



# SE350 RTX LAB 2

P2-A Interprocess Communications (IPC)

P2-B Timing Service

Irene Yiqing Huang


Electrical and Computer Engineering Department

# Reading the Lab Manual


	Section	Topics	How to Read
	1.2	Summary of RTX Requirements	Review
P2-A	2.5	Interprocess Communications	Study
P2-B	2.6	Timing Service	Study
	3.2.1	The Timer I-Process	Study
	3.4	Process ID Assignment	Skim
	4	RTX initialization	Review
	5.1.2	RTX P2 Requirements	Study
	5.2	Demo Procedure	Review
	5.3	Third-party testing framework	Review
	6.3 – 6.5	FAQ – IPC and Interrupts	Skim
	9.4	C and Assembly Programming	Study
	9.7	Timer Programming	Skim

# P2 Highlights

- Message Passing API
  - Message-based interprocess communication (IPC)
  - New process state of **BLK\_MSG**
- Timing Service
  - Timer Interrupt Handling
  - Interrupt Process
  - New process state of **IPROC**



P2-A



P2-B

# P2 User API

- Message Passing

```
/* non-blocking send */  
int send_message(int process_id, void *message_envelope)  
  
/* blocking receive */  
void *receive_message(int *sender_id)
```

P2-A

This is an output parameter!

- Timing Service

```
int delayed_send(int process_id, \  
                 void *message_envelope, \  
                 int delay)
```

P2-B

In milliseconds

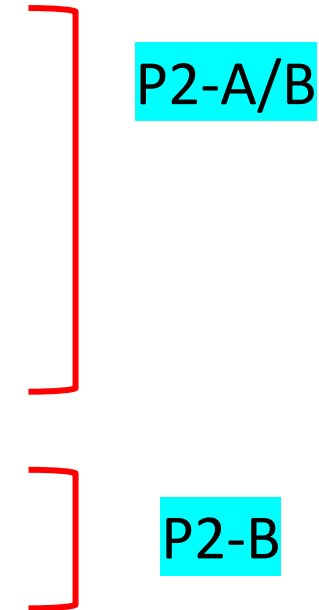
# P2 Assumptions

- Processes are non-malicious
- All processes are known
  - created at start-up
  - know each other

```
/* common.h */
#define PID_NULL      0
#define PID_P1        1
#define PID_P2        2
#define PID_P3        3
#define PID_P4        4
#define PID_P5        5
#define PID_P6        6
#define PID_A         7
#define PID_B         8
#define PID_C         9
#define PID_SET_PRIO  10
#define PID_CLOCK     11
#define PID_KCD       12
#define PID_CRT       13
#define PID_TIMER_IPROC 14
#define PID_UART_IPROC 15
```

# P2 Processes

- System Processes
  - Null process with PID = 0
- User Processes
  - Six test processes with PIDs = 1,2, ..., 6
  - User level processes, only calls the user APIs
- Interrupt Processes
  - Timer i-process, pid = 14



- What processor mode should each process be in?
- What privilege level should each process be at?

# P2 Processes

PID	PID Macro	Process	Mode	Privilege Level
14	PID_TIMER_IPROC	Timer I Process	Handler	Privileged
0	PID_NULL	Null Process	User	Unprivileged
1-6	PID_P[1-6]	Test Processes	User	Unprivileged

# P2-A IPC

Interprocess Communications



# IPC Overview

- Requirement specification:
  - message-based, asynchronous IPC
  - messages carried in shared message blocks (msg envelopes)
- Each process writes a message into a message envelope
- A process invokes `send_message(pid, msg_envelope)`
- Issues:
  - what is the format of the message envelope (i.e. envelope memory block)?
  - where do these envelope memory blocks come from?

