EGH456 Embedded Systems

Lecture 9

Scheduling algorithms

Dr Chris Lehnert

Previous Lecture

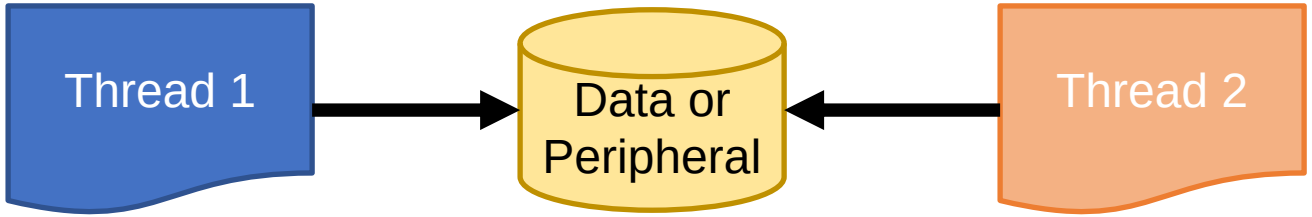
- Resource Sharing
  - Message Queues, Mailboxes
- Synchronisation
  - Events
- Clock Module

Resource Sharing

Producer – Consumer Model

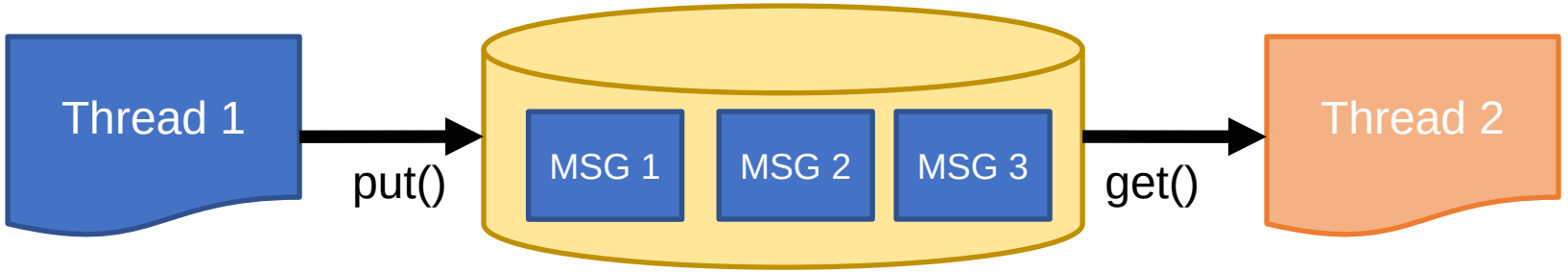
- Thread 1 produces a buffer or Msg and places into BIOS “container”
- Thread 2 blocks unit available, then consumes it when signalled (no contentions

Concurrent Access Model



- Any thread could access “Resource at any time (no structured protocol)
- Pre-emption of one thread by another can cause contention

Message Queues



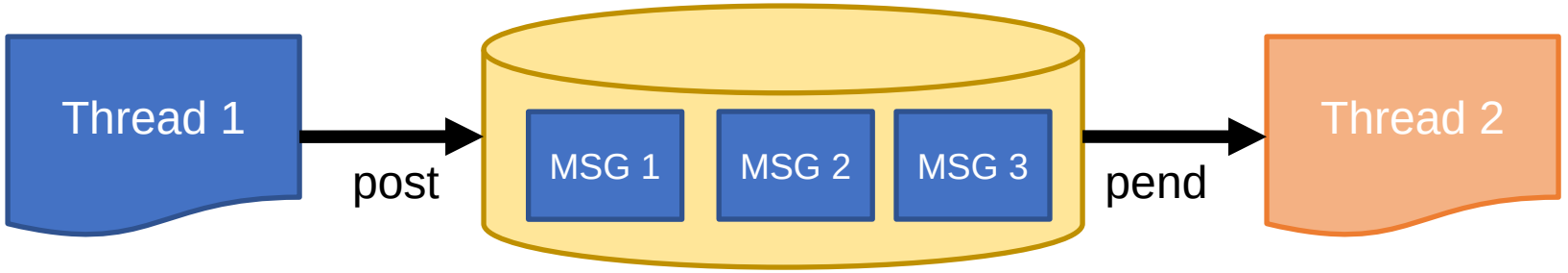
Queue

- A **Queue** is a bios object that can contain *anything*
- Data in a queue is called a Msg – simply a userdefined struct
- API:

```
Queue_put();
Queue_get();
```

- Simple but no signalling built in

Mailbox



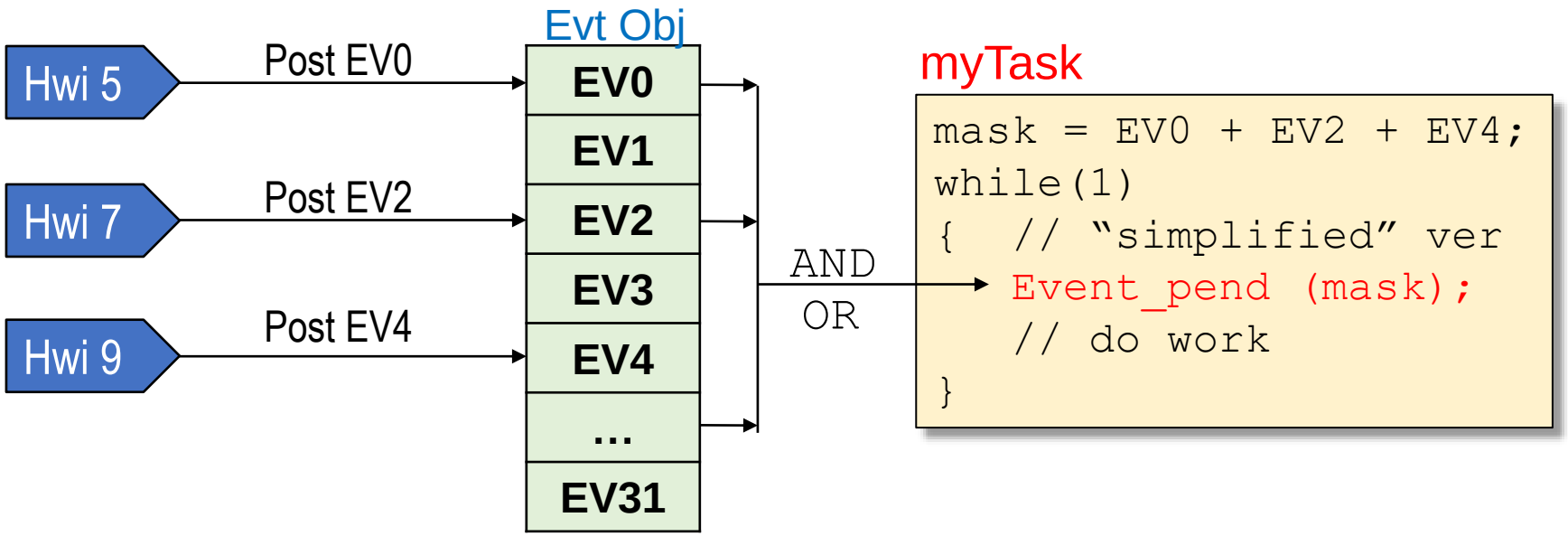
Mailbox

- Fixed size BIOS object that can contain anything you like
- Fixed length – Number of Msgs (length) and Message size
- API

```
Mailbox_post (&Mbx,&Msg,timeout); /*blocks if full*/
Mailbox_pend (&Mbx,&Msg,timeout); /*blocks if empty*/
```

