CS 452 Kernel 4

Benjamin Zhao, Kyle Verhoog

Feb 16, 2018

Contents

Operation

The ELF file is located at `/u/cs452/tftp/ARM/ktverhoo/kernel4.elf` and can be initiated using the typical `load` command.

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 512KB in size. So each consequent user stack is 512KB separated in memory and therefore, there can only be 32 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.

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

```
if(td->it) {
  PUSH_STACK("r0-r12, lr");
  WRITE_SPSR(td->psr);

  SET_CPSR(SYSTEM_MODE);
  WRITE_SP(td->sp);
  POP_STACK("lr");
  asm("mov r8, sp");
  asm("add sp, sp, #56");

  SET_CPSR(KERNEL_MODE);
  asm("ldmfd r8, {r0-r12, lr}");
  asm("SUBS pc, lr, #4");
}else{
  PUSH_STACK("r0-r12, lr");
  asm("mov r8, %0"::"r"(td->ret));
  PUSH_STACK("r8");
  WRITE_SPSR(td->psr);
```

```
  SET_CPSR(SYSTEM_MODE);
  WRITE_SP(td->sp);
  POP_STACK("lr");
  asm("mov r8, sp");
  asm("add sp, sp, #56");

  SET_CPSR(KERNEL_MODE);
  asm("ldmfd r8, {r0-r12, lr}");
  POP_STACK("r0");
  REVERSE_SWI();
}
```

User to Kernel (on IRQ or SWI)

```
asm("IRQ_ENTRY:");
asm("stmfd sp, {r0-r12};");

asm("mov r9, sp;"
    "mov r10, lr;");

SET_CPSR(SYSTEM_MODE);

asm("ldmdb r9!, {r0-r7}");
asm("stmdb sp!, {r0-r7, r10}");
asm("ldmdb r9!, {r0-r4}");
asm("stmdb sp!, {r0-r4}");
PUSH_STACK("lr");

SET_CPSR(KERNEL_MODE);
POP_STACK("r0-r12");
SET_CPSR(SYSTEM_MODE);
READ_SP(td->sp);
SET_CPSR(IRQ_MODE);
READ_SPSR(td->psr);
SET_CPSR(KERNEL_MODE);
td->it = 1;

asm("mov r0, #12"); //ENUM - TR_IRQ
POP_STACK("lr");
asm("b ACTIVATE_END");

asm("KERNEL_ENTRY:");
asm("stmfd sp, {r0-r12};");

asm("mov r9, sp;"
```

```
    "mov r10, lr;");

SET_CPSR(SYSTEM_MODE);

asm("ldmdb r9!, {r0-r7}");
asm("stmdb sp!, {r0-r7, r10}");
asm("ldmdb r9!, {r0-r4}");
asm("stmdb sp!, {r0-r4}");
PUSH_STACK("lr");

SET_CPSR(KERNEL_MODE);
POP_STACK("r0-r12");
SET_CPSR(SYSTEM_MODE);
READ_SP(td->sp);
SET_CPSR(KERNEL_MODE);
READ_SPSR(td->psr);
td->it = 0;
SWI_ARG_FETCH("r0");
POP_STACK("lr");

asm("ACTIVATE_END:");
```

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.

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.

Insertion

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.

Schedule

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.

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.

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.

IRQ Context Switch

Kernel To User Task

When activating a user task, the kernel now checks the TaskDescriptor interrupt flag to determine if it is reloading a SWI or a hardware interrupted task. When reloading an IRQ, all registers are restored and `SUBS pc, lr, #4` is called.

User Task to Kernel

The user state is first copied onto the irq stack and then migrated to the user stack. The kernel state is then restored, and the SPSR is read from irq_mode. A special flag is set in the user's TaskDescriptor to note that it has hardware interrupted and should do a full restore (reload r0).

Interrupt Handling

Interrupt Handler

The Interrupt Handler supports the syscall `AwaitEvent` which user tasks can invoke to block waiting for an interrupt and be awoken from the Interrupt Matrix. The kernel will run the `event_register` command to register the task on the interrupt specified by the user task's r0 (subsequently saved to the user stack after context switch).

When a hardware interrupt occurs, the kernel enters via IRQ_mode, notes itself as a hardware interrupt and then runs `event_wake` which does a lookup in the Interrupt Matrix and wakes up all tasks blocking on the highest priority, non-empty interrupt. Any active interrupts with higher priority (which have no waiting tasks) will also be cleared to avoid multiple context switch overhead.

Interrupt initialization and cleanup are handled at the beginning and end of the kernel. Interrupt init creates sets up the Interrupt Matrix, sets the IRQ Mode stack base, and enables interrupts.

Interrupt cleanup diables all interrupts (so we don't have to reset the box after every session).

Note: In the spirit of microkernels, enabling/disabling specific interrupts may be moved to a user task instead.

```
void init_irq(interrupt_matrix *im );
void cleanup_irq();
void event_register(interrupt_matrix *im, TaskDescriptor *td);
void event_wake(interrupt_matrix *im);
```

Interrupt Matrix

The Interrupt Matrix is a data strcture which holds all tasks blocked waiting on an interrupt. The matrix is a `struct` including circular buffers for each interrupt the kernel handles (improve as we go). Currently, the Interrupt Matrix only supports storing and loading tasks for the T3CUI interrupt.

Insertion and head deletion for the task both take $O(1)$ time. However, when waking all tasks for a corresponding interrupt requires $O(n)$ time, for n being the number of tasks waiting on the interrupt (rarely large).

```
void im_init(interrupt_matrix *im);
int im_push(interrupt_matrix *im, TaskDescriptor *task, InterruptEvent ie);
TaskDescriptor *im_top(interrupt_matrix *im, InterruptEvent ie);
int im_pop(interrupt_matrix *im, InterruptEvent ie);
unsigned int im_eventsize(interrupt_matrix *im, InterruptEvent ie);
```

Stored Events

For each interrupt event, there is a corresponding flag which indicates whether or not the event has been asserted but there was no event handler. This proves useful for finding out errors in timing. It also helps us fast track an AwaitEvent handler, while ensuring it does not block forever on a disabled interrupt.

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.

IO Servers

IO servers are used to facilitate communication over UART devices via interrupts. We implemented our servers in an abstract enough manner such that we could use the same functions for both UART1 and UART2.

Note that the IO servers, for simplicity have their FIFOs disabled.

Notifiers

All notifiers for IO events are instantiated based off of `IOServerNotifier`. It is a very basic notifier which enables the interrupt right before calling `AwaitEvent` on that event.

Transmitter

The transmitter works similarly to as described in the notes. It is abstract enough to support the use cases of both UART1 and UART2. It works by creating a notifier for the `TX` interrupt event and then enters the typical server loop, blocking on receive. It accepts `PutC` and `PutStr` requests. `PutStr` allows a user to add a string to the transmit buffer 'atomically'. This is useful for maintaining bytes stay together.

CTS Enabled

When CTS is enabled as an argument to the transmitter, it spawns an `IOServerNotifier` for the modem interrupt. It also uses flags stored locally to maintain a state machine to allow proper sending to the train controller.

Receiver

The receiver is even simpler than the transmitter. It simply, like the transmitter, spawns a notifier to listen for the `RX` interrupt. It then listens for a request from the notifier or a `GetC` request. It handles these as you might expect with a receive buffer which is read from and returned to `GetC` requests.

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
- `CreateArg`: Creates another task to be put on the kernel's task schedule with a given argument.
- `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

Source Code and Hashes

Source is located at https://git.uwaterloo.ca/bkcs452/kernel/tree/kernel4.