Linux process scheduling

David Morgan

General “neediness” categories

- realtime processes
  - whenever they demand attention, need it immediately
- other processes
  - interactive – care about responsiveness
    - demand no attention most of the time, don’t need it
    - demand it occasionally, need it immediately then
  - batch – don’t care about responsiveness
    - demand attention frequently, don’t need it immediately

General strategies

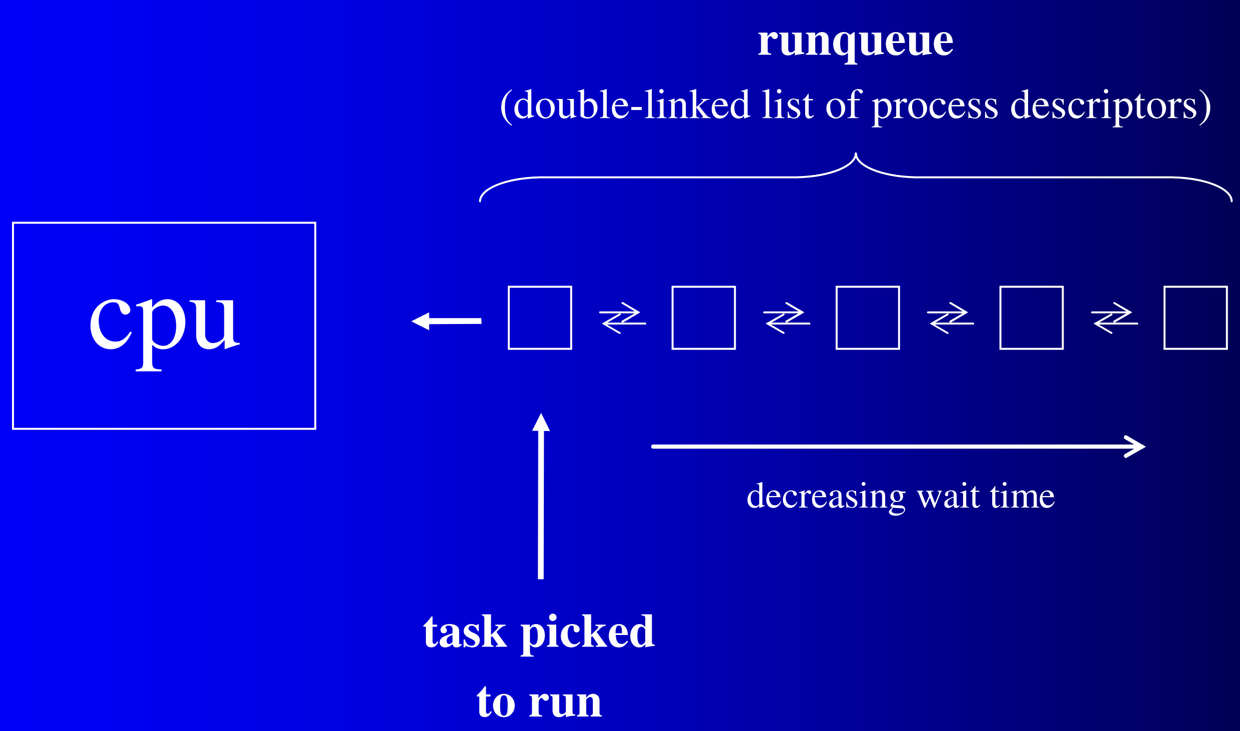
- favor all realtime processes ahead of all other processes

- favor interactive processes ahead of batch processes
  - by explicitly identifying and applying different formulas, or (pre-kernel-2.6.23 O(1) scheduler)
  - by applying a common formula (wait-time based) tending to float interactives and sinks batches (current kernel 2.6.23+ CFS scheduler)

