xv6 Lab: Implementing a Process Statistics Syscall

CSL301 - Operating System

Class Assignment 3

Objective: Create a Process Statistics Syscall

Your Task

You will implement a new system call, `getstats()`, in xv6. This syscall will allow a user program to retrieve statistics about the current process, such as how many times it has been scheduled and how many timer ticks it has been running.

Why is this important?

This functionality is fundamental for performance monitoring and profiling tools. It gives insight into how the operating system's scheduler is behaving and how resources are allocated to different processes.

Note: We will build on this syscall in future tasks, so it's important to implement it carefully.

Files You Will Modify

Kernel Source

- `proc.h`
- `proc.c`
- `trap.c`
- `syscall.h`
- `syscall.c`
- `sysproc.c`

User-space Interface

- `user.h`
- `usys.S`

Testing & Building

- `statstest.c` (create this)
- `Makefile`

Task 1: Update `proc.h` — Add Statistics Fields

Your Task

Your first step is to modify the core process structure to store the new information you want to track. Add two new integer fields to `struct proc`.

```
// In proc.h, inside struct proc:
struct proc {
  // ... existing fields ...
  // YOUR CODE HERE:
  // 1. Add an integer field to count how many times
  //    this process has been scheduled.  (e.g., sched_count;)

  // 2. Add an integer field to count how many timer
  //    ticks this process has been running.    (e.g., run_ticks;)
};
```

Hint

You are adding new members to a C struct. The names can be anything, but choose descriptive ones!

Task 2: Initialize Counters in `proc.c`

Your Task

A process's statistics must start at zero. The ideal place to initialize these counters is when a new process structure is first allocated.

```
1     // In proc.c, inside the allocproc() function
2   {
3     // YOUR CODE HERE:
4     // Initialize your new counter fields to 0 for the process 'p'.
5     // e.g., p->sched_count = 0;
6     return p;
7   }
```

Task 3: Update Scheduler in `proc.c`

Your Task

The `sched_count` field should be incremented every time the scheduler chooses a process to run. This is the logical place to count this specific event.

```
1     // In proc.c, inside the scheduler() function's main loop:
2     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
3       if(p->state != RUNNABLE)
4       continue;
5       // YOUR CODE HERE:
6       // This is the exact moment a process 'p' has been
7       // CHOSEN to run. Increment its `sched_count` field
8       // right before the context switch (swtch).
9       swtch(&(c->scheduler), p->context);
10      // ...
11    }
```

Task 4: Update Timer Interrupt in `trap.c`

Why?

The timer interrupt fires on every CPU tick. By checking if the current process is in the `RUNNING` state during the interrupt, you can accurately count how many ticks it has actively used the CPU.

```
1       // In the trap() function, inside the interrupt handler
2       case T_IRQ0 + IRQ_TIMER:
3       // ...
4       // YOUR CODE HERE:
5       // If there is a current process and its state is RUNNING,
6       //i.e., myproc () -> state == RUNNING
7       // increment its run_ticks counter.
8       // ...
```

Task 5: Define and Map the Syscall

Why?

Assign a unique number to your new syscall in `syscall.h`. Then, map this number to its handler function in `syscall.c`, so the kernel knows what code to execute when the syscall is invoked.

```
1       // In syscall.h - Choose an unused number (e.g., 25)
2       #define SYS_getstats 25
3
4       // In syscall.c - Add your handler to the table
5       extern int sys_getstats(void);
6       ...
7       [SYS_getstats]   sys_getstats,
```

Task 6: Implement Syscall Handler in `sysproc.c`

Your Task

Implement the kernel logic for `sys_getstats`. This function will copy the statistics from the current process's kernel structure to a user-provided structure.

```
1   int
2   sys_getstats(void)
3   {
4     int *user_stats_ptr;
5
6     if(argptr(0, (void*)&user_stats_ptr, 2 * sizeof(int)) < 0)
7     return -1;
8     struct proc *p = myproc();
9     int kernel_stats[2];
10    kernel_stats[0] = //counts
11    kernel_stats[1] = //ticks;
12
13    if(copyout(p->pgdir, (uint)user_stats_ptr, (char*)kernel_stats,
          sizeof(kernel_stats)) < 0)
14    return -1;
15    return 0; // Success
16  }
```

Task 6: Understanding the Code

1. Fetching the User Pointer with `argptr`

```
if(argptr(0, (void*)&user_stats_ptr, ...))
```

- The kernel cannot directly trust pointers from user space.
- `argptr` safely retrieves the pointer argument from the user stack and validates that the memory it points to is part of the user process's address space.

2. Preparing Data in a Kernel Buffer

```
int kernel_stats[2];
```

- We create a temporary array *inside the kernel*. This is a safe space where we can prepare the data we want to send to the user.
- We populate it with the statistics from the current process: `p->sched_count` and `p->run_ticks`.

Task 6: Understanding the Code

3. Securely Copying with copyout

```
if(copyout(p->pgdir, (uint)user_stats_ptr, ...))
```

- This is the most critical step. `copyout` is the function that securely bridges the kernel-user memory boundary.
- It copies data from our safe kernel source (`kernel_stats`) to the validated user destination pointer (`user_stats_ptr`).

Task 7 & 8: Create User-Space Interface

Why?

Define the `procstats` struct in `user.h` so user programs understand its layout. The assembly stub in `usys.S` handles the transition from user to kernel mode.

```
1       // In user.h - Define the struct and function prototype
2       struct procstats {
3         //with two integers for count and ticks
4       };
5       int getstats(int *stats_array);
6
7       // In usys.S - Add the syscall stub
8       SYSCALL(getstats)
```

Task 9: Create Test Program `statstest.c`

How to Test Your Work

Create this user program to verify that your syscall works correctly. It calls `getstats()` periodically and prints the retrieved statistics.

```
1       #include "types.h"
2       #include "stat.h"
3       #include "user.h"
4         int main(void)
5         {
6           int stats[2];
7           int i;
8           for(i = 0; i < 2; i++){
9             // When you pass 'stats', you are correctly passing a pointer to the
                  beginning of the array.
10            if(<call the system call> == 0){
11              // Access the elements using array indices.
12              printf(1, "Scheduled %d times, ran for %d ticks\n", stats[0], stats[1]);
13            } else {
14              printf(2, "getstats failed\n");
15            }
16            sleep(10);
17          }
18          exit();
19        }
```

Task 10: Build and Run

Final Steps

Add your new test program to the `UPROGS` variable in the `Makefile`. Then, build xv6 and run it in the QEMU emulator to see the output.

```
# In Makefile, add _statstest to the UPROGS list
UPROGS=\
...
_statstest\

# In your shell, from the xv6 root directory
make clean
make
make qemu
```

Submission Checklist

- Make sure your code compiles without warnings or errors.
- The `statstest` program should run and produce sensible output.
- Submit all the kernel and user-space codes that you have modified in a text file.
- Attach screenshots of your final output.

Good Luck!