Computer Science Study

Operating System

Subject 0. Review

Subject 1. Critical section

Subject 2. Sync instruction

Subject 3. Mutex, Semaphore

Subject 4. Monitor

Subject 5. Lock-Free algorithm

Subject 6. Deadlock

made by Seungtaek

홍승택

2024. 10. 28. 19시

subject 0.

Review

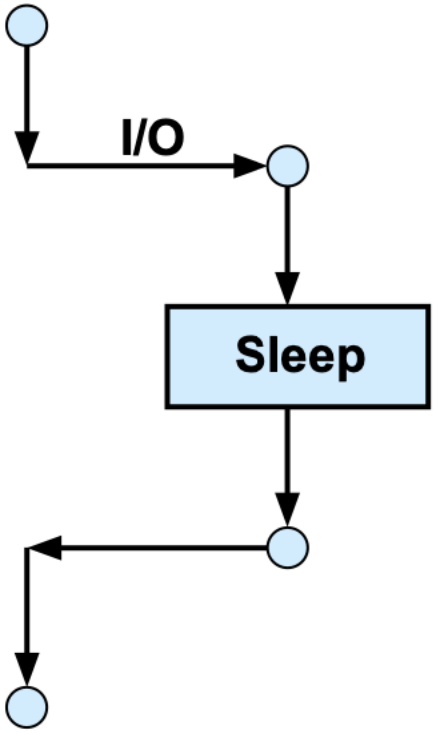
DMA (Direct Memory Access)

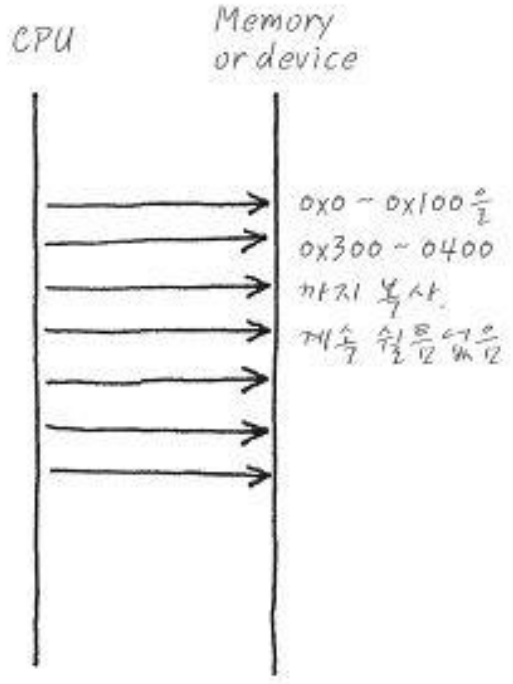
CPU를 장치의 상태 확인 및 데이터 이동에 사용하지 않고

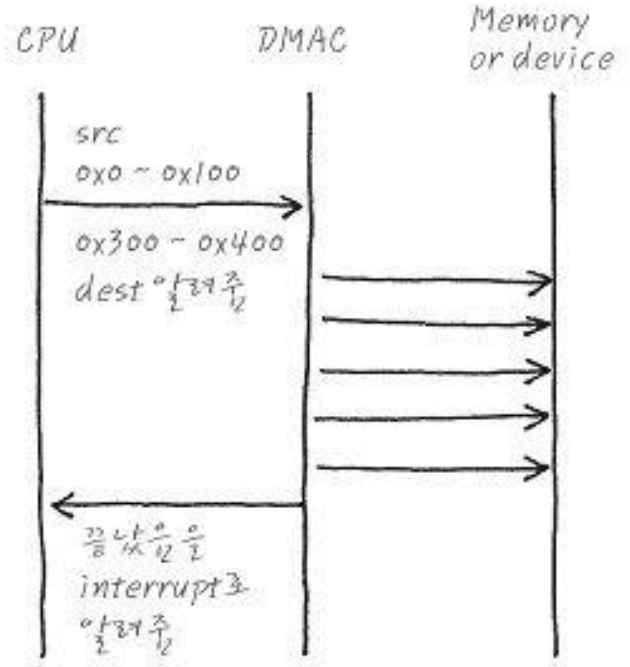
I/O를 위한 별도의 장치(DMA) 사용

DMA 를 사용하는 경우, I/O 를 호출한 프로세스는 **sleep** 함

멀티프로그래밍!





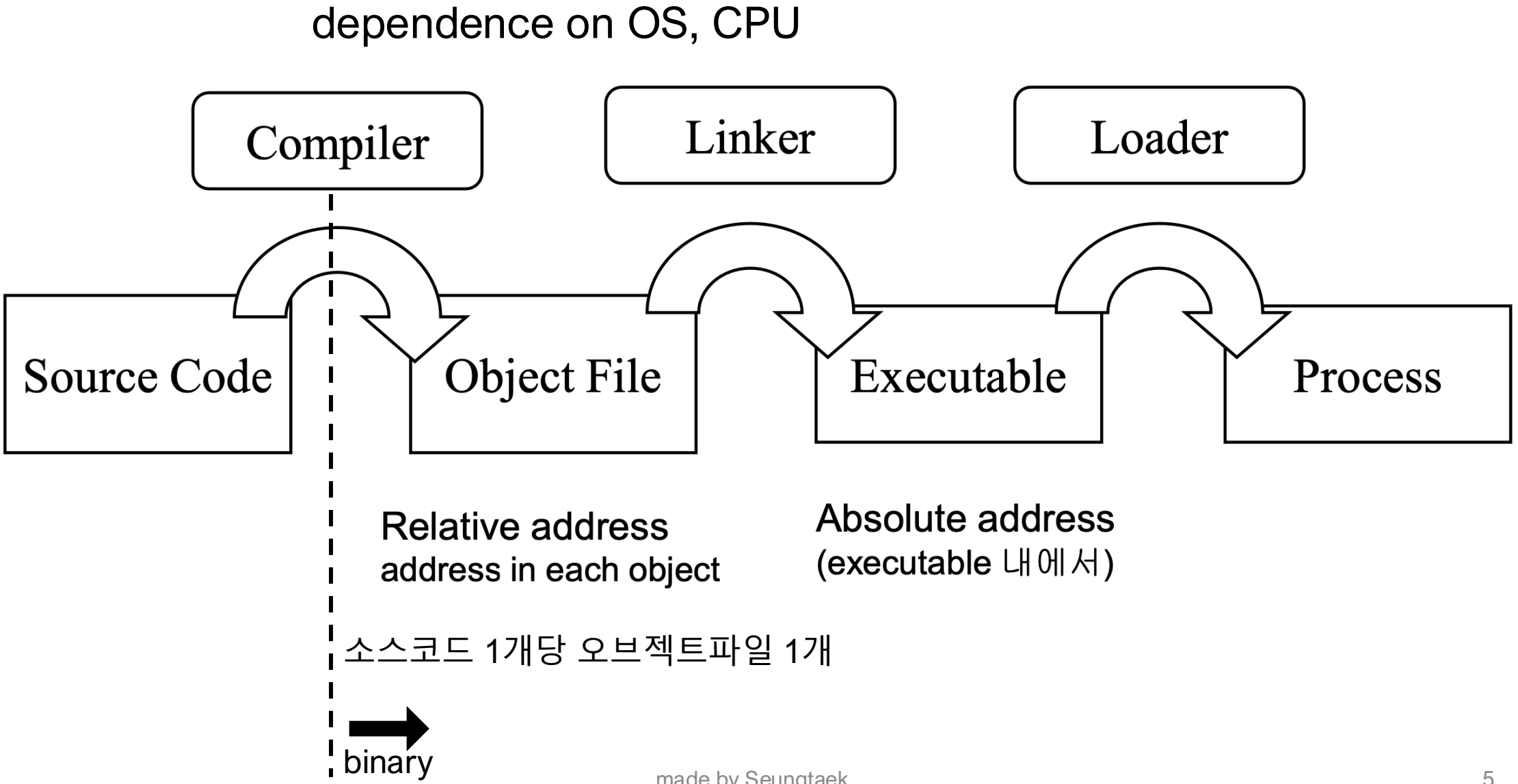


cycle stealing

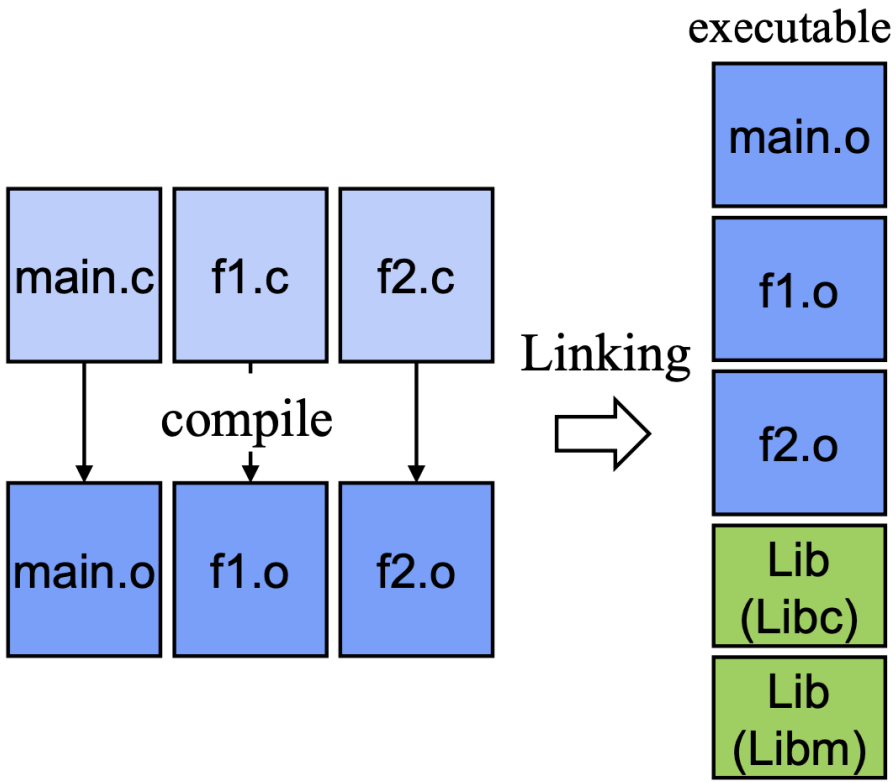
Interrupt handling philosophy in Kernel

- **Do as little as possible in the interrupt handler**
- **Defer non-critical actions till later**
- **Structure: top and bottom halves**
  - Top-half (Hardware Interrupt): do minimum work and return
    - ISR(Interrupt Service Routine)
  - Bottom-half (Software Interrupt): deferred processing
    - softirqs, tasklets, workqueues

