OS Assignment 2

GETPIDX()

In this assignment we will try to simulate tarp in XINU. When an software interrupt is raised with interrupt number 46. System dispatcher will check the function which called “int46” .

```
_Xint46:
    cmpl $20 , %eax
    je L1
    cmpl $21, %eax
    je L2
    cmpl $22, %eax
    je L3
    cmpl $23, %eax
    je L4
```

The value is compared with eax and jumped to respective label.

We will test two functions getpid() and getpidx() form main. If both returns the same value our test is passed.

getpidx.c calls int46 which calls get pid if both are equal our trapping is working fine.

```
        [0x00100000 to 0x001195EC]
    133448 bytes of data.
        [0x0011CCA0 to 0x0013D5E7]

I am K Meher Hasanth and mu username is kmeherha.
Test process is running main():
3
3
```

```c
#include <xinu.h>

process main(void)
{

    // kprintf("\nHello World!\n");
    //  kprintf("\nI'm the first XINU app and running function main() in
system/main.c.\n");
    //  kprintf("\nI was created by nulluser() in system/initialize.c using
create().\n");
    //  kprintf("\nMy creator will turn itself into the do-nothing null process.\n");
    //  kprintf("\nI will create a second XINU app that runs shell() in shell/shell.c
as an example.\n");
    //  kprintf("\nYou can do something else, or do nothing; it's completely up to
you.\n");
    //  kprintf("\n...creating a shell\n");

    // /* Run the Xinu shell */

    // recvclr();
    //resume(create(shell, 8192, 50, "shell", 1, CONSOLE));

    kprintf("%d\n",getpid());
    kprintf("%d\n",getpidx());
```

Main function test.

Getmemx()

```
_Xint46:
    cmpl $20 , %eax
    je L1
    cmpl $21, %eax
    je L2
    cmpl $22, %eax
    je L3
    cmpl $23, %eax
    je L4
```

This function calls interrupt 46 with 21 in eax. This gives control to L2

In L2 label we have simulated trap.

Testing this function.

We have tested this function by storing the 2 variables in pointer *a and *b of size int.

Initially we called getmem which allocated lowest available address for *a
We called Getmemx which called the the next available address for *b

If we notice carefully address location differ only by 8 bytes. Size of int. Hence we can confirm that our function is working correctly

```
103981 bytes of Xinu code.
       [0x00100000 to 0x0011962C]
133448 bytes of data.
       [0x0011CCA0 to 0x0013D5E7]

I am K Meher Hasanth and mu username is kmeherha.
Test process is running main():
a: 1591e0
b: 1591e8
```

Code in main

```
#include <xinu.h>

process main(void)
{



    // kprintf("%d\n",getpid());
    // kprintf("%d\n",getpidx());
    int *a=(int *)getmem(sizeof(int));
    *a=100;
    kprintf("a: %x\n",a);
    int *b=(int *)getmemx(sizeof(int));
    *b=200;
    kprintf("b: %x\n",b);
```

Chpriox()
In this function we are changing the priority of a process. We are passing the “pid” of the process whose priority has to be change and then we are calling the chprio(). This will demonstrate the functioning of the of chprio(). The immediate next call is chpriox() This will raise an interrupt 46. Here L3 is activated. When L3 in-turn calls chprio() and changes the priority.

```
I am K Meher Hasanth and mu username is kmeherha.
Test process is running main():
pid = 3
newprio = 200
3,20
200
pid = 3
newprio = 400
3,32
400
```

We notice that we change the priority from 20 to 200 in chprio call

In chpriox call we change the priority from 32 to 400

Main function

```c
#include <xinu.h>

process main(void)
{
    // kprintf("%d\n",getpid());
    // kprintf("%d\n",getpidx());
    // int *a=(int *)getmem(sizeof(int));
    // *a=100;
    // kprintf("a: %x\n",a);
    // int *b=(int *)getmemx(sizeof(int));
    // *b=200;
    // kprintf("b: %x\n",b);
    kprintf("%d,%d\n",currpid,chprio(currpid,200));
    kprintf("%d\n",getprio(currpid));
    kprintf("%d,%d\n",currpid,chpriox(currpid,400));
    kprintf("%d\n",getprio(currpid));
```

Q4)
In this we calculate time taken by function is user mode.
Then we are also calculating the efficiency of the cpu utilization

We used a while loop to let some time pass and then verify the amount of time passed by calling the function again.

We also call cpuutil to calculate the efficiency of the the cpu.

```
400
cpu time before while loop = 7
 cpu time after while loop = 10
 cpuutil = 60
```

Output
Testing code.

```c
    kprintf("cpu time before while loop = %d\n ",usercpu(currpid));
    int i = 0;
    while(i < 10000) {
        i++;
    }
    kprintf("cpu time after while loop = %d\n ",usercpu(currpid));
    kprintf("cpuutil = %d\n",cpuutil());
    return OK;
```