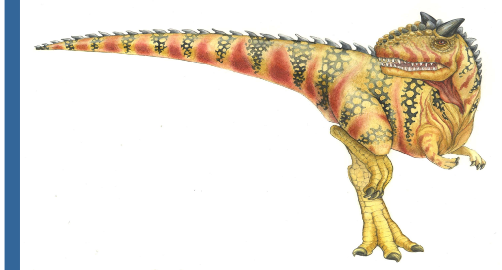
Chapter 6: Synchronization

School of Computing, Gachon Univ.
Jungchan Cho



Most slides from "Operating System Concepts – 10th Edition".
Many slides are taken from lecture notes of Prof. Joon Yoo.

Chapter 6: Synchronization

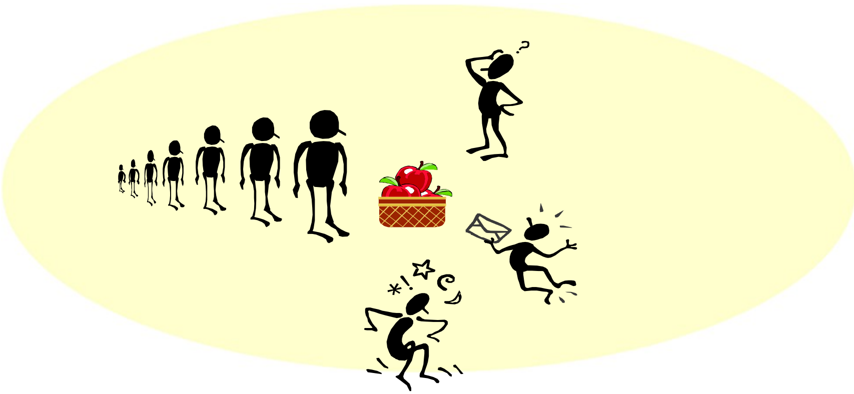
- Background
- The Critical-Section Problem
- Software C.S.: Peterson's Solution
- Hardware C.S.: Mutex Locks
- Semaphores
- Classic Problems of Synchronization

Objectives

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To examine several classical process-synchronization problems

Background

- Resource sharing
  - memory, files
    - interprocess communication (IPC)
    - multi-threads



- Why process synchronization?
  - <u>Concurrent or parallel access</u> to shared data may result in data <u>inconsistency</u>
    - Recall the multithread example in Ch. 4:

```
Output:
x is 2
x is 3
```

```
Output:
x is 3
x is 2
```

```
Output:
x is 3
x is 3
```

```
Output:
x is 2
x is 2
```

  - **Solution to the problems**
    : Maintaining data consistency requires mechanisms to ensure the *<u>orderly execution</u>* of cooperating processes
    - **Process synchronization** mechanisms

Possible output?

- Shared code:

```
int x = 1; //global variable
void* func(void* p){
   x = x + 1;
   printf("x is %d\n", x);
   return NULL;
}
```

- fork version:

```
main(…) {
     fork();
     func(NULL);
}
```

- threads version:

```
main(…) {
   pthread_t tid;

   pthread_create(&tid,NULL,func,NULL);
   func(NULL);
}
```

Possible output: threads case 1

```
int x = 1; //global variable
```

```
void* func(void* p){
   x = x + 1;
   printf("x is %d\n", x);
   return NULL;
}
```

```
void* func(void* p){
   x = x + 1;
   printf("x is %d\n", x);
   return NULL;
}
```

time

Parent thread

Child thread

Possible output: threads case 2

```
int x = 1; //global variable
```

```
void* func(void* p){
   x = x + 1;
```

```
void* func(void* p){
   x = x + 1;
   printf("x is %d\n", x);
   return NULL;
}
```

```
   printf("x is %d\n", x);
   return NULL;
}
```

time

Parent thread **Child thread**

Possible output: threads case 3

```
int x = 1; //global variable
```

```
void* func(void* p){
   x = x + 1;
   printf("x is %d\n", x);

   // interrupted during printf()

   printf("x is %d\n", x);

   return NULL;
}
```

