CPU Scheduling

(SGG 5.1-5.3)

Instructor: Dr. Tongping Liu

Outline

- Process queues and scheduling
- Different levels of schedulers
- Preemptive vs. non-preemptive
- Context switches and dispatcher
- Performance criteria
  - Fairness, efficiency, waiting time, response time, throughput, and turnaround time;
- Classical schedulers: FIFO, SFJ, PSFJ, and RR
- CPU Gantt chart vs. process Gantt charts

Reviews

- Process
  - Execution of program

- States of Process
  - New and terminated
  - Running \( \rightarrow \) using CPU
  - Ready \( \rightarrow \) in memory, ready for the CPU
  - Waiting \( \rightarrow \) waiting for I/O device or interrupt

**What may cause a process to move out of the CPU?**

State Transitions of A Process

**Possible state transitions** for a process:

Moving out of CPU: out edges of running state
When a Process is Moved Out of CPU

- **I/O request**
  - Need to read/write data from/to a file (on disk)
- A process waits for its child process
  - `fork` → return different values to parent/child processes:
    - parent: child process ID; Child: 0;
- An interrupt or signal
  - Timer interrupt: when time quantum is used up
  - Signal
  - Other synchronizations like wait/join
- A process exits

Process Queues

- **Job queue** (before a process gets into main memory):
  - Processes waiting for allocation of memory
- **Ready queue**:
  - Tasks in main memory, but waiting for the CPU
  - Usually a linked list manages PCBs of all processes
  - Pointers to the first and last PCBs

### Process Queues (cont.)

- **Device queues** (one for each device)
  - Containing all processes waiting for the device
  - Devices include disk drives, tape drives, and terminals
  - Shareable devices (disk drives): may have multiple processes
  - Dedicated devices (tape drives): have at most one process

### Process Queues (cont.)

- Once the process is allocated the CPU and is executing, one of following events may occur:
  - The process could issue an I/O request and then be placed in an I/O queue.
  - The process could create a new child process and wait for the child’s termination.
  - The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.
- A process continues this cycle until it terminates, then it is removed from all queues and has its PCB and other resources deallocated.
Scheduler

A process is migrated among various queues. Operating system must select, for scheduling purposes, processes from these queues in some fashion.

- The selection process is called "Scheduler".

When CPU Scheduler is Invoked?

1. Process switches from running to waiting
   - Requests for I/O operations
2. Process switches from waiting to ready
   - I/O completed
3. Process switches from running to ready
   - Due to timer interrupt
4. Process terminates
   - When a process completes its work

Outline

- Reviews on process
- Process queues and scheduling events
- Different levels of schedulers
- Preemptive vs. non-preemptive
- Context switches and dispatcher
- Performance criteria
  - Fairness, efficiency, waiting time, response time, throughput, and turnaround time;
- Classical schedulers: FIFO, SFJ, PSFJ, and RR
- CPU Gantt chart vs. process Gantt charts

Different Levels of Schedulers

- Short-term (CPU) scheduler
  - Selects which process to execute next
  - Must operate frequently and fast, several times a second
- Medium-term scheduler (for time-sharing systems)
  - Swapping --- moving processes in and out of memory
  - Too many processes → cause paging with decreased performance
- Long-term (job) scheduler
  - Decide which processes are admitted to the system
  - Determines the degree of multiprogramming
  - In stable conditions: invoke only when a process terminates
  - Time-sharing systems have no long-term scheduler
Non-Preemptive vs. Preemptive

- Non-preemptive scheduling: voluntarily give up CPU
  - A process only give up its CPU until it finishes or needs I/O
  - Not suitable for time-sharing

- Preemptive scheduling
  - Process may be taken off CPU non-voluntarily
  - Time-sharing systems have to be preemptive

Dispatcher and Context Switch

- Dispatcher: gives control of CPU to selected process
  - Context switch
  - Setting to program counter (PC)

- Context Switch: switch CPU to another process
  - Save running state of the old process: where to save?
  - Load the saved state of the new process: from where?
  - A few microsecond to 100’s of microseconds, depending on hardware support

Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch
- Context of a process represented in the PCB
- Context-switch time is overhead: the system does no useful work while switching
- Hardware support: Multiple set of registers then just change pointers
- Other performance issues/problems: Cache content: locality is lost

Representation of Processes

- Model of Processes
  - Interleaving between CPU and I/O operations
- CPU bursts
  - The amount of time the process uses CPU before it is no longer ready
- I/O bursts: time to use I/O devices

<table>
<thead>
<tr>
<th>Process</th>
<th>CPU Bursts</th>
<th>I/O Bursts</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>P2</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

(one CPU burst)

