CS1217 - Spring 2023 - Lab 2

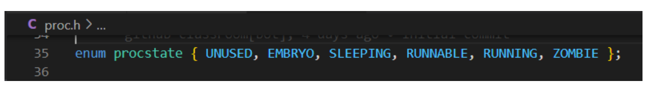
Gautam Ahuja, Nistha Singh

01. Exercise 01

The various states that an `xv6` process can be found in the `procstate enum` data type in the `proc.h` file. These are:

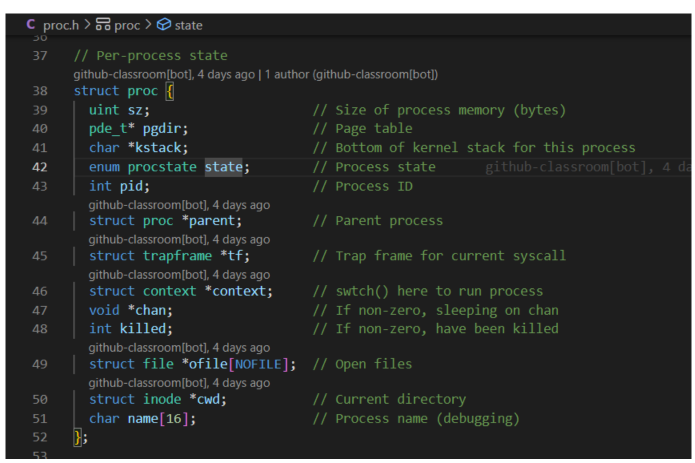
- **Unused**: This is the initial state of the process. It is not being used, executed or performing any tasks.
- **Embryo**: This is the state of the process when it is in formation. For example, when a `fork()` is called, the state in which the process is being created, its variables are being set, address state is being copies, etc is the `Embryo` state.
- **Sleeping**: It the waiting/blocking/sleeping state of a process. It occurs when the process is waiting on some other event to happen, or a process calls `sleep()`, etc.
- **Runnable**: This is equivalent to the ready state. The process in this state can start executing as soon as it gets hold of the CPU.
- **Running**:It the process which is currently executing instructions on the CPU.
- **Zombie**: This state is when a process is terminated (is finished executing). This can also be a process which has no parent process.

There are a total of **6 states** that an `xv6` process can be in. These states are used to manage the process by the operating system `xv6`.



```
proc.h > ...
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

The `state` variable of `enum procstate` type keeps the track of the current state of the process inside the `proc` data structure.



```
proc.h > proc > state
// Per-process state
github-classroom[bot], 4 days ago | 1 author (github-classroom[bot])
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
  github-classroom[bot], 4 days ago
  struct proc *parent;         // Parent process
  github-classroom[bot], 4 days ago
  struct trapframe *tf;        // Trap frame for current syscall
  github-classroom[bot], 4 days ago
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  github-classroom[bot], 4 days ago
  struct file *ofile[NOFILE];  // Open files
  github-classroom[bot], 4 days ago
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

02. Exercise 02

The scheduler code is:

```c
proc.c > scheduler(void)
322 void
323 scheduler(void)
324 {
325   struct proc *p;
326   struct cpu *c = mycpu();
327   c->proc = 0;
328
329   for(;;){
330     // Enable interrupts on this processor.      github-classroom[bot], last week • Initial commit
331     sti();      // STI - Set Interrupt Flag
332
333     // Loop over process table looking for process to run.
334     acquire(&ptable.lock);      // the &ptable.proc[NPROC] is the address of the last element of the ptable.proc array.
335     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
336       if(p->state != RUNNABLE)
337         continue;
338
339       // Switch to chosen process.  It is the process's job
340       // to release ptable.lock and then reacquire it
341       // before jumping back to us.
342       c->proc = p;
343       switchuvm(p);             // the switchuvm(p) will do the switching of the kernel virtual memory to the process's virtual memory.
344       p->state = RUNNING;       // Now the process is running, so we can switch to it.
345       // The swtch(&(c->scheduler), p->context)  is the content switch. It will save the current context and load the new context.
346       swtch(&(c->scheduler), p->context);
347       switchkvm();              // The switchkvm function will switch the process's virtual memory to the kernel virtual memory.
348
349       // Process is done running for now.
350       // It should have changed its p->state before coming back.
351       c->proc = 0;
352     }
353     release(&ptable.lock);
354
355   }
```

Line `329` has a `for` loop that looks like an infinite loop. This is done to make sure the `scheduler` always runs. As we know, the scheduler is an essential algorithm that makes sure the CPU is used at all the time.
Since the entire logic of scheduler is inside the loop, it makes sure that the kernel can keep track of all `RUNNABLE` processes.
Without the infinite loop, the scheduler would run only once and kernel will not be able to put any more processes on CPU after one exits.

