



SP - 4.3 Thread-based Concurrent Server

hyeok's Log · 2022년 4월 16일

팔로우

sp


2



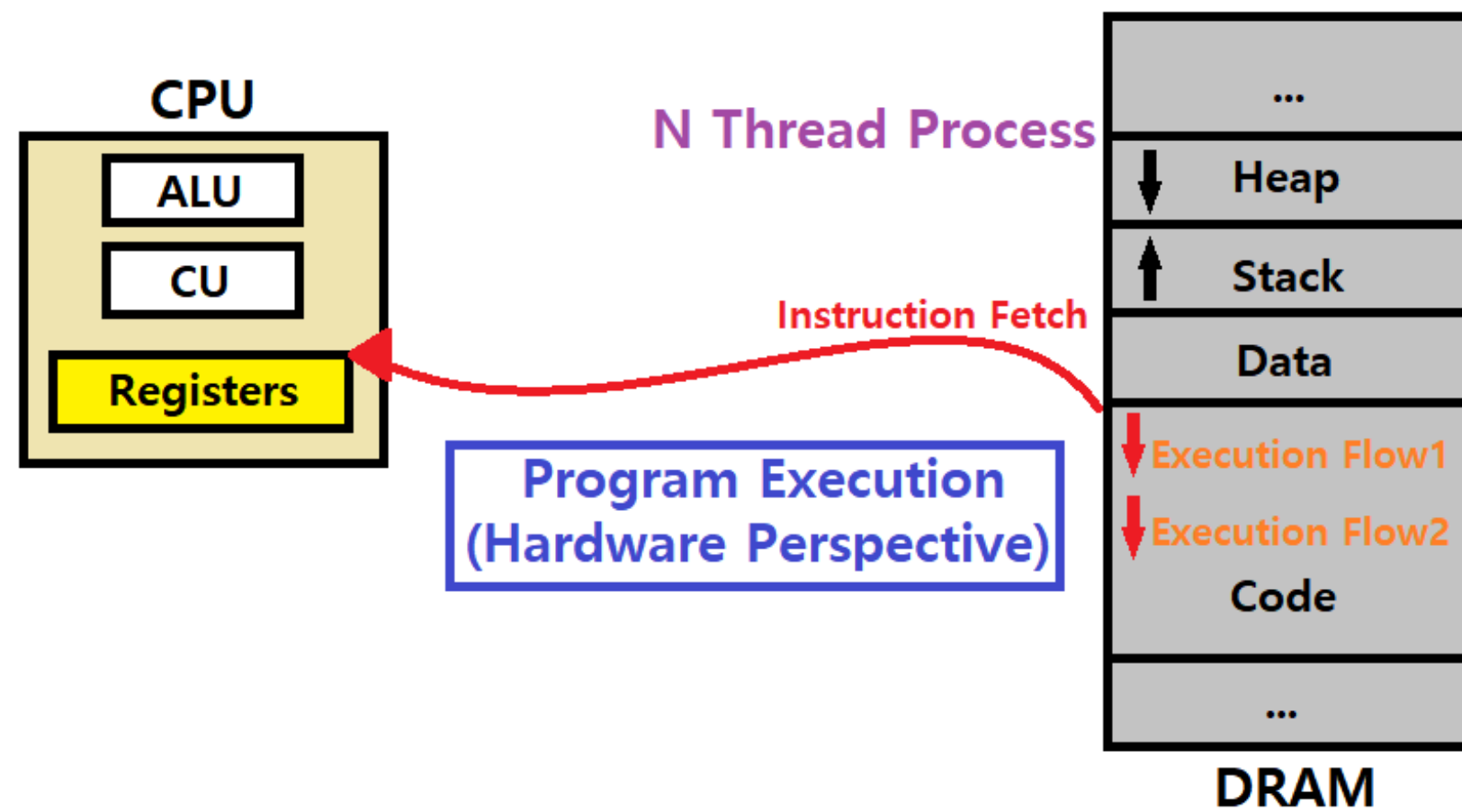
SystemProgramming



▼ 목록 보기

14/29





현재 우리는 Concurrent Server를 구축하는 3가지 방법론에 대해 논하고 있다. 지난 포스팅에서 우린 Process-based와 Event-based의 디테일과 차이점에 대해 알아보았다. 오늘은 마지막 방법론인 Thread-based Server에 대해 알아보는 시간을 가질 것이다.

Thread의 이해

Background Idea

우리가 프로그램을 구동한다고 해보자. 하드웨어에는 CPU, 메인 메모리(DRAM)가 있다. **알다시피, 메인 메모리 DRAM은, Power가 Charging되는 상태여야 메모리 셀의 데이터를 유지하는 Volatile Memory이다.**

한편, 이때 프로세스 Pa가 생기면, 메인 메모리에 해당 프로그램의 'Sequence of Instructions'가 적재된다. 적재될 때, 프로세스는 'Code-Data-Stack-Heap'이라는 영역으로 구분되어 적재된다.

동적인 메모리들이 Stack과, Heap에 들어가는데, Stack은 Local Variable이나 Function Return Address를, Heap은 Dynamic Allocation Variable을 담는다. **Stack과 Heap은 서로 마주보면서 나가는 방향으로 공간을 차지한다.** 메모리 주소 번지값이 High에서 Low로 가는 방향으로 **Code, Data, Stack, Heap**이 순서대로 자리한다(이는 다를 수 있다).

