Operating Systems

The program that runs most of the time

The user/system interface

The reason you need a new computer every 18 months :-)

Operating System Features

- Process Management
- Loader
- System Calls
- Exception Handling
- Concurrency Control
- Memory Management
- File Systems
- Disk Scheduling
- Networking

Process Management

Program is a passive entity, stored on disk

Process is active, a program in execution

- associates a program counter with program
- creates process control block

Can be in 3 states:

- ready, running, waiting

Multiple processes execute concurrently

some user, some system

Context switch **between processes**

Process Scheduling

Batch jobs use shortest job first

minimizes average response time

Interactive Jobs

- use round robin policy
- needs a time slice or *quantum*
- add priorities (multiple queues)

Relevant Unix commands

- `ps`, `top`, `renice`
- `kill -STOP`, `kill -CONT`, `kill -KILL`

Shells and Loading

Shell is a user process owned by you

When you run a program shell calls `fork`

- child calls `exec` to run program
- returns exit code to parent
- parent calls `wait` for child to exit

OS preloads first few pages into memory

- others on demand (*demand paging*)

Creates new page table with mappings

System Calls

How you *do* all the *cool* stuff

Example MIPS `syscall` functions:

- `fork`, `read`, `write`, `open`, `close`
- `create`, `chdir`, `mount`
- `send`, `recv`

Pass arguments in registers

Enters kernel mode, returns to user mode

What Happens on a `printf`?

`printf` is a user-level *library call*

lives in libc, automaticaly linked

Creates the final string to print

Calls the `write` `syscall`

- transition to kernel mode
- uncached writes to the memory-mapped console
- characters appear on the screen

`syscall` puts return value in r2

Exception Handling

Using only reserved kernel registers, save state of running process

- save EPC (and possibly branch PC)
- save register file to process control block
- dispatch specific handler based on cause

Can you take an exception in an exception handler?

- sometimes, OS must be careful
- *maskable interrupts*: lower priority
- *non-maskable interrupts*

Concurrency Control

OS manages shared resources

e.g. access to a printer, shared file

OS has many *critical sections*

Protect critical sections with:

- locks (mutexes)
- semaphores
- monitors

Avoid *deadlock*

- acquire resources in same order
- e.g. Dining Philosophers

Memory Management

Support page-based virtual memory systems

- protection, relocation, resource sharing
- tlb refill code
- page fault handler

Keeps data structure for page replacement

Page replacement algorithms

- optimal is unrealizable
- *clock algorithm*, using R and M bits

Manage swap space

File Systems

Root directory kept in a fixed place

- Unix: *inodes* scattered throughout
- directories hold inode number/name
- inodes hold size/time/permission and 10 disk block numbers
- big files: pointers to other inodes
- huge files: pointers to pointers, and pointers to pointers to pointers

DOS/Windows: FAT—File Allocation Table

- entry for every disk block
- entry holds next block of file or EOF

Disk Scheduling

Problem: How to optimally schedule disk requests to maximize throughput?

Alternatives

- shortest seek first
- minimize motion of r/w head
- *elevator algorithm*

File caching

- keep recently accessed file data in memory
- problem: what do you do with writes?

Networking

OS implements the TCP and IP layers of the network protocol

Gives applications a virtualized socket-based API to the network

Can open sockets via `syscall`

- returns a file descriptor
- read from and write to descriptor

Servers `listen` on sockets and `accept` connections

- server may `fork` or handle request itself