COP 4610

Operating System Principles

Scheduling

1

Objectives

- To introduce **CPU scheduling**, which is the basis for multi-programmed and multi-tasking systems
- To describe various CPU-scheduling **algorithms**
- To discuss **evaluation criteria** for selecting a CPU-scheduling algorithm for a particular system
- To examine the **scheduling algorithms** of several operating systems

2

Scheduling: Overview

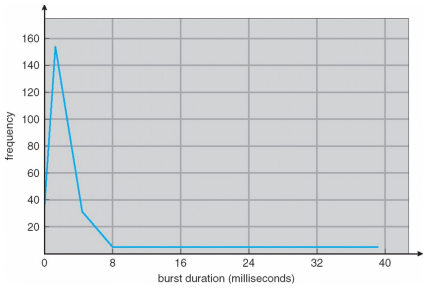
Whenever we need to decide which process to run next, we invoke the **scheduler**:

- A process **terminates**
- A process **blocks**
- A **timer interrupt** (preemptive multitasking)

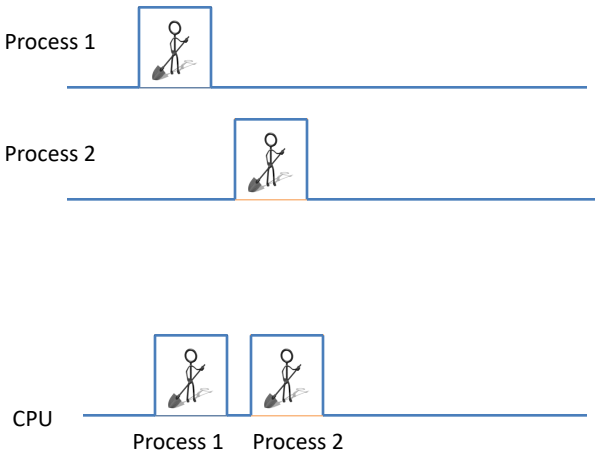
The decision-making process is called a **scheduling policy** or **discipline**.

Burstiness

- Computation and I/O tend to be **bursty**
  - Read some data
  - Compute a bunch
  - Write some data
  - Repeat

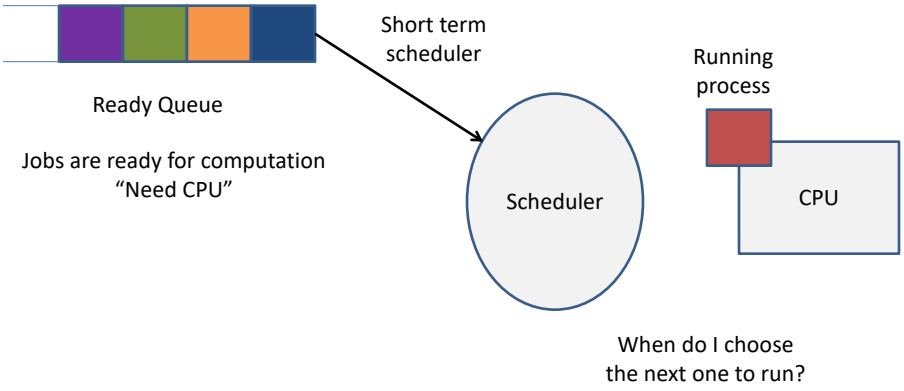


"Keep CPU busy"



5

Scheduling



6

Scheduling Choices

- Non-preemptive (voluntarily):
  - Process **yields**
  - Process goes from running to waiting state
  - Process terminates
- Preemptive (forced, can happen any time):
  - OS forces process from running to ready state

7

Dispatcher

- Mechanism that gives control of the CPU to selected process; includes:
  - Context switch
    - Save/restore stack, registers, …
  - Switch back to user mode
  - Resume PC for process

