Lecture # 4.5
Linux Process Scheduler
O(1) / CFS

Course: Advanced Operating System

Instructor: Arif Butt

Punjab University College of Information Technology (PUCIT)
University of the Punjab

Source Code files available at: `https://bitbucket.org/arifpucit/spvl-repo/src`
Lecture Slides available at: `http://arifbutt.me`

Agenda

- Overview of CPU Scheduling
- UNIX SVR3 CPU Scheduler
- Linux O(1) CPU Scheduler
- Linux CFS CPU Scheduler
- Linux **`schedtool`**
- Scheduling related system calls

CPU Scheduler

- Scheduling is a matter of managing queues to minimize queueing delay and to optimize performance in a queueing environment
- The process scheduler in a multitasking operating system is a kernel component that decides which process runs, when and for how long

A multitasking OS comes in two flavors:

- In **Preemptive multitasking**, the scheduler decides when a process is to cease running (e.g., time slice expires) and a new process is to begin running. On many modern OSs, the time slice is dynamically calculated as a fraction of process behavior and configurable system policy
- In **Cooperative multitasking**, a process does not stop running until it voluntary decides to do so. (e.g., Mac OS 9 and earlier, Windows 3.1 and earlier)

Preemptive vs Non-Preemptive Kernels

At any instant of time a system can either be executing in user mode (executing LOCs written by programmer) or kernel mode (executing LOCs written by the kernel developer). A process can be inn kernel mode in

A) process context (a system call made by programmer)

B) interrupt context

The three types of OS kernel are:

- **Preemptive Kernel**, a kernel that can be preempted both in A and B
- **Reentrant Kernel**, a kernel that can be preempted in A only
- **Nonpreemptive Kernel**, a kernel cannot be preempted

Types of Processes

- When speaking about process scheduling, processes are traditionally classified into three different classes:
  - **Interactive Processes:** These interact constantly with their users. When input is received, the average delay must fall between 50-150 ms, otherwise the user will find the system to be unresponsive. Typical interactive programs are command shells, text editors and graphical applications
  - **Batch Processes:** These do not need user interaction and often execute in the back ground and are often penalized by the scheduler. Typical batch programs are programming language compilers, database search engines and scientific computations
  - **Real-time Processes:** These processes should have a short guaranteed response time with a minimum variance. Typical real-time programs are multimedia applications, robot controllers, and programs that collect data from physical sensors

Types of Processes (cont...)

- **I/O-bound processes** spend much of their time submitting and waiting on I/O requests, e.g., waiting on user interactions via the keyboard and mouse (Text editors)
- **Processor-bound processes** spend much of their time executing code. The ultimate example is a process executing an infinite loop, or a video encoder
- These two classifications are not mutually exclusive, as processes can exhibit both behaviors simultaneously, e.g., a word processor doing spell checking or macro calculations

Optimization Criteria for Process Scheduler

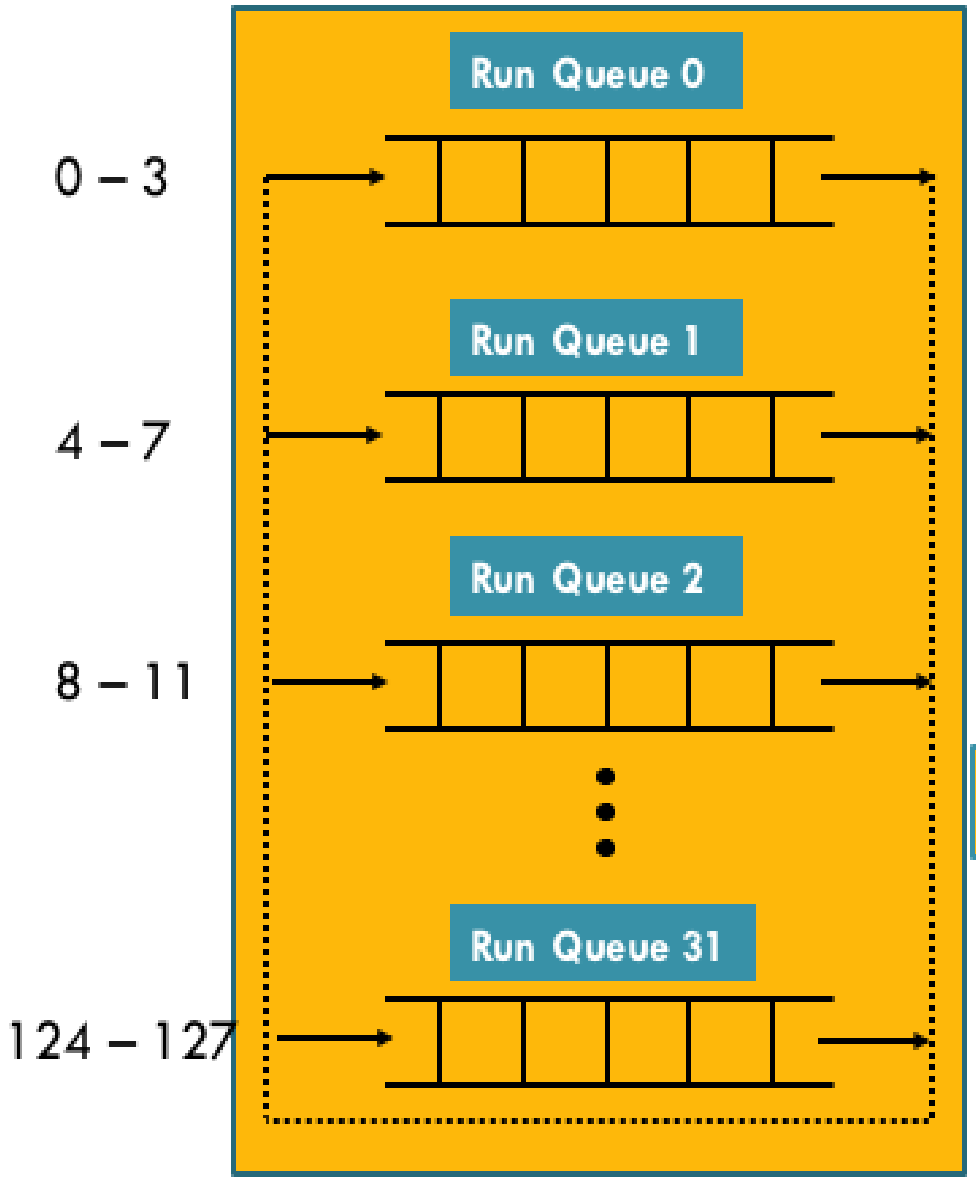
- Maximize CPU utilization
- Maximize throughput
- Maximize fairness
- Minimize waiting time
- Minimize response time
- Minimize turn around time

