CS240 – Advanced Topics in Operating Systems

Instructor: David Mazières
CA: Ryan Stutsman

Stanford University
Administrivia

- **Class web page:** [http://cs240.scs.stanford.edu/](http://cs240.scs.stanford.edu/)
- All handouts and lecture notes on-line
  - Please print them out yourselves
- **Part of each class will be spent discussing papers**
  - Print, read the papers before class
- **Slides and lecture notes will be on-line**
- **Grading based on two factors**
  - Midterm and final (bring all papers): ~60%
  - Participation in discussion: ~40%
Exams

- Both midterm and final open-book
  - Bring (marked up) print-outs of all papers
  - Bring any lecture notes and slides you wish
  - Don’t bring your laptop (open book, not laptop)

- Goal is to test your understanding of papers
  - Not a memory of vocab quiz
  - But definitely won’t have time to read papers during exam!

- Grade is based on $\max\left((\text{midterm} + \text{final}) / 2, \text{final}\right)$
  - Score will be used to rank students for grading
Class participation

- Class participation an important component of grade
  - Grading curve based mostly on participation
- If you need to miss class please email staff
  - Otherwise will hurt your final grade
- Read the papers before class!
- I need to learn who you are
  - End class early today to take photos
  - Please write your name in large letters on back of handout to hold up for photo
  - Make sure name takes up entire sheet of paper
Class structure

• Some classes start w. mini-lecture on background
  - Slides will be posted on web site

• Discussion notes will also be posted
  - I need to make notes to myself to lead discussion
  - Will post these on-line after class in ASCII format
  - Primarily intended for me, but will remind you of what happened during class
  - Should reduce the need to take notes for those who prefer not to

• Will sometimes cite papers w. first author in brackets
  - Look up with search engine or ask staff for more info

• If you have questions on papers you read, email us by 12pm on day of lecture
  - I will try to address your questions in discussion
  - Good questions may earn participation points
More administrivia

- **Class email list**: cs240@scs.stanford.edu
  - Should reach all students
  - Let us know if you don’t get mail test mail tonight (or if you want to subscribe under a different address)
  - Feel free to discuss papers on the list

- **Staff email list**: cs240-staff@scs.stanford.edu
  - Please mail staff list rather than Instructor or CA (class mail gets much higher priority in my mail queue)
Prerequisites & Goal

• We assume some familiarity with C
• Some familiarity with Unix may help
• An undergraduate “textbook” OS class (e.g., CS140)
  - Familiar with concepts like Virtual Memory, processes, etc.
  - Enough to read papers that employ these concepts
• Goals of class:
  - Get to the point of being able to read and understand contemporary OS papers
  - Provide background if you are interested in doing OS research
Course topics

- Concurrency & synchronization
- Scheduling
- Virtual memory
- Virtual machines
- File systems & storage
- Network stack implementation
- Distributed storage systems
- Kernel architectures
- OS Security and reliability
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
Why study 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-performance servers are an OS issue
  - Face many of the same issues as OSes

- Resource consumption is an OS issue
  - Battery life, radio spectrum, etc.

- Security is an OS issue
  - Hard to achieve security without a solid foundation

- New “smart” devices need new OSes
**Typical OS structure**

- Most software runs as user-level processes
- OS kernel handles “privileged” operations
  - Creating/deleting processes
  - Access to hardware
Unix architecture

- User-level
- Kernel “top half”
  - System call, page fault handler, kernel-only process, etc.
- Software interrupt
- Device interrupt
- Timer interrupt (hardclock)
- Context switch code
Transitions between contexts

- User $\rightarrow$ top half:
- User/top half $\rightarrow$ device/timer interrupt:
- Top half $\rightarrow$ user/context switch:
- Top half $\rightarrow$ context switch:
- Context switch $\rightarrow$ user/top half
Transitions between contexts

- User → top half: syscall, page fault
- User/top half → device/timer interrupt: hardware
- Top half → user/context switch: return
- Top half → context switch: sleep
- Context switch → user/top half
System calls

- **Goal:** invoke kernel from user-level code
  - Like a library call, but into more privileged OS code
- **Applications request 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
Example: POSIX/Unix interface

- Applications “open” files/devices by name
  - I/O happens through open files
- int open(char *path, int flags, ...);
  - Returns file descriptor—used for all I/O to file
- int read (int fd, void *buf, int nbytes);
- int write (int fd, void *buf, int nbytes);
- int close (int fd);
Example 2: Exokernel [Engler]

- Only one kernel context
  - Single kernel stack; interrupts disabled & never sleep in kernel
- bc_read_insert (dev, offset, len, u_int *res);
  - Schedule DMA from disk to new physical page
- insert_pte (capability, page-translation, VA, capability, target-process);
  - Sets a particular page table entry for process
- wkpred (wk_term *pred, int size);
  - Put process to sleep until particular condition is met
  - Condition is described in “wake predicate” language
  - pred is a program in this little language

- Can also implement printf, etc., with these
- Why would you want to?
System Interfaces

- System call interface is interface to kernel
- Historically also interface “most programmers” need
  - But most programmers ≠ all programmers
  - Standard OSes thwart many types of application
- No inherent reason for two interfaces to be the same
  - Exokernel possibly most extreme example
  - Philosophy: provide protection in kernel, but abstraction only in user-level libraries
What is OS research?

