Section 4: Sockets and Scheduling

February 21

Contents

1 Vocabulary

- **Socket** - Sockets are an abstraction of a bidirectional network I/O queue. It embodies one side of a communication channel, meaning that two must be required for a communication channel to form.The two ends of the communication channel may be local to the same machine, or they may span across different machines through the Internet. Most functions that operate on file descriptors like read() or write() work on sockets. but certain operations like lseek() do not.
- **file descriptors** - File descriptors are an index into a file-descriptor table stored by the kernel. The kernel creates a file-descriptor in response to an open call and associates the file-descriptor with some abstraction of an underlying file-like object; be that an actual hardware device, or a file-system or something else entirely. Using file descriptors, a process's read or write calls are routed to the correct place by the kernel. When your program starts you have 3 file descriptors.

File Descriptor	File
0	stdin
1	stdout
2	stderr

- **Scheduler** - Routine in the kernel that picks which thread to run next given a vacant CPU and a ready queue of unblocked threads.
- **Preemption** - The process of interrupting a running thread to allow for the scheduler to decide which thread runs next.
- **FIFO Scheduling** - First-In-First-Out (aka First-Come-First-Serve) scheduling runs jobs as they arrive. Turnaround time can degrade if short jobs get stuck behind long ones (convoy effect);
- **round-robin Scheduling** - Round-Robin scheduling runs each job in fixed-length time slices (quanta). The scheduler preempts a job that exceeds its quantum and moves on, cycling through the jobs. It avoids starvation and is good for short jobs, but context switching overhead can become important depending on quanta length;
- **Shortest Time Remaining First Scheduling** - A scheduling algorithm where the thread that runs is the one with the least time remaining. This is ideal for throughput but also must be approximated in practice.
- **Linux CFS** - Linux scheduling algorithm designed to optimize for fairness. It gives each thread a weighted share of some target latency and then ensures that each thread receives that much virtual CPU time in its scheduling decisions.
- **Earliest Deadline First** - Scheduling algorithm used in real time systems. It attempts to meet deadlines of threads that must be processed in real time by selecting the thread that has the closest deadline to complete first.

2 Scheduling

2.1 Round Robin Scheduling

Which of the following are true about Round Robin Scheduling?

1. The average wait time is less that that of FCFS for the same workload.
2. It requires pre-emption to maintain uniform quanta.
3. If quanta is constantly updated to become the # of cpu ticks since boot, Round Robin becomes FIFO.
4. If all threads in the system have the same priority, Priority Schedulers **must** behave like round robin.
5. Cache performance is likely to improve relative to FCFS.
6. If no new threads are entering the system all threads will get a chance to run in the cpu every `QUANTA*SECONDS_PER_TICK*NUMTHREADS` seconds. (Assuming `QUANTA` is in ticks).
7. It is the fairest scheduler

2,3
1. Easy to find counter example. 2. True. 3. True. 4. False. Not a requirement. 5. False. More context switches means worse cache performance. 6. Trick question. There is some overhead. 7. Trick question. Needs definition of fair.

2.2 Life Ain't Fair

Suppose the following threads denoted by THREADNAME : PRIORITY pairs arrive in the ready queue at the clock ticks shown. Assume all threads arrive unblocked and that each takes 5 clock ticks to finish executing. Assume threads arrive in the queue at the beginning of the time slices shown and are ready to be scheduled in that same clock tick. (This means you update the ready queue with the arrival before you schedule/execute that clock tick.) Assume you only have one physical CPU.

```
0   Yiming : 7
1
2   Alan : 1
3   Sarah: 3
4
5   Annie : 5
6
7   Neil: 11
8
9   Alex: 14
```

Determine the order and time allocations of execution for the following scheduler scenarios:

- Round Robin with time slice 3
- Shortest Time Remaining First (SRTF/SJF) WITH preemptions
- Preemptive priority (higher is more important)

Write answers in the form of vertical columns with one name per row, each denoting one clock tick of execution. For example, allowing Yiming 3 units at first looks like:

```
0   Yiming
1   Yiming
2   Yiming
```

It will probably help you to draw a diagram of the ready queue at each tick for this problem.

