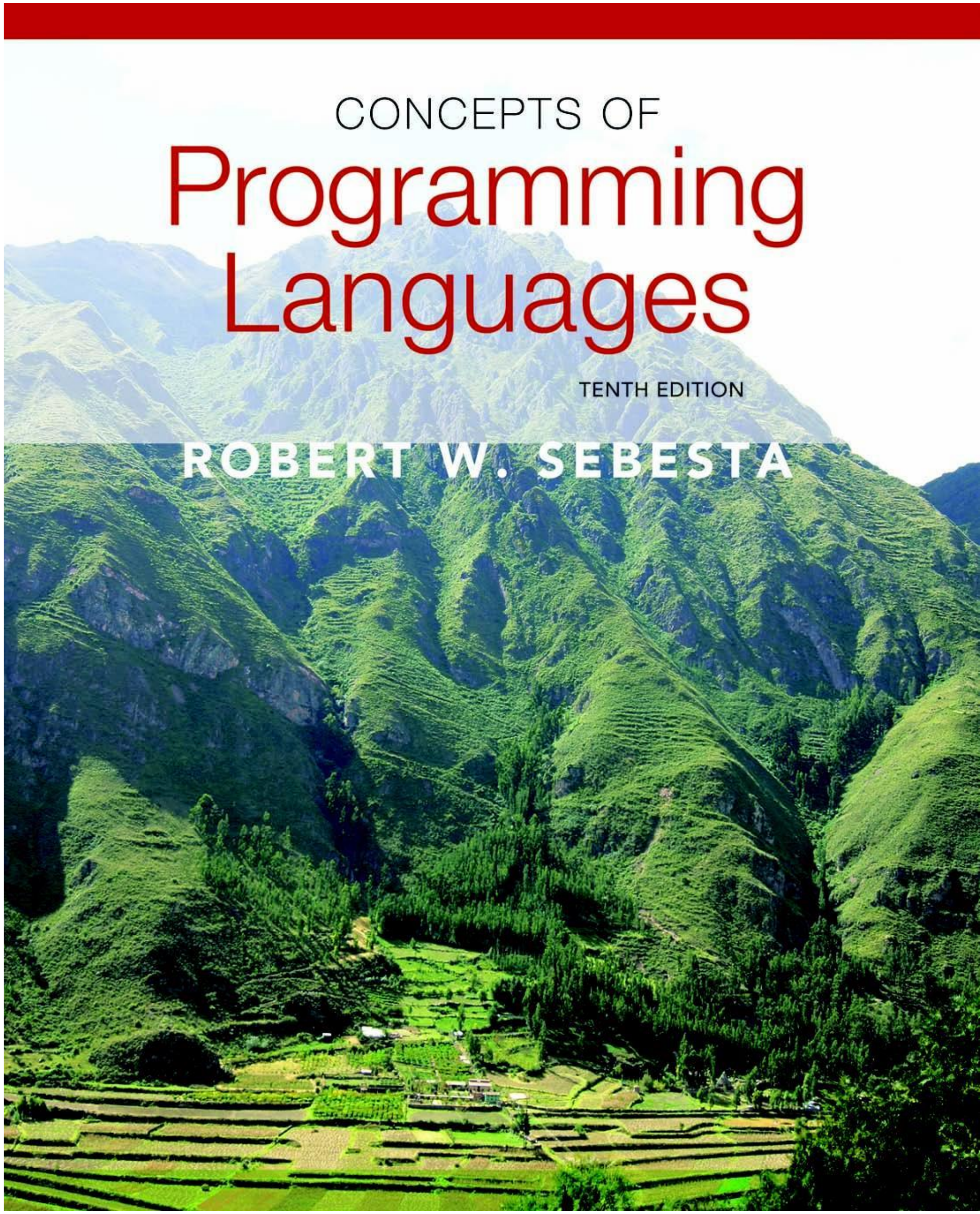
Chapter 13

Concurrency



Chapter 13 Topics

- Introduction
- Introduction to Subprogram-Level Concurrency
- Semaphores
- Monitors
- Message Passing
- Ada Support for Concurrency
- Java Threads
- C# Threads
- Concurrency in Functional Languages
- Statement-Level Concurrency

