Operating systems

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

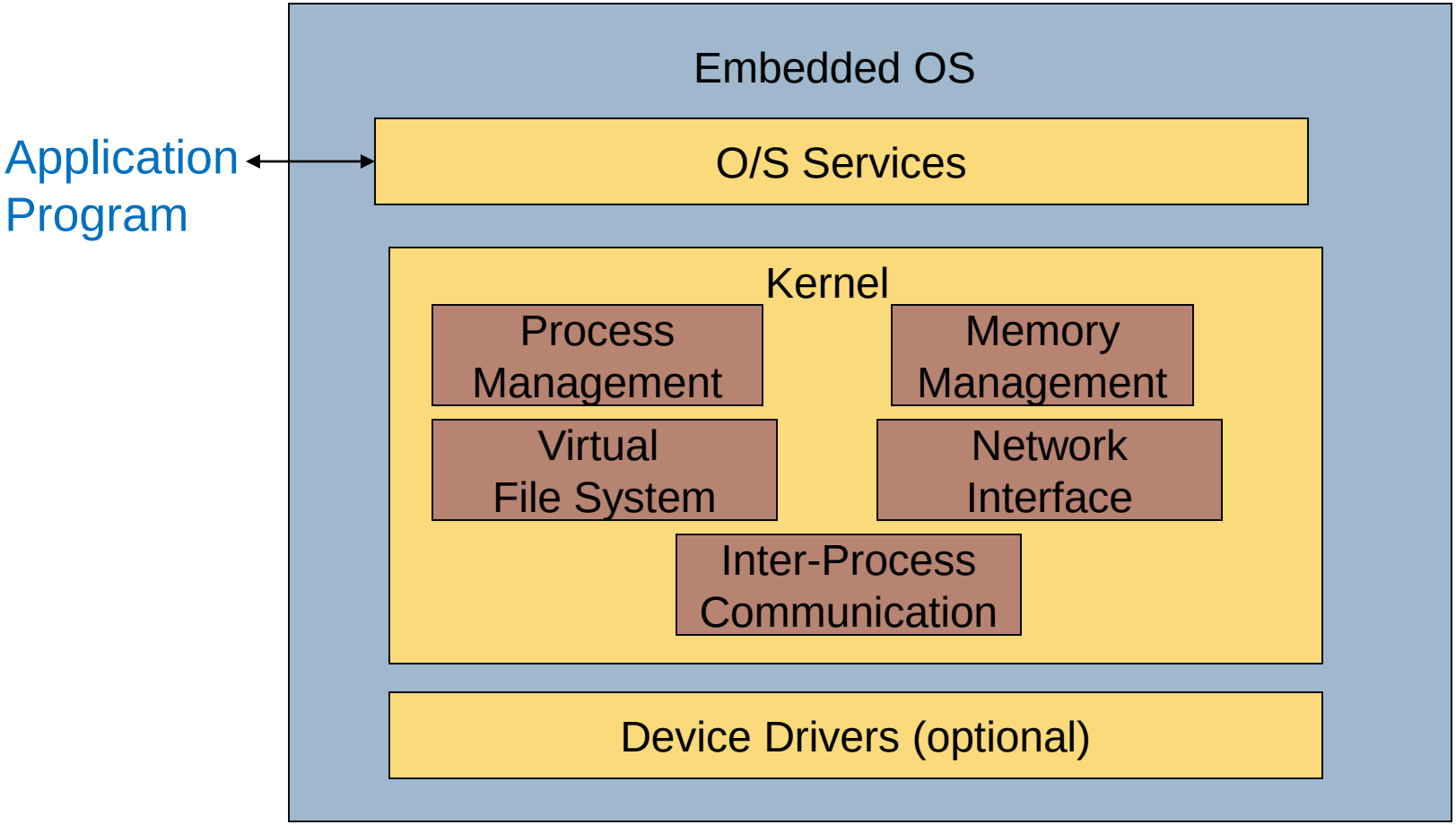
Embedded vs. general-purpose scheduling

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

Real-time operating system (RTOS) features

- Task scheduling
  - Priority, time-slice, fixed ordering, etc.
  - Meet real-time requirements
- Inter-task communication
- Task synchronization & mutual exclusion
  - Coordinate operations
  - Protect tasks from each other
- Memory management
- Scalability
  - Library of plug-ins at compile time to minimize RTOS size
  - Other features: Date/time, File system, Networking, Security

General OS model (Linux-like)



Commercial RTOSs (partial)

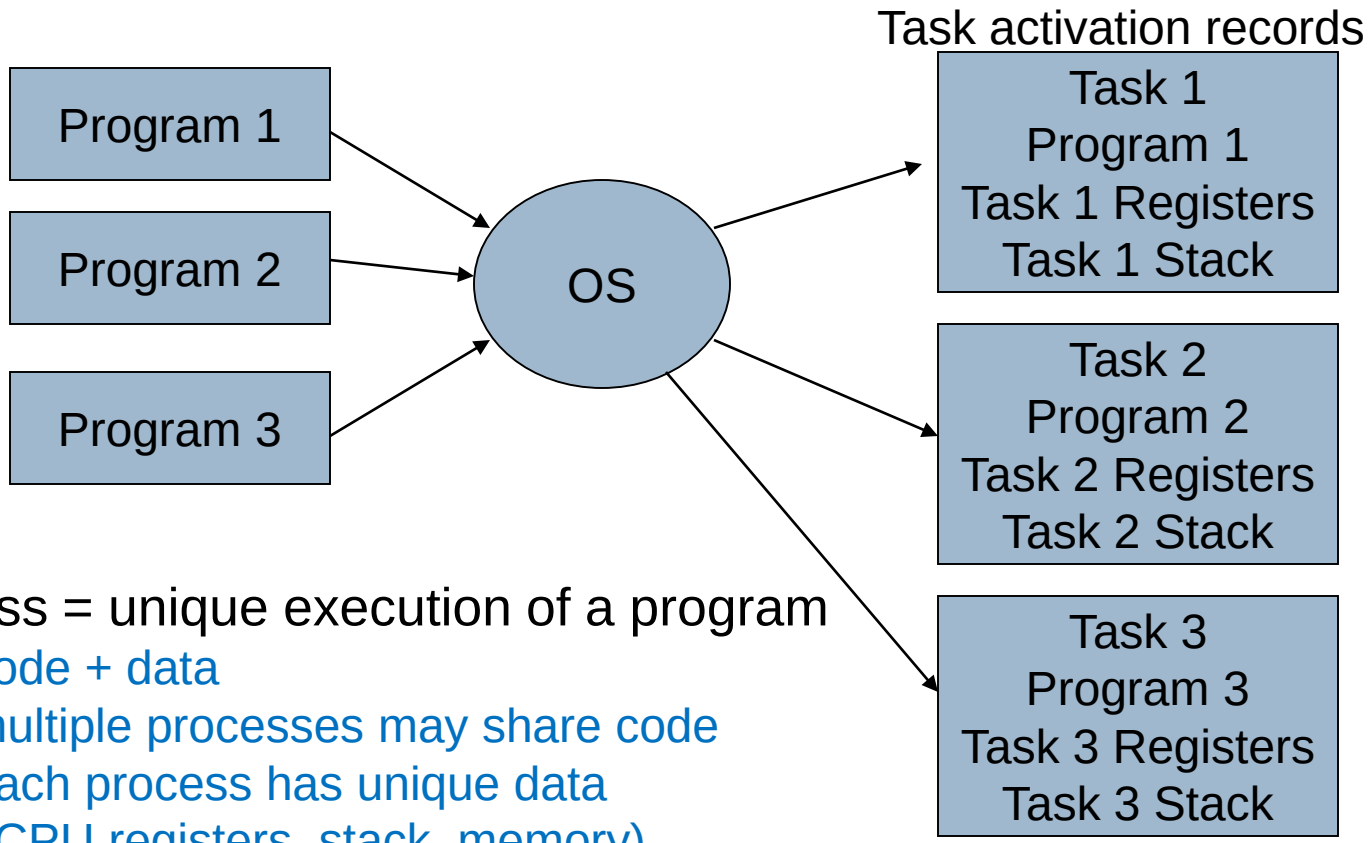
Keil ARM CMSIS Real-Time Operating System (CMSIS-RTOS)

