CS 452 - K4

Team Members

Matthew Geng
m3geng
20820439

Erica Wang
e52wang
20825331

Repository: https://git.uwaterloo.ca/mg-cs452/cs452

Commit SHA: 7cc7e5c81cb4c7474f5036c7819a8038607f9f2c

UART I/O

The UART I/O is implemented with a server/notifier pattern. Getc(), Putc() are implemented as a wrapper around Send() with a custom msg object IOMessage. Getc() is blocking and Putc() is non-blocking.

Console I/O

Console I/O consists of 4 components, console_out_notifier, console_in_notifier, console_out_server, and console_in_server.

Notifiers

The notifiers are implemented as an infinite loop that waits on AwaitEvent() for their respective interrupts. Upon being awakened by an interrupt, it sends to the server.

Servers

The servers are implemented as an infinite loop that waits on Receive() from either a user task or its notifier. Each server has two circular buffers, one for the characters to be processed (get/put), and the other for the tasks waiting on the server. Furthermore, we implement a notifier "parking" mechanism where the server only replies to the notifier upon a Send() from a notifier if there are more characters to write/more tasks waiting on Getc(). Otherwise, there is no point in processing the interrupts until the user tasks are ready to use UART I/O again.

Console-Out Server

When the console out server receives from the console out notifier, it runs a while loop that pops a character from the char circular buffer to write to the console's dr register until the register is no longer ready to be written to.

When the console out server receives from a user task, it pushes the character(s) into the char circular buffer, immediately replies to the user task, then runs a while loop to attempt to print the characters to console out.

Console-In Server

When the console in server receives from the console in notifier, it runs a while loop that checks for register readiness, then reads the character from the dr register to push into the char circular buffer until the register is no longer ready to be read from.

When the console in server receives from a user task, it pushes the task into the task circular buffer then runs a while loop to try to pop and match the console-in characters with the waiting tasks. Upon matching a task with a character, the server replies to that task with the char.

Marklin I/O

Marklin I/O consists of 4 components, marklin_out_tx_notifier, marklin_out_cts_notifier, console_in_notifier, and marklin_server. There is only 1 server as opposed to the 2 for console I/O since Marklin is half-duplex.

Notifiers

The Marklin notifiers are implemented in the same manner as the console notifiers.

Servers

The Marklin server is implemented as a combined UART in and out server. It receives from all 3 notifiers as well as user tasks for both Get() and Put() commands.

The Putc() logic works similarly as the console-in server and notifier.

The Get() logic differs from console-out in order to accommodate Marklin's processing time. We do this by including a CTS notifier which informs the server of CTS value changes. After sending a character to Marklin, we do not send another character until a Tx interrupt firing as well as two CTS interrupts firing (indicating the value going down then back up). Thus, we can only print one character at a time to Marklin.

Train Control

Set Up

- Print UI set up to console
- Print 96 and 0xC0 to Marklin
- Send switch change commands to set up the initial switch configuration
- Create 3 tasks respectively for time display, sensor update, and user input processing

Time Display

Implemented as an infinite polling loop that updates the console displayed timer every 100ms.

Get the current system time with the timer Time() function at the beginning. In each run of the loop, process the system time to obtain minutes, seconds, and tenth of seconds. Subsequently, use the timer DelayUntil() function to delay until 100ms after the displayed time.

Sensor Update

Implemented as an infinite polling loop that queries Marklin for triggered sensors every 100ms.

Instantiate a circular buffer of size 12 that overwrites oldest values when we exceed the buffer size.

Get the current system time with the timer Time() function at the beginning. In each run of the loop, print a 0x85 to Marklin and run the blocking Getc() function 10 times and process the received byte until we have all the recently-triggered sensors. We push these new sensors into the sensor circular buffer. Then, if a new sensor is detected, we write all sensors in the circular buffer to the console in order. Subsequently, use the timer DelayUntil() function to delay until 100ms after the displayed time.

User Input

Implemented as an infinite polling loop that waits on console input.

Instantiate an int array that tracks last speed for all trains (currently supporting 100 trains).
Instantiate a regular buffer of size 20 to store user input (thus supporting 20 maximum characters in a user command).

Enter an infinite loop that begins with a blocking Getc(). Upon a regular keyboard character, push it into the input buffer and update the console input display. Upon a backspace, remove the last character from the buffer and update the console input display. Upon a new line character, we display the complete command on the console for reference, clear the buffer as well as the console input display, and attempt to execute the command.

To execute the command, first string-parse the command stored in the input buffer. If it is not valid, display a specific error message. If it is a “tr” or “sw” command, simply send the intended characters to Marklin to execute the command. If it is a reverse command, to accommodate the delay required, we offload the Marklin prints to a worker task by sending the train number to reverse and its last speed via Send().

The reverse worker task is implemented as an infinite polling loop that waits on Receive(). Upon receiving a reverse command, it will first set the train speed to 0, delay for **5s/6s** depending on the train speed, then issue the reverse (15) speed command and then set the train to its previous speed. Then it will wait on Receive() again.

Regarding the delay, it is indeed very long, especially for the slower trains. However, we noticed that for the fastest trains, those are the times necessary for the trains to completely stop. If we do not allow the train to completely stop and do not add a manual delay between the 15 command and the last speed command, the train will fail to reverse. In the future, we will adapt the reverse delay to fit our chosen train better.

We primarily tested this command execution structure for the case of running only 1 train. In the future when we need to control multiple trains with precision, we will likely have one command execution task for every train instead.

Known Bugs

At times, the user input is not displayed upon program start and thus the user is unable to enter input. It seems to be caused by Marklin not being reset. It can be fixed by resetting Marklin and restarting the Pi.

CS 452 - K3

Team Members

Matthew Geng
m3geng
20820439

Erica Wang
e52wang
20825331

Repository: https://git.uwaterloo.ca/mg-cs452/cs452

Commit SHA: 0f252948fa6d05aee1cbedf79825a5f4303bca11

Note regarding K2 comments:

Due to time constraints between the release of K2 results and the K3 deadline, we have not addressed the stylistic comments made by the TA for our K2 in this assignment. However, we will address the following points in the upcoming K4 submission:

- Use structs instead of 2d array in name server for mapping
- Please use more folders for organizing the code.
- kmain function in main.c is too long.
- Try to use Macros.

Regarding this comment: Please use DEBUG flag for debugging, we use the function uart_**d**printf() instead of the regular uart_printf() for debugging prints which in its implementation uses a DEBUG flag.

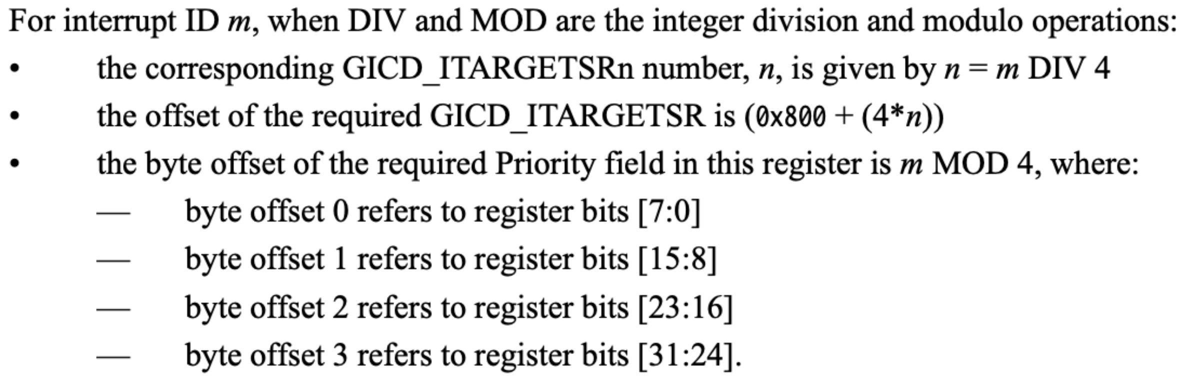
Interrupts

Set Up

To enable interrupts, we had to do 3 things.

IRQ Routing:

Based off of the documentation on page 107 of the GICv2 documentation, we created a function to route any specific IRQ id to a CPU. This is done through the following process.



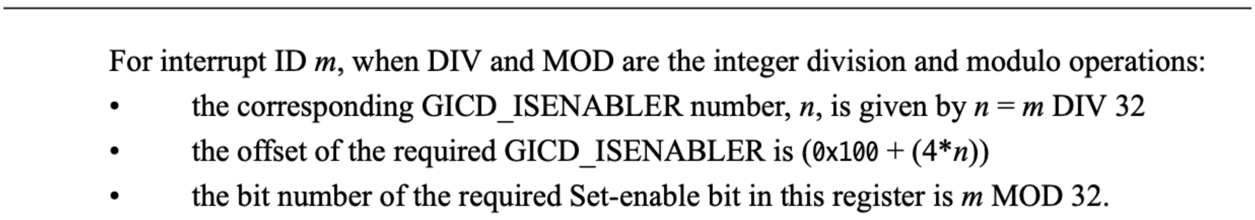
For interrupt ID m, when DIV and MOD are the integer division and modulo operations:

- the corresponding GICD_ITARGETSRn number, n, is given by $n = m$ DIV 4
- the offset of the required GICD_ITARGETSR is (`0x800` + (4*n))
- the byte offset of the required Priority field in this register is m MOD 4, where:
  - byte offset 0 refers to register bits [7:0]
  - byte offset 1 refers to register bits [15:8]
  - byte offset 2 refers to register bits [23:16]
  - byte offset 3 refers to register bits [31:24].

However, since we are only using a single CPU, the generic aspect of the function is not fully utilized.

IRQ Enabling:

Similar to IRQ routing, a generic function to enable specific IRQs from their id was implemented. Based on page 93 in the GICv2 documentation, the following process was created.



