Name: Rakesh Kannan
Student ID: 200535445
ECE 565 Operating Systems Design

Project 1 Report:

Q1. What is the readylist? List all the system calls that operate on it.
Ans:
The readylist is a queue containing processes in the ready state, indicated by PR_READY. The head of the ready list is assigned a value greater than NPROC using the newqueue() function. Processes are added to the ready list based on their priority, making it a priority-based scheduling system. The ready list is set up in the sysinit()function. When a process is resumed, the resume() function calls ready(), which adds the process to the ready list and updates its state to PR_READY. The resched() system call operates on the ready list by comparing the current process with the highest-priority process in the list to decide if a context switch is necessary.

Q2. What is the default process scheduling policy used in Xinu? Can this policy lead to process starvation? Explain your answer.
Ans:
By default, Xinu employs a priority-based round-robin scheduling policy. The process with the highest priority is executed first. If multiple processes share the same priority, round-robin scheduling is utilized to ensure these processes share CPU time equally. However, this approach can result in process starvation. If high-priority processes continually occupy the CPU, lower-priority processes may be deprived of execution time, potentially leading to starvation.

Q3. 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:
If the process calling the scheduler is in the PR_CURR state, it can either keep executing or be added back to the ready list, depending on the scheduling decision. The resched() function compares the priority of the current process with the highest-priority process in the ready list. If the current process has a higher priority, it keeps executing, no context switch happens, and control returns from the resched() function. However, if the first process in the ready list has a higher priority, the current process is set to PR_READY state, and the highest-priority process from the ready list is assigned as currpid, prompting a context switch. Additionally, if the current process is not in the PR_CURR state, it won't keep executing.

Q4. Analyze Xinu code and list all circumstances that can trigger a scheduling event.
Ans:
- kill() calls resched() to terminate a process.
- yield() triggers resched() when a process voluntarily relinquishes control of the CPU.
- ready() invokes resched() when a process enters the ready state.
- suspend() triggers resched() when a process is suspended.
- receive() calls resched() when a process is waiting for a message.
- sleepms() invokes resched() when a process enters the sleep state while waiting for an event or a delay.
- wait() calls resched() when a process waits for a semaphore to complete.
- rdssetprio() triggers resched() when the current process's priority is changed.
- clkhandler() calls resched() after each quantum (time slice) passes.

Part 2: Lottery Scheduling

Implementation approach:

1) User processes have a priority level of 5, while other system processes possess a higher priority than user processes.

2) User processes are placed in the lottery queue, while system processes are added to the ready list.

3) If a system process is running and another system process in the ready list has a higher priority, a context switch will occur; otherwise, the current process continues to execute.

4) If the null system process is running, any process in the ready list will be prioritized. If the ready list is empty, the function will check for a user process in the lottery queue. If neither condition is met, the function will return, allowing the null process to continue.

5) A user process can continue to execute only if the ready list is empty, as the null process has a lower priority. In this case, lottery scheduling is performed using the select_lottery() function, after adding the current user process to the lottery queue.

include:

process.h : added user_process, runtime, turnaroundtime, num_ctxsw, arrivaltime and tickets to PCB.

prototypes.h : Added extern print_ready_list (), extern burst_execution () , extern create_user_process () , extern select_lottery (), extern single_lsprocess () and extern lsprocess_count () functions in prototypes.h file.

clock.h : added `ctr1000` variable to count the time in milliseconds since boot up.

queue.h : Updated the NQENT value by `2 as to handle userlist head and tail`

kernel.h : Updated the time slice to 10 by modifying the `QUANTUM` variable and defined userlist.

system:

create.c : added the new create_user_process function according to the specs. It works similarly to create(), but without the priority argument, and sets the user_process Boolean flag to true. The user process is assigned to 5. Additionally, set_tickets and burst_execution functions are added to perform assigning tickets to particular processes and handling burst executions with sleep.

clkinit.c : initialized `ctr1000` variable to 0.

clkhandler.c : The ctr1000 counter and the runtime of the current process are incremented in clkhandler().

initialize.c : initialized all the new members of PCB to 0 and userlist is created here in sysinit().

insert.c : modified the insert() to handle when two processes have the same number of tickets, sorting them by ascending process id.

queue.c : added a mlfq struct array based on the number of UPRIORITY_QUEUES.

ready.c : added the print_ready_list() function for debugging purposes, as specified. Additionally, modified the ready function to check if the process is a user process; if so, it appends the process to the userlist queue; otherwise, it adds it to the ready list.

kill.c : added the turnaroundtime calculation of process during kill() function flow.

resched.c : modified the resched to accomodate the lottery scheduling logic as mentioned in the specs.

