EE5903 RTS
Chapter 2
Task Scheduling Basics

Bharadwaj Veeravalli

elebv@nus.edu.sg

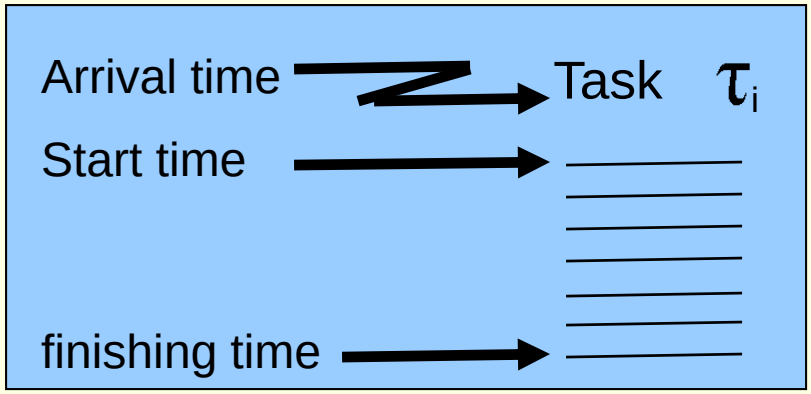
Acknowledgements: Parts of this material from Hard Real-Time Computing Systems, G.C, Buttazzo, Springer, 3rd Edition

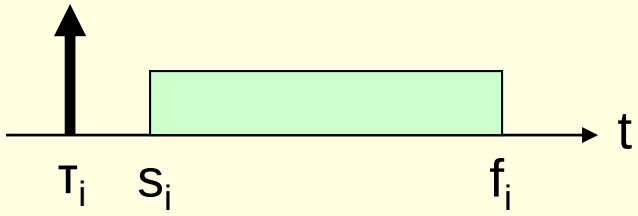
Outline – Task Scheduling

- Task specifications, definitions, examples of RTs
- RT task characteristics & task models,
- Timing & Precedence Constraints
- Resource and Resource Constraints
- Scheduling problem & Taxonomy
- Classification of RT Scheduling algorithms
- Classical scheduling policies – Some Real-time/Non-RT scheduling policies
- Scheduling a Bag of Tasks

Some useful definitions

- **Process (**or **task)**
  - is a sequence of instructions that in the absence of other activities is continuously executed by the processor until completion.

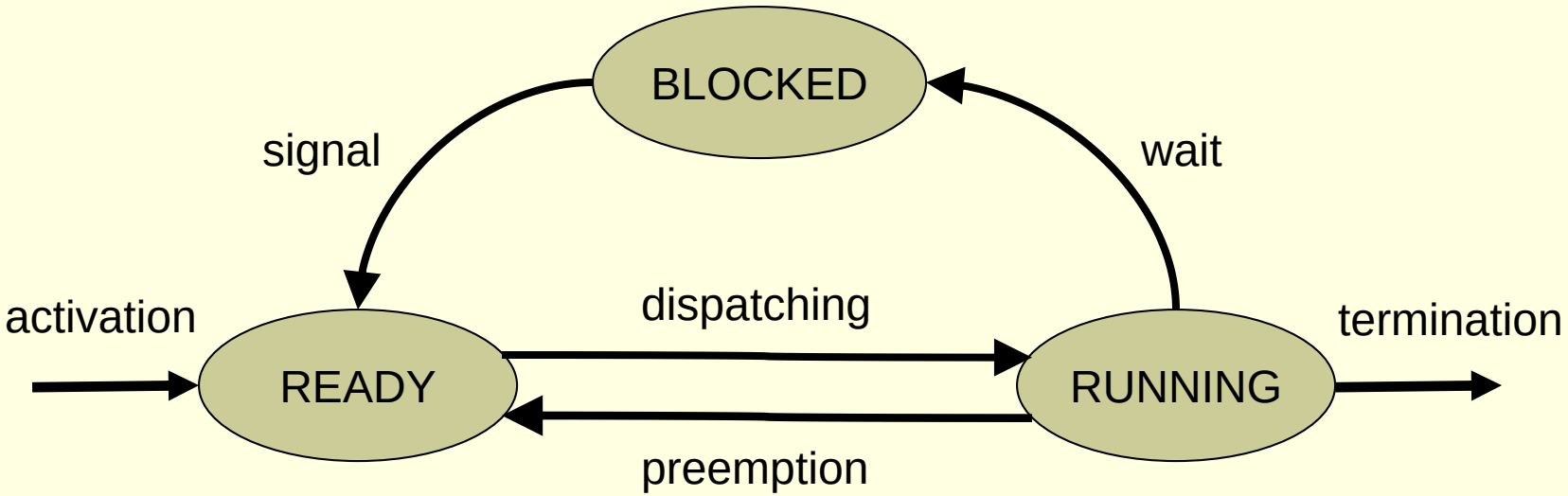




Task states

- A task is said to be:
  - **ACTIVE**: if it can be executed by the CPU;
  - **BLOCKED**: if it is waiting for an event;
- An **active** task can be:
  - **RUNNING**: if it is being executed by the CPU;
  - **READY**: if it is waiting for the CPU.

Task State Transitions

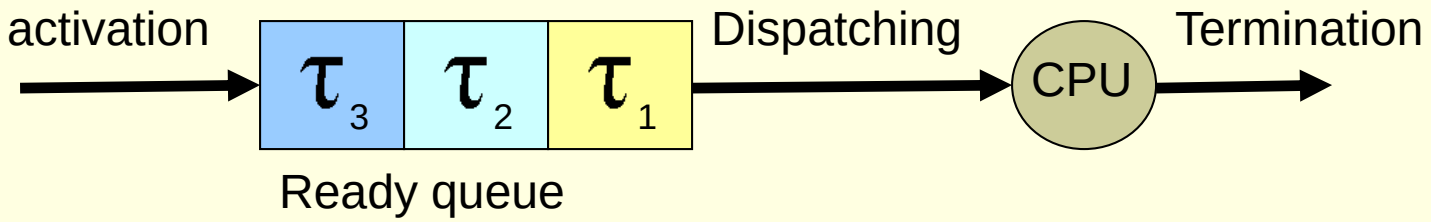


Generic task transition diagram

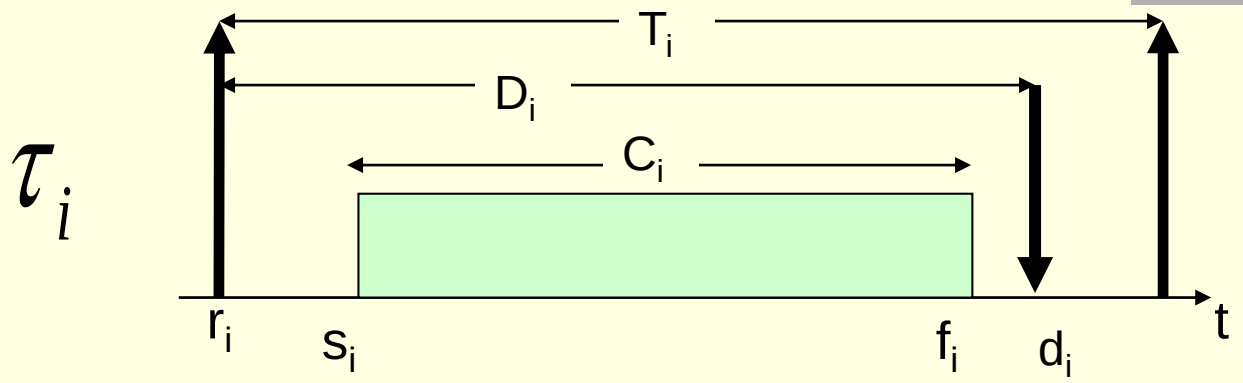
Ready Queue

- The **ready** tasks are kept in a waiting queue, called the **ready queue**;

- A strategy for choosing the ready task to be executed on the CPU is the **scheduling algorithm**.

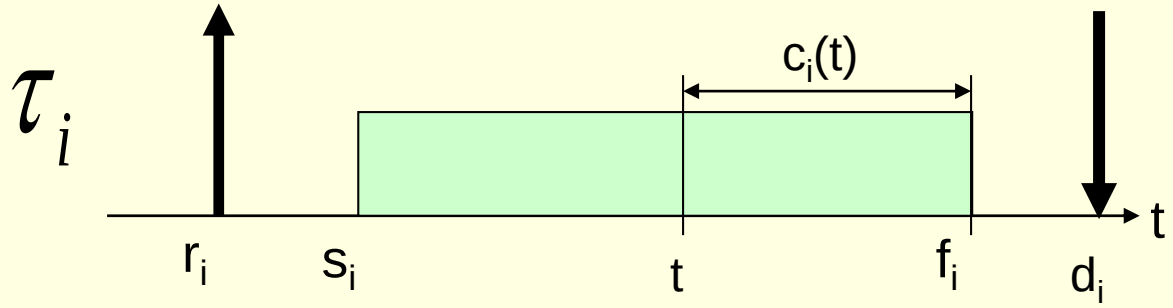


Real-Time tasks – Typical parameters



- T_i inter-arrival time (period) – applicable to periodic tasks
- **r_i** request time (arrival time **a_i**)
- **s_i** start time
- **C_i** worst-case execution time (wcet / avg-cet (acet))
- **d_i** absolute deadline
- **D_i** relative deadline (relative to the arrival time)
- **f_i** finishing time
- Response time = finish time – request time

Other parameters



- **Lateness**: $L_i = f_i - d_i$

Tardiness: $\max(0, L_i)$

(Exceeding time – Time a task stays active after its deadline)

Laxity: Maximum time a task can be delayed on its activation to complete within its deadline; Laxity = $(d_i - r_i - C_i)$

Laxity (or Slack) plays a crucial role in deciding schedulability/acceptance of a generated schedule for tasks, especially while handling aperiodic tasks.

Task Criticality

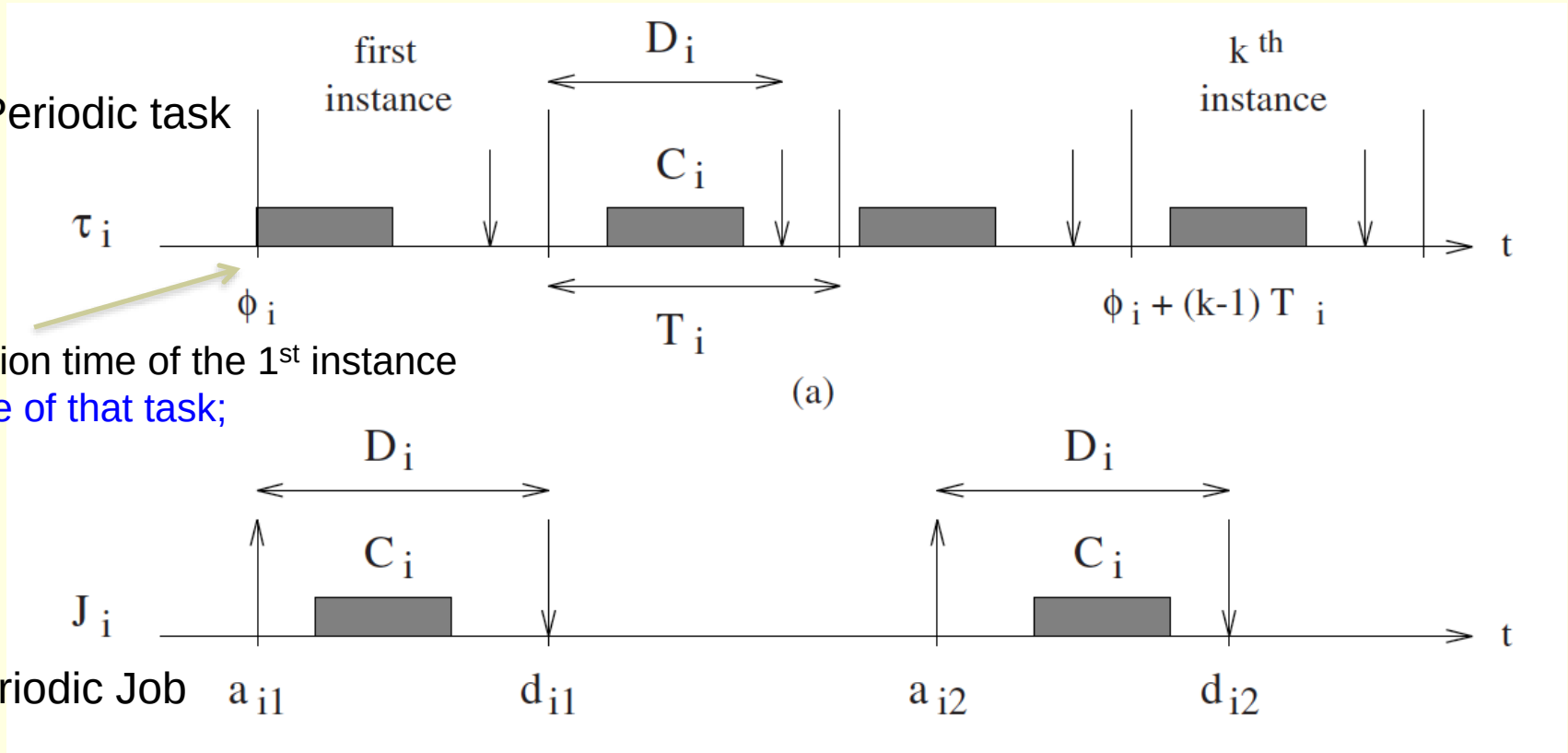
- **HARD real time tasks**
  - Missing a deadline is highly undesirable
  - May cause catastrophical effects on the system
- **FIRM real time tasks**
  - Missing a deadline is highly undesirable
  - Same as Hard RT, except safety is not an issue
- **SOFT tasks**
  - Occasional miss of a deadline is tolerable
  - Causes a performance degradation

An operating system able to handle hard RT tasks is called a **hard real-time** system.

