RTOS - SCHEDULING

EL-GY 6483 Real Time Embedded Systems
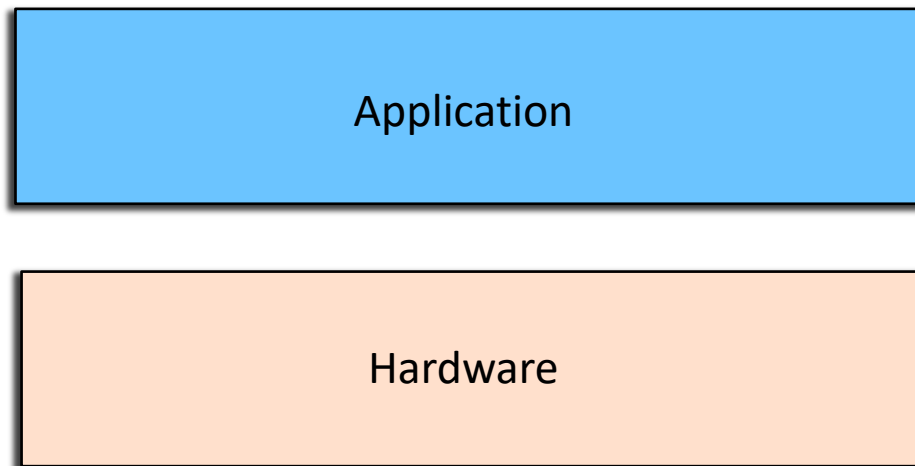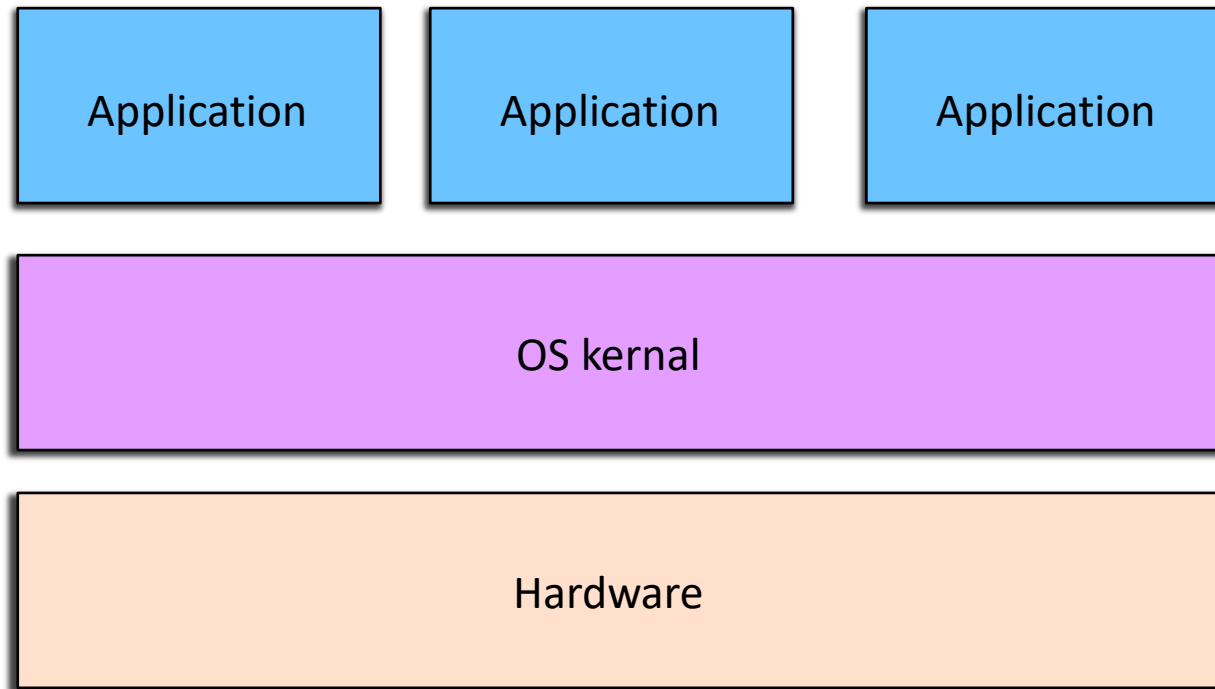
RTOS

CONTEXT

Until now, we've worked entirely in a bare metal (no OS) environment. Our systems consist of one application that sits directly above hardware:

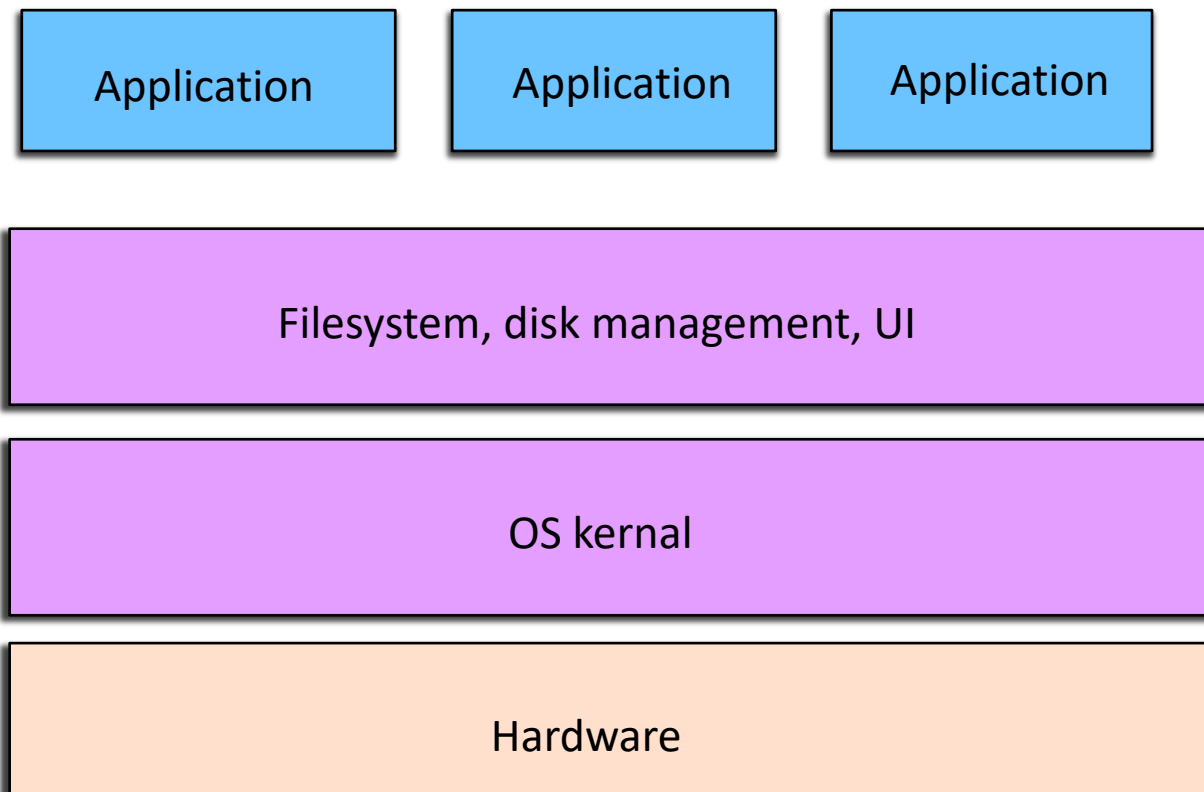| Application |
|:-----------:|

Hardware

CONTEXT

To run a system with multiple applications, we need an OS:

Application	Application	Application

OS kernal

Hardware

CONTEXT

Sometimes an OS can also provide a lot of utilities common to multiple applications:

Application	Application	Application

Filesystem, disk management, UI

OS kernal

Hardware

OS UTILITIES

An OS typically provides:

• Support for multitasking/concurrency (running multiple applications)

• Hardware abstraction layer (device drivers) for multiple devices

• Mechanisms for filesystems, communication, and others

RTOS VS. OTHER OSES

A real-time operating system is one that is designed to meet deadlines.

Other operating systems (not RT) include:

• Batch operating systems: take a set of jobs, run them in sequence

• Timesharing operating systems: try to make sure CPU time is shared fairly among users

An RTOS is not (primarily) concerned with fairness.

TASK MODEL

TASK

A task is the logical unit of work that can be scheduled by the operating system.

In a "big" operating system we would have multiple levels of tasks: processes and threads

In a small RTOS we typically only have one "task" level

TASK CLASSIFICATION: PERIODIC TASKS

Periodic tasks are executed at regular intervals (e.g. tasks driven by timers). They are characterized by:

c - computing time (a.k.a. execution time)

d - deadline, the time by which execution must be completed.

p - period (e.g. 20ms, or 50HZ), the duration between start of one execution and start of the next.

Typically, to satisfy deadlines, need $c \leq d \leq p$

TASK CLASSIFICATION: NON-PERIODIC TASKS

- Non-periodic (a.k.a. aperiodic or event-driven) tasks are not executed at regular intervals (e.g. tasks driven by interrupts).

