Contents

Submitted by: Neeharanshu Vilas Vaidya

Unity ID: nvaidya2

Part 1: Understanding Xinu's process management and context switch.

1. **What is the ready list? List all the system calls that operate on it.**

Ans. A ready list is a linked list implemented in the form of a queue to maintain the list of processes that are ready.

2. **What is the default process scheduling policy used in Xinu? Can this policy lead to process starvation? Explain your answer.**

Ans. Xinu schedules the process with the highest priority every time slice. Yes, this can lead to process starvation since none of the lower priority process get a fair chance to run unless the highest priority process is put to sleep.

3. **When a context switch happens, how does the resched function decide if the currently executing process should be put back into the ready list?**

Ans. **I**f the process is currently running and is not the highest priority process in the ready list then it puts it in the readylist.

4. **Explain what a stack frame is and why it is needed during a function call.**

Ans. A stack frame is a frame of data that gets pushed onto the stack. During a function call, the data of 1 function is saved on the stack and retrieved later when the function resumes execution.

5. **The function ctxsw takes two parameters. Explain the use of these parameters. Which assembly instruction(s) set(s) the Instruction Pointer register to make it point to the code of the new process?**

Ans. The ctxsw function takes in the old process stackpointer and the new process stack pointer.
These parameters are used so the CPU can load data to/from the currently executing function's stack frame.
The below instruction swaps the processes:

```
movl    (%eax),%esp    /* Pop up new process's SP    */
```

6. **Analyze Xinu code and list all circumstances that can trigger a scheduling event.**

Ans. The circumstances in which a scheduling event is triggered are:

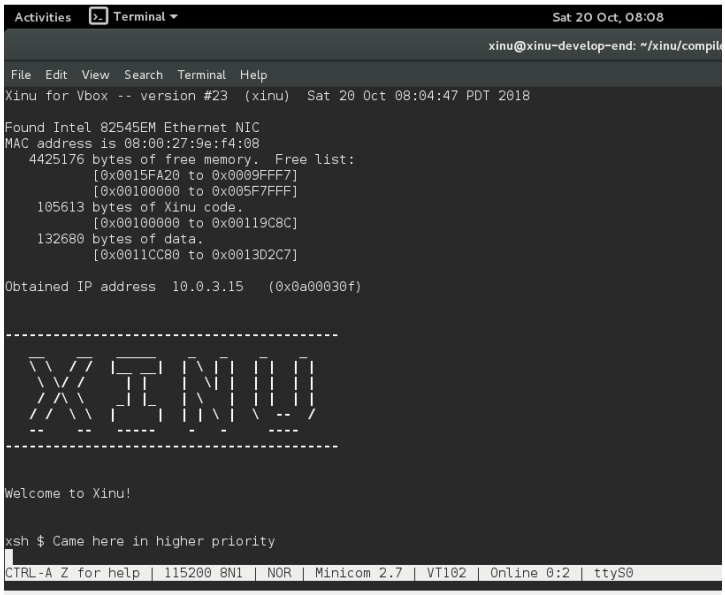
1. Expiry of pre-empt
2. Function call to ready either by resume or wake up.

7. *Coding tasks:*
    1. If your answer to question #2 above is affirmative, write a main.c that implements a use case demonstrating the problem. Explain your use case and code in the report.

```
/*  main.c  - main */
#include <xinu.h>
```

```
void timed_execution_higher(){
    kprintf("Came here in higher priority\n");
    while(1);
    //struct procent *ptr = &proctab[currpid];
    //while(1);
//  kill(currpid);
    return;
}
void timed_execution_lower(){
    kprintf("Came here in lower priority\n");
    return;
}
process     main(void)
{
    /* Run the Xinu shell */
    recvclr();
    resume(create(shell, 8192, 50, "shell", 1, CONSOLE));
    resume(create(timed_execution_higher, 8192, 49, "timed_execution_higher", 0));
    resume(create(timed_execution_lower, 8192, 2, "timed_execution_lower", 0));
    /* Wait for shell to exit and recreate it */
    while (TRUE) {
            receive();
            sleepms(200);
            kprintf("\n\nMain process recreating shell\n\n");
            resume(create(shell, 4096, 20, "shell", 1, CONSOLE));
    }
    return OK;

}
```



The code never reaches beyond the higher priority process. Hence, starving other process like main().

