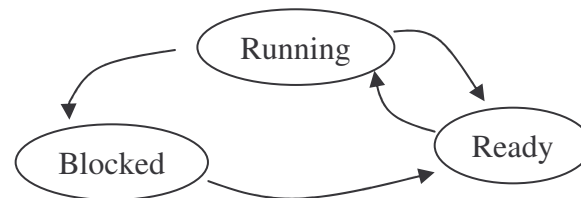




# Scheduling

# Issue

- When a computer is multiprogrammed it frequently has multiple processes competing for the CPU at the same time



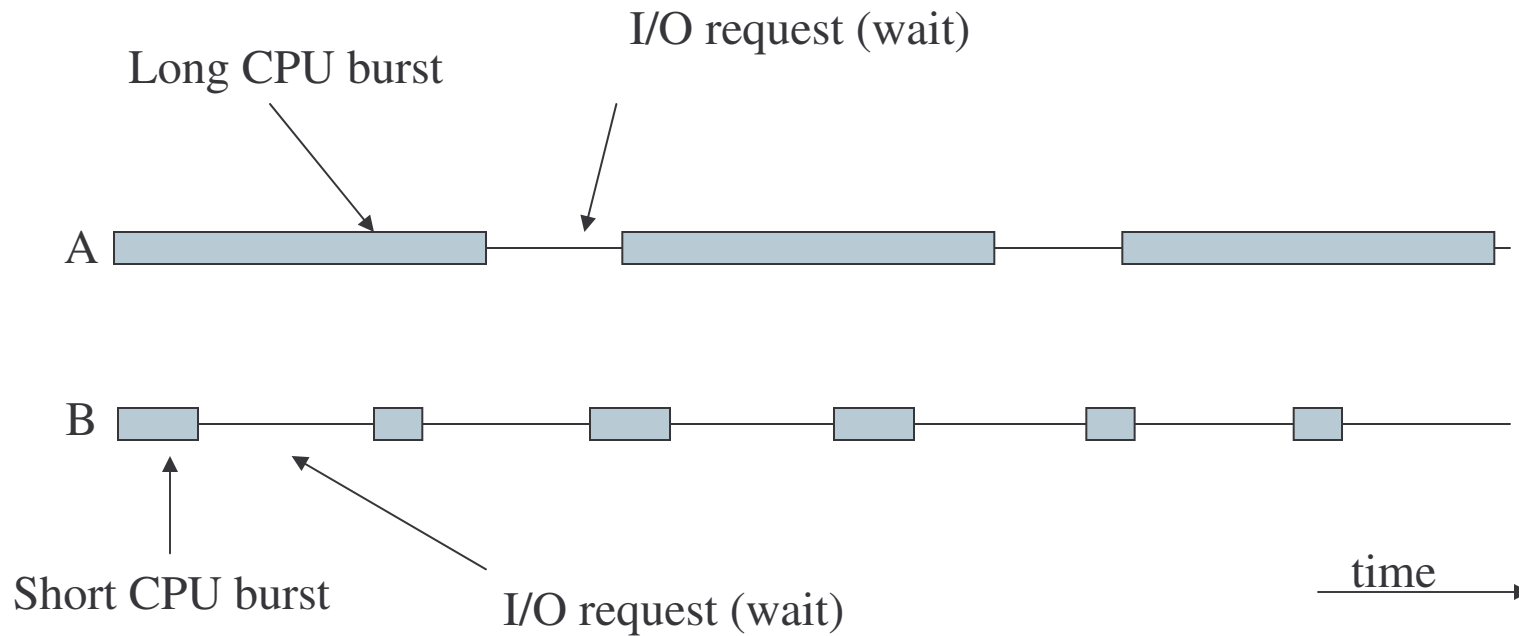
- A choice has to be made which process to run next

- the part of the operating system that makes this decision is called the **scheduler**
- the algorithm it uses is called the **scheduling algorithm**
- scheduling may involve both processes and threads

# Other issues

- depending on the application different scheduling strategies can make a difference  
example:  
simple PC  
networked server
- **process switching is expensive**  
user mode → kernel mode  
save the state of current process  
run the scheduler  
load MMU  
run new program  
→ the cache is now spoiled

# Process behavior



**A** spends most of his time computing, it is called **compute-bound**

**B** spends most of his time waiting for I/O, it is called **I/O-bound**

# When to schedule

- a new process is created
  - select the new one or keep the current one running
- a process terminates
  - select and run another process, if any
- a process blocks (semaphore, I/O)
  - dependencies btw processes may improve scheduling
- I/O interrupt
  - run a waiting process
- hardware clock
  - run the scheduler each clock interrupt or every k-th clock interrupt