- Sporadic tasks are non-periodic tasks with minimum interarrival time T_{min}

MULTITASKING AND CONCURRENCY

In a system with multitasking, the scheduler interleaves multiple tasks so that they appear to be running concurrently:
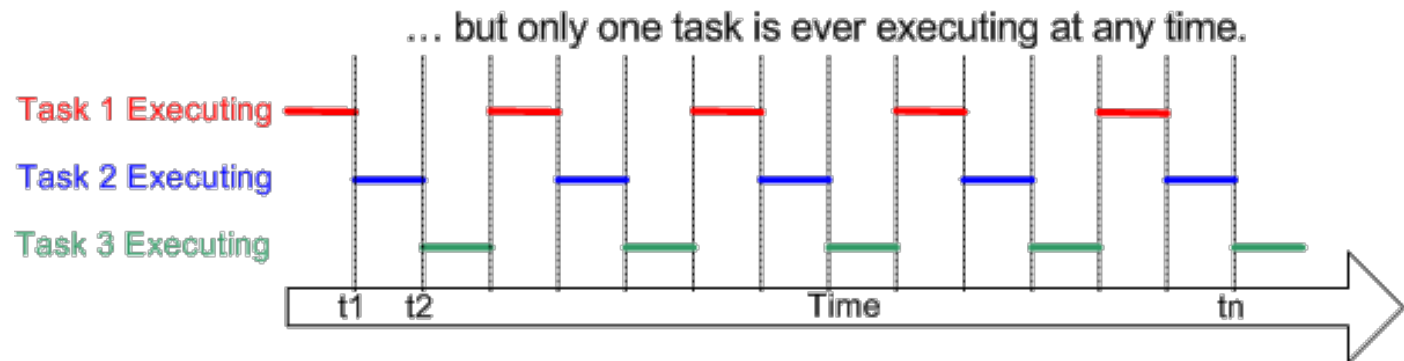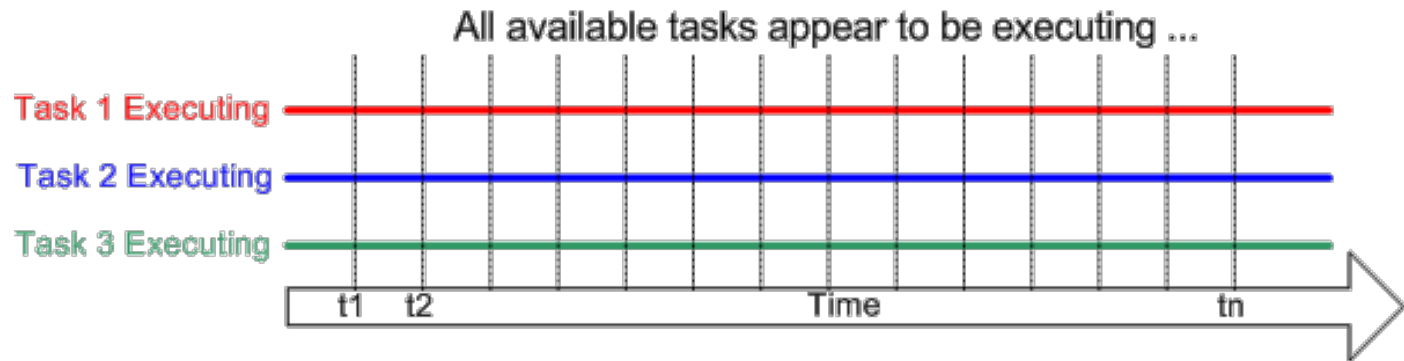
PSEUDOKERNELS

PSEUDOKERNELS

We can achieve some form of multitasking even without an operating system:

•Polling

•Cyclic executive

•Interrupt-driven system

POLLING

Polling is good when CPU is devoted to a single task, as a single task in cyclic executive system, or as a background task in interrupt-driven system.

```c
int main(void)
{
        while(1) {
                if (packet_here) {
                        process_data();
                        packet_here=0;
                }
        }
}
```

CYCLIC EXECUTIVE

A set of n tasks, execute in round-robin schedule:

```c
int main(void)
{
// initialization code here
        while (1)
        {
                work_on_task_1();
                work_on_task_2();
                work_on_task_3();
        }
}
```
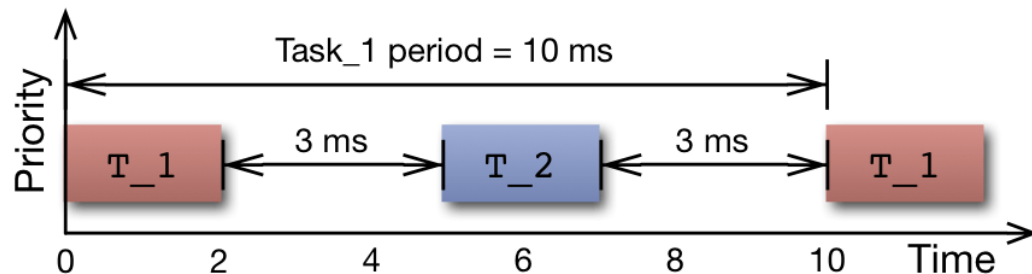
CYCLIC EXECUTIVE

