DATALOGISK INSTITUT
KØBENHAVNS UNIVERSITET

Principles of Computer System Design

Lecture 11

Kenneth Skovhede
skovhede@diku.dk

Intended Learning Outcomes

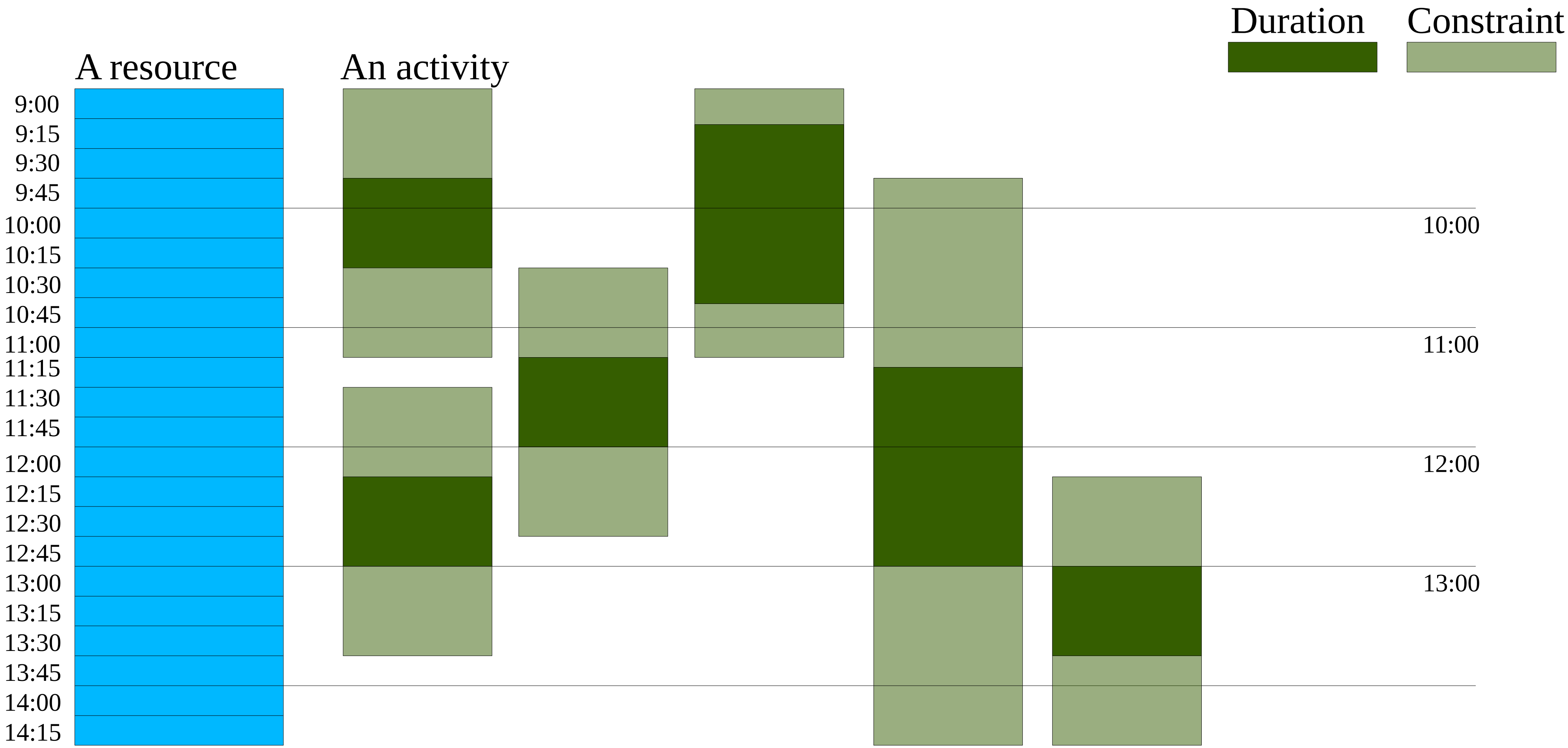
- Compare different types of scheduling goals
- Compare different types of scheduling methods
- Implement a simple scheduler with priorities
- Explain how the Linux scheduler works

What needs scheduling?

- Anything that is a limited resource
  - Harddisk
  - Memory
  - Processor
  - Network
  - Display
  - ...

The scheduling problem

- How can we make the most of a limited resource, such as an auditorium?



The scheduling problem

- How hard is scheduling?
- It can be viewed as a variant of the 0-1 knapsack problem, so it is NP complete
- Especially, greedy algorithms will not work
- The scheduler needs to run constantly, so it must be fast

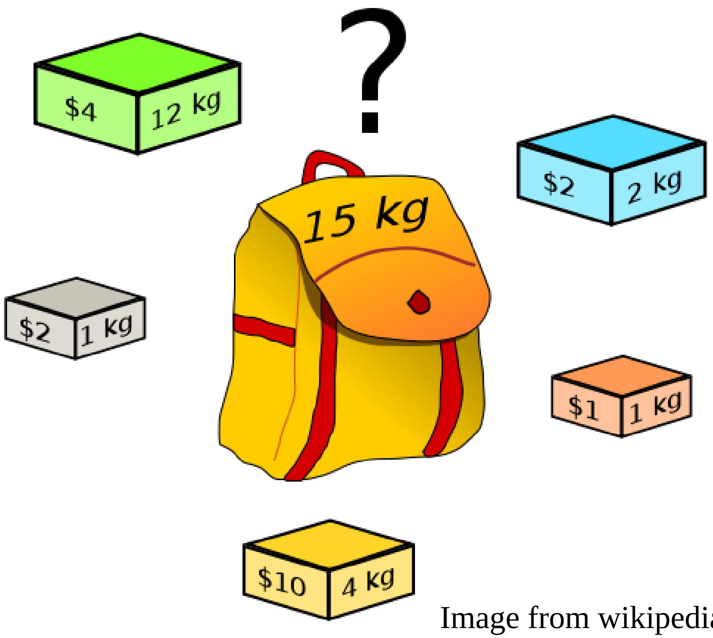


Image from wikipedia

Goal

- What is the objective?
  - Maximize utilization?
  - Maximize responsiveness?
  - Minimize waiting time?

- Obtaining the objective may be complicated

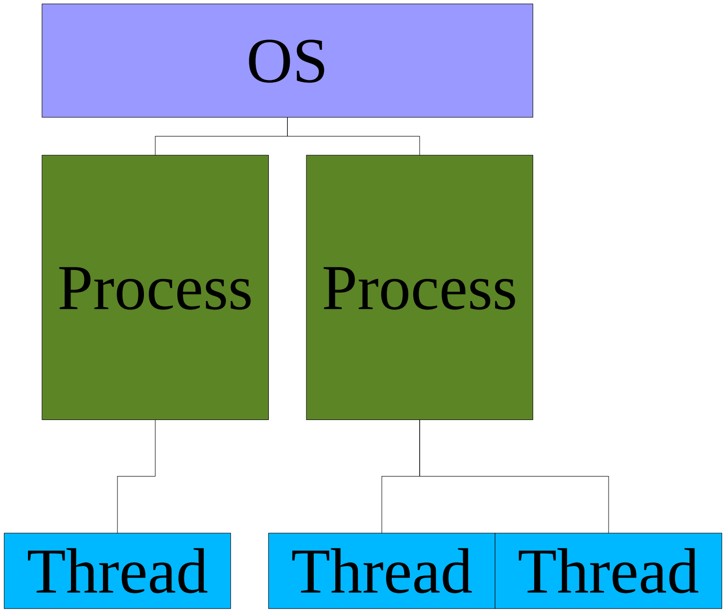
- Suppose we aim to maximize responsiveness
- What priority should the input handler have?
- What priority should the render thread have?
- What priority should the disk driver have?

Terminology

- A process is an operating system unit
  - Has an associated Process Control Block (PCB)
  - A PCB keeps track of:
    - Open handles (files, network, etc)
    - Permissions
    - Memory regions
    - Other (eg. Child processes)
- A thread is a lightweight process
  - Shares the process PCB
    - Can use process handles
    - Has process permissions
    - Has access to process memory

Thread strategy 1

- OS does not know about threads
- Advantages
  - Scheduler has less work
  - Time slice can be larger, which gives less overhead
- Disadvantages
  - Each process must implement a mini-scheduler
  - Hard to preempt threads
  - Not good for multi-core

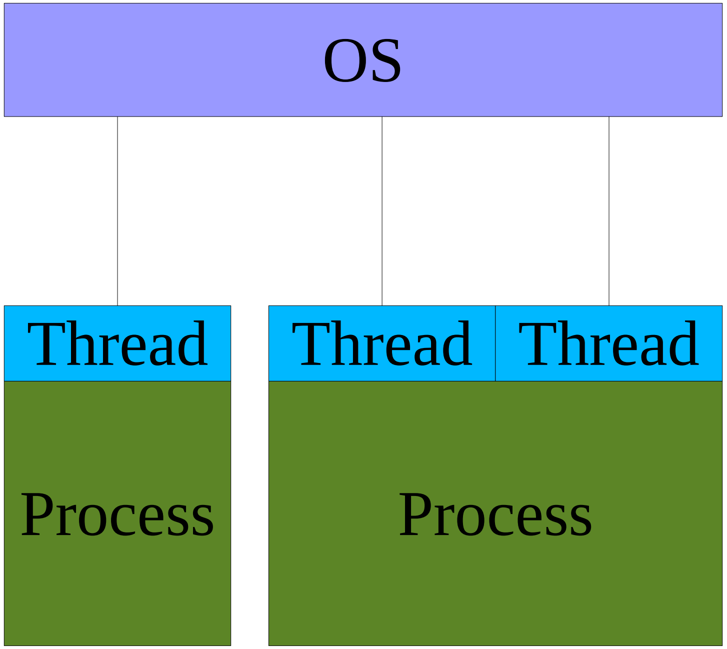


Thread strategy 2

- OS controls threads

- Advantages
  - A thread cannot block the process
  - Only one scheduler
  - Supports multi-core

