5 Lecture: Operating System Structures

Outline:

- Announcements
- Onwards
- The Process Model: a little deeper
- Pseudoparallelism and nondeterminism
- Possible process states
- Scheduling
- What about IPC?
- Operating System Structures
  - Monolithic Systems
  - Layered Systems
  - Virtual Machines (VM/370, vmware, bochs)
  - Client-server model
- The Layered Architecture of Minix
- Example: Description a MINIX disk interrupt

5.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn2	LWP	Mon	Jan 26	23:59	
asgn3	dine	Wed	Feb 4	23:59	
lab03	problem set	Mon	Feb 9	23:59	
midterm	stuff	Wed	Feb 11		
lab04	scavenger hunt II	Wed	Feb 18	23:59	
asgn4	/dev/secret	Wed	Feb 25	23:59	
lab05	problem set	Mon	Mar 9	23:59	
asgn5	minget and minls	Wed	Mar 11	23:59	
asgn6	Yes, really	Fri	Mar 13	23:59	
final (sec01)		Fri	Mar 20	10:10	
final (sec03)		Fri	Mar 20	13:10	

  Use your own discretion with respect to timing/due dates.

- ~pnico/longlines.pl!
- why minix 3.1.8
- lab grading scheme
- Lab02 due next wednesday (why not?)
- QEMU says it supports M1

5.2 Onwards

Last time we talked about context switching and the concept of a Process. Now that we have the general idea, let's look at some general structures.

This week we are going to look at general OS architectures, then move into scheduling.

5.3 The Process Model: a little deeper

Now, we need to talk about some details of how it's actually done and the issues involved.

The **process** is the most important concept in understanding operating systems.

The operating system provides **multiprogramming**:

pseudoparallelism time-sliced parallelism on a uniprocessor

(yield?, pre-empt?)

true parallelism provided by a multiprocessor

Keeping track of parallel activities is difficult, so OS designers have developed the model of the **sequential process**:

Each process (**running program**) has its own:

- IO Access: open file descriptors
- Register file
- Memory
  - Text (code) segment
  - Data (bss and heap) segment
  - Stack segment
- Program Counter

5.4 Pseudoparallelism and nondeterminism

(Ir?)regular (random) context switches mean programs can make no assumptions about:

- order of execution relative to other programs
- timing (example: a clock, or game timing)

(If timing *is* important, you need to get a Real-Time O.S.)

5.5 Possible process states

Not all processes can run at the same time:

At any given time, any process can be in one of three possible states: Running, Ready, or Blocked. Possible transitions between these states are shown in Figure 7.

In more detail, a process can be:

Running The process is loaded in memory and is currently executing on the cpu.

Ready The process is runnable, but another process has the cpu.

Blocked The process is waiting for some external event to enable it to run. E.g. completion of some IO event, or an alarm, or availability of more memory.

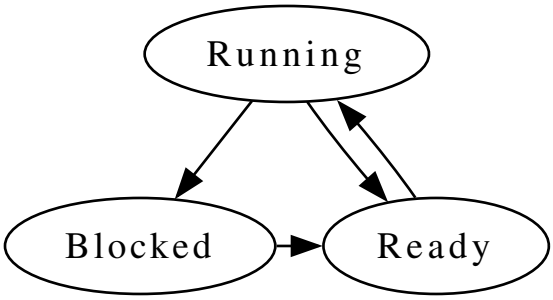

Figure 7: Possible states for a process

5.6 Scheduling

Transitions among the states of Figure 7 are made by the **scheduler**. The scheduler hides the details of starting and stopping processes:

- handles interrupts and dispatches them to the appropriate process
- chooses a process to run from among runnable processes
- Maintains the **process table** that contains
  - program counter
  - stack pointer
  - memory allocation information
  - status
  - open file descriptors
  - other stuff that must be saved (more record keeping: execution times, signal mask, etc.)

  Minix divides its process table into separate ones for scheduling, memory, and file information, but the principle is the same.

This leads to the model of the operating system shown in Figure 8.

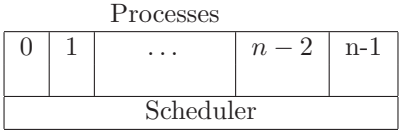

Figure 8: The lowest layer of the operating system

5.7 What about IPC?

Now we have separated processes so they don't know about each other, how do they communicate? IPC must be provided by the O.S.

5.8 Operating System Structures

Now that we've looked at the "external" view of an operating system, how would one be built?

> *Guts are so inelegant*
> —Patrick Beard

Operating systems have to do a lot of things:

- Scheduling
- Interprocess Communication
- Memory Management
- Filesystem Management
- Device Management
- Network Management
- *Lots of other things...*

How to put one of these together?

5.8.1 Monolithic Systems

Provide a set of system services through a single interface(a trap handler (Supervisor Call)):

- The system is one big object
- no information hiding
- requires programmer discipline; difficult to maintain
  Can be organized well, but be careful...
- Structure:
  - a main() program invoked by the trap
  - a set of service routines that carry out system calls
  - a set of utility routines

  Dispatched through a call table.. E.g. `open(name,flags,mode)`

```
movl   $0xXXXXXXXX,%ebx   ; String pointer (filename)
movl   $0x2,%ecx           ; flags (O_RDONLY)
movl   $0x0,%edx           ; mode (0)
movl   $0xb,%eax           ; syscall code for open()
int    $0x80               ; do the call
```

5.8.2 Layered Systems

Generalizes the approach above, separating tasks.

First exhibited in the `THE` operating system.

Technische Hogdschool[University] Eindhoven in the Netherlands, by Dijkstra in 1968

The machine: Electrologica X8, with 32K of 27-bit words (400kHz).