For interrupt ID m, when DIV and MOD are the integer division and modulo operations:

- the corresponding GICD_ISENABLER number, n, is given by $n = m$ DIV 32
- the offset of the required GICD_ISENABLER is (`0x100` + (4*n))
- the bit number of the required Set-enable bit in this register is m MOD 32.

SPSR IRQ Masking:

Lastly, to ensure our CPU doesn't ignore interrupts, we unmask the IRQ bit in the SPSR of our EL_0 tasks. Finally, this allows our EL_0 tasks to be interrupted by an interrupt and caught in our EL_1 kernel code.

System Timer

With the interrupt infrastructure setup, we setup our system timer to create a timer interrupt every 10ms.

First, we route and enable the system timer irq with an id of 97 which corresponds to the C1 register of the system timer compare register.

We then initialize the system timer by setting the C1 register to a value of 10ms + current time. This will cause an interrupt to fire after 10ms has happened in EL_0 space.

Handling the Interrupt:

Using our exception vector table, we differentiate a system call vs an interrupt through a different return value. Upon jumping back to our handler (run_task), we return the exception or a generic interrupt code to our kmain loop. This is done so as to not acknowledge the specific type of interrupt until needed.

Once in our main loop, we catch the generic interrupt code and acknowledge the interrupt while getting the ID of the interrupt thrown with **GICC_IAR**. Once we are sure of the system timer interrupt ID, we do the following in specific order:

- Update the **C1** register to be 10ms + current time.
- Clear the **CS** status register to make sure another interrupt isn't thrown immediately from an old **C1** comparison equal value
- Confirm the interrupt id through **GICC_EOIR**

AwaitEvent()

In our current implementation, we assume that only one task would be waiting on an event. Thus, we have an array of TaskFrames with length of the total number of interrupt events. If that were to change in the future, we would make that into a 2D array or equivalent struct.

Upon a task calling AwaitEvent(), the kernel stores the pointer to its TaskFrame in the aforementioned array until the event or interrupt arrives. If another task attempts to wait on the same event, we return an error code for that task's function call. Once an interrupt does occur, we check the type and attempt to unblock/reschedule the task currently waiting on that interrupt type.

Clock

Clock Notifier

The clock notifier first queries the name server to register itself as the clock notifier and find the tid of the clock server.

The clock notifier is implemented as a polling loop that waits on AwaitEvent() for the clock event at the beginning of the loop. Upon AwaitEvent() returning, it sends to the clock server notifying it of a time increment.

Clock Server

The clock server first queries the name server to register itself as the clock server and find the tid of the clock notifier.

DelayedTask is a struct that stores the tid and time it needs to delay until. We stack-allocate 20 instances of the object for the 20 active running tasks that our code currently accommodates. It is implemented with intrinsic linkage by including a next pointer. We also allocate a heap object to store all currently delayed tasks sorted by time to delay until.

The server is implemented as a polling loop that waits on Receive() at the beginning of the loop. It can either receive a message from the clock notifier or a task.

Upon receiving from the notifier, the clock server runs heap_peek() to get the next task to be released from the delay. If that time is greater than or equal to the current time (ticks), the clock server replies to the task to allow it to continue running and subsequently checks the sorted next delayed task until it is not ready to be released yet. It finally replies to the notifier task with the current time to allow it to continue.

Upon receiving from a task:

- If it is a Time() command, get the current system time and reply to the task with it.
- If it is a Delay() or DelayUntil() command, get the next free DelayedTask instance, populate the tid field and the delay_until fields accordingly and push it into the heap.

Send() Syscall Wrappers

Time(tid)

Send a message to the tid with a length 1 char array “t” for time. Upon returning from a reply, sanity-check the time tick received and return it. If the TID specified was invalid or the send didn’t go through we return -1.

Delay(tid, ticks)

Allocate a char array of length 5 with the first char as “d” and the int tick stored in the next 4 chars. Send a message to the tid with the char array. Upon returning from a reply, sanity-check the time tick received and return it. If the TID specified was invalid or the send didn’t go through we return -1. If the delay was negative, we return -2.

DelayUntil(tid, ticks)

Allocate a char array of length 5 with the first char as “u” and the int tick stored in the next 4 chars. Send a message to the tid with the char array. Upon returning from a reply, sanity-check the time tick received and return it. If the TID specified was invalid or the send didn’t go through we return -1. If the delay was negative, we return -2.

Idle Task

The idle task is Implemented as an infinite loop that prints out the idle usage and runs the assembly command “wfi” to put the CPU on low power mode. To have the idle percentage not change with other task prints, in our idle task, we store the cursor position, print our idle ratio, and restore the cursor position.

```
uart_printf(CONSOLE, "\0337\033[3;1HIdle percentage: %u%% \0338", idle_time_percent);
```

Performance Indicator

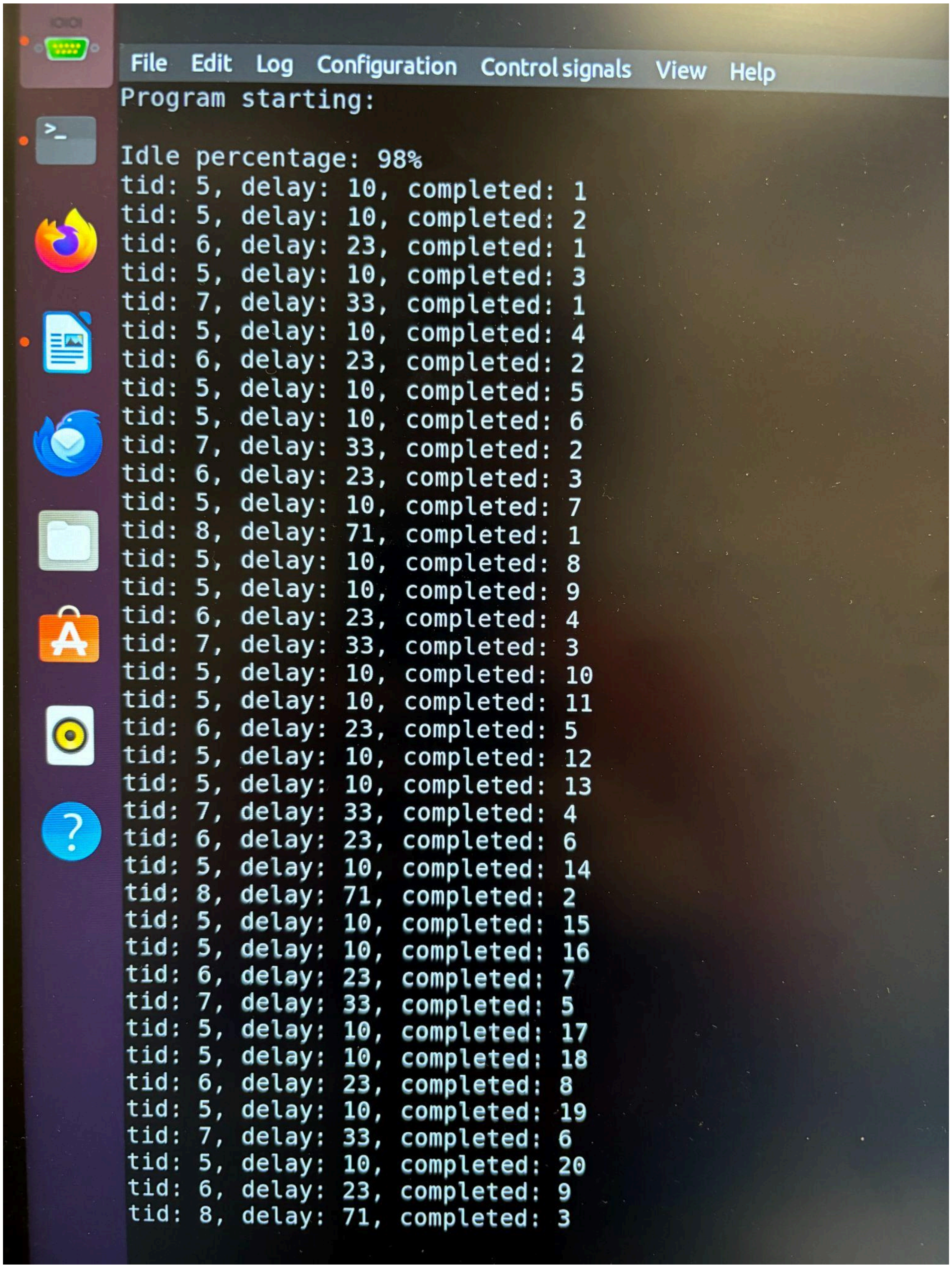
As mentioned by the professor in class, this indicator requires a “hack” between the kernel and the user task. We interpreted this as the idle task having to access kernel memory in order to print the ratio of idle task runtime so far. We store two pointers to uint64_t as global variables: p_idle_task_total and p_program_start. The uint values are set in the kernel and read by the idle task to compute the idle percentage which equates (*p_idle_task_total)/(sys_time() - *p_program_start) * 100.

The program start is set as the system time at the beginning of the program. To obtain the total idle task runtime, we store a start time whenever the idle task is scheduled as the next task and when the idle task returns to kernel due to an interrupt, we increment the total idle task runtime by the difference between the current system time and the idle task start time.

Change in WhoIs() Implementation

Previously, if the name server encounters a WhoIs() for a task that does not exist in its mapping, it presumes an error. However, while implementing the clock server and notifier, we run into an issue where both tasks registers itself and subsequently queries the name server for the other. Thus, querying for a name that doesn't exist yet becomes a valid scenario. To accommodate this, we modify the WhoIs() implementation to block a task upon it calling WhoIs() until the name appears in the mapping as a result of a RegsiterAs().

