CS 452 Assignment 2

Victor Scurtu (20678028) and Alex Danila (20516806)

January 31, 2020

Operating Notes

The executable file is found in the student environment under:

```
/u/cs452/tftp/ARM/vscurtu/k2/kernel
```

The code can be found in the following repository:

```
https://git.uwaterloo.ca/vscurtu/cs452_kernel
```

To build, use the following commands at the root of the repository on a student environment computer:

```
make
```

To run the precompiled kernel, use the following command in redboot:

```
load ARM/vscurtu/k2/kernel; go
```

The program executes, prints, then requires user input to run. Any keypresses can be used to advance its execution until it finishes.

Add Headings (Format > Paragraph styles) and they will appear in your table of contents.

Table of Contents

Program Description

This program description assumes that the report for K1 has been read before hand. It describes only the set of changes from the past assignment, and does not go into detail about kernel components implemented in K1.

Initialization

The initialization procedure remains the same as in K1, with a few additions. Namely, the kernel now also initializes the message queue and enables the L1I, L1D, and L2 caches.

Caching is enabled via a call into an assembly routine, enable_cache. The routine reads r1 of coprocessor 15 into r0. This is the control register of the system control coprocessor. It uses orr to set the required bits to enable L2 cache, L1I cache, L1D cache, and the MMU - bits 26, 12, 2 and 0 respectively - and writes that value back from r0 to CP15's r1.

After initializing the task list and priority queue in kinit(), the message queue is initialized by calling the init_message_queue function. This works similarly to the init_pqueue function, but works on the global message queue instead. It initializes a bank of id nodes to all point to their following neighbor. These are all considered free nodes at initialization and as such a pointer to the list of "free" nodes is initialized to the first node in this array. The message queues are stored as pointers to heads and tails in this node bank. Their heads and tails are initialized to null pointers and as such we begin with zero messages in any tasks message queue. There is one message queue for every task.

Message Sending

Message sending is built using a queue for every task where senders are processed on a first come first serve basis. Tasks that are waiting to send to another task are put on that task's message queue.

In order to support the extra arguments, syscall has been altered. Pray I don't alter it any further. It now takes 6 arguments, the first is the call code and the rest are the arguments from the syscall wrapper. Any unused arguments are 0.

When a task, say R, calls Receive, the kernel first puts the task into state TASK_RECV_WAIT and checks if R's queue has any waiting tasks. If so, it grabs the first task from the queue, copies its message into R's buffer, and puts R back into the TASK_READY state, and puts it back in the schedule queue. Otherwise, if there are no tasks waiting on R's message queue, R is left in TASK_RECV_WAIT until a task calls Send to it.

When a task, say S, calls Send to R, the kernel first puts the task into state TASK_SEND_WAIT and then checks if R is in TASK_RECV_WAIT. If so, then R must have called Receive and had an empty message queue, and so the kernel immediately copies S's buffer to R's without queuing S on R's message queue. Then it puts R into TASK_READY and back into the schedule queue, and S into TASK_RPLY_WAIT. Otherwise, if R is not in TASK_RECV_WAIT, S is placed into R's message queue.

S will wait on R's message queue until either it reaches the front and R calls Receive, as described above, or R terminates, either by a return or call to Exit. If R terminates, all the tasks in R's message queue are set to TASK_READY, have their return code set to -2 since the SRR transaction could not be completed, and are put back into the schedule.

In addition to the above, if the recipient task is ever invalid or in the TASK_ZOMBIE state, Send returns -1.

When a task is in TASK_RPLY_WAIT, it is not scheduled and must wait until another task calls Reply to it. Reply does not block. Reply either finds the specified task waiting in TASK_RPLY_WAIT and completes the message copy, or it returns an error code if the task does not exist or is not in TASK_RPLY_WAIT.