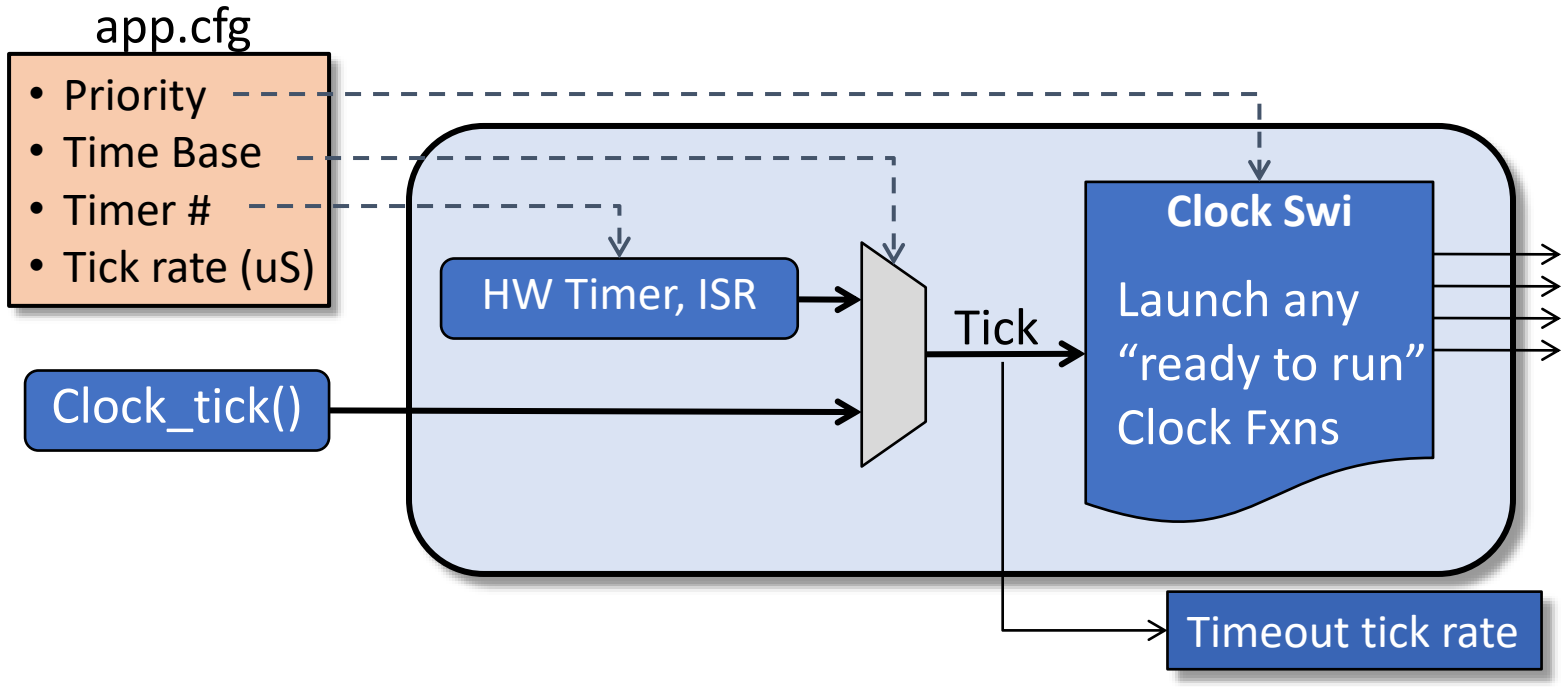
- Contains built in semaphore but makes copy of data
- Good practice to pass pointers through mailbox instead of data

Events



- Sometimes multiple conditions need to be satisfied before action
- A semaphore by itself is not enough. The semaphore object is modified to have an event field and support events.
- Semaphore_pend() only waits on one flag – a semaphore.
- Use Events. Can **OR** or **AND** event **ID**s with bit masks

SYS/BIOS Clock Module



- User can choose time base – timer or app calls Clock_tick()
- Explicit call of Clock_tick() could occur from your own ISR (GPIO, etc.)

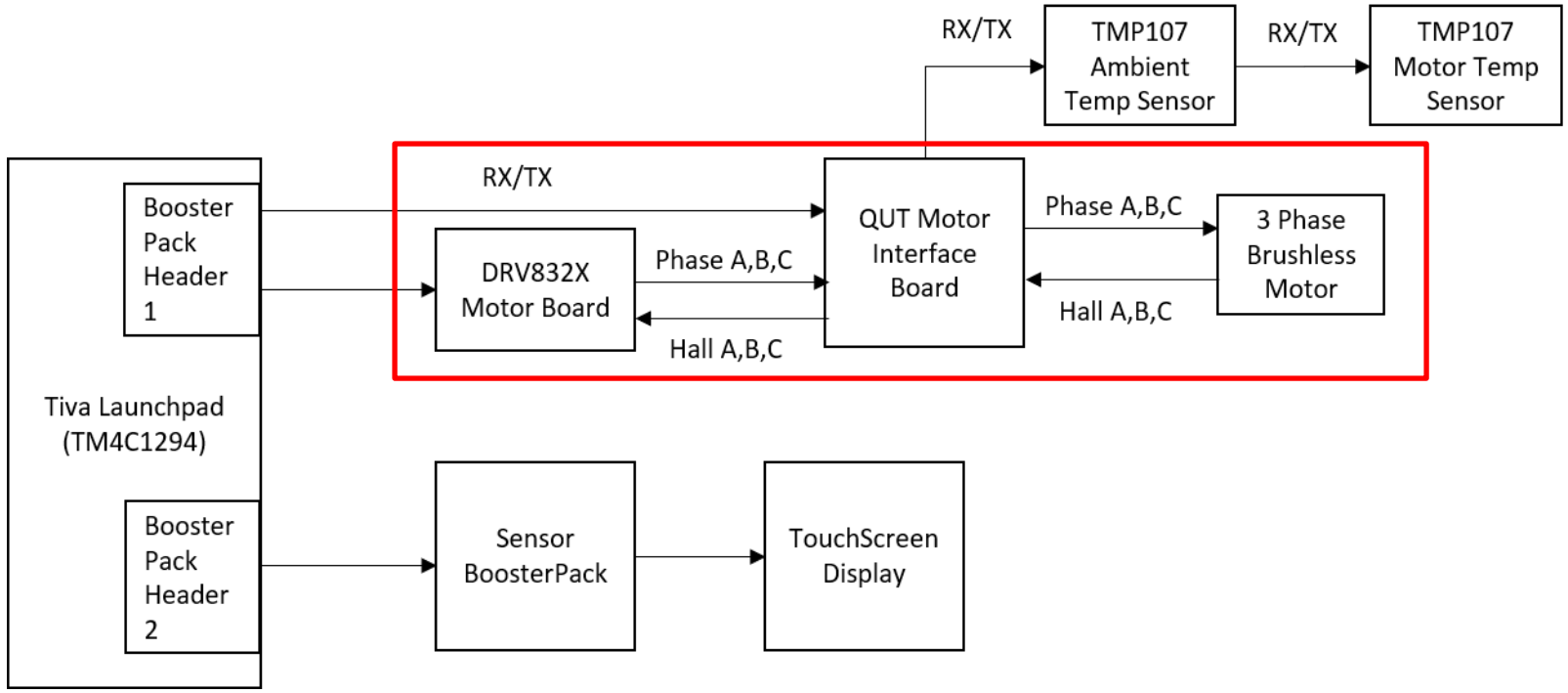
Adapted from TI

Contents

- Assignment Discussion
- Scheduling algorithms
  - Rate monotonic scheduling
  - Earliest Deadline First
- Scheduling Examples

Assignment - Motor Lib Overview

- Motor Lib provides low level functions via a software driver model to initialise and update the brushless motor



Motor Driver

1. Init Hall GPIO lines
2. Init Motor Lib
3. Start Motor
   - Force single `updateMotor ()`
4. Control Loop
   - Update pwm from PI code
   - `setDuty()`

GPIO Hwi Interrupt for Hall Lines

1. Read Hall Sensors (GPIO)
2. Increment speed counter
3. `updateMotor()`

Swi Interrupt for Hall Lines (Speed)

1. Read Hall Sensors (GPIO)
2. `updateMotor()`

Scheduling - Task switch

- Recall: Real-time software divided into threads:
  - interrupts and tasks etc…
- How does it switch from one to another?
  - Hardware interrupt – acknowledged at hardware level
  - What about others?
    - Time, Software Event?

