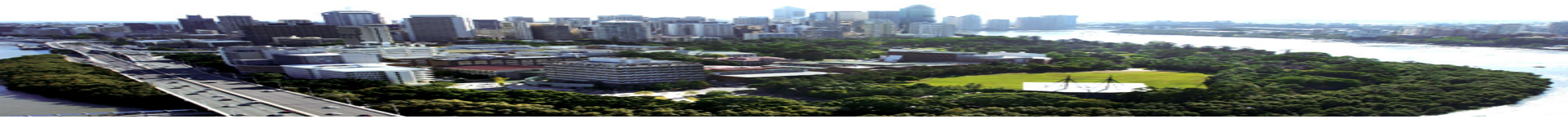
ENB 350 Real-time Computer based systems

Lecture 10 – REAL TIME SCHEDULING

V. Chandran

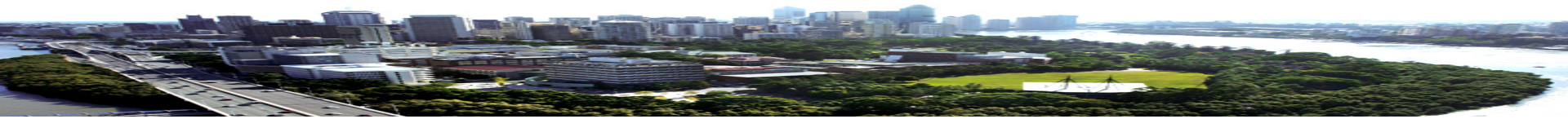


Scheduling

= deciding which task to run next

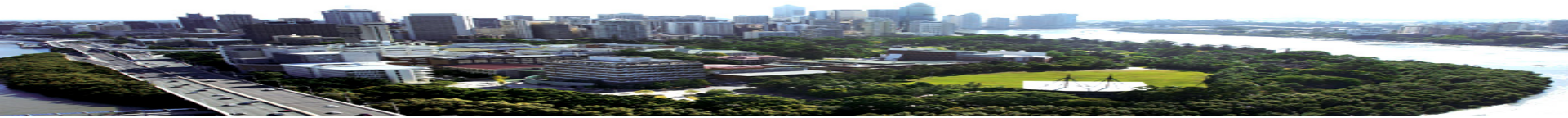
Scheduling Criteria

- fairness
- speed of response
- meeting start and finish deadlines, etc.



Hard and Soft deadlines

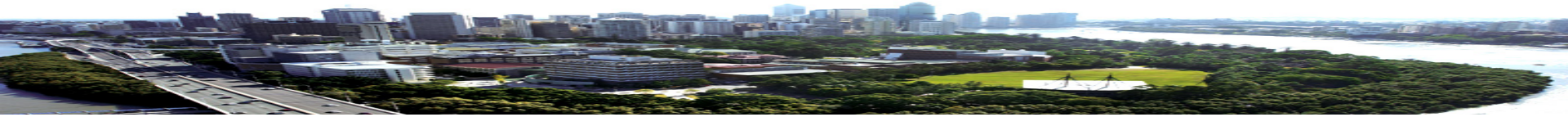
- Hard -> failure to meet will result in system failure and damage (for example, air traffic control, patient monitoring, chemical or nuclear process control)
- Soft -> failure to meet will result in less value for the response and loss of quality (on-line reservation systems)



Characteristics

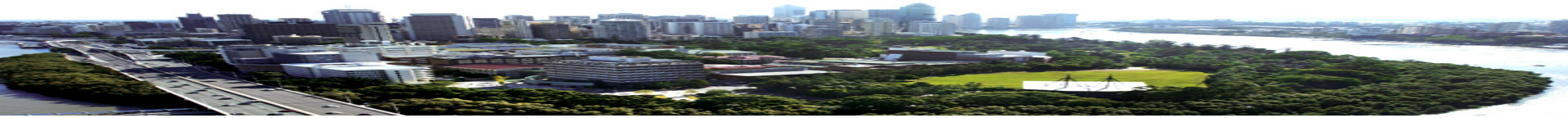
	Hard	Soft
Meet deadline	required	desired
Peak load performance	Predictable	Degraded
Safety	critical	Non-critical
Error detection	Must be autonomous	Can be user assisted
Control of response	Environment	Computer
Data integrity	Short term	Longer term

Adapted from the source:
http://www.ece.cmu.edu/~koopman/des_s99/real_time/



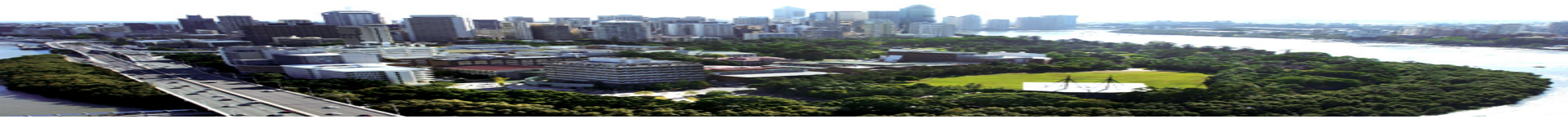
Why is scheduling important?

- All ‘hard’ deadlines associated with critical tasks must be met in a real-time system
- ISRs are kept short – so useful work is done by the tasks that compete for CPU time and need to be scheduled
- Response times for tasks with ‘soft’ deadlines should be within criteria for good performance.
- CPU must be utilized efficiently



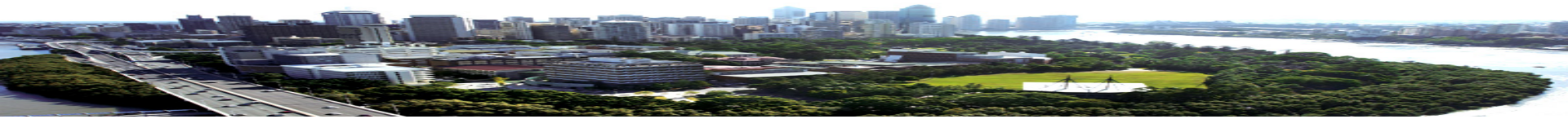
What does a scheduler do?

- Needs to take a decision – on the next task to run. Examines the ready queue or list
- An algorithm that considers a decision parameter for each task in the ready queue is used
- The decision parameter may be called the priority of the task.



When does the scheduler run?

- Upon return from any interrupt the scheduler may be invoked
- Timer interrupts (for slices of execution) can be scheduling decision points but not necessarily, depending on the scheduling algorithm.
- In non-preemptive systems task switching only occurs when the currently running processes ‘yields’ and relinquishes control voluntarily

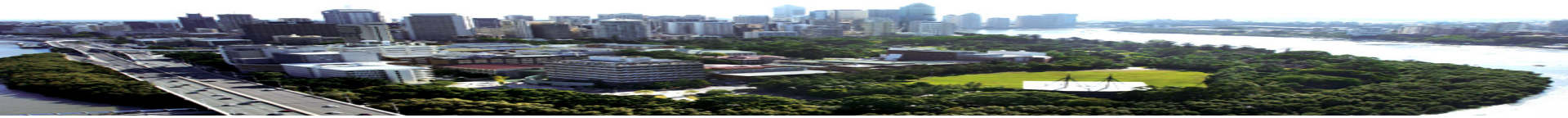


Types of scheduling

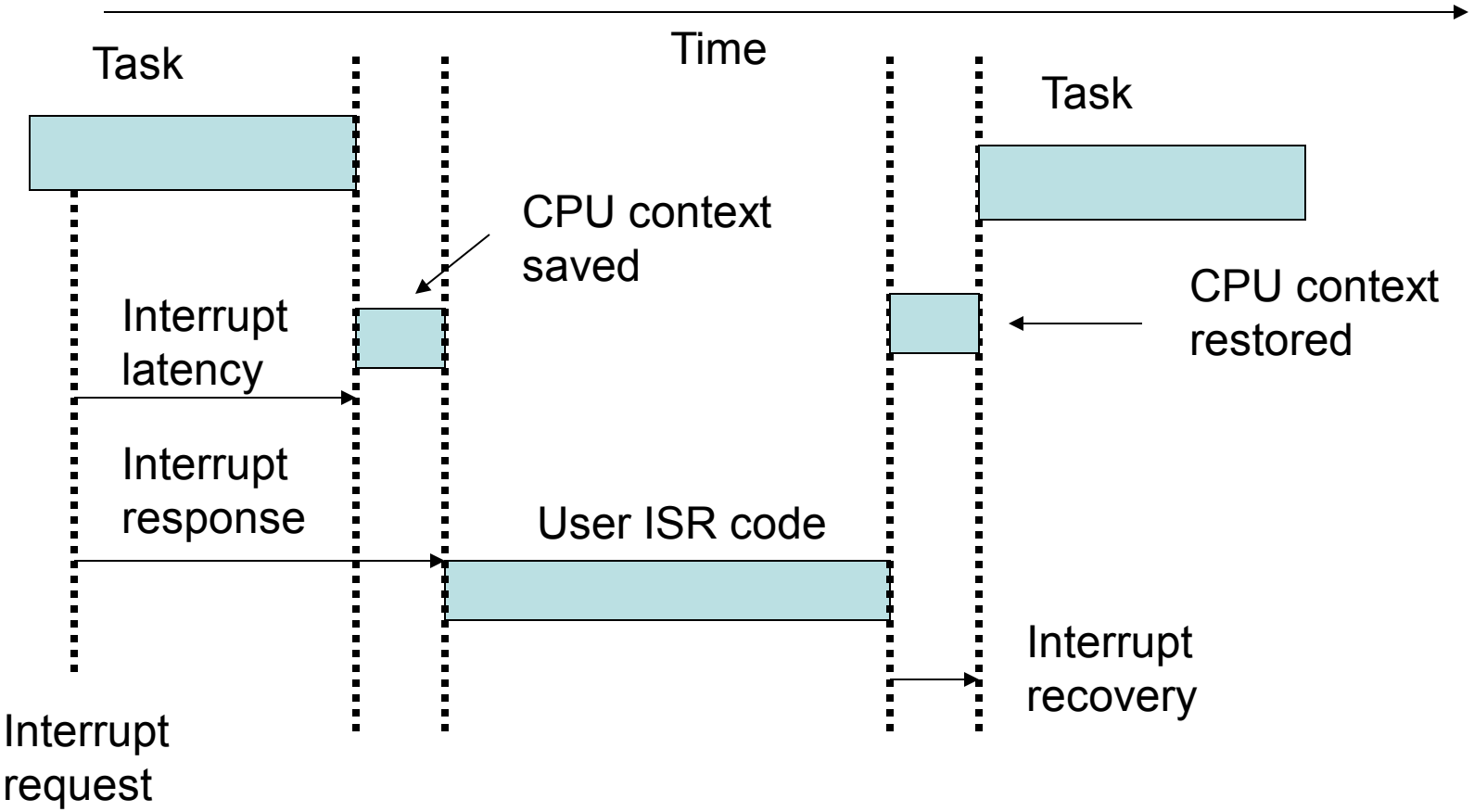
- Non pre-emptive
- Pre-emptive
- Immediate pre-emptive

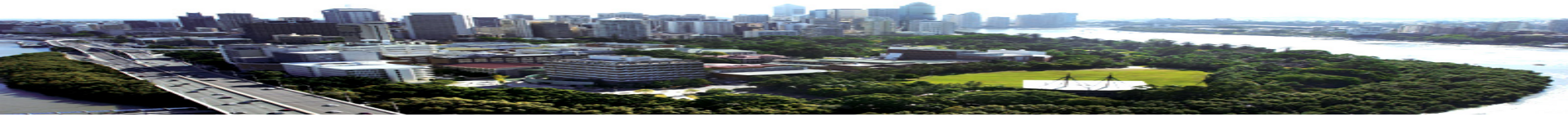
- Not priority driven
- Static priority
- Dynamic priority

In a real-time system, characteristics of the workload (how long a task might run, its frequency, peak loads) are often precisely known. By contrast, in a time-sharing system, these can be very random.

