REPORT - FINAL PROJECT

Name: Puya Fard (pfard@uci.edu)
Name: Aniruddha Chaki (achaki@uci.edu)
Name: Sauryadeep Pal (sauryadp@uci.edu)
Course: EECS211
Subject: QEMU

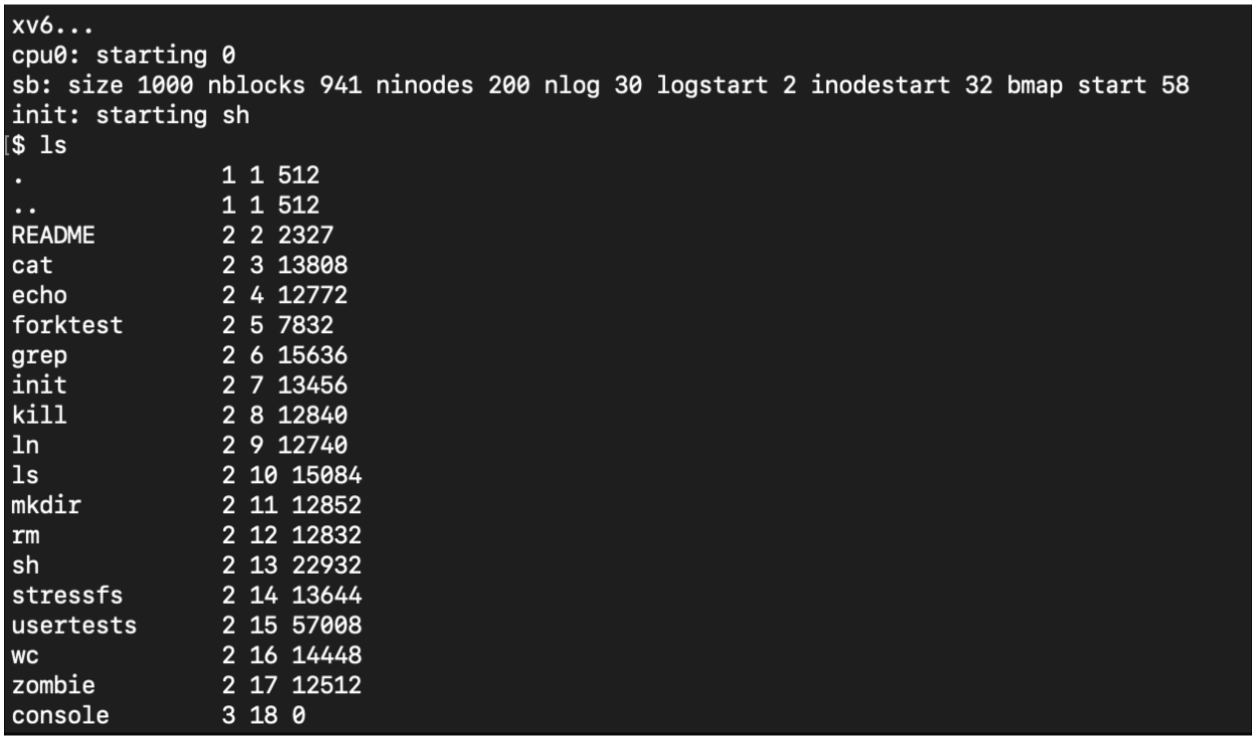
Instructor notes:

Objective

The objective of this final project is to enhance a **tick-based** kernel by implementing a **dynamic tick period** controlled by the kernel itself. The primary aim is to modify the kernel so that the tick period is adaptable rather than fixed. This project necessitates the formulation of a policy governing the adjustment of the tick period based on the behavior of processes. Through this endeavor, we seek to enhance kernel **performance** and **responsiveness**.

