Operating Systems and Concurrency

Lecture 6: Process Scheduling

G52OSC/COMP2007

Geert De Maere
(Isaac Triguero)
{Geert.DeMaere,Isaac.Triguero}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2018

Recap

Last Lecture

1. Threads vs. processes
2. Thread implementations (user, kernel and hybrid)
3. PThreads

Goals for Today

Overview

- **Multi-level feedback queues**
- Scheduling in **Window 7** – illustration
- Scheduling in **Linux** (implementation in labs)
- CPU **affinity** and **load balancing**
- Scheduling **related processes/threads**

Priority Queues

Recall

- Jobs can have **different priority levels** that are **fixed**
- Jobs of the **same priority** are run in **round robin** fashion
- Priority queues are 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**, i.e., 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 (high)
...
request X
receive X
...                  RUN
...                  request X
...                  blocked               RUN
...                  ...                   ...
```

- 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)

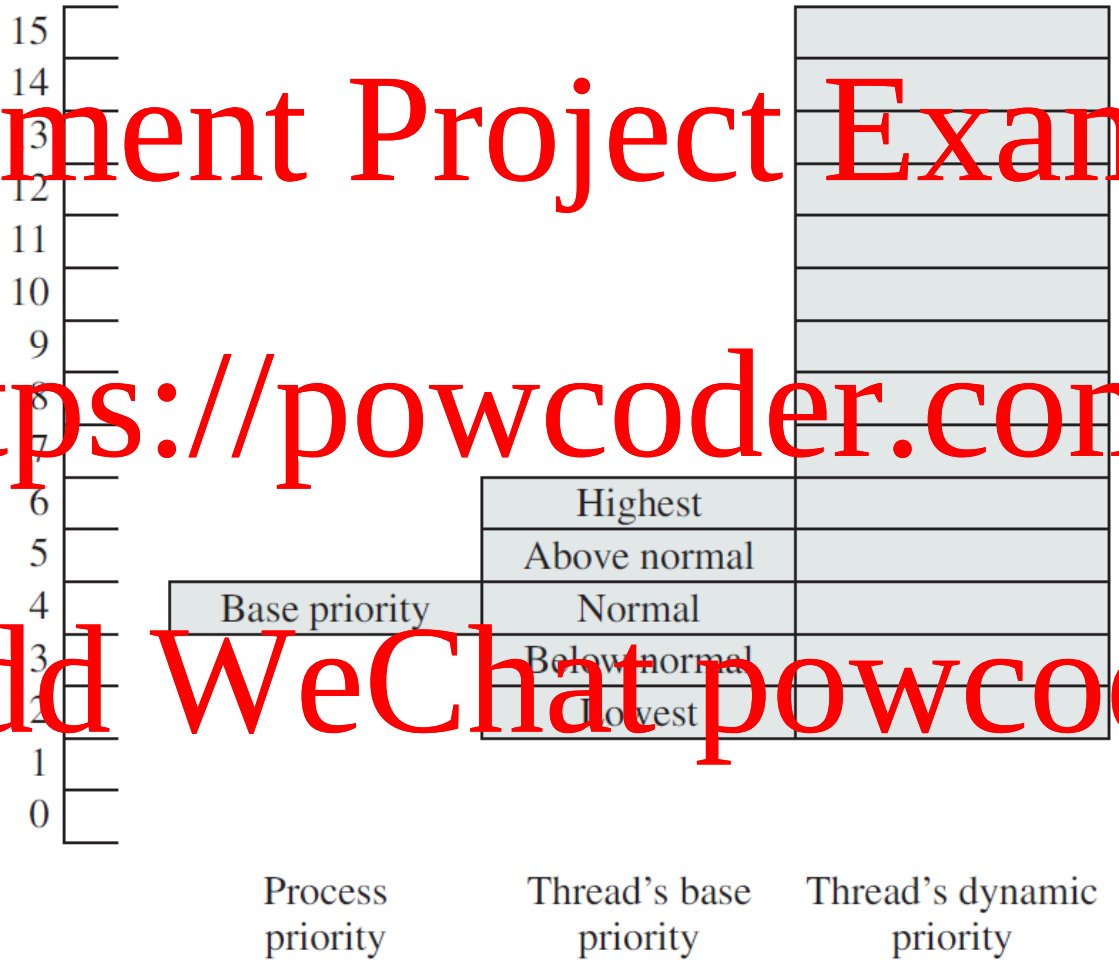


Figure: Priorities in Windows 7 (Stallings, 7^{th} edition)

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 **priority inversion**

- Code available on Moodle
- Examples:
  1. Variable process, seven threads, equal priority, 1 core/processor
  2. Variable process, three threads, different priorities, 1 core/processor
  3. Real time process, three threads, different priorities, 1 core/processor, admin privileges

- Process scheduling has **evolved** over different versions of Linux to account for **multiple processors/cores**, processor **affinity**, and **load balancing** between 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 (similar to **variable** in Windows)
