Embedded Systems Design

Synchronization

Contents

1 Race Condition

2 Critical Section

3 Synchronization Hardware

Data Inconsistency

- Cooperating process
- Can affect or be affected by another process
  - Can share data and code
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

병행 수행과 병행 제어

- 병행 수행(Concurrency)
  - 여러 개의 트랜잭션을 동시에 수행
  - 여러 트랜잭션이 차례로 번갈아 수행되는 인터리빙(Interleaving) 방식으로 진행

- 병행 제어(Concurrency Control) 또는 동시성 제어
  - 동시에 수행되는 여러 트랜잭션들이 마치 순차적으로 수행한 것과 같은 결과가 되도록 트랜잭션의 수행을 제어

병행 수행 문제	의미
갱신 분실(Lost Update)	하나의 트랜잭션의 변경 연산이 다른 트랜잭션의 변경 연산에 의해 무효화되는 현상
모순성(Inconsistency)	모든 데이터 변경 연산이 완료되지 않은 상태에서 다른 트랜잭션 연산을 수행
연쇄 복귀(Cascading Rollback)	rollback 연산 수행시, 장애 발생 전에 변경된 데이터를 사용한 다른 트랜잭션도 rollback 연산을 수행

< 병행 수행 문제 >

갱신 분실

- 갱신 분실(Lost Update)
- 하나의 트랜잭션의 변경 연산이 다른 트랜잭션의 변경 연산에 의해 무효화되는 현상

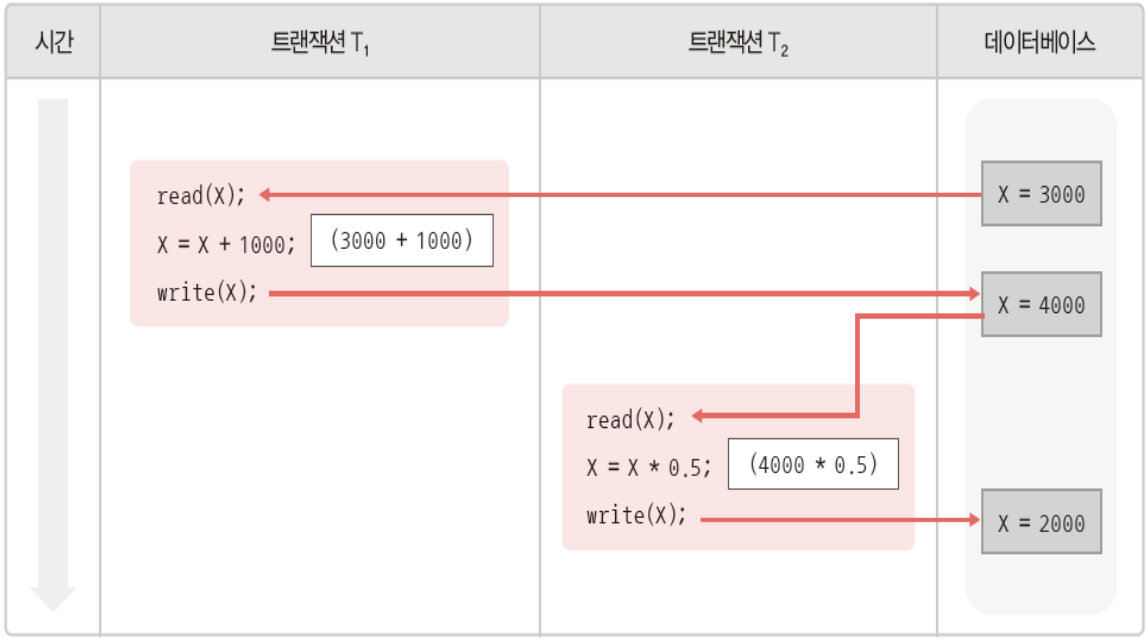


그림 10-31 트랜잭션 T_1을 수행한 후 트랜잭션 T_2를 수행한 결과

트랜잭션 T_1에 대해 갱신 분실이 발생함

(T_1의 변경 연산 결과가 데이터베이스에 반영되지 않음)