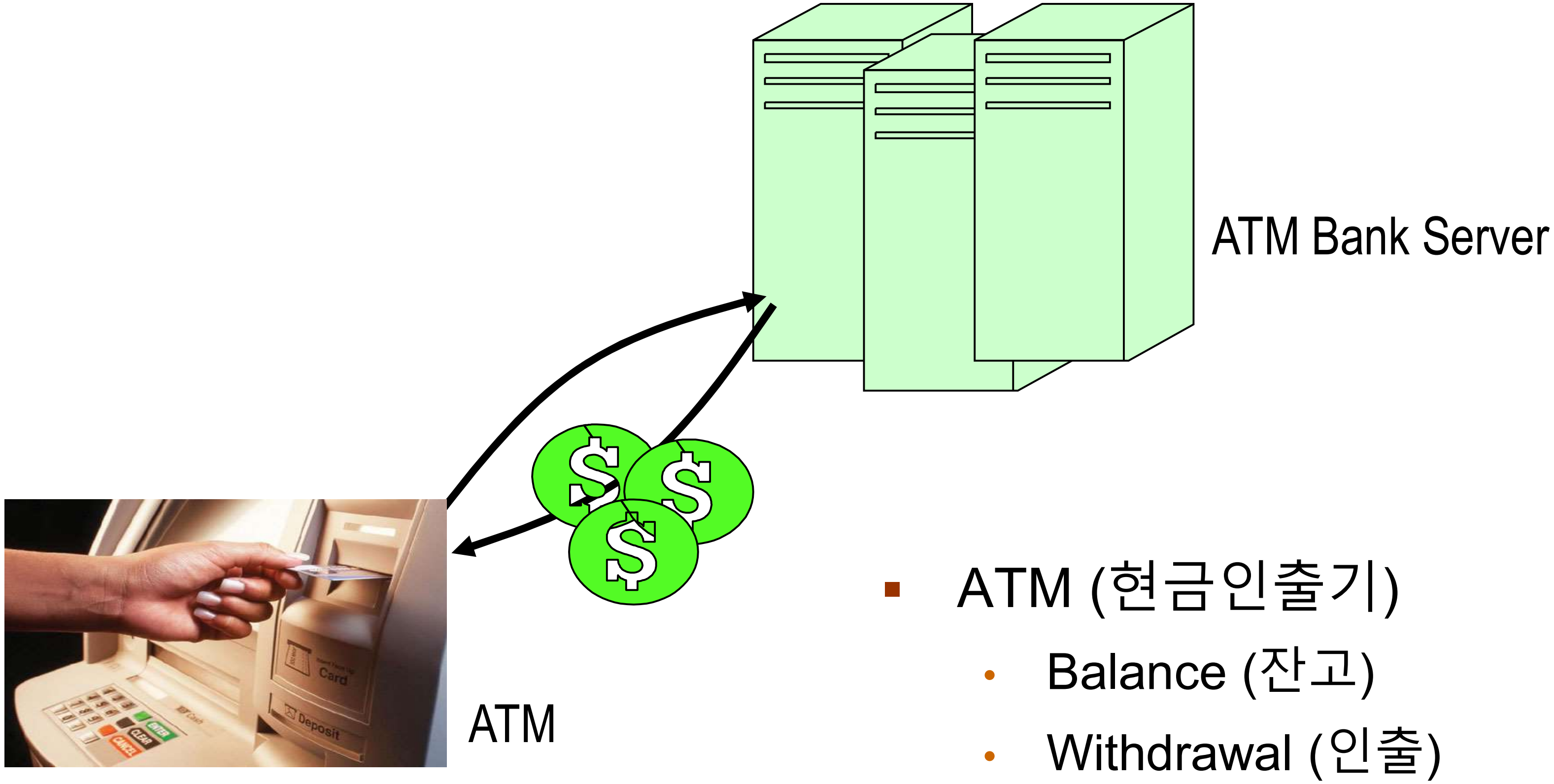
```
void* func(void* p){
   x = x + 1;
   printf("x is %d\n", x);
   return NULL;
}
```

