Processes and operating systems

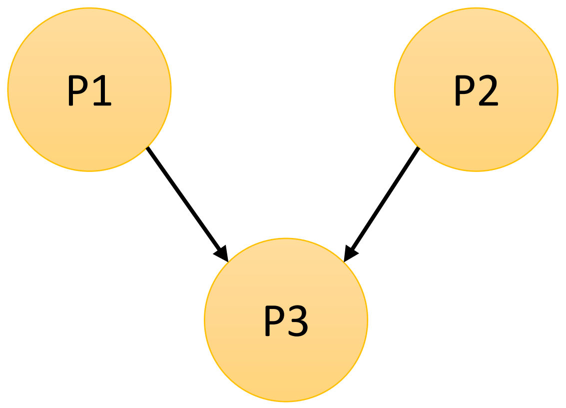
- Multiple tasks and multiple processes.
  - Specifications of process timing.
- Preemptive real-time operating systems.
- Processes and UML.

Reactive systems

- Respond to external events.
  - Engine controller.
  - Seat belt monitor.
- Requires real-time response.
  - System architecture.
  - Program implementation.
- May require a chain reaction among multiple processors.

Tasks and processes

- A task is a functional description of a connected set of operations.
- (Task can also mean a collection of processes.)



- A process is a unique execution of a program.
  - Several copies of a program may run simultaneously or at different times.
- A process has its own state:
  - registers;
  - memory.
- The operating system manages processes.

Why multiple processes?

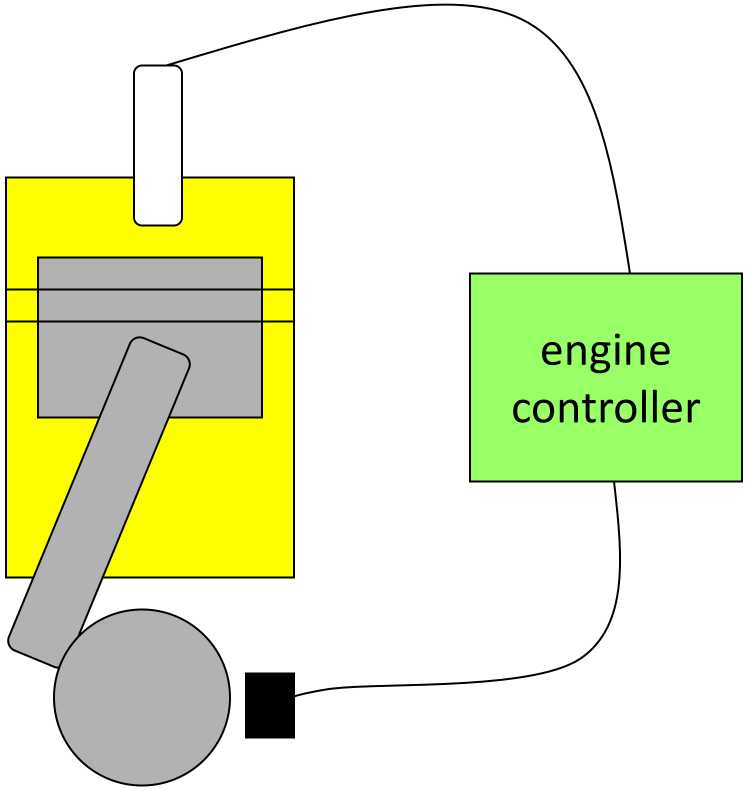
- Multiple tasks means multiple processes.
- Processes help with timing complexity:
  - multiple rates
    - multimedia
    - automotive
  - asynchronous input
    - user interfaces
    - communication systems

Multi-rate systems

- Tasks may be synchronous or asynchronous.
- Synchronous tasks may recur at different rates.
- Processes run at different rates based on computational needs of the tasks.

Example: engine control

- Tasks:
  - spark control
  - crankshaft sensing
  - fuel/air mixture
  - oxygen sensor
  - Kalman filter



Typical rates in engine controllers

Variable	Full range time (ms)	Update period (ms)
Engine spark timing	300	2
Throttle	40	2
Air flow	30	4
Battery voltage	80	4
Fuel flow	250	10
Recycled exhaust gas	500	25
Status switches	100	20
Air temperature	Seconds	400
Barometric pressure	Seconds	1000
Spark (dwell)	10	1
Fuel adjustment	80	8
Carburetor	500	25
Mode actuators	100	100

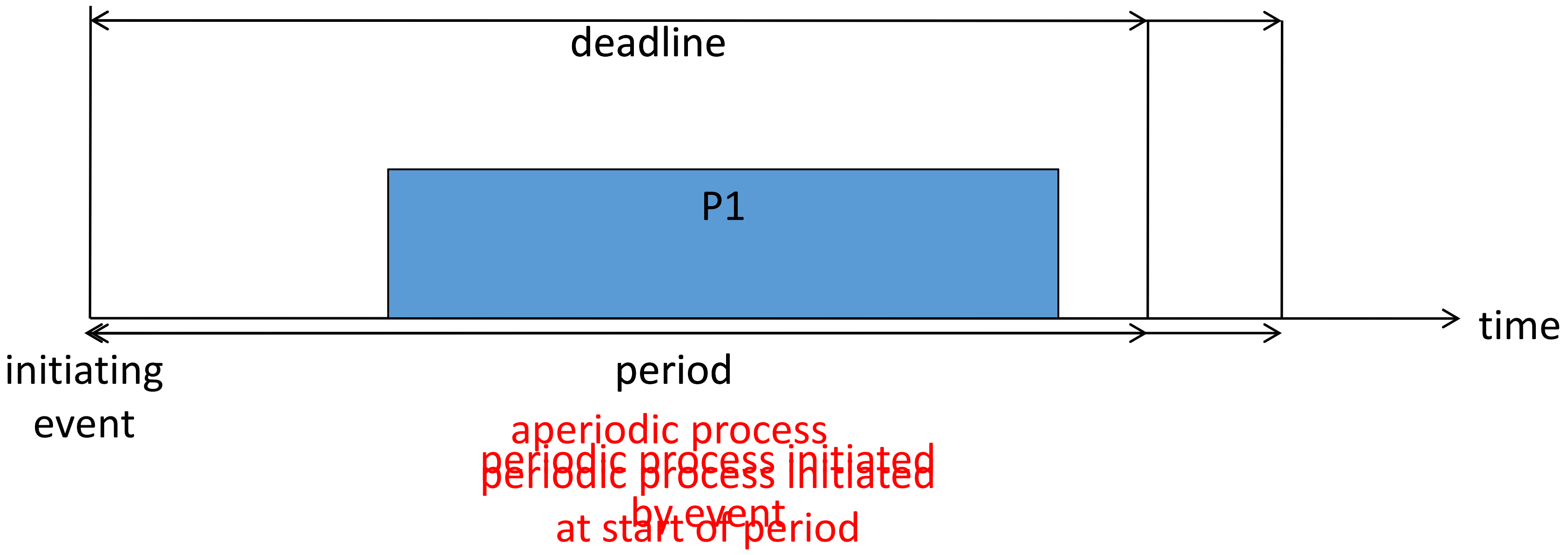
Real-time systems

- Perform a computation to conform to external timing constraints.
- Deadline frequency:
  - Periodic.
  - Aperiodic.
- Deadline type:
  - Hard: failure to meet deadline causes system failure.
  - Soft: failure to meet deadline causes degraded response.
  - Firm: late response is useless but some late responses can be tolerated.

Timing specifications on processes

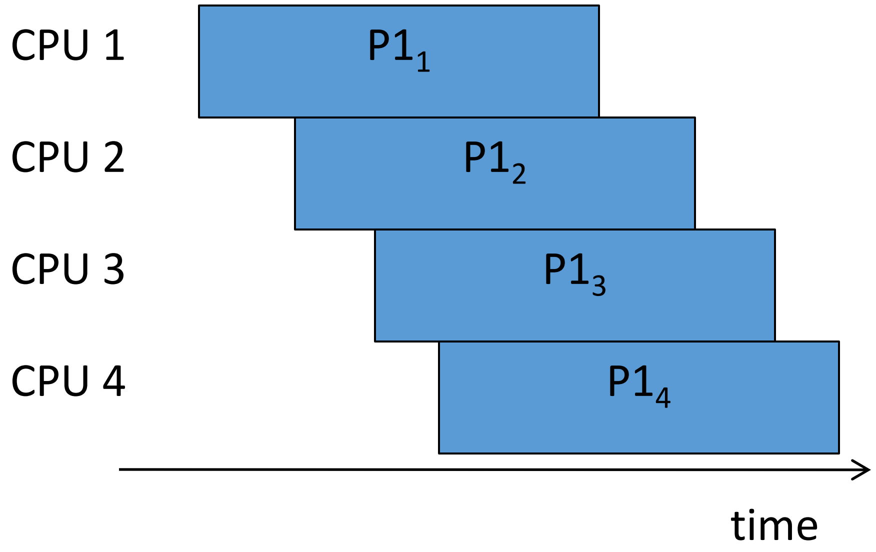
- Release time: time at which process becomes ready.
- Deadline: time at which process must finish.

Release times and deadlines



Rate requirements on processes

- Period: interval between process activations.
- Rate: reciprocal of period.
- Initiation rate may be higher than period---several copies of process run at once.



