Scheduling

Scheduler
- Part of the operating system that decides which ready process to run next

Scheduling Algorithm
- Algorithm the scheduler uses

Types of Processes
- I/O-bound
- CPU-bound

Scheduling Goals

Throughput
- Number of jobs per time period (or work/time period)

Fairness
- Some jobs aren’t arbitrarily treated differently from others

Response Time
- Time till some output

Turnaround time
- Time when process arrive to when process completes

Wait time
- Time spent waiting

Predictability
- Low variance

Meeting deadlines
- Multimedia, for example

CPU utilization
- Don’t waste (if critical path!)

Proportionality
- Simple things are quicker than complicated things
Scheduling Algorithms

First-Come First-Served
- No preemption
- Easy to understand
- Low throughput and CPU utilization given I/O-bound processes

A: arrives at time 0, CPU time 3
B: arrives at time 2, CPU time: 6
C: arrives at time 4, CPU time: 5
D: arrives at time 6: CPU time: 4

Scheduling Algorithms

Round-robin scheduling
- Give each job a timeslice (quantum). Preempt if still running
- Put job at end of ready queue and run next job in ready queue
- Quantum should be large relative to context-switch time

A: arrives at time 0, CPU time 3
B: arrives at time 2, CPU time: 6
C: arrives at time 4, CPU time: 5
D: arrives at time 6: CPU time: 4

Scheduling Algorithms

Shortest Process Next
- Look at CPU burst (not total CPU until completed)
- Let job run non-preemptively
- Predict based on history of CPU burst (\(T_i = \text{time used for } i\text{th period. } S_i = \text{estimate of } i\text{th period})
  - Straight average: \(S_{n+1} = \frac{1}{n}T_n + \frac{n-1}{n}S_n\)
  - Exponential average: \(S_{n+1} = \alpha T_n + (1-\alpha)S_n\)
- Reduces average turnaround time

A: arrives at time 0, CPU time 3
B: arrives at time 2, CPU time: 6
C: arrives at time 4, CPU time: 5
D: arrives at time 6: CPU time: 4

Scheduling Algorithms

Shortest Remaining Time
- Based on estimate of total time (given by user or estimated from history)- time spent so far
- Like Shortest Process next but preemptive at time a job arrives

A: arrives at time 0, CPU time 3
B: arrives at time 2, CPU time: 6
C: arrives at time 4, CPU time: 5
D: arrives at time 6: CPU time: 4
Scheduling Algorithms

Highest Response Ratio Next
- Response ratio = (waiting time + expected CPU time)/expected CPU time
- Chose job with highest response ratio
  A: arrives at time 0, CPU time 3
  B: arrives at time 2, CPU time: 6
  C: arrives at time 4, CPU time: 5
  D: arrives at time 6: CPU time: 4

Priority Scheduling
- Priority associated with each process
- High priority job runs before any lower priority jobs (if preemptive, stop current job if higher priority job becomes available).
- Starvation is a problem
  - Solution: Aging (slowly increase priority of waiting processes)
  A: arrives at time 0, CPU time 3 (priority L)
  B: arrives at time 2, CPU time: 6 (priority L)
  C: arrives at time 4, CPU time: 5 (priority H)
  D: arrives at time 6: CPU time: 4 (priority M)

Multilevel Queue Scheduling
- Distinguish between different classes of processes
  - Student/Instructor/interactive/system
  - batch/interactive
- Queues can have different scheduling algorithms
- Need scheduling between queues

Multilevel Feedback-Queue Scheduling
- Different queues, processes move from queue to queue based on their history
- For example:
  - Queue 1: 1 quantum. If a process uses its entire quantum, it moves to next Queue
  - Queue 2: 2 quanta. If a process uses its entire quantum, it moves to next Queue
  - Queue 3: 4 quanta. ...
- ...

