Process Scheduling

A study of scheduling of processes in the Unix operating system

Pratap Vikram Singh

October 20th, 1997

Contents

1. The Scheduler

The scheduler is a component of the operating system that determines which process to run at any given time and how long to let it run. A time-quantum or time-slice is the brief period of time to which a process is allotted the CPU by the scheduler.

The scheduler must focus on two aspects:

Policy	The rules used to decide which process to run and when to switch to another.
Implementation	The data structures and algorithms used to carry out these policies.

The scheduling policy must try to meet the following objectives :
1. Fast response time for interactive applications.
2. High throughput for background jobs.
3. Avoidance of process starvation.

1.1. Scheduler Goals

The scheduler must ensure that the system delivers acceptable performance to each application as long as the total workload is in the expected range.

The scheduler has to deal with three types of processes:

Interactive	Processes requiring user input.
Batch	Processes like compilations and scientific computations.
Real-Time	Time critical processes like video player.

Apart from the above types, the scheduler has to ensure that kernel functions such as paging, interrupt handling and process management can execute promptly when needed.

2. Traditional Unix Scheduling

The traditional UNIX scheduling is priority based. Each process has a scheduling priority that changes with time. The scheduler always selects the highest priority runnable process. It uses preemptive time-slicing to schedule processes of equal priority and dynamically varies process priorities based on their CPU usage patterns. If a higher priority process becomes ready to run, the scheduler preempts the current process even if it has not completed its time slice or quantum.

However, the scheduler cannot preempt a process that is running in the kernel mode, e.g. while executing a system call. In this case, the running process might voluntarily relinquish the CPU when blocking on a resource or the scheduler can preempt the process when it returns to the user mode.

3. Process Priorities

The priority of a process can be between the integer values 0 and 127. Numerically lower values, denote higher priorities. Priorities between 0 and 49 are reserved for the kernel.

The `proc` structure contains the following fields that contain priority-related information:

`p_pri`	Current scheduling priority
`p_usrpri`	User mode priority
`p_cpu`	Measure of recent CPU usage
`p_nice`	User controlled nice factor

Table 1 Priority related fields of the proc structure

The scheduler uses the `p_pri` value to decide which process to schedule. When a process is in the user mode, its `p_pri` and `p_usrpri` are identical. When a process executes a system call it transitions to the kernel mode. If the process sleeps in this mode due to the unavailability of a resource, the CPU would be given to another process. When the previous process wakes up, its `p_pri` is temporarily boosted in order to give preference to kernel mode processing. At this time the value of the user mode `p_pri` is stored in the `p_usrpri` field. When the process returns to the user mode, its priority is restored to the original value by replacing the `p_pri` value with the `p_usrpri`.

The kernel associates a *sleep priority* with every event on which a process can block. This is a kernel constant and are between the values 0 and 49. E.g. Terminal input is 28 while disk I/O is 20. When a process sleeps on the particular event and later wakes up, the value of `p_pri` is set to that event's sleep priority. Because the value of the sleep priority is between 0 and 49, the process gains more priority than the other user processes and gets preference for scheduling. This allows system calls to execute promptly.

The user mode priority depends on two factors:
1. The nice value
2. Recent CPU usage.

The nice value is between 0 and 39 with the default set to 20. Increasing the value decreases the priority. Background processes are automatically given higher nice values. Only a super-user can decrease the nice value of a process.

The CPU usage, `p_cpu`, is the recent CPU usage of a process. It is initialized to 0 when a process is created. At every tick, the clock handler increments the `p_cpu` for the currently executing process upto a maximum of 127. Moreover, every second, the kernel invokes a routine called `schedcpu()` that reduces the `p_cpu` of every process by a decay factor. The decay factor is generally 1/2 or sometimes also dependent on the number of processes executing. E.g. BSD uses:

```
decay = (2*load_average)/(2*load_average+1)
```

Where `load_average` is the average number of runnable processes over the last second. The `schedcpu()` also recomputes the user priorities of all processes using the formula:

```
p_usrpri = PUSER+(p_cpu/4)+(2*p_nice)
```

Where `PUSER` is the baseline user priority of 50. Thus the more a process uses the CPU, its priority will go lower. The more a process waits for the CPU, the priority of the process will increase due to the decaying of the `p_cpu` value. This scheme avoids the starvation of low priority processes and also favors I/O bound processes(more wait time) to compute bound processes(more CPU usage).