Timing violations

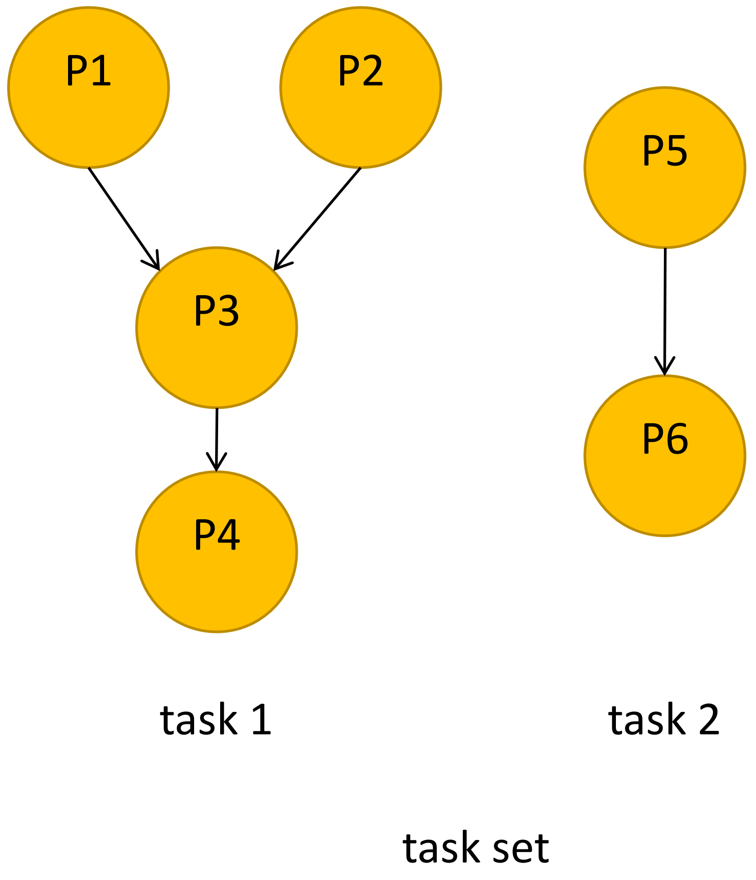
- What happens if a process doesn't finish by its deadline?
  - Hard deadline: system fails if missed.
  - Soft deadline: user may notice, but system doesn't necessarily fail.

Example: Space Shuttle software error

- Space Shuttle's first launch was delayed by a software timing error:
  - Primary control system PASS and backup system BFS.
  - BFS failed to synchronize with PASS.
  - Change to one routine added delay that threw off start time calculation.
  - 1 in 67 chance of timing problem.

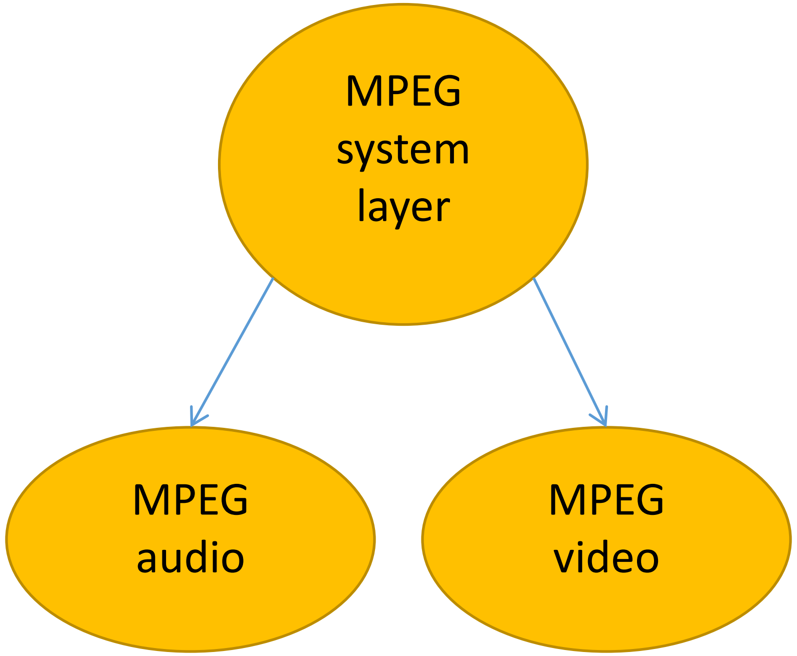
Task graphs

- Tasks may have data dependencies---must execute in certain order.
- Task graph shows data/control dependencies between processes.
- Task: connected set of processes.
- Task set: One or more tasks.



Communication between tasks

- Task graph assumes that all processes in each task run at the same rate, tasks do not communicate.
- In reality, some amount of inter-task communication is necessary.
  - It's hard to require immediate response for multi-rate communication.



Process execution characteristics

- Process execution time T_i.
  - Execution time in absence of preemption.
  - Possible time units: seconds, clock cycles.
  - Worst-case, best-case execution time may be useful in some cases.
- Sources of variation:
  - Data dependencies.
  - Memory system.
  - CPU pipeline.

Utilization

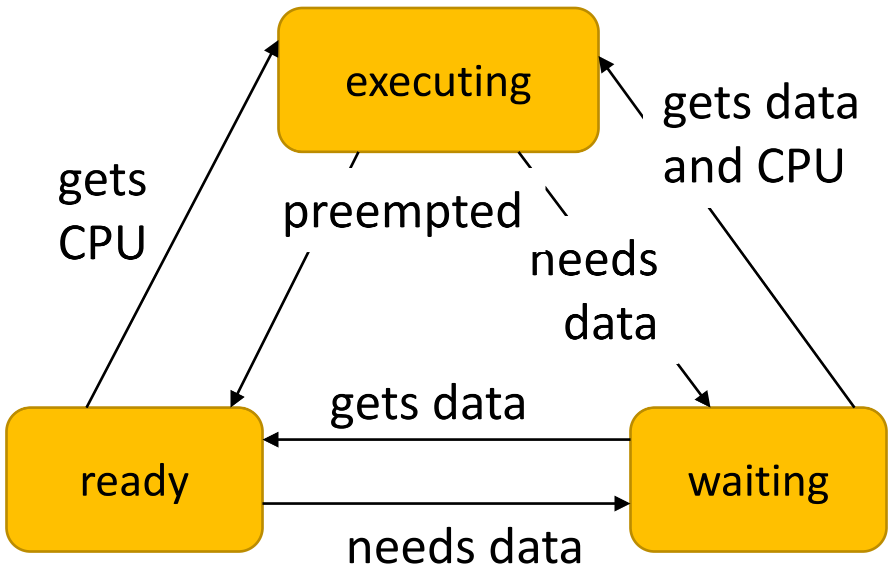
- CPU utilization:
  - Fraction of the CPU that is doing useful work.
  - Often calculated assuming no scheduling overhead.
- Utilization:
  - U = (CPU time for useful work)/ (total available CPU time)

    $= \frac{\sum_{t_1}^{t_2} T(t)}{t_2 - t_1}$

    = T/t

State of a process

- A process can be in one of three states:
  - executing on the CPU;
  - ready to run;
  - waiting for data.

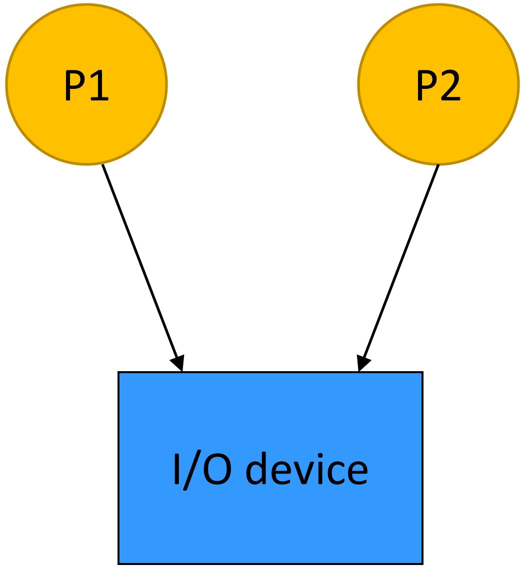


The scheduling problem

- Can we meet all deadlines?
  - Must be able to meet deadlines in all cases.
- How much CPU horsepower do we need to meet our deadlines?

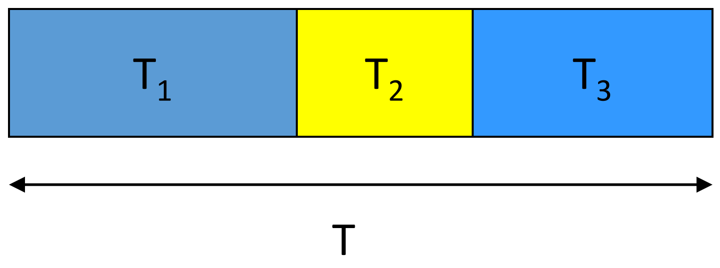
Scheduling feasibility

- Resource constraints make schedulability analysis NP-hard.
  - Must show that the deadlines are met for all timings of resource requests.



Simple processor feasibility

- Assume:
  - No resource conflicts.
  - Constant process execution times.
- Require:
  - $T \geq \Sigma_i T_i$
  - Can't use more than 100% of the CPU.



Hyperperiod

- Hyperperiod: least common multiple (LCM) of the task periods.
- Must look at the hyperperiod schedule to find all task interactions.
- Hyperperiod can be very long if task periods are not chosen carefully.

Hyperperiod example

- Long hyperperiod:
  - P1 7 ms.
  - P2 11 ms.
  - P3 15 ms.
  - LCM = 1155 ms.
- Shorter hyperperiod:
  - P1 8 ms.
  - P2 12 ms.
  - P3 16 ms.
  - LCM = 96 ms.

