운영체제
Lecture 09: 쓰레드

전남대학교 인공지능학부

박태준 (taejune.park@jnu.ac.kr)

Review the last lecture

- **프로세스 Process**
  - 실행파일(Program)이 메모리에 올라가 작동되는 상태!
  - OS는 Process Control Block (PCB)를 통해 프로세스를 관리
    - 프로세스의 생애 주기: New → ready → running → (blocked) → terminated
- **프로세스 메모리 구조:** 메모리에 그냥 올리는게 아니다!
  - 프로세스 가상 메모리: 각 프로세스는 마치 메모리를 혼자 독점하는 것 처럼 보인다!
  - Code, data, heap, stack 영역
- **프로세스 생성과 복사:** 프로세스가 매번 맨땅에 만들어지는건 아니에요!
  - fork() → exec()
  - wait(), exit()
- **프로세스 계층 구조:** 부모와 자식 관계 → 조상님이 계십니다!
  - 좀비 프로세스: 부모가 자식의 종료를 확인하지 않아 → 작업 종료 후 PCB가 남은 상태
  - 고아 프로세스: 부모가 먼저 죽은 상태 → init으로 입양!

Goal

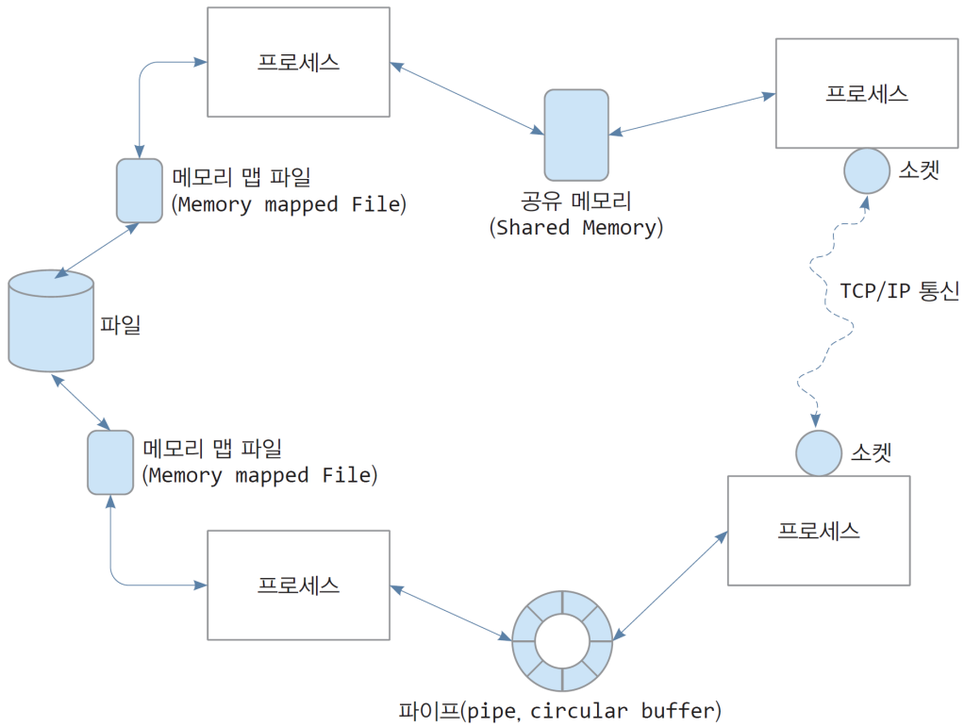
- 프로세스의 문제점과 쓰레드의 필요성!
  - 프로세스는 너무 무거워요!
  - 오늘날 운영체제의 실행 단위는 쓰레드!
  - 쓰레드 vs 프로세스 차이의 이해
- 쓰레드 관리 기법
  - 쓰레드 주소 공간
  - 쓰레드 생명 주기
  - TCB, Thread Control block
- Multi-threading
  - Kernel-level thread vs User-level thread

쓰레드 개요

프로세스의 문제점!

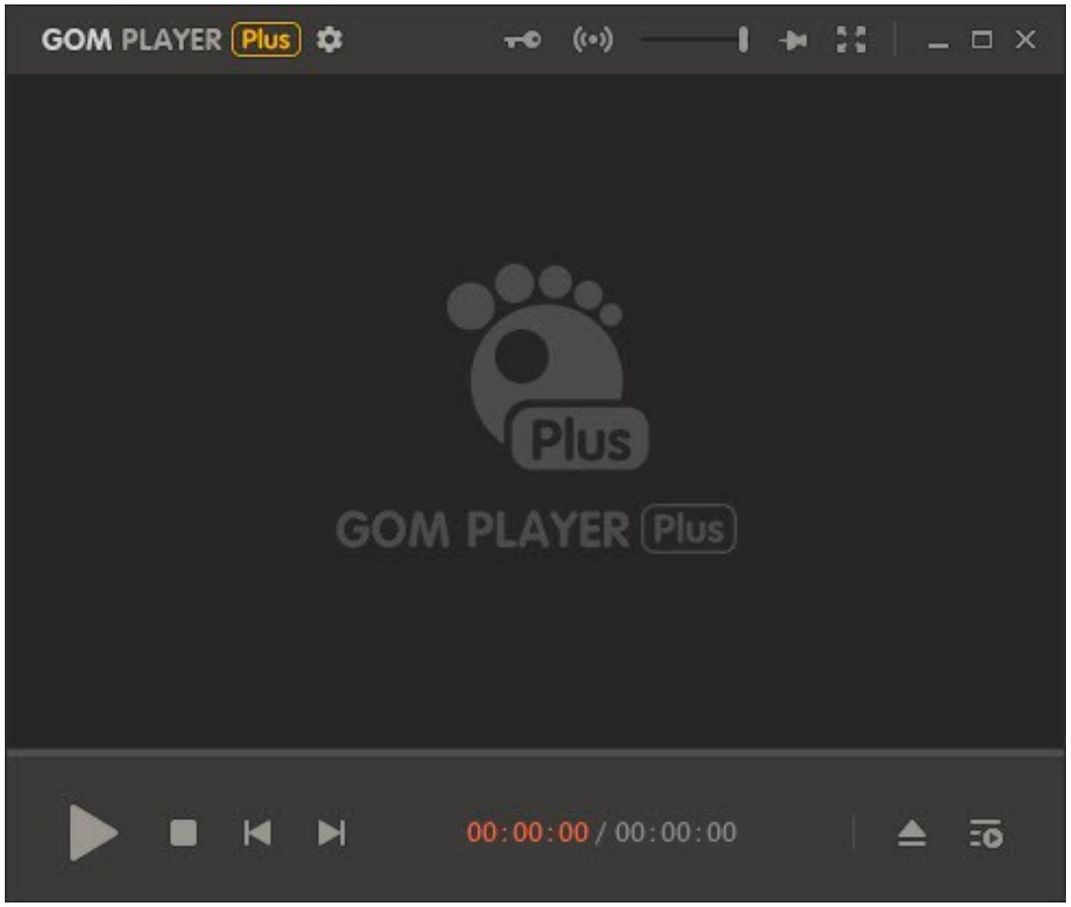
- 프로세스 생성의 오버헤드가 큼
  - 메모리 할당 → fork() → PCB → Page(segment) mapping table → ...
- Process context switching 의 오버헤드가 큼
  - 기존에 처리하던 레지스터 정보들 저장, 처리 영역 저장, 새로운 컨텍스트를 불러오고...
  - 새로 메모리에서 레지스터로 값을 가져오고, 처리 위치 가져오고...
- 프로세스간의 통신이 어려움
  - 프로세스들은 완전히 독립적인 주소공간을 가지고 있음

