Scheduler Design

CS 202: Advanced Operating Systems

Outline

- Proportional share scheduling
  - Lottery Scheduling
  - Stride Scheduling
  - Linux CFS

- Threading model
  - User-level vs. Kernel-level threads
  - Scheduler Activation

Problems with Traditional schedulers

- Priority assignment schemes are often ad hoc

Problems with Traditional schedulers

- Priority assignment schemes are often ad hoc
- Unfairness/Starvation
  - Highest priority always wins
  - Try to support fair share by adjusting priorities with a feedback loop?
    - May work over long term
    - But highest priority still wins all the time

Problems with Traditional schedulers

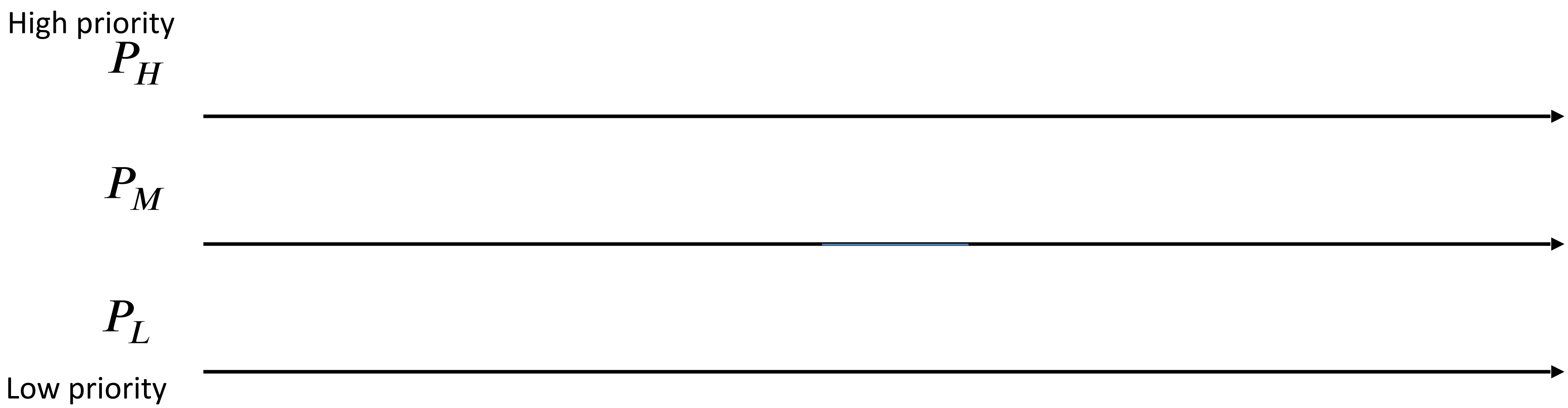
- Priority assignment schemes are often ad hoc
- Unfairness/Starvation
  - Highest priority always wins
  - Try to support fair share by adjusting priorities with a feedback loop?
    - May work over long term
    - But highest priority still wins all the time
- Priority inversion: high-priority jobs can be blocked indirectly by low-priority jobs
  - Effectively inverting assigned priorities; Unpredictability ↑

Problems with Traditional schedulers

- Priority assignment schemes are often ad hoc
- Unfairness/Starvation
  - Highest priority always wins
  - Try to support fair share by adjusting priorities with a feedback loop?
    - May work over long term
    - But highest priority still wins all the time
- Priority inversion: high-priority jobs can be blocked indirectly by low-priority jobs
  - Effectively inverting assigned priorities; Unpredictability ↑
- Schedulers are complex and difficult to control

Priority Inversion

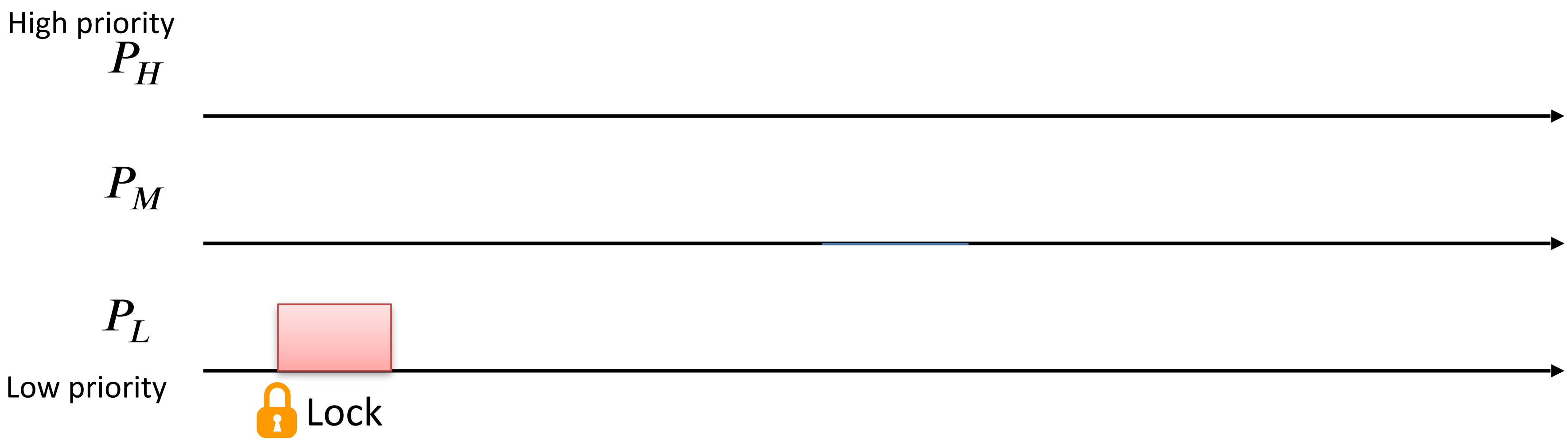
- Processes P_H and P_L share a mutex lock



- Known solution: Priority Inheritance Protocol
  - P_L blocks higher-priority processes, let P_L use the highest priority of the blocked processes → P_M can no longer preempt P_L

Priority Inversion

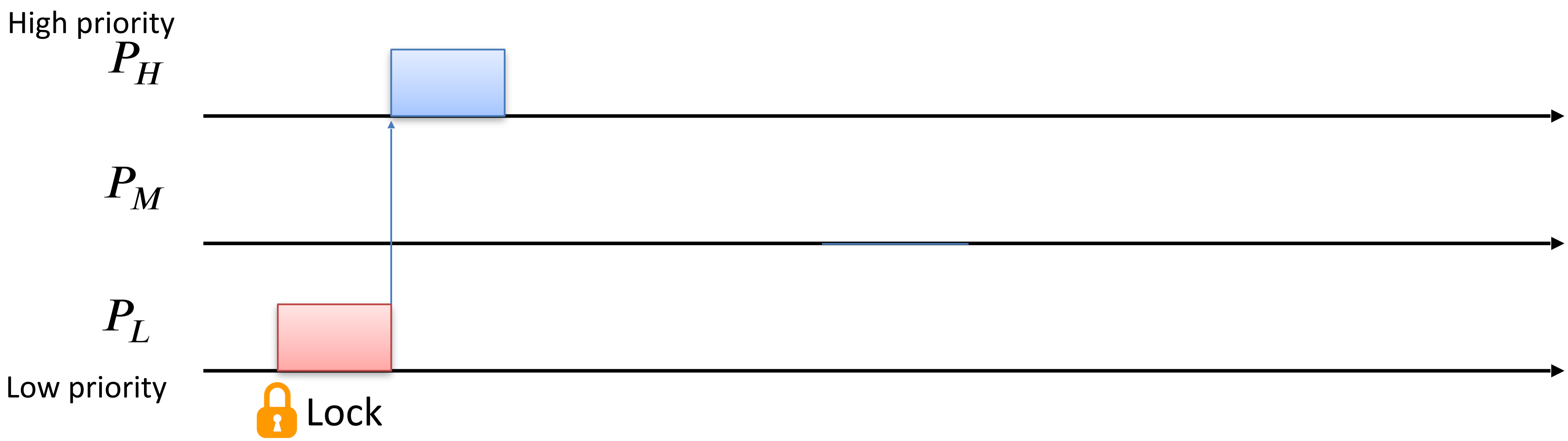
- Processes P_H and P_L share a mutex lock



- Known solution: Priority Inheritance Protocol
  - P_L blocks higher-priority processes, let P_L use the highest priority of the blocked processes → P_M can no longer preempt P_L

Priority Inversion

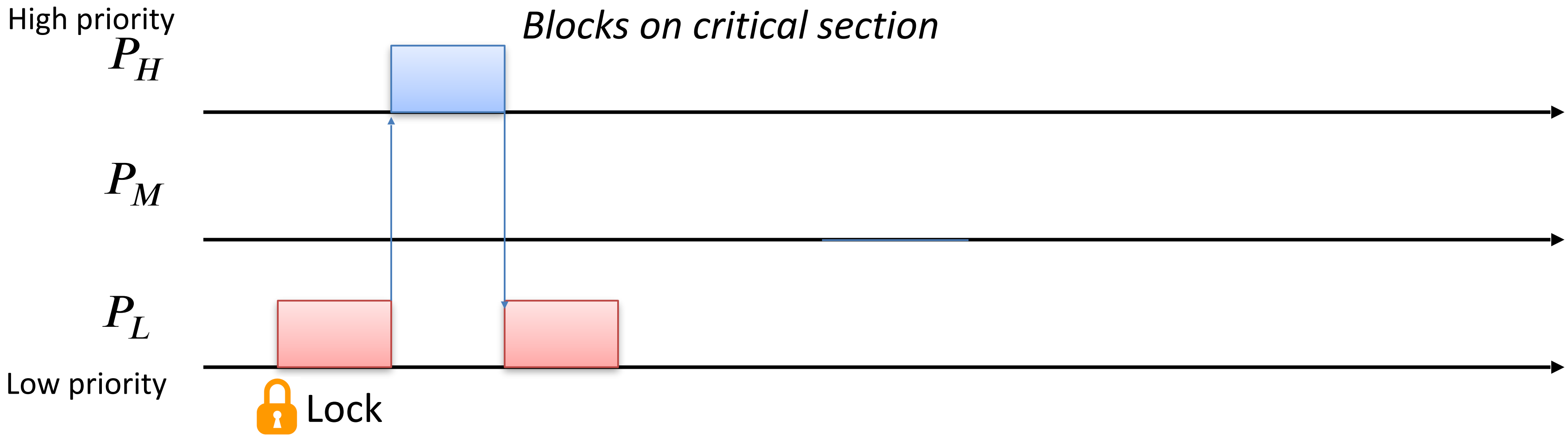
- Processes P_H and P_L share a mutex lock



- Known solution: Priority Inheritance Protocol
  - P_L blocks higher-priority processes, let P_L use the highest priority of the blocked processes → P_M can no longer preempt P_L

