Lecture 7

All about processes

Outline

- Processes and threads
- Representing and managing tasks (Linux)
- Process creation
- Context switching
- Digression: keeping time
- Process scheduling

Process, threads, tasks

- A process is an instance of a program in execution
  - It is also the entity to which the OS allocate resources to
- A process can have multiple threads of control
  - Main different: threads share the same virtual address space among other OS resources, processes do not
  - Threads may be user level
- Modern Linux do not draw a clear line between processes and threads – they are all called tasks
  - Process is a task with a single thread
  - A process can have several tasks
  - Tasks are scheduling entities in Linux
  - Sometimes referred to as lightweight processes

Pthreads

- POSIX threads – the standard for multithreaded programming
- Many programming languages and systems have a notion of threading. Not to be confused with OS threads.

Normal vs Real-time Process

- Real-time processes have deadlines to meet
  - Example: must run for *x* microseconds every *y* millisecond
  - Translates to scheduling constraints
- Normal (regular) process has no such requirements
- To fully support real-time tasks, a real-time OS (RTOS) is required
  - Eg., WindRiver, RTLinux
- Mainstream OS like Linux supports some real-time processes but not completely a RTOS

Information about processes

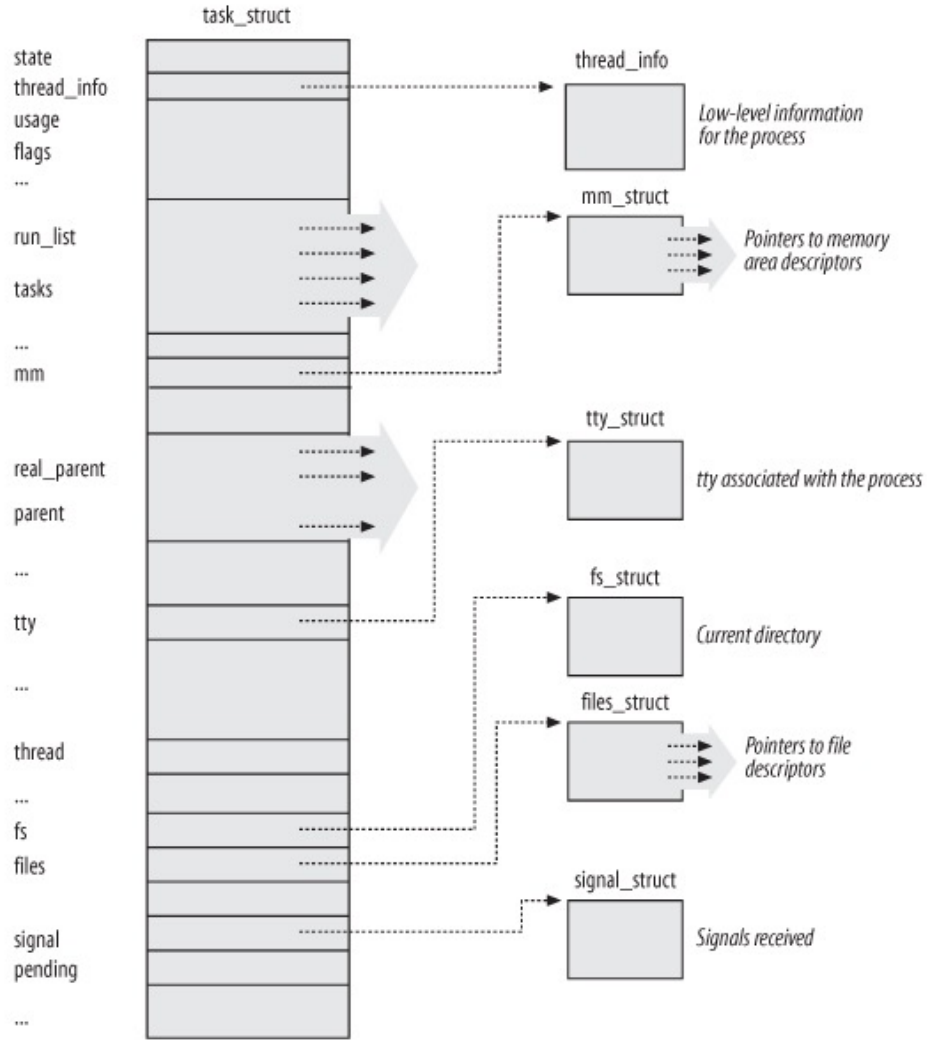
- Depends on kernel version
- In 5.16.1, a task is represented by:
  - **`task_struct`**
    - Traditional called the process control block (PCB)
    - Hold all information about a process/task
  - **`thread_info`**
    - Co-located with the per thread kernel stack
    - Used to hold the architecture dependent info
    - Now holds status flags that are used often
  - **`thread_struct`**
    - Located at the end of task_struct
    - Hold architecture dependent info such as register states etc.

/ include / linux / sched.h

```
1868
1869  union thread_union {
1870  #ifndef CONFIG_ARCH_TASK_STRUCT_ON_STACK
1871          struct task_struct task;
1872  #endif
1873  #ifndef CONFIG_THREAD_INFO_IN_TASK
1874          struct thread_info thread_info;
1875  #endif
1876          unsigned long stack[THREAD_SIZE/sizeof(long)];
1877  };
```

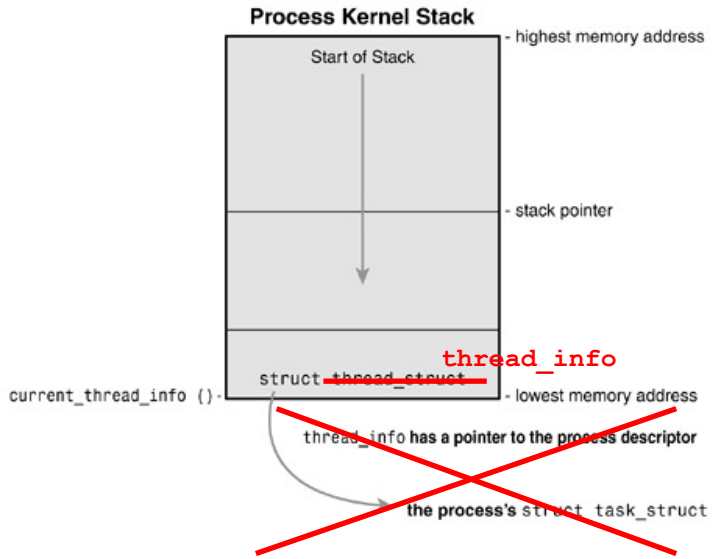
Tasks in Linux

- **`task_struct`** defined in **`include/linux/sched.h`**



Layout

- Processes are *dynamic*, so descriptors are kept in dynamic memory.
- An 8KB memory area (16KB for 64bit) is allocated for each process, to hold **`thread_info`** *and* kernel mode process stack.

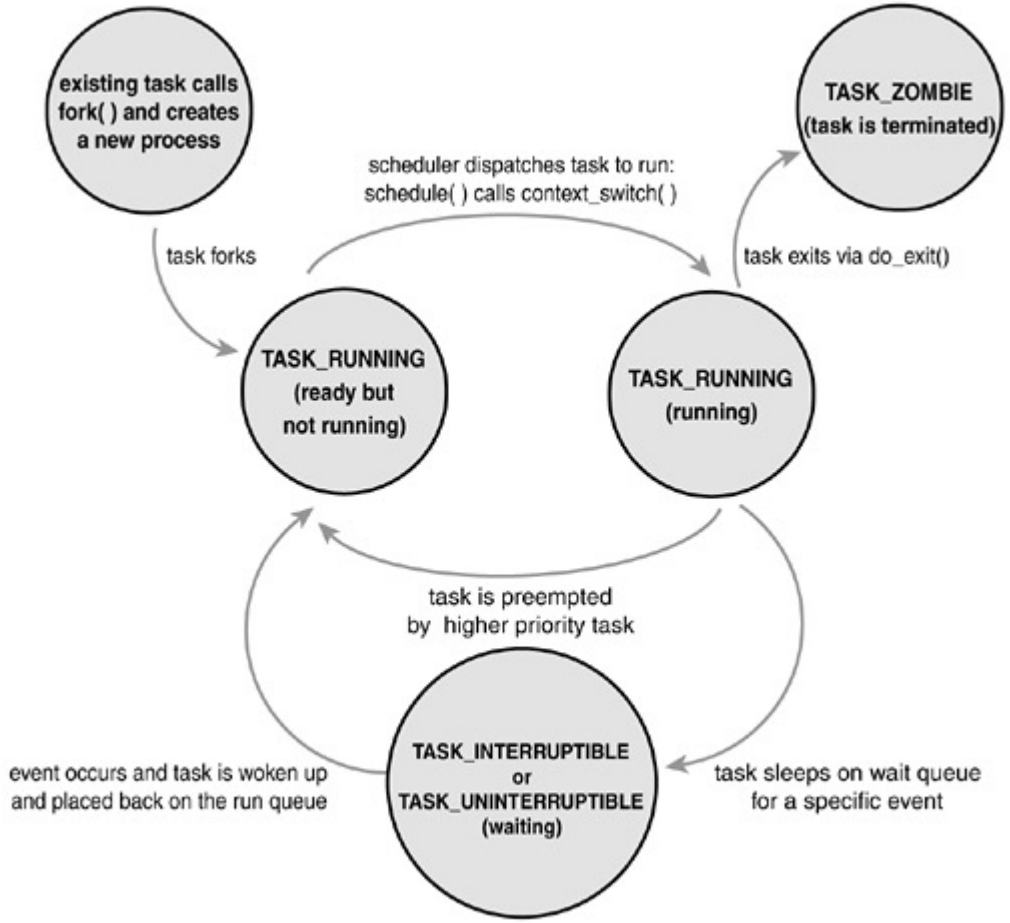


Since Linux 4.9

Task States in Linux

- **`TASK_RUNNING`** – task is either executing on a CPU or is awaiting execution
- **`TASK_INTERRUPTIBLE`** – task is sleeping but can be awaken by an interrupt
- **`TASK_UNINTERRUPTIBLE`** – task is sleeping. Any interrupt delivered to it will not change its state.
  - Seldom used but needed for certain critical wait situations
- **`TASK_STOPPED`** – task is stopped
- **`TASK_TRACED`** – task is stopped by a debugger (using **`ptrace`**)

State transition



Process Identification

- Processes in Unix (including Linux) is identified by a process identifier
- It is merely a number that wraps around 32-bit
- Each process also has user and group identifiers that are used for access control

Process hierarchy

- All processes (in Linux) forms a tree
  - Every process (except init) has exactly one parent
  - The first task is in variable **`init_task`**
  - A process can be the parent of several child processes
- A process awaiting full termination (and is no longer runnable) is a zombie process
- A process whose parent died is an orphaned process
  - An orphaned process will be automatically adopted by init

```
wongwf@wongwf-VirtualBox:~/Desktop$ uname -a
Linux wongwf-VirtualBox 5.16.1 #5 SMP PREEMPT Thu Feb 17 11:15:30 +08 2022 x86_64 x86_64 x86_64 GNU/Linux
wongwf@wongwf-VirtualBox:~/Desktop$ pstree -A
systemd-+-ModemManager---2*[{ModemManager}]
        |-NetworkManager---2*[{NetworkManager}]
        |-accounts-daemon---2*[{accounts-daemon}]
        |-acpid
        |-anacron
        |-avahi-daemon---avahi-daemon
        |-colord---2*[{colord}]
        |-cron
        |-cups-browsed---2*[{cups-browsed}]
        |-cupsd
        |-dbus-daemon
        |-fprintd---4*[{fprintd}]
        |-gdm3-+-gdm-session-wor-+-gdm-wayland-ses-+-gnome-session-b---2*[{gnom+
        |      |                 |                 `-2*[{gdm-wayland-ses}]
        |      |                 `-2*[{gdm-session-wor}]
        |      `-2*[{gdm3}]
        |-geoclue---2*[{geoclue}]
        |-gnome-keyring-d---3*[{gnome-keyring-d}]
        |-irqbalance---{irqbalance}
        |-2*[kerneloops]
        |-networkd-dispat
        |-packagekitd---2*[{packagekitd}]
        |-polkitd---2*[{polkitd}]
        |-power-profiles----2*[{power-profiles-}]
        |-rsyslogd---3*[{rsyslogd}]
        |-rtkit-daemon---2*[{rtkit-daemon}]
        |-snapd---13*[{snapd}]
        |-switcheroo-cont---2*[{switcheroo-cont}]
        |-systemd-+-(sd-pam)
        |         |-at-spi2-registr---2*[{at-spi2-registr}]
        |         |-dbus-daemon
        |         |-dconf-service---2*[{dconf-service}]
        |         |-evolution-addre---6*[{evolution-addre}]
        |         |-evolution-calen---9*[{evolution-calen}]
        |         |-evolution-sourc---3*[{evolution-sourc}]
        |         |-2*[gjs---6*[{gjs}]]
        |         |-gnome-session-b-+-at-spi-bus-laun-+-dbus-daemon
        |         |                 |                 `-3*[{at-spi-bus-laun}]
        |         |                 |-evolution-alarm---6*[{evolution-alarm}]
        |         |                 |-gsd-disk-utilit---2*[{gsd-disk-utilit}]
        |         |                 `-3*[{gnome-session-b}]
```

Parents of Processes

- Real parent: The process descriptor of the process that created it or to the descriptor of process 1 if the parent process no longer exists.

- Parent: The process that will handle the termination signal of the child. May be different due to tracing.
  - **`man ptrace`**

```
/*
 * Pointers to the (original) parent process, youngest child, younger sibling,
 * older sibling, respectively.  (p->father can be replaced with
 * p->real_parent->pid)
 */

/* Real parent process: */
struct task_struct __rcu        *real_parent;

/* Recipient of SIGCHLD, wait4() reports: */
struct task_struct __rcu        *parent;

/*
 * Children/sibling form the list of natural children:
 */
struct list_head                children;
struct list_head                sibling;
struct task_struct              *group_leader;
```

Process Creation

- Processes are created frequently
  - We want to do it quickly
- In Linux, process creation is actually process duplication
  - A child process starts life as a clone of the parent
- Three tricks of Linux kernel
  - Copy-on-Write (COW)
  - Lightweight process creation using `clone()`
  - `vfork()`

Copy-on-Write (COW)

- Parent and the child process initially share same physical pages
  - Linux terminology: physical pages are called page frames or just frames; "pages" refers to virtual pages.
- As long as they are shared, they cannot be modified
- Any attempt to write a shared frame triggers an exception
- Kernel duplicates the page into a new page frame and marks it as writable.
- Original page frame remains write-protected: kernel checks whether the writing process is the only owner of the page frame. If so, makes the page frame writable for the process

clone()

- Allows the child process to share parts of its execution context with the calling process, such as the memory space (page tables), the table of file descriptors, and the table of signal handlers

```
CLONE(2)                  Linux Programmer's Manual                 CLONE(2)

NAME      top

       clone, __clone2 - create a child process

SYNOPSIS      top

       /* Prototype for the glibc wrapper function */

       #define _GNU_SOURCE
       #include <sched.h>

       int clone(int (*fn)(void *), void *child_stack,
                 int flags, void *arg, ...
                 /* pid_t *ptid, void *newtls, pid_t *ctid */ );
```

