CIS 452 - Operating Systems Concepts

Nathan Bowman

Images taken from Silberschatz book

Process Synchronization

Recall that we can have processes (or now, threads) that work concurrently or in parallel

Concurrently: fast context switching between processes

Parallel: multiple processing cores allow actually doing more than one thing at once

We also have processes that cooperate via interprocess communication (IPC)

This leads to difficulties maintaining integrity of shared data, some of which are quite subtle

Consider an alternative solution to producer-consumer problem compared to what we saw previously

Just one shared variable, `counter`, is used

Do you see the bug?

Producer:

```
while(true) {
    /* produce an item in next_produced */

    while (counter == BUFFER_SIZE)
        ; /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Consumer:

```
while(true) {
    while (counter == 0)
        ; /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
```

```
    /* consume the item in next_consumed */
}
```

Somewhat a trick question... both programs run fine separately

Problem is access to shared variable `counter`

Assume `counter` is 5 and producer and consumer execute `counter++` and `counter--` concurrently

Correct result is 5, but let's see what else can happen

`counter++` in (pseudo-)machine code is

```
register1 = counter          // A
register1 = register1 + 1    // B
counter = register1          // C
```

`counter--` in (pseudo-)machine code is

```
register2 = counter          // 1
register2 = register2 - 1    // 2
counter = register2          // 3
```

Because execution of `counter++` and `counter--` is concurrent, interrupt can happen at any point in previous machine code

Instructions `{1, 2, 3}` and `{A, B, C}` can be interleaved in any order as long as their within-program order is maintained

For example, `1, 2, A, B, C, 3` is possible, but `2, 1, A, B, C, 3` is not because 2 cannot come before 1

```
producer: register1 = counter           // register1 = 5
producer: register1 = register1 + 1     // register1 = 6
--- CONTEXT SWITCH ---
consumer: register2 = counter           // register2 = 5
consumer: register2 = register2 - 1     // register2 = 4
--- CONTEXT SWITCH ---
producer: counter = register1           // counter = 6
--- CONTEXT SWITCH ---
consumer: counter = register2           // counter = 4
```

Final value of `counter = 4` is incorrect!

Could also have found `counter = 5` or `counter = 6` by rearranging instructions

This is a **race condition**

Occurs when two processes try to modify the same state simultaneously

Final result depends on order of execution

To fix, we must ensure that only one process can modify `counter` at once

Processes must somehow be **synchronized**