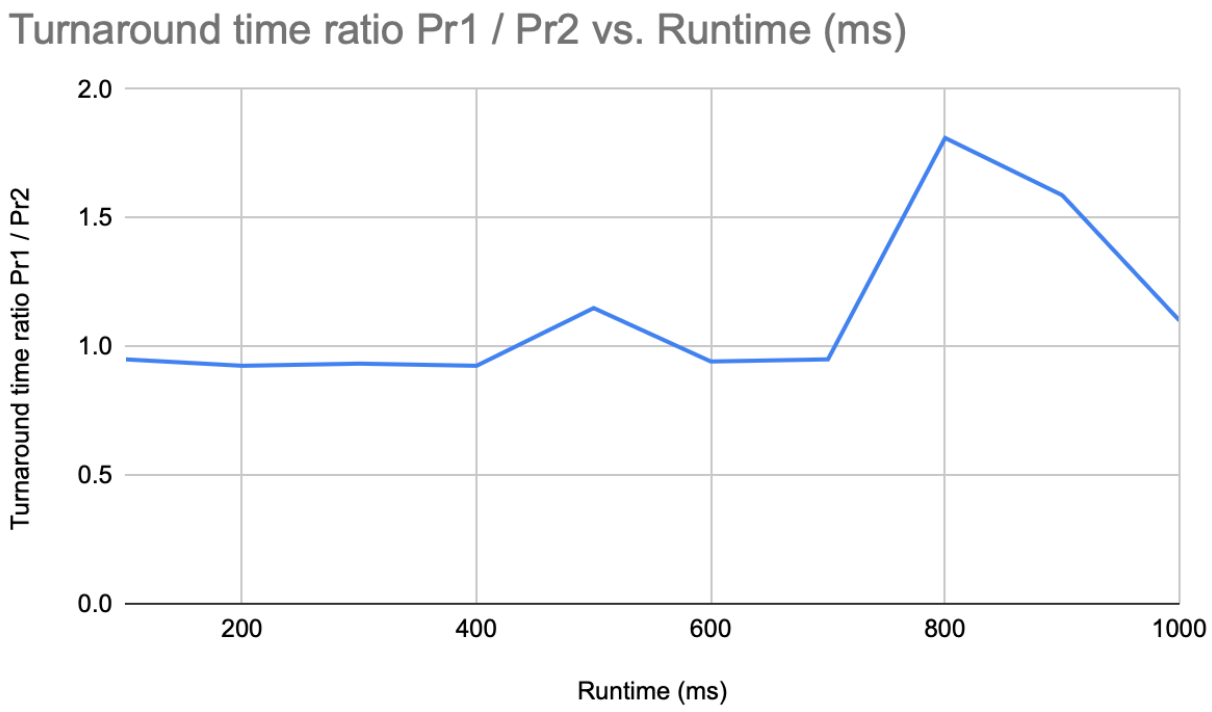
Fairness analysis:

A user application test case has been updated in main.fairness.c to evaluate fairness. Two processes, pr1 and pr2, were spawned and executed with the same runtime over 10 iterations, each featuring different runtime values. The ratio of their turnaround times was analyzed and plotted in an Excel graph.

```
#include <xinu.h>
#include <stdio.h>
void timed_execution(uint32 runtime) {
   while (proctab[currpid].runtime < runtime);
}
int main()
{
   uint32 j;
   pid32 pr1, pr2;
   float fairness; // Variable to store fairness ratio
   uint32 runtime_values[] = {100, 200, 300, 400, 500, 600, 700, 800, 900, 1000};
   uint32 num_points = sizeof(runtime_values) / sizeof(runtime_values[0]);
   // Loop through defined runtime values
   for (j = 0; j < num_points; j++) {
       uint32 current_runtime = runtime_values[j];
       // Create two user processes with the same runtime
       pr1 = create_user_process(timed_execution, 1024, "timed_execution", 1, current_runtime);
       pr2 = create_user_process(timed_execution, 1024, "timed_execution", 1, current_runtime);
       // Set the tickets for both processes
       set_tickets(pr1, 50);
       set_tickets(pr2, 50);
       // Resume both processes
       resume(pr1);
       resume(pr2);
       // Wait for both processes to complete
       receive();
       receive();
       // Sleep for a short duration to ensure processes are finished
       sleepms(50);
       // Calculate fairness ratio
       fairness = (float)proctab[pr1].turnaroundtime / proctab[pr2].turnaroundtime;
       // Print execution data
       kprintf("Runtime: %d, pr1 runtime: %d, pr1 turnaround time: %d, pr2 runtime: %d, pr2 turnaround
time: %d\n",
               current_runtime, proctab[pr1].runtime, proctab[pr1].turnaroundtime,
               proctab[pr2].runtime, proctab[pr2].turnaroundtime);

       kprintf("Fairness F = %f\n\n", fairness);
   }
   return OK;
}
```

```
Runtime: 100, pr1 runtime: 100, pr1 turnaround time: 190, pr2 runtime: 100, pr2 turnaround time: 200
Fairness F = 0.949999
Runtime: 200, pr1 runtime: 200, pr1 turnaround time: 370, pr2 runtime: 200, pr2 turnaround time: 400
```

```
Fairness F = 0.925000
Runtime: 300, pr1 runtime: 300, pr1 turnaround time: 560, pr2 runtime: 300, pr2 turnaround time: 600
Fairness F = 0.933333
Runtime: 400, pr1 runtime: 400, pr1 turnaround time: 740, pr2 runtime: 400, pr2 turnaround time: 800
Fairness F = 0.925000

Runtime: 500, pr1 runtime: 500, pr1 turnaround time: 1000, pr2 runtime: 500, pr2 turnaround time: 870
Fairness F = 1.149425

Runtime: 600, pr1 runtime: 600, pr1 turnaround time: 1130, pr2 runtime: 600, pr2 turnaround time: 120
Fairness F = 0.941666

Runtime: 700, pr1 runtime: 700, pr1 turnaround time: 1330, pr2 runtime: 700, pr2 turnaround time: 140
Fairness F = 0.949999

Runtime: 800, pr1 runtime: 800, pr1 turnaround time: 1600, pr2 runtime: 800, pr2 turnaround time: 140
Fairness F = 1.81081

Runtime: 900, pr1 runtime: 900, pr1 turnaround time: 1800, pr2 runtime: 900, pr2 turnaround time: 170
Fairness F = 1.58823

Runtime: 1000, pr1 runtime: 1000, pr1 turnaround time: 2000, pr2 runtime: 1000, pr2 turnaround time:0
Fairness F = 1.10100
```