- **New kernel architectures**
  - Typically <10% of papers in top OS conferences!

- **New kernel components**
  - E.g., new VM or network stack implementation, process or disk scheduling algorithm, buffer cache, file system, etc.
  - Also only a small fraction of papers

- **User-level code running on an existing kernel**
  - Challenging applications often face OS issues
  - Either hit your head against OS which does the wrong thing
  - Or have to re-implement OS-like abstractions (for better performance, distributed operation, etc.)
Example: Video Server

• **Buy hardware with the following capabilities:**
  - 40 MByte/sec SCSI disk
  - 100 Mbit/sec ethernet

• **Application requirements:**
  - Serve 200 Kbit/sec video streams
  - Many users spread around the Internet
  - Access control

• **Maximum capacity: 500 clients??**
Reality: Much lower capacity

- **CPU bottleneck**
  - Software structure may impose many context switches
  - Concurrency may introduce lock contention

- **Disk I/O limitations**
  - Multiple streams introduce disk seeks:
    E.g., 5ms/8K read = 1.6 MByte/sec
  - Must pipeline disk requests—requires prefetching
  - But then OS buffer cache may fill memory and induce paging

- **Network stack limitations**
  - OS may buffer stale data (dropped frames)
  - Congestion control improperly prioritizes packets
Maximizing throughput

• Goal: Service maximum number of clients over time
• Avoid letting resources stay idle
  - Different resources: CPU, disk, network
  - E.g., don’t leave disk idle while waiting for network
• Key technique: Concurrency
  - Concurrency ensures each resource can be utilized
• Example: while waiting for a disk read for one client
  - …can use CPU to compute data for another client
  - …can transmit data to a third client over network
Disks

• For many systems, disks are the limiting factor
• Can limit throughput far below network capacity
  - E.g., cut naïve video server throughput by factor of 7–8
• Can also limit capacity of a system
• Example: build a system to index CS240 lecture notes
  - Scan all files, create index file
  - Rely on kernel for low-level details of storage management
• Example: build a system to index the web
  - Can’t stuff enough disks in one computer to hold index
  - Need a user-level distributed storage system
Anatomy of a disk

- **Stack of magnetic platters**
  - Rotate together on a central spindle @3,600-15,000 RPM
  - Drive speed drifts slowly over time
  - Can’t predict rotational position after 100-200 revolutions

- **Disk arm assembly**
  - Arms rotate around pivot, all move together
  - Pivot offers some resistance to linear shocks
  - Arms contain disk heads—one for each recording surface
  - Heads read and write data to platters
Disk positioning system

- Move head to specific track and keep it there
  - Resist physical socks, imperfect tracks, etc.

- A seek consists of up to four phases:
  - speedup—accelerate arm to max speed or half way point
  - coast—at max speed (for long seeks)
  - slowdown—stops arm near destination
  - settle—adjusts head to actual desired track

- Very short seeks dominated by settle time (∼1 ms)
- Short (200-400 cyl.) seeks dominated by speedup
  - Accelerations of 40g
Seek details

• Head switches comparable to short seeks
  - May also require head adjustment
  - Settles take longer for writes than reads

• Disk keeps table of pivot motor power
  - Maps seek distance to power and time
  - Disk interpolates over entries in table
  - Table set by periodic “thermal recalibration”
  - 500 ms recalibration every 25 min, bad for AV

• “Average seek time” quoted can be many things
  - Time to seek 1/3 disk, 1/3 time to seek whole disk,
What does this mean for systems?

- More concurrency can mean higher throughput
  - If you have $n$ requests, you can sort them
  - Ask disk/controller to perform them in optimal order
  - Can vastly reduce seek times, increase throughput

- Cache data you will use again in faster memory

- More efficient to access nearby data than far
  - E.g., good idea to put related data close together
    (e.g., put file near metadata)

- Larger requests mean higher throughput
  - Sequential throughput much higher than random

- Note: Last three may require bypassing the OS
Preview of disk techniques

- Make many disks seem like one large disk (RAID)
- Structure storage for large writes (Logs)
- Expose information/control to applications
  - What pages are in virtual memory
  - What pages have been accessed
  - How should OS manage cache
  - What threads are blocked waiting for disk I/O
Network

- Many machines serve clients over the Internet
  - Speed of light means typical RTTs in 10s of milliseconds
- Often have to wait for client
  - E.g., send back HTML page, wait for image requests
  - Congestion control may require limiting send rate
- Again, concurrency achieves good throughput
- Also may care about space
  - E.g., if client takes one second to service and requires 10 MByte of memory, memory will be limiting factor
- Many kernels impose other limitations:
  - Number of file descriptors or processes
  - Size of hash table for TCP connections (mostly fixed today)
Achieving concurrency

