Project – User Level Threads

Department of Computer Science, SIUE
Course Name: Operating System
Instructor: Dr. Igor Crk
CS514 Spring ’25

Prashanna Raj Pandit *
800817018

March 1, 2025

*ppandit@siue.edu

Contents

1 Introduction

1.1 Overview of Project and User Level Thread

This project implements user-level many-to-one threads without using the *pthreads* library, as per the requirements. Since, in user-level threads, the kernel is unaware of the user threads, we need our own scheduler to manage them. This project also uses manual context switching, as required, by utilizing the *ucontext* library. The program initializes multiple execution contexts and switches between them using getcontext(), makecontext(), and swapcontext().

The project consists of three files:

- *uthreads.h* -Contains the blueprint for the thread structure.
- *uthreads.c* -Implements both the Round Robin scheduler and the Lottery Scheduler.
- *Makefile* -to make executable of files

The graphs generated in this project are created using Matplotlib in Python. Here is the link to the GitHub repository: **LINK**

The project uses a custom setTimer() function, which generates signals to the scheduler handler when the timer triggers. Its implementation is explained in detail in section 2. Two types of threads have been created in this project:

1. Threads that run longer than their assigned time and are preempted.
2. Threads that execute quickly and yield control back to the scheduler for the next thread.

The project also demonstrates Waldspurger's experiment 5.1 - Fairness, as discussed in the Lottery Scheduling paper.

1.2 Objective

1. Implement threads which successfully runs by yielding.
2. Implement threads that are being scheduled preemptively by the round-robin scheduler.
3. Implement threads that are being scheduled preemptively by the Lottery scheduler.
4. Prove one of Waldspurger's experiments that demonstrates some process behaviors using the lottery scheduler.
5. Compare Lottery scheduler and round-robin scheduler.

2 Scheduling Algorithm Implementations

As we all know, the execution starts from main(). The user gets prompt to select the How they want to create threads and which scheduler they want to choose. It looks something like this.

```
Choose your scheduler:
1. Round-Robin Scheduler:
   - Runs 4 threads.
   - 2 threads yield, 2 threads preempt.(press 1)
2. Round-Robin Scheduler (Simplified):
   - Runs 2 threads by preemption. (Press 2)
3. Lottery Scheduler:(Verifies Waldspurger's experiments 5.1)
   - Runs 2 threads by preemption. (Press 3)
```

Figure 1: Menu

Selecting **option 1** runs the Round Robin scheduler by creating 4 threads. This meets the objective 1 and 2. here two threads execute before virtual alarm rings and yields another thread to proceeds and two threads needs preemption by the scheduler.

Selecting **option 2** creates two threads which needs preemption to execute next threads. This option is created to compare the round robin scheduler with the Lottery Scheduler with same number of threads.

Selecting **option 3** creates two threads which needs preemption and is scheduled by Lottery Scheduler. This is created to collected sample data and prove the Waldspurger's experiments and compare with round robin scheduler.

2.1 Thread creation

As per the project requirement, the thread structure and thread creation blueprints are implemented in uthreads.h header file. The thread's structure stores the thread id, tickets associated with a thread, stack size of threads, number of iterations each thread has executed and the thread context. The stack size of threads has assigned as 8KB as default size. And the queue size is 4. Two threads are created as a simple yielding threads which yields execution to the next ready threads and two threads needs preemption to execute next thread from ready queue.