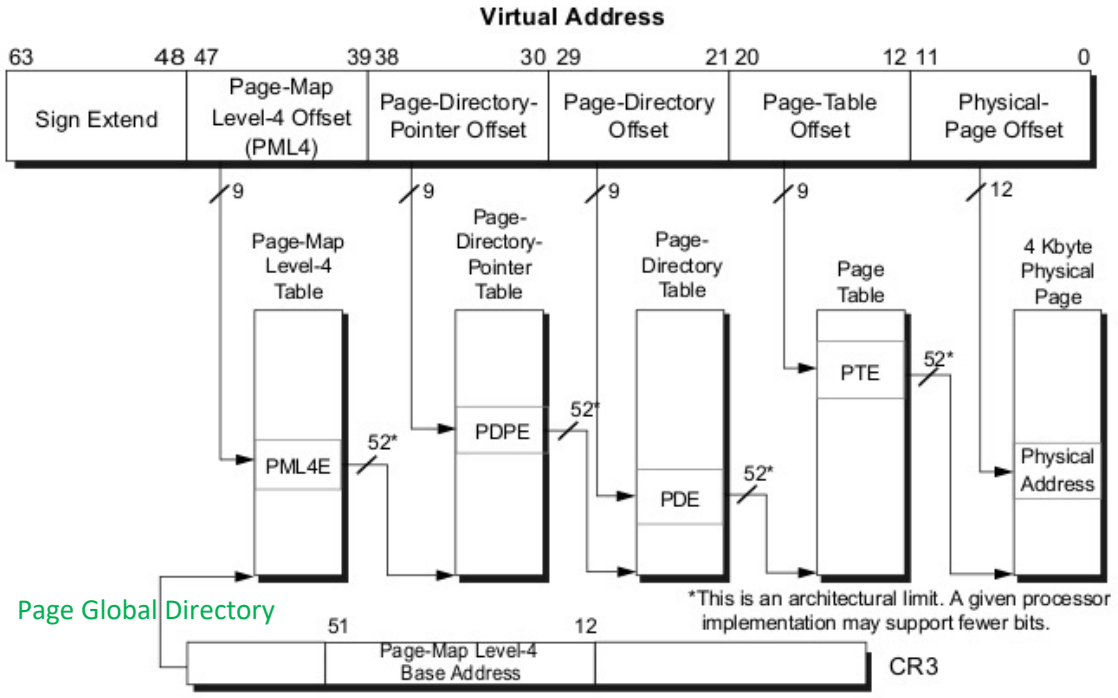
`vfork()`

- Similar to **`clone()`** except that the parent will block until the child exits or performs **`execve()`**
  - **`execve()`** loads a new executable and rewrites the memory space

- implemented as a **`clone()`** system call whose parameter specifies
  - a **`SIGCHLD`** signal
  - **`CLONE_VM`** and **`CLONE_VFORK`** flags
  - **`child_stack`** is the current parent stack pointer

Process switching

- Done in Linux **`schedule()`** function
- Switch the Page Global Directory to install a new address space
- Switch the Kernel Mode stack and the hardware context using the **`switch_to`** macro



Keeping time

A digression

Timer hardware on x86

- Real-time clock (RTC)
  - Use for keeping date
  - Power by a coin battery
- Time-stamp counter (TSC)
  - `rdtsc`
- Programmable Interval Timer (PIT)
  - Global timer interrupt at IRQ 0
- CPU Local Timer
  - Raise local interrupt to the CPU the local APIC is attached to
- High Performance Event Timer (HPET)
  - Chip co-developed by Intel and Microsoft
  - 8 32- or 64-bit independent counters driven by each its own (at least) 10MHz clock
- ACPI Power Management Timer (ACPI PMT)
  - 3.58MHz

Keeping time

- Updates the time elapsed since system startup
- Updates the time and date
- Determines, for every CPU, how long the current process has been running, and preempts it if it has exceeded the time allocated to it
  - A time slot is called a quantum; default = 10ms
- Updates resource usage statistics
- Checks whether the interval of time associated with each software timer has expired

Linux timekeeping architecture

- Uniprocessor: all time-keeping activities are triggered by interrupts raised by the global timer
  - Either the PIT or the HPET

- Multiprocessor:
  - General activities triggered by the interrupts raised by the global timer
  - CPU-specific activities (such as monitoring the execution time of the currently running process) are triggered by the interrupts raised by the local APIC timer

Important time variables

- **`jiffies`**: a counter that stores the number of elapsed ticks since the system was started.
  - It is increased by one when a timer interrupt at 1ms
  - In x86, **`jiffies`** is a 32-bit variable, therefore it wraps around in approximately 50 days
  - In x86-64, it is a 64 bit variable.

- **`xtime`**: stores the current time and date;

CPU load

- 3 values: moving average over 1, 5, and 15 minutes

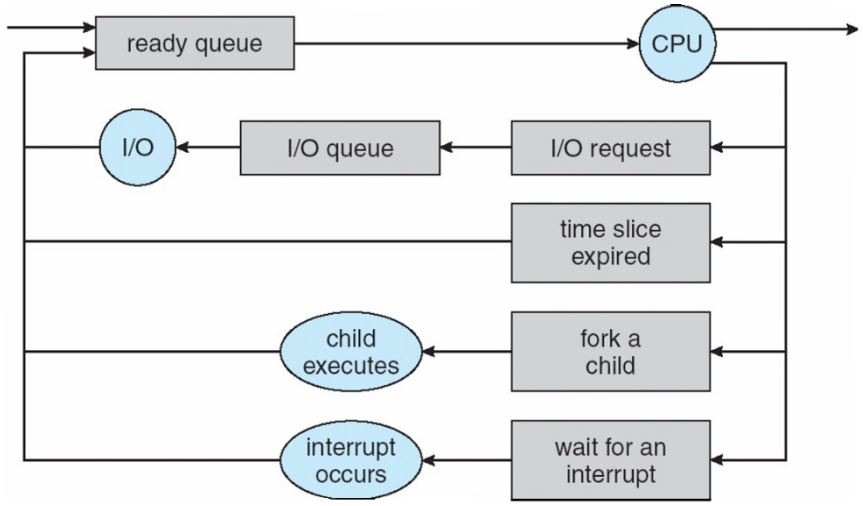
```
/*
 * kernel/sched/loadavg.c
 *
 * This file contains the magic bits required to compute the global loadavg
 * figure. Its a silly number but people think its important. We go through
 * great pains to make it work on big machines and tickless kernels.
 */

#include <linux/export.h>

#include "sched.h"

/*
 * Global load-average calculations
 *
 * We take a distributed and async approach to calculating the global load-avg
 * in order to minimize overhead.
 *
 * The global load average is an exponentially decaying average of nr_running +
 * nr_uninterruptible.
 *
 * Once every LOAD_FREQ:
 *
 *   nr_active = 0;
 *   for_each_possible_cpu(cpu)
 *      nr_active += cpu_of(cpu)->nr_running + cpu_of(cpu)->nr_uninterruptible;
 *
 *   avenrun[n] = avenrun[0] * exp_n + nr_active * (1 - exp_n)
 *
```

Process Scheduling

Source:
Silberschatz, Galvin and Gagne
Operating System Concepts – 8th Edition
2009

CPU Scheduling

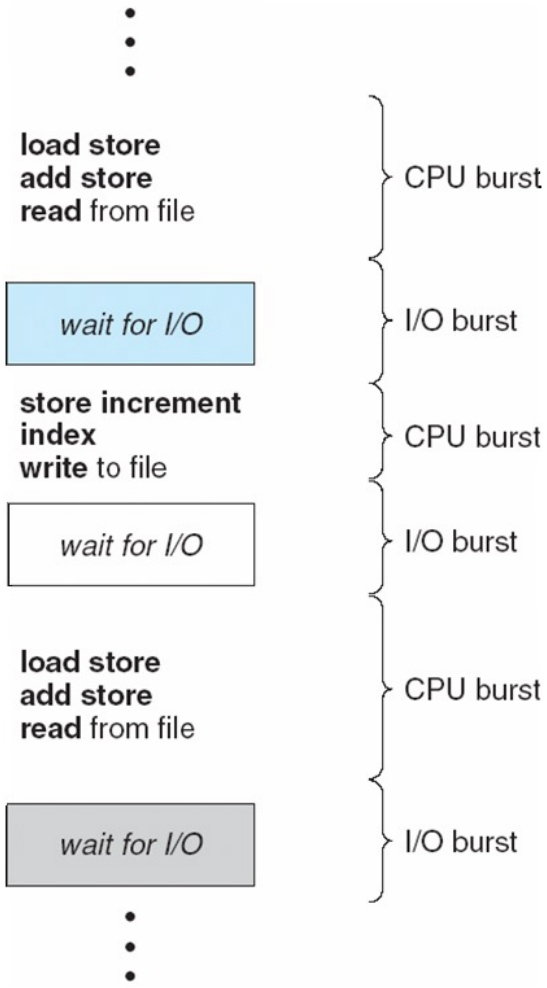
- How is the OS to decide which of several tasks to take off a queue?
- Scheduling: deciding which threads are given access to resources from moment to moment.

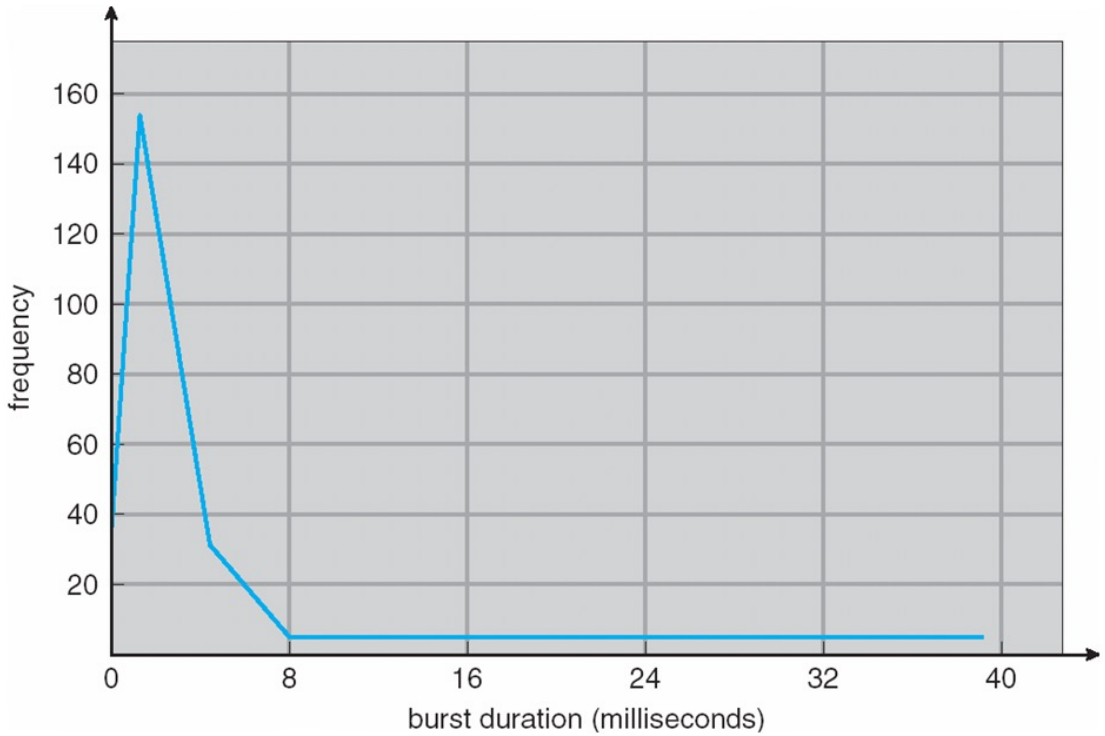


Assumptions about Scheduling

- CPU scheduling big area of research in early ‘70s
- Many implicit assumptions for CPU scheduling:
  - One program per user
  - One thread per program
  - Programs are independent
- These are unrealistic but simplify the problem
- Does “fair” mean fairness among users or programs?
  - If I run one compilation job and you run five, do you get five times as much CPU?
    - Often times, yes!
- Goal: dole out CPU time to optimize some desired parameters of the system.
  - What parameters?

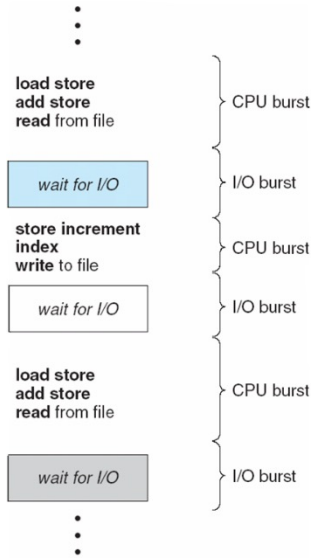
Assumption: CPU Bursts

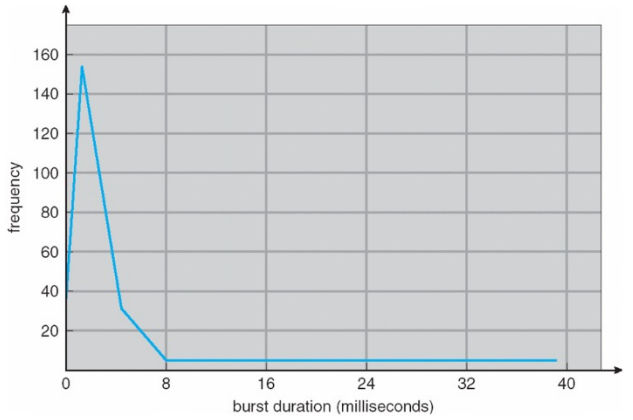




Assumption: CPU Bursts

- Execution model: programs alternate between bursts of CPU and I/O
  - Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
  - Each scheduling decision is about which job to give to the CPU for use by its next CPU burst
  - With timeslicing, thread may be forced to give up CPU before finishing current CPU burst.





What is Important in a Scheduling Algorithm?

- Minimize Response Time
  - Elapsed time to do an operation (job)
  - Response time is what the user sees
    - Time to echo keystroke in editor
    - Time to compile a program
    - Real-time Tasks: Must meet deadlines imposed by World
- Maximize Throughput
  - Jobs per second
  - Throughput related to response time, but not identical
    - Minimizing response time will lead to more context switching than if you maximized only throughput
  - Minimize overhead (context switch time) as well as efficient use of resources (CPU, disk, memory, etc.)
- Fairness
  - Share CPU among users in some equitable way
  - Not just minimizing average response time

Turnaround time is the amount of time elapsed from the time of submission to the time of completion.

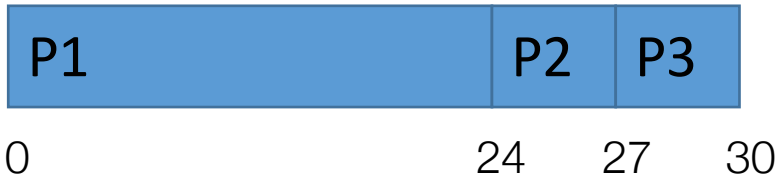
Response time is the average time elapsed from submission until the first response is produced.

Scheduling Algorithms: First-Come, First-Served (FCFS)

- “Run until Done:” FIFO algorithm
- In the beginning, this meant one program runs non-preemtively until it is finished (including any blocking for I/O operations)
- Now, FCFS means that a process keeps the CPU until one or more threads block
- Example: Three processes arrive in order P1, P2, P3.
  - P1 burst time: 24
  - P2 burst time: 3
  - P3 burst time: 3

Scheduling Algorithms: First-Come, First-Served (FCFS)

- Example: Three processes arrive in order P1, P2, P3.
  - P1 burst time: 24
  - P2 burst time: 3
  - P3 burst time: 3
- Waiting Time
  - P1: 0
  - P2: 24
  - P3: 27
- Completion Time:
  - P1: 24
  - P2: 27
  - P3: 30
- Average Waiting Time: (0+24+27)/3 = 17
- Average Completion Time: (24+27+30)/3 = 27



Scheduling Algorithms: First-Come, First-Served (FCFS)

- What if their order had been P2, P3, P1?
  - P1 burst time: 24
  - P2 burst time: 3
  - P3 burst time: 3
- Waiting Time
  - P1: 0
  - P2: 3
  - P3: 6
- Completion Time:
  - P1: 3
  - P2: 6
  - P3: 30
- Average Waiting Time: (0+3+6)/3 = 3 (compared to 17)
- Average Completion Time: (3+6+30)/3 = 13 (compared to 27)

Scheduling Algorithms: First-Come, First-Served (FCFS)

- Average Waiting Time: (0+3+6)/3 = 3 (compared to 17)
- Average Completion Time: (3+6+30)/3 = 13 (compared to 27)
- FIFO Pros and Cons:
  - Simple (+)
  - Short jobs get stuck behind long ones (-)
    - If all you're buying is milk, doesn't it always seem like you are stuck behind a cart full of many items
  - Performance is highly dependent on the order in which jobs arrive (-)

Round Robin (RR) Scheduling

- FCFS Scheme: Potentially bad for short jobs!
  - Depends on submit order
  - If you are first in line at the supermarket with milk, you don't care who is behind you; on the other hand…
- Round Robin Scheme
  - Each process gets a small unit of CPU time (time quantum)
    - Usually 10-100 ms
  - After quantum expires, the process is preempted and added to the end of the ready queue
  - Suppose N processes in ready queue and time quantum is Q ms:
    - Each process gets 1/N of the CPU time
    - In chunks of at most Q ms
    - What is the maximum wait time for each process?
      - No process waits more than (n-1)q time units

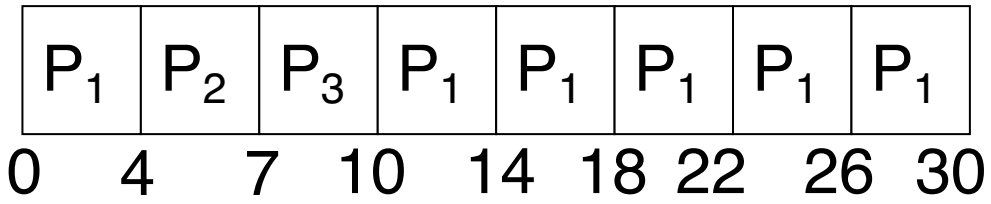
Round Robin (RR) Scheduling

- Round Robin Scheme
  - Each process gets a small unit of CPU time (time quantum)
    - Usually 10-100 ms
  - After quantum expires, the process is preempted and added to the end of the ready queue
  - Suppose N processes in ready queue and time quantum is Q ms:
    - Each process gets 1/N of the CPU time
    - In chunks of at most Q ms
    - What is the maximum wait time for each process?
      - No process waits more than (n-1)q time units
- Performance Depends on Size of Q
  - Small Q => interleaved
  - Large Q is like FCFS
  - Q must be large with respect to context switch time, otherwise overhead is too high (spending most of your time context switching!)

