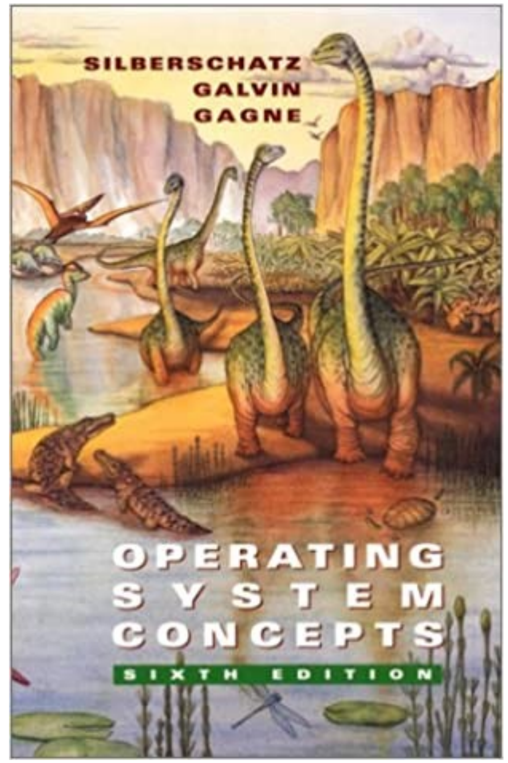
Chapter 6: Synchronization Tools

School of Computing, Gachon Univ.
Joon Yoo



Chapter 6: Synchronization

- Background
- The Critical-Section Problem
- Software C.S.: Peterson’s Solution
- Hardware C.S.
- Mutex Lock & Semaphores

