

Lecture 10 CPU scheduling

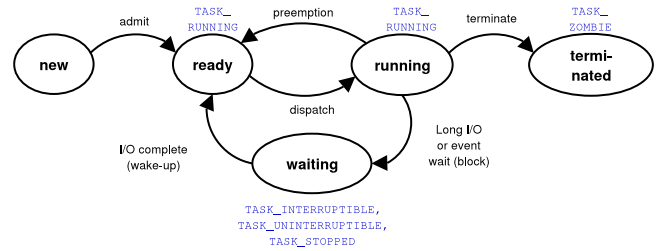
At the same time, there can be multiple running threads in the computer that can take the CPU. **Which ones should we choose to run?**

References:

- [OSC] Section 6.1–6.4 (CPU Scheduling).
- [ULK] pp. 277–295 (Process Scheduling).

POS(0230A)

Review: thread states



When a thread changes from running to another state, the scheduler must be executed to select another job to run.

But the causal relation can be reversed: the current thread don't want to give up the CPU, but the scheduler is somehow run (typically by interrupt handlers), preempting the currently running thread.

POS(0230A)-10.1

CPU and I/O Bursts

The thread is mostly in 3 state: "ready", "running" and "waiting".

- The time the thread spends on "ready" state is **wasted**: the thread could use the CPU, but the CPU is serving other threads.
- Ignoring that, the process would **switch** between **2** states: "running" and "waiting".
- The length of time during a consecutive period (ignoring ready periods) when the thread is **running** is called a **CPU burst**, and that when the thread is **waiting** is called an **I/O burst**.
- A thread with long CPU bursts is called CPU-bound, and one with short CPU bursts is called I/O bound.
- **Interactive threads** that wait for inputs are likely to be I/O bound. Although it only means it has long I/O bursts rather than short CPU bursts.

POS(0230A)-10.2

The scheduler

The **scheduler** is responsible for **choosing a thread to run**, when...

- A process give up the CPU, because it needs to do I/O, finish execution, or "yield" the CPU (i.e., nicely giving it up to let other process run).
- An interrupt occurs, leading to the **preemption** of a process.

Input: a list of ready threads; **Output:** which thread to execute.

Schedulers can **maintain information** along the threads in the PCB. E.g.,

- How long the process had previously sleep?
- How long is the previous CPU bursts?
- A list of ready threads in the order they are previously scheduled.

Of course, all these information adds to the size of PCB, so we must weigh the extra storage against the resulting gain in performance of scheduler.

POS(0230A)-10.3

A simple scheduler: First Come First Serve (FCFS)

A simple strategy... keep a queue of ready threads: **ready queue**; without any other extra information.

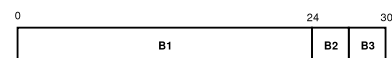
- When a thread is **created** or are **woken** up (i.e., when a CPU burst starts), the thread is put to the end of the **ready queue**.
- Whenever the scheduling algorithm is invoked, the thread in the head of the ready queue is selected.
- The thread runs until it completes or needs to wait (i.e., the CPU burst ends).

The algorithm does not perform well... let's illustrate it with an example.

FCFS example

Thread	Burst	Burst starts	Burst length
1	B1	0	24
2	B2	1	3
3	B3	1	3

Scheduling:



Such a diagram is called "Gantt charts", showing when each CPU burst uses the CPU. Here, each CPU burst comes from a different thread. For simplicity, we will only consider such cases in our examples.

The long CPU burst B1 seriously delays the completion of the short bursts B2 and B3.

POS(0230A)-10.4

POS(0230A)-10.5

Why is it bad?

But "B1 delays B2/B3 so it is bad" is not enough. Let's be more specific. After all, if we reverse the order, B2 and B3 will delay B1.

- It is **unfair**: the CPU burst of B1 can be very long, and all other processes will have no chance to run at all.
- It is **not responsive**. If B2 and B3 are interactive, users won't feel it reasonable arbitrarily longer than the time really needed.
- It lengthens the **average completion time**, since it is possible that all processes completes only after B1 completes.
- It reduces **I/O device utilization**, since all I/O-bound processes will be waiting excessively long for CPU-bound ones. We call this the **convoy effect**.

