## Contents

# The `choochoos` Kernel

Written by Daniel Prilik (dprilik - 20604934) and James Hageman (jdhagema - 20604974).

## Project Structure

### Navigating the Repo

`choochoos` follows a fairly standard C/C++ project structure, whereby all header files are placed under the `include` directory, with corresponding implementation files placed under the `src` directory.

At the moment, both the `choochoos` kernel and userspace live under the same `include` and `src` directory, though we plan to separate kernel-specific files and userspace-specific files at some point.

### Building `choochoos`

For details on building the project, see the README.md at the root of this repository.

## Architectural Overview

Our entry point is not actually `main`, but `_start` defined in src/boilerplate/crt1.c. `_start` does couple key things for the initialization of the kernel: - The link register is stored in the `redboot_return_addr` variable, so we can exit from any point in the kernel's execution by jumping directly to `redboot_return_addr` - The BSS is zeroed - All global variable constructors are run

Then we jump into `main`, which configures the `COM2` UART and jumps to `kmain`.

`kmain` executes our scheduling loop, using a singleton instance of a class called `Kernel`:

```
static kernel::Kernel kern;

int kmain() {
    kprintf("Hello from the choochoos kernel!");

    kern.initialize(FirstUserTask);

    while (true) {
        int next_task = kern.schedule();
        if (next_task < 0) break;
        kern.activate(next_task);
    }

    kprintf("Goodbye from choochoos kernel!");

    return 0;
}
```

The scheduling loop differs from the one described in lecture only in that `Kernel::activate` does not return a syscall request to handle. Instead, activate only returns after the syscall has been handled. Nonetheless, the kernel can be broken down into three main parts: scheduling, context switching, and syscall handling.

### Scheduling

`schedule()` has a remarkably simple implementation:

```
int schedule() {
    int tid;
    if (ready_queue.pop(tid) == PriorityQueueErr::EMPTY) return -1;
    return tid;
}
```

We have a `ready_queue`, which is a FIFO priority queue of Tids. Tasks that are ready to be executed are on the queue, and `schedule()` simply grabs the next one.

Our priority queue is implemented as a template class in include/priority_queue.h. A `PriorityQueue<T, N>` is a fixed size object containing up to `N` elements of type `T`, implemented as a modified binary max-heap. Usually, when implementing a priority queue as a binary heap, the elements within the heap are compared only by their priority, to ensure that elements of higher priority are popped first. However, only using an element's priority does not guarantee FIFO ordering *within* a priority.

Instead, we extend each element in the priority queue to have a `ticket` counter. As elements are pushed onto the queue, they are assigned a monotonically increasing `ticket`. When elements with the same priority are compared, the element with the lower `ticket` is given priority over the element with the higher `ticket`. This has the effect of always prioritizing elements that were added to the queue first, and gives us the desired per-priority FIFO.

Using this ticketed binary heap has both drawbacks and benefits over a fixed-priority implementation (be that a vector of queues, or an intrusive linked list per priority). The benefit is that we can permit `p` priorities in `O(1)` space instead of `O(p)` space. This allows us to define priorities as the entire range of `int`, and will allow us to potentially reliable arithmetic on priorities in the future (however, whether or not we'll need such a thing is yet to be discovered).

The drawback is that we lose FIFO if the ticket counter overflows. Right now, the ticket counter is a `size_t` (a 32-bit unsigned integer), so we would have to push `2^32 Tid`s onto `ready_queue` in order to see an overflow. This definitely won't happen in our `k1` demo, but it doesn't seem impossible in a longer running program that is rapidly switching tasks. We're experimenting with using the non-native `uint64_t`, and we will profile such a change as we wrap up `k2`.

### Context Switching - Task Activation

Once the scheduler has returned a Tid, the `kern.activate()` method is called with the Tid. This method updates the kernel's `current_task` with the provided Tid, fetch the task's saved stack pointer from its Task Descriptor, and hand the stack pointer to the `_activate_task` Assembly routine.

