CT420 REAL-TIME SYSTEMS

POSIX - INTRODUCTION

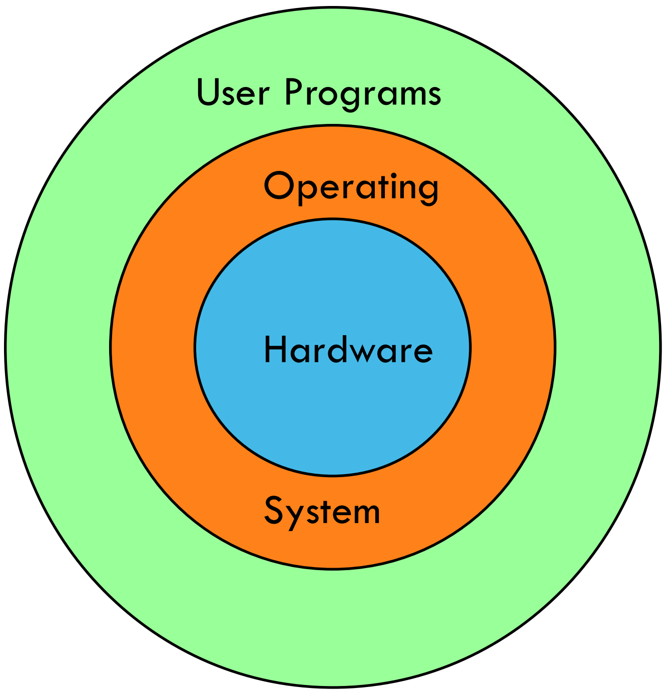
Dr. Michael Schukat

Ollscoil na Gaillimhe
University of Galway

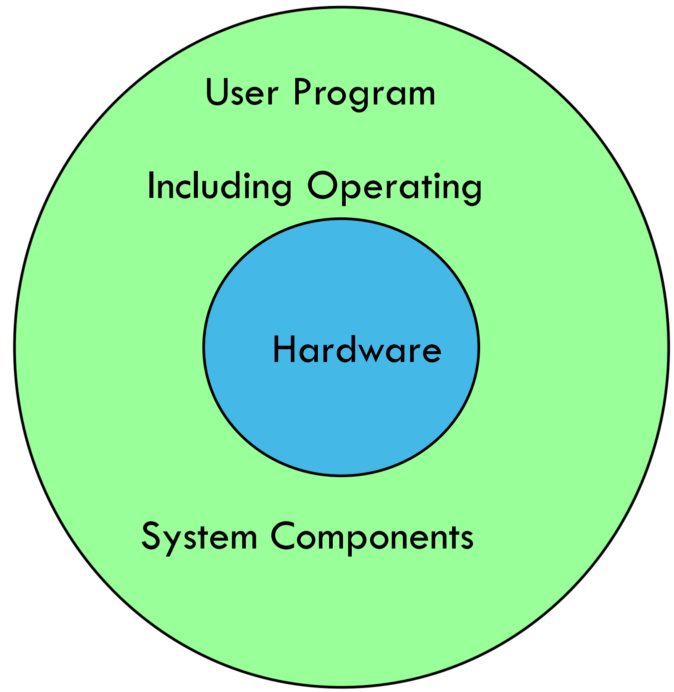
Lecture Overview

- Introduction POSIX
  - RTOS Challenges
  - POSIX standard
  - Process scheduling in POSIX
  - POSIX.4 clocks & timers
  - Memory locking

RTOS versus no RTOS



Typical OS Configuration



Typical CE Configuration

RTOS

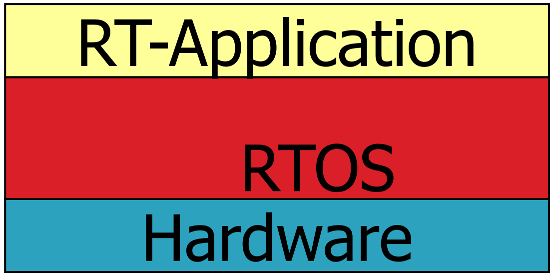
Real time in operating systems:

"The ability of the operating system to provide a required level of service in a bounded response time."

RTOS

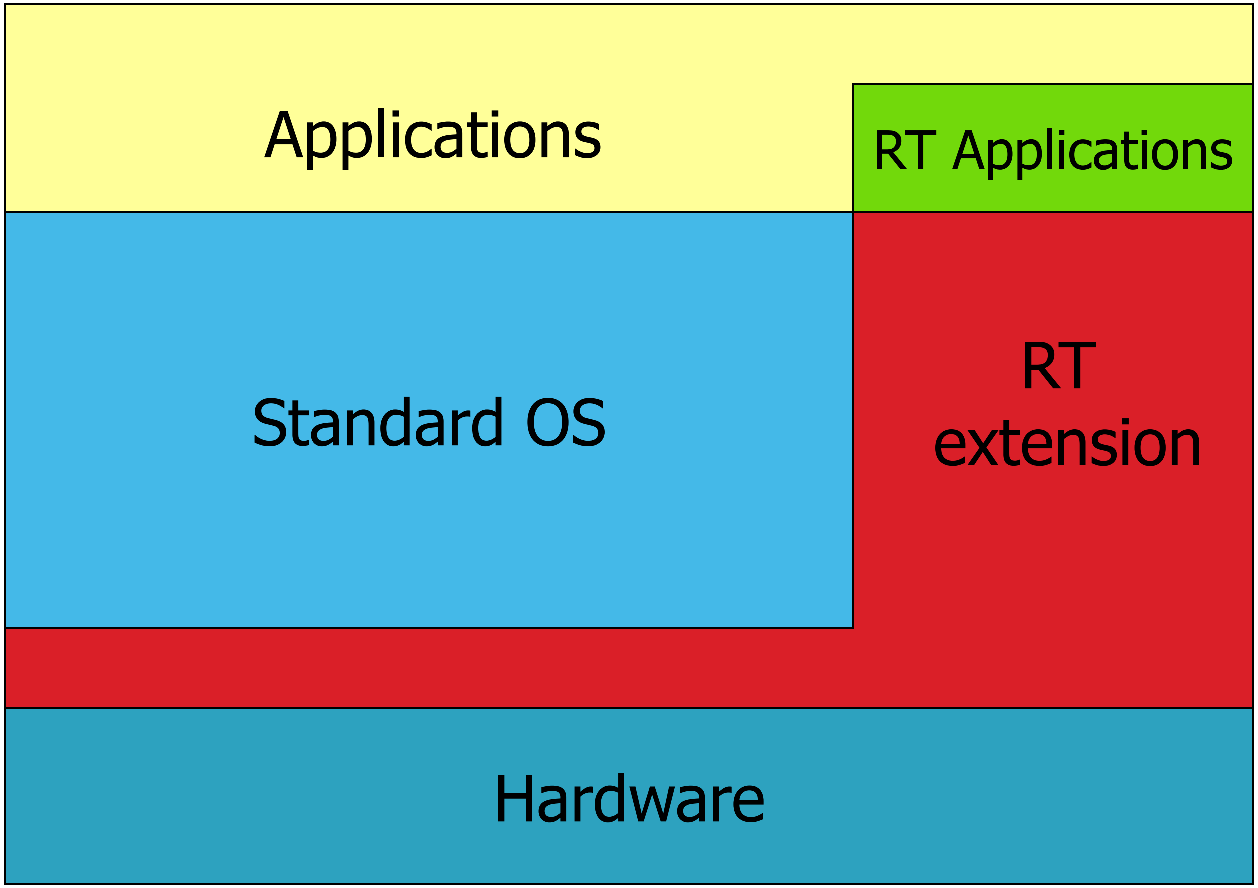
- A hard real-time operating system must, without fail, provide a **response** to some kind of event **within a specified time window**
  - **i.e. task scheduling (→ CE) is a response after a timer interrupt**
- This **response must be predictable and independent of other activities** undertaken by the operating system on behalf of other tasks. Providing this response implies that system calls will have a **specified, measured latency period**
  - This is NOT a common feature of many OS kernels and device drivers, see slides 9 ff

