CPU Scheduling

Loïc Guégan

Adapted from P. Ha @ UiT, J. Kubiatowicz @ 2010 UCB,
K. Li and J.P. Singh @ Princeton, A. S. Tanenbaum @ 2008, A. Silberschatz @ 2009

UiT The Arctic University of Norway

Spring - 2024

Outline

- What is CPU-scheduling?
- Why is CPU-scheduling needed?
- How does CPU-scheduling work?

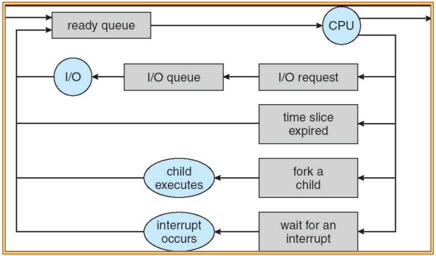
Recall: Dispatching Loop

- Conceptually, the dispatching loop of the operating system looks as follows:

```
Loop {
   RunThread();
   ChooseNextThread();
   SaveStateOfCPU(curTCB);
   LoadStateOfCPU(newTCB);
}
```

- This is an infinite loop
  - One could argue that this is all that the OS does

CPU Scheduling



- How OS decides the threads to take off from the queues?
  - One is important $\Rightarrow$ ready queue
  - Others can be scheduled as well but not the focus
- **CPU-scheduling:** deciding which threads are given access to CPU from moment to moment

Outline

- What is CPU-scheduling?
- Why is CPU-scheduling needed?
- How does CPU-scheduling work?
  - Scheduling in batch and interactive systems

Why CPU-scheduling?

- Multiple threads compete for CPU
- There aren't enough resources for all threads
- CPU time is a scarce resource

Scheduling Criteria

- Assumptions:
  - One program per user and one thread per program
  - Programs are independent
- Goals for batch and interactive systems:
  1. Maximize throughput
     - ⋆ Jobs/ hour
     - ⋆ Min overhead, max resource utilization
  2. Minimize completion time
     - ⋆ Average time from submission to completion

     } Batch systems
  3. Minimize response time
     - ⋆ Time until response (e.g. typing on keyboard)

     } Interractive systems
  4. Fairness

Outline

- What is CPU-scheduling?
- Why is CPU-scheduling needed?
- How does CPU-scheduling work?
  - Scheduling in batch and interactive systems

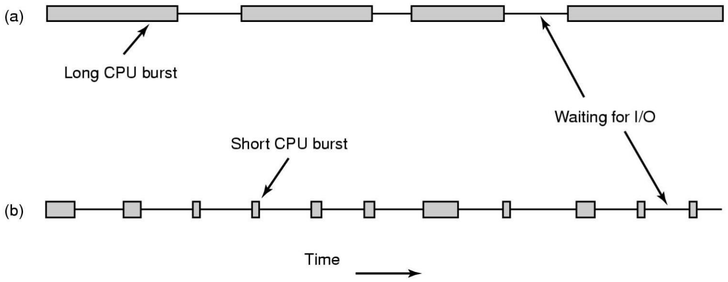
First-Come First-Serve (FCFS) Policy

- What does it mean?
  - Serve in the order of requests
  - Run to completion or until blocked or yields
- Example 1
  - P1 = 24sec, P2 = 3sec, and P3 = 3sec, submitted together
  - Average completion time = (24 + 27 + 30) / 3 = 27

- Example 2
  - Same jobs but come in different order: P2, P3 and P1
  - Average completion time =

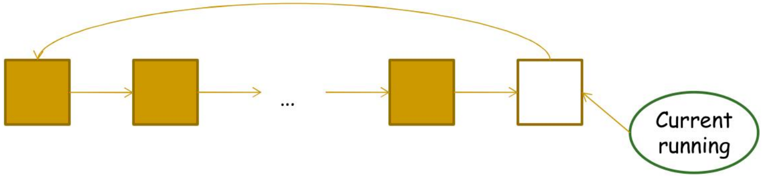
Scheduling – Process Behavior

- Bursts of CPU usage alternate with periods of waiting for I/O



(a) A CPU-bound process. (b) An I/O-bound process.

