What is Scheduling?

• When a thread is suspended...
  • Kernel must decide which thread gets to run next.
    – Only runnable threads considered.
      • Keep in mind: most threads are sleeping most of the time.
  • Issues to consider
    – Thread priority
    – Thread history
      • Interactive threads usually given attention ASAP on the theory
        that they will probably sleep again quickly anyway.
      • Keeps the user interface responsive.
    – Number of processors
      • Often desirable to schedule a thread on the same processor it
        was using in the past.
CPU vs I/O

- Threads alternate between using the CPU and doing I/O.
  - Here "I/O" also covers the case where a thread waits for another thread.
    - Waiting to acquire a lock.
    - Waiting for another thread to terminate.
  
  - CPU Burst: Time spent running on CPU
  - I/O Burst: Time spent waiting for "I/O"

- Scheduling is only concerned with threads that can use the CPU
  - That is, threads involved in or about to start a CPU burst.
Single CPU, Single Queue

Queue of runnable jobs

CPU

Preempt (Time Quantum Expires)

Explicit Yield

Waiting for I/O

I/O Completes

Sleeps waiting for I/O

Terminates
Run Queue

• The queue of runnable jobs is called the *run queue* or *wait queue*.

• Scheduling problem:
  • When the CPU is idle...
    – Because executing job terminated
    – Because executing job sleeps on I/O
    – Because executing job is preempted
    – Because executing job explicitly yielded the processor
  • ... which job from the run queue should be selected next?
    – This is the essence of the scheduling problem.
Run Queue Empty?

• If the run queue is empty (surprisingly common)
  • CPU idles
  • Most modern systems actually shut it off!
    • Although not fully off.
      – Conserves energy.
      – Keeps the system cooler.
  • CPU turned back on by the next interrupt.
    – This works...
      • If no jobs can run, the only thing that can happen next is a prior I/O request completing.
      • Hardware will generate an interrupt when that occurs.
      • Interrupt service routine will wake up some process, giving the scheduler something to think about.
Run Queue Not Empty

• If the run queue is not empty...
  • Several different algorithms exist for selecting the next job. Basics include...
    – FCFS (First Come First Served; also called FIFO)
      • Executes jobs in the order in which they were entered into the run queue.
    – SJN (Shortest Job Next)
      • Executes the shortest job next regardless of order in run queue.
      • Requires a way to predict which will be the shortest.
    – SRT (Shortest Remaining Time)
      • Similar to SJN.
      • Preempt current job if something shorter arrives on the queue.
Job Time

- Here "Job Time" means the time of the next CPU burst.
  - Example: Jobs A, B, C in the queue in that order.
    - A's next CPU burst will be 3.7 ms
    - B's next CPU burst will be 9.8 ms
    - C's next CPU burst will be 2.5 ms
  - In that case...
    - FCFS chooses A (at the head of the queue)
    - SJN chooses C
    - SRT chooses C as well but will replace job on the CPU if something shorter is added to the queue while C is running (note: C's burst will be shorter by then too).
CPU Bound

- Some jobs have very long CPU bursts
  - Lasting minutes, days, months...
- Typically split into time quantaums and preempted periodically.
  - For example, every 10 ms.
  - Some systems adjust time quantum size dynamically.
- Scheduler may assume next CPU burst is the size of the time quantum.
  - But may also take into account history.
    - If a job uses its entire quantum every time it runs, it may be penalized (get a forced priority reduction).
Turn Around Time

- **Normalized Turn Around Time**, $T_n$
  - $T_n = (\text{TimeInQueue} + \text{TimeExecuting}) / \text{TimeExecuting}$
  - Example: 18.5 ms in run queue. 2.7 ms executing.
    - $T_n = (18.5 + 2.7)/2.7 = 7.85$
  - Low $T_n$ is good.
    - Ideally $T_n = 1.0$ (zero time in the run queue).

- **Average Normalized Turn Around Time**...
  - A figure of merit for a scheduler.
    - Average of $T_n$ across every job. You want 1.0.
FCFS

- First Come First Served
  - Easy to implement.
    - Scheduler pulls job from the front of the queue. Done.
  - Lousy average $T_n$
    - Problem: Short jobs that wait experience a huge $T_n$
      - $(250 \text{ ms} + 1 \text{ ms})/1\text{ms} = 250$
      - McDonalds: You walk in behind a bus load of people who each order a huge meal. You just want a soda.
  - FCFS is fair.
    - Everyone will get a turn... eventually.
SJN

