Today

https://pollev.com/sprenkle

- Process Scheduling
  - Review and conclusions
- Cooperating Processes
  - Interprocess Communication

Review

- What are the goals for scheduling policy?
  - How do we measure the goodness of scheduling policies?
- What are examples of scheduling policies?
  - What are their characteristics?
  - What are their tradeoffs?
- What is the best scheduling policy?

Review: Scheduling Metrics/Policy Goals

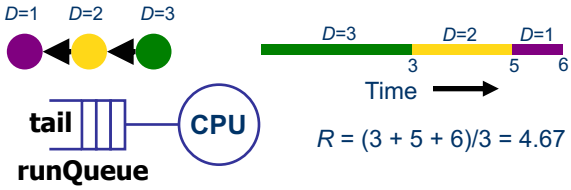
- CPU Utilization
  - percentage of time CPU is being used (not idle)
- Response (or turnaround) time or latency, responsiveness
  - How long does it take to complete a task or request? (R)
    - Typically concerned with average
  - Say a task takes D time units of work (its service demand)
    - But how long does it spend waiting for service?
- Throughput
  - How many tasks/requests complete per unit of time? (X)
- Fairness
  - how well is the CPU distributed among processes
- Meet deadlines, reduce jitter for periodic tasks
  - e.g., videos and other continuous media

CPU Scheduling: There is no one-size-fits-all "best" policy…

- Depends on the goals of the system.
- Often have multiple (conflicting) goals or primary metrics

Review: FCFS

- **Throughput.** FCFS is as good as any non-preemptive policy.
- **Fairness.** FCFS is intuitively fair…sort of.
  - "The early bird gets the worm"…and everyone is fed…eventually.
- **Response time.** Long jobs keep everyone else waiting.
  - Consider service demand (D) for a process/job/thread.



Review: Non-Preemptive vs Preemptive

- Depending upon which scheduling opportunities are used by a scheduler, the scheduling can be:
  - **Non-Preemptive**: The scheduler will allow the running process to continue to run as long as it remains ready (i.e., doesn't block or exit).
  - **Preemptive**: The scheduler may set aside the running process in favor of another at any scheduling opportunity
    - Enables time-sharing, priority scheduling

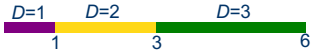
Review: Round Robin

- **Response time.** RR reduces response time for short jobs.
  - For a given load, wait time is proportional to the job's total service demand D.
- **Fairness.** RR reduces variance in wait times.
  - But: RR makes jobs wait for jobs that arrived later.
- **Throughput.** RR imposes extra context switch overhead.
  - Degrades to FCFS-RTC with large Q.

Review: Minimizing Response Time: SJF (STCF)

- **Shortest Job First** (SJF) is provably optimal if the goal is to minimize average-case R.
  - Also called Shortest Time to Completion First (STCF) or Shortest Remaining Processing Time (SRPT)
- Idea: get short jobs out of the way quickly to minimize the number of jobs waiting while a long job runs.
  - Intuition: longest jobs do the least possible damage to the wait times of their competitors.

Any limitations?



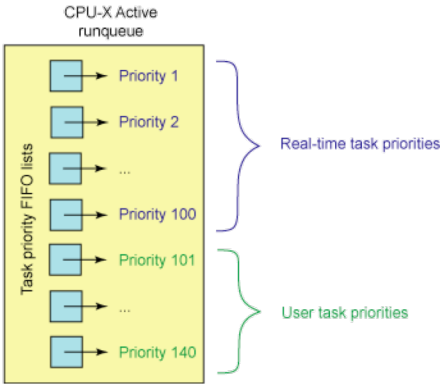
R = (1 + 3 + 6)/3 = 3.33

Could starve long-running processes

Priority

- Most modern OS schedulers use ***priority*** scheduling
  - Each task/process has a priority value (integer)
  - The scheduler favors higher-priority process
  - User-settable relative importance within application
  - Internal priority adjustments as an implementation technique within the scheduler.
  - How to set the priority of a process?
- How many priority levels?
  - 32 (Windows) to 128 (OS X)

Ordering Runqueues by Priority



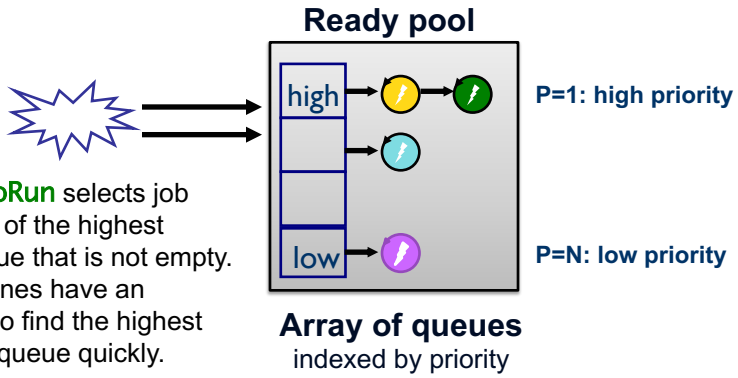
ready queue (runqueue)

In real systems, the simple "cartoon ready queue" may be a ***multi-level queue***: an ordered array of queues, one for each priority level.

