CS 452 Assignment 1

Victor Scurtu (20678028) and Alex Danila (20516806)

January 27, 2020

Operating Notes

The executable file is found in the student environment under:

```
/u/cs452/tftp/ARM/vscurtu/k1/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/k1/kernel; go
```

The program simply executes and finishes. No additional interaction is needed after issuing the above command.

Program Description

Initialization

The program starts at main and begins by initializing kernel memory by calling kinit(). All initialization is done within kinit(). It first configures COM2 to 115200 baud, FIFO disabled. After this, it calls init_task_list and init_pqueue to initialize the task and pqueue modules as follows:

init_task_list(): Initializes the values of an array of task structures by setting each structure's t_id to its array index, and their p_id and state to TASK_INVALID. The meaning of the fields and states is described in the Tasks section. The size of this array is defined in constants.h and can be reconfigured easily. The current running task is set to -1.

init_pqueue(): Initializes a bank of pqueue 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 priority 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 tasks. This means the kernel is not a real task.

After these two initialization procedures, the kernel initializes the interrupt vector table to point to our enter kernel function for software interrupts and an unhandled exception handler for all other interrupts for the purpose of debugging.

Printing

All printing is currently done using blocking IO from the bwio library. A set of macros are defined in logging.h for various verbosity levels - debug, log, warn, error, fatal; from most to least verbose - which also print the line number and filename from which they were called. A call to fatal if enabled will cause a kernel panic. An assert is also defined, which behaves similarly to a fatal message; it prints the failed assertion and its location and then panics the kernel. Calling panic prints a message and returns to redboot using an experimentally determined hardcoded address.

Tasks

The maximum number of tasks is set to 128. It is assumed that no individual task nor the kernel will consume more than 128KiB of memory. Therefore user task stack bases are allocated every 128KiB starting at 0x0100000 growing up towards a maximum address 0x1FE0000. This is in reverse of the direction of growth for individual stacks. Allocating user stack bases like this minimizes the probability that the tasks and kernel collide if the kernel stack grows too far.

The starting and maximum addresses, as well as the size and number of tasks, are defined in constants.h and can be easily changed if necessary. As configured, the addresses allow for about 31MiB of workable memory, but the system only has enough task descriptors for about 16MiB of tasks. It should be noted that we cannot use the full 31MiB of workable memory as the kernel stack grows from the end of it. Regardless, since we only use 16MiB of memory for the task descriptors, we keep well away from the kernel stack.

User tasks are tracked using the task module, which contains a task struct and several functions for interfacing with the array of tasks. The task struct has fields for the task ID (t_id), the parent ID, (p_id), priority, state, exit code, the stack pointer, and stack base, which point to the top and bottom of the stack respectively. The array is declared in task.c to protect it from direct access. This ensures that the array cannot be corrupted by writing invalid data, since the interface functions validate data using asserts before saving.

Tasks can have either TASK_INVALID, TASK_READY, TASK_RUNNING, or TASK_ZOMBIE for state. TASK_INVALID is used to designate unallocated task descriptors. TASK_READY is for tasks that can be scheduled to run. TASK_RUNNING is for the current running task. TASK_ZOMBIE is used by the Exit syscall; it is set for tasks that have exited but have not been deallocated.

There is a series of getters and setters for the state, exit_code, and stack_pointer task fields. There is also get_task_by_id which returns a copy of the task struct for a given task ID. The t_id, p_id, priority, and stack_base fields do not have getters or setters as they should not be set after a task is created.

All tasks have their t_id set when init_task_list is called. Individual tasks have their p_id, priority, and stack_base set when allocate_task is called. Available task descriptors are found by searching through the task_list until a descriptor with the state TASK_INVALID is found. This allows for reuse of tasks descriptors after they have been destroyed. Importantly, allocate_task does not initialize the task stack. It is expected that the caller (currently only kcreate) will initialize the stack after a newly allocated task is returned to it.

Additionally, the task module also has several record keeping and validation functions. It keeps track of the currently running task in a variable which is set to -1 on initialization. This is treated as a “virtual” ID for the kernel. Setting the running task can be done by either supplying a task ID to set_running_task or calling set_task_state with an ID and the TASK_RUNNING state as arguments, which then calls set_running_task. If set_task_state is called with the current running task ID and a state other than TASK_RUNNING, the current running task is set to -1.

In order to ensure that a task is not left in the TASK_RUNNING state when a context switch occurs, set_running_task asserts that the current task is set to -1 before attempting to set it to anything else. In order to do a context switch, set_task_state must first be called on the running task to set it in a state other than TASK_RUNNING, before set_running_task can be called on another task ID.

If a task state is set to TASK_ZOMBIE, set_task_sate searches through the task array for that task’s children. Any children have their p_id set to PARENT_ZOMBIE.

For validation of parameters, __is_defined_task_state and is_valid_task check if the task state is one of the defined states, and check if the task state is not TASK_INVALID respectively. These are used in asserts to ensure logical consistency.

