

# Lecture 13

## Synchronization

이재진

서울대학교 데이터사이언스대학원

서울대학교 공과대학 컴퓨터공학부

<http://aces.snu.ac.kr/~jlee>



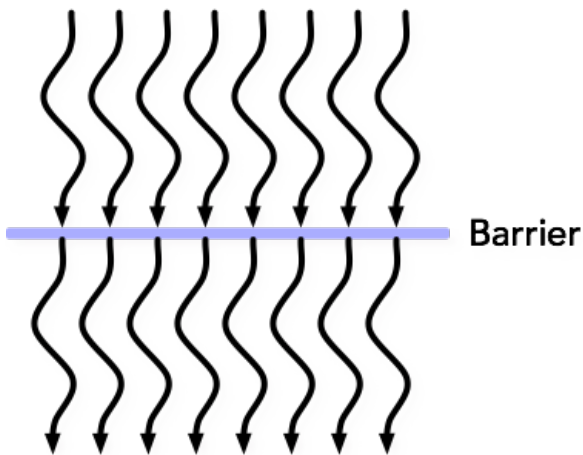
# 동기화가 필요한 이유

- 스레드 간 또는 프로세스 간 실행되는 순서를 정하기 위해
- 상호배제(Mutual Exclusion)를 달성하기 위해



# Barrier

- Barrier가 있는 지점에 도달한 스레드는 실행을 중단하고 다른 모든 스레드가 같은 배리어에 도달하기를 기다린 다음 실행을 계속함
- 같은 함수(코드)를 동시에 실행할 때 흔하게 이용됨
  - SPMD



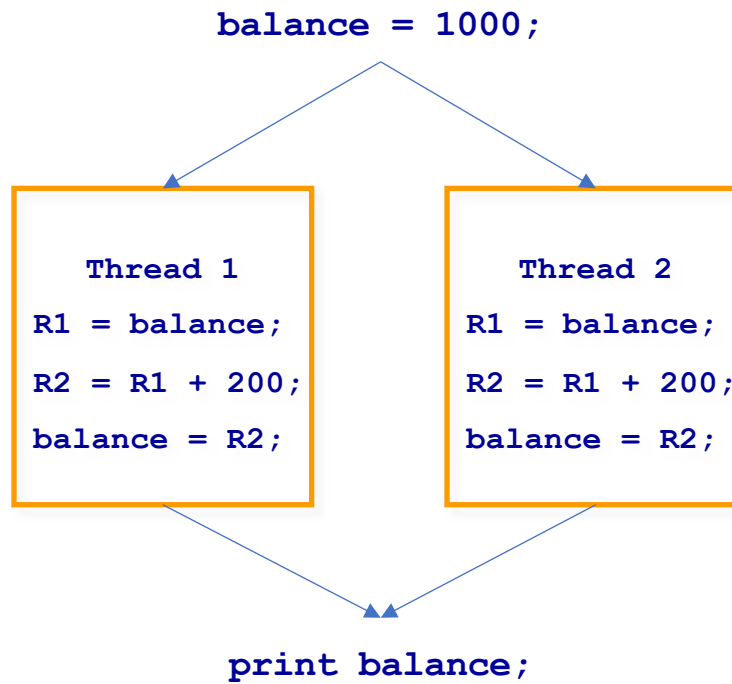
```
work_on_my_problem();  
Barrier();  
get_result_from_others();  
Barrier();
```

```
if (my_id == 0) {  
    work(); //초기화  
    barrier();  
} else {  
    barrier();  
}  
...
```



# Data Race

- 두 스레드가 변수 `balance`를 공유
- R1과 R2는 각 스레드에 대해 로컬 변수



# Data Race (cont'd)

시간 ↓

Thread 1	Thread 2	Balance
R1 = balance (1000)		1000
R2 = R1 + 200 (1200)		1000
balance = R2 (1200)		1200
	R1 = balance (1200)	1200
	R2 = R1 + 200 (1400)	1200
	balance = R2 (1400)	1400



# Data Race (cont'd)

시간 ↓

Thread 1	Thread 2	Balance
R1 = balance (1000)		1000
	R1 = balance (1000)	1000
	R2 = R1 + 200 (1200)	1000
R2 = R1 + 200 (1200)		1000
balance = R2 (1200)		1200
	balance = R2 (1200)	1200

