G22.0202-001
Honors Computer Architecture II
Lecture: CPU Scheduling

Outline

- CPU Scheduling
  - basic concepts
  - scheduling criteria
  - scheduling algorithms
  - algorithm evaluation

Silberschatz/Galvin: Sections 5.1 - 5.3, 5.6

Overview

- What is scheduling?
  - Simply deciding which process to execute and for how long

- Scheduling, why do we need it?
  - Better resource utilization
  - Improve the system performance for desired load pattern
  - Illusion of parallel task execution
  - Example: Editing and compiling
  - Can enable providing of specific guarantees

Scheduling: Components

- Processes
- Scheduler
  - focus on short-term scheduling (of the CPU)
  - decide which process to give the CPU to next
    - rationale: utilize CPU resource better
    - can also be necessary because of other factors: fairness, priorities, etc.

- Dispatcher:
  - suspends previous process and (re)starts new process
  - context switch, including adjusting and updating the various process queues
  - switch to user mode from the scheduler’s supervisor mode
  - jump to the appropriate point in user space and executing “running” process
Scheduler: Operating Details

- Process States / Queues
  - Running, Ready, Waiting
- Invoked in the following situations (triggers)
  - process switches from running to waiting state
    (e.g., block for I/O, wait for child)
  - process switches from running to ready state
    (e.g., expiration of timer)
  - process switches from waiting to ready state
    (e.g., completion of I/O)
  - process terminates

Component interactions

Process Behavior

- CPU versus I/O bursts
  - a given process’ behavior is broken into
    - a run of activity on the CPU referred to as a CPU burst
    - a run of non-CPU (usually I/O) activity or an I/O burst
  - the overall execution of a process is alternating CPU and I/O bursts

  - CPU burst lengths typically characterized as exponential or hyperexponential (make sure you understand this)
    - CPU bound processes: few, long CPU bursts
    - I/O bound processes: many, very short CPU bursts

Sample process characteristics

- Which one is CPU bound?

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>IO</th>
<th>CPU</th>
<th>IO</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 1</td>
<td>10</td>
<td>1000</td>
<td>15</td>
<td>4000</td>
<td>5</td>
</tr>
<tr>
<td>Process 2</td>
<td>20</td>
<td>2</td>
<td>20</td>
<td>2</td>
<td>20</td>
</tr>
</tbody>
</table>
Preemption

- **Preemptive versus non-preemptive scheduling**
  - non-preemptive scheduling policy
    - process switches to a waiting state only as a function of its own behavior
    - i.e., when it invokes OS services, or when it terminates
  - preemptive scheduling policy
    - if its state can be switched otherwise

- **Cost: Maintaining consistent system state**
  - suspension might need interrupts to be turned off
    - e.g., the process being suspended is updating sensitive kernel data-structures
    - however, interrupts cannot always be ignored - possible loss of data
  - coordination of the state of interrupted processes

- Simple(most Unix versions) implementation - wait until safe to context switch.
  - Bad for real-time and multiprocessing

---

Policies: How do we make decisions?

- Scheduler uses some policy(or a combination of policies) to determine which process to run next

  - Examples (we will see details later):
    - FCFS (non-preemptive)
    - RoundRobin (preemptive)

- Issues
  - what information can the policy use to make decisions?
  - what is the best scheduling algorithm?
  - how do we even compare (metrics)?
  - flexibility - scheduling algorithm can be tuned for different application requirements

---

Scheduling Metrics:

**User Oriented**

- Performance Related
  - response time: time it takes to produce the first response
  - turnaround time: time spent from the time of "submission" to time of completion
  - deadlines: the time within which the program must complete (the policy must maximize percentage of deadlines met)

- Other
  - predictability: expectation that the job runs the same regardless of system load

**System Oriented**

- Performance Related
  - waiting time: time spent waiting to get the CPU
  - throughput: the number of processes completed per unit time (directly affected by the waiting time)
  - CPU utilization: percentage of time the CPU is busy

- Other
  - fairness: no process should suffer starvation
  - enforcing priorities: higher priority processes should not wait

