Section 5: RPC and Scheduling

October 2-4

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 processs 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

- **RPC** - Remote procedures calls are a technique for distributed computation through a client server model. This process effectively calls a procedure on a possibly remote server from a client by wrapping communication over the network with wrapper stub functions. This six steps are:
  1. The client calls the client stub. The call is a local procedure call, with parameters pushed on to the stack in the normal way.
  2. The client stub packs the parameters into a message and makes a system call to send the message. Packing the parameters is called marshaling.
  3. The client's local operating system sends the message from the client machine to the server machine.
  4. The local operating system on the server machine passes the incoming packets to the server stub.
  5. The server stub unpacks the parameters from the message. Unpacking the parameters is called unmarshaling.
  6. Finally, the server stub calls the server procedure. The reply traces the same steps in the reverse direction
- **Endianness** - The order in which the bytes are stored for integers that are larger than a byte. The two variants are big-endian where byte the most significant byte is at the lowest address and the least significant byte is at the highest address and little-endian where the least significant byte is at the lowest address and the most significant byte is at the highest address. The network is defined to be big-endian whereas most of your machines are likely little-endian.
- `uint32_t htonl(uint32_t hostlong)` - Function to abstract away endianness by converting from the host endianness to the network endianness.
- `uint32_t ntohl(uint32_t netlong)` - Function to abstract away endianness by converting from the network endianness to the host endianness.
- **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 RPC

To explore the process for performing RPC we will consider implement the server end for two procedures. We want to implement the server side of an RPC version of the following code

```
// Returns the ith prime number (0 indexed)
uint32_t ith_prime (uint32_t i);

// Returns 1 if x and y are coprime, otherwise 0.
uint32_t is_coprime(uint32_t x, uint32_t y);
```

Assume the server has already implemented `ith_prime` and `is_coprime` locally.

2.1 Selecting Functions

As a first step we receive data from the client. How do we decide which procedure we are executing? Provide a sample header file addition that could be used to indicate this.

We need to provide a unique id for each function and transmit it in at the start of our RPC message. There are many ways to do this, for example if these two functions were a comprehensive list of the rpc calls we needed to support we could provide an enum or we could use defines with unique values for example.

```
enum RPC_ID {
    PRIME_ID = 1,
    COPRIME_ID = 2
};
```

2.2 Reading Arguments

When examining the arguments for your two functions you notice that the arguments require either 8 or 4 bytes, so you believe you can handle either case by attempting to read 8 bytes using the code below.

```
// Assume dest has enough space allocated
void read_args (int sock_fd, char *dest) {
    int byte_len = 0;
    int read_bytes = 0;
    while ((read_bytes = read (fd, dest, 8 - byte_len)) > 0) {
        byte_len += read_bytes
    }
}
```

However when you implement it you notice that for some inputs your server appears to be stuck? Why might this be happening and for which inputs could this happen?

The read is trying to return 8 bytes from the socket or indicate there is no more data. However, when the socket is empty the read will only return due to failure (or if the socket is closed). So long as the connection stays open the reads will never fail and will block instead.

2.3 Handling RPC

Realizing your previous solution was insufficient you decide to implement a slightly more complicated protocol. You settle on the following steps for the client:

1. The client sends an identifier of the function it wants as an integer (0 for `ith_prime`, 1 for `is_coprime`).
2. The client sends all the bytes for all the arguments.

The server then takes the following steps:

1. The server reads the identifier.
2. The server uses the identifier to allocate memory and set the read size.
3. The server reads the remaining arguments.

Complete the following function to implement the server side of handling data. You may find `ntohl` useful.

```
// Function to implement the server side of the protocol.
// Returns whether or not it was successful and closes the socket when finished.
// Assume get_sizes loads all our sizes based on id and call_server_stub selects
// our host function
void receive_rpc (int sock_fd) {
    uint32_t id;
    char *args;
    size_t arg_bytes;
    char *rets;
    size_t ret_bytes;
    int bytes_read = 0;
    int bytes_written = 0;
    int curr_read = 0;
    int curr_write = 0;
    while ((curr_read = read (sock_fd, ((char *)&id) + bytes_read,
            sizeof (uint32_t) - bytes_read)) > 0) {
        bytes_read += curr_read;
    }
    id = ntohl (id_data);
    get_sizes (id, &args, &arg_bytes, &rets, &ret_bytes);
    bytes_read = 0;
    while ((curr_read = read (sock_fd, &args[bytes_read], arg_bytes - bytes_read)) > 0) {
        bytes_read += curr_read;
    }
    call_server_stub (id, args, arg_bytes, rets, ret_bytes)
    while ((curr_read = write (sock_fd, &rets[bytes_written],
            arg_bytes - bytes_written)) > 0) {
        bytes_written += curr_write;
    }
    close (sock_fd);
}
```

2.4 Call Stubs

Finally we want to implement the call stubs, which are function wrappers between each individual function we support and the generic RPC library. For example these are the client stubs for our functions

