Advanced Operating Systems (CS 202)

Scheduling (1)

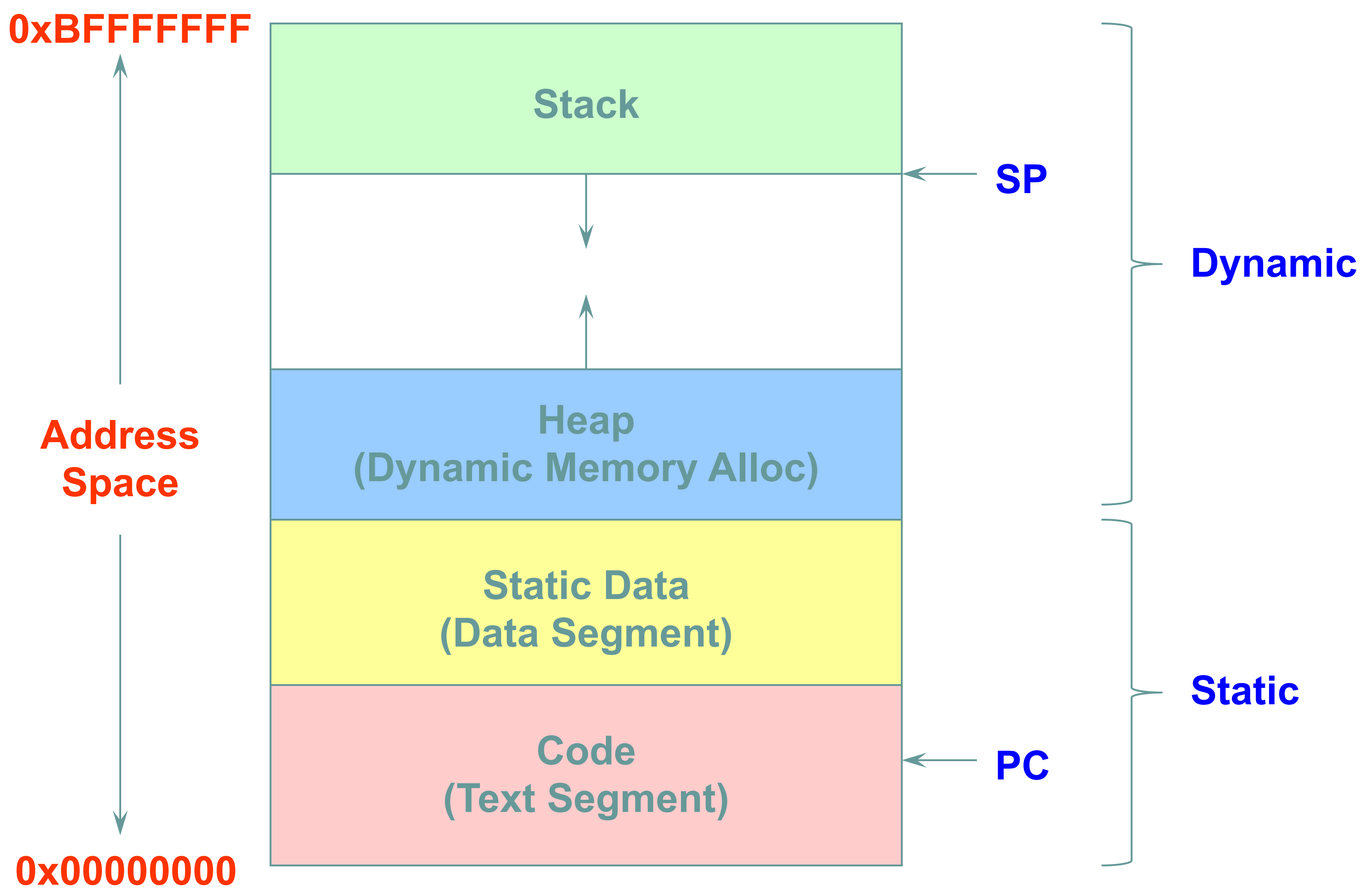
Today: CPU Scheduling

The Process

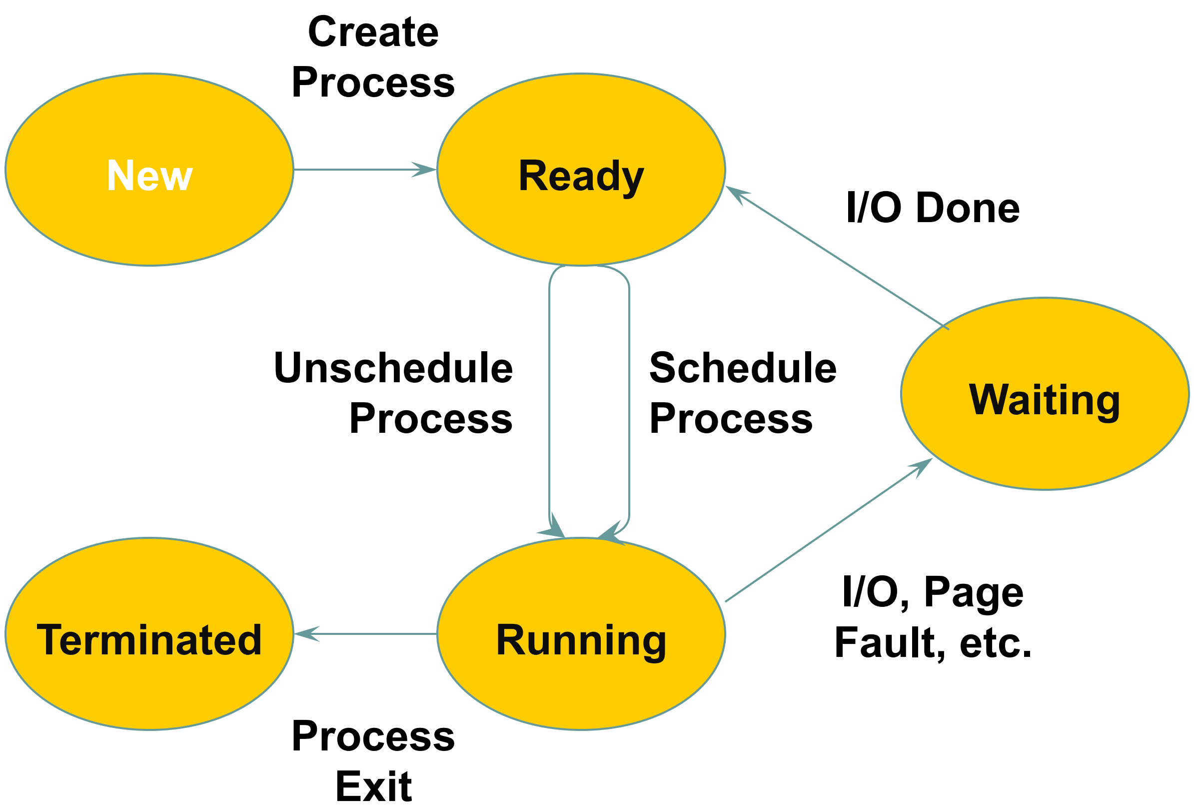
- The process is the OS abstraction for execution
  - It is the unit of execution
  - It is the unit of scheduling
  - It is the dynamic execution context of a program
  - A process is sometimes called a job or a task

