6. Process (thread) Synchronization

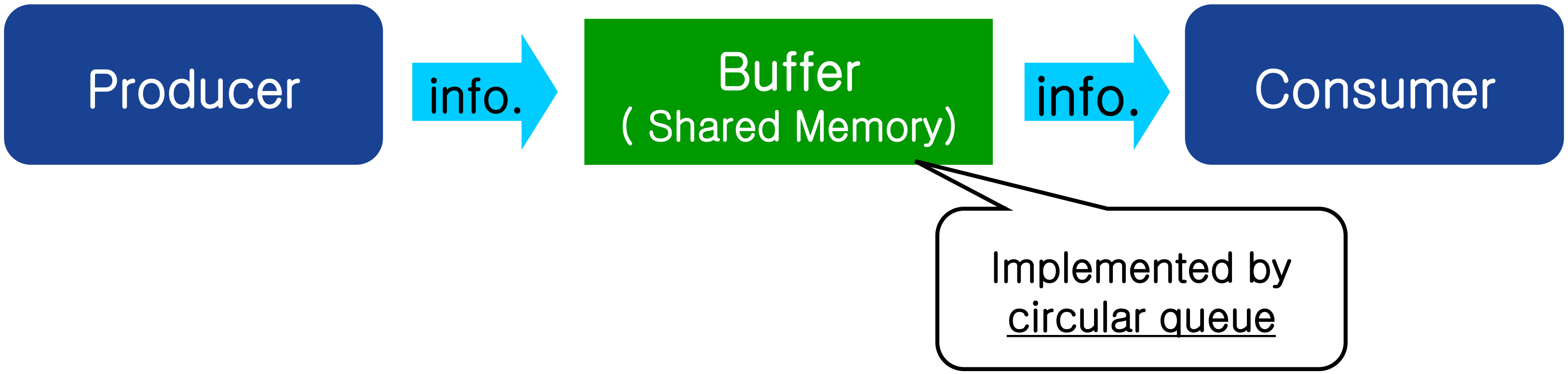
ECE30021/ITP30002 Operating Systems

Agenda

- Background
- The critical-section problem
- H/W Support for Synchronization
- Mutex and Semaphore
- Monitors
- Liveness

Background

- Process communication method
  - Message passing
  - Shared memory -> confliction can occur !!
- Producer-consumer problem
  - An example of communication through shared memory



Concurrent Access of Shared Data

- Producer queue insertion

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

현재 item의 개수

- Consumer

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

- Implementation of "**counter++**"

  $register_1$ = counter

  $register_1 = register_1 + 1$

  counter = $register_1$

- Implementation of "**counter--**"

  $register_2$ = counter

  $register_2 = register_2 - 1$

  counter = $register_2$

Switching can occur during modification operations !

Synchronization Problem

- Problematic situation
  - Initial value of counter is 5.
  - Producer increased counter, and consumer decreased counter concurrently.
  - Ideally, counter should be 5. But…

Producer	Consumer
$register_1$ = counter (*register1 = 5*)	
$register_1$ = $register_1$ + 1 (*register1 = 6*)	
	$register_2$ = counter (*register2 = 5*) $register_2$ = $register_2$ – 1 (*register2 = 4*)
counter = $register_1$ (*counter = 6*)	
	counter = $register_2$ (*counter = 4*)

counter can be 4 or 6 !

Process Synchronization

- Race condition
  - A situation, where several processes access and manipulate the same data concurrently
  - The outcome of execution depends on the particular order in which the access takes place.

- Synchronization: **the coordination of occurrences to operate in unison with respect to time.**

  Ex) ensuring only one process can access the shared data at a time

Agenda

- Background
- <u>The critical-section problem</u>
- H/W Support for Synchronization
- Mutex and Semaphore
- Monitors
- Liveness

The Critical-Section Problem

- Critical section problem: **designing a protocol that processes can use to cooperate**
  - *n* processes $P_0, \cdots, P_{n-1}$ are running on a system concurrently. Each process want access a shared data
  - The code of each process is composed of
    - Critical section: a segment of code which may change shared resources (common var., file, …)
    - Remainder section: a segment of code which doesn't change shared resources

