RTOS Functional Overview

- Process Management
  - create/terminate processes
  - perform reset or 'power-on' initialization
- Processor Scheduling
  - selection & dispatch of processes for execution
- Inter-Process Communication/Synchronization
  - send/receive messages
  - signal/wait on semaphores
- Storage Management
  - allocate/deallocate memory

RTOS Functional Overview [2]

- Interrupt Handling Framework
  - capture interrupts
  - if required, activate a user defined process
- Timing Services
  - relative [delay] services
  - absolute time services
- Device Driver Interfaces
  - provide standard i/o and specialized interrupt driven device handlers

Simple RTOS: Requirements

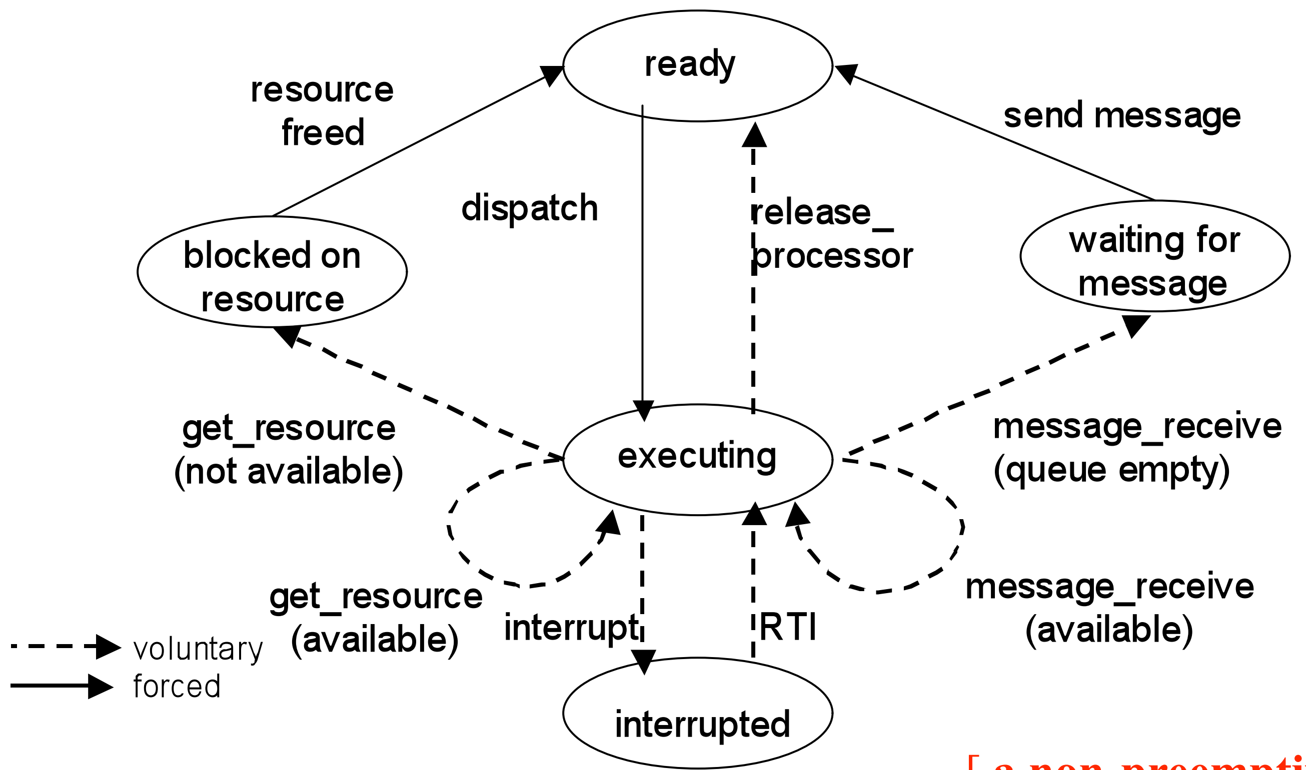
- basic requirements
  - non-preemptive
  - support for processes [creation at init time only]
  - priority scheduling [fixed priority]
  - message-based interprocess communication [asynchronous, messages sent in envelopes]
  - memory management: message envelopes
  - basic timing services

Simple RTOS: Setting

- Assumptions concerning RTOS setting
  - all processes are known and created at OS initialization time
  - processes are friendly, cooperating and non malicious
  - each process ‘knows’ the process_id of its co-workers

RTOS Process States

- simplified



[a non-preemptive system]

RTOS: Atomicity

[conceptual only]

- RTOS primitives must execute *indivisibly*
- define private kernel function *atomic(on / off)*
  - *atomic(on)* enables the atomic functionality
    - first executable statement in each primitive
  - *atomic(off)* disables the atomic functionality
    - last statement in each primitive (before 'RET')
- possible implementation
  - extra field: int atomic_count = 0; in **each** PCB
  - atomic(on) increments, atomic(off) decrements this field
  - whenever atomic_count > 0, atomicity must be enforced by kernel