- A process is a program in execution
  - Programs are static entities with the potential for execution
  - Process is the animated/active program
    - Starts from the program, but also includes dynamic state

Process Address Space



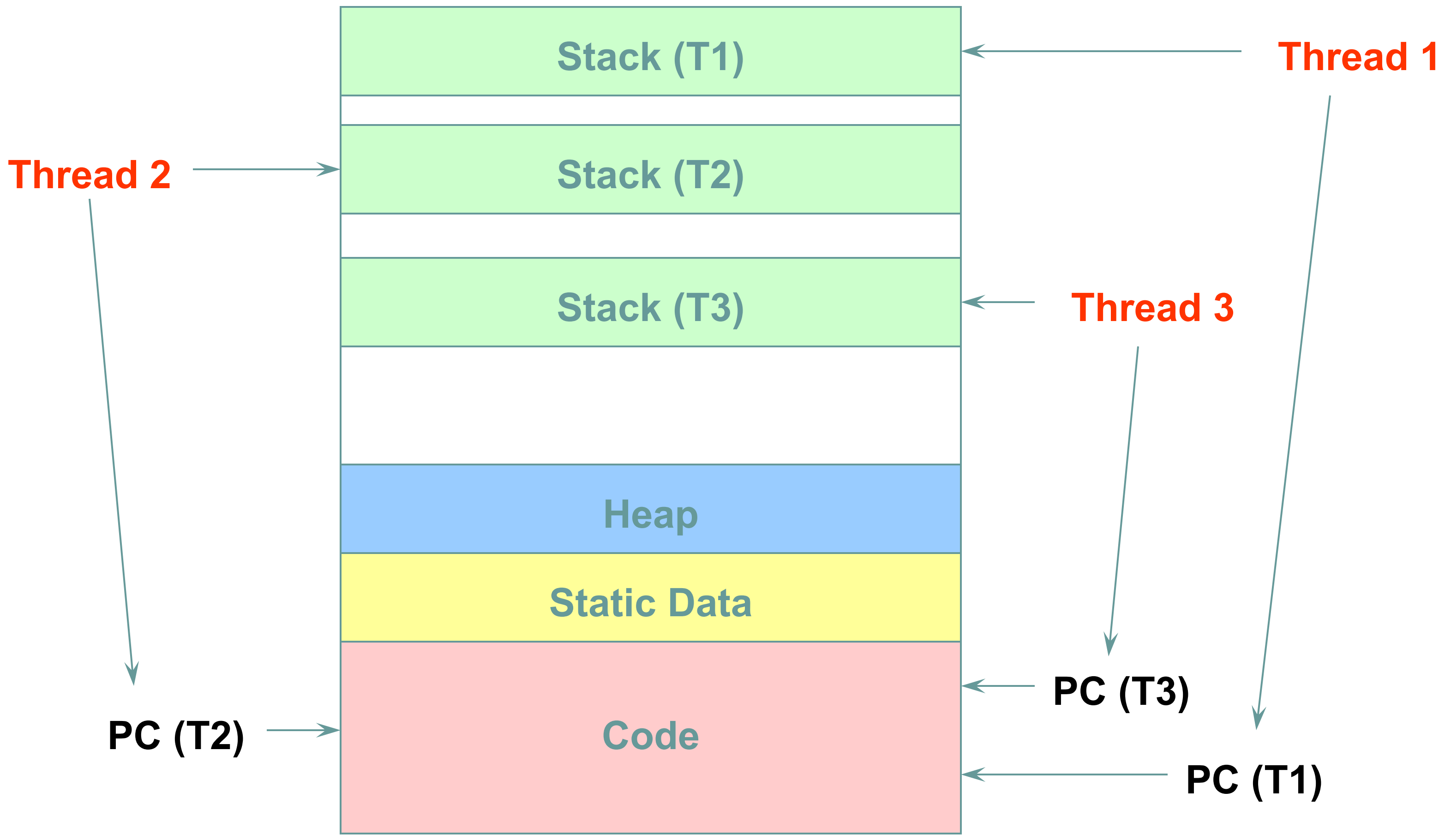
Process State Graph



Threads

- Separate dual roles of a process
  - Resource allocation unit and execution unit
  - A thread defines a sequential execution stream within a process (PC, SP, registers)
  - A process defines the address space, and resources (everything but threads of execution)
- A thread is bound to a single process
  - Processes, however, can have multiple threads
- Threads become the unit of scheduling
  - Processes are now the containers in which threads execute
  - Processes become static, threads are the dynamic entities

Threads in a Process



Today: CPU Scheduling

- Scheduler runs when we context switching among processes/threads on the ready queue
  - What should it do? Does it matter?
- Making the decision on what thread to run is called scheduling
  - What are the goals of scheduling?
  - What are common scheduling algorithms?
  - Lottery scheduling
  - Stride Scheduling
- Scheduling activations
  - User level vs. Kernel level scheduling of threads

