CSE 380
Computer Operating Systems

Instructor: Insup Lee & Dianna Xu

University of Pennsylvania, Fall 2003
Lecture Note 3: CPU Scheduling

CPU SCHEDULING

- How can OS schedule the allocation of CPU cycles to processes/threads to achieve “good performance”?

- Overview of topics
  - Issues in scheduling
  - Basic scheduling algorithms
    - First-come First-served
    - Round Robin
    - Shortest Job First
    - Priority based
  - Scheduling in Unix
  - Real-time scheduling (Priority Inheritance)

Scheduling Issues

- Application Profile:
  - A program alternates between CPU usage and I/O
  - Relevant question for scheduling: is a program compute-bound (mostly CPU usage) or I/O-bound (mostly I/O wait)
- Multi-level scheduling (e.g., 2-level in Unix)
  - Swapper decides which processes should reside in memory
  - Scheduler decides which ready process gets the CPU next
- When to schedule
  - When a process is created
  - When a process terminates
  - When a process issues a blocking call (I/O, semaphores)
  - On a clock interrupt
  - On I/O interrupt (e.g., disk transfer finished, mouse click)
  - System calls for IPC (e.g., up on semaphore, signal, etc.)

Scheduling Issues

- Is preemption allowed?
  - Nonpreemptive scheduler does not use clock interrupts to stop a process
- What should be optimized?
  - CPU utilization: Fraction of time CPU is in use
  - Throughput: Average number of jobs completed per time unit
  - Turnaround Time: Average time between job submission and completion
  - Waiting Time: Average amount of time a process is ready but waiting
  - Response Time: in interactive systems, time until the system responds to a command
  - Response Ratio: (Turnaround Time)/(Execution Time) -- long jobs should wait longer

Scheduling Issues

- Different applications require different optimization criteria
  - Batch systems (throughput, turnaround time)
  - Interactive system (response time, fairness, user expectation)
  - Real-time systems (meeting deadlines)
- Overhead of scheduling
  - Context switching is expensive (minimize context switches)
  - Data structures and book-keeping used by scheduler
- What's being scheduled?
  - Processes in Unix, but Threads in Linux or Solaris

Basic Scheduling Algorithm: FCFS

- FCFS - First-Come, First-Served
  - Non-preemptive
  - Ready queue is a FIFO queue
  - Jobs arriving are placed at the end of queue
  - Dispatcher selects first job in queue and this job runs to completion of CPU burst
- Advantages: simple, low overhead
- Disadvantages: inappropriate for interactive systems, large fluctuations in average turnaround time are possible.

Example of FCFS

- Workload (Batch system)

  Job 1: 24 units, Job 2: 3 units, Job 3: 3 units

- FCFS schedule:

```
  | Job 1 |   Job 2   |   Job 3   |
 0        24          27          30
```

- Total waiting time: 0 + 24 + 27 = 51
- Average waiting time: 51/3 = 17
- Total turnaround time: 24 + 27 + 30 = 81
- Average turnaround time: 81/3 = 27

SJF - Shortest Job First

- Non-preemptive
- Ready queue treated as a priority queue based on smallest CPU-time requirement
  - arriving jobs inserted at proper position in queue
  - dispatcher selects shortest job (1st in queue) and runs to completion

- Advantages: provably optimal w.r.t. average turnaround time
- Disadvantages: in general, cannot be implemented. Also, starvation possible !
- Can do it approximately: use exponential averaging to predict length of next CPU burst
  ==> pick shortest predicted burst next!

Example of SJF

- Workload (Batch system)

  Job 1: 24 units, Job 2: 3 units, Job 3: 3 units

- SJF schedule:

```
| Job 2 |   Job 3   |   Job 1   |
0       3           6           30
```

- Total waiting time: 6 + 0 + 3 = 9
- Average waiting time: 3
- Total turnaround time: 30 + 3 + 6 = 39
- Average turnaround time: 39/3 = 13
- SJF always gives minimum waiting time and turnaround time

