Final Review

Exam logistics

- Friday, August 17th
- 7-10pm
- Skilling Auditorium
- SCPD students may take the exam any time Thursday or Friday
- Email me if you need any accommodations

Study strategies

- Read through lecture notes
  - Things should Make Sense™
- Revisit assignments
  - Know why everything works the way it does. I'll almost certainly reference your assignments or ask you to extend them
  - May be helpful to put core pieces of code on your cheat sheet
- Work through practice exams
- Work through lab questions

Material

- Filesystems
- Multiprocessing
  - Signals
  - Virtual memory
  - Scheduling
- Multithreading
  - Thread management
  - Synchronization
  - Design decisions
- Networking
  - IP addresses and port numbers
  - DNS
  - Understanding sockets and network connections
  - socket, bind, listen, connect syscalls
  - HTTP protocol
- Other topics
  - Nonblocking I/O
  - MapReduce
  - Guest talks

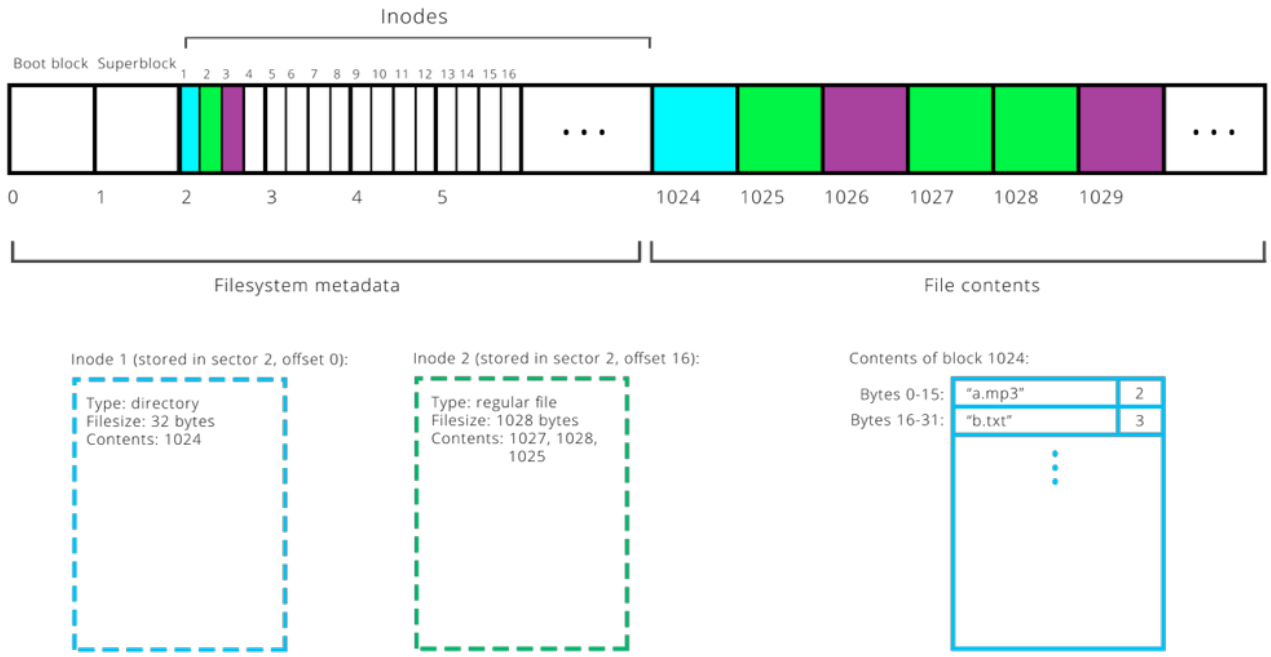
Material

- Filesystems
- Multiprocessing
  - Signals
  - Virtual memory
  - Scheduling
- **Multithreading**
  - Thread management
  - Synchronization
  - Design decisions
- **Networking**
  - IP addresses and port numbers
  - DNS
  - Understanding sockets and network connections
  - socket, bind, listen, connect syscalls
  - HTTP protocol
- Other topics
  - Nonblocking I/O
  - MapReduce
  - Guest talks