Scheduling

- Right from the start of multiprogramming, scheduling was identified as a big issue
  - CCTS and Multics developed much of the classical algorithms
- Scheduling is a form of resource allocation
  - CPU is the resource
  - Resource allocation needed for other resources too; sometimes similar algorithms apply
- Requires mechanisms and policy
  - Mechanisms: Context switching, Timers, process queues, process state information, …
  - Scheduling looks at the policies: i.e., when to switch and which process/thread to run next

Preemptive vs. Non-preemptive scheduling

- In *preemptive* systems where we can interrupt a running job (involuntary context switch)
  - We're interested in such schedulers…

- In *non-preemptive* systems, the scheduler waits for a running job to give up CPU (voluntary context switch)
  - Was interesting in the days of batch multiprogramming
  - Some systems continue to use cooperative scheduling

- Example algorithms:
  - RR, FCFS, Shortest Job First (how to determine shortest), Priority Scheduling

Scheduling Goals

- What are some reasonable goals for a scheduler?
- Scheduling algorithms can have many different goals:
  - CPU utilization
  - Job throughput (# jobs/unit time)
  - Response time ($Avg(T_{ready})$: avg time spent on ready queue)
  - Fairness (or weighted fairness)
  - Other?

- Non-interactive applications:
  - Strive for job throughput, turnaround time (supercomputers)
- Interactive systems
  - Strive to minimize response time for interactive jobs
- Mix?

Goals II: Avoid Resource allocation pathologies

- Starvation no progress due to no access to resources
  - E.g., a high priority process always prevents a low priority process from running on the CPU
  - One thread always beats another when acquiring a lock
- Priority inversion
  - A low priority process running before a high priority one
  - Could be a real problem, especially in real time systems
    - Mars pathfinder: http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Authoritative_Account.html
- Other
  - Deadlock, livelock, …

Non-preemptive approaches

- Introduced just to have a baseline
- FIFO: schedule the processes in order of arrival
  - Comments?

- Shortest Job first
  - Comments?

Preemptive scheduling: Round Robin

- Each task gets resource for a fixed period of time (time quantum)
  - If task doesn't complete, it goes back in line
- Need to pick a time quantum
  - What if time quantum is too long?
    - Infinite?
  - What if time quantum is too short?
    - One instruction?

Priority Scheduling

- Priority Scheduling
  - Choose next job based on priority
    - Airline check-in for first class passengers
  - Can implement SJF, priority = 1/(expected CPU burst)
  - Also can be either preemptive or non-preemptive
- Problem?
  - Starvation – low priority jobs can wait indefinitely
- Solution
  - “Age” processes
    - Increase priority as a function of waiting time
    - Decrease priority as a function of CPU consumption

Combining Algorithms

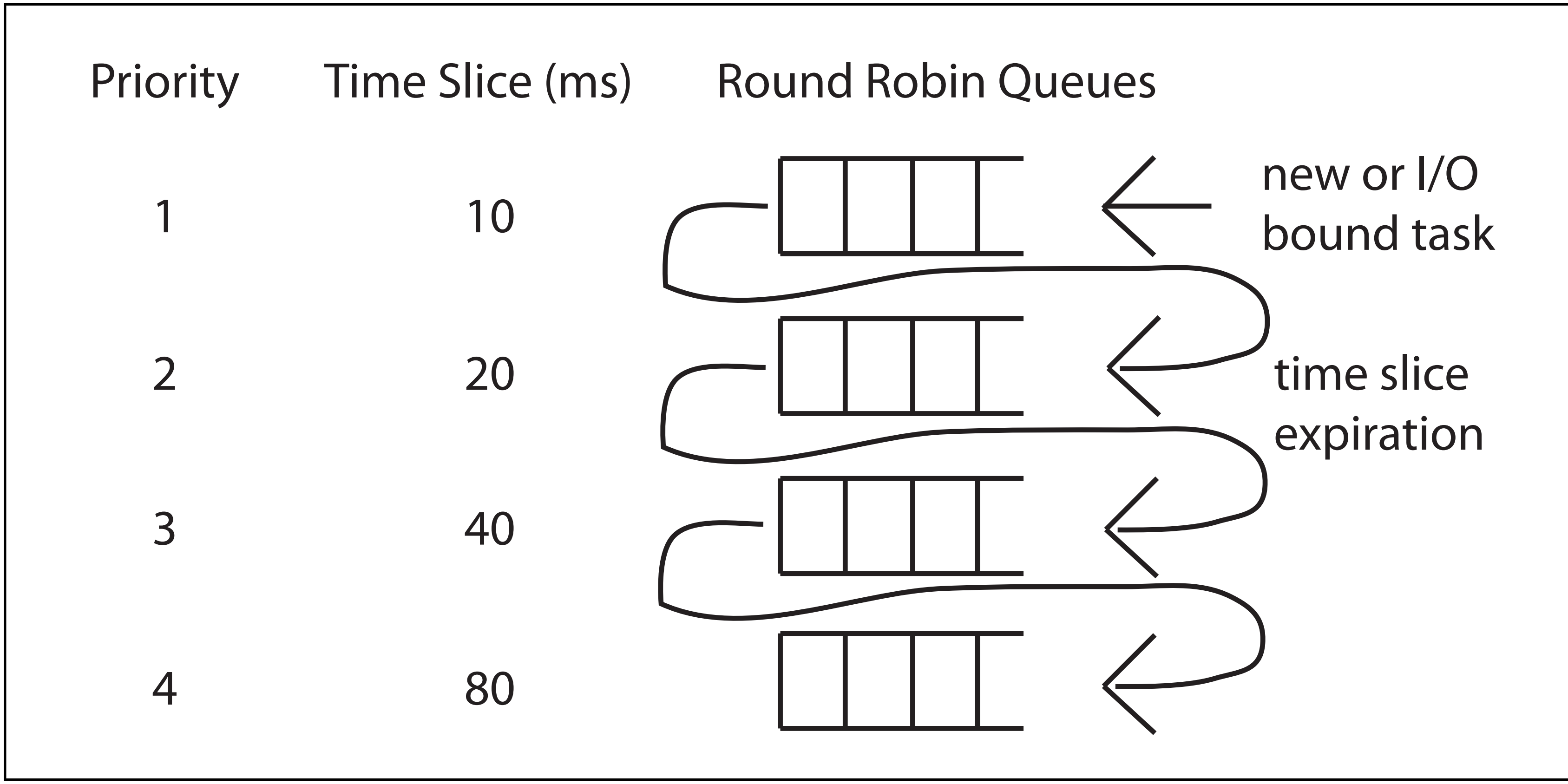
- Scheduling algorithms can be combined
  - Have multiple queues
  - Use a different algorithm for each queue
  - Move processes among queues