Scheduler

- The scheduler is the software that determines which task should be run next
  - What is involved in this switch?
  - How is the decision made?
  - Why is this important to real-time systems?
- This is the core functionality of a kernel
- Upon return from an interrupt, thread or via programmed kernel calls, the kernel makes a decision to continue with the current thread or to switch

Types of Scheduling

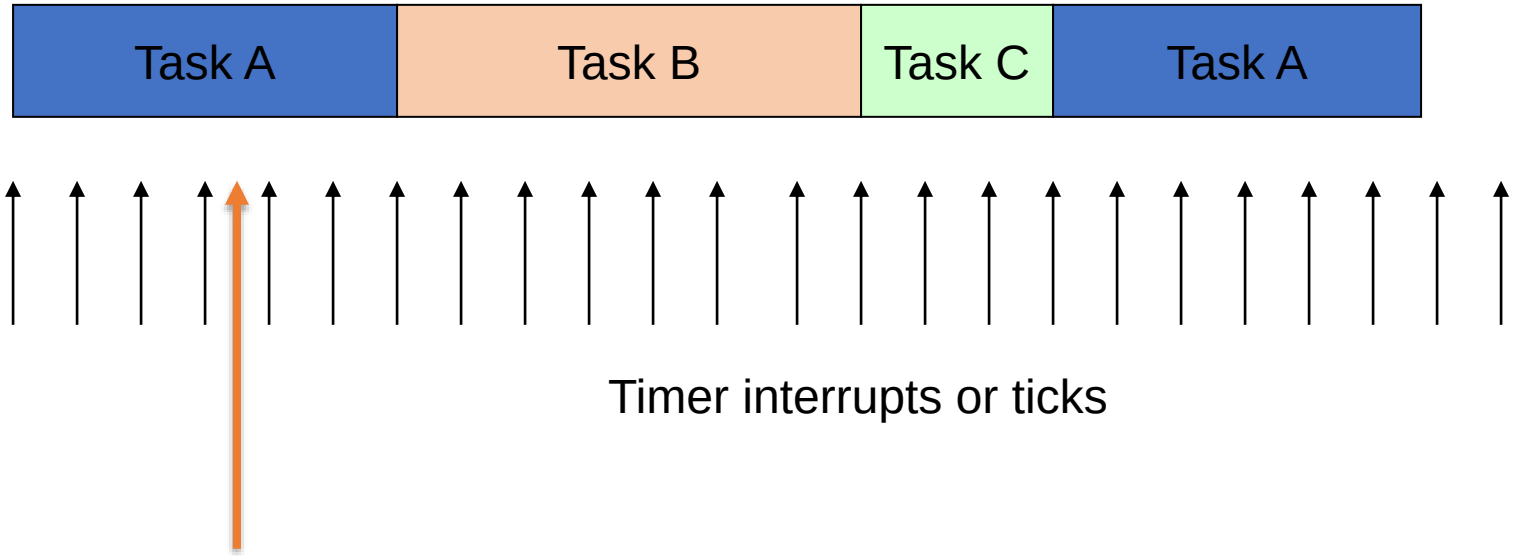
- **Non-pre-emptive**
  - Threads are given control of the master CPU until they have finished execution, regardless of the length of time or the importance of the other tasks that are waiting
  - Up to software developer to decide when switches occur (using yields etc…)
- **Pre-emptive (Priority based)**
  - Context switching is triggered when events are serviced.
  - This could be any interrupt or syncronisation event
- **Time Slicing / Round Robin**
  - Every thread may also be given a maximum time slice.
  - Each process in the queue is allocated an equal time slice (the duration each process has to run), where an interrupt is generated at the end of each of these intervals to start the pre-emption process

Can have a mixture of different methods

Pre-emptive – after allocated ticks

Response time = can be very long.

The remaining slice for A if Task B were the next ready task.