- Disadvantages
  - More work for scheduler
  - Costly to do thread switch

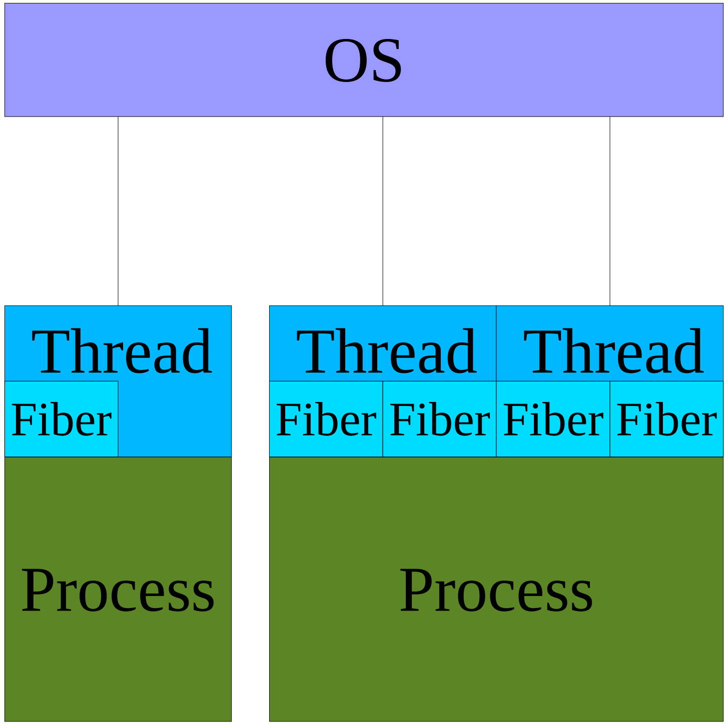


Thread strategy 3

- OS controls threads
- Threads run fibers

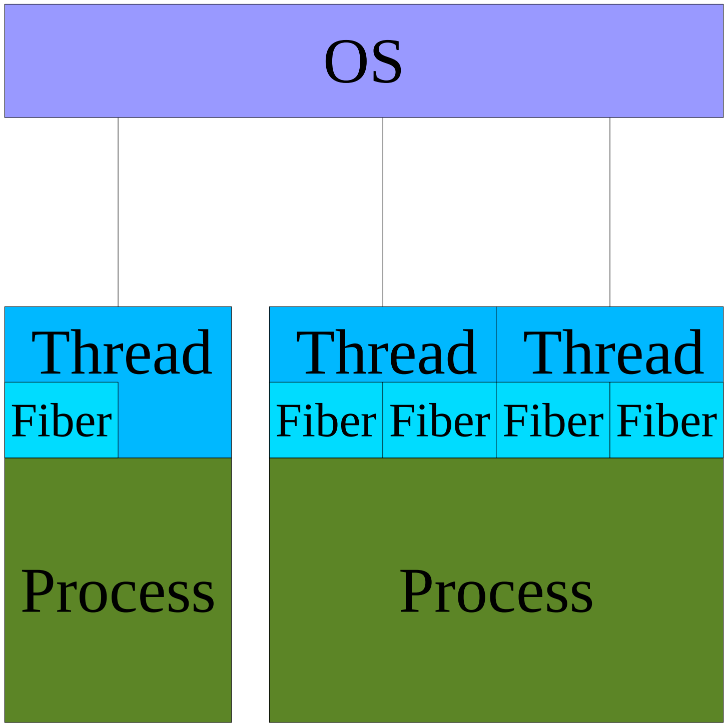
- Advantages
  - A thread cannot block the process
  - Only one scheduler
  - Supports multi-core

- Disadvantages
  - Each thread must implement a mini scheduler



Thread strategy 3, continued

- Variations exist where the OS is also fiber aware
- Best of both worlds
- Allows a thread to use the OS scheduler



The thread state data

- The thread state is usually limited to:
  - The current set of registers
  - The current stack

- To switch thread we need to
  - Preserve the current thread state
  - Select a new thread (scheduling)
  - Restore the new thread state

Non-preemptive scheduling

- Also known as cooperative scheduling
- Simple to implement, just switch thread state
- Can be extremely fast, usermode only
- Used in programming libraries, eg. CSP
- Not good for OS because one thread can lock system
- Does not allow multi-core execution
- Not really possible to implement priorities

```
for(i = 0; i < count; i++)
{
    … do some work …
    yield();
}
```

A simple yield() function

```
struct pcb* cur_pcb;

void yield() {
    if (set_jmp(pcb->jumpbuf) == 0) {
        cur_pcb = get_next_thread();
        long_jmp(cur_pcb->jumbuf, 1);
    }
}
```

Preemptive time slicing

- Each thread gets a slice of CPU time
- When the thread starts, a timer is activated
- If the thread yields, the timer is deactivated
- If the timer expires, a kernel level function does:
  - Copies all registers onto the thread stack
  - Selects another thread
  - Restores registers
  - Resumes thread

- A single process cannot lock the OS/process
- Available time can vary with thread priority
- Expensive due to kernel ring switches

Preemptive time slicing, continued

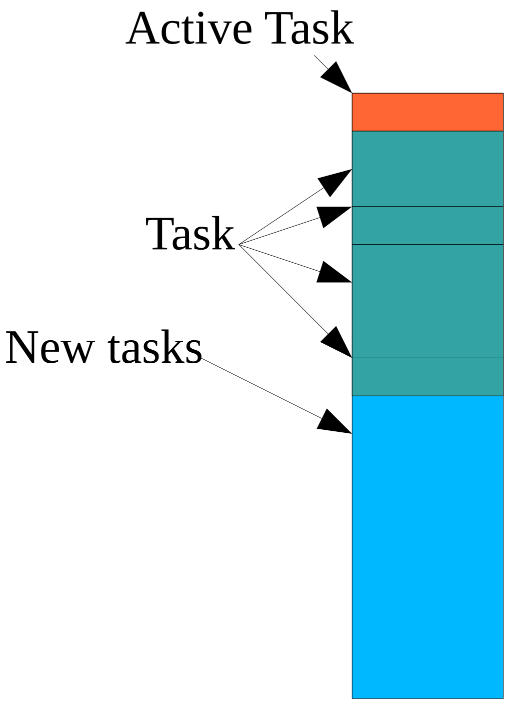
- Usually implemented with a timer
- The timer issues an interrupt
- The interrupt code calls the yield function

- Even though it is conceptually simple to perform, it is hard to implement correctly
  - What happens if the thread is doing IO?
  - What happens if the thread holds a mutex?

Scheduling policies

- First-Come, First-Served
- Aka First-In, First-Out (FIFO)
- Very simple to implement, just a queue
- Priorities can cause starvation

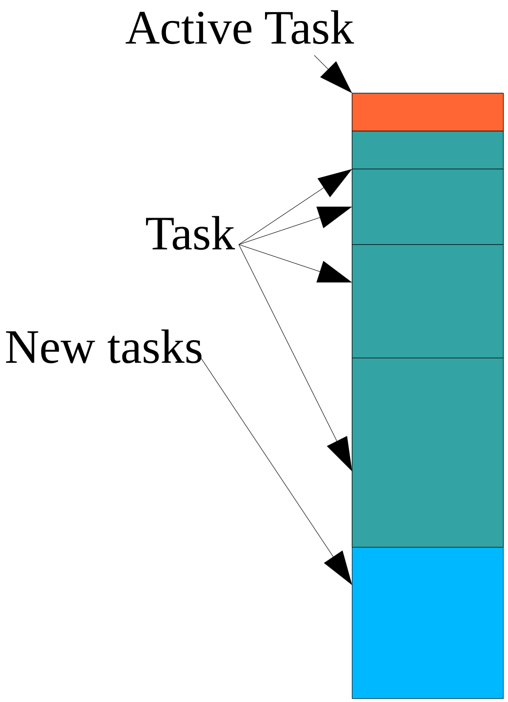
- Maximizes what?



Scheduling policies

- Shortest job first
- Simple to implement, just a sorted list
- Priorities can cause starvation
- Requires that we know the length
- A greedy type algorithm

- Maximizes responsiveness?

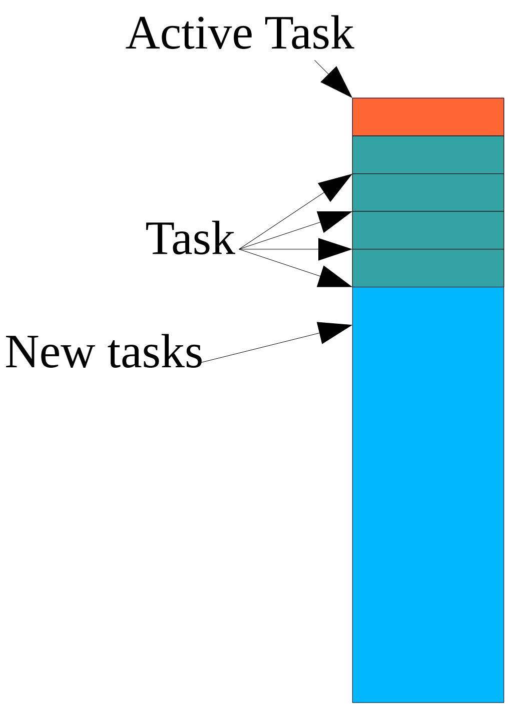


Scheduling policies

- Round robin
- Split up large items
- Process chunks in turns

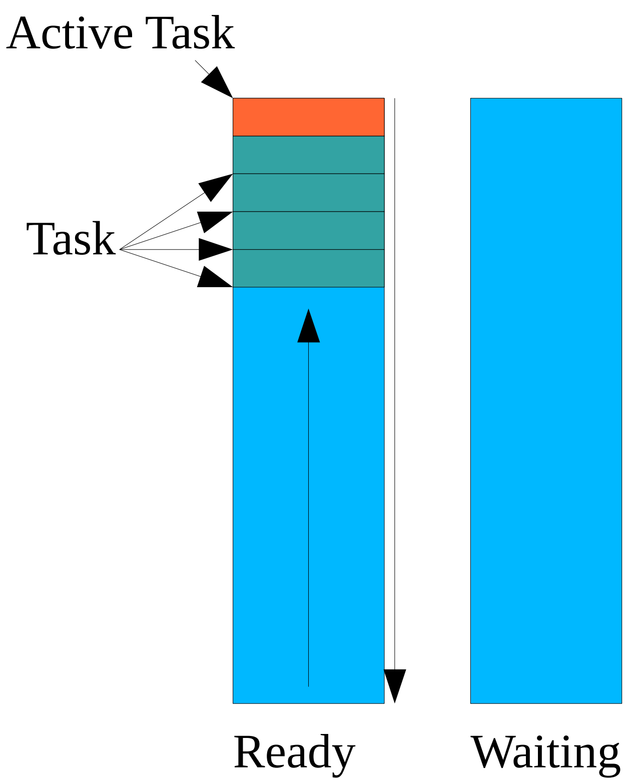
- Maximizes responsiveness

- What size should a “chunk” be?



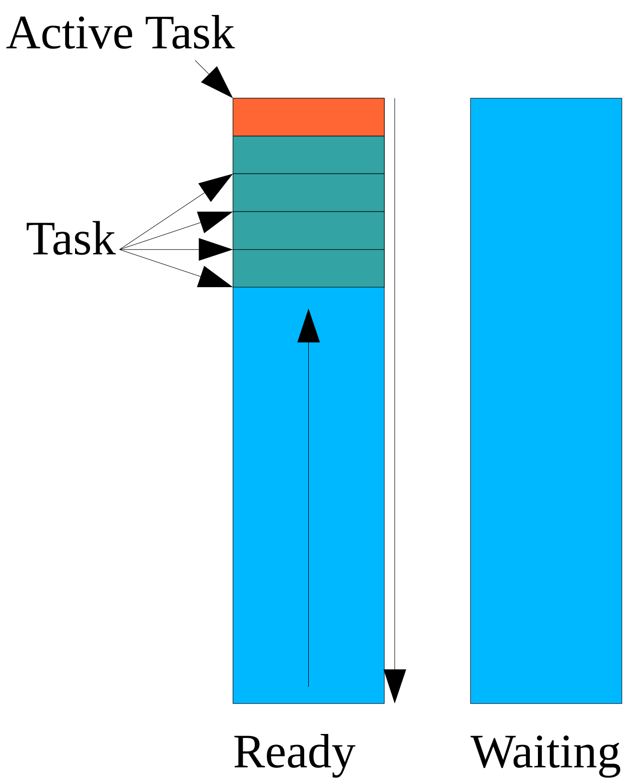
Simple round-robin scheduler

- Pick the top task in ready queue
- When time expires or thread yields, move task to bottom of queue
- If thread is blocked (eg I/O), move it to wait queue
- When block condition stops, move it back to ready queue