- Example: Multiple-level feedback queues (MLFQ)
  - Multiple queues representing different job types
    - Interactive, CPU-bound, batch, system, etc.
  - Queues have priorities, jobs on same queue scheduled RR
  - Jobs can move among queues based upon execution history
    - Feedback: Switch from interactive to CPU-bound behavior

Multi-level Feedback Queue (MFQ)

- Goals:
  - Responsiveness
  - Low overhead
  - Starvation freedom
  - Some tasks are high/low priority
  - Fairness (among equal priority tasks)
- Not perfect at any of them!
  - Used in Unix (and Windows and MacOS)

MFQ



Priority
Time Slice (ms)
Round Robin Queues
1
10
2
20
3
40
4
80
new or I/O bound task
time slice expiration

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

Linux scheduler

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

Problems with Traditional schedulers

- Priority systems are ad hoc: highest priority always wins
- Try to support fair share by adjusting priorities with a feedback loop
  - Works over long term
  - highest priority still wins all the time, but now the Unix priorities are always 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

It’s 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
  $\boldsymbol{\sigma/np = \sqrt{(1-p)/np}}$
  - 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

- Let 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 thread 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 process' tickets expressed in the base currency is still equal to 10

Compensation tickets (I)

- I/O-bound threads 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 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

Compensation tickets (III)

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

Implementation

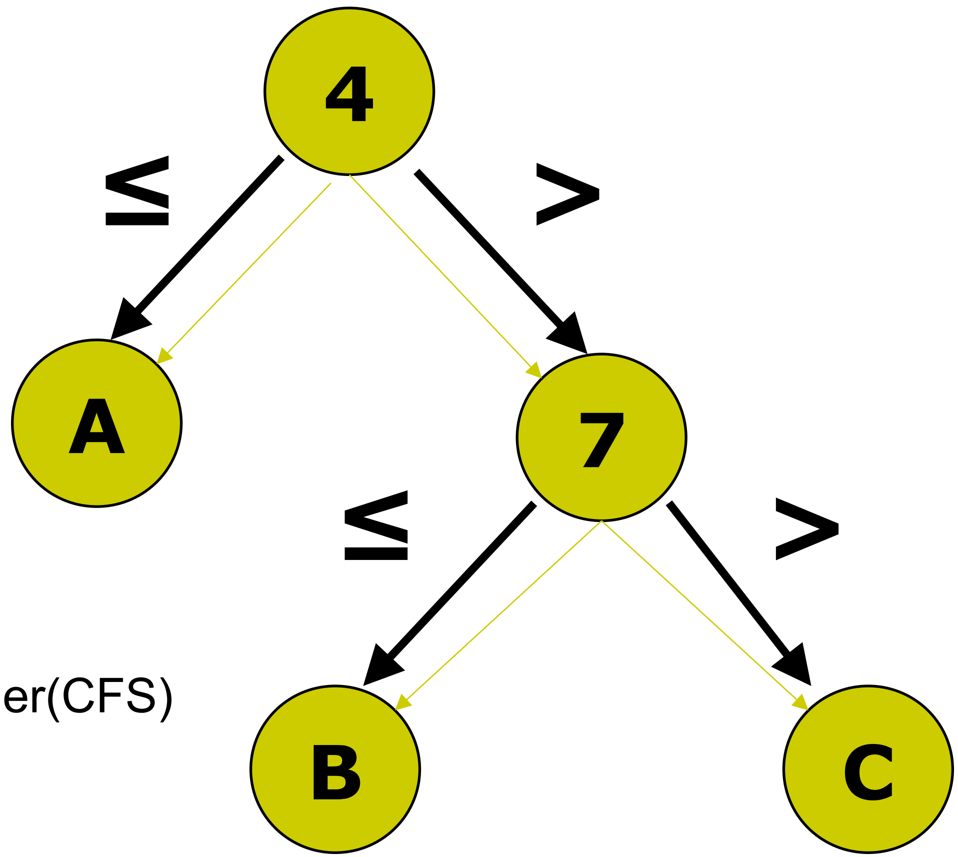
- On a MIPS-based DEC station running Mach 3 microkernel
  - Time slice is 100ms
    - *Fairly large as scheme does not allow preemption*
- Requires
  - A fast RNG
  - A fast way to pick lottery winner

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)

