# CS 344-OS_LAB Assignment-2 Report

**Group Name: G7**
**Submission Date:14-10-2020**

**Team Members**:
Ashish Kumar Baranwal (180123006) Dept:MnC
Bhargab Gautom (180123008) Dept:MnC
Harsh Yadav (180123015) Dept:MnC
Karan Gupta (180123064) Dept:Mnc

## Question A) Implementing system Calls

The process for implementing a system call and user program was described in report of Lab assignment-1.

1. Implement **getNumProc()** and **getMaxPid()** : To add both these system calls we followed the same process of adding system call and user program which we mentioned in lab1 report except one step. These system call requires some information to be read from the process table which can't be accessed in sysproc.c file, so the actual code/ function was implemented in the file proc.c instead of sysproc.c. Below are the screenshot from proc.h file which shows the structure of proc after some changes and from terminal of outputs of userprograms of getNumProctest and getMaxPidtest:

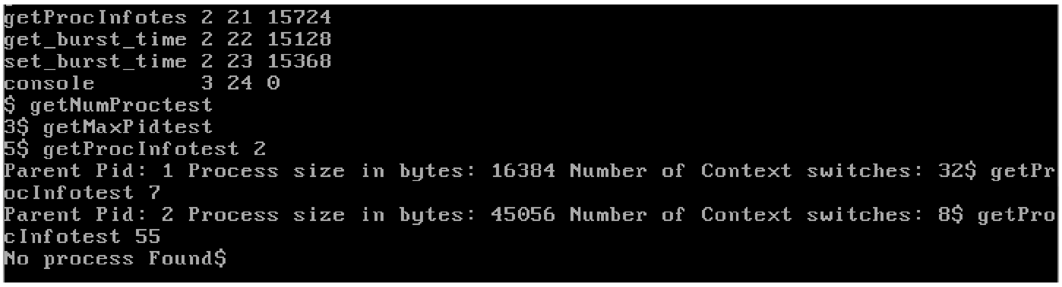
```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int numcntxtswtchs;          // Number of context switches
  int bursts;                  // Burst time
};
```

```
getNumProctest 2 19 15112
getMaxPidtest  2 20 15104
getProcInfotes 2 21 15724
get_burst_time 2 22 15128
set_burst_time 2 23 15368
console        3 24 0
$ getNumProctest
3$ getMaxPidtest
5$
```

- **getNumProc():** This system call should return the number of active processes in the system. We implemented this function by iterating over every process of the ptable and counting the processes whose state was not 'UNUSED' (i.e. either in embryo, running, runnable, sleeping, or zombie states) by using 'p->state' which tells the state of the process (we determined this by observing the process struct defined in proc.h file, which is shown in above screenshot). This function as said earlier was implemented in proc.c file (line no. 539-563) and was just called in the sysproc.c (line no. 155-159) under the function **int sys_getNumProc(void).** User program file name for getNumProc() is getNumProctest.c which outputs the number of active processes in the system on qemu terminal.

- **getMaxPid():** This system call should return the maximum PID amongst the PIDs of all currently active in the system. This function was also implemented in the similar manner like the last system call and a variable maxpid was initialised to keep the max pid amongst the active processes in the loop. Actual code/function was implemented in proc.c file (line no. 565-581) and was called in sysproc.c (line no. 161-165). User program name for is getMaxPidtest which outputs Maximum pid amongst active processes in the system on qemu terminal.

