Agenda - Scheduling

- First-come First-served
- Shortest Job First
- Round Robin
- Priority based scheduling on UNIX
- Priority Inversion Problem
- Real-time systems
  - Earliest Deadline First
  - Rate Monotonic Scheduling

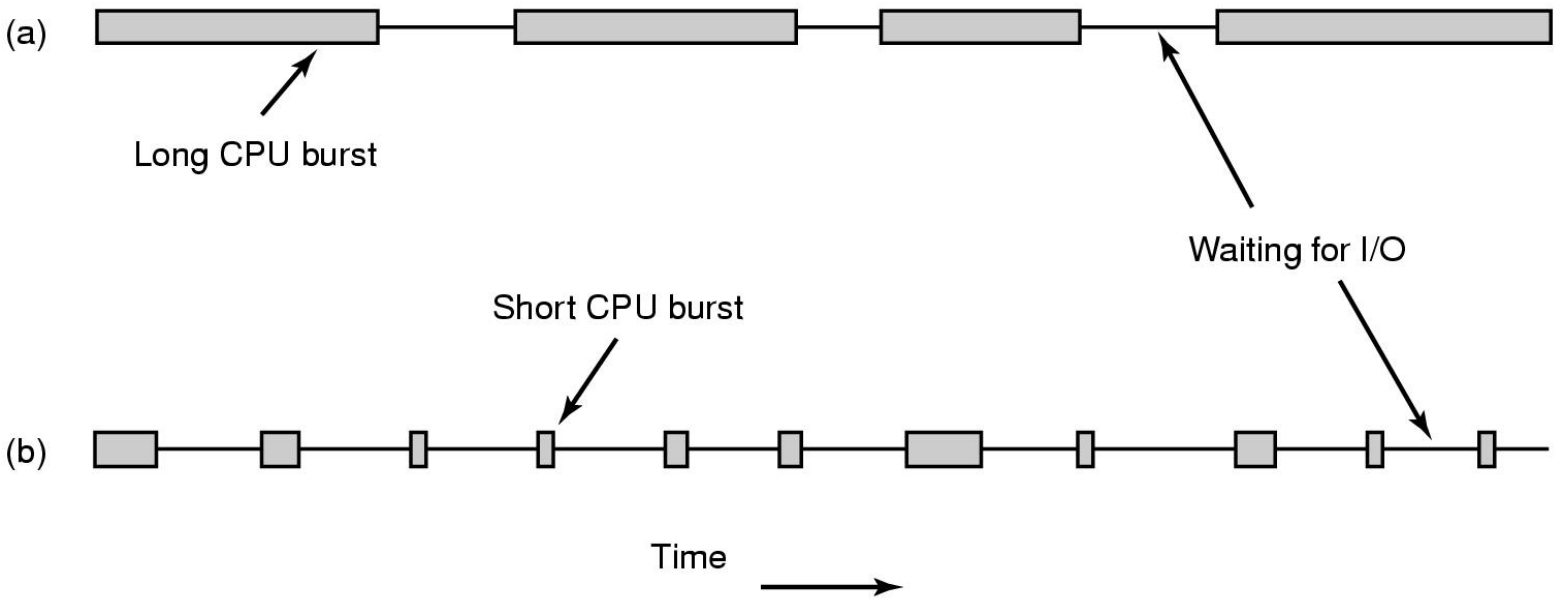
With slides by Dianna Xu

Scheduling

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

What are the characteristics of a good scheduling algorithm for processes/threads?

Process/Thread Behavior



- Bursts of CPU usage alternate with periods of I/O wait
  - (a) CPU-bound process
  - (b) I/O bound process

Scheduling Issues

- Application Profile:
  - Is a program CPU-bound or I/O-bound
- 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 mutex, signal, etc.)

Preemptive and Non-preemptive

- **A preemptive scheduling algorithm** picks a process and lets it run for some fixed time
  - Preemptive scheduling will interrupt a process at a clock interrupt

- **Non-preemptive scheduling** picks a process and lets it run until it blocks or otherwise voluntarily gives up the CPU
  - Non-preemptive scheduling does not schedule during clock interrupts

Scheduling Challenges

What should be optimized?