4. The Traditional Scheduler Implementation

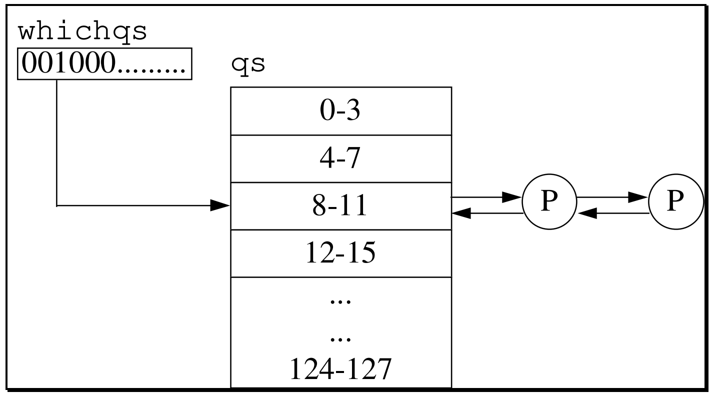

Figure 1 SVR3 Scheduler Queue

The scheduler maintains an array called `qs` of 32 run queues. Each queue corresponds to four adjacent priorities. Thus queue 0 is used for priorities 0-3, queue 1 is used for priorities 4-7 and so on. Each queue contains the head of a doubly linked list of `proc` structures. A global variable `whichqs` contains a bitmask with one bit for each queue. The bit for a queue is set if there is a process on that queue. Only runnable processes are kept in the scheduler queues. This simplifies the task of selecting a process to run.

The `swtch()` routine, which performs the context switch, examines the `whichqs` to find the index of the first set bit. This index identifies the scheduler queue containing the highest priority runnable process. `swtch()` removes a process from the head of the queue and switches context to it. When `swtch()` returns, the newly scheduled process resumes execution.

The highest priority process always runs, unless the current process is executing in the kernel mode. A process is assigned a fixed quantum, e.g. 100ms in BSD. This only affects the scheduling of the processes on the same queue. After every time quantum, the kernel invokes a routine called `roundrobin()` to schedule the next process from the same queue. If a higher priority process was runnable, it would be scheduled without waiting for `roundrobin()`. If all other runnable processes are on lower priority queues, the current process continues to run even though its quantum has expired.

There are three situations when a context switch occurs:

1. The current process blocks on a resource or exits. This is a voluntary context switch.
2. The priority recomputation procedure results in the priority of another process becoming greater than that of the current one.
3. The current process, or an interrupt handler, wakes up a higher priority process.

In case of a voluntary context switch, the kernel directly calls `swtch()` from the `sleep()` or `exit()` routines. An involuntary context switch occurs only when a process is running in the kernel mode. Thus the process does not get preempted immediately. Instead, the kernel sets a flag called `runrun` which indicates that a higher priority process is waiting to be scheduled. When the process is about to return to the user mode, the kernel checks the `runrun` flag and calls `swtch()` if it is set.

4.1. Drawbacks of the traditional method

1. If the number of processes is very large, it becomes very inefficient for the scheduler to recompute the priorities of each process every second.
2. There is no way to guarantee a portion of CPU resources to a specific process or a group of processes.
3. There are no considerations for processes with real-time characteristics.

4. Since the kernel is non-preemptive, higher priority processes may have to wait a significant amount of time even after being made runnable. This is called *priority inversion*.

5. The SVR4 Scheduler

The SVR4 scheduler supports a diverse range of applications including those requiring real-time response. There is a *scheduling class* which defines the scheduling policy for all processes that belong to it. The system may provide several scheduling classes.