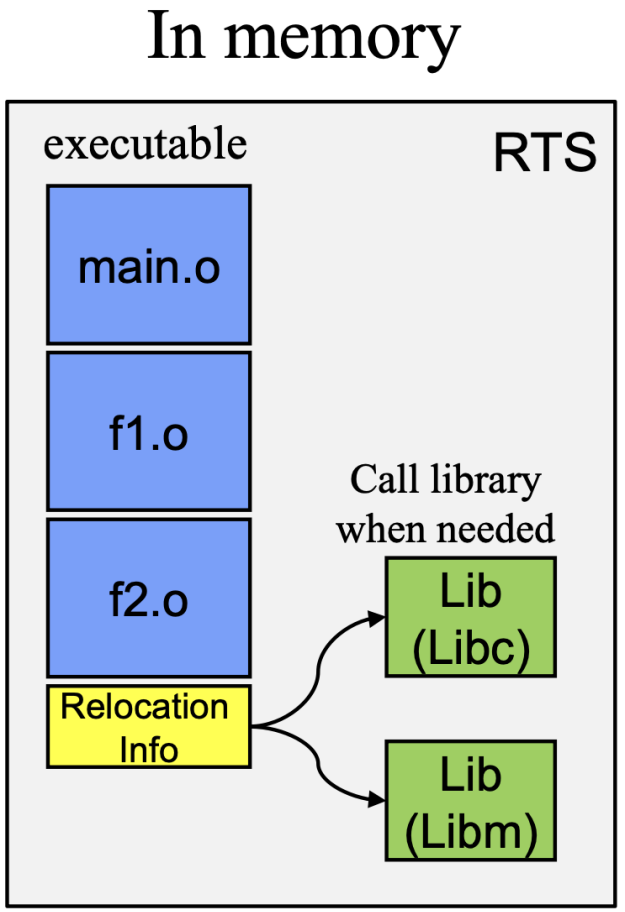
Compilation Process



Static Linking vs. Dynamic Linking

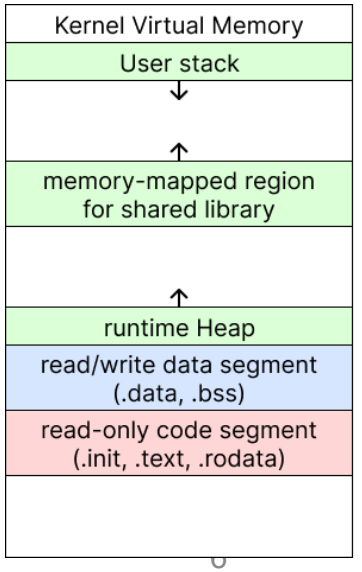


<Static Linking>
Big size executable



<Dynamic Linking>
ex) window: **.dll**
linux: **.so**

1. 로딩 시점
2. 런타임 시점

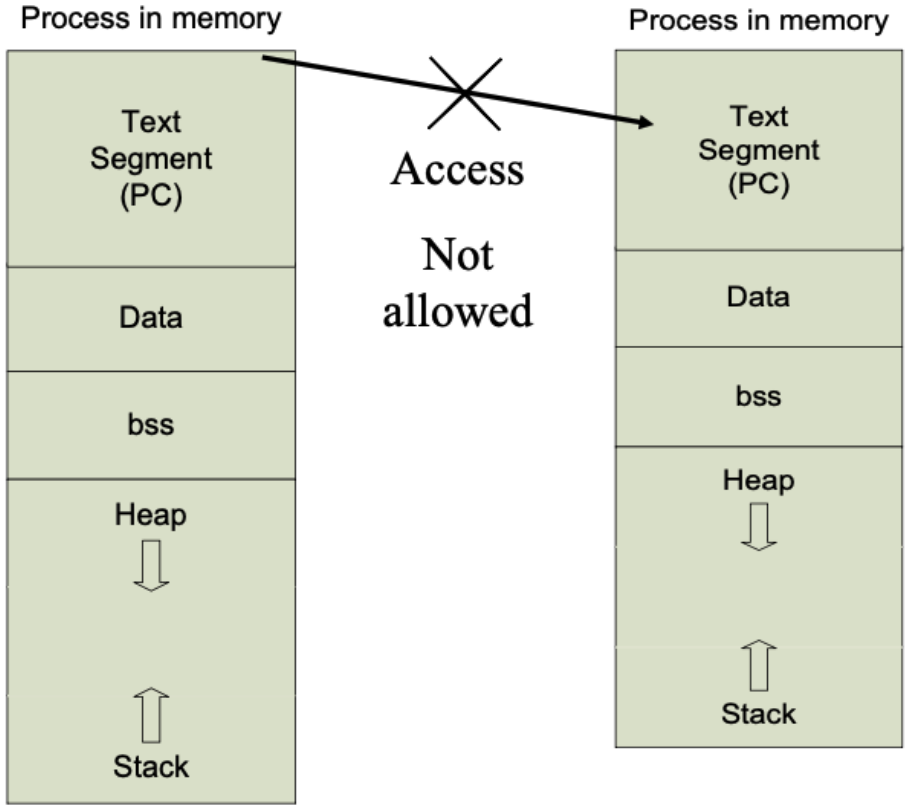


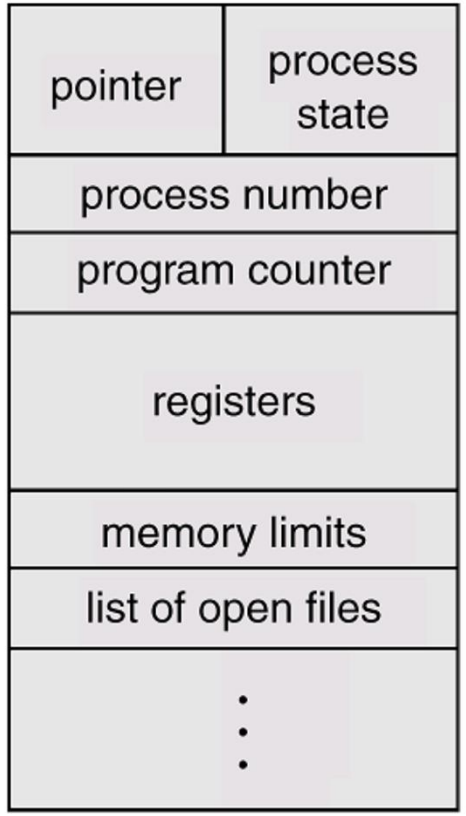
Process

실행 중인 프로그램

Process – abstraction for

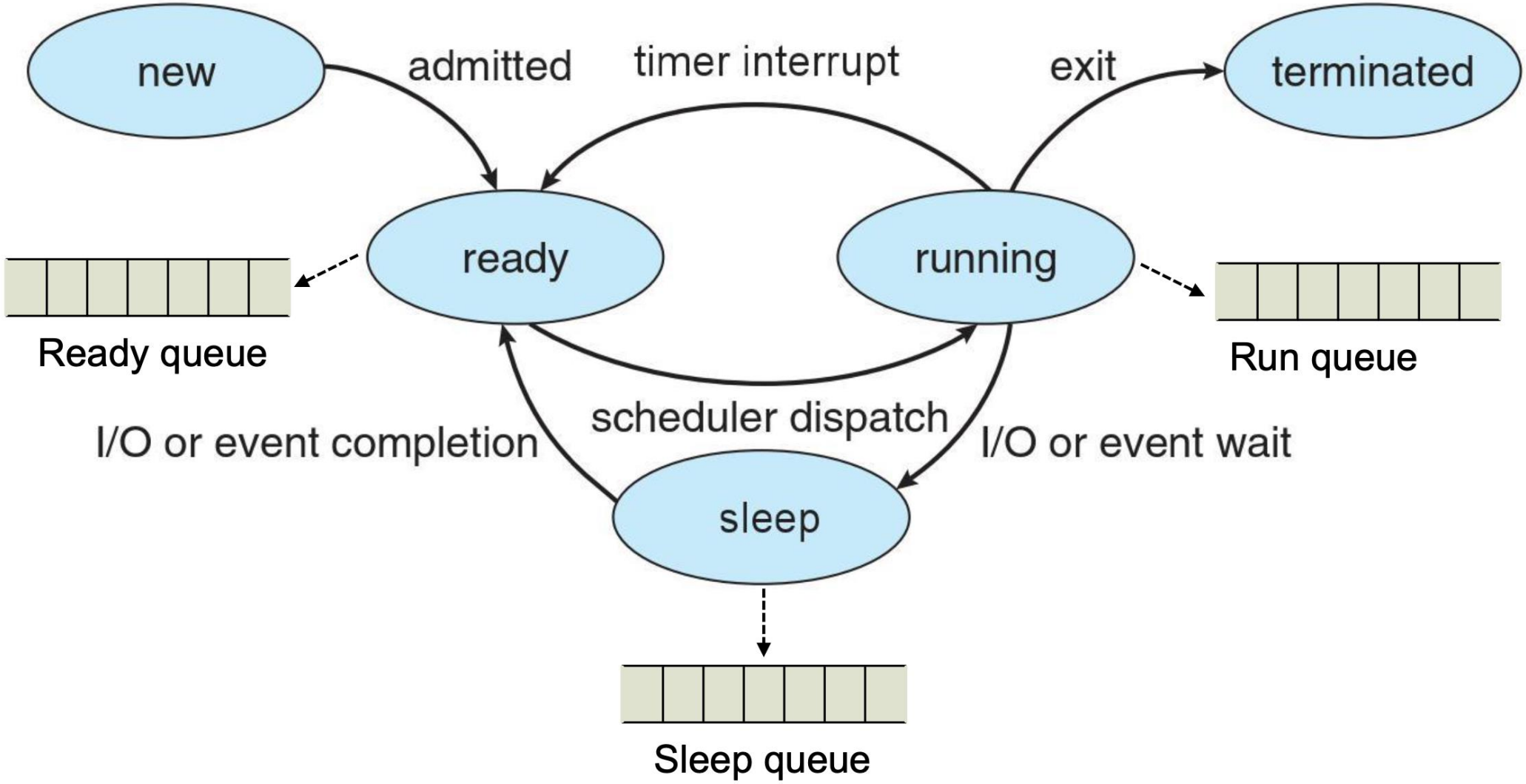
Protection domain: 서로 침범하지 못함





PCB(Process Control Block)

Process State



Interprocess Communication(IPC)

Shared memory

공유한 메모리 영역에 읽기/쓰기를 통해서 프로세스간 통신 수행

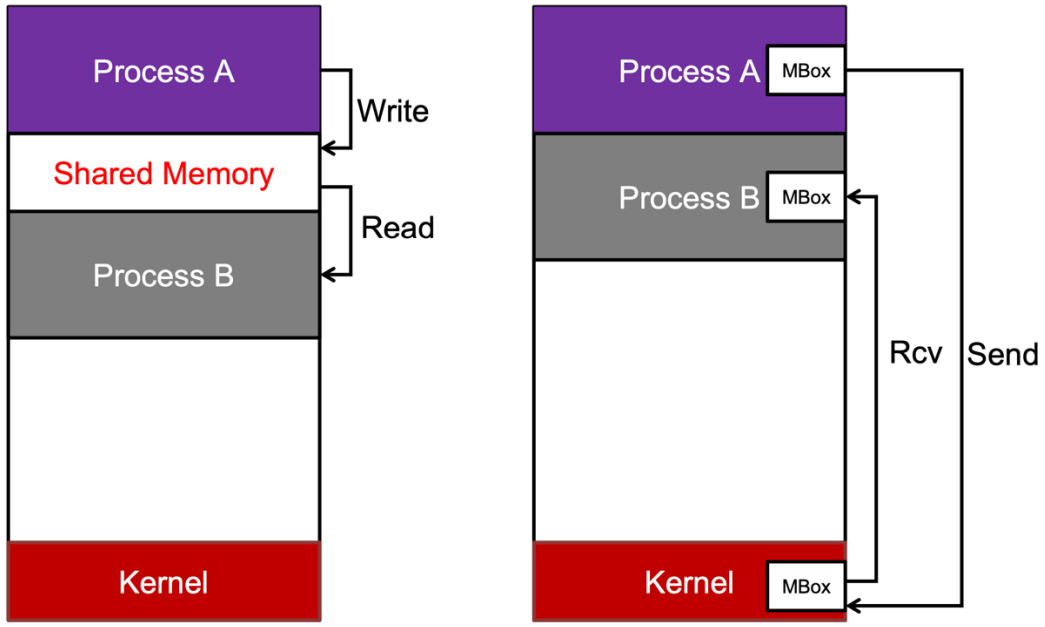
공유 메모리가 설정된 후부터는 **커널 관여 없음**

동기화!!

Message passing

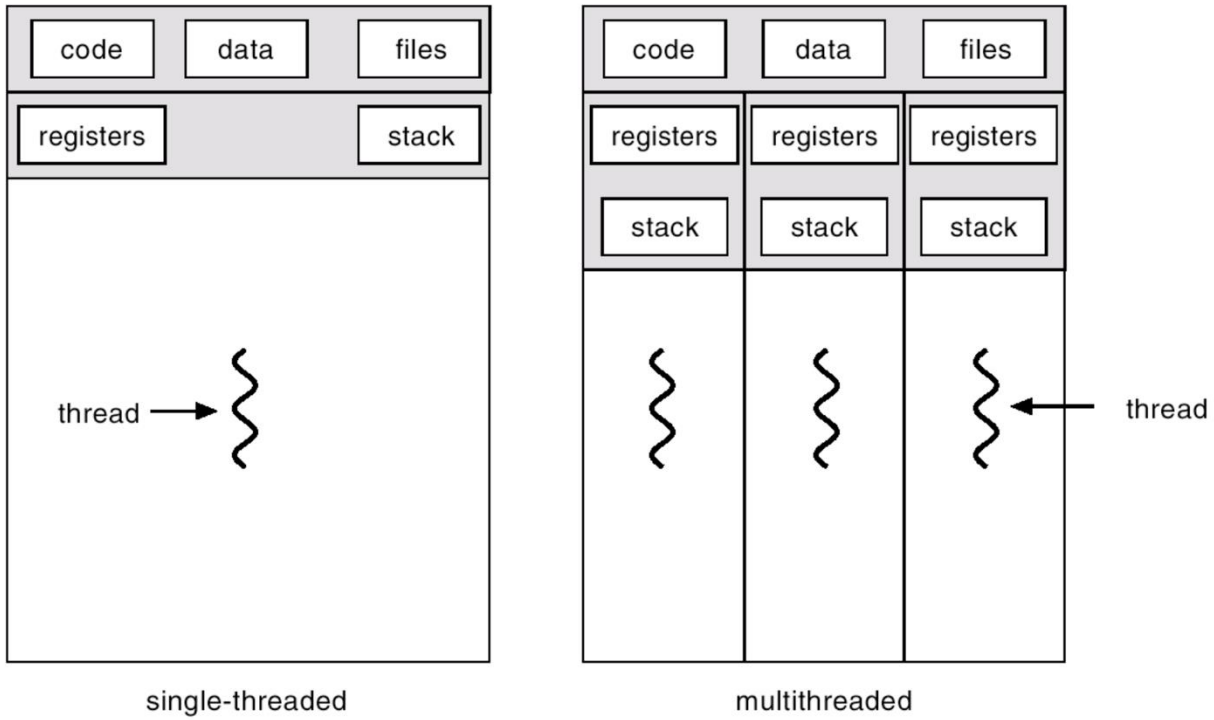
커널을 경유해서 메시지 전송

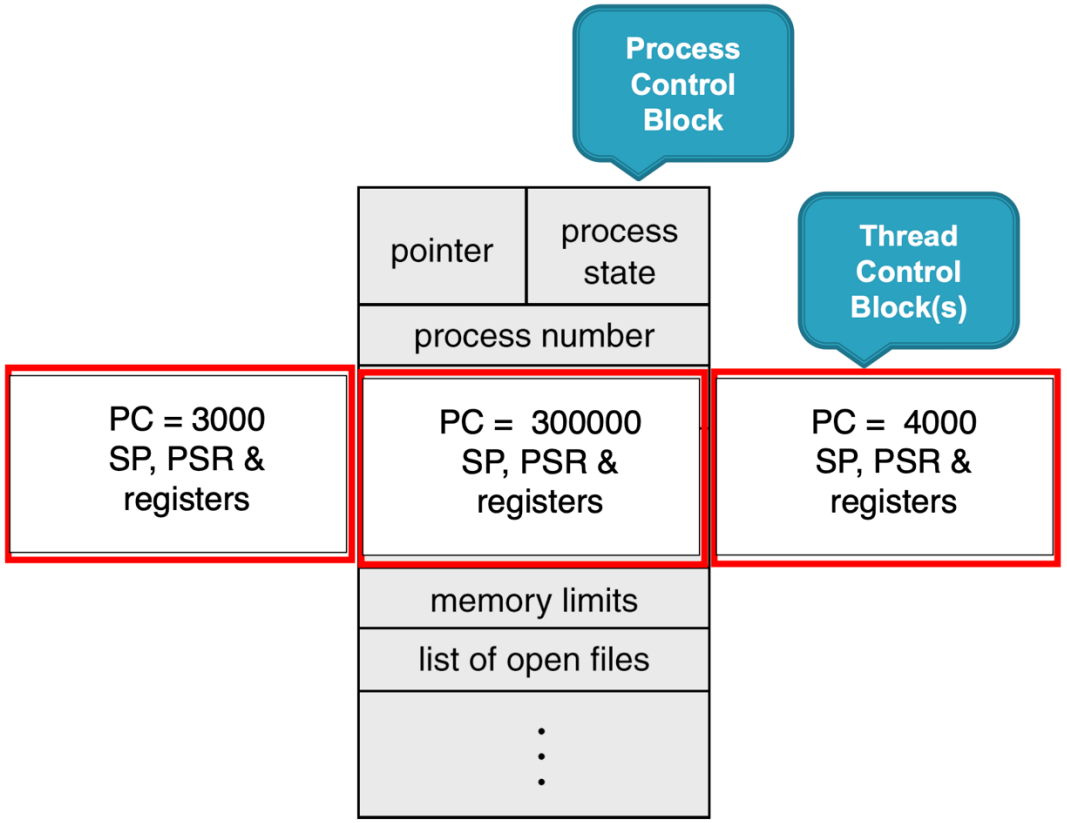
Pipe, Message Queue, Socket



공유 메모리 기법 메시지 교환 기법

Thread


code
data
files
registers
stack
thread
single-threaded
code
data
files
registers
registers
registers
stack
stack
stack
thread
multithreaded