    → 서로 다른 프로세스끼리 개입 불가
  - 프로세스간의 통신을 위해서 별도의 방법이 필요
    - Shared memory, socket, message queue 등



- → 하나의 작업을 가지고 여러 모듈(?) 단위로 쪼개서 일할때 참 문제더라...

예를 들어...



- 미디어 플레이어의 경우...
  - 영상 처리
  - 소리 처리
  - 자막 처리
  - 기타 등등...
- 저 기능들이 '동시에' 작동이 이루어져야한다!
  - 사실 진짜 동시는 아니다! → '시분할'
- 각 기능들이 프로세스 기반 '멀티태스킹' 이라면?
  - context switching 하기에는 하나하나가 너무 무겁다!
  - **시분할의 사이시간이 길어질 수 밖에 없다!**

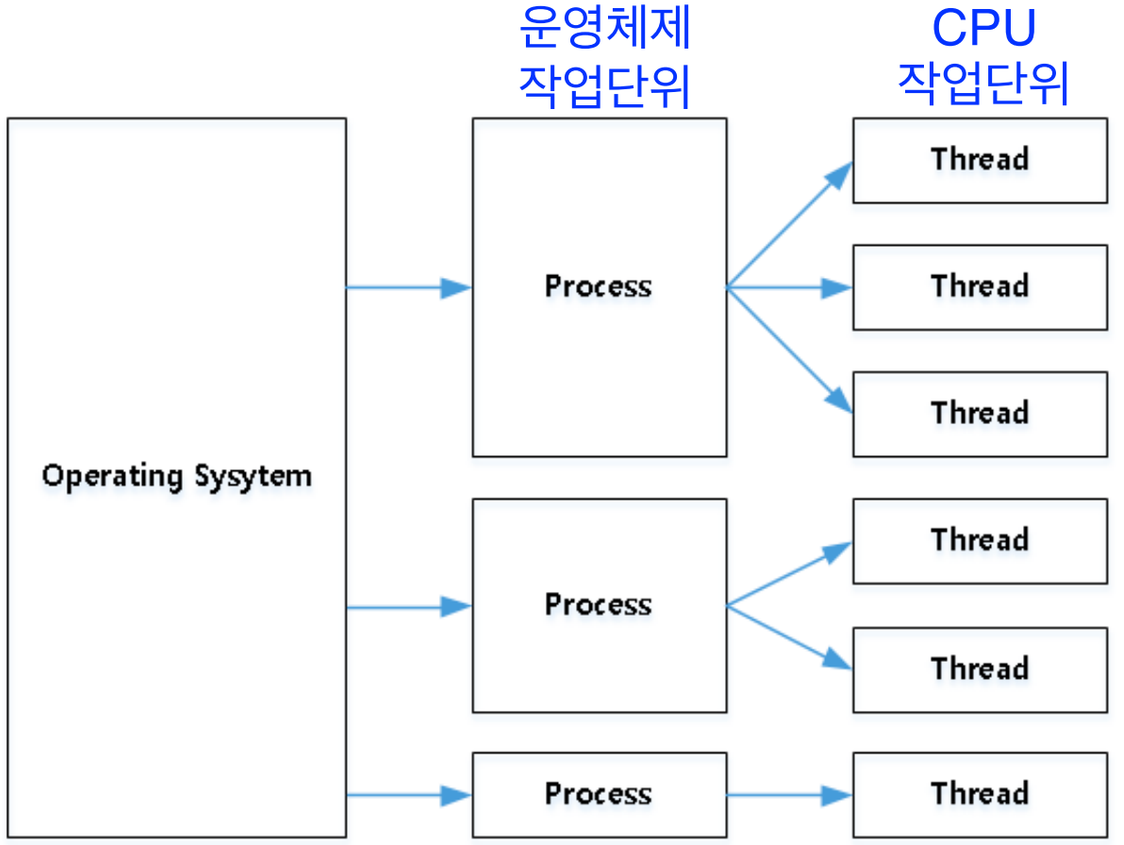
그래서 쓰레드가 등장!

- 프로세스를 사용하는 문제점을 해결하기 위해서 고안!