On the back end, the kernel has exposed two new functions to support message passing: kcopymessage, kcopyreply. Each of these functions determines the correct arguments for the source and destination buffer and their lengths by probing the source and destination task stacks using the Frame structure. The kernel uses these two functions for processing Send/Receive and Reply respectively, since the former requires the first three arguments to Send and the latter requires the last two.

Frame has two new fields, stk0 and stk1, to support calls to Send which are described in the Structures Changed section. These arguments are found in r1, r2, and r3 for kcopymessage since syscalls include the call code first. This makes the arguments needed by kcopyreply the 5th and 6th argument which will be in stk0 and stk1, by C calling convention.

In addition, there is a new internal function, copy, that is called by both kcopymessage and kcopyreply to handle the actual copy. It finds shorter of the source buffer and destination buffer length, and the copies the full length of the shortest buffer from the source to the destination directly. It does not assume anything about the content of the message. Specifically, it does not assume messages are null-terminated strings, or that they have any other format, so it does not copy any less than the length of the shortest buffer.

Once the copy has been completed, kcopymessage will set the recipient task's state to TASK_READY and its return code to the actual copied length, and put it back on the schedule queue. It will also set the sender to TASK_RPLY_WAIT. Likewise, kcopyreply will set the sender's state to TASK_READY, set its return code, and put it back on the schedule, as well as the replier's.

Name Server

The name server is a user task that provides name resolution for other tasks. The interface and implementation are defined in name_server.h and name_server.c respectively. The name server is communicated with using the RegisterAs and WhoIs functions in name_server.h. These are wrappers for message sends to the name_server which combine the name provided with a prefix that encodes the operation type (RegisterAs or WhoIs) before sending it to the server. They also return -1 if the name server task id is invalid. The name server is created within the umain task in user.c, the entry point for all user tasks.

As discussed in the assignment, the task id of the name server must be known to the RegisterAs() and WhoIs() routines, which are executed by other tasks. This task id is stored in the name_server_id variable in name_server.h. It is populated at the beginning of the umain task after the name_server task is created.

The _register_as and _who_is functions are the implementations for RegisterAs and WhoIs respectively and are called in the name_server task after a new message is received and parsed. The name server is implemented using an array of strings, indexed by the id of each task. Every time the name for a task is needed, the array is indexed using the task id. Every time a task id is needed for a name, the entire list of names is searched until a matching string is found.

The name_server function is the task's entry point. It initializes the list of names to all empty strings and then begins an infinite loop which works as follows. At the start of the loop a Receive call is made for a message. The message is then parsed, using the value of the first character to pick what the operation is to be done on the name, and the remainder of the string as the name.

The first character of the message will have been set appropriately by one of the two wrappers. The corresponding _who_is or _register_as function is called. Once these return, their result is sent back to the task that made the request using a Reply call. This marks the end of the iteration.

_who_is works by iterating through the list of names, comparing each to the name provided, then returning the index of the name that matched which represents the task id associated with that name. If the name is not in the list, indicating that the name has not been registered in the name server, TASK_DNE is returned.

_register_as works by taking first accepting a name that is guaranteed to not be too long. The max size of a name is defined as MAX_NAME_LENGTH. The name provided will not be too long since it has been automatically truncated by the receive function only accepting a message of size MAX_NAME_LENGTH. This has to be done since the array of strings is a fixed size 2d array. Inside _register_as, _who_is is first called on the name to find out if its already in the name server. If it is, the name for that task is set to an empty string. The name is then copied into the name array at the requesting task id's index and 0 is returned to indicate success.

Since we do no reuse task descriptors currently, tasks that exit do not deregister from the name server. Since the RPS server uses a single signed char in the reply message as the returned task ID and uses the negative range of the char for errors, it's only able to return 0-127 for valid IDs. This means MAX_TASKS_ALLOWED must be at most 128 and there is an assert that checks this.

In order to work with the strings in the nameserver, a few string utility functions were added in string_utility.h and string_utility.c. These closely mirror their counterparts in the standard libraries string.h.