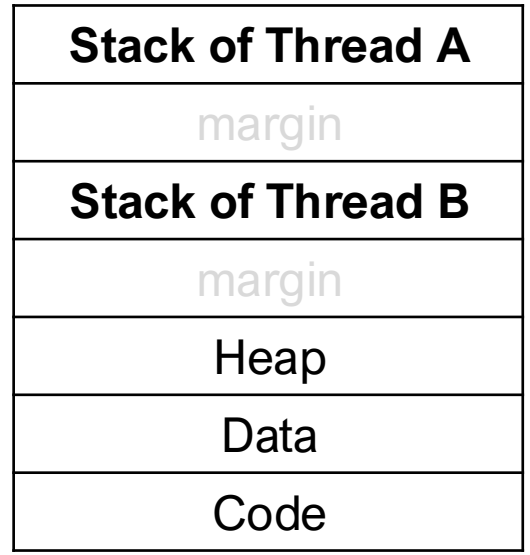

Process Control Block
pointer
process state
process number
memory limits
list of open files
Thread Control Block(s)
PC = 3000
SP, PSR &
registers
PC = 300000
SP, PSR &
registers
PC = 4000
SP, PSR &
registers

Thread

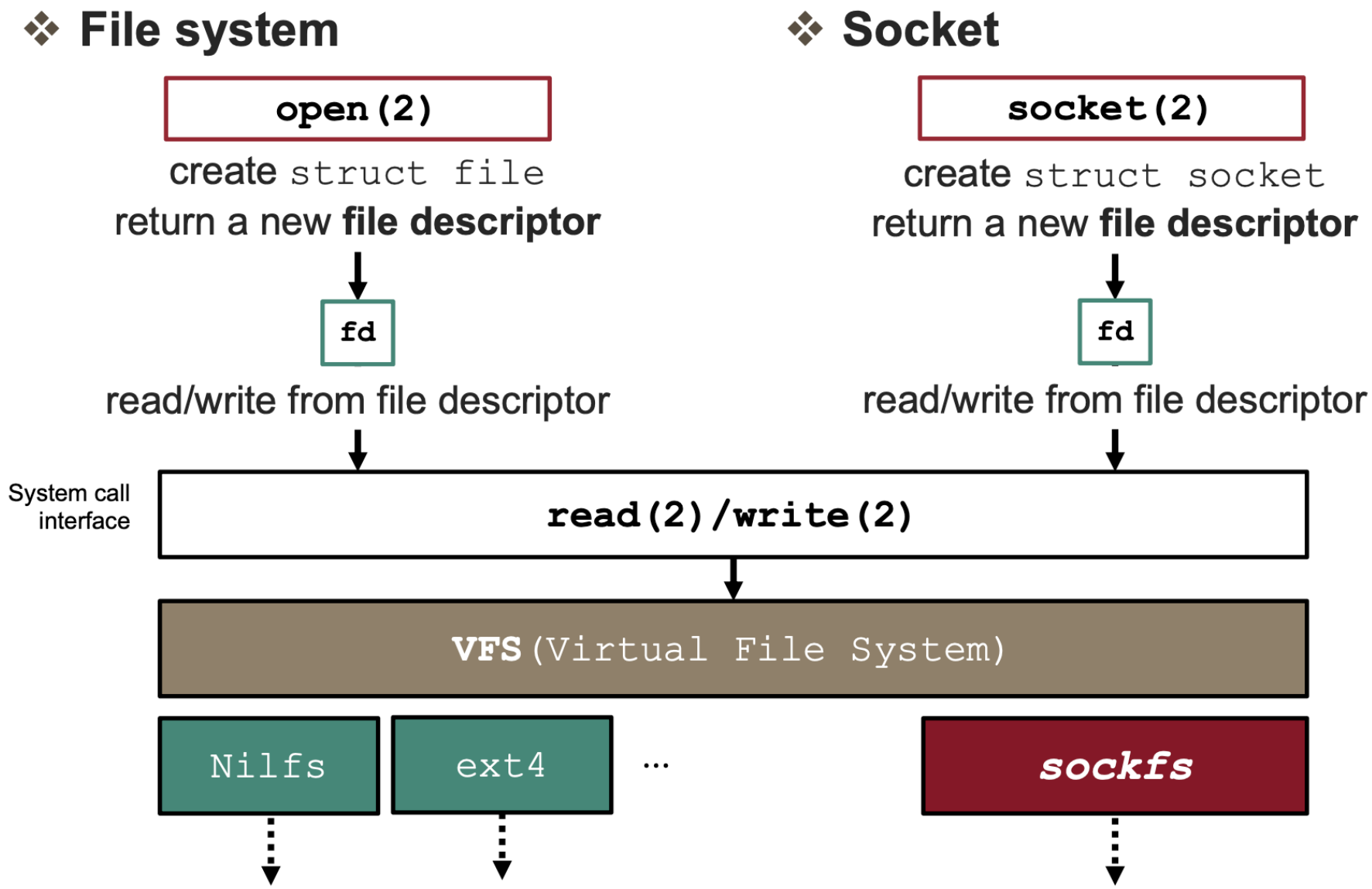
```
void foo(int *p) {
    cout << "Sub thread: " << this_thread::get_id() << "\n";
    *p = 2;
}

int main() {
    cout << "Main thread: " << this_thread::get_id() << "\n";

    int a = 1;
    thread t(foo, &a);
    t.join();

    cout << a;
}
```

```
Main thread: 0x1e75ab240
Sub thread: 0x16bc87000
2
```

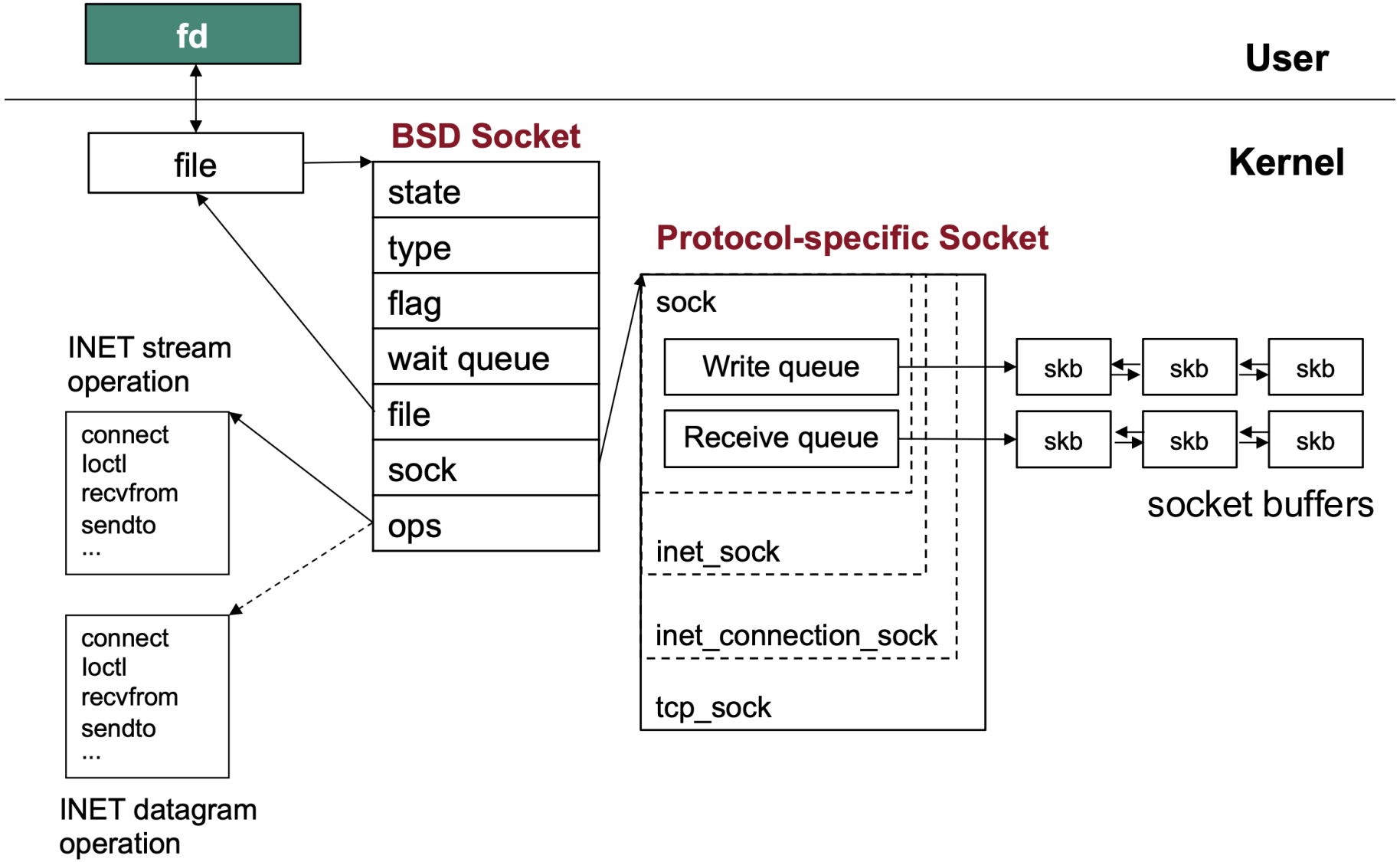
Not protected stack section of each threads



