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
    - invoked in the following situations
      - 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
    - 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

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
    - CPU bound processes: few, long CPU bursts
    - I/O bound processes: many, very-short CPU bursts
Preemption

- Major algorithm classification
- **Preemptive versus non-preemptive** scheduling
  - the corresponding scheduling policy is non-preemptive
    - if a process switches to a waiting state only as a function of its own behaviour
      - i.e. when it invokes OS services, or when it terminates
    - it is preemptive
      - if its state can be switched otherwise
- Cost: Maintaining consistent system state while the processes are suspended in the midst of critical activity
  - 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
  - poses challenging problems to coordinating the states of processes interrupted in a preemptive way
  - details in Lectures 7-11

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

- **Average wait time to finish**
  \[
  \frac{d_1 + (d_1 + d_2) + (d_1 + d_2 + d_3) + \cdots + \sum_{j=1}^{i=1} d_j + \cdots + \sum_{j=1}^{i=n} d_i}{n}
  \]
- **Average wait time to start**
  \[
  \frac{0 + d_1 + (d_1 + d_2) + \cdots + \sum_{j=1}^{i=1} d_j + \cdots + \sum_{j=1}^{i=n} d_i}{n}
  \]
Performance of FCFS

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

• **Con**: Can lead to large average waiting times
  – for example:
    • four processes P1, P2, P3, P4 arriving in that order
    • needing CPU times 20, 4, 3, 3
    • FCFS scheduling: an average wait time to finish of 25.25 units
      \[ \frac{(20 + (20+4) + (20+4+3) + (20+4+3+3))}{4} \]
    • If arrival order is P4, P3, P2, P1, average wait time to finish is 12.25 units

• 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

Estimating the CPU Burst

• For long-term scheduling
  – the user can be “encouraged” to give an estimate
  – 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,
      – \( \tau_n \) is the estimated value for the n’th CPU burst
      – \( T_n \) is the actual most recent burst value
    – the estimate lags the (potentially) sharper transitions of the CPU bursts

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
  – from the previous example,
    • P1, P2, P3, P4 arriving in that order, needing CPU times 20, 4, 3, 3
    • FCFS yielded an average wait time to finish of 25.25 units
    • SJF yields order P3, P4, P2, P1, with average wait time to finish of 12.25

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

• **Con**
  – it is difficult to estimate CPU burst times

Estimating the CPU Burst (contd.)
Modifications to SJF

• Preemptive SJF
  – if the shortest estimated 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: A More General Notion

• Assign a numerical priority to each process
  – convention: a smaller number means higher priority
  – examples of priority use
    • value of the next CPU burst
    • importance of interactive responsive
  – priorities can be based upon external considerations ….
    • the importance of the user group running the process
    • the amount of economic investment that the group might have in the system
  – … or upon internal considerations
    • memory and other needs of the job
    • ratio of CPU to I/O burst times
    • number of open files etc.
  – Priority-based scheduling
    – assign the CPU to the process with highest priority
    – may be used with or without preemption

Disadvantages of 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
        • 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)
    • more about this in Lectures 7-11

Example of Priority Ageing: Unix

• Priority goes up with lack of CPU usage
  – process accumulates CPU usage
  – every time unit (~ 1 second)
    • recalculates priority
      \[
      \text{priority} = \text{CPUusage} + \text{basepriority}
      \]
    • halves CPUusage carried forward
      \[
      \text{CPUusage} = (\text{CPUusage}) / 2
      \]
    • recall that smaller number implies a higher priority
  – basepriority 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 $(n-1) \cdot q$ 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

Multilevel Queue Scheduling

- 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

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
Choosing a Scheduling Approach

- Identify metrics for evaluation
  - we have already seen a variety of metrics
    - throughput, wait time, turnaround time, ...
  - the goal is to start with an expectation or specification of what the scheduler should do well
    - for example, we might wish to have a system in which
      - the CPU utilization is maximized, subject to a bound on the response time

- Evaluate how different scheduling algorithms perform
  - deterministic modeling
    - requires accurate knowledge of job and system characteristics
    - practical only for real-time and embedded systems
  - more detailed performance evaluation
    - queuing models
    - simulation
    - measurement

Queueing Analysis

- Characterize system components as random processes
  - random variables for CPU, I/O bursts
    - typically have exponential distributions
  - random variable for process arrival time
  - we are interested in the average values of metrics
    - e.g., average throughput, utilization

- System modeled as a network of servers

- Queueing theory:
  - for a queue, Little’s formula is $n = \lambda W$
    - where $\lambda$ is the arrival rate; $W$ is the average waiting time; and $n$ is the average queue length

  Has limitations on the degree to which realistic systems can be modeled and analyzed

Simulations

- Use computerized models of the system
  - when analytical solutions are infeasible, the models are simulated
  - workloads can be either
    - distribution driven as in the previous case,
    - or actual (possibly synthetic) workloads

- Leads to more realistic predictions than the above forms of analysis
- Computationally more expensive than either of the above approaches

Actual Measurements

- Instrument a real system and capture the extra information
- Can make very precise determination
- Very cumbersome and time-consuming to achieve
  - a problem is that observation changes the system
    - may require hardware monitoring
Lecture Summary

• CPU Scheduling
  – components: processes, scheduler, dispatcher
  – process behavior
  – scheduling criteria
  – scheduling algorithms
    • FCFS
    • SJF
    • Priority-based
    • Round Robin
    • Multilevel Queues
  – evaluation of scheduling algorithms

Next Lecture

• CPU Scheduling (contd.)
  – multiple-processor scheduling
  – real-time and fair-share scheduling
  – case study: lottery scheduling
    • rationale and design
    • evaluation
    • performance

Readings

  – Silberschatz/Galvin: Sections 5.4 - 5.5
    Lottery Scheduling: Flexible, Proportional-Share Resource Management,
    OSDI'94, pp. 1-11.