Advanced Operating Systems (CS 202)

Scheduling (2)

Today: CPU Scheduling

- What should the scheduler algorithm do?
  - Are the ad hoc schedulers ok?
  - What do commercial OS' do?
  - Lottery and Stride scheduling

- Scheduling activations
  - User level vs. Kernel level scheduling of threads
  - Can be thought of as extensibility for scheduling
  - May skip and give you a quick summary

- How do we schedule on emerging machines?
  - Multicores/many-cores? Decade of wasted cores
  - Cloud, embedded-- Hawk

LOTTERY SCHEDULING

Unix Scheduler

- The canonical Unix scheduler uses a MLFQ
  - 3-4 classes spanning ~170 priority levels
    - Timesharing: first 60 priorities
    - System: next 40 priorities
    - Real-time: next 60 priorities
    - Interrupt: next 10 (Solaris)
- Priority scheduling across queues, RR within a queue
  - The process with the highest priority always runs
  - Processes with the same priority are scheduled RR
- Processes dynamically change priority
  - Increases over time if process blocks before end of quantum
  - Decreases over time if process uses entire quantum

Problems with Traditional schedulers

- Priority systems are ad hoc: highest priority wins
- Try to support fair share by adjusting priorities with a feedback loop
  - Works over long term
  - highest priority still wins but now the priorities are changing
- Priority inversion: high-priority jobs can be blocked behind low-priority jobs
- Schedulers are complex and difficult to control

Lottery scheduling

- Elegant way to implement proportional share scheduling
- Priority determined by the number of tickets each thread has:
  - Priority is the relative percentage of all of the tickets whose owners compete for the resource
- Scheduler picks winning ticket randomly, gives owner the resource
- Tickets can be used for a variety of resources

Example

- Three threads
  - A has 5 tickets
  - B has 3 tickets
  - C has 2 tickets
- If all compete for the resource
  - B has 30% chance of being selected
- If only B and C compete
  - B has 60% chance of being selected

Its fair

- Lottery scheduling is ***probabilistically fair***
- If a thread has a t tickets out of T
  - Its probability of winning a lottery is $p = t/T$
  - Its expected number of wins over n drawings is np
    - Binomial distribution
    - Variance $\sigma^2 = np(1 - p)$

Fairness (II)

- Coefficient of variation of number of wins $\sigma/np = \sqrt{(1-p)/np}$
  - Decreases with $\sqrt{n}$
- Number of tries before winning the lottery follows a ***geometric distribution***
- As time passes, each thread ends receiving its share of the resource

Ticket transfers

- How to deal with dependencies?
  - Explicit transfers of tickets from one client to another
- Transfers can be used whenever a client blocks due to some dependency
  - When a client waits for a reply from a server, it can temporarily transfer its tickets to the server
    - Server has no tickets of its own
  - Server priority is sum of priorities of its active clients
    - Can use lottery scheduling to give service to the clients
- Similar to priority inheritance
  - Can solve priority inversion

Ticket inflation

- Lets users create new tickets
  - Like printing their own money
  - Counterpart is ***ticket deflation***
  - Lets mutually trusting clients adjust their priorities dynamically without explicit communication
- Currencies: set up an exchange rate
  - Enables inflation within a group
  - Simplifies mini-lotteries (e.g., for mutexes)

Example (I)

- A process manages three threads
  - A has 5 tickets
  - B has 3 tickets
  - C has 2 tickets
- It creates 10 extra tickets and assigns them to process C
  - Why?
  - Process now has 20 tickets

Example (II)

- These 20 tickets are in a new currency whose exchange rate with the base currency is 10/20
- The total value of the processes tickets expressed in the base currency is still equal to 10

Compensation tickets (I)

- I/O-bound threads are likely get less than their fair share of the CPU because they often block before their CPU quantum expires
- Compensation tickets address this imbalance

Compensation tickets (II)

- A client that consumes only a fraction *f* of its CPU quantum ***can*** be granted a ***compensation ticket***
  - Ticket inflates the value of all client tickets by 1/f until the client starts gets the CPU

Example

- CPU quantum is 100 ms
- Client A releases the CPU after 20ms
  - f = 0.2 or 1/5
- Value of ***all*** tickets owned by A will be multiplied by 5 until A gets the CPU
- Is this fair?
  - What if A alternates between 1/5 and full quantum?