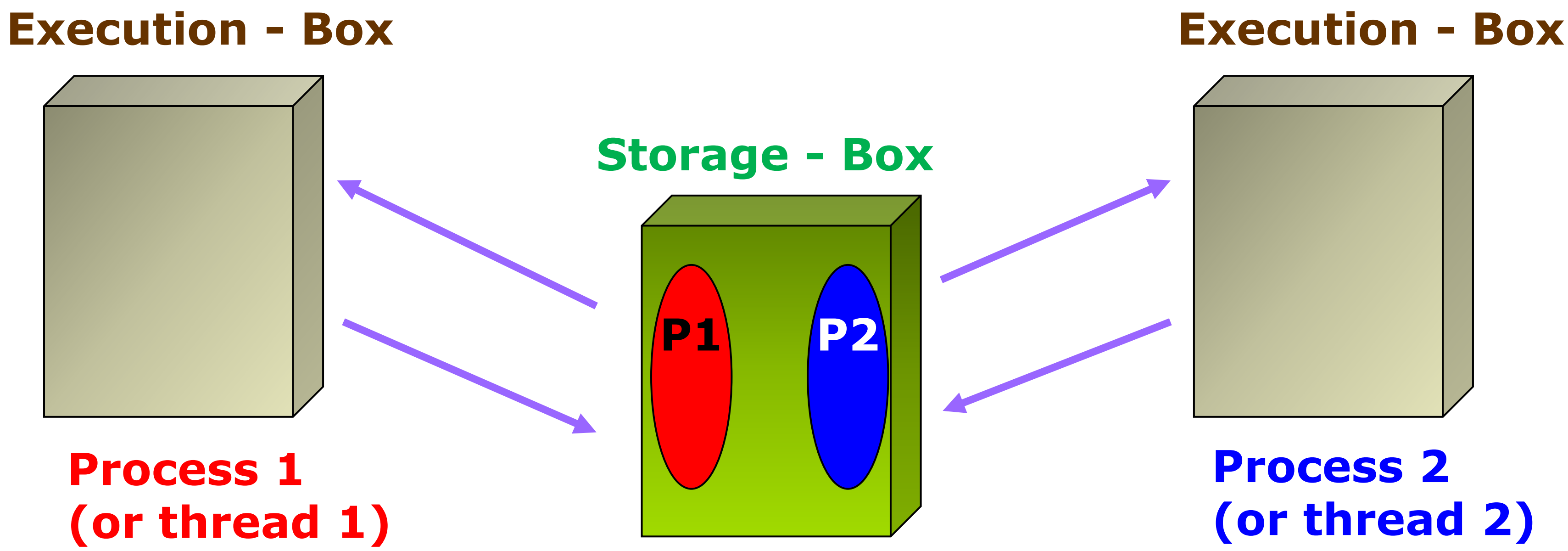
Objectives

- Describe the critical-section problem and illustrate race condition

  보여준다

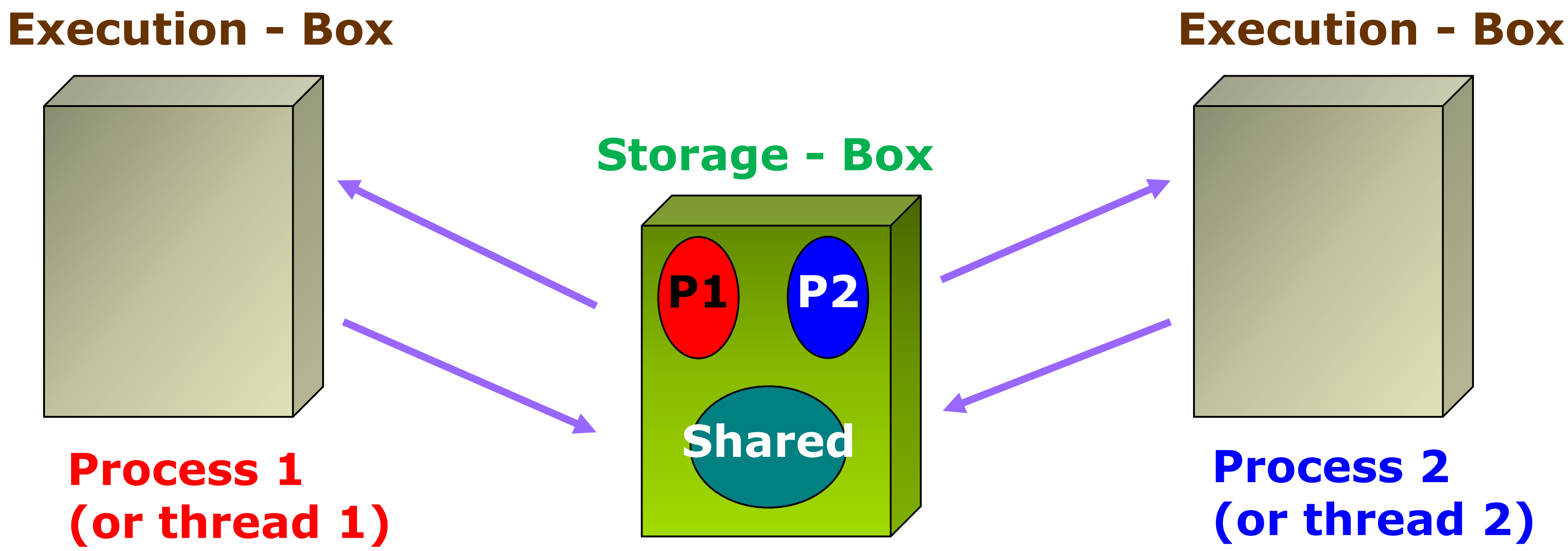
- To present both software and hardware solutions of the critical-section problem

2 Processes using each memory space



- S-Box (Memory), E-Box (CPU Process)

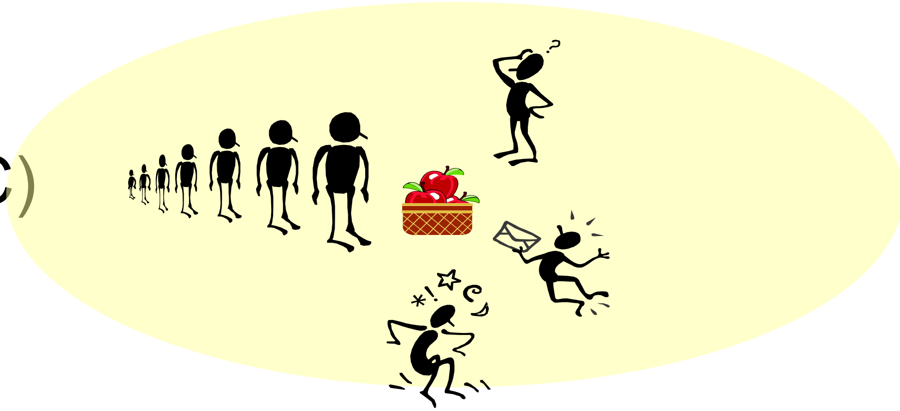
2 Processes sharing memory space



- S-Box (Memory), E-Box (CPU Process)
  - Case 1: two processes running in parallel on a multicore CPU
  - Case 2: two processes running concurrently on single core

