XV6 Modification report

> Pramod Rao B
>
> 2020111012

Specification 1: `strace` command

Implemented a system call `strace`,

`strace mask command [args]`

Features:

- Runs the specified `<command>` with the specified arguments `[args]`
- Traces the `i`th system call if and only if the bit `1 << i` (or, equivalently `2^i`) is set in `mask`. That is, if `mask & (1 << i) == 1`

Steps taken to achieve this:

1. Added a new variable `traceMask` in the process structure `struct proc` defined in the file `proc.h` which is initialized to 0 so that no bit is set
2. Added a function `sys_trace` in `sysproc.c` which acts as a wrapper for the actual system call. It calls the function `trace` with the first argument representing the mask.
3. Added a function `trace` in `proc.c` which sets the `traceMask` variable of the current mask to the argument supplied.
4. Modified the `fork` function in `proc.c` to copy the parents `traceMask` onto the child
5. Added a declaration for `trace` in `defs.h` and a declaration for `sys_trace` in `syscall.c`
6. In order to know the number of arguments for each system call, added a table entry for each system call which stores its name and the required number of arguments
7. Added new entries for the `trace` system call in `syscall.h`
8. Modified the `syscall` function in `syscall.c` to print the traced information if the bit set condition is met. Additionally make sure to save the value stored in the first register `a0` before executing the system call since the return value if overwritten in `a0`
9. Created a new user program `strace.c` in the user directory which calls the `trace` function with the mask argument and then proceeds to call `exec` to execute the given `<command>`
10. Added an entry `$U/_strace` in the Makefile to enable compilation of the user program and a declaration for `trace` in `user.h`
11. Added a stub entry for `trace` in `user.pl` to call the appropriate syscall

---

Specification 2: Scheduling

The default scheduler of xv6 is round-robin-based. The modified xv6 contains 3 other scheduling policies such that the kernel shall only use one scheduling policy which will be declared at compile time using the `SCHEDULER` macro

Use `SCHEDULER=X` while compilation to use one of the following scheduling policies

- FCFS: First Come First Serve
- PBS: Priority Based Scheduling
- MLFQ: Multilevel Feedback Queue

First come first serve (FCFS)

Features:

A non-preemptive scheduling algorithm which picks the process which arrived the earliest among all runnable processes

Steps taken to achieve this:

1. Added an entry in the Makefile for defining the macro `FCFS`
2. Added a new integer variable `creationTime` in the process structure `struct proc` defined in `proc.h` which is set to the current `tick` value when created (in `allocproc()` function)
3. For the scheduling, iterated through the array of processes checking for `RUNNABLE` processes. When found, if the process at hand has a lower `creationTime` than the current minima `creationTime`, then release the old lock while holding onto the current process lock. We then context switch to the held process if any.
4. In order to make the scheduling non-preemptive, disabled the timed interrupt (`which_dev == 2`) in `usertrap()` and `kerneltrap()` functions in `trap.c`

Priority based schedling (PBS)

Features:

A non-preemptive scheduling algorithm which picks the process with the lowest dynamic priority. To break ties, we pick the process which has been scheduled lesser number of times. To further break ties, we pick the process with the earlier creation time.

The dynamic priority of a process is an integer in the range [0, 100] and is calculated as,

`DP = max(0, min(SP - niceness + 5, 100))`

where,

- Static pririty (SP) is an integer in the range [0, 100] and is set to 60 by default during creation of a process
- `niceness` is an integer in the range [0, 10] is calculated as,

  `niceness = (ticks spent in sleeping state) * 10 /(ticks spent in (running + sleeping) state)`

  Note: The ticks mentioned above are measured since the last time the process was scheduled

Steps taken to achieve this:

1. Added an entry in the Makefile for defining the macro `PBS`
2. Added a new integer variables `totalRTime, numSched, prevRTime, prevSleepTime` in the process structure `struct proc` defined in `proc.h` which is set to the current `tick` value when created (in `allocproc()` function). They measure the total runtime of a process since creation, the number of times it was scheduled till now, total number of ticks spent `RUNNING` since the last time it was scheduled and `SLEEPING` respectively.
3. For the scheduling, iterated through the array of processes checking for `RUNNABLE` processes. When found, if the process at hand "is better" (that is, has higher priority including all the tie breaks as mentioned above) than the current best process, then, we release the old lock while holding on to the new lock. We then context switch into the held process if any.

4. Set the `prevRTime` and `prevSleepTime` variables to 0 everytime the process is scheduled and similarly increment `numSched`
5. To update the time fields, we call the function `update_runtime` defined in `proc.c` which iterates over the process table and increments the `totalRTime, prevRTime, prevSleepTime` variables as needed.
6. We use the `get_DP` function (defind in `proc.c`) to calculate the dynamic priority of the given process. We initalize `prevRTime` to -1 indicating that the process has not been run a single time (or that the `set_priority` system call has been called as we shall see later) and if so, considers the `niceness` to be 5 (the default value). If not, it uses the aforementioned formula
7. In order to make the scheduling non-preemptive, disabled the timed interrupt (`which_dev == 2`) in `usertrap()` and `kerneltrap()` functions in `trap.c`

setpriority new_priority pid

Features:

Added a new system call `setpriority` which takes in two arguments, the new priority and the pid of the process to be affected and sets the static priority to the given value if the process has been found.

