K1 Writeup

1 Summary

This is a writeup for Kernel Assignment 1 of CS452, where we set up a kernel, basic context switching functionality and some primitive system calls. In this writeup, we will describe the structure of the kernel, talk about the choices we made for critical system parameters and limitations, and also talk about the kernel's runtime behaviour.

2 Kernel Structure

2.1 Kernel Class

The main component of our kernel is the `Kernel` class:

```
class Kernel
{
public:
    enum HandlerCode
    {
        NONE = 0,
        CREATE = 1,
        MY_TID = 2,
        MY_PARENT_ID = 3,
        YIELD = 4,
        EXIT = 5
    };
    Kernel();
    ~Kernel();
    void schedule_next_task();
    void activate();
    void handle();

private:
    int p_id_counter = 0;                       // keeps track of new task creation id
    int active_task = 0;                        // keeps track of the active_task id
    InterruptFrame *active_request = nullptr; // a storage that saves the active user request
    Scheduler scheduler;                        // scheduler doesn't hold the actual task descriptor
    TaskDescriptor* tasks[USER_TASK_LIMIT];    // points to the starting location of taskDescriptors

    SlabAllocator<TaskDescriptor, int, int, int, void (*)()> task_allocator =
        SlabAllocator<TaskDescriptor, int, int, int, void (*)()>(
            (char*)USER_TASK_START_ADDRESS, USER_TASK_LIMIT);

    void allocate_new_task(int parent_id, int priority, void (*pc)());
    void queue_task();

protected:
    void check_tasks(int task_id); // check the state of a particular task descriptor
};
```

This class wraps the kernel functionality and data structures, so it's a useful reference point for talking about structure and functionality. Most of the class contents are straightforward or self-explanatory, so we will only cover the components that hold nontrivial complexity.

Scheduling: To schedule the order of user tasks, the kernel defers to the `Scheduler`, with `schedule_next_task()` also making use of the class functionality:

```
#define NUM_PRIORITIES 3

const static int NO_TASKS = -1;

class Scheduler
{
    public:
        Scheduler();
        int get_next();
        void add_task(int priority, int task_id);

    private:
        RingBuffer<int> ready_queue[NUM_PRIORITIES];
};
```

The scheduler itself is very simple, being little more than a wrapper around 3 stack-allocated ring buffers (the ring buffers themselves are a template class with a fixed size of 512 slots; 512 was chosen because we reasoned this would be large enough for any use case). Each buffer stores task IDs and corresponds to a different priority level – high, medium and low – where each queue can be added to on request, and `get_next()` will loop through queues in order to determine which queue should be popped from when called.

Task Storage: The kernel keeps track of a SlabAllocator structure, which keeps track of slab allocations. It is a templated class with variadic arguments, which allows it to construct arbitrary classes in slabs, and it internally uses a ring buffer to keep track of free memory locations. For user task descriptors, slab allocation is set to start at the hardcoded address of 0x10000000, which we determined to be a safe address for slab allocations (will not collide with important memory addresses).

```
template <typename T, typename... Args>
class SlabAllocator
{
public:
    SlabAllocator(char* starting_location, int total_slabs);
    ~SlabAllocator();
    T* get(Args... arguments);
    void del(T* target);            // when you done with the memory please push it back
    int get_remaining_size();

private:
    int size;
    int T_size;
    RingBuffer<char*> slabs; // N
};
```

The slab allocator maintains basic free and delete functions, allowing it to perform similar functionality to that of a heap.

Task descriptors store the task ID, task parent ID, the stack of the task that they represent and some other useful status information.

```
class TaskDescriptor
{
public:
    TaskDescriptor();
    TaskDescriptor(int id, int parent_id, int priority, void (*pc)());
    int task_id;
    int parent_id; // id = -1 means no parent
    int priority;
    int prepared_response;
    bool alive;
    bool initialized;
    void (*pc)();              // program counter, typically only used as a reference value
    char *sp;                  // stack pointer
    char *kernel_stack[4096]; // approximately 4 kbytes per stack

    void show_info(); // used for debug
    bool is_alive();
    bool kill();
};
```

Activate/Handle: These functions are called in our `kmain` function, which schedules tasks and calls these two functions in an infinite loop. `activate()` initializes user tasks if they have not been initialized and context switches into them, while `handle()` responds to system calls using the kernel-defined `HandlerCodes`.

2.2 Context Switching

Most of our context switching procedure is relatively standard/inflexible, so I'll just jot down some key features here:

- As a safety precaution, we save/load all registers when switching. Hopefully this doesn't affect performance too greatly.
- When the context switch returns to the kernel, it passes the kernel a pointer to an `InterruptFrame` struct, which is essentially a format of the 32 main registers, allowing easy access.
- The first time a user task is switched into is different from subsequent switchings, as registers do not need to be loaded the first time.
- The x0 register is used liberally to pass around arguments and return values.

I will also note that our context switch is still slightly unfinished – we don't make use of saving/restoring the `SPSR_EL0` register when switching to/from user-land. At present, omitting this has no ill-effects on our program, but we are aware of the shortcoming and are looking to add it in subsequent assignments.

3 Kernel Output

Here is a sample of what typical output from our kernel looks like:

```
init kernel
finished kernel init, started scheduling user tasks
entered into user task 0
completed kernel cycle
created: task 1
completed kernel cycle
created: task 2
completed kernel cycle
completed kernel cycle
completed kernel cycle
my task id: 3; my parent id: 0
completed kernel cycle
my task id: 3; my parent id: 0
completed kernel cycle
created: task 3
completed kernel cycle
completed kernel cycle
completed kernel cycle
my task id: 4; my parent id: 0
completed kernel cycle
my task id: 4; my parent id: 0
completed kernel cycle
created: task 4
exiting task 0
completed kernel cycle
completed kernel cycle
completed kernel cycle
completed kernel cycle
completed kernel cycle
my task id: 1; my parent id: 0
completed kernel cycle
my task id: 2; my parent id: 0
completed kernel cycle
my task id: 1; my parent id: 0
completed kernel cycle
my task id: 2; my parent id: 0
completed kernel cycle
no tasks available...
no tasks available...
no tasks available...
no tasks available...
no tasks available...
```

Here's what each of the different lines mean:

- `init kernel`: The `kmain` function has been called.
- `finished kernel init, started scheduling user tasks`: The `Kernel` class has been created.
- `Entered into user task 0`: The first user task, which spawns the other user tasks as per the K1 assignment description.
- `completed kernel cycle`: One loop of `kmain` has finished.
- `created: task N`: Task N has been created. Note that this line could be displayed long after the actual creation of Task N in the kernel (To a point where the created task already exited). This is due to higher priority task always takes over Task 0 if created, and will run to completion before returning to Task 0.
- `my task id: X; my parent id: Y`: The sub-task with task ID X, whose parent ID is Y, has been scheduled and activated by the kernel. Note that the tasks with IDs 3 and 4 run before 1 and 2, indicating that scheduling priorities are working as expected.
- `exiting task 0`: The user task with ID 0, i.e. the first user task, has finished creating the four subtasks and is exiting.
- `no tasks available....`: The scheduler has no tasks remaining, prompting the kernel to display the `no tasks available....` message and block for 3,000,000 cycles. If left running, this message will continue to be printed to the screen.

4 Acknowledgements

Special thanks to Mike Krinkin for his excellent ARMv8 tutorial, available here. Without it, writing a stable context switch would likely have been much more difficult.