CPU Scheduling

1DV512 – Operating Systems

Dr. Kostiantyn Kucher

kostiantyn.kucher@lnu.se

November 11, 2020

Based on the Operating System Concepts slides by Silberschatz, Galvin, and Gagne (2018)

Suggested OSC book complement: Chapter 5

Agenda

- Motivation and Introduction
- Process Scheduling Approaches
- Scheduling Multiple Threads and Multiple Processors
- Summary

Motivation

- Maximum CPU utilization obtained with multiprogramming and multitasking, rather than supporting a single process
- *CPU–I/O Burst Cycle* ⇒ process execution consists of a cycle of CPU execution and I/O wait
- CPU burst distribution is of main concern
  - On average, a large number of short CPU bursts and a small number of long CPU bursts
  - *CPU-bound* processes ⇒ spend most time doing intensive computations ("number crunching") and generate I/O requests infrequently ⇒ infrequent, but long CPU bursts
  - *I/O-bound* processes ⇒ spend most time waiting for input/output events ⇒ frequent, but short CPU bursts

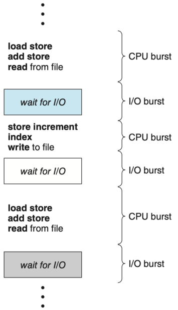


Fig. 5.1 in OSC book

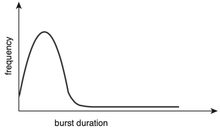


Fig. 5.2 in OSC book

CPU Scheduler

- The *CPU scheduler* selects from among the processes in the ready queue, and allocates a CPU core to one of them
  - *Non-preemptive* (or *cooperative*) scheduling ⇒ a process must voluntarily relinquish control of the CPU
  - *Preemptive* scheduling ⇒ the CPU can be taken away from a process (used by most modern OS)
- The selection of the process can be done in various ways ⇒ criteria and approaches discussed below!
- After the *scheduler* selects a process for execution, the *dispatcher* module actually provides it with control of the CPU, including:
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program
- This takes some time ⇒ *dispatch latency*

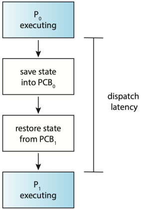


Fig. 5.3 in OSC book

Scheduling Criteria

- *CPU utilization* $\upuparrows \Rightarrow$ keep the CPU as busy as possible
- *Throughput* $\upuparrows \Rightarrow$ number of processes that complete their execution per time unit
- *Turnaround time* $\downdownarrows \Rightarrow$ amount of time to execute a particular process
- *Waiting time* $\downdownarrows \Rightarrow$ amount of time a process has been waiting in the ready queue
- *Response time* $\downdownarrows \Rightarrow$ amount of time it takes from when a request was submitted until the first response is produced

Agenda

- Motivation and Introduction
- Process Scheduling Approaches
- Scheduling Multiple Threads and Multiple Processors
- Summary

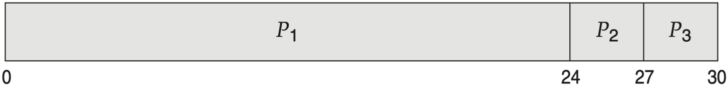
First-Come, First-Served (FCFS) Scheduling

- *First-Come, First-Served (FCFS)* scheduling ⇒ the process that requests the CPU first is allocated the CPU first; usually implemented with a FIFO queue
- Simple and straightforward approach, but can lead to long average waiting time

- Consider the processes arriving at moment 0 with the given burst time (ms):

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Assuming the order of arrival P_1, P_2, P_3:

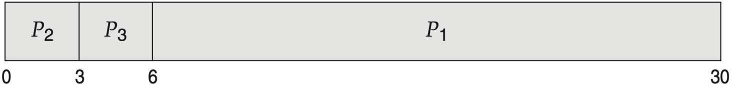


Sec. 5.3.1 in OSC book

- The waiting times ⇒ P_1: 0 ms, P_2: 24 ms, P_3: 27 ms
- The average waiting time is $(0 + 24 + 27)/3 = 17$ ms

FCFS Scheduling (cont.)

- Assuming the order of arrival P_2, P_3, P_1:



Sec. 5.3.1 in OSC book

- The average waiting time is $(6 + 0 + 3)/3 = 2$ ms
- Thus, the average waiting time may vary substantially if the processes' CPU burst times vary greatly!
- *Convoy effect* $\Rightarrow$ short processes (I/O-bound) have to wait behind a long process (CPU-bound) $\Rightarrow$ lower CPU and device utilization
- FCFS is non-preemptive $\Rightarrow$ can be troublesome for interactive multitasking systems!