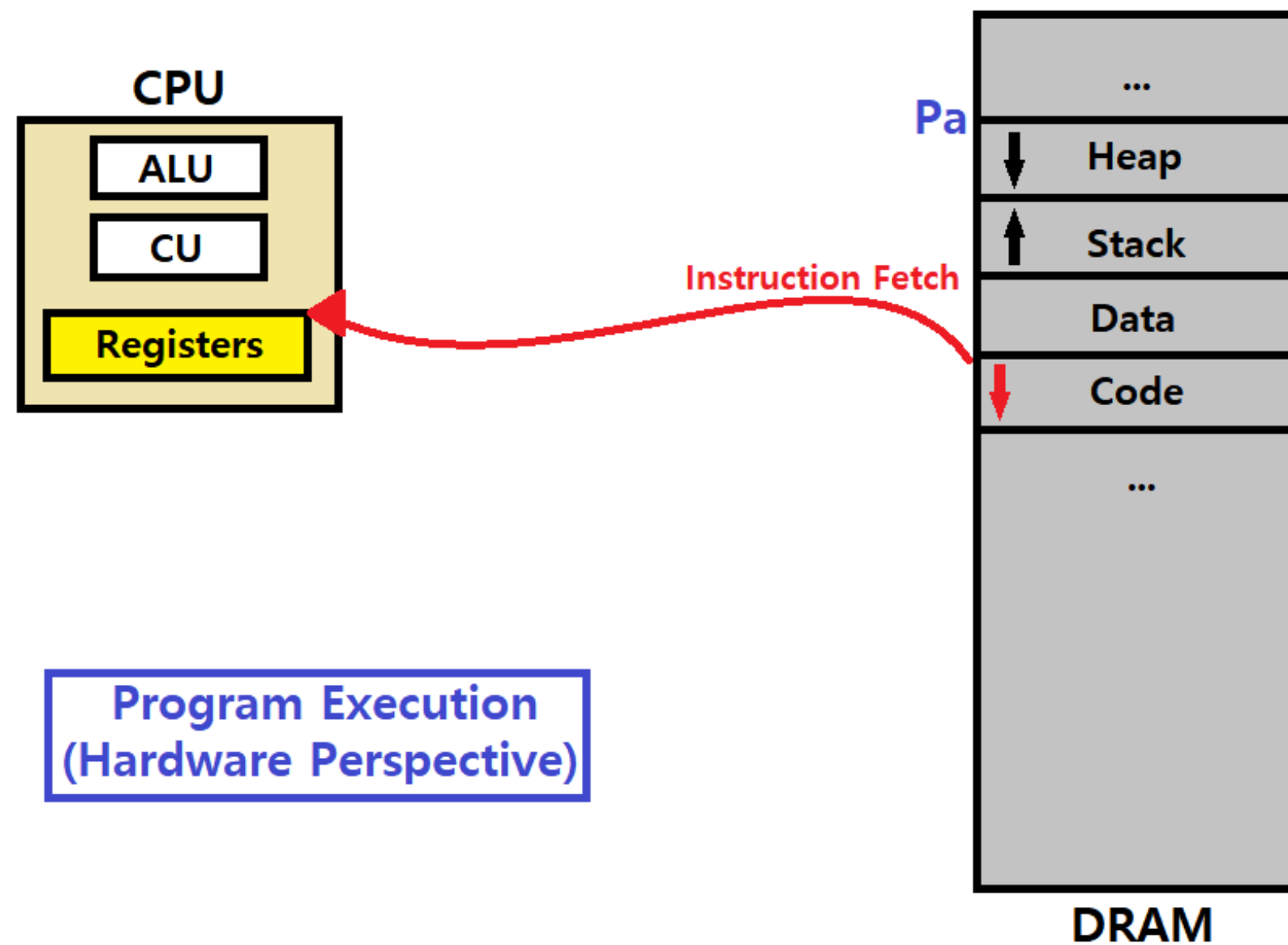
앞서 말했듯이, 프로세스는 'Object Code / Machine Code / Instruction의 Set'이다.

한편, CPU 내부를 보자. 안에는 Register Set이 있다. 그 안에는 PC/IP(Program Counter, Instruction Pointer), IR(Instruction Register), SP(Stack Pointer), 범용 레지스터(x86 기준 EAX..) 등이 있다. 이 Registers와 더불어, ALU(Arithmetic and Logical Unit), CU(Control Unit)가 함께 CPU에 존재한다.

메인 메모리에 연결되어 있는 Bus에는 Address Bus, Control Bus, Data Bus가 있다. 이 세 가지 Bus를 통해서 CPU와 Main Memory가 데이터를 교환한다.

이때, CPU가 구동되면, 'Instruction Fetch -> Instruction Decode -> Execution -> Write Back'의 과정이 반복적으로 수행된다. PC를 Increment해가면서 말이다. CPU 입장에서, **Fetch를 위해 Pa의 Code 메모리 영역 내에 저장된 명령어를 가져와야한다.** 첫 번째 명령어부터 말이다.

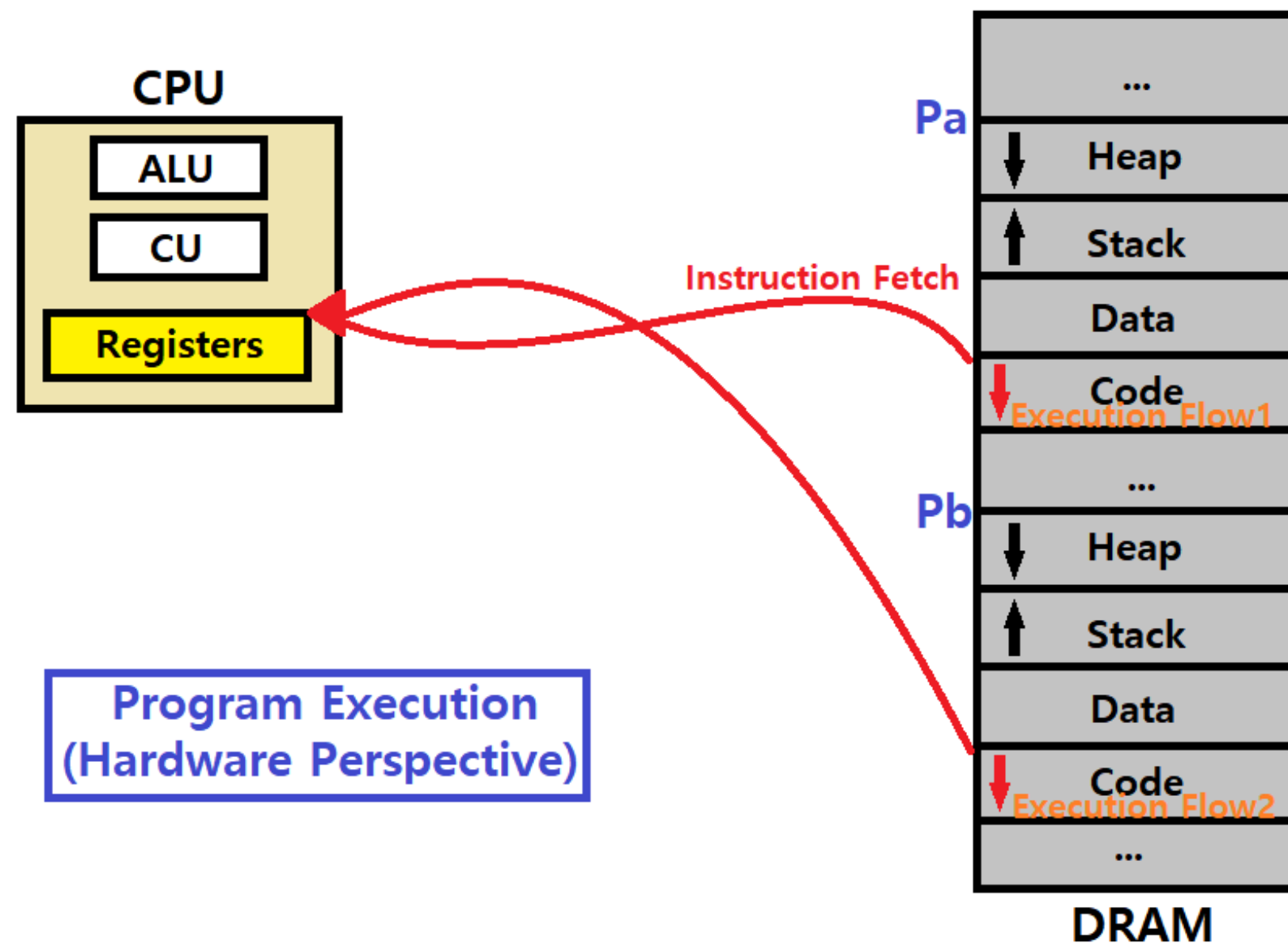
여기까지가 하드웨어 관점에서의 프로그램 실행 루틴이자, Execution Flow이다.