Output Explanation:



```
Program starting:

Idle percentage: 98%
tid: 5, delay: 10, completed: 1
tid: 5, delay: 10, completed: 2
tid: 6, delay: 23, completed: 1
tid: 5, delay: 10, completed: 3
tid: 7, delay: 33, completed: 1
tid: 5, delay: 10, completed: 4
tid: 6, delay: 23, completed: 2
tid: 5, delay: 10, completed: 5
tid: 5, delay: 10, completed: 6
tid: 7, delay: 33, completed: 2
tid: 6, delay: 23, completed: 3
tid: 5, delay: 10, completed: 7
tid: 8, delay: 71, completed: 1
tid: 5, delay: 10, completed: 8
tid: 5, delay: 10, completed: 9
tid: 6, delay: 23, completed: 4
tid: 7, delay: 33, completed: 3
tid: 5, delay: 10, completed: 10
tid: 5, delay: 10, completed: 11
tid: 6, delay: 23, completed: 5
tid: 5, delay: 10, completed: 12
tid: 5, delay: 10, completed: 13
tid: 7, delay: 33, completed: 4
tid: 6, delay: 23, completed: 6
tid: 5, delay: 10, completed: 14
tid: 8, delay: 71, completed: 2
tid: 5, delay: 10, completed: 15
tid: 5, delay: 10, completed: 16
tid: 6, delay: 23, completed: 7
tid: 7, delay: 33, completed: 5
tid: 5, delay: 10, completed: 17
tid: 5, delay: 10, completed: 18
tid: 6, delay: 23, completed: 8
tid: 5, delay: 10, completed: 19
tid: 7, delay: 33, completed: 6
tid: 5, delay: 10, completed: 20
tid: 6, delay: 23, completed: 9
tid: 8, delay: 71, completed: 3
```

Inspecting the delays and the completion number, we can verify that the above output is logically correct. For example, we always see at least two 10 delays from tid 5 before a delay of 23 from tid 6 and three 10 delays from tid 5 before a delay of 33 from tid 7. Additionally, by adding the delays up in order, we can see that the specified tasks are printing their delays in increasing order.
Tid 5 completes two 10 delays before tid 6's delay of 23.
Another 10 delay from tid5, and we should be at ~30 delay. At this point, tid 7 should be and is the next task to print at delay 33. We can continue this analysis until the end of the program.

Furthermore, our idle percentage is ~98% which is the correct value given that we are not running any complex or intensive work. All that is happening are delays, send/recieves, and our clock interrupt handling. As a result, we should expect our idle percentage to be > 90% and ours is at 98%.

Appendix:

CS 452 - K2

Team Members

Matthew Geng
m3geng
20820439

Erica Wang
e52wang
20825331

Repository: https://git.uwaterloo.ca/mg-cs452/cs452

Commit SHA for RPS: 481abeb6112aa02a653ab977ffb5308ba6ac981b

Commit SHA for measurements: d80ede9b862737e2e3aaeb52f30ccc3e95a92ba7

Send-Receive-Reply

The send-receive-reply is implemented as explained in the lecture such that data flows when both sender and receiver are ready and there is no asynchronous buffering.

Design Choices

- For implementation, we added four new fields in our task descriptor: status, SendData, ReceiveData, and a sender queue of TIDs.
- There are five valid statuses: inactive, ready, send-wait, receive-wait, and reply-wait.
- The SendData object stores the five parameters of the Send() function and the ReceiveData object stores the 3 parameters of the Receive() function. Furthermore, to effectively allocate and reclaim SendData and ReceiveData instances, they are implemented with intrinsic linkage such that the struct is defined to have a pointer to the next object.
- The sender queue is implemented as a circular buffer to ensure O(1) push and pop operation and reuse of memory

Send

When a user task returns to kernel with a send syscall, here are the action the kernel performs

- Verify that the recipient TID is an active user task by checking the status flag of the task descriptor. If it's not, reschedule the current task with a -1 return.
- Verify that the sender status is ready and recipient task status is either ready or receive-wait. If it's not, reschedule the current task with a -2 return.
- Get the next free SendData object and populate the fields with the syscall arguments
- Add a pointer to this SendData object to the current task's descriptor
- If the recipient is in ready state, or in other words, doing something else,
    - Add the current task into the sender queue of the recipient task
    - Set the status to send-wait
- If the recipient is in receive-wait,
    - Do some sanity checks
    - Set the TID pointer of the recipient's ReceiveData to the current task's TID and copy the message from the send buffer to the receive buffer
    - Reclaim the ReceiveData object
    - Set the sender status to reply-wait and the recipient status to ready and reschedule the recipient task.

Receive

When a user task returns to kernel with a receive syscall, here are the action the kernel performs

- Perform sanity checks to ensure current task is in ready status
- If there is no tasks in the current task's sender queue,
    - Get the next free ReceiveData object and populate the fields with the syscall arguments
    - Add a pointer to this ReceiveData object to the current task's descriptor
    - Set the current task's status to receive-wait
- If there is one or more task in the sender queue,
    - Pop the next sender
    - Sanity check the TID and task status
    - Get sender task's task descriptor from the TID
    - Copy the data from send buffer to receive buffer
    - Set the current task's status to ready and the sender task's status to reply-wait
    - Reschedule the current task

Reply

When a user task returns to kernel with a receive syscall, here are the action the kernel performs

- Perform sanity check on the TID and both tasks' statuses
- Get sender task's task descriptor from the TID
- Copy data from the replier's data buffer to the sender's reply data buffer
- Set sender task's status to ready (and replier's status stays ready)
- Reschedule both tasks

Name Server

In order to allow user-tasks to determine their own identifier as opposed to the system designated TID, we implement a name server that resolves TID to user-defined names and vice versa using two functions.

Design Choices

- As a default starting point, we assume that the name server's TID is always 1, or in other words, it is always the first task being scheduled.
- When RegisterAs() encounters a name that already exists in the mapping, we overwrite it and the previous TID associated to the name now has no name registered.
- When WhoIs() is called with an unregistered name, we assume that this is a result of a bug. Thus the program logs an error and halts.
- User tasks are not expected to register or query for names that are null-terminated but the names stored in the name server mapping will always be null-terminated.
- The maximum name length is set to 20 characters. If a user task attempts to register or query for a name longer than that, the system will log an error and the name in the resulting mapping will get truncated but will still be null-terminated. Thus, assuming there doesn't exist another name with the same first 20 characters, the functionality should not be affected by the truncation.
- Under our current assumption of there never being more than 20 tasks running simultaneously, we have decided that the name server will not benefit greatly from having a O(1) query time data structure such as a hashmap. Instead, we have an array of length 20 (max number of simultaneously active tasks) that map to the corresponding names.

Name Server Task

The name server task is designed to be a task that infinitely waits to receive queries from other tasks and never exits.
It first stack-allocates 2D array of size 20 (maximum number of simultaneously active tasks) by 20 (maximum size of task name). All names are initialized to "\0". It then enters a polling loop starting with a Receive function. If the intended message sent is greater than the maximum size of task name defined, the string is truncated to length 20 and terminated with a null character.

If the message is received with a starting character of “r” (for RegisterAs),

- Run str_cmp on every existing name in the mapping to check if the name in the message already exists. If it does, overwrite it to “\0”.
    - Although this is an inefficient operation, we have decided that RegisterAs() will only be called once per task. Thus, the maximum number of operations is 20 (max # of tasks that will call RegsterAs) * 20 (length of mapping array) * 20 (max length of task names) = 8000 which is insignificant at runtime.
- Linearly set the name mapped to the current task’s TID to the name in message, terminated by a null character.
- Reply with a buffer of size 0.

If the message is received with a starting character of “w” (for WhoIs),

- Run str_cmp on every existing name in the mapping to find the TID that is registered under the name in the message.
- If the name is not found, reply with a buffer of size 0 indicating an error.
- If the name is found, stack-allocate a buffer of size 1 storing the resulting TID as a char.

RegisterAs()

RegisterAs() takes one parameter of the name the current task is to be registered as. It then stack-allocates another char buffer of size 1+len(name), sets the first character to ‘r’ (representing a RegisterAs call to the name server) then linearly copies the name into the new buffer.
Although this is an inefficient operation, we have decided that RegisterAs() will only be called once per task. Thus, the maximum number of operations is 20(max # of tasks) * 20(max length of task names) = 400 which is insignificant at runtime.
We then make a Send syscall with this new char array as the msg buffer and no reply buffer. Send will then return the intended reply length. If the reply length is not 0, we log the error and halts since the name server is designed to reply with a char array of size 0 upon RegisterAs(). If the Send syscall failed with a return value of <0, RegisterAs() also returns -1 indicating a fail.

WhoIs()

WhoIs() takes one parameter of the task name to be queried. It then stack-allocates another char buffer of size 1+len(name), sets the first character to ‘w’ (representing a WhoIs call to the name server) then linearly copies the name into the new buffer.
Although this is an inefficient operation, we have decided that WhoIs() will be called at maximum by each task for every other task. Thus, the maximum number of operations is 20(max # of tasks) *19(max # of other currently active tasks) * 20(max length of task names) = 7600 which is insignificant at runtime.
We then make a Send syscall with this new char array as the msg buffer and a reply buffer of size 1 which should be filled with the TID value as a char. Send will then return the intended

reply length. If the reply length is not 1, we log the error and halts since the name server is designed to reply with a char array of size 1 upon WhoIs().
If the Send syscall failed with a return value of <0, RegisterAs() also returns -1 indicating a fail.

