COMP3231/9201/3891/9283 Operating Systems 2021/T1

UNSW

Tutorial Week 2

Questions and Answers

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 user-mode:
   - CPU control registers are inaccessible.
   - CPU management instructions are inaccessible.
   - Part's of the address space (containing kernel code and data) are inaccessible.
   - Some device memory and registers (or ports) are inaccessible.

   The two modes of operation are required to ensure that applications (running in user-mode) cannot bypass, circumvent, or take control of the operating system.

---

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

   1) It provides a high-level abstract machine for programmers (hides the details of the hardware)
   2) It is a resource manager that divides resources amongst competing programs or users according to some system policy.

---

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

   At the level of the hardware, storage involves low-level controller hardware and storage devices that store blocks of data at many locations in the store. The OS filesystem abstracts above all these details and provides an interface to store, name and organise arbitrary unstructured data.

   The filesystem also arbitrates between competing processor by managing allocated and free space on the storage device, in addition to enforcing limits on storage consumption (e.g. quotas).

---

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`).

1,3,4,5 need to be restricted to kernel mode.

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);
    }
  }
}
```

   a. 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. `my_pid` in a parent is not updated by any action of the child and the child and parent are independent after forking.
c. `execl` replaces the *content* of a running process with specified executable. The process id does not change.
d. A successful `execl` results in the current code being replaced. `execl` does not return if it succeeds as there is no previous code to return to.
e. *Hello World* is printed 4 times if the `FORK_DEPTH` is 3.
f. There are 8 processes involved in the execution of the code.

---

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);

}
```

a. The code writes a string to a file. It will create a new file if needed (`O_CREAT`).
b. The other ways of opening a file are read-only (`O_RDONLY`) and read-write (`O_RDWR`).
c. In case of failure `fd` is set to -1 to signify an error. In the case of success, `fd` is set to a *file descriptor* (an integer) that becomes a handle to the file. The file descriptor is used in the other file related systems cases to identify the file to operate on.

---

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);

}
```

a. 50 bytes. For each open file, the operating system keeps track of the current offset within the file. The current offset is where the next read or write will start from. The current offset is usually at the location of offset of the end of the previous read or write. So one would expect the file size to be 90 bytes after two 45 byte writes, except for `lseeks` interference (see below).
b. `lseek` sets the current offset to a specific location in the file. The `lseek` in the code moves the current offset from 45 bytes (after the initial write) to 5 bytes from the start of the file. The second write begins from offset 5, writes 45 bytes, giving 50 bytes in total in the file.
c. See the man page for details on `SEEK_CUR` and `SEEK_END`

---

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?

        a. `strace` is printing a trace of all system calls invoked by a process, together with the arguments to the system call. There are a lot of system calls at the beginning of a trace related to dynamically loading code libraries. Towards the end of the trace you will see the system calls you expect to see.
        b. 3
        c. `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?

```
#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!");
}
```

        a. The code sets the current working directory of the process to be the parent directory (one higher in the directory hierarchy), and then runs `ls` to list the directory.
        b. The shell forks a child process that runs the code. Each process has its own current working directory, so the code above changes the current working directory of the child process, the current working directory of the parent process remains the same.
        c. `exec` replaces the content of the child process with `ls`, not the environment the child process runs in. The current working directory is part of the environment that the OS manages on behalf of every process, so `ls` runs in the current directory of child process.

---

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

    1. `read()`
    2. `printf()`
    3. `memcpy()`

4. `open()`
5. `strncpy()`

   1 and 4 are system calls, 2 is a C library functions which can call `write()`, 3 and 5 a simply library functions.

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?

    The three states are: *Running* , the process is currently being executed on the CPU; *Ready*, the process is ready to execute, but has not yet been selected for execution by the dispatcher; and *Blocked* where the process is not runnable as it is waiting for some event prior to continuing execution.

    Possible transitions are *Running to Ready*, *Ready to Running*, *Running to Blocked*, and *Blocked to Ready*.

    Events that cause transitions:

    - *Running to Ready*: timeslice expired, yield, or higher priority process becomes ready.
    - *Ready to Running*: Dispatcher chose the next thread to run.
    - *Running to Blocked*: A requested resource (file, disk block, printer, mutex) is unavailable, so the process is blocked waiting for the resource to become available.
    - *Blocked to Ready*: a resource has become available, so all processes blocked waiting for the resource now become ready to continue execution.

---

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?

    - Running threads = 0 or 1.
    - Blocked = N - Running - Ready
    - Ready = N - Running - Blocked

---

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?

    In the single-threaded case, the cache hits take 15 msec and cache misses take 90 msec. The weighted average is $2/3 \times 15 + 1/3 \times 90$. Thus the mean request takes 40 msec and the server can do 25 per second. For a multithreaded server, all the waiting for the disk is overlapped, so every request takes 15 msec, and the server can handle 66 2/3 requests per second.

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;
```

The single code statement is compiled into multiple machine instructions. The memory location corresponding to x is loaded into a register, incremented, and then stored back to memory. During the interval between the load and store in the first thread, another thread may perform a load, increment, and store, and when control passes back to the first thread, the results of the second are overwritten are them overwritten. Another outcome would be for the results of the first to be overwritten by the second (as the first thread loads, increments, then the second thread loads, increments, then the first thread stores, then the second thread stores).

---

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 */
}
```

There is no race condition on `j`, since it is a local variable per thread. However, `i` is a variable shared between threads. Thus `i = i + 1` would form a critical section (assuming no *random stuff* is dependent on `i`).

---

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;
}
```

Whether `*iptr = *iptr + 1` forms a critical section depends on the scope of the pointer passed to `inc_mem`. If the pointer points to a local variable, then there is no race. If the pointer points to a shared global variable there is potential for a race, and thus the increment would become a critical section.

Page last modified: 9:13am on Tuesday, 9th of February, 2021

Screen Version