이러한 Execution Flow는 프로세스 하나마다 하나씩 존재한다. 논리적인, Conceptual한 관점에서 말이다. 실행해야하는 명령어들의 Sequence가 곧 Execution Flow인 것이다.

이때, Pa가 fork하여 Pb가 만들어졌다고 해보자. 동일하게 메인 메모리 공간에 Code-Data-Stack-Heap 영역이 적재된다. CPU는 하나이고, 프로세스는 두 개인 상황이다. 알다시피, CPU는 Time Quantam을 기준으로 Pa와 Pb를 왔다 갔다 하면서 수행한다.

fork 후 exec을 하지 않았다고 해보자. 그렇다면, Pb는 Pa와 동일한 일을 하고 있다. 동일한 일을 하지만, 프로세스는 두 개이기 때문에 Execution Flow는 두 개인 상황이다.

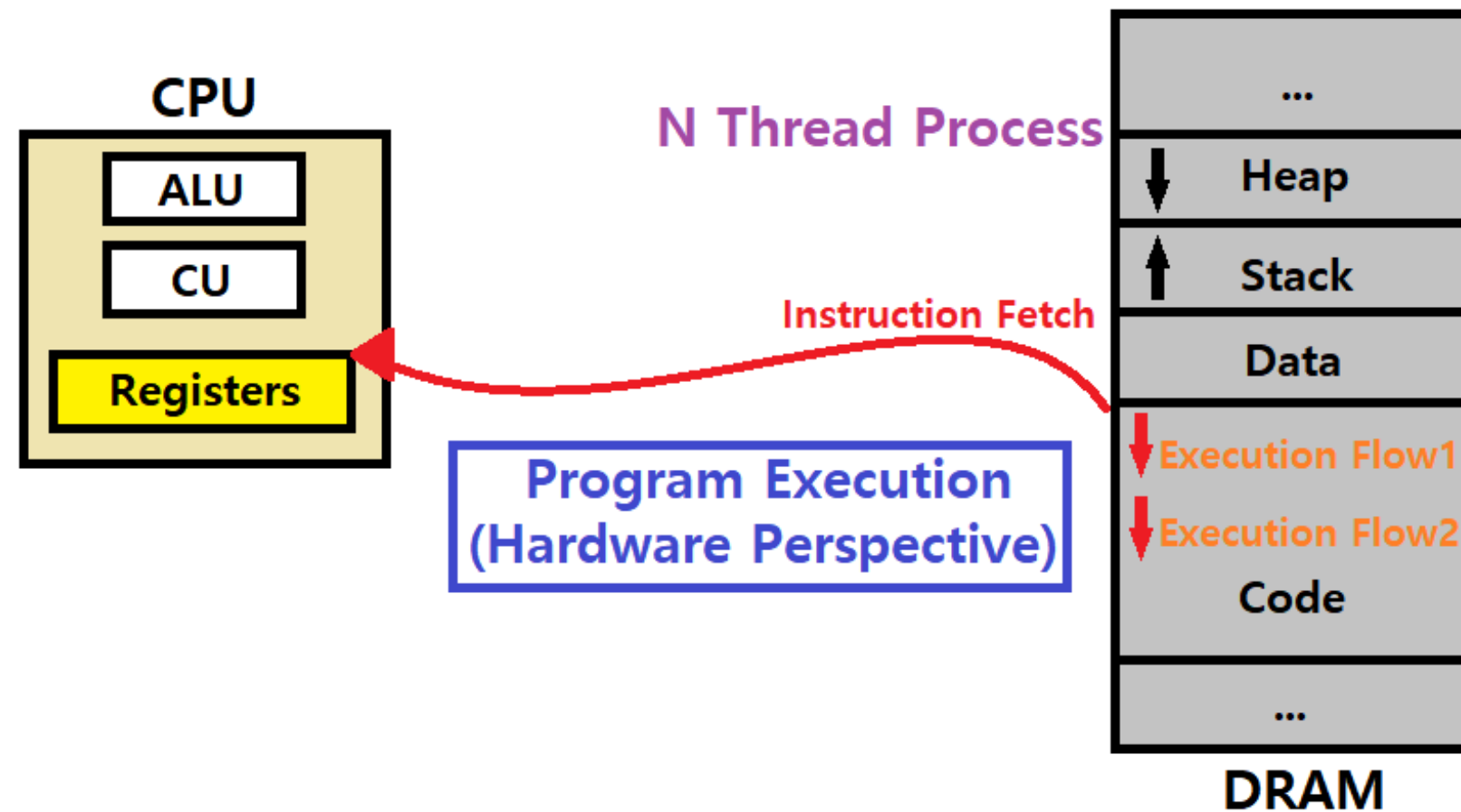


Process-based Concurrent Server를 구현한다고 하면, Pa와 Pb의 메인 메모리 적재 공간 위치는 서로 **Overlap되지 않은 별개의 Address Space**이다. 그렇기 때문에, 이전 포스팅에서 언급한 것처럼, Process-based Server는 Overhead가 심하다고 했다. 이때, 우리는 다음과 같은 생각을 해볼 수 있다.

복수의 별도 Execution Flow를, 굳이 **fork**를 띄워서 만들지 말고, 즉, 프로세스를 추가로 생성하지 말고, 어차피 서버 구축 상황에서는 **Pa나 Pb나 둘 다 같은 코드를 기반으로 한 Execution Flow**인데, 이를 더 효율적으로 처리할 수 있는 방법이 있지 않을까? ★

이런 아이디어로 등장한 개념이 **Thread**이다.

N Thread Process : 하나의 프로세스에 대해 여러 개의 Execution Flow를 띄울 수 있는 프로세스. 새로 프로세스를 생성하지 않고, 동일한 일을 하는 Execution Flow를 새로 만드는 것이다.