2. **getProcInfo(pid, &processInfo):** **processInfo** is a pointer to a **structure processInfo** passed as an argument, this structure is used for passing information between user and kernel mode. On calling this system call this function returns either 0 which means the necessary process information of the given PID is added to the pointer passed as argument and -1 if such a process with passed Pid doesn't exist. With the help of user program this call outputs the parent id, process size and number of context switches or No process found on 0 and -1 respectively. To implement this following piece of code were added:
    - proc.h: At line no. 52 "**int numcntxtswtchs;**" // Number of context switches
      This adds a new variable to proc struct to keep a count of context switches for the process.
    - proc.c: At line no. 92 **"p->numcntxtswtchs = 0;"** //Initialise count with 0 in allocproc function
      At line no. 347 **"p->numcntxtswtchs=p->numcntxtswtchs+1;"** //Increase this count by 1 in scheduler function when the process switches it's state
      From line no. 583-600: code that adds info to the processInfo pointer parent pid and process size can be determined "**a->ppid=p->parent->pid;" "process_a->psize=(int)p->sz;"**
    - other files which needs to be changed for adding system call and user program



```
getProcInfotes 2 21 15724
get_burst_time 2 22 15128
set_burst_time 2 23 15368
console        3 24 0
$ getNumProctest
3$ getMaxPidtest
5$ getProcInfotest 2
Parent Pid: 1 Process size in bytes: 16384 Number of Context switches: 32$ getPr
ocInfotest 7
Parent Pid: 2 Process size in bytes: 45056 Number of Context switches: 8$ getPro
cInfotest 55
No process Found$
```

3. Implement **set_burst_time(n)** and **get_burst_time()**: These two system calls were also implemented in a similar manner with user programs set_burst_timetest and get_burst_timetest respectively. These functions were implemented with the help of inbuilt function myproc() which stores the pointer to current running process. On calling the user program with an input between 1 and 20, set_burst_time system call sets the burst time of the process as given input and outputs Invalid Input if the input is not in the given range. Whereas get_burst_time system call returns the burst time of current process. To implement this following piece of code were added:
    - proc.h: At line no. 53 "**int bursts**" //Burst time
      This adds a new variable to proc struct to store the burst time for the process.
    - proc.c: Line no. 602-605 code for set_burst_time(n)
      Line no. 607-610 code for get_burst_time()

- sysproc.c: handled input using argint() for set_burst_time line no.184-191
- other files which needs to be changed for adding system call and user program

```
get_burst_time 2 22 15136
set_burst_time 2 23 15376
console        3 24 0
$ set_burst_timetest 6
Burst time set successfully$ set_burst_timetest 31
Invalid Input$ get_burst_time
6$ _
```

## **Question B)** Implementing a Shortest job first + Round Robin hybrid scheduler

The following files were created/edited for implementing the hybrid scheduler algorithm:

### 1. proc.c :

- Added default value of 100 for burst time
- Created new scheduling algorithm based on **Shortest Job First** (modified void scheduler(void))
- This algorithm finds the shortest job available before running each process, so it is of O(n) time complexity for scheduling.

```c
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

      struct proc *lowBursts = 0;
      struct proc *p1 = 0;

      if(p->state != RUNNABLE)
        continue;

      // choose process with lowest bursts
      lowBursts = p;
      for(p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++){
        if((p1->state == RUNNABLE) && (lowBursts->bursts > p1->bursts))
          lowBursts = p1;
      }

      if(lowBursts != 0)
        p = lowBursts;

      if(p != 0)
      {
        // Switch to chosen process.  It is the process's job
        // to release ptable.lock and then reacquire it
        // before jumping back to us.
        c->proc = p;
        switchuvm(p);
        p->state = RUNNING;

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

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

  }
}
```

- Added function int cps(void) which lists processes and their burst values for debugging
- int chbrst(int pid, int bursts) : It is used to change the burst time to bursts for process with process id pid.
- Locks were implemented in the scheduler to make sure the code is safe to run over multiple CPU cores using
  - acquire(&ptable.lock);
  - release(&ptable.lock);

