CHAPTER III
SCHEDULING

Jehan-François Pâris
jfparis@uh.edu
Chapter overview

- The problem

- Non-preemptive policies:
  - FCFS, SJF

- Preemptive policies:
  - Round robin, multilevel queues with feedback, guaranteed scheduling
  - Examples: UNIX, Linux, Windows NT and after
The scheduler

- Part of the OS that decides *how to allocate the processor cores and the main memory* to processes

- Will focus here on the **CPU scheduler**
  - Decides which ready process should get a processor core
  - Also called short-term scheduler
Objectives

- A good scheduler should
  - Minimize *user response times* of all interactive processes
    - *Major objective today*
  - Maximize *system throughput*
  - Be *fair*
  - Avoid *starvation*
What is starvation?

- Starvation happens whenever some ready processes never get core time
  - Typical of schedulers using priorities
    - Lowest-priority processes keep getting set aside
- Remedy is to *increase* the priorities of processes that have waited *too long*
Fairness

- Ensuring fairness is more difficult than avoiding starvation
  - *If I give freshly-baked cookies to half of my nephews and stale bread to the others, I am not fair but I still ensure that nobody starves*
Non-preemptive Schedulers

- **non-preemptive** CPU scheduler will never remove a core from a running process
- Will wait until the process releases the core because
  - It issues a system call
  - It terminates
- Now **obsolete**
How SJF works

- Five students wait for their instructor at the beginning of her office hours
  - Ann needs 20 minutes of her time
  - Bob needs 30 minutes
  - Carol needs 10 minutes
  - Dean needs 5 minutes
  - Emily needs 5 minutes
Examples (I)

- **First-Come First-Served (FCFS):**
  - Simplest and easiest to implement
    - Uses a FIFO queue
  - Seems a good idea but
    - Processes requiring a few ms of core time have to wait behind processes that make much bigger demands
    - *Inacceptable*
Examples (II)

- **Shortest Job First (SJF):**
  - Gives a core to the process requesting the least amount of core time
    - Will reduce average wait
    - Must know ahead of time how much core time each process needs
      - *Not possible*
    - Still lets processes monopolize a core
## FCFS schedule

<table>
<thead>
<tr>
<th>Student</th>
<th>Time</th>
<th>Wait</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ann</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>Bob</td>
<td>30</td>
<td>20</td>
</tr>
<tr>
<td>Carol</td>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>Dean</td>
<td>5</td>
<td>60</td>
</tr>
<tr>
<td>Emily</td>
<td>5</td>
<td>65</td>
</tr>
</tbody>
</table>
The outcome

- Average wait time:
  - \((0 + 20 + 50 + 60 + 65)/5 = 39\) minutes
SJF schedule

<table>
<thead>
<tr>
<th>Student</th>
<th>Time</th>
<th>Wait</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dean</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Emily</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Carol</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Ann</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Bob</td>
<td>30</td>
<td>40</td>
</tr>
</tbody>
</table>
The outcome

- Average wait time:
  - \((0 + 5 + 10 + 20 + 40)/5 = 15\) minutes

- Less than half the wait time of the FCFS schedule
  - The data were rigged
Preemptive Schedulers

- A **preemptive** scheduler can return a running process to the ready queue whenever another process requires that core in a more urgent fashion
  - Has been for too long in the ready queue
  - Has higher priority

- **Sole acceptable solution**
  - Prevents processes from “hogging” a core
Types of preemptive schedulers

- *Preemptive schedulers w/o priorities:*
  - All processes have the same priority
  - Ready queue is FIFO

- *Preemptive schedulers with priorities:*
  - Use multiple queues
  - Differ in the way they adjust process priorities
Round robin (I)

- Assumes all processes have same priority
  - Guaranteed to be starvation-free

- Similar to FCFS but processes only get the a core for \( \text{up to } T_{CPU} \) time units
  - *Time slice* or *time quantum*

- Processes that exceed their time slice return to the end of the ready queue
Round robin (II)

- Ready queue
- System request completion
- Core
- Process exceeds time slice
- System call
How RR works