In a typical OS, each thread has a ***priority***.
When a core is idle, pick a thread with highest priority.
If a higher-priority thread becomes ready, then preempt the thread currently running on the core and switch to the new thread.

Multi-level queue

Multi-level priority queue structures are commonly used in OSs to represent the run queue == ready pool == ready list.



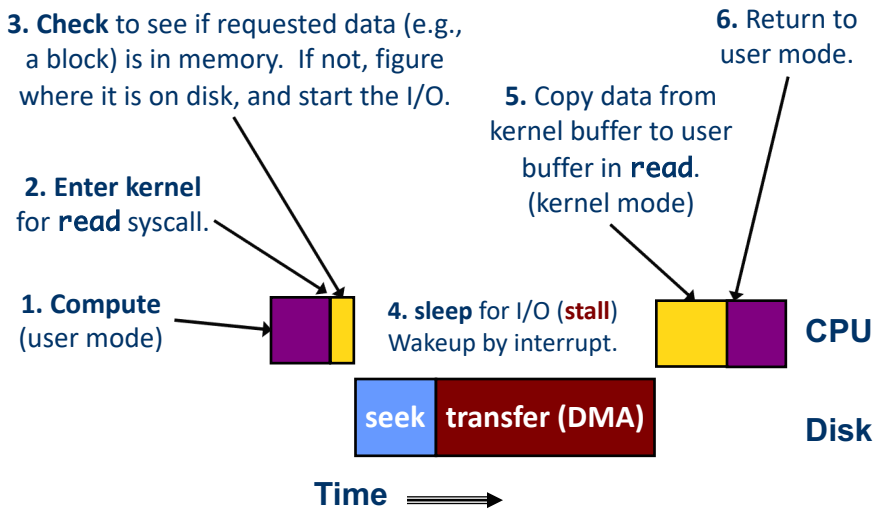
`GetNextToRun` selects job at the head of the highest priority queue that is not empty. Most machines have an instruction to find the highest non-empty queue quickly.

constant time, no sorting

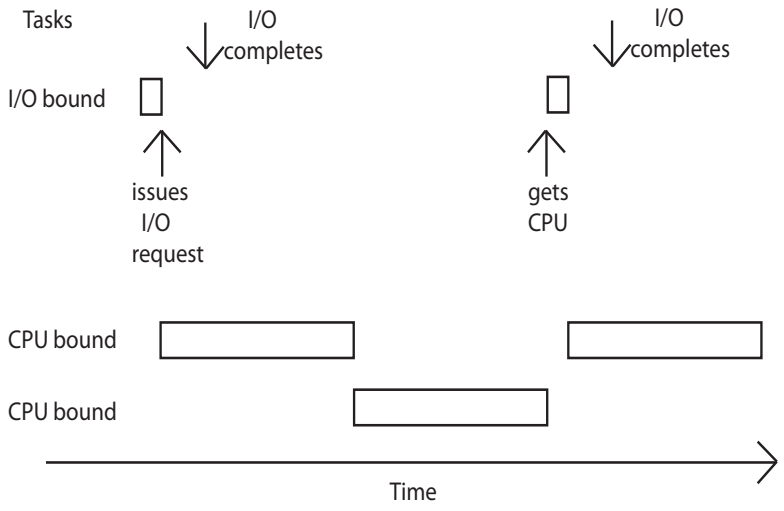
The Process Mix

- Two broad classes of processes:
  - **CPU Bound**: A process that is spending most of its time doing CPU operations.
  - **I/O Bound**: A process that is spending most of its time doing I/O operations.
- Processes can switch between being CPU Bound and being I/O Bound during their execution

Anatomy of a read

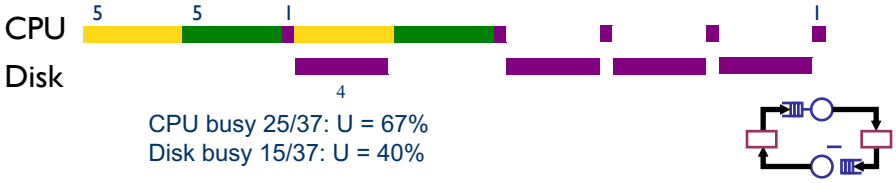


Mixed Workload



Two Schedules for CPU/Disk

1. Naive **Round Robin**



CPU busy 25/37: U = 67%
Disk busy 15/37: U = 40%

2. Add internal priority boost for I/O completion

CPU busy 25/25: U = 100%
Disk busy 15/25: U = 60%

33% improvement in utilization
When there is work to do,
U == efficiency.
More U means better throughput.

