat forever*/
```

```
subsection = produce subsection( ); /*generate next subsection*/
if (count == N) sleep(); /*if buffer is full, go to sleep*/
insert subsection(subsection); /*put subsection in buffer*/
count = count + 1; /*increment count of subsections in buffer*/
if (count == 1) wakeup(consumer); /*was buffer empty?*/
}
}
void consumer(void)
{
int subsection;
while (TRUE) { /*repeat forever*/
if (count == 0) sleep(); /*if buffer is empty, got to sleep*/
subsection = remove subsection( ); /*take subsection out of buffer*/
count = count - 1; /*decrement count of subsections in buffer*/
if (count == N - 1) wakeup(producer); /*was buffer full?*/
consume subsection(subsection); /*print subsection*/
}
}
```

1.10 semaphore

semaphore 说白了只是一种特殊的 data structure 只不过确保了所谓 **atomic action** 罢了。这种 **indivisibility** 的性质是在硬件的层面而实现的。我不太理解的是这里 *mutex* 的作用。

> A mutex is essentially the same thing as a binary semaphore and sometimes uses the same basic implementation. The differences between them are in how they are used. While a binary semaphore may be used as a mutex, a mutex is a more specific use-case, in that only the thread that locked the mutex is supposed to unlock it. This constraint makes it possible to implement some additional features in mutexes:
>
> - Since only the thread that locked the mutex is supposed to unlock it, a mutex may store the id of the thread that locked it and verify the same thread unlocks it.
> - Mutexes may provide priority inversion safety. If the mutex knows who locked it and is supposed to unlock it, it is possible to promote the priority of that thread whenever a higher-priority task starts waiting on the mutex.
> - Mutexes may also provide deletion safety, where the thread holding the mutex cannot be accidentally deleted.
> - Alternately, if the thread holding the mutex is deleted (perhaps due to an unrecoverable error), the mutex can be automatically released.

- A mutex may be recursive: a thread is allowed to lock it multiple times without causing a deadlock.

1.11 mutex

感觉没啥可说的。

1.12 Monitors and Deadlocks

Semaphore 的问题在于太容易出错。请看代码:

```
#define N 100 /*number of slots in the buffer*/
typedef int semaphore; /*semaphores are a special kind of int*/
semaphore mutex = 1; /*controls access to critical region*/
semaphore empty = N; /*counts empty buffer slots*/
semaphore full = 0; /*counts full buffer slots*/
void producer(void)
{
int subsection;
while (TRUE) { /*TRUE is the constant 1*/
subsection = produce subsection( ); /*generate something to put in buffer*/
down(&empty); /*decrement empty count*/
down(&mutex); /*enter critical region*/
inser t subsection(subsection); /*put new subsection in buffer*/
up(&mutex); /*leave critical region*/
up(&full); /*increment count of full slots*/
}
}
void consumer(void)
{
int subsection;
while (TRUE) { /*infinite loop*/
down(&full); /*decrement full count*/
down(&mutex); /*enter critical region*/
subsection = remove subsection( ); /*take subsection from buffer*/
up(&mutex); /*leave critical region*/
up(&empty); /*increment count of empty slots*/
consume subsection(subsection); /*do something with the subsection*/
}
```

}

这里 Producer 的两个 down 如果反一反，那么 empty=0 的时候 Producer 被 block，同时 mutex 被锁死，那么 consumer 也会 block，这样两个 process 都会被无限 block，形成所谓的 deadlock.

