CS 344 OS Lab

Assignment 2

Mathematics and Computing

Group 2

by
Aayush Suthar, 220123004
Aaditya Jalan, 220123001
Avanish Gadhikar, 220123075
Chaitanya Chhabra, 220123012

The patch file for the lab can be found here: Assignment 2 Drive - Group 2 MNC, CS344

Task A

Q1- Add support for tracking the number of "tickets" in a process: adding a system call called settickets which sets the number of "tickets" for a process with the following prototype:int settickets(int number). By default, processes should have 10 tickets.

Q2- Add a system call called getprocessesinfo with the following prototype
*int getprocessesinfo(struct processes_info *p);*

Final Output

```
init: starting sh
$ processlist
3 running processes
PID     TICKETS TICKS
1       10      24
2       10      18
3       10      8
MAX PID: 3
Process Count: 3
Attempting to fetch pid 3
p->pid = 3
p->psize = 12288
p->numberContextSwitches = 9
Burst time: 5
$
```

```
$ timewithtickets 500 100 100 100 100
TICKETS TICKS
100     125
100     125
100     125
100     125
$
```

An exhaustive list of all the changes we made:

1. **proc.h**

Added the three variables as shown below.

```
// 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)
  /// NEW BELOW
  int ticks;
  int tickets;              // Added tickets variable to incorporate Lottery Scheduling.
  int burst_time;
};
```

2. **proc.c**

Modified function *scheduler* → Added Ticket system

```c
/* TASK A VOID SCHEDULER (TICKET WALA)
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  int total_tickets, winning_ticket, current_ticket_sum;
  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Total tickets initialissed ot 0.
    total_tickets = 0;

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);

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

    if(total_tickets > 0){
      winning_ticket = rand() % total_tickets;

      current_ticket_sum = 0;
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE)
          continue;

        current_ticket_sum += p->tickets;

        if(current_ticket_sum > winning_ticket){
          c->proc = p;
          switchuvm(p);
          p->state = RUNNING;
          p->ticks++;
          swtch(&(c->scheduler), p->context);
          switchkvm();
```

3. **usys.S**

Added SYSCALLs at the end of declarations

```
SYSCALL(settickets)
SYSCALL(getprocessesinfo)
SYSCALL(yield)
```

4. **syscall.h**

```c
#define SYS_settickets 22         // Added system call set tickets and assigned it a number.
#define SYS_getprocessesinfo 23 // Added getprocessesinfo to the list of system calls.
#define SYS_yield 24              // Added yield to the list of available system calls.
```

5. **user.h**

Added forward declarations to the list in this file

```c
int settickets(int);                            // Adding support for system call.
int getprocessesinfo(struct processes_info*); // Adding support for system call.
void yield(void); // Added as timewith tickets invoked yield, and support was needed.
```

6. **sysproc.c**

Added functions with argument error-catching.

```
// defining the settickets system call
int
sys_settickets(void){
  int n;
  if (argint(0, &n) < 0)   // Retrieve the integer argument passed from user space
      return -1;
  if (n < 1 || n > 100000) // Validate the number of tickets
      return -1;
  myproc()->tickets = n;       // Set the number of tickets for the current process
  return 0;
}

int
sys_getprocessesinfo(void){
// Get the pointer to the processes_info structure
struct processes_info *pinfo;
  if(argptr(0, (void*)&pinfo, sizeof(*pinfo)) < 0)
    return -1;
  return getprocessesinfo(pinfo);
}

void
sys_yield(void){
  yield();
}
```

7. **syscall.c**

Added extern forward declarations

```
extern int sys_settickets(void); // added the definition of settickets.
extern int sys_getprocessesinfo(void); // added the definition of getprocessesinfo.
extern int sys_yield(void); // Added the definition of yield.
```

```
[SYS_settickets] sys_settickets,
[SYS_getprocessesinfo] sys_getprocessesinfo,
[SYS_yield] sys_yield,
```

8. **defs.h**

Added forward declaration for yield

```
void            yield(void);
```

9. **[NEW] processlist.c**

Made this file to print out the output of the task

```c
#include "types.h"
#include "mmu.h"
#include "param.h"
#include "proc.h"
#include "user.h"
#include "processInfo.h"

int main(int argc, char *argv[])
{
    struct processes_info info;
    getprocessesinfo(&info);
    printf(1, "%d running processes\n", info.num_processes);
    printf(1, "PID\tTICKETS\tTICKS\n");
    for (int i = 0; i < info.num_processes; ++i) {
        printf(1, "%d\t%d\t%d\n", info.pids[i], info.tickets[i], info.ticks[i]);
    }

    struct processInfo p;
    int x = getmaxpid();
    printf(1, "MAX PID: %d\n", x);
    printf(1, "Process Count: %d\n", getnumproc());
    printf(1, "Attempting to fetch pid %d\n", x);
    if (getprocinfo(x, &p)!=-1)
        printf(1, "p->pid = %d\np->psize = %d\np->numberContextSwitches = %d\n", p.ppid, p.psize, p.numberContextSwitches);
    else printf(1, "Bad process ID. Exit with code -1\n");

    set_burst_time(5);
    // // burst times
    // for (int i = 0; i < info.num_processes; ++i) {
    //     printf(1, "Burst for pid = %d is %d\n", info.pids[i]);
    // }
    printf(1, "Burst time: %d\n", get_burst_time());
    exit();
}
```