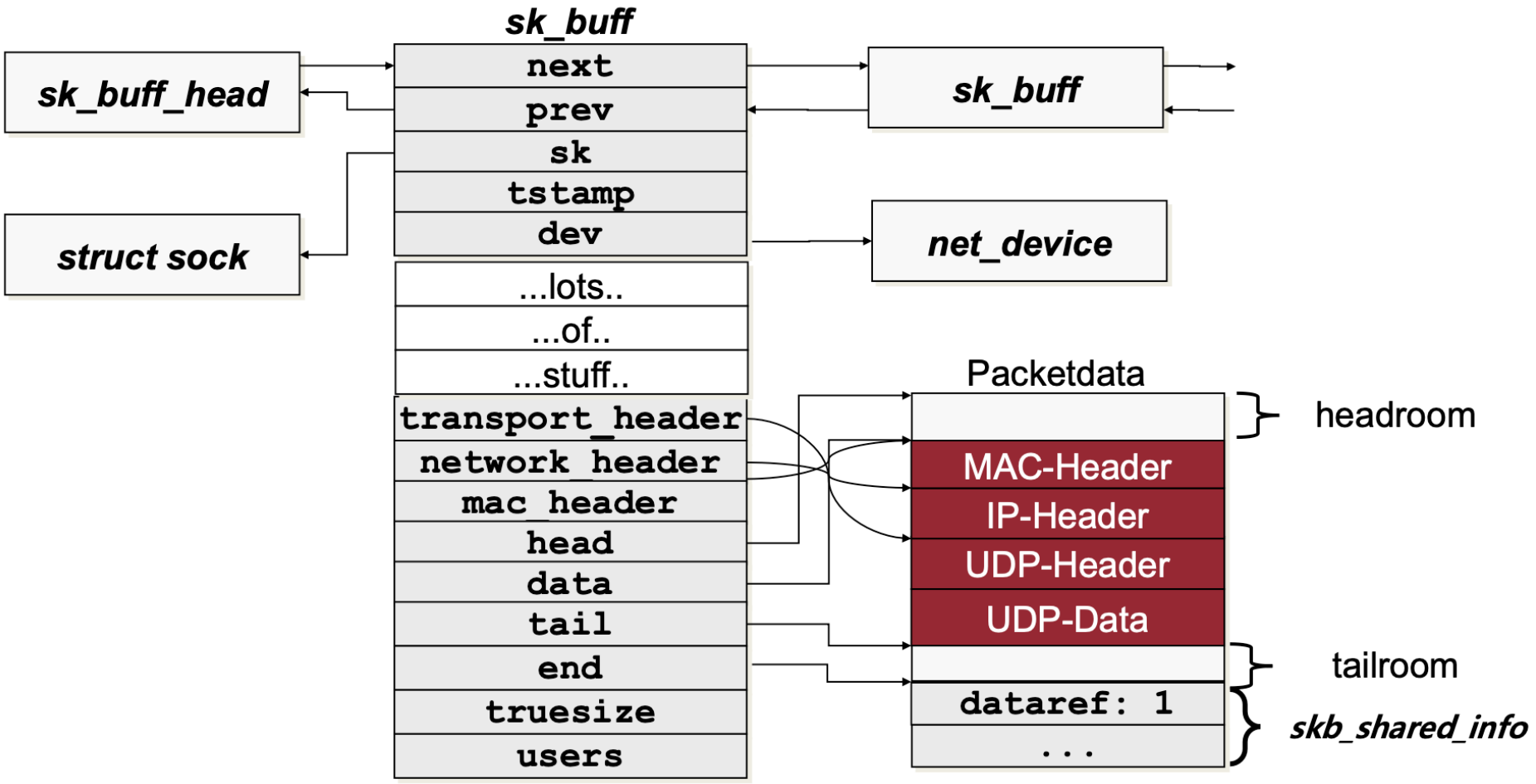
Socket as a File



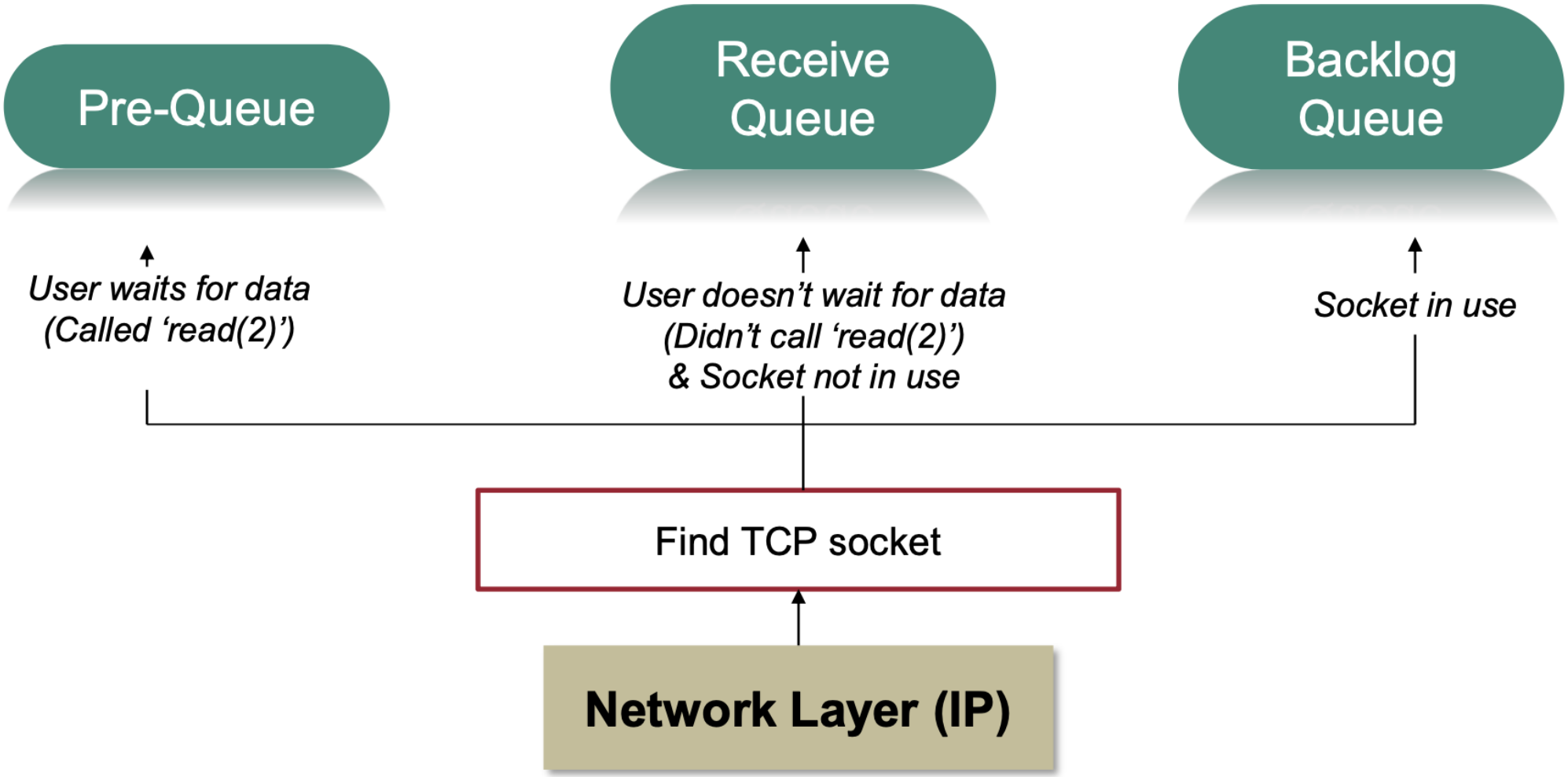
Overall structure



Structure of sk_buff

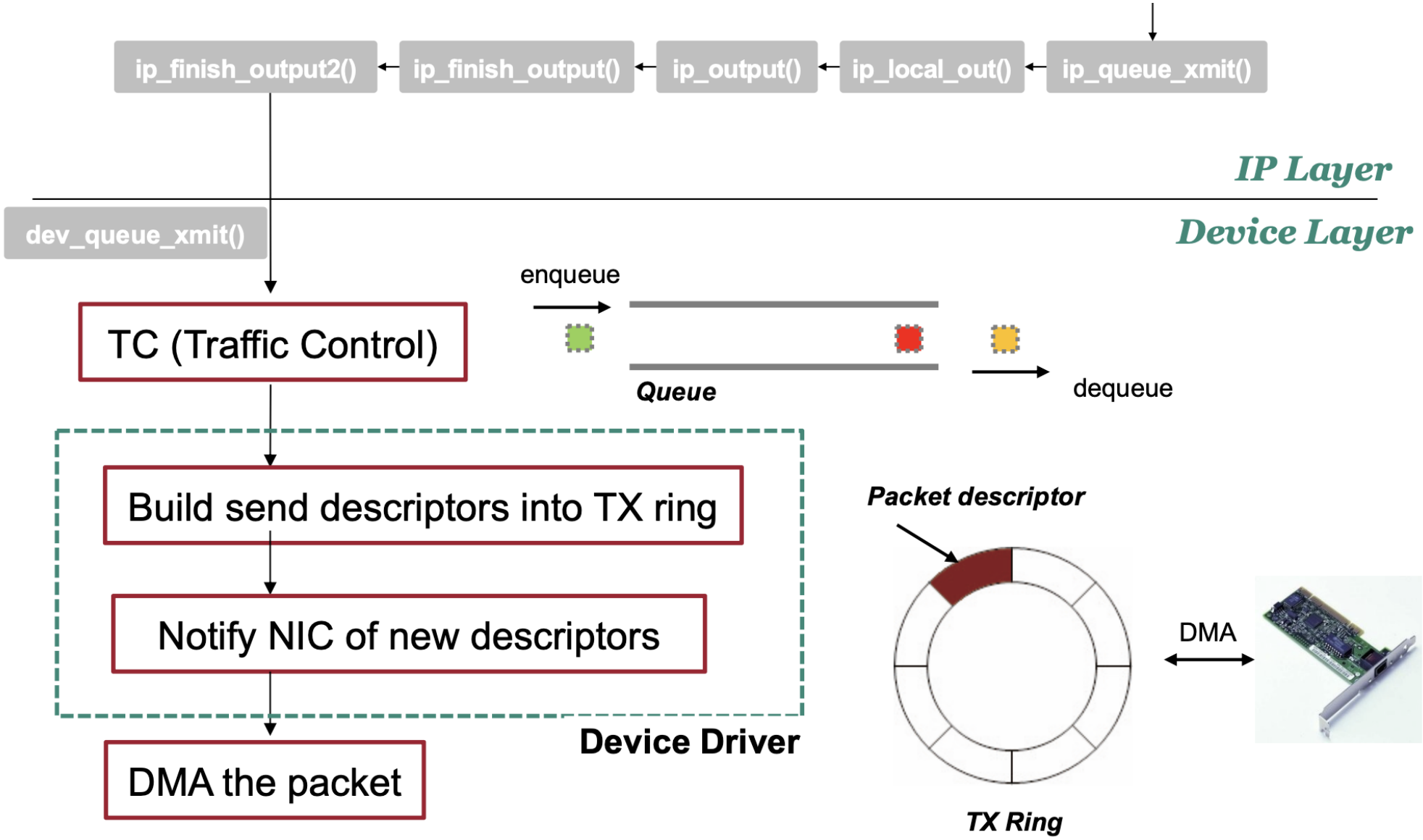


Receiving segments in TCP

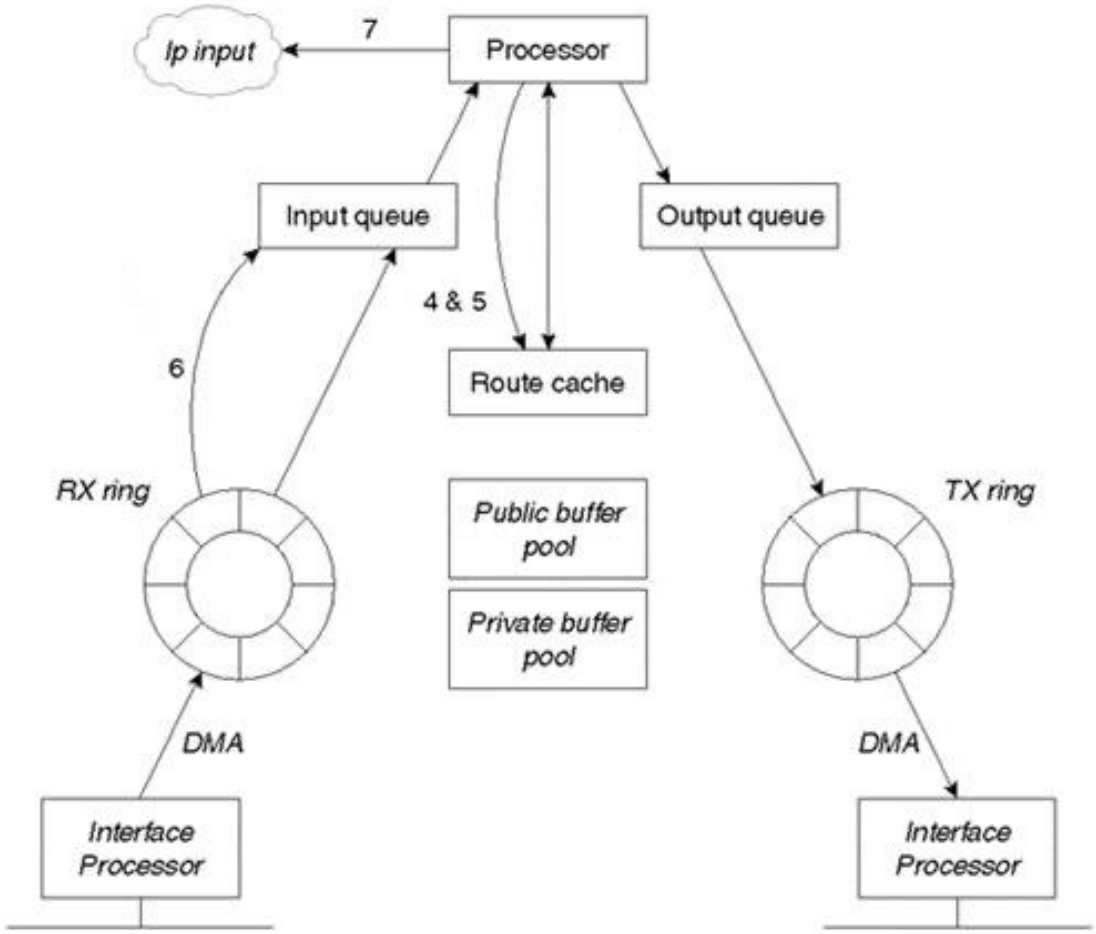


+ Out-of-order queue
: 패킷 순서가 순서대로 도착하지 않았을

Sending data frames (data link layer)



TX / RX Ring




```
root@CT100:~# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.0.39  netmask 255.255.255.0  broadcast 192.168.0.255
        inet6 fe80::be24:11ff:fed0:bd8a  prefixlen 64  scopeid 0x20<link>
        ether bc:24:11:d0:bd:8a  txqueuelen 1000  (Ethernet)
        RX packets 718994  bytes 316807709 (316.8 MB)
        RX errors 0  dropped 14  overruns 0  frame 0
        TX packets 442852  bytes 58145609 (58.1 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

subject 1.

Critical section

Race condition

init. Balance = 0

< Process A : 입금 처리>

Balance = Balance + 500;

< Process B : 출금 처리>

Balance = Balance - 500;

```
< Process A >
Register1 = Balance
Register1 = Register1 + 500
Balance = Register1
```

```
< Process B >
Register2 = Balance
Register2 = Register2 - 500
Balance = Register2
```

Case 1

```
Register1 = Balance
Register1 = Register1 + 500
Balance = Register1
Register2 = Balance
Register2 = Register2 - 500
Balance = Register2
```

최종 Balance = 0

Case 2

```
Register1 = Balance
Register2 = Balance
Register2 = Register2 - 500
Register1 = Register1 + 500
Balance = Register2
Balance = Register1
```

최종 Balance = 500

Case 3

```
Register1 = Balance
Register2 = Balance
Register2 = Register2 - 500
Register1 = Register1 + 500
Balance = Register1
Balance = Register2
```

최종 Balance = -500

결과를 예측할 수 없다.

Critical section

- Atomic 실행을 위하여 각 프로세스 혹은 쓰레드가 공유변수, 자료구조, 파일 등을 **배타적으로 읽고 쓸 수 있도록 설정한 코드 세그먼트**
- **동기화(Synchronization)**: 코드 상에 임계 구역을 설정하여 진입/진출을 순서화(Serialize) 하는 것

```
entry section
    critical section
exit section
    remainder section
```

Critical section 해결을 위한 조건

Mutual Exclusion

- 한 프로세스가 critical section에 진입해 있다면, 다른 모든 프로세스는 진입할 수 없어야 함

Progress

- critical section에 진입하려는 프로세스가 존재한다면, 그 중에서 한 프로세스는 유한 시간 내에 critical section 에 진입할 수 있어야 함

Bounded Waiting

- 어떤 프로세스가 Critical section에 진입할 때까지 걸리는 시간에 “limit”이 존재하여야 함

Solution 1

Shared variables:

```
int turn = 0; // 초기화
              // turn = 0 일 때,  P0 가 critical section에 진입 가능
```

Process P_0

```
while (turn != 0) ; //waiting
        critical section
turn = 1;
        remainder section
```

Process P_1

```
while (turn != 1) ; //waiting
        critical section
turn = 0;
        remainder section
```

만족 조건 : Mutual exclusion

불만족 조건 : Progress, Bounded waiting

- 두 프로세스의 수행 순서가 alternate 안되면 진행 안됨 (P0, P0, P0 이면 두 번째 P0부터 멈춤)
- 물론 P0, P1, P0 처럼 번갈아가며실행되면 알맞은 알고리즘이다.

Solution 2

Shared variables:

```
boolean flag[2]; // flag [0] = flag [1] = false 로 초기화
                 // flag [0] = true일 때, P0 이 critical section에 진입
                 // flag [1] = true일 때, P1 이 critical section에 진입
```

Process P0

```
flag[0] = true;
while (flag[1]) ; //waiting
        critical section
flag[0] = false;
        remainder section
```

Process P1

```
flag[1] = true;
while (flag[0]) ; // waiting
        critical section
flag[1] = false;
        remainder section
```

만족 조건 : Mutual exclusion

불만족 조건 : Progress, Bounded waiting

- 두 프로세스가 동시에 flag[] 를 true 로 할 수 있음

Peterson algorithm

Shared variables

```
int turn;
boolean flag[2]; // flag[0] = flag[1] = false 로 초기화
```

Process P_0

```
flag [0] = true;
turn = 1;
while (flag [1] && (turn == 1)) ;
        critical section
flag [0] = false;
        remainder section
```

Process P_1

```
flag [1] = true;
turn = 0;
while (flag [0] && (turn == 0)) ;
        critical section
flag [1] = false;
        remainder section
```

만족 조건 : Mutual exclusion, Progress, Bounded waiting

- turn은 두 프로세스가 동시에 접근해도 상관없다. (마지막에 write 한 사람이 늦게 들어감)

Peterson algorithm

- When working at the hardware level,
  Peterson's algorithm is typically not needed to achieve atomic access
- Most modern CPUs reorder memory accesses to improve execution efficiency

Shared variables
boolean flag = false;
int x = 0;

Process P0

