Name: ID#: Lab Time:

CS 4061: Practice Exam 2 SOLUTION

Spring 2019
University of Minnesota

Exam period: 30 minutes
Points available: 40

Background: Sigblo C'Ker runs an application called `coordinated_changer` which makes changes to a single file in a safe way. According to the documentation for the code, any number of such processes can be run and they will be coordinated using a semaphore so no data will be lost. While running the program Sigblo accidentally hits the keystroke `Ctrl-c` and finds that `coordinated_changer` closes immediately but on trying to re-run it, Sigblo finds that he cannot get any more instances to run: all seem to "hang" immediately on starting. Looking at the source code for `coordinated_changer`, Sigblo would like to alter it so that `Ctrl-c` will kill `coordinated_changer` safely.

```
1  // rough code for coordinated_changer.c
2  int main(){
3    sem_t *file_lock = sem_open(..);
4
5    perform_setup();
6
7    sem_wait(file_lock);
8    modify_file_for_a_while();
9    sem_post(file_lock);
10
11   perform_cleanup();
12   return 0;
13 }
```

Problem 1 (5 pts): Based on the provided source code, explain why killing one instance of `coordinated_changer` at the wrong time causes all others to stall.

SOLUTION: Ctrl-C will send SIGINT to the `coordinated_changer` *and kill it. If this happens while the process has locked the semaphore with* `sem_wait()` *but before the semaphore is released via* `sem_post()`*, then the semaphore will remain locked. This will cause all subsequent processes to block on the* `sem_wait()` *and unable to proceed. The critical section in which the semaphore is locked must be protected somehow.*

Problem 2 (10 pts): Advise Sigblo on what changes should be made to prevent deadlock in `coordinated_changer`.

SOLUTION: Signal handling of some kind could be added. One solution establishes a signal handler for SIGINT and probably SIGTERM which would track that a signal has been received, call `sem_post()` *and* `exit()` *when it is received. This signal handling function is installed via* `sigaction()`*. However, there is danger that this handler may be called after the* `sem_post()` *has been called which will increment the semaphore inappropriately.*

An alternative solution is to block signals during the critical section (lines 6-10). Creating a `sigset_t` *with a call to* `sigprocmask()` *will block signals while a subsequent call can restore them to their default disposition.*

Problem 3 (5 pts): Pam Elif is writing a small database system. She would like to support multiple client programs reading and writing the database system simultaneously so is thinking of using a shared memory segment such as is provided by POSIX `shm_open()`. She also would like the database to be backed up by a disk file which a daemon process will occasionally copy from shared memory to disk but is finding the whole arrangement to seem overly complex.

Suggest a simpler mechanism that Pam can use which allows multiple processes to share memory that is automatically written to disk periodically.

Using a Memory Mapped File seems a good choice here. A file on disk that is opened and then `mmap()`*'d can be shared by several running processes. Changes made to this memory mapped data will be visible by other processes. The operating system automatically* `sync()`*'s the memory mapped data to disk periodically.*

Problem 4 (10 pts): Contrast FIFOs and POSIX Message Queues as means for inter-process communication. Describe at least 3 aspects that are similar or different between them.
SOLUTION: Similarities include

1. *Both obey first-in, first-out semantics and do not allow backtracking in the data.*
2. *Both persist beyond the life an individual process.*
3. *Both allow arbitrary, unrelated processes to communicate so long as a name/location is known by both processes.*
4. *Both use read/receive and write/send semantics for data that return quickly if space/data is available but block when not.*

Differences include

1. *FIFOs have a fixed limit in size, 64K in most cases. Message queues can be adjusted somewhat on their size when created.*
2. *FIFOs use names on the file system but Message Queues use a global name for specification which may or may not be on the file system.*
3. *FIFOs allow read/write of arbitrary numbers of bytes while Message Queues enforce chunks of data as individual messages which are received in units.*

Background: Consider the small application setup given in the nearby code. The intent is for the program to read commands interactively from a prompt or to allow the program to be launched in the background and read commands from a FIFO that is created. Answer the following questions about the program which reads input from two different sources.

Problem 5 (5 pts): Explain why the `select()` system call is used here rather than simply performing `read()` on the FIFO and standard input sources.

SOLUTION: If no data is ready for input, read() will block the calling process. Since it is not know whether data will be coming from stdin, the FIFO, or both, there is a real danger of blocking on input that will never come. `select()` *avoids this by checking both sources and returning when one or both actually have input. The process checks which is ready in the subsequent code blocks.*

Problem 6 (5 pts): Curiously the FIFO called `input.fifo` is opened in Read/Write mode at line 4 despite the program only reading from it. What problems does this approach avoid?

SOLUTION: FIFOs have the odd behavior of blocking a process open()'ing them for reading until another process opens the FIFO for writing. Since it may be that no process ever writes to the FIFO, this would cause the program to block and never make its way into the main loop. To that end, opening the FIFO in read/write mode allows the program to act as its own "partner" and keep the FIFO open indefinitely irrespective of other processes writing to it or not.

```
1 int main() {
2   mkfifo("input.fifo", S_IRUSR | S_IWUSR);
3   int stdin_fd = STDIN_FILENO;
4   int altin_fd = open("input.fifo", O_RDWR);
5
6   while(quit ==0){
7     printf("prompt> "); fflush(stdout);
8
9     fd_set fdset;
10    FD_ZERO(&fdset);
11    FD_SET(stdin_fd, &fdset);
12    FD_SET(altin_fd, &fdset);
13    int maxfd = stdin_fd;
14    maxfd = (maxfd < altin_fd ? altin_fd : maxfd);
15    select(maxfd+1, &fdset, NULL, NULL, NULL);
16
17    char buf[1024];
18    if(FD_ISSET(stdin_fd, &fdset)){
19      int n = read(stdin_fd, buf, 1024);
20      buf[n-1] = '\0';
21      execute_command(buf);
22    }
23    if(FD_ISSET(altin_fd, &fdset)){
24      int n = read(altin_fd, buf, 1024);
25      buf[n-1] = '\0';
26      execute_command(buf);
27    }
28  }
29
30  close(altin_fd);
31  remove("input.fifo");
32  return 0;
33 }
```