Part 1: Nice System call

To implement the nice system call, we made the following changes:

1. Added a field in **proc.h** struct proc with **int nice** in order to store the nice value.

```
  int nice;                            // Storing nice value
```

2. In **proc.c** allocproc() function, we assigned the initial p->nice value to 20, so that initial processes will be assigned to the lowest priority we have.

```
  p->nice = 20;
```

3. Added a system call in **syscall.c**

```
extern int sys_nice(void);
[SYS_nice]    sys_nice,
```

4. Added a system call constant in **syscall.h**

```
#define SYS_nice   22
```

5. Implemented nice function in **syspro.c**

6. Added SYSCALL(nice) in **usys.S**

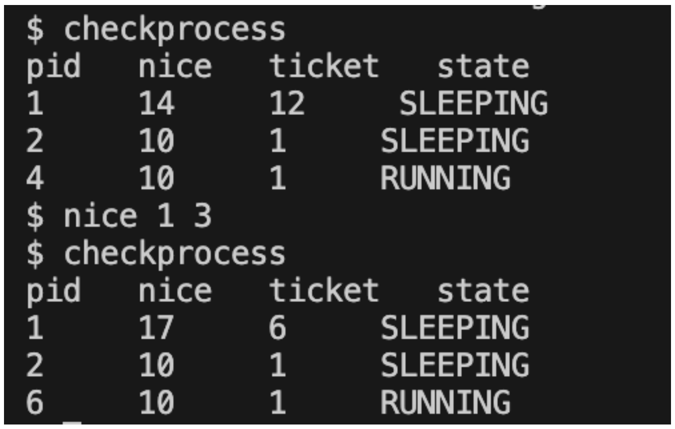
```
SYSCALL(nice)
```

7. Added nice prototype in **user.h**

```
int nice(int, int);
```

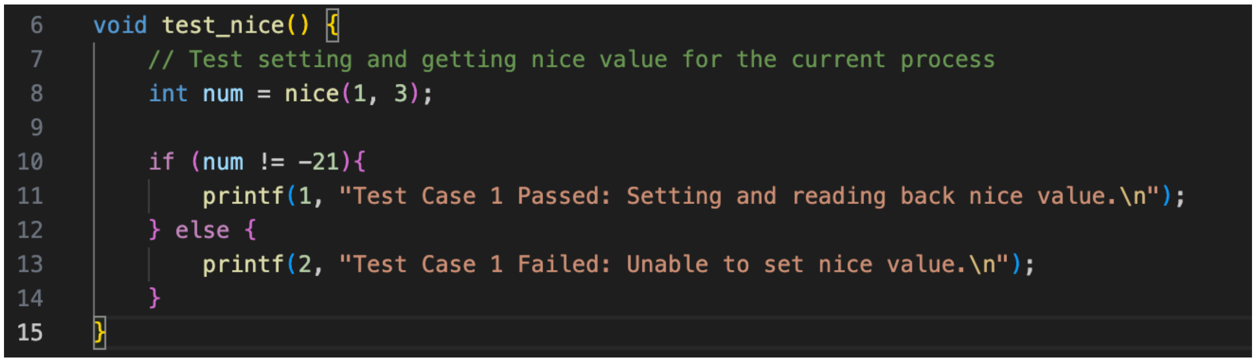
We created **nice.c** to accomplish nice call from the command line. The command has the following usage: **nice <pid> <value>**. The value is incremented/decremented on the initial nice value 20. For example, nice 1 3, would set add 3 to pid = 1's nice value.

To test the functionality of command line nice, we have implemented **checkprocess** system call and in command line, which would print all sleeping, running, runnable, and zombie processes.



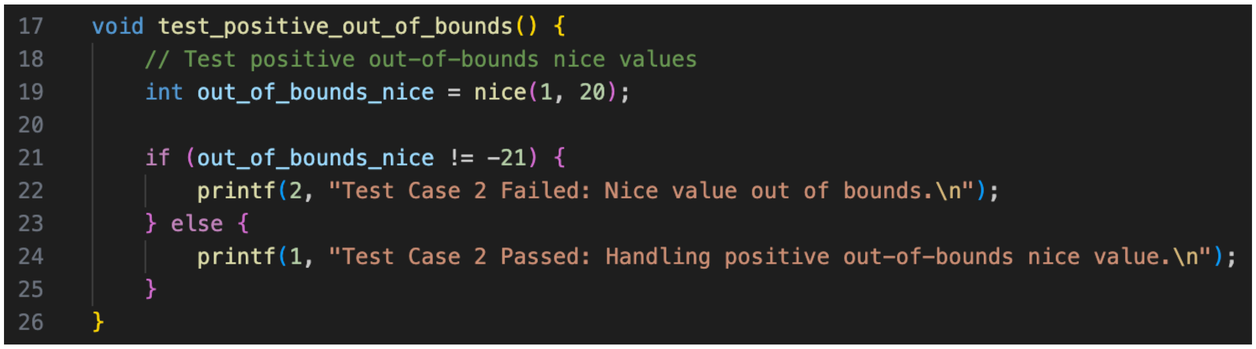
```
$ checkprocess
pid    nice    ticket    state
1      14      12      SLEEPING
2      10      1       SLEEPING
4      10      1       RUNNING
$ nice 1 3
$ checkprocess
pid    nice    ticket    state
1      17      6       SLEEPING
2      10      1       SLEEPING
6      10      1       RUNNING
```