Shared Memory

- Shared Memory
  - Inter-process Communication (IPC)
  - Multi-threads



- Why do we need process synchronization?
  - Concurrent or parallel access to shared data may result in data inconsistency
    - Recall the multithread example in Ch. 4:

Output:
x is 2
x is 3

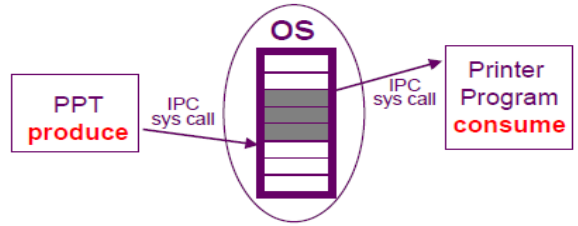
Output:
x is 3
x is 2

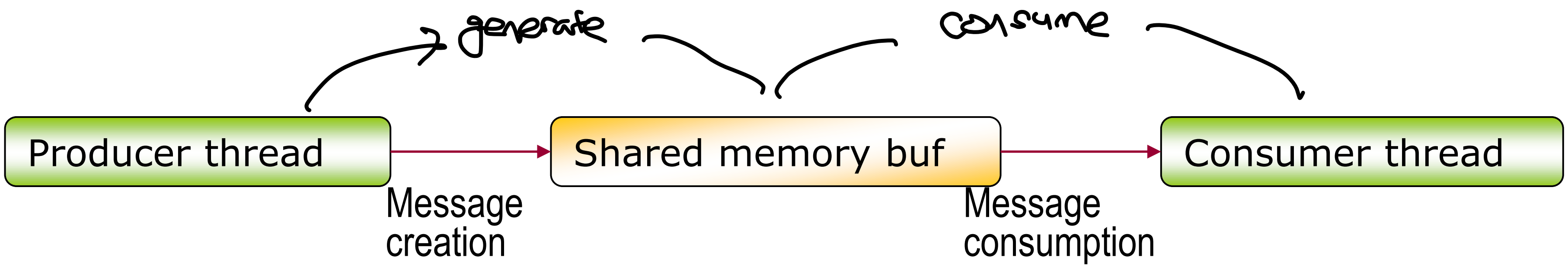
Output:
x is 3
x is 3

Output:
x is 2
x is 2

Example: Producer-Consumer problem

- **Producer-consumer problem** (Ch.3.6 : Shared memory)
  - Producer threads
    - Set of threads that generates data
  - Consumer threads
    - Set of threads that consumes data





