Kai Lin Huang
cs162-ad
Notes for 2/2/2009 Lecture

Question: How overhead is handled in HW?
Answer: TA will find FAQ from previous semesters. Simulation is sarcastic, if simulation is similar, won't affect result. Whole idea: get a feel of what it feels

Run time is always highly skewed, like time for I/O

Goal: limit the time job will run for.
Don't want a job to wait too long, and it also mean other jobs will wait longer also.

Round Robin:

- Run process for one time slice (or max of 1 time slice), then move to back of queue. Each process gets equal share of the CPU.
-any reasonable algorithm has weakness
-if max time is infinite=First Come First Serve
Waiting time too long: one process can monopolize the CPU
Waiting time too small: too much time wasted in context swaps.

Advantage:
-if get the quality the right size, won't get job stuck. Less time in queue.
-low variance of f(i)/s(i)time. no starvation (no need to wait for long job that often<-not sure)

UNIX: 1sec time slices.

Important issue is what a good algorithm is a function of job indicator.
Some algorithm from last lecture doesn't work well or work well only for time is skewed, but it works bad result if jobs have equal time

Goal: optimize or minimize average flow time by prioritizing.

Little's Formula:

Formula to calculate mean number of users in system
(Same idea as average number of customers).
$N = \lambda * F$
N=mean number of users in system
F: mean flow (or waiting) time
λ: arrival rate of users (ie. $1/\lambda$=mean time between arrivals)

Shortest Job First (Shortest Processing Time)

If OS is not preemptive, we can get the shortest run time's job done first.
What it means is, when CPU becomes idle, pick a job requires least processing time to be efficient.

Advantage: optimal if no preemption. no knowledge of future arrivals, assuming we don't have that info.

Disadvantage:

- Still need certain amount of the future because it still needs to know how long the job will take.
- Can do better if use preemption with SJF so it can switch to the process with the current shortest remaining time. New arrivals wait even if they are short.
- high variance of flow time - long jobs wait

Shortest Remaining Processing Time (SRPT) with preemption.

When a job is the shortest when it arrives, run that job first if the current running one is longer, then return to the long one.

Advantage:

- Run the short one first, then go back to the longer one.
- This algorithm can be proved to minimize average flow time (response/waiting time).

Disadvantage:

- require future knowledge
- overhead//but not too much
- high variance of flow time

i.e. Walk to bank and long line, can get interrupt all the time

We can't predict future, so we use past performance to predict.

Q: how much preemption per job? Is there only 1 preemption per job.
A: Number of preemptions will not be more than the total number of jobs.

Shortest elapsed time (SET)

Run the job that has run the shortest amount of time so far.
Because run times are highly skewed, the job that has run shortest is most likely to finish first.

Advantage

- short job get out fast
- variance is low (flow time/service time)
  //it is supposed to be much better than FCFS.

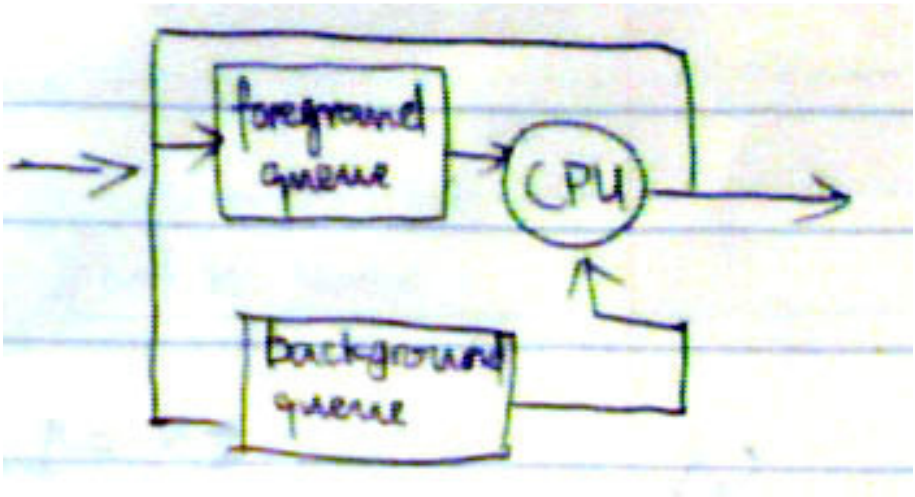
Disadvantage

- potential high overhead because constantly doing time switching
- if all job of same length, this is the worse algorithm because all job need to wait for all other jobs and will finish at the same time.
- high variance of flow time

The point: Algorithm works well if run time is skewed is good, otherwise will get the bad result; same as round robin.
The above algorithm made assumption that the run time is skewed.

