Operating Systems

Assignment-1

Guruvu Surya Sai Prakash
2019EE10481
Vineeth Kumar Ponuganti
2019EE10545

February 27, 2022

1 Introduction to the xv6 operating system

1.1 Adding System calls

We typically need to modify 5 files to add our own system call.They are:

1. syscall.h
2. usys.S
3. user.h
4. syscall.c
5. sysproc.c

In **syscall.h**, we add the system call number.(Ex: define SYS_syscall_name 25).
In **usys.S**, we add SYSCALL(syscall_name), whivh acts as a interface to user program system call.
In **user.h**, we add the function prototype of the wrapper function which the user program is using to call the system call. Call to below function utilizes the macro defined in usys.S and will move the corresponding system call number to %eax register and will call an interrupt.
In **syscall.c**, we add pointers to the syscall which maps the system call number to corresponding function. The function is not defined in the file, hence we add the protype of the function.(Ex: extern int syscall_name(void)).
In **sysproc.c**, we implement the actual system call.

1.1.1 Listing Running Process

Here, in addition to above 5 files, we modify proc.c and defs.h.
In xv6, we have a ptable which keeps account of all the processes. To find the currently running process we must walk through this ptable, but it is accessible only in proc.c. So, we implement the function here and add the signature of the function in defs.h, which we call in sysproc.c.
We acquire the ptable lock and then traverse to locate the processes which has state assigned as RUNNING and print the process pid and name and then release the lock.

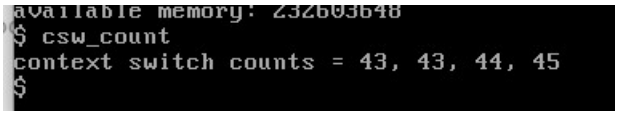
```
process_list   2 18 14964
avail_memory   2 19 15064
csw_count      2 20 15412
console        3 21 0
$ process_list
pid:4 name:process_list
$
```

1.1.2 Printing the available memory

In xv6 the free pages are maintained by the freelist linked list which is present inside kmem struct. This is accessible only inside kalloc.c.So, we implement the function here and later call it inside sysproc.c.
So,we traverse through this linked list to find the number of free pages, and later multiply this number with the PGSIZE, which gives us the available memory.

1.1.3 Context switching

For, this we add a additional data member switch_count to proc struct, whose value will be incremented each time it preempts another process (i.e., each time when it is scheduled).
Now, for finding the total context switches occurred, we traverse the ptable and count the total no of context switches.



Since, we have sleep call between cs2 and cs3, they differ by 1. Similarly, cs3 and cs4.

1.1.4 Change priority

Added a system call int chpr(int pid,int priority). We traverse through the process table and change the priority of the process with the given pid.
If there is no process with given pid then we just return -1.

1.2 Scheduling Policies

The default scheduling policy does Round robin with premption done for each clock tick, but we changed it a QUANTA=(5 Clock ticks), for this we have modified trap.c, where we call yield after a QUANTA, instead of a clock tick.
For this we have added a new data member schedTime to proc struct, which is assigned to 0 each time the process is scheduled.Now, when the process has ran for a QUANTA (i.e., schedTime=QUANTA), we call yield. The schedTime is incremented each time when the timer ends.
The code looks as below

```
1    if(myproc() && myproc()->state == RUNNING &&
2       tf->trapno == T_IRQ0+IRQ_TIMER){
3
4       myproc()->schedTime=myproc()->schedTime + 1;
5
6          if(myproc()->schedTime == QUANTA){
7              yield();
8          }
9       }
```

1.2.1 Scheduling-FCFS

For this we added a data member createTime to proc struct. Now, each time we loop over the processes and find the process with state=RUNNABLE and having the minimum createTime and schedule it.