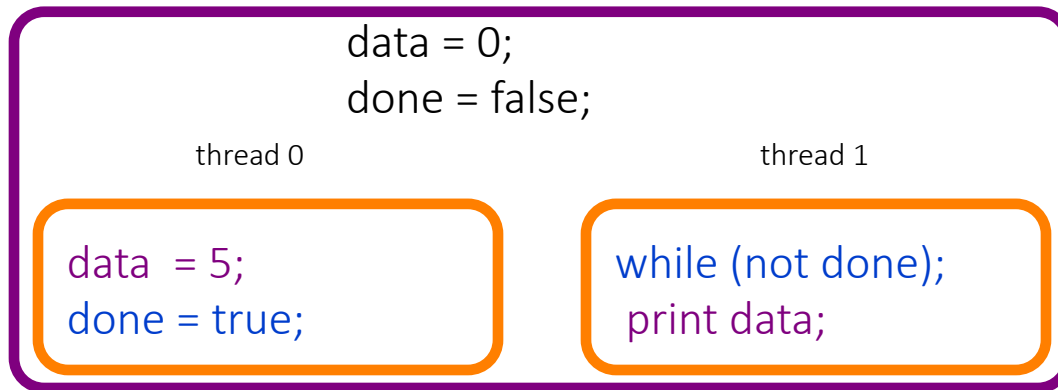


## Data Race (cont'd)

- 데이터 레이스가 일어나는 조건
  - 두 개 이상의 스레드가 같은 메모리 위치를 액세스하고
  - 그 중 하나의 스레드가 그 위치에 쓰기를 하고,
  - 그 위치가 동기화에 의해 보호되지 않을 때
- 액세스 순서가 비결정적(non-deterministic)
  - 실행 될 때마다 다른 결과가 출력될 수 있음
- 어떤 데이터 레이스는 의도적으로 사용
  - 비지 웨이트(busy-wait) 동기화
  - Lock-free 알고리즘
- 하지만 보통의 경우, 데이터 레이스는 버그



# 비지 웨이트(Busy Wait) 동기화





# Thread Safety

- 두 개 이상의 스레드가 어떤 함수를 동시에 호출했을 때 그 함수가 항상 올바르게 동작하면 그 함수를 Thread Safe 하다고 함
  - Thread-safe library



# 아토미시티(Atomicity)

- 한 묶음의 연산들이 아토믹(atomic)하다는 것은 모든 연산이 다 수행되거나 하나도 수행되지 않는 것을 의미(all or nothing)
  - 부분적인 수행으로 인한 결과를 볼 수 없음



## 뽕꺏꺏꺏꺏(Mutual Exclusion)

- 항상 많아야 하나의 스레드만 어떤 코드부분를 실행할 수 있을 때 그 코드부분이 상호배제(mutual exclusion)적으로 실행된다고 함
- 상호배제에 의해 atomicity를 보장할 수 있음
- 어떤 코드부분을 동기화로 보호하여 아톰믹하게 실행할 때 그 코드부분을 크리티컬 섹션(critical section)이라고 함