- Assume
  - Single core
  - Time slice is 100ms (reasonable choice)
  - Ready queue contains processes A, B and C
- A gets core at $t = 0\text{ms}$
- A releases the core at $t = 24\text{ms}$ to do an I/O
- B gets core at $t = 24\text{ms}$
- A returns to ready queue at $t = 32\text{ms}$
- B forced to release the core at $t = 124\text{ms}$
Finding the right time slice (I)

- A small time slice means a good response time
  - No process will ever have to wait more than
    \((n_{\text{readyQueue}} + 1)T_{\text{CPU}}\) time units
    where \(n_{\text{readyQueue}}\) is the number of processes already in the ready queue

- A large time slice means a better throughput
  - Fewer context switches
Finding the right time slice (II)

Ideal CPU schedule

True CPU schedule
The problem

- Want to adjust the time slice to guarantee a maximum waiting time in the ready queue

\[ T_{CPU} = \frac{T_{max}}{(n_{ready\_queue} + 1)} \]

- Works well as long as system is lightly loaded
- Produces very small time slices when system is loaded
  - Too much context switch overhead!
An observation

- The throughput of a system using a RR scheduler actually decreases when its workload exceeds some threshold
  - *Rare* among *physical systems*
  - *Frequent* among systems experiencing *congestion*
    - Freeway throughput actually decreases when its load exceeds some threshold
The solution (I)

- Add *priorities*

- Distinguish among
  - *Interactive processes*
  - *I/O-bound processes*
    - Require small amounts of core time
  - *CPU-bound processes*
    - Require large amounts of core time (*number crunching*)
The solution (II)

- Assign
  - *High priorities* to interactive processes
  - *Medium priorities* to I/O-bound processes
  - *Low priorities* to CPU-bound processes
The solution (III)

- Assign
  - *Smallest time slices* to interactive processes
  - *Medium time slices* to I/O-bound processes
  - *Biggest time slices* to CPU-bound processes

- Allow higher priority processes to *steal cores* from lower priority processes
The outcome

- Interactive processes will get good response times
- CPU-bound processes will get the CPU
  - Less frequently than with RR
  - For longer periods of time
  - Less context switch overhead
Two problems

- **How to assign priorities to processes?**
  - Process behaviors may change during their execution
    - *Should adjust process priorities*

- **How to avoid starvation?**
  - *Adjust process priorities*
Multi-Level with Feedback Queues

- Use *dynamic priorities*

- **Reward**
  - Processes that issue system calls
  - Processes that interact with user
  - Processes that have been a long time in the ready queue

- **Penalize**
  - Processes that exceed their time slice
Implementation (I)

- High priority queue
- Medium priority queue
- Low priority queue

System call

CPU
Implementation (II)

- Time slice increase when priority decreases, say
  - T for high priority processes
  - 2T for medium priority processes
  - 4T for low priority processes
The priority game

- Different systems have different conventions for priorities
  - 0 is highest
    - Most UNIX systems, Linux
  - 0 is lowest
    - UNIX System V Release 4 (V.4)
    - Windows NT and after
System V.4 scheduler

- Three process classes:
  - Real-time
  - Time-sharing
  - System (for kernel processes)

- Each process class has its own priority levels
  - Real-time processes have highest priority
  - Time-sharing lowest
Real-time processes

- Have **fixed priorities**
  - As in Windows scheduler
- System administrator can define
  - A different *quantum size* (**rt_quantum**) for each priority level
Timesharing processes (I)

- Have *variable priorities*
- System administrator can specify the parameters of each priority level
  - Maximum flexibility
  - Maximum risk of making a bad choice

*Leaving too many tuning options for the system administrator increases the chances that some options will be poorly selected*
Parameters include

- Quantum size (ts_quantum)
- New priority for processes that use their whole CPU quantum (ts_tqexp)
- New priority for processes returning from blocking state (ts_slpret)
Timesharing processes (III)

- Maximum amount of time a process can remain in the ready queue without having its priority recomputed ($ts\_maxwait$)
- New priority for processes that have been in the ready queue for $ts\_maxwait$ ($ts\_lwait$)
Example

- System has four priority levels
  - 0 is lowest
  - 3 is highest
- Anything after a pound sign is a comment

