CS110 Practice Midterm 2 Solution

Solution 1: `tee`

The **`tee`** user program copies everything from standard input to standard output, making zero or more extra copies in the named files supplied as user program arguments. For example, if the file **`alphabet.txt`** contains 27 bytes—the 26 letters of the English alphabet followed by a newline character, then the following would print the alphabet to standard output and to three files named **`one.txt`**, **`two.txt`**, and **`three.txt`**.

```
jcjr$ tee one.txt two.txt three.txt < alphabet.txt
abcdefghijklmnopqrstuvwxyz
jcjr$ more one.txt
abcdefghijklmnopqrstuvwxyz
jcjr$ diff one.txt two.txt
jcjr$ diff one.txt three.txt
jcjr$
```

If the file **`vowels.txt`** contains the five vowels and the newline character, and **`tee`** is invoked as follows, **`one.txt`** would be rewritten to contain only the English vowels, but **`two.txt`** and **`three.txt`** would be left alone.

```
jcjr$ tee one.txt < vowels.txt
aeiou
jcjr$ more one.txt
aeiou
jcjr$ more two.txt
abcdefghijklmnopqrstuvwxyz
```

For this question, you're to implement the entire **`tee`** program. You may assume that all system calls and C library functions succeed, so you needn't do any error checking at all. You'll implement the overall program in two parts over the next few pages.

a. [4 points] First, implement the **`writeall`** function, which accepts a valid file descriptor (open for writing), a character buffer, and a buffer length, and repeatedly calls **`write`** until the sequence of bytes residing in the buffer have all been published to the destination at the other end of the descriptor. Your implementation should assume that all incoming parameters are valid and that there are no failures (e.g. **`write`** never returns -1) that require any error checking. Your function should return the number of times **`write`** must be called in order to copy all of provided data.

```
static size_t writeall(int fd, const char buffer[], size_t len) {
   size_t count = 0;
   while (len > 0) {
      ssize_t numWritten = write(fd, buffer, len);
      count++;
      len -= numWritten;
      buffer += numWritten;
   }
   return count;
}
```

b. Using the **writeall** function you wrote for part a, present the **main** function that implements the **tee** user program. Recall that **tee** copies everything from standard input to standard output, making zero or more **extra** copies in the named files supplied as user program arguments. Your implementation should open these extra files using the supplied **#define** constants, pass the contents of standard input to all destinations, and close any descriptors it opened as part of execution. Any calls to **read** should attempt to read up to 2048 bytes of data. You should assume all system calls and library functions succeed (provided they're used properly, of course) and that you needn't do any error checking. You shouldn't dynamically allocate any memory as part of your implementation.

```
#define DEFAULT_FLAGS (O_WRONLY | O_CREAT | O_TRUNC)
#define DEFAULT_PERMISSIONS 0666

int main(int argc, char *argv[]) {
   int fds[argc];
   fds[0] = STDOUT_FILENO;
   for (size_t i = 1; i < argc; i++)
      fds[i] = open(argv[i], DEFAULT_FLAGS, DEFAULT_PERMISSIONS);

   char buffer[2048];
   while (true) {
      ssize_t numRead = read(STDIN_FILENO, buffer, sizeof(buffer));
      if (numRead == 0) break;
      for (size_t i = 0; i < argc; i++)
         writeall(fds[i], buffer, numRead);
   }

   for (size_t i = 1; i < argc; i++) close(fds[i]);
   return 0;
}
```

Solution 2: File Systems Redux

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, complete, relevant responses. Just because everything you write is technically true doesn't mean you get all the points.

a. The block layer of a file system needs to keep track of which blocks are allocated and which ones aren't. One scheme sets aside a portion of the underlying device to store a bitmap, where a single bit notes that some block is free if it's 0, or allocated if it's 1. The 0^{th} bit tracks the allocation status of the 0^{th} block beyond the bitmap, the 1^{th} bit tracks the allocation status of the 1^{th} block, and so forth. Another scheme might thread all of the unallocated blocks into a linked list (called a free list), where the first four bytes (i.e. the improvised next pointer) of each unallocated block store the block number of the next unallocated block in the free list, and so forth. The super block could store the block number of the first unallocated block in this list, and the last unallocated block could store 0 in its next pointer to signal the end of the free list.

   Briefly present one advantage of each approach.

   Advantages of the bitmap scheme:

   - no **`lseek`** required
   - more cache friendly

   Advantages of the free list approach:

   - zero memory footprint/overhead (clearly the better answer of the two I'm presenting)
   - arguably simpler to implement (though eventually both of these are equally easy to implement, so this is the weaker of the two answers)

b. If block sizes are 1024 (or 2^{10}) bytes and inodes are 32 (or 2^5) bytes, what percentage of the storage device should be allocated for the inode table if we never want to run out of inodes? Your answer can be approximate, and the 50-words-or-less defense of your answer should include the necessary math. Assume that all files require at least one block of payload, and assume a minimum storage device size of 1 terabyte (or 2^{40} bytes).