For these reasons, FCFS is usable only when **all CPU bursts are known to be short**, e.g., within the kernel.

POS(0230A)-10.6

Round Robin (RR)

One simple way to avoid the problem: **chop up the CPU bursts!**

- How to implement it? We just need to setup the **timer device** in the dispatcher, so that an **interrupt** will be generated, say, 1ms later.
- If the timer interrupt occurs, the thread is **preempted**.
- The "1ms" is a rather arbitrary value. It is called the **time-slice** or **time quantum** of the scheduler.
- We also must deal with synchronizations problems. By preventing preemptions or interrupts, or by locking...

The result: we **gain responsiveness**, although the scheduler is called **more often** and eat an additional amount of CPU time.

A less obvious effect: process switches more frequently, so **locality is reduced**, resulting in less effective cache (cost to pay anyway).

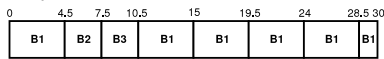
POS(0230A)-10.7

Example

The same CPU burst sequence:

Thread	Burst starts	Burst length
B1	0	24
B2	1	3
B3	1	3

Gantt chart with a quantum size of 4.5:



Note that B2 and B3 completes much earlier than in FCFS.

N.B.: The time quantum of B2 and B3 are not completely used up, but since nothing else needs to be done, the time slice is cut short.

POS(0230A)-10.8

How to set the time quantum?

- **Shorter** time quantum means the scheduler is invoked more often, which means **more scheduling overheads**—which reduce the time available to the processes.
- **Longer** time quantum risks making the scheduler too "FCFS-like": **unresponsive, unfair, low I/O device utilization, etc.**
- **Best value**: such a value that **most CPU bursts** of I/O-bound threads can complete within **1 time quantum**.
Shorter time quantum gives no benefit but increase scheduling overheads. Longer time quantum causes FIFO behaviour to show up.
- How to find that "best time quantum", then? By magic! (i.e., experiments.)

POS(0230A)-10.9

Priority Scheduling

A different approach would consider threads to have **different priorities**. These priorities are specified when the threads are created.

It is designed to be unfair.

- The ready queue is implemented as a **priority queue**. Usually, a small number (around 100) priorities to reduce cost of queue lookup.
- Every time the scheduling algorithm is invoked, the **ready thread with the highest priority** is selected.
- Can be **preemptive**, which allows a high priority thread to preempt a low priority thread when it is *released* or is *waken up*.
- Note that priority scheduling involves **no timer interrupt**.

This is suitable if some threads are absolutely more important than the other, and must be serviced first.

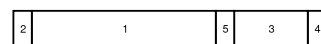
Usually in real-time settings, or for system threads.

POS(0230A)-10.10

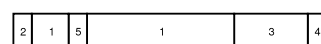
Gantt chart

Burst	Burst starts	Burst length	Priority
B1	0	10	3
B2	0	1	1
B3	0	2	4
B4	0	1	5
B5	3	5	2

Gantt chart: Non-preemptive priority scheduling



Gantt chart: Preemptive priority scheduling



POS(0230A)-10.11

Starvation and aging

- Since priority schemes are not fair, it is possible that a low priority thread never get serviced. The thread is said to be **starved** from the CPU.
- One solution: gradually increase the priority of threads when it is in ready state: **aging**. E.g., one may decrement priority per 200 ms inactivity.
- This results in a more **complicated** algorithm: we must now tune the rate at which aging occurs.
- This also introduces timer interrupt, and increases the **scheduling overhead** to periodically change the priority of each thread.

POS(0230A)-10.12

Priority inversion and its solution

- Suppose a low priority thread **locks** some resource shared with a high priority thread process, in a computer busy with high priority threads.
- What is the result? The **low priority thread works slowly**, which is expected since it is "low priority".
- But... if a **high priority** thread uses the shared data structure, it will **also works slowly** because it has to wait for the lock \Rightarrow **Priority inversion!**
High priority threads wait for low priority ones, just reverse of the normal case.
- Solution: **temporarily raise priority** of processes with waited lock, to that of the waiting process.
Difficulty: it is usually not immediately clear who is waiting for whom.