RTOS: Atomicity Cont'd

- if there is direct access to CPU interrupt masking, then explicit *atomic()* function is not needed
- possible implementation of *atomic*
  - *on:* save interrupt system mask,mask all interrupts
  - *off:* restore interrupt system mask

must use assembly language instructions

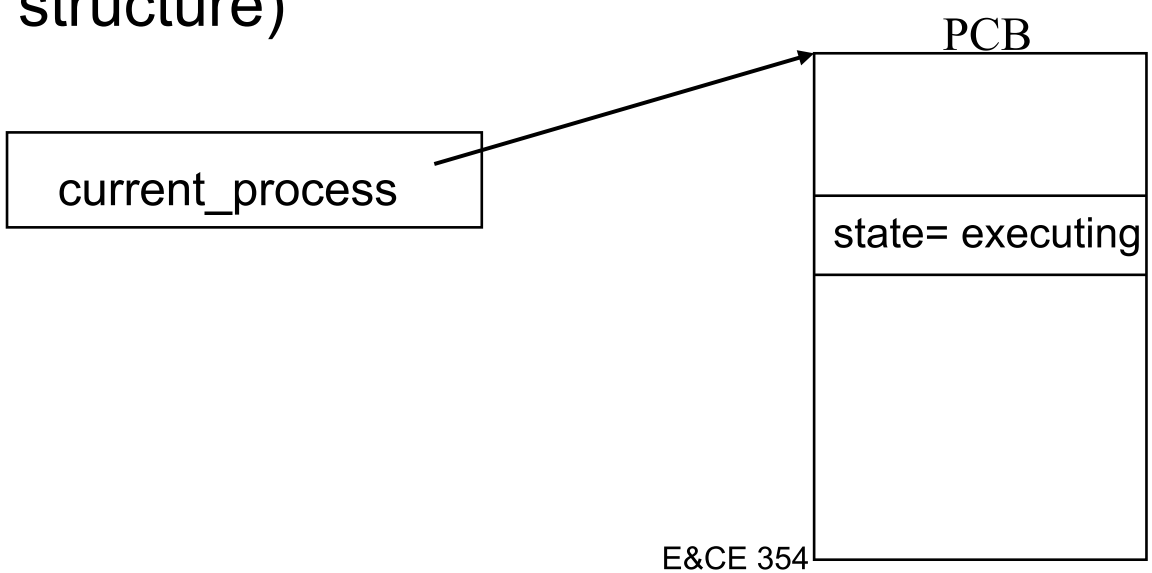
RTOS: Atomicity Cont'd

- in the following slides, the *atomic(on/off)* functionality is used as to indicate the need to enforce <u>indivisibility</u> of kernel primitives
- Question:
  – must the atomic(on/off) functionality be implemented exactly as described?
- Answer:
  – discussed in lectures……

Atomic(on/off) concept used only in slides to **<u>stress</u>** the requirement of atomicity -- since in reality, all the primitive functionality will be performed in "kernel mode" with interrupts disabled by a kernel version of the user visible API functions, after the user API 'traps' into the kernel.

RTOS: *current_process*

- RTOS must know which process currently executes
- RTOS design includes a private kernel variable *current_process*
- *current_process* always refers to the currently executing process (more exactly, to its internal representation, e.g. process object or PCB data structure)



RTOS: *process_switch()*

- frequently needed procedure: remove the currently executing process from the CPU, select the next process to execute and give the CPU to it
- RTOS design includes a private kernel function *process_switch*()
  - invokes the scheduler to select the next process to be executed
  - invokes *context_switch(next_process)*

RTOS: *context_switch(next_proc)*

- save context of currently executing process into its PCB/process object
- sets *current_process* to refer to *next_proc*
- sets the state of the *current_process* to executing
- restores the context of *current_process*
- causes the *current_process* execution to begin

RTOS: *process_switch()* cont'd

- after a primitive executes *process_switch*:
  - the invoking process will eventually regain control again
  - execution will eventually resume on the instruction immediately following the process_switch instruction
  - when?