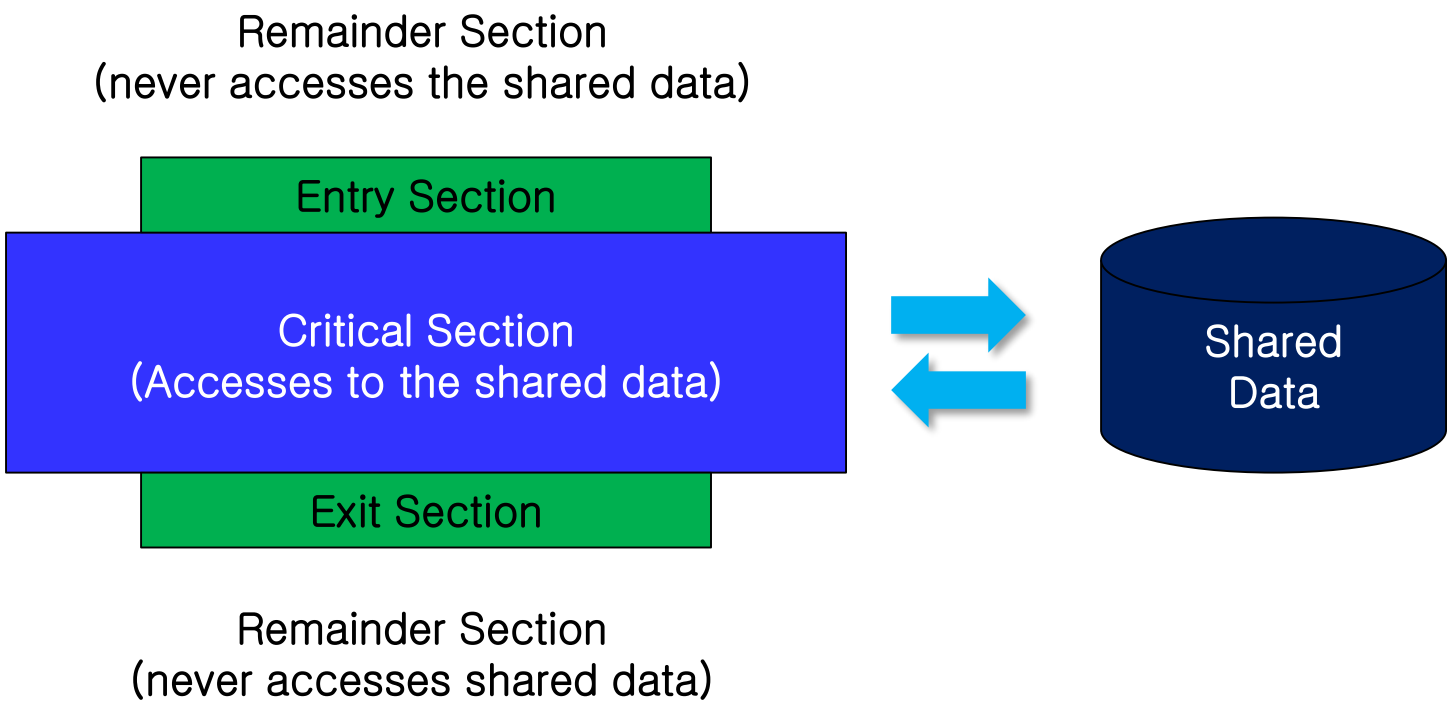
race condition 해결

    - Entry section: code section to request permission to enter critical section
    - Exit section: code section to notice it leaves the critical section

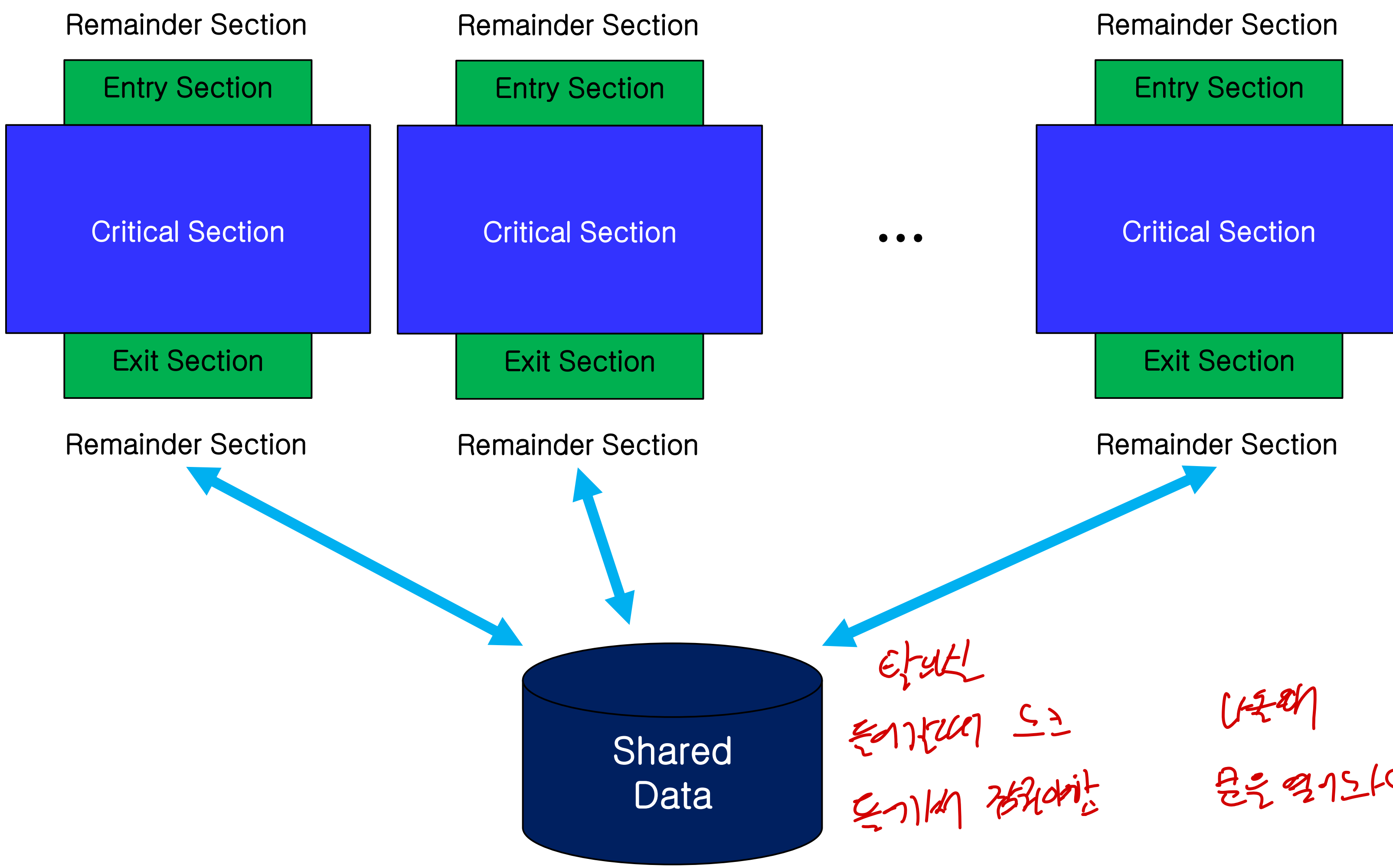
Critical Section

may access and

- A process modifies shared data only in the critical section
- At a time only one process can exist in its critical section.



