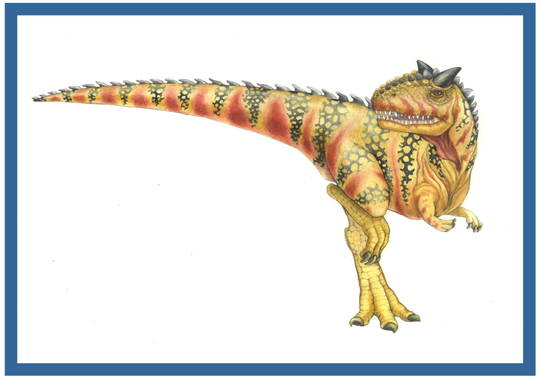
Chapter 6: Synchronization



Objectives

- To examine several **classical process-synchronization problems**

- To explore several tools that are used to solve process synchronization problems
  - **Windows API** for Process/Thread Synchronization **: Semaphores, Event, CRITICAL_SECTION, Mutex**
  - **POSIX System Calls** for IPC(Interprocess Communication) : Pipies, FIFIOs, Streams, **Semaphores, Message Queues, Shared Memory**, Sockets

Background

- **Cooperating process** is one that can affect or be affected by other processes
- Cooperating processes can either
  - directly **share a logical address space**
    - **shared memory**, both code and data or
    - Through the use of **threads**
  - be allowed to share data through files or **messages**

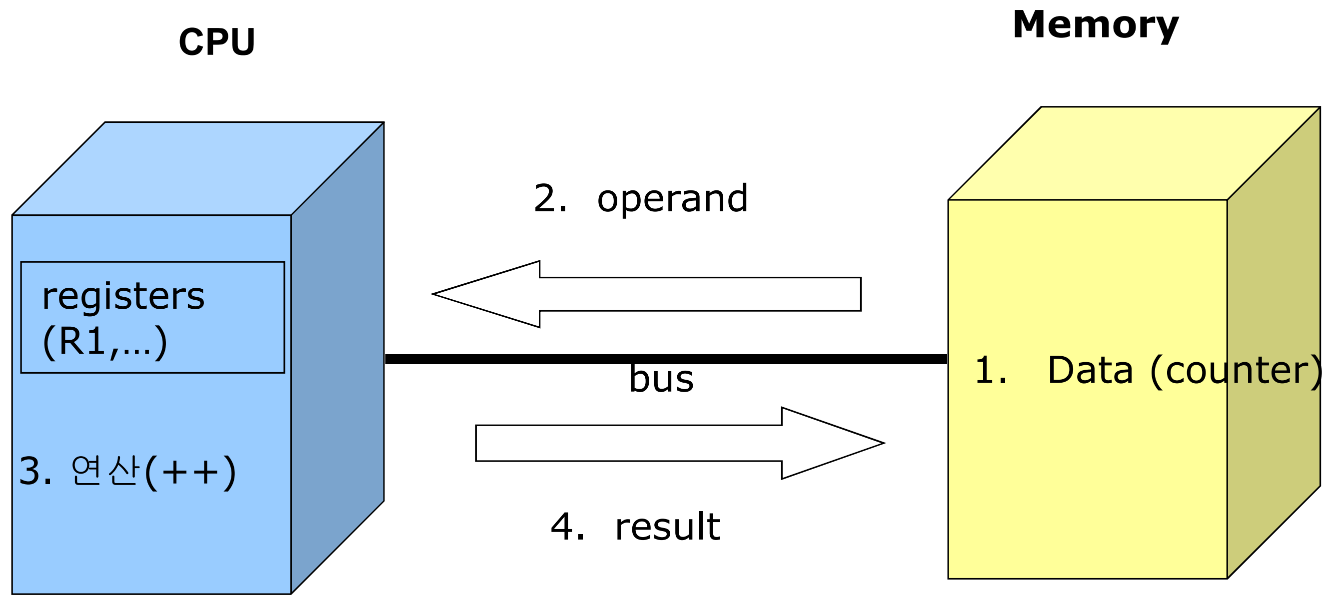
- Processes can **execute concurrently**
  - May be interrupted(context switching) at any time, partially completing execution

- **Concurrent access to shared data** may result in **data inconsistency**
- **Maintaining data consistency** requires **mechanisms** to ensure the orderly execution of cooperating processes

