Principles of Operating Systems

Mike Spivey
Michaelmas Term, 1993

Warm-up exercises

1. How long would it take you to fill a floppy disk by typing?

2. Over what distance are an undergraduate, a floppy disk and a bicycle faster at delivering data than a telephone line transmitting 1200 bits per second?

3. If everything went a million times slower, how long would it take for (a) an IBM PC/AT to do one 16-bit addition, (b) the same machine to read one disk block, (c) You to type your name?

Chapter 1: Introduction

Books

The set book for this course is

Andrew. S. Tanenbaum, *Operating Systems: Design and Implementation*, Prentice-Hall International, 1987.

Other useful books:

Jon Bentley, *Programming Pearls* and *More Programming Pearls*, Addison-Wesley, 1986 and 1988.

Jim Welsh and Michael McKeag, *Structured System Programming*, Prentice-Hall International, 1980.

Douglas Comer and Timothy V. Fossum, *Operating Systems Design*, Prentice-Hall International, 1988.

Practicals

1. [Processes] Use concurrent processes to implement the software for a 'gardener's thermometer'. Start 2nd week; due 6th week.

2. [File systems] Simulate the cache of disk blocks used in MINIX. Start 4th week; due early next term.

Both practicals involve programming in C.

Learning C

- Appendix A of Tanenbaum's book gives a quick introduction – enough for reading C programs.
- Copying the code in the book will help.
- The practicals start from a working base program.
- If desperate, look at

  Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall International, 1978 or 1988.
- Avoid books entitled "How to hack it in Turbo C".

1.1 What is an operating system?

A systems program that

- provides an extended machine
- manages sharing of resources

for client programs

Things that are not the operating system

- Command interpreters (= 'shells')
- Compilers
- Editors
- Database systems
- Applications programs
- System utilities
- Subroutine libraries

But the techniques of 'programming in the large' apply to many of these.

Example

You type "`ls`" on the keyboard and press the `RETURN` key. A listing of the current directory appears on the screen. That's the user's view: but what 'really' happens?

One point of view:

- The keyboard interface puts scan code 37 into its buffer and raises the interrupt request line. The processor executes interrupt 40 .... Later, the processor issues a seek command for cylinder 437 on hard disk 0, then a read command for sector 6 on surface 3 .... The processor writes the string `hello.c hello.o ...` into display memory .... *[This is the hardware's point of view].*

Same event, other points of view

- The shell receives the line of terminal input it requested, so it is ready to run. The background tasks are suspended, and the shell gets the processor. After a while, the shell makes a read request on a disk file, so it is suspended, and the background tasks can run again .... The shell starts a new process running the `ls` program and waits for it to finish. During executiong of the `ls` program, it pauses to read disk files and write to the screen. During these pauses, background tasks can use the processor again. *[This is the scheduler's point of view].*

- User `mike` has permission to execute the `ls` program. It is stored in blocks 21236 to 21238 of hard disk 0; these blocks are read and loaded into memory. The `ls` program reads the current directory, stored in blocks 2344 and 2345. These blocks are fetched and copied into memory. The output of the `ls` program is connected to the screen, so the characters it outputs are sent there. *[This is the file system's point of view].*

More points of view

- Hard disk 0 has 1023 cylinders, 5 surfaces, and 19 blocks per track. So block 21236 is cylinder 223, surface 2, block 3. To read it, we issue a seek command for cylinder 223, wait for the arm to move, and check the status. If that's OK, we issue a read command for surface 2, block 3, wait for it to complete, and check the status again .... To write characters to the screen, we copy them to display memory at the place indicated by the cursor and move it on. *[This is the device drivers' point of view].*

- To make a directory listing, open the current directory as a file and read it 16 bytes at a time. The first 14 bytes of each record are the file name: collect these names, sort them into alphabetical order, and write them on the standard output. [This is the point of view of the `ls` program].

1.2 Parts of an operating system

Not all operating systems have all these parts, but most support

- Processes
- Input/output devices
- Memory management
- File system

In this course, we'll look at each of these in turn.

Providing an extended, virtual machine

- The scheduler provides a virtual machine that can run many independent processes concurrently.
- The memory manager provides each process with the memory it needs to store its code and data.
- Device drivers make each storage device look like a uniform array of blocks, or a uniform stream of characters.
- The file system uses these to store flat files in a tree-structured directories.

Managing sharing of resources

- The scheduler shares processor time among all the concurrent processes, making sure each gets its share.
- The memory manager shares memory among all the processes, protecting each from reading or writing by the others.
- The file system shares disk blocks among all the files, and prevents one user from reading another user's file without permission.

1.3 Principles of programming 'in the large'

- Build large programs from small modules
- Make the interfaces simple
- Take care to document the interfaces precisely
- Isolate hardware dependencies in a few small modules
- Separate mechanism from policy

Example: Block device drivers

Floppy disks, hard disks, 'RAM disks' all have different physical characteristics, but we can use the same interface for all of them:

- Read block *n*
- Write block *n*

Advantages:

- The rest of the system is simpler, because device details are hidden in the device driver.
- Adding a new kind of device is easy.
- The same policy for organising a disk can be used whatever the mechanism for reading and writing.

Client-server model

Like many modern operating systems, MINIX itself is structured as a collection of concurrent processes:

- File system,
- Memory manager,
- Device drivers,

leaving a tiny *kernel* that passes messages between them.

This architecture is well suited to distributed operating systems. MINIX is a distributed operating system that runs on a single machine!

Summary

- Operating systems share the machine's resources among many client programs and users, providing each with a convenient extended machine.

- Operating systems have four main functions: process management, input/output, memory management, and a file system.

- Modern operating systems have a strong modular structure, based on separation of concerns like mechanism and policy.

Chapter 2: Concurrent processes

A well-designed operating system is constructed from modules with precise interfaces.

Often helpful to think of modules as *layers*, with each layer used to implement the ones on top.

Example: we'll use *device drivers* to give each kind of disk the same interface, then build the *file system* on top of it.

The first layer implements *concurrent processes*.

They are useful because

- Several user jobs can share the machine.
- Each job might be made up of several processes (e.g. pipelines, concurrent make).
- The operating system itself is naturally concurrent.

Implementing concurrency by time-sharing: the mechanism.

A table contains a *saved state* for each process.

When a process blocks or runs out of time

- Save the registers in its slot in the table
- Choose another process
- Load the registers for the new process from the table

Time-sharing was invented by Christopher Strachey.

Policy issues

- Which process should run next?
- How long should it run?

These policy decisions can depend on lots of process parameters:

- How much time has the process had recently?
- What priority has it been given?
- How big is it?
- Is it I/O bound or compute bound?

2.1 Synchronization and communication

There are lots of ways of providing safe communication between processes. We'll look at three:

- Messages (as used by MINIX)
- Semaphores
- Monitors and condition variables

Why bother with synchronization?

Two processes are trying to add a job to a queue for printing:

```
Process 0:                          Process 1:

i = next_slot;

                                    j = next_slot;
  <wait>                            queue[j] = job1;
                                    j++;
                                    next_slot = j;
queue[i] = job0;
i++;
next_slot = i;
```

This is a *race condition*. We need to protect the shared variable *next_slot*.