Round Robin



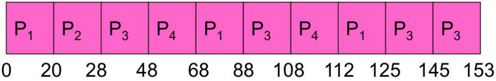
- Similar to FCFS, but add a time slice for timer interrupt
- FCFS for preemptive scheduling

Example of RR with Time Slice = 20s

- Example:

Thread	Burst Time (s)
P_1	53
P_2	8
P_3	68
P_4	24

- The Gantt chart is:



- Waiting time for

$$P_1 = (68 - 20) + (112 - 88) = 72$$
$$P_2 = (20 - 0) = 20$$
$$P_3 = (28 - 0) + (88 - 48) + (125 - 108) = 85$$
$$P_4 = (48 - 0) + (108 - 68) = 88$$

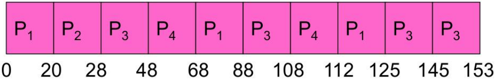
- Average waiting time $= (72 + 20 + 85 + 88)/4 = 66\frac{1}{4}$
- Average completion time $= (125 + 28 + 153 + 112)/4 = 104\frac{1}{2}$
- Thus, Round-Robin Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)

Example of RR with Time Slice = 20s

- Example:

Thread	Burst Time (s)
P_1	53
P_2	8
P_3	68
P_4	24

- The Gantt chart is:



- Waiting time for

$$P_1 = (68 - 20) + (112 - 88) = 72$$
$$P_2 = (20 - 0) = 20$$
$$P_3 = (28 - 0) + (88 - 48) + (125 - 108) = 85$$
$$P_4 = (48 - 0) + (108 - 68) = 88$$

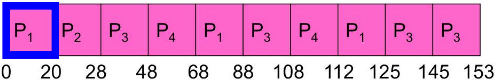
- Average waiting time $= (72 + 20 + 85 + 88)/4 = 66\frac{1}{4}$
- Average completion time $= (125 + 28 + 153 + 112)/4 = 104\frac{1}{2}$
- Thus, Round-Robin Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)

Example of RR with Time Slice = 20s

- Example:

Thread	Burst Time (s)
P_1	53
P_2	8
P_3	68
P_4	24

- The Gantt chart is:



- Waiting time for

$$P_1 = (68 - 20) + (112 - 88) = 72$$
$$P_2 = (20 - 0) = 20$$
$$P_3 = (28 - 0) + (88 - 48) + (125 - 108) = 85$$
$$P_4 = (48 - 0) + (108 - 68) = 88$$

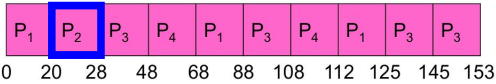
- Average waiting time $= (72 + 20 + 85 + 88)/4 = 66\frac{1}{4}$
- Average completion time $= (125 + 28 + 153 + 112)/4 = 104\frac{1}{2}$
- Thus, Round-Robin Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)

Example of RR with Time Slice = 20s

- Example:

Thread	Burst Time (s)
P_1	53
P_2	8
P_3	68
P_4	24

- The Gantt chart is:



- Waiting time for

$$P_1 = (68 - 20) + (112 - 88) = 72$$
$$P_2 = (20 - 0) = 20$$
$$P_3 = (28 - 0) + (88 - 48) + (125 - 108) = 85$$
$$P_4 = (48 - 0) + (108 - 68) = 88$$

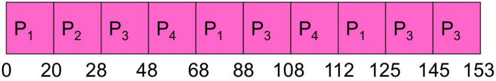
- Average waiting time $= (72 + 20 + 85 + 88)/4 = 66\frac{1}{4}$
- Average completion time $= (125 + 28 + 153 + 112)/4 = 104\frac{1}{2}$
- Thus, Round-Robin Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)

Example of RR with Time Slice = 20s

- Example:

Thread	Burst Time (s)
P_1	53
P_2	8
P_3	68
P_4	24

- The Gantt chart is:



- Waiting time for

$$\boxed{P_1 = (68 - 20) + (112 - 88) = 72}$$

$$P_2 = (20 - 0) = 20$$

$$P_3 = (28 - 0) + (88 - 48) + (125 - 108) = 85$$

$$P_4 = (48 - 0) + (108 - 68) = 88$$

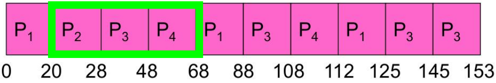
- Average waiting time $= (72 + 20 + 85 + 88)/4 = 66\frac{1}{4}$
- Average completion time $= (125 + 28 + 153 + 112)/4 = 104\frac{1}{2}$
- Thus, Round-Robin Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)

