파이썬 - HW9

임베디드스쿨1기

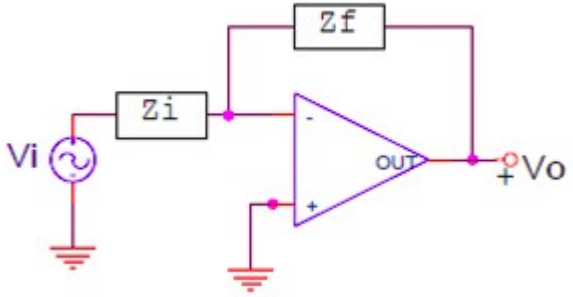
Lv1과정

2020. 10. 27

박하늘

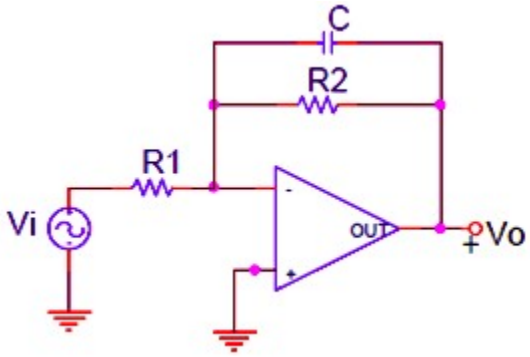
1. OP-AMP

1) 연산 증폭기 이득



$$H(\omega) = \frac{V_o}{V_i} = -\frac{Z_f}{Z_i}$$

2) 능동 저역통과 필터 (LPF)



$$Z = \frac{1}{j\omega C}$$

$$K = -\frac{R_2}{R_1} \text{ : passband gain}$$

$$\omega_0 = \frac{1}{R_2 C} \text{ : cutoff frequency}$$

$$H(\omega) = \frac{K}{1 + j\omega_0 / \omega}$$

- 저주파에서 C가 개방된 것과 동일
- 고주파에서 C가 단락된 것과 동일
- 차단 주파수: R2 임피던스와 C의 임피던스가 같게 되는 주파수
- 캐패시터의 임피던스
- 차단특성을 첨예하게 하고자 할 경우 동일한 차단 주파수를 가지는 LPF를 여러 개 직렬로 연결한다.