Recap of Process Scheduling Algorithms

	FCFS	SJF/SRTF	HRRN	Priority Based	Round Robin	MLFQ	RSDL
Select Function	max (w)	min(s)/ min(s-e)	$\max(\frac{w+s}{s})$	min(p)	const	-	-
Pre-emptive	No	N/Y	No	Yes	Yes	Yes	Yes
Starvation	Possible	Possible	No	Possible	No	Possible	No
Throughput	-	Hi	Hi	-	H/L	-	Hi
Response time	Hi	Good	Good	Hi	Good	-	Hi
Overhead	Min	Hi	Hi	Hi	Min	Hi	Hi
Effect on processes	Penalize short and I/O bound	Penalize long process	Good balance	-	Penalize I/O bound	May favor I/O bound	Good balance

Traditional UNIX SVR3 Scheduler

Traditional UNIX Scheduler (cont...)



- 128 Priority values
  - 0–49: Kernel
  - 50–127: User level programs

$$usrpri_i(i) = Base_i + cpu_i(i) + nice_i$$

Where $Base_i = 50$

$cpu_i(i) = DR * cpu_i(i-1)$

$nice_i$ = -20 to +19

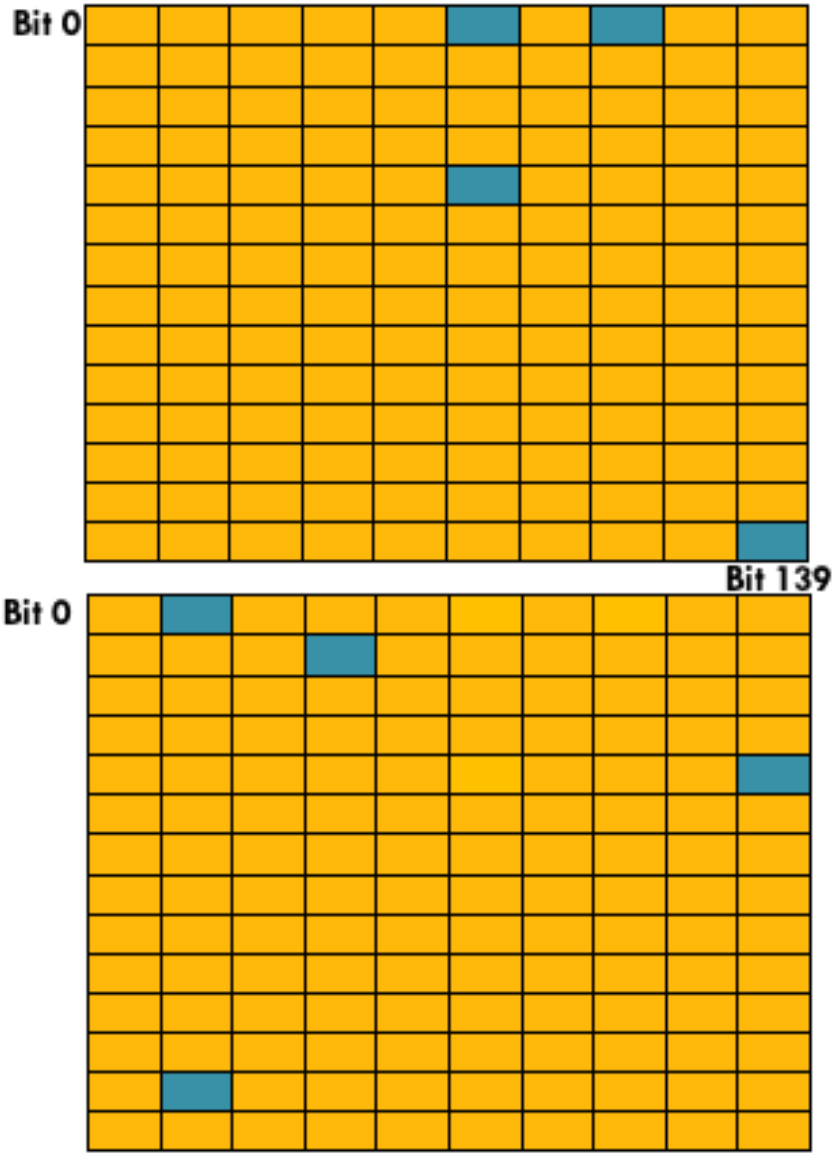
Traditional UNIX Scheduler (cont...)