Priority Inversion

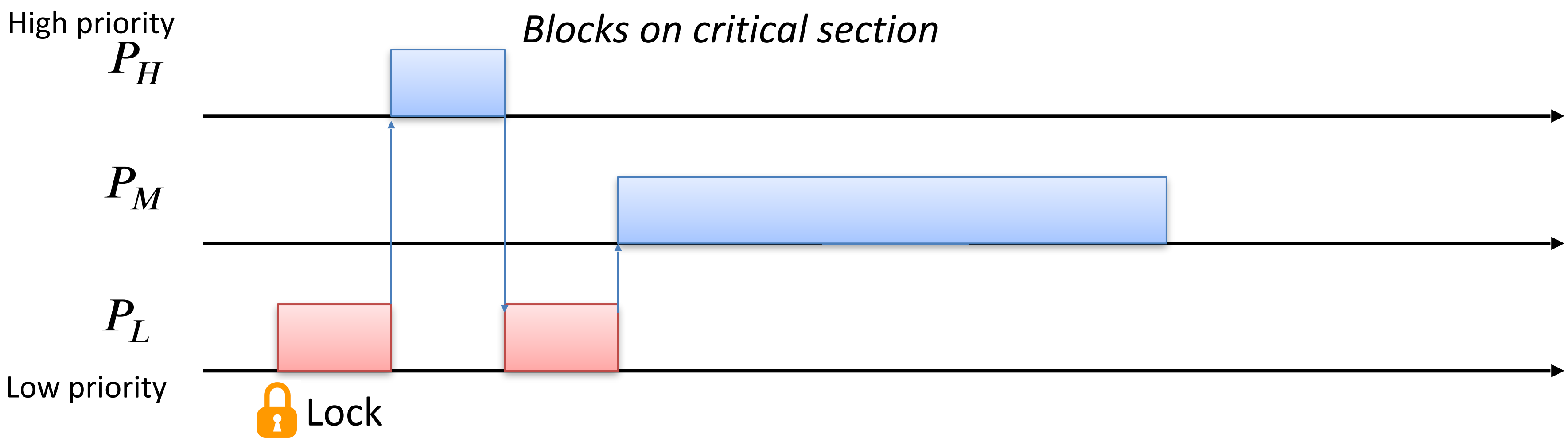
- Processes P_H and P_L share a mutex lock



- Known solution: Priority Inheritance Protocol
  - P_L blocks higher-priority processes, let P_L use the highest priority of the blocked processes → P_M can no longer preempt P_L

Priority Inversion

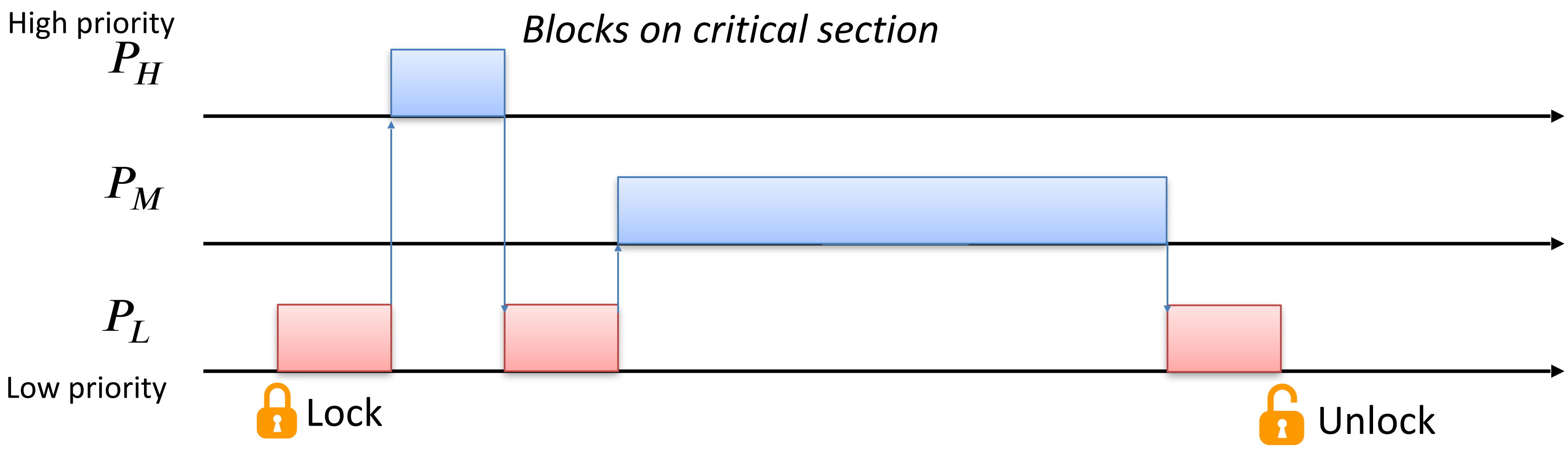
- Processes P_H and P_L share a mutex lock



- Known solution: Priority Inheritance Protocol
  - P_L blocks higher-priority processes, let P_L use the highest priority of the blocked processes → P_M can no longer preempt P_L

Priority Inversion

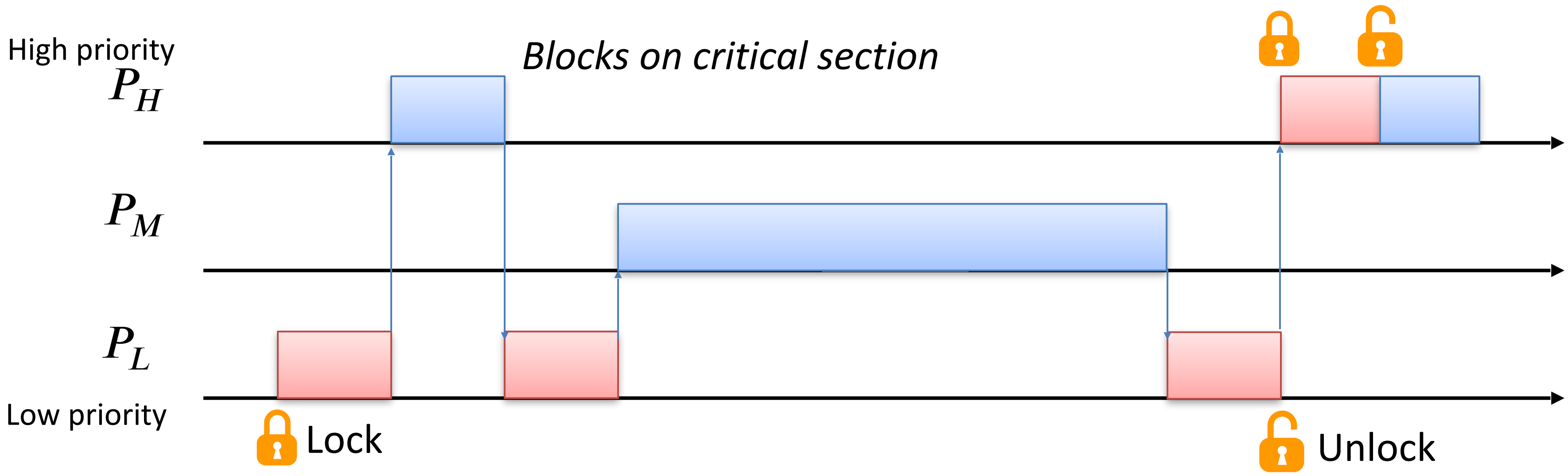
- Processes P_H and P_L share a mutex lock



- Known solution: Priority Inheritance Protocol
  - P_L blocks higher-priority processes, let P_L use the highest priority of the blocked processes → P_M can no longer preempt P_L

Priority Inversion

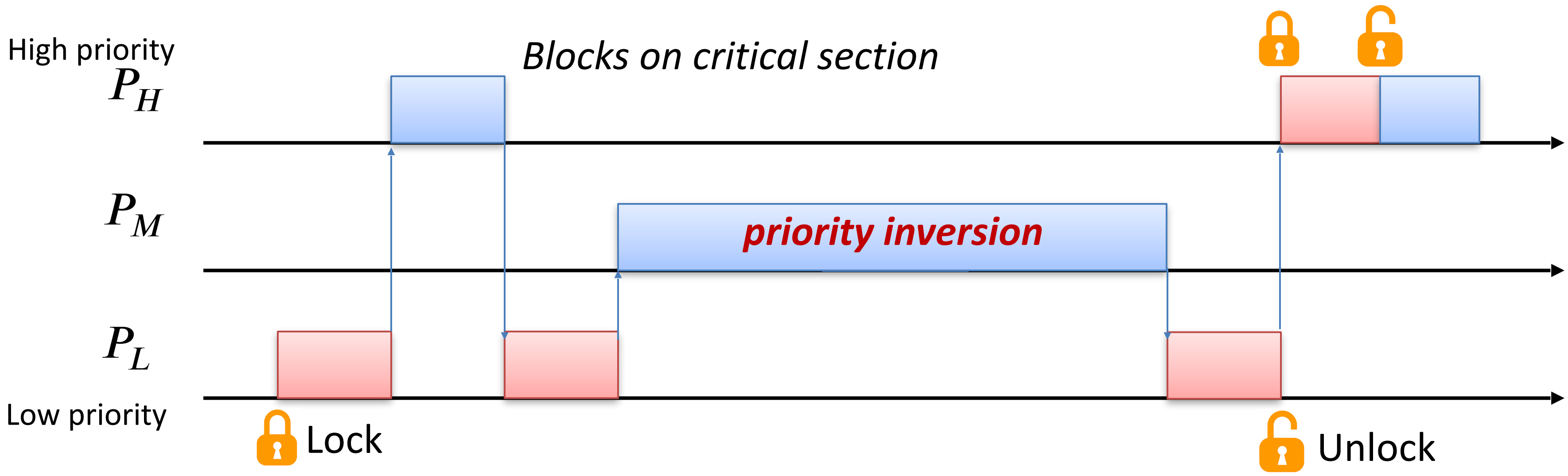
- Processes P_H and P_L share a mutex lock



- Known solution: Priority Inheritance Protocol
  - P_L blocks higher-priority processes, let P_L use the highest priority of the blocked processes → P_M can no longer preempt P_L

Priority Inversion

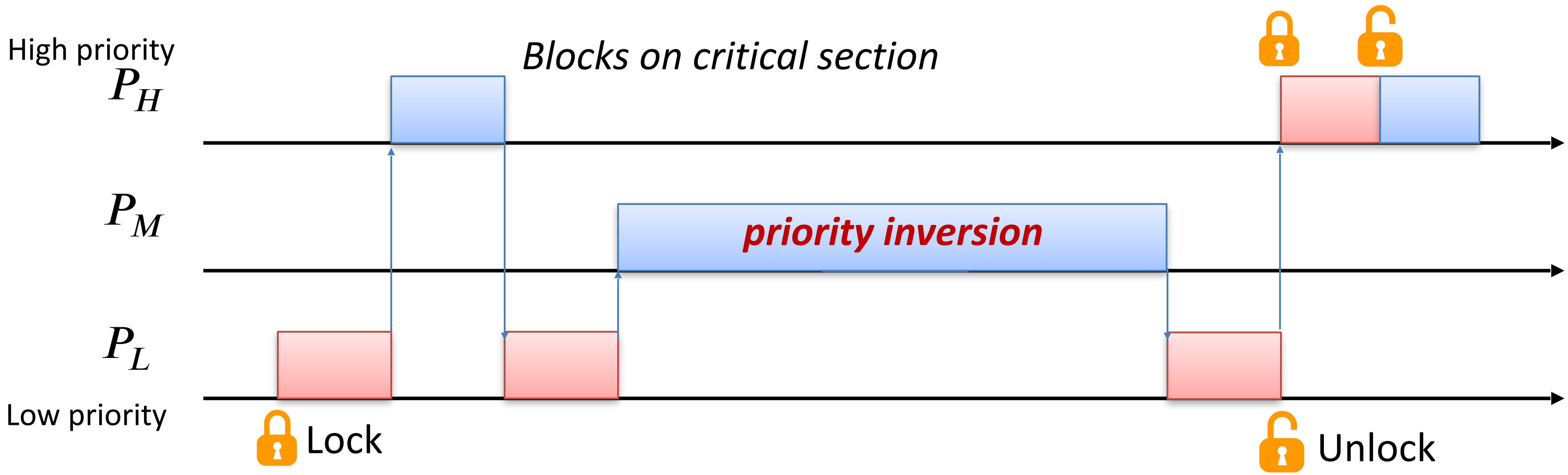
- Processes P_H and P_L share a mutex lock



- Known solution: Priority Inheritance Protocol
  - P_L blocks higher-priority processes, let P_L use the highest priority of the blocked processes → P_M can no longer preempt P_L

Priority Inversion

- Processes P_H and P_L share a mutex lock



- High-priority process gets delayed and can be left starved
- Serious problem in real-time systems (e.g., Mars Pathfinder)

- Known solution: Priority Inheritance Protocol
  – P_L blocks higher-priority processes, let P_L use the highest priority of the blocked processes → P_M can no longer preempt P_L

