Max-min Fairness

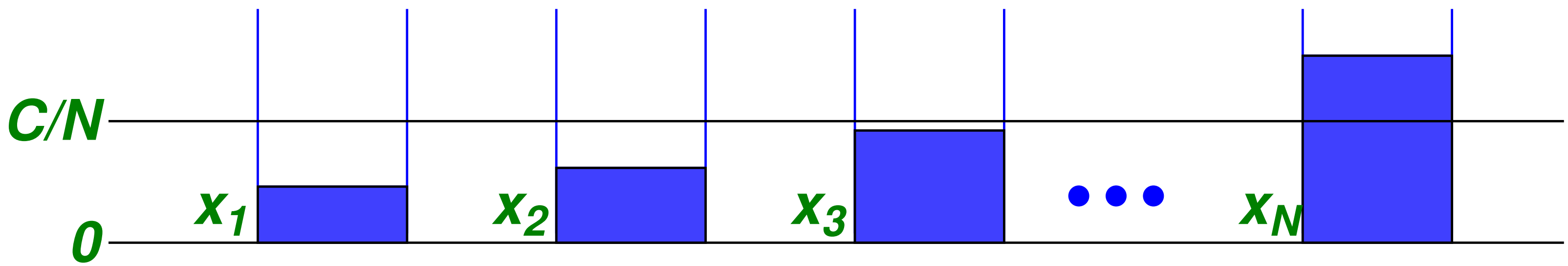
- ***Max-min Fairness:*** **a fair service maximizes the service of the customer receiving the poorest service**

- ***Max-min Fairness*** **criterion:**
  1) **no user receives more than its *request***
  2) **no other allocation scheme satisfying condition 1 has a higher minimum allocation**
     - **i.e., *maximize the minimum allocation***
  3) **condition 2 remains recursively true as we remove the minimal user and reduce total resource accordingly**

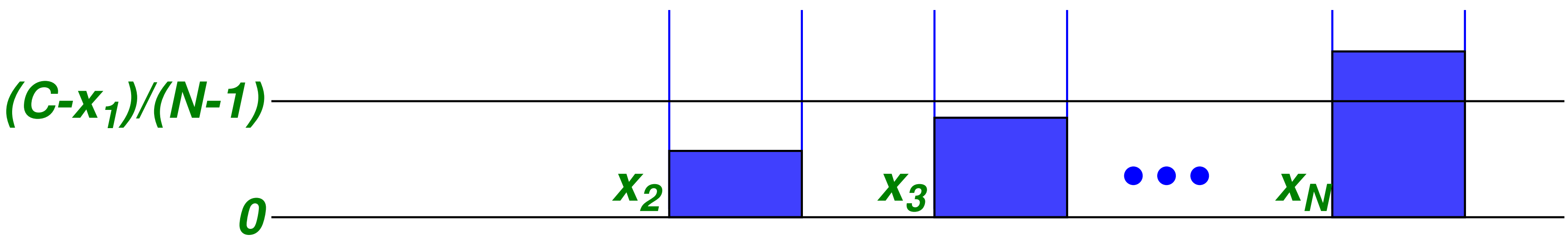
- **Ex: if the government has $1,000,000 to give to 1,000 citizens, what's the fair way to distribute the money?**
  - **is it fair to give everyone $1,000?**
    - **no, you are not allowed to waste resources**

Max-min Fairness Example

- Total capacity C divided among N jobs
  - x_i is the request of job i
  - sort jobs based on x_i
  - initially, assign C/N to each job



