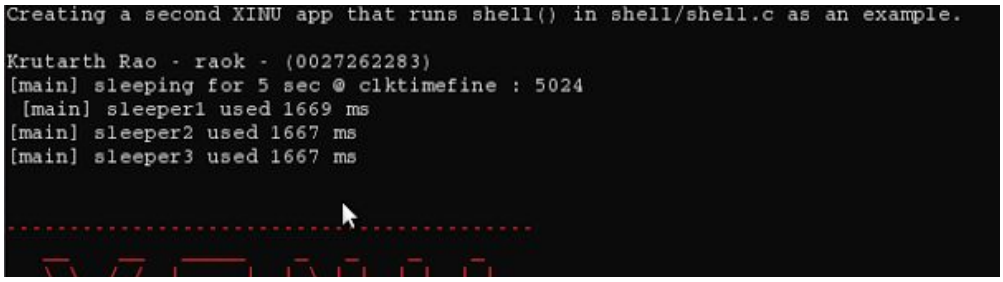
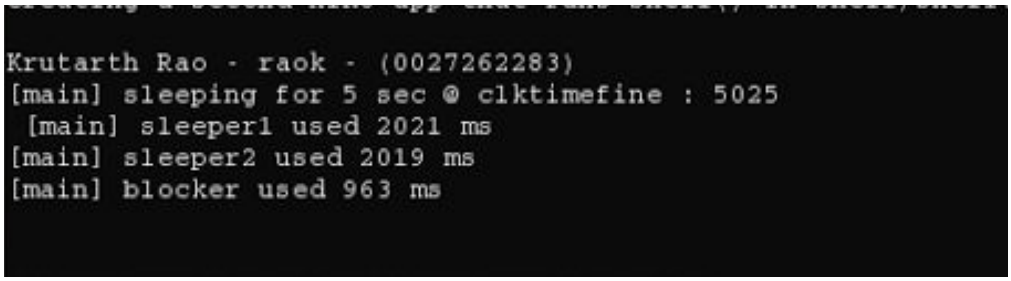
3

When 3 processes are created with equal priority and less than main's priority, they receive equal share of the CPU. Actual output:



Once a process sleeps between computations it emulates a blocking call. The blocker process receives lower CPU time.



If one of the three processes is created with a lower priority than others and other methods do not have blocking calls, the lower priority is starved since it is never context switched in.

4.2

Xinu was modified such that the ready list is sorted in ascending order of cpu used. Two new fuctions `insert_new()` and `newqueuereverse()` were created to implement this functionality. The keys in the ready list priority queue were the `prcpuused` field for the process. Priorities (`prprio` field) of the processes were left unchanged as they no longer affect scheduling. `nullproc` cpuusage is set to `MAXKEY` at initialization so it is never at the head of the readylist unless there are no other processes in the queue.
The scheduling was tested using the functions below:

```
void looper(int n)
{
    while(TRUE)
        kprintf("[looper%d] \n",n);
    return;
}
void blocker()
{
    while(TRUE)
    {
        int i;
        kprintf("[blocker]\n");
        for(i = 0; i < 500000; i++);
            kprintf("[blocker_for%d]\n", i);
        sleepms(12);
    }
    return;
}
```

`looper()` simulates a CPU bound process that doesn't give up CPU voluntarily.
`blocker()` simulates an I/O bound process that gives up CPU during execution.

The main() method used the following code to test the scheduling correctness:

```
pid32 s_id1 = create(looper, 515, 19, "looper1", 1, 1);
pid32 s_id2 = create(looper, 515, 19, "looper2", 1, 2);
pid32 s_id3 = create(blocker, 585, 19, "blocker", 0);
resume(s_id1);
resume(s_id2);
resume(s_id3);
sleepms(5000);
kill(s_id1);
kill(s_id2);
kill(s_id3);


kprintf("[main] Resumed after sleep for 5 sec @ clktimefine : %d\n", clktimefine);
```

The results were as follows with dynamic scheduling:

```
[main] sleeping for 5 sec @ clktimefine : 5132
[main] looper1 used 1702 ms
[main] looper2 used 1706 ms
[main] blocker used 1702 ms
```

The results show that CPU and I/O bound processes received almost equal share of CPU cycles. Therefore, dynamic scheduling was successful.

4.3

1. When all processes are CPU bound, the following code tests the runtime of these processes (main sleeps for 10 seconds to allow the scheduling to reach a "steady state"):

NOTE: main's initial cputime is set to 0 so it can create all processes without getting switched out.

```
pid32 s_id1 = create(cpubnd, 260, 19, "cpubnd1", 0);
pid32 s_id2 = create(cpubnd, 260, 19, "cpubnd2", 0);
pid32 s_id3 = create(cpubnd, 260, 19, "cpubnd3", 0);
pid32 s_id4 = create(cpubnd, 260, 19, "cpubnd4", 0);
resume(s_id1);
resume(s_id2);
resume(s_id3);
resume(s_id4);
sleepms(10000);
kill(s_id1);
kill(s_id2);
kill(s_id3);
kill(s_id4);

kprintf("[main] Resumed after sleep for 5 sec @ clktimefine : %d\n", clktimefine);
```

The CPU time used by each process is accessed as follows:

```
struct  procent *prptr;

prptr = &proctab[s_id1];
uint32 sleeperTime =  prptr->prcpuused;
kprintf("[main] cpubnd1 used %d ms\n", sleeperTime);
```

The results are as follows:

```
currpid: 6, for_i : 471, priority(cpuuse[main] Resumed after sleep for 5 sec @ clktimefine : 10224
[main] cpubnd1 used 2551 ms
[main] cpubnd2 used 2551 ms
[main] cpubnd3 used 2551 ms
[main] cpubnd4 used 2551 ms
```

All processes received equal shares of the CPU.