Fair-share scheduling
- Divide user community into a set of fair-share groups.
- Allocate a fraction of the processor resource to each group
- Each group gets its fair-share of the CPU time
- Priority of a process depends on:
  - How much CPU time its group has had recently
  - How much CPU time it itself has had recently
  - Base priority of the process
- Example
  - 3 process: A, B, C. A is in group 1, B&C are in group 2.
  - Assume fair-share is group 1: 50%, group 2: 50%
  - Possible scheduling sequence:
    - A B A C A B A C
**Scheduling Algorithms**

**Lottery Scheduling**
- Give processes lottery tickets
- Randomly choose a ticket: whoever has ticket gets to run
- Some processes can have more tickets than others
- If a process holds 20% of the tickets, in the long run, it'll get 20% of the CPU
- A process can give tickets to another process
  - Example: When a client makes a blocking request to a server, it give the server its tickets. Server doesn’t normally need any tickets
- If a process doesn’t use its entire quantum, give a compensation ticket that increases its tickets by a certain amount until next time it runs
  - For example, if process A and process B each have 400 tickets, but A uses its entire quantum and B uses 1/5, then A will get 5 times as much time as B
  - When B uses only 1/5 of a quantum, give compensation ticket worth 1600. Next lottery, B has 2000, A has 400. B is 5 times more likely to win the lottery

**Lottery Scheduling Example**

Deals well with mixture of CPU-bound and IO-bound processes
- Example:
  - I/O bound processes: 10 tickets each
  - CPU-bound processes: 1 ticket each
- 2 I/O-bound processes
- 2 CPU-bound processes
- 1 I/O bound, 1 CPU-bound
- 10 I/O bound, 1 CPU-bound
- 1 I/O bound, 10 CPU-bound

**Priority Inversion**

Image three levels of priority
- High
- Medium
- Low

We want high priority processes to run before any medium or low

L (a low priority process) holds a mutex.
H (a high priority process) blocks trying to obtain the mutex
M (a medium priority process) runs since:
- H is blocked
- L is of lower priority

Meanwhile, H can’t run because it’s waiting for L

Solution:
- Priority Donation:
  - While L holds a resource, it gets (temporarily) priority of higher processes waiting for it

**Multiprocessor Operating System Types**

Master-slave multiprocessors (Asymmetric multiprocessing)

[Diagram of multiprocessor system]
Multiprocessor Operating System Types (3)

Symmetric multiprocessors (SMP)

Multiprocessor Scheduling

Uniprocessor
- What process to run?

Multiprocessor
- What process to run?
- Where to run it?

Processes
- Related/unrelated

Problems:
- Contention for single data structure
- Caching
  - If process A ran on machine B last time, some of its data may still be in B’s cache
  - Prefer to rerun on same machine
- Two-level scheduling
  - Affinity, process prefers to stay on a machine
    - Soft: only a preference
    - Hard: always stays there
  - Each machine has its own ready queue
    - Good for caching
    - Contention for single ready list gone
    - If ready queue is empty, grab a process from another machine

Space Sharing
- Related threads are scheduled together across multiple machines
  - Stay on machine until done
  - Machine idle if thread blocks on I/O
Multiprocessor Scheduling

Gang Scheduling
- Groups of related threads scheduled as a gang
- All members of a gang run simultaneously (on different CPUs)
- All gang members start and stop their time slices together

Deadlock
- A chain of processes exist that are blocked waiting on one another
  - Each process has requested a resource that another process is holding
Necessary Conditions for Deadlock

Can’t have deadlock unless we have all four conditions:

- **Mutual Exclusion**: If process A requests a resource that process B is using, process A is blocked.
- **Hold and wait**: At least one process must be holding a resource and waiting for another (blocked).
- **No preemption**: A process can’t be forced to release a process; it must do so voluntarily after it has finished its task.
- **Circular wait**: A set of resources \( P_0, P_1, \ldots, P_n \) of waiting processes must exist where \( P_0 \) is waiting for a resource held by \( P_1 \), \( P_1 \) waiting for a resource held by \( P_2 \), …, \( P_n \) waiting for a resource held by \( P_0 \).