   One block of inodes can lead to 2^5 files, each with at least one block. 1 out of every 2^5 blocks must store inodes. That's ~3%. (Arguments involving 16 bytes of **`dirent`** structure are also good, but remember that answer only needed to be approximate.)

c. A soft symbolic link is an alias for another pathname (e.g. **`cs110`** is a soft symbolic link in my home directory that aliases **`/usr/class/cs110`**). Identify two of the layers contributing to your assign1 file system you would need to change (and how you would

change them) to support this new file type.

- inode layer would need to change to include a new file type
- pathname layer would need to change to replace the symbolic link component with its expansion
- other layers might be impacted, but above two layers require the most obvious changes.

d. A variant of the **open** system call—called **openat**—is equivalent to **open** unless the path argument is relative. In the case where the provided path argument is relative, the file to be opened is relative to the directory associated with the provided descriptor (instead of the current working directory). Leveraging your understanding of how **open** works with the file descriptor tables, the file entry table, and vnode entry table, and the layers of the file system, briefly explain how **openat** might be implemented.

```
int openat(int fd, const char *path, int oflag, ...);
```

If path is absolute, ignore **fd** and resolve as usual.
If path is relative, drill through **fd** to file entry to vnode entry to get companion inode number (not the same as the **fd**), and use that inode number as the start inode (instead of **IROOT_NUMBER**).

e. Describe a simple modification to your assign1 file system that would allow arbitrarily large file names while respecting the fact that **most** file names are short. We'd like the directory file payload to be reasonably compact.

If the file name is 14 or more characters, let the companion inumber refer to a new type of inode (similar to that used for symlinks in part 2b) or let it identify a block (or the head of a linked list of blocks) containing the full string.

Solution 3: Control Flow, Processes, and Signals

a. Consider the following C program and its execution. Assume all processes run to completion, all system and **printf** calls succeed, and that all calls to **printf** are atomic. Assume nothing about scheduling or time slice durations.

```
int main(int argc, char *argv[]) {
  pid_t pid;
  int counter = 0;
  while (counter < 2) {
    pid = fork();
    if (pid > 0) break;
    counter++;
    printf("%d", counter);
  }

  if (counter > 0) printf("%d", counter);
```

```
  if (pid > 0) {
    waitpid(pid, NULL, 0);
    counter += 5;
    printf("%d", counter);
  }

  return 0;
}
```

- List all possible outputs

  Possible Output 1: 112265
  Possible Output 2: 121265
  Possible Output 3: 122165

- If the **>** of the **`counter > 0`** test is changed to a **>=**, then **`counter`** values of zeroes would be included in each possible output. How many different outputs are now possible? (No need to list the outputs—just present the number.)

  18 [or 6 times whatever your answer to the first part were.]

b. Now consider this next program and its execution. Again, assume that all processes run to completion, all system and **`printf`** calls succeed, and that all calls to **`printf`** are atomic. Assume nothing about scheduling or time slice durations.

```
static void bat(int unused) {
  printf("pirate\n");
  exit(0);
}

int main(int argc, char *argv[]) {
  signal(SIGUSR1, bat);
  pid_t pid = fork();

  if (pid == 0) {
    printf("ghost\n");
    return 0;
  }

  kill(pid, SIGUSR1);
  printf("ninja\n");
  return 0;
}
```

For each of the five columns, write a **`yes`** or **`no`** in the header line. Place a **`yes`** if the text below it represents a possible output, and place a **`no`** otherwise. (Criteria: 1 point for each correct answer!)

yes!	yes!	no!	no!	no!
ghost **ninja**	**pirate** **ninja**	**ninja** **ghost**	**ninja** **pirate**	**ninja** **pirate**

pirate			**ninja**	**ghost**

Solution 4: `thyme`

The **`thyme`** program runs another program in a child process, and once the child process finishes, **`thyme`** publishes the number of seconds it took for the child process to run from start to finish. Assume, for instance, that I can invoke the **`make`** executable directly to compile two target programs like this:

```
myth15> make fd-puzzle fork-puzzle
gcc -g -Wall -pedantic -O0 -std=gnu99 -c -o fd-puzzle.o fd-puzzle.c
gcc fd-puzzle.o -o fd-puzzle
gcc -g -Wall -pedantic -O0 -std=gnu99 -c -o fork-puzzle.o fork-puzzle.c
gcc fork-puzzle.o -o fork-puzzle
```