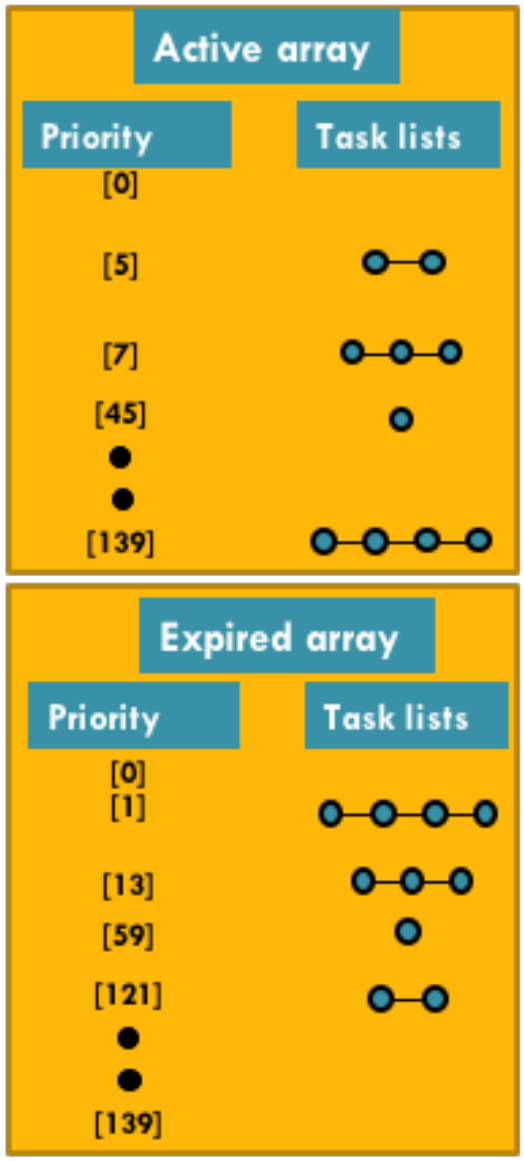
Limitations

- With large number of processes, overhead of re-computing process priorities every second is very high
- Since the kernel itself is non-preemptive, high priority processes may have to wait for low priority processes executing in kernel mode

Linux O(1) Scheduler

Linux O(1) Scheduler






```
struct runqueue{
  long nr_running;
  struct prio_array *
       active;
  struct prio_array *
       expired;
  int static_prio;
  int sleep_avg;
   - - - -
}
```

```
struct prio_array{
  int nr_active;
  long bitmap[S];
  struct list_head
queue[MAX_PRIO];
}
```

Linux O(1) Scheduler (cont...)

- Dynamic Priority:

$$DP = \max(100, \min(SP - bonus + 5, 139))$$

Avg Sleep Time (ms)	Bonus / **sleep_avg**
0 <= s < 100	0
100 <= s < 200	1
200 <= s < 300	2
300 <= s < 400	3
…	---
1000	10

```
struct task_struct{
   ---
   unsigned long sleep_avg;
   ---
}
```

- Heuristic to determine interactive process

$$Bonus - 5 \quad >= \quad SP/4 - 28$$

Linux O(1) Scheduler (cont...)

- Time slice calculation of a process

Time Slice	Duration	Interactivity	Nice Value / SP
Initial	Parent's Half	N/A	Parent's
Minimum	10 ms	Low	High
Default	100 ms	Average	Zero
Maximum	200 ms	High	Low

- CPU Affinity
  - Soft CPU affinity
  - Hard CPU affinity

```
struct task_struct{
   ---
   cpumask_t cpus_allowed;
   ---
}
```