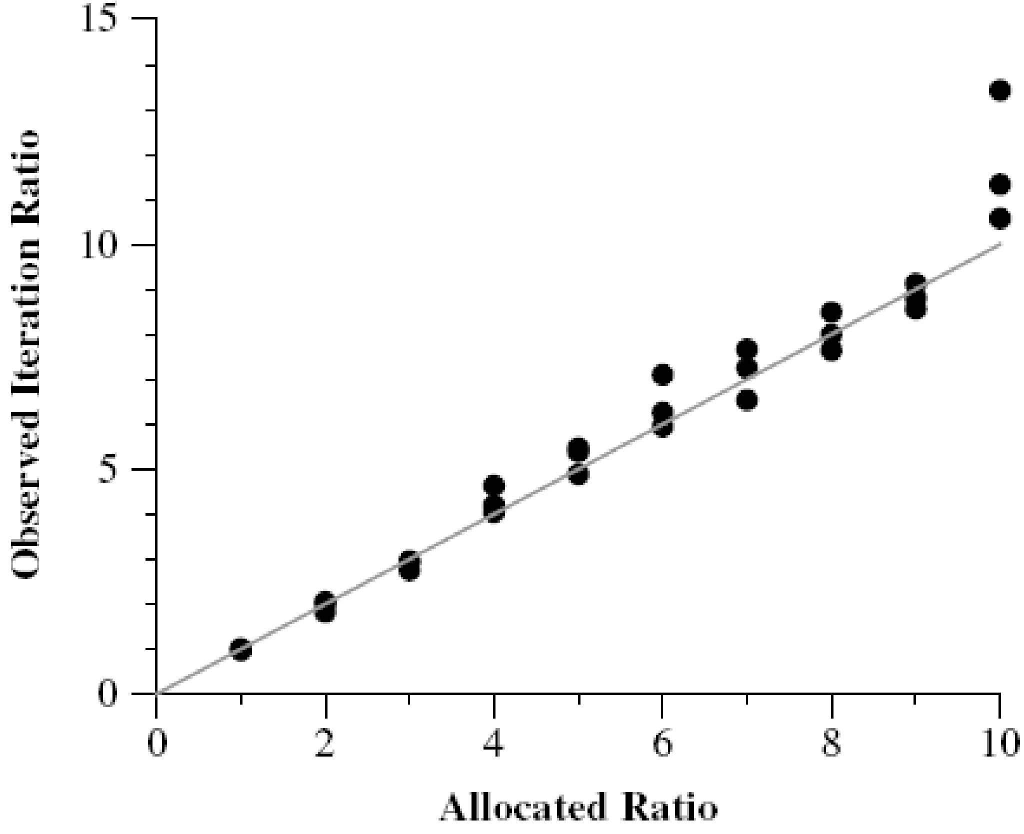
Search time is O(n) where n is list length

Optimization – use tree

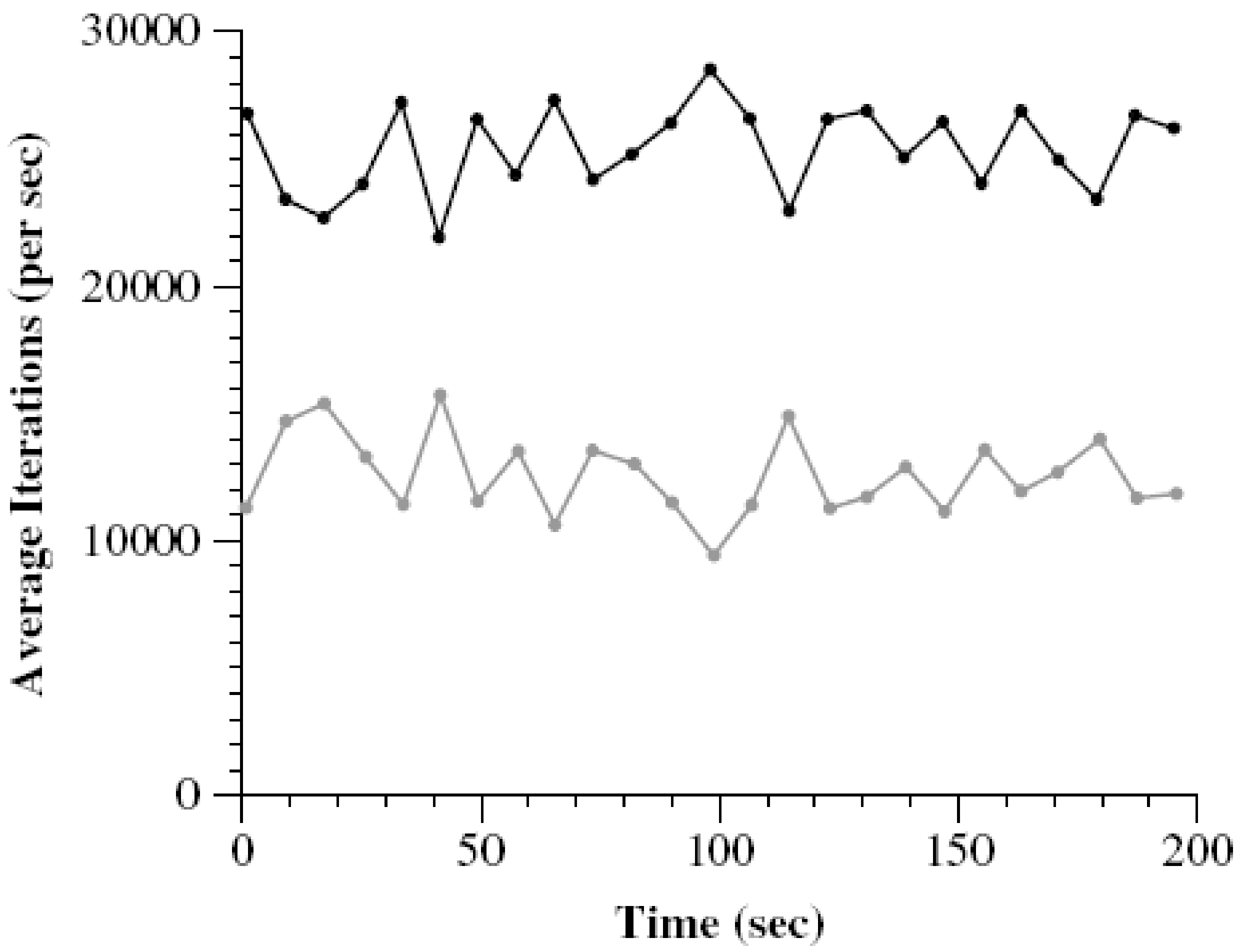


RB Tree used in Linux
Completely fair scheduler(CFS)
--not lottery based

Long-term fairness (I)



