# OS 4/2

*Reagan Shirk*

*April 2, 2020*

## Peterson's Solution to Critical Section Access

- An okay solution, not great. Works most of the time
- I fell asleep during the last 15 minutes of lecture on Tuesday so hopefully I understand all of this
- But before we jump right into Peterson's Solution, we're gonna talk about some examples of protecting the critical section

### Examples of Protecting the Critical Section

- Pretend this is in a producer/consumer loop or that it's getting called frequently

```
Process 0:

while (turn != 0);
// Critical Section
turn = 1;

Process 1:

while (turn != 1);
// Critical Section
turn = 0;
```

- The shared variable `turn` protects the two critical sections by allowing the code to bounce back and forth between the loops
  - Pros: very simple
  - Cons: We don't want the pace of execution to be dictated by the slower process, but in this code it is. If one process fails, the other will be blocked
- Here's another code snippet to analyze:

```
Process 0:

while(flag[1]);
flag[0] = true;
// Critical Section
flag[0] = false;

Process 1:

while (flag[0]);
flag[1] = true;
// Critical Section
flag[1] = false;
```

- Pros: They depend on their own flags so they won't get stuck if there's a crash outside of the critical section

- Cons: You could have a situation where the order of operations is: `p0: flag[1] == false, p1: flag[0] == false, p0: flag[0] == true, p1: flag[1] == true` which isn't good. It's also an issue that if failed in the critical section, the other process is blocked
- And for our third attempt

```
Process 0:

flag[0] = true;
while(flag[1]);
// Critical Section
flag[0] = false;

Process 1:

flag[1] = true;
while(flag[0]);
// Critical Section
flag[1] = false;
```

- Pros: Declaring our intentions ("which is always good in every relationship" - Dr. Grant lmao)
- Cons: Could permanently block a process if it fails in the critical section, could create a deadlock in the situation `p0: flag[0] == true, p1: flag[1] == true`
- Fourth attempt damn how many of these are we going to do

```
Process 0:

flag[0] = true;
while(flag[1])
{
    flag[0] = false;
    // Delay
    flag[0] = true;
}
// Critical Section
flag[0] = false;

Process 1:

flag[1] = true;
while(flag[0])
{
    flag[1] = false;
    // Delay
    flag[1] = true;
}
// Critical Section
flag[1] = false;
```

- Pros:
- Cons: In the situation `p0: flag[0] = true, p1: flag[1] = true, p0: flag[1] = true, p1: flag[0] = true, p0: flag[0] = false, p1: flag[1] = false, p0: flag[0] = true, p1: flag[1] = true`, since the delay is timer based we can't rely on it to help us here. Both could get the same exact delay over and over again, and we end up with both execution elements looping in the while loop over and over and over. Creates livelock: processes change states but do not do any meaningful/useful work
- The rules we learned with all of these:

  - Impose an order with a "turn"
  - each process should have the ability to progress independently
  - each process should be able to back off

### Back to Peterson's Solution

```
i = 0;
j = 1;
// i should never equal j
flag[i] = true;
turn = j;
while (flag[j] && turn = j);
// Critical Section
flag[i] = false;
```

- There's a built in "back off" by having `turn = j` in the while loop
- Struggles in modern machines where you have multiple load/store... things

## Hardware Support for Synchronization

### Memory Barrier

```
while (!flag)
{
    memory_barrier();
}
```

- This is usually used in the kernel
- Considered a low level operation

### Test and Set

- A function that sets a variable that is passed in and returns the original variable