When a *create_thread()* function is called, new thread is created dynamically and it saves the current execution state (registers, stack pointer, instruction pointer, signal mask) in thread $\rightarrow$ context by *getcontext(thread $\rightarrow$ context)*. The default *getcontext()* saves the current stack, but we need a separate stack for the new thread. It is created by *uc.stack*. The default stack size is 8KB. The function initializes the id, tickets and iteration to each threads. The *uc_link* pointer of new thread is pointed to the current thread context structure. This is because When the thread's function (func) finishes execution or dies, the context specified by *uc_link* will be restored. After that, the thread context is modified by *makecontext()* to setup thread execution. so that when thread is scheduled, it starts execution at func (thread_function).

```c
pp_threads* create_thread(void (*func)(), int id, int tickets) {
    pp_threads* new_thread = (pp_threads*)malloc(sizeof(pp_threads));
    getcontext(&new_thread->context);
    new_thread->id = id;
    new_thread->tickets = tickets;
    new_thread->iterations = 0;
    new_thread->context.uc_stack.ss_sp = malloc(STACK_SIZE);  // Allocate stack memory
    new_thread->context.uc_stack.ss_size = STACK_SIZE;
    new_thread->context.uc_link = &current_thread->context;
    makecontext(&new_thread->context, func, 0);
    return new_thread;
}
```

Figure 2: Thread creation

2.2 Timer and Preemption

After creating threads and pushing into the ready queue, the setTimer() function is called which sets the virtual time (*it_value*) of 100 miliseconds and the interval time (*it_interval*) of 100 miliseconds, meaning the timer will automatically reset and continue to expire every 100 miliseconds. When the virtual time expires, it send the **SIGVTALRM** signal to the *schedulerHandler()* which preempts the current executing thread and push in to ready queue and get the new thread by either Round robin or Lottery scheduler (whatever the user selects). Before switching to the new thread, the current context is saved by swapcontext() to ensure threads resume from the same point instead of restarting when they get scheduled again. But when the first time virtual clock is set there is no thread running and hance the timer doesn't start unless user level thread is executed. For that *schedulerHandler()* is called explicitely from main to start the execution of first thread.

To stop the program, the user is asked to enter the stopping time in seconds. Eg.(8, 16, 24, 32, 40, 48, 56, 64, 72, 80..........200)

```
2
Enter the stopping time. (in seconds):
```

Figure 3: A prompt that asks the user for a stop time

This is the total CPU time consumed by the process. This is implemented by setting the timer **ITIMER_PROF**, which calculates the total CPU time. And register the stopExecution() function to handle the **SIGPROF** signal after the expiration of timer. The stopExecution() function end the program when it is called.

2.3 Round robin scheduler

This is implemented in *getNextThread_by_RR()* function in *uthreads.c* file. Round robin is a resource/CPU scheduling algorithm where each process get cyclically assigned a fixed time slot. Here each threads get the same priority and follows the queue. This algorithm selects first index of the ready queue as the next threads. After selecting the next threads, it shifts the threads in the ready queue by 1, decrement the size of queue and return the next thread to the *schedulerHandler()* function.

2.4 Lottery scheduler

Lottery scheduling is a probabilistic scheduling algorithm that ensures fair CPU time distribution among threads or processes based on assigned ”tickets.” Each thread holds a certain number of tickets, representing its share of CPU time. I also call it priority because the threads having more tickets have a higher priority of execution than the threads having fewer tickets. In this project, two threads has implemented. Thread 1 has 10 tickets and thread 2 has assigned 20 tickets. When the scheduler needs to select the next thread to run, it calculates the total number of tickets and randomly draws a ticket by *rand()% total_tickets*, and the thread holding that ticket is chosen. To select the thread, which holds that ticket, the code walks through the list of processes, adding each ticket value to the *ticket_collection* variable until the value exceeds the selected ticket.

This approach naturally ensures that threads with more tickets are more likely to run but still allows lower-ticket threads a chance, promoting fairness and flexibility which is explained by Waldspurger in his paper “Lottery Scheduling: Flexible Proportional-Share Resource Management” in 5.1.

3 Experiment

3.1 Waldspurger Experiment 5.1 (Fairness)

3.1.1 Threads with same ticket allocation ratio (1:2)

In my implementation I am going to prove how Lottery Scheduling ensure fairness in a similar way as Waldspurger did.

To visualize how the fairness is achieved over time in Lottery Scheduling. Two threads were executed with 1:2 the ticket allocation. Thread 1 was assigned 10 tickets and threads 2 was assigned 20 tickets. 25 sample data was taken from 8 second of interval and the number of iteration executed by each thread was noted down as shown in following table.