Based on this example, what would make a better scheduling algorithm?

More Realistic General-Purpose Policy

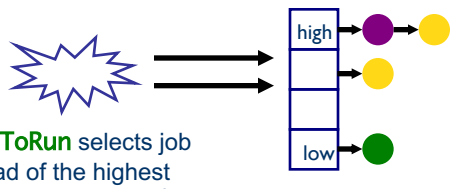
- Special class gets special treatment
  - varies – requires configuration
- Everything else: *roughly* equal time quantum
  - "Round robin"
  - Give priority boost to processes that frequently perform I/O
  - Why?
- "I/O bound" processes frequently block.
  - If we want them to get equal CPU time, we need to give them the CPU more often.

Estimating Time-to-Yield

- How to predict which job/task/thread will have the shortest demand on the CPU?
  - If you don't know, then guess
    - Weather report strategy: predict future D from the recent past
- We can guess well by using adaptive internal priority
  - Common technique: multi-level feedback queue
  - Set N priority levels, with a timeslice quantum for each
  - If thread's quantum expires, drop its priority down one level
    - "It must be CPU bound." (mostly exercising the CPU)
  - If a job yields or blocks, bump priority up one level
    - "It must be I/O bound." (blocking to wait for I/O)

Multilevel Feedback Queue: MFQ

- Used by many systems (e.g., Unix variants) implement internal priority
- **Multilevel**. Separate queue for each of N priority levels.
  - Use RR on each queue
  - Look at queue i+1 only if queue i is empty
  - Run selected process for 2^i quanta (for queue i)
- **Feedback**. Factor previous behavior into new job priority.



I/O bound jobs

jobs holding resources
jobs with high external priority

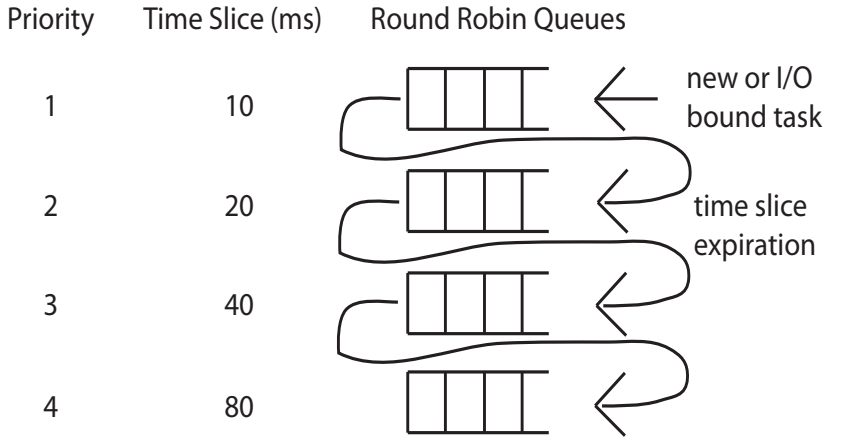
CPU-bound jobs

`GetNextToRun` selects job at the head of the highest priority queue: constant time, no sorting

ready queues
indexed by priority

Priority of CPU-bound jobs decays with system load and service received.

Multilevel Feedback Queue: MFQ



Effect of this model on our metrics?

MFQ Tradeoffs

Benefits

- High CPU utilization
- Fewer context switches
  - If you need more CPU, you get more CPU (overtime)
- Auto-adjust priorities

Limitations

- Time to look through the queues for a process to run
- "fairness"?

Linux's "Completely Fair Scheduler"
(default since Oct 2007)

- "real time" process classes – always run first (rare)
- Other processes:
  - Red-black BST of processes, organized by CPU time they've received
  - Pick the ready process that has run for the shortest (normalized) time thus far
  - Run it, update its CPU usage time, add to tree
- Interactive processes: Usually blocked, low total run time, high priority.