Exponential Averaging

$$\tau_{n+1} = \alpha \; t_n + (1 - \alpha) \; \tau_n$$

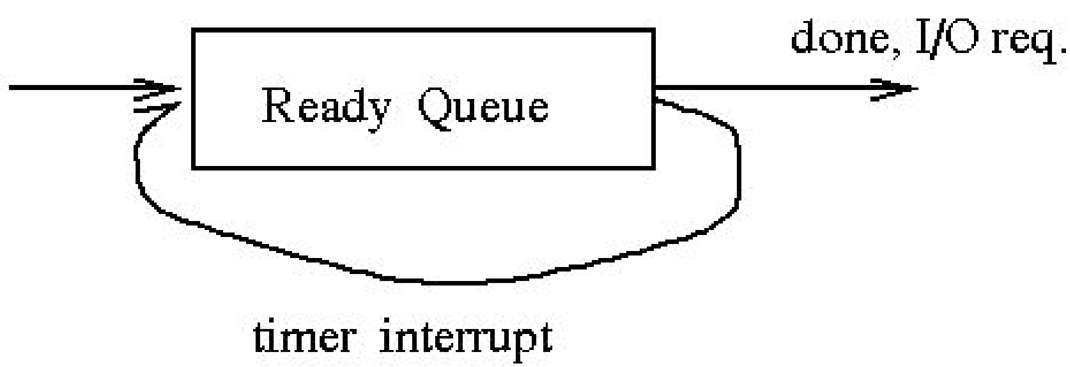
- τ_{n+1} : predicted length of next CPU burst
- t_n : actual length of last CPU burst
- τ_n : previous prediction

- α **= 0** implies make no use of recent history

  ($\tau_{n+1} = \tau_n$)
- α **= 1** implies $\tau_{n+1} = t_n$ (past prediction not used).
- α **= 1/2** implies weighted (older bursts get less and less weight).

RR - Round Robin

- Preemptive version of FCFS
- Treat ready queue as circular
  - arriving jobs are placed at end
  - dispatcher selects first job in queue and runs until completion of CPU burst, or until time quantum expires
  - if quantum expires, job is again placed at end



Example of RR

- Workload (Batch system)

  Job 1: 24 units, Job 2: 3 units, Job 3: 3 units

- RR schedule with time quantum=3:

```
| Job 1 |  Job 2  |  Job 3  |    Job 1     |
0       3         6         9              30
```

- Total waiting time: 6 + 3 + 6 = 15
- Average waiting time: 5
- Total turnaround time: 30 + 6 + 9 = 45
- Average turnaround time: 15
- RR gives intermediate wait and turnaround time (compared to SJF and FCFS)

Properties of RR

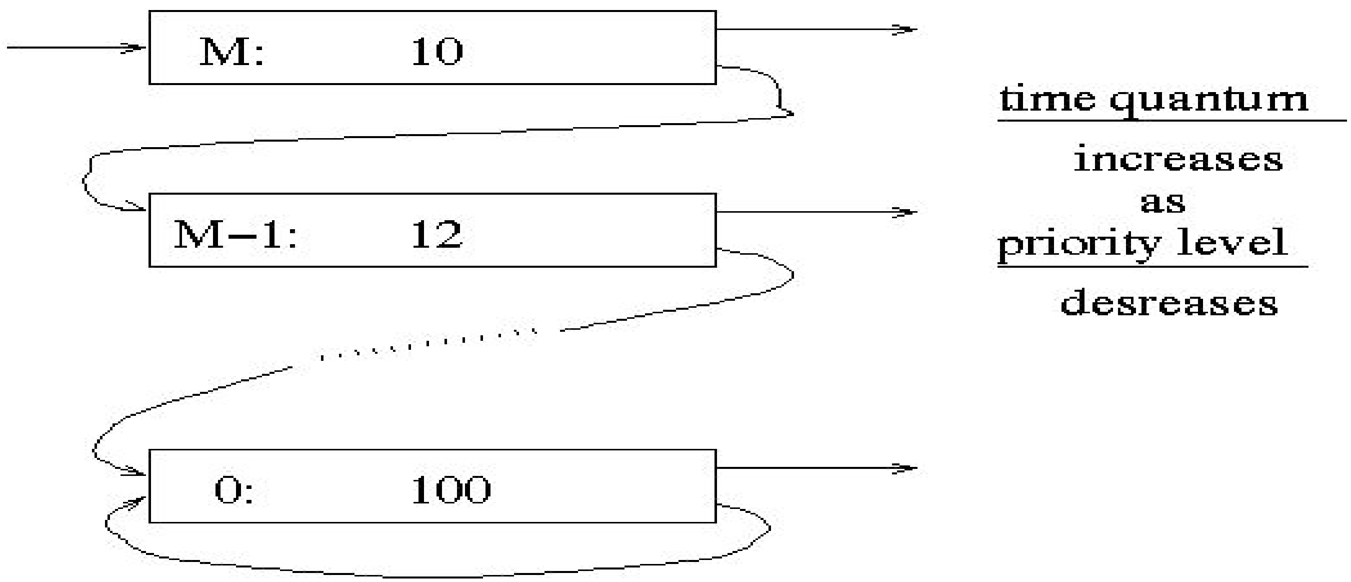
- Advantages: simple, low overhead, works for interactive systems
- Disadvantages: if quantum is too small, too much time wasted in context switching; if too large (i.e. longer than mean CPU burst), approaches FCFS.
- Typical value: 20 – 40 msec
- **Rule of thumb:** Choose quantum so that large majority (80 – 90%) of jobs finish CPU burst in one quantum
- RR makes the assumption that all processes are equally important

