Prelim 2 Review Guide

Ambar Soni

April 28, 2014

Contents

1 Real Time Systems **2**
1.1 Performance vs Real Time . . . 2
1.2 Jobs and Tasks . . . 2

2 Scheduling **3**
2.1 Scheduling Algorithms . . . 4
2.2 Resource Constraints . . . 4
2.3 First Come Fist Serve (FCFS) . . . 5
2.4 Shortest Job First (SJF) . . . 5
2.4.1 Proof for SJF . . . 6
2.5 Priority Scheduling . . . 8
2.6 Round Robin (RR) . . . 8

3 Aperiodic RealTime Scheduling **9**
3.1 Earliest Due Date (EDD) . . . 9
3.1.1 Proof for EDD . . . 10
3.2 Earliest Deadline First (EDF) . . . 11

4 Periodic RealTime Scheduling **12**
4.1 Timeline Scheduling . . . 12
4.2 Rate Monotonic Scheduling . . . 13

5 Priority Inheritance **13**
5.1 Non-Preemptive Protocol . . . 14
5.2 Highest Locker Priority . . . 14
5.3 Priority Inheritance Protocol . . . 15
5.4 Priority Ceiling Protocol . . . 15

6 Communication Protocols **16**
6.1 RS-232 . . . 16

1 Real Time Systems

A real time systems is a hardware and software system that is subject to a real-time constraint, for example operational deadlines from event to system response. Real-time programs must guarantee response within precise time constraints, often referred to as *deadlines*.

- Correctness depends on the usual properties such as correct output as well as on the *time* at which the output is produced.
- Time between different entities must be synchronized (non-trivial)
- Real time is not the same as performance or speed!

1.1 Performance vs Real Time

Real time means to have to *guarantee* timing properties. In contrast, Performance aims to minimize *average* response time. There are a lot of sources of unpredictability. All sources of uncertainty must be minimized. Even system tasks like interrupt handling must be predictable.

- Architecture: cache, pipelining
- Run-time system: scheduling, other tasks
- Environment: Bursty information flow, extreme conditions
- Input: no explicit notion of time in most languages

1.2 Jobs and Tasks

- A job is a sequence of operations that, in the absence of any other activities, is executed by the processor.
- A task is a sequence of jobs
- Jobs have:

  A request time r_i (arrival time)

  A start time s_i

  A finishing time f_i

  An absolute deadline d_i

There are different types of tasks and each one is important in its own applications:

- Hard Tasks: All jobs must meet their deadlines. Missing deadlines has a catastrophic effect.

  low-level control

  sensor-actuator interactions for critical functions

- Soft Task: Missing deadlines is undesirable, but only causes performance degradation.

  reading input from keyboard

  updating graphics

- Tasks can be assigned *priorities*
- Tasks can be time driven (periodic) or event driven (aperiodic)

A single job:

$f_i - s_i$ is the *worst case execution time* (wcet)

A task:

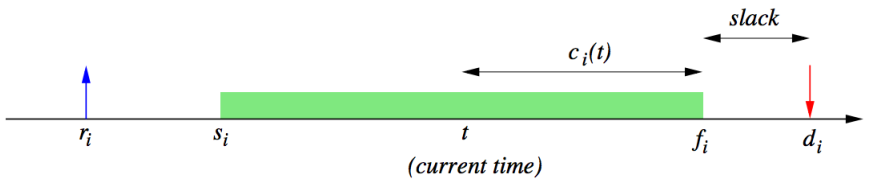
2 Scheduling

A Scheduling algorithm is a strategy used to pick a *ready* task for execution. There are two types:

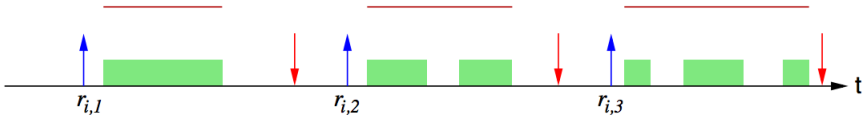
1. Preemptive: The running task can be temporarily suspended to execute another task
2. Non preemptive: The running task cannot be suspended until completion or until it is blocked.

A *schedule* is a particular assignment of tasks to the processor. Here is some important derived terminology with regard to scheduling:

- Lateness: $L_i = f_i - d_i$ Want this to be ≤ 0
- Tardiness: $max(0, L_i)$
- Residual wcet: $c_i(t)$
- Slack: $d_i - t - c_i(t)$ How sloppy can I be
- Jitter: time variation of a periodic event