The implemented scheduler policy is a **Round Robin Scheduling Algorithm**.

As discussed in class, a round-robin policy maintains a list of processes and iterates over to execute every runnable process.

In above algorithm, line **334** acquired the lock of the page-table (`ptable`) which holds the list of all the processes (`ptable.proc`). The maximum number of processes can only be 64 as defined by `NPROC` inside `param.h`.
After this, the condition at line **336** checks if the process is in runnable state, if not, increment the pointer `p` to point to next process on the list. After identifying the runnable process, the scheduler puts in on the CPU (`c->proc = p;`) on line **342** and changes the state from `RUNNABLE` to `RUNNING` on line **344**.
The kernel also switches to the processes virtual address space via `switchuvm(p);` on line **343** which loads the process's page table and then do a context switch via `swtch(&(c->scheduler), p->context);` on line **346**. At this point the new user process is running on the CPU until an event occurs (`exit()`, `yield()`, `wait()`, `syscall`, or timer interrupt). The definition of `swtch()` can

be found in the `swtch.S` file.

```
swtch.S
github-classroom[bot], last week | 1 author (github-classroom[bot])
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
# Save the current registers on the stack, creating
# a struct context, and save its address in *old.
# Switch stacks to new and pop previously-saved registers.
```

At this point (after completion) the scheduler switched the memory back from process' view to kernel-only global page table view (line **347** - `switchkvm();`). The CPU's current running process is set to `NULL` and the loop runs again to put the next `RUNNABLE` process onto the CPU.

In case the process is not completed in time, the time interrupt will occur which will be identified by `sti();`. After addressing the IRS the context will switch to kernel mode where the scheduler will run again from line **347**, the kernel address space, and continue to choose the next `RUNNABLE` process.

In such way the **Round Robin Scheduling policy** is implemented.

the definition of `switchuvm()` and `switchkvm()` is as follows:

```
void switchuvm(struct proc *)
Switch TSS and h/w page table to correspond to process p.
switchuvm(p);            // the switchuvm(p) will
```

```
void switchkvm(void)
Switch h/w page table register to the kernel-only page table,
for when no process is running.
switchkvm();             // The switchkvm function
```

03. Exercise 03

The definition of `fork()` is:

```c
proc.c > fork(void)

// Create a new process copying p as the parent.
// Sets up stack to return as if from system call.
// Caller must set state of returned proc to RUNNABLE.
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();
  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }
  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;
  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;
  for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);
  safestrcpy(np->name, curproc->name, sizeof(curproc->name));
  pid = np->pid;
  acquire(&ptable.lock);
  np->state = RUNNABLE;
  release(&ptable.lock);
  return pid;
}
```

The `fork()` is executed in an order of:

1. It first checks if there is any slots available in the `ptable` through the `alloproc()` function; which allocates and = returns the pointer to new `proc` of new function, if available, else returns 0; in which case the `fork()` return -1.

   ```
   static struct proc *allocproc(void)

   PAGEBREAK: 32
   Look in the process table for an UNUSED proc.
   If found, change state to EMBRYO and initialize
   state required to run in the kernel.
   Otherwise return 0.
   allocproc()) == 0){
   ```

2. It also copies over the parent process's memory space to the child process's memory space with `copyuvm(curproc->pgdir, curproc->sz)`. If the copying over was unsuccessful (in which case `copyuvm()` returns 0), the `fork()` return -1 and the state of the process is set as `UNUSED`.

   ```
   pde_t *copyuvm(pde_t *, uint)

   Given a parent process's page table, create a copy
   of it for a child.
   copyuvm(curproc->pgdir, curproc->sz)) == 0)
   ```

   When the process of copying of address space is executing, the child is in the state of `EMBRYO`

3. In normal scenario where the address space is copied successfully, the stack pointer for child is set, `pid` of parent is recorded and the process is put into a `RUNNABLE` state as the child is now ready to be executed. The `fork()` returns the `pid` of the child in parent process.