time

Parent thread

Child thread

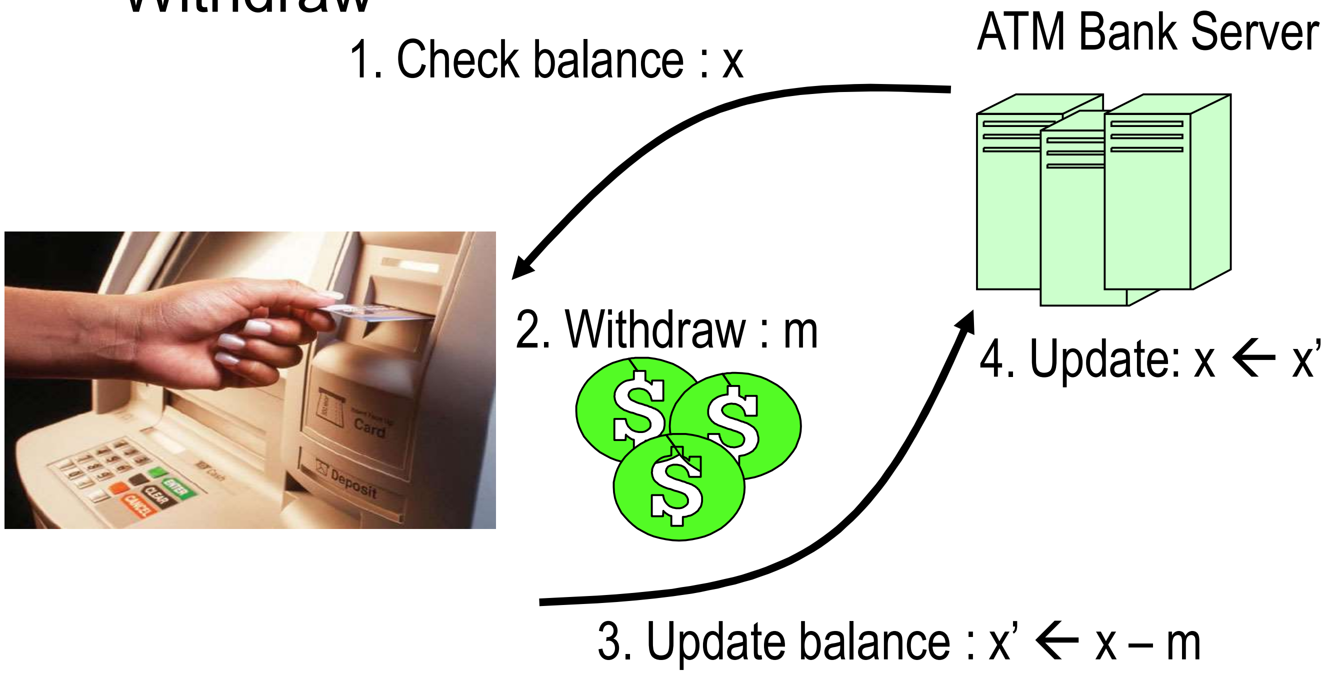
Example: ATM Bank Server



- ATM (현금인출기)
  - Balance (잔고)
  - Withdrawal (인출)
  - Deposit (입금)

Example: ATM Bank Server

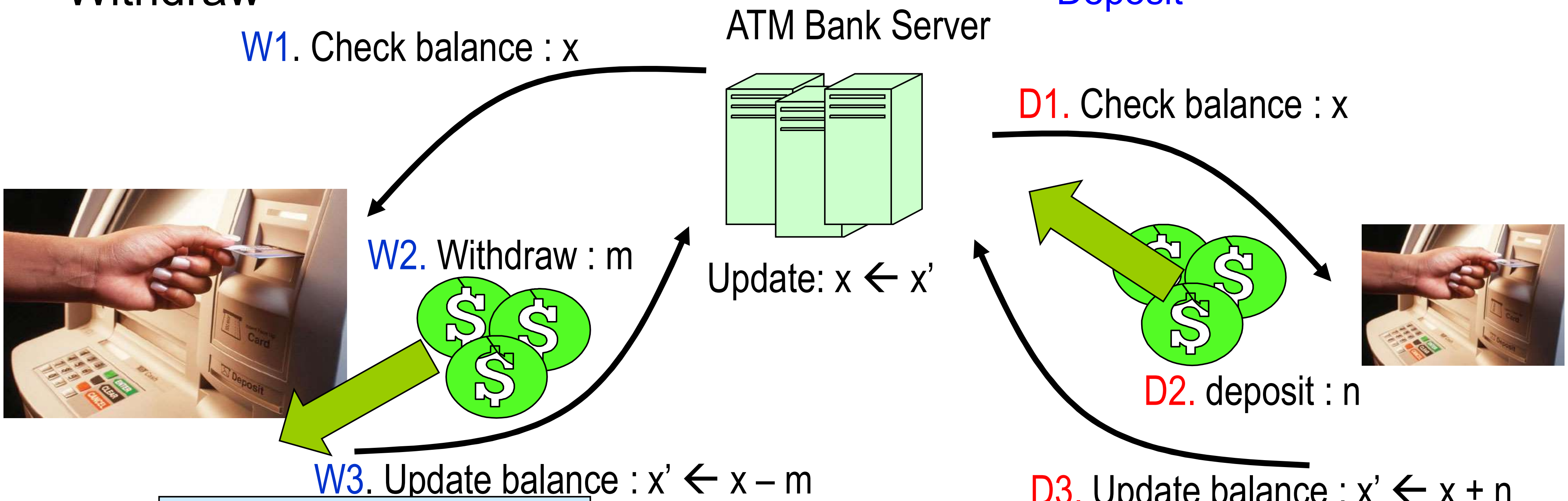
- **Single transaction**
    - Withdraw



- Deposit
    - x ← x + m

Example: ATM Bank Server

- <u>Concurrent</u> **transactions**
  - Withdraw
  - Deposit



Order of execution

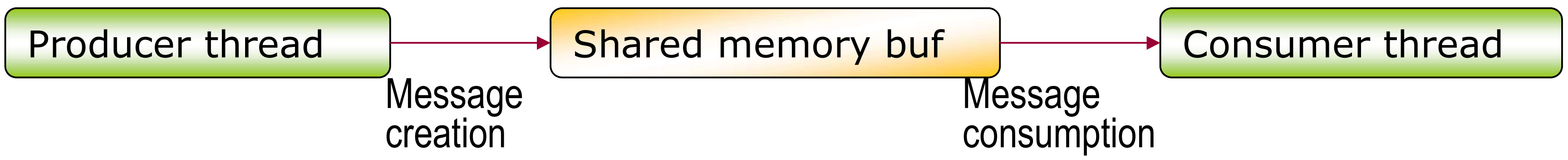
W1 → W2 → W3 → D1 → D2 → D3 : final x=?

vs.

W1 → W2 → D1 → D2 →W3 → D3 : final x = ?