By default, SVR4 defines two classes:

1. Time Sharing
2. Real Time

The scheduler provides a set of class independent routines that implement common services such as context switching, run-queue management and preemption. It also defines interfaces for class dependent functions such as priority computation. Each class implements these functions separately.

5.1. The class independent layer

Like before, the highest priority process always runs. The number of priorities has been increased to 160 and there is a separate dispatch queue for each priority. Numerically larger priorities correspond to higher priorities. The assignment and recomputation of process priorities are done by the class dependent layer.

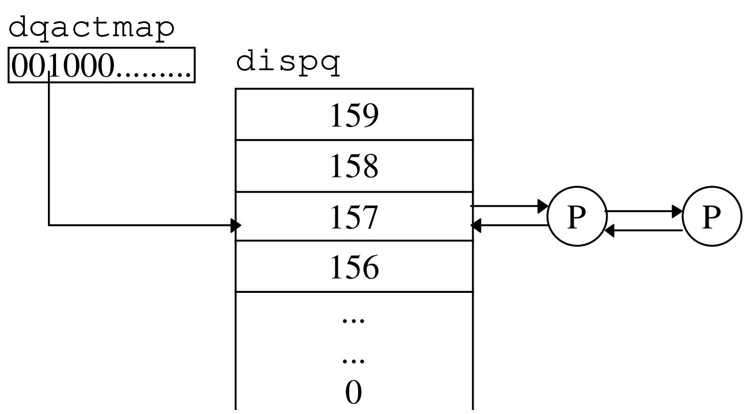

Figure 2 SVR4 Dispatch Queues

The `dqactmap` is a bitmap that shows which dispatch queues have at least one runnable process. Processes are placed on the queue by `setfrontdq()` and `setbackdq()`, and removed by `dispdeq()`. Typically, a new runnable process is placed at the back of a run queue, while a process that was preempted before its quantum expired is returned to the front of the queue.

5.2. Kernel Preemption

A major limitation of UNIX for use in real-time applications is the non-preemptive nature of the kernel. Real-time processes need to have a low *dispatch latency*, which is the delay between the time they become runnable and the time they actually begin running.

To address this problem, the SVR4 kernel defines several *preemption points*. These are places in the kernel code where all kernel data structures are in a stable state. When such a preemption point is reached, the kernel checks a flag called `kprunrun`. If this flag is set, it indicates that a real-time process is ready to run and the kernel preempts the current process.

The PREEMPT() macro checks kprunrun and calls the preempt() routine to actually preempt the process. The preempt() function invokes the CL_PREEMPT operation to perform class-dependent processing, and then calls swtch() to initiate the context switch.

swtch() calls pswtch() to perform the machine-independent part of the context switch, and then invokes lower-level assembly code to manipulate the register context, flush translation buffers, etc.

pswtch() clears the runrun and kprunrun flags, selects the highest-priority runnable process and removes it from the dispatch queue. It updates the dqactmap, and sets the state of the process to SONPROC (running on a processor). Finally, it updates the memory management registers to map the u-area and virtual address translation maps of the new process.