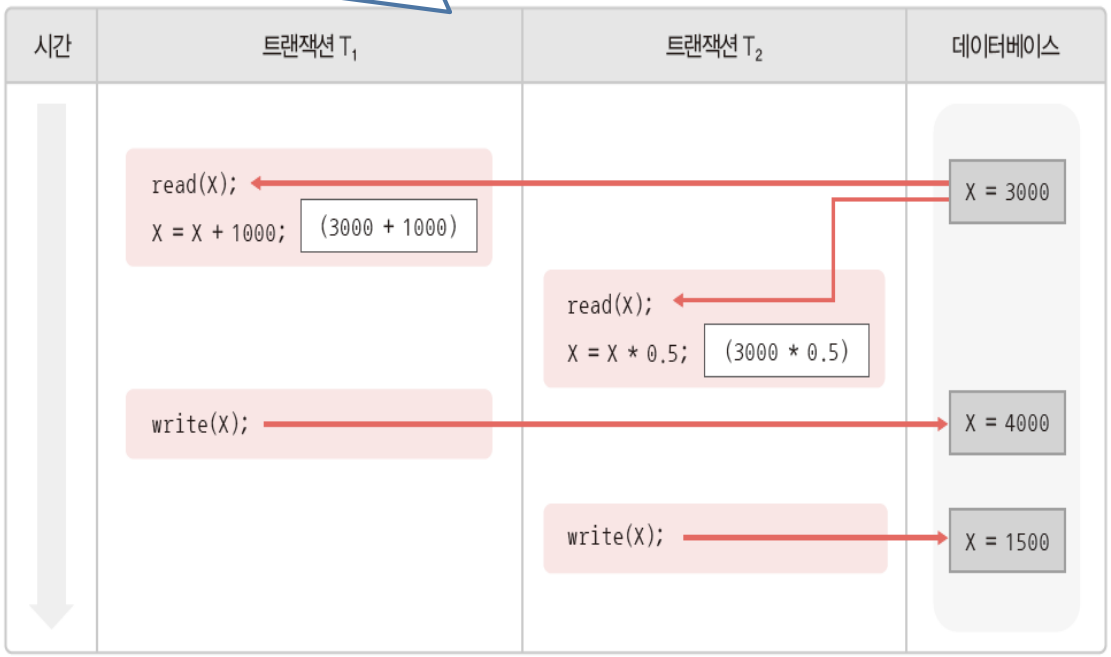
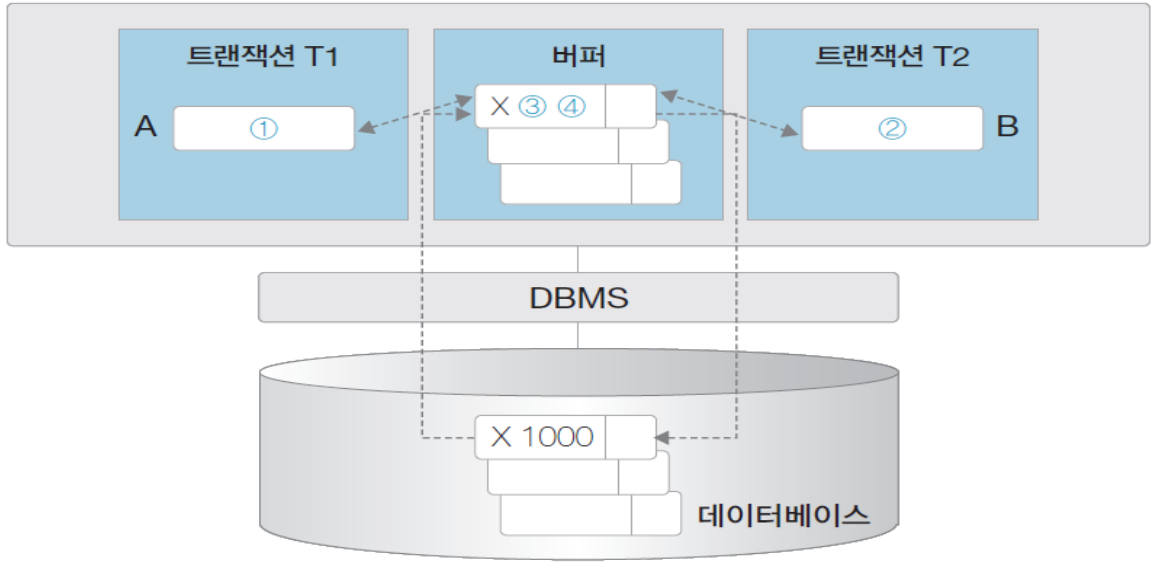