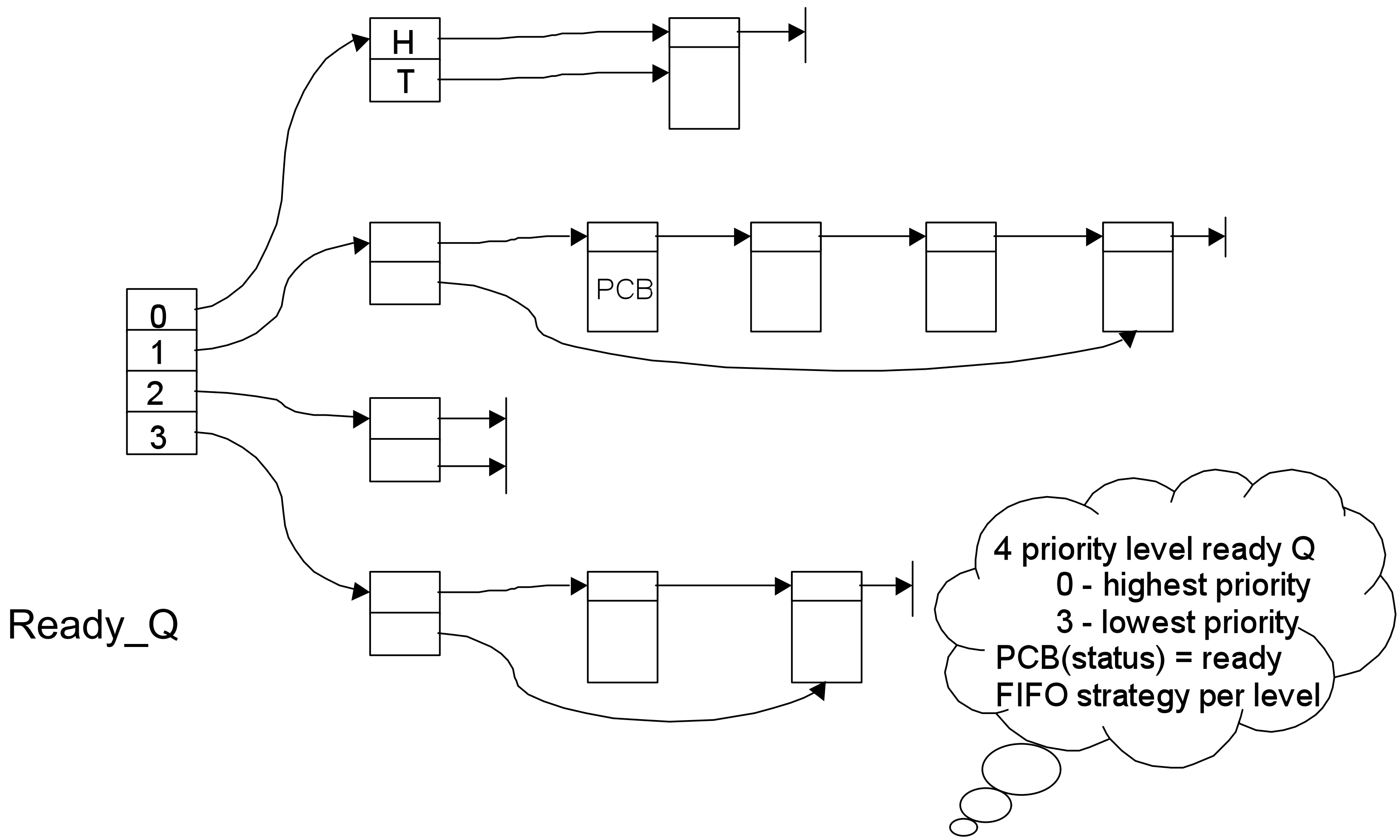
RTOS: Scheduling

- requirement spec: fixed priority based
- each process assigned a priority (urgency)
  - highest priority ready process will get to execute
  - processes with equal priority treated as FCFS
- applicable to both preemptive and nonpreemptive RTOS
- possibility of <u>indefinite-blocking</u> (starvation, livelock)
  - arrival rates of high priority processes may be so high that a low priority process may wait for extended time/forever to execute

RTOS: Scheduling

- *process_switch()* invokes the scheduler
- scheduler
  - selects the highest priority ready process
- *process_switch* then invokes *context_switch(next_proc)* to let the selected process execute
- note: other scheduling algorithms used in more complex real-time operating systems

RT OS: Priority Scheduling



Real-Time OS – *rpq_enqueue/dequeue*

- fixed priority based scheduling ⇒ our design includes:
  - private kernel functions

    rpq_enqueue(PCB/proc object)
    enqueues the PCB/process object on the appropriate ready process queue based on its priority

    rpq_dequeue()
    dequeues and returns reference to highest-priority ready process

RTOS: Null Process

- CPU must always execute something
- what should the RTOS do when the scheduler finds that the ready queue is empty?
  - possible solution:
    - loop within RTOS, periodically check **(be careful!)**
    - make sure that the ready queue is never empty!!
- how?
  - include a process (*null process*) with the lowest priority that is <u>always</u> ready to run

RTOS: Null Process Cont'd

- basic null process functionality:

```
null_process:
    while (true) {
        release_processor();
    }
```

- should the null process do more than that?
- two views
  - strict view: one process, one function, hence no
  - permissive view: let it do something useful
    - e.g. ROM checksum check, low level OS checks

RTOS: *release_processor()*

- RTOS design includes the following primitive:

  release_processor( )

- release_processor:
  - set current_process state to ready
  - rpq_enqueue (current_process)
  - process_switch()

RTOS: Initialization

- what operations need to be carried out at OS startup (i.e. after power up, reset)?
- initialize all HW, OS structures, create processes and start process execution

RTOS: Process Initialization Table

- RTOS must know which processes to create
- our design: array of records (initialization table, IT)
- each record contains the information necessary to start its respective process
- a record could have the following structure:

Record_i

process_i_id
process_i_priority
process_i_initial_SP
process_i_initial_PC

may hold max size of stack needed, OS assigns actual SP

usual case, why??