- FreeRTOS.org
- POSIX (IEEE Standard)
- AMX (KADAK)
- C Executive (JMI Software)
- RTX (CMX Systems)
- eCos (Red Hat)
- INTEGRITY (Green Hills Software)
- LynxOS (LynuxWorks)
- µC/OS-II (Micrium)
- Neutrino (QNX Software Systems)
- Nucleus (Mentor Graphics)
- RTOS-32 (OnTime Software)
- OS-9 (Microware)
- OSE (OSE Systems)
- pSOSystem (Wind River)
- QNX (QNX Software Systems)
- Quadros (RTXC)
- RTEMS (OAR)
- ThreadX (Express Logic)
- Linux/RT (TimeSys)
- VRTX (Mentor Graphics)
- VxWorks (Wind River)

OS process management

- OS needs to keep track of:
  - process priorities;
  - scheduling state;
  - process activation records.
- Processes may be created:
  - statically before system starts;
  - dynamically during execution.
    - Example: incoming telephone call processing

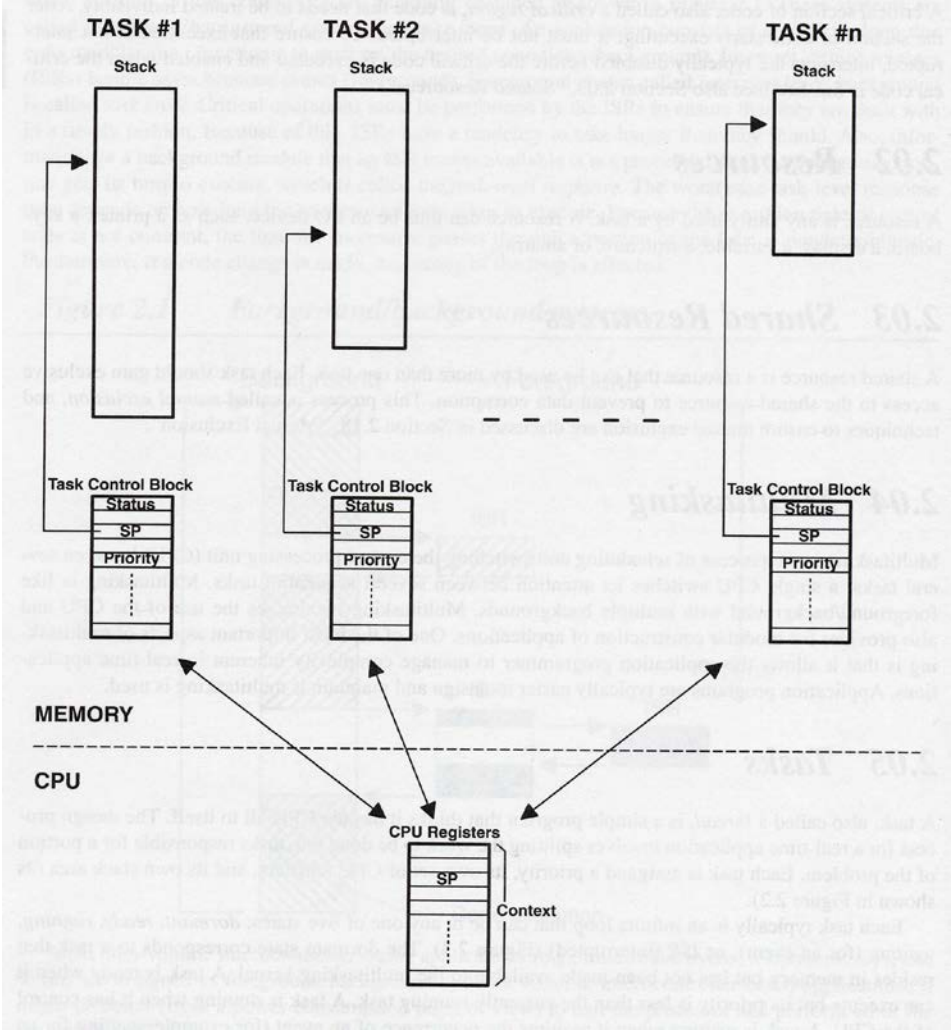
Multitasking OS



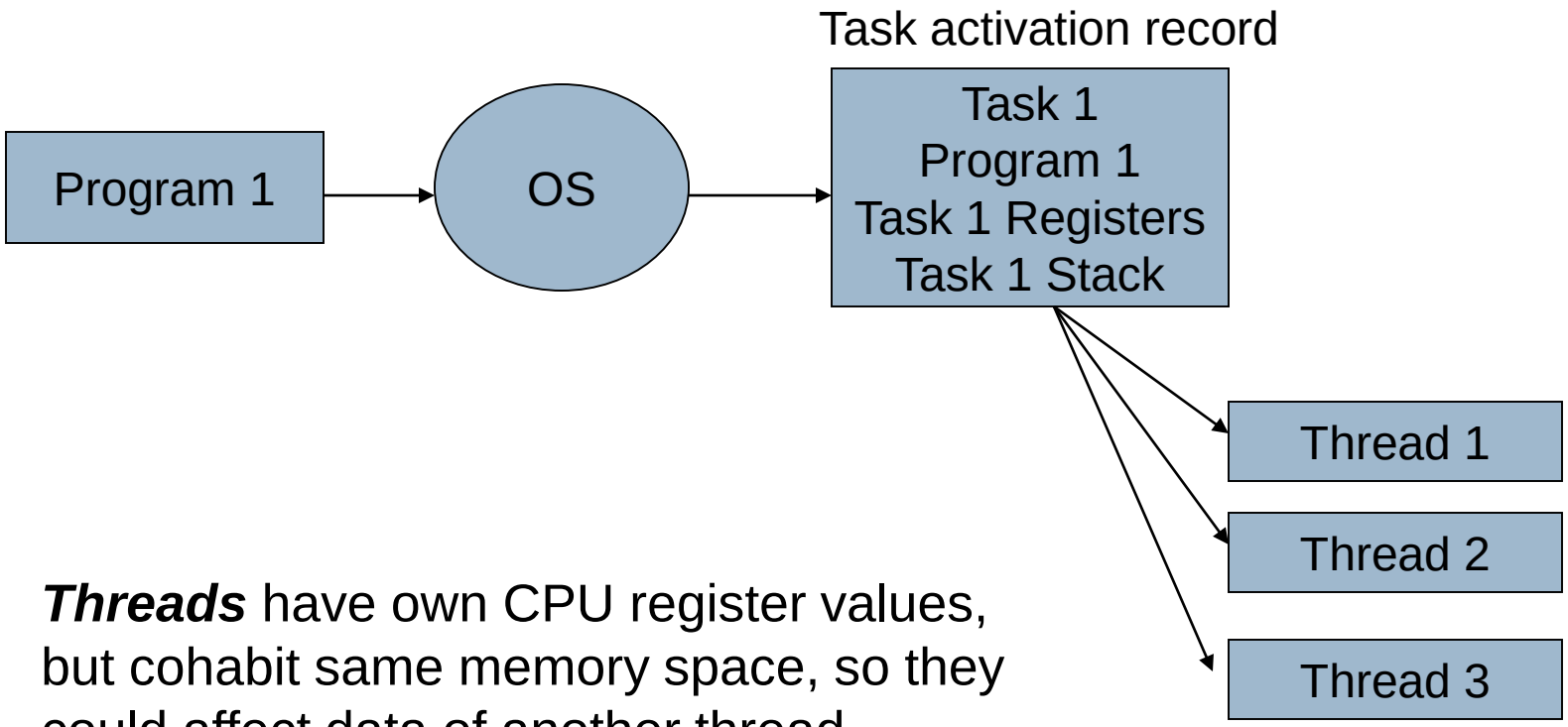
Process = unique execution of a program

- code + data
- multiple processes may share code
- each process has unique data
  (CPU registers, stack, memory)
- process defined by its “activation record”

Multitasking OS



Process threads
(lightweight processes)



Threads have own CPU register values, but cohabit same memory space, so they could affect data of another thread.