This assembly routine performs the following steps in sequence: - Saves the kernel's register context onto the kernel's stack (i.e: r4-r12,lr) - r0-r3 are not saved, as per the ARM C-ABI calling convention - *Note:* this does mean that in the future, when implementing interrupt-handling into our kernel, this routine will either have to be modified and/or supplemented by a second routine which saves / restores the entirety of a user's registers. - Switches to System mode, banking in the last user tasks's SP and LR - Moves the new task's SP from r0 to SP - Pops the saved user context from the SP into the CPU - Pops the SWI return value into r0, the SWI return address into r1, and the user SPSR into r2 - Registers r4-12 and LR are restored via a single ldmfd instruction (r0 through r3 are *not* restored, as the ARM C ABI doesn't require preserving those registers between method calls) - Moves the saved SPSR from r2 into the SPSR register - Calls the `movs` instruction to jump execution back to the saved swi return address (stored in r1), updating the CPSR register with the just-updated SPSR value

The task will then continue its execution until it performs a syscall, at which point the "second-half" of context-switching is triggered.

### Context Switching - SWI Handling

Invoking a syscall switches the CPU to Supervisor mode, and jumps execution to the SWI handler. Our SWI handler is written entirely in Assembly, and was given the extremely original name `_swi_handler`.

This routine is a bit more involved than the `_activate_task` routine, mainly due to some tricky register / system-mode juggling, but in a nutshell, the `_swi_handler` routine preforms the following steps in sequence: - Save the user's r0 through r12 on the user's stack (requires temporarily switching back to System mode from supervisor mode) - Keep a copy of the user's SP in r4, which is used to stack additional data onto the user stack - Save the SWI return address & user SPSR from the *Supervisor* mode LR and SPSR onto the *User* mode SP (using the SP saved in r4) - Use the SWI return address offset by -4 to retrieve the SWI instruction that was executed, and perform a bitmask to retrieve it's immediate value (i.e: the syscall number) - Call the `handle_syscall` routine (implemented in C) with the syscall number and the user's SP - See the "Handling Syscalls" section below for more details - Push the `handle_syscall` return value onto the user stack - Restore the saved kernel context from the kernel stack, returning execution back to the instruction immediately after the most recently executed `_activate_task` call (i.e: back to the `kern.activate()` method)

The `kern.activate()` method proceeds to check to see what state the last-executed task is in, and queues it up to be re-scheduled if it's READY. This completes a single context switch, and execution flow is returned back to the scheduler.

### Handling Syscalls

`handle_syscall` switches on the syscall number, calling the corresponing method on the `Kernel` instance. The user's stack pointer is cast to a `SwiUserStack*`, which allows the user's registers (and the rest of their stack) to be read by our

C++ code. This allows us to pass parameters to our C++ syscall handlers. Let's enumerate the handlers:

- `MyTid()` returns the kernel's `current_tid`, which is updated to the most recent Tid whenever as task is activated.
- `Create(priority, function)` determines the lowest free Tid, and constructs a task descriptor in `tasks[tid]`. The task descriptor is assigned a stack pointer, and the user stack is initialized by casting the stack pointer to a `FreshUserStack*`, and writing to that struct's fields. Notably, we write `stack->lr = (void*)User::Exit`, which sets the return address of `function` to be the `Exit()` syscall, allowing the user to omit the final `Exit()`. The task's `parent_tid` is the current value of `MyTid()`.
- `MyParentTid()` looks up the `TaskDescriptor` by `current_tid`, which holds the parent Tid. (The parent Tid is set to the value of `current_tid` when
- `Yield()` does nothing. Since it leaves the calling task in the `READY` state, the task will be re-added to the `ready_queue` in `activate()`.
- `Exit()` clears the task descriptor at `tasks[MyTid()]`. This prevents the task from being rescheduled in `activate()`, and allows the Tid to be recycled in future calls to `Create()`.

### System Limitations

Our linker script, ts7200_redboot.ld, defines our allocation of memory. Most notably, we define the range of memory space allocated to user stacks from `__USER_STACKS_START__` to `__USER_STACKS_END__`. Each user task is given 256KiB of stack space, so the maximum number of concurrent tasks in the system is `(__USER_STACKS_END__ - __USER_STACKS_START__) / (256 * 1024)`. However, given the variable size of our BSS and data sections (as we change code), we can't compute the optimal number of concurrent tasks until link time. So instead, we hardcode a `MAX_SCHEDULED_TASKS`, and assert at runtime that no task could possibly have a stack outside of our memory space. Currently, this value is set to 48 tasks. The kernel is given 512KiB of stack space.