Simple processor feasibility example

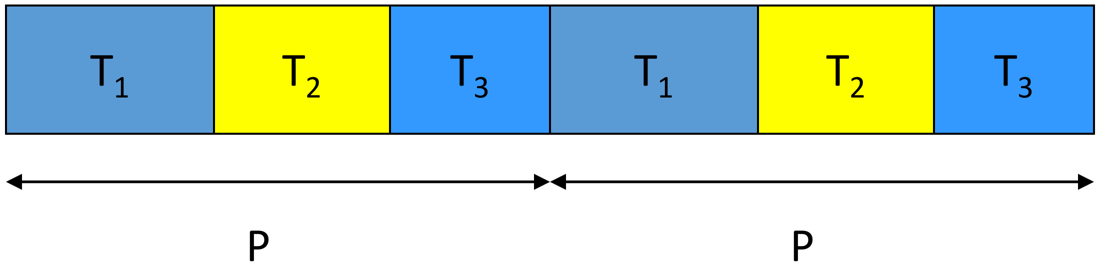
- P1 period 1 ms, CPU time 0.1 ms.
- P2 period 1 ms, CPU time 0.2 ms.
- P3 period 5 ms, CPU time 0.3 ms.

LCM		5.00E-03	
	peirod	CPU time	CPU time/L
P1	1.00E-03	1.00E-04	5.00E-04
P2	1.00E-03	2.00E-04	1.00E-03
P3	5.00E-03	3.00E-04	3.00E-04
	total CPU/LCM		1.80E-03
	utilization		3.60E-01

$$\text{Task execution time} = \left\lceil \frac{LCM}{period} \right\rceil T_i$$

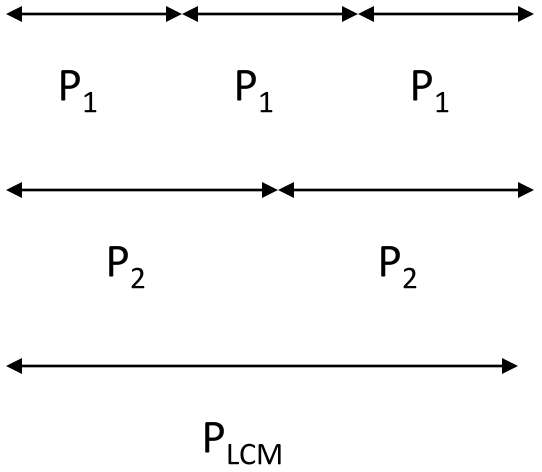
Cyclostatic/TDMA

- Schedule in time slots.
  - Same process activation irrespective of workload.
- Time slots may be equal size or unequal.



TDMA assumptions

- Schedule based on least common multiple (LCM) of the process periods.
- Trivial scheduler -> very small scheduling overhead.



TDMA schedulability

- Always same CPU utilization (assuming constant process execution times).
- Can't handle unexpected loads.
  - Must schedule a time slot for aperiodic events.

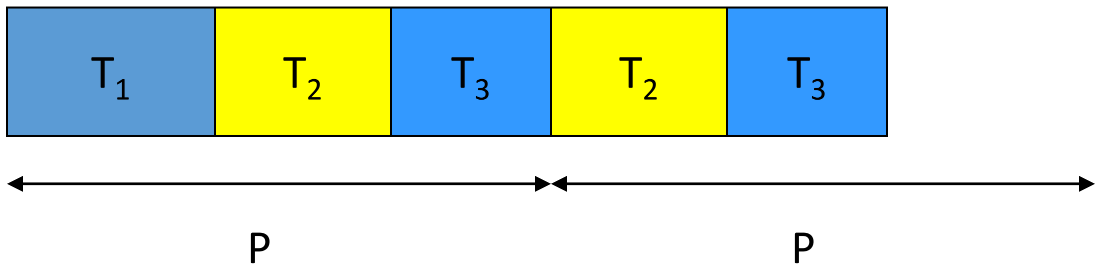
TDMA schedulability example

- TDMA period = 10 ms.
- P1 CPU time 1 ms.
- P2 CPU time 3 ms.
- P3 CPU time 2 ms.
- P4 CPU time 2 ms.

TDMA period		1.00E-02	
	CPU time		
P1	1.00E-03		
P2	3.00E-03		
P3	2.00E-03		
P4	2.00E-03		
total	8.00E-03		
utilization	8.00E-01		

Round-robin

- Schedule process only if ready.
    - Always test processes in the same order.
- Variations:
    - Constant system period.
    - Start round-robin again after finishing a round.



Round-robin assumptions

- Schedule based on least common multiple (LCM) of the process periods.
- Best done with equal time slots for processes.
- Simple scheduler -> low scheduling overhead.
  - Can be implemented in hardware.

Round-robin schedulability

- Can bound maximum CPU load.
  - May leave unused CPU cycles.
- Can be adapted to handle unexpected load.
  - Use time slots at end of period.

Schedulability and overhead

- The scheduling process consumes CPU time.
  - Not all CPU time is available for processes.
- Scheduling overhead must be taken into account for exact schedule.
  - May be ignored if it is a small fraction of total execution time.

Running periodic processes

- Need code to control execution of processes.
- Simplest implementation: process = subroutine.

while loop implementation

- Simplest implementation has one loop.
  - No control over execution timing.

```
while (TRUE) {
  p1();
  p2();
}
```

Timed loop implementation

- Encapuslate set of all processes in a single function that implements the task set,.
- Use timer to control execution of the task.
  - No control over timing of individual processes.

```
void pall(){
  p1();
  p2();
}
```

Multiple timers implementation

- Each task has its own function.
- Each task has its own timer.
  - May not have enough timers to implement all the rates.

```
void pA(){ /* rate A */
   p1();
   p3();
}
void B(){ /* rate B */
   p2();
   p4();
   p5();
}
```

Timer + counter implementation

- Use a software count to divide the timer.
- Only works for clean multiples of the timer period.

```
int p2count = 0;
void pall(){
   p1();
   if (p2count >= 2) {
         p2();
         p2count = 0;
         }
   else p2count++;
   p3();
}
```

Cooperative multitasking in PIC16F887

- Timing is controlled by timer 0.
  - Enabled by T0IE.
- Global variable timer_flag tells main() when timer is done.

```
void interrupt timer_handler() {
    if (T0IE && T0IF) {
    timer_flag = 1;
    T0IF = 0;
}
main() {
    init();
    while (1) {
    if (timer_flag) {
                task1();
                task2();
                task3();
                timer_flag = 0;
                }
    }
}
```

Implementing processes

- All of these implementations are inadequate.
- Need better control over timing.
- Need a better mechanism than subroutines.

Processes and operating systems

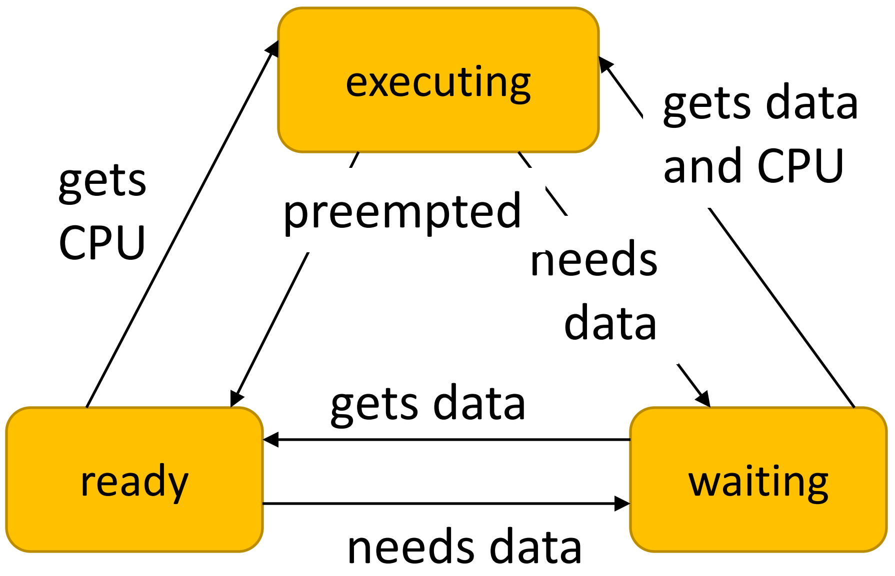
- Real-time operating systems.
- Processes.
- Scheduling policies:
  - RMS;
  - EDF.
- Scheduling modeling assumptions.

Operating systems

- The operating system controls resources:
  - who gets the CPU;
  - when I/O takes place;
  - how much memory is allocated.
- The most important resource is the CPU itself.
  - CPU access controlled by the scheduler.

Process state

- A process can be in one of three states:
  - executing on the CPU;
  - ready to run;
  - waiting for data.



Operating system structure

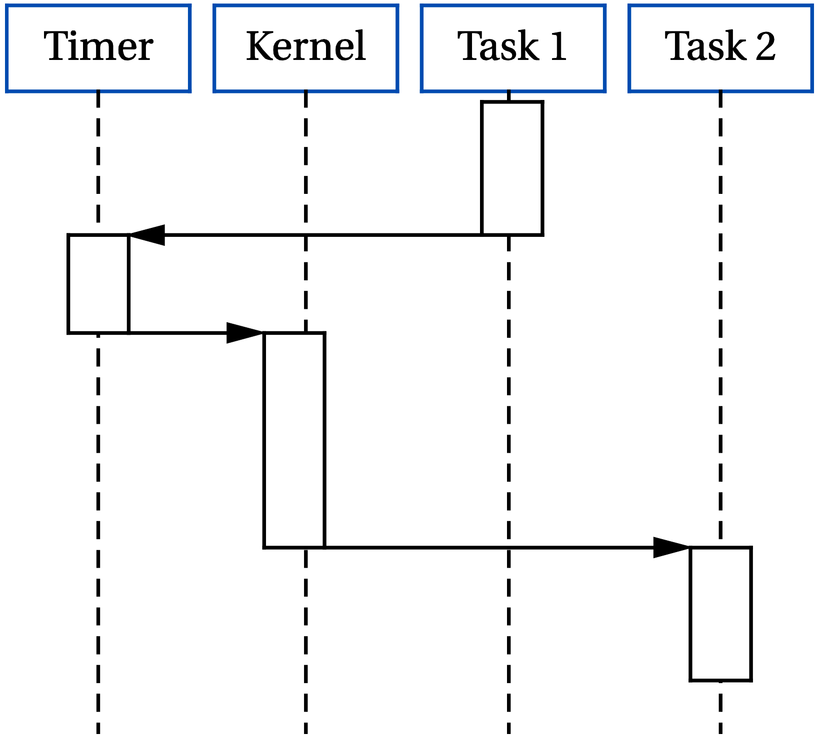
- OS needs to keep track of:
  - process priorities;
  - scheduling state;
  - process activation record.
- Processes may be created:
  - statically before system starts;
  - dynamically during execution.