Critical Section



탈의실
들어갈때 ㄷㄷ
동기화 정리해야함

나올때
문을 열고나와야함

The Critical-Section Problem

- General structure of typical processes

```
do {
    entry section

        critical section

    exit section

        remainder section

} while(TRUE);
```

The Critical-Section Problem

- Requirements of critical-section problem
  - Mutual exclusion: If a process is in its critical section, no other processes can be executing in their critical sections
    - -> Other processes should wait entry section에서 기다림
  - Progress: If no process is executing in its critical section, and some processes wish to enter their critical sections, only processes not executing in their remainder section can participate in the decision of next process to enter its critical section next. This selection cannot be postponed indefinitely.

    누가 잡은 건 상황이 아니라면 자연스럽게 하나가 들어간다.
  - Bounded waiting: There exists a bound or limit on number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before it is granted.

The Critical-Section Problem

- Kernel is an example of critical-section problem
  - Kernel data structures such as list of open files,

- Approaches
  - Non-preemptive kernel: switching cannot occur when a process is executing in kernel mode
    - Free from race condition

    Ex) Windows 2000, Windows XP, early version of UNIX, Linux prior to v. 2.6.
  - Preemptive kernel accompanied with a solution of critical-section problem
    - Suitable for real-time programming
    - More responsive

    Ex) Linux v. 2.6, Solaris, IRIX

Peterson's Solution

- A S/W solution for critical section problem
  ➔ No guarantee to work correctly on some architectures due to **load/store** instructions, but helps to understand the problem

- Two processes P_0 and P_1 (or P_i and P_j) other process
  current process
- Data items

```
int turn;           // indicates process allowed to execute
                    // in its critical section
                    // initial value is 0

boolean flag[2]; // if flag[i] is true, Pi is ready to enter
                    // its critical section
```

Erroneous Algorithm 1

- Algorithm 1
  - Process P_i

```
do {
        while (turn != i);
                critical section
        turn = j;
                remainder section
} while (1);
```

Erroneous Algorithm 1 progress 보장X

- Process P_0

```
do {
        while (turn != 0) ;
                critical section
        turn = 1;
                remainder section
} while (1);
```

- Process P_1

```
do {
        while (turn != 1) ;
                critical section
        turn = 0;
                remainder section
} while (1);
```

- **Mutual exclusion is guaranteed, but progress is not guaranteed.**

  Ex) P_1 is trying to enter its critical section, but P_0 is in its remainder section -> turn is not switched to 1

Erroneous Algorithm 2

- Algorithm 2
  - Process P_i

```
do {
        flag[i] = true;
        while (flag[j]) ;
                critical section
        flag[i] = false;
                remainder section
} while (1);
```

Erroneous Algorithm 2

progress 보장 X

- Process P_0

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

- Process P_1

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

- Mutual exclusion is guaranteed, but progress is not guaranteed.

  Ex) P_0 and P_1 enter simultaneously. Both of flag[0] and flag[1] can be true

Peterson's Solution

- Peterson's Solution
  - Process P_i

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] and turn == j);
        critical section
    flag[i] = false;
        remainder section
} while (1);
```

  - Process P_j

```
do {
    flag[j] = true;
    turn = i;
    while (flag[i] and turn == i);
        critical section
    flag[j] = false;
        remainder section
} while (1);
```

- Satisfies the three conditions

Peterson's Solution

- Peterson's Solution
  - Process P_0

```
do {
    flag[0] = true;
    turn = 1;
    while (flag[1] and turn == 1);
        critical section
    flag[0] = false;
        remainder section
} while (1);
```

  - Process P_1

```
do {
    flag[1] = true;
    turn = 0;
    while (flag[0] and turn == 0);
        critical section
    flag[1] = false;
        remainder section
} while (1);
```

- Satisfies the three conditions

Peterson's Solution

- Proof of mutual exclusion
  - If both P_i and P_j enter their critical section, it means
    - flag[0] = flag[1] = true
    - turn can be either 0 or 1, but turn cannot be both
- Proof of progress and bounded waiting
  - Blocking condition of P_i: flag[j] == true and turn == j
    - If P_j is not ready to enter critical section: flag[j] == false

    -> <u>P_i can enter critical section</u>
    - If P_j waiting in its while statement, turn is either i or j
    - If turn is i, <u>P_i will enter critical section</u>.
    - Otherwise, P_j will enter critical section.

    -> When P_j exits critical section, P_j sets flag[j] to false and <u>P_i can enter critical section</u> because P_j modifies turn to i.
    - Therefore, P_i will enter critical section (Progress) and waits at most one process (Bounded waiting).

Peterson's Solution

process가 2개일때만 가능

- Not guaranteed to work on modern computer architectures
  - To improve system performance, processors and/or compilers may reorder read and write operations that have no dependencies.

    컴파일하면 순서가 바뀔수 있음
  - For a multithreaded application with shared data, the reordering of instructions may render inconsistent or unexpected results.

Peterson's Solution

- **Example**
  - Shared variables

    boolean flag = false;

    int x = 0;

  - Thread1
    1. while(!flag);
    2. print x;

  - Thread2
    1. x = 100;
    2. flag = true;

    (CPU / 컴파일러)가 optimization 때문에 순서는 바뀔 수 있음

    ➔ No dependency between line 1 and 2 ➔ can be reordered

Agenda

- Background
- The critical-section problem
- <u>H/W Support for Synchronization</u>
- Mutex and Semaphores
- Monitors
- Liveness

Memory Model

- **Memory model:** how a computer architecture determines what memory guarantees it will provide to an application program.

- Categories of memory models
  - Strongly ordered: a memory modification on one processor is <u>immediately visible</u> to all other processors
  - Weakly ordered: modifications to memory on one processor <u>may not be immediately visible</u> to other processors

Memory Barriers

low-level instruction

kernel 개발자들이 사용

- **Memory barrier** (or **memory fences**): instructions that can *force* any changes in memory to be propagated to all other processors

  전파하다
  - All loads and stores are completed before any subsequent load or store operations are performed

- Example
  - Thread1