Example: completion-time jitter



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

2.1 Scheduling Algorithms

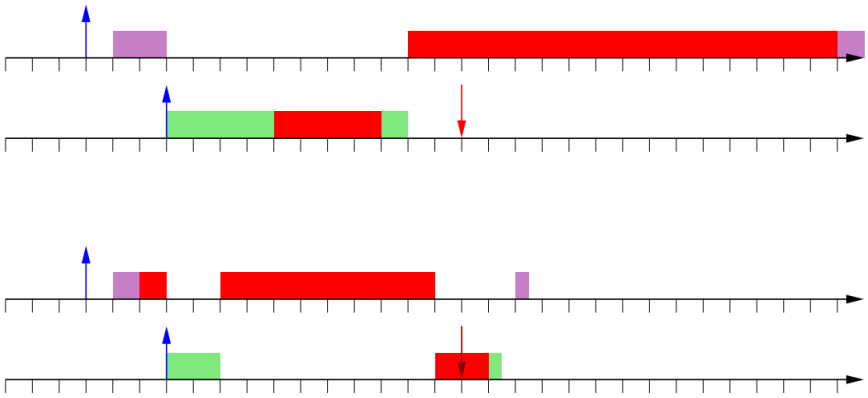
A schedule is *feasible* if all tasks are able to complete with their set of constraints. A set of tasks K is set to be *schedulable* if a feasible schedule exists. Scheduling algorithms can be:

- Preemptive or non-preemptive
- Static or Dynamic: are the scheduling decisions based on parameters that change with time? For static, the parameters for all processes needed for the schedule are known before hand.
- Online or Offline: are the decisions made prior with knowledge of task activations, or are they taken at run time based on the set of active tasks? These are different from Static/Dynamic in the sense that they describe when you are going to run the algorithm for scheduling.
- Optimal or heuristic: can you prove that the algorithm optimizes a certain criteria or not?

Optimality simply means are we trying to minimize or maximize some parameter. For example, minimizing the maximum lateness, minimizing the number of missed deadlines, or maximize the value of of feasible tasks given than tasks are assigned some utility value.

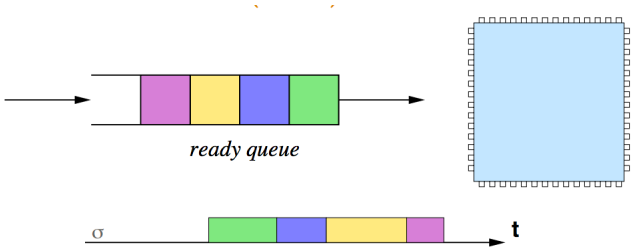
2.2 Resource Constraints

Resources (locks) can often be limited or even unavailable. Shared resources often require mutual exclusion. Both of these concepts lead to delays. Therefore simply have a *faster* processor doesn't always mean it is easier to meed deadlines:

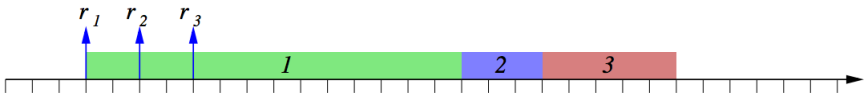


2.3 First Come Fist Serve (FCFS)

A simply policy in which processes are given access to the CPU in the order of their arrival time. However, it is very unpredictable as the response time depends strongly on task arrivals. It is not suitable for real time systems because it is not concerned with feasibility. It has the following properties:



- Non-preemptive
- Dynamic
- Online
- Heuristic

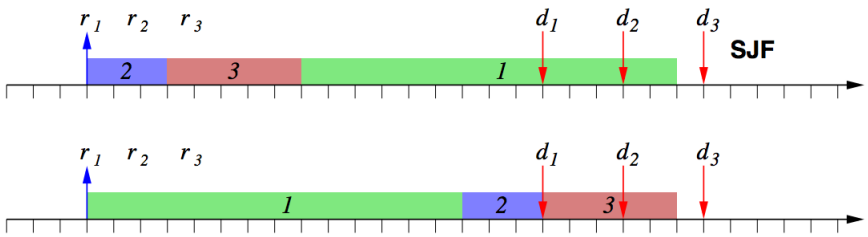


2.4 Shortest Job First (SJF)

This policy picks the next task with the shortest computation time. The goal of this algorithm is to minimize the *average response time*. It also has the following properties:

- Non preemptive or preemptive

- Static (c_i is known and fixed)
- Online or offline
- Optimal: It minimizes the average response time



Not suitable for real-time in the sense of feasibility!

2.4.1 Proof for SJF

Formal Proof In this section, we'll prove that the SJF algorithm produces a schedule that minimizes average response time. First recall a few definitions and points to note.

- A schedule is a function from jobs to natural numbers, $\sigma : \Gamma \to \mathbb{R}$. Abstractly, for all time t, $\sigma(t) = \gamma$ implies that the task γ is scheduled to run at time t. Often it is convenient to assume that there is no idle time in the schedule and to let σ denote a sequenced collection of the set of tasks, Γ. For example, $\sigma = \gamma_1, \gamma_2, \ldots, \gamma_n$.
- The response time of a task γ_i is $f_i - r_i$. The average response time of a schedule is $\frac{\sum_i f_i - r_i}{|\Gamma|}$. For a given set of tasks, Γ, $|\Gamma|$ is a constant. Thus, if we aim to minimize average response time, we can equivalently minimize the sum of response times, $\sum f_i - r_i$.
- An inversion in a schedule σ is a pair of tasks (γ_j, γ_i) such that γ_j is scheduled before γ_i and γ_i is shorter than γ_j.

