

1 Data Structures

The following structures support our kernel:

Arrays

We use an Array as a map for the name server to map task ids to the name they register as. We chose to use a fixed length character array with size equal to the maximum number of concurrent tasks we allow (100). The maximum length of an entry is 100 characters.

Linked Lists

We are using Linked lists for memory management through slab allocation. On boot, the kernel will allocate a big slab of memory from 0x10000000 to 0x70000000 and hand out 128kB sections for tasks (note the kernel stack starts at the default location set in the linker script given to us, at a much lower address).

We use intrusive linking to build the linked list into the data structure we would like to store together. We have not made use of any non-trivial algorithms at this point.

For K2, we added 3 new Linked lists for Send, Receive, Reply blocked queues. These linked lists are scanned when a SRR request comes through. We chose to use a linked list rather than a hashmap for our blocked queues because we didn't expect a significant performance benefit for a fixed (100) number of concurrent tasks given the overhead of creating a working hashmap implementation.

Priority Queues

We implement our priority queues assuming a fixed amount of priority levels (5) for the purpose of scheduling tasks.

We are using Singly Linked Lists for our priority queues, thus have 2 arrays of size the amount of priority levels storing the head and tail of each priority. Each priority in the list points at the head of a linked list of task descriptors. Each task descriptor keeps a pointer to the next task descriptor in the same priority (or NULL if it is the last element).

Ring Buffers

We use ring buffers for the allocation of task IDs by the kernel, as well as for a fixed memory player queue for our K2 Rock-Paper-Scissors Server. We chose to use a ring buffer so we could enforce the maximum concurrent tasks we allow in our system, as well as queue players for the RPS server to register and play against each other.

2 System Parameters and Design Choices

Maximum tasks

As we currently do not free the space as tasks exit, this allows us to spawn a total of ~12582 tasks. This seemed like a reasonable amount to meet the requirements in this course, once we start freeing the memory on task exit we will be able to run a maximum of 12582 tasks concurrently.

Stack Size

Task descriptors are given 128kB with task metadata taking up some of this space (few dozen bytes). 128kB seemed like it would be sufficient for the tasks we are writing and allows us to run a large number of tasks. Our program is written in a way that this number can be adjusted if we need more stack space/more tasks down the line.

Request Passing

User syscalls populate a request object that specifies what request they would like and the potential arguments for the syscall on their own stack. A pointer to this struct is returned to the kernel which uses it to perform the syscall.

Interrupts

The way we handle interrupts is done in a way such that we can re-use as much of the context switch code as possible. In both scenarios to enter the kernel (system call and interrupt) we save all user registers onto the user stack and swap to kernel mode. The entry point for interrupts is a different function that attempts to find the handler for the interrupt (task waiting for the event). It readies this task and then returns to 'activate' with a dummy request object for the scheduler to ignore.

The general flow of handling interrupts goes as follows:

1. Some task enables the interrupt via a syscall.
2. The interrupt is received by a handler and identified.
3. The handler asserts that there is a task waiting for this interrupt.
4. The specific interrupt is disabled (for uart interrupts they are disabled via `UART_IER` so we can continue to get other UART interrupts)
5. The handler frees the waiting task and returns control to the scheduler.

3 Kernel Loop

Our kernel follows the pattern given in class. It creates the first user task and then enters a loop that continues until there are no more tasks in any ready queue. The loop:

1. Gets a task from the scheduler (first task in the highest priority that is ready to run).
2. Sets up the appropriate values for the task (stack pointer, pstate, return value, pc) and enters the task through `eret`.
3. Upon return to the kernel it expects to receive a pointer to a request on the user task's stack containing information on the syscall made. It then handles this syscall (performs the action required and adds the task back onto the appropriate queue if necessary).

4 Idle Time

The system spends around 96-97% of its time idling (as of K3). We measure this in the kernel by tracking when the idle task gets scheduled and interrupted. This value is outputted to the screen every 100 loops of the idle task (to prevent the cursor from flickering too much) by dividing the amount of time spent in the idle task divided by the current system time.

5 Syscalls

CREATE

On Create Syscall, the kernel creates a task and appends it to the requested priority queue, then it returns the TID of the new task. Returns the unique positive integer task id of the newly created task.

