Processes - Part III

Amir H. Payberah
payberah@kth.se
2022

CPU Scheduling

CPU Scheduling

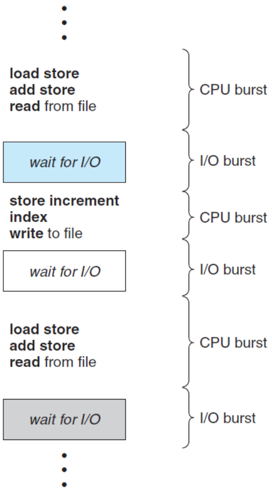
- CPU scheduling is the basis of multiprogrammed OSs.
- By switching the CPU among processes, the OS makes the computer more productive.

Basic Concepts

- In a single-processor system, only one process can run at a time.
- Others must wait until the CPU is free and can be rescheduled.
- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.

Basic Concepts

- CPU-I/O burst cycle: process execution consists of a cycle of CPU execution and I/O wait.
- CPU burst followed by I/O burst.
- CPU burst distribution is of main concern.



CPU Scheduler

- CPU scheduler selects from among the processes in ready queue, and allocates the CPU to one of them.
- CPU scheduling decisions may take place when a process:
  1. Terminates.
  2. Switches from running to waiting (e.g., an I/O request).
  3. Switches from running to ready (e.g., interrupt).
  4. Switches from waiting to ready (e.g., I/O completion).
- For situations 1 and 2, there is no scheduling choice, as a new process must be selected for execution (non-preemptive).
- But, There is a choice, for situations 3 and 4 (preemptive).

Scheduling Criteria

- Different CPU-scheduling algorithms have different properties.
- CPU utilization: keep the CPU as busy as possible (Max).
- Throughput: # of completed processes per time unit (Max).
- Turnaround time: amount of time to execute a particular process (Min).
- Waiting time: amount of time a process has been waiting in the ready queue (Min).
- Response time: amount of time it takes from when a request was submitted until the first response is produced (Min).

Scheduling Algorithms

Scheduling Algorithms

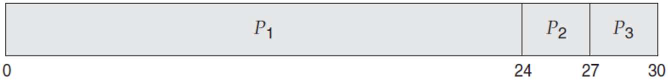
- First-Come, First-Served Scheduling
- Shortest-Job-First Scheduling
- Priority Scheduling
- Round-Robin Scheduling
- Multilevel Queue Scheduling

First-Come, First-Served (FCFS) Scheduling

FCFS Scheduling (1/2)

- Suppose that the processes arrive in the order: P_1, P_2, P_3

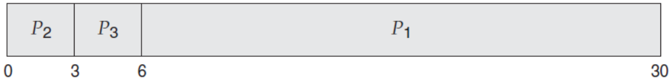
Process	Burst Time
P_1	24
P_2	3
P_3	3



- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $\frac{0+24+27}{3} = 17$
- FCFS scheduling is non-preemptive: process keeps the CPU until it releases the CPU (either by terminating or by requesting I/O).
- Convoy effect: all the other processes wait for the one big process to get off the CPU.

FCFS Scheduling (2/2)

- Suppose that the processes arrive in the order: P_2, P_3, P_1



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $\frac{6+0+3}{3} = 3$
- Much better than previous case.

Shortest-Job-First (SJF) Scheduling

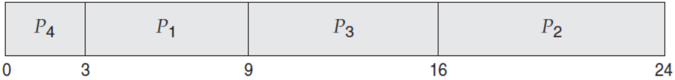
SJF Scheduling (1/2)

- 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.

SJF Scheduling (2/2)

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- Select the processes according to their burst time (from shorter to longer).



- Waiting time for $P_1 = 3$; $P_2 = 16$; $P_3 = 9$, $P_4 = 0$
- Average waiting time: $\frac{3+16+9+0}{4} = 7$

Determining Length of Next CPU Burst

- Estimate the length, and pick process with shortest predicted next CPU burst.
- The next CPU burst
  1. t_n = actual length of n^{th} CPU burst
  2. τ_{n+1} = predeicted value for the next CPU burst
  3. $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$, where $0 \leq \alpha \leq 1$
- $\alpha = 0$ then $\tau_{n+1} = \tau_n$
- $\alpha = 1$ then $\tau_{n+1} = t_n$
- Commonly, α set to $\frac{1}{2}$

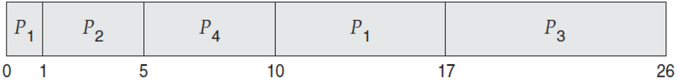
Preemptive SJF

- The SJF algorithm can be either preemptive or non-preemptive.
- Preemptive version called shortest-remaining-time-first

Example of Shortest-Remaining-Time-First

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

- Now we add the concepts of varying arrival times and preemption to the analysis.



