Lab Tutorial 2

Course Project Part 2

Ahmed Elbagoury
ahmed.elbagoury@uwaterloo.ca

University of Waterloo

February 10, 2016

Concepts

Inter-Process Communication

Interrupt Handling

Timing Services

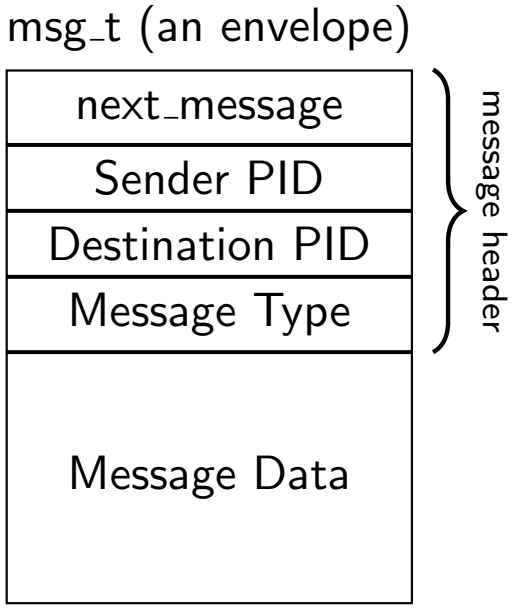
Interprocess Communication (IPC)

Requirements:

- Message-based, asynchronous
- Messages are carried in "envelopes"

Procedure:

- Process allocates memory for envelope
- Process writes data into envelope
- Process invokes message API
  send_message(proc_id, envelope)
- Other process invokes message API
  msg_t *env = receive_message()
  and blocks if no message is available



IPC: Process Message Queues

Where do we refer to messages that haven't been read yet?

IPC: Process Message Queues

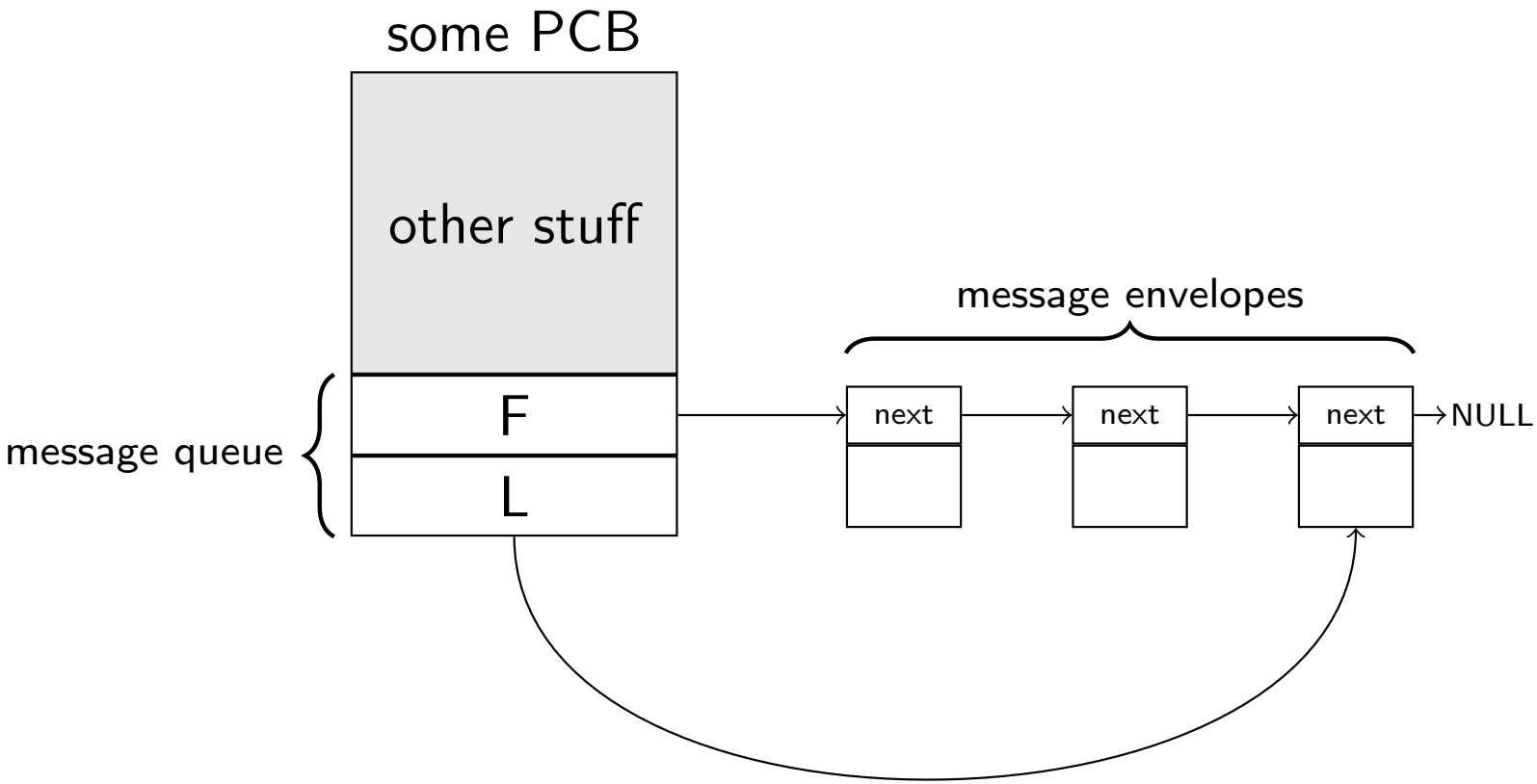
Where do we refer to messages that haven't been read yet?