Short term fluctuations



For
2:1
ticket
alloc.
ratio

Stride scheduling

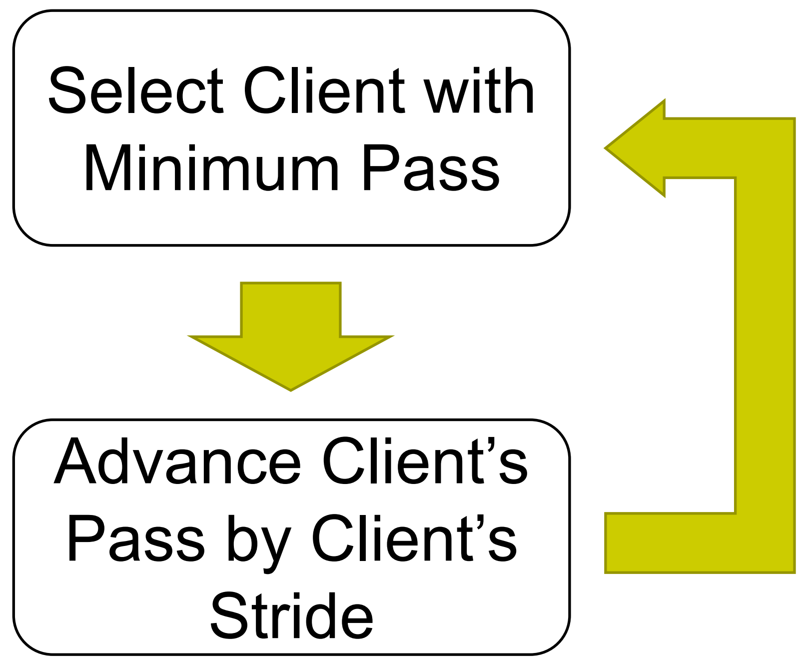
- Deterministic version of lottery scheduling
- Mark time virtually (counting passes)
  - Each process has a stride: number of passes between being scheduled
  - Stride inversely proportional to number of tickets
  - Regular, predictable schedule
- Can also use compensation tickets
- Similar to weighted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

Stride Scheduling – Basic Algorithm

Client Variables:

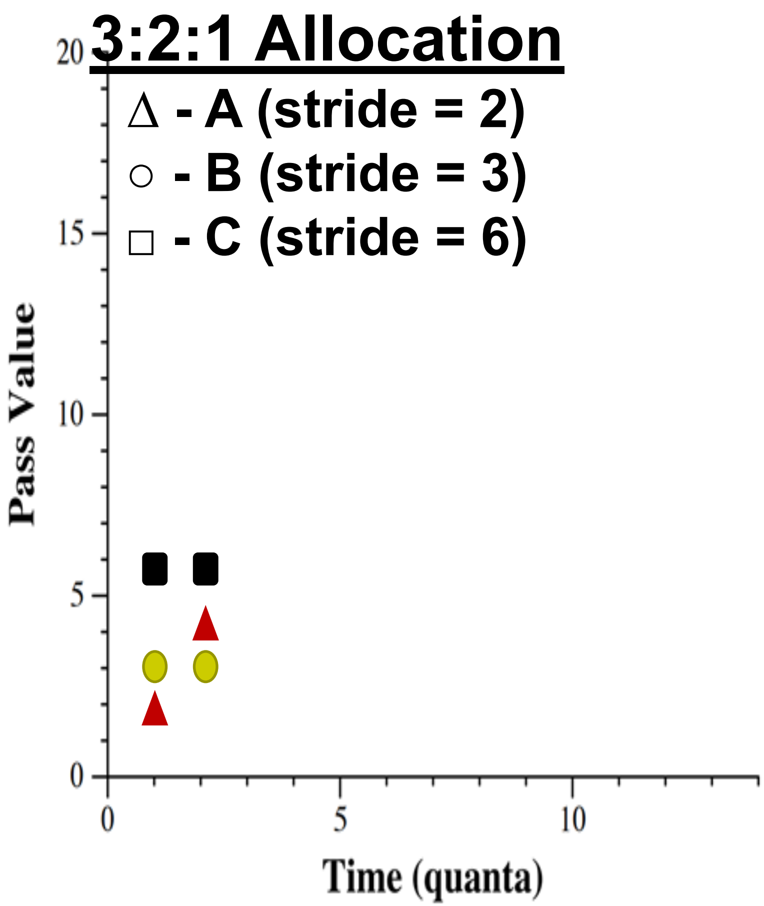
- Tickets
  - Relative resource allocation
- Strides (
  - Interval between selection
- Pass (
  - Virtual index of next selection