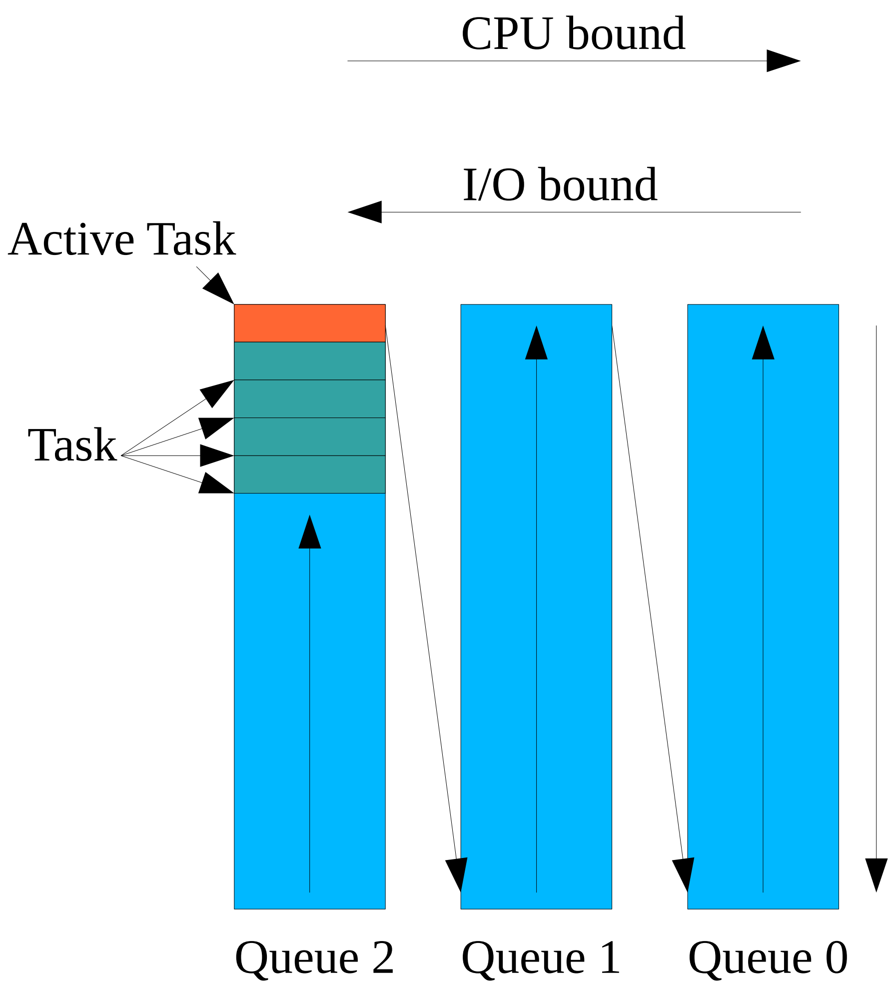
Simple round-robin scheduler with aging

- Same as before, but each task has an age
- Before a task is picked from the ready queue, the age of all ready tasks is incremented
- When task is re-inserted in ready queue, its age is reset
- After ages are incremented, queue is sorted with oldest first
- Can be modified with priorities, by setting the start/reset age



Multilevel feedback queue

- Start with high priority queue
- Run top task
- If task completes within time frame, move it to wait queue
- Else, move it to lower priority queue
- When queue is empty, process next queue
- If process blocks on IO, move it to higher priority queue



Activity #1

Why prioritize threads blocking on IO?

Activity #1

- Perceived responsiveness
  - If the IO was a user event, eg. keypress, we want to process it quickly to avoid lag

- It increases throughput
  - While a thread blocks on IO, it does not consume CPU cycles
  - If IO bound threads have high priority, they can issue IO requests faster at almost no CPU cost

Multilevel feedback queue, continued

- Multilevel feedback queue is the most common
- Each queue can have different policies
- FreeBSD uses 255 queues, some are kernel reserved

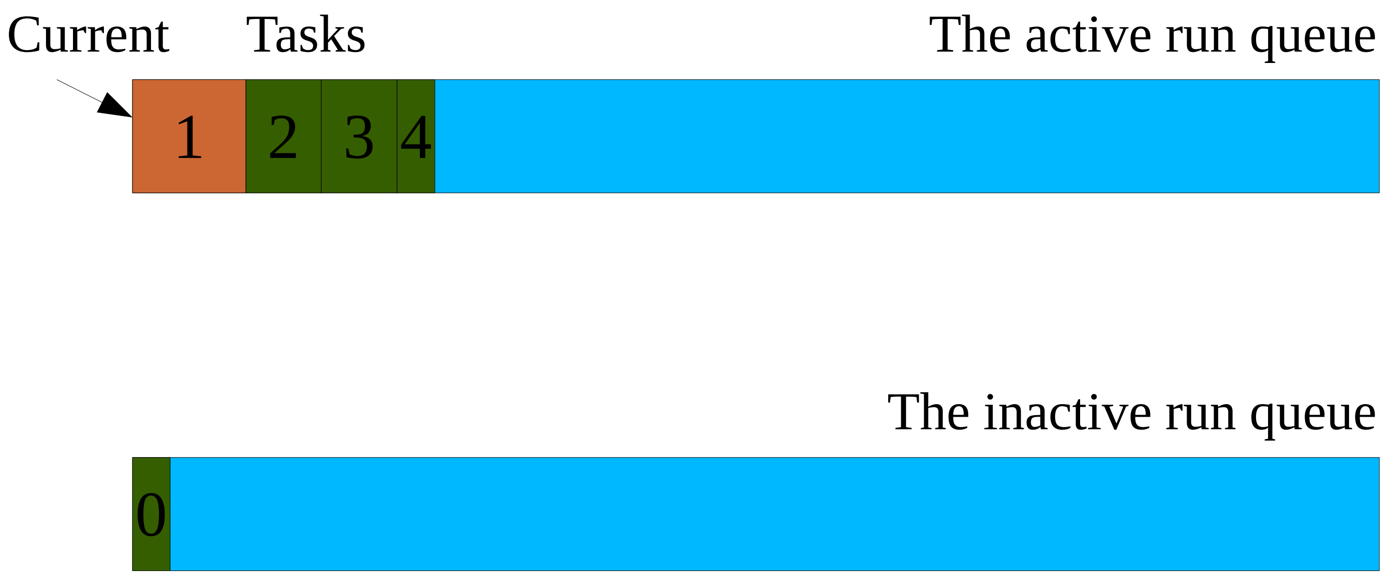
The O(n) scheduler in 2.4

- Runs in O(n)
- Reschedules occasionally
  - Pre 2.4 each second
  - Post 2.4 each epoch (and on reschedule events)

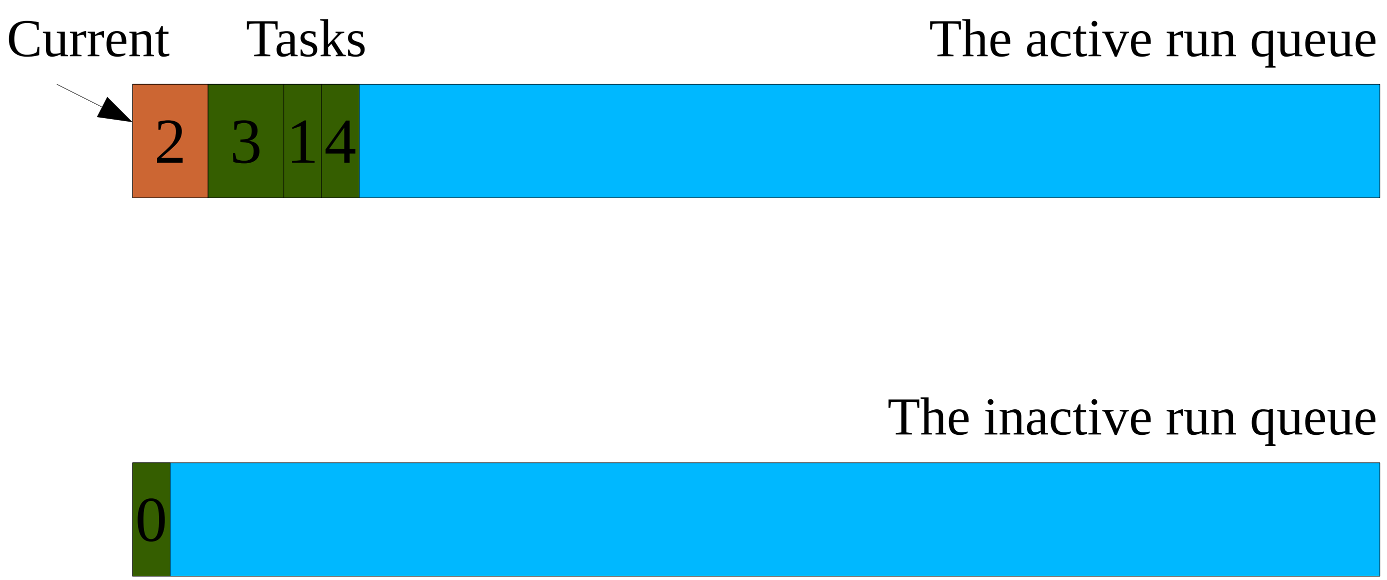
- Each thread is allocated a quantum (time slice)
- The quantum is measured in ticks, 10ms on x86
- The epoch is the time required to use all quantums
- Does not necessarily exhaust the entire quantum in one continuous execution
- Uses the “goodness()” function to determine if a quantum should be interrupted

