↳ 공유된 자원의 데이터를 여러 프로세스가 접근하는것을 막음

- Semaphore S ⇒ 두가지 연산에 의해서만 접근 가능.
  - ↳ 추상적으로 정의한것임.
  - ↳ 자원의 갯수
  - atomic

① p(S) : 공유자원 획득 (possess)
lock을 건다 : while (S≤0) do no-op; // S≤0 이면 연산X ⇒ wait
S--;

② V(S) : 자원 다 쓰고 반납하는 과정.
lock을 푼다. 반납.
: S++;

* 공유된 자원에 여러 (프로세스 / 쓰레드) 가 동시 접근하면 문제 발생
⇒ 공유된 자원속 소개의 데이터는 1번에 1개의 프로세스만 접근하도록 해야함.
⇒ 이를 위해 고안 된것이 Semaphore (세마포어)

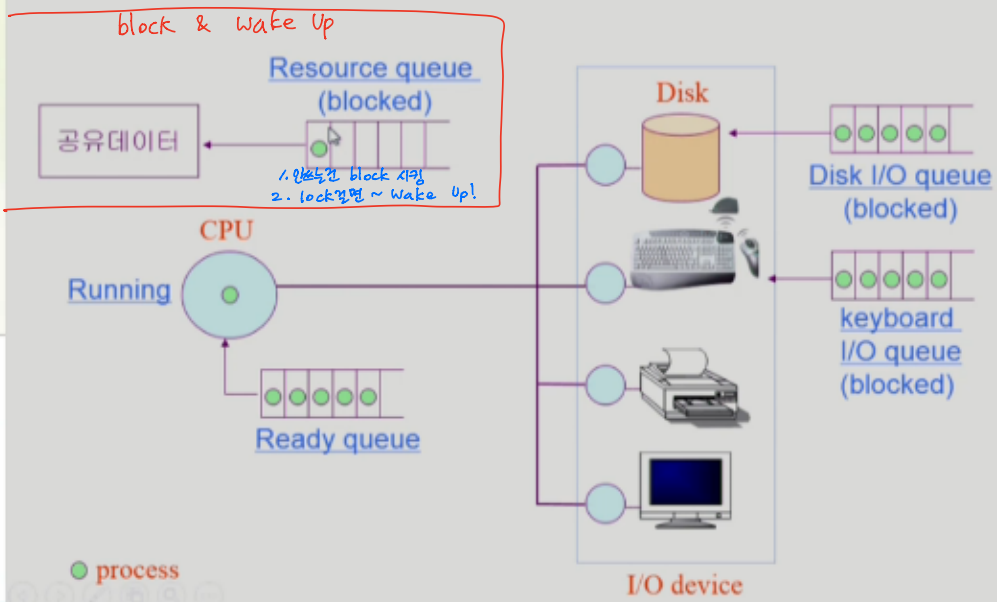
Critical Section of *n* Processes

Synchronization variable
semaphore mutex; /* initially 1: 1개가 CS에 들어갈 수 있다 */

Process Pi
```
do {
    P(mutex); (S--) /* If positive, dec-&-enter, Otherwise, wait. */
    critical section
    V(mutex); (S++) /* Increment semaphore */
    remainder section
} while (1);
```

busy-wait는 효율적이지 못함 = spin lock → lock을 못걸면 계속 spin 한다.
Block & Wakeup 방식의 구현 (next page)
= sleep lock
→ lock을 못걸면 걍 sleep 한다.

프로세스의 상태



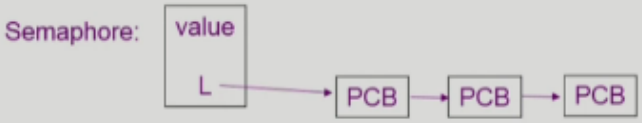
Block / Wakeup Implementation

- Semaphore를 다음과 같이 정의

```
typedef struct
{    int value;              /* semaphore */
     struct process *L;      /*process wait queue*/
} semaphore;
```

값이 있고 process가 있음.