04. Exercise 04

As discussed in Question 1, a `ZOMBIE` process is the one which has no parent or in other words it is abandoned.
As discussed in class, the abandoned are reassigned to the grandfather (if available) or the `init` process itself. Doing a search for all occurrences of keyword `ZOMBIE` in `proc.c` we find out that the process `exit()` is used to reassign the abandoned processes to the `init` process.

```c
proc.c > ...
226   // until its parent calls wait() to find out it exited.
227   void
228   exit(void)
229   {
230     struct proc *curproc = myproc();
231     struct proc *p;
232     int fd;
233     if(curproc == initproc)
234       panic("init exiting");
235     // Close all open files.
236     for(fd = 0; fd < NOFILE; fd++){
237       if(curproc->ofile[fd]){
238         fileclose(curproc->ofile[fd]);
239         curproc->ofile[fd] = 0;
240       }
241     }
242     begin_op();
243     iput(curproc->cwd);
244     end_op();
245     curproc->cwd = 0;
246     acquire(&ptable.lock);
247     // Parent might be sleeping in wait().
248     wakeup1(curproc->parent);
249     // Pass abandoned children to init.
250     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
251       if(p->parent == curproc){
252         p->parent = initproc;
253         if(p->state == ZOMBIE)
254           wakeup1(initproc);
255       }
256     }
257     // Jump into the scheduler, never to return.
258     curproc->state = ZOMBIE;
259     sched();
260     panic("zombie exit");
261   }
```

Here we can see in the section of lines **249-256**, if a process is abandoned (`ZOMBIE`), it is being reassigned to the `init` function itself.

I. Part 01: MLFQ Scheduler

Final File: *proc_mlfq_v02.c*

As done in assignment 3, we know to add a new system call we need to modify the following files: `syscall.c`, `syscall.h`, `user.h`, `usys.S`, `sysproc.c`, `Makefile`.
We will also need to edit the following files: `proc.c`, `proc.h`, `param.h` as per the question.

To add a new system call, we first define it in `syscall.h` as follows

syscall.h > ...

```c
#define SYS_close   21
// Two new system calls - start
#define SYS_setpriority 22
#define SYS_getpriority 23
// Two new system calls - end
```

We also add `MAXPRIORITY` and `DEFAULT_BUDGET` in files `proc.h` as per the question:

proc.h > DEFAULT_BUDGET

```c
// Definig the maxpriority and max budget
#define MAXPRIORITY 2
#define DEFAULT_BUDGET 5
#define TICKS_TO_PROMOTE 20
```

We also add a new parameter of `priority` and `budget` in the `proc` structure in `proc.h` to hold the priority of current process.

proc.h > ...

```c
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  // Lab 2
  int priority;                // Process priority
  // int budget;                  // Process budget
};
```

Then in `syscall.c` we add a reference to each system call function and we also add it to the system call table which maps the functions to system call numbers defined in `syscall.h` syscall.c > argint(int, int *)

```c
extern int sys_uptime(void);
// Two new system calls - start
extern int sys_setpriority(void);
extern int sys_getpriority(void);
// Two new system calls - end
```

syscall.c > argint(int, int *)

```c
[SYS_close]    sys_close,
// Two new system calls - start
[SYS_setpriority] sys_setpriority,
[SYS_getpriority] sys_getpriority,
// Two new system calls - end
};
```

The declaration of functions (to which the syscall will call) is added in the file `user.h`

user.h > read(int, void *, int)

```c
int uptime(void);
// Two new system calls - Edit Start
int setpriority(int, int);
int getpriority(int);
// Two new system calls - Edit End
```

The declaration of these functions are also added in the assembly for user system calls in the file `usys.S`

usys.S

```asm
SYSCALL(uptime)
; Two new system calls - start
SYSCALL(setpriority)
SYSCALL(getpriority)
; Two new system calls - end
```

