

## CS110 Midterm Examination

---

This is a closed book, closed note, closed computer exam (although you are allowed to use your single double-sided cheat sheet). You have 120 minutes to complete all problems. You don't need to **#include** any header files, and you needn't guard against any errors unless specifically instructed to do so. Understand that the majority of points are awarded for concepts taught in CS110. If you're taking the exam remotely, you can call me at 415-205-2242 should you have questions.

Good luck!

SUNet ID (username): \_\_\_\_\_@**stanford.edu**

Last Name: \_\_\_\_\_

First Name: \_\_\_\_\_

I accept the letter and spirit of the honor code.

[signed] \_\_\_\_\_

		Score	Grader
1. File System Redux	[15]	_____	_____
2. <b>triple</b>	[10]	_____	_____
3. Distributed Bubble Sort Analysis	[20]	_____	_____
<b>Total</b>	<b>[45]</b>	_____	_____

## Relevant Prototypes

```
// filesystem access
int open(const char *path, int oflag, ...); // returns descriptor
ssize_t read(int fd, char buffer[], size_t len); // returns num read, 0 at eof
size_t write(int fd, char buffer[], size_t len); // returns num written
int close(int fd); // ignore retval
int pipe(int fds[]); // argument should be array of length 2, ignore retval
int dup(fd);
int dup2(int old, int new); // ignore retval
#define STDIN_FILENO 0
#define STDOUT_FILENO 1
#define STDERR_FILENO 2

// exceptional control flow and multiprocessing
pid_t fork();
pid_t waitpid(pid_t pid, int *status, int flags);
typedef void (*sighandler_t)(int sig);
sighandler_t signal(int signum, sighandler_t handler); // ignore retval
int execvp(const char *path, char *argv[]); // ignore retval
int kill(pid_t pid, int signal); // ignore retval
#define WIFEXITED(status) // macro
#define WEXITSTATUS(status) // macro
```

### Problem 1: File Systems Redux [15 points]

Unless instructed otherwise, your answers to the following questions should be 50 words or less. Responses longer than 50 words will receive 0 points. You needn't write in complete sentences provided it's clear what you're saying. Full credit will only be given to clear, correct, and thorough responses. Just because everything you write is technically true doesn't mean you get all the points.

- a. [2 points] The **close** system call detaches the supplied file descriptor from a file (or something that behaves like a file) so that the descriptor number can be reused. Briefly explain what happens to the relevant file entry table and vnode table entries as a result of **close** being called.
- b. [2 points] Explain what happens when you type **cd ..** at the shell prompt. Frame your explanation in terms of your Assignment 1 and 2 file systems and the fact that the inode number of the current directory is the only relevant global variable maintained by your shell.

- c. [4 points] Custom file server implementations may optimize for the type of file most commonly stored (e.g. a high-definition video file system used by Netflix versus a source code file system used by **github.com**). Would you go with a small block size (e.g.  $2^8 = 512$  bytes) or a larger block size (e.g.  $2^{15} = 32768$ ) for the high-definition video server if:
- you want to minimize the size of the inode table? Explain why.
  - you want to minimize the amount of unused space in trailing block? Explain why.
- d. [3 points] The **struct inode** used in Assignments 1 and 2 is 32 bytes in size, and among other fields includes space for 8 two-byte block numbers, where direct indexing is used to identify the sequence of payload blocks for small files, and indirect indexing is used to reach the sequence of payload blocks for large files. Assuming you can't remove any of the other **inode** fields, and you can't change their types, you want to shrink the memory footprint of the **inode** down so more **inodes** can fit in the same amount of space, or you can make the inode table smaller. You propose reducing the number of two-byte block numbers from 8 to 1, and relying on indirect indexing for all files, small and large. Briefly explain three distinct problems with this idea.

