Operating Systems (OS)

Synchronization & Deadlock Avoidance

Prof. Eun-Seok Ryu (esryu@skku.edu)
Multimedia Computing Systems Laboratory
http://mcsl.skku.edu
Department of Immersive Media Engineering
Department of Computer Education (J.A.)
Sungkyunkwan University (SKKU)

전체 정리 - first

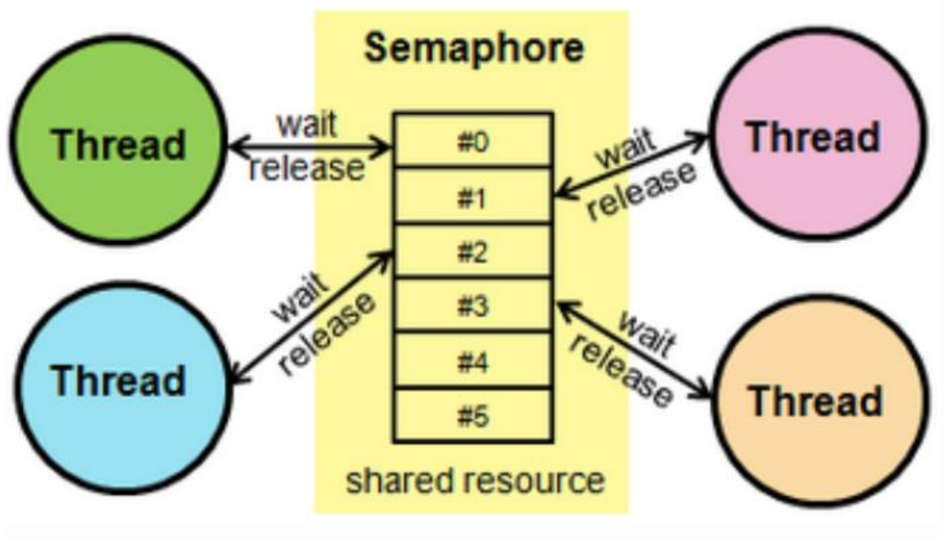
Q: 2 Processes가 하나의 자원에 접근할 때 관리 방법?

- Sync를 위한 용어 정리
  - 동기화
    - 한정적인 자원에 여러 thread들이 동시에 접근하면 문제 발생하므로 thread들에게 하나의 자원에 대한 처리 권한을 주거나 순서를 조정
  - Race Condition
    - 프로세스간 자원을 사용하려고 경쟁 -> 문제 발생할 여지
  - 임계구역(Critical Section: CS)
    - 공유자원에 접근하는 프로세스 내부의 코드 영역, 한 프로세스가 이 영역을 수행 중일 때 다른 프로세스가 같은 영역을 수행한다면 문제가 발생
  - 상호배제(Mutual Exclusion)
    - 한 프로세스가 공유 자원을 접근하는 CS영역 코드를 수행 중이면, 다른 프로세스들은 공유 자원을 접근하는 CS영역 코드를 수행할 수 없다는 조건

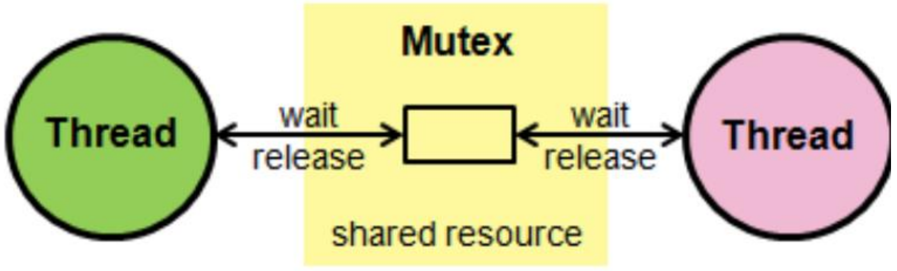
      1. Mutex: 일종의 Locking기법 (예: 화장실 열쇠)
      2. Semaphore: 동시에 리소스에 접근할 수 있는 허용 가능한 Counter의 개수 (예: 방에 의자가 5개면, 나머지는 문 밖)
      3. Monitor기법: JAVA는 가졌다지만 C에선 지원 안하므로 Skip