```
while (!flag)
    memory barrier();
print x;
```

  - Thread2

```
x = 100;
memory barrier();
flag = true;
```

Hardware Instructions

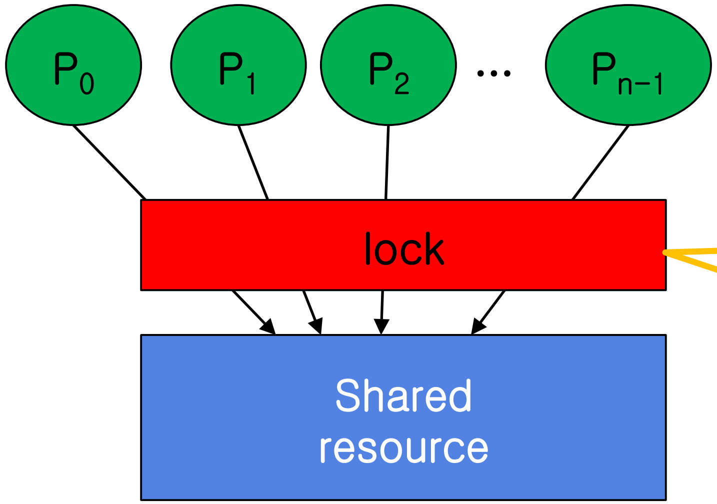
- Critical section problem requires lock
  - Race condition can be prevented by protecting critical section by lock

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

- H/W support makes it easier and improve efficiency

Mutual Exclusion by Lock variable

- A shared variable *lock*
  - Boolean lock = 0;
  - If *lock* is false, any process can enter its critical section
    - When a process enters its critical section, it sets *lock* to true
  - If *lock* is true, a process (P_i) is in its critical section.
    - Other processes should wait until P_i leaves its critical section and sets *lock* to false



Mutual Exclusion by Lock variable

- Initially, lock = false

- P_0 knocking

```
do {
    while(lock);
    lock = true;
        critical section
    lock = false;
        remainder section
} while(TRUE);
```

locking

- P_1

```
do {
    while(lock);
    lock = true;
        critical section
    lock = false;
        remainder section
} while(TRUE);
```

- Note! Checking and locking must not be separated

Interrupt Disable

- In single processor system, critical section problem can be solved by simply disabling interrupt
  - Non-preemptive kernel

- Problems
  - Inefficiency in multiprocessor environment
  - In some systems, clock is updated by interrupt

- This approach is taken by non-preemptive kernels.

Hardware Instructions

쪼개지지 않고 한번에 실행

- If H/W provides atomic instructions, locking is easer to implement
  - TestAndSet: set a variable to true and returns its previous value

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

lock 변수

function X

기계어 명령

한번실행되면 끝까지 실행

중간에 멈춤 X

  - Swap: exchanges two variables

```
void Swap(boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

lock 변수 , local 변수

Hardware Instructions

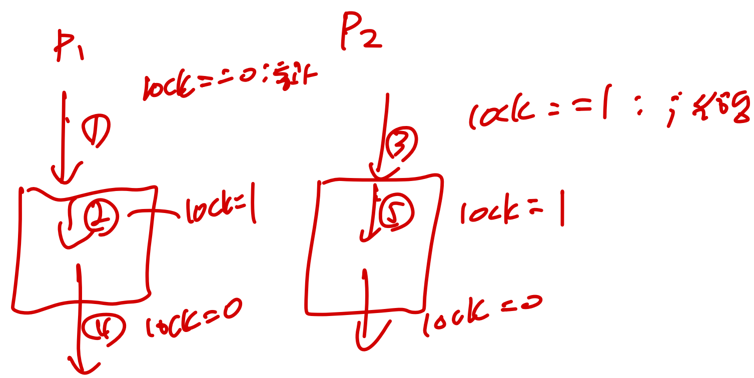
- If H/W provides atomic instructions, locking is easer to implement
  - Compare and swap (CAS): The operand value is set to new value only if the expression (*value== expected) is true.

```
int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
}
```

Mutual Exclusion using TestAndSet

lock=~~0~~ ~~1~~ 0

- Shared variable
  - boolean lock = false;



- Process P_i

return value ==0 (통과)

```
do {
    while (TestAndSet(&lock));

        /* critical section */

    lock = false;

        /* remainder section */
} while(1);
```

Entry section

Atomic instruction

exist section

Mutual Exclusion using Swap

lock=φ 1 φ 1

key=1 0

key=1 0

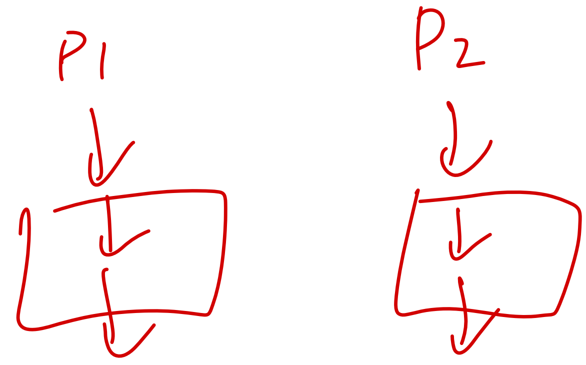
1 2 3 4 5

- Shared variable
  - boolean lock = false;
- Process P_i

```
do {
    key = true;                    // local var.
    while (key == true)
        Swap(&lock, &key);

    /*  critical section */

    lock = false;

    /* remainder section */
} while(1);
```

