ELL783 Operating Systems
Assignment - 1

Bhawna Kumari (2018EE10454)
Preeti Saharan (2018EE10486)

Submitted: 27 February 2022

1 Introduction to the xv6 operating system

1.1 Installing and Testing xv6

QEMU and xv6 operating system were installed following the steps given in the assignment manual. A An extra package *qemu-system-x86* was installed by running the command *sudo apt install qemu-system-x86* to remove the error *Couldn't find a working QEMU executable* while running the command *make qemu*.

1.2 System calls

For adding a new system call following files have to be modified-

- syscall.h - In this file we map a system call name to a unique number like
  #define SYS_ps 22
  This assigns number 22 to system call *ps* which lists the running tasks.
- syscall.c - This file contains the syscalls[] table where we give the pointer to the system call function.
  [SYS_ps] sys_ps - here sys_ps is the pointer Another thing to add here is
  extern int sys_ps(void);
- sysproc.c - This file contains the implementation of the system call function and is called from syscall() function in syscall.c
- usys.S - It contains user level code.
  SYSCALL(ps)
- user.h - It contains the function signature of the system calls which the user can use.
  int ps(void);
- User program and Makefile - If there is a user program then to run it some changes have to be made in the Makefile as given in the assignment manual.
- defs.h- If any other .c file is modified with a function then the function's signature has to be included in the file *defs.h*. For example, *int ps()* is a helper function in proc.c file for *ps* system call, so it is included in *defs.h*.

1.2.1 Listing the running processes

System call name - ps
Function - Lists the currently running processes.
User program name - process_list.c
Command to run - process_list
Implementation - We will travserse through the whole page table and check if a process is in *RUNNING* state and if it is then we will print it.
Figure 1 shows the code snippet of *ps* and figure 2 shows the output of the user program.

```
int
ps(void)
{
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == RUNNING)
      cprintf("pid:%d name:%s\n", p->pid, p->name);
  }
  release(&ptable.lock);
  return 22;
}
```

Figure 1: Code of *ps* in file proc.c

```
$ process_list
pid:4 name:process_list
```

Figure 2: Output of user program *process_list*

1.2.2 Printing the available memory

System call name - memtop
Function - tells the amount of memory available or free(in bytes) in the system
User program name - available_mem.c
Command to run - available_mem
Implementation - For this we have counted the number of free pages in the memory by traversing the *freelist* of the memory allocator and then multipying the no. of free pages with the size of a page in bytes to get the total free available memory. This is done by creating a function *int availableMemory()* in the file kalloc.c which returns the available memory.
Figure 3 shows the code snippet of *memtop* and figure 4 shows the output of the user program.

```
int
availableMemory(void)
{
  int count = 0;
  struct run *r = kmem.freelist;
  while(r){
    count++;
    r = r->next;
  }
  return count*PGSIZE;
}
```

Figure 3: Helper function for *memtop* in file kalloc.c

```
$ available_mem
available memory: 232603648
```

Figure 4: Output of user program *available_mem*

1.2.3 Context switching

System call name - csinfo
Function - To find the number of context switches of a process.
User program name - context_switch.c
Command to run - context_switch
Implementation - For this we have a new variable named *numSwitch* in the process struct in proc.h file to count the number of context switches. When a new process is created and allocated memory, then we initialize *numSwitch = 0* in allocproc() function. Every time the scheduler runs a particular process, we increment its numSwitch by 1. So the system call just returns the process' numSwitch value.
Output - Since cs1 and cs2 are updated consecutively and there will not be in context switch of the process, so the first two values cs1 and cs2 will be same. Then we are calling sleep for 1 time unit so value of cs3 will be one greater than cs2 and again sleep is called so value of cs4 will be one greater than cs3. Hence the output is like cs1 = cs2 = x(let) , cs3= x+1 and cs4 = x+2.
Figure 5 shows the code snippet of *csinfo* and figure 6 shows the output of the user program.