- 프로세스보다 더 '작은' 실행단위, 그리고 현대 운영체제가 작업을 '스케쥴링' 하는 단위
  - CPU 스케쥴러가 CPU에 작업을 전달하는 단위.
    - (c.f., 운영체제 입장에서 작업 관리 단위는 프로세스)
  - 쓰레드를 lightweight process라고도 부름

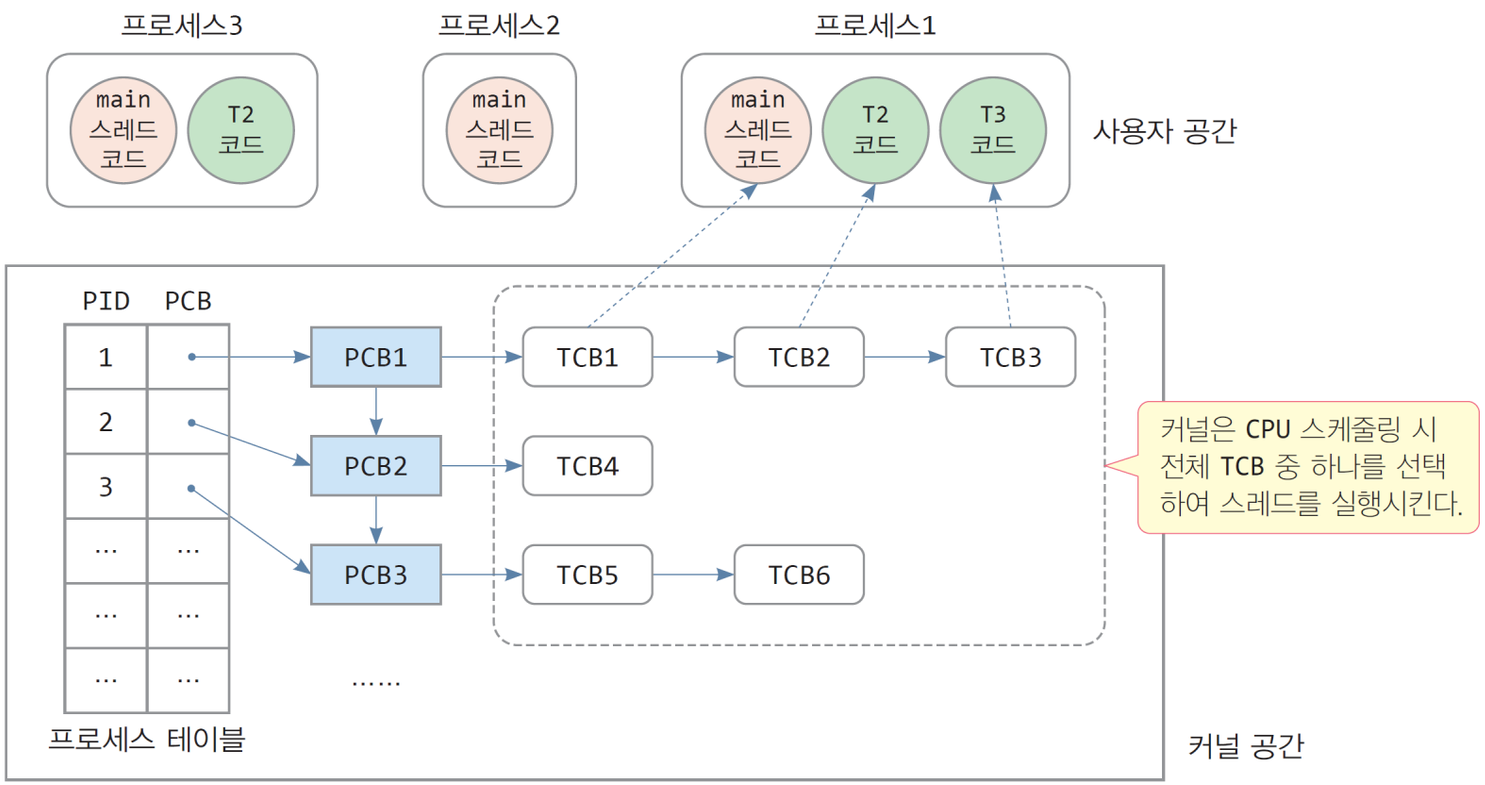
- 프로세스의 생성 및 소멸에 따른 오버헤드 감소

- 빠른 Context switching, 손쉬운 통신



프로세스는 쓰레드들의 컨테이너! (1)

- **쓰레드는 곧 함수! 프로세스는 반드시 1개 이상의 쓰레드로 구성**
  - 프로세스가 생성될 때 운영체제에 의해 자동으로 1개의 쓰레드 생성: **메인 쓰레드(main)**
  - 하나의 컨테이너가 여러개의 쓰레드를 가질 수도 있음! → **멀티쓰레드**
    - 다른 쓰레드들은 함수를 쓰레드로 만들어줄 것을 요청하여 생성됨
    - 그리고 각 쓰레드 별로 'TCB (Thread control block)'이 생성되고, 이 TCP는 PCB에 등록됨



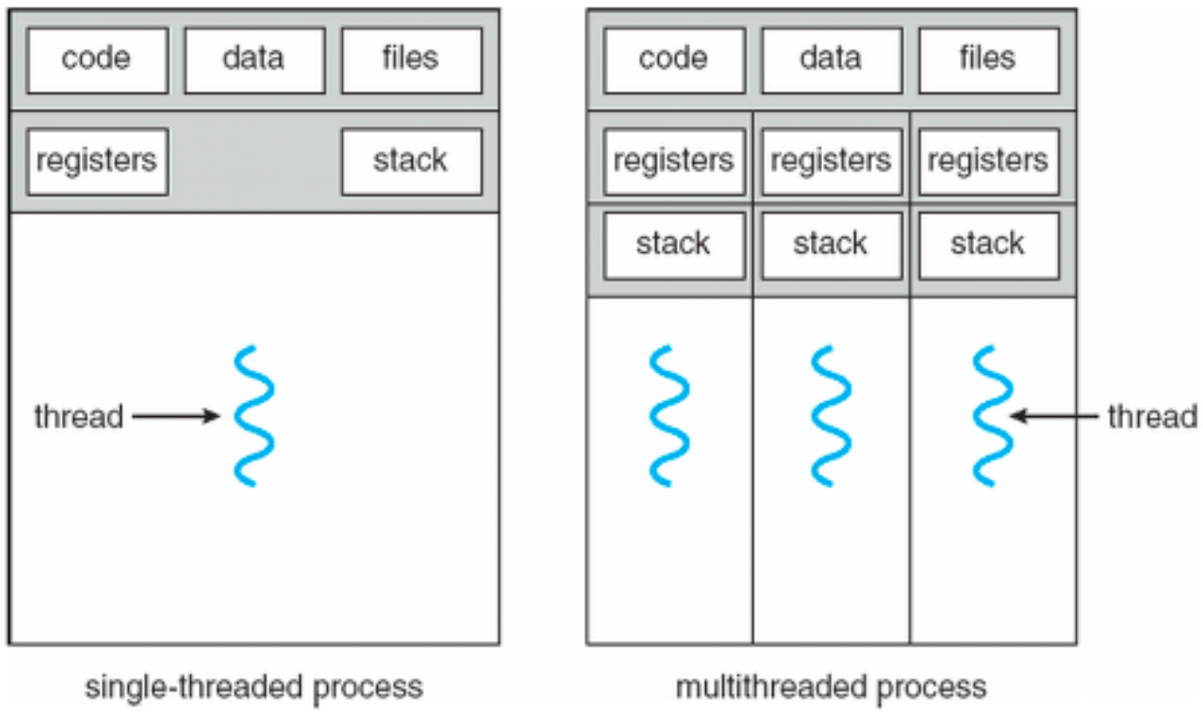
※ 스레드마다 TCB가 만들어지고 서로 연결된다. 프로세스에 속한 스레드들을 관리하기 위해 PCB는 TCB와 연결된다.