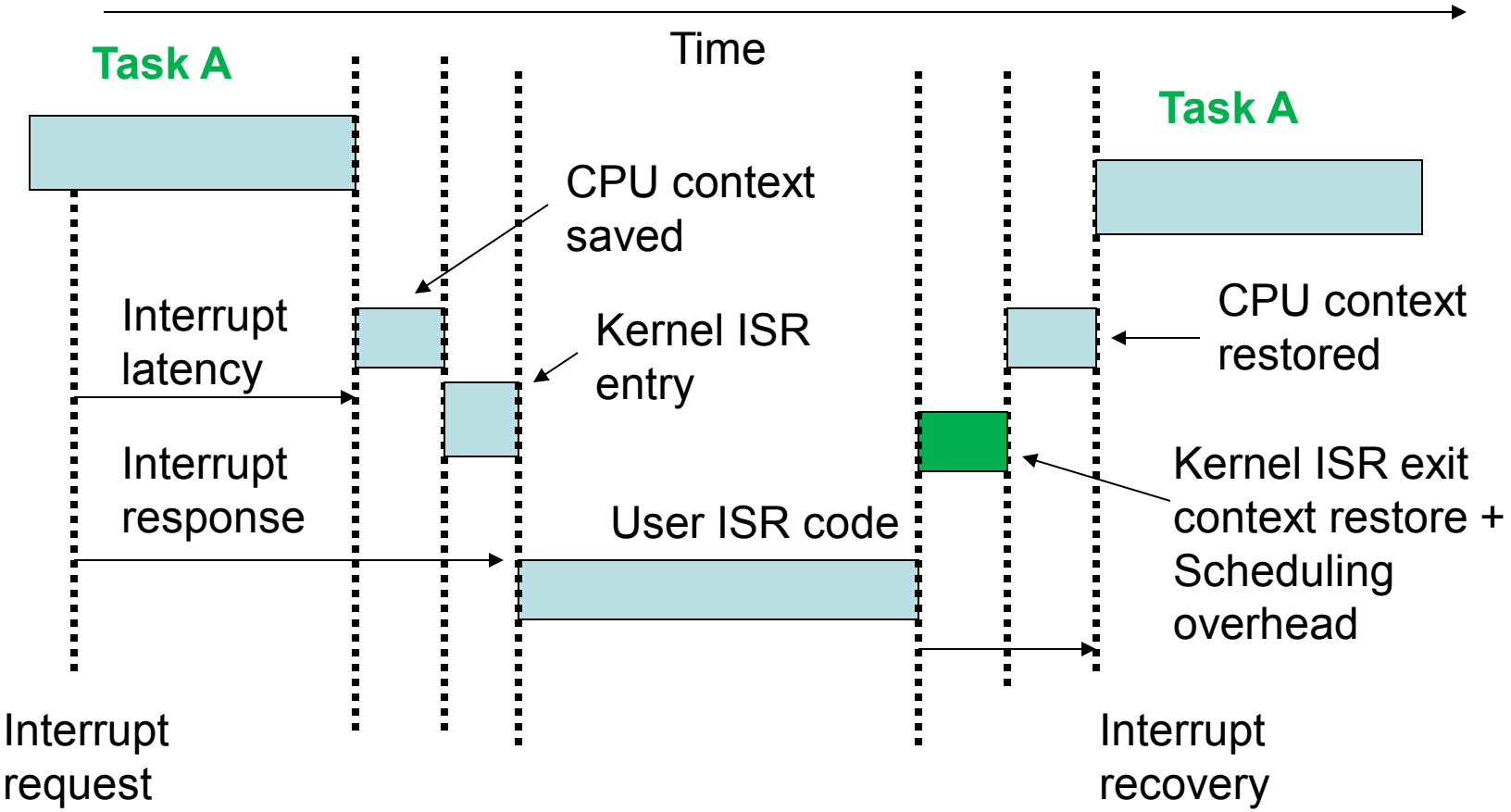


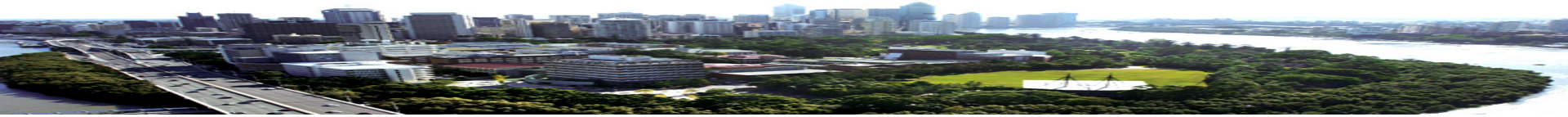
Interrupt: non pre-emptive kernel



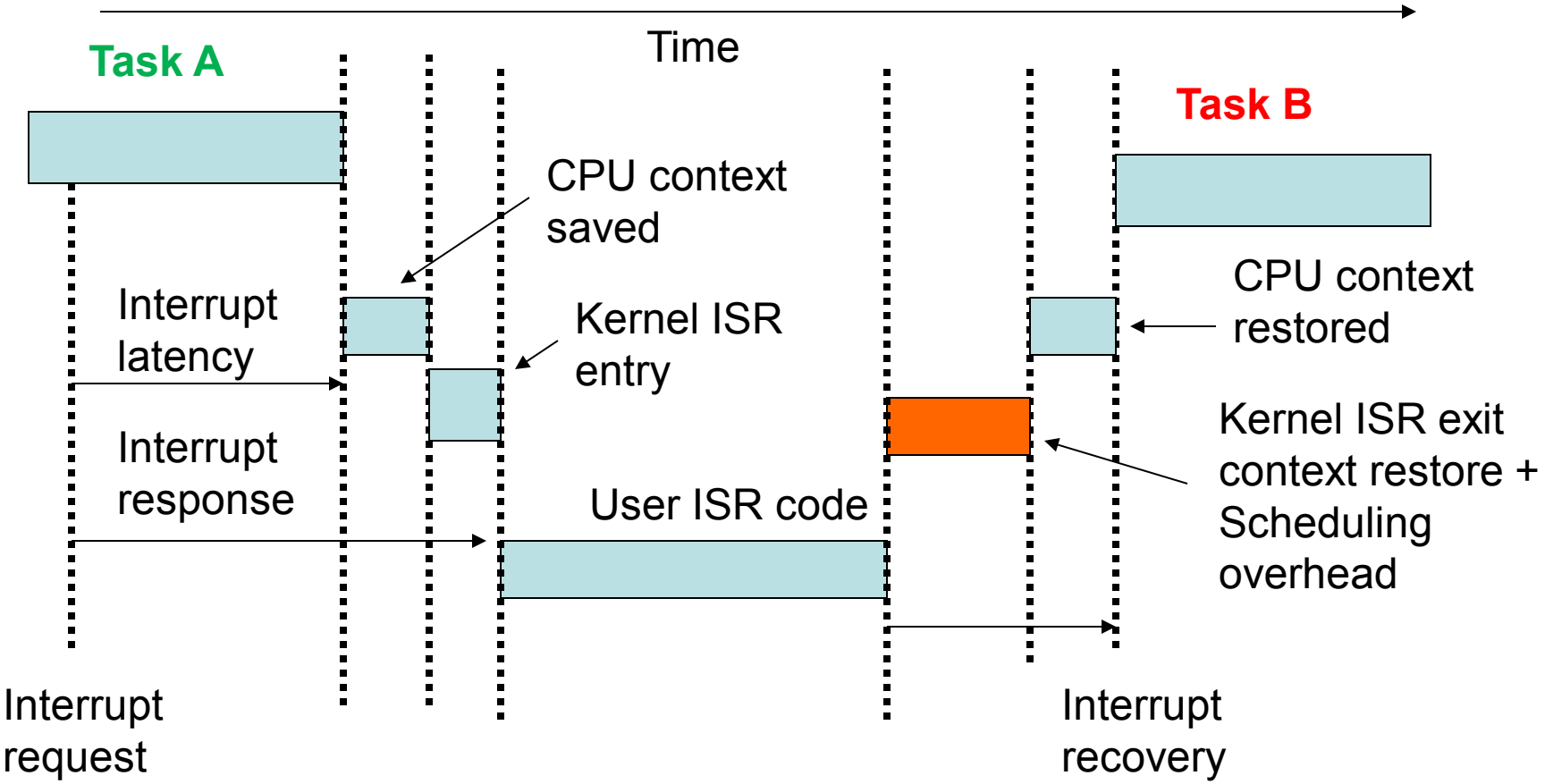


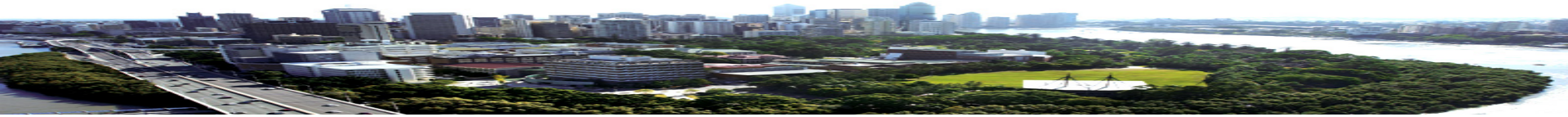
Interrupt: pre-emptive kernel





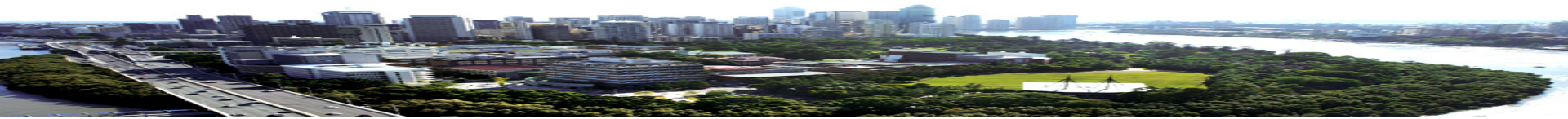
Interrupt: pre-emptive kernel



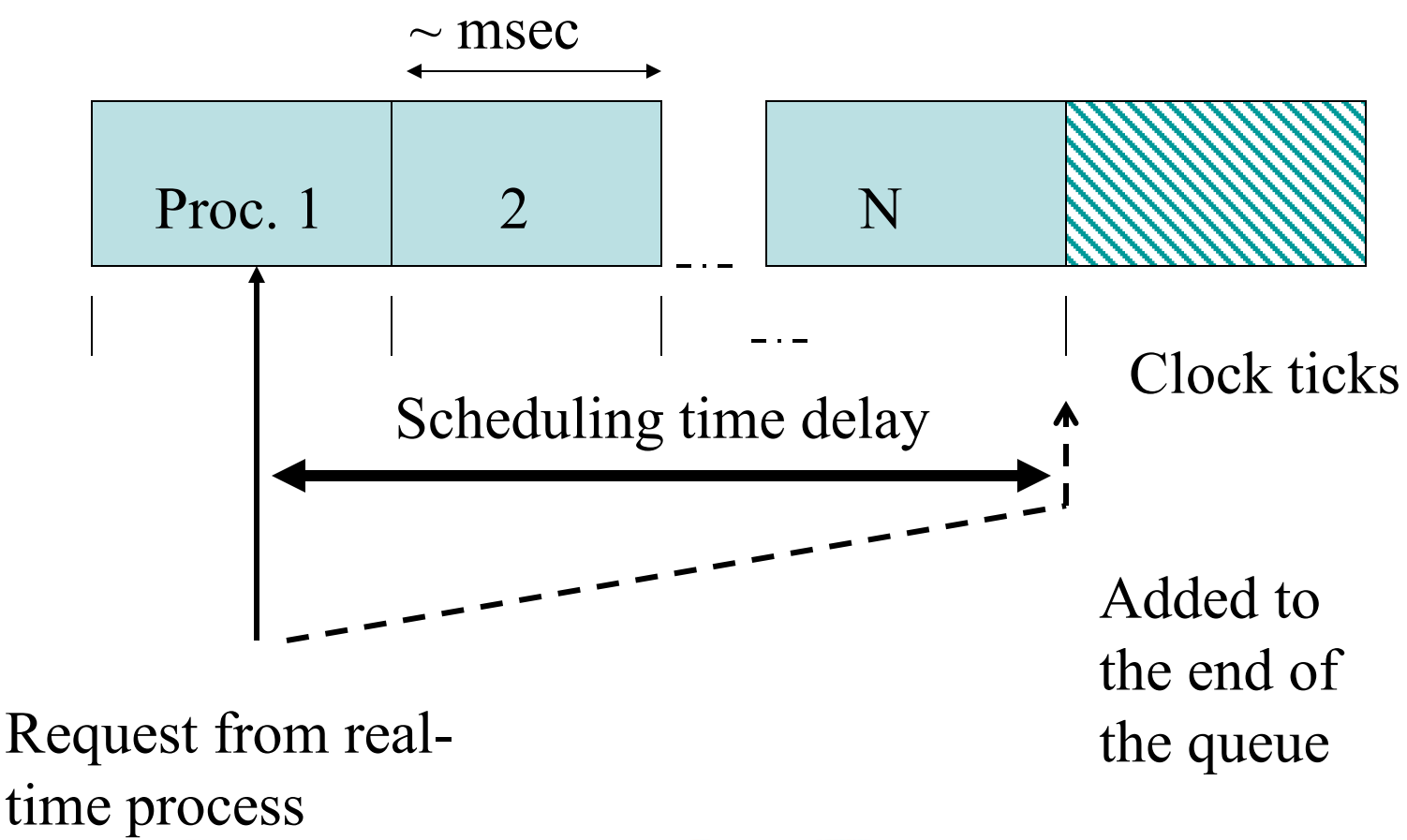


Scheduling algorithms

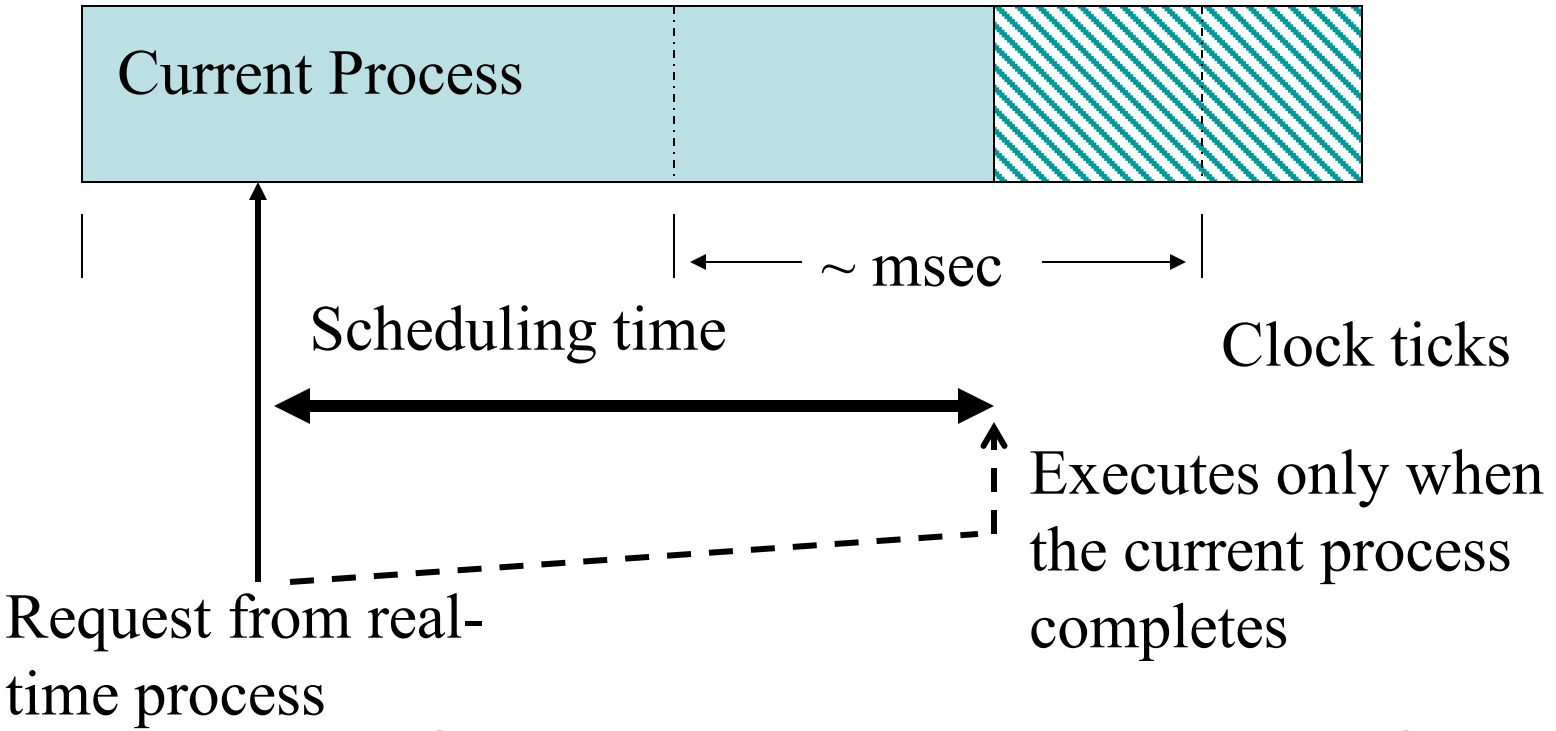
- Round Robin
- First come first serve