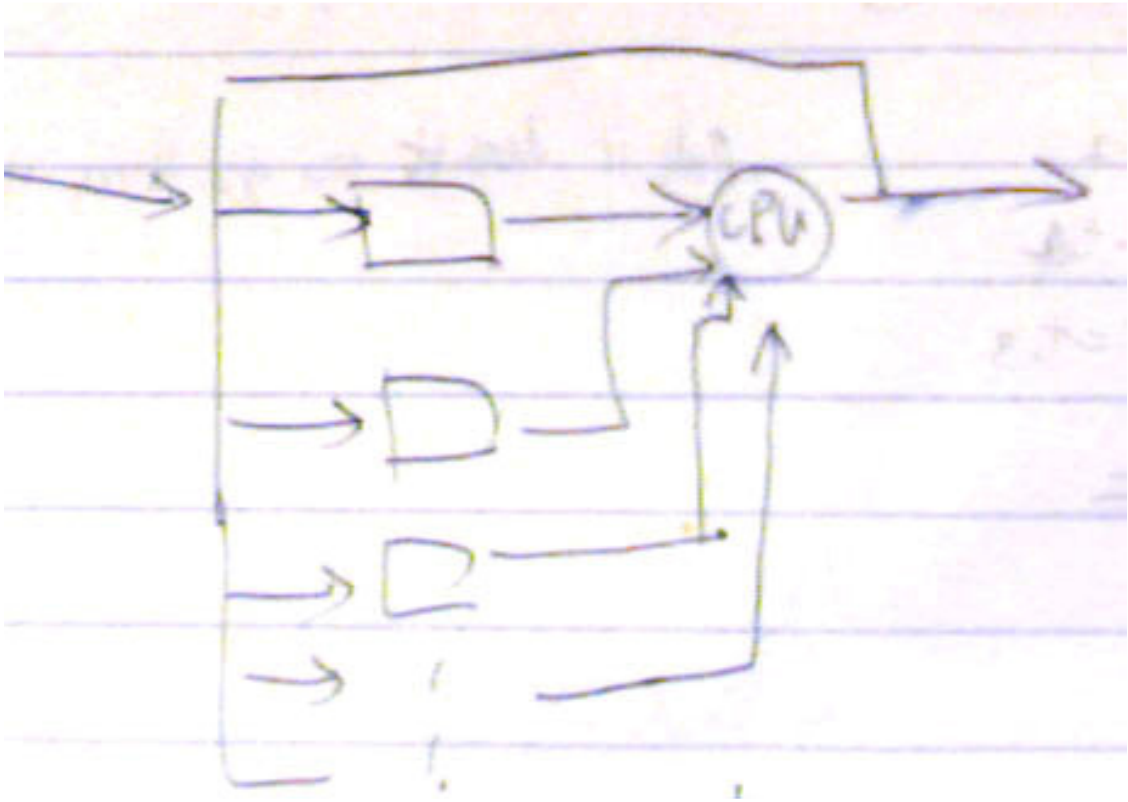
Foreground/Background scheduling:

2 queues: 1st priority over bkgd.



Foreground job always has priority over background jobs. You pick what job go to what ground.

- Any scheduling algorithm can be used for each of the queues, but there's a time slice. Jobs that have run for a while are placed back in either the foreground or background queue
- Assignment to foreground/background can be decided in some criterion .
- Allows good performance
- Foreground or background is good way to get CPU I/O overlap.



Graph shows many foreground and background queues.

Priority of the queues can be decided to pick what to be optimized.
No memory, only 1 parameter: run time.
In real sys, this job needs I/O, memory, etc. Real sys scheduler has a lot of things to consider (more dimensions).
Combine the consideration of overhead.
Idea: Job gets to run a quantum can't get to run because of scheduling overhead.

- jobs can be assigned to a level based on any of above criteria

- exponential queue : (a specific kind of foreground or background queue)
  - Combine consideration of overhead.
  - If get foreground get memory, if run in background probably no memory left (this queue give longer quantum to swap memory)
  - CTSS system was 1st to use exponential queue
    - can let queue more up toward foreground to gain priority

A huge amount of study in scheduling in 60, 70s because CPU expensive & machine are slow.

Adaptive Scheduling Algorithms

Ones which change the priority of the job depends on its run time behavior (FB, MLFB, SET, Exponential queue)

Fair share scheduling:

- keep history of recent CPU usage for each process
- give highest priority to process that has used the least CPU time recently.
- priority = wait time -10*cpu_use

Q: how is it differ from shortest elapsed time
A: Job that requires shorter time gets higher priority. Many algorithms are similar and different small tricks.

Example. if group get 500 to buy super computer, 200 hours in normal rate, 400 hr normal priority, ..not won't need to pay for running time.