The OS Six layers:

Layer	Function
5	The Operator
4	User Programs
3	IO Management (abstract IO devices)
2	Operator-process communication (individual operator consoles)
1	Memory and drum (512K words) management
0	Processor allocation and multiprogramming

Level 3 is above level 2 because the devices might need to speak to the operator in case of malfunction.

Multics did this with concentric rings.

How did THE enforce these layers? It didn't, but it was a good design tool.

5.8.3 Virtual Machines

(VM/370, vmware, bochs)

Go all the way: If we're going to virtualize this machine, let's do it:

- VM/370: Virtual Machine Monitor
- Different users can run different OSs
  - CMS: Conversational Monitor System (interactive)
  - OS/360: batch processing
  - vm/370 in itself...
- Used by windows to run MS-DOS programs. (full emulation vs. partial emulation)
- VMare does the same thing completely (as do bochs and qemu)

5.8.4 Client-server model

Mach/Minix

Pull it all out of the kernel. (Well, not *all*)

servers and clients communicate via message passing.

The (now) *micro*-kernel does only things really requiring privilege and delegates to user-level processes.

- Separates **mechanism** from **policy**
- message-passing
- modifiable
- could be remote(?)

e.g.: Mach

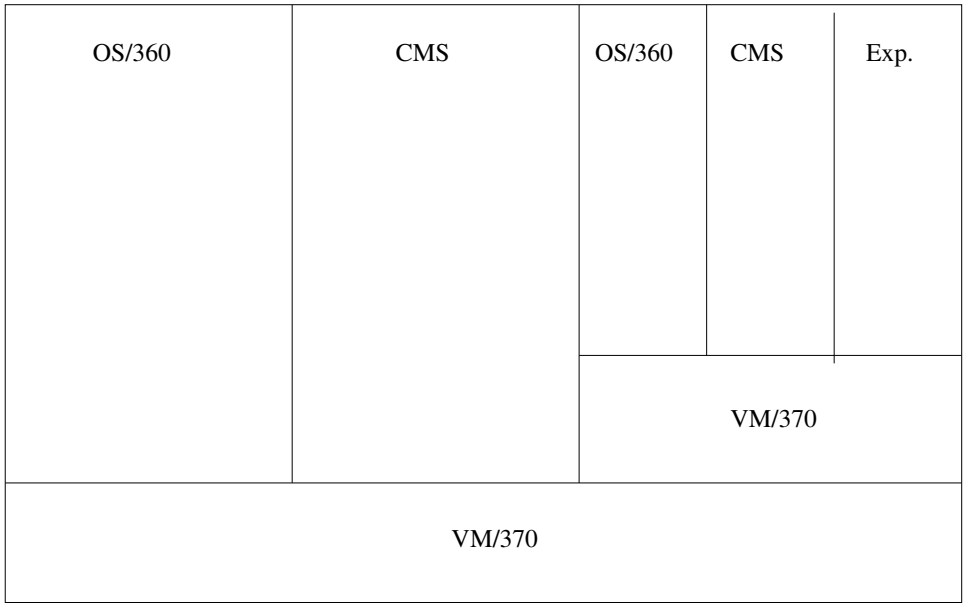

Figure 9: Structure of VM/370

5.9 The Layered Architecture of Minix

The structure of minix is as in Figure 10.

Layer 1 Sheduling, handling traps and interrupts, facilitating message passing.

The bottom is written in assembly(must be), the rest is C.

Layer 2 IO Processes, one per device type. Compiled into the kernel, but have separate identities, (and, if supported, privilege levels)

Layer 3 Server processes: File System, Memory Manager, Network Server, etc. These are user-level processes that implement the system calls.

These run with higher priority than user processes and never terminate while the system is running.

Layer 4 User processes

5.10 Example: Description a MINIX disk interrupt

- Structure of Minix
- How a disk read happens in Minix

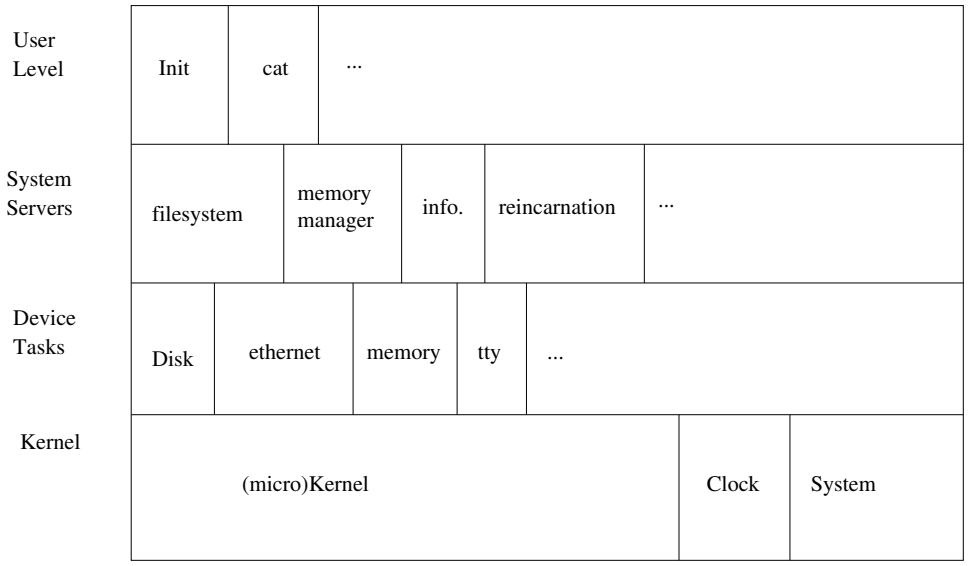

Figure 10: Layout of the Minix system