G22.3250 – Honors Operating Systems

David Mazières
715 Broadway, #708
dm@cs.nyu.edu

Administrivia

- **All assignments are on the web page**
  `http://www.scs.cs.nyu.edu/G22.3250/`
- **Part of each class will be spent discussing papers**
  - Read the papers before class
- **Grading based on four factors**
  - Participation in discussion (so read the papers before class!)
  - Midterm and Final Quiz
  - Lab assignments
  - Final project

Handouts today

- **Account information form**
  - Will give you access to dedicated class machines for lab
  - Accounts will be created Friday
  - Email me if you don't hear from me by Friday
- **Access form for 7th floor of 715 Broadway**
  - So you can come to my office hours
  - Only if you don't already have access (PhD students do)
- **Using TCP through sockets (on web page)**
- **First lab goes on-line Friday (on web page)**

Course topics I

- **Core operating systems**
  - User/kernel APIs & performance issues
  - Concurrency—threads & async programming
  - Virtual memory
  - Scheduling
  - Implementing network protocol stacks
  - High-performance device and driver issues
  - File systems
  - I/O abstractions & Kernel design

Course topics II

- **Distributed systems topics**
  - Distributed shared memory
  - Distributed file systems
  - Network objects
  - Scalability
  - Replication & consistency
  - Cryptography and security
  - Peer-to-peer systems
- **Most contemporary OS work focuses on distributed systems**
  - Labs will stress distributed programming

What is an operating system?

- **Makes hardware useful to the programmer**
- **Provides abstractions for applications**
  - Manages and hides details of hardware
  - Accesses hardware through low/level interfaces unavailable to applications
- **Provides protection**
  - Prevents one process/user from clobbering another

What is a distributed operating system?

- **The holy grail: Transparency**
  - Have a bunch of machines look just like one machine
  - As easy to manage as one machine
  - Save applications & users from worrying about it
  - Just add more machines to scale to higher workloads
- **The reality: Numerous complications**
  - Failures, especially partial & network failures
  - Concurrency
  - Long latencies
  - Security issues

Successful distributed system architectures

- **Client/server architecture**
  - Clients request services from servers with network messages
  - Modular architecture, isolates servers from client faults
  - Can potentially scale by adding more servers
- **Peer-to-peer (decentralized)**
  - Ad hoc configuration can survive loss of any machine
  - Potential scalability problems (e.g., can’t broadcast)
- **Single name or address space**

Why Operating Systems?

- **Operating systems are a maturing field**
  - Most people use a handful of mature OSes
  - Hard to get people to switch operating systems
  - Hard to have impact with a new OS
- **High-performancs servers are an OS issue!**
  - Need to manage hardware resources, often at low level
  - Much server software faces the same issues as OSes
  - OS abstractions often even interfere with servers
  - Big open problem: OSes don't support flexibility needs of high-performance servers

Example: A video server

- **Hardware capabilities**
  - 20 MByte/sec SCSI disk
  - 100 Mbit/sec Ethernet
- **Server requirements**
  - 200 Kbit/sec video streams
  - Many users spread around the Internet
  - Access control
- **Maximum capacity: 500 clients**

The reality: Much lower capacity

- **CPU bottleneck**
  - Software structure may impose many context switches
  - Concurrency may introduce lock contention
- **Disk I/O limitations**
  - Multiple video streams can introduce disk seeks: 5ms seek per 8K read $\rightarrow$ 1.6 MByte/sec
  - Must pipeline disk requests: prefetching
  - Must deal with OS buffer cache (may fill memory and cause paging)
- **Network complications**
  - OS may buffer stale data (dropped frames)
  - Introduces latency to congestion feedback (received packets not prioritized)

Concurrency

- **Goal: Maximize throughput**
  - Service the maximum number of clients over time
- **Benefit: Overlap latencies**
  - Dedicate CPU time to other clients during network transmission/client computation
  - Present disk with simultaneous requests
    $\rightarrow$ achieve better disk arm scheduling
  - Amortize interrupts over multiple packets
- **Dangers: Reducing throughput with overload**
  - Introducing context switches
  - Increasing cache misses
  - Increased memory/buffer cache usage $\rightarrow$ Paging / thrashing
- **Two basic approaches: Threads & Asynchronous I/O**

Threads

- **Write sequential-looking code:**

```
for (;;) {
  read_from_disk;
  write_to_network;
  wait_until_next_frame_needed:
}
```