10. Makefile

Made changes to two places in this file: UPROGS and EXTRA

```makefile
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
    _processlist\
    _timewithtickets\
    _lotterytest\
```

```
EXTRA=\
    mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
    ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
    printf.c umalloc.c timewithtickets.c processlist.c lotterytest.c\
    README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
    .gdbinit.tmpl gdbutil\
```

Task B

We have implemented system call getNumProc(), to return the total number of active processes in the system (either in embryo, running, runnable, sleeping, or zombie states) and system call getMaxPid() that returns the maximum PID amongst the PIDs of all currently active processes in the system.

Final Output of lotterytest

```
$ lotterytest
one process: passed 3 of 3
two processes, unequal ratio: passed 7 of 7
two processes, unequal ratio, small ticket count: passed 7 of 7
two processes, equal: passed 7 of 7
two processes, equal, small ticket count: passed 7 of 7
three processes, unequal: passed 9 of 9
three processes, unequal, small ticket count: passed 9 of 9
three processes, but one io-wait: passed 9 of 9
three processes, but one exits: passed 9 of 9
seven procceses: passed 17 of 17
two processes, not all yielding: passed 8 of 8
overall: passed 92 of 92
$
```

An exhaustive list of all the changes we made:

11. **proc.h**

Added the burst time variable as shown below.

```
// 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)
  /// NEW BELOW
  int ticks;
  int tickets;             // Added tickets variable to incorporate Lottery Scheduling.
  int burst_time;
};
```

12. proc.c

Modified function *scheduler*

```
void
scheduler(void)
{
  struct proc *p, *p1;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();
    struct proc *highP;
    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      highP = p;
      //choose one with highest burst_time
      for(p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++){
        if(p1->state != RUNNABLE)
          continue;
        if(highP->burst_time > p1->burst_time)   //larger value, lower burst_time
          highP = p1;
      }
      p = highP;
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;
      p->ticks++;

      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);

  }
}
```

13. usys.S

Added SYSCALLs at the end of declarations

```c
#define SYS_getnumproc 25
#define SYS_getmaxpid 26
#define SYS_getprocinfo 27
#define SYS_set_burst_time 28
#define SYS_get_burst_time 29
```

14. syscall.h

```c
#define SYS_getnumproc 25
#define SYS_getmaxpid 26
#define SYS_getprocinfo 27
#define SYS_set_burst_time 28
#define SYS_get_burst_time 29
```

15. user.h

Added forward declarations to the list in this file

```c
int getnumproc(void);
int getmaxpid(void);
int getprocinfo(int, struct processInfo*);
int set_burst_time(int);
int get_burst_time(void);
```

16. sysproc.c

Added functions with argument error-catching.

```
int
sys_getnumproc(void){
  return getnumproc();
}
int
sys_getmaxpid(void){
  return getmaxpid();
}

int
sys_getprocinfo(void){
  int pid;
  struct processInfo* pinfo;
   if(argint(0, &pid) < 0)
    return -1;
   if(argptr(1, (void*)&pinfo, sizeof(struct processInfo*)) < 0)
    return -1;
  return getprocinfo(pid,pinfo);
}

int
sys_set_burst_time(void){
  int n;
  if(argint(0, &n) < 0)
    return -1;
  return set_burst_time(n);
}

int
sys_get_burst_time(void){
  return get_burst_time();
}
```

17. syscall.c

Added extern forward declarations

```
extern int sys_getnumproc(void);
extern int sys_getmaxpid(void);
extern int sys_getprocinfo(void);
extern int sys_set_burst_time(void);
extern int sys_get_burst_time(void);
```

```
[SYS_getnumproc] sys_getnumproc,
[SYS_getmaxpid] sys_getmaxpid,
[SYS_getprocinfo] sys_getprocinfo,
[SYS_set_burst_time] sys_set_burst_time,
[SYS_get_burst_time] sys_get_burst_time,
```

18. defs.h

Added forward declarations

```
int             getprocessesinfo(struct processes_info*);
int             getnumproc(void);
int             getmaxpid(void);
int             getprocinfo(int, struct processInfo*);
int             set_burst_time(int);
int             get_burst_time(void);
```