Embedded vs. general-purpose scheduling

- Workstations try to avoid starving processes of CPU access.
  - Fairness = access to CPU.
- Embedded systems must meet deadlines.
  - Low-priority processes may not run for a long time.

Preemptive scheduling

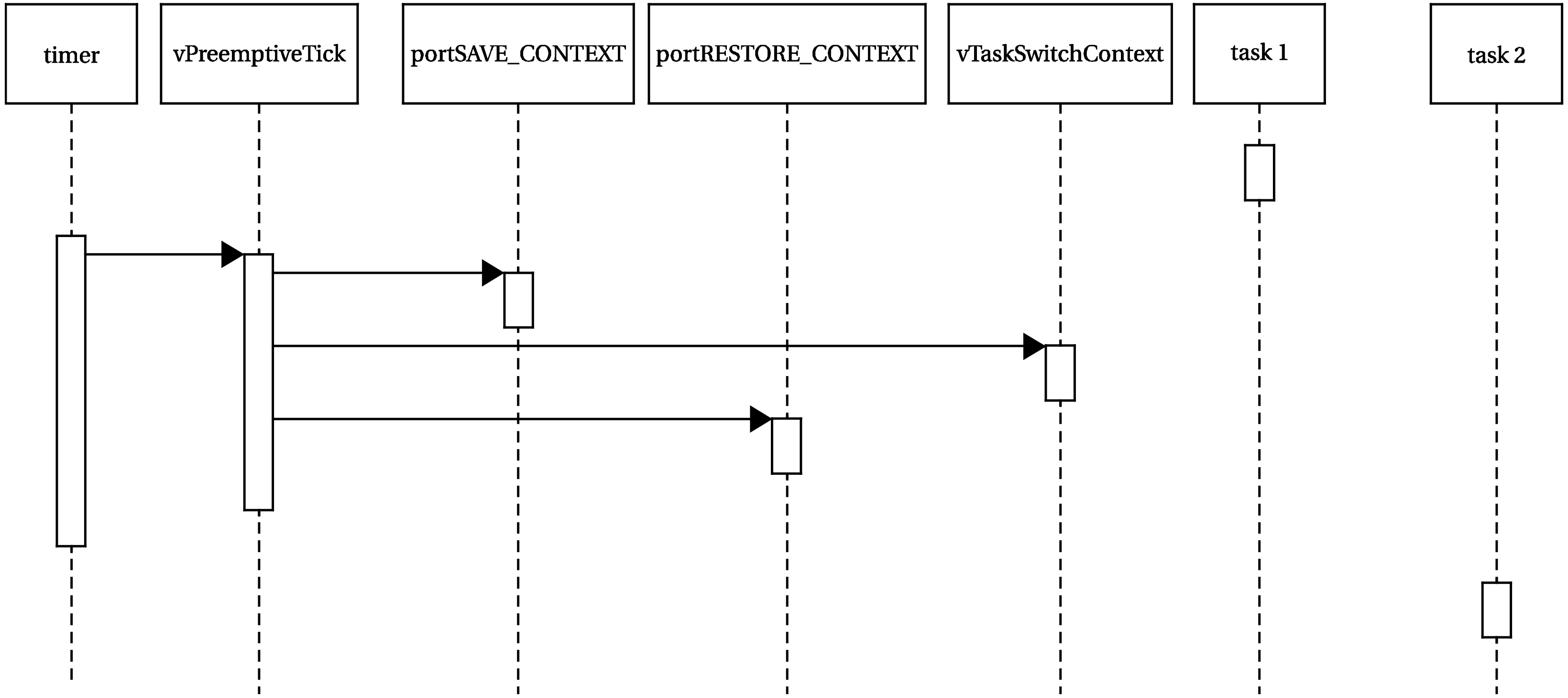
- Timer interrupt gives CPU to kernel.
  - Time quantum is smallest increment of CPU scheduling time.
- Kernel decides what task runs next.
- Kernel performs context switch to new context.



Context switching

- Set of registers that define a process's state is its context.
  - Stored in a record.
- Context switch moves the CPU from one process's context to another.
- Context switching code is usually assembly code.
  - Restoring context is particularly tricky.

freeRTOS.org context switch



freeRTOS.org timer handler

```
void vPreemptiveTick( void )
    {
    /* Save the context of the current task. */
    portSAVE_CONTEXT();
    /* Increment the tick count - this may wake a task. */
    vTaskIncrementTick();
    /* Find the highest priority task that is ready to run. */
    vTaskSwitchContext();
    /* End the interrupt in the AIC. */
    AT91C_BASE_AIC->AIC_EOICR = AT91C_BASE_PITC->PITC_PIVR;;
portRESTORE_CONTEXT();
    }
```

freeRTOS.org save context

```
push r0
in r0, __SREG__
cli
push r0
push r1
clr r1
push r2
```

```
; continue pushing all the registers
push r31
lds r26, pxCurrentTCB
lds r27, pxCurrentTCB + 1
in r0, __SP_L__
st x+, r0
in r0, __SP_H__
st x+, r0
```

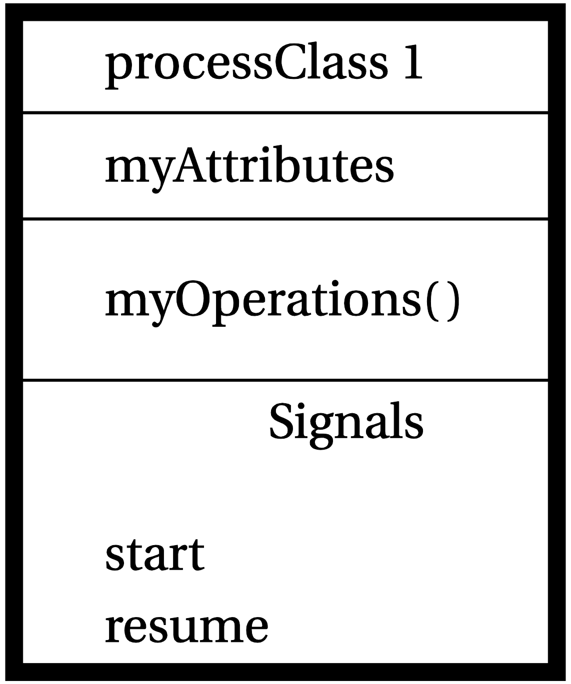
freeRTOS.org restore context

```
lds r26, pxCurrentTCB
lds r27, pxCurrentTCB + 1
ld r28, x+
out __SP_L__, r28
ld r29, x+
out __SP_H__, r29
pop r31
```

```
; pop the registers
pop r1
pop r0
out __SREG__, r0
pop r0
```

Processes in UML

- An active object has an independent thread of control.
- Specified by an active class.