Example of RR with Time Quantum = 4

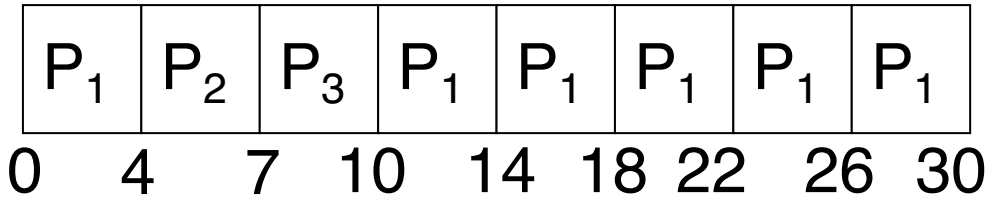
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



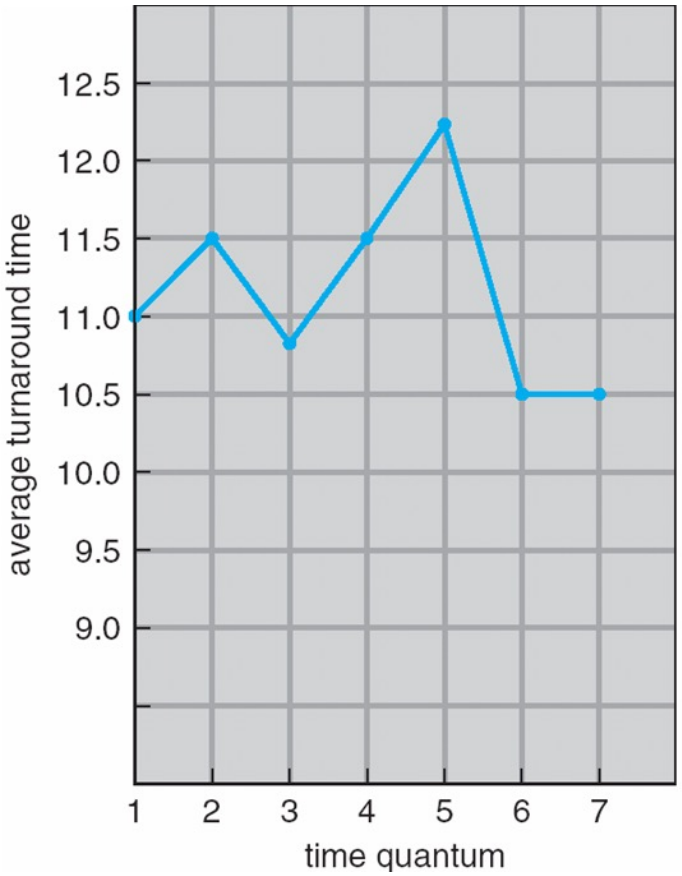
Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3



- Waiting Time:
  - P1: (10-4) = 6
  - P2: (4-0) = 4
  - P3: (7-0) = 7
- Completion Time:
  - P1: 30
  - P2: 7
  - P3: 10
- Average Waiting Time: (6 + 4 + 7)/3= 5.67
- Average Completion Time: (30+7+10)/3=15.67

Turnaround Time Varies With The Time Quantum

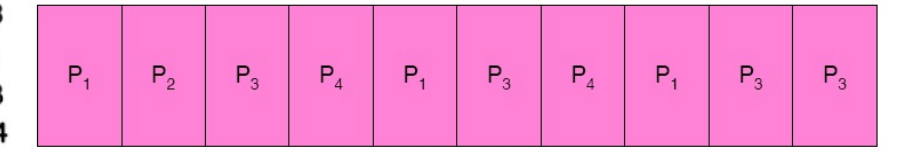


process	time
P_1	6
P_2	3
P_3	1
P_4	7

Example of RR with Time Quantum = 20

- Waiting Time:
  - P1: (68-20)+(112-88) = 72
  - P2: (20-0) = 20
  - P3: (28-0)+(88-48)+(125-108) = 85
  - P4: (48-0)+(108-68) = 88
- Completion Time:
  - P1: 125
  - P2: 28
  - P3: 153
  - P4: 112
- Average Waiting Time: (72+20+85+88)/4 = 66.25
- Average Completion Time: (125+28+153+112)/4 = 104.5

Process	Burst Time
P_1	53
P_2	8
P_3	68
P_4	24



A process can finish before the time quantum expires, and release the CPU.

RR Summary

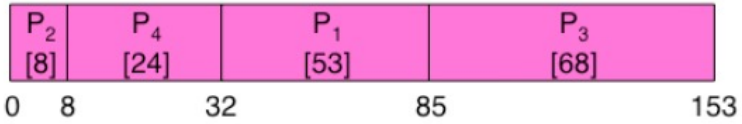
- Pros and Cons:
  - Better for short jobs (+)
  - Fair (+)
  - Context-switching time adds up for long jobs (-)
    - The previous examples assumed no additional time was needed for context switching – in reality, this would add to wait and completion time without actually progressing a process towards completion.
    - Remember: the OS consumes resources, too!
- If the chosen quantum is
  - too large, response time suffers
  - infinite, performance is the same as FIFO
  - too small, throughput suffers and percentage overhead grows
- Actual choices of timeslice:
  - UNIX: initially 1 second:
    - Worked when only 1-2 users
    - If there were 3 compilations going on, it took 3 seconds to echo each keystroke!
  - In practice, need to balance short-job performance and long-job throughput:
    - Typical timeslice 10ms-100ms
    - Typical context-switch overhead 0.1ms – 1ms (about 1%)

Comparing FCFS and RR

- Assuming zero-cost context switching time, is RR always better than FCFS?
- Assume 10 jobs, all start at the same time, and each require 100 seconds of CPU time
- RR scheduler quantum of 1 second
- Completion Times (CT)
  - Both FCFS and RR finish at the same time
  - But average response time is much worse under RR!
    - Bad when all jobs are same length
- Also: cache state must be shared between all jobs with RR but can be devoted to each job with FIFO
  - Total time for RR longer even for zero-cost context switch!

Job #	FCFS CT	RR CT
1	100	991
2	200	992
…	…	…
9	900	999
10	1000	1000

Comparing FCFS and RR



	Quantum	P_1	P_2	P_3	P_4	Average
Wait Time	Best FCFS	32	0	85	8	$31\frac{1}{4}$
	Q = 1	84	22	85	57	62
	Q = 5	82	20	85	58	$61\frac{1}{4}$
	Q = 8	80	8	85	56	$57\frac{1}{4}$
	Q = 10	82	10	85	68	$61\frac{1}{4}$
	Q = 20	72	20	85	88	$66\frac{1}{4}$
	Worst FCFS	68	145	0	121	$83\frac{1}{2}$
Completion Time	Best FCFS	85	8	153	32	$69\frac{1}{2}$
	Q = 1	137	30	153	81	$100\frac{1}{2}$
	Q = 5	135	28	153	82	$99\frac{1}{2}$
	Q = 8	133	16	153	80	$95\frac{1}{2}$
	Q = 10	135	18	153	92	$99\frac{1}{2}$
	Q = 20	125	28	153	112	$104\frac{1}{2}$
	Worst FCFS	121	153	68	145	$121\frac{3}{4}$

Scheduling

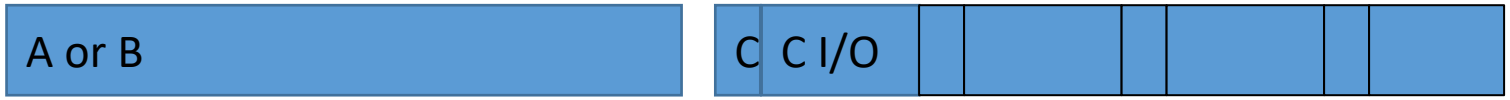
- The performance we get is somewhat dependent on what "kind" of jobs we are running (short jobs, long jobs, etc.)
- If we could "see the future," we could mirror best FCFS
- Shortest Job First (SJF) a.k.a. Shortest Time to Completion First (STCF):
  - Run whatever job has the least amount of computation to do
- Shortest Remaining Time First (SRTF) a.k.a. Shortest Remaining Time to Completion First (SRTCF):
  - Preemptive version of SJF: if a job arrives and has a shorter time to completion than the remaining time on the current job, immediately preempt CPU
- These can be applied either to a whole program or the current CPU burst of each program
  - Idea: get short jobs out of the system
  - Big effect on short jobs, only small effect on long ones
  - Result: better average response time

Scheduling

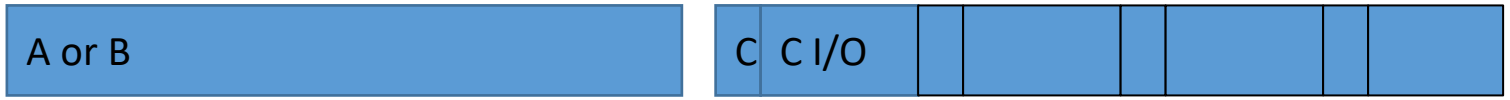
- But, time to completion, remaining time, burst time etc. are hard to estimate
- We could get feedback from the program or the user, but they have incentive to lie!
- SJF/SRTF are the best you can do at minimizing average response time
  - Provably optimal (SJF among non-preemptive, SRTF among preemptive)
  - Since SRTF is always at least as good as SJF, focus on SRTF
- Comparison of SRTF with FCFS and RR
  - What if all jobs are the same length?
    - SRTF becomes the same as FCFS (i.e. FCFS is the best we can do)
  - What if all jobs have varying length?
    - SRTF (and RR): short jobs are not stuck behind long ones

Example: SRTF

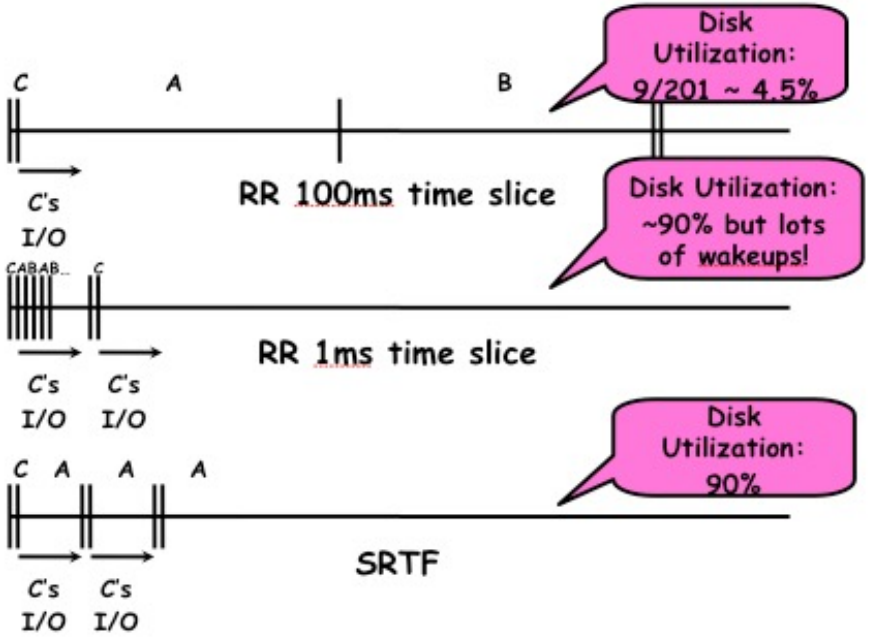
- A,B: both CPU bound, run for a week
- C: I/O bound, loop 1ms CPU, 9ms disk I/O
- If only one at a time, C uses 90% of the disk, A or B could use 100% of the CPU
- With FIFO: Once A and B get in, the CPU is held for two weeks
- What about RR or SRTF?
  - Easier to see with a timeline



Example: SRTF



- A,B: both CPU bound, run for a week
- C: I/O bound, loop 1ms CPU, 9ms disk I/O



SRTF summary

- Starvation
  - SRTF can lead to starvation if many small jobs!
  - Large jobs never get to run
- Somehow need to predict future
  - How can we do this?
  - Some systems ask the user
    - When you submit a job, you have to say how long it will take
    - To stop cheating, system kills job if it takes too long
  - But even non-malicious users have trouble predicting runtime of their jobs
- Bottom line, can't really tell how long job will take
  - However, can use SRTF as a yardstick for measuring other policies, since it is optimal
- SRTF Pros and Cons
  - Optimal (average response time) (+)
  - Hard to predict future (-)
  - Unfair, even though we minimized average response time! (-)

Predicting the Future

- Back to predicting the future… perhaps we can predict the next CPU burst length?
- Iff programs are generally repetitive, then they may be predictable
- Create an adaptive policy that changes based on past behavior
  - CPU scheduling, virtual memory, file systems, etc.
  - If program was I/O bound in the past, likely in the future
- Example: SRTF with estimated burst length
  - Use an estimator function on previous bursts
  - Let T(n-1), T(n-2), T(n-3), …, be previous burst lengths. Estimate next burst T(n) = f(T(n-1), T(n-2), T(n-3),…)
  - Function f can be one of many different time series estimation schemes

Determining Length of Next CPU Burst

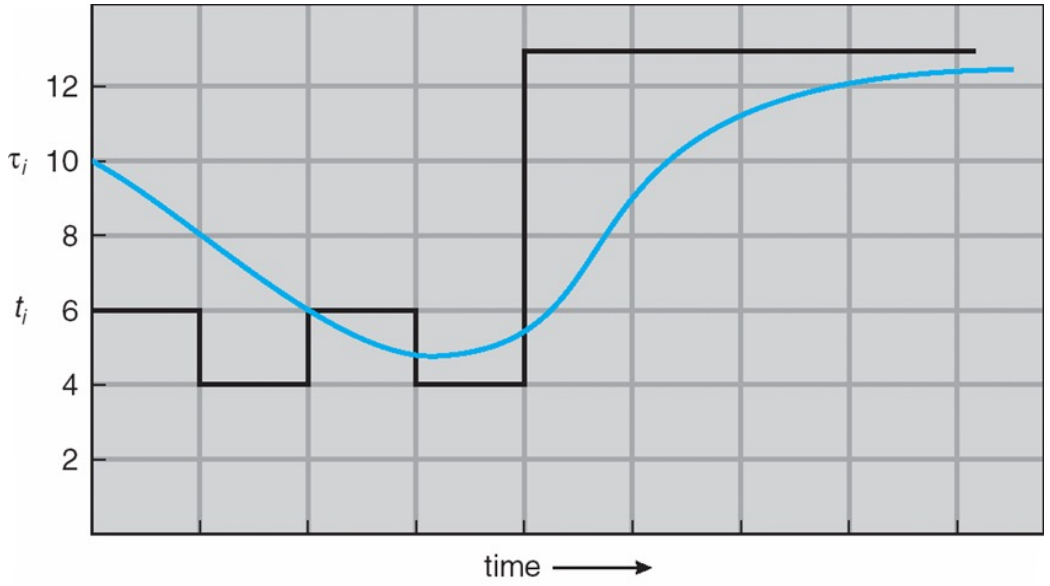
- Can only estimate the length
- Can be done by using the length of previous CPU bursts, using exponential averaging