In order to test nice's functionality, we have created **nice1.c** with different test cases.
To run the tests, run command "make", "make qemu-nox", "nice1".
Since we defined nice priority range is -20 to +19, we have included tests for out-of-bound values. The error return value is -21.



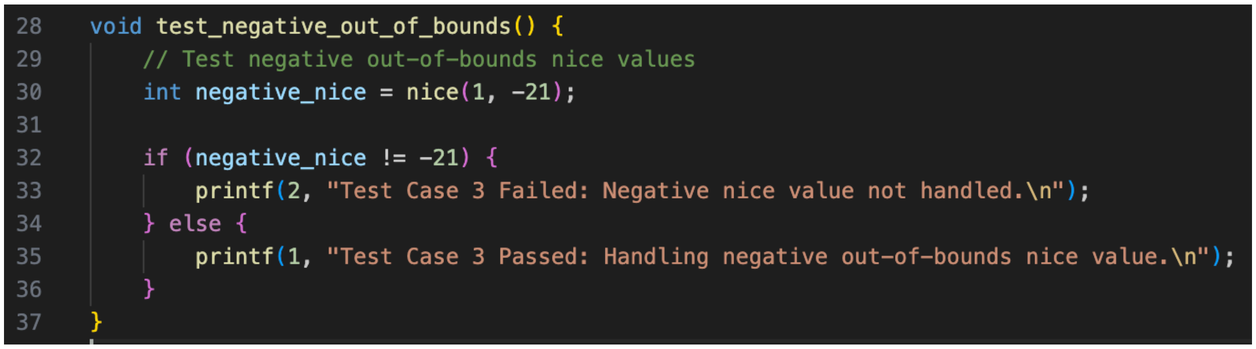
```c
void test_nice() {
    // Test setting and getting nice value for the current process
    int num = nice(1, 3);

    if (num != -21){
        printf(1, "Test Case 1 Passed: Setting and reading back nice value.\n");
    } else {
        printf(2, "Test Case 1 Failed: Unable to set nice value.\n");
    }
}
```

Test 1 for basic nice functionality. The expected return value is 13



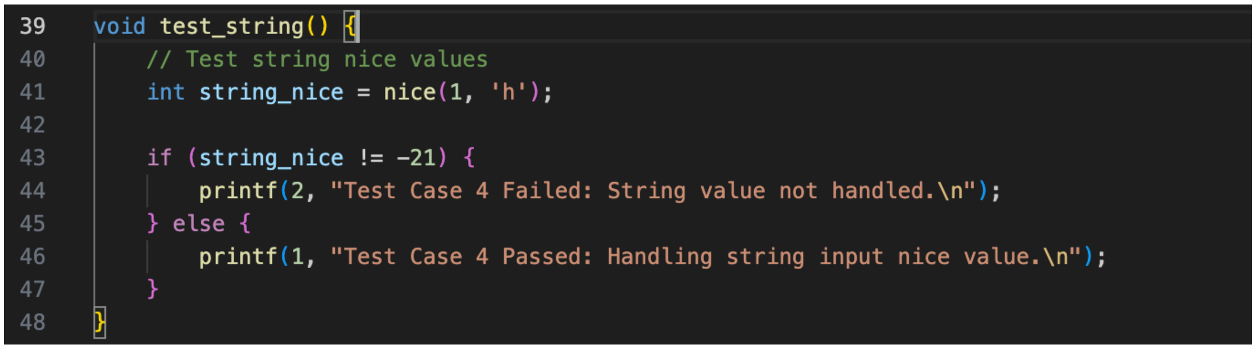
```c
void test_positive_out_of_bounds() {
    // Test positive out-of-bounds nice values
    int out_of_bounds_nice = nice(1, 20);

    if (out_of_bounds_nice != -21) {
        printf(2, "Test Case 2 Failed: Nice value out of bounds.\n");
    } else {
        printf(1, "Test Case 2 Passed: Handling positive out-of-bounds nice value.\n");
    }
}
```