Using messages

Instead of having the two processes enter jobs in the queue directly, they send messages to a queue manager process:

```
Process 0:

message m0;
...
m0.m_type = JOB;
m0.m_job = job0;
send(MANAGER, &m0);
...
```

```
Process 1:

message m1;
...
m1.m_type = JOB;
m1.m_job = job1;
send(MANAGER, &m1);
...
```

Manager process

```
PUBLIC void manager_task()
{
     message m;

     while (TRUE) {
          receive(ANY, &m);

          switch (m.m_type) {
          case JOB:
               queue[next_slot] = m.m_job;
               next_slot++;
               break;

          ...
          }
     }
}
```

What do *send* and *receive* mean?

- *send(p, &m)* waits until the process p is ready to receive from the calling process, then transfers the message in the variable m.

- *receive(q, &m)* waits until process q (or any process, if q is ANY) is ready to send to the calling process, and stores the message received in variable m.

Send and *receive* are implemented as primitives, so can ensure safe communication.

Messages are C structures (= records), with fields

- *m.m_type* – the type of message, filled in by the sender.
- *m.m_source* – the sender, filled in by the Post Office.
- other fields according to the application.

Minix itself uses 24-byte messages in one of 6 formats: 3 ints and 3 pointers; 2 ints, a pointer and a short string, ...

Reasoning about message-passing

Messages remind us of CSP:

- *send(p, &m); S* is a bit like $p!m \rightarrow S$.
- *receive(q, &m); S* is a bit like $q?x \rightarrow S(x)$.

We can reason about networks of processes using process algebra, just like CSP.

[Though MINIX-style messages are labelled with sender and recipient, not with a channel].

Inside a process

We can use a 'ghost variable' *tr* to keep track of the *trace* of a process.

- *send(p, &m)* implements the specification

  $tr : [true, tr = tr_0 \frown \langle out(p, m) \rangle]$

- *receive(q, &m)* implements the specification

  $tr, m : [true, tr = tr_0 \frown \langle in(q, m) \rangle]$

The *receive* operation is non-deterministic, because the choice of message is made by the environment of the process.

Example: the print queue

Invariant:
$queue[0 \rightarrow next\_slot] = \langle\, m.m\_job \mid in(q, m) \leftarrow tr; m.m\_type = JOB \,\rangle$

This invariant says that *queue* contains all the jobs that have ever been received.

It remains true forever because it is maintained by the loop body.

But the loop never terminates!

[We ignore the possibility of jobs ever leaving the queue (!)].

Remote procedure call

A useful pattern is a *send* to a server process, followed by a *receive* from the same process – a *remote procedure call*.

- the client can be sure that the server had carried out the request.
- the server can send status information in the reply.

Used over a network, RPC is a good way of structuring a distributed operating system.

Example: print queue again

A client process communicates with the queue manager like this:

```
m.m_type = JOB;
m.m_job = job0;
send(MANAGER, &m);
receive(MANAGER, &m);
if (m.m_status != OK)
     printf("Please try later\n");
```

The queue manager now looks like this:

```
PUBLIC void manager_task()
{
     message m;
     int client;          /* Source of current request */
     int status;          /* Outcome of the request */

     while (TRUE) {
          receive(ANY, &m);
          client = m.m_source;
          status = OK;

          switch (m.m_type) {
               ...
```

The manager task continued

```
            case JOB:
                 if (next_slot >= MAX)
                      status = FULL;
                 else
                      queue[next_slot++] = m.m_job;
                 break;

            ...
            }

            m.m_type = REPLY;
            m.m_status = status;
            send(client, &m);
       }
}
```

RAM disk driver: a server process in MINIX

Pp. 479–80, lines 2289–2331.

The RAM disk driver loops forever:

- get a read or write request,
- transfer a block of information,
- send a reply.

Readers and writers: a classical IPC problem

A shared database is accessed by two classes of clients:

- *readers* may be allowed simultaneous access with each other.
- *writers* need exclusive access.

Each client surrounds its access to the database with calls to e.g. *start_read* and *end_read*:

```
...
start_read();
/* Read the database */
*/
end_read();
...
```

```
...
start_write();
/* Write on the database

end_write();
...
```

One implementation uses a process to manage access to the database.

Start_read and *end_read* look like this:

```
PUBLIC void start_read()
{
     message m;

     m.m_type = START_READ;
     send(MANAGER, &m);
     receive(MANAGER, &m);
     /* m.m_type == GRANT */
}
```

```
PUBLIC void end_read()
{
     message m;

     m.m_type = END_READ;
     send(MANAGER, &m);
}
```

The manager process keeps track of the number of readers and writers active, and keeps queues of waiting readers and writers:

```
int n_readers;          /* No. of active readers */
int n_writers;          /* No. of active writers (<= 1) */

queue read_q;           /* Processes waiting to read */
queue write_q;          /* Processes waiting to write */

init_q(&read_q);
init_q(&write_q);
```

It responds to *START_READ* and *END_READ* messages like this:

```
case START_READ:
     if (n_writers > 0)
          put_q(&read_q, client);
     else {
          n_readers++;
          send_grant(client);
     }
     break;

case END_READ:
     n_readers--;
     if (n_readers == 0 && ! empty_q(&write_q)) {
          n_writers++;
          client = get_q(&write_q);
          send_grant(client, &m);
     }
     break;
```

START_WRITE is left as an exercise; and *END_WRITE* messages are treated like this, giving readers priority over writers:

```
case END_WRITE:
     n_writers--;
     while (! empty_q(&read_q)) {
          n_readers++;
          client = get_q(&read_q);
          send_grant(client, &m);
     }
     if (n_readers == 0 && ! empty_q(&write_q)) {
          n_writers++;
          client = get_q(&write_q);
          send_grant(client, &m);
     }
     break;
```

Summary

- Concurrent processes are a useful service, and also help in implementing the rest of the OS.
- Undisciplined access to shared variables leads to chaos.
- Message-passing provides a secure and convenient IPC facility.

2.2 Semaphores (Dijkstra 1965)

An earlier and simpler synchronisation mechanism than message-passing.

Easier to implement, harder to use and reason about.

- A semaphore is a protected natural number variable.
- *up* operation increases value by one.
- *down* operation decreases value by one *but blocks if it is zero*.

Example: Print queue

The queue is protected by a semaphore `mutex`:

```
sema mutex = 1;
```

Client processes queue print jobs like this:

```
down(&mutex);
queue[next_slot++] = job;
up(&mutex);
```

The semaphore ensures that only one process may access the queue at once.

Implementing semaphores

The *down* and *up* operations must be atomic, or semaphores will suffer from race conditions of their own.

Make them atomic by disabling interrupts (including the clock) momentarily.

(This is better than disabling interrupts in larger sections of the operating system).

Each semaphore has:

- a current value (an integer).
- a queue of processes waiting for a *down*.

Remarks:

- the queue is empty unless the current value is zero.
- each process can be in the queue for at most one semaphore at once – so the queues can be linked lists with pointers stored in the process table.