For periodic tasks with specific periods, can use timers to get desired behavior:

```
while (1) {

        start_timer(5);

        work_on_task_1();

        wait_for_timer();


        start_timer(5);

        work_on_task_2();

        wait_for_timer();

}
```

CYCLIC EXECUTIVE

Advantages

•Simple to implement

•Low overhead

•Predictable performance (if tasks are short and have mostly uniform execution time)

Drawbacks

•Manual schedule: cannot deal with any runtime changes

•Handling sporadic events difficult

•Difficult to divide tasks into short pieces, uniform execution time

•Potentially lengthy response times

18

INTERRUPT-DRIVEN SYSTEM

Main function (background) is an empty loop, all tasks are handled in ISR (foreground).

```c
int main(void)

{

while (1);

}
void isr_1() {...}
void isr_2() {...}
void isr_3() {...}
```

INTERRUPT-DRIVEN SYSTEM

Advantages

•Handling sporadic events is easy (external hardware interrupts)

•Handling small periodic events is easy (timer interrupts)

Drawbacks

•No way to enforce any particular CPU sharing

•Must be careful to use reentrant functions, avoid data corruption

•Unpredictable performance if tasks are triggered by external hardware interrupts, or if there are many interrupt sources

•Potentially lengthy response times

You can combine methods:

```c
int main(void)
{
        while (1) {
                start_timer(5);
                work_on_task_1();
                wait_for_timer();
                start_timer(5);
                work_on_task_2();
                wait_for_timer();
        }
}

void timer1_isr() {...}
```
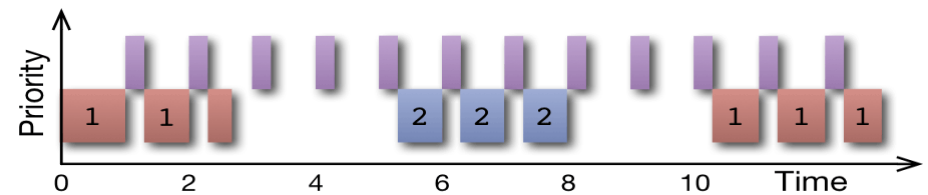
FOREGROUND/BACKGROUND SCHEDULING

- Works well if there are some tasks that need immediate response and others that are less sensitive.

- Does not scale well

None of these solutions scale well - for complicated systems, we need an operating system with a real time scheduler.
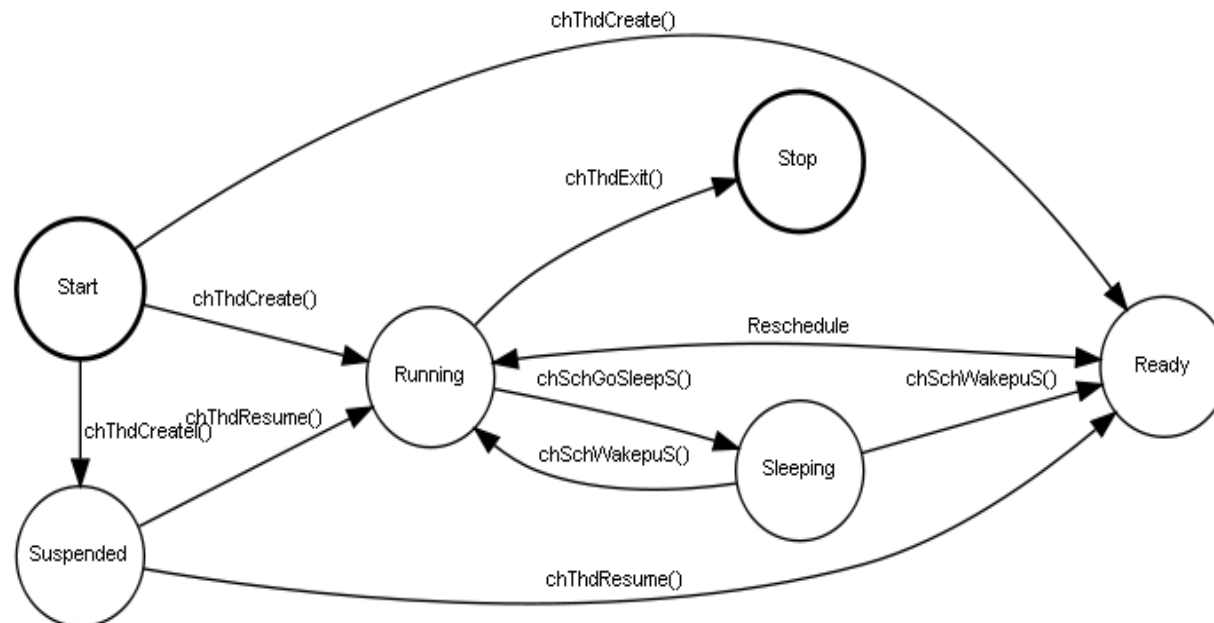
MULTITASKING

TASK STATE

On a single-processor, only one task may be running at a time

•A task that may be scheduled but isn't running is in ready state

•A task that can't be scheduled right now (e.g. waiting for some input) is in a suspended or blocked state

•Different OSes may have different names for task states.

