CS452: Real-Time Programming

Kernel (Part 2): Send-Receive-Reply and Name Server

January 30, 2020

Hugh Pan
20868342

Leanora McVittie
20513649

Usage

The git repository can be found at https://git.uwaterloo.ca/lmcvitti/trains_kernel, and the exact commit hash will be emailed to Professor Karsten; it can also be found as the final commit on the k2 branch in that repository. `performance.txt` is also in the main folder of this repository.
The executable file, `k2.elf`, can be compiled by cloning the above repository and then running `make` in the root directory. It can be run by loading it into RedBoot and using RedBoot's `go` command.
A pre-existing executable, `k2.elf` can be found in `/u/cs452/tftp/ARM/y247pan/` and can also be loaded into RedBoot and run.
The executable will run until the first round of the rock-paper-scissors game has been completed and a winner for that round has been determined. The players' moves and the final outcome of the round will be printed on the terminal. At this point the user can press any key to continue the game to the next round. By default there are two players who play a total of ten rounds of rock paper scissors before exiting. To change the number of players or the number of rounds they play, the constants `MAX_NUM_PLAYERS` and `MATCHUPS_PER_PLAYER` in `include/rps.h` can be edited and the executable recompiled.
The task will not exit on its own. The box must be restarted. This is because the name server never exits and thus the kernel never finishes either.

Send-Receive-Reply

In the send-receive-reply structure, two tasks communicate with each other through a three-stage process. The sending task provides the id of the task it wants to communicate with, the message it wants to send, and a buffer in which to receive the reply. The receiving task provides a buffer in which to receive a message, and once a task has sent to it, it processes the message that is put into the receive buffer. Once it has the information to pass back to the task that the task was waiting for, it replies to that task with a reply message, which is inserted into the sending task's reply buffer, and the sending task is unblocked.

Data Structures and Storage

Because there is so much data that needs to be passed between tasks, and tasks that can be blocked for an indefinite amount of time, during which the data needs to be stored in the kernel, there are a number of techniques used to manage the tasks and the data that belong to those tasks during the message passing process.
The messages, length information, and buffers of blocked tasks are stored within a struct in that task's task descriptor, and can be accessed at any point in the future as long as the task id in question is known. A strictly-FIFO queue is in the task descriptor of any receiver task, which holds the task ids of all tasks waiting to send to it. Any receiving task also manages an unordered list of any tasks that have sent to it and are waiting for a reply. This list exists solely

so that the message passing process can be aborted properly if the receiving task exits before replying and closing the loop.

Send

When a task wishes to send a message, it passes the id of the task it wants to send to, a message, and a reply buffer into the kernel. The pointer to the reply buffer is stored in the sending task's task descriptor. Then the kernel checks to see if the task identified as the receiver is `RECEIVE_BLOCKED` or not. If that task is `RECEIVED_BLOCKED`, it immediately passes the message along to the receiver, moves the receiver back into the ready state and onto the ready queue, and marks the sending task as `REPLY_BLOCKED`. If the receiving task is not `RECEIVE_BLOCKED`, the message is stored in the sending task's task descriptor, the sending task id is added to the receiver's send queue, and the sending task is set to the `SEND_BLOCKED` and will wait for the receiving task to pull it from its send queue.

Receive

The process of receiving is a mirror of the process of sending. When a task wishes to receive a message it passes a receive buffer and a pointer to an int on its stack. When it gets to the kernel, if there is a task on its send queue, the message from that task is put into the send buffer, the send task is set as `REPLY_BLOCKED`, and the receiving task is put back onto the ready queue.
If the send queue is empty, then the buffer and int pointer are put into the receiving task's task descriptor, its status is set to `RECEIVE_BLOCKED`, and it waits for a task to want to send a message to it.

Reply

When a receiving task has received and processed a message from a sender and is ready to reply with the results, it calls `Reply` and passes it the id of the task to reply to, and the message to send to that same task. The kernel then copies that message into the sending task's reply buffer. Then both tasks' states are set to `READY` and they are put back onto the ready queue.

Name Server

The name server initializes itself, create hash table with dynamic bucket size to store name to id mapping using the heap described below. It then calls Receive, blocking itself until a task wants to register with it or request an id. Tasks register with it by calling `RegisterAs`, which sends the task name and id to the server. A task can request a task's id by calling `WhoIs` with the name of the task whose id it needs, which also initiates a send-receive-reply sequence. The name server keeps track of the names and ids it has received through a hashmap, which facilitates fast lookup and reduces the space required to store the name-id matches.

Heap

There are often times we want an arbitrarily-sized container to hold data that can live outside the scope of stack and can be destroyed and created as needed. All information about the heap remains within the kernel state struct, and is allocated and freed by the kernel. When we initialize the kernel, we allocate `0x300000` byes for the heap space which starts from `0x1F00000` and goes to `0x1C00000`. This memory space is further divided into three block groups for finer grained management and to prevent internal fragmentation.
Each block allocates memory of different sizes: `64` bytes, `256` bytes and `1024` bytes. The actual user usable amount is `16` bytes less which is used for metadata. Each time a user requests heap space, with a size provided, for some size space, they receive a block of the smallest size that works.
A memory allocator to manages the block heap. The strategy is that we have six looped linked list in total. And each entry in three of the linked list holds an empty block that a user can use and its metadata. The other linked list is comprised of the blocks that are currently in use. They are called `heap_block_used` and `heap_block_unused` respectively, and each block size has its own set of circular linked lists. Blocks are moved from the unused list to the used list when they are allocated, and then back to the unused list when freed, which recycles the memory block for the next allocation.
The kernel maintains information about the state of the heap, which includes the number of used blocks. One downside of this approach is that even if a user requests 4 bytes, they are allocated a 64 byte block. This is unavoidable with this heap implementation strategy.
Because the kernel manages everything about heap, `Malloc` and `Free` are system calls which switch the execution into kernel mode to request heap space. These are implemented in the same manner as the other kernel system calls for message passing and task management.