Activation modes

- **Time driven**: **periodic** tasks
  - Task is automatically activated by the kernel at regular intervals.
- **Event driven**: **aperiodic/ sporadic** tasks
  - Task is activated upon the arrival of an event or through an explicit invocation of the activation primitive.
  - Sporadic – Consecutive tasks have *known minimum inter-arrival times*; often done for worst case calculations for aperiodic tasks;

Periodic & Aperiodic Tasks



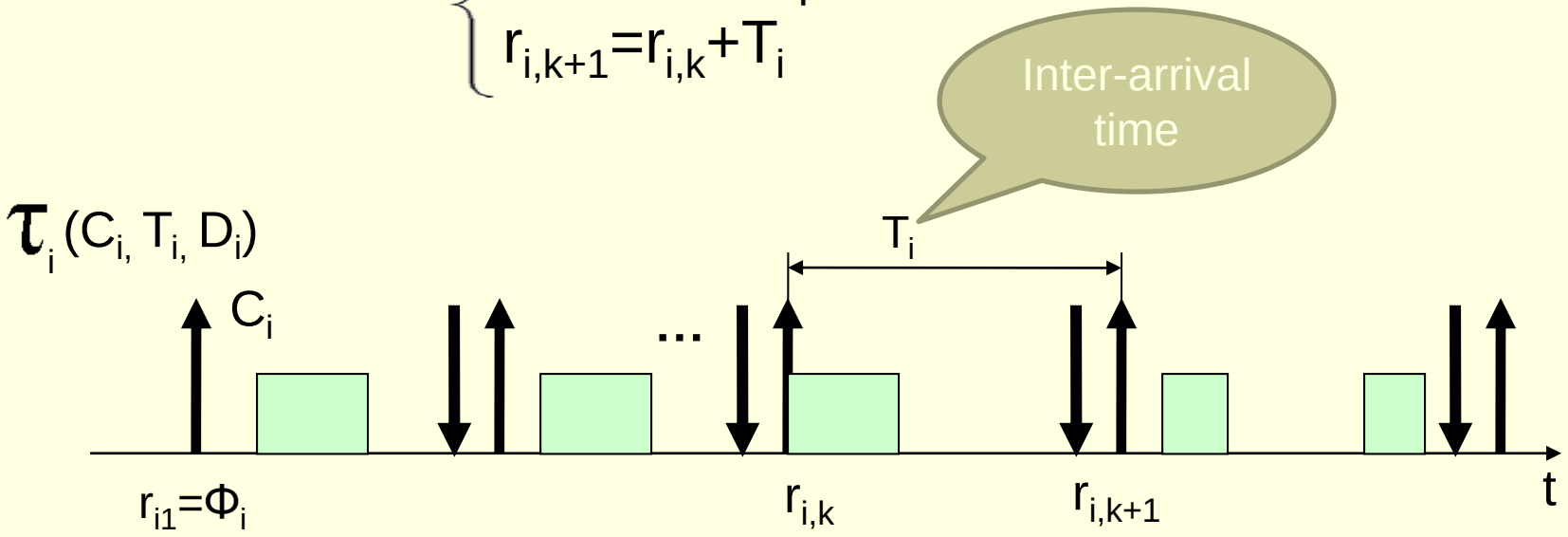
Activation time of the 1st instance - phase of that task;

Periodic tasks consist of an infinite sequence of identical activities, called instances or jobs , that are regularly activated at a constant rate.

Periodic task model

$$\begin{cases} r_{i1} = \Phi_i \\ r_{i,k+1} = r_{i,k} + T_i \end{cases}$$

Phase – Activation time of the first instance of a periodic task i



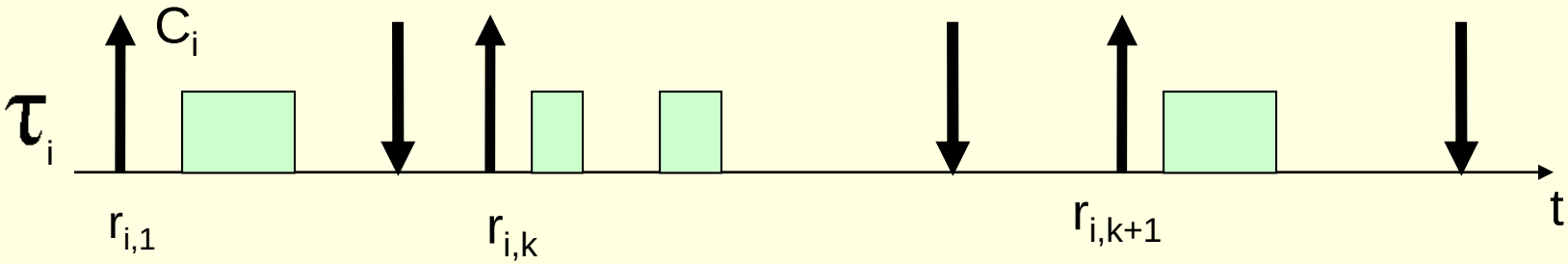
Activation time of the k-th instance of the periodic task: r_{ik}

$$r_{i,k} = \Phi_i + (k-1)T_i$$
$$d_{i,k} = r_{i,k} + D_i$$

[often $D_i = T_i$ but need not be true always]

Aperiodic task model

- **Aperiodic:** $r_{i,k+1} > r_{i,k}$
- **Sporadic:** $r_{i,k+1} \geq r_{i,k} + T_i$



Aperiodic tasks also consist of an infinite sequence of identical jobs (or instances); however, their activations are not regularly interleaved.

An aperiodic task where consecutive jobs are separated by a minimum inter-arrival time is called a sporadic task .

Types of Real-time constraints

- **Timing constraints**
  - Activation, completion, jitter.
- **Precedence constraints**
  - They impose an ordering in the execution
- **Resource constraints**
  - They enforce a *synchronization* in the access of mutually exclusive resources.

Timing constraints

- Can be explicit or implicit.
  - **Explicit constraints**
    - Usually included in the specification of the system activities, in the context of the problem.
  - Examples
    - CPU signals to open the valve **in** 10 seconds
    - Send the position information **within** 40 ms
    - Read the altimeter buffer **every** 200 ms

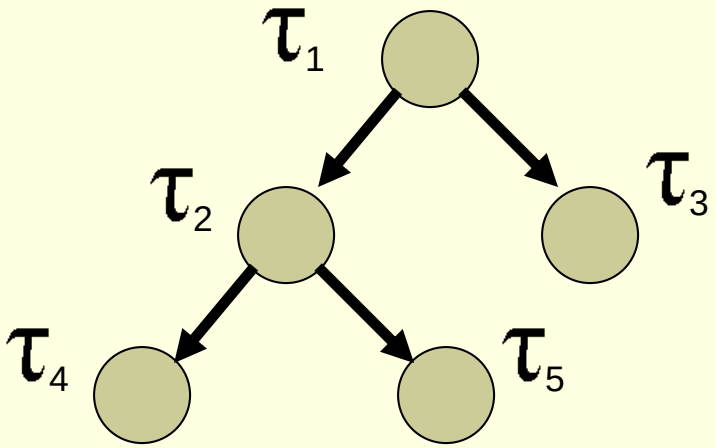
Timing constraints

- **Implicit constraints**
  - Do not appear in the system specification, but must be respected to meet the requirements

- Example
  - Avoid obstacles while driving at speed **v**; In this case, the speed has to be carefully calculated such that all obstacles are avoided

Precedence constraints

- Sometimes tasks must be executed with specific precedence relations, specified by a **Directed Acyclic Graph**:



predecessor

$\tau_1 < \tau_4$

Immediate predecessor

$\tau_1 \rightarrow \tau_2$

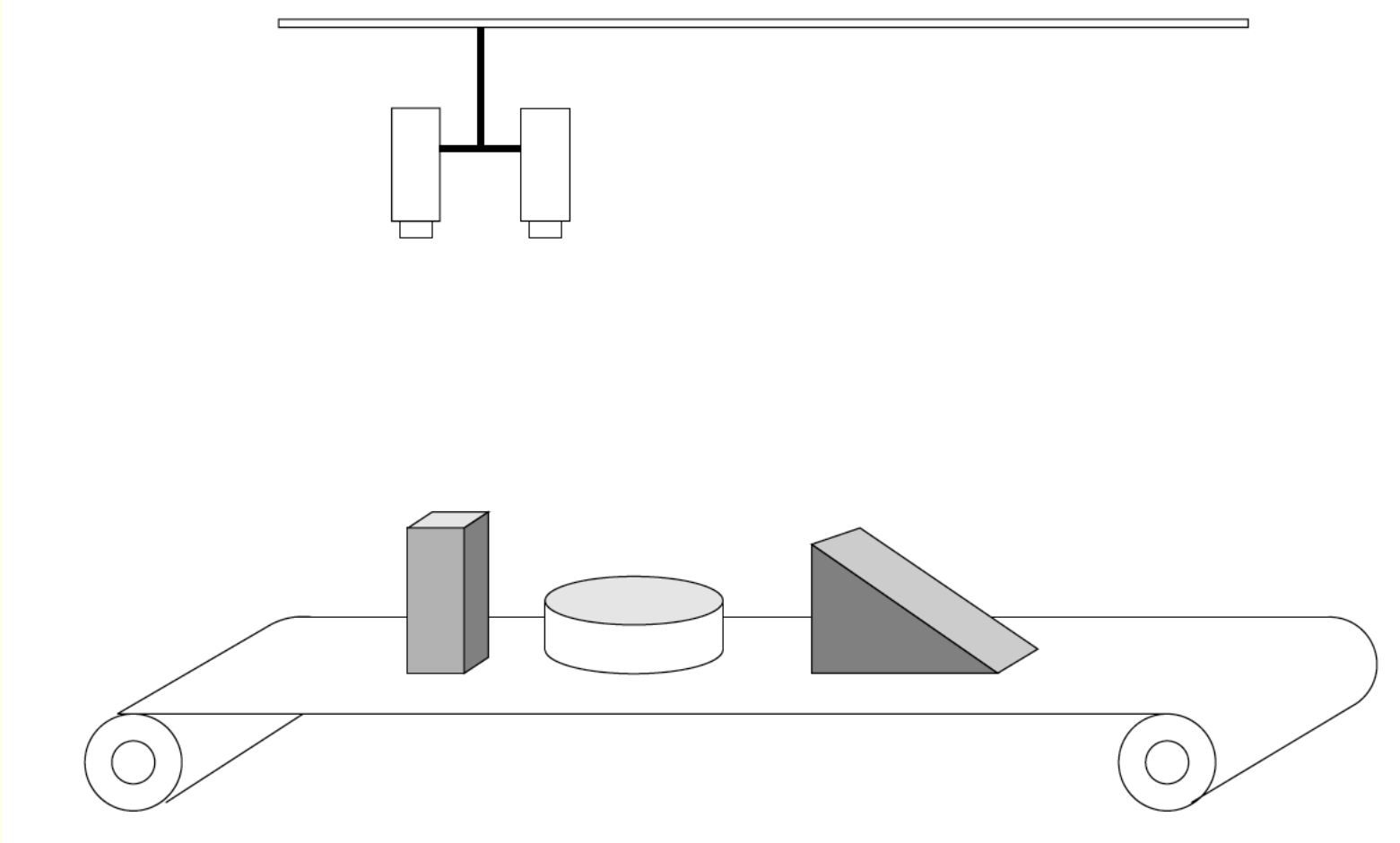
How does an application specific DAG gets generated?

- What is a DAG in a given context and how it is formed? – Organization of computational activities pertaining to an application

- Camera based automatic device inspection in an industrial environment – a RT challenge!

Number of objects moving on a conveyor belt must be recognized and classified using a stereo vision system, consisting of two cameras mounted in a suitable location.

How does an application specific DAG gets generated?



Stereo vision system – DAG generation

- Objective : To recognize the objects moving on the conveyor belt.

The recognition process is carried out by integrating the 2D features of the top view of the objects;

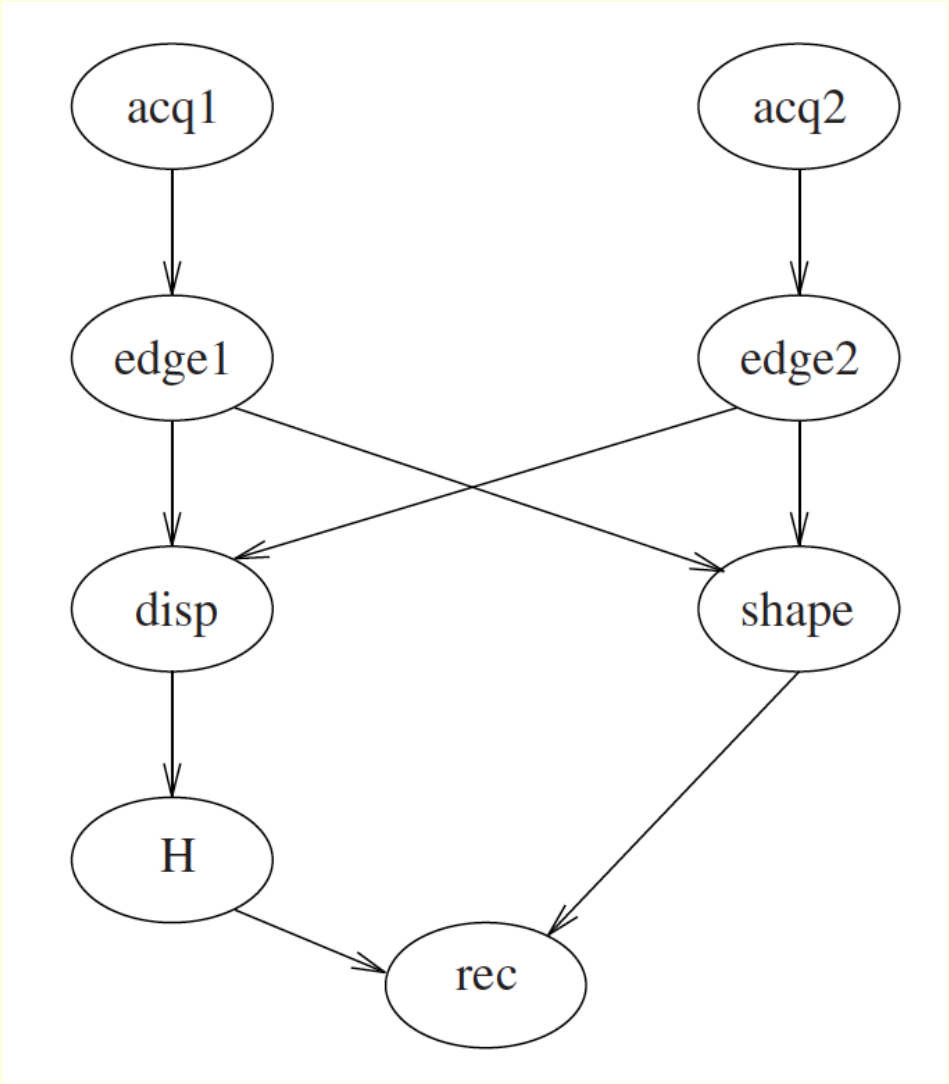
Height information of the objects - Extracted by the computing the pixel disparity on the two images.

As a consequence, the computational activities of the application can be organized by defining the following tasks:

How does an application specific DAG gets generated?

- Two tasks (one for each camera) dedicated to image acquisition, whose objective is to transfer the image from the camera to the processor memory (they are identified by *acq1* and *acq2*);
- Two tasks (one for each camera) dedicated to low-level image processing (typical operations performed at this level include digital filtering for noise reduction and edge detection; we identify these tasks as *edge1* and *edge2*);
- A task for extracting two-dimensional features from the object contours (it is referred as *shape*);
- A task for computing the pixel disparities from the two images (it is referred as *disp*);
- A task for determining the object height from the results achieved by the *disp* task (it is referred as *H*);
- A task performing the final recognition (this task integrates the geometrical features of the object contour with the height information and tries to match these data with those stored in the data base; it is referred as *rec*).

How does an application specific DAG gets generated?



Data acquisition

Edge detection

Disp & Shape detection

Height detection

Recognition

Resource & Resource Constraints

- **Resource** – S/w or sometimes H/w structure used by a process to *advance* its execution

- Examples: A resource can be a data structure, a set of variables, a main memory area, a file, a piece of program, or a set of registers of a peripheral device.