PREEMPTION

In a preemptive multitasking setting, a task may be preempted (put into the background so another task can run):

- if its alloted time is up

- if a higher priority task needs to run.

In a non-preemptive (cooperative) setting, tasks voluntarily puts themselves in the background (e.g. calls a yield() function).

CONTEXT SWITCH

Since we may switch between tasks, we need to save the task's context when we do.

Each task gets its own "stack" (own region of memory that it uses as stack)

When switching between tasks, the stack pointer is updated to point to the top of the switched-in task's stack

We push the registers (including PC, PSR) on the stack of the switched-out thread and pop the registers of the switched-in thread

The time required to switch between tasks is called the thread switching latency or the context switching latency.

BASIC SCHEDULERS
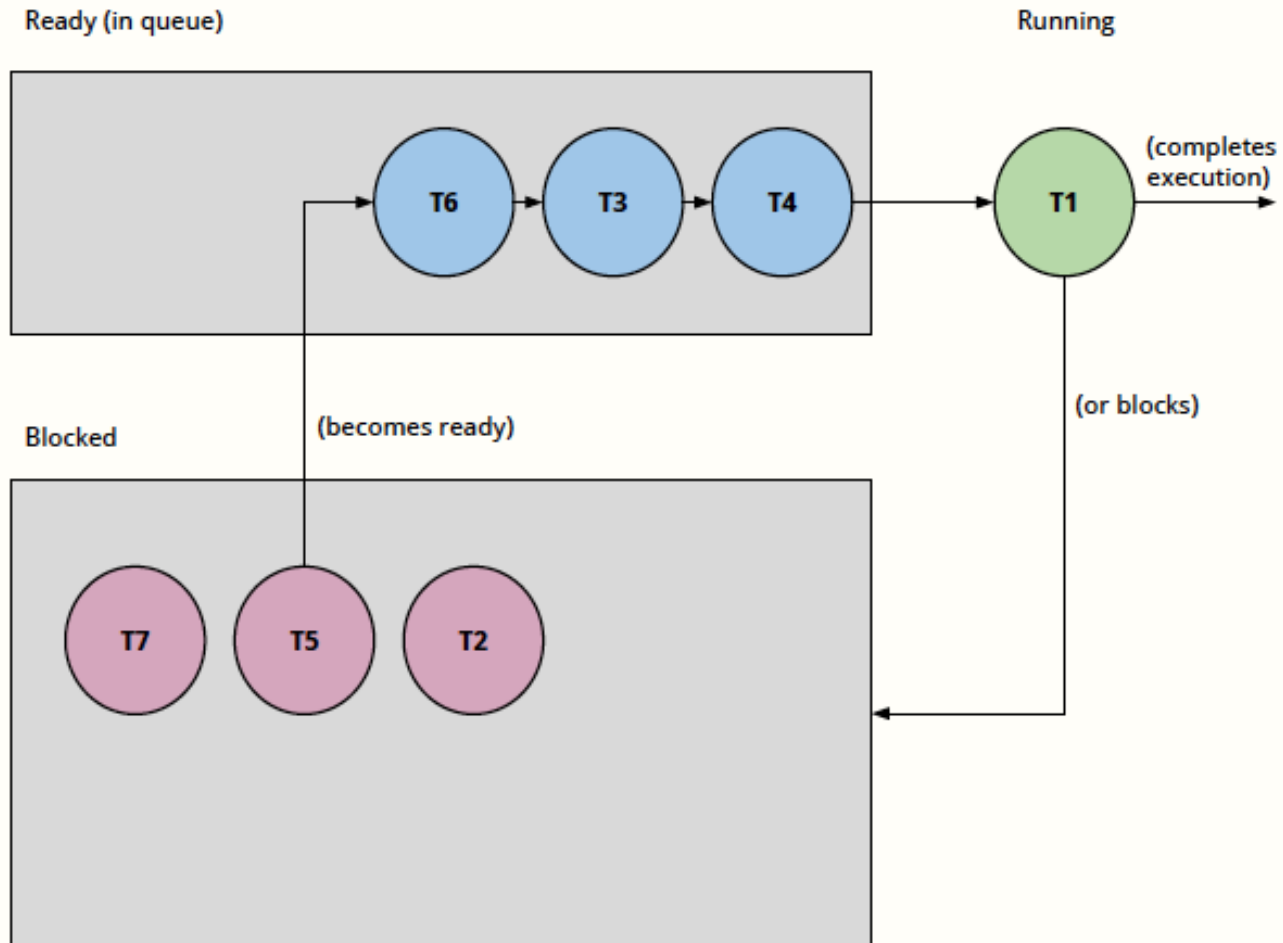
FIRST COME FIRST SERVED

New tasks that are ready to run go to the end of a queue.