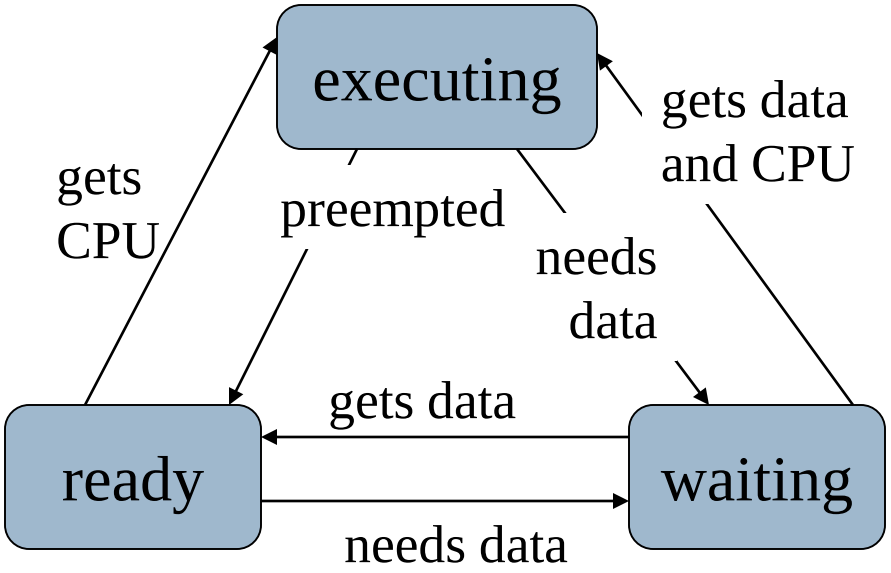
- a process may have multiple threads
- threads may run on separate CPU cores

Typical process/task activation records (task control blocks)

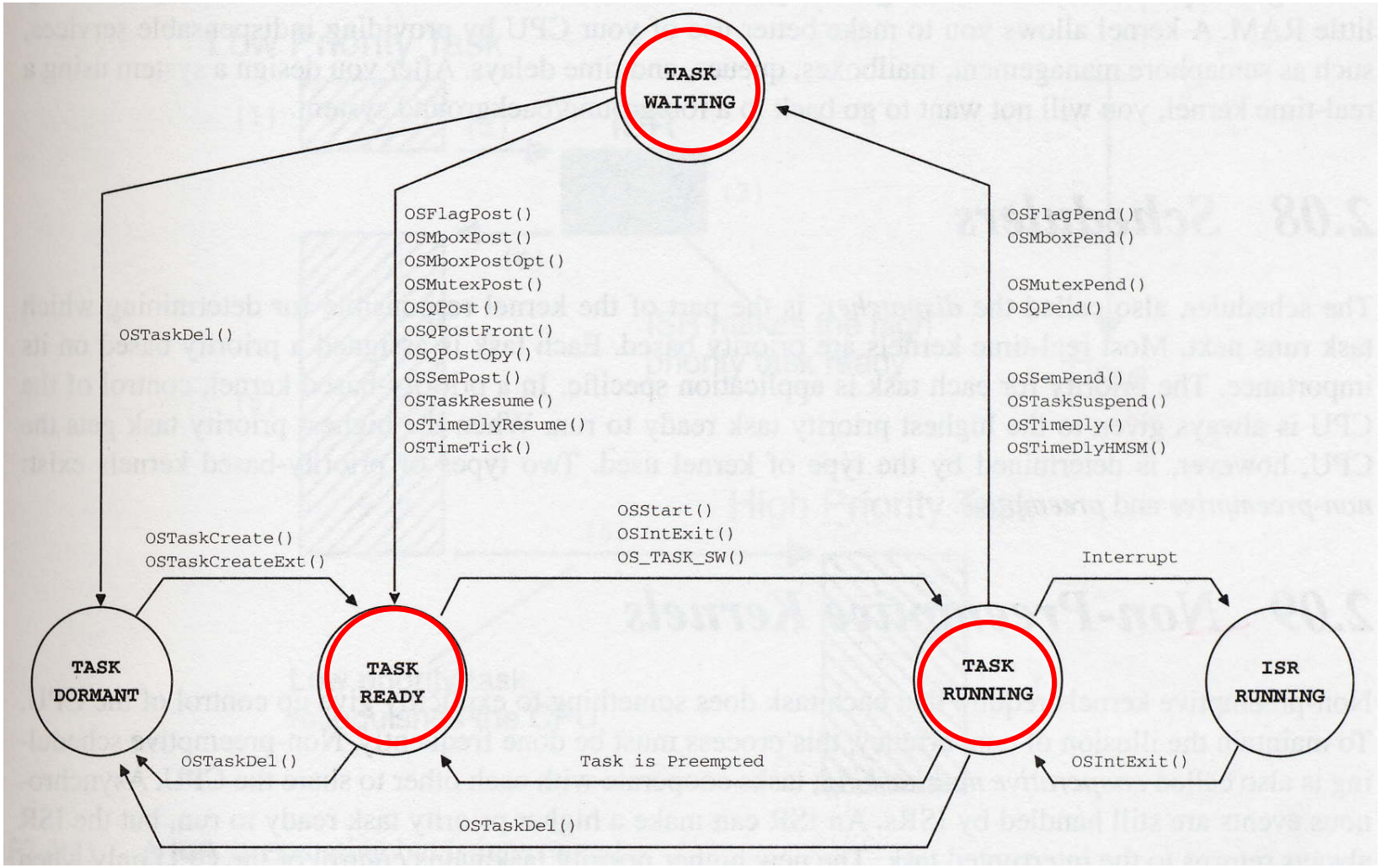
- Task ID
- Task state (running, ready, blocked)
- Task priority
- Task starting address
- Task stack
- Task CPU registers
- Task data pointer
- Task time (ticks)