HPF - Highest Priority First

- General class of algorithms ==> priority scheduling
- Each job assigned a priority which may change dynamically
- May be preemptive or non-preemptive

- Key Design Issue: how to compute priorities?

Multi-Level Feedback (FB)

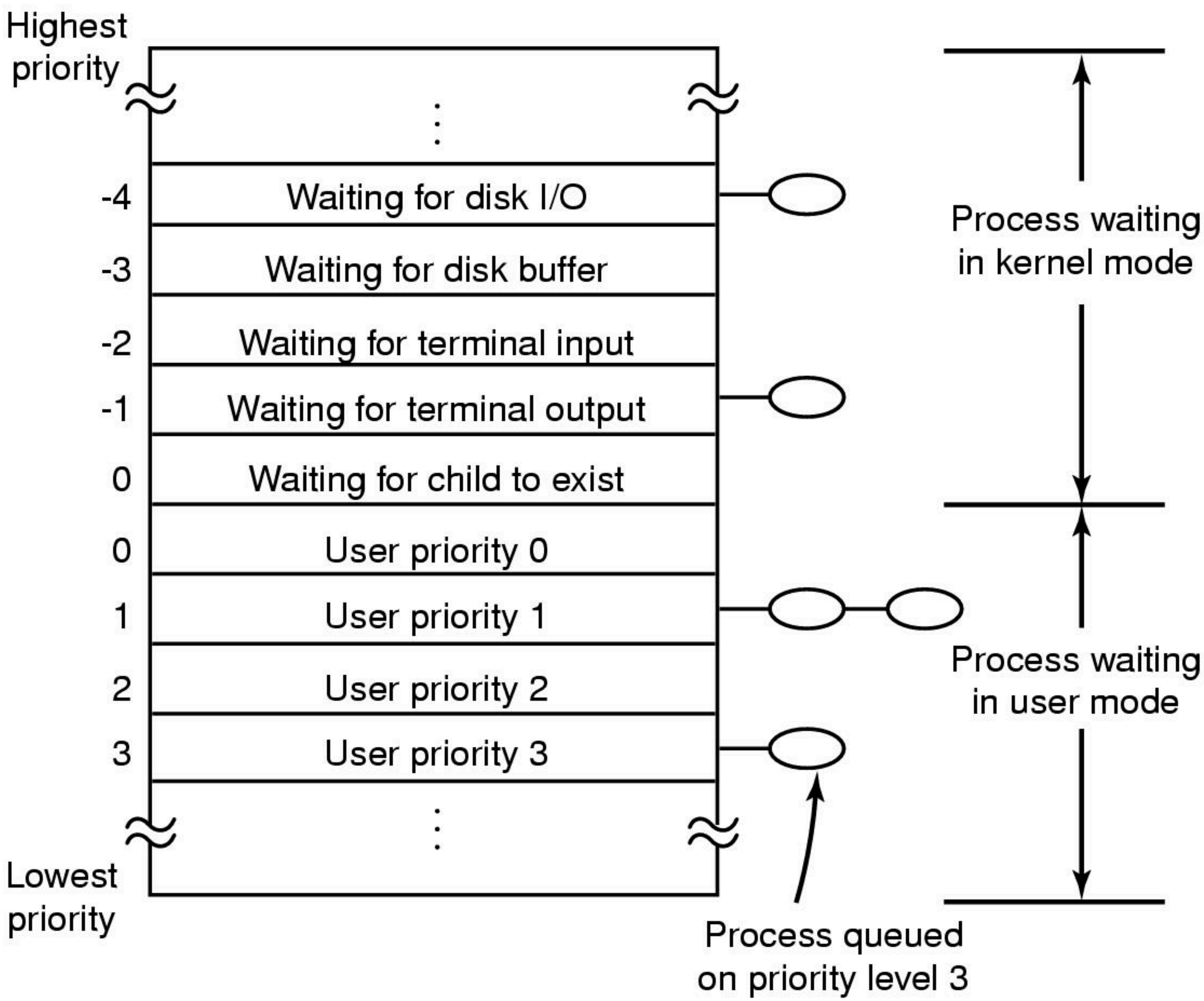


- Each priority level has a ready queue, and a time quantum
- process enters highest priority queue initially, and (next) lower queue with each timer interrupt (penalized for long CPU usage)
- bottom queue is standard Round Robin
- process in a given queue are not scheduled until all higher queues are empty

FB Discussion

- I/O-bound processes tend to congregate in higher-level queues. (Why?)
- This implies greater device utilization
- CPU-bound processes will sink deeper(lower) into the queues.
- large quantum occasionally versus small quanta often
- Quantum in top queue should be large enough to satisfy majority of I/O-bound processes
- can assign a process a lower priority by starting it at a lower-level queue
- can raise priority by moving process to a higher queue, thus can use in conjunction with aging
- to adjust priority of a process changing from CPU-bound to I/O-bound, can move process to a higher queue each time it voluntarily relinquishes CPU.

UNIX Scheduler



Process Scheduling in Unix

- Based on multi-level feedback queues
- Priorities range from -64 to 63 (lower number means higher priority)
- Negative numbers reserved for processes waiting in kernel mode (that is, just woken up by interrupt handlers) (why do they have a higher priority?)
- Time quantum = 1/10 sec (empirically found to be the longest quantum that could be used without loss of the desired response for interactive jobs such as editors)
  - short time quantum means better interactive response
  - long time quantum means higher overall system throughput since less context switch overhead and less processor cache flush.
- Priority dynamically adjusted to reflect
  - resource requirement (e.g., blocked awaiting an event)
  - resource consumption (e.g., CPU time)

Unix CPU Scheduler

- Two values in the PCB