Compensation tickets (III)

- Compensation tickets
  - Favor I/O-bound—and interactive—threads
  - Helps them getting their fair share of the CPU

IMPLEMENTATION

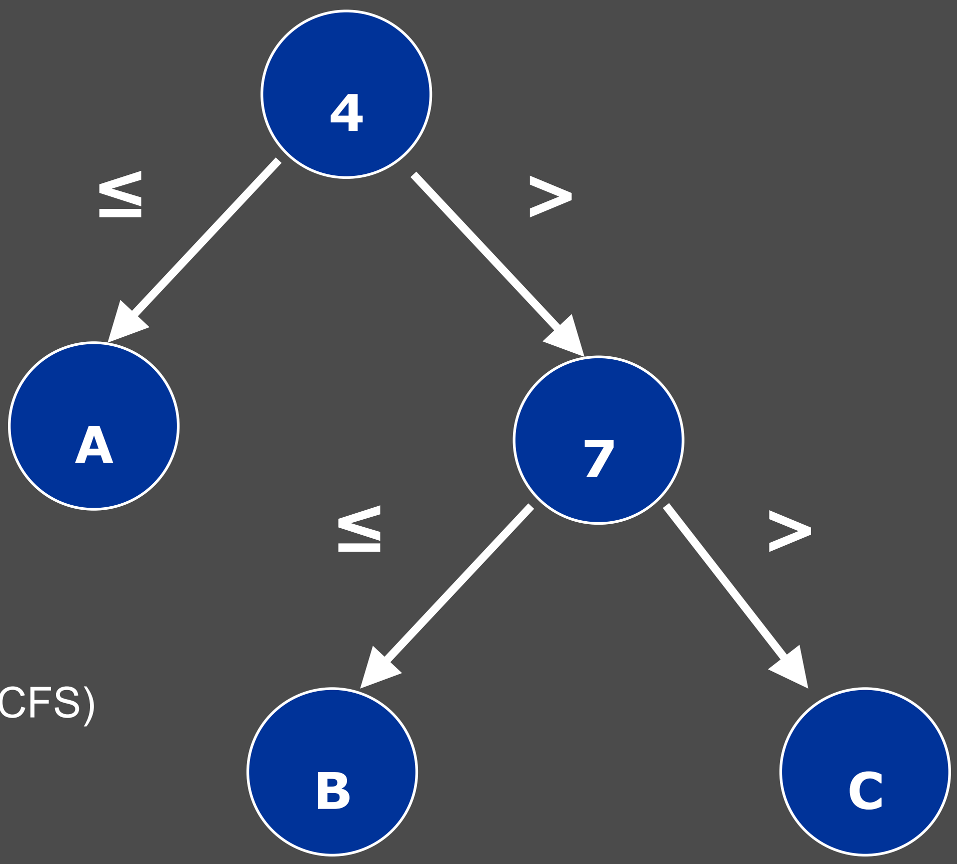
- On a MIPS-based DECstation running Mach 3 microkernel
  - Time slice is 100ms
    - *Fairly large as scheme does not allow preemption*
- Requires
  - A fast RNG
  - A fast way to pick lottery winner

Example

- Three threads
  - A has 5 tickets
  - B has 3 tickets
  - C has 2 tickets
- List contains
  - A (0-4)
  - B (5-7)
  - C (8-9)

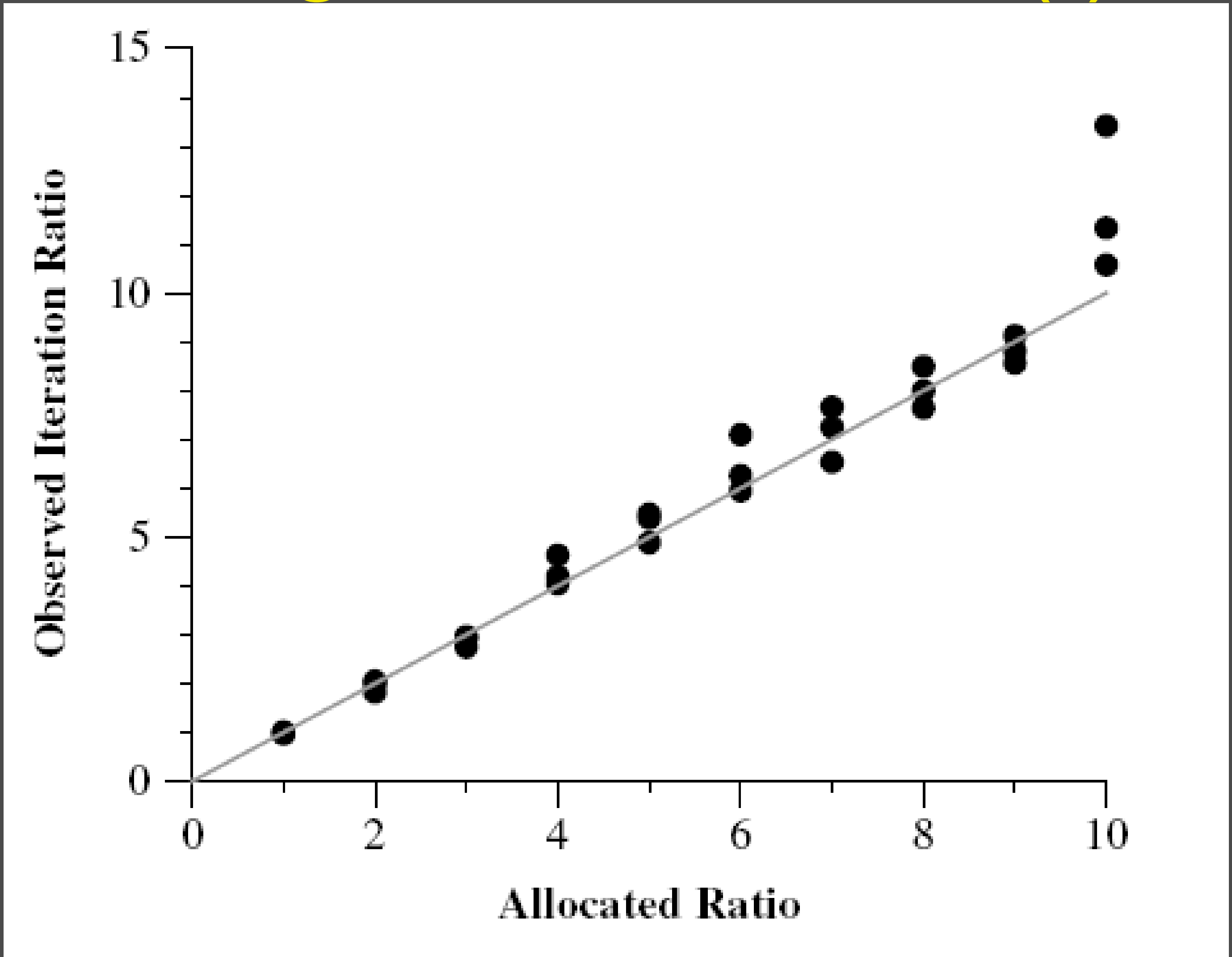
Search time is O(*n*)
where *n* is list length

