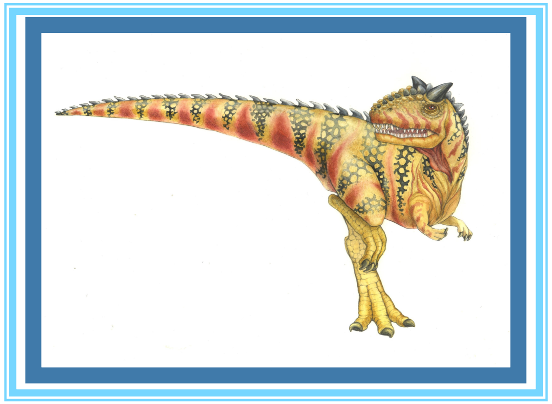
Chapter 6: Process Synchronization



Silberschatz, Galvin and Gagne ©2009

Shared Buffer by Circular Array

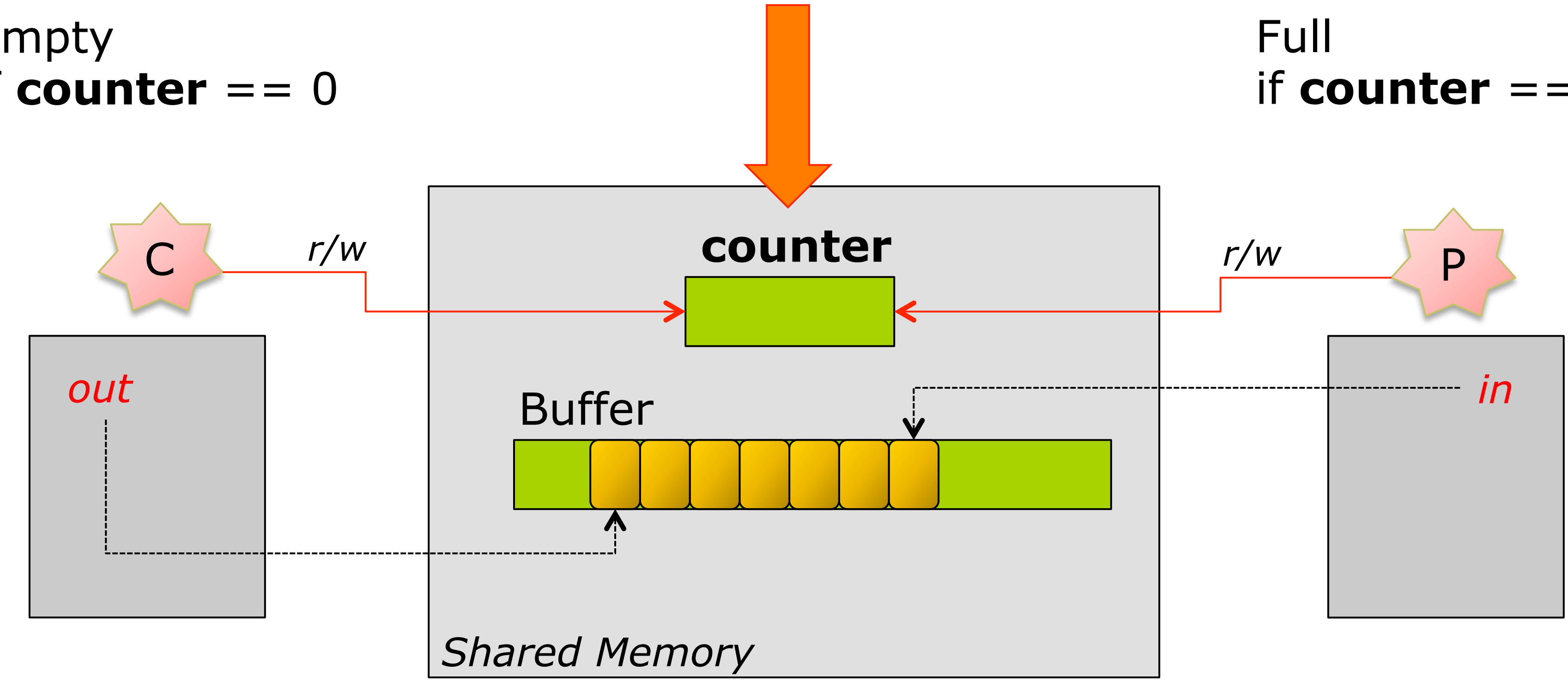
```
counter++:
    register1 = counter
    register1 = register1 + 1
    counter = register1
```

```
counter--:
    register2 = counter
    register2 = register2 - 1
    counter = register2
```

