Implementing Round Robin and Priority Scheduling in xv6

Tools used :

OS : GNU-Linux Ubuntu 20.04.04 and xv6
Kernel : Generic Kernel 5.13
Shell : Fish shell, Bourne shell (sh)
C Compiler : GNU C Compiler (gcc)
Text Editor : Vim and Atom
Terminal : Gnome-Terminal

Research resources from : Articles from MIT (mit.edu), IIT B, IIT D, xv6-Book, Youtube Lectures, GitHub, Medium, Book : Operating System Concepts by Abraham Silberschatz, Greg Gagne, Peter B. Galvin and Class lectures+notes.

Q.1 Implement Round Robin and Priority Scheduling technique both in the single scheduler of xv6. The default schedular of xv6 is round-robin. Try to implement by combinig both in one.

-> For Implementing this we need to create two system calls **ps** and **chpr** for displaying processes context at that time and for changing priority respectively.

Setting up the system calls :

-> Before that we need to add a priority attribute of integer type in the PCB definition located in the proc.h.

```c
// 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 priority;                // Process Priority added by me
};
```

-> Now in the allocproc function in proc.c file we will set the default priority as 10.

```c
found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->priority = 10; //default
```

-> Now set the child process default priority in exec.c file.

```c
curproc->pgdir = pgdir;
curproc->sz = sz;
curproc->tf->eip = elf.entry;  // main
curproc->tf->esp = sp;
curproc->priority = 3; // set by me as 3
```

STEPS :

1. In syscall.h add the following :-

   #define SYS_ps 22
   #define SYS_chpr 23

2. In syscall.c add the follwing :-

   extern int sys_ps(void);
   extern int sys_chpr(void);

[SYS_ps] sys_ps;
[SYS_chpr] sys_chpr;

3. In defs.h in the proc.c section add the following :-

int ps(void);
int chpr(int pid, int priority);

4. In user.h add the following :-

int ps(void);
int chpr(int pid, int priority);

5. In proc.c add the definition of the system calls :-

(i) ps

```
 int
 ps()
{
struct proc *p;
int d;
sti();    // some sort of kernerl-user switch gotta happen and it
ain't happen without interrupts
acquire(&ptable.lock);    // to avoid the race condition +
cprintf("NAME \t PID \t PPID \t STATE \t    \t PRIORITY\n");
// now u have to look in the proc table from the kernel space
for(p=ptable.proc; p<&ptable.proc[NPROC]; p++)
{
    if(p->state == SLEEPING)
    {
        d=p->parent->pid;
        if(p->pid == 1)
        {
           d=0;
        }
        cprintf("%s \t %d \t %d \t SLEEPING \t %d \n",p->name,
        p->pid,d,p->priority)
    }
    else if(p->state == RUNNING)
    {
        d=p->parent->pid;
```

```
                if(p->pid == 1)
                {
                   d=0;
                }
                cprintf("%s \t %d \t %d \t RUNNING \t %d \n",p->name,
                p->pid,d,p->priority);
            }
            else if(p->state == RUNNABLE)
            {
                d=p->parent->pid;
                if(p->pid == 1)
                {
                   d=0;
                }
                cprintf("%s \t %d \t %d \t RUNNABLE\t %d \n",p->name,
                p->pid,d,p->priority);
            }
        }
        release(&ptable.lock);
        return 22;
}
```

(ii) chpr

```
int
chpr(int pid, int priority)
{
        struct proc *p;
        acquire(&ptable.lock);
        for(p=ptable.proc; p<&ptable.proc[NPROC]; p++)
        {
            if(p->pid == pid)
            {
                p->priority = priority;
                break;
            }
        }
        release(&ptable.lock);
        return pid;
}
```

-> ps system call will simply look over the process table and will fetch the process attributes.

-> chpr will takes two arguments (pid and priority) and run loop over the process table to fetch the pid whose priority is going to be changed.

-> Here the priority variable in the function argument is from user space and the priority variable in the proc table is one of the attribute mentioned in the process context defined in the proc. h header file.