- Thread는 Heap과 Code, Data 부분은 그대로 공유하고, Stack만 별도로 가지는 '가상의 프로세스'라고 생각하면 된다.
 - 별도의 Execution Flow를 유지하기 위해서 Stack만 달리하는 것이다. ★

Thread vs Process

Process에 대한 Traditional View

Process = Process Context + Code/Data/Stack (Heap은 생략)

Process = **Process Context**(**Program Context + Kernel Context**) +
Code/Data/Stack

- **Process Context**

- **Program Context**

- Data Registers
 - Condition Codes
 - Stack Pointer
 - Program Counter

- **Kernel Context**

- VM Structures
 - File Descriptor Table
 - brk Pointer

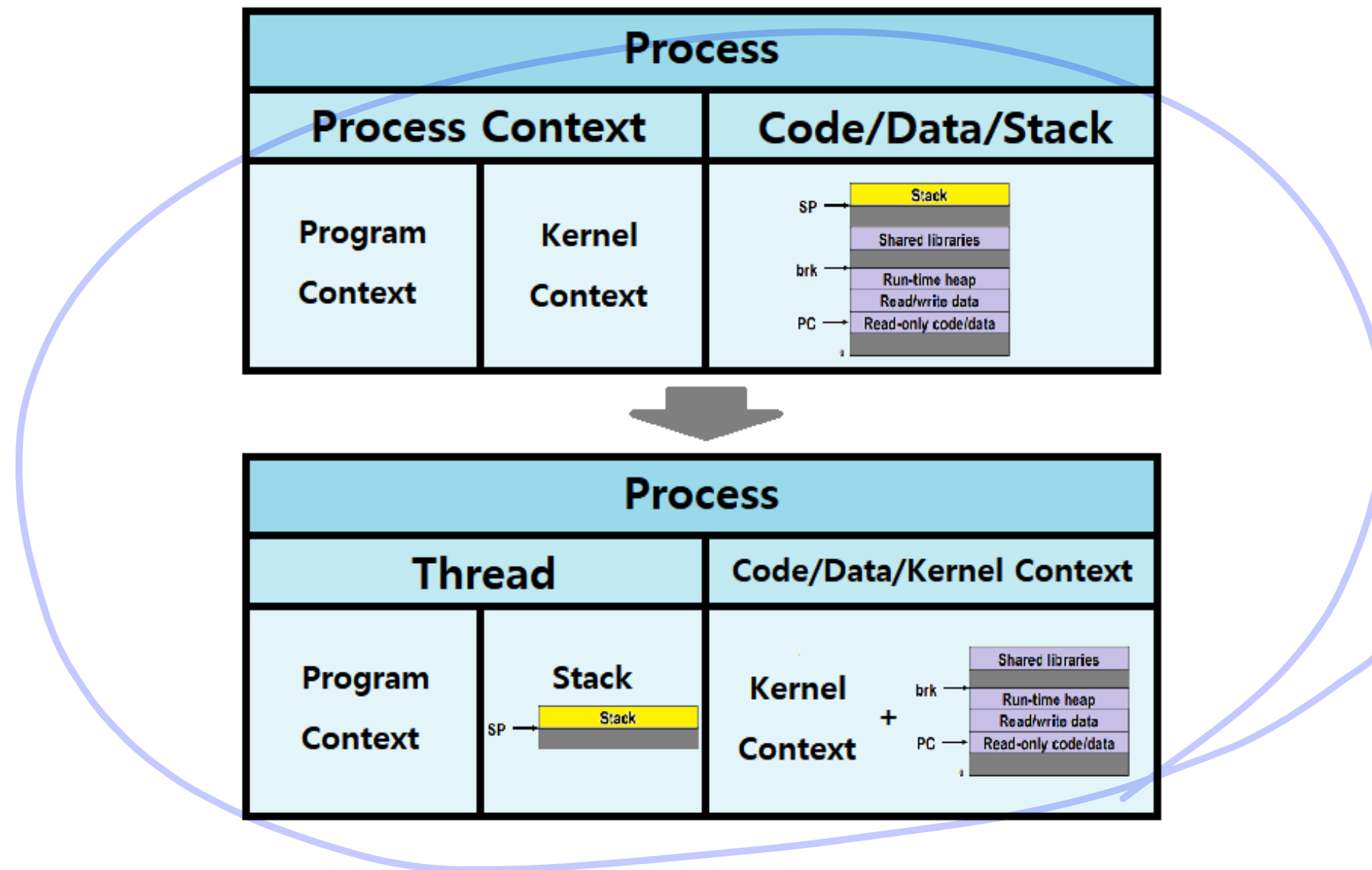
- **Code/Data/Stack/(Heap)** : 메인메모리에 적재된 프로세스의 영역들

Process에 대한 Alternative View

Process = Thread + Code/Data/KernelContext

Process = **Thread**(**Program Context + Stack**) + Code/Data + Kernel
Context

- 앞선, 전통적인 Process View에서, Stack을 Process Context로 가져오고, 거기서 Kernel Context를 빼서 Code/Data에 붙여준다.
- 변화된 것은 하나도 없다. 오로지 '관점(View)'만 바뀐 것이다. ★★★



- 이러한 관점에서 기반하여 Thread 개념을 뽑아낸 것이다.
 - Program Context를 Thread 관점에서는 Thread Context라고 부른다.
- Traditional View of Process에서 프로세스를 fork하면, 그대로 양쪽 모두, 즉, Process Context와 Code/Data/Stack이 모두 복제되어 Child Process가 생성된다. ★

- 반면, **Alternative View of Process**에서 프로세스를 복제하면(여기선 fork라 하지 않음), **Code/Data/KernelContext**는 그대로 유지하고, 오로지 **Thread(Stack + Program Context)**만 복제한다. ★



- Thread와 Process의 공통점

- **둘 다 모두 고유의 Logical Control Flow, Execution Flow를 가진다.**
- 둘 다 Concurrent Flow를 가질 수 있다.
- **둘 다 Context Switch에 영향받는다. ★**
 - Thread도 OS Scheduling의 대상이다. 사실, OS Scheduling, Context Switch의 기준은 Process가 Thread다. Thread는 Process보다 더 큰 범위이므로. ★