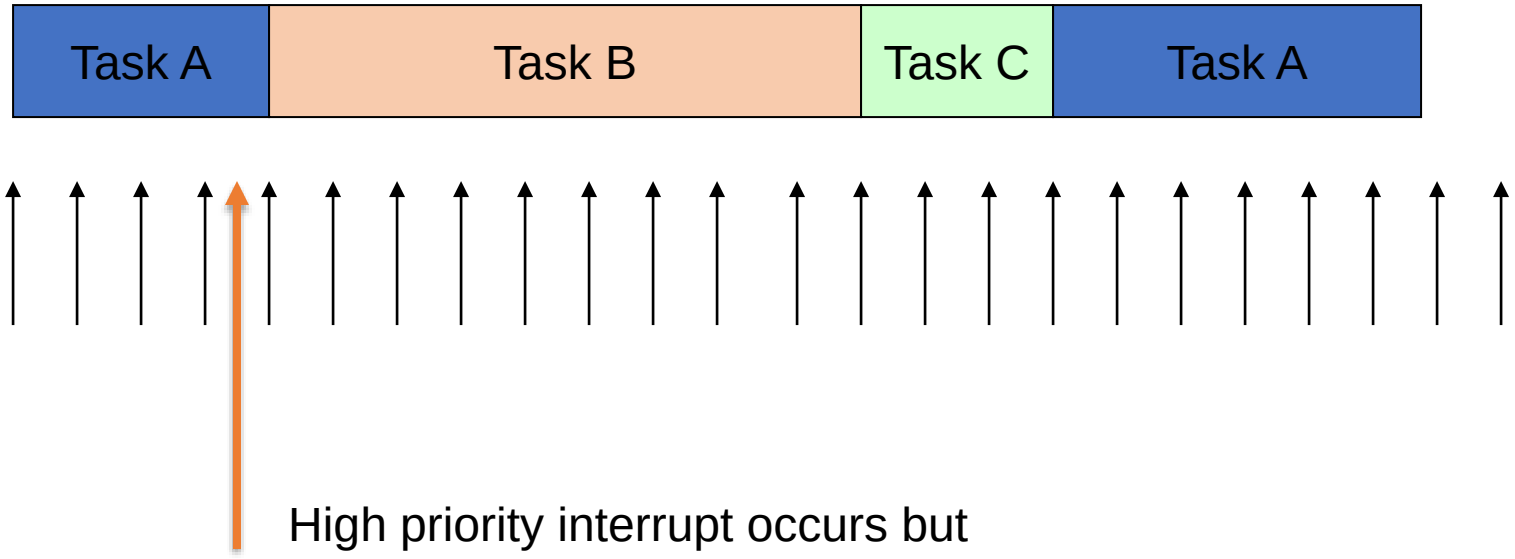


High priority interrupt should be serviced by Task B but has to wait

Task A runs for a specified number of ticks that is allocated

Pre-emptive at the next time tick

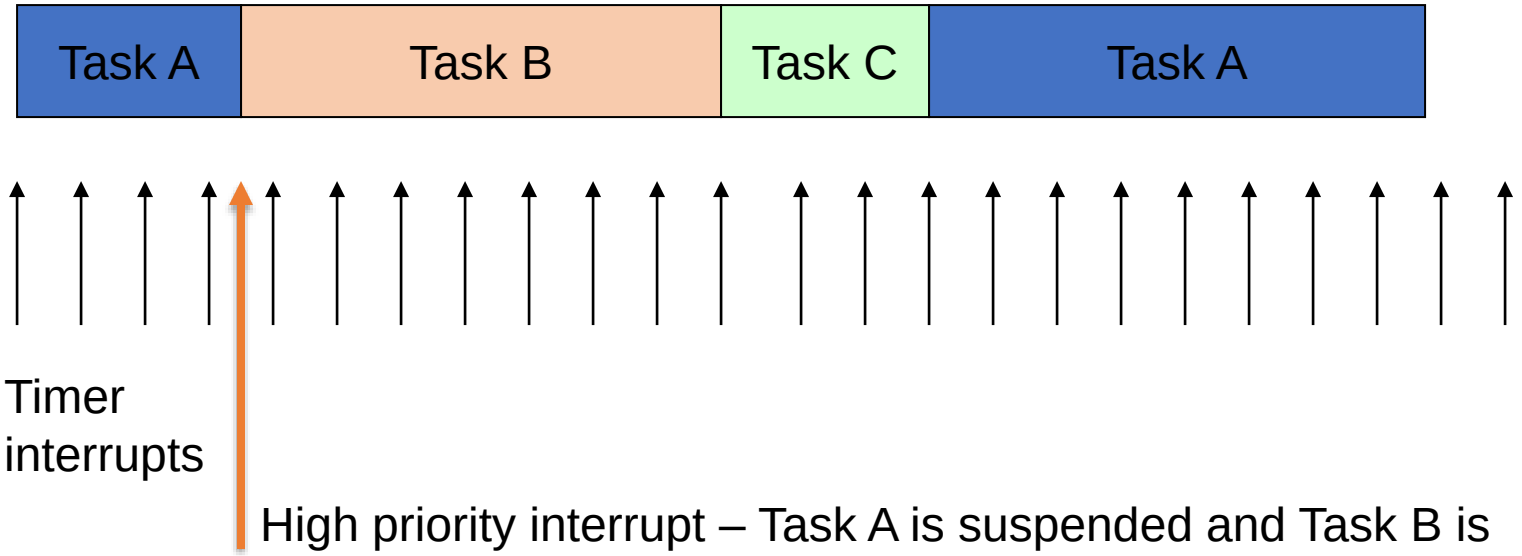
Response time = 1 tick in the worst case if tasks are prioritized



Immediate pre-emptive

Response time = interrupt latency + scheduling latency

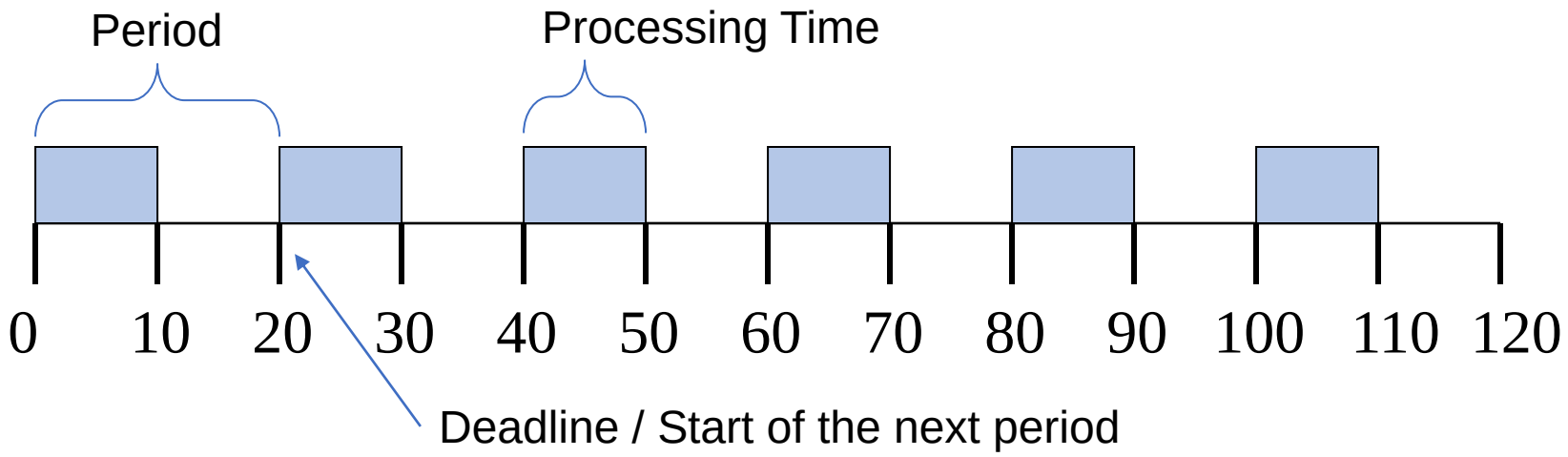
Assuming task B is the highest priority task



High priority interrupt – Task A is suspended and Task B is made to run to process some critical event. There is a small delay arising from interrupt processing and context switch

Real-time Scheduling of Periodic tasks

- **Task** has two properties **Period** and **Processing Time**
- **Deadline** start of the next **Period**
- Simplistic treatment with no blocking for resources
- Focus on Task threads and not hardware interrupts
- Assume scheduling decision every 10 msec



Completion deadlines

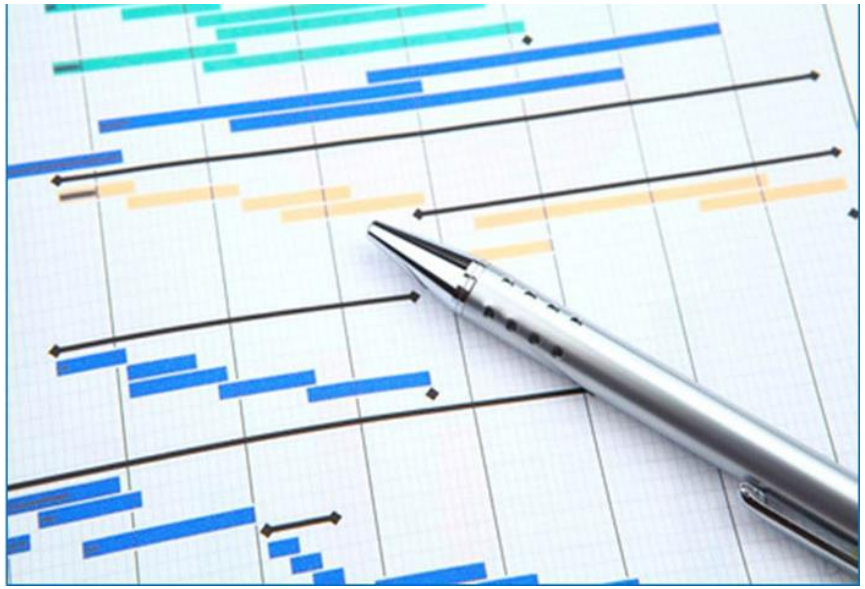
- Short processing is done using Hwi or Hwi and Swi efficiently.

  (Hwis order of microseconds, Swis order of ms typically with the lab platform)

- Tasks do bulk of the work
- Tasks must be scheduled to run appropriately to meet completion deadlines

Scheduling Strategies

- **Rate Monotonic Scheduling**
  - Highest priority for highest frequency task
  - Static priority
- **Earliest Deadline First**
  - Highest priority to closest deadline
  - Dynamic priority
- **Stu Monotonic**
  - All threads at equal priority, raise threads that don't meet deadlines
- Minimum Laxity, Deadline monotonic and many more…



Real-time Scheduling of Periodic Processes

Example 1.