- When the CPU is available, the scheduler picks the task at the head of the queue

- When running task finishes or blocks, the next task in the queue runs

- A long task may hold up entire system

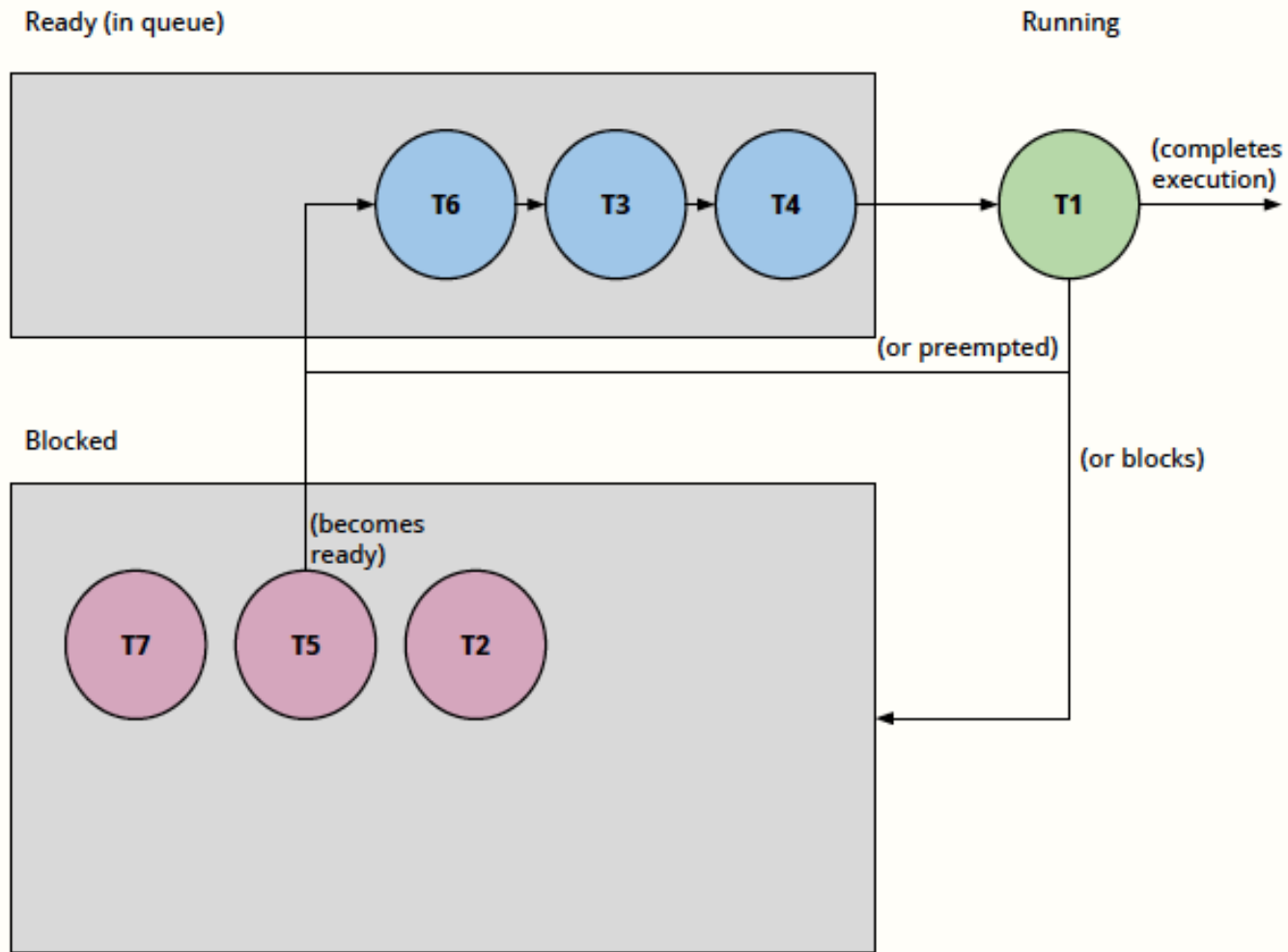No preemption.

FIRST COME FIRST SERVED

ROUND ROBIN

- Same thing, with preemption

- Each task is assigned a time slice

- A clock (e.g. SysTick) triggers an interrupt when time slice is up

- If the task isn't finished, its context is saved, and the task is placed at the back of the queue. (The task is preempted.)