Introduction

- Concurrency can occur at four levels:
  - Machine instruction level
  - High-level language statement level
  - Unit level (두 개 이상의 subprogram 단위 동시 실행)
  - Program level
- Because there are no language issues in instruction- and program-level concurrency, they are not addressed here

Multiprocessor Architectures

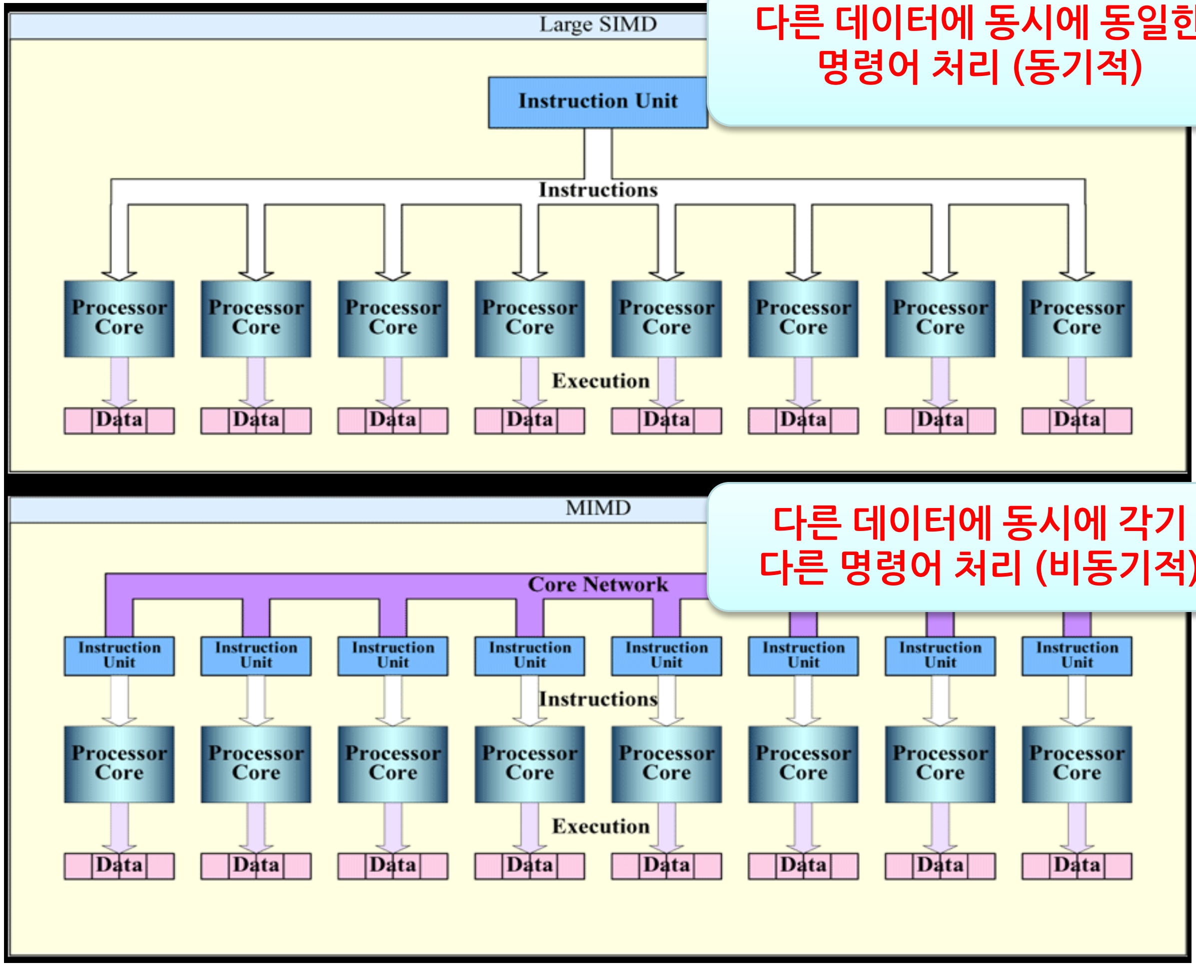
- Late 1950s - one general-purpose processor and one or more special-purpose processors for input and output operations
- Early 1960s - multiple complete processors, used for program-level concurrency
- Mid-1960s - multiple partial processors, used for instruction-level concurrency
- Single-Instruction Multiple-Data (SIMD) machines
- Multiple-Instruction Multiple-Data (**MIMD**) machines
- **A primary focus of this chapter** is **shared memory MIMD machines (multiprocessors)**

SIMD vs MIMD

SIMD - 병렬 프로세서의 한 종류로, 하나의 명령어로 여러 개의 값을 동시에 계산하는 방식. 벡터 프로세서에서 많이 사용되는 방식으로, 비디오 게임 콘솔이나 그래픽 카드와 같은 멀티미디어 분야에 자주 사용됨.