We assume that the priority is always valid and the kernel always has available task descriptors since we are the ones creating the tasks. These assumptions are checked with assertions when the resources are being allocated. If we start to approach or exceed the limit of task descriptors available (100), we use the doubling strategy (by manually increasing the hardcoded value) to increase the maximum amount of task descriptors up to the maximum amount of tasks available (12582).

MY TID

The kernel reads and returns the Task ID from the requesting task.

PARENT TID

The kernel reads and returns the Parent Task ID from the requesting task. If the parent has been destroyed, we return the dead parent's ID.

YIELD

The kernel reschedules the task to the back of the same priority queue

EXIT

We use exit as a destroy, the kernel frees the task's ID and frees the section of memory the task occupied by appending it to a freelist.

SEND

The kernel first checks if the Send target is in the Received blocked queue with a scan, if the target is Receive Blocked, the kernel copies the minimum of the sender's msg length and the receiver's expected msg length in bytes from the sender's msg buffer to the receiver's receive buffer. The receiver task is then rescheduled at its original priority and the task is put in the Reply Blocked queue.

The return value of the receiver task is set to the length of the message copied in bytes.

If the target is not receive blocked, the kernel puts the task on the Send Blocked queue.

RECEIVE

The kernel first checks if any messages are trying to be sent to the task by scanning the Send Blocked queue. If a sender task exists, the kernel copies the minimum of the sender's msg length and the receiver's expected msg length in bytes from the sender's msg buffer to the receiver's receive buffer. The task is then rescheduled at its original priority and the Sender is put in the Reply Blocked queue.

The return value of the task is set to the length of the message copied in bytes.

If there is no message to be received, the kernel puts the task on the Receive Blocked queue.

REPLY

The kernel scans the reply blocked queue for the task to be replied to, this task must exist. The reply message is copied as the minimum of the task's msg length and the sender's expected reply length in bytes from the receiver's msg buffer to the receiver's reply buffer. Both the task and the reply blocked sender is then rescheduled at their original priority.

The return value of the sender task is set to the length of the message copied in bytes.

AWAITEVENT

The kernel puts the task on the queue for tasks waiting for events. When an event arrives, the kernel scans all tasks waiting on this queue and frees the task when the corresponding event happens.

TIME

This is a wrapper for SEND. The kernel directs this message to the clock server which replies with the current time in ticks.

DELAY

This is a wrapper for SEND. The kernel directs this message to the clock server which waits the specified amount of ticks and then replies with the current time.

DELAYUNTIL

This is a wrapper for SEND. The kernel directs this message to the clock server which waits until the system time reaches the specified time then replies with the current time.

UPDATETIME

This is a wrapper for SEND, intended only to be used by the clock notifier. The kernel directs this message to the clock server which updates its current time to the specified time.

IDLE

This is a system call used by the idle task that yields back to the kernel. The kernel counts how many of these it gets and updates the idle time every 1000 idles.

UARTREADREGISTER/UARTWRITEREGISTER

These syscalls read/write to the given register.

PUTS

This is a wrapper for a send to the appropriate UART channel. It sends the given null terminated string to the server to be transmitted through the uart.

PUTC/GETC

These syscalls get or put a character onto the appropriate uart channel, is actually a send to the appropriate server.

GETNEXTTERMINALCOMMAND

This syscall is a send to the uart terminal rx server. It gets the next command (separated by carriage return) from the terminal. Currently supports commands up to 100 bytes.

EXIT

This syscall reboots the program.

6 UART Servers and K4

We have 4 UART servers for Terminal RX/TX and Train RX/TX.

Terminal Rx/Tx

For transmitting to the terminal, we transmit as many bytes as we can (checking TXLVL every time we send). Once we can't send anymore, we enable the TX interrupt for UART channel 0 and wait for the interrupt. Once we get the interrupt we go back to a ready state, ready to transmit more bytes.

For the RX server, we start with the RX interrupt active. Once we get an interrupt, we read as many bytes as we can, storing them in a buffer. Once we can't read anymore, we renable the interrupt and go back to waiting. The flow is the same whether we get the RX timeout or regular RX interrupt. In fact, we don't differentiate between the two. Upon receiving a carriage return from the user, we reply to a terminal admin task with the last command up to the enter. The terminal admin parses the command and performs the appropriate action by sending bytes to the train set.