→ Sallen-Key

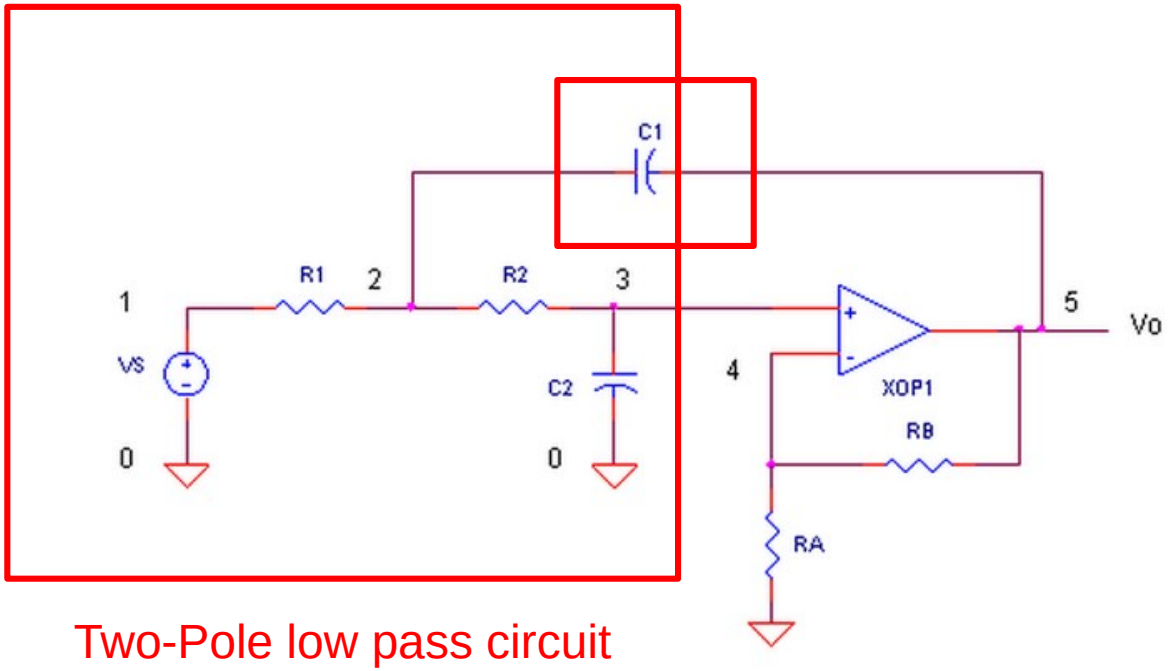
2. Sallen-key 2차 LPF

1) 특성

: OP-AMP 구성으로 수동 필터(R,C,L 수동 소자로만 이루어진)가 아닌 능동 필터로써 역할 함
(능동 필터: OP-AMP로 인해 자동적으로 임피던스 매칭이 이루어지면서 통과 대역의 증폭이 가능함)

①통과 대역에서 0dB(전압 1배, 그대로 통과)를 유지해 주다가 차단 주파수 이전에 서서히 떨어지기 시작
②차단 주파수 fc에서 -3dB(전압이 0.707배)를 기록,
③보드 선도가 -40dB/decade (주파수 10배 커질 수록 -40dB 감쇄 즉 전압값 0.01배)를 유지하며 감쇄
(RC필터에서, 필터 차수 1차당 -20dB/decade)

통과 대역 모서리 근처에서 예리한 응답을 갖음



Two-Pole low pass circuit
: -40dB/decade의 기울기를 가지는
2개의 저역통과 필터

$$f_c = \frac{1}{2\pi\sqrt{R_A R_B C_A C_B}}$$

만약 $R_A = R_B = R,\ C_A = C_B = C$

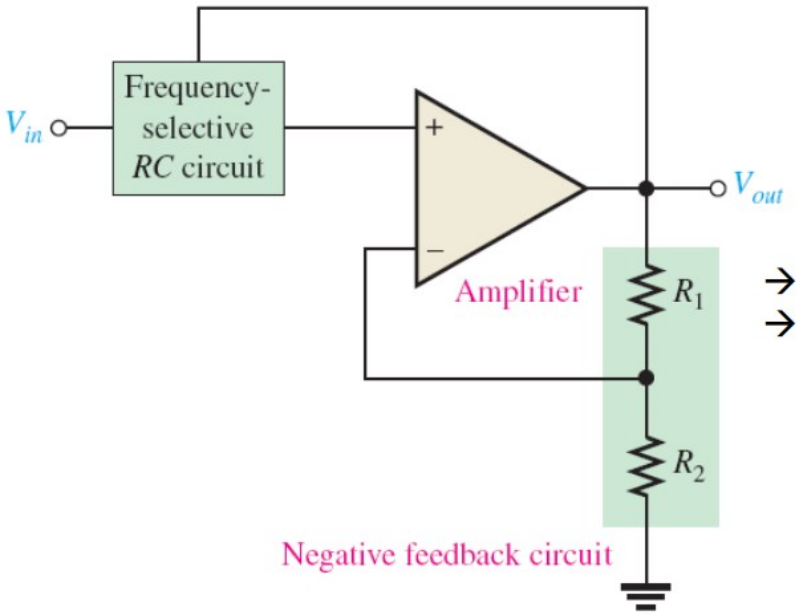
$$f_c = \frac{1}{2\pi RC}$$

2. Sallen-key 2차 LPF

2) Damping factor (DF)