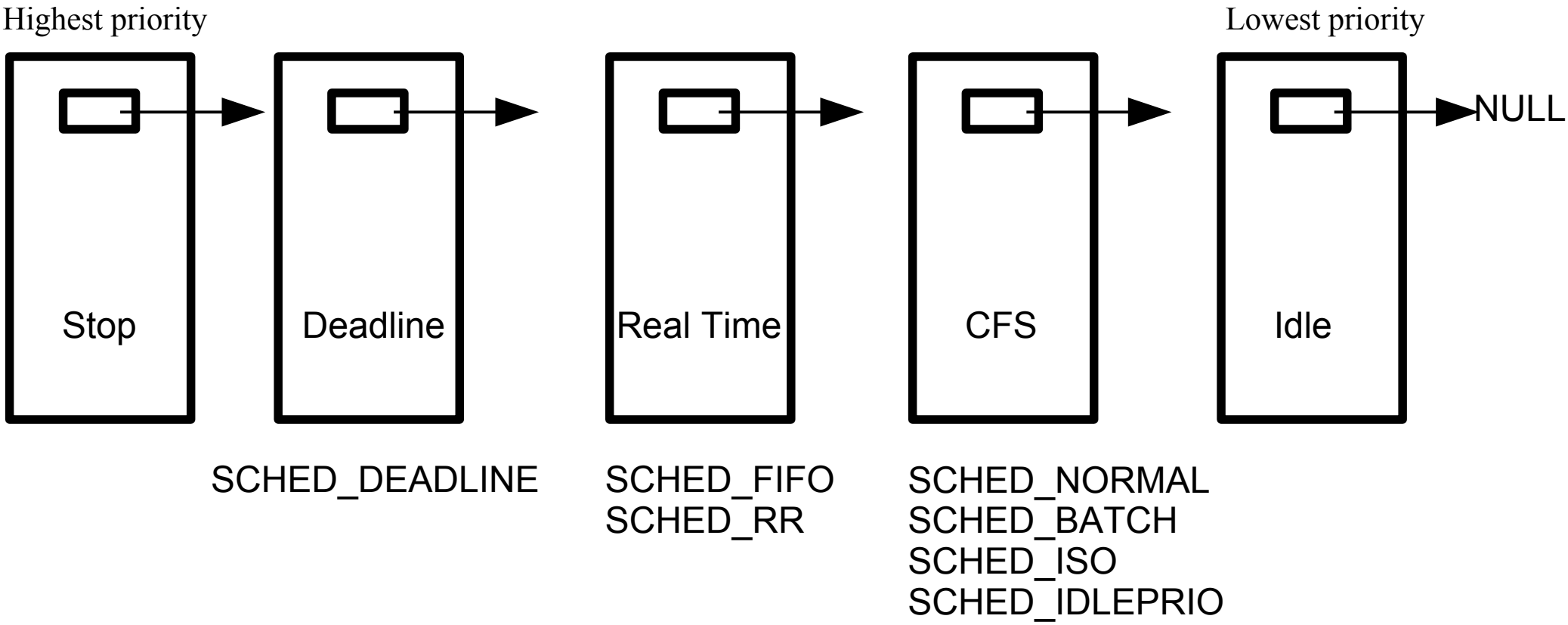
Linux Kernel Scheduler (cont...)

Limitations of O(1) Scheduler

- It uses complex heuristics to determine if a process is I/O bound or CPU bound to benefit one over the other
- Lot of code to manage priority queues, at least 140 per processor