### SJF Scheduler in O(logn) time complexity (using min_heap as process table)

- We also tried implementing the SJF scheduling algorithm by creating a min heap and storing all the runnable processes in it along with the pre-existing process table with the process with the least burst time on top of heap.
- The idea was to return the top element of the heap until the heap was empty, instead of linearly interating through the process table for the lowest burst time and executing it, as done in the above O(n) algorithm.
- The implementation of the priority queue/min heap as the process table is shown below:

```
struct Q{
    struct spinlock lock;
    struct proc proc[NPROC];
    int heapSize;
    int pidmap[NPROC+5];
};

struct Q pqueue = {
    .heapSize = 0
};


void insert_pqueue(struct proc p) {
    pqueue.heapSize++;
    pqueue.proc[pqueue.heapSize] = p; /*Insert in the last place*/
    /*Adjust its position*/
    int now = pqueue.heapSize;
    while (pqueue.proc[now / 2].bursts > p.bursts) {
        pqueue.proc[now] = pqueue.proc[now / 2];
        now /= 2;
    }
    pqueue.proc[now] = p;
}

struct proc pop_pqueue() {
    int child, now;
    struct proc minElement, lastElement;
    minElement = pqueue.proc[1];
    lastElement = pqueue.proc[pqueue.heapSize--];
    /* now refers to the index at which we are now */
    for (now = 1; now * 2 <= pqueue.heapSize; now = child) {
        /* child is the index of the element which is minimum among both the children */
        /* Indexes of children are i*2 and i*2 + 1*/
        child = now * 2;
        if (child != pqueue.heapSize && pqueue.proc[child + 1].bursts < pqueue.proc[child].bursts) {
            child++;
        }
        /* To check if the last element fits ot not it suffices to check if the last element
         is less than the minimum element among both the children*/
        if (lastElement.bursts > pqueue.proc[child].bursts) {
            pqueue.proc[now] = pqueue.proc[child];
        } else /* It fits there */
        {
            break;
        }
    }
    pqueue.proc[now] = lastElement;
    return minElement;
}
```

We changed the way of retrieving lowest burst time process in the scheduler function as follows:

```
acquire(&pqueue.lock);


while(pqueue.heapSize > 0) {
    struct proc lowBursts = pop_pqueue();   //Complexity of pop_pqueue() is O(logn)
    int ptable_index = pqueue.pidmap[lowBursts.pid];
    p = &ptable.proc[ptable_index];


  if(p != 0)
  {
    // Switch to chosen process.  It is the process's job
    // to release ptable.lock and then reacquire it
    // before jumping back to us.
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;

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

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

release(&pqueue.lock);
```

**Hybrid Scheduler (using heapsort and queue)**

- We also implemented a hybrid scheduling algorithm based on sorting all the runnable processes in ptable based on burst times and then storing them in a queue. Then using Round-Robin scheduling (as already there in xv6) to run the processes from the queue.
- This algorithm used <u>heap sort</u> for sort the processes according to burst times and <u>array data structure</u> to store the processes in a queue.
- This gave a random execution order of execution of processes and not in strictly increasing order of burst times (as in case of SJF).
- The time complexity will be O(logn) because heap sort(O(nlogn) is applied for n processes which gives O(logn) time for each process.
- Following is the code of the hybrid scheduler implemented:

```c
void heapify(struct proc* arr[], int n, int i)
{
    int largest = i;
    int l = 2*i + 1;
    int r = 2*i + 2;

    if (l < n && arr[l]->bursts > arr[largest]->bursts)
        largest = l;

    if (r < n && arr[r]->bursts > arr[largest]->bursts)
        largest = r;

    if (largest != i)
    {
        struct proc* tmp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = tmp;


        heapify(arr, n, largest);
    }
}

void heapSort_queue(struct proc* arr[], int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    for (int i=n-1; i>0; i--)
    {

        struct proc* tmp = arr[0];
        arr[0] = arr[i];
        arr[i] = tmp;

        heapify(arr, i, 0);
    }
}
```

```c
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  struct proc *queue[NPROC];
  int qsize = 0;
  for(;;){
    // enable interrupts
    sti();

    acquire(&ptable.lock);
    qsize = 0;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state == RUNNABLE)
        queue[qsize++] = p;
    }

    heapSort_queue(queue, qsize);
//  for(int i = 0; i < qsize; i++) {
//      printf(1,"Queue elt %d bursts = %d\n", i+1, queue[i]->bursts);
//    sort_queue(queue, qsize);

    for(int pitr = 0; pitr < qsize; pitr++){
      p = queue[pitr];
      if(p->state != RUNNABLE)
        continue;
      if(p != 0)
      {
        // Switch to chosen process.  It is the process's job
        // to release ptable.lock and then reacquire it
        // before jumping back to us.
        c->proc = p;
        switchuvm(p);
        p->state = RUNNING;

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

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

    release(&ptable.lock);
  }
```