Timing

Timing Task

The timing task runs the message passing procedure `10000` times and then divides the total time taken by `10000` to determine the average time taken for an individual send-receive-reply operation. The main task will do this timing test for all the different conditions except compiler optimization and cache, because turning that off requires a separate executable file and we also want machine cache to be clean.

Compiler Optimization

Compiler optimization can be turned on with the `-O3` flag during compile time, and we ensured all pointers to memory locations that belongs to external device were declared with the `volatile` keyword.
When compiler optimization is turned off, on average, message passing performs `3.97` times worse

Message Size

Message size is increased as a constant in the main timer testing task.
When message size is increased from `4` to `64` bytes, message passing performs, on average, `1.61` times worse. When it is increased from `64` to `256`, it becomes `2.05` times worse again.

Caches

When caches are turned off, message passing performs a whopping `14.92` times worse than with caches on. This is by far the most significant single improvement we could make to the operating speed of message passing.

Send-Receive Order

Send-Receive order can be easily flipped by inverting the priorities and the creation order of the sending and receiving tasks.
This change produced by far the least interesting results. Performance was very slightly better when `Receive` is called before `Send`, but the difference in results is well within the margin of error and random chance.

Conclusion

The time is mostly spent in the kernel to schedule the tasks, and to copy and paste between buffers between sender and receiver. As we could see the more buffer size, the more time it spent. Additionally, a lot of time is spent in fetching memory, as evidenced by the drastic performance increase when caches are enabled.

Rock-Paper-Scissors Game

Server

The rock paper scissors server facilitates the playing of the game by acting as the intermediary between all players. The players never send messages to each other directly, going instead through the server. The server accepts three types of messages: register, play, and quit.

Data Structures

The server assigns a unique player id to each player in the registration phase. This id integer is important because the server uses it to index into the three arrays that it uses to keep track of the players. The first array, `player_tids`, holds each player's task id at the index of its unique server-assigned id. The second array, `pairings`, holds the paired up players: if players `4` and `7` were paired in the registration phase, `pairings[4]` is `7`, and `pairings[7]` is `4`. The final array, `moves`, holds the moves that a player has played if its partner has not yet messaged with its move, again at the index of the player id.

Execution

After initialization, the server loops, and on every loop it calls `Receive` and waits (if necessary) to receive a message. It then processes that message and then loops again. It expects to receive one of three types of message.
If it receives a registration message, it checks to see if it has another task waiting on registration, and if it does, it pairs the waiting task and the task that sent the just received message together and prints the pairing to the terminal. It replies to each of the tasks and sends them their player id. It expects to get this player id as a part of all messages the player sends in the future.
If a move message is received, the server checks if the sending task's partner has made a move. If it has not, the move is stored in the `moves` array, and the server calls `Receive` again to get a new message. If the player's partner has made their move, the server determines the result of the match, prints it to the terminal, pauses to wait for user input, and then replies to both the sending task and its partner with their results: win, loss, or tie.
If a quit message is received, the server sets the partner's entry in the partners array to `-1`, to indicate that it's partner has quit, and decrements its player counter by one.
When the player counter reaches `0` the server stops looping and exits.

Players

A player task starts by sending a registration request to the server. It then plays a predefined (`MATCHUPS_PER_PLAYER`) number of rounds of rock paper scissors. In each round it randomly

chooses rock, paper, or scissors, prints that on the terminal, and sends that choice to the server. It receives back as a reply, whether it won, lost, or tied the round, and increments the associated counter. In this way the player keeps track of its own win/loss record. After playing the predefined number of matches, it sends a quit message to the game server, prints its win/loss/tie record, and exits.

Program Output

Because the rock-paper-scissor moves are chosen at random, the output is non-deterministic, but an example, with two players and ten matchups, is below (this was a particularly sad run of form from task 4):

```
1.   <4> partners with <3>
2.   <3> plays <r>
3.   <4> plays <r>
4.   <4> tied with <3>
5.   <3> plays <s>
6.   <4> plays <p>
7.   <3> won and <4> lost this round
8.   <3> plays <p>
9.   <4> plays <r>
10.  <3> won and <4> lost this round
11.  <3> plays <s>
12.  <4> plays <p>
13.  <3> won and <4> lost this round
14.  <3> plays <s>
15.  <4> plays <p>
16.  <3> won and <4> lost this round
17.  <3> plays <r>
18.  <4> plays <s>
19.  <3> won and <4> lost this round
20.  <3> plays <r>
21.  <4> plays <s>
22.  <3> won and <4> lost this round
23.  <3> plays <r>
24.  <4> plays <s>
25.  <3> won and <4> lost this round
26.  <3> plays <s>
27.  <4> plays <p>
28.  <3> won and <4> lost this round
29.  <3> plays <s>
30.  <4> plays <p>
31.  <3> won and <4> lost this round
```

```
32.   <3> game record: w: 9 t: 1 l: 0
33.   <4> game record: w: 0 t: 1 l: 9
```