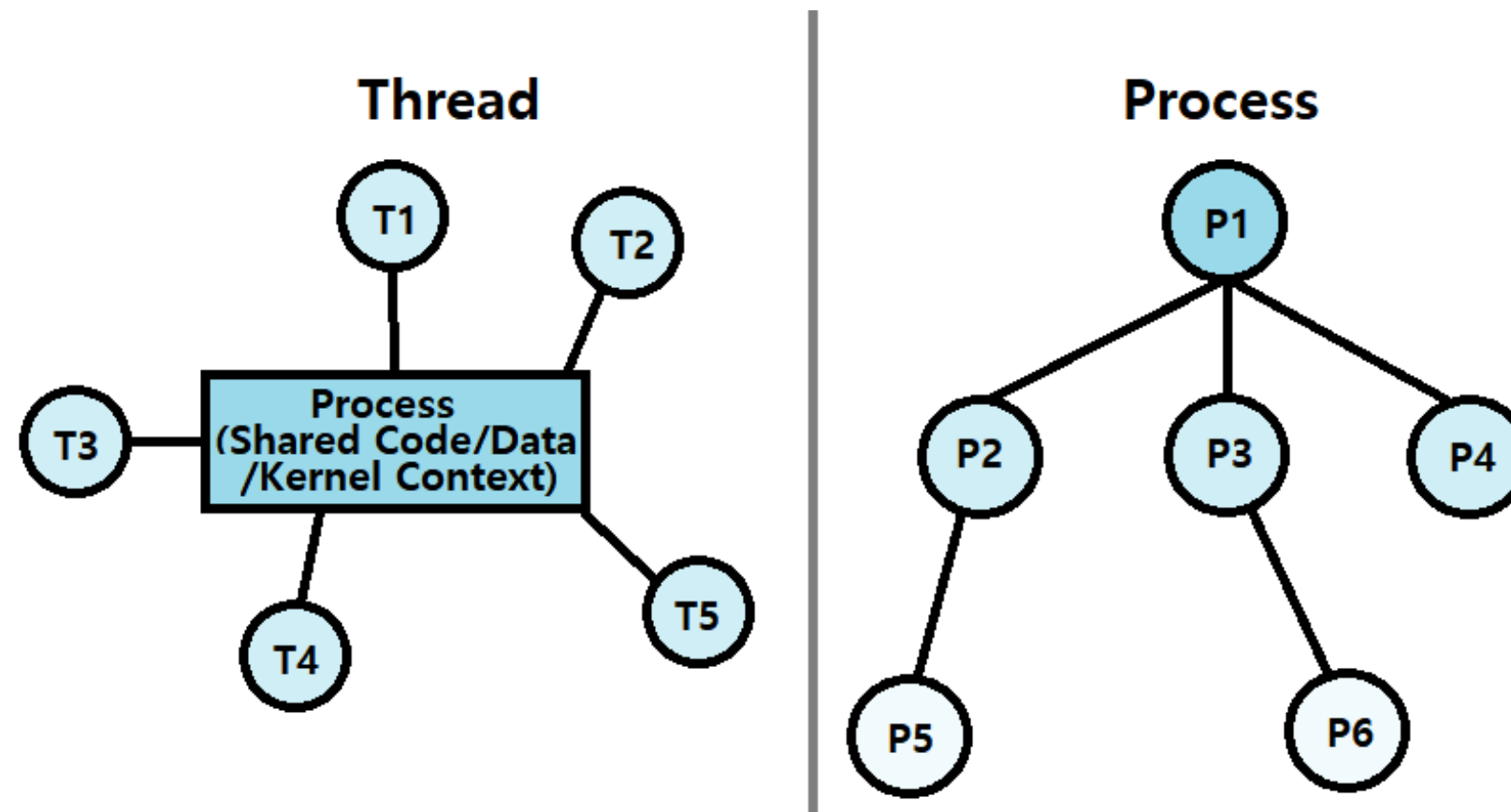
- Thread와 Process의 차이점

- Thread는 Local Variable을 위한 Stack을 제외하고는 모두 공유한다.
 - **Code/Data/Kernel Context를 공유한다.**
 - 프로세스는 모두 별도로 존재한다.
- Thread 관리는 Process 관리에 비해 Overhead가 상대적으로 덜하다.
 - Process의 생성(fork)과 Reaping은 Thread의 그것에 비해 약 2배 정도 더 Overhead를 보인다. ★

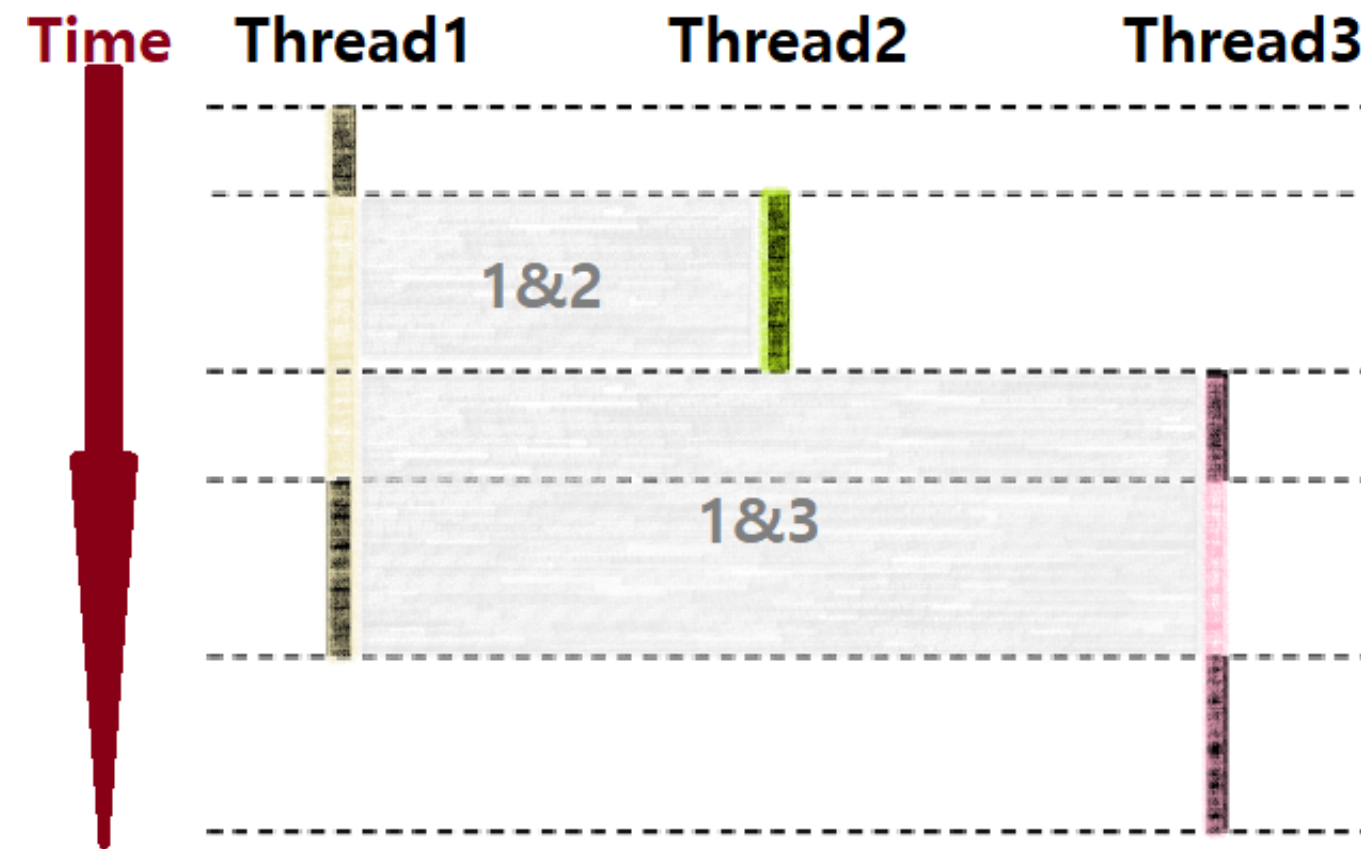
Thread Details

- 프로세스 하나에는 여러 개의 Thread를 만들 수 있다.
 - 각 Thread는 고유의 Logical Control Flow를 가진다. ★
 - 각 Thread는 동일한 Code/Data/Kernel Context를 가진다. ★
 - 각 Thread는 별도의 Stack을 가진다. ★
 - 이때, 각 Thread의 Stack부분은 다른 Thread에게 참조될 수 있다.
 - 즉, Thread의 Stack은 보호되지 않는다. (보안 이슈 발생 가능)