- The most recent scheduling algorithm in Linux for **time sharing tasks** is the "**completely fair scheduler**" (CFS, before the 2.6 kernel, this was an $O(1)$ scheduler)

- **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 CFS **divides the CPU time** between all processes
- If all N processes have the **same priority**:
  - They will be allocated a "time slice" equal to $\frac{1}{N}$ times the available CPU time
  - I.e., if N equals 5, every process will receive 20% of the processor's time
- The length of the **time slice** and the "available CPU time" are based on the **targeted latency** ($\Rightarrow$ every process should **run at least once** during this interval)
- If N **is very large**, the **context switch time will be dominant**, hence a lower bound on the "time slice" is imposed by the minimum granularity
  - A process's time slice can be **no less** than the **minimum granularity** (response time will deteriorate)

- A **weighting scheme** is used to take different priorities into account
- If process have **different priorities**:
  - Every process i is allocated a **weight** w_i that reflects its priority
  - The "time slice" allocated to process i is then **proportional to** $\frac{w_i}{\sum_{j \in N} w_j}$
- The tasks with the **lowest proportional amount** of "**used CPU time**" are **selected first**

- **Single processor** machine: **which process (thread)** to run next (one dimensional)
- Scheduling decisions on a **multi-processor/core** machine include:
  - Which process (thread) to run **where**, i.e., which CPU?
  - Which process (thread) to run **when**?

- A single or multi-level queue **shared** between all CPUs
- Advantage: automatic **load balancing**
- Disadvantages:
  - **Contention** for the queues (locking is needed)
  - *"All CPUs are equal, but some are more equal than others"*: does not account for **processor affinity**:
    - **Cache** becomes invalid when moving to a different CPU
