LAB3pre Work: Processes in an OS Kernel
DUE: 9-23-2021
Answer questions below. Submit a (text-edit) file to TA

1. READ List: Chapter 3: 3.1-3.5

What's a process? (Page 102) A process is the execution of an image through a sequence of executions regarded by the OS kernel as a single entity for using system resources.

Each process is represented by a PROC structure.
Read the PROC structure in 3.4.1 on Page 111 and answer the following questions:

What's the meaning of:
pid, ppid? pid is the process ID while ppid is the parent process ID
status? status represents the status of the process
priority? corresponds to the scheduling priority of the process
event? the event value to sleep on
exitCode? the exit value

READ 3.5.2 on Process Family Tree.
What are the PROC pointers child, sibling, parent used for? The child points to the first child of the process while sibling points to a list of other children of the same parent which is was parent is used for.

2. Download samples/LAB3pre/mtx. Run it under Linux.
MTX is a multitasking system. It simulates process operations in a Unix/Linux kernel, which include
fork, exit, wait, sleep, wakeup, process switching

```
/*********** A Multitasking System ************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "type.h"    // PROC struct and system constants

// global variables:
PROC proc[NPROC], *running, *freeList, *readyQueue, *sleepList;

running    = pointer to the current running PROC
freeList   = a list of all FREE PROCs
readyQueue = a priority queue of procs that are READY to run
sleepList  = a list of SLEEP procs, if any.

Run mtx. It first initialize the system, creates an initial process P0.
P0 has the lowest priotiry 0, all other processes have priority 1

Ater initialization,
     P0 forks a child prcoess P1, switch process to run P1.

                The display looks like the following
------------------------------------------------------------------------------
Welcome to KCW's Multitasking System
1. init system
```

```
freeList = [0 0]->[1 0]->[2 0]->[3 0]->[4 0]->[5 0]->[6 0]->[7 0]->[8 0]-
>NULL

2. create initial process P0
init complete: P0 running

3. P0 fork P1 : enter P1 into readyQueue

4. P0 switch process to run P1
   P0: switch task
   proc 0 in scheduler()
   readyQueue = [1 1]->[0 0]->NULL
   next running = 1
   proc 1 resume to body()

proc 1 running: Parent=0 childList = NULL
freeList  = [2 0]->[3 0]->[4 0]->[5 0]->[6 0]->[7 0]->[8 0]->NULL
readQueue = [0 0]->NULL
sleepList = NULL
input a command: [ps|fork|switch|exit|sleep|wakeup|wait] :
--------------------------------------------------------------------------
5.                      COMMANDS:
ps     : display procs with pid, ppid, status; same as ps in Unix/Linux
fork   : READ kfork()   on Page 109: What does it do? creates a child task
and enters it into the readyQueue. Each newly created task begins execution
from the same body() function
switch : READ tswitch() on Page 108: What does it do? acts as a process
switch box where one process goes in and another process emerges
exit   : READ kexit()   on Page 112: What does it do? closes file
descriptors, releases resources, deallocates memory, then it disposes of any
children and records the exitValue for parent and then becomes zombie. If
needed, it will also initialize process P1
sleep  : READ ksleep()  on Page 111: What does it do? records event value,
then changes status to sleep then entering PROC *sleepList
wakeup : READ kwakeup() on Page 112: What does it do? iterates through
*sleepList, if index-> event is SLEEP, it then deletes from the sleepList and
enters it into the readyQueue
wait   : READ kwait()   on Page 114: What does it do? checks if caller has
any children, if they do checks if zombie, if so, it gets the pid (so it can
return in the end), copies exitCode to *status, and puts its PROC back into
freeList. If there are no zombies, it sleeps on PROC address
--------------------------------------------------------------------------

------------------------ TEST REQUIREMENTS --------------------------------
6. Step 1: test fork
While P1 running, enter fork: What happens? PROC 1 kforked child, from
freeList into readyQueue.

Enter fork many times;
      How many times can P1 fork? 7 times WHY? Once it reaches end of
freeList there are no more children to fork

Enter Control-c to end the program run.


7. Step 2: Test sleep/wakeup
Run mtx again.
```

While P1 running, fork a child P2;
Switch to run P2. Where did P1 go? P1 is now in the readyQueue WHY? because only one process can be running at a time

P2 running : Enter sleep, with a value, e.g.123 to let P2 SLEEP.
What happens? P2 is added to the sleepList WHY? P2 is added to sleepList with a specific event value.

Now, P1 should be running. Enter wakeup with a value, e.g. 234
Did any proc wake up? Nothing woke WHY? because 234 is not the event value p2 is sleeping on

P1: Enter wakeup with 123
What happens? P2 is entered back into readyQueue WHY? because the event value p2 is sleeping on was called

8. Step 3: test child exit/parent wait

When a proc dies (exit) with a value, it becomes a ZOMBIE, wakeup its parent.
Parent may issue wait to wait for a ZOMBIE child, and frees the ZOMBIE

Run mtx;
P1: enter wait; What happens? Nothing WHY? There was no zombie child to wait for and no child in readyQueue

CASE 1: child exit first, parent wait later

P1: fork a child P2, switch to P2.
P2: enter exit, with a value, e.g. 123 ==> P2 will die with exitCode=123.
Which process runs now? P1 WHY? Because P1 was forked prior P2's exit, where it resumes as it was next in the readyQueue
enter ps to see the proc status: P2 status = Zombie

(P1 still running) enter wait; What happens? P2 was added to end of freeList
enter ps; What happened to P2? Its status is now FREE

CASE 2: parent wait first, child exit later

P1: enter fork to fork a child P3
P1: enter wait; What happens to P1? P1 was added to sleepList WHY? P1 waited for P3 and then went to sleep.
P3: Enter exit with a value; What happens? P3 is returned to readyList and P1 is now running
P1: enter ps; What's the status of P3? FREE WHY? because P3 exited and initialized P1

9. Step 4: test Orphans

When a process with children dies first, all its children become orphans.
In Unix/Linux, every process (except P0) MUST have a unique parent.
So, all orphans become P1's children. Hence P1 never dies.

Run mtx again.
P1: fork child P2, Switch to P2.
P2: fork several children of its own, e.g. P3, P4, P5 (all in its childList).
P2: exit with a value.

P1 should be running WHY? Because it was at the beginning of the readyQueue
P1: enter ps to see proc status: which proc is ZOMBIE? P2
What happened to P2's children? All added to the readyQueue
P1: enter wait; What happens? P2 is added to the freeList
P1: enter wait again; What happens? P1 added to sleepList, P3 now running
WHY? Because there is no zombie, when P1 waits for zombie, it finds the next running which is P3 and then goes to sleep.

How to let P1 READY to run again? Exit with a value and then switch until P1 is running.