6. In sysproc.c add the following :-

```
int
sys_ps()
{
	return ps();
}

int
sys_chpr()
{
	int pid, pr; // pid and pr to accept priority value and pid from
	the user respectively
	if(argint(0,&pid)<0)
		return -1;
	if(argint(1,&pr)<0)
		return -1;

	return chpr(pid,pr);
}
```

7. In usys.S add the following

```
SYSCALL(ps)
SYSCALL(chpr)
```

8. Write the command ps.c and pri.c

-> ps.c

```c
#include"types.h"
#include"stat.h"
#include"user.h"
#include"fcntl.h"

int main()
{
  ps();
  exit();
}
```

-> pri.c

```c
#include"types.h"
#include"stat.h"
#include"user.h"
#include"fcntl.h"

int main(int argc, char *argv[])
{
    int pri, pid;
    pid = atoi(argv[1]);    // will convert the string in int from CLA
    pri = atoi(argv[2]); // will do the same
    if(pri<0 || pri>20)
    {
        printf(2,"NOT VALID priority. (0-20)!\n");
        exit();
    }
    else
    {
        chpr(pid, pri); // calling the system call to change the
        priority
    }
    printf(1,"Priority of process with PID %d changed.\n",pid);
    exit();
}
```

9. Make changes in the Makefile

UPROGS=

_ps
_chpr

=> <u>About scheduler() of xv6</u>

-> The default scheduler of xv6 is based on Round Robin algorithm.

-> It's scheduler runs on as a separate thread with its own stack.

-> The job of the scheduler() is to look through the list of process find a RUNNABLE process, set it's state to RUNNING and swtch to perfrom a context switch to the target's process kernel thread. swtch saves the current registers and loads the saved registers of the kernel thread in the hardware register, including the stack pointer and instruction ptr.

=> <u>Process creation mechanism in xv6</u>

-> After xv6 boot up, initcode.S runs and invoke first system call exec which basically starts init which is forked and then execs child with sh.

-> intit opens STDIN, STDOUT and STDERR file descriptors.

-> initcode.S is written in assembly language.

-> <u>Creating first user process</u>

  -> Firstly main() intializes several devices and subsystem.

  -> main() calls userinit() to create first process then it calls the scheduler() to start running processes.

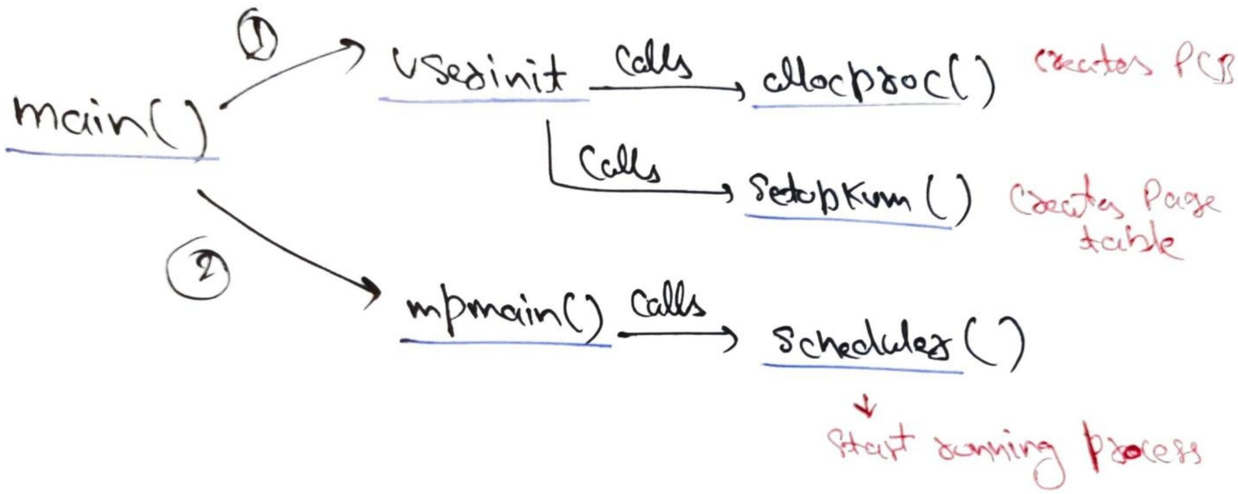
  -> userinit calls allocproc() to create the process PCB. It also calls setupkvm() which creates the process page table.

  -> allocproc is called for each processes, it also gives the unique PID to the processes. (returns 0 in faileure)

  -> The job of allocproc is to allocate a slot (a struct proc) in the process table and to initialize the parts of the process state required for its kernel thread to execute.

-> fork allocates new process via allocproc.

-> userinit is called only for the very first process.



=> <u>Making changes in the scheduler() of xv6 to enable priority scheduling along with round robin.</u>

-> Priority based Round-Robin CPU Scheduling algorithm is based on the integration of round-robin and priority scheduling algorithm. It retains the advantage of round robin in reducing starvation and also integrates the advantage or priority scheduling.

-> The idea is that a runnable high-priority process will be prefered by the scheduler over a runnable low-priority process.

-> In Multiplexing the context switching is so fast that it appears that each process has its own CPU. Basically multiplexing creates the illusion.

-> Create two two pointers *p1 and *high_p of struct proc type.

-> Look for the process in the process table whose priority we want to change, that process should be in RUNNABLE state.