그림으로 이해

- Semaphore



- Mutex



그림출처: https://sycho-lego.tistory.com/11

Background – Why Sync가 필요한지

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:
  Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer `counter` that keeps track of the number of full buffers. Initially, `counter` is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {
    /* produce an item in next produced */

    while (counter == BUFFER_SIZE) ;
        /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Consumer

```
while (true) {
    while (counter == 0)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
        counter--;
    /* consume the item in next consumed */
}
```

Q: 이 경우 문제점?
-부디 미리 뒷 슬라이드 보면서 답을 찾지 말고 스스로 생각해보는 시간을 갖기를…-

Race Condition

- `counter++` could be implemented as

```
      register1 = counter
      register1 = register1 + 1
      counter = register1
```

- `counter--` could be implemented as

```
      register2 = counter
      register2 = register2 - 1
      counter = register2
```

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

S0: producer execute `register1 = counter`	{register1 = 5}
S1: producer execute `register1 = register1 + 1`	{register1 = 6}
S2: consumer execute `register2 = counter`	{register2 = 5}
S3: consumer execute `register2 = register2 - 1`	{register2 = 4}
S4: producer execute `counter = register1`	{counter = 6}
S5: consumer execute `counter = register2`	{counter = 4}

Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, … p_{n-1}}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc (공통의 리소스에 접근)
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design 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**

Critical Section

- General structure of process P_i

```
do {

    entry section

        critical section

    exit section

        remainder section

} while (true);
```

Algorithm for Process P_i

```
do {

 while (turn == j);

     critical section
 turn = j;

     remainder section
 } while (true);
```

- While문에 ; 가 붙어 있음에 주의
- I입장에서 J Turn(차례)면 Busy waiting,
- CS이후 J로 turn 수정 후
- 나머지 처리

처음에 서로 다 자기 차례가 아니면?

Q: 여러개 Processes가 하나의 자원에 접근할 때 하나만 독점하거나 하는 문제는 어찌 해결? 주차장 또는 화장실 예

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process ***P_i*** is executing in its critical section, then no other processes can be executing in their critical sections
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 selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **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
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the ***n*** processes

임계 구역 문제를 해결하기 위한 3가지 조건

- 상호 배제(Mutual exclusion) : 하나의 프로세스가 임계 구역에 들어가 있다면 다른 프로세스는 들어갈 수 없어야 한다.

- 진행(Progress) : 임계 구역에 들어간 프로세스가 없는 상태에서, 들어가려고 하는 프로세스가 여러 개 있다면 어느 것이 들어갈지를 적절히 결정해주어야 한다.

- 한정 대기(Bounded waiting) : 다른 프로세스의 기아(Starvation)를 방지하기 위해, 한 번 임계 구역에 들어간 프로세스는 다음 번 임계 구역에 들어갈 때 제한을 두어야 한다.

SW적 방법 있으나 복잡. 따라서, CPU가 지원하는 동기화 명령어를 활용

Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - **`int turn;`**
  - **`Boolean flag[2]`**

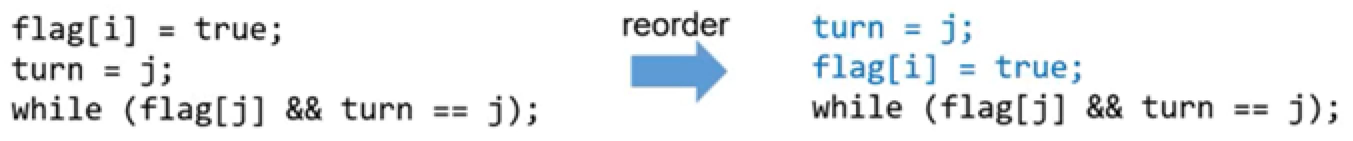
- The variable **`turn`** indicates whose turn it is to enter the critical section
- The **`flag`** array is used to indicate if a process is ready to enter the critical section. **`flag[i] = true`** implies that process P_i is ready!

