Winter 2016 February 10th, 2016

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): **`root@stanford.edu`**

Last Name: Key

First Name: Answer

I accept the letter and spirit of the honor code.

[signed] _____

		Score	Grader
1. File System Redux	[15]	_____	_____
2. **`triple`**	[10]	_____	_____
3. Distributed Bubble Sort Analysis	[20]	_____	_____
Total	**[45]**	_____	_____

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
sizet_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.

- The reference count maintained by the file entry table entry is decremented, and if it falls to zero, it's marked is invalid. (1 point for decremented refcount and invalidation/irrelevance at 0 refcount).
- The same story for the relevant vnode table entry. (1 point for getting reference count part).

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.

Examine the current working directory's inode and search its payload for a directory entry name of **`".."`**, set relevant global variable to companion inode number. (1 point for searching inode payload for **`".."`**, another point for knowing its companion inode number is of for the parent directory, the new current working directory.)

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 would choose the larger block size, because the number of possible files (and therefore the maximum number of files) would be much smaller [2 points]

- you want to minimize the amount of unused space in trailing block? Explain why.

  smaller block size, since at most 511 bytes can be wasted, not 32767 [2 points]

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 **`inode`**s can fit in the same about 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.

- very small files require >= 2 blocks, lots of wasted space [1 point]
- small files require more cache-unfriendly indirection [1 point]
- **`inode`** size is 18 bytes, doesn't proliferate well with block size of 2^k [1 point]

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.
- **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.

  If a lock is to be respected across all processes, information about it must be stored in a data structure that's referenced by all processes, not just the one requesting the lock. Each process has its own, independent descriptor table. [2 points]

- [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.

  dup returns a descriptor referencing an existing session and promotes its refcount to be at least 2, whereas open creates a new session with a refcount of 1. [2 points]

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`**.

Solution 2: `triple` [continued]

```
static pid_t createProcess(int in, char *argv[], int out) {
  pid_t pid = fork();
  if (pid == 0) {
    dup2(in, STDIN_FILENO);
    dup2(out, STDOUT_FILENO);
    execvp(argv[0], argv);
  }

  return pid;
}
```

Criteria 2 for `createProcess`: 4 points

- Properly **`fork`**s and executes child-specific code in child: 1 point
- Properly call **`dup2`** for each descriptor (don't worry if they reverse the order, provided they're consistent): 1 point
- Properly invokes **`execvp`** with the correct arguments (if they do error checking on **`execvp`**, that's fine): 1 point
- *Ignore issues of variable declaration and return value, since that's not something that would be a problem with a compiler.*

```
static bool triple(int incoming, int outgoing,
                   char *one[], char *two[], char *three[]) {
  pid_t children[3];
  int bridge[4];
  pipe(bridge);
  children[0] = createProcess(incoming, one, bridge[1]);
  pipe(bridge + 2);
  children[1] = createProcess(bridge[0], two, bridge[3]);
  children[2] = createProcess(bridge[2], three, outgoing);
  bool good = true;
  for (size_t i = 0; i < 3; i++) {
    int status;
    waitpid(children[i], &status, 0);
    good = good && WIFEXITED(status);
  }
  return good;
}
```

Criteria 2 for `triple`: 6 points

- Properly creates two separate pipes and initialized each of them: 2 points
  - Declares two pipes instead of one: 1 point
  - Properly initializes all declared pipes: 1 point
- Supplies correct endpoints to **`createProcess`** calls: 2 points
  - 2 points if all correct (or they're all correct save for the problem where they think index 0 is for writing and index 1 is for reading… accept both orderings)
  - 1 point if there's an isolated hiccup
  - 0 points if there are two or more hiccups or more serious problems
- Properly computes return value: 1 point
- Properly waits for all three processes to finish before returning: 1 point

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 i = 0; i < kNumSwappers; i += 2) {
      numSwappers--; kill(pids[i], SIGCONT);
    }
    waitForAllSwappers();
    for (size_t i = 1; i < kNumSwappers; i += 2) {
      numSwappers--; kill(pids[i], SIGCONT);
    }
  }
  killAllSwappers(pids);
  printArray(array, kArraySize);
  freeSharedArray(array, kArraySize);
}
```

The system presented on the previous page is a nod to parallel programming and whether parallelism can reduce the asymptotic running time of an algorithm (in this case, a variation on bubble sort). We'll lead you through a series of short questions—some easy, some less easy—to test your multiprocessing and signal chops and to understand why (sometimes) the asymptotic running time of an algorithm can be improved in a parallel programming world.

For reasons I'll discuss shortly, this system works because the address in the **`array`** variable is cloned across the 23 **`fork`** calls, and this particular address **maps to the same set of physical addresses in all 24 processes** (and that's different than what usually happens).

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?

   Each one relies on signal handling to inform the orchestrator that it's waiting for permission to continue swap. [2 points]

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?

   Swappers with the same index parity can't move the same numbers, so it's safe to let them swap simultaneously. Permit neighboring swappers to examine and swap simultaneously and you have race conditions. [2 points]

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)?

   Yes, because **`numSwappers--`** is the parent's step back from condition required to transition from even to odd or odd to even, and the parent has taken all these steps by the time it reaches **`waitForAllSwappers`**. (Bonus point for pointing out that **`numSwappers--`** is technically a race condition, since it's not executed while **`SIGCHLD`** is being blocked). [2 points, extra +1 if mention race condition]

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?

trackIdleSwappers runs because one or more self-halting swappers have stopped. Without the **while** loop, the value of **numSwappers** might not (and rarely will) reach **kNumSwappers**. [2 points]

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?

It'll still sort the array and exit without crashing. (The handler will still be invoked and **waitpid** calls in the main thread of execution will return -1, but it won't crash or exit the program prematurely). [2 points]

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.

Nope. The program might need more than one invocation of **trackIdleSwappers** before **numSwappers** is promoted to **kNumSwappers** and have confirmation that all workers are stalled. [2 points]

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?

The OS must maintain information about the range of virtual addresses introduced by **`mmap`** and ensure that the same range of virtual addresses in the clones all map to the same set of physical pages in main memory, so they're all aliasing the same bytes.

h. [3 points] Why is the running time of our parallel sorting program still $O(n^2)$?

Each phase takes O(1) time to do all swaps simultaneously plus O(n) time to count all the **`waitpid`** calls, and there are n phases. [3 points]

i. [2 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.

We no longer have the count the **`waitpid`** calls or track **`numSwappers`**, so each of the O(n) phases takes O(1) time. [2 points]