We put them in a queue, tied to the PCB.

IPC: Process Message Queues

Where do we refer to messages that haven't been read yet?

We put them in a queue, tied to the PCB.



IPC: send_message()

```
void send_message(uint32 receiving_pid, msg_t *env) {
        atomic(on);
        set sender_procid, destination_procid;
        pcb_t *receiving_proc = get_pcb_from_pid(receiving_pid);
        enqueue env onto the msg_queue of receiving_proc;
        if (receiving_proc->state is BLOCKED_ON_RECEIVE) {
                set receiving_proc state to ready;
                rpq_enqueue(receiving_proc);
        }
        atomic(off);
}
```

Do you think send_message would ever block?

IPC: send_message()

```
void send_message(uint32 receiving_pid, msg_t *env) {
        atomic(on);
        set sender_procid, destination_procid;
        pcb_t *receiving_proc = get_pcb_from_pid(receiving_pid);
        enqueue env onto the msg_queue of receiving_proc;
        if (receiving_proc->state is BLOCKED_ON_RECEIVE) {
                set receiving_proc state to ready;
                rpq_enqueue(receiving_proc);
        }
        atomic(off);
}
```

Do you think send_message would ever block? Nope, never!

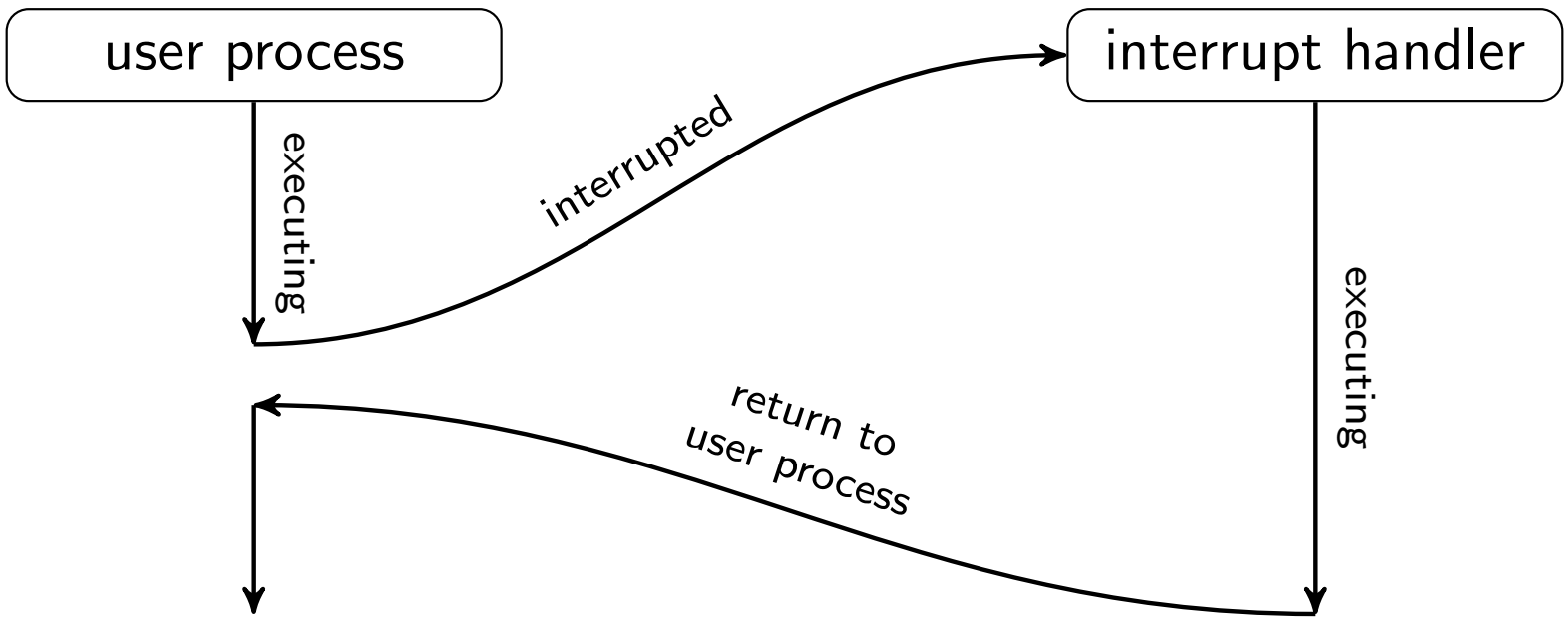
IPC: receive_message()

```
msg_t * receive_message() {
        atomic(on);
        while (current_process msg_queue is empty) {
                set current_process state to BLOCKED_ON_RECEIVE;
                release_processor();
        }
        msg_t *env = dequeue current_process msg queue;
        atomic(off);
        return env;
}
```

Note: this version blocks, you will need a non-blocking one as well

Interrupt Handling

- Interrupts are hardware messages that need immediate action
- Interrupts invoke pre-registered procedures that "interrupt" currently executing code.



Interrupt Handler: Design

Requirements:

- Interrupts must be handled by processes called i-processes
- i-processes must be OS processes
  (i.e., they can receive messages, use APIs)

Design:

- i-processes are scheduled by the interrupt handler
  - are always ready to run, but never in the ready queue
- i-processes can never block when invoking a kernel primitive
  - Primitives which can block need non-blocking alternatives
- Each i-process has a PCB just like other processes

Interrupt Handler: Example

Here is some pseudo C-code for a generic interrupt handler routine.

```
__asm void Timer0_IRQHandler {
        save the context of the current_process;
        switch the current_process with timer_i_process;
        load the timer_i_process context;
        call the timer_i_process C function;
        invoke the scheduler to pick next to run process;
        restore the context of the newly picked process;
}
```