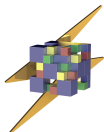
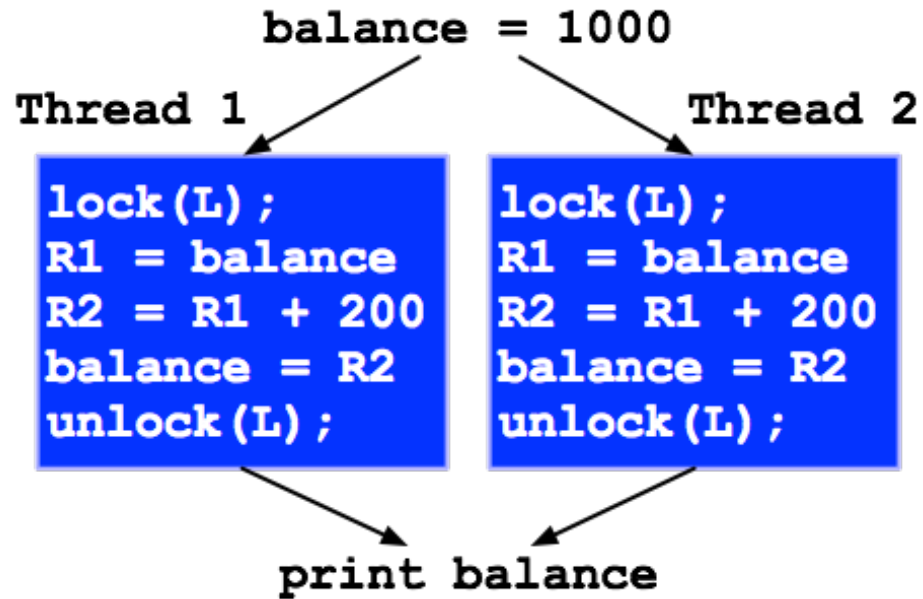


# Lock

- Atomicity를 보장하기 위해 사용
- 록 변수(lock variable)는 두 개의 상태(state)를 가짐
  - 록이 걸림(locked)과 록이 걸리지 않음(unlocked)
- 두 개의 함수 - lock(L)과 unlock(L)
  - 하나의 스레드가 록 변수 L에 록을 걸고자 할 때 먼저 록이 걸려있는지 확인
  - 록이 걸려 있으면 록을 건 스레드가 록을 풀어주기를 기다린 후 록을 건
- 스레드를 이용한 병렬 프로그램의 동기화에 이용



# Lock페 텅



# Spinlocks

- A thread requesting a lock does not release CPU but rather spins constantly checking the lock until it is released
  - The thread proceeds as soon as the lock is released
  - May save time for locks that are held for short time
    - No context switching
  - The thread wastes CPU cycles spinning
  - Cannot be used on uniprocessor systems
- Sleeplock
  - A thread requesting a lock blocks and is put back on the ready queue only when the lock is released
  - Can be used on uniprocessor
  - Saves CPU time on locks held long



# Spinlocks (cont'd)

- Test and set instruction
  - Takes a memory location
  - If the value in the memory location is false (0) then it sets the memory location to true (1) and returns false (0)
  - Otherwise, it returns true (1)
- Two while loops avoid performance degradation due to frequent memory accesses by test\_and\_set
  - test\_and\_set always go to memory

```
void lock(bool *L)
{
    while (test_and_set(L) {
        while (*L){};
    }
}

void unlock(bool *L)
{
    *L = 1;
}
```



# Pros and Cons of Locks

- Easy to understand and use
- However,
  - Deadlock
  - Priority inversion
  - Expensive

Thread 1	Thread 2
<code>lock (L) ;</code>	<code>lock (M) ;</code>
<code>lock (M) ;</code>	<code>lock (L) ;</code>
<code>...</code>	<code>...</code>
<code>unlock (M) ;</code>	<code>unlock (L) ;</code>
<code>unlock (L) ;</code>	<code>unlock (M) ;</code>





# Priority Inversion

- Only one processor
- Thread T1 has a higher priority than thread T2
- T1 is not ready
- T2 is scheduled and enters its critical section
- T1 becomes ready
- The scheduler suspends T2 and schedules T1
- When T1 tries to enter its critical section, it waits for T2 to release the lock (waits forever)
- Solutions exist
  - Disable preemption while holding a lock
  - Priority inheritance
  - ...



# Semaphores

- A semaphore,  $S$ , is a variable that can take only nonnegative integer values
- Manipulated by two special atomic operations, called P and V
  - $S$  is atomically incremented or decremented
  - P(S)
    - If  $S > 0$ , then P decrements  $S$  and returns
    - Otherwise, P suspends the calling thread until  $S$  becomes nonzero and the thread is restarted by a V operation performed by another thread
    - After restarting, P decrements  $S$  and returns
  - V(S)
    - V increments  $S$  by one
    - If there are any threads blocked in a P waiting for  $S$ , V restarts exactly one of these threads
- To schedule shared resources
- A thread suspended on a semaphore no longer executes instructions checking variables in a busy-wait loop