Train Rx/Tx

The TX server uses a state machine for flow control. After sending a byte to the train, we enter a non-ready state. In order to go back to the ready state, we require 2 modem interrupts (for CTS down and up) and a tx interrupt. Once we receive these 3 interrupts we go back to our ready state and are able to transmit another byte.

An interesting side effect of using null terminated strings as messages to the server is that we can never send the byte 0 to the train set. To work around this (instead of refactoring our server communication), we simply add 16 to every speed command, so our trains always have their lights on.

The RX server works the same as the RX server for the terminal in that it read as much as it can always. We have a sensor reading server that polls the RX server every tick for additional sensor data, upon reading all 10 bytes we output the appropriate sensor data.

UART Notifiers

We have uart notifiers that are responsible for processing uart interrupts. They await for a uart interrupt. We have multiple uart notifiers to ensure there is always one waiting for a uart interrupt. When one arrives, they are placed on the ready queue by the irq handler and they forward this interrupt to the appropriate uart server. The uart server uses this interrupt to change its internal state (towards a ready state).

7 Task Structure for Train Control

This is the tentative task structure we have chosen for doing train control.

UART Servers

UART servers act as black boxes for communicating with the terminal and train. Our other tasks send the bytes that it wants to send and request bytes by SENDING to the appropriate server.

Terminal Admin

This task sits and waits for a carriage return and processes the command given. This is the only server that communicates the terminal RX server so the terminal RX server will buffer characters and pass back the entire command to the terminal admin.

Upon receiving the command it checks if it is a valid command. If the command is:

1. a train command (tr, rv), the terminal admin sends the command to the appropriate train worker (see below).
2. a switch command (sw), the terminal admin spawns a switch worker (see below) which turns the switch.

Train Worker

These workers are responsible for the commands to a single train. They act as couriers between the terminal admin and the train UART TX server. At startup, they wait for a message (sent by the bootstrap task) that tells it what train it is responsible for. It then registers itself as “TRAIN_WORKER_XX“ where XX is the train number it is responsible for.

It then waits for requests to send to the train. For speed commands it verifies that the speed is valid, adds 16 to the given speed (for lights) and then sends it to the uart server. The worker also notes down the speed of the train (initialized to 0 on startup). For reverse commands it sends the speed 0 command (with lights on) and then delays itself 5s. It then sends the reverse command, delays 100ms and then sends the speed up command. We were having an issue where sometimes the reverse command would not reverse train 2 (train will stop, wait 5 seconds, lights would flicker and the train would keep going in the same direction). Our hypothesis is that the train is in the process of reversing when the marklin gets the speedup command and drops the reverse command. With the 100ms delay this is no longer an issue.

With one worker per train, we have a way to easily serialize our commands to the trains (when a train is reversing, since the worker is delaying itself, all other commands will queue). A consequence of this is that terminal admins will also block, so we need to spawn sufficiently enough terminal admins such that we can buffer all commands inside an admin.

Once we get to train control programmatically, I can imagine us spawning tasks whose responsibility it is to send a command to the train worker and then exit. For now, this solution is sufficient for the terminal commands.

Switch Worker

These workers are spawned by the terminal admin to turn switches. They parse the given command (checks that the switch number is valid and the direction is valid). It executes the command and sends the direction and switch number to the switch server and then exits.

Switch Server

This server keeps track of switch states and outputs them to the screen. It gets messages from the switch workers. In the future this will probably be merged in with whatever server does the train control as switch states determine the track state.

Sensor Query Server

This server is responsible for sending sensor queries. It resets the sensors and then sends sensor queries to the train tx uart server every 200ms.

Sensor Read Server

This server is responsible for reading sensor data from the train server. It polls the train rx server for bytes and upon receiving 2 bytes (bytes for one sensor), it parses the bytes and outputs if necessary.

We also keep track of which sensors were recently pressed so that a train that holds down a sensor while passing over only counts as one sensor reading. If a sensor never goes to being not triggered, we don't count the reading. There is the obvious issue where if two trains pass over the same sensor very quickly (in 200ms), then we will interpret this was only one sensor trigger, but we don't foresee this being an issue.

