Operating Systems and Concurrency

Processes 4: Further Scheduling

COMP2007

Dan Marsden
(Geert De Maere)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Recap

Last Lecture

1. Threads are an abstraction of execution traces.
2. Threads vs. processes.
3. Thread implementations - user, kernel and hybrid.
4. PThreads.

Goals for Today

Overview

- **Multi-level feedback queues**.
- Scheduling in **Windows 7**.
- Scheduling in **Linux**.
- **Load balancing**.
- Scheduling **related processes/threads**.

Priority Queues

Recall

- Jobs can have **different priority levels**.
- Jobs of the **same priority** are run in **round robin** fashion.
- Usually implemented by using **multiple queues**, one for each priority level.

Multi-level Feedback Queues

Moving Beyond Priority Queues

- Different **scheduling algorithms** can be used for the **individual queues** (e.g., round robin, SJF, FCFS)
- **Feedback queues** allow **priorities to change dynamically**. Jobs can **move between queues**:
  - Move to **lower priority queue** if too much CPU time is used (prioritise I/O and interactive processes)
  - Move to **higher priority queue** to prevent **starvation** and avoid **inversion of control**

Exam 2013-2014: Explain how you would prevent starvation in a priority queue algorithm

Inversion of Control

Illustration

```
Process A (low)     Process B (high)     Process C (medium)
...
request X
receive X
...                 RUN
...                 request X
...                 blocked              RUN
...                 ...                  ...
```

Multi-level Feedback Queues

Moving Beyond Priority Queues

- Defining characteristics of feedback queues include:
  - The **number of queues**
  - The **scheduling algorithms** used for the individual queues
  - **Migration policy** between queues
  - Initial **access** to the queues
- Feedback queues are highly **configurable** and offer significant flexibility

Multi-level Feedback Queues

Windows 7

- An **interactive system** using a **preemptive scheduler** with **dynamic priority levels**
  - **Two priority classes** with **16 different priority levels** exist
    - "**Real time**" processes/threads have a **fixed priority level**
    - "**Variable**" processes/threads can have their priorities **boosted temporarily**
- A **round robin algorithm** is used within the queues

Multi-level Feedback Queues

Windows 7 (Cont'ed)

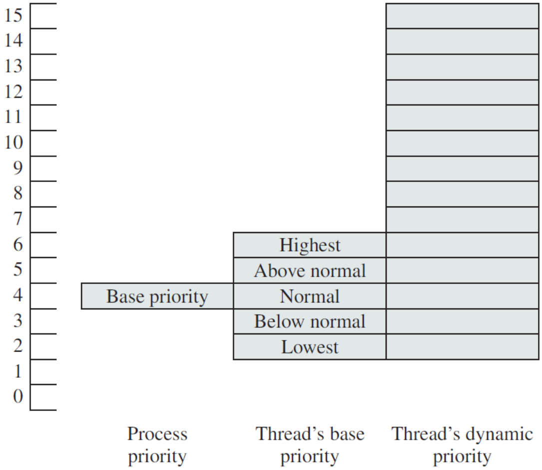


Figure: Priorities in Windows 7 (Stallings, 7^{th} edition)

Multi-level Feedback Queues

Windows 7 (Cont'ed)

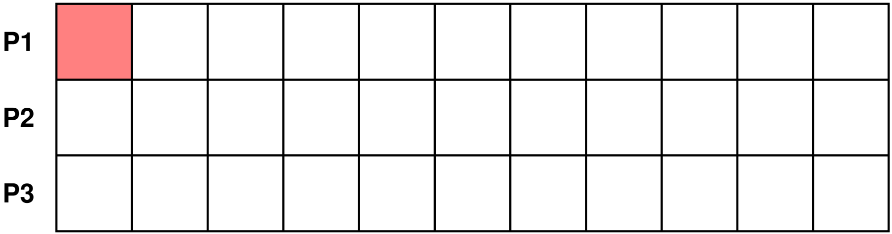
- Priorities are based on the **process base priority** (between 0-15) and **thread base priority** (± 2 relative to the process priority)
- A thread's **priority dynamically changes** during execution between its base priority and the maximum priority within its class
  - **Interactive I/O bound processes** (e.g. keyboard) receive a **larger boost**
  - Boosting priorities prevents **starvation** and **priority inversion**