- Average waiting time: $\frac{(10-1)+(1-1)+(17-2)+(5-3)}{4} = \frac{26}{4} = 6.5$

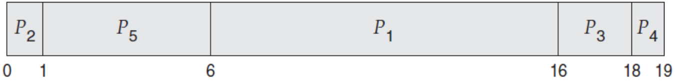
Priority Scheduling

Priority Scheduling (1/2)

- 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 and non-preemptive
- 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

Priority Scheduling (2/2)

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



- Average waiting time: $\frac{0+1+6+16+18}{5} = 8.2$

Round-Robin (RR) Scheduling

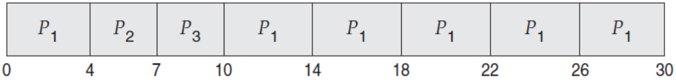
RR Scheduling (1/2)

- 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.

RR Scheduling (2/2)

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Time quantum $q = 4$



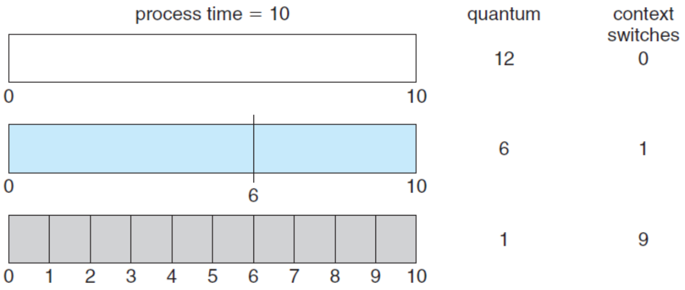
- Average waiting time: $\frac{(10-4)+4+7}{3} = 5.66$

Time Quantum and Context Switch Time (1/2)

- Timer interrupts every quantum to schedule next process.
- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high.

Time Quantum and Context Switch Time (2/2)

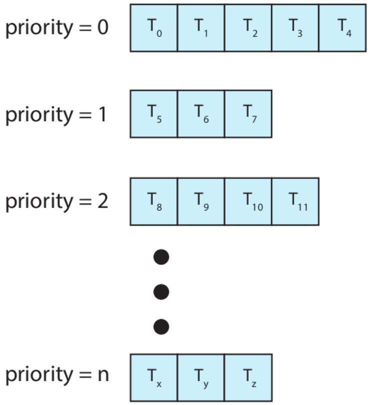
- q should be large compared to context switch time.



Multilevel Queue Scheduling

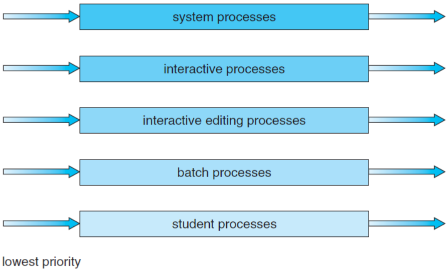
Multilevel Queue Scheduling (1/2)

- Ready queue consists of multiple queues.

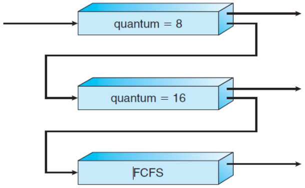


Multilevel Queue Scheduling (2/2)

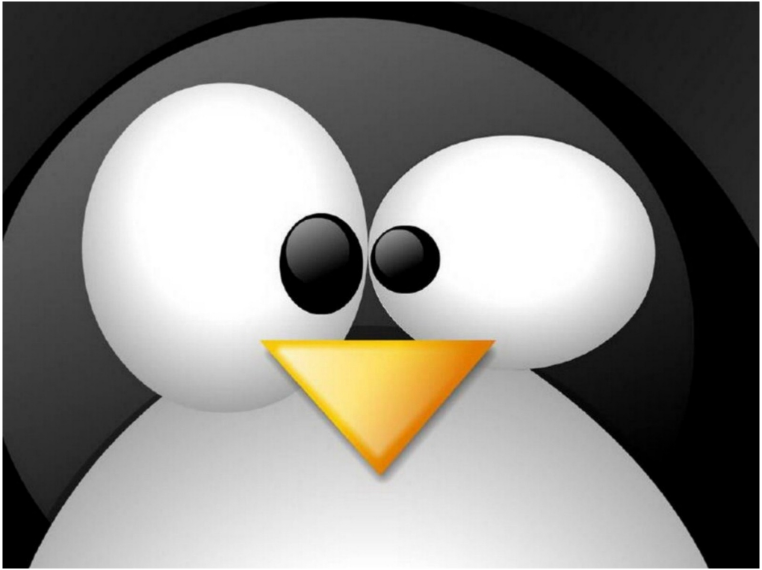
- A process can move between the various queues.
- Multilevel queue scheduler defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine which queue a process will enter when that process needs service
  - Scheduling among the queues