# Binary Semaphores vs. Counting Semaphores

- Binary semaphores
  - Its value is initially 1 and always 0 or 1
- Counting (general) semaphores
  - Its value can be any integer greater than or equal to 0
  - Represents a resource with many units available
    - The initial value is usually the number of resources



# 편안꺅꺅꺅 펍툐꺅 꺅꺅꺅꺅꺅

- Initially,  $S = 1$

```
void T0()  
{  
    while (1) {  
S00: non-critical-section;  
S01: P(S);  
S02: critical-section;  
S03: V(S);  
    }  
}
```

```
void T1()  
{  
    while (1) {  
S10: non-critical-section;  
S11: P(S);  
S12: critical-section;  
S13: V(S);  
    }  
}
```



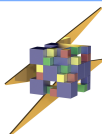
# An Atomic Counter with Semaphores

```
sem_t m;
int count;
void count_thread(int id)
{
    /* compute the portion of the array that this thread needs to work on */
    /* array_length: the size of the array */
    /* n: the number of threads */

    int private_count = 0;
    int length = array_length/n;
    int start = id*length;

    for (int i=start; i < start+length; i++){
        if (array[i] == 7)
        {
            private_count++;
        }
    }

    sem_wait(m); /* P(m) */
    count += private_count;
    sem_post(m); /* V(m) */
}
```



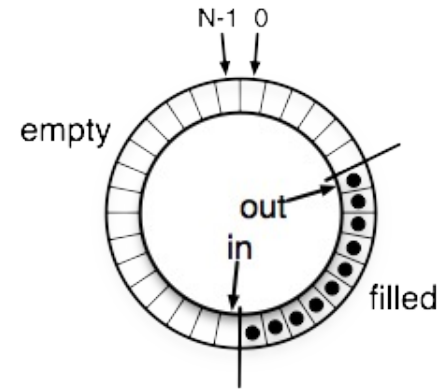
# Producers and Consumers

- There are two types of threads
  - Producers - these threads create a data element that is then sent to the consumers
  - Consumers - upon receipt of a data element, these threads perform some computation with the data element
- Assume a single producer and a single consumer



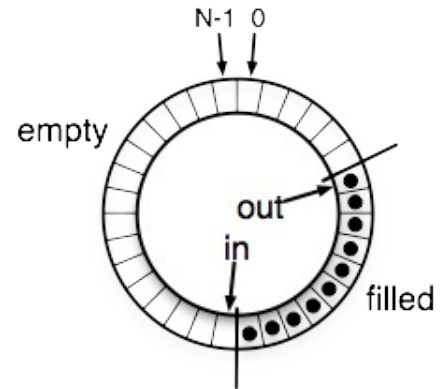
# Single-Producer and Single-Consumer

- Circular buffer
  - A circular queue
    - Implemented with an array
    - The index is computed modulo the length of the array
  - For asynchronous communication
  - The producer appends a data element to the tail of the queue
  - The consumer removes a data element from the head of the non-empty queue
- The consumer does not try to remove an element from an empty buffer
- The producer does not append to a full buffer



# Single-Producer and Single-Consumer (contd.)

- buffer is an array of length N
  - $in = 0$
  - $out = 0$
- elements and slots are general semaphores
  - $elements = 0$
  - $slots = N$



```
void producer() {  
    while(1) {  
        data = produce();  
        P(slots);  
        buffer(in) = data;  
        in = (in + 1) mod N;  
        V(elements);  
    }  
}
```

```
void consumer() {  
    while(1) {  
        P(elements);  
        data = buffer(out);  
        out = (out + 1) mod N;  
        V(slots);  
        consume(data);  
    }  
}
```





# Single-Producer and Single-Consumer (cont'd)

- buffer is an array of length N
  - $in = 0, out = 0, count = 0$
- S is a binary semaphore
  - $S = 1$
- not\_full and not\_empty are binary semaphores
  - $not\_full = 0, not\_empty = 0$

