Scheduling

Scheduling Overview

1. CPU Scheduler: Decides how and when processes/threads access shared CPUs
    - Choose one ready task to run on CPU
    - Runs when…
        - CPU becomes idle
        - New task becomes ready
        - Timeslice expired timeout
    - After a thread is selected, it is dispatched on the CPU
        - Context switch, enter user mode, set PC, go!
2. Assign tasks immediately (first come first served (FCFS))
3. Assign simple tasks first to maximize throughput (shortest job first (SJF))
4. Assign complex tasks first (maximize utilization of CPU, devices, memory)
5. Scheduler needs to consider…
    - Which task should be selected? Scheduling policy/algorithm
    - How is this done? Depends on runqueue data structure

Run to Completion Scheduling

1. As soon as task is assigned to CPU, it will run until it completes
2. Initial assumptions:
    - Group of tasks/jobs
    - Known execution times
    - No preemption
    - Single CPU
3. Metrics:
    - Throughput
    - Average job completion time
    - Average job wait time
    - CPU utilization
4. T1 = 1s, T2 = 10s, T3 = 1s (arrive T1, T2, T3)
    - First-Come First-Serve (FCFS)
        - Schedules tasks in order of arrival
        - Organize runqueue as a queue structure (FIFO)
        - Throughput = 3 / (1+10+1) = 0.25 tasks/second
        - Average completion time = (1+11+12) / 3 = 8 seconds
        - Average wait time = (0+1+11) / 3 = 4 seconds
    - Shortest Job First (SJF)
        - Schedules tasks in order of their execution time (T1->T3->T2)
        - Organize runqueue as an ordered queue structure or tree
        - Throughput = 3 / (1+10+1) = 0.25 tasks/second
        - Average completion time = (1+1+11) / 3 = 4 seconds
        - Average wait time = (0+1+2) / 3 = 1 seconds

Preemptive Scheduling

1. Tasks don't have to arrive all at once
2. Impossible to precisely know the runtime of a job
    - Use heuristics based on history to predict
    - How long did a task run last time?
    - How long did a task run last n times? (windowed average)
3. Priority Scheduling: Tasks have different priority levels
    - Run highest priority task next (preempt)

- Use a different runqueue structure for each priority level or tree based on priority of task
- Starvation: Low priority task stuck in runqueue
- Priority aging: Priority is a function of actual priority and time spent in runqueue, so eventually task will run -> prevents starvation

Priority Inversion

1. If a lower priority thread holds the lock of a higher priority thread, the higher priority thread can block and finish after the lower priority thread
2. Priority: T1, T2, T3
3. Order of execution: T2, T3, T1 (inverted)
4. Solution: Temporarily boost priority of mutex owner so it finishes
    - Lower priority when mutex is released

Round Robin Scheduling

1. Pick up first task from queue
2. Task may yield to wait on I/O (unlike FCFS), so schedule a different task
3. Can apply priorities with preemption
4. Timeslicing: Interleaving tasks

Timesharing and Timeslices

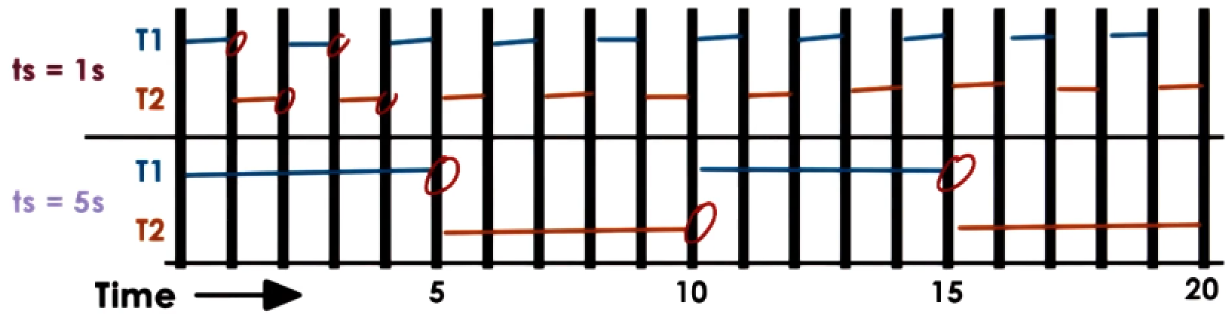
1. Timeslice: Maximum amount of uninterrupted time given to a task
    - Also referred to as a time quantum
2. Task may run less than timeslice time
    - Has to wait in I/O, synchronization -> will be placed on queue
    - Higher priority task becomes runnable
3. Using timeslices, tasks are interleaved (timesharing the CPU)
    - For CPU bound tasks, preempted after timeslice
4. Shown below, obtain good performance without requiring knowledge of task runtimes
5. Pros:
    - Shortest jobs finish first (low completion time)
    - More responsive (low wait time)
    - Lengthy I/O operations initiated earlier
6. Cons:
    - Overhead associated with context switching frequently
7. Length of timeslice » context switch time