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$$

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$

Predicting the Future

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$$



CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive (if a higher priority process enters, it receives the CPU immediately)
  - Nonpreemptive (higher priority processes must wait until the current process finishes; then, the highest priority ready process is selected)
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ Starvation – low priority processes may never execute
- Solution ≡ Aging – as time progresses increase the priority of the process
- Example: Windows gives foreground processes higher priority than background ones

Linux niceness

- Base priority of processes is known as niceness in Linux
  - Value is between -20 (most favorable for scheduling) to 19 (least favorable)
- User can change niceness value within certain limits
- Superseded by `getpriority()` and `setpriority()` calls
  - Range is kernel version dependent

Priority Inversion Example

- Three processes: one with high priority (H), one with medium priority (M), and one with low priority (L)
- Process L is running and successfully acquires a lock
- Process H kicks in and preempts process L
- Process H tries to acquire L's lock, and blocks (because it is held by L)
- Process M begins running, and, since it has a higher priority than L, it is the highest priority ready process. It preempts L and runs, thus starving high priority process H.
- This is known as priority inversion

Fixing Priority Inversion

- Fix: Process L temporarily granted the high priority of process H
  - Priority inheritance
- This enables process L to preempt process M and run.
- When process L is finished, process H becomes unblocked.
- Process H, now being the highest priority ready process, runs, and process M must wait until it is finished.
- Note that if process M's priority is actually higher than process H, process L having H's priority is insufficient to preempt M.
  - This is expected behavior

Multi-level Feedback Scheduling

- Another method for exploiting past behavior
  - Multiple queues, each with different priority
    - Higher priority queues often considered “foreground” tasks
  - Each queue has its own scheduling algorithm
    - E.g. foreground → RR, background → FCFS
    - Sometimes multiple RR priorities with quantum increasing exponentially (highest queue: 1ms, next: 2ms, next: 4ms, etc.)
  - Adjust each job’s priority as follows (details vary)
    - Job starts in highest priority queue
    - If entire CPU time quantum expires, drop one level
    - If CPU is yielded during the quantum, push up one level (or to top)

Scheduling Details

- Result approximates SRTF
  - CPU bound jobs drop rapidly to lower queues
  - Short-running I/O bound jobs stay near the top
- Scheduling must be done between the queues
  - Fixed priority scheduling: serve all from the highest priority, then the next priority, etc.
  - Time slice: each queue gets a certain amount of CPU time (e.g., 70% to the highest, 20% next, 10% lowest)
- Countermeasure: user action that can foil intent of the OS designer
  - For multilevel feedback, put in a bunch of meaningless I/O to keep job's priority high
  - But if everyone does this, it won't work!

Scheduling Details

- It is apparent that scheduling is facilitated by having a “good mix” of I/O bound and CPU bound programs, so that there are long and short CPU bursts to prioritize around.
- There is typically a long-term and a short-term scheduler in the OS.
- We have been discussing the design of the short-term scheduler.
- The long-term scheduler decides what processes should be put into the ready queue in the first place for the short-term scheduler, so that the short-term scheduler can make fast decisions on a good mix of a subset of ready processes.
- The rest are held in memory or disk
  - This also provides more free memory for the subset of ready processes given to the short-term scheduler.

Fairness

- What about fairness?
  - Strict fixed-policy scheduling between queues is unfair (run highest, then next, etc.)
    - Long running jobs may never get the CPU
    - In Multics, admins shut down the machine and found a 10-year-old job
  - Must give long-running jobs a fraction of the CPU even when there are shorter jobs to run
    - Tradeoff: fairness gained by hurting average response time!
- How to implement fairness?
  - Could give each queue some fraction of the CPU
    - i.e., for one long-running job and 100 short-running ones?
    - Like express lanes in a supermarket – sometimes express lanes get so long, one gets better service by going into one of the regular lines
  - Could increase priority of jobs that don't get service (as seen in the multilevel feedback example)
    - This was done in UNIX
    - Ad hoc – with what rate should priorities be increased?
    - As system gets overloaded, no job gets CPU time, so everyone increases in priority
      - Interactive processes suffer

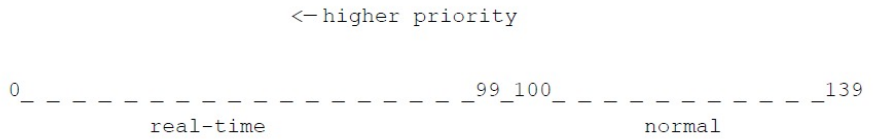
Lottery Scheduling

- Yet another alternative: Lottery Scheduling
  - Give each job some number of lottery tickets
  - On each time slice, randomly pick a winning ticket
  - On average, CPU time is proportional to number of tickets given to each job over time
- How to assign tickets?
  - To approximate SRTF, short-running jobs get more, long running jobs get fewer
  - To avoid starvation, every job gets at least one ticket (everyone makes progress)
- Advantage over strict priority scheduling: behaves gracefully as load changes
  - Adding or deleting a job affects all jobs proportionally, independent of how many tickets each job possesses

Process Scheduling in Linux

Type of Processes

- Normal processes
  - Priorities are dynamic
- Real-time processes
  - Standard Linux does not have hard realtime processes
  - Two scheduling policies: FIFO or round-robin
  - Static priorities
  - Dictated by POSIX standard



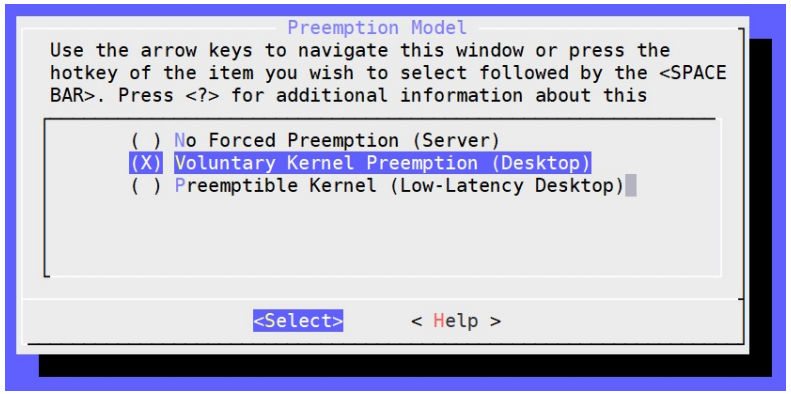
Priority scale used inside the Linux kernel

Miscellaneous issues

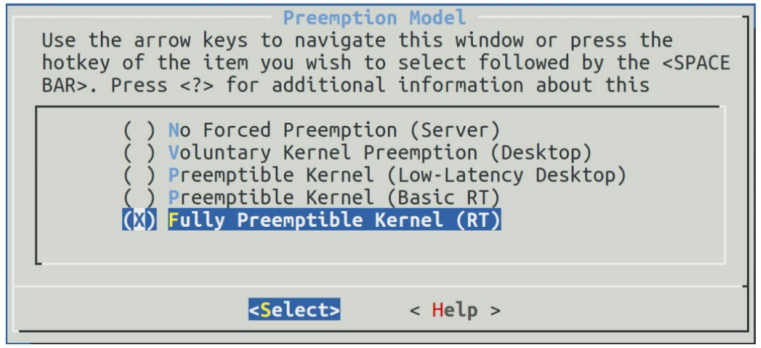
- The main entry of the scheduler is **`schedule()`**
  - Called in many places when a CPU control is to be relinquished
- There is one or more runqueues for tasks per CPU
  - Exact configuration depends on the scheduler used
- Since version 2.6, the kernel itself is preemptible
  - Except when local interrupts (timer) are disabled
  - Structure of kernel spinlocks had to be changed
- Each task has a processor affinity mask which tells which CPU the task can run on

Preemption options

Standard default:



Real-time patch applied:



`schedule()`

- Defined in `kernel/sched/core.c`
- Goal: replace currently executing task with another one
- Either:
  - Called explicitly in many places where CPU is to be relinquished or
  - in a lazy way by activating the “need scheduling” flag that the kernel often checks
- May not always switch task
  - No other task in the runqueue
  - No task of higher priority
  - Timeslice (quantum) not expired

scheduler_tick()

- The timer interrupt is used for scheduling
- On a timer interrupt, `update_process_times()` is called.
- It in turns calls `scheduler_tick()`
  - Triggers preemption if necessary

Scheduling Realtime Processes in Linux

The realtime scheduler

- Code in **`kernel/sched/rt.c`**
- A real-time task inhibits the execution of every lower-priority task while it remains runnable
- A real-time task is replaced by another task only when:
  - The task is preempted by another task of higher real-time priority
  - The task performs a blocking operation, and it is put to sleep.
  - The task is stopped, or it is killed.
  - The task voluntarily relinquishes the CPU
  - The task is **`SCHED_RR`** and it has exhausted its time quantum.

Linux real-time scheduling policies

- **`SCHED_FIFO`**
  - No time slicing
  - Can be used only with static priorities higher than 0
    - “Low class” citizen, easily preempted
  - The preempted FIFO task will stay at the head of the list for its priority and will resume execution as soon as all tasks of higher priority are blocked again.
  - When a blocked SCHED_FIFO task becomes runnable, it will be inserted at the end of the list for its priority.

- **`SCHED_RR`**
  - Given a time quantum, pre-empted when it runs out
  - Preempted task will be put at the end of the list for its priority.

Scheduling Normal Processes in Linux

The early scheduling algorithm

- First scheduler introduced in kernel version 0.01 in 1991
- Only one task queue
- Code scans entire queue to look for next task to schedule
- Not scalable
- Implements preemptive multitasking

Linux kernel 0.01 Scheduler

```
17      /* this is the scheduler proper: */
18      while (1) {
19          c = -1;
20          next = 0;
21          i = NR_TASKS;
22          p = &task[NR_TASKS];
23          while (--i) {
24              if (!*--p)
25                  continue;
26                  if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
27                      c = (*p)->counter, next = i;
28          }
29          if (c) break;
30          for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
31              if (*p)
32                  (*p)->counter = ((*p)->counter >> 1) + (*p)->priority;
33      }
34      switch_to(next);
```

O(n) Scheduler

- Used in kernel version 2.4
- Similar to the version 0.01 in idea
- Time is divided up into epoch
- Every task can execute up to its timeslice in the current epoch
- Half of the unused time in the current epoch is added to the next epoch
- Each task has a goodness factor
  - Goodness is a function of the remaining time and the priority of the task
  - Runqueue is scanned and the “most good” task is chosen to run next

O(n) Scheduler

- Advantage: Simple
- Disadvantage: Cannot scale

Current state: The duel of two hackers

Ingo Molnár

Nationality	Hungarian
Occupation	Programmer
Employer	Red Hat
Known for	Completely Fair Scheduler

VS

Con Kolivas

Computer programmer

Con Kolivas is an Australian anaesthetist. He has worked as a computer programmer on the Linux kernel and on the development of the cryptographic currency mining software CGMiner. Wikipedia

Born: Melbourne, Australia

Residence: Melbourne, Australia

Australian doctors design and make life-saving equipment needed for coronavirus pandemic

By Dan Colasimone

Posted Mon 6 Apr 2020 at 5:36am, updated Mon 6 Apr 2020 at 8:25am

Con Kolivas is a retired anaesthetist and software engineer who says the pandemic needs novel solutions. *(ABC News)*

O(1) Scheduler

- Ingo Molnár introduced a constant time task scheduler in 2.6 kernel
- Global priority scale, from 0 to 139
  - lower value, higher priority
  - real-time task: 0 to 99 and normal task: 100 to 139
- Higher priority tasks got larger timeslices – important tasks would run longer
- Early preemption: when a task entered `TASK_RUNNING` state, the kernel checks whether its priority was higher than the one of currently running task's, and if it was, the scheduler was invoked to grant CPU to the newly runnable task;
- Static priority for real-time tasks;
- Dynamic priority for normal tasks, depending on their interactivity.
  - A task's priority was recalculated when it finished its timeslice.
  - According to its behavior in the past, level of interactivity was determined.
  - Interactive tasks were preferred by the scheduler.

Compared to before

- Previous schedulers explicitly recalculating each task's priority at the end of an epoch
- The tasks were kept in one runqueue (linked list or array)
- A loop over all the existing tasks on the system was run through every time a switch was in place
- O(1) scheduler instead maintains two priority arrays: active and expired per CPU runqueue

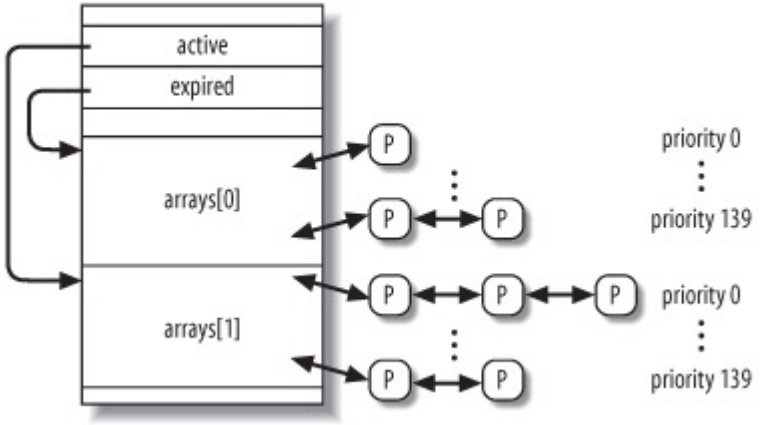
The two priority arrays

- Active array: all the tasks in the associated runqueue that had timeslice left
- Expired array: all the tasks that had exhausted their timeslice
- Every time a task exhausted its timeslice, it is moved from active to expired, and a priority recalculation is made
- When active array is empty, swap the two arrays

Storage layout of the two arrays

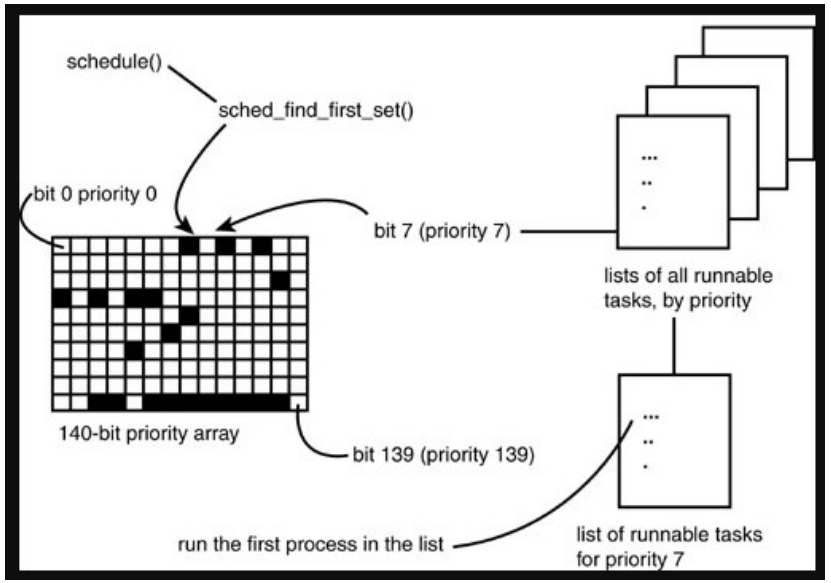
Each priority array consists of

- 140 doubly linked list heads (one list for each priority value)
- a priority bitmap
- number of tasks in the set.



The priority bitmap

- One bit for each priority level
- Bit position ordered by priority level
- If bit is set, that priority level has at least one task in it
- For easy finding of whether a particular priority level is empty or not
- Updated when a new task is inserted into the priority array



Major flaw of O(1) scheduler

- “Interactive-ness” very difficult to say for sure
  - But it is important to give interactive tasks higher priority

- Tried to identify interactive tasks by analyzing the average sleep time (waiting for input) and the scheduler gave a priority bonus to such tasks for better throughput and user experience.

