UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING

SE350 RTX LAB 2

P2-A Interprocess Communications (IPC)

P2-B Timing Service

Irene Yiqing Huang

Electrical and Computer Engineering Department

P2-B Timing Service

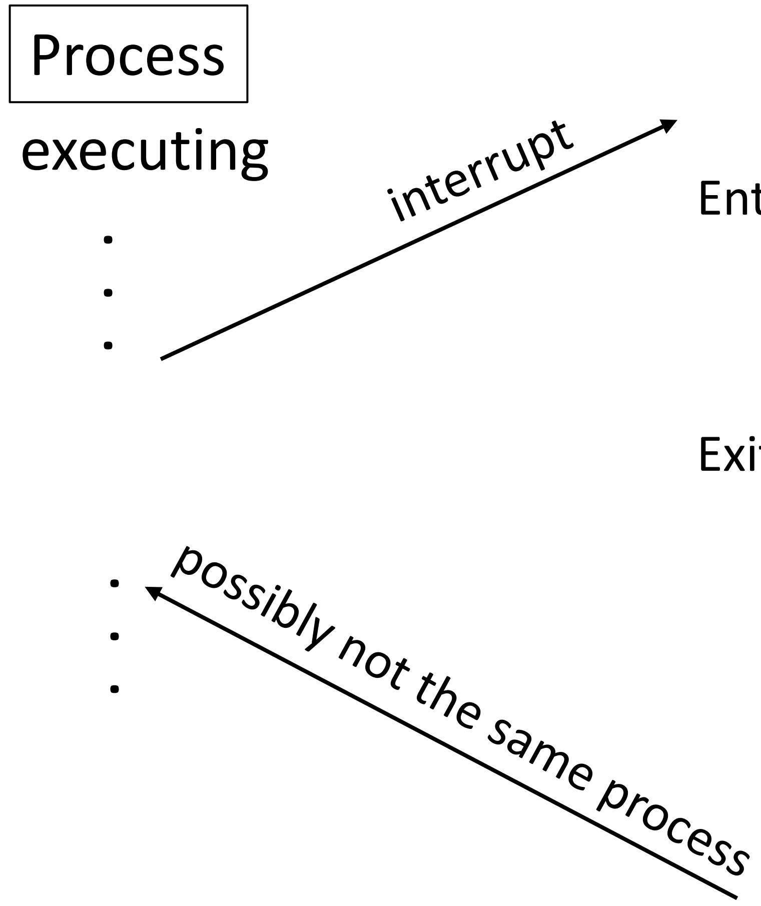
Interrupt Handling

- In real-time OS, interrupt handling must be fast
  - short latency to respond to interrupt
  - fast processing by interrupt handler
- Interrupts may cause a change in state for some blocked process
  - e.g. a process blocked for external event to occur will have its state changed to ready and be placed on the ready process queue when the event occurs

Interrupt Handling Issues

- Possible interpretation of an interrupt:
  - an interrupt is a *hardware message* usually requiring a short latency and quick service
- Design issues:
  - does the interrupt handling code run as part of kernel, within a process (if yes, which?)
  - are interrupt handlers themselves interruptible?
  - if OS is preemptive, need to deal with the possibility that an interrupt results in a higher priority process becoming ready