- Earliest deadline
- Minimum laxity
- Rate monotonic
- Maximum urgency etc.



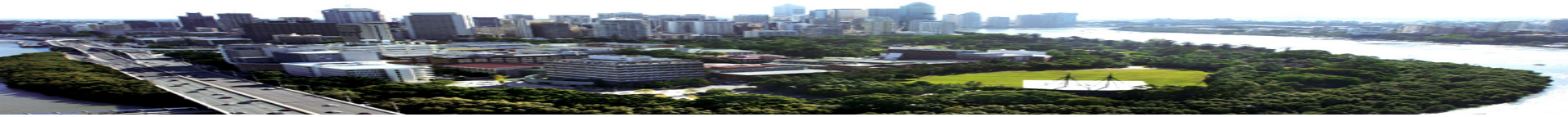
Round Robin


~ msec
Proc. 1
2
N
Clock ticks
Scheduling time delay
Added to the end of the queue
Request from real-time process

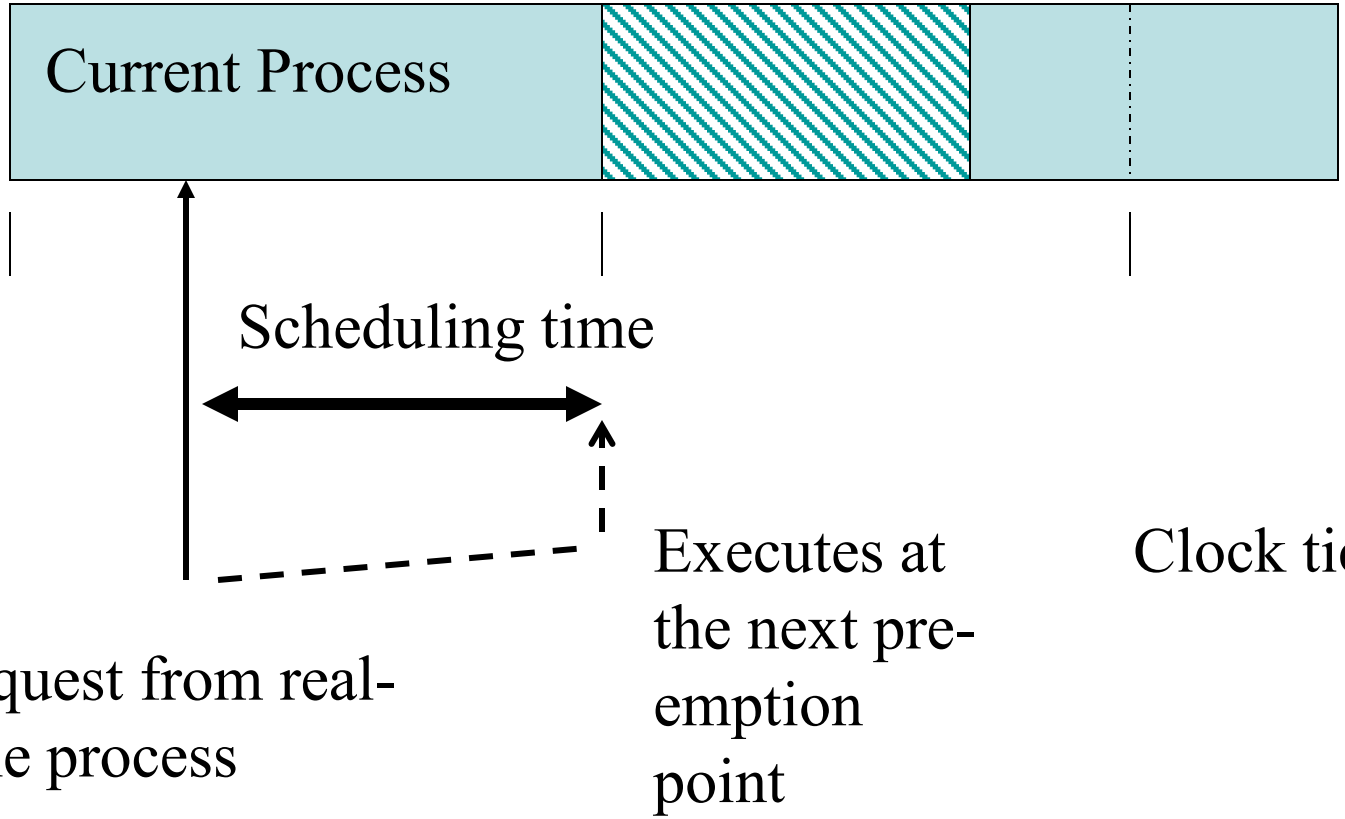
Priority driven, not pre-emptive

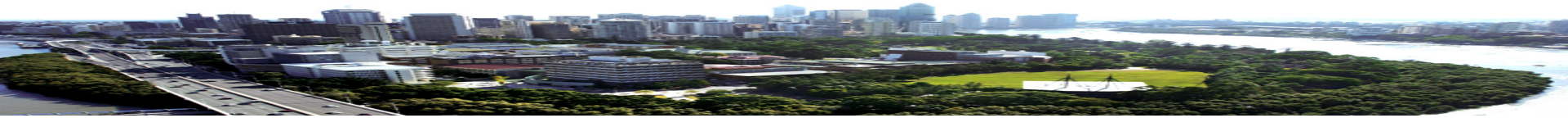


Clock ticks are usually 10 or 100 ms. Derived from a timer. Faster clock ticks also imply greater overhead.