Staircase Scheduler

- Introduced by Con Kolivas in 2004
- Runqueue is a ranked array of tasks (for each CPU)
- Initially all tasks sorted by priority in the runqueue
- The difference: no expired list
  - If task P with priority x exhaust its timeslice of t, its priority becomes (x-1) (smaller is lower priority) and P is inserted into the runqueue
  - When P reaches lowest priority (say L) and exhaust its timeslice ("falling off the stairs"), it is given priority (L+1) and a timeslice of $2t$, and reinserted into the runqueue
  - If it happens again, priority increased to (L+2) timeslice increased to $3t$ and so on

Rationale of the Staircase Scheduler

- Interactive tasks sleep a lot (waiting for I/O) and don't exhaust their timeslice as frequently as batch jobs
- Batch jobs sink to the lower priorities but "earns" bigger timeslices
  - Gradually given more CPU time to finish up
- Interactive jobs tend to stay at the top

The Rotating Staircase Scheduler

- Released by Con Kolivas in 2007
- Reintroduced the expired array and the concept of an epoch
- Each task has a quota:
  - `RR_INTERVAL` (6ms) for those between niceness 0 and 19
  - Progressively higher for those between niceness -1 to -20
- Each runqueue (one per CPU) has an array of quota for each priority
- Each time we deduct the quota of a task, we also do so for its priority level's quota

Basic idea

- Select task from the front of the highest priority level queue to run next – O(1) using the bitmap trick

- Check when a task is enqueued at a particular priority level:
  1. If first time for current epoch, timeslice is assigned the fixed quota value. Same value added to runqueue's quota for this priority.
  2. If not (1), but neither timeslice for the task nor the total quota for this priority exhausted, task simply enqueued
  3. If not (2), task lowered to next priority level, repeat (1) at the new level
  4. If cannot do (3), task move to expired list at its original priority

Rotations (epoch)

- Minor rotation: When a quota for a priority level is exhausted, any remaining tasks at the current priority level is forced to be lowered to the next level of priority

- Major rotation: When all the quotas in the active array are exhausted, swap active and expired arrays
  - Time between major rotation is defined as an 'epoch'

Deadline

- Unlike previous schedulers, no dependence on sleep duration
- Duration of a epoch (time between major rotation) can be predicted

So for example, if two nice 0 tasks are running, and one has just expired as another is activated for the first time receiving a full quota for this runqueue rotation, the first task will wait:

```
nr_tasks * max_duration + nice_difference * rr_interval
1 * 19 * RR_INTERVAL + 0 = 114ms
```

In the presence of a nice 10 task, a nice 0 task would wait a maximum of

```
1 * 10 * RR_INTERVAL + 0 = 60ms
```

In the presence of a nice 0 task, a nice 10 task would wait a maximum of

```
1 * 19 * RR_INTERVAL + 9 * RR_INTERVAL = 168ms
```

Niceness is between -20 (most favorable for scheduling) to 19 (least favorable)

Favoring interactivity

- Interactive jobs sleep for a long time
  - Note: When asleep, such task do not appear in the runqueue
- Unlikely to wake up in the same epoch
- They may have used some of the timeslice in another epoch, but when they wake up in the current epoch, they get the full quota
- They also less likely to get priority demotion

Completely Fair Scheduling

Completely Fair Scheduler

- None of Con Kolivas code accepted in main branch
- Ingo Molnar came up with the completely fair scheduler (CFS)
- Standard scheduler in Linux since kernel version 2.6.23
- CFS basically models an "*ideal, precise multitasking CPU*" on real hardware.
  - With *n* running tasks, each task would be having 1/*n* amount of CPU-time while running constantly
  - Based on the idea of weighted fair queuing in network scheduling

Ideal Fair Scheduling

- If there are N processes, each should get 1/N of the CPU time
  - If at time x, 1 process finishes, then the remaining N-1 processes should get the 1/(N-1) for the rest of the time

- What about priority?
  - Weighted fairness

Approximating fairness

- No single, fixed timeslice: the scheduler calculates how long a task should run as a function of the total number of currently runnable tasks
  - The situation is dynamic
  - Need a lower bound to prevent thrashing as the number of tasks scales
- Priority should be considered.
- Two challenges:
  - How to be fair?
  - How to be efficient (least overhead)?

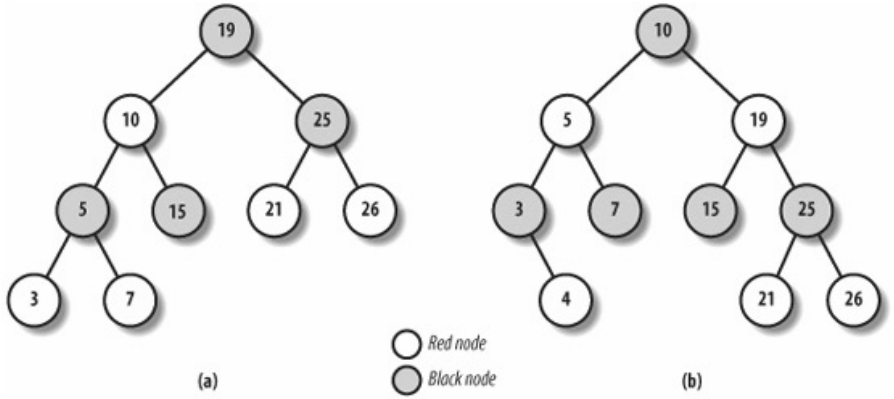
Time-ordered Red-black Tree

- Instead of lists and arrays, CFS uses a red-black tree as its central data structure
- Node is indexed by "virtual time" – a function of how long a task has run
- The smaller the key, the more to the left of the tree a node is
- The scheduler always picks the leftmost node as the next task to run

Red-black Trees

A digression

Red-black trees



- A variant of binary search tree (BST)

- Balance: the path from the root to the farthest leaf is no more than twice as long as the path from the root to the nearest leaf

- Every red-black tree with n internal nodes has a height of at most $2 \times \log(n + 1)$

Properties of Red-Black Trees

Property 1: Every node must be either red or black

Property 2: The root of the tree must be black

Property 3: All leaves ("nodes" containing NIL/NULL) is black

Property 4: If a node is red, then both its children are black

Property 5: Every path from a node to a descendant leaf must contain the **same** number of black nodes (black depth)

Operations on RB Trees

- All operations can be performed in $O(\log n)$ time.
- The query operations, which don't modify the tree, are performed in exactly the same way as they are in BSTs.
- Insertion and Deletion are not straightforward.

Source: Wikipedia

Query Operation

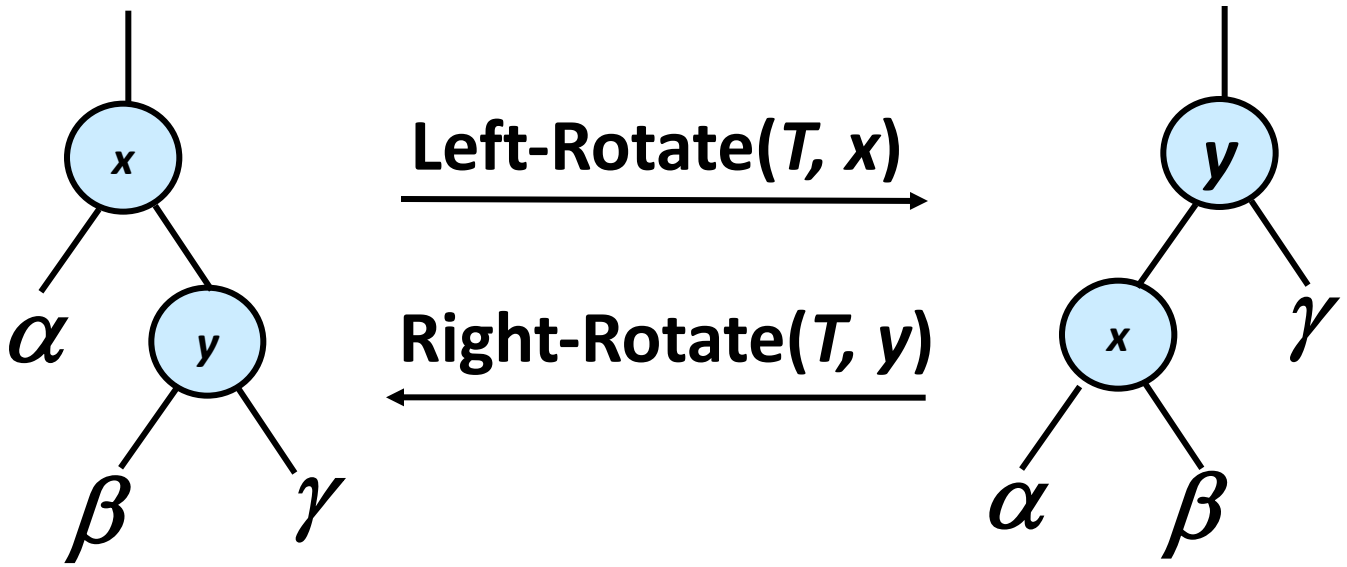
- Relies on the BST property – for a given node X with key k:
  - all the nodes on its left branch are either null or have keys less than k
  - all the nodes on its right branch are either null or have keys greater than k

- To find a node with a given key is $O(\log n)$

Rotations

- Rotations are the basic tree-restructuring operation for almost all *balanced* search trees.
- Rotation takes a red-black-tree and a node,
- Changes pointers to change the local structure, and
- Won't violate the binary-search-tree property.
- Left rotation and right rotation are inverses.

Rotations



Insert into a Red-Black Tree

- Colour the node to be inserted by red initially
- Insert just like in a binary search tree
- Check for property violation and do one of two things (may need repetition):
  - Change the a red node to black
  - Rotation
- Terminology: an uncle of a node X is the sibling of the parent of X
- Notation: let node inserted be N, its parent P, and its uncle U, and the grandparent (i.e., parent of P) be G

Insertion Case 1

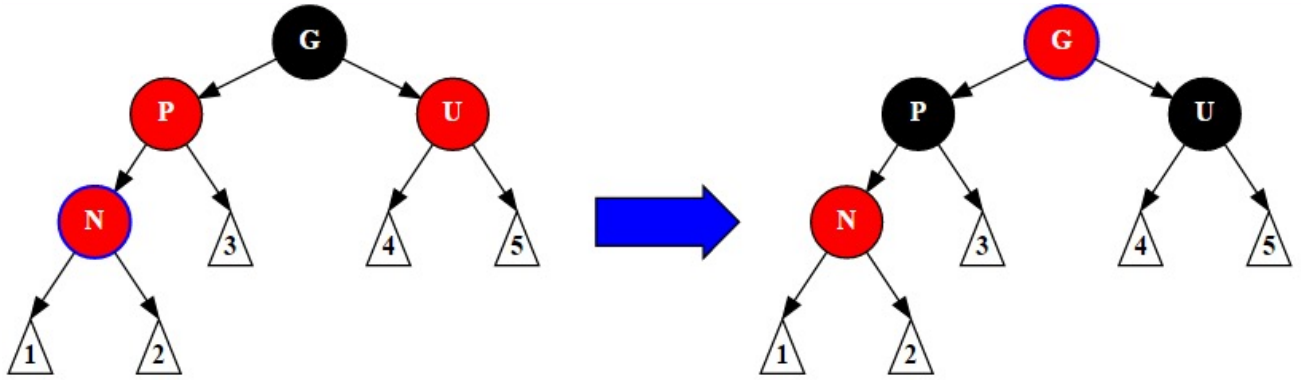
- The node inserted, N, is the root
- Violates Property 2 (root must be black)
- Change colour of N to black
- Otherwise, go to Case 2

Insertion Case 2

- If the parent of N, i.e., P, is black, then we are done; return
- Otherwise proceed to Case 3

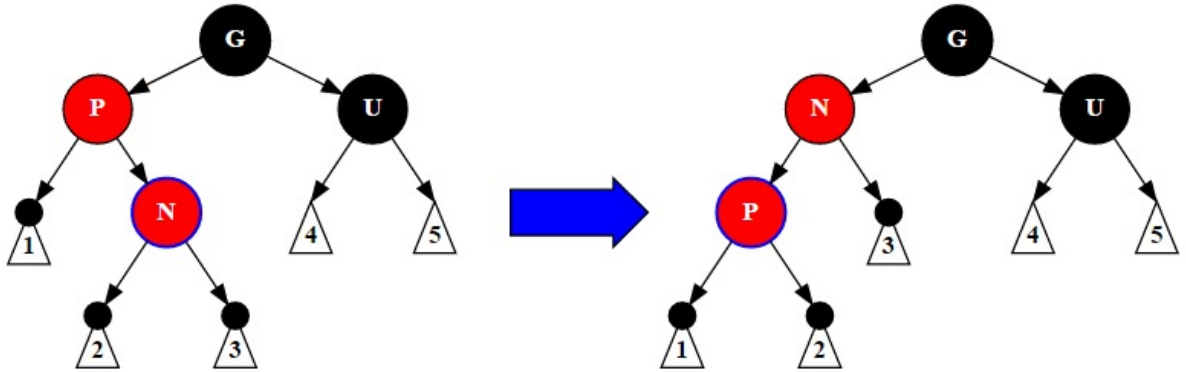
Insertion Case 3

- P is red, and U is not null and is red
- Change P and U to black and G to red, and go to Case 1
- Otherwise, go to Case 4



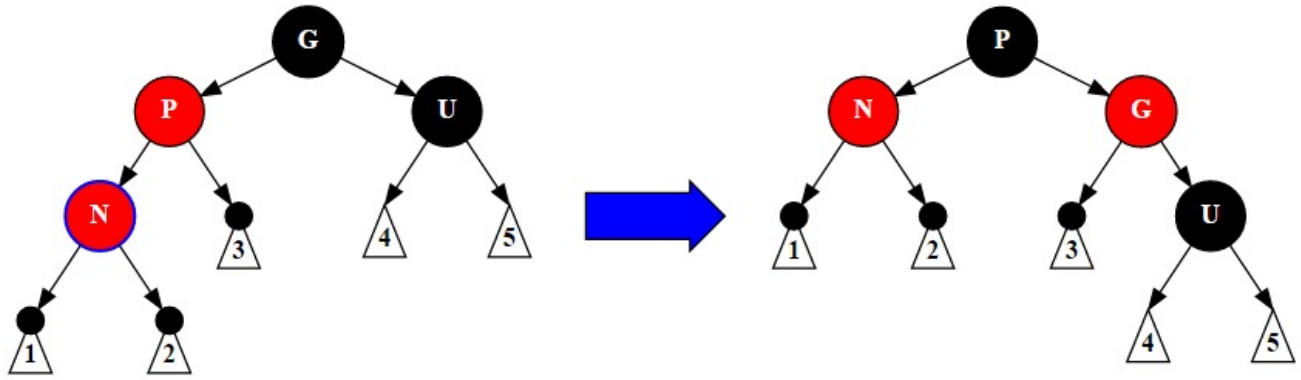
Insertion Case 4

- P is red but U is black
- If N is the right child of P, and P is the left child of G then remedy is left rotation on P
- If N is the left child of P, and P is the right child of G then remedy is right rotation on P
- (Must) go to Case 5



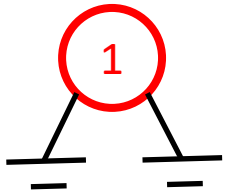
Insert Case 5

- P is red but U is black
- If N is left child of P, and P is left child of G, then remedy is right rotation on G
- Otherwise, rotate left on G
- Switch colours of P and G



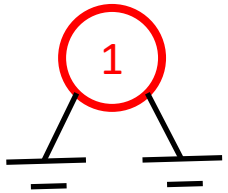
An Example

- Empty tree, insert 1



An Example

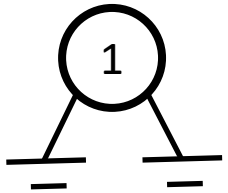
- Property 2 violated



- Recolour to black!