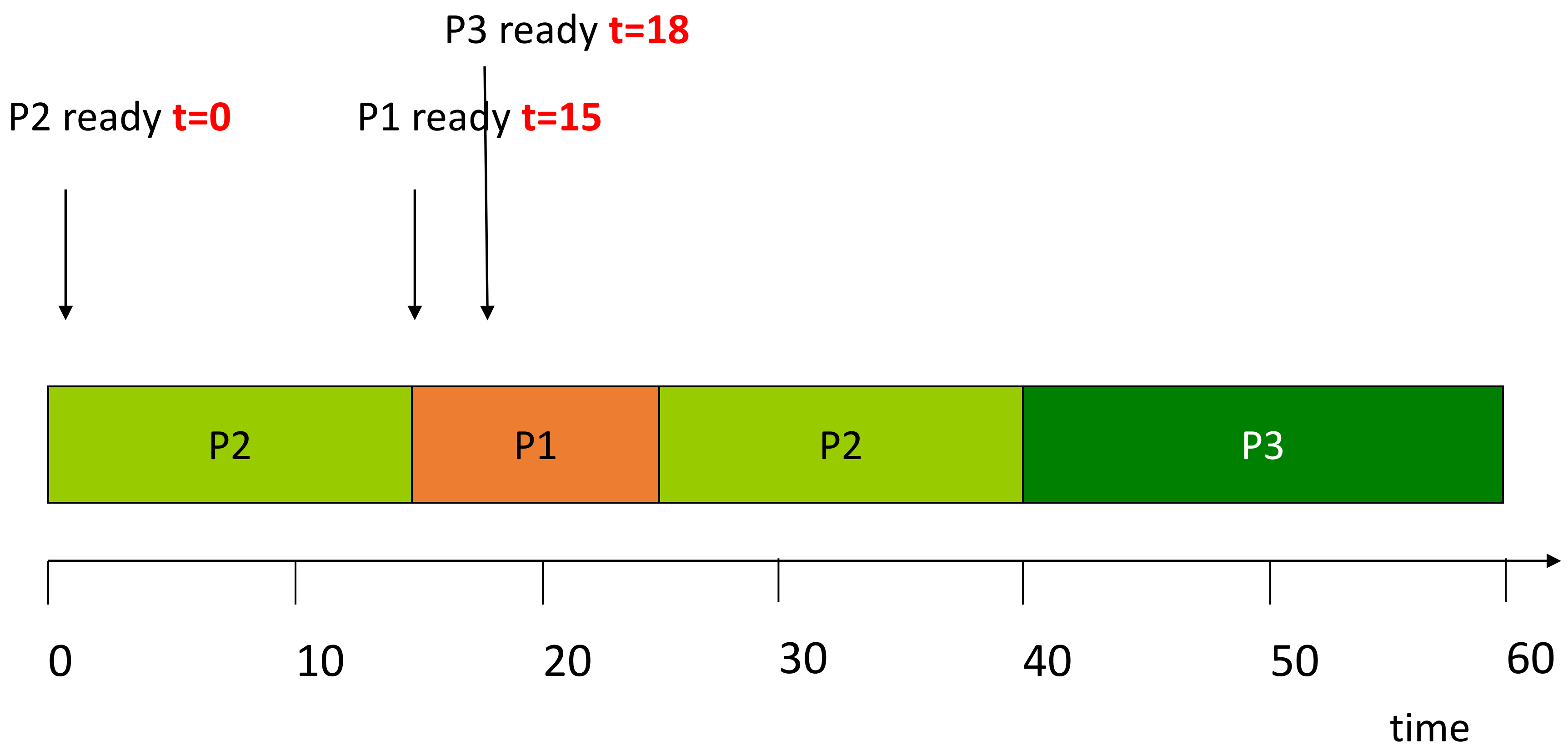
Priority-driven scheduling

- Each process has a priority.
- CPU goes to highest-priority process that is ready.
- Priorities determine scheduling policy:
  - fixed priority;
  - time-varying priorities.

Priority-driven scheduling example

- Rules:
  - each process has a fixed priority (1 highest);
  - highest-priority ready process gets CPU;
  - process continues until done.
- Processes
  - P1: priority 1, execution time 10
  - P2: priority 2, execution time 30
  - P3: priority 3, execution time 20

Priority-driven scheduling example



The scheduling problem

- Can we meet all deadlines?
  - Must be able to meet deadlines in all cases.
- How much CPU horsepower do we need to meet our deadlines?

Process initiation disciplines

- Periodic process: executes on (almost) every period.
- Aperiodic process: executes on demand.
- Analyzing aperiodic process sets is harder---must consider worst-case combinations of process activations.

Timing requirements on processes

- Period: interval between process activations.
- Initiation interval: reciprocal of period.
- Initiation time: time at which process becomes ready.
- Deadline: time at which process must finish.

Timing violations

- What happens if a process doesn't finish by its deadline?
  - Hard deadline: system fails if missed.
  - Soft deadline: user may notice, but system doesn't necessarily fail.

Scheduling metrics

- How do we evaluate a scheduling policy:
  - Ability to satisfy all deadlines.
  - CPU utilization---percentage of time devoted to useful work.
  - Scheduling overhead---time required to make scheduling decision.

Rate monotonic scheduling

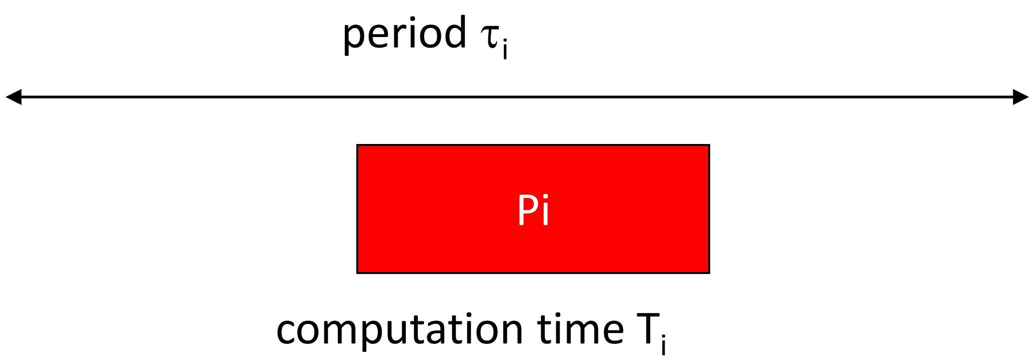
- RMS (Liu and Layland): widely-used, analyzable scheduling policy.
- Analysis is known as Rate Monotonic Analysis (RMA).

RMA model

- All process run on single CPU.
- Zero context switch time.
- No data dependencies between processes.
- Process execution time is constant.
- Deadline is at end of period.
- Highest-priority ready process runs.

Process parameters

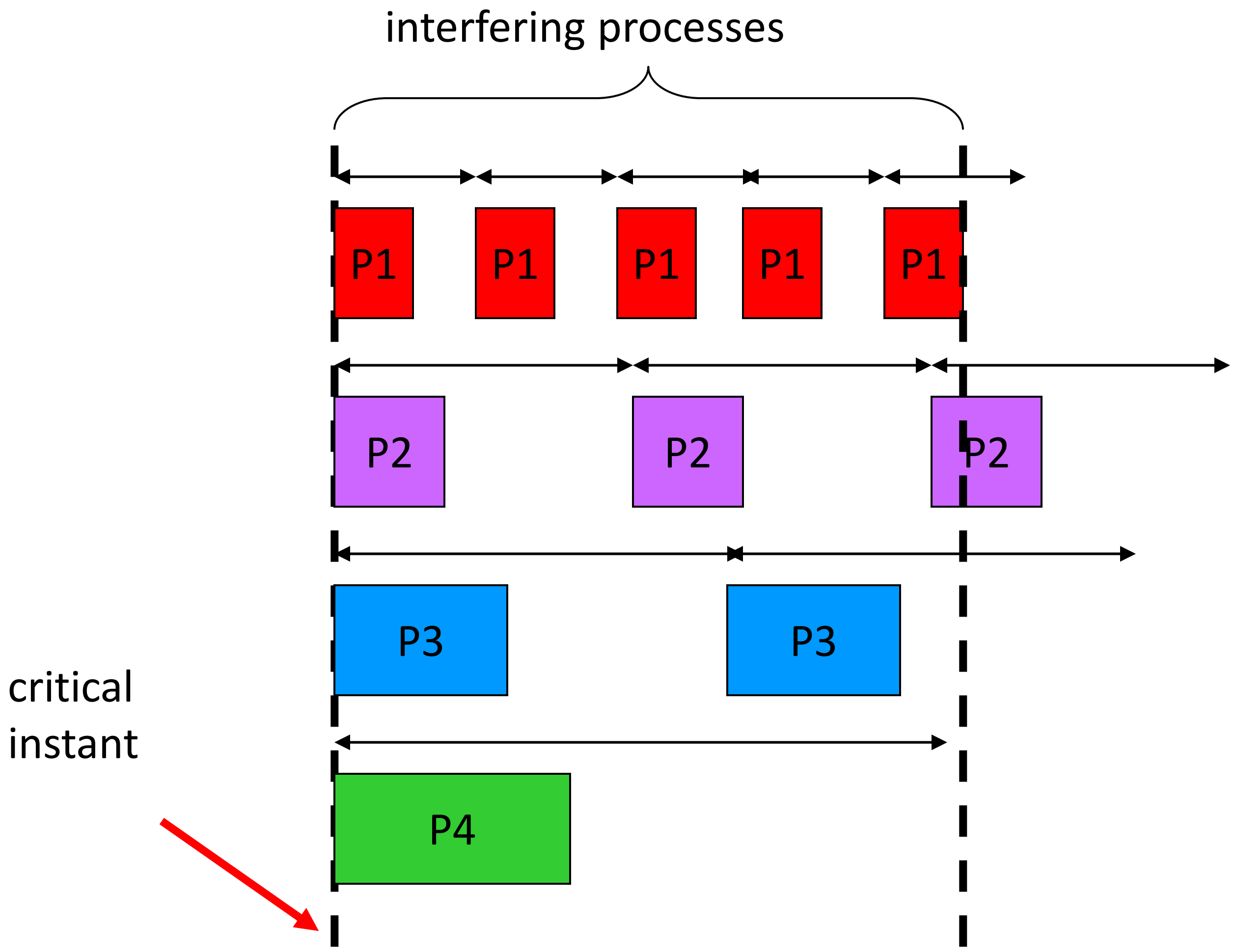
- T_i is computation time of process i; τ_i is period of process i.



Rate-monotonic analysis

- Response time: time required to finish process.
- Critical instant: scheduling state that gives worst response time.
- Critical instant occurs when all higher-priority processes are ready to execute.