- block과 wakeup을 다음과 같이 가정
  - ✓ **block** 커널은 block을 호출한 프로세스를 suspend시킴
    이 프로세스의 PCB를 semaphore에 대한 wait queue에 넣음
  - ✓ **wakeup(P)** block된 프로세스 *P*를 wakeup시킴
    이 프로세스의 PCB를 ready queue로 옮김



Implementation
block/wakeup version of P() & V()

- Semaphore 연산이 이제 다음과 같이 정의됨

```
P(S):   S.value--;      /* prepare to enter */
        if (S.value <0)  /* Oops, negative, I cannot enter*/
        {
                add this process to S.L;
                block();
        }
```

S.L에 들어가면 block 된다!
자원이 없으면 block 된다!
S.List

```
V(S):   S.value++;
        if (S.value <= 0) {
                remove a process P from S.L;
                wakeup(P);
        }
```

자원을 내놓았는데 S.value <= 0 라는건 block 된 놈이 있다. ⇒ 깨워줍다! (wakeUp!)

* block = sleep

Which is better?

- Busy-wait v.s. Block/wakeup
  - 만쓰면 CPU 반납.
  - overhead 든다 ← ready에서 block으로 바꿔야함.
- Block/wakeup overhead v.s. Critical section 길이
  - ✓ Critical section의 길이가 긴 경우 Block/Wakeup이 적당
  - ✓ Critical section의 길이가 매우 짧은 경우 Block/Wakeup 오버헤드가 busy-wait 오버헤드보다 더 커질 수 있음
  - ✓ 일반적으로는 Block/wakeup 방식이 더 좋음

busy-wait가 좋다!

* overhead = 특정 기능을 수행하기 위해 추가적으로 (부가적으로) 필요한 특정 자원.

* critical section = 여러 프로세스가 데이터를 공유하며 수행될 때, 각 프로세스에서 공유data를 액세스하는 프로그램 코드부분. (접근)

***Critical Section**

다중 프로그래밍 운영체제에서 여러 프로세스가 데이터를 공유하면서 수행될 때,
각 프로세스에서 공유데이터를 엑세스하는 프로그램 코드부분을 말합니다.

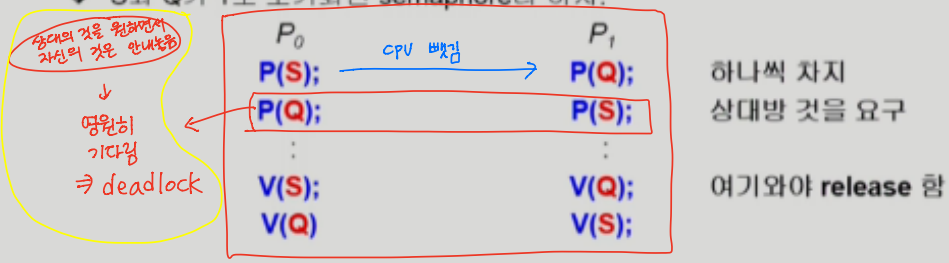
공유데이터를 여러 프로세스가 동시에 엑세스하면 시간적인 차이 때문에 잘못된 결과를 만들수 있습니다.
이를 막기 위해 한 프로세스가 위험부분을 수행하고 있을 때, 즉 공유데이터를 액세스하고 있을 때는
다른 프로세스들은 절대로 그 데이터를 접근할 수 없도록 해야합니다.

Two Types of Semaphores

- ***Counting semaphore*** ⇒ S = 5, 10.. 일때 사용
  - ✓ 도메인이 0 이상인 임의의 정수값
  - ✓ 주로 resource counting에 사용
- ***Binary semaphore*** (=mutex)
  - ✓ 0 또는 1 값만 가질 수 있는 semaphore
  - ✓ 주로 mutual exclusion (lock/unlock)에 사용

Deadlock and Starvation

- ***Deadlock*** ⇒ 데드락의 선조건. 세마포어나 뮤텍스가 일어났을때 발생할 수 있는 Error.
  - ✓ 둘 이상의 프로세스가 서로 상대방에 의해 충족될 수 있는 event를 무한히 기다리는 현상
- S와 Q가 1로 초기화된 semaphore라 하자.