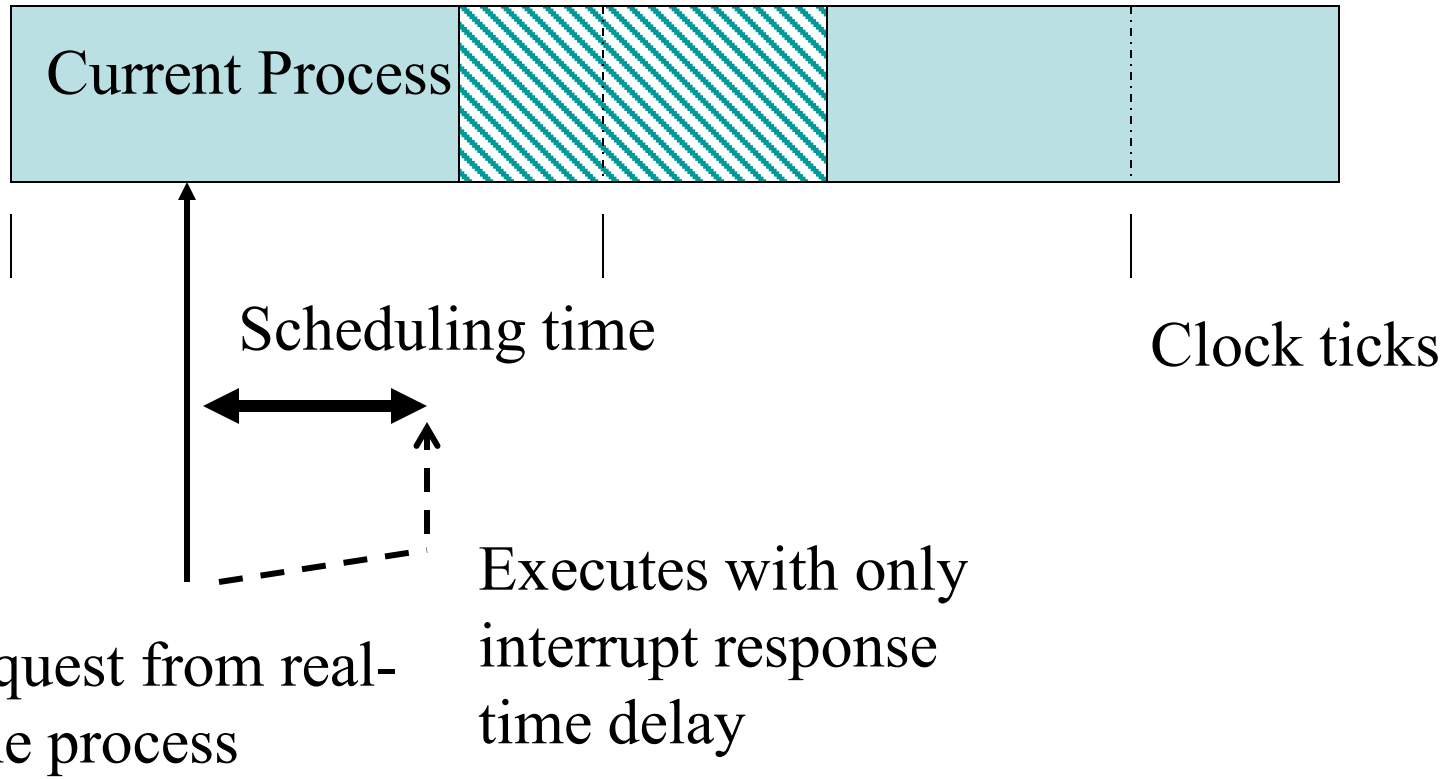


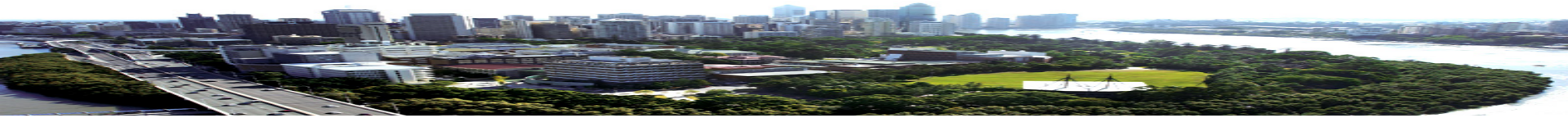
Priority driven pre-emptive



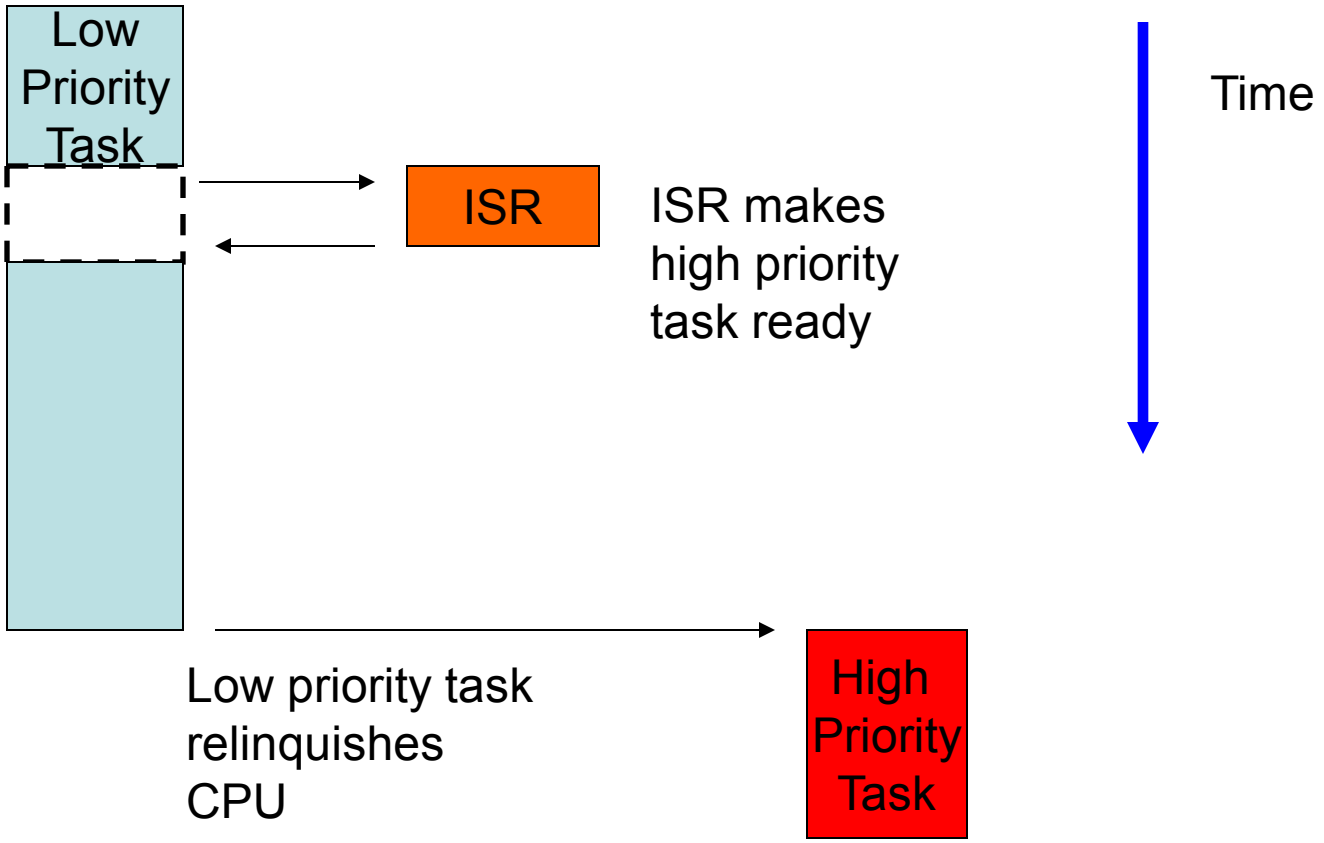


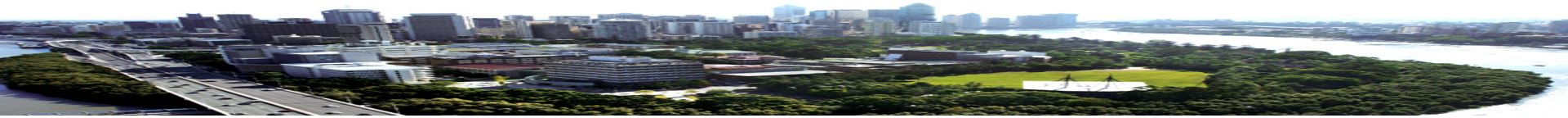
Priority driven, immediate pre-emptive



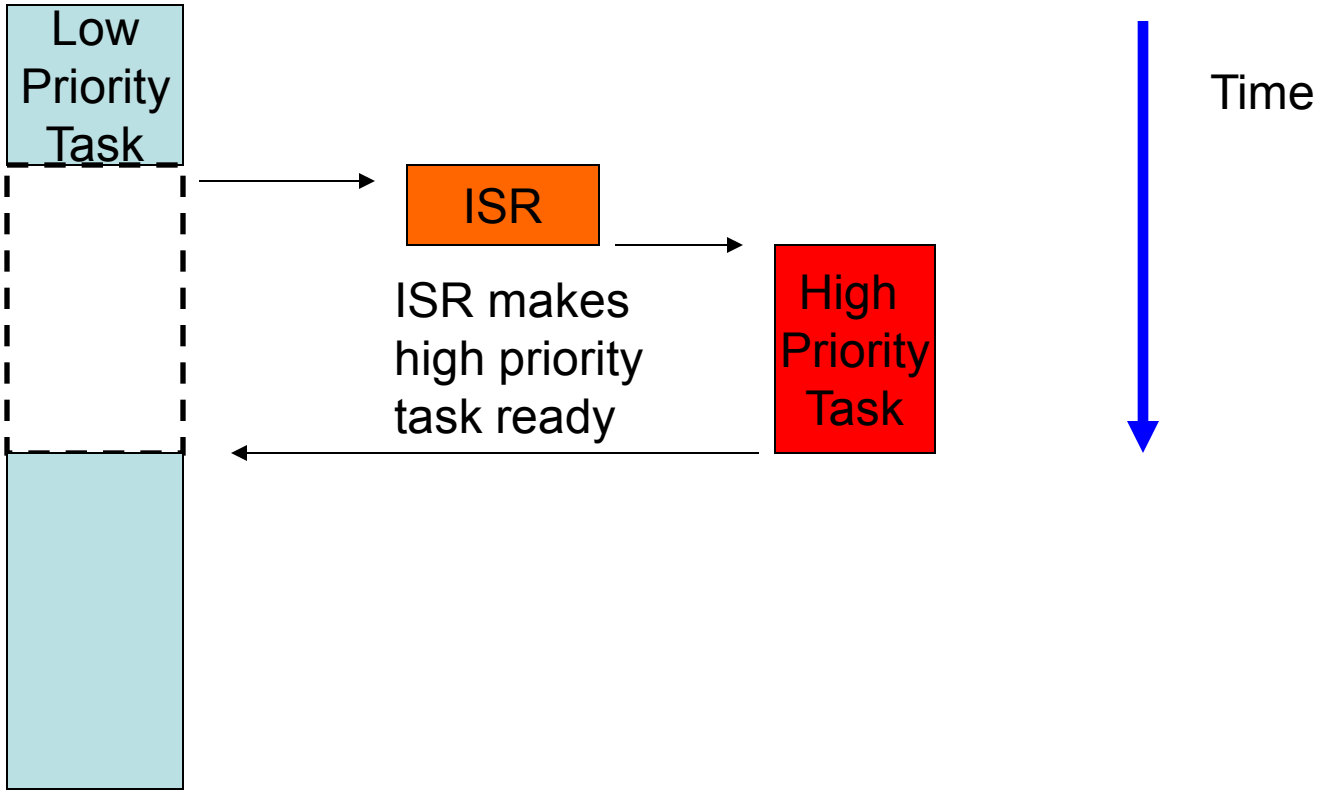


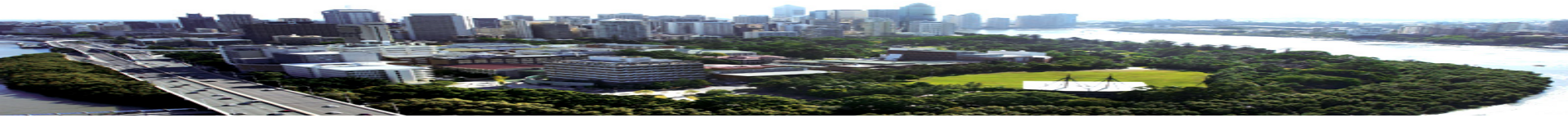
Non pre-emptive





Immediate pre-emptive





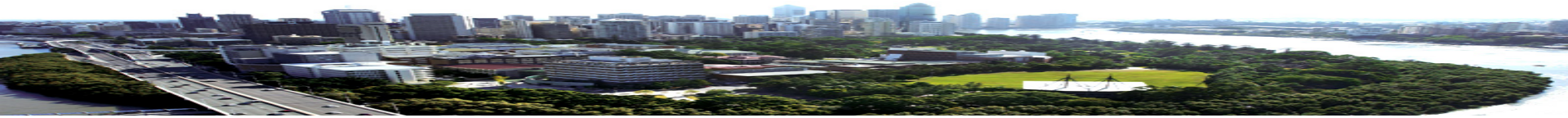
Priorities

Static

- fixed regardless of time or mix of processes
- example: rate monotonic algorithm

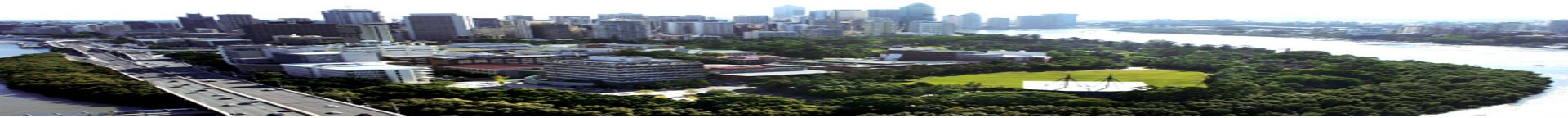
Dynamic

- change with time and task set
- examples: earliest deadline, minimum laxity, etc.



CPU utilisation : The fraction of time spent executing useful tasks (as opposed to idling)

Schedulable bound : For a task set, the maximum CPU utilisation for which the set of tasks can be guaranteed to meet their deadlines.

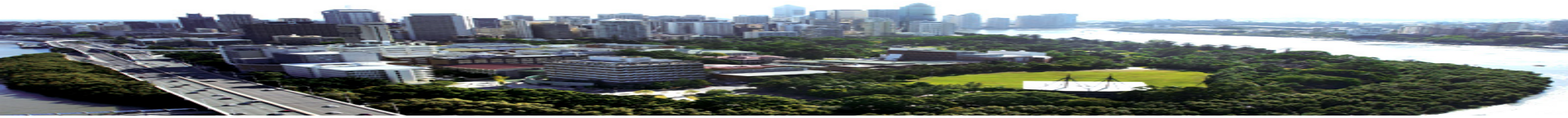


Real-time Scheduling of Periodic Processes

- Assume scheduling decision every 10 msec
- deadline = completion time = start of the next period
- only ready processes are considered

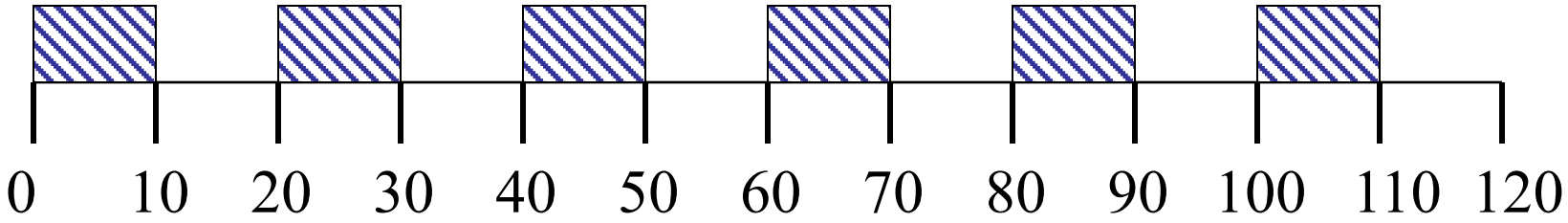
Example 1.

- Two processes, CPU utilisation 100%
- Static priority scheduling cannot guarantee completion within deadlines
- Earliest deadline first does guarantee timely completion for this particular task set