Critical instant



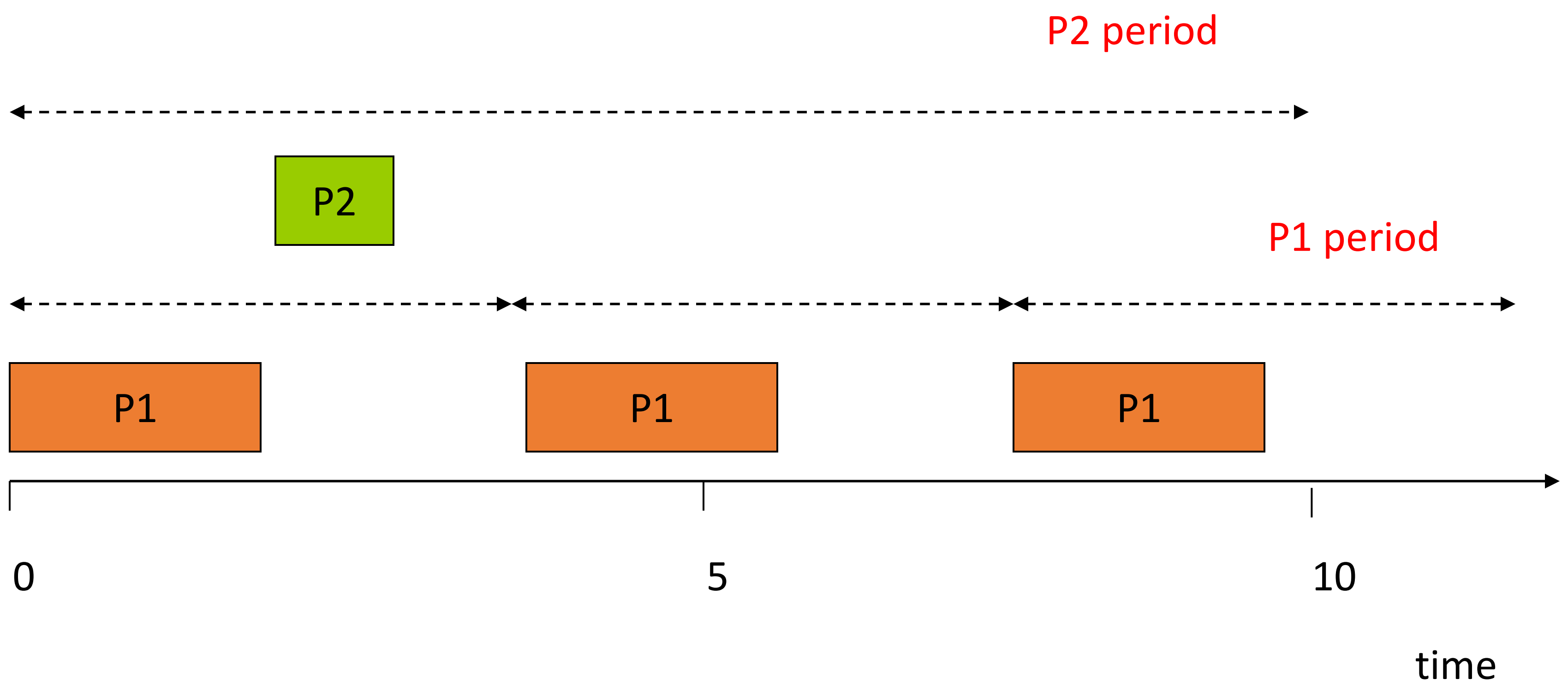
RMS priorities

- Optimal (fixed) priority assignment:
  - shortest-period process gets highest priority;
  - priority inversely proportional to period;
  - break ties arbitrarily.
- No fixed-priority scheme does better.

RMS optimality

- Consider two processes with $\tau_1 < \tau_2$.
- Case 1: If P1 (shorter period) has higher priority, then worst case is over P2's period to execute P2 once and P1 as many times as required:
  - $\left\lfloor \frac{\tau_2}{\tau_1} \right\rfloor T_1 \quad + T_2 \leq \tau_2$
- Case 2: If P2 (longer period) has higher priority, then worst case is to execute all of P2 and all of P1 in one of P1's periods:
  - $T_1 + T_2 \leq \tau_1$
- In some circumstances, second inequality cannot be satisfied but first can be.
- In some circumstances, first inequality can be satisfied but second cannot be.
- Therefore, it is always better to give the process with a shorter period the higher priority.
- Use induction to generalize for $3 \ldots n$ processes.

RMS example



RMS CPU utilization

- Utilization for n processes is
  - $\sum_i T_i / \tau_i$
- Given *m* tasks and ratio between any two periods less than 2:
  - $U = m\left(2^{1/m} - 1\right)$
- As number of tasks approaches infinity, maximum utilization approaches 69%.

RMS CPU utilization, cont'd.

- RMS may not be able to use 100% of CPU, even with zero context switch overhead.
- Must keep idle cycles available to handle worst-case scenario.
- However, RMS guarantees all processes will always meet their deadlines.

RMS implementation

- Efficient implementation:
  - scan processes;
  - choose highest-priority active process.

Earliest-deadline-first scheduling

- EDF: dynamic priority scheduling scheme.
- Process closest to its deadline has highest priority.
- Requires recalculating processes at every timer interrupt.

EDF analysis

- EDF can use 100% of CPU.
- But EDF may fail to miss a deadline.

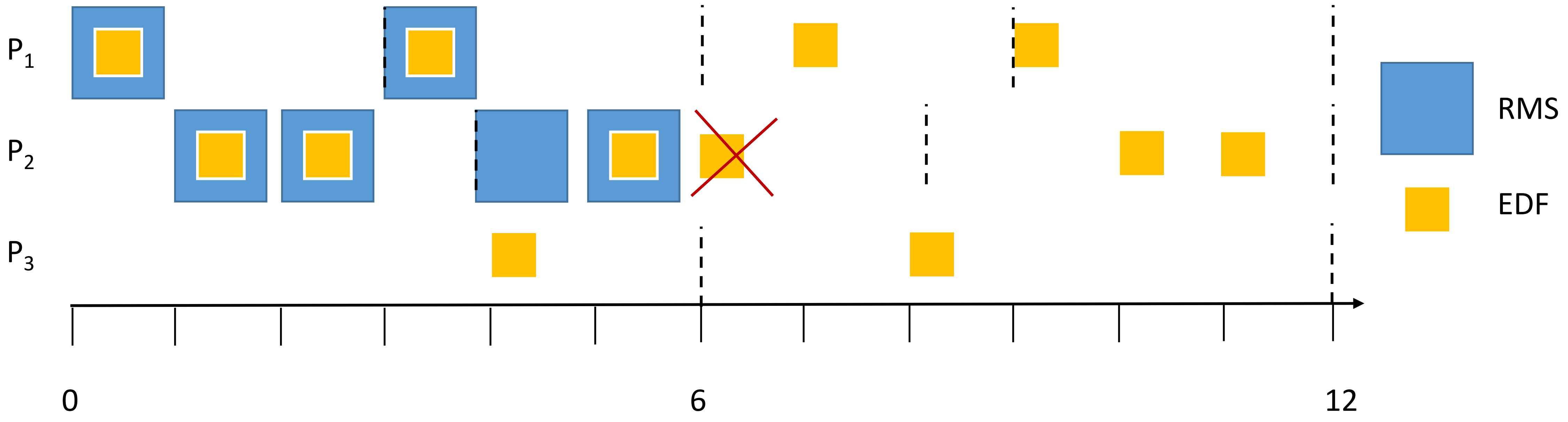
EDF implementation

- On each timer interrupt:
  - compute time to deadline;
  - choose process closest to deadline.
- Increasingly used in high-performance embedded systems to provide higher CPU utilization.
  - Must verify that schedule is not overloaded.

RMS vs. EDF

	C	T
P1	1	3
P2	2	4
P3	1	6

RMS vs. EDF



Fixing scheduling problems

- What if your set of processes is unschedulable?
  - Change deadlines in requirements.
  - Reduce execution times of processes.
  - Get a faster CPU.

Race condition in shared memory

- Problem when two CPUs try to write the same location:
  - CPU 1 reads flag and sees 0.
  - CPU 2 reads flag and sees 0.
  - CPU 1 sets flag to one and writes location.
  - CPU 2 sets flag to one and overwrites location.

Atomic test-and-set

- Problem can be solved with an atomic test-and-set:
  - single bus operation reads memory location, tests it, writes it.
- ARM test-and-set provided by SWP:

```
        ADR r0,SEMAPHORE
        LDR r1,#1
GETFLAG SWP r1,r1,[r0]
        BNZ GETFLAG
```

Semaphores

- Semaphore: OS primitive for controlling access to critical regions.
- Protocol:
  - Get access to semaphore with P().
  - Perform critical region operations.
  - Release semaphore with V().

Critical regions

- Critical region: section of code that cannot be interrupted by another process.
- Examples:
  - writing shared memory;
  - accessing I/O device.

Priority inversion

- Priority inversion: low-priority process keeps high-priority process from running.
- Improper use of system resources can cause scheduling problems:
  - Low-priority process grabs I/O device.
  - High-priority device needs I/O device, but can't get it until low-priority process is done.
- Can cause deadlock.

Solving priority inversion

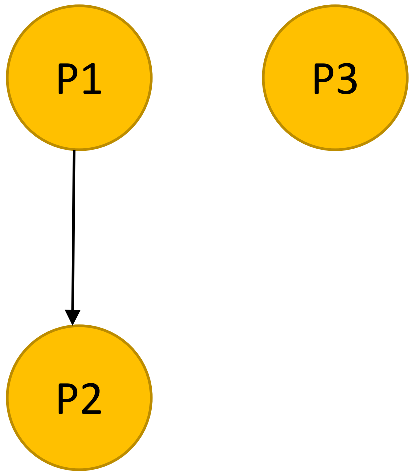
- Give priorities to system resources.
- Have process inherit the priority of a resource that it requests.
  - Low-priority process inherits priority of device if higher.

Scheduling for low power

- EDF with DVFS:
  - First set the clock speed to meet the performance goal in the critical interval.
  - Set clock speed for less-critical intervals in order of importance.
- RMS with DVFS is NP-complete.
- RMS/EDF with race-to-dark is currently handled with heuristics.

Data dependencies

- Data dependencies allow us to improve utilization.
  - Restrict combination of processes that can run simultaneously.
- P1 and P2 can't run simultaneously.
- P3 can preempt P1 or P2 but not both.



Context-switching time

- Non-zero context switch time can push limits of a tight schedule.
- Hard to calculate effects---depends on order of context switches.
- In practice, OS context switch overhead is small (hundreds of clock cycles) relative to many common task periods (ms – μs).

Processes and operating systems

- Interprocess communication.
- Evaluating RTOS performance.
- Example---POSIX.