8 Kernel Output (for K1-K3)

K1

On running the task specified in K1 it outputs the following:

```
Created: 2
Created: 3
My Task Id: 4 , My Parent's Task Id: 1
My Task Id: 4 , My Parent's Task Id: 1
Created: 4
My Task Id: 5 , My Parent's Task Id: 1
My Task Id: 5 , My Parent's Task Id: 1
Created: 5
FirstUserTask: exiting
My Task Id: 2 , My Parent's Task Id: 1
My Task Id: 3 , My Parent's Task Id: 1
My Task Id: 2 , My Parent's Task Id: 1
My Task Id: 3 , My Parent's Task Id: 1
```

Explanation

The first task (task 1) is created in priority 1 (our priority goes from 0 → 4, highest priority → lowest priority).

Task 1 then creates two tasks (2, 3) in priority 2. When Create() reschedules task 1, it will be on a higher priority than the created tasks so it will first print “Created: 2/3”.

Task 1 then creates the third task (task 4) in priority 0 so when Create() goes back to the kernel and gets rescheduled, task 4 gets scheduled before task 1. It prints “My Task Id: 4 , My Parent's Task Id: 1”, Yields()s, but gets scheduled again since it is of higher priority than any other task. It prints “My Task Id: 4 , My Parent's Task Id: 1” again and Exit()s.

At this point task 1 gets scheduled and prints “Created 4”. The same thing happens again with task 5. When task 5 completes, task 1 is scheduled again and Exit()s, printing “FirstUserTask: exiting”.

Then, the first two created tasks (2,3) get scheduled printing the last four lines in alternating fashion as each Yield() call allows the other task to run and print to the screen. Upon each printing twice they exit, concluding the program.

K2

Running the RPS server and clients in K2 outputs the following

Starting Name Server

```
Created: 2
Created: 3
Created: 4
Created: 5
Created: 6
Created: 7
Created: 8
Created: 9
Created: 10
Created: 11
bootstrap_k2: exiting
```

STARTING RPS Server

```
STARTING GAME WITH 4 5
STARTING GAME WITH 6 7
STARTING GAME WITH 8 9
STARTING GAME WITH 10 11
5 LOSE 4 WIN
ROUND FINISHED, PRESS ANY KEY TO CONTINUE
7 LOSE 6 WIN
ROUND FINISHED, PRESS ANY KEY TO CONTINUE
9 LOSE 8 WIN
```

```

ROUND FINISHED, PRESS ANY KEY TO CONTINUE
11 LOSE 10 WIN
ROUND FINISHED, PRESS ANY KEY TO CONTINUE
4 AND 5 TIED
ROUND FINISHED, PRESS ANY KEY TO CONTINUE
6 LOSE 7 WIN
ROUND FINISHED, PRESS ANY KEY TO CONTINUE
8 LOSE 9 WIN
ROUND FINISHED, PRESS ANY KEY TO CONTINUE
10 AND 11 TIED
ROUND FINISHED, PRESS ANY KEY TO CONTINUE
11 AND 10 TIED
ROUND FINISHED, PRESS ANY KEY TO CONTINUE
5 EXITING
4 EXITING
6 EXITING
7 EXITING
8 EXITING
9 EXITING
11 EXITING
10 EXITING

```

Exiting Main

Explanation

The kernel first bootstraps the Name Server with TID 2, then the RPS Server with TID 3. The RPS Server then registers its name with the name server.

We have 2 different RPS Clients, client 1 will signup, play 2 games, then quit. Client 2 will signup, play 3 games, then quit.

In order to cover all our test cases, we play 4 games, with different permutations of clients and the order in which they play. In order, we have Client 2 vs Client 1, Client 1 vs Client 2, Client 1 vs Client 1, Client 2 vs Client 2. These 4 games cover all the requirements of SIGNUP, PLAY, and QUIT.

The first “STARTING GAME WITH ...” outputs signifies that we are starting games concurrently, as New Clients will not be blocked from signing up as games between other clients are starting. Since the Clients are booted sequentially, we can determine who is playing who in our bootstrap code.

First, we have Client 2 with TID 4 playing against Client 1 with TID 5. 4 Wins and 5 loses. The “PRESS ANY KEY TO CONTINUE” line uses a blocking `uart_getc` so the games can be played slowly and with an interval in between. The results of the games are pseudorandom, we use the system timer to generate which of Rock, Paper, or Scissors each client will choose when it sends a play request by taking the modulo 3 of the CLO register.

