Scheduling Algorithms

- Primarily interested in
  - Short-term scheduling
    - Context-switch Processes
    - Utilize CPU resource better
  - Necessary
    - when a process terminates
    - other factors
Context Switching

• Change state as follows:
  • Running to ready state
    • Caused by an interrupt by a timer for example
      • e.g. time allotment finished
  • Ready to running state
    • Scheduler decision
  • Running to waiting state
    • Caused by I/O or other suspensions
  • (Waiting to ready state)
    • Upon completion of I/O waits for example
    • Upon termination and exiting the system
The Dispatcher

• An important component in scheduling
• Handles the following functions once the decision is made --- based on policy --- to suspend a process
  • Switching context includes: adjusting and updating the various process queues
  • Switching to user mode from the scheduler's supervisory mode
  • Jumping to the appropriate point in the user space and executing the now “running” process
Scheduling metrics

- **CPU Utilization**: the percentage of time CPU spends actively computing on user processes
- **Throughput**: the “amount” of processing completed per time unit
- **Turnaround Time**: time spent from the time of “submission” to time of completion
- **Waiting Time**: sum of times spent in the ready queue
- **Response Time**: time it takes to produce the first response (and not the completion of the whole process or the output actually reaching the user)
- (The last two are directly under the CPU scheduler's control)
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 OS) activity or an *OS burst*
• The overall execution of a process is alternating CPU and OS bursts
Pre-emption

• 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
FCFS Scheduling

• Non-preemptive
• Algorithm implements:
  • 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
Given a Task Execution System
Two Definitions

• **Average wait time to finish**

\[
\frac{d_1 + (d_1+d_2) + (d_1+d_2+d_3) + \ldots + i d_j + \ldots + j d_k}{n} \]

\[
= \frac{1}{n} \sum_{j=1}^{i} (d_1 + \ldots + d_j) + \frac{1}{n} \sum_{k=1}^{n} j d_k
\]

• **Average wait time to start**

\[
\frac{0 + d_1 + (d_1+d_2) + \ldots + 2d_j + \ldots + j d_k}{n}
\]

\[
= \frac{1}{n} \sum_{j=1}^{i-1} (d_1 + \ldots + d_j) + \frac{1}{n} \sum_{k=1}^{n-1} j d_k
\]
Pros of FCFS

• Very simple code, data-structures and hence overhead
Cons of FCFS

• 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
      • $(20 + (20+4) + (20+4+3) + (20+4+3+3)) / 4$
    • If arrival order is P4, P3, P2, P1, average wait time to finish is 12.25 units
Cons of FCFS (continued)

• General disadvantage due to lack of preemption
• More specific problem
  • 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
FCFS in practice

• Algorithm similar to above, but also:
  • 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
  • Still has problems with CPU-bound processes holding up I/O bound processes
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 scheduling yielded an average wait time to finish of 25.25 units
  - SJF scheduling yields order P3, P4, P2, P1, with average wait time to finish of 12.25
SJF pros & cons

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

• Con
  • It is difficult to estimating CPU burst times
Estimating the CPU Burst

- For long-term scheduling, the user can be “encouraged” to give an estimate
- Not so easy in the case of short-term scheduling
- We can attempt to predict its value
  - The approach is to assume some locality in the CPU burst times of processes
  - Use $t_{n+1} = a \times T_n + (1 - a) \times t_n$
  - where
    - $t_n$ is the estimated value for the n’th CPU burst which stores the past history of estimates and $T_n$ is the actual most recent burst value
SJF Profile

- The actual value of the estimate lags the (potentially) sharper transitions of the CPU bursts
- The smoother envelope is called *exponential averaging*

- $T = 4, 8, 3, 7$
Modifications to SJF

• Preemptive SJF
  • if the shortest estimated CPU burst of all the processes in the ready queue (say it is \( P_j \)) 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 called the \textit{shortest-remaining-time-first}
    • Suggested exercise:
      • Construct a sequence of burst times for which the preemptive version yields lower average waiting time per-process
Priorities: A More General Notion

• Assign a numerical priority to each process
  • convention: a smaller number means higher priority
  • Examples
    • priority = value of the next CPU burst
• Assign the CPU to the process with highest priority
• May be used with or without pre-emption
• Priorities can in general be based on
  • *External* or *Internal* considerations
Priority considerations

• External
  • Based on considerations such as
    • The importance of the user group running the process
    • The amount of economic investment that the group might have in the system

• Internal
  • Based on considerations such as
    • Memory and other needs of the job
    • Ratio of CPU to I/O burst times
    • Number of open files etc.
A Disadvantage 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 OSs unless special measures are taken
A Common Solution

• A process' priority goes up with its age
  • FCSF 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
A variant: Unix

- Priority goes up with lack of CPU usage
- Unix
  - accumulates CPU usage
  - every time unit (~ 1 second)
    - recalculates priority
      - priority = CPUusage + basepriority
    - halves CPUusage carried forward
      - CPUusage = (CPUusage) / 2
  - basepriority is settable by user
    - within limits
    - using “nice”
Summary

• So Far
  • We started out with a strictly non-preemptive scheduling approach
  • Ended up with one which allows preemption as an option
  • We will now look at a widely used scheduling algorithm which is strictly preemptive
Round Robin (RR) Scheduling

- At a general level
  - Choose a fixed time unit, called a *quantum*
  - Allocate CPU time in quanta
  - Pre-empt 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 pre-empted, 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 wait 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
  • Can include things such as “background” versus “foreground” or other distinctions
• 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
• The CPU is allotted to each group also by using some fixed scheduling algorithm usually with a fixed priority scheme
Real-time Operating Systems

- In *hard* real-time systems, the timing constraints are very rigid
  - The scheduler must know exactly what timing penalties are involved in various kinds of accesses
  - Impractical in a system with virtual memory, secondary storage and so on
  - Limited general purpose OS functionality is offered

- In contrast, *soft* real-time systems
  - have less restrictive demands on timing behavior
  - Possible to achieve a level of general OS functionality
Considerations for RTOSs

- Need to ensure that a memory access can be performed in a predictable manner
- The scheduler is carefully designed to make sure that higher priorities are strictly obeyed
- Priorities are used to control process scheduling
- Preemption is usually permitted and aging is typically not used
- Dispatch latencies are kept very low
- Motivates the need for a very careful design of the kernel and all of its data structures which must be interruptible
Choosing a Scheduling Approach

• 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
Deterministic Modeling

• Pros
  • precise performance estimates for comparison purposes
  • sharp formulae for calculating the various metrics
  • valuable in demonstrating the concept etc.

• Con: not very useful in practice
  • except for RTOSs
More Detailed Performance Evaluation

- Queueing Models
- Simulation
- Measurement
Queueing Analysis

• Characterize the system as a *random process*
  • random variables have *exponential* distributions
  • we are interested in the *average* values

• Queueing theory:
  • For a queue, Little’s formula is
    • $n = RW$
  • where R 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
  - observation changes the system
  - may require hardware monitoring
Some Difficulties of Preemption

• Maintaining consistent system state while the processes are suspended in the midst of critical activity
  • Suspension might need interrupts to be turned off
    • For example, the process being suspended might be 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 later in Lecture 5