Interprocess communication

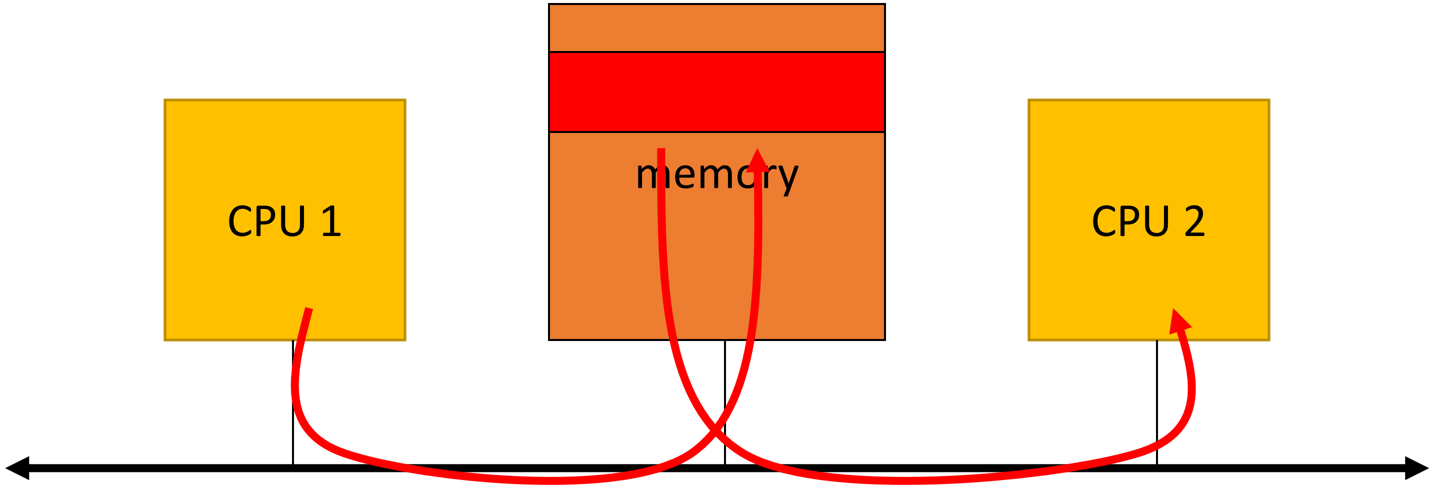
- Interprocess communication (IPC): OS provides mechanisms so that processes can pass data.
- Two types of semantics:
  - blocking: sending process waits for response;
  - non-blocking: sending process continues.

IPC styles

- Shared memory:
  - processes have some memory in common;
  - must cooperate to avoid destroying/missing messages.
- Message passing:
  - processes send messages along a communication channel---no common address space.

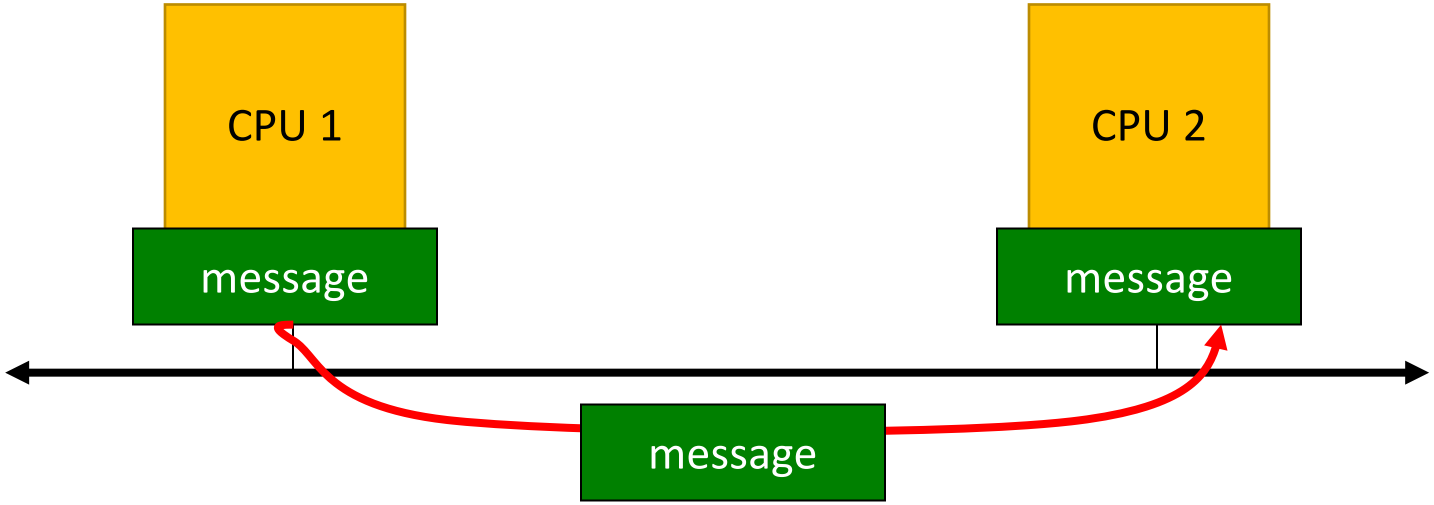
Shared memory

- Shared memory on a bus:



Message passing

- Message passing on a network:



freeRTOS.org queues

- Queues can be used to pass messages.
- Operating system manages queues.

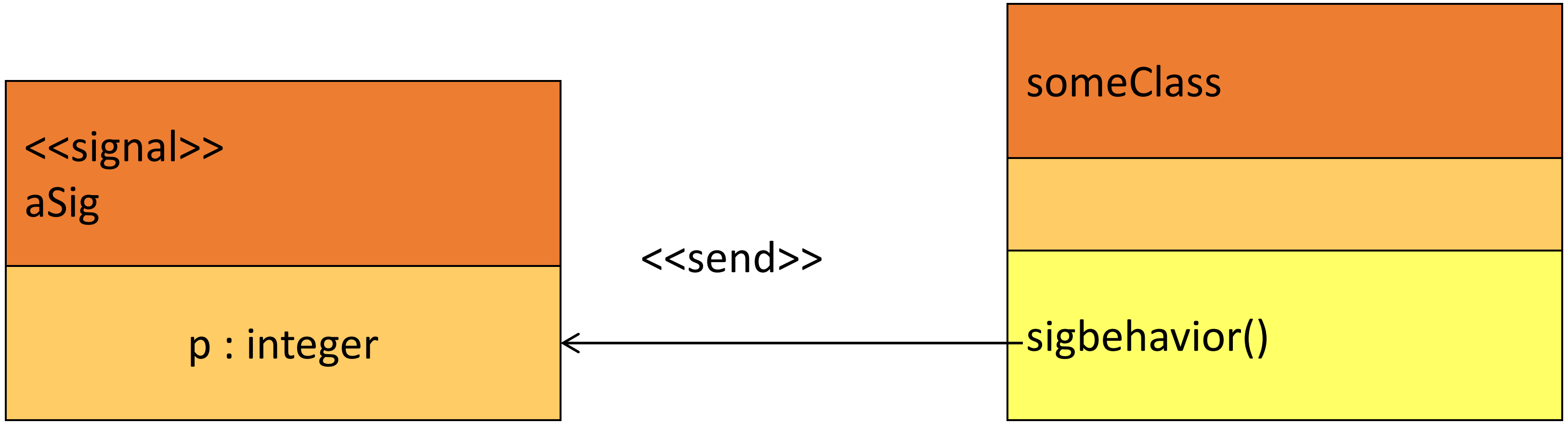
```
xQueueHandle q1;

q1 = xQueueCreate( MAX_SIZE,
   sizeof(msg_record));

if (q1 == 0) /* error */

xQueueSend(q1,(void *)msg,(portTickType)0);
   /* queue, message to send, final parameter
   controls timeout */

if (xQueueReceive(q2,&(in_msg),0); /* queue,
   message received, timeout */
```

Signals

- Similar to a software interrupt.
- Changes flow of control but does not pass parameters.
  - May be typed to allow several types of signals.
  - Unix ^c sends kill signal to process.

Signals in UML

- More general than Unix signal---may carry arbitrary data:



Mailbox

- Fixed memory or register used for interprocess communication.
- May be implemented directly in hardware or by RTOS.

```
void post(message *msg) {
    P(mailbox.sem);
    copy(mailbox.data,msg);
    mailbox.flag = TRUE
    V(mailbox.sem);
    }
boolean pickup(message *msg) {
    boolean pickup = FALSE;
    P(mailbox.sem);
    pickup = mailbox.flag;
    mailbox.flag = FALSE;
    copy(msg, mailbox.data);
    V(mailbox.sem);
    return(pickup);
    }
```

RTOS performance

- Assumptions:
  - Context switch takes zero time.
  - Interrupts do not interfere with scheduling.
  - Process execution time is known and fixed.
  - Process interaction times do not interact.

Evaluating operating system performance

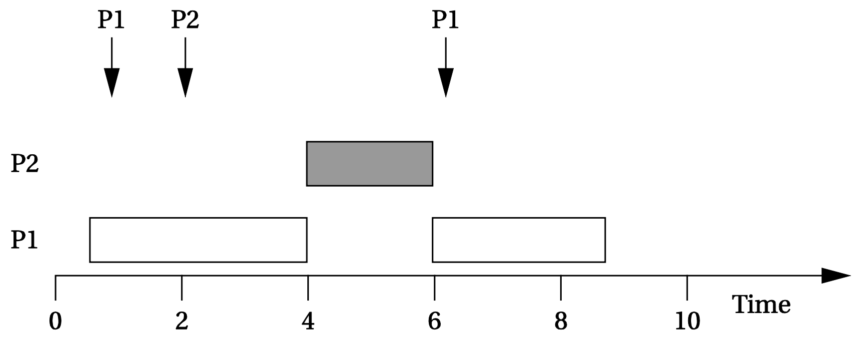
- Context-switching time.
- Interrupt latency and critical sections.
- Interrupt priorities and interrupt latency.
- RTOS performance evaluation.
- Caches and performance

Context-switching time

- Non-zero context switch time can push limits of a tight schedule.
- Hard to calculate effects---depends on order of context switches.
- In practice, OS context switch overhead is small (hundreds of clock cycles) relative to many common task periods (ms – μs).