The O(n) scheduler in 2.4

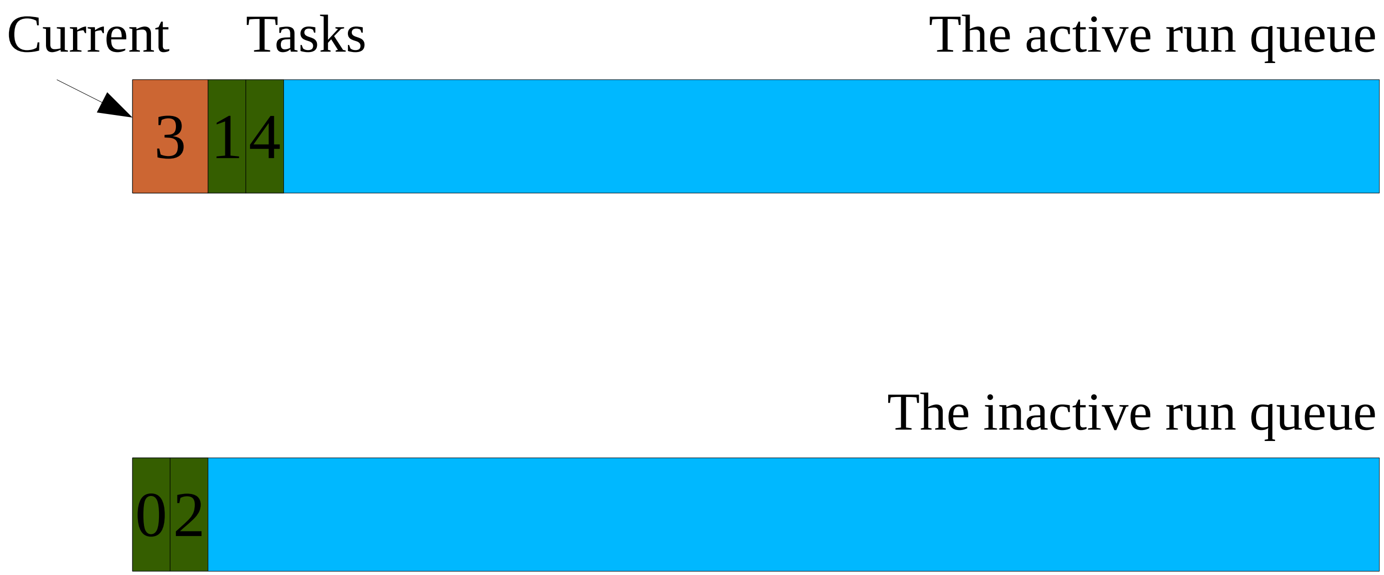
The O(n) scheduler in 2.4



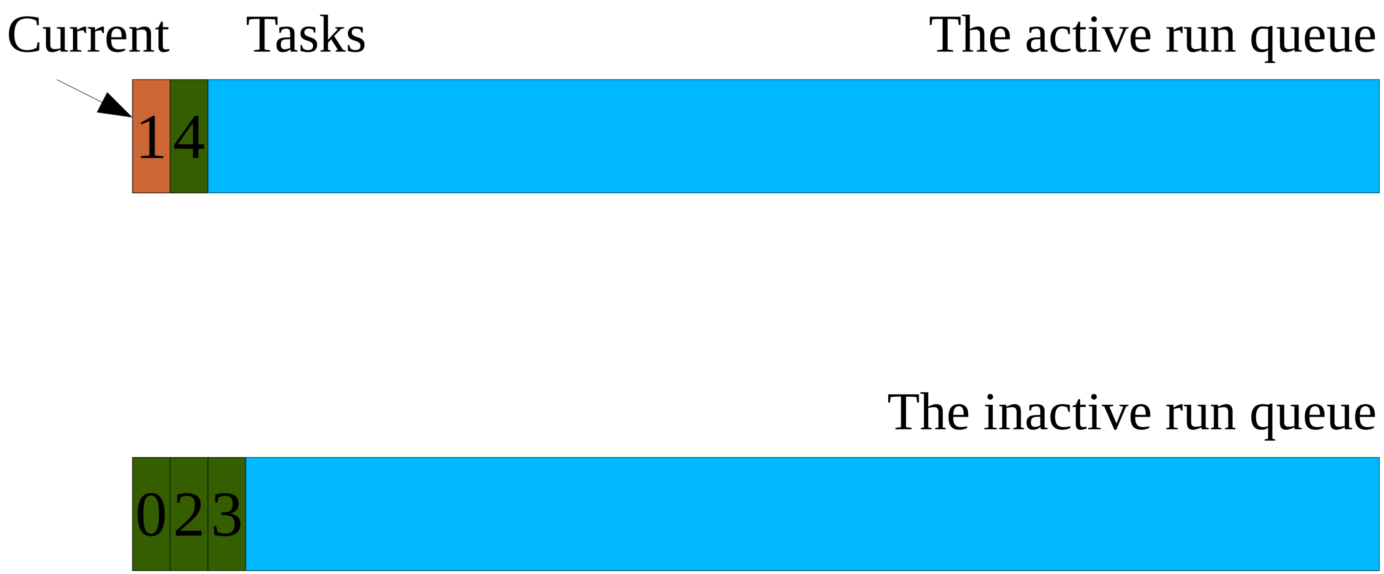
The O(n) scheduler in 2.4



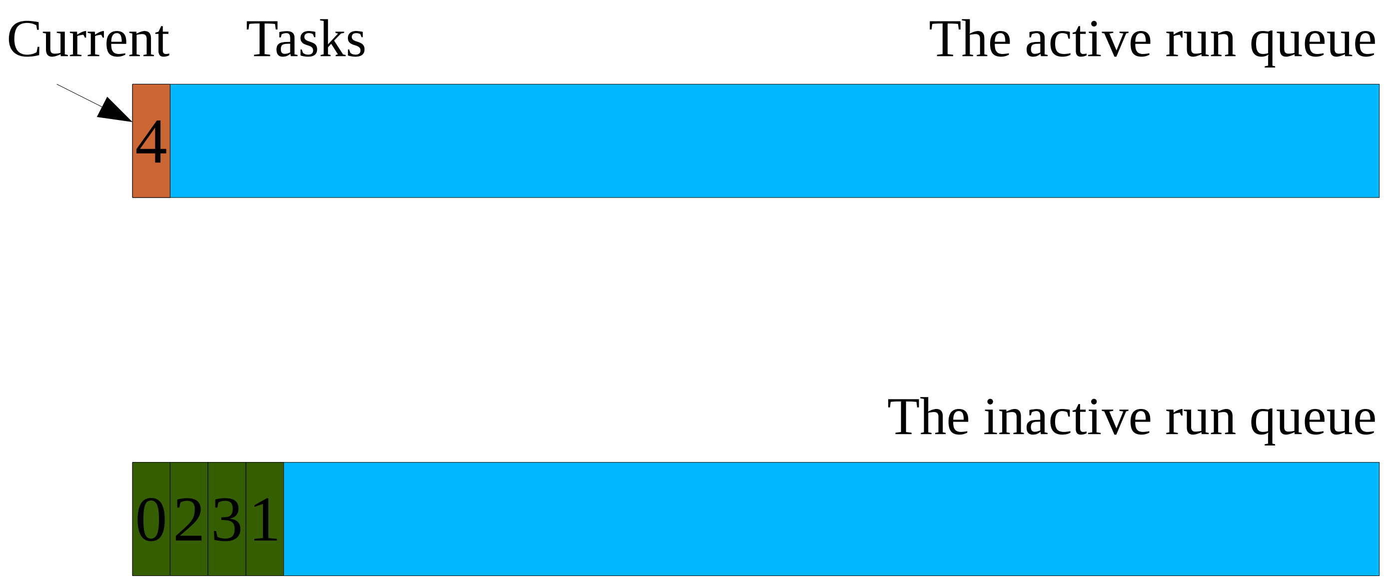
The O(n) scheduler in 2.4



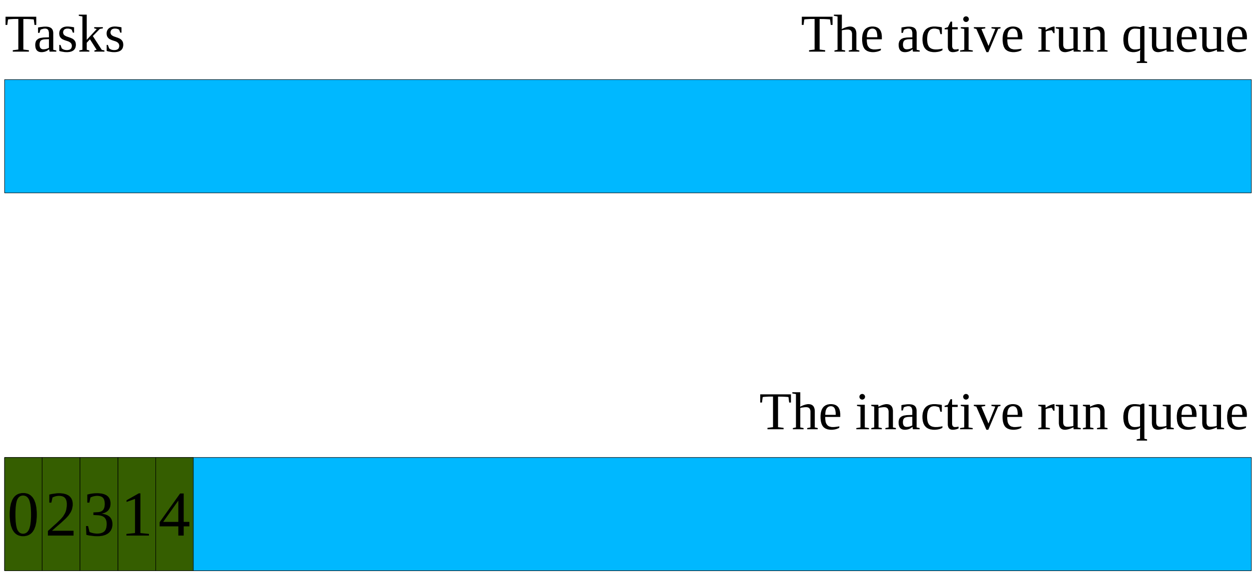
The O(n) scheduler in 2.4



The O(n) scheduler in 2.4

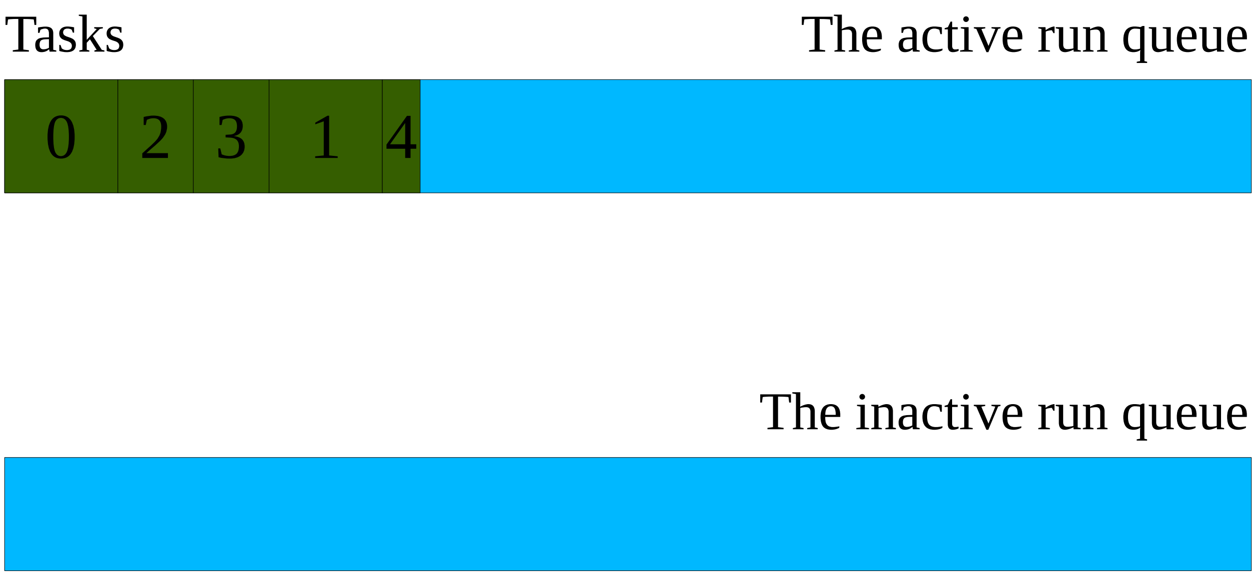


The O(n) scheduler in 2.4



- This is the epoch
- We now need to re-assign quantums and re-insert