  - satisfy x_1, redistribute remaining capacity evenly

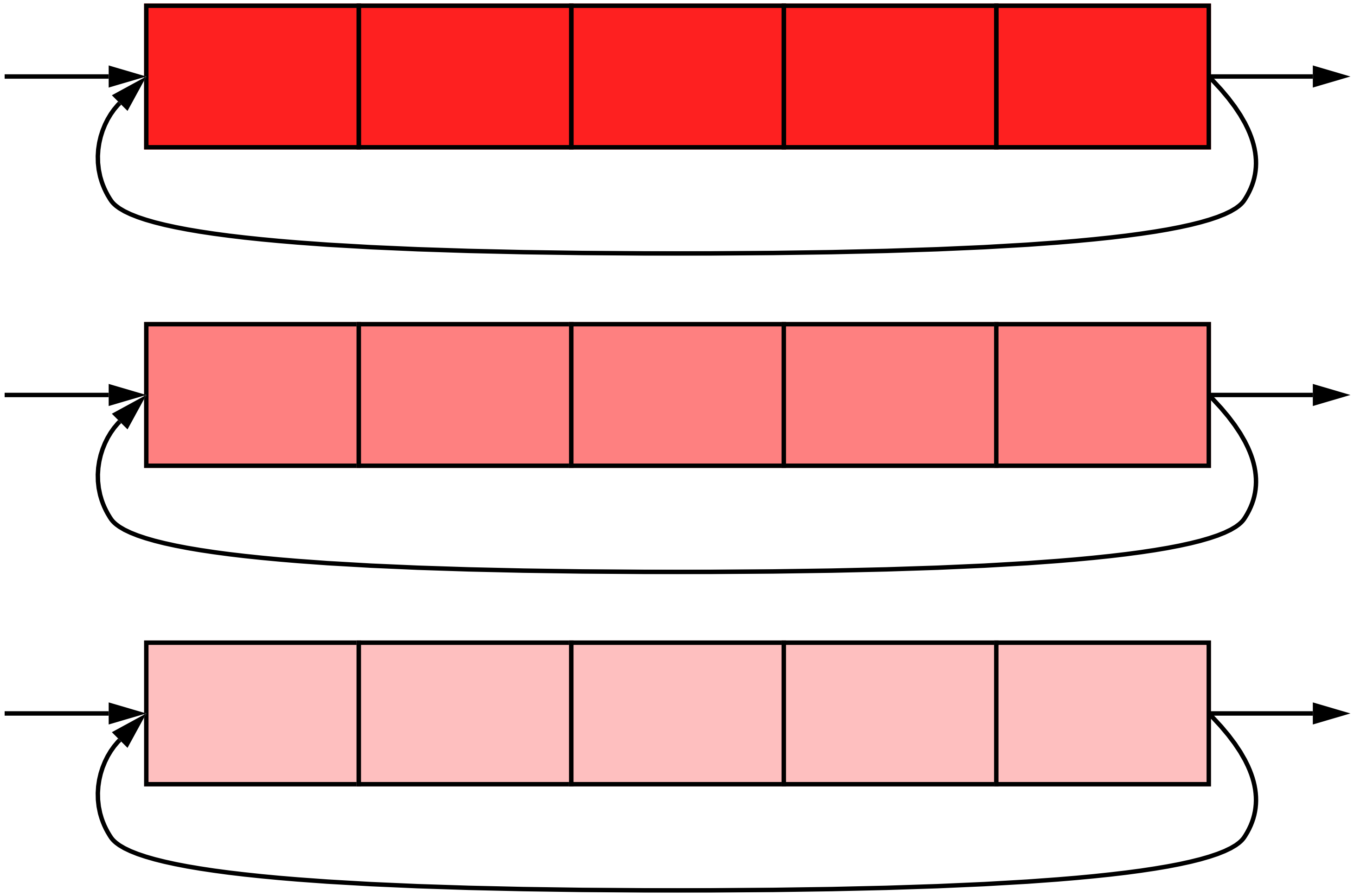


  - recursion
- This is basically *"processor sharing"*

Scheduling for Interactive Systems

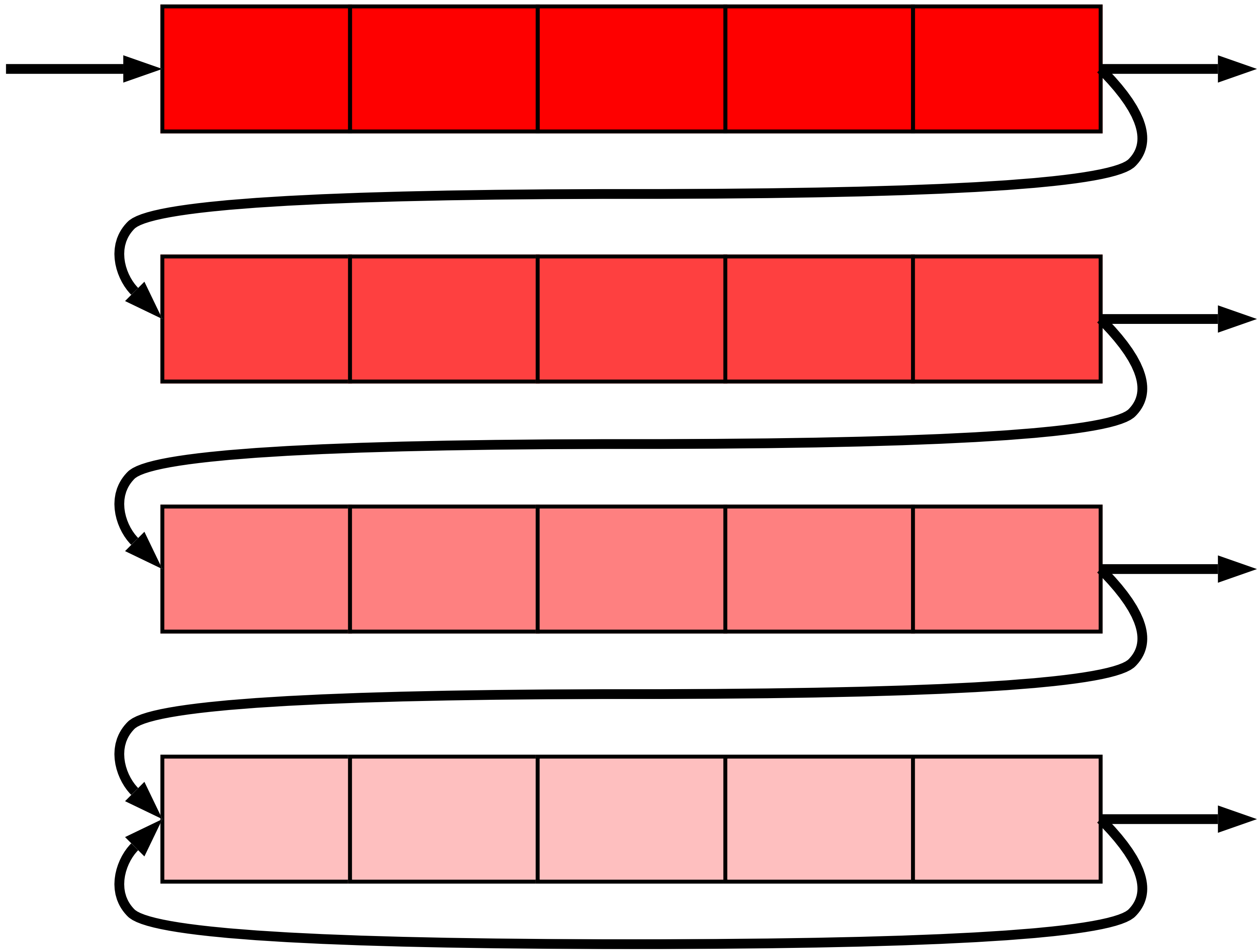
- *Length of "jobs" not known*
  - by the way, the round-robin scheduler works just fine without knowing the length of jobs
- Threads would give up CPU voluntarily
  - they *block* for user input
- Would like to *favor interactive jobs*
  - use *priority queueing*
  - what is an "interactive job"?
    - "interactive user" can run non-interactive job (like warmup2)
    - an "interactive jobs" is a job that requires "user interaction" (with mouse and/or keyboard)
      - e.g., the command shell

Round Robin with Priority



- *how* to determine priority?
  - let the threads themselves decide?