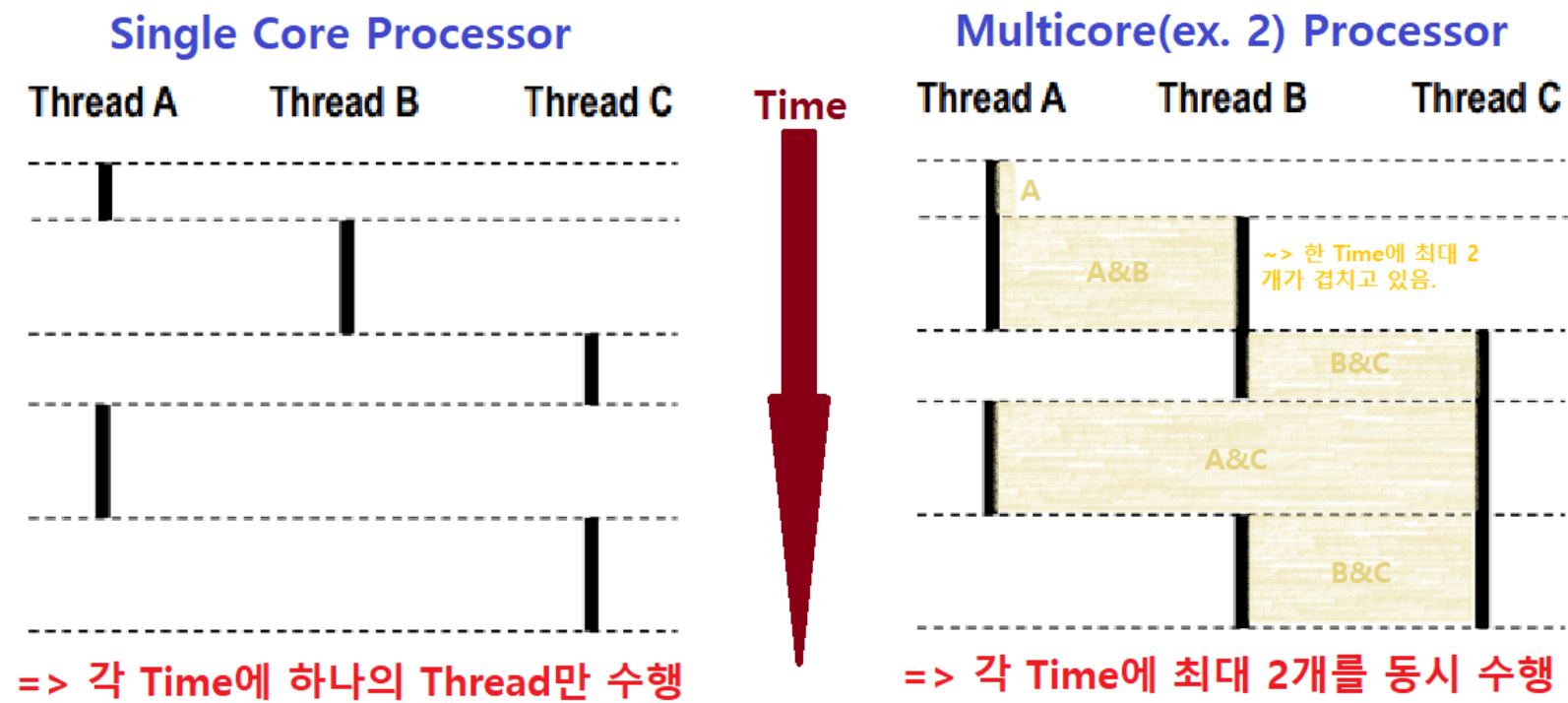
- 각 Thread는 고유의 Thread ID인 'TID'를 가진다. (Just like PID)
- 하나의 프로세스(스레드)에서 스레드를 새로 생성하면, 이를 'Peer Thread'라고 부른다.
- Thread의 Logical View
 - Thread는 자신의 Process에 대해 동등한 Peer들과 함께 연결되어 있다.
 - Process의 Child들이 Hierarchy를 구축하던 것과 대조적이다.



- Concurrent Threads
 - Thread를 가장 쉽게 이해하는 방법은, 'Code/Data/Kernel 영역을 공유하는 별도의 가상 프로세스'라고 생각하는 것이다.
 - 따라서, Thread 간의 관계를 볼 때, 앞서 프로세스의 관계를 보았던 것처럼 보면 된다. ★



- 따라서, 위의 그림을 보면, T1 & T2, T1 & T3가 서로 **Concurrent Flow** 관계이다. ★
- 한편, T2 & T3는 각 Thread의 시작과 끝이 겹치지 않으므로 **Sequential Flow** 관계이다. ★
- **Concurrent Threads Execution**
 - **Single Core Processor** : 시간 단위로 쪼개서 **Parallelism** 시뮬레이션을 구현
 - **Multicore Processor** : 진정한 의미의 **Parallelism**을 구현
 - 왜 '진정한 의미'이냐? 코어가 **N개**이면, 최대 **N개의 Thread**를 동시에 수행할 수 있기 때문이다.



- Multicore일 때를 주목하자. Core가 2개인 CPU이다.
 - Time에 따라 A, A&B, B&C, A&C, B&C가 병렬처리(Parallelism)되고 있다. ★
 - 한편, 세 Thread가 모두 Concurrent Flow 관계이다. ★

POSIX Thread Interface

우리는 아래와 같은 POSIX 제공 Interface를 이용해 Thread를 다룰 것이다.

```
pthread_create()      // 프로세스의 fork같은 개념
pthread_join()        // 프로세스의 wait같은 개념
pthread_self()        // TID를 알아낸다.
pthread_cancel()       // Thread 종료
pthread_exit()         // Thread 종료
exit()                // Thread 종료
```