  - p_cpu: an estimate of the recent CPU use
  - p_nice: a user/OS settable weighting factor (-20..20) for flexibility; default = 0; negative increases priority; positive decreases priority
- A process' priority calculated periodically

  priority = base + p_cpu + p_nice

  and the process is moved to appropriate ready queue
- CPU utilization, p_cpu, is incremented each time the system clock ticks and the process is found to be executing.
- p_cpu is adjusted once every second (time decay)
  - Possible adjustment: divide by 2 (that is, shift right)
  - Motivation: Recent usage penalizes more than past usage
  - Precise details differ in different versions (e.g. 4.3 BSD uses current load (number of ready processes) also in the adjustment formula)

Example

- Suppose p_nice is 0, clock ticks every 10msec, time quantum is 100msec, and p_cpu adjustment every sec
- Suppose initial base value is 4. Initially, p_cpu is 0
- Initial priority is 4.
- Suppose scheduler selects this process at some point, and it uses all of its quantum without blocking. Then, p_cpu will be 10, priority recalculated to 10, as new base is 0.
- At the end of a second, p_cpu, as well as priority, becomes 5 (more likely to scheduled)
- Suppose again scheduler picks this process, and it blocks (say, for disk read) after 30 msec. p_cpu is 8
- Process is now in waiting queue for disk transfer
- At the end of next second, p_cpu is updated to 4
- When disk transfer is complete, disk interrupt handler computes priority using a negative base value, say, -10. New priority is -6
- Process again gets scheduled, and runs for its entire time quantum. p_cpu will be updated to 14

Summary of Unix Scheduler

- Commonly used implementation with multiple priority queues
- Priority computed using 3 factors
  - PUSER used as a base (changed dynamically)
  - CPU utilization (time decayed)
  - Value specified at process creation (nice)
- Processes with short CPU bursts are favored
- Processes just woken up from blocked states are favored even more
- Weighted averaging of CPU utilization
- Details vary in different versions of Unix