Linux CFS Scheduler
Completely Fair Share Scheduler

Scheduling Classes and Scheduling Policies

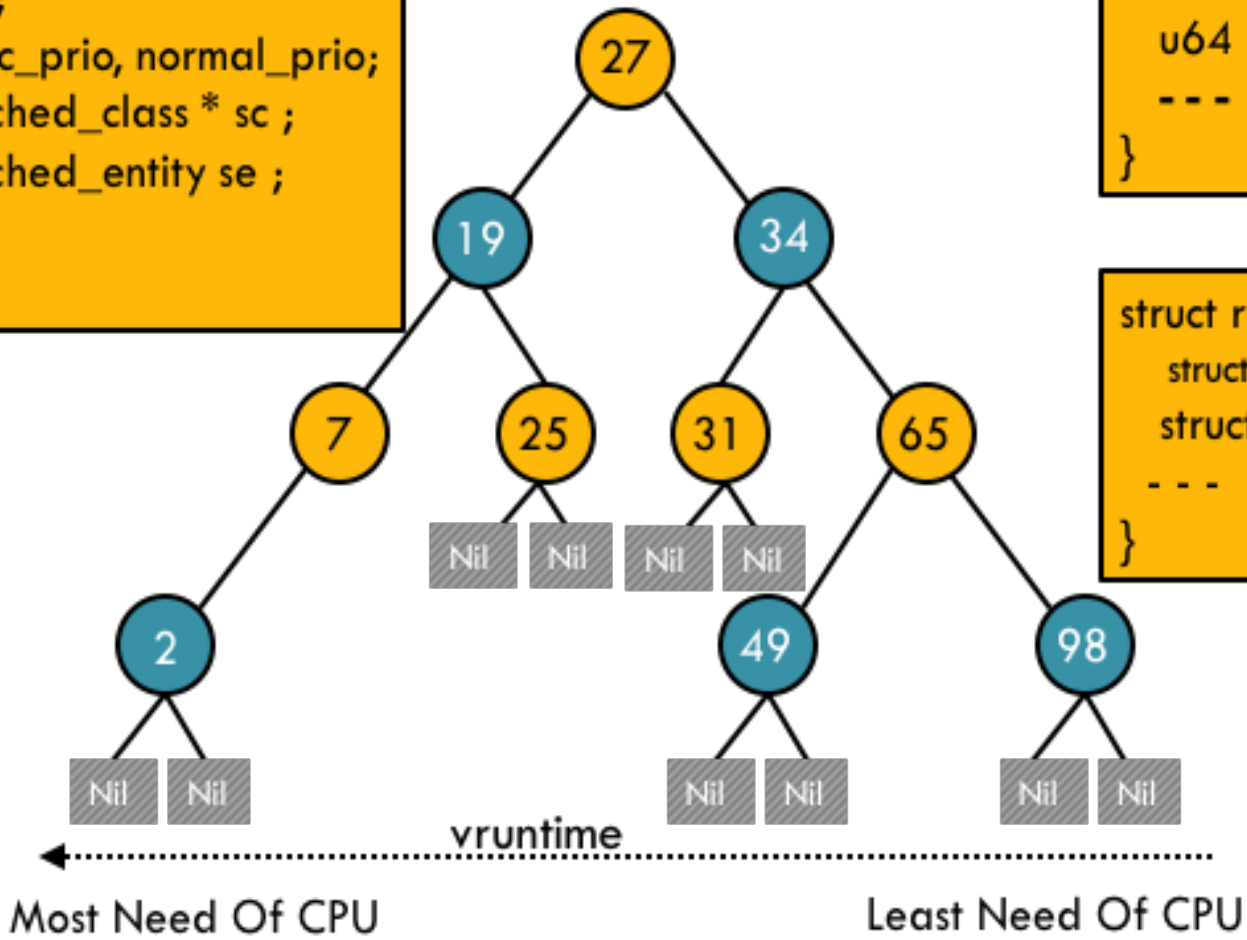


Linux CFS Scheduling Class

```
struct task_struct{
   volatile long state;
   int prio;
   int static_prio, normal_prio;
   struct sched_class * sc ;
   struct sched_entity se ;
   - - -
}
```

```
struct sched_entity{
   struct load_wait  load ;
   struct rb_node    rn ;
   u64   vruntime;
   - - -
}
```

```
struct rb_node{
    struct rb_node * rb_left ;
   struct rb_node * rb_right;
  - - -
}
```



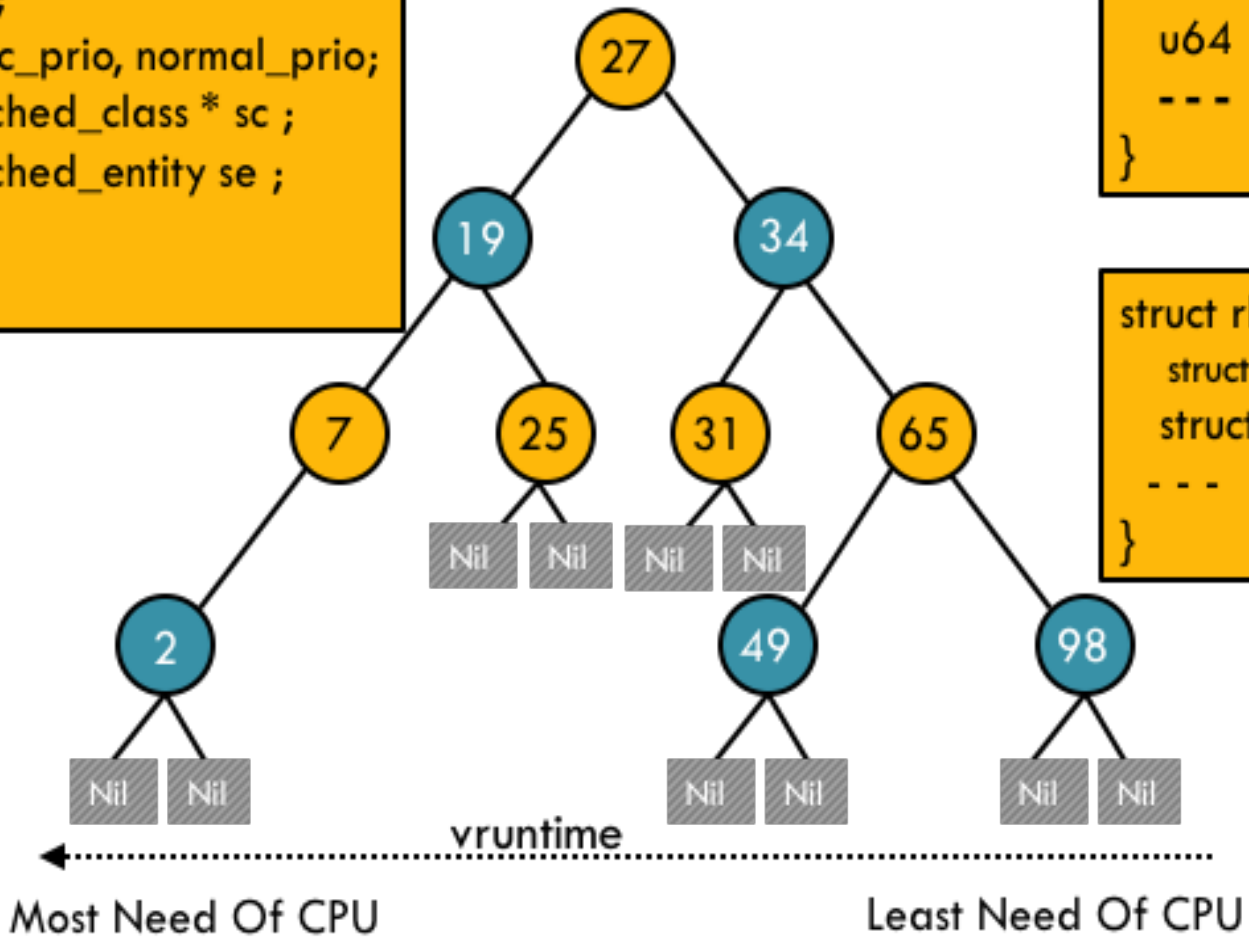
Linux CFS Scheduling Class

Context Switch and Time Slice

```
struct task_struct{
   volatile long state;
   int prio;
   int static_prio, normal_prio;
   struct sched_class * sc ;
   struct sched_entity se ;
   - - -
}
```

```
struct sched_entity{
   struct load_wait  load ;
   struct rb_node    rn ;
   u64   vruntime;
   - - -
}
```

```
struct rb_node{
    struct rb_node * rb_left ;
   struct rb_node * rb_right;
  - - -
}
```