Thread Interface를 이용해서 간단한 프로그램을 아래와 같이 만들어볼 수 있다. Stevens Style의 Wrapper를 씌웠다.

```
/* 생성될 Thread가 수행할 Routine */
void *whatToDo(void *vargp) {
    printf("친구들아 안녕? 나는 Thread라고 해!\n");
    return NULL;
}
```

```

}

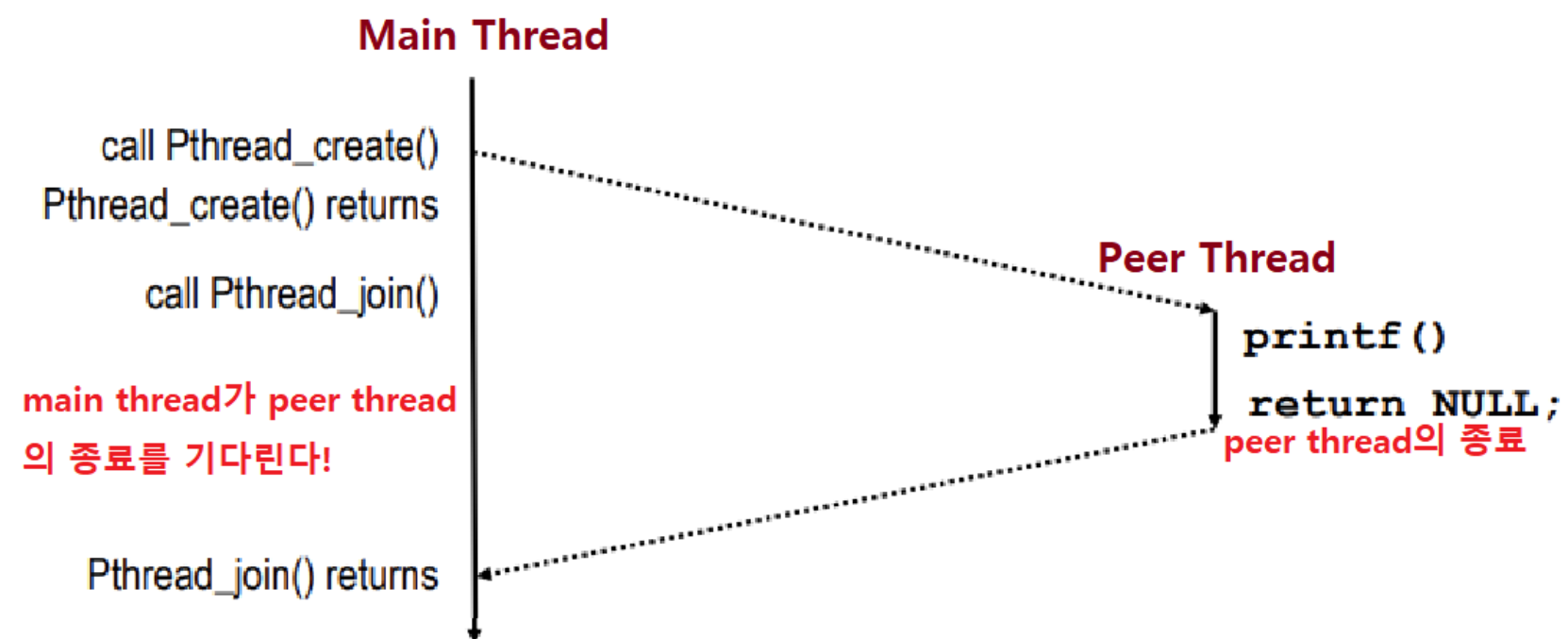
int main(void) {
    pthread_t tid;

    Pthread_create(&tid, NULL, whatToDo, NULL);    // Thread 생성
    Pthread_join(tid, NULL);                      // Thread Reaping

    return 0;
}

```

- Main Thread(=Process)가 있고, pthread_create를 이용해 새로운 Thread를 Create하면, Peer Thread가 생성된다.
 - *Peer Thread는 동일한 코드를 공유하지만, 별도의 Execution Flow를 가진다.*
- pthread_join을 이용해 생성된 Thread가 종료되는 것을 기다린다.
- pthread_create(TID 변수 주소값, 일반적으로 NULL, Thread Routine, Routine의 Arguments);
- pthread_join(TID 변수값, 리턴값);
- 아래와 같은 흐름을 가진다.



Thread-based Server

Implementation Details

Thread-based Server : 기본적으로 **Process-based Server** 방법론과 매우 유사하다. 단지, **Process** 대신 **Thread**를 이용하는 것이다.

- Thread 개념을 이해했다면, **Thread-based**가 **Process-based**보다 시공간적 **Overhead**가 상대적으로 적다는 것을 바로 알 수 있다. ★
- 반면에, 앞서 **Thread**는 특성상 다른 **Thread**의 **Stack** 참조를 막을 수 없어 보안적 측면에서는 취약하다는 것도 알 수 있다.
 - *Process는 IPC라는 수단을 거쳐야하기 때문에 보안적 측면에서 우수하다.*

이론적 설명은 앞선 'Thread의 이해'에서 충분히 진행했으므로, 바로 Code-Level Analysis를 해보자. 프로세스 기반 서버 코드와 매우 유사하기 때문에 어렵지 않게 이해할 수 있다.

```
/* Routine of Peer Thread */
void *thread(void *vargp);                                     // 후술

/* Thread-based Concurrent Server */
int main(int argc, char **argv) {
    int listenfd, *connfdp;
    pthread_t tid;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;

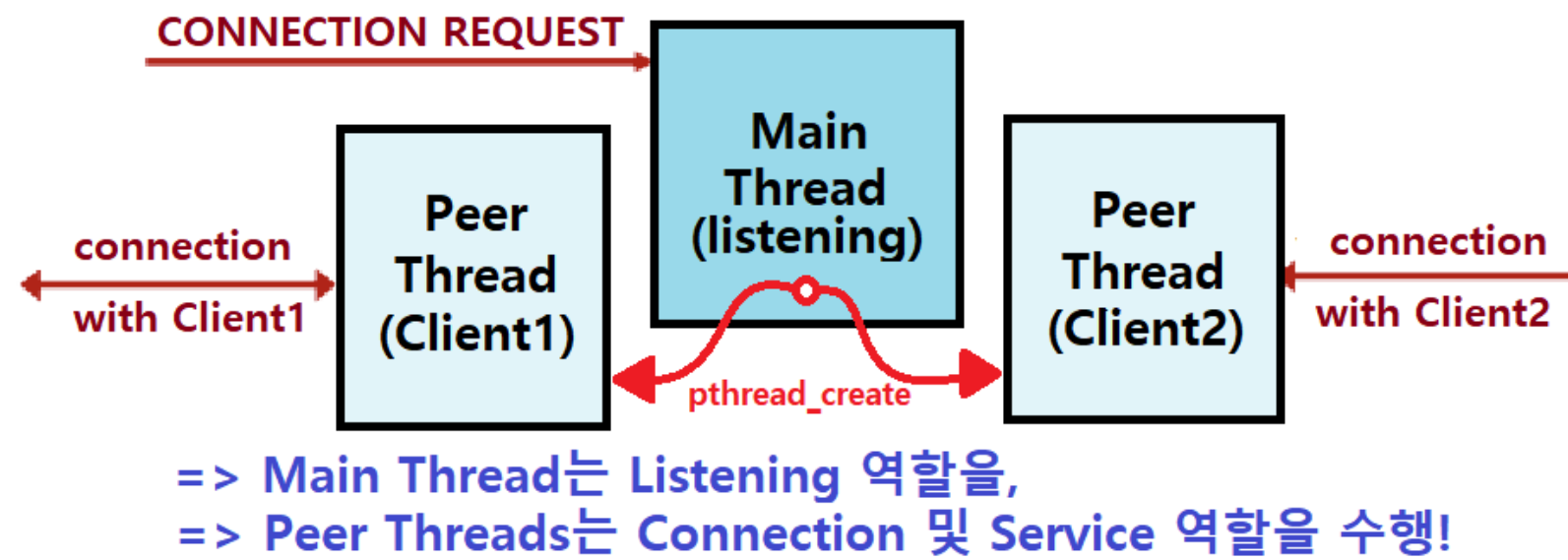
    listenfd = Open_listenfd(argv[1]);                          // listen까지의 작업 수행
    while (1) {
        clientlen=sizeof(struct sockaddr_storage);              // 늘 말하듯이 중요한 과정
        connfdp = Malloc(sizeof(int));
        *connfdp = Accept(listenfd, (SA *) &clientaddr, &clientlen); // Accept!
        Pthread_create(&tid, NULL, thread, connfdp);           // Peer Thread로 만들자!
```

```
}  
}
```