Example of RR with Time Slice = 20s

- Example:

Thread	Burst Time (s)
P_1	53
P_2	8
P_3	68
P_4	24

- The Gantt chart is:



- Waiting time for

$$\boxed{P_1 = (68 - 20) + (112 - 88) = 72}$$
$$P_2 = (20 - 0) = 20$$
$$P_3 = (28 - 0) + (88 - 48) + (125 - 108) = 85$$
$$P_4 = (48 - 0) + (108 - 68) = 88$$

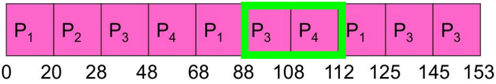
- Average waiting time $= (72 + 20 + 85 + 88)/4 = 66\frac{1}{4}$
- Average completion time $= (125 + 28 + 153 + 112)/4 = 104\frac{1}{2}$
- Thus, Round-Robin Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)

Example of RR with Time Slice = 20s

- Example:

Thread	Burst Time (s)
P_1	53
P_2	8
P_3	68
P_4	24

- The Gantt chart is:



- Waiting time for

$$\boxed{P_1 = (68 - 20) + (112 - 88) = 72}$$
$$P_2 = (20 - 0) = 20$$
$$P_3 = (28 - 0) + (88 - 48) + (125 - 108) = 85$$
$$P_4 = (48 - 0) + (108 - 68) = 88$$

- Average waiting time $= (72 + 20 + 85 + 88)/4 = 66\frac{1}{4}$
- Average completion time $= (125 + 28 + 153 + 112)/4 = 104\frac{1}{2}$
- Thus, Round-Robin Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)

Example of RR with Time Slice = 20s

- Example:

Thread	Burst Time (s)
P_1	53
P_2	8
P_3	68
P_4	24

- The Gantt chart is:

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3	
0	20	28	48	68	88	108	112	125	145	153

- Waiting time for

$$P_1 = (68 - 20) + (112 - 88) = 72$$

$$P_2 = (20 - 0) = 20$$

$$P_3 = (28 - 0) + (88 - 48) + (125 - 108) = 85$$

$$P_4 = (48 - 0) + (108 - 68) = 88$$

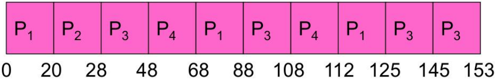
- Average waiting time = $\boxed{(72 + 20 + 85 + 88)/4 = 66\frac{1}{4}}$
- Average completion time = $(125 + 28 + 153 + 112)/4 = 104\frac{1}{2}$
- Thus, Round-Robin Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)

Example of RR with Time Slice = 20s

- Example:

Thread	Burst Time (s)
P_1	53
P_2	8
P_3	68
P_4	24

- The Gantt chart is:

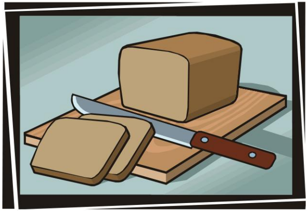


- Waiting time for

$$P_1 = (68 - 20) + (112 - 88) = 72$$
$$P_2 = (20 - 0) = 20$$
$$P_3 = (28 - 0) + (88 - 48) + (125 - 108) = 85$$
$$P_4 = (48 - 0) + (108 - 68) = 88$$

- Average waiting time $= (72 + 20 + 85 + 88)/4 = 66\frac{1}{4}$
- Average completion time $= \boxed{(125 + 28 + 153 + 112)/4 = 104\frac{1}{2}}$
- Thus, Round-Robin Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)

Round-Robin Discussion



- How do you choose time slice?
  - What if too big?
    - Response time suffers
  - What if infinite (∞)?
    - Get back FCFS
  - What if time slice too small?
    - Throughput suffers!
- Actual choices of time slice:
  - In practice, need to balance short-job performance and long-job throughput:
    - Typical time slice today is between **10ms – 100ms**
    - Typical context-switching overhead is **0.1ms – 1ms**
    - Roughly **1%** overhead due to context-switching

Comparisons between FCFS and RR

- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Simple example:
  10 jobs, each takes 100s of CPU time RR time slice of 1s.