Process state

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



Task/process states & OS functions



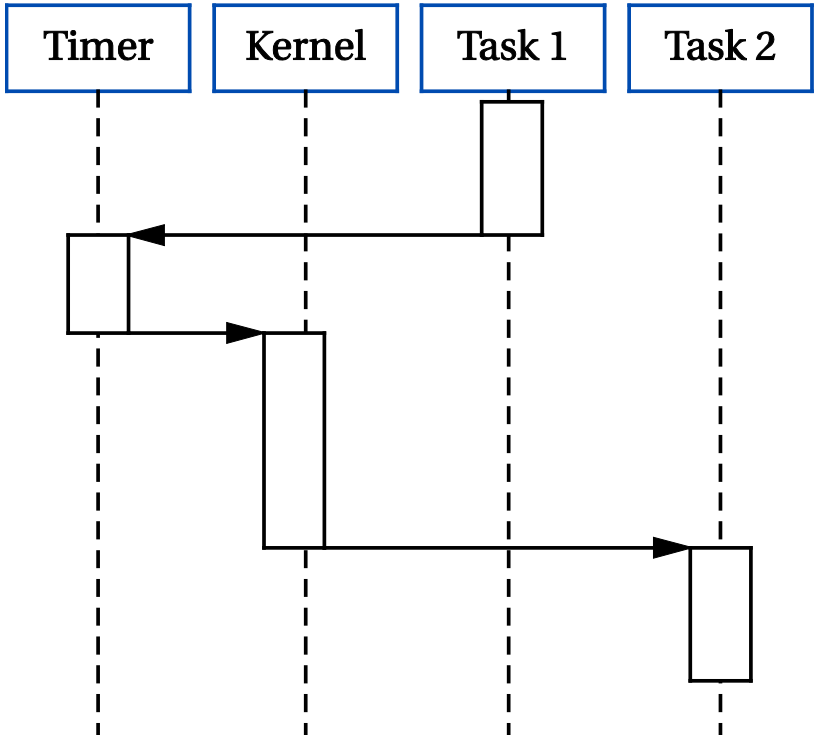
Priority-driven scheduling

- Each process has a **priority**, which determines scheduling policy:
  - fixed priority;
  - time-varying priorities.
  - CPU goes to highest-priority process that is ready.
- 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?
  - Consider CPU utilization

Preemptive scheduling

- Timer interrupt gives CPU to O/S kernel.
  - Time quantum is smallest increment of CPU scheduling time.

    "System tick timer"
- Kernel decides what task runs next.
- Kernel performs context switch to new context.

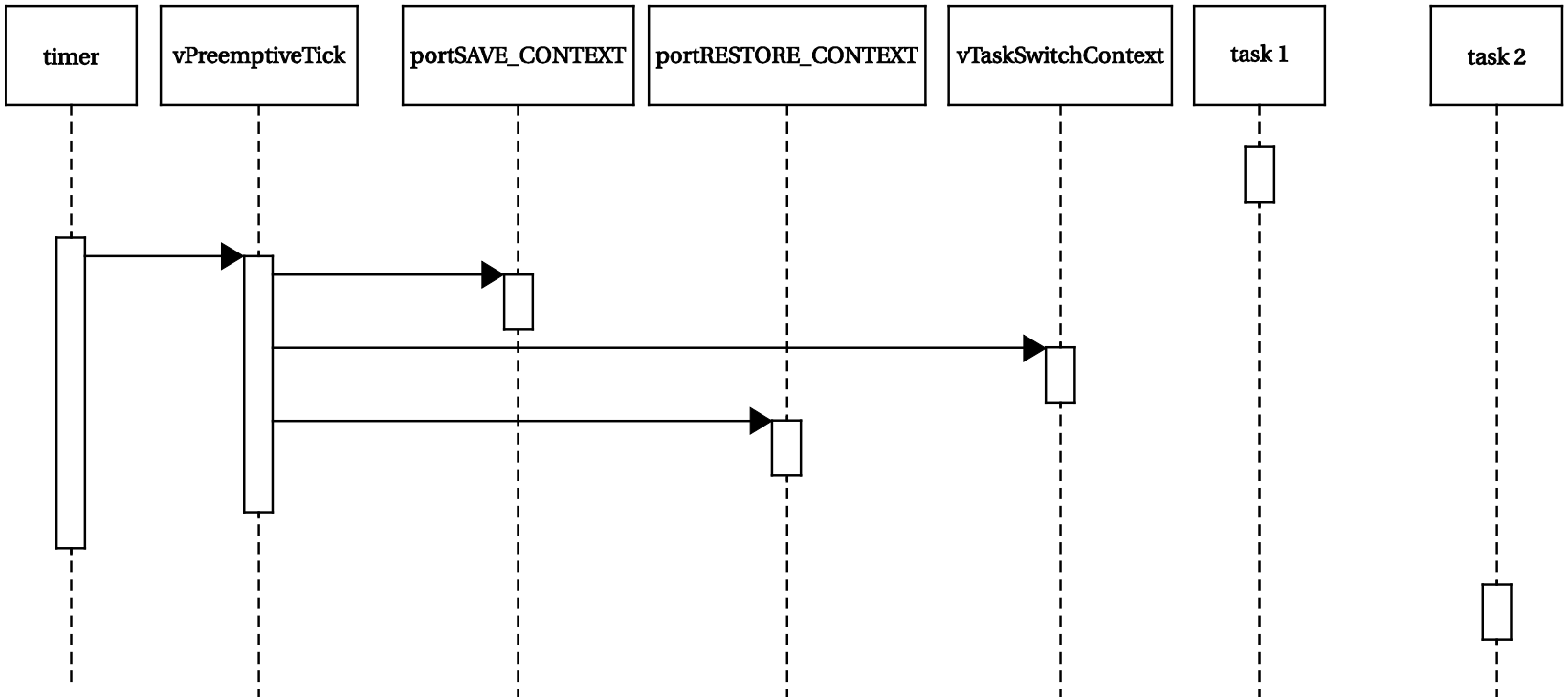


Context switching

- 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

(Handler on next slide)



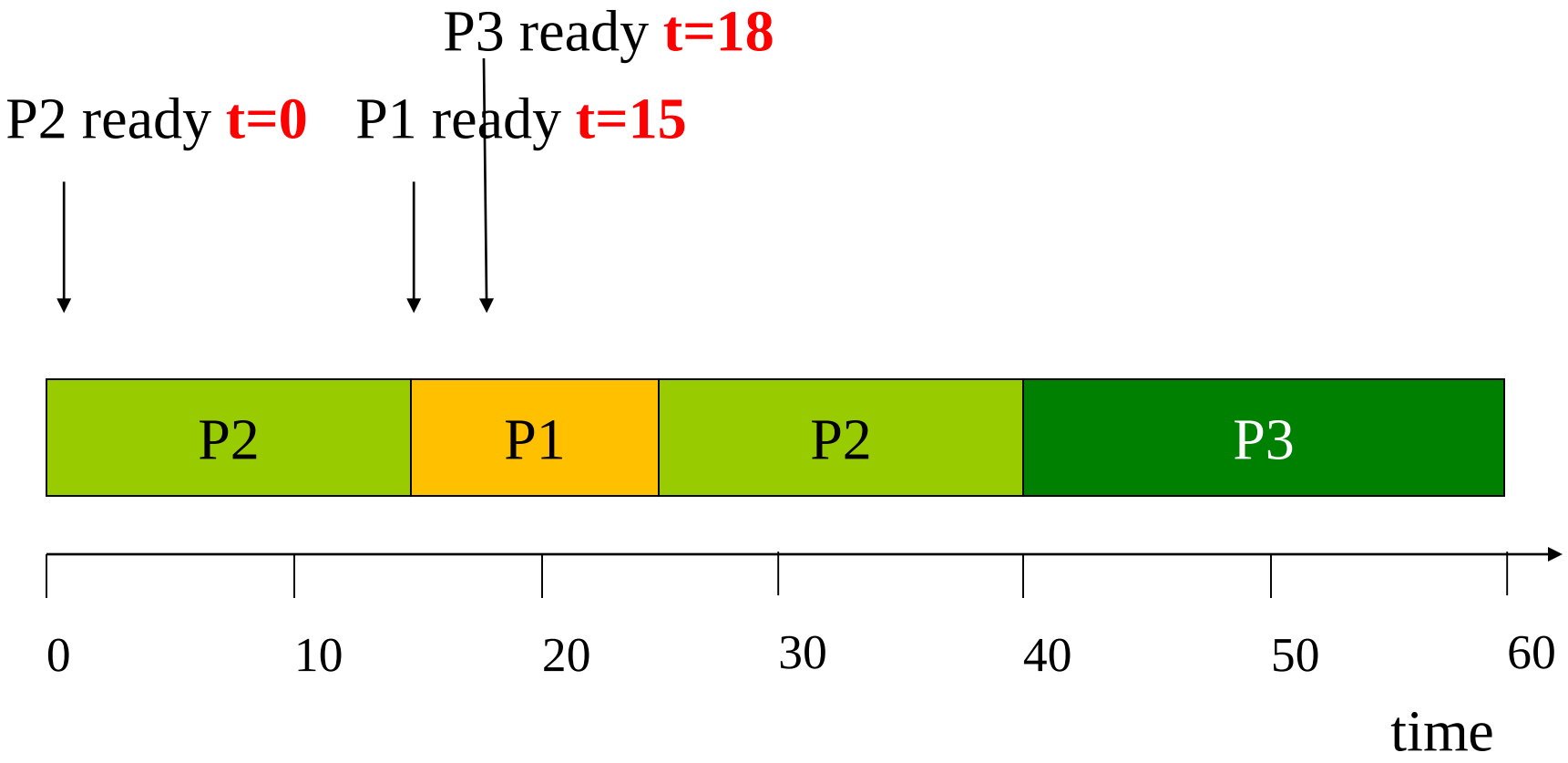
freeRTOS.org timer handler

```
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();
    }
```

