Lab Tutorial 1

Course Project Part 1

Jean-Christophe Petkovich
j2petkov@uwaterloo.ca

University of Waterloo

January 15, 2014

Concepts

Memory Management

Process States

Scheduling

Initialization

Concepts in Lab Part 1

- Memory management systems
- Process states/transitions
- Context Switching
- Priority based scheduling
- Operating System (OS) initialization

Memory Management

You will be implementing a simple memory management system.

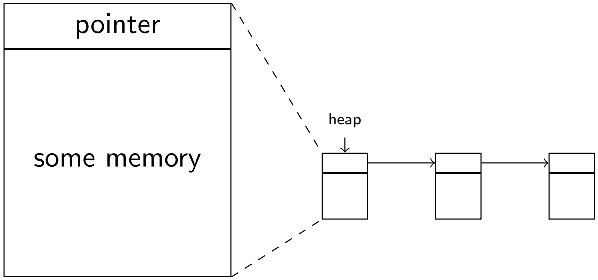
- Memory allocation will primarily be used for allocating blocks of memory to hold messages.
- It will be much easier in the long run to implement a generic memory allocation system.

Your allocation Application Programming Interface (API) will be analogous to malloc/free, it will have two methods:

- One for asking the OS for free memory blocks
- One for telling the OS you're finished with a block, so it can be returned to the heap

Memory Model

The simplest implementation of a heap is a linked list of memory blocks[1]. Here is what your datastructure might look like if you use linked lists.



How do I turn a memory block into a useful thing? Casting! If the current block's pointer is NULL, you're out of blocks!

[1] But there are many possible implementations.

Some notes...

A memory block like the previous one might look like this:

```
typedef struct mem_blk {
        uint32_t *next_blk;
}
```

- Note the lack of a mention of the memory region.
- Size and number of memory blocks should be configured with global constants, not by the struct.
- The list of free blocks is created during the initialization phase.

Request Memory Block Pseudo Code

```
int k_request_memory_block() {
        atomic(on);
        while (no memory block is available) {
                put PCB on blocked_resource_q;
                set process state to BLOCKED_ON_RESOURCE;
                release_processor();
        }
        int mem_blk = next free block;
        update the heap;
        atomic(off);
        return mem_blk;
}
```

Hint: Create your own generic utilities for handling linked lists and queues, you will need them for other things.

Release Memory Block Pseudo Code

```
int k_release_memory_block(void* memory_block) {
        atomic(on);
        if (memory block pointer is not valid)
                return ERROR_CODE;
        put memory_block into heap;
        if (blocked on resource q not empty) {
                handle_process_ready(pop(blocked resource q));
        }
        atomic(off);
        return SUCCESS_CODE;
}
```

Hint: Create your own generic utilities for handling linked lists and queues, you will need them for other things.

Process Control Blocks

Each process will have a Process Control Block (PCB), this stores information about the process's status including things such as the:

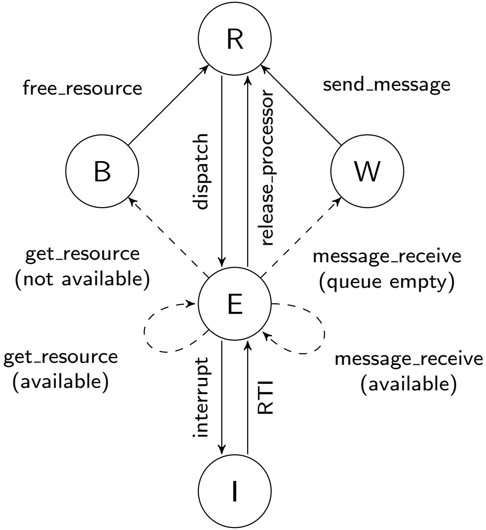
- Process's state
- ID
- Priority
- Program Counter (PC)
- CPU Registers
- Stack Pointer (SP)

State is what we're going to deal with next.