---

FCFS Scheduling

- **Non-preemptive**

  - **Implementation**
    - a queue of processes
    - new processes enter the ready queue at the end
    - when a process terminates
    - the CPU is given to the process at the beginning of the queue (in practice) when a process blocks
    - it goes to the end of the queue
    - the CPU is given to the process at the beginning of the queue

  - **How does FCFS perform?**
Two Scheduling Metrics

<table>
<thead>
<tr>
<th>Task 1</th>
<th>Task 2</th>
<th>Task 3</th>
<th>...</th>
<th>Task i</th>
<th>...</th>
<th>Task n</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_1$</td>
<td>$d_2$</td>
<td>$d_3$</td>
<td>...</td>
<td>$d_i$</td>
<td>...</td>
<td>$d_n$</td>
</tr>
</tbody>
</table>

- Average wait time to finish
  $$d_1 + (d_1 + d_2) + (d_1 + d_2 + d_3) + \ldots + \sum_{j=1}^{i} d_j + \ldots + \sum_{j=1}^{n} d_j$$

- Average wait time to start
  $$0 + d_1 + (d_1 + d_2 + \ldots + \sum_{j=1}^{n} d_j + \ldots + \sum_{j=1}^{i} d_j)$$

Worst case FCFS scenario

- Consider: 3 CPU bound processes with average time to start metric
  - $P_1 = 24$, $P_2 = 3$, $P_3 = 3$ come in this order

<table>
<thead>
<tr>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
</tr>
</thead>
</table>

  - This means $(0+24+27)/3 = 17$ msec, can we do better?

<table>
<thead>
<tr>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_1$</th>
</tr>
</thead>
</table>

  - Compare to: $(0+3+6)/3 = 3$ msec !!!

Evaluation of FCFS

- **Pro**: Very simple code, data-structures and low overhead

- **Con**: Can lead to large average waiting times

- General disadvantage due to lack of preemption
  - when a poorly (long-term) scheduled collection has one large task with lots of CPU needs and a collection of others with I/O intensive needs
    - the CPU intensive process can cause very large delays for the processes needing (mostly) I/O

Shortest Job First (SJF)

- The next process to be assigned the CPU is one that is ready and with *smallest next CPU burst*
  - FCFS is used to break ties

  - **Pro**
    - if times are accurate, SJF gives *minimum* average waiting time

  - **Con**
    - it is difficult to estimate CPU burst times - remember this is sort of like predicting the future
    - can we do a reasonable job, well let's see

2/11/00  
Palem, Harrison, and Karamcheti  
13

2/11/00  
Palem, Harrison, and Karamcheti  
14

2/11/00  
Palem, Harrison, and Karamcheti  
15

2/11/00  
Palem, Harrison, and Karamcheti  
16
Estimating the CPU Burst

- For long-term scheduling
  - the user can be "encouraged" to give an estimate (cheating :)
  - part of the job submission requirements

- For short-term scheduling
  - we can attempt to predict its value
    - the approach assumes some locality in process CPU burst times
    - use exponential averaging
      \[ \tau_{n+1} = \alpha \cdot \tau_n + (1 - \alpha) \cdot T_n \]
      where,
      - \( T_n \) is the estimated value for the \( n \)th CPU burst
      - \( \tau_n \) is the actual most recent burst value
    - Notice: the bigger the \( \alpha \) (\( \alpha \in [0,1] \)) the faster we adjust, but we are also more sensitive to unrepresentative fluctuation
  - the estimate lags the (potentially) sharper transitions of the CPU bursts

---

Modifications to SJF

- Preemptive SJF
  - if the shortest CPU burst among all the processes in the ready queue (say this belongs to \( P_j \)) is less than the remaining time for the one that is running,
    - preempt the currently running job;
    - use its remaining time as its next CPU burst and add it to the ready queue
    - start process \( P_j \)

  - this policy is also called: shortest-remaining-time-first

  - Suggested exercise: Construct a sequence of burst times for which the preemptive version yields lower average waiting time per-process

  - policy prioritizes jobs with short CPU bursts ...