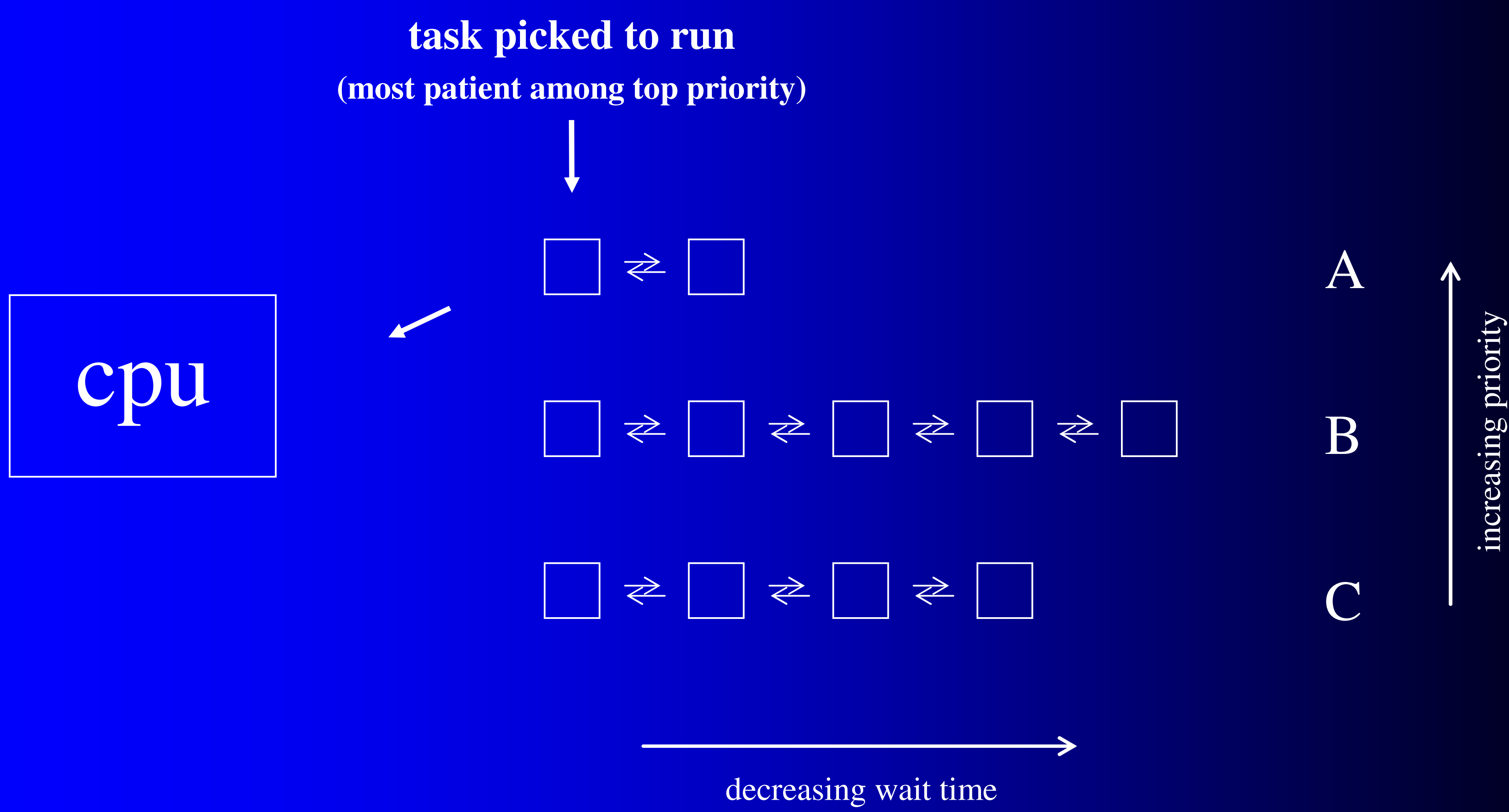
General scheduling basics

- multiple processes chosen to run for brief intervals one-after-the-other
- choice based on process “merit” or “deservedness”
- different possible “merit” characteristics
  - time a process has spent waiting (patience)
  - relative importance of a process (priority)
- linux considers several characteristics in combination
- always chooses the “most deserving” process

Patience may be meritorious



Priority may also be meritorious



Five “scheduling classes”

FOR REALTIME PROCESSES

SCHED_FIFO (first-in first-out)

SCHED_RR (round robin)

FOR ‘REGULAR’ PROCESSES

SCHED_NORMAL a.k.a SCHED_OTHER

SCHED_BATCH

SCHED_IDLE

Different schedulers

The “realtime scheduler”

SCHED_FIFO (first-in first-out)

SCHED_RR (round robin)

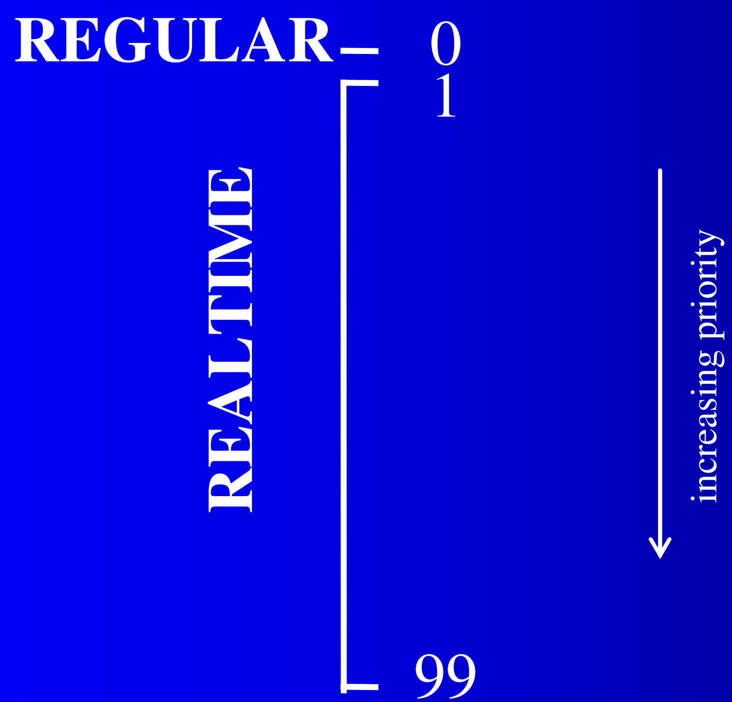
The “completely fair scheduler (CFS)”