Since the game result is only determined once the second player sends a PLAY request, whenever the 2 clients play consecutively, the order in which they play will be flipped. For example, when Client 1 plays first against Client 2, Client 2 will receive the Reply for the result first, and thus will send it’s next play request before Client 1’s next play request.

After all the games conclude, the players start quitting.

First, Task 5 (Client 1) sends a QUIT request to the server and the server replies acknowledging the quit, causing Task 5 to Exit. When Task 4 (Client 2) sends it’s third play request, the server replies with a quit signal instead, causing Task 4 to exit. Second, task 7 (Client 1) sends a PLAY request to the server and waits for task 6 (Client 2) to send a PLAY request. However, Task 6 sends a QUIT request instead, resulting in the Server replying to Task 6, then Task 7 that the game has ended, causing them to exit in order of the replies. Third, Task 8 and Task 9 (both Client 1) both send a Quit Request, causing them both to exit in the same order they Quit. Finally, Task 11 and Task 10 (both Client 2) both send a Quit Request, causing them both to exit in the same order they Quit.

Once all the Clients Exit, the RPS Server becomes Receive Blocked, and since there is nothing left to schedule, the kernel loop exits.

Performance

The performance measurements in a separate file, a brief explanation of the methodology, and your conclusions where in your code you think the time is being spent.

The Performance of Send Receive Reply is detailed in **performance.txt**.

Our test methodology was the one described in class, We would initialize 2 tasks, A sender task and a Receiver/Replying task.

In the Sender Task, we would send a message of the specified size 100,000 times, and take the average (floor division) time in microseconds it took for all 100,000 Sends to be replied to from the receiver task. The receiver task would just expect 100,000 Receives, reply immediately, and exit.

We used the System Timer for time periods in microseconds, retrieving the timestamp before and after the loop. We believe that 100,000 iterations of SRR, taking a few seconds to execute is enough to filter out the noise caused by looping, and retrieving the timestamp from the system timer.

Based on additional performance testing, on the highest compiler optimization levels, the majority of the time is spent in the context switch, and without compiler optimization, the majority of the time is spent in memcpy. SRR requires us to context switch 6 times (3 in 3 out) of the user task. Based on our tests on our context switch, switching in and out of the kernel takes $\sim 7\mu s$ On the Highest optimization level, this accounts for $\sim 21\mu s$ of the $\{29, 30, 38\}\mu s$ time. On the other hand, without compiler optimization, the context switch takes takes $\sim 13\mu s$, resulting in $\sim 39\mu s$ of the $\{117, 210, 510\}\mu s$ time required. This suggests that a lot more slow work is happening in the kernel related to memory, and the only operation we have that scales up with message size is memcpy.

When enabling the icache, we observed that the latency decreases by $\sim 10\mu s$ flat across all operations. Enabling the dcache did nothing.