Multi-Level Feedback Queues

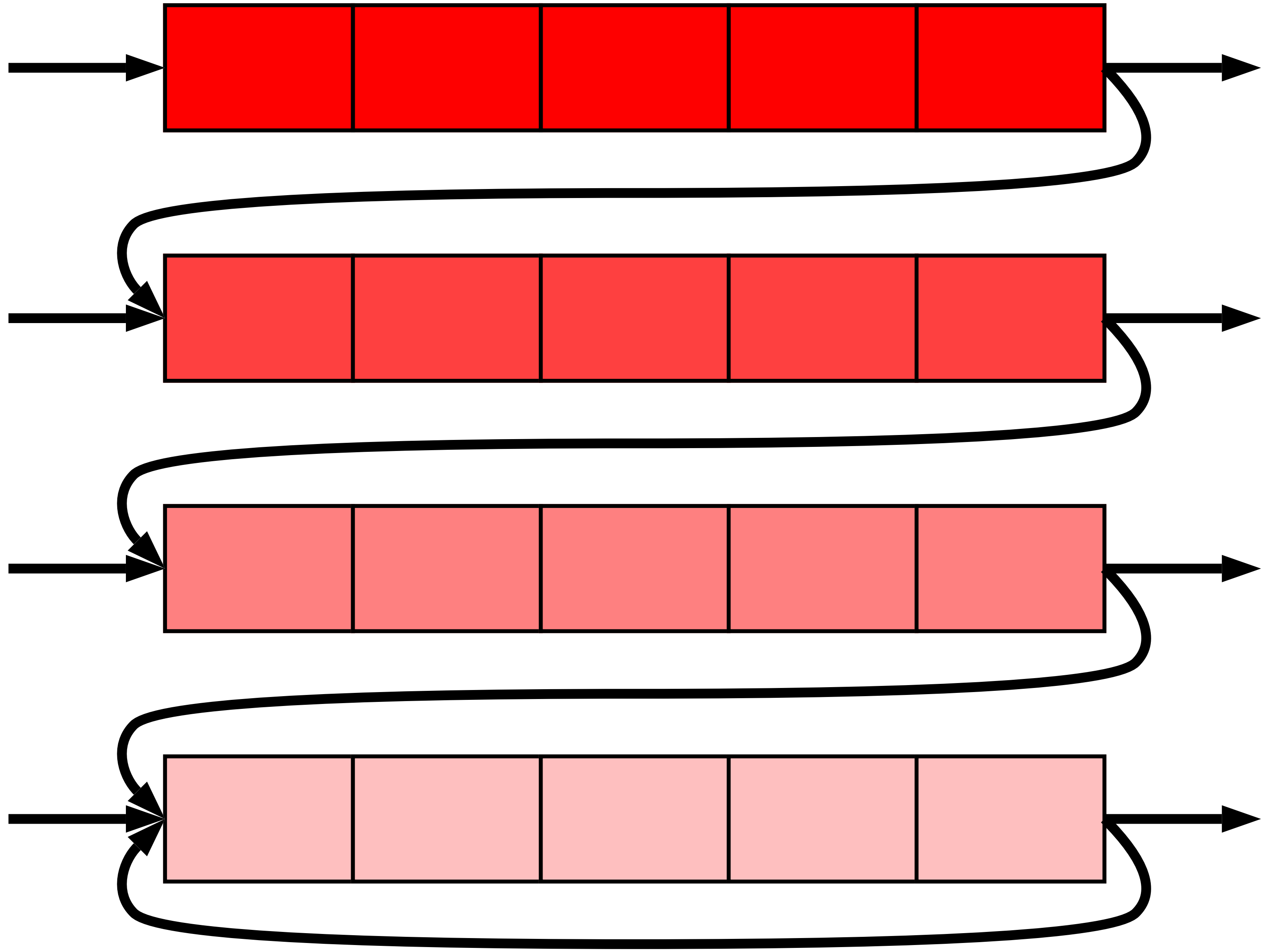


- maybe it makes more sense to have *dynamic priority*

Multi-Level Feedback Queues

- When a thread arrives to the run queue, it gets highest priority
  - observe what it does when it uses the CPU
    - if it *uses a full time slice*
      - *decrease its priority*
    - if it *blocks* before using up a full time slice
      - update information (statistics?) in thread control block
- To avoid starvation, use *aging*
  - if a job hasn't been run for a long time, increase its priority
- Clearly, not a fair scheduling algorithm
- Priority used in this scheme is *dynamic priority*
  - this is *"short term priority"*
  - when we use the word "priority", we typically mean that "priority" would reflect how "important" a job is
    - this is a different kind of priority, but is it the right kind?
    - it may not be right to give everything the user runs high priority

Multi-Level Feedback Queues



- can have variations

Real-Life Example

- Your iPod is broken
  - run mp3 player on your PC
- The baseball playoffs are on
  - streaming video
- An OS assignment is due
  - editor, compiler, debugger
- You've got to do everything on one computer
- Can your scheduler hack it?
- What scheduler is suitable for a *general purpose* system?

```
static int msync_interval(struct vm_area_struct * vma,
   unsigned long start, unsigned long end, int flags)

   int ret = 0;
   struct file * file = vma->vm_file;

   if ((flags & MS_INVALIDATE) && (vma->vm_flags & VM_LOCKED))
      return -EBUSY;

   if (file && (vma->vm_flags & VM_SHARED)) {
      ret = filemap_sync(vma, start, end-start, flags);

      if (!ret && (flags & MS_SYNC)) {
         struct address_space *mapping = file->f_mapping;
         int err;
```

Interactive Scheduling

- ***Time-sliced***, ***priority-based***, ***preemptive***
  - e.g., multi-level feedback queues
    - every time slice, pick the thread with the highest priority to run in the CPU for the next time slice
- Priority depends on expected time to block
  - interactive threads should have high priority
  - compute threads should have low priority
- Other heuristics
  - e.g., determine priority using ***long term history*** (not just immediate history)
    - processor usage causes decrease
    - sleeping causes increase

Scheduling for Fairness

- If *fairness* is really important, what would you do?
- Ex: shared servers
  - you and four friends each contribute $1000 towards a server
    - you, rightfully, feel you own 20% of it
  - your friends are into threads, you're not
    - they run *4*-threaded programs
    - you run a *1*-threaded program
  - their programs each get *4/17* of the processor
  - your programs get *1/17* of the processor