Concurrency Problem !

Empty
if **counter** == 0

Full
if **counter** == BS



Critical Section Problem

- Consider system of *n* processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has a **critical section**
  - If one process in critical section, no other process can
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**
- Critical section problem is to design protocol to solve this

Silberschatz, Galvin and Gagne ©2009

Critical Section

- General structure of process p_i is

```
do {
    entry section

        critical section

    exit section

        remainder section

} while (TRUE);
```

Figure 6.1 General structure of a typical process P_i.

Requirements of Critical-Section Prob.

1. **Mutual Exclusion** - If process P_i is in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and some processes wish to enter their critical section, then the selection of the next process cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes enter critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the n processes

1st: Use lock

```
shared int locked = 0;
do {
        while (locked == 1);
        locked = 1;
        critical section
        locked = 0;
        remainder section
} while (true);
```

- Fails to meet
- Solution: Allow only one process to

1st: Use lock

Process 0:

```
shared int locked = 0;
do {
        while (locked == 1);
        locked = 1;
        critical section
        locked = 0;
        remainder section
} while (true);
```

Process 1:

```
shared int locked = 0;
do {
        while (locked == 1);
        locked = 1;
        critical section
        locked = 0;
        remainder section
} while (true);
```

2^{nd}: Take turns

```
shared int turn = 0;
do {
        while (turn != me);
        critical section
        turn = ! me;
        remainder section
} while (true);
```

- Fails to meet
- Solution: Check if the other process

2^{nd}: Take turns

Process 0:

```
shared int turn = 0;
do {
        while (turn == 1);
        critical section
        turn = 1;
        remainder section
} while (true);
```

Process 1:

```
shared int turn = 0;
do {
        while (turn == 0 );
        critical section
        turn = 0;
        remainder section
} while (true);
```

3rd : Check intention

```
shared int flag[2];
do {
        flag[me] = true;
        while (flag[ !me] == true);
        critical section
        flag[me] = false;
        remainder section
} while (true);
```

- Fails to meet
- Solution: check both

3rd : Check intention

Process 0:

```
shared int flag[2];
do {
        flag[ 0 ] = true;
        while (flag[ 1 ] == true);
        critical section
        flag[ 0 ] = false;
        remainder section
} while (true);
```

Process 1:

```
shared int flag[2];
do {
        flag[ 1 ] = true;
        while (flag[ 0 ] == true);
        critical section
        flag[ 1 ] = false;
        remainder section
} while (true);
```

Peterson's Solution

```
shared int turn, flag[2];
do {
        flag[me] = true;
        turn = ! me;
        while (flag[! me] && turn == ! me);
        critical section
        flag[me] = false;
        remainder section
} while (true);
```

- Provable that

1. Mutual exclusion:
2. Progress:
3. Bounded-waiting:

Peterson's Solution

Process 0:

```
shared int turn, flag[2];
do {
        flag[me] = true;
        turn = ! me;
        while (flag[! me] && turn == ! me);
        critical section
        flag[me] = false;
        remainder section
} while (true);
```

Process 1:

```
shared int turn, flag[2];
do {
        flag[me] = true;
        turn = ! me;
        while (flag[! me] && turn == ! me);
        critical section
        flag[me] = false;
        remainder section
} while (true);
```