Background

The xv6 kernel, tailored for the RISC-V 64-bit architecture, serves as the foundational framework for this project. Derived from the Unix Version 6 (V6) operating system, xv6 is designed to be simple yet comprehensive, making it an ideal platform for educational purposes and kernel hacking.



```
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
[$ ls
.              1 1 512
..             1 1 512
README         2 2 2327
cat            2 3 13808
echo           2 4 12772
forktest       2 5 7832
grep           2 6 15636
init           2 7 13456
kill           2 8 12840
ln             2 9 12740
ls             2 10 15084
mkdir          2 11 12852
rm             2 12 12832
sh             2 13 22932
stressfs       2 14 13644
usertests      2 15 57008
wc             2 16 14448
zombie         2 17 12512
console        3 18 0
```

Figure 1: The xv6 Kernel

Experiment

We know that our timer interrupt is set high from the scheduler, which is located in the proc.c file. The default timer interval is set to a constant of 1,000,000. We want to make this dynamic depending on the number of ready (queued) processors in our program. How do we approach this problem? First, we need to identify the lines of code which the scheduler takes. In this case in the proc.c file starting line 445 our scheduler takes place. We want to set up a logic loop that will dynamically change the timer interval whenever the number of threads ready and running changes. Therefore, we implement the following code:

```c
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();

  c->proc = 0;
  int prev_int = SYSINTERVAL(0);
  int curr_int = SYSINTERVAL(0);
  for(;;){
    // Avoid deadlock by ensuring that devices can interrupt.
    intr_on();

    int n_proc = 0;

    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        n_proc++;
        // Switch to chosen process.  It is the process's job
        // to release its lock and then reacquire it
        // before jumping back to us.
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);

        // Process is done running for now.
        // It should have changed its p->state before coming back.
        c->proc = 0;
      }
      release(&p->lock);
      if (n_proc <= 1) curr_int = 2000000;
      else if (n_proc > 1 && n_proc < 4) curr_int = 1000000;
      else curr_int = 500000;
      if (curr_int != prev_int) {
        UpdateInterval(curr_int);
        // printf("Ready processes: %d, interval changed: %d -> %d\n", n_proc,
prev_int, SYSINTERVAL(0));
        prev_int = curr_int;
      }
    }
  }
}
```

Code 1: Modified scheduler( )

In the scheduler given to us, we will modify it as it is shown above. We add a counter for the number of ready and processes (n_proc) in our kernel and adjust the timer interval accordingly:

- If $0 \leq$ n_proc ≤ 1, set interval to 2,000,000
- If $2 \leq$ n_proc ≤ 3, set interval to 1,000,000
- If n_proc ≥ 4, set interval to 500,000

This dynamically changes the time interval depending on the number of ready processes.

The main reason we chose these values for time intervals is because the default was set to 1,000,000 and as a group, we decided that twice this value should be used in the case where only 1 process is ready. This will allow the process to execute longer without being interrupted by the scheduler since there are no other processes ready to be run. We set delay to 1,000,000 for an intermediate number of processes which fall between 2 to 3. We set the delay to 500,000, the shortest, when 4 or more processes are ready.

An optional print statement displays the number of ready processes and delay changes when they happen. This is mostly used for debugging and is commented out when analyzing the scheduler as the I/O overhead from the print statement may affect the analysis.

```
if (curr_int != prev_int) {
        UpdateInterval(curr_int);
           printf("Ready processes: %d, interval changed: %d -> %d\n", n_proc,
prev_int, SYSINTERVAL(0));
        prev_int = curr_int;
      }
```

Code 2: Debug statement to print out ready processes and interval information

Furthermore, to test our design. We implemented a timing measurement protocol to compare the previous static timing vs the dynamic timing algorithm that was implemented. The way it works is that we will measure the ticks taken for the user functions in the original xv6 kernel and compare them one by one to the user functions run after we changed the scheduler to be more dynamic.

```c
#include "kernel/types.h"
#include "user/user.h"

int
main(int argc, char* argv[]) {

    if (argc < 2) {
        fprintf(2, "Usage: profile <prog> <prog_args...> (optional)\n");
        exit(1);
    }

    char* prog = argv[1];

    int n_args = argc - 1;
    char* prog_args[n_args + 1];
    if (argc > 2) {
        prog_args[0] = prog;
        for (int i = 0; i < n_args; i++) prog_args[i + 1] = argv[2 + i];
    }
    prog_args[n_args] = 0;

    int n_runs = 10;

    int total_ticks = 0;
    for (int i = 0; i < n_runs; i++) {
        int start_ticks = uptime();
        int pid = fork();
        if (pid < 0) {
            fprintf(2, "Error while executing %s (fork() failed)\n", prog);
            exit(1);
        } else if (pid == 0) {
            int e_ret = exec(prog, prog_args);
            if (e_ret == -1) {
                fprintf(2, "Error while executing %s (exec() failed)\n", prog);
                exit(1);
            }
        } else {
            wait(0);
            sleep(1); // to get >0 tick readings
            int end_ticks = uptime();
            total_ticks += end_ticks - start_ticks;
        }
    }

    // integer trickery to emulate round()
    int avg_ticks = (total_ticks + n_runs / 2) / n_runs;

    fprintf(2, "\nRun statistics for program \"%s\":\n", prog);
    fprintf(2, "Total ticks after %d runs: %d\n", n_runs, total_ticks);
    fprintf(2, "Average ticks per run: %d\n", avg_ticks);

    exit(0);
}
```