- **CPU utilization**: Fraction of time CPU is in use
  - e.g. $1 - p^n$, where p = % working and n = #processes
- **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

Example:

Total waiting time:

Average waiting time:

Total turnaround time:

Average turnaround time:

Scheduling Challenges

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

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

Can you think of some disadvantages?

Example of FCFS

- Workload (Batch system)
  Job 1: 24 units, Job 2: 3 units, Job 3: 3 units

- FCFS schedule:

- 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

Exercise: FCFS

- Workload (Batch system)
  Job 1: 10 units, Job 2: 12 units, Job 3: 9 units

FCFS schedule:

Total waiting time:

Average waiting time:

Total turnaround time:

Average turnaround time:

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

Aside: Exponential Averaging (Aging)

$$\tau_{n+1} = \alpha T_n + (1 - a)\tau_n$$

alpha = weighting factor

Controls how much we change in response to new values

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

- $\alpha = 0 \longrightarrow (\tau_{n+1} = \tau_n)$ //make no use of recent history
- $\alpha = 1 \longrightarrow (\tau_{n+1} = T_n)$ //past prediction not used
- $\alpha = \frac{1}{2}$ → weighted older bursts get less and less weight

Exercise: Exponential averaging

Question: How might aging with a factor of 0.5 be efficiently implemented?

Linear combination of the previous and current values

$$\tau_o = T_0$$
$$\tau_{n+1} = \alpha T_n + (1 - a)\tau_n, \qquad t > 0$$

Last value τ_n	Update T_n	Exponential moving average (EMA) τ_{n+1}
5 = $\tau_o = T_0$	5	5
6		
8		
7		

Example: 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

Exercise: SJF

- Workload (Batch system)
  Job 1: 10 units, Job 2: 12 units, Job 3: 9 units

FCFS schedule:

Total waiting time:

Average waiting time:

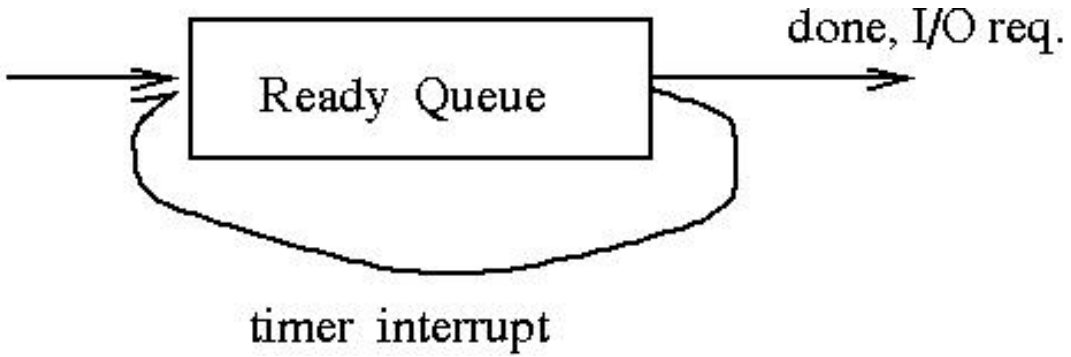
Total turnaround time:

Average turnaround time:

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: 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)

Exercise: RR

- Workload (Batch system), Quantum = 5 units
  Job 1: 10 units, Job 2: 12 units, Job 3: 9 units

FCFS schedule:

Total waiting time:

Average waiting time:

Total turnaround time:

Average turnaround time:

Properties of RR

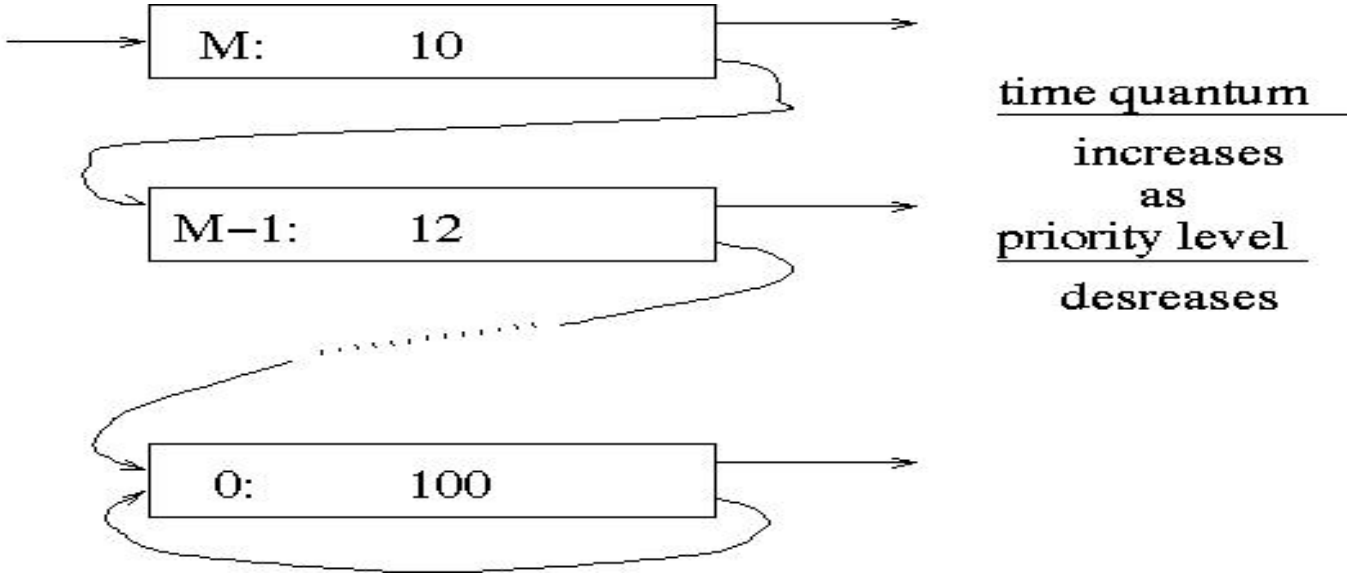
- Advantages: simple, low overhead, works for interactive systems
- Disadvantages: if quantum is too small, too much time wasted in context switching; if too large (i.e. longer than mean CPU burst), approaches FCFS.
- Typical value: 20 – 40 msec
- **Rule of thumb:** Choose quantum so that large majority (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

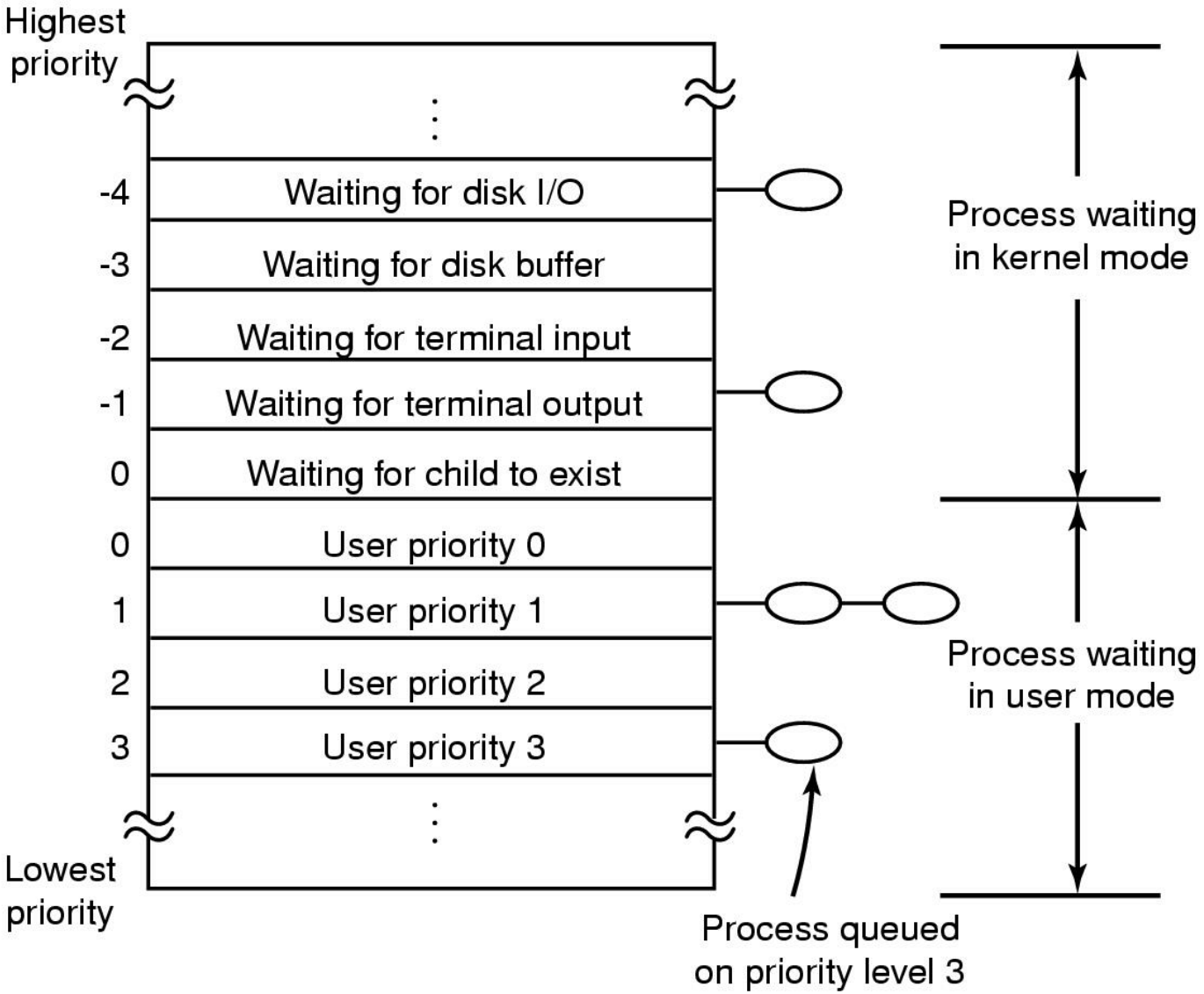
- Key Design Issue: how to compute priorities?