RTOS: Initialization Sequence

- during initialization, RTOS
  - initializes all hardware
  - creates and initializes all kernel data structures
  - reads IT
    - creates PCBs/process objects as needed (proc_status=ready)
    - places each PCB into its respective scheduling ready queue
    - invokes scheduler to select first process to execute
    - lets the selected process start executing

RTOS: IPC

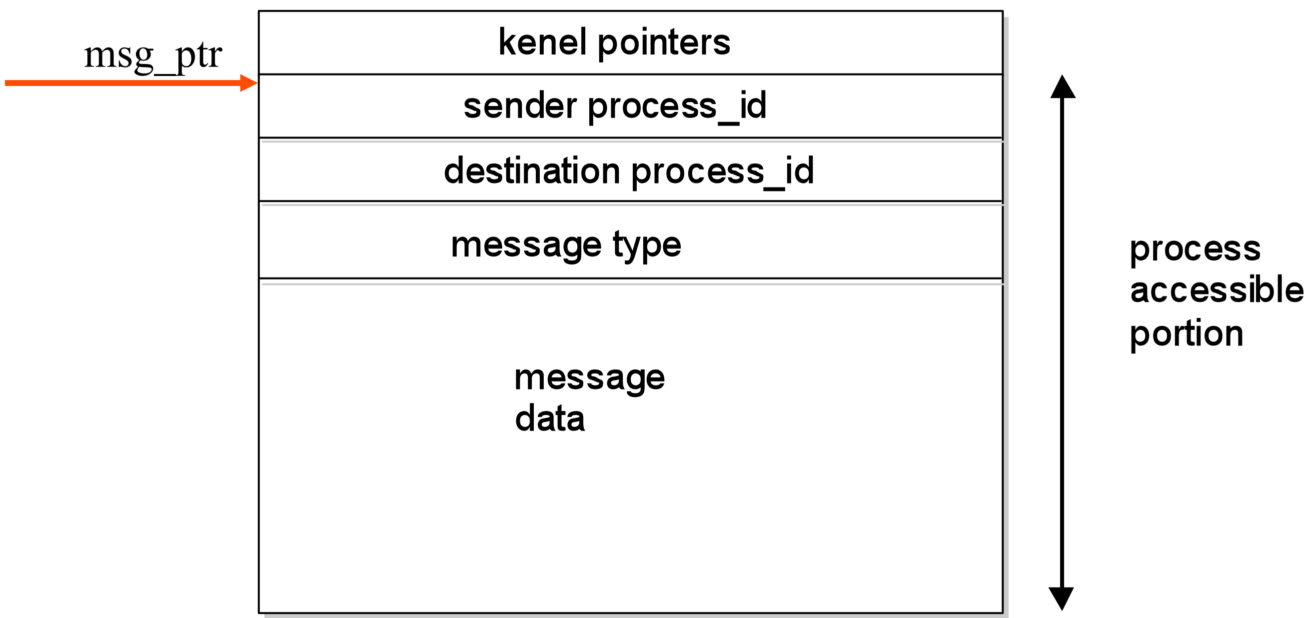
- requirement spec:
  - message-based, asynchronous IPC
  - messages carried in shared message blocks (msg envelopes)
- each process writes a message into a msg envelope
- process invokes *send(msg_envelope)*
- issues:
  - what is the format of the message envelope (i.e. envelope memory block)?
  - where do these memory blocks come from?

RTOS: Message Envelopes

- message envelopes managed by the kernel
  - an appropriate number of message envelopes (blocks of memory) created at system init time
  - a process allocates a message envelope to send a message
  - a process deallocates an envelope when it is no longer needed (<u>current owner of the envelope!</u>)
  - a process owns a message envelope that it receives or allocates (until it is sent)

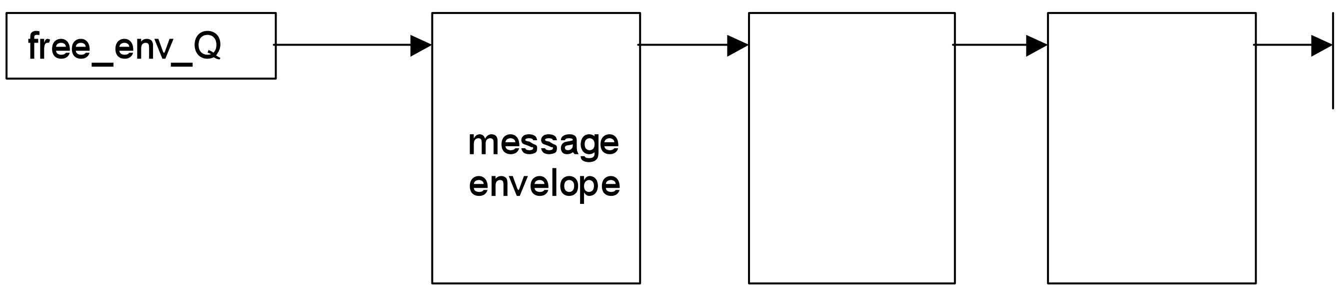
RTOS: Msg Envelope Format

- design of message envelope format:
  - fixed size block of memory
  - layout:



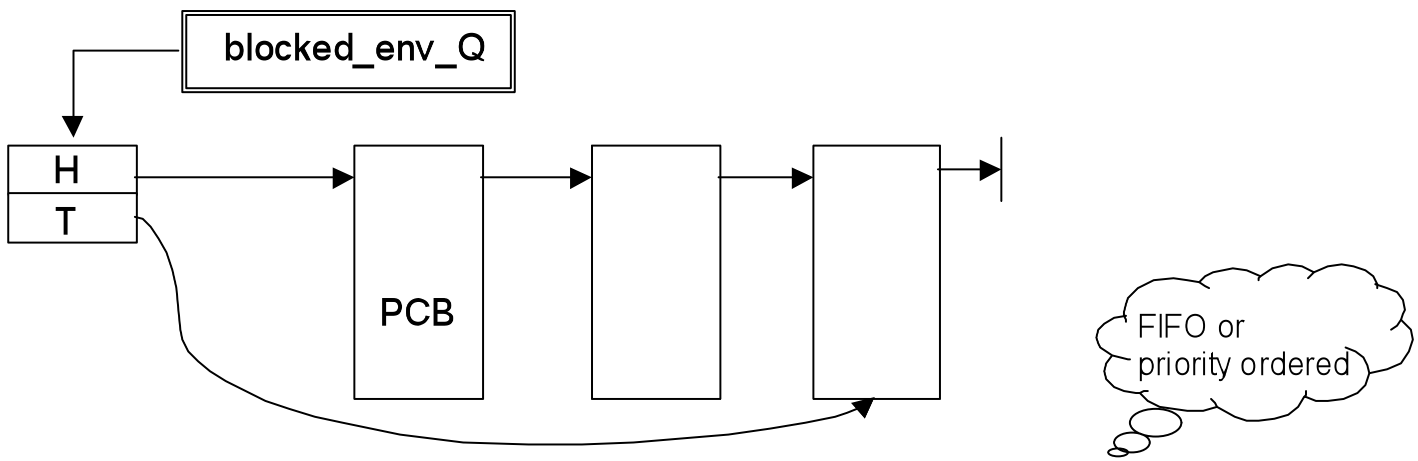
RTOS: Msg Envelope Management

- where do message envelopes come from?
  – at initialization, kernel creates a fixed number of envelops and put them on a free envelope queue



RTOS: Msg Envelope Management

- if a process does not already have an envelope then it first allocates an envelope from the kernel
- what if no envelopes left?
  - requesting process blocks
  - our design: blocked processes kept on blocked_env_Q



PCB(status)=blocked_on_env_Q

RTOS: *allocate_envelope*

- functionality of *allocate_envelope* primitive:

```
allocate_envelope() {
    atomic(on);
    while (free_env_Q is empty) {
            put process object/PCB on blocked_env_Q
            set process state to blocked_on_env_allocate
        process_switch();
        ***restart here when blocked process executes eventually
        }
        env← reference to de-queued envelope
return env;
atomic(off);
}
```

RTOS: *deallocate_envelope*

- functionality of *deallocate_envelope*:

```
deallocate_envelope( in: env ) : {
    atomic(on);
    put env onto free_env_Q
    if ( blocked_env_Q not empty)
       { dequeue one of the blocked processes
         set its state to ready and enqueue it on ready process queue    }
    atomic(off);
}
```

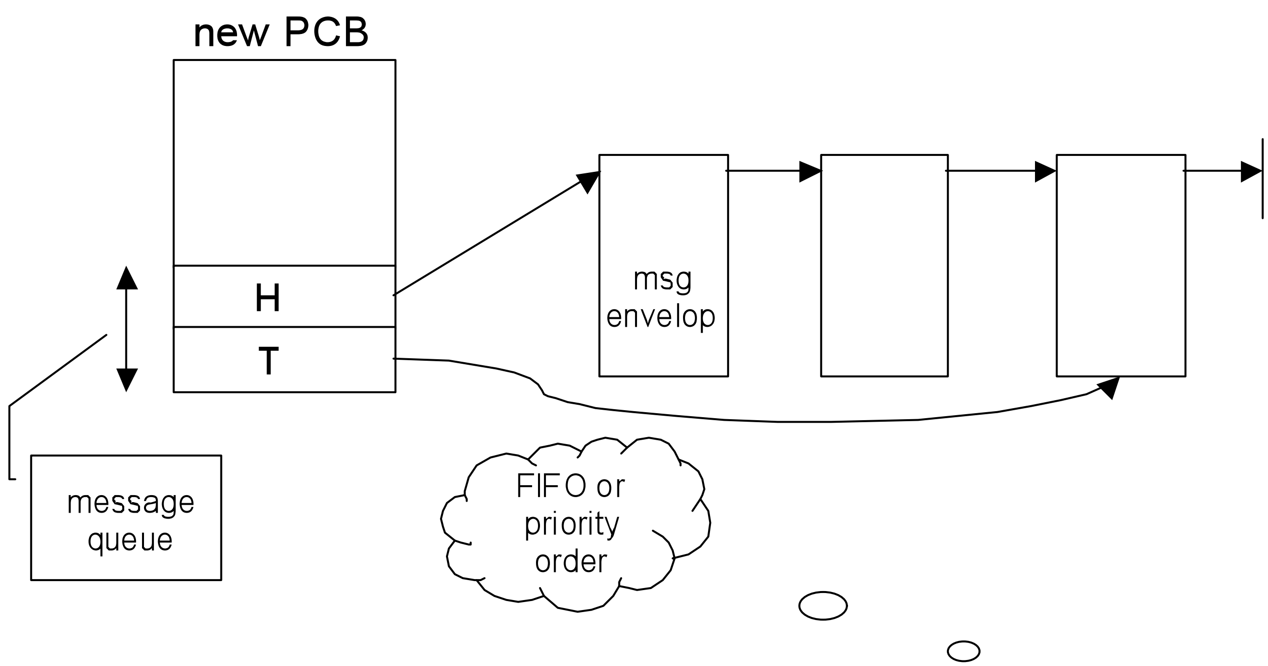
- invoking process never blocks!
- how does it work with respect to *allocate_envelope* kernel primitive??