- e. [4 points] Consider the prototype for the **flock** system call, which is as follows

```
int flock(int fd, int operation);
```

which can be used to gain exclusive access to the file session bound to **fd**. The **operation** parameter can (for the purposes of this problem) be one of two constants, and those constants are as follows:

- **LOCK\_EX**, which is a request to grab exclusive access to a file session that should be respected by all other processes. If the resource isn't locked at the time of the call, then it is locked and flock returns right away. If the resource is locked, then the calling process blocks indefinitely until the lock is lifted by another process.
  - **LOCK\_UN**, which releases the lock held on a resource (or is a no-op if the lock wasn't held in the first place)
- [2 points, 25 words] Explain why information about the locked state of a file session needs to be stored in a file entry table instead of a file descriptor table.
- 
- 
- 
- 
- 
- 
- 
- 
- 
- 
- [2 points, 25 words] Explain why descriptors created using **dup** might reference locked file sessions, but descriptors returned from **open** initially reference a file session that is guaranteed to be unlocked.

## Problem 2: **triple** [10 points]

Leverage your **pipe**, **fork**, **dup2**, and **execvp** skills to implement **triple**, which has the following prototype:

```
static bool triple(int incoming, int outgoing,
                  char *one[], char *two[], char *three[]);
```

**incoming** is a valid, read-oriented file descriptor, **outgoing** is a valid, write-oriented file descriptor, and **one**, **two**, and **three** are well-formed, **NULL**-terminated argument vectors. **triple** launches three peer processes, each of which reboots using one, two, and three as their main function arguments, and waits for all three to exit, at which point it returns **true** if and only if all three processes terminated normally with exit status 0.

The first process's standard input is rewired to draw bytes from **incoming**, and its standard output is rewired to feed bytes to the standard input of the second process. The standard output of this second process is rewired to feed bytes to the standard input of the third process, whose standard output is rewired to direct its standard out to whatever resource is bound to **outgoing**.

For this problem, you need to implement everything, except you don't need to do any error checking (i.e. you can assume all system calls succeed) and you don't need to **close** any unused file descriptors. (You do in real code that you submit with assignments, but not for this exam problem).

Rather than writing out one long, monolithic implementation that replicates code, you are **required** to decompose by first implementing this helper function:

```
static pid_t createProcess(int in, char *argv[], int out);
```

and then calling it three times from **triple**. Your implementation of **createProcess** is the only function for your answer to this question where you can call **fork**, **dup2**, and **execvp**. You are otherwise free to use the three parameters in any way you like.

Use the next page to present your implementation of **createProcess** and **triple**.

**Problem 2: triple [continued]**

```
static pid_t createProcess(int in, char *argv[], int out) {
```

```
static bool triple(int incoming, int outgoing,  
                  char *one[], char *two[], char *three[]) {
```

### Problem 3: Distributed Bubble Sort Analysis [15 points]

Consider the following program on a 24-core machine, which launches 23 peer processes to cooperatively sort an array of 24 randomly generated numbers. (The implementations of **createSharedArray**, **printArray**, **freeSharedArray**, and **swap** are omitted for brevity.)

```
static const size_t kArraySize = 24, kNumSwappers = kArraySize - 1;
static size_t numSwappers = 0;

static void trackIdleSwappers(int unused) {
    while (waitpid(-1, NULL, WNOHANG | WUNTRACED) > 0) numSwappers++;
}

static void launchSwappers(pid_t pids[], int *array, size_t count) {
    for (size_t i = 0; i < count; i++) {
        if ((pids[i] = fork()) == 0) {
            while (true) {
                raise(SIGSTOP); // same as kill(getpid(), SIGSTOP)
                if (array[i] > array[i + 1]) swap(&array[i], &array[i + 1]);
            }
        }
    }
}

static void waitForAllSwappers() {
    sigset_t mask, existing;
    sigemptyset(&mask);
    sigaddset(&mask, SIGCHLD);
    sigprocmask(SIG_BLOCK, &mask, &existing);
    while (numSwappers < kNumSwappers) sigsuspend(&existing);
    sigprocmask(SIG_UNBLOCK, &mask, NULL);
}

static void killAllSwappers(pid_t pids[]) {
    waitForAllSwappers();
    signal(SIGCHLD, SIG_IGN); // SIG_IGN means ignore SIGCHLD signals from here on
    for (size_t i = 0; i < kNumSwappers; i++) kill(pids[i], SIGKILL);
    for (size_t i = 0; i < kNumSwappers; i++) waitpid(pids[i], NULL, 0);
}

int main(int argc, char *argv[]) {
    signal(SIGCHLD, trackIdleSwappers);
    int *array = createSharedArray(kArraySize); // note: see discussion on next page
    pid_t pids[kNumSwappers];
    launchSwappers(pids, array, kNumSwappers);
    for (size_t i = 0; i < kArraySize/2; i++) {
        waitForAllSwappers();
        for (size_t j = 0; j < kNumSwappers; j += 2) {
            numSwappers--; kill(pids[j], SIGCONT);
        }
        waitForAllSwappers();
        for (size_t j = 1; j < kNumSwappers; j += 2) {
            numSwappers--; kill(pids[j], SIGCONT);
        }
    }
    killAllSwappers(pids);
    printArray(array, kArraySize);
    freeSharedArray(array, kArraySize);
}
```