Filesystems

- FS design/layering: inodes, files, directories, links
- FS usage: file descriptors, file entries, nodes

Filesystem layout



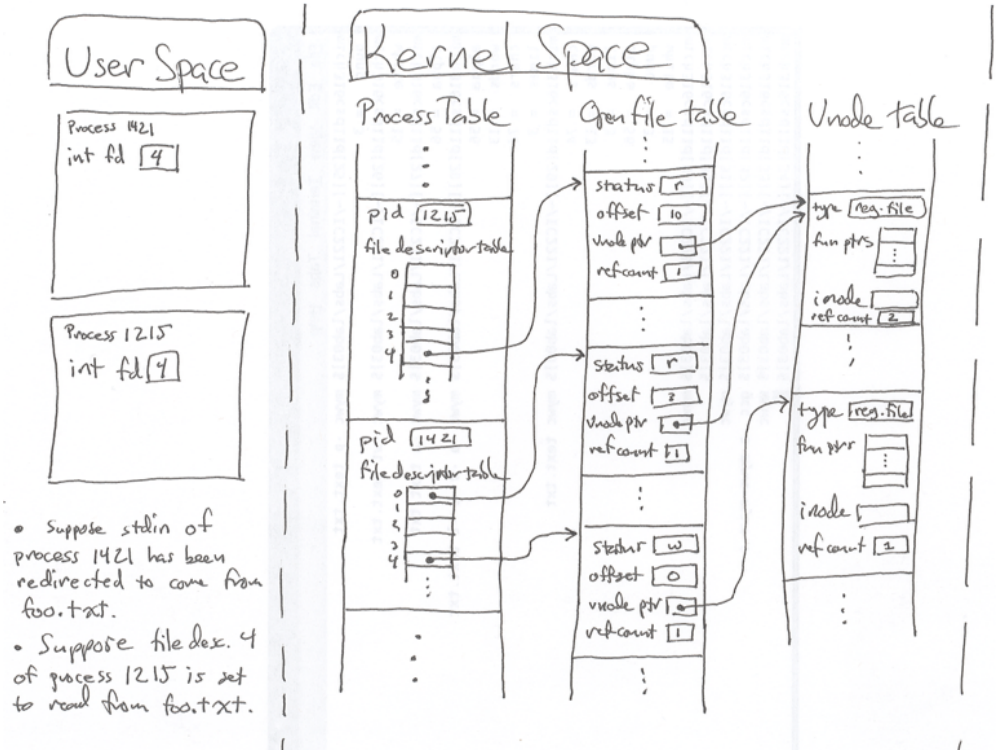
Directories

- Directories are just a special type of file
- Payload consists of (inode, name) pairs
- Know how the path resolution process works

Links

- Hard links are directory entries pointing to a file
- Soft links are special files whose payload is a path to a file

File descriptor/file entry/vnode tables



Great resource: https://www.usna.edu/Users/cs/wcbrown/courses/IC221/classes/L09/Class.html

File descriptor/file entry/vnode tables

- What do dup, dup2, open, close do to the tables?
- What is/isn't shared across processes?
- How do pipes work?

More multiprocessing

Signals, handlers, waitpid, sigsuspend
Scheduling

Signals

- A form of inter-process communication
- SIGINT, SIGTSTP, SIGCONT, SIGSTOP, SIGKILL
- signal(SIGCHLD, reapChild)
- Signal handlers are per-process and exist in the code segment (they are preserved across fork but not across execvp)
- Signals are handled when the process is on the CPU
  - If a process is in the blocked set, it will be moved to the ready queue upon receipt of a signal, then (usually) moved back to the blocked set when the signal is handled
- If a SIGCHLD arrives while executing the SIGCHLD handler, delivery of the second signal will be deferred until the handler finishes handling the first signal