```
while(!flag);
print x;
```

Process P1

```
x = 100;
flag = true;
```

print x == 100 or 0 ?

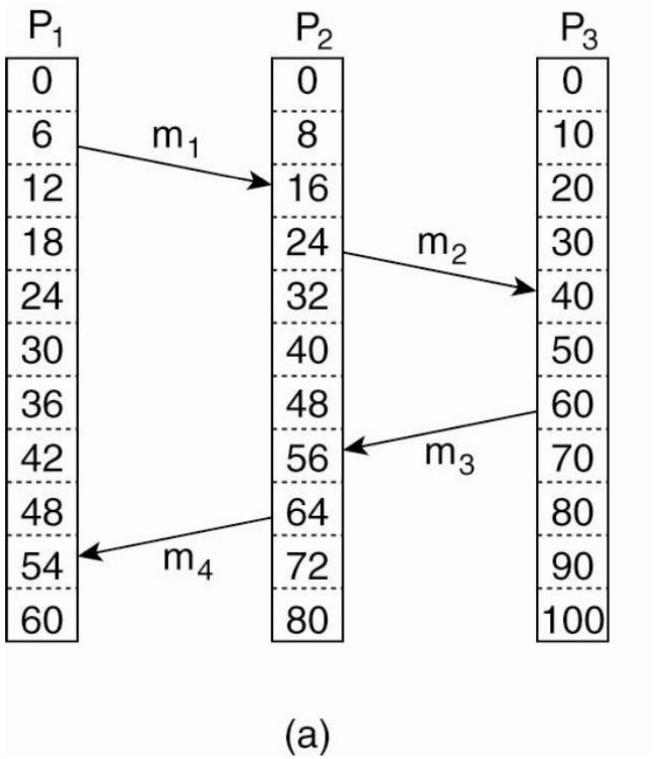
Bakery algorithm (Lamport)

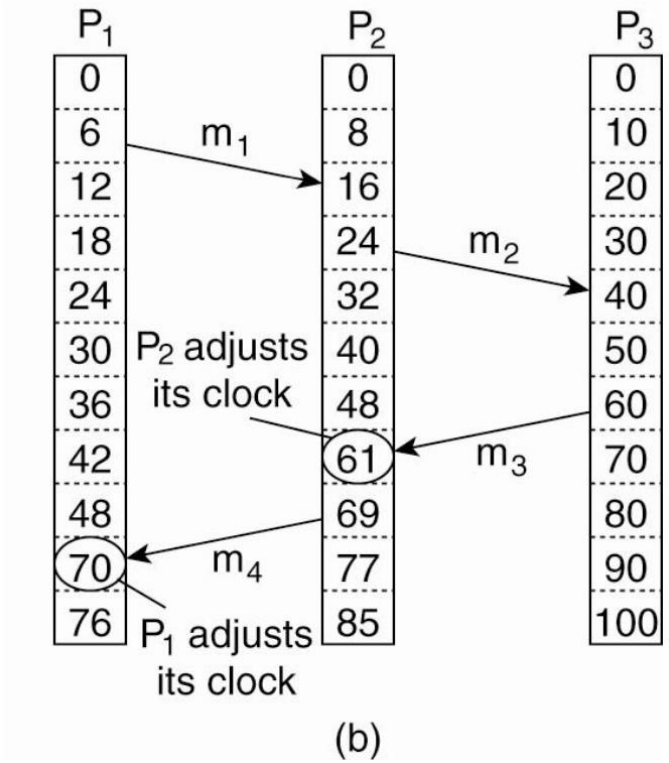
- 대기하는 각 프로세스는 번호표를 하나씩 뽑는다.
- 각 프로세스는 **자신의 번호가 가장 작은 번호가 될 때까지 기다린다.**
- 만약 발급 받은 숫자가 같을 경우 PID가 작은 것을 먼저 처리

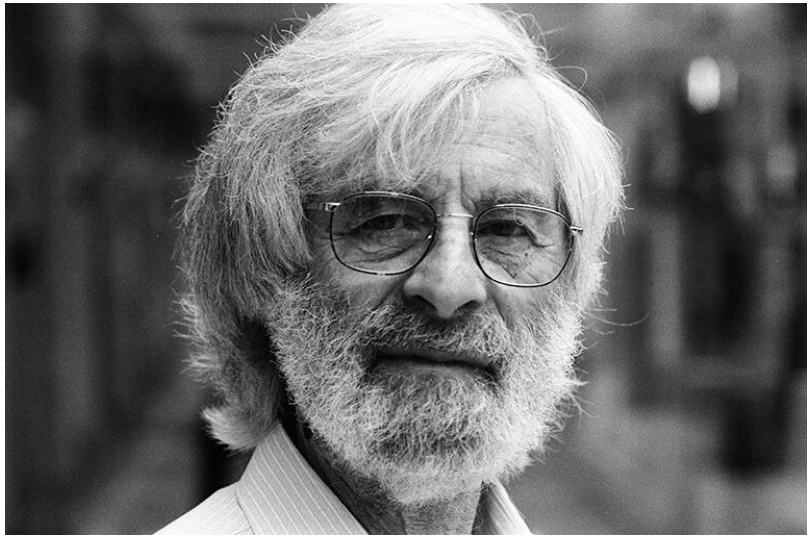
```
// 번호표 뽑기
choosing[i] = true;
number[i] = max(number[0], number[1], ..., number[n-1]) + 1;
choosing[i] = false;

// entry section
for (j = 0; j < n, i!=j; j++) {
        while (choosing[j]); // Pj가 번호를 뽑고 있는 동안은 기다려 줌
        // (number[j] < number[i]) || (number[j] == number[i] && j < i)
        while (number[j] != 0 && (number[j], j) < (number[i], i));
}
// ...임계 구역...
number[i] = 0; // exit section
```

Leslie Lamport

Logical clocks







1941 ~ (U.S.)

- LaTex
- Bakery algorithm
- Logical clocks
- **2013 Turing Award**

Happens-before relation, Concurrent event

subject 2.

Synchronization instruction

Hardware solution

- CPU에서 원자적(atomically)으로 수행되는 명령어를 (instruction) 이용
  - 명령어가 수행되는 동안 인터럽트 발생 못함 (non-interruptible)

x86

CMPXCHG (Compare and exchange)
If accumulator (AL/AX/EAX/RAX) compares equal to first operand, then EFLAGS.ZF is set to 1 and the first operand is overwritten with the second operand. Otherwise, EFLAGS.ZF is set to 0, and first operand is copied into the accumulator.

XADD (Exchange and add)
Exchanges the first operand with the second operand, then stores the sum of the two values into the destination operand.

Test and Set

```
boolean TestAndSet(boolean *target) {
        boolean rv = *target;
        *target = true;
        return rv;
}
```

```
boolean lock = false; // Shared data

// Process Pi
do {
        while (TestAndSet(&lock)); // if return 1, loop
        // do critical section
        lock = false;
}
```

Compare And Swap (CAS)

```
void compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;
    if(*value == expected) *value = new_value;
    return temp;
}
```

```
// lock : 공유변수
while(compare_and_swap(&lock, 0, 1));
// critical section..
lock = 0;
```

Swap

```
void Swap(boolean *a, boolean *b) {
        boolean temp = *a;
        *a = *b;
        *b = temp;
}
```

```
// lock: 공유 변수, key: 지역변수

key = 1;
do {
     Swap(&lock, &key);
} while (key == 1);
// ...critical section...
lock = 0;
```

Intel 80x86 instruction set

- BTS - Bit Test and Set (386+)
- BSWAP - Byte Swap (486+)

subject 3.

Mutex, Semaphore

Mutex

Wikipedia

In computer science, a **lock** or **mutex** (from mutual exclusion) is a **synchronization primitive** that prevents state from being modified or accessed by multiple threads of execution at once.

Locks enforce mutual exclusion concurrency control policies,

and with a **variety of possible methods**

there exist multiple unique implementations for different applications.

Mutex

Wikipedia

Language support [edit]

See also: Barrier (computer science)

Programming languages vary in their support for synchronization:

- Ada provides protected objects that have visible protected subprograms or entries[7] as well as rendezvous.[8]
- The ISO/IEC C standard provides a standard mutual exclusion (locks) API since C11. The current ISO/IEC C++ standard supports threading facilities since C++11. The OpenMP standard is supported by some compilers, and allows critical sections to be specified using pragmas. The POSIX pthread API provides lock support.[9] Visual C++ provides the `synchronize` attribute of methods to be synchronized, but this is specific to COM objects in the Windows architecture and Visual C++ compiler.[10] C and C++ can easily access any native operating system locking features.
- C# provides the `lock` keyword on a thread to ensure its exclusive access to a resource.
- VB.NET provides a `SyncLock` keyword like C#'s `lock` keyword.
- Java provides the keyword `synchronized` to lock code blocks, methods or objects[11] and libraries featuring concurrency-safe data structures.
- Objective-C provides the keyword `@synchronized`[12] to put locks on blocks of code and also provides the classes NSLock,[13] NSRecursiveLock,[14] and NSConditionLock[15] along with the NSLocking protocol[16] for locking as well.
- PHP provides a file-based locking [17] as well as a `Mutex` class in the `pthreads` extension. [18]
- Python provides a low-level mutex mechanism with a `Lock` class from the `threading` module.[19]

Mutex using POSIX API

```
void *run(void *mutex) {
    ...
    pthread_mutex_lock(&mutex);       /* locks the database   */
    ...                               /* handles the request  */
    pthread_mutex_unlock(&mutex);     /* unlocks the database */
}

int main() {
    pthread_mutex_t mutex;
    pthread_t thread_id[10];

    pthread_mutex_init(&mutex, NULL);  /* creates the mutex      */

    for (i = 0; i < 10; i++)           /* loop to create threads */
        pthread_create(&thread_id[i], NULL, rtn, &mutex);

    //pthread_join(thread_id[i], NULL) /* waits end of session   */

    pthread_mutex_destroy(&mutex);     /* destroys the mutex     */
}
```

Implement Mutex …

Peterson's Solution

Bakery Algorithm(Lamport)

Test-And-Set

Compare-And-Swap

https://www.ibm.com/docs/ko/aix/7.3?topic=programming-using-mutexes

Semaphore

Wikipedia

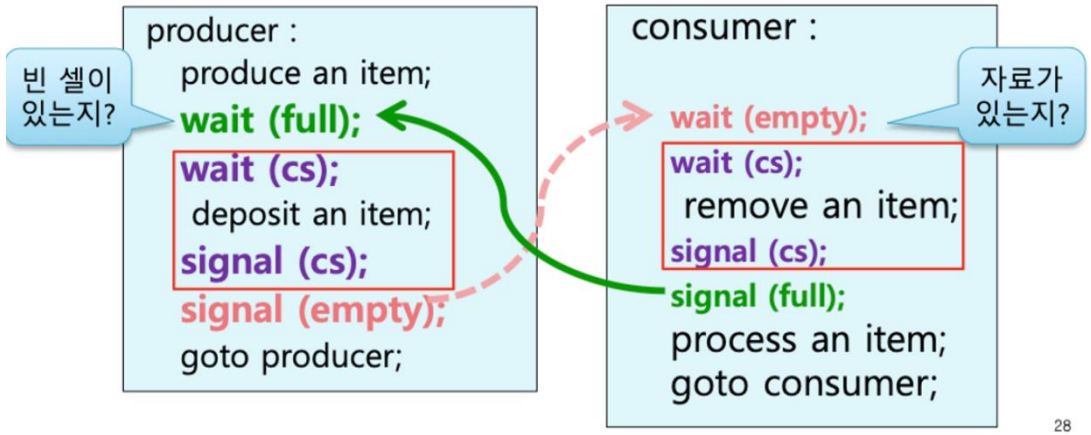
In computer science, a semaphore is a variable or abstract data type used to control access to a **common resource by multiple threads** and **avoid critical section problems** in a concurrent system such as a multitasking operating system.

Semaphores are a type of synchronization primitive.

Semaphore

```
typedef struct {
    int value = n;
    struct process *list;
} semaphore;

wait(semaphore *S) { // P operation
    S -> value--;
    if (S -> value < 0) {
        (S -> list).push(this.thread);
        block();
    }
}