-> Compare the priority of the selected process with the already running process.

```
if( high_p->priority > p1->priority )

then high_p = p1 ;
```

-> Now the current process pointer p will acquire the process with high priority. (p = high_p)

-.> CPU will be allocated to the selected pointer. (c->proc = p)

-> Now swtch() will do the context switching. swtch just saves and restores process context.

-> Each context is represented by a struct context* , a pointer to a structer on the kernel stack.

-> Now the process state will be changed to RUNNING.

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

  for(;;){
    sti(); // some sort of kernel-user switch gotta happen so u will have to enable this interrupt.

    struct proc *high_p; // this for getting the process whose priority I will change
    // Loop over process table looking for process to run.
    acquire(&ptable.lock); // to avoid the race condition
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      high_p = p;
      for(p1=ptable.proc; p1 < &ptable.proc[NPROC]; p1++ ){
          if(p1->state != RUNNABLE)
              continue;
          if(high_p->priority > p1->priority)
              high_p = p1;
      }
      p = high_p;
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&(c->scheduler), p->context);
      switchkvm();
      c->proc = 0;
    }
    release(&ptable.lock);
  }
}
```

=> Now to test the scheduler, I will use a dummy program lyceum.c which will be invoked twice at same time, it will consume some time by doing some unnecessary mathematical calculation, meanwhile ps command will be invoked to see the changes reflected in the processes PCB.

-> **lyceum.c**

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

int main(int argc, char *argv[]) {
   int t,n,x;

   if(argc < 2)
  n = 1;
   else
  n = atoi(argv[1]);
   if (n < 0 ||n > 20)
  n = 2;
   x = 0;
   for ( int i = 0; i < n; i++ ) {
      t = fork ();
      if ( t < 0 ) {
         printf(1, "Fork Failed!\n");
      } else if ( t > 0) {
                printf(1, "Parent with PID %d creating child whose PID
   is %d\n",getpid(),t);
            wait();
         }
         else{
  printf(1,"Child with PID %d is created.\n",getpid());
  for(int i = 0; i < 4000000000; i++)
                x = x + 3.14*89.64; // this loop will be used to consume
cpu time so that ps command can be used in that time and record the
processes details.
  break;
         }
   }
   exit();
}
```

=> Runnig the whole configuration

-> Initially the dummy program “lyceum” is called twice. (creates 2 child)

$ lyceum &; lyceum &;

-> Atfter that I changed the priority of one of the child process using pri command.

-> I am using ps command regularly to track the processes PCB.

OUTPUT :

```
$ ps
NAME : [redacted]
ROLL :

NAME      PID     PPID    STATE           PRIORITY
init      1       0       SLEEPING        3
sh        2       1       SLEEPING        3
ps        3       2       RUNNING         3
```

```
$ lyceum &; lyceum &;
Parent with PID 14 creating child whose PID is 15
Child with PID 15 is created.
Parent with PID 17 creating child whose PID is 18
Child with PID 18 is created.
$ ps
NAME : [redacted]
ROLL :

NAME      PID     PPID    STATE           PRIORITY
init      1       0       SLEEPING        3
sh        2       1       SLEEPING        3
ps        19      2       RUNNING         3
lyceum    6       1       SLEEPING        3
lyceum    8       1       SLEEPING        3
lyceum    9       6       RUNNABLE        10
lyceum    10      8       RUNNABLE        10
lyceum    14      1       SLEEPING        3
lyceum    15      14      RUNNING         10
lyceum    17      1       SLEEPING        3
lyceum    18      17      RUNNABLE        10
$
```

-> Currently one child of process (PID 14) is running i.e process (PID 15).

-> Now I will change the priority of process with PID 18 to 4. Currently it is in RUNNABLE state.

```
$ pri 18 4
Priority of process with PID 18 changed.
```

-> After this context switch will happen, process (PID 18) will be allocated CPU as its priority(PID 18) > priority(PID 15), process (PID 15) will go in RUNNABLE state.

```
$ ps
NAME : [redacted]
ROLL : [redacted]

NAME      PID    PPID   STATE       PRIORITY
init      1      0      SLEEPING    3
sh        2      1      SLEEPING    3
ps        21     2      RUNNING     3
lyceum    6      1      SLEEPING    3
lyceum    8      1      SLEEPING    3
lyceum    9      6      RUNNABLE    10
lyceum    10     8      RUNNABLE    10
lyceum    14     1      SLEEPING    3
lyceum    15     14     RUNNABLE    10
lyceum    17     1      SLEEPING    3
lyceum    18     17     RUNNING     4
$
```

so, yeah...Priority based Round Robin Scheduling is implemented...well :)

Thank you :)