<table>
<thead>
<tr>
<th>#ts_quantum</th>
<th>ts_tqexp</th>
<th>ts_slpret</th>
<th>ts_maxwait</th>
<th>ts_lwait</th>
<th>LEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
<td>1</td>
<td>50000</td>
<td>1</td>
<td># 0</td>
</tr>
<tr>
<td>500</td>
<td>0</td>
<td>2</td>
<td>20000</td>
<td>2</td>
<td># 1</td>
</tr>
<tr>
<td>200</td>
<td>1</td>
<td>3</td>
<td>10000</td>
<td>3</td>
<td># 2</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>3</td>
<td>10000</td>
<td>3</td>
<td># 3</td>
</tr>
</tbody>
</table>
How to read it

<table>
<thead>
<tr>
<th>#ts_quantum</th>
<th>ts_tqexp</th>
<th>ts_slpret</th>
<th>ts_maxwait</th>
<th>ts_lwait</th>
<th>LEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
<td>1</td>
<td>50000</td>
<td>1</td>
<td># 0</td>
</tr>
<tr>
<td>500</td>
<td>0</td>
<td>2</td>
<td>20000</td>
<td>2</td>
<td># 1</td>
</tr>
<tr>
<td>200</td>
<td>1</td>
<td>3</td>
<td>10000</td>
<td>3</td>
<td># 2</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>3</td>
<td>10000</td>
<td>3</td>
<td># 3</td>
</tr>
</tbody>
</table>

- New priorities can be
  - Rewarding a “good” behavior: ts_slpret and ts_lwait
  - Penalizing CPU “hogs”: ts_tqexp
How?

- We *increase* the priority of processes that
  - Have completed a system call
    - They might become less CPU-bound
  - Have waited a long time in the ready queue
    - To prevent starvation

- We *decrease* the priority of processes that
  - Have exhausted their time quantum
    - They might be more CPU-bound
Second example (I)

<table>
<thead>
<tr>
<th>#ts_quantum</th>
<th>ts_tqexp</th>
<th>ts_slpret</th>
<th>ts_maxwait</th>
<th>ts_lwait</th>
<th>LEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
<td>1</td>
<td>50000</td>
<td>1</td>
<td># 0</td>
</tr>
<tr>
<td>500</td>
<td>X</td>
<td>2</td>
<td>20000</td>
<td>2</td>
<td># 1</td>
</tr>
<tr>
<td>200</td>
<td>1</td>
<td>3</td>
<td>10000</td>
<td>3</td>
<td># 2</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>Y</td>
<td>10000</td>
<td>4</td>
<td># 3</td>
</tr>
<tr>
<td>100</td>
<td>3</td>
<td>4</td>
<td>10000</td>
<td>Z</td>
<td># 4</td>
</tr>
</tbody>
</table>

- Table now defines five priority levels
- What are the **correct values** for X, Y and Z?
Second example (II)

<table>
<thead>
<tr>
<th>ts_quantum</th>
<th>ts_tqexp</th>
<th>ts_slpret</th>
<th>ts_maxwait</th>
<th>ts_lwait</th>
<th>LEVEL</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
<td>1</td>
<td>50000</td>
<td>1</td>
<td>#</td>
<td>0</td>
</tr>
<tr>
<td>500</td>
<td>X</td>
<td>2</td>
<td>20000</td>
<td>2</td>
<td>#</td>
<td>1</td>
</tr>
<tr>
<td>200</td>
<td>1</td>
<td>3</td>
<td>10000</td>
<td>3</td>
<td>#</td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>Y</td>
<td>10000</td>
<td>4</td>
<td>#</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>3</td>
<td>4</td>
<td>10000</td>
<td>Z</td>
<td>#</td>
<td>4</td>
</tr>
</tbody>
</table>

- X is the new priority for processes at level 1 that exceed their time quantum
  - Must be lower than current priority
- X=0
Second example(III)

