Modifications to the xv6 operating system

By Shradha Sehgal (2018101071)

Overview

Various improvements have been made to the xv6 operating system such as the waitx and getpinfo syscall. Scheduling techniques such as FCFS, PBS, and MLFQ have also been implemented.

Run the shell

1. Run the command `make qemu`.
2. Add the flag `SCHEDULER` to choose between RR, FCFS, PBS, and MLFQ

Task 1

Waitx Syscall

Adding System Call

The files that have been modified are:

1. user.h
2. usys.S
3. syscall.h
4. syscall.c
5. sysproc.c
6. defs.h
7. proc.c
8. proc.h

- The fields ctime (CREATION TIME), etime (END TIME), rtime (calculates RUN TIME) & iotime (IO TIME) fields have been added to proc structure of proc.h file

- proc.c contains the actual waitx() system call:

  Code is same as wait() and does the following:

  - Search for a zombie child of parent in the proc table.
  - When the child was found , following pointers were updated :
    - *wtime= p->etime - p->ctime - p->rtime - p->iotime;
    - *rtime=p->rtime;
  - sysproc.c is just used to call waitx() which is present in proc.c. The sys_waitx() function in sysproc.c passes the parameters (rtime,wtime) to the waitx() of proc.c, just as other system calls do.

Calculating ctime, etime and rtime

- ctime is recoreded in allocproc() function of proc.c (When process is born). It is set to ticks.
- etime is recorded in exit() function (i.e when child exists, ticks are recorded) of proc.c.
- rtime is updated in trap() function of trap.c. IF STATE IS RUNNING , THEN UPDATE rtime.
- iotime is updated in trap() function of trap.c.(IF STATE IS SLEEPING , THEN UPDATE iotime.

Tester file - time command

- time inputs a command and exec it normally
- Uses waitx instead of normal wait
- Displays wtime and rtime along with status that is the same as that returned by wait() syscall

Getpinfo Sys Call

A separate header file pstat.h has been created for struct proc_stat.

Along with this the files modified are:

1. user.h
2. usys.S
3. syscall.h
4. syscall.c

5. sysproc.c
6. defs.h
7. proc.c

- `getpinfo(struct proc_stat*, int pid)` has been declared in proc.c and `sys_getpinfo(void)` in sysproc.c just passes arguments to it.
- pid is assigned according to the parameter passed.
- For runtime we traverse through the process table to find a process with the same pid and assign proc_stat's run time as that process' `rtime` .
- `num_run` is incremented in al scheduler whenever a process is run.
- Current_queue is assigned only in case of MLFQ.
- `p->ticks[5]` is copied into the proc_stat objects ticks[5] in case of MLFQ

The code where all this assigning takes place is as follows:

```
int getpinfo(struct proc_stat *p_proc, int pid)
{
    struct proc *p;
    int ret = -1;
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
        if(p -> pid == pid)
        {
            p_proc -> pid = pid;
            p_proc->runtime = p->rtime;
            p_proc->num_run = p->num_run;
            #ifdef MLFQ
            p_proc -> current_queue = p->queue;
            for(int i=0; i < 5;i++)
                p_proc->ticks[i] = p->ticks[i];
            #endif
            ret = 1;
            //cprintf("")
            break;
        }
    }

    return ret;
}
```

The code for sys_getpinfo() is as follows:

```
int sys_getpinfo(void)
{
  int pid; struct proc_stat *p;

  if(argptr(0,(char**)&p,sizeof(p)) < 0)
    return -1;

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

  return getpinfo(p, pid);
}
```

Task 2 - Scheduling techniques

All scheduling techiniques have been added to the `scheduler` function in `proc.c` . Add the flag `SCHEDULER` to choose between RR, FCFS, PBS, and MLFQ. This has been implemented in `Makefile` .