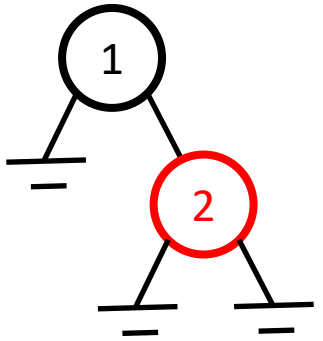
An Example

- Property 2 violated
- Recolour to black
- All properties satisfied



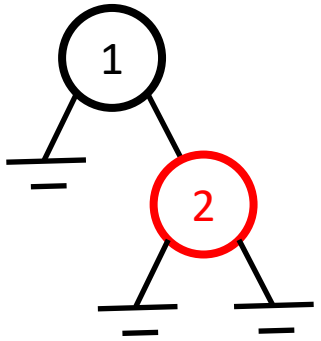
An Example

- Insert 2



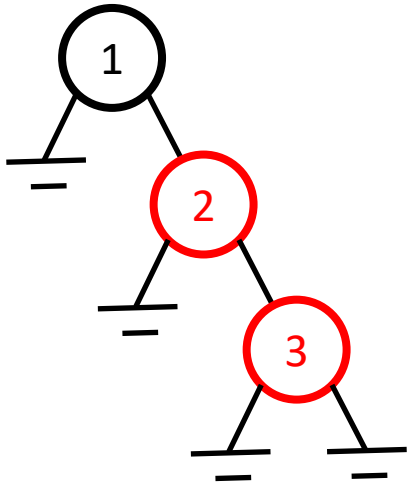
An Example

- All properties satisfied



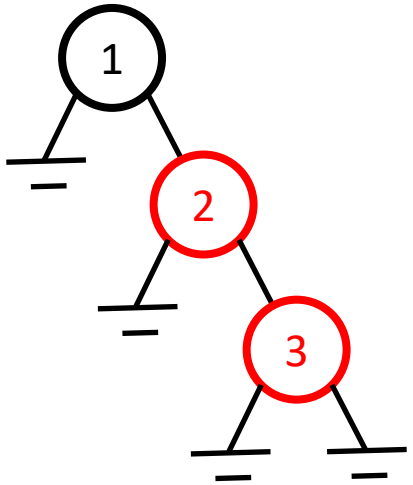
An Example

- Insert 3



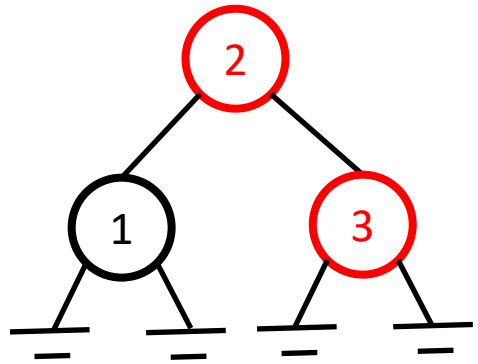
An Example

- Property 4 violated
- Case 5 – rotate left on grandparent



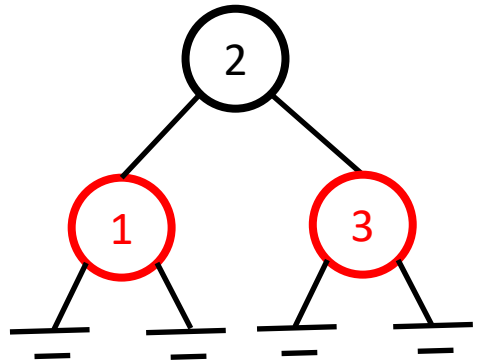
An Example

- Property 4 violated
- Case 5 – rotate left on grandparent



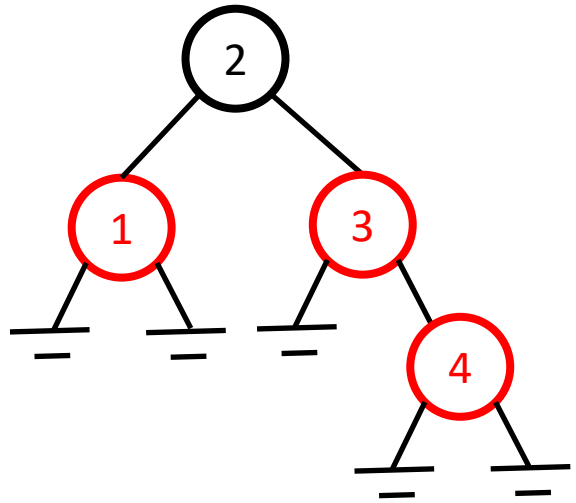
An Example

- Recolour



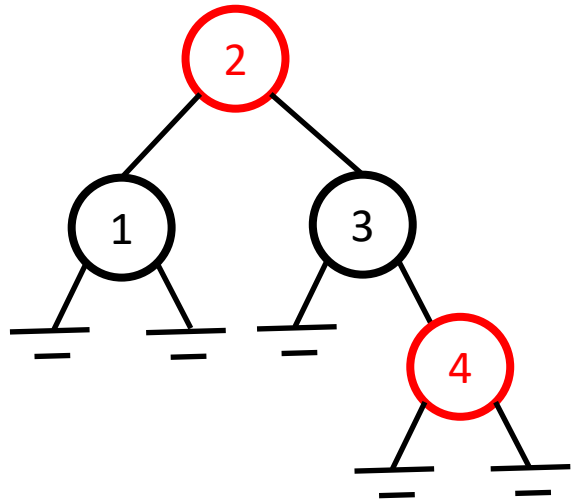
An Example

- Insert 4
- Property 4 violated



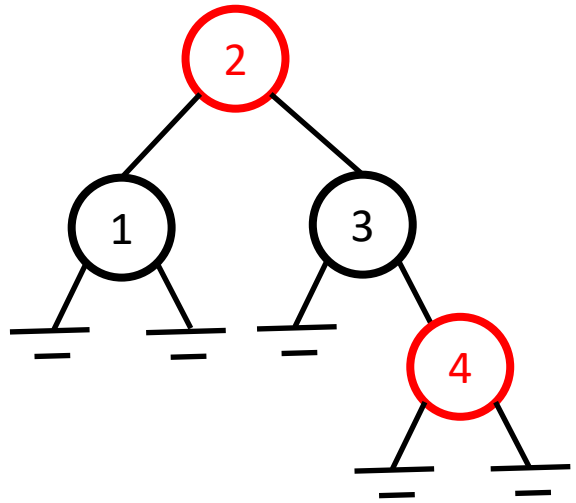
An Example

- Invoke Case 3



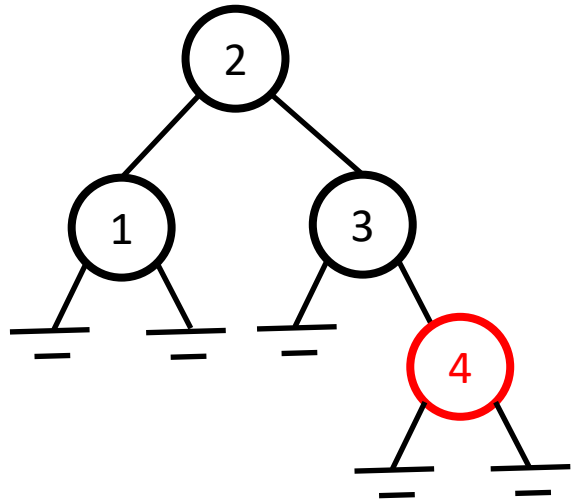
An Example

- Violates Property 2
- Recolour!