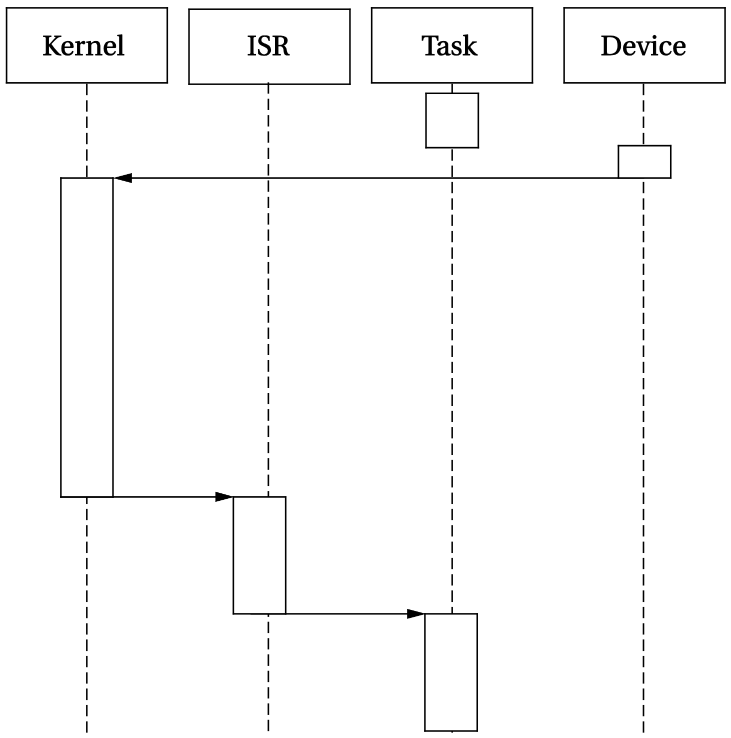
Scheduling and context switch overhead

Process	Execution time	deadline
P1	3	5
P2	3	10



With context switch overhead of 1, no feasible schedule.
2TP1 + TP2 = 2*(1+3)+(1+3)=12

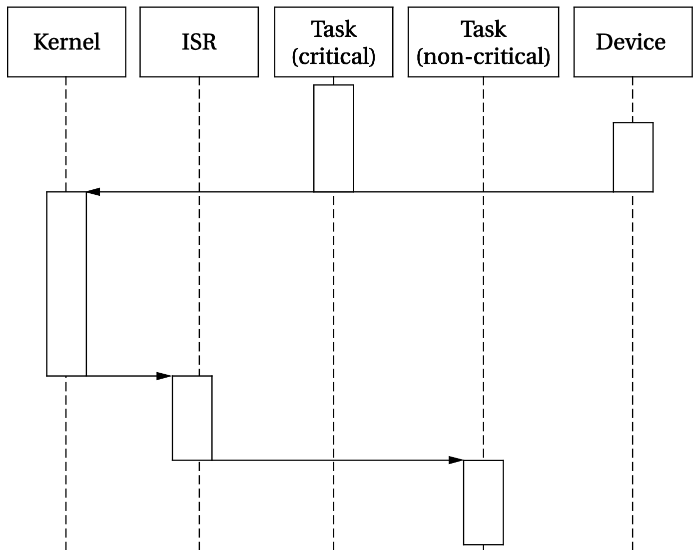
Interrupt latency



- Interrupts override process priorities.
- Interrupt handling latency has non-zero hardware latency.
- Interrupt service routine (ISR) takes time to execute.

Interrupt latency in critical sections

- Interrupts are turned off in critical section.
- Long critical sections add software delays to interrupt latency.
- General-purpose operating systems may have long critical sections.



Interrupt handling architecture

- Use two levels of service:
  - Interrupt service handler (ISH) is called at interrupt, provides minimal functions.
  - Interrupt service routine (ISR) is process invoked by ISH, performs most of the device handling.

RTOS simulation

- Some RTOSs provide scheduling simulators.
- Schedule a mix of processes using I/O traces.

Process execution time

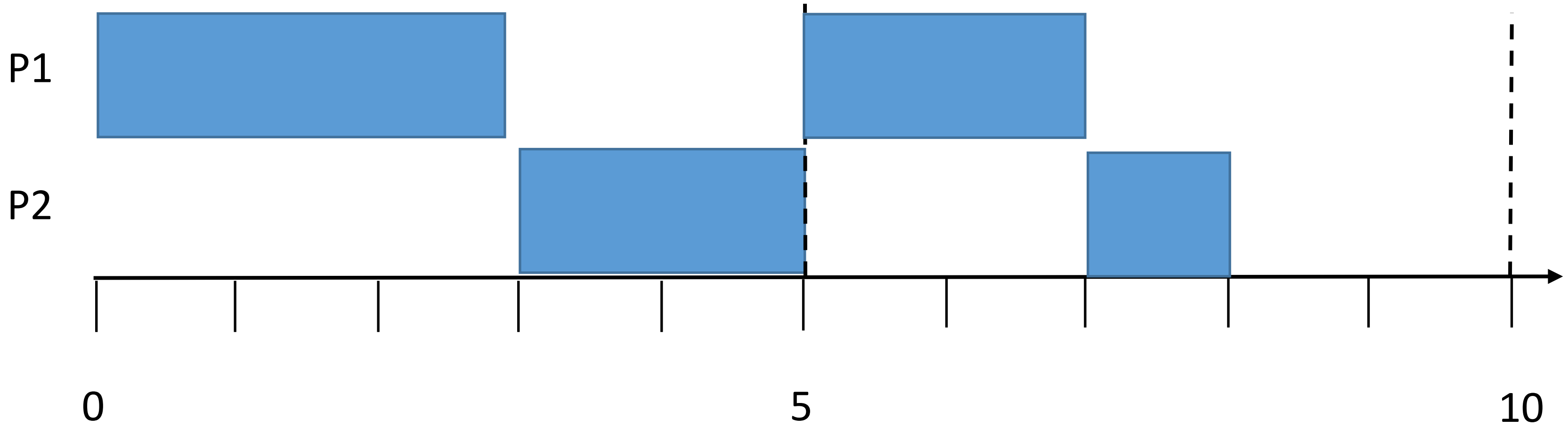
- Process execution time is not constant.
- Extra CPU time can be good.
- Extra CPU time can also be bad:
  - Next process runs earlier, causing new preemption.

Processes and caches

- Processes can cause additional caching problems.
  - Even if individual processes are well-behaved, processes may interfere with each other.
- Worst-case execution time with bad behavior is usually much worse than execution time with good cache behavior.

Effects of scheduling on the cache

Process	WCET	Avg. CPU time
P1	3	2
P2	2	1



Unix

- Unix developed in 1960's at Bell Laboratories to support text processing.
- POSIX is a standard version of Unix.
- Linux is an open-source POSIX-compliant operating system.
  - Linux versions have been developed to improve real-time responsiveness.

POSIX process creation

- fork() makes two copies of executing process.
- Child process identifies itself and overlays new code.

```
if (childid == 0) {
   /* must be child */
   execv("mychild",childargs);
   perror("execv");
   exit(1);
   }
else { /* is the parent */
   parent_stuff();
   wait(&cstatus);
   exit(0);
   }
```

POSIX real-time scheduling

- Processes may run under different scheduling policies.
- _POSIX_PRIORITY_SCHEDULING resource supports real-time scheduling.
- SCHED_FIFO supports RMS.

```
int i, my_process_id;
struct sched_param my_sched_params;
…
i =
    sched_setscheduler(my_process_id,SCHED_FI
    FO,&sched_params);
```

POSIX interprocess communication

- Supports counting semaphores in _POSIX_SEMAPHORES.
- Supports shared memory.

```
i = sem_wait(my_semaphore); /* P */
/* do useful work */
i = sem_post(my_semaphore); /* V */
/* sem_trywait tests without blocking */
i = sem_trywait(my_semaphore);
```

POSIX pipes

- Pipes directly connect programs.
- pipe() function creates a pipe to talk to a child before the child is created.

```
if (pipe(pipe_ends) < 0) {
    perror("pipe");
    break;
    }
childid = fork();
if (childid == 0) {
    childargs[0] = pipe_ends[1];
    execv("mychild",childargs);
    perror("execv");
    exit(1);
    }
else { ... }
```

POSIX message queues

- Supports message queues under _POSIX_MESSAGE_PASSING
- mq_open() creates named queue.

```
myq = mq_open("/q1",O_CREAT |
   RDWR,S_IRWXU,&mq_attr);
…
if (mq_send(myq,data,len,priority) < 0) { /*
   error */ }
nbytes =
   mq_receive(myq,rcvbuf,MAXLEN,&prio);
```

Summary

- A process is a single thread of execution.
- Preemption is the act of changing the CPU's execution from one process to another.
- A scheduling policy is a set of rules that determines the process to run.
- Rate-monotonic scheduling is a simple but powerful scheduling policy.
- Interprocess communication mechanisms allow data to be passed reliably between processes.
- Scheduling analysis often ignores certain real-world effects. Cache interactions between processes are the most important effects to consider when designing a system.