Pure RTOS



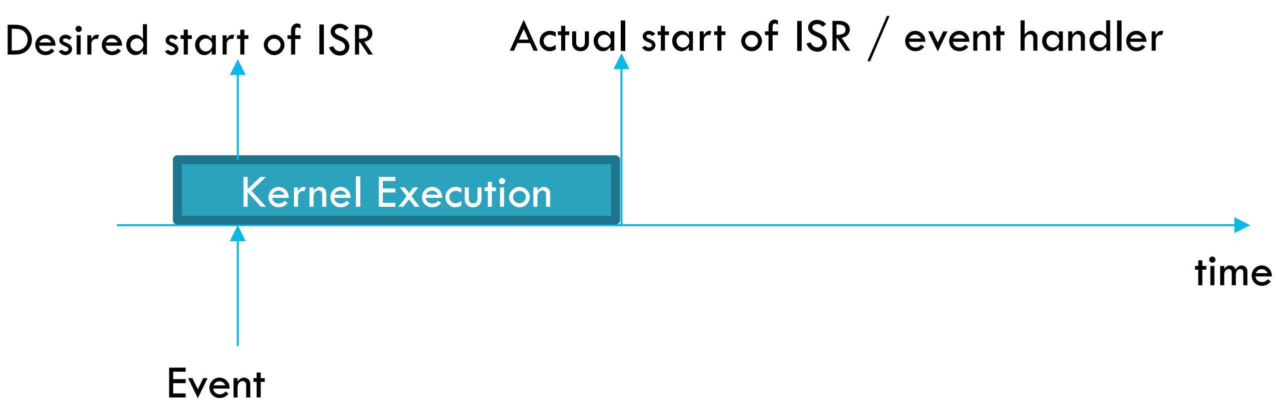
- The entire RTOS is built from scratch
- Example VxWorks
  - Proprietary RTOS by Wind River Systems
    - See http://www.windriver.com/products/vxworks/
    - Fully POSIX.4 Compliant included pre-emptive FIFO priority scheduling
    - Continuously improved since the 1990s
  - Widely used, even in safety critical systems
    - Boeing 787 (aviation industry)
    - Router/Switches
    - Mars Pathfinder

OS Real-Time Extensions



Problem 1 with RTOS Extensions: Re-entrant Code and ISRs

- Interrupts are disabled at several sections in the kernel to protect OS data from corruption (saves the effort of making functions re-entrant)
- This adds unpredictability to the amount of time it takes to respond to an event, i.e., the execution of an interrupt service routine (ISR)



Example for non-re-entrant Code [Wikipedia]

Some kernel call of swap()

```c
int t;

void swap(int *x, int *y)
{
 t = *x;
 *x = *y;
 // hardware interrupt might invoke isr() here!
 *y = t;
}

void isr()
{
 int x = 1, y = 2;
 swap(&x, &y);
}
```

t	*x	*y
undef	3	4
3	3	4
3	4	4
3	1	2
1	1	2
1	2	2
1	2	1
1	4	1

In-class Activity

Example for non-re-entrant Code [Wikipedia]

```
int t;

void swap(int *x, int *y)
{
 t = *x;
 *x = *y;
 // hardware interrupt might invoke isr() here!
 *y = t;
}

void isr()
{
 int x = 1, y = 2;
 swap(&x, &y);
}
```

t	*x	*y
undef	3	4
3	3	4
3	4	4
3	1	2
1	1	2
1	2	2
1	2	1
1	4	1

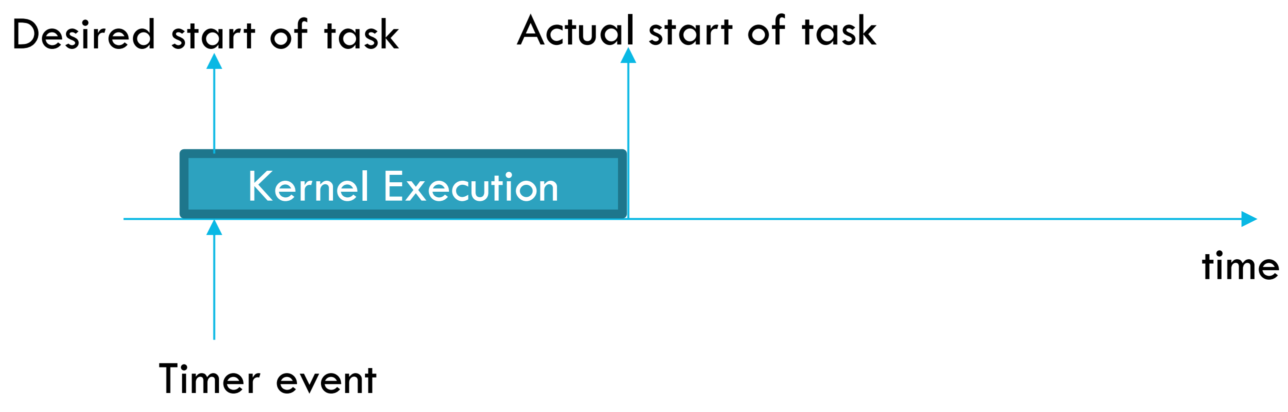
- How can you make swap() re-entrant?

Problem 2 with RTOS Extension: Process Scheduling

- Most standard OS implement a <u>CPU-time sharing system</u> designed to optimise average performance (as weighted by priorities), considering
  - good user experience
  - fairness
  - maximising throughput
- This means that as the load increases any real time processes (i.e. processes with tight response times) will suffer the most
- Subsequently an RTOS extension must implement its own scheduler

Problem 3: Process Scheduling

- Standard OS do not provide a reliable mechanism **to wake a task up at a certain time**
- Long non-pre-emptible system calls executed by the kernel can interfere with timing constraints → Problem 1



Real-Time Linux Options: RTLinux

- RTLinux, short for "Real-Time Linux," is a microkernel operating system that provides a hard real-time computing environment
- It is an extension to the standard Linux kernel to provide real-time capabilities, as follows:
  - **Real-Time Microkernel**: RTLinux uses a microkernel architecture. A small real-time kernel runs beneath the standard Linux kernel. This real-time kernel handles real-time tasks, ensuring that they meet strict timing requirements
  - **Standard Linux Kernel**: The standard Linux kernel is modified to run as the lowest-priority task in the real-time system. This allows non-real-time applications to run alongside real-time tasks without interfering with their timing guarantees
- See https://wiki.linuxfoundation.org/realtime/start

Real-Time Linux Options: The PREEMPT_RT Patch

- This patch transforms the Linux kernel into a fully preemptible kernel by
  - making critical sections of the kernel preemptible and
  - implementing priority inheritance (later)

  thereby reducing latency and improve determinism in scheduling (in combination with the POSIX FIFO scheduler (next slide)
- The patch
  - has been fully integrated into the mainline Linux kernel, as of kernel version 6.12, released in September 2024
  - is included by default for most supported architectures like x86, x86_64, RISC-V, and ARM64
- See https://wiki.linuxfoundation.org/realtime/preempt_rt_versions

POSIX

- Portable Operating System Interface [for Unix]
  - Set of IEEE standards
  - Mandatory + Optional parts
  - Initiated in 1988, latest version is POSIX:2008
- **Objective: Source code portability of applications across multiple OS**
  - Standard way for applications to interface to OS
  - Mostly but not exclusively Unix type OS
  - *Total portability is not achievable*

POSIX Support

- Compiler Support
  - Options to include/invoke POSIX support
  - Eg. GNU C Compiler

```
gcc -lrt -o name name.c
```

- Headers
  - Set of header files that define POSIX interface supported on particular system

```
#include <unistd.h>
```

- Libraries
  - Implement POSIX functionality

POSIX.4 (Real-Time Extension)

- **Priority Scheduling**
- **Real-Time Signals**
- **Clocks and Timers**
- Semaphores
- Message Passing
- Shared Memory
- Asynch and Synch I/O
- Memory Locking Interface

POSIX.4 – Where can it be found?