RTOS: IPC

- how does a process send/receive a message?
- requirement spec: message-based, asynchronous
- design decision
  - non-blocking *send*
  - blocking *receive*
- design issue:
  - how are messages buffered by kernel?
    - if multiple processes send a message to a process but that process does not do a receive for some time
    - how does kernel keep track of such messages?

RTOS: Waiting Messages

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



RTOS: IPC

- 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

RTOS: *receive*

- functionality of *receive* primitive:

```
receive() {
  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
   return env
atomic(off);
}
```

RTOS: *send*

- functionality of *send* primitive:

```
send( 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);
}
```

What about a preemptive system?

RTOS: Interrupt Handling

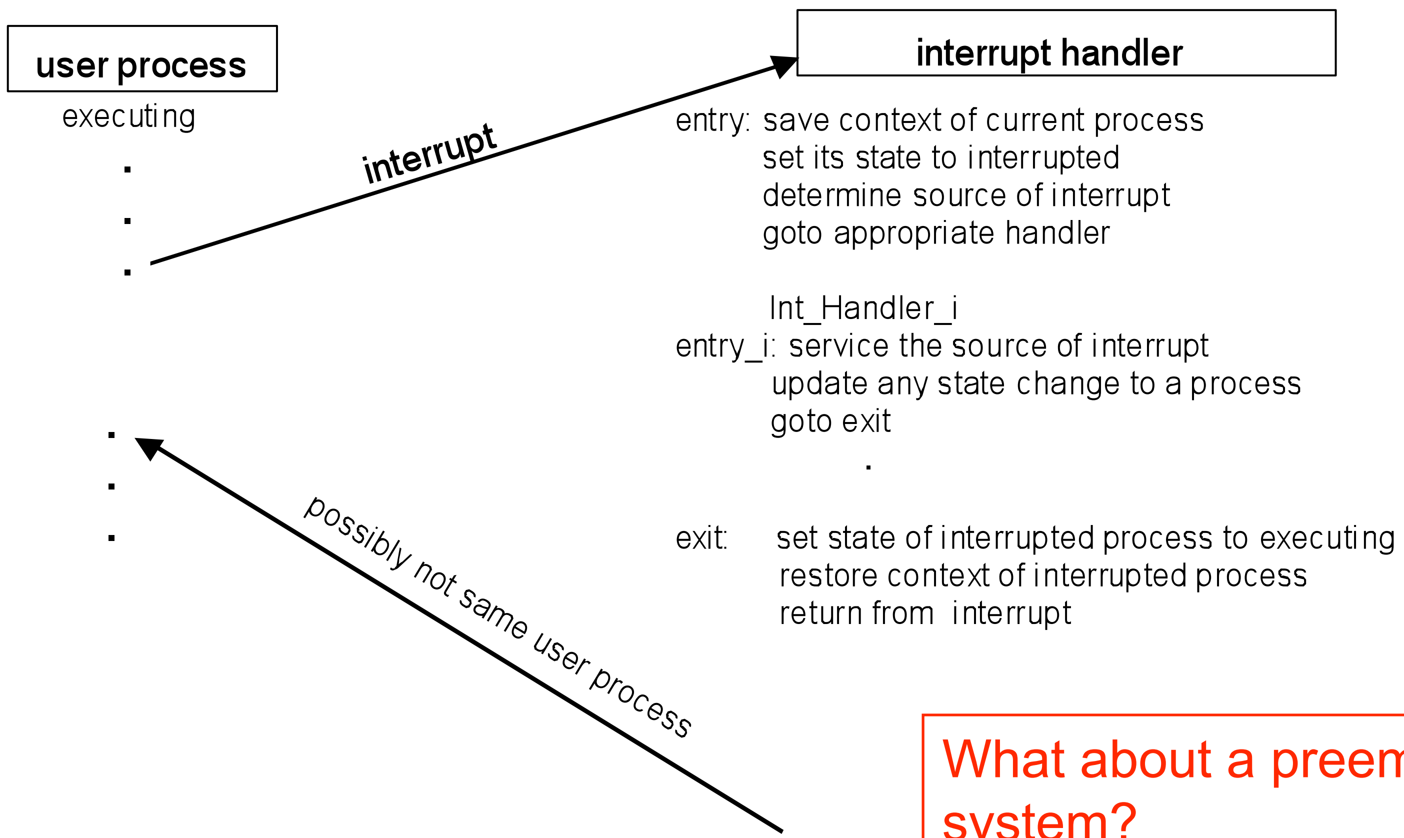
- in real-time OS, interrupt handling must be fast
  - short latency to respond to interrupt
  - fast processing by interrupt handler
- interrupts may cause a change in state for some blocked process
  - e.g. a process blocked for external event to occur will have its state changed to ready and be placed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