- Private or Shared

Resource & Resource constraints

- To maintain data consistency, many shared resources do not allow simultaneous accesses by competing tasks, but require their *mutual exclusion*. This means that a task cannot access a resource R if another task is inside R manipulating its data structures.

- In this case, R is called a mutually exclusive resource. A piece of code executed under mutual exclusion constraints is called a *critical section*.

More on these in rigor in Chapter 6. Stay tuned!

Resource & Resource constraints – data consistency & use of ME

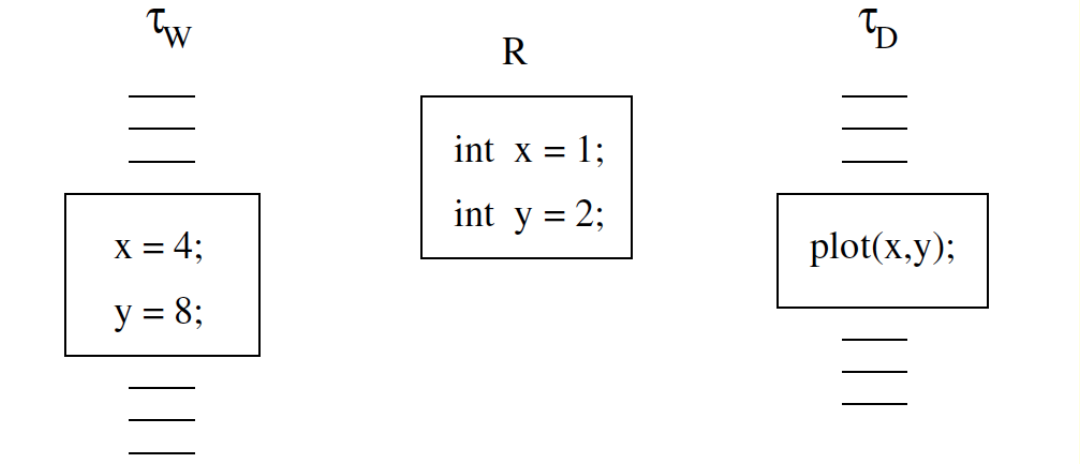
- Example: Object tracking system

Two tasks cooperate to track a moving object

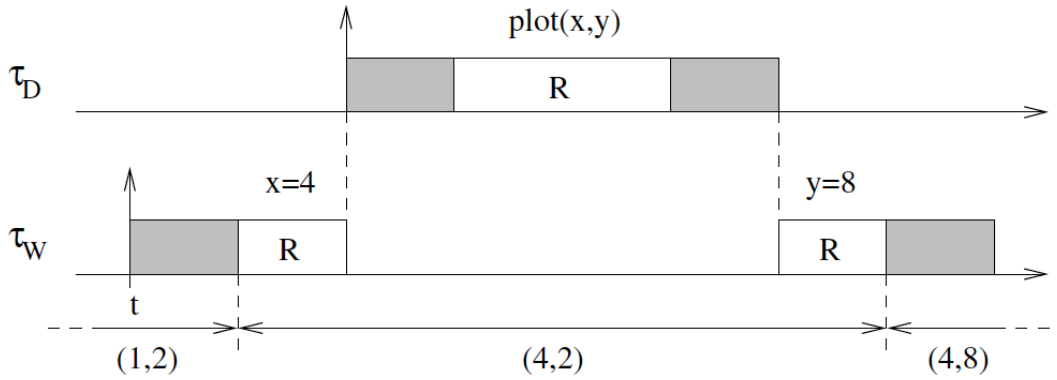
Task τ_W gets the object coordinates from a sensor and writes them into a *shared buffer R* , containing two variables (x, y);

Task τ_D reads the variables from the buffer and plots a point on the screen to display the object trajectory.

Resource constraints – Use of ME Principle - Consistency



Two tasks sharing a buffer with two variables.



Example of schedule creating data inconsistency.

R – Buffer –
Shared resource;

W is a low-priority task than D and can be Preempted by D;

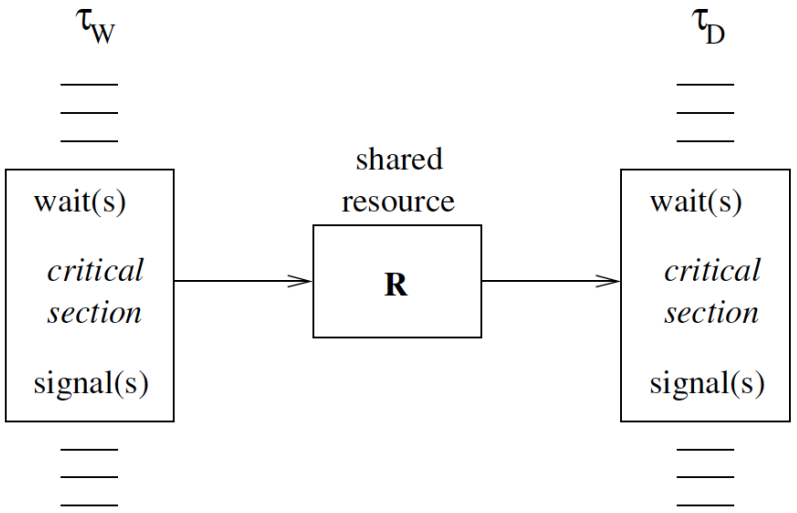
(1,2) – Current (x,y)

(4,8) – Correct (x,y)

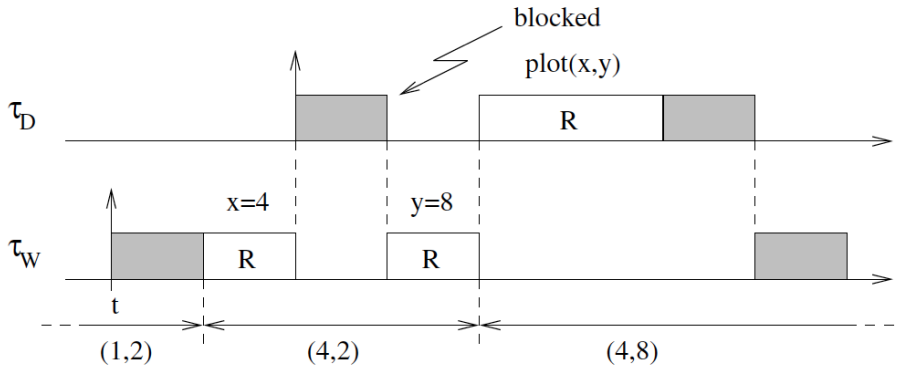
((4,2) – Result of no MEP

Mutual exclusion is desired

Mutual exclusion



s: Semaphore variable; wait(s) and Signal(s) are primitive/atomic functions

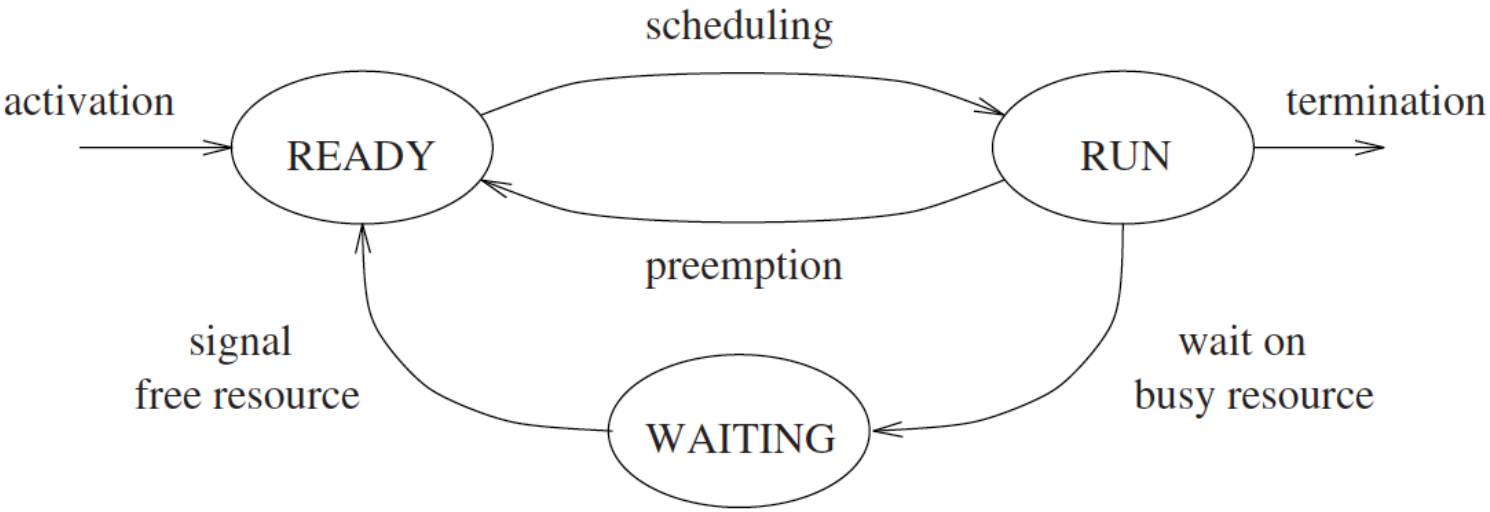


Example of schedule when the resource is protected by a semaphore.

Note that ME introduces Some extra delays!

No choice, but to live with it!

Mutual exclusion…(cont'd)



Waiting state caused by resource constraints

Task waiting for a shared resource implies BLOCKED on that resource;

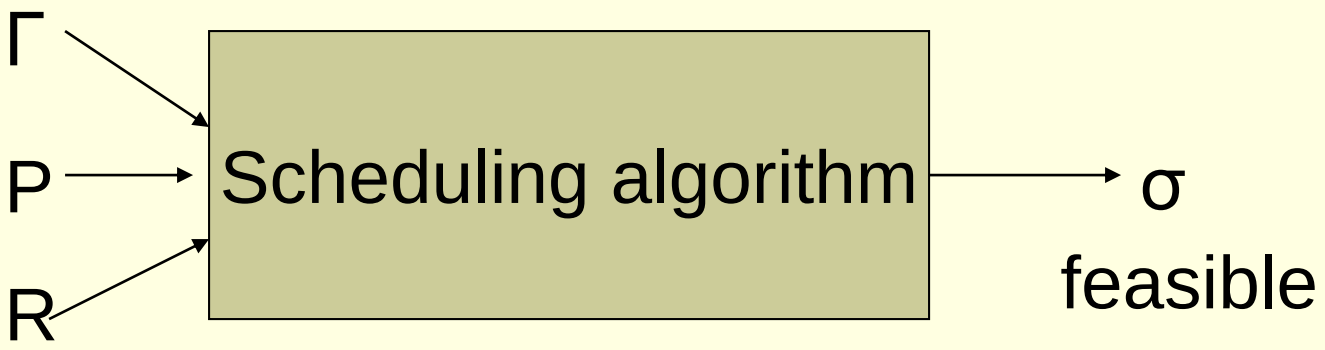
Running task -> Waiting -> Ready (scheduling algorithm decides)

General Definitions - Scheduling

- A **schedule** is a particular assignment of tasks to the processor in time.
- A set of tasks Γ (Gamma) is said to be **schedulable** if there exists a feasible schedule for it.
- A schedule σ is said to be **feasible** if all the tasks are able to complete within a set of constraints.

The general scheduling problem

- Given a set Γ of n tasks, a set **P** of m processors, and a set **R** of r resources, find an assignment of **P** and **R** to Γ which produces a feasible schedule.



Complexity

- In 1975, Garey and Johnson showed that the general scheduling problem is NP hard.
- However, polynomial time algorithms can be found under particular / application specific conditions – Greedy algorithms!
- Greedy algorithms – Exploit certain key characteristics of the problem / application to derive near/sub-optimal solutions;

  -- Important to derive how far you are away from an optimal solution – Quality of the solution generated by your greedy algorithm.

Simplified & *tough* assumptions

- Single CPU/core/thread / Multi-core
- No platform related delays / Delays exist
- Homogeneous task sets / Heterogeneous
- Fully preemptive tasks / Hybrid set
- Simultaneous activations / Aperiodic/sporadic
- No precedence constraints / Hybrid
- No resource constraints / Resource constraints present

- In case of DAGs – WCET / ACET are known (obtained from task profiling exercise;) / + time varying execution times

Scheduling algorithm taxonomy

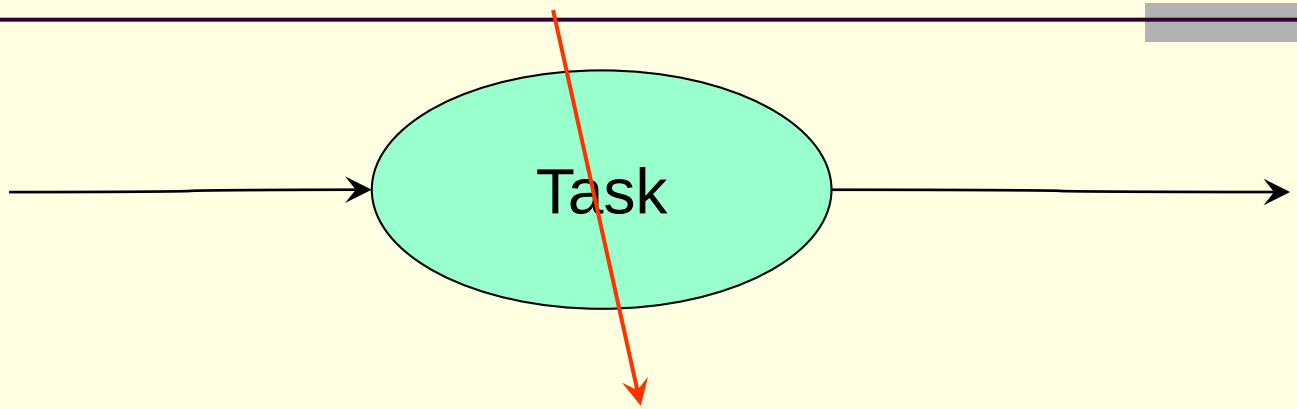
- Preemptive vs. Non Preemptive
- Static vs. dynamic
- On line vs. Off line
- Best Effort vs. Optimal

Preemptive, Non-preemptive & Deferred scheduling

- A scheduling algorithm is said to be:
  - **preemptive**: if the running task can be temporarily suspended in the ready queue to execute a more important task.
  - **non preemptive**: if the running task cannot be suspended until completion.
  - **deferred preemptive**: the running task is allowed to run until a bounded time

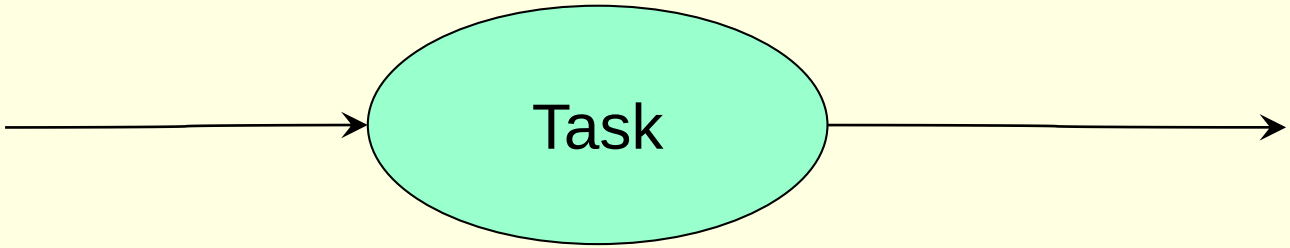
Preemptions – Often guided by priority levels of the tasks;