The definition of system call functions - `sys_getpriority()` and `sys_setpriority()` (which will call to the actual function - `getpriority()` and `setpriority()`) is implemented in `sysproc.c`

sysproc.c > sys_setpriority(void)

```c
// Two new system calls - start
int
sys_setpriority(void)          You, 5 minutes ago •
{
  int pid;
  int priority;

  if(argint(0, &pid) < 0)
    return -1;
  if(argint(1, &priority) < 0)
    return -1;
  return setpriority(pid, priority);
}

int
sys_getpriority(void)
{
  int pid;

  if(argint(0, &pid) < 0)
    return -1;
  return getpriority(pid);
}
// Two new system calls - end
```

Finally the functions `getpriority()` and `setpriority()` are definied in `proc.c` as follows:

The `setpriority()` takes the `pid` of the process and sets the `priority` which is passed as an argument. It first checks if the `priority` is valid and if the process with given `pid` exists, it assigns the `priority` and sets the budget to `DEFAULT_BUDGET`. In other cases it returns an error code.

Similarly the `getpriority()` takes the `pid` as an argument. If the process `pid` is valid and the process is not in `UNUSED` state, it returns the `pid`. Else it returns error code in other cases.

proc.c > setpriority(int, int)

```c
// New system calls
// int setpriority(int pid, int priority)
int
setpriority(int pid, int priority)
{
  struct proc *p;
  int returnCode = -1;     // returnCode 0 if success, -1 if fail

  // check if priority is valid
  if(priority < 0 || priority > MAXPRIORITY){
    cprintf("Invalid priority value detected");
    return returnCode;
  }

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->priority = priority;
      p->budget = DEFAULT_BUDGET;
      returnCode = 0;
      break;
    }
  }
  release(&ptable.lock);

  // check if pid is valid
  if(returnCode == -1){
    cprintf("Invalid pid value detected");
  }

  return returnCode;
}
```

proc.c > getpriority(int)

```c
// int getpriority(int pid)
int
getpriority(int pid)
{
  struct proc *p;
  int priority = -1;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid && p->state != UNUSED){
      priority = p->priority;
      break;
    }
    if(p->pid == pid && p->state == UNUSED){
      cprintf("Process with pid %d is in UNUSED state", pid);
      return priority;
    }
  }
  release(&ptable.lock);

  // check if pid is valid
  if(priority == -1){
    cprintf("Invalid pid value detected");
  }
}
// End of new system calls
```

At last we edit the `def.h` to include all the changes and add the function definitions.

defs.h

```c
// Two new system calls - start
int             setpriority(int, int);
int             getpriority(int);
// Two new system calls - end
```

As per the question, we add a new field to `proc` structure which holds the original priority in case the process goes to any other state from `RUNNABLE`. It will help us determine where to put the process once it is `RUNNABLE` again.

```c
// proc.h > proc
  // Lab 2
  int priority;            // Process priority
  int budget;              // Process budget
  int original_priority;   // Process original priority, keep track of if process state changes
};
```

We also edit the `fork()` in `proc.c` as upon allocation, each process will have the same initial (default) priority value, the highest priority. We set the process priority to `MAXPRRIORITY`.

```c
// proc.c > fork(void)
  np->state = RUNNABLE;
  // set the priority of the child process to highest priority
  np->priority = MAXPRIORITY;
```

Periodic Priority Adjustment:

1. We define a new field of `PromoteAtTime` in the `ptable` structure. Which will store the ticks value at which promotion will occur.

```c
// proc.c > ptable
struct {
  ...
  struct spinlock lock;
  ...
  struct proc proc[NPROC];
  // timer variables
  uint PromoteAtTime;

} ptable;
```

2. We set the page table value of `PromoteAtTime` in `userinit()` as follows:

```c
// proc.c > userinit(void)
  acquire(&ptable.lock);

  p->state = RUNNABLE;
  // set promotion time
  ptable.PromoteAtTime = ticks + TICKS_TO_PROMOTE;

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

3. We define `TICKS_TO_PROMOTE` as:

```c
// proc.h > TICKS_TO_PROMOTE
#define TICKS_TO_PROMOTE 20
```

4. Checking priorities each time `scheduler()` runs:
   Note: We did not use the `getpriority()` in the implementation of the scheduler. They themselves being a system call (which also calls `getpid()` system call) adds an overhead to the scheduler (and kernel in general) and thus slowing it down and sometimes not working at all.
   We define a new 2D array of `struct proc`. Each rows is the priority level (3rd being the highest) and each row can contain up to 64 (`NPROC`) processes each. The `mlfq_queue_size` keeps track of the amount of `non-ZOMBIE` process on each levels