Multilevel Queue Scheduling - Example

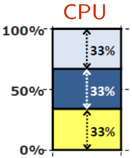


- For example, three queues:
  - Q_0: RR with time quantum 8 milliseconds
  - Q_1: RR time quantum 16 milliseconds
  - Q_2: FCFS
- A new job enters queue Q_0 which is served RR:
  - When it gains CPU, job receives 8 milliseconds.
  - If it does not finish in 8 milliseconds, job is moved to queue Q_1.
- At Q_1 job is again served RR and receives 16 additional milliseconds.
  - If it still does not complete, it is preempted and moved to queue Q_2.

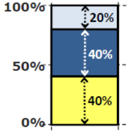


Linux Scheduling (1/2)

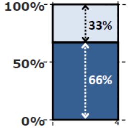
- Completely Fair Scheduler (CFS)
- n users want to share a resource, e.g., CPU.
  - Solution: allocate each $\frac{1}{n}$ of the shared resource.



- Generalized by max-min fairness.
  - Handles if a user wants less than its fair share.
  - E.g., user 1 wants no more than 20%.

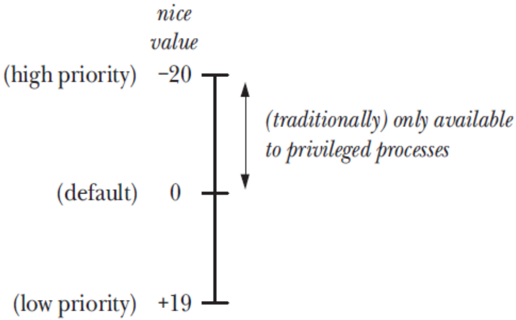


- Generalized by weighted max-min fairness.
  - Give weights to users according to importance.
  - E.g., user 1 gets weight 1, user 2 weight 2.



Linux Scheduling (2/2)

- Quantum calculated based on nice value from -20 to +19.



Modifying the Nice Value

- nice() increments a process's nice value by inc and returns the newly updated value.
- Only processes owned by root may provide a negative value for inc.

```
#include <unistd.h>

int nice(int inc);
```

Retrieving and Modifying Priorities

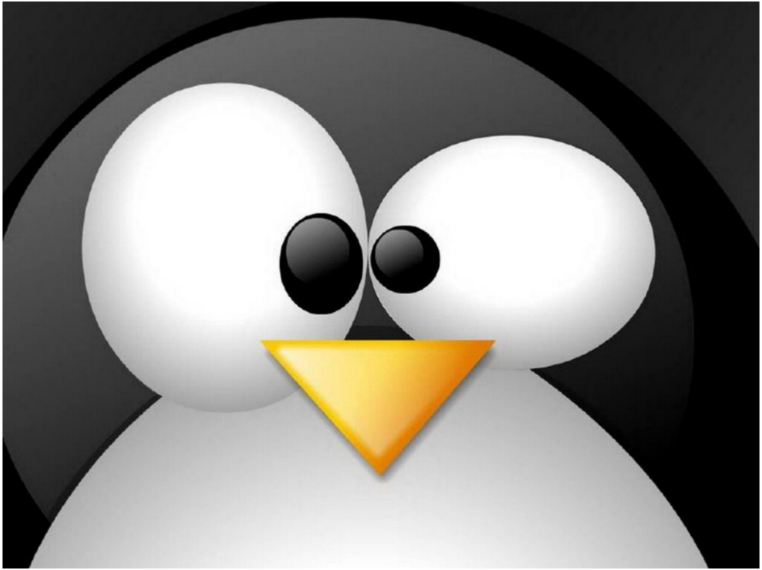
- The `getpriority()` and `setpriority()` system calls allow a process to retrieve and change its own nice value or that of another process.

```
#include <sys/resource.h>

int getpriority(int which, id_t who);

int setpriority(int which, id_t who, int prio);
```

Thread Scheduling

Thread Scheduling (1/2)

- Distinction between user-level and kernel-level threads.
- Process-Contention Scope (PCS)
  - In many-to-one and many-to-many models.
  - Scheduling competition is within the process.
- System-Contention Scope (SCS)
  - In one-to-one model.
  - Scheduling 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.
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling.