Time (sec)	Thread 1 Iterations	Thread 2 Iterations
8	1413	2540
16	2817	5288
24	4212	7920
32	5699	10489
40	7012	13271
48	8435	15905
56	9831	18480
64	11208	21290
72	12641	23815
80	14068	26444
88	15472	29064
96	16799	31844
104	18093	34496
112	19447	37321
120	20714	40062
128	22095	42544
136	23427	45363
144	24859	48230
152	26099	50820
160	27297	53838
168	28616	56594
176	29776	59151
184	31209	62005
192	32561	64953
200	33748	67688

Table 1: The number of iterations executed by threads with different time intervals

Observation 1: The higher ticket allocation task consistently executes more iterations per second, approximately twice as many as the lower ticket allocation task which proves the Waldspurger Experimanet of fairness.

- Thread 2, with double the tickets, consistently completes more iterations than Thread 1.

- The gap between Thread 1 and Thread 2 increases proportionally over time, confirming the probabilistic fairness of the lottery approach.

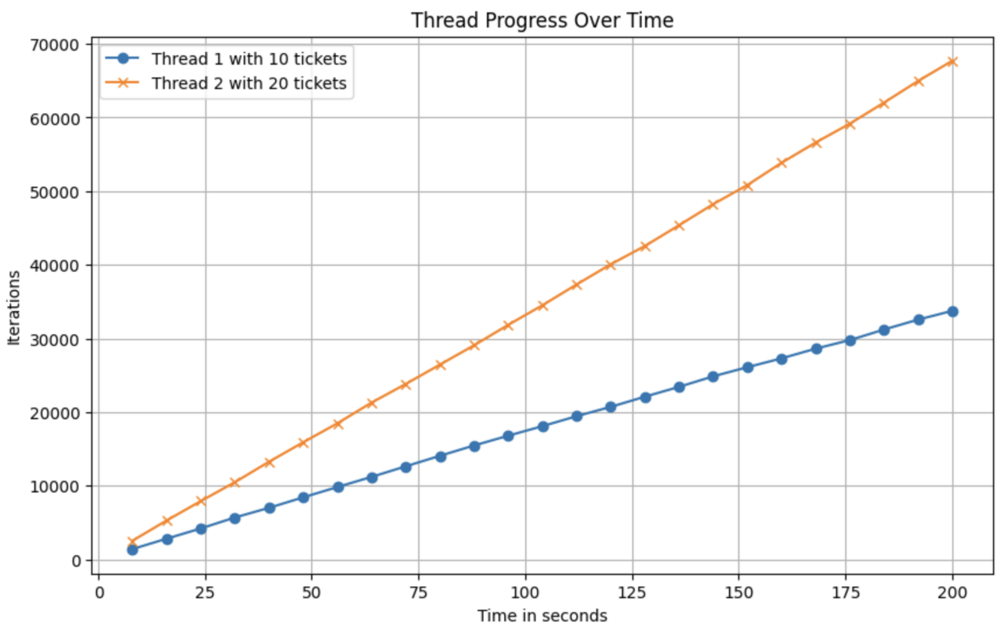

Figure 4: The plot shows the cumulative number of iterations completed by each thread over time under lottery scheduling.