## New in K2

In K2 we implemented the Send-Receive-Reply mechanism, the Name Server, and the Rock Paper Scissor Server & Client.

### Send-Receive-Reply

SRR is implemented by adding a `TaskState state;` field to each `TaskDescriptor`. `TaskState` is a tagged union with the following structure:

```c
struct TaskState {
    enum uint8_t { UNUSED, READY, SEND_WAIT, RECV_WAIT, REPLY_WAIT } tag;
    union {
        struct {
        } unused;
        struct {
        } ready;
        struct {
            const char* msg;
            size_t msglen;
            char* reply;
            size_t rplen;
            int next;
        } send_wait;
        struct {
            int* tid;
            char* recv_buf;
            size_t len;
        } recv_wait;
        struct {
            char* reply;
            size_t rplen;
        } reply_wait;
    };
};
```

So each `TaskDescriptor` can be in any of the states described by `tag`, at which point the fields in the corresponding struct under the `union` will be populated:

- `UNUSED`: the `TaskDescriptor` does not represent a running task.
- `READY`: the task is on the `ready_queue`, waiting to be scheduled.
- `SEND_WAIT`: the task is waiting to `Send()` to another task that hasn't called `Receive()` yet.
- `RECV_WAIT`: the task has called `Receive()`, but no task has sent a message to it yet.
- `REPLY_WAIT`: the task called `Send()` and the receiver got the message via `Receive()`, but no other task has called `Reply()` back at the task.

To implement these state transitions, each task has a `send_queue` of tasks that are waiting to send to it (and must therefore be in the `SEND_WAIT` state). This send queue is built as an intrusive linked list, where the `next` "pointer" is actually just another Tid, or `-1` to represent the end of the list. When a task wants to `Send()` to another task that is not in `RECV_WAIT`, it will be put on the receiver's send queue (denoted by the fields `send_queue_head` and `send_queue_tail`, also Tids), and the sending task will be put in `SEND_WAIT`. Note that the `send_wait` branch of the union contains everything that the sender passed to `Send()`, plus the `next` pointer, if any other task get added to that same send queue.

If a task is in SEND_WAIT, it can only be blocked sending to one receiving task, so we only need at most one next pointer per task. The result is very memory efficient: by storing one word on each task descriptor, we can support send queues up to length MAX_SCHEDULED_TASKS - 1.

Let's step through the two possibilities for an SRR transaction: sender-first and receiver-first.

### Sender-first

If a sender arrives first, that means the receiver is not in RECV_WAIT, and therefore does not yet have a recv_buf to be filled. If this is the case, we add the sender to the end of the receiver's send queue, and put the sender in SEND_WAIT.

When the receiver finally calls Receive(), it will see that is has a non-empty send queue. So it will pop the first TaskDescriptor off the front of its send queue, copy the msg into recv_buf, and transition the sender into REPLY_WAIT, using the same char* reply that the sender saved when they went into SEND_WAIT.

### Receiver-first

If the receiver arrives first, it checks its send queue. If the send queue is non-empty, then we follow the same procedure as sender-first, and the new sender is placed on the send of the receiver's send queue. If the send queue is empty, the receiver goes into RECV_WAIT, holding onto the recv_buf to be copied into once a sender arrives.

When the sender arrives, it notices that the receiver is in RECV_WAIT, so it writes the msg directly into recv_buf, wakes up the receiver (transitioning it to READY and pushing it onto the ready_queue), and goes into REPLY_WAIT.

### Reply

Reply(tid, reply, rplen) only delivers the reply if the task identified by tid is in REPLY_WAIT. The kernel can then immediately copy the reply into the char* reply stored in the reply_wait branch of the task's state union. This way, reply never blocks - it only returns an error code if the task is not in REPLY_WAIT.

### Error cases

The SRR syscalls return two possible error codes: -1 and -2. -1 is returned whenever a tid does not represent a task - either it is out of range, or it points to a TaskDescriptor in the UNUSED state. -2 is returned in two cases where a SRR transaction cannot be completed:

- If a task tries to Reply()to a task that is not in REPLY_WAIT - this would mean that the task never called Send() and thus is not expecting a reply.
- If a task is in SEND_WAIT in a receiver's send queue, and the receiver exits.