```c
// proc_mlfq_v02.c > wakeup1(void *)
// Define three levels of priority for MLFQ Scheduler
struct proc *mlfq_queue[MAXPRIORITY+1][2*NPROC];
// top queue has highest priority
int mlfq_queue_size[MAXPRIORITY+1] = {-1,-1,-1};
```

To adjust the new process according to new priority we do as follows:

```c
// proc_mlfq_v02.c > scheduler(void)
/****       Periodic Priority Adjustment: Part 3       *****/
if(ticks >= ptable.PromoteAtTime){
  //cprintf("Promotion Time");
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != ZOMBIE && p->priority < MAXPRIORITY){
      // delete the process from the queue at that priority level
      for(int i = 0; i <= mlfq_queue_size[p->priority]; i++){
        if(mlfq_queue[p->priority][i] == p){
          for(int j = i; j < mlfq_queue_size[p->priority]; j++){
            mlfq_queue[p->priority][j] = mlfq_queue[p->priority][j+1];
          }
          mlfq_queue_size[p->priority]--;
          break;
        }
      }
      // p->priority++;
      setpriority(p->pid, p->priority + 1);
      p->budget = DEFAULT_BUDGET;
      // // send it to the end of the next priority queue
      mlfq_queue_size[p->priority]++;
      mlfq_queue[p->priority][mlfq_queue_size[p->priority]] = p;
    }
  }
  ptable.PromoteAtTime = ticks + TICKS_TO_PROMOTE;
}
```

We check if the `ticks` have been exceeded or not. Then going through all the processes, we skip the `ZOMBIE` processes. For every other process, we follow the logic of:
ţ We assume that max processes in top priority can be 64. Any more than that will not be possible.

- Pop the process out of the current priority level and shift all processes in that level left by one to fill the gap.
- Increase priority of this popped out process.
- Insert this new process at the end of new priority queue.

At the end, update the `PromoteAtTime` to `ticks + TICKS_TO_PROMOTE`.

We also edit the `fork()` so that every new child process created gets the `MAXPRIORITY` and `DEFAULT_BUDGET` allocated to it.

```c
// proc.c > fork(void)
// set the priority of the child process to highest priority
np->priority = MAXPRIORITY;
np->original_priority = MAXPRIORITY;
// set the budget of the child process to default budget
np->budget = DEFAULT_BUDGET;
```

MLFQ Scheduler:

As discussed, whenever a new process is created, it is assigned a budget of `DEFAULT_BUDGET` and a priority of `MAXPRIORITY`.

```c
// proc_mlfq_v02.c > fork(void)
np->state = RUNNABLE;
// set the priority of the child process to highest priority
np->priority = MAXPRIORITY;
np->original_priority = MAXPRIORITY;
// // set the budget of the child process to default budget
np->budget = DEFAULT_BUDGET;
// // New process created is inserted at the end (tail) of the queue
mlfq_queue_size[MAXPRIORITY]++;
mlfq_queue[MAXPRIORITY][mlfq_queue_size[MAXPRIORITY]] = np;
// cprintf("MAXPRIORITY queue size: %d\n",mlfq_queue_size[MAXPRIORITY]);
```

We also edit the `wakeup1()` function. Whenever a process is awaken, it is checked if it belongs to the priority queue as per the `p->priority` of the state. If not, we send it to the tail of the priority queue it belongs to.

```c
proc_mlfq_v02.c > wakeup1(void *)
static void
wakeup1(void *chan)
{
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == SLEEPING && p->chan == chan){
      p->state = RUNNABLE;
      // find if already in priority queue
      int check = 0;
      for(int i = 0; i < mlfq_queue_size[p->priority]; i++){
        if(mlfq_queue[p->priority][i] == p){
          check = 1;
          break;
        }
      }
      if(check == 0){
        mlfq_queue_size[p->priority]++;
        mlfq_queue[p->priority][mlfq_queue_size[p->priority]] = p;
      }      You, 16 hours ago • MLFQ v02 …
```

For the actual implementation of the MLFQ Scheduling algorithm, we have our 2D array with rows as priority levels and columns as the process entry in those priority queues.

We simply iterate through each level (startig with `MAXPRIORITY`) untill its all the processes in that level are finished (size of that queue becomes less than zero).

For every `RUNNABLE` process, we put it on the `cpu` and note down the time of its execution.