- Implemented
  - in Linux kernels
  - by many pure RTOS (QNX, LynxOS, VxWorks, RT Linux, Integrity)
- And subsequently used in many Soft RTS …
  - Network switches
  - Multimedia applications
  - Navigation systems
  - IoT applications
- … and even Hard RTS in
  - Aviation
  - Robotics
  - Manufacturing

CE Schedule → POSIX

Task	Period p [ms]	Exec Time [ms]
A	25	10
B	25	8
C	50	5
D	50	4
E	100	2

- Tasks become processes
- We replace the a CE task schedule with a proper process scheduler

POSIX.4 RT Main Scheduling Policies

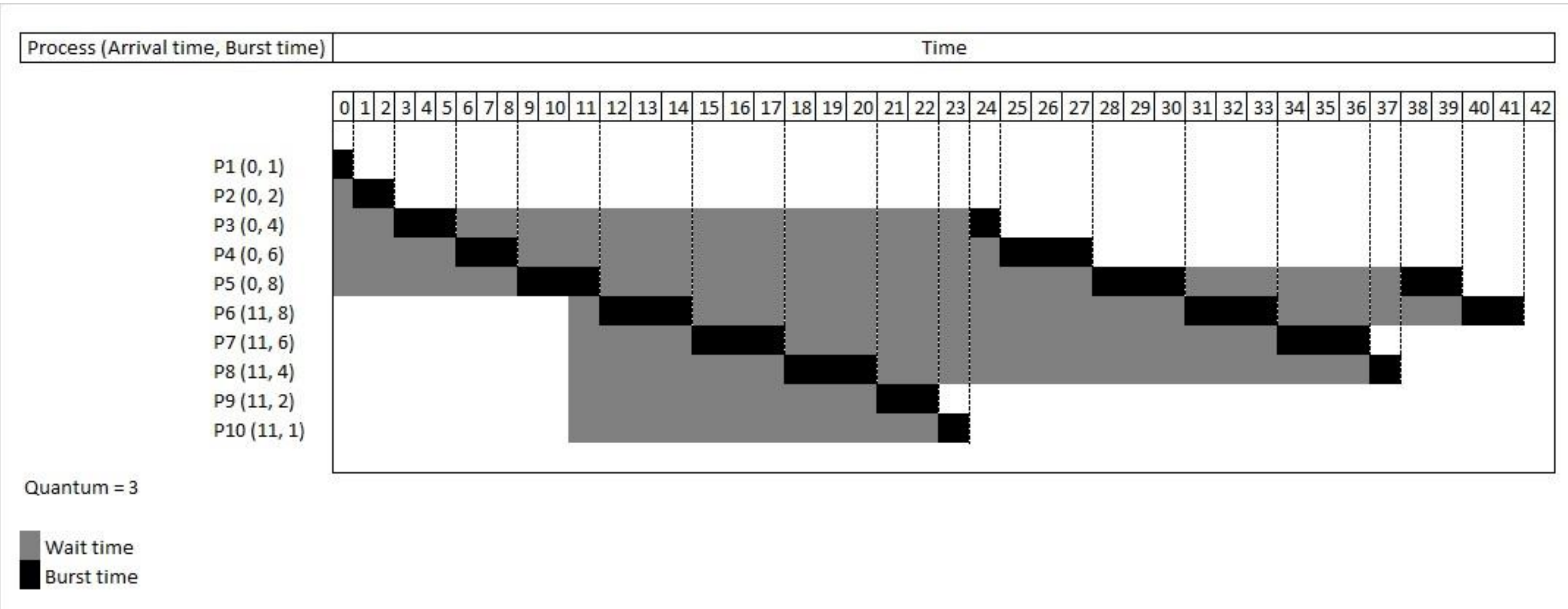
- `SCHED_FIFO`
  - SCHED_FIFO is a queue-based scheduler with different queues for each priority level (typically 32 levels)
  - Most common scheduler in RTS
  - An executed process either terminates, or is suspended if it
    - is blocked (e.g. waiting for timer signal → CE) – and is placed at end of the queue
    - invokes sched_yield(), i.e. suspends itself – and is placed at end of the queue
    - Is pre-empted by a higher priority process – and placed at top of its queue

POSIX.4 RT Scheduling

- SCHED_RR
  - SCHED_RR is a round-robin scheduler with each process having an execution time quota (i.e., timeslice or quantum)
  - The size of the timeslice (or quantum) can be system-wide (fixed or configurable) or specific for a priority level or a process
  - This scheduler typically used for lower priority tasks
  - Higher priority SCHEDS_FIFO tasks can pre-empt SCHED_RR tasks
  - Pre-empted processes are placed at top of queue
  - Processes that have used their quantum are placed at end of queue

POSIX.4 RT Scheduling

- Example Sched_RR (Source: Wikipedia):



POSIX.4 RT Scheduling

- Different policies can be used at the same time via concept of layers
- Note that the order of process execution is driven by their process priorities (1 … 10 in the example)

Layer	#Tasks	Tasks	Policy	Priority	Quantum
λ_1 (FPP)	4	τ_1	SCHED_FIFO	**1**	-
		τ_2	SCHED_FIFO	**2**	-
		τ_3	SCHED_FIFO	**3**	-
		τ_4	SCHED_FIFO	**4**	-
λ_2 (RR)	3	τ_5	SCHED_RR	**5**	15
		τ_6	SCHED_RR	**5**	5
		τ_7	SCHED_RR	**5**	10
λ_3 (RR)	5	τ_8	SCHED_FIFO	**6**	-
		τ_9	SCHED_FIFO	**7**	-
		τ_{10}	SCHED_FIFO	**8**	-
		τ_{11}	SCHED_FIFO	**9**	-
		τ_{12}	SCHED_FIFO	**10**	-

E.g. uncritical background tasks

Example


```
#include <sched.h>
void vMyProcess() {
  struct sched_param scheduling_parameters;
  int scheduling_policy;
  int i;
  scheduling_parameters.sched_priority=17;
  // getpid() returns the process id
  // i is just a return value
  i = sched_setscheduler(getpid( ),SCHED_FIFO,
  &scheduling_parameters);
  // continue
  // …
}
```