Lottery Scheduling

Lottery Scheduling: Flexible Proportional-Share Resource Management

Carl A. Waldspurger * William E. Weihl *

MIT Laboratory for Computer Science
Cambridge, MA 02139 USA

Lottery scheduling

- Key idea: Give each process a bunch of tickets
  - Every time slice, the scheduler holds a lottery
  - The process holding the winning ticket gets to run
- Chance to get scheduled is determined by # of tickers
  - Elegant way to implement fair-share scheduling
- Tickets can be used for a variety of resources

Example

- Three processes
  - A has 5 tickets
  - B has 3 tickets
  - C has 2 tickets

10 tickets in total

- 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 (3 out of 5)

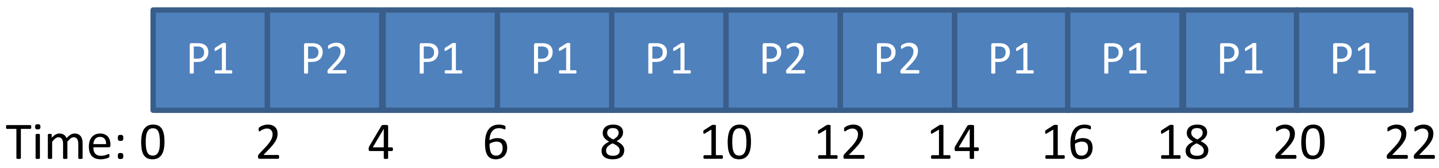
Example

Process	Arrival Time	Ticket Range
P1	0	0-74 (75 total)
P2	0	75-99 (25 total)

Example

Process	Arrival Time	Ticket Range
P1	0	0-74 (75 total)
P2	0	75-99 (25 total)

- **P1 ran 8 of 11 slices – 72%**
- **P2 ran 3 of 11 slices – 27%**



Example

Process	Arrival Time	Ticket Range
P1	0	0-74 (75 total)
P2	0	75-99 (25 total)

- **P1 ran 8 of 11 slices – 72%**
- **P2 ran 3 of 11 slices – 27%**

P1	P2	P1	P1	P1	P2	P2	P1	P1	P1	P1

Time: 0 2 4 6 8 10 12 14 16 18 20 22

- Probabilistic scheduling
  – Over time, run time for each process converges to the correct value (i.e. the # of tickets it holds)

It's fair

- Lottery scheduling is ***probabilistically fair***

- If a process has a t tickets out of T
  - Probability of winning a lottery: $p = t/T$
  - Expected number of wins over n drawings: np
    - Throughput (share of resource) improves with t
    - Binomial distribution, Variance $\sigma^2 = np(1-p)$
  - Coefficient of variation (CV) for the number of wins
    - $\sigma/np = \sqrt{(1-p)/np}$
    - CV decreases with $\sqrt{n}$ → Accuracy improves with $\sqrt{n}$

Fairness (II)

- Number of tries required to win the first lottery follows a *geometric distribution*
  - Average response time is inversely proportional to its ticket allocation
- As time passes, each process ends up receiving its share of the resource

Implementation Advantages

Implementation Advantages

- Very fast scheduler execution
  - All the scheduler needs to do is run *random()*
  - No need to manage *O(log N)* priority queues

Implementation Advantages

- Very fast scheduler execution
  - All the scheduler needs to do is run *random()*
  - No need to manage *O(log N)* priority queues
- No need to manage lots of scheduling state
  - Scheduler needs to know the total number of tickets
  - No need to track process behavior or history

Implementation Advantages

- Very fast scheduler execution
  - All the scheduler needs to do is run *random()*
  - No need to manage *O(log N)* priority queues
- No need to manage lots of scheduling state
  - Scheduler needs to know the total number of tickets
  - No need to track process behavior or history
- Automatically balances CPU time across processes
  - New processes get some tickets, adjust the overall size of the ticket pool

Implementation Advantages

- Very fast scheduler execution
  - All the scheduler needs to do is run *random()*
  - No need to manage *O(log N)* priority queues
- No need to manage lots of scheduling state
  - Scheduler needs to know the total number of tickets
  - No need to track process behavior or history
- Automatically balances CPU time across processes
  - New processes get some tickets, adjust the overall size of the ticket pool
- Easy to prioritize processes
  - Give high-priority processes many tickets
  - Give low-priority processes a few tickets
  - Priorities can change via *ticket transfers*

Ticket transfers

Ticket transfers

- How to deal with dependencies?
  - e.g., a client waits for a reply from a server

Ticket transfers

- How to deal with dependencies?
  - e.g., a client waits for a reply from a server
- Ticket transfers: Explicit transfers of tickets from one process to another
  - Can be used whenever a process blocks due to some dependency
  - When a client waits for a reply from a server, it can temporarily transfer its tickets to the server
    - Increases server priority (server's chance to win the lottery)

Ticket transfers

- How to deal with dependencies?
  - e.g., a client waits for a reply from a server
- Ticket transfers: Explicit transfers of tickets from one process to another
  - Can be used whenever a process blocks due to some dependency
  - When a client waits for a reply from a server, it can temporarily transfer its tickets to the server
    - Increases server priority (server's chance to win the lottery)
- Similar to priority inheritance
  - Can solve priority inversion

Ticket inflation

- Alternative to explicit ticket transfers
- Escalate one's resource rights by creating more tickets
  - Allow mutually-trusting clients in the same group adjust their priorities dynamically without explicit communication
    - Group: threads/processes of the same application or the same user
  - But now this group has more tickets than other groups

    → Ticket inflation!

- <u>Currencies</u>: Set up an exchange rate between groups
  - Enables inflation to be contained within a certain group of users or processes

Example (I)

- A process has three threads:
  - A has 5 tickets
  - B has 3 tickets
  - C has 2 tickets
- It creates 10 extra tickets and gives them to Thread C
  - In what cases?
  - Process now has 20 tickets in total

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's tickets expressed in the base currency is still equal to 10
  - This is the value used when competing with *other* processes
  - Ticket inflation is locally contained within the process boundary

Compensation tickets (I)

- Problem: 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 time quantum ***can*** be granted a ***compensation ticket***
  - Ticket inflates the value by ***1/f*** until the client starts gets the CPU

- Compensation tickets
  - Favor I/O-bound and interactive threads
  - Help them getting their fair share of 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

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 really fair? Can a process cheat?
  - What if A alternates between 1/5 and full quantum? (imagine a CPU-bound client B with the same # of tickets)

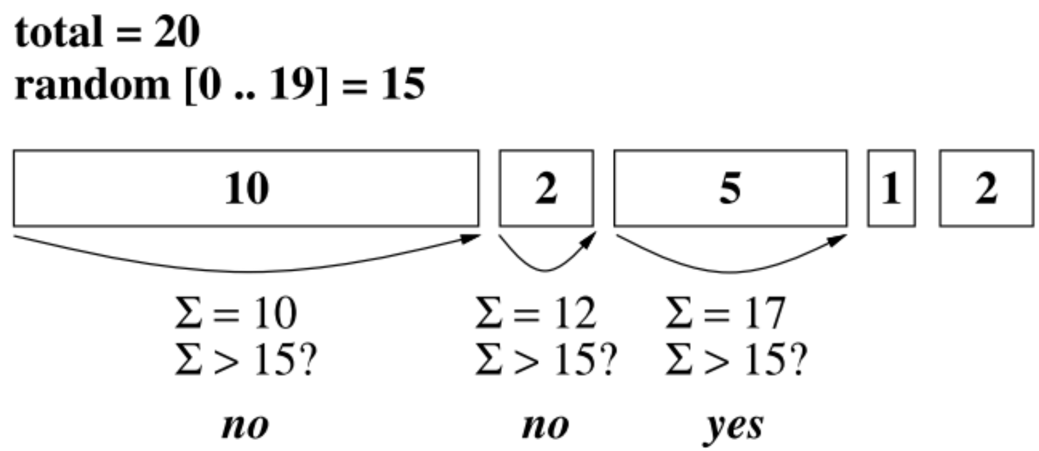
Implementation

- On a MIPS-based DEC station running Mach 3 microkernel
  - Time slice is 100ms
- Requires
  - A fast random number generator
  - A fast way to pick a lottery winner

Example: List-based lottery

- Five clients

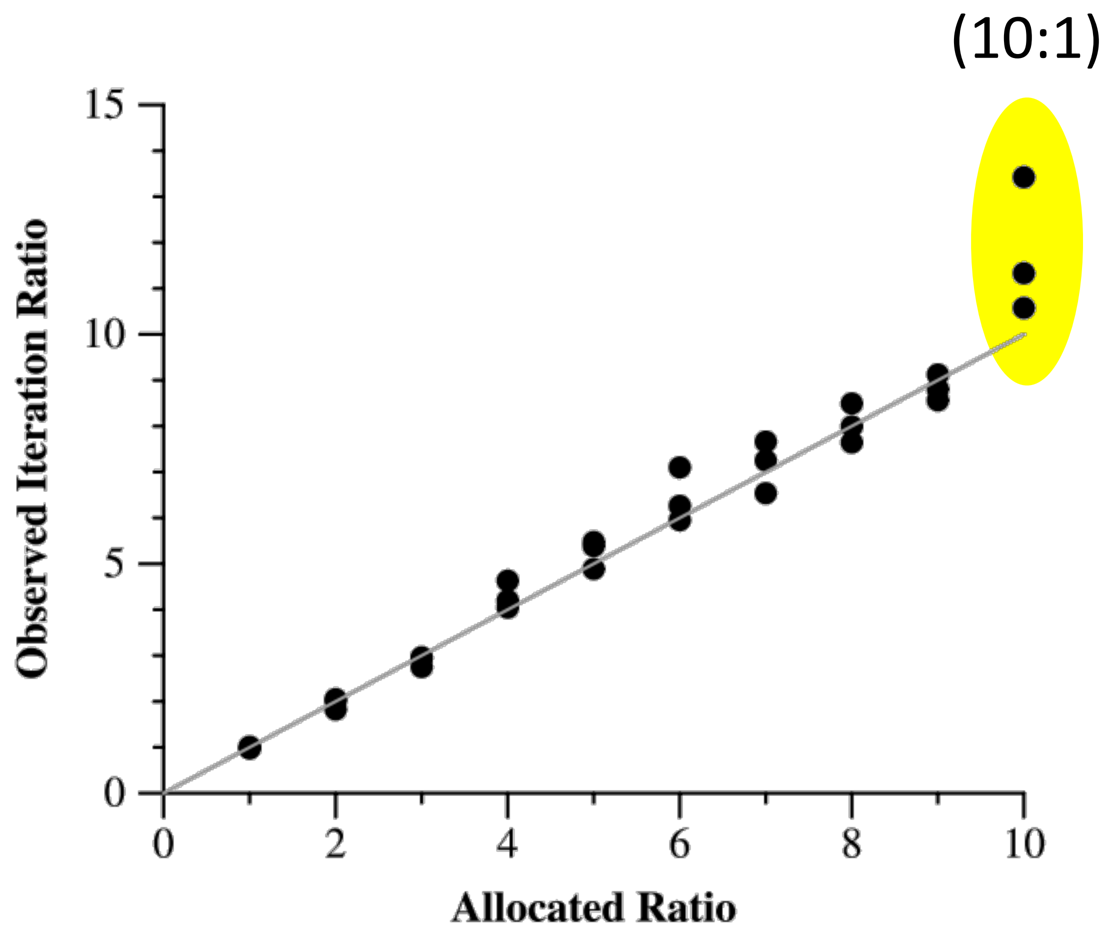
Client ticket list:



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

- More efficient implementation: using *trees*
  - O(log *n*) is possible

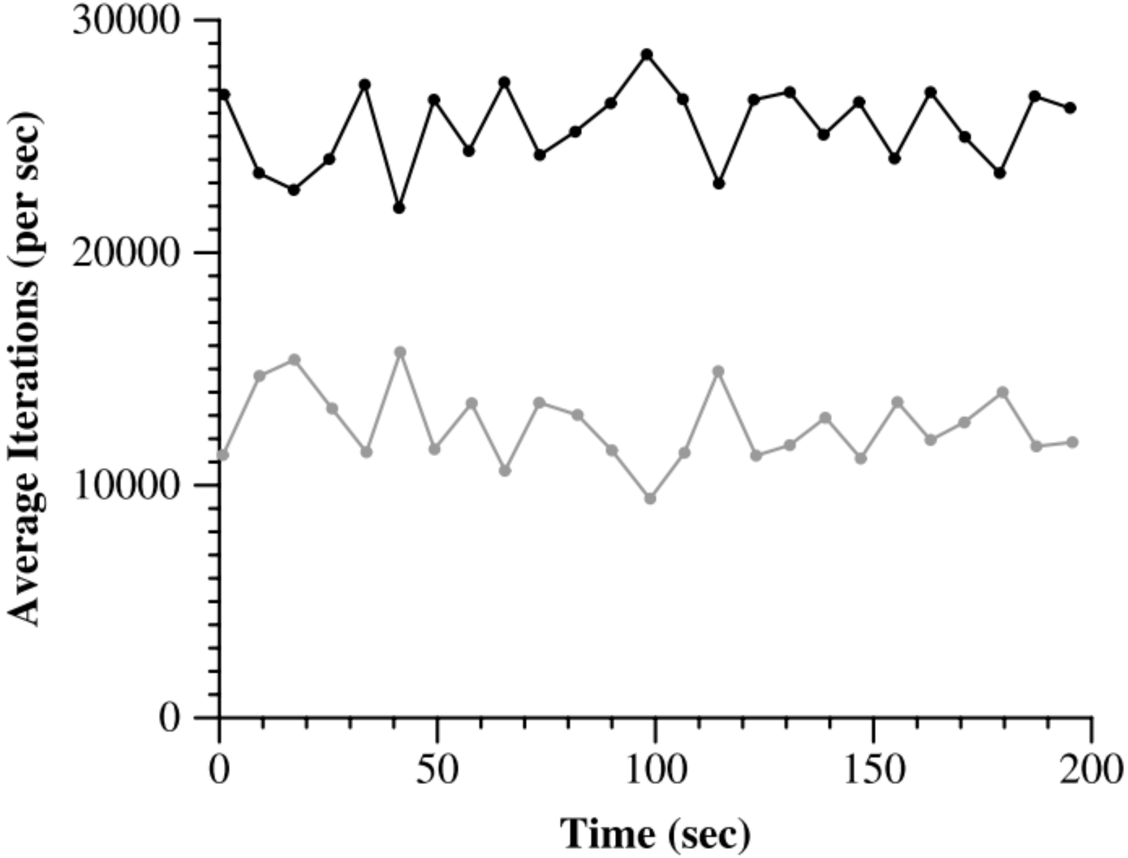
Long-term fairness (I)



(three 60 second runs)

Short term fluctuations

For 2:1 ticket allocation ratio



(single 200 second run)

Discussion

Discussion

- Fairness not great
  - Mutex 1.8:1 instead of 2:1
  - Multimedia apps 1.9:1.5:1 instead of 3:2:1

Discussion

- Fairness not great
  - Mutex 1.8:1 instead of 2:1
  - Multimedia apps 1.9:1.5:1 instead of 3:2:1
- Real time? Multiprocessor?

Discussion

- Fairness not great
  - Mutex 1.8:1 instead of 2:1
  - Multimedia apps 1.9:1.5:1 instead of 3:2:1
- Real time? Multiprocessor?
- Short term unfairness
  - This leads to *stride scheduling* from same authors

Stride Scheduling

Stride Scheduling:
Deterministic Proportional-Share Resource Management

Carl A. Waldspurger *   William E. Weihl *

Technical Memorandum MIT/LCS/TM-528
MIT Laboratory for Computer Science
Cambridge, MA 02139

Stride scheduling

- Deterministic version of lottery scheduling
  - Randomness does not guarantee fairness

Stride scheduling

- Deterministic version of lottery scheduling
  - Randomness does not guarantee fairness
- Stride scheduling:
  - Each process is given some tickets

Stride scheduling

- Deterministic version of lottery scheduling
  - Randomness does not guarantee fairness
- Stride scheduling:
  - Each process is given some tickets
  - Each process has a stride = big constant K / # of tickets
    - Stride: Time interval a process must wait.
      Inversely proportional to # of tickets

Stride scheduling

- Deterministic version of lottery scheduling
  - Randomness does not guarantee fairness
- Stride scheduling:
  - Each process is given some tickets
  - Each process has a stride = big constant K / # of tickets
    - Stride: Time interval a process must wait.
      Inversely proportional to # of tickets
  - Each time a process runs, its pass += stride

Stride scheduling

- Deterministic version of lottery scheduling
  - Randomness does not guarantee fairness
- Stride scheduling:
  - Each process is given some tickets
  - Each process has a stride = big constant K / # of tickets
    - Stride: Time interval a process must wait.
      Inversely proportional to # of tickets
  - Each time a process runs, its pass += stride
  - Scheduler chooses process with the lowest pass to run next
    - Pass keeps track of next selection time

Stride scheduling

- Deterministic version of lottery scheduling
  - Randomness does not guarantee fairness
- Stride scheduling:
  - Each process is given some tickets
  - Each process has a stride = big constant K / # of tickets
    - Stride: Time interval a process must wait.
      Inversely proportional to # of tickets
  - Each time a process runs, its pass += stride
  - Scheduler chooses process with the lowest pass to run next
    - Pass keeps track of next selection time
  - Can use compensation tickets

Example

Process	Arrival Time	Tickets	Stride (K = 10000)
P1	0	100	100
P2	0	50	200
P3	0	250	40

⋮

Example

Process	Arrival Time	Tickets	Stride (K = 10000)
P1	0	100	100
P2	0	50	200
P3	0	250	40

pass is initialized to stride value

	P1 pass	P2 pass	P3 pass
init	100	200	40

Time slices

⋮

Example

Process	Arrival Time	Tickets	Stride (K = 10000)
P1	0	100	100
P2	0	50	200
P3	0	250	40

pass is initialized to stride value

Time slices

	P1 pass	P2 pass	P3 pass
init	100	200	40
t1: P3	100	200	80

⋮

Example

Process	Arrival Time	Tickets	Stride (K = 10000)
P1	0	100	100
P2	0	50	200
P3	0	250	40

pass is initialized to stride value

Time slices

	P1 pass	P2 pass	P3 pass
init	100	200	40
t1: P3	100	200	80
t2: P3	100	200	120

⋮

Example

Process	Arrival Time	Tickets	Stride (K = 10000)
P1	0	100	100
P2	0	50	200
P3	0	250	40

pass is initialized to stride value

Time slices

	P1 pass	P2 pass	P3 pass
init	100	200	40
t1: P3	100	200	80
t2: P3	100	200	120
t3: P1	200	200	120

⋮

Example

Process	Arrival Time	Tickets	Stride (K = 10000)
P1	0	100	100
P2	0	50	200
P3	0	250	40

pass is initialized to stride value

Time slices

	P1 pass	P2 pass	P3 pass
init	100	200	40
t1: P3	100	200	80
t2: P3	100	200	120
t3: P1	200	200	120
t4: P3	200	200	160
t5: P3	200	200	200

⋮

Example

Process	Arrival Time	Tickets	Stride (K = 10000)
P1	0	100	100
P2	0	50	200
P3	0	250	40

pass is initialized to stride value

Time slices

	P1 pass	P2 pass	P3 pass
init	100	200	40
t1: P3	100	200	80
t2: P3	100	200	120
t3: P1	200	200	120
t4: P3	200	200	160
t5: P3	200	200	200
t6: P1	300	200	200
t7: P2	300	400	200
t8: P3	300	400	240

⋮

- **P1 ran 2 of 8 slices – 25%**
- **P2 ran 1 of 8 slices – 12.5%**
- **P3 ran 5 of 8 slices – 62.5%**

Example

Process	Arrival Time	Tickets	Stride (K = 10000)
P1	0	100	100
P2	0	50	200
P3	0	250	40

- **P1: 100 of 400 tickets – 25%**
- **P2: 50 of 400 tickets – 12.5%**
- **P3: 250 of 400 tickets – 62.5%**

pass is initialized to stride value

Time slices

	P1 pass	P2 pass	P3 pass
init	100	200	40
t1: P3	100	200	80
t2: P3	100	200	120
t3: P1	200	200	120
t4: P3	200	200	160
t5: P3	200	200	200
t6: P1	300	200	200
t7: P2	300	400	200
t8: P3	300	400	240

⋮

- **P1 ran 2 of 8 slices – 25%**
- **P2 ran 1 of 8 slices – 12.5%**
- **P3 ran 5 of 8 slices – 62.5%**

Stride Scheduling – Basic Algorithm

Client Variables:

- Tickets
  - Relative resource allocation
- Strides = $stride_1/tickets$
  - Interval between selection
- Pass $+=$ $stride$
  - Virtual index of next selection (initially, pass = stride)

$stride_1$: a large number

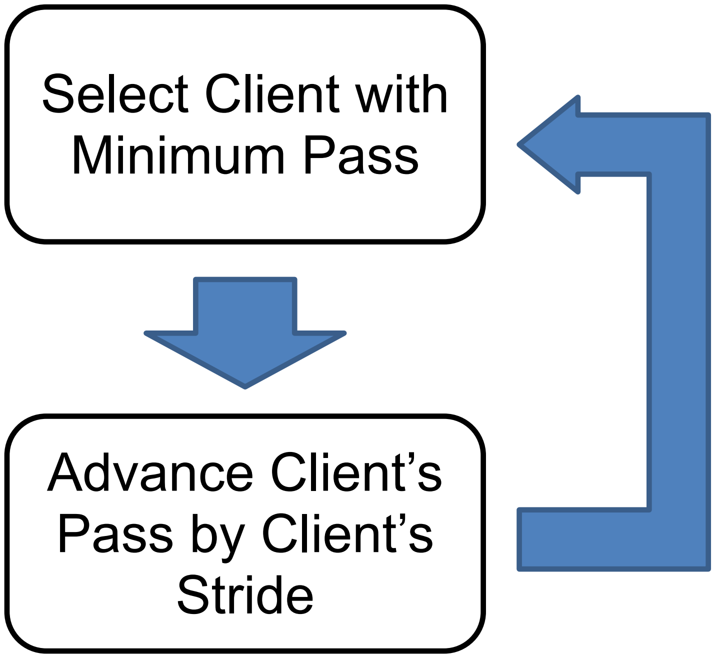
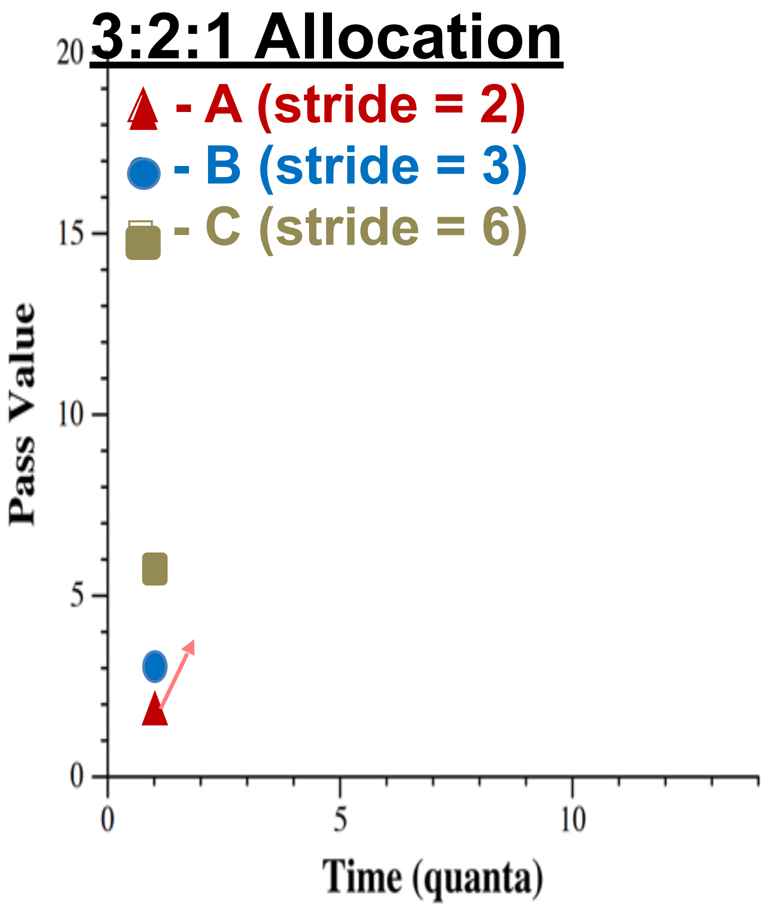