Multi-level Feedback Queues

P1											
P2											
P3											

We consider three processes running on a single CPU machine:

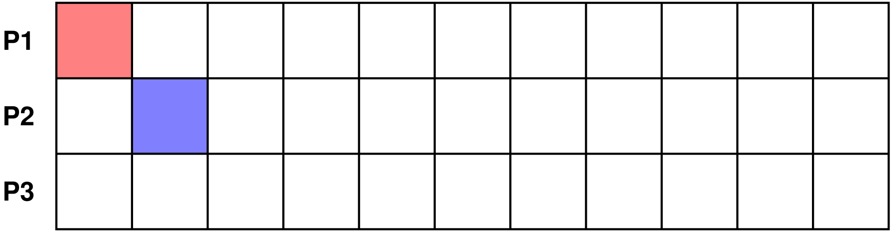
- Processes P1 and P2 have initial priority 1 (higher priority!).
- Process P3 has initial priority 2 (lower priority!).



We consider three processes running on a single CPU machine:

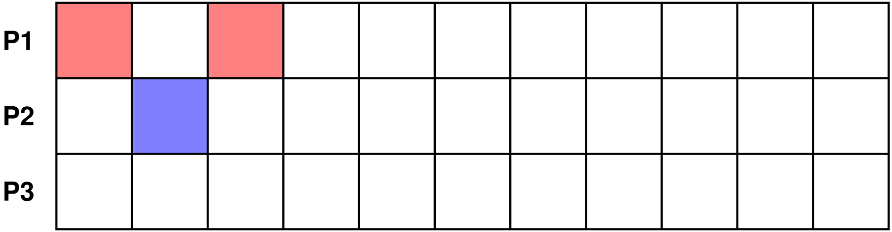
- Processes P1 and P2 have initial priority 1 (higher priority!).
- Process P3 has initial priority 2 (lower priority!).

Multi-level Feedback Queues



We consider three processes running on a single CPU machine:

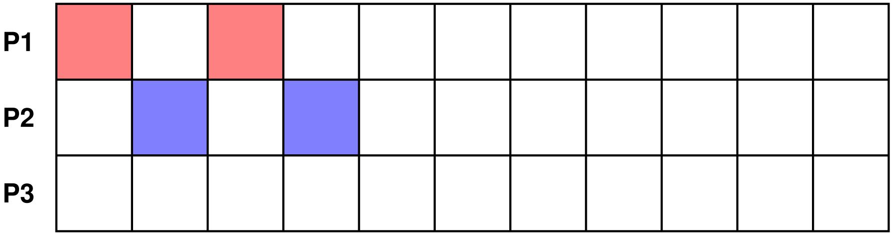
- Processes P1 and P2 have initial priority 1 (higher priority!).
- Process P3 has initial priority 2 (lower priority!).



We consider three processes running on a single CPU machine:

- Processes P1 and P2 have initial priority 1 (higher priority!).
- Process P3 has initial priority 2 (lower priority!).

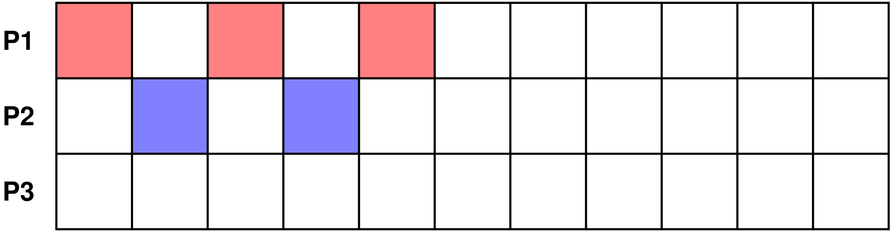
Multi-level Feedback Queues



We consider three processes running on a single CPU machine:

- Processes P1 and P2 have initial priority 1 (higher priority!).
- Process P3 has initial priority 2 (lower priority!).