그림 10-30 두 트랜잭션의 병행 수행으로 발생한 갱신 분실의 예

갱신 분실

- 갱신 분실(Lost Update)

트랜잭션 T1	트랜잭션 T2	버퍼의 데이터 값
A=read_item(X); ① A=A-100;		X=1000
	B=read_item(X); ② B=B+100;	X=1000
write_item(A → X); ③		X=900
	write_item(B → X); ④	X=1100



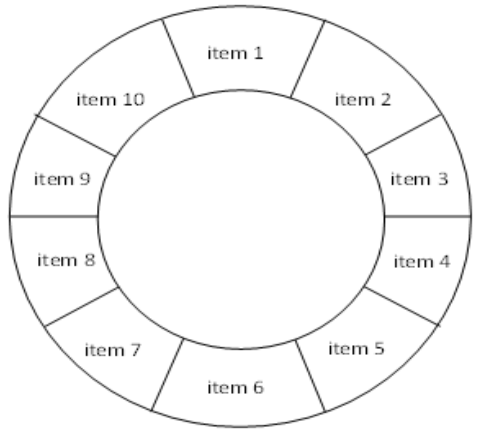
Race Condition

- **Race Condition**
- Several processes access the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place
- Synchronization is required
  - To guard against the race condition, only one process manipulate the shared data at a time
- Increasing the prominence of multicore systems
  - Multithreaded applications are running in parallel on different processing cores
- OS is subject to several possible race conditions because many kernel mode processes are active

Producer-Consumer (Revisited)

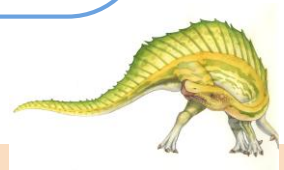
- Producer-Consumer paradigm
- Producer process produces information
- Consumer process consumes the information

- Bounded-buffer
- There is a fixed buffer size
  - BUFFER_SIZE-1 items
- Producer must wait if all buffers are full
- Consumer waits if there is no buffer to consume




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

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

< Data structure >



Bounded-Buffer

- Producer-Consumer processes

```
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;
}
```

< Producer process >

```
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 */
}
```

< Consumer process >

Bounded-Buffer with Counter

- Producer-Consumer processes

```
while (true) {
/* produce an item in next produced */
   while (counter == BUFFER SIZE) ;
   /* do nothing */

   buffer[in] = next produced;
   in = (in + 1) % BUFFER SIZE;
   counter++;
}
```

< Producer process >

```
while (true) {
   while (counter == 0)
   ; /* do nothing */

   next consumed = buffer[out];
   out = (out + 1) % BUFFER SIZE;
   counter--;
/* consume the item in next consumed */
}
```

< Consumer process >

Implementation of Bounded-Buffer with Counter

- `counter++` could be implemented as

```
reg1 = counter
reg1 = reg1 + 1
counter = reg1
```