I can do precisely the same thing using **`thyme`** to execute **`make fd-puzzle fork-puzzle`**, get the same output and generate the same compilation products, and also get the number of seconds it took to execute make using this:

```
myth15> thyme make fd-puzzle fork-puzzle
gcc -g -Wall -pedantic -O0 -std=gnu99 -c -o fd-puzzle.o fd-puzzle.c
gcc fd-puzzle.o -o fd-puzzle
gcc -g -Wall -pedantic -O0 -std=gnu99 -c -o fork-puzzle.o fork-puzzle.c
gcc fork-puzzle.o -o fork-puzzle
Elapsed time: 0.103602930 sec
```

To compute timing information, you should rely the following type and function declarations:

```
struct timespec {
   long tv_sec;  // seconds amount
   long tv_nsec; // nanoseconds amount
};

int clock_gettime(clockid_t clk_id, struct timespec *ts); // ignore return value
void print_elapsed_time(const timespec *start, const timespec *finish);
```

The first argument to **`clock_gettime`** should always be the constant **`CLOCK_REALTIME`**, and the second argument should be the address of a legitimate **`timespec`** record that, because the first argument is **`CLOCK_REALTIME`**, is populated with the number of seconds and nanoseconds that have elapsed since January 1st, 1970 at midnight. The **`print_elapsed_time`** function computes the difference between the two records addressed by **`start`** and **`finish`** and prints that difference on its own line in the format you need, as with **`Elapsed time: 0.103602930 sec`**.

Implement the full **`thyme.c`** program. Your implementation should execute the program being timed, wait for it to finish, and then print how long it took. (You may not use **`system`**, **`mysystem`**, **`popen`**, **`subprocess`**, or any other functions implemented in terms of **`fork`**, **`execvp`**, and so forth. You must explicitly call **`fork`**, **`execvp`**, etc. in the code you write.)

Solution 4: `thyme`

```
int main(int argc, char *argv[]) {
  struct timespec start;
  clock_gettime(CLOCK_REALTIME, &start);
  pid_t pid = fork();
  if (pid == 0) execvp(argv[1], argv + 1);
  waitpid(pid, NULL, 0);
  struct timespec finish;
  clock_gettime(CLOCK_REALTIME, &finish);
  print_elapsed_time(&start, &finish);
  return 0;
}
```

Solution 5: Multiprocessing Redux

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, complete, relevant responses. Just because everything you write is technically true doesn't mean you get all the points.

a. Recall that one vnode table and one file entry table are maintained on behalf of all processes, but that each process maintains its own file descriptor table. What problems would result if just one file descriptor table were maintained on behalf of all processes?

   All processes would be able to read and/or write to all other processes' files by polling all possible file descriptors. Huge security issues abound.

b. Give an example of a system call (with clear argument values) that might (but might not) move the calling process to the blocked queue, and give an example of a system call (with clear argument values) that will *definitely* move the calling process to the blocked queue.

   May or may not: **`waitpid(-1, NULL, 0)`** might return immediately without blocking, because there are no child processes, or it might wait years for the only child process to finish. (Other examples include **`read`**, **`write`**, and any other that's dependent on some resource that may or may not be available.)

   Definitely will: **`sleep(1)`**

c. Why are the pages used to map virtual addresses to physical ones typically a perfect power of 2 in size?

   Virtual pages of size 2^n are trivially mapped to physical pages of size 2^n by masking upper 64 - n bits and replacing with something else, while the lower n bits can be left alone as the offset within page. Other sizes require complicated, computationally expensive mapping schemes.

d. Recall the implementation of the **subprocess** routine and test program we presented in lecture to illustrate how the **pipe** function worked:

```
subprocess_t subprocess(const char *command) {
   int fds[2];
   pipe(fds);
   subprocess_t process = { fork(), fds[1] };
   if (process.pid == 0) {
      dup2(fds[0], STDIN_FILENO);
      close(fds[0]);
      close(fds[1]);
      char *argv[] = {"/bin/sh", "-c", (char *) command, NULL};
      execvp(argv[0], argv);
   }
   close(fds[0]);
   return process;
}

int main(int argc, char *argv[]) {
   subprocess_t sp = subprocess("/usr/bin/sort");
   const char *words[] = { "f", "u", "s", "h", "p", "a", "s", "i" };
   for (size_t i = 0; i < 8; i++) dprintf(sp.infd, "%s\n", words[i]);
   close(sp.infd);
   waitpid(sp.pid, NULL, 0);
   return 0;
}
```

Explain why the test program would stall without printing anything if the implementation of **subprocess** accidentally omitted its three calls to **close**.

The **close(sp.infd)** within the test program can only send EOF to the slave (and signal the end of input that **sort** must process) if the reference count within the relevant file entry drops from a 1 to a 0. That doesn't happen if other processes reference the file entry.