Scheduling

1. Limited resources
2. Too many users/proceses
3. Long term scheduling
   1. Which processes are admitted into system, which is how many can enter and how many want to
4. Medium term
   1. Which processes reside in RAM and which are in storage, which means they will run immediately
5. Short term
   1. Which processes have a CPU?
6. There are different algorithm that runs at different level, for now assume single level
7. Scheduling Metrics

Context switching time -from one process to another. Expected to be small

* 1. Want to be easily understood

process

exit

first

exec

Latency/ turnaround time

Response time

Wait time

Model As

Model Ts

finish

1st output

Start building

arrival

fork

Time

* Want these times to be close to zero
* To maximize response and turnaround time, aggregate statistics by utilization: % of CPU time spent, and throughput: rate of useful work being done: sum of runtime / total time
* To minimize response and turnaround time, these goals can conflict

Simple Scheduler

* First come, first serve (FCFS) – assume 1 CPU

|  |  |  |  |
| --- | --- | --- | --- |
| Job | Arrival | Run | Wait-time |
| A | 0 | 5 | 0 + ξ |
| B | 1 | 2 | 4 + ξ |
| C | 2 | 9 | 5 + ξ |
| D | 3 | 4 | 13 + ξ |

AAAAA|BB|CCCCCCCCC|DDDD

Our integration is close to 1

Average wait time is 5.5 + 2.5 ξ

Short Job First (SJF): assumption knows job

|AAAAA|BB|DDD|CCCCCCCCC

Let’s add preemption

SJF+ preemption

|A|BB|DDDD|AAAA|CCCCCCCCC

* Average wait (ξ + 2 ξ+ (9+5 ξ) + 3 ξ) / 4 = 2.25 + 2.5 ξ
* Wait time is until when starts running, not until finish running
* Average run time = 2.25 + 2.5 ξ + 5 + (6 + 3 ξ) / 4
* When wait-time increases, average run time decreases

FCFS + preemption = round robin (RR)

* |A|B|C|D|A|B|C|D|A|C|D|A|C|D|A|CCCCC
* Average wait = ξ + 2 ξ + 3 ξ + 4 ξ = 2.5 ξ
* Average turn around = ( 15 + 15 ξ) + ( 5 + 5 ξ) + ( 18 + 14 ξ) + (11 + 11 ξ) = 49/4 + 45 ξ/4 = 12.25 + 11.25 ξ

Priorities – set by user or by system (real time applications)

* FCFS priority = arrival time
* SJF priority = run time

Real time scheduling

* Constraint must be met, or else (…disaster happens)
* hard real time

- Ex: brakes in car 🡪they have to engage as soon as it’s pressed, or car will crash

* Soft real time
  + If deadline missed, you drop that task
  + Several different schedulers
    - Earliest noticeable deadline first (most recent deadline first, approach not fair)
    - Rate – monotonic scheduling

Real time apps

* Mars Rover
* 3 priorities: low, medium, high

|  |  |  |
| --- | --- | --- |
| Low priority | Medium priority | High priority |
| Lock ( &m)  Run  Unlock(&m) | Started up | Lock (&m) // waits on the low priority task, until then spinning |

* Priority conversion: the example stated above is a problem because when medium is running, low can’t run, while high is waiting on low.
* Solution: if we have high waiting on low, then we high temporarily gives its high priority to low.

More Scheduling

* 1 CPU, 1 disk (too many I/O request: which to do first, and expensive to see and rotational latency) – Simple abstraction
* Simplest disk scheduler FCFS
  + Negative : lots of seeking – average seek time (assume random I/O) = ƒ(h\*h/2 + (1-h) (1-h)/2) dh
    - Assume : ¼ < answer < ½
  + Positive : no starvation – no processes coming in the meantime
* SSTF
  + Positive: minimizes seek time, maximizes throughput
  + Negative: more CPU time, starvation

Elevator scheduling – SSTF in a positive direction

Cyclic elevator scheduling – moves one direction only circular.

Anticipatory scheduling

* Thread A: 0,1,2,3
* Thread B: 1000000, 1000001, 1000002
  + Terrible performance moving back and forth
* If a process is processing and gets an I/O request, we anticipate what thread A is going to request and then complete thread B
  + It will complete all A, then B
  + Called DALLYING, minimizes seeks by guessing future requests
  + Downside: if guess wrong, performance time is bad

Clouds computing intro

Basic idea: shipping container, 20 feet long, jam CPU and disks as much as you can and do all computing on it and ship it

* Advantages
  + 1 organization running it, and one using it. That way you can get a large shared resource and economize at scale
  + Customer point of view – short term commitment: can grown computing needs as needed, “instantly”
  + Works well with computing applications where demand is unknown and varying
* Disadvantages
  + Question of payment because it’s done at arms length, so you have to make sure your OS must meter activities.
    - You or your customer down any each other to fiddle with the meter
  + Resource management – can you satisfy all incoming demands
  + Security: Many organizations use your clouds, and competitors using it want their nemesis’ information to be leaked.
  + data confidentiality
    - standard techniques – encrypt connectors to cloud and all flies in the cloud
      * trading CPU time for confidentiality
      * trust the guy running the cloud
  + software licensing
  + data transfer bottlenecks – CPU in constantly writing on data on nodes they were compile and transfer to the