Signal-related functions

- kill, raise, sigprocmask, signal, sigsuspend
- You can use sigprocmask to defer signals
- Signal masks are preserved across fork *and* execvp
- Sigsuspend sleeps until a signal is delivered

```
sigsuspend(&mask):
//ATOMICALLY:
   sigset_t old;
   sigprocmask(SIG_SETMASK, &mask, &old);
   sleep(); // wait for signal to wake us up
   sigprocmask(SIG_SETMASK, &old, NULL)
```

Signal puzzle

```
static pid_t pid;
static int counter = 0;

static void parentHandler(int unused) {
  counter += 2;
  printf("counter = %d\n", counter);
}

static void childHandler(int unused) {
  counter += 1;
  printf("counter = %d\n", counter);
  kill(getppid(), SIGUSR1);
}

1. Can this program DEADLOCK?
BONUS: How many outputs are there?
```

```
int main(int argc, char *argv[]) {
  signal(SIGUSR1, parentHandler);
  if ((pid = fork()) == 0) {
    signal(SIGUSR1, childHandler);
    sigset_t mask; sigemptyset(&mask);
    sigsuspend(&mask);
    return 0;
  }

  sleep(1); // hmmm...
  kill(pid, SIGUSR1);
  waitpid(pid, NULL, 0);
  counter += 3;
  printf("counter = %d\n", counter);
  return 0;
}
```

Signal puzzle

```
static pid_t pid;
static int counter = 0;

static void parentHandler(int unused) {
  counter += 2;
  printf("counter = %d\n", counter);
}

static void childHandler(int unused) {
  counter += 1;
  printf("counter = %d\n", counter);
  kill(getppid(), SIGUSR1);
}

1. Can this program DEADLOCK?
BONUS: How many outputs are there?
```

```
int main(int argc, char *argv[]) {
  signal(SIGUSR1, parentHandler);
  if ((pid = fork()) == 0) {
    signal(SIGUSR1, childHandler);
    sigset_t mask; sigemptyset(&mask);
    sigsuspend(&mask);
    return 0;
  }

  sleep(1); // hmmm...
  kill(pid, SIGUSR1);
  waitpid(pid, NULL, 0);
  counter += 3;
  printf("counter = %d\n", counter);
  return 0;
}
```

3 Outputs:
Output 1. - The child prints 2, and both the child and parent deadlock.
Output 2. - The child prints 1, the parent prints 2, and both the child and parent deadlock.
Output 3. - The child prints 1, the parent prints 2, and then the parent prints 5. Both processes exit.

Scheduling

- Process control block: struct representing a process's state
  - PID, register values, file descriptor table, performance statistics, etc
- Running set, ready queue, blocked set
- What causes a process to move from one queue to another?

Multithreading

Threads vs processes
Synchronization: locks, semaphores, condition variables
Design decisions

Threads vs processes

- Threads
  - Lightweight
  - Easier to synchronize and share information
  - Easier to make mistakes
- Processes
  - OS provides isolation and security
  - Harder to communicate and synchronize

Locks and lock guards

- Mutex motivation
  - Prevent race conditions: secure access to shared data structures
- Mutex gotchas:
  - Program can deadlock if you forget to unlock
  - Program can deadlock if you have too many locks and have circular dependencies
  - Program may run slower than necessary if you have too few locks or hold them for too long

Locks and lock guards

- mutex m;
  - Constructs mutex in unlocked state
- m.lock ( );
  - If unlocked, secures lock and proceeds
  - If locked, does not get lock and blocks
- m.unlock ( );
  - Should only call if you have the lock :D
  - Everyone else waiting on this lock wakes up and tries to acquire it
- lock_guard <mutex> lg (m);
  - Same as m.lock, except will automatically unlock when it goes out of scope

Semaphores

- Sempahore motivation:
  - “Bucket of balls” analogy
  - Easy primitive to sleep when we need to wait for something, and wake up when it becomes available
- Semaphore gotchas:
  - Program can deadlock if you take something from the bucket and forget to put it back