Lemma 1. *There exists an optimal schedule with no idle time.*

Proof. Assume there did not exist an optimal schedule with no idle time. Take some optimal schedule σ_O. We can remove the idle time and improve the average response time. However, σ_O was optimal. This is a contradiction. ■

Lemma 2. *All schedules with no inversions and no idle time are equally optimal.*

Proof. Two schedules with no inversions and no idle time can vary only in the ordering of tasks with equal length. If we swap two such tasks, the average response time does not change. Thus, if we begin with some inversion-free, idle-free schedule σ_1 and perform a sequence of swaps, we will get some other schedule σ_2 that is equally optimal. ■

Lemma 3. *There exists an optimal schedule with no inversions and no idle time.*

Proof. We will prove this claim with an exchange argument. We'll begin with some optimal schedule σ_O and perform a series of operations to remove inversions. Each operation will not decrease the optimality of the algorithm. This will lead to a schedule that has no idle time and no inversions that is still optimal.

Consider some optimal schedule σ_O with an inversion γ_j, γ_i and no idle time. Partition the schedule into three segments.

- σ_{O1}: The set of tasks scheduled before γ_j
- σ_{O2}: The set of tasks between γ_j and γ_i, inclusive
- σ_{O3}: The set of tasks scheduled after γ_i

We will now exchange γ_i and γ_j and claim that the optimality of σ_O does not decrease. First note that the sum of response times of σ_{O1} and σ_{O3} are not altered. σ_{O2} can be divided into γ_i, γ_j, and the other tasks scheduled between them. The response time of the intermediate tasks strictly decrease. The response time of γ_j is equal to the old response time of γ_i. The response time of γ_i is strictly smaller than the old response time of γ_j. Thus, the sum of response time of σ_{O2} does not increase.

Any schedule contains a finite number of inversions. Each time we perform a swap, we decrease the number of inversions. Thus, we are guaranteed to make a finite number of swaps.

After we complete the finite number swaps, we have constructed a schedule with no inversions and no idle time that is not less optimal than our original optimal schedule σ_O. ■

Claim 1. *SJF produces optimal schedules.*

Proof. We can trivially combine Lemma 1, Lemma 2, and Lemma 3. There exists an optimal schedule that is SJF. All SJF schedules are equally optimal. Thus, all SJF schedules are optimal. ■

Pragmatic Proof The following proof is not rigorous but will suffice for the purposes of this class.

To prove SJF we will use a simple interchange argument. We start with a schedule σ where $\sigma = j_1, j_2 \dots j_k \dots j_n$. In this schedule σ we have two jobs j_i and j_k such that $c_i \leq c_k$. Now lets assume that the schedule σ is such that $\sigma = j_1, j_2 \dots j_{i-1}, j_k, j_{i+1} \dots j_{k-1}, j_l, j_{k+1} \dots j_n$. We must show that if we interchange the positions of j_i and j_k to produce a schedule σ' that the new schedule σ' has at least as good an average response time as σ. If we look at both schedule σ and σ we first realize that the average response time between $j_1 \dots j_{i-1}$ and $j_{k+1} \dots j_n$ in both schedules is equal. For the jobs $j_i \dots j_k$ in σ' and $j_k, j_{i+1} \dots j_{k-1}, j_i$ we every job in σ' will have a lower resposne time since $c_i \leq c_k$ and every job after the first one in σ' (for this subset of jobs) will finish earlier than the corresponding job in σ. Thus, we can conclude that the average resposne time for this subset of jobs is less than or equal in σ' than in σ. Consequently, the average response time in σ' is less than equal to that of σ. If we make a finite number of such transpositions, we can see that σ_{SJF} will minimize average response time.

2.5 Priority Scheduling

Each task is assigned a priority p_i and the task with the highest priority is selected first. Tasks with the same priority are scheduling using FCFS. It has the following properties:

- Preemptive
- Static or Dynamic based on how you define priority
- Online
- Optimal

A common issue that pops up is *Starvation:* low priority tasks may experience long delays due to preemption by higher priority tasks. This is commonly resolved by *Aging:* the priority of tasks increases with waiting time.

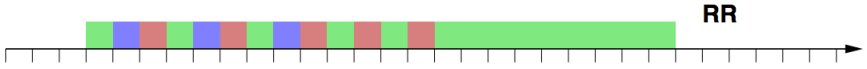
A few things to note with regards to defining priority:

- If $p_i = 1/c_i$: shortest job first!
- If $p_i = const$: first come first served!