< counter++ >

- `counter--` could be implemented as

```
reg2 = counter
reg2 = reg2 - 1
counter = reg2
```

< counter-- >

- Consider this execution interleaving with "count = 5" initially

```
P: reg1 = counter   {reg1 = 5}
P: reg1 = reg1 + 1  {reg1 = 6}
P: counter = reg1   {counter = 6}
C: reg2 = counter   {reg2 = 6}
C: reg2 = reg2 - 1  {reg2 = 5}
C: counter = reg2   {counter = 5}
```

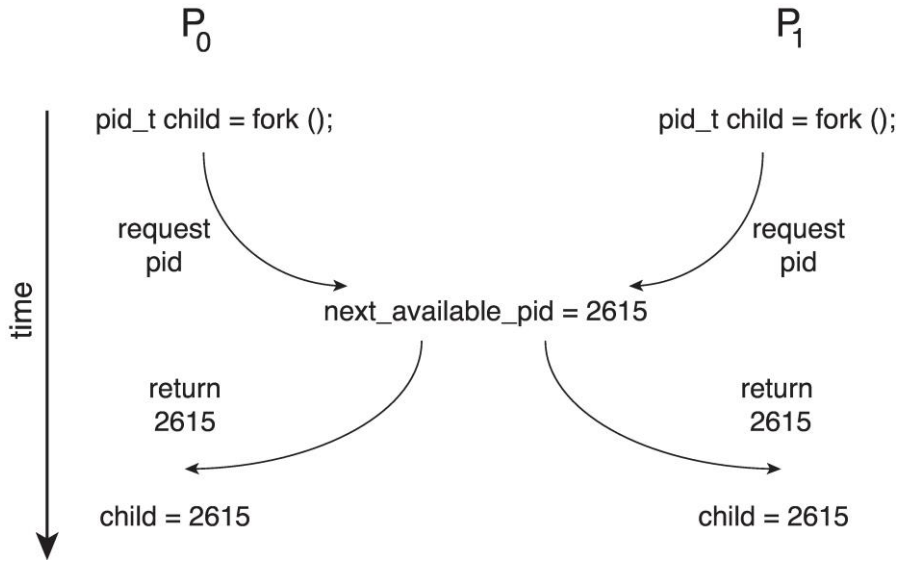
< Data consistency case >

```
P: reg1 = counter   {reg1 = 5}
P: reg1 = reg1 + 1  {reg1 = 6}
C: reg2 = counter   {reg2 = 5}
C: reg2 = reg2 - 1  {reg2 = 4}
P: counter = reg1   {counter = 6}
C: counter = reg2   {counter = 4}
```

< Data inconsistency case >

Example – Race Condition

- Processes P_0 and P_1 are creating child processes using the `fork()` system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)
- Unless there is a mechanism to prevent P_0 and P_1 from accessing the variable `next_available_pid` the same pid could be assigned to two different processes!



< Race condition for pid >

Contents

1 Race Condition

2 Critical Section

3 Synchronization Hardware

Critical Section

- **Critical section**
- A code segment in which the process accesses or manipulates the shared data
- 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 a 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

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

< Structure of critical section >

Critical Section Problem

- Requirements for critical section problem
- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the n processes
- Mutual exclusion
  - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
- Progress
  - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely
  - Avoiding deadlock or livelock
- Bounded waiting
  - A bound must exist on the 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 that request is granted
  - Avoiding starvation problem

Solution to Critical Section Problem

- Single core environment
  - Preventing interrupts from occurring whilea shared variable was being modified

- Multicore environment
  - Nonpreemptive kernel
    - Runs until exits kernel mode, blocks, or voluntarily yields CPU
    - Essentially free of race conditions in kernel mode
  - Preemptive kernel
    - Allows preemption of process when running in kernel mode

Peterson's Solution (1)

- Peterson's Solution
- Good algorithmic description of solving the problem
  - A classic software-based solution
- The two processes share two variables:

```
int turn;
Boolean flag[2];
```