# IPC: Message Envelopes

- Memory blocks are managed by the kernel
  - a process allocates a memory block and turns it into a message envelope by **casting**.
  - a process deallocates an envelope when it is no longer needed (current owner of the envelope!)
  - a process owns a message envelope that it receives or allocates (until it is sent)

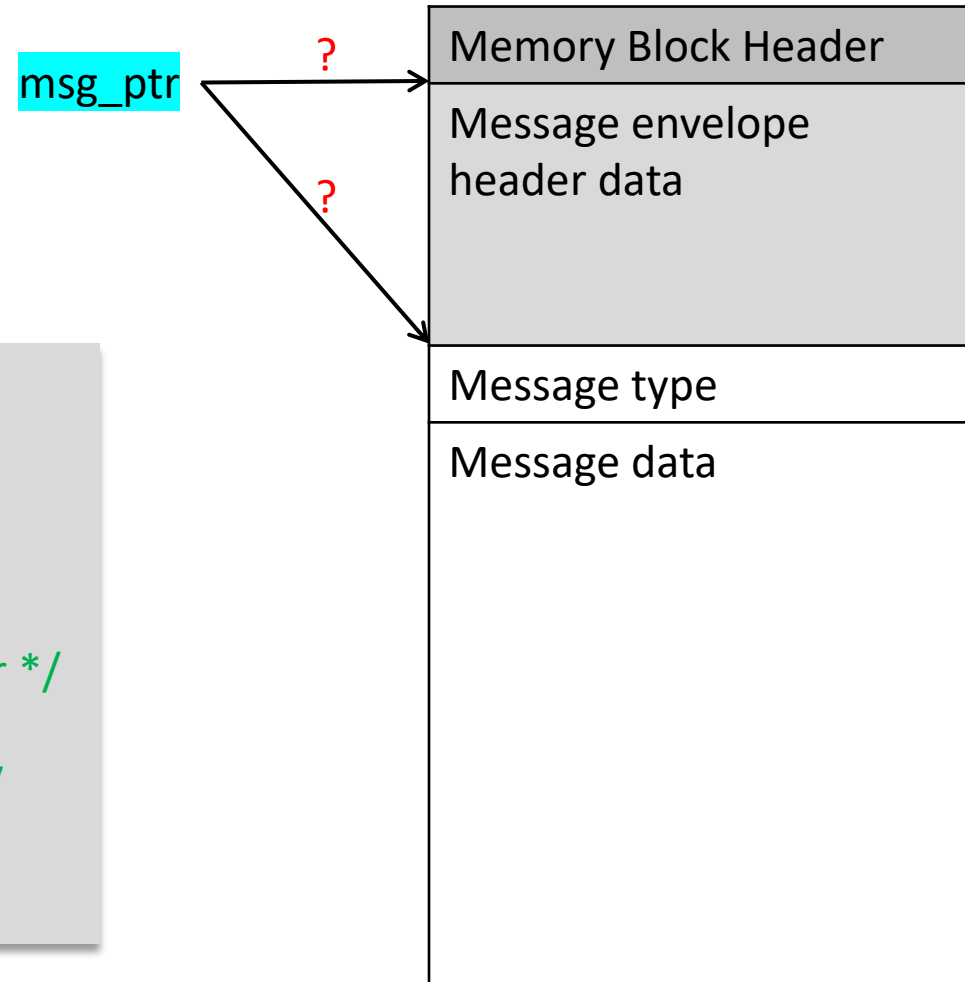
**Note the ownership requirement of message envelopes!**

# IPC: Message Envelope Format

## Design issues:

- Kernel updates the header
- Can user access the header?

```
struct msgbuf {  
#ifdef K_MSG_ENV  
    void *mp_next; /* ptr to the next message */  
    int m_send_pid; /* sender pid */  
    int m_recv_pid; /* receiver pid */  
    int m_kdata[5]; /* extra kernel data place holder */  
#endif  
    int mtype;      /* user defined message type */  
    char mtext[1];  /* body of the message */  
};
```



# IPC: Message Buffering

- Problem
  - Multiple processes send messages to a process but that process does not do a receive for some time.
  - How does kernel keep track of such messages?
- Design issue:
  - How are messages buffered by kernel?