Semaphores

- semaphore s (initial val);
  - Constructs semaphore with “initial val” balls in the bucket
  - This is *not* a maximum size of the bucket
- s.wait ( );
  - If val > 0, atomically decrements val and proceeds
  - If val == 0, blocks
- s.signal ( );
  - “Returns” ball to the bucket by atomically incrementing val
  - Potentially wakes up threads which have blocked on s.wait ( ) so they can try again :)

Condition variables

- Motivation
  - Wait for some condition to become true
  - More flexible than a semaphore

Condition variables

```
while (!predicate) {
    wait();
}
```

Condition variables

```
m.lock();

while (!predicate) {
    m.unlock();
    wait();
    m.lock();
}

m.unlock();
```

Condition variables

cv.wait(m, predicate) {

```
m.lock();

while (!predicate) {
    // ATOMICALLY:
    m.unlock();
    wait();
    m.lock();
    // END ATOMICALLY
}

m.unlock();
```

Condition variables

- Condition_variable_any cv;
  - Condition variable constructor
- cv.wait (m, predicate);
  - Uses mutex m to safely evaluate whether predicate is true or false
  - If false, blocks until woken up
- cv.notify_one ( ); cv.notify_all ( );
  - Wake up one, or all (depends on which one you call) threads blocked bc of cv.wait. This only wakes them up so they can re-evaluate the predicate--if the predicate is false, they'll go back to sleep!

Condition variables

- Motivation
  - Wait for some condition to become true
  - More flexible than a semaphore
- Gotchas
  - You need to pass a *single* lock that protects any variables in the predicate

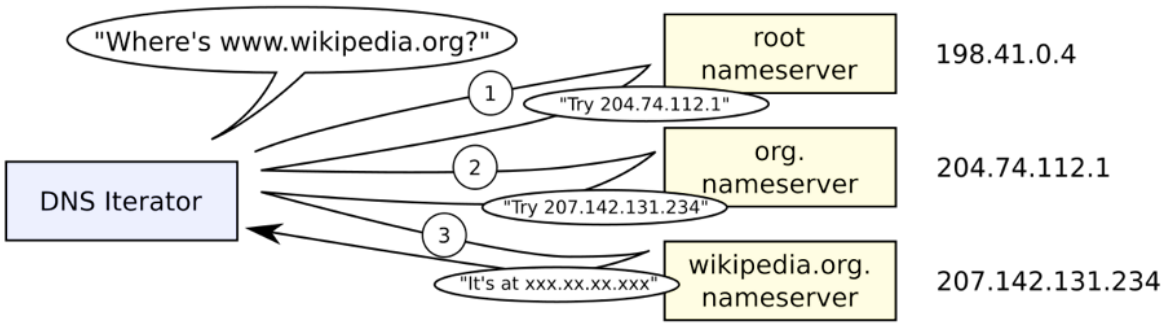
Design decisions

- How many threads should you spawn?
  - It depends. CPU-heavy or not?

Networking

IP addresses and port numbers
DNS: how resolution works, gethostbyname()
Understanding sockets and network connections
socket, bind, listen, connect syscalls
HTTP protocol

DNS resolution



https://en.wikipedia.org/wiki/File:Example_of_an_iterative_DNS_resolver.svg

Sockets

- **Communicating between processes on a machine:**
  - Use pipes
- **Communicating between different machines:**
  - We'd like to keep using the same kinds of abstractions
  - Use sockets!

Sockets

- Socket descriptors are returned by the socket() and accept() syscalls
- Nearly interchangeable with file descriptors
- Bidirectional
- Can be used to talk between processes on different machines
- Can be used to establish interprocess communication even after a process has started running