- **Run multiple instances of code in parallel**
  - While one thread paused/waiting for I/O, schedule another
  - Protect shared data by locks
- **Benefit: threaded code can exploit multiprocessor**

Limitations of threads

- **High memory overhead**
  - Need one stack per thread
- **High context switch overhead for kernel threads**
  - But user threads suck, too (no multiprocessing)
- **Lock contention can kill performance**
  - Even uncontested synchronization operations expensive
  - Coarse-grained locking kills concurrency
  - Fine-graned locking costs CPU time
  - Preemption may happen at inopportune moments
    → priority inversion
- **Brutally hard to program!**

Why thread programming is hard

- **Data races**
- **Deadlock**
- **Threads break abstraction**
  - Must worry about what locks modules assume & aquire

$$T1 \implies \text{Module A} \implies \text{Module B} \implies \text{wait}$$
$$T2 \implies \text{Module A} \implies \text{Module B} \implies \text{signal} \quad \text{Deadlock!}$$

  - Breaks callbacks

$$T1 \implies \text{Module A} \implies \text{Module B} \implies \text{Module A} \quad \text{Deadlock!}$$

- **Hard to debug**
  - Non-determinism based on internal scheduler

Asynchronous I/O

- **I/O operations never block**
  - e.g., if no data, read immediately returns error
- **Single blocking operation: select/poll**
  - Returns list of I/O operations that are ready
- **Event driven architecture**
  - Maintain list of callbacks awaiting I/O events
  - Main dispatch loop makes callbacks when event happens

```
struct callback {
  void (*cb) (void *);   void *arg;
};
main () {
  initialize_callbacks;
  foreach (pending I/O) { run_callback; }
}
```

Benefits of Asynchronous programming

- **Low overhead**
  - Callback typically much smaller than thread stack
  - No context switch overhead (just a procedure call)
- **Implicit coordination**
  - No data races
  - No deadlock
  - No priority inversion

Limitations of Asynchronous programming

- **Cannot have long-running callbacks**
- **Not automatically scalable to multiprocessor**
- **Hard to program**
  - Must explicitly package up state across callbacks
    Cannot share stack-allocated state
  - Lots of dynamic memory allocation
    (who is resposible for freeing what?)
  - Logical flow of events broken into many event handlers

Other issues for high-performance servers:

- **Coordination & scheduling**
- **Disk allocation & scheduling**
- **Memory management (including buffer cache)**
- **Address spaces (VM)**
- **Distributed system abstractions**
- **Efficient data movement**
  - Kernel effectively a data mover
  - IPC, memory → network, disk → network, network → network, etc.

Example: Coordination

- **Interrupts are expensive (microseconds)**
  - Under heavy load, can spend all time servicing interrupts
  - Receiver livelock occurs when more packets arrive than can be processed
- **Polling**
  - Amortize one driver invocation over many packets
  - Adds latency (unreasonable under low loads)
  - Fits naturally into asynchronous I/O model
- **Solution: switch dynamically between interrupts & polling**

Scaling to multiple CPUs

- **Multiprocessors help if user-level CPU bottleneck**
  - Might hurt system time, though
  - Non-linear cost vs. speedup
- **Server clusters**
  - Inexpensive if scalable
- **Distributed server clusters**
  - When client-server bandwidth is low

Clusters

- **Naming transparency**
  - Should client be aware of cluster?
- **Server selection**
- **Consistency**
  - Multiple servers must agree on state of things
- **Availability**
  - Chances of one node failing increase
  - Replication helps availability, complicates consistency

Distributed clusters

- **Many issues:**
  - Replication policies
  - Efficient data distribution
  - Consistency
  - Network monitoring and modeling
  - Global load-balancing
- **Rethink traditional OS abstractions**
  - File system semantics, etc.
  - Trade-off between accuracy, latency, and network load

Summary

- **High performance servers an OS issue**
  - Pipelining disk & network requests
  - Coordination
  - Caching
- **True scalability requires distributed system**
  - Reliability/Availability
  - Security
  - Consistency
  - Tolerating latency
- **Difficult**
  - If a fast server bypasses OS abstractions, how does this affect other applications?

