
CSC 369

Week 5: Scheduling
Reading: Text Chapter 2.5

Review

- The OS manages resources for processes
- A process is an instance of a program in execution
 - Includes execution state and resources
- Processes may contain multiple threads
 - Threads within a process share resources
- Synchronization is needed to coordinate sharing
 - Deadlock needs to be addressed

Process Life Cycle

- Processes repeatedly alternate between computation and I/O
 - Called CPU bursts and I/O bursts
 - Last CPU burst ends with a call to terminate the process (`_exit()` or equivalent)
 - **CPU-bound**: very long CPU bursts, infrequent I/O bursts
 - **I/O-bound**: short CPU bursts, frequent (long) I/O bursts
- During I/O bursts, CPU is not needed
 - Opportunity to execute another process!

What is processor scheduling?

- Have discussed scheduling informally already
 - E.g., Deadlock, starvation
- The allocation of processors to processes over time
 - This is the key to **multiprogramming**
 - We want to increase CPU utilization and job throughput by overlapping I/O and computation
 - Mechanisms: process states, process queues
 - **Policies:**
 - Given more than one runnable process, how do we choose which to run next?
 - When do we make this decision?

Scheduling Goals

- All systems
 - Fairness - each process receives fair share of CPU
 - Avoid starvation
 - Policy enforcement - usage policies should be met
 - Balance - all parts of the system should be busy
- Batch systems
 - Throughput - maximize jobs completed per hour
 - Turnaround time - minimize time between submission and completion
 - CPU utilization - keep the CPU busy all the time