We were unable to coerce the compiler to find and include string.h, however the names strcpy, strlen, and strcmp could not be used since the compiler specifically checks for those function names (even if string.h is not included) and throws a redefinition error. As a work around, the standard library function names prepended with a '_' were used.

RPS Server and Clients

The RPS server is implemented in much the same way as the nameserver when it comes to how it parses requests, and how the interface functions wrap message sends. The RPS server is communicated with by using the Signup, Play, and Quit functions in rps_server.h. These are wrappers for 2 byte message sends to the rps_server which store the operation done in the first

byte and the argument in the second. Currently, the argument is only used by the Play function to communicate what move was played. The task id of the RPS server is obtained by making a WhoIs call to the nameserver for “rps_server”. The RPS server is created within the umain task in user.c.

The _signup, _play, and _quit functions are the implementations for Signup, Play, and Quit respectively and are called in the rps_server task when a new message is received and parsed.

The RPS server is implemented using several variables that store the state of the ongoing games. The queued_client variable stores the players queued up to play. Since this can only ever be one player, it’s a simple integer storing its task id. The games array stores all the currently running games, each game being a struct storing the players involved, their moves, and whether or not its running. The running field is used to tell if a game struct has been allocated or not. In order to find which game an id is in, the entire array of games is iterated over and each game is checked for the presence of the id. At the moment there is a maximum of 16 games, if this is exceeded a fatal message is printed.

The rps_server function is the task’s entry point. It first registers itself as “rps_server” on the names server. Afterwards, it initializes the list of games to be all not running and with no players (denoted by setting both player fields to -1) and then begins an infinite loop which works as follows. At the start of the loop a Receive call is made for a message. The message is then parsed, using the value of the first character to pick what the operation is to be done, and the second as an argument if the operation requires it. The first character of the message will have been set appropriately by one of the two wrappers. The corresponding _signup, _play, or _quit function is then called. These functions are responsible for calling Reply to requesting tasks when appropriate. When they return, the iteration is over.

The _signup, _play, and _quit functions are implemented in rps_server.c. Their functioning is outlined by the provided inline comments. The RPS clients are found in rps_client.c and rps_client.h. They are created in user.c.

Structures Changed

Frame: From K1, this is a structure used to cleanly access the stack-saved task registers in memory. This structure has the same layout in memory as the saved registers that we save and restore in enter_user and enter_kernel, so a pointer to a stack that has saved registers at the top can simply be cast to a “Frame” struct then all registers and the cspr can be accessed using its fields.

All previous uses for the struct have remained the same but two extra fields (stk0 and stk1) have been added. These are used to pass additional arguments during a syscall, mainly SYSCALL_SEND.

Structures Added

MessageQueue: This structure is very similar to the pqueue implemented in K1. It keeps track of all message queues. There is one message queue for each task, which stores the id's of all the tasks which have sent a message to the task in a FIFO order. The actual messages sent are stored on the sending task's stacks. Unlike the pqueue struct, there is no interaction between the message queues for different tasks in the functions provided in the interface.

The interface for the global message queue includes the following:

init_message_queue: Initializes the global MessageQueue and must be called before calling any of the following functions.

push_message: Takes an id and messaging id and pushes the messaging task onto the back of the first id's message queue.

pop_message: Takes an id, removes the first messaging task on it's message queue, and returns it.

peek_message: Takes an id and returns the first messaging task on it's message queue.

This structure contains an array of id_nodes called "data" that's the size of the maximum number of tasks, since no more than one message per task can be sending at one time. This works as a fixed size allocator, providing nodes for all message queues. All nodes in this list are initially considered to be free, and have their next pointers pointing to their neighbor. Thus, the first node represents the head of a linked list of free nodes. The field "free" holds this head. Finally we have the queues field which holds an array the size of the maximum number of tasks. The elements in the array are message_queue structs which simply hold a head and a tail for a linked list representing one priority level queue. The indices of this array correspond to task ids.

pop_message and push_message work by moving nodes between queue lists and the free list. All nodes sit in the same "data" array, but the lists they represent change. When we push_message, a node is taken from the free list and becomes the new tail of the desired tasks queue. When we

pop_message a node is taken from the desired task's queue and becomes the new head of the free list. These two processes represent allocating and deallocating a id_node respectively.