The O(n) scheduler in 2.4



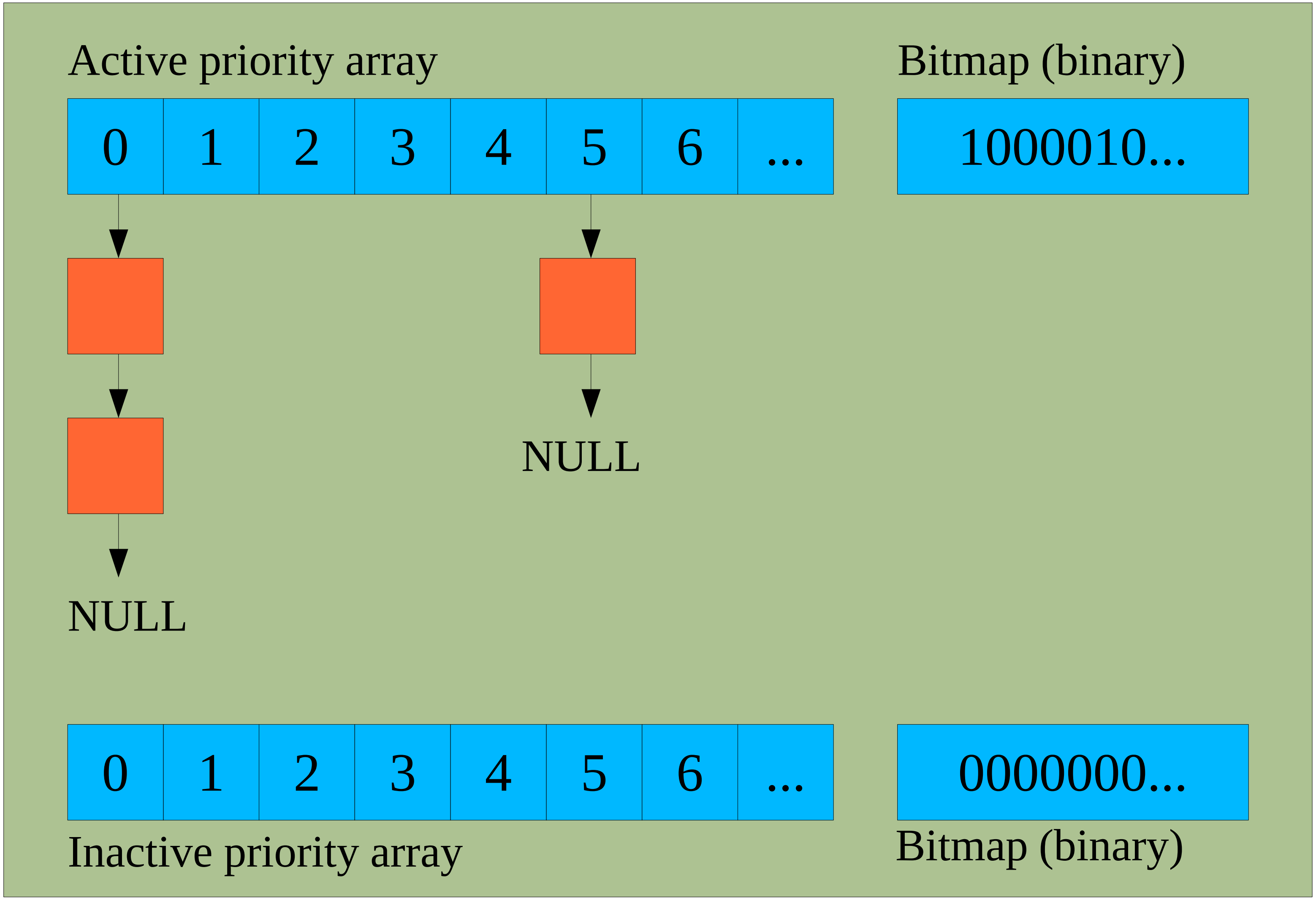
- We must now re-sort before we can run

The O(n) scheduler in 2.4

The O(1) scheduler in 2.6

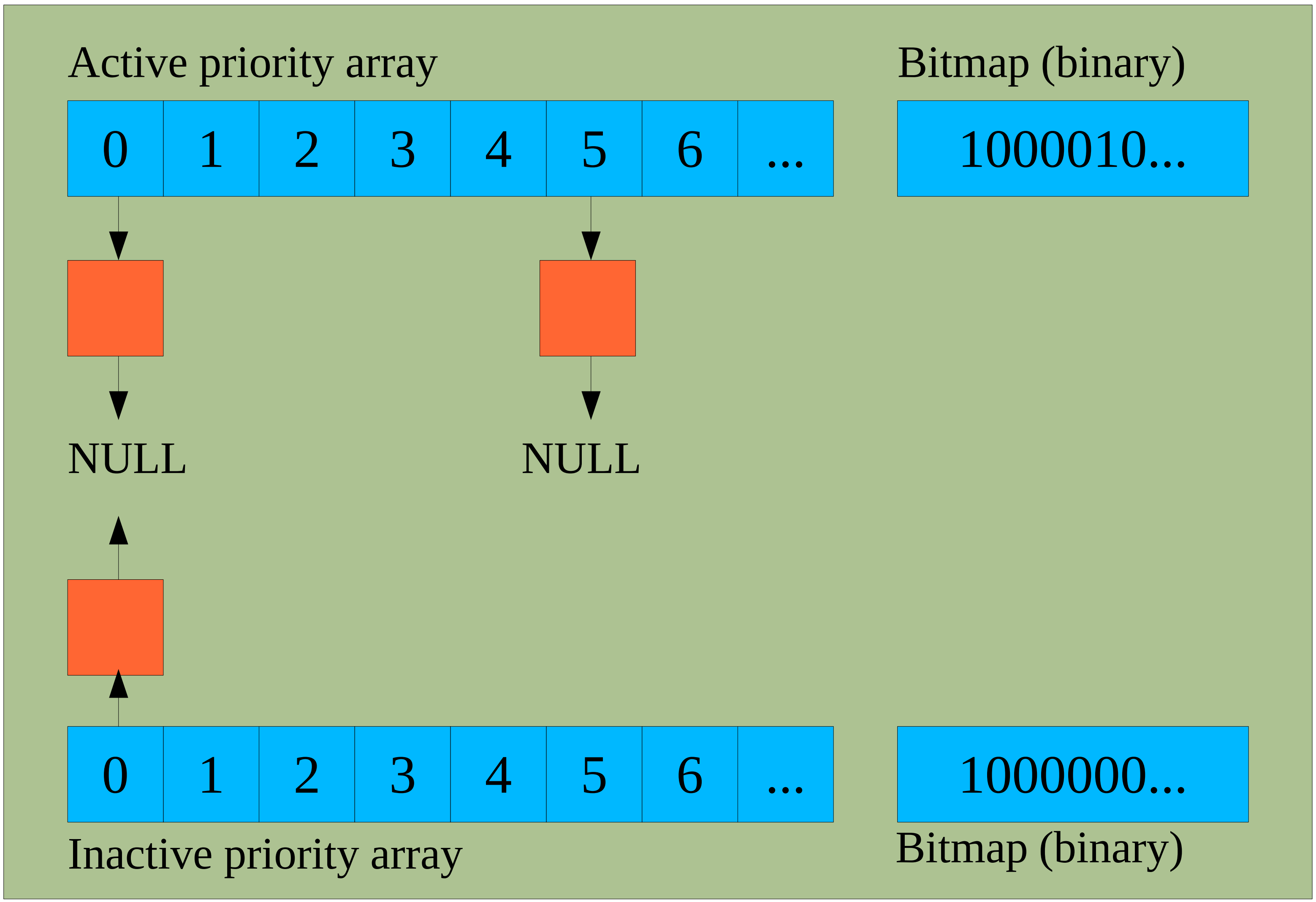
- Based on run queues and priority arrays
- Each CPU has an attached run queue
- Each run queue has two priority arrays, active and inactive
- The priority array has an entry for each priority
- Each entry in the priority array is a linked list
- A bitmap keeps track of empty and non-empty entries

