멀티테스킹, 동시성: 여러 작업이 동시에 진행되는 것처럼 보임
멀티프로세스, 병렬성: 여러 작업이 실제로 동시에 진행됨

Concurrency

- 동시 실행

시스템에서 logical control flow가 동시에 하나 이상 발생했을 때를 말한다.
하나의 프로세스 내에서 발생하며 **multitasking**이라고 한다.

여러 작업이 동시에 실행되는 것처럼 보이지만 하나의 CPU에서 여러가지 작업이 번갈아 실행되는 것이다.

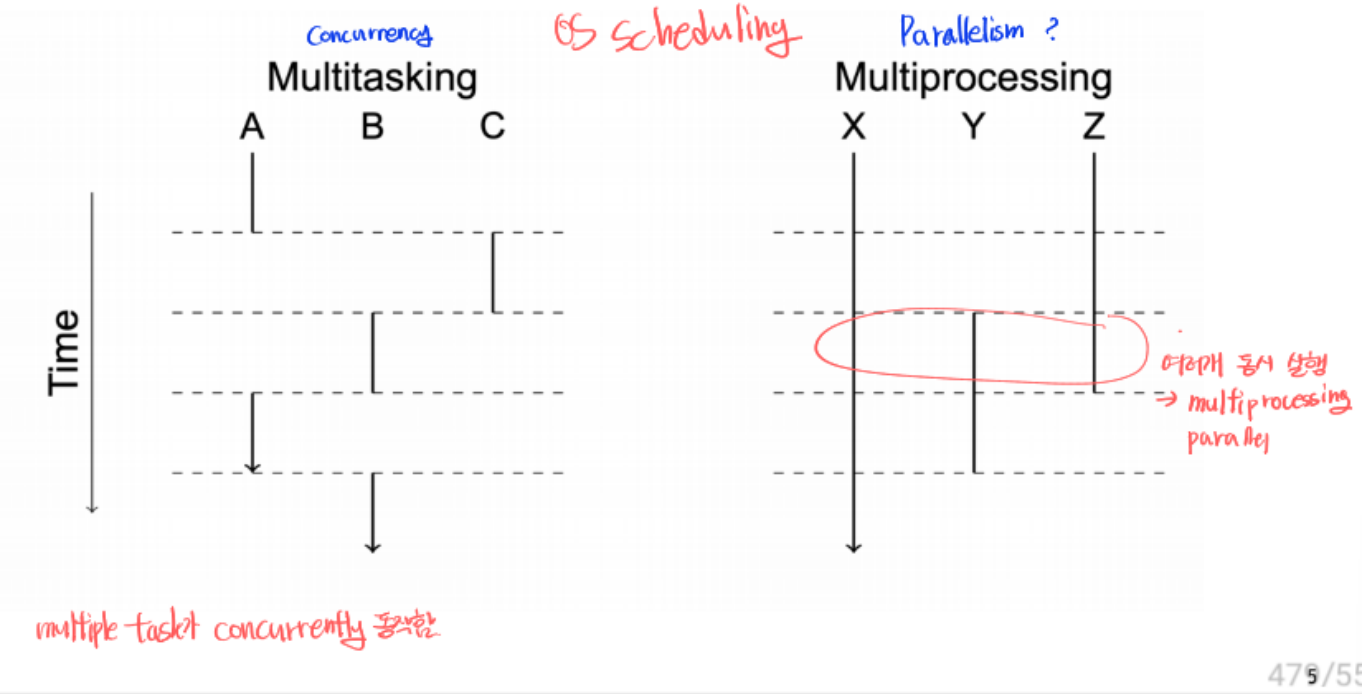
- multitasking 범위 내 동시 실행

Process

- 모든 프로세스는 자신이 메모리를 독점하고 있다고 생각하기 때문에 OS 스케줄링이 필요하다.
- process간 통신은 signal/pipe등을 이용해 사용한다. (IPC) -> 오버헤드 큼

Thread

- 한 프로세스에서 여러 스레드를 발생시킬 수 있다.
- 프로세스 메모리를 공유한다.
- MultiProcess보다 MultiThread의 오버헤드가 적다.



Normal func call vs Thread func call

- Normal 호출부에서 jmp 후 함수처리, 스택의 PC값으로 호출시점으로 돌아옴
  --> logical flow 하나

- Thread 호출부에서 thread를 생성해 함수처리하면서 호출부의 다음코드 실행
  --> main thread, func thread로 logical flow 두개

Multitasking

동시에 실행되는 것처럼 보이게 하기 위해 context switch로 스레드 왔다갔다함

1. L인 PC값에서 **context switch**가 발생하면 CPU에서 A를 지우고 B로 넘어간다
2. B 실행한다
3. context switch 발생하면 B에서 A로 넘어간다
   --> Thread를 이용하면 시스템 오버헤드가 적게 context switch가 가능하다.