Optimization – use tree

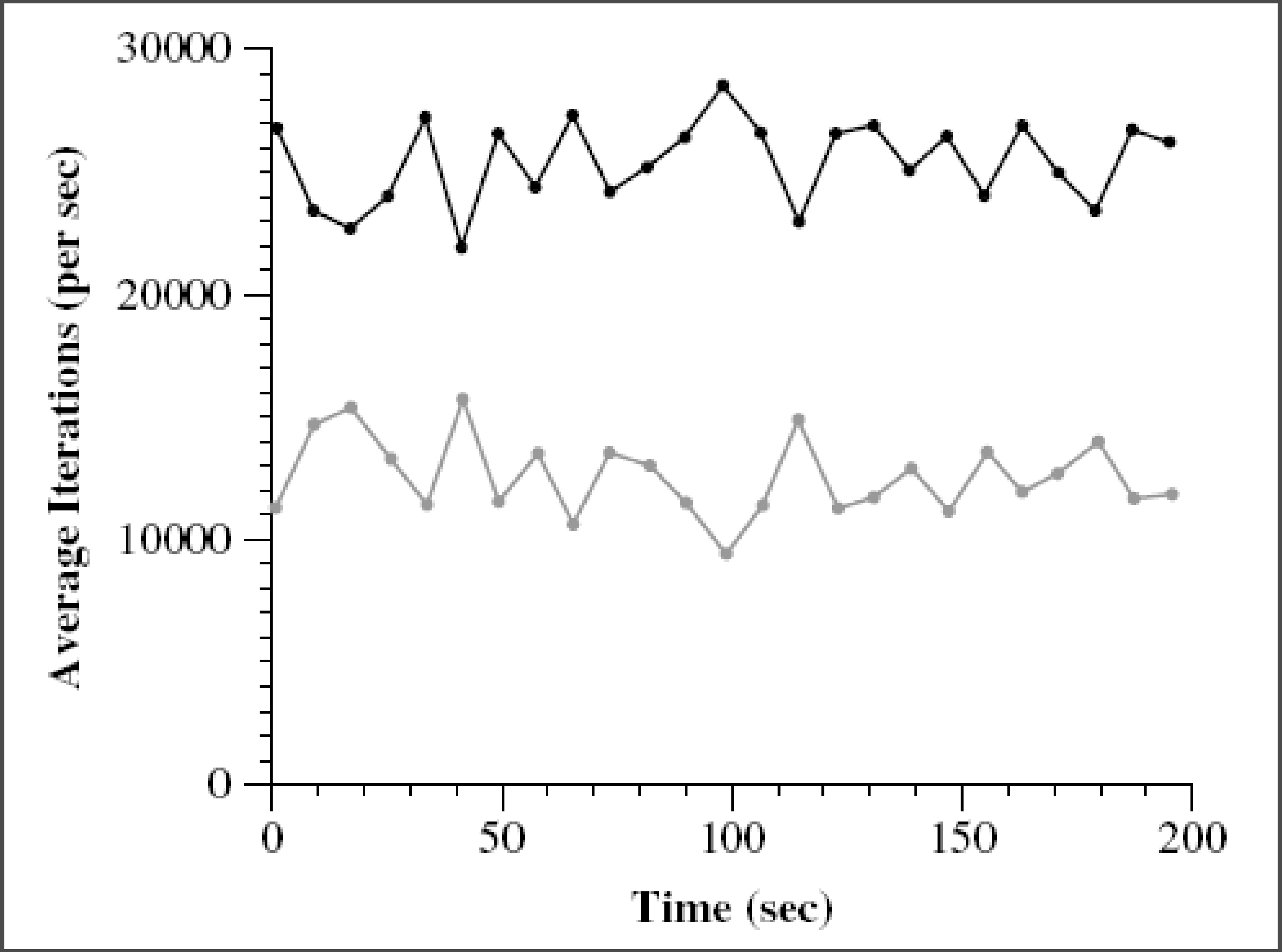


RB Tree used in Linux
Completely fair scheduler(CFS)
--not lottery based

Long-term fairness (I)



Short term fluctuations



For
2:1
ticket
alloc.
ratio

Discussion

- Opinions of the paper and contributions?
  - Fairness not great
    - Mutex 1.8:1 instead of 2:1
    - Multimedia apps 1.9:1.5:1 instead of 3:2:1
  - Can we exploit the algorithm?
    - Consider also indirectly – processes getting kernel cycles by using high priority kernel services
  - Real time? Multiprocessor?
  - Short term unfairness
    - Later this lead to stride scheduling from same authors

Stride scheduling

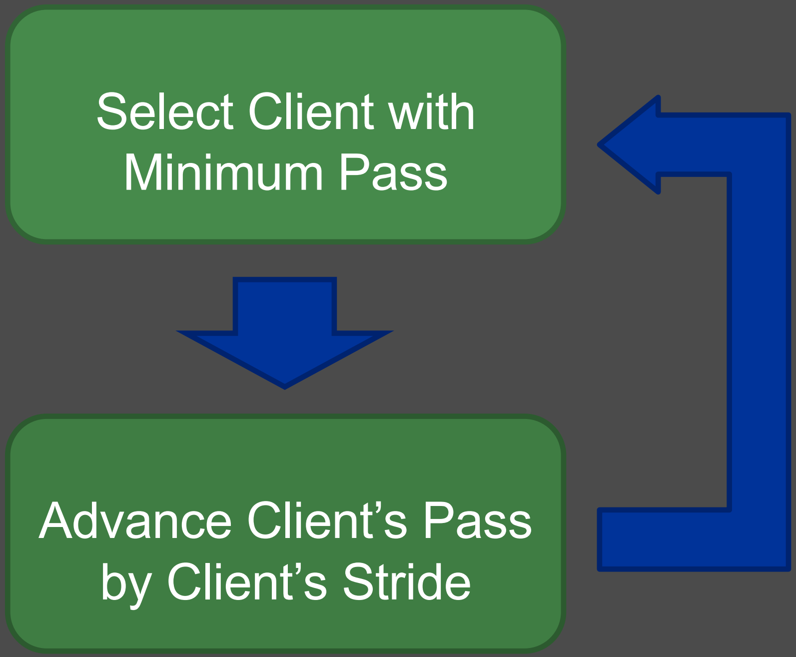
- Deterministic version of lottery scheduling
- Mark time virtually (counting passes)
  - Each process has a stride: number of passes between being scheduled
  - Stride inversely proportional to number of tickets
  - Regular, predictable schedule
- Can also use compensation tickets
- Similar to weighted fair queuing
  - Linux CFS is similar

Stride Scheduling – Basic Algorithm

Client Variables:

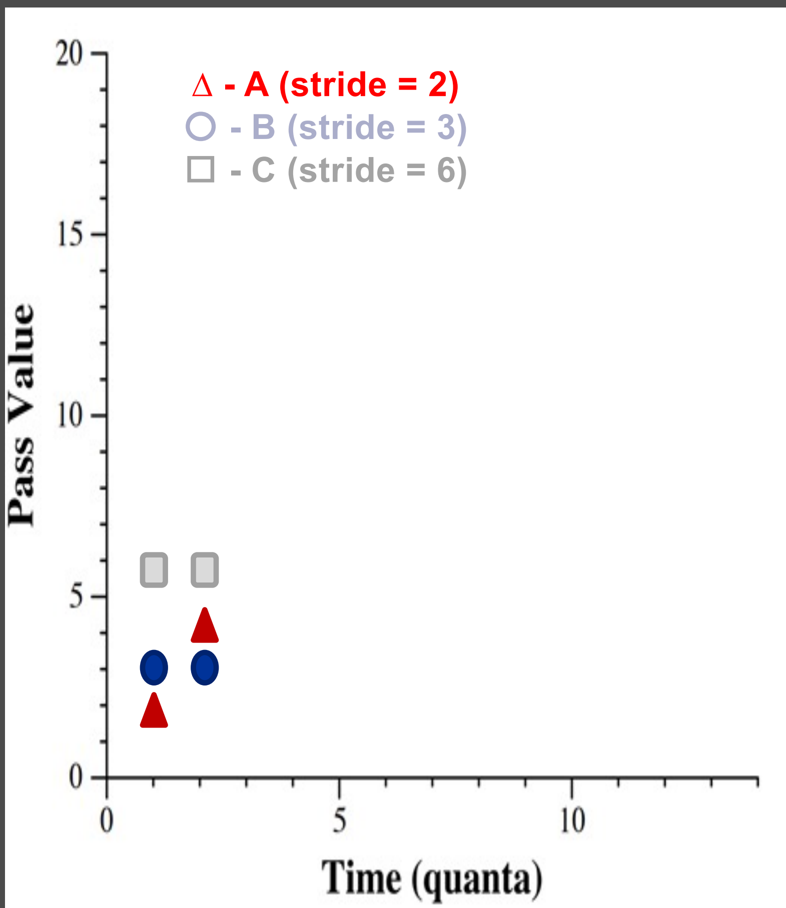
- Tickets
  - Relative resource allocation
- Strides (
  - Interval between selection
- Pass (
  - Virtual index of next selection