We're assuming that threads that arrive always get scheduled earlier than threads that have already been running or have just finished.

Explanation for RR:

From t=0 to t=3, Yiming gets to run since there is initially no one else on the run queue. At t=3, Yiming gets preempted since the time slice is 3. Alan is selected as the next person to run, and Sarah gets added to the run queue (t=2.9999999) just before Yiming (t=3).

Alan is the next person to run from t=3 to t=6. At t=5, Annie gets added to the run queue, which consists of at this point: Sarah, Yiming, Annie

At t=6, Alan gets preempted and Sarah gets to run since he is next. Alan gets added to the back of the queue, which consists of: Yiming, Annie, Alan.

From t=6 to t=9, Sarah gets to run and then is preempted. Yiming gets to run again from t=9 to t=10, and then finishes executing. Annie gets to run next and this pattern continues until everyone has completed running.

```
RR:
0   Yiming
1   Yiming
2   Yiming
3   Alan
4   Alan
5   Alan
6   Sarah
7   Sarah
8   Sarah
9   Yiming
10  Yiming
11  Annie
12  Annie
13  Annie
14  Alan
15  Alan
16  Neil
17  Neil
18  Neil
19  Alex
20  Alex
21  Alex
22  Sarah
23  Sarah
24  Annie
25  Annie
26  Neil
27  Neil
28  Alex
29  Alex
```

```
Preemptive SRTF
0   Yiming
1   Yiming
2   Yiming
3   Yiming
4   Yiming
5   Alan
6   Alan
7   Alan
8   Alan
9   Alan
...
(Pretty much just like FIFO since every thread takes 5 ticks)

Preemptive Priority
0   Yiming
1   Yiming
2   Yiming
3   Yiming
4   Yiming
5   Annie
6   Annie
7   Neil
8   Neil
9   Alex
10  Alex
11  Alex
12  Alex
13  Alex
14  Neil
15  Neil
16  Neil
17  Annie
18  Annie
19  Annie
20  Sarah
21  Sarah
22  Sarah
23  Sarah
24  Sarah
25 - 29 Alan
```

2.3 All Threads Must Die

You have three threads with the associated priorities shown below. They each run the functions with their respective names. Assume upon execution all threads are initially unblocked and begin at the top of their code blocks. The operating system runs with a preemptive priority scheduler. You may assume that set_priority commands are atomic.

Tyrion : 4
Ned: 5
Gandalf: 11

Note: The following uses references to Pintos locks and data structures.

```
struct list braceYourself;    // pintos list. Assume it's already initialized and populated.
struct lock midTerm;          // pintos lock. Already initialized.
struct lock isComing;

void tyrion(){
    thread_set_priority(12);
    lock_acquire(&midTerm);
    lock_release(&midTerm);
    thread_exit();

}

void ned(){
    lock_acquire(&midTerm);
    lock_acquire(&isComing);
    list_remove(list_head(braceYourself));
    lock_release(&midTerm);
    lock_release(&isComing);
    thread_exit();
}

void gandalf(){
    lock_acquire(&isComing);
    thread_set_priority(3);
    while (thread_get_priority() < 11) {
        printf("YOU .. SHALL NOT .. PAAASS!!!!!!);
        timer_sleep(20);
    }
    lock_release(&isComing);
    thread_exit();
}
```

