K2 Writeup

1 Summary

This is a writeup for Kernel Assignment 2 of CS452, where we have added message passing functionality, a name server and an implementation of a rock/paper/scissors game to our kernel, alongside some performance measurements. 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 and performance measurements.

2 Kernel Structure

2.1 Kernel Class

2.1.1 K1 Contributions

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`.

2.1.2 K2 Contributions

The main architecture contributions of K2 are task state and message passing functionality. To support this, task descriptors have been supplemented with some new functions and data structures:

```
class TaskDescriptor {
public:
    enum TaskState { ERROR = 0, ACTIVE = 1, READY = 2, ZOMBIE = 3, SEND_BLOCK = 4, ... };
    TaskDescriptor(int id, int parent_id, int priority, void (*pc)());
    // message related api
    void queue_message(int from, char* msg, int message_length); // queue_up a message
    bool have_message();
    int fill_message(Message msg, int* from, char* msg_container, int msglen);
    int fill_response(int from, char* msg, int msglen);
    Message pop_inbox();
    // state modifying api
    InterruptFrame* to_active();
    void to_ready(int system_response, Scheduler* scheduler);
    bool kill();
    void to_send_block(char* reply, int replylen);
    void to_receive_block(int* from, char* msg, int msglen);
    void to_reply_block();
    void to_reply_block(char* reply, int replylen);

    // state checking api
    bool is_active();
    bool is_ready();
    bool is_zombie();
    bool is_send_block();
    bool is_receive_block();
    bool is_reply_block();

    const int task_id;
    const int parent_id; // id = -1 means no parent

    friend class Kernel;
protected:
    void show_info();

private:
    TaskState state;
    int priority;
    int system_call_result;
    bool initialized;
    void (*pc)();
    MessageReceiver response;
    RingBuffer<Message> inbox;
    char* sp;
    char* kernel_stack[USER_STACK_SIZE];
};
```

These different functions can place tasks into different states, which may or may not remove them from scheduling or flag them as having passed into different states of the message passing loop. Tasks also now maintain a message inbox, which allows them to receive messages even if they aren't actively waiting for them.

The message passing functions themselves, then, are responsible for putting tasks into different states, as well as performing memcpy on the messages to transfer them to the right spots. Here's the `handle_send()` function, as an example.

```
void Kernel::handle_send() {
    int rid = active_request->x1;
    if (tasks[rid] == nullptr) {
        // communicating a non existing task
        tasks[active_task]->to_ready(NO_SUCH_TASK, &scheduler);
    } else {
        char* msg = (char*)active_request->x2;
        int msglen = active_request->x3;
        char* reply = (char*)active_request->x4;
        int replylen = active_request->x5;
        if (tasks[rid]->is_receive_block()) {
            tasks[rid]->fill_response(active_task, msg, msglen);
            tasks[rid]->to_ready(msglen, &scheduler);
            tasks[active_task]->to_reply_block(reply, replylen);
        } else {
            // reader is not ready to read we just push it to its inbox
            tasks[rid]->queue_message(active_task, msg, msglen);
            tasks[active_task]->to_send_block(reply, replylen);
        }
    }
}
```

Beyond this, we also create two servers using these message passing functions: the name server and the Rock/Paper/Scissors server. Both servers wait for send requests in an infinite loop, and process decisions only when their receive calls are fulfilled.

- The name server handles name registration and name lookup requests through the `WhoIs()` and `RegisterAs()` system calls. It uses a templated hashmap to keep track of names and task IDs, where names are mandated to be at most 16 characters in length (longer names are truncated). Duplicate names in `RegisterAs()` overwrite the the name server's mapping, while unregistered names in `WhoIs()` cause it to return a pre-specified error code.
- The Rock/Paper/Scissors server allows special clients to sign up for and play in Rock/Paper/Scissors matches against each other by sending special messages to the server. It mediates interactions between clients, keeps track of matches and results and even returns some output showing the results. This will be further discussed in the Kernel Output section.

The templated hashmap used by the name server uses a generic FNV hash function with separate chaining to resolve collisions, and is courtesy of the Embedded Template Library for C++.

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.

2.3 RPS Server/Clients

The RPS Server, like the name server, handles sends/receives from RPS clients, coordinates match logic and keeps track of matches (storing matches in a large, linearly-searched array).

Clients loop through the following logic loop:

1. Ask for the task ID of the RPS server, and send it a signup request.
2. Query the current system time to make a "random" play decision, and send it to the server.
3. Repeat the previous step $3 + \text{randint}(0, 2)$ times, then send a quit message.
4. After sending a quit message, or after receiving a quit message from the server, either destroy self (75% chance) or rejoin the match pool (25% chance).

