CS 452 Assignment 1

Authors

Sebastian Gaweda (20143503) and Shashwat Pratap (20638036)

Make and Run

make - run make in /u6/spratap/cs452/cs452-rtos-aka-trains/k1/src
run - in gtk terminal run load -g 10.15.167.5 ARM/spratap/k1.elf; go

Context Switching Failures

Infinitely Looping FirstTask

There was a bug in the context switch which was caused by forgetting to store the program counter when saving the user program state. The program counter would be set the first time the task was created, and on subsequent schedules of the task this would cause the task to start over, creating new low priority tasks until running out of memory. It a particularly nasty bug to track down.

Corrupted Stack Pointers

For a long time there appeared to be a bug in our code which caused the user space stack pointer to begin pointing to the wrong address. Eventually, careful inspection of the ordering of our context switching code revealed that there were multiple errors in our logic storing and restoring values during context switch.

Current State

Crisis averted.

Kernel 1 Structure

Using C++

The decision to make C++ was made very early on. We wanted to be able to leverage templating to get around some of the pitfalls we both experienced during A0 when creating data structures that use different types. We also both agreed that classes would give us an abstraction that makes it easier to organize our code.

Kernel Class

The main component of our kernel is the Kernel class. This class runs the main loop of kernel which configures the environment and then loops through scheduling tasks, switching to them, and then handling any requests that the task may produce. It also stores the ready queue and an array of TaskDescriptors which also serves the function of setting aside stack space for our tasks.

TaskDescriptor Class

TaskDescriptors store important task information like the tid and parent tid. They also store the stack space for any task.

ContextSwitch

We originally tried to write our context switch using *gcc* extended asm, and were even able to get 1 switch to occur, but it made it much harder to debug with gdb because the correspondence between our context switch code and generated code wasn't always obvious. With time running out we pulled the plug and decided to implement the context switch fully in assembly.

Data-structures

We created a templated RingBuffer which can be templated with a type and the number of objects to be stored. It exposes push and pop operations and is currently used in our code as a queue.

We created a templated PriorityQueue which can be templated with a type, a number of rows, and a number of columns. The rows correspond to priorities and define how many RingBuffers the PriorityQueue has. The columns correspond to how many elements can be in each RingBuffer. The PriorityQueue's pop function selects the first item from the highest priority non-empty buffer.

Program Output

Output

1. Created: 2
2. Created: 3
3. TestTask: My tid is 4 and my parent's tid is 1

4. TestTask: My tid is 4 and my parent's tid is 1
5. Created: 4
6. TestTask: My tid is 5 and my parent's tid is 1
7. TestTask: My tid is 5 and my parent's tid is 1
8. Created: 5
9. FirstUserTask: exiting
10. TestTask: My tid is 2 and my parent's tid is 1
11. TestTask: My tid is 3 and my parent's tid is 1
12. TestTask: My tid is 2 and my parent's tid is 1
13. TestTask: My tid is 3 and my parent's tid is 1

Explanation

Our first task, with tid 1, is created during kernel config and begins creating 4 tasks. The first 2 have lower priority and the last 2 have higher priority. This is what happens during that process:

1. Create call 1 triggers a software interrupt. The kernel creates a new task with tid 2 and schedules tid 1 because it has higher priority. tid 1 prints (1.)
2. Create call 2 triggers a software interrupt. The kernel creates a new task with tid 3 and schedules tid 1 because it has higher priority. tid 1 prints (2.)
3. Create call 3 triggers a software interrupt. The kernel creates a new task with tid 4 and schedules tid 4 because it has higher priority. tid 4 prints (3.)
4. tid 4 Yields and the kernel schedules tid 4 because it has the highest priority. tid 4 prints (4.)
5. tid 4 Exits and the kernel schedules tid 1 because it has the highest priority. tid 1 prints (5.)
6. Create call 4 triggers a software interrupt. The kernel creates a new task with tid 5 and schedules tid 5 because it has higher priority. tid 5 prints (6.)
7. tid 5 Yields and the kernel schedules tid 5 because it has the highest priority. tid 5 prints (7.)
8. tid 5 Exits and the kernel schedules tid 1 because it has the highest priority. tid 1 prints (8.)
9. tid 1 Exits and the kernel schedules tid 2 because it has the same priority as tid 3 but was in the ready queue first. tid 2 prints (10.)
10. tid 2 Yields and the kernel schedules tid 3. tid 3 prints (11.)

11. tid 3 Yields and the kernel schedules tid 2. tid 2 prints (12.)
12. tid 2 Exits and the kernel schedules tid 3. tid 3 prints (13.)
13. tid 3 Exits