The O(1) scheduler in 2.6, continued



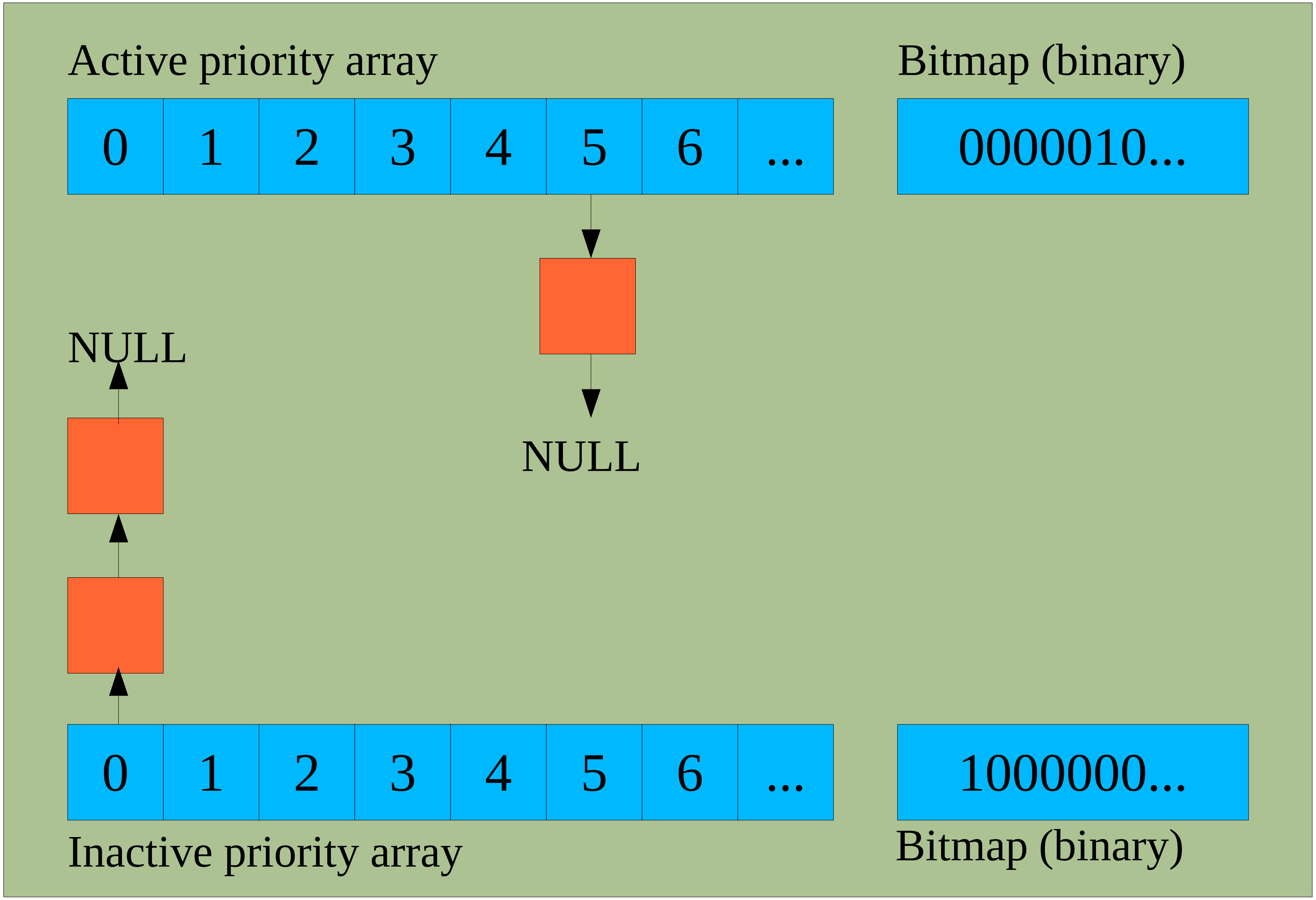
A run queue

The O(1) scheduler in 2.6, continued



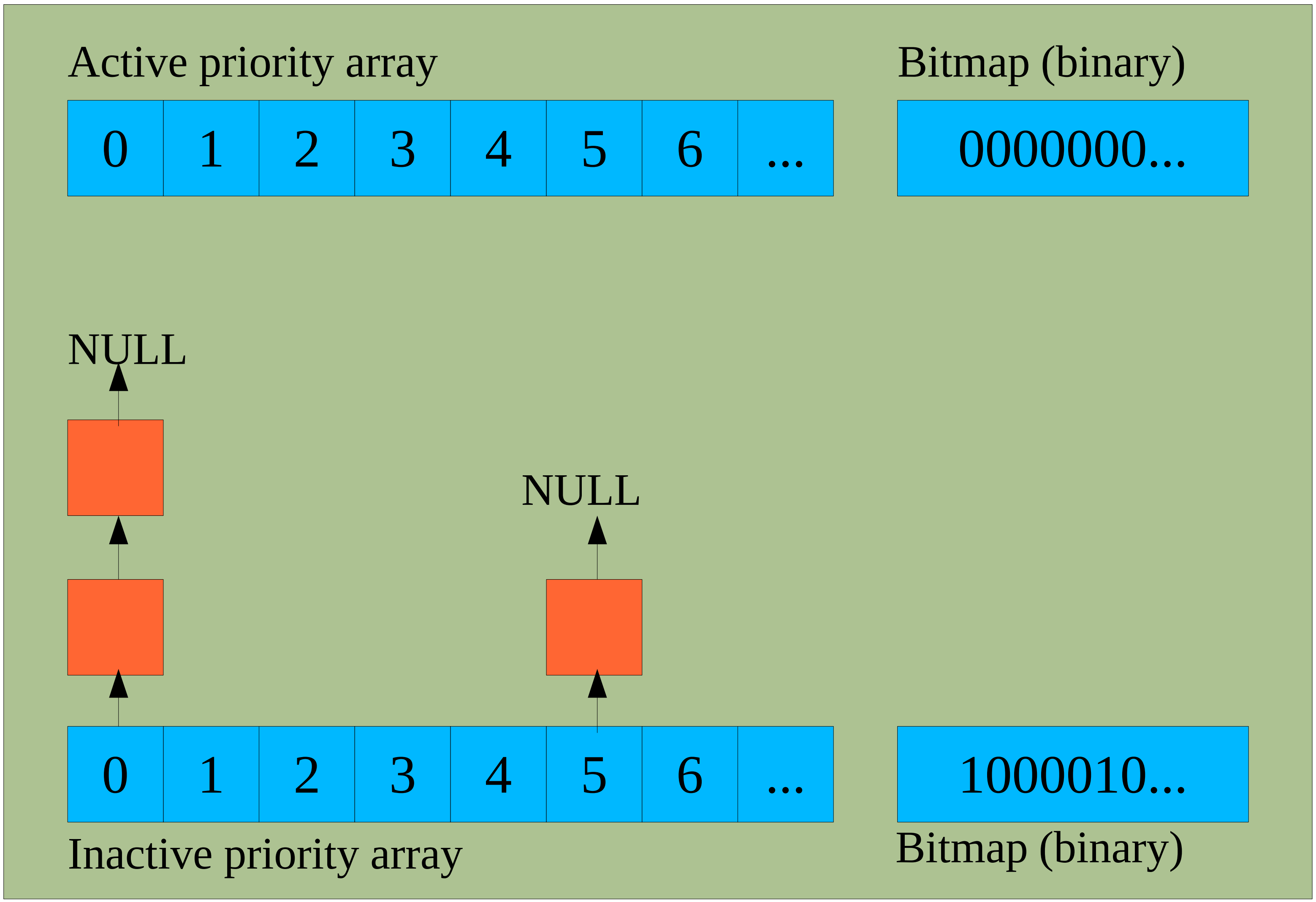
A run queue

The O(1) scheduler in 2.6, continued



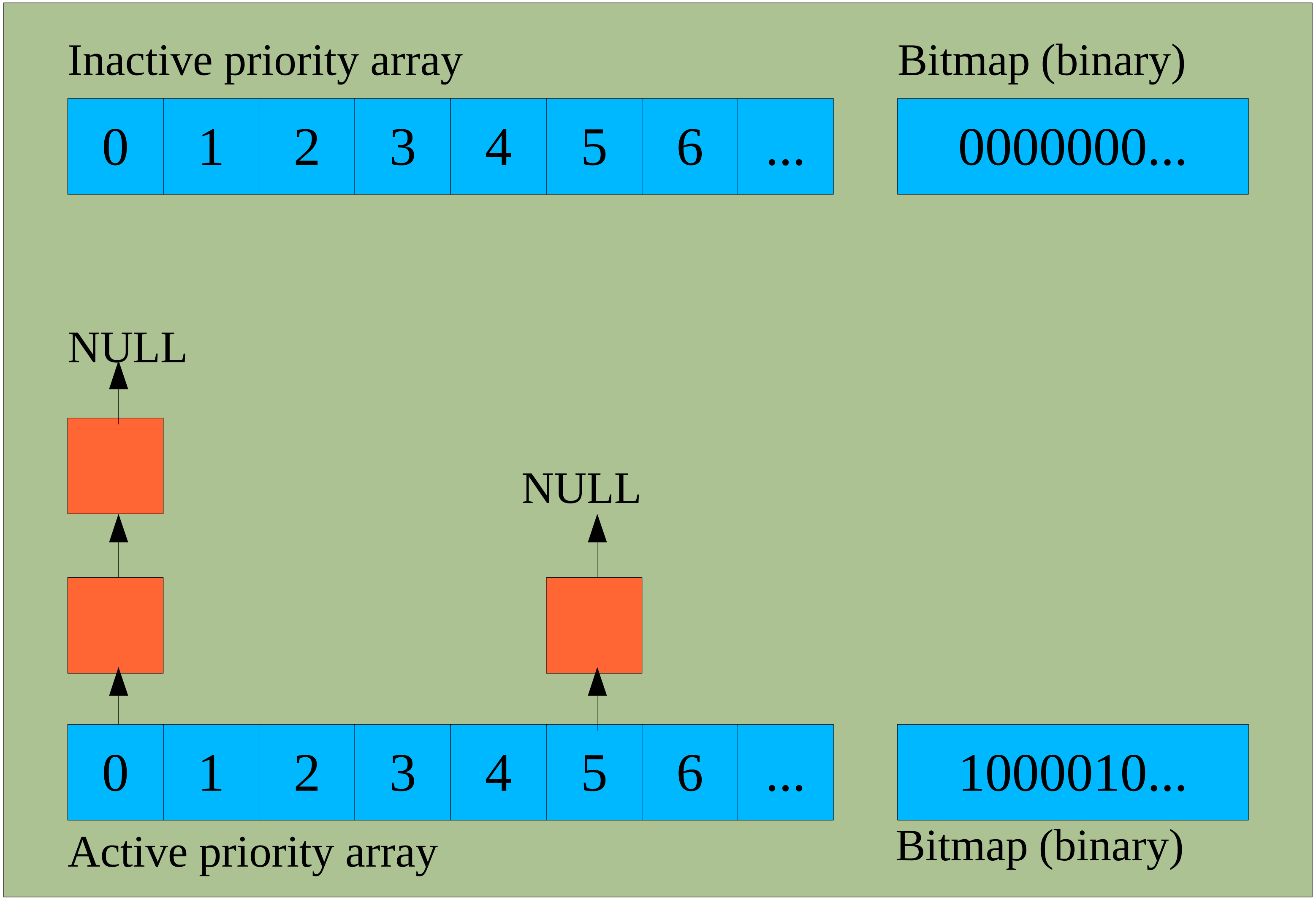
A run queue

The O(1) scheduler in 2.6, continued



A run queue

The O(1) scheduler in 2.6, continued



A run queue

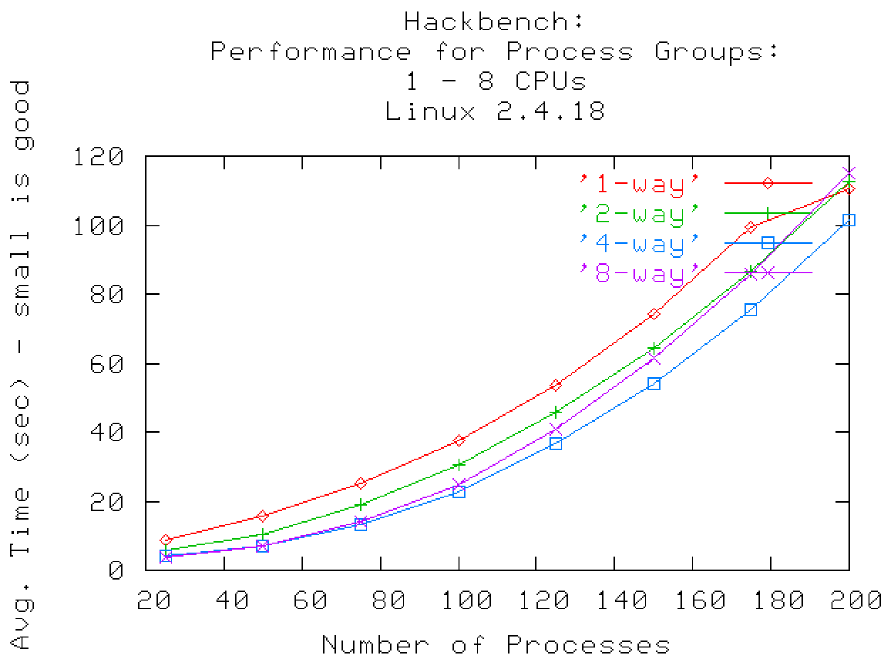
Activity #2

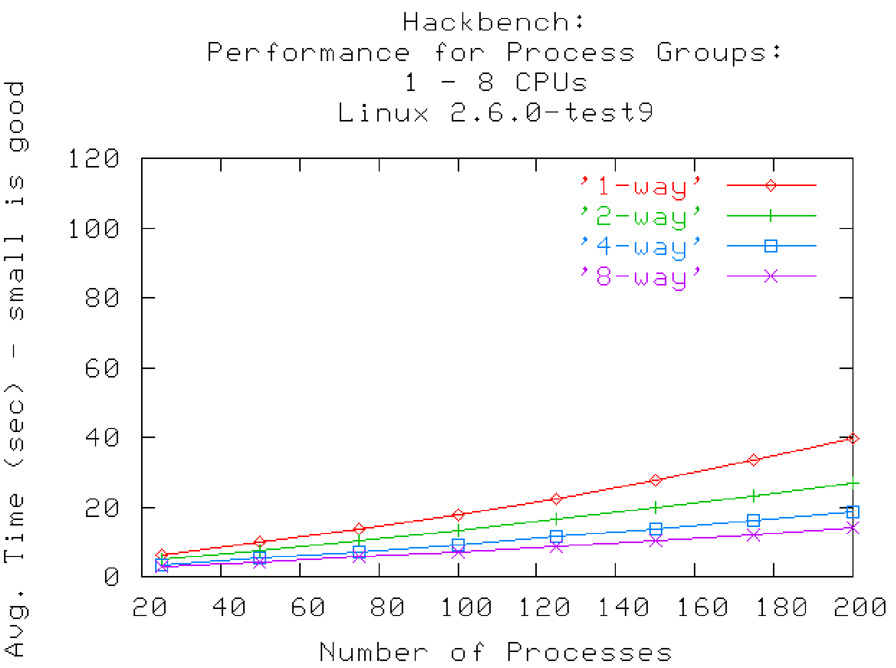
Why do we use a bitmap?