2.6 Round Robin (RR)

There is a ready queue which follows FCFS. However, each task cannot execute more than Q time units. When Q time units have elapsed, the task is put back into the ready queue. It has to following properties:

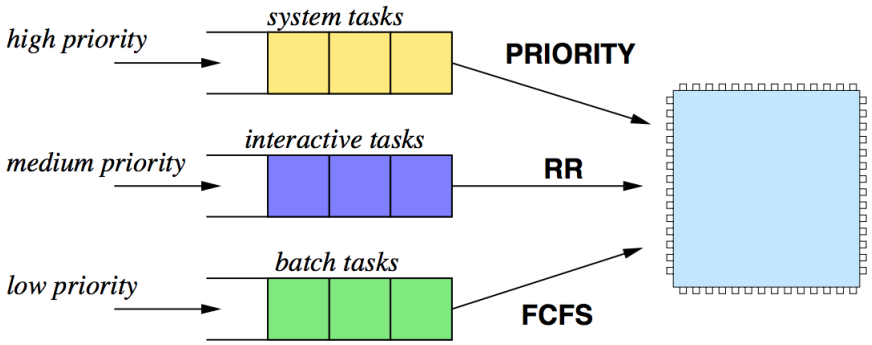
- Preemptive
- Dynamic
- Online
- Heuristic
- For very small Q: Each task runs as if it were executing on a virtual processor that is n(number processes) times slower than real one
- For very large Q: RR = FCFS when $Q \geq C_i$



3 Aperiodic RealTime Scheduling

There are three main types of tasks we are concerned about:

1. *System Tasks:* Each task has a priority and is usually preemptive. This are the most important tasks and usually have a dedicated system priority queue.
2. *Interactive Tasks:* These are tasks such as commands run on the command line. We use RR to maintain fairness and the response time is proportional to the load (medium priority).
3. *Batch Task:* These tasks are run when there is free time to run them later and the output is redirected to some file to be read later. These are lowest priority. $P_i = 1/r_i$

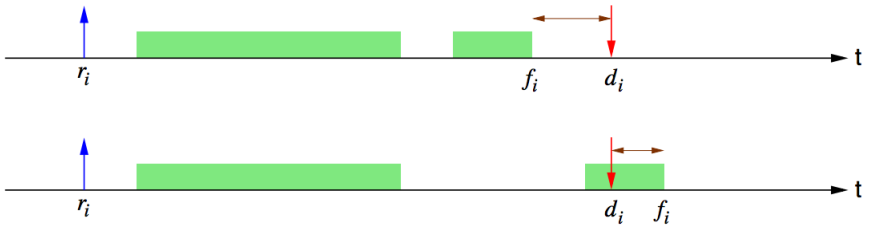


3.1 Earliest Due Date (EDD)

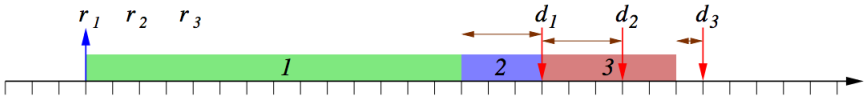
The algorithm selects the task with the earliest *relative* deadline. All the tasks arrive simultaneously and have a fixed known priority D_i. Preemption is not an issue and the goal is to *minimize the maximum lateness* L_{max}. EDD guarantees a feasible (every task will meet its deadline) schedule as long as feasible schedule exists. The priority of a task will be $1/D_i$

- Non-preemptive
- Static
- Offline
- Optimal - minimize max lateness

$$L_i = f_i - d_i$$



$$L_{max} = \max_i(L_i)$$



- $L_{max} < 0 \Rightarrow$ no task misses its deadline

EDD can be proved using *Jackson's Rule:* Given a set of n independent tasks, any algorithm that executes the tasks in order of increasing deadlines is optimal with respect to maximum lateness.

3.1.1 Proof for EDD

Formal Proof TODO

Pragmatic Proof The following proof is not a completely rigorous attempt at proving EDD but will suffice for the purposes of this class. First we realize that we are essentially trying to prove **Jackson's Rule**.

Theorem (Jackson's Rule) : *Given a set of n independent tasks, any algorithm that executes the tasks in the order of non-decreasing deadlines is optimal with respect to maximum lateness*

Proof To prove Jackson's theorem we will use an interchange argument. Let σ be a schedule produced by any algorithm A. If A is not the same as EDD then, there are two jobs J_a and J_b such that $d_a \leq d_b$ where J_b immediately precdes J_a in the schedule σ. Lets switch J_b and J_a to produce a new schedule σ'. In σ', J_a immediately precedes J_b. These two schedules are shown in the Figure 1.

From Figure 1 we can see that for σ the maximum lateness is $f_a - d_a$. Lets compare this with the maximum lateness of σ'. Here we have two cases :

1. $L'_a \geq L'_b$ Then we know that the maximum lateness between J_a and J_b in σ' is less

than or equal to $f_a - d_a$ and thus the maximum lateness of σ' is less than or equal to that of σ.

2. $L'_b \geq L'_a$ Then we know that $f'_b = f_a$ and thus since $d_b \geq d_a$, $L'b \leq L_a$. That is, $f'_b - d_b \leq f_a - d_a$ and thus the maximum lateness of σ' is less than or equal to the maximum latness of σ.