Shortest Job First (SJF) Scheduling

- *Shortest Job First (SJF)* scheduling ⇒ the process with lowest expected CPU burst is selected
- If several processes have the same expected length of the next CPU burst ⇒ apply FCFS to break the tie!
- Example in the next slide

- SJF is provably optimal w.r.t. average waiting time, but there is no way to know the exact length of the next CPU burst
- Solutions: ask the user or estimate the burst length (e.g., using *exponential average* for recent and past history)

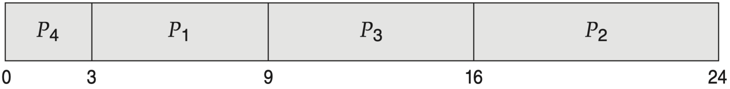
- SJF is non-preemptive, but a preemptive version exists ⇒ discussed below

SJF Scheduling (cont.)

- Consider the processes with the given burst time (ms):

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- Resulting SJF scheduling:



Sec. 5.3.2 in OSC book

- The waiting times $\Rightarrow$ P_1: 3 ms, P_2: 16 ms, P_3: 9 ms, P_4: 0 ms
- The average waiting time is $(3 + 16 + 9 + 0)/4 = 7$ ms $\Rightarrow$ would be 10.25 ms for FCFS!

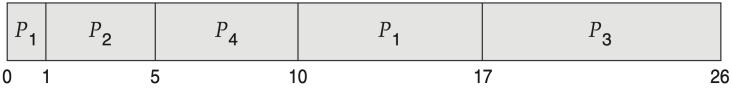
Preemptive SJF Scheduling

- *Preemptive SJF*, or *Shortest Remaining Time First* scheduling $\Rightarrow$ check the predicted next CPU burst of a newly arrived process and preempt, if necessary

- Consider the processes with the given arrival moment and burst time (ms):

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- Resulting SRTF scheduling (notice how P_1 is preempted by P_2):



Sec. 5.3.2 in OSC book

- The average waiting time is $[(0-0+10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5$ ms $\Rightarrow$ would be 7.75 ms for non-preemptive SJF!

Round Robin (RR) Scheduling

- *Round Robin (RR)* scheduling $\Rightarrow$ similar to FCFS + preemption after a predefined *time quantum* / *time slice* (typically, 10–100 ms); implemented with a circular queue
- Example in the next slide

- Timer interrupts every time quantum to schedule the next process
- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once $\Rightarrow$ no process waits more than $(n-1) \cdot q$ time units

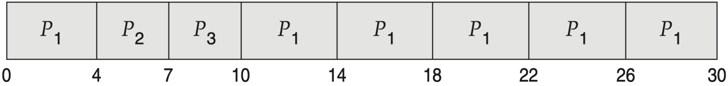
RR Scheduling (cont.)

- Consider the processes arriving at moment 0 with the given burst time (ms):

Process	Burst Time
P_1	24
P_2	3
P_3	3

Sec. 5.3.3 in OSC book

- Resulting RR scheduling (assuming time quantum of 4 ms):



Sec. 5.3.3 in OSC book

- The average waiting time is $[(0-0+10-4)+(4-0)+(7-0)]/3 = 17/3 = 5.66$ ms
- Typically, higher average turnaround than SJF, but better response

RR Scheduling Performance

- Performance of RR scheduling depends heavily on the choice of time quantum:
  - If q is extremely large $\Rightarrow$ RR is equivalent to FCFS
  - If q is extremely small (e.g., 1 ms) $\Rightarrow$ RR can result in a large number of context switches, leading to slowdowns!
  - Thus, we want the time quantum to be large with respect to the context-switch time (which is typically $<10\mu$s)
  - Also, as a rule of thumb, 80% of CPU bursts should be shorter than the time quantum

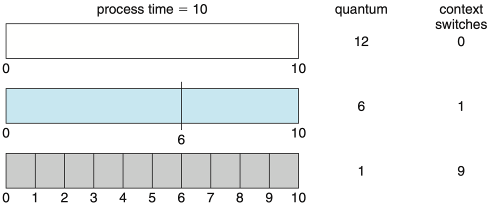


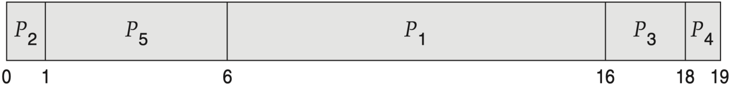
Fig. 5.5 in OSC book

Priority Scheduling

- *Priority* scheduling $\Rightarrow$ select the process with the highest priority (e.g., smallest integer number = the highest priority)
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Consider the processes arriving at moment 0 with the given burst time (ms) and priority:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Resulting priority scheduling:



Sec. 5.3.4 in OSC book

- The average waiting time is 8.2 ms

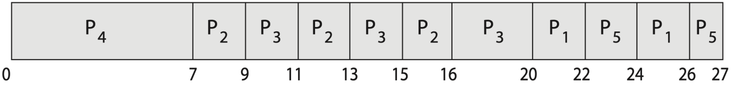
Priority Scheduling (cont.)

- *Starvation* ⇒ low-priority processes can be blocked/waiting indefinitely!
- One solution is *aging* ⇒ increase the priority of long-awaiting processes
- Another option is to combine priority scheduling with RR (for the processes with equal priorities)

- Consider the processes arriving at moment 0 with the given burst time (ms) and priority:

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

- Resulting scheduling (assuming time quantum of 2 ms):



Sec. 5.3.4 in OSC book

- Notice that P_3 has the highest priority at time 16

Multilevel Queue

- *Multilevel queue* scheduling ⇒ have separate queues for each priority and schedule the process from the suitable highest-priority queue!
- Also works well when combined with RR scheduling, as in the previous slide ⇒ in fact, each queue can use its own scheduling algorithm
- Each queue can either have absolute priority over the lower ones, or use a dedicated portion of CPU time *(time-slicing)*

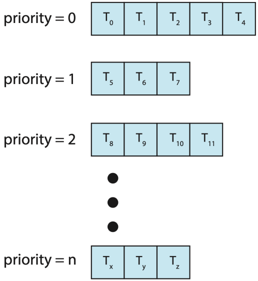


Fig. 5.7 in OSC book

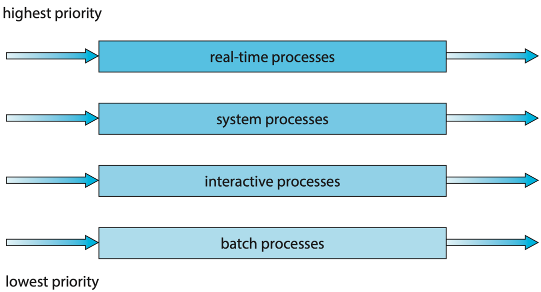


Fig. 5.8 in OSC book

Multilevel Feedback Queue

- *Multilevel feedback queue* scheduling ⇒ allow a process to move between the various queues according to the characteristics of CPU bursts
- If a process uses too much CPU time, it will be moved to a lower-priority queue
- I/O-bound and interactive processes left in the higher-priority queues
- A process that waits too long in a lower-priority queue may be moved to a higher-priority queue

- In the example on the right:
  - A process starts in the first RR queue
  - If not finished in 8 ms ⇒ moved to the second RR queue
  - If not finished in 16 ms ⇒ moved to the FCFS queue
- Overall, most general and flexible CPU scheduling approach ⇒ but also most complex to implement!

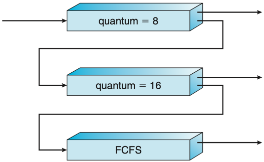
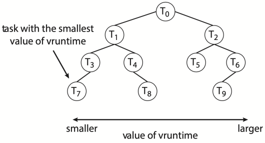


Fig. 5.9 in OSC book

Linux CFS Scheduler

- *Completely Fair Scheduler (CFS)* ⇒ maintain several *scheduling classes* with different scheduling algorithms
- The scheduler picks the highest priority task in the highest scheduling class
- Two standard scheduling classes: default and real-time
- CFS scheduler does not use fixed time quantums ⇒ instead, it allocates CPU time proportional to the priority, targeted latency, and the number of currently active tasks

- CFS scheduler maintains *virtual run time* per each task ⇒ uses a red-black tree data structure instead of a queue
- To decide next task to run, scheduler picks a task with lowest virtual run time



Sec. 5.7.1 in OSC book

- Real-time tasks ⇒ run at higher priority than normal tasks
- CFS also supports *load balancing* between multiple processing cores

Agenda

- Motivation and Introduction
- Process Scheduling Approaches
- Scheduling Multiple Threads and Multiple Processors
- Summary