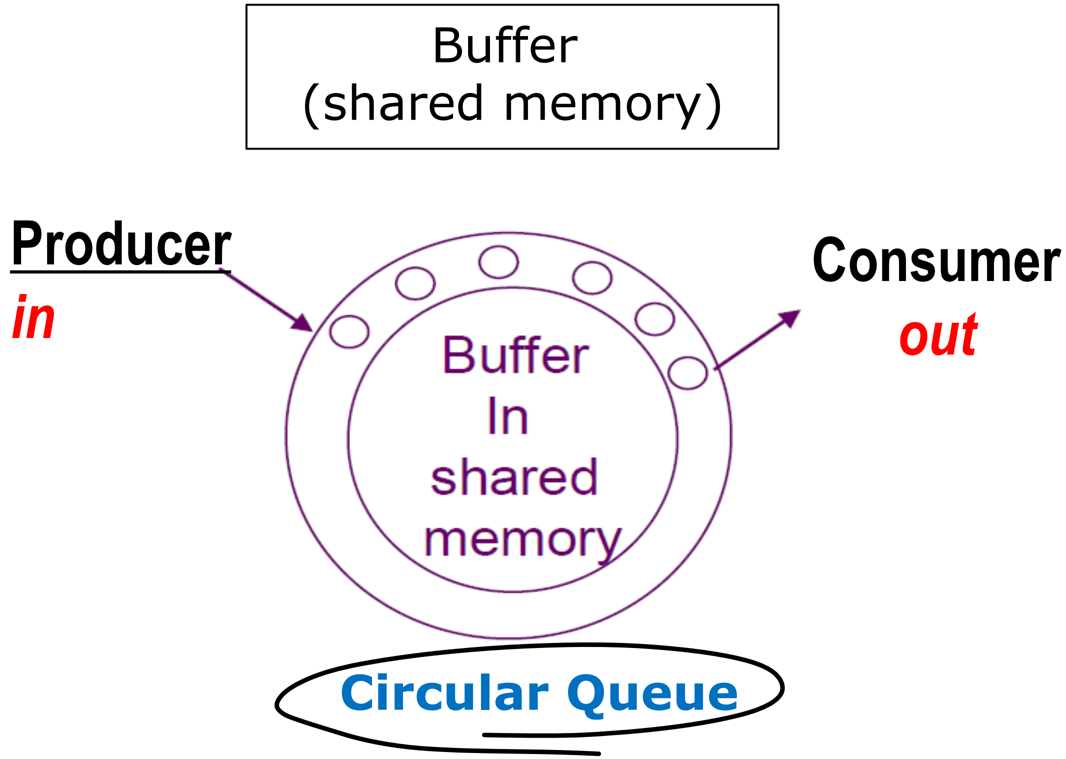
Shared-Memory Systems

- The producer and consumer must be ***synchronized*** so that
  - **Producer** does not try to produce an item when **buffer** (shared memory) is *full* buffer이 full일 경우 produce하지 않아야 한다.
  - **Consumer** does not try to consume an item that has not yet been produced (i.e., buffer is *empty*)

buffer가 empty일 경우 consume하지 않아야 한다.

Shared Memory (Ch. 3.6)

```
#define BUFFER_SIZE 6

typedef struct
{
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;  //next free position in buffer  = enqueue
int out = 0; // first full position in buffer = dequeue
```



Producer-Consumer problem

int counter=0; // global variable ← Number of items in buffer

또한 buffer[] 도 share중

공유데이터

PRODUCER thread = shared data

consume을

```
item   nextProduced;

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

→ True이면 이 문장을 계속 돌면서 기다리게 된다

CONSUMER thread

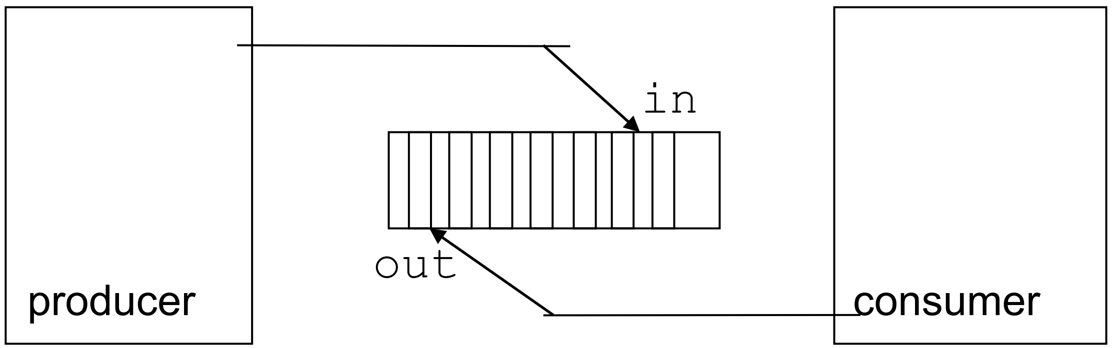
produce를 마칠때까지 True면 기다림

```
item  nextConsumed;

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

arr[100],

100%100 = 0

101%100 = 1



C code → Assembly code

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

  $register_1$ = counter #load from memory 메모리에서 로드
  $register_1 = register_1 + 1$ 연산
  counter = $register_1$ #store in memory 메모리에 저장
) 3단계

- 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?