Now, we don't want to the processes to be preempted, so, we do not call yield for each quanta when the SCHEDFLAG=FCFS.
The code looks as below,

```
1      struct proc *FC_proc=0;
2
3      // Loop over process table looking for process to run.
4      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
5        if(p->state != RUNNABLE)
6          continue;
```

```
      if(FC_proc==0 || p->createTime < minCreateTime){
         minCreateTime=p->createTime;
         FC_proc=p;
      }
    }


    if(FC_proc){

      c->proc = FC_proc;
      switchuvm(FC_proc);
      FC_proc->state = RUNNING;
      int a=FC_proc->switch_count;
      FC_proc->switch_count=a+1;
      FC_proc->schedTime=0;


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

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

1.2.2 Scheduling-MLQ

For this we add another data member priority to the proc-struct.We assign each process a priority of 2 for each processes when we allocate the process.
Now, we first loop through the table find a process with higher priority (i.e., lower value). If there is no process then only we go to the next high priority process. We do Round-robin for the processes with same priority, for which we take help of a pointer for each priority. Which is defined inside a array arr of three pointers, each for each priority.
The code for priority=1 is as follows

```
struct proc* MLQ_p1(int *arr){
    struct proc *MLQ_proc=0;
    struct proc *p;

    int p1=0;
    int count=arr1[0];

    while(p1<NPROC){
      p=&ptable.proc[count];

      if(p->state == RUNNABLE && p->priority==1){
        MLQ_proc=p;
        count=(count+1)%NPROC;
        break;
      }
      p1+=1;
      count=(count+1)%NPROC;

    }

    arr[0]=count;

    if(MLQ_proc){

      return MLQ_proc;
    }
    else{
      return MLQ_p2(arr1);
    }
```

```
30 }
```

The pointer arr[0] is modified each time a process is scheduled. So,that next time if we have another process with same priority, we schedule it. So, we could follow Round-robin. We are also traversing the ptable in a circular manner.

1.2.3 Scheduling-DMLQ

The code is almost same as before, that we change the exec.c file such that, when the exec call is called then we reset the priority as 2.

We changed trap.c such that we reduce the priority by 1, after running the whole quanta. The code is as follows

```
1     if(myproc()->schedTime == QUANTA){
2         if(myproc()->priority!=3){
3             myproc()->priority=myproc()->priority+1;
4         }
5         yield();
6     }
```

We changed proc.c such that after a process returning from a sleep, we assign it a priority of 1.

```
1  static void
2  wakeup1(void *chan)
3  {
4    struct proc *p;
5
6    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
7      if(p->state == SLEEPING && p->chan == chan){
8        p->state = RUNNABLE;
9        #ifdef DMLQ
10       p->priority = 1; //Highest priority for processes returning from sleeping mode.
11       #endif
12     }
13 }
```

1.2.4 The output of user program schedule_user

For this we added three additonal data members(runTime,readyTime,sleepTime) and added getStatistics system call. The getStatistics system call will return values of these three data members of a child process which is now a zombie.
FCFS:

```
pid: 6, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 7, process: scpu_bound, readyTime: 0 , runTime: 2 , sleepTime: 0 , turnaroundTime: 2
pid: 9, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 10, process: scpu_bound, readyTime: 1 , runTime: 0 , sleepTime: 0 , turnaroundTime: 1
pid: 12, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 13, process: scpu_bound, readyTime: 1 , runTime: 1 , sleepTime: 0 , turnaroundTime: 2
pid: 15, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 16, process: scpu_bound, readyTime: 2 , runTime: 0 , sleepTime: 0 , turnaroundTime: 2
pid: 18, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 19, process: scpu_bound, readyTime: 1 , runTime: 2 , sleepTime: 0 , turnaroundTime: 3
pid: 21, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 22, process: scpu_bound, readyTime: 2 , runTime: 0 , sleepTime: 0 , turnaroundTime: 2
pid: 24, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 25, process: scpu_bound, readyTime: 1 , runTime: 1 , sleepTime: 0 , turnaroundTime: 2
pid: 27, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 28, process: scpu_bound, readyTime: 1 , runTime: 1 , sleepTime: 0 , turnaroundTime: 2
pid: 30, process: cpu_bound, readyTime: 0 , runTime: 1 , sleepTime: 0 , turnaroundTime: 1
pid: 31, process: scpu_bound, readyTime: 1 , runTime: 1 , sleepTime: 0 , turnaroundTime: 2
pid: 33, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 34, process: scpu_bound, readyTime: 4 , runTime: 4 , sleepTime: 0 , turnaroundTime: 8
pid: 5, process: io_bound, readyTime: 222 , runTime: 791 , sleepTime: 989 , turnaroundTime: 2002
pid: 8, process: io_bound, readyTime: 1015 , runTime: 12 , sleepTime: 976 , turnaroundTime: 2003
pid: 11, process: io_bound, readyTime: 1029 , runTime: 8 , sleepTime: 964 , turnaroundTime: 2001
pid: 14, process: io_bound, readyTime: 1039 , runTime: 8 , sleepTime: 960 , turnaroundTime: 2007
pid: 17, process: io_bound, readyTime: 1050 , runTime: 10 , sleepTime: 953 , turnaroundTime: 2013
pid: 20, process: io_bound, readyTime: 1064 , runTime: 7 , sleepTime: 943 , turnaroundTime: 2014
pid: 23, process: io_bound, readyTime: 1068 , runTime: 21 , sleepTime: 930 , turnaroundTime: 2019
pid: 26, process: io_bound, readyTime: 1091 , runTime: 14 , sleepTime: 919 , turnaroundTime: 2024
pid: 29, process: io_bound, readyTime: 1106 , runTime: 18 , sleepTime: 909 , turnaroundTime: 2033
pid: 32, process: io_bound, readyTime: 1127 , runTime: 16 , sleepTime: 895 , turnaroundTime: 2038
```

```
Average Statistics

CPU-Bound Statistics
Average readyTime: 0
Average runTime: 0
Average sleepTime: 0
Average turnAroundTime: 0

SCPU-Bound Statistics
Average readyTime: 1
Average runTime: 1
Average sleepTime: 0
Average turnAroundTime: 2

IO-Bound Statistics
Average readyTime: 981
Average runTime: 90
Average sleepTime: 943
Average turnAroundTime: 2015
```

MLQ

```
$ schedule_user 6
pid: 6, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 7, process: scpu_bound, readyTime: 0 , runTime: 2 , sleepTime: 0 , turnaroundTime: 2
pid: 9, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 10, process: scpu_bound, readyTime: 1 , runTime: 1 , sleepTime: 0 , turnaroundTime: 2
pid: 12, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 13, process: scpu_bound, readyTime: 1 , runTime: 1 , sleepTime: 0 , turnaroundTime: 2
pid: 15, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 16, process: scpu_bound, readyTime: 2 , runTime: 1 , sleepTime: 0 , turnaroundTime: 3
pid: 18, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 19, process: scpu_bound, readyTime: 2 , runTime: 2 , sleepTime: 0 , turnaroundTime: 4
pid: 21, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 22, process: scpu_bound, readyTime: 1 , runTime: 1 , sleepTime: 0 , turnaroundTime: 2
pid: 24, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 25, process: scpu_bound, readyTime: 1 , runTime: 1 , sleepTime: 0 , turnaroundTime: 2
pid: 27, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 28, process: scpu_bound, readyTime: 2 , runTime: 0 , sleepTime: 0 , turnaroundTime: 2
pid: 30, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 31, process: scpu_bound, readyTime: 1 , runTime: 1 , sleepTime: 0 , turnaroundTime: 2
pid: 33, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 34, process: scpu_bound, readyTime: 7 , runTime: 3 , sleepTime: 0 , turnaroundTime: 10
pid: 5, process: io_bound, readyTime: 121 , runTime: 896 , sleepTime: 988 , turnaroundTime: 2005
pid: 8, process: io_bound, readyTime: 1019 , runTime: 15 , sleepTime: 974 , turnaroundTime: 2008
pid: 11, process: io_bound, readyTime: 1040 , runTime: 22 , sleepTime: 952 , turnaroundTime: 2014
pid: 14, process: io_bound, readyTime: 1063 , runTime: 22 , sleepTime: 932 , turnaroundTime: 2017
pid: 17, process: io_bound, readyTime: 1089 , runTime: 25 , sleepTime: 910 , turnaroundTime: 2024
pid: 20, process: io_bound, readyTime: 1114 , runTime: 21 , sleepTime: 893 , turnaroundTime: 2028
pid: 23, process: io_bound, readyTime: 1131 , runTime: 28 , sleepTime: 874 , turnaroundTime: 2033
pid: 26, process: io_bound, readyTime: 1164 , runTime: 16 , sleepTime: 857 , turnaroundTime: 2037
pid: 29, process: io_bound, readyTime: 1183 , runTime: 25 , sleepTime: 829 , turnaroundTime: 2037
pid: 32, process: io_bound, readyTime: 1197 , runTime: 21 , sleepTime: 827 , turnaroundTime: 2045
Average Statistics