Example2: Producer-Consumer problem

- **Producer-consumer problem**
  - Producer threads
    - Set of threads that generates messages
  - Consumer threads
    - Set of threads that consumes messages



Example3: Producer-Consumer problem

int counter=0; // global variable

Number of items in buffer

PRODUCER thread

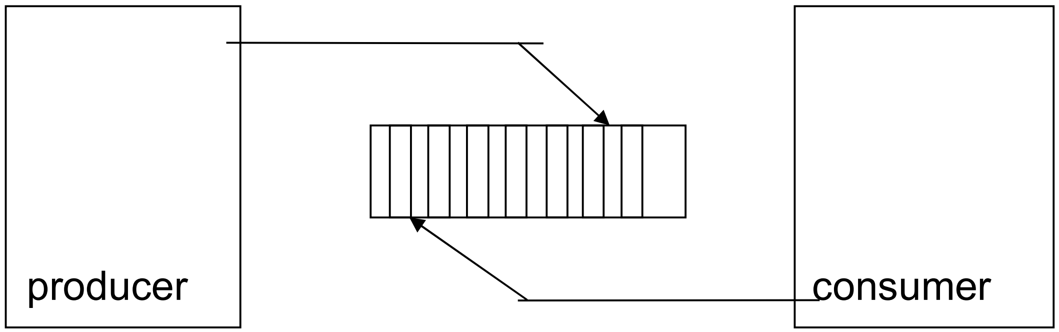
```
item   nextProduced;

while (TRUE) {
    while (counter == BUFFER_SIZE);
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

CONSUMER thread

```
item  nextConsumed;

while (TRUE) {
    while (counter == 0);
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```



C code → Assembly code

- Note that counter++; ← Actually, this C code is compiled into 3 assembly codes:

  $register_1$ = counter #load from memory
  $register_1$ = $register_1$ + 1
  counter = $register_1$ #store in memory

- counter--; ← is compiled into:

  $register_2$ = counter #load from memory
  $register_2$ = $register_2$ - 1
  count = $register_2$ #store in memory

- Consider this execution with "count = 5" initially. **Producer** adds item (counter++) then **Consumer** deletes an item (counter--). What is the result?

  T_0: producer execute $register_1$ = counter {$register_1$ = 5}
  T_1: producer execute $register_1$ = $register_1$ + 1 {$register_1$ = 6}
  T_4: producer execute counter = $register_1$ {counter = 6}
  T_2: consumer execute $register_2$ = counter {$register_2$ = 6}
  T_3: consumer execute $register_2$ = $register_2$ - 1 {$register_2$ = 5}
  T_5: consumer execute counter = $register_2$ {**counter = 5**}

Race Condition

- But not always!
- Consider this execution with "count = 5" initially. **Producer** adds item (count++) then **Consumer** deletes an item (count--). What is the result?
- Interleaving instructions

Context switch P1→P2

Context switch P2→P1

Context switch P1→P2

T_0: producer execute $register_1$ = counter {$register_1$ = 5}
T_1: producer execute $register_1$ = $register_1$ + 1 {$register_1$ = 6}
T_2: consumer execute $register_2$ = counter {$register_2$ = 5}
T_3: consumer execute $register_2$ = $register_2$ - 1 {$register_2$ = 4}
T_4: producer execute counter = $register_1$ {counter = 6 }
T_5: consumer execute counter = $register_2$ {counter = 4}

?

- Result (counter) can be either 4, 5 or 6!!
  - This is called **race condition**

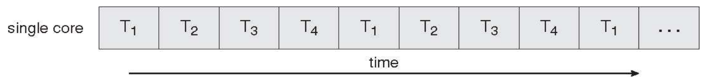
Race Condition

- Although both the producer and consumer routines are correct separately, they may not function correctly when executed concurrently (or in parallel)

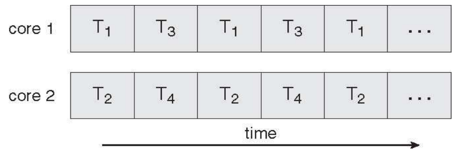
- **Race Condition**:
  - Several processes access and manipulate the same data concurrently
  - The outcome of the execution depends on the particular order in which the access takes place
- To prevent race conditions, concurrent processes must be *synchronized*

But why does Race Condition happen?

- Processes can execute **concurrently** or in **parallel**
- Concurrent execution: thread scheduling



  - CPU scheduler switches rapidly between threads for concurrent execution
  - One thread may only partially complete execution before another thread is scheduled – interrupt



- Parallel execution: multicore
  - Two threads execute simultaneously on separate processing cores
  - Each thread can share data
- Race condition!

Enforcing mutual exclusion

- How can we avoid the race condition?
- Answer: We must **synchronize** the execution of the threads
  - i.e., need to guarantee **mutually exclusive access** to **critical sections**
- Classic solution
  - Software C.S.: Peterson's Solution
  - Hardware C.S.: Mutexes Locks
  - Semaphores (Edsger Dijkstra)

Chapter 6: Synchronization

- Background
- The Critical-Section Problem
- Software C.S.: Peterson's Solution
- Hardware C.S.: Mutex Locks
- Semaphores
- Classic Problems of Synchronization

Critical-Section Problem

- *N* processes all competing to use some shared data
- Each process has a code segment, called **critical section (C.S.)**, in which the **shared data** *is accessed*.
  - Only one process execute in its critical section at a time
    - ensure that when *one* process is executing in its **critical section**, *no other processes* are allowed to execute in its critical section
  - **Critical section**
    - a piece of code that accesses a shared resource (e.g., data structure or device)
- The critical-section problem
  - Design a protocol that the processes can use to cooperate