First come - First Served (FCFS)

- Non preemptive policy so we removed the the `yield()` call in trap.c in case of FCFS as shown:

```
#ifndef FCFS
    yield();
#endif
```

- Iterate through the process table to find the process with min creation time as follows:

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

    if (p->state != RUNNABLE)
        continue;

    //if(p->pid > 1)
    {
        if (min_proc == 0)
            min_proc = p;
```

```
        else if (p->ctime < min_proc->ctime)
            min_proc = p;
    }
}
```

- Check if the process found is runnable, if it is, execute it

```
if (min_proc != 0 && min_proc->state == RUNNABLE)
{
    cprintf("Process with PID %d and start time %d running\n", min_proc->pid,
    p = min_proc;

    c->proc = p;
    switchuvm(p);
    p->num_run++;
    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;
}
```

Priority Based Scheduler

- Assign default priority 60 to each entering process
- Find the minimum priority process by iterating through the process table (min priority number translates to maximum preference):

```
struct proc *p;
struct proc *min_pr_proc = 0;

for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
{

    if (p->state != RUNNABLE)
        continue;
```

```
    if (min_pr_proc == 0)
        min_pr_proc = p;

    else if (p-> priority < min_pr_proc-> priority)
        min_pr_proc = p;

}
```

- To implement RR for same priority processes, iterate through all the processes again. Whichever has same priority as the min priority found, execute that. yield() is enabled for PBS in proc.c() so the process gets yielded out and if any other process with same priority is there, it gets executed next.
- We also have a check within the 2nd loop to ensure no other process with lower priority has come in. If it has, we break out of the 2nd loop, otherwise RR is executed for same priority processes.
- set_priority() calls yield() when the priority of a process becomes lower than its old priority.

```
int set_priority(int pid, int priority)
{
    struct proc *p;
    int to_yield = 0, old_priority = 0;

    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
        if(p->pid == pid)
        {
            to_yield = 0;
            acquire(&ptable.lock);
            old_priority = p->priority;
            p->priority = priority;
            cprintf("Changed priority of process %d from %d to %d\n", p->pid,
            if (old_priority > p->priority)
                to_yield = 1;
            release(&ptable.lock);
            break;
        }
    }

    if (to_yield == 1)
        yield();

    return old_priority;
```

```
}
```

MLFQ

- We declared 5 queues with different priorities based on time slices, i.e. 1, 2, 4, 8, 16 timer ticks, as shown:

```
struct proc *queue[5][NPROC];
```

- These queues contain runnable processes only.
- The add process to queue and remove process from queue functions take arguments of the process and queue number and make appropriate changes in the array(pop and push).

```
int add_proc_to_q(struct proc *p, int q_no);
int remove_proc_from_q(struct proc *p, int q_no);
```

- We add a process in a queue in userinit() and fork() and kill() functions in proc.c i.e. wherever the process state becomes runnable.
- Ageing is implemented by iterating through queues 1-4 and checking if any process has exceeded the age limit and subsequently moving it up in the queues.

```
for(int i=1; i < 5; i++)
{
    for(int j=0; j <= q_tail[i]; j++)
    {
        struct proc *p = queue[i][j];
        int age = ticks - p->enter;
        if(age > 30)
        {
            remove_proc_from_q(p, i);
            cprintf("Process %d moved up to queue %d due to age time %d\n", p-
            add_proc_to_q(p, i-1);
        }

    }
}
```

- Next, we iterate over all the queues in order, increase the tick associated with that

process and its number of runs.

- In the trap file, we check if the `curr_ticks` of the process >= permissible ticks of the queue. If that's the case, we call yield and push the process to the next queue. Otherwise increment the ticks and let the process remain in queue.

The trap function is as follows:

```
if (myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0 + IRQ_TIMEF
{
    #ifdef MLFQ
        if(myproc()->curr_ticks >= q_ticks_max[myproc()->queue])
        {
            change_q_flag(myproc());
            cprintf("Process with PID %d on Queue %d yielded out as ticks comp
            yield();
        }

        else
        {
            incr_curr_ticks(myproc());
            cprintf("Process with PID %d continuing on Queue %d with current t
        }

    #else
    #ifndef FCFS
        // cprintf("ysfn");
        yield();

    #endif
    #endif
}
```

- Release table lock once the process is over

Tester files

`time.c` , `pinfo_test.c` and `test.c` are the tester files.