From both cases we can see that by interchanging J_b and J_a the maximum lateness is at least as good as it was before the interchanging. Thus we can conclude that interchanging J_b and J_a does not increase maximum lateness and a finite number such such transpositions can transpose σ into an EDD schedule which in turn will have minimum maximum latness.

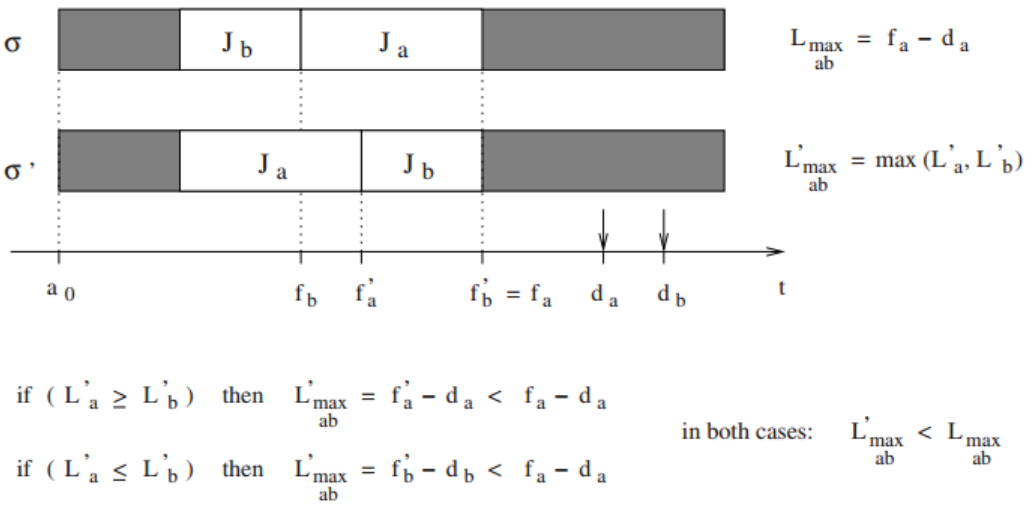
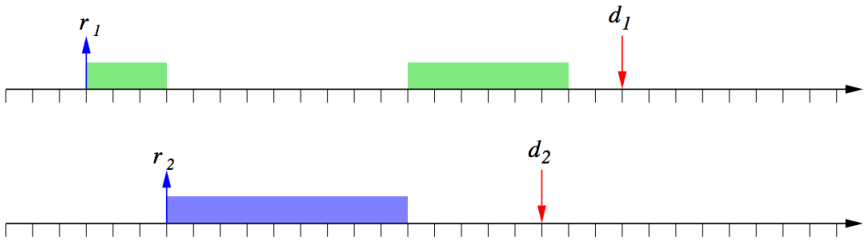


Figure 1: Jackson's Rule

3.2 Earliest Deadline First (EDF)

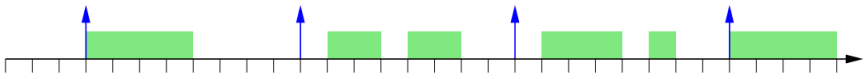
The algorithm selects the task with the earliest *absolute* deadline. Tasks can arrive anytime and have a dynamic priority d_i depending on when the tasks arrive. The tasks are preemptive and the goal is to again *minimize the maximum lateness* L_{max}. Under non-preemptive scheduling, EDF, is not optimal and cannot produce a feasible schedule.

- Preemptive
- Dynamic
- Online
- Optimal - minimize max lateness



4 Periodic RealTime Scheduling

Period tasks are those that have jobs that repeat at a regular interval in time.

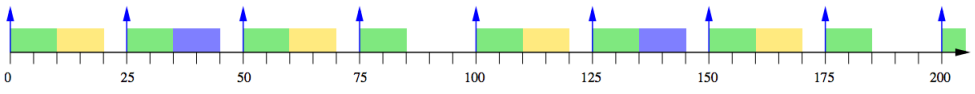


4.1 Timeline Scheduling

A classic technique in which the time axis is divided into intervals of equal length (slots). Each task is *statically* allocated in order to meet desired request rate. Timers are used to activate execution in each slot.

- Minor Cycle: GCD of periods - it tells you when you need to make decisions on what to run or context switching.
- Major Cycle: LCM of periods - it tells you when the whole cycle repeats and how big a program would be.

Task	Period	WCET
A	25ms	10ms
B	50ms	10ms
C	100ms	10ms



- Δ = GCD (minor cycle) = 25ms
- T = LCM (major cycle) = 100ms

Advantages:

- Simple implementation
- Low run-time overhead
- Jitter can be controlled

Disadvantages:

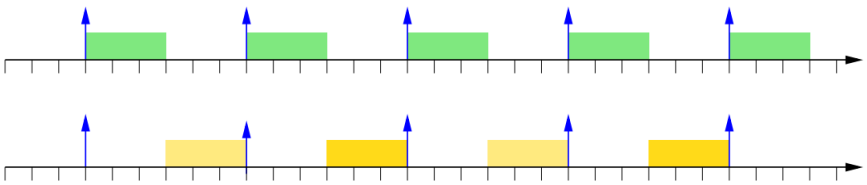
- Not robust to overloads/overruns
- Difficult to expand the schedule
- Not easy to handle aperiodic activities

If the schedule becomes overloaded, a task might not meet its deadline and there can be a domino effect. If the task is aborted, the system could be in an inconsistent state (undefined). If a certain task is updated and now takes more time, you may have to re-do the entire schedule. A common approach to this problem is to split the task into two sub tasks and then re-build the schedule.