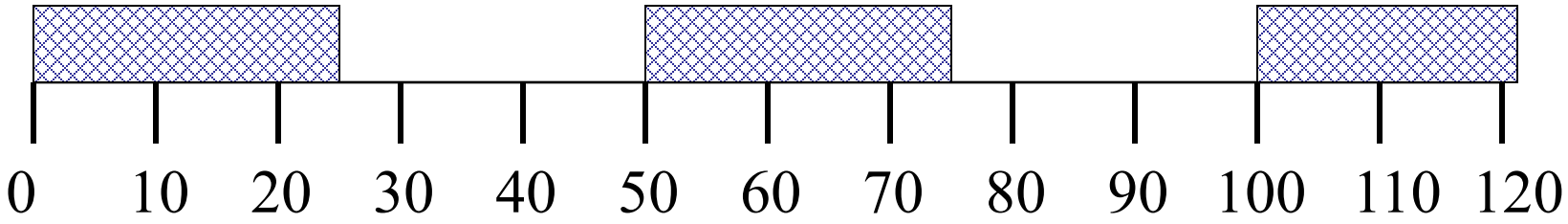


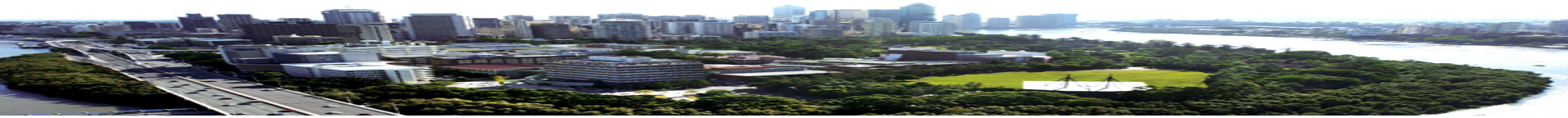
Task set for example 1

Periodic task A

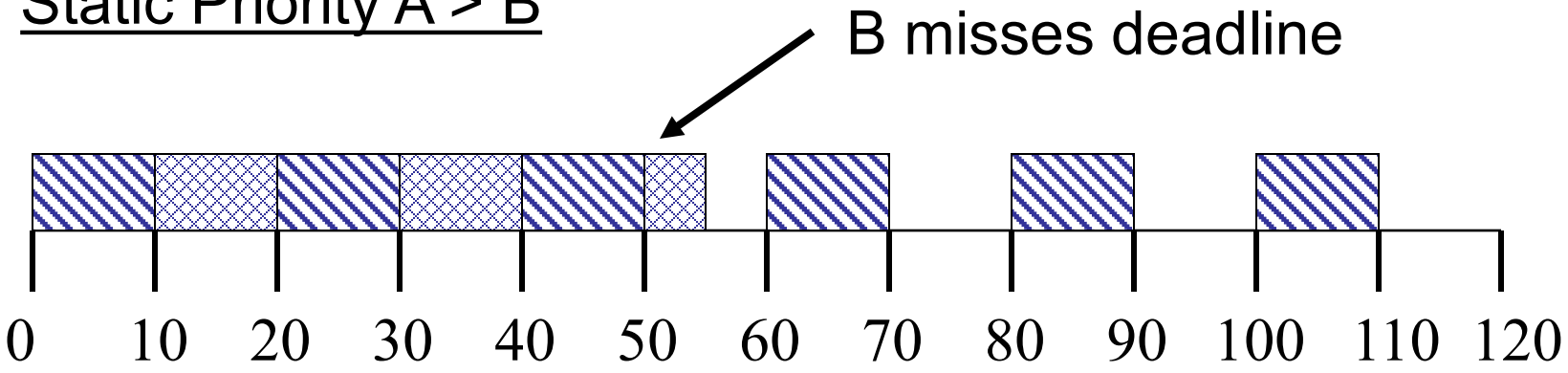


Periodic task B

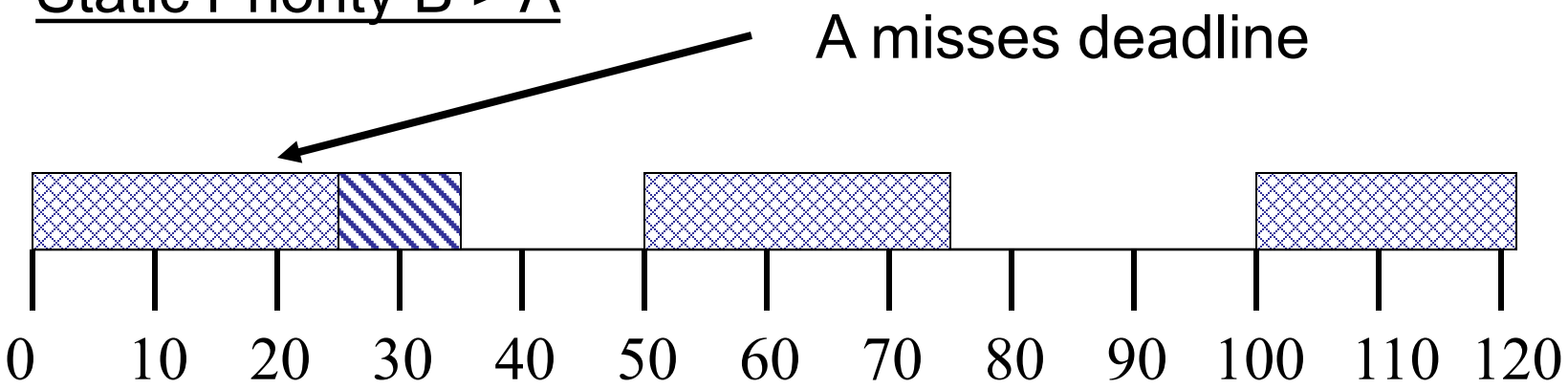


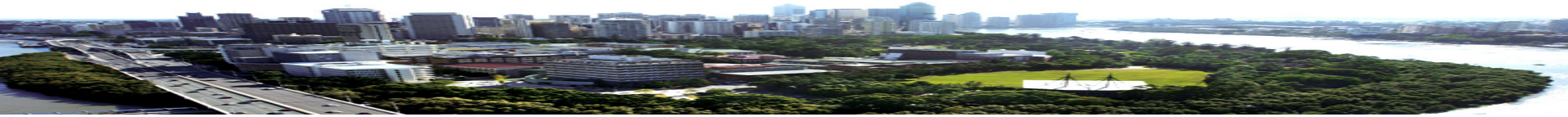


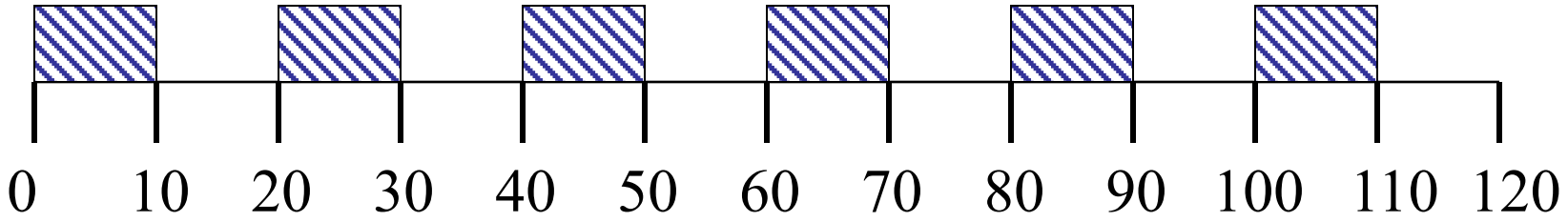
Static Priority A > B

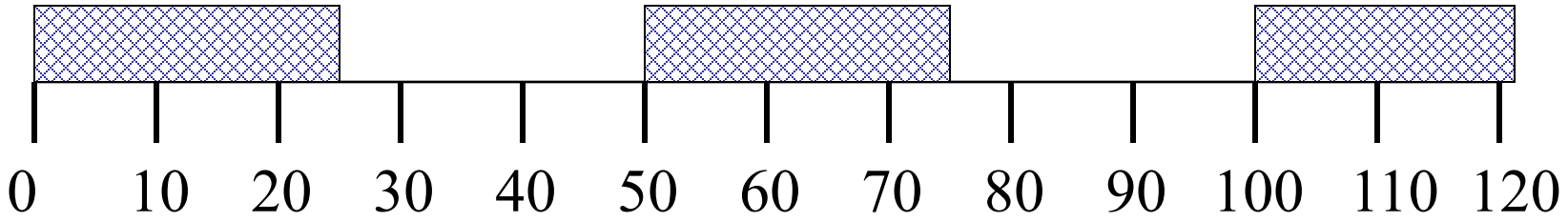


Static Priority B > A



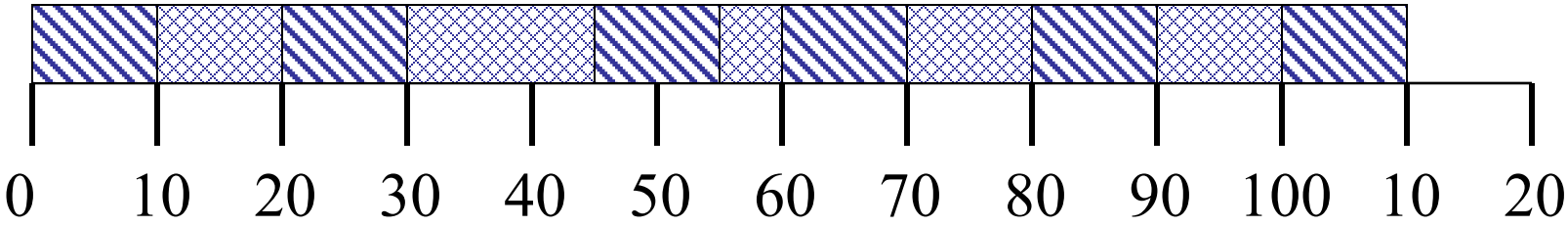


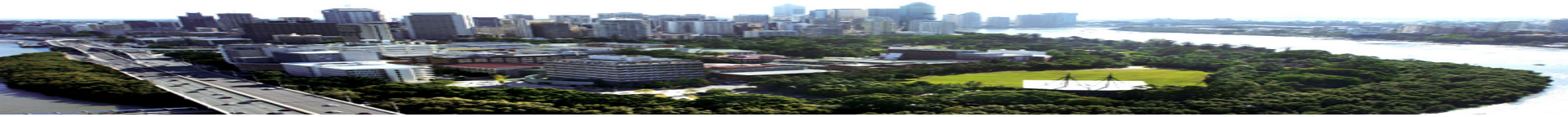




A > B B > A

Earliest deadline first





Rate monotonic algorithm