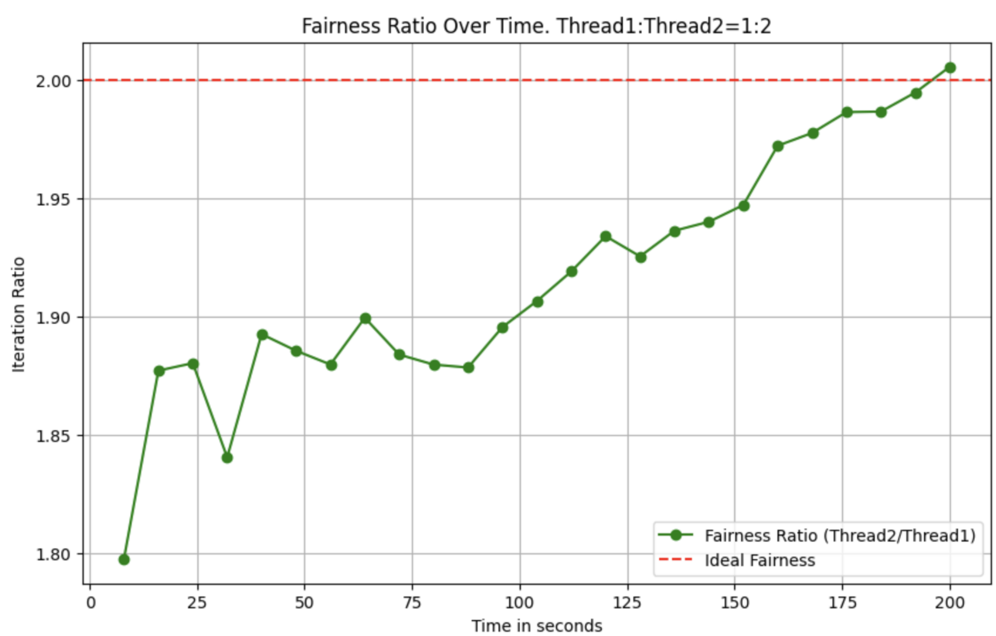

Figure 5: The ratio of iterations completed by Thread 2 relative to Thread 1 over time. (Ideal vs Observed)

Observation 2: The ideal fairness ratio, given the 1:2 ticket allocation, is 2.0, represented by the red dashed line. Figure 5 shows that the fairness approaches to the ideal when the process is executed for a longer time by decreasing randmness.