因此我们要发展所谓的 monitor，所谓的 monitor 是把 mutual exclusion 更加抽象的一种数据结构。他的特点是 monitor 里的 process 一次只能有一个 active，否则会被 block。

monitor 的结构是两个操作，wait 和 signal，以及 condition variable。参见 java 代码：

```
public class ProducerConsumer {
static final int N = 100; // constant giving the buffer size
static producer p = new producer( ); // instantiate a new producer thread
static consumer c = new consumer( ); // instantiate a new consumer thread
static our monitor mon = new our monitor( ); // instantiate a new monitor
public static void main(String args[]) {
p.star t(); // star t the producer thread
c.star t(); // star t the consumer thread
}
static class producer extends Thread {
public void run( ) {// run method contains the thread code
int subsection;
while (true) { // producer loop
subsection = produce subsection( );
mon.inser t(subsection);
}
}
pr ivate int produce subsection( ) { ... } // actually produce
}
static class consumer extends Thread {
public void run( ) {run method contains the thread code
int subsection;
while (true) { // consumer loop
subsection = mon.remove();
consume subsection (subsection);
}
}
pr ivate void consume subsection(int subsection) { ... }// actually consume
}
static class our monitor { // this is a monitor
pr ivate int buffer[ ] = new int[N];
```

```
pr ivate int count = 0, lo = 0, hi = 0; // counters and indices
public synchronized void insert(int val) {
if (count == N) go to sleep( ); // if the buffer is full, go to sleep
buffer [hi] = val; // inser t an subsection into the buffer
hi = (hi + 1) % N; // slot to place next subsection in
count = count + 1; // one more subsection in the buffer now
if (count == 1) notify(); // if consumer was sleeping, wake it up
}
public synchronized int remove() {
int val;
if (count == 0) go to sleep( ); // if the buffer is empty, go to sleep
val = buffer [lo]; // fetch an subsection from the buffer
lo = (lo + 1) % N; // slot to fetch next subsection from
count = count - 1; // one few subsections in the buffer
if (count == N - 1) notify(); // if producer was sleeping, wake it up
retur n val;
}
pr ivate void go to sleep( ) { try{wait( );} catch(Interr uptedException exc) {};}
}
}
```

这里的 wait 和 signal 与 naive implementation 里的 sleep 和 wakeup 的区别在于:

> You may be thinking that the operations wait and signal look similar to sleep and wakeup , which we saw earlier had fatal race conditions. Well, they are very similar, but with one crucial difference: sleep and wakeup failed because while one process was trying to go to sleep, the other one was trying to wake it up. With monitors, that cannot happen. The automatic mutual exclusion on monitor proce- dures guarantees that if, say, the producer inside a monitor procedure discovers that the buffer is full, it will be able to complete the wait operation without having to worry about the possibility that the scheduler may switch to the consumer just be- fore the wait completes. The consumer will not even be let into the monitor at all until the wait is finished and the producer has been marked as no longer runnable.
>
> Java guarantees that once any thread has started executing that method, no other thread will be allowed to start executing any other synchronized method of that object. Without synchronized , there are no guarantees about interleaving.

Java synchronized 保护了 go_to_sleep, insert 和 remove 不被打扰。理论上可以被 interrupt, 所以要 explicit exception handling。

1.13 Message Passing

Producer-Consumer 的问题也可以用 message passing 的方式来解决。

```
#define N 100 /*number of slots in the buffer*/
void producer(void)
{
int subsection;
message m; /*message buffer*/
while (TRUE) {
subsection = produce subsection( ); /*generate something to put in buffer*/
receive(consumer, &m); /*wait for an empty to arrive*/
build message(&m, subsection); /*constr uct a message to send*/
send(consumer, &m); /*send subsection to consumer*/
}
}
void consumer(void)
{
int subsection, i;
message m;
for (i = 0; i < N; i++) send(producer, &m); /*send N empties*/
while (TRUE) {
receive(producer, &m); /*get message containing subsection*/
subsection = extract subsection(&m); /*extract subsection from message*/
send(producer, &m); /*send back empty reply*/
consume subsection(subsection); /*do something with the subsection*/
}
}
```

message passing 有两种机制：mailbox 和 rendezvous。后者可能需要知道一下。

> The other extreme from having mailboxes is to eliminate all buffering. When this approach is taken, if the send is done before the receive , the sending process is blocked until the receive happens, at which time the message can be copied direct- ly from the sender to the receiver, with no buffering. Similarly, if the receive is done first, the receiver is blocked until a send happens. This strategy is often known as a rendezvous.

1.14 Barriers

非常 trivial，没啥可说的。

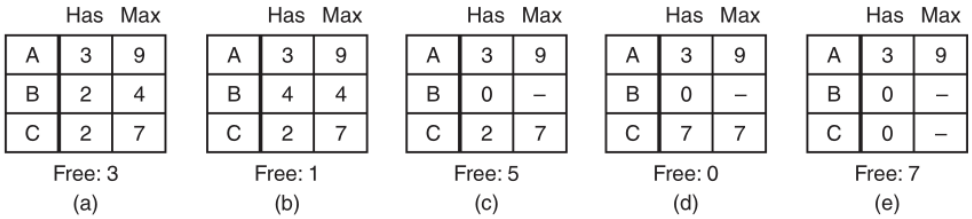


Figure 6-9. Demonstration that the state in (a) is safe.