```c
proc_mlfq_v02.c > scheduler(void)
    // /****       MLFQ Scheduler        *****/
    for(int i = MAXPRIORITY; i >= 0; i--){
      if(mlfq_queue_size[i] < 0){
        continue;
      }
      while(mlfq_queue_size[i]>=0){
        int check = 0;
        for(int k=0; k <= mlfq_queue_size[i]; k++){
          if(mlfq_queue[i][k]->state != RUNNABLE && k==mlfq_queue_size[i]){
              break;
          }
          if(mlfq_queue[i][k]->state != RUNNABLE){
            continue;
          }
          p = mlfq_queue[i][k];
          // remove the process from the queue
          for(int temp = k; temp < mlfq_queue_size[i]; temp++){
            mlfq_queue[i][temp] = mlfq_queue[i][temp+1];
          }
          mlfq_queue_size[i]--;
          check = 1;
          // Note the start time of the process
          int start_time = ticks; //uptime();
          c->proc = p;
          switchuvm(p);
          p->state = RUNNING;
          swtch(&(c->scheduler), p->context);
          switchkvm();
          c->proc = 0;
          int end_time = ticks; //uptime();
          // Update the budget of the process
          p->budget = p->budget - (end_time - start_time);
```

We update the `budget` of each program as: `p->budget = p->budget - (end_time - start_time)`. Now as per given condition if budget < 0, we demote the process to lower priority level and insert it at the tail of the queue. Else we move it to the tail of current priority level.

proc_mflq_v02.c > scheduler(void)

```c
          p->budget = p->budget - (end_time - start_time);
          if(p->budget<=0){
            //process already removed from current queue
            // demote priority and place it at the tail of the new quque
            if(p->priority > 0){
              setpriority(p->pid, p->priority - 1);
              p->original_priority = p->priority;
              mlfq_queue_size[p->priority]++;
              mlfq_queue[p->priority][mlfq_queue_size[p->priority]] = p;
            }
            else{
              // if priority is 0 then send to the end of the queue
              mlfq_queue_size[p->priority]++;
              mlfq_queue[p->priority][mlfq_queue_size[p->priority]] = p;
            }
          }
          else{
            //if process id still RUNNABLE then put it at the end of the queue
            mlfq_queue_size[p->priority]++;
            mlfq_queue[p->priority][mlfq_queue_size[p->priority]] = p;
          }
        }
        if(check==0){
          break;
        }
      }
    }
    release(&ptable.lock);
  }
}
```

Along with periodic priority updating, the MLFQ Scheduling policy is implemented in `xv6` operating system.

II. Part 02: Lottery Scheduler

Final File: *proc_lottery_v02.c*

As per the question, we start off by implementing two new system calls by editing a bunch of files (same as in previous parts): `setticket()` and `getpinfo()`. The definition of these functions are as follows:

```c
int
settickets(int numTickets)
{
  struct proc *p = myproc();
  int returnCode = -1;     // returnCode 0 if success, -1 if fail

  // check if numTickets is valid
  if(numTickets < 0 || numTickets > MAXTICKET){
    cprintf("Invalid numTickets value detected");
    return returnCode;
  }

  acquire(&ptable.lock);
  p->tickets = numTickets;
  returnCode = 0;
  release(&ptable.lock);

  return returnCode;
}
```

```c
int
getpinfo(struct pstat *procstate){
    acquire(&ptable.lock);
    int returnCode = -1;    // returnCode 0 if success, -1 if fail
    if(procstate == 0){
        cprintf("Invalid procstate value detected");
        return returnCode;
    }
    int i = 0;
    struct proc *p;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != UNUSED){
            procstate->pid[i] = p->pid;
            procstate->tickets[i] = p->tickets;
            procstate->ticks[i] = p->ticks;
            procstate->inuse[i] = p->inuse;
            i++;
        }
    }
    returnCode = 0;
    release(&ptable.lock);
    return returnCode;
}
```

Then we also add new fields of `tickets`, `inuse` and `ticks` to the `proc` structure in file `proc.h`. This will also be helpful when populating the `pstat` structure.