signal(semaphore *S) { // V operation
    S -> value++;
    if (S -> value <= 0) {
        process* P = (S -> list).pop();
        wakeup(P);
    }
}
```

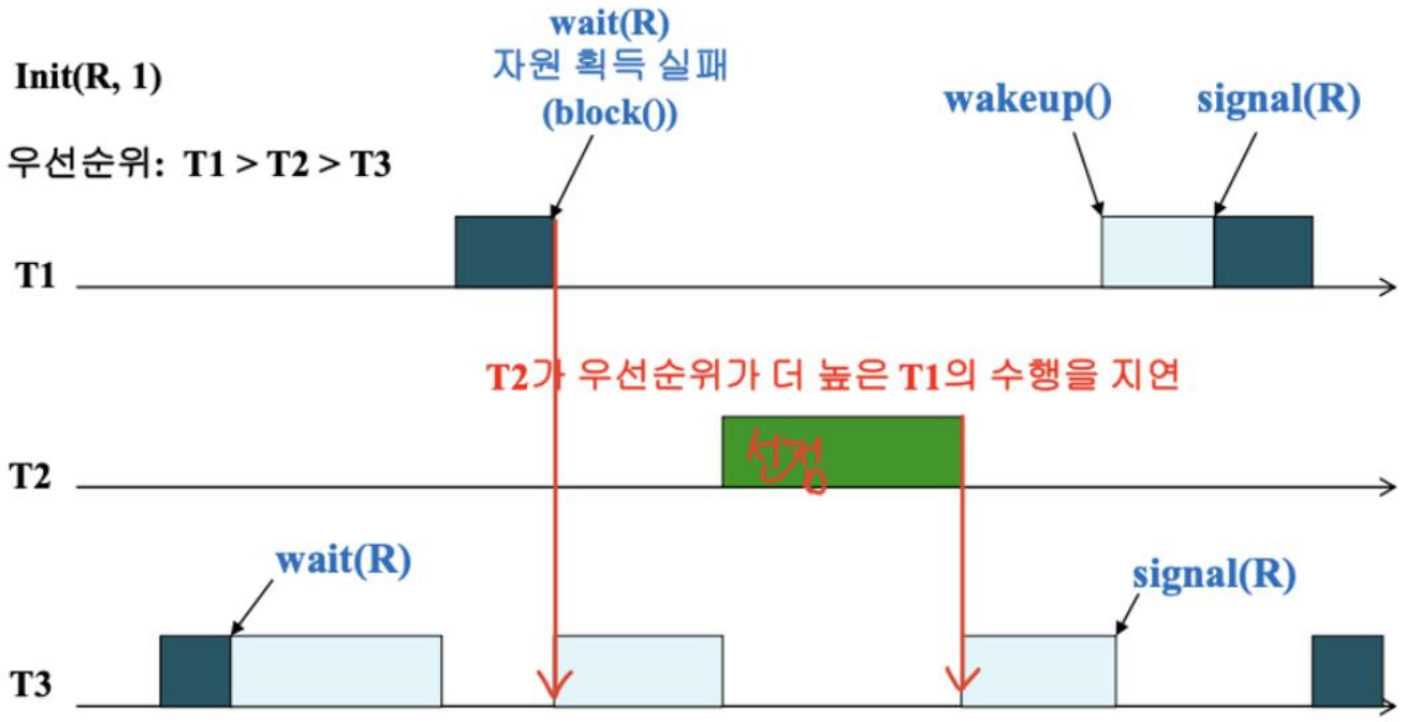
n > 1 : multiple resource
n = 1 : mutex (binary semaphore)
n = 0 : serialization



Mutex vs. binary Semaphore

1. **Priority inversion**
2. Premature task termination
3. Termination deadlock
4. Recursion deadlock
5. Accidental release

Priority Inversion Problem



Simple solution: 우선순위 상속 (T3에게 T1의 우선순위를 임시 부여)

Semaphore의 단점

- P와 V의 연산이 분리 되어 있기 때문에,
  (의도적으로 또는 실수로) 잘못 사용할 경우에 대한 대책이 없음

- V를 호출하지 않은 경우 : **Deadlock**
- P를 호출하지 않은 경우 : **Not mutual exclusion**

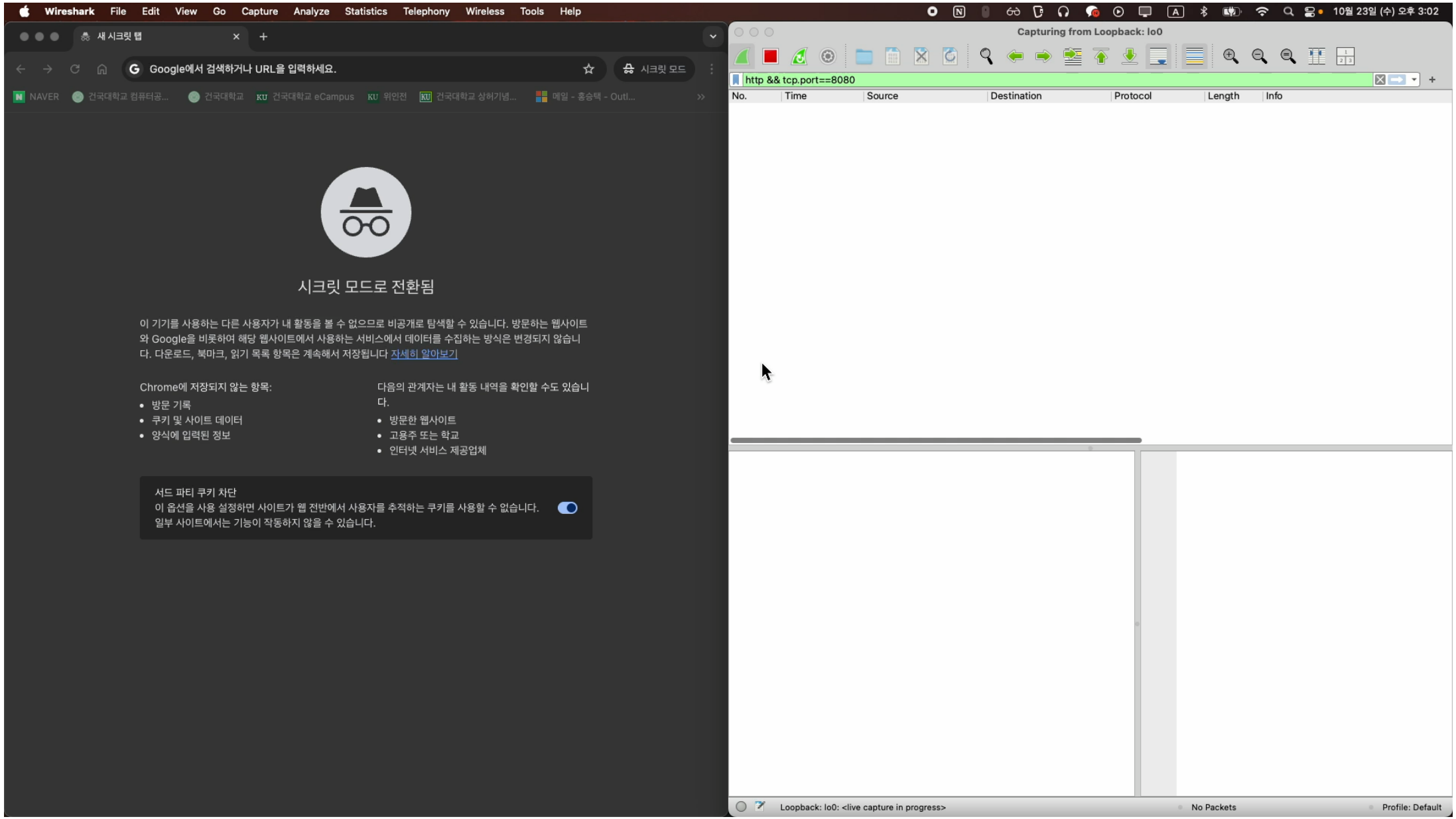
CS

Common Sense

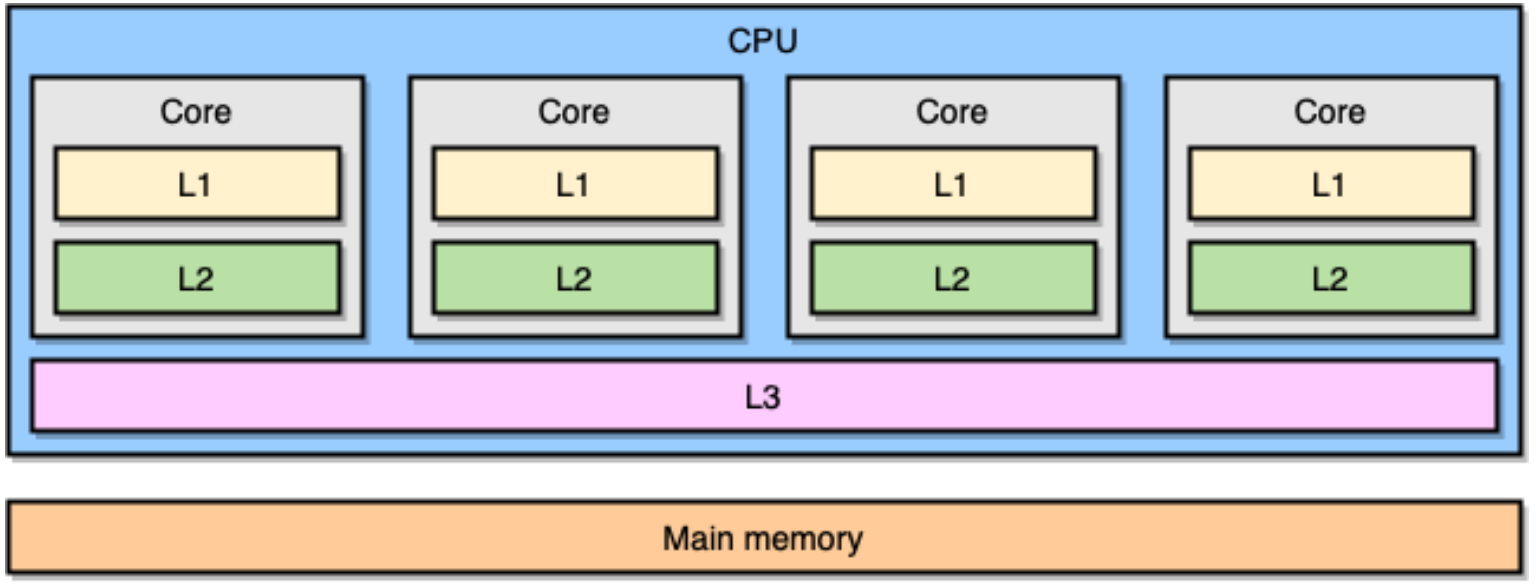
CS

Chrome

video



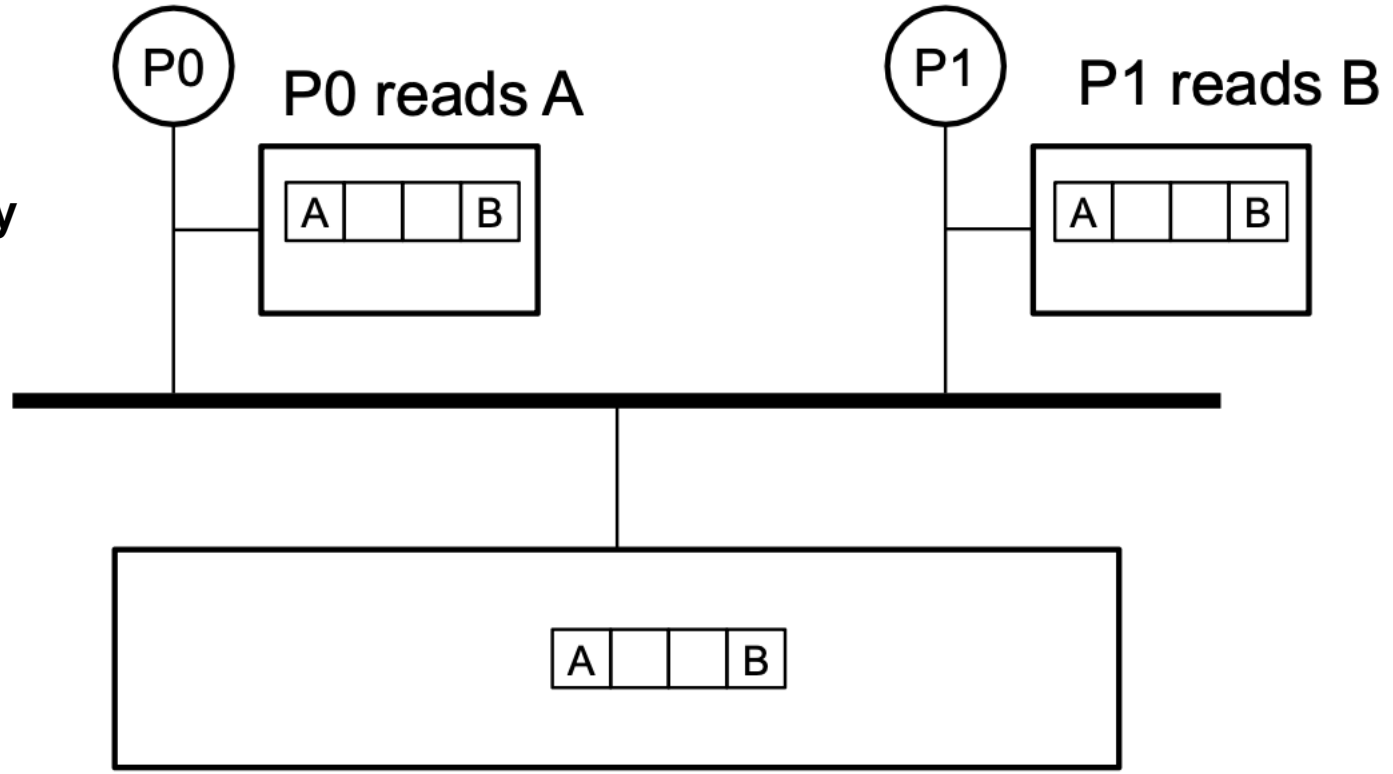
Cache Coherence



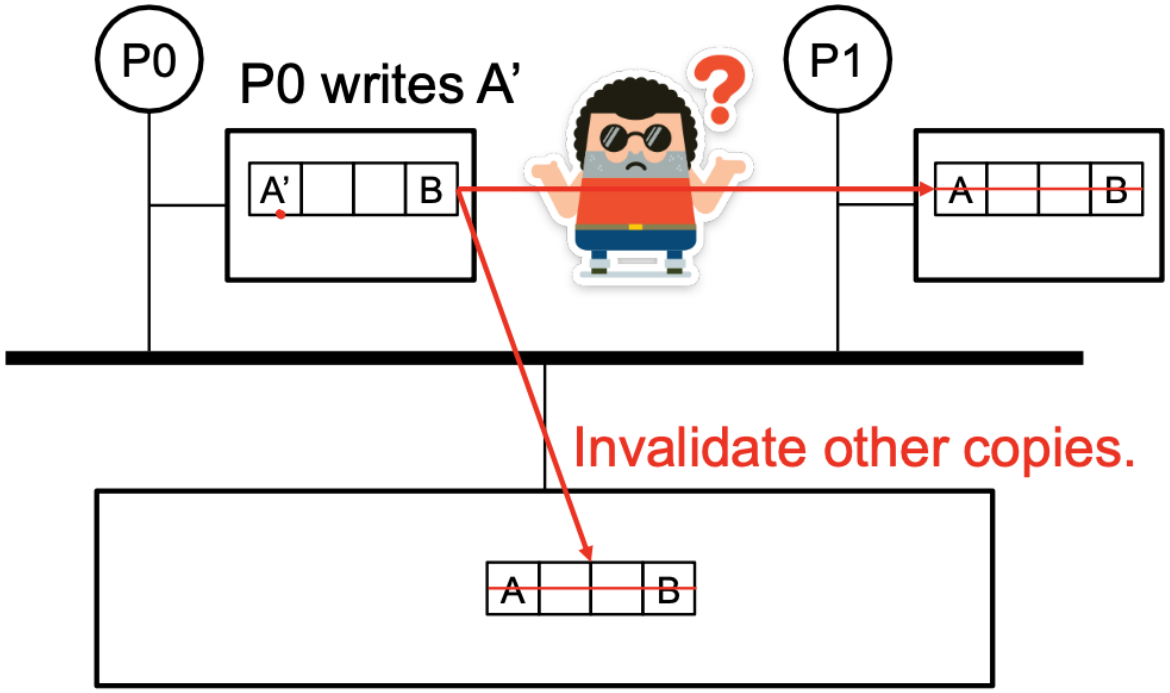
False Sharing

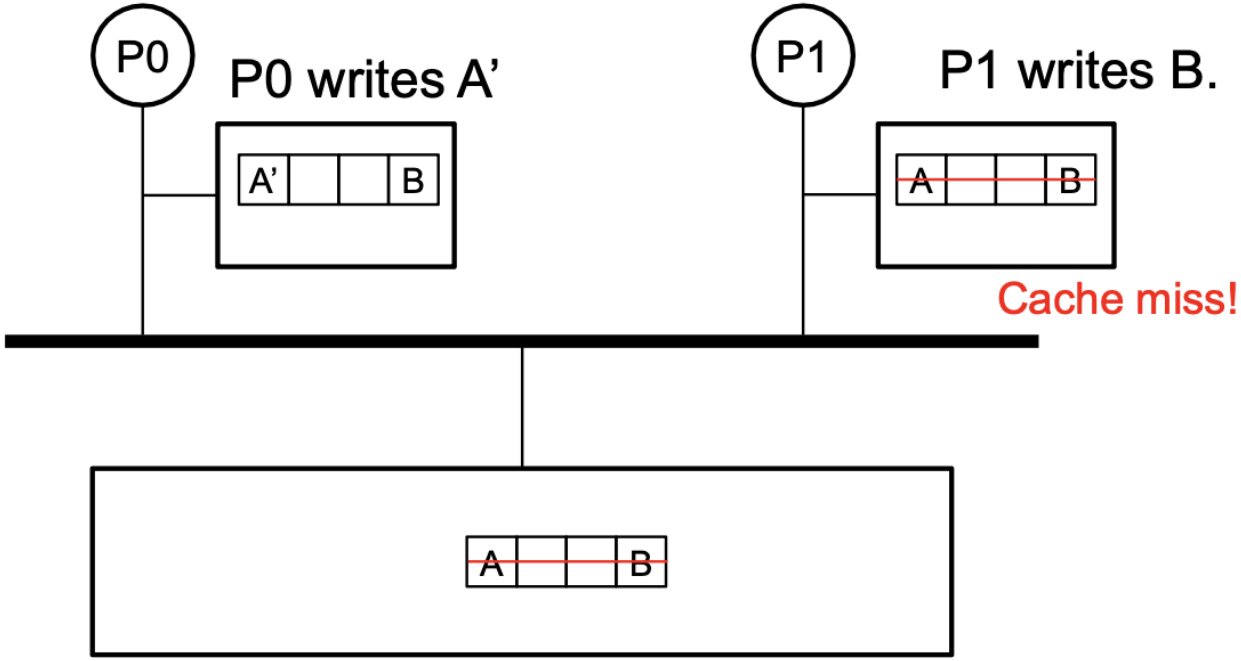
P0와 P1은 각각 다른 변수를 사용하지만 같은 캐시 라인을 사용함으로써 마치 두 변수를 공유하는듯한 효과

Spatial Locality



False Sharing





False Sharing