4.2 Rate Monotonic Scheduling

Each task in assigned a fixed priority proportional to its rate. The big question here is how can we determine feasibility?

- Each task uses the processor for a fraction of time: $U_i = C_i/T_i$
- The total processor utilization is: $U_{cpu} = \sum_i U_i$
- U_{cpu} measures the processor load. If $U_{cpu} > 1$, the processor is overloaded so the task set cannot be scheduled.
- Liu: For n periodic tasks if $U_{cpu} \leq n(2^{1/n} - 1)$, then RM will produce a feasible schedule. In the limit $n \to \infty$, RHS is $ln(2)$



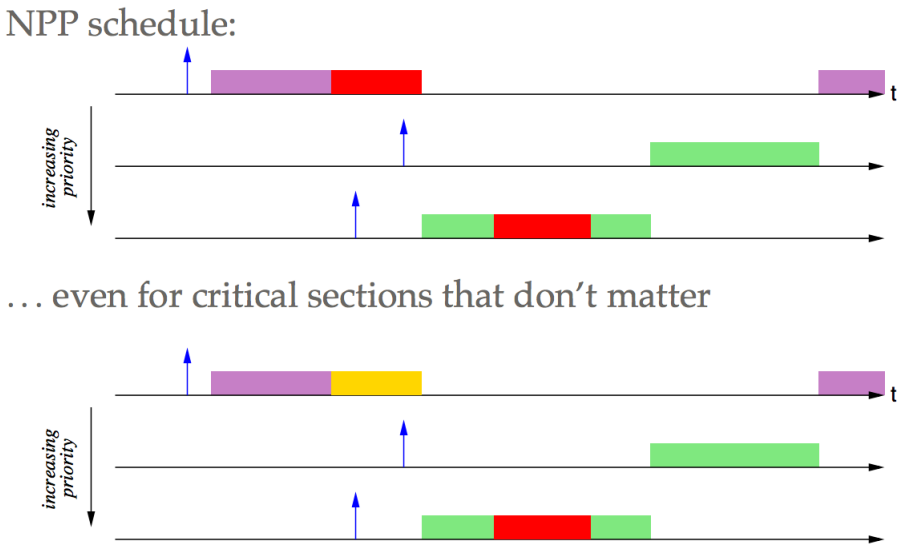
Here $U_{cpu} = 1$ and the schedule is feasible.

5 Priority Inheritance

Priority Inversion is a problem in which a high priority task is indirectly preempted by a medium priority task effectively 'inverting' the relative priorities of the two tasks for an unbounded interval of time. To mitigate, this problem, there are four protocols presented below.

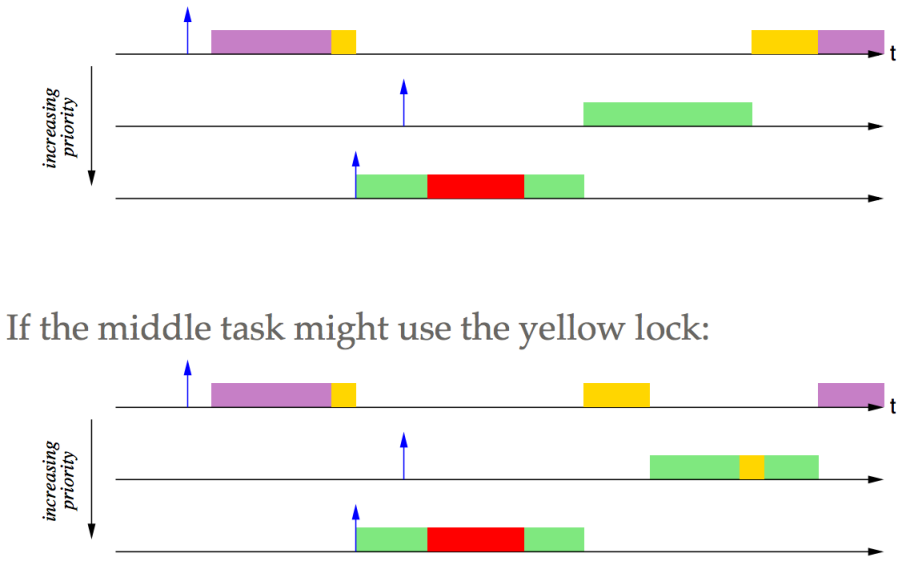
5.1 Non-Preemptive Protocol

Preemption is forbidden in Critical Sections. To implement this: when a task enters a critical section, increase its priority to max of all the other processes priorities. The problem with this design is that high priority tasks that do not interfere with the CS will be blocked.