SCHED_NORMAL

SCHED_BATCH

SCHED_IDLE

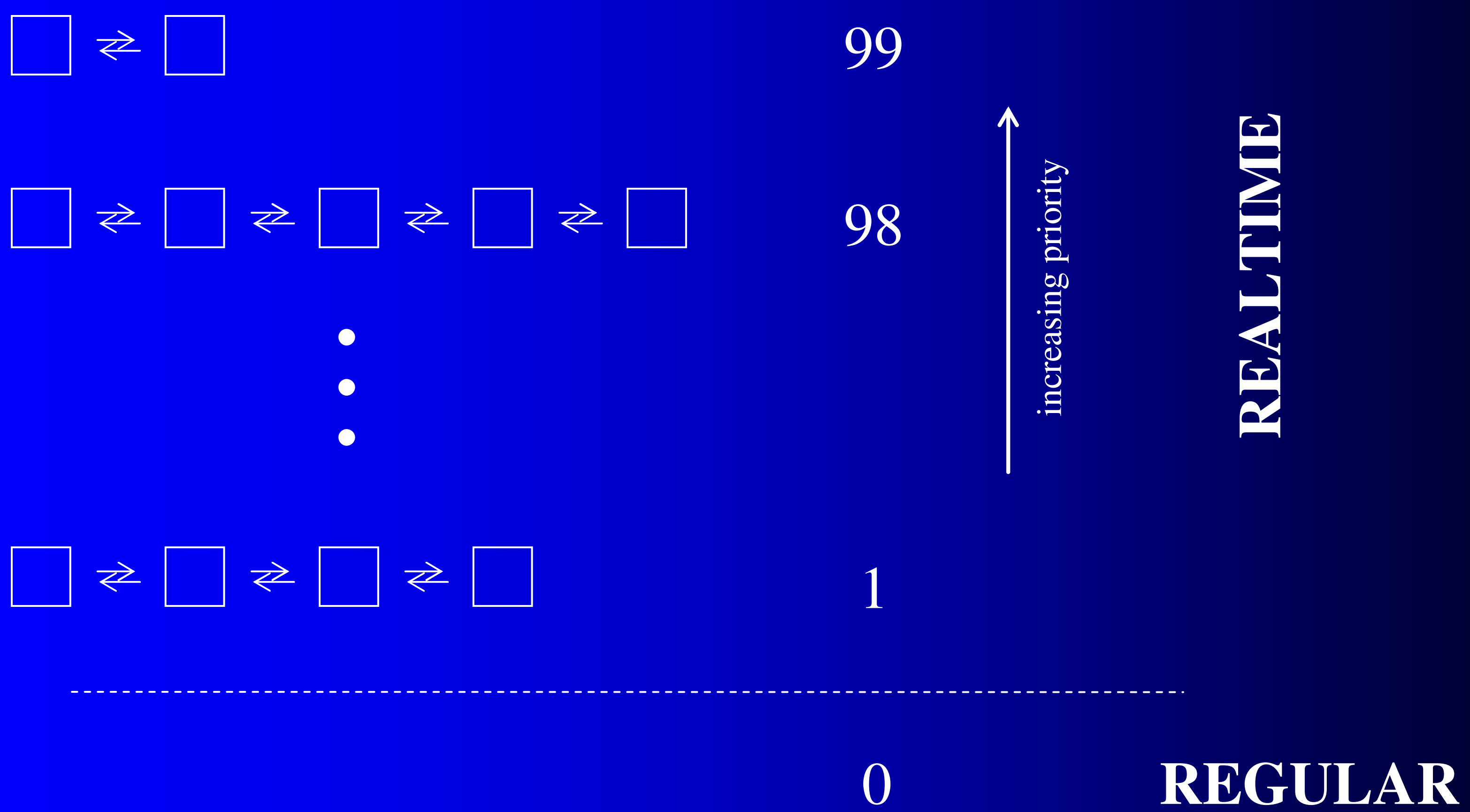
Priority scale



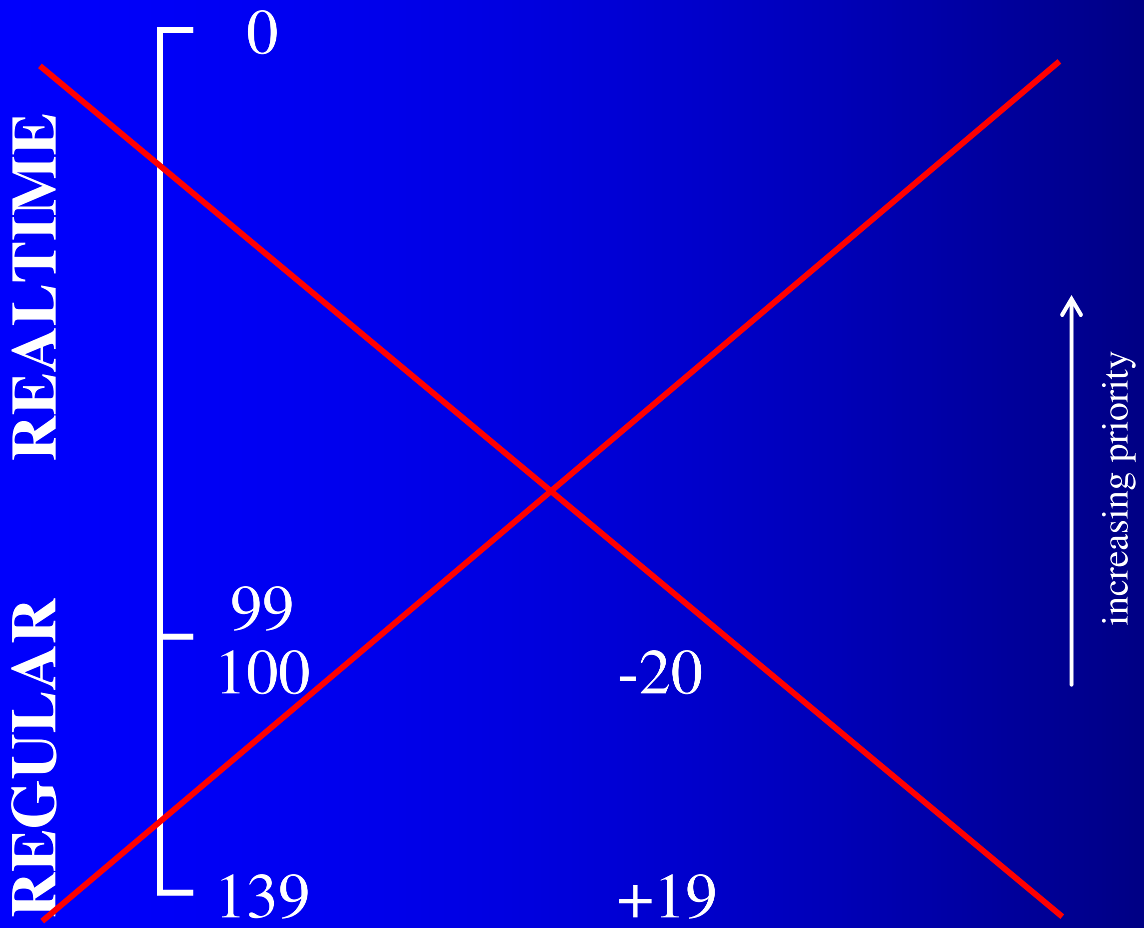
Realtime requirements

- low latency
- deterministic response time
- settings
  - financial trading
  - medical devices
  - defense
  - industrial automation
  - autonomous (self-driving) cars

