LAB3pre Work: Processes in an OS Kernel
DUE: 9-23-2021
Answer questions below. Submit a (text-edit) file to TA
Name: Koji Natsuhara ID: 11666900

1. READ List: Chapter 3: 3.1-3.5

What's a process? (Page 102)
A process is the execution of an image. It is 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 : process ID & ppid : parent process ID
   status   ? PROC status=FREE|READY|ZOMBIE|SLEEP
   priority ? Scheduling priority
   event   ? Event Value to sleep on
   exitCode ? Exit value

READ 3.5.2 on Process Family Tree. What are the
   PROC pointers child, sibling, parent used for?
        The child pointers point to the first child of a process.
        The sibling points to a list of other children of the same parent.
        PROC structures uses parent pointers to point to their parent PROC.

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.
switch : READ tswitch() on Page 108: What does it do?
Implements process context switching.
exit   : READ kexit()   on Page 112: What does it do?
Erases process user-mode context, closes file descriptions, releases
resources. Disposes children process and records the exitValue for the parent
to get. Becomes a ZOMBIE, but does not free the PROC. Wakes up the parent and
initializes process p1.
sleep  : READ ksleep()  on Page 111: What does it do?
Adds an event field to the PROC structure and implements a ksleep() function
which lets a process go to sleep. The ksleep() function records the event
value, changes the status to SLEEP, enters the PROC to a sleepList and then
frees the process (gives up CPU).
wakeup : READ kwakeup() on Page 112: What does it do?
Wakes up ALL the processes sleeping on the event value. If no process is
sleeping on the event, kwakeup() has no effect.
wait   : READ kwait()   on Page 114: What does it do?
At any time, a process may call a kernel function to wait for a ZOMBIE child
process. Kwait() returns pid of the ZOMBIE child and status contains the
exitCode of the ZOMBIE child. It then releases the ZOMBIE child PROC back to
the freeList for reuse.
--------------------------------------------------------------------------------

------------------------ TEST REQUIREMENTS --------------------------------
6. Step 1: test fork
While P1 running, enter fork: What happens?
Proc 1 kforks a child. Free process list Proc 2 leaves and joins readyQueue.
```

Enter fork many times;
How many times can P1 fork?
You can enter fork 7 times.
WHY?
P1 can only fork 7 times because there are only 7 other processes that P1 can 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?
Task calls scheduler() to pick the next running task which is P2. P1 then gets pushed to the readyQueue.
WHY?
Switch makes the P1 go to readyQueue and runs the next running process which is P2.

P2 running : Enter sleep, with a value, e.g.123 to let P2 SLEEP.
What happens?
P2 gets added to the sleepList with an event value 0x7b = 123. P1 begins to run and P2 gets added to childList with its status set to SLEEP.
WHY?
Whenever we set a process to sleep. It goes to sleep on an event value which represents the reason to sleep. P2 does not get added to readyQueue because it is not runnable until it is woken up by another process.

Now, P1 should be running. Enter wakeup with a value, e.g. 234
Did any proc wake up?
No proc woke up.
WHY?
Since we did not enter the event value P2 slept on, P2 event will continue to sleep unless the event value is called.
P1: Enter wakeup with 123
What happens?
SleepList is now empty. P2 is now set to READY and is added to readyQueue. P1 continues to run.
WHY?
Because we entered the event value P2 was sleeping on. P2’s status is now set to READY and added to readyQueue.

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 happens. P1 waits for ZOMBIE child, but there is no ZOMBIE child, so the wait fails.

WHY?
Process will wait for a ZOMBIE child process. If it is successful, then it will return the ZOMBIE child PID and status. It will also release the ZOMBIE child PROC as FREE for reuse.

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 is running now.
WHY?
P1 is running because we terminated P2. P2 has been terminated with an exit value 123.

enter ps to see the proc status: P2 status = ? ZOMBIE

(P1 still running) enter wait; What happens?
P2 no longer is on the childList and is added to freeList

enter ps; What happened to P2?
P2's status is set to FREE and its PPID is P1.

CASE 2: parent wait first, child exit later

P1: enter fork to fork a child P3
P1: enter wait; What happens to P1?
P1 waits for ZOMBIE child and P3 begins to run. P1 is added to the sleepList.

WHY?
Since P1 could not find a ZOMBIE child, P1 goes to sleep on its own PROC address, waiting for a child to terminate.

P3: Enter exit with a value; What happens?
P3 is terminated with event value 123. P3 then gives all children to P1 and wakes up P1. P1 waited for ZOMBIE child P3.

P1: enter ps; What's the status of P3?
The status of P3 is FREE.

WHY?
Although we terminated with P3 with an exit value. P1 was set to wait and search for a ZOMBIE child to FREE. Thus when we terminate P3, it is immediately freed by 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?
P1 should be running because P1 was next in queue in the readyQueue. When we terminate P2, the next process in the readyQueue will begin to run.
P1: enter ps to see proc status: which proc is ZOMBIE?
P2 is a ZOMBIE

What happened to P2's children?
P2’s children are now P1’s children. P1 is now the parent process of all of P2’s children.
P1: enter wait; What happens?
P1 FREES P2 and P1 still has P2’s children.

P1: enter wait again; What happens?
P1 is added to the sleepList with its own ID as it’s event. P3 begins to run.

WHY?
Since there are no ZOMBIE children for P1 to FREE, P1 sleeps on its own PROC address.

How to let P1 READY to run again?
EXIT P3 with an event value to set P1 to READY. Then switch twice so that P1 is now running.