Fig 1 of the paper; Slide and example from Dong-hyeon Park

Stride Scheduling – Basic Algorithm

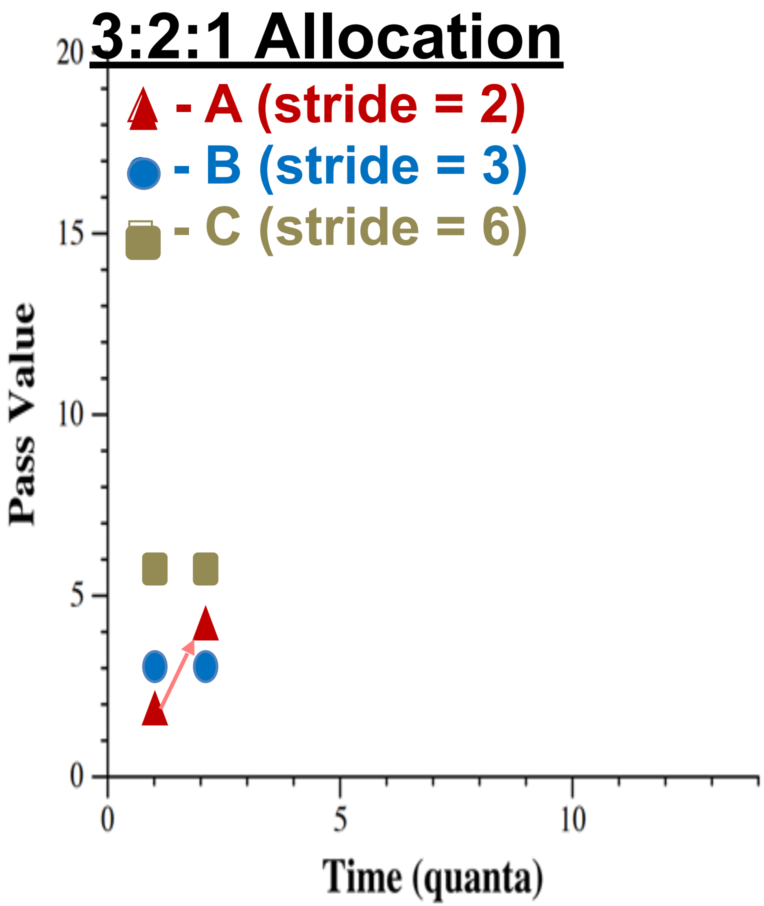


(Fig 2 of the paper)

Pass value

	▲	●	■
Time 1:	2	3	6

Stride Scheduling – Basic Algorithm

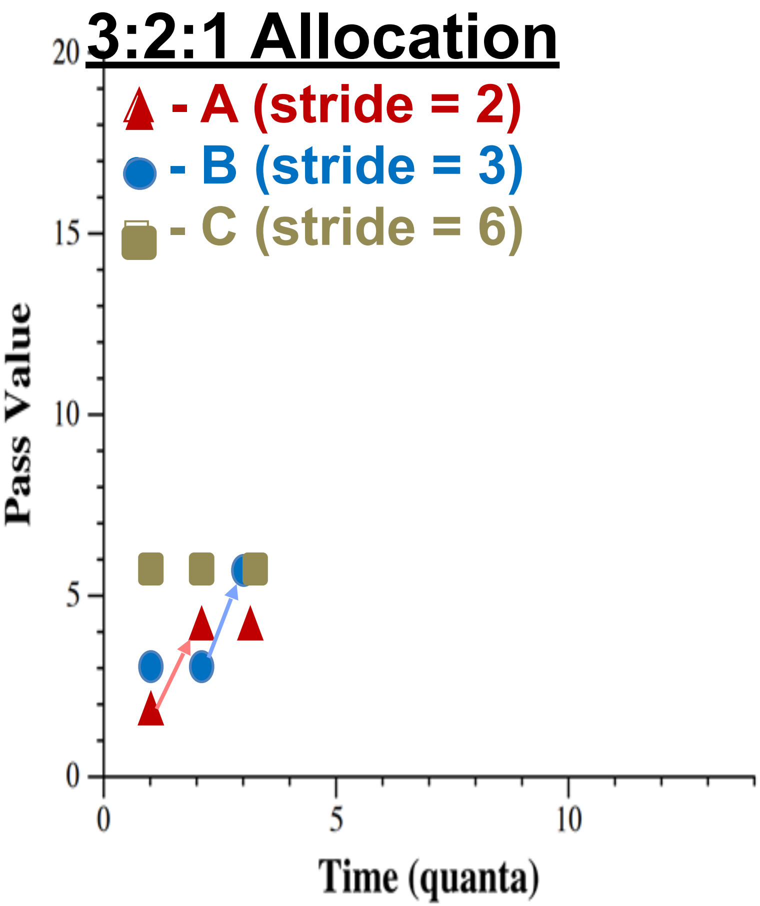


(Fig 2 of the paper)

Pass value

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

Stride Scheduling – Basic Algorithm

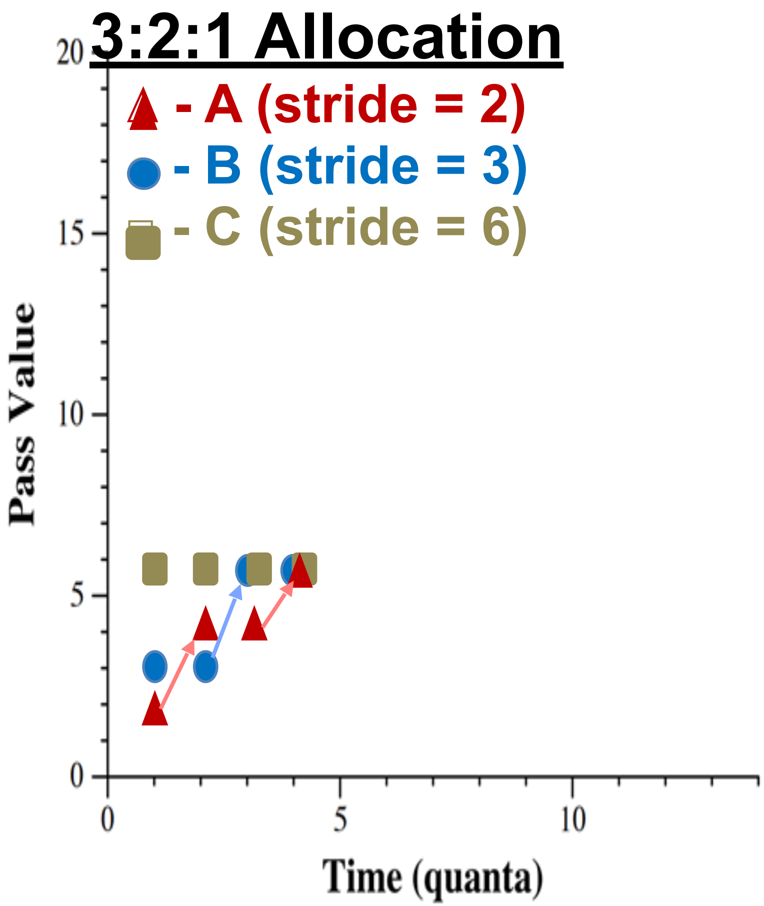


(Fig 2 of the paper)

Pass value

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

Stride Scheduling – Basic Algorithm



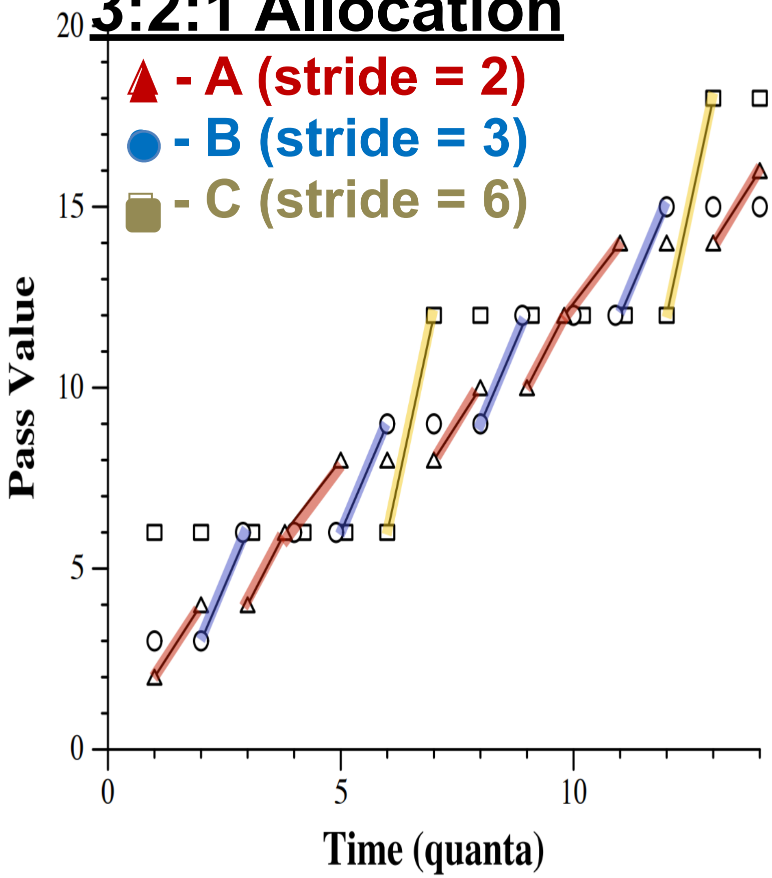
(Fig 2 of the paper)

Pass value

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

3:2:1 Allocation



(Fig 2 of the paper)