Preemptive Systems



- Interrupt a task when a higher priority task wants to execute
- For a large class of scheduling problems, it is shown that PSs are easy to manage in providing timing guarantees (*although not proven!*)
- Higher overhead of switching and memory
- Cache pollution; Can be resolved by constraining the amount of cache the prefetched data can occupy or via software techniques;

Non-Preemptive Systems



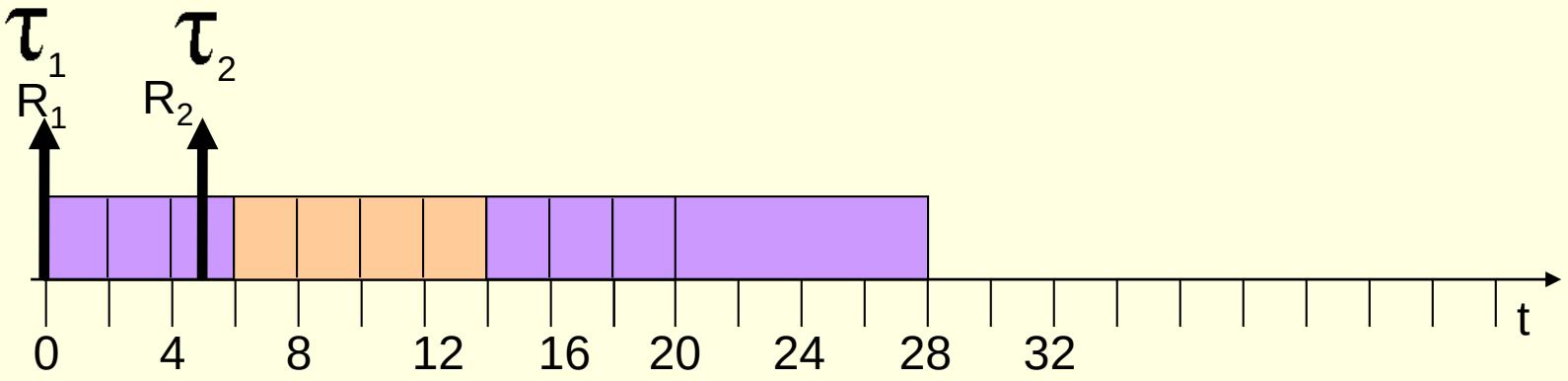
- State-space needed is smaller
- Lower implementation cost
- Less overhead at run-time
- Less cache pollution, smaller memory needed

Deferred Preemptive

- When a high priority task arrives, the low priority task is allowed to continue for a bounded time (not necessary towards completion)

- Essentially each task can be considered to be composed of small non-preemptible tasks – each small task with a bounded time

Deferred Preemptive



Task 1 – current task
Task 2, a high priority task arrives;

Static vs. Dynamic

- **Static**
  - Scheduling decisions are taken based on fixed parameters (e.g. computation time), statically assigned to tasks before activation.
- **Dynamic**
  - Scheduling decisions are taken based on parameters that can change with time (e.g. arrival time, exe time, etc).

Off-line vs. On-line

- **Off-line**
  - All scheduling decisions are taken before task activation: the schedule is stored in a table.
- **On-line**
  - Scheduling decisions are taken at run time on the set of active tasks.

Best-Effort vs. Optimal

- **Best-Effort**
  - Do their best to find a feasible schedule, if there exists one, but they do not guarantee if a best schedule would be generated.
- **Optimal**
  - Always attempts to determine an optimal schedule, if there exists one;

Classification of RT Scheduling Algorithms

- Classification schemes are based on the following three categories:

1) Based on how scheduling points are defined

1) Based on the type of task acceptance tests

1) Based on the target platform used

Classification of RT Scheduling Algorithms

(1) Three possible scheduler designs exist on how the scheduling points are defined:

(a) Clock-driven - Scheduling points are determined by the interrupts received from a clock

(b) Event-driven - Scheduling points are defined by certain events which precludes clock interrupts

(c) Hybrid - Uses both clock interrupts and event occurrences to define their scheduling points

RT Scheduling Classification… (cont'd)

- Clock Driven – simple and efficient; used frequently in several embedded applications
  - -- Table Driven schedulers
  - -- Cyclic schedulers
- Event Driven
  - -- Simple Priority based schedulers
  - -- Rate Monotonic Algorithm (RMA) *(Chapter 4)*
  - -- Earliest Deadline First (EDF) *(Chapter 4)*
- Hybrid
  - -- Round-robin

RT Scheduling – Clock-driven algorithms

Clock-driven schedulers are off-line / static schedulers because scheduling points are determined by clock timer interrupts.

That is, even before the schedule starts the scheduler decides which task to run, when, and hence makes them unsuitable for RT application needs (especially in handling sporadic and aperiodic tasks).

For periodic tasks, with known periods, Table-driven schedulers can be used (as the theorem on Slide 48 demonstrates).

RT Scheduling – Clock-driven algorithms: table-driven scheduler

Table-driven schedulers usually precompute a schedule and keep them ready. As an example, with 5 tasks, we have,

Task	Start time of execution
T1	0
T2	3
T3	7
T4	10
T5	15

Question: How does a table-driven scheduler handle periodic tasks (as the periods are usually known apriori)?

Example: If p_1=20, p_2=100, p_3=250, then the major cycle for this set of periodic tasks is LCM(p_1,p_2,p_3) = 500. Since LCM can be decided, table-driven schedulers can handle period tasks with known periods.

e_1=5msecs, e_2=10msecs e_3=5msecs

Table-driven schedulers

- In general:

T_i = (phase, period, exe.time, relative deadline)

T_i = (1,10,3,6) implies:

Task 1st instance has a phase 1, released at t=1, must be completed by t=7;

Task 2nd instance will be released at t=11and must be completed by 17, and so on

Table-driven schedulers

- If default phase value = 0:

T_i = (period, exe.time, relative deadline)

T_i = (10,3,6) implies:

Task instance 1 released at t=0, must be completed by t=6;

Task instance 2 will be released at t=10 and must be completed by 16, and so on

Table-driven schedulers

- If default phase value = 0 & relative deadline is the period of the task:

T_i = (period, exe.time)

T_i = (10,3) implies:

Task instance 1 released at t=0, must be completed by t=10;

Task instance 2 will be released at t=10 and must be completed by 20, and so on

Table-driven schedulers

- When all the parameters of a task is known, static schedules of a task can be developed offline;
- CPU time allocated to the task is its maximum exe time
- Scheduler dispatches the tasks according to the static schedule constructed, an repeats at each major cycle
- Static schedules guarantee that each task completes by its deadline

Table-driven schedulers

- **Exercise:** Design a table-driven static scheduler for the following periodic tasks. Note that a scheduler always attempts to use a task that has less execution time first if it is available.

Given task set:

$T_1 = (4,1)$; $T_2 = (5,1.8)$; $T_3 = (20,1)$; $T_4 = (20,2)$; Clearly, the phases of the tasks are zero;

Draw a schedule using a timing diagram. Determine the table entries- show at least the top 8 entries using the timing diagram drawn.

RT Scheduling – Clock-driven algorithms: table-driven scheduler

LCM() is referred to as a major cycle. A major cycle is a period of time such that in each major cycle, the different tasks recur identically. But the following theorem also claims that regardless of task phasings the above result is true.

Theorem (Periodic tasks, Table-driven schedulers): The major cycle of a set of tasks $S = \{T_i, i=1,2,\ldots,N\}$ with respective periods $\{p_1,p_2,\ldots,p_N\}$ is given by $LCM(p_1,p_2,\ldots,p_N)$, regardless of task phasings.

Proof: ***(Classroom discussion)***

RT Scheduling – Clock-driven algorithms: cyclic scheduler

Cyclic schedulers – Majority of small embedded applications use this scheduler (Ex: temperature controller in computer controlled air-conditioners)

Characterized by periodic scheduling tasks – sampling period is fixed, that is, periodicity does not vary (unlike in earlier table-driven scheduler case of handling periodic tasks); So deciding a major cycle is straightforward;

Cyclic scheduler… (Cont'd)

A cyclic scheduler **repeats** a pre-computed schedule. Thus, we need to store the information for only one major cycle => less memory occupancy.

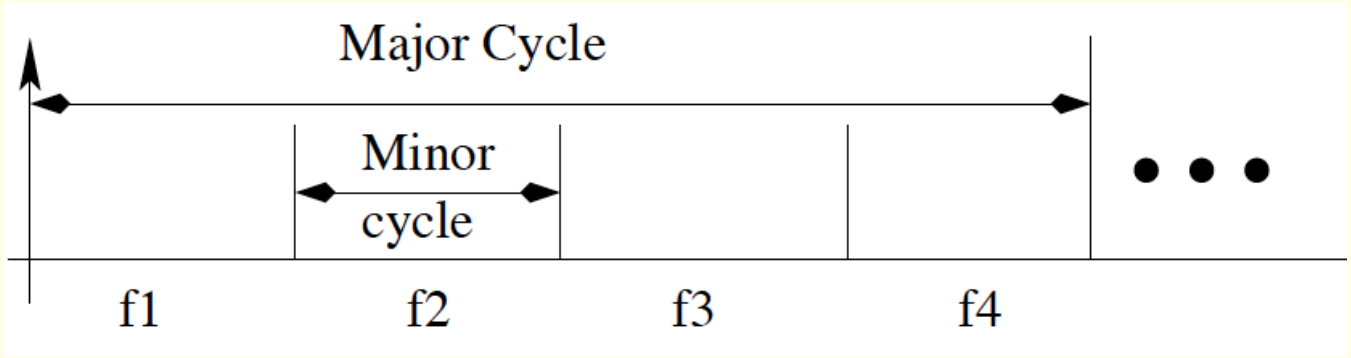
Each task in the current task set to be scheduled repeats identically in **every major cycle**. Refer to a fig - next slide. Each minor cycle is referred to as a **frame**.

Scheduling points of this scheduler occur at frame boundaries. That is, task starts to get executed at the beginning of the frame.

Frame boundaries are defined via interrupts generated by a periodic timer. Each task is assigned to run in one or more frames. A schedule table is constructed which captures the assignment of tasks to frames.

See an example in the next slide.

Cyclic scheduler…. (Cont'd)



Task Number	Frame Number
T3	F1
T1	F2
T3	F3
T4	F4

Key challenge is to decide on the **size of the frame**; Embedded devices need to consider the following three constraints in their design for the success of the product.

1. Context switching time
2. Minimization of schedule table size
3. Satisfaction of task deadline

Cyclic scheduler…. (Cont'd)

- Constraint 1: $\max\{e_i\} \leq F$ (lower bound on F)

To avoid unnecessary context switches, the frame size must be at least more than the largest execution time of a task;

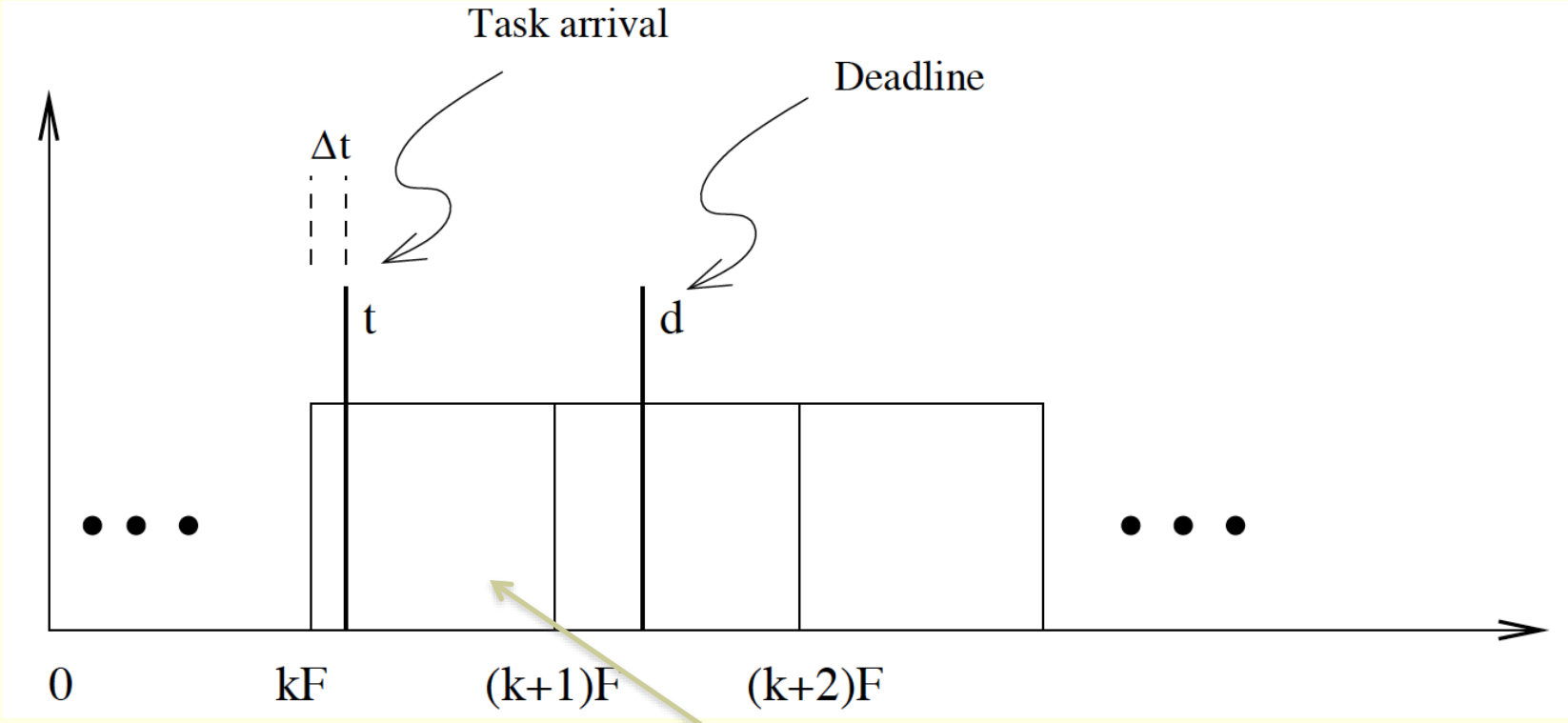
- Constraint 2: floor(M / F) assures integral # of frames in a given major cycle;

This ensures integral number of frames in a major cycle;

- Constraint 3: Consider the following scenarios:

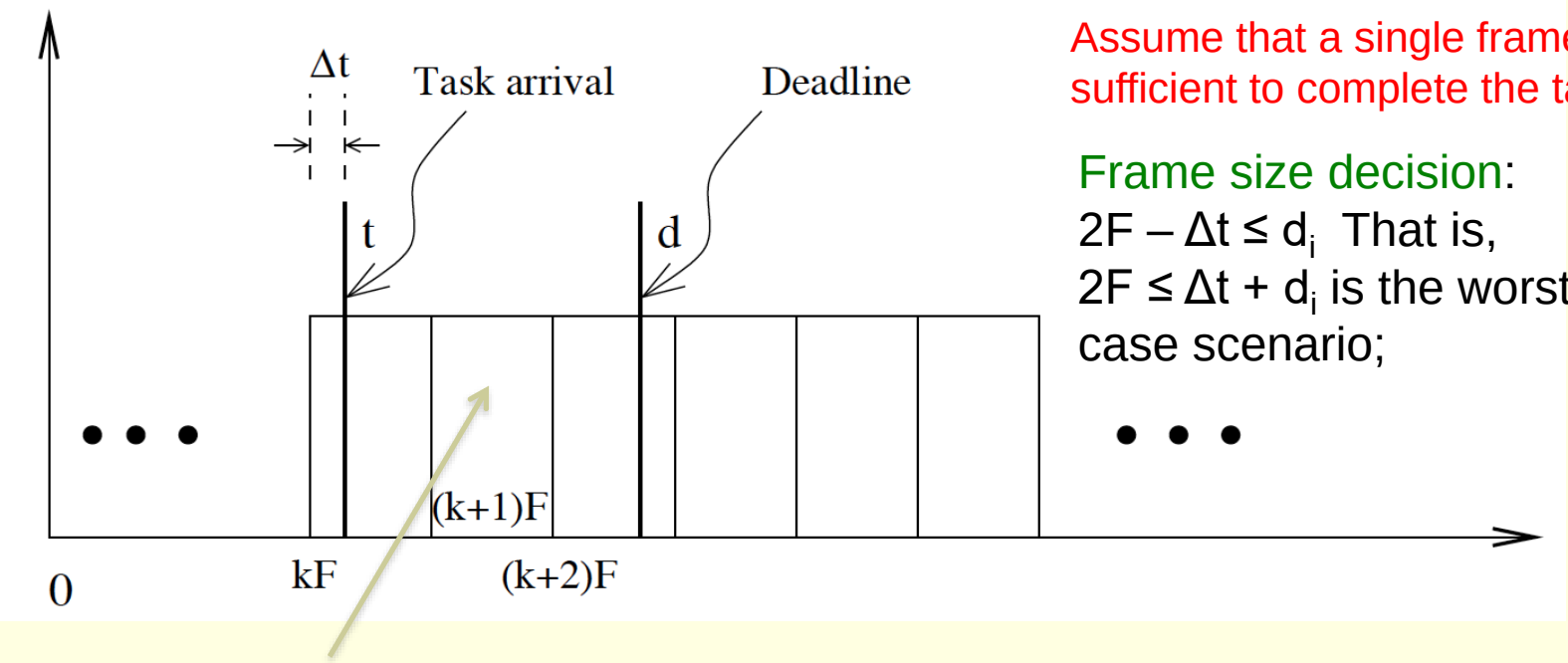
Task will be taken only from the start of a frame; A task may not arrive at the frame boundary, but it needs at least integral number of frames to execute; Consider a situation shown in the figure in the next slide.

Cyclic scheduler…. (Cont'd)



In this situation task may miss the deadline!

Cyclic scheduler … (Cont'd)



Assume that a single frame is sufficient to complete the task:

Frame size decision:
$2F - \Delta t \leq d_i$ That is,
$2F \leq \Delta t + d_i$ is the worst case scenario;

Full frame exists

Frame size decision: Note that not all $\Delta t > 0$ satisfy the deadline. In general, the $\min\{\Delta t\} > 0$ is given by,

$\Delta t = GCD\{F, p_i\}$. Thus, using the above criteria: $2F - GCD\{F, p_i\} \leq d_i$

Cyclic scheduler…. (Cont'd)

Thus, considering all tasks, the resulting criteria for frame size (F_R) is given by,

$$F_R < \max(\gcd\{F, p_i\} + d_i)/2$$ must satisfy for all tasks

- Exercise 2.1: Determine the frame size of the following periodic tasks using a cyclic scheduler.

$T_1 = (e_1=1, p_1=4)$ Deadline of the tasks are their periods;

$T_2 = (e_2=1, p_2=5)$

$T_3 = (e_3=1, p_3=20)$

$T_4 = (e_4=2, p_4=20)$

Example – Cyclic scheduler design: A practical approach

task	f	T	C
A	40 Hz	25 ms	10 ms
B	20 Hz	50 ms	10 ms
C	10 Hz	100 ms	10 ms

In a practical setting, one can simply use GCD and LCM to design a CS;

Δ = GCD (minor cycle) = 25 ms

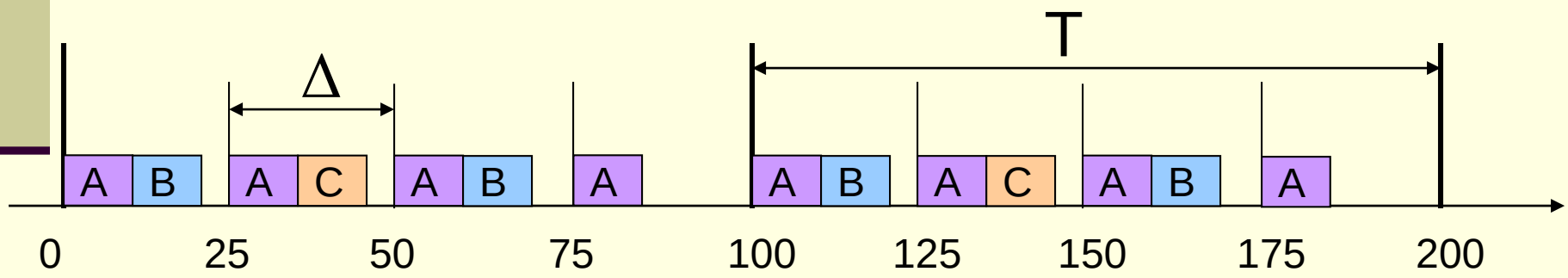
T = LCM (major cycle) = 100 ms

Example… (cont'd)

task	f	T	C
A	40 Hz	25 ms	10 ms
B	20 Hz	50 ms	10 ms
C	10 Hz	100 ms	10 ms

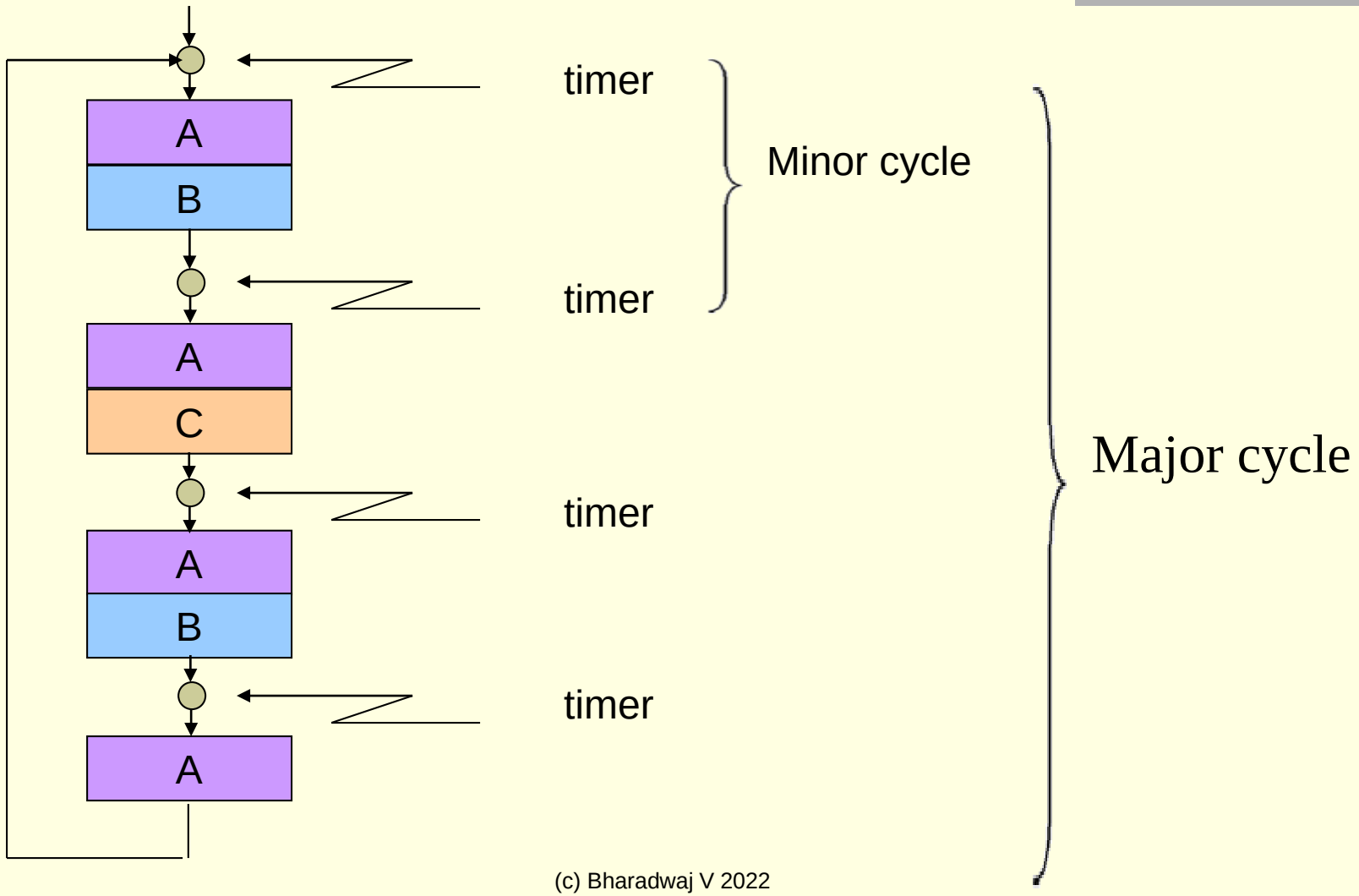
$\Delta = 25$ ms

$T = 100$ ms



Observe the repetition of the pattern betwee the major cycles

Implementation



Implementation Code

```
loop
        Wait_For_Interrupt;
        Procedure_For_A;
        Procedure_For_B;
        Wait_For_Interrupt;
        Procedure_For_A;
        Procedure_For_C;
        Wait_For_Interrupt;
        Procedure_For_A;
        Procedure_For_B;
        Wait_For_Interrupt;
        Procedure_For_A;
end loop;
```

Also known as synchronization points

Table-driven vs Cyclic schedulers

- In CS, a scheduler needs to set a periodic timer only once at the start of the application. This timer continues to give an interrupt exactly at every frame boundary.
- In TDS, a timer has to be set every time a task starts to run. So, a call to the timer needs to be triggered every few millisecs.

This is one of the overhead factors and hence CS is more preferred on embedded devices than TDS.

Table-driven vs Cyclic schedulers

What if this overhead is negligible?

- In CS, the lower bound on the size of the frame (max of exec times) may be a costly aspect, since if the task size of a task is much smaller than the frame size!

In this situation, TDS is preferred! It is truly a tie between timer triggered overhead in TDS and frame size in CS.

RT Scheduling Classification… (cont'd)

(2) Type of Task acceptance tests

Before a newly arrived task is taken up for scheduling task acceptance test is performed by the scheduler to decide to accept or reject.

Based on the test two types of schedulers exist:

(a) Planning based

(b) Best effort

(a) Deadline satisfaction together with all other currently ready tasks; If yes, take the current task for processing; If not, reject;

(b) No acceptance test; All tasks are taken up and best effort is made to meet its deadline; no guarantees;

RT Scheduling Classification… (cont'd)

(3) Classification using Target Platforms

(a) Uniprocessor

(b) Multiprocessor / Multi-core

(c) Distributed / network-based

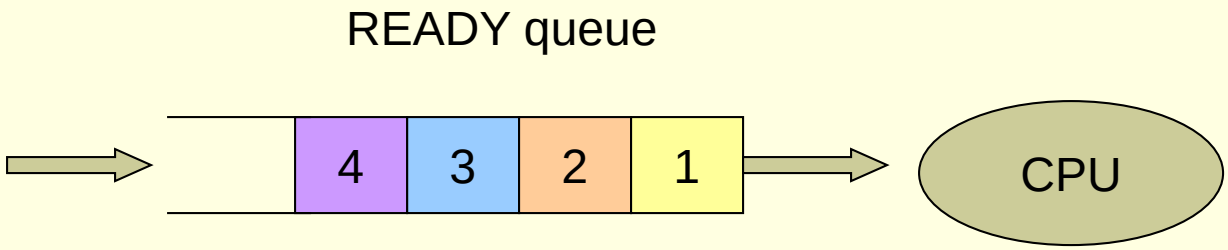
Common scheduling algorithms

- First Come First Served
- Shortest Job First
- Priority Scheduling
- Round Robin & its variant

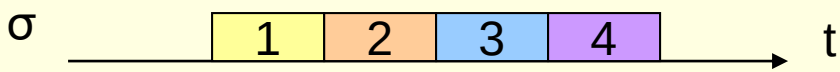
Not all are suited for real-time systems

First Come First Served

- It assigns the CPU to tasks purely based on their arrival times.

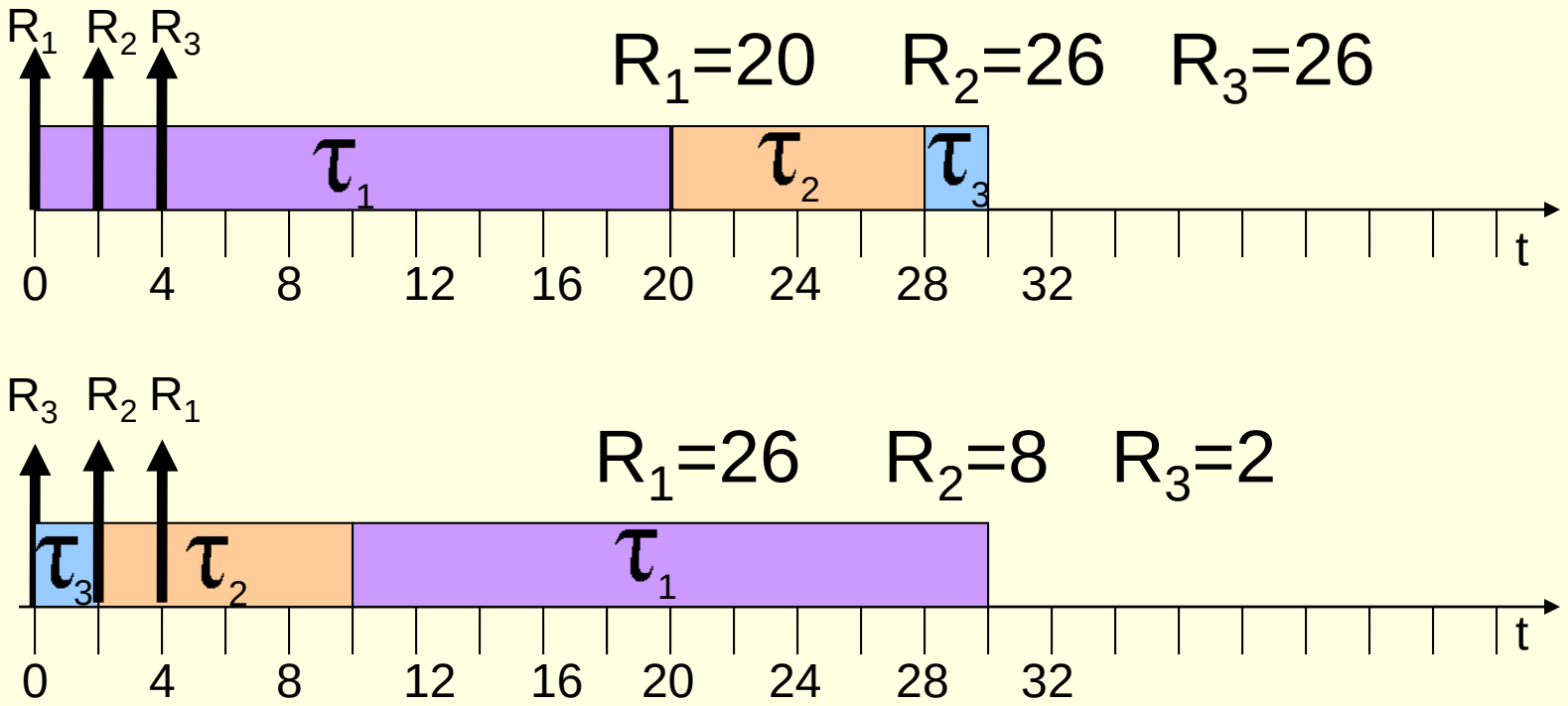


- Non preemptive
- Dynamic
- On line
- Best effort



First Come First Served

- **Very unpredictable**
  - Response times strongly depend on task arrivals.