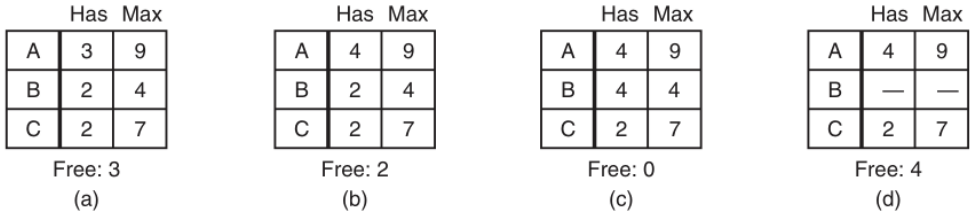


Figure 6-10. Demonstration that the state in (b) is not safe.

图 1

2 6. Deadlocks

这一章是和上一张在一个 PPT 里的，concurrency。

2.1 Deadlock Detection

算法见 446 页，对于单一资源来说，只需要有向图就可以了。然而对于多种资源来说，则需要一个 nonderterministic 的算法。多种资源的算法他根本没讲。

有几种 recovery 的办法。

- Recovery through Preemption
- Recovery through Rollback
- Recovery through Killing Processes

2.2 Deadlock Avoidance

450.Deadlock 是可以避免的。

2.2.1 Safe State

一个很 trivial 的概念 1。unsafe 不一定会 deadlock，但是 safe 可以保证不 deadlock。

2.3 Banker's Algorithm

其实和 detection 的算法几乎一样。

Condition	Approach
Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

图 2

2.4 Deadlock Prevention

- Mutual exclusion condition. Each resource is either currently assign- ed to exactly one process or is available. 解决这个问题就是不要把资源 exclusively 分配。
- Hold-and-wait condition. Processes currently holding resources that were granted earlier can request new resources. 提前说清所有的需求一次性分配。或者在要求新的资源的时候短暂释放所有的资源。缺点是资源的分配不是最优。
- No-preemption condition. Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.virtualiz resources. 但不是所有都可以 virtualize。
- Circular wait condition. There must be a circular list of two or more processes, each of which is waiting for a resource held by the next member of the chain. 一次只能用一个资源，但是不现实。把所有的需求进行编号，必须要按照顺序进行 request。见 458 页。

总结如 2.

3 2.1 & 2.2 Processes and Threads

这章感觉好像没什么可考，感觉基本上都是基本的概念什么的为之后的东西做铺垫。

Thread 有部分 Process 的特征，有自己的 stack，contains the execution history. Thread 在很大程度上互相独立，但是共享一个 Process environment。不同的 Thread 并没有不同的 Process 那样独立，他们共享共同的 address space，共同的 global variable。不同的 Thread 之间并没有互相保护。One thread could read, write or even wipe out another thread's stack.

书上 104 页有一张表 3 展示了二者的区别。

假如一个 thread 打开了一个文件，那么另外一个 thread 应该是可以 read 这个文件的，因为 Process is the unit of resource management, not the thread.

It is important to realize that each thread has its own stack.

Per-process items	Per-thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Figure 2-12. The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

图 3

3.1 Implemention of Thread

Thread 有两种 implementation，一种是在 user-level，由 process 自己实现。有几个好处。

- thread switching 比 trapping to the kernel 要快很多。
- process can have customized scheduling algorithm.

