CS110 Winter 2020
Midterm Review

Shrey Gupta + Andrew Benson

Slides contributed by Kristine Guo, Peter McEvoy, Armin Namavari, Ryan Eberhardt, Grace Hong, Hemanth Kini
Exam Date & Time

Friday, February 14, 2020
NVIDIA Aud: A-S
Cubberley Aud: T-Z
1:30pm - 2:50pm
Exam Deets

- Bring your laptop and charger
- Exam will be administered on Bluebook
  - Download software before exam.
- If you don’t have a working laptop, we need to know by midnight tonight
- Exam material emphasizes assignments, sections, lecture, readings (in order of decreasing emphasis)
Outline

Part 1: Filesystems
- Inodes, Directories, Links
- File descriptor table, open file table, vnode table
- System calls (open, close, read, write, dup, dup2, pipe)

Part 2: Multiprocessing
- Processes and Virtual Memory
- fork, execvp, waitpid
- Pipes and multiprocessing

Part 3: Signals
- Signal blocking and handlers
- Race conditions and sigsuspend

Part 4: Scheduling

Part 5: Threads
- Thread Syntax
- Race Conditions, Mutex
Part 1: Filesystems
### Inodes & Files

<table>
<thead>
<tr>
<th>BOO T + SUPER BLOCKS</th>
<th>Inode 1</th>
<th>Inode 2</th>
<th>Inode 16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td></td>
<td></td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td></td>
<td></td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INODE TABLE</th>
<th>DATA, INDIRECT, &amp; UNALLOCATED BLOCKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>• • •</td>
<td></td>
</tr>
</tbody>
</table>

- Super block contains info on the type and config of the filesystem
- Inodes contain all the metadata regarding a file/directory.
- Data blocks contain actual file data or directory entries
- Indirect blocks contain lists of block numbers to data/indirect blocks
What’s in an inode?

```c
struct inode {
    uint16_t i_mode;       // bit vector of file type and permissions
    uint8_t  i_nlink;      // number of references to file
    uint8_t  i_uid;        // owner
    uint8_t  i_gid;        // group of owner
    uint8_t  i_size0;      // most significant byte of size
    uint16_t i_size1;      // lower two bytes of size
    uint16_t i_addr[8];    // device addresses constituting file
    uint16_t i_atime[2];   // access time
    uint16_t i_mtime[2];   // modify time
};
```

(From ino.h, assign2)
Direct Addressing

file size $\leq 8 \times 512$

324 | 21 | 972 | 123 | 273 | 453 | 284 | 312

512 bytes | 512 bytes | 512 bytes | 512 bytes | 512 bytes | 512 bytes | 512 bytes | 512 bytes
### Singly Indirect Addressing

\[ 8 \times 512 < \text{file size} \leq 7 \times 256 \times 512 \]

#### Single Indirect Block Numbers

<table>
<thead>
<tr>
<th>Num</th>
<th>Block Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>623 bytes</td>
</tr>
<tr>
<td>1</td>
<td>142 bytes</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>254</td>
<td>9232 bytes</td>
</tr>
<tr>
<td>255</td>
<td>1029 bytes</td>
</tr>
</tbody>
</table>

#### Doubly Indirect Block Num

- Block Num: 324
Doubly Indirect Addressing

7 * 256 * 512 < file size ≤
7 * 256 * 512 + 256 * 256 * 512

Single Indirect Block Nums:

623  142  ...  9232  1029

Doubly Indirect Block Num:

643

512 bytes

512 bytes

512 bytes

512 bytes
Directories

- Directories are just a special type of file
- Payload blocks consist of (inode block number, name) pairs
Links

- Hard links vs. symbolic (soft) links
- All three of these links go to the same file!
- Can’t create hard link for directories (breaks pathnames, allows loops)
How the operating system manages files
How System Calls Affect the File Tables

<table>
<thead>
<tr>
<th>fd table</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>open file table</td>
<td>In</td>
<td>Out</td>
<td>Err</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Effect of `pipe` on the File Tables

```
int fds[2];
pipe(fds);
// fds[0] has 8, fds[1] has 9
// Q: How do we redirect STDOUT to the pipe?
```
# The Effect of `dup2` on the File Tables

```
// int dup2(int oldfd, int newfd);
// have newfd point to what oldfd points to
dup2(fds[1], STDOUT_FILENO)
```
The Effect of `close` on the File Tables

```c
close(fds[1]);
// Q: What happens when we call dup(STDOUT_FILENO)?
```
The Effect of `dup` on the File Tables

```
// Returns a new fd that points to what the
// fd passed in is pointing to.
dup(STDOUT_FILENO);
```
The Open File Table is Shared Across Processes

Q: Suppose the parent forks, then reads 4 bytes. What is the next byte the child would read from shared.txt?

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
<th>Err</th>
<th>...</th>
<th>shared.txt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mode: r</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cursor: 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>refcount: 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vnode: *</td>
</tr>
</tbody>
</table>
The Open File Table is Shared Across Processes

A: ‘e’
Q: What changes in this picture if the child closes shared.txt?

parent

child

shared.txt

d a b c d e f g

In   Out  Err   ...  shared.txt
mode: r
cursor: 4
refcount: 2
vnode: *


The Open File Table is Shared Across Processes

A: The refcount drops to 1. The child no longer has an fd that points to the open file table entry.