5.2 Highest Locker Priority

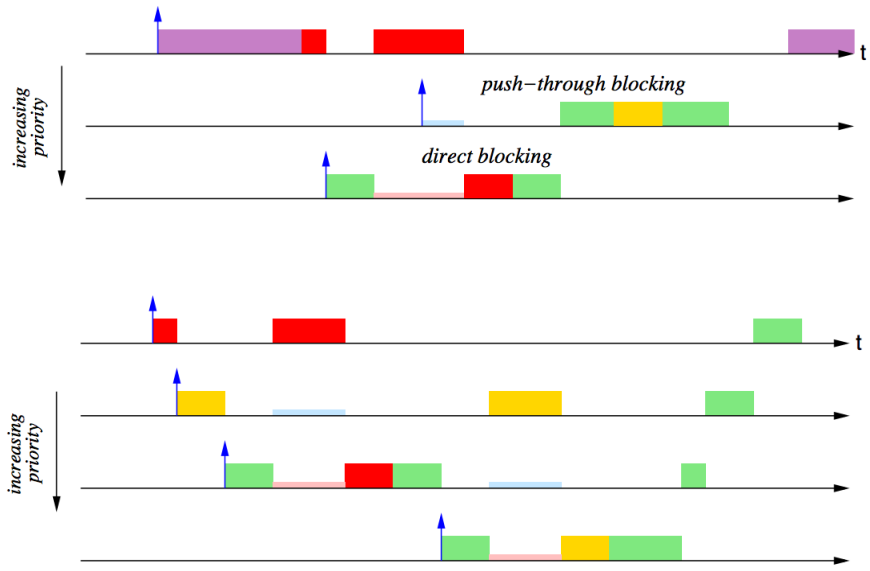
A task in the critical section gets the highest priority among the tasks that use the CS. To implement: when a task enters a CS, increase its priority to the max value of the tasks that may access the same critical section. A process could be blocked because it *might* enter the critical section, not because it is in the critical section.



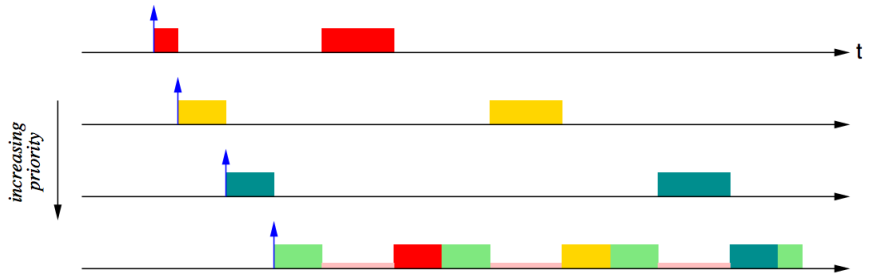
5.3 Priority Inheritance Protocol

A task in a critical section increases its priority only if it blocks other tasks. A task in a critical section inherits the highest priority among those tasks that it blocks. There are two types of blocking:

1. Direct: task blocked on a lock
2. Push-through: task blocked because a lower priority task inherited a higher priority (indirectly)



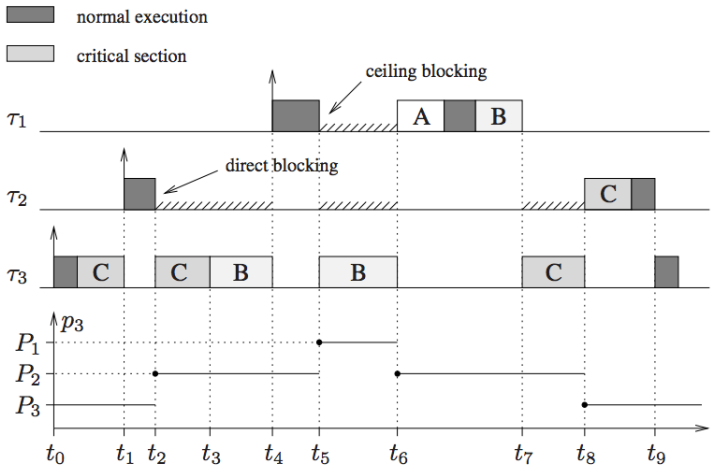
Problem: *Chained Blocking:* higher priority processes get constantly blocked while waiting for lower priority processes to finish CS.



5.4 Priority Ceiling Protocol

This algorithm aims to reduce chained blocking and is a modification to the PIP protocol. Each semaphore is assigned a priority ceiling equal to the highest priority of the tasks that can lock it. Then, a task T_i is allowed to enter a critical section only if its priority is higher than all priority ceilings of the semaphores currently locked by tasks. If you are inside a CS

and end up blocking a task, you inherit the priority of the task that you blocked until you relinquish the CS that gave you that authority. T_i.