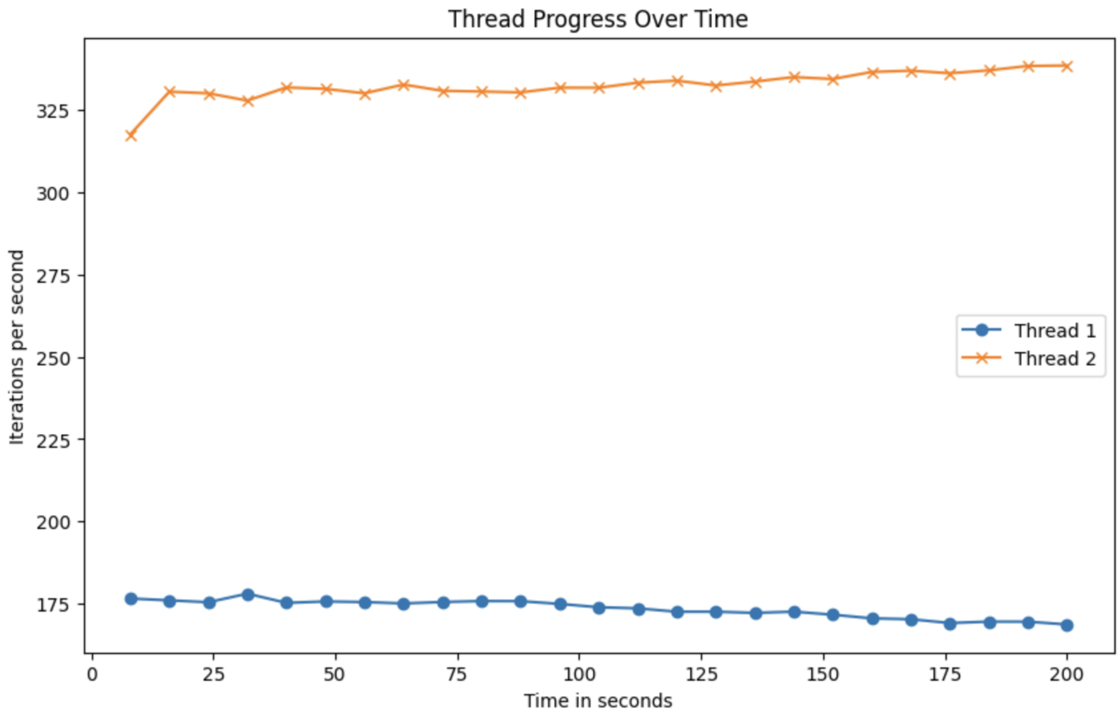

Figure 6: Thread Progress over time. The number of iteration per second executed by two threads in 1:2 ticket allocation oer 200 seconds

Observation 3: A line plot of cumulative iterations/sec over time also showed that Thread 2 consistently completed more iterations than Thread 1. This is same nature of plot discussed by Waldspurger in his paper.

3.1.2 Threads with different ticket allocation ratio

I applied the same concept as Waldspurger ‘s by allocating different ticket ratios (1:1, 1:2, 1:3, 1:4……, 1:10) to two threads and observing how the scheduling influenced their execution in each allocation. The number of iteration each thread executed in each allocation ratio is noted by running the program 60 seconds. The collected data is shown in table 2.

Allocated Ratio	Thread 1 Iterations	Thread 2 Iterations
1	15041	15044
2	10432	19884
3	7574	22862
4	5938	24514
5	5475	24954
6	4406	26029
7	3834	26614
8	3581	26861
9	2881	27545
10	2851	27566

Table 2: Number of iteration executed by threads in different ticket allocation ratio

Observation: From the plot of Figure 7 and 8 we can see that the number of times the threads execute depends on their tickets allocations. The more thickets a threads have more it get chance to execute. This observations also proves the fairness in lottery scheduling

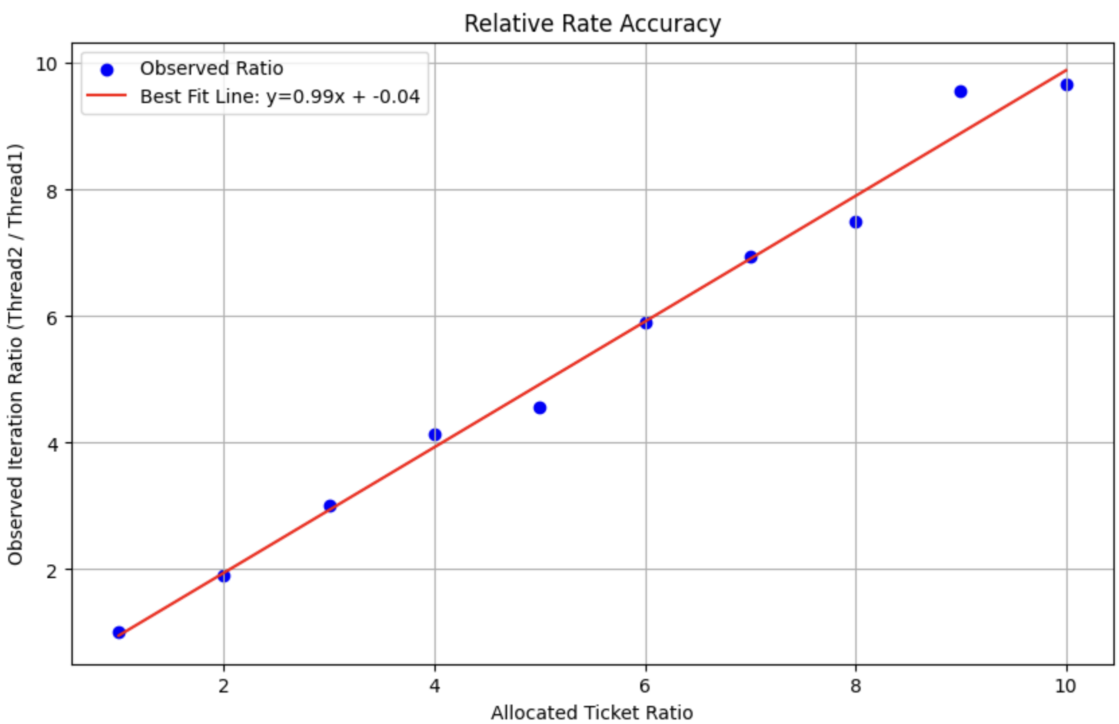

Figure 7: Relative Rate Accuracy. For each allocated ratio, the observed ratio is plotted for each of three 60 second runs. The red line indicate the ideal where the two ratios are identical