Algorithm	Throughput	Average Wait	Average Completion
FCFS	0.25 req/s	4 seconds	8 seconds
SJF	0.25 req/s	1 seconds	5 seconds
RR(ts=1)	0.25 req/s	1 seconds	5.33 seconds

Time Slice Length

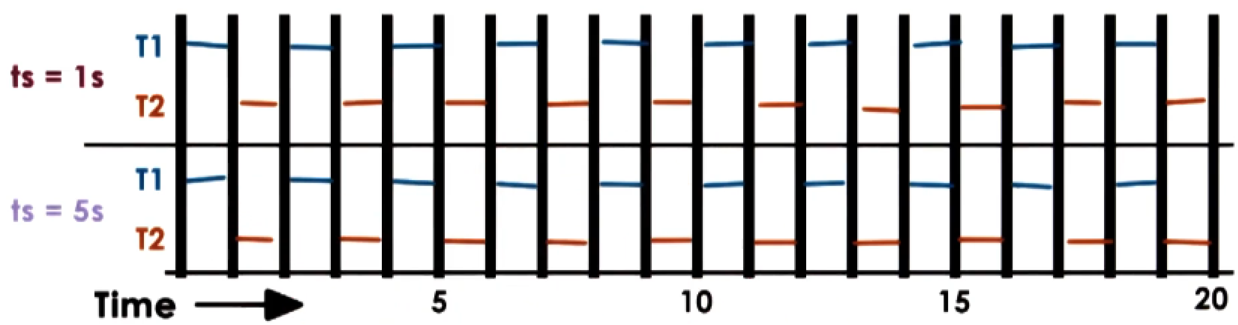
1. 2 tasks, execution time = 10s
2. Context switch time = 0.1s
3. I/O operations issued every 1s
4. I/O completes in 0.5s
5. For CPU-bound tasks, longer timeslices are preferred
    - Limits context switching overhead
    - Keeps CPU utilization and throughput high

6. For I/O-bound tasks, shorter timeslices are preferred
    - I/O-bound tasks can issue I/O operations earlier
    - Keeps CPU and device utilization high
    - Better user-perceived performance
7. CPU utilization = (cpu_runtime / (cpu_runtime + ctx_switch_overhead)) * 100



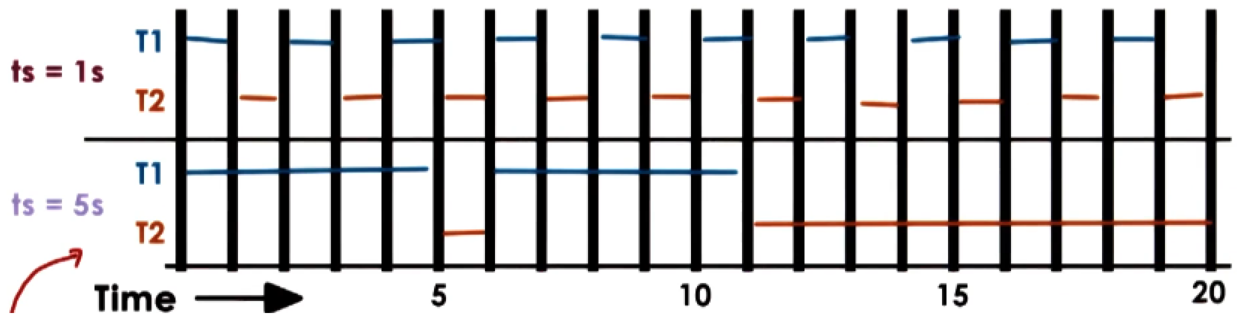
CPU-Bound Timeslice Scheduling

Algorithm	Throughput	Average Wait	Average Completion
RR(ts=1)	.091 req/s	0.55 seconds	20.85 seconds
RR(ts=5)	.098 req/s	3.05 seconds	17.75 seconds



I/O-Bound Timeslice Scheduling

Algorithm	Throughput	Average Wait	Average Completion
RR(ts=1)	.091 req/s	0.55 seconds	20.85 seconds
RR(ts=5)	.091 req/s	0.55 seconds	20.85 seconds