Critical-Section Problem

- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

No one else can use the CS

Critical-Section Problem

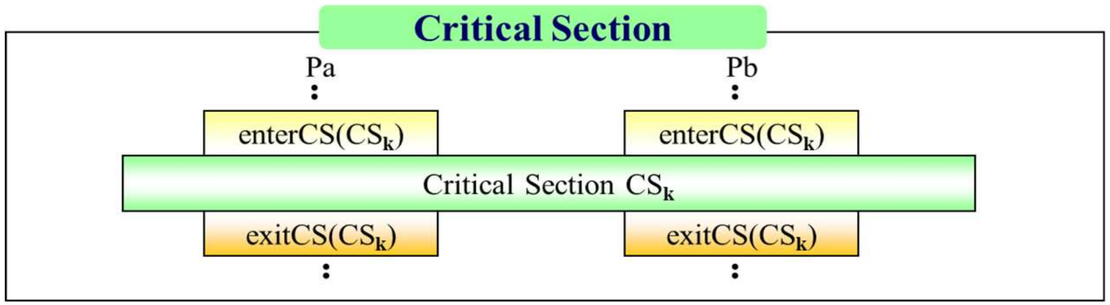
- General Structure of Critical Section

```
Do
{
        entry section

        critical section

        exit section
        remainder section

} while ( TRUE );
```



Example

- In the producer-consumer problem,

PRODUCER

```
item   nextProduced;

while (TRUE) {
    while (counter == BUFFER_SIZE);
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;

    counter++;

}
```

CONSUMER

```
item  nextConsumed;

while (TRUE) {
    while (counter == 0);
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    counter--;
```

Critical section

Example – Lock & Unlock

- In the producer-consumer problem,

```
static pthread_mutex_t cs_mutex = PTHREAD_MUTEX_INITIALIZER;
```

PRODUCER

```
item   nextProduced;

while (TRUE) {
    while (counter == BUFFER_SIZE);
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;

    pthread_mutex_lock( &cs_mutex );

    counter++;

    pthread_mutex_unlock( &cs_mutex );
}
```

CONSUMER

```
item  nextConsumed;

while (TRUE) {
    while (counter == 0);
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    pthread_mutex_lock( &cs_mutex );

    counter--;

    pthread_mutex_unlock( &cs_mutex );
}
```

entry

exit

Critical section

No one else can use the CS

C.S. Requirements *

1. **Mutual Exclusion**

   If a process P_i is executing in its C.S., then no other processes can be executing in their C.S.

2. **Progress**

   If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the process that will enter the critical section next cannot be postponed indefinitely – **deadlock free**

3. **Bounded Waiting**

   Each process should be able to enter its C.S. after a finite number of trials – **starvation free**

Chapter 6: Synchronization

- Background
- The Critical-Section Problem
- Software C.S.: Peterson’s Solution
- Hardware C.S.: Mutex Locks
- Semaphores
- Classic Problems of Synchronization

C.S. Algorithm

- Shared variables
  - array wants [] : true or **false**
    - wants[i] indicates if P_i wants to enter C.S.

```
do {
        wants[ i ] = true;
        while (wants[ j ]) {;}
                CRITICAL SECTION
        wants[ i ] = false;
                REMAINDER SECTION
} while (TRUE);
```

Algorithm (cont.)

- Consider two P_i where i = {1, 2}, i.e., P_1, P_2

```
do {
        wants[ i ] = true;
        while (wants[ j ]) {;}
                CRITICAL SECTION
        wants[ i ] = false;
                REMAINDER SECTION
} while (TRUE);
```

- **Process** P_1

```
do {
        wants[ 1 ] = true;
        while (wants[ 2 ]) {;}
                CRITICAL SECTION
        wants[ 1 ] = false;
                REMAINDER SECTION
} while (TRUE);
```

- **Process** P_2

```
do {
        wants[ 2 ] = true;
        while (wants[ 1 ]) {;}
                CRITICAL SECTION
        wants[ 2 ] = false;
                REMAINDER SECTION
} while (TRUE);
```

- What happens if both wants[1] and wants[2] are true?

Peterson's Solution

- A **software-based** solution
  - Solution to CS problem w/o H/W support

- Suppose two processes : P_i (=P_0), P_j (=P_1)
  The two processes share two variables:
  - bool **wants**[i]; – wants[i] indicates if P_i wants to enter C.S.
  - int **not_turn**; – not this thread's turn to enter C.S. Other threads can enter C.S. if they want to…

```
for (;;) { /* assume i is thread number (0 or 1) */
    wants[i] = true;
    not_turn = i;            /* "Not my turn, you go first!" – yield (양보) */
    while (wants[j] && not_turn == i)
     ; /* other thread wants in and not our turn, so loop */
    CRITICAL SECTION
    wants[i] = false;        /* I'm finished, so others can enter now */
    REMAINDER SECTION
}
```

Algorithm for Process P_i

```
while (true) {
 // entry section
    wants[i] = TRUE;
    not_turn = i;
    while ( wants[j] && not_turn == i);

    CRITICAL SECTION
 // exit section
    wants[i] = FALSE;

    REMAINDER SECTION
}
```

