Scheduling Policies

Case Study: CPU Scheduling

Jobs are created and removed dynamically. Need a fair and efficient way of scheduling the jobs. Some objectives:

- Throughput
- Response time (or waiting time)
- Fairness

- FIFO
  - But process lengths (time on CPU) can be long. Cannot afford such high response times in interactive systems.
- Shortest Time to completion first : provably efficient in terms of waiting time. But can completely starve long processes!
- Round Robin (fair)
  - But response times can be high. Convoy effect: I/O bound application keeps having to queue behing several compute bound applications
- Proportional CPU capacity reservation (e.g., provide proportionally more CPU for some jobs than others)
  Some jobs need higher CPU proportion than others: e.g., system processes (e.g., disk driver) over user processes
- Stride Scheduling
  - If P1:P2's proportions are X:1, schedule P1 X times more often than P1, i.e., schedule P1 for X quanta before scheduling P2.
- Lottery Scheduling
  - Probabilistic method of proportional CPU capacity reservation. Each process assigned some number of lottery tickets, and the scheduler draws a random ticket to select the next process.
  - Distribution of tickets based on proportional share. e.g., give X tickets to process P1 for one ticket to process P2.
  - Solves the problem of starvation: giving at least one lottery ticket to each process ensures that there is a non-zero probability of that process being scheduled at every scheduling quantum.
- Priority Scheduling Some jobs need higher *priority* than others: e.g., interrupts over others, I/O bound jobs over compute-bound jobs. Priorities are different from proportional scheduling: priorities are strict, i.e., a higher priority process always wins over a lower priority process.
  - Priority Inversion
  - Priority Donation
- Multi-level feedback queue
  - Maintain multiple priority levels: highest to lowest
  - At each priority level, serve processes in round-robin (or proportional CPU) way
  - Lower priority threads are scheduled only if no higher priority threads are ready to run.
  - Higher priority threads given smaller time quanta to run than lower priority threads (provides fairness and better throughput)
  - Typical usage: I/O bound jobs are put at higher priority, compute bound jobs are put at lower priority
  - How to identify I/O bound jobs: Use past = future.
    - On the first scheduling instance of a process, give it a high priority and a short time slice
    - If the process uses up the time slice without blocking (implies compute bound):
      - priority = priority - 1
      - time_slice = time_slice * 2
    - Draw diagram
  - But: won't low priority threads starve?
    - Solution: increase priority of threads at constant rate, while waiting. So the priority is increased for all threads which are not chosen for a scheduling quantum
    - Also if a thread blocks before exhausting its time quantum, have a way to increase its priority (e.g., CPU bound job converts to I/O bound)
- Affinity Scheduling : *much* better cache utilization
- Gang Scheduling : Can significantly increase throughput

Scheduling is Complex as multiple schedulers need to interact with each other

- Example: thread blocks (or is preempted) after holding spinlock. All other threads simply waste CPU cycles
- Threads have producer consumer relationship, that is completely hidden from the scheduler
- Multiple resources need to be scheduled in tandem for best throughput:
  - No use if process has priority on CPU but is given very little memory, so it page faults every time it gets to run, needs to wait for other (lower priority) processes' pages to be written to disk
  - Cache scheduling
  - Disk scheduling : Should the disk driver prioritize requests based on process priority
  - Server processes :
    - Suppose an interactive process uses the X server for display: does the X server know that the interactive process needs to be given priority over the compute-bound process
    - Similarly for NFS server

Scheduling was very important in the days of time sharing, when there was a shortage of resources all around. Many scheduling problems become not very interesting when you can just buy a faster CPU or a faster network. The topic becomes important again for modern datacenters or cloud computing environments desirous of extracting maximum utilization from their hardware resources.