- minimum ticket allocation



Slide and example from Dong-hyeon Park

Stride Scheduling – Basic Algorithm

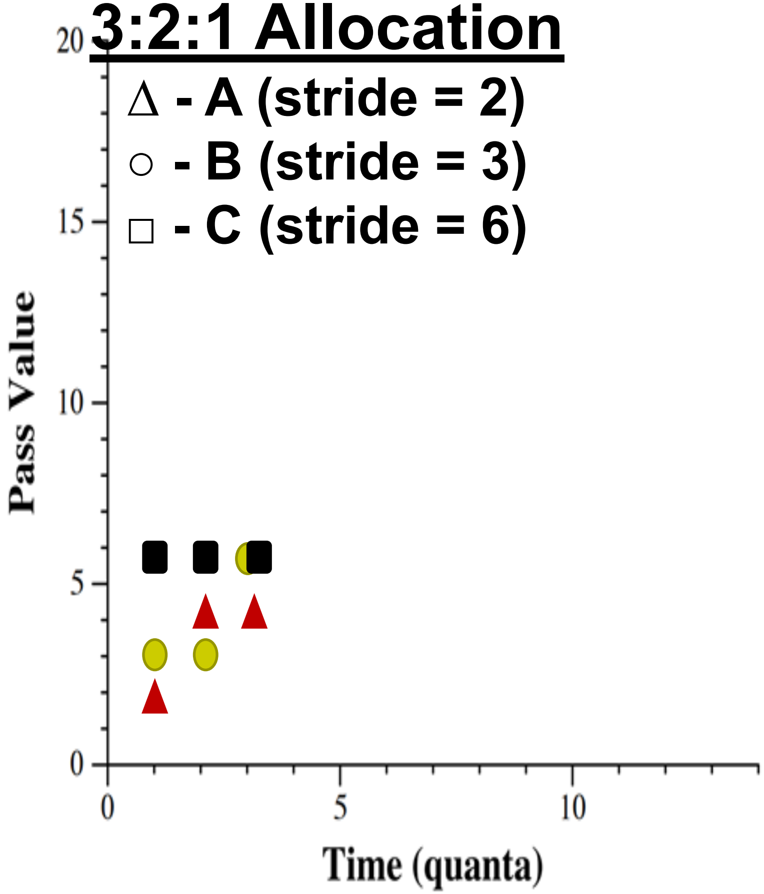


Time 1:	2	3	6
	+2		
Time 2:	4	3	6

Stride Scheduling – Basic Algorithm

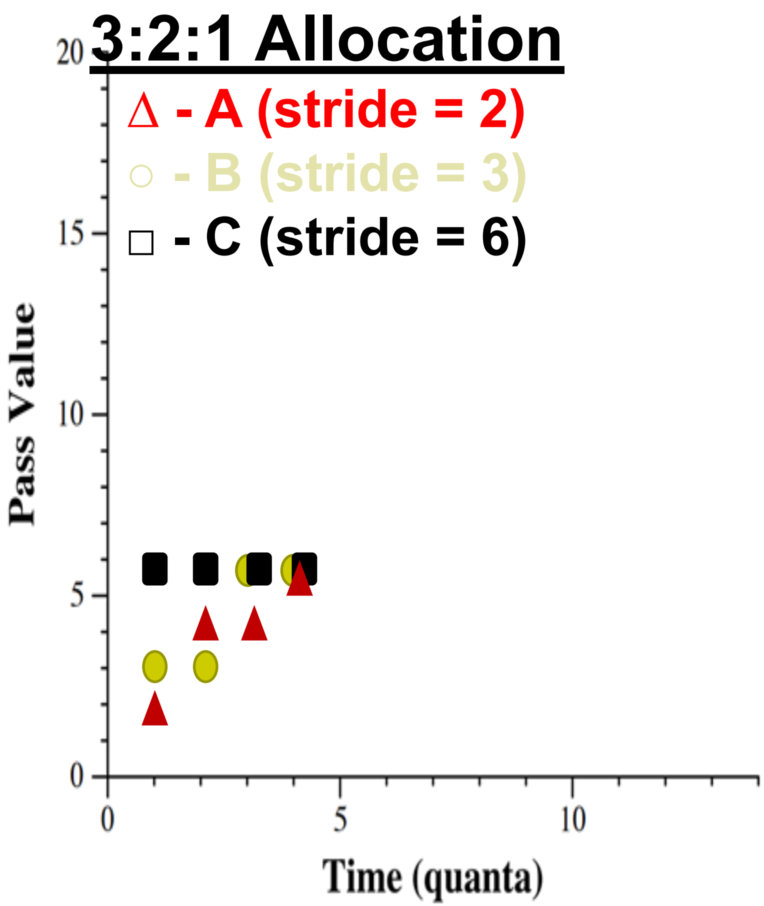
3:2:1 Allocation

△ - A (stride = 2)
○ - B (stride = 3)
□ - C (stride = 6)



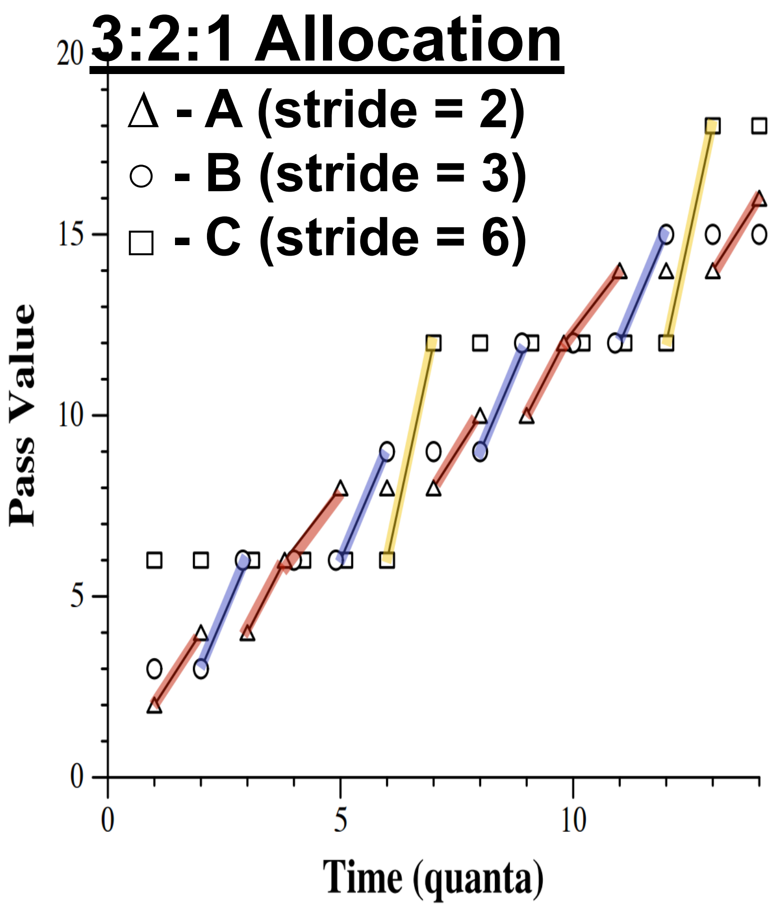
Time 1:	2	3	6
	+2		
Time 2:	4	3	6
		+3	
Time 3:	4	6	6