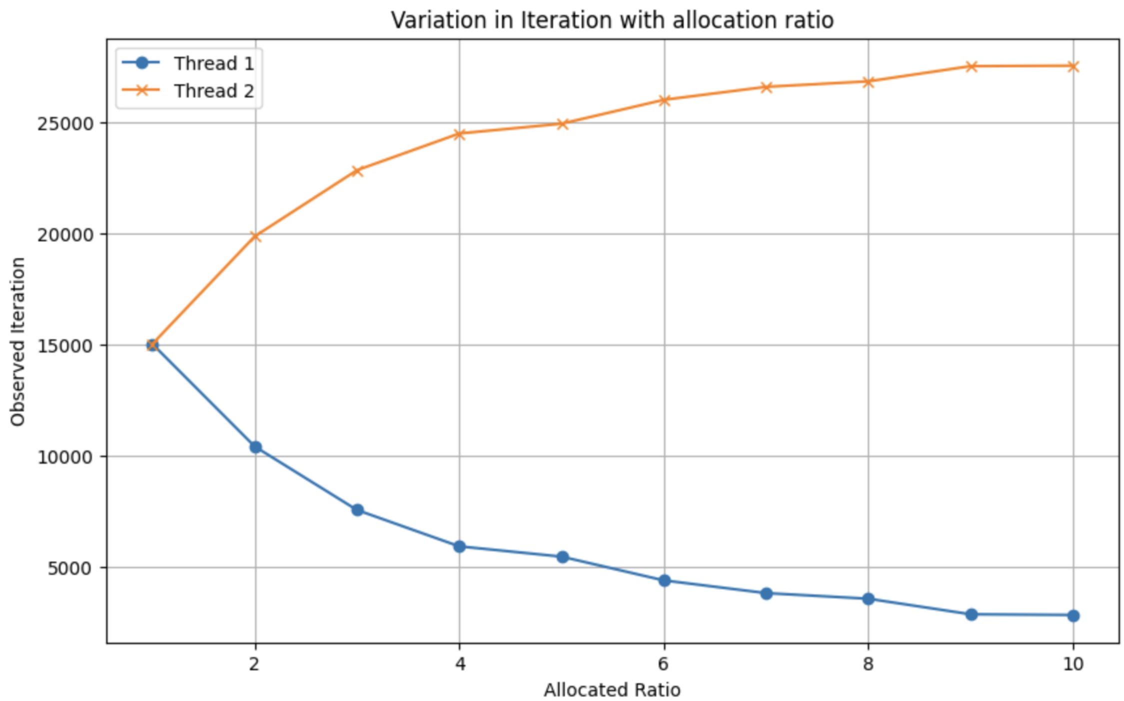

Figure 8: This figure shows both the threads execute 15000 iterations when 1:1 tickets and then the rate at which higher allocated threads increases, with the same rate the the lower allocated thread decreases.

3.2 Round robin vs Lottery scheduling

Two threads were executed by using round robin scheduler and the 25 sample data was collected with the interval of 8 seconds. The data is shown below in Table 3.

Round Robin: In the Round Robin scheduler, each thread is given an equal time slice (quantum) to execute, regardless of its priority or workload. This ensures that all threads get CPU time in a strict sequence, leading to predictable and fair scheduling when all threads have similar workloads. However, it may not be ideal when threads have varying execution times or priorities.