Lastly, a free_task function is defined for later use with the Destroy syscall. It sets the state of the task with the provided ID to TASK_INVALID so that __get_next_available_id can find it.

Scheduling

Scheduling is done using a priority queue structure, the implementation of which is outlined in the Structures section of this report. Our priority queue structure contains 8 priority levels (0-7), each represented by a FIFO queue. 0 represents the "highest" priority and vice versa. Aside from the initialization function, described in the Structures section of this report, the pqueue structure provides the functions push_task, pop_task, and peek_task in its interface. push_task pushes a task onto the back of the queue corresponding to the priority level of the task as defined inside its task structure. pop_task removes the front task from the highest priority level queue and returns it. peek_task returns the front task on the highest priority level queue, but does not remove it. Both pop_task and peek_task return -1 if all priority queues are empty. A current running task variable is stored in task.c that is accessed with the set_running_task and get_running_task functions.

Reschedules are only triggered by tasks since there is no pre-emption. A reschedule is always triggered every time a task calls a kernel primitive. For all kernel primitives so far except Exit, this causes the current running task to be pushed to the back of the priority queue and the next task to be popped from the queue. For Exit, its simply removed from the front of its priority queue.

In all cases a new task is will now be at the front of the highest priority queue (if there are 2 or more tasks). This is due to exposing a new element in the same queue, or making a new queue the highest priority non empty queue. This new task will be the next running task by calling pop_task after the previous manipulations have been done.

Kernel Primitives

The following is the list of kernel primitives required for this assignment and their implementations. Every kernel primitive directly has a syscall associated with it. The following implementations occur in a syscall handler function handle_swi and occur while in kernel.

Create: Calls kcreate with the arguments in r1 and r2 which represent the priority and function pointer of the task respectively. kcreate works by first calling get_running_task which accesses the global currently running task ID, and allocate_task which will allocate a new task structure in the task_list array. It then obtains the actual task structure using the ID returned by allocate_task and initializes a trap frame using the stack_base pointer with an offset that provides enough space to store all the main registers and the cspr. This initial trap frame is needed since enter_user will require that there be a trap frame to pop off the stack to enter the process. Using the frame, important registers r13, r14, r15, and cspr are initialized on the task's stack to be restored when the task is entered. This is the stack_base address in r13, the exit_handler in r14, which calls Exit when a task returns, the function pointer in r15, designating the entry point, and the user processor mode in cspr. The stack_pointer is then set to point to the top of the frame and the task is added to the priority queue with push_task, placing it on the appropriate internal queue for its priority level. Finally, kcreate prints the created task ID and returns the ID to the caller, handle_swi. If the priority provided was not a valid value or allocate_task ran out of task descriptors, either -1 or -2 is returned respectively.

MyTid: Calls get_running_task which accesses global currently running task id.

MyParentTid: Gets the parent from the task structure associated with the current running task obtained the same as in MyTid.

Yield: Does nothing other than reschedule the task. As described before, this moves it to the back of its current priority queue.

Exit: Does nothing but set the state in the task structure as zombie. As described in Scheduling, the task is later popped off its priority queue by default.

The handler function, handle_swi, uses a trap frame to inspect the stack of the calling task and extract the syscall type and arguments. It also uses the trap frame to set the return value of the call, if any, in r0 on the caller's stack. The Frame structure used for trap frames is discussed further in the Structures section of this report.

Context Switching

Context switching is done between user tasks and the kernel. A context switch will always happen between a user task and the kernel or vice versa, so in order to switch to a new task the kernel will always be entered first, handle the switch, then switch back into the new task.

In user mode a context switch is triggered by a call to syscall which can be done directly or indirectly through the kernel primitives. This causes a software interrupt (SWI) which then jumps to the handler we've configured in our vector table, enter_kernel. From the kernel, a context switch occurs by calling enter_user.

At the core of context switches are the functions enter_kernel and enter_user. These functions handle saving registers to the stack and restoring the new contexts registers from their stack. enter_kernel and enter_user are both assembler routines found in arm_lib.asm and function as such:

enter_kernel: Care is taken to preserve all the registers as they arrived so that this routine may be used for hardware interrupts later. We begin by saving r0 on the kernel stack so that we can use it. We then switch into system mode to access the stack pointer of the user in r13, place this in r0, then switch back to supervisor mode. From here, the main goal is to create a trap frame holding all the registers of the user at the time of interrupt onto the users stack to be restored once we need to enter_user again. We copy the link register which is the pc (r15) of the user task onto the user stack, then proceed to copy r1-r14 onto the the stack. We then pop the users original r0 that we stored on the kernels stack and place it onto the user stack. At this point, r0-r15 are stored in order on the user stack so they may be loaded at once later (by enter_user) with a single ldmia arm instruction. After the registers are saved, we save the spsr and place it onto the top of the users stack. We then pop r4-r11 and r14 from the kernel stack (r0-r3 are scratch registers). Finally, we call bx lr which will return to kernel where it last called enter_user.

enter_user: This is a simpler routine since kernel will never be interrupted, and so enter_user will always be explicitly called. Enter_user takes one argument, the stack pointer of the task to be entered. First registers r4-r11 and r14 are pushed onto the kernel stack. We then load the spsr into r1 from the user stack pointer stored in r0. Finally we switch to user mode, then load all the users registers (r0-r15) from the top of its stack using an LDM instruction. Note that the last register loaded is the pc (r15), and so after loading it we have effectively jumped back into the tasks flow of execution.

