CS307 Operating Systems

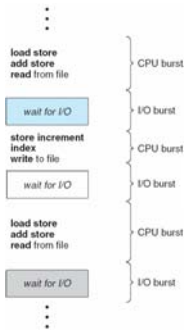
CPU Scheduling

Fan Wu
Department of Computer Science and Engineering
Shanghai Jiao Tong University
Spring 2020

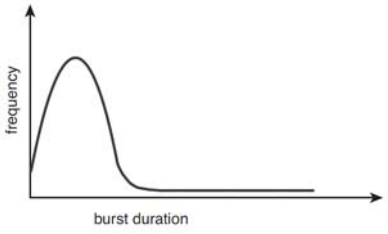
Basic Concepts

- Maximize CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait
- **CPU burst** distribution

Alternating Sequence of CPU and I/O Bursts



Histogram of CPU-burst Times



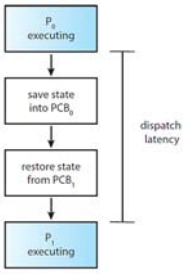
Histogram of CPU-burst durations.

CPU Scheduler

- Selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready state
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**, scheduling under 2 and 3 is **preemptive**

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler. This involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running



Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process. The interval from the time of submission of a process to the time of completion is the turnaround time.
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

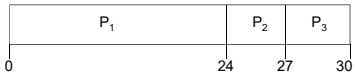
CPU Scheduling Algorithms

- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-First (SJF) Scheduling
- Priority Scheduling (PS)
- Round-Robin Scheduling (RR)
- Multilevel Queue Scheduling
- Multilevel Feedback Queue Scheduling

First-Come, First-Served (FCFS) Scheduling

Process	CPU Burst	*Arrival* Time
P_1	24	0
P_2	3	1
P_3	3	2

- The **Gantt Chart** for the schedule is:

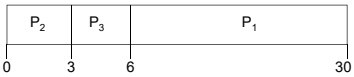


- Waiting time for P_1 = 0; P_2 = 24-1=23; P_3 = 27-2=25
  Average waiting time: (0 + 23 + 25)/3 = 16
- Turnaround time for P_1 = 24; P_2 = 27-1=26; P_3 = 30-2=28
  Average turnaround time: (24 + 26 + 28)/3 = 26

FCFS Scheduling (Cont.)

Process	CPU Burst	*Arrival* Time
P_1	24	2
P_2	3	0
P_3	3	1

- The Gantt chart for the schedule is:

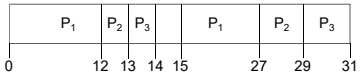


- Waiting time for P_1 = 4; P_2 = 0, P_3 = 2
- Average waiting time: (4 + 0 + 2)/3 = 2
- Much better than previous case
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

FCFS Scheduling (Cont.)

Process	CPU Burst	I/O Burst	CPU Burst	*Arrival* Time
P_1	12	3	12	0
P_2	1	2	2	1
P_3	1	2	2	2

- The **Gantt Chart** for the schedule is:



- Waiting time for
  - P_1 = 15-12-3=0
  - P_2 = (12-1)+(27-13-2)=23
  - P_3 = (13-2)+(29-14-2)=24
- Turnaround time for P_1 = 27; P_2 = 29-1=28; P_3 = 31-2=29
- CPU utilization 30/31 = 96.77%

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

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 moving average

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

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

- Commonly, α is set to ½

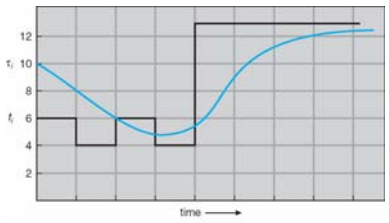
Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

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

  - Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Prediction of the Length of the Next CPU Burst

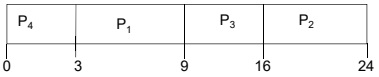


CPU burst (t_i)	6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



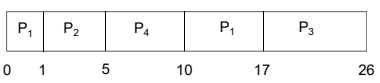
- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Example of Shortest-remaining-time-first

- We now add the concepts of varying arrival times and preemption to the analysis