1.3 Scheduling Policy

1.3.1 Current Scheduling Policy

- Code location - in proc.c file *scheduler* and *sched* functions
- Scheduling Policy - Round robin where each process runs until a timer interrupt.

```
int
sys_csinfo(void)
{
  return myproc()->numSwitch;
}
```

Figure 5: Code for *csinfo* in file sysproc.c

```
$ context_switch
context switch counts = 8 8 9 10
```

Figure 6: Output of user program *context_switch*

- What happens when a process returns from an I/O operation? - for an I/O operation to happen the process goes into sleeping state and invokes *sched()* which executes *swtch* to switch context and save the current process's state and the control goes to *scheduler()* which will then loop over the process table to find a RUNNABLE process and start running it. After the IO operation is complete, wakeup system call changes the state of all the process sleeping to RUNNABLE so that they can be scheduled again.
- What happens when a new process is created and when/how often does the scheduling take place? - the *userinit()* function creates the first user process and after that all child processes are created by using fork. *allocproc()* is used to initialize the process parameters and memory. Since it is a round robin scheduler, each processor runs for about 100ms before a timer interrupt takes place.

1.3.2 First Come First Serve: (SCHEDFLAG= FCFS)

FCFS is a non-preemptive policy and the processes are run on the basis of their creation time. For this we have created a new variable *ctime* in process struct in proc.h file. When a process is created and allocated memory and its paramters are initialized in the function *allocproc()* then we define the *ctime* of the process as:
p− >ctime = ticks;
where ticks is the counter for time used in xv6. Figure 7 shows the code for FCFS. In FCFS we traverse the process table and find the process with least creation time and run it. The ready times depend on the sequence in which processes are created and FCFS can lead to starvation.

1.3.3 Multi-level Queue Scheduling: (SCHEDFLAG= MLQ)

MLQ is a preemptive policy so for this we have used QUANTA = 5 and the processes are run on the basis of their priority. For this we have created a new variable *priority* in process struct in proc.h file. When a process is created, it is assigned a priority 2. When a child process is created using fork, then the parent's priority is copied to the child also. In MLQ we traverse the process table and find the process with the highest priority and run it. To prevent starvation, round robin policy is followed in each priority queue by keeping track of the current process which is being executed and the next highest priority we need to execute . Figure 8 shows the code for MLQ. In this we can change the priority of the process using the system call *chpr* which takes the process pid and required priority as argument. Figure 9 shows the code for *chpr*.

1.3.4 Dynamic Multi-level Queue Scheduling: (SCHEDFLAG=DMLQ)

DMLQ is same as MLQ with the following modifications:

- We cannot use *chpr* to change priority manually.
- In file exec.c (line 101-105) which contains the code of exec system call, we check if the macro DMLQ is defined and if it is then we change the process' priority to 2.
- In the function *wakeup1()* (file proc.c, line 692-695) which is called when we return from sleeping mode, again we check if the macro DMLQ is defined and if it is then we change the process' priority to 1(highest priority).
- After every time quanta the priority is decreased by 1 if it is greater than 1. Figure 10 shows the snippet of code for this. In this code after every time quanta, for MLQ and DMLQ we yield the CPU and for DMLQ we also decrease the priority.

1.3.5 Computing statistics - ready, run, sleep and turnaround time

To find the ready time, run time and sleep time of a process we created three new variables *readytime, runtime, sleeptime* in process struct in proc.h and they are updated using the function *computeTime()* as shown in Figure 11. The function *computeTime()* traverses the process table and then depending on the state of the process it increases the ready (RUNNABLE state), run (RUNNING state) or sleep (SLEEPING state) time of that process. This function is run i.e. the time values are updated in every tick of the clock. To make these values available to the

```c
#ifdef FCFS
acquire(&ptable.lock);

struct proc *proc_min_ctime = 0;

// Loop over the process table to find the process with minimum ctime
struct proc *current_process = ptable.proc;
while(current_process < &ptable.proc[NPROC]){
  if(current_process->state == RUNNABLE){
    if(proc_min_ctime == 0){
      proc_min_ctime = current_process;
    }
    else if(current_process->ctime < proc_min_ctime->ctime){
      proc_min_ctime = current_process;
    }
  }
  current_process++;
}

if(proc_min_ctime){
  // Switch to chosen process.
  c->proc = proc_min_ctime;
  switchuvm(proc_min_ctime);
  proc_min_ctime->state = RUNNING;

  // Increase the no. of context switches.
  proc_min_ctime->numSwitch = proc_min_ctime->numSwitch + 1;

  swtch( &(c->scheduler),proc_min_ctime->context);
  switchkvm();

  // Process is done running for now.
  // It should have changed its p->state before coming back.
  c->proc = 0;
}

release(&ptable.lock);
```

Figure 7: Code for FCFS in file proc.c

user side, a new system call *waitstat* is created which is a slight modification of the wait system call and takes three integer pointers **readytime, *runtime, *sleeptime* as argument. In *waitstat* whenever a zombie process is found then during its clean up we update the values at these pointers so that we can retrieve them at the user end.