정상적

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**}

Producer-Consumer problem: 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 그런데 이렇게 중간에 context switch가 발생 시

Context switch P1→P2

Context switch P2→P1

Context switch P1→P2

T_0: producer execute $register_1 = counter$ $\{register_1 = 5\}$ race-c

T_1: producer execute $register_1 = register_1 + 1$ $\{register_1 = 6\}$ 발생

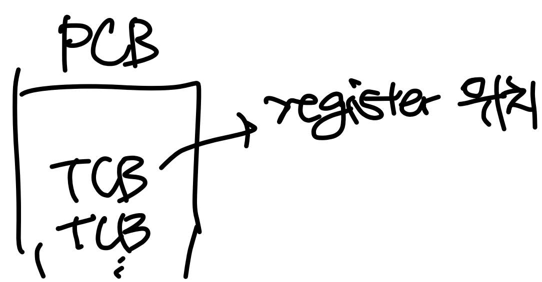
T_2: consumer execute $register_2 = counter$ $\{register_2 = 5\}$

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 **<u>race condition</u>**



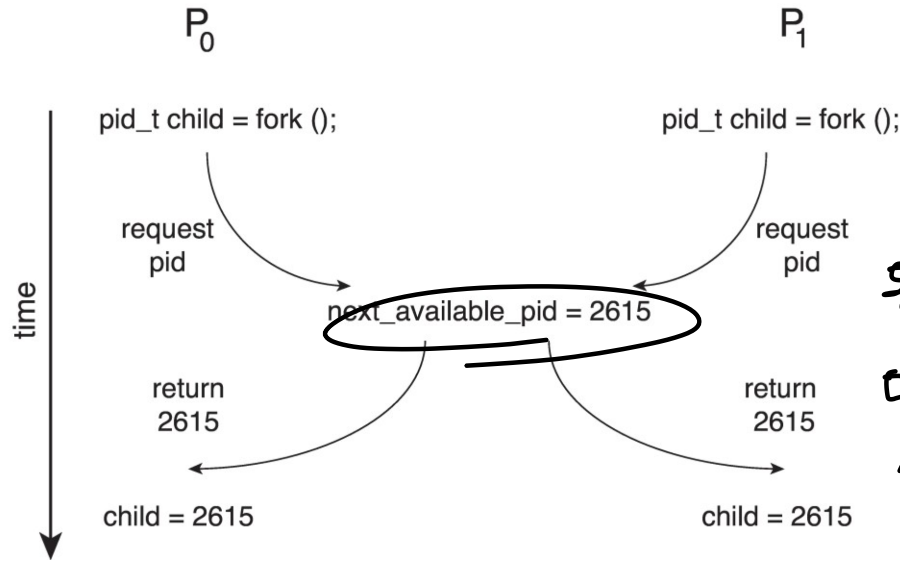
Race Condition

- **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*

동기화 필요함

Race Condition

- Processes P_0 and P_1 are creating child processs using the `fork()` system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)



동시에 fork(); 실행시
다른 프로세스인데 pid가 같아진다.
?!

- Unless there is mutual exclusion, the same pid could be assigned to two different processes!

Chapter 6: Synchronization

- Background
- The Critical-Section Problem
- Software C.S.: Peterson’s Solution
- Hardware C.S.
- Mutex Lock & Semaphores

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 (data structure or device)

공유데이터를 사용하는 코드 = Critical Section

- 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**

entry가 동시에 일어나면...??

P1

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

다른 process가 CS를
사용하고 있는가?

No one else can use the CS

연산수행 → P2 entry section

CS사용중이므로 entry X

CS사용가능 entry O

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 );      entry

   counter++;

   pthread_mutex_unlock( &cs_mutex );    exit
}
```

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

Chapter 6: Synchronization

- Background
- The Critical-Section Problem
- Software C.S.: Peterson's Solution C code
- Hardware C.S.
- Mutex Lock & Semaphores

CS완벽하게 구현되지는 않음!

C.S. Algorithm 1

wants[1] = F
wants[2] = F

- Shared variables
  - array wants [] : true or **<u>false</u>**
    - 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);
```

entry

다른 wants[ ]가 True라면 while을 돌면서 기다림

exit

Algorithm 1 (cont.)

— initialize —
w[1] = F
w[2] = F

1. Mutual Exclusion

- **Process P_1**

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

interrupt

wants[1] = T
wants[2] = T

- **Process P_2**

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

동시에 CS를 들어올 수 없다

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

서로 CS 사용중도 아닌데
서로 계속 기다려야 함
→ progress를 이루지 못함

1. Mutual Exclusion? O 상호배제
2. Progress? X

C.S. Requirements

1. Mutual Exclusion
상호배제

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

2. Progress
CS가 사용중이 아닐경우, 사용할수 있어야함

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**

deadlock 상태

3. Bounded Waiting
기다리는 시간이 적절해야함

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

process가 적당히 기다리면 CS 수행할 수 있어야함

C.S. Algorithm 2

- Shared variable
  - variable not_turn: 0 or 1
    - take turns in entering C.S.

```
do {
        while (not_turn==i) {;}
                CRITICAL SECTION
        not_turn=i;
                REMAINDER SECTION
} while (TRUE);
```

Algorithm 2 (cont.)

