Lottery Scheduling

1. Student presentation:

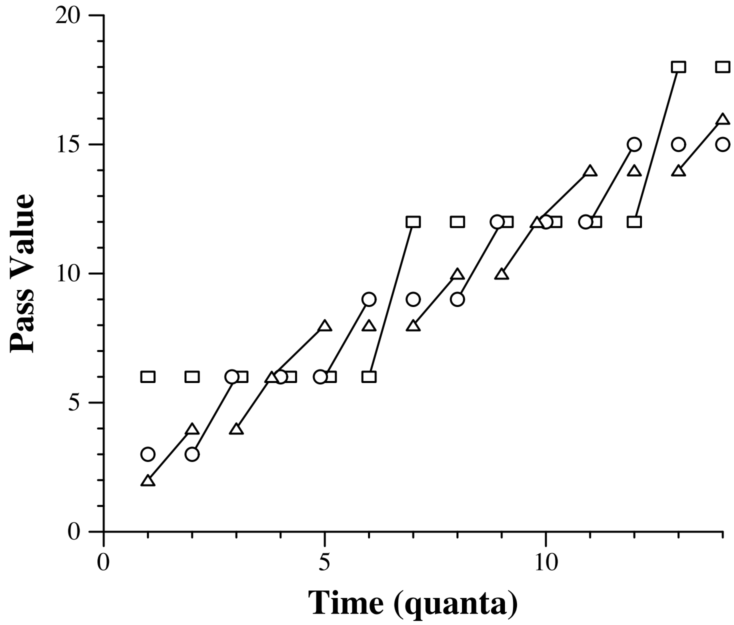
Bhat	Adithya
DHELIA	VISHAKHA
ROY CHOWDHURY	AMRITA
SINGARA VADIVELU	NIVETHA

    a.
2.
3. Questions
    a. How apply to disk schedulers?
    b. How important is throughput
    c. Relationship to fair-share scheduling?
    d. Scale to multicore & distributed systems?
    e. How extend to multiple resources?
        i. Can choose where to use tickets – stop using CPU, use for disk bandwidth. Can split tickets – some for CPU, some for disk
        ii. If have per-resource tickets, Can trade tickets with other applications – give tickets for disk in exchange for tickets for CPU
4. Review of normal schedulers – multi-level feedback queue (Unix, Windows)
    a. N priorities, each with a ready queue
    b. Execute at level N if level N+1 and higher empty
    c. Threads can be assigned an initial priority
    d. Threads can move between queues
        i. Priority lowered if exhausts quantum
        ii. Priority raised if sleeps early / when woken up from sleep
        iii. Quantum shorter for higher-priority queues
        iv. Quantum longer for lower-priority queues
        v. Priority raised if have been pre-empted before full quantum
    e. Issues:
        i. System knows nothing about users
        ii. Scheduling done based on processes
            1. Gives more share to users with more processes
        iii. Only mechanism for limiting consumption are quotas / charging

        iv. Starvation: if high priorities are busy, low priority starves
        v. Hard to transfer priority to thread being waited on; if two threads are waiting, does it get double priority?
        vi. Hard to control quality for multimedia – may need 20% of CPU to decode smoothly
5. Goal: proportional share scheduling
    a. Share according to users / higher-level groups, not processes
        i. N users want something
        ii. Each gets 1/N
        iii. Easy (relatively) for evenly weighted
        iv. What if want more flexible distribution of weights?
    b. Want to handle case where not all are active; if n/2 are active, each process gets 1/n/2
6. QUESTIONS:
    a. When do you want proportional share?
        i. Good for throughput-oriented systems
        ii. Good for equal-priority applications to ensure get equal access to CPU
        iii. Can guarantee quality-of-service
    b. What is needed?
        i. Someone has to assign shares
        ii. Default: everybody gets the same amount
    c. How many clients?
    d. What properties do you want?
        i. Timeliness
7. Lottery scheduling ideas
    a. **Biggest idea: Tickets/shares**
        i. Resource rights are abstract
            1. Independent of machine details
            2. Not tied to cpu cyles, memory pages
        ii. Uniform rights
            1. Can apply to heterogeneous pool of resources (but may need a conversion factor)
        iii. **Can be allocated/ transferred like memory**
    b. **Second idea: proportional share is a useful idea**
        i. Gives access to resources independent of how program works
            1. E.g. 1 thread or 1000 threads if share tickets
        ii. Compare to normal scheduling:
            1. Low predictability of how much time a process gets
                a. Based on interactivity/batch, priority