What is the output of this program when there is no priority donation? Trace the program execution and number the lines in the order in which they are executed.

```
void tyrion(){
5    thread_set_priority(12);
6    lock_acquire(&midTerm); //blocks
    lock_release(&midTerm);
    thread_exit();

}

void ned(){
3    lock_acquire(&midTerm);
4    lock_acquire(&isComing);  //blocks
    list_remove(list_head(braceYourself));
    lock_release(&midTerm);
    lock_release(&isComing);
    thread_exit();
}

void gandalf(){
1    lock_acquire(&isComing);
2    thread_set_priority(3);
7    while (thread_get_priority() < 11) {
8        printf("YOU .. SHALL NOT .. PAAASS!!!!!!); //repeat till infinity
9        timer_sleep(20);
    }
    lock_release(&isComing);
    thread_exit();
}

Gandalf, as you might expect, endlessly prints "YOU SHALL NOT PASS!!" every 20 clock ticks or so.
```

What is the output and order of line execution if priority donation was implemented? Draw a diagram of the three threads and two locks that shows how you would use data structures and struct members (variables and pointers, etc) to implement priority donation for this example.

```
void tyrion(){
8    thread_set_priority(12);
9    lock_acquire(&midTerm); //blocks
    lock_release(&midTerm);
    thread_exit();

}

void ned(){
3    lock_acquire(&midTerm);
4    lock_acquire(&isComing);  //blocks
11   list_remove(list_head(braceYourself));  //KERNEL PANIC
    lock_release(&midTerm);
    lock_release(&isComing);
    thread_exit();
}

void gandalf(){
1    lock_acquire(&isComing);
2    thread_set_priority(3);
5    while (thread_get_priority() < 11) {   //priority is 5 first, but 12 at some later loop
6        printf("YOU .. SHALL NOT .. PAAASS!!!!!!);
7        timer_sleep(20);
    }
10    lock_release(&isComing);
      thread_exit();
}

It turns out that Gandalf generally does mean well. Donations will make
Gandalf allow you to pass.
At some point Gandalf will sleep on a timer and leave Tyrion alone in the
ready queue.
Tyrion will run even though he has a lower priority (Gandalf has a 5
donated to him)
```

```
Tyrion then sets his priority to 12 and chain-donates to Gandalf. Gandalf
breaks his loop.
After releasing the isComing lock, Gandalf's priority drops back to his priority without
donations (i.e. 3). Ned unblocks and acquires the isComing lock.
However, allowing Ned to remove the head of a list will trigger an ASSERT
failure in lib/kernel/list.c.

Gandalf will print YOU SHALL NOT PASS at least once.
Then Ned will get beheaded and cause a kernel panic that crashes Pintos.
```

3 Socket Programming

3.1 Multi-threaded Echo Server

Please look at the three versions of server code provided with Lecture 8. The first version uses a single process and single thread, the second version sequentially handles each client connection in a child process, and the third version allows child processes to handle connections concurrently.

Write a fourth version of the server implementation that uses multiple threads in a single process. Each connection is handled in its own thread, and threads should be allowed to handle connections concurrently.

```
#define BUF_SIZE 1024

struct addrinfo *setup_address(char *port) {
    struct addrinfo *server;
    struct addrinfo hints;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;

    int rv = getaddrinfo(NULL, port, &hints, &server);
    if (rv != 0) {
        printf("getaddrinfo failed: %s\n", gai_strerror(rv));
        return NULL;
    }
    return server;
}

void *serve_client(void *client_socket_arg) {
    int client_socket = (int)client_socket_arg;
    char buf[BUF_SIZE];
    ssize_t n;

    while ((n = read(client_socket, buf, BUF_SIZE)) > 0) {
        buf[n] = '\0';
        printf("Client Sent: %s\n", buf);

        if (write(client_socket, buf, n) == -1) {
            close(client_socket);
            pthread_exit(NULL);
        }
    }
    if (n == -1) {
        close(client_socket);
        pthread_exit(NULL);
    }

    close(client_socket);
    pthread_exit(NULL);
}

int main(int argc, char **argv) {
    if (argc < 2) {
        printf("Usage: %s <port>\n", argv[0]);
        return 1;
    }

    struct addrinfo *server = setup_address(argv[1]);
    if (server == NULL) {
        return 1;
    }
    int server_socket = socket(server->ai_family,
                               server->ai_socktype, server->ai_protocol);
    if (server_socket == -1) {
        return 1;
    }
    if (bind(server_socket, server->ai_addr,
         server->ai_addrlen) == -1) {
            return 1;
    }
    if (listen(server_socket, 1) == -1) {
        return 1;
    }

    while (1) {
        int connection_socket = accept(server_socket, NULL, NULL);
        if (connection_socket == -1) {
            perror("accept");
            pthread_exit(NULL);
        }

        pthread_t handler_thread;
        int err = pthread_create(&handler_thread, NULL,
                serve_client, (void *)connection_socket);
        if (err != 0) {
            printf("pthread_create: %s\n", strerror(err));
            pthread_exit(NULL);
        }
        pthread_detach(handler_thread);
    }
    pthread_exit(NULL);
}
```