# IPC: Waiting Messages

- Design: let each process have a queue of waiting messages
  - extend the PCB to include:

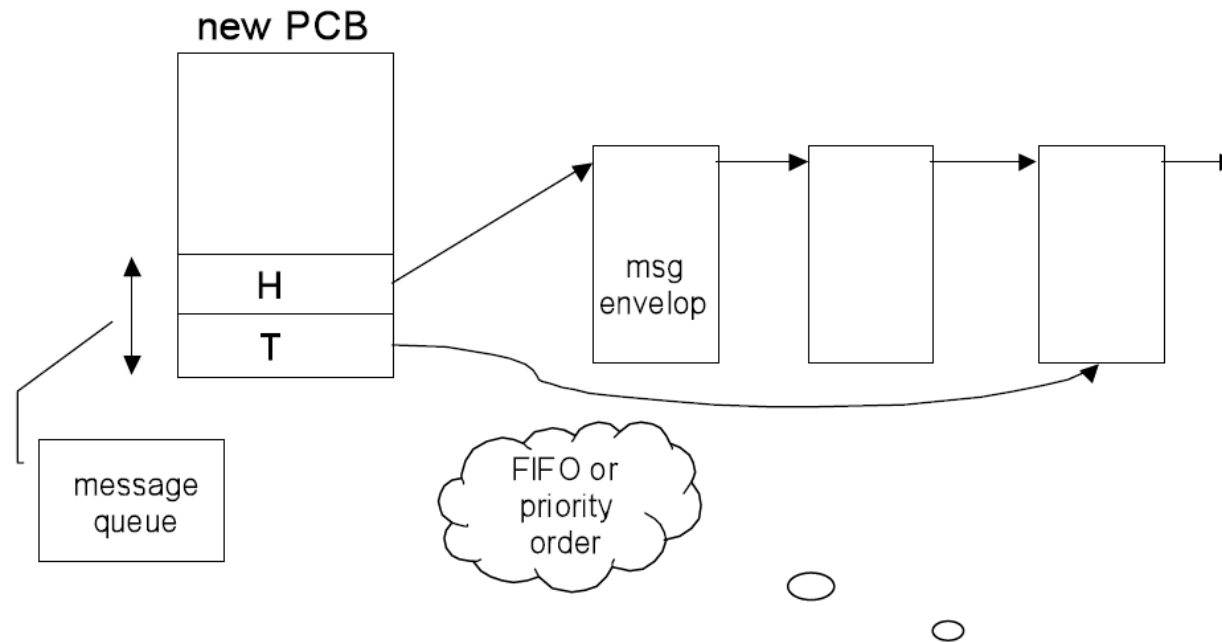


Image Courtesy of [1]