Realtime trumps regular

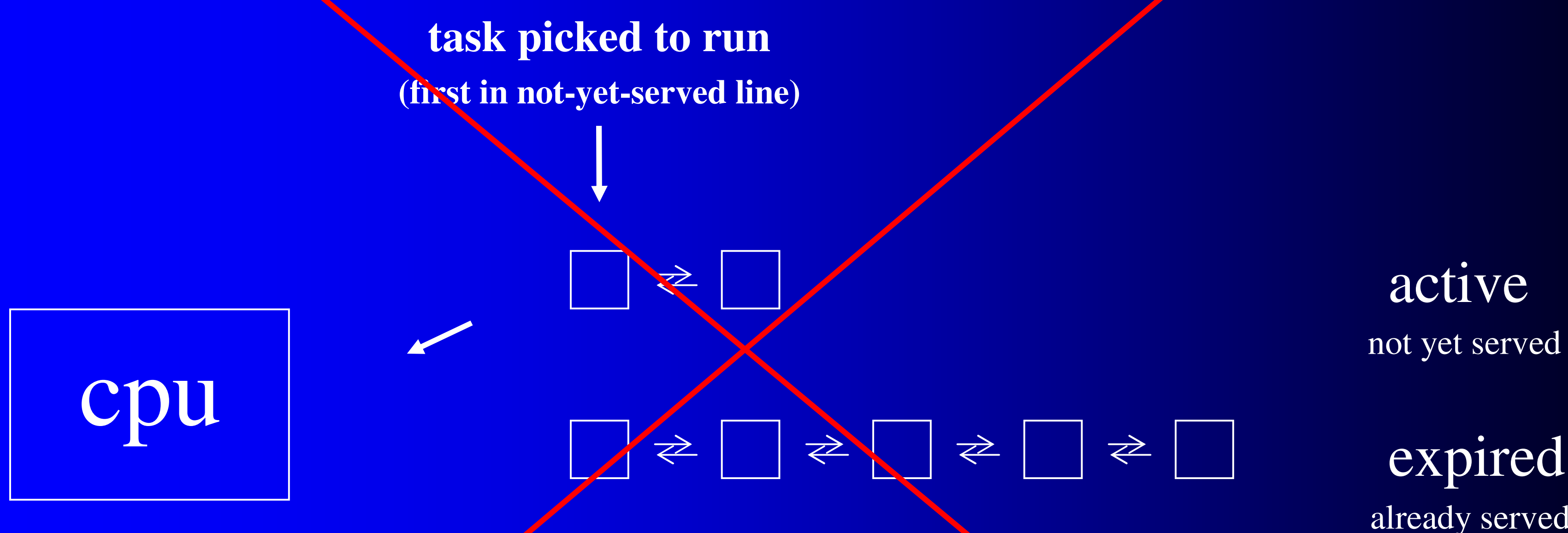


Former priority scale*



*reflected in literature

Former priority implementation



“Each process is given a fixed time quantum, after which it is preempted and moved to the expired array. Once all the tasks from the active array have exhausted their time quantum and have been moved to the expired array, while the expired array becomes the active array.”

Input to scheduling decisions

SCHED_SETSCHEDULER(2) Linux Programmer's Manual SCHED_SETSCHEDULER(2)

NAME

sched_setscheduler

"policy" here

SYNOPSIS

```
int sched_setscheduler(   pid_t pid,    int policy,    const struct sched_param *param   );

struct sched_param {      ...
    int sched_priority;   ...
};
```

"priority" here

DESCRIPTION

Scheduling Policies

The scheduler is the kernel component that decides which runnable process will be executed by the CPU next. Each process has an associated scheduling policy and a static scheduling priority, sched_priority; these are the settings that are modified by sched_setscheduler(). The scheduler makes it decisions based on knowledge of the scheduling policy and static priority of all processes on the system.

(see sched_setscheduler.man.abridged.txt)

Input to scheduling decisions

DESCRIPTION

Currently, Linux supports the following "normal" (i.e., non-real-time) scheduling policies:

SCHED_OTHER the standard round-robin time-sharing policy;
SCHED_BATCH for "batch" style execution of processes; and
SCHED_IDLE for running very low priority background jobs.

The following "real-time" policies are also supported, for special time-critical applications that need precise control over the way in which runnable processes are selected for execution:

SCHED_FIFO a first-in, first-out policy; and
SCHED_RR a round-robin policy.

Scheduling Policies

SCHED_FIFO: First In-First Out scheduling

SCHED_FIFO can only be used with static priorities higher than 0, which means that when a SCHED_FIFO processes becomes runnable, it will always immediately preempt any currently running SCHED_OTHER, SCHED_BATCH, or SCHED_IDLE process.

SCHED_OTHER: Default Linux time-sharing scheduling

SCHED_OTHER can only be used at static priority 0. SCHED_OTHER is the standard Linux time-sharing scheduler that is intended for all processes that do not require the special real-time mechanisms.

Scheduling class implementation

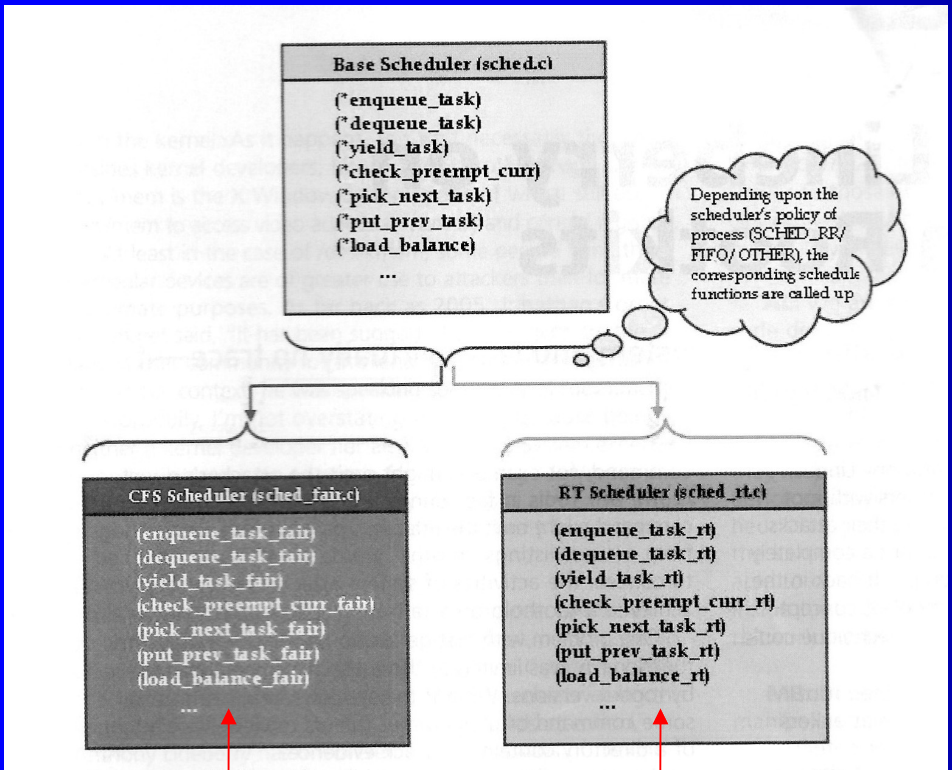


Figure 3. Modular Scheduler

Resembles object-oriented class hierarchy

Correct handler selected per scheduling class of each particular process

Extensible, for implementing future scheduling classes with new scheduling algorithms

for SCHED_OTHER SCHED_BATCH SCHED_IDLE

(the "normal" classes)

for SCHED_FIFO SCHED_RR

(the "realtime" classes)

"Completely Fair Scheduler," Linux Journal, August 2009

Two demo programs (heavy loops)

root@frausto:~/priority

```
[root@frausto priority]#
[root@frausto priority]# diff process_priority.c process_priority-realtime-version.c
47,48c47,48
<         sp.sched_priority=0;
<         sched_setscheduler(0,SCHED_OTHER,&sp);
---
>         sp.sched_priority=50;
>         sched_setscheduler(0,SCHED_FIFO,&sp);
[root@frausto priority]#
```

non-realtime

realtime

DESCRIPTION

Currently, Linux supports the following "normal" (i.e., non-real-time) scheduling policies:

SCHED_OTHER the standard round-robin time-sharing policy;
SCHED_BATCH for "batch" style execution of processes; and
SCHED_IDLE for running very low priority background jobs.