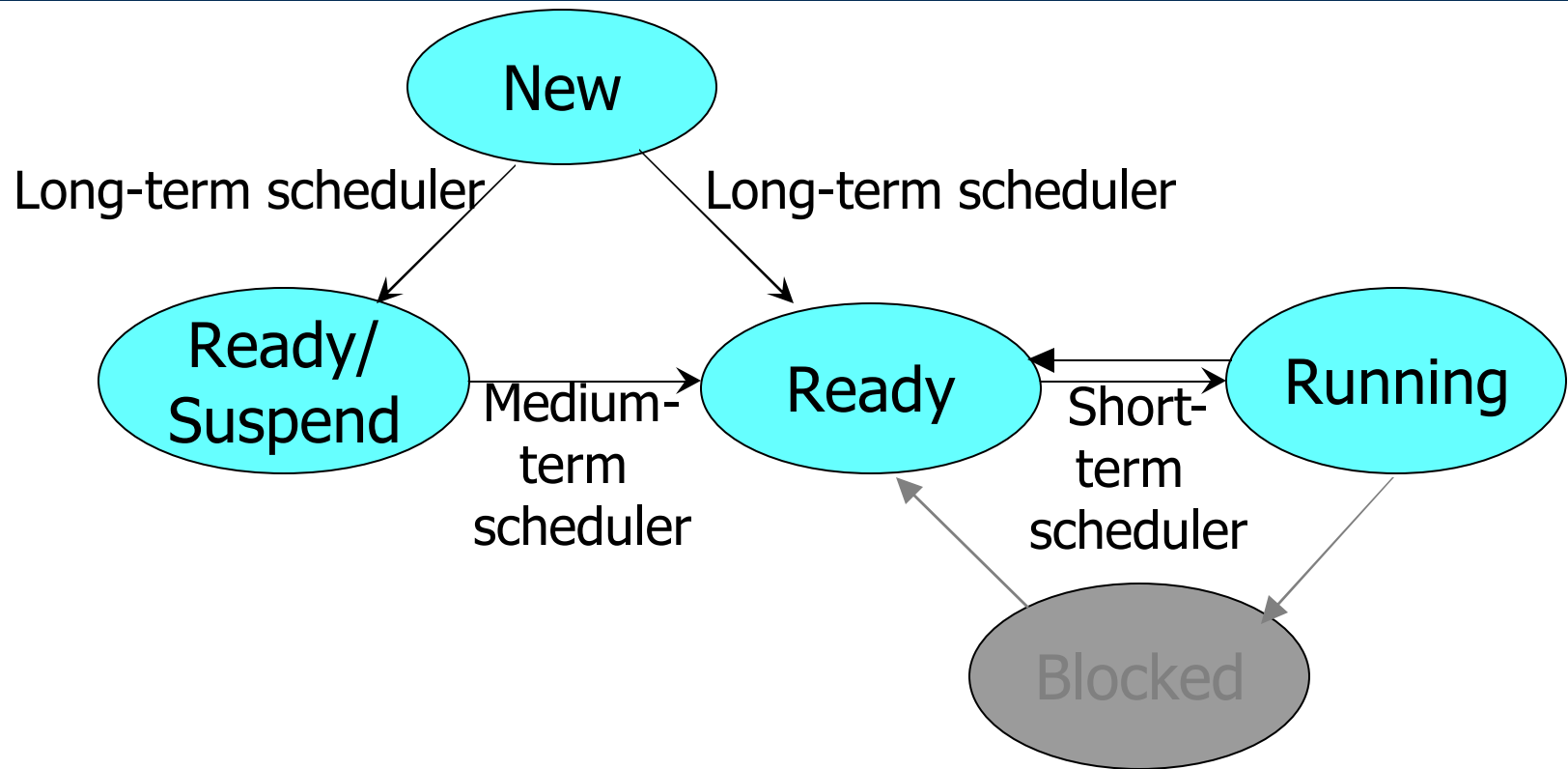
More Goals

- Interactive Systems
 - Response time - minimize time between receiving request and starting to produce output
 - Proportionality - "simple" tasks complete quickly
- Real-time systems
 - Meet deadlines
 - Predictability
- Goals sometimes conflict with each other!

Types of CPU Scheduling

- Long-term scheduling
 - aka **admission scheduling, admission control**
 - Used in batch systems, not common today
- Medium-term scheduling - happens infrequently
 - aka **memory scheduling**
 - Decides which processes are swapped out to disk
 - We'll talk about this later with memory mgmt
 - Sometimes called "long-term" with admission control ignored
- Short-term scheduler - happens frequently
 - aka **dispatching**
 - Needs to be efficient (fast context switches, fast queue manipulation)

Process State Diagram



- Dispatching a process from the ready queue is often called **context switching**

Review: What happens on dispatch?

- Save currently running process state
 - Unless the current process is exiting
- Select next process from ready queue
- Restore state of next process
 - Restore registers
 - Restore OS control structures
 - Switch to user mode
 - Set PC to next instruction in this process

When to Schedule

- When a process enters Ready state
 - I/O interrupts
 - Signals
 - Process creation (or admission)
- When the running process blocks (or exits)
 - Operating system calls (e.g., I/O)
 - Signals
- At fixed intervals
 - Clock interrupts
 - See `kern/thread/hardclock.c`

Types of Scheduling

- **Non-preemptive scheduling**
 - once the CPU has been allocated to a process, it keeps the CPU until it terminates or blocks
 - Suitable for batch scheduling
- **Preemptive scheduling**
 - CPU can be taken from a running process and allocated to another
 - Needed in interactive or real-time systems