```
float result[4];
```

```
#pragma omp parallel for num_threads(4)
for (int y = 0; y < norow; y++) {
    for (int x = 0; x < nocol; x++) {
        result[y] += matrix_d[y][x] * vector[x];
    }
}
```

```
#pragma omp parallel for num_threads(4)
for (int y = 0; y < norow; y++) {
    float local_result = 0.0;
    for (int x = 0; x < nocol; x++) {
        local_result += matrix_d[y][x] * vector[x];
    }
    result[y] += local_result;
}
```

Code	1회	2회	3회	4회	5회	평균
false sharing 발생	43.9705	52.4872	46.5939	49.6930	44.6498	47.48
false sharing 해결	5.3261	4.8387	4.9272	4.7860	4.7302	4.92
직렬 코드	10.2030	10.3013	10.4348	10.1392	10.2308	10.26

subject 4.

Monitor

Monitor

Operating System Concepts, 10th. Ch 6.7

Semaphore, Mutex 락을 이용하면 임계구역 문제를 해결할 수 있다.
하지만 잘못 사용하면 다양한 유형의 오류가 너무나도 쉽게 발생할 수 있다.

```
signal(mutex);
    ... criticial section...
wait(mutex);
```

```
wait(mutex);
    ... criticial section...
wait(mutex);
```

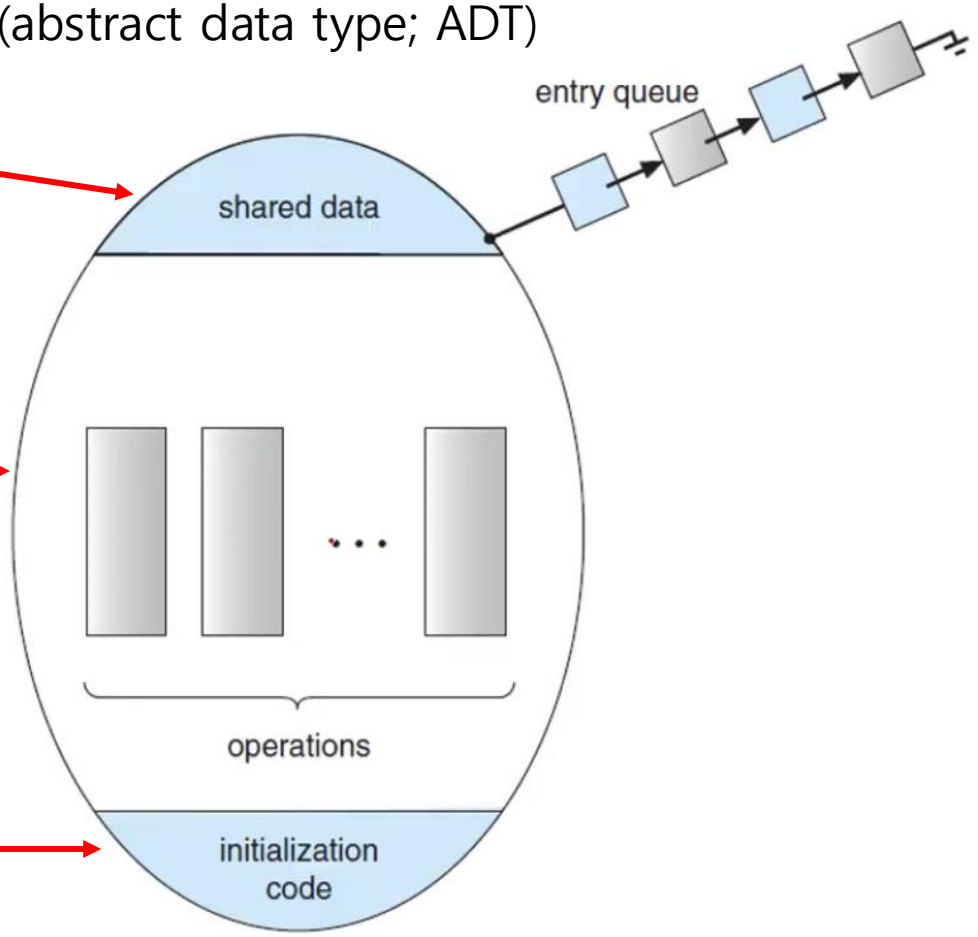
이러한 오류를 처리하기 위한 한 가지 전략은
간단한 동기화 도구를 통합하여 고급 언어 구조물을 제공하는 것이다.

| 버그는 적게, 프로덕트 생산성은 높게

Monitor

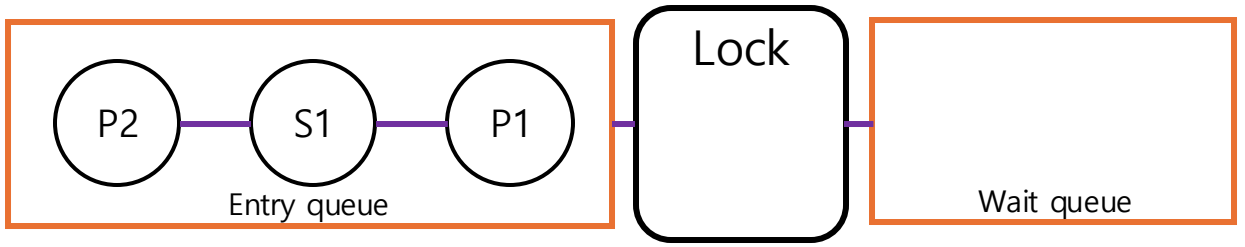
```
monitor monitor name
{
    /* shared variable declarations */

    function P1 ( . . . ) {
        . . .
    }

    function P2 ( . . . ) {
        . . .
    }
        .
        .
        .
        .
    function Pn ( . . . ) {
        . . .
    }

    initialization_code ( . . . ) {
        . . .
    }
}
```

Monitor data type
(abstract data type; ADT)

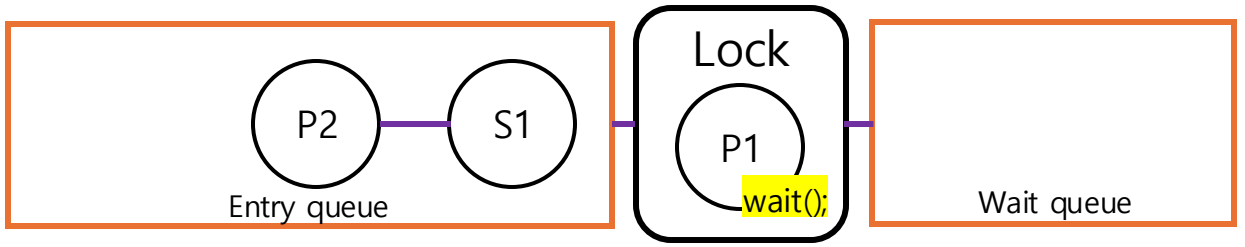


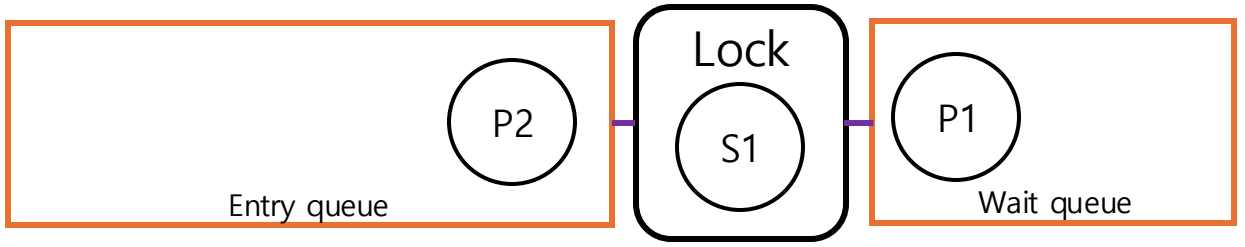
Java Monitors

```
public synchronized void insert(E item) {
  while (BUFFER.isFull()) wait();

  buffer[in] = item;
  in = (in + 1) % BUFFER_SIZE;
  count++;

  notify();
}

public synchronized E remove() {
  while(BUFFER.isEmpty()) wait();

  E item = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  count--;

  notify();
  return item;
}
```



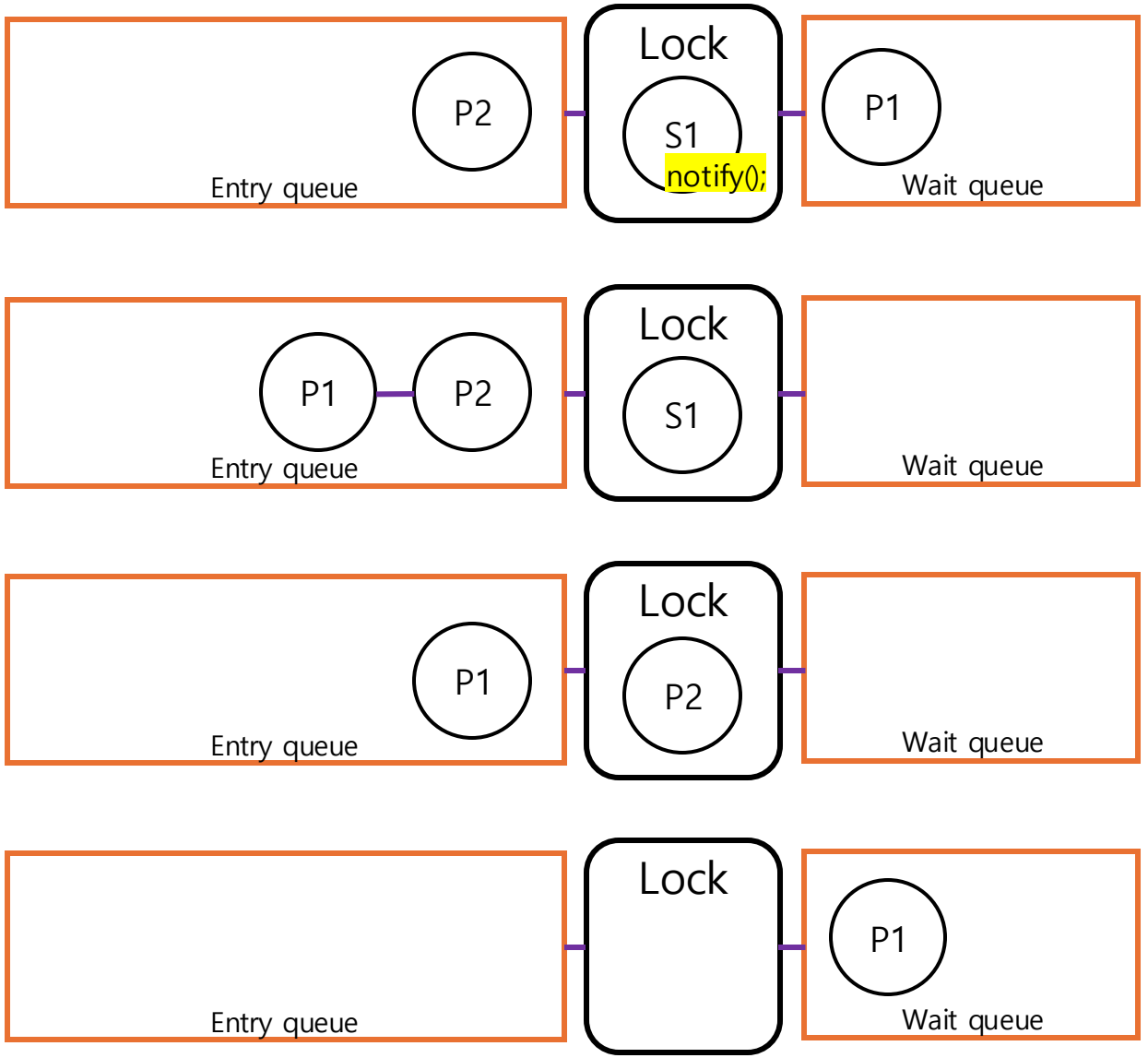
가정: BUFFER is Full





Java Monitors

```
public synchronized void insert(E item) {
  while (BUFFER.isFull()) wait();

  buffer[in] = item;
  in = (in + 1) % BUFFER_SIZE;
  count++;

  notify();
}

public synchronized E remove() {
  while(BUFFER.isEmpty()) wait();