也有几个缺点, 缺点主要是一些 system call 的问题，因为 system 意识不到 thread 的存在，所以会选择影响整个程序。

- blocking system call 怎么实现是个问题。一个 thread 要 i/o 的时候假如 system call 会导致整个 process block。
- page fault 如何 handle，同样也是整个 process 也会被 block
- In pure ULT, multithreading cannot take advantage of multiprocessing. 这个地方我不是很理解，因为估计不同 *thread* 之间的 *cpu* 资源是怎么分配的可能要看具体操作系统的实现吧。

另外一种就是在 kernel 里面实现，基本上存的是 process 的一个子集，由于 kernel 创建的成本较高，但是可以调用所有的 system call 。

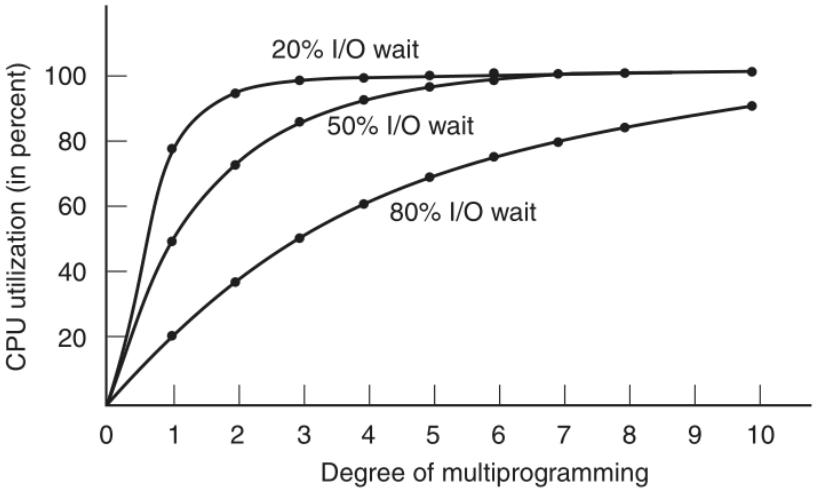
3.2 State Model

3.2.1 Three State Model

running, block, ready

3.2.2 Five State Model

new, running, block, ready, exit



3.3 Simple Model of Multiprogramming

4 File Systems

Files from a user's perspective:

File 的几个属性，Naming，Structure，

Naming two-part file names

Structure Byte sequence, record sequence(这个是上古 punch card 的遗留), tree(这个和 record sequence 差不多，只不过长度不一样，所以要换个形式组织罢了。)

Type regular file(binary or ASCII files), directory(system files to maintain the structure of the file system), block special files(model disks).

Access sequencial and random access(可以读取文件里的任意一个位置).

Attributes(metadata) 就是一些文件的属性，其实没什么可说的。

Directory 感觉没什么可说的，主要是会区别 hard link 和 symbolic link 吧。

A link of this kind, which increments the counter in the file's i-node (to keep track of the number of directory entries containing the file), is sometimes called a hard link.

A variant on the idea of linking files is the symbolic link. Instead, of having two names point to the same internal data structure representing a file, a name can be created that points to a tiny file naming another file.

4.1 Implementation

感觉这个是考试的重点。

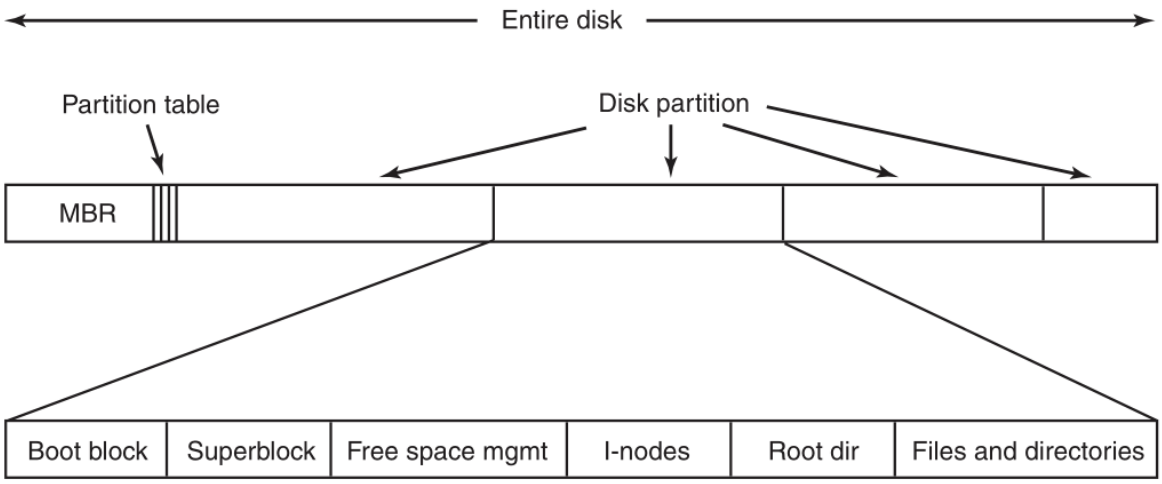


图 4

4.1.1 Layout

文件是存在硬盘上的，一般来说硬盘都会进行分区。每个分区都有自己的结构，这个结构就叫做 layout。Sector 0 总是 MBR(Master Boot Record), 这个是用来 boot the computer. 这个每个单独的分区。结构如图 4, 这里注意 I-node 是存在硬盘的空间里的，并不是属于 directory 或者 file 本身的一种性质：

4.1.2 Implementing Files: Allocation

有两种 allocation 的策略，contiguous 和 linked-list。

contiguous:

- Simple to implement.