RTOS: 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

RTOS: Interrupt Handling Sequence

- abstracted interrupt processing sequence (nonpreemptive system)



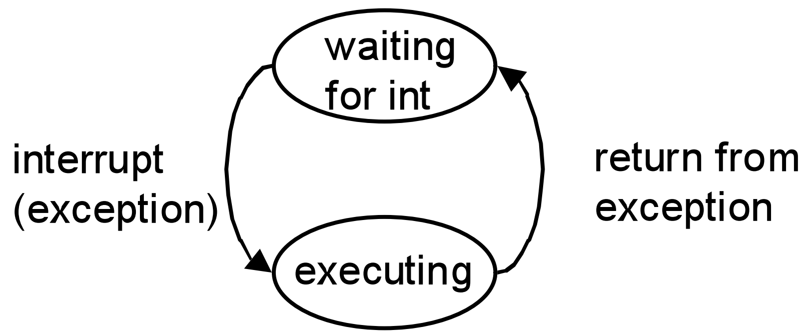
What about a preemptive system?

RT OS: Interrupt Handler Design

- interrupt handler must interact with OS processes
- alternatives:
  - multiple ad-hoc interaction mechanisms
  - i_process
    - an i_process gets the CPU from an interrupt handling sequence, not through the dispatcher.
    - <u>never</u> blocks if it invokes a kernel primitive
    - the interrupt (exception) handling routine ~~and~~ starts the appropriate i-process.
    - conceptually: i-process has the max priority, is scheduled by interrupt

RTOS: i-process

- state diagram for an i_process:



- a process object/PCB is associated with each i_process with state = i_process (permanently)

- always ready to run, but not on any ready Q

RTOS: 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

RTOS: Interrupt Handling

- we can now detail the previous exception handling sequence
- note: *save_proc* stores reference to the process/PCB which was executing (was the current_proc) when the interrupt occurred

exception_handler:

begin

set the state of current process to interrupted

save_proc = current_process

select interrupt source

A: context_switch (i_proc_A)

break

RTOS: Interrupt Handling

```
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
end;
```

RTOS: Timing Services

- fundamental service in real-time operating systems
- service categories:
  - sleep: defer execution for *n* seconds
    - voluntarily give up CPU until the specified time expires, then be put back on ready queue
  - timeout notification:
    - request kernel to inform process when a specified time period has expired; process continues execution
  - repetitive timeout notification:
    - repeated timeout notification until cancelled

RTOS: Timing Services 2

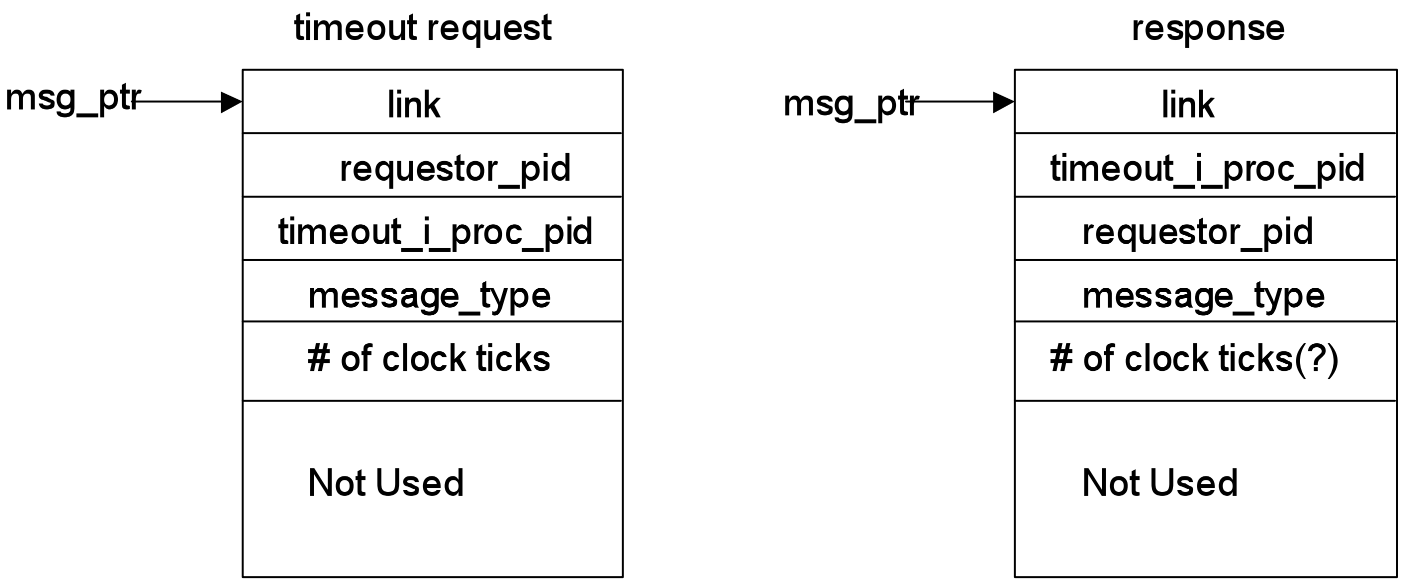
- service requests could be stated in:
  - relative time (*x* clock ticks)
  - absolute time (February 24, 10:34:22 AM, 2002)
  - others as appropriate
- related functionality
  - cancellation of earlier request

