CPU Scheduling

- Scheduling Basics
- Scheduling Algorithms
- Proportional-share and Real-time Scheduling
- Multiprocessor Scheduling

Scheduling

- OS runs one process (thread) on a CPU at a time
- Ready processes waiting in Run queue
- Scheduling:
  - When to switch between processes?
  - Which process to run next?

CPU-I/O Burst Cycle

- Most processes exhibit a sequence of CPU-I/O bursts
  - Compute for sometime
  - Block for I/O
- Processes differ in terms of lengths and frequency of these bursts
  - CPU-bound: Longer CPU bursts, few I/O bursts
  - I/O-bound: Short CPU bursts, many I/O bursts
- Choice of scheduler depends on process behavior in the system
Scheduling: When to Switch?

- Non-Preemptive: Allow running process to voluntarily give up CPU
  - Running->waiting transition (Block on I/O)
  - Running->terminated transition (Process exit)
- Preemptive: Take CPU away from a running process
  - Running->ready transition (Quantum expire)
  - Waiting->ready transition (Another process unblocks)

Preemptive Pros and Cons

- Pros:
  - Allows higher fairness, avoid runaway processes
  - Enables more important processes to run quickly
- Cons:
  - Interrupt handling becomes complex
  - Might cause poor coordination between communicating processes

Scheduling Criteria

- What metric should be optimized?
  - CPU utilization
  - Throughput: # processes finished per unit time
  - Turnaround time: Total time for a process from creation to termination
  - Waiting time: Time spent waiting in the run queue
  - Response time: Time to first response (first CPU allocation)
- What quantity to optimize?
  - Average, median, max, variance?

Scheduling Algorithms

- Which process to run next?
- Algorithms differ in their choice
  - Can have substantial effect on the desired metric
**First Come First Serve (FCFS)**

- Processes scheduled in the order of arrival
  - Non-preemptive
- What does the run queue look like?

**Impact on wait time and utilization**

- What is the average wait time for a set of processes?
- How about CPU utilization?
- Convoy effect:
  - Several I/O-bound jobs may have to wait for a CPU-bound job
  - I/O Device utilization may also be bad

**Round-Robin Scheduling**

- Run each process for a fixed time quantum
  - Preemptive
- How big should the quantum be?
  - Context switch overhead
  - CPU burst size
  - Interactive vs. batch processing
- Processor Sharing: idealized algorithm
  - Assumes really small quantum size, no scheduling overhead
  - $n$ processes $\Rightarrow$ Effective $1/n$ CPU speed

**Shortest Job First (SJF)**

- Schedules the shortest job on the CPU
- What about the average waiting time?
  - Can you do better?
- Shortest remaining processing time first (SRPT):
  - Preemptive version
- Problem: How do you know the burst length a priori?
Priority Scheduling
- Execute in priority order
- Can be preemptive or non-preemptive
- How to determine priorities?
  - External: Assigned by user, admin based on process importance
  - Internal: Assigned by OS based on process behavior, attributes

Starvation Problem
- How long will it take for a low priority process to run?
- Aging: Technique to avoid starvation
  - Increase priority of waiting processes

Multi-level Queue Scheduling
- Partition the run queue into multiple queues
  - Each queue has a different class of processes
  - E.g.: interactive, batch, real-time, ...
- Two-level scheduling:
  - Select a queue
  - Schedule a process from the selected queue
- Queue selection:
  - Priority-based
  - Time-slicing
  - Proportional-share
- Intra-queue scheduling: Class-based scheduler

Multi-level Feedback Queue Scheduling
- Same idea as Multi-level Queue Scheduling
- Allows processes to move between different queues
- Feedback-driven:
  - Monitor process's behavior
  - E.g.: Demote a CPU-bound process
  - E.g.: Promote a low priority process through aging
Proportional-Share Scheduling
- Provides weighted fairness
- Assign weights to processes
- Processes get CPU service in proportion to weights

Generalized Processor Sharing
- Generalization of Processor Sharing
- Achieves perfect proportionate allocation
  - Each process gets a virtual CPU proportional to its weight
- Practical algorithms:
  - Weighted Fair Queuing, Start-time Fair Queuing, Deficit Round Robin, ...