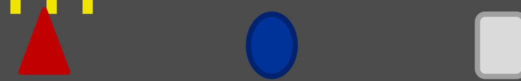
- minimum ticket allocation



Slide and example from Dong-hyeon Park

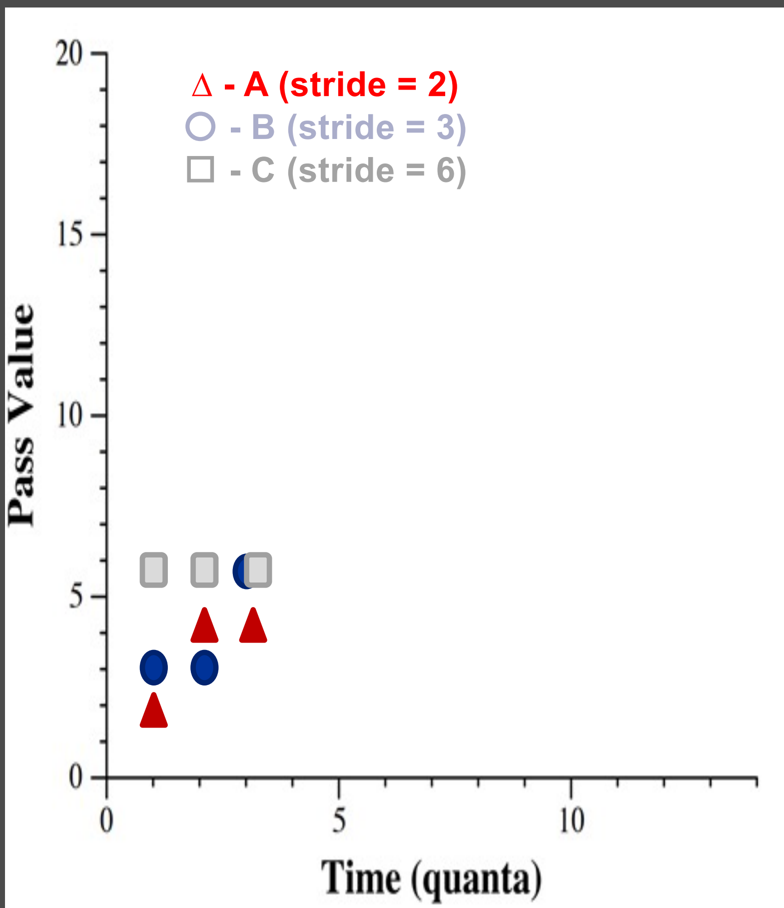
Stride Scheduling – Basic Algorithm