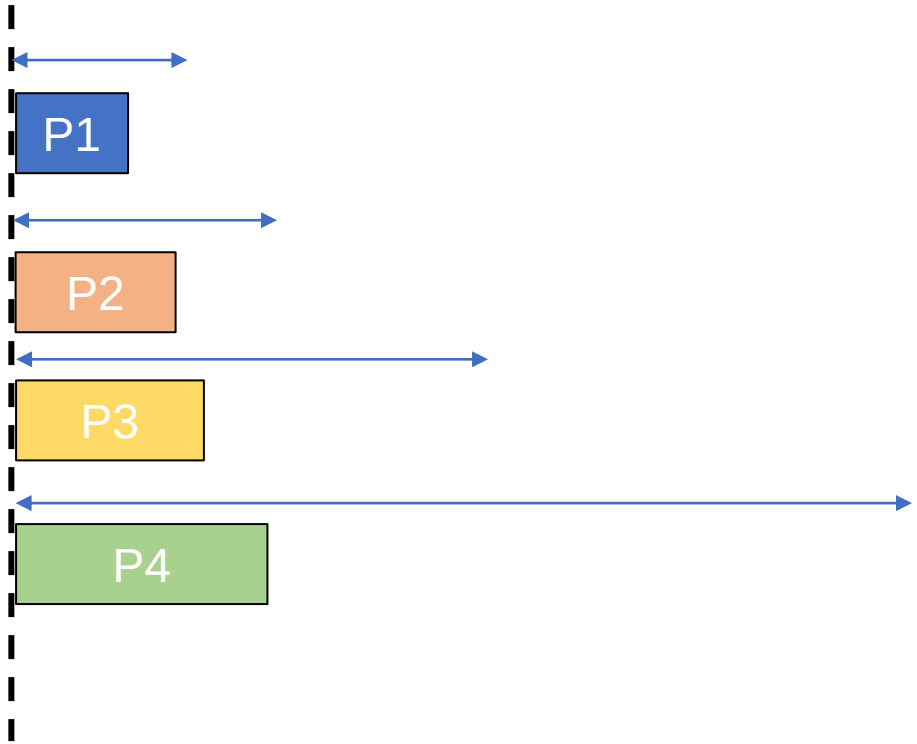
Task A: Period 20 ms, Processing time 10 ms