    - Translation look aside buffers (**TLBs** - part of the MMU) become invalid
- Windows will allocate the **highest priority threads** to the individual CPUs/cores

- Each CPU has a **private (set) of queues**
- Advantages:
  - **CPU affinity** is automatically satisfied
  - **Contention** for shared queue is minimised
- Disadvantages: less **load balancing**
- **Push** and **pull migration** between CPUs is possible

- **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

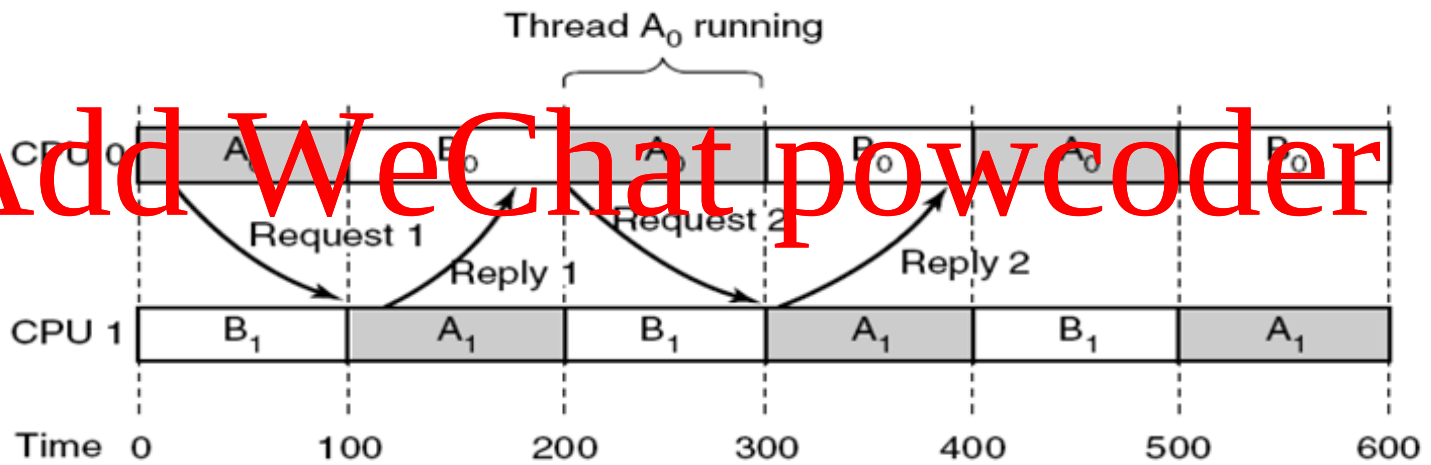


Figure: Tanenbaum, Chapter 8

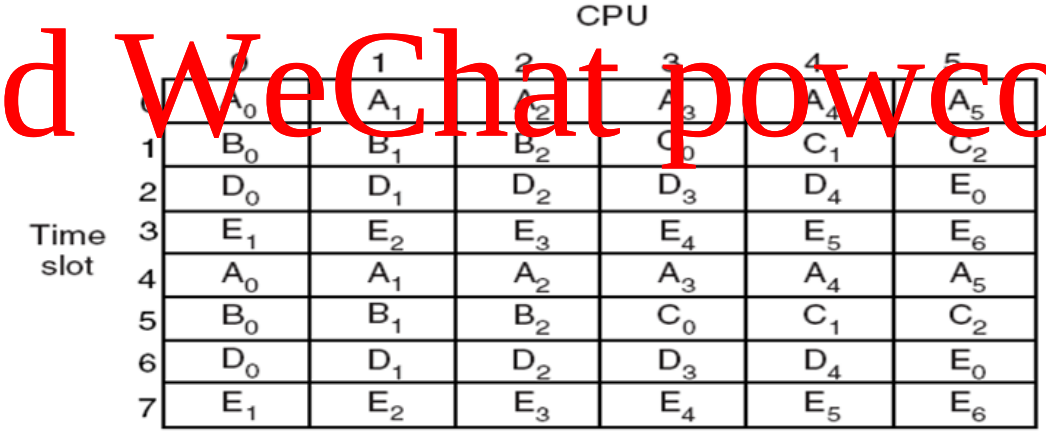
- The aim is to get threads **running**, as much as possible, at the **same time** across **multiple CPUs**
- Approaches include:
  - **Space** sharing
  - **Gang** scheduling

- Approach:
  - N threads are allocated to N **dedicated CPUs**
  - N threads are kept waiting until N CPUs are available
  - **Non-preemptive**, i.e. blocking calls result in **idle CPUs** (less context switching overhead but results in CPU idle time)
- The number N can be **dynamically adjusted** to match processor capacity

Scheduling Related Threads

Gang scheduling

- Time slices are **synchronised** and the scheduler **groups threads** together to run simultaneously (as much as possible)
- A **preemptive** algorithm (timer interrupts are still used)
- **Blocking threads** result in idle CPUs
  - I.e. if a thread blocks, e.g., due to an I/O call, the rest of the time slice will be unused (due to the time slice synchronisation across all CPUs)



Time slot	CPU 0	1	2	3	4	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

- Scheduling on **Windows** and **Linux**
- **Multi-processor/core** scheduling is "a bit different" (load balancing, processor affinity, etc.)
  - **Related** and **unrelated** threads
  - **Shared** or **private** queues
  - **Space** scheduling or **gang** scheduling