CPU Scheduler

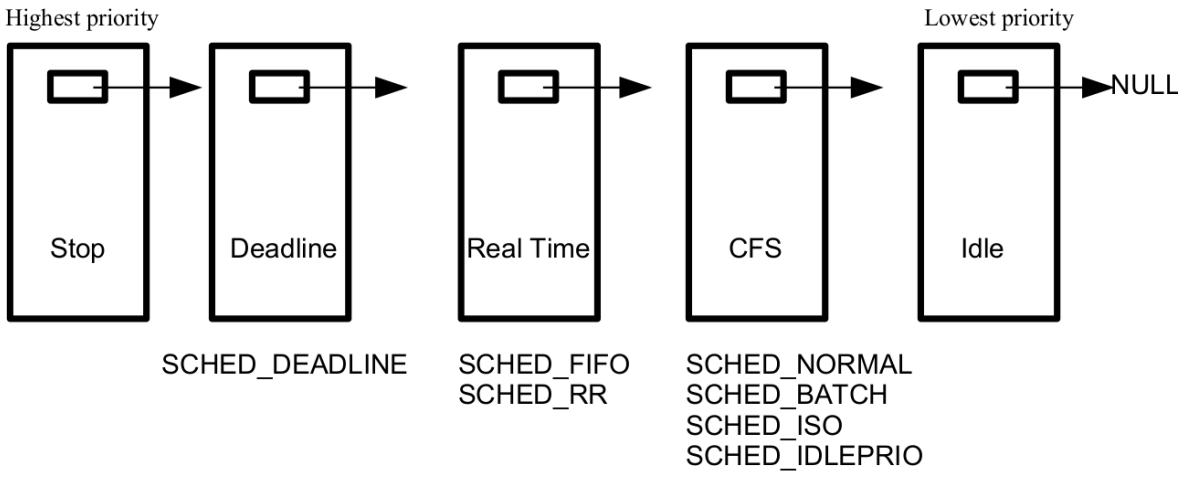
- Scheduling is a matter of managing queues to minimize queueing delay and to optimize performance in a queueing environment
- The process scheduler in a multitasking operating system is a kernel component that decides which process runs, when and for how long

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)

Linux CFS Scheduler

Scheduling Classes and Scheduling Policies



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 It is maintained for backward compatibility only; it has been replaced by the `setpriority()` system call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