Project 2 Design Document

Lottery Scheduler

Desmond Vehar *dvehar*
Dave Lazell *dlazell*
Neil Killeen *nkilleen*
Melanie Dickinson *mldickin*

May 2, 2014

1 Design

We present a modification to the default Minix user space process scheduler. Like the original, ours manipulates CPU time afforded to processes via priority queues and "niceness" values. Both are repurposed for an implementation of lottery scheduling, where niceness of a process is reinterpreted as probabilistic execution time allocated to it relative to others, i.e., weighted random chance of selection for execution (or its number of lottery tickets).

Each time a process needs to be scheduled, a lottery is run, randomly selecting one from all tickets held by runnable processes in user space. The lottery method is lightweight, prevents starvation, and allows the scheduler to relatively easily discriminate among types of processes based on their behaviors.

Its implementation is divided into two primary tasks: re-prioritization of processes based on random selection of winning tickets, and dynamic adjustment of these tickets to favor certain processes for higher responsiveness.

1.1 Lottery

The kernel maintains 15 priority queues, and schedules (in kernel space) with simple round robin protocol, always picking the head of the highest ordinal queue. Any user space scheduler (including ours and the default) interfaces with this behavior via system calls, `sys_schedctl, sys_schedule`, to edit the queue number of a particular active/runnable process.

We use queue 15 as the holding queue, in which all user space processes picked up by the user scheduler are kept, and from which a winner is chosen for each lottery. Queue 14 (a higher priority) is where the winner is placed, ensuring it begins execution before any in the holding queue. Note that this doesn't guarantee it finishes before any in the lower priority queue—see section

1.1.2. Once the winner runs out of quantum, it is replaced in the holding queue and a new lottery is held.

An alternative design would add new queues, 16 and up. But adapting existing queues is more consistent with the goal of creating a user space scheduler, one of whose virtues lies in not having to modify the kernel.

1.1.1 Special case: One process

When our scheduler has responsibility of only one process, it doesn't need to move it back and forth between the holding and winning queue, or hold a lottery at all.

1.1.2 Special case: Blocking winner

If the winning process blocks, the kernel will select the next task from the next highest priority, the holding queue. Thus, losing tasks may complete their quanta while a winner was scheduled. This case is detected by `SCHED` reception of an out of quantum (OOQ) message from a process currently in the holding queue, while there is another process in the winning queue. If this happens regularly or for a prolonged period, the user scheduler will degenerate into kernel round robin scheduling of processes in queue 15.

We solve this by replacing the blocking winner in the holding queue, but with additional tickets, for greater probability of being selected again. This was thought to be more succinct than managing multiple winners, but still prevent the winner from blocking indefinitely and corrupting the scheduler. A more sophisticated implementation (that would not solve the problem entirely, but contribute) would incorporate ticket transfer from blocking processes to their dependencies.

1.2 Dynamic Ticket Adjustment

To reap the benefits of the lottery as a cheap but informed decision-maker, our scheduler needs the ability to adapt the distribution of tickets to the runtime behaviors of processes being scheduled.

When processes run out of quantum, the kernel sends a message to the scheduler on the process' behalf, containing accounting information, like number of IPC calls performed during the last run. If there are many such calls consistently through multiple runs of the same process, it is probably I/O-bound, and should be prioritized with more tickets, relative to those with less IPC calls during their quanta.

2 Implementation

The implementation is divided among six files.

- `schedule.c` defines the lottery scheduler

- `schedproc.h` defines (structure and fields of) the user space PTEs
- `proc.c`, `proc.h` declaration and implementation of the process table
- `utility.c`'s representation of niceness is changed to the less constrained ticket value abstraction, so some error-checking is removed
- `main.c`'s `announce()` message is modified to publicize the modified OS

2.1 `schedule.c`

This file contains the implementation of the user mode scheduler, and therefore the bulk of the lottery functionality. Constants for lottery-specific priorities (`HOLDING_Q, WINNING_Q`) are defined in terms of queue constants from `config.h`, which remains unchanged.

Control messages from the Process Manager, including notifications that a new process needs to be scheduled, requests for process termination, and scheduler initialization are handled similarly. The main modification here is in populating or addressing the added fields of the `schedproc` struct.

New functions `change_tickets()` and `do_lottery()` are defined. The former takes as arguments a pointer to the relevant process entry (a `schedproc` struct), and the quantity to increase that process' ticket collection by. This is used for dynamic ticket adjustment, described in section 1.2.

2.1.1 `do_lottery()`

This function counts tickets of "winnable" processes, uses a cheap Time Stamp Counter register access to generate a pseudo-random ticket number, finds the process with that ticket number, and re-prioritizes that winner to `WINNING_Q`. Winnable processes are those that have a valid `schedproc` (PT) entry, that are being scheduled by our lottery scheduler, and that are currently in the holding queue. There can be at most 256 user processes in the PT at a time (a user-settable parameter in `config.h`), and is usually much less, so counting winnable process tickets anew for each lottery is relatively cheap. Might change to global variable.

2.1.2 `do_noquantum()`

Each time a process runs out of quantum, the kernel will call `do_noquantum()` on its behalf. We check here for the identity and behavior of the returned process, granting additional tickets if it was a system process or an I/O-bound winner (see 1.1.2), and less if . Non-system processes will be returned to the holding queue with refreshed quantum (via `schedule_process()`), and another lottery held. Incomplete function? Check kernel feedback?.

2.2 `schedproc.h`

The file declaring the process table values for the user space scheduler is mostly unchanged. Two extra fields are added; for the tickets value and the flag indicating this process is being scheduled by our lottery scheduler. Tickets are adjusted in `do_nice()`, and used in `do_lottery()`. The `USER_PROCESS` flag is checked before making any decisions about a process we may schedule, to ensure we have that responsibility. Also blocking, for #times another process executed while this was the winner?

3 Testing

X files were constructed for testing.

- `cpu` defines a CPU-bound process (calculate pi)
- `io` defines an I/O-bound process
- `test1` — two CPU-bound processes, equal ticket numbers
- `test2` — two CPU-bound processes, ticket ratio 1:2
- `test3` — three CPU-bound processes, ticket ratio 1:2:4
- `test4` — two CPU-bound processes, ticket ratio 1:100
- `test5` — none
- `test6` — one CPU and one I/O-bound process, with equal amounts of tickets, first run sequentially, then concurrently

Should elaborate on rationale for specific test methods and results of testing here.