Task B: Period 50 ms, Processing time 25 ms

Total CPU utilisation 100%

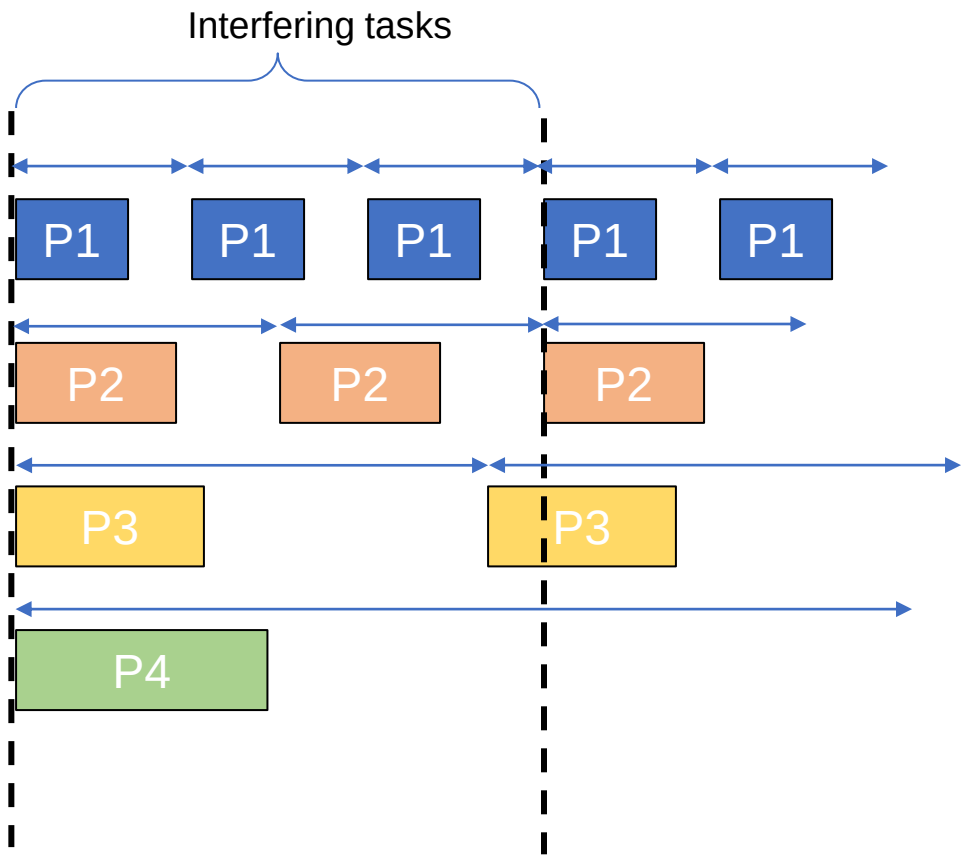
Critical Instant

- Scheduling state that gives worst response time



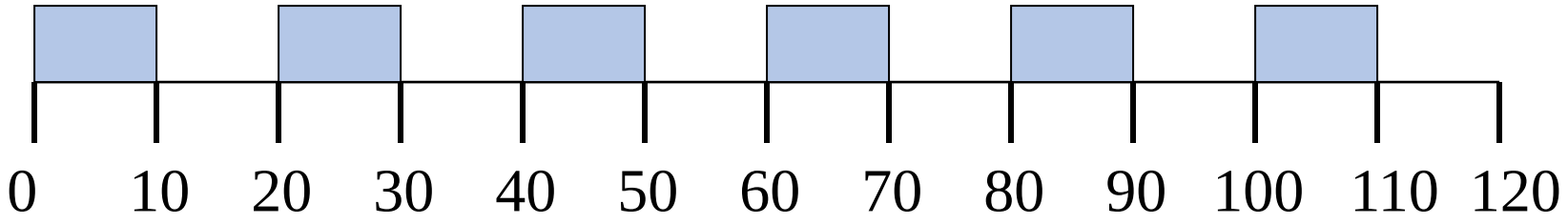
Critical Instant

- Scheduling state that gives worst response time

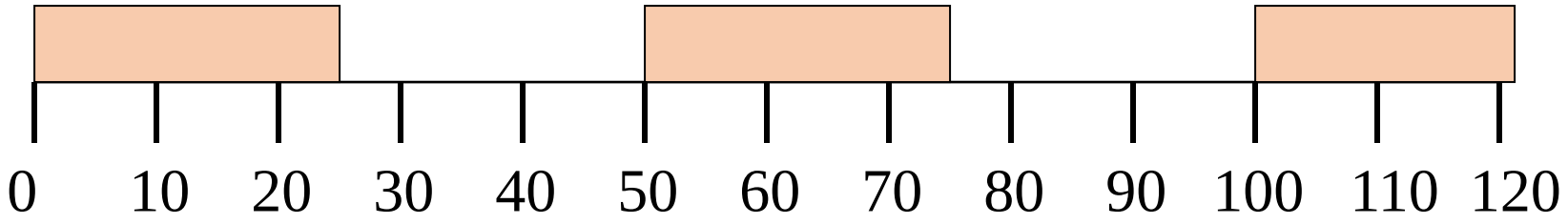


Task set for example 1

Periodic task A

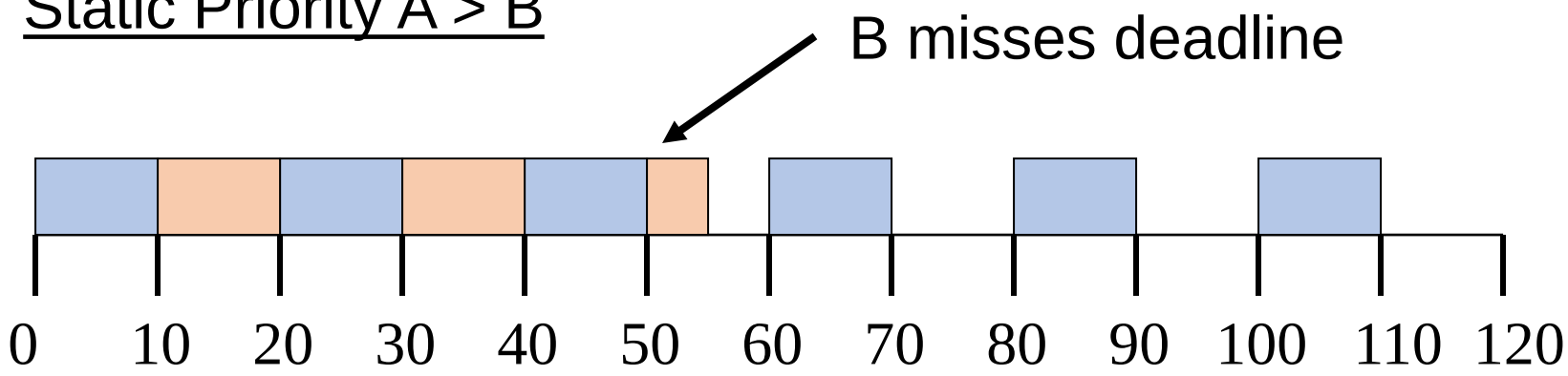


Periodic task B

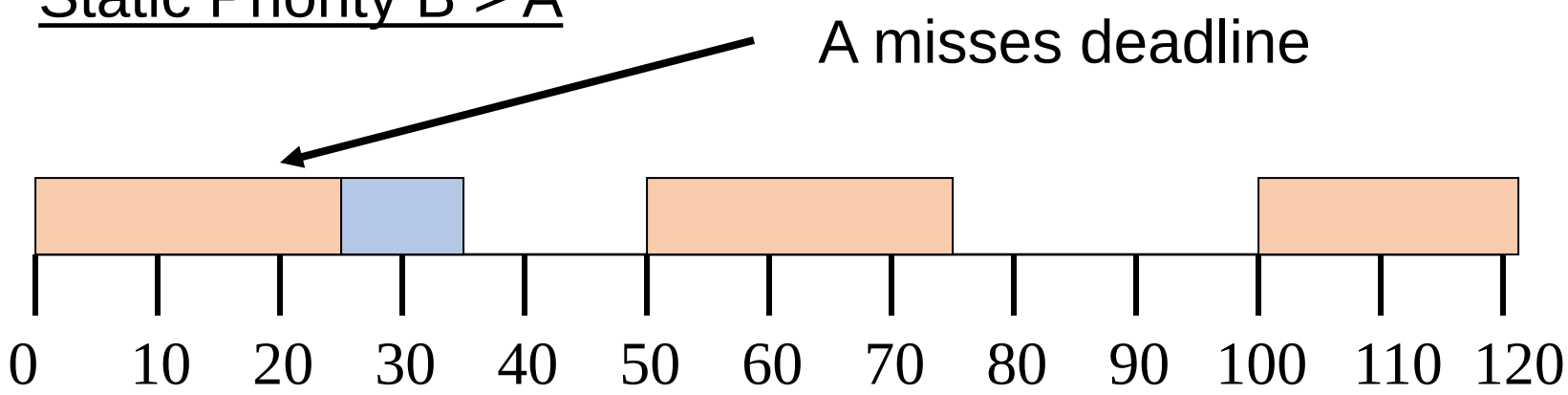


Static Priority

Static Priority A > B



Static Priority B > A



Schedulable bound

- Given a task set, there is a schedulable bound on the CPU utilization – the maximum CPU utilisation such that the tasks can still be scheduled to meet deadlines.
- What will it be in general for periodic tasks?
- What is the best for static priorities?
- How do we assign the priorities?

Rate Monotonic Scheduling

- Assumptions
  - All processes run on single CPU
  - Zero context switch time
  - Process execution time is constant
  - Deadline is at end of period

Rate Monotonic Scheduling

- Priority proportional to the frequency of the task
- Static Priorities

Tasks $i = 1,2,...,N$

Computing times C_i

Periods T_i

Total CPU utilisation is $U_N = \sum_{i=1}^{N} \frac{C_i}{T_i}$

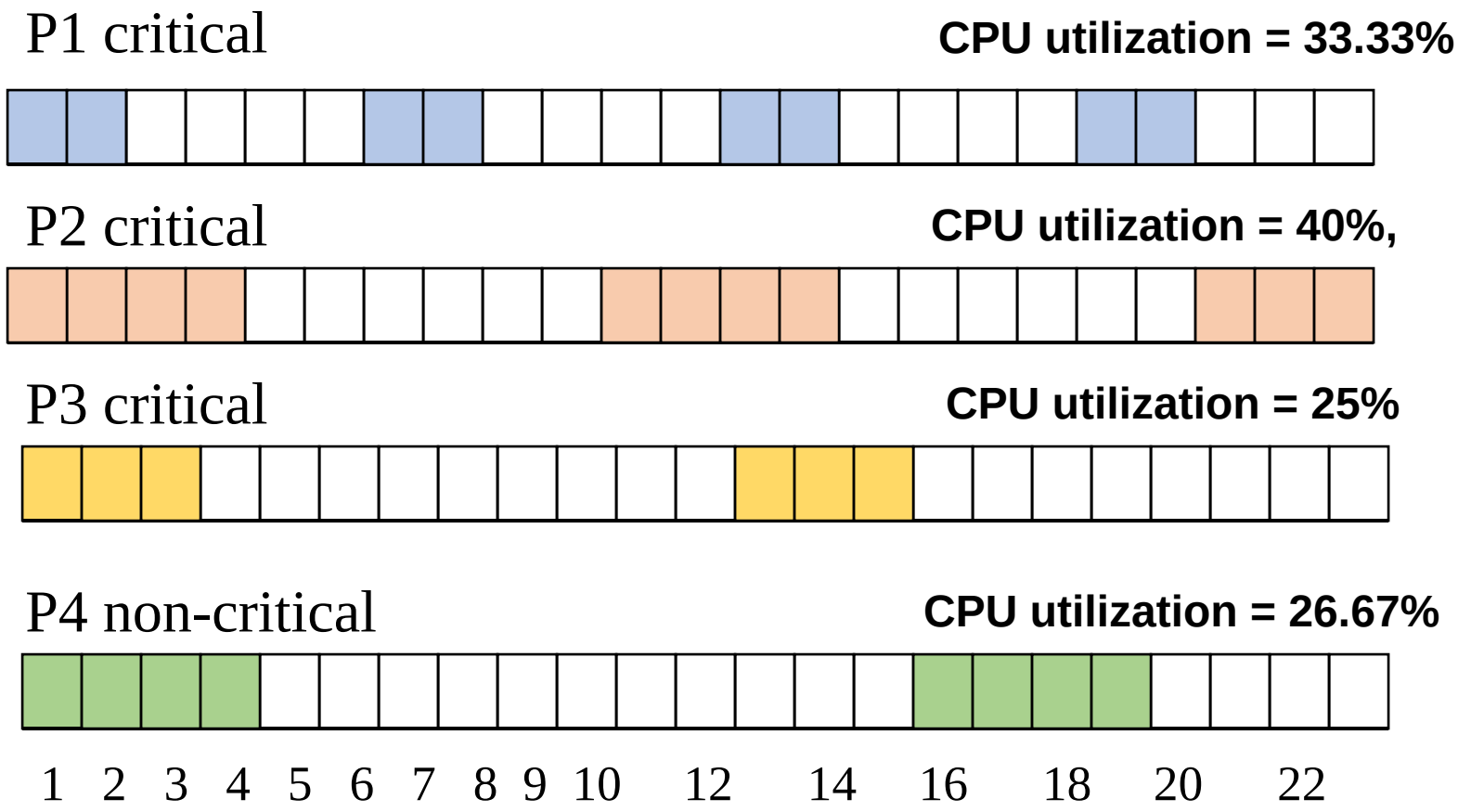
Worst Case Schedulable Bound is $W_N = N\left(2^{1/N} - 1\right)$