Game Server

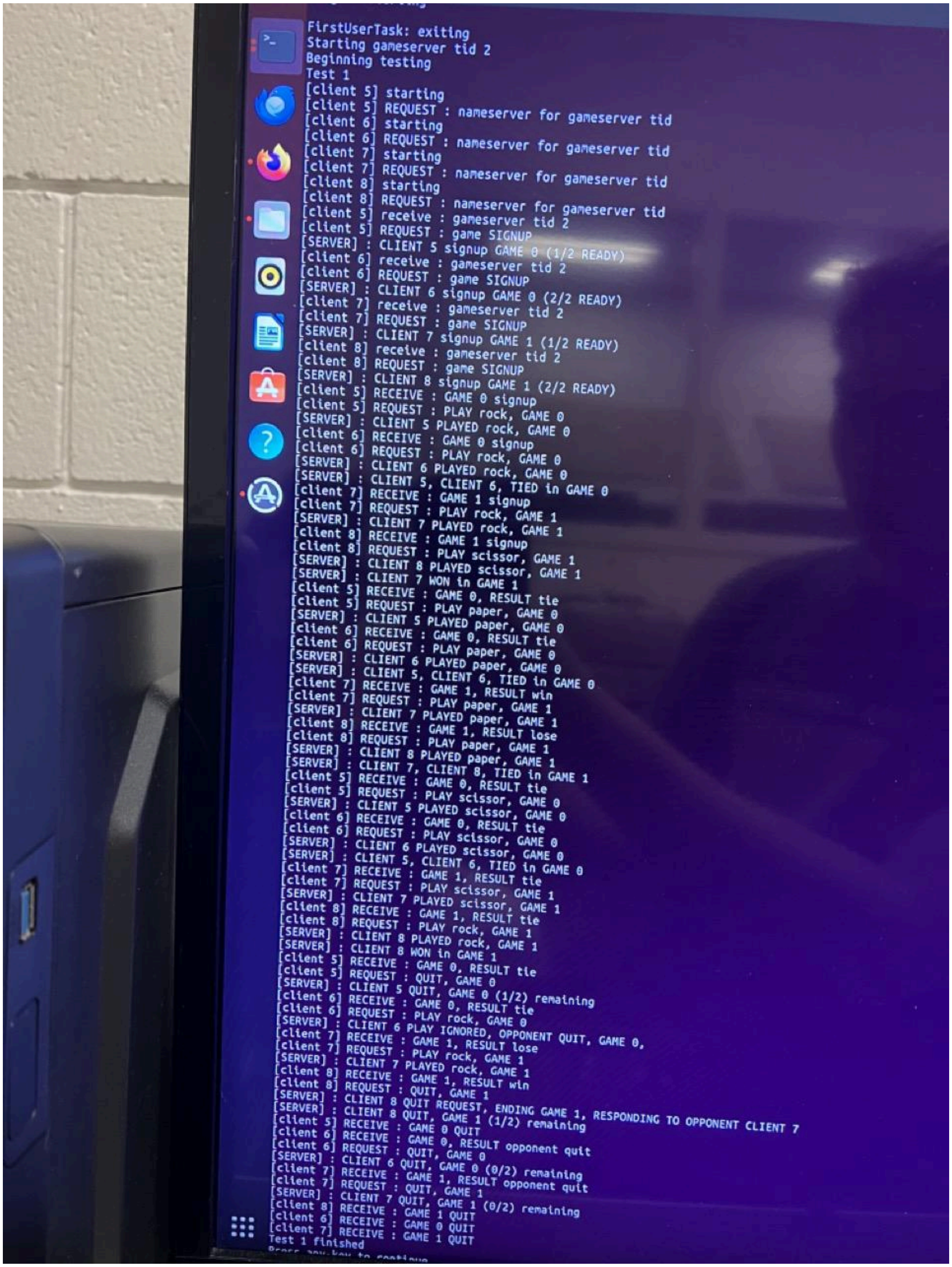
The game server is an infinite loop that receives messages from the game clients and acts on them respectively. There is a max of 5 games and when games are done, they are recycled to allow future players to use an old game. Each of the tests are a user task at a higher priority than the game clients, allowing all the game clients to be created first. However, since each client first calls MyTid(), it may be preempted before the “[client x] starting” print statement.

RPS Game Test Results

Test 1:

```
Create(3, &rps_quit); //  task 5
Create(3, &rps_move_quit); // task 6
Create(3, &rps_move_quit); // task 7
Create(3, &spr_quit); // task 8
```

In the first test, we created 4 clients all at the same priority. At a high level, the first 2 clients (5, 6) always tie with each other by playing ROCK, PAPER, SCISSORS. However, the first will attempt to quit and then second will attempt to play first and then quit. The last 2 clients (7,8) win, tie, and lose respectively where the first client plays ROCK, PAPER. SCISSORS and the last client plays SCISSORS, PAPER, ROCK. Afterward, Client 5 quits first, causing 6 to receive a quit result and later quit. On the other hand, 8 quits after 7 makes a final move. The server then responds to 7, indicating 8 quit, allowing 7 and 8 to quit.



```
FirstUserTask: exiting
Starting gameserver tid 2
Beginning testing
Test 1
[client 5] starting
[client 5] REQUEST : nameserver for gameserver tid
[client 6] starting
[client 6] REQUEST : nameserver for gameserver tid
[client 7] starting
[client 7] REQUEST : nameserver for gameserver tid
[client 8] starting
[client 8] REQUEST : nameserver for gameserver tid
[client 5] receive : gameserver tid 2
[client 5] REQUEST : game SIGNUP
[SERVER] : CLIENT 5 signup GAME 0 (1/2 READY)
[client 6] receive : gameserver tid 2
[client 6] REQUEST : game SIGNUP
[SERVER] : CLIENT 6 signup GAME 0 (2/2 READY)
[client 7] receive : gameserver tid 2
[client 7] REQUEST : game SIGNUP
[SERVER] : CLIENT 7 signup GAME 1 (1/2 READY)
[client 8] receive : gameserver tid 2
[client 8] REQUEST : game SIGNUP
[SERVER] : CLIENT 8 signup GAME 1 (2/2 READY)
[client 5] RECEIVE : GAME 0 signup
[client 5] REQUEST : PLAY rock, GAME 0
[SERVER] : CLIENT 5 PLAYED rock, GAME 0
[client 6] RECEIVE : GAME 0 signup
[client 6] REQUEST : PLAY rock, GAME 0
[SERVER] : CLIENT 6 PLAYED rock, GAME 0
[SERVER] : CLIENT 5, CLIENT 6, TIED in GAME 0
[client 7] RECEIVE : GAME 1 signup
[client 7] REQUEST : PLAY rock, GAME 1
[SERVER] : CLIENT 7 PLAYED rock, GAME 1
[client 8] RECEIVE : GAME 1 signup
[client 8] REQUEST : PLAY scissor, GAME 1
[SERVER] : CLIENT 8 PLAYED scissor, GAME 1
[SERVER] : CLIENT 7 WON in GAME 1
[client 5] RECEIVE : GAME 0, RESULT tie
[client 5] REQUEST : PLAY paper, GAME 0
[SERVER] : CLIENT 5 PLAYED paper, GAME 0
[client 6] RECEIVE : GAME 0, RESULT tie
[client 6] REQUEST : PLAY paper, GAME 0
[SERVER] : CLIENT 6 PLAYED paper, GAME 0
[SERVER] : CLIENT 5, CLIENT 6, TIED in GAME 0
[client 7] RECEIVE : GAME 1, RESULT win
[client 7] REQUEST : PLAY paper, GAME 1
[SERVER] : CLIENT 7 PLAYED paper, GAME 1
[client 8] RECEIVE : GAME 1, RESULT lose
[client 8] REQUEST : PLAY paper, GAME 1
[SERVER] : CLIENT 8 PLAYED paper, GAME 1
[SERVER] : CLIENT 7, CLIENT 8, TIED in GAME 1
[client 5] RECEIVE : GAME 0, RESULT tie
[client 5] REQUEST : PLAY scissor, GAME 0
[SERVER] : CLIENT 5 PLAYED scissor, GAME 0
[client 6] RECEIVE : GAME 0, RESULT tie
[client 6] REQUEST : PLAY scissor, GAME 0
[SERVER] : CLIENT 6 PLAYED scissor, GAME 0
[SERVER] : CLIENT 5, CLIENT 6, TIED in GAME 0
[client 7] RECEIVE : GAME 1, RESULT tie
[client 7] REQUEST : PLAY scissor, GAME 1
[SERVER] : CLIENT 7 PLAYED scissor, GAME 1
[client 8] RECEIVE : GAME 1, RESULT tie
[client 8] REQUEST : PLAY rock, GAME 1
[SERVER] : CLIENT 8 PLAYED rock, GAME 1
[SERVER] : CLIENT 8 WON in GAME 1
[client 5] RECEIVE : GAME 0, RESULT tie
[client 5] REQUEST : QUIT, GAME 0
[SERVER] : CLIENT 5 QUIT, GAME 0 (1/2) remaining
[client 6] RECEIVE : GAME 0, RESULT tie
[client 6] REQUEST : PLAY rock, GAME 0
[SERVER] : CLIENT 6 PLAY IGNORED, OPPONENT QUIT, GAME 0,
[client 7] RECEIVE : GAME 1, RESULT lose
[client 7] REQUEST : PLAY rock, GAME 1
[SERVER] : CLIENT 7 PLAYED rock, GAME 1
[client 8] RECEIVE : GAME 1, RESULT win
[client 8] REQUEST : QUIT, GAME 1
[SERVER] : CLIENT 8 QUIT REQUEST, ENDING GAME 1, RESPONDING TO OPPONENT CLIENT 7
[SERVER] : CLIENT 8 QUIT, GAME 1 (1/2) remaining
[client 5] RECEIVE : GAME 0 QUIT
[client 6] RECEIVE : GAME 0, RESULT opponent quit
[client 6] REQUEST : QUIT, GAME 0
[SERVER] : CLIENT 6 QUIT, GAME 0 (0/2) remaining
[client 7] RECEIVE : GAME 1, RESULT opponent quit
[client 7] REQUEST : QUIT, GAME 1
[SERVER] : CLIENT 7 QUIT, GAME 1 (0/2) remaining
[client 8] RECEIVE : GAME 1 QUIT
[client 6] RECEIVE : GAME 0 QUIT
[client 7] RECEIVE : GAME 1 QUIT
Test 1 finished
Press any key to continue
```

At the beginning, all 4 clients query our nameserver for the game server tid. Since all 4 clients are at the same priority, they act like a FIFO queue due to using the current time as a tiebreak.