The following "real-time" policies are also supported, for special time-critical applications that need precise control over the way in which runnable processes are selected for execution:

SCHED_FIFO a first-in, first-out policy; and
SCHED_RR a round-robin policy.

Scheduling Policies

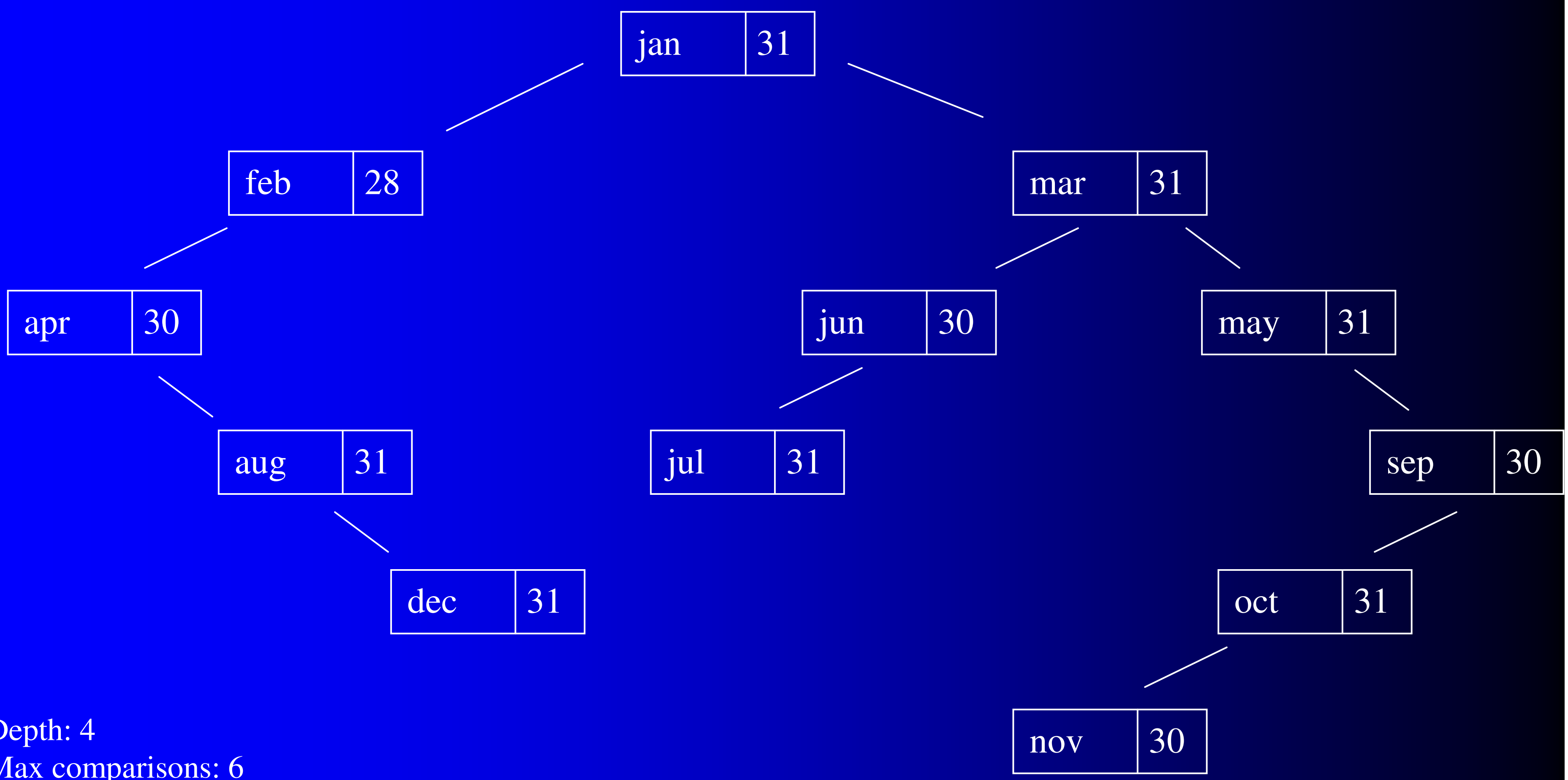
SCHED_FIFO: First In-First Out scheduling
SCHED_FIFO can only be used with static priorities higher than 0, which means that when a SCHED_FIFO processe becomes runnable, it will always immediately preempt any currently running SCHED_OTHER, SCHED_BATCH, or SCHED_IDLE process.

SCHED_OTHER: Default Linux time-sharing scheduling
SCHED_OTHER can only be used at static priority 0. SCHED_OTHER is the standard Linux time-sharing scheduler that is intended for all processes that do not require the special real-time mechanisms.

Binary trees

- elements have up to 2 child elements
- left child sorts less, right more, than parent
- tree has a depth
- tree has a balance, comparing depths of its left and right trees (greater difference, less balance)