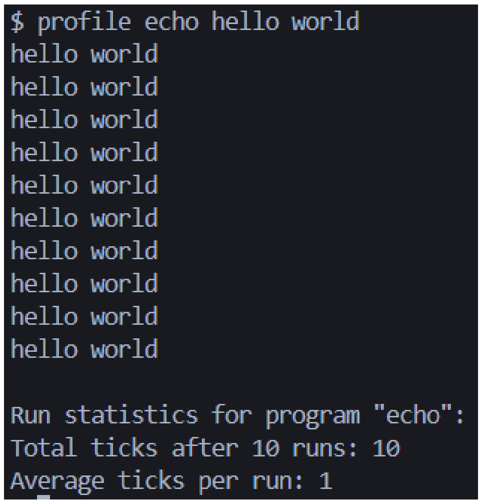
Code 3: profile.c used to time program execution

The code above is a custom C program that will run a desired user program 10 times and calculate the average number of **ticks** per run and the total ticks for all 10 runs. The program takes in another program name and optional program arguments and executes them 10 times via a for loop. We use **exec()** to execute the program, however, we fork first and run exec() through a child process as exec doesn't return to its caller.

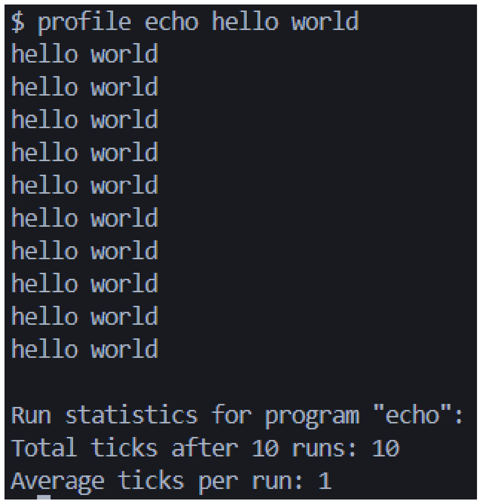
The ticks are recorded (via the **uptime()** function) before and after each execution inside the for loop, and their difference is added to the total ticks. The reason we chose ticks over **SYSTIME** is because the latter seems to return 0 every time it's called. It should be noted that we also had to add a 1 tick delay of **sleep(1)** after each exec() call to complete our analysis. This is because our test computer is capable of executing some of the user programs (such as forktest) 10 times before even a single tick increment is reported by uptime().

Analysis

We analyze our dynamic scheduler program by comparing it to the original static time interval scheduler. Our results will be shown side by side, the right side being the dynamic and, left side being the static.



```
$ profile echo hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world

Run statistics for program "echo":
Total ticks after 10 runs: 10
Average ticks per run: 1
```

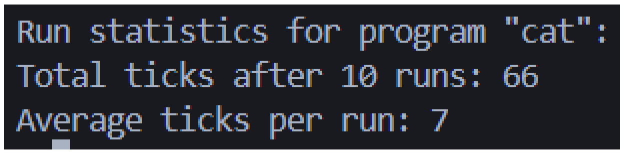


```
$ profile echo hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world

Run statistics for program "echo":
Total ticks after 10 runs: 10
Average ticks per run: 1
```

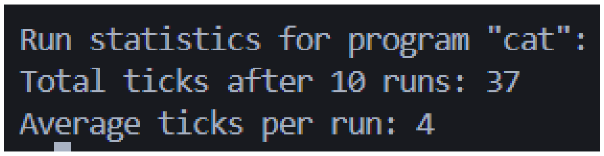
Test 1: echo (static)