Dispatch Latency: Time it takes to stop one process and swap to another

8

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible ("how busy is the CPU")
- **Throughput** – # of processes that complete their execution per time unit ("how much work is getting done")
- **Turnaround time** – amount of time to execute a particular process ("how long does it take to execute a process")
- **Waiting time** – amount of time a process has been waiting in the READY QUEUE (RUNQUEUE)
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced

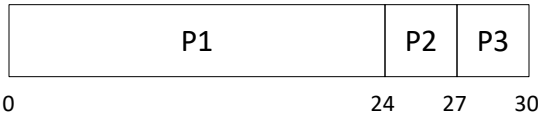
9

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

Computation Time

- Suppose that the processes arrive in the order P_1 , P_2 , P_3 and ready for execution at time 0



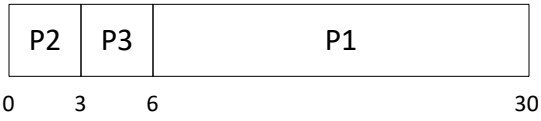
Waiting Time
P_1 = 0; P_2 = 24; P_3 = 27

Average Waiting Time
(0 + 24 + 27)/3 = 17

10

Different Order

- Suppose that the processes arrive in the order P_2 , P_3 , P_1 (ready for execution at time 0)



Waiting Time
P_1 = 6; P_2 = 0; P_3 = 3

Average Waiting Time
(6 + 0 + 3)/3 = 3

Convoy Effect

FCFS (FIFO)

- Very simple (add processes to end of runqueue, take processes from beginning of queue)
- Note: processes returning from waitqueues always go to the back of the runqueue!
- **NON-PREEMPTIVE** (time-sharing/interactive?)

Shortest-Job-First (SJF) Scheduling

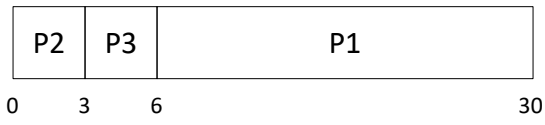
- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is **optimal** – gives **minimum average waiting time** for a given set of processes
  - The **difficulty** is knowing the length of the next CPU request

Revisiting with SJF

Process	Burst Time
P_1	24
P_2	3
P_3	3

Computation Time

- Suppose that the processes arrive in the order P_1 , P_2 , P_3 (all in queue at time 0)



Waiting Time
P_1 = 6; P_2 = 0; P_3 = 3

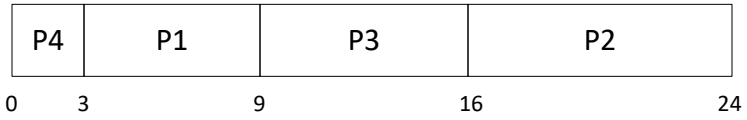
Average Waiting Time
(6 + 0 + 3)/3 = 3

Shortest-Job First

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Waiting Time
P_1 = 3; P_2 = 16; P_3 = 9

Average Waiting Time
(3 + 16 + 9 + 0)/4 = 7



Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n.$$

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
4. Define :

Also called EWMA **E**xponential **W**eighted **M**oving **A**verage

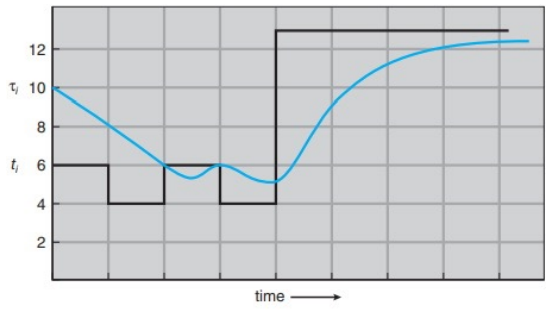
- Commonly, α set to ½

Exponential Averaging

- $\alpha = 0$? $\alpha = 1$?

$$\tau_{n+1} = \alpha^* t_n + (1-\alpha)\,\alpha^* t_{n-1} + (1-\alpha)^2 * \alpha t_{n-2}$$
$$\ldots + (1-\alpha)^j\,\alpha t_{n-j} + \ldots$$
$$\ldots + (1-\alpha)^{n+1}\tau_0$$