CPU-Bound Statistics
Average readyTime: 0
Average runTime: 0
Average sleepTime: 0
Average turnAroundTime: 0

SCPU-Bound Statistics
Average readyTime: 1
Average runTime: 1
Average sleepTime: 0
Average turnAroundTime: 3

IO-Bound Statistics
Average readyTime: 1012
Average runTime: 109
Average sleepTime: 903
Average turnAroundTime: 2024
$
```

DMLQ

```
pid: 6, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 7, process: scpu_bound, readyTime: 2 , runTime: 3 , sleepTime: 0 , turnaroundTime: 5
pid: 9, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 10, process: scpu_bound, readyTime: 2 , runTime: 0 , sleepTime: 0 , turnaroundTime: 2
pid: 12, process: cpu_bound, readyTime: 0 , runTime: 1 , sleepTime: 0 , turnaroundTime: 1
pid: 13, process: scpu_bound, readyTime: 1 , runTime: 1 , sleepTime: 0 , turnaroundTime: 2
pid: 15, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 16, process: scpu_bound, readyTime: 0 , runTime: 2 , sleepTime: 0 , turnaroundTime: 2
pid: 18, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 19, process: scpu_bound, readyTime: 2 , runTime: 0 , sleepTime: 0 , turnaroundTime: 2
pid: 21, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 22, process: scpu_bound, readyTime: 2 , runTime: 0 , sleepTime: 0 , turnaroundTime: 2
pid: 24, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 25, process: scpu_bound, readyTime: 0 , runTime: 2 , sleepTime: 0 , turnaroundTime: 2
pid: 27, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 28, process: scpu_bound, readyTime: 1 , runTime: 1 , sleepTime: 0 , turnaroundTime: 2
pid: 30, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 31, process: scpu_bound, readyTime: 0 , runTime: 2 , sleepTime: 0 , turnaroundTime: 2
pid: 33, process: cpu_bound, readyTime: 0 , runTime: 0 , sleepTime: 0 , turnaroundTime: 0
pid: 34, process: scpu_bound, readyTime: 9 , runTime: 1 , sleepTime: 0 , turnaroundTime: 10
pid: 5, process: io_bound, readyTime: 997 , runTime: 83 , sleepTime: 931 , turnaroundTime: 2011
pid: 8, process: io_bound, readyTime: 985 , runTime: 93 , sleepTime: 935 , turnaroundTime: 2013
pid: 11, process: io_bound, readyTime: 983 , runTime: 97 , sleepTime: 932 , turnaroundTime: 2012
pid: 14, process: io_bound, readyTime: 975 , runTime: 104 , sleepTime: 933 , turnaroundTime: 2012
pid: 17, process: io_bound, readyTime: 977 , runTime: 103 , sleepTime: 928 , turnaroundTime: 2008
pid: 20, process: io_bound, readyTime: 982 , runTime: 93 , sleepTime: 930 , turnaroundTime: 2005
pid: 23, process: io_bound, readyTime: 976 , runTime: 98 , sleepTime: 935 , turnaroundTime: 2009
pid: 26, process: io_bound, readyTime: 994 , runTime: 89 , sleepTime: 930 , turnaroundTime: 2013
pid: 29, process: io_bound, readyTime: 986 , runTime: 102 , sleepTime: 923 , turnaroundTime: 2011
pid: 32, process: io_bound, readyTime: 988 , runTime: 96 , sleepTime: 926 , turnaroundTime: 2010
Average Statistics
```

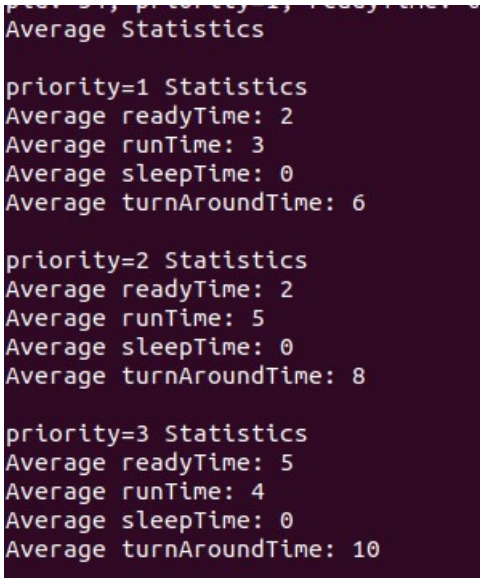
```
CPU-Bound Statistics
Average readyTime: 0
Average runTime: 0
Average sleepTime: 0
Average turnAroundTime: 0