Implementing *up* and *down*

For *up*:

- if processes are waiting, wake up the first one
- otherwise, increment the current value.

For *down*:

- if the current value is non-zero, decrement it
- otherwise, add the current process to the queue and go to sleep.

Snags with semaphores:

- they are difficult to use, because even simple patterns of synchronization need elaborate patterns of *down*'s and *up*'s.

- bugs are difficult to find, because they result in race conditions that are not reproducible.

- they are inefficient, because each *down* or *up* achieves so little.

Using semaphores to implement messages

Assume that the first argument of *receive* is always *ANY*, meaning "accept messages from any source".

Each process *p* has:

- a semaphore *listening[p]* (initially 0).
- a semaphore *ready[p]* (initially 0).
- a message buffer *mbuff[p]*.

We assume that variable *self* always contains the PID of the current process.

Implementing *send* and *receive*

```
PUBLIC void send(dst, m)
int dst;
message *m;
{
     down(listening[dst]);
     mbuff[dst] = *m;
     up(ready[dst]);
}
```

```
PUBLIC void receive(m)
message *m;
{
     up(listening[self]);
     down(ready[self]);
     *m = mbuff[self];
}
```

Correctness argument

- A sending process blocks at *down(listening[p])* until the receiving process is ready to communicate.
- Now the receiving process is blocked at *down(ready[self])*, and the sending process has exclusive access to the message buffer.
- Finally, the *up(ready[p])* from the sending process unblocks the receiving process, which now has exclusive access to the buffer.

Reasoning with semaphores: "this process waits *here* until that process reaches *there* . . ."

2.3 Monitors (Hoare, Brinch-Hansen 1974)

- a programming language feature (needs compiler support)
- like a Modula-2 module, but *only one process at once* may use its procedures
- local variables of the monitor are protected from simultaneous access by more than one process
- usually combined with *condition variables* for synchronization.
- emphasis is on protecting access to shared data structures.

Example: print queue again

```
monitor PrintQueue;

     var  queue: array [0 .. Max-1] of job;
          next_slot: integer;

     procedure Enter(j: job);
     begin
          queue[next_slot] := j;
          inc(next_slot);
     end

     ...

begin
     next_slot := 0;
end.
```

Condition variables

- *wait(s)* suspends the current process.
- *signal(s)* wakes one of the processes waiting for *s*.

Again, implementation must be atomic.

Monitors and condition variables can be implemented using semaphores – by having the compiler insert *down*'s and *up*'s in the right places.

Reasoning with monitors and condition variables

- Each monitor has an invariant I; each condition variable has a condition P.
- Each monitor procedure must maintain I.
- *wait*(x) implements the specification $[I, I \wedge P]$
- *signal*(x) implements the specification $[I \wedge P, I]$

These rules allow us to prove *safety* properties, but not *liveness*.

A safe print stack

```
monitor PrintStack;

     const MAX = 16;

     var  stack: array [0 .. MAX-1] of job;
          next_slot: integer;
          non_full, non_empty: condition;

     procedure Enter(j: job);
     begin
          if (next_slot = MAX) wait(nonfull);
          stack[next_slot] := j; inc(next_slot);
          signal(nonempty)
     end;
```

Safe print stack (cont'd)

```
     procedure Remove(var j: job);
     begin
          if (next_slot = 0) wait(nonempty);
          dec(next_slot); j := stack[next_slot];
          signal(nonfull);
     end;

begin
     next_slot := 0
end.
```

- A queue is just slightly more complicated!

Monitors vs. Server processes

A server process with a remote procedure call interface behaves like a monitor:

- The cycle: get request – do the work – send reply ensures only one client is active at once.
- But deferred requests have to be handled by explicit queuing of waiting clients.

2.4 Scheduling policy

A policy for CPU scheduling may have many goals:

1. Keeping the CPU as busy as possible.
2. Making sure each process gets a fair share.
3. Minimizing interactive response time.
4. Minimizing waiting time for batch jobs.
5. Maximizing throughput (no. of jobs per hour).

Obviously, trade-offs are unavoidable.

Also, the scheduler doesn't have perfect information.

Pre-emptive scheduling

- run a process only until it uses up its time quantum.
- almost essential for interactive time-sharing services.

Run to completion

- used in early batch systems (with or without multi-programming).
- still works well in embedded systems (no clock, predictable clients).

Round robin scheduling

Processes are run in rotation for one quantum each, or until they block.

- Implemented with a single queue of runnable processes. When you've had your turn, join the end of the queue again.

- Good performance depends on the right choice of quantum.

Priority classes

Processes are divided into classes according to priority.

High priority processes may be

- the ones that are I/O bound – since a small amount of CPU time allows them to keep busy with I/O.
- the ones that are interactive (a special case of the above).
- the ones for users that are paying more for CPU time.
- the ones that have not had much CPU time recently.

Whenever a process switch occurs, it is the waiting process with highest priority that is chosen.

Aging

Scheduling parameters are often estimated by averaging historical values, using a process known as *aging*.

If the sequence of observed values of a parameter are x_0, x_1, ..., then the averaged values are y_0, y_1, ..., given by

$$y_0 = x_0,$$
$$y_{i+1} = \alpha x_{i+1} + (1 - \alpha) y_i.$$

So if $\alpha = 1/2$, each new estimate is the simple average of the new datum and the old estimate.

Two level scheduling

Scheduling interacts with memory management, because processes that are *swapped out* cannot run.

So swapping systems commonly run the processes in memory for a while, pausing periodically to swap in the processes that have been swapped out for too long.

2.5 Processes in MINIX

MINIX divides processes into three priority classes:

1. Device drivers
2. The processes that implement the file system and memory management.
3. All user processes.

The process that is chosen to run is the one with highest priority.

Within each class, the scheduling is round-robin, with a quantum of 100 msec.

Process switching

To switch from one process to another, the whole CPU state is saved in the process slot for the old process, and restored from the slot for the new process:

- The general-purpose registers – so register contents don't change randomly.
- The segment registers – so virtual addresses in the process mean the right thing (see later).
- The program counter – so the process continues from where it left off.
- The processor status – so conditional branches work properly and (typically) so the process runs with the right permissions.

Look at `save` and `restore` in `kernel/mpx88.s`.

Organisation of MINIX source code

- *Common header files*. Define things like the numbers used for system call messages (line 0100), and the format of messages (line 0550).

  All parts of the O/S had better agree on these!

- *The kernel source*. This consists of the kernel itself (partly in assembler), and the device drivers (which run in kernel mode).

- *The memory manager*. Implements policy for memory allocation. Runs in user mode.

- *The file system*. Implements UNIX files using block device drivers. Runs in user mode.

A portable operating system

- Only layer 1 (the kernel) contains any assembly language code.
- Only layer 2 (the device drivers) depends on the I/O devices.
- The file system and memory manager run in user mode, contain hardly any machine-dependent code.
- Exactly the same file system and memory manager run on 68000 machines.

Kernel data structures

The kernel's process table (line 0756) contains for each process:

- Space to save registers.
- Addresses of the physical memory occupied by the process.
- A record of CPU time used by this process and its children.
- Links for organising a queue of processes waiting to send a message.
- Links for organising queues of processes that are ready to run.