---

Priorities: Making Policies Flexible

- To create an instance of priority scheduler we
  - define priority range (for example 0.0 to 1.0)
    - convention: a smaller number means higher priority
  - define tie-breaker mechanism (for example FCFS)
  - map priority to considerations we have in mind:
    - internal
      - memory and other needs of the job
      - ratio of CPU to I/O burst times
      - number of open files etc.
    - external
      - the amount of money paid by the process owner
      - the importance of the user group running the process

- Priority-based scheduling
  - assign the CPU to the process with highest priority
  - may be used with or without preemption
Problems with Priority Schemes

- A process can continuously be overtaken by higher priority processes arriving later
  - can lead to starvation
  - leads to better overall performance perhaps
    - but not from the point of view of the process in question
  - happens in real OSes unless special measures are taken

- Common solution:
  - a process' priority goes up with its age
    - FCFS is used to break ties between processes with equal priorities
    - a process will not wait forever
      - given enough time in the ready queue, its priority will eventually be the highest

- What should happen if low-priority process holds resources required by the high-priority process? *(priority inversion)*

---

Example of Priority Ageing: Unix

- Priority goes up with lack of CPU usage
  - process accumulates CPU usage
  - every time unit (~1 second)
    - recalculates priority
      - priority = CPUusage + basepriority
    - halves CPU usage carried forward
      - CPUusage = (CPUusage) / 2
    - recall that smaller number implies a higher priority
  - Base priority is settable by user
    - within limits
    - using "nice"

---

Round Robin (RR) Scheduling

- A strictly preemptive policy

- At a general level
  - choose a fixed time unit, called a quantum
  - allocate CPU time in quanta
  - preempt the process when it has used its quantum

  - typically, FCFS is used as a sequencing policy
    - each new process is added at the end of the ready queue
    - when a process blocks or is preempted, it goes to the end of the ready queue

  - very common choice for scheduling interactive systems

---

Choice of Quantum Size

- Quantum size $q$ is critical

- Affects waiting and turnaround times
  - if $q$ is the quantum size and there are $n$ processes in the ready queue,
    - the maximum wait is $(n-1) * q$ units of time
  - as $q$ increases, we approach FCFS scheduling
  - as $q$ decreases
    - the rate of context switches goes up, and the overhead for doing them
    - the average wait time goes down, and the system approaches one with $1/n$ the speed of the original system

- Determining the right tradeoff between responsiveness and cycles wasted on context switching helps to select the optimal $q$
Towards greater flexibility: Multilevel Queues

- Processes are partitioned into groups based on static criteria
  - background (batch)
  - foreground (interactive)

- All the processes in a fixed group of the partition share the same scheduling strategy and a distinct family of queues
  - different scheduling algorithm can be used across different groups
    - background: FCFS
    - foreground: Round Robin

- Need to schedule the CPU between the groups as well
  - fixed-priority: e.g., serve all from foreground, then from background
  - possibility of starvation
  - time slice: each group gets a certain fraction of the CPU
    - e.g., 80% to foreground in RR, 20% to background in FCFS

Even more flexible: Multilevel Feedback Queues

- Provide a mechanism for jobs to move between queues
  - ageing can be implemented this way

- Complete specification
  - queues: number, scheduling algorithms (within and across queues)
  - promotion and demotion policies
  - which queue should a process enter when it needs service?

- Example
  - 3 queues: Q_0 (FCFS, 8ms), Q_1 (FCFS, 16ms), Q_2 (FCFS)
  - scheduling:
    - process starts in Q_0, gets 8ms of the CPU (FCFS)
    - if not done, it moves to Q_1, where it gets a further 16 ms of the CPU (FCFS)
    - if still not done, it moves to Q_2 and stays there till completed

Summary

- Scheduling – a critical component of OS
- Metrics
- Policies: FCFC, SJF, PSJF ...
- Priority Scheduling
- Next: How do we choose? (Analysis, simulation, etc)