Lottery Scheduling
- Randomized proportional-share scheduling algorithm
- Each process allotted lotteries in proportion to its weight
- Scheduling:
  - Conduct lottery
  - Lottery winner gets to run

Lottery Scheduling: Properties
- Probabilistically Fair
  - Expected allocation proportional to weight
  - No guarantee on actual allocation
- Expected throughput: Proportional to weight
- Expected response time: Inversely proportional to weight
- No starvation
Real-Time Scheduling

- Used more generally in Real-time OS
  - Timing is critical
  - A process must run/finish before a deadline
- Examples:
  - Embedded devices. E.g.: automobiles
  - Cyber-physical systems: E.g.: robots
  - Multimedia servers. E.g.: Video streaming

Periodic Processes

- Each process is characterized by:
  - Period of work generation $P_i$
  - Processing time $C_i$
  - Deadline to finish the work $D_i$
- Question: When should each process be scheduled so that all processes finish their work before their deadlines?
- Feasibility condition: $\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$

Example Periodic Processes

- Processes A, B, C:
  - Periods 30, 40, 50
  - Work/period: 10, 15, 5

Real-Time Scheduling Algorithms

- Static:
  - Each process is assigned a static priority
  - Higher priority process always preempts lower-priority process
  - E.g.: Rate-Monotonic Scheduling
  - Static priorities work under low utilization
- Dynamic:
  - Priorities are chosen dynamically
  - Currently highest priority process would run
  - E.g.: Earliest Deadline First
  - Dynamic priorities work under any feasible utilization
Rate-Monotonic Scheduling
- Used for periodic processes with fixed frequency, same amount of work per period
- Static Priority = Frequency
- RMS guaranteed to work only if utilization < ~ln2

Earliest Deadline First
- Dynamic Algorithm
- Processes run in the order of their deadlines
- Does not require:
  - Processes to be periodic
  - Have same amount of work every period
  - Any process can come in any time with any work amount and deadline
- EDF always works if utilization <= 1

RMS vs. EDF

Multiprocessor Scheduling
- Uniprocessor: Which process to run next?
- Multiprocessor: Which process to run next on which CPU?
- Asymmetric multiprocessing: Master-slave model
  - Master CPU executes OS code, others execute user code
- Symmetric multiprocessing (SMP)
  - All CPUs can run OS/user code
**SMP Scheduling**
- Run queue organization:
  - Global run queue vs. private run queue
- Scheduling decision can depend on:
  - Load on CPUs
  - Cache affinity of a process

**Global Scheduling**
- Single run queue
  - Idle CPU picks the next process from the queue
  - Could use priority, FCFS, SJF, etc.
  - Problem?

**Local Scheduling**
- Each CPU has its own private run queue
  - A process is created and scheduled on a CPU
  - Each CPU runs a local scheduler
  - Problem?

**Load Balancing**
- Two-level scheduling
  - Each CPU performs scheduling on private queue
  - Balance load occasionally
- How to define load?
  - Number of processes
  - CPU usage
  - Total Priority
- When to load balance?
  - Push migration: Periodically balance load
  - Pull migration: Idle CPU pulls load from other CPUs
  - Linux scheduler does both
Affinity Scheduling
- Processes have cache affinity with recent CPUs
- Would like to reschedule on same CPU
- Types of affinity:
  - Soft: Likely to run on same CPU but not guaranteed
  - Hard: Guaranteed to run on a set of CPUs (Pinning)
- What could be the impact on overall system utilization?

NUMA-Awareness
- NUMA architectures:
  - Large number of CPUs and multiple memory banks
  - Memory access latency is non-uniform
  - CPU scheduler and memory manager have to work together
    - Place process close to its data or vice versa

Domain-based Scheduling
- Load balancing and affinity scheduling are in conflict
- Scheduling domain: Set of cores that can be balanced among themselves
  - Cores on the same CPU
  - CPUs sharing the same cache
  - CPUs on the same NUMA-node
- Hierarchical domain-based scheduling: Balance load at the lowest domain first
- Example: Linux CFS

Heterogeneous Multiprocessing
- CPUs may have different speeds. Why?
  - Different types of CPUs in the same system
  - Dynamic frequency scaling for power savings. E.g.: mobile systems
- big.LITTLE architecture:
  - Big CPUs: Faster, but more power-hungry
  - Little CPUs: Slower, but more energy-efficient
- Scheduling: Based on performance needs
  - E.g.: interactive apps on Big CPUs, background apps on Little CPUs
Algorithm Evaluation

- Analytic Modeling
- Simulation
- Implementation

Analytic Modeling

- Queuing Models: Represent a compute system as a network of servers each with its queue
  - Input: distributions about arrival rates and service rates
  - Output: expected response time, throughput, etc.
- Little’s Law: Relation between average queue length, arrival rate, and average waiting time
  \[ n = \lambda W \]

Simulation

- Program that models major components of a system and replays system execution
- Process arrivals/departures could be generated using
  - Distributions
  - Traces of real systems

Implementation

- Build it in a real OS and then test
  - E.g.: in a VM
  - Could also be deployed in production
- Goal: Make the scheduler flexible and extensible
  - E.g.: use mechanisms vs. policies, such as in priority scheduling