Test 1: echo (dynamic)

We can see that the average ticks for echo with dynamic and static timing interval scheduler is both 1.



```
Run statistics for program "cat":
Total ticks after 10 runs: 66
Average ticks per run: 7
```

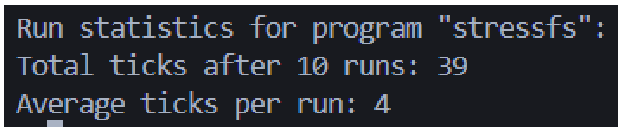


```
Run statistics for program "cat":
Total ticks after 10 runs: 37
Average ticks per run: 4
```

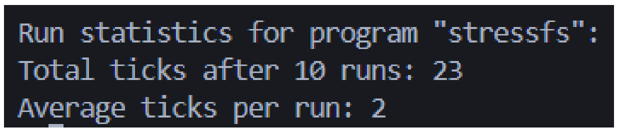
Test 2: cat (static)

Test 2: cat (dynamic)

The actual file being read by the cat is README.MD and we can observe that the average ticks per run for this process are 4 with the dynamic time interval scheduler. However, the original static interval time scheduler runs slower by 7 ticks, which is almost twice as slow.



```
Run statistics for program "stressfs":
Total ticks after 10 runs: 39
Average ticks per run: 4
```

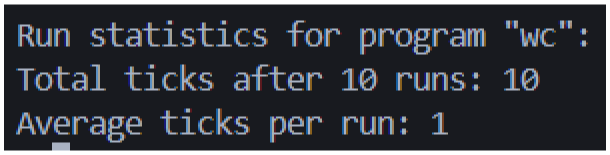


```
Run statistics for program "stressfs":
Total ticks after 10 runs: 23
Average ticks per run: 2
```

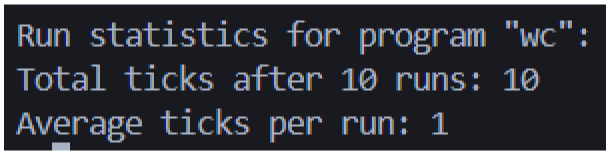
Test 3: stressfs (static)

Test 3: stressfs (dynamic)

Stressfs requires 4 ticks per run on average with the static timing interval scheduler. Whereas with the dynamic scheduler, it needs only 2 ticks per run. This is a 2x improvement in execution speed.



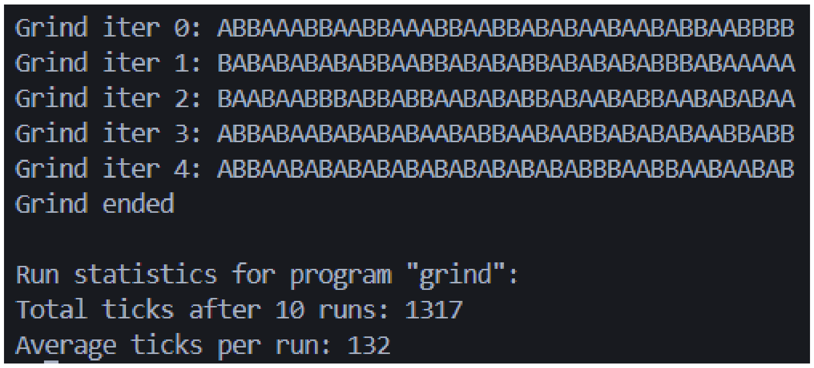
```
Run statistics for program "wc":
Total ticks after 10 runs: 10
Average ticks per run: 1
```



```
Run statistics for program "wc":
Total ticks after 10 runs: 10
Average ticks per run: 1
```