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

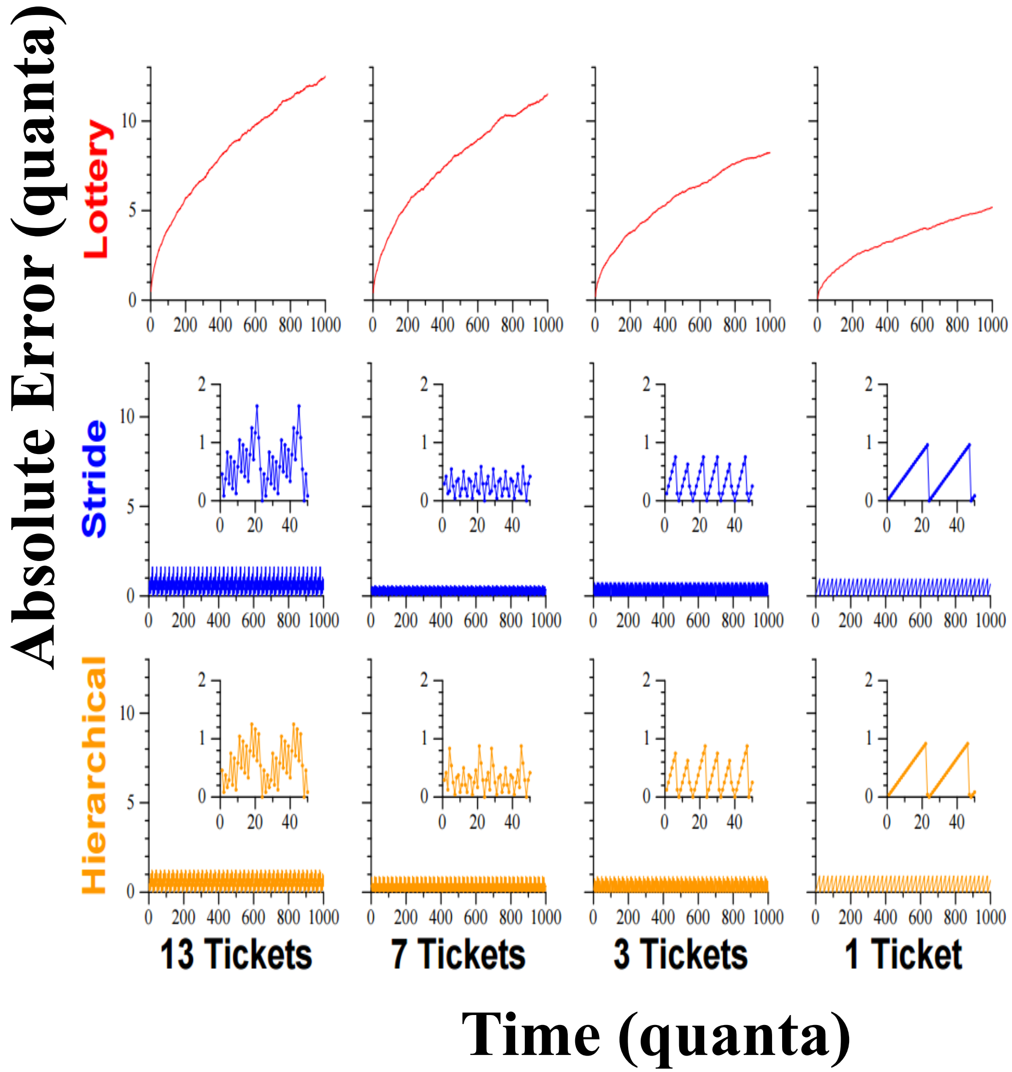
Stride Scheduling – Basic Algorithm



	▲	●	■
Time 1:	2	3	6
	+2		
Time 2:	4	3	6
		+3	
Time 3:	4	6	6
	+2		
Time 4:	6	6	6

⋮

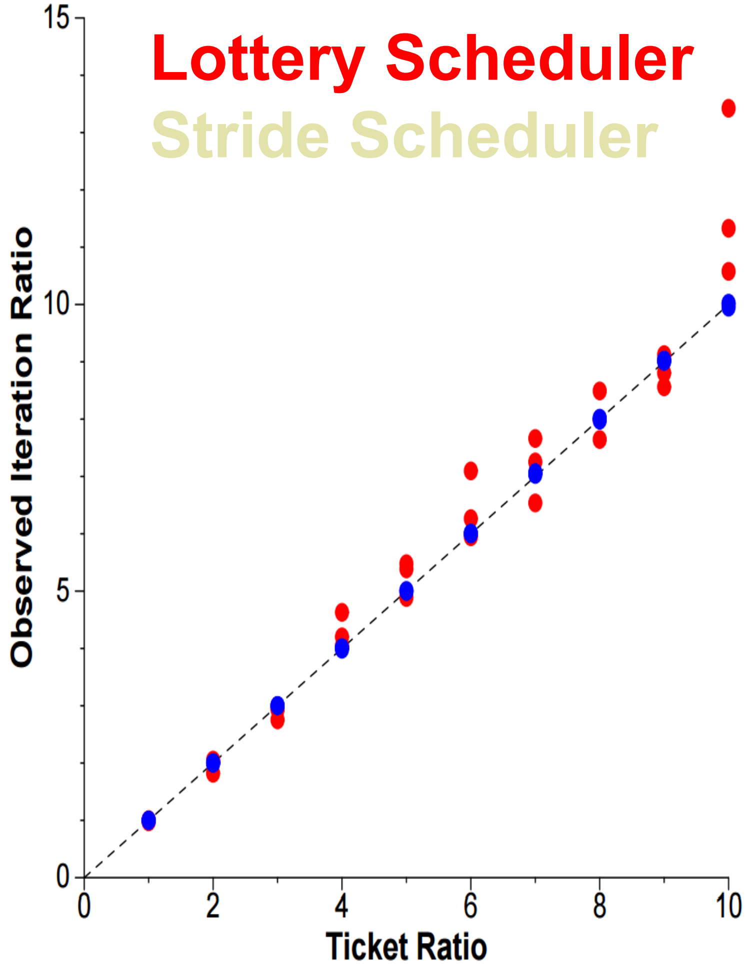
Throughput Error Comparison



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?