: 필터의 응답 특성을 결정

$$DF = 2 - \frac{R_1}{R_2}$$



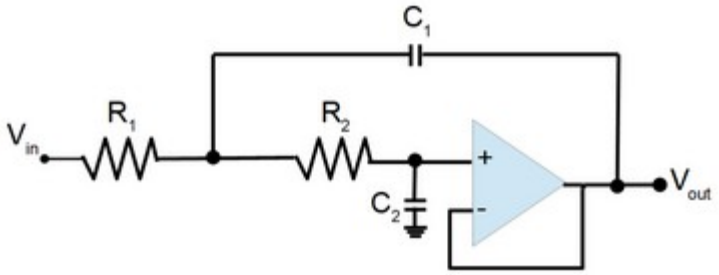
→ 부귀환 동작으로 필터응답에 영향
→ 필터의 차수(극점 수)와 관련된 응답 특성을 구현하기 위해서 댐핑 계수의 값이 요구

2. Sallen-key 2차 LPF

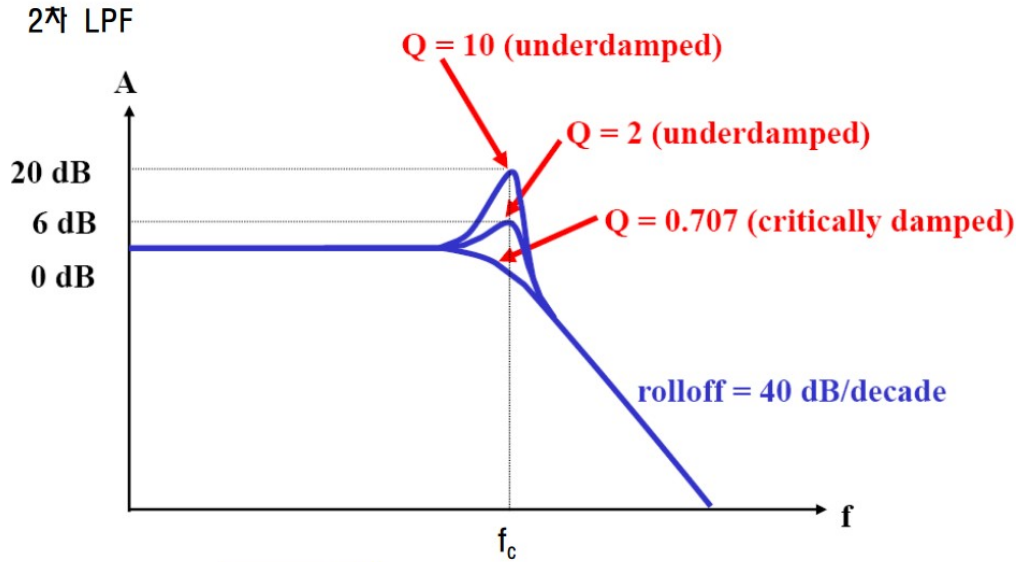
3) Q Factor (R&C)

: Q 필터의 품질. (High-Q: Q>10)