Test 2 for positive out-of-bound nice values. The expected return value is -21

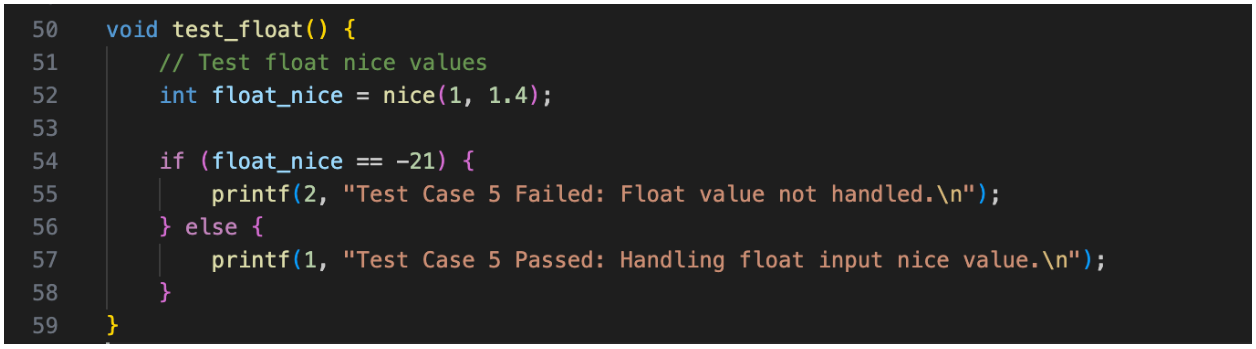


```c
void test_negative_out_of_bounds() {
    // Test negative out-of-bounds nice values
    int negative_nice = nice(1, -21);

    if (negative_nice != -21) {
        printf(2, "Test Case 3 Failed: Negative nice value not handled.\n");
    } else {
        printf(1, "Test Case 3 Passed: Handling negative out-of-bounds nice value.\n");
    }
}
```

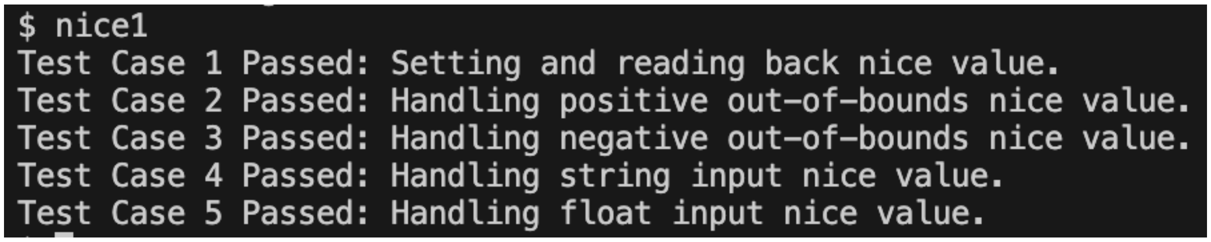
Test 3 for negative out-of-bound nice values. The expected return value is -21


```c
void test_string() {
    // Test string nice values
    int string_nice = nice(1, 'h');

    if (string_nice != -21) {
        printf(2, "Test Case 4 Failed: String value not handled.\n");
    } else {
        printf(1, "Test Case 4 Passed: Handling string input nice value.\n");
    }
}
```

Test 4 for string input values. The expected return value is -21 since strings should not be allowed


```c
void test_float() {
    // Test float nice values
    int float_nice = nice(1, 1.4);

    if (float_nice == -21) {
        printf(2, "Test Case 5 Failed: Float value not handled.\n");
    } else {
        printf(1, "Test Case 5 Passed: Handling float input nice value.\n");
    }
}
```

Test 5 for float nice values. The expected return value is 11 when the input is 1.4, since float will be converted to int and proceed.


```
$ nice1
Test Case 1 Passed: Setting and reading back nice value.
Test Case 2 Passed: Handling positive out-of-bounds nice value.
Test Case 3 Passed: Handling negative out-of-bounds nice value.
Test Case 4 Passed: Handling string input nice value.
Test Case 5 Passed: Handling float input nice value.
```

As expected, these tests have all passed.

Part 2: Random number generator

In order to implement the random number generator, we have made random() into a system call so that it would be easier to test.