Interrupt Handling Sequence



Interrupt Handler

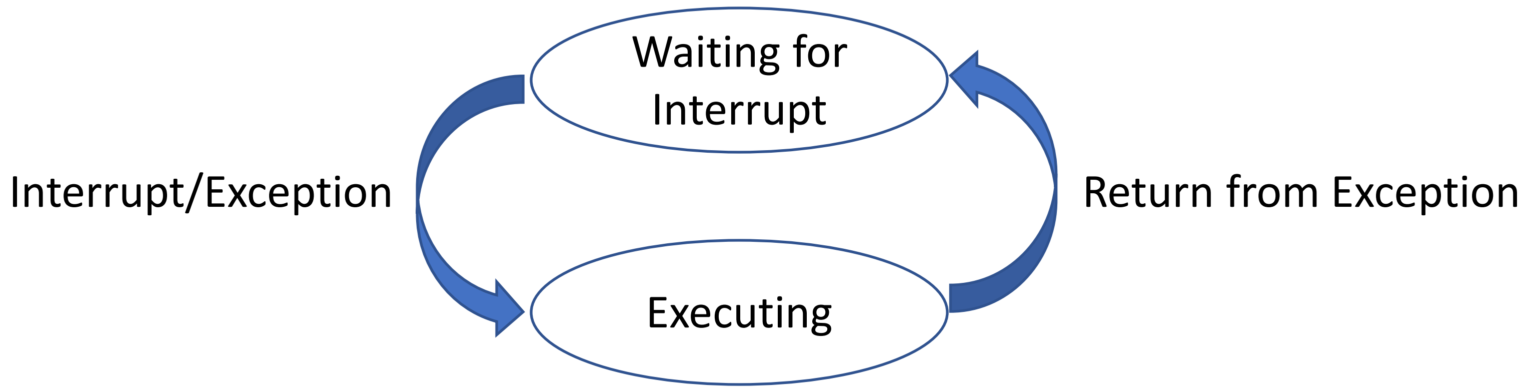
```
Entry: save context of current process
       service the device
       update any state change to a process
       goto Exit
Exit:  check ready Q for a ready process
           of higher priority than current
        If found:
           put current process on ready Q
           deque selected new process
           restore new process's context to
                match pre-interrupt state
        endif
        return from interrupt handler
```

Interrupt Handler Design

- Interrupt handler must interact with OS processes
- An i-process gets the CPU from an interrupt handling sequence, not through the scheduler.
- It never blocks if it invokes a kernel primitive
- The interrupt (exception) handling routine starts the appropriate i-process.
- Conceptually: i-process has the highest priority, is scheduled by interrupt.
- No nested interrupt handling

I-process

- State diagram for an i-process:



- A PCB is associated with each i-process with a special state = i-process (permanently)
- It is always ready to run, but not on any ready Q

I-process: Constraints

- An i-process can invoke kernel primitives:
  - however, an i-process is not allowed to block!
- Primitives which can block a process **must** be modified to ensure that i-process does not block!
  - e.g. the synchronous receive message primitive
    - return null if the invoking process is an i-process and there is no message waiting
  - similarly for other primitives

I-process: Example

```
__asm  void TIMER0_IRQHandler {
        atomic(on)
        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 ;
        atomic(off)
        return
}
```

- Embedded assembly programming
- Context_Switching_IRQ project in the starter code on github can be modified to become a Timer i-process (add interrupt off/on et. al..).