POS(0230A)-10.13

Shortest Job First (SJF)

An interesting special case for priority scheduling: use CPU burst length as priority, with **shorter bursts** having **higher priority** \Rightarrow SJF.

Again, can be preemptive or non-preemptive. Preemptive SJF is called **SRPT** (*Shortest Remaining Processing Time*).

An interesting property of SRPT: it **minimizes average completion time** if each thread has exactly one burst:

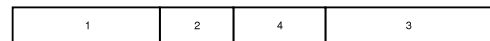
- By completely working on a shorter burst before a longer one, the shorter burst is completed earlier, at the expense that the longer burst may need more time to complete.
- But the amount of increase of turnaround time for the long burst is never more than the reduction in turnaround time for the short burst.

POS(0230A)-10.14

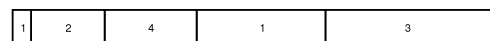
Example

Thread	Burst starts	Burst length
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Non-preemptive SJF



Preemptive SJF (SRPT)



POS(0230A)-10.15

Problems and approximations of SJF

- But... how to know the burst lengths? They are **future** information!
- General strategy: **guess** the CPU burst by the **history** of the job.
Again, guess the future using past information (remember LRU?).
- Simple solution: use the burst time of the **last burst**. E.g., if a thread has a burst time of 3ms in the last burst, we predict that next burst is 3ms.
- The guess could be incorrect, and if so we will preempt the job and call the scheduler again. E.g., if our guess is 3ms, we might give the thread a time quantum of 4ms, and reschedule if it is used up.
- But... this strategy is too easily fooled but a single short burst. Can we do better?

POS(0230A)-10.16

Better method of guessing

- Better solution: keep a **weighted average** of recent burst times.
- E.g., we might give a weight of 0.5 to last burst, 0.25 to next last burst, 0.125 to the further next burst, etc.
- If so, if the first 3 CPU bursts are 8ms, 8ms and 2ms respectively, guess for 4th burst= $8ms \cdot 0.125 + 8ms \cdot 0.25 + 2ms \cdot 0.5 = 4ms$.
- Benefit: **exceptional lengths are averaged out**, and **recent** values give **more influence** because their weight is larger.
- Difficult calculation? Luckily, a simple way can find the averages easily:
 - At the beginning, have a small guess (e.g., 0ms).
 - After each burst, add the (now known) burst length to the guess, and divide the guess by 2 (so 2nd guess is 4, 3rd is 6, 4th is 4).
Because $(0+8)/2=4$, $(4+8)/2=6$, $(6+2)/2=4$.

POS(0230A)-10.17

Mixing scheduling algorithms

We usually want different algorithm for different threads...

- Some threads are **very important**, and we want them to always be done first using **priority scheduling**. E.g., system threads that swap out memory when memory is tight.
- Some threads are **batch jobs**, i.e., the user is not actively waiting for its completion. E.g., large calculation. We want such threads to have large quantum, with low priority.
- Some threads are **time-critical** (e.g., your MP3 player), and we want to give them higher priority, with smaller quantum.
- Some threads **affects many other threads** (e.g., the X server), and we want to give them large quantum, with high priority.

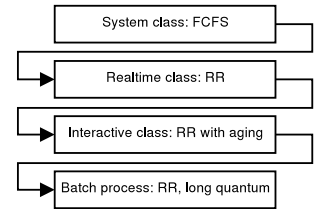
More important problem... the OS typically don't know which is which!
Except that it probably know which are system threads.

POS(0230A)-10.18

Multi-Level Queue (MLQ)

Many current Unix systems use the following idea:

- In general, the scheduler is a **fixed priority** scheduler, with higher priority threads always done before lower priority threads.
- Priority is divided into **classes**, e.g., system, real-time, interactive, etc. Each priority class uses a different scheduling strategy.
- This gives a queue structure like this...



POS(0230A)-10.19