- Read performance is excellent.
- Disk becomes fragmented.
- Need to know the final size of a file when the file is created.

linked list:

- No (external) fragmentation
- The directory entry needs just to store the disk address of the first block.
- Random access is extremely slow.
- The amount of data storage is no longer a power of two, because the pointer takes up a few bytes.

由于 linked list 的一些缺点，演化除了所谓的 FAT(File Allocation Table), 这个其实就是把 linked list 里的指针单独拿出来存在一张表里然后一口气读到内存里。所以缺点是 does not scale to large disks.

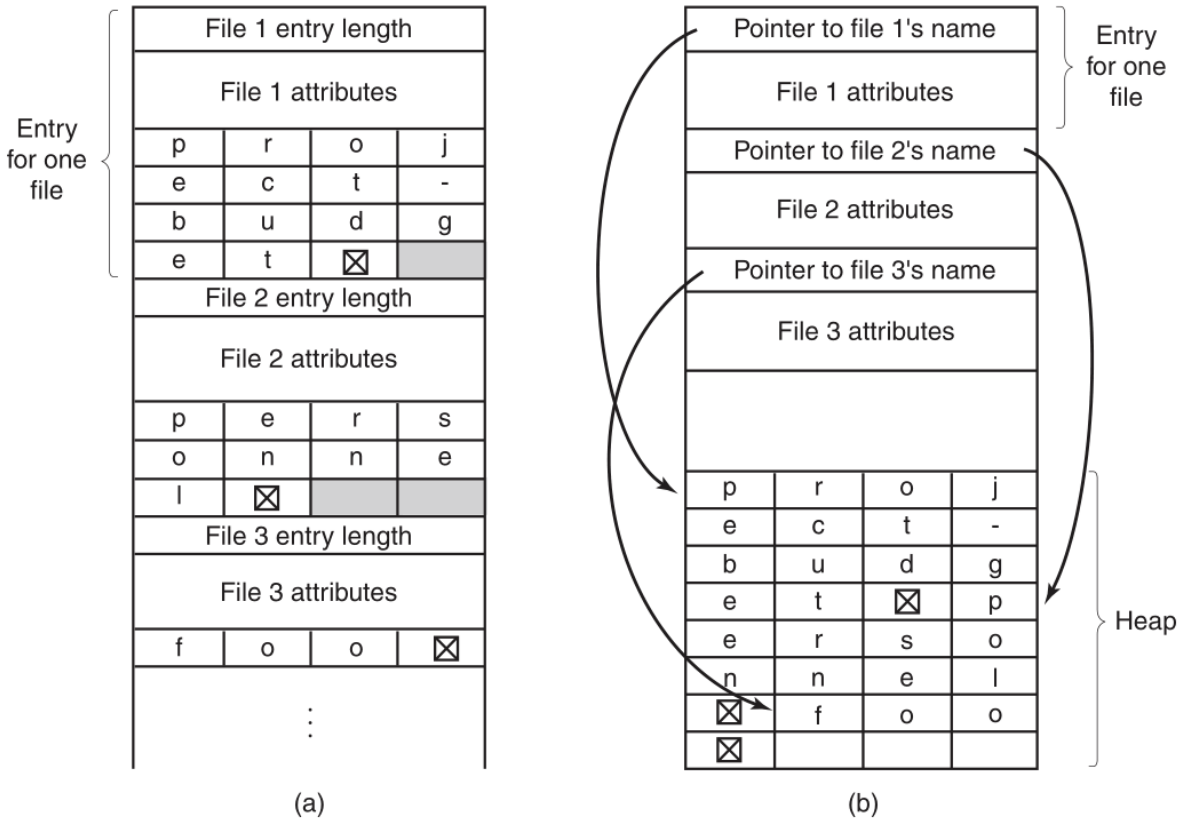


Figure 4-15. Two ways of handling long file names in a directory. (a) In-line. (b) In a heap.

