UNIVERSITY OF WISCONSIN-MADISON
Computer Sciences Department

CS 537
Spring 2000

A. Arpaci-Dusseau

Multilevel Feedback Queue Schedulers

In this handout, we give a brief overview of the behavior of the Solaris 2.6 Time-Sharing (TS) scheduler, an example of a *Multilevel Feedback Queue* scheduler. The goal of a multilevel feedback queue scheduler is to fairly and efficiently schedule a mix of processes with a variety of execution characteristics. By controlling how a process moves between priority levels, processes with different characteristics can be scheduled as appropriate. Priority-based schedulers attempt to provide a compromise between several desirable metrics (e.g., response time for interactive jobs, throughput for compute-intensive jobs, and fair allocations for all jobs).

The queues in the system are ranked according to priority. Processes waiting in higher priority queues are always scheduled over those in lower priority queues. Processes at the same priority are usually scheduled in a round-robin fashion.

Such schedulers tend to be *preemptible* in order to support interactive processes. That is, a higher priority process is immediately scheduled if a lower priority process is running on the CPU.

Scheduling in Solaris

The Solaris operating system is based on Unix System V Release 4 (SVR4). Scheduling in Solaris, as in all SVR4-based schedulers, is performed at two levels: class-independent routines and class-dependent routines. Class-independent routines are those that are responsible for dispatching and preempting processes (the low-level mechanisms). Class-dependent routines are those that are responsible for setting the priority of each of its processes (the high-level policy).

By default, Solaris supports three scheduling classes: time-sharing (TS), real-time (RT), and system (SYS). Users with root privileges can easily implement and add new scheduling classes by adhering to a predefined interface. Each scheduling class gives each of its processes a priority, the range of which is shown below.

Scheduling Class	Priorities
Real-Time	100-159
System	60-99
Time-Sharing	0-59

Table 1: Range of Priorities for Scheduling Classes in Solaris

As long as a user has the correct privileges, he or she can submit jobs to any scheduling class. (See the man pages for **priocntl** on any machine running Solaris for information on how to submit jobs to different scheduling classes; however, since you probably don't have root privileges on your machine, you won't be able to do much.) By default, jobs are executed in the same scheduling class as the parent process that forked the job. Since your shell is running in the time-sharing class, all of your jobs run by default in the time-sharing class.

To see the scheduling class of each process in the system, run "**ps -edaflc**" (**-c** is the flag that shows the scheduling class). The fourth column shows the scheduling class of the running process. Most jobs will be running in the TS class, with a few (owned by root) running in the SYS class.

```
elaine1:~> /usr/bin/ps -edafc
     UID   PID  PPID  CLS PRI    STIME TTY    TIME CMD
    root     0     0  SYS  96   Aug 01 ?      0:00 sched
    root     1     0   TS  58   Aug 01 ?      1:06 /etc/init -
    root     2     0  SYS  98   Aug 01 ?      0:02 pageout
    root     3     0  SYS  60   Aug 01 ?     15:22 fsflush
    root   245   239   TS  59   Aug 01 ?      0:00 ttymon
    root   181     1   TS  48   Aug 01 ?      0:00 sendmail -q15m
    root   239     1   TS  59   Aug 01 ?      0:00 sac -t 300
    root    96     1   TS  58   Aug 01 ?      0:00 rpcbind
    root   125      1   TS  59   Aug 01 ?      0:32 syslogd
```

In this document, we only discuss the Solaris time-sharing (TS) class. Note the priorities of each of the processes, as listed in the fifth column.

Class Independent Functionality

The class independent routines arbitrate across the scheduling classes. This involves three basic responsibilities.

- The process with the highest priority must be dispatched, and the state of the preempted process saved.

- The class independent functions must notifying the class-dependent routines when the state of its processes changes (for example, at creation and termination, when a process changes from blocked to runnable, or runnable to blocked, and when a 10ms timer expires).

- Processes must be moved between priority queues in the class independent data structures, as directed by its scheduling class, and must be moved between blocked and ready queues.