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

int → 0

not_turn = 2

- **Process P_1**

```
do {

        while (not_turn==1) {;}
                CRITICAL SECTION
        not_turn=1;
                REMAINDER SECTION
} while (TRUE);
```

- **Process P_2**

```
do {
만약 P2만 실행할 경우, progress가 이뤄지지 않는다.
        while (not_turn==2) {;}
                CRITICAL SECTION
        not_turn=2;
                REMAINDER SECTION
} while (TRUE);
```

- What happens if only P_1 wants to enter C.S.?

↳ 다른 Process가 실행되지 않는 한, P_1는 실행되지 않을 가능성이 있다.

1. **Mutual Exclusion?** O
2. **Progress?** X

Algorithm 1+2 ⇒

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 Ti 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
}
```

AND

만약 wants[i]가 True인데
not_turn이 i가 아니라면
실행가능하다.

역의 상황도 마찬가지

Peterson's algorithm

- Peterson's algorithm

not_turn도 결국 공유데이터 아닌가? → 그래서 Peterson이 불완전함

algorithm 1 : interrupt, wants[0]=F, wants[0]=T

Peterson's algorithm

P0

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

Critical Section

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

wants[0]: T

wants[1]: T

not_turn: 0 or 1

P1

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

Critical Section

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

P_0 P_2 둘중 하나는 반드시 실행된다~

Peterson's algorithm

- Peterson's algorithm

algorithm 2: execute only one process.

Peterson's algorithm

P0

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

wants[0] T

wants[1] F

not_turn 0

P1

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

Critical Section

P0:

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

P1:

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

P1이 안들어왔기 때문에, 그냥 P_0이 실행된다

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)?**
  - 애초에 안들어갈경우 false
  - Can Pi be stuck in while-loop (wants[j]==true && not_turn==i) forever?
    - Case1: Pj is does not want to enter C.S then wants[j] == false
    - Case2: Pj 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.

둘중하나는 들어간다

Does Peterson's solution work?

- Peterson's Solution is not guaranteed to work on modern architectures. 현대
  - Interrupt may happen any time – leading again to race condition
  - Only works for 2 processes
  - **Instruction reordering** may result in failed mutual exclusion (next slide)
  - However, useful for demonstrating an algorithm

Instruction Reordering

- To improve system performance, processors and/or compilers may reorder read/write operations that have no dependencies
- Instruction reordering may result in failed mutual exclusion!

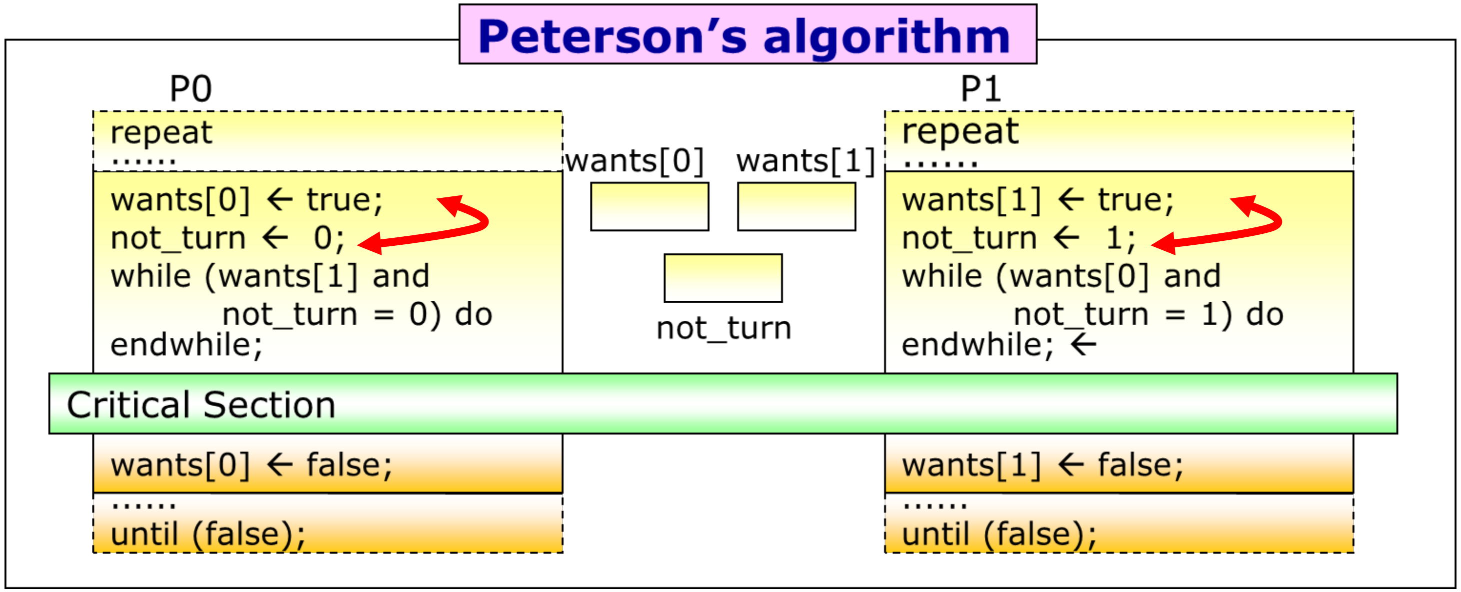
```
while (true) {
 // entry section
   wants[i] = TRUE;
   not_turn = i;
   while ( wants[j] && not_turn == i);

