xv6 -Implementing ps, nice system calls and priority scheduling

The ps (i.e., process status) command is used to provide information about the currently running processes, including their process identification numbers (PIDs). A process, also referred to as a task, is an executing (i.e., running) instance of a program.

The nice system call is used to change the priority of a given process.

The PCB of the process is stored in proc.h file.

In the struct proc in the proc.h file, add a new attribute 'priority' of int data type.

Step 1:

The cps is for the ps system call and chpr (change priority) is for the nice system call.

Open syscall.h, add the following two system calls:

```
#define SYS_cps     22
#define SYS_chpr   23
```

Step 2:

Next, in the PCB of the process, we have to add a new attribute 'priority'.

The PCB of the process is stored in proc.h file.

In the struct proc in the proc.h file, add a new attribute 'priority' of int data type.

Next, we have to include the declaration of these functions in defs.h and user.h files.

```
//Add this below the //proc.c section in defs.h
// proc.c
int cps(void);
int chpr(int pid, int priority);
```

```
//Add this below the system calls section in user.h
int cps(void);
int chpr(int pid, int priority);
```

Step 3:

Next, we have to include the definition of the cps and chpr functions in proc.c

//Add this in the end of the proc.c file

```
int
cps()
{
struct proc *p;
//Enables interrupts on this processor.
sti();

//Loop over process table looking for process with pid.
acquire(&ptable.lock);
cprintf("name \t pid \t state \t priority \n");
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state == SLEEPING)
            cprintf("%s \t %d \t SLEEPING \t %d \n ", p->name,p->pid,p->priority);
          else if(p->state == RUNNING)
            cprintf("%s \t %d \t RUNNING \t %d \n ", p->name,p->pid,p->priority);
          else if(p->state == RUNNABLE)
            cprintf("%s \t %d \t RUNNABLE \t %d \n ", p->name,p->pid,p->priority);
}
release(&ptable.lock);
return 22;
}

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;
}
```

Step 4:

Next, in sysproc.c, we have to define a function in which our cps and chpr functions will be called.

//Add this in the end of the sysproc.c file

```
int
sys_cps(void)
{
  return cps();
}

int
sys_chpr(void)
{
  int pid, pr;
  if(argint(0, &pid) < 0)
    return -1;
  if(argint(1, &pr) < 0)
    return -1;

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

Step 5:

Next, we have to make some minor changes in the usys.S file. The '.S' extension indicates that this file has assembly level code and this file interacts with the hardware of the system.

//Add this in the end of the usys.S file

```
SYSCALL(cps)
SYSCALL(chpr)
```

Step 6:

Next, we open the syscall.c file and add the two system calls.

```
//Add this where the other system calls are defined in syscall.c
extern int sys_cps(void);
extern int sys_chpr(void);

/*
  .
  .
  .
*/

//Add this inside static int (*syscalls[])(void)

[SYS_cps]     sys_cps,
[SYS_chpr]    sys_chpr,
```

Step 7:

Next, we have to create a ps.c and nice.c file in which our cps and chpr functions will be called respectively.

```
//     ps.c

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

int main(void){
  cps();
  exit();
}
```

```
// nice.c

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

int
main(int argc, char *argv[])
{
  int priority, pid;
  if(argc < 3){
    printf(2,"Usage: nice pid priority\n");
    exit();
  }
  pid = atoi(argv[1]);
  priority = atoi(argv[2]);
  if (priority < 0 || priority > 20){
    printf(2,"Invalid priority (0-20)!\n");
    exit();
  }
  chpr(pid, priority);
  exit();
}
```

Step 8:

Now that we have our system calls done, we have to work on the process priority assignment. For this, firstly we define the default priority of a process in the allocproc function in the proc.c file.

Here, I have assumed higher the number, lower is the priority of the process.

```
//Add this under the "found:" part of the allocproc function in proc.c

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->priority = 10;  //Default Priority of a process is set to be 10
```

Step 9:

Now, the child process is expected to have higher priority than the parent process. So we have to change the priority of child process when it is created. For this, we will make the changes in exec.c file.

/* Add this above the "bad:" part in the exec.c file where all other child process
attributes are mentioned */

curproc->priority = 2; //Giving child process default priority of 2

Step 10:

Now, we have to create a c program which creates a number of child process as mentioned by the user and consumes CPU time for testing our system calls and scheduling. So we create a new file dpro.c(dummy program)and write the following code:

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

int main(int argc, char *argv[]) {
  int pid;
  int k, n;
  int x, z;

  if(argc < 2)
          n = 1; //Default
  else
          n = atoi(argv[1]);
  if (n < 0 ||n > 20)
          n = 2;
  x = 0;
  pid = 0;

  for ( k = 0; k < n; k++ ) {
    pid = fork ();
    if ( pid < 0 ) {
      printf(1, "%d failed in fork!\n", getpid());
    } else if (pid > 0) {
      // parent
      printf(1, "Parent %d creating child %d\n",getpid(), pid);
      wait();
```

```
    }
    else{
        printf(1,"Child %d created\n",getpid());
        for(z = 0; z < 400; z+=1)
           x = x + 3.14*89.64; //Useless calculation to consume CPU Time
        break;
    }
 }
 exit();
}
```

Step 11:

Now, we make the appropriate changes in the Makefile. In Makefile, under 'UPROGS', add the following:

```
_ps\
_dpro\
_nice\
```

Also in the EXTRAS section of the Makefile, add nice.c, dpro.c and ps.c.

Priority based round robin scheduling walkthrough

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 of priority scheduling. Existing round robin CPU scheduling algorithm cannot be implemented in real time operating system due to their high context switch rates, large waiting time, large response time, and large turnaround time and less throughput. The proposed algorithm improves all the drawbacks of round robin scheduling algorithm.

Step 1:

For implementing this, we make the required changes in scheduler function in proc.c file.

//Replace the scheduler function with the one below for priority round robin scheduling

```
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 priority
   for(p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++){
```

```
            if(p1->state != RUNNABLE)
              continue;
            if(highP->priority > p1->priority)   //larger value, lower priority
              highP = p1;
      }
      p = highP;
      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);

  }
}
```

And we are done! We have implemented the system calls and changed the scheduling policy in xv6. Now, let us try it out.