

600.211: Unix Systems Programming Midterm 2

Peter H. Fröhlich
phf@cs.jhu.edu

April 10, 2007

Time: 40 Minutes

Start here: Please fill in the following important information using a **permanent pen** before you do **anything** else! Your exam will **not** be graded if you use a pencil or erasable ink on this page.

Name (print): _____

Login (print): _____

Ethics Pledge: With your signature you **certify** the information above and you also **affirm** the following:
“I agree to complete this exam without unauthorized assistance from any person, materials, or device.”

Signature: _____

Date: _____

Instructions: Please read these instructions carefully before you start. **Switch off** your phones, pagers, and other noisy gadgets! You are **not** allowed to have anything but a pen (pencil, eraser) and this exam on your desk. You are **not** allowed to talk to anyone during the exam. If you have a question, please raise your hand **quietly**. You must **remain seated quietly** until all exams have been collected. Remember that you can **not** claim grading errors if you do not use a **permanent** pen for your answers.

Do not open before you are told to do so!

You got _____ out of 40 points.

1 Binary Warmup

(10 points)

For each of the following statements, determine whether it is either **true** or **false**. (1 point each)

1. System calls set `errno` to 0 (that's zero) on success.
2. Sockets can be closed even if they still have data to send in their buffer.
3. A session is a group of processes.
4. The `pthread_testcancel()` call provides explicit cancellation points for threads.
5. The whole UNIX API uses the same conventions for return and error codes.
6. The “execute” bit has meaning for files but not for directories.
7. We cannot change the disposition of the `SIGSTOP` signal.
8. Advisory record locking is enforced by the kernel on each and every file-related call.
9. Each `pthread_create()` call results in a new kernel-level thread being created.
10. The network socket API does not use any of the same signals that other IPC mechanisms use.

2 Tough Choices

(8 points)

For each of the following questions, circle **one** answer out of the choices given. (2 points each)

1. The **record locks** held by process p on file f are **released** when which of the following events occur?
 - (a) Process p exits cleanly.
 - (b) Process p closes a file descriptor for file f .
 - (c) Process p aborts due to a segmentation fault.
 - (d) All of the above.
 - (e) None of the above.
2. Which of the following is **not** inherited from parent to child after a `fork()` call?
 - (a) Process ID
 - (b) Parent Process ID
 - (c) Accumulated execution time
 - (d) All of the above.
 - (e) None of the above.
3. The facilities for **memory-mapped I/O** allow us to **avoid** which of the following system calls?
 - (a) The `ioctl()` and `fcntl()` calls.
 - (b) The `read()` and `write()` calls.
 - (c) The `open()` and `close()` calls.
 - (d) The `mmap()` and `munmap()` calls.
 - (e) None of the above.
4. The facilities for **asynchronous I/O** rely on which of the following **IPC** mechanisms?
 - (a) Pipes.
 - (b) Mutexes.
 - (c) Ports.
 - (d) Semaphores.
 - (e) None of the above.

4 Concurrent Sets

(14 points)

Consider a **concurrent** implementation of a set-like data structure for integer values. The data structure supports the operations **insert** (to add a new integer to the set if it's not there already), **remove** (to remove an integer from the set completely if it's there), and **has** (to check whether an integer is in the set or not).

We'll implement the set using a **growing** array (i.e. once we're out of space we double the size of the array and copy the integers) which should be maintained in **sorted** order to allow **binary search**; we expect sets to be pretty big on average, $n > 1000$ elements.

As an added twist, we want the operations on our set data structure to be **concurrent** in **two** ways: First, it should be possible for **multiple application threads** to use the data structure without having to worry about race conditions. Second, the data structure should be concurrent itself, allowing all three operations to return to the caller **quickly**; this requires threads **inside** the data structure that take care of certain time-consuming aspects such as maintaining the array in sorted order and resizing the array.

Describe in detail but without much code how you would structure the implementation of this data structure in terms of threads and mutexes. Be sure to address all three operations, describe how fast you can get them to return, and describe what "left-over work" will be done concurrently in the data structure. Feel free to use diagrams!

This page is intentionally **mostly** blank in case you run out of space elsewhere. If you ended up here early, please go over **everything** again and remain seated **quietly**! Make sure that the title page is filled out correctly and in **permanent** pen. Maybe you want to "rewrite" your **answers** in permanent pen as well?