**Ⅰ. Semaphores and Bounded Buffer[[1]](#footnote-1)**

**Motivation**

- Synchronization is a way of coordinating multiple concurrent activities that are using shared state.

- What are the right synchronization abstraction?

- To make it easy to build correct concurrent program.

**Definition of Semaphore**

- First defined by Dijkstra in the late 60`s

- A kind of generalized lock

- Main synchronization primitive[[2]](#footnote-2) use in UNIX

**Condition**

- Have a positive integer value

- No negative values

- Only operation are P and V – can’t read or write value, except to set it initially

- Operation must be atomic

- Have two operations

1. P() : an atomic operation that waits for semaphore to become positive, then decrements it by 1

2. V() : an atomic operation that increments semaphore by 1, waking up a waiting P, if any

**Binary Semaphore (Lock)**

- Instead of an integer value, has a boolean value.

- Execute : P waits until value is 1, then set it to 0. V sets value to 1, waking up a waiting P if any

**Two use of Semaphore**

**1. Mutal exclusion**

//initial value of 1

semaphore->P()

//critical section goes here // mutal exclusion

semaphore->V()

**2. Scheduling constraints**[[3]](#footnote-3)

-Semaphore provide a way for a thread to wait for something, Usually, in this case, the initial value of the semaphore is 0, but not always.

**Producer-consumer with a bounded buffer**

**Problem**

- Producer puts things into a shared buffer, consumer takes them out. Need synchronization for coordinating producer and consumer.

**Correctness [[4]](#footnote-4)constrains for solution**

Semaphore fullBuffers // Consumer must wait for producer to fill buffers, if none full (scheduling constraint)

Semaphore emptyBuffers // Producer must wait for consumer to empty buffers, if all full (scheduling constraint)

Semaphore mutex // Only one thread can manipulate buffer queue at a time (mutal exclusion)

**Semaphore Weakness**

- Critical Section 에 직접 접근하지 못하기 때문에 간접적으로 접근해야 하고, Semaphore를 많이 사용하는 경우 혼동이 생길 확률이 높다.

**Ⅱ. Monitors, Condition Variables**

**Motivation**

- Semaphores are huge set up. Used for both mutex and scheduling constraints. This makes the code hard to read, and hard to get right.

**Definition of Monitor**

- Have a lock[[5]](#footnote-5)

- Zero or more condition variable for managing concurrent access to shared data

**Condition Variable**

- A queue of threads waiting for something inside a critical section

- Have three operations

1. Wait() : release lock, go to sleep, re-acquire lock,

Releasing lock and going to sleep is atomic

2. Signal() : wake up a waiter, if any

3. Broadcast() : wake up all waiters

Rule : Must hold lock when doing condition variable operations

**Mesa vs Hoare monitors**

- Need to be careful about the precise definition of signal and wait

**Mesa-style**

- Most real operating system.

- Signaller keeps lock, processor

**Hoare-style**

- Most textbook.

- Signaller gives up lock, CPU to waiter; waiter runs immediately.

Waiter gives lock, processor back to singaller when it exits critical section or if it waits again

**What the best thing between Mesa-style and Hoare-style**

- General Principle, we almost always need to check the condition after the wait, with Mesa-style monitors. But if we use Hoare-style, we don’t do to check the condition after the wait. (in other words, use a “while” instead of an “if”)

AddToQueue()

lock.Acquire();

put item on queue;

condition.signal();

lock.Release();

}

RemoveFromQueue()

lock.Acquire();

while nothing on queue

condition.wait(&lock); // release lock; go sleep;

// re-acquire-lock

remove item from queue;

lock.Release();

return item;

}

**Ⅲ. Readers & Writers**

**Motivation**

- Shared database ( ex. bank balance, airline seat)

**Constraints**

- Reader can access database when no writers(Condition okToRead)

- Writers can access database when no readers or writers(Condition okToWrite)

- Only one thread manipulate state variable s at a time

**Solution**

- Reader

1. check in - wait until no writes

2. access database

3. check out – wake up waiting writer

- Writer

1. check in - wait until no readers or writers

2. access database

3. check out – wake up waiting readers or writer

To use monitor implements readers and writers

Monitor{

int //Shared Data  
AR // Active Reader  
AW // Active Writer  
WR // Waiting Reader  
WW // Waiting Writer

Condition okToRead = NIL;

Condition okToWrite = NIL;

Lock lock = FREE;

Reader\_CheckIn();

Reader\_CheckOut();

Writer\_CheckIn();

Writer\_CheckOut();

// Call CheckIn()

// Access Database

// Call CheckOut()

}

Reader\_CheckIn(){

lock->Acquire();

while( (AW+WW) > 0){

WR++;

okToRead->Wait(&lock);

WR--;

}

AR++;

lock->Release();

}

Reader\_CheckOut(){

lock->Acquire();

AR--;

if( AR == 0 && WW > 0){

okToWrite->Signal();

