CS 452 Kernel 2

Benjamin Zhao, Kyle Verhoog

Jan 29, 2018

Contents

Operation

The ELF file is located at `/u/cs452/tftp/ARM/yl4zhao/kernel2.elf` and can be initiated using the typical `load` command.

After loading the kernel should execute the sample test required for the assignment. The games are separated by `bwgetc` which requires a key to be pressed in order to simulate the next game.

Structure

Implementation

Our implementation follows the same basic loop shown in class:

```
int main(void) {
  initialize();
  while(true) {
    TaskDescriptor *td = schedule();

    if (!td) break;

    TaskRequest req = activate(td);
    handle(td, req);
  }
}
```

Stack & Memory Layout

We implement our stacks growing downward in memory. The kernel stack begins in the middle of the 32MB of memory and the user stacks start at the top of memory.

User Stacks

The user stacks start at the top of memory and are each (for now) 1MB in size. So each consequent user stack is 1MB separated in memory and therefore, there can only be 16 active tasks. Of course, this number is arbitrary for now, and will change once we deterimine the number of tasks needed for our kernel.

The memory layout is defined in `/include/kernel/kernel.h`. Here we define where the kernel and user stacks begin and how large user stacks are.

Task Descriptor

The task descriptor contains the following fields:

- `tid`: explained in the next section
- `sp`: user task stack pointer
- `psr`: user task status register
- `task`: pointer to function of task
- `parent`: pointer to parent task descriptor
- `next`: pointer to next task descriptor in priority queue
- `priority`: the tasks priority
- `ret`: the value to be returned to the task on syscall
- `sendq`: A circular buffer of size equal to the number of tasks

See: `include/task/task.h` for details.

Task Identification

Tasks are uniquely indentified by tids which are tracked as unsigned 32-bit integers. Every task has a unique tid, and currently all zombie tasks return the tid upon exiting. The upper 16 bits of the tid represent a version number and the lower 16 bits represent an id. All tids start at version 0.

For example, the tid `0x0000 0001` represents a task with id `1` and version `0`.

See: `include/task/task.h` for details.

TidTracker

The TidTracker is a distributer which distributes unique tids upon request. The tids are pre-generated when the kernel starts running. Tids are re-used (with an incremented version) when a task exits and the tid is returned to the distributer for re-use. When the distributer runs out of tids, likely either two things have happened:

- All tids are in use

- The use of a tid has exceeded 2^{16} re-issues in which an overflow may cause undefined behaviours

The TidTracker uses a circular buffer, prefilled with some maximum number of tids allowed to be allocated to tasks at once. When a task is created, the kernel requests a tid using `tt_get()` from the TidTracker. The tracker then takes the first tid from the queue, pops it and gives it to the kernel. When the task exits, the kernel calls `tt_return()` to return the tid to the tracker. The tracker appends `1 << 16` to the tid, and inserts to the end of the buffer.

```
int tid = tt_get(&tid_tracker);
tt_return(td->tid, &tid_tracker);
```

See: `include/task/task.h` for details.

The circular buffer is implemented using a fixed-sized array, with a start and end index pointing to the head and tail of the queue respectively. The queue has constant $O(1)$ time insertion as well as deletion of head. The circular buffer itself does not have any overflow guards, however we rely on the limited number of tids to ensure we never reach an overflow.

See `src/lib/circularBuffer/circularbuffer.c` for details.

Context Switch

Instead of rephrasing the context switch, here is an annotated version of the function `activate` which handles both the kernel to user and user to kernel switches.

The activate function runs a set of inline assembly macros which perform the saving of states to stacks, register manipulation, priviledge changes and jumps to and from user land.

Kernel to User

```
PUSH_STACK("r0-r12, lr");  // Store Kernel State
asm("mov r8, %0"::"r"(td->ret));
PUSH_STACK("r8");          // Push ret val to stack as temp
WRITE_SPSR(td->psr);       // Install the SPSR from the TaskDescriptor
SET_CPSR(SYSTEM_MODE);     // Change to System mode
WRITE_SP(td->sp);          // Change the stack pointer to the task's stack
POP_STACK("r4");           // Load instruction after swi (r4) from user stack
SET_CPSR(KERNEL_MODE);     // Change to Kernel mode
asm("mov lr, r4;");        // Save into kernel lr for loading
SET_CPSR(SYSTEM_MODE);     // Change to System mode
POP_STACK("r0-r12, lr");   // Load the User Trap Frame
SET_CPSR(KERNEL_MODE);     // Switch back to Kernel mode
```

```
POP_STACK("r0");            // Set r0 with the new return value from stack
REVERSE_SWI();              // Move to the user task
```