BSD UNIX Scheduling
- Uses multilevel feedback queues (128 priority level, 32 queues)

- System runs jobs at front of highest priority occupied queue. In strict priority system, job is run for a quantum. Time quantum for 4.3 BSD is 0.1 sec.
- User priority = PUSER + PCPU = PNICE
  NICE: nice function: nice a job to very low priority (like. calculating prime)

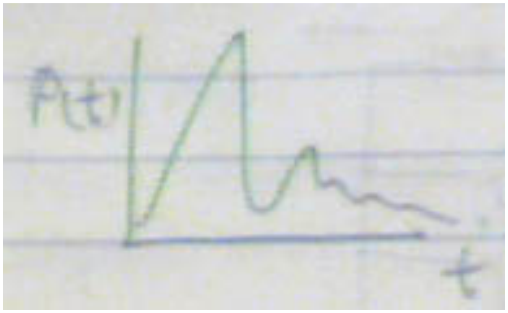
Q: Doesn't this introduce a lot of overhead?
A: No. the highest priority occupied queue.
Q: The speed of system won't be optimum
A: Simplicity isn't the most efficient often

How to fool machine to let it run faster?

- Make long job looks like short job.
- Do spurious I/O (to/dev/null), to get promoted to higher queue
- ->but please don't do it

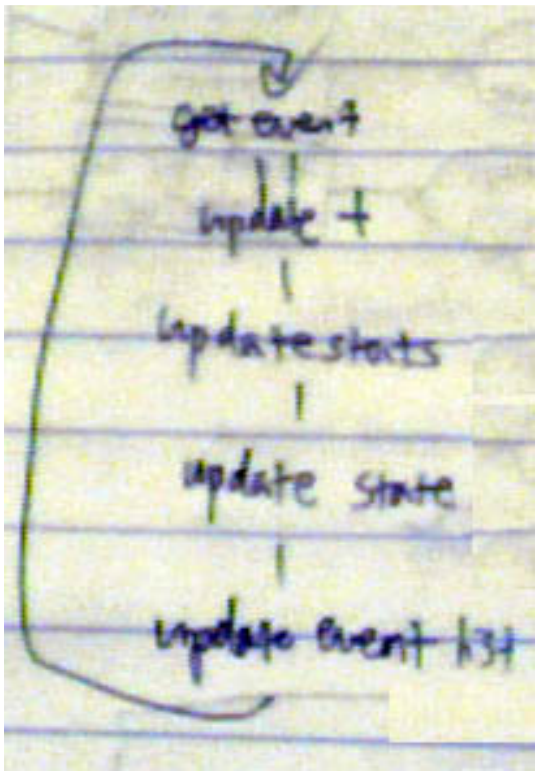
Summary:

- Scheduling algorithm can be arbitrary, b/c sys should produce same results in any event
- But algorithm have strong effects on the system's overhead, throughput, & response time
- Best scheme are adaptive and preemptive. To do best, need to be able to predict which job is long.
- Best scheduling algorithm tend to give highest priority to the process that need the least



Discrete event simulation

- event list
- LOOP



A simple system:
First arrival: arrival time 0, service time = t1, number of customer = 1
completion = t1;
Second arrival: next arrival come at t2; service time =t2, number of customer=2
Create an event for completion for 1.
and so on.

Independent and Cooperating processes

Independent process: one that can't affect of be affected by the rest of the universe
(Background information is in section 3.1, 3.3 of textbook)

- Its state isn't shared in any way by any other process
- Deterministic and reproducible: input state alone determines results
- can stop and restart w/ no bad effects(only time varies)
- Example. program that sums the integers from 1 to 100
- i.e. run the same thing will get the same output
- Example: add 1 to 100 always get same answer.

Usually some lack of complete independence. Only concentrate on "cooperative processes"

Cooperating processes

- Process that share some state/cooperate in some way. we are interesting in result computation.
- Technically speaking, timing won't affect process result.
- We want reproducible result.
- Example: run times and relative processes scheduling generally varies.

If start w/ quiescent machine, same thing happen or not?

- maybe, hard to get same condition -i.e. require angular position of hard disk same
- require all system data structure same
- disk layout same
- require system clock to set to same

Distinction between macro and micro behaviors

Micro: irreproducible.
Macro: reproducible.

Micro behavior: at the level of gates
Macro: result of computation. Same computation should produce same result each time.

Why let process to coop?
i.e.. go to airport and check-in station, everyone gets the same seat from different system,

Abbreviations:
eqn: equation
tbl: table
troff: trade off

Example: counting contest
Process A and Process B
While loop from i=0 to i=-10 and the other from 0 to 10
If start them at the same time,
One process will never complete. If process of same rate, never finish
One finishes. (One is quantum and the other still execute)
Both can finish at the same time if not the same i

Student Question: if different processes, thne i won't be same I, how can it happen?
Answer**:** assumption here is the same i. and it reference to the same place of memory.
Answer: in memory and not updated in register. Those are copies of i.