Q가 높다는 것은 첨예도가 높고, 그만큼 필터의 구현 설계 부담이 커지게 됨



$$Q = \frac{\sqrt{R_1 \cdot R_2 \cdot C_1 \cdot C_2}}{C_2 \cdot (R_1 + R_2)}$$



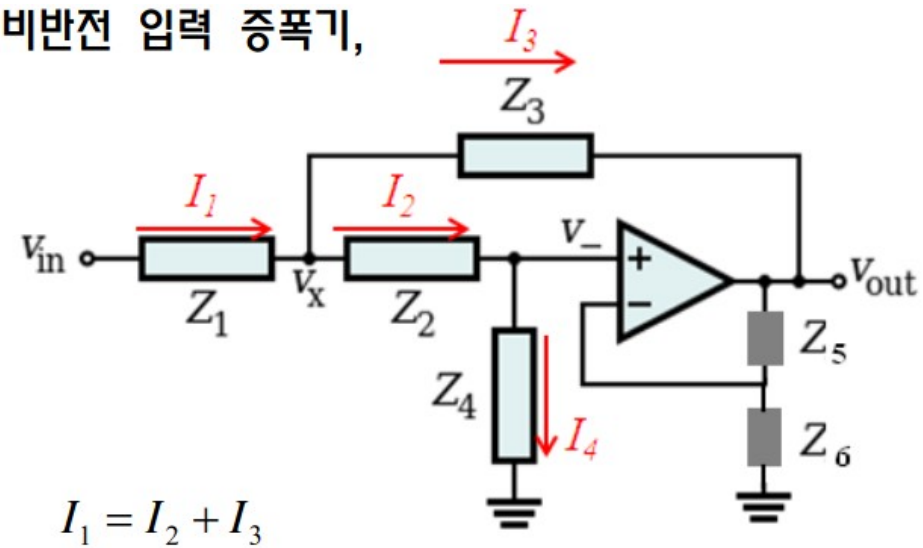
$$DF = 1/Q$$

필터의 보드 선도에서 체비셰프나 버터워쓰나 필터 구성 방식에 따라, 차단 주파수 근처에서 피크가 발생 할 수 있는데 피크가 커지면 커질 수록, 처음에는 Ringing이라고 해서 발진의 초기 증상이 나오다가 필터 구성을 피크가 엄청 높게 되면 발진한다.

2. Sallen-key 2차 LPF

4-1) Sallen-key 회로 계산

비반전 입력 증폭기,



$$I_1 = I_2 + I_3$$

$$I_2 = I_4$$

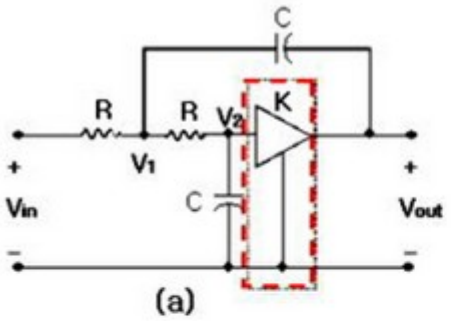
$$I_+ \approx I_- \approx 0$$

$$V_+ \approx V_- = \frac{Z_6}{Z_5 + Z_6} V_{out} = BV_{out}$$

포기하면 얻는 건 아무것도 없다.

2. Sallen-key 2차 LPF

4-2) Sallen-key 회로 계산



Sallen 과 key 필터

그림 20.2 : 2차 저역통과 필터

-그림 20.2 (a) 의 전달 함수

: V_1 과 V_2의 노드에서 KCL 을 적용하면

$$\frac{V_1 - V_{in}}{R} + sC(V_1 - V_{out}) + \frac{V_1 - V_2}{R} = 0 \qquad (A)$$

$$\frac{V_2 - V_1}{R} + sCV_2 = 0 \qquad (B)$$

식 (A)와 (B)에 $V_2 = V_{out} / K$ 를 대입한 다음 V_1을 소거하면

$$H(s) = \frac{V_{out}}{V_{in}} = K\frac{1}{(sCR)^2 + (3-K)sCR + 1} = K\frac{1}{\left(\frac{s}{\omega_0}\right)^2 + \frac{1}{Q}\left(\frac{s}{\omega_0}\right) + 1}$$

$$\omega_0 = \frac{1}{RC}, \quad Q = \frac{1}{3-K} \quad (K < 3)$$

2. Sallen-key 2차 LPF

4-3) Sallen-key 회로 계산

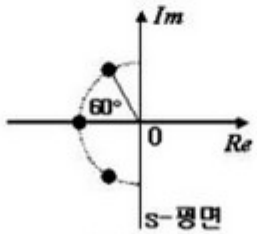
$$H(s) = \frac{V_{out}}{V_{in}} = K\frac{1}{\left(\frac{s}{\omega_0}\right)^2 + \frac{1}{Q}\left(\frac{s}{\omega_0}\right) + 1}, \quad \omega_0 = \frac{1}{RC}, \quad Q = \frac{1}{3-K} \quad (K<3)$$

- 증폭기의 이득 K가 3에 가까울수록 Q가 커지고 $\omega=\omega_0$ 에서 피크 발생
- K≥3 이면 발진
- $Q = \frac{1}{\sqrt{2}}$