There are three ready queues, one for device drivers, one for MM and FS, and one for user processes.

Scheduling policy

`pick_proc` (line 2086) chooses the most deserving process to run next.

It finds the highest priority queue that is non-empty (lines 2092–4), then sets `cur_proc` to the process at its head.

If no process is ready, a special process `IDLE` is chosen.

Send

`Mini_send` (line 1970) does the work of `send`.

- `Mini_send` checks that user processes don't try to communicate directly with device drivers (line 1987).
- Next, it checks that the message buffer lies inside the client's address space (lines 1994–8).
- If the destination is waiting to receive, it copies the message and wakes the destination (lines 2002–7).
- Otherwise, it makes the sender block, and hangs it on the queue for the receiver (lines 2010–23).

Receive

`Mini_rec` (line 2031) does the work of `receive`

- User processes only do `send` and `receive` together, so `mini_send` has already checked the arguments.
- If a sender is waiting, transfer the message and wake up the sender (lines 2051–2067).
- Otherwise, the process blocks (lines 2070–3).

Summary

- Semaphores and monitors are alternative synchronization mechanisms. Both are widely used; monitors suggest a style of programming for server processes.

- Each synchronization mechanism can be used to implement the others.

- Well designed implementations separate mechanism from policy, and isolate machine dependencies.

- Scheduling policy is about achieving contradictory goals given imperfect information.

Chapter 3: Input/Output

3.1 A little system architecture

The IBM PC is more than just an 8088 CPU:

- it has some *memory* (see under MM later).
- it has *I/O devices*: keyboard, screen, floppy disk, ...
- there are *device controllers* that link the I/O devices to the system bus.

How does a program control I/O devices?

By IN and OUT instructions.

- Controllers have *device registers* in a separate I/O address space.
- IN and OUT instructions read or write these registers.
- The controller translates the register contents into signals that move the actual I/O device.

(Cf. memory-mapped I/O).

Examples

- Controllers for the keyboard and system clock are on the motherboard.

- There's often a plug-in card that contains two controllers, one for floppy disks, the other for hard disks.

  As a compromise, each of these can control two devices.

- The video controller is another plug-in card. The screen contents are memory-mapped, but the cursor is controlled via a device register.

Interrupts

When an I/O operation completes, the controller can signal the processor by causing an interrupt.

- The controller raises a special line on the bus.
- The processor saves the current values of PSW, CS, and PC on the stack.
- The processor loads CS and PC from the *interrupt vector* for the device.
- The interrupt handler finishes the process switch to the kernel.
- The kernel sends a message to the appropriate device driver.
- The interrupt handler switches back to the previously running process (or maybe another one).

Levels of I/O software

You write a program to print "Hello world":

- the program calls `printf("Hello world\n");`
- the standard I/O library turns this into `write(1, buf, 12);`
- the system call library turns this into a message to FS with `m_type == WRITE` and the arguments in other fields.
- the FS turns this into a message to the TTY driver with `m_type == TTY_WRITE`
- the TTY driver copies the characters to the screen.

System calls

If each user process runs in its own (hopefully protected) address space, how can it call the O/S?

It uses a special INT instruction that simulates an interrupt.

- The user process is suspended, and the context switches to the kernel.
- The kernel sends a message to the appropriate server process (FS or MM).
- Eventually, the server process sends a reply.
- The kernel resumes the user process.

This is exactly remote procedure call.

Block and character devices

UNIX classifies devices into two kinds: *block devices* and *character devices*.

Block devices:

- Are read and written in blocks (e.g. of 1 KB).
- Allow random access.

Character devices:

- Read and write a stream of characters.
- Allow only serial access.

3.2 Disks: a block device

A typical hard disk drive has several platters, each with read/write heads for one or both surfaces.

The arms carrying the read/write heads are all moved by a single stepping motor.

- Moving the heads takes a long time ($\approx 30\,ms$).
- Reading a block without moving the heads is quicker ($\approx 8\,ms$).

So it's worth trying to read and write several blocks on the same *cylinder* at once.

Disk strategies

To get the best performance:

- at the device level: queue requests and service them in the best order.
- at the file system level: cache writes and send them to the device in a batch.
- at the file system level: allocate blocks for the same file close together.
- at the file system level: read ahead blocks that may be needed soon.

Blocks, sectors and zones

Some terminology:

- a *sector* is the smallest chunk that the hardware can read or write. On the IBM PC, 512 bytes.

- a *block* is the smallest chunk that the operating systems reads or writes as a unit. For MINIX, 1024 bytes. Advantage: fewer individual I/O requests.

- a *zone* is the unit of allocation in the file system. Advantages (of zone-size $>$ block-size): better locality, shorter disk addresses.

Disk head scheduling

First Come First Served. The simplest method (the one used in MINIX 1.1) is to have the device driver accept I/O requests one at a time, service them, and return the results.

We can do better by having the device driver accept more requests whilst it is waiting for the disk to do something.

Typical sequence of events:

- The device driver chooses a request and starts a seek.
- During the seek, more requests arrive, and are added to the queue.

The device driver needs a strategy for selecting the request to service next.

Shortest Seek First

Choose the request that is on the closest cylinder to where the arm is now.

- A good idea, because short seeks are quicker than long ones.
- Requests near the ends of the arm travel get poor service.
- No upper bound on the time to service a request.

Elevator algorithm

Make a sweep up the disk, then another sweep downwards. Choose the next request in the current direction of motion.

- Fairer than SSF.
- Any request served with total seek of at most twice the total number of cylinders.
- Slightly more seeking on average than SSF.

Plenty of scope to apply queueing theory here.

Error handling

Disks (especially floppy disks) are unreliable. They suffer from:

- *programming errors*. Other software (e.g. the FS) may request a non-existent sector.
- *transient read/write errors*. Caused by dust or power spikes: just try again, but not too often.
- *permanent read/write errors*. Caused by surface defects.
- *seek errors*. I asked for cylinder 6, but got cylinder 7. Caused by mechanical play in disk arm, etc.: recalibrate.
- *controller errors*. E.g. controller refuses to respond. Reset it.

Every disk operation has to be followed by careful checking that it worked properly.

Low-level caching

Another way to improve performance is to have the controller or the device driver read a whole track, even if only one block is needed.

- Some controllers do it in hardware, transparently to the software.
- Otherwise, the device driver can do it – but this defeats DMA.

This kind of geometry-dependent optimisation can be done at the device level, but is more difficult at the device-independent file system level.

3.3 Terminals: a character device

The terminal driver in Minix is 1200 lines of code: 30 times as much as the scheduler.

Why is this? And why don't we spend 30 times as long studying it?

Because the terminal driver has a lot to do. And because most of it is fairly tedious.

Keyboard input

The part that looks after the keyboard must:

- Handle keyboard interrupts and save the scan-codes
- Translate the scan codes to ASCII codes (keeping track of the state of the shift, CTRL and ALT keys)
- Deal with erase-character and erase-line keys
- Arrange for echoing
- Also work in *RAW* mode.