    b. Can reason about relative priority (who runs next) but not total run time s

c. **Third idea: economic models for resource allocation**
    i. Example: inflation, deflation, currencies
    ii. Auctions – bid how much resources needed (how valuable a resource is)
        1. Give to the program that benefits the most (and has enough money to spend)
d. **Fourth idea: randomness/Lotteries for making choice**
    i. Each client gets some number of tickets
    ii. Chance of winning = # of tickets / # of tickets contending
    iii. **Why good?**
        1. Fast – doesn't need much state (e.g. tracking execution time)
        2. Hard to game – randomness makes it hard to predict what will happen
e. Randomness for making decision
    i. Randomly pick a process at each time
    ii. Converges with sqrt(# lotteries)
    iii. Expected time to win is 1/p (p = proportion)
    iv.
f. **NOTE: most of system works just fine if lotteries are not random, but deterministically pick a schedule to run threads that follows the allocation**
g. Implementation
    i. Hold lotteries in base units (== sum of base tickets for ready processes)
    ii. Scan ready list accumulating partial sum until hit process
    iii. Move large ticket holders to front to minimize average scan length
    iv. Optimizations: tree with partial sums
h. NOTE: lottery implementation is not used; randomness hard to reason about. instead, strides:
    i. let thread run, compute next run time as 1/fraction tickets = stride. Always run earliest thread
    ii. Example:



i.

j. When a client consumes fraction F of its allocated time quantum, its pass should be advanced by F x stride instead of stride.
    i. When rescheduled, pass value will be lower, will be scheduled early
        1. Oldest waking thread runs first
k. QUESTION: Tickets don't get consumed. Why?

8. General ideas:
    a. Randomness
    b. Lotteries
    c. Currencies – conversion between resources (e.g. i/o bandwidth, memory, cpu) or users
9. Extensions
    a. Ticket inflation: mint more tickets in a currency
        i. QUESTION: Who should be able to do this?
            1. If have N processes, should all of them?
    b. Ticket transfers
        i. Move tickets from one client to another
        ii. E.g. rpc client gives to rpc server
        iii. Lock waiter give to lock holder
    c. Currency
        i. QUESTION: What problem does it solve?
        ii. System provides base tickets
        iii. Clients can issue tickets denominated in their own currency
        iv. Allows dividing resources.
        v. Easy to have all children have equal shares

        1. QUESTION: How?
        2. Just give each one same # of tickets
        3. No need to adjust tickets for other clients
        4. (INFLATION)
   d. Compensation tickets
      i. QUESTION: What problem do they solve?
      ii. If use only fraction F of allocated resource, tickets inflated by 1/F until next starts to use resource
      iii. Makes client more likely to win lottery
         1. If run for N times shorter, should win N times more often to achieve same utilization!
      iv. Keep proportional share property
      v. Makes system more responsive for interactive processes, because expected waiting time is lower
         1.