- **connfd**라는 포인터변수를 만든다. Main Thread의 Stack 공간에 포인터변수로서 존재한다. Heap 공간에 있는 Word Size(ex. 4Bytes) 공간을 가리킨다.
- Main Thread가 **pthread_create**를 하면, Peer Thread에게 이 포인터변수가 가리키는 Heap 공간을 넘긴다.
 - 그래서, **Process-based**에서는 **Non-Pointer**를 사용해서 **Close**하던 것과 다르게, 따로 **Close**하지는 않는다. ★★★
 - **listenfd** 자체는 닫을 필요도 없다. 왜냐면, 별도의 공간에 있는 것이 아니니까. ★
 - 물론, 후술할 것이지만, 이 **malloc** 방식은 좋지 않은 방식이다. 일단은 이어가자.

```
/* Routine of Peer Thread */  
void *thread(void *vargp) {  
    int connfd = *((int *)vargp);          // 넘겨받은 Heap 공간값을 connfd가 가리킨다.  
    Pthread_detach(pthread_self());        // 아래에서 설명할 것! (Reaping 관련)  
    Free(vargp);                          // 값을 추출했으므로, 힙공간은 해제하자.  
    echo(connfd);                          // connfd에 대해서 Service를 제공한다!  
    Close(connfd);                         // 서비스 끝나면 디스크립터를 닫자!  
    return NULL;                          // pthread_create 함수에게 NULL을 넘김(관습)  
}
```

- **pthread_detach** 함수
 - 다른 Thread와 상관없이 독립적으로 수행된다. ★★
 - **OS Kernel이 TID에 해당하는 Thread를 알아서 Reaping**해주도록 설정하는 역할 ★★★



- 각 Client는 개별의 Peer Thread에 의해 핸들링된다.
 - 100개의 Client가 Connection Request를 보내면, 100개의 Thread가 생성된다.
- 각 Thread는 TID를 제외한 모든 Process(Main Thread) State를 공유한다. ★
- Thread-based Server 구현 시 주의점1
 - 반드시 *pthread_detach* 함수를 호출해서 Thread Reaping을 진행해야한다.
 - Memory Leakage를 방지해야한다.

앞서, *pthread_join*이 Process 관점에서의 wait 역할이라 하지 않았나? 근데 왜 *pthread_detach*를 사용하는가?

*pthread_join*은 Main Thread 뿐만 아니라, Peer Thread에서도 언제 어디서든 호출하여 다른 Thread를 Reaping할 수 있다. (자유 접근 문제)

반면, *pthread_detach*를 호출하면, 다른 Peer Thread에서는 해당 Thread를 건들지 못하면서 동시에 OS Kernel이 알아서 Reaping해준다. ★

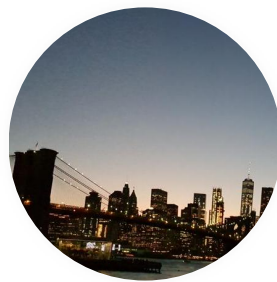
- **Peer Thread** 생성 시 기본 Default는 'Joinable Thread(pthread_join으로 Reaping해야하는)'이다.
 - 프로그래머가 명시적으로 pthread_detach를 심어서 'Detached Thread(OS Kernel이 알아서 Reaping해주는)'로 바꿔주어야 한다. ★★★
- **Thread-based Server 구현 시 주의점2**
 - **'Unintended Sharing'** 문제도 주의해야한다.
 - Thread Programming의 어찌보면 가장 큰 위험이다. Thread끼리 하나의 Heap/Data/Code 등의 영역을 공유하면, **의도치 않은 데이터 공유 및 Corruption** 문제가 발생할 수 있다.
 - 대표적이 예시가 바로 위의 예시 서버 코드이다. Pthread_create함수에 conncfd라는 포인터 변수를 넘기고 있는데, 만약, **Context Switch**가 일어나고, 다른 **Thread가 Accept**해버리면, 매우 심각한 문제가 발생한다. 서비스 제공이 이상하게 돌아갈 수 있는 것이다.
 - **OS Scheduling, Context Switch에 의한 Race**가 발생한 것이다. ★★★
 - 이러한 문제들을 해결하기 위해, 과거 Process를 다룰 때 'Async-Signal-Safety'가 중요했던 것처럼, **'Thread-Safe'**한 함수들을 사용해야한다. 이를 다음 포스팅에서 소개하겠다.

Pros and Cons

- **장점**
 - **Thread**끼리 자료구조의 공유가 용이하다. **IPC**같은 것이 필요없으므로!
 - **Process-based**에 비해 확실히 성능이 우수하다. (Efficient)
- **단점**
 - 자료구조의 공유가 반대로 매우 위험하게 작용할 수 있다. (장점과 단점을 모두 가짐)

- 어떤 데이터가 공유되는지 알기 어렵다.
- **Corruption** 일 발생했을 때 알아차리기가 어렵다.
- **Race**가 매우 미묘하고, 가끔 발생하기 때문에 알아차리기가 어렵다.

금일 포스팅은 여기까지이다. Concurrent Programming 개념의 막바지에 다다르고 있다. 어서 더 열심히 공부해보자.



hyeok's Log

팔로우



이전 포스트

SP - 4.2 Process / Event-base...

다음 포스트

SP - 5.1 Thread Programming ...



0개의 댓글

댓글을 작성하세요

댓글 작성