벡터 프로세서(Vector processor) 또는 어레이 프로세서(Array processor)는 벡터라고 불리는 다수의 데이터를 처리하는 명령어를 가진 CPU

MIMD
- 병렬화의 한 기법으로, MIMD를 사용하는 기계는 비동기적이면서 독립적으로 동작하는 여러개의 프로세서가 있음.
- 언제든지 각각의 다른 프로세서들은 각기 다른 데이터를 이용하는 각기 다른 여러 명령어들이 실행할 수 있음
- MIMD기계는 공유 메모리이거나 분산 메모리이며 이러한 분류는 MIMD가 어떻게 메모리를 이용하느냐에 따라 나뉨



다른 데이터에 동시에 동일한 명령어 처리 (동기적)

다른 데이터에 동시에 각기 다른 명령어 처리 (비동기적)

Categories of Concurrency

- Categories of Concurrency:
  - Physical concurrency - Multiple independent processors (multiple threads of control)
  - Logical concurrency - The appearance of physical concurrency is presented by time-sharing one processor (software can be designed as if there were multiple threads of control)
- Coroutines (quasi-concurrency) have a single thread of control
- A thread of control in a program is the sequence of program points reached as control flows through the program

Motivations for the Use of Concurrency

- Multiprocessor computers capable of physical concurrency are now widely used
- Even if a machine has just one processor, a program written to use concurrent execution can be faster than the same program written for nonconcurrent execution
- Involves a different way of designing software that can be very useful—many real-world situations involve concurrency
- Many program applications are now spread over multiple machines, either locally or over a network

Introduction to Subprogram-Level Concurrency

- A task or process or thread is a program unit that can be in concurrent execution with other program units
- Tasks differ from ordinary subprograms in that:
  - A task may be implicitly started
  - When a program unit starts the execution of a task, it is not necessarily suspended
  - When a task’s execution is completed, control may not return to the caller
- Tasks usually work together

Two General Categories of Tasks

- **Heavyweight tasks** execute in their own address space
- **Lightweight tasks** all run in the same address space – more efficient
- A task is **disjoint** if it does not communicate with or affect the execution of any other task in the program in any way
  - Disjoint task **(독립 태스크)**

Task Synchronization

- A mechanism that controls the order in which tasks execute
- Two kinds of synchronization
  – Cooperation synchronization
  – Competition synchronization
- Task communication is necessary for synchronization, provided by:
  - Shared nonlocal variables
  - Parameters
  - Message passing

