CS 452 Kernel 3

Benjamin Zhao, Kyle Verhoog

Feb 05, 2018

Contents

Operation

The ELF file is located at `/u/cs452/tftp/ARM/ktverhoo/kernel3.elf` and can be initiated using the typical `load` command.

The kernel should run the FirstUserTask and then exit printing usage statistics.

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.

Clock Server

Event Types

- `Delay(tid, ticks)` - queues a task for the given number of ticks
- `DelayUntil(tid, tick)` - queues a task until the given tick

- `Update()` - tells the clock server to increase the tick count (meant to be sent from the clock server notifier)
- `Halt()` - tells the clock server to shut-down

Implementation

The clock server is designed to be as simple as possible to minimize the chance of bugs. Under the hood it is very simple. The clock server maintains an ordered queue of task ids. The task ids are ordered based on the tick that the task is waiting for. At the head of the linked-list is the task waiting for the lowest tick. We prioritized the `pop` operation to be as fast as possible making the trade-off of having a more expensive `insert` operation.

Insertion to the clock server queue occurs in $O(n)$ where n is the number of elements in the queue.

Popping the next ready task id off of the queue is $O(1)$. Since the queue is ordered no other adjustment has to occur when popping a task id.

The clock server itself is a typical send/receive server which loops infinitely on a `Request` and handles the different events.

When the clock server receives an `Update` event it updates the tick count and checks the queue for ready elements. Currently the clock server will pop off up to `CS_PROCESS_NUM == 5` tasks per `Update`.

See `src/lib/clockserver_queue.c` `src/user/clockserver.c` for implementation details.

Clock Server Notifier

In order to notify the clock server of when an interrupt occurs, a separate task is used to handle the interrupt and send a request to the clock server. The clock server notifier calls `AwaitEvent` in an infinite loop and then `Send`s to the clock server an `Update` request.

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

Output

Raw Output (Snippet)

```
t6,d10,i1
t6,d10,i2
t7,d23,i1
t6,d10,i3
t8,d33,i1
t6,d10,i4
t7,d23,i2
t6,d10,i5
t6,d10,i6
t8,d33,i2
t7,d23,i3
t6,d10,i7
t9,d71,i1
t6,d10,i8
t6,d10,i9
t7,d23,i4
t8,d33,i3
t6,d10,i10
t6,d10,i11
t7,d23,i5
t6,d10,i12
t6,d10,i13
t8,d33,i4
t7,d23,i6
t6,d10,i14
t9,d71,i2
t6,d10,i15
t6,d10,i16
t7,d23,i7
t8,d33,i5
t6,d10,i17
t6,d10,i18
t7,d23,i8
t6,d10,i19
```

```
t8,d33,i6
t6,d10,i20
t7,d23,i9
t9,d71,i3
FINAL METRICS
=============
Idle ran for: 211 of 213 ticks
Percentage Idle: 99%
Worst Running Sesson: 0.99 tick
Best Running Sesson: 1.0 tick
```

Explanation

First, you may notice the very terse output formatting. This is because having a longer print statement actually delays tasks enough to mess with the ordering.

In our kernel, the larger the priority number, the more priority the task has. 31 is the highest priority and 0 is the lowest. The tasks are labeled `t6` through `t9`. `t6` has priority 6 (highest), `t7` has priority 5, `t8` has priority 4 and `t9` has priority 3.

To understand the ordering of the output we ran through the test case by hand. Consider the following timeline which demonstrates the ordering of events worked out by hand, assuming the amount of time it takes between a task resuming, printing and the next task delaying is less than a tick.

```
| time (ms) | event                            | output      |
| --------- | -------------------------------- | ----------- |
| 0         | t6 unblocked, delays for 100ms   |             |
| 0         | t7 unblocked, delays for 230ms   |             |
| 0         | t8 unblocked, delays for 330ms   |             |
| 0         | t9 unblocked, delays for 710ms   |             |
| 100       | t6 resumes, prints, delays 100ms | t6,d10,i1   |
| 200       | t6 resumes, prints, delays 100ms | t6,d10,i2   |
| 230       | t7 resumes, prints, delays 230ms | t7,d23,i1   |
| 300       | t6 resumes, prints, delays 100ms | t6,d10,i3   |
| 330       | t8 resumes, prints, delays 330ms | t8,d33,i1   |
| 400       | t6 resumes, prints, delays 100ms | t6,d10,i4   |
| 460       | t7 resumes, prints, delays 230ms | t7,d23,i2   |
| 500       | t6 resumes, prints, delays 100ms | t6,d10,i5   |
| 600       | t6 resumes, prints, delays 100ms | t6,d10,i6   |
| 660       | t8 resumes, prints, delays 330ms | t8,d33,i2   |
| 690       | t7 resumes, prints, delays 230ms | t7,d23,i3   |
| 700       | t6 resumes, prints, delays 100ms | t6,d10,i7   |
| 710       | t9 resumes, prints, delays 710ms | t9,d71,i1   |
| 800       | t6 resumes, prints, delays 100ms | t6,d10,i8   |
```

900	t6 resumes, prints, delays 100ms	t6,d10,i9
920	t7 resumes, prints, delays 230ms	t7,d23,i4
990	t8 resumes, prints, delays 330ms	t8,d33,i3
1000	t6 resumes, prints, delays 100ms	t6,d10,i10
1100	t6 resumes, prints, delays 100ms	t6,d10,i11
1150	t7 resumes, prints, delays 230ms	t7,d23,i5
1200	t6 resumes, prints, delays 100ms	t6,d10,i12
1300	t6 resumes, prints, delays 100ms	t6,d10,i13
1320	t8 resumes, prints, delays 330ms	t8,d33,i4
1380	t7 resumes, prints, delays 230ms	t7,d23,i6
1400	t6 resumes, prints, delays 100ms	t6,d10,i14
1420	t9 resumes, prints, delays 710ms	t9,d71,i2
1500	t6 resumes, prints, delays 100ms	t6,d10,i15
1600	t6 resumes, prints, delays 100ms	t6,d10,i16
1610	t7 resumes, prints, delays 230ms	t7,d23,i7
1650	t8 resumes, prints, delays 330ms	t8,d33,i5
1700	t6 resumes, prints, delays 100ms	t6,d10,i17
1800	t6 resumes, prints, delays 100ms	t6,d10,i18
1840	t7 resumes, prints, delays 230ms	t7,d23,i8
1900	t6 resumes, prints, delays 100ms	t6,d10,i19
1980	t8 resumes, prints, delays 330ms	t8,d33,i6
2000	t6 resumes, prints, delays 100ms	t6,d10,i20
2070	t7 resumes, prints, delays 230ms	t7,d23,i9
2130	t9 resumes, prints, delays 710ms	t9,d71,i3

With the assumption that the time between a task resuming, printing and delaying again is less than a tick, the above ordering holds. We noticed that increasing the size of the print message caused some lines to be out of order. Namely, `t7,d23,i3` and `t6,d10,i7` were out of order when we had a longer print message for each line.

As you can see the output of the kernel matches the hand example.

Source Code and Hashes

Source is located at https://git.uwaterloo.ca/bkcs452/kernel/tree/kernel3.