10. Uses
   a. Variable scheduling for simulation
      i. Prioritize computations with large error over those refining errors
   b. Donate tickets from client to server
      i. Encourages server to run faster and complete more quickly and be scheduled sooner
   c. Multimedia
      i. Degrading service when handling multiple clients; don't want to freeze some out
      ii. Use proportional share based on weights

11. Space-shared resources

VMWARE POLICY: proportional share (we'll see this later)

Key idea:

- some pool of resources R
- Want to allocate fractions of it to different users
- would like a minimum guarantee, but efficient use of excess capacity

Solution:

- give each user a set of shares, like stock shares in a company
- value of a share is #shares / total # shares—this is minimum guarantee
- At any time, amount of resource is # shares / total # shares **demanded**
- **Shares represent *relative resource rights that depend on the total number of shares contending for the resource***

Idea: under heavy use, get strict proportion. Under light use, can get more in proportion to others who want more and their shares/

Way to think about it: everybody who wants a resource buys lottery tickets with shares. Winner picked at random from all shares bid. If not need, don't buy tickets

So: under full demand by everyone, all pay same price per page: shares / pages granted. When not everybody has full demand, some with fewer shares will get more pages

RECLAMATION: when pages needed, search for VM that is paying the **least** for its memory (e.g. got some memory when others didn't want it.)

Algorithm: dynamic min-funding revocation.

Example

VM 1: 100 shares

VM 2: 100 shares

Total memory: 400 mb

VM 1 starts running, acquire 256 mb for 100 shares

price = 100/256 = 0.4

VM2 starts running, gets remainder: 144 MB for 100 shares

price = 100/144 = 0.69

When VM2 wants more memory, it comes from VM1

VM2 needs more pages, asks for 56

VM2 price = 100/200 =0.5

VM1 price = 100/200 = 0.5

Now VM1 has 200 MB, VM1 has 200 MB, both pay same price - in equilibrium

NOTE: reclamation is kind of expensive; need to activate balloon or swap pages.

QUESTION: is this the right policy? It doesn't guarantee timeliness, just a minimum.

NOTE: Real problem is not minimum guarantee, but how to efficiently use memory above that.

a.

12. Nice properties

a. Handles priority inversion

        i. Donate tickets to lock holder
        ii. Lock holder holds lottery when releasing to find next holder
            1. Gets tickets from all waiters
    b. Easy to donate resources – give them your tickets
        i. E.g. client/server model – client gives server tickets
        ii. All clients give server tickets, so runs longer to return more quickly
    c. When don't use full resource quanta, are given tickets inversely proportional to used fraction (e.g. if use 1/5, get 4x tickets for next lottery), assuming next usage will be similar

13. Issues
    a. Schedulers give higher priority to threads holding kernel resources (so they release them more quickly)
        i. Classic LS solution: contending users donate resources to holder
            1. Problem: Too expensive to hold lottery
            2. Problem: API for waiting not have enough information for lottery
        ii. Solution: Maintain priority queues for threads that woke up from being blocked on kernel resource; schedule these before holding lottery
        iii. Charge them tickets according to how long they ran from this method.
    b. Implementing NICE
        i. What does NICE do: ensure a process only runs if there are no higher-priority processes in the system
        ii. QUESTION: How do you do this with proportional share scheduling?
            1. Can't really; want a priority mechanism not a proportional share.
        iii. Problem: lowering user-denominated tickets doesn't help:
            1. QUESTION: Why?
                a. What if nice'd process is only one of a user – it will get entire user's share
        iv. Issue: need to adjust priority relative to other users, not just to one user
        v. Solution:
            1. At scheduling time, adjust base tickets to be at most or at least a value proportional to NICE priority
    c. Supporting interactive users: issue
        i. Force context switch when sleeping process wakes up

            1. Pre-empted process gets appropriate compensation tickets
        ii. Issue: pre-empted cpu bound process with compensation tickets competes with i/o bound process
            1. Solution: see who has received less CPU than their # of tickets should indicate
            2. These are interactive, because they often block waiting for input
            3. Give them a boost – e.g. multiplicative factor to tickets.
    d. CPU is not the only resource; unclear how well you can balance between resources (despite the goal)

14. My sense:
    a. Best used as a scheduler layer in a system with other schedulers as well.
    b. e.g. within a priority level
15. Challenges
    a. Responsiveness for interactive tasks
        i. no guarantee of low latency
        ii.