Kinds of synchronization

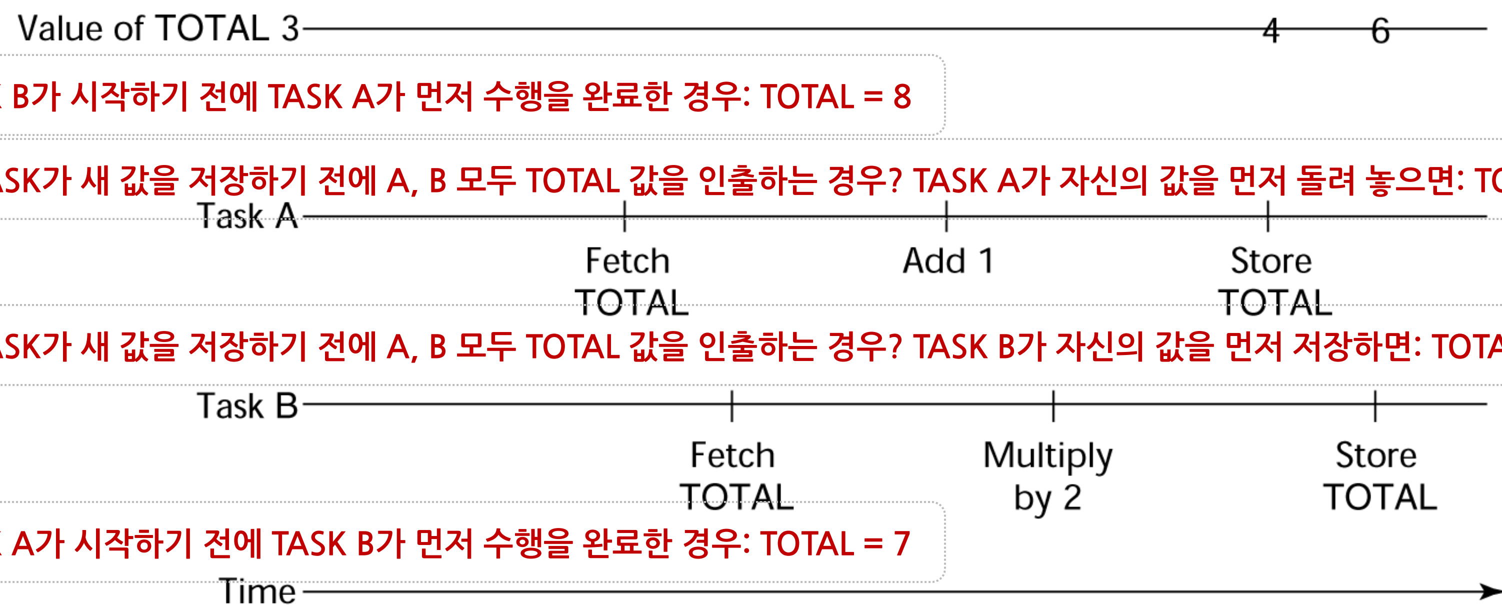
- **Cooperation**: Task A must wait for task B to complete some specific activity before task A can continue its execution, e.g., the producer-consumer problem

- **Competition**: Two or more tasks must use some resource that cannot be simultaneously used, e.g., a shared counter
  - Competition is usually provided by mutually exclusive access (approaches are discussed later)

Need for Competition Synchronization

Task A: `TOTAL = TOTAL + 1`
Task B: `TOTAL = 2 * TOTAL`

A와 B가 값을 변경하기 전 TOTAL = 3



TASK B가 시작하기 전에 TASK A가 먼저 수행을 완료한 경우: TOTAL = 8

각 TASK가 새 값을 저장하기 전에 A, B 모두 TOTAL 값을 인출하는 경우? TASK A가 자신의 값을 먼저 돌려 놓으면: TOTAL = 6

각 TASK가 새 값을 저장하기 전에 A, B 모두 TOTAL 값을 인출하는 경우? TASK B가 자신의 값을 먼저 저장하면: TOTAL = 4

TASK A가 시작하기 전에 TASK B가 먼저 수행을 완료한 경우: TOTAL = 7

경쟁 동기화가 없다면?

Race condition

- Depending on order, there could be four different results

Scheduler

- Providing synchronization requires a mechanism for delaying task execution
- Task execution control is maintained by a program called the **scheduler**, which maps task execution onto available processors

