Raghav Sakhuja
2021274

# Documentation

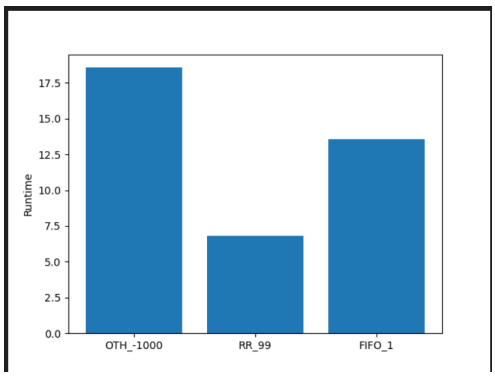
## Q1.1: Simultaneous Execution of Threads

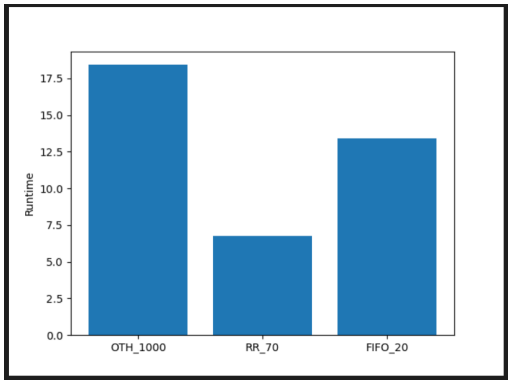
In this we are executing 3 threads at the same time and changing their priorities to check how it affects their execution time and how the scheduler manages them

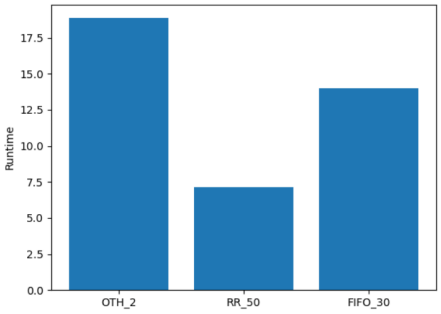
We create 3 threads using the pthread_create() function to create threads, and call different functions CountA, CountB, CountC. These functions count till $2^{32}$

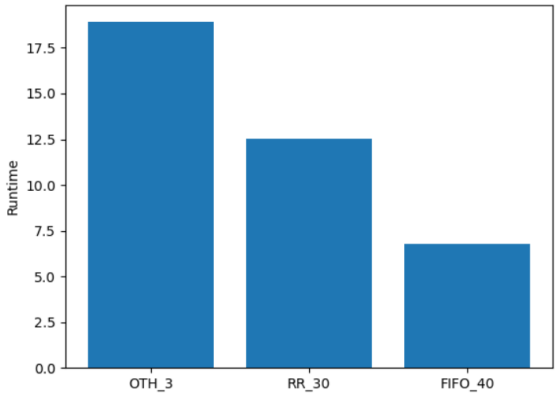
The functions internally change the beginning of the clock and then changes the scheduling policy and the priority of that thread. This is done with the help pthread_setschedparam(). After the counting ends, the clock is stopped and the time for execution is calculated .

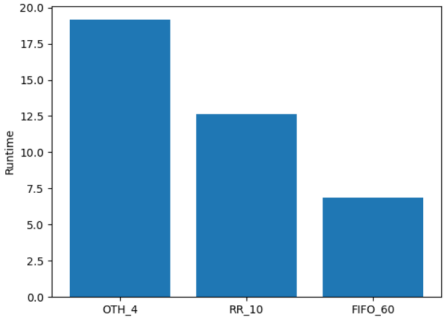
The execution times of the threads are then stored in 3 different text files which are used to create the graphs using the Matplotlib module of python.

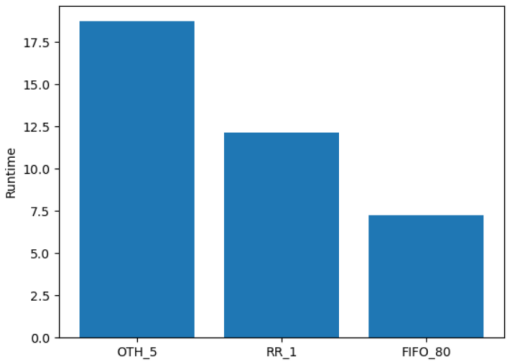

Runtime
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
OTH_-1000
RR_99
FIFO_1

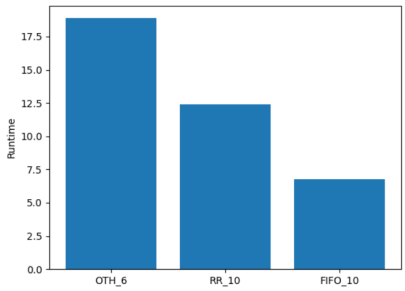

Runtime
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
OTH_1000
RR_70
FIFO_20

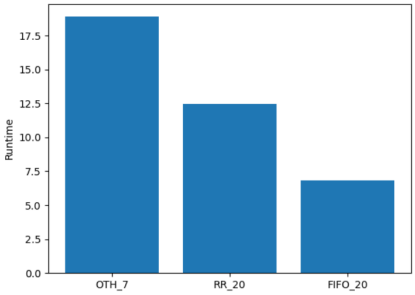

Runtime
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
OTH_2
RR_50
FIFO_30

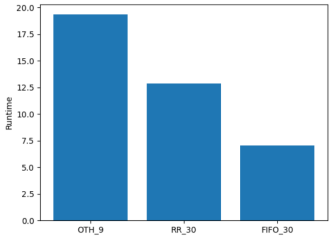

Runtime
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
OTH_3
RR_30
FIFO_40

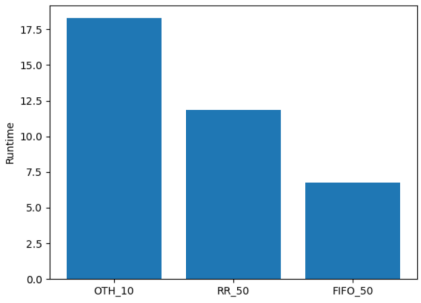

Runtime
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
OTH_4
RR_10
FIFO_60


Runtime
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
OTH_5
RR_1
FIFO_80




With these outputs, we can see how the changing the priorities and policies affects the time.