System calls

- **Problem: How to access resources other than CPU**
  - Disk, network, terminal, other processes
  - CPU prohibits instructions that would access devices
  - Only privileged OS "kernel" can access devices
- **Applications request I/O operations from kernel**
- **Kernel supplies well-defined *system call* interface**
  - Applications set up syscall arguments and *trap* to kernel
  - Kernel performs operation and returns result
- **Higher-level functions built on syscall interface**
  - `printf`, `scanf`, `gets`, etc. all user-level code

I/O through the file system

- **Applications "open" files/devices by name**
  - I/O happens through open files
- `int open(char *path, int flags, ...);`
  - `flags`: `O_RDONLY`, `O_WRONLY`, `O_RDWR`
  - `O_CREAT`: create the file if non-existent
  - `O_EXCL`: (w. `O_CREAT`) create if file exists already
  - `O_TRUNC`: Truncate the file
  - `O_APPEND`: Start writing from end of file
  - `mode`: final argument with `O_CREAT`
- **Returns file descriptor—used for all I/O to file**

Error returns

- **What if `open` fails? Returns -1 (invalid fd)**
- **Most system calls return -1 on failure**
  - Specific kind of error in global int `errno`
- `#include <sys/errno.h>` **for possible values**
  - 2 = `ENOENT` "No such file or directory"
  - 13 = `EACCES` "Permission Denied"
- `perror`, `strerror` **print human-readable messages**
  - `perror ("initfile");`
  - `printf ("initfile: %s\n", strerror (errno));`
    → "`initfile: No such file or directory`"

Operations on file descriptors

- `int read (int fd, void *buf, int nbytes);`
  - Returns number of bytes read
  - Returns 0 bytes at end of file, or -1 on error
- `int write (int fd, void *buf, int nbytes);`
  - Returns number of bytes read, -1 on error
- `off_t lseek (int fd, off_t pos, int whence);`
  - `whence`: 0 – start, 1 – current, 2 – end
    - Returns previous file offset, or -1 on error
- `int close (int fd);`
- `int fsync (int fd);`
  - Guarantee that file contents is stably on disk

Other system calls on pathnames

- `int chdir (const char *dir);`
  - Change working directory (what `cd` command does)
- `int mkdir (const char *dir);`
- `int rmdir (const char *dir);`
  - Make and remove direcories
- `int unlink (const char *path);`
  - Delete pathname specified by path
- `int link (const char *p1, const char *p1);`
  - Creates `p2`; `p1` & `p2` identical directory entries
- `int symlink (const char *p1, const char *p2);`
  - Creates `p2`; `p2` is an *alias* for name `p1`

The rename system call

- `int rename (const char *p1, const char *p2);`
  - Changes name `p2` to reference file `p1`
  - Removes file name `p1`
- **Guarantees that `p2` will exist despite any crashes**
  - `p2` may still be old file
  - `p1` and `p2` may both be new file
  - but `p2` will always be old or new file
- `fsync`/`rename` **idiom used extensively**
  - E.g., emacs: Writes file `.#file#`
  - Calls `fsync` on file descriptor
  - `rename (".#file#", "file");`

File descriptor numbers

- **File descriptors are inherited by processes**
  - When one process spawns another, same fds by default
- **Descriptors 0, 1, and 2 have special meaning**
  - 0 – "standard input" (`stdin` in ANSI C)
  - 1 – "standard output" (`stdout, printf` in ANSI C)
  - 2 – "standard error" (`stderr, perror` in ANSI C)
  - Normally all three attached to terminal

Manipulating file descriptors

- `int dup2 (int oldfd, int newfd);`
  - Closes `newfd`, if it was a valid descriptor
  - Makes `newfd` an exact copy of `oldfd`
  - Two file descriptors will share same offset (`lseek` on one will affect both)
- `int fcntl (int fd, F_SETFD, int val)`
  - Sets *close on exec* flag if `val` = 1, clears if `val` = 0
  - Makes file descriptor non-inheritable by spawned programs

Pipes

- `int pipe (int fds[2]);`
  - Returns two file descriptors in `fds[0]` and `fds[1]`
  - Writes to `fds[1]` will be read on `fds[0]`
  - When last copy of `fds[1]` closed, `fds[0]` will return EOF
  - Returns 0 on success, -1 on error
- **Operations on pipes**
  - `read`/`write`/`close` – as with files
  - When `fds[1]` closed, `read(fds[0])` returns 0 bytes
  - When `fds[0]` closed, `write(fds[1])`:
    - Kills process with `SIGPIPE`, or if blocked
    - Fails with EPIPE