Your answers to the following questions should be 50 words or less.

- a. [2 points] Examine the implementation of **launchSwappers** on the previous page, and notice that each one conditionally reorders the integers at indices **i** and **i + 1**. Why is the **raise(SIGSTOP)** call needed?
  
  
  
  
  
  
  
  
  
  
- b. [2 points] The parent process instructs all even-indexed swappers to conditionally exchange the two numbers it has access to, and then instructs all odd-indexed swappers to do the same in a second wave. Why not allow all to swap simultaneously?

- c. [2 points] Notice the body of the parent's inner **for** loops include these two lines:

```
numSwappers--;  
kill(pids[i], SIGCONT);
```

Is the order of these two lines arbitrary (i.e. can these two lines be exchanged without impacting the correctness of the program)?

- d. [2 points] You temporarily change the implementation of **trackIdleSwappers** to look like this:

```
static void trackIdleSwappers(int unused) {  
    waitpid(-1, NULL, WUNTRACED);  
    numSwappers++;  
}
```

Very, very rarely the system prints a sorted array, but usually the program stalls and never exits. Why is that?

- e. [2 points] Notice the **signal(SIGCHLD, SIG\_IGN)** line within **killAllSwappers**. Will the program still produce the correct output if you remove it but otherwise leave everything else intact?

- f. [2 points] You temporarily change the implementation of **waitForAllSwappers** by replacing the **while** keyword with **if**, but change nothing else. Will the program still work 100% of the time? Explain why or why not.
- g. [3 points] The **createSharedArray** function sets aside space for an array of 24 integers and seeds it with random numbers. It does so using the **mmap** function you certainly saw a bunch while testing **trace**, and it does it like this:

```
static int *createSharedArray(size_t count) {
    int *array = mmap(NULL, count * sizeof(int), PROT_READ | PROT_WRITE,
                      MAP_ANONYMOUS | MAP_SHARED, -1, 0);
    for (size_t i = 0; i < kArraySize; i++) array[i] = random() % kArraySize;
    return array;
}
```

The **mmap** function takes the place of **malloc** here, because it sets up space not in the heap, but in an undisclosed segment that other processes can see and touch (that's what **MAP\_ANONYMOUS** and **MAP\_SHARED** mean).

Normally virtual address spaces are private and inaccessible to other processes, but that's clearly not the case here. Given what you know about virtual-to-physical address mapping, explain what the operating system must do to support this so that only the swappers have shared access but arbitrary, unrelated processes don't?

- h. [2 points] Why is the running time of our parallel sorting program still  $O(n^2)$ ?
- i. [3 points] Explain why the running time would be  $O(n)$  if trying to sort  $n$  numbers using  $n$  processors and just **trusted** the even-indexed swappers to all work at the same time, then take long enough of a break to allow the off-indexed swappers to work at the same time, back and forth and back and forth until the array is sorted. No **SIGCHLD** handlers, no **SIGSTOP** and **SIGCONT**. Just trust and really good scheduling.