Process Synchronization

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to zero (0). It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {

        /*  produce an item and put in nextProduced  */
         while (counter == BUFFER_SIZE)
                ; // do nothing
                buffer [in] = nextProduced;
                in = (in + 1) % BUFFER_SIZE;
                counter++;
}
```

Consumer

```
while (true)  {
        while (counter == 0)
            ; // do nothing
            nextConsumed =  buffer[out];
             out = (out + 1) % BUFFER_SIZE;
              counter--;

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

Race Condition

- counter++ could be implemented as

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

- counter-- could be implemented as

```
register2 = counter
register2 = register2 - 1
count = register2
```

- Consider this execution interleaving with “count = 5” initially:

S0: producer execute register1 = counter {register1 = 5}
S1: producer execute register1 = register1 + 1 {register1 = 6}
S2: consumer execute register2 = counter {register2 = 5}
S3: consumer execute register2 = register2 - 1 {register2 = 4}
S4: producer execute counter = register1 {count = 6 }
S5: consumer execute counter = register2 {count = 4}

- Incorrect state due to the concurrent execution of P and C processes.
- Processes access same data concurrently and the result depends on a particular ordering of process execution is called Race Condition

Critical Section Problem

- Consider system of n processes $\{p_o, p_1, \ldots p_{n-1}\}$
- Each process has critical section segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- Critical section problem is to design protocol to solve this
- Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section
- Especially challenging with preemptive kernels

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.