Multi-Level Feedback (FB)



- Each priority level has a ready queue, and a time quantum
- process enters highest priority queue initially, and drops to 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

UNIX Scheduler



Case Study: 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

- 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)

Example: Unix CPU Scheduler

- Two values in the PCB (Process Control Block)
  - **`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 is calculated periodically and the process is moved to appropriate ready queue

```
priority = base + p_cpu + p_nice
```

- 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: age by 1/2 (that is, add and shift right)
  - Motivation: Recent usage penalizes more than past usage
  - Precise details differ in different versions

Example: Unix scheduler

Suppose **`p_nice`** is 0, clock ticks every 10msec, time quantum is 100msec, and **`p_cpu`** ages every sec. Suppose initial base value is 4.

- Initially, **`p_cpu`** is 0
- Initial priority is 4.

1. Suppose scheduler selects this process at some point, and it uses all of its quantum without blocking.

2. Suppose again scheduler picks this process after 1 second, and it blocks (say, for disk read) after 30 msec.

3. Process is now in waiting queue for disk transfer

4. When disk transfer is complete after 1 second, disk interrupt handler computes priority using a negative base value, say, -10.

5. Process gets scheduled immediately, and runs for its entire time quantum.

Example: UNIX scheduler

clock tick: 10 msec
quantum: 100 msec
p_cpu aging: 1 sec

Time	base	c_cpu	p_nice	priority	queue
0	4	0	0	4	User 4
100 (Runs 100ms)	0	100/10 = 10	0	10	User 10
1100 (Waits 1s)	0	(10+0)/2 = 5	0	5	User 5
1130 (Runs 30ms)	0	8	0	8	User 8
2130 (Waits 1s)	0	(8+0)/2 = 4	0	4	User 4
2160 (Disk Transfer - Block)	-10	4	0	-6	Kernel -6
3000 (Runs 100ms)	0	14	0	14	User 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

Multi-level Feedback Discussion

I/O-bound processes tend to congregate in higher-level queues. (Why?)

CPU-bound processes will sink deeper(lower) into the queues. (Why?)

How does quantum affect scheduling?

