NAME

curpriority_cmp, **maybe_resched**, **resetpriority**, **roundrobin**, **roundrobin_interval**, **sched_setup**, **schedclock**, **schedcpu**, **setrunnable**, **updatepri** - perform round-robin scheduling of runnable processes

SYNOPSIS

#include <sys/param.h>
#include <sys/proc.h>

int
curpriority_cmp(*struct proc *p*);

void
maybe_resched(*struct thread *td*);

void
propagate_priority(*struct proc *p*);

void
resetpriority(*struct ksegrp *kg*);

void
roundrobin(*void *arg*);

int
roundrobin_interval(*void*);

void
sched_setup(*void *dummy*);

void
schedclock(*struct thread *td*);

void
schedcpu(*void *arg*);

void
setrunnable(*struct thread *td*);

void
updatepri(*struct thread *td*);

DESCRIPTION

Each process has three different priorities stored in *struct proc*: *p_usrpri*, *p_nativepri*, and *p_priority*.

The *p_usrpri* member is the user priority of the process calculated from a process' estimated CPU time and nice level.

The *p_nativepri* member is the saved priority used by **propagate_priority**(). When a process obtains a mutex, its priority is saved in *p_nativepri*. While it holds the mutex, the process's priority may be bumped by another process that blocks on the mutex. When the process releases the mutex, then its priority is restored to the priority saved in *p_nativepri*.

The *p_priority* member is the actual priority of the process and is used to determine what runqueue(9) it runs on, for example.

The **curpriority_cmp**() function compares the cached priority of the currently running process with process *p*. If the currently running process has a higher priority, then it will return a value less than zero. If the current process has a lower priority, then it will return a value greater than zero. If the current process has the same priority as *p*, then **curpriority_cmp**() will return zero. The cached priority of the currently running process is updated when a process resumes from tsleep(9) or returns to userland in **userret**() and is stored in the private variable *curpriority*.

The **maybe_resched**() function compares the priorities of the current thread and *td*. If *td* has a higher priority than the current thread, then a context switch is needed, and KEF_NEEDRESCHED is set.

The **propagate_priority**() looks at the process that owns the mutex *p* is blocked on. That process's priority is bumped to the priority of *p* if needed. If the process is currently running, then the function returns. If the process is on a runqueue(9), then the process is moved to the appropriate runqueue(9) for its new priority. If the process is blocked on a mutex, its position in the list of processes blocked on the mutex in question is updated to reflect its new priority. Then, the function repeats the procedure using the process that owns the mutex just encountered. Note that a process's priorities are only bumped to the priority of the original process *p*, not to the priority of the previously encountered process.

The **resetpriority**() function recomputes the user priority of the ksegrp *kg* (stored in *kg_user_pri*) and calls **maybe_resched**() to force a reschedule of each thread in the group if needed.

The **roundrobin**() function is used as a timeout(9) function to force a reschedule every *sched_quantum* ticks.

The **roundrobin_interval**() function simply returns the number of clock ticks in between reschedules triggered by **roundrobin**(). Thus, all it does is return the current value of *sched_quantum*.

The **sched_setup**() function is a SYSINIT(9) that is called to start the callout driven scheduler functions. It just calls the **roundrobin**() and **schedcpu**() functions for the first time. After the initial call, the two functions will propagate themselves by registering their callout event again at the completion of the respective function.

The **schedclock**() function is called by **statclock**() to adjust the priority of the currently running thread's ksegrp. It updates the group's estimated CPU time and then adjusts the priority via **resetpriority**().

The **schedcpu**() function updates all process priorities. First, it updates statistics that track how long processes have been in various process states. Secondly, it updates the estimated CPU time for the current process such that about 90% of the CPU usage is forgotten in 5 * load average seconds. For example, if the load average is 2.00, then at least 90% of the estimated CPU time for the process should be based on the amount of CPU time the process has had in the last 10 seconds. It then recomputes the priority of the process and moves it to the appropriate runqueue(9) if necessary. Thirdly, it updates the %CPU estimate used by utilities such as ps(1) and top(1) so that 95% of the CPU usage is forgotten in 60 seconds. Once all process priorities have been updated, **schedcpu**() calls **vmmeter**() to update various other statistics including the load average. Finally, it schedules itself to run again in *hz* clock ticks.

The **setrunnable**() function is used to change a process's state to be runnable. The process is placed on a runqueue(9) if needed, and the swapper process is woken up and told to swap the process in if the process is swapped out. If the process has been asleep for at least one run of **schedcpu**(), then **updatepri**() is used to adjust the priority of the process.

The **updatepri**() function is used to adjust the priority of a process that has been asleep. It retroactively decays the estimated CPU time of the process for each **schedcpu**() event that the process was asleep. Finally, it calls **resetpriority**() to adjust the priority of the process.

SEE ALSO

mi_switch(9), runqueue(9), sleepqueue(9), tsleep(9)

BUGS

The *curpriority* variable really should be per-CPU. In addition, **maybe_resched**() should compare the priority of *chk* with that of each CPU, and then send an IPI to the processor with the lowest priority to trigger a reschedule if needed.

Priority propagation is broken and is thus disabled by default. The *p_nativepri* variable is only updated if a process does not obtain a sleep mutex on the first try. Also, if a process obtains more than one sleep mutex in this manner, and had its priority bumped in between, then *p_nativepri* will be clobbered.