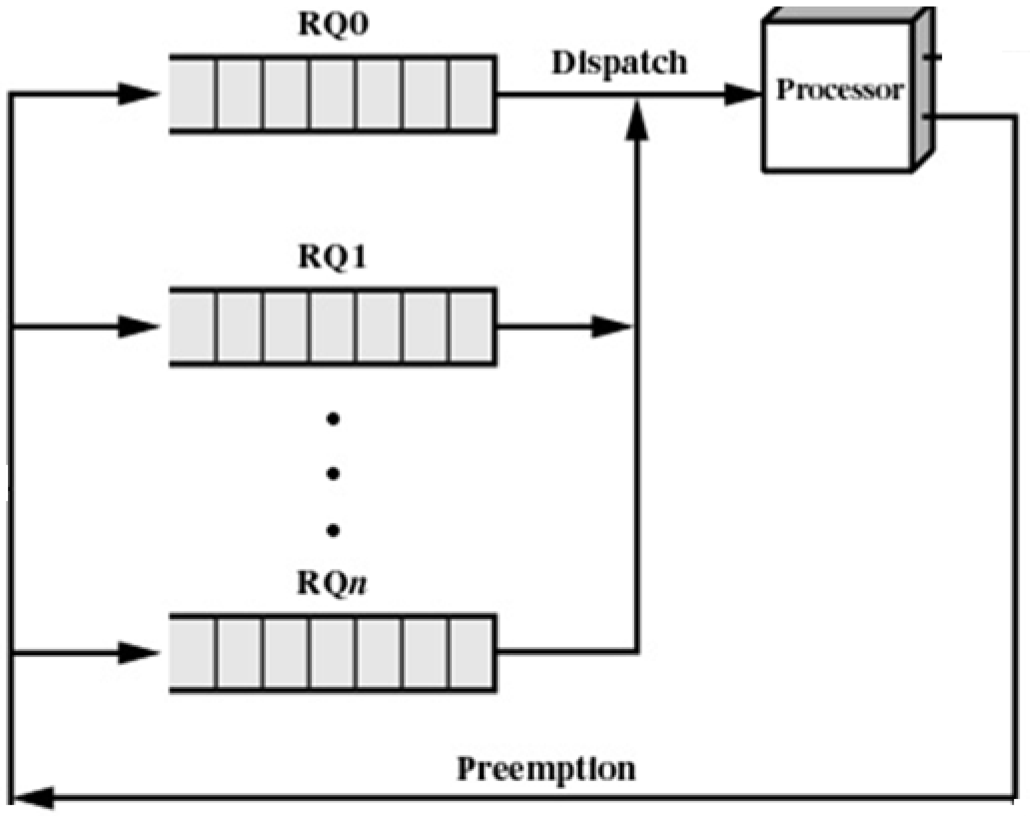
POSIX.4 RT Scheduling

- Process priority ranges differ among OS
  - Need this info before setting priority level

```
int sched_rr_min, sched_rr_max;
int sched_fifo_min, sched_fifo_max;

sched_rr_min = sched_get_priority_min(SCHED_RR);
sched_rr_max = sched_get_priority_max(SCHED_RR);
sched_fifo_min = sched_get_priority_min(SCHED_FIFO);
sched_fifo_max = sched_get_priority_max(SCHED_FIFO);
```

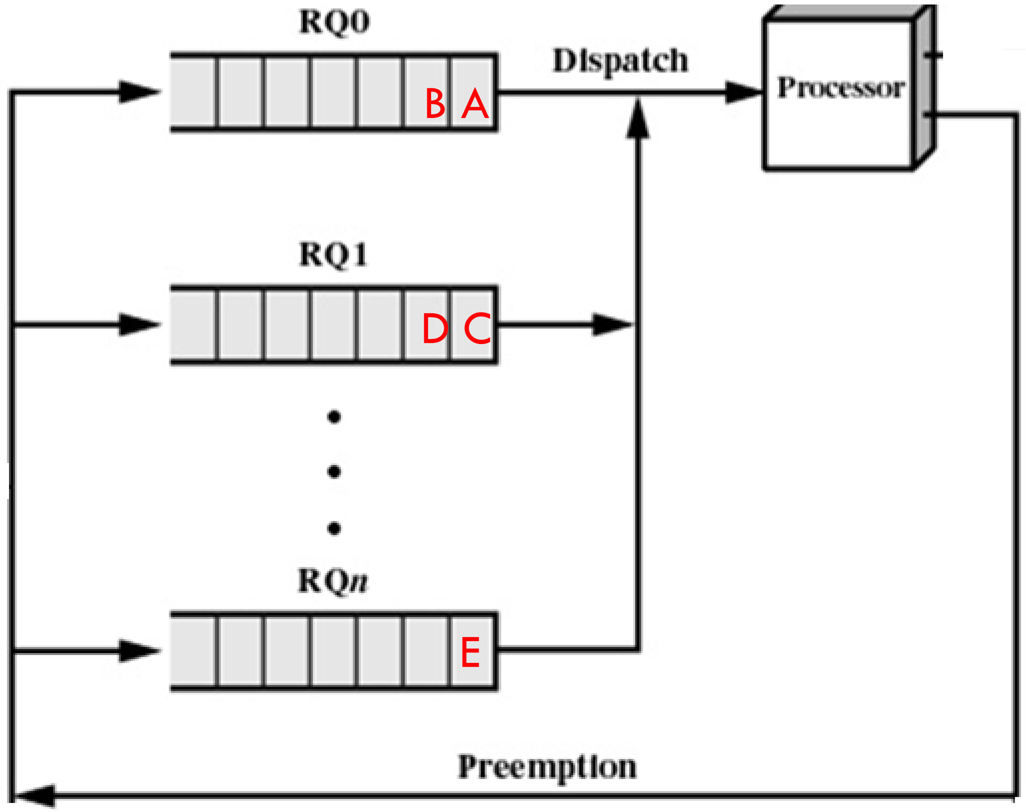
Scheduling with multiple Process Priorities



Blocked / suspended processes not included

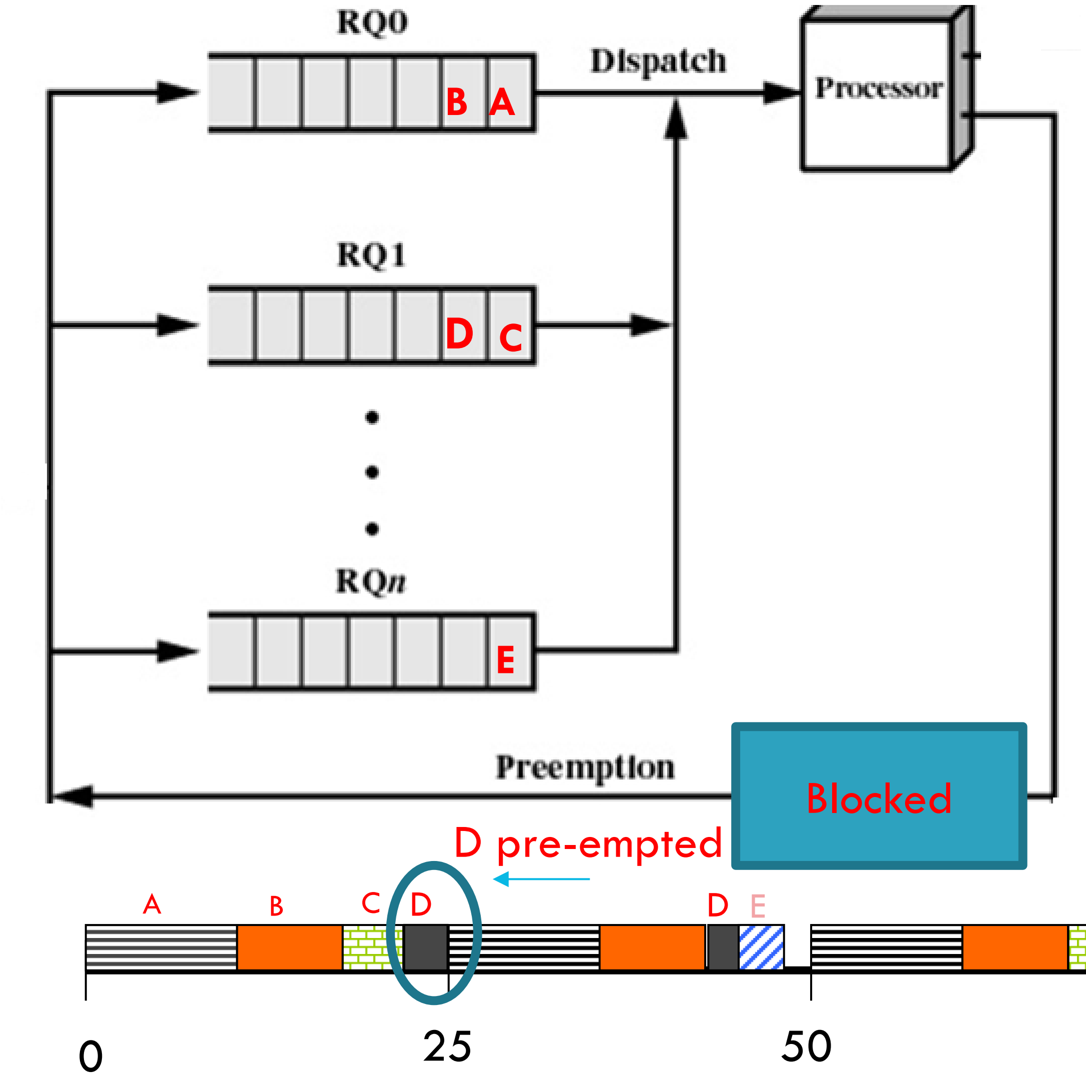
FIFO Scheduling with Multiple Process Priorities