Screen output

The part that looks after the screen must:

- Put ordinary characters on the screen at the right place
- Parse escape sequences and carry out the commands
- Handle tabs, backspace and carriage return properly
- Scroll the screen.

The MINIX terminal driver

Accepts the following messages:

- *CHAR_INT* from the interrupt routine, when an input character has arrived, or an output character has been sent.
- *READ* to start a read request.
- *WRITE* for a write request.
- *CANCEL* if a suspended read request must be abandoned.
- *IOCTL* to set the baud rate, erase and kill characters, etc.

Input protocol

With (quick) block devices, it's not too bad to make the file system wait until a requested block is ready.

But with the terminal, this would be a disaster: a request for terminal input would bring everything to a stop until the user typed something.

Background processes should still run and do disk I/O even if a foreground process is waiting for keyboard input.

A typical story

1. A user process requests terminal input.
2. The file system asks the terminal driver.
3. The terminal driver has no characters ready, so it replies with *SUSPEND*.

Now other processes can run, and the FS can still do disk I/O

4. As characters arrive, the interrupt handler sends *CHAR_INT* messages to the driver.
5. When enough characters have arrived, the terminal driver copies them into the user's address space by kernel-mode magic.
6. The terminal driver replies to the file system.
7. The file system replies to the user process.

Memory-mapped screens

- Directly attached to the computer's bus.
- Writing to the screen just means storing the characters in the right memory locations.
- The video controller reads the characters from its memory and renders them on the screen.
- Escape sequences have to be interpreted by the driver.
- In graphics mode, the video memory holds pixels instead of characters.

Serial terminals

- Connected by a serial interface.
- Writing involves character-by-character transmission (at least 1 msec per character).
- With simple interfaces, one interrupt per character sent.
- Escape sequences are interpreted by the terminal.

Modern serial terminals are really small computers with a serial interface and a memory-mapped screen.

3.4 The clock: a special device

The clock is invisible as an I/O device to higher layers of the system. But it has a device driver anyway, that

- keeps track of the time of day.
- informs the scheduler when a quantum has expired.
- charges CPU time to the running process.
- provides *ALARM* signals for user processes.
- provides watchdog timers for other parts of the operating system.

Watchdog timers

Various sorts of alarm clocks are needed by other device drivers:

- A printer driver may want to time out if the printer has been taken off line.
- The floppy disk driver must wait 1/4 sec after starting the disk motor, and leaves it running for 4 sec after finishing.
- A hard-copy terminal may need a delay after CR.

Clock hardware

A crystal oscillator (frequency f) decrements a counter once every cycle.

When the counter reaches zero, an interrupt happens, and the counter is re-loaded from a device register.

Setting the device register to N gives f/N interrupts per second.

A simpler approach: an interrupt happens every mains cycle (50 or 60 Hz), not adjustable.

The MINIX clock driver

Accepts messages to

- Get the current time.
- Set the current time (used at system boot).
- Register for an alarm call.
- Do a clock tick (sent by interrupt routine).

Maintaining the current time

Each tick increments a counter – so it contains the number of ticks since system boot.

Another variable contains the UNIX time that the system was booted (in seconds since 12:00 AM on January 1st, 1970).

- *GET_TIME* and *SET_TIME* do some easy arithmetic.
- *CLOCK_TICK* only has to increment the counter; but it also takes care of 'lost ticks'.

Scheduling and billing

Ticks happen 60 times per second (even in Europe).

A quantum is 1/10 sec or 6 ticks.

- Every tick, the user or system time for the current user process is incremented.
- When 6 ticks have arrived, the clock driver calls *sched* to switch to a different user process.

Alarm calls

- Kernel tasks can send the clock driver a function to call after a certain time.

- User processes (via MM) can ask to receive a signal.

A sensible approach is to make a queue of processes sorted by alarm time. MINIX just keeps track of the next alarm time, and searches the process table for the right process to signal.

Summary

The aim of device drivers is to provide a simple interface to I/O devices by hiding:

- The physical nature of devices
- The protocol for using controllers
- Error handling
- Low-level scheduling and caching

Chapter 4: Memory management

Different operating systems have very different schemes for allocating memory to processes.

The simplest schemes keep all running processes in memory:

- One process gets all of memory (monoprogramming).
- Memory is divided into fixed-size partitions, one per process (some batch systems).
- When it starts, each process is allocated as much memory as it needs (MINIX).
- Processes may grow dynamically, with shuffling when one bumps into another.

These schemes work well in simple situations.

Swapping

Larger time-sharing systems may have more active processes than will fit in memory at once.

- *Swapping* systems keep some processes on disk.
- Only processes in memory can run.
- Periodically, processes are moved between memory and disk so everyone gets a chance.

Virtual memory (paging)

We may even want to run programs that are individually too big to fit in memory.

- *Virtual memory* systems keep only some parts of a running process in memory.
- Need hardware assistance in mapping virtual to physical addresses.
- Hope for hardware assistance in finding which pages are really being used.

Another British invention.

Mechanism and policy

The *mechanisms* of memory management are implemented

- in hardware: segmentation registers, memory protection, address translation hardware.
- at the kernel level: establishing segmentation registers and page tables.
- at the device driver level: providing disk I/O for swapping or paging.

But there are still plenty of policy decisions to make.

4.1 Swapping

Policy decisions for swapping systems:

- How much memory to allocate a process.
- How to allocate blocks of memory.
- Which processes to swap out when memory runs out.

Allocating memory

- With variable-size partitions that come and go, memory is split into allocated and free blocks of different sizes.
- To allocate a partition, we have to choose which free block to use. If the chosen block is too big, a smaller free block is left over.
- If no free block is available, perhaps *compacting memory* will help – but it takes time.
- When memory runs out, choose a victim and swap him out.

Data structures for mapping free memory

- Bit maps – each ‘click’ has a bit to say whether it is free or not. A bad idea: why?

- Free list – a linked list of free memory blocks. When a block becomes free, merge it with its neighbours.

- Buddy system – all blocks are a power of two clicks. Smaller blocks are made by splitting larger ones; a small block is merged only with its ‘buddy’. Fast but wasteful.

Allocation policies

- *Best fit* chooses the smallest block that is big enough. But it tends to leave lots of little fragments.

- *First fit* chooses the first block it finds that is big enough. This is faster, and gives less fragmentation.

- *Worst fit* is not a very good idea.

Analysis of the free list

The *fifty-percent rule* says that (if the memory is in equilibrium) there are about half as many holes as there are processes.

Proof: let N be the number of processes, M the number of holes. We can divide the processes into three categories:

A those between two holes: freeing them decreases M by one, because three blocks coallesce.

B those at one end or the other of a run of adjacent processes: freeing them leaves M unchanged.

C those between two other processes: freeing them increases M by one.

Arithmetic gives us $N = A + B + C$ and $M = \frac{1}{2}(2A + B + \epsilon)$ – each free block has two ends; ϵ is 0, 1 or 2, depending on what happens at the ends of memory.

Equilibrium gives us that when a block is allocated or freed, Prob(M increases by 1) = Prob(M decreases by 1). This leads to $C/N = A/N$, neglecting the possibility of an exact fit.