<table>
<thead>
<tr>
<th>#ts_quantum</th>
<th>ts_tqexp</th>
<th>ts_slpret</th>
<th>ts_maxwait</th>
<th>ts_lwait</th>
<th>LEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
<td>1</td>
<td>50000</td>
<td>1</td>
<td># 0</td>
</tr>
<tr>
<td>500</td>
<td>0</td>
<td>2</td>
<td>20000</td>
<td>2</td>
<td># 1</td>
</tr>
<tr>
<td>200</td>
<td>1</td>
<td>3</td>
<td>10000</td>
<td>3</td>
<td># 2</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>Y</td>
<td>10000</td>
<td>4</td>
<td># 3</td>
</tr>
<tr>
<td>100</td>
<td>3</td>
<td>4</td>
<td>10000</td>
<td>Z</td>
<td># 4</td>
</tr>
</tbody>
</table>

- Y is a new priority for processes at level 3 that exceed their time quantum
  - Must be higher than current priority
- Y=4
Second example (IV)

<table>
<thead>
<tr>
<th>#ts_quantum</th>
<th>ts_tqexp</th>
<th>ts_slpret</th>
<th>ts_maxwait</th>
<th>ts_lwait</th>
<th>LEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
<td>1</td>
<td>50000</td>
<td>1</td>
<td># 0</td>
</tr>
<tr>
<td>500</td>
<td>0</td>
<td>2</td>
<td>20000</td>
<td>2</td>
<td># 1</td>
</tr>
<tr>
<td>200</td>
<td>1</td>
<td>3</td>
<td>10000</td>
<td>3</td>
<td># 2</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>4</td>
<td>10000</td>
<td>4</td>
<td># 3</td>
</tr>
<tr>
<td>100</td>
<td>3</td>
<td>4</td>
<td>10000</td>
<td>Z</td>
<td># 4</td>
</tr>
</tbody>
</table>

- Z is a the new priority for processes at level 4 that have waited too long in the ready queue
  - Should be higher than current priority
  - Level 4 already is the highest priority

- \( Z = 4 \)
Second example (V)

<table>
<thead>
<tr>
<th>#ts quantum</th>
<th>ts_tqexp</th>
<th>ts_slpret</th>
<th>ts_maxwait</th>
<th>ts_lwait</th>
<th>LEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
<td>1</td>
<td>50000</td>
<td>1</td>
<td># 0</td>
</tr>
<tr>
<td>500</td>
<td>0</td>
<td>2</td>
<td>20000</td>
<td>2</td>
<td># 1</td>
</tr>
<tr>
<td>200</td>
<td>1</td>
<td>3</td>
<td>10000</td>
<td>3</td>
<td># 2</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>Y</td>
<td>10000</td>
<td>4</td>
<td># 3</td>
</tr>
<tr>
<td>100</td>
<td>3</td>
<td>4</td>
<td>100000</td>
<td>Z</td>
<td># 4</td>
</tr>
</tbody>
</table>

- Recall that
  - `ts_slpret` and `ts_lwait` reward “good” behaviors
  - `ts_tqexp` penalizes a “bad” one
An exercise

- Fill the missing values

<table>
<thead>
<tr>
<th>ts_quantum</th>
<th>ts_tqexp</th>
<th>ts_slpret</th>
<th>ts_maxwait</th>
<th>ts_lwait</th>
<th>LEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>X</td>
<td>1</td>
<td>50000</td>
<td>1</td>
<td># 0</td>
</tr>
<tr>
<td>500</td>
<td>Y</td>
<td>2</td>
<td>20000</td>
<td>2</td>
<td># 1</td>
</tr>
<tr>
<td>200</td>
<td>1</td>
<td>3</td>
<td>10000</td>
<td>3</td>
<td># 2</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>Z</td>
<td>10000</td>
<td>V</td>
<td># 3</td>
</tr>
<tr>
<td>100</td>
<td>3</td>
<td>U</td>
<td>10000</td>
<td>W</td>
<td># 4</td>
</tr>
</tbody>
</table>
The solution