Pass value

Time 1: 2 3 6

+2

Time 2: 4 3 6

+3

Time 3: 4 6 6

+2

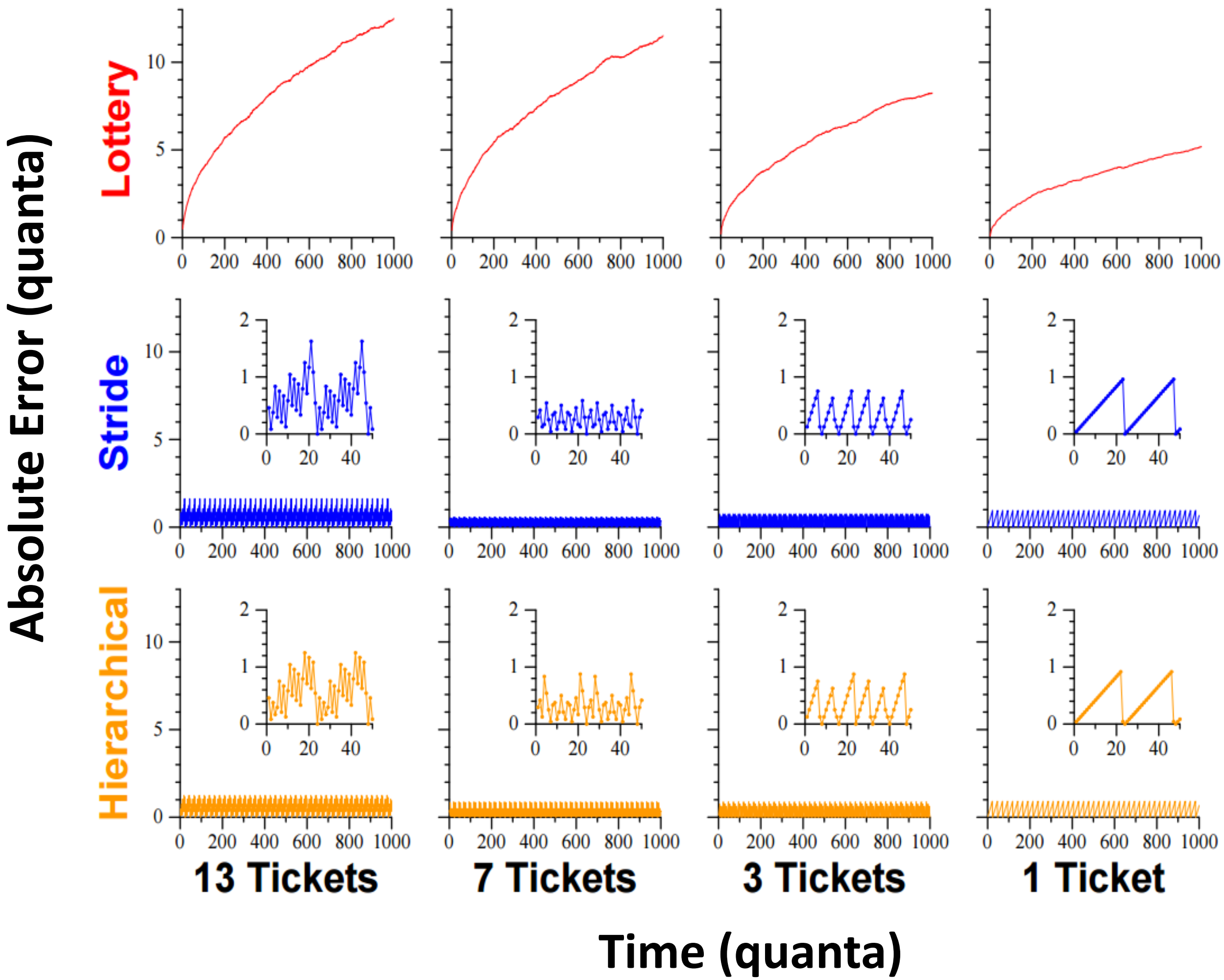
Time 4: 6 6 6

⋮

How Stride scheduling achieves deterministic fairness

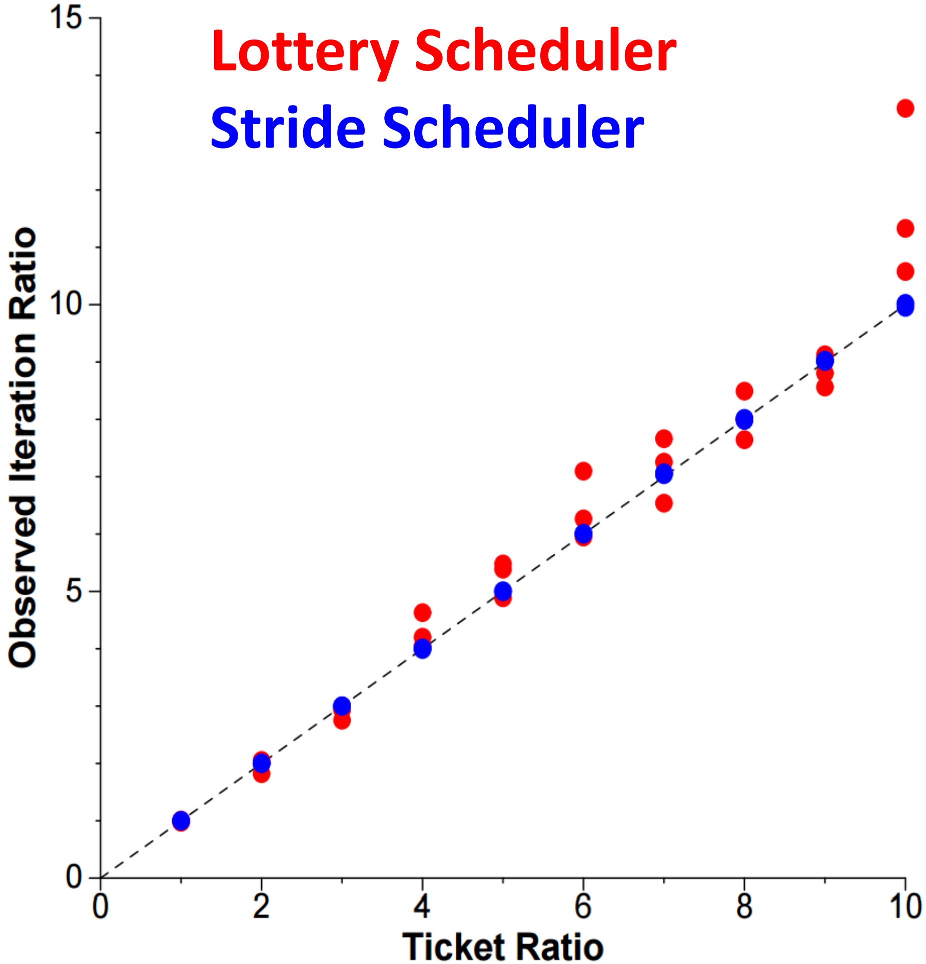
- Each process is given a fair share of the CPU time based on its stride value
- Processes with smaller strides will receive CPU time before processes with larger strides
- This deterministic nature of stride scheduling ensures that processes are treated fairly, as their entitlements are strictly defined by their stride values

Throughput Error Comparison



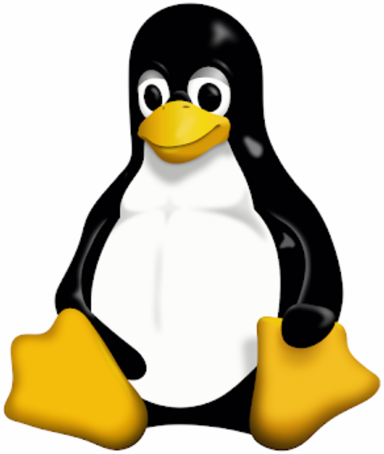
- Error is independent of the allocation time in stride scheduling
- Hierarchical stride scheduling has more balance distribution of error between clients

Accuracy of Prototype Implementation



- Lottery and Stride Scheduler implemented in a real system
- Stride scheduler stayed within 1% of ideal ratio
- Low system overhead relative to the standard Linux scheduler at that time

Linux Completely Fair Scheduler (CFS)

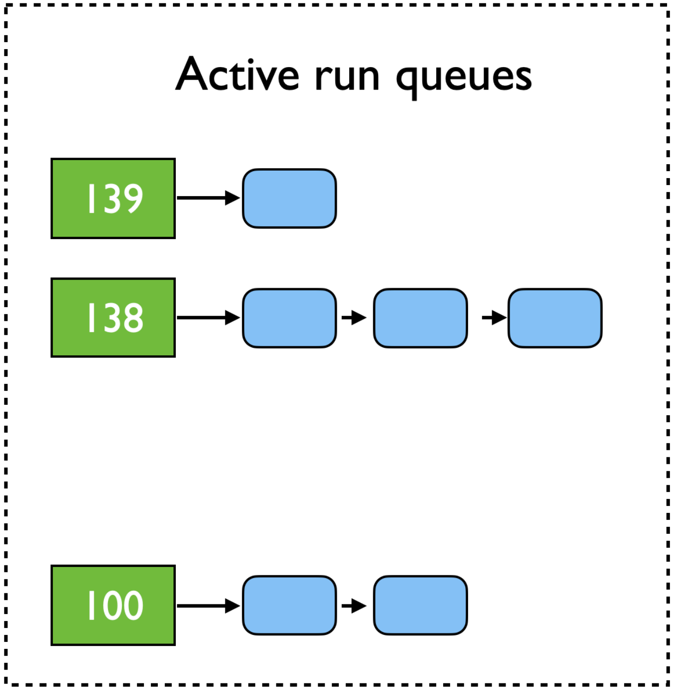


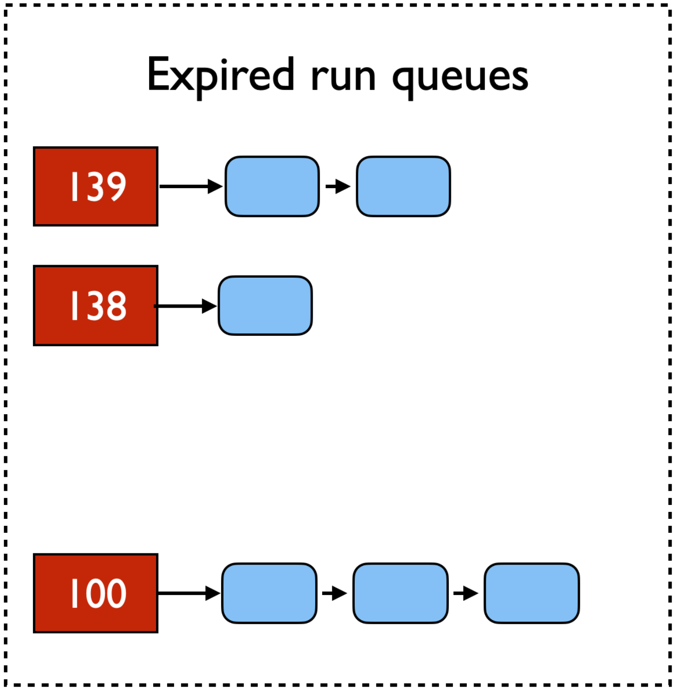
Linux Schedulers (for normal processes)

- O(n): Linux 2.4 – 2.6
  - Scan the runnable queue and select the best process to run
  - Used a global run-queue in multiprocessor systems
  - O(n) complexity…not scalable!
- O(1): Linux 2.6 – 2.6.22
- CFS: Linux 2.6.23 - present

Linux O(1)