6 Communication Protocols

With regard to what we previously talked about with Daisy Chaining, devices can send signals to the CPU. However, we have not yet discussed how the signals are transmitted between the wires. Here is some vocabulary:

- Synchronous: Having one line for data and a separate line for clock to keep track of when a bit is sent or received.
- Parity: Way to check for error, it is the XOR of all the bits

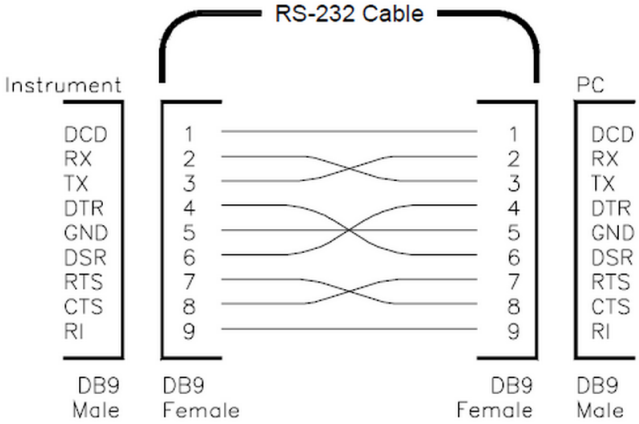
  Even: xor of bits = 0

  Odd: xor of bits = 1

  Error checking is very weak bc can only check for 1 bit of error
- Baud Rate: frequency of you are transmitting/receiving at
- Flow Control: method of tell tranismitter to Stop/Start. It is used if you have error or if using a finite buffering system.

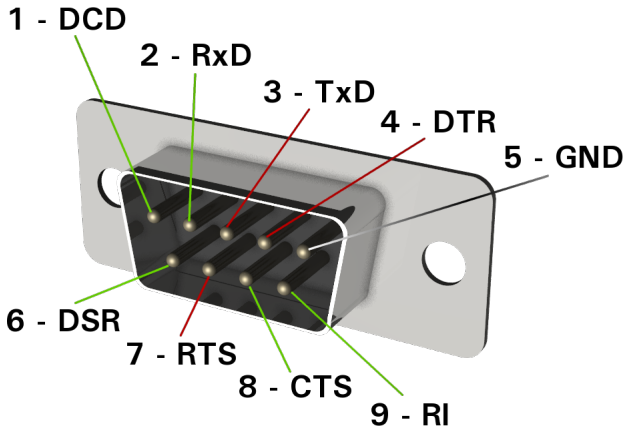
6.1 RS-232

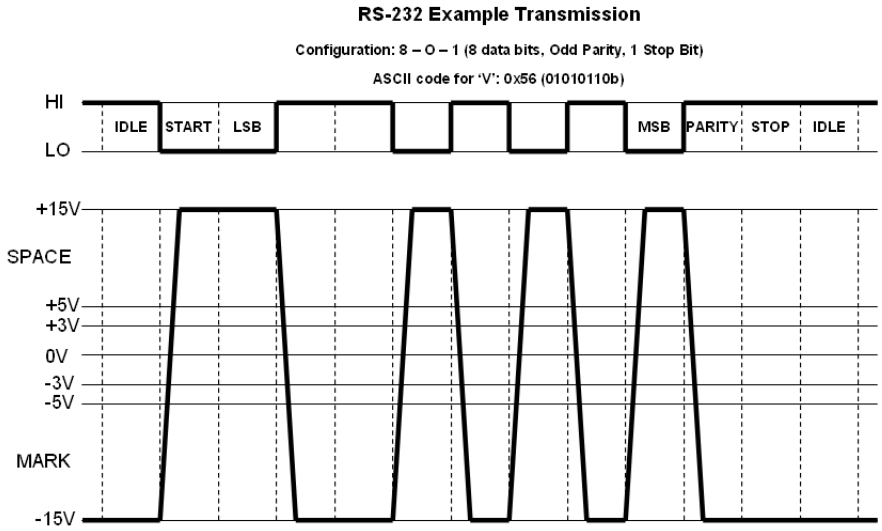
This is a very simple and resilient protocol that was used initially to connect to a modem.



- Large voltage swings to amplify signal to noise ratio (1:-12V 0:+12V)
- Can be half duplex (1 comm direction at a time) or full duplex
- Asynchronous
- Limited to 7-8 bits at a time per transmission for data and 0-1 parity bits per transmission.

  8 bits are are communicating bytes. 7 bits are used for communicating ASCII
- Both transmiter and receiver have to agree on how many bits sent, what type of parity, and baud rate before communicating.
- Advantages: Simple, resilient to noise, can transmit over long distances
- Disadvantages: Slow b/c there is no internal clock, drains power
- Software Flow Control: XON/XOFF (17)/(19)
- Hardware Flow Control: RTS/CTS - assigned bits on the serial port




RS-232 Example Transmission
Configuration: 8 – O – 1 (8 data bits, Odd Parity, 1 Stop Bit)
ASCII code for 'V': 0x56 (01010110b)
HI
LO
IDLE
START
LSB
MSB
PARITY
STOP
IDLE
+15V
SPACE
+5V
+3V
0V
-3V
-5V
MARK
-15V