Mutual Exclusion using CAS

lock = ~~0~~ ~~1~~ ~~0~~ ~~1~~ 0



- Shared variable
  - boolean lock = false;

- Process P_i

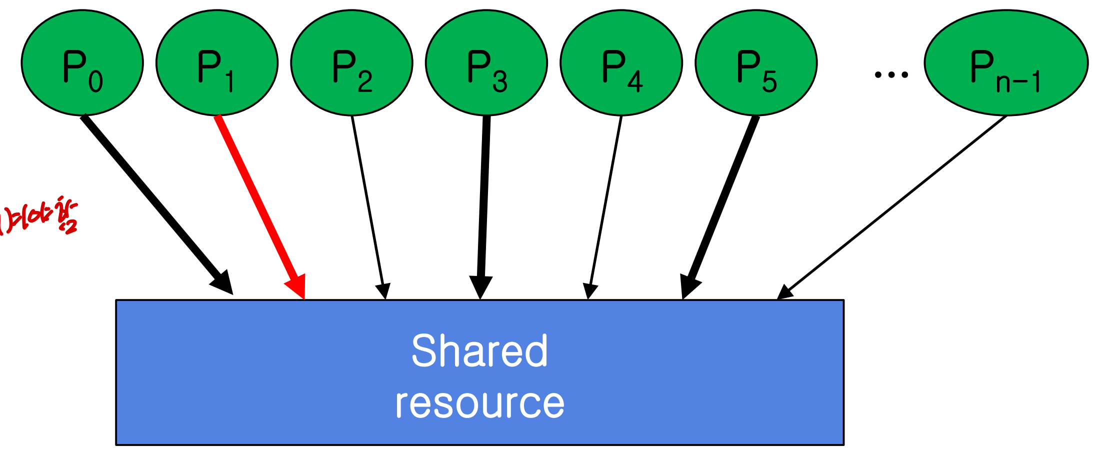
```
while (true) {
      while(compare_and_swap(&lock, 0, 1) != 0);

      /* critical section */

      lock = 0;

      /* remainder section */
}
```

Bounded Waiting Mutual Exclusion

- Bounded waiting for n processes
  - Some processes want to enter their critical sections
    Ex) P_0, P_1, P_3, P_5
  - One of them, (ex: P_1) is in its critical section

P_1 : 들어가고 싶고 실제로 들어감
$P_{0,3,5}$: 들어가고 싶은데 P_1이 있어서 못들어감

bounded waiting: 순서를 따라야함
예) $P_2 \to P_5 \to P_0$

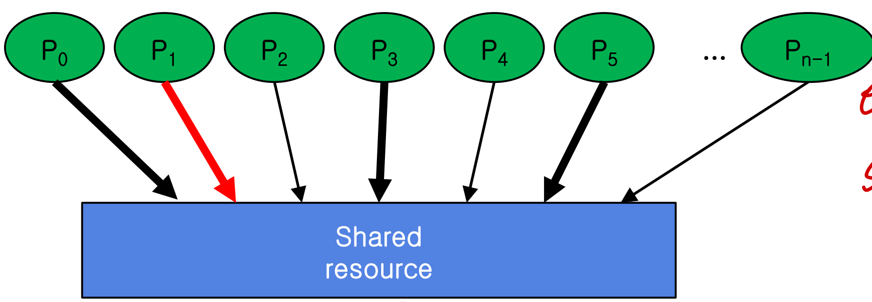


- Idea: Whenever a process leaves its critical section, it designates the next process to enter the critical section.
  - The next process: The first waiting process in right direction.

Bounded Waiting Mutual Exclusion

각 변수가 어떤 목적인지 확실히 하기

- Shared variables
  - boolean lock;
    - For mutual exclusion
  - boolean waiting[n];
    - if waiting[i] is true, it means P_i wants to enter critical section, but it didn't enter yet.



process	0	**1**	2	3	4	5	6	…	n-1
waiting	1	**0**	0	1	0	1	0	…	0

↑ P_i

- Algorithm

lock: 0 / 1

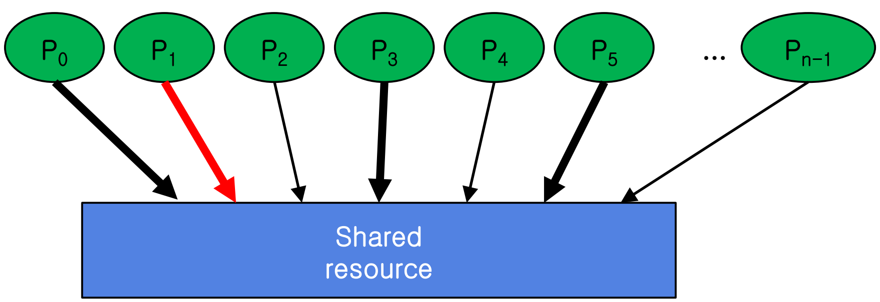
entry section

exist section

```
while(TRUE){
    waiting[i] = TRUE;
    key = TRUE;        // local var.
    while(waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    /* critical section */

    j = (i + 1) % n;
    while(j != i && !waiting[j])
        j = (j + 1) % n;

    if(j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;

    /* remainder section */
}
```

Bounded Waiting Mutual Exclusion

- Shared variables
  - boolean lock;
    - For mutual exclusion
  - boolean waiting[n];
    - if waiting[i] is true, it means P_i wants to enter critical section, but it didn't enter yet.