Working with sockets

- Since they act like file descriptors, we can use the read/write/close syscalls
- In practice, we more often use the sockbuf and iosockstream abstractions

time-client

```
int main(int argc, char* argv[]) {
    int client = createClientSocket("myth51.stanford.edu",
        12345);
    sockbuf sb(client);
    iosockstream ss(&sb);
    string timestr;
    getline(ss, timestr);
    cout << timestr << endl;
    return 0;
}
```

time-client

```
int createClientSocket (const string& host, unsigned short port)
{









}
```

time-client

```
int createClientSocket (const string& host, unsigned short port)
{
    struct hostent *he = gethostbyname(host.c_str());









}
```

time-client

```
int createClientSocket (const string& host, unsigned short port)
{
    struct hostent *he = gethostbyname(host.c_str());
    int client = socket(AF_INET, SOCK_STREAM, 0);









}
```

time-client

```
int createClientSocket (const string& host, unsigned short port)
{
    struct hostent *he = gethostbyname(host.c_str());
    int client = socket(AF_INET, SOCK_STREAM, 0);

    struct sockaddr_in serverAddress;
    memset(&serverAddress, 0, sizeof(serverAddress));
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_port = htons(port);
    serverAddress.sin_addr.s_addr = (struct in_addr *)he->h_addr)->s_addr;









}
```

time-client

```
int createClientSocket (const string& host, unsigned short port)
{
    struct hostent *he = gethostbyname(host.c_str());
    int client = socket(AF_INET, SOCK_STREAM, 0);

    struct sockaddr_in serverAddress;
    memset(&serverAddress, 0, sizeof(serverAddress));
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_port = htons(port);
    serverAddress.sin_addr.s_addr = (struct in_addr *)he->h_addr)->s_addr;

    connect(client, (struct sockaddr *) &serverAddress, sizeof(serverAddress));


}
```

time-client

```
int createClientSocket (const string& host, unsigned short port)
{
    struct hostent *he = gethostbyname(host.c_str());
    int client = socket(AF_INET, SOCK_STREAM, 0);

    struct sockaddr_in serverAddress;
    memset(&serverAddress, 0, sizeof(serverAddress));
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_port = htons(port);
    serverAddress.sin_addr.s_addr = (struct in_addr *)he->h_addr)->s_addr;

    connect(client, (struct sockaddr *) &serverAddress, sizeof(serverAddress));

    return client;
}
```

time-server

```
int main(int argc, char* argv[]) {
    int server = createServerSocket(12345);
    ThreadPool pool(8);
    while (true) {
        int client = accept(server, NULL, NULL);
        pool.schedule([client] { publish(client); });
    }
    return 0;
}
```

time-server

```
int createServerSocket(unsigned short port) {









}
```

time-server

```
int createServerSocket(unsigned short port) {
    int serverSocket = socket(AF_INET, SOCK_STREAM, 0);









}
```

time-server

```
int createServerSocket(unsigned short port) {
    int serverSocket = socket(AF_INET, SOCK_STREAM, 0);

    struct sockaddr_in serverAddress;
    memset(&serverAddress, 0, sizeof(serverAddress));
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);
    serverAddress.sin_port = htons(port);









}
```

time-server

```
int createServerSocket(unsigned short port) {
    int serverSocket = socket(AF_INET, SOCK_STREAM, 0);

    struct sockaddr_in serverAddress;
    memset(&serverAddress, 0, sizeof(serverAddress));
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);
    serverAddress.sin_port = htons(port);

    bind(serverSocket, (struct sockaddr *) &serverAddress,
        sizeof(serverAddress));


}
```

time-server

```
int createServerSocket(unsigned short port) {
    int serverSocket = socket(AF_INET, SOCK_STREAM, 0);

    struct sockaddr_in serverAddress;
    memset(&serverAddress, 0, sizeof(serverAddress));
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);
    serverAddress.sin_port = htons(port);

    bind(serverSocket, (struct sockaddr *) &serverAddress,
        sizeof(serverAddress));
    listen(serverSocket, 128);

}
```

time-server

```
int createServerSocket(unsigned short port) {
    int serverSocket = socket(AF_INET, SOCK_STREAM, 0);

    struct sockaddr_in serverAddress;
    memset(&serverAddress, 0, sizeof(serverAddress));
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);
    serverAddress.sin_port = htons(port);

    bind(serverSocket, (struct sockaddr *) &serverAddress,
        sizeof(serverAddress));
    listen(serverSocket, 128);
    return serverSocket;
}
```

Socket API calls

Used by both client and server

Socket

- Create an endpoint for communication
- Returns file descriptor that you can use to create sockbuf, iosockstream