After, 5 and 6 are registered in game 0, 7 and 8 are registered in the next game. All clients begin playing. 5 and 6 always tie with both choosing ROCK, PAPER, SCISSOR in each respective round. 7 and 8 choose ROCK, SCISSOR in the first round with a win and lose result respectively. Next round they play PAPER, PAPER resulting in a tie. In the last round, they play SCISSOR, ROCK with a lose and win result. After the final rounds for each pair of clients, they both attempt to quit.

For 5 and 6, 5 requests to quit, and 6 requests to play ROCK. The server receives 5's request to quit responds, and then sees 6's request to play. The server then responds to 6 indicating that 5 quit.

In opposite order of 5 and 6, 7 requests to play, and 8 requests to quit. The server receives 7's request to play then sees 8's request to quit. The server allows 8 to quit and then responds to 7 indicating that 8' quit.

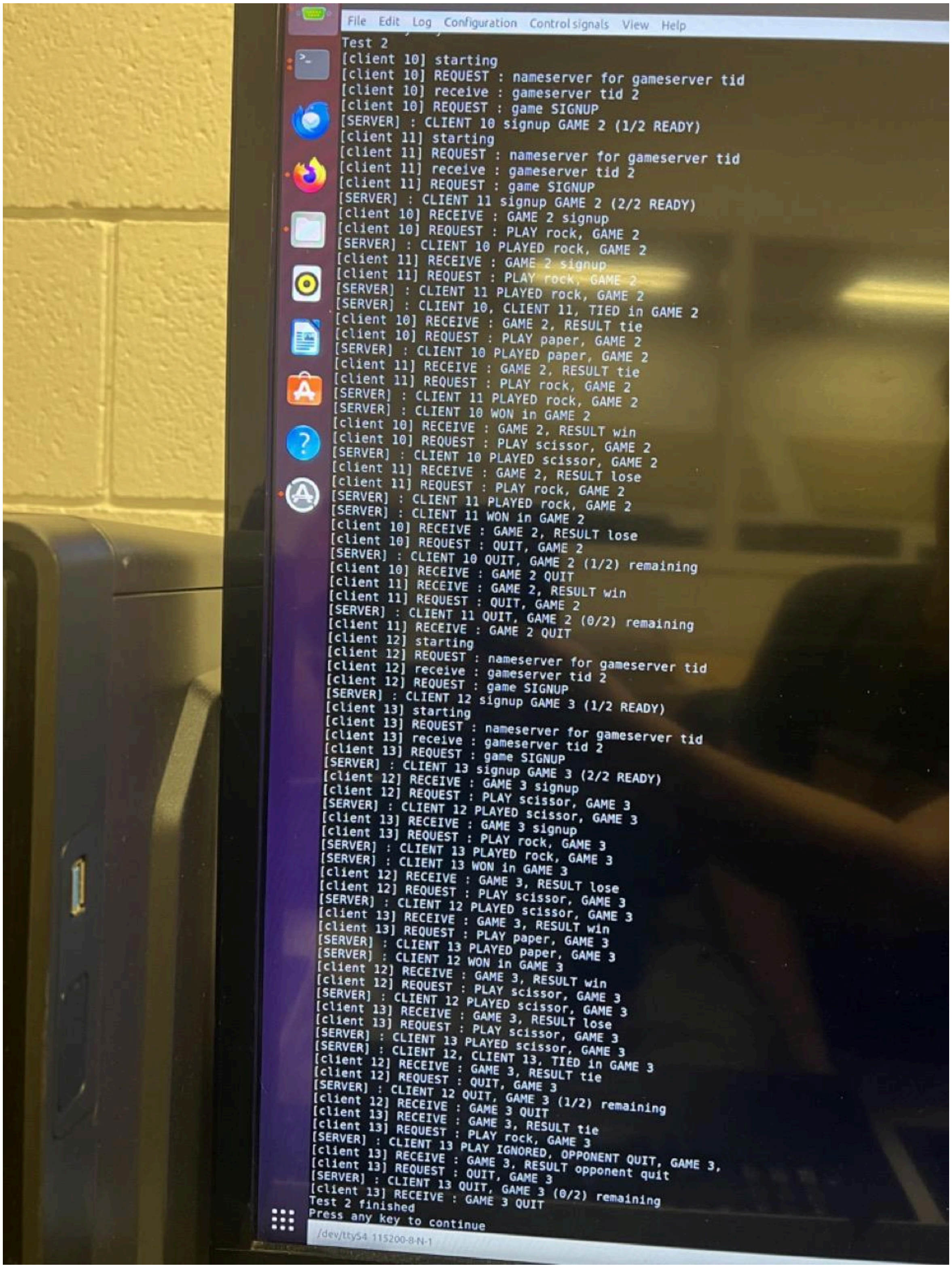
Eventually, both 6 and 7 also quit after being notified that their partner quit.

Test 2:

```
Create(4, &rps_quit); // task 10
Create(5, &rock_quit); // task 11
Create(6, &scissor_quit); // task 12
Create(7, &rps_move_quit); // task 13
```

In the second test, we have 4 tasks in decreasing priority levels. Task 10 will play ROCK, PAPER, SCISSOR at the highest priority level, task 11 will only play ROCK at a lower priority. Task 12 will only play SCISSOR at a lower priority and Task 13 will play ROCK, PAPER, SCISSOR, ROCK.

Given the priority scheme, tasks 10 and 11 will play completely until they are finished. At which the lower priority clients, 12 and 13 will then play their game.



```
Test 2
[client 10] starting
[client 10] REQUEST : nameserver for gameserver tid
[client 10] receive : gameserver tid 2
[client 10] REQUEST : game SIGNUP
[SERVER] : CLIENT 10 signup GAME 2 (1/2 READY)
[client 11] starting
[client 11] REQUEST : nameserver for gameserver tid
[client 11] receive : gameserver tid 2
[client 11] REQUEST : game SIGNUP
[SERVER] : CLIENT 11 signup GAME 2 (2/2 READY)
[client 10] RECEIVE : GAME 2 signup
[client 10] REQUEST : PLAY rock, GAME 2
[SERVER] : CLIENT 10 PLAYED rock, GAME 2
[client 11] RECEIVE : GAME 2 signup
[client 11] REQUEST : PLAY rock, GAME 2
[SERVER] : CLIENT 11 PLAYED rock, GAME 2
[SERVER] : CLIENT 10, CLIENT 11, TIED in GAME 2
[client 10] RECEIVE : GAME 2, RESULT tie
[client 10] REQUEST : PLAY paper, GAME 2
[SERVER] : CLIENT 10 PLAYED paper, GAME 2
[client 11] RECEIVE : GAME 2, RESULT tie
[client 11] REQUEST : PLAY rock, GAME 2
[SERVER] : CLIENT 11 PLAYED rock, GAME 2
[SERVER] : CLIENT 10 WON in GAME 2
[client 10] RECEIVE : GAME 2, RESULT win
[client 10] REQUEST : PLAY scissor, GAME 2
[SERVER] : CLIENT 10 PLAYED scissor, GAME 2
[client 11] RECEIVE : GAME 2, RESULT lose
[client 11] REQUEST : PLAY rock, GAME 2
[SERVER] : CLIENT 11 PLAYED rock, GAME 2
[SERVER] : CLIENT 11 WON in GAME 2
[client 10] RECEIVE : GAME 2, RESULT lose
[client 10] REQUEST : QUIT, GAME 2
[SERVER] : CLIENT 10 QUIT, GAME 2 (1/2) remaining
[client 10] RECEIVE : GAME 2 QUIT
[client 11] RECEIVE : GAME 2, RESULT win
[client 11] REQUEST : QUIT, GAME 2
[SERVER] : CLIENT 11 QUIT, GAME 2 (0/2) remaining
[client 11] RECEIVE : GAME 2 QUIT
[client 12] starting
[client 12] REQUEST : nameserver for gameserver tid
[client 12] receive : gameserver tid 2
[client 12] REQUEST : game SIGNUP
[SERVER] : CLIENT 12 signup GAME 3 (1/2 READY)
[client 13] starting
[client 13] REQUEST : nameserver for gameserver tid
[client 13] receive : gameserver tid 2
[client 13] REQUEST : game SIGNUP
[SERVER] : CLIENT 13 signup GAME 3 (2/2 READY)
[client 12] RECEIVE : GAME 3 signup
[client 12] REQUEST : PLAY scissor, GAME 3
[SERVER] : CLIENT 12 PLAYED scissor, GAME 3
[client 13] RECEIVE : GAME 3 signup
[client 13] REQUEST : PLAY rock, GAME 3
[SERVER] : CLIENT 13 PLAYED rock, GAME 3
[SERVER] : CLIENT 13 WON in GAME 3
[client 12] RECEIVE : GAME 3, RESULT lose
[client 12] REQUEST : PLAY scissor, GAME 3
[SERVER] : CLIENT 12 PLAYED scissor, GAME 3
[client 13] RECEIVE : GAME 3, RESULT win
[client 13] REQUEST : PLAY paper, GAME 3
[SERVER] : CLIENT 13 PLAYED paper, GAME 3
[SERVER] : CLIENT 12 WON in GAME 3
[client 12] RECEIVE : GAME 3, RESULT win
[client 12] REQUEST : PLAY scissor, GAME 3
[SERVER] : CLIENT 12 PLAYED scissor, GAME 3
[client 13] RECEIVE : GAME 3, RESULT lose
[client 13] REQUEST : PLAY scissor, GAME 3
[SERVER] : CLIENT 13 PLAYED scissor, GAME 3
[SERVER] : CLIENT 12, CLIENT 13, TIED in GAME 3
[client 12] RECEIVE : GAME 3, RESULT tie
[client 12] REQUEST : QUIT, GAME 3
[SERVER] : CLIENT 12 QUIT, GAME 3 (1/2) remaining
[client 12] RECEIVE : GAME 3 QUIT
[client 13] RECEIVE : GAME 3, RESULT tie
[client 13] REQUEST : PLAY rock, GAME 3
[SERVER] : CLIENT 13 PLAY IGNORED, OPPONENT QUIT, GAME 3,
[client 13] RECEIVE : GAME 3, RESULT opponent quit
[client 13] REQUEST : QUIT, GAME 3
[SERVER] : CLIENT 13 QUIT, GAME 3 (0/2) remaining
[client 13] RECEIVE : GAME 3 QUIT
Test 2 finished
Press any key to continue
```