Task	Period p [ms]	Exec Time [ms]
A	25	10
B	25	8
C	50	5
D	50	4
E	100	2



- Blocked / suspended processes not included
- Processes are free-running and do not adhere to major / minor cycle

FIFO Scheduling with Multiple Process Priorities




```
Process {A,B,C,D.E}:
int iProcessX() {
// Initialise process
// Setup interval timer x to notify it
// about begin of every cycle, e.g.
// T1: 25ms; T2 = 50ms; T3 = 100ms
while (1) {
  do_something();
  block_until_timer_signal();
}
}
```

POSIX.4 Clocks & Timers

- POSIX supports at least one clock
  - `CLOCK_REALTIME`
  - `CLOCK_REALTIME` clock is a system-wide clock, visible to all processes running on the system
  - Returns time in seconds and nanoseconds
  - But tick increment may be in the order of microseconds
- `timespec` structure (sec + nsec)
  - ```
    struct timespec{
      time_t tv_sec;
      time_t tv_nsec;
      }
    ```
- Typical specifies the number of seconds and nanoseconds since the base time of 00:00:00 GMT, 1 January 1970
    ```

POSIX.4 Clocks & Timers


```
#include<unistd.h>
#include<time.h>
#include <stdio.h>

int main(){
   struct timespec clock_res;
   int stat;
   stat=clock_getres(CLOCK_REALTIME, &clock_res);
   printf("Clock resolution is %d seconds, %ld
   nanoseconds\n",clock_res.tv_sec,clock_res.tv_nsec);
   return 0;
}
```

```
gcc -lrt -o name name.c
```

declan@crunchbang: ~/rts-lab

```
declan@crunchbang:~/rts-lab$ ./q1
Clock resolution is 0 seconds, 1 nanoseconds
declan@crunchbang:~/rts-lab$
```

POSIX.4 Clocks & Timers

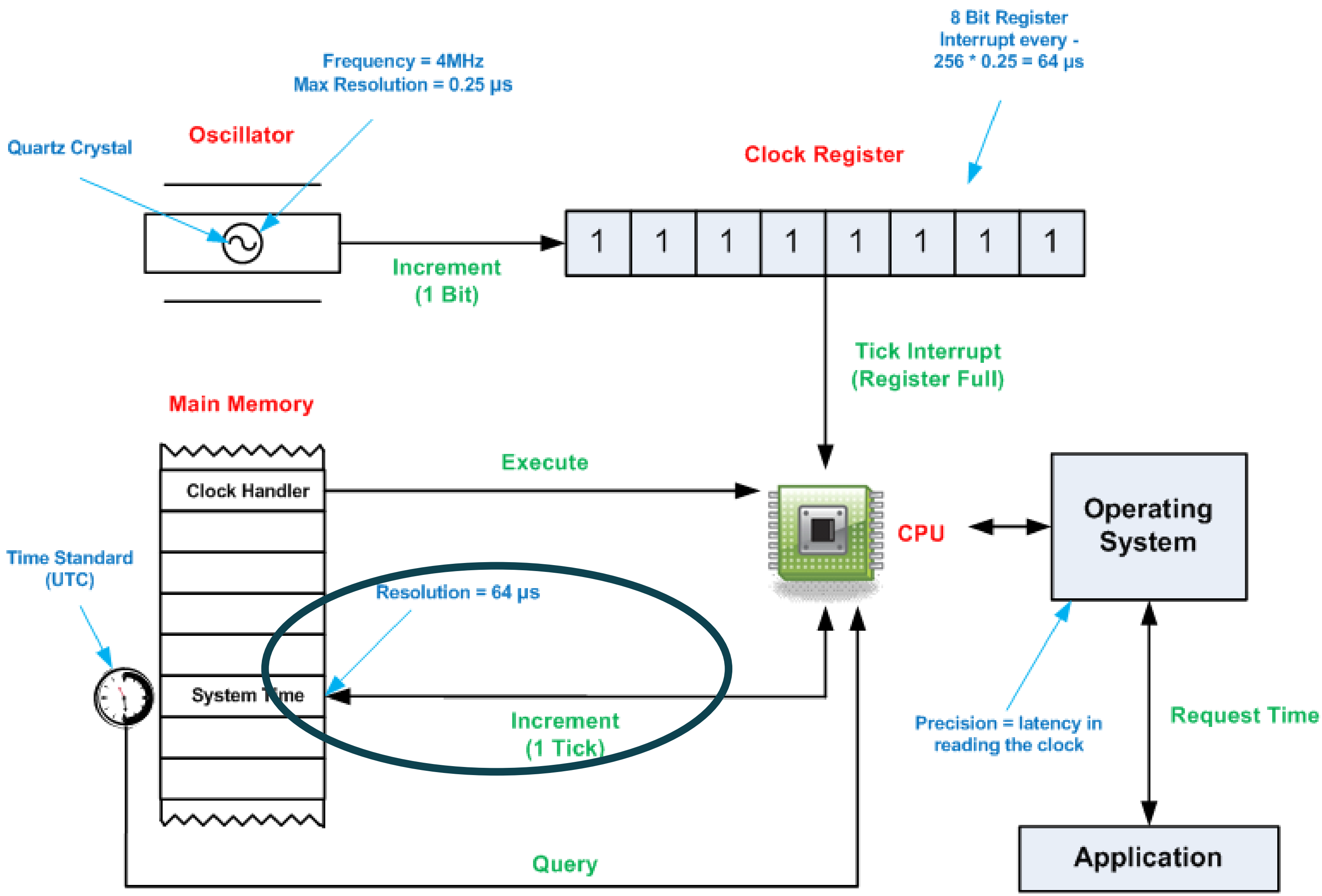
- `clock_getres(CLOCK_REALTIME,&realtime_res)`
  - `realtime_res` is `timespec` structure
- `clock_gettime(CLOCK_REALTIME, &time)`
  `clock_settime(CLOCK_REALTIME, &time)`
  (the latter requires appropriate privileges)

Linux and clock_getres()

- The clock_getres() function shall return the resolution of any clock, i.e. the increment value of a clocks' tick
- clock_getres() is often not appropriately implemented in Linux kernels; i.e., it shows either
  - (correctly) the actual increment between two clock ticks
  - (incorrectly) 1 ns, i.e. the smallest possible increment in a timer structure:

```
struct timespec {
    time_t tv_sec;
    time_t tv_nsec;
}
```

Recall: Clocks and Time Keeping in Computers



POSIX.4 Clocks & Timers

- `nanosleep(&nap,&time_left)`
- Delays the execution of the program for at least the time specified in &nap (of type `timespec`).
- The function can return earlier if a signal has been delivered to the process. In this case, it returns -1, sets errno to EINTR, and writes the remaining time into the `timespec` structure pointed to by time_left unless time_left is NULL. The value of time_left can then be used to call nanosleep() again and complete the specified pause

In-class Activity: Is this an acceptable Design to run a Task at 30ms intervals?