Known Bugs

- A few keypresses are needed on the last RPS test that do not advance the print. Pressing again should advance the print. This was intentionally left in since the current placement of bwgetc calls is relatively simple. Please make several keypresses if it seems like the program has hung.

General Notes

- There is an implementation of memset added in memset.c/memset.h. It is used by gcc when the -O3 flag is enabled.
- The timer module communicates with the 40-bit timer. Reading the timer reads only the lower 32 bits. This is done for speed in the timing tests, as overflowing into the higher bits has not been a problem.

Timing

Results in microseconds; also available in performance.txt

opt	cache	S	4	9.36
opt	cache	S	64	11.37
opt	cache	S	256	19.07
opt	cache	R	4	8.709
opt	cache	R	64	10.69
opt	cache	R	256	18.37
opt	nocache	S	4	163.1
opt	nocache	S	64	199.4
opt	nocache	S	256	342.1
opt	nocache	R	4	148.6
opt	nocache	R	64	183.7
opt	nocache	R	256	326.8
noopt	cache	S	4	30.92
noopt	cache	S	64	58.5
noopt	cache	S	256	147.4
noopt	cache	R	4	26.61
noopt	cache	R	64	54.54
noopt	cache	R	256	143.2
noopt	nocache	S	4	514.5
noopt	nocache	S	64	974.4
noopt	nocache	S	256	2447
noopt	nocache	R	4	439.8
noopt	nocache	R	64	899.2
noopt	nocache	R	256	2372

Methodology and Analysis

Each test was run 10,000 times to average out minor differences in timing and minimize the effects of one-time overhead such as cache warmup. There was no pre-test warmup runs as it was assumed that the effects of warmup would become negligible over 10,000 iterations.

The body of a test loop contained only a single call to Send that would attempt to send to a task calling Receive and Reply to the sender in an infinite loop. There were 6 test loops, one for every possible combination of message size and sender/receiver priority at a given optimization and caching setting. Priorities were used to enforce sender-first or receiver-first.

The 40-bit debug timer was used to measure the cumulative time of all 10,000 iterations of each test. The timer was reset and started before running any test loop and the number of ticks was measured immediately after completion of the test loop. By timing before and after calls to Send, the timing results produced encompassed the entire SRR operation. Information about the test that ran and the number of ticks was then printed to the screen.

The number of ticks was then used to calculate the average running time of a single SRR operation according to the following equation:

$$t = \frac{num\ ticks}{num\ tests} \times \frac{1000000}{983040}$$

The constant $\frac{1000000}{983040}$ is the ratio of ticks to microseconds based on the frequency of the 40-bit timer - 984.03 kHz or 983040 Hz. Thus the equation averages the number of ticks per test and converts the average to microseconds. If this seems needlessly verbose, it is only because I got into a 30 minute argument about why this math is correct.

The results show several interesting things. First, there is a minor but consistent difference between sender-first and receiver-first running, with receiver-first being faster. This is likely due to the fact that if the sender runs first, it must be put on the receivers message queue, but if the receiver runs first, the sender will find it in TASK_RECV_WAIT and the kernel will copy the message without accessing the message queue. This implies one less trip to memory when the receiver runs first.

Compared to no optimizations and no cache, enabling the cache has a bigger impact on performance than enabling optimizations, suggesting that the majority of the time in SRR is spent accessing memory. This makes sense since the kernel must access at least two tasks' stacks and also some scheduling structures to process an SRR, potentially more depending on which task runs first.