Prediction of the Length of the Next CPU Burst



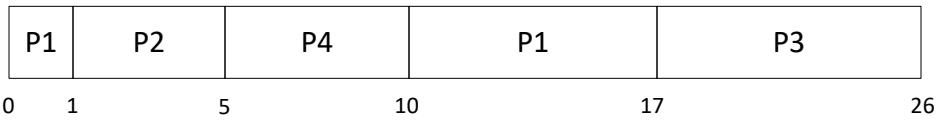
CPU burst (t_i)	6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Figure 5.4 Prediction of the length of the next CPU burst.

Shortest Job First (SJF)

- **Non-preemptive!**
- Preemptive version called **shortest-remaining-time-first**

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



0 1 5 10 17 26

19

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Non-preemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- **Fixed priorities:** do not change over time
- **Dynamic priorities:** can change over time
- Problem: **Starvation** – low priority processes may never execute
- Solution: **Aging** – as time progresses increase the priority of the process

20

Example of Priority Scheduling

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

- Low number = high priority
- Dynamic versus static/fixed priority

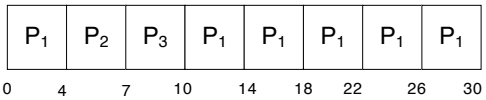
Round Robin (RR)

- Switch between processes at a time interval
  - **Time quantum, q**
  - 10-100 ms
  - **Preemptive**
- What does it mean?
  - N tasks?
  - Maximum wait time
- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

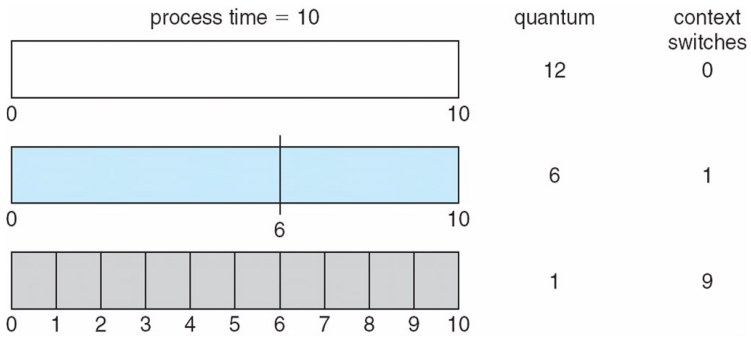
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

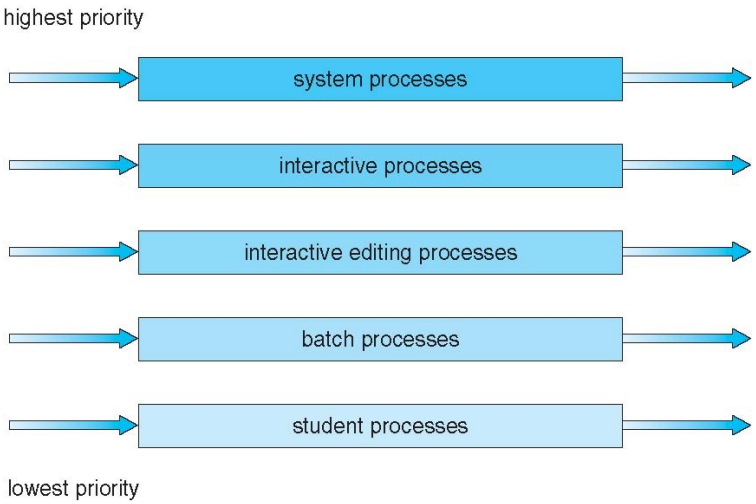
Time Quantum and Context Switch Time



Multilevel Queue

- Ready queue is partitioned into separate queues, e.g.:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of **starvation**!
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR & 20% to background in FCFS