# Scheduling can be divided:

- **non preemptive**

- picks a process to run
- lets it run until it blocks, terminates or voluntary releases the CPU
- after clock interrupt, resume the process that was running before

- **preemptive**

- picks a process to run
- after a maximum amount of some fixed time suspends it (if still running)
- picks another process to run (if any available)
- requires clock

# Scheduling: common goals

- **fairness**
  - comparable processes should get comparable service (CPU time)
- **policy enforcement**
  - different categories of processes may be treated differently
- **balance**
  - try to keep all the part of the system busy when possible

# Scheduling: specific goals

- **batch systems**
  - throughput
  - turnaround time
  - CPU utilization
- **interactive systems**
  - response time/proportionality
- **real-time systems**
  - meeting deadlines
  - predictability

# Scheduling in Batch Systems (1)

- **First-Come First-Served**

nonpreemptive

the CPU is assigned in the order processes require it

when the running process blocks the following one in the queue is selected

when a blocked process becomes ready it is put on the end of the queue

simple (a single queue), fair

not optimal

- **Shortest Job First**

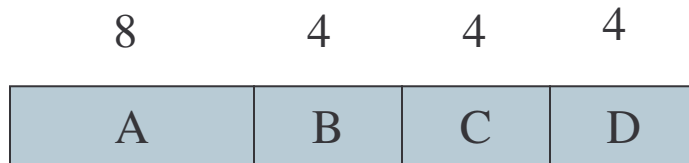
nonpreemptive

suppose we know the run-time in advance

the CPU is assigned to the shortest job in the queue

optimal if all the jobs are available at the same time

# Example (1)



turnaround:

$$A = 8$$

$$B = 12$$

$$C = 16$$

$$D = 20$$

$$\text{average} = 16$$



turnaround:

$$B = 4$$

$$C = 8$$

$$D = 12$$

$$A = 20$$

$$\text{average} = 11$$

suppose a,b,c,d

$$t_a = a$$

$$t_b = a+b$$

$$t_c = a+b+c$$

$$t_d = a+b+c+d$$

---

average =  $\frac{1}{4}(4a+3b+2c+d)$   $\rightarrow$  shortest time first is optimal