FCFS – workings

Exercise 2.2: Consider a scenario as follows: We have 3 ready tasks in the system. Task indices and respective worst case execution times are given as a tuple:

(Task index, execution time)

>> (1,24), (2,3), (3,3)

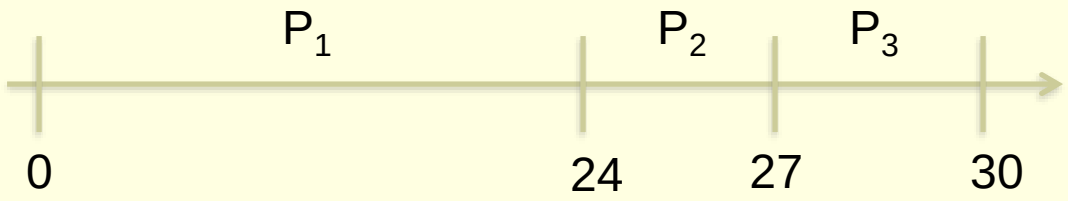
WT: (Start time of a task – Start time of the entire Scheduling process)

Derive a FCFS schedule for the above ordering of tasks and compute average waiting time of a task. Repeat the problem when the task order is (2,3,1).

Verify: AWT (1,2,3) = 17; AWT (2,3,1) = 3

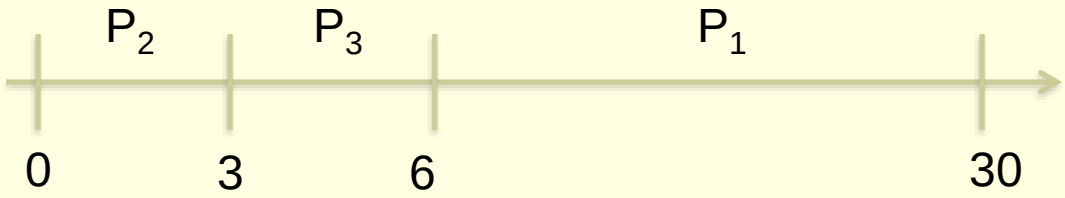
FCFS – Solution

FCFS Seq: 1,2,3



AWT = (0 + 24 + 27) / 3 = 17

FCFS Seq: 2,3,1



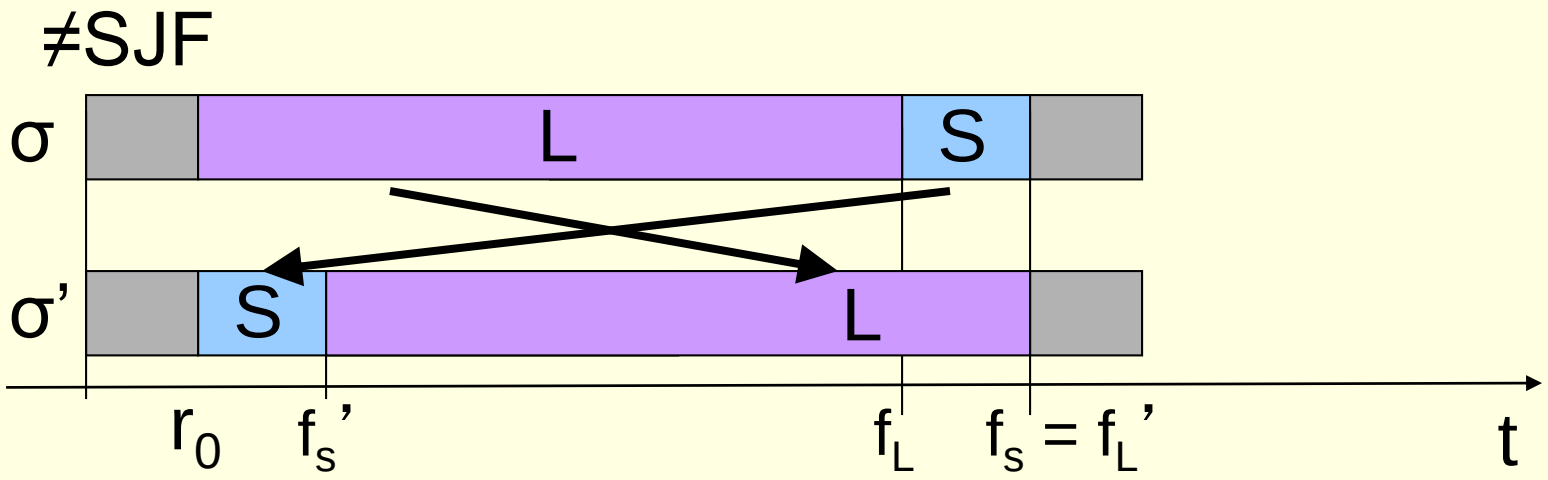
AWT = (0 + 3 + 6) / 3 = 3

Shortest Job First (SJF)

- SJF selects the task with the shortest computation time.
  - Can be either **Non preemptive** or **preemptive**
  - **Static** (constant WCET parameter)
  - It can be used **on-line** or **off-line**
  - It minimizes the average response time

WCET – Worst Case Execution Time

SJF Optimality

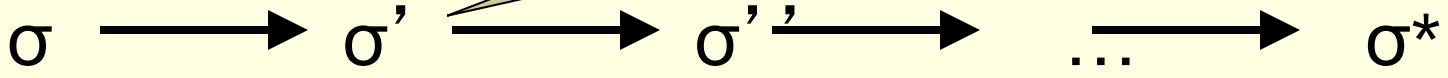


$f_{S'} < f_L$
$f_{L'} = f_S$ implies

$$f_S' + f_L' \leq f_L + f_s$$

SJF Optimality

Replace tasks one-by-one to move shorter tasks earlier



$$\overline{R}(\sigma) \geq \overline{R}(\sigma') \geq \overline{R}(\sigma'')... \geq \overline{R}(\sigma^*)$$

$\sigma^* = \sigma_{SJF}$

$\overline{R}(\sigma_{SJF})$ is the minimum response time achievable by any algorithm

SJF (Cont'd)

Exercise 2.3: Consider a scenario as follows: We have 3 ready tasks in the system. Task indices and respective worst case execution times are given as a tuple:

(Task index, execution time)

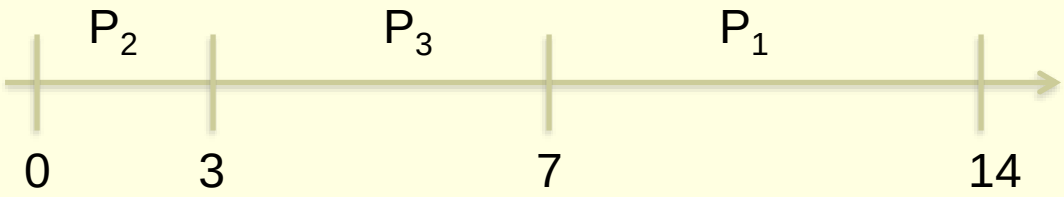
>> (1,7), (2,3), (3,4)

Derive a non-preemptive SJF schedule and compute an average waiting time of a task. Compare with FCFS with an order (1,2,3).

Verify: AWT (SJF: 2,3,1) = 3.33; AWT (FCFS) = 5.66

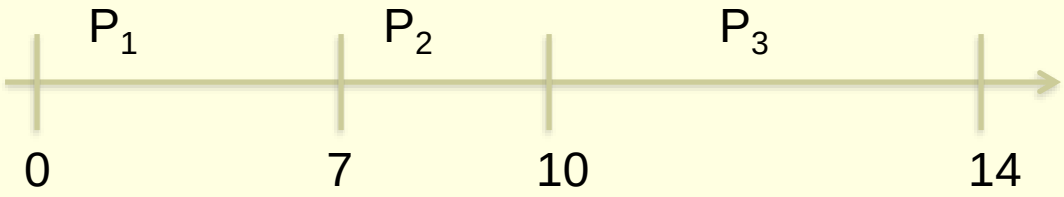
SJF & FCFS – Solution

SJF:



AWT = (0 + 3 + 7) / 3 = 3.33

FCFS Seq: 1,2,3



AWT = (0 + 7 + 10) / 3 = 5.66

SJF – non-preemptive random arrival times

Exercise 2.4: Consider a scenario as follows:
We have 4 ready tasks in the system. All tasks arrive at random times and respective worst case execution times are given as a 3-tuple:

(Task index, arrival time, execution time)

>> (1,0,7), (2,2,4), (3,4,1), (4,5,4).

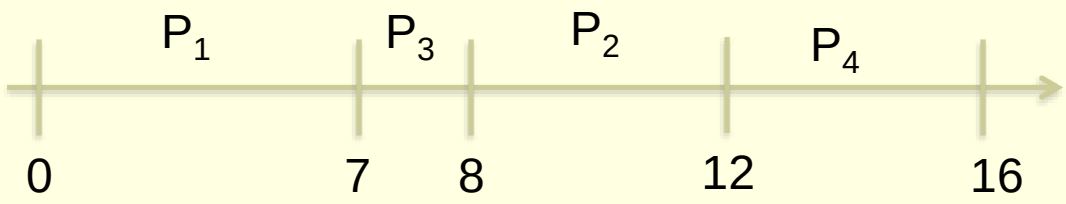
Derive a non-preemptive SJF schedule and compute an average waiting time of a task.

Verify: AWT = 4

Non-Preemptive-SJF – Solution

Non- Preemptive SJF:

Tasks: (1,0,7), (2,2,4), (3,4,1), (4,5,4).



WT of a task = (start time – arrival time)

AWT = (0 + 3 + 6 + 7) / 4 = 4

Shortest Remaining Time First (SRTF) – Preemptive strategy

Suppose we attempt to use a strategy in which we give importance to a task that has shortest remaining time to run first, then how will be the performance?

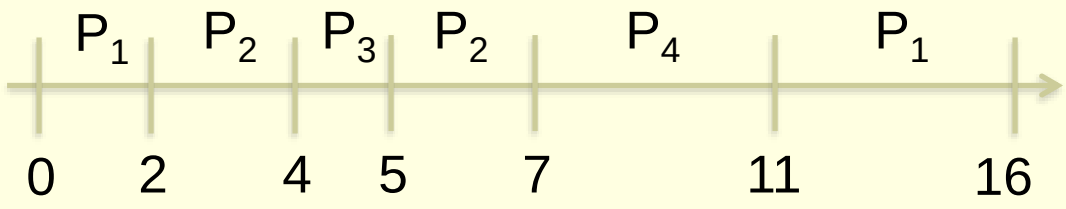
Waiting time = (Start time – arrival time) + wait time for next execution

Exercise 2.5: For the same task set in the previous slide, derive a preemptive SRTF schedule and compute an average waiting time of a task and compare with non-preemptive SJF.

Verify: AWT = 3

SRTF – Preemptive strategy - Solution

Tasks: (1,0,7), (2,2,4), (3,4,1), (4,5,4).



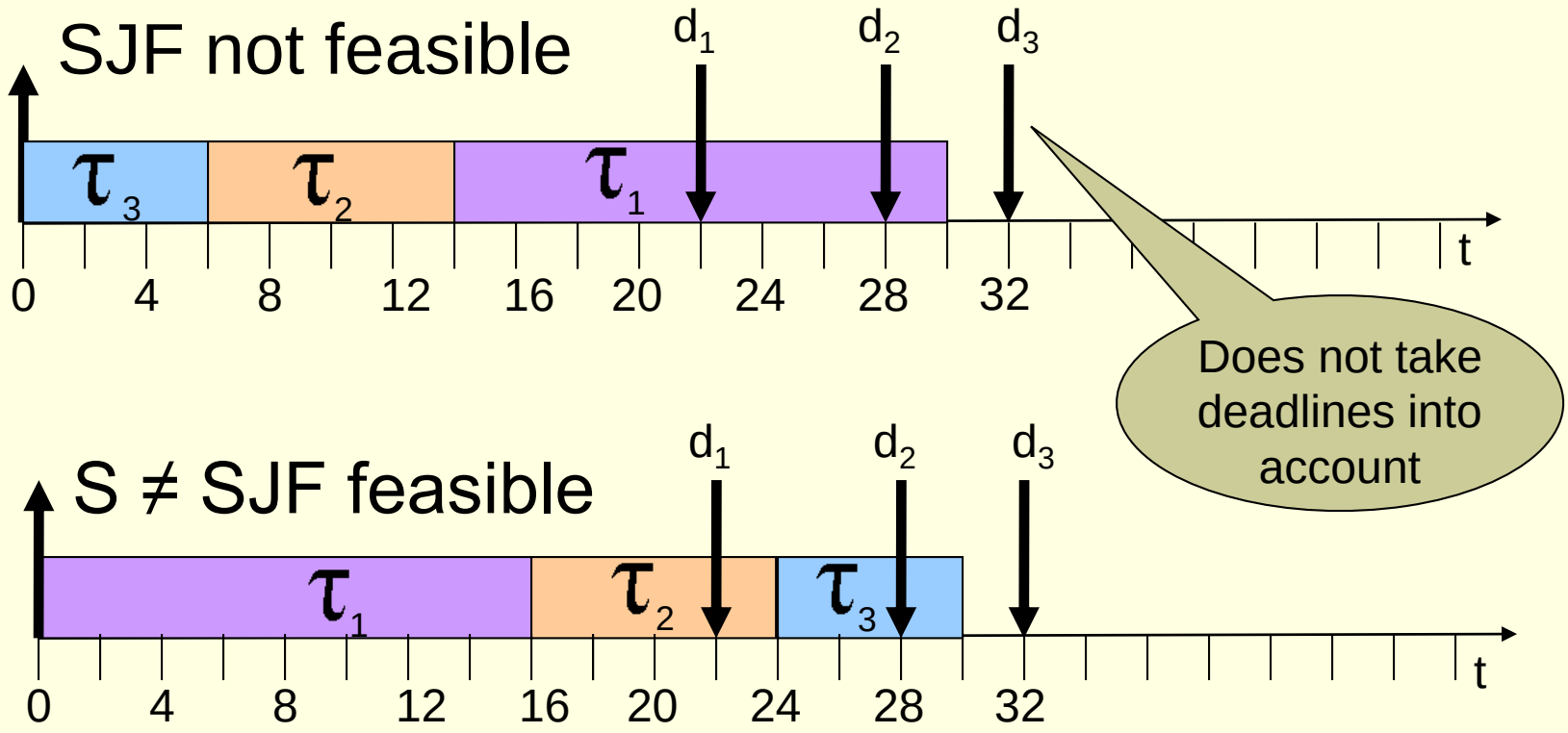
Waiting time = (Start time – arrival time) + wait time for next execution

P_1 P_2 P_3 P_4

AWT = (9 + 1 + 0 + 2) / 4 = 3

SJF - *suited for Real-Time*?