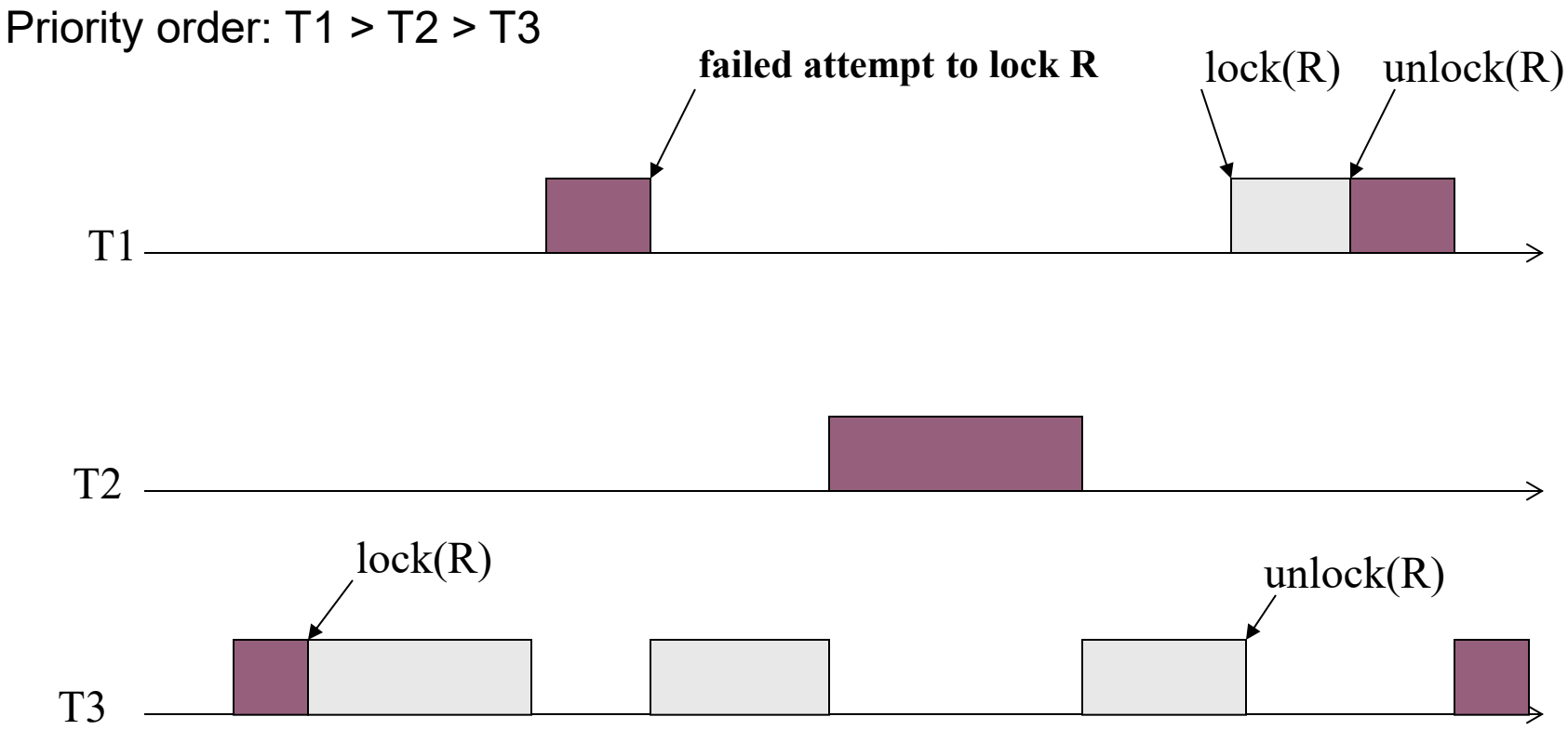
Multilevel feedback discussion

Queues could run with different quantum. How should quantum be chosen for different queues?

What happens when we start a process with a low priority?

What happens when a process moves from a CPU-bound to a I/O-bound process?