Simple priority-driven scheduling example

- Rules:
  - each process has a fixed priority (1 = highest);
  - highest-priority ready process gets CPU;
  - process continues until done or wait state.
- Example (continued on next slide)
  - P1: priority 1, execution time 10
  - P2: priority 2, execution time 30
  - P3: priority 3, execution time 20

Priority-driven scheduling example



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 by which process must finish.
- Response time: time from occurrence of an "event" until the CPU responds to it.

- 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.

Process scheduling considerations

- Response time to an event
- Turnaround time
- Overhead
- Fairness (who gets to run next)
- Throughput (# tasks/sec)
- Starvation (task never gets to run)
- Preemptive vs. non-preemptive scheduling
- Deterministic scheduling (guaranteed times)
- Static vs. dynamic 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.

Some scheduling policies

- Round robin
  - Execute all processes in specified order
- Non-preemptive, priority based
  - Execute highest-priority ready process
- Time-slice
  - Partition time into fixed intervals
- **RMS** – rate monotonic scheduling (static)
  - Priorities depend on task periods
- **EDF** – earliest deadline first (dynamic)

Round-robin/FIFO scheduling

- Tasks executed sequentially
- No preemption – run to completion
- Signal RTOS when finished

```
while (1) {
   Task1();
   Task2();
   Task3();
}
```

$$T_{response} = \sum_{i=1}^{N} T_{Ti} + T_{TDn} + T_{cir} + \sum T_{\text{int},srv}$$

task times

context switch & OS overhead

circuit delays

service interrupts

Non-preemptive, priority-based schedule

- Task readiness checked in order of priority
- Task runs to completion

```
while (1) {
   if (T1_Ready)
     {Task1(); }
   else if (T2_Ready)
     {Task2(); }
   else if (T3_Ready)
     {Task3(); }
}
```

$$T_{response} = \sum_{i<n} N_i T_{Ti} + \max[T_n, T_{n-1}, \ldots] + T_{TDn} + T_{cir} + \sum T_{\text{int},srv}$$

- $N_i T_{Ti}$: higher priority tasks; Ni = #times Ti ready
- $\max[T_n, T_{n-1}, \ldots]$: time to finish a lower priority task
- T_{TDn}: context switch & OS overhead
- T_{cir}: circuit delays
- $\sum T_{\text{int},srv}$: service interrupts

Time-slice scheduler

- Timing based on "tick" = min. period
- Non-preemptive, priority-based :
  - execute all task once per "tick"
  - task runs to completion
- Minimum time slice:

$$T_{time-slice} > \sum_{i<n} T_{Ti} + \sum T_{\text{int},srv}$$

- Can make all execution times $k*T_{slice}$

$$T_{time-slice} \leq \gcd(T_{P1}, T_{P2}, ..., T_{Pn})$$

greatest common divisor

- RTOS provides timer functions
  - set, get, delay

```
while (1) {
   wait_for_timer();
   if (T1_Ready)
     {Task1(); }
   else if (T2_Ready)
     {Task2(); }
   else if (T3_Ready)
     {Task3(); }
 }
```

ARM CMSIS-RTOS scheduling policies

- Round robin schedule (OS_ROBIN = 1)
  - All threads assigned same priority
  - Threads allocated a fixed time
    - OS_SYSTICK = 1 to enable use of the SysTick timer
    - OS_CLOCK = CPU clock frequency (in Hz)
    - OS_TICK = "tick time" = #microseconds between SysTick interrupts
    - OS_ROBINTOUT = ticks allocated to each thread
  - Thread runs for designated time, or until blocked/yield
- Round robin with preemption
  - Threads assigned different priorities
  - Higher-priority thread becoming ready preempts (stops) a lower-priority running thread
  - When thread blocked, highest-priority ready thread runs
- Co-operative Multi-Tasking (OS_ROBIN = 0)
  - All threads assigned same priority
  - Thread runs until blocked (no time limit) or executes osThreadYield();
  - Next ready thread executes

Rate monotonic scheduling (RMS)

- RMS (Liu and Layland): widely-used, analyzable, static scheduling policy.
- Time-slice based, preemptive scheduling
- Tasks assigned priority according to how often they must execute
- Higher priority task preempts a lower-priority one
- Analysis is known as Rate Monotonic Analysis (RMA).

RMA model assumptions

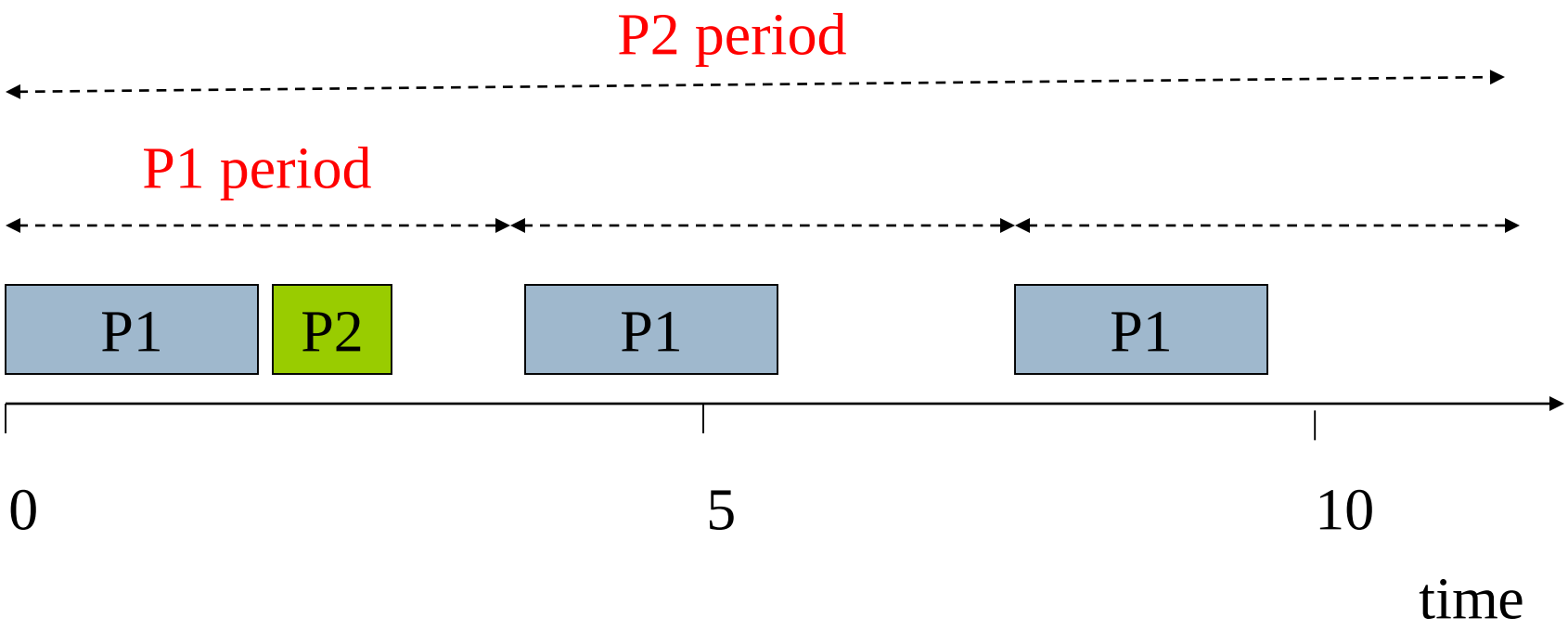
- All processes run on single CPU.
- Processes are periodic
- Zero context switch time.
- No data dependencies between processes.
- Process execution time is constant.
- Deadline is at end of period.
- Highest-priority ready process runs.