We have defined **random()** system call to take 2 parameters with lower and upper bound of the range to pick the random number from. In this case, it eases the pressure from testing and further picking the winning lottery ticket.

1. Added a system call in **syscall.c**

```
extern int sys_random(void);
[SYS_random]   sys_random,
```

2. Added a system call constant in **syscall.h**

```
#define SYS_random 23
```

3. Implemented random system call function in **syspro.c**, which uses random() in **proc.c**

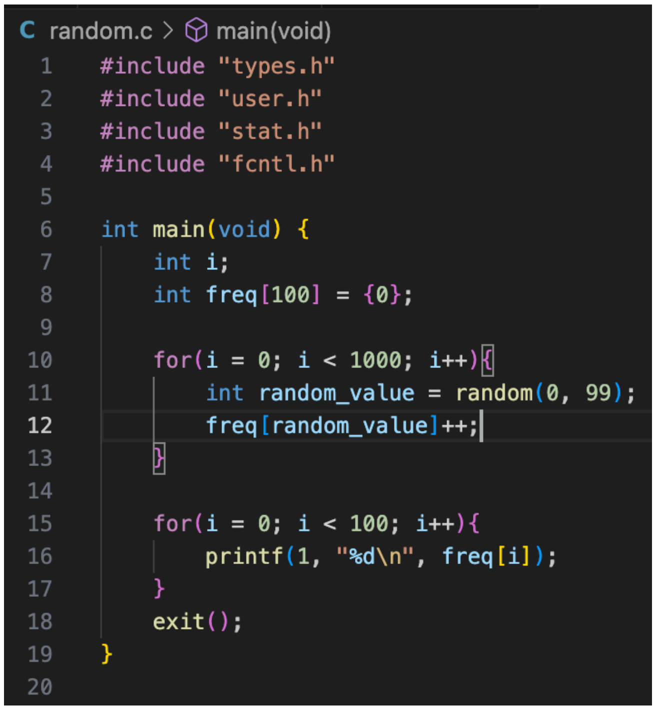
4. Added SYSCALL(random) in **usys.S**

SYSCALL(random)

5. Added nice prototype in **user.h**

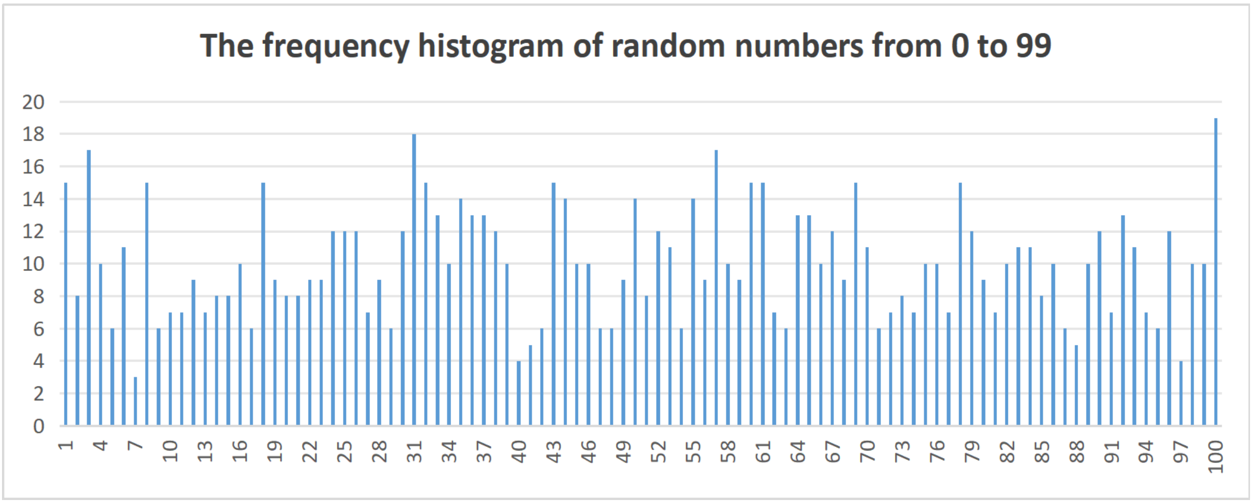
```
int random(int, int);
```

We have made the following test to test the random functionality. We tested to get random numbers within the range of 0 to 99. After running it for 1000 times, the chosen numbers are added as frequencies into a freq[100] array, so that the histogram with the freq[100] output would be more direct.



```
#include "types.h"
#include "user.h"
#include "stat.h"
#include "fcntl.h"

int main(void) {
    int i;
    int freq[100] = {0};

    for(i = 0; i < 1000; i++){
        int random_value = random(0, 99);
        freq[random_value]++;
    }

    for(i = 0; i < 100; i++){
        printf(1, "%d\n", freq[i]);
    }
    exit();
}
```

The output of the test case was then made into a histogram. From the histogram, we can see that the output is not particularly skewed to a direction.