Thread Scheduling

- When *kernel-level threads* are supported by OS ⇒ threads scheduled rather than processes! (e.g., Linux prefers the neutral term "task")
- *User-level* threads are managed by a thread library, and the kernel is unaware of them
- To run on a CPU, user-level threads must be mapped to an associated kernel-level thread ⇒ *many-to-one*, *many-to-many*, or *one-to-one* models can be applied
- Intermediate data structure between user- and kernel-level threads ⇒ *lightweight process (LWP)*
- *Process-contention scope (PCS)* ⇒ "competition" for the CPU time among the threads in the same process
- *System-contention scope (SCS)* ⇒ "competition" between all kernel-level threads

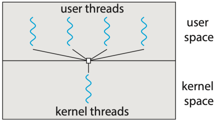


Fig. 4.7 in OSC book

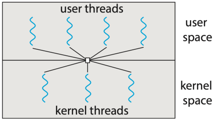


Fig. 4.9 in OSC book

Multi-Processor Scheduling

- CPU scheduling becomes more complex when multiple CPUs are available!
- Multiprocess architectures: multicore CPUs, multithreaded cores, Non-Uniform Memory Access (NUMA) systems, heterogeneous multiprocessing, . . .
- Asymmetric multiprocessing ⇒ one processor oversees the others (can lead to performance bottlenecks!)
- *Symmetric multiprocessing (SMP)* ⇒ each processor is self-scheduling:
  - All threads in one common ready queue, or
  - Each processor/core has a private queue of threads

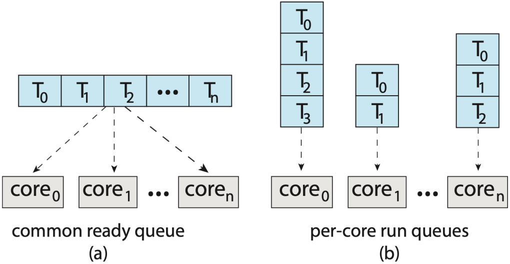


Fig. 5.11 in OSC book

Multithreaded Multicore Processors

- *Memory stall* ⇒ CPU can spend up to 50% of its time waiting for data from the memory

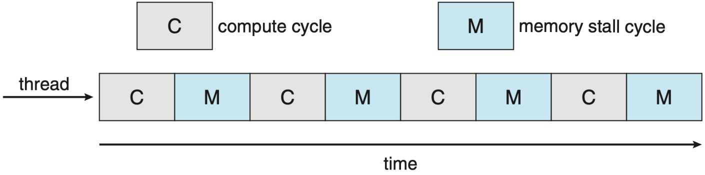


Fig. 5.12 in OSC book

- Solution ⇒ two or more *hardware threads* assigned to each CPU core

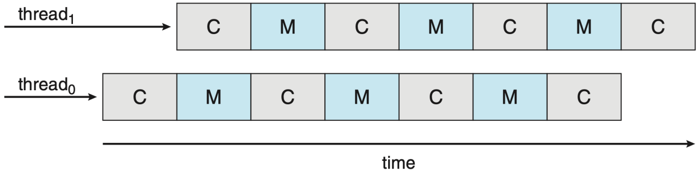


Fig. 5.13 in OSC book

Multithreaded Multicore Systems

- *Chip-multithreading (CMT)* assigns each core with multiple hardware threads (Intel refers to this as *hyperthreading*)
- On a quad-core system with 2 hardware threads per core, OS sees 8 logical processors
- A multithreaded, multicore processor actually requires two different levels of scheduling:
    1. OS deciding which software thread to run on a logical CPU ⇒ any of the algorithms discussed above!
    2. How each core decides which hardware thread to run on the physical core ⇒ RR or priority/urgency-based approach

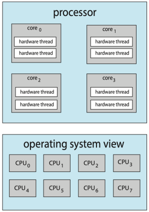


Fig. 5.14 in OSC book

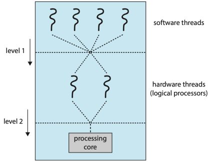


Fig. 5.15 in OSC book

Agenda

- Motivation and Introduction
- Process Scheduling Approaches
- Scheduling Multiple Threads and Multiple Processors
- Summary

Summary

- CPU scheduling is the task of selecting a waiting process from the ready queue and allocating the CPU to it
- The CPU is then allocated to the selected process by the dispatcher
- Scheduling algorithms can be either non-preemptive or preemptive (most modern OS)
- Variety of existing scheduling algorithms and strategies, from FCFS to multilevel feedback queue scheduling
- Evaluating a CPU scheduling approach can be carried out via modeling, simulation, or testing the implementation in real-world use case scenarios