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

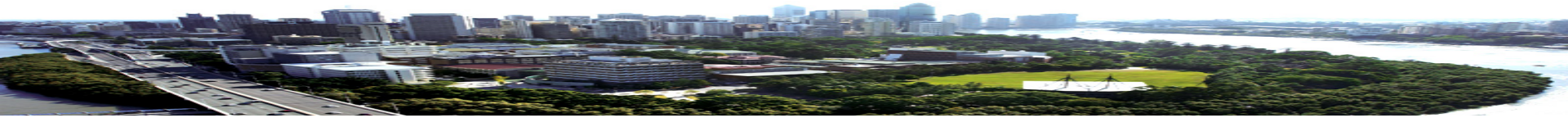
Tasks $i = 1,2,...,N$

Computing times C_i

Periods T_i

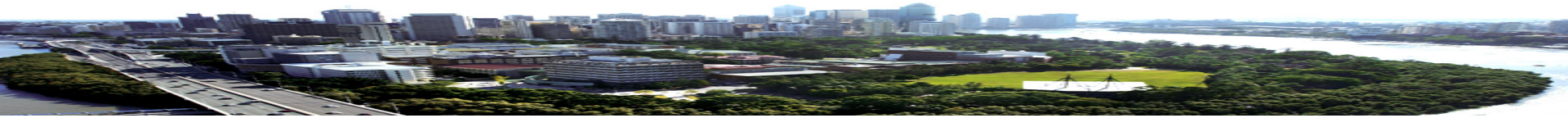
Total CPU utilisation is $U_N = \sum_{i=1}^{N} \frac{C_i}{T_i}$

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



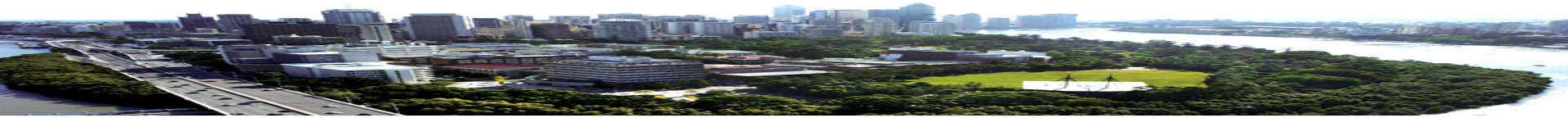
Schedulable bounds for the Rate monotonic algorithm

Number of Processes	Schedulable bound
1	100%
2	83%
3	78%
very large	tends to 69%



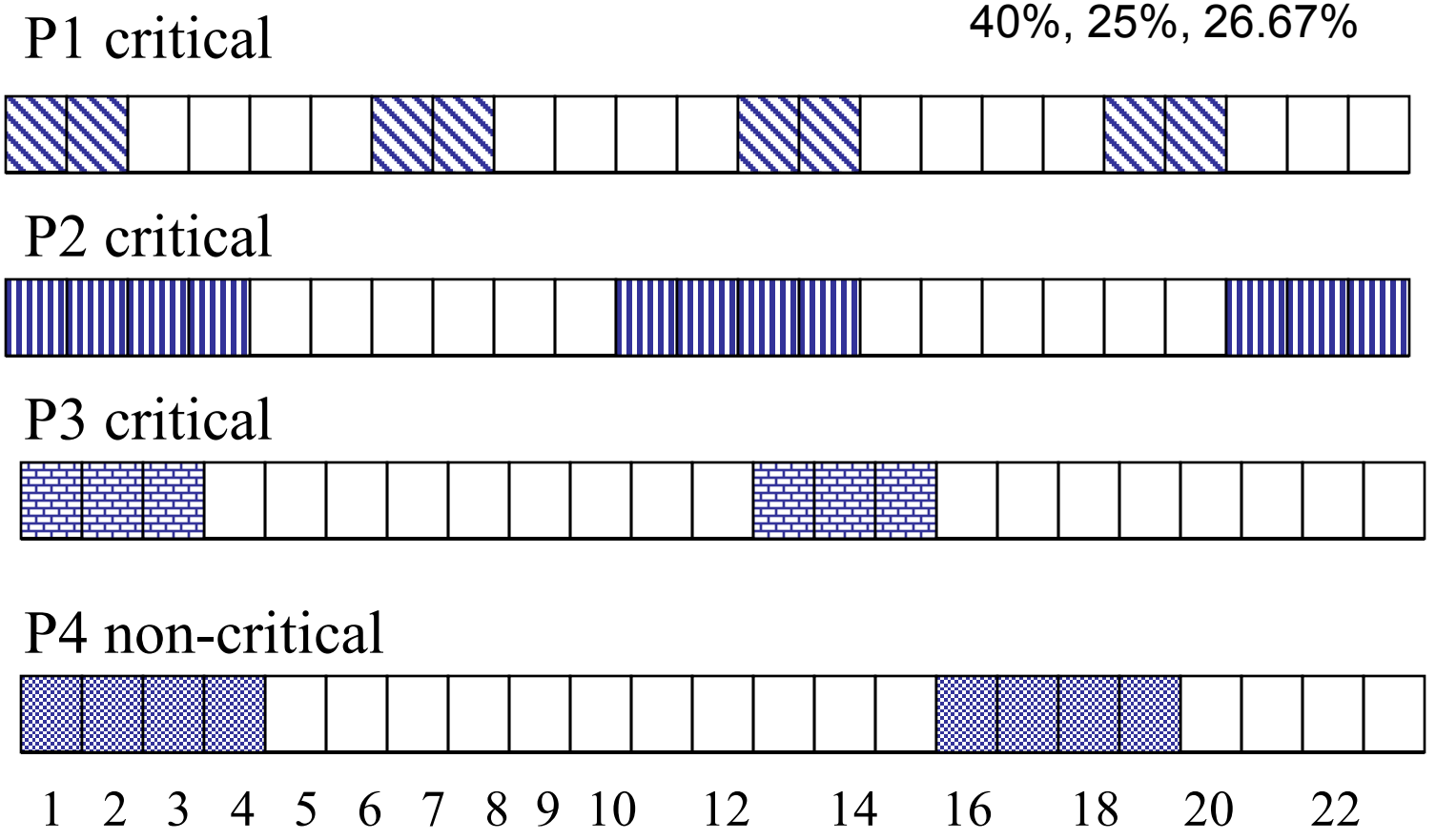
Schedulable bounds for the Rate monotonic algorithm