When a receiver exits, if it has a non-empty send queue, those tasks will never get a reply because they won't make the transition into `REPLY_WAIT`, and thus would never wake up. Instead, we iterate through the send queue, waking ever `SEND_WAIT` task up, and writing the syscall return value of `-2`.

## Name Server

The Name Server task provides tasks with a simple to use mechanism to discover one another's Tids.

### Public Interface

The name server's public interface can be found under `include/user/tasks/nameserver.h`, a header file which defines the methods and constants other tasks can use to set up and communicate with the nameserver.

We've decided to use an incredibly simple closure mechanism for determining the Tid of the name server, namely, we enforce that the name server is the second user task to be spawned (after the FirstUserTask), resulting in it's Tid always being 1.

To make working with the name server easier, instead of having tasks communicate with it directly via SRR system calls, a pair of utility methods are provided which streamline the registration and query flows:

- `int WhoIs(const char*)`: returns the Tid of a task with a given name, `-1` if the name server task is not initialized, or `-2` if there was no registered task with the given name.
- `int RegisterAs()`: returns `0` on success, `-1` if the name server task is not initialized, or `-2` if there was an error during name registration (e.g: nameserver is at capacity.)
    - *Note:* as described later, the nameserver currently panics if it runs out of space, so user tasks will never actually get back `-2`.

### Implementation

The name server's implementation can be found under `src/user/tasks/nameserver.cc`.

**Preface: A Note on Error Handling** At the moment, the name server's implementation will simply panic if various edge-conditions are encountered, namely, when the server runs out of space in any of it's buffers. While it would have been possible to return an error code to the user, we decided not to, as it's unlikely that the user process would be able to gracefully recover from such an error.

Instead, we will be tweaking the size of our name server's buffers over the course of development, striking a balance between memory usage and availability. Alternatively, we may explore implementing userspace heaps at some point, in which case, we may be able to grow the buffer on-demand when these edge conditions are hit.

**Associating Strings with Tids** The core of any name server is some sort of associative data structure to associate strings to Tids.

While there are many different data structures that fit this requirement, ranging from Tries, Hash Maps, BTree Maps, etc…, we've decided to use a simple, albeit potentially inefficient data structure instead: a plain old fixed-length array of ("String", Tid) pairs. This simple data structure provides O(1) registration, and O(n) lookup, which shouldn't be too bad, as we assume that most user applications won't require too many named tasks (as reflected in the specification's omission of any sort of "de-registration" functionality).

**Aside: Efficiently Storing Names** Note that we've put the term "String" in quotes. This is because instead of using a fixed-size char buffer for each pair, we instead allocate strings via a separate data structure: the `StringArena`.

`StringArena` is a simple class which contains a fixed-size `char` buffer, and a index to the tail of the buffer. It exposes two methods:

- `size_t StringArena::add(const char* s, const size_t n)`: Copy a string of length `n` into the arena, returning a handle to the string's location within the arena (represented by a `size_t`). This handle can then be passed to the second method on the arena…
- `const char* StringArena::get(const size_t handle)`: Return a pointer to a string associated with the given handle, or `nullptr` if the handle is invalid.

Whenever `add` is called, the string is copied into the arena's fixed-size internal buffer, incrementing the tail-index to point past the end of the string. `get` simply returns a pointer into the char buffer associated with the returned handle (which at the moment, is simply an index into the char array).

The `StringArena` approach allows us to avoid having to put an explicit limit on the size of name strings, as strings of varying lengths can be "packed" together in the single char buffer.

**Incoming and Outgoing Messages** The `WhoIs` and `RegisterAs` functions abstract over the name server's message interface, which is comprised of two tagged unions: `Request` and `Response`.

```
enum class MessageKind : size_t { WhoIs, RegisterAs, Shutdown };

struct Request {
    MessageKind kind;
    union {
        struct {
            char name[NAMESERVER_MAX_NAME_LEN];
            size_t len;
        } who_is;
        struct {
            char name[NAMESERVER_MAX_NAME_LEN];
            size_t len;
            int tid;
        } register_as;
    };
};

struct Response {
    MessageKind kind;
    union {
        struct {
        } shutdown;
        struct {
            bool success;
            int tid;
        } who_is;
        struct {
            bool success;
        } register_as;
    };
};
```

There are 3 types of request, each with a corresponding response - `Shutdown` - terminate the name server task - `WhoIs` - Return the Tid associated with a given name - `RegisterAs` - Register a Tid with a given name

The latter two messages return a non-empty response, indicating if the operation was successful, and for `WhoIs`, a Tid (if one was found).