(CPU + I/O bursts)

(CPU, I/O, CPU)
Models/Assumptions for CPU Scheduling

- CPU model
  - By default, assume only a single CPU core
  - Exclusive use of CPU: only one process can use CPU

- I/O model
  - Multiple I/O devices
  - Processes can access different I/O devices concurrently, which indicates that I/O operation time of different processes can overlap

An Example: No Multiprogramming

Suppose 2 processes, where each process
- Require 20 seconds of CPU time
- Wait 10 second for I/O for every 10 seconds execution

Without multiprogramming: runs one after another
- Each takes 40 seconds: 20s run → 10s wait → 10s run → total 80 sec
- CPU utilization is about 50%

Multiprogramming

- Multiprogramming is a form of parallel processing in which several programs are run at the same time on a uniprocessor.
- Objective?
  - Maximize CPU utilization. When a process wait for IO, all waiting time is wasted and no useful work is accomplished.

An Example: with Multiprogramming

Multiprogramming: both processes run together
- The first process finishes in 40 seconds
- The second process uses CPU (I/O) alternatively with first one and finishes 10 second later → 50 seconds

Total time: 50 seconds
CPU-bound vs. IO-Bound

- Bursts of CPU usage alternate with periods of I/O wait
- **CPU-bound**: high CPU utilization, interrupts are processed slowly
- **I/O-bound**: more time is spending on requesting data than processing it

![Diagram showing CPU-bound and I/O-bound processes]

Outline

- Reviews on process
- Process queues and scheduling events
- Different levels of schedulers
- Preemptive vs. non-preemptive
- Context switches and dispatcher
- Performance criteria
  - Fairness, efficiency, waiting time, response time, throughput, and turnaround time;
  - Classical schedulers: FIFO, SFJ, PSFJ, and RR
  - CPU Gantt chart vs. process Gantt charts

Performance Criteria

- Methods of measuring performance of CPU schedulers
- Fairness
  - Each process gets a fair share of CPU in Multiprogramming
- Efficiency: CPU Utilization
  - Percentage of time CPU is busy; e.g., 40%, 80%, 100%?
- Throughput
  - Number of processes completed per unit time
  - E.g., 10 per second; or 1 per hour
- Turnaround Time
  - Time from submission to termination
  - Sum of CPU time, I/O time, and waiting time

Performance Criteria (cont.)

- Waiting time
  - Time for a process waiting for CPU in a ready queue
  - Scheduling algorithms: **no effect on CPU or I/O time**
- Response time
  - Time between submission and the first response
  - Good metric for interactive systems
- Response time variance
  - For interactive systems, response time should **NOT** vary too much
Calculate total, wait, response times

- Given a process:
  - Arrival time: \( t_a \)
  - First response time: \( t_r \)
  - Finish time: \( t_f \)
  - Total CPU burst time: \( t_{cpu} \)
  - Total I/O time: \( t_{io} \)

- Turnaround time: the process spent in the system
  \[ T_{turn\_arround} = t_f - t_a = t_{cpu} + t_{io} + t_{wait} \]

- Waiting time: the process spent in the ready queue
  \[ t_{wait} = T_{turn\_arround} - t_{cpu} - t_{io} \]

- Response time: the process waited until the first response
  \[ t_{response} = t_r - t_a \]

Scheduling Algorithm

- Deciding which of the processes in the ready queue is to be assigned with the CPU:

- Parameters may be set by user processes:
  - Don’t allow a user process to take over the system!
  - Allow a user process to voluntarily lower its own priority
  - Allow a user process to assign priority to its threads

Classical Scheduling Algorithms

- First-Come First Served (FCFS)
- Shortest Job First (SJF)
- Preemptive Shortest (remaining) Job First (PSJF)
- Round Robin Scheduling (RR)

First-Come First Served (FCFS)

- Managed by a strict FIFO queue
- CPU Gantt chart:
  - show which process uses CPU at any time
- An Example of 3 processes arrive in order:
  - \( P_1: 24 \) (CPU burst time), \( P_2: 3 \), \( P_3: 3 \)
  - CPU Gantt chart for the example:

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Average waiting time (AWT) = \( (0 + 24 + 27)/3 = 17 \)

What about processes arrive in reverse order?
First-Come First Served (FCFS): cont.

- An Example of 3 processes arrive in reverse order
  - P3:3, P2:3, and P1: 24 (CPU burst time)
  - CPU Gantt chart for the example

- AWT = (0 + 3 + 6)/3 = 3 !!!
- Big improvement of AWT over the previous case!