4 Files

4.1 Files vs File Descriptor

What's the difference between `fopen` and `open`?

```
fopen is implemented in libc whereas open is a syscall. fopen will use open
in it's implementation. fopen will return a FILE * and open will return an int.
The FILE * object allows you to call utility methods from
stdio.h like fscanf. Also the FILE * object comes with some library
level buffering of writes.


--------------
|  libc      |
-------------
| syscall    |
--------------
```

4.2 Reading and Writing with File Pointers vs. Descriptors

Write a utility function, **void copy(const char *src, const char *dest)**, that simply copies the file contents from src and places it in dest. You can assume both files are already created. Also assume that the src file is at most 100 bytes long. First, use the file pointer library to implement this. Fill in the code given below:

```
void copy(const char *src, const char *dest) {
  char buffer [100];
  FILE* read_file = fopen(________, ____);
  int buf_size = fread(__________, ____, __________, __________);
  fclose(read_file);
  FILE* write_file = fopen(________, ____);
  fwrite(__________, ____, __________, __________);
  fclose(write_file);
}
```

```
void copy(const char *src, const char *dest) {
  char buffer [100];
  FILE* read_file = fopen(src, "r");
  int buf_size = fread(buffer, 1, sizeof(buffer), read_file);
  fclose(read_file);

  FILE* write_file = fopen(dest, "w");
  fwrite(buffer, 1, buf_size, write_file);
  fclose(write_file);
}
```

Next, use file descriptors to implement the same thing.

```
void copy(const char *src, const char *dest) {
  char buffer [100];
  int read_fd = open(________, ________);
  int bytes_read = 0;
  int buf_size = 0;
  while ((bytes_read = read(__________, __________, __________)) > 0) {
    _____________________
  }
  close(read_fd);
  int bytes_written = 0;
  int write_fd = open(________, ________);
  while (______________________) {
    __________ += write(__________, __________, __________);
  }
  close(write_fd);
}
```

```
void copy(const char *src, const char *dest) {
  char buffer [100];
  int read_fd = open(src, O_RDONLY);
  int bytes_read = 0;
  int buf_size = 0;

  while ((bytes_read = read(read_fd, &buffer[buf_size], sizeof(buffer) - buf_size)) > 0) {
    buf_size += bytes_read;
  }
  close(read_fd);

  int bytes_written = 0;
  int write_fd = open(dest, O_WRONLY);
  while (bytes_written < buf_size) {
    bytes_written += write(write_fd, &buffer[bytes_written], buf_size - bytes_written);
  }
  close(write_fd);
}
```

Compare the file pointer implementation to the file descriptor implementation. In the file descriptor implementation, why does **read** and **write** need to be called in a loop?

```
Read and write need to be called in a loop because there is no guarentee
that both functions will actually process the specified number of bytes
(they can return less bytes read / written). However, this functionality
is already handled in the file pointer library, so a single call to
fread and fwrite would suffice.
```