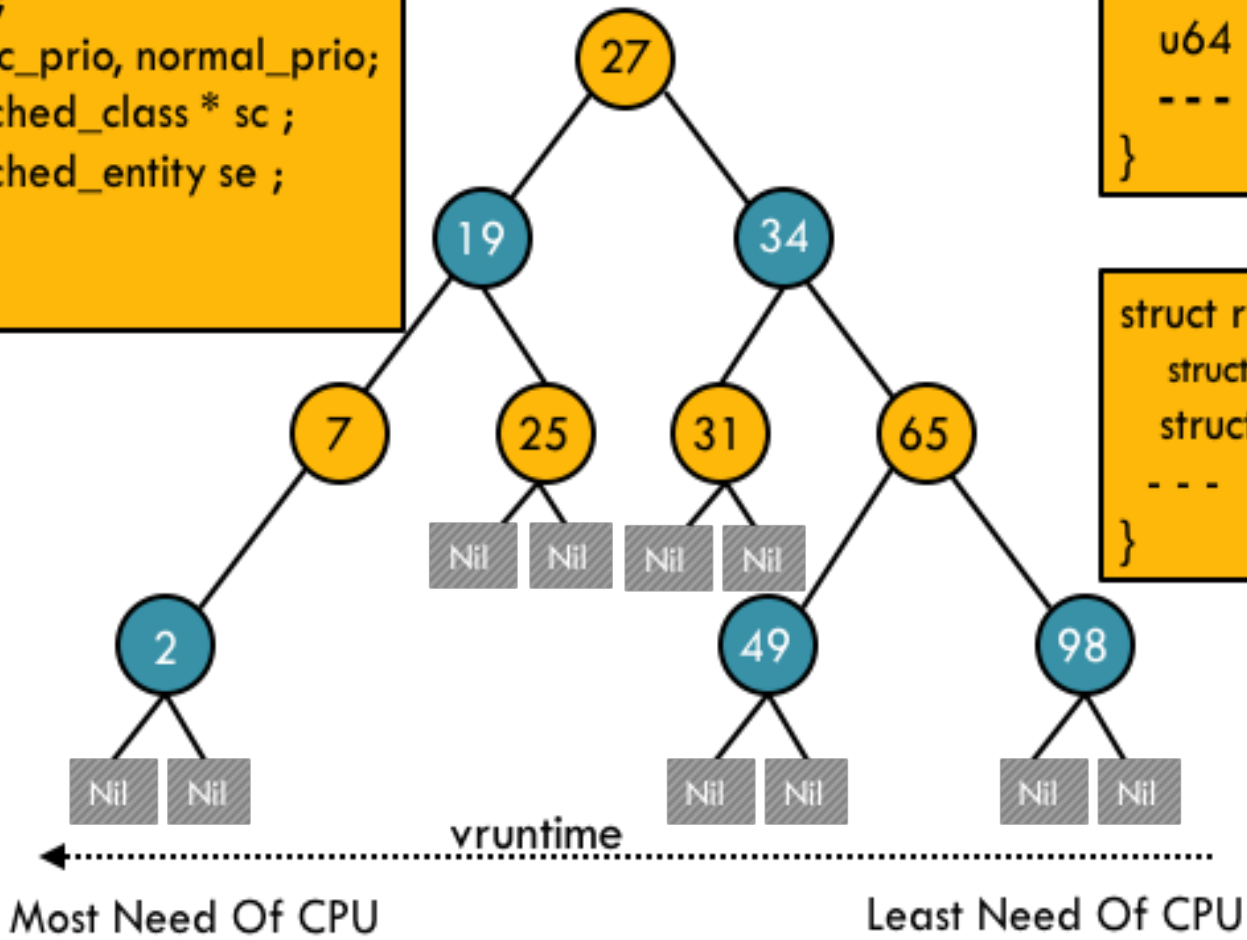
Linux CFS Scheduling Class

vruntime of a new process

```
struct task_struct{
   volatile long state;
   int prio;
   int static_prio, normal_prio;
   struct sched_class * sc ;
   struct sched_entity se ;
   - - -
}
```

```
struct sched_entity{
   struct load_wait  load ;
   struct rb_node    rn ;
   u64   vruntime;
   - - -
}
```

```
struct rb_node{
    struct rb_node * rb_left ;
   struct rb_node * rb_right;
  - - -
}
```