Activity #2

- To preserve the O(1) bounds
  - We can look up the index of the first set bit in O(1) time
  - We cannot search the priority array in O(1), only O(n)

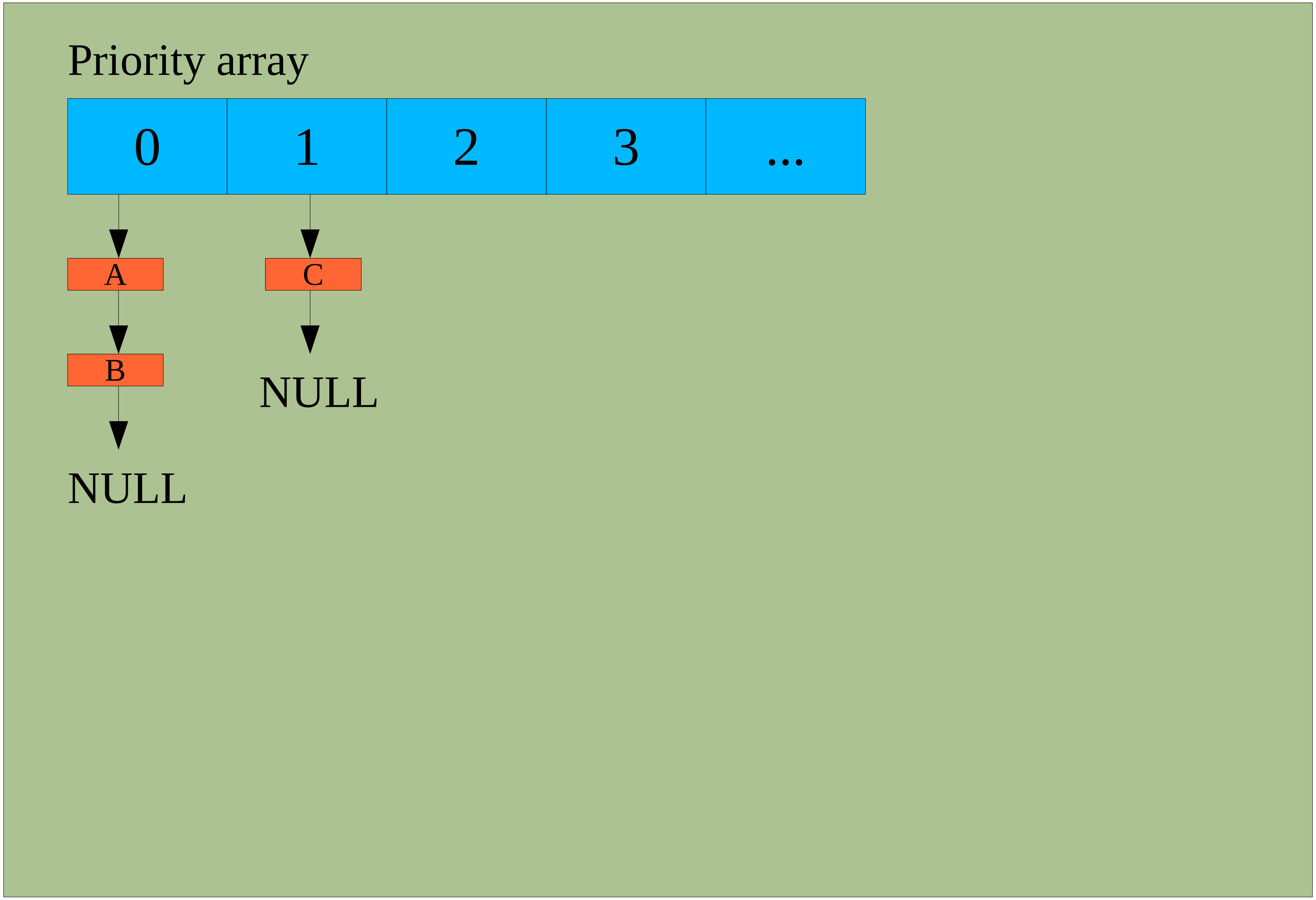
Performance 2.4 vs. 2.6





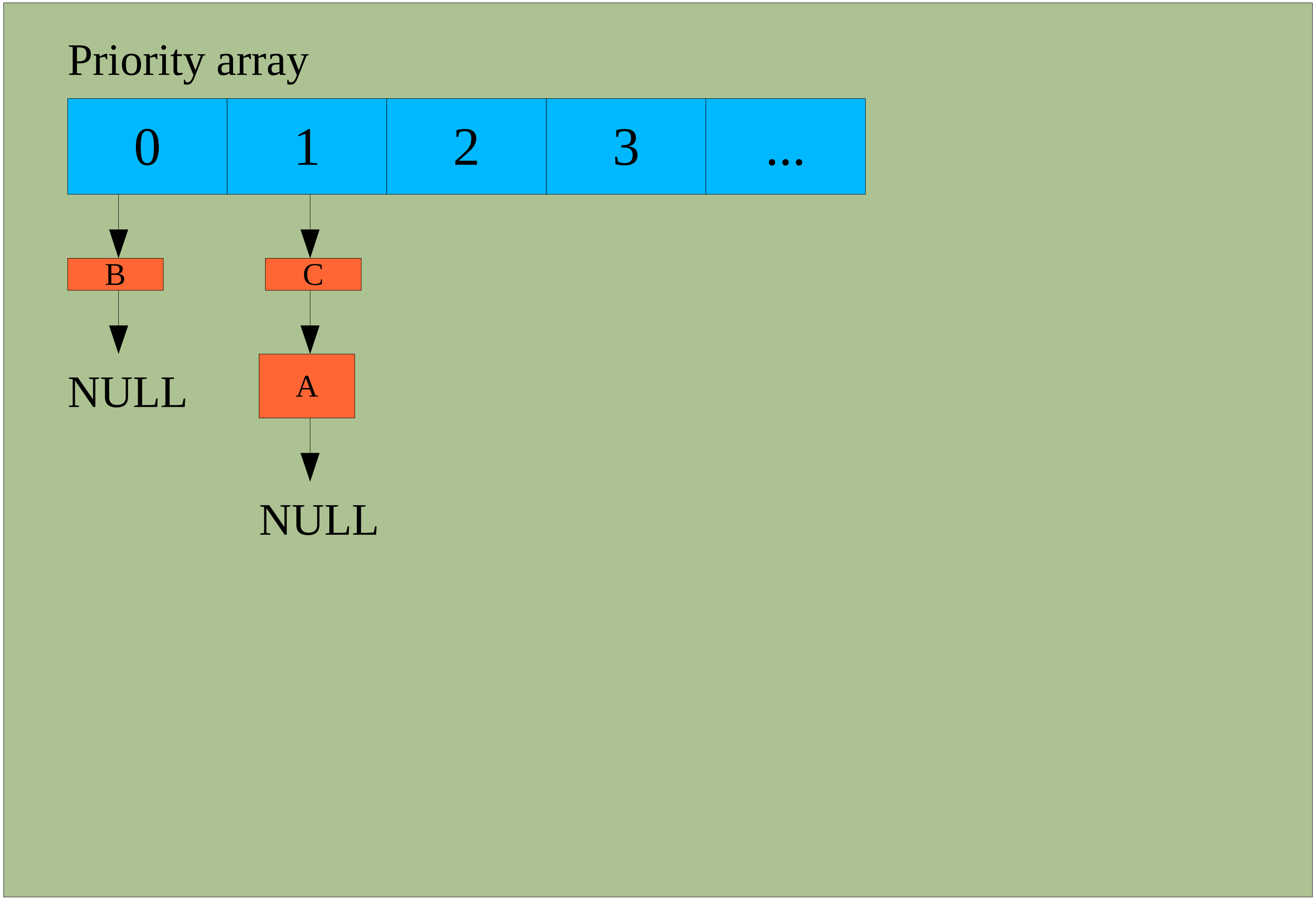
Images from http://devresources.linuxfoundation.org/craiger/hackbench/