process	0	**1**	2	3	4	5	6	…	n-1
waiting	1	**0**	0	1	0	1	0	…	0

$\uparrow$
P_i

- Algorithm using CAS

```
while(true) {
    waiting[i] = true;
    key = 1;
    while (waiting[i] && key)
        key =
            compare_and_swap(&lock,0,1);
    waiting[i] = false;

    /* critical section */

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = 0;
    else
        waiting[j] = false;

    /* remainder section */
}
```

Atomic Variables

- Atomic variables: variables that provide atomic operations on basic data types such as integers and Booleans
  - Provided as “special data type + operation”
  - Can be used in to ensure mutual exclusion

v = 5

  - Typically, implemented by CAS instruction

```
void increment(atomic int *v)
{
    int temp = 0;
    do {
        temp = *v;
    } while (temp != compare_and_swap(v, temp, temp+1));
}
```

increment 자체는 atomic 하지 않음

atomic

V를 보고 바뀌었는지 확인

P1

temp=5

v=5

v=6

P2

temp=5

v=6

P1이 바꾸었기 때문

temp=6

v=6

v=7

Agenda

- Background
- The critical-section problem
- H/W Support for Synchronization
- <u>Mutex and Semaphore</u>
- Monitors
- Liveness

Mutex Locks

- A synchronization tool to implement mutual exclusion
  - A Boolean variable *available*   available 하지 않으면 lock, lock 변수의 반대
    - Opposite to the lock variable in the previous section

- Atomic operations on mutex locks
  - acquire() function entry section 에서 호출

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

  - release() function exist section에서 호출

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

Mutex Locks

- Solution to the critical-section problem using mutex lock

```
do {
    mutex.acquire()
        critical section
    mutex.release()
        remainder section
} while(TRUE);
```

Semaphores

- Semaphore: an integer variable accessed only through two atomic operations: wait() and signal()
  - Initial value of S is 1 or a positive integer.
  - Wait // correspond to mutex.acquire()

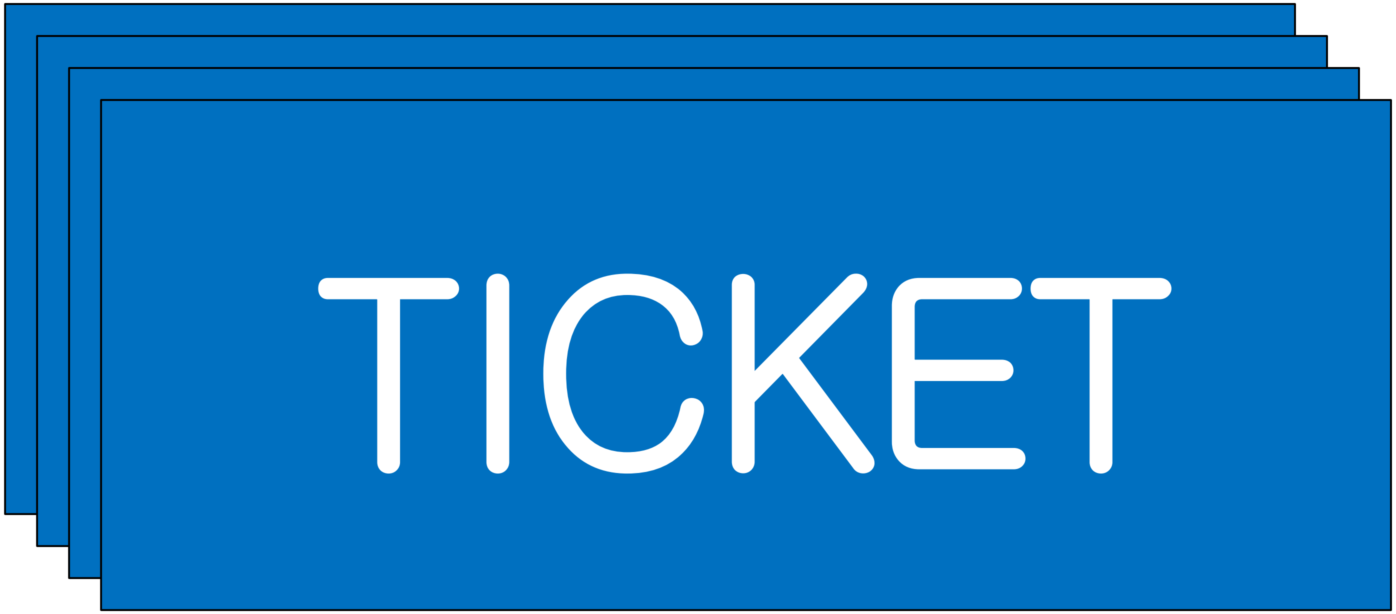
```
wait(S) {
    while(S <= 0);        // waits for the lock
    S--;                  // holds the lock
}
```

  - Signal // correspond to mutex.release()

```
signal(S) {
    S++;                  // releases the lock
}
```

Semaphores

- S is similar to the number of reusable tickets to enter the critical section.



Semaphores

- Synchronization using semaphore

```
do {
      wait(S);
                critical section
      signal(S);
                remainder section
} while(1);
```

  - If initially value of S is 1,
    - wait(S) acquires lock
    - signal(S) releases lock

- Wait

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

- Signal

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

Implementation of Semaphore

- Problem of previous definition of wait(): spinlock

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

- Alternative implementation: block instead of spinlock
  - If a process invokes wait(S), put the process into a waiting queue of S and block itself.

