Assignment 4

By Bo Henderson (rojahend) and Burde Prerana Kamath (bkamath)

Question 1

> Consider a modification to the code that stores processor state in the process table instead of on the process's stack (e.g., assume the process table entry contains an array that holds the contents of registers). What are the advantages of each approach?

Process Table:

- Pro: Everything we need for a context switch is in the process table; that should speed things up
- Con: Redundant data storage, the stack inherently needs to store some of the same info that the registers would dump into the process table (eg, argument values)
- Con: Process table is bloated & we might need more storage space for relative pointers

Process Stack:

- Pro: Process table is smaller, more compact
- Pro: no need to store redundant data, register data is stored in one place: the stack
- Con: Needs to move between process table and stack during context switch; that should slow things down

Question 2

When a process kills itself, kill deallocates the stack and then calls resched, which means the process continues to use the deallocated stack. Redesign the system so a current process does not deallocate its own stack, but instead moves to a new state, PR_DYING. Arrange for whatever process searches the process table to look for dying processes, free the stack, and move the entry to PR_FREE.

```
/* kill.c - kill */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  kill  -  Kill a process and remove it from the system
 *------------------------------------------------------------------------
 */
syscall kill(
      pid32     pid     /* ID of process to kill    */
    )
{
    intmask mask;           /* Saved interrupt mask     */
    struct  procent *prptr;     /* Ptr to process' table entry  */
    int32   i;          /* Index into descriptors   */

    mask = disable();
    if (isbadpid(pid) || (pid == NULLPROC)
        || ((prptr = &proctab[pid])->prstate) == PR_FREE) {
        restore(mask);
        return SYSERR;
    }

    if (--prcount <= 1) {       /* Last user process completes  */
        xdone();
    }

    send(prptr->prparent, pid);
    for (i=0; i<3; i++) {
        close(prptr->prdesc[i]);
    }

    switch (prptr->prstate) {
    case PR_CURR:
        prptr->prstate = PR_DYING;
        resched();

    case PR_SLEEP:
    case PR_RECTIM:
        unsleep(pid);
        prptr->prstate = PR_FREE;
        break;

    case PR_WAIT:
        semtab[prptr->prsem].scount++;
        /* Fall through */
```

```
    case PR_READY:
        getitem(pid);       /* Remove from queue */
        /* Fall through */

    default:
        prptr->prstate = PR_FREE;
    }

    restore(mask);
    return OK;
}
```

I think that newpid() would be a good place to free the stack instead. It can deliver the coup de grace to a dying process and then use that pid the same way it might use one that's PR_FREE.

```
/* create.c - create, newpid */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  newpid  -  Obtain a new (free) process ID
 *------------------------------------------------------------------------
 */
local   pid32   newpid(void)
{
    uint32  i;          /* iterate through all processes*/
    static  pid32 nextpid = 1;  /* position in table to try or  */
                    /*  one beyond end of table */

    /* check all NPROC slots */

    for (i = 0; i < NPROC; i++) {
        nextpid %= NPROC;   /* wrap around to beginning */
        if (proctab[nextpid].prstate == PR_FREE) {
            return nextpid++;
        } else if (proctab[nextpid].prstate == PR_DYING) {
            prptr = &proctab[nextpid];
            freestk(prptr->prstkbase, prptr->prstklen);      // Free stack here instead
            prptr->prstate = PR_FREE;
            return nextpid++;
        } else {
            nextpid++;
        }
    }
    return (pid32) SYSERR;
}
```

Question 3

> Function resume saves the resumed process's priority in a local variable before calling ready. Show that if it references prptr->prprio after the call to ready, resume can return a priority value that the resumed process never had (not even after resumption).

A modified copy of resume.c can be found below. It has been modifed to reflect the question being asked.

```
/* resume.c - resume */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  resume  -  Unsuspend a process, making it ready
 *------------------------------------------------------------------------
 */
pri16   resume(
      pid32     pid     /* ID of process to unsuspend   */
    )
{
    intmask mask;           /* Saved interrupt mask     */
    struct  procent *prptr; /* Ptr to process' table entry  */
    pri16   prio;           /* Priority to return       */

    mask = disable();
    if (isbadpid(pid)) {
        restore(mask);
        return (pri16)SYSERR;
    }
    prptr = &proctab[pid];
    if (prptr->prstate != PR_SUSP) {
        restore(mask);
        return (pri16)SYSERR;
    }
    ready(pid);                 /* Opportunity 1 */
    prio = prptr->prprio;       /* save a copy of the priority to return later */
    restore(mask);              /* Opportunity 2 */
    return prio;
}
```

The important thing to notice is that, in the last few lines there are 2 opportunities for an arbirary number of other functions to execute.

1. Ready calls resched() internally
2. restore(mask) might re-enable interrupts causing another process to start executing.

Note that at either of these points, another process could call chprio() and change the priority of some process. If we swapped the lines `prio = prptr->prprio` and `ready(pid)` then we're saving our local variable in between these two points in time when the target process's prioritiy might change. As a result, there is a change that this processes's priority might change, this changed priority is saved, and then it's changed again right before the saved value is returned. In this case, the returned priority is neither the priority of the process before it was resumed nor the priority of the process after it was resumed.