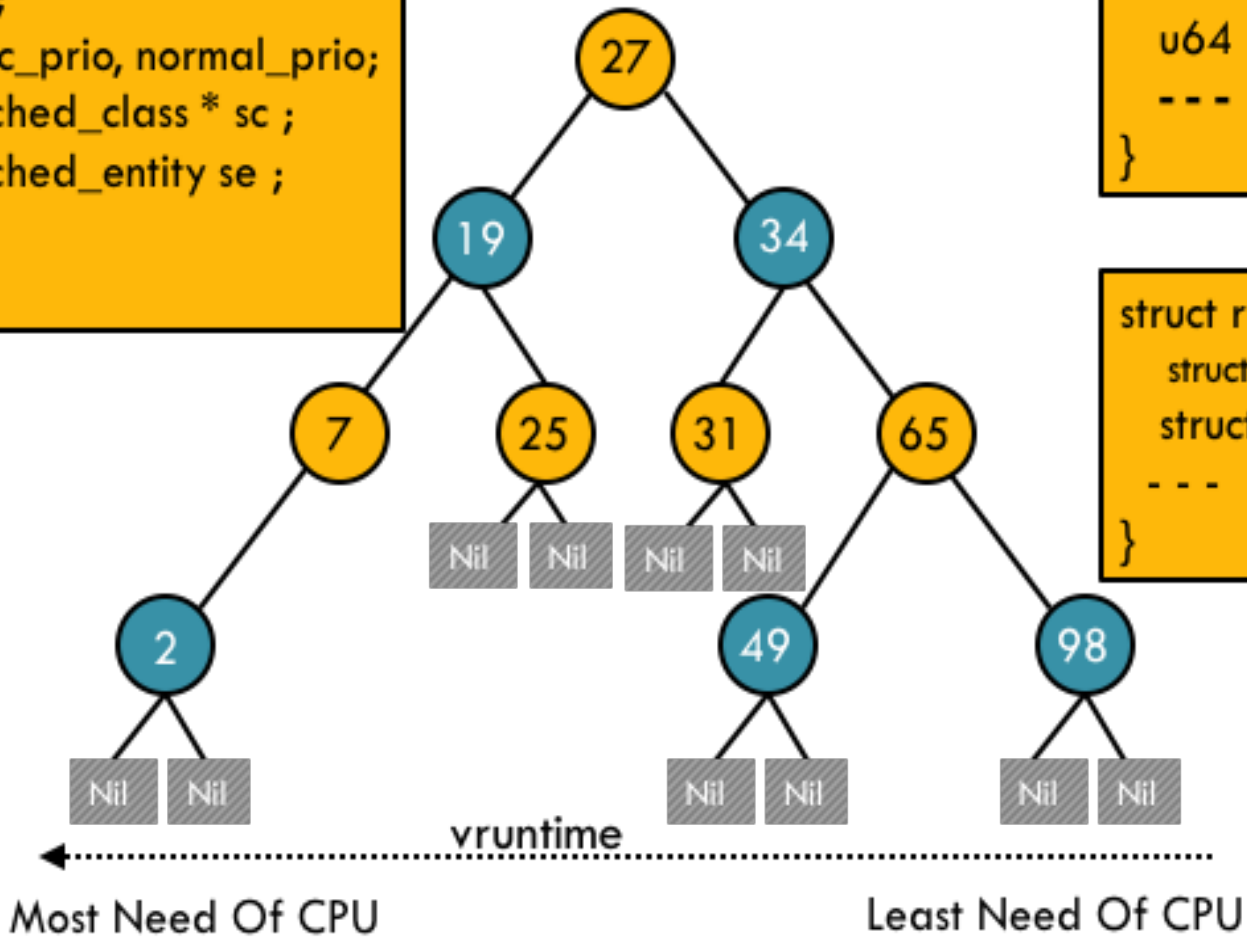
Linux CFS Scheduling Class

What about priorities within a class?

```
struct task_struct{
   volatile long state;
   int prio;
   int static_prio, normal_prio;
   struct sched_class * sc ;
   struct sched_entity se ;
   - - -
}
```

```
struct sched_entity{
   struct load_wait  load ;
   struct rb_node    rn ;
   u64   vruntime;
   - - -
}
```

```
struct rb_node{
    struct rb_node * rb_left ;
   struct rb_node * rb_right;
  - - -
}
```