Lottery Scheduler: The Lottery Scheduler provides probabilistic fairness by assigning tickets to each thread. Threads with more tickets have a higher chance of being scheduled. In experiments, threads with higher ticket counts received more CPU time, demonstrating weighted fairness. This allows the scheduler to prioritize certain threads while still giving lower-ticket threads a chance to execute.

Time (sec)	Thread 1 Iterations	Thread 2 Iterations
8	2011	2010
16	4024	4024
24	6080	6079
32	8098	8100
40	10126	10122
48	12163	12153
56	14187	14174
64	16247	16253
72	18264	18252
80	20300	20277
88	22286	22254
96	24436	24404
104	26371	26351
112	28446	28408
120	30424	30416
128	32166	32242
136	32457	32471
144	33929	33846
152	35840	35980
160	37832	37945
168	39515	39664
176	41510	41654
184	43367	43508
192	45466	45578
200	47031	47193

Table 3: The number of iteration executed by threads with different time interval (RR)

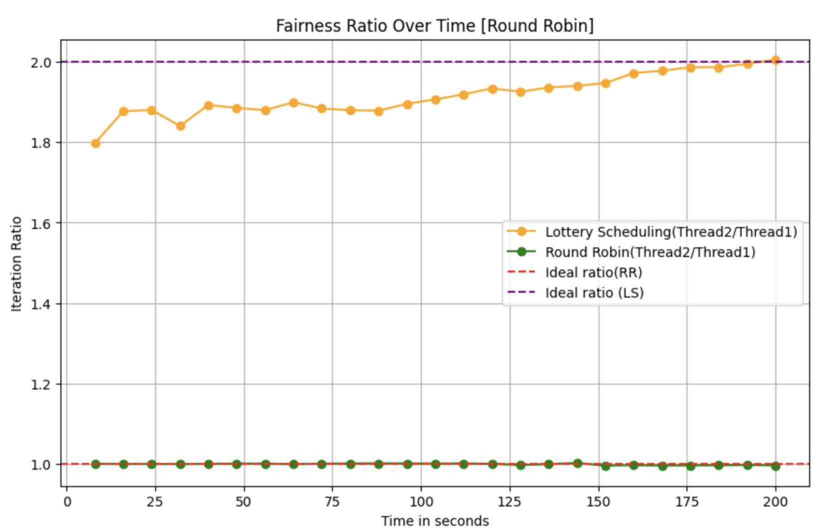

Figure 9: Plot of iteration ratio of two threads in round robin and lottery scheduling

4 Conclusion

The project implements many-to-one user-level threads without the pthreads library. It successfully teaches the creation of preemptively scheduled user-level threads using round-robin and lottery scheduler techniques. It also teaches how to implement context switching between threads manually when the timer generates a signal. Using lottery scheduling, one Waldsperger's experiment is demonstrated and compared with the round-robin scheduler.

5 Appendix

Things i learned from LLM (Deepseek) and Internet:

1. I was using usleep() to delay the execution of threads and using ITIMER_VIRTUAL as a result, my thread my not preempted. I learn that SIGVTALRM does not interrupt usleep() Prompt: i want to use signal virtual alarm to track the execution time of threads but the threads is not preempted?

```c
void setTimer() {
    signal(SIGVTALRM, schedulerHandler);  // For preemption every
       100ms

    struct itimerval virtual_timer;
    virtual_timer.it_value.tv_sec = 0;
    virtual_timer.it_value.tv_usec = TIME_SLICE;
    virtual_timer.it_interval = virtual_timer.it_value;
    setitimer(ITIMER_VIRTUAL, &virtual_timer, NULL);
}

void thread1() {
    while (1) {
        current_thread->iterations++;
        printf("Thread 1: Iteration %d\n", current_thread->
           iterations);
        usleep(5000); // Simulate work
    }
}
```

Listing 1: Timer and Thread Functions in C

Response: Your code intends to use SIGVTALRM (virtual alarm) via setitimer(ITIMER_VIRTUAL, timer, NULL) to trigger preemption of threads in a user-level threading system. However, the problem is that SIGVTALRM is only delivered to the process as a whole, not to individual threads. Here are the main issues preventing preemption:

Issues in the Code SIGVTALRM does not interrupt usleep()

- usleep(50000); (inside thread1, thread2, thread3) blocks execution for 50ms.
- usleep() uses nanosleep(), which is not interrupted by SIGVTALRM in many systems.