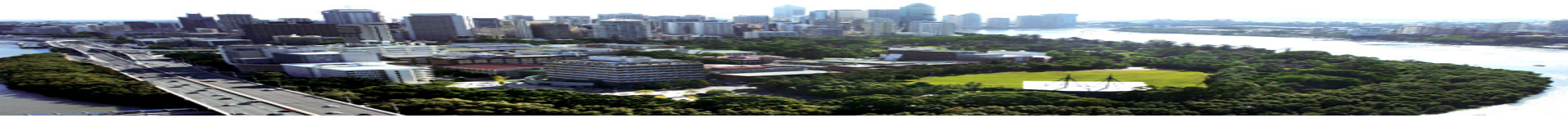
- A set of periodic tasks whose CPU utilisation is less than 69% CAN ALWAYS BE SCHEDULED USING THE RATE MONOTONIC ALGORITHM TO MEET ALL REAL-TIME DEADLINES
- This holds true regardless of the relative timing (phases) of the tasks
- A scheduling decision is not necessary at every time slice (few milliseconds). Scheduling points for a task are multiples of the periods of all higher frequency tasks.



Task set : Example 2

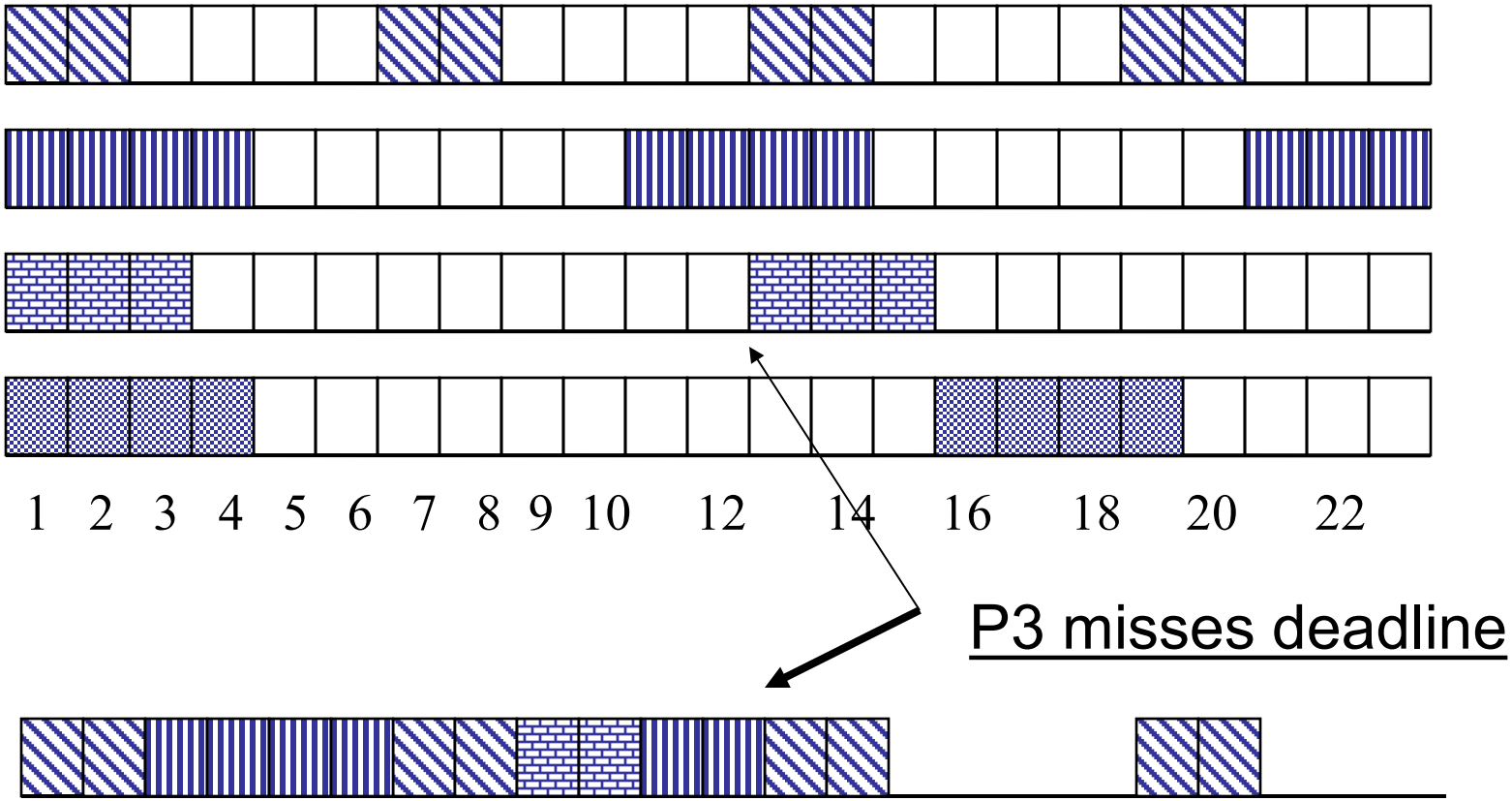
CPU utilization = 33.33%,
40%, 25%, 26.67%

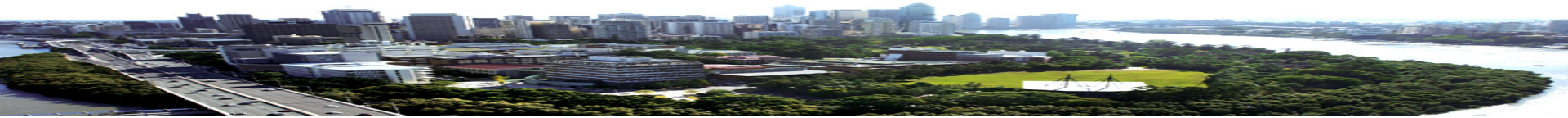




Rate Monotonic Scheduling

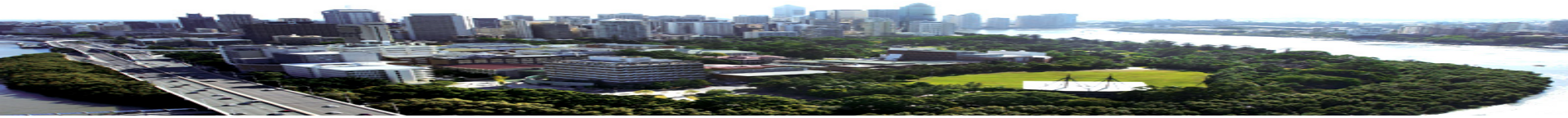
Schedulable bound for 3 tasks = 78%. Combined utilization of P1, P2, P3 = 98.33% NOT SATISFIED





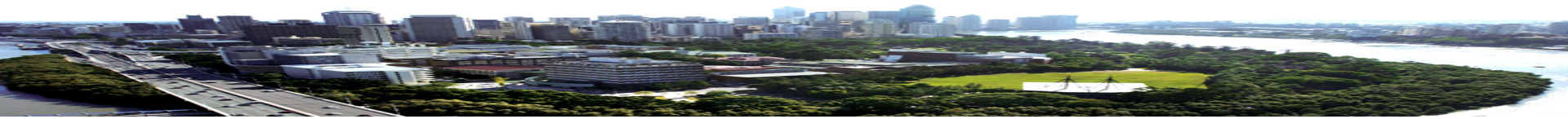
Rate monotonic - advantages

- Works well for periodic tasks
- Rule of thumb: Keep CPU utilization below 60-70%
- Simple because it is a static priority algorithm
- Guaranteed real-time performance under the assumptions (schedulable bound and predictable load conditions)



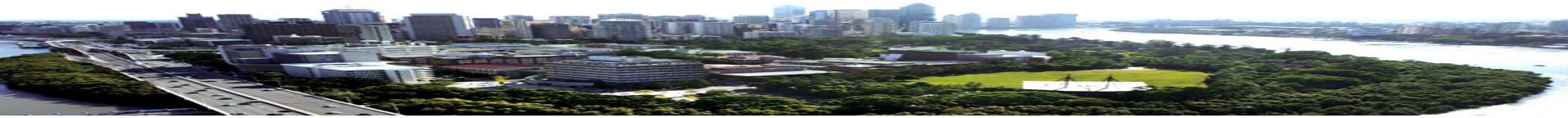
Rate monotonic -problems

- 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.



