Paul Kummer

Dr. Hanku Lee

Operating Systems CSIS 430-01

11/11/2021

Assignment 07: Priority Scheduling

Purpose

This assignment's goal was to modify XV6 to have a priority scheduling algorithm instead of the round robin algorithm. To prove that priority scheduling is happening, three previously created user commands of myfork, ps, and cpr are used to test the results.

Code

Three commands are reused to test the scheduling algorithm. The myfork command will create processes that will waste cpu time and remaining running for a long duration. The ps command will list all the processes and their status, which contains their priority level. Lastly, the cpr command will change a processes priority level. The implementation of the priority algorithm was within the proc.c file and modified the scheduler function.

proc.c:

```
74  static struct proc*
75  allocproc(void)
76  {
77    struct proc *p;
78    char *sp;
79
80    acquire(&ptable.lock);
81
82    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
83      if(p->state == UNUSED)
84        goto found;
85
86    release(&ptable.lock);
87    return 0;
88
89  found:
90    p->state = EMBRYO;
91    p->pid = nextpid++;
92
93    //default initial process priority value
94    p->priority = 10;
95
96    release(&ptable.lock);
97
98    // Allocate kernel stack.
99    if((p->kstack = kalloc()) == 0){
100     p->state = UNUSED;
101     return 0;
102   }
103   sp = p->kstack + KSTACKSIZE;
104
105   // Leave room for trap frame.
106   sp -= sizeof *p->tf;
107   p->tf = (struct trapframe*)sp;
108
109   // Set up new context to start executing at forkret,
110   // which returns to trapret.
111   sp -= 4;
112   *(uint*)sp = (uint)trapret;
113
114   sp -= sizeof *p->context;
115   p->context = (struct context*)sp;
116   memset(p->context, 0, sizeof *p->context);
117   p->context->eip = (uint)forkret;
118
119   return p;
120 }
```

From the previous assignment, when a new process is created it has its default priority level to be 10. Since all processes will be the same priority if nothing is changed, the last runnable process will run.

proc.c: scheduler

```
378  void scheduler(void)
379  {
380    struct proc *firstLoopP;
381    struct proc *chosenP;
382    struct cpu *c = mycpu();
383    c->proc = 0;
384    //higestPriority is used because chosenP may not be initialized to check priority
385    int highestPriority = 0;
386
387    for(;;)
388    {
389      // Enable interrupts on this processor.
390      sti();
391
392      // Loop over process table looking for process to run.
393      acquire(&ptable.lock);
394      for(firstLoopP = ptable.proc; firstLoopP < &ptable.proc[NPROC]; firstLoopP++)
395      {
396        //skip over any process that isn't runnable and select the highest priority
397        if(firstLoopP->state == RUNNABLE && firstLoopP->priority >= highestPriority)
398        {
399          chosenP = firstLoopP;
400          highestPriority = firstLoopP->priority;
401        }
402      }
403
404      // Make sure that process actually exists to switch to.
405      // All processes start at 10, 0 means no processes
406      if(highestPriority > 0)
407      {
408        // Switch to chosen process.  It is the process's job
409        // to release ptable.lock and then reacquire it
410        // before jumping back to us.
411        c->proc = chosenP;
412        switchuvm(chosenP);
413        chosenP->state = RUNNING;
414
415        swtch(&(c->scheduler), chosenP->context);
416        switchkvm();
417
418        // Process is done running for now.
419        // It should have changed its p->state before coming back.
420        c->proc = 0;
421        highestPriority = 0;
422      }
423
424      release(&ptable.lock);
425    }
426  }
```

The scheduler will now loop over all processes and update the variable highestPriority with whatever the priority level of the process with the highest priority. When a process is found to be the current highest priority, it will be switched to a running state and start to run. After the process is done running, the variable highestPriority is set to -1 so that another process can claim the highest priority.

myfork.c

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

int main(int argc, char* argv[])
{
    int n;
    int workLoad = (1000 * 1000 * 1000);

    int k; //loop counter
    int id = 0; //process id
    double x = 0;
    double z = 0;
    double d = 1;

    if(argc <2)
    {
        n = 1;
    }
    else
    {
        n = atoi(argv[1]);
    }

    if(n < 0 || n > 20)
    {
        n = 2;
    }



    for(k = 0; k < n; k++)
    {
        id = fork();

        //sti();

        if(id < 0)
        {
            printf(1,"%d failed in fork! \n", getpid());
        }

        else if(id == 0)
        {
            printf(1, "Child %d created \n", id);

            for(z = 0; z < workLoad; z += d)
            {
                x = x + 3.14 * 200.19;
            }

            break;
        }

        else
        {
            printf(1, "Parent %d creating child %d\n", getpid(), id);

            wait();
        }
    }

    //myfork(n, workLoad);

    exit();
}
```

This command will call multiple fork commands and, in each child process, a pointless mathematical operation will occur, using many cpu cycles. This pointless operation will allow time for the process' priority to be changed so that the scheduling algorithm can be observed to be working.

Execution

```
$ myfork 10 &;
$ Parent 11 creating child 12
Child 0 created
ps
name     pid     state           priority
init     1       SLEEPING        10
sh       2       SLEEPING        10
myfork   6       RUNNABLE        10
myfork   12      RUNNING         10
myfork   5       SLEEPING        10
ps       13      RUNNING         10
myfork   11      SLEEPING        10
$ cpr 6 50;
 pid=6, pr=50
$ ps
name     pid     state           priority
init     1       SLEEPING        10
sh       2       SLEEPING        10
myfork   6       RUNNING         50
myfork   12      RUNNABLE        10
myfork   5       SLEEPING        10
ps       16      RUNNING         10
myfork   11      SLEEPING        10
$ cpr 12 50;
 pid=12, pr=50
$ cpr 6 10;
 pid=6, pr=10
$ ps
name     pid     state           priority
init     1       SLEEPING        10
sh       2       SLEEPING        10
myfork   6       RUNNABLE        10
myfork   12      RUNNING         50
myfork   5       SLEEPING        10
ps       21      RUNNING         10
myfork   11      SLEEPING        10
$
```

With the use of cpr, ps, and myfork the effects of changing a processes priority can be observed. In this example, all processes start with the same priority. Then process 6, myfork, has its priority changed to 50. Once the priority level was changed to higher than the other processes, it began to run. Next, process 12, myfork, is set to a priority of 50 and process 6 is brought back down to 10. Then, process 12 began to run and 6 went back to runnable.

Conclusion

In this assignment I learned how scheduling processes works in XV6. The implementation of the algorithm was easy, but much more advanced algorithms can be built. They could allow a process to use a quantum and have its level decay as it utilizes the cpu. It was fun to see real affects within the OS because of the changes. I also learned about interrupts in the process of figuring out why I couldn't enter the ps command after running myfork. I determined that I needed to enter the ampersand symbol after the myfork, which would allow the sh to interrupt the current process to issue a command.