Schedulable bounds for Rate monotonic Scheduling

Number of Processes	Schedulable bound
1	100%
2	83%
3	78%
Approaching infinity	69%

- A set of periodic tasks whose CPU utilisation is less than 69% can always be scheduled using rate monotonic scheduling
- Always meets real-time deadlines
- This holds true regardless of the relative timing (phases) of the tasks
- Statistically looks more like 80-90% as critical instant is rare

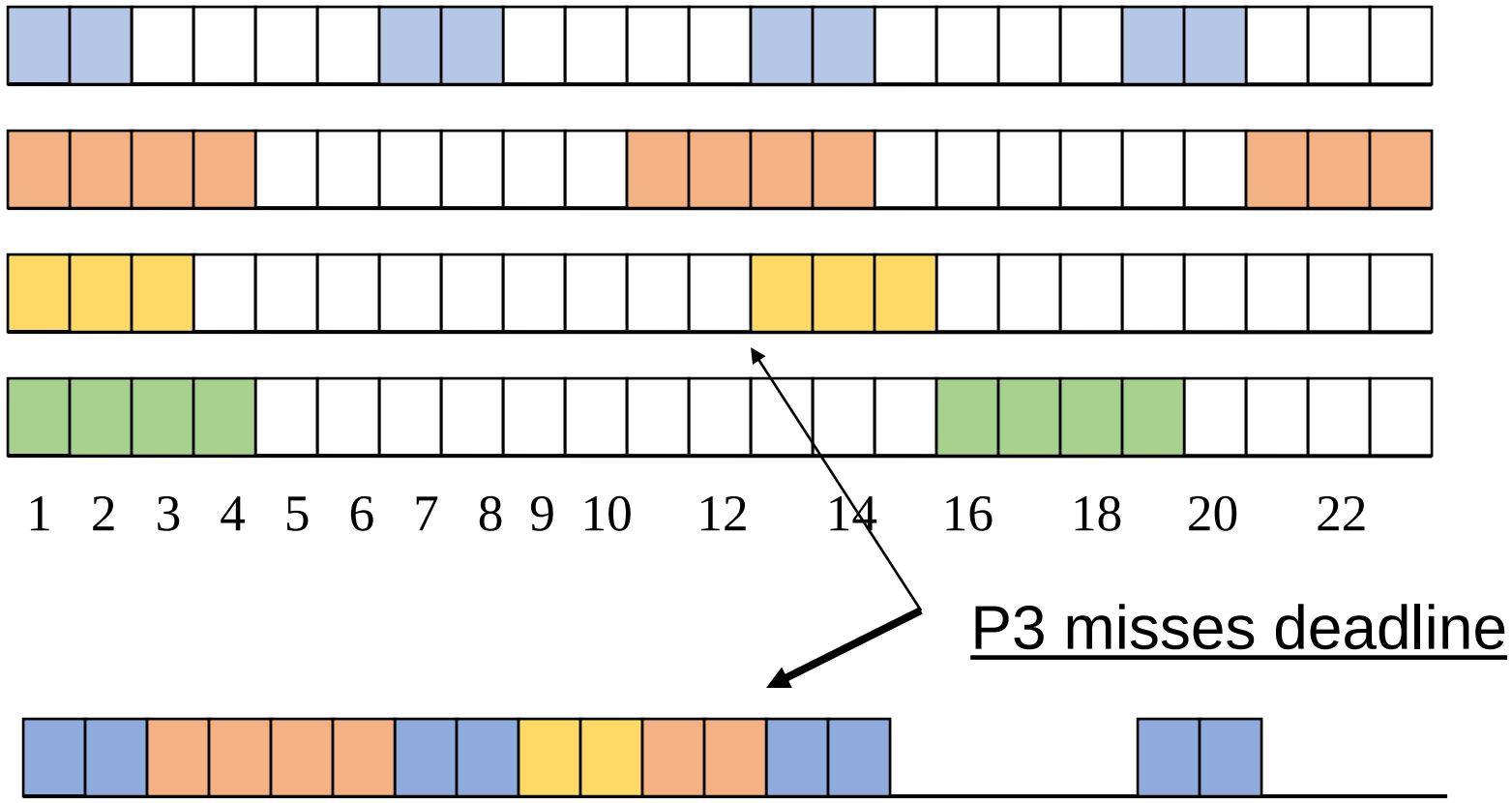
Task set : Example 2



Periods are 6, 10, 12 and 15. Computing times are 2, 4, 3 and 4.