  All jobs start at the same time.
- Completion times:

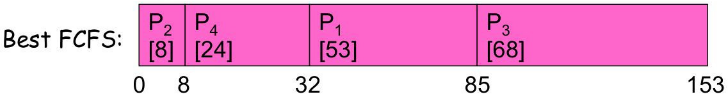
Job #	FIFO	RR
1	100	991
2	200	992
...	...	...
9	900	999
10	1000	1000

  - Both RR and FCFS finish at the same time
  - Average response time is much worse under RR!
- Cache state must be shared between all jobs with RR but can be devoted to each job with FIFO

FCFS vs. Round Robin

- FCFS Scheduling:
  - Run threads to completion in order of submission
  - Pros: Simple
  - Cons: Short jobs get stuck behind long ones
- Round-Robin Scheduling:
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs
  - Cons: Poor when jobs have same length

Earlier Example with Different Time Slices

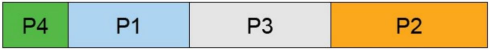


	Time slice	P_1	P_2	P_3	P_4	Average
Wait Time	Best FCFS	32	0	85	8	$31\frac{1}{4}$
	Q = 1	84	22	85	57	62
	Q = 5	82	20	85	58	$61\frac{1}{4}$
	Q = 8	80	8	85	56	$57\frac{1}{4}$
	Q = 10	82	10	85	68	$61\frac{1}{4}$
	Q = 20	72	20	85	88	$66\frac{1}{4}$
	Worst FCFS	68	145	0	121	$83\frac{1}{2}$
Completion Time	Best FCFS	85	8	153	32	$69\frac{1}{2}$
	Q = 1	137	30	153	81	$100\frac{1}{2}$
	Q = 5	135	28	153	82	$99\frac{1}{2}$
	Q = 8	133	16	153	80	$95\frac{1}{2}$
	Q = 10	135	18	153	92	$99\frac{1}{2}$
	Q = 20	125	28	153	112	$104\frac{1}{2}$
	Worst FCFS	121	153	68	145	$121\frac{3}{4}$

SJF and SRTF

Assumption: job lengths are known in advance.

- Shortest Job First (SJF)
  - Non-preemptive
- Shortest Remaining Time First (SRTF)
  - Preemptive version
- Example
  - P1 = 6sec, P2 = 8sec, P3 = 7sec, P4 = 3sec
  - All arrive at the same time



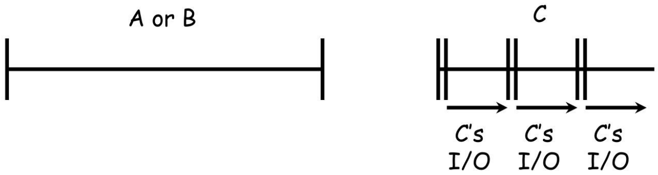
- Can you do better than SRTF in terms of average response time?

Discussion

- SJF/SRTF $\Rightarrow$ best at minimizing average response time
  - Provably optimal:
    - SJF among non-preemptive
    - SRTF among preemptive
  - SRTF at least as good as SJF $\Rightarrow$ focus on SRTF

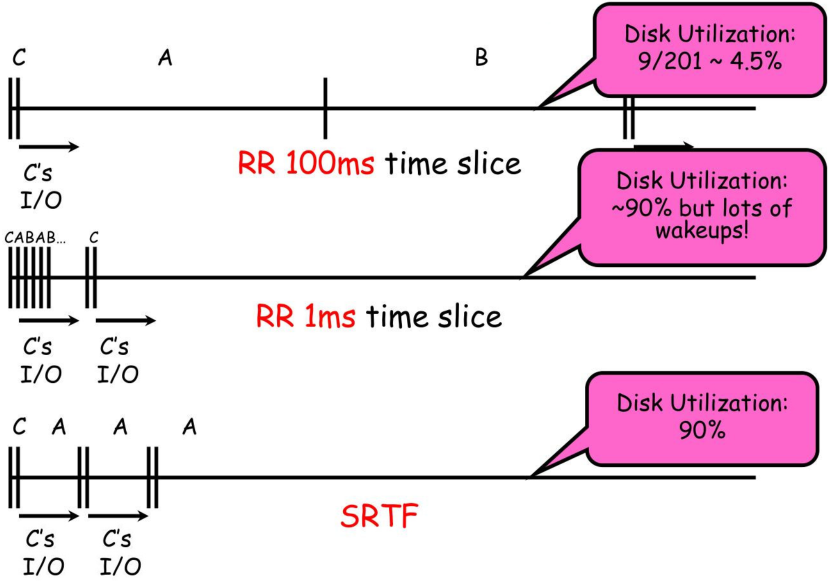
- Comparison of SRTF with FCFS and RR
  - What if all jobs the same length?
    - SRTF becomes the same as FCFS
      i.e. FCFS is best can do if all jobs the same length
  - What if jobs have varying length?
    - SRTF vs RR $\Rightarrow$ short jobs not stuck behind long ones