- The `turn`
  - Indicates the process to enter the critical section
- The `flag` array
  - Indicates if a process is ready to enter the critical section
  - `flag[i] = true` implies that process P_i is ready
- There are two processes (P_0 and P_1)
  - If $P_0 = P_i$, $P_1 = P_j$ / If $P_1 = P_i$, $P_0 = P_j$

```
while (true){
   flag[i] = true;  // entry section
   turn = j;
   while (flag[j] && turn == j)
    ;

   /* critical section */

   flag[i] = false; // exit section

   /* remainder section */

}
```

< Structure of Peterson's solution >

Example 1 - Peterson's Solution (1)

- When only P_0 uses the shared data

```
while (true){
   flag[0] = true;
   turn = 1;
   while (flag[1] && turn == 1)
    ;

   /* critical section */

   flag[0] = false;

   /* remainder section */

}
```

< P_0 >

```
flag[0] = false
flag[1] = false
turn = 0
```

< Shared variables >

```
while (true){
   flag[1] = true;
   turn = 0;
   while (flag[0] && turn == 0)
    ;

   /* critical section */

   flag[1] = false;

   /* remainder section */

}
```

< P_1 >

Example 1 - Peterson's Solution (2)

- When only P_0 uses the shared data

```
while (true){
   flag[0] = true;
   turn = 1;
   while (flag[1] && turn == 1)
    ;

   /* critical section */

   flag[0] = false;

   /* remainder section */

}
```

< P_0 >

```
while (true){
   flag[1] = true;
   turn = 0;
   while (flag[0] && turn == 0)
    ;

   /* critical section */

   flag[1] = false;

   /* remainder section */

}
```

< P_1 >

```
flag[0] = true
flag[1] = false
turn = 1
```

< Shared variables >

Example 2 - Peterson's Solution (1)

- When P_0 firstly updates the turn as 1

```
while (true){
   flag[0] = true;
   turn = 1;
   while (flag[1] && turn == 1)
    ;

   /* critical section */

   flag[0] = false;

   /* remainder section */

}
```

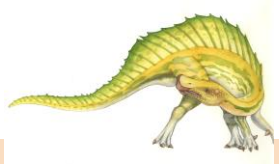
< P_0 >

```
while (true){
   flag[1] = true;
   turn = 0;
   while (flag[0] && turn == 0)
    ;

   /* critical section */

   flag[1] = false;

   /* remainder section */

}
```

< P_1 >

```
flag[0] = true
flag[1] = true
turn = 1
```

< Shared variables >



Example 2 - Peterson's Solution (2)

- When P_0 firstly updates the turn as 1

```
while (true){
   flag[0] = true;
   turn = 1;
   while (flag[1] && turn == 1)
    ;

   /* critical section */

   flag[0] = false;

   /* remainder section */

}
```

< P_0 >

```
while (true){
   flag[1] = true;
   turn = 0;
   while (flag[0] && turn == 0)
    ;

   /* critical section */

   flag[1] = false;

   /* remainder section */

}
```

< P_1 >

```
flag[0] = true
flag[1] = true
turn = 0
```

< Shared variables >

Example 2 - Peterson's Solution (3)

- When P_0 firstly updates the turn as 1

```
while (true){
   flag[0] = true;
   turn = 1;
   while (flag[1] && turn == 1)
    ;

   /* critical section */

   flag[0] = false;

   /* remainder section */

}
```

< P_0 >

```
while (true){
   flag[1] = true;
   turn = 0;
   while (flag[0] && turn == 0)
    ;

   /* critical section */

   flag[1] = false;

   /* remainder section */

}
```

< P_1 >

```
flag[0] = true
flag[1] = true
turn = 0
```

< Shared variables >

Example 2 - Peterson's Solution (4)