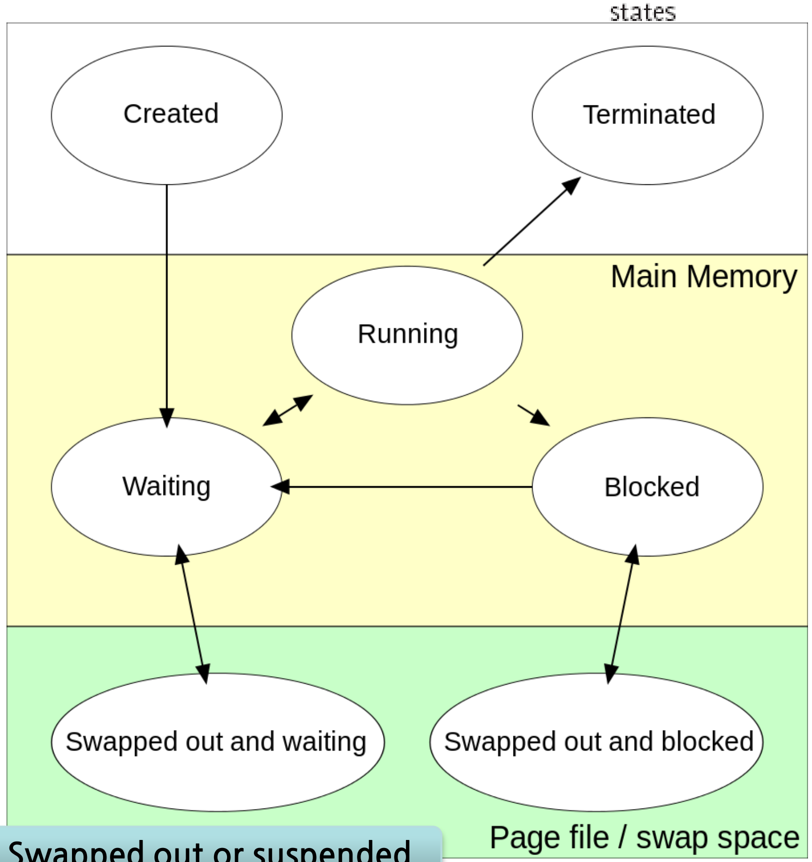
Task Execution States

- *New* - created but not yet started
- *R*eady - ready to run but not currently running (no available processor)
- *Running*
- *Blocked* - has been running, but cannot now continue (usually waiting for some event to occur)
- *Dead* - no longer active in any sense

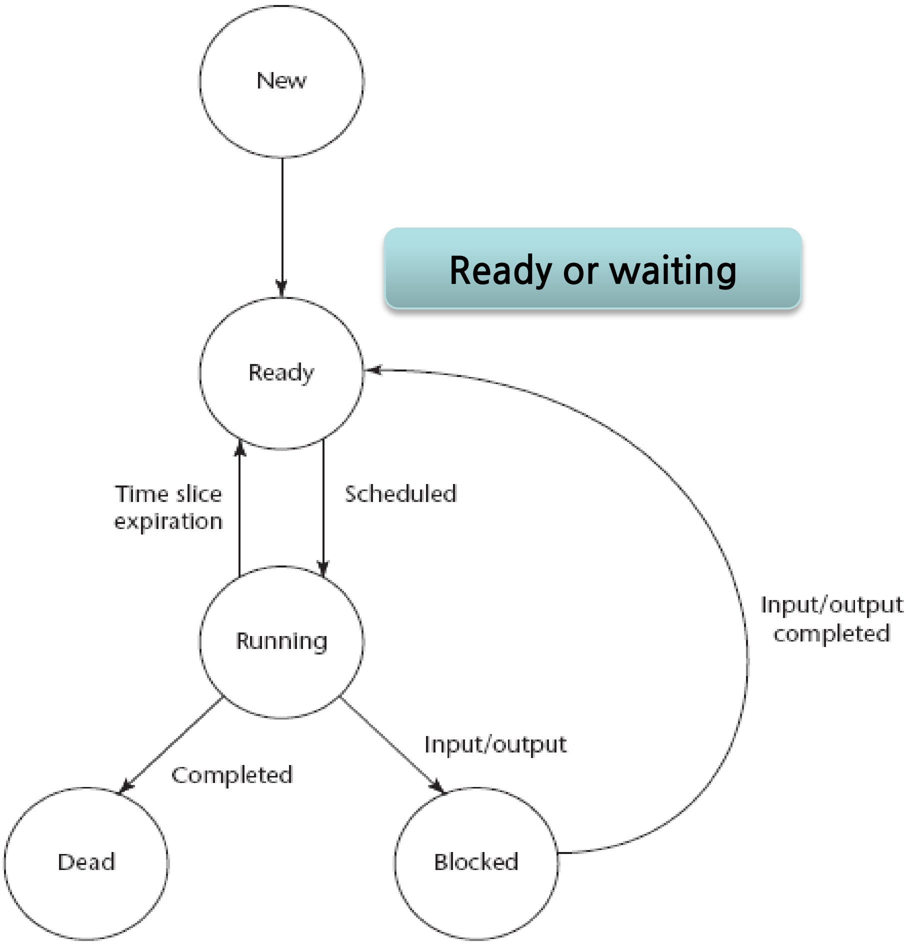
Task Execution States (continued)