- This means your scheduler will not be called until usleep() finishes, preventing preemption.

Solution: Replace usleep() with a loop that checks for preemption f a thread calls usleep(), the process enters kernel mode and the timer pauses, preventing preemption.

2. My threads was not resuming but it was restarting when it was rescheduled. I forget to use swapcontext and used setcontext which results me this error. ChatGPT helped me to debug this error.
   Prompt: My threads are restarting. not resuming. Here is the screenshot.

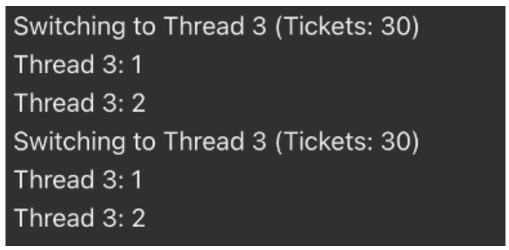

Figure 10: prompt

   Fix:

   - Use swapcontext(current_thread $\rightarrow$ context, next_thread $\rightarrow$ context) instead of setcontext(next_thread $\rightarrow$ context).
   - swapcontext() ensures that the current thread's execution is saved, so when it is scheduled again, it resumes instead of restarting.

3. I wanted to use virtual timer to stop the program execution after 60 seconds and use it but in doing that ITIMER_VIRTUAL was being overwritten when setting the second timer to stop execution after 60 CPU seconds. As a result my thread execution was stuck and was not preempted.

```c
void setup_timer() {
    // Set up signal handlers
    signal(SIGVTALRM, scheduler);
    signal(SIGXCPU, stop_execution);

    // Configure virtual timer for preemption
    struct itimerval virtual_timer;
    virtual_timer.it_value.tv_sec = 0;
    virtual_timer.it_value.tv_usec = TIME_SLICE; // 100 ms time slice
    virtual_timer.it_interval = virtual_timer.it_value;
    setitimer(ITIMER_VIRTUAL, &virtual_timer, NULL);

    // Configure stop timer to halt execution after 60 CPU seconds
    struct itimerval stop_timer;
```

```c
    stop_timer.it_value.tv_sec = 60; // Stop execution after 60 CPU seconds
    stop_timer.it_value.tv_usec = 0;
    stop_timer.it_interval.tv_sec = 0; // No repeat
    stop_timer.it_interval.tv_usec = 0;
    setitimer(ITIMER_VIRTUAL, &stop_timer, NULL);
}
```

Listing 2: Timer Setup Function in C

Your threads are not getting preempted because ITIMER_VIRTUAL is being overwritten when setting the second timer to stop execution after 60 CPU seconds. When you call setitimer(ITIMER_VIRTUAL, stop_timer, NULL), it replaces the first timer responsible for preemption.

What's Wrong? Overwriting the Virtual Timer:

- setitimer(ITIMER_VIRTUAL, stop_timer, NULL) replaces the first timer that was set for preemption, so no more SIGVTALRM signals are sent every 100ms.
- Fix: Use ITIMER_PROF for stopping execution after 60 CPU seconds instead of ITIMER_VIRTUAL. ITIMER_PROF counts both user and system time.

4. Learn about signals, setitimer(), ucontext from UNIX manual

6 Reference

1. Linux manual page, `https://man7.org/linux/man-pages/man2/setitimer.2.html`

2. Assignment resource, `https://www2.it.uu.se/education/course/homepage/os/vt18/module-4/simple-threads/`

3. Waldspurger's Lottery Scheduling: Flexible Proportional-Share Resource Management