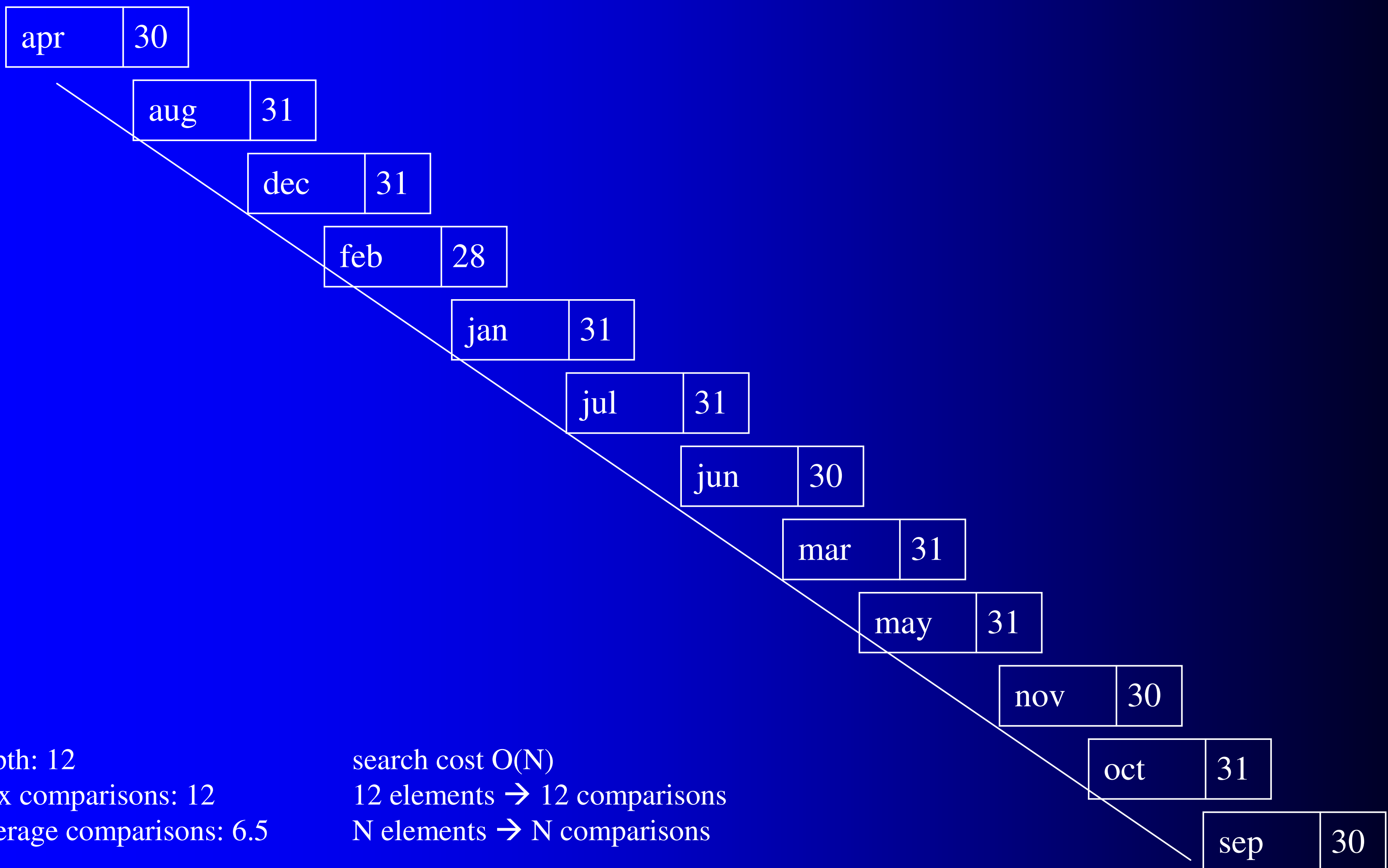
Binary tree of months, for days-per-month determination



Depth: 4
Max comparisons: 6
Average comparisons: 3.5

input sequence: jan, feb, mar, apr, may, june, july, aug, sept, oct, nov, dec (chronological)

A skewed tree

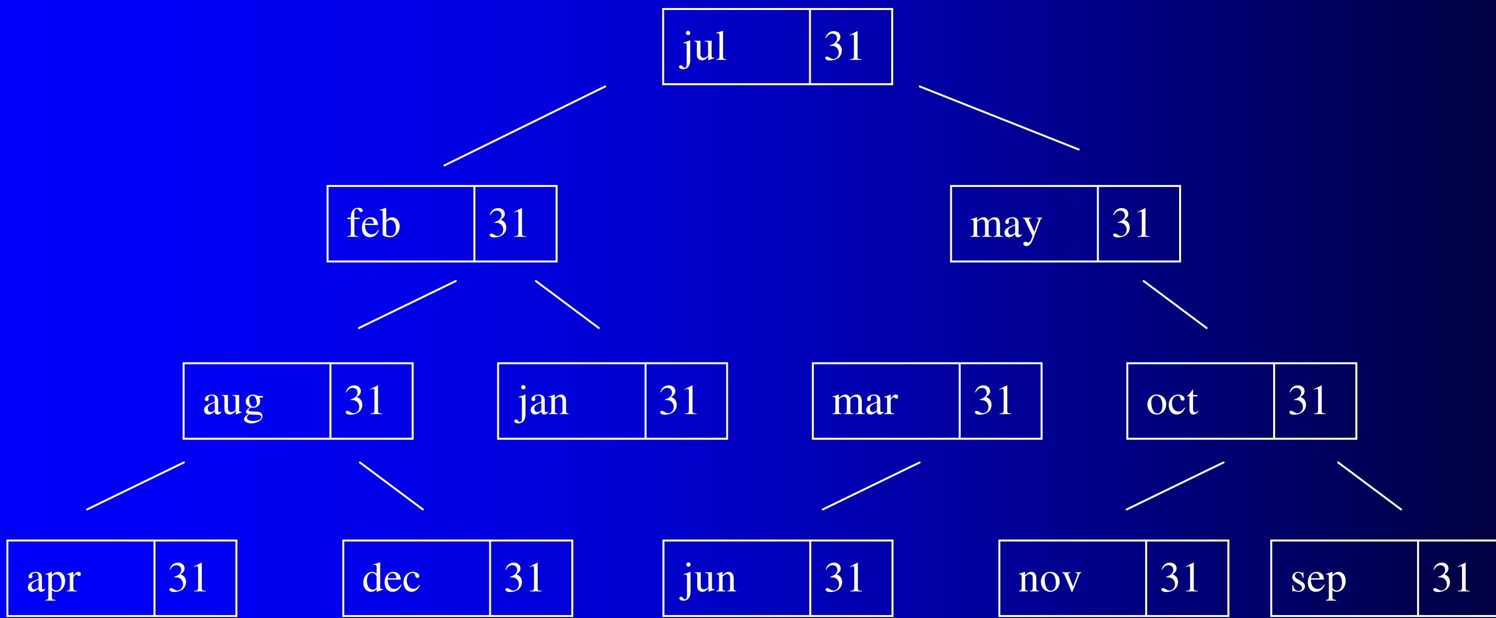


Depth: 12
Max comparisons: 12
Average comparisons: 6.5

search cost O(N)
12 elements → 12 comparisons
N elements → N comparisons

input sequence: apr, aug, dec, feb, jan, july, june, mar, may, nov, oct, sept (alphabetical)

