ELL783 Assignment 1

Ishvik Kumar Singh

2018EE10616

---

1. Introduction to xv6 operating system

1.1 Installing and Testing xv6

The following commands were used to install QEMU and xv6 from `Assignment_1` folder:

```
sudo apt-get install qemu
sudo apt-get install libc6-dev:i386
sudo apt-get install qemu-system-x86
tar xzvf xv6-rev11.tar.gz
cd xv6-public
make
make qemu
```

1.2 System Calls

System call numbers are defined in `syscall.h`. System call routines are declared in `syscall.c`. `syscalls[]` in `syscall.c` is the array of function pointers which point to respective system call routines. The system call routines are called using the corresponding function pointer in the function `syscall()`. System call routines are defined in `sysproc.c` and `sysfile.c`.

The wrapper functions for the system calls are declared in `user.h`. When wrapper functions are called in user programs, control passes to the macro in `usys.S`, which assigns the system call number to `%eax` register and generates and interrupt with the code `T_syscall`. Interrupts are handled in the files `trapasm.S` and `trap.c`.

Thus, to implement new system calls changes have to be made in the following files:

- `user.h`
- `usys.S`
- `syscall.h`
- `syscall.c`
- `sysproc.c` or `sysfile.c`

System call routines defined in `sysproc.c` or `sysfile.c` may call functions implemented in other kernel files.

1.2.1 Listing the running processes

The system call routine, defined in `sysproc.c`, is `sys_ps()`, and the wrapper function is `ps()`.

`ptable` in `proc.c` contains an array of all the processes `proc[]`

A process in xv6 can have one of the following states:

- `UNUSED`: unallocated
- `EMBRYO`: when a process is allocated, the state is converted from `UNUSED` to `EMBRYO`
- `SLEEPING`: sleeping process
- `RUNNABLE`: ready process
- `RUNNING`: currently scheduled process
- `ZOMBIE`: when a process calls `exit()`

To list of current running processes, the `ps()` function in `proc.c` iterates over `ptable.proc[]`, and prints the details of processes with `RUNNING`, `RUNNABLE`, or `SLEEPING` state.

User program `process_list.` has been added to test the system call.

```
$ process_list
pid:1  name:init
pid:2  name:sh
pid:5  name:process_list
```

1.2.2 Printing the available memory

The system call routine, defined in `sysproc.c` is `sys_memtop()`, and the wrapper function is `memtop()`.

`sys_memtop()` calls `kmemtop()` defined in `kalloc.c`. `kalloc.c` handles physical memory allocation, and contains a linked list of free frames `freelist` (of type `struct run`, `run` is used to address a free page) in the structure `kmem`. The available physical memory in the system is `number of free frames * page size (in bytes)`. To determine the number of free frames, the length of `freelist` is determined.

User program `mtop` has been added to test the system call.

```
$ mtop
available memory: 232607744
```

1.2.3 Context Switching

The system call routine, defined in `sysproc.c`, is `sys_csinfo()`, and the wrapper function is `csinfo()`.

To count the number of context switches, a counter `switchnum` has been added to the process control block `proc`, defined in `proc.h`. The counter is initialised to zero when a process is allocated in `allocproc()` in `proc.c`. Every process has a `context` structure on its kernel stack, and the job of switching between two contexts is performed by `swtch()` function. `swtch()` is called in `sched()` in `proc.c` where context switch occurs between the kernel mode of a running process and the scheduler thread. Thus, `switchnum` of the running process is incremented in `sched()` just before `swtch()` is called.