At the beginning, task 1 queries for the tid of game server and then blocks on waiting for a reply from the game server. Task 1 is able to execute all the way until blocking on a reply due to its high priority. Afterward, task 2 does the same as task 1 and registers to the game server. At this point, the game server has 2 valid clients and responds to them both allowing them to execute their play. Since these 2 clients have higher priority than the others, clients 12 and 13 wait until they are completely done before starting their game. For clients 10 and 11's games, we expect ROCK, ROCK in round 1 leading to a tie. PAPER, ROCK in round 2 leading to a win, lose respectively. In the last round, we expect SCISSORS, ROCK leading to a lose, win again. In the end, both quit at the same time and we see this illustrated in the above output.

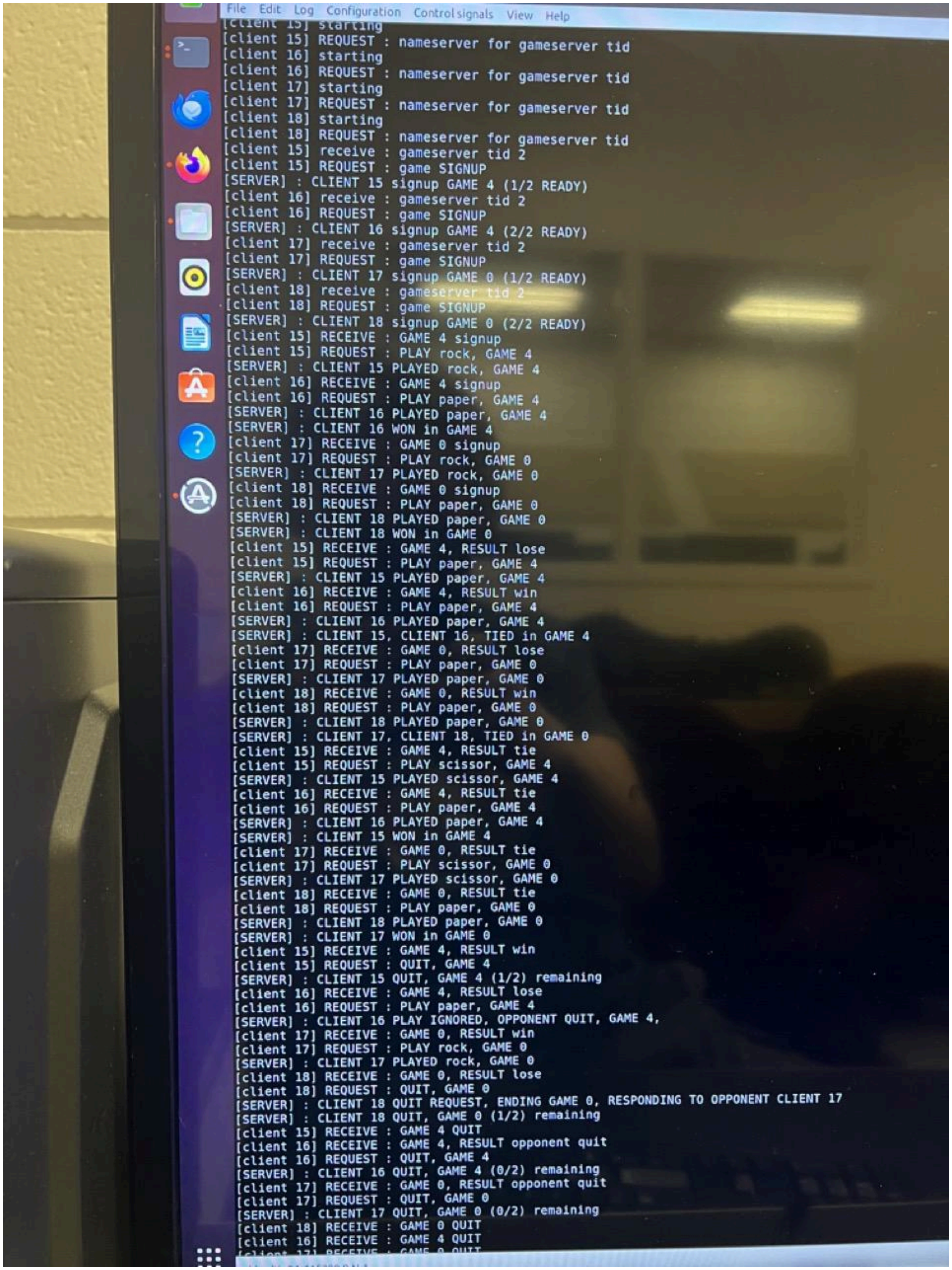
After the first pair are done, clients 12 and 13 begin their execution. We expect, SCISSOR, ROCK in round 1 for a lose, win. In round 2, we expect SCISSOR, PAPER for a win, lose, and in the last round we expect a SCISSOR, SCISSOR for a tie respectively for clients 12 and 13. Afterward, 12 quits and 13 attempts to play ROCK. 13's play is ignored and replied to with the partner quitting. 13 should then quit as well.

Test 3:

```
Create(3, &rps_quit); // task 15
Create(3, &paper_move_quit); // task 16
Create(3, &rps_move_quit); // task 17
Create(3, &paper_quit); // task 18
```

In the final test, we have 4 tasks at the same priority levels. Task 15 will play ROCK, PAPER, SCISSOR at the highest priority level, task 16 will only play PAPER and make an extra PAPER move before quitting. Task 17 will play ROCK, PAPER, SCISSOR, ROCK, and task 18 will only play PAPER. However, since we only have a max of 5 active games, we should expect the final game to wrap around and have game id of 0.

The order of tasks should match Test 1.



```
[client 15] starting
[client 15] REQUEST : nameserver for gameserver tid
[client 16] starting
[client 16] REQUEST : nameserver for gameserver tid
[client 17] starting
[client 17] REQUEST : nameserver for gameserver tid
[client 18] starting
[client 18] REQUEST : nameserver for gameserver tid
[client 15] receive : gameserver tid 2
[client 15] REQUEST : game SIGNUP
[SERVER] : CLIENT 15 signup GAME 4 (1/2 READY)
[client 16] receive : gameserver tid 2
[client 16] REQUEST : game SIGNUP
[SERVER] : CLIENT 16 signup GAME 4 (2/2 READY)
[client 17] receive : gameserver tid 2
[client 17] REQUEST : game SIGNUP
[SERVER] : CLIENT 17 signup GAME 0 (1/2 READY)
[client 18] receive : gameserver tid 2
[client 18] REQUEST : game SIGNUP
[SERVER] : CLIENT 18 signup GAME 0 (2/2 READY)
[client 15] RECEIVE : GAME 4 signup
[client 15] REQUEST : PLAY rock, GAME 4
[SERVER] : CLIENT 15 PLAYED rock, GAME 4
[client 16] RECEIVE : GAME 4 signup
[client 16] REQUEST : PLAY paper, GAME 4
[SERVER] : CLIENT 16 PLAYED paper, GAME 4
[SERVER] : CLIENT 16 WON in GAME 4
[client 17] RECEIVE : GAME 0 signup
[client 17] REQUEST : PLAY rock, GAME 0
[SERVER] : CLIENT 17 PLAYED rock, GAME 0
[client 18] RECEIVE : GAME 0 signup
[client 18] REQUEST : PLAY paper, GAME 0
[SERVER] : CLIENT 18 PLAYED paper, GAME 0
[SERVER] : CLIENT 18 WON in GAME 0
[client 15] RECEIVE : GAME 4, RESULT lose
[client 15] REQUEST : PLAY paper, GAME 4
[SERVER] : CLIENT 15 PLAYED paper, GAME 4
[client 16] RECEIVE : GAME 4, RESULT win
[client 16] REQUEST : PLAY paper, GAME 4
[SERVER] : CLIENT 16 PLAYED paper, GAME 4
[SERVER] : CLIENT 15, CLIENT 16, TIED in GAME 4
[client 17] RECEIVE : GAME 0, RESULT lose
[client 17] REQUEST : PLAY paper, GAME 0
[SERVER] : CLIENT 17 PLAYED paper, GAME 0
[client 18] RECEIVE : GAME 0, RESULT win
[client 18] REQUEST : PLAY paper, GAME 0
[SERVER] : CLIENT 18 PLAYED paper, GAME 0
[SERVER] : CLIENT 17, CLIENT 18, TIED in GAME 0
[client 15] RECEIVE : GAME 4, RESULT tie
[client 15] REQUEST : PLAY scissor, GAME 4
[SERVER] : CLIENT 15 PLAYED scissor, GAME 4
[client 16] RECEIVE : GAME 4, RESULT tie
[client 16] REQUEST : PLAY paper, GAME 4
[SERVER] : CLIENT 16 PLAYED paper, GAME 4
[SERVER] : CLIENT 15 WON in GAME 4
[client 17] RECEIVE : GAME 0, RESULT tie
[client 17] REQUEST : PLAY scissor, GAME 0
[SERVER] : CLIENT 17 PLAYED scissor, GAME 0
[client 18] RECEIVE : GAME 0, RESULT tie
[client 18] REQUEST : PLAY paper, GAME 0
[SERVER] : CLIENT 18 PLAYED paper, GAME 0
[SERVER] : CLIENT 17 WON in GAME 0
[client 15] RECEIVE : GAME 4, RESULT win
[client 15] REQUEST : QUIT, GAME 4
[SERVER] : CLIENT 15 QUIT, GAME 4 (1/2) remaining
[client 16] RECEIVE : GAME 4, RESULT lose
[client 16] REQUEST : PLAY paper, GAME 4
[SERVER] : CLIENT 16 PLAY IGNORED, OPPONENT QUIT, GAME 4,
[client 17] RECEIVE : GAME 0, RESULT win
[client 17] REQUEST : PLAY rock, GAME 0
[SERVER] : CLIENT 17 PLAYED rock, GAME 0
[client 18] RECEIVE : GAME 0, RESULT lose
[client 18] REQUEST : QUIT, GAME 0
[SERVER] : CLIENT 18 QUIT REQUEST, ENDING GAME 0, RESPONDING TO OPPONENT CLIENT 17
[SERVER] : CLIENT 18 QUIT, GAME 0 (1/2) remaining
[client 15] RECEIVE : GAME 4 QUIT
[client 16] RECEIVE : GAME 4, RESULT opponent quit
[client 16] REQUEST : QUIT, GAME 4
[SERVER] : CLIENT 16 QUIT, GAME 4 (0/2) remaining
[client 17] RECEIVE : GAME 0, RESULT opponent quit
[client 17] REQUEST : QUIT, GAME 0
[SERVER] : CLIENT 17 QUIT, GAME 0 (0/2) remaining
[client 18] RECEIVE : GAME 0 QUIT
[client 16] RECEIVE : GAME 4 QUIT
[client 17] RECEIVE : GAME 0 QUIT
```