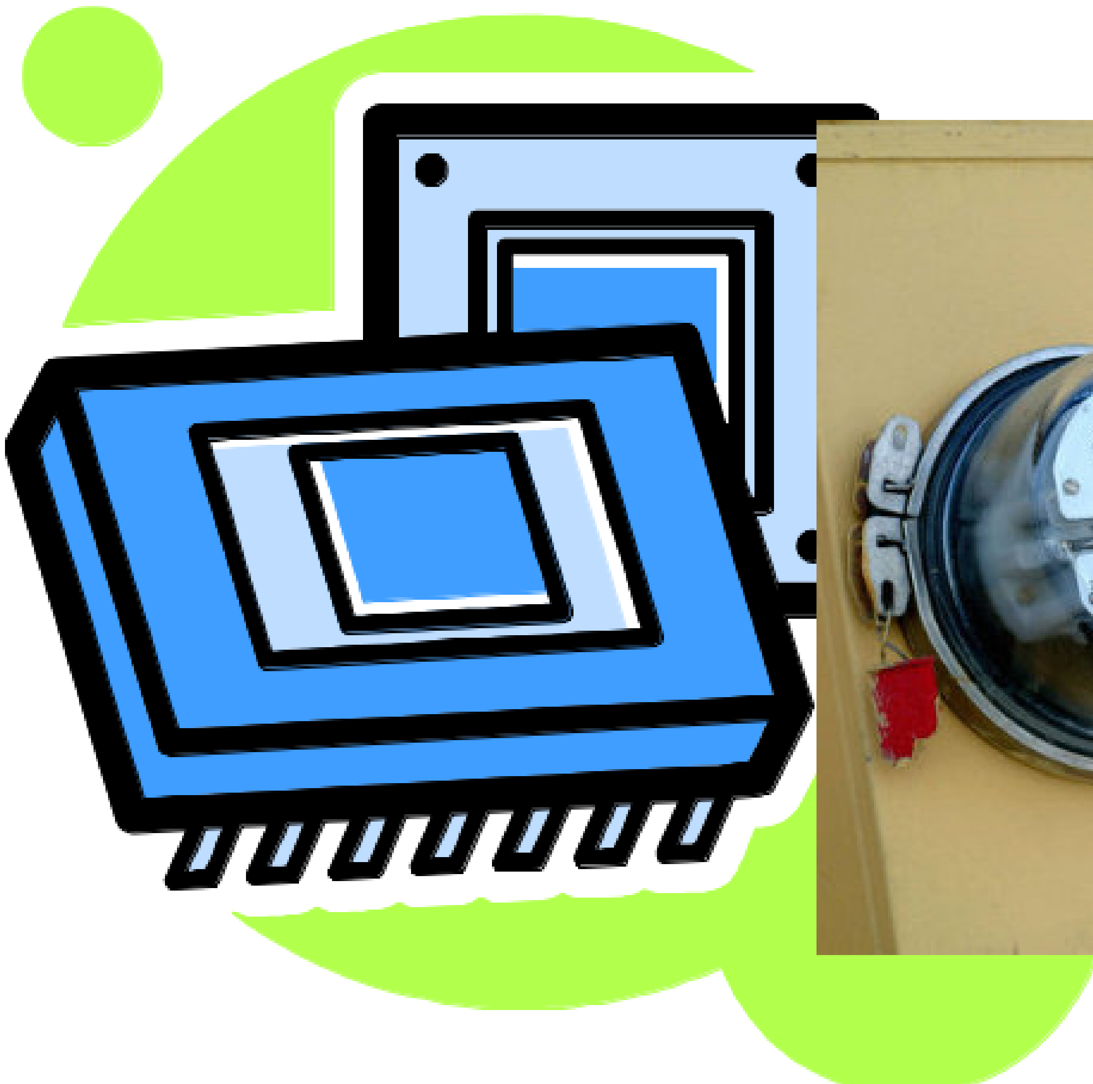
Lottery Scheduling

- *20* lottery tickets are distributed equally to you and your four friends
  - you give *4* tickets to your one thread
  - they give one ticket each to their threads
- A lottery is held for every scheduling decision
  - your thread is *4* times more likely to win than the others
- But how do you implement a fair and efficient lottery scheme in the kernel if there are lots of jobs at the run queue?

Proportional-Share Scheduling

- Stride scheduling
  - 1995 paper by Waldspurger and Weihl
- Completely fair scheduling (CFS)
  - added to Linux in 2007

Metered Processors



- the textbook presented *Stride Scheduling* differently
  - as far as *exam* goes, you must follow *lecture slides*

Stride Scheduling Algorithm

- ***Time-sliced***, ***priority-based***, ***preemptive***
  - every time slice, pick the thread with the ***highest priority*** to run in the CPU for the next time slice
  - every thread is assigned a (dynamic) priority, called a ***pass*** value
    - single queue, ***sorted based on pass values***, ***smallest first***
  - every thread is assigned a ***stride*** value
    - stride values are computed according to ***distribution of tickets*** in a lottery scheduling scheme
      - stride values are ***integers***