Resource-Allocation Graph

Arrow from resource instance to process if process has resource allocated
Arrow from process to resource if process has requested resource
Ways to Deal with Deadlock

Deadlock Prevention
- Change rules so deadlock can’t happen

Deadlock Avoidance
- Check each allocation to see whether that could lead to a future deadlock situation

Deadlock Detection
- Detect
- Recover

Deadlock Prevention
Ensure no deadlocks by removing one of the 4 necessary conditions
- Mutual Exclusion
  - Some resources are sharable: read-only files, for example.
  - Others are intrinsically non-sharable (CD-burner, for example)
  - Can make some non-sharable resources sharable
  - Spooler for printer, for example
  - Can’t remove this condition for all resources, though
- Hold and Wait
  - Require each process to allocate all resources before it begins execution
  - Request resources in a group: can only request if not holding any
- No Preemption
  - If process requests a resource but must wait, it gives up all its existing resources. These are added to list of resources it is waiting for
  - Or, if process is waiting holding a resource, resource may be preempted if needed by another process. First process wakes up only when all resources it needs are available
  - Problem: some resources state can’t be maintained when taken away and then given back: in the middle of writing to a tape, for example.
- Circular Wait
  - Impose an ordering on the resources
  - Processes must request resources in increasing order

Deadlock Avoidance
Process provides additional information about their future resource usage
- for example, maximum number of resources of each type that it may need

System is kept in a safe state
- Unsafe states can lead to deadlock
- If the state is safe now, based on the knowledge of possible future requests, it can be kept safe

Safe/Unsafe states
2 Processes each acquiring two resources
- B
- Printer
- Plotter

A
- Printer
- Plotter

deadlock unsafe safe
Bankers Algorithm

Imagine a bank with:
- \( n \) customers each with a line of credit
- A certain amount of money on hand

Want to make sure that if we lend any money to a customer
- We’ll be able to satisfy all future requests up to the line of credit.
- Don’t want to be in a situation where we’ve got no money and all customers with current loans out want more money
- Leave enough money-on-hand to satisfy all needs of at least one customer (if their payback is enough to satisfy one more customer, etc.)

Deadlock Avoidance

Don’t grant resource request if it takes system to an unsafe state
- Even if the resource is currently available!

Banker’s Algorithm

Let \( n \) = number of processes, \( m \) = number of resource types
Define \( \text{Avail}[m] \): # of resources of a particular type currently available
Define \( \text{Max}[n,m] \): Maximum demand of each process (specified when process begins)
Define \( \text{Alloc}[n,m] \): current allocation of each process/type pair
Define \( \text{Need}[n,m] \): How much more a process may need (=\( \text{Max} - \text{Alloc} \))

When request is made:
- Assume temporarily that the request is granted
- Update state and determine whether it is still safe
- If not, restore the state and don’t satisfy the request

Determining whether a state is safe
- Recursively:
  - Can some process’ maximum demand be satisfied by their current allocation and the available resources?
  - If so, that process could finish, and return its resources. Update state as if resources were returned and continue
- If any processes still remaining, state is not safe

Banker’s Algorithm Example

Five process \( P_0..P_5 \)
Three resource types: A, B, C
- Ten instances of A, 5 of B, 7 of C

Current situation

<table>
<thead>
<tr>
<th>Allocation A B C</th>
<th>Max A B C</th>
<th>Need A B C</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_0 )</td>
<td>0 1 0</td>
<td>7 5 3</td>
</tr>
<tr>
<td>( P_1 )</td>
<td>2 0 0</td>
<td>3 2 2</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>3 0 2</td>
<td>9 0 2</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>2 1 1</td>
<td>2 2 2</td>
</tr>
<tr>
<td>( P_4 )</td>
<td>0 0 2</td>
<td>4 3 3</td>
</tr>
</tbody>
</table>

Assumptions:
- Maximum resource requirements for a process stated in advance
- Processes must be independent (A can’t be waiting on B for anything other than a resource it has)
- Fixed number of resources
- A process can’t exit while keeping its resources
Deadlock Detection

Detection
- Recursively:
  - Can some process’ current request be satisfied by the available resources?
  - If so, that process could finish, and return its resources. Mark it, update available resources, and continue
- Any processes that remain unmarked are deadlocked.
- Must run algorithm to detect deadlock.
  - How often?
    - When system is slow
    - Once per time period
    - Every time a resource allocation would block?

Recovery
- Process Termination
  - Abort all deadlocked processes
  - Abort one process at a time until the deadlock cycle is eliminated
- Resource Preemption
  - Take away already allocated resources from some process
  - Rollback that process to known good state?