Time-Sharing Scheduling Class

The Time-Sharing scheduler in Solaris is an example of a multi-level feedback queue scheduler. A job begins at priority 29. Compute-bound jobs then filter down to the lower priorities, where they are scheduled less frequently (but for longer time-slices) and interactive jobs propagate to the higher priorities (where they are scheduled whenever they have work to perform, on the assumption that they will soon relinquish the processor again). In the TS scheduler, the priority of a process is lowered after it consumes its allocated time-slice. Its priority is raised if it has not consumed its time-slice before a *starvation interval* expires.

Dispatch Table

The durations of the time-slices, the changes in priorities, and the starvation interval are specified in a user-tunable dispatch table. The system administrator (or anyone with root privileges again) can change the values in this table, thus configuring how the time-sharing scheduler manages its jobs. While this has the noble intention of allowing different systems to tune the scheduler to better handle their workloads, in reality no one really knows how to configure these tables well. Therefore, we will focus on the default dispatch table.

To see how this table is configured in your system, run

```
dispadmin -c TS -g
```

You should see something like that in Table 2. (Looking at the man pages on dispadmin and ts_dptbl may also be helpful). You will see one row for every priority in the scheduling class: from 0 to 59. For each priority, there are five columns:

- **ts_quantum:** Length of the time-slice (in the actual table, this value is specified in 10ms clock ticks; in the output from running dispadmin, the value is specified in units of 1ms).
- **ts_tqexp:** Priority level of the new queue on which to place a process if it exceeds its time quantum. Normally this field links to a lower priority time-sharing level.
- **ts_slpret:** the priority to change to (generally a higher priority) when the job returns from sleeping (i.e., from the blocked queue) if ts_dispwait exceeds ts_maxwait (see next entry).
- **ts_maxwait**: A per process counter, **ts_dispwait,** is initialized to zero each time a process is placed back on the dispatcher queue after its time quantum has expired or when it is awakened (ts_dispwait is not reset to zero when a process is preempted by a higher priority process). This counter is incremented once per second. If a process's ts_dispwait value exceeds the ts_maxwait value for its level, the process's priority is changed to that indicated by ts_lwait. The purpose of this field is to prevent starvation.
- **ts_lwait:** the priority to change to (generally a higher priority) if the starvation timer expires before the job consumes its time-slice (i.e., if ts_dispwait exceeds ts_maxwait).

In this table, the priority of jobs ranges from a high of 59 down to 0. Time-slices begin at 20ms at the highest priority and gradually increase in duration up to 200ms at the lowest priorities. Generally, the priority of a process decreases by 10 levels after it consumes its time-slice; the priority of

# ts_quantum	ts_tqexp	ts_slpret	ts_maxwait	ts_lwait	#PRIORITY LEVEL
200	0	50	0	50	# 0
200	0	50	0	50	# 1
200	0	50	0	50	# 2
200	0	50	0	50	# 3
200	0	50	0	50	# 4
200	0	50	0	50	# 5
200	0	50	0	50	# 6
200	0	50	0	50	# 7
200	0	50	0	50	# 8
200	0	50	0	50	# 9
160	0	51	0	51	# 10
160	1	51	0	51	# 11
160	2	51	0	51	# 12
160	3	51	0	51	# 13
160	4	51	0	51	# 14
160	5	51	0	51	# 15
160	6	51	0	51	# 16
160	7	51	0	51	# 17
160	8	51	0	51	# 18
160	9	51	0	51	# 19
120	10	52	0	52	# 20
120	11	52	0	52	# 21
120	12	52	0	52	# 22
120	13	52	0	52	# 23
120	14	52	0	52	# 24
120	15	52	0	52	# 25
120	16	52	0	52	# 26
120	17	52	0	52	# 27
120	18	52	0	52	# 28
120	19	52	0	52	# 29
80	20	53	0	53	# 30
80	21	53	0	53	# 31
80	22	53	0	53	# 32
80	23	53	0	53	# 33
80	24	53	0	53	# 34
80	25	54	0	54	# 35
80	26	54	0	54	# 36
80	27	54	0	54	# 37
80	28	54	0	54	# 38
80	29	54	0	54	# 39
40	30	55	0	55	# 40
40	31	55	0	55	# 41
40	32	55	0	55	# 42
40	33	55	0	55	# 43
40	34	55	0	55	# 44
40	35	56	0	56	# 45
40	36	57	0	57	# 46
40	37	58	0	58	# 47
40	38	58	0	58	# 48
40	39	58	0	59	# 49
40	40	58	0	59	# 50
40	41	58	0	59	# 51
40	42	58	0	59	# 52
40	43	58	0	59	# 53
40	44	58	0	59	# 54
40	45	58	0	59	# 55
40	46	58	0	59	# 56
40	47	58	0	59	# 57
40	48	58	0	59	# 58
20	49	59	32000	59	# 59