T1 CPU-Bound, T2 I/O-Bound Timeslice Scheduling

Algorithm	Throughput	Average Wait	Average Completion
RR(ts=1)	.091 req/s	0.55 seconds	20.85 seconds
RR(ts=5)	.082 req/s	2.55 seconds	17.75 seconds

Runqueue Data Structure

1. If we want I/O and CPU bound tasks to have differen timeslice values. . .
   - Same runqueue, check type
   - Two different structures
2. Three queues
   - Most I/O intensive tasks, shortest timeslice
   - Mix of I/O and CPU intensive tasks, medium timeslice
   - Most CPU intensive tasks, longest timeslice (FCFS)
3. Pros:
   - Timeslicing benefits for I/O bound tasks
   - Timeslicing overheads avoided for CPU bound tasks
4. How do we know where to store tasks?
   - Make one continuous data structure, not three separate
   - When task enters, put it in shortest timeslice queue
   - If task yields voluntarily, leave it at this level
   - If task uses entire timeslice, push down a level
   - If task in lower level repeatedly releases CPU, move up a level
   - Called Multi-Level Feedback Queue (Fernando Corbato)
     - MLFQ != Priority queues (feedback mechanism)
     - Linux O(1) scheduler uses ideas from MLFQ

Linux O(1) Scheduler

1. O(1) == Constant time to select/add task, regardless of task count
2. Preemptive, priority-based scheduler with 140 priority levels
   - Real time: 0-99
   - Timesharing 100-130
   - User processes receive timesharing priority levels
     - Default 120
     - Nice value (-20 to 19)
3. Timeslice Value
   - Depends on priority (smallest for low priority, largest for high)
4. Feedback
   - Sleep time: waiting/idling

    - Longer sleep -> Interactive (priority -= 5 to boost priority)
    - Shorter sleep -> Compute intensive (priority += 5 to lower priority)
5. Runqueue
    - Two arrays of tasks
    - Active: Used to pick next task to run
        - Used to pick next task to run
        - Constant time to add/select
        - Tasks remain in queue in active array until timeslice expires
    - Expired
        - Inactive list: Scheduler won't select if there are tasks in active
        - When no more tasks in active array, swap active and expired
6. Introduced in 2.5 by Ingo Molnar
    - Affected performance of interactive tasks significantly (streaming)
7. As workloads changed, O(1) was replaced by CFS in 2.6.23