Interrupt Handler: Example

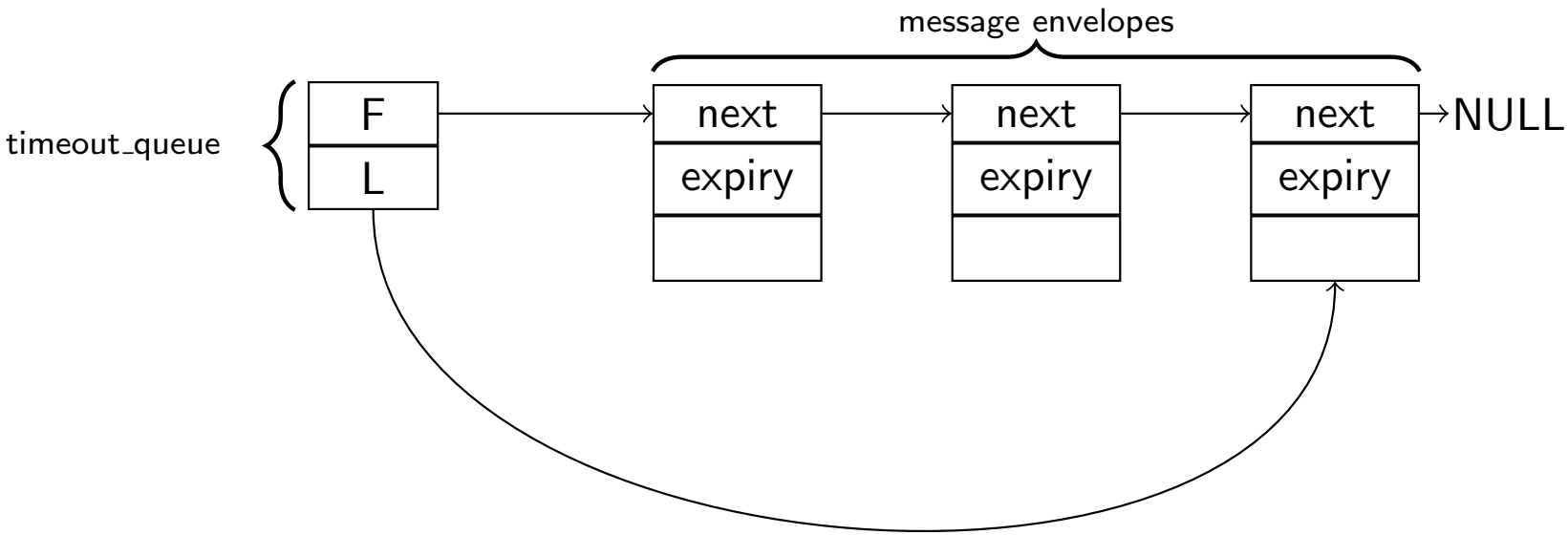
Here is some pseudo C-code for a generic interrupt handler routine.

```
__asm void Timer0_IRQHandler {
        save the context of the current_process;
        switch the current_process with timer_i_process;
        load the timer_i_process context;
        call the timer_i_process C function;
        invoke the scheduler to pick next to run process;
        restore the context of the newly picked process;
}
```

Note: in-case you haven't noticed, this will be written in assembly

Timing Service: Design

- Timing services are fundamental to RTX
- The timer_i_process receives messages to deliver after a delay (i.e., delayed_send(PID, env, delay) API), which is non-blocking!
- After the delay expires, the i-process forwards the message
- The timer_i_process maintains requests in sorted list:



Timing Service: I-Process

At each clock tick (i.e., interrupt), the timer_i_process:

- Increments current_time
- Calls receive_message repeatedly to retrieve new requests (non-blocking)
- If there are new requests, it adds them to the queue (maintaining sorted order)
- Checks if any timing requests have expired
  - If yes, send the message to the destination

Timing Service: I-Process Example

Here is some pseudo code for the timer_i_process.

```
void timer_i_process () {
        // get pending requests
        while (pending messages to i-process) {
                insert envelope into the timeout queue;
        }
        while (first message in queue timeout expired) {
                msg_t *env = dequeue(timeout_queue);
                int target_pid = env->destination_pid;
                //forward msg to destination
                send_message(target_pid, env);
        }
}
```

Questions

Do you have any questions?