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
<th>Err</th>
<th>...</th>
<th>shared.txt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mode: r</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cursor: 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>refcount: 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>vnode: *</td>
</tr>
</tbody>
</table>
Part 2.1: Processes
Processes

- Unique PID
- Not-necessarily unique PGID
- At least one thread
- Its own file descriptor table
- Its own Virtual Memory space
Virtualization

- DREAM: Every process thinks it has its own address space
- i.e., each process thinks it has sole access to the addresses ranging from 0x00000000 to 0xffffffff
Virtual Memory

- REALITY: Process address space isn’t really where data lives
- Translation facilitated by kernel on every “dereference” of an address.
Questions

Suppose P1 and P2 are separate process running /usr/bin/ls. Which of the following are possible?

- Both P1 and P2 call open("foo.txt") and in both the returned fd was 5.
- Both P1 and P2 read 7 chars from the returned fd and read the same thing.
- Both P1 and P2 store variable foo at virtual address 0xdeadbeef.
- Both P1 and P2 store variable foo at physical address 0xdeadbeef.
Part 2.2: Multiprocessing
int fork_child(char **argv) {
    pid_t pid = fork();
    if (pid == 0) {
        execvp(argv[0], argv);
        exit(0);
    }

    int status;
    waitpid(pid, &status, 0);
    return WEXITSTATUS(status);
}
fork()

- Duplicates almost everything: all of virtual memory, file descriptor table, signal handlers, signal mask, etc (not pending signals)
- Return value: child pid for parent, 0 for child
- I lied above: virtual memory is copied lazily i.e. Copy on Write (CoW)
execvp() (and friends)

- Replaces memory-related things: all of virtual memory, signal handlers, etc (not file descriptor table, pending signals, signal mask which are managed by the kernel)

- Return value: doesn’t return unless error (e.g. unknown program)
waitpid()

- Block until change in child processes’ running state (by default: only termination, but can detect stopped and signaled)
- Return value: -1 (error), 0 (WNOHANG specified and no children waiting), pid (of child who changed state)
- If you don’t reap, you’ll have zombie children!
## waitpid()

<table>
<thead>
<tr>
<th>Request Detection</th>
<th>Terminated</th>
<th>Signaled</th>
<th>Stopped</th>
<th>Continued</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>&lt;default&gt;</td>
<td>&lt;default&gt;</td>
<td>WUNTRACED</td>
<td>WCONTINUED</td>
</tr>
<tr>
<td>Check State</td>
<td>WIFEXITED</td>
<td>WIFSIGNALED</td>
<td>WIFSTOPPED</td>
<td>WIFCONTINUED</td>
</tr>
<tr>
<td>Additional Info</td>
<td>WEXITSTATUS</td>
<td>WTERMSIG</td>
<td>WSTOPSIG</td>
<td>&lt;none&gt;</td>
</tr>
</tbody>
</table>

- First row are flags for `waitpid`’s third argument, last two rows are macros that take in the status.
- Also - WNOHANG, which makes `waitpid` return early if nothing has already changed.
Questions

- The following is from a commit Linus Torvalds made to Linux last Saturday.

fork(); fork(); fork(); fork(); fork(); fork();

How many processes does it create?

- Review the following code. What possible issues could occur? (3-4 issues)

```c
int main(int argc, char *argv[]) {
    pid_t pid = fork();
    if (pid == 0) {
        execvp(argv[0], argv);
    }
    int* status;
    waitpid(pid, status, WNOHANG);
    assert(WIFEXITED(*status));
    return WEXITSTATUS(*status);
}
```
Part 2.3: Multiprocess Communication
How do processes communicate?

What we’ve seen in CS 110:

- pipes
- mmap
- signals
Pipes

What happens in the following code?

```c
static void writeOutput(const char *array[]) {
    printf("Writing output to file named \"%s\".\n", array[0]);
    int outfile = open(array[0], O_WRONLY | O_CREAT | O_TRUNC, 0644);
    dup2(outfile, STDOUT_FILENO);
    close(outfile);
    if (fork() > 0) return;
    execvp(array[1], array+1);
    exit(0);
}

int main(int argc, char *argv[]) {
    char *array1[] = {"cal.txt", "cal", NULL};
    char *array2[] = {"date.txt", "date", NULL};
    writeOutput(array1); waitpid(-1, NULL, 0);
    writeOutput(array2); waitpid(-1, NULL, 0);
    return 0;
}
```
mmap

- Like malloc, but is able to allocate memory that is shared between processes (e.g. after fork()).
Part 3: Signals
What are signals?

Asynchronous notifications sent to a process by the kernel or another process.