图 5

4.1.3 I-node

这应该是非 *lab* 里的重中之重。I-node 是和每个 file associated 的，纪录了 file attribute 和 block disk address， 和 FAT 相比的优点是 *Need only be in memory when the corresponding file is open.*，也就是 *file metadata.*

4.2 Implementing Directories

Directory 其实就是 mapping name to I-node 的过程，以前都是比较短的 file name。对于比较长的话有两种方案 5，可以用 hash 加快寻找速度。

4.2.1 Shared Files

两个用户同时使用一个文件。第一种解决方案 hard link，增加了 i-node 的 count number，但是 C 假如删了这个文件之后，B 面临一个 invalid i-node. 对 i-node 来说也不可能找到 directory entry(dentry) 来删除它。因为没有从 i-node 指向 direcrtory 的 pointer，否则太多了存不下。

symbolic linking 的是创造一个 link 文件，里面存着 path name 指向另一个文件。

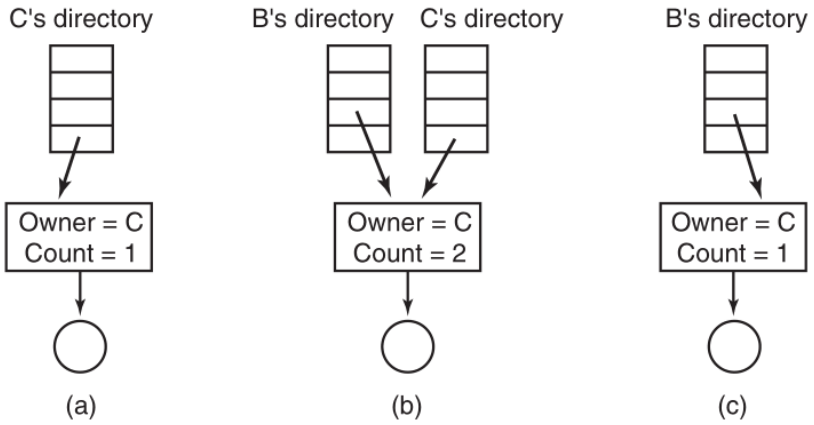


Figure 4-17. (a) Situation prior to linking. (b) After the link is created. (c) After the original owner removes the file.

图 6

概括来说 symbolic 的问题就是读起来慢，所谓的 overhead，优势是可以读世界上任何一台机器的文件。

The problem with symbolic links is the extra overhead required. The file con- taining the path must be read, then the path must be parsed and followed, compo- nent by component, until the i-node is reached. All of this activity may require a considerable number of extra disk accesses. Furthermore, an extra i-node is needed for each symbolic link, as is an extra disk block to store the path, although if the path name is short, the system could store it in the i-node itself, as a kind of opti- mization. Symbolic links have the advantage that they can be used to link to files on machines anywhere in the world, by simply providing the network address of the machine where the file resides in addition to its path on that machine.

4.2.2 Log-structured and journaling

这个初中是 writing 太慢了，所以吧 writing 攒着一起写，i-node 散落在各处。
衍生出 jounaling，保证文件系统的安全。

4.3 Management and Optimization

Block Size, 越大越快，越浪费空间。越小越慢，越节约空间。

4.3.1 Backup

incremental dumps. physical dump. logical dump.
其他两个没什么好说的，就 logical dump 可以提一下。

Starts at one or more specified directories · Recursively dumps all files and directories found there and have changed since some given base date.

5 Memory Management

前面主要是说 memory abstraction 是必要的，这个先不管。

5.1 Swapping

Base and Limit are two special hardware registers. 当这两个被使用的时候，程序会被加载进连续的 memory space。

Memory Space 不够用的时候就把部分 Program 拿出来放到 disk 上，之后再 bring in，这个过程就叫做 swapping。这个过程之中会产生 swapping hole 可以通过 memory compaction 完成。

空闲空间的管理是用 bitmap 或者 linked list。Bitmap 有点慢，必须要寻找合适的大小。linked list 的每个 entry 都要记录起始点，长度，以及上一个和下一个 entry。

重点来了，有好几个 allocate 的策略。

- first fit, 永远总开头开始找，第一个
- next fit, 这个和 first fit 比较像，区别是每次是从上次找到的位置开始而不是每次都从开头。
- best fit，找遍整个 list， 找到最小的满足条件的那个
- worst fit, 找最大的那个。

5.2 Virtual Memory

Motivation: It is not necessary that all of the pieces of a process be in main memory during execution.

