Assignment Week#6

Implementing XV6 Priority Scheduling

1. **Adding Process Priority**

In this lab, we will walk you through the steps of adding a priority attribute to a process in xv6 and changing its value. We assign a process with a value between 0 and 20, the smaller the value, the higher the priority. The default value is 10.

i. Add *priority* to *struct proc* in *proc.h*

```
// Per-process state
struct proc {
  uint sz;                    // Size of process memory (bytes)
  ......
  char name[16];              // Process name (debugging)
  int priority;               // Process priority
};
```

ii. Assign default priority in **allocproc()** in *proc.c*

```
static struct proc*
allocproc(void)
{
  struct proc *p;
  char *sp;
  ........

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

iii. Modify **cps()** in *proc.c* discussed in the last lab to include the printout of the priority like the following :

```
$ ps
name     pid     state          priority
init     1       SLEEPING       10
 sh      2       SLEEPING       10
 ps      3       RUNNING        10
```

(Do the modification by yourself.)

iv. Write a dummy program named *foo.c* that creates some child processes and consumes some computing time:

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

int
main(int argc, char *argv[])
```

```
{
  int  k, n, id;
  double x = 0,  z;

  if(argc < 2 )
    n = 1;       //default value
  else
    n = atoi ( argv[1] ); //from command line
  if ( n < 0 || n > 20 )
    n = 2;
  x = 0;
  id = 0;
  for ( k = 0; k < n; k++ ) {
    id = fork ();
    if ( id < 0 ) {
      printf(1, "%d failed in fork!\n", getpid() );
    } else if ( id > 0 ) {  //parent
      printf(1, "Parent %d creating child  %d\n", getpid(), id );
      wait ();
   } else {   // child
      printf(1, "Child %d created\n",getpid() );
      for ( z = 0; z < 8000000.0; z += 0.01 )
         x =  x + 3.14 * 89.64;   // useless calculations to consume CPU time
      break;
    }
  }
  exit();
}
```

v. Add the function **chpr()** (meaning *change priority*) in *proc.c*

```
//change priority
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;
}
```

vi. Add **sys_chpr()** in *sysproc.c*

```
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 );
}
```

vii. Add **chpr()** as a system call to xv6 as discussed in the last lab.

Do this yourself.

viii. Create the user file *nice.c* with which calls **chpr**

```
#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();
}
```

Add *nice* to the system as discussed in the last lab.

ix. Test *nice* using *foo*

- Run *foo* to create a few processes, which run in the background and check them using *ps*

```
$ foo 4 &
$ ps
```

The commands will display something like:

```
name     pid     state          priority
init     1       SLEEPING       10
 sh      2       SLEEPING       10
 foo     6       RUNNABLE       10
foo      5       RUNNING        10
 foo     7       RUNNABLE       10
foo      8       RUNNABLE       10
foo      9       RUNNABLE       10
ps       11      RUNNING        10
```

- Change the priority of a process using *nice* and check the status using *ps* again:

```
$ nice 9 18
$ ps
name     pid     state          priority
init     1       SLEEPING       10
 sh      2       SLEEPING       10
 foo     6       RUNNABLE       10
foo      5       RUNNING        10
 foo     7       RUNNABLE       10
foo      8       RUNNABLE       10
foo      9       RUNNABLE       18
ps       13      RUNNING        10
```

Work to do

1. Do the experiment as described above. Copy-and-paste your outputs and commands to your report. Summarize all the steps, including those not presented explicitly above.

2. XV6 Process Priority Scheduling

In the previous section, we have learned how to change the priority of a process. In this section, we will implement a very simple priority scheduling policy. We simply choose a *runnable* process with the highest priority to run. (In practice, multilevel queues are often used to put processes into groups with similar priorities.) As we have done in the previous lab, we assume that a process has a value between 0 and 20, the smaller the value, the higher the priority. The default value is 10. The program *nice* that we implemented in the previous lab is used to change the priority of a process.