- pthread_attr_setscope and pthread_attr_getscope set/get contention scope attribute in thread attributes object.

```
#include <pthread.h>

int pthread_attr_setscope(pthread_attr_t *attr, int scope);

int pthread_attr_getscope(const pthread_attr_t *attr, int *scope);
```

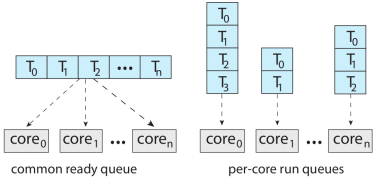
Pthread Scheduling API

```
int main(int argc, char *argv[]) {
  pthread_t t1, t2;
  pthread_attr_t attr;

  pthread_attr_init(&attr);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

  pthread_create(&t1, &attr, thread_func, NULL);
  pthread_create(&t2, &attr, thread_func, NULL);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
}

void *thread_func(void *param) {
  /* do some work ... */
  pthread_exit(0);
}
```

Multi-Processor Scheduling

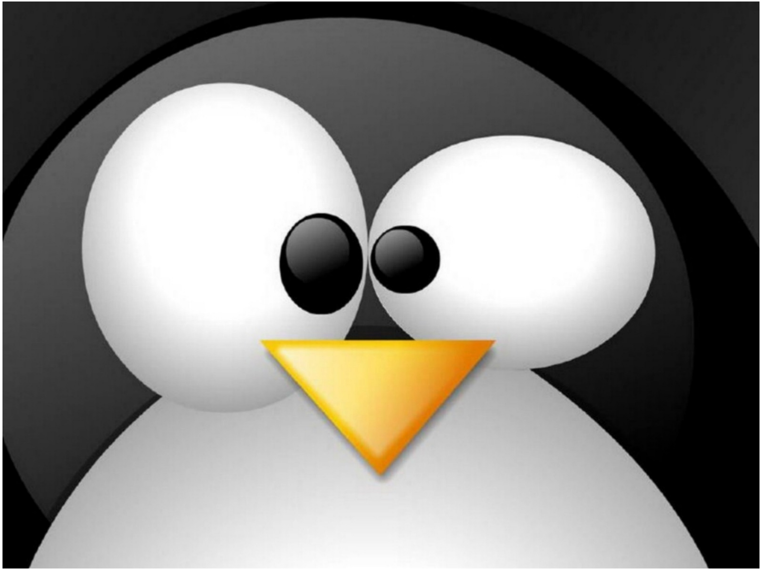
Multiple-Processor Scheduling

- Asymmetric multiprocessing
  - Only one processor does all scheduling decisions, I/O processing, and other system activities.
  - The other processors execute only user code.
- Symmetric multiprocessing (SMP)
  - Each processor is self-scheduling
  - All processes in common ready queue, or each has its own private queue of ready processes.



Processor Affinity

- Processor affinity: keep a process running on the same processor.
- Soft affinity: the OS attempts to keep a process on a single processor, but it is possible for a process to migrate between processors.
- Hard affinity: allowing a process to specify a subset of processors on which it may run.



CPU Affinity

- `sched_setaffinity()` and `sched_getaffinity()` sets/gets the CPU affinity of the process specified by `pid`.

```
#define _GNU_SOURCE
#include <sched.h>

int sched_setaffinity(pid_t pid, size_t len, cpu_set_t *set);

int sched_getaffinity(pid_t pid, size_t len, cpu_set_t *set);
```

CPU Affinity Macros

- `CPU_ZERO()` initializes set to be empty.
- `CPU_SET()` adds the CPU cpu to set.
- `CPU_CLR()` removes the CPU cpu from set.
- `CPU_ISSET()` returns true if the CPU cpu is a member of set.

```
#define _GNU_SOURCE
#include <sched.h>

void CPU_ZERO(cpu_set_t *set);
void CPU_SET(int cpu, cpu_set_t *set);
void CPU_CLR(int cpu, cpu_set_t *set);
int CPU_ISSET(int cpu, cpu_set_t *set);
```

CPU Affinity Macros

- The process identified by `pid` runs on any CPU other than the first CPU of a four-processor system.

```
cpu_set_t set;

CPU_ZERO(&set);
CPU_SET(1, &set);
CPU_SET(2, &set);
CPU_SET(3, &set);

sched_setaffinity(pid, sizeof(set), &set);
```

Summary

Summary

- CPU scheduling
- Scheduling criteria: cpu utilization, throughput, turnaround time, waiting time, response time
- Scheduling algorithms
  - FCFS, SJF, Priority, RR, Multilevel
- Thread scheduling: PCS and SCS
- Multi-processor scheduling: SMP, processor affinity

Questions?

Acknowledgements

Some slides were derived from Avi Silberschatz slides.