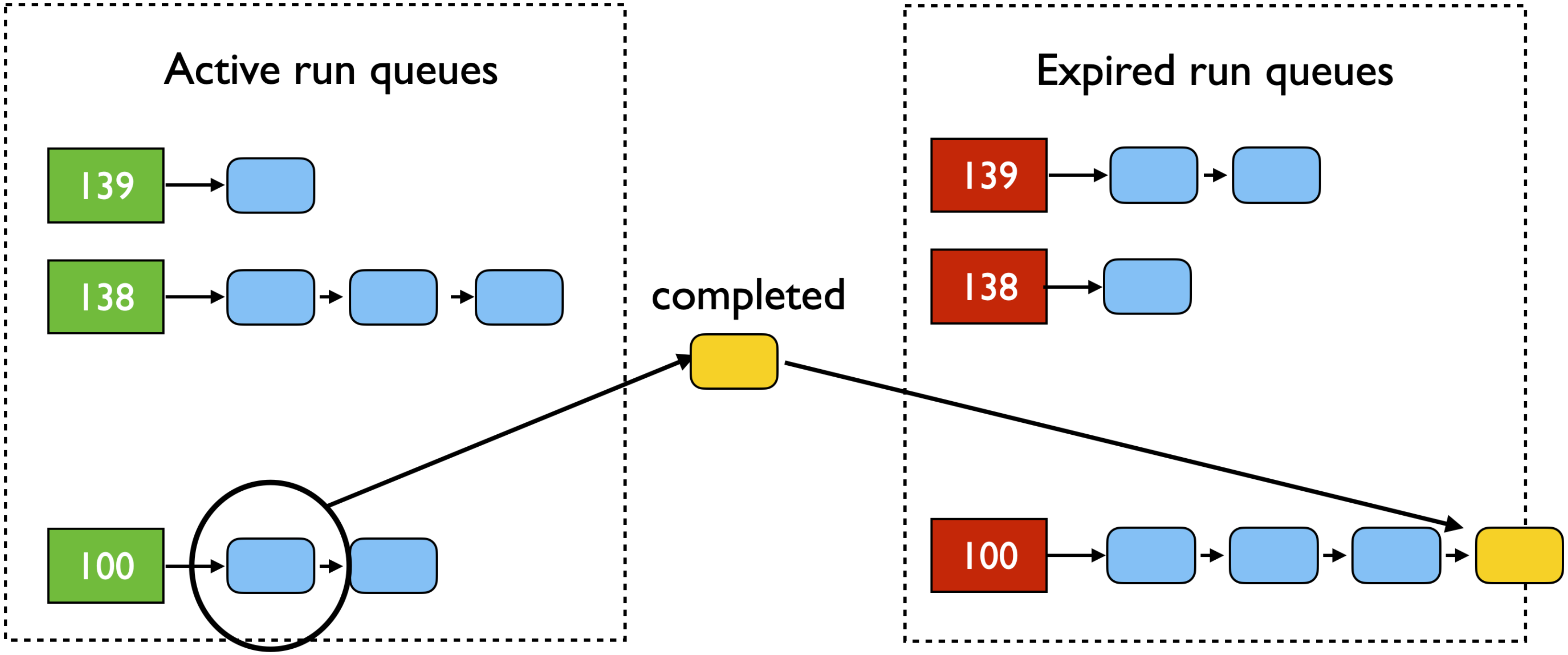
- **Two run queues maintained within each CPU**
  - Each has 40 priority classes (100 - 139) for normal processes
- Pick the first task from the lowest numbered run queue
  - When completed, put this task in the appropriate queue in the expired run queue





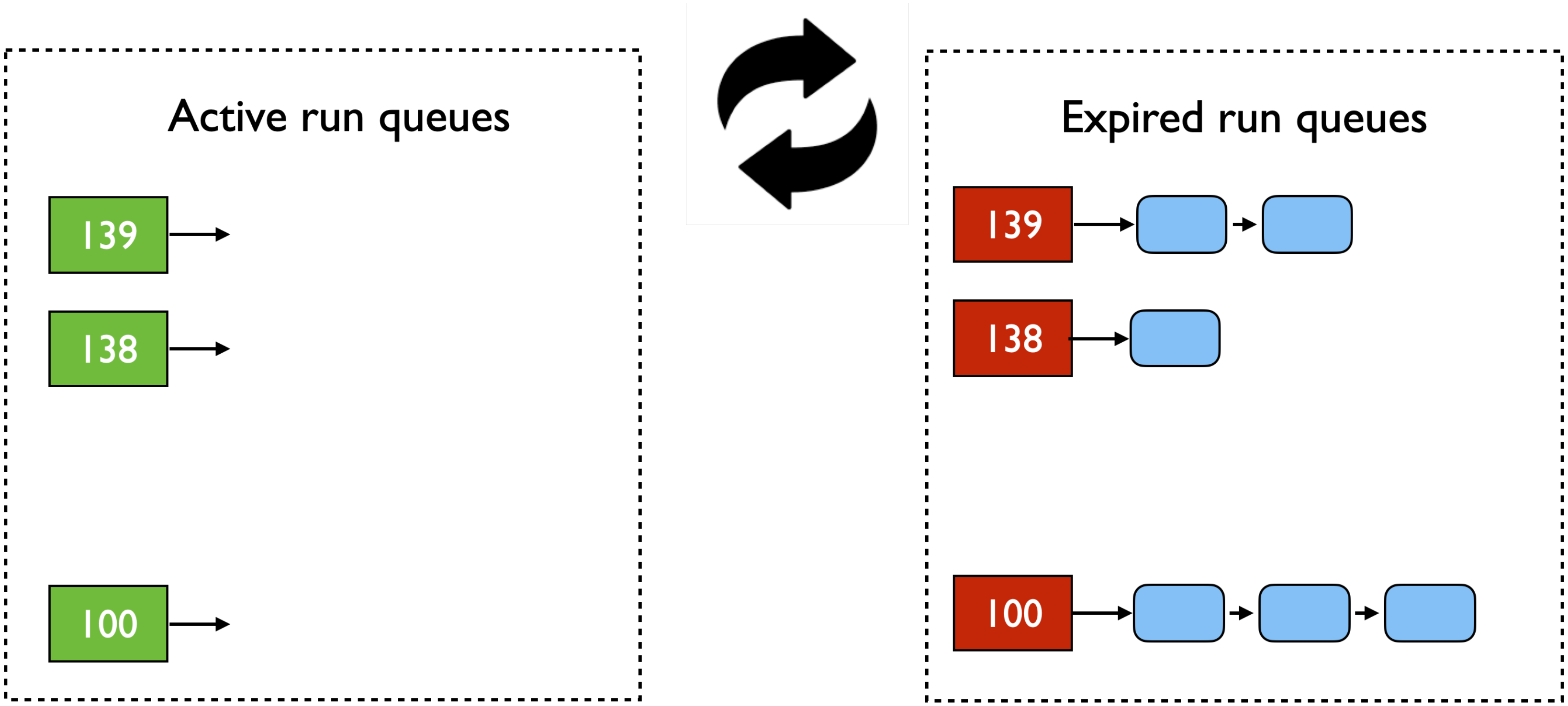
Linux O(1)

- Two run queues maintained within each CPU
  - Each has 40 priority classes (100 - 139) for normal processes
- **Pick the first task from the lowest numbered run queue**
  - When completed, put this task in the appropriate queue in the expired run queue



Linux O(1)

- **Once active run queues are complete**
  - Swapping the two queues (i.e., make the old expired run queues active, and the old active run queues expired)



Linux O(1) properties

- **Prevent starvation**
  - Swapping the active and expired queues
- **Constant time complexity**
  - 1. Find the lowest numbered queue with at least 1 process
    - Obviously not, but
    - Store bitmap of run queues with non-zero entries, and use special instruction “find-first-bit-set’ (e.g., bsfl on intel)
  - 2. Choose the first task from that queue
    - Obviously constant time
- **Able to set dynamic priorities to distinguish between batch and interactive processes**

Linux Completely Fair Scheduler (CFS)

- Key motivation: design a scheduler which gives every process a fair share of resources in a simple and highly efficient way
- Key ideas
  - Time slices are not fixed; diff. process have diff. time slices

Linux Completely Fair Scheduler (CFS)

- Key motivation: design a scheduler which gives every process a fair share of resources in a simple and highly efficient way
- Key ideas
  - Time slices are not fixed; diff. process have diff. time slices
  - Keeps track of process's virtual runtime
    - This is priority-weighted run \time
    - Virtualruntime_currentprocess+= t*weight where t*weight denotes the weighted time for which the process executed recently
    - Achieves "proportional" fair share

Virtual time increases faster for lower priority processes

Linux Completely Fair Scheduler (CFS)

- Key motivation: design a scheduler which gives every process a fair share of resources in a simple and highly efficient way
- Key ideas
  - Time slices are not fixed; diff. process have diff. time slices
  - Keeps track of process's virtual runtime
    - This is priority-weighted run \time
    - Virtualruntime_currentprocess+= t*weight where t*weight denotes the weighted time for which the process executed recently
    - Achieves "proportional" fair share
  - Picks the one with lowest virtual runtime
    - "Repair" illusion of complete fairness
    - Using ~~real~~ red-black trees (simple: the left most node has the lowest virtual runtime)

Linux CFS: Time Slice

- Constraint 1: *Target Latency*
  - Period of time over which every process runs at least once
  - Preserves response time

- Target Latency: 20ms, 4 Processes
  - Each process gets 5ms time slice
- Target Latency: 20 ms, 200 Processes
  - Each process gets 0.1ms time slice (!!!)
  - Recall Round-Robin: Huge context switching overhead

Linux CFS: Time Slice

- Constraint 1: *Target Latency*
  - Period of time over which every process runs at least once
  - Preserves response time

> - Target Latency: 20ms, 4 Processes
>   - Each process gets 5ms time slice
> - Target Latency: 20 ms, 200 Processes
>   - Each process gets 0.1ms time slice (!!!)
>   - Recall Round-Robin: Huge context switching overhead

- Constraint 2: *Minimum Granularity*
  - Minimum length of any time slice
  - Protects throughput

> - Target Latency 20ms, Minimum Granularity 1ms, 200 processes
>   - Each process gets 1ms time slice

Linux CFS: Proportional Share

- Key Idea: Assign a weight w_i to each process i

Originally (equal share): $Q = \text{Target Latency} \ * \frac{1}{N}$

Now (weighted share): $Q_i = (w_i / \sum_p w_p) * \text{Target Latency}$

Linux CFS: Proportional Share

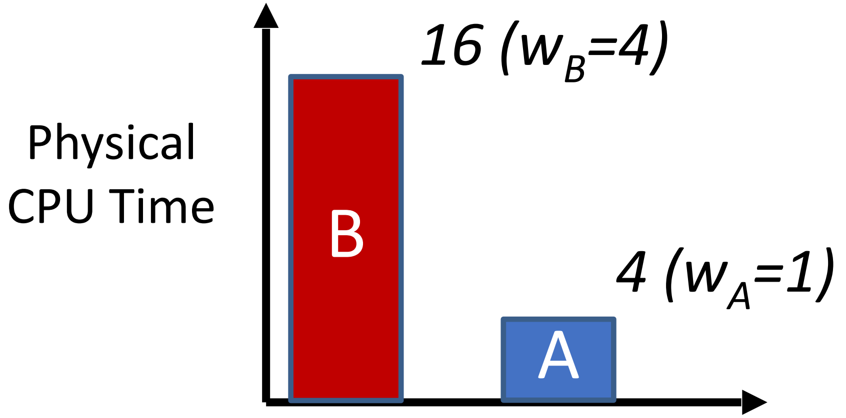
- Target Latency = 20ms,
- Minimum Granularity = 1ms
- Two CPU-Bound Threads
  - Thread *A* has weight 1
  - Thread *B* has weight 4
- Time slice for *A*? 4 ms
- Time slice for *B*? 16 ms

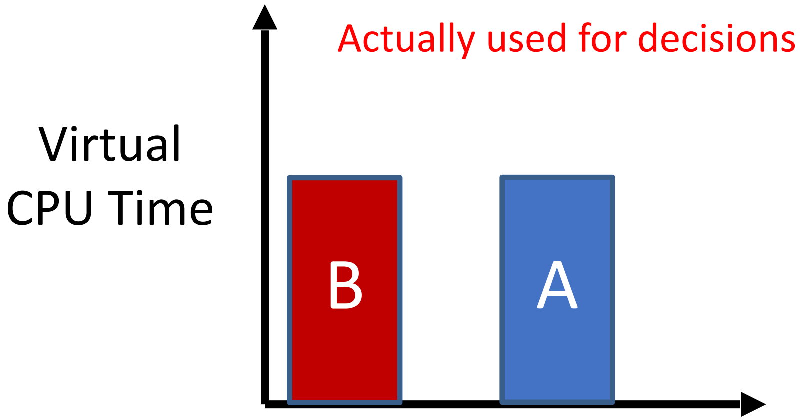
Linux CFS: Proportional Share

- In Linux, normal (non-real-time) tasks use *nice* value for priority
  - Lower nice value → Higher priority
  - Nice values range from 20 to +19
- The weight calculation in CFS is a simple inverse relationship with the priority value
  - A nice value is mapped to a weight value (e.g., 20 is mapped to 88761, 0 is mapped to 1024)
  - Every process that changes nice value up by one level gets 10% less CPU power; Every process that changes nice value down by one gets 10% more CPU power