It must be noted that here a context switch is considered to be the act of storing the context of a process so that another process can be scheduled. Therefore, `switchnum` of the newly scheduled process has not been incremented when the context of the process is restored in `scheduler()` using a `swtch()` call. Further, switching from user mode to kernel mode or vice-versa, for example, during interrupts, of the same process is not considered as context switching (performed by interrupt instruction `int`, and `alltraps` code in `trapasm.S`)

User program `contextswitch` has been added to test the system call.

```
$ contextswitch
context switch counts = 6 6 7 8
```

1.3 Scheduling policy

`Makefile` has been modified in the following way to support scheduling flag `SCHEDFLAG`:

```
ifndef SCHEDFLAG
SCHEDFLAG := DEFAULT
endif
...
CFLAGS += -D $(SCHEDFLAG)
```

Further, to allow process preemption every time quantum `QUANTA`, a field `timescheduled` is created in the structure `proc`. `timescheduled` of a process is assigned `ticks`, every time the process is scheduled. A timer interrupt occurs every `tick`, and`yield()` is called in `trap.c` only if `myproc()->timescheduled + QUANTA <= ticks`.

To implement different scheduling policies, changes have been made to `proc.c`. These changes are enclosed in `ifdef` blocks for compiling the kernel code with different scheduling policies. Apart from First Come First Serve, all other scheduling policies are preemptive and therefore, in their case `yield()` in `trap.c` is called every `QUANTA` units of time.

NOTE: Use `make clean qemu SCHEDFLAG=<flag>` for compiling with appropriate scheduling scheme.

Following are different scheduling policies along with details of implementation and correctness:

1.3.1 Default (`SCHEDFLAG=DEFAULT`)

The default scheduling policy is round-robin. A process is preempted every `QUANTA` units of time.

1.3.2 First Come First Serve (`SCHEDFLAG=FCFS`)

To implement FCFS, a priority queue `pqueue` of ready (`RUNNABLE`) processes is implemented in `proc.c`, which gives higher priority to the process with smaller process creation time. An additional field is created in the structure `proc` to store process creation time `timecreated`. `timecreated` is assigned `ticks` in `allocproc()`.

Whenever the state of a process changes to `RUNNABLE`, the process is inserted into the priority queue. At the time of scheduling, the process with the lowest `timecreated` is removed from the queue and subsequently scheduled.

The priority queue implementation is based on minheap with `O(log n)` insertion and removal time.

1.3.3 Multilevel Queue (`SCHEDFLAG=MLQ`)

To implement MLQ, a multilevel queue `multiqueue` of ready (`RUNNABLE`) processes with `NUMPR` priority levels is implemented. `NUMPR` is defined in `param.h` along with different priority levels. In the current implementation, `NUMPR = 3` with priority level `1` being the highest priority.

Whenever the state of a process changes to `RUNNABLE`, the process is inserted into the multilevel queue. At time of scheduling, a `RUNNABLE` process from a queue is removed for scheduling, only if all the higher level queues are empty. Further, queue data structure ensures round-robin scheduling among the processes with same priority since once a `RUNNING` process yields, the process state is changed to `RUNNABLE` and it is inserted from the other end of the queue than the one from which it was removed.

An additional field `priority` is created in the structure `proc` to store the priority of a given process. The priority of the initial process `initproc` is `2` and priority is inherited upon fork. System call `sys_chpr()` has been added, which calls `chpr()` in `proc.c` to manually change the priority of a process with a given `pid`. If that process is in `RUNNABLE` state, the process has to be transferred to the queue of the given priority.

NOTE: `chpr()` changes priority only when `SCHEDFLAG=MLQ`, otherwise it does nothing.

1.3.4 Dynamic Multilevel Queue (`SCHEDFLAG=DMLQ`)

The implementation of DMLQ is same as MLQ. It uses the same data structure `multiqueue` as MLQ. Further, policies of insertion and removal of processes in `multiqueue` are the same. The only difference is that checkpoints have been created at following points in the kernel code where priority of the processes is changed.

- In `exec.c` the priority of the process which has called `exec` system call is restored to default priority.
- In the function `wakeup1()` in `proc.c`, when a process returns from highest priority, the priority is changed to the highest priority.
- In `trap.c`, when the time-slice of the running process has expired, the priority is reduced by `1`, before calling `yield()`.