- It is not optimal in the sense of feasibility

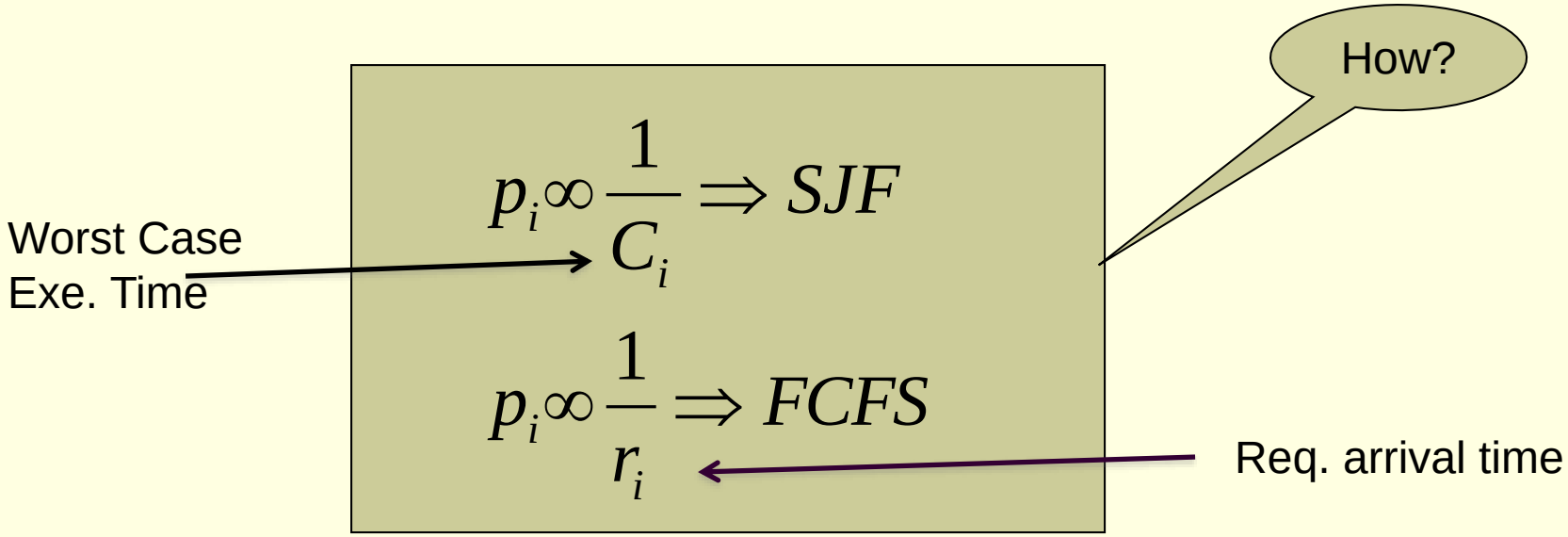


Priority Scheduling

- Each task is assigned a priority: $p_i \in [0, 255]$
- The task with the highest priority is selected for execution.
- Tasks with the same priority are served FCFS.
  (This is by default and can be changed)
- Preemptive
- Static or dynamic
- Online

Priority Scheduling

- **Problem: starvation**
  - Low priority tasks may experience long delays due to the preemption of high priority tasks.
- **Solution: Aging**
  - priority p_i of a task increases with waiting time



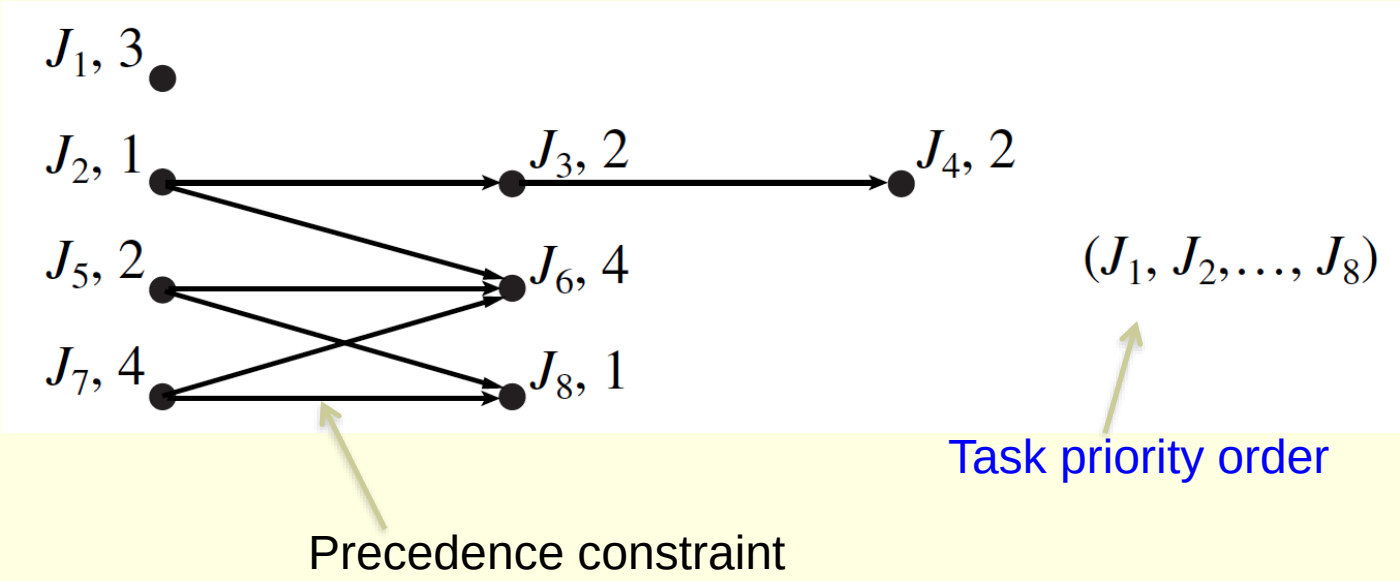
Priority-driven approach – *Some take-away points!*

- Never leave any resource to stay idle intentionally - Key aspect of Priority-driven schedulers.
- This does not mean optimal schedules may be guaranteed.
- Scheduling decisions are made when events such as releases and completions of jobs occur. Hence, priority-driven algorithms are mostly event-driven.
- A priority-driven algorithm is ***greedy*** because it tries to make locally optimal decisions. *Leaving a resource idle while some job is ready to use the resource is not locally optimal.* So when a processor or resource is available and some job can use it to make progress, such an algorithm never makes a job to wait.

Priority-driven approaches

- Preemptive and Non-preemptive criteria using 2 processors

Given **DAG** (task index, execution time)



Task 5 is available at t = 4;

All other tasks are available from t =0 onwards

Problem Exercise 2.6: Derive a schedule for the above two cases.

Remarks on Priority-driven approaches

- A few fundamental questions are:

Q: When is preemptive scheduling better than non-preemptive scheduling and vice versa?

Q: Is there any rule with which we could determine from the given parameters of the jobs whether to schedule them preemptively or non-preemptively?

Remarks on priority-driven approaches

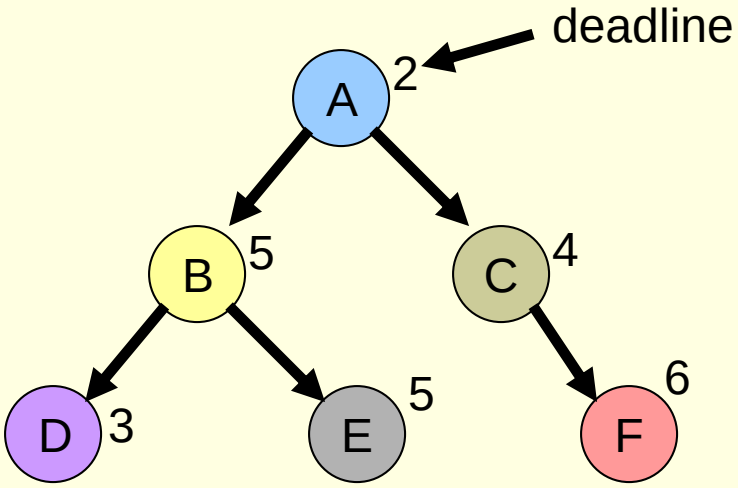
- In a multiprocessor system, the minimum makespan (i.e., the response time of the job that completes last among all jobs) achievable by an optimal preemptive algorithm is shorter than the makespan achievable by an optimal non-preemptive algorithm.

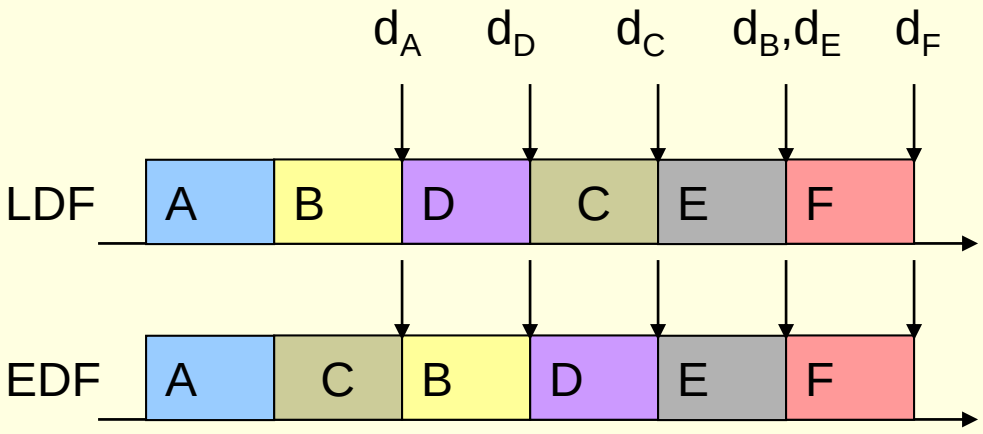
- The above claim has been shown to be true only for a 2 processor case in the literature.

Lemma: *The minimum makespan achievable by non-preemptive algorithms is never more than 4/3 times the minimum makespan achievable by preemptive algorithms, when the cost of preemption is negligible*.

Using precedence constraints with deadlines

- Latest Deadline First (LDF) [Lawler 73]
  - Given a precedence graph, it constructs the schedule from the tail; We have deadlines for the sub-tasks too
  - Among the nodes with no successors, LDF selects the task with the latest deadline:

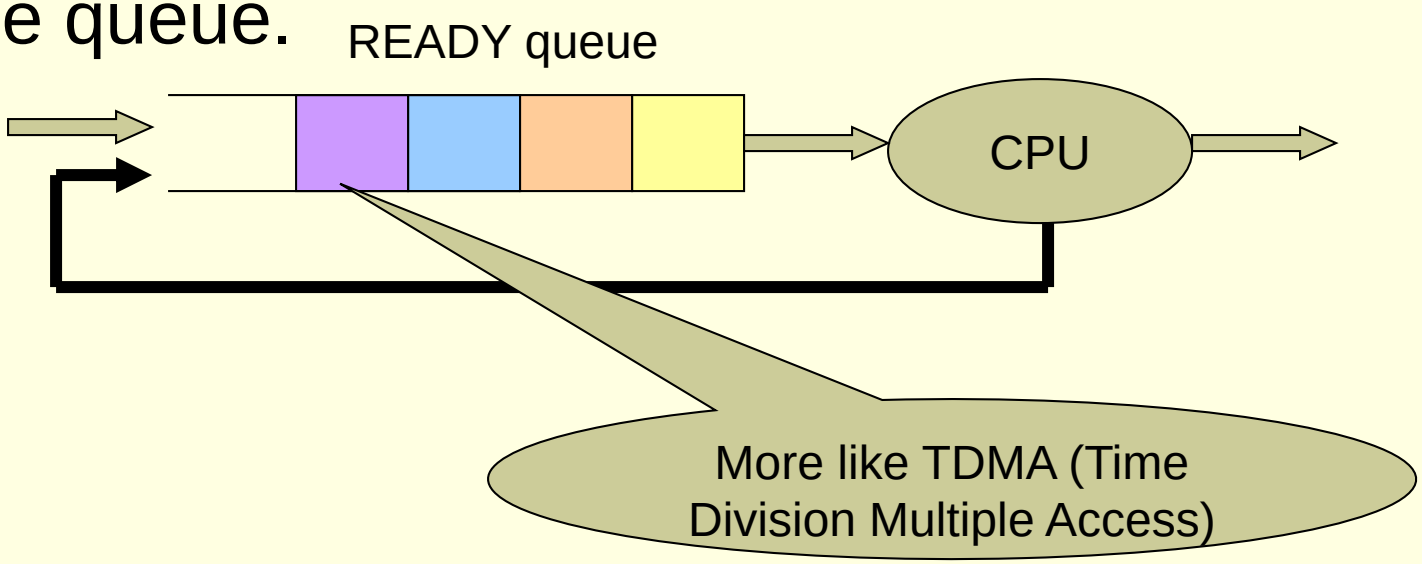




D misses its deadline

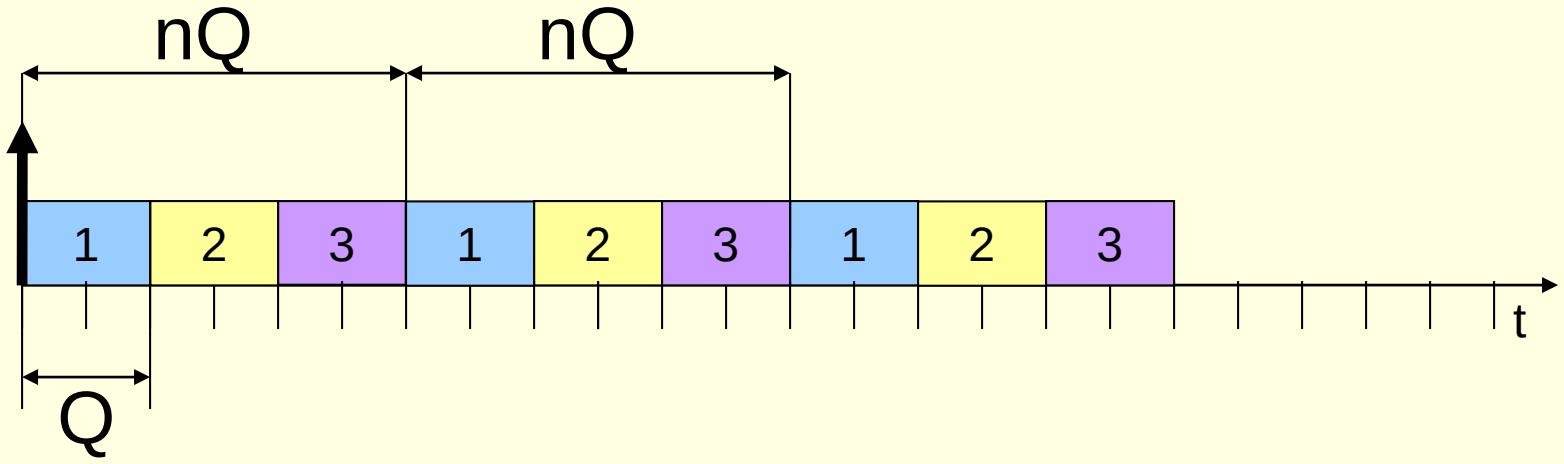
Round Robin (a.k.a *processor-sharing algorithm*)

- The ready queue is served as FCFS
- Each task τ_i cannot execute more than Q time units (Q = fixed time quantum).
- When Q expires, τ_i is put back at the end of the queue.