RTOS: Timing Service Design

- 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 send a request message to the timing i_process
  - timeout notification: the i_process sends a message back

RTOS: Timing Service Request

- *send(timeout_i_process, message)*
  - message contains a timeout request
  - request format:

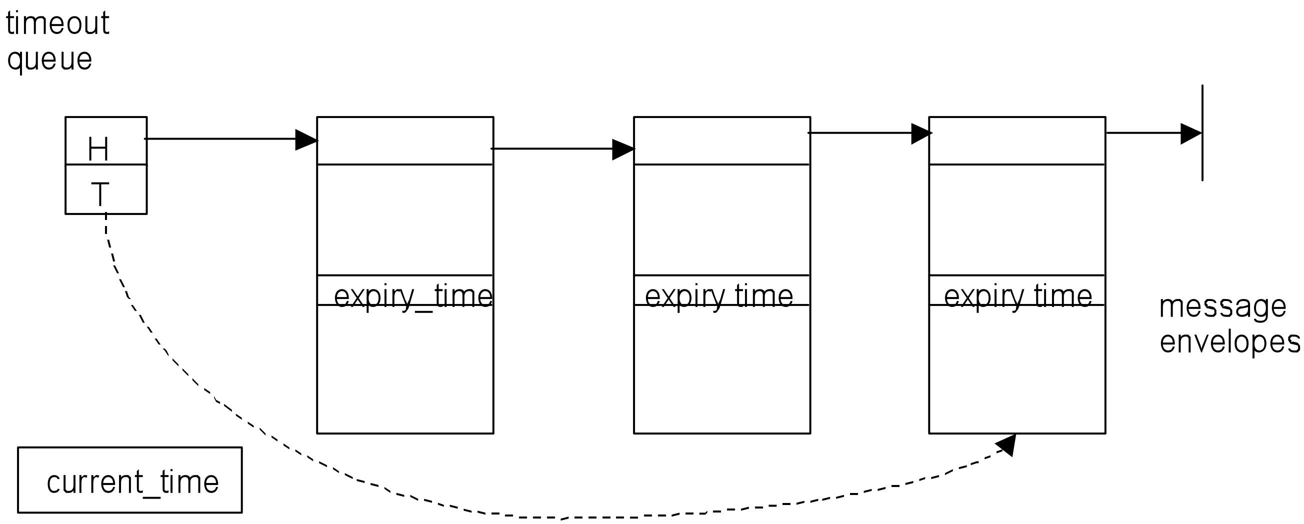


(sent to timeout i_process) (sent from timeout i_process)

RTOS: Timing Service Request 2

- user processes ‘know’ the pid of the timeout i_process
- after the expiration of the time, the timeout i_process sends the original message envelope back to requestor
- timeout service maintains requests in a sorted list

RTOS: Timing Service Design



- 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

RTOS: timeout i-process

- at each clock tick (hardware timer interrupt), the timeout i-process:
  - increments current_time
  - invokes *receive*() to see new requests
    - since it is an i_process, it cannot block!!
  - if any new requests, adds them to sorted list
  - checks whether any timing requests have expired, if yes, it sends the notification using the request message envelope back to the requester

RTOS: timeout i-process 2

- basic outline:

```
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
   }
   //continued next slide
```

RTOS: timeout i-process 3

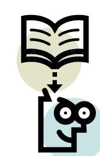
```
if (timeout_list is not empty)
   {
          while    (head_of_timeout_list.expiry_time    <=
   cur_time)
     {
       env ← timeout_list.dequeue();
       target_pid ← env.source_pid;
       env.source_pid ← timeout_i_process_pid;
       send( target_pid, env );    //return envelope
     }
  }
}
```

RTOS: Preemption

- certain RTOS primitives can make ready a process whose priority is higher than that of current process
  - *send*
  - *deallocate_envelope*
- pre-emptive OS $\Rightarrow$ highest priority ready process should execute
- preemption is relatively easy to add to the design presented

Design Changes for Preemption

- on return from *send,* etc primitives, include check
  – if hpr process priority > current process priority, do context switch
- problem when i-process executes
- possible solution
  – let priority of i-processes be max priority
  – on return from interrupt, check whether the priority of the interrupted process still the highest
  – if not, context switch to hpr process



entry