- When P_0 firstly updates the turn as 1

```
while (true){
   flag[0] = true;
   turn = 1;
   while (flag[1] && turn == 1)
    ;

   /* critical section */

   flag[0] = false;

   /* remainder section */

}
```

< P_0 >

```
while (true){
   flag[1] = true;
   turn = 0;
   while (flag[0] && turn == 0)
    ;

   /* critical section */

   flag[1] = false;

   /* remainder section */

}
```

< P_1 >

```
flag[0] = false
flag[1] = true
turn = 0
```

< Shared variables >

Example 2 - Peterson's Solution (5)

- When P_0 firstly updates the turn as 1

```
while (true){
    flag[0] = true;
    turn = 1;
    while (flag[1] && turn == 1)
      ;

    /* critical section */

    flag[0] = false;

    /* remainder section */

}
```

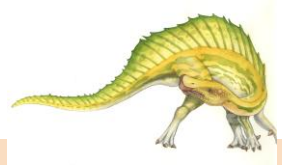
< P_0 >

```
flag[0] = true
flag[1] = true
turn = 1
```

< Shared variables >

```
while (true){
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn == 0)
      ;

    /* critical section */

    flag[1] = false;

    /* remainder section */

}
```

< P_1 >

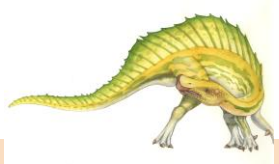


Correctness of Peterson's Solution (1)

- Mutual exclusion
- P_i enters the critical section only if
  - either `flag[j] = false` or `turn = i`
- For `flag[j] = false`
  - The other process doesn't want to enter the critical section
- For `flag[i] = flag[j] = true`,
  - `turn` has either 0 or 1 but cannot be both
  - One process must have successfully executed '`while`'
  - `turn` is not modified in the critical section

```
while (true){
   flag[i] = true;  // entry section
   turn = j;
   while (flag[j] && turn == j)
    ;

   /* critical section */

   flag[i] = false; // exit section

   /* remainder section */

}
```

< Struction of Peterson's solution >



Correctness of Peterson's Solution (2)

- Progress
  - P_i enters the critical section when `flag[j] = false`
- Bounded waiting
  - Once P_j exits its critical section, `flag[j] = false`
  - If `flag[j] = true` for P_j, it must also set turn to i
  - Since P_i does not change `turn` while executing
    - P_i will enter the critical section next

```
while (true){
   flag[i] = true;  // entry section
   turn = j;
   while (flag[j] && turn == j)
    ;

   /* critical section */

   flag[i] = false; // exit section

   /* remainder section */

}
```

< Struction of Peterson's solution >

Example 2 - Peterson's Solution (3) (Revisited)

- When P_0 firstly updates the turn as 1

```
while (true){
   flag[0] = true;
   turn = 1;
   while (flag[1] && turn == 1)
    ;

   /* critical section */

   flag[0] = false;

   /* remainder section */

}
```

< P_0 >

```
while (true){
   flag[1] = true;
   turn = 0;
   while (flag[0] && turn == 0)
    ;

   /* critical section */

   flag[1] = false;

   /* remainder section */

}
```

< P_1 >

```
flag[0] = false
flag[1] = true
turn = 0
```

< Shared variables >

Example 2 - Peterson's Solution (4) (Revisited)

- When P_0 firstly updates the turn as 1

```
while (true){
    flag[0] = true;
    turn = 1;
    while (flag[1] && turn == 1)
      ;

    /* critical section */

    flag[0] = false;

    /* remainder section */

}
```

< P_0 >

```
while (true){
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn == 0)
      ;

    /* critical section */

    flag[1] = false;

    /* remainder section */

}
```

< P_1 >

```
flag[0] = true
flag[1] = true
turn = 0
```

< Shared variables >

Example 2 - Peterson's Solution (5) (Revisited)

- When P_0 firstly updates the turn as 1

```
while (true){
    flag[0] = true;
    turn = 1;
    while (flag[1] && turn == 1)
      ;

    /* critical section */

    flag[0] = false;

    /* remainder section */

}
```