Figure 13.2

Flow diagram of task states



Swapped out or suspended



Ready or waiting

https://en.wikipedia.org/wiki/Process_state#Ready_and_waiting

Liveness and Deadlock

- **Liveness** is a characteristic that a program unit may or may not have
  - In sequential code, it means the unit will eventually complete its execution
- In a concurrent environment, a task can easily lose its liveness
- If all tasks in a concurrent environment lose their liveness, it is called **deadlock**

Design Issues for Concurrency

- Competition and cooperation synchronization*
- Controlling task scheduling
- How can an application influence task scheduling
- How and when tasks start and end execution
- How and when are tasks created

* The most important issue

Methods of Providing Synchronization

- Semaphores
- Monitors
- Message Passing

Semaphores

- Dijkstra - 1965
- A **semaphore** is a data structure that consists of a **counter** (integer) and **a queue** for storing task descriptors
  - A task descriptor is a data structure that stores all of the relevant information about the execution state of the task
- Semaphores can be used to implement **guards** on the code that accesses shared data structures
- Semaphores have only two operations, **wait** and **release** (originally called **P** and **V** by Dijkstra)
- Semaphores can be used to provide both **competition and cooperation synchronization**

Cooperation Synchronization with Semaphores

- Example: A shared buffer
- The buffer is implemented as an ADT with the operations `DEPOSIT` and `FETCH` as the only ways to access the buffer
- Use two semaphores for cooperation: `emptyspots` and `fullspots`
- The semaphore counters are used to store the numbers of empty spots and full spots in the buffer

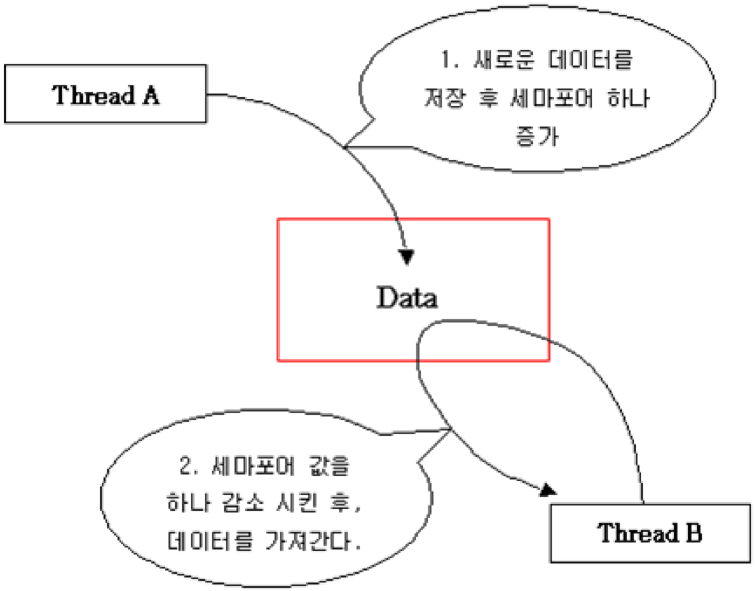
Cooperation Synchronization with Semaphores (continued)

- `DEPOSIT` **must first check** `emptyspots` **to see if there is room in the buffer**
- **If there is room, the counter of** `emptyspots` **is decremented and the value is inserted**
- **If there is no room, the caller is stored in the queue of** `emptyspots`
- **When** `DEPOSIT` **is finished, it must increment the counter of** `fullspots`

Cooperation Synchronization with Semaphores (continued)

- `FETCH` must first check **`fullspots`** (semaphore 변수) to see if there is a value
  - If there is a full spot, the counter of `fullspots` is decremented and the value is removed
  - If there are no values in the buffer, the caller must be placed in the queue of `fullspots`
  - When `FETCH` is finished, it increments the counter of `emptyspots`
- The operations of `FETCH` and `DEPOSIT` on the semaphores are accomplished through two semaphore operations named *wait* and *release*