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 example

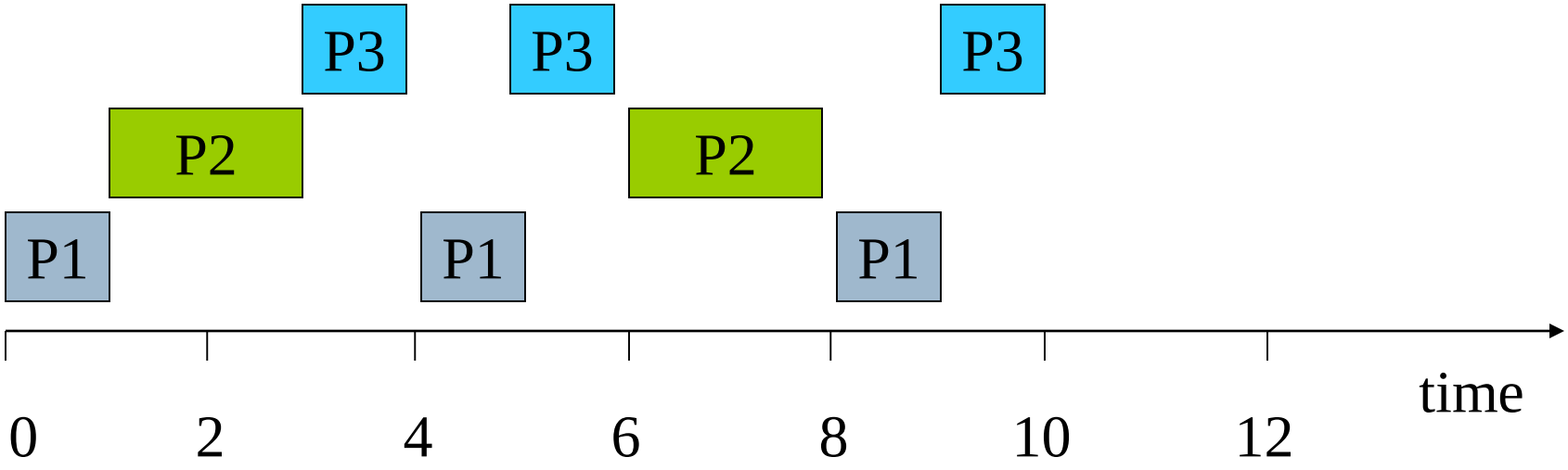
P1: Period = 4, Execution time = 2
P2: Period = 12, Execution time = 1
LCM of Period = 12
P1 higher priority



RMS example (Ex. 6-3)

Process	Execution time	Period	
P1	1	4	- highest priority
P2	2	6	
P3	3	12	- lowest priority

Unrolled schedule – LCM of process periods:



RMS example 2 (Ex. 6-4)

Process	Execution time	Period	
P1	2	4	- highest priority
P2	3	6	
P3	3	12	- lowest priority

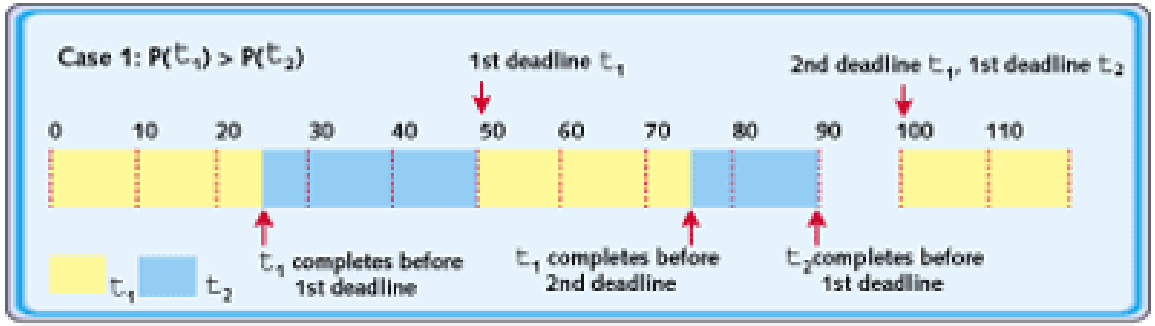
No feasible priority assignment to guarantee schedule
Consider CPU time over longest period (12 = LCM):

(3x2 for P1) + (2x3 for P2) + (1x3 for P3)
= 6 + 6 + 3
= 15 units > 12 units available

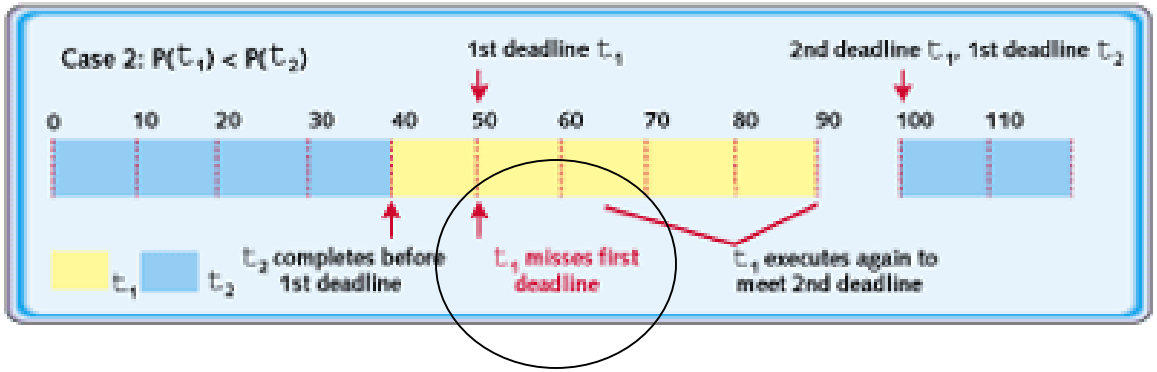
RMS Example

(http://www.netrino.com/Publications/Glossary/RMA.html)