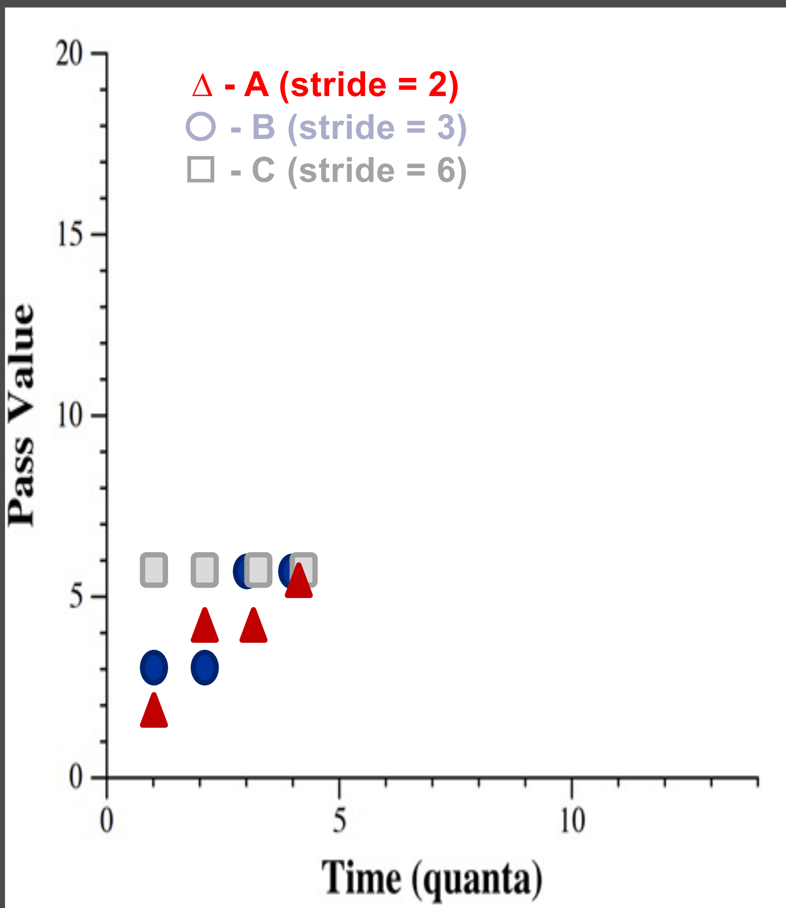
Time 1:	2	3	6
	+2		
Time 2:	4	3	6

Stride Scheduling – Basic Algorithm



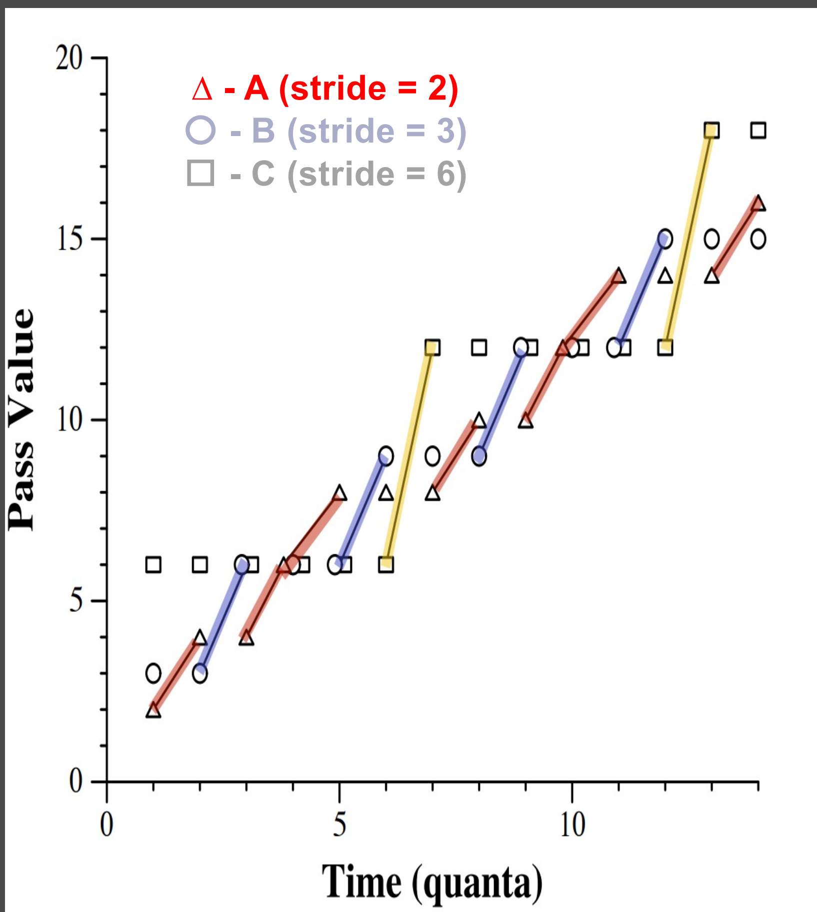
Time 1:	2	3	6
	+2		
Time 2:	4	3	6
		+3	
Time 3:	4	6	6