Background

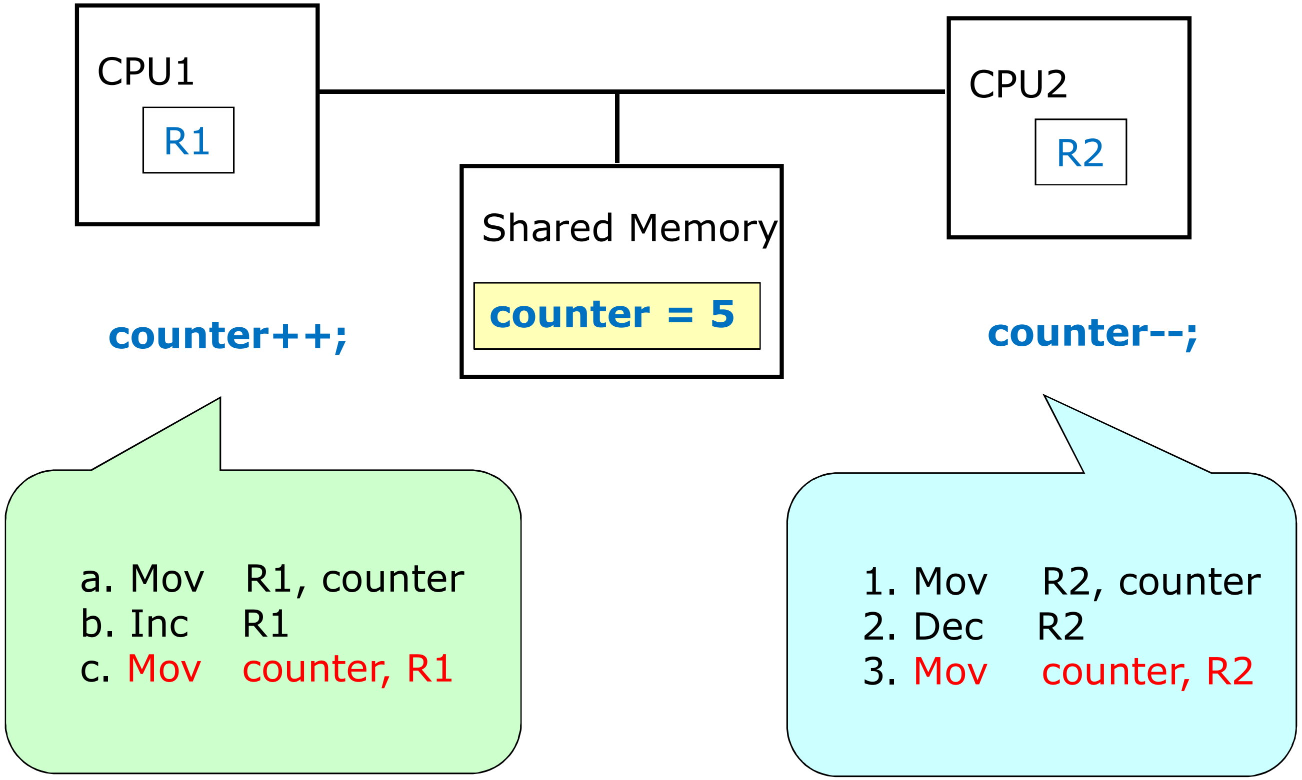
- **Race Condition**
  - The situation where several processes access and manipulate shared data concurrently.
  - The final value of the shared data depends upon which process finishes last.
- To prevent race conditions, **concurrent processes must be synchronized**.

"counter++" is not ATOMIC




```
2:  Mov   R1, counter
3:  Inc   R1
4:  Mov   counter, R1
```

Example of a Race Condition



Interleaved execution을 하면?

a,b,c,1,2,3 -> 5, a,1,b,2,c,3 -> 4, a,1,b,2,3,c -> 6

- We need to ensure that **only one process at a time can be manipulating the variable `counter`**
  - Serial component of Amdahl's Law

Synchronization

- **Race Condition**

  A situation in which multiple threads or processes read and write the same data concurrently and the outcome of the execution depends on the particular order in which the access take place

