CS 452 Kernel 1

Benjamin Zhao, Kyle Verhoog

Jan 25, 2018

Contents

Operation

ELF location

The ELF file is located at `/u/cs452/tftp/ARM/ktverhoo/kernel1.elf` and can be initiated using the typical `load` command.

After loading the kernel should execute the sample test required for the assignment.

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

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`.

We can mask the lower bits to obtain the id:

```
Tid_id = Tid & 0xffff
```

We can shift the upper bits and mask to obtain the version:

```
Tid_ver = (Tid >> 16) & 0xffff
```

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.

Inserting a Task

Inserting a task into a priority queue is done via `pq_push(priority_q, priority, task)` which pushes the given task it to the corresponding task queue for the priority.

When a task is added to one of the task queues the corresponding indicator bit is flipped indicating that the queue has tasks.

```
int pq_push(priority_queue *pq, int priority, TaskDescriptor *t) {
  tq_push(&pq->pqs[priority], t);
  pq->state |= 1 << priority;
  return 0;
}
```

This is done in constant time.

Getting the Next Task

Using the 32-bit state integer we can read off the number of leading zeros using `__builtin_clz()`.

We then just pop off the task from the queue for the priority found using the above code. Lastly we update, if necessary, the state integer.

```
int pq_pop(priority_queue *pq, TaskDescriptor **t) {
  int p = 31 - __builtin_clz(pq->state);
  task_queue *tq = &pq->pqs[p];
  tq_pop(tq, t);
  if (tq->size == 0)
    pq->state ^= 1 << p;
  return 0;
}
```

Thus we get the next task in constant time as well.

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

The return value of the syscall is stored into the `TaskDescriptor` and is later loaded into the user state through `activate`. A function in the kernel called handle takes the argument from the task's syscall to determine which request the user task made.

Output

Raw Output

```
Created: 1
Created: 2
Tid: 3, parentTid: 0
Tid: 3, parentTid: 0
Created: 3
Tid: 4, parentTid: 0
Tid: 4, parentTid: 0
Created: 4
FirstUserTask: exiting.
Tid: 1, parentTid: 0
Tid: 2, parentTid: 0
Tid: 1, parentTid: 0
Tid: 2, parentTid: 0
```

Explanation

The first task created is `FirstUserTask` which has Tid 0 with priority 3. Then `FirstUserTask` creates an `OtherTask` with Tid 1 and priority 0. This task has lower priority than `FirstUserTask`. Both tasks are pushed onto the priority queue.

`FirstUserTask` is scheduled and prints `Created: 1`. Then it calls `create`, creating another `OtherTask` with Tid 2 and priority 0. Similarly to task 1, `FirstUserTask` is scheduled again and prints `Created: 2`. Then `FirstUserTask` creates an `OtherTask` with Tid 3 and priority 6.

Since task 3 has higher priority, it is scheduled next and will continue running until it `Exit`s, printing `Tid 3, parentTid 0` twice. After task 3 `Exit`s then `FirstUserTask` is the next scheduled task and similarly creates another higher task with Tid 4 and priority 6 again. Which does the same thing task 3 did but printing out `Tid 4, parentTid 0`.

`FirstUserTask` is then scheduled again and `Exit`s printing `FirstUserTask: exiting.`.

Now all that is in the queue are task 1 and task 2 both with priority 0. Since task 1 was pushed onto the queue before task 2, task 1 is scheduled and alternates execution with task 2. Thus printing out the last four lines:

```
Tid: 1, parentTid: 0
Tid: 2, parentTid: 0
Tid: 1, parentTid: 0
Tid: 2, parentTid: 0
```

Source Code

TODO

Hashes

TODO