ROUND ROBIN

Ready (in queue)　　　　　　　　　　　Running

T6 → T3 → T4 → T1 (completes execution)

(or preempted)

Blocked

(or blocks)

(becomes ready)

T7　T5　T2

MULTI-PRIORITY-LEVEL

- Tasks are assigned a priority class

- There's a separate "ready" queue for each class

- When the CPU becomes available, the scheduler picks the highest-priority queue (class) that has at least one task in it.

- Internally, each queue can be round-robin (for scheduling tasks at same priority level)

RATE MONOTONIC SCHEDULING

Scheduling algorithm that gives deterministic guarantees in some cases, i.e. promises that deadlines will be met.

- Rate monotonic algorithm is a procedure for assigning priorities to tasks to maximize their schedulability

- Tasks with shorter period given higher priority

- Under certain conditions, we can be assured that all tasks will meet their deadlines using this algorithm

- Tasks are run in order from high priority to low priority. A high-priority task preempts a low-priority running task.

RATE MONOTONIC ALGORITHM

Priority assignment rule: Each task is assigned a (unique) priority according to its period, so that the shorter the period, the higher the priority.

Process	Period	Priority
a	25	4
b	60	2
c	41	3
d	75	1

Table: Example: Rate monotonic algorithm

SCHEDULABILITY FOR RMS

Given certain assumptions[1], we say a set of tasks is schedulable (i.e. there is a schedule that will meet deadlines) if CPU utilization is below a certain bound:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1)$$

Here U is called CPU utilization, C_i is computation time of the i_{th} task, T_i is the period or shortest interarrival time for i_{th} task, n is the number of tasks. We assume deadline is equal to period.
(This condition is sufficient but not necessary.)

[1]See Liu & Layland (1973)

EARLIEST DEADLINE FIRST

Scheduling rule: Always schedule the task with the closest deadline.

Under certain conditions, it is optimal: if a set of tasks can be scheduled (by any algorithm) in a way that ensures all the jobs complete by their deadline, then EDF will schedule them so that they meet their deadline.
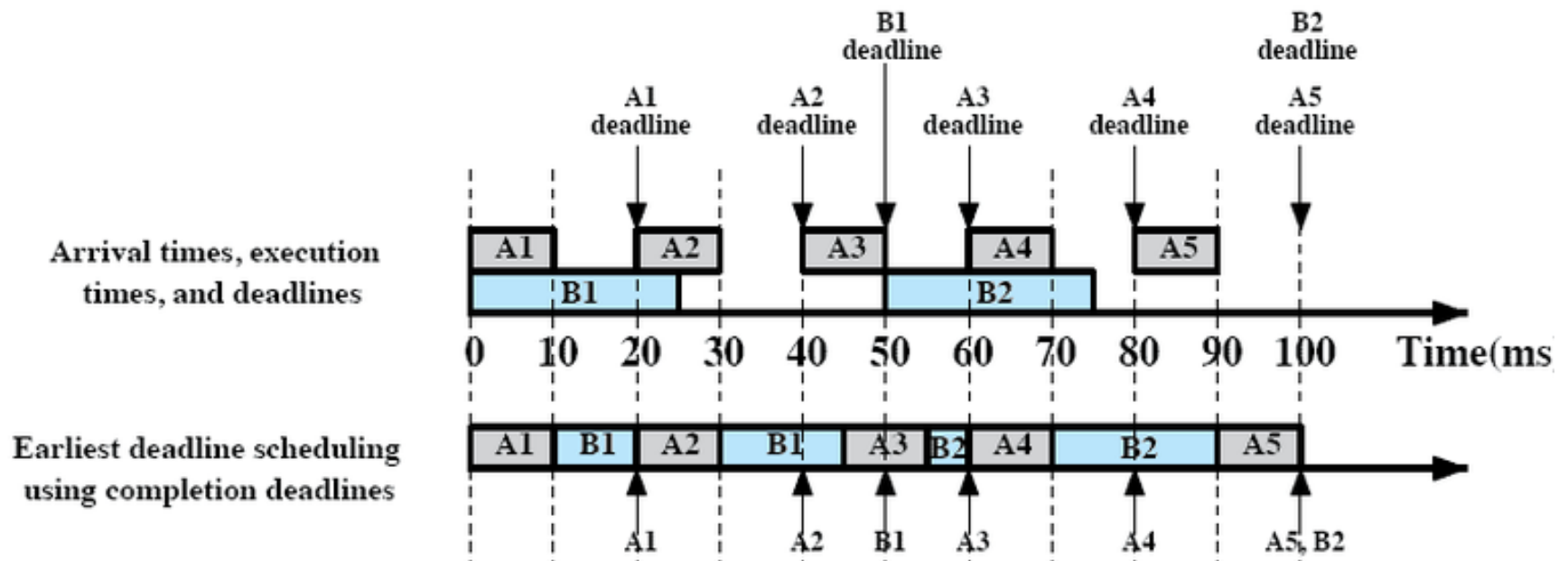
SCHEDULABILITY FOR EDF

EDF can guarantee that all deadlines are met if total CPU utilization is not more than 100%.