Concurrency and Separation

coucurrent flow는 서로 연관될수도 연관되지 않을 수도 있다.
desing, inplementation, memory space(thread share), timing.. 등 이것들과 연관될 수도 안 될수도

coucurrent flow가 **연관되면 될수록 복잡**해진다.

- 독립적이며 연관되지 않은 일..
  코딩하고 음악듣는건 완전히 다른 일.

- 독립적이지만 연관된 일
  pipe, char client/server 등..
  연관되게 디자인되진 않았더라도 비슷한 일을 처리한다.

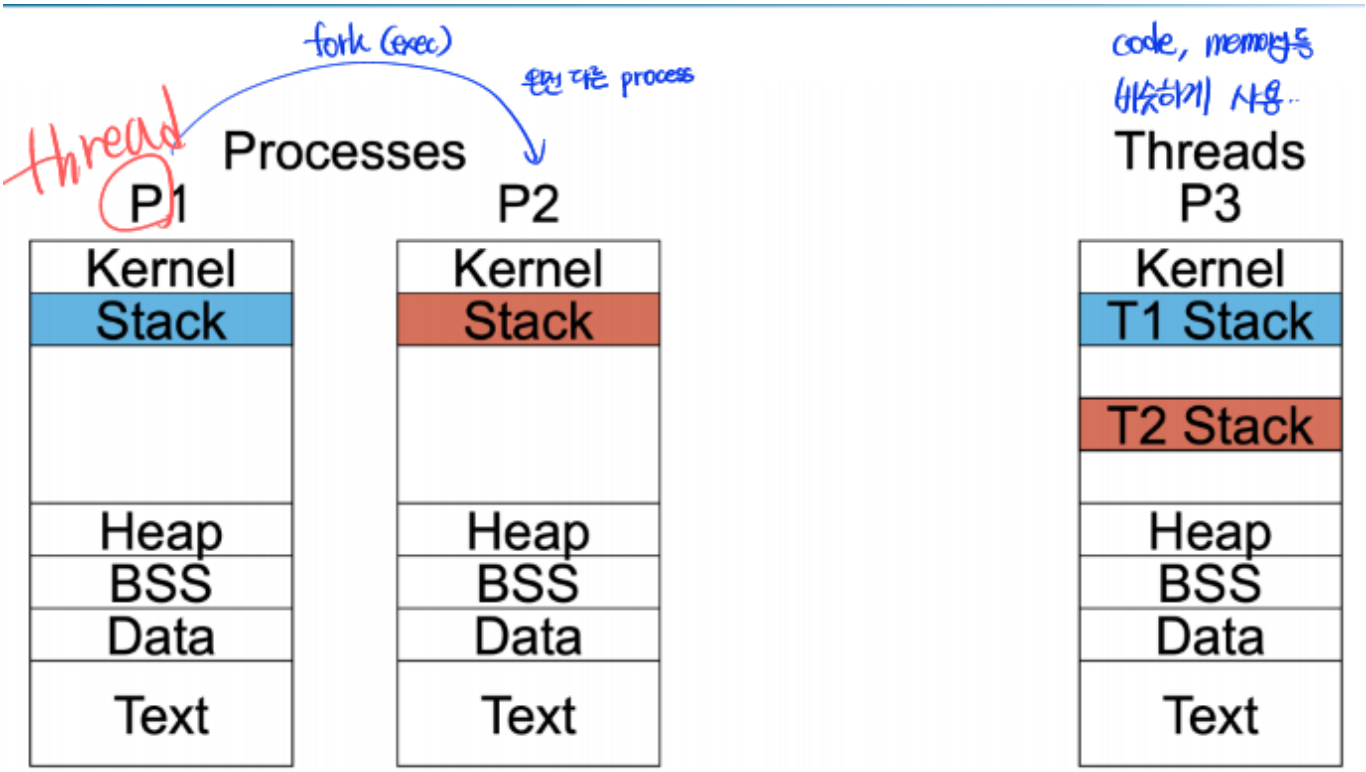
> 독립적인 메모리는 sys 보호, 옆집 서버 다운으로부터 안전한 등 여러 영향에 대해 보호해준다.
> 하지만 독립적인 메모리 사이의 연관(통신)은 느리다.
> 왜? -> syscall을 사용하는 IPC를 주로 이용하기 때문이다.