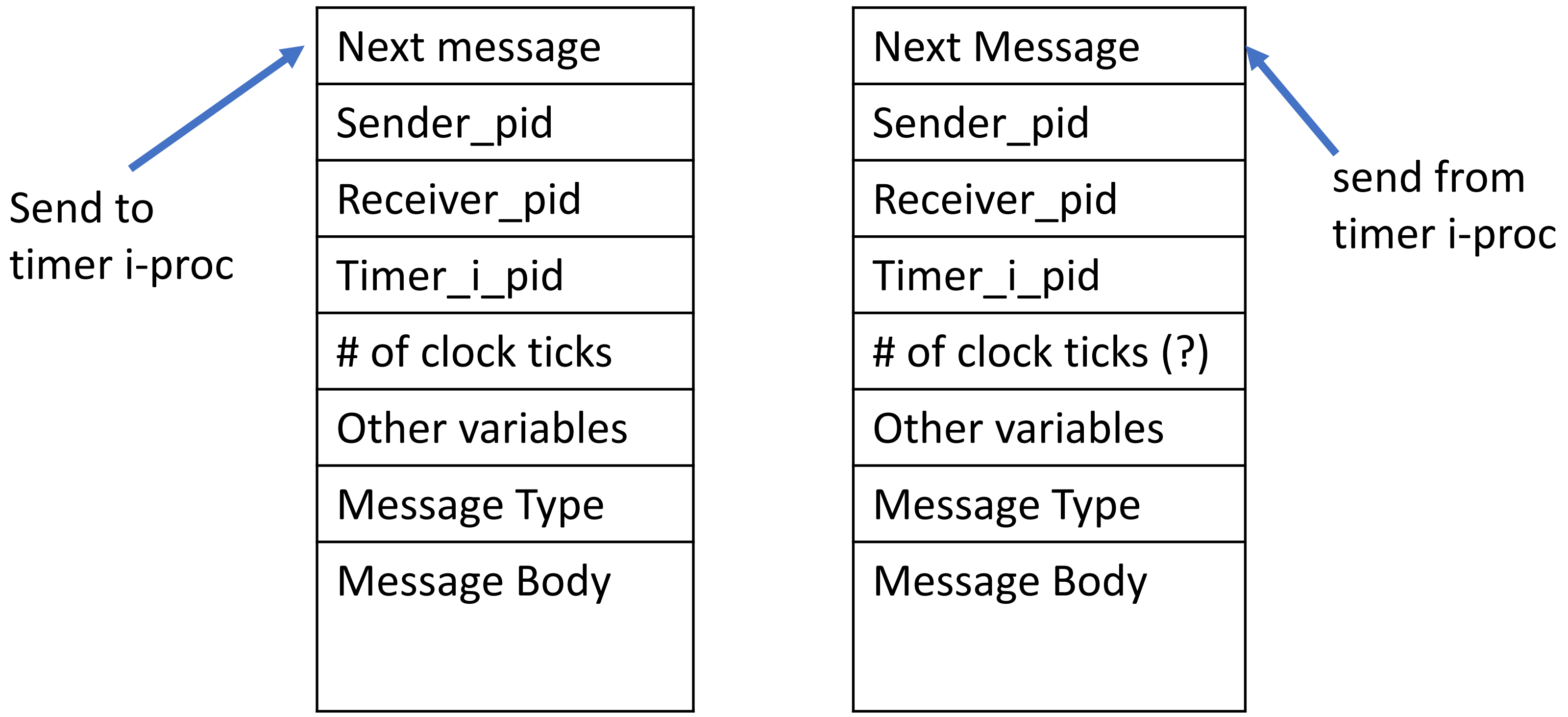
Timing Services: Design

- Fundamental to Real-time OS
- Two parts: interface/protocol design, internal design
- Interface/protocol design
    - basic service only (timeout), no cancellation
    - service request, expiry notification: by messages
- Internal design
    - timing service implemented by *i-process*
    - service request: a user process sends a request message to the timing i-process
    - timeout notification: the i-process sends message back

