COMP3231/9201/3891/9283 Operating Systems 2020/T1

Tutorial Week 2

Questions

Operating Systems Intro

1. What are some of the differences between a processor running in *privileged mode* (also called *kernel mode*) and *user mode*? Why are the two modes needed?

2. What are the two main roles of an Operating System?

3. Given a high-level understanding of file systems, explain how a file system fulfills the two roles of an operating system?

4. Which of the following instructions (or instruction sequences) should only be allowed in kernel mode?

    1. Disable all interrupts.
    2. Read the time of day clock.
    3. Set the time of day clock.
    4. Change the memory map.
    5. Write to the hard disk controller register.
    6. Trigger the write of all buffered blocks associated with a file back to disk (`fsync`).

OS system call interface

5. The following code contains the use of typical UNIX process management system calls: `fork()`, `execl()`, `exit()` and `getpid()`. If you are unfamiliar with their function, browse the man pages on a UNIX/Linux machine get an overview, e.g: `man fork`

    Answer the following questions about the code below.

    a. What is the value of `i` in the parent and child after `fork`.
    b. What is the value of `my_pid` in a parent after a child updates it?
    c. What is the process id of `/bin/echo`?
    d. Why is the code after `execl` not expected to be reached in the normal case?
    e. How many times is *Hello World* printed when `FORK_DEPTH` is 3?
    f. How many processes are created when running the code (including the first process)?

```
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
```

```
#define FORK_DEPTH 3

main()
{
  int i, r;
  pid_t my_pid;

  my_pid = getpid();

  for (i = 1; i <= FORK_DEPTH; i++) {

    r = fork();

    if (r > 0) {
      /* we're in the parent process after
         successfully forking a child */

      printf("Parent process %d forked child process %d\n",my_pid, r);

    } else if (r == 0) {

      /* We're in the child process, so update my_pid */
      my_pid = getpid();

      /* run /bin/echo if we are at maximum depth, otherwise continue loop */
      if (i == FORK_DEPTH) {
        r = execl("/bin/echo","/bin/echo","Hello World",NULL);

        /* we never expect to get here, just bail out */
        exit(1);
      }
    } else { /* r < 0 */
      /* Eek, not expecting to fail, just bail ungracefully */
      exit(1);
    }
  }
}
```

---

6. a. What does the following code do?
   b. In addition to `O_WRONLY`, what are the other 2 ways one can open a file?
   c. What open return in `fd`, what is it used for? Consider success and failure in your answer.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char teststr[] = "The quick brown fox jumps over the lazy dog.\n";

main()
{
  int fd;
  int len;
  ssize_t r;


  fd = open("testfile", O_WRONLY | O_CREAT, 0600);
  if (fd < 0) {
    /* just ungracefully bail out */
    perror("File open failed");
    exit(1);
  }

  len = strlen(teststr);
  printf("Attempting to write %d bytes\n",len);

  r = write(fd, teststr, len);

  if (r < 0) {
    perror("File write failed");
    exit(1);
  }
  printf("Wrote %d bytes\n", (int) r);

  close(fd);

}
```

---

7. The following code is a variation of the previous code that writes twice.

   a. How big is the file (in bytes) after the two writes?
   b. What is `lseek()` doing that is affecting the final file size?
   c. What over options are there in addition to `SEEK_SET`?.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char teststr[] = "The quick brown fox jumps over the lazy dog.\n";

main()
{
  int fd;
  int len;
  ssize_t r;
  off_t off;


  fd = open("testfile2", O_WRONLY | O_CREAT, 0600);
  if (fd < 0) {
    /* just ungracefully bail out */
    perror("File open failed");
    exit(1);
  }

  len = strlen(teststr);
  printf("Attempting to write %d bytes\n",len);

  r = write(fd, teststr, len);

  if (r < 0) {
    perror("File write failed");
    exit(1);
  }
  printf("Wrote %d bytes\n", (int) r);

  off = lseek(fd, 5, SEEK_SET);
  if (off < 0) {
    perror("File lseek failed");
    exit(1);
  }

  r = write(fd, teststr, len);

  if (r < 0) {
    perror("File write failed");
    exit(1);
  }
  printf("Wrote %d bytes\n", (int) r);

  close(fd);

}
```

8. Compile either of the previous two code fragments on a UNIX/Linux machine and run `strace ./a.out` and observe the output.

    a. What is `strace` doing?
    b. Without modifying the above code to print `fd`, what is the value of the file descriptor used to write to the open file?
    c. `printf` does not appear in the system call trace. What is appearing in it's place? What's happening here?

9. Compile and run the following code.

    a. What do the following code do?
    b. After the program runs, the current working directory of the shell is the same. Why?
    c. In what directory does `/bin/ls` run in? Why?

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

main()
{
  int r;
  r = chdir("..");
  if (r < 0) {
    perror("Eek!");
    exit(1);
  }

  r = execl("/bin/ls","/bin/ls",NULL);
  perror("Double eek!");
}
```

10. On UNIX, which of the following are considered system calls? Why?

    1. `read()`
    2. `printf()`
    3. `memcpy()`
    4. `open()`
    5. `strncpy()`

Processes and Threads

11. In the *three-state process model*, what do each of the three states signify? What transitions are possible between each of the states, and what causes a process (or thread) to undertake such a transition?

---

12. Given N threads in a uniprocessor system. How many threads can be *running* at the same point in time? How many threads can be *ready* at the same time? How many threads can be *blocked* at the same time?

---

13. Compare reading a file using a single-threaded file server and a multithreaded file server. Within the file server, it takes 15 msec to get a request for work and do all the necessary processing, assuming the required block is in the main memory disk block cache. A disk operation is required for one third of the requests, which takes an additional 75 msec during which the thread sleeps. How many requests/sec can a server handled if it is single threaded? If it is multithreaded?

Page last modified: 12:29pm on Sunday, 24th of February, 2019

Screen Version