The server uses a standard message handling loop, whereby the body of the server task is an infinite loop, which continuously waits for incoming messages, switches on their type, and handles them accordingly.

**Future Improvements** We would like to split up the Request into two parts: a request "header", followed by an optional request "body." This would lift the artificially imposed `NAMESERVER_MAX_NAME_LEN` limit on names, as the request header could specify the length of the upcoming message, which the name server could then read into a variable-sized stack-allocated array (allocated via `alloca` / a VLA). This comes with a minor security risk, whereby a malicious and/or misbehaving task could specify a extremely large name, and overlow the nameserver's stack, but given the the operating system will only be running code we write ourselves, this shouldn't be too much of an issue.

# K2 Output - RPS

## RPS Server & Client

The RPS server and client are implemented in `src/assignments/k2/rps.cc`. Both tasks are configured by receiving a configuration message from their parent task after being created. When the executable is run, the `FirstUserTask` prompts the user for configuration, spawns the RPS server, and spawns the appropriate number of client tasks.

### Client

Client tasks are very straightforward. After receiving their configured number of games, the client looks up the RPS server via `WhoIs()` and sends it a "signup" request. Once the server ACKs the signup request, it will play `num_games` games. During each game, the client generates a random move (via `rand()`), sends the move to the server, and waits for a response. The response either tells the client their result of the game (win, loss, draw), or that the other player quit and there are no other players to play against.

Once a client has played `num_games` games, or there are no other players to play against, the client exits.

### Server

The server task is more intricate. It has an array of `Game` objects, each representing the state of a single game. A `Game` can either be full or empty, but it cannot be half-full. The server also has a queue of size 1, to keep track of players that have not yet been matched in a game.

When a player signs up and the queue is empty, it is added to the queue, and the server does not immediately reply. When a player signs up and the queue is full, that player and the enqueued player are matched. Matching involves finding the first empty game in the `games` array - if there are no empty games, the server replies to both players with an `OUT_OF_SPACE` message. Otherwise, it replies to both players with an `ACK`, acknowledging the signup requests.

As clients send `PLAY` messages to the server, the server finds the `Game` associated with that client, and updates that client's `choice` in the game. When both players in a game have submitted their `choice`, the server determines the winner and sends the result as `PLAY_RESP` replies.

When a client sends a `QUIT` message, the server checks the queue. If another player is waiting to join a game, that player will be matched with the player that the quitting client was playing against. In this swap, any move that the existing player had made will be preserved. If there is no queued player, the server sends the `OTHER_PLAYER_QUIT` message to the remaining player, who is then expected to stop sending `PLAY` messages.