Timing Services: Request

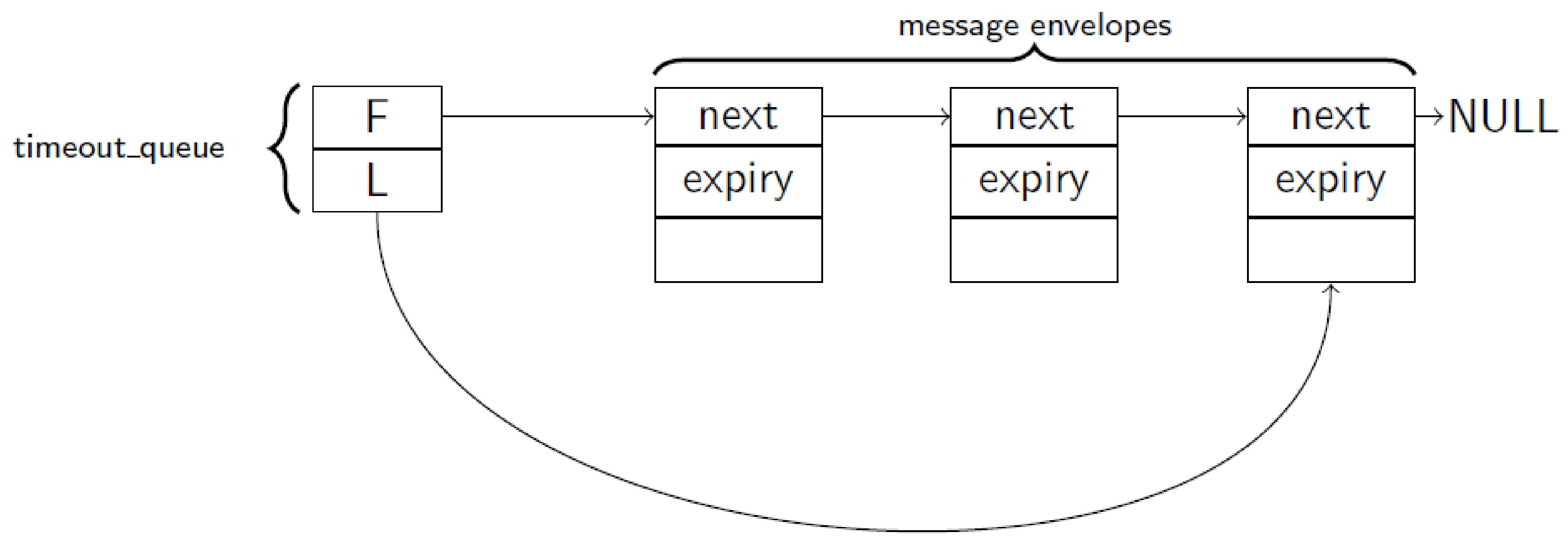
```
int delayed_send(int recv_pid, void *msg_env, int delay)
```

- The timer i-process receives messages to deliver with a time-out value (i.e. the dealy).



Timing Services: Request

- After the delay expires, the timer i-process forwards the message to the receiver
- The timer i-process holds requests in a sorted list.



Timing Services: Expiry Time

- To reduce CPU overhead, expiry time can be replaced by the # of clock ticks after the expiry of the predecessor in the list
- Example: queue timeout list {25, 30, 0, 10}
  - one timeout for 25 clock ticks
  - two timeouts for 55 clock ticks
  - one timeout for 65 clock ticks

Timer 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

Timer I-process

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

Design for Preemption

- On return from send() or any primitives that may make another process ready
  - Check if the highest priority ready process’s priority is greater than the current process priority
- What the current process is i-process?
- Possible solution
  - Let priority of i-process be the highest
  - On return from interrupt, check whether the priority of the interrupted process still the highest
  - If not, context switch to the higher priority ready process

RTX Initialization

- What operations need to be carried out at start-up?
- Initialize all hardware, incl.
  - Board system Initialization
  - Interrupts (hardware and software: vector table & traps)
  - Timer(s) ~~and Serial port(s)~~
- Create all kernel data structures
  - Memory management kernel data structure
  - Process-control kernel data structure: PCB, kernel stacks
- Create PCBs of all processes
  - allocate stacks
  - privilege level setting using CONTROL register
  - Exception stack frame creation for new processes

Git Submission

- Tag your commit with “p2-submit”

References

1. Dasiewicz, Paul, A non-preemptive RTX Design Documentation
2. LPC17xx User’s Manual
3. ARM Compilation Tools Version 5.0 Developer Guide
4. Software Interface Standard for Arm Cortex-based Microcontrollers, CMSIS Version 5.7.0

Acknowledgement

Slides modified from

- P. Dasiewicz
- J.C. Petkovich

UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING

Thank you!

Electrical and Computer Engineering Department