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

Reagan Kelley
011663871

1. READ List: Chapter 3: 3.1-3.5

What's a process? (Page 102)

A process is the execution of an image (A memory area containing the execution's code, data, and stack). A process is a sequence of executions recognized 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? Process ID that identifies a process / Parent Process ID
- status ? Current Status of the process, PROC status = FREE | READY, etc.
- priority ? Indicates process scheduling priority
- event ? Indicates the event for the process 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 Proc pointers (child, sibling, and parent) are used to implement a binary tree data structure. The child pointer points to the first child of the process, and the sibling pointer points to a list of the other children who share the same parent. The parent pointer is initialized out of convenience, points up to the parent that all the siblings share.

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 priority 0, all other processes have priority 1

After initialization,
     P0 forks a child process 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, consisting of 3 separated steps (SAVE,
scheduler(), RESUME)

exit   : READ kexit()   on Page 112: What does it do?

Handles process termination. Follows the following procedure: Erase process
user-mode context, dispose of child processes (if any), record exitValue in
PROC.exitCode for parent to get, become a ZOMBIE, wake up parent (Also P1 if
needed).

sleep  : READ ksleep()  on Page 111: What does it do?

Puts process to sleep. Records event value in PROC.event, change process
status to sleep, enters caller into sleepList, and gives up the CPU using
tswitch().

wakeup : READ kwakeup() on Page 112: What does it do?
```

A process or interrupt handler calls wakeup with an event value parameter. kwakeup() then traverses the sleepList awakening all processes with that event value, adding them to the readyQueue and then removing it from the sleepList.

wait : READ kwait() on Page 114: What does it do?

A parent process calls kwait(), and returns the pid and exitCode of a ZOMBIE child if it has one. kwait() will then release the ZOMBIE child PROC back to the freeList for reuse. If the process has children but none are dead yet, the process will sleep.

---------- TEST REQUIREMENTS ----------

6. Step 1: test fork
While P1 running, enter fork: What happens?

A child task was created with pid 2, whose parent is P1. P2's status is ready.

Enter fork many times;
How many times can P1 fork? P1 can fork seven times.

WHY? When mtx was initialized, NPROC was pre-set to 9, and thus initialized nine PROCs. At the time P1 was forked (before user input), two of those PROCs (P0 and P1) had been added to the readyQueue. So, at the time when I repetitiously called fork(), there were only 7 PROCs left to add to the readyQueue.

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 was put into the readyQueue

WHY? Within the implementation switch, scheduler() is called. This function's job is to pick the next running task, which in this case is P2. Since P1 is still READY to run, it is put into the readyQueue by priority.

P2 running : Enter sleep, with a value, e.g.123 to let P2 SLEEP.
What happens? P2 was added to the sleepList and its status was changed to SLEEP. P1 is now the running task.

WHY? When the sleep function is called, the event parameter is recorded on the running process. That process' status will change to SLEEP, and then for convenience added to the sleepList. Then, tswitch() is called. Due to the nature of tswitch(), P2 in this case is not added back onto the readyQueue, and P1, being the next process in the readyQueue, it becomes the new running task.

Now, P1 should be running. Enter wakeup with a value, e.g. 234
Did any proc wake up? No.

WHY? Within the wakeup function call, each process in sleepList is checked to see if it matches the wakeup event value. If the event value of any process matches that of the wakeup event value, it is removed from SleepList and added to readyQueue with a READY status. The only process currently in sleepList is P2, and its event value is 123. The wakeup event value is 234 which won't match P2, so P2 doesn't wakeup. Thus, nothing changed in the environment.

P1: Enter wakeup with 123
What happens? P2 awoke and was added to the readyQueue was a READY status.

WHY? Unlike the last wakeup call, this wakeup event value is 123, which does match P2. So, P2 was removed from sleepList and added to the readyQueue with a READY status. It is important to note wakeup() never calls tswitch() so the running process doesn't change.

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.

WHY? The function wait() will search the caller's children to check if any of them are a ZOMBIE child. If so, they are sent back to the freeList for reuse. In this case, P1 has no children, so wait() returned a -1 and raised an error message. The error is non-fatal and results in no change.

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.

WHY? Exit will set P2's status to ZOMBIE and will wakeup its parent. In this case it is P1. P1, also being the next in the readyQueue becomes the new running process.

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

(P1 still running) enter wait; What happens? P1's ZOMBIE children are sent back to freeList for reuse. P2 was a ZOMBIE child of P1, and thus was freed .
enter ps; What happened to P2? P2's status was set to FREE and put back into freeList.

CASE 2: parent wait first, child exit later

P1: enter fork to fork a child P3
P1: enter wait; What happens to P1? It was put to SLEEP.

WHY? When wait() is called by a process with children, but none of them are ZOMBIE children, that process will sleep on its PROC address. P1 has a child, P3, but it is not a ZOMBIE child yet. So, P1 was put to sleep and P3 became the new running process.

P3: Enter exit with a value; What happens? P3 was freed and P1 awoke as the new running process.

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

WHY? When P3 calls P1 to wakeup towards the end of the exit() algorithm. This will raise a flag next time P1 becomes the running process. When tswitch() is ran, wait() will be ran on P1. P1 will see that it has a ZOMBIE child and will insert it back into freeList.

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 was next in readyQueue
P1: enter ps to see proc status: which proc is ZOMBIE? P2
What happened to P2's children? P2's children became the children of P2's parent. In this case, that is P1.

P1: enter wait; What happens?

P1 will look at all its children and if any are ZOMBIE children, he will bury them, setting their status to FREE and adding them back to freeList. P2 is it's only ZOMBIE child, so it was freed and set back to freeList.

P1: enter wait again; What happens? P1 was put to sleep, and P3 became the new running process.

WHY? P1 adopted P2's children when P2 exited. So, when we called wait on P1 this time, it had children but no ZOMBIE children. When this happens, the process will sleep on its PROC address. P3 then became the new running process when tswitch() was called by sleep().

How to let P1 READY to run again?

We simply need to call exit on any of P1's children. P3 being the current running process, we can call exit, and that will wake up its parent, that being P1. It is also important to note P1 won't be the running process yet as it is in the back of the readyQueue.