All tasks fetch the nameserver after each other due to the same priority level and both begint their games after registering. However, note that since we allow a max of 5 games, clients 17 and 18 are put into game 0 again. For clients 15 and 16 we should expect ROCK, PAPER for a lose win in round 1. Then clients 17 and 18 do the same. For round 2, clients 15 and 16 should play PAPER, PAPER resulting in a tie. This also occurs for 17 and 18 right after. Lastly, 15 and 16 should play SCISSOR, PAPER resulting in a win, lose respectively. Similarity for 17 and 18. At this point, 15 quits but 16 attempts to play again. 16’s move is ignored and replied to with the opponent quit result. This also happens to 17 and 18 in a flipped order.

Changes in Kernel Design:

Changes from our K1 submission. Aside from these, all other kernel design decisions can be found in our K1 description.

Syscall Argument Passing

As mentioned in our K1 submission, we planned on changing the process in which user tasks pass arguments to the kernel. Previously we were using scratch registers (x9 & x10), however, we hoped to use the intended parameter registers (x0-x7) instead.

In our K2 implementation, the syscall parameters are moved to the user task registers starting from x0 as planned. When the kernel attempts to access those values, it simply reads from the current task descriptor register array.

During development, we noticed that compiler optimization level -O2 copies and runs the assembly code generated for some functions directly into the call sites of those functions as opposed to branching as one would expect. Thus, when we attempt to move the parameter values into the argument registers using inline assembly, we end up losing some parameter values because they are stored out of order in the same registers x0-x7. To mitigate this, we first move all the values into scratch registers x9 onwards, then move them back into x0-x7 in order.

Create() Error Handling

We added the missing create() return values for when an invalid priority value is provided and when the kernel is out of task descriptors.

SRR Performance Measurements

SRR performance test code can be found in the same git repo under branch k2-measurements or under this SHA: 71427f8e9be10cd233f48e14d2da5ad51e600ad3

The Send-Receive-Reply implementation is identical to the one in branch k2 but we removed all uart_dprintf() statements to get a more accurate measurement of performance. We observed a speed increase of approximately 10 microseconds from these removals even though the print function is implemented to not actually print unless the debug flag is on.

Methodology

To test the runtime of a complete SRR cycle, I create two tasks with the same priority. Sender_task is responsible for sending and for measuring time since the SRR operation ends in a return in the Send() function while receiver_task is responsible for receiving and immediately replying.
In order to account for the variability of the performance, we run the SRR functions a total of n=1000 times per configuration. We decided that this is a high enough number of repetition since we observed no difference between results using an n value of 1000, 2000, and 5000.
To reduce the impact of the counter function overhead, we get the difference before the clock count at the beginning and the clock count after all 1000 iterations. Furthermore, we also measure and log the clock runtime itself prior to each experiment but it has never exceeded 1 microsecond and is thus insignificant to our experiment result.

Observations

From the 48 configurations we have experimented on, we have observed these key properties:

- When observing results without optimization, we notice that the runtime has a linear relationship with message size. Specifically, SRR has approximately 280-300 microseconds of overhead and when the message size increases by 1 byte, the runtime increases by ~3.2 microseconds. Notably, this property does not hold true for the optimized results.
- Turning on optimization (level O3) has a significant impact on SRR performance. Furthermore, the level of effect optimization has is positively correlated to the message size, indicating that it efficiently reduces the runtime of linearly copying of one buffer into another:
    - 4 bytes: ~20% of noopt
    - 64 bytes: ~13% of noopt
    - 256 bytes: ~7% of noopt
- Modifying the execution order of sender-first vs. receiver-first has no observable impact on runtime.
- Turning on instruction cache when optimization is turned on seems to reduce runtime by ~9%. Other than that, changing the cache setup has no observable impact on SRR runtime.

Appendix: K1 documentation

CS 452 - K1

Team Members

Matthew Geng
m3geng
20820439

Erica Wang
e52wang
20825331

Repository: https://git.uwaterloo.ca/mg-cs452/cs452

Commit SHA: 745f77084b8b7e9041cb3a1658a64725bb0f4ee1

Kernel Design:

Our kernel design is based on a microkernel where code is non-interruptable and synchronous. System calls are kept short and fast by doing as little as possible on the kernel side.

The kernel is comprised of the following core components: a scheduler, a context switcher, an exception handler, and an exception code handler.

Before diving deep, we'll look at our kernel's high-level flow.

Kernel Main

After kernel initialization, the **scheduler** finds the next task to run based on task priority. The kernel **context switches** to that task, saving all the required information about the current kernel state and loading the task state. Afterward, either once the user task finishes or makes another system call, we hit our exception handler. The **exception handler** does the opposite of the initial context switch by saving the task state and loading the previous kernel state. The kernel resumes where it last left, and we appropriately respond to the exception code with our **exception code handler**. The code handler takes the exception code and executes various functionalities such as creating new tasks, yielding, finding information, etc based on the exception code. This process repeats from the top until there are no tasks to run.

Scheduler

The scheduler is a generic priority queue that takes in a fixed-sized array and a custom comparator. In our case, the fixed-sized array is an array of task frame pointers, and a function to compare task priorities.

The priority queue is implemented as a min heap to provide O(log(n)) operations for pushing and popping the minimum element. These time complexities are guaranteed from a binary heap implementation with siftup and siftdown operations.

Since task priorities go from high to low starting at 0, a min heap suited our use case. For tasks that have the same priority, the timestamp of when a task is added to the queue is used for tiebreaks, where the oldest have a higher priority. This results in a FIFO queue for tasks with the same priority.

Context Switcher

The context switcher is an assembly function that stores registers for the kernel and loads registers of the scheduled task.

It begins by grabbing the address of the kernel task frame. Registers x0-x30 (note x30 is the link register) and the stack pointer are stored using offsets from the address of the kernel frame with help from C struct memory alignment. We don’t need to store the pc because after switching back into the kernel, we want the code to jump directly to the lr address which is immediately after the context switching function. We also don’t need to store any special registers such as elr_1, spsr_1, etc in our kernel because these values are specific to an individual user task.

Afterward, the address of the current task frame is loaded. Registers x0-x30, the stack pointer, the pc, and spsr have been set from task initialization or a previous system call, and are loaded into their respective registers.

Specifically for loading our task, we store wherever a task is supposed to resume into elr_el1 in such that when we do an exception return, the exception return will jump to the correct spot. To illustrate, for the first context switch, we store the task’s function address in elr_el1.

For other special registers:

We store the user task’s SP into sp_el0 because since we are in the kernel, using “sp” will refer to sp_el1.

Lastly, we also set spsr_el1 to the correct bits such that SError and IRQ bits are masked and the sp_el0 bit is set.

Exception Handler

The exception handler is composed of 2 parts: the exception vector table and a c function. As part of kernel initialization, the VBAR (vector base address register) is populated with the start of our vector table label so that for every exception, we have a defined exception handler.

As a result, from a system call using SVC, we've defined a c function for that specific exception to jump to.

In that c function, we enter our kernel again and do the opposite of what our initial context switch does.

We store all registers x0-x30, sp, elr_el1, spsr_el1 into our current task frame, and load our previously saved kernel frame x0-x30 and sp registers.