  E item = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  count--;

  notify();
  return item;
}
```



Spinlock

- a **spinlock** is a lock that causes a thread trying to acquire it to simply wait in a loop ("spin") while repeatedly checking whether the lock is available
- Because they avoid overhead from operating system process **rescheduling** or **context switching**, spinlocks are efficient if threads are likely to be blocked for only **short periods**

- **Single core :** 어차피 문맥교환 필요
- **Multi core :** 스케줄링 시간 내에서 락이 풀릴거라 예상한다면?

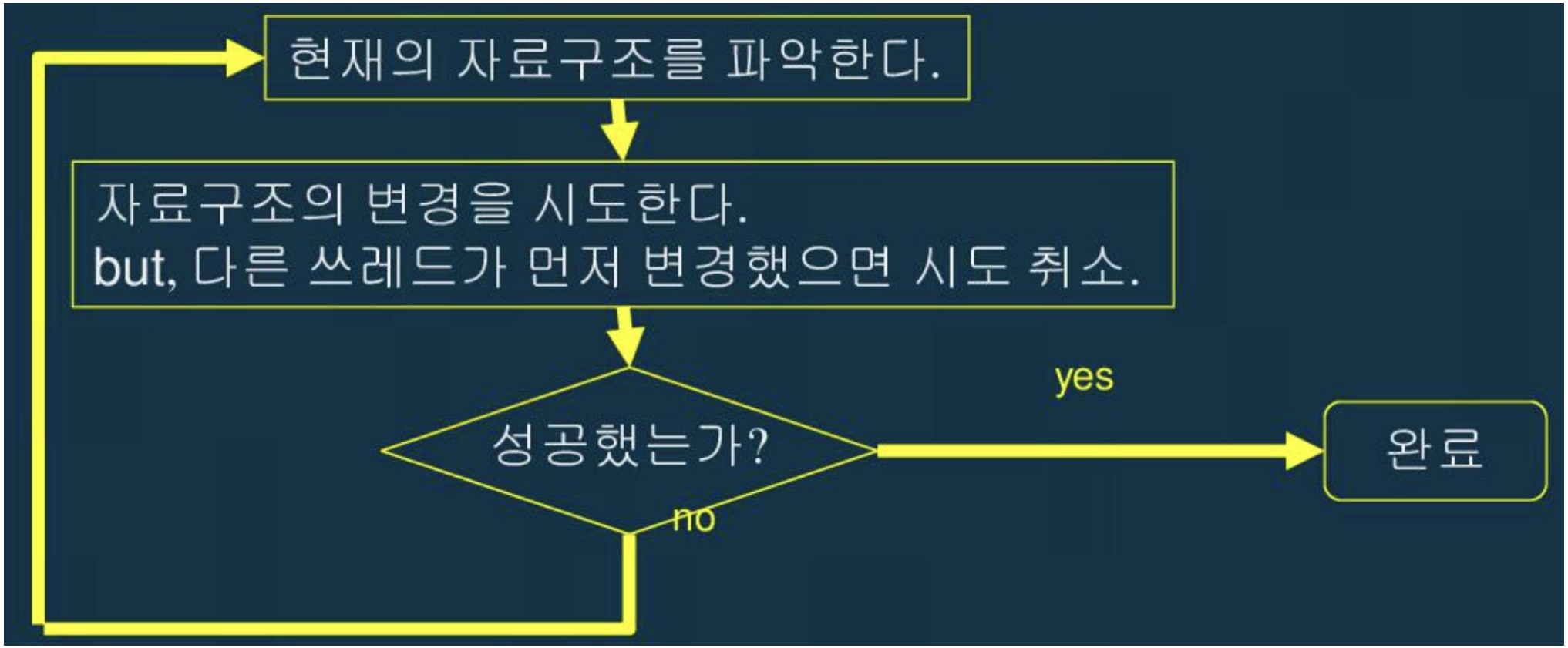
- **특히 다중 코어 컴퓨팅 시스템의 커널에서 널리 사용**

subject 5.

Lock-Free algorithm

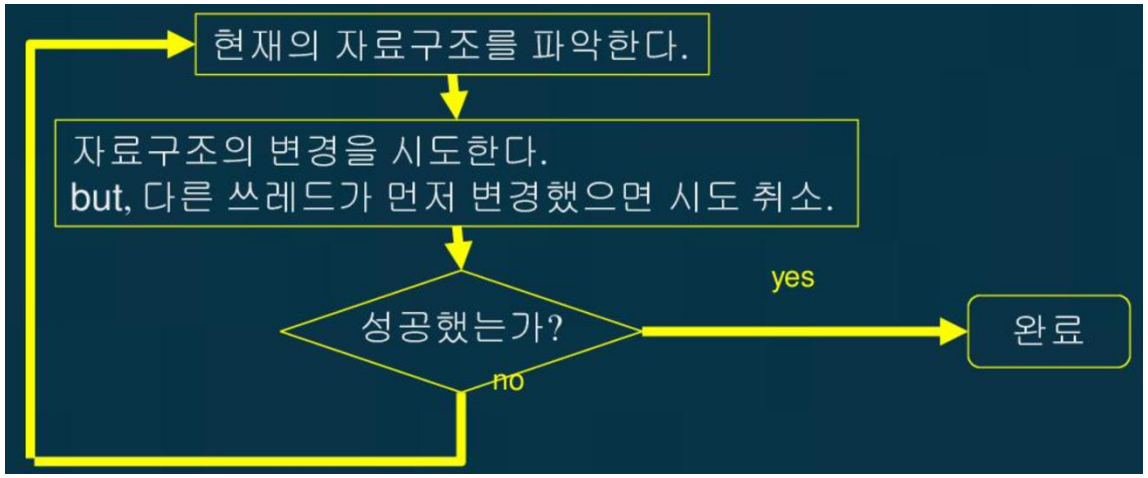
Lock-Free Algorithm

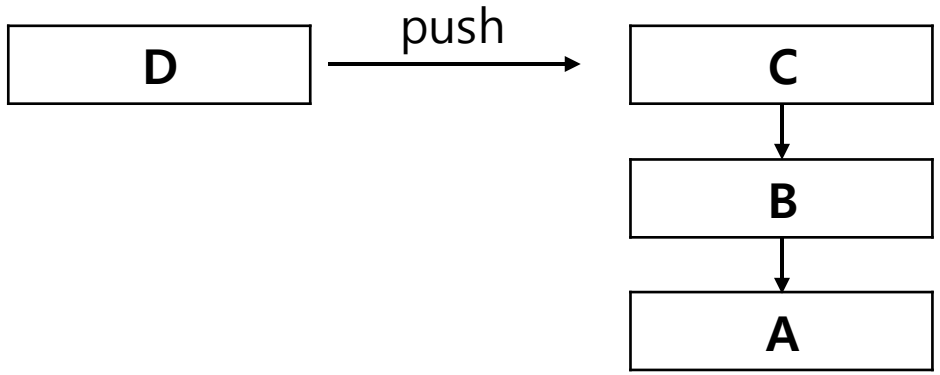
- 여러 쓰레드에서 동시 호출했을 때에도, 적어도 한 개의 호출이 완료되는 알고리즘
- No using Lock, Non-Blocking Algorithm
- Wait-Free Algorithm : 모든 스레드가 한정된 시간 내에 진행할 수 있는 것을 보장

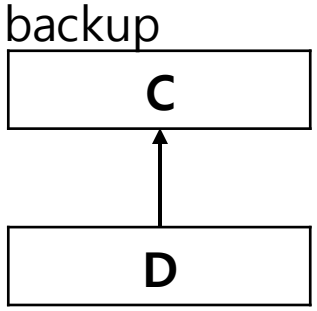


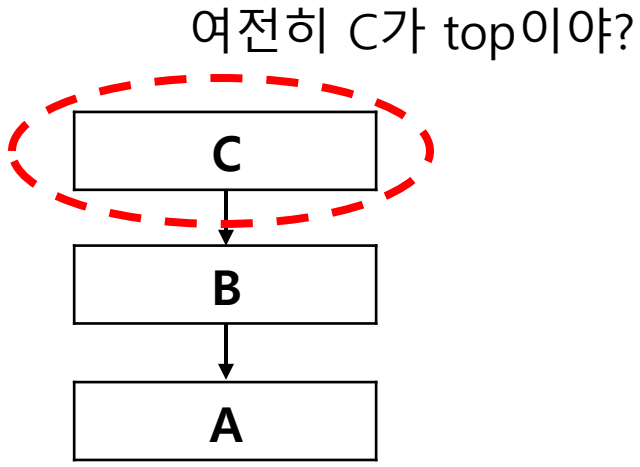
Lock-Free Algorithm

```
void push(T val) {
    do {
        Node* nNode = new Node(val);
        Node* top = st.top();
        nNode->next = top;
    while(!CAS(st.top(), top, nNode))
}
```









Lock-Free Algorithm

Lock-Free 알고리즘의 장단점

- 장점
  - 성능!!
  - 높은 부하에도 안정적!!
- 단점
  - 생산성 : 복잡한 알고리즘
  - 신뢰성 : 정확성을 증명하는 것은 어렵다.
  - 확장성 : 새로운 메소드를 추가하는 것이 매우 어렵다.
  - 메모리 : 메모리 재사용이 어렵다. ABA문제

subject 6.

Deadlock

Deadlock의 필요충분 조건

1. 상호 배제(Mutual Exclusion)

- 할당 후 반환까지 한 프로세스만 사용하는 자원이어야 한다.

2. 점유하며 대기(Hold-and-wait)

- 프로세스는 최소한 하나의 자원을 점유한 채, 현재 다른 프로세스에게 **점유된 자원을 추가로 얻기 위해 반드시 대기**하여야 한다.

3. 비선점(No preemption)

- 자원이 그 자원을 점유하고 있는 프로세스로부터 **강제적으로 방출될 수 없고**
- 점유하고 있는 프로세스가 작업을 종료한 후 그 프로세스에 의해 **자발적으로만 방출될 수 있다.**

4. 환형 대기(Circular wait)

- 대기 프로세스의 집합 {P0, P1, … , Pn}이 존재할 때
- P0는 P1이 가진 자원을 기다리고, P1은 P2를, … P(n-1)은 Pn을, Pn은 P0가 가진 것을 기다린다.

Methods for Handling Deadlocks

문제를 무시하고, 교착 상태가 시스템에서 절대 발생하지 않은 척한다.

- 현대의 대부분의 운영체제가 사용하는 방법

시스템이 결코 교착 상태가 되지 않도록 예방하거나 회피하는 프로토콜을 사용한다.

- 응용 소프트웨어 개발자가 취해야 하는 태도

교착 상태가 되도록 허용한 다음에 복구 시킨다.

- 데이터베이스 등

Deadlock Prevent

1. “상호 배제” 조건을 부정

일반적으로 불가능

2. “점유하며 대기” 조건을 부정

2가지 방법

- 모든 자원을 실행 전에 확보하도록 한다. (대기 없음, but 사용도 안하면서 가지고 있으면서 성능하락)
- 자원을 가지고 있지 않은 경우에만 요청하도록 한다.(가지고 있는 자원을 해제 후 다시 요청)

3. “비선점” 조건을 부정

나의 모든 자원을 방출하거나, 상대방을 선점시키거나

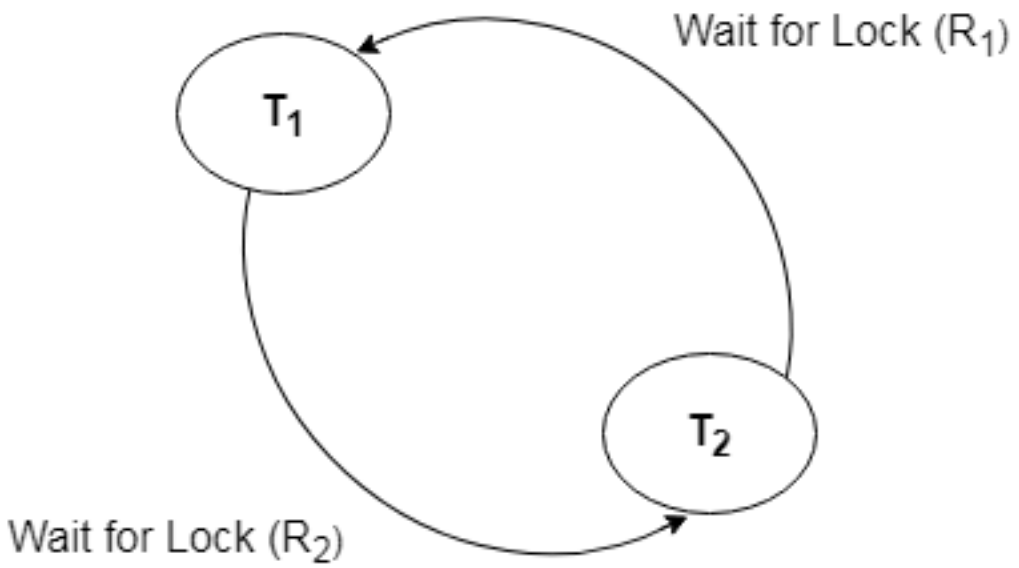
4. “환형대기” 조건을 부정

각 자원 유형에 증가하는 순서의 일련번호를 부여

번호가 증가하는 순서대로만 자원을 요청하도록 강제

Wait-for graph

- Directed graph
- $T_i \rightarrow T_j$: T_i은 필요한 리소스를 T_j가 놓을 때까지 wait
- wait-for graph가 사이클을 갖는다 = 시스템이 deadlock 상태



Deadlock detection in InnoDB

- Deadlock 상태에 빠지지 않았는지 체크하기 위해 잠금 대기 목록을 Wait-for graph 형태로 관리
- 데드락 감지 스레드가 주기적으로 그래프 검사
- Deadlock에 빠진 트랜잭션들을 찾아서 그 중 하나를 강제 종료
  - 언두 로그 양이 적은 트랜잭션을 롤백

- innodb_deadlock_detect : 데드락 감지 스레드를 작동할거냐?
- innodb_lock_wait_timeout : 락 대기가 일정 시간이 지나면 자동으로 롤백

Reference

operating system(korea university, 유혁)

Operating System Concepts (10/E, Silberschatz)

Computer Systems A Programmer’s Perspective (3/E, Randal E. Bryan)

Computer Organization And Design (6/E, David A. Patterson)

Web Application Server (with Java)

임) 2024.11.05. 19시

Java] Socket, Stream

HTTP] Basic

Java] Multi-threads

HTTP] Add required

HTTP] HttpRequest, HttpResponse

HTTP] Servlet

Java] Reflection, Annotation

HTTP] Final