# SE350 – Project Overview

## Third Part

**Fall 2012**

# Outline

1. Processor Management
2. Scheduling
3. Initialization
4. Memory Management
5. Inter Process Communication
6. Interrupts Handling
7. Timing Services

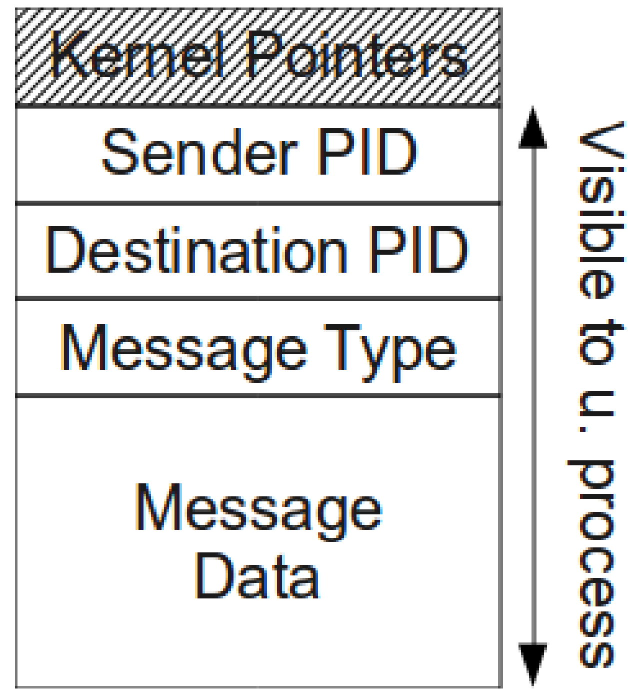
# 5. Inter Process Communication (IPC)

**Requirements:**

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

**Procedure:**

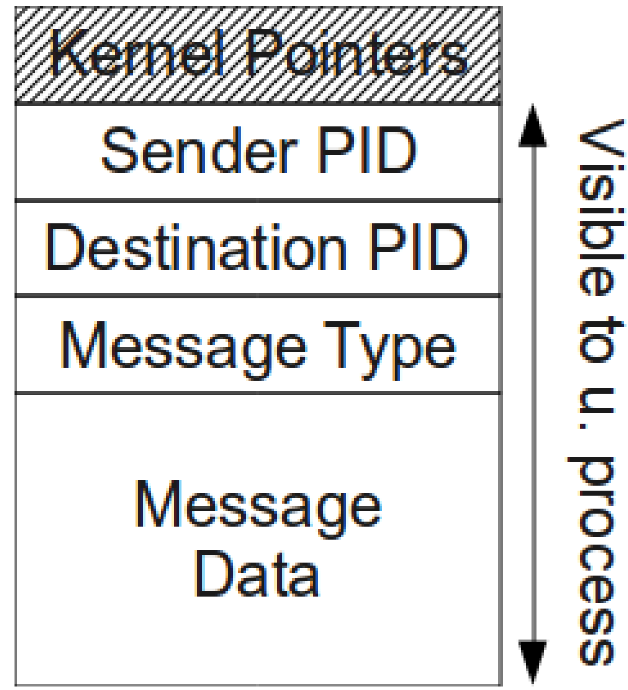
- **Process writes into envelope**
- **Process invokes API function** **`send_message(proc_id, envelope)`**
- **Other process invokes API function** **`env = receive_message()`** **and <u>blocks</u> if no message is available**



# 5. IPC: Message Envelopes

Message envelopes are managed by kernel:

- Create appropriate number of envelopes during init.
- Process allocates envelope using <u>public</u> API function `request_memory_block()`
- Process deallocates envelope using <u>public</u> API function `release_memory_block(env)` when not longer needed



# 5. IPC: Exchanging Messages

Requirement: messaged-based, asynchronous IPC

Design decisions:

- Non-blocking send
- Blocking receive

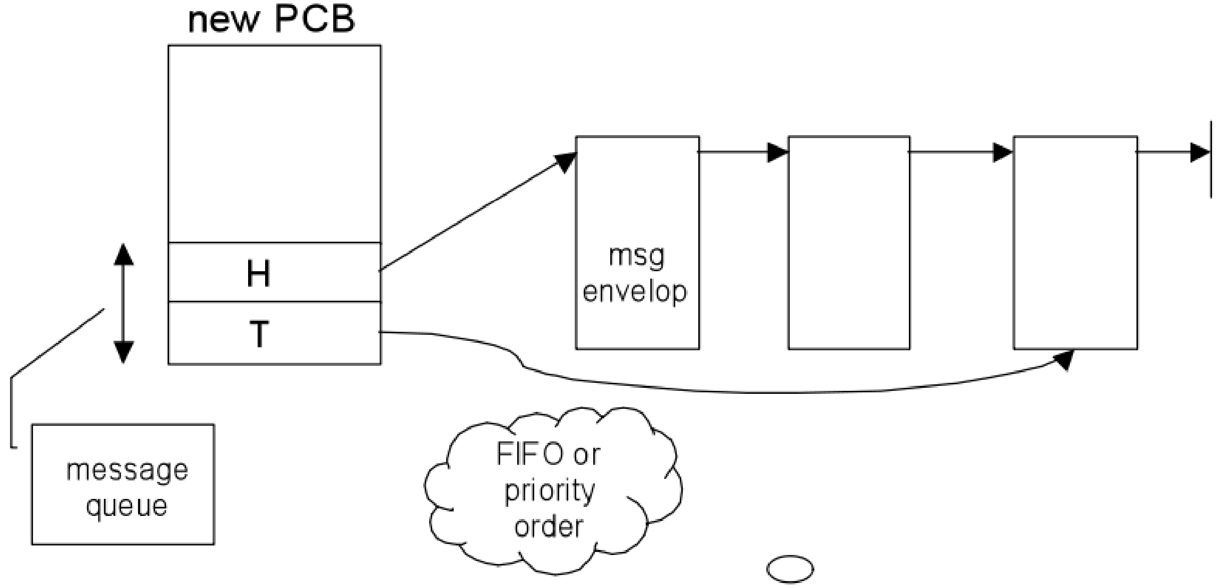
Issues:

- How do we handle multiple messages sent to a process?
- How does kernel track such messages?

# 5. IPC: Waiting Messages for Process

**Idea: have each process have a queue of waiting messages**

- **Extend PCB as follows**



# 5. IPC: send_message(pid, env)

```
send_message( target_pid, env) : {
   atomic(on);
   set sender_procid, destination_procid…
   …fields in env
   target_proc -> convert target_pid …
   …to process obj/PCB ref
   enqueue env onto the msg_queue of target_proc

   if ( target_proc.state is blocked_on_receive) {
   set target_proc state to ready
   rpq_enqueue( target_proc );
   }
   atomic(off);
}
```

- Send never blocks!
- Send performs check if target_pid is blocked, if yes, it is unblocked and put into the ready state

# 5. IPC: receive_message()

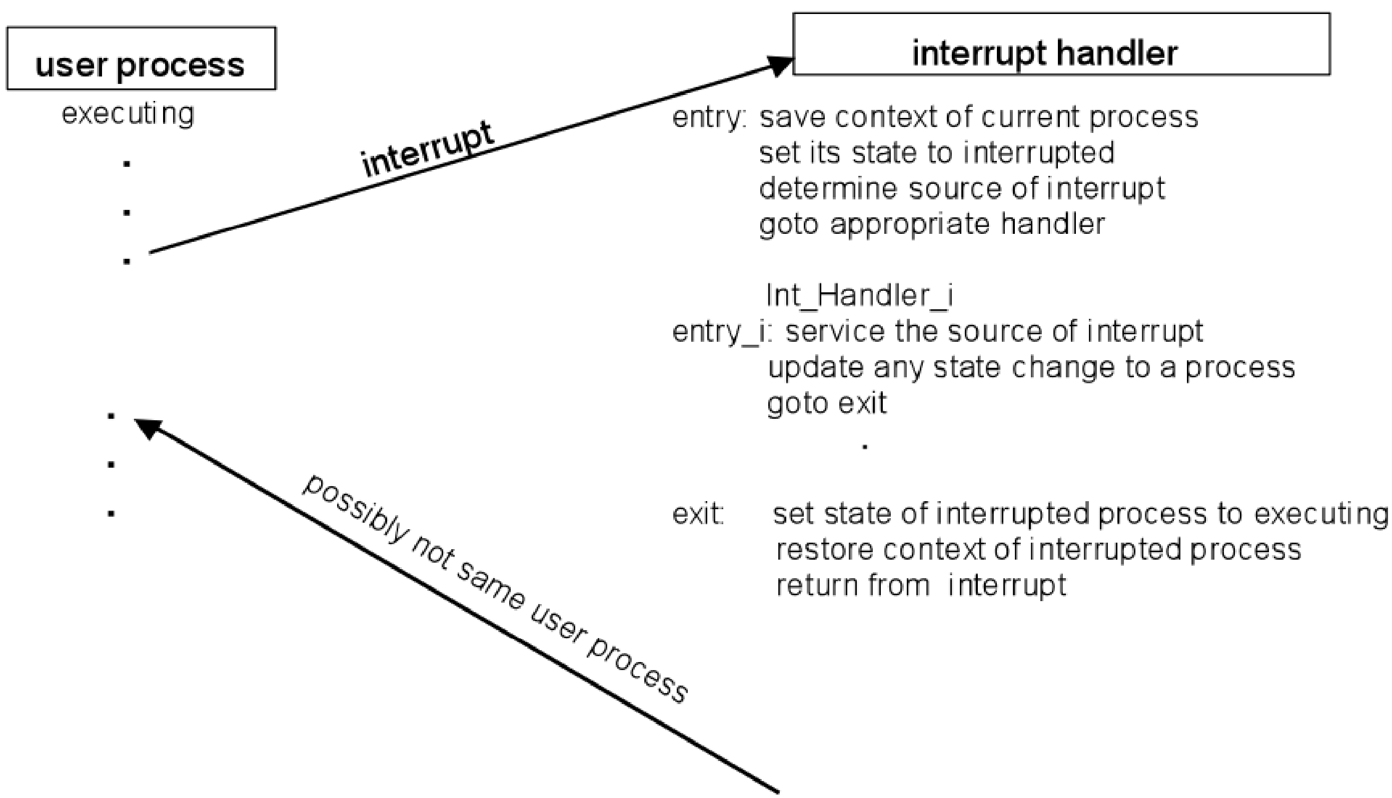
```
*env receive_message() {
   atomic(on);
   while ( current_process’s msg_queue is empty) {
   set state of current_process to blocked_on_receive
   process_switch( );

   *** return here when this process executes again
   }

   env -> dequeued envelope from the process’ message queue
   atomic(off);
   return env;
}
```