1.3.6 Testing the code

Performance comparison of all three policies
A user program *schedule_user* tests the three policies. In this program, we are yielding the CPU for short tasks based CPU bound processes so yield system call was implemented additionally. Table 1 shows the average statistics for all the scheduling policies and three types of tasks and figure 12 shows a sample output in terminal. To run it type command-
make clean qemu SCHEDFLAG=FCFS/MLQ/DMLQ
schedule_user (No of. processes to be created)

We have the following observations from the statistics:

- For IO processes, since we are running a loop of 1000 iterations and calling sleep system call, hence the sleep time for IO in all 4 policies is 1000. The sleep time for other cases is zero.
- For the three different types of process, the average ready time is in order: CPU <S-CPU <IO and this order is present in FCFS, MLQ and DMLQ. S-CPU tasks are yielded for 100 times so they spend more time in runnable state and for IO tasks sleep() system call is called 1000 times so when they return from sleep state then they go into runnable state hence we get this order.
- For the default scheduling policy, the average ready time is in the order: S-CPU <IO <CPU. This may be because of the frequent context switching due to timer interrupt which causes the CPU bound tasks to go into runnable state for more time.

Table 1: schedule_user - Average statistics for all policies for 10 processes

Policy	Process type	Ready time	Run Time	Sleep time	Turnaround time
DEFAULT	CPU	2005	4070	0	6075
DEFAULT	S-CPU	112	0	0	112
DEFAULT	IO	500	0	1000	1500
FCFS	CPU	1343	4319	0	5662
FCFS	S-CPU	2030	0	0	2030
FCFS	IO	4090	0	1000	5090
MLQ	CPU	1425	4497	0	5922
MLQ	S-CPU	2142	0	0	2142
MLQ	IO	3072	0	1000	4072
DMLQ	CPU	1727	5104	0	6831
DMLQ	S-CPU	3889	0	0	3889
DMLQ	IO	3960	0	1000	4960

- The average run time is non - zero for CPU bound processes in all the policies as we have added extra loops in the code to consume CPU time.
- For DMLQ as compared to other policies, the ready time of IO process is more or less same to that of S-CPU process because in DMLQ policy we are updating the priority of the process to highest after it returns from sleep.
- For CPU bound process we see that the performance is comparable for all 3 policies.

Performance of MLQ policy

A user program *MLQ_user_check* tests MLQ policy by creating n processes (taken as argument and n >5) and then changing their priority to either 1,2 or 3 depending on a condition. Table 2 shows the average statistics for each priority and overall average for 15 processes and figure 13 shows sample output in terminal. To run it type command-
make clean qemu SCHEDFLAG=MLQ
MLQ_user_check (No of. processes to be created)
Based on the statistics, we have the following observations:

- The average ready time for the 3 priorities is in the order: 1 <2 <3. This is because we are executing all the processes with higher priority before moving to any lower priority process.
- Since there are no explicit sleep calls so the sleep time is zero.
- The run times are comparable for differen priorities.

Table 2: MLQ_user_check - Average statistics for all priorities for 15 processes

Priority	Ready time	Run Time	Sleep time	Turnaround time
1	98	43	0	141
2	101	42	0	144
3	258	52	0	311
Overall average	153	46	0	199

2 Introduction to Linux Kernel Modules

Header files

- linux/module.h - Need to include for every module
- linux/kernel.h - used it for printk
- linux/sched/signal.h - included it to make use of function "for_each_process()" to iterate over all the process.

Meta Information

MODULE_LICENSE("GPL") - GNU Public License

MODULE_AUTHOR("Preeti") - Author of the module
MODULE_DESCRIPTION("..") - Gives a short description about the module.