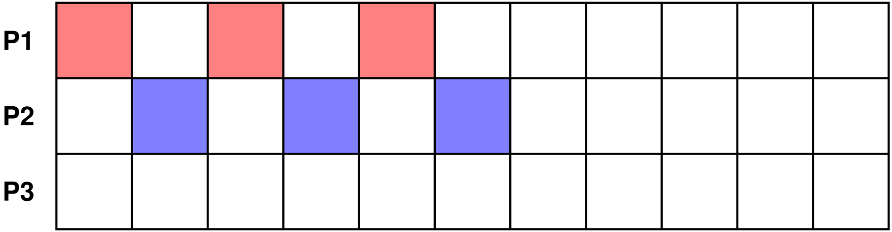
Multi-level Feedback Queues



We consider three processes running on a single CPU machine:

- Processes P1 and P2 have initial priority 1 (higher priority!).
- Process P3 has initial priority 2 (lower priority!).

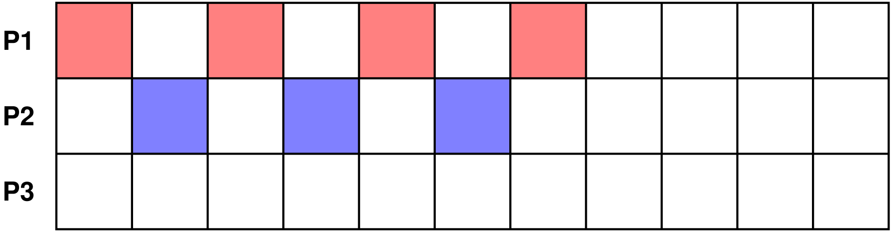
Multi-level Feedback Queues



We consider three processes running on a single CPU machine:

- Processes P1 and P2 have initial priority 1 (higher priority!).
- Process P3 has initial priority 2 (lower priority!).

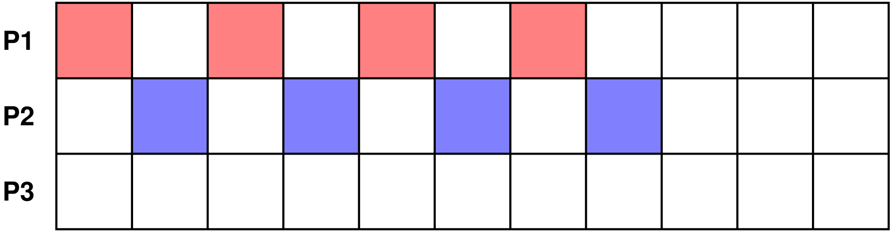
Multi-level Feedback Queues



We consider three processes running on a single CPU machine:

- Processes P1 and P2 have initial priority 1 (higher priority!).
- Process P3 has initial priority 2 (lower priority!).

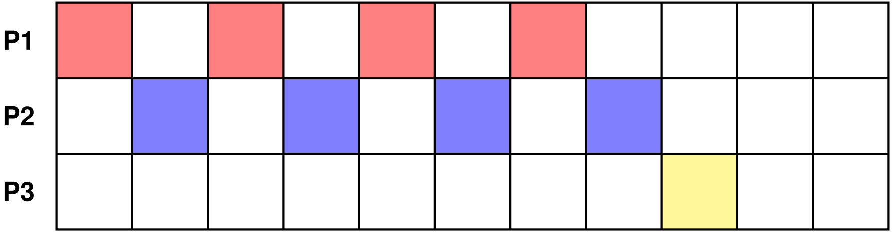
Multi-level Feedback Queues



We consider three processes running on a single CPU machine:

- Processes P1 and P2 have initial priority 1 (higher priority!).
- Process P3 has initial priority 2 (lower priority!).
- The scheduler concludes P3 is being starved of CPU time.

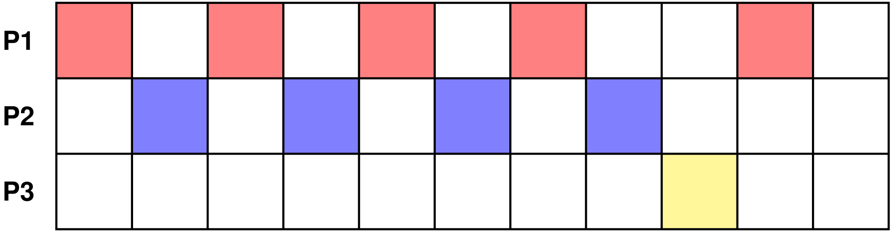
Multi-level Feedback Queues



We consider three processes running on a single CPU machine:

- Processes P1 and P2 have initial priority 1 (higher priority!).
- Process P3 has initial priority 2 (lower priority!).
- The scheduler concludes P3 is being starved of CPU time.
- Process P3 has it's priority temporarily promoted to prevent starvation.