- ***Starvation***
  - ✓ *indefinite blocking*. 프로세스가 suspend된 이유에 해당하는 세마포어 큐에서 빠져나갈 수 없는 현상

- 세마포어 vs 뮤텍스

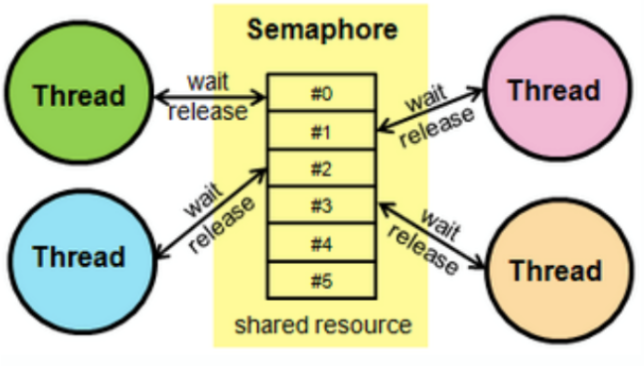
Semaphore(세마포어)

공유된 자원의 데이터를 여러 프로세스가 접근하는 것을 막는 것!

그리고 세마포어는 리소스의 상태를 나타내는 간단한 카운터라고 할 수 있습니다.
일반적으로 비교적 긴 시간을 확보하는 리소스에 대해 이용하게 되며, 유닉스 시스템의 프로그래밍에서 세마포어는 운영체제의 리소스를 경쟁적으로 사용하는
다중 프로세스에서 행동을 조정하거나 또는 동기화 시키는 기술입니다.

위 화장실 예제로 다시 살펴보면, 세마포어는 1개 이상의 열쇠라고 할 수 있습니다.
만약 화장실 칸이 4개이고 열쇠가 4개라면, 4명까지는 대기없이 바로 사용할 수 있고
그 다음 부터는 대기를 해야하죠. 이것이 바로 세마포어입니다.

그러므로 몇개의 세마포어로 구성해서 운영체제의 리소스를 경쟁적으로 사용할지는 꽤 중요한 이슈입니다.
그림으로 표현하면 아래와 같습니다.



Mutex(뮤텍스, 상호배제)

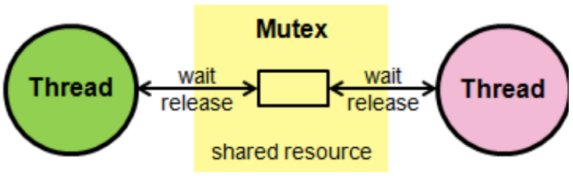
공유된 자원의 데이터를 여러 쓰레드가 접근하는 것을 막는 것!

즉, *Critical Section을 가진 쓰레드들의 Running tme이 서로 겹치지 않게 각각 단독으로 실행되게 하는 기술입니다.
다중 프로세스들이 공유 리소스에 대한 접근을 조율하기 위해 locking과 unlocking을 사용합니다.

간단히 말해, Mutex객체를 두 쓰레드가 동시에 사용할 수 없다는 말입니다.

위 화장실 예제로 다시 살펴보면, 뮤텍스는 무조건 1개의 열쇠만 가질 수 있습니다!

그림으로 표현하면 아래와 같습니다.



세마포어와 뮤텍스의 차이?

세마포어는 뮤텍스가 될수 있지만, 뮤텍스는 세마포어가 될 수 없습니다.

뮤텍스는 항상 열쇠 1개이고, 세마포어는 여러개 가질 수 있기 때문에
세마포어의 열쇠가 1개라면 뮤텍스와 같습니다.

세**마포어는 파일시스템 상 파일형태로 존재, 뮤텍스는 프로세스 범위입니다.**
즉, 프로세스가 사라질 때 뮤텍스는 clean up 됩니다.

세마포어는 소유할 수 없는 반면, 뮤텍스는 소유할 수 있습니다.

Dining-Philosophers Problem

젓가락 두개가 있어야 먹을 수 있음!

Synchronization variables

젓가락 공유

```
semaphore chopstick[5];
/* Initially all values are 1 */
```

Philosopher i

```
do {
    P(chopstick[i]);              왼쪽 젓가락
    P(chopstick[(i+1) % 5]);      오른쪽 젓가락
    ...
    eat();
    ...
    V(chopstick[i]);
    V(chopstick[(i+1) % 5]);
    ...
    think();
    ...
} while (1);
```

⇒ starvation

: D가 먹은 후 E는 먹을수있겠다 생각. D가 내려놓으니 A가 먹음.

→ ∴결국 E는 못먹음

→ starvation.

⇒ deadlock

: 모든 사람이 자기 왼쪽에 있는 젓가락을 들면 아무도 못먹음

→ deadlock!