Multilevel Queue Scheduling

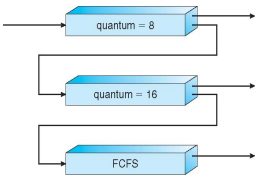


Multilevel Feedback Queue

- A process can move between the various queues; **aging** can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS



- Scheduling
  - A new job enters queue Q_0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2

Thread Scheduling

- Distinction between user-level and kernel-level threads
- Kernel-level: threads scheduled, not processes
  - **system-contention scope (SCS)** – competition among all threads in system
- User-level: thread library schedules user-level threads to run on "LWP" ("lightweight process")
  - called **process-contention scope (PCS)** since scheduling competition is within the process
  - typically done via priority set by programmer

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
         fprintf(stderr, "Illegal scope value.\n");
   }
```


Pthread Scheduling API

```
   /* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);
   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */
   pthread_exit(0);
}
```

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
  - Currently, most common
- **Processor affinity** – process has affinity for processor on which it is currently running
  - **soft affinity**
  - **hard affinity**
  - **"migration":** process changes processor

33

Load Balancing

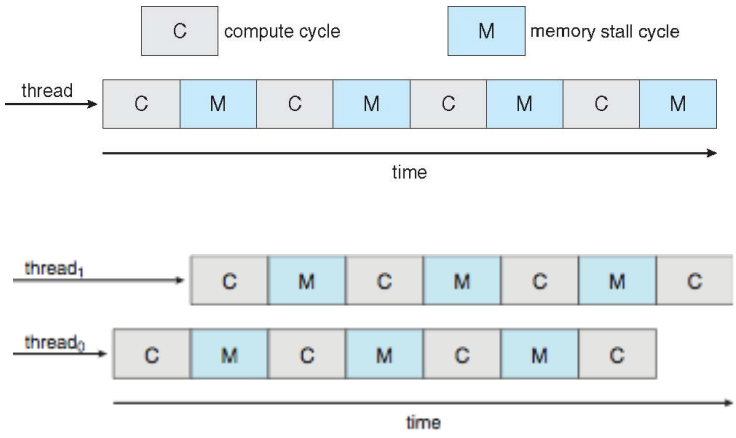
- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if imbalanced found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor

34

Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens

Multithreaded Multicore System



Linux Scheduling Through Version 2.5

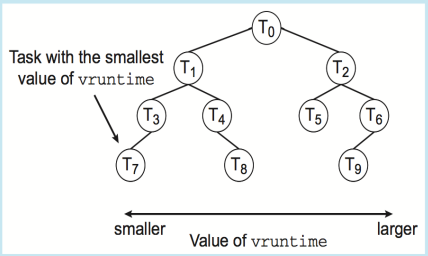
- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm
- Version 2.5 moved to constant order *O*(1) scheduling time
  - Preemptive, priority based
  - Two priority ranges: time-sharing and real-time
  - **Real-time** range from 0 to 99 and **nice** value from 100 to 140
  - Map into global priority with numerically lower values indicating higher priority
  - **Higher priority gets larger q**
  - Task run-able as long as time left in time slice (**active**)
  - If no time left (**expired**), not run-able until all other tasks use their slices
  - All run-able tasks tracked in per-CPU **runqueue** data structure
    - Two priority arrays (active, expired)
    - Tasks indexed by priority
    - When no more active, arrays are exchanged
  - Worked well, but poor response times for interactive processes

Linux Scheduling in Version 2.6.23 +

Completely Fair Scheduler (CFS)

- **Scheduling classes**
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - Rather than quantum based on fixed time allotments, based on proportion of CPU time
  - 2 scheduling classes included, others can be added
    1. default
    2. real-time
- Quantum calculated based on **nice value** from -20 to +19
  - Lower value is higher priority
  - Calculates **target latency** – interval of time during which task should run at least once
  - Target latency can increase if say number of active tasks increases
- CFS scheduler maintains per task **virtual run time** in variable **`vruntime`**
  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with lowest virtual run time

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(lg\,N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.