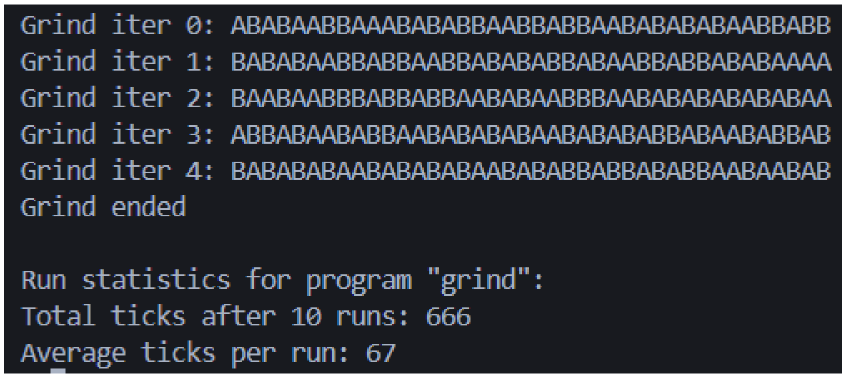
Test 4: wc (static) Test 4: wc (dynamic)

wc runs for average ticks per run of 1 with both static and dynamic time interval schedulers. This is because it completes so fast. The actual file being read by wc is README.MD.



```
Grind iter 0: ABBAAABBAABBAAABBAABBABABAABAABABBAABBBB
Grind iter 1: BABABABABABBAABBABABABBABABABABBBABAAAAA
Grind iter 2: BAABAABBBABBABBAABABABBABAABABBAABABABAA
Grind iter 3: ABBABAABABABABAABABBAABAABBABABABAABBABB
Grind iter 4: ABBAABABABABABABABABABABABBBAABBAABAABAB
Grind ended

Run statistics for program "grind":
Total ticks after 10 runs: 1317
Average ticks per run: 132
```

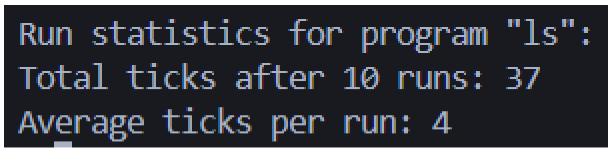
Test 5: grind (static)



```
Grind iter 0: ABABAABBAAABABABBAABBABBAABABABABAABBABB
Grind iter 1: BABABAABBABBAABBABABABBABAABBABBABABAAAA
Grind iter 2: BAABAABBBABBABBAABABAABBBAABABABABABABAA
Grind iter 3: ABBABAABABBAABABABABAABABABABBABAABABBAB
Grind iter 4: BABABABAABABABABAABABABBABBABABBAABAABAB
Grind ended

Run statistics for program "grind":
Total ticks after 10 runs: 666
Average ticks per run: 67
```

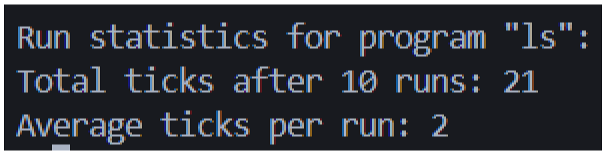
Test 5: grind (dynamic)

We modified grind.c so that it didn't run indefinitely, instead running for 5 iterations. This modified grind required an average of 132 ticks to complete using a static time interval scheduler, whereas it completed under 67 average ticks using the dynamic time interval scheduler. Again we can observe an almost 2x execution time improvement within our dynamic time interval scheduler.



```
Run statistics for program "ls":
Total ticks after 10 runs: 37
Average ticks per run: 4
```

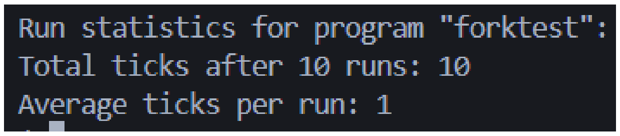
Test 6: ls (static)



```
Run statistics for program "ls":
Total ticks after 10 runs: 21
Average ticks per run: 2
```

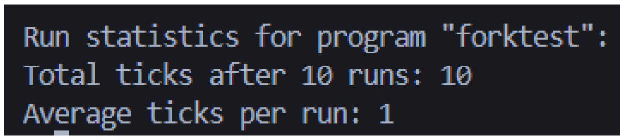
Test 6: ls (dynamic)

For ls, the static interval scheduler takes 4 ticks on average to execute whereas the dynamic interval scheduler is nearly twice as fast, taking 2 ticks on average to execute.



```
Run statistics for program "forktest":
Total ticks after 10 runs: 10
Average ticks per run: 1
```