2.1 Kernel module writing

1. int init_module(void)
It is called whenever this module is loaded into kernel.It uses printk to print the message "Kernel Module Loaded". We used KERN_INFO log level(priority = 6). Returns 0 when successfully loaded.Figure 14 and Figure 15 shows corresponding code and output.
2.cleanup_module(void)
It is called whenever module is removed from kernel. Uses printk to print the message "Kernel Module Removed".
3. Commands
make clean
make
sudo insmod KernelModuleWriting.ko - inserted module
dmesg -t—tail -1 - used tail -1 to print recent 1 line to print the message written in init_module.
sudo rmmod KernelModuleWriting - removed module

2.2 Listing the running tasks

1. int init_module(void)
It is called whenever this module is loaded into kernel. Used for_each_process() to go through all the tasks, choose only those process which are runnable using the state of process. Created a function get_task(state) to obtain state-(runnable,interruptible, or other). Then used printk to print task id, task command and task state. Figure 16 and Figure 17 shows corresponding code and output.
2.cleanup_module(void)
It is called whenever module is removed from kernel. Uses printk to print the message "ListingRunningTasks Kernel Module Removed".
5. Commands
make clean
make
sudo insmod ListingRunningTasks.ko
dmesg
sudo rmmod ListingRunningTasks

```
#ifdef MLQ
acquire(&ptable.lock);
struct proc *proc_high_priority = 0, *p, *new_sml_proc = sml_proc;
for(p = sml_proc; p < &ptable.proc[NPROC]; p++){
  if(p->state == RUNNABLE){
    if(proc_high_priority == 0){
      proc_high_priority = p;
      new_sml_proc = p;
    }
    if(p->priority < proc_high_priority->priority){
      proc_high_priority = p;
      new_sml_proc = p;
    }
  }
}
if(proc_high_priority){
  c->proc = proc_high_priority;
  switchuvm(proc_high_priority);
  proc_high_priority->state = RUNNING;
  // No. of clock cycles run by the process in a quanta.
  proc_high_priority->ticks_elapsed = 0;
  // Increase the no. of context switches.
  proc_high_priority->numSwitch = proc_high_priority->numSwitch + 1;
  swtch(&(c->scheduler), proc_high_priority->context);
  switchkvm();
  // Process is done running for now.
  // It should have changed its p->state before coming back.
  c->proc = 0;

  int next_max_priority = 3;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->priority < next_max_priority){
      next_max_priority = p->priority;
    }
  }
  if(next_max_priority == proc_high_priority->priority){
    new_sml_proc++;
    if(new_sml_proc < &ptable.proc[NPROC])
      sml_proc = new_sml_proc;
    else
      sml_proc = ptable.proc;
  }
  else{
    sml_proc = ptable.proc;
  }
}
release(&ptable.lock);
#endif
```

Figure 8: Code for MLQ in file proc.c

```c
// Changes priority of the process with given pid.
// Returns 1 after changing priority if a process is found with given pid.
// Returns -1 if no process found with given pid.
int
chpr(int pid, int priority)
{
  struct proc *curr;
  acquire(&ptable.lock);
  for(curr = ptable.proc; curr < &ptable.proc[NPROC]; curr++){
    if(curr->pid == pid){
      curr->priority = priority;
      release(&ptable.lock);
      return 1;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

Figure 9: Code for *chpr* in file proc.c

```c
// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to check nlock.
#if defined(MLQ) || defined(DMLQ)
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER && (++myproc()->ticks_elapsed) == QUANTA){
  #ifdef DMLQ
  if(myproc()->priority > 1)
    myproc()->priority -= 1;
  #endif
  yield();
}
#endif
```

Figure 10: Code run after every time QUANTA in file trap.c

```c
void
computeTime(void)
{
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == RUNNING)
      p->runtime++;
    else if(p->state == SLEEPING)
      p->sleeptime++;
    else if(p->state == RUNNABLE)
      p->readytime++;
  }
  release(&ptable.lock);
}
```

Figure 11: Code for computing statistics in file proc.c