Example to illustrate benefits of SRTF

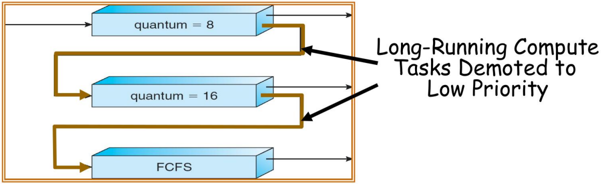


- Three jobs:
  - A,B: both CPU bound, run for 1 week
    ⇒ A or B alone use 100% of the CPU
  - C: I/O bound, loop 1ms CPU, 9ms disk I/O
    ⇒ C alone uses 90% of the disk
- With FCFS/FIFO:
  - Once A or B get in, keep CPU for two weeks
- What about RR or SRTF?

SRTF Example continued:



Multi-Level Feedback Scheduling



- A method for exploiting past behavior
  - **Multiple queues, each with different priority**
    - Higher priority queues often considered "foreground" tasks
  - **Each queue has its own scheduling algorithm**
    - e.g. foreground – RR, background – FCFS
    - Sometimes multiple RR priorities with quantum increasing exponentially (highest:1ms, next:2ms, next: 4ms, etc)
- Adjust each job's priority as follows (details vary)
  - Job starts in highest priority queue
  - If timeout expires, drop one level
  - If timeout doesn't expire, push up one level (or to top)

Scheduling Details

- Result approximates SRTF:
  - CPU bound jobs drop like a rock
  - Short-running I/O bound jobs stay near top
- Scheduling must be done between the queues
  - **Fixed priority scheduling:**
    - Serve all from highest priority, then next priority, etc.
  - **Time slice:**
    - Each queue gets a certain amount of CPU time
    - E.g., 70% to highest, 20% next, 10% lowest
- **Countermeasure:** user action that can foil intent of the OS designer
  - For multilevel feedback, put in a bunch of meaningless I/O to keep job's priority high
  - Of course, if everyone did this, wouldn't work!

Priority Scheduling

- Obvious
  - Not all threads are equal, so rank them
- The method
  - Assign each thread a priority
  - Run the thread with highest priority in the ready queue first
  - Adjust priority dynamically
- Why adjust priorities dynamically?
  - Problem: **Starvation**
  - Solution: **Aging**

Lottery Scheduling

- Motivations
  - SRTF does well with average response time, but unfair
- Lottery method
  - Give each job a number of tickets
  - Randomly pick a winning ticket
  - To approximate SRTF, give short jobs more tickets
  - To avoid starvation, give each job at least one ticket
  - Cooperative processes can exchange tickets
- Question: how do you compare this method with priority scheduling?
  - Behaves gracefully as load changes

Lottery Scheduling Example

- Assume short jobs get 10 tickets, long jobs get 1 ticket

# short jobs/ # long jobs	% of CPU each short jobs get	% of CPU each long jobs get
1/1	91%	9%
0/2	N/A	50%
2/0	50%	N/A
10/1	9.9%	0.99%
1/10	50%	5%

Summary

- Different scheduling goals
  - Depend on what systems you build
- Scheduling algorithms
  - FCFS is simple and good for jobs with the same length
  - RR is good for short jobs with different length
  - SJF and SRTF (theoretically) give the optimal average response time
  - Priority scheduling and its variations (e.g., multi-level feedback scheduling) are most common in practice
  - Lottery scheduling is flexible

References

- A. S. Tanenbaum, Modern Operating Systems.
- A. Silberschatz et. al., Operating System Concepts.

Thanks for your attention!

Questions?