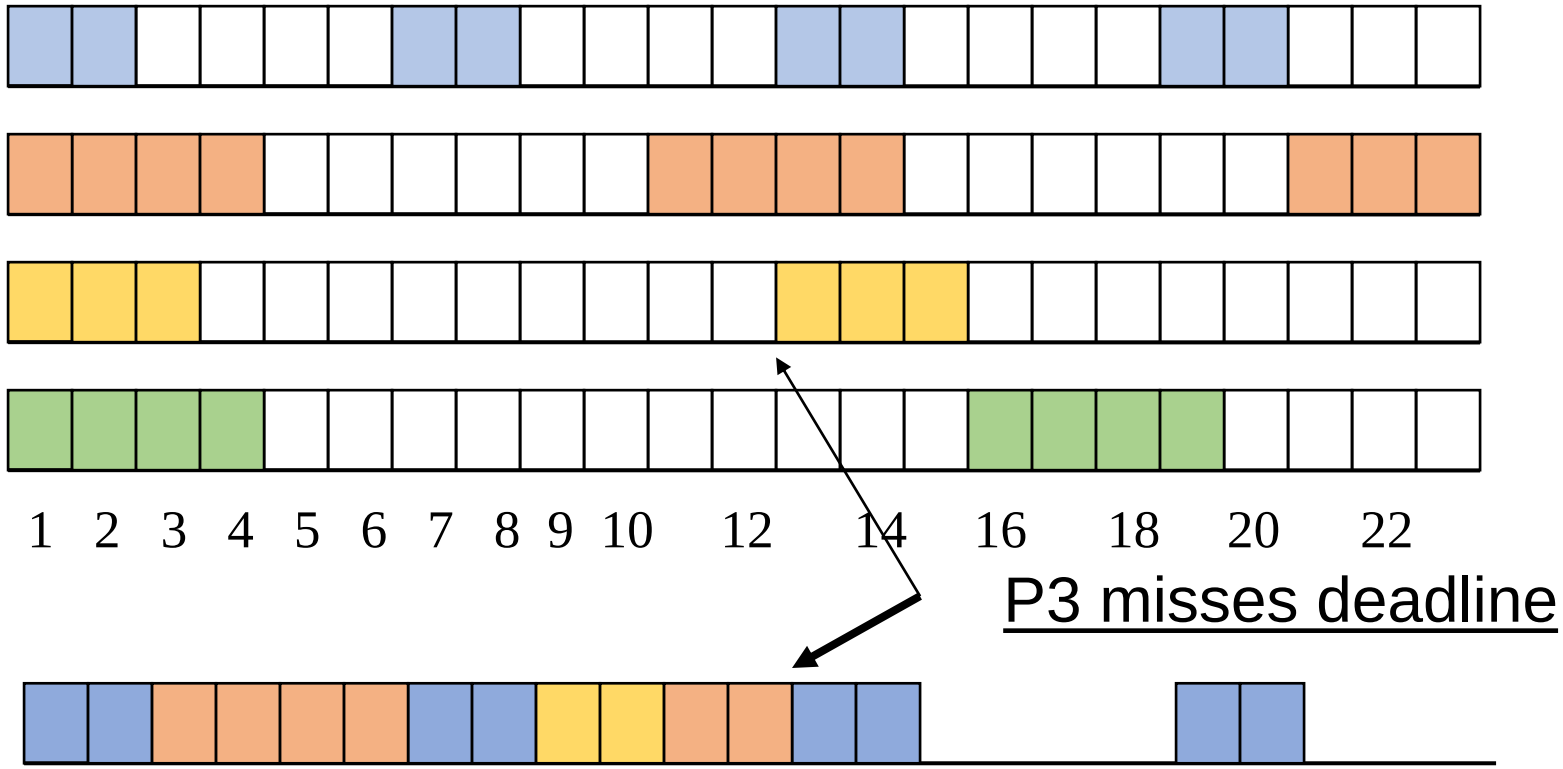
Rate Monotonic Scheduling

Schedulable bound for 3 tasks = 78%. Combined utilization of P1, P2, P3 = 98.33% NOT SATISFIED



Rate Monotonic Scheduling

Schedulable bound for 2 tasks = 83%. Combined utilization of P1, P2 = 73.33% SATISFIED



P1 and P2 can be guaranteed to meet deadlines. In general CPU utilisations are lower and more tasks can be guaranteed to meet deadlines.

Rate monotonic - advantages

- Works well for periodic tasks
- Rule of thumb: Keep CPU utilization below 69%
- Simple because it is a static priority algorithm
- Guaranteed real-time performance under the assumptions (schedulable bound and predictable load conditions)

Rate monotonic Problems

- It does not support dynamically changing periods (for example with sensor based robotic systems, the rate of data acquisition from sensors can vary)

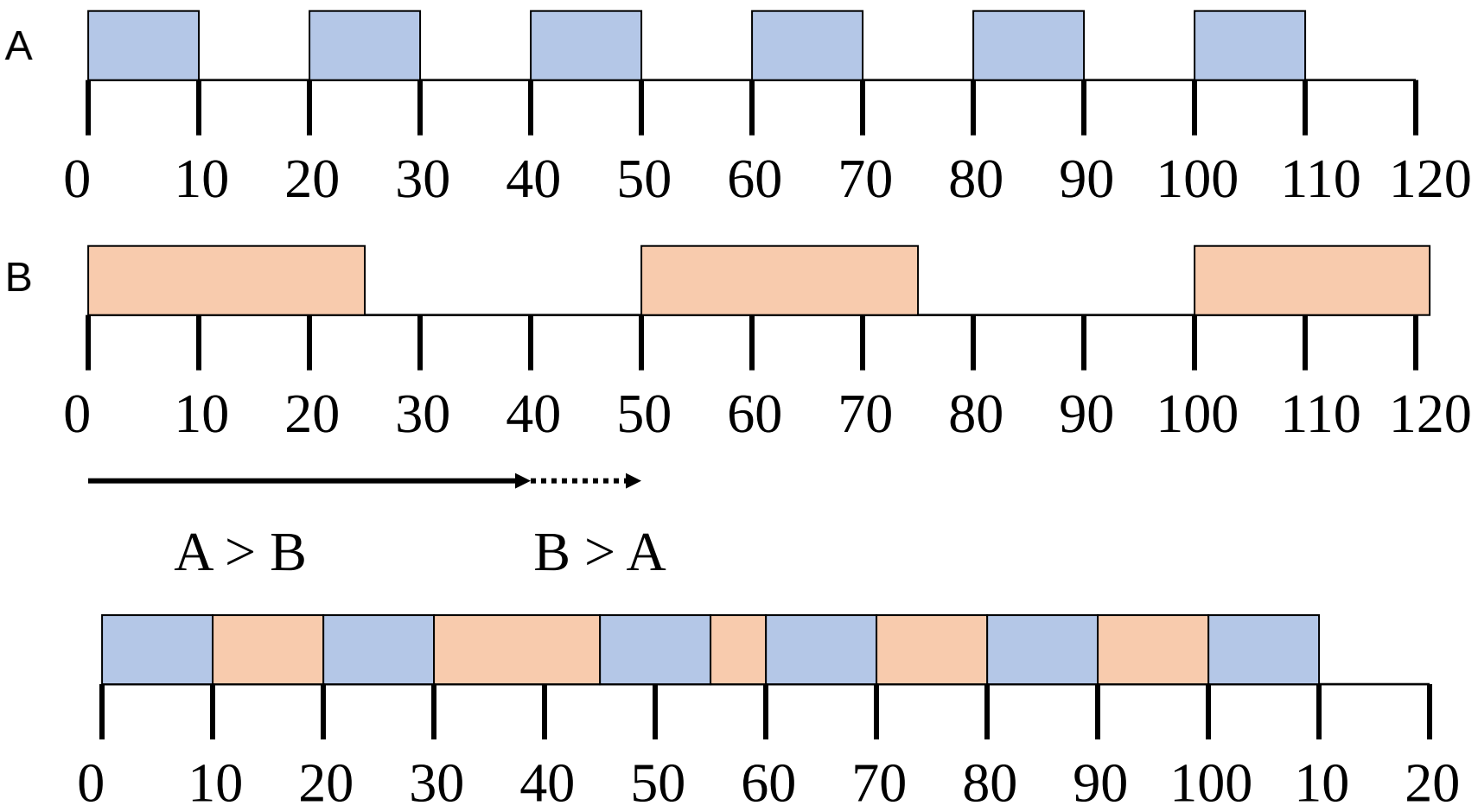
- the worst case schedulable bounds are pessimistic. Often, the CPU can be better utilised while satisfying all time constraints.

- Resource sharing can lead to priority inversion. A solution to that will involve dynamic adjustment of priorities.

Earliest Deadline First

- Dynamically changes priority of tasks during runtime
- Priority determined by distance to deadline
  - Task closest to its deadline has the highest priority
- Can schedule tasks if $U \leq 100\%$

Earliest deadline first



Earliest Deadline First

- Dynamically changes priority of tasks during runtime
  - Harder to implement in practice
- Priority determined by distance to deadline
  - Task closest to its deadline has the highest priority
- Can schedule tasks if $U \leq 100\%$
- Can create a domino effect if a task is overloaded
  - Fails to recover and subsequent tasks will miss deadlines due to dynamic priority (brittle compared to Rate monotonic)

What are other considerations?

- Many systems are now multi-core (or multi-processor) and uni-processor scheduling is not sufficient
  - TI-RTOS supports SMP – symmetric multiprocessing
  - Task affinities are used to determine the N highest priority threads
- Many systems are now distributed (cloud based for example) and message passing overheads must be considered
- Resource sharing incurs wait times when a task is blocked.
  - The effect of this on priorities and scheduling is a critical factor. The problem has been studied by researchers and solutions have been proposed.