```
$ schedule_user 10
Printing statistics for each process --------
PID | Type | Ready time | Run time | Sleep time | Turnaround time
4    | S-CPU| 0          | 0          | 0          | 0
7    | S-CPU| 4          | 0          | 0          | 4
6    | CPU  | 0          | 4021          | 0          | 4021
9    | CPU  | 4          | 4016          | 0          | 4020
10    | S-CPU| 4020          | 1          | 0          | 4021
13    | S-CPU| 4022          | 0          | 0          | 4022
5    | IO   | 4019          | 0          | 1000          | 5019
8    | IO   | 4021          | 0          | 1000          | 5021
11    | IO    | 4023          | 0          | 1000          | 5023
12    | CPU   | 4020          | 4730          | 0          | 8750

Printing Average Statistics -------
Type    | Avg Ready time | Avg Run time | Avg Sleep time | Avg turnaround time
CPU     | 1341              | 4255              | 0              | 5597
S-CPU | 2011              | 0              | 0              | 2011
IO      | 4021              | 0              | 1000              | 5021
```

Figure 12: Output for schedule_user with FCFS

```
$ MLQ_user_check 15
Printing statistics for each process --------
PID | Priority | Ready time | Run time | Sleep time | Turnaround time
15    | 1          | 0          | 56          | 0          | 56
16    | 2          | 9          | 54          | 0          | 63
18    | 1          | 61          | 40          | 0          | 101
19    | 2          | 62          | 40          | 0          | 102
21    | 1          | 102          | 40          | 0          | 142
22    | 2          | 105          | 39          | 0          | 144
24    | 1          | 144          | 40          | 0          | 184
25    | 2          | 146          | 40          | 0          | 186
27    | 1          | 186          | 39          | 0          | 225
28    | 2          | 187          | 39          | 0          | 226
17    | 3          | 227          | 46          | 0          | 273
20    | 3          | 229          | 46          | 0          | 275
23    | 3          | 267          | 41          | 0          | 308
26    | 3          | 269          | 40          | 0          | 309
29    | 3          | 301          | 90          | 0          | 391

Printing Average Statistics -------
Priority    | Avg Ready time | Avg Run time | Avg Sleep time | Avg turnaround time
1        | 98              | 43              | 0              | 141
2        | 101              | 42              | 0              | 144
3        | 258              | 52              | 0              | 311

For all process with MLQ scheduling policy
Average ready time = 153
Average run time = 46
Average sleep time = 0
Average turnaround time = 199
```

Figure 13: Output for MLQ_user_check with 15 processes

```c
#include <linux/module.h> /* need to include for every kernel module */
#include <linux/kernel.h> /* used it only for printk log levels */

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Preeti");
MODULE_DESCRIPTION("A simple module which prints KERNEL MODULE LOADED message when loaded and KERNEL MODULE REMOVED w

int init_module(void)
{
        printk(KERN_INFO "Kernel Module Loaded\n");/*KERN_INFO priority = 6 , current console log level = 4*/
        return 0;

        /* 0 is returned when module is successfully loaded, in case of an error--> o will not be returned*/
}

void cleanup_module(void)
{
        printk(KERN_INFO "Kernel Module Removed\n");
}
```

Figure 14: Code for init and cleanup functions

```
[ 3739.580081] Kernel Module Loaded
[ 3772.309816] Kernel Module Removed
```

Figure 15: Output-printing module loaded and removed message

```c
char * task_state(long state)
{

    switch (state) {
        case TASK_RUNNING:
            return "TASK_RUNNING";

        default:
        {

            return "TASK_NOT_RUNNING";
        }
    }
}
int init_module(void)
{

        struct task_struct *task; // Pointer to Stask

        for_each_process(task)
        {

                if(strcmp(task_state(task->state), "TASK_RUNNING") == 0){   /* picking only the processes which are running*/
                printk( KERN_INFO "task_command: %s | state: %ld | process_id: %d\n", task->comm, task->state, task->pid );
                }
                /*printk( KERN_INFO "task_command: %s | state: %ld | process_id: %d \n", task->comm, task->state, task->pid);*/

        }

        return 0;

        /* 0 is returned when module is successfully loaded */
}

void cleanup_module(void)
{
        printk(KERN_INFO "ListingRunningTasks Kernel Module Removed\n");
}
```

Figure 16: Code for module of listing running tasks

```
[ 1171.884926] task_command: gnome-terminal- | state: 0 | process_id: 3397
[ 1171.884931] task_command: insmod | state: 0 | process_id: 4768
```

Figure 17: Output-printing running processes