User to Kernel (on `SWI`)

```
asm("KERNEL_ENTRY:");
SET_CPSR(SYSTEM_MODE);     // Change to System mode
PUSH_STACK("r0-r12, lr"); // Save the user state
SET_CPSR(KERNEL_MODE);     // Change to Kernel mode
asm("mov r3, lr");         // Save lr to stratch r3
SET_CPSR(SYSTEM_MODE);     // Change to System mode
PUSH_STACK("r3");          // Save the lr(r3)
SET_CPSR(KERNEL_MODE);     // Change back to Kernel mode
POP_STACK("r0-r12");       // Restore the kernel stack
SET_CPSR(SYSTEM_MODE);     // Change back to System mode
READ_SP(td->sp);           // Save the user sp to TaskDescriptor's sp
SET_CPSR(KERNEL_MODE);     // Change back to Kernel mode
READ_SPSR(td->psr);        // Save the spsr to the TaskDescriptor's psr
SWI_ARG_FETCH("r0");       // Manually put swi arg in r0, avoid overhead of return
POP_STACK("lr");           // Restore link register to return properly
```

See `include/asm/asm.h` and `src/kernel/kernel.c` for details.

With our implementation of the context switch all three of the link registers are saved and restored correctly.

Limitations

Number of Mode Switches

As depicted above you can see that there are a number of changes in mode. This could have potential performance issues and is probably an indicator that we should refactor.

Switching Modes

Currently `SET_CPSR(MODE)` is dependent on the usage of a register, namely `r12`. This means that when we re-enter the kernel, we must switch to system mode to access the user stack pointer, corrupting `r12`.

Scheduling

Scheduling is done by managing a set of task queues. There are 32 priorities and hence 32 task queues. Tasks are placed in a task queue corresponding to its

priority. The next task that is scheduled is the one at the head of the highest non-empty priority queue.

A 32-bit integer is used to maintain state information about which priority has tasks available. When the i-th bit is flipped, them there are tasks available in the priority i queue.

Refer to `k1.pdf` for more information.

Messaging

Messaging is done using `Send`, `Receive` and `Reply`. The implementations of these is similar to as described in class and the notes.

Send

```
int Send(int tid, void *msg, int msg_len, void *reply, int reply_len);
```

Sends a message to a receiver task `tid` by copying `msg` to the receivers `msg`. If there is no receiver waiting, the sender is placed in the receiver's `sendq` and blocked.

Receive

```
int Receive(int *tid, void *msg, int msg_len);
```

Receives a message from a sender `*tid` into `msg`, or waits until there is one.

Reply

```
int Reply(int tid, void *reply, int reply_len);
```

Meant to be called from a receiver to return results back to a sender. `reply` is copied from the receiver to the sender.

Note on Error Handling

Due to time limitations there is little to no error handling done around messaging, besides asserts. It is in our backlog to implement error handling for messaging.

Name Server

The name server is implemented as a user task with a couple special system calls which allow it to initialize with the kernel.

RegisterNS

```
int RegisterNS();
```

Registers the calling task with the kernel as the nameserver.

GetNS

```
int GetNS();
```

Returns the tid of the current nameserver registered with the kernel.

Implementation

The Nameserver uses a basic `<int, int>` mapping from an integer name to a tid with a fixed size array in O(1) time.

Tasks can register their own tid to a static integer name using `RegisterAs`. Tasks can also query the tid of a static name from the nameserver using `WhoIs`. Both are described in more detail below.

RegisterAs

```
int RegisterAs(int id);
```

`RegisterAs` queries the tid of the registered NameServer using `GetNS`, then calls the `Send` syscall with the nameserver tid and expects the NameServer to return a success or failure on registering the user task into the nameserver.

WhoIs

```
int WhoIs(int id);
```

`WhoIs` queries the tid of the registed NameServer using `GetNS`, and calls the `Send` syscall with the name server tid and expects the NameServer to return the associated tid to the name.

RPS Game

A Rock-Paper-Scissor Client/Server Test on the communication with the kernel. (Note: We named acronym RPS as RPC (Rock-Paper-sCissor) by accident.)

```
void RPCClient();  \\Plays a move based on Tid
void RPCClient2(); \\Always quits
void RPCServer();  \\Server handles ONLY ONE GAME AT A TIME (All other players are queued)
```

Data Structures and Algorithms

The RPS Client/Server tasks communicate with each other to play the game. Clients signup to the Server, which are then queued and paired if two or more clients have signed up. Note that only one game can happen at a time, so all subsequent players who signed up are queued until the first pair has finished playing.