Implementation of Semaphore

- New definition of semaphore

```
typedef struct {
    int value;
    struct process *list;     // head of linked list for waiting queue
} semaphore;
```

PCB ready queue도 linked list로 구현

- Wait

```
wait(semaphore *S){
    S->value--;
    if(S->value < 0){
      add this process to S->list;
      block(); // suspend
    }
}
```

- Signal

```
signal(semaphore *S)
{
    S->value++;
    if(S->value <= 0){
      remove a process P from S->list;
      wakeup(P);   // resume
    }
}
```

Implementation of Semaphore

- A critical aspect of semaphore: atomicity
  - Atomicity is often enforced by mutual exclusion → atomic 보장과 같은 효과
  - Single processor environment: disable interrupt → mutual exclusion → atomic
  - Multiprocessor environment: spinlock
    - Disable interrupt of all processors ➔ performance degradation
    - Spinlock is much short than previous algorithms

Agenda

- Background
- The critical-section problem
- H/W Support for Synchronization
- Mutex and Semaphores
- <u>Monitors</u>
- Liveness

Monitors

- Motivation: semaphore is still too low-level tool.

  Example)

correct	wrong	wrong

wait(mutex);
…
critical section
…
signal(mutex);
	signal(mutex);
	…
	critical section
	…
	wait(mutex)
		wait(mutex);
		…
		critical section
		…
		wait(mutex);

- Monitor: a high-level language construct to support synchronization
  - Private data: accessible only through the methods
  - Public methods: mutual exclusion is provided

Class/Structures in OOP Language

- C language

```
struct Stack {
    int array[100];
    int top;
};

void Push(struct Stack *s,
    int item);
int Pop(struct Stack *s);

struct Stack stack1;
Push(&stack1, 10);
```

- C++, Java

```
class Stack {
    int array[100];
    int top;
private:
    void Push(int item);
    int Pop();
};

Stack stack2;
stack2.Push(10);
```

Monitors

- Syntax of monitor

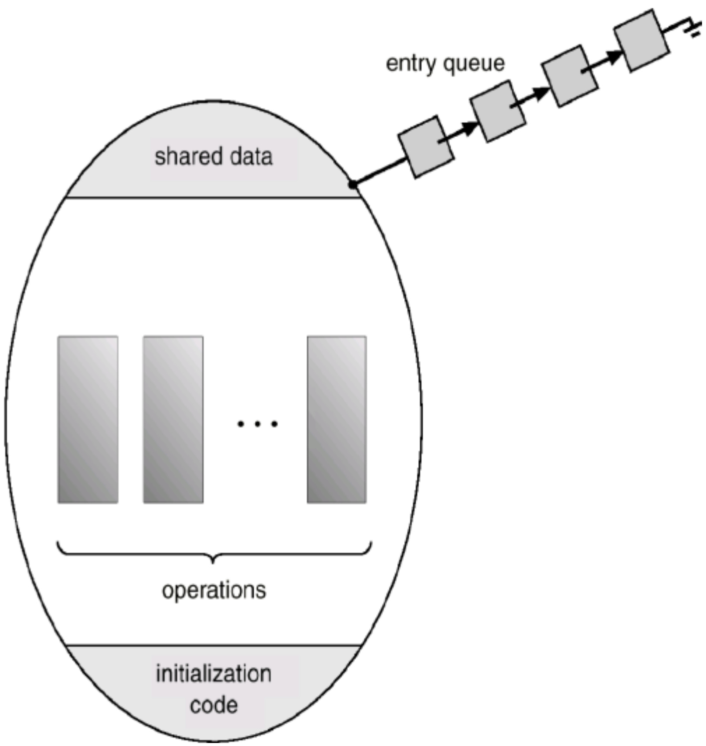
monitor *monitor-name*
{
    // shared variable declarations

    procedure body *P1* (…) {
        wait
        …
        signal
    }
    procedure body *P2* (…) {
        wait
        …
        signal
    }
    …
    procedure body *Pn* (…) {
        wait
        …
        signal
    }
    initialization code (…) {
        …
    }
}

Only one process can be active within the monitor at a time

Monitors

- Only one process can be active within the monitor at a time.
  - The programmer doesn’t have to concern about synchronization.



Monitor in Java

- Synchronized method

  Ex)

```
class Producer {
    private int product;
    ...
    private synchronized void produce();      // mutually exclusive
}
```

mutex나 semaphore 없이 구현

c.f. C#: System.Threading.Monitor class

Condition Variables

- Condition: additional synchronization mechanism
  - Variable declaration

    condition x, y;