- In every iteration / time-slice
  1) shedule the thread with the ***smallest pass value*** (at the head of the queue) and set the ***global pass value*** to be the pass value of this thread
  2) when time-slice is over, increment the thread's ***pass*** value by its ***stride*** value (i.e., pass += stride)
  3) loop

Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets

Thread	Tickets	Stride
A	3	
B	2	
C	1	

Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets

Thread	Tickets	Stride
A	3	1/3
B	2	1/2
C	1	1

then multiply by smallest common multiplier of denominators to get *interger stride widths* (here and for *exams*)

Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets

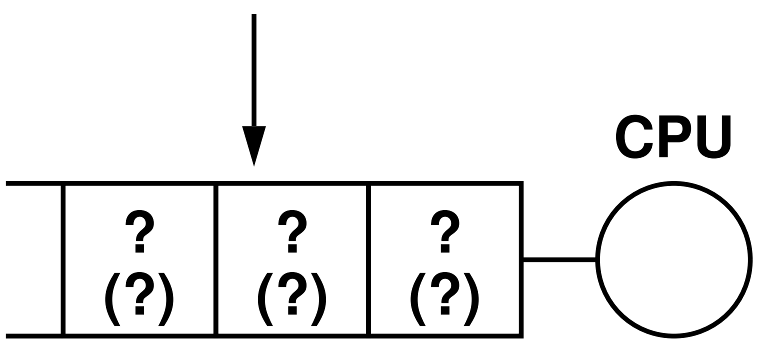
Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets
  - every thread is initialized with a *pass* value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

can start with any pass values (e.g., determined by the *current state* of the stride scheduler)



Stride Scheduling Example

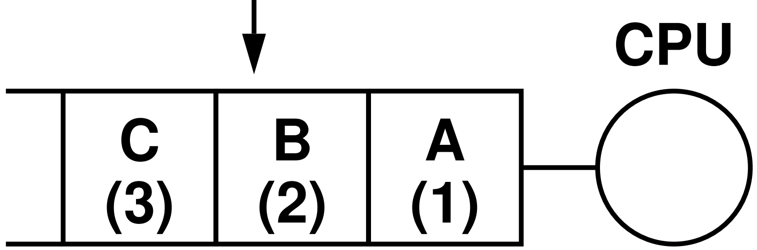
- Stride ∝ 1 / number of tickets
  - every thread is initialized with a ***pass*** value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

keep track of scheduling history

itr	A	B	C
1	1	2	3

can start with any pass values
(this is just an arbitrary example)

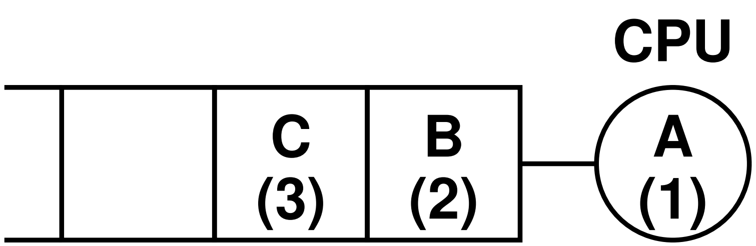


Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets
  - every thread is initialized with a *pass* value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

itr	A	B	C
1	1	2	3

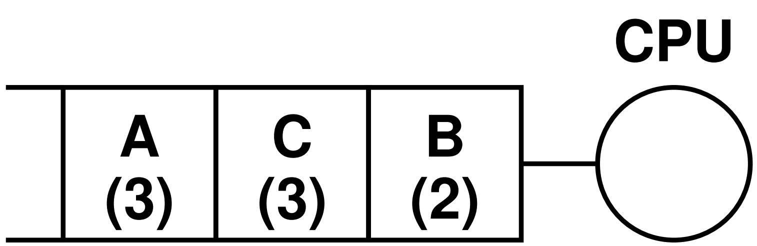


Stride Scheduling Example

- Stride ∝ 1 / number of tickets
  - every thread is initialized with a *pass* value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

itr	A	B	C
1	1	2	3
2	3	2	3

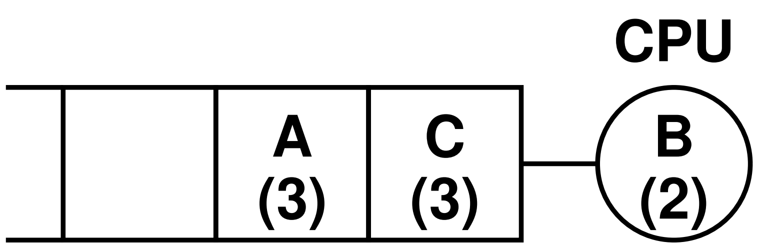


Stride Scheduling Example

- Stride ∝ 1 / number of tickets
  - every thread is initialized with a *pass* value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