- Case 1: Priority(Task1) > Priority(Task2)
- Case 2: Priority(Task2) > Priority(Task1)
  - **P1 = 50ms, C1= 25ms (CPU uti. = 50%)**
  - **P2 = 100ms, C2= 40ms (CPU uti. = 40%)**



Case 1

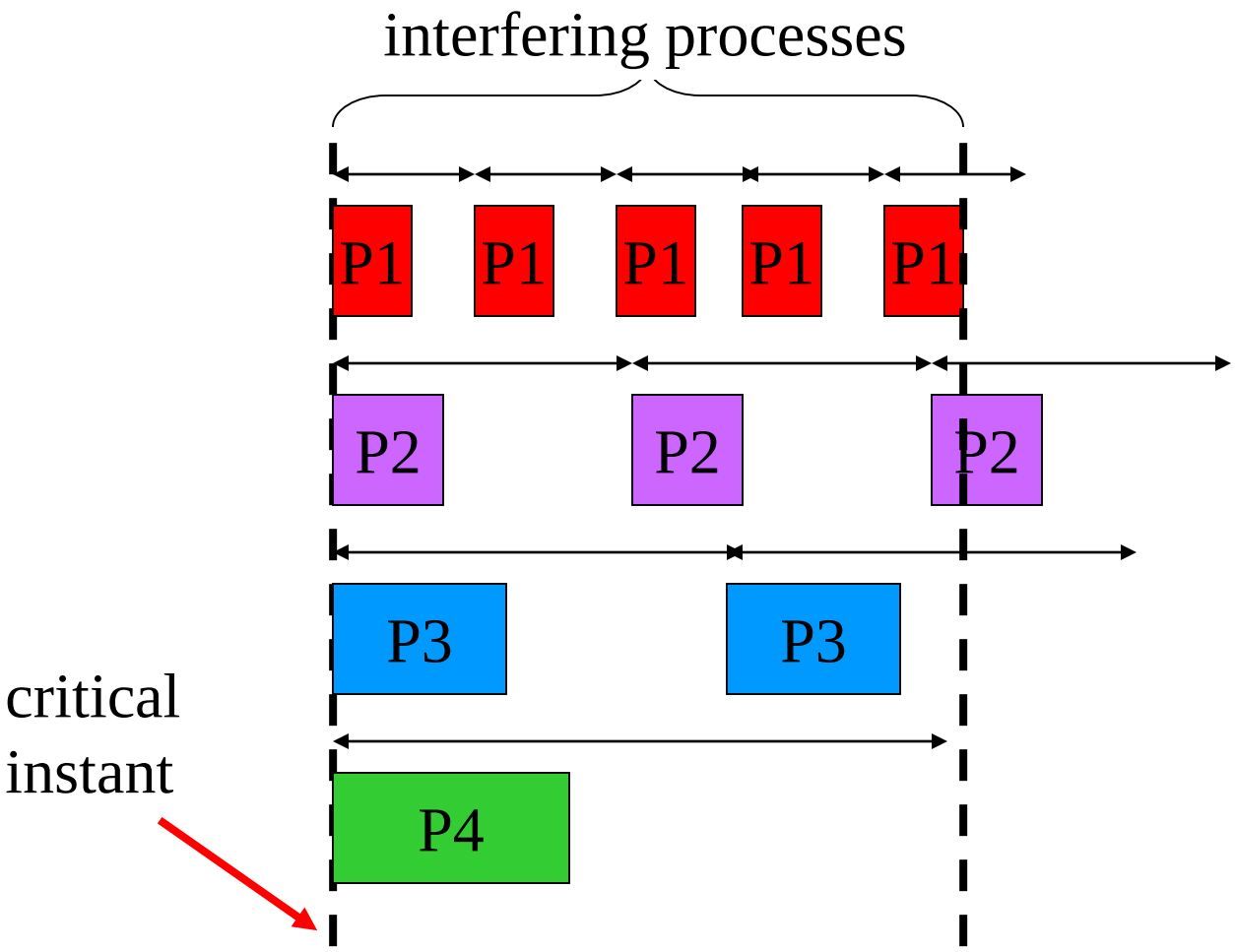


Case 2

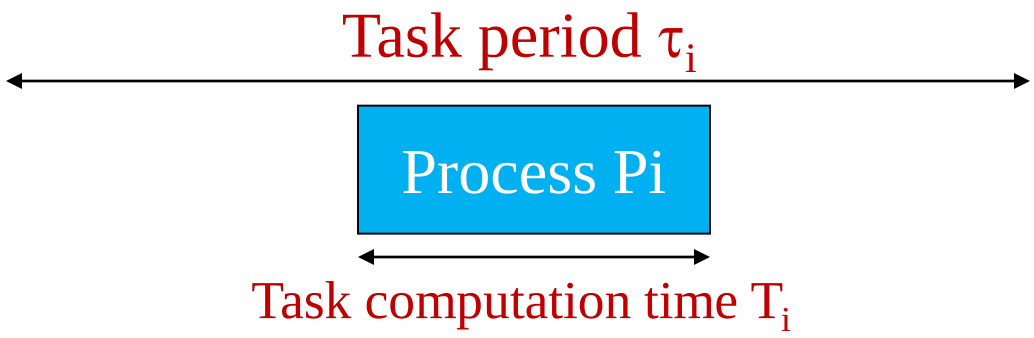
Rate-monotonic analysis

- Response time: time required to finish process.
- Critical instant: scheduling state that gives worst response time.
  - Critical instant for any process occurs when it is ready and all higher-priority processes are also ready to execute.
  - Consider whether the low-priority process can meet its deadline

Critical instant



CPU utilization for RMS



- CPU utilization for n processes is: $\sum_i T_i / \tau_i$
- All timing deadlines for *m* tasks can be met (***guaranteed***) if:

$$\sum T_i / \tau_i \leq m(2^{1/m} - 1)$$

- As number of tasks approaches infinity, maximum utilization approaches ln 2 = 69%.
  - *Liu & Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment", Journal of the ACM, Jan. 1973*

RMS CPU utilization, cont'd.

- **RMS guarantees all processes will always meet their deadlines.**
- RMS cannot asymptotically guarantee using 100% of CPU, even with zero context switch overhead.
- Must keep idle cycles available to handle worst-case scenario.

RMS implementation

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

(C code in figure 6.12 – pg. 330)

Earliest-deadline-first (EDF) scheduling

- **Process closest to its deadline has highest priority**.
- Dynamic priority scheduling scheme
  - Requires ***recalculating*** process priorities at every timer interrupt.
  - then select highest-priority ready process
- Priorities based on
  - frequency of execution
  - deadline
  - execution time of the process
- Usually clock-driven
- More complex to implement than RMS
  - must re-sort list of ready tasks

EDF example (ex. 6-4)

Process	Execution time	Period
P1	1	3
P2	1	4
P3	2	5

CPU utilization = 1/3 + 1/4 + 2/5 = .98333333 (too high for RMS)

Time	Running	Deadlines
0	P1	
1	P2	P1
2	P3	P2
3	P3	P3
4	P1	P1
5	P2	
6	P1	P2
7	P3	P1
8	P3	P3
9	P1	

Time	Running	Deadlines
10	P2	P1,P2
11	P3	
12	P3	P1,P3
13	P1	P2
14	P2	
15	P1	P1
16	P2	
17	P3	P2,P3
18	P3	
19	P1	

EDF analysis

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

EDF implementation

- More complex than RMS.
- On each timer interrupt:
  - compute time to deadline;
  - choose process closest to deadline.
- Generally considered too expensive to use in practice due to changing priorities.