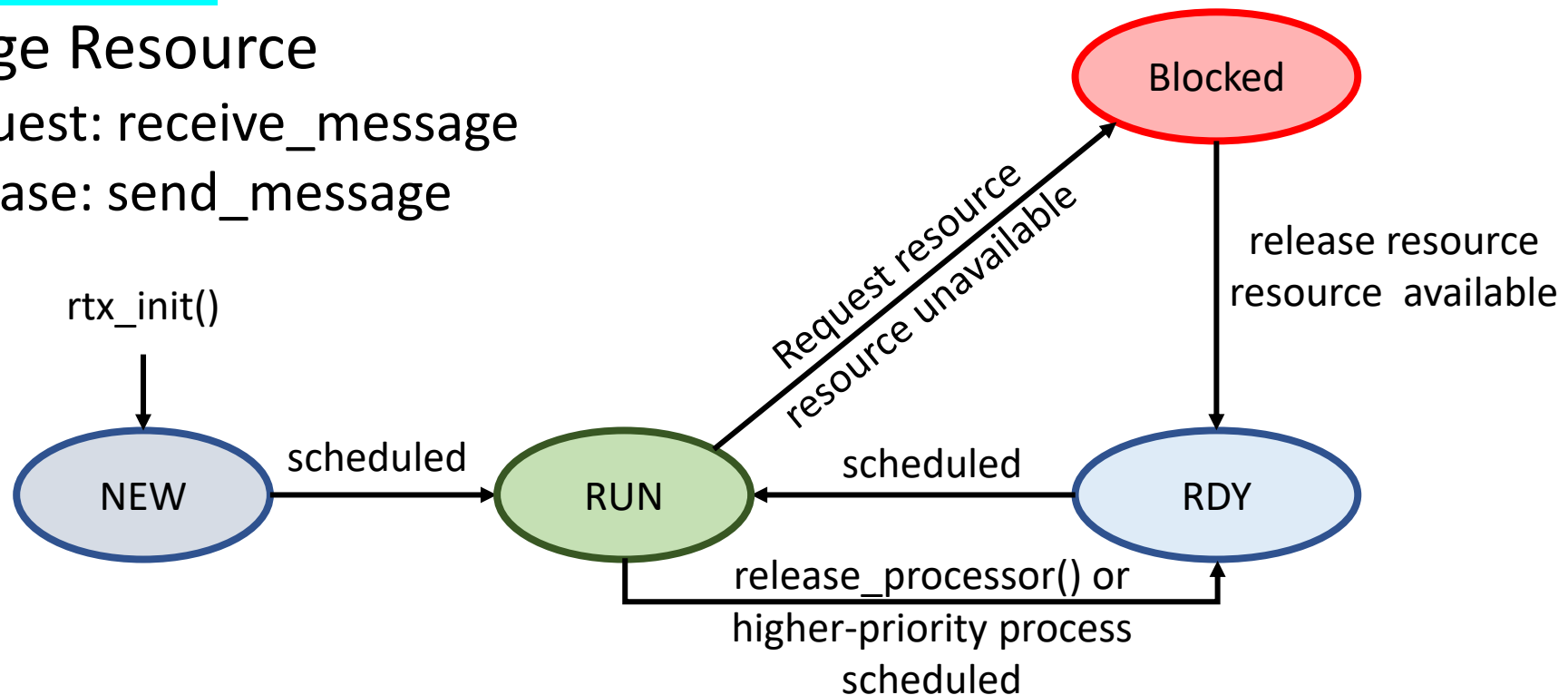
# IPC: Blocked on Receive

- What happens to a process that executes *receive* but no message available?
  - it blocks
  - its state is set to `blocked_on_receive`
- Design issue: should processes in this state be kept somewhere (queue, set ?)
  - why put on some Q?
  - no real need to do so.
  - just set its status to “`blocked_on_receive`” and that’s all

# P2 Process State Transition

```
/* k_inc.h starter code */  
typedef enum {NEW = 0, RDY, RUN, BLK_MEM, BLK_MSG} PROC_STATE_E;
```

- Add **BLK\_MSG** state
- Message Resource
  - Request: receive\_message
  - Release: send\_message



# IPC: send\_message

```
int send_message(int rcv_pid, void *msg_env) {  
    atomic(on);  
    set sender_pid, destination_pid fields in env  
    rcv_pcb ← convert rcv_pid to process obj/PCB reference  
    enqueue env onto the msg_queue of rcv_pcb  
    if ( rcv_pcb.state is blocked_on_receive ) {  
        set rcv_pcb state to ready  
        rpq_enqueue( rcv_pcb);  
    }  
    atomic(off);  
    return RTX_OK;  
}
```

How to change this for a preemptive system?

It is a non-blocking call!



# IPC: receive\_message

```
void *receive_message(int *sender_id) {  
    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  
    *sender_id ← get the sender ID info from the env  
    atomic(off)  
    return env  
}
```

How to change this for a preemptive system?

It is a blocking call.

We also need a non-blocking version of it.

# IPC: Code Excerpt

- Check Section 6.3 of Lab manual for IPC FAQs
- Code excerpt of a user process to send a message (page 36 of the lab manual)

```
struct msgbuf *p_msg_env = (struct msgbuf *) request_memory_block();  
p_msg_env->mtype = KCD_REG;  
p_msg_env->mtext[0] = '%';  
p_msg_env->mtext[1] = 'W';  
p_msg_env->mtext[2] = '\\0';  
send_message(PID_KCD, (void *)p_msg_env);
```

# 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

# Starter Code

- Use your existing P1
- Create a directory called P2
  - Copy existing P1/Context\_Switching Project to P2

```
├── P1
│   └── Context_Switching
├── P2
│   └── Context_Switching
└── README.md
```

# 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



# Thank you!

Electrical and Computer Engineering Department