Stride Scheduling – Basic Algorithm



Time 1:	2	3	6
	+2		
Time 2:	4	3	6
		+3	
Time 3:	4	6	6
	+2		
Time 4:	6	6	6

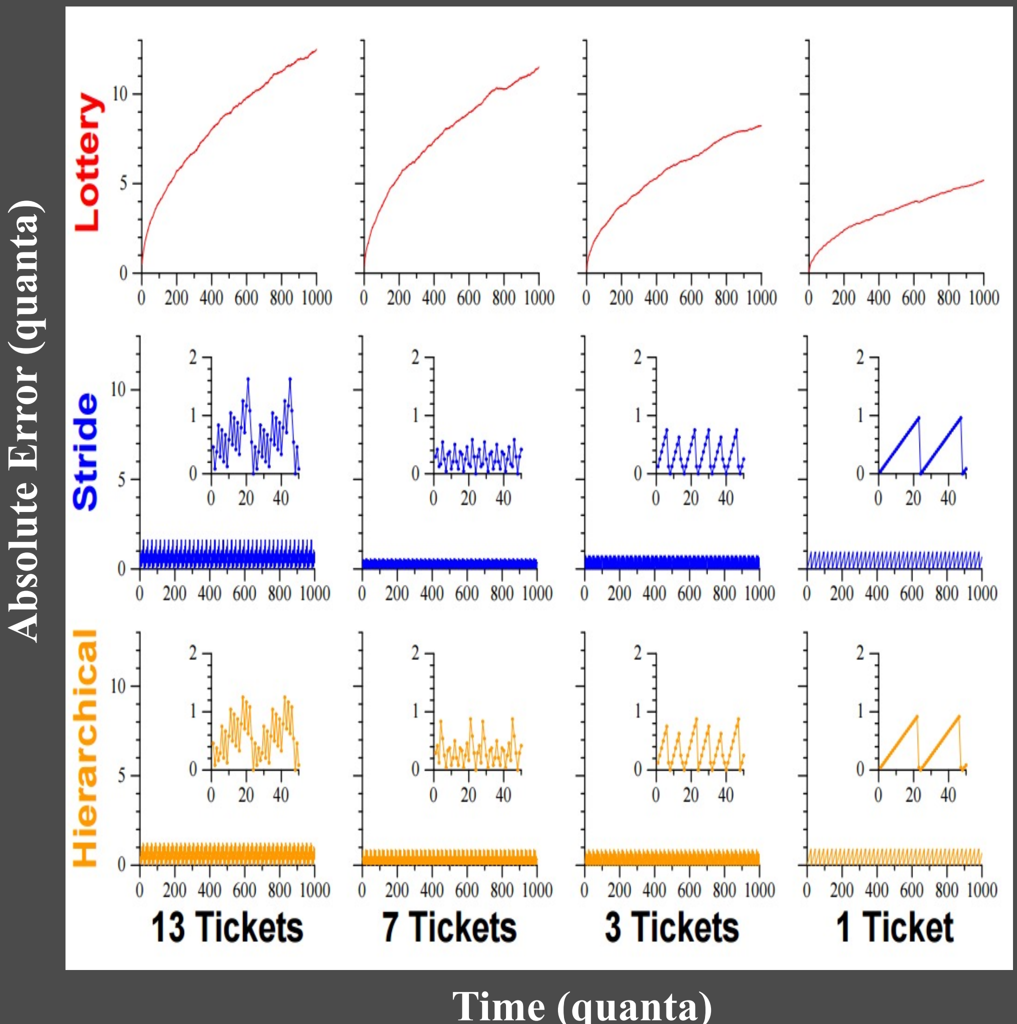
Stride Scheduling – Basic Algorithm



Time 1:	2	3	6
	+2		
Time 2:	4	3	6
		+3	
Time 3:	4	6	6
	+2		
Time 4:	6	6	6

⋮

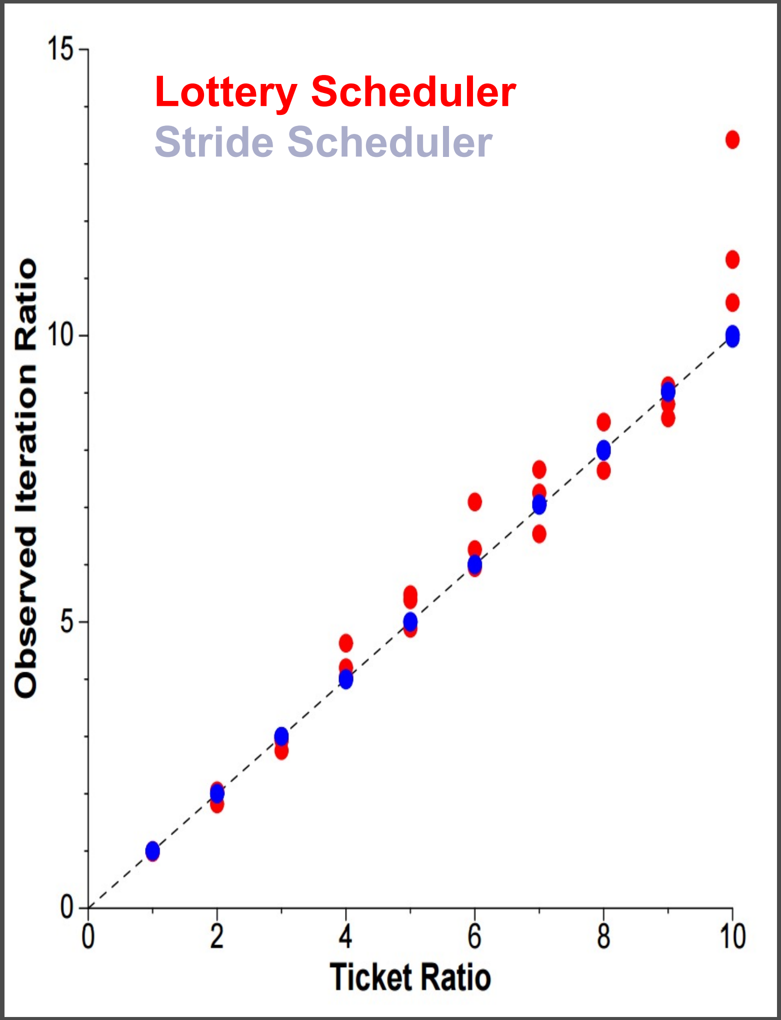
Throughput Error Comparison



Error is independent of the allocation time in stride scheduling

Hierarchical stride scheduling has more balance distribution of error between clients.

Accuracy of Prototype Implementation



- Lottery and Stride Scheduler implemented on real-system.
- Stride scheduler stayed within 1% of ideal ratio.
- Low system overhead relative to standard Linux scheduler.

Linux scheduler

- Went through several iterations
- Currently CFS
  - Fair scheduler, like stride scheduling
  - Supersedes O(1) scheduler: emphasis on constant time scheduling –why?
  - CFS is O(log(N)) because of red-black tree
  - Is it really fair?
- What to do with multi-core scheduling?

SCHEDULER ACTIVATIONS BREWER

Context

- Neither user level threads nor kernel level threads work ideally
  - User level threads have application information
    - They are also cheap
    - But not visible to kernel
  - Kernel level threads
    - Expensive
    - Lack application information

Idea

- Abstraction: threads in a shared address space
  - Others possible?
- Can be implemented in two ways
  - Kernel creates and dispatches threads
    - Expensive and inflexible
  - User level
    - One kernel thread for each virtual processor

User level on top of kernel threads

- Each application gets a set of virtual processors
  - Each corresponds to a kernel level thread
- User level threads implemented in user land
  - Any user thread can use any kernel thread (virtual processor)
    - Fast thread creation and switch – no system calls
    - Fast synchronization!
  - What happens when a thread blocks?
  - Any other issues?

Goals (from paper)

- Functionality
  - No processor idles when there are ready threads
  - No priority inversion (high priority thread waiting for low priority one) when its ready
  - When a thread blocks, the processor can be used by another thread