K3

The output of our program is:

```
Bootstrap init
Created uart_server: 3
Created clock_server: 4                Idle Time: 97.56%
Created clock_notifier: 5
Created clock_client 1 tid 6
Created clock_client 2 tid 7
Created clock_client 3 tid 8
Created clock_client 4 tid 9
Created idle_server: 10
CLIENT WITH TID 6 STARTING AT: 1
CLIENT WITH TID 7 STARTING AT: 1
CLIENT WITH TID 8 STARTING AT: 2
Bootstrap Completed: Exiting
CLIENT WITH TID 9 STARTING AT: 2
Client with TID 6-Delay Interval: 10, Return from Delay at: 11 , Delay Number: 1 /20
Client with TID 6-Delay Interval: 10, Return from Delay at: 21 , Delay Number: 2 /20
Client with TID 7-Delay Interval: 23, Return from Delay at: 25 , Delay Number: 1 /9
Client with TID 6-Delay Interval: 10, Return from Delay at: 31 , Delay Number: 3 /20
Client with TID 8-Delay Interval: 33, Return from Delay at: 35 , Delay Number: 1 /6
Client with TID 6-Delay Interval: 10, Return from Delay at: 41 , Delay Number: 4 /20
Client with TID 7-Delay Interval: 23, Return from Delay at: 48 , Delay Number: 2 /9
Client with TID 6-Delay Interval: 10, Return from Delay at: 51 , Delay Number: 5 /20
Client with TID 6-Delay Interval: 10, Return from Delay at: 61 , Delay Number: 6 /20
Client with TID 8-Delay Interval: 33, Return from Delay at: 68 , Delay Number: 2 /6
Client with TID 6-Delay Interval: 10, Return from Delay at: 71 , Delay Number: 7 /20
Client with TID 7-Delay Interval: 23, Return from Delay at: 71 , Delay Number: 3 /9
Client with TID 9-Delay Interval: 71, Return from Delay at: 74 , Delay Number: 1 /3
Client with TID 6-Delay Interval: 10, Return from Delay at: 81 , Delay Number: 8 /20
Client with TID 6-Delay Interval: 10, Return from Delay at: 91 , Delay Number: 9 /20
Client with TID 7-Delay Interval: 23, Return from Delay at: 94 , Delay Number: 4 /9
Client with TID 6-Delay Interval: 10, Return from Delay at: 101, Delay Number: 10/20
Client with TID 8-Delay Interval: 33, Return from Delay at: 101, Delay Number: 3 /6
```

```

Client with TID 6-Delay Interval: 10, Return from Delay at: 111, Delay Number: 11/20
Client with TID 7-Delay Interval: 23, Return from Delay at: 117, Delay Number: 5 /9
Client with TID 6-Delay Interval: 10, Return from Delay at: 121, Delay Number: 12/20
Client with TID 6-Delay Interval: 10, Return from Delay at: 131, Delay Number: 13/20
Client with TID 8-Delay Interval: 33, Return from Delay at: 134, Delay Number: 4 /6
Client with TID 7-Delay Interval: 23, Return from Delay at: 140, Delay Number: 6 /9
Client with TID 6-Delay Interval: 10, Return from Delay at: 141, Delay Number: 14/20
Client with TID 9-Delay Interval: 71, Return from Delay at: 145, Delay Number: 2 /3
Client with TID 6-Delay Interval: 10, Return from Delay at: 151, Delay Number: 15/20
Client with TID 6-Delay Interval: 10, Return from Delay at: 161, Delay Number: 16/20
Client with TID 7-Delay Interval: 23, Return from Delay at: 163, Delay Number: 7 /9
Client with TID 8-Delay Interval: 33, Return from Delay at: 167, Delay Number: 5 /6
Client with TID 6-Delay Interval: 10, Return from Delay at: 171, Delay Number: 17/20
Client with TID 6-Delay Interval: 10, Return from Delay at: 181, Delay Number: 18/20
Client with TID 7-Delay Interval: 23, Return from Delay at: 186, Delay Number: 8 /9
Client with TID 6-Delay Interval: 10, Return from Delay at: 191, Delay Number: 19/20
Client with TID 8-Delay Interval: 33, Return from Delay at: 200, Delay Number: 6 /6
Client 8 Exiting
Client with TID 6-Delay Interval: 10, Return from Delay at: 201, Delay Number: 20/20
Client 6 Exiting
Client with TID 7-Delay Interval: 23, Return from Delay at: 209, Delay Number: 9 /9
Client 7 Exiting
Client with TID 9-Delay Interval: 71, Return from Delay at: 216, Delay Number: 3 /3
Client 9 Exiting

```

Explanation

We can analyze the output by looking at the messages of each client separately. There are a total of 20 lines for Client 6. Client 6 sends its starting message at tick 1 and delays 20 times for 10 ticks each time. This is consistent with our output.

The same applies for client 7. It starts at time 1. It has 9 print messages all spaced 23 ticks apart, consistent with the requirements. The first delay for client 7 seems to start on the 2nd tick. This makes sense as the client 8 starts on tick 2, so by the time client 7 gets rescheduled after its send to the print server, it is tick 2, so the return from delay is at tick 25.

The same analysis tells us the output for client 3 and 4 are also correct.

The ordering of the output is in ascending based on return time of the delay command. This is evident from the values in the "return from delay" column that tell us when the tasks have returned from their delay.

There are tasks that come back from the delay command at the same time at tick 71 and 101, in these cases, the client with the lower tid (higher priority) print first, consistent with the requirement.