Steps taken to achieve this:

1. Added the declarations (for `set_priority(), sys_set_priority()`) and user program (`setpriority()`) by following the steps followed in specification 1 for the `strace` system call
2. Iterate through the process table searching for a process with the given pid. If found, set its static priority to the supplied value.
3. Additionally, we set the `prevRTime` variable to -1. This way, the `get_DP` functions calculates the `niceness` of the process the next time as 5, which is the default expected behaviour of `setpriority`

Multi-Level Feedback Queue (MLFQ)

Features:

A preemptive scheduling algorithm which uses queues of varying priority to keep track of processes. Lower priority processes get more time slices but can also be preempted when a process with higher priority enters the queue.

The MLFQ scheduling policy has 5 (variable NMLFQ) queues of levels 0 through 4 representing the highest to lowest priority queues. When a new process enters the system it gets added to the highest priority queue. It is then assigned a time quanta specific to its queue level (1 << i ticks for level i). After the time quanta, if the process happens to exhaust it, it implies that the process is CPU bound and hence, is moved to the queue which is one lower in priority. If not, then the process stays in the same queue.

Steps taken to achieve this:

1. Created [NMLFQ = 5] number of queues and added basic queue functionality
2. Added variables `qLevel, timeSlice, qEnter, timeSpent[NMLFQ]` in process structure in `proc.h` to store the priority level, time slice for that level, the time at which the process enters the queue. `qLevel` is initialized by 0, `timeSlice` to 1, `qEnter` to creation time, and `timeSpent` to 0.

3. For the scheduling, iterate through the process table looking for runnable processes. If found, insert it into the appropriate queue according to the `qLevel` variable.
4. Going from level 0 to NMLFQ, get the first process that needs to be run.
5. We update the `timeSpent[i]` variable at every clock interrupt in the `update_runtime` function for the current level of the process.
6. We set the variable `qEnter` to the tick value whenever the process is scheduled
7. We set the variable `timeSlice` to 1 << qLevel whenever the process is scheduled
8. In `trap.c` in `kerneltrap` and `usertrap`, after every timer interrupt check if the process exhausted its time slice. If so, then increment its level (thereby decreasing priority) by 1 and `yield()` to reschedule. If not, then the process may stay in the same queue for now. Additionally, check if any queues before the current level has any processes in it. If so, then those processes need to be executed before the current process. So `yield()` to let the rescheduling take over.
9. By way of implementation, processes in a certain level are done via round-robin basis since we iterate through

---

In MLFQ scheduling, a process that voluntarily relinquishes control of the CPU leaves the queuing network and when the process becomes ready again, is inserted at the tail of the same queue from which it was relinquished earlier. **This feature could be exploited by a process**

A process might voluntarily relinquish control to the CPU just at the end of its time slice and immediately go into the ready state again. This helps the process masquerade as an I/O bound process thereby making the CPU let it stay in its high priority queue without being moved to a lower priority queue.

Specification 3: Procdump

`procdump` is a function (present in `kernel/proc.c`) that prints a list of processes to the console when a user types `ctrl-p` on the console. The functionality of `procdump` has been extended to print additional information for other scheduling policies.

Prints the following information about the processes

- **PBS:**
  - PID of the process
  - Priority: Dynamic priority of the process obtained using the `get_DP` function which calculates as mentioned in the PBS section
  - State of the process ("sleep", "runble", "run" or "zombie")
  - Total runtime of the process which is stored in the variable `totalRTime`
  - Total wait time of the process which is calculated as `total - totalRTime` (where, `total = ticks - creationTime` which represents the total ticks elapsed since the process was created)
  - Number of times the process was scheduled (nrun) which is stored in the variable `numSched`
- **MLFQ:**
  - PID of the process
  - Priority: Level of the queue of the process (-1 if the process is not in any queue) which is obtained by checking the variables `inQueue` and if it is set to 1, then reading from `qLevel`
  - State of the process ("sleep", "runble", "run" or "zombie")

- Total runtime of the process which is stored in the variable `totalRTime`
- Total wait time of the process which is calculated as `total - totalRTime` (where, `total = ticks - creationTime` which represents the total ticks elapsed since the process was created)
- Number of times the process was scheduled (nrun) which is stored in the variable `numSched`
- q_i (5 entries for 5 levels of the queue) representing the total ticks the process has spent in the `i`th queue which is stored in the variable `timeSpent[i]`
  Note: We observe the priority being -1 for multiple processes. **This is the expected behaviour**. This is because, when a process is runnable, it might mean that the process is in the proc table but has not been picked into any of the queues. If a process is sleeping, then by definition it has been popped from the queue and is sleeping so has not been added to the queue yet.

Benchmarking

The `schedulertest` user program was run (three times and then averaged the results) with the `CPU` variable set to 1 to obtain the following running and waiting times.

Scheduling policy	Average run time	Average wait time
Default Round-Robin (RR)	21	186
First Come First Serve (FCFS)	43	196
Policy Based Scheduling (PBS)	21	153
Multi-Level Feedback Queue (MLFQ)	22	177

We see that the running time for FCFS is higher than the other policies by a significant amount. This is justified because in FCFS, we do not introduce I/O bound processes in schedulertest and so, since it has more number of CPU bound processes it increases the average run time.

We also observe that PBS gives the lowest wait time. We would expect MLFQ to have lower wait times because it takes into account more factors than PBS. However, using the array-based queue implementation and iterating through the process table and the queue to find the process to run causes a lot more overhead. If implemented using a different data-structure perhaps the MLFQ would perform better. But as of the current implementation, PBS clearly performs better with a much lower average waiting time.