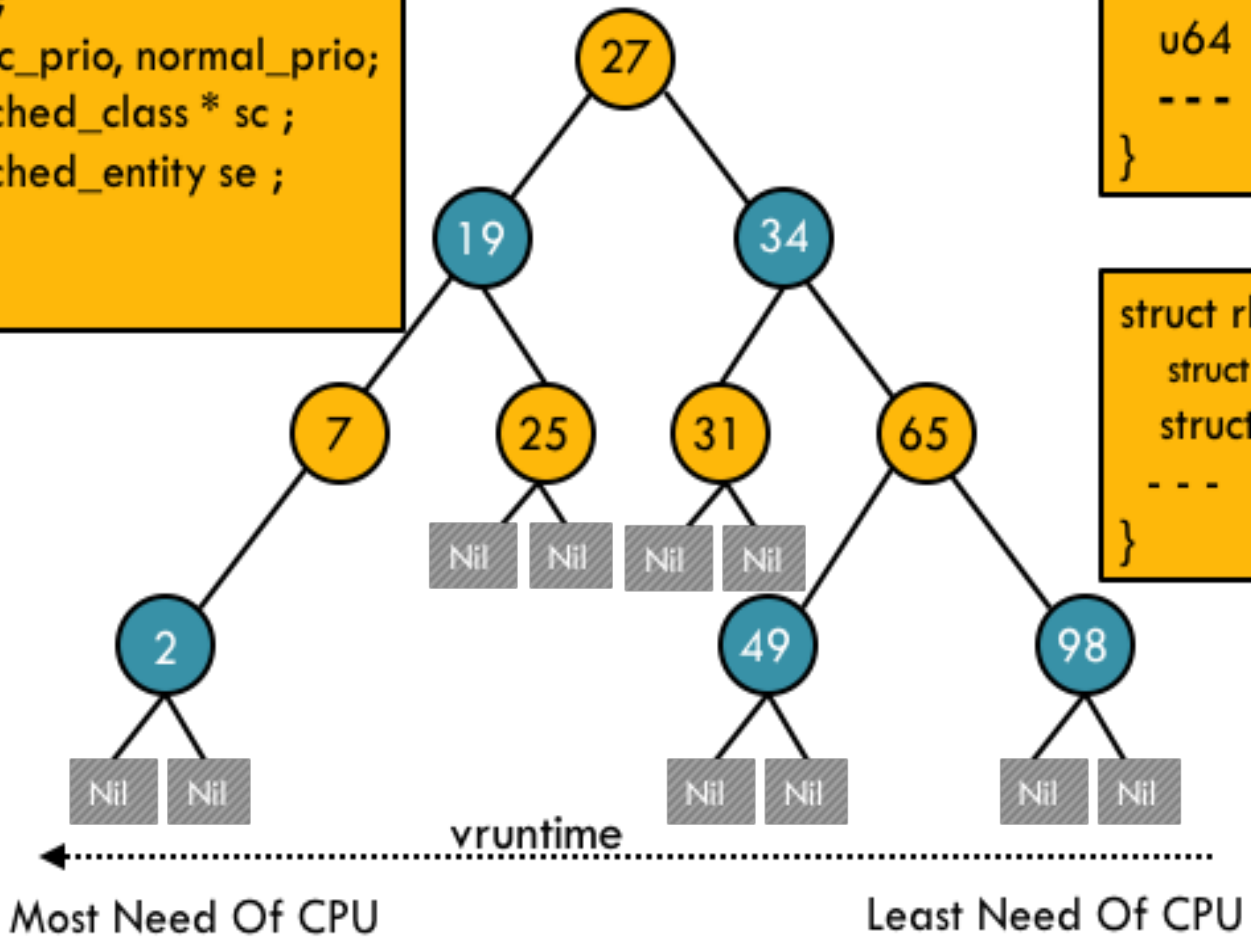
Linux CFS Scheduling Class

How CFS handles CPU bound and I/O bound processes?

```
struct task_struct{
   volatile long state;
   int prio;
   int static_prio, normal_prio;
   struct sched_class * sc ;
   struct sched_entity se ;
   - - -
}
```

```
struct sched_entity{
   struct load_wait  load ;
   struct rb_node    rn ;
   u64   vruntime;
   - - -
}
```

```
struct rb_node{
    struct rb_node * rb_left ;
   struct rb_node * rb_right;
  - - -
}
```



The Linux `schedtool` Utility

System calls related to Scheduling

System Calls related to Scheduling

```
int nice()
int getpriority()
int setpriority()
int sched_get_priority_min()
int sched_get_priority_max()
int sched_getscheduler()
int sched_setscheduler()
int sched_getparam()
int sched_setparam()
int sched_yield()
int sched_rr_get_interval()
int sched_getcpu()
int sched_getaffinity()
int sched_setaffinity()
```

Retrieving and Modifying nice Value

```
int nice(int inc);
```

- This call changes the base priority of the calling process by adding the *inc* to the nice value of the calling process. Only a superuser may specify a negative argument
- On success, the new nice value is returned and on error -1 is returned and *errno* is set appropriately
- Since `nice()` may legitimately return a value of -1 on successful call, we must test for error by setting *errno* to 0 prior to the call, and then checking for a -1 return status and a nonzero *errno* value after the call
- In case of a negative increment, the function invokes the `capable()` function to verify whether the process has a CAP_SYS_NICE capability
- The `nice()` system call affects only the process that invokes it. It is maintained for backward compatibility only; it has been replaced by the `setpriority()` system call

Retrieving and Modifying nice Value (cont...)

```
int getpriority(int which,int who);
int setpriority(int which,int who,int prio);
```

- The `getpriority()` and `setpriority()` system calls allow a process to get and set its own nice value or that of another process
- Both system calls take the argument *`which`* and *`who`*, identifying the process(es) whose priority is to be retrieved or modified. The *`which`* argument determines how *`who`* is interpreted. The *`which`* argument takes on of following values:
  - PIRO_PROCESS: Operates on the process whose PID equals who. If who is 0, use the caller's PID
  - PRIO_GRP: Operate on all of the members of the process group whose PGID equals who. If who is 0, use the caller's process group
  - PRIO_USER: Operate on all processes whose RUID equals who. If who is 0, use the caller's RUID

Getting Priority Ranges

```
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

- Above two calls return the maximum/minimum priority value that can be used with the scheduling algorithm identified by policy
- Processes with numerically higher priority values are scheduled before processes with numerically lower priority values
- Linux allows the static priority value range 1 to 99 for SCHED_FIFO and SCHED_RR and the priority 0 for SCHED_OTHER and SCHED_BATCH
- Scheduling priority ranges for the various policies are not alterable

Getting Scheduling Policy/Relinquishing CPU

```
int sched_getscheduler(pid_t pid);
```

- The `sched_getscheduler()` queries the scheduling policy currently applied to the process/thread identified by *pid*. If *pid* equals 0, the policy of the calling thread will be retrieved. On success, returns the policy number, 0 for SCHED_NORMAL, 1 for SCHED_FIFO and so on

```
int sched_yield();
```

- A process may voluntarily relinquish the CPU in two ways: by invoking a blocking system call or by calling `sched_yield()`
- If there are any other queued runnable processes at the same priority level, then the calling process is placed at the back of the queue, and the process at the head of the queue is scheduled
- If no other runnable processes are queued at this priority, then `sched_yield()` does nothing, the calling process simply continues using the CPU

Things To Do

If you have problems visit me in counseling hours. . . .