프로세스는 쓰레드들의 컨테이너! (2)

- **프로세스는 쓰레드들의 공유 공간(환경) 제공**
  - **모든 쓰레드는 프로세스의 코드, 데이터, 힙을 공유** → 쓰레드 사이 통신이 용이
  - 단, '스택'은 쓰레드별로 별도의 공간을 사용!

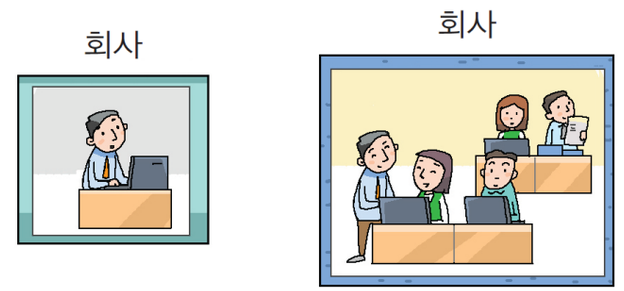
- **쓰레드의 생명과 프로세스의 생명**
  - 쓰레드로 만든 함수가 종료되면 쓰레드가 종료됨.
    - 쓰레드가 종료되면 TCB도 제거됨.
  - 프로세스에 속한 모든 쓰레드가 종료될 때, 프로세스 종료
  - 프로세스가 강제로 종료되면?

    → 쓰레드도 당연히 종료됨

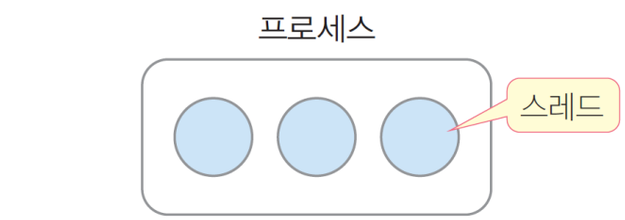


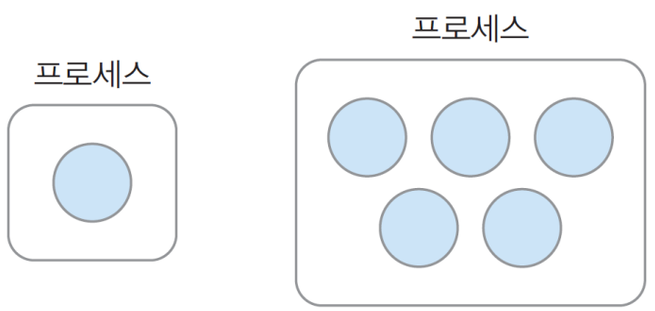
이해를 돕기 위한 비유: 은행과 창구

- 프로세스: 각각의 은행 지점
- 쓰레드: 각 은행 내부의 창구들



- **새로운 은행을 만드는 것은...**
  - 돈이 당연히 많이 든다!
  - 다른 지점과 연락도 쉽지 않다!
  - 하지만 독립적인 개체로서, 은행 업무에 대한 수행이 가능하다!
    자본도 있다!





- **은행 내부에 새로운 창구를 만드는 것은...**
  - 은행을 통채로 만드는 것과 비교해서는 말도안되게 저렴하다.
  - 이웃 창구와의 연락이 쉽다!
  - 독립적인 개체는 아니다! 혼자서는 존재하지 못한다! 자본이 없다!
  - 창구가 닫는다고 은행이 닫는건 아니다! 은행이 닫으면? 창구도 닫힌다!

Thread 예제

```
myThread1's tid: 2465000
myThread2's tid: 24E8000
          myThread 2 starts
          myThread 1 starts
myThreads have been finished
sum = -40164
ret1 = 200000
ret2 = 200000
```

```
myThread1's tid: 4AE0000
myThread2's tid: 4B63000
          myThread 1 starts
          myThread 2 starts
myThreads have been finished
sum = 32517
ret1 = 200000
ret2 = 200000
```

```c
#include <pthread.h> // pthread lib
#include <stdio.h>
#include <stdlib.h>

int sum = 0; // global variable

void* myThread1(void *p) { // for the thread 1
  printf("\t myThread 1 starts\n");

  int *i = (int*)malloc(sizeof(int));
  for(i = 0; i < (*(int*)p); i++) sum += 1;

  return (void*)i;
}

void* myThread2(void *p) { // for the thread 2
  printf("\t myThread 2 starts \n");

  int *i = (int*)malloc(sizeof(int));
  for(i = 0; i < (*(int*)p); i++) sum -= 1;

  return (void*)i;
}

int main() {
  pthread_t tid1, tid2; // thread id
  int count = 200000;
  int *ret1, *ret2;

  // create thread (id, attribute, function_pointer, argument)
  pthread_create(&tid1, NULL, myThread1, &count);
  printf("myThread1's tid: %0X \n", (int)tid1);

  pthread_create(&tid2, NULL, myThread2, &count);
  printf("myThread2's tid: %0X \n", (int)tid2);

  pthread_join(tid1, (void**)&ret1); // waiting for 'tid1'
  pthread_join(tid2, (void**)&ret2); // waiting for 'tid2'

  printf("myThreads have been finished \n");
  printf("sum = %d\n", sum);
  printf("ret1 = %d\n", (int)ret1);
  printf("ret2 = %d\n", (int)ret2);

  return 0;
}
```

생각해보아야 할 점

- 이래나 저래나 프로그램의 실행 단위는 함수! 쓰레드 단위는 결국 함수!