# Scheduling in Batch Systems (2)

- **Shortest Remaining Time Next**

preemptive

the scheduler here chooses the process whose remaining run-time is the shortest

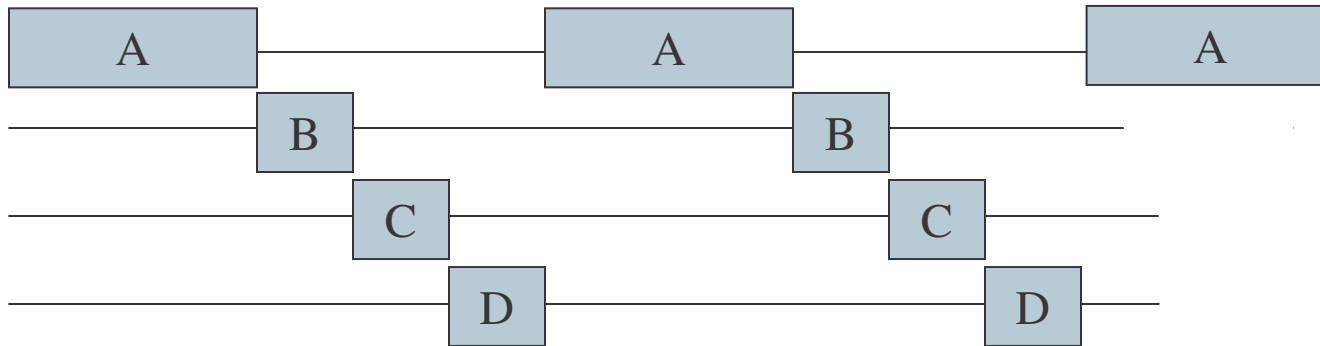
the time has to be known in advance

new short jobs get good service

# Example (2)

A, runs for 1s and blocks for I/O

B, C, D blocks after short time, needs to perform 1000 disk reads



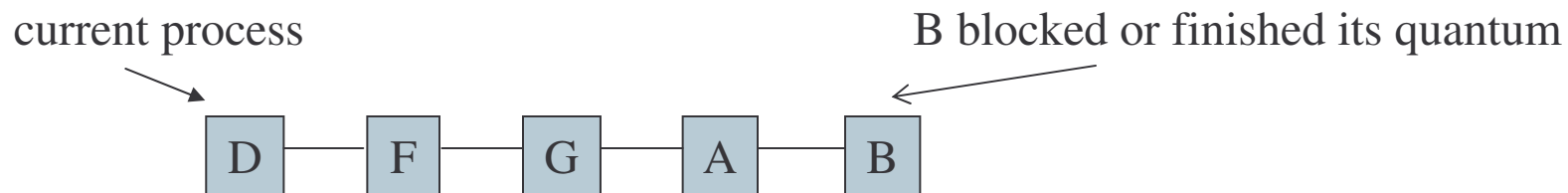
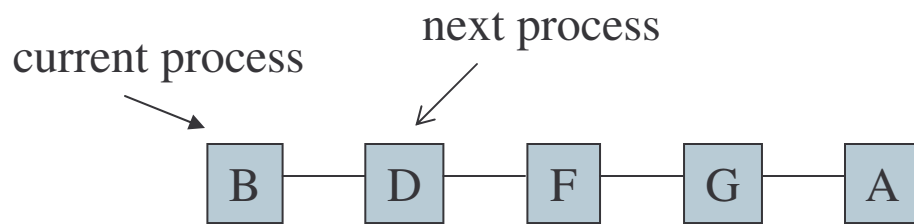
B, C, D, take at least 1000s to complete

# Scheduling in Interactive Systems (1)

- **Round Robin**

each process is assigned a time interval, called **quantum**

if the process is still running at the end of its quantum, the CPU is **preempted** and given to another process



# Scheduling in Interactive Systems (2)

- **issues with Round Robin**

length of the quantum

too short → context switch overhead

too long → poor response to short interactive requests

usually a reasonable value is 20-50 ms

- **Priority Scheduling**

each process is assigned a priority

priorities can be assigned:

- statically

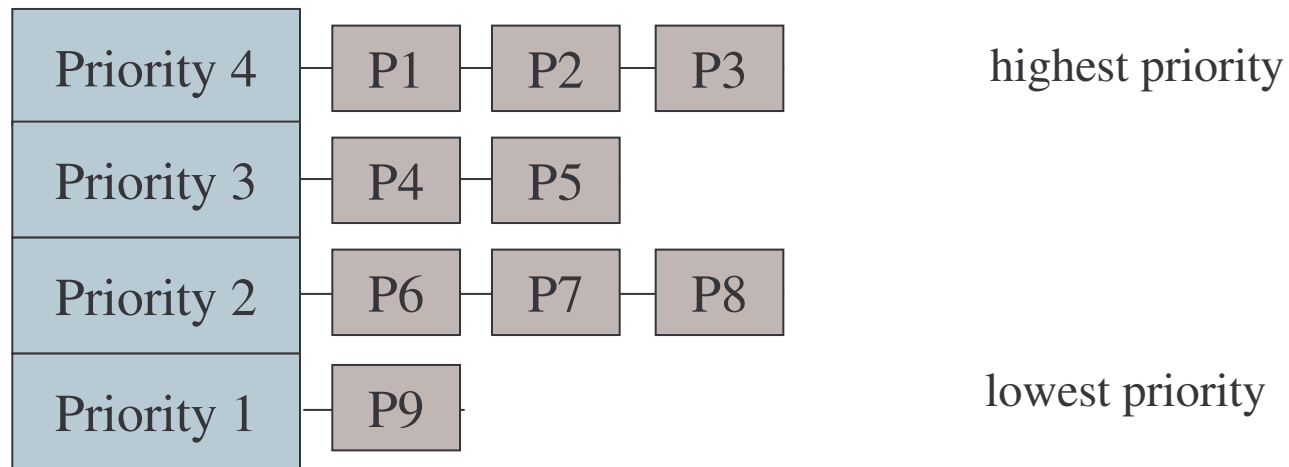
- dynamically: e.g. assign more CPU to I/O bound processes