## Transcript

```
Hello from the choochoos kernel!
random seed (>= 0): 2
pause after each game (y/n)? y
num players (0-32): 3
player 1 priority  (default 1): 1
player 1 num games (default 3): 3
player 2 priority  (default 1): 2
player 2 num games (default 3): 5
player 3 priority  (default 1): 3
player 3 num games (default 3): 3
[Client tid=3 id=?] waiting for player config
[Client tid=4 id=?] waiting for player config
[Client tid=5 id=?] waiting for player config
[Client tid=5 id=?] received player config (num_games=3, id=3)
[Client tid=5 id=3] querying nameserver for 'RPSServer'
[Client tid=4 id=?] received player config (num_games=5, id=2)
[Client tid=4 id=2] querying nameserver for 'RPSServer'
[Client tid=3 id=?] received player config (num_games=3, id=1)
[Client tid=3 id=1] querying nameserver for 'RPSServer'
[RPSServer] accepting signups...
[Client tid=5 id=3] received reply from nameserver: RPSServer=2
[Client tid=5 id=3] I want to play 3 games. Sending signup...
[Client tid=4 id=2] received reply from nameserver: RPSServer=2
[Client tid=4 id=2] I want to play 5 games. Sending signup...
[RPSServer] matching tids 4 and 5
[Client tid=4 id=2] received signup ack
[Client tid=4 id=2] I want to play 5 more games. Sending scissors...
[Client tid=3 id=1] received reply from nameserver: RPSServer=2
[Client tid=3 id=1] I want to play 3 games. Sending signup...
[Client tid=5 id=3] received signup ack
[Client tid=5 id=3] I want to play 3 more games. Sending paper...
[Client tid=4 id=2] I won!
[Client tid=4 id=2] I want to play 4 more games. Sending paper...
[Client tid=5 id=3] I lost :(
[Client tid=5 id=3] I want to play 2 more games. Sending rock...
~~~~~~~~~ press any key to continue ~~~~~~~~~
[Client tid=4 id=2] I won!
[Client tid=4 id=2] I want to play 3 more games. Sending rock...
[Client tid=5 id=3] I lost :(
[Client tid=5 id=3] I want to play 1 more game. Sending rock...
~~~~~~~~~ press any key to continue ~~~~~~~~~
[Client tid=4 id=2] it's a draw
[Client tid=4 id=2] I want to play 2 more games. Sending rock...
[Client tid=5 id=3] it's a draw
```

```
45  [Client tid=5 id=3] sending quit
46  ~~~~~~~~~ press any key to continue ~~~~~~~~~
47  [RPSServer] tid 5 quit, but tid 3 is waiting. Matching tids 4 and 3
48  [Client tid=3 id=1] received signup ack
49  [Client tid=3 id=1] I want to play 3 more games. Sending scissors...
50  [Client tid=5 id=3] exiting
51  [Client tid=4 id=2] I won!
52  [Client tid=4 id=2] I want to play 1 more game. Sending scissors...
53  [Client tid=3 id=1] I lost :(
54  [Client tid=3 id=1] I want to play 2 more games. Sending rock...
55  ~~~~~~~~~ press any key to continue ~~~~~~~~~
56  [Client tid=4 id=2] I lost :(
57  [Client tid=4 id=2] sending quit
58  [Client tid=3 id=1] I won!
59  [Client tid=3 id=1] I want to play 1 more game. Sending paper...
60  ~~~~~~~~~ press any key to continue ~~~~~~~~~
61  [RPSServer] tid 4 quit, but no players are waiting.
62  [Client tid=4 id=2] exiting
63  [Client tid=3 id=1] other player quit! I guess I'll go home :(
64  [Client tid=3 id=1] exiting
65  Goodbye from choochoos kernel!
```

## Explanation

```
 1  Hello from the choochoos kernel!
 2  random seed (>= 0): 2
 3  pause after each game (y/n)? y
 4  num players (0-32): 3
 5  player 1 priority  (default 1): 1
 6  player 1 num games (default 3): 3
 7  player 2 priority  (default 1): 2
 8  player 2 num games (default 3): 5
 9  player 3 priority  (default 1): 3
10  player 3 num games (default 3): 3
```

Game setup (The user's input is included in the transcript). The random number generator is configured with a seed of 2, and we pause for input after each game is finished. We are running 3 client tasks, with priotities (1,2,3) and wanting to play (3,5,3) games respectively.

```
11  [Client tid=3 id=?] waiting for player config
12  [Client tid=4 id=?] waiting for player config
13  [Client tid=5 id=?] waiting for player config
14  [Client tid=5 id=?] received player config (num_games=3, id=3)
15  [Client tid=5 id=3] querying nameserver for 'RPSServer'
16  [Client tid=4 id=?] received player config (num_games=5, id=2)
17  [Client tid=4 id=2] querying nameserver for 'RPSServer'
```

```
[Client tid=3 id=?] received player config (num_games=3, id=1)
[Client tid=3 id=1] querying nameserver for 'RPSServer'
```

The client tasks wait for their configuration (sent from the main task), and then query the NameServer for “RPSServer”. Note that while the clients ask for their configuration in order, the highest priority task is (Tid 5) is woken up first, and thus queries the NameServer first.

```
[RPSServer] accepting signups...
[Client tid=5 id=3] received reply from nameserver: RPSServer=2
[Client tid=5 id=3] I want to play 3 games. Sending signup...
[Client tid=4 id=2] received reply from nameserver: RPSServer=2
[Client tid=4 id=2] I want to play 5 games. Sending signup...
[RPSServer] matching tids 4 and 5
```

