

CSIS0230A Principles of Operating Systems (Class A)

Tutorial 11

Scheduling and nice level

The computer in front of you has a modified kernel in the file `/home/os/bzImage`. Like the one we made during tutorial 4, it keeps a global non-negative counter allowing user to increment, decrement and get the value of the counter through system calls. It also has an extra system call to reset the counter. The user program `testcounter.c` demonstrates how to call these system calls.

Write a program which creates 4 new processes using `fork()`, adjusted to run at nice levels 0, 5, 10 and 15 respectively, by using `nice()` to add 5 to the nice level after each `fork()`. For convenience, we will call these processes 0, 1, 2 and 3. All of them should then call `inccount()` `NUM_ITERATIONS=1000000` times. Every time an unexpected number is returned (i.e., some other processes have called `inccount()`), the process outputs the previous block of counter values it gets from the system call, in the following format:

```
1: curr=60275, last range 23946-38280(14335)
```

This means the process with name 1 makes the `inccount()` system calls 14335 times, getting values 23946 to 38280, but after that the next value obtained is 60275 rather than 38281. Output similar lines for the last group of values returned by the system call. This can be done using the following code, stored in `do_work.c`:

```
/* define NUM_ITERATIONS here */
```

```
void do_work(int id) {
    int left = -1, last, i, curr = -1;
    for (i = 0; i < NUM_ITERATIONS; ++i) {
        if (left == -1)
            last = left = inccount();
        else {
            curr = inccount();
            if (curr != last + 1) {
                printf("%d: curr=%d, last range %d--%d(%d)\n",
                    id, curr, left, last, last-left+1);
                left = curr;
            }
            last = curr;
        }
    }
    printf("%d: curr=%d, last range %d--%d(%d) (last)\n",
        id, curr, left, last, last-left+1);
}
```

After testing your program, **answer the following questions:**

1. The compiler compiles the increment operation within the kernel as a non-atomic operation. That is, increment is done by loading the value of the counter, adding one to it, and store it back; and there is no guarantee that another instruction will not be done between the load and the store. The kernel itself is not preemptive, i.e., will not switch to another process until in user mode. Is it possible that two processes update the counters and execute the

same instruction simultaneously, causing the kernel counter to be corrupted? Why?

2. If our machine has a preemptive kernel, will the non-atomicity of the instruction causes problems? What if it has two or more processors?
3. When a process outputs a message, did the process wait (i.e., sleep for a while allowing other processes to execute)? Evidence? (Hint: try to see what would happen if it does wait, say for a microsecond. Explain why that happens.)
4. By looking at the counts given to each process, reconstruct the complete scheduling sequence, i.e., show exactly which process is executed after which. Why the ordering is not exactly the ordering of the printed messages? (Hint: consider when each process prints a message.)
5. Use a larger number of iterations (e.g., 3000000) and run your program again. Find the number of counts given to each process during one epoch. Relate this with the nice-level.
6. If time permits, add one more “I/O bound” process into the mix to see how it behave. The new process should be similar to other processes, using *inccount()* to increment the counter and print out messages to show what counts it gets. It should have a nice level of 5 (create it before all othe processes). The process doesn’t really do I/O, so instead we just call *usleep(200000)* to ask it to sleep for 0.2s for every 10000 counts received to emulate I/O. Observe whether this new process can preempt each of the other processes.