Part 2: Lottery Scheduling

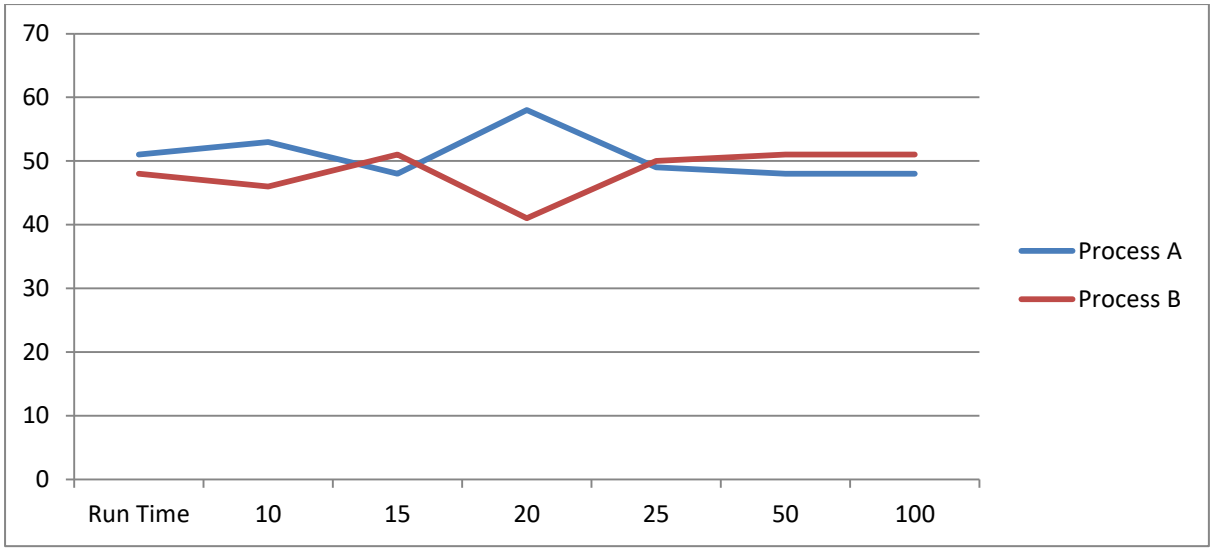
1. **A *brief* description of your implementation approach, indicating the files involved in the implementation of lottery scheduling.**

    a. Create a new file named create_user_proc: This file contains the function to create a user process as well as to call the function "Timed Execution" that defines this user process.
    b. Queue.c and Queue.h: Implement a data structure to arrange the list of user processes in order of ticket assigned.
    c. Resched.c: Implement the core functionality of lottery scheduling here.
        1. If a sys call is currently running then check if there is any other sys call with higher priority. If not, return.
        2. If a user process calls resched then check if a sys call other than NULL Process is running. Prioritize the sys call over user process. If there is no sys call in ready list then pick a PID at random and schedule that process.
        3. If a user process or a sys call has called resched but there is no other ready process then switch to NULL PROC.

2. Analyze the fairness of your scheduler. To this end, write a test case file that spawns and runs two user processes with the same run_time parameter value. Run your test case multiple times using increasing run_time values. Plot the ratio between the actual execution time of the two processes when run together (i.e., execution-timeP1/execution-timeP2, where P1 and P2 are the two processes spawned) against the value of parameter run_time. You can select the number of data points and the value of the run_time parameter as you like, provided that you make a selection that leads to a meaningful plot. Include the plot and a brief discussion of it (no more than a couple of sentences) in the report.

    Ans. Here we will vary the run time of each process and see the %execution of the CPU.
    Process A Tickets = 1000
    Process B Tickets = 1000

Run Time(ms)	Sleep Duration (ms)	Process A: CPU Utilization (%)	Process B: CPU Utilization (%)
10	50	51	48
15	50	53	46
20	50	48	51
25	50	58	41
50	100	49	50
100	200	48	51
500	1000	48	51



Fairness Chart

Part 3: Multi-Level Feedback Queue (MLFQ) scheduling policy

Guidelines:

1. Re-programmed the scheduler to follow the below rules:
    a. *Rule 1*: if Priority(A)>Priority (B), A runs
    b. *Rule 2*: if Priority(A)=Priority(B), A&B run in RR fashion
    c. *Rule 3*: initially a job is placed at the highest priority level
    d. *Rule 4*: once a job uses up its time allotment TA_L at a given level, its priority is reduced
    e. *Rule 5*: after some time period *S*, move all jobs in the topmost queue
2. Given highest priority to sys calls.
3. Modified the Process Control block to maintain number_bursts, burst_duration, sleep_duration, run_time, sys_call, qnum, time_allot, burst_done_flag
4. In clkhandler.c, decrement burst duration every cycle. If burst duration is 0 then put the user process to sleep for sleep_duration ms and decrement number_bursts. If time_allot is 0 then move the user process to a lower priority queue.