Algorithm for Process P_i

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

While문에 ; 가 붙어 있음에 주의

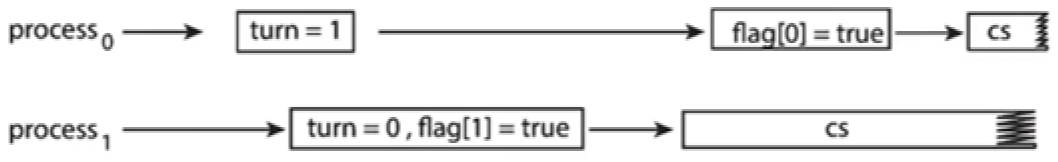
Q: 이런 코드면 충분?
Compiler가 메모리 접근을 고려해서 Assembly 레벨에서 순서를 바꾸면?



```
flag[i] = true;
turn = j;
while (flag[j] && turn == j);
```

reorder

```
turn = j;
flag[i] = true;
while (flag[j] && turn == j);
```



Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

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

이후 몇가지 Solution이 교재 등에 나와
있으나 잘 쓰이지 않는 관계로 Skip

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

스핀 락은 불필요한 CPU 싸이클을 낭비하므로 좋지 못한 구현

acquire() and release()

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

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

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

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process 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()`**
- Definition of the **`wait() operation`**

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

- Definition of the **`signal() operation`**

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

Semaphore Usage

- **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**

잘 안쓰이니 무시

- Can solve various synchronization problems
- Consider ***P_1*** and ***P_2*** that require ***S_1*** to happen before ***S_2***

  Create a semaphore "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

- Can implement a counting semaphore ***S*** as a binary semaphore

Semaphore Implementation

- Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the `wait` and `signal` code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

그럼 왜 썼는지...

Deadlock 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

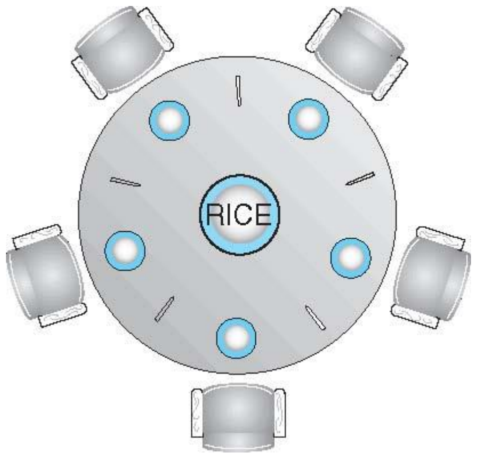
```
wait(S);              wait(Q);
wait(Q);              wait(S);
  ...                   ...
signal(S);            signal(Q);
signal(Q);            signal(S);
```

- **Starvation – indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

뒤에 예제
Dining Philosophers Problem

Deadlock

Dining-Philosophers Problem



아무도 굶기지 않기 – Solution

1. 최대 4명 제한
2. 젓가락 2개 모두 잡을 수 있을 때만 집도록 허용
3. 비대칭 해결안: 홀수/짝수 나눠서 왼쪽/오른쪽을 먼저 집게 함

- Philosophers spend their lives alternating thinking and eating
- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$
  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**