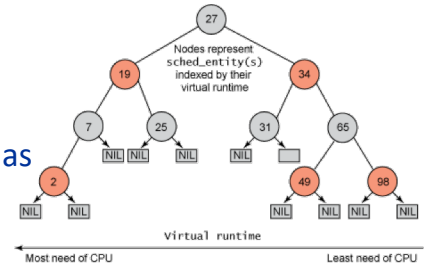


Image source: https://www.ibm.com/developerworks/library/l-completely-fair-scheduler/

Windows

"Each thread has a *dynamic priority*. This is the priority the scheduler uses to determine which thread to execute. Initially, a thread's dynamic priority is the same as its base priority. The system can boost and lower the dynamic priority, to ensure that it is responsive and that no threads are starved for processor time."

- Priority is boosted when:
  - Process's window is brought to foreground.
  - Process's window receives input.
  - Process was waiting for I/O, which has now completed.

Source: https://docs.microsoft.com/en-us/windows/desktop/ProcThread/priority-boosts

Scheduling Policy

- Large variation between OSes and their goals
  - Need to know ***your*** system, its workload, and its goals
- Lots of different scheduling policies
  - Designed with the goals in mind
  - Still an active area of development
- NEVER make assumptions about what the scheduler will do!

PROCESS COOPERATION

Concurrency

- Single CPU: *logical* concurrency
- Multiple CPU cores (commonly 4-16)
  - Still WAY more processes than CPUs
- With multiple cores (more hardware), performance is clearly important.
- There are other benefits too though!

Process Cooperation

- Independent process cannot affect or be affected by the execution of another process
- Cooperating process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Non-Performance Benefits

- Modularity: divide a task among specialized processes
  - develop / debug / test independently
  - reusable processes for other tasks
- Fault tolerance: if one process fails, user can interact with another
  - typically more distributed systems than OS
- I/O and blocking: if one process performs I/O and blocks, other(s) can keep executing.

Common Model: Producer-Consumer

- ***Producer*** process produces information that is consumed by a ***Consumer*** process
- Enabling model:
  - Common design patterns for coordinating processes (or threads)

Interprocess Communication: Pipeline

Example: `$ ls | sort | grep py`

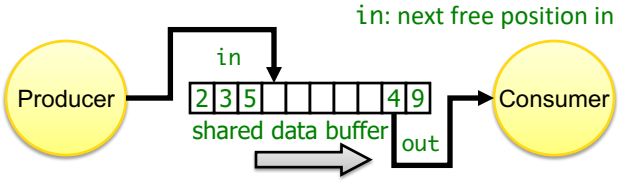
Unix philosophy: Do one thing, and do it well. (Modularity)
Result: lots of small utilities chained together at the command line.

Interprocess Communication: Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the user processes not the operating system.
- Major issue is to provide mechanism that will allow the user processes to ***synchronize*** their actions when they access shared memory.

Interprocess Communication: Shared Memory

One implementation: Circular buffer
`out`: first full position in the buffer
`in`: next free position in the buffer



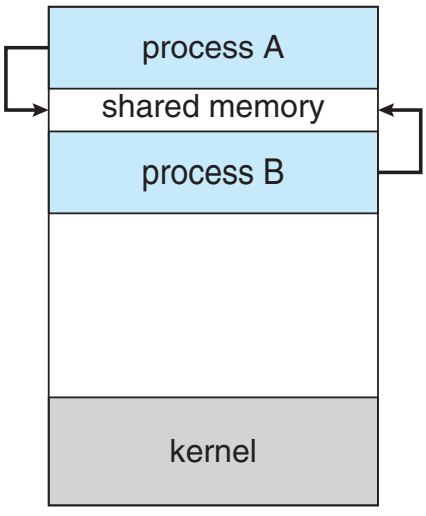
Example: media player

- Producer: read file from disk into buffer
- Consumer: decode file and send to output device

Each side can perform I/O and block independently of the other.

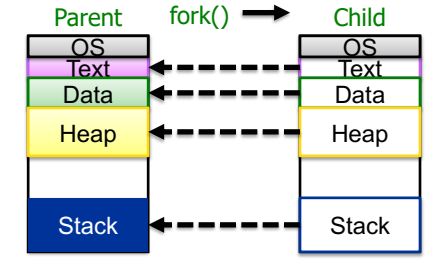
IPC via Shared Memory

- Process B uses system calls to create and attach to a shared memory segment.
- Process A uses a system call to map (attach) B's shared memory segment to its own address space.
- Shared memory is then accessed like any other portion of the process' address space.



Implicit Shared Memory: fork()

- When new process is created, it shares a (read only) copy of its parent's memory.
- Copy-on-write (COW) – only make a private copy of memory when process attempts to write. (Why?)
- Only works on shared hardware.
- Used frequently (every process creation!), e.g., shell commands.



Usually isn't used for long-term communication, but it does cause shared memory.

Looking Ahead

- Project 2: System Calls
- Threads!