- **param.h** : Changed NCPU to 1, for easier debugging

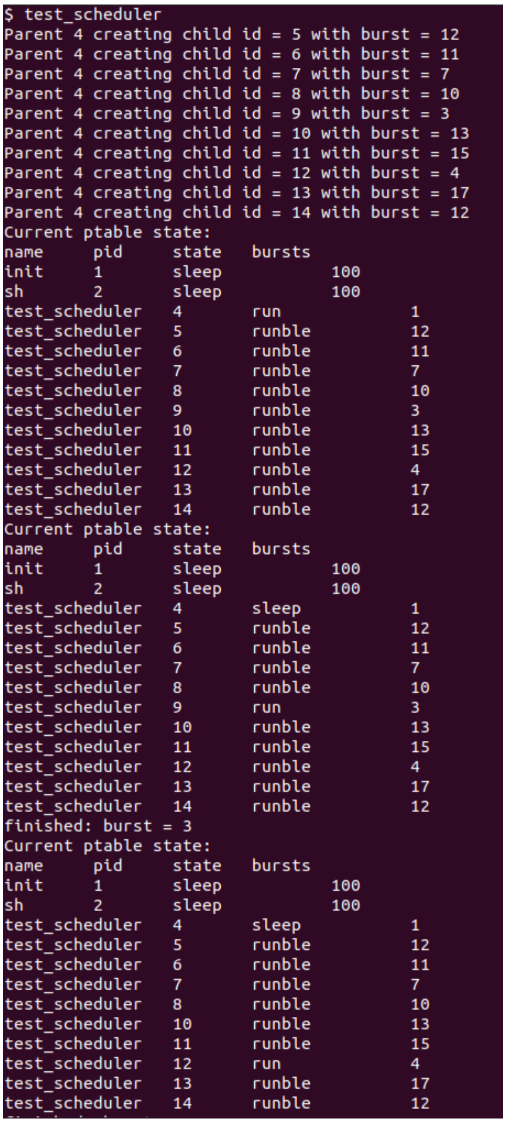
Following new system calls were added:

- **Int cps(void)**: prints status of currently active processes in the ptable namely, their name, pid, state(sleeping, runnable, running etc.) and bursts.
- **Int chbrst(int pid, int burst):** changes burst time to burst for process with process id pid
  To implement these system calls, the following code was added:
  - defs.h: Line 130 – function defintion was added
  - proc.c : Lines 856-870 – code for chbrst(int,int) function