19. Makefile

Made changes to two places in this file: UPROGS and EXTRA

```
UPROGS=\
	_cat\
	_echo\
	_forktest\
	_grep\
	_init\
	_kill\
	_ln\
	_ls\
	_mkdir\
	_rm\
	_sh\
	_stressfs\
	_usertests\
	_wc\
	_zombie\
	_processlist\
	_timewithtickets\
	_lotterytest\
```

```
EXTRA=\
    mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
    ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
    printf.c umalloc.c timewithtickets.c processlist.c lotterytest.c\
    README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
    .gdbinit.tmpl gdbutil\
```

Task C

We have modified the scheduler to take into account user-defined process burst time and implement a shortest job first scheduler.

As we made the set_burst_time and get_burst_time system calls in the previous task, we will now use them in our user program sjftest.c that shows that the burst times cause the child processes to behave differently.

Testing functionality of out user program sjftest.c

```
for(int i=0;i<n;i++){
    burst_times[i] = 1+rand()%20;
}

printf(1, "\tPID \t Type \t Burst Time \t Context Switches\n");
printf(1, "\t___ \t ____ \t __________ \t ________________\n\n");
for(int i=0;i<n;i++){
    if(!fork()){

        // CPU Bound process
        set_burst_time(burst_times[i]);

        int it=0;
        for(int j=0;j<100;j++){
            for(int k=0;k<1000000;k++){
                if((j+k)%2)it++;
                else it--;
            }
        }
        useless+=it;

        struct processInfo info;
        getprocinfo(getpid(), &info);
        int bt = get_burst_time();
        if(bt<10)
            printf(1, "\t%d \t CPU         %d                  %d\n", getpid(), bt, info.numberContextSwitches);
        else
            printf(1, "\t%d \t CPU         %d                 %d\n", getpid(), bt, info.numberContextSwitches);
        exit();


    }
}
```

We set the burst time and then fork ‘n’ processes, each of which will now be scheduled as per the shortest job first algorithm, which will in turn consider the burst times of the processes as set by the user (generated randomly in this program).

Sjfttest.c output

We have changed the code for the scheduler in proc.c such that instead of using round robin scheduling it uses the shortest job first for scheduling .
In order to implement the shortest job first, we iterated through all the processes and if we got a process that has a burst time less than the burst time of the current process then we will select that process for running.

Complexity Analysis

We iterate through n = NPROC processes to find the one with the minimum burst time
This takes Θ(n) steps

Next we call the switchuvm and swtch APIs on this process, which is only a constant Θ(1) overhead.

The final time complexity is therefore Θ(n).

The results of the test are as follows:

```
sjftest 10
```

PID	Type	Burst Time	Context Switches
40	CPU	2	26
43	CPU	3	27
45	CPU	5	27
38	CPU	8	29
37	CPU	9	27
44	CPU	10	28
41	CPU	13	29
36	CPU	14	27
39	CPU	15	29
42	CPU	16	27

Below is the image of the scheduler code.

Scheduler for part A:

```
/* TASK C PART A Scheduler */
// void
// scheduler(void)
// {
//   struct proc *p = 0;
//   struct cpu *c = mycpu();
//   c->proc = 0;

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

//     //process with minimum bursttime be infinite
//     int minbt = 100000000;

//     // Loop over process table looking for process to run.
//     acquire(&ptable.lock);
//     struct proc *highP = 0;
//     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

//       //continue to next iteration if state is not runnable
//       if(p->state!=RUNNABLE)
//         continue;

//       //if found a process with shorter burst time than current we take that process
//       if(minbt > p->burst_time){
//                 //updating current process
//                 highP = p;
//                 //updating current minimum burst time
//                 minbt = p->burst_time;
//       }
//       // Switch to chosen process.  It is the process's job
//       // to release ptable.lock and then reacquire it
//       // before jumping back to us.
//     }
//     p = highP;

//     if(p==0){
//         release(&ptable.lock);
//         continue;
//       }
//       c->proc = p;
//       switchuvm(p);
//       p->state = RUNNING;
//       if(p->burst_time>1)
//       swtch(&(c->scheduler), p->context);
//       switchkvm();

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

//   }
// }
```

Task C [Bonus]

We have implemented the Shortest Job Hybrid Round Robin, as described in the assignment. This required changes in the following locations:

1. Makefile
2. Proc.c
3. We have made a file pqueue.h

We made a struct for the priority queue and then we have function like heapify, insert, isEmpty, extractMin

1. Heapify is basically a method that makes the values in the array to follow the basic property of a heap
2. Insert is a method that can used for the insertion in the heap
3. isEmpty is a method that checks whether the heap is empty and it returns true when heap is empty
4. extractMin is a method that gives the minimum value that is present in the heap and it also removes that min value in the heap

Using this data structure min heap we have implemented the hybrid of round robin and shortest job first scheduling in the scheduler instead of round robin scheduling
To implement this we have maintained a priority queue and when we get a job that is shorter than the current job we will preempt the running process and make the job with a shorter burst time run.