Frequency mean = 10
Frequency max = 19, value = 99
Frequency min = 3, value = 6

Part 3: Lottery scheduling

Before implementing lottery scheduling, we created a setticket() system call, which assigns ticket number to a specific pid. It returns the ticket number (which is between 1 and 100) on success, otherwise it returns -1.

We used a list data structure to hold the assignment from nice value to ticket numbers.

```
int nice_list[40] =
{80, 78, 76, 74, 72, 70, 68, 66, 64, 62,
60, 58, 56, 54, 52, 50, 48, 46, 44, 42,
40, 38, 36, 34, 32, 30, 28, 26, 24, 22,
20, 18, 16, 14, 12, 10, 8, 6, 4, 2};
```

To implement lottery scheduling, we used the **random()** function that was implemented in part 2 to randomly draw a winner. The process table keeps track of processes. Every time we are adding up the tickets from the running processes and drawing a winner. After that, we subtract process's ticket from the winning ticket number. If the winning ticket is still bigger than 0, it means the process isn't the winner. If the process is the runner, we run it.

To run the tests using lottery scheduler:
"make clean"
"make qemu-nox SCHEDPOLICY=LOTTERY"
"<test case file>"
Ctrl-A and X to stop the test from running and view the result.

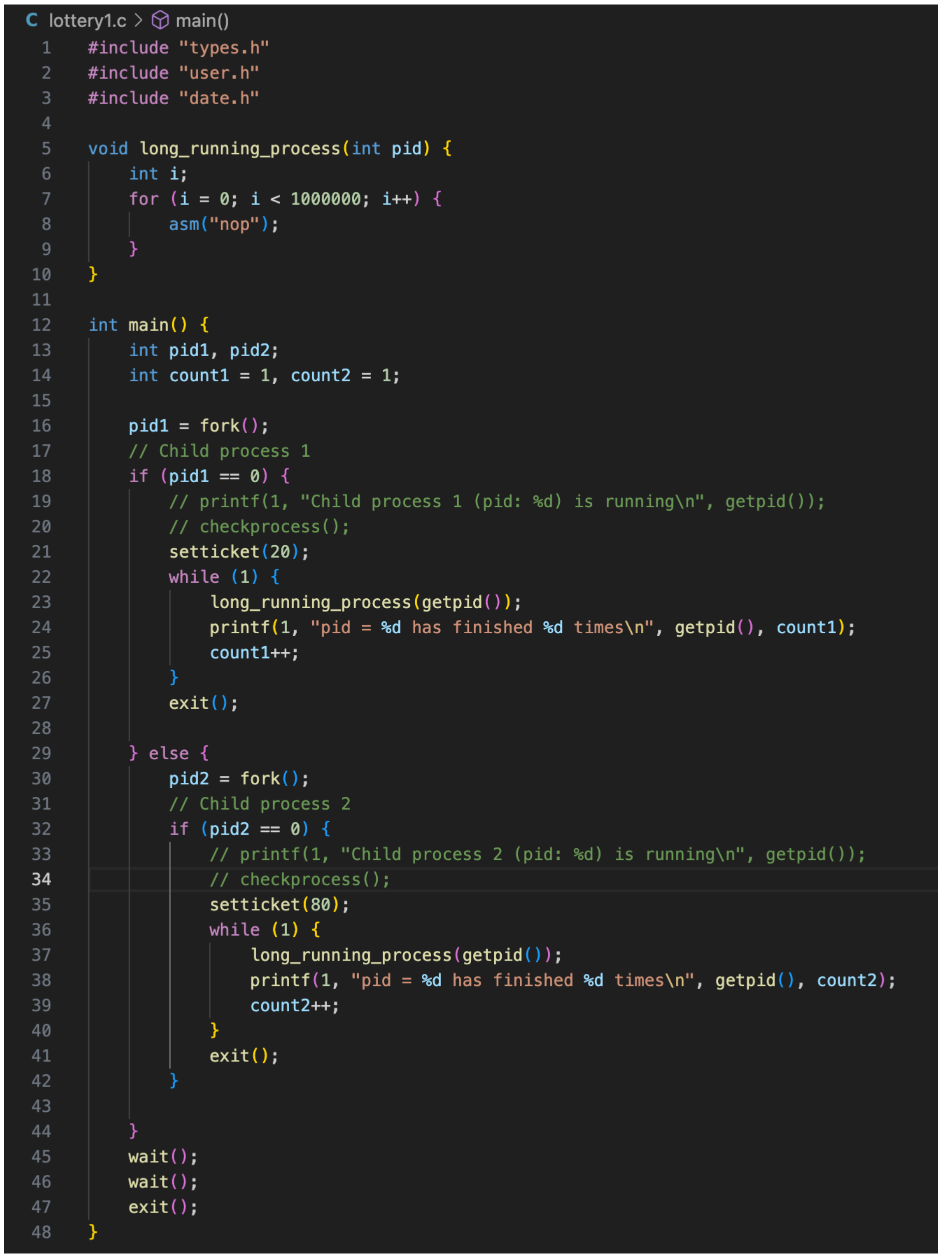
To run the tests using round robin scheduler:
"make clean"
"make qemu-nox" or "make qemu-nox SCHEDPOLICY=DEFAULT"
"<test case file>"
Ctrl-A and X to stop the test from running and view the result.