Problem of FCFS: long jobs delay every job after them. Many processes may wait for a single long job. *Conveyor effect: short process behind long process*

---

Process Gantt Chart

- For each process, PGT show its state at any time
  - For the example: P1: 24 (CPU burst time), P2: 3, P3: 3
  - Process Gantt chart

---

Another Example: CPU and I/O Bursts

- Two processes

<table>
<thead>
<tr>
<th>Process</th>
<th>CPU Burst</th>
<th>I/O Burst</th>
<th>CPU Burst</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>3</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

- Process Gantt chart for FCFS

- AWT = (0+9) / 2 = 4.5  CPU Utilization=19/20

Notes:
- Waiting time for process is the number of r's in the string
- AWT is total number of r's divided by number of processes
- CPU utilization is the total number of R's divided by the total time (the length of the longer string)

---

Shortest Job First (SJF)

- SJF: run the job with the shortest CPU burst first!

Example of 4 ready processes

- CPU Gantt chart

- AWT = (0+3+9+16)/4 = 7

Notes:
- Which job has the shortest CPU burst?
  - Use past history to predict
  - But it is not correct if the cpu burst changes
Shortest Job First (SJF): cont.

Example 3: SJF
- Process Gantt chart
  - P1: \( rrrrrRRRRRRRRRRRRRRRRRRRRRRRR \)
  - P2: \( RRRRRRRRRRRRRRRRRR \)
- Average waiting time:
  - P1: waits for 6 units first, then CPU, then I/O, and CPU
  - P2: runs for 6, does I/O, waits for 5 units, and CPU
  - AWT = \( \frac{6 + 5}{2} = 5.5 \)

**Whether SJF has the least AWT?**

Problem: how long a job will take?

Preemptive Shortest Job First (PSJF)

- SJF can be non-preemptive or **preemptive**
- **Preemptive** Shortest Job First
  - If a new process enters the ready queue that has a shorter next CPU burst compared to what is expected to be left (remaining) of the currently executing process
  - Current running job will be replaced by the new one
- **Shortest-remaining-time-first** scheduling
- Example 3: Process Gantt chart
  - AWT = \( \frac{8+0}{2}=4 \)

Round Robin Scheduling (RR)

- Non-preemptive: process keeps CPU until it terminates or requests I/O
- Preemptive: allows higher priority process preempt an executing process (if its priority is lower)
- Time sharing systems
  - Need to avoid CPU intensive processes that occupy the CPU too long
- Round Robin scheduler (RR)
  - Quantum: a small unit of time (10 to 100 milliseconds)
  - Processes take turns to run/execute for a quantum of time

Round Robin Scheduling (RR): Examples

- Example 1: RR with quantum of 4
  - AWT = \( \frac{6+4+7}{3} = 5.67 \)
- Example 3: RR with quantum of 3
  - AWT = \( \frac{6+6}{2} = 6.0 \)
Round Robin Scheduling (RR): Quantum

- If quantum is small enough
  - For \( n \) processes, each appears to have its own CPU that is \( \frac{1}{n} \) of CPU’s original speed

- Quantum: determines efficiency & response time

- How to decide quantum size? → **10 to 100 ms**
  - Too small → too many context switches → no useful work
  - Too long → response time suffers
  - Rule of thumb: 80% of CPU bursts \( \leq \) quantum

Priority Based Scheduling

- Assign a priority to each process
  - “Ready” process with highest priority allowed to run
  - Same priority: round-robin

- Priorities may be assigned dynamically
  - Reduced when a process uses CPU time
  - Increased when a process waits for I/O

Comparison of Schedulers

- FCFS: simple, non-preemptive
  - Process with long CPU bursts → starve out shorter ones

- SJF: short processes go first, non-preemptive
  - Hard to know the exact next CPU burst, need approximate
  - Long CPU bursts can delay shorter processes

- PSJF: preemptive
  - Still needs to be approximated

- RR: preemptive
  - Uses a quantum, and common for time sharing systems

Classical Scheduling Algorithms

- FCFS: non-preemptive, based on arrival time
  - Long waiting time, e.g. long process before SSH console?

- SJF(shortest job first): preemptive & non-preemptive
  - Optimal in term of waiting time

- RR (Round-robin): preemptive
  - Processes take turns with fixed time quantum e.g., 10ms

- Priority-based scheduling
  - Real-time systems: earliest deadline first (EDF)
Summary

- Reviews on process
- Process queues and scheduling events
- Different levels of schedulers
- Preemptive vs. non-preemptive
- Context switches and dispatcher
- Preference criteria
  - Fairness, efficiency, waiting time, response time, throughput, and turnaround time;
- Classical schedulers: FIFO, SJF, PSJF, and RR
- CPU Gantt chart vs. process Gantt charts