The Priority Inversion Problem

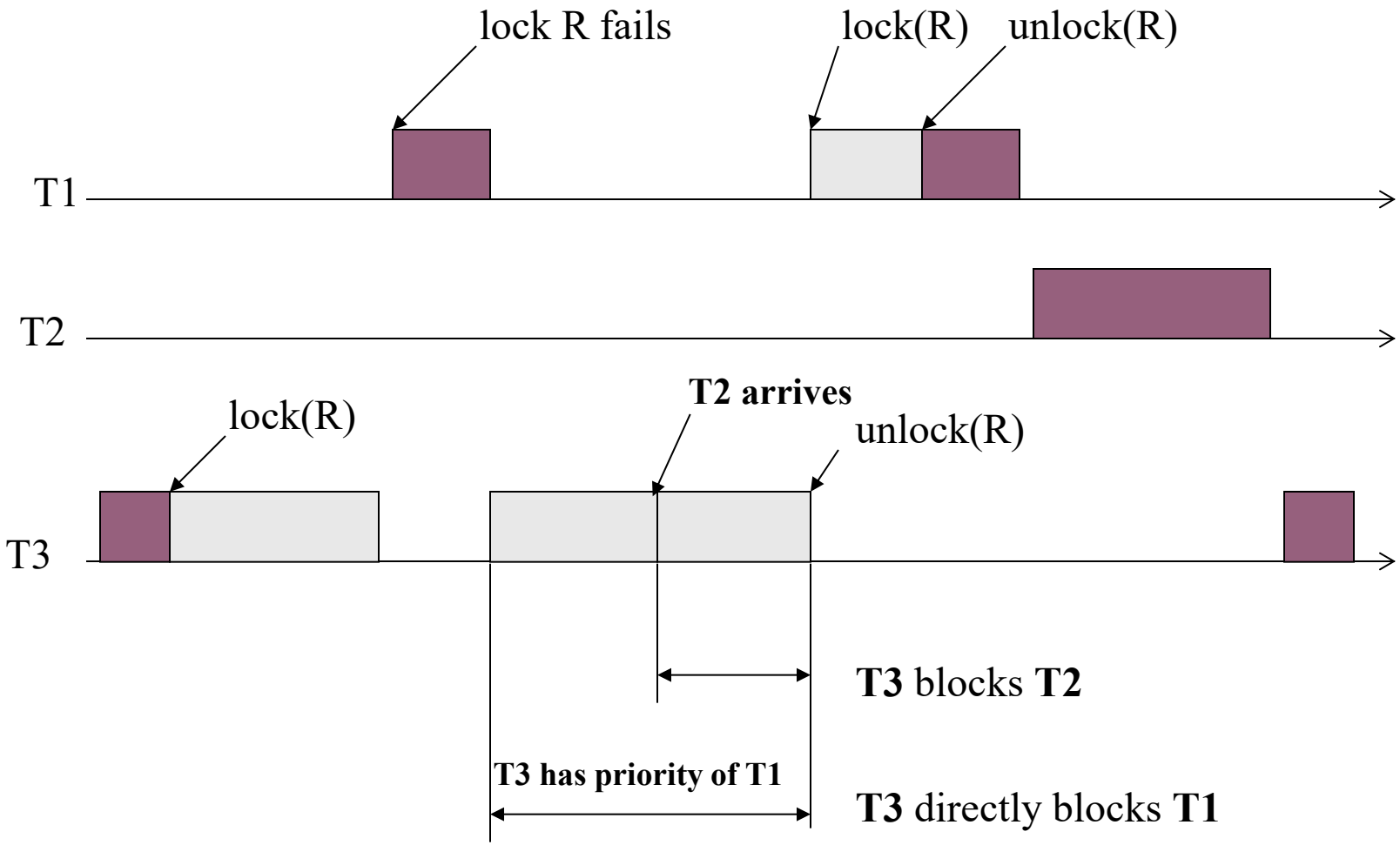


T2 is causing a higher priority task T1 wait !

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. Even though T1 has higher 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



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.

$\Rightarrow$ 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 $T_1, \ldots, T_n$. 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:

- Simple test:
  - Task T_i needs to use CPU for $\frac{C_i}{P_i}$ fraction per unit
  - Utilization = Sum of $\frac{C_i}{P_i}$
  - Task set is schedulable if and only if utilization is 1 or less.

Scheduling Algorithm: EDF

- Earliest Deadline First (EDF)
- Based on dynamic priorities.
- At each deadline, schedule the task with the next closest 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

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

EDF Example

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

RMS: Example

- Task T1 with period 4 and CPU time 1
- Task T2 with period 6 and CPU time 2
- Task T3 with period 12 and CPU time 3

What is the utilization?

RMS: 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

RMS Example