So $M \approx \frac{1}{2}N$.

This fact can give us a relationship between average process and hole sizes and the fraction of memory in holes. But most other properties of the allocation scheme have to be found by simulation.

In single-user non-swapping systems like MINIX, allocation is almost stack-like, so the allocation policy doesn't matter very much.

Choosing a process to swap

A sensible swapping policy prefers to swap out:

- Processes that will not run soon: ones waiting for another to finish, ones waiting for an alarm call.

- Processes that have been in core for longest.

It swaps in processes that are ready to run, and have been swapped out for a long time.

Swapping and sanity

To prevent an embarrassing catastrophe:

- Don't swap out the kernel or device drivers.
- Don't swap out any process doing direct I/O by DMA.
- Don't swap out the swapping process.

4.2 Virtual memory

What makes this program:

```
for (j = 0; j < N; j++)
     for (i = 0; i < N; i++)
          c[i][j] = a[i][j] + b[i][j];
```

1000 times slower than this one:

```
for (i = 0; i < N; i++)
     for (j = 0; j < N; j++)
          c[i][j] = a[i][j] + b[i][j];
```

on the same machine?

Memory layout

Pages:

Rows of a:

A col of a:

With the arrays laid out like this

- Accessing each element of a row touches only a few pages, so working row-by-row causes only a few thousand page faults (one for each page).

- Accessing each element of a column touches 1000 pages. So working column-by-column causes a million page faults.

For small arrays, this wouldn't matter, because all the pages would fit in memory at once. But for big arrays, the difference is crucial.

Well-behaved programs

- Exhibit good *locality of reference*.
- Have a small *working set*.

The *working set* of a process is the set of pages it is using "at the moment".

Implementing Orwell

Two approaches to garbage collection:

- Link all the free cells into a free list, randomly distributed through the address space. When free space runs out, find the inaccessible cells and link them into the free list.

- Use only half the allocated memory, with allocated cells at the bottom and free cells at the top. When free space runs out, stop and copy the accessible cells into the other half of memory. Swap halves and continue.

The first option has dreadful locality of reference. With careful design of the copying algorithm, the second option actually improves locality as it goes.

Address translation

With virtual memory, not all the pages of a process are in main memory, and the ones that are in main memory are in random locations.

Hardware assistance (in the form of a *page table*) is needed to translate virtual addresses (in the address space of the process) to physical addresses (in main memory).

Each process has its own page table, because address 1234 in one process should refer to a different location from address 1234 in another. The page table of (at least) the running process should be held in special fast memory.

Sophisticated hardware allows the page tables to be split into segments that can be shared, and so on.

Address translation formula

If the page size is P, the virtual address v is split into the virtual page no. $p = \lfloor v/P \rfloor$ and the offset $x = v \bmod P$.

The page table is used to find the physical page $p' = f(p)$, provided page p is present in memory.

The offset is combined with the physical page no. to get the physical address $a = p' \times P + x$.

If page p is not present, a page fault is signalled.

Example

Hardware parameters

- the *virtual address space* is the size of the addresses used inside processes. It limits the maximum size of a process. Typically 24 bits (= 16 MB).

- the *page size* is the smallest unit of memory allocation. Typically 4KB. The page table contains one slot for each page in the virtual address space: 4096 in our example.

- the *phyical address space* is the size of physical addresses after translation. It limits the maximum amount of real memory that can be attached to the processor.

Physical address space can be larger or smaller than virtual address space.

Page replacement policies

When a page fault occurs, we must choose a page to evict from main memory to make room for a new one. What policy do we use?

Optimal: evict the page that will not be used for the longest time. Unimplementable!

Random: choose a page at random to evict.

FIFO: evict the page that has been in memory the longest.

LRU: evict the page that has been unused for longest.

NRU: choose a page that has not been used during the last clock tick.

NFU: use aging to estimate how frequently a page has been used in the recent past.

Monotonicity

A well-behaved page replacement algorithm is *monotonic*:

- increasing the number of pages allocated to a process will decrease the number of page faults

FIFO, for example is not monotonic, but LRU is.

Proof for LRU: the pages in memory at any time are the N pages that have been used most recently. Increasing N adds more pages, but doesn't remove any: so if a page fault happens with more pages, it would also have happened with fewer.

Implementing LRU

With hardware assistance: have a special clock that ticks every memory reference. Store in each page table slot the time it was last used.

Without hardware assistance: can't do it. But can implement approximations like NRU and NFU.

To find out which pages are used during a system clock period, remove them from the page table at the start of the period. When a page fault happens, mark the page as used, and fill in the page table with the proper location of the page.

(The same method can be used to find out which pages are dirty.)

Design issues for virtual memory

- Replacement policy.
- Local *vs*. global allocation.
- Demand paging *vs*. prepaging.
- Choice of page size. Bigger pages waste more space, but give fewer page faults and need a smaller page table.

Implementation issues

- A page fault can happen in the middle of an instruction that occupies several words and changes several registers. The page fault handler has to undo the changes made by the instruction so far. Stupid hardware can make this impossible.

- *Locking pages in memory*: the O/S kernel and pages involved in I/O (and the page fault handler) must stay in memory.

- *Shared pages*: if several processes are running the same program (and it doesn't modify itself), then they can share the code and save memory and page faults.

4.3 Memory management in Minix

- Each process has a single, fixed block of memory
- Processes never grow or shrink
- Unix system calls to grow memory are just ignored

Free memory is allocated from a *hole table*.

A little processor architecture ...

The (original) IBM PC contains an 8088 microprocessor.

- Physical address space of 20 bits (= 1MB).
- Virtual address space is 16 bits (= 64KB).
- ... so *segment registers* are used to locate programs in memory.

Big DOS programs are always fiddling with the segment registers.

MINIX programs may not change the segment registers, and are limited to 64K code + 64K data.

On a proper machine

- segment registers can only be altered in *kernel mode*, and user processes run in *user mode*.
- each segment of memory has a base and a limit.
- This gives each process its own address space,
- and protects the memory of each process from reading and writing by other processes.

The 8088 doesn't have limit registers, and doesn't have user mode. So we'll just have to pretend!

Memory layout

Summary

Much variety in memory management:

- Simple in-core schemes
- Swapping
- Virtual memory

But all implement a simple virtual machine on (fairly) simple hardware

Chapter 5: File systems

The device driver layer provides large and small arrays of storage for fixed-size blocks.

But users and applications programs want a different view of file storage:

- Variable-sized files bigger or smaller than a block ...
- stored under file-names, not by number ...
- organised into hierarchical directories ...
- with protection against unauthorised reading or writing.

5.1 Files and directories

The UNIX view: A file is a file is a file.

The contents of each file is a uniform sequence of bytes.

Any higher-level organisation (into lines, records, indexes) is the job of user-level software.

Advantages: the same utility programs can be used on every sort of file, more or less. (Though text editors don't work very well on files of object code).

Other views of files

Other operating systems (especially on mainframes) may provide built-in indexing mechanisms like ISAM that assist with random access. *Advantages*: the indexing operations can be made faster and don't have to be replicated in every user-level program.