< P_0 >

```
while (true){
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn == 0)
      ;

    /* critical section */

    flag[1] = false;

    /* remainder section */

}
```

< P_1 >

```
flag[0] = true
flag[1] = true
turn = 1
```

< Shared variables >

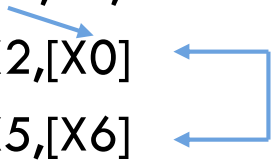
Out-of-order Execution

- Out-of-order Execution(OOO)
- Instruction 수행 순서를 CPU에서 내부적으로 변경
  - 준비된 Instruction을 먼저 수행하여, 전체 Program의 실행 시간을 단축
  - 순서를 바꾸어도 문제가 없는지를 판단하는 복잡한 Logic이 필요

- Ex) ADD X0,X3,X2

  LDR X2,[X0]

  LDR X5,[X6]



Example - Out-of-order Execution

- The expected behavior is that Thread 0 prints the value 100
- However, it is possible that a processor may reorder the instructions of Thread 1 so that Thread 1 would output 0

```
boolean flag = false;
int x = 0;

while (!flag);
print x
```

< Thread 0 >

```
x = 100;
flag = true;
```

< Thread 1 >

Limitation of Peterson's Solution

- Limitation of Peterson's Solution
- Although useful for demonstrating an algorithm, Peterson's solution is not guaranteed to work on modern architectures
- To improve performance, processors and/or compilers may reorder operations that have no dependencies
- For single-threaded this is OK as the result will always be the same
- For multithreaded the reordering may produce inconsistent or unexpected results!

Contents

1 Race Condition

2 Critical Section

3 Synchronization Hardware

Synchronization Hardware

- Synchronization Hardware
  - Three hardware instructions that provide support for synchronization
  - Memory Barriers
  - Hardware instructions
  - Atomic Variables

Memory Barrier

- Memory barrier (or Memory fense)
- Memory barrier instructions ensure that all loads and stores instructions are completed before any subsequent load or store operations are performed
- Therefore, even if instructions were reordered, the memory barrier ensures that the store operations are completed in memory and visible to other processors before future load or store operations are performed

```
boolean flag = false;
int x = 0;

while (!flag)
   memory_barrier();
print x
```

< Thread 0 >

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

< Thread 1 >

Atomic Instructions

- Atomic instructions
- Special hardware instructions that allow us to either `test-and-set` the content of a word, or to `swap` the contents of two words atomically (uninterruptedly)
  - test-and-set
  - compare-and-swap

test-and-set

- test-and-set
- Writes 1 (set) to a memory location and return its old value as a single atomic
- Can't solve the bounded waiting problem

```
boolean test_and_set(boolean *target)
{
   boolean rv = *target;
   *target = true;

   return rv:
}
```

< test-and-set >

```
lock = false

do {
   while (test_and_set(&lock))
   ; /* do nothing */

   /* critical section */

   lock = false;

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

< Mutual exclusive with test-and-set >

compare-and-swap (1)

- `compare-and-swap`
- Returns the original value of passed parameter value
- Set *`value` of the passed parameter `new_value` but only if `*value == expected`
  - That is, the swap takes place only under this condition
  - Executed atomically

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

   return temp;
}
```

< compare-and-swap >

compare-and-swap (2)

- Mutual exclusive with `compare-and-swap`
- Can't solve the bounded waiting problem

```
lock = 0

while (true){
   while (compare_and_swap(&lock, 0, 1) != 0)
   ; /* do nothing */

   /* critical section */

   lock = 0;

   /* remainder section */
}
```

< Mutual exclusive with `compare-and-swap` >

compare-and-swap (3)

- Bounded waiting with `compare-and-swap`

```
while (true) {
   waiting[i] = true;
   key = 1;
   while (waiting[i] && key == 1)
      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 */
}
```