semaphore랑 조금 다름

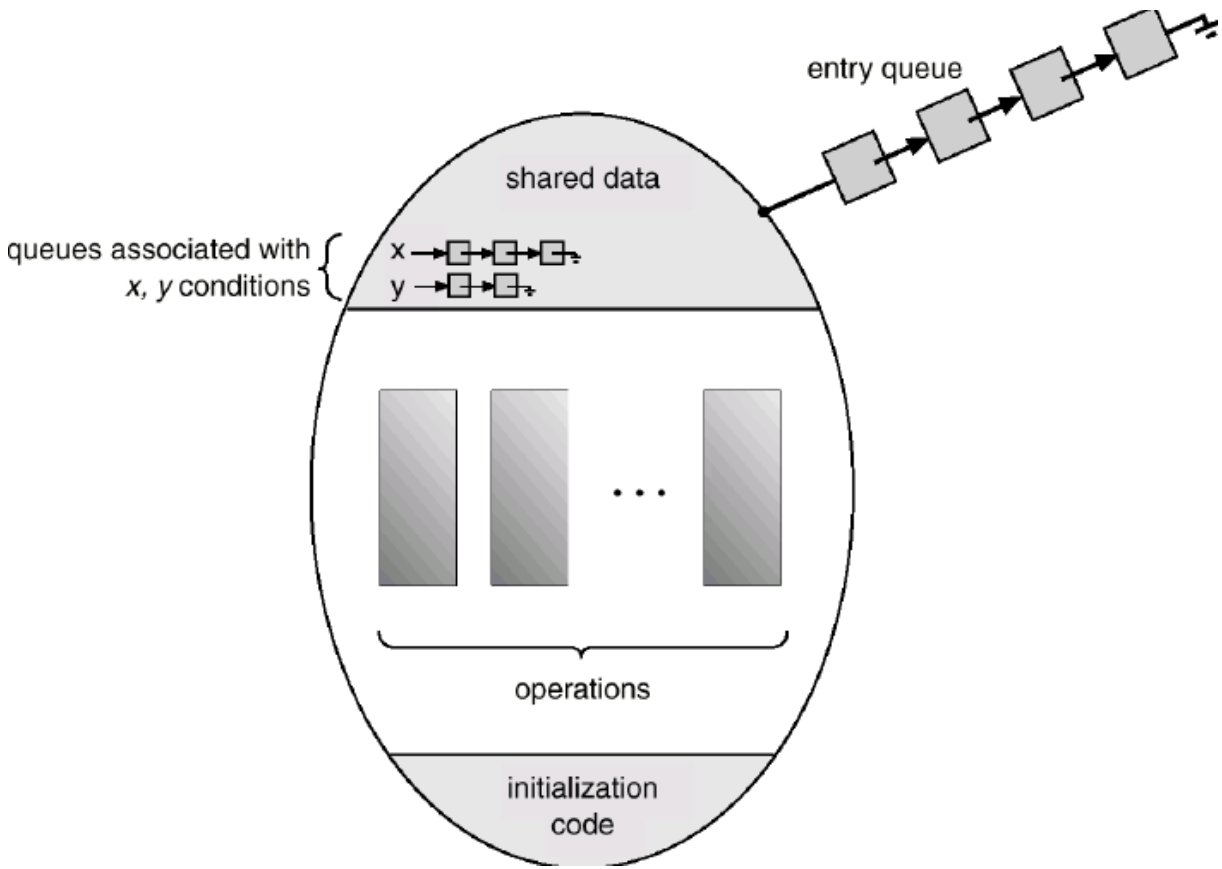
  - Wait

    x.wait(); // invoking process is suspended until
    // other process call x.signal()

  - Signal

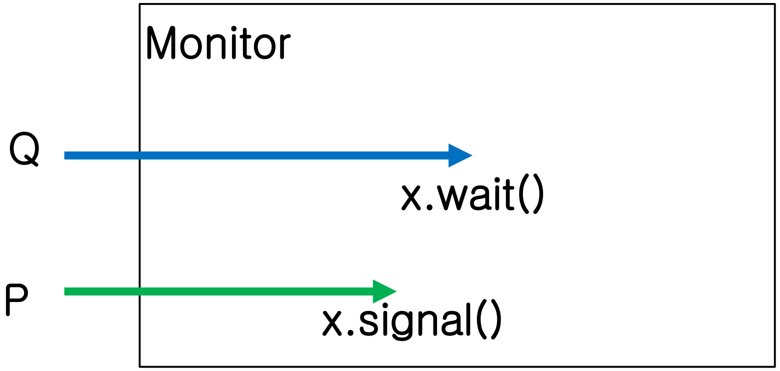
    x.signal(); // resumes exactly one suspended process
    // <u>if no process is waiting, do nothing</u>

Condition Variables



Condition Variables

- Problem
  - Process P wakes up another process Q by invoking x.signal().
  - Both P and Q are executing in monitor.
- Solution
  - Signal and wait: P waits until Q leaves monitor or waits
  - Signal and continue: Q waits until P leaves monitor or waits



Implementing a Monitor Using Semaphores

- Functions to implement
  - External procedures
  - wait() of condition variable
  - signal() of condition variable

- Two semaphores are required:
  - semaphore *mutex = 1*; // mutual exclusion
  - semaphore *next = 0*; // signaling process should wait
    // until the resumed process leaves monitor
  - int next_count = 0;

Implementing a Monitor Using Semaphores

- External procedure F

```
wait(mutex);
    …
    body of F;
    …
signal(mutex);
```

Implementing a Monitor Using Semaphores

- Condition variables
  - Required data
    - semaphore x_sem; // (initially, 0)
    - int x_count = 0;

- x.wait()

```
{
    x_count++;
    if (next_count > 0)
      signal(next);
    else
      signal(mutex);
    wait(x_sem);
    x_count--;
}
```

- x.signal()

```
{
    if (x_count > 0) {
       next_count++;
       signal(x_sem);
       wait(next);
       next_count--;
    }
}
```

Resuming Processes within a Monitor

- If a process invoke x.signal() and there are several processes waiting on x, which one should be resumed?
  - Sometimes, FIFO is not enough

- Conditional-wait
  - x.wait(c);   // c: priority number

Agenda

- Background
- The critical-section problem
- H/W Support for Synchronization
- Mutex and Semaphores
- Monitors
- <u>Liveness</u>

Liveness

- Liveness: a set of properties that a system must satisfy to ensure that processes make progress during their execution life cycle.

- Reasons of liveness failure
  - Deadlock
  - Priority inversion
  - Many other reasons

Deadlock and Starvation

- Deadlock

P_0	P_1
wait(S);	*wait(Q);*
wait(Q);	*wait(S);*
⋮	⋮
signal(S);	*signal(Q);*
signal(Q);	*signal(S);*

- Starvation (infinite blocking): a situation in which a process waits indefinitely within the semaphore

  Ex) waiting list is implemented by LIFO order

Priority Inversion

- Priority inversion: a higher-priority process needs to read or modify kernel data that are currently being accessed by a (chain of) lower-priority process
  - Three processes L, M, and H
    - Priorities: L < M < H
  - H waits for a semaphore S acquired by L.
  - L should wait for M because of its lower priority
  - Consequently, H waits for M

- Can be avoided by priority-inheritance protocol