CS460 Takehome Midterm Exam

DUE: 2-27-2020
DEMO to TA
Turn in a hardcopy of your code

1. READ Process Management PDF

YOUR LAB4 supports the commands
|switch kfork exit sleep wakeup|
operations of processes. Problem 1 uses YOUR LAB4 as the base code

Processes begins from P0, which kfork a P1, switch to run P1, which may kfork other procs. The processes form a family tree, which looks like this:

```
               P0
               |
               P1   (never let P1 exit)
               |
     ------------------------
        P2     P3    P4
               |
            --------
            P5   P6
```

(1). Implement process family tree as a BINARY tree by using the child, sibling, parent pointers. When a process runs, show its pid, ppid and child list.

(2). Modify kexit(int exitCode) as in Section 2 of the PDF file

(3). Implement pid = kwait(int *status) as in Section 4 of the PDF file

For testing: ADD a wait command for ANY process to wait for a ZOMBIE child.

2. Timer Service

With a hardware timer, e.g. timer0, the OS kernel can provide each process with a virtual timer. A process may request an interval timer of t seconds by the command 't', which asks for a time value in t seconds. The 't' command causes the process to sleep for t seconds. When its interval time expires, the timer interrupt handler wakes up the process, allowing it to continue.

The timer requests of processes are maintained in a timer queue containing Time Queue Elements (TQEs), which looks like the following:

```
     tq ->  tqe ->     tqe ->    tqe -> NULL
           ------    ------    ------
           | 5  |    |  8 |    | 17 |   (time in seconds to expire)
           | *2 |    | *1 |    | *3 |   (*pid means &proc[pid])
           ------    ------    ------
```

At each second, the timer interrupt handler decrements the time field of each TQE by 1. When a TQE's time decrements to 0, the interrupt handler deletes its TQE from tq and wakes up the process.

For example, after 5 seconds, it deletes the tqe of PROC2 and wakes up process P2.

In the above timer queue, the time field of each TQE contains the exact time remaining. The disadvantage of this scheme is that the interrupt handler must decrement the time field of each and every TQE. In general, an interrupt handler should complete an interrupt processing as quickly as possible. This is especially important for the timer interrupt handler. Otherwise, it may loss ticks or even never finish. In contrast, when a process enters a timer request, it also manipulates the timer queue but the process does not have the same kind of critical time constraints. We can speed up the timer interrupt handler by modifying the timer queue as follows.

```
     tq ->  tqe ->    tqe ->    tqe -> NULL
           ------    ------    ------
           | 5  |    |  3 |    |  9 |   (relative time)
           | *2 |    | *1 |    | *3 |   (pointer to proc[pid])
           ------    ------    ------
```

In the modified timer queue, the time field of each TQE is relative to the cummulative time of all the preceeding TQEs. At each second, the timer interrupt handler only needs to decrement the time of the first TQE and process any TQE whose time has expired. With this setup, insertion/deletion of a TQE must be done carefully.

REQUIREMENT:

Implement a timer queue to support interval timer requests of processes.
Add a 't' command, which
ask for a time value t in seconds, e.g. 20;
enter a TQE into the timer queue;
process goes to sleep (e.g. on its TQE)

Modify the timer interrupt handler to:
display a wall clock;
dispaly the current timer queue every second;
handle TQEs and wake up any process whose time has expired;

During demo, let P1 kfork() serveral processes, e.g. P2, P3.
P1: 't' : 30
P2: 't' : 20
P3: 't;:   5

3. Program C6.1 of the textbook uses one-level paging, each page size is 1MB.
It assumed 256MB RAM followed by 2MB I/O space at 256MB-258MB.

REQUIREMENT
Modify the page table to have only 128MB RAM and the 2MB I/O pages.

For testing: int *p;

```
              printf("test MM at VA=2MB\n");
              p = (int *)(2*0x100000); *p = 123;

              printf("test MM at VA=127MB\n");
              p = (int *)(127*0x100000); *p = 123;

              printf("test MM at VA=128MB\n");
              *p = (int *)(128*0x100000); *p = 123;

              printf("test MM at VA=512MB\n");
              *p = (int *)(512*0x100000); *p = 123;
```

(1). Which of these will generate data_abort faults? WHY?__________________

(2). When a data_abort fault occurs, the program displays some error messages.
DRAW a diagram (with reason) to show the control flow of the CPU from where the fault occurred to where it shows the error messages.