세마포어 동기화 원리



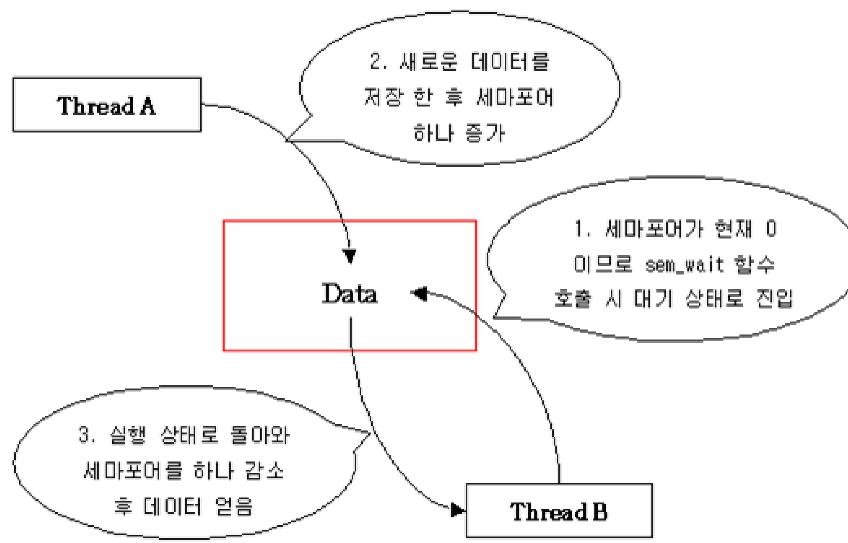
세마포어는 0과 1의 값을 가지는데,
현재 임계영역에 진입중인 쓰레드가 없으면 1을 가지고 있고,
어떤 쓰레드가 임계영역에 들어오게되면 0이 됨.

그래서 세마포어가 0인 상태에서 어떤 쓰레드가 임계영역에 들어오려고 하면 기다림

1. **쓰레드B 가 임계영역에 들어가려고 하지만, 세마포어가 0이라 기다림**

2. **쓰레드A 가 수행을 마치고, 세마포어를 다시 +1 하여 1로 만들어 놓음**

3. **다시 쓰레드B 가 세마포어를 -1 하고 자신이 임계영역으로 들어감**



Semaphores: Wait and Release Operations

```
wait(aSemaphore)
if aSemaphore's counter > 0 then    (counter 검사)
   decrement aSemaphore's counter
else
   put the caller in aSemaphore's queue
   attempt to transfer control to a ready task
     -- if the task ready queue is empty,
     -- deadlock occurs
end

release(aSemaphore)
if aSemaphore's queue is empty then  (대기 중인 task가 없는 경우)
   increment aSemaphore's counter
else
   put the calling task in the task ready queue
   transfer control to a task from aSemaphore's queue
end
```

Producer and Consumer Tasks

```
semaphore fullspots, emptyspots;
fullstops.count = 0;
emptyspots.count = BUFLEN;
task producer;
    loop
    -- produce VALUE --
    wait (emptyspots); {wait for space}
    DEPOSIT(VALUE);
    release(fullspots); {increase filled}
    end loop;
end producer;
task consumer;
    loop
    wait (fullspots);{wait till not empty}}
    FETCH(VALUE);
    release(emptyspots); {increase empty}
    -- consume VALUE --
    end loop;
end consumer;
```