One in every 3 players spawned is toxic and will quit the game when paired. Every other player will make a move based on their tid mod 3 (Rock - 0, Paper - 1, Scissor - 2). A total of 50 players are created, (starting from tid 3), thus 25 games are played.

```
typedef enum RPCservice{
    S_Signup = 0,
    S_Play = 1,
    S_Quit = 2,
    S_Close = 3
}RPCservice;
```

`Signup` is a service call to the server to queue the client up for play. Once matched, the server will reply to both players.

`Play` is a service call to the server to play the move of their choice. Once both players have made their move (or quit), the server will reply to both players with the outcome.

`Quit` (same as play).

`Close` is a service call to shutdown the server.

Syscalls

The kernel supports the following syscalls:

- `Assert`: Invoked via `assert` provides a method of testing in tasks
- `Create`: Creates another task to be put on the kernel's task schedule
- `GetTid`: Get the task's tid
- `GetParentTid`: Get the parent's task tid
- `Pass`: Give control away
- `Exit`: Become a zombie
- `Send`: sends a message to a tid
- `Receive`: receives a message from another task
- `Reply`: replies to a sender with a result
- `RegisterNS`: Registers a user task as the nameserver
- `GetNS`: Returns the tid of the current nameserver

Priorities

- `InitTask`: 3
- `NameServer`: 31
- `RPCServer`: 30
- `RPCClient`: 5
- `RPCClient`: 5
- `StopServer`: 1

`InitTask` is the first task to be run. It creates the `NameServer` first, which is of highest priority since we want all lookup requests to be handled first. The `RPCServer` has the second highest priority since it needs to pair up players and perform rounds as soon as possible.

Players have the next highest priority as all players need to signup and play while `InitTask` creates more players (when the two clients have queued and `InitTask` is the only one left ready). `InitTask` will also create the `StopServer` task which stops the `NameServer` and the `RPCServer`. This task needs to have the lowest priority as it should only stop these services when we can guarantee nothing is left to run.

Output

Raw Output (Snippet)

```
Player: 3 VS. Player 4
----------------------
Game End! - Player 3 resigned!

Press Any Key to Continue:


Player: 5 VS. Player 6
----------------------
Game End! - Player 6 resigned!

Press Any Key to Continue:


Player: 7 VS. Player 8
----------------------
Game End! - Player 8 Won!

Press Any Key to Continue:
```

```
.
. (25 total games)
.

STOPPING NAME SERVER
```

Explanation

The ordering of the output is straightforward by the nature of the server. It matches pairs of players in the order they arrive to the server. Since 50 players are created in order player 3 plays player 4. The player numbers are just their tid. Since the initial task, name server and game server are running, the first player is tid 3 (version 0).

Every 3 players is a toxic player and quits the game after a match but before the game is played. (Thus every two games ends in a resignation).

A players move is determined as described in the RPS Game section.

Since there are 50 players, 25 games are played.

Measurements

We made the measurements using the 40-bit timer and averaged over a number of iterations to get accurate values. (However we may have been running our code with asserts in place, oops!).

Msg Length	Caches	Send before Reply	-O2	Time (μs)
4	off	yes	off	354.00391
64	off	yes	off	846.35417
4	on	yes	off	25.43132
64	on	yes	off	59.00065
4	off	no	off	325.52083
64	off	no	off	817.87109
4	on	no	off	23.396810

Msg Length	Caches	Send before Reply	-O2	Time (μs)
64	on	no	off	56.966146
4	off	yes	on	188.19173
64	off	yes	on	329.58984
4	on	yes	on	12.20703
64	on	yes	on	22.37956
4	off	no	on	224.81283
64	off	no	on	316.36556
4	on	no	on	11.18978
64	on	no	on	21.36230

Explanation

Although we have not gotten around to verifying, we believe that the majority of time is being spent copying bytes. This is made apparent by the huge difference in timing between tests using 64 bytes versus using 4 bytes.

With optimizations and caching on we see that our SRR takes only 11 microseconds for 4 bytes. This time is most likely in the context switch and the code for send, receive and reply. Most notably the reads and writes from the user stacks when copying messages.

We simply loop through each of the void pointers and copy. However, this is cache inefficient as we will load the senders stack into the cache and then cache-miss on the receivers stack and the other way on the next iteration.

A better, future solution would be to load and store in batches when copying data between two different parts of memory.