```
void ProcessA() {
        timespec start, delay, nextCall, current;
        clock_gettime(CLOCK_REALTIME,  &start);
        int count = 0;
        while (1) {
                do_something(); // Main activity of task
                count++;
                // Note that nextCall and delay calculations don't distinguish
                // between .tv_sec and .tv_nsec – this is a simplification for
                // this example
                nextCall = start + (count * 30);
                clock_gettime(CLOCK_REALTIME,  &current);
                delay = nextCall – current;
                nanosleep(delay, null);
        }
}
```

In-class Activity: Is this an acceptable Design to run a Task at 30ms intervals?


```
void ProcessA() {
        timespec start, delay, nextCall, current;
        clock_gettime(CLOCK_REALTIME,  &start);
        int count = 0;
        while (1) {
                do_something(); // Main activity of task
                count++;
                // Note that nextCall and delay calculations don't distinguish
                // between .tv_sec and .tv_nsec – this is a simplification for
                // this example
                nextCall = start + (count * 30);
                clock_gettime(CLOCK_REALTIME,  &current);
                delay = nextCall – current;
                nanosleep(delay, null);
        }
}
```

Process could be pre-empted around here

POSIX.4 Clocks & Timers

- Interval Timers
  - Useful to specify precise intervals
- `struct itimerspec{`

```
        struct timespec it_value;
        struct timespec it_interval;
        }
```

  `it_value` = 1st occasion of timer event

  `it_interval` = interval between subsequent events

  - it_interval = 0 => One time
  - it_value = 0 => Disable timer
- System calls
  - `timer_create( )` and `timer_delete( )`
  - Can have multiple timers within any process

POSIX.4 Clocks & Timers

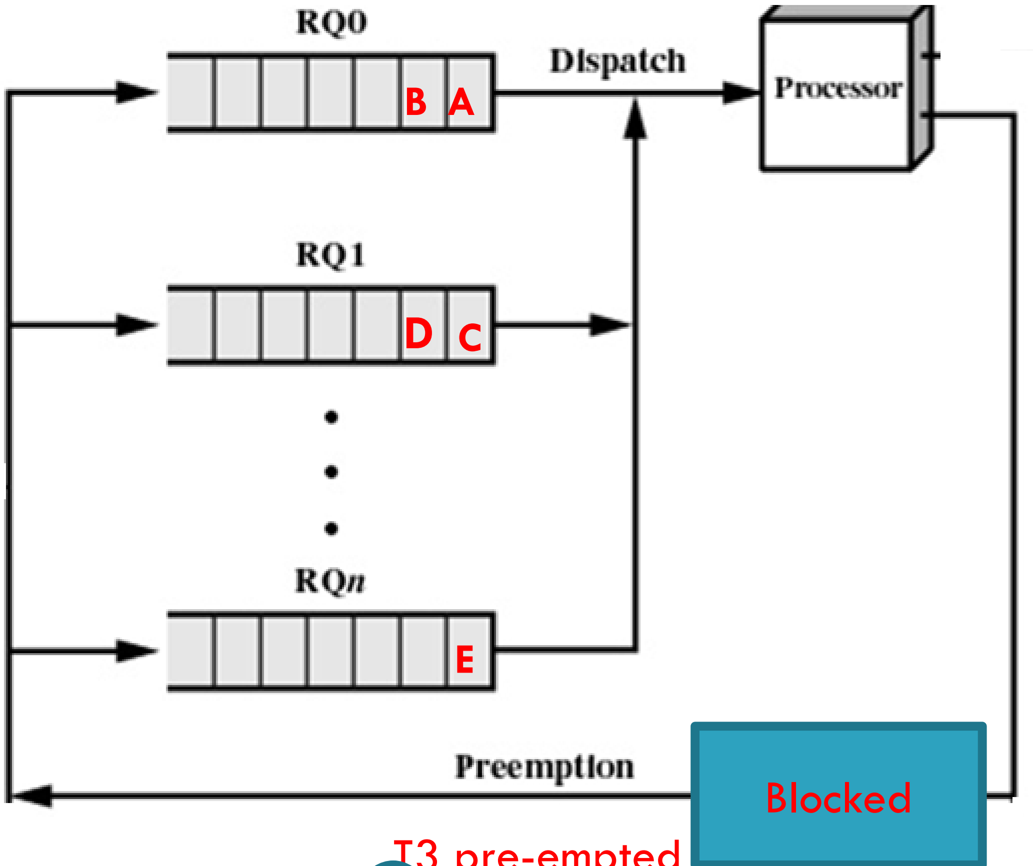
- Interval Timer example

```
timer_t created_timer;
// Second argument below relates to signal structure
// (later), that indicate what signals to be generated
// after timer expiration
// CLOCKID = CLOCK_REALTIME etc.
i = timer_create(CLOCKID, _ ,  &created_timer);
struct itimerspec new, old;
new.it_value.tv_sec=1;
new.it_value.tv_nsec=0;
new.it_interval.tv_sec=0;
new.it_interval.tv_nsec=100000;
i=timer_settime(created_timer, 0,&new, &old);
..
i=timer_delete(created_timer);
```

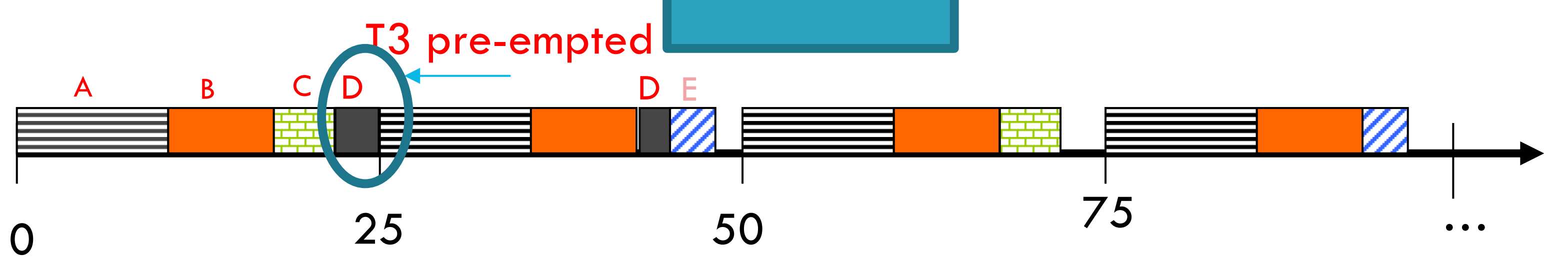
This parameter comes later

Value interpreted relative to the timer's content at the time of the call

Recap: Task Invocation using Timer



Process {A,B,C,D.E}:
```
int iProcessX() {
// Initialise process
// Setup interval timer x to notify it
// about begin of every cycle, e.g.
// T1: 25ms; T2 = 50ms; T3 = 100ms
while (1) {
  do_something();
  block_until_timer_signal();
}
}
```



Task Invocation using Interval Timer

Process A // do_something() invoked every 30 ms