Unsurprisingly, enabling both caching and optimizations is even more performant and longer messages make the SRR process take longer. However, an additional reason that enabling optimizations makes the code more performant is due to inlining, which requires optimizations enabled in order to run.

In order to maintain code clarity and maintainability, inlining was done programmatically using a Python 3 script. The script is described in the Inlining section at the end of this document.

RPS Output

When running the kernel, it will print the output of various RPS client interactions. To progress through these, user input is required. Any keypresses can be used to advance until all interactions have finished. A request for input will occur after each RPS round, and once each time the test description is printed. Due to the scheduling changing the order of prints unpredictably, the round results may be at the end, middle, or start of the prints proceeding a keypress. Regardless, it is guaranteed that a maximum of one round will be printed per keypress. In other words, each keypress will show the results of either 0 or 1 rounds. A round is only considered as both sides making a play.

Prints that describe what command was finished are printed after each Signup, Play, or Quit command in each client. Every print inside clients is preceded by their task id in square brackets.

If not otherwise mentioned in the output, all tasks within each test were run at the same priority level as each other.

The following are the outputs for each RPS test:

```
Both clients play once then quit:
[4]rps_rock_client signed up.
[5]rps_paper_client signed up.
[4]rps_rock_client played rock and LOST.
[5]rps_paper_client played paper and WON.
[4]rps_rock_client quit.
[5]rps_paper_client quit.
```

Both clients play 3 times before quitting:

```
[6]rps_rock_lover_client signed up.
[7]rps_paper_lover_client signed up.
[6]rps_rock_lover_client played rock and LOST.
[7]rps_paper_lover_client played paper and WON.
[6]rps_rock_lover_client played rock and LOST.
[7]rps_paper_lover_client played paper and WON.
[6]rps_rock_lover_client played rock and LOST.
[7]rps_paper_lover_client played paper and WON.
[6]rps_rock_lover_client quit.
[7]rps_paper_lover_client quit.
```

Both clients don't play and quit immediately:

```
[8]rps_quitter_client signed up.
[9]rps_quitter_client signed up.
[8]rps_quitter_client quit.
[9]rps_quitter_client quit.
```

Second client doesnt play and quits immediately:

```
[10]rps_rock_lover_client signed up.
[11]rps_quitter_client signed up.
[11]rps_quitter_client quit.
[10]rps_rock_lover_client played rock and OPPONENT QUIT.
[10]rps_rock_lover_client played rock and OPPONENT QUIT.
[10]rps_rock_lover_client played rock and OPPONENT QUIT.
[10]rps_rock_lover_client quit.
```

First client doesn't play and quits immediately:

```
[12]rps_quitter_client signed up.
[13]rps_rock_lover_client signed up.
[12]rps_quitter_client quit.
[13]rps_rock_lover_client played rock and OPPONENT QUIT.
[13]rps_rock_lover_client played rock and OPPONENT QUIT.
[13]rps_rock_lover_client played rock and OPPONENT QUIT.
[13]rps_rock_lover_client quit.
```

An abusive client and a normal client that plays once are created.
Without initially signing up, the abusive client tries to Quit -> Play(PAPER) -> Signup -> Quit -> Play(PAPER):

```
[14]rps_abusive_client quit before signing up.
[14]rps_abusive_client played paper and WASNT SIGNED UP
[14]rps_abusive_client signed up.
[15]rps_rock_client signed up.
[14]rps_abusive_client quit.
[15]rps_rock_client played rock and OPPONENT QUIT.
[14]rps_abusive_client played paper and WASNT SIGNED UP
[15]rps_rock_client quit.
```

First client higher priority than second:

```
[16]rps_rock_client signed up.
[17]rps_paper_client signed up.
[16]rps_rock_client played rock and LOST.
[16]rps_rock_client quit.
[17]rps_paper_client played paper and WON.
[17]rps_paper_client quit.
```