itr	A	B	C
1	1	2	3
2	3	2	3

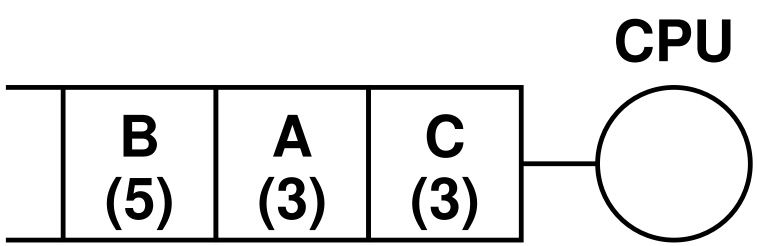


Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets
  - every thread is initialized with a *pass* value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

itr	A	B	C
1	1	2	3
2	3	2	3
3	3	5	3

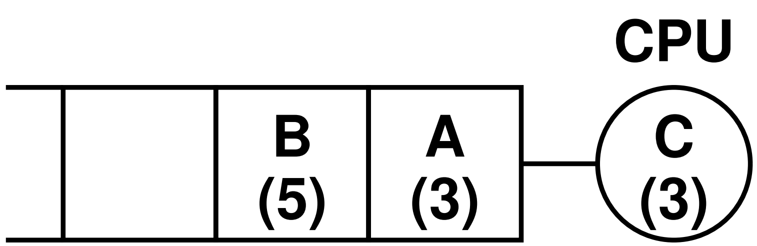


Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets
  - every thread is initialized with a *pass* value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

itr	A	B	C
1	1	2	3
2	3	2	3
3	3	5	3

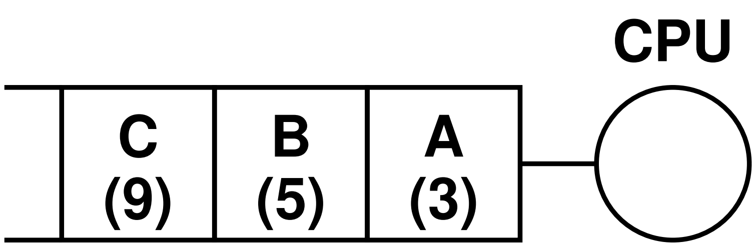


Stride Scheduling Example

- Stride ∝ 1 / number of tickets
  - every thread is initialized with a *pass* value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

itr	A	B	C
1	1	2	3
2	3	2	3
3	3	5	3
4	3	5	9

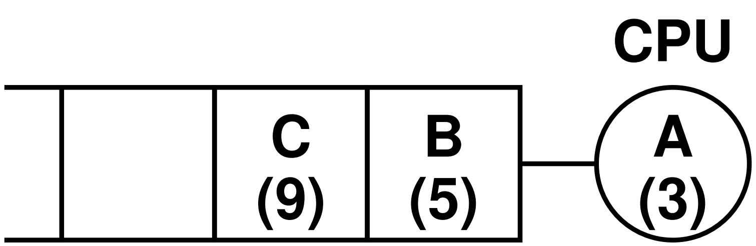


Stride Scheduling Example

- Stride ∝ 1 / number of tickets
  - every thread is initialized with a *pass* value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

itr	A	B	C
1	1	2	3
2	3	2	3
3	3	5	3
4	3	5	9

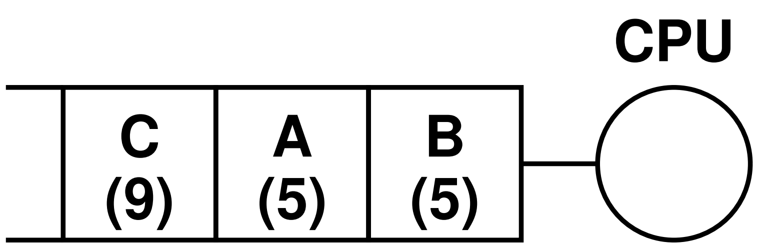


Stride Scheduling Example

- Stride ∝ 1 / number of tickets
  - every thread is initialized with a *pass* value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

itr	A	B	C
1	1	2	3
2	3	2	3
3	3	5	3
4	3	5	9
5	5	5	9

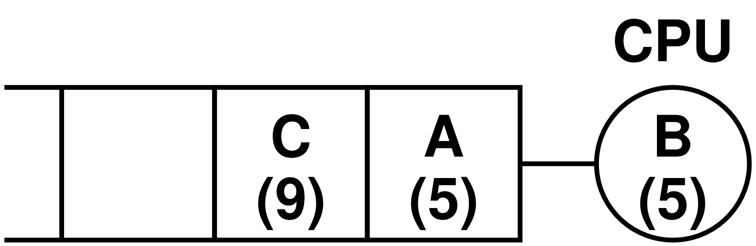


Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets
  - every thread is initialized with a *pass* value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