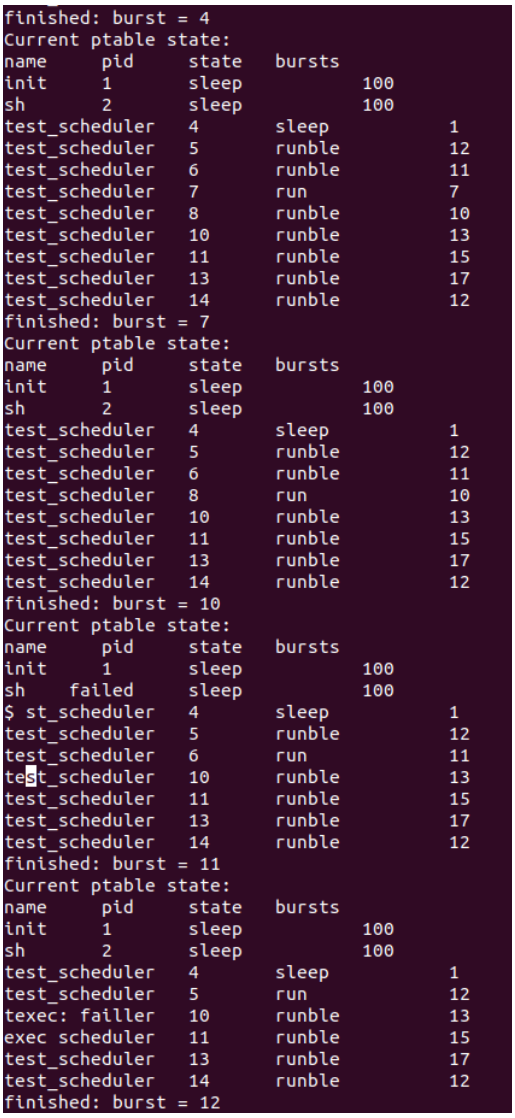
For testing the newly implemented scheduler, the following test schedulers were created, which ran for various test cases with different burst times and CPU-bound or I/O bound processes.