Used by client

Connect

- Initiate a connection on a socket

Used by server

Bind

- Bind a name to a socket

Listen

- Listen for connections on a socket

Accept

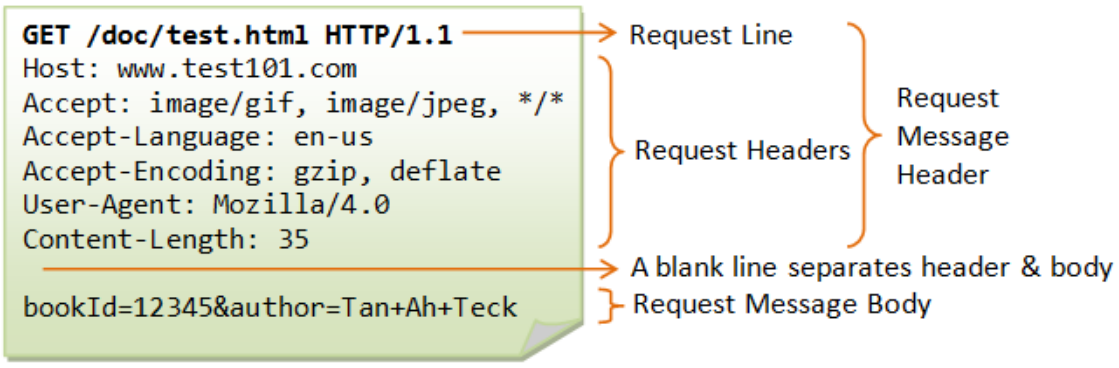
- Waits until someone “rings up” the server
- Returns the fd of the client who put in a request

Networking questions

- What are the similarities and differences between sockets and pipes?
- Why do we need the reentrant gethostbyname_r?
- Which socket API calls could block?
- Why do we handle requests in separate threads?
- How many open, yet-to-be-accepted requests can one server maintain? What about open connections (after accepting)?

HTTP requests/responses

Just know what they look like



Nonblocking I/O

Understand at a conceptual level
What's the point?
What do the epoll functions do?
Edge triggering vs level triggering

Nonblocking I/O

- We've overcome latency on blocking I/O operations by using threads
- However…
  - Threads are expensive
  - We only have a limited number of them
- Alternative: Nonblocking I/O
  - Configure file/socket descriptors as nonblocking descriptors. If we call read() or write() on them, those sys calls will return immediately instead of blocking

Nonblocking I/O

- Writing a nonblocking server:
  - Every time we accept() an incoming network request, configure the descriptor to be nonblocking
  - Add the descriptor to a vector
  - Loop over the vector, calling read() on each file descriptor to check whether there is any new information that has come in on each file descriptor
    - read() returns -1 with errno EAGAIN if there's no data to read right now

Nonblocking I/O

- Writing a nonblocking server:
  - Every time we accept() an incoming network request, configure the descriptor to be nonblocking
  - Add the descriptor to a vector
  - Loop over the vector, calling read() on each file descriptor to check whether there is any new information that has come in on each file descriptor

- Isn't this busy waiting? (Yes.)
  - Solve this problem with the epoll library
  - Similar to waitpid in some sense

Nonblocking I/O

- epoll_create: creates a “watch set” of file descriptors
  - Similar to how sigset_t is a set of signals you’re waiting for
- epoll_ctl: modifies a watch set
  - Similar to sigprocmask
- epoll_wait: waits until there is activity on a file descriptor in the watch set
  - Similar to sigsuspend

Nonblocking I/O questions

- Why does HTTP long polling not work well with your proxy implementation?
- How could you make it better?
- What are some disadvantages of nonblocking I/O?
- Would nonblocking I/O be a good addition for InternetArchive?

MapReduce

Know your implementation
Make sure you understand the point

Principles of System Design

Know what each principle means (but don't memorize them)
Be able to give examples of each principle

Guest talks

Have listened to them
(I won't ask anything obscure)

Take a deep breath! It's going to be okay.

Don't forget to sleep!