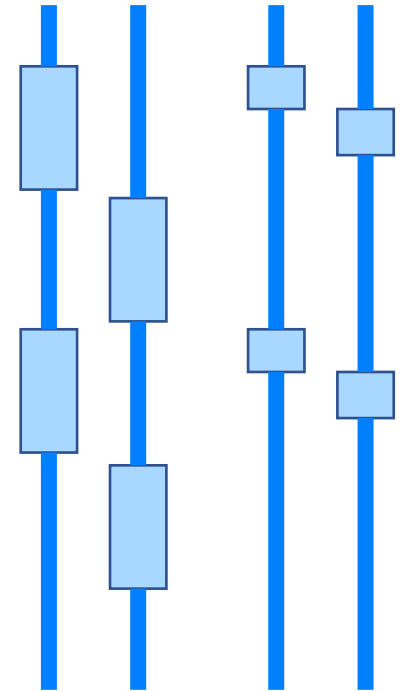
```
void producer() {
    int lcount = 0;
    while(1) {
        data = produce();
        if (lcount == N) P(not_full);
        buffer(in) = data;
        P(S);
        count = count + 1;
        lcount = count;
        V(S);
        if (lcount == 1) V(not_empty);
        in = (in + 1) mod N;
    }
}
```

```
void consumer() {
    int lcount = 0;
    while(1) {
        if (lcount == 0) P(not_empty);
        data = buffer(out);
        P(S);
        count = count - 1;
        lcount = count;
        V(S);
        if (lcount == N-1) V(not_full);
        out = (out + 1) mod N;
        consume(data);
    }
}
```



# Non-blocking Synchronization

- An algorithm is non-blocking if the suspension of one or more threads will not stop the potential progress of the other threads
- Blocking a thread is undesirable
  - When blocked, cannot accomplish anything
  - Coarse-grained locking
    - Reduces opportunities of parallelism
  - Fine-grained locking
    - Increases overhead
- Many non-blocking synchronization algorithms exist in these days
  - Hard to understand though



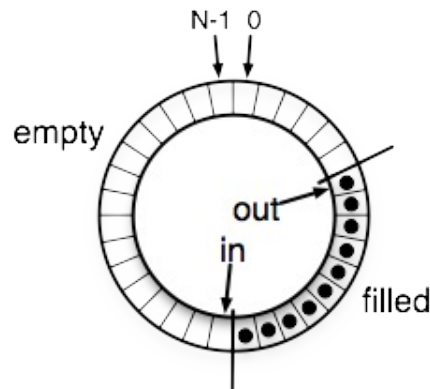
# Advantages of Non-blocking Synchronization

- No deadlocks
- No priority inversion
- Does not suffer performance degradation from
  - Context switching
  - Page faults
  - Cache misses



# Non-blocking Single-Producer and Single-Consumer

- Producer updates in
- Consumer updates out
- Without a non-blocking algorithm, either the the entire queue is locked or the shared variables in and out are locked between the producer and consumer to guarantee correctness
- Assume atomic assignments
- No synchronization for the shared variables in and out or the entire queue



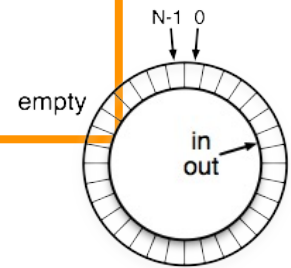
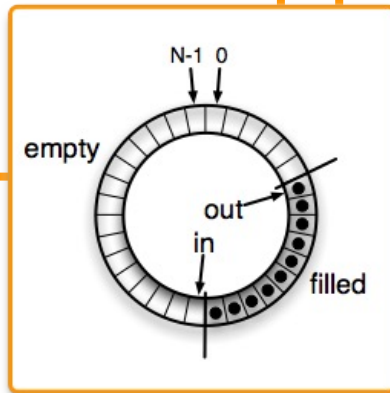
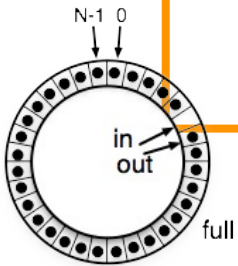
# Non-blocking Single-Producer and Single-Consumer (cont'd)

## Producer

```
while (...) {  
  while ( true ) {  
    Pin = in;  
    Pout = out;  
    if ((Pin + 1) mod N ≠ Pout)  
      /* full ? */  
      break;  
  }  
  ... /* produce */  
  Pin = ... ;  
  in = Pin;  
  ...  
}
```