Sockets: Communication between machines

- **Datagram sockets: Unreliable message delivery**
  - On Internet: User Datagram Protocol (UDP)
  - Send atomic messages, which may be reordered or lost
  - Special system calls to read/write: `send`/`recv`
- **Stream sockets: Bi-directional pipes**
  - On Internet: Transmission Control Protocol (TCP)
  - Bytes written on one end read on the other
  - Reads may not return full amount requested—must re-read

Socket naming

- **Every Internet host has a unique 32-bit *IP address***
  - Often written in "dotted-quad" notation: 204.168.181.201
  - DNS protocol maps names (www.nyu.edu) to IP addresses
  - Network routes packets based on IP address
- **16-bit *port number* demultiplexes TCP traffic**
  - Well-known services "listen" on standard ports: finger—79, HTTP—80, mail—25, ssh—22
  - Clients connect from arbitrary ports to well known ports
  - A connection consists of five components: Protocol (TCP), local IP, local port, remote IP, remote port
- **All Internet traffic routed as small packets**
  - Each packet contains address information in header

System calls for using TCP

Client	Server
	`socket` – make socket
	`bind` – assign address
	`listen` – listen for clients
`socket` – make socket	
`bind` – assign address	
`connect` – connect to listening socket	
	`accept` – accept connection

Example client

```
struct sockaddr_in {
        short   sin_family;  /* = AF_INET */
        u_short sin_port;    /* = htons (PORT) */
        struct  in_addr sin_addr;
        char    sin_zero[8];
} sin;

int s = socket (AF_INET, SOCK_STREAM, 0);
bzero (&sin, sizeof (sin));
sin.sin_family = AF_INET;
sin.sin_port = htons (13);  /* daytime port */
sin.sin_addr.s_addr = htonl (IP_ADDRESS);
connect (s, (sockaddr *) &sin, sizeof (sin));
while ((n = read (s, buf, sizeof (buf))) > 0)
  write (1, buf, n);
```

Example server

```
struct sockaddr_in sin;
int s = socket (AF_INET, SOCK_STREAM, 0);
bzero (&sin, sizeof (sin));
sin.sin_family = AF_INET;
sin.sin_port = htons (9999);
sin.sin_addr.s_addr = htonl (INADDR_ANY);
bind (s, (sockaddr *) &sin, sizeof (sin));
listen (s, 5);

for (;;) {
  socklen_t len = sizeof (sin);
  int cfd = accept (s, (sockaddr *) &sin, &len);
  /* do something with cfd */
  close (cfd);
}
```

Concurrent connections

- **Servers must handle multiple clients concurrently**
  - Read or write of a socket connected to slow client can block
  - Overlap network latency with CPU, transmission, disk I/O
  - Keep disk queues full when server accesses disk
- **Can use one process per client: accept, fork, close**
  - High overhead, cannot share state between clients
- **Can use threads for concurrency**
  - Data races and deadlock make programming tricky
  - Must allocate one stack per request
- **Use non-blocking read/write calls**
  - Unusual programming model

Non-blocking I/O

- `fcntl` **sets** `O_NONBLOCK` **flag on descriptor**

```
int n;
if ((n = fcntl (s, F_GETFL)) >= 0)
  fcntl (s, F_SETFL, n | O_NONBLOCK);
```

- **Non-blocking semantics of system calls:**
  - `read` immediately returns -1 with `errno` `EAGAIN` if no data
  - `write` may not write all data, or may return `EAGAIN`
  - `connect` may “fail” with `EINPROGRESS` (or may succeed, or may fail with real error like `ECONNREFUSED`)
  - `accept` may fail with `EAGAIN` if no pending connections

How do you know when to read/write?

```
struct timeval {
  long    tv_sec;         /* seconds */
  long    tv_usec;        /* and microseconds */
};

int select (int nfds, fd_set *readfds, fd_set *writefds,
            fd_set *exceptfds, struct timeval *timeout);
FD_SET(fd, &fdset);
FD_CLR(fd, &fdset);
FD_ISSET(fd, &fdset);
FD_ZERO(&fdset);
```

Asynchronous programming model

- **Many non-blocking file descriptors in one process**
  - Wait for pending I/O events on file many descriptors
  - Each event triggers some *callback* function
- **Lab:** `libasync` **– supports event-driven model**
  - Register callbacks on file descriptors
  - Call `amain()` – main select loop
  - Add/delete callbacks from within callbacks