```c
    // Lottery Scheduling
    int tickets;                // Process tickets
    int inuse;                  // Process in use
    int ticks;                  // Process ticks
};
```

The `pstat` structure is defined in the file `pstat.h` file.

```c
#ifndef _PSTAT_H_
#define _PSTAT_H_
#include "param.h"
struct pstat {
int inuse[NPROC]; // whether this slot of the process table is in use (1 or 0)
int tickets[NPROC]; // the number of tickets this process has
int pid[NPROC]; // the PID of each process
int ticks[NPROC]; // the number of ticks each process has accumulated
};
#endif // _PSTAT_H_
```

Algorithm to implement the Lottery Scheduling algorithm we would need a random number generator. We use the C-program for MT19937: Real number version provides us with `sgenrand()`, `genrand()` and `random_at_most()` functions. To do this we add the provided `rand.c` and `rand.h` file. We also add the `rand.o` to the `OBJS` section of `Makefile` to compile the random number generator.

```
rand.h > random_at_most(long)
You, 20 hours ago | 1 author (You)
void sgenrand(unsigned long);
long genrand(void);
long random_at_most(long);
```

```
Makefile
rand.o\
```

As per the question, we the tickets to one and ticks to zero for each process when it is initialize. To do this we edit the `allocproc()` as follows:

```
proc_lottery_v02.c > allocproc(void)
p->state = EMBRYO;
p->pid = nextpid++;
p->tickets = 1;        // By default, each process gets 1 ticket on creation.
p->ticks = 0;          // By default, each process gets 0 ticks on creation.
```

We also assign, for every child process should have same number of tickets as parent process. To do this we edit the `fork()` function as follows:

```
proc_lottery_v02.c > fork(void)
np->parent = curproc;
np->tickets = curproc->tickets; // Copy the number of tickets from the parent
```

We also initialize the seed which will be used for random number generator.

```
proc_lottery_v02.c > fork(void)
np->parent = curproc;
np->tickets = curproc->tickets; // Copy the number of tickets from the parent
```

Algorithm:

```
proc_lottery_v02.c > scheduler(void)
// calculate the total number of tickets
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state != RUNNABLE)
    continue;
  ticketSum += p->tickets;        // Count the number of tickets of the RUNNALE processes
}

// Generate a random number between 0 and the total number of tickets to determine which process to run in this lottery round
// Change the seed
// seed++;
sgenrand(seed++);
// cprintf("Random Number: %d at time: %d\n", random_at_most(10), uptime());

uint lotteryNumber = random_at_most(ticketSum);

for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state != RUNNABLE)
    continue;
  // We will use the count variable to keep track of the number of tickets we have counted so far
  // Then we will compare the count to the random number generated
  // If the count is less than the random number, we will continue to the next process
  // If the count is greater than the random number, we will run the current process
  count += p->tickets;
  // This is lottery as the process with the most tickets will have the highest chance of being chosen
  if(count < lotteryNumber){
    continue;
  }
  // 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();
  // if the process is not completed, update its tickets
  if(p->state == RUNNABLE && p->tickets < MAXTICKET){
    p->tickets = p->tickets + 1;
    // cprintf("INCREMENTED\n");
  }
  // Process is done running for now.
  // It should have changed its p->state before coming back.
  c->proc = 0;
  break;
```

- The algorithm is simple, since each process has an initialized ticket of one, we sum all the tickts of `RUNNABLE` processes. After getting the count, we generating a random number between 0 and the sum. This sum will be used to generate a range for the max random number.
- Then we iterate over all processes and maintain the count of tickets and as soon as the count exceed over the ticket of recent process, that process gets to run. As a result, the process which has larger number of tickets has the higher probability to run.
- If the process has exited, it has left the system. If the process is still in `RUNNABLE` state, it means it is compute intensive and thus we increase its ticket by one.
- We maintain a `MAXTICKET` count of 15.
- We edit the `pstat` structure each time the scheduler is run so we maintain the latest information.

C proc_lottery_v02.c > scheduler(void)

```c
    // fill the pstat with the processes
    int i = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      pstat.inuse[i] = (p->state == UNUSED) ? 0 : 1;
      pstat.tickets[i] = p->tickets;
      pstat.pid[i] = p->pid;
      pstat.ticks[i] = p->ticks;
    }
```

III. Part 03: Comparison and BenchMarking

Final Files: *tester-IO.c* and *tester-CPU.c*

We can compare the performance of the default Round Robin, MLFQ, and Lottery schedulers in xv6 by measuring the performance of each scheduler in terms of throughput and completion time.
We take two tester files using the system call `fork()`. One file will check the CPU and other will check the IO .

To do this we write the code and add the name in `Makefile` under `UPROGS` flag.