## Consumer

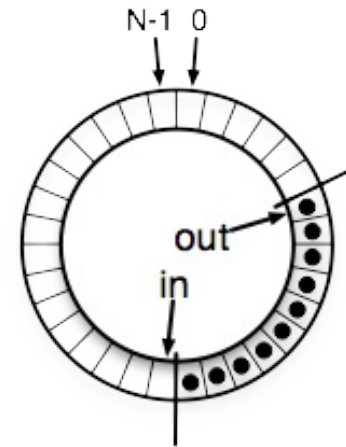
```
while (...) {  
  while ( true ) {  
    Cin = in;  
    Cout = out;  
    if (Cin ≠ Cout)  
      /* empty? */  
      break;  
  }  
  ... /* consume */  
  Cout = ... ;  
  out = Cout;  
  ...  
}
```



## Non-blocking Single-Producer and Single-Consumer (cont'd)

### Producer

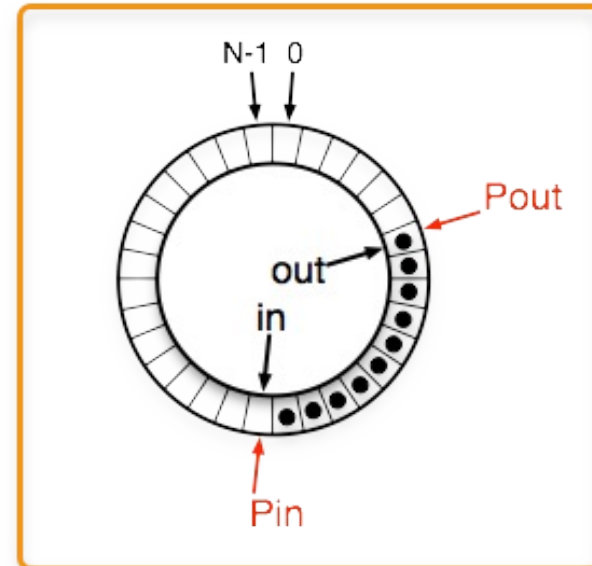
```
while (...) {  
    while ( true ) {  
        Pin = in;  
        Pout = out;  
        if ((Pin + 1) mod N ≠ Pout)  
            break;  
    }  
    ... /* produce */  
    Pin = ... ;  
    in = Pin;  
    ...  
}
```



## Non-blocking Single-Producer and Single-Consumer (cont'd)

### Producer

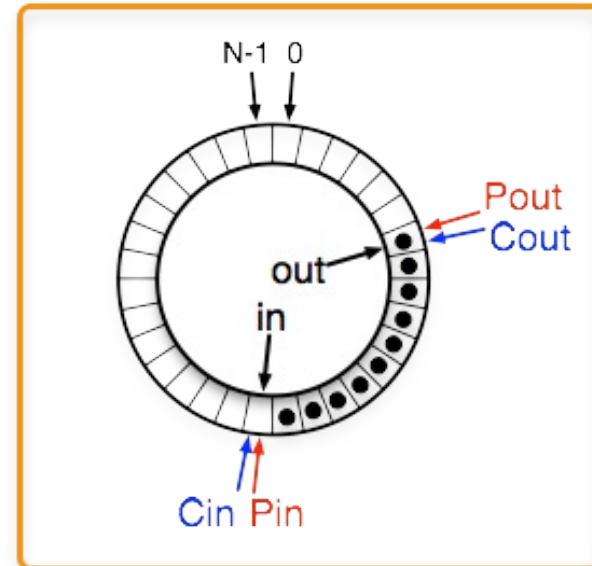
```
while (...) {  
    while ( true ) {  
        Pin = in;  
        Pout = out;  
        if ((Pin + 1) mod N ≠ Pout)  
            break;  
    }  
    ... /* produce */  
    Pin = ... ;  
    in = Pin;  
    ...  
}
```



## Non-blocking Single-Producer and Single-Consumer (cont'd)

### Consumer

```
while (...) {  
    while ( true ) {  
        Cin = in;  
        Cout = out;  
        if (Cin ≠ Cout)  
            break;  
    }  
    ... /* consume */  
    Cout = ... ;  
    out = Cout;  
    ...  
}
```