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

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5

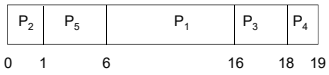
Priority Scheduling

- 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
  - Nonpreemptive
- SJF is priority scheduling, where priority is the inverse of predicted next CPU burst time
- Problem: **Starvation** – low priority processes may never execute
- Solution: **Aging** – as time progresses increase the priority of the process

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

- Priority scheduling Gantt Chart



- Average waiting time = 8.2 msec

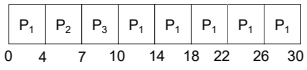
Round Robin (RR)

- Round Robin (RR) is similar to FCFS scheduling, but preemption is added to switch between processes.
- Each process gets a small unit of CPU time (**time quantum** q), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- 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. No process waits more than (*n*-1)*q* time units before next execution.
- Timer interrupts every quantum to schedule next process
- Performance
  - *q* large ⇒ FIFO
  - *q* small ⇒ *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

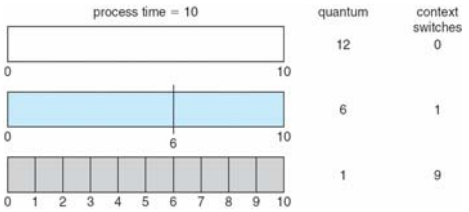
- The Gantt chart is:



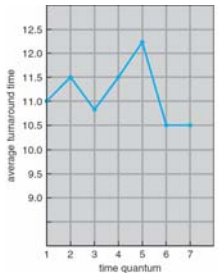
- Typically, higher average turnaround than SJF, but better *response*
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec
- What's the number of context switch?

Time Quantum and Context Switch Time

- Context switching



Turnaround Time Varies With The Time Quantum



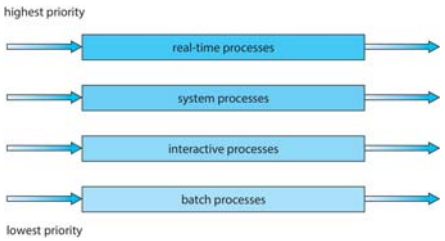
process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should be shorter than q

Multilevel Queue

- Ready queue is partitioned into separate queues, e.g.:
  - foreground (interactive)
  - background (batch)
- Process joins 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, e.g., 80% to foreground in RR, 20% to background in FCFS

Multilevel Queue Scheduling

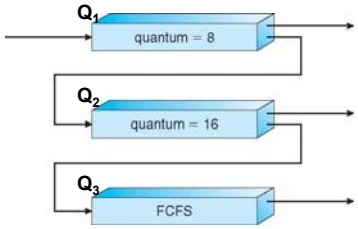


Multilevel Feedback Queue

- 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_1 – RR with time quantum 8 milliseconds
  - Q_2 – RR with time quantum 16 milliseconds
  - Q_3 – FCFS



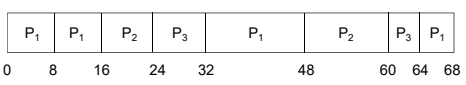
Multilevel Feedback Queue

- Scheduling
  - A new job enters queue Q_1 which is served FCFS/RR
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_2
  - At Q_2 job is again served FCFS/RR and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_3
  - If a process does not use up its quantum in the current level, it will keep its current queuing level and be put into the end of the queue. Then, it can still get the same amount of quantum (not remaining quantum) next time when it is picked.

Example of Using Multilevel Feedback Queue

Process	*Arrival* Time	Burst Time
P_1	0	36
P_2	16	20
P_3	20	12

- The Gantt chart is:



Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on kernel-level threads
  - Known as **process-contention scope (PCS)** since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
    - Schedules user-level threads onto available LWPs
    - Number of LWPs is maintained by the thread library
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
    - Creates and binds an LWP for each user-level thread
    - In fact, implements the one-to-one mapping
- 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;
    pthread_t tid[NUM THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* set the scheduling algorithm to PROCESS or SYSTEM */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], &attr, runner,NULL);
```


Pthread Scheduling API