After the raw data is processed we need to perform the actual computation and return the result to the client. To do this we introduce a stub procedure which unpacks our arguments, calls the procedure to execute on the server, and finally packs the return results to give to the transport layer.

```
// addrs is used for setting up our socket connection
uint32_t ith_prime_cstub (struct addrinfo *addrs, uint32_t i) {
    uint32_t prime_val;
    i = htonl(i);
    call_rpc (addrs, RPC_PRIME, (char *)&i, 1, (char *) &prime_val, 1);
    return ntohl(prime_val);
}

uint32_t is_coprime_cstub (struct addrinfo *addrs, uint32_t x, uint32_t y) {
    uint32_t coprime_val;
    uint32_t args[2] = {htonl(x), htonl(y)};
    call_rpc (addrs, RPC_COPRIME, (char *) args, 2, (char *) &coprime_val, 1);
    return ntohl(coprime_val);
}
```

`call_rpc` is our generic rpc handler that our stubs provide an abstraction around. Notice that we marshal arguments with `htonl()` and unmarshal them with `ntohl()`.

Implement `ith_prime_sstub` and `is_coprime_sstub` which should unmarshal the arguments, call the implementation functions, and finally marshal the return data.

```
void ith_prime_sstub (char *args, char *rets) {
    uint32_t* args_ptr = (uint32_t *) args;
    uint32_t i = ntohl (args_ptr[0]);
    uint32_t result = ith_prime (i);
    result = htonl (result);
    memcpy (rets, &result, sizeof (uint32_t));
}


void is_coprime_sstub (char *args, char *rets) {
    uint32_t* args_ptr = (uint32_t *) args;
    uint32_t x = ntohl (args_ptr[0]);
    uint32_t y = ntohl (args_ptr[1]);
    uint32_t result = is_coprime (x, y);
    result = htonl (result);
    memcpy (rets, &result, sizeof (uint32_t));
}
```

3 Scheduling

3.1 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.

3.2 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   Sam : 7
1
2   Jack : 1
3   Sharie: 3
4
5   Annie : 5
6
7   Will: 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 Sam 3 units at first looks like:

```
0   Sam
1   Sam
2   Sam
```

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

Jack 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: Sharie, Sam, Annie

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

From t=6 to t=9, Sharie gets to run and then is preempted. Sam 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   Sam
1   Sam
2   Sam
3   Jack
4   Jack
5   Jack
6   Sharie
7   Sharie
8   Sharie
9   Sam
10  Sam
11  Annie
12  Annie
13  Annie
14  Jack
15  Jack
16  Will
17  Will
18  Will
19  Alex
20  Alex
21  Alex
22  Sharie
23  Sharie
24  Annie
25  Annie
26  Will
27  Will
28  Alex
29  Alex
```

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

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

4 Additonal Questions

These are questions we may not get to in discussion but you may find useful when studying for the exam.

4.1 RPC Consistency

In our RPC example we opted to use the type `uint32_t`. What is a potentially issue that could result if we opted to use an `unsigned int` instead? If a value can only take either 0 or 1 is could your previous answer still be an issue?

> `uint32_t` is defined to be exactly 4 bytes whereas an `unsigned int` has a machine specific size. This is a problem because our server and client may not agree on the size which could cause data to be incorrectly unmarshaled. This is still an issue if the value are only 0 and 1 because size issues could incorrectly cause us to read part of another argument for example or reproduce us waiting on data like in 2.2.

4.2 Socket Choices

RPC calls are implemented with socket implementations. In lecture you saw running a server with 1 thread and last week in discussion you looked at sockets with multiple threads. Below is the code for running a server where each request is its own process.

```
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;

    iF (getaddrinfo(NULL, port, &hints, &server) != 0) {
        perror ("Address");
        return NULL;
    }
    return server;
}
```

```
int setup_server_socket (struct addrinfo *server) {
    // Setup the addr socket
    bool connected = false;
    int sock = -1;
    for (struct addrinfo *addr = server; addr != NULL && !connected; addr = addr->ai_next) {
        int sock = socket (addr->ai_family, addr->ai_socktype, addr->ai_protocol);
        if (sock != -1) {
            if (bind(sock, addr->ai_addr, addr->ai_addrlen) == -1)
                close (sock);
                sock = -1;
            } else {
                return sock;
            }
        }
    }
    return sock;
}

void run_server (char *port) {
    struct addrinfo *server = setup_address(port);
    int server_fd = setup_server_socket (server);
    if (server_fd == -1) {
        perror ("socket");
    } else {
        int rv = listen (server_fd, BACKLOG);
        if (rv < 0) {
            perror ("socket");
        } else {
            while (1) {
                int connection_socket = accept (server_fd, NULL, NULL);
                int pid = fork ();
                if (pid < 0) {
                    perror ("fork");
                } else if (pid == 0) {
                    receive_rpc (connection_socket);
                    exit (0);
                } else {
                    close (connection_socket);
                }
            }
        }
    }
}
```

What's a possible advantage of playing each RPC request in a separate process instead of a separate thread?

The biggest advantage for us here is safety. Because we are exposing function calls it is possible we could be fed an input that doesn't meet our assumptions which if not properly handled could cause a segfault. If we have this code in a separate thread it will crash the process while if its in a separate process it won't crash the whole server.

4.3 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
12   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);
11    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.
Ned unblocks after Gandalf exits.
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.
```