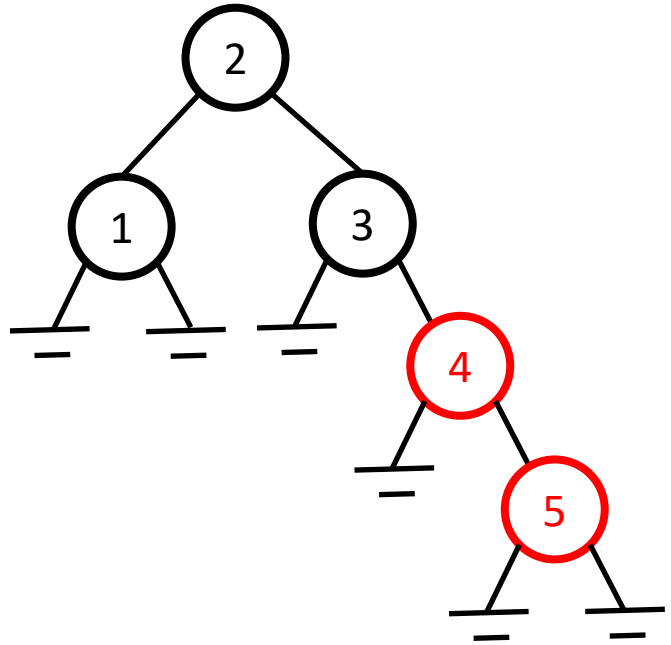
An Example

- Black depth is 3
- All properties satisfied



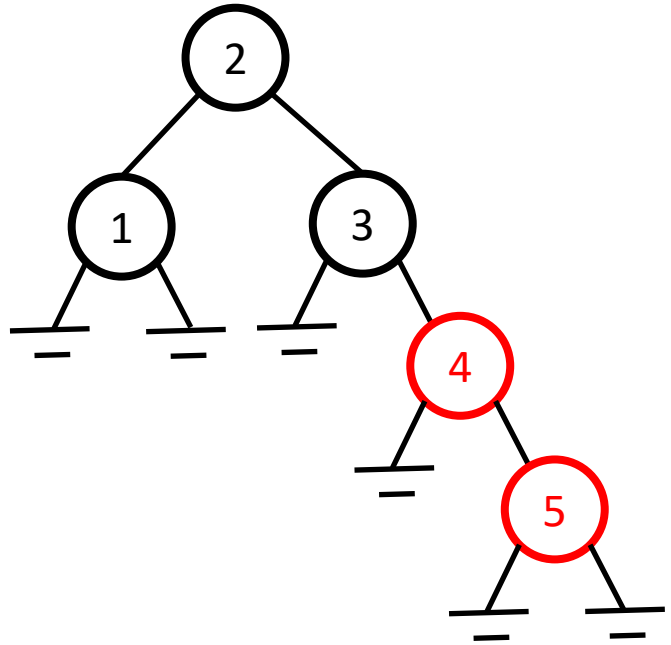
An Example

- Insert 5



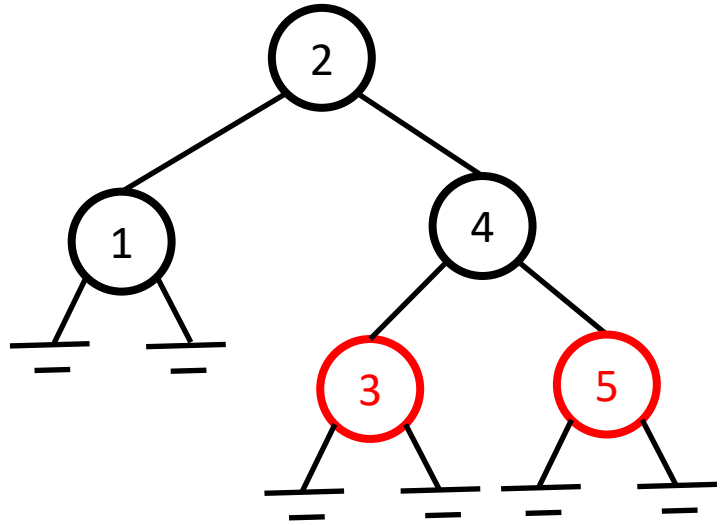
An Example

- Invoke Case 5
- Left rotate on grandparent 3
- And switch colour



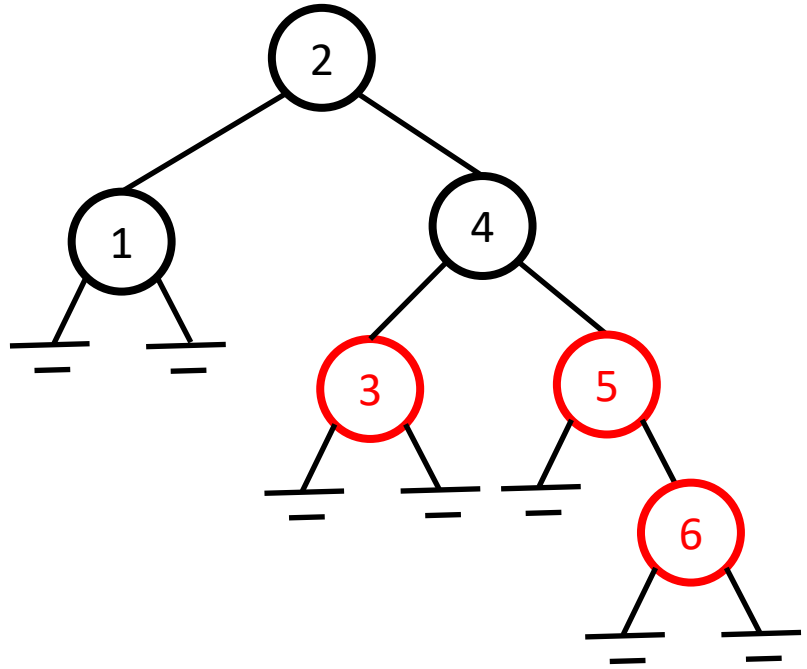
An Example

- All properties satisfied



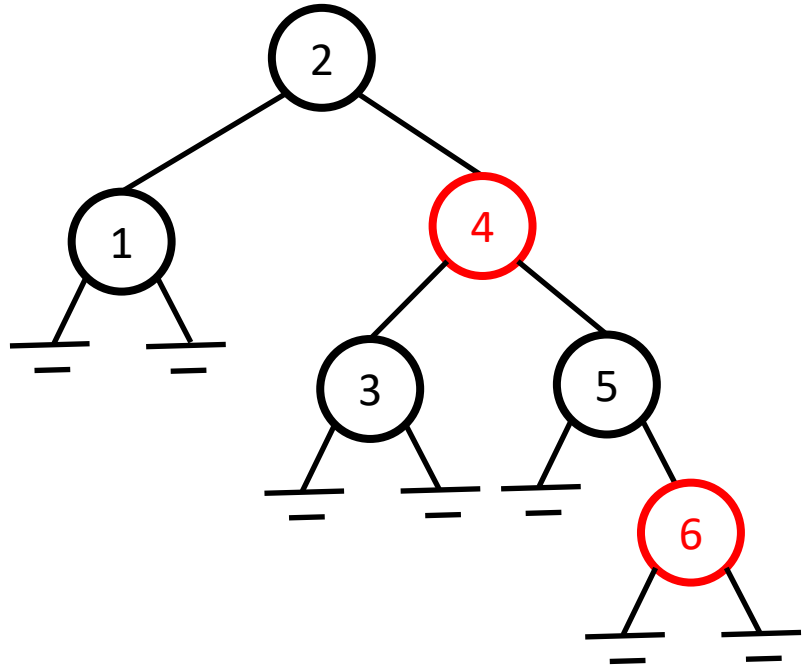
An Example

- Insert 6



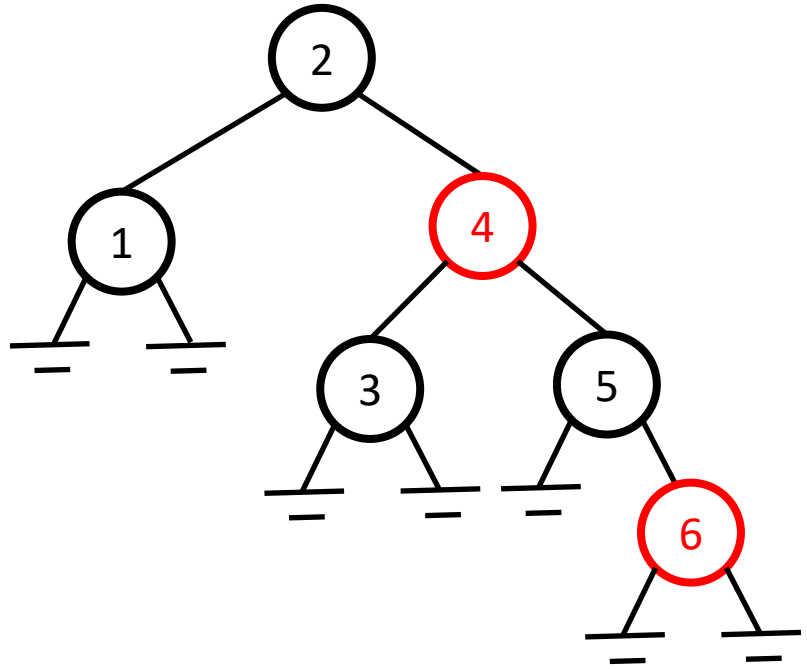
An Example

- Invoke Case 3 - recolour



An Example

- All properties satisfied



Deletion Operation in a Red-Black Tree

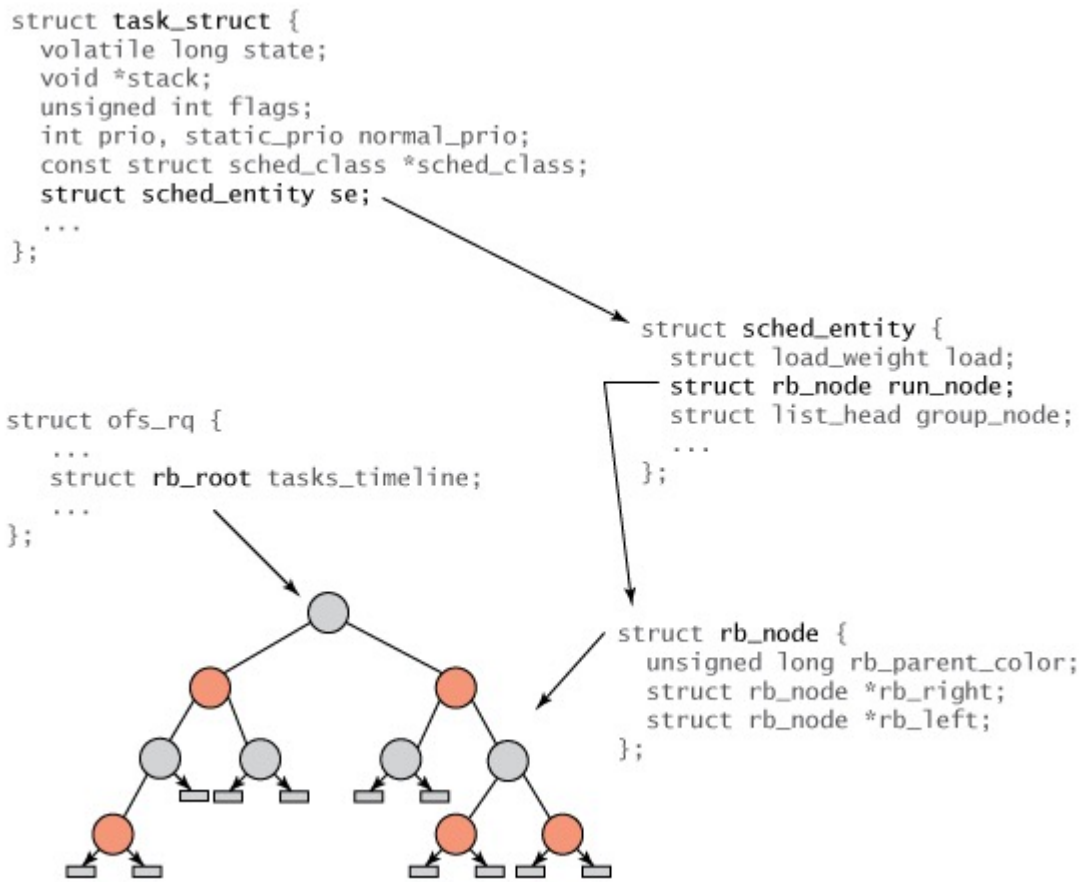
- Deletion is tedious involving recursion and more cases to check
  - Can trigger rebalancing in other parts of the tree
- We shall skip this
- If you are interested, many tutorials and description available online
- Now back to our Completely Fair Scheduler…

Back to CFS

- CFS uses a red-black tree of runnable tasks with **`vruntime`** as the key

- CFS picks the task with the smallest **`vruntime`** is the next task to run
  - It is always the leftmost non-null node of the red-black tree
  - This node is "cached" – pointed to by a special pointer; no need to do the O(log n) traversal to find

CFS Data Structure



Virtual Clock and `vruntime`

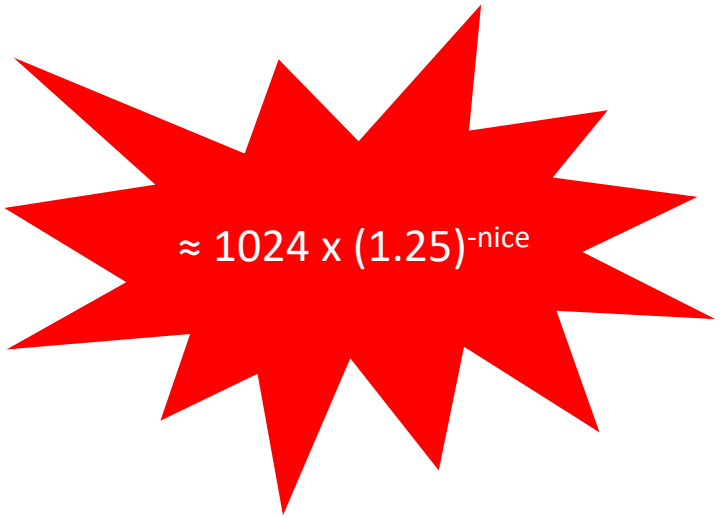
- A measure the amount of time a waiting task would have been allowed to run on a completely fair processor

- Calculated using actual time and a task's load weight
  - Uses high precision timer and timer interrupts
  - Priority implicitly accounted for
  - No per priority runqueue
  - There is a minimum to prevent thrashing when the runqueue is very large

Load weight

- The relation between niceness and timeslices of tasks is maintained by calculating load weights

```
static const int prio_to_weight[40] = {
    /* -20 */     88761,     71755,     56483,     46273,     36291,
    /* -15 */     29154,     23254,     18705,     14949,     11916,
    /* -10 */      9548,      7620,      6100,      4904,      3906,
    /*  -5 */      3121,      2501,      1991,      1586,      1277,
    /*   0 */      1024,       820,       655,       526,       423,
    /*   5 */       335,       272,       215,       172,       137,
    /*  10 */       110,        87,        70,        56,        45,
    /*  15 */        36,        29,        23,        18,        15,
};
```



- Two tasks, running by default at nice level 0 – weight 1024. Each of these tasks receives 50% of CPU time, because 1024/(1024+1024) = 0.5
- If one task now becomes nice -1, it should get around 10% CPU time more, and the other 10% less: 1277/(1024+1277) ≈ 0.55; 1024/(1024+1277) ≈ 0.45

The “10% effect”

- From *any* nice level,
  - Go up 1 level (priority lowered by 1): -10% CPU usage
  - Go down 1 level (priority raised by 1): +10% CPU usage

- Is relative and cumulative

Concept of a “period”

- Time is divided into periods.
- A period is a length of time in which every task ran at least once.

```
if #runnable tasks > some threshold (default 8)
        period = #runnable task * min_granularity (default 0.75ms)
 else
        period = base (default 6ms)
```

- The time slice given to each task is weighted by its weight

Time slice for a task = period * (weight of task) / (total weight of runqueue)

Time slices

- If there are many tasks, a low level task may get 0 time!
  - Guarantee: all tasks will get at least 0.75ms (`sched_min_granularity`) to run
  - Period will be lengthened appropriately to ensure this when necessary
- Number of runnable tasks may change
  - Period recomputed at every time slice computation
  - Time slice is computed only when a task is picked for execution next

How CFS works

- When the scheduler is called, the current task' **`vruntime`** is updated
  - Actual time the task ran for weighted as follows:

`vruntime +=` (time process ran) * (load weight niceness 0) / (load weight of this process)

Or in other words:

`vruntime +=` (time process ran) * 1024 / (load weight of this process)

- Overall virtual time tick is maintained in **`min_vruntime`** for the entire runqueue
  - The kernel takes vruntime of the leftmost element in the tree, if it exists, or that of a currently running task, depending on which is smaller.
- Next task is the leftmost node in the red-black tree, i.e., the one with the lowest **`vruntime`**, i.e., **`min_vruntime`**
- Note: because they wanted to avoid division as well as extend the arithmetic to 64 bit, the actual code in **`kernel/sched/fair.c`** contains some arithmetic voodoo.

How CFS works

- For new tasks, **`vruntime = min_vruntime`**

- For newly woken tasks
  - Before sleeping, compute and store delta between **`vruntime`** and current **`min_vruntime`**
  - After waking, recompute **`vruntime`** by adding delta to current **`min_vruntime`**
  - Prevents a task that had slept for a long time from hogging the processor

`vruntime` and nice value

- Nice value = 0: `vruntime` = real time spent by task
- Nice < 0: `vruntime` < real time spent by task and `vruntime` grows slower than real time
- Nice > 0: `vruntime` > real time spent by task and `vruntime` grows faster than real time

CFS Principles

- When a task is executing, its virtual run time increases, so it moves to the right in the red-black tree
  - Only updated during a `schedule()` call
- A compute intensive job would have run for a long time so it will move to the far right
- A I/O intensive job would have ran for a short period so it will move only a little to the right
- Virtual clock ticks more slowly for more important tasks

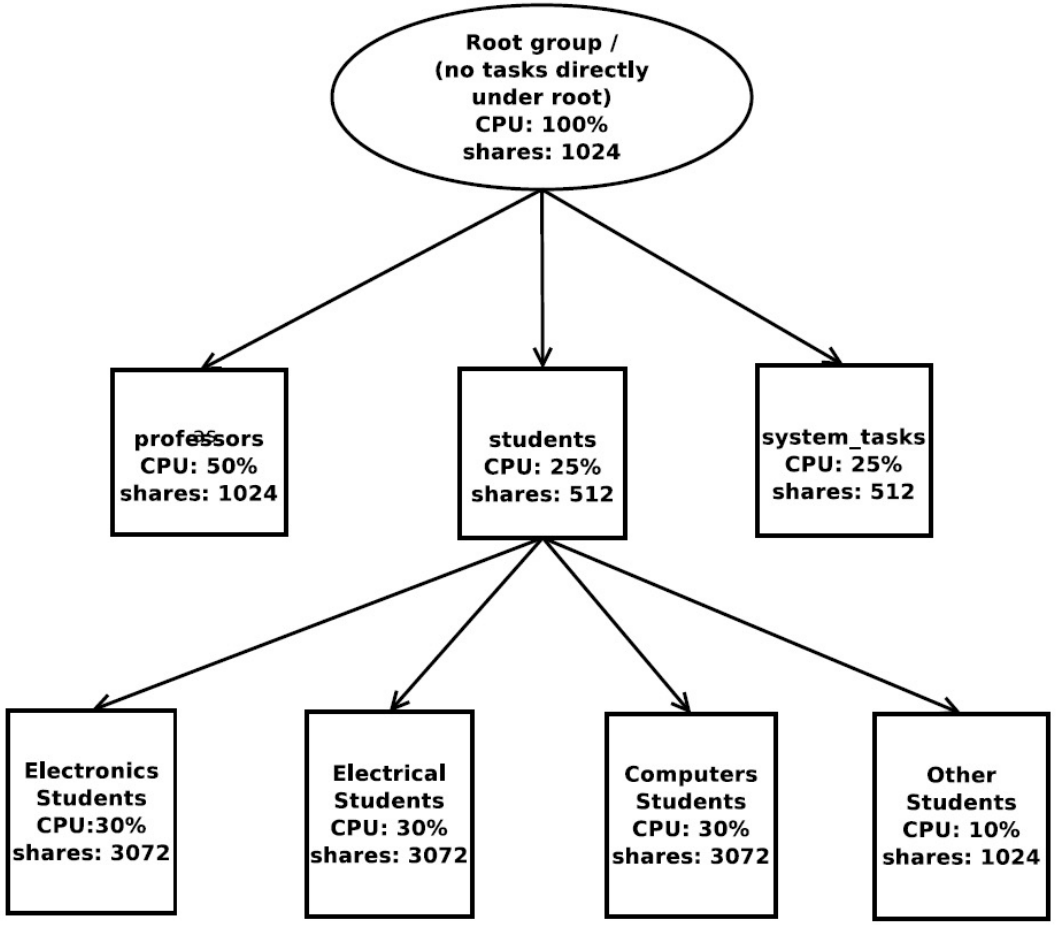
Small digression: floating point and the kernel

- There are many more floating point registers than general purpose ones
  - High overhead in saving
  - Kernel code not linked against the math library
- Hence, use of floating point computation in the kernel is discouraged
  - Still doable if you insists but with the proper API to save and restore
- Use (scaled) integer (a.k.a. fixed point) instead
  - Example: 6ms is stored as an integer 6000000
  - Use approximated integer calculation as in the scheduler

CFS Group Scheduling

- Starting from kernel version 2.6.24, CFS group scheduling was introduced
- Issue: Suppose there are 50 tasks, each would get 2% of CPU. But what if 48 of them are child tasks of a single task? Is it fair?
- Previously, a scheduling entity is a task. But in group scheduling, it can be many tasks.
- Time pie is divided among scheduling entities – not tasks directly
  - A.k.a. fair share scheduling
  - Can be task hierarchy, user group, interactive vs. non-interactive etc.

Use of shares



CFS Bandwidth Control

- Optional feature of CFS
- Allows user to specify a limit to the CPU time given to a group or process hierarchy

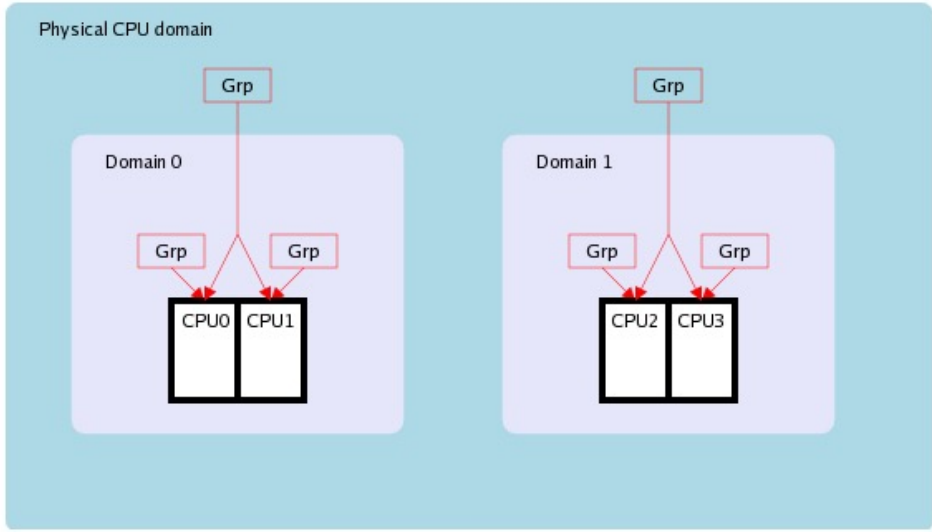
SMP Load Balancing

- Each CPU has its own runqueue (red-black tree)

- To tackle imbalance, tasks are moved from busy runqueues to less busy ones occasionally
  - Tries to balance out the total weight (load) of the runqueues

Scheduling Domain

- Multi-CPU machines may be NUMA
  - A CPU may be “closer” to some and “farther” from others
- Bad scenario: A cooperating group of tasks is scheduled “all over”
- Scheduling domains was introduced to account for CPU asymmetry



Load balancing with domains

- Try to locate an entire group in one domain

- Rebalancing of CPU workload has to take this into consideration
  - Need to aggregate the scheduling group's demand
  - Need to estimate target domain's capacity
  - Do not want rebalancing to cause new imbalances!