Table 2: Default Solaris Time-Sharing Dispatch Table

a process is increased to 50 or above when the starvation timer expires.

Implementation

For each job in the TS class, the following data structure is maintained (we've removed a few of the fields for simplicity):

```
/*
 * time-sharing class specific thread structure
 */
typedef struct tsproc {
     long           ts_timeleft;  /* time remaining in quantum */
     long           ts_dispwait;  /* number of seconds since */
                                  /* start of quantum (not reset */
                                  /* upon preemption) */
     pri_t          ts_pri;       /* priority (0-59) */
     kthread_t      *ts_tp;       /* pointer to thread */
     struct tsproc  *ts_next;     /* link to next tsproc on list */
     struct tsproc  *ts_prev;     /* link to previous tsproc */
} tsproc_t;
```

The kthread_t structure tracks the necessary information to context-switch to and from this process. This structure is kept separate from the time-sharing class in order to separate the mechanisms of the dispatcher from the policies of the scheduler.

There are seven interesting routines in the TS class:

- **ts_enterclass(thread *T):** called when a new thread is added to the TS class.
  It initializes a tsproc structure for this process and adds it to the list of processes

- **ts_exitclass(thread *T):** called when the thread terminates or exits the class.
  The tsproc structure is removed from the list of processes.

- **ts_tick(thread *T):** called once every 10ms with a pointer to the currently running thread.
  The ts_timeleft variable of the running thread is decremented by one. If ts_timeleft goes to zero, then its new priority is calculated according to the value of ts_tqexp in the table, its time-slice is reset, and ts_dispwait is cleared; the thread is then added to the back of the appropriate priority queue and a new job is scheduled.

- **ts_update():** called once a second to check the starvation qualities of each job.
  The routine increments the ts_dispwait counter of every process in the class (even those that are blocked) by one. If the job is on the ready queue (i.e., the job is neither running nor blocked), then if its ts_dispwait exceeds ts_maxwait, the priority and the ts_dispwait value of this process are reset (but not ts_timeleft). Note that this may involve rearranging the priority queues.

- **ts_sleep(thread *T):** called when the thread blocks (e.g., due to I/O or synchronization).
  The TS routine does not need to do anything in these circumstance. However, the dispatcher, or class-independent routines, must add the thread to the blocked queue and schedule a new thread.

- **ts_wakeup(thread *T):** called when the blocked thread becomes ready.

If the ts_dispwait value associated with this process is greater than the value of ts_maxwait in the dispatch table, then the priority of the process is set to that specified by ts_slpret, its time-slice (ts_timeleft) is reset, and ts_dispwait is cleared. If the priority of this job is less than that of the running job, then the newly awoken job is added to the back of its priority queue; otherwise, it preempts the currently running job.

- **ts_preempt(thread *T):** called when the thread is preempted by a higher priority thread. The preempted thread is added to the front of its priority queue.

Fairness

The Solaris time-sharing scheduler approximates fair allocations by decreasing the priority of a job the more that it is scheduled. Therefore, a job that is runnable relatively infrequently remains at a higher priority and is scheduled over lower priority jobs. However, due to the configuration of the default dispatch table (i.e., the starvation interval is set to zero), you will note that the priority of every process is raised once a second, regardless of whether or not it is actually starving. Thus, the allocation history of each process is erased every second and compute-bound processes tend to acquire more than their fair share of the resources.