We used 3 test cases (**lottery1.c**, **lottery2.c**, **lottery3.c**) to test the lottery scheduling.

In **lottery1.c**, we assigned one process pid = 4 with 20 tickets, and another pid = 5 with 80 tickets. Our expectation is that the process that is assigned with more tickets should get drawn more often in the lottery and get to run more.

One instance of the result is:
pid = 4 has finished 752 times
pid = 5 has finished 2848 times

It matches our expectation that pid = 4 gets to run less times than pid = 5. The ratio is roughly 1:4.



lottery1.c > main()

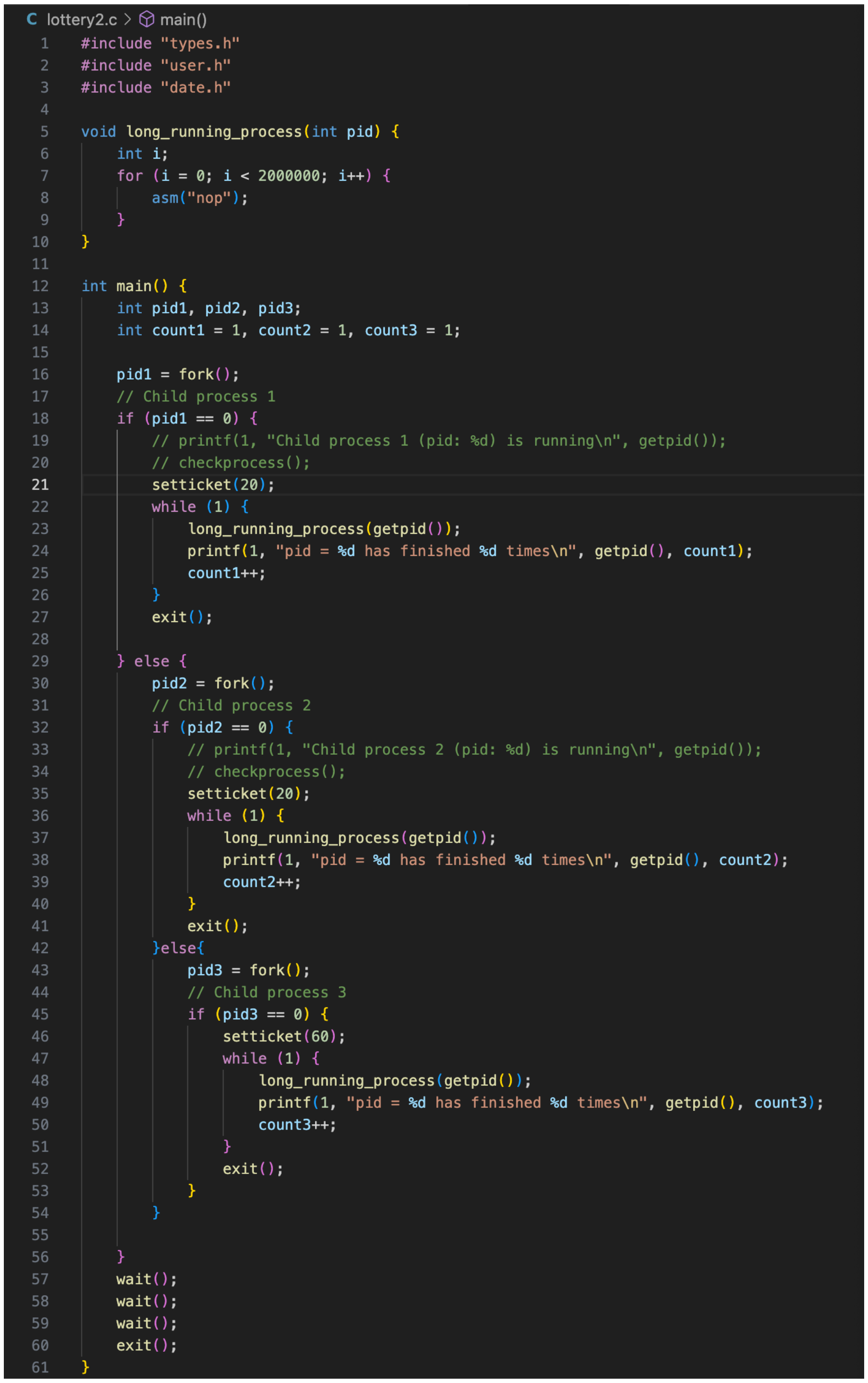
```c
#include "types.h"
#include "user.h"
#include "date.h"

void long_running_process(int pid) {
    int i;
    for (i = 0; i < 1000000; i++) {
        asm("nop");
    }
}

int main() {
    int pid1, pid2;
    int count1 = 1, count2 = 1;

    pid1 = fork();
    // Child process 1
    if (pid1 == 0) {
        // printf(1, "Child process 1 (pid: %d) is running\n", getpid());
        // checkprocess();
        setticket(20);
        while (1) {
            long_running_process(getpid());
            printf(1, "pid = %d has finished %d times\n", getpid(), count1);
            count1++;
        }
        exit();

    } else {
        pid2 = fork();
        // Child process 2
        if (pid2 == 0) {
            // printf(1, "Child process 2 (pid: %d) is running\n", getpid());
            // checkprocess();
            setticket(80);
            while (1) {
                long_running_process(getpid());
                printf(1, "pid = %d has finished %d times\n", getpid(), count2);
                count2++;
            }
            exit();
        }

    }
    wait();
    wait();
    exit();
}
```