--> 메모리 낭비하며 느릴수도..
Communication and cooperation can be expensive, though: Many IPC mechanisms require system calls

Thread의 특징

code, data, files를 공유한다.
register, stack, counter(PC)는 thread별로 생성된다.

Process	**Thread**
독립적이다	프로세스의 한 부분이다
분리된 주소를 가진다	주소 공간을 공유한다
멀티 프로세스 IPC(signal, pipe)로 통신한다	공유자원이 존재한다



장점

- memory share를 통해 훨씬 싸다 (memory map, kernel, context switch)
- inter-thread 커뮤니케이션은 공유자원으로 하면 된다. (pipe 불필요)

단점

- 공유자원이 번거롭다 (동기화 필요)
- 대부분의 API는 not thread-safe하다.

사용에 적합한 경우

1. 작업간 빠른 control change가 필요한 경우
2. shared data가 큰 경우
3. 단일 CPU에서 빠른 실행을 원한다면
4. blocking operation that do not inhibit other progress
   한 스레드 block해도 다른 스레드에 영향을 미치지 않는다.

IPC

스레드에서 IPC(signal, pipe, semaphore)등 사용할 수 있지만 권장하지 않음.
다른 API가 많이 필요하기 때문에..
kill() vs pthread_kill() -> 둘 다 signal 보내는 함수이다.

Pthread

멀티 스레드를 위한 라이브러리

왜?

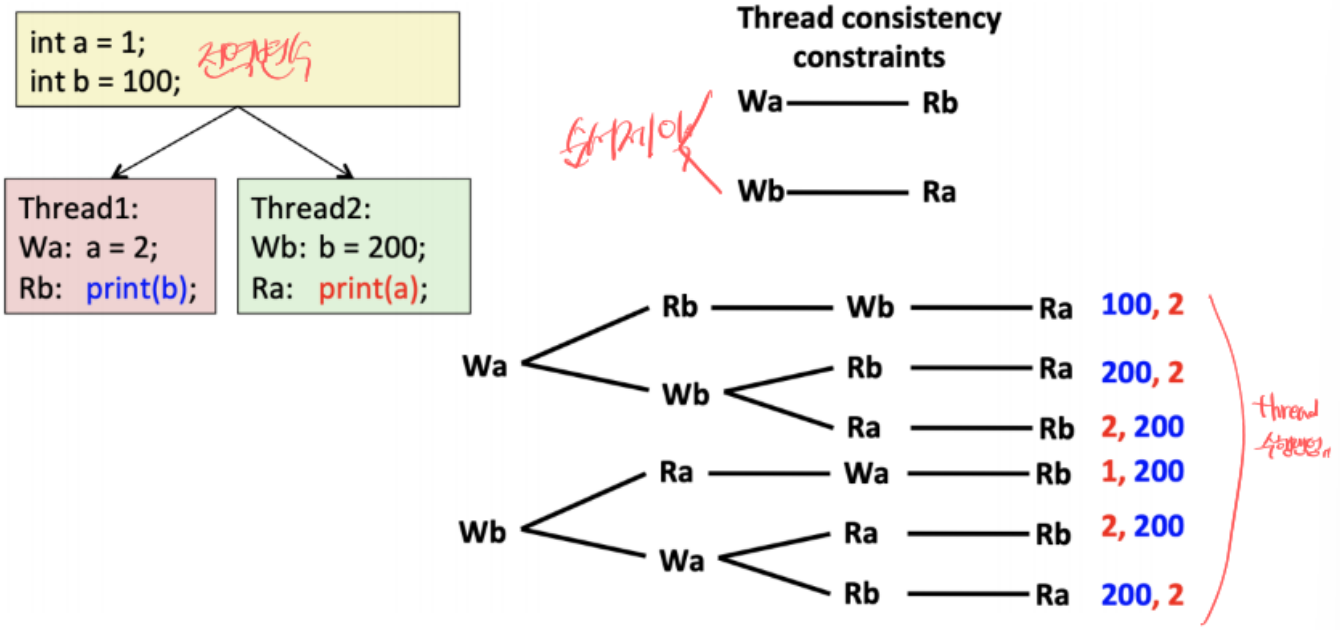
1. OS 오버헤드가 적다
2. 멀티프로세스를 관리하는 것보다 필요한 시스템이 적다
3. 모든 스레드는 하나의 프로세스와 같은 주소 공간을 공유한다.
   하나의 프로세스에서 여러 스레드가 실행되는 형태이기 때문이다.
4. IPC보다 inter-thread 통신이 쉽고 빠르다

fork보다 pthread_create가 빠르다.