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

But if deadlines are not met, the results are highly unpredictable.

EDF AND RMS EXAMPLE

- A must finish by t = 30 ms, B must finish by t = 40 ms, and C must finish by t = 50 ms
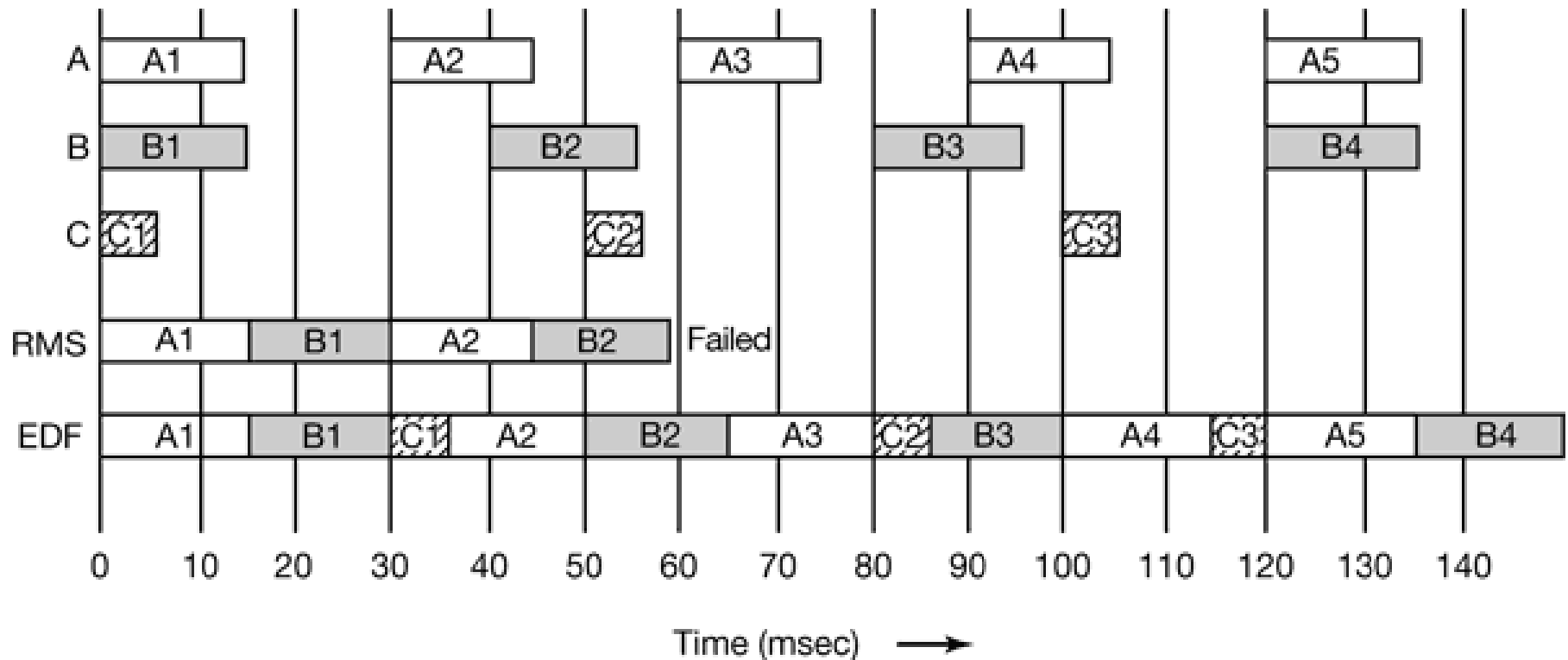
- A and B need 15 ms of CPU time, C needs 5 ms

Schedulability test:

$$\square = \frac{15}{30} + \frac{15}{40} + \frac{5}{50} = 0.975$$

Schedulable with EDF (U < 1), not guaranteed to be schedulable

with RMS $(U > 3(2^{\frac{1}{3}} - 1))$

EXAMPLE

- A must finish by t = 30 ms, B must finish by t = 40 ms, and C must finish by t = 50 ms

- A needs 10 ms of CPU time, B needs 15 ms, C needs 5 ms

Compute utilization. Is this set of tasks guaranteed to be schedulable with RMS? EDF?

Show task arrival times and schedule with RMS and EDF in a diagram, as in previous example.