- Performance
  - Closer to user threads than kernel threads
- Flexibility
  - Allow application level customization or even a completely different concurrency model

Problems

- User thread does a blocking call?
  - Application loses a processor!
- Scheduling decisions at user and kernel not coordinated
  - Kernel may de-schedule a thread at a bad time (e.g., while holding a lock)
  - Application may need more or less computing
- Solution?
  - Allow coordination between user and kernel schedulers

Scheduler activations

- Allow user level threads to act like kernel level threads/virtual processors
- Notify user level scheduler of relevant kernel events
  - Like what?
- Provide space in kernel to save context of user thread when kernel stops it
  - E.g., for I/O or to run another application

Kernel upcalls

- New processor available
  - Reaction? Run time picks user thread to use it
- Activation blocked (e.g., for page fault)
  - Reaction? Runtime runs a different thread on the activation
- Activation unblocked
  - Activation now has two contexts
  - Running activation is preempted – why?
- Activation lost processor
  - Context remapped to another activation
- What do these accomplish?

Runtime->Kernel

- Informs kernel when it needs more resources, or when it is giving up some
- Could involve the kernel to preempt low priority threads
  - Only kernel can preempt
- Almost everything else is user level!
  - Performance of user-level, with the advantages of kernel threads!

Preemptions in critical sections

- Runtime checks during upcall whether preempted user thread was running in a critical section
  - Continues the user thread using a user level context switch in this case
    - Once lock is released, it switches back to original thread
    - Keep track of critical sections using a hash table of section begin/end addresses

Discussion

- Summary:
  - Get user level thread performance but with scheduling abilities of kernel level threads
  - Main idea: coordinating user level and kernel level scheduling through scheduler activations
- Limitations
  - Upcall performance (5x slowdown)
  - Performance analysis limited
- Connections to exo-kernel/spin/microkernels?