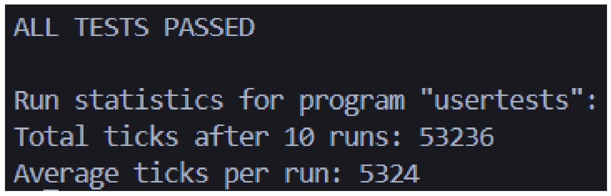
Test 7: forktest (static)



```
Run statistics for program "forktest":
Total ticks after 10 runs: 10
Average ticks per run: 1
```

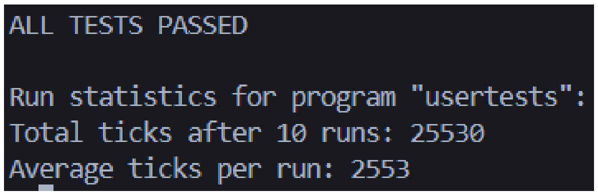
Test 7: forktest (dynamic)

Not much difference can be seen in forktest as both static and dynamic interval schedulers need 1 tick on average to run the program.



```
ALL TESTS PASSED

Run statistics for program "usertests":
Total ticks after 10 runs: 53236
Average ticks per run: 5324
```

Test 8: usertests (static)



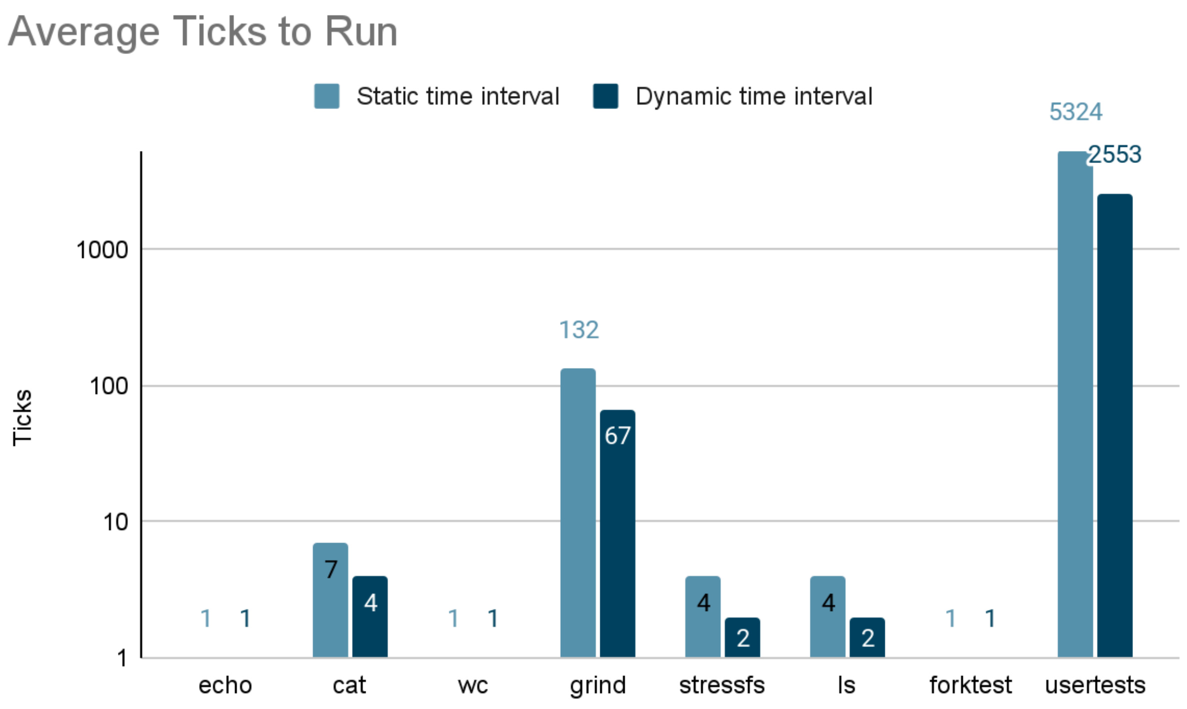
```
ALL TESTS PASSED

Run statistics for program "usertests":
Total ticks after 10 runs: 25530
Average ticks per run: 2553
```