}

lock->Release();

}

–

\

Writer\_CheckOut(){

lock->Acquire();

while( (AW+AR) > 0 ){

WW++;

okToWrite->Wait(&lock);

WW--;

}

AW++;

lock->Release();

}

Writer\_CheckOut(){

lock->Acquire();

AW--;

if( WW > 0 ){

okToWrite->Signal();

}else if(WR > 0){

okToRead->BroadCast();

}

lock->Release();

}

■ Critical Section = Mutal Exclusion

**Ⅳ. Implement Condition Variable using Semaphore**

**Step1. Dead Lock**

Wait(){

semaphore->P(); // Problem

}

Signal(){

semaphore->V();

}

Program Execute

lock->Acquire()

if not satisfied condition

condition->Wait()

semaphore->P();

※ Lock 을 가진 상태로 P()를 실행하여 Lock 을 풀어줄 누군가가 없다.

**Step2. Solving Problem (Dead Lock)**

Wait(){

Wait(Lock \*lock){

lock->Release()

semaphore->P()

lock->Acquire()

}

}

Signal(){

semaphore->V();

}

Program Execute

lock->Acquire()

if not satisfied condition

condition->Wait(lock)

lock->Release()

semaphore->P()

lock->Acquire()

※ Dead Lock 문제 해결

But Semaphore를 항상 사용하고 있는 상태는 아니다. 하지만 Wait 를 깨워주는 역할은 수행됨으로 상관없다.

**Step3. First execute Signal() after execute Wait()**

Wait(){

… // No problem here

}

Signal(){

semaphore->V(); // Problem

}

Program Execute

lock->Acquire()

condition->Singal()

semaphore->V()

※ 아무것도 실행되지 않았음에도 수행할 것이 있다고 Wait()에 알려주게 된다

- Wait()를 먼저 실행시킨 것이 아닌 Signal()을 실행시킨 경우 문제가 발생된다. 이 때 Signal()의 경우 아무 의미 없이 지나가야 하는 공간인데 semaphore->V() 때문에 흔적이 생긴다. 이 흔적은 후에 Wait()가 실행되었을 때 Queue에 Thread가 저장되어 있으니 사용하라고 명령해준 것이 된다.

**Step4. Solving Problem**

Wait(){

… // No problem here

}

Signal(){

if semaphore queue is not empty

semaphore->V();

}

Program Execute

lock->Acquire()

condition->Singal()

※ 아무것도 실행되지 않고 지나간다.

- Semaphore queue에 조건을 추가 해줌으로 써 Signal이 먼저 실행 된 경우 아무것도 흔적을 남기지 않고 실행할 수 있다.

**Step4. Semaphore Queue in Condition Variable Can you access?**

Queue Condition

* Can’t read or write value, except to set it initially

- 위의 조건 때문에 Condition Variable에서 semaphore queue is not empty 에 대해서 알 수 있는 방법이 없다. 이 점을 해결해야 한다.

**Step5. Solving Problem**

- Using Semaphore Lock and Integer Value

- Confirm Homework #2

**Ⅴ. DeadLock**

**Definition**

- Circular waiting for resources

**Resources**

- Passive, things needed by thread[[6]](#footnote-6) to do its job

- CPU, disk, space, memory, Mutal Exclusion(Critical Section)

- Two kinds of resources

1. Preemptable – Can take it away ( CPU )

2. Non-preemptable [[7]](#footnote-7)– Must leave with thread ( disk space, Mutal Exclusion )

**vs Starvation[[8]](#footnote-8)**

- thread waits indefinitely ( some other threads are using resources )

**Motivation**

- Deadlock can happen with any kind of resources

**Conditions**

- Without all of these, can’t have deadlock

1. Mutal exclusion condition

2. Hold and wait condition

3. No preemption condition

4. Circular wait condition

1. Bounded Buffer - 경계가 있는 버퍼 [↑](#footnote-ref-1)
2. Primitive – 원시적인 [↑](#footnote-ref-2)
3. Constraint – 제한, 제약 [↑](#footnote-ref-3)
4. Correctness – 정확한 [↑](#footnote-ref-4)
5. The **lock** provides mutual exclusion to the shared data.

   Lock::Acquire : wait until lock is free, then grab it

   Lock::Release : unlock, wake up anyone waiting in Acquire

   Rules of using a lock

   - Always acquire before accessing shared data structure

   - Always release after finishing with shared data

   - Lock is initially FREE (Before Lecture) [↑](#footnote-ref-5)
6. Thread : Active ↔ Resources Passive [↑](#footnote-ref-6)
7. Non-preemptable – 절대 공유하지 못하는 것 들을 말함 [↑](#footnote-ref-7)
8. Strarvation 의 경우 발생되지 않을 사건을 무한히 기다리는 것이고 Deadlock의 경우는 순환적으로 무한히 도는 것이다. ( Deadlock implies starvation, but not vice versa ) [↑](#footnote-ref-8)