- 프로세스는 쓰레드들간 공유 자원 제공
  - Data영역은 확실히 공유됨 → 각 함수에 대한 전역변수이니 너무나도 당연하다면 당연
  - stack영역은 각자 별개!
    - 각 지역 변수의 값들은 이쁘게 누적됨!

- 쓰레드의 실행 순서를 알 수 없음.
  - 공유자원(전역변수)의 합이 0이 되질 않는다!
  - 하나의 자원을 여럿이 쓰려고 한다? → 다시 운영체제의 문제...!
    - 나중에 자세한 원인과 해결 방법들을 배웁니다 :)

(macOS or Linux only)

그대를 위한 복붙용 텍스트

→ 한번 테스트 해보세요!

```c
#include <pthread.h> // pthread lib
#include <stdio.h>
#include <stdlib.h>

int sum = 0; // global variable

void* myThread1(void *p) { // for the thread 1
  printf("\t myThread 1 starts\n");

  int *i = (int*)malloc(sizeof(int));
  for(i = 0; i < (*(int*)p); i++) sum += 1;

  return (void*)i;
}

void* myThread2(void *p) { // for the thread 2
  printf("\t myThread 2 starts \n");

  int *i = (int*)malloc(sizeof(int));
  for(i = 0; i < (*(int*)p); i++) sum -= 1;

  return (void*)i;
}

int main() {
  pthread_t tid1, tid2; // thread id
  int count = 200000;
  int *ret1, *ret2;

  // create thread (id, attribute, function_pointer, argument)
  pthread_create(&tid1, NULL, myThread1, &count);
  printf("myThread1's tid: %0X \n", (int)tid1);

  pthread_create(&tid2, NULL, myThread2, &count);
  printf("myThread2's tid: %0X \n", (int)tid2);

  pthread_join(tid1, (void**)&ret1); // waiting for 'tid1'
  pthread_join(tid2, (void**)&ret2); // waiting for 'tid2'

  printf("myThreads have been finished \n");
  printf("sum = %d\n", sum);
  printf("ret1 = %d\n", (int)ret1);
  printf("ret2 = %d\n", (int)ret2);

  return 0;
}
```

(For Windows user)

그대를 위한 복붙용 텍스트 → 한번 테스트 해보세요!
(풀 코드는 아닙니다! 직접 해보는 것도 좋을듯 합니다!)

```
#include <thread>  // thread lib.
#include <stdio.h>
#include <stdlib.h>

void func1() {
  for (int i = 0; i < 100; i++)
    printf("I am thread 1");
}
void func2() {
  for (int i = 0; i < 100; i++)
    printf("I am thread 2");
}
void func3() {
  for (int i = 0; i < 100; i++)
    printf("I am thread 3");
}

int main() {
  thread t1(func1);
  thread t2(func2);
  thread t3(func3);

  t1.join();
  t2.join();
  t3.join();
}
```

쓰레드 장점 및 단점

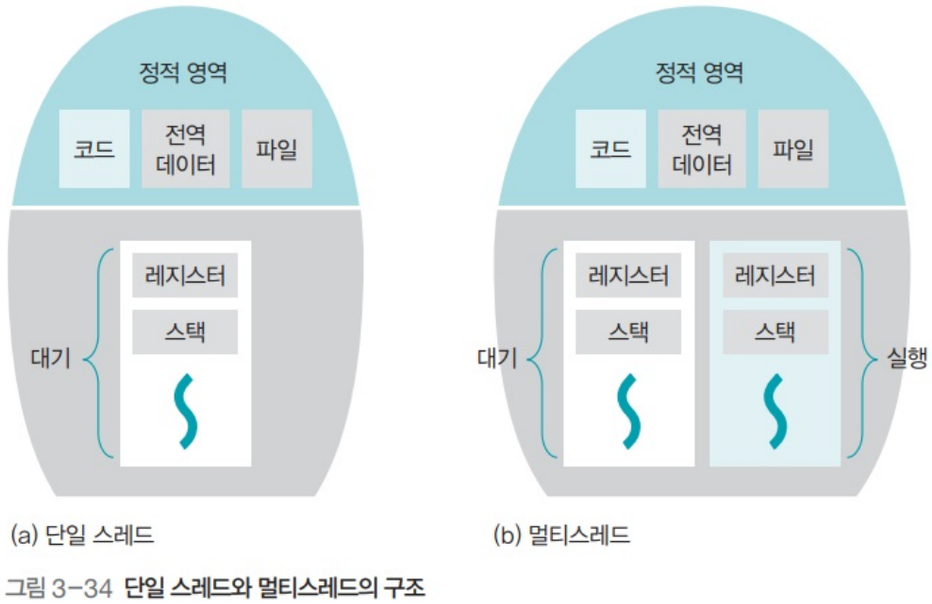
▸ 장점

- CPU 응답성 향성

  자원 공유, 효율성 향상,

  다중 CPU 운용 용이

- c.f., 멀티쓰레드 vs 멀티태스킹 vs 멀티프로세싱

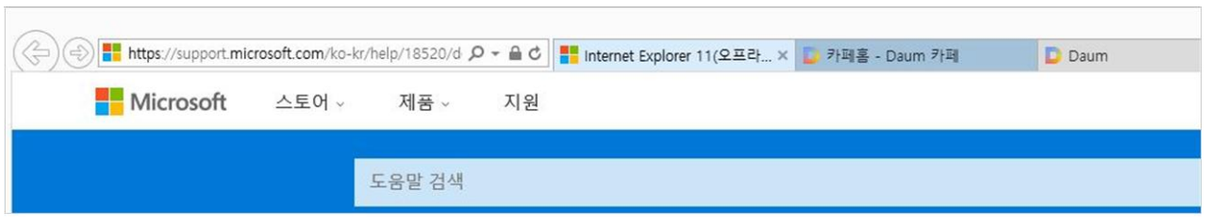


(a) 단일 스레드 (b) 멀티스레드

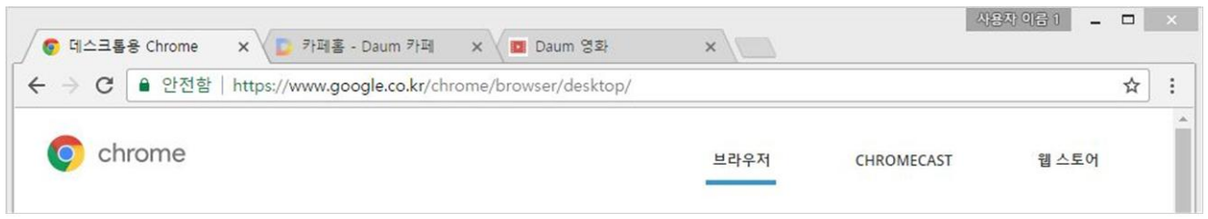
그림 3-34 **단일 스레드와 멀티스레드의 구조**

▸ 단점

- 모든 자원을 공유한다? → 하나의 쓰레드가 잘 못되면 프로세스 전체가 다 죽을 수 있음...!
- 너무 많은 쓰레드? → 너무 많은 Context switching!