NOTE: In MLQ and DMLQ scheduling, if the priority of a `RUNNING` process is changed, the process continues to run until it yields the CPU. The change of priority takes effect from the next run of the process.

1.3.5 Testing the code

User programs, `scheduler_user` and `MLQ_user_check`, have been added for testing the scheduling policies. Further, two new system calls are implemented - `sys_waitnstats()` and `sys_yield()`. `sys_waitnstats()` calls `waitnstats()` in `proc.c`. `waitnstats()` is the same as `wait()` function, but it also returns process statistics. Additional fields have been added to the structure `proc` to maintain process statistics - ready time `timeready`, run time `timerun`, sleeping time `timesleep`. These fields are updated by calling `updateprocesstimes()`, in `proc.c`, every clock tick. On the other hand, `sys_yield()` calls `yield()` in `proc.c`, and is used for manually yielding a process.

Following are the results obtained by `scheduler_user` and `MLQ_user_check`

1. `scheduler_user`

Command line input: `n = 30`

NOTE: For CPU bound processes dummy loop is run `1e9` times, for S-CPU processes dummy loop is run `1e4` times with each dummy loop running for `1e9` iterations, for I/O bound processes dummy sleep calls are run `1e3` times.

1. Default scheduling

```
Average times:
avg run time: 13
avg ready time: 64
avg sleep time: 333
avg turnaround time: 410
```

2. FCFS scheduling

```
Average times:
avg run time: 11
avg ready time: 138
avg sleep time: 333
avg turnaround time: 482
```

3. MLQ scheduling

```
Average times:
avg run time: 10
avg ready time: 86
avg sleep time: 333
avg turnaround time: 429
```

4. DMLQ scheduling

```
Average times:
avg run time: 10
avg ready time: 39
avg sleep time: 333
avg turnaround time: 382
```

We can observe that the average sleep time is the same for all the scheduling policies. This is expected since only I/O processes go into `SLEEPING` state, and the number of I/O processes and their sleep time is the same in each case. Also, the average run time is by and large same. The differences observed in average turnaround time are mostly due to variation in average ready times for different scheduling policies.

Average ready time for FCFS scheduling is the highest because of the convoy effect.

This is followed by default scheduling and MLQ scheduling. Since no `chpr()` calls are invoked in the test program, the priority of all the processes in MLQ scheduling is `2`. This implies that both MLQ and default scheduling are working are as round-robin scheduling policy. The variation in average ready time in the two cases is due to differences in implementation. In MLQ scheduling, a process is always inserted, into one of the ready queues, at the end, whereas, in default scheduling, a process can be inserted into the ready queue at any place (no ready queue exists in default scheduling, however the net effect is the same) .

Finally, the average ready time is the minimum for DMLQ scheduling policy. This is because of dynamic priority rules. For example, when I/O has completed, *i.e.*, the process can return from the `SLEEPING` state, the priority changes to the highest priority. This ensures that the process can run as soon as possible, and reduces the ready time. Similarly, when a process has run for the whole `QUANTA`, the priority reduces by one, so that other higher priority processes can be scheduled quickly, which in turn reduces their ready time. Also, if a child process loads a new program to run using `exec()` system call, the priority is restored to default priority, so that if the child process is not as critical as the parent, it doesn't contribute to the ready time of processes with the highest priority.

Based on the results, the following order is observed among the scheduling policies (in terms of average turnaround time):

$$FCFS < Default \approx MLQ < DMLQ$$

Thus, DMLQ scheduling is the best scheduling policy, since it has the lowest turnaround time.

NOTE: MLQ scheduling is expected to perform better than default, if process priorities were to be manually changed using `chpr` system call

2. `MLQ_user_check`

Command line input: `n = 30`

NOTE: For CPU bound processes dummy loop is run `1e7` times.