divide processes in classes depending on priority

use priority scheduling within classes

round robin within classes

# Example, 4 priority classes



# Example: dynamic priority

assign priority depending on the fraction of quantum each process has used

$$P = \frac{1}{f}$$

example: time slice 50ms

process A uses 1ms,  $f=1/50$ , priority = 50

process B uses 50 ms,  $f=50/50$ , priority = 1

# Scheduling in Interactive Systems (3)

- **Shortest Process Next**

shortest job produces the minimum average response time for batch systems

the problem here is figuring which of the runnable processes is the shortest one

solution: use estimates based on past behavior

$T_i$  measured run-time at time  $i$

$\hat{T}_i$  estimate run-time at time  $i$

$$\hat{T}_n = a\hat{T}_{n-1} + (1-a)T_n$$

Example:  $a = 0.5$

$$T_0, \frac{T_0}{2} + \frac{T_1}{2}, \frac{T_0}{4} + \frac{T_1}{4} + \frac{T_2}{2}, \frac{T_0}{8} + \frac{T_1}{8} + \frac{T_2}{4} + \frac{T_3}{2}$$

# Scheduling in Interactive Systems (4)

- **Guaranteed Scheduling**

- make promises about performance to the users/processes
  - compute the real amount of CPU a user/process has consumed
  - increase priority accordingly
  - difficult to implement

- **Lottery Scheduling**

- basic idea: give processes lottery tickets for various system resources (CPU time)

- whenever a scheduling decision is required a lottery ticket is randomly chosen

- similar to priority scheduling, but:

- the rule is clearer

- interesting properties: tickets can be exchanged

# Scheduling in Interactive Systems (4)

- Fair-Share Scheduling

Example:

User A has 9 processes, User B has 1 process

A and B have same priority, Round Robin:

B1, A1, A2, A3, A4, ... A9, B1, A1, A2, ..., A9

A gets 90% of the CPU, B gets 10%

Possible solution: take into account who owns a process before scheduling it:

B1, A1, B1, A2, B1, A3, B1, A4..., B1, A9

# Policy versus Mechanism

- Often a process has many children running under its control performing different tasks. In this case only the process itself knows which one is the most important or time critical
- For this reason it is important to separate **scheduling mechanism** from the **scheduling policy**
- The scheduling mechanism (algorithm) defines the parameters used by the scheduler
- The user process is responsible for filling in those parameters for its children

# Scheduling in Real Time Systems

In **real time** systems time plays a crucial role. Usually the system is connected to one or more external devices which generate stimuli and has to react appropriately to them within a fixed amount of time.

Examples: aircraft control, over-temperature monitor in nuclear power station, ABS, biomedical systems, robotics

- **hard-real time**, missing a deadline has catastrophic effects
- **soft-real time**, missing a deadline is undesirable but tolerable

Stimuli (events) may be:

- **periodic** (occurring at regular intervals)
- **aperiodic** (unpredictable)

# Schedulability

- Depending on the situation, it may happen that not all the events can be handled
- Consider  $m$  periodic events  
event  $i$  occurs with period  $P_i$  and requires  $C_i$  second of CPU time  
the system is **schedulable** if:

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

# Let's consider the following situation:

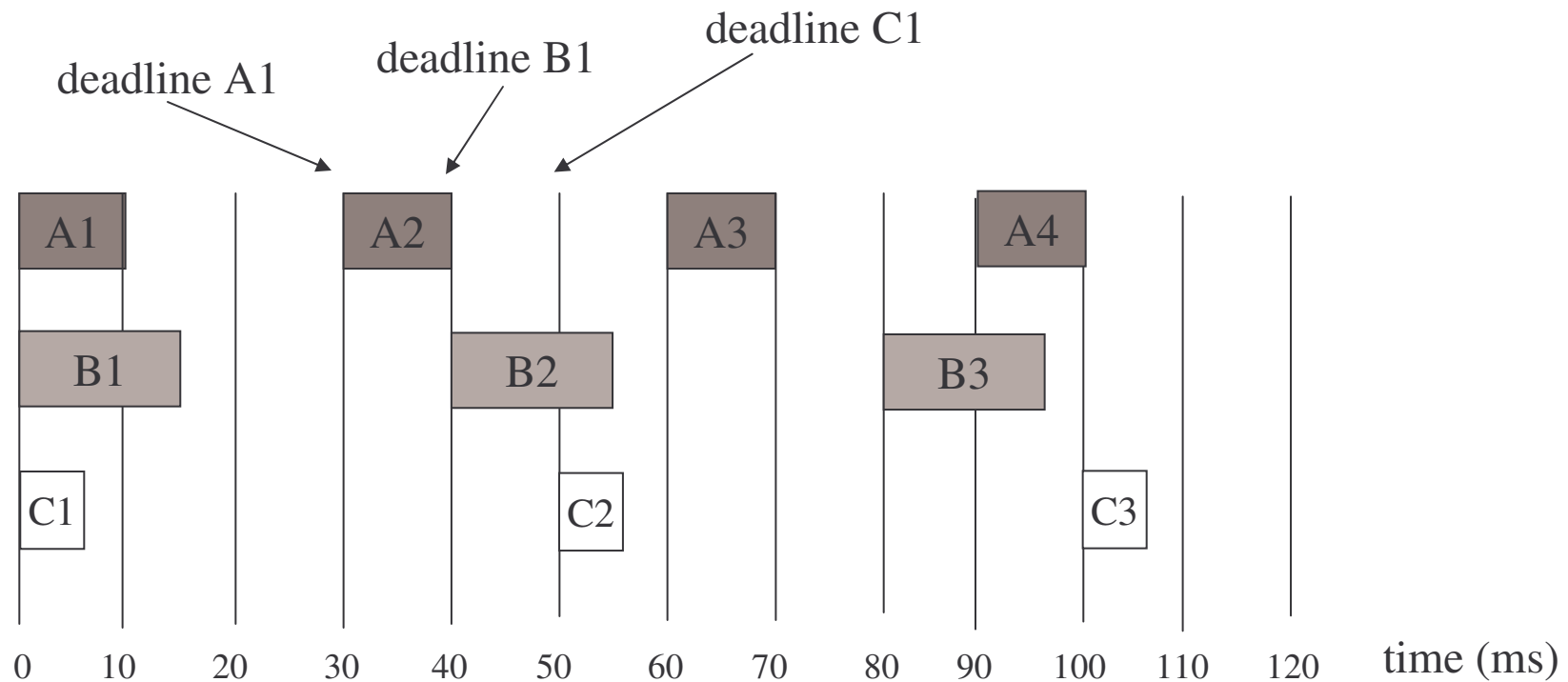
Multimedia system: three processes A, B, C

A is periodic,  $T = 30\text{ms}$ , and uses 10 ms of CPU time