Exception Code Handler

The exception code handler is a group of "if" statements that check for various exception codes immediately after the exception handler and return to the kernel main. The specific action depends on the type of exception. As an example, for create, we allocate a new task frame, initialize it, and add it and the original task to the scheduler queue. Furthermore, for exception types that need to pass in certain arguments, registers x9 and x10 are used and for returning values, x0 is used. Note the reason why we are currently not using x0-x7 for parameters is because intermediary function calls that overwrite x0-x7. In order to compensate for this, we would have to revamp how we store passed in arguments, which is something we will look into for the next assignment. However, for our current use case other scratch registers x9-x18 satisfy our current use case.

Memory Layout

.stack (kernel stack)
.text.boot
.text.evt (exception vector table)
.text (code)
.rodata
.data
.bss
start of user task stack base

Memory Allocation/Task Frames

There is no dynamic memory allocation in this kernel at this point. All memory allocations are done at initialization. The most “dynamic” structure we have is our array/list of task frame descriptions.

Each of our task frames are given a base stack address to represent their stack. In order to have contiguous memory and decouple from the need of a memory allocator, each of these task frames are intrinsically linked to the next task frame through a field which is a pointer to the next task frame.

Instead of utilizing the stack directly with push and pop operations in order to save user and kernel registers, we have opted into storing our registers with a global pointer to a memory address, representing our task frame, on the kernel stack. We have done this in order to more effectively debug our code by being able to examine specific registers globally.

Kernel Initialization

Our kernel initialization comprises of the following steps

- Initialization the base of our exception vector table
- GPIO + uart initialization
- Kernel task frame initilziation and setting global pointer to kernel task frame (usage in context switching)
- User task initializations (setting TID’s and stack base addresses)
- Scheduler (heap) initialization
- First task initialization

System Parameters

- Maximum of 20 user task frames (excluding task frame for
    - After taking a look at our future use case of a train system, we estimated each train would need ~5 tasks (controller, input, output, sensor, etc) which at most we’d probably have ~3 trains at a time. This is a total of 15 tasks, which leaves 5 other tasks for our kernel including our clock and other items. This was a lower bound estimate.
- Kernel stack base address is 0x7FFF0
    - To minimize potential kernel stackoverflow issues, we’ve ensure that our stack can “grow” downwards until 0x0, such that when we do reach stackoverflow, we should be able to handle that promptly or automatically through a hardware error.
- Each user stack size is 1024 bytes
    - This was chosen as a rough estimate by checking the runtime stack size of our kernel which was < 1024 bytes. As a result, we thought this would be a decent start for our user task size

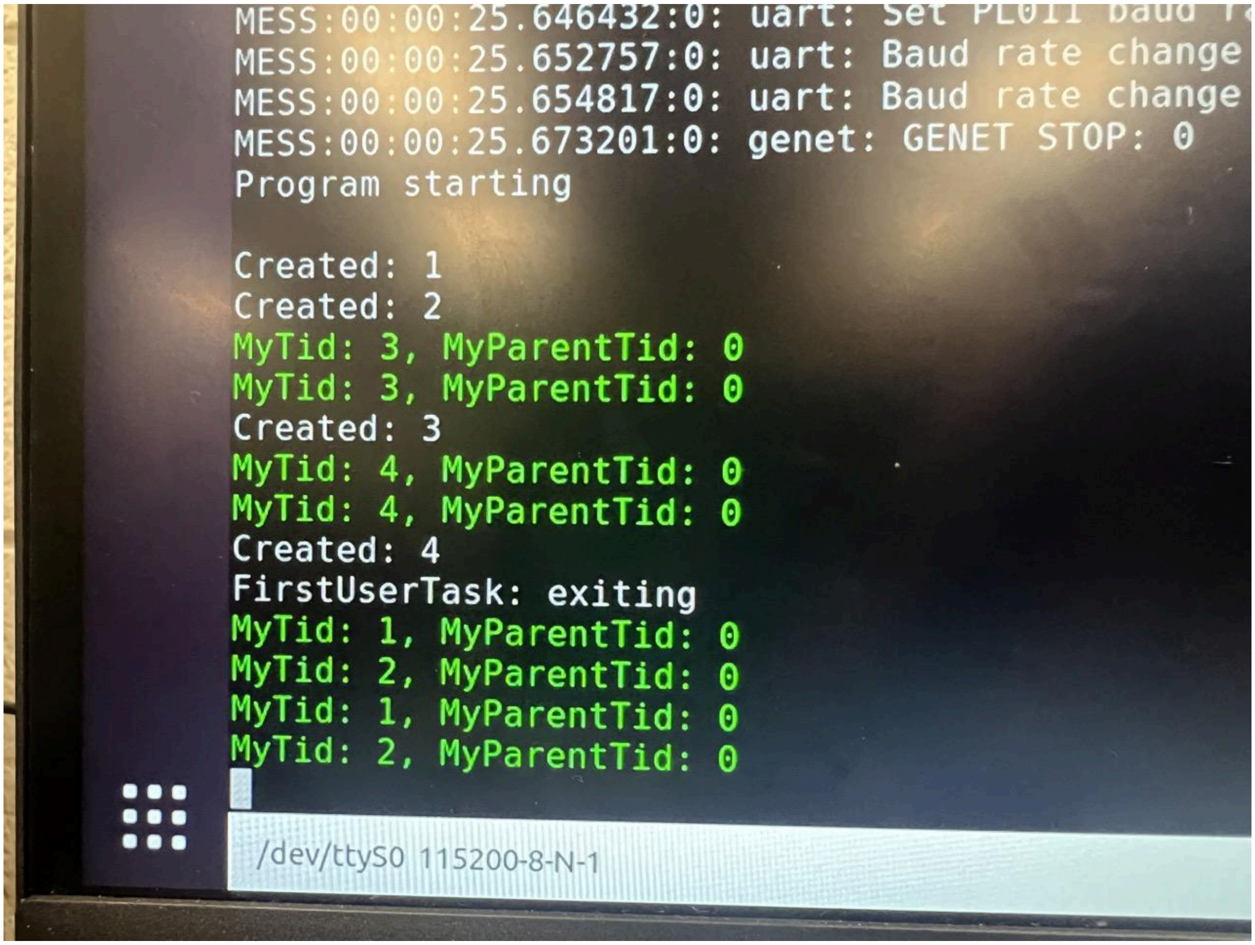
- First user stack base address is 0x8D9A0 (grows upwards for every base stack address, so next stack base address would be 0x8DDA0 that grows downwards)
    - With our current stack size, the lowest address a user stack “should” touch is 0x8D5A0 which is far higher than our highest piece of memory defined in our linker, bss, which ends at 0x82f88. As a result, we this stack base should be adequate for our usage especially for A1.

Bugs and Limitations:

- Creating more than 20 task frames (without reclaiming a task frame)
- No check for stack overflow for kernel or user tasks

Output Explanation:

Output:



```
MESS:00:00:25.646432:0: uart: Set PL011 baud r
MESS:00:00:25.652757:0: uart: Baud rate change
MESS:00:00:25.654817:0: uart: Baud rate change
MESS:00:00:25.673201:0: genet: GENET STOP: 0
Program starting

Created: 1
Created: 2
MyTid: 3, MyParentTid: 0
MyTid: 3, MyParentTid: 0
Created: 3
MyTid: 4, MyParentTid: 0
MyTid: 4, MyParentTid: 0
Created: 4
FirstUserTask: exiting
MyTid: 1, MyParentTid: 0
MyTid: 2, MyParentTid: 0
MyTid: 1, MyParentTid: 0
MyTid: 2, MyParentTid: 0
```

Lets walkthrough the output. At the beginning of our kernel, we create a root task that creates these 4 sub tasks.

Initially, our root task is at priority 2. The root task first tries to create its first task at priority 3. Since priority 3 is less than our root task's priority, we continue and get our first line of output

Created: 1

The root task creates a new task at priority 3 as well. Since that is also less than our root's priority, we again continue and get out second line of output with a new TID.

Created: 2

The root task now creates a new task at priority 1. Since that is greater than our current root task's priority, we are pre-empted and the new task runs to get our new line of input.

MyTid:3, MyParentTid: 0

At this point, the task yields control back to the kernel. However, since it has the highest priority, it is rescheduled and re-prints.

MyTid:3, MyParentTid: 0

Once the task is finished, it exits, which allows us to context switch back into our root task to get our next output line.

Created: 3

Simialr to task 3, the root task now creates another task at priority 1. Since that is also greater than our current root task's priority, we are pre-empted and the new task runs to get our next line of input.

MyTid:4, MyParentTid: 0

Afterwards, the task yields and is rescheduled due to its high priority of 1 causing us to come back to this task and print the next line.

MyTid:4, MyParentTid: 0

Afterwards, task 4 exits and the root task is rescheduled and outputs.

Created: 4

Now, since the root task is finished, the root task exits and prints the following.

```
FirstUserTask: exiting
```

However, we are not done yet as we have two tasks, 1 and 2, still in our queue due to their low priority. Since these two tasks are at the same priority, the timestamp of when they are added to the queue is used to tiebreak these tasks. Because we always scheduled the oldest first to provide FIFO behaviour, task 1 should run and does.

```
MyTid:1, MyParentTid: 0
```

Task 1 yields and is added back to our queue. However, unlike tasks 3 and 4, task 1 has the same priority as task 2 and using our timestamp tiebreak, task 2 is scheduled instead by being the oldest task.

```
MyTid:2, MyParentTid: 0
```

Task 2 now yields, and using similar logic as above, task 1 is now scheduled.

```
MyTid:1, MyParentTid: 0
```

As this task now exits, the last task to be scheduled is task 2.

```
MyTid:2, MyParentTid: 0
```