(C code example in figure 6.13 – pg. 336)

POSIX scheduling policies

- SCHED_FIFO: RMS
  - FIFO within priority level
- SCHED_RR: round-robin
  - Within priority level, processes time-sliced in round-robin fashion
- SCHED_OTHER: undefined scheduling policy used to mix non-real-time and real-time processes.

```
/* POSIX example – set scheduling policy */
#include <sched.h>
int I, my_process_id;
struct sched_param my_sched_params;
….
i = sched_setschedule(my_process_id,SCHED_FIFO,&sched_params)
```

ARM CMSIS-RTOS scheduling policies

- Round robin schedule (OS_ROBIN = 1)
  - All threads assigned same priority
  - Threads allocated a fixed time
    - OS_SYSTICK = 1 to enable use of the SysTick timer
    - OS_CLOCK = CPU clock frequency (in Hz)
    - OS_TICK = "tick time" = #microseconds between SysTick interrupts
    - OS_ROBINTOUT = ticks allocated to each thread
  - Thread runs for designated time, or until blocked/yield
- Round robin with preemption (OS_ROBIN = 1)
  - Threads assigned different priorities
  - Higher-priority thread becoming ready preempts (stops) a lower-priority running thread
- Pre-emptive (OS_ROBIN = 0)
  - Threads assigned different priorities
  - Thread runs until blocked, or executes osThreadYield(), or higher-priority thread becomes ready (no time limit)
- Co-operative Multi-Tasking (OS_ROBIN = 0)
  - All threads assigned same priority
  - Thread runs until blocked (no time limit) or executes osThreadYield();
  - Next ready thread executes

Fixing scheduling problems

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

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.

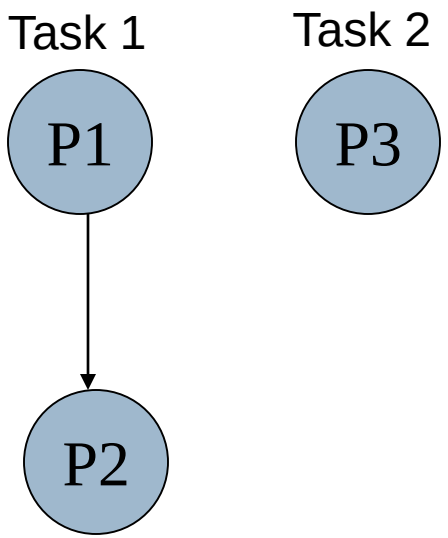
Solving priority inversion

- 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.
  - Allows it to finish without preemption

Data dependencies

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

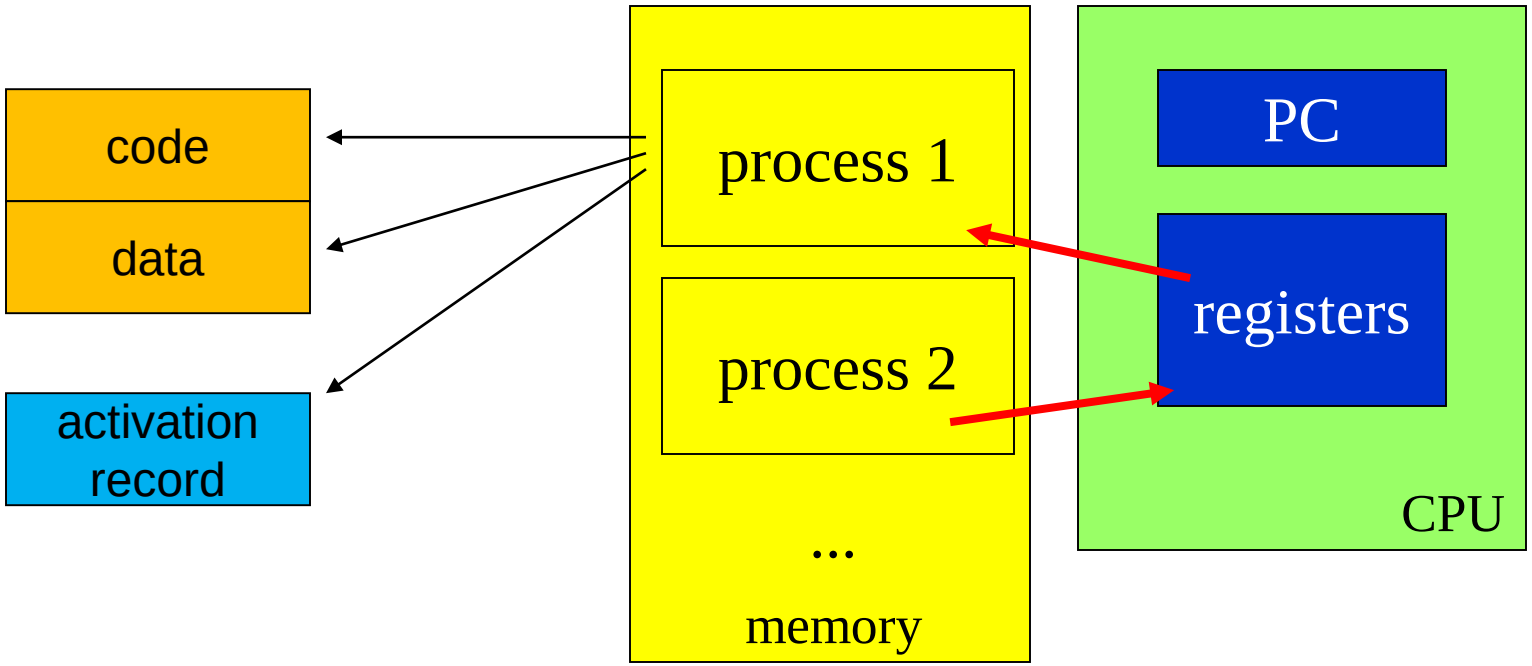
(prevents both P1 and P2 from running)



"Task graph"

Processes and CPUs

- Activation record: copy of process state (to reactivate)
- Context switch:
  - current CPU context goes out;
  - new CPU context goes in.



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.
  - Copy all registers to activation record, keeping proper return value for PC.
  - Copy new activation record into CPU state.
  - How does the program that copies the context keep its own context?

Context switching in ARM

- Save old process:

```
STMIA r13,{r0-r14}^
MRS r0,SPSR
STMDB r13,{r0,r15}
```

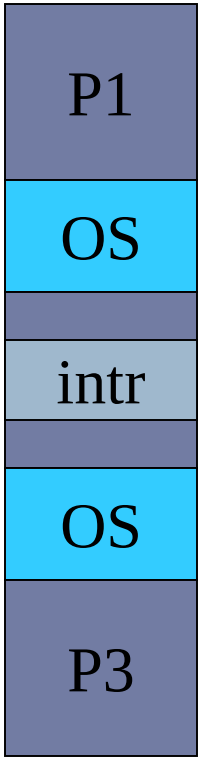
- Start new process:

```
ADR r0,NEXTPROC – get pointer
LDR r13,[r0]      - get context block ptr
LDMDB r13,{r0,r14} – status & PC
MSR SPSR,r0       - restore CPSR
LDMIA r13,{r0-r14}^ - rest of reg's
MOVS pc,r14       - resume process
```

STMIA: store multiple & increment address, ^ = user-mode registers
STMDB: save status register & PC

What about interrupts?

- Interrupts take time away from processes.
- Perform minimum work possible in the interrupt handler.
- Interrupt service routine (ISR) performs minimal I/O.
  - Get register values, put register values.
- Interrupt service process/thread performs most of device function.



Evaluating performance

- May want to test:
  - context switch time assumptions;
  - scheduling policy.
- OS simulator can exercise a process set and trace system behavior.

Processes in UML

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