B is periodic,  $f = 25\text{ Hz}$  ( $T=40\text{ms}$ ) and uses 15 ms of CPU time

C is periodic,  $f = 20\text{ Hz}$ , ( $T=50\text{ms}$ ) and uses 5 ms of CPU time

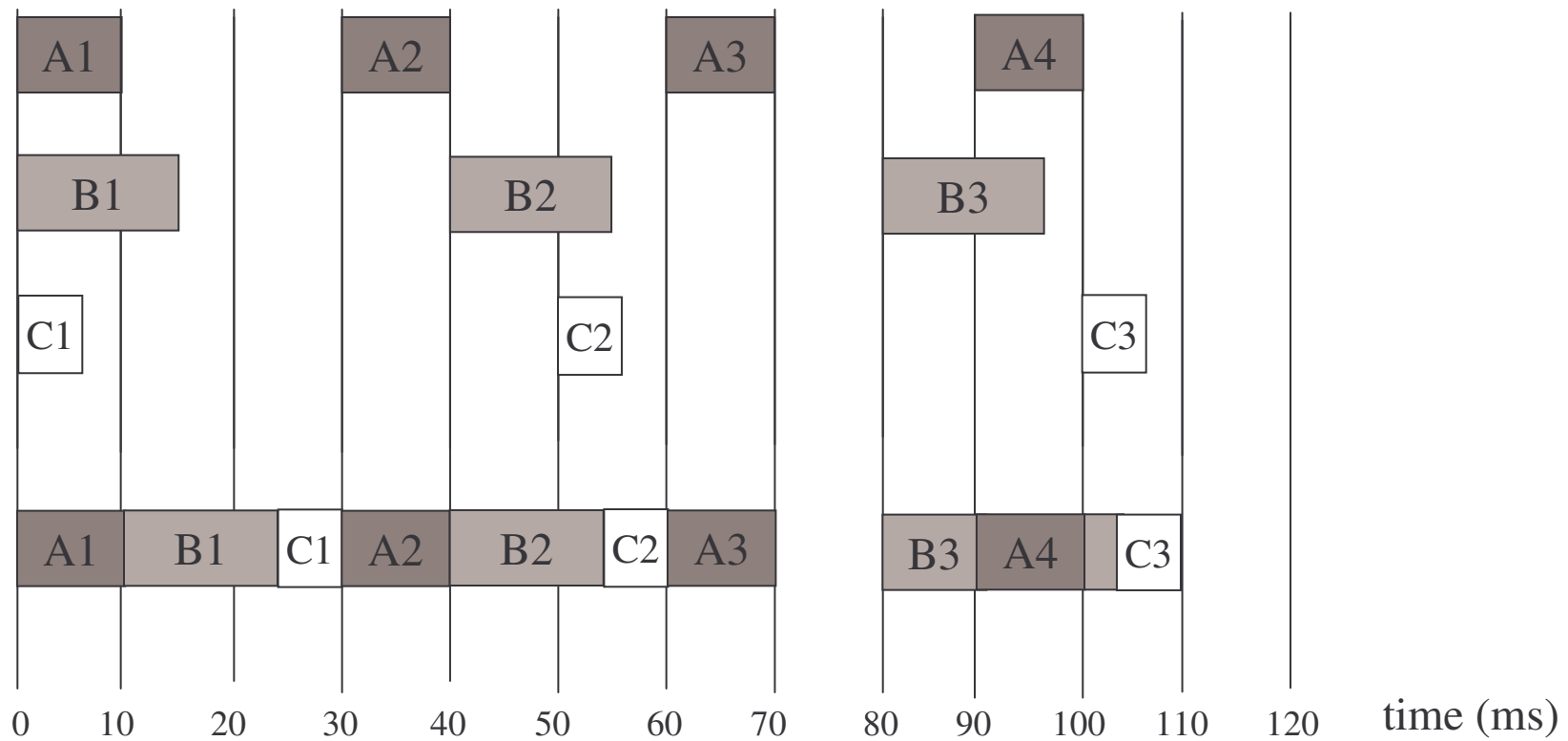
Schedulability ?  $10/30 + 15/40 + 5/50 = 0.808 < 1$