IE



Chrome

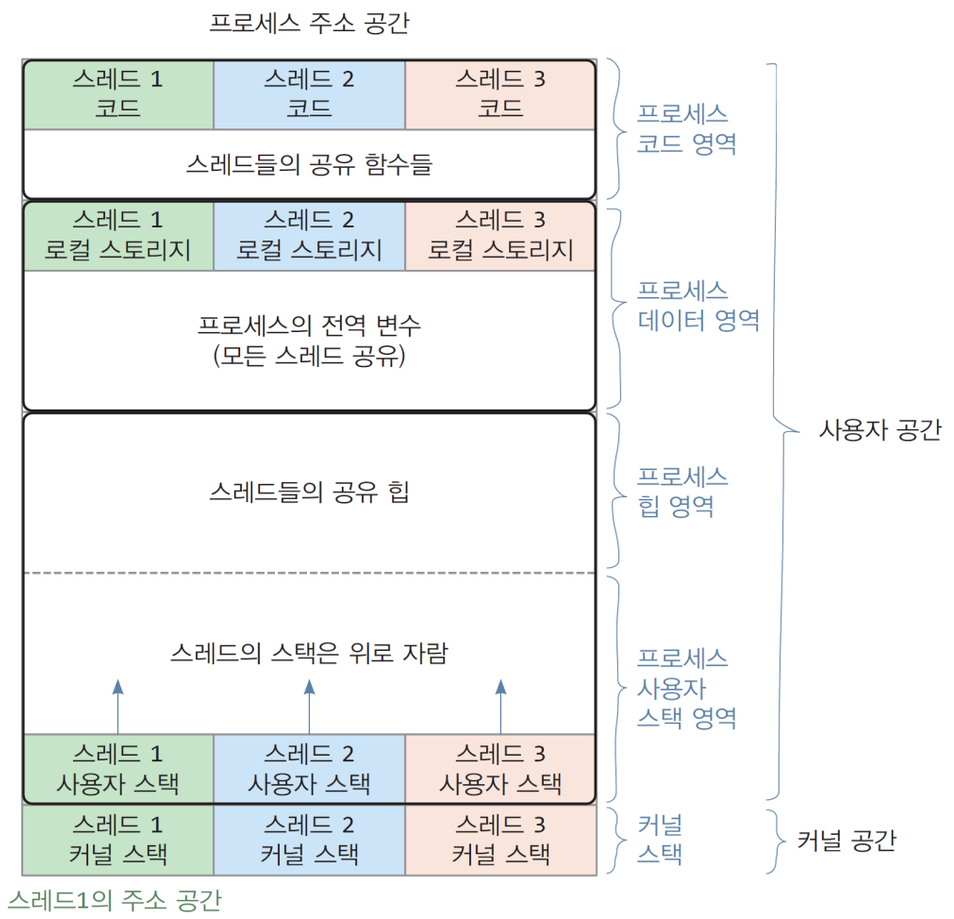
쓰레드 Deep dive

쓰레드 주소공간

- 쓰레드가 생성되고 실행되는 동안 접근 가능한 메모리 영역
- 쓰레드 주소 공간은 프로세스의 주소 공간 내에 형성
  - 일반 함수가 수직적인 관계라하면, 쓰레드는 side-by-side 형태

- **쓰레드 사적 공간**
  - **쓰레드 코드(Thread code)**
  - **쓰레드 로컬 스토리지 (TLS, Thread local storage)**
  - **쓰레드 스택**

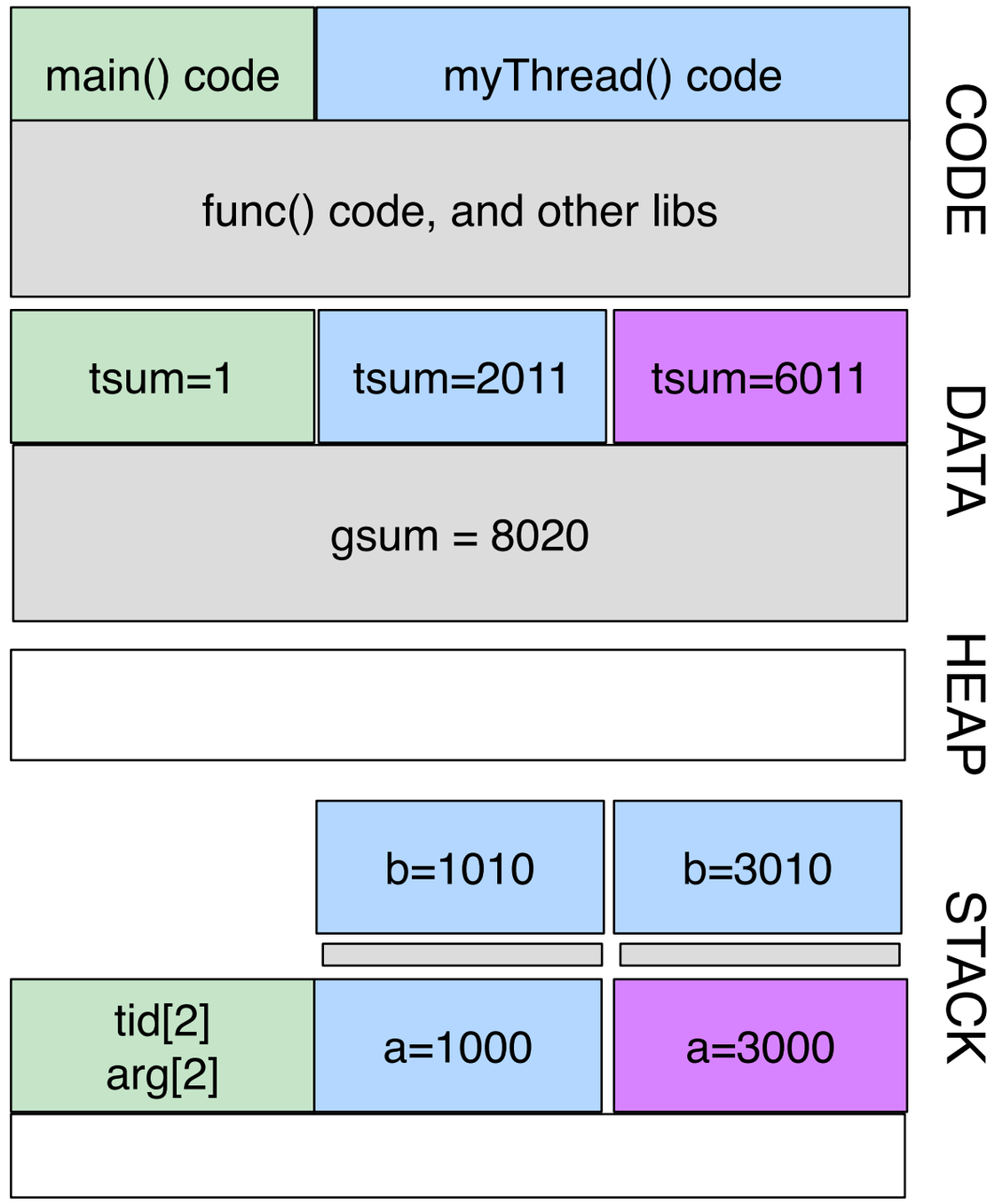
- **쓰레드 사이의 공유 공간**
  - 프로세스의 코드
  - 프로세스의 데이터 공간(로컬 스토리지 제외)
  - 프로세스의 힙 영역