Definition: Mapping from logical (virtual) address

5.3 Lab 03

这里是结合 *lab* 来讲的。每个 process 有自己的 address space，最小单位是 page，CPU 通过 MMU（一个模块）将 virtual address 翻译成 physical address。

PTE（Page Table Entry），是每个 page 都有的，记录了 referenced，modified，present，protection 等信息。得到 PTE 之后如果 valid，就直接读取。否则的话就用算法选出一个 PTE，使之无效并使用之。

TLB(Translation Lookaside Buffer) 是在 MMU 里的一个硬件，他可以绕过 PTE，直接把 page 翻译成 Physical frame。

For large memory 来说，要么使用 multilevel page table, 要么使用 inverted Page Table。Inverted Page Table 就是每一个 frame 纪录一个 PTE，纪录 virtual page 的位置。缺点是查找太慢。

5.3.1 Replacement Algorithm

重中之重。是把正在正在内存里的 *Pte* 占用的 *frame* 挑选。其中 LRU 实现的成本比较高，因为每个 page 都要记录下使用的时间，一般是想办法模拟。

NRU(Not Recently Used) 分成四类，referenced 周期性清空，在最低的一类中随机挑选一个扔掉。

FIFO(First in First Out) too trivial to talk about.

Second Chance Modification to fifo, 仍然按照时间顺序检查，不过只要 R=1，清空 R，并视作年龄最新，检查下一个。

Clock 本质上和 second chance 是一个算法，只不过实现上更巧妙罢了。

LRU(Least Frequently Used) 一般可以在硬件层面上实现，实现算法挺有趣的。其他就是每个 pte 里有一个 field 记录时间。

Aging 这个其实挺有趣的，不过也没什么特别的。就是用一列向量记录时间罢了。

5.4 Working Set Problem

其实就是决定 working set 的问题。load process 的时候是没有内存的，如果不提前 load 一些内存的话，那么就会一直 page fault，这个现象称之为 thrashing。我们希望 prepaging。所以要决定 working set。

有两种算法，一种就很 basic，就是不不停的 track 最近使用的 page。另外一种称之为 WSClock，其实就是 Clock 算法的改良，唯一的区别就是纪录了时间的信息，而这个时间也是虚拟的，称之为 virtual current time, 是程序实际的运行时间。

5.5 Design Issues

还有对 global algorithm 与 local algorithm 的讨论，就是 replacement 是在 process 的层面进行与否。global 显然更好，local 的话每个 process 的配额可以通过 page fault frequency 动态的调整。

还有就是对 page size 大小的讨论，都一样。大了浪费内存空间（internal fragmentation），小了浪费 TLB 的空间以及 page table。因为读写速度 doesn't matter here。

所谓的 loadcontrol:

What if PFF indicates that some processes need more memory but none need less? Swap some processes to disk and free up all the pages they are holding.

还要注意 locking pages in memory when doing I/O.

另外 swapping when page out. 这个称之为 backing store。

这东西怎么考啊，感觉还是要看看代码。

6 Scheduling

不同种类的操作系统有不同的算法。主要有三种: Batch System，Interactive System, Real-time System。

lab 里需要实现以下几种算法:

FCFS(First Come First Serve) 按照先后顺序，最为 trivial。

LCFS(Last Come First Serve) 按照先后顺序，也很 trivial。

SJF(Shortest Job First) 剩余时间最短的。

RR(Round Robin) 剩余时间最短的, promote fairness。

PRIO(Priority) 将不同的 Process 分级，内部仍然是 RR。

书上还提到了几种额外的:

lottery 随机分配，几率根据 ticket 的比例决定。

fair share 主要是多用户的时候每个用户给予一定的配额。

Batch System 没有用户交互，因为希望尽可能快的完成任务。比如 FIFO，SJF。