- Created via `kill()` or `raise()` (what’s the difference?).
- Handled by signal handlers registered via `signal()`.
- Can be blocked via `sigprocmask()`.
- Can be awaited (e.g. sleep until signal) via `sigsuspend()`.
- Examples: (why are these two lines separated?)
  - SIGINT, SIGTSTP, SIGCONT, SIGCHLD
  - SIGSTOP, SIGKILL
Signal Delivery

- If a process blocks a signal, delivery of the signal is delayed until it’s unblocked.
- If a process is not on CPU during signal delivery, it is delivered once it is.
- Signals aren’t queued!
  - The kernel tracks only what signals are delivered, not how many
- Signal handlers block the signal they are handling (e.g. can be interrupted by other signals).
Signal Handlers

- A function _you_ write that can be registered to run upon signal delivery.
  - `sighandler_t signal(int signum, sighandler_t handler);`
- Since signals are async, the signal handler might be run at any time! => beware of race conditions
- Avoid race conditions by blocking signals appropriately
  - `sigset_t set;`
  - `int sigemptyset(sigset_t *set);`
  - `int sigaddset(sigset_t *set, int signum);`
  - `int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);`
**Signal Blocking (find the bugs)**

```c
static int sum = 0;
static int children = 0;
int main(int argc, char *argv[]) {
    for (int i = 0; i < 5; i++) {
        pid_t pid = fork();
        if (pid == 0) {
            // do nonsense work
            exit(i);
        }
        children++;
    }
    children++;
}
signal(SIGCHLD, &handle);
while (children > 0) {
    // busy wait :(
}
cout << "sum: " << sum << endl;
return 0;
}
```

```c
static void handle(int signal) {
    int status;
    waitpid(-1, &status, 0);
    assert(WIFEXITED(status));
    sum += WEXITSTATUS(status);
    children--;
    cout << "One child exited!" << endl;
}
```
static int sum = 0;
static int children = 0;
int main(int argc, char *argv[]) {
    signal(SIGCHLD, &handle);
    for (int i = 0; i < 5; i++) {
        pid_t pid = fork();
        if (pid == 0) {
            // do nonsense work
            exit(i);
        }
        sigset_t set;
        sigemptyset(&set);
        sigaddset(&set, SIGCHLD);
        sigprocmask(SIG_BLOCK, &set, NULL);
        children++;
        sigprocmask(SIG_UNBLOCK, &set, NULL);
    }
    while (children > 0) {
        // busy wait :(
    }
    cout << "sum: " << sum << endl;
    return 0;
}

static void handle(int signal) {
    int status;
    while (true) {
        if (waitpid(-1, &status, WNOHANG) <= 0) {
            break;
        }
        assert(WIFEXITED(status));
        sum += WEXITSTATUS(status);
        children--;
        cout << "One child exited!" << endl;
    }
}
sigsuspend()

sigsuspend(&mask):

    //ATOMICALLY:
    sigprocmask(SIG_SETMASK, &mask, &old);
    pause(); // wait for signal to wake us up
    sigprocmask(SIG_SETMASK, &old, NULL);
Another kill-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?
Part 4: Scheduling
Scheduling
(not emphasized on midterm)
Scheduling

- Process control block (PCB) - struct representing a process’ state
  - What code it last was executing, register values, PID, etc

- PCBs put into one of:
  - blocked queue, ready/runnable queue, running queue

What causes a process to move from one queue to another?
Part 5: Threads
What are threads?

- A thread is an independent execution sequence within a single process.
- Threads share global parts of a virtual address space (text, data, heap) but have their own stack + registers.
- Threads are multiplexed onto processors
- Threads are often called lightweight processes.
C++ Thread Syntax

```cpp
#include <iostream>
#include <thread>

using namespace std;

int main()
{
    thread t([](a, b){
        cout << a + b << endl;
    }, 3, 3);
    t.join();
    return 0;
}
```

Thread constructor accepts a function and args. Processor schedules the thread to run the new function (i.e. adds thread to ready queue).

Parent thread waits for thread to finish executing.
Race Conditions with Threads

```c
static float balance = 100.0;

void withdraw(float money) {
    if (money <= balance)
        balance -= money;
}

int main()
{
    thread t2(withdraw, 100);
    thread t2(withdraw, 100);
    t1.join();
    t2.join();
    return 0;
}
```

Can the balance ever be negative?
What are mutexes?

- A mutex allows you to control access to a critical section of code.
- It’s like a key: if you have it, you can enter, otherwise you must wait till someone else gives you the key.
- When a thread encounters a mutex:
  - If it’s unlocked, lock the mutex and continue.
  - If it’s locked, block until it’s unlocked.
Race Conditions Fixed with Mutex

```c
static float balance = 100.0;

void withdraw(float money) {
    if (money <= balance)
        balance -= money;
}

int main()
{
    thread t1(withdraw, 100);
    thread t2(withdraw, 100);
    t1.join();
    t2.join();
    return 0;
}
```

Where should we put mutexes?
Race Conditions Fixed with Mutex

static float balance = 100.0;
static mutex lock;

void withdraw(float money) {
    lock.lock();
    if (money <= balance)
        balance -= money;
    lock.unlock();
}

int main()
{
    thread t1(withdraw, 100);
    thread t2(withdraw, 100);
    t1.join();
    t2.join();
    return 0;
}