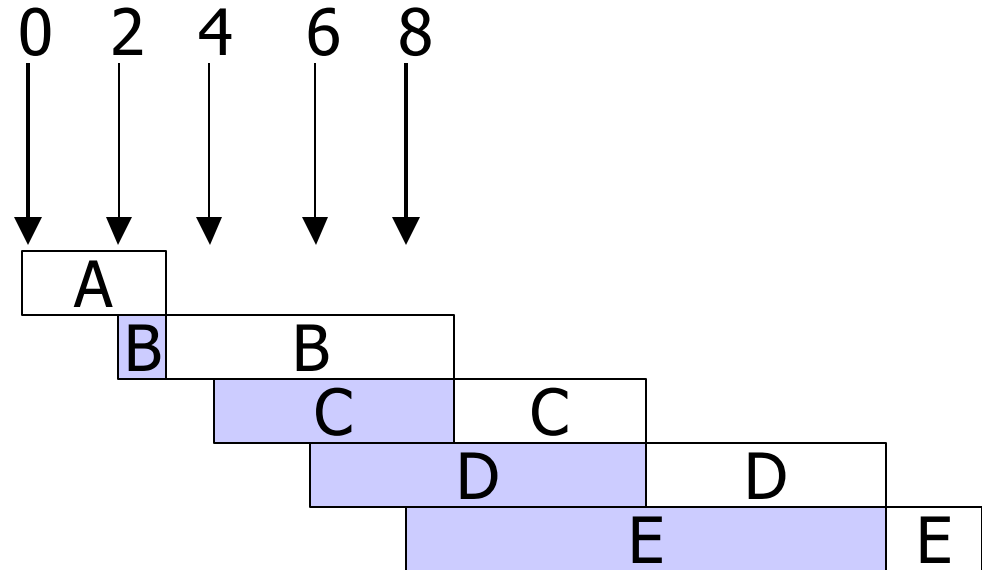
Scheduling Algorithms: FCFS

- "First come, first served"
- Non-preemptive
- Choose the process at the head of the FIFO queue of ready processes
- Average waiting time under FCFS is often quite long
 - convoy effect: all other processes wait for the one big process to release the CPU



FCFS Example

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



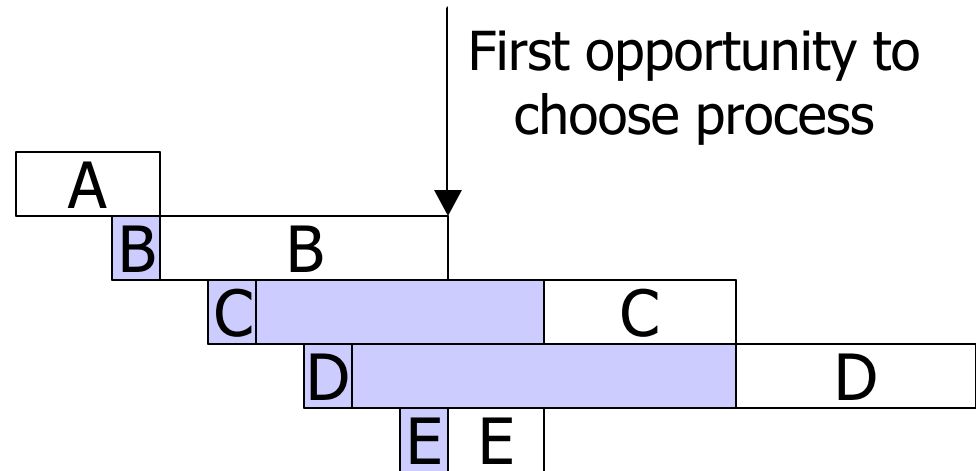
- Note E waits five times as long as it runs!
 - Total run time is 20, total wait time is 23, avg. wait is 4.6

Algorithm: Shortest-Job-First

- aka Shortest Process Next, SJF or SPN
 - Pre-emptive version is "shortest remaining time"
- Choose the process with the shortest expected processing time
 - Programmer estimate
 - History statistics
 - Can be shortest-next-CPU-burst for interactive jobs
- Provably optimal w.r.t. average wait time

Example: SJF

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



- Total run time still 20, total wait time now 17, avg. wait time now 3.4

Algorithm: Round Robin

- Designed for time-sharing systems
- Pre-emptive
- Ready queue is circular
 - Each process is allowed to run for time quantum q before being preempted and put back on queue
- Choice of quantum (aka time slice) is critical
 - as $q \rightarrow \infty$, RR \rightarrow FCFS; as $q \rightarrow 0$, RR \rightarrow processor sharing (PS)
 - we want q to be large w.r.t. the context switch time

Priority Scheduling

- A priority, p , is associated with each process
- Highest priority job is selected from Ready queue
 - Can be pre-emptive or non-preemptive
- Enforcing this policy is tricky
 - A low priority task may prevent a high priority task from making progress by holding a resource (**priority inversion**)
 - A low priority task may never get to run (**starvation**)

Priority Inversion Example

- Mars Rover Pathfinder bug
 - Shared “information bus” - essentially shared memory area
 - Mutual exclusion provided by lock on info bus
 - High priority “bus management” task moves data in/out of information bus
 - Low priority “data gathering” task writes data to the information bus
 - Medium priority, compute-bound “communications” task that does not use the information bus
 - See the problem?
 - Data gathering task locks bus and is preempted by higher priority bus management task, which blocks on the lock. If communications task becomes runnable, data gathering task can't complete and release the lock so high priority task stays blocked.