Round Robin

- **n** = number of tasks in the system



Time sharing

Q - Run time quantum

Each task gets (1/n)-th share of the processor when there are n tasks in the ready queue for execution

Round robin (Cont'd)

Exercise 2.7 *(DIY!)*

N = 3 processes with their resp. execution times ($P_1 = 24$, $P_2 = 3$, $P_3 = 3$)

Run time quantum Q = 4 msecs

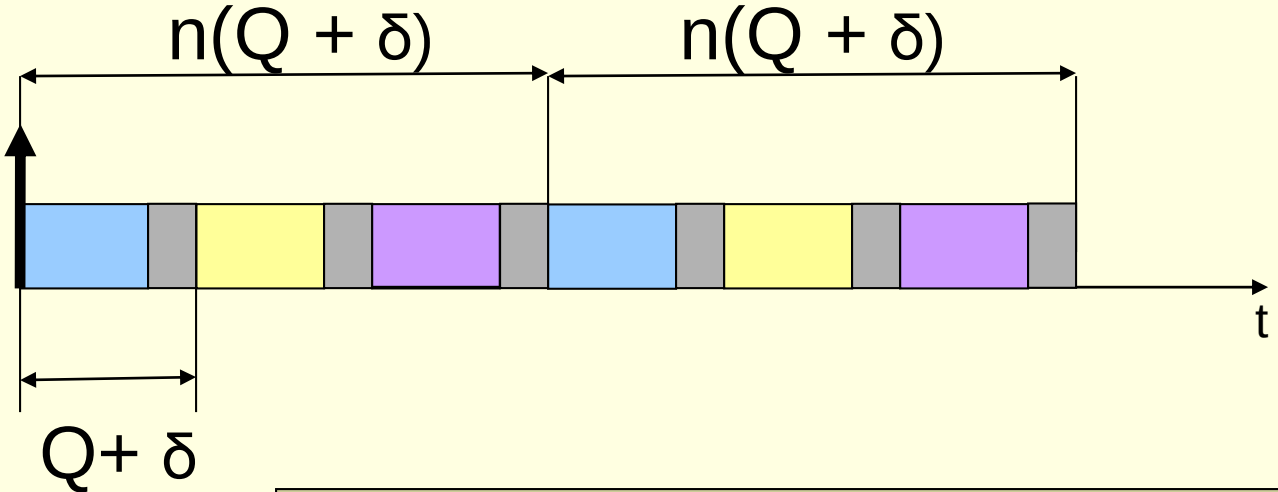
Draw a timing diagram (Gantt Chart) and determine the AWT.

Soln: Seq you will see: (1,2,3,1,1,1,1,1)

Verify AWT: 5.6

Round Robin

- **if** Q + context switch time (δ) **then**



$$R_i \cong n(Q+\delta)\frac{C_i}{Q} = nC_i(\frac{Q+\delta}{Q})$$

Delay Bounded Aperiodic Scheduling – *Bag of tasks*

- Set/Bag of tasks – Time-to-Complete execution is known (given parameter), Deadline for each task is given;
- You are free to choose any task sequence, which means: (1,2,3,4), (2,1,3,4),….(3,4,1,2)... are all possible solutions! (*What is the size of the solution space?*)
- **Lateness** is a metric to focus on for some applications **–** Non-real-time data distribution; Often a frequently confronted problem on Content Distribution Networks (CDNs)

- **Objective**: Choose that sequence which generates minimum cumulative lateness

Download the material from our course website!

Large-scale workload scheduling on networked compute systems

- Workload model – Arbitrarily divisible
- Platform – Loosely-coupled system with non-zero communication and compute delays
- Objective – Minimize the total processing time
- Communication & computation time model - Linear/Affine model
- Network topologies – Bus, star, mesh(2d,3d), multi-level tree, arbitrary graphs.

Classroom discussion: Derive an optimal schedule for a bus network handling a large-scale arbitrarily divisible workload model.

Metrics for Performance Evaluation

- What are the available metrics?
- How to choose a metric **or** a set of metrics for a given problem? – *A major challenge!*

- ***Exercise 2.8:*** *Refer to Slides #7 , #8 – Write the expression for the following metrics:*

Average Response Time;

Total Response Time;

Weighted sum of completion times;

Maximum lateness;

Maximum number of late tasks

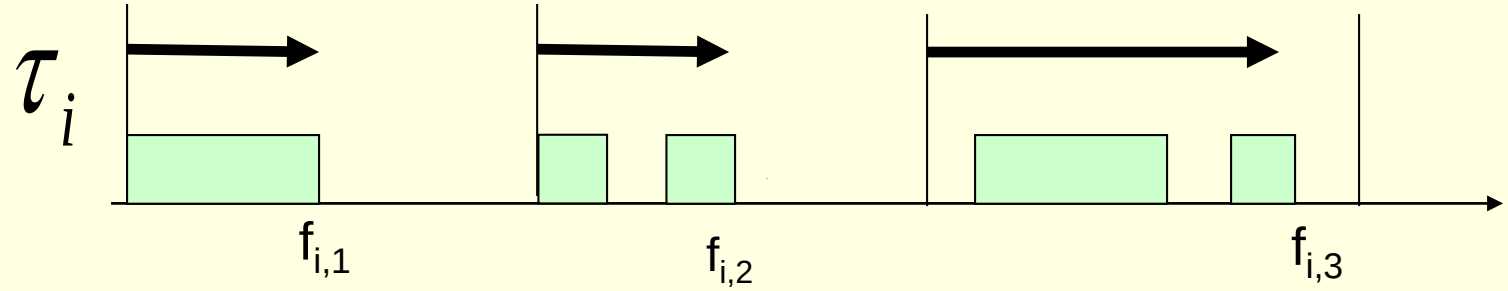
THANK YOU!

Annexure

Jitter

- It is the time variation of a periodic event

Finishing-time Jitter

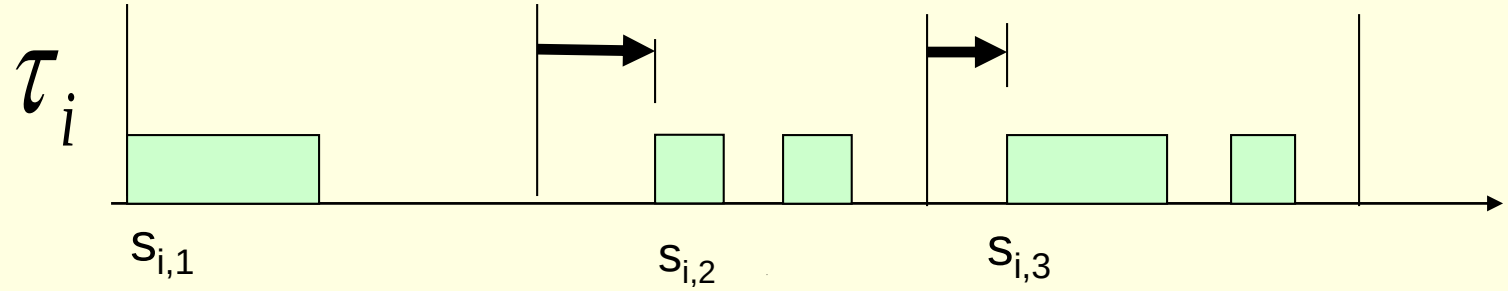


Absolute $\max_k(f_{i,k} - r_{i,k}) - \min_k(f_{i,k} - r_{i,k})$

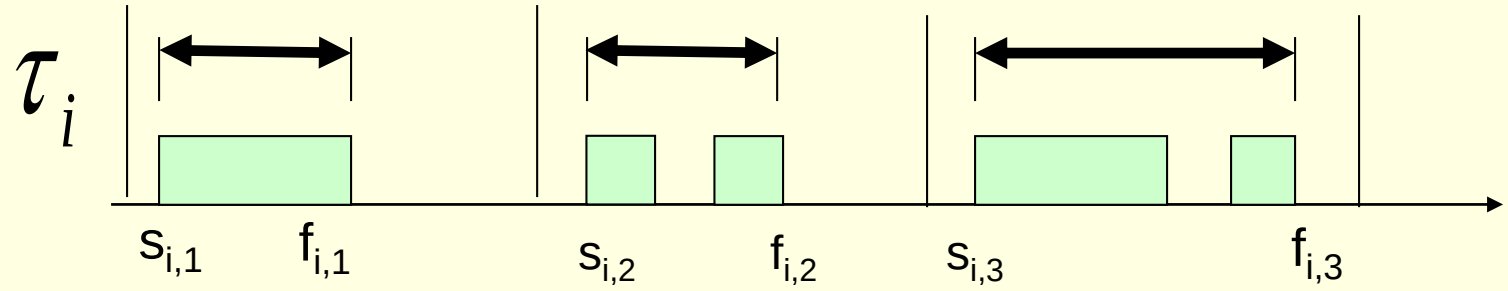
Relative $\max_k |(f_{i,k} - r_{i,k}) - (f_{i,k-1} - r_{i,k-1})|$

Other types of Jitter

Start-time Jitter



Completion-time Jitter (I/O jitter)



Weighted Round Robin (suitable for Real-time)

Follows Round Robin algorithm discussed earlier;

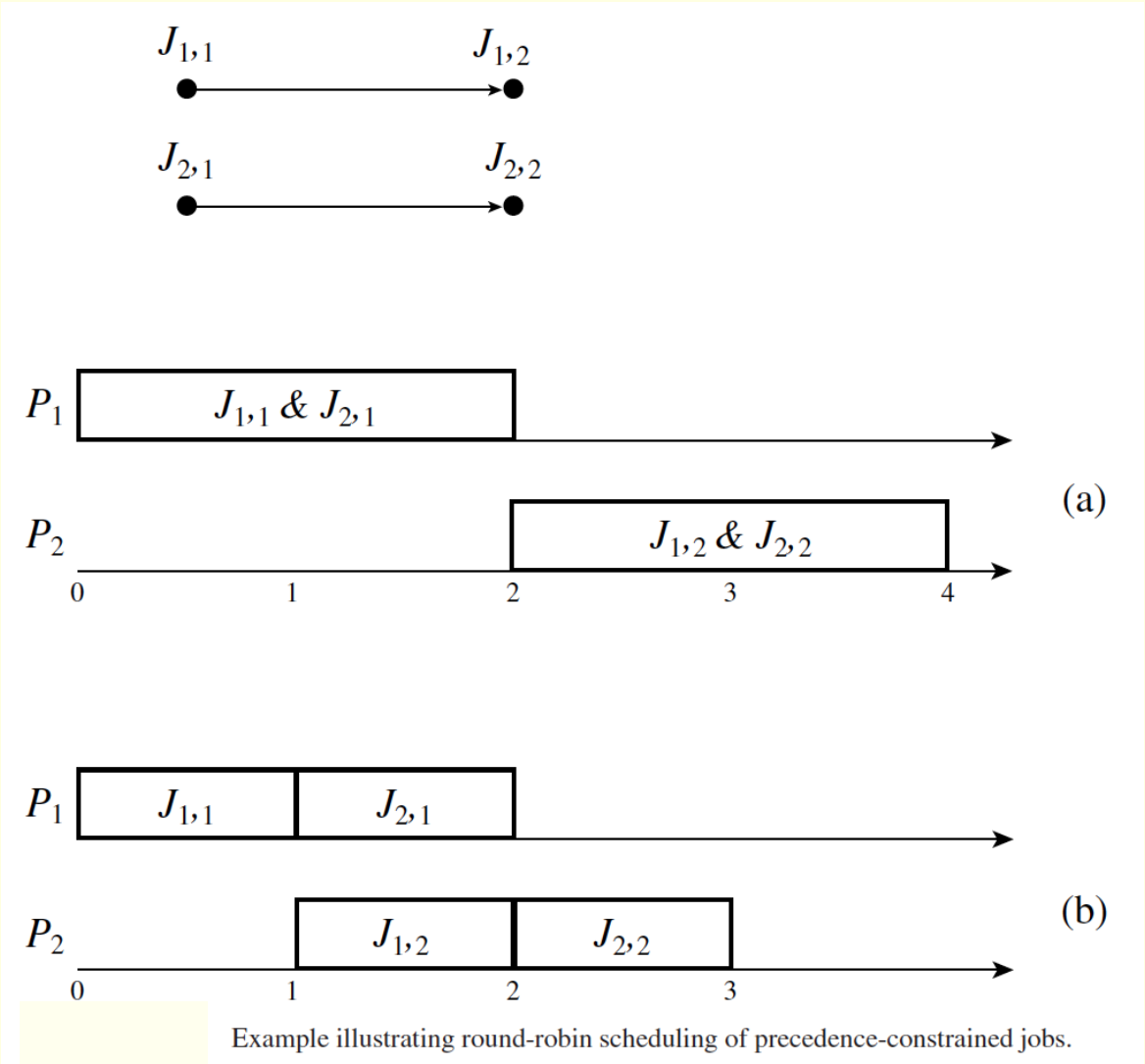
- **Key Idea**: Rather than giving all the ready jobs equal shares of the processor, different jobs may be given different weights.
- **Weight** of a job refers **to the fraction of processor time allocated to the job**.
- In this algorithm, this implies: A job with weight k gets k time slices every round, and the length of a round is equal to the sum of the weights of all the ready jobs.
- Thus, by adjusting the weights of jobs, we can speed up or retard the progress of each job toward its completion.

Weighted RR (Cont'd)

- Weighted RR can delay the completion of the jobs and thus unsuitable for precedence constrained jobs (DAGs);
- However, it has an advantage – a predecessor job can take advantage of partially completed processed results from its successor and start the processing in a pipelined fashion. (See an example – next slide)

- Network switches can avoid using priority queues (expensive to realize) can take advantage of weighted RR– *How?*

Example of a Weighted RR for precedence-constrained tasks



Example illustrating round-robin scheduling of precedence-constrained jobs.