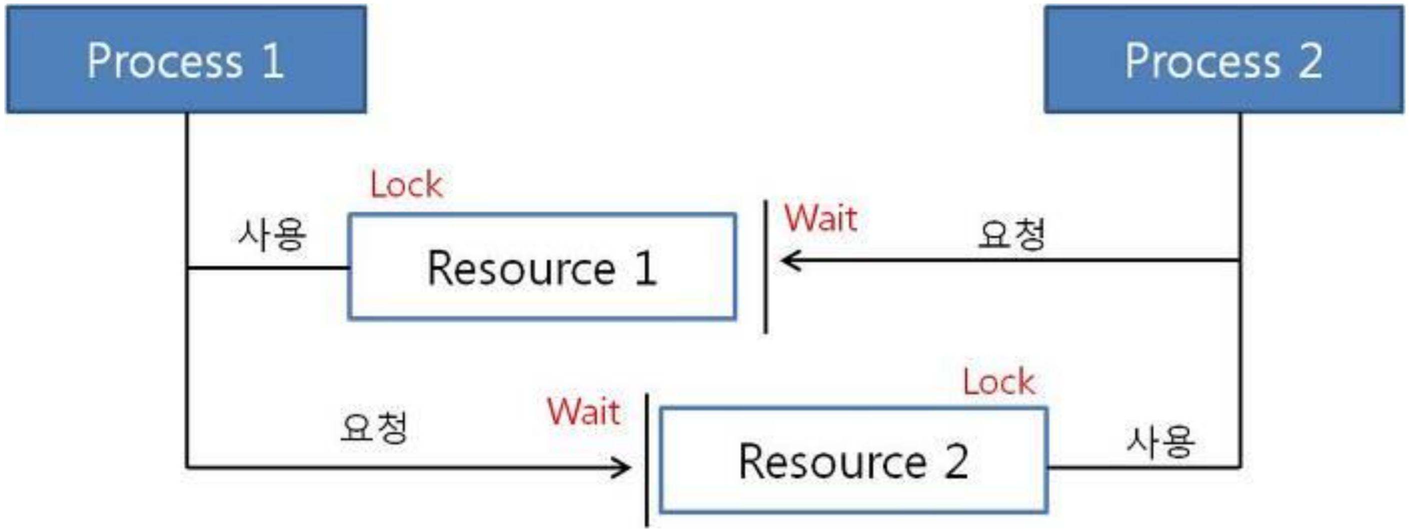
Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

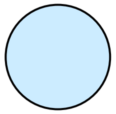
Need to prevent four conditions above.