i. Give high priority to a newly loaded process by adding a *priority* statement in *exec.c*

```
int
exec(char *path, char **argv)
{
  char *s, *last;
  .....
  proc->tf->esp = sp;
  proc->priority = 2;    // Added statement
  switchuvm(proc);
  freevm(oldpgdir);
  .....
}
```

ii. Modify *foo.c* so that the parent waits for the children and adjust the loop for your convenience observations of

```
  int
main(int argc, char *argv[])
{
  .....
  for ( k = 0; k < n; k++ ) {
    id = fork ();
    if ( id < 0 ) {
      printf(1, "%d failed in fork!\n", getpid() );
    } else if ( id > 0 ) {  //parent
      printf(1, "Parent %d creating child  %d\n", getpid(), id );
      wait ();
   } else {   // child
      printf(1, "Child %d created\n",getpid() );
      for ( z = 0; z < 8000000.0; z += 0.01 )
         x =  x + 3.14 * 89.64;   // useless calculations to consume CPU time
      break;
    }
  }
  exit();
}
```

iii. Observe the default round-robin (RR) scheduling.
Round-robin (RR) is the default scheduling alogorithm used by xv6. You can see how this works by creating a few *foo* processes in the background and running *ps* a few times at random time intervals in xv6:

```
$ foo &; foo &; foo &

$ ps
name     pid     state           priority
init     1       SLEEPING        2
 sh      2       SLEEPING        2
 foo     9       RUNNING         10
 foo     8       SLEEPING        2
 foo     5       SLEEPING        2
 foo     7       SLEEPING        2
 foo     10      RUNNABLE        10
 foo     11      RUNNABLE        10
 ps      13      RUNNING         2

$ ps
name     pid     state           priority
init     1       SLEEPING        2
 sh      2       SLEEPING        2
 foo     9       RUNNABLE        10
 foo     8       SLEEPING        2
 foo     5       SLEEPING        2
 foo     7       SLEEPING        2
 foo     10      RUNNING         10
 foo     11      RUNNABLE        10
 ps      14      RUNNING         2

.....
```

You can see that the three *foo* child processes are run alternately while the parents are sleeping.

iv. Implement Priority Scheduling.

We can modify the **scheduler** function in *proc.c* to select the highest priority runnable process:

```
#define NULL 0

void
scheduler(void)
{
  struct proc *p;
  struct proc *p1;

  for(;;){
    sti();

    struct proc *highP = NULL;
    // Looking for runnable process
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      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;
      proc = p;
      switchuvm(p);
      .....
    }
    release(&ptable.lock);
  }
}
```

v. Observe the priority scheduling.

We run xv6 with the scheduler and again use *foo* and *ps* to see how it works. We use *nice* to change the priority of a process.

```
$ foo &; foo &; foo &

$ ps
name     pid     state            priority
init     1       SLEEPING         2
 sh      2       SLEEPING         2
 ps      13      RUNNING          2
 foo     10      SLEEPING         2
 foo     5       SLEEPING         2
 foo     7       RUNNING          10
 foo     8       SLEEPING         2
 foo     9       RUNNABLE         10
 foo     11      RUNNABLE         10

$ nice 11 8

$ ps
name     pid     state            priority
init     1       SLEEPING         2
 sh      2       SLEEPING         2
 ps      15      RUNNING          2
 foo     10      SLEEPING         2
 foo     5       SLEEPING         2
 foo     7       RUNNABLE         10
 foo     8       SLEEPING         2
 foo     9       RUNNABLE         10
 foo     11      RUNNING          8

$ ps
name     pid     state            priority
init     1       SLEEPING         2
```

```
sh      2       SLEEPING        2
ps      16      RUNNING         2
foo     10      SLEEPING        2
foo     5       SLEEPING        2
foo     7       RUNNABLE        10
foo     8       SLEEPING        2
foo     9       RUNNABLE        10
foo     11      RUNNING         8
.....
```

We can see that after we have changed the priority of process*11*to 8, which is higher than the priority 10 of processes *7* and *9*, process *11*is always selected to run.

Work to do

1. Do the experiment as described above. Copy-and-paste your outputs and commands to your report. Summarize all the steps, including those not presented explicitly above.

3. Report

Write a report that shows all your work; make sample screen shots of your graphics outputs if there is any, otherwise use script to capture your text outputs and copy-and-paste into your report. Include in your report the text source codes of your programs. Comment on and self-evaluate your work; state explicilty whether you have finished each part successfully! Discuss various issues such as difficulties you encountered or what you have learnt.