```
timer_t created_timer;
int CLOCKID = CLOCK_REALTIME;
i = timer_create(CLOCKID, _ ,  &created_timer);
struct itimerspec new;
new.it_value.tv_sec=0;
new.it_value.tv_nsec=30000000;
new.it_interval.tv_sec=0;
new.it_interval.tv_nsec=30000000;
i=timer_settime(created_timer, 0,&new, null);
while (1) {
  do_something():
  waitforTimer();
}
```

POSIX.4 Clocks & Timers

- By default, the initial expiration time specified in new_value->it_value is interpreted **relative to the current time on the timer's clock** at the time of the call

- How about **absolute timer events?**
  - E.g. Timer event required ONCE at time t_{abs}

POSIX.4 Clocks & Timers

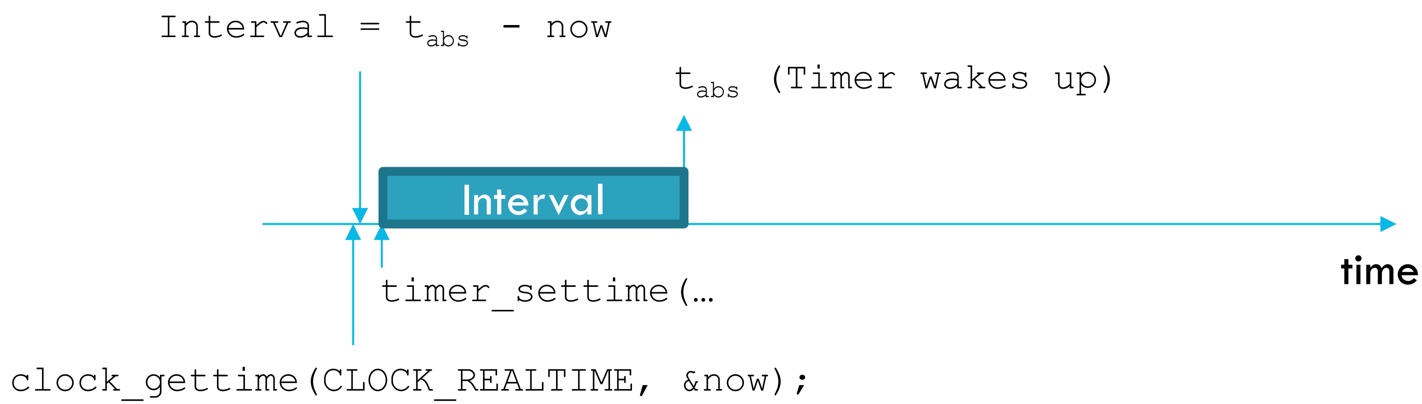
- Determine interval and use interval timer

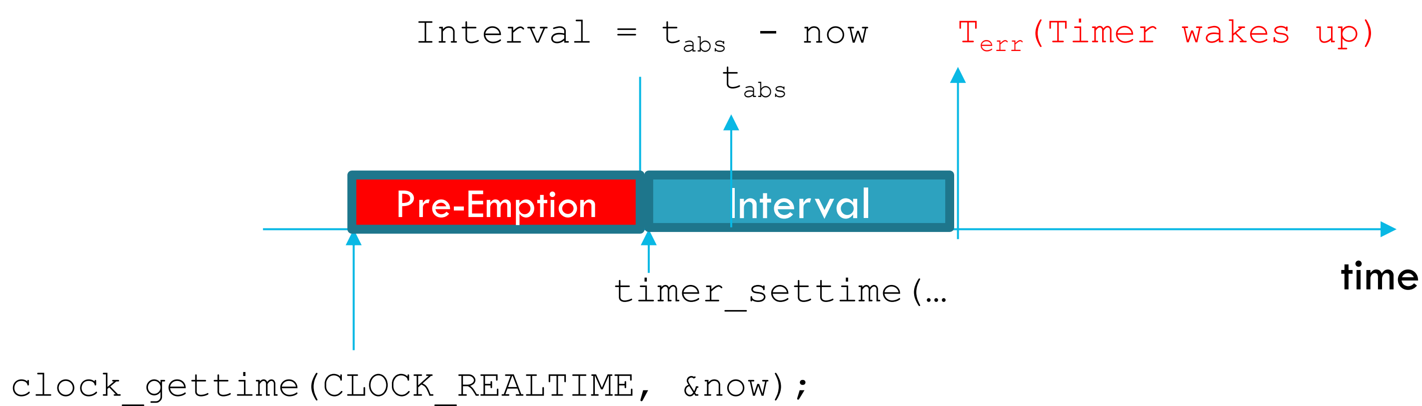
```
clock_gettime(CLOCK_REALTIME, &now);
// Calculate interval (simplified):
Interval = t_abs - now
// Create and set Interval timer:
timer_t created_timer;
struct itimerspec new,old;

timer_create(CLOCKID, _ ,  &created_timer);

new.it_value.tv_sec=Interval.tv_sec;
new.it_value.tv_nsec=Interval.tv_nsec;
new.it_interval.tv_sec=0;// Set interval to 0
new.it_interval.tv_nsec=0;
i=timer_settime(created_timer, 0,&new, &old);
// Block and wait for timer signal
...
```

Problem: Process Pre-Emption





POSIX.4 Clocks & Timers

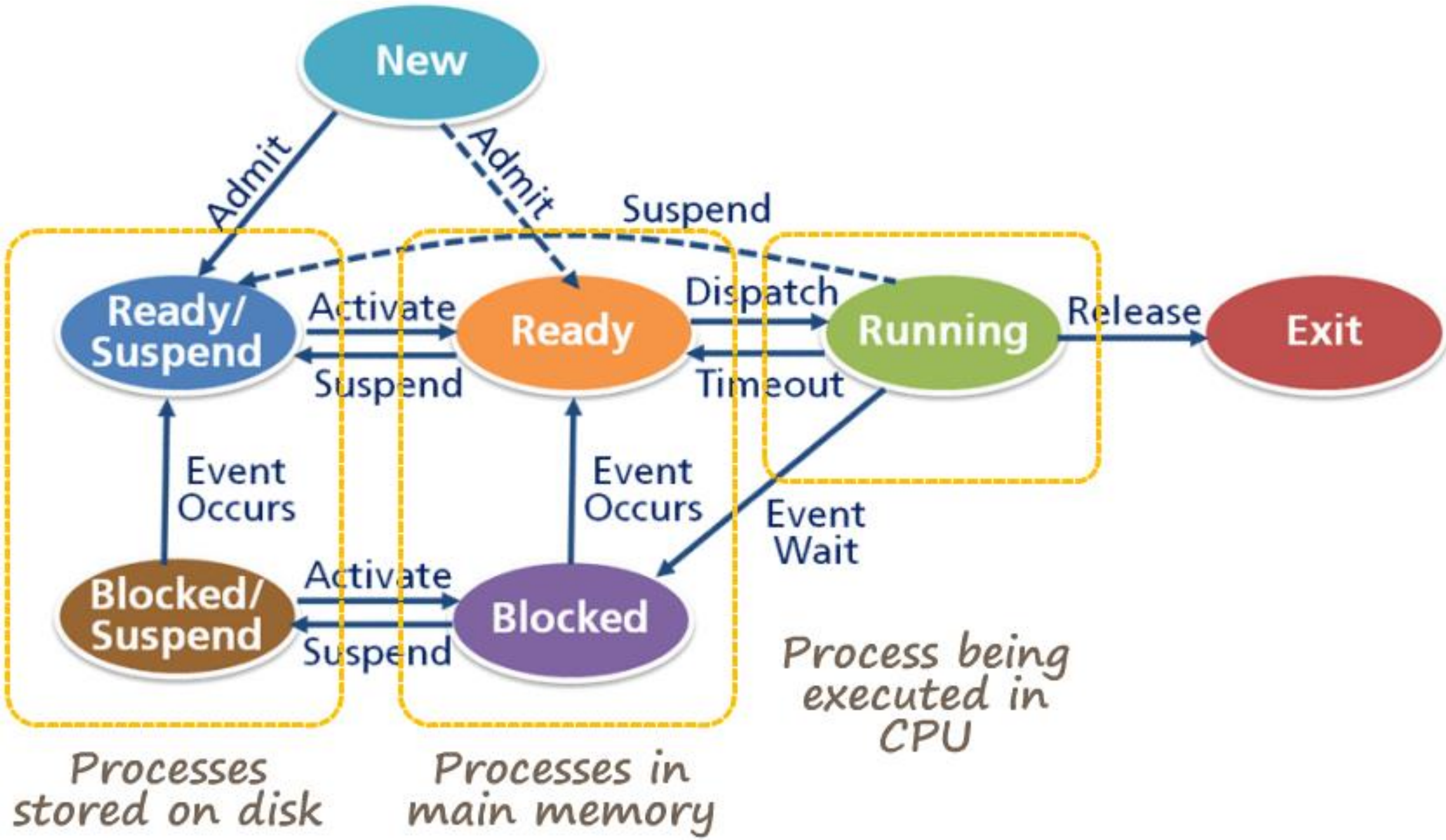
- Use absolute time!