**test_scheduler.c and test_scheduler_io.c:** User programs to test new scheduling algorithm:

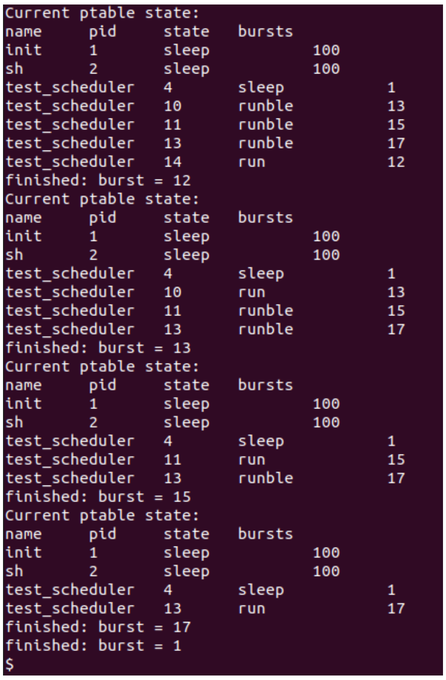
- Forked 10 child processes with different burst times set for each process.
- After forking and adding all the child processes to the ptable, the scheduler executed all the processes starting with the lowest burst time and then terminating and removing it from the table.
- Added dummy arithmetic calculations to use cpu time for simulating burst time of each process, doing the number of arithmetic operations proportional to the assigned burst time for each process in order to simulate the running of the process more accurately. Hence, a process with higher assigned burst time was taking longer to run as compared to one with short burst times.
- test_scheduler_io.c for I/O bound processes; used mkdir() (along with arithmetic operations) in each child process to write to disk.
- This program returned execution order of all the child processes which was in ascending order of their burst times. In Shortest job First, after executing each process, the scheduler looks for the shortest available job and executes it. So, it schedules the process in O(n) time on an average.
- Output for the test_scheduler run and the state of ptable after execution of each process is shown below:



```
$ test_scheduler
Parent 4 creating child id = 5 with burst = 12
Parent 4 creating child id = 6 with burst = 11
Parent 4 creating child id = 7 with burst = 7
Parent 4 creating child id = 8 with burst = 10
Parent 4 creating child id = 9 with burst = 3
Parent 4 creating child id = 10 with burst = 13
Parent 4 creating child id = 11 with burst = 15
Parent 4 creating child id = 12 with burst = 4
Parent 4 creating child id = 13 with burst = 17
Parent 4 creating child id = 14 with burst = 12
Current ptable state:
name      pid     state    bursts
init      1       sleep             100
sh        2       sleep             100
test_scheduler    4       run               1
test_scheduler    5       runble            12
test_scheduler    6       runble            11
test_scheduler    7       runble            7
test_scheduler    8       runble            10
test_scheduler    9       runble            3
test_scheduler    10      runble            13
test_scheduler    11      runble            15
test_scheduler    12      runble            4
test_scheduler    13      runble            17
test_scheduler    14      runble            12
Current ptable state:
name      pid     state    bursts
init      1       sleep             100
sh        2       sleep             100
test_scheduler    4       sleep             1
test_scheduler    5       runble            12
test_scheduler    6       runble            11
test_scheduler    7       runble            7
test_scheduler    8       runble            10
test_scheduler    9       run               3
test_scheduler    10      runble            13
test_scheduler    11      runble            15
test_scheduler    12      runble            4
test_scheduler    13      runble            17
test_scheduler    14      runble            12
finished: burst = 3
Current ptable state:
name      pid     state    bursts
init      1       sleep             100
sh        2       sleep             100
test_scheduler    4       sleep             1
test_scheduler    5       runble            12
test_scheduler    6       runble            11
test_scheduler    7       runble            7
test_scheduler    8       runble            10
test_scheduler    10      runble            13
test_scheduler    11      runble            15
test_scheduler    12      run               4
test_scheduler    13      runble            17
test_scheduler    14      runble            12
```



```
finished: burst = 4
Current ptable state:
name      pid     state    bursts
init      1       sleep             100
sh        2       sleep             100
test_scheduler    4       sleep             1
test_scheduler    5       runble            12
test_scheduler    6       runble            11
test_scheduler    7       run               7
test_scheduler    8       runble            10
test_scheduler    10      runble            13
test_scheduler    11      runble            15
test_scheduler    13      runble            17
test_scheduler    14      runble            12
finished: burst = 7
Current ptable state:
name      pid     state    bursts
init      1       sleep             100
sh        2       sleep             100
test_scheduler    4       sleep             1
test_scheduler    5       runble            12
test_scheduler    6       runble            11
test_scheduler    8       run               10
test_scheduler    10      runble            13
test_scheduler    11      runble            15
test_scheduler    13      runble            17
test_scheduler    14      runble            12
finished: burst = 10
Current ptable state:
name      pid     state    bursts
init      1       sleep             100
sh    failed      sleep             100
$ st_scheduler    4       sleep             1
test_scheduler    5       runble            12
test_scheduler    6       run               11
test_scheduler    10      runble            13
test_scheduler    11      runble            15
test_scheduler    13      runble            17
test_scheduler    14      runble            12
finished: burst = 11
Current ptable state:
name      pid     state    bursts
init      1       sleep             100
sh        2       sleep             100
test_scheduler    4       sleep             1
test_scheduler    5       run               12
texec: failler    10      runble            13
exec scheduler    11      runble            15
test_scheduler    13      runble            17
test_scheduler    14      runble            12
finished: burst = 12
```



```
Current ptable state:
name      pid     state    bursts
init      1       sleep             100
sh        2       sleep             100
test_scheduler    4       sleep             1
test_scheduler    10      runble            13
test_scheduler    11      runble            15
test_scheduler    13      runble            17
test_scheduler    14      run               12
finished: burst = 12
Current ptable state:
name      pid     state    bursts
init      1       sleep             100
sh        2       sleep             100
test_scheduler    4       sleep             1
test_scheduler    10      run               13
test_scheduler    11      runble            15
test_scheduler    13      runble            17
finished: burst = 13
Current ptable state:
name      pid     state    bursts
init      1       sleep             100
sh        2       sleep             100
test_scheduler    4       sleep             1
test_scheduler    11      run               15
test_scheduler    13      runble            17
finished: burst = 15
Current ptable state:
name      pid     state    bursts
init      1       sleep             100
sh        2       sleep             100
test_scheduler    4       sleep             1
test_scheduler    13      run               17
finished: burst = 17
finished: burst = 1
$
```