Peterson's algorithm

- Peterson's algorithm

Peterson's algorithm

P0

```
repeat
......
wants[0] ← true;
not_turn ←  0;
while (wants[1] and
        not_turn == 0) do
endwhile;
```

P1

```
repeat
......
wants[1] ← true;
not_turn ←  1;
while (wants[0] and
        not_turn == 1) do
endwhile; ←
```

wants[0] wants[1]

not_turn

Critical Section

P0

```
wants[0] ← false;
......
until (false);
```

P1

```
wants[1] ← false;
......
until (false);
```

Does Peterson’s solution work?

```
for (;;) { /* assume i is thread number (0 or 1) */
    wants[i] = true;
    not_turn = i;
    while (wants[j] && not_turn == i)
            ; /* other thread wants in and not our turn, so loop */
    CRITICAL SECTION
    wants[i] = false;
    REMAINDER SECTION
}
```

- **Mutual exclusion?**
  - can’t both be in C.S. - Would mean wants[0] == wants[1] == true and – both cannot be in critical section!
  - Also, **not_turn** would have blocked one thread from C.S.

Does Peterson's solution work?

```
for (;;) { /* assume i is thread number (0 or 1) */
    wants[i] = true;
    not_turn = i;
    while (wants[j] && not_turn == i)
        ; /* other thread wants in and not our turn, so loop */
    CRITICAL SECTION
    wants[i] = false;
    REMAINDER SECTION
}
```

- **Progress (Deadlock free)? Bounded–waiting?**
  - Can P_i be stuck in while-loop (wants[j]==true && not_turn==i) forever?
    - Case1: P_j is does not want to enter C.S then wants[j] == false
    - Case2: P_j wants to enter C.S then wants[j] == true, but not_turn == i or j
      - if not_turn == i then j enters C.S, if not_turn == j then i enters C.S.
  - Since P_i does not change the value of the 'not_turn' while executing the while statement, P_i will enter the CS (progress) after at most one entry by P_j (bounded waiting).
  - Peterson's solution considers strict alternation so, alternatively Pi and Pj will get access to critical section.

Chapter 6: Synchronization

- Background
- The Critical-Section Problem
- Software C.S.: Peterson's Solution
- Hardware C.S.: Mutex Locks
- Semaphores
- Classic Problems of Synchronization

Hardware Approach

- Peterson's algorithm is expensive, only works for 2 processes
- Peterson's algorithm is a software solution
  - Low speed
- Many systems provide **hardware support** for C.S.
  - All solutions below based on idea of protecting critical regions via **locks**
- Modern machines provide special atomic hardware instructions
  - **Atomic** = non-interruptible

> **Note**
>
> **Characteristics of a machine instruction**
>
> - Atomicity, indivisibility
>
> (No interrupt during the execution of a machine instruction)

Synchronization Hardware: TestAndSet

- Mutual exclusion with **TestAndSet()** instruction

Initially
lock = FALSE;

```
do {
        while ( TestAndSet (&lock ) )
              ;   // do nothing

        CRITICAL SECTION

        lock = FALSE;

        REMAINDER SECTION

} while ( TRUE);
```

```
boolean TestAndSet(boolean *target) {
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

Atomic (indivisible) operation Supported by H/W

Uninterruptible
Machine Instructions

Mutex Locks

- Hardware based solutions are complex and unavailable to the application programmer
- **Mutex lock**
  - A simple software tool to solve the C.S problem
- Product critical regions with it by first **acquire()** a lock then **release()** it

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

```
release() {
   available = true;
}
```

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

Mutex Locks are provided as **API**

Hardware Approach: Mutex Locks

- Calls to **acquire()** and **release()** must be atomic
  - Usually implemented via <u>hardware</u> atomic instructions
    - e.g., TestAndSet

```
// initialization
mutex->available = 0;

// acquire using test_and_set()
void acquire(lock *mutex) {
      while (test_and_set(&mutex->available) != 0)
            ;
      return;
}

void release(lock *mutex) {
      mutex->available = 0;

      return;
}
```

- But this solution requires [busy waiting]

***Busy waiting** is a process synchronization technique in which a process/task waits and <u>constantly checks</u> for a condition to be satisfied before proceeding with its execution.

Example Code (POSIX pthread)

- Example Code For Critical Sections with POSIX pthread library

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

Busy waiting in Mutex

- **Busy waiting**
  - loop continuously while waiting
  - (=**spinlock**): spins while waiting for the lock to become available

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

- Cons
  - Can do nothing while waiting
  - Waste CPU cycles – some other process could have used it
- Alternative: Process goes to “waiting” state and context switch to another process
- Pros
  - No context switch is required during spinlock
  - Useful when locks are expected to be held for short times
    - tradeoff: context switch time (waiting) vs. spinlock time

Chapter 6: Synchronization

- Background
- The Critical-Section Problem
- Software C.S.: Peterson's Solution
- Hardware C.S.: Mutex Locks
- Semaphores
- Classic Problems of Synchronization

Semaphore

- 1972 Turing Award
- ACM Dijkstra Prize
- Known for Dijkstra Algorithm, Semaphore

- Proposed by Dijkstra in 1965
- Semaphore ***S*** –integer variable
- Two atomic standard operations modify S: **wait()** and **signal()**

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

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

will see other version later

Compare this with Mutex Lock

- Only one process can modify the semaphore value (S)
  - S<=0, S-- and S++ are atomic operations – hardware support

Semaphore

- Semaphore provides more sophisticated ways than mutex lock for synchronization
- **Binary semaphore**
  - The semaphore can be set to 0 or 1,   i.e., S= 0 or 1
  - Same as **Mutex locks**
- **Counting semaphore**
  - The semaphore can initially have nonnegative integer values, i.e., the ***number of resources available***
  - Used for solving producer-consumer problems and etc.

Binary Semaphore

- Provides mutual exclusion

- **Semaphore S – shared by all processes**
- **Process P_i**

Semaphore S; // initialized to 1