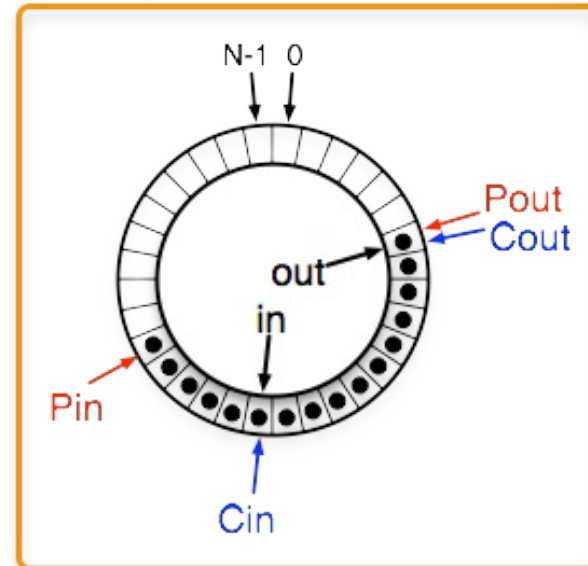




## Non-blocking Single-Producer and Single-Consumer (cont'd)

### Producer

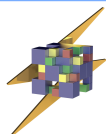
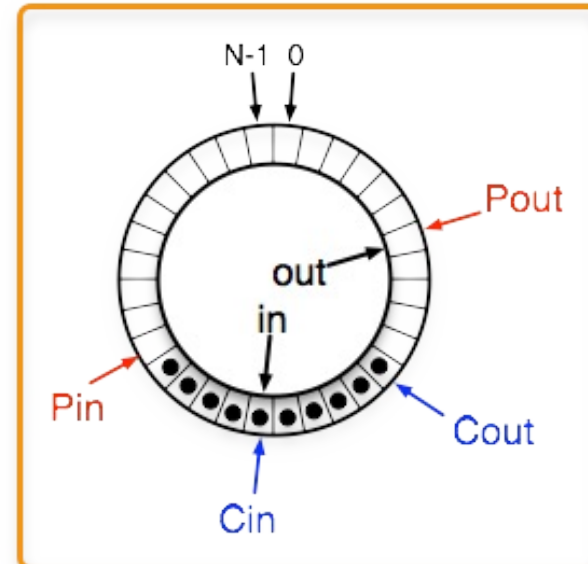
```
while (...) {  
  while ( true ) {  
    Pin = in;  
    Pout = out;  
    if ((Pin + 1) mod N ≠ Pout)  
      break;  
  }  
  ... /* produce */  
  Pin = ... ;  
  in = Pin;  
  ...  
}
```



## Non-blocking Single-Producer and Single-Consumer (cont'd)

### Consumer

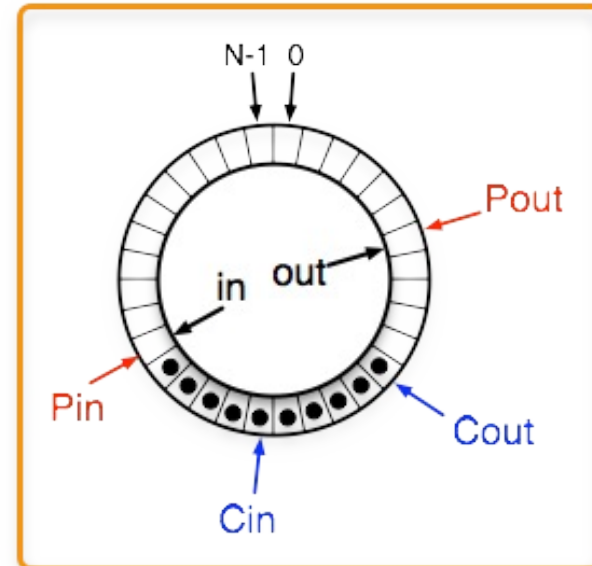
```
while (...) {  
  while ( true ) {  
    Cin = in;  
    Cout = out;  
    if (Cin ≠ Cout)  
      break;  
  }  
  ... /* consume */  
  Cout = ... ;  
  out = Cout;  
  ...  
}
```