Con’s Response

Brain F*ck Scheduler

- In August 2009, Con Kolivas introduced BFS

- Aimed desktop Linux
  - Moderate number of CPUs
  - Make things simple, hence low overhead

BFS Global Runqueue

- A single runqueue for the entire system
  - Has a lock
  - A niffies variable (jiffies in nanoseconds) – “now”
- The runqueue consists of 103 priority queues
  - Each are doubly linked lists
  - First 100 are dedicated to real-time tasks
  - 101st is for SCHED_ISO (isochronous)
  - 102nd for SCHED_NORMAL
    - All “normal” tasks regardless of priority go into this queue
  - 103rd for SCHED_IDLEPRIO (idle priority scheduling)

Virtual Deadline

- Each runnable task gets a timeslice of **`rr_interval`** and a virtual deadline
  - **`rr_interval`** default is 6ms
  - Timeslice is same for all tasks of all priorities

- Virtual deadline = **`niffies`** + **`prio_ratios[priority]`** × **`rr_interval`** × scaling_factor
  - **`prio_ratios[priority]`** is an array of constants to scale the timeslice by the priority

Virtual deadline

- The virtual deadline formula (given by Con) is:

$$T = 6$$
$$N = 1 << 20$$
$$d(n, t) = t + g(f(n)) * T * (N / 128)$$

$N = 10^9 / 10^3$ (nano/milli)
$= 1{,}000{,}000$

Scaling factor $= N / 128$
$= 1{,}000{,}000 / 128$
$= 7{,}812.5$

Approximated by 8192 (so that arithmetic can use shifts instead of multiplication)

- $d(n, t)$ – virtual timeline in unsigned 64 bit integer
- n – nice value
- t – current time in nanoseconds
- $g(i)$ is the prio ratio table lookup
- $f(n)$ is the task's nice to index mapping function
- T is the round robin time slice in milliseconds
- N is a constant of 1 millisecond in terms of nanosecond approximated by the nearest power of 2 value

The parameters

- "Scaling factor" is 8192
  - `rr_interval` is in millisecond while `niffies` are in nanosecond, hence the need for scaling. So need to scale by 1,000,000.
  - Need to divide by 128 is to reduce the magnitude of `prio_ratio`, so that at niceness -20, it will be effectively 1.
  - 1,000,000 / 128 = 7,812.5. To optimize the computation, round to 8192 and use shifts instead of multiply.
- "priority" maps -20 to +19 to 0 to 39 for array indexing
- "prio_ratios" starts with 128 for niceness -20, scaled by 1.1 (with rounding to integer) thereafter.

```
prio_ratios[-20] = 128
prio_ratios[-19] = 141
prio_ratios[-18] = 155
prio_ratios[-17] = 171
prio_ratios[-16] = 188
prio_ratios[-15] = 207
prio_ratios[-14] = 228
prio_ratios[-13] = 251
prio_ratios[-12] = 276
prio_ratios[-11] = 304
prio_ratios[-10] = 334
prio_ratios[-9]  = 367
prio_ratios[-8]  = 404
prio_ratios[-7]  = 444
prio_ratios[-6]  = 488
prio_ratios[-5]  = 537
prio_ratios[-4]  = 591
prio_ratios[-3]  = 650
prio_ratios[-2]  = 715
prio_ratios[-1]  = 787
prio_ratios[0]   = 866
prio_ratios[1]   = 953
prio_ratios[2]   = 1048
prio_ratios[3]   = 1153
prio_ratios[4]   = 1268
prio_ratios[5]   = 1395
prio_ratios[6]   = 1535
prio_ratios[7]   = 1689
prio_ratios[8]   = 1858
prio_ratios[9]   = 2044
prio_ratios[10]  = 2248
prio_ratios[11]  = 2473
prio_ratios[12]  = 2720
prio_ratios[13]  = 2992
prio_ratios[14]  = 3291
prio_ratios[15]  = 3620
prio_ratios[16]  = 3982
prio_ratios[17]  = 4380
prio_ratios[18]  = 4818
prio_ratios[19]  = 5300
```

How it works

- Task exhausts timeslice:
  - task taken off the CPU
  - timeslice refilled
  - deadline recomputed
- When a task sleeps:
  - Timeslice and deadline are untouched
- Preemption by a newly woken task having higher priority or an earlier deadline than that of a currently running task's:
  - New task takes over the processor

BFS finding the next task

- Use the priority bit map trick to see what is the first queue that is non-empty

- Then loop through the queue to find the task with the earliest deadline
  - New items are simply inserted to the end of the queue
  - Hence, O(*n*) to find an item

Isochronous and Idleprio scheduling

- Isochronous scheduling (**`SCHED_ISO`**) designed to provide users without super user privileges with performance close to real-time
  - Priority higher than normal
  - A task requesting for realtime priority but lacks the privilege will be bumped to **`SCHED_ISO`**
  - A **`SCHED_ISO`** task needs superuser privilege to go back to normal
  - Scheduled in round robin with **`rr_interval`** timeslice

- Idleprio for really low priority work – when system is idle
  - Example: run SETI@home

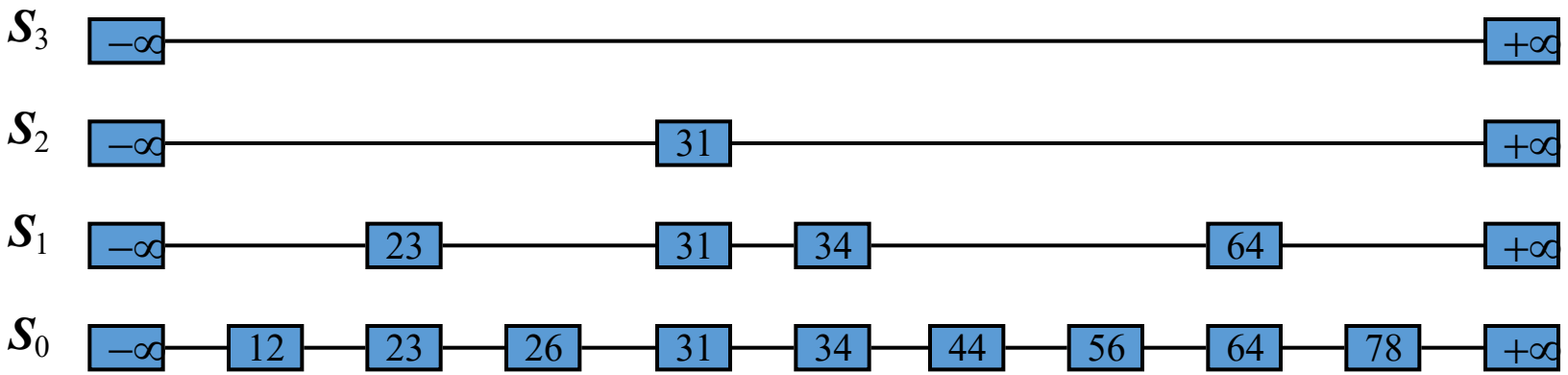
Skip List

A Digression

Source: M.T. Goodrich and R. Tamassia, UCI

What is a Skip List

- A skip list of level h for a set S of distinct items is a series of lists $S_0, S_1, \ldots, S_h$ such that
  - Each list S_i contains the special keys $+\infty$ and $-\infty$
  - List S_0 contains the keys of S in non-decreasing order
  - Each list is a subsequence of the previous one, i.e.,
    $$S_0 \supseteq S_1 \supseteq \ldots \supseteq S_h$$
  - List S_h contains only the two special keys
- An example:



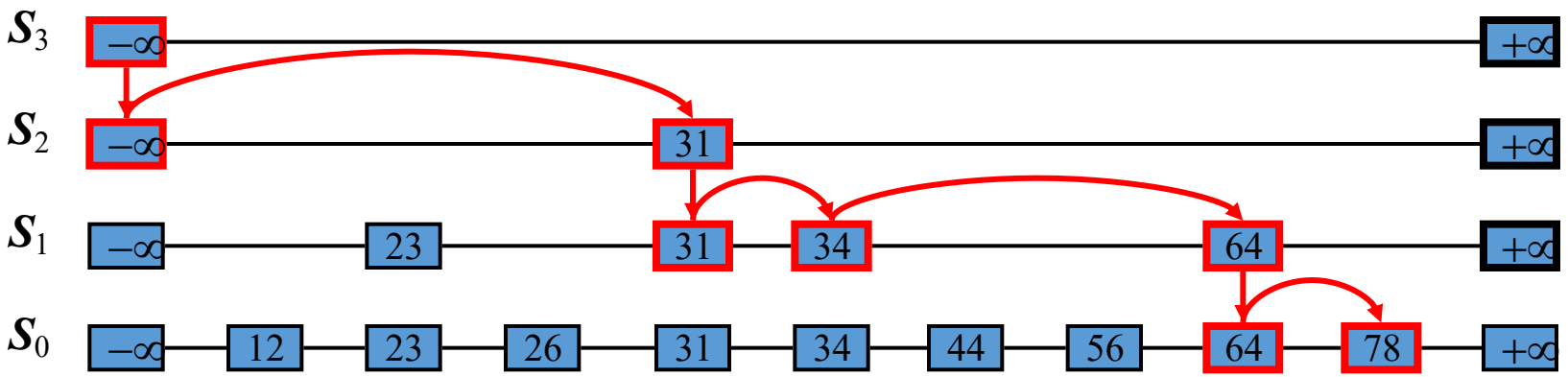
Search

- We search for a key x in a skip list as follows:
  - We start at the first position of the top list
  - At the current position p, we compare x with $y \leftarrow key(next(p))$

    $x = y$: we return $element(next(p))$

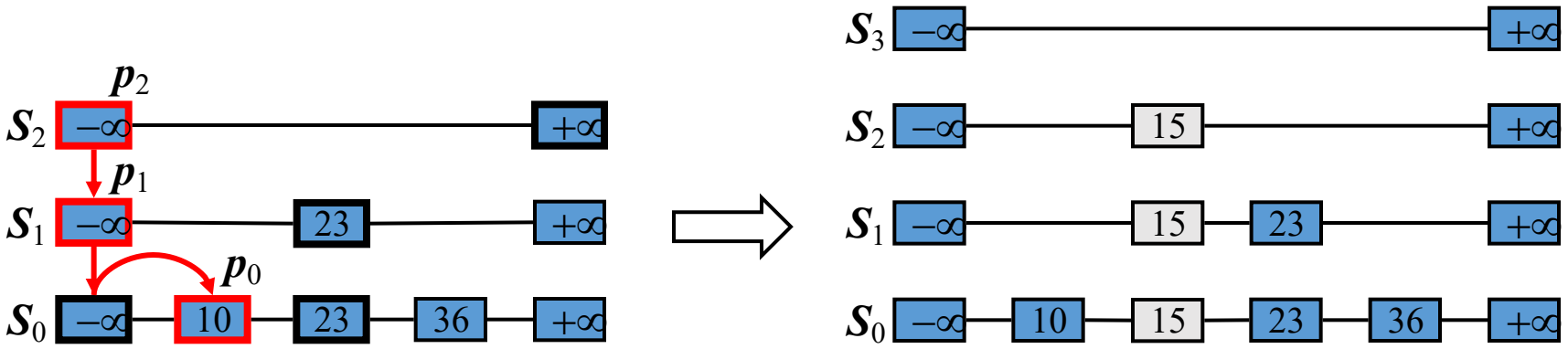
    $x > y$: we "scan forward"

    $x < y$: we "drop down"
  - If we try to drop down past the bottom list, we return ***null***
- Example: search for 78



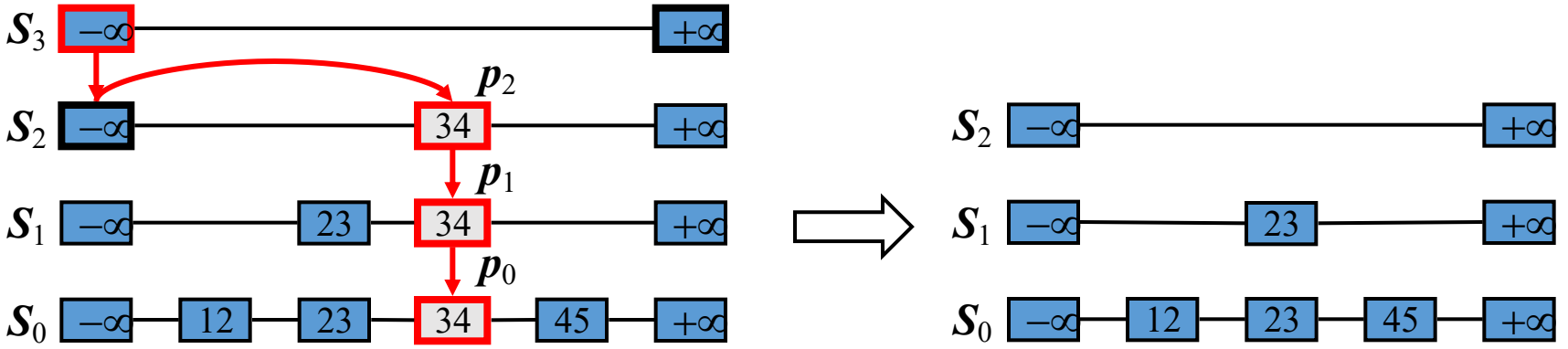
Insertion

- To insert an entry (x, o) into a skip list, we use a randomized algorithm:
  - We repeatedly toss a coin until we get tails, and we denote with i the number of times the coin came up heads
  - If $i \geq h$, we add to the skip list new lists $S_{h+1}, \ldots, S_{i+1}$, each containing only the two special keys
  - We search for x in the skip list and find the positions $p_0, p_1, \ldots, p_i$ of the items with largest key less than x in each list $S_0, S_1, \ldots, S_i$
  - For $j \leftarrow 0, \ldots, i$, we insert item (x, o) into list S_j after position p_j
- Example: insert key 15, with $i = 2$



Deletion

- To remove an entry with key x from a skip list, we proceed as follows:
  - We search for x in the skip list and find the positions $p_0, p_1, \ldots, p_i$ of the items with key x, where position p_j is in list S_j
  - We remove positions $p_0, p_1, \ldots, p_i$ from the lists $S_0, S_1, \ldots, S_i$
  - We remove all but one list containing only the two special keys
- Example: remove key 34



Complexity

- The expected space usage of a skip list is $O(n)$
- A skip list with n entries has height at most $3\log n$ with probability at least $1 - 1/n^2$
- Expected times for search, insertion, deletion all of $O(\log n)$

Improvements to BFS

- In Sep 2011, Con Kolivas announced major changes to BFS
- No more priority map and doubly-linked list
- Instead global runqueue is a single skip list sorted by virtual deadlines
- First task in the "bottom list" is the highest priority
  - Lookup is $\mathrm{O}(1)$
- Other tweaks yielded $\mathrm{O}(\log n)$ insertion, $\mathrm{O}(1)$ lookup, and $\mathrm{O}(k)$ removal
  - k is the height of the node in question, up to 16, but usually only 1-4

Multiple Queue Skiplist Scheduler (MuQSS)

- Released by Con Kolivas in Oct 2016
- Problems with (original) BFS:
  1. The single runqueue results in serious lock contention
  2. O(n) lookup
- Skiplist BFS solved 2
- To solve 1, go back to one runqueue per CPU

MuQSS

- Goal: Good desktop interactivity and responsiveness
- Earliest Eligible Virtual Deadline First (EEVDF) policy
- Multiple per-CPU runqueues
  - O(log n) insertion
  - O(1) lookup

Skiplists in MuQSS

- Each processor has one runqueue
  - Each runqueue has one lock
- Each runqueue has exactly 8 levels
  - Array of pointers to each level fits cache line size
  - Bidirectional skiplists
  - Can handle up to 256 tasks
  - Total number of tasks = 256 * number of logical CPUs

Virtual deadlines

- "Niffies": monotonic forward moving timer of nanosecond resolution
  - Implemented using high resolution TSC timers
  - Calculated per-CPU
  - Synchronous across CPUs
- `rr_interval`: the longest duration two tasks of the same nice level will be delayed for
- `prio_ratio`: a ratio compared to the baseline of nice -20 and increases by 10% per nice level
- The **virtual deadline** is offset from the current time in niffies by this equation:

```
niffies + (prio_ratio * rr_interval)
```

Selecting a task

1. **`time_slice`** expiration: If a task runs out of its **`time_slice`**, it is descheduled, the **`time_slice`** is refilled, and the deadline reset to that formula
2. Sleep: **`time_slice`** and deadline not adjusted, carried over to the next time it is scheduled
3. Pre-emption: newly waking task is deemed higher priority than a currently running task on any CPU by virtue of the fact that it has an earlier virtual deadline than the currently running task

Operations

- Task insertion: O(log n)
  - Ordering:
    1. first by static priority, then
    2. by virtual deadlines
- Task lookup
  - Two modes: interactive and !interactive

`interactive` mode task lookup

1. Lockless read access to all CPU runqueues to pick one for possible scheduling

2. Suppose a candidate is found in CPU x's runqueue. Perform a "trylock" on the chosen runqueue. If fail, go to 1.

3. If succeed, schedule that entry on oneself.

`!interactive` mode task lookup

- Similar to **`interactive`** mode BUT
  - Only check runqueues of other CPU's that are more heavily loaded than the current CPU
  - If current CPU most heavily loaded, look only at ownself runqueue

In summary

- CFS regulates access by giving proportionate time to tasks according to priorities

- BFS and variants implements priority via pre-emption and the frequency of a task's access to the CPU

The End