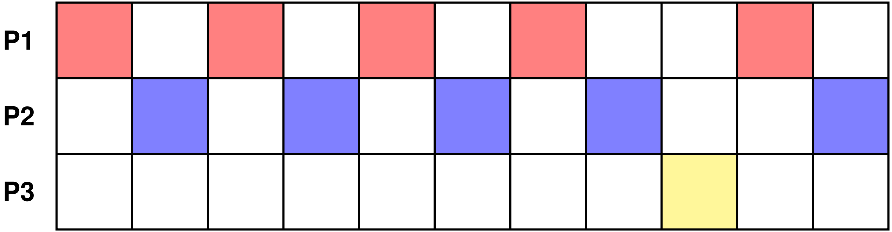
Multi-level Feedback Queues



We consider three processes running on a single CPU machine:

- Processes P1 and P2 have initial priority 1 (higher priority!).
- Process P3 has initial priority 2 (lower priority!).
- The scheduler concludes P3 is being starved of CPU time.
- Process P3 has it's priority temporarily promoted to prevent starvation.
- Computation continues as before.

Multi-level Feedback Queues



We consider three processes running on a single CPU machine:

- Processes P1 and P2 have initial priority 1 (higher priority!).
- Process P3 has initial priority 2 (lower priority!).
- The scheduler concludes P3 is being starved of CPU time.
- Process P3 has it’s priority temporarily promoted to prevent starvation.
- Computation continues as before.

Scheduling in Linux

The Completely Fair Scheduler

- Process scheduling has **evolved** over different versions of Linux to make efficient use of **multiple processors/cores**
- Linux distinguishes between two types of tasks for scheduling:
  - **Real time tasks** (to be POSIX compliant), divided into:
    - Real time FIFO tasks
    - Real time Round Robin tasks
  - **Time sharing tasks** using a **preemptive** approach which are similar to **variable** in Windows.
- The most recent scheduling algorithm in Linux for **time sharing tasks** is the **completely fair scheduler (CFS)**.

- **Real time FIFO** tasks have the **highest priority** and are scheduled using a **FCFS approach**, using **preemption if a higher priority** job shows up
- **Real time round robin tasks** are preemptable by **clock interrupts** and have a **time slice** associated with them
- Both approaches **cannot guarantee hard deadlines**

The Ideal Fair Scheduler

We imagine a hypothetical ideal scenario:

- Our CPU allows all N current tasks to be run simultaneously, with each receiving $\frac{1}{N}$ of the CPU power.
- For example, with 5 tasks wanting to run, each gets 20% of the available computational power.
- Unfortunately real CPUs cannot run an arbitrary number of tasks in parallel in this way - but can we approximate this ideal?

Deciding how to divide up the CPU time

- We choose a **target latency** - this is the amount of time before every task gets access to the CPU. The target latency also bounds how far we will drift from being fair.
- To hit this target latency, for N tasks, each task is allowed to run for $\frac{1}{N}$ of the target latency.
- To avoid excessive context switching when N is large, we also choose a **minimum granularity** - a minimum amount of time we will allow a task to run on the CPU before being considered for replacement.

Approximating Fairness

- We record a **virtual time** that each task has had on the CPU, and order tasks by their virtual CPU time.
- Tasks are ordered in **ascending order of virtual time used** - implemented using a red-black tree.
- The task with the lowest virtual time on the CPU is considered to have been **treated least fairly**, and will be the next one chosen to run on the CPU.
- After that task has had $\frac{1}{N}$ of the target latency in virtual time, we replace it with the next task with lowest virtual run time.
- Note - system calls may lead to a task using less than its full allocated time. **This will mean they will get back on the CPU more quickly.**

Accounting for priorities

- A **weighting scheme** is used to take different priorities into account - we will assume that the weight is literally the task priority - a simplification!
- The recorded virtual time on the CPU is the really time on the CPU scaled up by the weight. After 100ms of actual computation time:
  - A priority 1 (higher priority) process is considered to have used 100ms of virtual time.
  - A priority 2 (lower priority) process is considered to have used 200ms of virtual time.
- **Virtual time runs at different speeds for different priority processes!**
- Note that tasks will be given varying windows of time to run - unlike traditional time slicing.