```
timer_t created_timer;
timer_t tabs;
// Set tabs
// …
struct itimerspec new,old;
timer_create(CLOCKID, _ ,  &created_timer);
clock_gettime(CLOCK_REALTIME, &now);

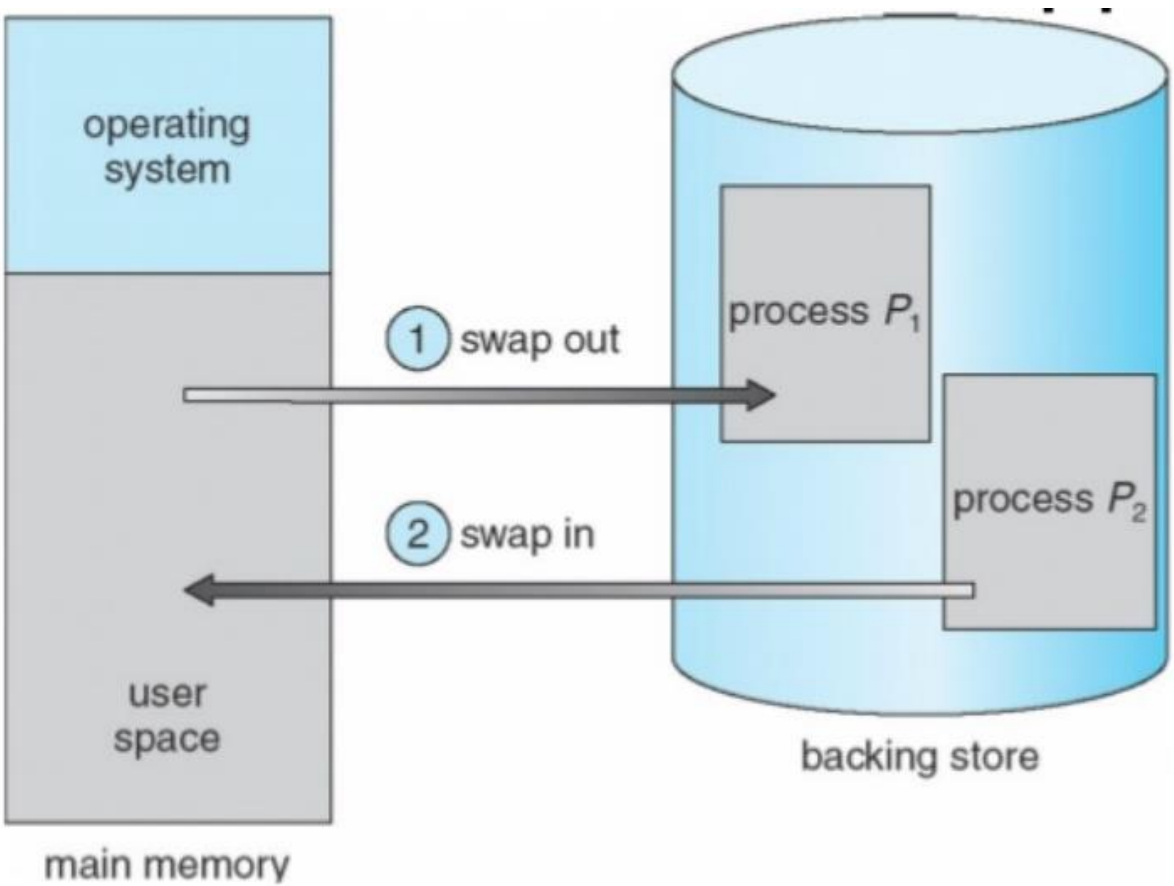
if (now < tabs) { // simplified comparison

      new.it_value.tv_sec=tabs.tv_sec;
      new.it_value.tv_nsec=tabs.tv_nsec;
      new.it_interval.tv_sec=0;// Set interval to 0
      new.it_interval.tv_nsec=0;
      i=timer_settime(created_timer, TIMER_ABSTIME,&new,
      &old);
}
```

POSIX.4 Memory Locking

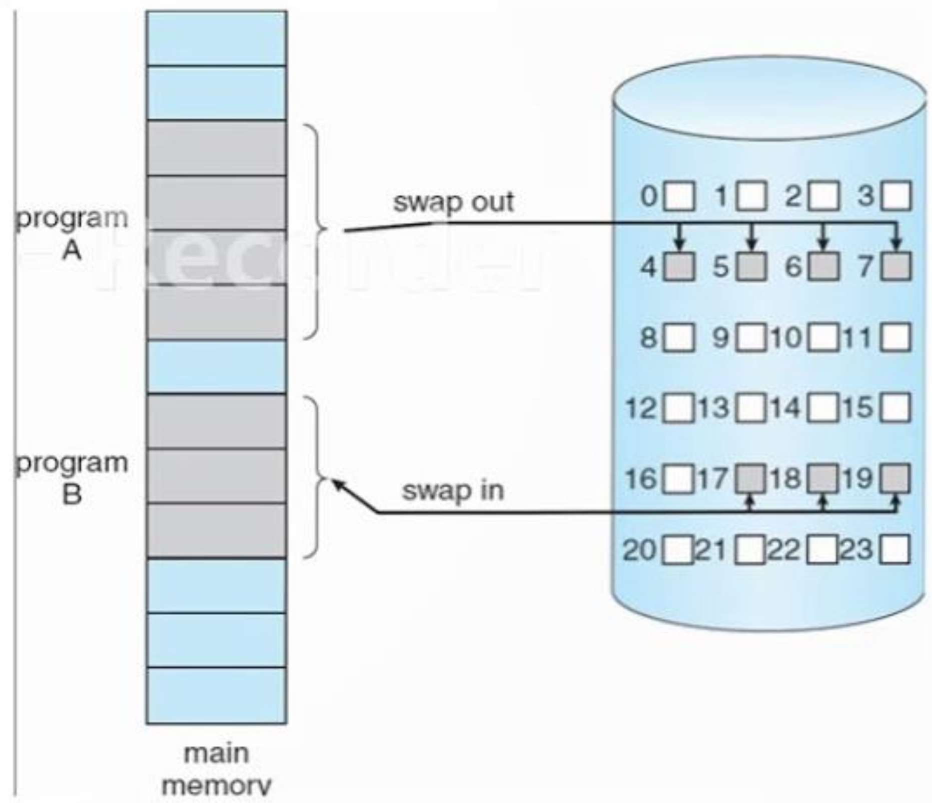


Problem: Swapping of entire Processes



Problem: Demand Paging

- **Demand paging** system loads pages only on demand, not in advance
- Instead of swapping the entire process into memory, the pages or the **lazy swapper** is used
- A lazy swapper brings only the necessary pages into memory



POSIX.4 Memory Locking

```
#include <unistd.h>
/*Main routine */
int main(){
/* Lock all process down */
mlockall(MCL_CURRENT|MCL_FUTURE);

… process code

munlockall();
return 0;
}
```

- Locks currently and future mapped pages belonging to process in memory
  - Locked Memory will vary as process runs
  - Physical memory can be exceeded!