Real-time Systems

- On-line transaction systems
- Real-time monitoring systems
- Signal processing systems
  - multimedia
- Embedded control systems:
  - automotives
  - Robots
  - Aircrafts
  - Medical devices …

Desired characteristics of RTOS

- **Predictability**, not speed, fairness, etc.
  - Under normal load, all deterministic (hard deadline) tasks meet their timing constraints – avoid loss of data
  - Under overload conditions, failures in meeting timing constraints occur in a predictable manner – avoid rapid quality deterioration.
  - Interrupt handling and context switching should take bounded times
- **Application- directed resource management**
  - Scheduling mechanisms allow different policies
  - Resolution of resource contention can be under explicit direction of the application.

Periodic Tasks

- Typical real-time application has many tasks that need to be executed periodically
  - Reading sensor data
  - Computation
  - Sending data to actuators
  - Communication
- Standard formulation: Given a set of tasks T_1, … T_n. Each task T_i has period P_i and computation time C_i
- Schedulability problem: Can all the tasks be scheduled so that every task T_i gets the CPU for C_i units in every interval of length P_i

Periodic Tasks

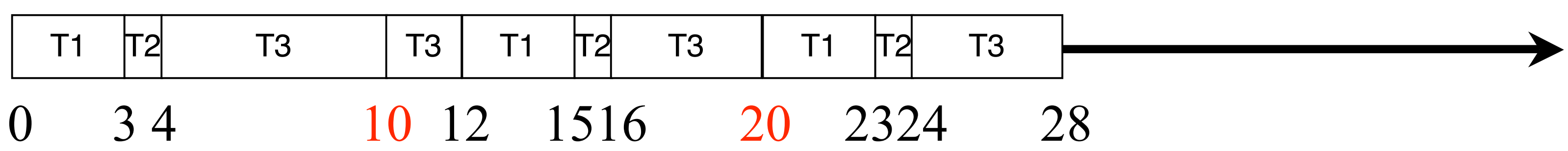
- Example:
  - Task T1 with period 10 and CPU time 3
  - Task T2 with period 10 and CPU time 1
  - Task T3 with period 15 and CPU time 8
- Possible schedule: repeats every 30 sec
  - T1 from 0 to 3, 12 to 15, 24 to 27
  - T2 from 3 to 4, 15 to 16, 27 to 28
  - T3 from 4 to 12, 16 to 24
  - If T2 has period 5 (instead of 10) then there is no schedule
- Simple test:
  - Task T_i needs to use CPU for C_i/P_i fraction per unit
  - Utilization = Sum of C_i/P_i
  - Task set is schedulable if and only if utilization is 1 or less.

Scheduling Algorithm: EDF

- Earliest Deadline First (EDF)
- Based on dynamic priorities.
- For a task T with period P, the i-th instance of T is active during the time interval (i-1)*P to i*P
- So the deadline for task T during the interval (i-1)*P to i*P is i*P (it must finish before the next instance of the same task arrives)
- EDF scheme: Choose the task with the earliest (current) deadline
- Preemptive: scheduling decision made when a task finishes as well as when a new task arrives
- Theorem: If there is a possible schedule, then EDF will find one
- Example: Let's see what happens on the last example

EDF: example

- Task T1 with period 10 and CPU time 3
- Task T2 with period 10 and CPU time 1
- Task T3 with period 15 and CPU time 8