# Q1.2: Simultaneous Execution of Processes

In this we are executing 3 processes at the same time and changing their priorities to check how it affects their execution time and how the scheduler manages them

We create 3 processes using fork() function to create child processes, and call different functions run1, run2, run3.

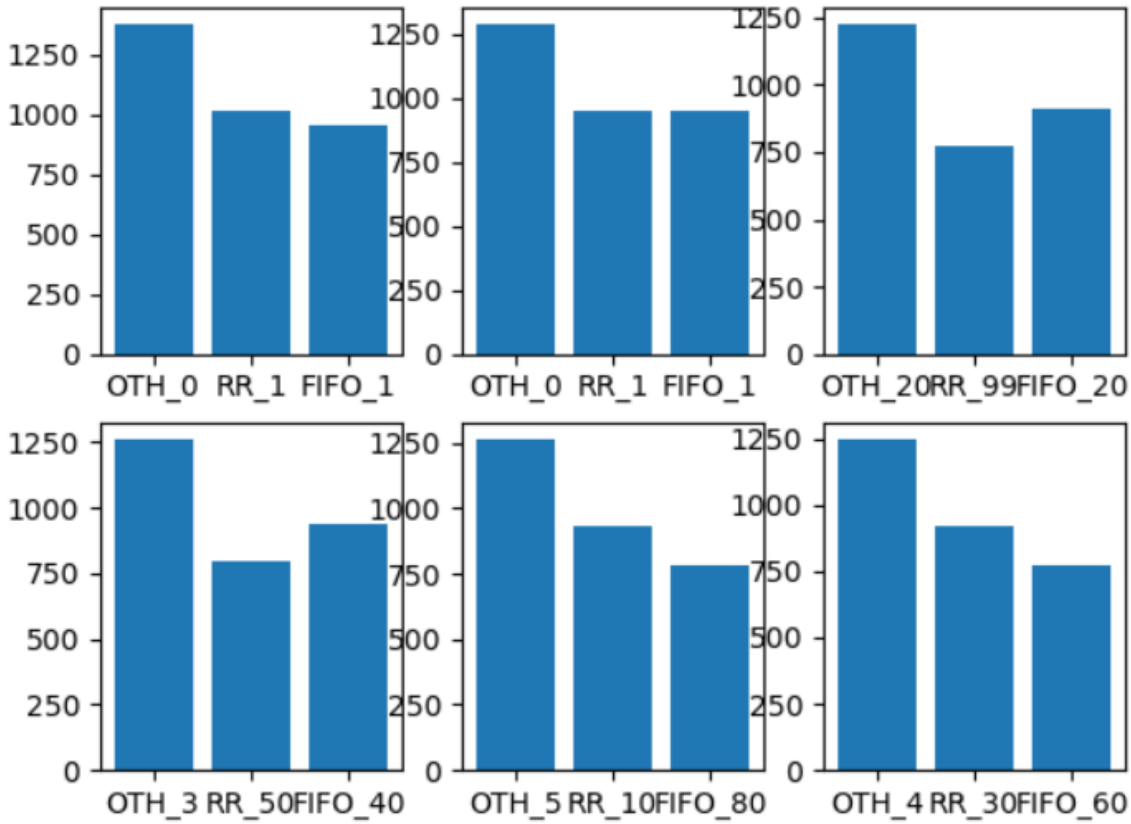
For creation of the children, we call 3 forks in the following form:

```
Int rc1=fork()
if(rc>0){
      //call run//
}
else{
      Int rc2=fork()
      if(rc2>0){
            //Call run//
      }
      else{int rc3=fork()
            if(rc3>0){
                  //call run//
            }
            else{
            }
      }
}
```

After which waitpid() is used to wait for the complete execution of the processes. This is done in order to avoid race conditions and proper simultaneous execution of the processes.

The functions internally call different bash files which are used to traverse different directories and compile kernels.

The execution times of the children processes are then stored in 3 different text files which are used to create the graphs using the Matplotlib module of python.




With these outputs, we can see how the changing the priorities and policies affects the time.

# Q2. Syscall-Implementation

A Syscall to copy a 2D array of floating b=point integers can be implemented using the following methods.

We first add the syscall into the table of already existing syscalls in linux-5.xx.xx/arch/x86/entry/syscalls/syscall_64.tbl

And add: 451 kern_2D_memcpy sys_kern_2D_memcpy

Next we create a new directory in root directory of the kernel and create the C code file to define the syscall

kernel_2d_memcpy_syscall.c

> SYSCALL_DEFINE4(kernel_2d_memcpy, float*, src, float*, dest, int, rows,int ,cols)

The syscall copies the memory from src into dest using the copy_from_user and copy_to_user system calls.

Further we have to configure the kernel using the following commands:

```
make
make modules_install
cp arch/x86_64/boot/bzImage /boot/vmlinuz-2D
mkinitcpio -k 5.19.8 -g /boot/initramfs-2Dimg
grub-mkconfig -o /boot/grub/grub.cfg
reboot
```

## Testing

To test the syscall, we can run the kernel_2dtest.c file in which a 5x5 matrix is copied using the syscall.