In **lottery2.c**, we are having three processes.
We assign pid = 4 with 20 tickets, pid = 5 with 20 tickets, pid = 6 with 60 tickets.

We expect that three processes should get a ratio of 1:1:3

One instance of output:
pid = 4 has finished 700 times
pid = 6 has finished 1837 times
pid = 5 has finished 633 times

The result has the ratio we are expecting.


lottery2.c > main()

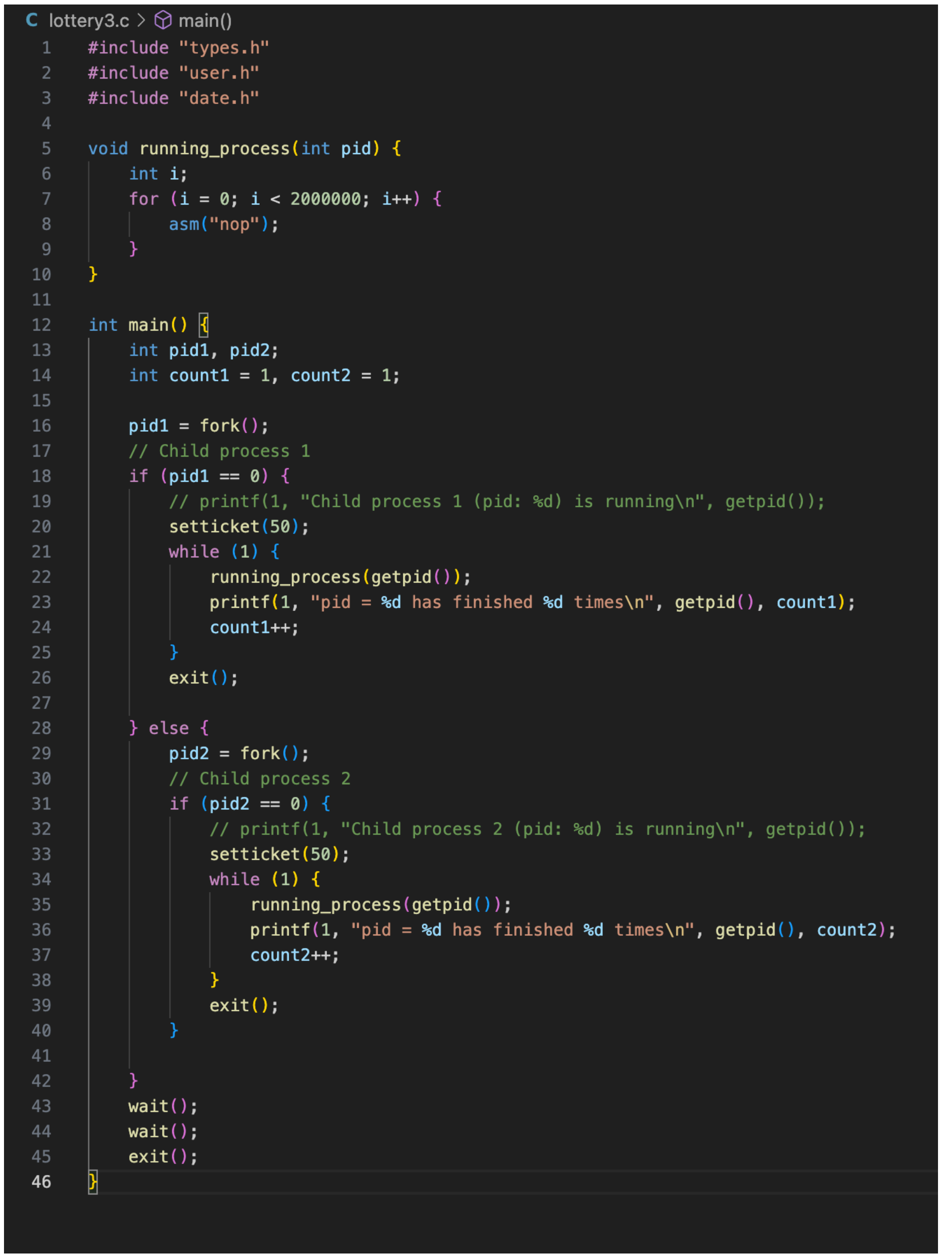
```c
#include "types.h"
#include "user.h"
#include "date.h"

void long_running_process(int pid) {
    int i;
    for (i = 0; i < 2000000; i++) {
        asm("nop");
    }
}

int main() {
    int pid1, pid2, pid3;
    int count1 = 1, count2 = 1, count3 = 1;

    pid1 = fork();
    // Child process 1
    if (pid1 == 0) {
        // printf(1, "Child process 1 (pid: %d) is running\n", getpid());
        // checkprocess();
        setticket(20);
        while (1) {
            long_running_process(getpid());
            printf(1, "pid = %d has finished %d times\n", getpid(), count1);
            count1++;
        }
        exit();

    } else {
        pid2 = fork();
        // Child process 2
        if (pid2 == 0) {
            // printf(1, "Child process 2 (pid: %d) is running\n", getpid());
            // checkprocess();
            setticket(20);
            while (1) {
                long_running_process(getpid());
                printf(1, "pid = %d has finished %d times\n", getpid(), count2);
                count2++;
            }
            exit();
        }else{
            pid3 = fork();
            // Child process 3
            if (pid3 == 0) {
                setticket(60);
                while (1) {
                    long_running_process(getpid());