Deadlock Example

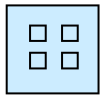


Resource Allocation Graphs

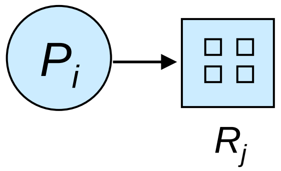
- Process



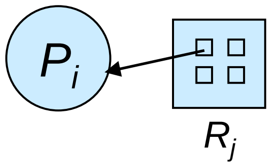
- Resource Type with 4 instances



- P_i requests instance of R_j



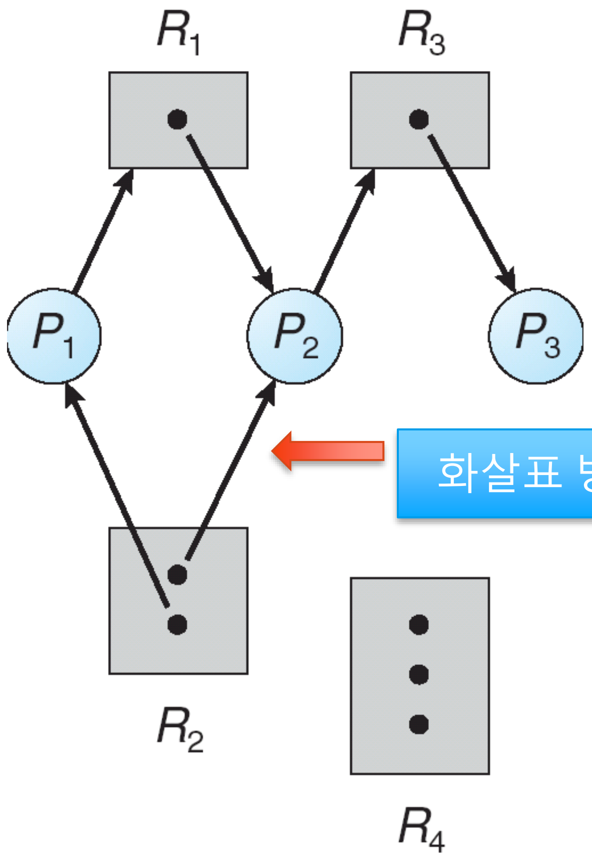
- P_i is holding an instance of R_j



request edge – directed edge $P_i \rightarrow R_j$ **: P(i)가 R(j)자원 요청함을 나타냄**

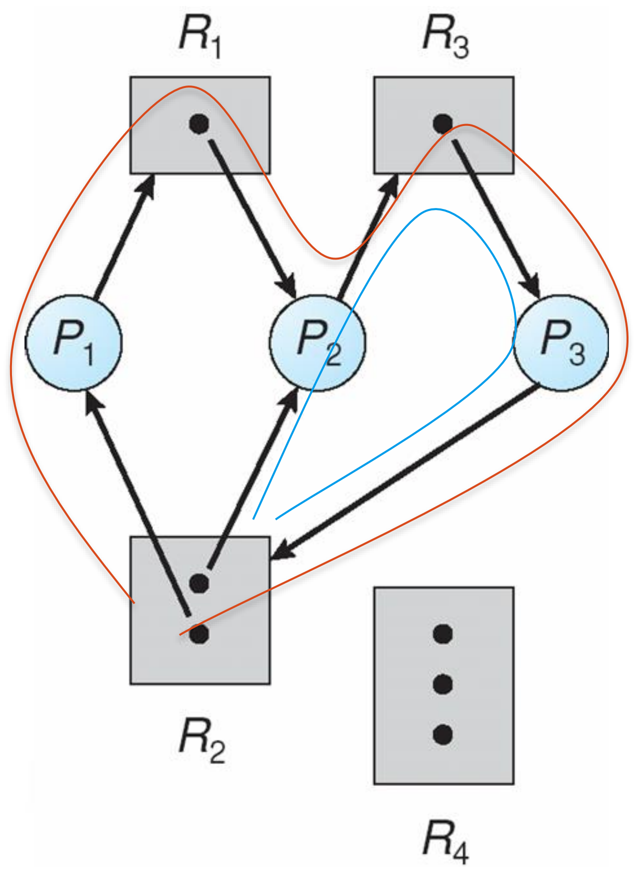
assignment edge – directed edge $R_j \rightarrow P_i$ **: R(j)가 P(i)에 할당되어 있음을 나타냄**

Example of a Resource Allocation Graph



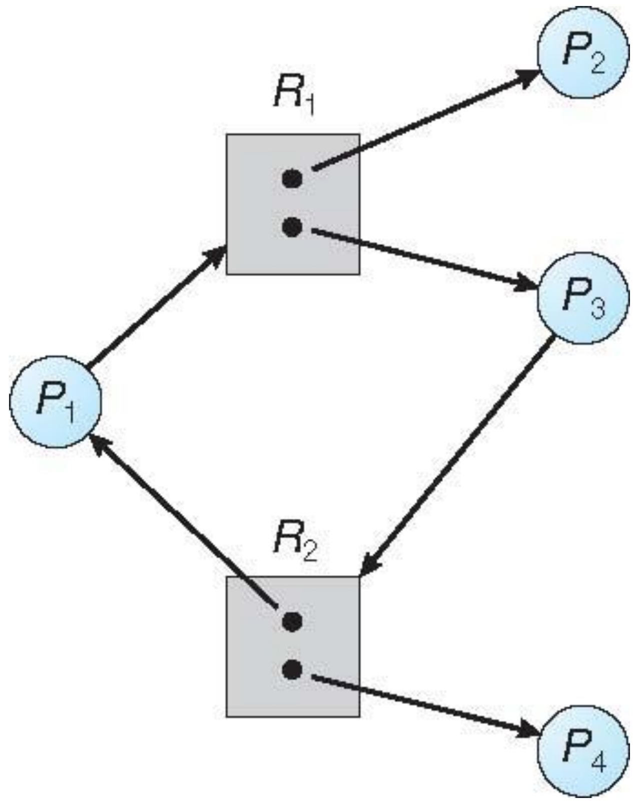
화살표 방향, Cycle이 아님

Resource Allocation Graph With A Deadlock



두군데 Cycles

Graph With A Cycle But No Deadlock



Q&A: Why?
P4…

Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if <u>several instances per resource type, possibility of deadlock</u>