A balanced tree



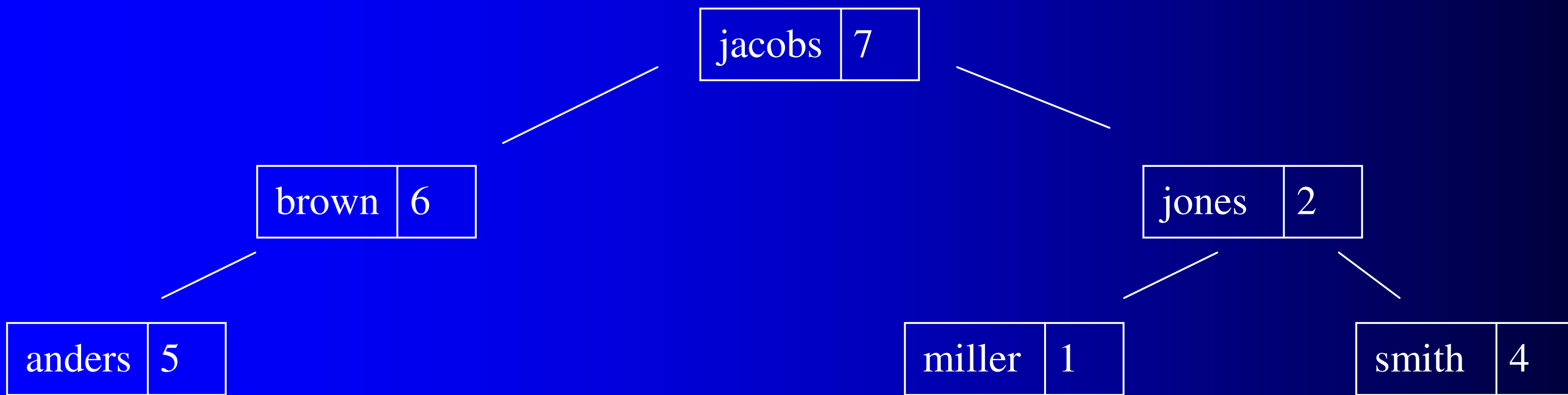
search cost O(log N)
2 levels → 3 elements → 2 comparisons
3 levels → 7 elements → 3 comparisons
4 levels → 15 elements → 4 comparisons

L levels → 2^L-1 elements → L comparisons, or
log(N+1) levels → N elements → log(N+1) comparisons

Depth: 4
Max comparisons: 4
Average comparisons: 3.1

input sequence: july, feb, may, aug, dec, mar, oct, apr, jan, june, sept, nov

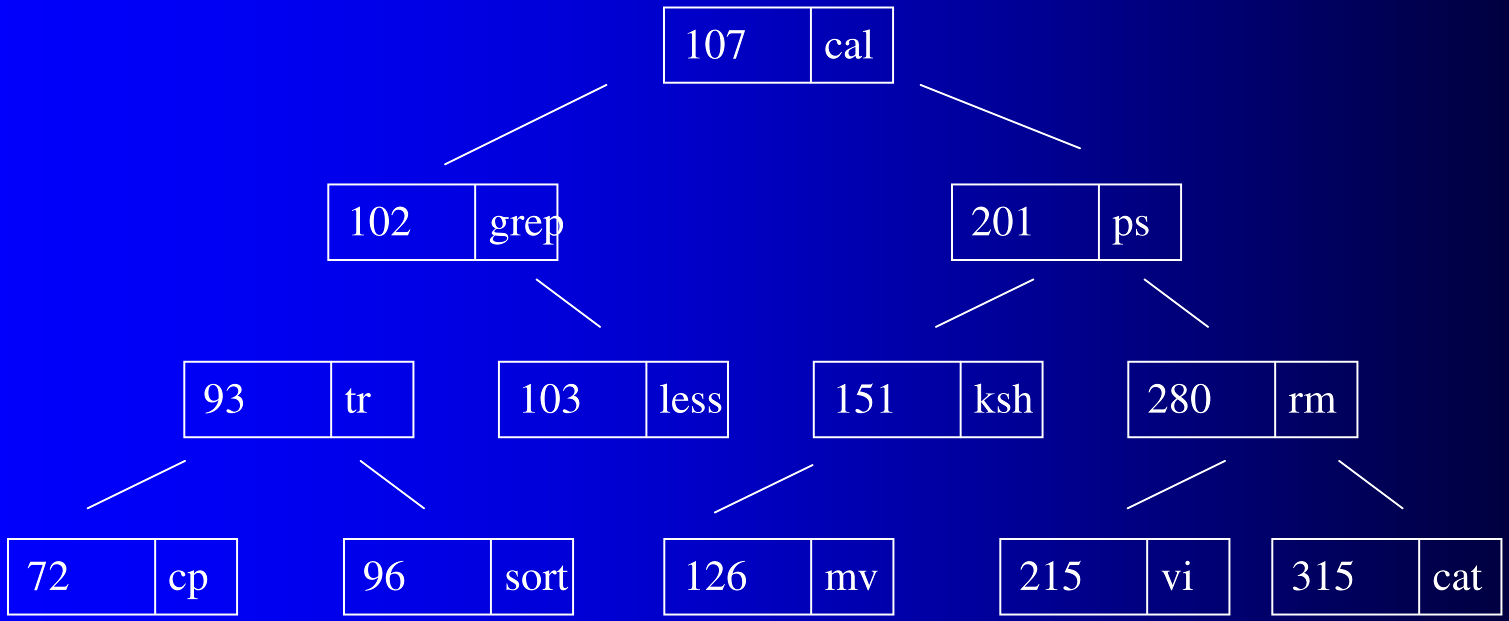
Binary tree of last names, for data record determination



Database

Recno	name	rank	serial no
1	miller	corporal	4-139
2	jones	major	3-209
3	baker	private	7-981
4	smith	lieutenant	3-101
5	anders	private	8-388
6	brown	sargeant	8-231
7	jacobs	captain	6-495
8	johnson	general	4-556

Binary tree of number metrics, for process determination



Numbers developed to reflect variance between ideal and actual CPU utilization for each process
Smallest number → greatest variance (most "underserved")

