Operating System: Scheduling

Alex Chi

Update: March 19, 2020

Contents

1 Basic Concepts

- Maximize CPU utilization
- CPU-I/O burst cycle - Process execution consists of a cycle of CPU execution and I/O wait (refer to figure in slides)
- CPU burst distribution

2 CPU Scheduler

- select among processes in ready queue, allocate a CPU to one of them
- takes place when a process
  1. switch from running to waiting (request I/O, syscall)
  2. switch from running to ready (time slice, interrupt)
  3. switch from waiting to ready (I/O finish)
  4. terminate
- non-preemptive scheduling: (1) and (4)
- preemptive scheduling: (2) and (3)
- personal note:
  - kernel takes charge to pause a process in preemptive scheduling
  - kernel passively waits for a process to call into kernel in non-preemptive scheduling

2.1 Example: xv6 (preemptive)

- each core runs a scheduler (thread)
- scheduler schedules kernel thread

- kernel thread give CPU to scheduler when
  - timer interrupt (time slice ends)
  - process sleep (I/O operation, sleep lock, etc.)
- scheduler give CPU to kernel thread when process is "runnable"
  - a process will be "runnable" from "waiting" after I/O completes

2.2 Q&A

1. What if a high-priority task join the queue when the CPU just finished 1 or 4Then the task will never be checked again?
   If a process is "waiting", it means that there are some background tasks to be done. Even if it has higher priority, it can't be scheduled for now.
   After it is put into ready queue, CPU will schedule higher priority task first from ready queue.
2. Does scheduler pick in random?
   In pre-defined algorithm. For example, round-robin.
3. Does the scheduler run twice in case 2?
   In case 2, time slice is used up, the process is put into ready queue.
   The scheduler only runs once.
4. Can we say that in non-preemptive scheduling, a kernel passively waits for a process to do I/O or do sys call?
   Yes. Kernel is passive in non-preemptive scheduling. "passive scheduler" or "lazy scheduler"
   From textbook: "In non-preemptive scheduling,CPU is allocated to a process until it terminates or turning to a waiting state, the CPU then be released."
   Reference: https://www.geeksforgeeks.org/preemptive-and-non-preemptive-scheduling/
5. Is CPU scheduling deterministic? (e.g. given the same input, the scheduler always schedule the processes in a same way)
   Classic and modern scheduling algorithm is all deterministic. Therefore the scheduler is fair.
   Update: However, multi-core scheduling is not deterministic. Refer to VMware's paper "Fault Tolerance Virtual Machine" and it's multi-core version.
6. Explain case 2 in CPU scheduling
   This case means that a process leaves CPU and the CPU is idle. Therefore, the scheduler is called to schedule next process.
7. Does a process access to the ready queue with a priority, or the cpu scheduler should do the choice when getting it from the queue?

Waiting queue is not just one queue. There might be multiple sub-queues. An example is multi-level feedback queue scheduling.

3 Dispatcher

- refer to figure in slides
- the dispatcher module gives control of CPU to process selected by short-term scheduler
- note: This implies that dispatcher does not run scheduling algorithm. It takes a selected process and switch to that process.
  - switching context
  - switching to user mode
  - jumping to proper location
  - example: xv6, all these three steps happen in `trap.S`.
- dispatch latency: time it takes for the dispatcher to stop one process and start another

4 Scheduling Criteria

- CPU utilization
- throughput: how many process
- turnaround time: when a process will be interrupted
- waiting time: how long does a process wait in ready queue
- response time: (time-sharing env) how long does it take to get a response
- according to *OSTEP*: higher turnaround time means lower response time

4.1 Q&A

1. What is a time-sharing environment?
   Process share CPU, and they get a fixed-length CPU.
   Other types of OS: batch, time-sharing, real-time
2. When does waiting time matter?
   Consider a HTTP server. It takes a request, and do some process, then connect to MySQL server. If waiting time is too long, then it will takes longer time to invoke I/O to database, and the latency will be high. This affects user's experience.
3. What is first response?
   In HTTP, there is TTFB (time-to-first-byte). Typically, TTFB is comparable to time for

receiving all contents. For example, it takes 100ms to resolve domain, establish TLS connection and send request, and you'll receive first byte. Then it takes 100ms to transfer a 10KB web page. This is an important criteria.

4. Is dispatcher itself a process?
   No. It could be a kernel code snippet or a kernel module. It runs in a kernel thread. In xv6, it's just a single assembly file.
5. Can we limit maximum time a process can stay on CPU to improve waiting time?
   Yes. But frequent context switch causes larger dispatcher overhead.
6. Should we count dispatch time in waiting time?
   It's much shorter than CPU burst. So we won't consider them.

5 CPU Scheduling Algorithm

5.1 First-Come, First Served

- Convoy effect: shortest process behind long process
  - CPU-bound and many I/O-bound process
- after introducing I/O burst, it is possible that CPU is not fully utilized
- tie: programmer can specify how to handle this (in this case, smaller pid)

5.2 Shortest-Job-First

- associate each process the length of its next CPU burst
- schedule the process with shortest time
- It's optimal: gives minimum average waiting time
- But it's hard to predict next length of CPU burst
- preemptive: interrupt current process and switch to the one with shorter burst time

5.3 Priority Scheduling

- a order is associated with each process
- CPU allocated to process with highest priority
- mostly, smaller number means higher priority
- SJF is a kind of priority scheduling
- issue: starvation, process of low priority have no chance to run
- solution: as time progresses priority is increased

5.4 Round Robin (RR)

- preemptive FCFS
- each process gets small unit of CPU time (time quantum q).
- performance
  - large $q \implies$ FIFO
  - small $q \implies$ q must be large with respect to context switch, or dispatcher overhead is too high.
- 8/2 rules: 80% CPU bursts should be shorter than q
- higher average turnaround than SJF, better response

5.4.1 Q&A

1. If there's only one process, is it required to do context switch?
   Yes.
2. Where does priority come from?
   Linux kernel adjusts it. And programmer can specify a number.

5.5 Multi-level Queue

- different levels

5.6 Multi-level Feedback Queue

- three queues
  - Q_1 - RR with $q = 8$
  - Q_2 - RR with $q = 16$
  - Q_3 - FCFS
  - if a program can finish in q, then it can leave queue freely
  - otherwise it will be degraded to next level of queue
- be sure you've run the examples by hand, I think this process is important to learn this scheduling algorithm

5.6.1 Q&A

1. Will this suffer from starvation?
   Yes. And most OS implements process upgrade.

6 Thread Scheduling

6.1 When thread is supported

- thread library schedules threads to run on kernel thread
  - process-contention scope (PCS)
  - priority set by programmer
- kernel thread scheduled onto available CPU is system-contention scope (SCS)

6.2 pthread Scheduling API

- refer to slides

6.3 Linux Scheduling

- (2.5) O(1) scheduler
  - preemptive priority based
  - only real-time task will degrade
  - pointer switches between expired and active array
  - fixed quantum time
- (2.6.23+) Completely Fair Scheduler (CFS)
  - proportion of CPU time
  - two scheduling class
  - quantum calculated based on nice value, calculate target latency, latency is dynamically adjusted
  - per task virtual run time `vruntime`, associated with decay factor

6.4 Other Scheduling Case Study

- refer to textbook