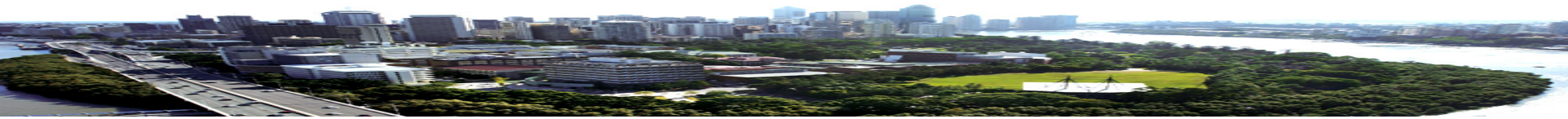
ucos-II support for RT scheduling

- Priority can be set and changed
- Priority is an integer. 0 is highest priority. OS_LOWEST_PRIO is the lowest priority (assigned to the idle task on initialization).
- Priority must be unique. No two tasks can have the same priority
- Highest priority task runs next



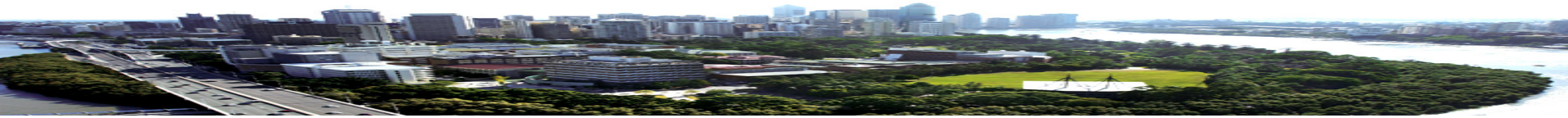
Key considerations

- Scheduling time is an overhead – it must be kept small
- Preferably all scheduling decisions must be made in the same time
- Preferably this overhead must be independent of the number of tasks

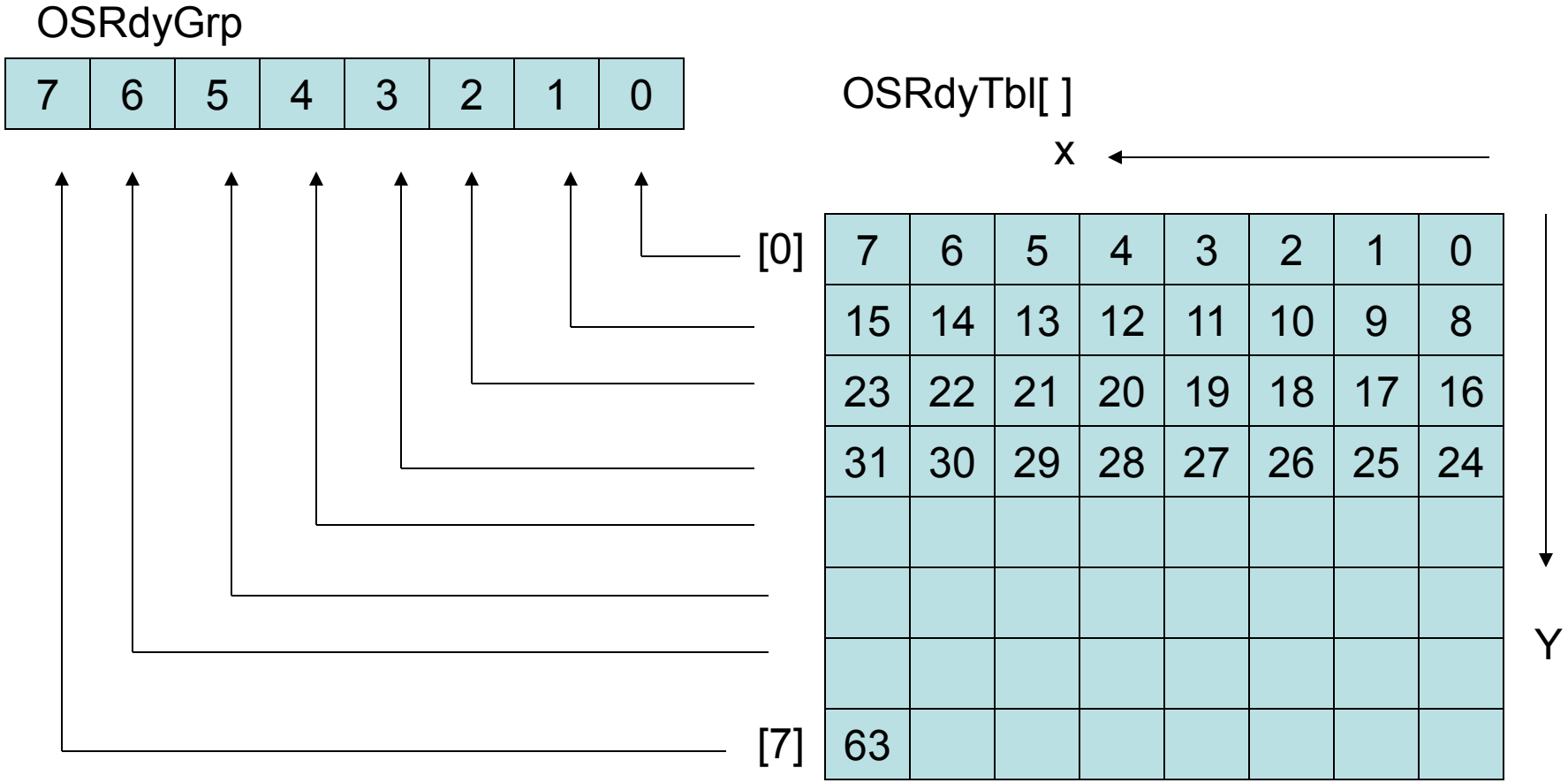


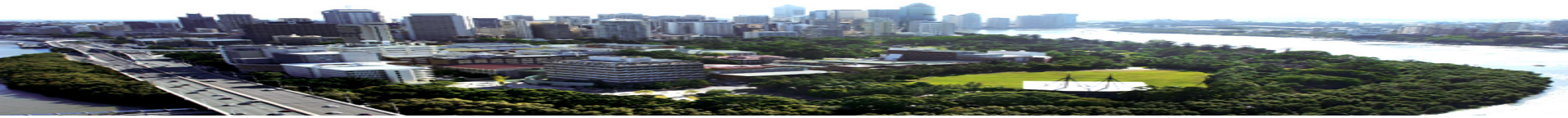
Ucos-ii scheduling

- Each task is placed in a ready list
- Two variables: OSRdyGrp and OSRdyTbl[].
- Task priorities are grouped (8 tasks per group) in OSRdyGrp.
- The size of OSRdyTbl[] depends on OS_Lowest_Prio which sets the number of priority levels. If lowest priority is 63, size is 63/8+1 = 8 bytes.
- Each bit in OSRdyGrp indicates when any task in a group is ready to run (any bit set in the row)

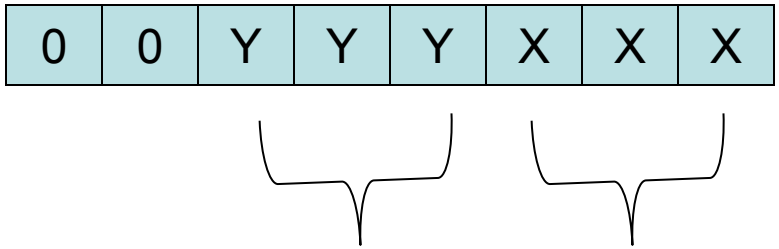


Ucos-ii ready list



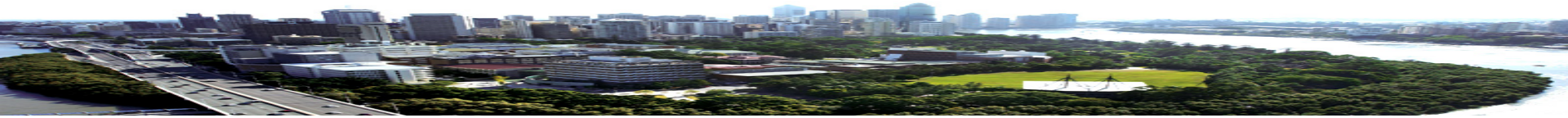


Ucos-ii ready task priority



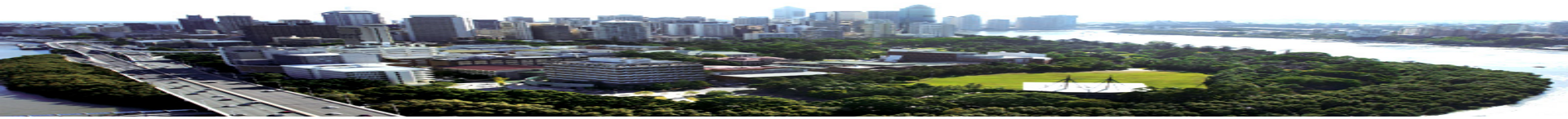
Bit position in OSRdyGrp and index into OSRdyTbl[]

Bit position in the byte read from OSRdyTbl[]



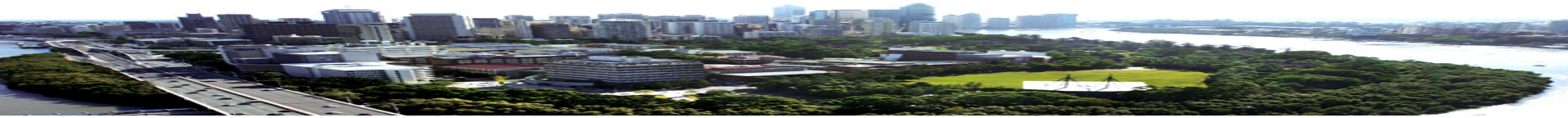
Ucos-ii ready list operations

- Making a task ready to run involves setting a bit in the table to 1 and a bit in OSRdyGrp to 1. The position is determined from the task priority.
- Removing a task involves clearing a bit in the table. OSRdyGrp bit is cleared only when all tasks in the group are not ready.



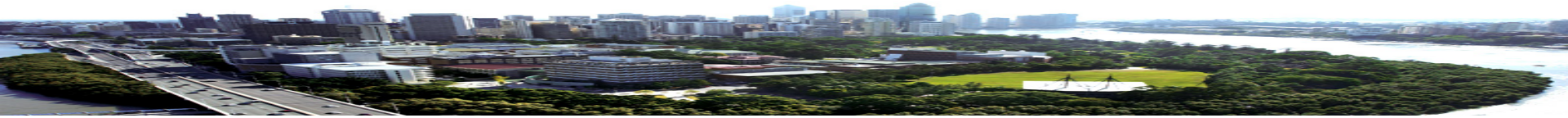
Finding the highest priority task

- Finding the priority of the highest priority task involves two table look up operations and an arithmetic computation on the returned values. Finding the TCB for this requires another table look up.
- **Task scheduling time is constant and independent of the number of tasks**
- Task level scheduling is done by OS_Sched()



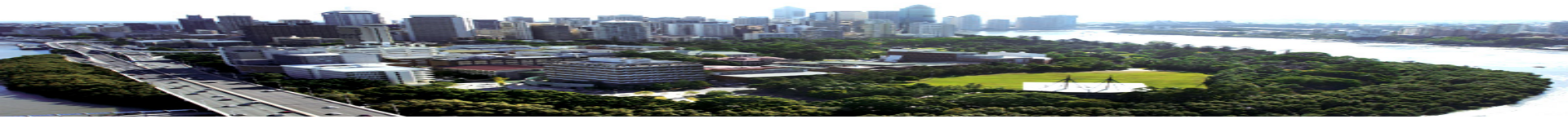
ucos-II support for real time scheduling

- OSTaskCreate() takes an integer argument which assigns a priority to the task created
- OSTimeDly() can be used to delay a task
- OSTimeDlyResume() will resume the task
- OSTimeGet() and OSTimeSet() can be used to get and set system time. This can be used in calculating execution time and dynamic priority. A 32 bit counter is used for clock ticks.



ucos-II support for real time scheduling

- OSTaskChangePrio(INT8U oldprio, INT8U newprio) allows priorities to be changed dynamically
- OSTaskSuspend() and OSTaskResume() can be used to suspend and resume execution
- OSSchedLock() and OSSchedUnlock() can be used to suspend and resume scheduling



ucos-II support for real-time scheduling

- A task can be made to wait for an event. The time required for insertion of the task in the wait list is constant and independent of the number of tasks. (ISRs can call a non-blocking version)
- The event control block can be examined to find the highest priority task waiting on the event
- No built-in real-time scheduler