- **Shortest Job Next**
  - Scheduler scans the queue looking for the job with the shortest estimated service time. Runs it immediately.
  - Much better average $T_n$
    - Short jobs don't have to wait.
    - "You just want a soda? Come to the head of the line!"
  - Long jobs might starve.
    - At McDonald's starvation might be literal!
    - Not always fair.
Estimated Service Time

- SJN requires estimates of a job's service time.
  - Use past behavior.
  - Processes burst on the CPU then sleep.
    - Build up a history of a process's CPU burst durations.
    - Use that history to form guess of future behavior.
    - Not always accurate (of course)
    - Often very close.
  - Different ways to compute estimate can produce different estimates
    - ... can change the performance of basic SJN scheduling.
Real Operating Systems

• Real systems are more complex.
  • Multiple queues... one for each priority.
    – Typically pull job from highest priority queue.
    – Only consults lower priority queues if the high priority queue is empty.
      • Not as bad is it sounds: high priority jobs are typically not CPU bound and usually are waiting for I/O. High priority queues are normally empty.
    – BUT... will bump up process priority automatically (to avoid starvation of low priority processes).
Multiple CPUs

- Real systems have more than one CPU
  - This doesn't change things that much.
  - Whenever any CPU is idle, the scheduler steps in to give it something to do.
    - Can use the same basic algorithms.
    - Sometimes useful to bind a process to the same CPU (to make use of memory cache more efficient).
- Goal: Keep all CPUs busy all the time.
  - Otherwise you are wasting your money!
• High level overview...
  • Scheduler works with "schedulable entities."
    – Each such entity needs a struct sched_entity.
      • Such a structure is embedded in the task_struct of each task.
    – Allows groups of threads to be scheduled as a unit.
      • All threads owned by a particular user.
      • All threads in a particular process.
      • Once the unit is scheduled, then the component tasks can be.
  • Different "scheduling classes" are supported.
    – "Completely fair scheduler" is the default.
    – Also a real-time scheduler to handle SCHED_RR and SCHED_FIFO policies.
    – Each class works independently of the other(s).
Linux

• High level overview (continued)...
  • Each CPU has a run queue of its own.
    – The CPU run queue tracks total execution time on CPU
    – Contains class-specific run queues for each class.
  • A task is in exactly one run queue.
    – Waiting on exactly one CPU...
    – Handled by exactly one scheduling class.
  • Under special circumstances tasks can change run queues.
    – Switch to a different scheduling class.
    – Migrate to a different CPU (be careful!)
Linux

High level overview (continued)...

"Virtual" run time tracked for each task.

- Updated when task pulled from CPU or at each timer tick.
  - Timer ticks HZ times per second. Default is 250 (4 ms tick interval).
  - Only currently executing tasks (on each CPU) needs updating.
- Weighted by task priority.
  - High priority tasks have virtual run times that advance slowly.
  - Scheduler believes they haven't run very much and runs them again sooner than otherwise.

No time quanta in the usual sense.

- Task preempted from CPU if virtual run time is too high.
Completely Fair Scheduler

• Tries to ensure all tasks get the same (virtual) run time.
  - High priority tasks get more real time since their virtual run time advances more slowly.

• Basic idea: Pick the task with the smallest virtual run time to run next.
  - Task may not be preempted at each timer interrupt, but it will be preempted eventually.
  - New tasks get more attention because their virtual run times are small initially.
    • Interactive tasks automatically preferred over CPU bound tasks. No special handling of interactive tasks is necessary.
Real Time Scheduler

• Real time class is independent.
  • Threads considered before any CFS threads.
  • SCHED_FIFO
    – Thread runs for as long as it wants. All other threads on the system are suspended indefinitely.
      • Of course such threads should sleep quickly.
    – Important if real time deadlines are to be met.
  • SCHED_RR (Round Robin)
    – Threads switch among themselves, blocking all other threads on the system indefinitely.
• BUT... there are real time priorities to consider also.
Real Time Priorities

- CFS threads can be temporarily boosted to real time priority...
  - Using RT Mutexes.
  - Intended to avoid "priority inversion."
    - See the slide set on locking.
  - Still scheduled by the CFS (as I understand it).
CFS Run Queue

- The CFS uses a red black tree for its run queue.
  - Sorted in order of increasing virtual run time.
    - Not exactly... but this is the general idea.
  - Next task to run is the leftmost tree node.
- R/B trees have $O(\log n)$ running time for most operations.
  - Here 'n' is the number of runnable tasks.
  - Older 2.6.x kernels used an $O(1)$ scheduler.
    - Now obsolete. Required a lot of special case handling and complex heuristics.