```
Makefile
_zombie\
_tester-CPU\
_tester-IO\
```

Test Case 1: CPU-Bound Processes
File: `tester-CPU.c`, the definition is:

```c
tester-CPU.c > main()
#include "types.h"   // header files for xv6 system calls
#include "user.h"
int cpu_load() {
    volatile int i, j, k;               // volatile to prevent compiler optimization
    for (i = 0; i < 1000; i++) {
        for (j = 0; j < 100000; j++) {
            // CPU performs loops and waste the time
            k = i + j;
        }
    }
    return k;
}
int main() {
    int pid1, pid2;
    int start = uptime();
    // fork the first child process and waste time
    pid1 = fork();
    if (pid1 == 0) {   // child process 1
        cpu_load();
        exit();
    }
    // fork the second child process and waste time
    pid2 = fork();
    if (pid2 == 0) {   // child process 2
        cpu_load();
        exit();
    }
    // wait for both child processes to finish before going back to parent process
    wait();
    wait();
    // print time elapsed
    int end = uptime();
    printf(1, "Time elapsed: %d\n", end - start);
    // exit the parent process
    exit();
}
```

Here, we will create CPU-bound processes that performs large number of calculations (addition) for each process. We create two processes and check the time taken by the program in each of the scheduler.

Test Case 2: I/O-Bound Processes

```c
tester-IO.c > main()
#include "types.h"   // header files for xv6 system calls
#include "user.h"
int main() {
    int i, j;
    int start = uptime();
    // create 15 child processes to put the IO load on the cpu
    for (i = 0; i < 15; i++) {
        if (fork() == 0) {   // child process
            for (j = 0; j < 100000; j++) {
                // just waste time
            }
            exit();
        }
    }
    // wait for all child processes to finish
    for (i = 0; i < 15; i++) {
        wait();
    }
    // Print time elapsed
    int end = uptime();
    printf(1, "Time elapsed: %d\n", end - start);
    // exit the parent process
    exit();
}
```

Here, we will create IO-bound processes that creates a large number of child processes (using `fork()`) for each process. We check the time taken by the program in each of the scheduler. When the child processes are created they write to memory which is an IO operation.

Results:
Overall, we can evaluate the effectiveness of the given schedulers in different scenarios. The time taken in each scenario is:

1. Default (Round Robin) Scheduler:

```
gautam-ahuja@LAPTOP-FV7627LB:~/.../cs1217-lab-2-julius-stabs-back-1$ make qemu-nox
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
Team Julius-Stabs-Back's Schedulers
init: starting sh
$ tester-CPU
Time elapsed: 169
$ tester-IO
Time elapsed: 4
$ 
```

   The default scheduler takes `169 ticks` for CPU and `4 ticks` for IO operation.

2. MLFQ Scheduler:

```
gautam-ahuja@LAPTOP-FV7627LB:~/.../cs1217-lab-2-julius-stabs-back-1$ make qemu-nox
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
Team Julius-Stabs-Back's Schedulers
init: starting sh
$ tester-CPU
Time elapsed: 161
$ tester-IO
Time elapsed: 6
```

The default scheduler takes `161 ticks` for CPU and `6 ticks` for IO operation.

3. Lottery Scheduler:

```
gautam-ahuja@LAPTOP-FV7627LB:~/.../cs1217-lab-2-julius-stabs-back-1$ make qemu-nox
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
Team Julius-Stabs-Back's Schedulers
init: starting sh
$ tester-CPU
Time elapsed: 632
$ tester-IO
Time elapsed: 3
```

The default scheduler takes `632 ticks` for CPU and `3 ticks` for IO operation.

From above results, we can see that for `CPU` intense operation the `MLFQ` scheduler is the best option and for `IO` intense operations, the `Lottery` scheduler is the best option.