`MLQ_user_check` prints process statistics of each of the forked processes, along with average statistics for each priority level.

```
Priority: 1
avg run time: 8
avg ready time: 9
avg sleep time: 0
avg turnaround time: 17

Priority: 2
avg run time: 9
avg ready time: 58
avg sleep time: 0
avg turnaround time: 67

Priority: 3
avg run time: 9
avg ready time: 103
avg sleep time: 0
avg turnaround time: 112
```

Priority 1 is the highest priority level, followed by 2 and 3. Accordingly, the average ready times, and therefore, average turnaround times are lowest for priority 1 and highest for priority 3.

It must be noted that in `MLQ_user_check`, parent process calls `chpr()` for each of the child processes. Therefore, some of the child processes may get scheduled before their priority is changed, and the scheduling order may not be as expected. However, this is nullified by forking a large number of processes and calculating average ready and turnaround times.

Since, the scheduling order may not be always as expected, to verify the correctness of MLQ scheduling, the multilevel queues can be printed. `print()` function, along with some additional code inside `scheduler()`, is commented in `proc.c`, and can be used check the scheduling order while running `MLQ_user_check`.

```c
static void print(void);
...
static void
print(void)
{
  for(int i = 0; i < 3; i++){
    int j = multiqueue.start[i];
    while(j != multiqueue.end[i]){
      cprintf("%d ", multiqueue.proc[i][j]->pid);
      j = (j + 1) % (NPROC+1);
    }
    cprintf("\n");
  }
}
...
// inside scheduler()
// for printing the dequed process and all the queues
cprintf("\npid:%d at %d\n", p->pid, mycpu()->apicid);
print();
```

2. Introduction to Linux Kernel Modules

2.1 Kernel Module Writing (`module km1`)

Following header files are required to write a basic kernel module: `linux/init.h`, `linux/kernel.h`, and `linux/module.h`. All kernel modules must include the header file `linux/module.h`. The kernel module contains two function: `init_module()` and `cleanup_module()`. `init_module()` is called when a module is added to the kernel using `insmod`, and `cleanup_module` is called when the module is removed using `rmmod`. Using `module_init` and `module_exit` macros, we can use any names for the functions `init_module()` and `cleanup_module()`. These macros are defined in `linux/init.h`.

`printk()` has been used to print the messages. By default, `printk()` writes to `/var/log/syslog` as shown below.

```
cryogene@poopypants:~/Documents/ELL783/Assignment1/kernel-modules$ sudo insmod km1.ko
cryogene@poopypants:~/Documents/ELL783/Assignment1/kernel-modules$ sudo rmmod km1.ko
cryogene@poopypants:~/Documents/ELL783/Assignment1/kernel-modules$ tail -n 2 /var/log/syslog
Feb 27 18:24:45 poopypants kernel: [ 5665.421695] Kernel Module Loaded
Feb 27 18:24:47 poopypants kernel: [ 5666.716257] Kernel Module Removed
```

`printk()` function issues messages with different log levels, which describe the message priority. The log level used in this module is `KERN_INFO`, *i.e.* `6` (defined in `linux/kernel.h`). The kernel prints the `printk()` messages if the message priority is higher (lower log level value) than the `console_loglevel`. The current `console_loglevel` is usually `4`. In this implementation, `console_loglevel` is changed to `7` using `setlevel_in.c`. `setlevel_in.c` uses `klogctl()`, which is a wrapper for `syslog` system call to change `console_loglevel`. Similarly `setlevel_rm.c` restores `console_loglevel` to `4`.

Call `make` to compile the code, and use the following commands to add and remove the kernel module:

```
sudo ./setlevel_in
sudo insmod km1.ko
sudo rmmod km1
sudo ./setlevel_rm
```

`printk()` messages will now be displayed on the console.

NOTE: It is not possible to redirect kernel logs and messages to GNOME-terminal. These messages can only be printed on a console.

2.2 Listing the running tasks (`module km2`)

This module is defined in a similar way as above. It uses `for_each_process()`, defined in `linux/sched/signal.h`, to iterate through the list of currently running processes. A process can be in one of the following states: `R` (running or runnable), `S` (sleeping), `D` (sleeping in an uninterruptible wait), `I` (idle state).