- When no messages available, process will block!
- Issue: Where should the blocked processes go?

# 6. Interrupt Handling & Atomicity

Interrupt = HW message, usually requiring short latency and quick service

- Interrupts invoke pre-registered procedures that “interrupt” currently executing code



# 6. Interrupt Handler: Design

Requirement: Interrupt handler must interact with RTX.

Design: i-process

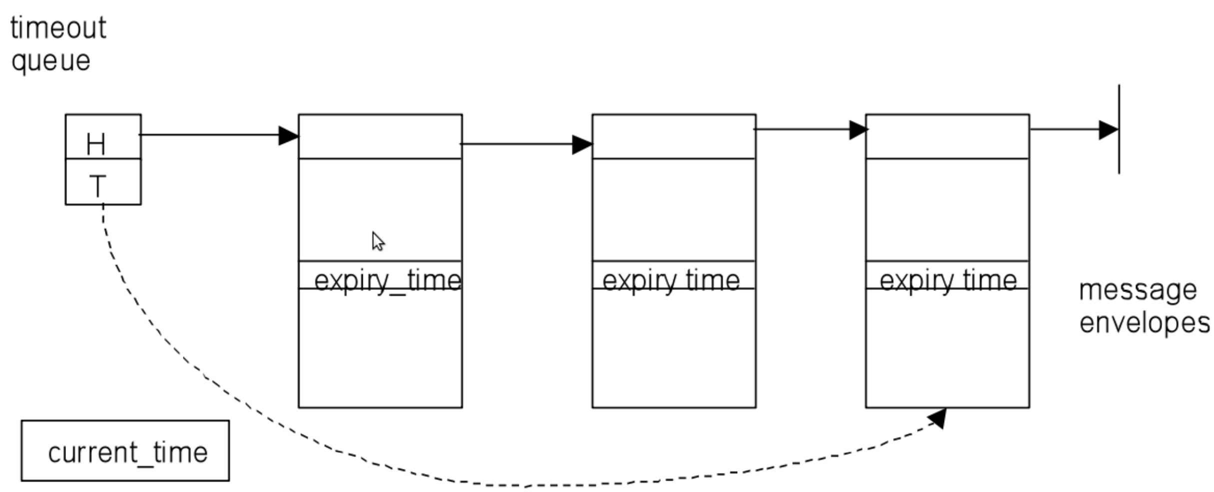
- i-process is scheduled by interrupt handler
  - Is always ready to run, but not in ready queue
- i-process never blocks when invoking a kernel primitive
  - Primitives which can block the i-process need to be adapted (e.g., IPC)
- The interrupt handler starts the appropriate i-process
- Each i-process has a PCB

# 6. Interrupt Handler: Example

```
interrupt_handler {

    set the state of current process to interrupted
    save_proc = current_process

    select interrupt source
    A:
            context_switch (i_proc_A)
            break
    …

    Z:
            context_switch (i_proc_Z);
            break
    end select

    //code to save context of interrupt handler (i_process)
    current_process = save_proc;
    context_switch (save_proc);
    //perform a return from exception sequence
    //this restarts the original process before i_handler
}
```

# 7. Timing Service: Design

- Timing service is fundamental in RTX
- Timing service i-process receives message with service request (i.e., delayed_send(PID, env, delay) API), which is non blocking!!!
- After expiration time the timing i-process sends original message to the destination
- Timeout service maintains requests in <u>sorted</u> list:



# 7. Timing Service: i-process

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

- Increments current_time
- Invokes receive() to see new requests (non-blocking!)
- If any new requests, it adds them to the list
- Checks if any timing requests have been expired
  - If yes, send the message to the destination.

# 7. Timing I-Process Example

```
timeout_i_process:
{
  env = receive(); //to get pending requests
  while (env is not null)
  {
    //code to insert the envelope into the timeout_list

    env = receive(); //see if any more requests
  }

  if (timeout_list is not empty)
  {
      while (head_of_timeout_list.expiry_time <= cur_time)
      {
        env = timeout_list.dequeue();
        target_pid = env.destination_pid;
        send( target_pid, env ); //forward msg to destination
      }
  }
}
```