<table>
<thead>
<tr>
<th>#ts_quantum</th>
<th>ts_tqexp</th>
<th>ts_slpret</th>
<th>ts_maxwait</th>
<th>ts_lwait</th>
<th>LEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>X=0</td>
<td>1</td>
<td>50000</td>
<td>1</td>
<td># 0</td>
</tr>
<tr>
<td>500</td>
<td>Y=0</td>
<td>2</td>
<td>20000</td>
<td>2</td>
<td># 1</td>
</tr>
<tr>
<td>200</td>
<td>1</td>
<td>3</td>
<td>10000</td>
<td>3</td>
<td># 2</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
<td>Z=4</td>
<td>10000</td>
<td>V=4</td>
<td># 3</td>
</tr>
<tr>
<td>100</td>
<td>3</td>
<td>U=4</td>
<td>10000</td>
<td>W=4</td>
<td># 4</td>
</tr>
</tbody>
</table>

- Recall that the only valid priorities are 0 to 4!
MacOS X Scheduler (I)

- Mac OS X uses a multilevel feedback queue
  - Manages threads, not processes
  - Four priority bands for threads
    - Normal
    - System high priority
    - Kernel mode only
    - Real-time
MacOS Scheduler (II)

- Thread priorities will vary
  - Must remain within their bands
  - Real-time threads tell the scheduler the number $A$ of clock cycles they will need out of the next $B$ clock cycles
    - Say 4000 out of the next 9000 clock cycles
Windows Scheduler

- An update of the old VMS scheduler
- Scheduler manages *threads* rather than processes.
- Has 32 priority levels:
  - 16 to 31 for *real-time threads*
  - 0 to 15 for *other threads*
- *Priority zero reserved* for the system thread zeroing free pages
Priority classes

- Apply to processes
- Five classes of process priorities
  - IDLE_PRIORITY_CLASS
  - BELOW_NORMAL_PRIORITY_CLASS
  - NORMAL_PRIORITY_CLASS
  - ABOVE_NORMAL_PRIORITY_CLASS
  - HIGH_PRIORITY_CLASS
  - REALTIME_PRIORITY_CLASS
Base priorities

- Apply to threads
- Defined within each process class
  - `THREAD_PRIORITY_IDLE`
  - `THREAD_PRIORITY_LOWEST`
  - `THREAD_PRIORITY BELOW NORMAL`
  - `THREAD_PRIORITY NORMAL`
  - `THREAD_PRIORITY ABOVE NORMAL`
  - `THREAD_PRIORITY HIGHEST`
  - `THREAD_PRIORITY TIME CRITICAL`
Real-time threads

- Real-time processes belong to **REALTIME_PRIORITY_CLASS**
- Threads at *fixed priorities* between 16 and 31
  - Specified by their *base priority*
- Scheduling is *round-robin* within each priority level
Other threads (I)

- Run at **variable priorities** between 1 and 15
- Each thread has a **base priority**
  - Value depends on process class and thread priority level within class
    - 1 for all threads with THREAD_PRIORITY_IDLE
    - 15 for all threads with THREAD_PRIORITY_TIME_CRITICAL
Other threads (II)

- Thread priorities *never go below* their base priority
- These priorities are
  - "Boosted" whenever they return from the blocked state
  - *Decrementated* when they exhaust their time slice
Thread affinity

- **Thread affinity** specifies the set of processors on which the thread can run.
  - "Setting thread affinity should generally be avoided because it can interfere with the scheduler's ability to schedule threads effectively across processors."
Thread ideal processor

- Instructs the scheduler to run the thread on that processor whenever possible
  - Does not guarantee that processor will always be chosen
Note

- Do not be confused by the two different usages of "suspended"
  - Suspending a process is the same as swapping it out
  - Suspending a thread in this context means moving it to the blocked state
Guaranteed scheduling

- Class of scheduling algorithms that want to ensure that its process has its *fair share* of CPU time
- Penalize processes that have used a large amount of CPU
- Most versions of UNIX, Windows NT and after, Linux
Old UNIX Scheduler (I)

- Priorities take into account *past CPU usage*
  
  
  \[ p_{\text{usrpri}} = \text{PUSER} + \frac{p_{\text{cpu}}}{2} + p_{\text{nice}} \]

  where
  - \text{PUSER} is the user's base priority
  - \text{p_cpu} its current CPU usage
  - \text{p_nice} a user-settable parameter
Old UNIX Scheduler (II)