SCPU-Bound Statistics
Average readyTime: 1
Average runTime: 1
Average sleepTime: 0
Average turnAroundTime: 3

IO-Bound Statistics
Average readyTime: 984
Average runTime: 95
Average sleepTime: 930
Average turnAroundTime: 2010
```

If we look into the IO-bound statistics.We could see that DMLQ has lower turn around time compared to FCFS and MLQ. This is because after returning from sleep the process is attaining higher priority.
Also, Turn around time for FCFS is lower than MLQ. This is because, MLQ is nothing but round-robin in this case, because all the processes have same priority of 2.But, for finding the runnable processs, we have to traverse the ptable twice, once to check for a process with priority=1 and then priority=2. So,it has larger turn around time.

1.2.5 Output of user program-MLQ_user_check



The turn-around time are in the order priority-1<priority-2<priority-3, which was what actually we expected.

1.3 Extra Code added to actual xv6

1.3.1 For system calls

A . Files Modified

1 . Modified the 5 files discussed earlier.
2 . Modified proc.c, kalloc.c, defs.h, for adding ps, csinfo, memtop, chpr, genstat system calls.

B. Test Programs
1 . process_list.c for ps syscall.
2 . avail_memory.c for memtop.
3 . csw_count.c for csinfo.

1.3.2 For scheduling

A . Files Modified

1 . Modified trap.c and params.h for Round-robin with quanta.
2 . Modified proc.c and trap.c for FCFS.
3 . Modified proc.c and trap.c for MLQ.
4 . Modified proc.c,exec.c and trap.c for DMLQ.
B. Test Programs

1. schedule_user.c
2. MLQ_user_check.c

2 Introduction to Linux Kernel Modules

A typical LKM has two functions. One is function_init and function_exit which are called when a module is loaded and when a module is cleaned up respectively.

2.1 Kernel Module Writing

Generally, printk logs the messages in kernel log buffer. Now, for printing to the console, it depends on loglevel. loglevel decides the importance of the message. The kernel decides whether to print the message to console or not depending on the current console log level.

If log level of the message is lower than (higher priority) the current console log level, then the message is printed to the console immediately.
In our code we are using KERN_EMERG to log our message which has the loglevel 0, so it get printed to console.
In the folder **Ass21.c** represents the module for this part.

2.2 Listing the running tasks

To find the running tasks we need to traverse through all the process. In linux each process has a process descriptor (task_struct), which has the information of state,pid, etc.

In linux we have a circular doubly-linked list, which we can traverse using for_each_process() MACRO present in sched/signal.h, which loops over this linked list. Using this we can print the running processes.

The code of MACRO is as follows

```
#define for_each_process(p) \
        for (p = &init_task ; (p = next_task(p)) != &init_task ; )

#define next_task(p)  list_entry((p)->tasks.next, struct task_struct, tasks)
```

We print the processses which have p− >state= TASK_RUNNING, which account for all the running processes.
In the folder **Ass22.c** represents the module for this part.