Multi-Level Queue Scheduling

- Have multiple ready queues
 - Each runnable process is on only one queue
- Processes are **permanently** assigned to a queue
 - Criteria include job class, priority, etc.
- Each queue has its own scheduling algorithm
 - Another level of scheduling decides which queue to choose next
 - Usually priority-based

Feedback Scheduling

- Adjust criteria for choosing a particular process based on past history
 - Can boost priority of processes that have waited a long time (aging)
 - Can prefer processes that do not use full quantum
 - Can boost priority following a user-input event
 - Can adjust expected next-CPU-burst
- Combine with MLQ to move processes between queues

Fair Share Scheduling

- Group processes by user or department
- Ensure that each group receives a proportional share of the CPU
 - Shares do not have to be equal
- FSS - priority of a process depends on own priority and past history of entire group
- Lottery scheduling - each group is assigned "tickets" according to its share
 - Hold a lottery to find next process to run

Unix CPU Scheduling

- interactive processes are favoured; small CPU time slices are given to processes by a priority algorithm that reduces to RR for CPU-bound jobs
- the more CPU time a process accumulates, the lower its priority becomes (negative feedback)
- process aging prevents starvation
- newer Unixes reschedule processes every 0.1 seconds and recompute priorities every second
- RR scheduling results from a timeout mechanism

More Unix Scheduling

- MLFQ with RR within each priority queue
- priority is based on process type and execution history

$$P_j(i) = \text{base}_j + [\text{CPU}_j(i-1)]/2 + \text{nice}_j$$

$$\text{CPU}_j(i) = U_j(i)/2 + \text{CPU}_j(i-1)/2$$

- $P_j(i)$: priority of process j at beginning of interval i ; lower values equal higher priorities
- base_j : base priority of process j
- $U_j(i)$: processor utilization of process j in interval i
- $\text{CPU}_j(i)$: exponentially weighted average processor utilization by process j through interval i
- nice_j : user-controllable adjustment factor

Linux (2.4) CPU Scheduling

- 2 separate process scheduling algorithms
 - time-sharing and real-time tasks
- Time-sharing: use a prioritized, credit-based algorithm
 - the STS chooses process with the most credits
 - at every timer interrupt, the running process loses one credit
 - when its credits reach 0, it is suspended
 - if no runnable processes have any credits, Linux performs a recrediting, adding credits to every process: $\text{credits} = \text{credits}/2 + \text{priority}$

Improving the 2.4 Scheduler

- Re-credit step takes time proportional to the number of processes - $O(N)$
 - For large scale systems, spend too much time making scheduling decisions
- 2.5 kernel introduced $O(1)$ scheduler
 - Each process gets a time quantum, based on its priority
 - Two arrays of runnable processes, active and expired
 - Processes are selected to run from the active array
 - When quantum exhausted, process goes on expired
 - When active is empty, swap the two arrays

Windows CPU Scheduling

- dispatcher uses a 32-level MLFQ priority scheme; the real-time class has threads with priorities 16 - 31; the variable class has threads with priorities 0 - 15
- dispatcher traverses the queues from highest to lowest until it finds a ready thread
- when a variable class thread's quantum expires, its priority is lowered; when it is released from a wait, its priority is raised

More on Win NT Scheduling

- preemptive scheduler
- real-time:
 - all threads have a fixed priority and at a given priority are in a RR queue
- variable:
 - a thread's priority begins at some initial assigned value and then may change; there is a FIFO queue at each priority level
 - if it has used up its current time quantum, NT lowers its priority; if it is waiting on an I/O event, NT raises its priority (more for interactive waits than for other types of I/O waits)