Below are all the changes we made in the files:

Proc.c

```c
/* Task C part B (bonus) */
void
scheduler(void)
{
  // struct pqueue pq;
  initPQueue(&pq);
  struct proc *p1 = myproc();
  struct proc *p = p1;
  struct cpu *c = mycpu();
  c->proc = 0;
  static int turn = 0;
  for(;;){
    // Enable interrupts on this processor.
    sti();

    //process with minimum bursttime be infinite
    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    struct proc* allprocs[64];
    struct proc *initp=0, *shellp=0;
    int size=0;
    // insert(&pq, p1);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if (p->state == RUNNABLE){
        if (p->pid>2) {allprocs[size]=p; size++;}
        else{
          if (p->pid==1) initp = p;
          if (p->pid==2) shellp = p;
        }
      }
```

```
    for(int i=0; i<size; i++){
      p = allprocs[i];
      if(p->pid == 1 || p->pid == 2){

      }
      p->burst_time -= 2;
    }
    p1 = extractMin(&pq);
    if (size != 0)
      p = allprocs[0];
    else{
      turn = 1 - turn;
      p = initp;
      if (shellp && !turn) p = shellp;
    }

    if(p==0){
        release(&ptable.lock);
        continue;
      }
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;
      if(p->burst_time>1)
      swtch(&(c->scheduler), p->context);
      switchkvm();

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

  }
}
```

Complexity Analysis

We fill a priority queue data structure with every process from the ptable. Assuming n processes, this heapify operation takes O(nlogn) time.

ExtractMin from a heap allows us to pop the current min burst time process in only O(logn) time.

Next we spend a time quantum (constant, we took =2) on this process which is only a constant Θ(1) overhead. The extracted process can be enqueued again when the round robin restarts.

This procedure repeats every round for all of the n processes. So each round has nϴ(logn) = ϴ(nlogn) time taken. The total number of rounds is at most max_burst_time/time_quantum = 20/2 = 10 = ϴ(1). So the overall complexity turns out to be O(nlogn).

We have also made a test program bonustest.c for checking the scheduling process

Bonustest.c output

```
$ bonustest 5
      PID      Type    Burst Time      Context Switches
      ___      ____    __________      ________________

      4        CPU        68                  3
      5        CPU        81                  4
      6        CPU        62                  3
      7        CPU        87                  4
      8        CPU        54                  4
$ _
```

pqueue.h

```c
#include "proc.h"  // Assuming this contains the definition for struct proc

#define MAX_QUEUE_SIZE 128  // Arbitrary size, adjust based on your need

struct pqueue {
    struct proc* heap[MAX_QUEUE_SIZE];
    int size;
};

// Helper function to swap two proc pointers
void swap(struct proc** a, struct proc** b) {
    struct proc* temp = *a;
    *a = *b;
    *b = temp;
}

// Function to heapify the queue (min-heap property)
void heapify(struct pqueue *pq, int i) {
    int smallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    // Compare burst times of parent, left child, and right child
    if (left < pq->size && pq->heap[left]->burst_time < pq->heap[smallest]->burst_time)
        smallest = left;
    if (right < pq->size && pq->heap[right]->burst_time < pq->heap[smallest]->burst_time)
        smallest = right;

    // If the smallest isn't the parent, swap and continue heapifying
    if (smallest != i) {
        swap(&pq->heap[i], &pq->heap[smallest]);
        heapify(pq, smallest);
    }
}
```

```c
struct proc* extractMin(struct pqueue *pq) {
    if (pq->size <= 0) {
        // printf(1, "Priority queue is empty\n");
        return 0;
    }

    if (pq->size == 1) {
        pq->size--;
        return pq->heap[0];
    }

    // Root holds the minimum process
    struct proc* min = pq->heap[0];

    // Move the last element to root and reduce the size
    pq->heap[0] = pq->heap[pq->size - 1];
    pq->size--;

    // Heapify the root to restore the heap property
    heapify(pq, 0);

    return min;
}

// Check if the priority queue is empty
int isEmpty(struct pqueue *pq) {
    return pq->size == 0;
}

// Check if the priority queue is full
int isFull(struct pqueue *pq) {
    return pq->size == MAX_QUEUE_SIZE;
}

// Initialize the priority queue
void initPQueue(struct pqueue *pq) {
    pq->size = 0;
}
```