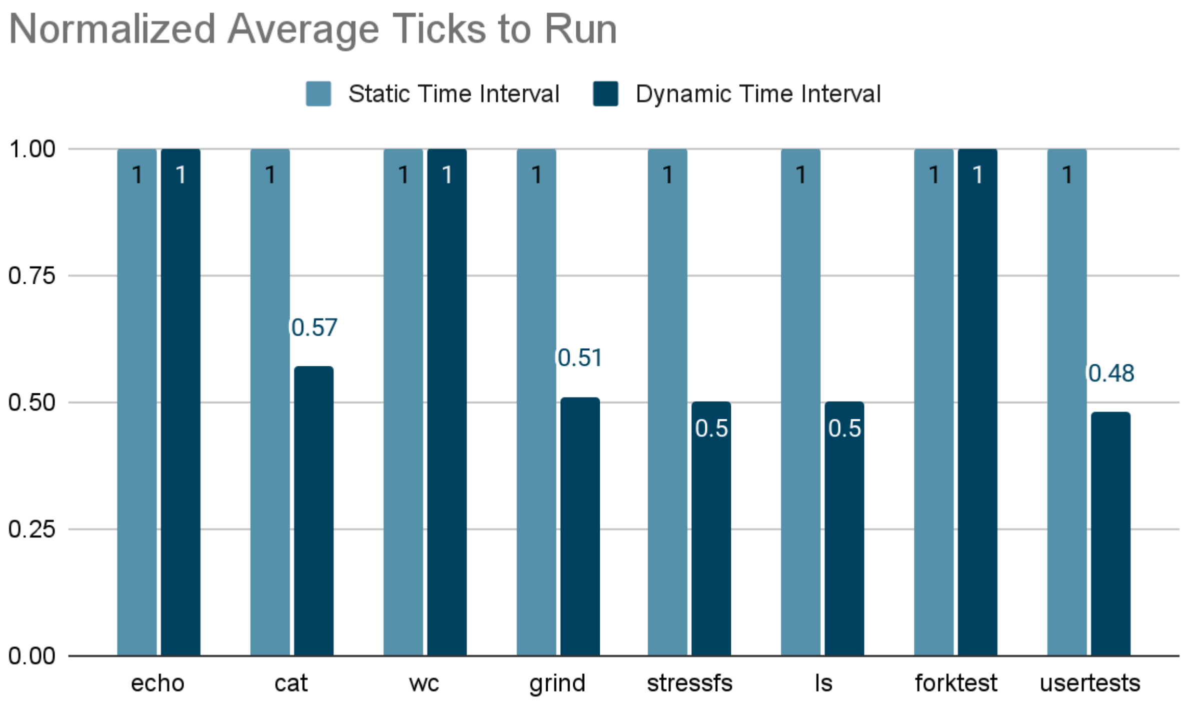
Test 8: usertests (dynamic)

Running usertests, the most time consuming program by far, reveals that the dynamic interval scheduler takes 2553 ticks on average to execute, while the static interval scheduler takes 5324 ticks on average to execute. So with this program, the dynamic interval scheduler is more than twice as fast compared to the static interval scheduler.

Now we calculate the normalized values of our test run outcomes:

Program	min	max	Normalized avg. ticks per run (static)	Normalized avg. ticks per run (dynamic)
echo	0	1	1.0	1.0
cat	0	7	1.0	0.57142857
wc	0	1	1.0	1.0
grind	0	132	1.0	0.50757576
stressfs	0	4	1.0	0.5
ls	0	4	1.0	0.5
forktest	0	1	1.0	1.0
usertests	0	5324	1.0	0.47952667





From the graphs we can observe that while the execution ticks is the same for the programs that can be executed very quickly, such as forktest, for other programs the dynamic time interval scheduler on average takes half as many ticks to execute, so it is 2x faster.

Conclusion

To conclude, we have accomplished 2x execution time improvement with our dynamic time interval scheduler algorithm. We have tested user processes given to us and analyzed the time it takes to complete them using ticks. Could we have done a better job? Yes, we could further improve our dynamic time interval algorithm by making it more precise and efficient by improving adaptivity to the number of ready processes. However, our current results adequately demonstrate the advantage of having a dynamic time interval over a static time interval in the scheduler.