The RPSServer (priority 0) starts accepting signups. Clients 3 and 2 are the highest priority, so their signup requests are received first, and they are matched.

```
[Client tid=4 id=2] received signup ack
[Client tid=4 id=2] I want to play 5 more games. Sending scissors...
[Client tid=3 id=1] received reply from nameserver: RPSServer=2
[Client tid=3 id=1] I want to play 3 games. Sending signup...
[Client tid=5 id=3] received signup ack
[Client tid=5 id=3] I want to play 3 more games. Sending paper...
```

Client 2 receives the signup ack, and sends scissors. Client 1 (the one with the lowest priority), only just receives the response from the NameServer. It sends a signup request, but won’t hear back for a while. Client 3, which was matched with client 2, also receives the signup ack and sends paper.

```
[Client tid=4 id=2] I won!
[Client tid=4 id=2] I want to play 4 more games. Sending paper...
[Client tid=5 id=3] I lost :(
[Client tid=5 id=3] I want to play 2 more games. Sending rock...
~~~~~~~~~ press any key to continue ~~~~~~~~~
```

Our first game! Client 2 send scissors and client 3 sent paper, so client 2 wins. Both clients are informed of the result and send their next move. Since clients send their next move as soon as they see the results, client 2 sends their next move before client 3 has heard that it lost. The program pauses waiting for the user to press a key.

```
[Client tid=4 id=2] I won!
[Client tid=4 id=2] I want to play 3 more games. Sending rock...
[Client tid=5 id=3] I lost :(
[Client tid=5 id=3] I want to play 1 more game. Sending rock...
~~~~~~~~~ press any key to continue ~~~~~~~~~
```

Client 2 sent paper and client 3 sent rock, so client 2 wins again. For the third game, both clients send rock. For client 3, it is their last game.

```
[Client tid=4 id=2] it's a draw
[Client tid=4 id=2] I want to play 2 more games. Sending rock...
[Client tid=5 id=3] it's a draw
[Client tid=5 id=3] sending quit
~~~~~~~~~~ press any key to continue ~~~~~~~~~~
```

Since both clients sent rock in the last round, it's a draw. Client 2 wants to play another game, so it sends rock. But client 3 doesn't want to play any more games, so it sends quit.

```
[RPSServer] tid 5 quit, but tid 3 is waiting. Matching tids 4 and 3
[Client tid=3 id=1] received signup ack
[Client tid=3 id=1] I want to play 3 more games. Sending scissors...
[Client tid=5 id=3] exiting
[Client tid=4 id=2] I won!
[Client tid=4 id=2] I want to play 1 more game. Sending scissors...
[Client tid=3 id=1] I lost :(
[Client tid=3 id=1] I want to play 2 more games. Sending rock...
~~~~~~~~~~ press any key to continue ~~~~~~~~~~
```

The RPS receives client 3's quit message and removes it from the game. Since client 1 (Tid 3) is still waiting to join a game, and client 2 is still playing, the RPSServer matches client 1 and 2, and finally sends client 1 the signup ack. Client 1 sends scissors, and since client 2 sent rock (in the last segment), client 2 wins. Clients 1 and 2 send another move, and client 3, after quitting, exits.

```
[Client tid=4 id=2] I lost :(
[Client tid=4 id=2] sending quit
[Client tid=3 id=1] I won!
[Client tid=3 id=1] I want to play 1 more game. Sending paper...
~~~~~~~~~~ press any key to continue ~~~~~~~~~~
```

Client 1's rock beats client 2's scissors. Client 2 doesn't want to play anymore, so it sends quit. Client 1 doesn't know this and sends another move.

```
[RPSServer] tid 4 quit, but no players are waiting.
[Client tid=4 id=2] exiting
[Client tid=3 id=1] other player quit! I guess I'll go home :(
[Client tid=3 id=1] exiting
Goodbye from choochoos kernel!
```

After client 2 quit, there are no other players waiting to join a game, so the RPSServer sends `OTHER_PLAYER_QUIT` to client 1. When client 1 receives this message, it exits. Client 2 also exits, after sending the quit message.

## Performance Measurement