First client lower priority than second:

```
[19]rps_paper_client signed up.
[18]rps_rock_client signed up.
[19]rps_paper_client played paper and WON.
[19]rps_paper_client quit.
[18]rps_rock_client played rock and LOST.
[18]rps_rock_client quit.
```

Two clients try every combination of games possible:

```
[20]player_1 signed up.
[21]player_2 signed up.
[20]player_1 played rock and TIED.
[21]player_2 played rock and TIED.
[20]player_1 played rock and LOST.
[21]player_2 played paper and WON.
[20]player_1 played rock and WON.
[21]player_2 played scissors and LOST.
[20]player_1 played paper and WON.
[21]player_2 played rock and LOST.
[20]player_1 played paper and TIED.
[21]player_2 played paper and TIED.
[20]player_1 played paper and LOST.
[21]player_2 played scissors and WON.
[20]player_1 played scissors and LOST.
[21]player_2 played rock and WON.
[20]player_1 played scissors and WON.
[21]player_2 played paper and LOST.
[20]player_1 played scissors and TIED.
[21]player_2 played scissors and TIED.
[20]player_1 quit.
[21]player_2 quit.
```

```
Four clients join and begin playing.
First two play 4 times and the last two play 3 times.
[22]rps_player_1 signed up.
[23]rps_player_2 signed up.
[24]rps_player_3 signed up.
[25]rps_player_4 signed up.
[22]rps_player_1 played paper and WON.
[23]rps_player_2 played rock and LOST.
[24]rps_player_3 played rock and TIED.
[25]rps_player_4 played rock and TIED.
[22]rps_player_1 played paper and WON.
[23]rps_player_2 played rock and LOST.
[24]rps_player_3 played rock and TIED.
[25]rps_player_4 played rock and TIED.
[22]rps_player_1 played paper and WON.
[23]rps_player_2 played rock and LOST.
[24]rps_player_3 played rock and TIED.
[25]rps_player_4 played rock and TIED.
[22]rps_player_1 played paper and WON.
[23]rps_player_2 played rock and LOST.
[24]rps_player_3 quit.
[25]rps_player_4 quit.
[22]rps_player_1 quit.
[23]rps_player_2 quit.
Kernel: exiting
```

Inlining

Okay this is a fun one. There's a Python 3 script - kernel/inline.py - which handles combining all of the kernel source into a single monolithic file. It takes two arguments - a path to the source directory and a path to the build directory. This is passed in by the Makefile that calls it.

First, it uses ctags to find all the typedefs, structs, and functions defined in the source. It processes the typedefs and structs into a list of types it uses to identify function signatures, and stores the functions themselves for later. It then pairs off all the .c and .h files into objects. The one unpaired .c file is main.c, which cannot be inlined, and the rest of the .h files are header-only definitions which also get their own objects.

Next, it outputs code into a file called inline-all.c in the build directory. First it outputs the .h files starting with constants.h the rest of the unpaired .h files, followed by the rest of the .h files. During output, it comments out any function declarations that were found by ctags, as they will be declared later. It also comments out #include that have quoted headers, as all local headers will be copied in.

Then, it writes out static inline declarations for all the functions found by ctags in a declaration block in inline-all.c, with exceptions for panic and print_lr, as they are called from assembly and cannot be inlined.

Finally, it copies out all the paired .c files into inline-all.c, but when it sees a function signature, it outputs it with __attribute__((always_inline)) above and static inline before the original signature. The last file to be copied is main.c, which is a special case since main cannot be inlined.

This is obviously all super janky and is not considered stable enough to run as default. It may never be, but it sure was interesting to make. It exists just in case we run into performance issues later that can be fixed by inlining, but without requiring us to change our code significantly to inline.

If you wish to run it anyway, replace `make` with `make inline` in the build instructions. It will attempt to run the script to produce build/inline-all.c. If this succeeds, it will then attempt to compile the created inline-all.c file. If by some miracle that also succeeds, k2.elf will be generated as expected.

I hope you at least find this amusing. :)