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

Implementation approach:

1) System processes possess a higher priority than user processes. User processes are placed in the queue, whereas system processes are added to the ready list.

2) If a system process is currently running and another system process in the ready list has a higher priority, a context switch will occur to that process; otherwise, the current process will continue executing.

3) If a null system process is executing, any process in the ready list will be given priority. If the ready list is empty, the function will check for a user process in the queue. If none of these conditions are met, the function will return, allowing the null process to continue executing.

4) A user process can continue executing only if the ready list is empty since the null process has a lower priority than the user process. In this case, multi-level feedback queue (MLFQ) scheduling will be performed using the mlfq_scheduling() function.

5) This function returns the PID of the new process, checks the queues in order of priority, and schedules the first available process from the highest priority queues. The priority boosting condition is determined by the input `PRIORITY_BOOST_PERIOD` value. If the counter reaches this value, the process allotment for all processes is reset to 0, and all processes from lower priority queues, as well as those in the sleep queue, are moved to the highest priority queue after the current process is enqueued in its own queue.

include:

process.h : added user_process, runtime, turnaroundtime, num_ctxsw, arrivaltime, upgrades, downgrades, process_assigned to PCB.

prototypes.h : Added extern print_ready_list (), extern burst_execution () , extern reset_timing () and extern mlfq_scheduling () functions in prototypes.h file

clock.h : added `priority_update` variable to handle period for priority upgrade and also `ctr1000` variable to count the time in milliseconds since boot

queue.h : Updated the NQENT value by `2 * UPRIORITY_QUEUES to handle userlist head and tail`

Also added struct to handle different levels of queue along with their priority and time assigned.

```
struct mlfq_queue
{
  uint16 priority;
  qid16  level;
  uint32 time_assigned;
};
extern  struct mlfq_queue   mlfq[];
```

kernel.h : Updated the time slice to 5 by modifying the `QUANTUM` variable.

resched.h : added all these mentioned below

```
#define TIME_ALLOTMENT 10
#define PRIORITY_BOOST_PERIOD 500
#define UPRIORITY_QUEUES 5
```

system:

create.c : added the new create_user_process function according to the specs. It works similarly to create(), but without the priority argument, and sets the user_process Boolean flag to true. The user

process is assigned the priority of the highest priority queue. Additionally, other relevant fields in procent[currpid] are initialized within this function. Also added a func to perform burst executions included with sleep.

clkinit.c : initialized `priority_update` and `ctr1000` to 0.

clkhandler.c : The ctr1000 counter, the runtime of the current process, the process priority assigned, and the priority_update are all incremented in clkhandler().

initialize.c : initialized all the new members of PCB and the members of mlfq struct to 0.

insert.c : added a new function insert_mlfq() to add a process to mlfq.

queue.c : added a mlfq struct array based on the number of UPRIORITY_QUEUES.

ready.c : added the print_ready_list() function for debugging purposes, as specified. Additionally, modified the ready function to check if the process is a user process; if so, it appends the process to the mlfq queue; otherwise, it adds it to the ready list.

kill.c : added the turnaroundtime calculation of process during kill() function flow.

resched.c : modified the resched to accomodate the mlfq scheduling logic as mentioned in the specs.