Source Code and Hashes

- 48e59cb1b1ec07bbddb440c303c00893 ./include/asm/asm.h
- 1301fccfac1792e069328bf872105a6c ./include/types.h
- e103db461c9019705195e7b965feaf58 ./include/bwio/bwio.h
- 1665ad3c8f2988556b8d1381a293da88 ./include/lib/circularbuffer.h

- aa73bb56e49e6ab222e428207eac10cb ./include/ts7200.h
- 95730ae3bfc1619bc03087b87d92574d ./include/kernel/kernel.h
- de694d347f568a8d62cf2c5f4e815f3d ./include/kernel/handlers/msg.h
- d1c9cf39d89b5b7df251448988db0b9a ./include/kernel/handlers/nameserver.h
- 75ba899de21e5a56cd03e5463b55454d ./include/kernel/kernel_task.h
- 73ac21cb5ecbadf022626ec84ae0dd88 ./include/kernel/syscalls.h
- b69ecef68e0837b6848e53f2119ddafe ./include/debug/debug.h
- 282620ffa16a43b3abdffc85c3d16f4f ./include/user/test/k2_task.h
- 85cd1fc78d9cec6519b1669f9c6f38d6 ./include/user/test/messaging.h
- 6d34f1f42ad665a8c93b034aac1bbb49 ./include/user/test/k1_task.h
- c6467f4a5c4fdeb07e5eca30b2d1d9b4 ./include/user/test/test_task.h
- 3f64a50d40c64cc5119eb287d5385a54 ./include/user/test/k2_metric.h
- bae6deab4c8decc7a32eeecd2e8f0839 ./include/user/nameserver.h
- 0387f008252d94a5e6f20532ca668d30 ./include/test/priority_queue.h
- de904d764844422a1ea910c5487bca2e ./include/test/circularbuffer.h
- 4f6384d9a9a8acb841868d1124f123e0 ./include/test/nameserver.h
- 87dda4bf560ec7f2fa9e8b78f02537dd ./include/test/task_queue.h
- 6e139e1dbc80b4e07ada400f2ddc09e6 ./include/defines.h
- 66883059f1e01b6d0873bd7f4774a8cf ./include/system.h
- d81ff69cda76f0fd9e82496dee58d35b ./include/task/priority_queue.h
- fdd0630a68e8c59069c1f0a55c4462a3 ./include/task/task.h
- 3ba7bf8bc6390be93bcd1412808e92e9 ./include/task/task_queue.h
- ca0df9fc349ef064b27bc743a5397b3c ./src/bwio/bwio.c
- 2ef398ca0f3de46f133dbe4caca186ff ./src/lib/circularbuffer.c
- 5bf365f7ee0a16385ac10653c4d36834 ./src/kernel/kernel.c
- ed8d4b37a784f21f5f1d114f29c91363 ./src/kernel/handlers/msg.c
- 08cee571550f520b5ffd8967f62a8519 ./src/kernel/handlers/nameserver.c
- 8839cf1c82d29911dd2c1c56b8888653 ./src/kernel/kernel_task.c
- ec08bb3b9a9cf555e42af8ea90075249 ./src/kernel/syscalls.c
- 610569b5215ac3725c559a58aca083a6 ./src/kernel/system.c
- 9fb137f9e6237e05346f43560c3b6b83 ./src/user/nameserver.c
- ae2b867f6cc725c06412ccebe558742e ./src/user/test/k2_metric.c
- 85266a4e7217fc48d7250b148388383b ./src/user/test/k1_task.c
- 291fe051a62af197bdcd1107be55803e ./src/user/test/test_task.c
- 46ec7a52524a95ce81644818e4e50b51 ./src/user/test/k2_task.c
- 18464892c11f608e804698039221fc35 ./src/user/test/messaging.c
- 158cc659fb5436528f9a4f64c22254e3 ./src/test/circularbuffer/circularbuffer.c
- 01ff056b122b739d0f131d926ecae363 ./src/test/test.c
- 2d9433795750abe54611e2bd58433b6d ./src/test/nameserver/nameserver.c
- 719ed077f10ea60fddf9b234b7f79420 ./src/test/task/priority_queue.c
- f43e61a5ea50e5c2c7a34aab46a9c885 ./src/test/task/task_queue.c
- 49ae81eda219a859562be10486ba854e ./src/task/priority_queue.c
- 1d5b3582fe0fdde4c80e6c773dde15a3 ./src/task/task.c
- e0a93e5f5b3d54fde762c8f94c6cbb26 ./src/task/task_queue.c