※ 스레드의 주소 공간은 프로세스 주소 공간 내에 존재한다.

e.g., Thread memory layout and TLS and

```c
#include <pthread.h> // pthread lib
#include <stdio.h>
#include <stdlib.h>

int gsum = 0;
int __thread tsum = 1;

void func(int a) {
  printf("5_%d. gsum= %d / tsum= %d \n", a, gsum, tsum);
  int b = a + 10;
  gsum += b;
  tsum += b;
  printf("6_%d. gsum= %d / tsum= %d \n", a, gsum, tsum);
}

void* myThread(void *p) {
  int a = (*(int*)p);
  printf("2_%d. gsum= %d / tsum= %d \n", a, gsum, tsum);
  for(int i = 0; i < 30000000/a; i++);

  gsum += a;
  tsum += a;
  printf("3_%d. gsum= %d / tsum= %d \n", a, gsum, tsum);

  func(a);
  printf("7_%d. gsum= %d / tsum= %d \n", a, gsum, tsum);
}

int main() {
  pthread_t tid[2];
  int arg[2] = {1000, 3000} ;

  printf("1_main. gsum= %d / tsum= %d \n", gsum, tsum);
  pthread_create(&tid[0], NULL, myThread, &arg[0]);
  pthread_create(&tid[1], NULL, myThread, &arg[1]);
  pthread_join(tid[0], NULL);
  pthread_join(tid[1], NULL);
  printf("8_main. gsum= %d / tsum= %d \n", gsum, tsum);

  return 0;
}
```



(macOS or Linux only)

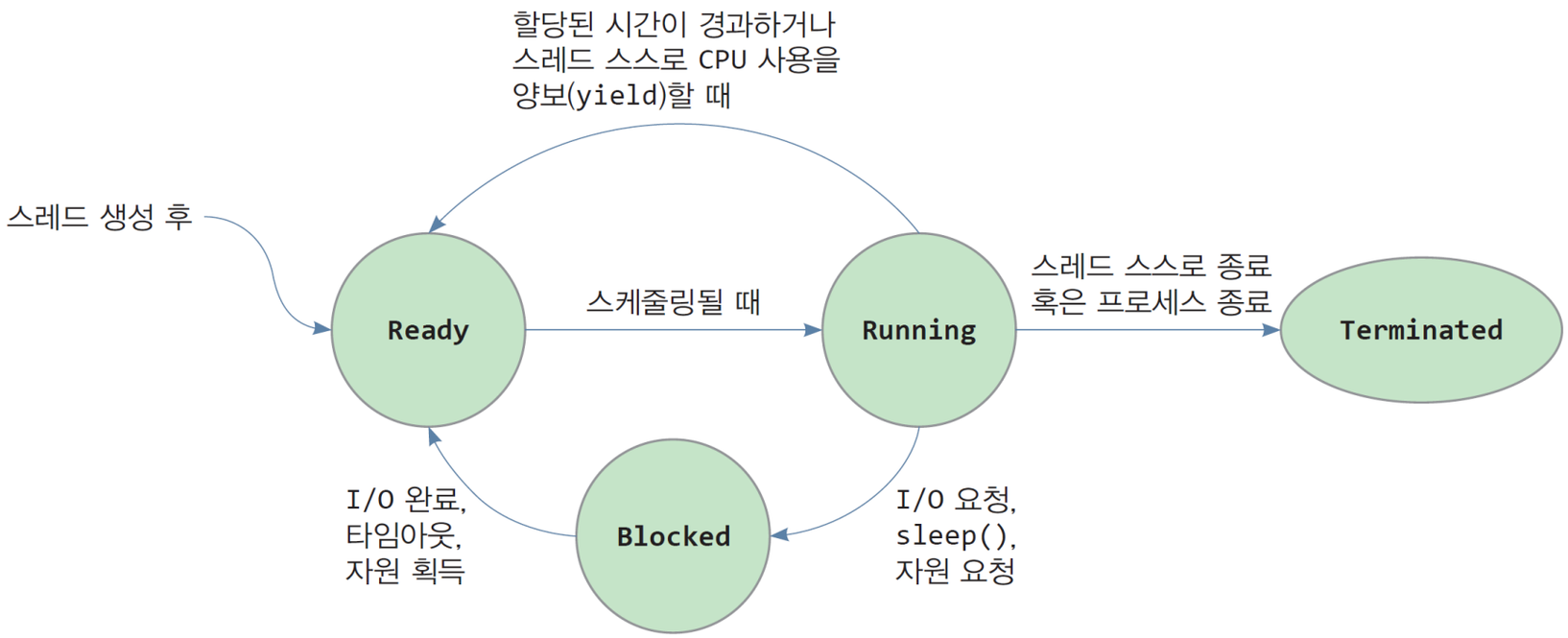
그대를 위한 복붙용 텍스트
→ 한번 테스트 해보세요!

```
1_main. gsum= 0 / tsum= 1
2_3000. gsum= 0 / tsum= 1
3_3000. gsum= 3000 / tsum= 3001
5_3000. gsum= 3000 / tsum= 3001
6_3000. gsum= 6010 / tsum= 6011
7_3000. gsum= 6010 / tsum= 6011
2_1000. gsum= 6010 / tsum= 1
3_1000. gsum= 7010 / tsum= 1001
5_1000. gsum= 7010 / tsum= 1001
6_1000. gsum= 8020 / tsum= 2011
7_1000. gsum= 8020 / tsum= 2011
8_main. gsum= 8020 / tsum= 1
```

```c
#include <pthread.h> // pthread lib
#include <stdio.h>
#include <stdlib.h>

int gsum = 0;
int __thread tsum = 1;

void func(int a) {
  printf("5_%d. gsum= %d / tsum= %d \n", a, gsum, tsum);
  int b = a + 10;
  gsum += b;
  tsum += b;
  printf("6_%d. gsum= %d / tsum= %d \n", a, gsum, tsum);
}

void* myThread(void *p) {
  int a = (*(int*)p);
  printf("2_%d. gsum= %d / tsum= %d \n", a, gsum, tsum);
  for(int i = 0; i < 30000000/a; i++);

  gsum += a;
  tsum += a;
  printf("3_%d. gsum= %d / tsum= %d \n", a, gsum, tsum);

  func(a);
  printf("7_%d. gsum= %d / tsum= %d \n", a, gsum, tsum);
}

int main() {
  pthread_t tid[2];
  int arg[2] = {1000, 3000} ;

  printf("1_main. gsum= %d / tsum= %d \n", gsum, tsum);
  pthread_create(&tid[0], NULL, myThread, &arg[0]);
  pthread_create(&tid[1], NULL, myThread, &arg[1]);
  pthread_join(tid[0], NULL);
  pthread_join(tid[1], NULL);
  printf("8_main. gsum= %d / tsum= %d \n", gsum, tsum);

  return 0;
}
```