Deadlock Prevention

- **Mutual Exclusion 부정** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
  - 시스템 내에 있는 상호 배타적인 모든 자원, 즉 독점적으로 사용할 수 있는 자원을 없애버리는 방법
  - 예) 여러 개의 프로세스가 read-only file과 같은 공유 자원을 사용할 수 있도록 한다. But, 기본적으로 공유불가능 자원이 있기에 Mutual Exclusion만으론 예방 불가능

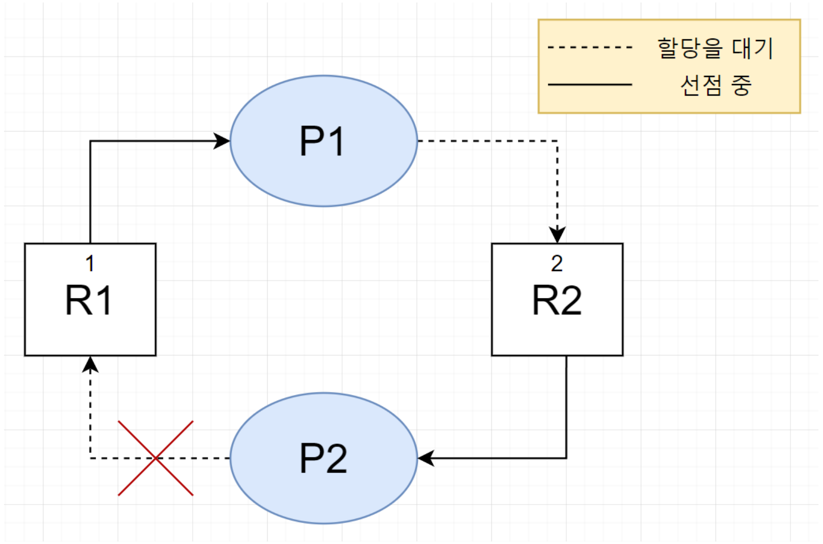
- **Hold and Wait 부정** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
  - Low resource utilization; starvation possible
  - 프로세스가 실행되기 전 필요한 모든 자원을 할당한다.
  - 문제점: 모든 자원 파악 어려움. 미리 확보 == 낮은 자원 활용성

Deadlock Prevention (Cont.)

- **No Preemption 부정**
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
  - 모든 자원에 대한 선점을 허용
  - 자원을 점유하고 있는 프로세스가 다른 자원을 요구할 때 점유하고 있는 자원을 반납하고, 요구한 자원을 사용하기 위해 기다리게 한다.

Deadlock Prevention (Cont.)

- **Circular Wait 부정**
  - impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration
  - 점유와 대기를 하는 프로세스들이 원형을 이루지 못하도록 막는 방법
  - 자원에 고유한 번호를 할당하고, 번호 순서대로 자원을 요구하도록 한다.



Source: https://velog.io/@chappi/

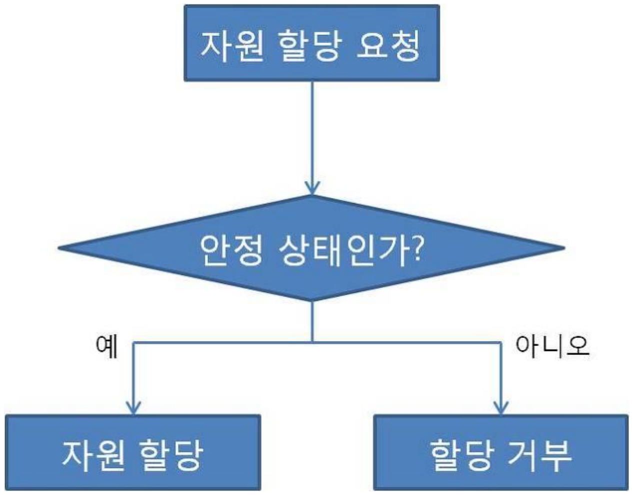
Deadlock Example

```
/* thread one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);
   pthread_exit(0);
}
/* thread two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);
   pthread_exit(0);
}
```