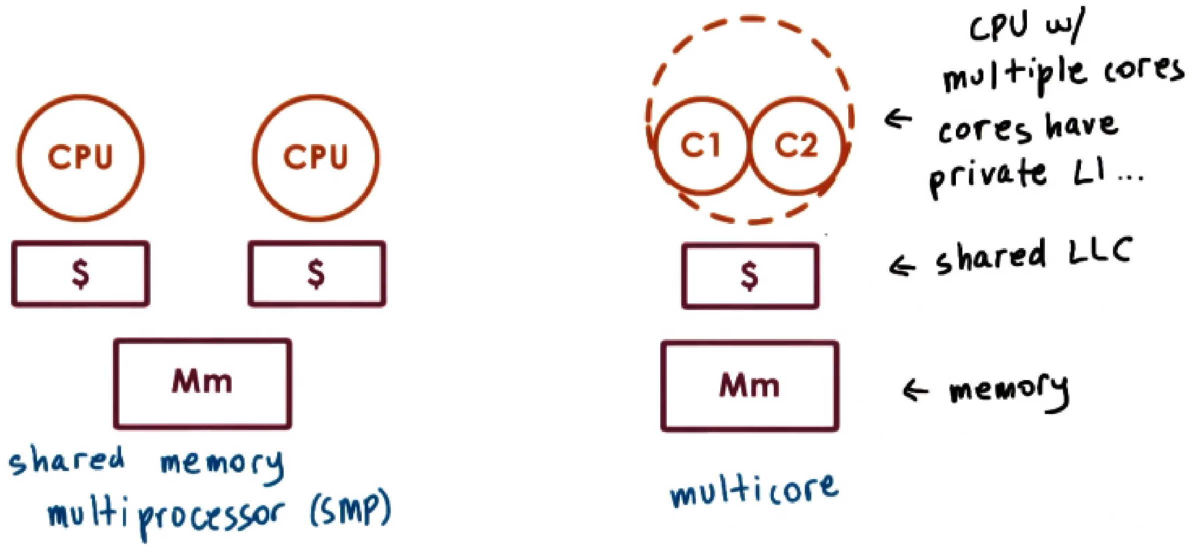
Linux CFS Scheduler

1. With O(1) scheduler, performance of interactive tasks is subpar because they spend more time in the expired queue
2. Fairness: Tasks should run for an amount of time proportional to priority
3. CFS: Completely Fair Scheduler
4. Uses red-black tree as runqueue structure (self-balancing)
    - Ordered by "vtruntime" (virtual runtime == time spent on CPU)
5. CFS Scheduling: Always pick leftmost node (shortest vruntime)
    - Periodically adjust vruntime
    - Compare to leftmost vruntime
        - If smaller, continue running
        - If larger, preempt and place appropriately in the tree
    - vruntime progress rate depends on priority and niceness
        - Rate progresses faster for low priority
        - Rate progresses slower for high priority
    - Uses same tree for all priorities
6. Performance
    - Select task: O(1)
    - Add task: O(logN)

Scheduling on Multiprocessors

1. Multi-CPU Architecture (shared memory multiprocessor (SMP))
    - Each CPU has private L1, L2, ... on-chip caches
    - Last Level Cache (LLC) may or may not be shared
    - Memory is shared
2. Multi-Core Architecture
    - Each core has private L1, L2, ... caches
    - LLC is shared among cores
    - Memory is shared
3. Cache affinity is important; schedule task on same CPU (hot cache)
    - Keep tasks on same CPU as much as possible
    - Hierarchical scheduler architecture
    - Per-CPU runqueue and scheduler
        - Load balance across CPUs based on queue length
        - Or when CPU is idle
4. NUMA: Non-Uniform Memory Access
    - Multiple memory nodes

- Memory node closer to a “socket” of multiple processors
  - Access to local memory node faster than access to remote node
- Keep tasks on CPU closer to memory node where their state is



Scheduling Multiprocessors

Hyperthreading

1. Multiple hardware-supported execution contexts (multiple sets of registers per thread)
   - Still one CPU, but with very fast context switch
   - Hardware multithreading
   - Hyperthreading
   - Chip multithreading (CMT)
   - Simultaneous multithreading (SMT)
2. If time_idle > 2 * time_context_switch, switch contexts to hide latency
   - SMT context switch: Order of cycles
   - Memory load: Order of 100s of cycles
3. Hyperthreading can mask memory access latency

Scheduling for Hyperthreading Platforms

1. Assumptions:
   - Thread issues instruction on each cycle (max instruction/cycle=1)
   - Memory access takes 4 cycles
   - Hardware switching instantaneous
   - SMT with 2 hardware threads
2. If we co-schedule CPU-bound threads, every other cycle is wasted as only one thread can execute at a time; memory is also idle
3. If we co-schedule I/O-bound threads, the CPU is idle, wasting cycles
4. Mixing CPU- and I/O-bound threads…
   - Allows both the CPU and memory to be utilized
   - Avoid/limit contention on processor pipeline
   - Still some interference and degradation, but minimal

CPU-Bound or Memory-Bound?

1. Use historic information
   - "Sleep time" won't work as the thread is not sleeping when waiting on memory access
   - Software takes too much time to compute
   - Need hardware-level information
2. Hardware counters
   - L1, L2, . . . , LLC misses
   - IPC
   - Power and energy data
   - Used to estimate what kind of resources a thread needs
3. Software interface and tools
   - oprofile, Linux perf, . . .
4. Schedulers can make informed decisions using hardware counters
   - Typically use multiple counters
   - Models with per-architecture thresholds
   - Based on well-understood workloads

Scheduling with Hardware Counters

1. Is cycles per instruction (CPI) useful?
   - Memory bound -> High CPI
   - CPU bound -> 1 (or low) CPI
2. Computing 1/IPC requires software computation, so not acceptable
   - Instead, simulation-based evaluation
3. Testbed
   - 4 cores x 4-way SMT
   - Total of 16 hardware contexts
4. Workload
   - CPI of 1, 6, 11, 16
   - 4 threads of each kind
5. Metric is instructions per cycle
   - Max IPC = 4

CPI Experiment Results

1. Mixed CPI -> processor pipeline well-utilized -> high IPC
2. Same CPI -> Contention on some cores results in wasted cycles
3. However, realistic workloads don't vary from 1-16
   - 2-4 is much more realistic
   - CPI isn't a particularly useful metric
4. Takeaways
   - Resource contention in SMTs for processor pipeline
   - Hardware counters can be used to characterize workload
   - Schedulers should be aware of resource contention, not just load balancing
   - LLC usage would have been a better choice