# Rate Monotonic Scheduling (RMS)

- Assumptions:
  - each periodic process must complete within its period
  - no process is dependent on any other process
  - each process needs the same amount of CPU time on each burst
  - any non periodic processes have no deadlines
  - preemption has no overhead
- Assign each process a fixed (static) priority equal to the frequency of occurrence of its triggering event (priorities are linear with the rate)

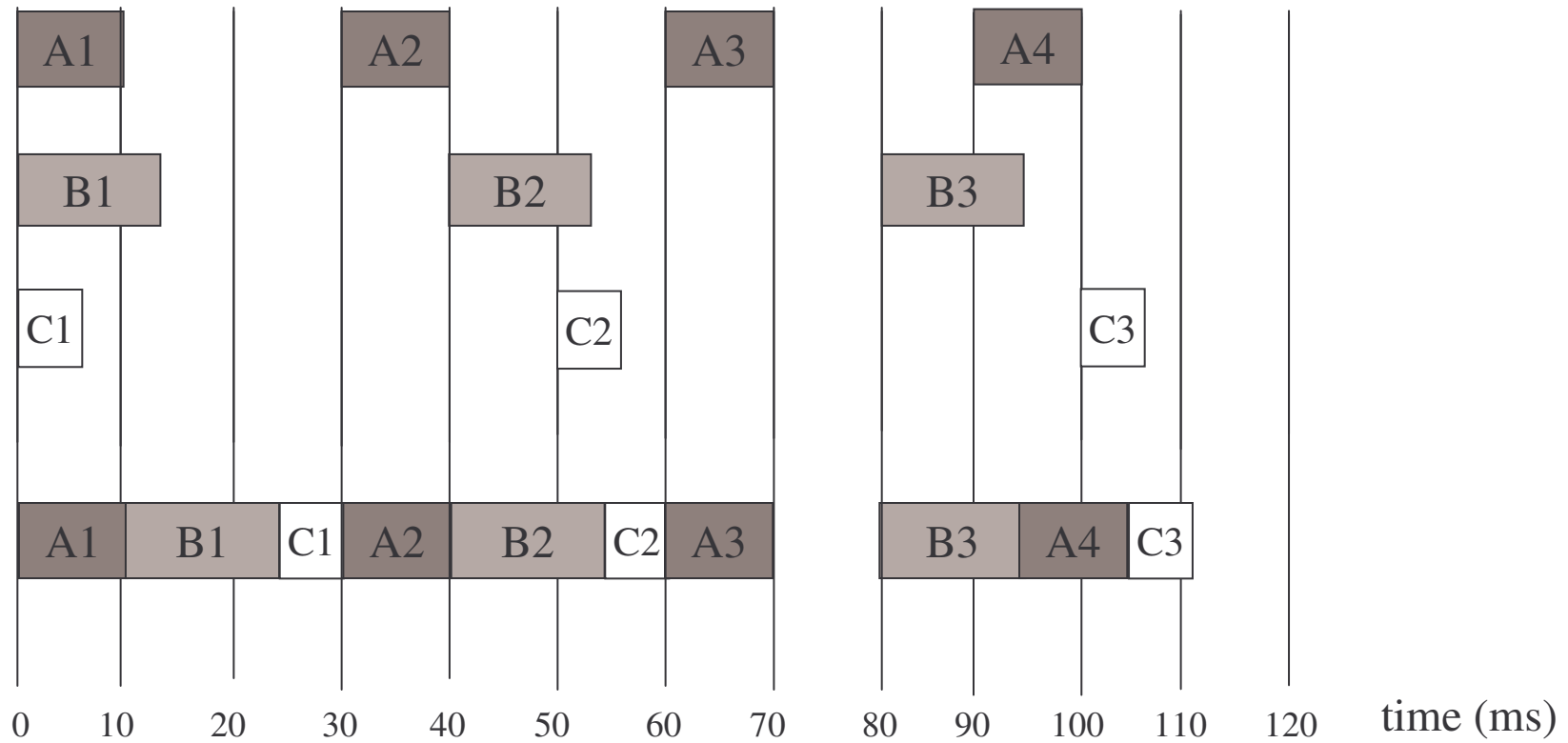
# Example: Rate Monotonic



# Earliest Deadline First Scheduling (EDF)

- Assumptions:
  - the same as rate monotonic but
  - it doesn't require processes to be periodic
  - processes can use different amounts of CPU for different bursts
- runnable processes are kept in a list with their deadline
- the scheduler runs the process with the closest deadline
- preempts the current process if another one with a closer deadline is ready