## Non-blocking Single-Producer and Single-Consumer (cont'd)

### Producer

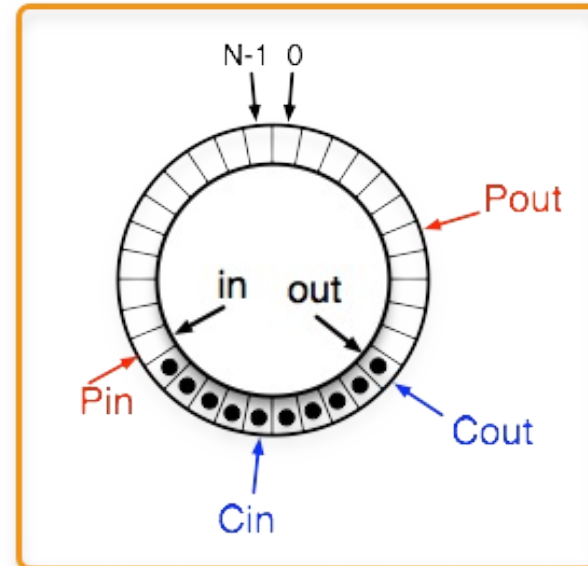
```
while (...) {  
  while ( true ) {  
    Pin = in;  
    Pout = out;  
    if ((Pin + 1) mod N ≠ Pout)  
      break;  
  }  
  ... /* produce */  
  Pin = ... ;  
  in = Pin;  
  ...  
}
```



## Non-blocking Single-Producer and Single-Consumer (cont'd)

### Consumer

```
while (...) {  
    while ( true ) {  
        Cin = in;  
        Cout = out;  
        if (Cin ≠ Cout)  
            break;  
    }  
    ... /* consume */  
    Cout = ... ;  
    out = Cout;  
    ...  
}
```



## 병렬화 시 고려 사항

- 프로그램에 든 병렬성의 양
  - Amdahl의 법칙
  - 순차실행을 할 수 밖에 없는 부분에 의해 Speedup이 결정됨
- 로컬리티
  - 될 수 있으면 로컬 데이터를 이용하여 계산하는 것이 좋음(캐시를 잘 이용하도록)
  - 스레드나 프로세스 간 데이터의 이동에 시간이 걸림



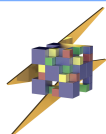
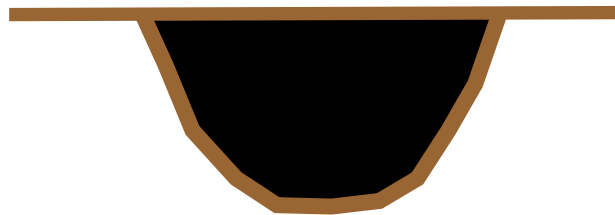
## 병렬화 시 고려 사항 (cont'd)

- Load balancing
  - 각 스레드 또는 프로세스가 같은 양의 일을 하도록 일을 분배하는 것이 중요
  - 전체 실행 시간은 가장 늦게 일을 끝내는 스레드나 프로세스에 의해 결정됨
- 병렬화 오버헤드
  - 스레드나 프로세스를 시작하는 비용
  - 통신 비용
  - 동기화 비용
  - 병렬화를 위한 가외의 계산



## 삽질하기

- 지름이 50cm, 깊이가 50cm 인 구덩이를 파는 경우



## 순차(sequential) 프로그래밍의 경우

- 한 사람이 삽질하는 경우





## 병렬(parallel) 프로그래밍의 경우

- 여러 사람이 삽질하는 경우



## 병렬(parallel) 프로그래밍의 경우 (cont'd)

- 지름이 3m, 깊이가 15m인 구덩이를 파는 경우



## 확장성

- Scalability
- 더 많은 프로세서가 있으면 성능이 향상됨을 일컬음
- 하지만 프로세서의 수가 증가할 수록 병렬화 오버헤드가 증가하기 때문에 달성하기 어려움
  
- Weak scalability vs. strong scalability