Linux CFS: Proportional Share

- Track a thread's *virtual* runtime rather than its true physical runtime
  - Higher weight: Virtual runtime increases more slowly
  - Lower weight: Virtual runtime increases more quickly
- Scheduler's decisions are based on Virtual Runtime
  - Red-Black tree to sort processes by virtual runtime → O(lg n)





Choosing the Right Scheduler

I Care About:	Then Choose:
CPU Throughput	FCFS
Avg. Response Time	SRTF (preemptive SJF)
I/O Throughput	SRTF (preemptive SJF)
Fairness (Wait Time to Get CPU)	Round Robin
Fairness (Proportional)	Lottery, Stride, Linux CFS
Meeting Deadlines	EDF
Favoring Important Tasks	Priority

CFS v.s. Stride scheduler

- Simplicity & scheduling overhead
  - Stride performs calculations based on assigned strides and pass values
  - CFS maintains a red-black tree
- Latency
  - Stride may allow tasks with smaller stride to dominate CPU
  - CFS's v_runtime ensures fairness
- Dynamic priority adjustment
  - Once a task's stride is set, it remains constant
  - CFS dynamically adjusts priority based on CPU usage
- Responsiveness
  - Immediate response needs smaller stride which could intensively consume CPU
  - CFS develops time slicing and target latency techniques

Scheduler Activations

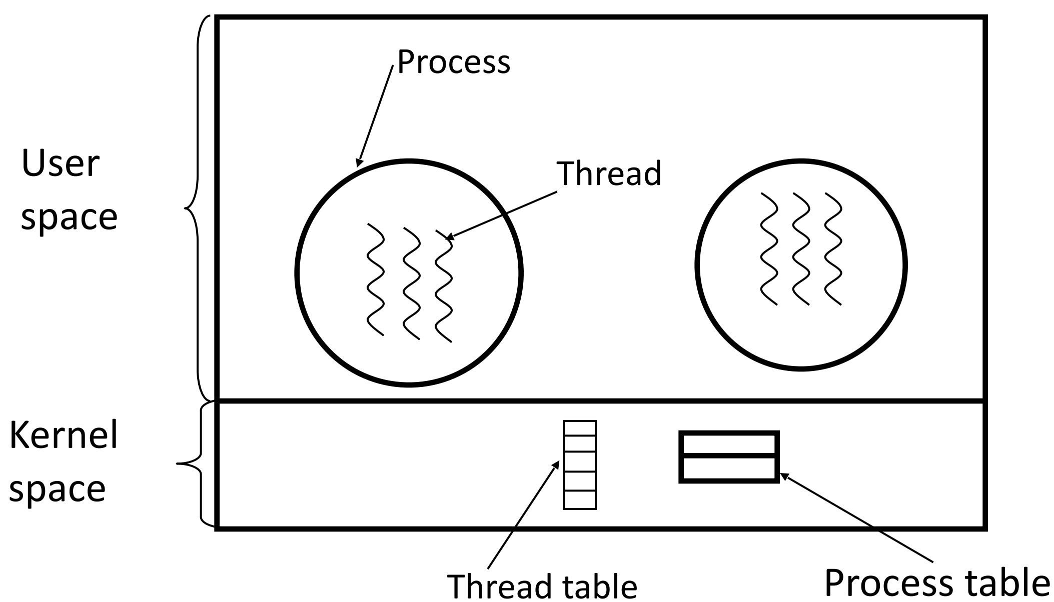
Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism

Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

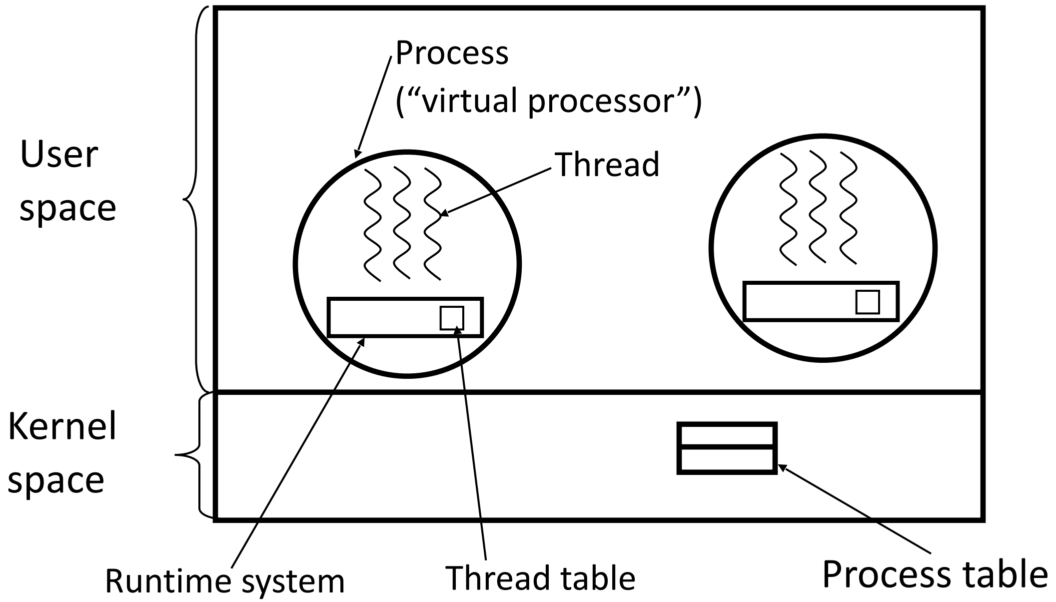
Kernel-level Threads

- Kernel maintains context information of all threads
- No problem with blocking system calls == high functionality
- Switching between threads require the kernel == poor performance
  - Requires mode switching and parameter checking of thread operations
- General-purpose scheduling algorithm in the kernel == lack of flexibility



User-level Threads

- All thread management is done in user space
  - Kernel is not aware of existence of threads
- No kernel intervention == high performance
- Supports customized scheduling algorithms == flexible
- Entire process blocked during system services == lack of functionality
  - E.g., blocking system calls, I/O, page faults, other processes



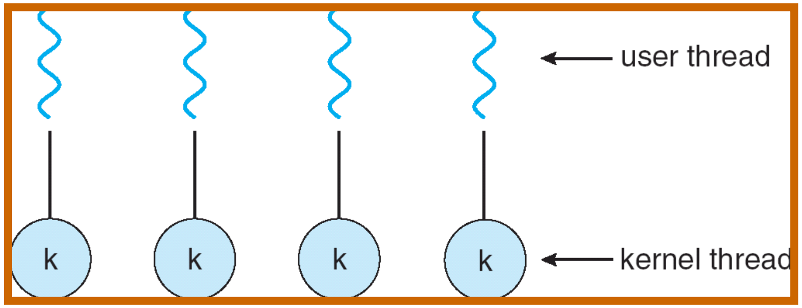
User-level vs. Kernel-level Threads

- Neither user-level threads nor kernel-level threads work ideally
  - User-level threads
    - Cheap
    - Have application information
    - But not visible to kernel
    - System integration problem
  - Kernel-level threads
    - Expensive
    - Lack application information

Threading Models

One-to-One Model

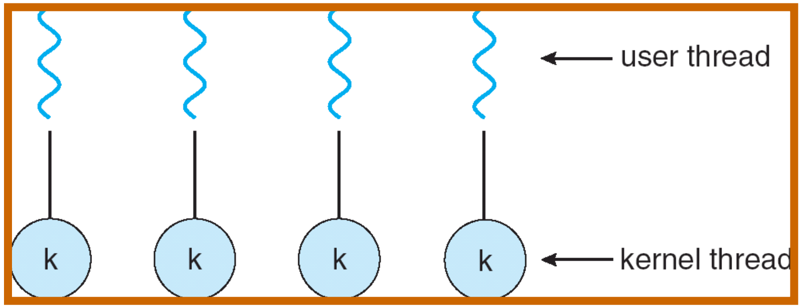
- Almost all current implementations
- Kernel-level threading



Threading Models

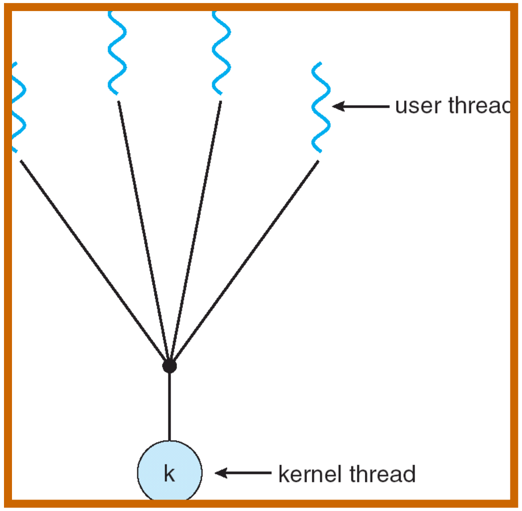
One-to-One Model

- Almost all current implementations
- Kernel-level threading



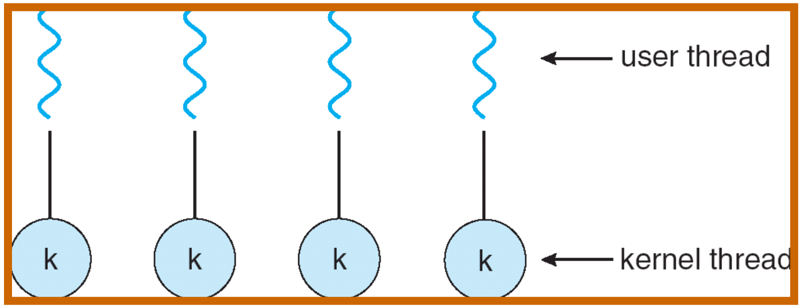
Many-to-One

- User-level threading

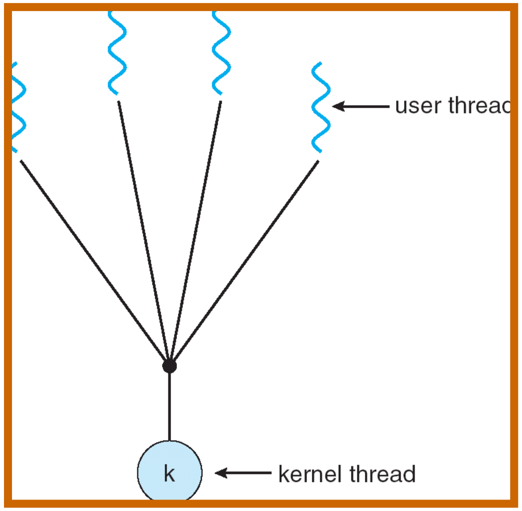


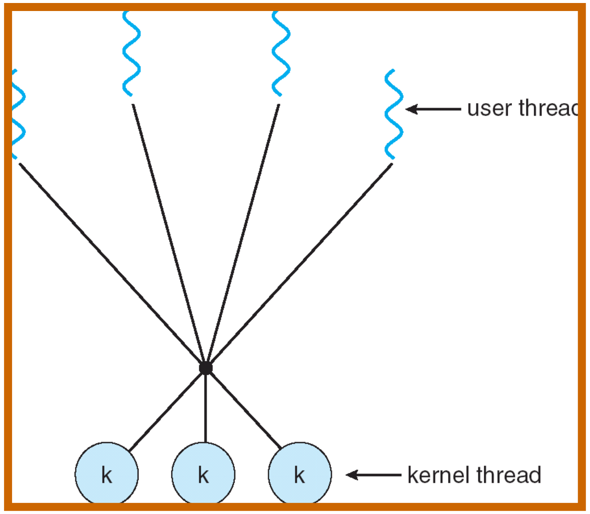
Threading Models

One-to-One Model
- Almost all current implementations
- Kernel-level threading



Many-to-One
- User-level threading





Many-to-Many

Scheduler Activations

- Address the problems of user-level threads
- Allow coordination between user and kernel schedulers
  - Application is free to implement any scheduling policy
  - Kernel notifies user-level scheduler of relevant kernel events (e.g., blocking)
- N:M threading
  - N user-level threads mapped to M kernel-level threads
  - More complex to implement