Week7 Report in Class (Fri56)

Q1

> 阅读lab7代码，详细描述 `user\rr.c` 中相关的进程是如何进行调度的。描述中需要包含各进程的执行顺序，何时进入被调度的队列，何时被切换，执行结束时发生了什么（重复的内容只需描述一遍）

```c
#include <ulib.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define TOTAL 5
/* to get enough accuracy, MAX_TIME (the running time of each process) should >1000 mseconds. */
#define MAX_TIME  10000
unsigned int acc[TOTAL];
int status[TOTAL];
int pids[TOTAL];

static void
spin_delay(void)
{
     int i;
     volatile int j;
     for (i = 0; i != 200; ++ i)
     {
          j = !j;
     }
}

int
main(void) {
     int i,time;
     memset(pids, 0, sizeof(pids));

     for (i = 0; i < TOTAL; i ++) {
          acc[i]=0;
          if ((pids[i] = fork()) == 0) {
               acc[i] = 0;
               while (1) {
                    spin_delay();
                    ++ acc[i];
                    if(acc[i]%4000==0) {
```

```
37                          if((time=gettime_msec())>MAX_TIME) {
38                              cprintf("child pid %d, acc %d, time
    %d\n",getpid(),acc[i],time);
39                              exit(acc[i]);
40                          }
41                      }
42                  }
43              }
44              if (pids[i] < 0) {
45                   goto failed;
46              }
47         }
48
49         cprintf("main: fork ok,now need to wait pids.\n");
50
51         for (i = 0; i < TOTAL; i ++) {
52             status[i]=0;
53             waitpid(pids[i],&status[i]);
54             //cprintf("main: pid %d, acc %d, time
    %d\n",pids[i],status[i],gettime_msec());
55         }
56         cprintf("main: wait pids over\n");
57         return 0;
58
59    failed:
60         for (i = 0; i < TOTAL; i ++) {
61              if (pids[i] > 0) {
62                   kill(pids[i]);
63              }
64         }
65         panic("FAIL: T.T\n");
66    }
67
```

1. user_main **(PID=2)** executes the program `main` and `fork()` 5 times. It creates child processes 3, 4, 5, 6, 7. These child processes will be added to the ready queue.

```
1   for (i = 0; i < TOTAL; i ++) {
2               acc[i]=0;
3               if ((pids[i] = fork()) == 0) {
4                   ...
5               }
```

2. user_main **(PID=2)** will than go to the rest of the program. It waits for reclaim child processes in order `waitpid(pid)` 3, 4, 5, 6, 7

```
 cprintf("main: fork ok,now need to wait pids.\n");

      for (i = 0; i < TOTAL; i ++) {
          status[i]=0;
          waitpid(pids[i],&status[i]);
          //cprintf("main: pid %d, acc %d, time
%d\n",pids[i],status[i],gettime_msec());
      }
      cprintf("main: wait pids over\n");
```

3. user_main now is waiting for child process `PID=3`. `schedule()` function will executes process (PID=3)
4. Child process (PID=3) is now in execution. We set the timestamp for its execution. It performs `spin_delay()` to consume some times. After reaching the time limitation, PID=3 will cause a trap and `schedule()` will activate next process, inside the ready queue, the mechanism is RR, so it will wake up PID=4. PID=3 is re-enqueued to its ready queue.

```
while (1) {
                    spin_delay();
                    ++ acc[i];
                    if(acc[i]%4000==0) {
                        if((time=gettime_msec())>MAX_TIME) {
                            cprintf("child pid %d, acc %d, time
%d\n",getpid(),acc[i],time);
                            exit(acc[i]);
                        }
                    }
                }
```

5. Child process (PID=4) is now in execution. This is the same process as PID=3. After causing trap, PID=5 will start execution.
6. PID5 -> PID6
7. PID6 -> PID7
8. PID7 -> PID3
9. RR still works so this loop will continue
10. Until once PID=3 reaches its `MAX_TIME`. It attempts to stop its execution by calling `exit()`. This will call `do_exit()` to notify user_main (PID=2). Then PID3 will temporarily become zombie process. Until user_main reclaim it. user_main will be re-enqueued to the ready queue. Now the ready queue is `4, 5, 6, 7, 2`. Then `schedule()` function works by calling PID=4 in execution. Still PID3 -> PID4.

```
if((time=gettime_msec())>MAX_TIME) {
                            cprintf("child pid %d, acc %d, time
%d\n",getpid(),acc[i],time);
                            exit(acc[i]);
```

11. PID4 -> PID5, PID5 -> PID6, PID6 -> PID7. Notice that they all reach their `MAX_TIME` and all become the zombie processes.
12. Then, after PID7 is finished, user_main will be in execution. It reclaims all the child processes in order. (3, 4, 5, 6, 7).
13. After that, user_main finishes its execution and return.

In qemu, the print result will be shown as:

```
The next proc is pid:1 // init process
The next proc is pid:2 // user_main process
kernel_execve: pid = 2, name = "rr".
Breakpoint
main: fork ok,now need to wait pids.
//////// Round Robin ////////
The next proc is pid:3
The next proc is pid:4
The next proc is pid:5
The next proc is pid:6
The next proc is pid:7
The next proc is pid:3
The next proc is pid:4
The next proc is pid:5
The next proc is pid:6
The next proc is pid:7
The next proc is pid:3
The next proc is pid:4
The next proc is pid:5
The next proc is pid:6
The next proc is pid:7
...
//////// Round Robin ////////

The next proc is pid:3 // child process 3 reaches its TIME_MAX
child pid 3, acc 3664000, time 10010
The next proc is pid:4 // child process 4 reaches its TIME_MAX
child pid 4, acc 3640000, time 10010
The next proc is pid:5 // child process 5 reaches its TIME_MAX
child pid 5, acc 3636000, time 10010
The next proc is pid:6 // child process 6 reaches its TIME_MAX
child pid 6, acc 3612000, time 10010
The next proc is pid:7 // child process 7 reaches its TIME_MAX
```

```
child pid 7, acc 3656000, time 10010
The next proc is pid:2 // user_main reclaims all the child process in sequence
main: wait pids over
The next proc is pid:1
all user-mode processes have quit.
The end of init_main
```