# Example: Earliest Deadline First



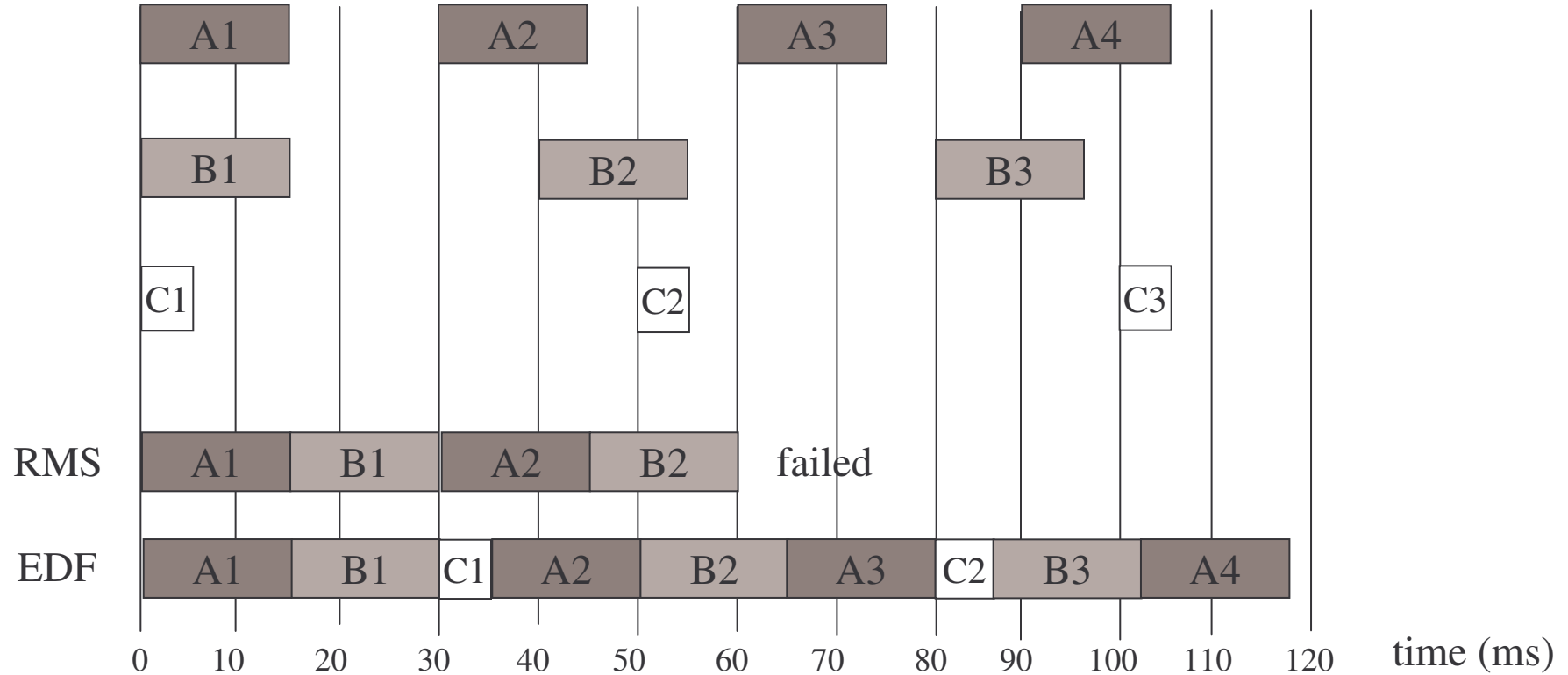
# RMS versus EDF

RMS uses static priorities and fails if CPU utilization is too high.

EDF always works if CPU utilization is  $< 100\%$

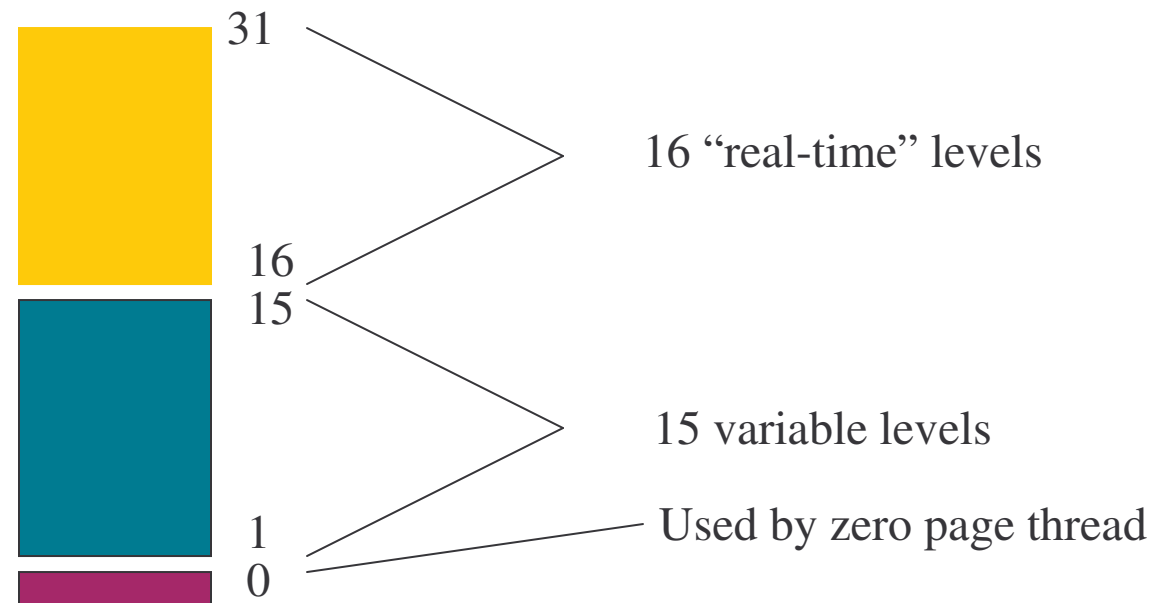
now A takes 15 ms of CPU time to complete

Schedulability ?  $15/30 + 15/40 + 5/50 = 0.975 < 1$



# Case study: scheduling in win32

- Only threads are scheduled, not processes
- Time-sliced, round robin with priorities
- Threads have priorities 0 through 31



# How are priorities assigned ?

## Win32 Process Classes

### Win32 Thread Priorities

	Real Time	High	Above Normal	Normal	Below Normal	Idle
Time-critical	31	15	15	15	15	15
Highest	26	15	12	10	8	6
Above-normal	25	14	11	9	7	5
Normal	24	13	10	8	6	4
Below-Normal	23	12	9	7	5	3
Lowest	22	11	8	6	4	2
Idle	16	1	1	1	1	1

# Priority Boost

- dynamic boost ( $< 15$ )
  - foreground threads get doubled time slice
  - if resumed by keyboard/mouse + 6
  - if resumed on wait +1
- decay: after boost priority is reduced of one level until it reaches base priority (the priority before boost)

# CPU Starvation

- Balance Set Manager (priority 16, every second)
  - looks for “starved thread” that have been ready for more than 4 seconds
- Special boost:
  - set priority to 15
  - doubled quantum
- Apply only to non realtime threads