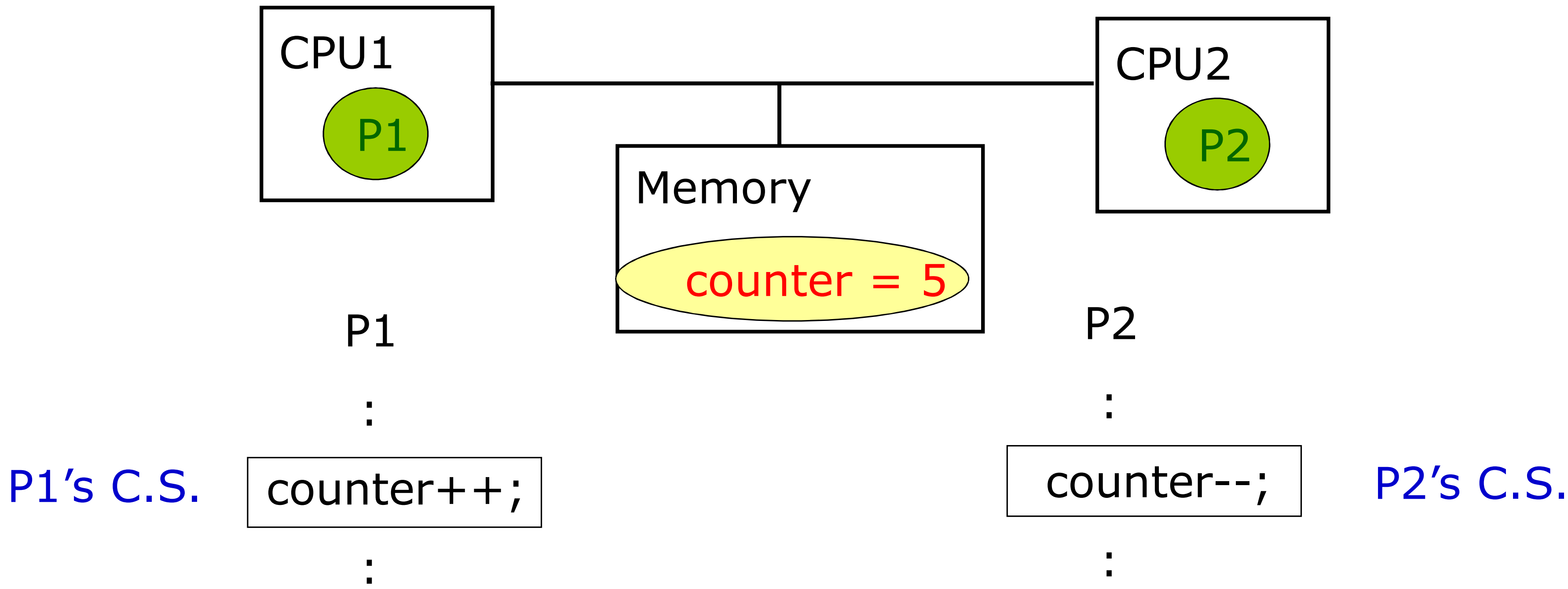
- **Critical Section**

  A section of code within a process or a thread that requires access to shared resources and that may not be executed while another process is in a corresponding critical section.

- **Mutual Exclusion**

  The requirement that when one process is in a critical section, no other process may be in a corresponding critical section.

Critical Section



- **Maintaining data consistency** requires **mechanisms** to ensure the orderly execution of cooperating processes

Mutex Locks

- **mutex lock**
  - *mutex* is short for *mut*ual *ex*clusion
  - to protect critical regions and thus prevent race conditions
  - A process must acquire the lock before entering a critical section
    - `acquire()` function
  - It releases the lock when it exits the critical section
    - `release()` function
  - a boolean variable `available` indicating if lock is available or not
  - Calls to `acquire()` and `release()` must be atomic
    - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

acquire() and release()

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (true);
```

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;;
}
release() {
    available = true;
}
```

Semaphores

- A more robust synchronization tool that provide more sophisticated ways for processes to synchronize their activities
- Semaphore *S* : integer variable
- Can only be accessed via **two indivisible (atomic) operations**
  - Originally called **P**(or **wait**) and **V**(or **signal**)

```
wait (S) {
     while (S <= 0)
        ; // busy wait
     S--;
}
```

P (*S*): while (*S*≤ 0) do *no-op*;
**   *S*--;**

i.e. wait

If positive, decrement-&-enter.
Otherwise, wait until it gets positive

```
signal (S) {
     S++;
}
```

V (*S*):
**   *S*++;**

Usage of Semaphore - Critical Section of *n* Processes

- Shared data:

  semaphore s; // shared semaphore among *n* processes
  // initialized to *1*

- Process P_i :

```
repeat
  wait(s);        // entry section
      critical section
  signal(s);      // exit section

      remainder section
until false;
```

Usage of Semaphore - Synchronization of Two Processes

- Problem
  - Two concurrently running processes, P1 and P2
  - The statement S2 of P2 should be executed only after the statement S1 of P1.
- Implementation

  Shared data:

  Semaphore synch; // P1 and P2 share a common semaphore

  // initialized to 0

Process P1:

```
S1;
signal(synch);
```

Process P2:

```
wait(synch);
S2;
```

Two Types of Semaphores

- ***Binary semaphore***
  - integer value can range only between 0 and 1;
- ***Counting semaphore***
  - integer value can range over an unrestricted domain.