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

   REMAINDER SECTION
}
```

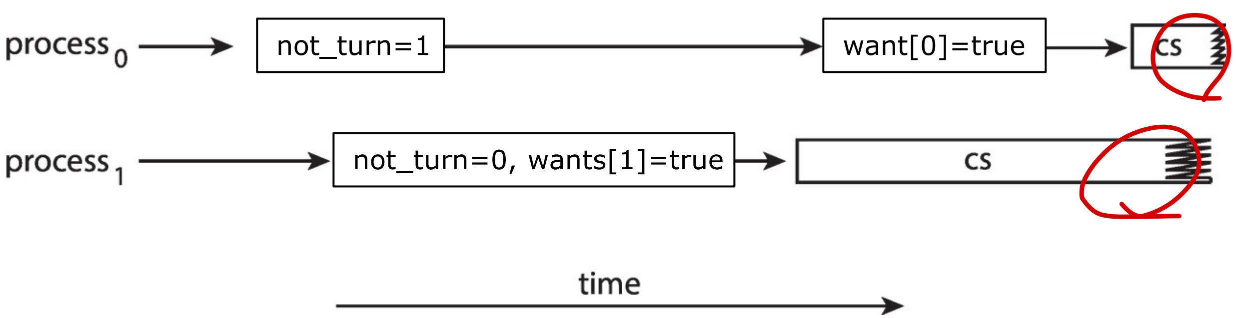
위치 변경되어도
dependency X
이어야 하는데

이걸 entry section을 사용하게 되면
문제가 발생할 수 있음..

이것이 Instruction Reordering!

Peterson's solution not working: Instruction Reordering





둘다 CS 접근...?

- This allows both processes to be in their critical section at the same time!

Chapter 6: Synchronization

- Background
- The Critical-Section Problem
- Software C.S.: Peterson's Solution
- Hardware supported C.S.
- Mutex Lock & Semaphores

Hardware Support: Memory Barriers

- A **memory barrier** is an instruction that forces any change in memory to be propagated (made visible) to all other processors
  - All loads and stores are completed before any subsequent load or store operations are performed
- Example: Assignment to x occurs before the assignment to flag

```
x = 100;
memory_barrier();
flag = true
```

- In Peterson's Solution
  - If a memory barrier is placed between the first two assignments of the Peterson's solution, reordering can be avoided
  - However, memory barriers are very low-level operations typically only used by kernel developers

Hardware Support: Atomic Instructions

- Many systems provide **hardware support**
  - 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 operation Supported by H/W

Uninterruptible
Machine Instructions

Atomic Variables

- Typically, instructions such as test-and-set are used as building blocks for other synchronization tools.
- One tool is an **atomic variable** that provides *atomic* (uninterruptible) updates on basic data types such as integers and Booleans.
- For example, the **`increment()`** operation on the atomic variable **`sequence`** ensures **`sequence`** is incremented without interruption:

```
increment(&sequence);
```

Chapter 6: Synchronization

- Background
- The Critical-Section Problem
- Software C.S.: Peterson's Solution
- Hardware C.S.
- Mutex Lock & Semaphores

Mutex Locks

- **Atomic instructions** are low-level language thus unavailable to the high-level language application programmer
- **Mutex lock** (**Mut**ual **ex**clusion)
  - A simple API tool
- Example Code For Critical Sections with POSIX pthread library

Mutex Locks are provided as **API**

```
#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);
```

Mutex Locks

```
while (true) {
        acquire lock

        critical section

        release lock

        remainder section
}
```

Mutex Locks

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

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

These two functions must be implemented **atomically**.
test-and-set can be used to implement these functions.

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(S)** and **signal(S)**

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

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

will see other version later

- Only one process can modify the semaphore value (S)
  - **wait(S)** and **signal(S)** are **atomic instructions**

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 (=Mutex lock)

- 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
- *Initial S* = 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

Counting Semaphore (2 resources)

- Provides mutual exclusion

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

Semaphore S; // initialized to 2

```
do {

    wait (S);

      //Critical Section

    signal (S);

      //Remainder Section

} while (true);
```

- *N processes* shares semaphore S
- *Initial S* = 2 (# of resources)

P1	S	P2
wait(S) {	2	
while(S<=0) ; //pass		
S--; }	1	
//Critical Section		wait(S) {
		while(S<=0) ;
	0	S--; }
signal(S) {		//Critical Section
S++; }	1	
//Remainder Section		signal(S) {
	2	S++; }
		//Remainder Section

Busy waiting

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

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

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

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

- 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

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

Semaphore **S.value=1**; *number of resources*

```
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 (value <= 0) {
        remove a process P from the waiting queue
        wakeup(P);
    }
}
```

Q1: Can S become negative?
Q2: What does this mean?

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);
```

Problem 1: Deadlocks

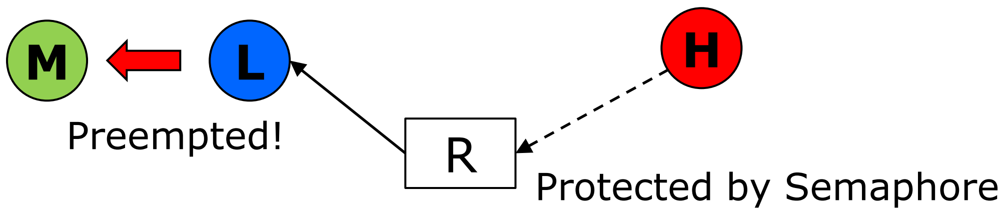
- **Deadlock** (More in Ch. 8)
  - two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

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

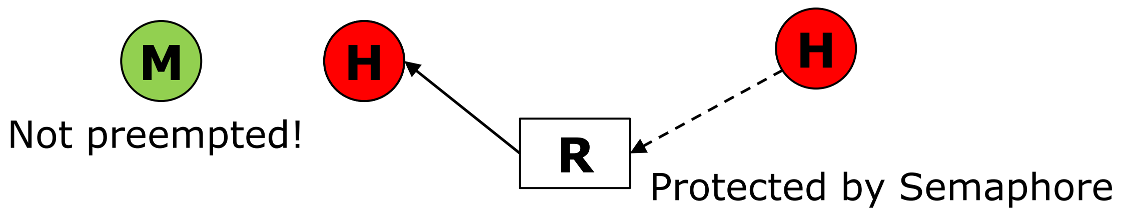
- Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Example: 3 process **L**, **M**, **H** with scheduling priorities **L** < **M** < **H** (**L** has lowest priority, **H** has highest)



  - Resource **R** is protected by a Semaphore
  - Indirectly, Process **M** (lower priority) has affected Process **H** 's (higher priority) waiting time for resource R! – **Priority Inversion!**

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



Priority Inversion in Mars Pathfinder (1997)

Problem 3: Incorrect Use of Semaphores

- 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++;
}
```

Read page 274!

Conclusion

- Multi-tasking (multi-threaded) systems with shared resources
  - Race condition happens!
  - Needs process synchronization mechanisms
- SW solution for ME and synchronization
  - Algorithm 1&2, Peterson's algorithm
- HW support solution: Atomic instruction
- Mutex lock & Semaphore: using HW support
- Other solutions: Monitor, Liveness, ...