itr	A	B	C
1	1	2	3
2	3	2	3
3	3	5	3
4	3	5	9
5	5	5	9

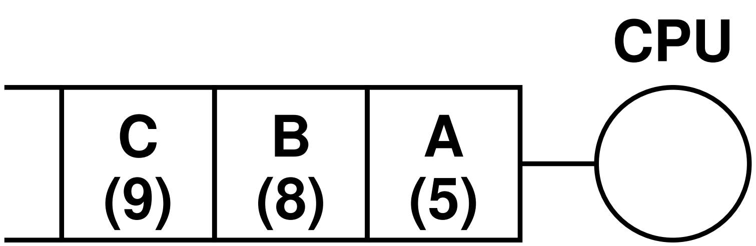


Stride Scheduling Example

- Stride ∝ 1 / number of tickets
  - every thread is initialized with a *pass* value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

itr	A	B	C
1	1	2	3
2	3	2	3
3	3	5	3
4	3	5	9
5	5	5	9
6	5	8	9

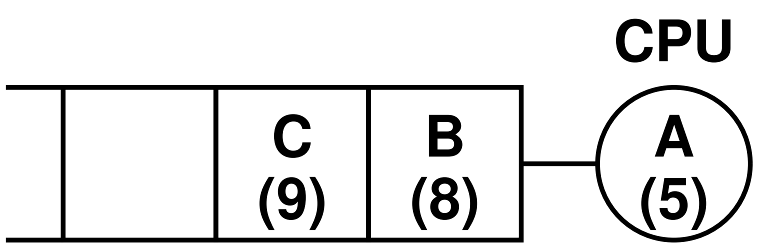


Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets
  - every thread is initialized with a *pass* value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

itr	A	B	C
1	1	2	3
2	3	2	3
3	3	5	3
4	3	5	9
5	5	5	9
6	5	8	9

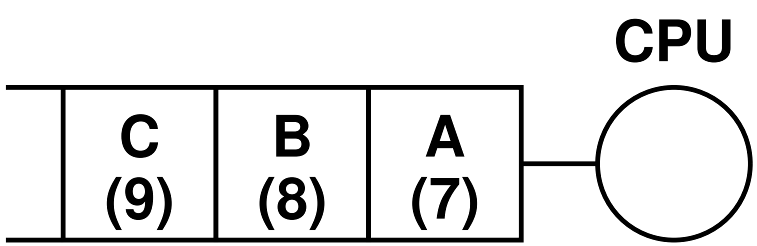


Stride Scheduling Example

- Stride ∝ 1 / number of tickets
  - every thread is initialized with a *pass* value

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6

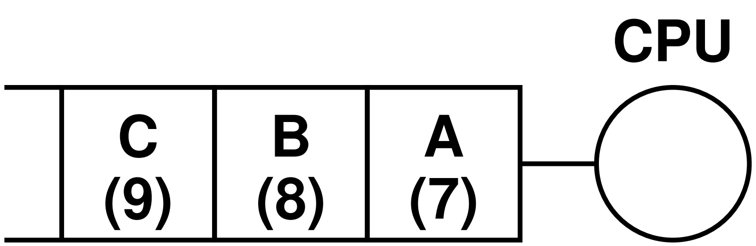
itr	A	B	C
1	1	2	3
2	3	2	3
3	3	5	3
4	3	5	9
5	5	5	9
6	5	8	9
7	7	8	9



Stride Scheduling Example

- Stride $\propto$ 1 / number of tickets
  - every thread is initialized with a *pass* value
- Conforms to the distribution of tickets!

Thread	Tickets	Stride
A	3	2
B	2	3
C	1	6



itr	A	B	C
1	1	2	3
2	3	2	3
3	3	5	3
4	3	5	9
5	5	5	9
6	5	8	9
7	7	8	9

Stride Scheduling - Additional Details

- New thread
  - allocate the *global pass value*
    - so it gets to run first
- Thread uses less than its quantum
  - let $0 < f < 1$ be the fraction of the quantum actually used
  - pass += $f \times$ stride
  - the result is that *interactive threads* get *higher priority*
- Is this better than Multi-level Feedback Queue?
  - unless you can assign dollar amount to threads, you still have the same problem with how to allocate lottery tickets to threads
- Isn't sorting pass values slow?
  - you can use a "heap" (data structure) to implement a priority queue
    - *insertion* and *deleting_min* is $O(\log_2 n)$ where n is the number of jobs at the priority queue