Old systems that used to be based on punched cards sometimes have a type of file divided into 80-byte records. *Advantages*: none, really.

Device independence

All operating systems aim at device independence: it doesn't make any difference whether your files are stored on hard disk, floppies, RAM disk, ...

Some are more successful than others: with MS-DOS, programs can work with files on any medium.

```
A:program <B:input >B:output
```

But you have to know where the files are.

UNIX lets you organise all your storage devices into a single tree.

Directories

Users don't want to remember the block numbers for their files.

Instead, (most) operating systems provide some kind of directory, a mapping from names to files.

Each file has a sequence of blocks that store its contents; also some other information like the actual length of the file, the date of creation, etc.

The simplest directory system (e.g. CP/M) keeps a single table that maps file names to their block numbers and other data.

Hierarchical directories

For large disks, a single directory is inconvenient.

Better is a system of *hierarchical* directories, organized into a tree. (Why not a graph?) MS-DOS copied this idea from UNIX, which copied it from MULTICS.

But UNIX has something even better: the directory tree is just an index for the files, which exist separately. A (single copy of a) file can appear at several places in the tree.

For example, you can make a directory that gathers together the executable files for all your utility programs, without having to make extra copies of them all.

Files, i-nodes and directories

Each UNIX file on a device has a unique i-number, an index into a table of i-nodes.

The i-node for a file contains its block numbers, its true length, its owner and date of creation, its protection bits ...

Directories are just files of (name, i-number) pairs. You don't need to know the i-numbers of your files, but you do need to know (e.g.) that changing the protection of a file in one place will change it in all other places too.

Protection

Users want to protect their files against

- Other people changing them.
- Other people reading the secret ones.
- Themselves overwriting valuable ones in moments of stupidity.

Ideally, there would be a matrix that for each person and file gave the operations (read, write, delete) that the person could use on the file.

In practice, a more restrictive scheme is used for simplicity and to save space. UNIX associates with each file a string of 9 bits that say what certain classes of people can do.

5.3 File system implementation

The user view of a file is a uniform sequence of bytes.

The disk provides a (fairly uniform) array of blocks.

How do we represent one with the other?

- Allocate each file a contiguous sequence of enough blocks. *Advantages*: fast access to the whole file, directories need just base and length.

- Allocate scattered blocks for the file. *Advantages*: files can grow without being moved, free space fragmentation doesn't matter.

How big should a block be?

- most files are fairly small (typical median size = 1K), so big blocks will waste a lot of space.

- each scattered segment of a file needs a seek to read it, so small blocks will waste a lot of time.

There's a fundamental conflict here. A practical compromise is a block size of 512 bytes – 2KB.

Keeping track of free space

How do we keep track of free blocks on the disk?

- A linked list that chains together all free blocks?
- A linked list of blocks, each containing the numbers of 511 free blocks?
- A bit map?

Organizing the blocks of a file

The linked-list method: the directory entry contains the number of the file's first block. Each block contains 510 bytes of data and the number of the next block.

The CP/M method: each directory entry directly contains the numbers of the file's blocks. If a file is more than 16 blocks, it has several directory entries.

The MS-DOS method: the directory entry contains the number of the first block, and subsequent blocks are found using the disk's *file allocation table*. For each block, the FAT gives the number of the next block in the file.

The UNIX method

- The i-node contains the numbers of a file's first 10 blocks.
- If that's not enough, the i-node points to a *single indirect block* that contains the numbers of 256 more blocks.
- If that's not enough, the i-node points to a *double indirect block* that contains the numbers of 256 more single indirect blocks.
- If that's not enough, there's a triple indirect block.

Differences between the methods

The methods differ most in their ability to support random access:

- An in-core FAT requires no disk accesses to find any block of a file.
- But a FAT held on disk may need as many disk accesses as there are FAT blocks.
- The UNIX scheme requires at most three disk accesses to find any block.

The UNIX file system

System calls:

- `fd = open("/usr/mike/mbox", O_READ)`

  opens a file for reading or writing, returning a small integer called a *file descriptor*.

- `nread = read(fd, buf, n)`

  reads up to *n* bytes from the file attached to *fd* into the character array *buf*.

- `close(fd)`

  closes the file, making the file descriptor unattached.

→ Why bother separating *open* and *read*?

Data structures

- The file descriptor table (for each process) maps file descriptors to *file pointers*.
- The file pointer table maps file pointers to *i–numbers* and offsets.
- The i–node table maps i–numbers to *i–nodes* that contain the block numbers.

→ Why so many levels of indirection?

→ Why not let user-level software keep the file pointers directly?

Shared file pointers

Suppose you write a shell script `foo` like this:

```
echo "Directory listing"
date
echo
ls -C
```

This produces a directory listing with a nasty title.

Now we say `sh foo >listing`. → What should happen?

→ What happens if two processes independently read the same file?

Finding a file

To open the file `/usr/mike/mbox`, the file system:

- Gets the root i–node (always number 1).
- Searches the blocks of the root directory for an entry with name `usr` and finds the i–number for the directory `/usr`.
- Gets the i–node for `/usr`
- Searches that directory for the name `mike`, and finds the i–number for directory `/usr/mike`
- Gets the i–node for `/usr/mike`
- Searches that directory for the name `mbox`, and finds the i–number for the file.
- Gets the i–node for `/usr/mike/mbox`

Design considerations

→ The i–node for the file is kept in memory during the time that the file is open. Why?

Opening this file requires about 6 disk references.

→ What could make it more?

→ What could make it less?

→ How does the system find the file `../tools/editor`?

Why is the file system so big?

What could go wrong with the system call that makes a new link to a file?

`link("usr/mike/mbox", "gpo/mbox")`

- The arguments may not point inside the user's address space.
- The arguments may be too long.
- The old path `/usr/mike/mbox` may not exist.
- The calling user may not have permission to search the old path.
- The file may already have 127 links to it.
- The file may be a directory (& only the super-user may make links to it).

More errors ...

- The new directory gpo may not exist.
- The calling user may not have permission to write on the new directory.
- The new directory and the file may be on different devices.
- A new block may need to be added to the new directory, but the device may be full.

None of these errors may crash the system.

Some of the errors can be detected only after some of the work has been done. If so, must undo what has been done before reporting the error.

Performance issues

- The bigger the block cache, the better!
- The same replacement policies apply as for virtual memory, but no hardware assistance is needed. Heuristics can be added to throw away blocks that won't be needed again soon, and to write critical blocks more quickly.
- The update process ensures that the disk is updated every few seconds.
- Read ahead to exploit sequential access to files.

5.4 Reliability

- Bad blocks: must not be used for storing data.
- Back-ups.
- Consistency checking.

File system consistency

Crashes may leave the system in a state where:

- Blocks are not marked as free, but are not in any file.
- Blocks are both part of a file and marked as free.
- Blocks are in more than one file.
- I–nodes with non-zero link count are marked as free
- I–nodes with zero link-count are not marked as free
- Link-counts do not accurately reflect the number of links.
- A directory appears more than once in the tree.