- pthread_create (=fork)
  (thread id, arg, func, farg) 스레드 생성하며 특정 함수를 수행하고 아규먼트까지 넣을 수 있다.
- pthread_join (=wait)
  (thread id, void** thread_return) 스레드 실행 끝날때까지 기다린다. (다음으로 못 넘어가는 건 아님)
- pthread_detach()
  리턴값 없이 끝내기
- 컴파일시 -pthread 옵션 필요함

Problems on thread scheduling

스레드별 stack, register, PC값을 갖고있기 때문에 동시 실행시 공유자원에 랜덤으로 접근한다. 서로 공유자원 변경사항이 반영 안될 수도 있다.
--> 순서 제어가 필요하다..



Code

▶ pthread create.c

```c
void *mythread(void *arg) {
    printf("%s", (char *) arg);
    pthread_exit(0); // or return NULL;
}


int main(int argc, char *argv[]) {
    pthread_t p[10];
    printf("Expected: SYSTEMPROG\n");
    printf("Result  : ");
    // Create thread
    pthread_create(&p[0], NULL, mythread, "S");
    pthread_create(&p[1], NULL, mythread, "Y");
    pthread_create(&p[2], NULL, mythread, "S");
    pthread_create(&p[3], NULL, mythread, "T");
    pthread_create(&p[4], NULL, mythread, "E");
    pthread_create(&p[5], NULL, mythread, "M");
    pthread_create(&p[6], NULL, mythread, "P");
    pthread_create(&p[7], NULL, mythread, "R");
    pthread_create(&p[8], NULL, mythread, "O");
    pthread_create(&p[9], NULL, mythread, "G");
    // Wait for threads to exit
    pthread_join(p[0],NULL);
    pthread_join(p[1],NULL);
    pthread_join(p[2],NULL);
    pthread_join(p[3],NULL);
```

input

```
Expected: SYSTEMPROG
Result  : EGSMRYOPTS
```

스레드 실행 순서는 스레드 생성 순서와 관련 없다. (랜덤)

▶ with shared data.c

```c
void* print(void* arg) {
    while (i<100){
        printf("%s", (char*)arg);
        sum+=atoi(arg);
        i++;
        usleep(1);
    }
}

int main(int argc, char** argv) {

    pthread_t thd1, thd2;

    //0과 1이 번갈아 실행되어 sum=50을 기대함ㄴ
    printf("Expected: ");
    for (int a=0; a<50; a++)
        printf("01");
    printf("\n");

    printf("Result  : ");
    pthread_create(&thd1, NULL, print, "0");
    pthread_create(&thd2, NULL, print, "1");

    void* result;
    pthread_join(thd1, &result);
    pthread_join(thd2, &result);
```

input

```
      |         ^~~~~
Expected: 01010101010101010101010101010101010101010101010101010101010101010101010101010101
Result  : 11111111111111111111000000000000000000000000000000000000000000000000000000000000
sum = 22
```

▶ with shared data 2.c

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

static volatile int counter = 0;

void *mythread(void *arg) {
    printf("%s: begin with counter %d\n", (char *) arg, counter);
    int i;
    for (i = 0; i < 1000000; i++){
        counter += 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    printf("Counter = %d\n", counter); //0
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");

    pthread_join(p1,NULL);
    pthread_join(p2,NULL);

    printf("Counter = %d\n",counter);
    return 0;
}


```



```
B: begin with counter 0
A: begin with counter 122646
B: done
A: done
Counter = 1329956
```

B 조금 실행된 후 A가 시작된다.
2000000 이하 값이 나온다. (카운터 ++ 씹히기 때문에)

공유자원에 대한 thread 접근은 랜덤이기 때문에 공유자원 결과 예측하지 못한다.

Summary

1. Coucurrent flow는 같이 동작되는 것처럼 보인다. (OS Scheduling)
2. Multitasking: 한 프로세스에서 concurrent flow를 switch하는 것
   동시실행처럼 보이기 위해!
3. Councurrency는 쉽고, 빠르고, 간단하지만 동기화 이슈가 발생한다.
   공유자원 접근에 대한 동기화 이슈
4. 각 프로레스와 스레드는 coucurrent flow가 여러 작업들이 동시실행되는것처럼 보이게 하여 추상화를 제공한다. (프로그래머에게 편의성)

5. 스레드는 프로세스보다 싸지만 덜 독립적이다.
6. 스레드에서 공유자원 이용시 tricky하다.
7. Coucurrenct: 여러 logical flow가 하나의 프로세스 위에서 자원 공유하며 실행된다.