< Bounded waiting with compare-and-swap >

Atomic Variables

- Atomic Variables
- Provide atomic (uninterruptible) updates on basic data types such as integers and Booleans
- The `increment()` function can be implemented as follows:

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

< Implementation of `increment()` >

- Although atomic variables provide atomic updates, they do not entirely solve race conditions in all circumstatnces

Mutex Locks

- **Mutex (Mutual exclusion)**
- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
  - Boolean variable indicating if lock is available or not
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

Mutex Operations

- Two atomic operations
- `acquire()` a lock
- `release()` a lock
- Usually implemented via hardware atomic instructions such as `compare-and-swap`

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

< acquire() >

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

< release() >

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

< Mutex locks >

Semaphores

- Semaphores
- Provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities
- Semaphore **S** (integer variable)
- Can only be accessed via two indivisible (atomic) operations
  - `wait()` and `signal()`
  - Originally called `P()` and `V()`

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

< wait() >

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

< signal() >

- Types of semaphores
- Counting semaphore
  - integer value can range over an unrestricted domain
- Binary semaphore
  - integer value can range only between 0 and 1
  - Same as a mutex lock
- Can implement a counting semaphore **S** using binary semaphores

Semaphore Usage Example

- `wait()` / `signal()`

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

< wait() >

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

< signal() >

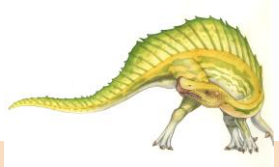
- Usage example
- The requirement that S_1 to happen before S_2
- Create a semaphore `synch` initialized to 0

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

< P_1 Code >

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

< P_2 Code >



Semaphore Implementation without busy waiting

- Each entry in a waiting queue has two data items
  - `value` (of type integer)
  - Pointer to next record in `list`
- When a process must wait on a semaphore, it is added to `list`
  - Then, the process can suspend itself by `sleep()`
  - The process state is changed to the waiting state
- `signal()` removes one process from `list` and awakens that process
  - The process state is switched to the ready state

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

< Waiting queue >

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

< wait() without busy waiting >

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

< signal() without busy waiting >

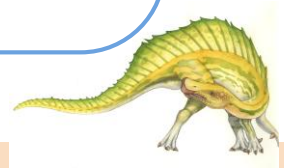
Monitors

- Problems with semaphores
- Programmers easily use semaphores incorrectly
  - Omitting wait() or signal()
- Incorrect use of semaphore operations results in timing errors that are difficult to detect
  - Because these errors happen only if particular execution sequences take place and these sequences do not always occur

- Monitor
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Abstract data type, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
  // shared variable declarations
  function P1 (…) { …. }
  function P2 (…) { …. }
  function Pn (…) {……}
  initialization code (…) { … }
}
```

< Structure of monitor >



Liveness

- Liveness
- Liveness refers to a set of properties that a system must satisfy to ensure processes make progress
- Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a mutex lock or semaphore
  - Waiting indefinitely violates the progress and bounded-waiting criteria discussed at the beginning of this chapter.
  - Indefinite waiting is an example of a **liveness failure**

- Deadlock
- Two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

- Priority Inversion
- Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via priority-inheritance protocol

Priority Inheritance Protocol

- Consider the scenario with three processes P1, P2, and P3. P1 has the highest priority, P2 the next highest, and P3 the lowest
- Assume a resource P3 is assigned a resource R that P1 wants
  - Thus, P1 must wait for P3 to finish using the resource. However, P2 becomes runnable and preempts P3
  - What has happened is that P2 - a process with a lower priority than P1 - has indirectly prevented P3 from gaining access to the resource
- To prevent this from occurring, a priority inheritance protocol is used. This simply allows the priority of the highest thread waiting to access a shared resource to be assigned to the thread currently using the resource
- Thus, the current owner of the resource is assigned the priority of the highest priority thread wishing to acquire the resource