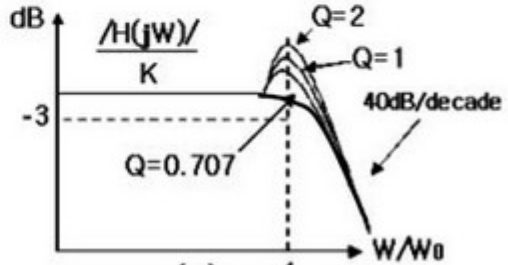
$$H(s) = K\frac{1}{(\frac{s}{\omega_0})^2 + \sqrt{2}\frac{s}{\omega_0} + 1}$$

$$\therefore |H(\omega)| = \frac{|K|^2}{\sqrt{1+\left(\frac{\omega}{\omega_0}\right)^4}}$$

: Butterworth filter



(b)

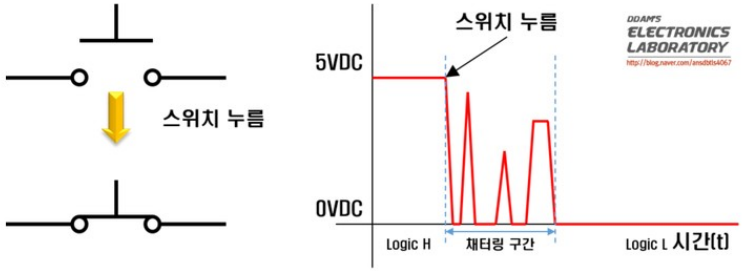


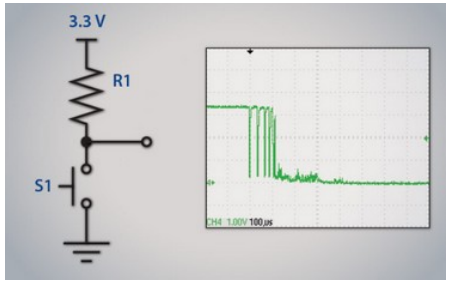
(c)

전달함수의 극점의 위치 이득을 K로 정규화 한 보드선도

3. 채터링(Chattering) 및 방지 RC회로

1) 전자회로 내의 스위치나 계전기의 접점이 붙거나 떨어질때 기계적인 진동으로 인해 접점이 붙었다 떨어지는 것을 반복하는 현상 → 스위치 신호를 입력받는 MCU 등에서 스위치 신호를 정상적으로 인식하지 못하는 상황 발생 .





2) [방지 1] Software적 방법

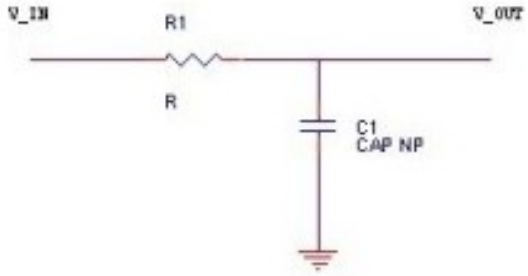
: Time delay 루틴을 이용한다.

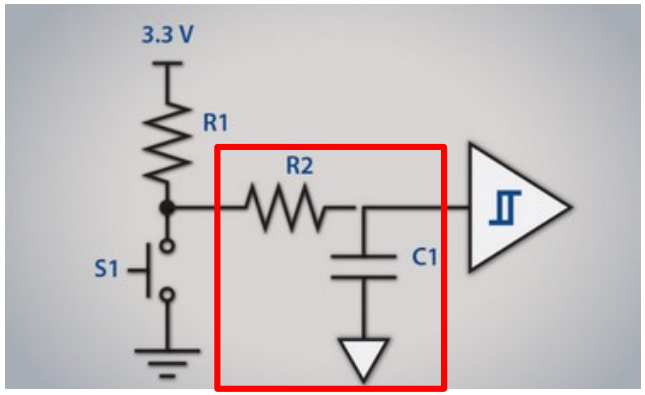
(키가 누르고 20ms 후 상태 확인, 키를 떼고 20ms 후 상태 확인후 인식)

3) [방지 2] Hardware적 방법

:스위치 전에 저항 및 캐패시터를 이용하여 RC 회로를 구성함으로써 High차단, Low 통과

- 주파수가 높을수록 Xc값이 작아짐, 주파수 커지면 Xc = 0 으로 고주파수가 모두 캐패시터쪽으로 빠져나감
- 이때, Low 신호는 저항쪽으로 흐름 → 캐패시터를 병렬로 연결하여 입력신호에 인 노이즈 (리플, 고주파)를 제거하여 출력 신호를 평활화 하는 용도로 사용.