Consistency checking

A program that traverses the whole directory tree can check for this kind of inconsistency, and can fix most of them.

→ How can the errors occur?

→ Some are more serious than others. Which?

→ How does the block cache make things worse? How could it be the danger be minimized, and how much would it cost?

5.5 File servers and transactions

In a distributed operating system, the filing service may be provided by a server machine. The split between machines may be in several places:

- Remote disks put the file system on the client machine, but the disk and device driver on the server. Sun's *ND* protocol is like this.

- The file server may provide a 'flat' filing service where files are identified by numbers chosen by the server. The client machine (or the user) must remember the numbers, or the files are lost. PRG's distributed operating system was like this.

- The server may offer a complete UNIX-like file system. Remote file systems may be mounted on the file tree of a workstation, as in Sun's *NFS*.

Atomicity

Especially with distributed systems, it's important that an update to a file either completes successfully or (if the system crashes) doesn't happen at all.

This is called *atomic update*.

One way to implement it keeps two copies of the disk. Each block write is performed first on one drive, then on the other.

Writes that fail part way through result in checksum errors, so the other copy of the block can be used.

Locking

Also more important with distributed systems is mutually exclusive access to files or records.

Example: two simultaneous deposits to a bank account

Teller 1	Teller2
Read old balance ($500)	
	Read old balance ($500)
Add $200 to get $700	Add $300 to get $800
Write new balance ($700)	
	Write new balance ($800)

Implementing locking

The file server keeps in memory a list of locks on files or records.

There's a system call to lock a file or record; if it's already locked, the calling process is blocked or the call is rejected.

Attempts to read or write files that are not locked are rejected.

Another system call unlocks the file.

Problems:

- Deadlock when two processes need locks on the same files.
- Processes that acquire a lock and crash before releasing it.

Transactions

Atomic updates and locking are often combined to support transactions.

Transactions are effectively atomic updates with automatic locking.

Much research into providing transactions in a secure way, but without too much ineffeciency.

5.6 Security and protection

Security =

- protection against loss +
- protection against disclosure

There are lots of social issues too profound to discuss here.

General issues

- Security isn't a modular add-on feature – because a bug anywhere in the system can cause a security breach.

- Having priveleged programs or users spreads the risk even further. (Example: early version of `lpr` daemon).

- Even subtle bugs in priveleged programs can cause problems. (Example: `mkdir` does *mknod* followed by *chown*).

- Trojan horse attack threatens systems with a super-user.

Design principles

- Public design: no secret algorithms.
- No access should be the default.
- Frequent checks of authority.
- Minimum privelege for each process.
- Protection system must be simple, uniform, universal.

Authentication

How do we tell that a person is who they claim to be?

1001 methods are discussed in Tanenbaum's book; some are disgusting.

UNIX uses passwords with encryption: both encryption algorithm and encrypted passwords are public.

Protection mechanisms

General model:

- a *domain* determines a binary relation between *objects* and *rights*. Think of a domain $\times$ object matrix of lists of rights.
- the *protection state* is a mapping from processes to domains.
- domains themselves may be objects, with a right to *enter* the domain.

Access control lists

Cut up the domain $\times$ object matrix into columns: each object has a list of domains and their rights over the object.

- UNIX is a special case: domains are (uid, gid) pairs, and there are three rights called r, w, x. Each object (file) has a uid u and a gid g and nine bits that show the rights for uid u for gid g and for others.

- More general access control lists might use a disk block to store them. Multics uses a list of patterns; Sun UNIX uses a similar scheme for remote login rights (`.rhosts`).

Capabilities

Slice the matrix into rows called capability lists: each domain is a collection of 'capabilities'.

- capabilities are objects that can be passed between processes.
- forgery must be prevented: by encryption, by special hardware, by having the OS manage a table.
- especially useful in distributed systems. Server processes get more rights than their users by *rights amplification*.

Interlude A: About practical 1

Implement the software for a fancy digital thermometer.

- Build a network of processes that communicate by passing messages.
- Arrange for mutually exclusive access to shared resources.

Provided: a package that implements MINIX-style message passing *inside* an ordinary UNIX process.

C program structure

A program is built from several modules or *files*. Each file can have some private and some public functions and data.

Header files describe the interfaces between modules. A module will include lines like

```
#include "parts.h"
```

to insert a header file into the module. Like `#define` macros, this is textual substitution.

Typically, header files contain `typedef`'s, declarations of `extern` variables, and function declarations (which don't specify arguments).

The three great mistakes

There are three mistakes you *will* make in writing C programs. You will spend hours trying to find them.

- writing = when you meant ==.
- forgetting the `break;` between arms of a switch.
- calling a function with the wrong number of arguments.

The *lint* program can help to find these mistakes.

Creating processes

```
PRIVATE int keyboard, update, display;

PUBLIC int main()
{
     keyboard = co_call("keyboard", do_keyboard);
     update   = co_call("update",   do_update);
     display  = co_call("display",  do_display);
     co_start();
     printf("\n\n");           /* tidy up the display */
     return (0);
}
```

- `co_call` sets up (but doesn't start) a process returning an integer – its PID. Save the PID for use later.

- `co_start` starts and runs all the processes

Messages

```
/* Message format */
typedef struct {
     int m_source;                  /* PID of the sender */
     int m_type;                    /* Type of message   */
     union {                        /* Contents:         */
          int m__int;               /*   a number,       */
          char m__char;             /*   a character,    */
          char m__string[10];       /*   or a string.    */
     } m_u;
} message;
```

- A structure – like a record in Modula-2
- The data is a union – like a record, but only one component at a time.

Abbreviated selectors

These definitions:

```
#define m_int     m_u.m__int
#define m_char    m_u.m__char
#define m_string  m_u.m__string
```

let you write e.g.

```
m.m_string
```

instead of

```
m.m_u.m__string
```

They work by *textual substitution* in the C pre-processor.

Defining your own message types

Use `#define` to give symbolic names to integer constants:

```
#define ADJUST  1
#define SWITCH  2
#define GETTEMP 3
#define TEMP    4
#define INIT    5
```

then write, e.g.,

```
m.m_type = ADJUST;
m.m_int = -1
send(update, &m);
```

Reading the temperature

```
PUBLIC void start_adc(chan)
int chan;
```

- Starts an A-to-D converter to read the temperature on channel *chan*.
- Some time later, the hardware sends a message of type *TEMPVAL* containing the temperature.
- Uses (simulated) hardware that can't do two things at once.
- It's your job to ensure mutual exclusion.

Make

Make is a program that manages (re-) compilation and linking of programs.

- You make a file called `Makefile` that describes how to compile and link your program.
- When you're ready, just say `make`, and the *make* program works out what needs doing.
- If you change something, *make* looks at timestamps to work out what needs rebuilding.

Makefile for the practical

```
SRC = /usr/local/opsys/prac1
CFLAGS = -I$(SRC)

thermo: thermo.o
        $(CC) $(CFLAGS) -o thermo thermo.o \
                $(SRC)/threads.o $(SRC)/parts.o -lm
```

- Macros written `$(NAME)`
- Implicit rule takes `file.c` to `file.o`
- You don't need to worry about the details!