Assume three tasks T1, T2, and T3, with priorities 1,2 and 3 respectively, and target latency is 300ms. Write $T(v, r)$ to indicate task T has had v units of virtual run time, and r units of real time. The state after each 100ms of **virtual time** is:

1. CPU: $T1(0, 0)$, queue: $T2(0, 0), T3(0, 0)$

Assume three tasks T1, T2, and T3, with priorities 1,2 and 3 respectively, and target latency is 300ms. Write $T(v, r)$ to indicate task T has had v units of virtual run time, and r units of real time. The state after each 100ms of **virtual time** is:

1. CPU: $T1(0, 0)$, queue: $T2(0, 0), T3(0, 0)$
2. CPU: $T2(0, 0)$, queue: $T3(0, 0), T1(100, 100)$

Scheduling in Linux

Time Sharing Tasks - Example

Assume three tasks T1, T2, and T3, with priorities 1,2 and 3 respectively, and target latency is 300ms. Write $T(v, r)$ to indicate task T has had v units of virtual run time, and r units of real time. The state after each 100ms of **virtual time** is:

1. CPU: $T1(0, 0)$, queue: $T2(0, 0)$, $T3(0, 0)$
2. CPU: $T2(0, 0)$, queue: $T3(0, 0)$, $T1(100, 100)$
3. CPU: $T3(0, 0)$, queue: $T1(100, 100)$, $T2(100, 50)$

Assume three tasks T1, T2, and T3, with priorities 1,2 and 3 respectively, and target latency is 300ms. Write $T(v, r)$ to indicate task T has had v units of virtual run time, and r units of real time. The state after each 100ms of **virtual time** is:

1. CPU: $T1(0, 0)$, queue: $T2(0, 0), T3(0, 0)$
2. CPU: $T2(0, 0)$, queue: $T3(0, 0), T1(100, 100)$
3. CPU: $T3(0, 0)$, queue: $T1(100, 100), T2(100, 50)$
4. CPU: $T1(100, 100)$, queue $T2(100, 50), T3(100, 33)$

Assume three tasks T1, T2, and T3, with priorities 1,2 and 3 respectively, and target latency is 300ms. Write $T(v, r)$ to indicate task T has had v units of virtual run time, and r units of real time. The state after each 100ms of **virtual time** is:

1. CPU: $T1(0, 0)$, queue: $T2(0, 0), T3(0, 0)$
2. CPU: $T2(0, 0)$, queue: $T3(0, 0), T1(100, 100)$
3. CPU: $T3(0, 0)$, queue: $T1(100, 100), T2(100, 50)$
4. CPU: $T1(100, 100)$, queue $T2(100, 50), T3(100, 33)$
5. CPU: $T2(100, 50)$, queue $T3(100, 33), T1(200, 200)$

Assume three tasks T1, T2, and T3, with priorities 1,2 and 3 respectively, and target latency is 300ms. Write $T(v, r)$ to indicate task T has had v units of virtual run time, and r units of real time. The state after each 100ms of **virtual time** is:

1. CPU: $T1(0, 0)$, queue: $T2(0, 0)$, $T3(0, 0)$
2. CPU: $T2(0, 0)$, queue: $T3(0, 0)$, $T1(100, 100)$
3. CPU: $T3(0, 0)$, queue: $T1(100, 100)$, $T2(100, 50)$
4. CPU: $T1(100, 100)$, queue $T2(100, 50)$, $T3(100, 33)$
5. CPU: $T2(100, 50)$, queue $T3(100, 33)$, $T1(200, 200)$
6. ...

Further enhancements

To avoid potential pathological behaviours:

- New tasks have their virtual run time set to the **current minimum virtual run time** - Think about how unfairly advantaged they might be if this was set to zero!

Further enhancements

To avoid potential pathological behaviours:

- New tasks have their virtual run time set to the **current minimum virtual run time** - Think about how unfairly advantaged they might be if this was set to zero!
- Blocked tasks have their virtual run time set to the greater of:
  - The **current minimum virtual run time, minus a small offset** - to ensure it gets to run.
  - Its **old virtual run time** - in this case it is already getting a good share of the CPU.

  - Think about how long a task that was blocked for a long time might get on the CPU otherwise!

- **Single processor** machine: **which thread** to run next?
- Scheduling decisions on a **multi-core** machine include:
  - Which thread to run **when**?
  - Which thread to run **where**?