5.3. Interface to the Scheduling class

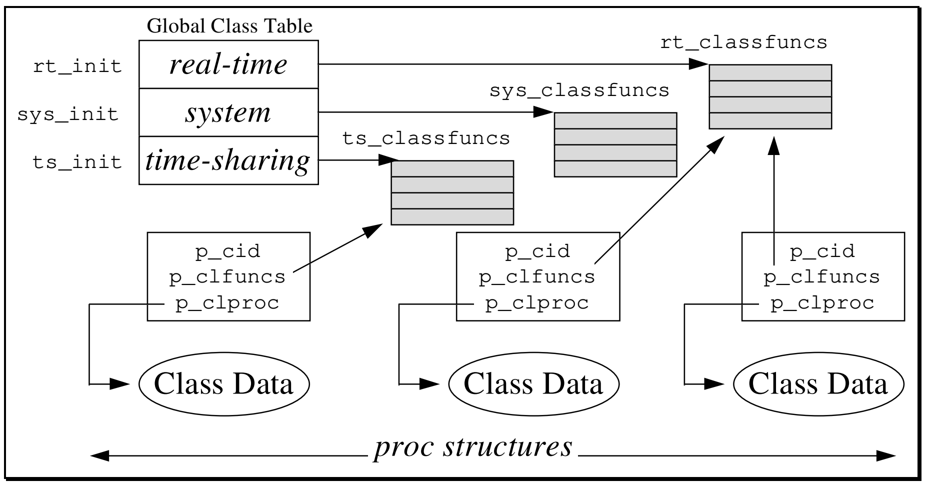

Figure 3 SVR4 Class dependent interface

All class dependent functionality is provided by a generic interface whose virtual functions are implemented differently by each scheduling class. Figure 3 shows how the class-dependent interface is implemented. The classfuncs structure is a vector of pointers to the functions that implement the class dependent interface for any class.

A *global class table* contains one entry for each class. This entry has:

1. Class name
2. Pointer to an initialization function
3. Pointer to the classfuncs vector for that class

When a process is first created, it inherits the priority class from its parent. It may be moved to a different class via the prionctl() system call. The proc of each process has three fields that are used by the scheduling class:

p_cid	This is the class ID of the process and simply indexes into the global class table.
p_clfuncs	Pointer to the classfuncs vector for that class to which the process belongs. It is copied from the global class table.
p_clproc	Pointer to a class-dependent private data structure.

Table 2 Scheduling related fields of the proc structure

A set of macros resolves calls to the generic interface functions and invokes the correct class-dependent functions.

```
#define CL_SLEEP(procp, clprocp, ...)\
(*(procp)->p_clfuncs->cl_sleep)(clprocp, ...)
```

The class-dependent functions can be accessed in this manner from the class-independent code. The scheduling class decides what actions each function will take. This allows for a very versatile approach to scheduling. By default, the 160 priorities are divided into the following three ranges:

0-59	Time-sharing class
60-99	System priorities
100-159	Real-time class

Table 3 Division of priorities in SVR4

6. Time-Sharing Class

The time-sharing class is the default class for a process. It changes the process priorities dynamically and uses *round-robin scheduling* for processes with the same priority. It uses a static *dispatcher parameter table* to control process priorities and time slices.

The time slice given to a process depends on its scheduling priority. The parameter table defines the time slice for each priority. By default, the lower the priority of a process, the larger the time slice. This is done because low priority processes are scheduled after a long time thus they should get more time to execute.

The time-sharing class uses event driven scheduling instead of recomputing the priorities of all the processes every second, e.g. The scheduler penalizes a process every time it exhausts its time slice by lowering its priority. On the other hand, it boosts up the priority of a process if it blocks on an event or resource or if it takes too much time to complete its time slice.

The time-sharing class uses a `struct tsproc` to store class dependent data. Its fields are:

`ts_timeleft`	Time remaining in quantum.
`ts_cpupri`	System part of the priority.
`ts_upri`	User part of the priority (nice value).
`ts_umdpri`	User mode priority (`ts_cpupri+ ts_upri`, but not more than 59).
`ts_dispwait`	Number of seconds of clock time since start of quantum.

Table 4 Fields of the `tsproc` structure

When a process resumes after sleeping in the kernel, its priority is a kernel priority and is determined by the sleep condition. When it later returns to the user mode, the priority is restored from `ts_umdpri`. The user priority is the sum of its `ts_upri` and `ts_cpupri`, but is restricted to a value between 0 and 59. `ts_upri` ranges from -20 to +19 with the default value of 0. This value can be changed by the super-user by using the `priocntl()` call. `ts_cpupri` is changed according to the dispatcher parameter table.

The parameter table contains one entry for each priority in the class. Each entry in the table contains the following fields:

`ts_globpri`	Global priority for the entry.
`ts_quantum`	Time quantum for this priority.
`ts_tqexp`	New `ts_cpupri` to set when quantum expires.
`ts_slpret`	New `ts_cpupri` to set when returning to user mode.
`ts_maxwait`	Number of seconds to wait for quantum expiry before using `ts_lwait`.
`ts_lwait`	Use instead of `ts_tqexp` if process took longer than `ts_maxwait` to use up its quantum.