4. [Review] Python Process Control

1) multiprocessing

```
import time
import multiprocessing

def countDown(x):
    while True:
        if x == 0:
            break

        print("CountDown ... %d" %x)
        x -= 1
        time.sleep(1)

p = multiprocessing.Process(target = countDown, args = (5,))
p.start()
```

```
CountDown ... 5
CountDown ... 4
CountDown ... 3
CountDown ... 2
CountDown ... 1
```

```
import time
import multiprocessing

class countDown(multiprocessing.Process): #상속
    def __init__(self, x): #생성자
        multiprocessing.Process.__init__(self)
        self.x = x

    def run(self):
        while True:
            if self.x == 0:
                break

            print("CountDown ... %d" % self.x)
            self.x -= 1
            time.sleep(1)

p = countDown(5)
p.start()
```

```
CountDown ... 5
CountDown ... 4
CountDown ... 3
CountDown ... 2
CountDown ... 1
```

4. [Review] Python Process Control

1) multiprocessing

: p1, p2, p3, p4의 프로세스가 있고, 그것을 총괄하는 Main Tread가 있는 꼴 → 마치 5개의 프로세스가
(cnt +=1 // 1초마다 갱신 되고 있음)
- 프로세스 각자의 주기대로 동작, context switching시 무결성 보장 못함

```python
import time
import multiprocessing

def process(message, interval):
    while True:
        print("I'm working ... %s" % message)
        time.sleep(interval)

p1 = multiprocessing.Process(target = process, args = ("p1",1,))
p2 = multiprocessing.Process(target = process, args = ("p2",3,))
p3 = multiprocessing.Process(target = process, args = ("p3",5,))
p4 = multiprocessing.Process(target = process, args = ("p4",2,))

p1.start()
p2.start()
p3.start()
p4.start()

cnt = 0

while cnt < 100:
    cnt += 1
    print("Main Thread")
    time.sleep(1)
```

```
I'm working ... p1
I'm working ... p2
I'm working ... p3
I'm working ... p4
Main Thread
I'm working ... p1
Main Thread
I'm working ... p1
I'm working ... p4
Main Thread
I'm working ... p1
I'm working ... p2
Main Thread
I'm working ... p1
I'm working ... p4
Main Thread
I'm working ... p1
I'm working ... p3
Main Thread
```

```
Main Thread
I'm working ... p1
I'm working ... p3
Main Thread
I'm working ... p1
I'm working ... p2
I'm working ... p4
Main Thread
I'm working ... p1
Main Thread
I'm working ... p1
I'm working ... p4
Main Thread
I'm working ... p2
I'm working ... p1
Main Thread
```

Context switching 하는데
p1의 비용이 더 커지는 부분

4. [Review] Python Process Control

1) multiprocessing

:생산자가 반드시 있어야 소비자가 동작하는 구조 (즉, 한 영역이 다른 특정한 영역에 종속될 수 밖에 없는 구조)