- \( p_{cpu} \) is updated every second according to a decay function
  \[
  \text{decay}(p_{cpu}) = \frac{p_{cpu}}{2}
  \]
- After \( k \) seconds, penalty is decreased by a factor \( 1/2^k \)
The time quantum is 100 ms

\[ p_{\text{usrpri}} = PUSER + \frac{p_{\text{cpu}}}{4} + 2 \times p_{\text{nice}} \]

\( p_{\text{cpu}} \) is updated every second according to:

\[ p_{\text{cpu}} = \frac{(2 \times ld)}{(2 \times ld + 1)} \times p_{\text{cpu}} + p_{\text{nice}} \]

where \( ld \) is a sampled average of the length of the run queue over the last minute
BSD scheduler (II)

- Unlike the old UNIX scheduler, the BSD scheduler takes into account the system load
  - Through length of ready queue
    - “Load average”
  - Forgives old CPU usage *more slowly* when system load is *high*
Linux 2.4 scheduler (I)

- Partitions the CPU time into *epochs*.
- At the beginning of each epoch, each process is assigned a *time quantum*
  - Specifies the maximum CPU time the process can have during that epoch.
- Processes that exhaust their time quantum cannot get CPU time until the next epoch starts.
Linux 2.4 scheduler (II)

- Processes that release the CPU before their time quantum is exhausted can get more CPU time during the same epoch.
- Epoch ends when all ready processes have exhausted their time quanta.
- Priority of a process is the sum of its base priority plus the amount of CPU time left to the process before its quantum expires.
Stride scheduling (I)

- Deterministic fair-share scheduler
- Start by allocating tickets to processes/threads
  - More tickets mean more core time
- Each thread has a **stride**
  - *Inversely* proportional to the number $n$ of tickets it has
  - If thread A has 10 tickets, thread B has 5 tickets and thread C has 20 tickets
    - Stride of A is 10, stride of B is 20 and stride of C is 5

*NOT COVERED THIS SEMESTER*
Stride scheduling (II)

- Each process has a *pass* value
  - Initially set to process stride

- Each time a process releases the CPU
  - Scheduler selects process with *lowest pass*
  - Gives it the CPU for a *fixed time slide*

- Each time a process gets the CPU
  - Scheduler *adds* the process stride to its pass value
The key idea

- Use epochs
- Have a thread priority ("pass")
  - Initially set to "stride"
    - Inversely proportional to the number of tickets allocated to
- Always schedule thread with lowest pass
- Penalize differently past core usage
Stride scheduling (II)

- Scheme is *starvation free*
  - Processes that do not get any CPU time keep their original pass values
  - Other processes will see their pass values increase

NOT COVERED THIS SEMESTER
### Example

<table>
<thead>
<tr>
<th>Round</th>
<th>Pass values</th>
<th>Scheduler will pick thread</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Thread A</td>
<td>Thread B</td>
</tr>
<tr>
<td></td>
<td>10 tickets stride is 10</td>
<td>5 tickets stride is 20</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

*NOT COVERED THIS SEMESTER*
Explanations

- Process C gets first slot
  - Lowest pass value (4)
- Process C gets second slot
  - Lowest pass value (8)
- Process A gets third slot
  - Lowest pass value (10)
- Process C gets fourth slot
  - Lowest pass value (12)
Note

Whenever two threads have the same pass value, the scheduler will pick the thread with the *lowest* *stride*
FreeBSD 5.0 ULE scheduler

- Designed for threads running on multicore architectures
  - For more details

- Two parts
  - Low-level scheduler
    - Run every time a core is released
  - High-level scheduler
    - Run every second
Low-level scheduler

- Kernel maintains a set of *run queues* for each CPU
  - With different priorities
- Low-level scheduler selects first thread on highest-level non-empty run queue
High-level scheduler

- Reevaluates thread priorities
  - Real-time threads have fixed priorities
  - Scheduler detects interactive threads on the base of their interactivity score:
    - \(\text{Scaling factor} \times \frac{\text{Sleep time}}{\text{Run time}}\)
- Also assigns threads to CPUs
  - Complex process
Observations

- Low-level scheduler is kept simple
  - Quick decisions

- High-level scheduler uses a very clever method to detect interactive processes

\[
\frac{(Voluntary) Sleep time}{Run time}
\]

- Must still pick length of observation period
  - Short term v. long term behavior