Classical Problems of Synchronization

- Classical problems used to test nearly every newly-proposed synchronization schemes
  - Producer-consumer Problem(Bounded-Buffer Problem)
    - **제한된 버퍼에 데이터를 채우고/가져가는 문제**
  - Readers and Writers Problem
    - 데이터의 공유문제
  - Dining-Philosophers Problem
    - 한 자원을 가지고 다른 자원을 요청하는 문제

Producer-Consumer Problem

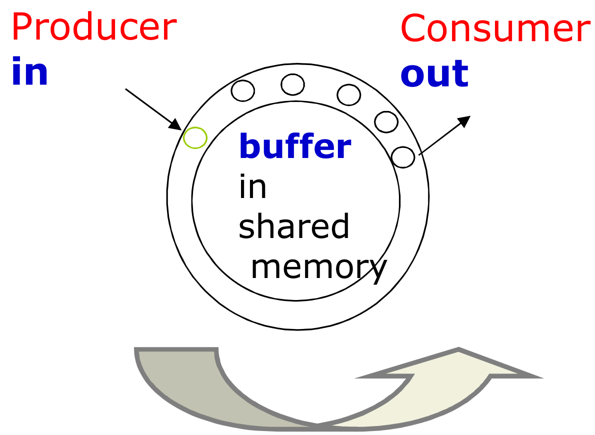
- Paradigm for cooperating processes
- ***producer*** process produces information that is consumed by a ***consumer*** process
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size

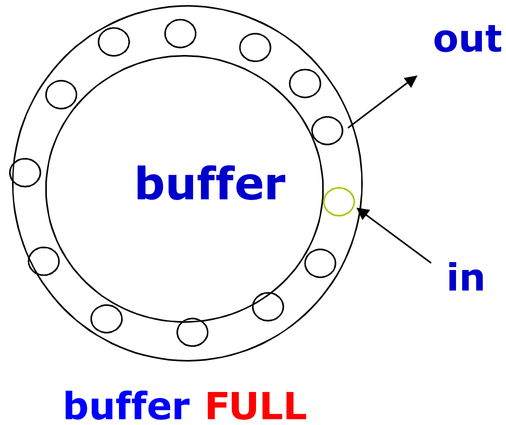
Bounded-Buffer – Shared-Memory Solution

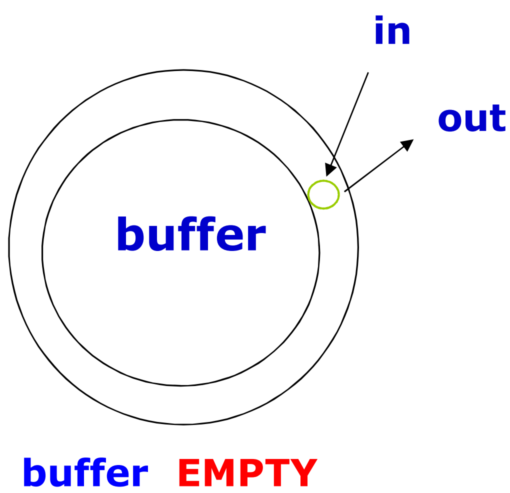
- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements

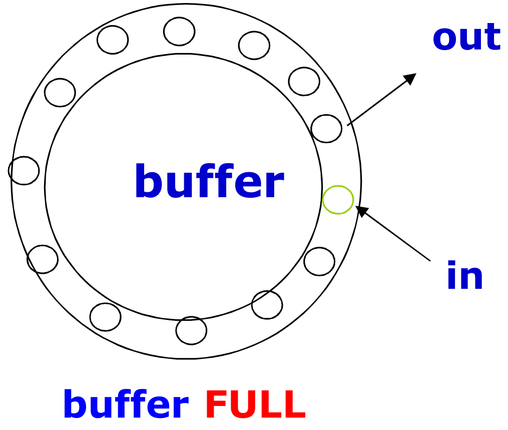






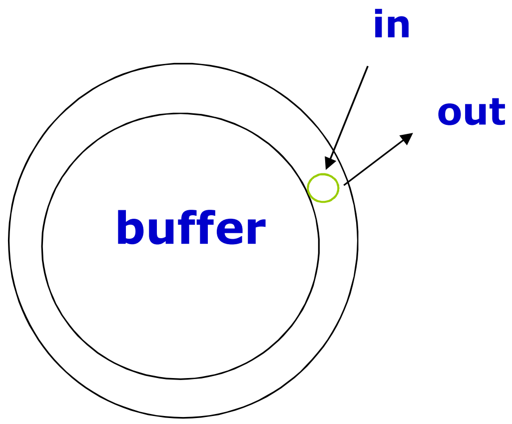
Producer - Busy Waiting Solution

```
item next_produced;
while (true) {
   /* produce an item in next_produced */
   while (((in + 1) % BUFFER_SIZE) == out)
      ; /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
}
```