Process State Transition Diagram

R = ready, B = blocked on resource, W = waiting for message, E = executing, I = interrupted.

⟶ = forced, - - ➤= voluntary



Process Switching

Process switching:

1. Select next process to execute using scheduler
2. Invoke context switch to new process

Context switching[2]:

1. Save context of currently executing process
2. Change the process's state back to ready
3. Update current_process to new process
4. Set state of new process to executing
5. Restore context of current_process
6. Execute current_process

[2] You will get (have gotten?) *intentionally insufficient* sample code for context switching.

Scheduling

Your OS must have a scheduling algorithm that meets the following requirements:

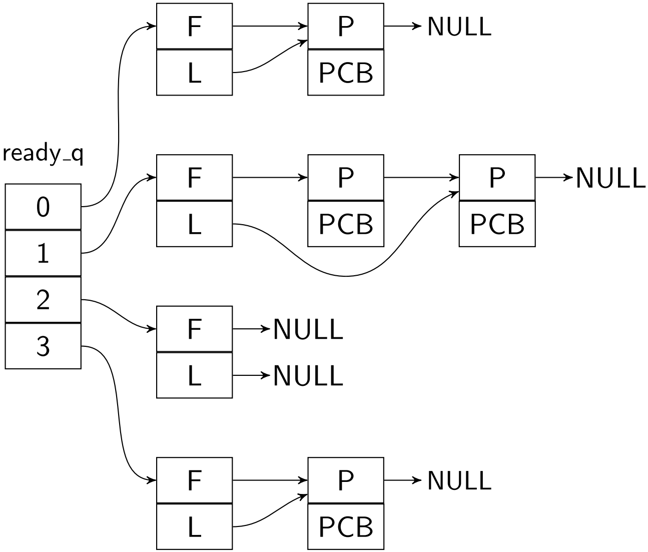
- Fixed, priority-based scheduling
- Each process has assigned priority
  - Highest priority process executes first
  - First-come-serve for processes of same priority

Scheduling Procedure

Procedure:

1. process_switch invokes scheduler
2. Scheduler selects highest priority ready process
3. context_switch(new_proc) lets the selected process execute

Scheduling Ready Queues



Note: 0 is traditionally the highest priority.

Scheduling: Null Process

What does the CPU do when it has nothing to do (i.e., all the ready queue(s) are empty)?

Scheduling: Null Process

What does the CPU do when it has nothing to do (i.e., all the ready queue(s) are empty)? NULL. . .

Scheduling: Null Process

What does the CPU do when it has nothing to do (i.e., all the ready queue(s) are empty)? NULL...

- CPU has to do something, unfortunately it cannot do nothing
  - Simple solution: NULL process
  - The NULL process should have the lowest priority possible (What will happen if you give it the highest?)
  - Example NULL process:

```c
void null_process() {
        while (1) {
                release_processor();
        }
}
```

Scheduling: release_processor()

1. Set current_process to state ready
2. rpq_enqueue(current_process) put current_process in ready queues
3. process_switch invokes scheduler and context-switches to the new process

Initialization

When the OS starts up, what initialization needs to happen?

- Initialize all hardware, including:
  - Serial port(s) and timer(s)
  - Memory mapping (memory allocation for memory blocks and stacks. . .)
  - Interrupts (hardware and software: vector table & traps)
- Create all kernel data structures
  - PCBs (status=ready), queues. . .
  - Place PCBs into respective queues

Initialization: Initialization Table

How does RTX know which processes to create?

Initialization: Initialization Table

How does RTX know which processes to create? You'll have a pre-defined initialization table.

Initialization: Initialization Table

How does RTX know which processes to create? You'll have a pre-defined initialization table.

- The table is an array of records
- Each record contains specifications of its process and any additional data structures required (unless you want to allocate those dynamically).

Example record contents:

Process ID
Priority
Initial SP
Initial PC

Questions

Do you have any?