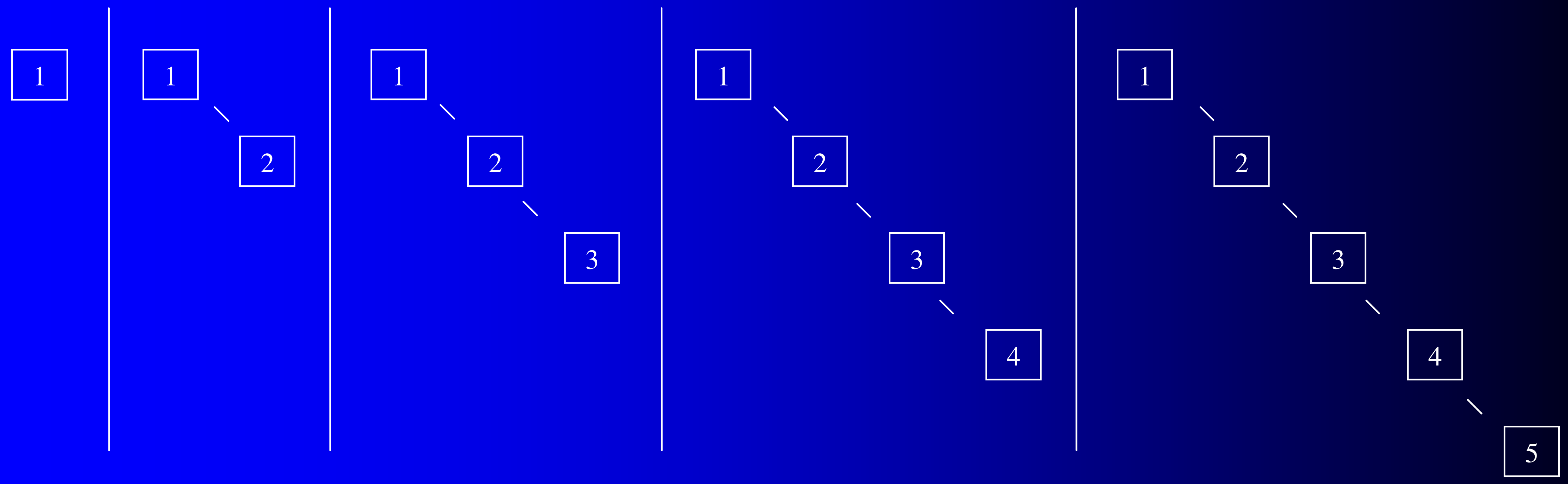
Smallest gets CPU. While it runs its metric rises while the others' all fall till one of them undercuts, then it becomes the new running process

Tree balance

- depends on insertion sequence
- balance achievable independent of sequence, by performing mid-course re-balancing
  - during insertion, whenever an insertion upsets the balance, re-balance dynamically before inserting next element
  - tree never gets unbalanced, so final result is always balanced

Building tree, no rebalancing

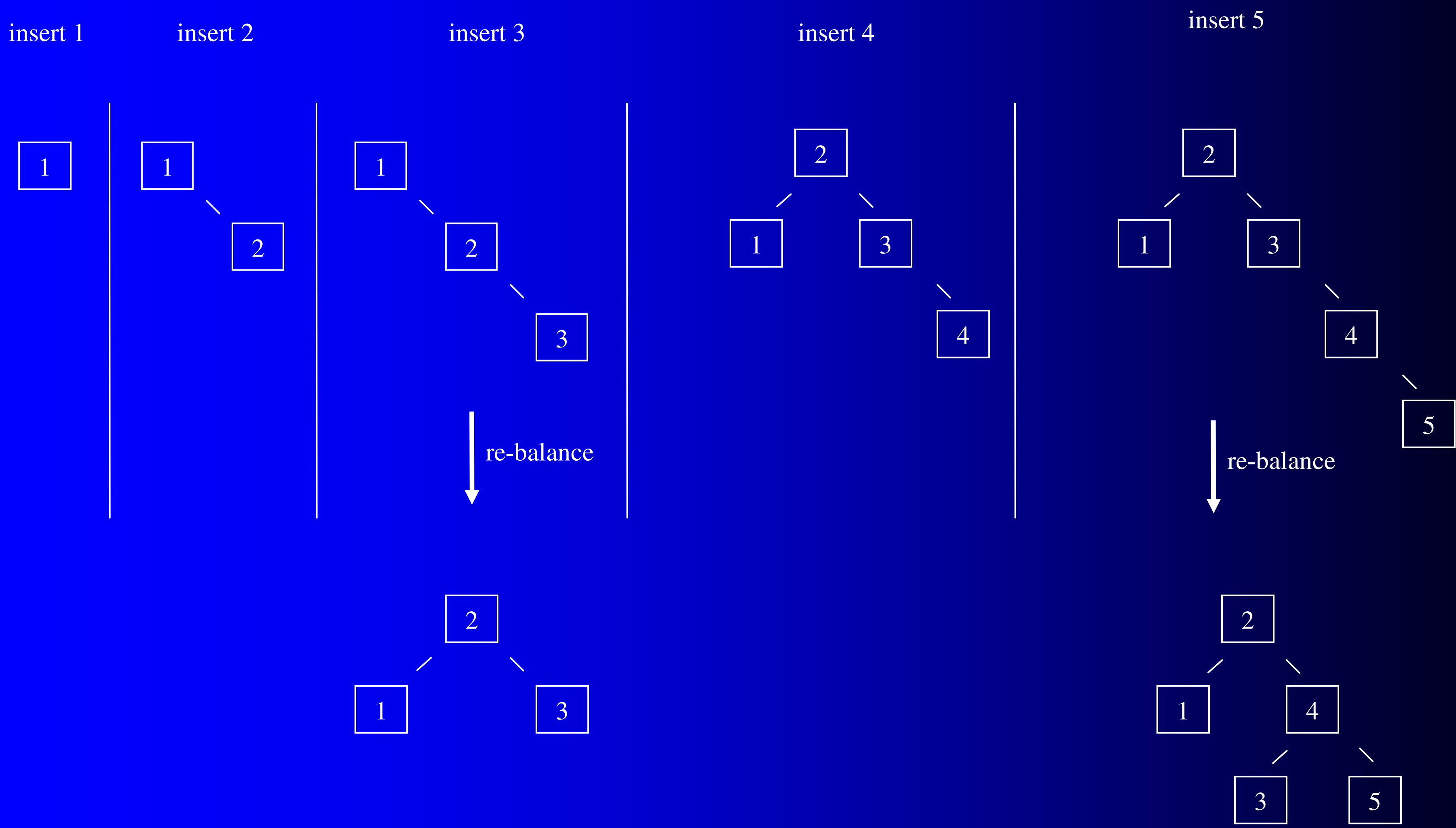
insert 1 insert 2 insert 3 insert 4 insert 5



input sequence: 1, 2, 3, 4, 5

final tree unbalanced

Building tree, mid-course re-balancing



input sequence: 1, 2, 3, 4, 5

final tree balanced

/proc/sched_debug

More information

- scheduler author Ingo Molnar
  - http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt
- "Multiprocessing with the Completely Fair Scheduler"
  - http://www.ibm.com/developerworks/linux/library/l-cfs/index.html