Consumer - Busy Waiting Solution

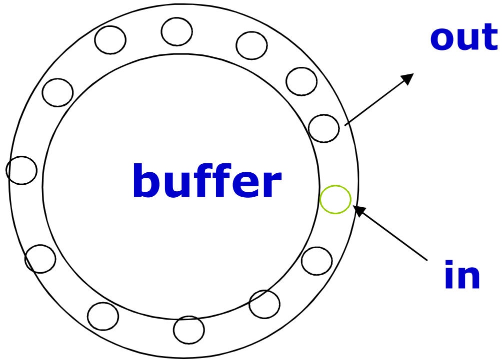
```
item next_consumed;
while (true) {
      while (in == out)
            ; /* do nothing */
      next_consumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;

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



buffer **EMPTY**

Producer-Consumer Problem : Busy Waiting Solution

```
while (true) {
     /* produce an item in next produced */
     while (counter == BUFFER SIZE) ;
          /* do nothing */
     buffer[in] = next produced;
     in = (in + 1) % BUFFER SIZE;  counter++;
}
```

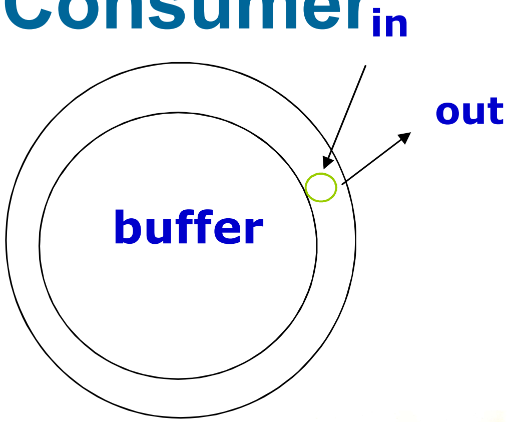


buffer FULL

Producer

Consumer

```
while (true) {
     while (counter == 0)
          ; /* do nothing */
     next consumed = buffer[out];
     out = (out + 1) % BUFFER SIZE; counter--;
     /* consume the item in next consumed */
}
```



buffer EMPTY

Producer-Consumer Problem : Semaphore Solution

- *n* buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value *0*
- Semaphore **`empty`** initialized to the value *n*

Producer – Semaphore Solution

- The structure of the producer process

```
do {
       ...
      /* produce an item in next_produced */
       ...
   wait(empty);
   wait(mutex);
       ...
      /* add next produced to the buffer */
       ...
   signal(mutex);
   signal(full);
} while (true);
```

Consumer – Semaphore Solution

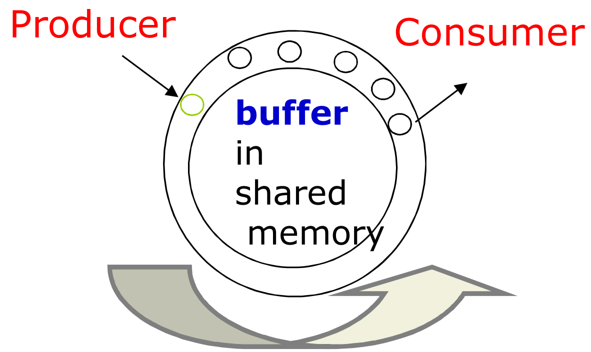
- The structure of the consumer process

```
do {
    wait(full);
    wait(mutex);
        ...
       /* remove an item from buffer to next_consumed */
        ...
    signal(mutex);
    signal(empty);
        ...
       /* consume the item in next consumed */
        ...
    } while (true);
```

Producer-Consumer Problem : Semaphore Solution

Shared data

```
semaphore full = 0, empty = n, mutex =1;
```



Producer

```
do {                    ...
   produce an item
      ...
   wait(empty);   // P(empty)
   wait(mutex);
      ...
   add item to buffer
      ...
   signal(mutex);
   signal(full);      // V(full)


} while (1);
```

Consumer

```
do {

   wait(full)      // P(full)
   wait(mutex);
     ...
   remove an item from buffer
     ...
   singal(mutex);
   signal(empty); // V(empty)
    ...
   consume the item
    ...
} while (1);
```