Structures

PQueue: The interface of this function is described in “Scheduling”. This structure contains an array of pqueue_nodes called “data” that’s the size of the maximum number of tasks (since no more can be scheduled). This works as a fixed size allocator, providing nodes for all priority level 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 number of priority levels we have (8). The elements in the array are pqueue_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 the priority levels.

pop_task and push_task 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_task, a node is taken from the free list and becomes the new tail of the desired prioirty level queue. When we pop_task a node is taken from the highest non empty priority level queue and becomes the new head of the free list. These two processes represent allocating and deallocating a pqueue_node respectively.

Task: An array of "task" structures which each hold information about a task is maintained. Each task structure holds the task id, parent id, its priority (which is crucial when providing a task id to pqueue which reads this value), its state (running, ready, invalid, or zombie), an exit code is currently unused, a pointer to its stack and its stack base. Currently zombie tasks are not invalided so a task structure and ID cannot be reclaimed. The task states work as follows:

TASK_READY: the task is ready to execute
TASK_RUNNING: the task is currently running
TASK_ZOMBIE: the task has exited but not been deallocated
TASK_INVALID: the task is unallocated

Frame: A structure used to cleanly access a trap frame in memory. Mainly used when creating a new task and setting up an initial trap frame in C. This structure has the same layout in memory as the trap frames that we create and destroy in enter_user and enter_kernel, so a stack pointer with a trap frame at the top can simply be cast to a "Frame" struct then all registers and the cspr can be accessed using its fields.

Known Bugs

- No currently known bugs.

Conventions and Assumptions

The user task created must be called "umain" and declared in user.h. The kernel must create the first task and the user should not be editing the kernel code. Thus, the kernel simply calls a function "umain" which must be defined by the user in user.h.

When a task calls Exit, all of its children have their parent ID set to PARENT_ZOMBIE, which is defined to be -11 in task.h.

There is also a PARENT_DEAD defined as -10 in task.h, which will become the parent ID of a task’s children once Destroy is implemented.

Currently there is no checking for collision of task stacks. It is assumed that 128KiB will be sufficient for tasks. It is further assumed that the kernel stack will not grow the ~15MiB from the end of memory into the currently configured lowest user task stack.

Output

Terminal Output

```
Created: 0
Created: 1
Created: 2
Created: 3
Task ID: 3, Parent ID: 0
Task ID: 3, Parent ID: 0
Created: 4
Task ID: 4, Parent ID: 0
Task ID: 4, Parent ID: 0
FirstUserTask: exiting
Task ID: 1, Parent ID: -11
Task ID: 2, Parent ID: -11
Task ID: 1, Parent ID: -11
Task ID: 2, Parent ID: -11
Kernel: exiting
```

Description

The first task “umain” is created with priority 2 and assigned the id 0 by the scheduler. Inside the umain function a “test_task” is created with a lower priority (3) and assigned the id 1. Since its priority is lower, when the umain task is rescheduled, its the only one on its queue which is the highest priority so will be run again. In effect, execution continues after creating the test_task. A second test task of priority 3 is created, assigned the id 2, and execution continues in umain since it once again remains the highest priority task.

Next, a test_task of higher priority then umain (1) is created and assigned the id 3. Since it is higher priority, when Create triggers a reschedule, task 3 will now be the highest priority task causing it to run. It prints its task id and parent id. When it yields, it is the only one on its priority queue so remains the first element when being pushed to the back. Thus it is chosen as the next task to run, continues its execution and prints its task id and parent id again. After this it calls Exit(), booting itself off its priority queue and making umain's priority queue (2) the highest one again.

umain continues execution, printing "FirstUserTask: exiting" then calling Exit, removing it from the priority level 2 queue and making the priority level 3 queue the highest priority non empty queue. Thus, the second task created (id = 1) is now the first element on the highest priority non empty queue. It prints its task ID and parent ID. Since the parent has exited, it is now a zombie task, so the id -11 is returned which signifies the parent is a zombie. If it were invalid -10 would be returned, but we currently do not invalidate zombie tasks.

When task 1 yields, it moves to the back of the priority level 3 queue, making task 2 the front of the queue now. Task 2 begins execution and prints its task ID and parent ID. Once again its parent umain is a zombie so the id is -11. When it yields, task 1 is at the front of the queue now, and continues execution. It prints its task ID and -11 again, then exits leaving task 2 as the only task left on the queue. Task 2 continues execution and prints its task ID and -11 again and exits.

At this point there are no more tasks left on any queue. This causes the kernel to exit.