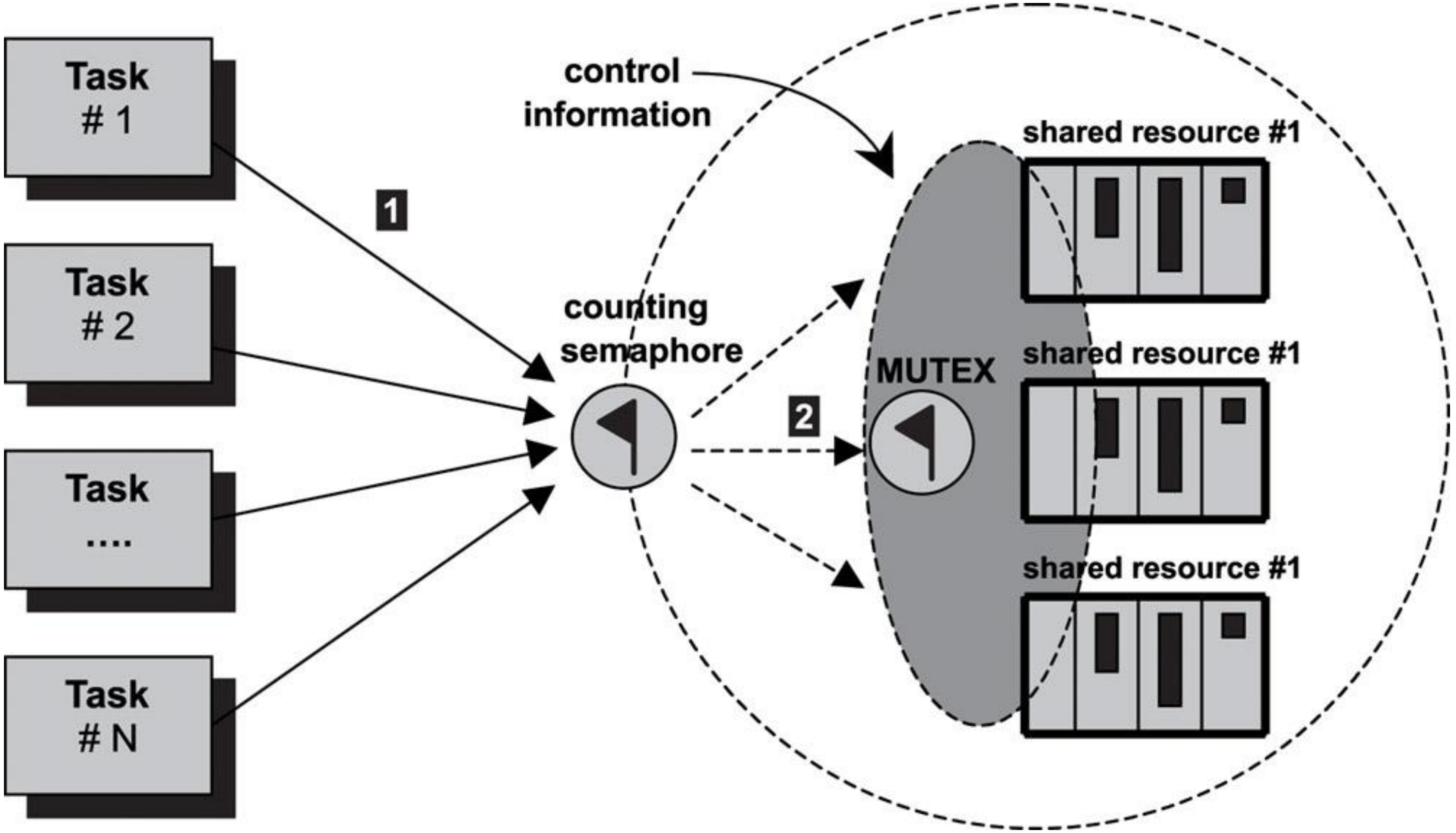
Competition Synchronization with Semaphores

- A third semaphore, named `access`, is used to control access (competition synchronization)
  - The counter of `access` will only have the values 0 and 1
  - Such a semaphore is called a *binary semaphore*
- Note that wait and release must be atomic!

Sharing Multiple Instances of Resources Using Counting Semaphores and Mutexes

뮤텍스(MUTEX)

- MUTual EXclusion으로 우리말로 해석하면 '상호 배제'라고 함
- 상호 배제해서 실행하는 것이며, 임계 구역(critical section)을 가진 스레드(thread)들이 동시에 실행되지 않고 서로 배제되어 실행되게 하는 기술



Producer Code for Semaphores

```
semaphore access, fullspots, emptyspots;
access.count = 0;
fullstops.count = 0;
emptyspots.count = BUFLEN;
task producer;
    loop
    -- produce VALUE —-
    wait(emptyspots); {wait for space}
    wait(access);     {wait for access)
    DEPOSIT(VALUE);
    release(access); {relinquish access}
    release(fullspots); {increase filled}
    end loop;
end producer;
```

Consumer Code for Semaphores

```
task consumer;
   loop
   wait(fullspots);{wait till not empty}
   wait(access);   {wait for access}
   FETCH(VALUE);
   release(access); {relinquish access}
   release(emptyspots); {increase empty}
   -- consume VALUE –-
   end loop;
end consumer;
```

Evaluation of Semaphores

- Misuse of semaphores can cause failures in cooperation synchronization, e.g., the buffer will overflow if the wait of `fullspots` is left out
- Misuse of semaphores can cause failures in competition synchronization, e.g., the program will deadlock if the release of `access` is left out