```
    /* 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)
{
    printf("I am a thread\n");
    pthread exit(0);
}
```

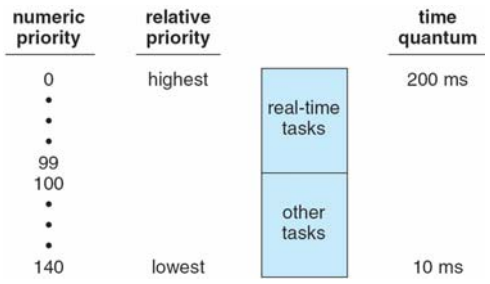

Operating System Examples

- Linux scheduling
- Solaris scheduling
- Windows XP scheduling

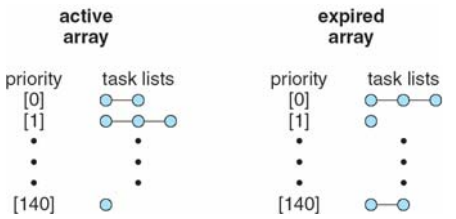
Linux Scheduling

- Constant order *O*(1) scheduling time (Version 2.5)
- **Preemptive, priority based**
- Two priority ranges: real-time and time-sharing
- **Real-time** range from 0 to 99 with **nice** value from -20 to 19, which maps to global priority from 100 to 139
- Map into global priority with numerically **lower values indicating higher priority**
- Higher priority gets larger time quantum
- 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

Priorities and Time-slice length



List of Tasks Indexed According to Priorities



Linux Scheduling (Cont.)

- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities
- All other tasks dynamic based on *nice* value plus or minus 5
  - Interactivity of task determines plus or minus
    - More interactive -> more minus
  - Priority recalculated when task expired
  - This exchanging arrays implements adjusted priorities

Linux Scheduling in Version 2.6.23 +

- ***Completely Fair Scheduler*** (CFS) -- *O*(*log* N)
- **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
  - Two scheduling classes included, others can be added
    1. default
    2. real-time

Linux Scheduling in Version 2.6.23 + (Cont.)

- 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

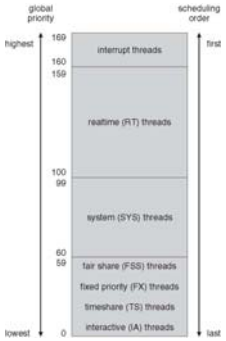
Solaris

- Priority-based scheduling
- Six classes available
  - Time sharing (default)
  - Interactive
  - Real time
  - System
  - Fair Share
  - Fixed priority
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is multi-level feedback queue
  - Loadable table configurable by sysadmin

Solaris Dispatch Table

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Solaris Scheduling



Solaris Scheduling (Cont.)

- Scheduler converts class-specific priorities into a per-thread global priority
  - Thread with highest priority runs next
  - Runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
  - Multiple threads at same priority selected via RR

Windows Scheduling

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- *Dispatcher* is scheduler
- Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- Real-time threads can preempt non-real-time
- 32-level priority scheme
- **Variable class** is 1-15, **real-time class** is 16-31
- Priority 0 is memory-management thread
- Queue for each priority
- If no run-able thread, runs **idle thread**

Windows Priority Classes

- Win32 API identifies several priority classes to which a process can belong
  - REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS,NORMAL_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS
  - All are variable except REALTIME
- A thread within a given priority class has a relative priority
  - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE
- Priority class and relative priority combine to give numeric priority
- Base priority is NORMAL within the class
- If quantum expires, priority lowered, but never below base
- If wait occurs, priority boosted depending on what was waited for
- Foreground window given 3x priority boost

Windows XP Priorities

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Pop Quiz

Process	Burst Time	Priority	Arrival Time
P1	10	3	0
P2	1	1	1
P3	4	3	3
P4	2	4	4
P5	5	2	5

1. Draw **Gantt charts** to illustrate the execution of the processes using the following scheduling algorithm:

   (1) FCFS, (2) nonpreemptive SJF, (3) preemptive SJF, (4) nonpreemptive priority, (5) preemptive priority, and (6) RR with time quantum=2
2. Calculate the average turnaround time when using each of the above scheduling algorithms
3. Count the number of context switches when using each of the above scheduling algorithms

Homework

- Reading
  - Chapter 5
- Exercise
  - See course website