COMP3231/9201/3891/9283 Operating Systems 2021/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?
   In the kernel mode, OS has complete access to all hardware and can execute any instruction the machine is capable of executing. User mode only has access to what the OS allows
2. What are the two main roles of an Operating System?
   1. Resource manager 2. Provide abstraction to user programs
3. Given a high-level understanding of file systems, explain how a file system fulfills the two roles of an operating system?
   The OS automatically manages storage for us, we don't need to worry about how file is stored under the hood in the disk blocks. We only need to understand how the file system interface works
4. Which of the following instructions (or instruction sequences) should only be allowed in kernel mode?

   a) Switch from user to kernel mode
   Unprivileged because it's how applications invoke system calls. The catch is the application cannot control where the program counter goes when this switch happens.
   b) Read the clock
   Unprivileged, every process should be able to read the clock.
   c) Clear locations in virtual memory
   Unprivileged because this only harms the process calling it
   d) Turn off interrupts
   Privileged so that a process cannot monopolize the cpu.

   The simple approach is to determine which instructions you can use to screw the other users; those are the ones that should be 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`.
   the child is a new independent process that is a copy of the parent, i in the child will have whatever the value was in the parent at the point of forking
   b. What is the value of `my_pid` in a parent after a child updates it? the my_pid in a parent doesn't change
   c. What is the process id of `/bin/echo`? same as the pid in child process
   d. Why is the code after `execl` not expected to be reached in the normal case? because the program is either executed without errors, or will terminate in the first iteration
   e. How many times is *Hello World* printed when `FORK_DEPTH` is 3? 4 times
   ? f. How many processes are created when running the code (including the first process)?
   8? why

```
#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? O_RDONLY O_RDWR
c. What open return in `fd`, what is it used for? Consider success and failure in your answer.

the return value of open is a file descriptor, a small, non negative integer that is used in subsequent syscalls to refer to the open file. Because we have O_CREAT specified in the flags, if the specific file does not exist, open() creates the file in that path, if we don't have O_CREAT, then it signify an error with fd = -1

```
#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);
```

The following code opens a file called testfile in the program folder, if the file doesn't exist, it creates the file, the it writes teststr to the file and finally it closes the file

```
  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? 45 + 5 = 50
    b. What is `lseek()` doing that is affecting the final file size?
    c. What over options are there in addition to `SEEK_SET`?.

SEEK_CUR, the file offset is set to its current location plus offset bytes
SEEK_END, the file offset is set to the size of the file plus offset bytes

```
#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);
  }
```

lseek with SEEK_SET sets the starting byte of the second write to 5 bytes, the teststr is of 45 bytes, hence 5 bytes + 45 bytes = 50 bytes

```
  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? strace keeps trace of
    b. Without modifying the above code to print `fd`, what is the value of the file descriptor used to write to the open file? 3
    c. `printf` does not appear in the system call trace. What is appearing in it's place? What's happening here? printf is a library function that creates a buffer based on the string specification that it is passed. The buffer is then written to the console using write() to file descriptor 1

---

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?

this section of code list the files in the parent directory

```
#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

1. running to blocked: waiting for input, waiting for a timer, waiting for a resource to become available
2. running to ready: end of time-slice, voluntary yield(), scheduler picks a process with higher priority
3. ready to running: scheduler picks this process
4. blocked to ready: a resource has become available, so all processed blocked waiting for resource now become ready to continue execution

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?

---

there can only be one thread running

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? N - Running - Blocked N - Running - Ready

---

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

$15 \times \frac{2}{3} + (75+15) \times \frac{1}{3}$

10 + 30 = 40

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?

25 66

Critical sections

14. The following fragment of code is a single line of code. How might a race condition occur if it is executed concurrently by multiple threads? Can you give an example of how an incorrect result can be computed for x.

```
x = x + 1;
```

---

15. The following function is called by multiple threads (potentially concurrently) in a multi-threaded program. Identify the critical section(s) that require(s) mutual exclusion. Describe the race condition or why no race condition exists.

```
int i;

void foo()
{
    int j;

    /* random stuff*/

    i = i + 1;
    j = j + 1;

    /* more random stuff */
}
```

---

16. The following function is called by threads in a multi-thread program. Under what conditions would it form a critical section.

```
void inc_mem(int *iptr)
{
    *iptr = *iptr + 1;
}
```