2. When All processes are I/O-bound, they also receive appx. equal share of the CPU although each process didn't require the amount of CPU time the CPU bound processes did.

```
[main] Resumed after sleep for
[main] iobnd1 used 5 ms
[main] iobnd2 used 6 ms
[main] iobnd3 used 6 ms
[main] iobnd4 used 7 ms
```

3. When a *Half-and-half* mix of processes is used, the cpu bound processes receive equal share of the cpu and the I/O bound processes receive equal share of the cpu. (after experimenting with *LOOP1* and *LOOP2*, I/O bound processes were made to run for longer)

```
[main] cpubnd1 used 4591 ms
[main] cpubnd2 used 4581 ms
[main] iobnd1 used 466 ms
[main] iobnd2 used 466 ms
```

Since the I/O bound processes finish execution before the CPU bound processes, the CPU bound processes share the remaining time amongst themselves before they are terminated.

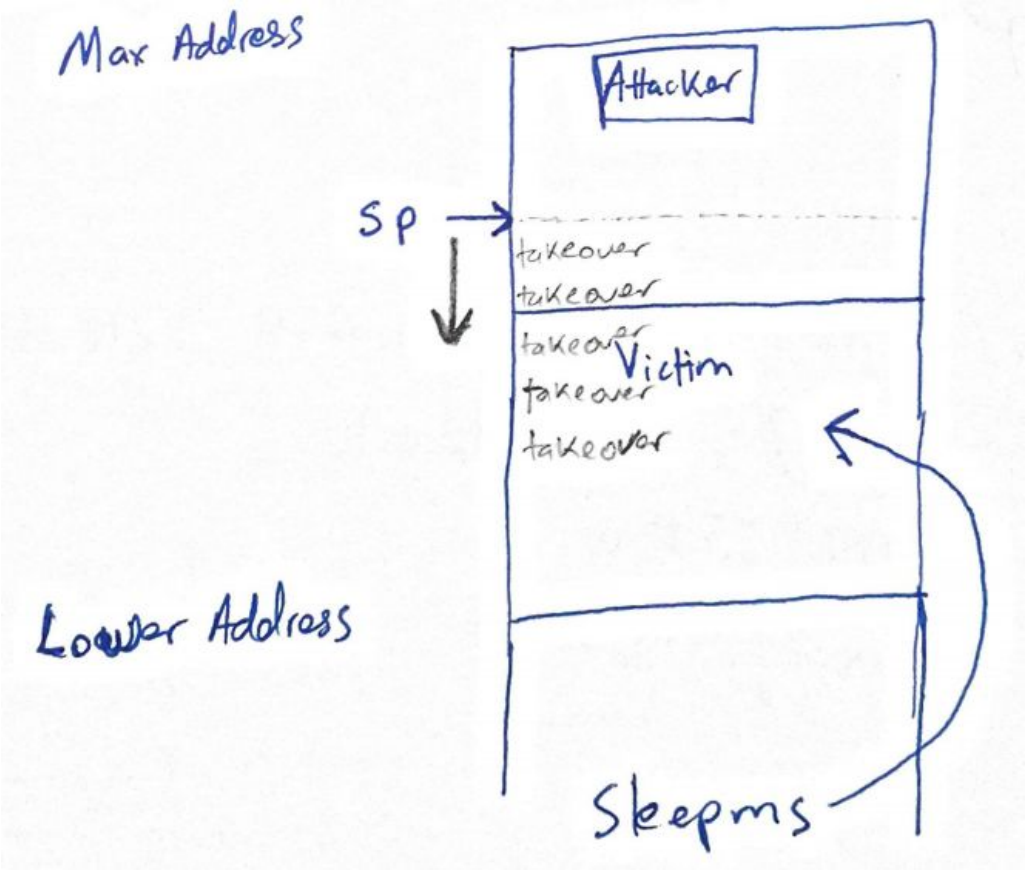
5

Output:

```
Creating a second XINU app that runs shel

Krutarth Rao - raok - (0027262283)
Vtakeover succeeded
takeover succeeded
takeover succeeded
takeover succeeded
takeover succeeded
takeover succeeded
```

The attacker stack is above the victim stack so once the victim sleeps, the attacker obtains the value of it's own stack pointer from `%esp`, stores it in a local variable **sp** and overwrites the content in ***sp** to `takeover` and keeps decrementing the pointer and overwriting till the return address for the victim function is overwritten. Once sleep returns, the address at the returned location points to takeover. (see diagram below)



Bonus

To implement an RT process, add an entry in the process table `isrt`. This is set to TRUE when a RT process is created. The resched() function is modified such that:

```
if ( ptold->prcpuused < firstkey(readylist) ) {
                return;
        }
```

is changed to:

```
if ( ptold->prcpuused < firstkey(readylist) || ptold->isrt == TRUE ) {
                return;
        }
```

Using the new `isrt` flag, the scheduler will ever execute the code to context switch the RT process out.