The results of this loop are sent to the shell, and the output is discussed below.

3 Kernel Output (K2)

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

```
init kernel
finished kernel init, started scheduling user tasks
creating name server
creating rps server
creating rps client
creating rps client
creating rps client
creating rps client
creating rps client
exiting task 0
My tid is: 3 and I am ready to play!
My move is: 0
Task 3 played ROCK
My tid is: 4 and I am ready to play!
My move is: 1
Task 4 played PAPER
The winner is: 4!
Print any key to continue...

My tid is: 5 and I am ready to play!
My move is: 1
Task 5 played PAPER
My tid is: 6 and I am ready to play!
My move is: 1
Task 6 played PAPER
It's a draw!
Print any key to continue...

My tid is: 7 and I am ready to play!
My move is: 2
Task 7 played SCISSORS
My move is: 2
Task 4 played SCISSORS
My move is: 2
Task 3 played SCISSORS
It's a draw!
Print any key to continue...

[some games are played]

My move is: 1
Task 3 played PAPER
My move is: 3
Task 4 quits!
Print any key to continue...

My move is: 3
Task 6 quits!
My move is: 1
Task 5 played PAPER
```

```
Print any key to continue...

:(
:(
Task 3 is rejoining!
:(
:(
My tid is: 3 and I am ready to play!
My move is: 0
Task 3 played ROCK
The winner is: 3!
Print any key to continue...

[some more games are played]

My move is: 2
Task 7 played SCISSORS
My move is: 3
Task 3 quits!
Print any key to continue...

:(
:(
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.

· `creating <task>`: A task is being created.

· `exiting task 0`: The user task with ID 0, i.e. the first user task, has finished creating subtasks and is exiting.

· `completed kernel cycle`: One loop of `kmain` has finished.

· `My tid is: <N> and I am ready to play!`: The RPS client with task ID N has been created and has signed up for RPS.

· `My move is: <move>`: The task has just played `<move>`, which is a number from 0 to 2, inclusive, representing ROCK, PAPER or SCISSORS.

· `Task <N> played <move>`: The server has received the information about the move that was played, and reports it.

· `The winner is: <N>!`: Task N has won a match!

· `It's a draw!`: The match played by two of the tasks was a draw!

· `Print any key to continue...`: The previous match has just concluded, and the program is waiting for user input before showing the next match.

· `Task <N> quits!`: Task N has just sent a quit message.

· `:(`: A task has just received a quit message from the server.

· `Task <N> is rejoining!`: A task has just announced that they are rejoining the game.

· `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.

Note also that there are an odd number of tasks (3 to 7) and that initially, task 7 has no partner. However, when task 3 quits and rejoins, it matches up with task 7, finding an available partner instead of starting a new match and waiting for someone else.

4 Performance Measurements/Discussion

To perform our performance measurements, we created two size-templated sender and receiver functions, which attempt to pass messages to each other 150,000 times, tracking the total time taken and dividing the result by 150,000 to obtain a single performance measurement. The size templates are used to shuffle between the three different message sizes of 4, 64 and 256 bits, with senders and receivers being created in different orders within the code. The code is manually re-run 8 times to test the 4 different cache configurations, as well as testing with or without compiler optimization. Our fastest result for send + receive, 41 μs, appears with receiver-first, icache-enabled, full optimization, with the smallest message size of 4 bytes.

As for where this time is being spent, the biggest culprit is easily the context switch itself. Even in the best case, which is receiver first, we need to switch:

- From the receiver to the kernel (on calling Receive)
- From the kernel to the sender
- From the sender to the kernel (on calling Send)
- From the kernel to the receiver
- From the receiver to the kernel (on calling Reply)
- From the kernel back to the receiver

and all of these context switches require saving/loading registers as well as calling the `svc` hardware instruction, which adds up quickly. There is also additional cost accumulated by if/else handling (branching), copying messages around and calling the scheduler, so all in all, there is quite a bit of computation associated with sending and receiving that would comprise the 41 μs.

The raw performance measurement numbers for our kernel can be found in `k2/performance.txt`.

5 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.
- John Wellbelove and other contributors for their work on the Embedded Template Library (ETL) which we have shamelessly lifted directly for use in our kernel thanks to the wealth of data structures it provides.
- Marco Paland for his work on reimplementing `printf`, a function that is incredibly useful for debugging purposes, and requiring only a simple `putchar()` implementation to function properly.