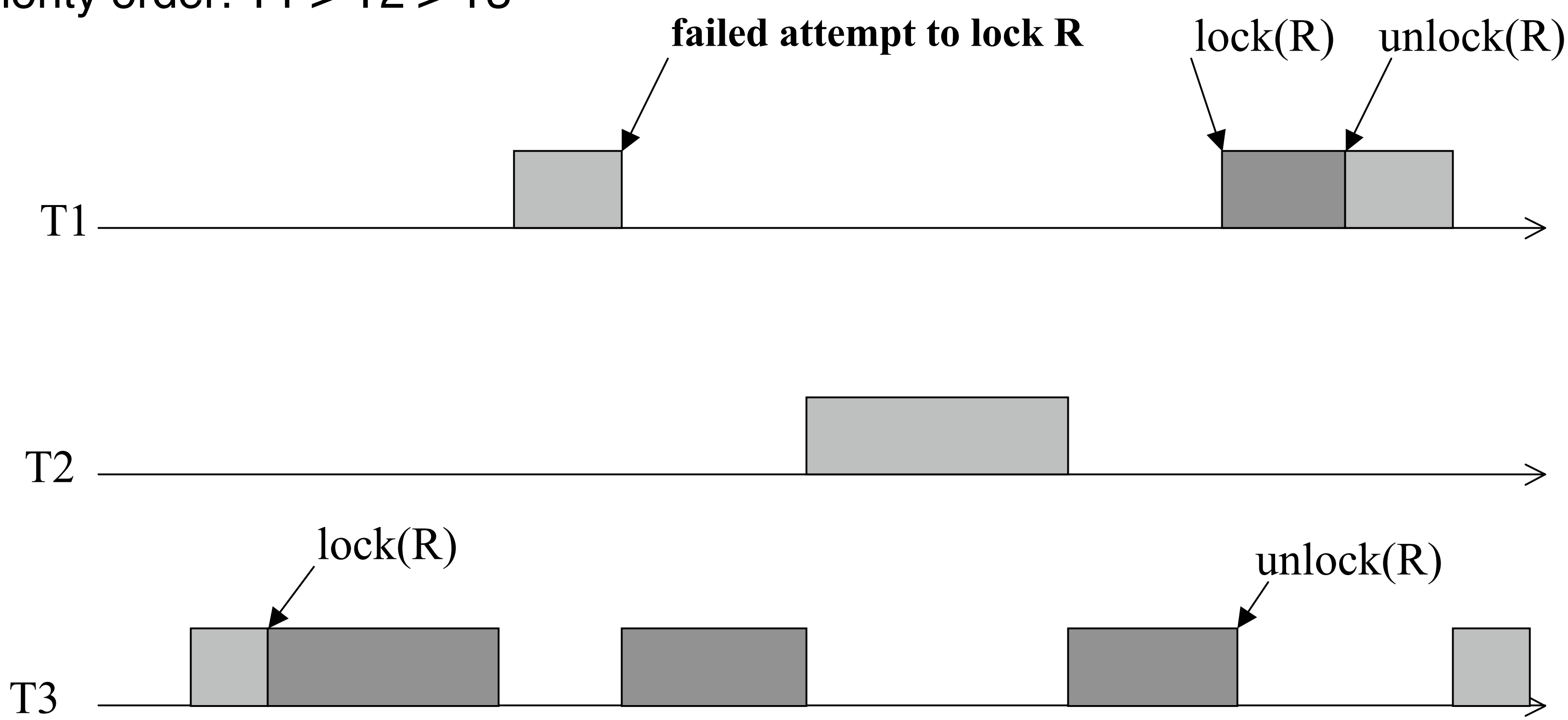


Scheduling Algorithm: RMS

- Rate Monotonic Scheduling (Liu and Layland, 1973)
- Based on static priorities.
- Preemptive: scheduling decision made when a task finishes as well as when a new task arrives
- Scheduling algorithm: Choose the task with smallest period (among ready tasks)
- It may happen that a set of tasks is schedulable by EDF, but not by RMS
- Theorem: If utilization is smaller than 0.7, then RMS is guaranteed to find one
  - If utilization is between 0.7 to 1, RMS may or may not succeed
- Example: Let's see what happens on the last example

The Priority Inversion Problem

Priority order: T1 > T2 > T3



T2 is causing a higher priority task T1 wait !

Priority Inversion

1. T1 has highest priority, T2 next, and T3 lowest
2. T3 comes first, starts executing, and acquires some resource (say, a lock).
3. T1 comes next, interrupts T3 as T1 has higher priority
4. But T1 needs the resource locked by T3, so T1 gets blocked
5. T3 resumes execution (this scenario is still acceptable so far)
6. T2 arrives, and interrupts T3 as T2 has higher priority than T3, and T2 executes till completion
7. In effect, even though T1 has priority than T2, and arrived earlier than T2, T2 delayed execution of T1
8. This is “priority inversion” !! Not acceptable.
9. Solution T3 should inherit T1’s priority at step 5

Priority Inheritance Protocol