- Can use OS processes
  - Heavy weight (expensive to create) and memory intensive

- Can use non-blocking I/O operations
  - Read/write, but return immediately if not possible
  - Single process handles many clients
  - Not good for disk concurrency

- Can use threads implemented at user level
  - Build on non-blocking I/O primitives

- Can use kernel-level threads
  - But more heavy weight than user-level threads

- More on threads in next few lectures…
# System calls for using TCP

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>socket – make socket</td>
<td>socket – make socket</td>
</tr>
<tr>
<td>bind – assign address (optional)</td>
<td>bind – assign address</td>
</tr>
<tr>
<td>connect – connect to listening socket</td>
<td>listen – listen for clients</td>
</tr>
<tr>
<td>write – send data</td>
<td>accept – accept connection</td>
</tr>
<tr>
<td>read – receive data</td>
<td>read – receive data</td>
</tr>
<tr>
<td></td>
<td>write – send data</td>
</tr>
</tbody>
</table>

- Anything **red** might block, waiting for network
  - Obviously bad for applications that need concurrency
Non-blocking I/O

- **Use `fcntl` to set O_NONBLOCK flag on descriptor**
- **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 such as ECONNREFUSED)
  - accept may fail with EAGAIN if no pending connections
How to know when to read/write?

struct pollfd {
    int fd;       /* file descriptor */
    short events; /* Events you are interested in */
    short revents; /* Events that have happened (results) */
};

int poll(struct pollfd *fds, nfds_t nfds, int timeout);

/* Some possible events: */
#define POLLIN 0x0001 /* Can read fd without blocking */
#define POLLOUT 0x0004 /* Can write fd without blocking */
#define POLLERR 0x0008 /* Error on fd (only in revents) */
#define POLLHUP 0x0010 /* ‘‘Hangup’’ has occurred on fd */

- **Note: BSD used select to achieve same thing**
  - Most OSes support both select and poll today
epoll

- **Newer Linux provides** `epoll`
- **Interface allows more efficient implementation**
  - Register interest with `epoll_create, epoll_ctl` syscalls
  - Wait with `epoll_wait` syscall
  - Kernel doesn’t have to re-scan `pollfd` array on each wait
- **New option bits reduce calls to** `epoll_ctl`
  - `EPOLLONESHOT` – only wait for event once
  - `EPOLLET` – “edge triggered” (as opposed to level triggered)
- `epoll` is **Linux specific**
  - But BSD has `kqueue/kevent` which is similar idea
typedef union epoll_data {
    int fd;
    /* ... */
} epoll_data_t;

struct epoll_event {
    __uint32_t events; /* Epoll events */
    epoll_data_t data; /* User data variable */
};

int epoll_create(int size);
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
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

• E.g., build “callback harness”:

```c
/* Register callback for when fd is readable or writable */
void cb_add (int fd, int write, void (*fn)(void *), void *arg);

/* Unregister callback */
void cb_free (int fd, int write);

/* Loop forever checking callbacks */
void cb_check (void);
```

• Often called event-based programming
Simplified example

```c
struct state {
    int fd;
    /* ... */
};

void doit (void) {
    struct state *st = malloc (sizeof (*st));
    st->fd = create_new_tcp_socket ();
    connect (st->fd, &someplace, sizeof (someplace));
    cb_add (st->fd, 1, doit_2, st);
}

static void doit_2 (void *_st) {
    struct state *st = _st;
    write (st->fd, "request\n", 8);
    cb_free (st->fd, 1);
    cb_add (st->fd, 0, doit_3, st);
}

static void doit_3 (void *_st) {
    struct state *st = _st;
    /* read more from st->fd until you get full response */
}
```
Syntactic sugar

• Problem: Need state from one callback to next
• E.g., C++ can implement `wrap` that bundles a function with its arguments

```cpp
callback<void, int>::ref errwrite = wrap (write, 2);
(*errwrite) ("hello", 5); // calls write (2, "hello", 5);
```

• Possible to build large event-driven apps this way
  - E.g., I have built large library to do this
  - Debugging features include recording where callbacks created to facilitate tracing

• Google reportedly does similar things
Pros & cons of event-based code

- **Advantages**
  - Fewer nasty bugs than threads (will discuss next week)
  - No locking, so no coarse- vs. fine-grained locking issues
  - Works with legacy, non-reentrant code (e.g., strtok, getpwnam)
  - Very efficient in terms of memory per client
  - Callbacks usually more efficient than thread switches

- **Disadvantages**
  - “Stack ripping” makes code ugly (but see [Adya])
  - Long running events make program unresponsive
  - Harder to take advantage of multiprocessors (but [Zeldovich])
  - Harder to do non-blocking disk I/O with existing OSes