                    printf(1, "pid = %d has finished %d times\n", getpid(), count3);
                    count3++;
                }
                exit();
            }
        }

    }
    wait();
    wait();
    wait();
    exit();
}
```

In **lottery3.c**, we assign pid = 4 with 50 tickets, pid = 5 with 50 tickets.

We expected the two processes should get roughly equal amount of chances to run.

One instance of output:
pid = 4 has finished 1150 times
pid = 5 has finished 1005 times

It clearly matches our expectation that both processes get to run for around the same amount time.


C lottery3.c > main()

```c
#include "types.h"
#include "user.h"
#include "date.h"

void running_process(int pid) {
    int i;
    for (i = 0; i < 2000000; i++) {
        asm("nop");
    }
}

int main() {
    int pid1, pid2;
    int count1 = 1, count2 = 1;

    pid1 = fork();
    // Child process 1
    if (pid1 == 0) {
        // printf(1, "Child process 1 (pid: %d) is running\n", getpid());
        setticket(50);
        while (1) {
            running_process(getpid());
            printf(1, "pid = %d has finished %d times\n", getpid(), count1);
            count1++;
        }
        exit();

    } else {
        pid2 = fork();
        // Child process 2
        if (pid2 == 0) {
            // printf(1, "Child process 2 (pid: %d) is running\n", getpid());
            setticket(50);
            while (1) {
                running_process(getpid());
                printf(1, "pid = %d has finished %d times\n", getpid(), count2);
                count2++;
            }
            exit();
        }

    }
    wait();
    wait();
    exit();
}
```