Table 5 Entries of the parameter table

Globpri	Quantum	Tqexp	Slpret	Maxwait	Lwait
0	100	0	10	5	10
1	100	0	11	5	11
...	...	...	...	...	...
40	20	30	5	5	50
...	...	...	...	...	...
59	10	49	5	5	59

Table 6 Sample dispatch table

7. Real-Time Class

The real-time class uses priorities in the range 100-159. These priorities are higher than any time-sharing process, even those in the kernel mode. This means that a real-time process will be scheduled before any kernel process.

If a process is already executing in kernel mode when a real-time process becomes runnable, the real-time process must wait until the current process is about to return to the user mode or until it reaches a kernel preemption point.

Only superuser processes can enter the real-time class by calling `priocntl()`, specifying the priority and the time quantum.

Real-time processes are characterized by a fixed priority and time quantum. The only way to change these values is by using the `priocntl()` system call.

The real-time dispatcher parameter table is very simple. It contains the following two fields:

`rt_globpri`	Default global priority (100-159).
`rt_quantum`	Default time quantum for the priority.

Table 7 Real-time dispatch table entries

The default parameter table assigns larger time slices for lower priorities. The class dependent data of a real-time process is stored in a `struct rtproc`, which has the following fields:

`rt_pquantum`	Time quantum assigned to the process.
`rt_timeleft`	Time remaining in time quantum.
`rt_pri`	Priority level of the process.

Table 8 Fields of the `rtproc` structure

Real-time processes require bounded *dispatch latency* as well as bounded *response time*. The dispatch latency is the time between when the process becomes runnable and when it begins to run. The response time is the time between the occurrence of an event that requires the process to respond and the response itself.

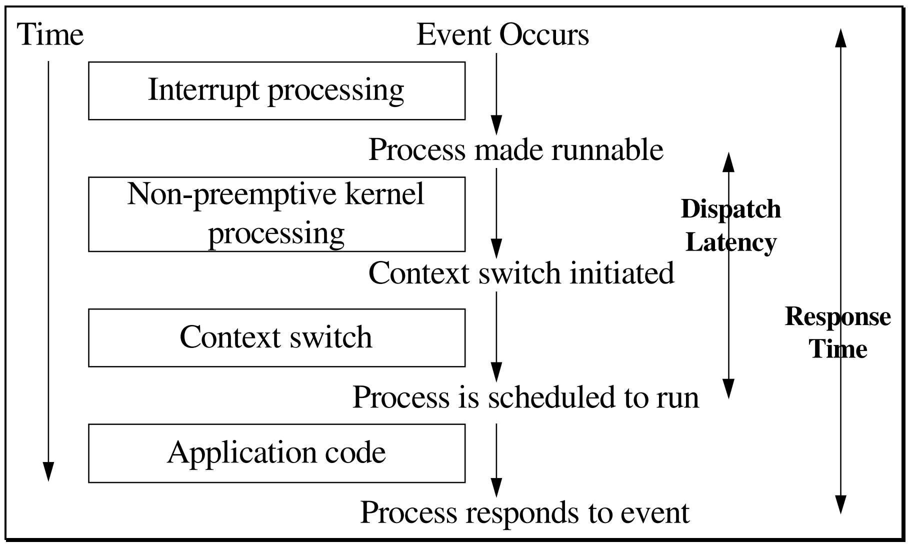

Figure 4 Response Time & Dispatch Latency

When a real-time process becomes runnable, the `rt_wakeup()` routine, that handles the class-dependent wakeup processing, sets the kernel flag `kprunrun`. When the current process reaches a preemption point, it checks this flag and initiates a context switch to the waiting real-time process.

8. References

1. Unix Internals - Uresh Vahalia.
2. 2. Magic Garden