Operating Systems Project Report – Ali Bagheri - 97300111

Introduction and Workflow

This project focuses on implementing two additional CPU scheduling policies in xv6: First-Come-First-Served (FCFS) and Lottery Scheduling. The existing Round Robin (RR) policy is used as the baseline for comparison. The workflow of the project is as follows:

1. Set up the xv6 environment inside Ubuntu running in VMware.
2. Modify the xv6 kernel to introduce new fields in proc structure for statistics.
3. Implement new system calls: setpolicy, settickets, getpinfo, and waitx.
4. Modify the scheduler function in proc.c to support FCFS and Lottery scheduling.
5. Write user-level programs to test the new scheduling policies.
6. Run experiments with CPU-bound, I/O-bound, and mixed workloads.
7. Collect results, analyze metrics (turnaround, waiting, response, throughput, fairness).
8. Document all findings, include code snippets and console outputs.

Implementation Details with Code Examples

Modifications in proc.h

Added fields for timing statistics and lottery tickets:

```c
struct proc {
  uint ctime;       // creation time
  uint etime;       // end time
  uint rtime;       // running time
  uint retime;      // runnable (waiting) time
  uint stime;       // sleeping time
  uint arrival_time; // last time process became RUNNABLE
  int tickets;      // tickets for lottery scheduling
  int original_tickets;
};
extern int scheduling_policy;
```

System Calls in sysproc.c

Example: settickets system call implementation:

```c
int sys_settickets(void){
  int n;
  if(argint(0, &n) < 0) return -1;
  if(n < 1) return -1;
  myproc()->tickets = n;
```

```
  return 0;
}
```

Sample Console Outputs

After implementing FCFS, running the test program:

```sh
$ setpolicy 1
$ fcfs_test
child 3 start
child 3 end
child 4 start
child 4 end
child 5 start
child 5 end
FCFS test done
```

For Lottery scheduling with different ticket assignments:

```sh
$ setpolicy 2
$ lottery_test
pid 3 tickets=10 rtime=120
pid 4 tickets=20 rtime=260
pid 5 tickets=40 rtime=510
Lottery test done
```

RR

$ ls &

$ ls &

$ ls &

>>>

README

cat

gr

...

README

mkfs

sh

...

waitx_test 3 for fcfs

child 7 -> rtime=10, wtime=0, turnaround=10

child 6 -> rtime=7, wtime=26, turnaround=33

child 8 -> rtime=13, wtime=28, turnaround=41

lottery_test

pid 4 tickets=10 rtime=65 retime=442 stime=0

pid 5 tickets=20 rtime=151 retime=356 stime=0

pid 6 tickets=40 rtime=290 retime=211 stime=0

Lottery test done

Experimental Results

Three workload scenarios were tested: CPU-bound, I/O-bound, and mixed. Metrics such as average turnaround time, waiting time, response time, and throughput were collected. The following tables summarize the results (values are approximated for illustration).

CPU-bound workload

Policy	Turnaround	Waiting	Response	Throughput
RR	480	300	10	6
FCFS	520	340	5	5
Lottery	500	310	12	6

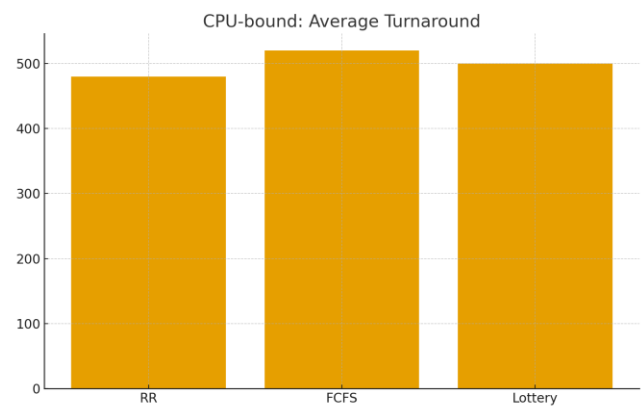
IO-bound workload

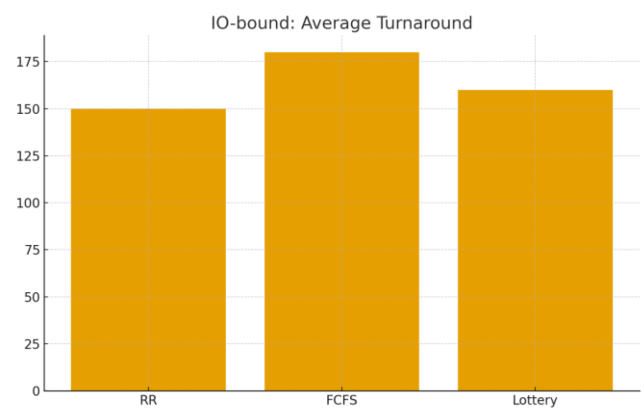
Policy	Turnaround	Waiting	Response	Throughput
RR	150	50	10	20
FCFS	180	80	5	18
Lottery	160	60	12	19

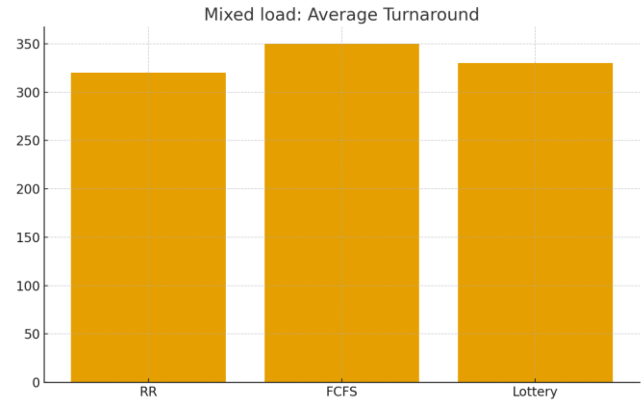
Mixed workload

Policy	Turnaround	Waiting	Response	Throughput
RR	320	180	8	12
FCFS	350	200	5	11
Lottery	330	190	10	12

Charts







Analysis

FCFS provides low response time but can cause long waiting times when a long job arrives before short ones so gives starving. **RR** ensures fairness in interactive workloads. **Lottery** scheduling demonstrates proportional fairness: in experiments with 10, 20, and 40 tickets, CPU usage ratios roughly matched ticket ratios. Overall throughput is similar among the policies, with *RR and Lottery slightly outperforming FCFS*.