```
do {

      wait (S);

        //Critical Section

      signal (S);

        //Remainder Section

} while (true);
```

- *N processes* shares semaphore S
- *S* is initialized to 1 (# of resources?)

P1	S	P2
wait(S) {	1	
while(S<=0) ; //pass		
S--; }	0	
//Critical Section		wait(S) {
		while(S<=0) ; //wait
signal(S) {		
S++; }	1	//pass
//Remainder Section	0	S--; }
		//Critical Section
	1	signal(S) {
		S++; }
		//Remainder Section

Q: what happens if there are 2 resources?
→ counting semaphore

Problem 1: Deadlocks and Starvation

- **Deadlock**
  - two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

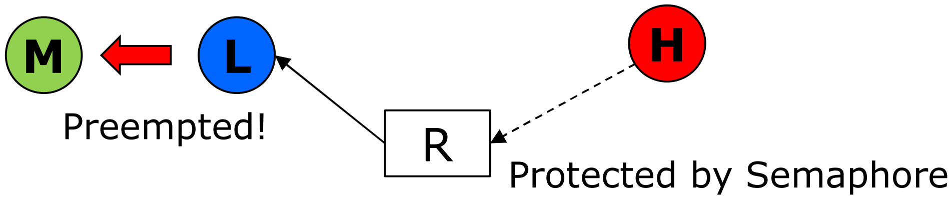
P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

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

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

Problem 2: Priority Inversion

- Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Example: 3 process **L**, **M**, **H** with scheduling priorities **L < M < H** (**L** has lowest priority, **H** has highest)

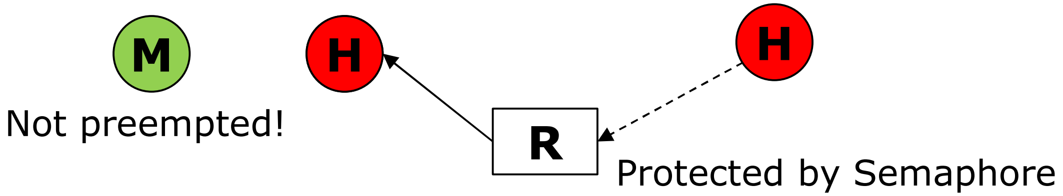


  - Resource **R** is protected by a Semaphore
  - Indirectly, Process **M** has affected Process **H** 's waiting time for resource R! – **Priority Inversion!**
    - **M** → **L** → **H**

Here, we can see that running of **M** has delayed the running of both **L** and **H**. Precisely speaking, <u>**H** is of higher priority and doesn't share **R** with **M**; but **H** had to wait for **M**.</u>

Problem 2: Priority Inversion

- Priority Inversion solved via **priority-inheritance protocol**
  - A process (process L) that is accessing resources needed by a higher-priority process (process H) inherits the higher priority (H) until they are finished with the resources



Problem 3: Incorrect Use of Semaphores

- Correct us of semaphore operations may not be easy.
- Suppose semaphore variable called mutex is initialized to 1.
- Incorrect use of semaphore operations:
  1) signal (mutex) …. wait (mutex)
     - What happens?
  2) wait (mutex) … wait (mutex)
     - What happens?
  3) Omitting of wait (mutex) or signal (mutex) (or both)
     - **Mutual exclusion is violated (1)** or **deadlock will occur (2)** or **both (3)**

```
do {
    wait (S);
      //Critical Section
    signal (S);
      //Remainder Section
} while (true);
```

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

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

Semaphore with Block Operation

- Avoid **busy waiting** (spinlock)
- **block()**
  - If S<=0 then wait using block
  - Instead of using busy waiting, the process is placed into **waiting queue** (process state?)
  - CPU scheduler selects another process (in ready queue) to execute
- **wakeup()**
  - The blocked process restarts when some other process executes a **signal()** operation
  - The blocked process is moved from waiting queue to ready queue

Counting Semaphore with Block Operation

```
typedef struct {
        int value;
        struct process *list;
} semaphore
```

- **Implementation of *wait()*:**

```
wait (S) {
    S.value--;
    if (S.value < 0) {
        add this process to waiting queue
        block();
    }
}
```

- **Implementation of *signal()*:**

```
signal (S) {
    S.value++;
    if (S.value <= 0) {
        remove a process P from the waiting queue
        wakeup(P);
    }
}
```

Pthreads Semaphore

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

```
/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```

Chapter 6: Synchronization

- Background
- The Critical-Section Problem
- Software C.S.: Peterson's Solution
- Hardware C.S.: Mutex Locks
- Semaphores
- Classic Problems of Synchronization