→ mutex, semaphore의 비동기 처리 방식과 유사 (운영체제가 내부적으로 기다리고 있음)

```
import time
import random
import multiprocessing

def producer(seq, output_queue): #생성자
    for obj in seq:
        output_queue.put(obj)
        print("[Producer] %d has been queued." %obj)
        randTime = random.randrange(5) + 1
        print("producer randTime = ", randTime)
        time.sleep(randTime)

def consumer(input_queue): #소비자
    while True:
        obj = input_queue.get() #데이터가 들어올때까지 블로킹
        print("[Consumer] %d "% obj)
        input_queue.task_done() #후속처리(데이터 들어오면)
        randTime = random.randrange(5) + 1
        print("consumer randTime", randTime)
        time.sleep(randTime)

shared_queue = multiprocessing.JoinableQueue()

cons_proc = multiprocessing.Process(target = consumer, args = (shared_queue,))
cons_proc.daemon = True #상주서비스 (터미널창 닫아도 상시 실행됨)
cons_proc.start()

seq = range(1,10)
producer(seq, shared_queue)

shared_queue.join()
```

```
[Consumer] 1
consumer randTime 1
[Producer] 1 has been queued.
producer randTime =  2
[Consumer] 2
consumer randTime 3
[Producer] 2 has been queued.
producer randTime =  4
[Consumer] 3
consumer randTime 1
[Producer] 3 has been queued.
producer randTime =  3
[Consumer] 4
consumer randTime 5
[Producer] 4 has been queued.
producer randTime =  1
[Producer] 5 has been queued.
producer randTime =  2
[Producer] 6 has been queued.
producer randTime =  3
[Consumer] 5
consumer randTime 2
[Producer] 7 has been queued.
producer randTime =  3
[Consumer] 6
consumer randTime 5
[Producer] 8 has been queued.
producer randTime =  2
[Producer] 9 has been queued.
producer randTime =  4
```

생산자가 2개 쌓임
(생산자가 더
빠르기 시작)

4.[Review] Spinlock Mutex Semaphore

1) Spinlock: 특정한 자료구조를 lock / unlock 함으로서 공유 데이터에 대한 접근 권한을 관리하는 방법. 하나의 컴포넌트만 접근할 수 있다. 권한을 획득하기 전까지 CPU는 무의미한 코드를 수행하는 busy waiting 상태로 대기하고 있다가 접근 권한을 얻으면 내부 코드를 수행하고 종료 후 권한을 포기한다.

```
1  // arch/arm64/kernel/debug-monitors.c
2  static DEFINE_SPINLOCK(step_hook_lock);
3
4  void register_step_hook(struct step_hook *hook)
5  {
6      spin_lock(&step_hook_lock);
7      list_add_rcu(&hook->node, &step_hook);
8      spin_unlock(&step_hook_lock);
9  }
```

2) Mutex : 권한을 획득 할 때 까지 busy waiting 상태에 머무르지 않고 sleep 상태로 들어가고 wakeup 되면 다시 권한 획득을 시도한다.시스템 전반의 성능에 영향을 주고 싶지 않고 길게 처리해야하는 작업인 경우에 주로 사용한다. 주로 쓰레드 작업에서 많이 사용된다.

```
1  // arch/arm64/mm/dma-mapping.c
2  mutex_lock(&iommu_dma_notifier_lock);
3  list_for_each_entry_safe(master, tmp, &iommu_dma_masters, list) {
4     if (data == master->dev && do_iommu_attach(master->dev,
5           master->ops, master->dma_base, master->size)) {
6      list_del(&master->list);
7      kfree(master);
8      break;
9     }
10 }
11 mutex_unlock(&iommu_dma_notifier_lock);
```

3) Semaphore: 스핀락, 뮤텍스와는 다르게 표현형이 정수형이며, 하나 이상의 컴포넌트가 공유자원에 접근하도록 허용할 수 있다.스핀락과 뮤텍스와 달리 세마포어는 해제(Unlock)의 주체가 획득(Lock)과 같지 않아도 된다.

```
1  // fs/btrfs/disk-io.c
2      down_read(&fs_info->cleanup_work_sem);
3      if ((ret = btrfs_orphan_cleanup(fs_info->fs_root)) ||
4          (ret = btrfs_orphan_cleanup(fs_info->tree_root))) {
5          up_read(&fs_info->cleanup_work_sem);
6          close_ctree(tree_root);
7          return ret;
8      }
9      up_read(&fs_info->cleanup_work_sem);
```

감사합니다.