- A single or multi-level queue **shared** between all CPUs
- Advantage: automatic **load balancing**
- Disadvantages:
  - **Contention** for the queues.
  - Does not take advantage of the current state of the CPU's
    - **Cache** becomes invalid when moving to a different CPU
    - Translation look aside buffers (**TLBs** - part of the MMU) become invalid

Multi-processor Scheduling

Private Queues

- Each CPU has a **private queue or queues**.
- Advantages:
  - Often can reuse existing CPU state such as cache and TLB
  - **Contention** for shared queue is minimised
- Disadvantages: less **load balancing**
- To mitigate the lack of load balancing **migration** between CPUs is possible

Related vs. Unrelated Threads

Thread Types

- **Related**: **multiple threads** that **communicate** with one another and **ideally run** together (e.g. search algorithm)
- **Unrelated**: e.g. processes threads that are **independent**, possibly started by **different users** running **different programs**

Scheduling Related Threads

Working Together

- E.g., threads belong to the same process and are **cooperating**, e.g. they **exchange messages** or **share information**, e.g
  - Process A has thread A_0 and A_1, A_0 and A_1 cooperate
  - Process B has thread B_0 and B_1, B_0 and B_1 cooperate
  - The scheduler selects A_0 and B_1 to run first, then A_1 and B_0, and A_0 and A_1, and B_0 and B_1 run on different CPUs
  - They try to send messages to the other threads, which are still in the ready state

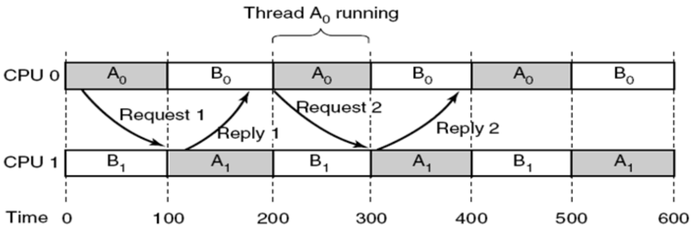


Figure: Tanenbaum, Chapter 8

Scheduling Related Threads

Working Together

- The aim is to get **collaborating threads** running, as much as possible, at the **same time** across **multiple CPUs**
- Approaches include:
  - **Space** sharing
  - **Gang** scheduling

Scheduling Related Threads

Space sharing

- Approach known as **space sharing scheduling**:
  - *N* related threads, typically from a single process, are allocated to *N* **dedicated CPUs** when enough CPUs are available.
  - *M* related threads, typically from another process, are **kept waiting until *M* CPUs are available.**
  - At any point in time the available CPUs are partitioned into blocks of related threads.
  - As thread complete, their dedicated CPUs are returned to the collection of available CPUs.
  - The CPUs are **not multiprogrammed** to keep related threads running together. This means blocking calls result in **idle CPUs**.

Scheduling Related Threads

Gang scheduling

- Space sharing scheduling shares work by space (CPU)
  - Keeps related threads running together.
  - Lack of multiprogramming avoids context switching overhead, but leads to wasted CPU cycles.
  - **Gang scheduling** is an attempt to schedule “in both time and space” to avoid this waste of CPU time.

Scheduling Related Threads

Gang scheduling

- The scheduler **groups related threads** together into **gangs** to run simultaneously on different CPUs.
- This is a **preemptive** algorithm, with time slices synchronised across all CPUs.
- **Blocking threads** result in idle CPUs
  - If a thread blocks the rest of the time slice will be unused, due to the time slice synchronisation across all CPUs.

Time slot	CPU 0	CPU 1	CPU 2	CPU 3	CPU 4	CPU 5
0	A_0	A_1	A_2	A_3	A_4	A_5
1	B_0	B_1	B_2	C_0	C_1	C_2
2	D_0	D_1	D_2	D_3	D_4	E_0
3	E_1	E_2	E_3	E_4	E_5	E_6
4	A_0	A_1	A_2	A_3	A_4	A_5
5	B_0	B_1	B_2	C_0	C_1	C_2
6	D_0	D_1	D_2	D_3	D_4	E_0
7	E_1	E_2	E_3	E_4	E_5	E_6

Test your understanding

- Why would boosting thread priorities prevent priority inversion?
- Why is it efficient to schedule threads that communicate with each other at the same time?
- How does the CFS avoid starvation?