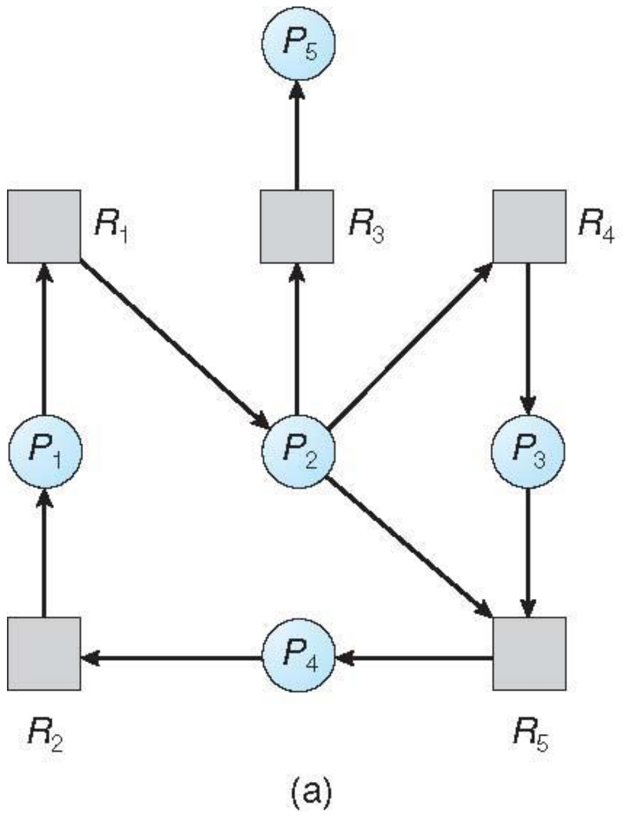
Banker's Algorithm (Dijkstra)

(In Korean)

- 프로세스가 자원을 요구할 때 시스템은 자원을 할당한 후에도 안정 상태로 남아있게 되는지를 사전에 검사하여 교착 상태를 회피하는 기법
- 안정 상태에 있으면 자원을 할당하고, 그렇지 않으면 다른 프로세스들이 자원을 해지할 때까지 대기함

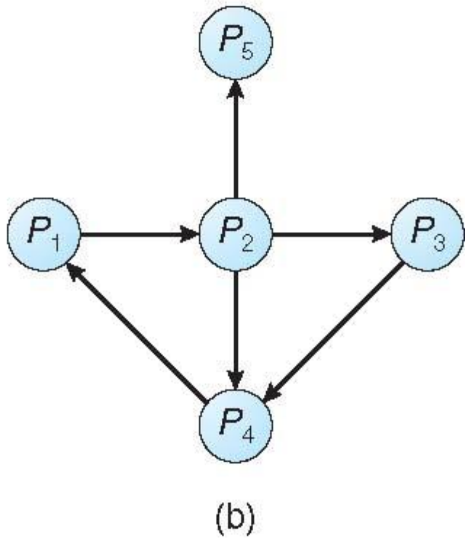


Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph

자원요청시 탐지 알고리즘 실행할 경우 문제점은? O(n^2)



Corresponding wait-for graph

(대응 되는 대기 그래프)

Recovery from Deadlock: Process Termination

문제점? – File쓰던 중이면?

- Abort all deadlocked processes
  - 교착 상태의 프로세스를 모두 중지

- Abort one process at a time until the deadlock cycle is eliminated
  - 교착 상태가 제거될 때까지 한 프로세스씩 중지

- In which order should we choose to abort?
  1. Priority of the process
  2. How long process has computed, and how much longer to completion
  3. Resources the process has used
  4. Resources process needs to complete
  5. How many processes will need to be terminated
  6. Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Preemption(자원선점)을 통해 교착상태 제거 -> 교착상태가 깨어질 때 까지 자원을 계속적으로 다른 프로세스가 선점하도록 함.
  - Selection of a victim – 비용최소화 선점 순서
  - Rollback – 프로세스 중지 후 재시작
  - Starvation – 동일한 프로세스가 항상 희생이 될 수 있음

Q&A