The spiraling staircase scheduler



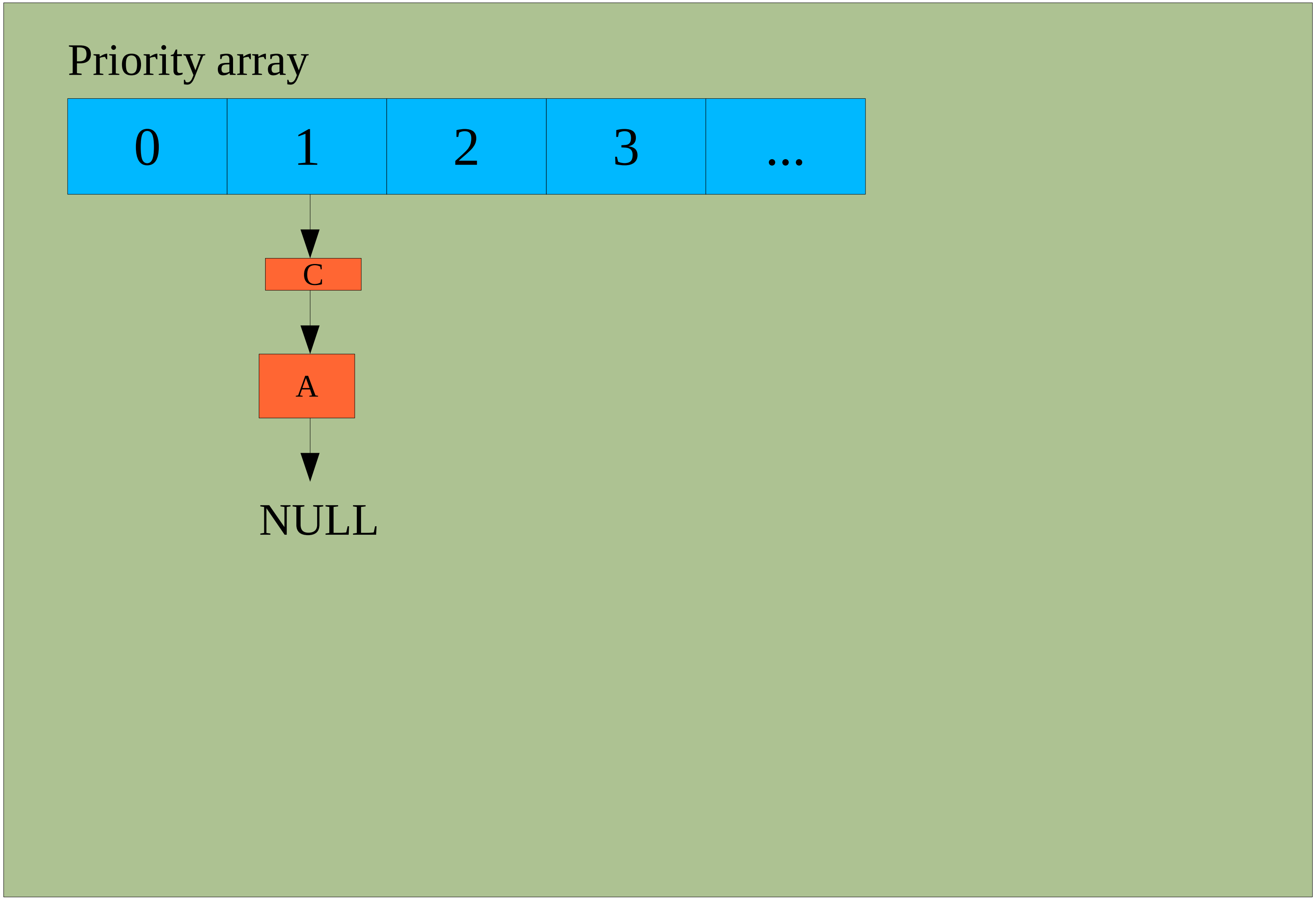
A run queue

The spiraling staircase scheduler



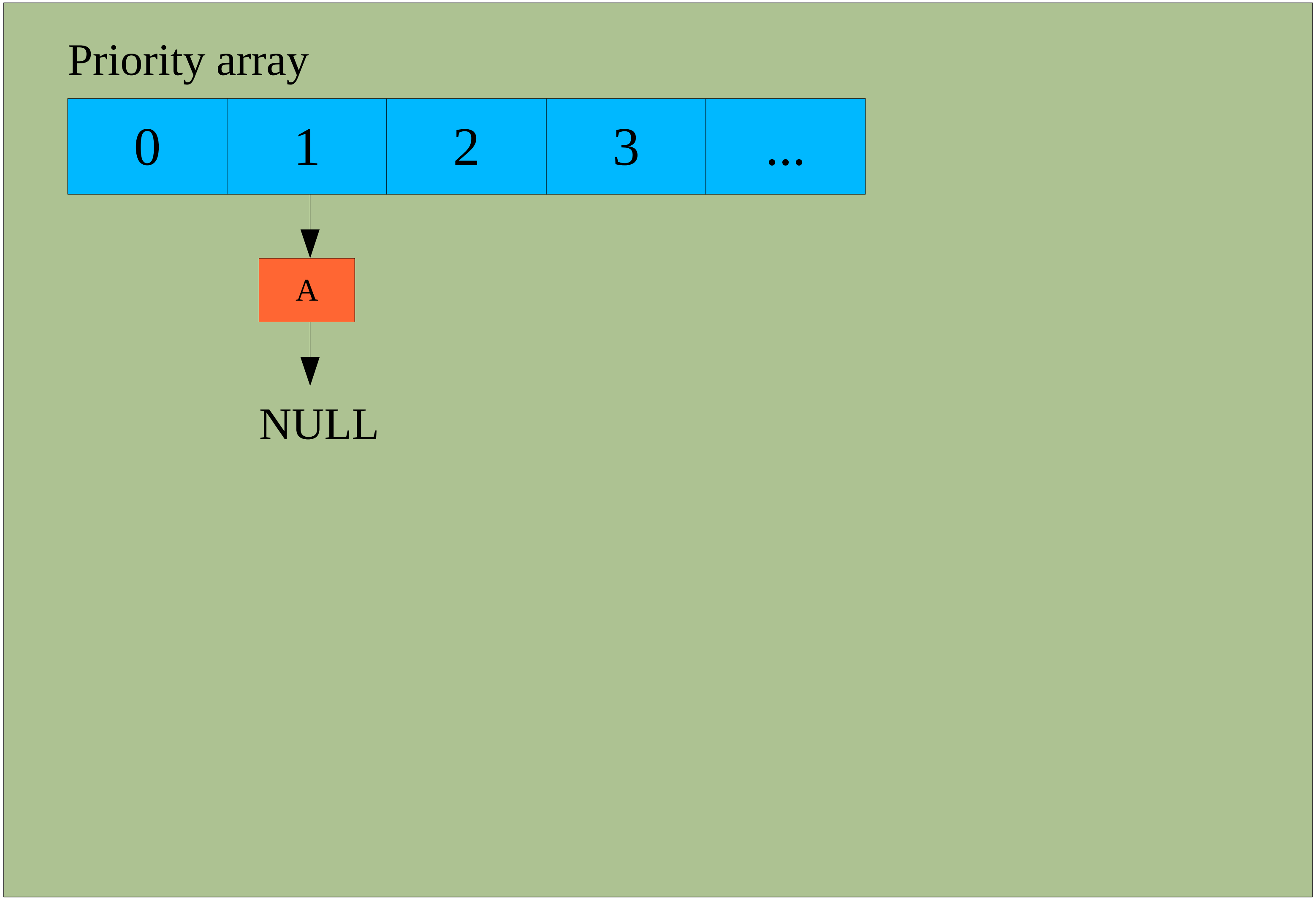
A run queue

The spiraling staircase scheduler



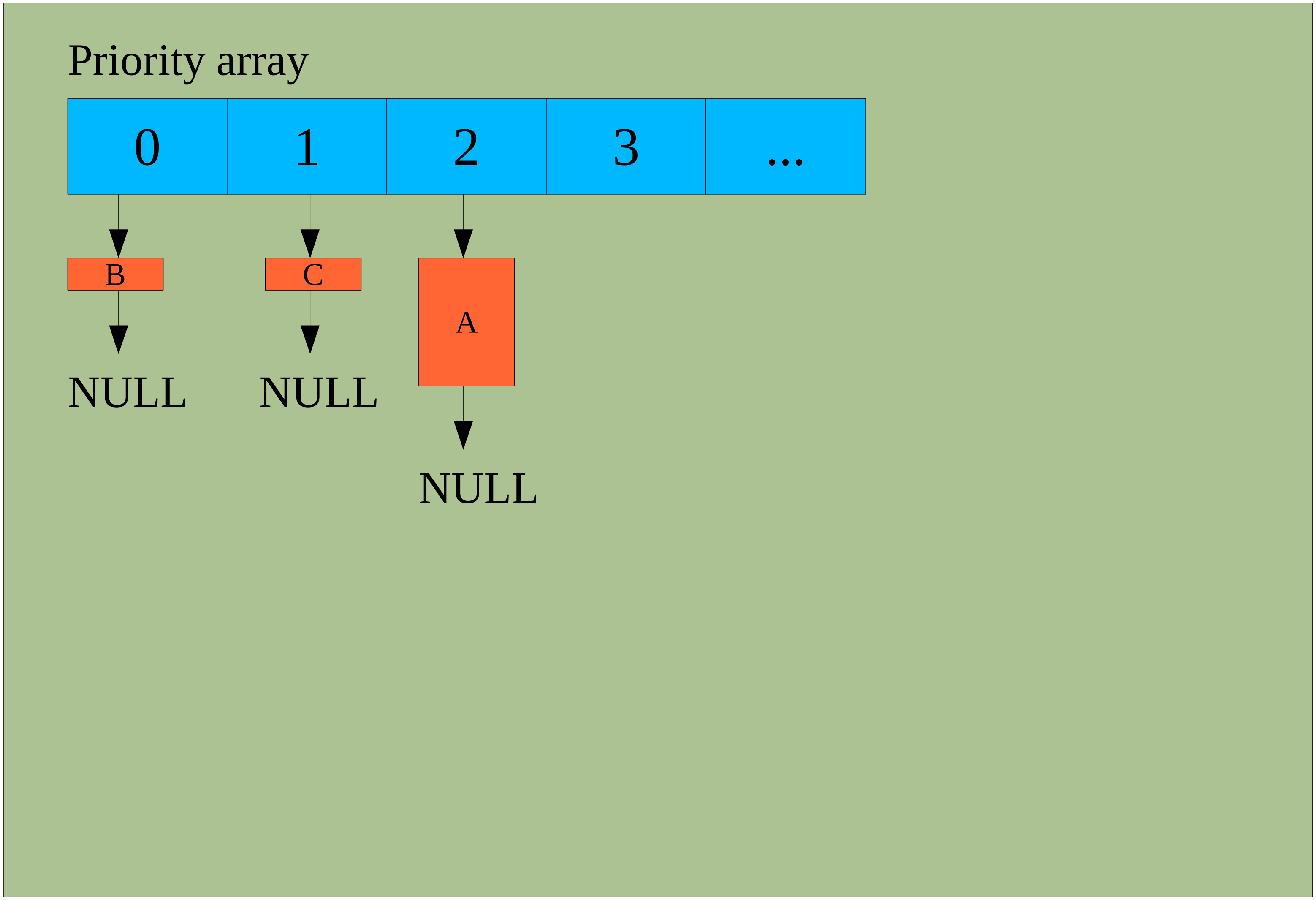
A run queue

The spiraling staircase scheduler



A run queue

The spiraling staircase scheduler



A run queue

The spiraling staircase scheduler

- Benefits:
  - Only a single array
  - Easier to understand
  - Seeks to improve fairness
  - Limits thread switches for CPU intensive applications

The Completely Fair Scheduler

- Basic idea is to emulate a perfect scheduler :)
- In a perfect scheduler all tasks get their fair share
- The CFS measures how far from the ideal a task is
- The task furthest away from the ideal is activated
- Uses a red/black tree to keep entries sorted
- Runs in O(log n), due to the red/black tree
- Uses only nanoseconds for task picking

The Completely Fair Scheduler

- Pick the rightmost node
- Run it for the allocated period
- Adjust the “unfair” count (wait_runtime)
- Re-insert into tree

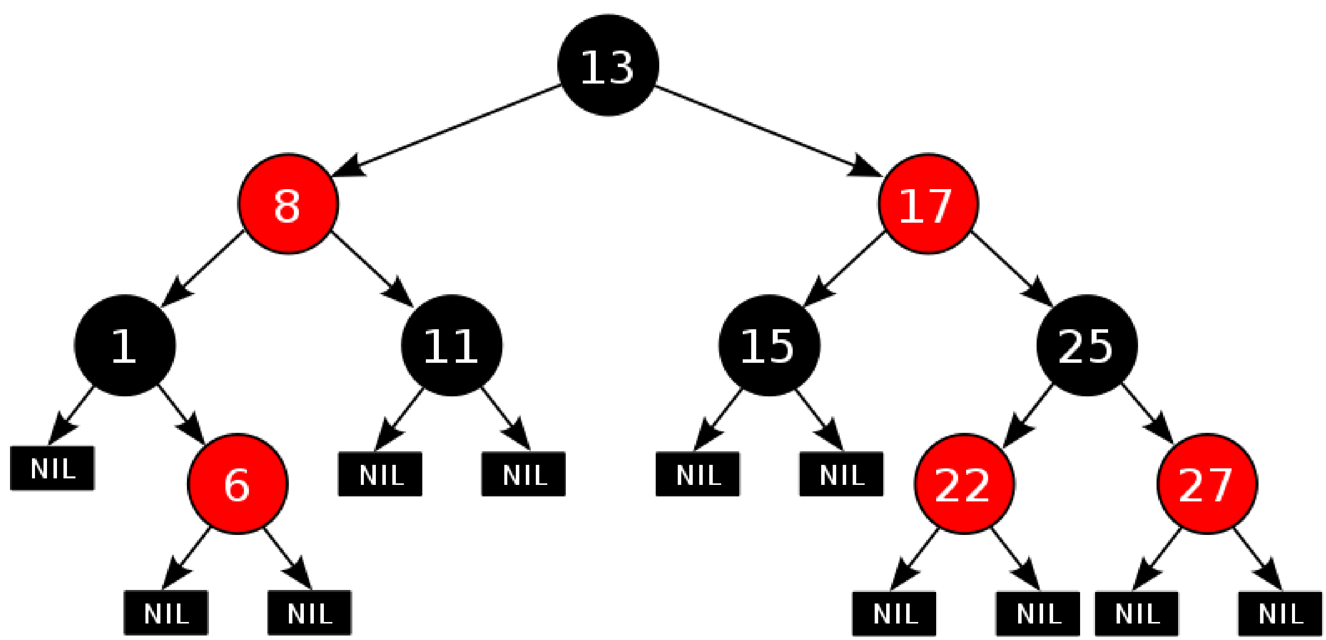


Image from wikipedia

Real-time schedulers

- If a task has a hard deadline, normal schedulers are not sufficient
- Most operating systems have a special priority setting called “Real-time”, which essentially bypasses scheduling and ensures that the thread will not get interrupted and always run before any other priority
- This is called a “soft real-time” scheduler

- Real-time and responsive are mutually exclusive goals
- A special OS is required for supporting “hard real-time”

Real-time schedulers

- For a real-time scheduler to work, it must know the maximum time a task needs to execute
- Real-time schedulers are used in many monitoring applications, where a late response can cause disasters
- The strategy is usually that there is a planning phase that lays out all tasks according to their requirements
- If it is not possible to schedule all tasks, the system will stop or activate an emergency procedure

Real-time schedulers

- RT schedulers can be preemptive which can increase responsiveness

- Common RT schedulers
  - Earliest deadline first (non-preemptive)
  - Rate-monotonic (aka shortest job first)
  - Fixed priority

Summary

- Scheduling is a hard problem
- Timeslicing enables a greedy algorithm
- There is no simple goal to optimize for
- The Linux kernel scheduler is always changing

Questions ?