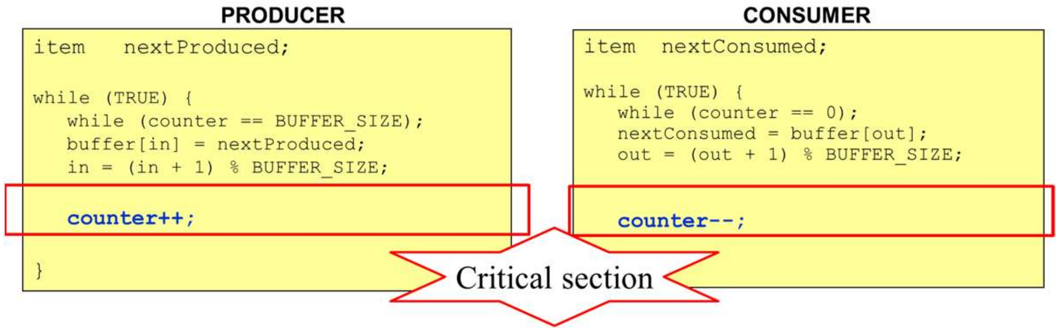
Classical Problems of Synchronization

- Semaphore-based concurrent programming
  - Bounded-Buffer Problem
  - Reader-Writer problem
  - Dining philosopher problem
  - Etc.

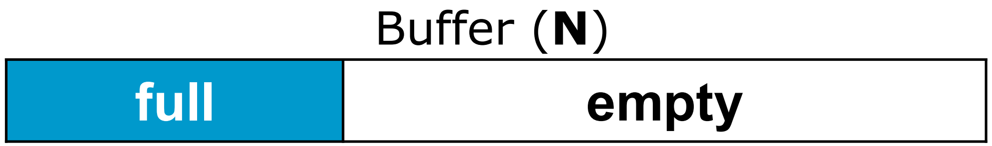
The Bounded-Buffer Problem

- Producer processes
- Consumers processes

- **N** size buffer (Limited buffer)
  - Buffer can hold up to **N** items



- Solutions with semaphore – use 3 semaphores
  - Semaphore **mutex** = 1 // critical section
  - Semaphore **full** = 0 // number of items in buffer
  - Semaphore **empty** = N // number of empty slots in buffer
    - full + empty = N



Bounded Buffer Problem (Cont.)

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

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

- The structure of the <u>producer</u> process

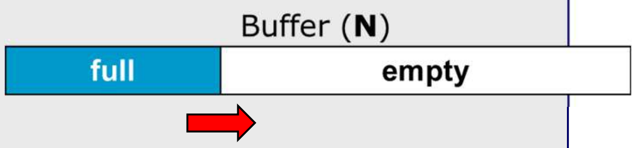
```
do
{
        //   Produce an Item

        wait (empty);              // Check if buffer is full
        wait (mutex);              // Enter into Critical Section

      // Critical Section
      //  add an item to buffer

        signal (mutex);                        // Leave C.S.
        signal (full);                         // Produce an Item

} while (TRUE);
```



Bounded Buffer Problem (Cont.)

- The structure of the <u>consumer</u> process

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

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

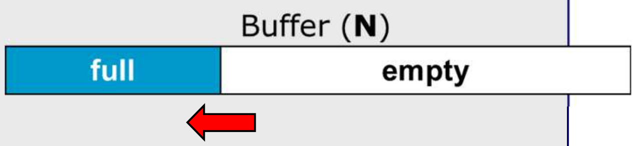
```
do
{

          wait (full);          // Check if buffer is empty
          wait (mutex);         // Enter into Critical Section


        // Critical Section
        //  remove an item from  buffer


          signal (mutex);                    // Leave C.S.
          signal (empty);                    // Consume an Item


} while (TRUE);
```



Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write.

- Problem
  - allow **multiple readers** to read at the same time.
  - **One single writer** can access the shared data at the same time.

- Shared Data

  To protect readcount!

  - Data set
  - Semaphore **mutex** initialized to 1 // critical section
  - Semaphore **wrt** initialized to 1. // mutual exclusion for writers
  - Integer **readcount** initialized to 0. // number of readers

Readers-Writers Problem (Cont.)

- The structure of a <u>writer</u> process

```
do
{
        wait (wrt);

        // writing is performed

        signal (wrt);

} while (TRUE);
```

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

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

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
        wait (mutex);
        readcount ++;
        if ( readcount == 1 )
                wait(wrt);
        signal(mutex)

        // reading is performed

        wait(mutex);
        readcount --;
        if ( readcount == 0 )
                signal(wrt);
        signal (mutex);
} while (TRUE);
```

writer process

```
do
{
        wait (wrt);

        // writing is performed

        signal (wrt);

} while (TRUE);
```

Conclusion

- Processes in multiprogramming systems
  - Asynchronous, concurrent
  - Needs mutual exclusion (ME) and process synchronization mechanisms
- SW solution for ME and synchronization
  - Peterson's algorithm
- HW solution for ME and synchronization
  - Test-and-Set instruction (atomic)
- Semaphore with no busy waiting