쓰레드 라이프 사이클

- 프로세스 라이프 사이클과 거의 흡사! → TCB로 관리함
- 쓰레드 상태 변화
  - 준비 상태(Ready) - 쓰레드가 스케줄 되기를 기다리는 상태
  - 실행 상태(Running) - 쓰레드가 CPU에 의해 실행 중인 상태
  - 대기 상태(Blocked) - 쓰레드가 입출력을 요청하거나 sleep()과 같은 시스템 호출로 인해 커널에 의해 중단된 상태
  - 종료 상태(Terminated) - 쓰레드가 종료한 상태

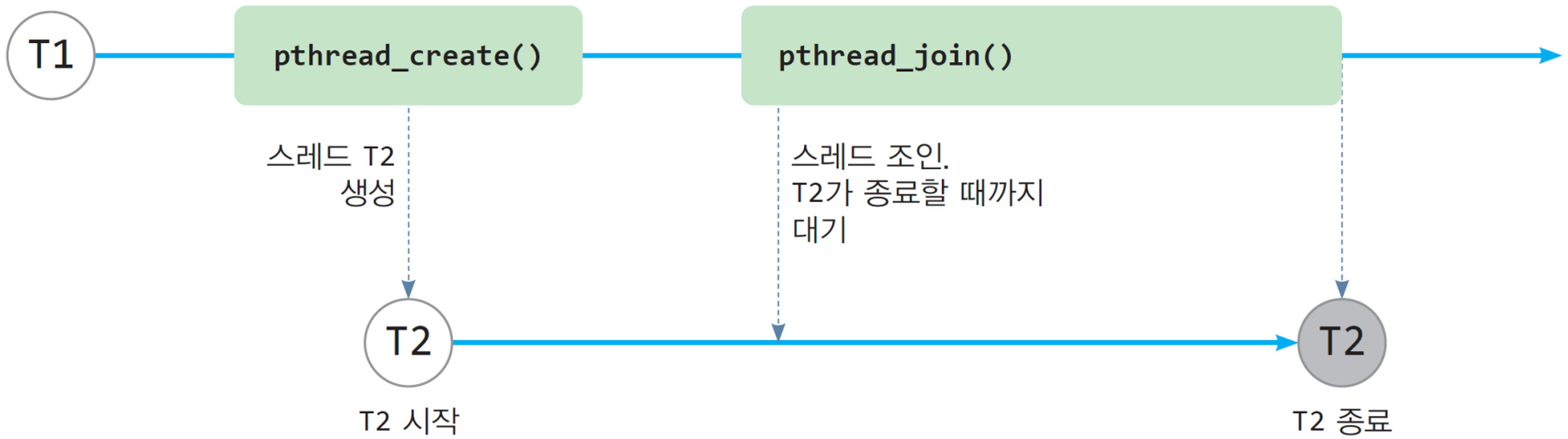


Thread operation (1)

- **쓰레드 생성**
  - 쓰레드는 쓰레드를 생성하는 시스템 호출이나 라이브러리 함수를 호출하여 다른 쓰레드 생성 가능
  - 프로세스가 생성되면 자동으로 main 쓰레드 생성됨
- **쓰레드 종료**
  - 프로세스 종료와 쓰레드 종료 구분 필요
  - **프로세스 종료**
    - 프로세스에 속한 어떤 쓰레드라도 exit() 시스템 호출을 부르면 프로세스 종료(모든 쓰레드 종료)
    - 메인 쓰레드의 종료(C 프로그램에서 main() 함수 종료) - 모든 쓰레드도 함께 종료
    - 모든 쓰레드가 종료하면 프로세스 종료
  - **쓰레드 종료**
    - pthread_exit()과 같이 쓰레드만 종료하는 함수 호출 시 해당 쓰레드만 종료
    - main() 함수에서 pthread_exit()을 부르면 역시 main 쓰레드만 종료

Thread operation (2)

- **쓰레드 조인 (join)**
  - 쓰레드가 다른 쓰레드가 종료할 때까지 대기
    - 주로 부모 쓰레드가 자식 쓰레드의 종료 대기

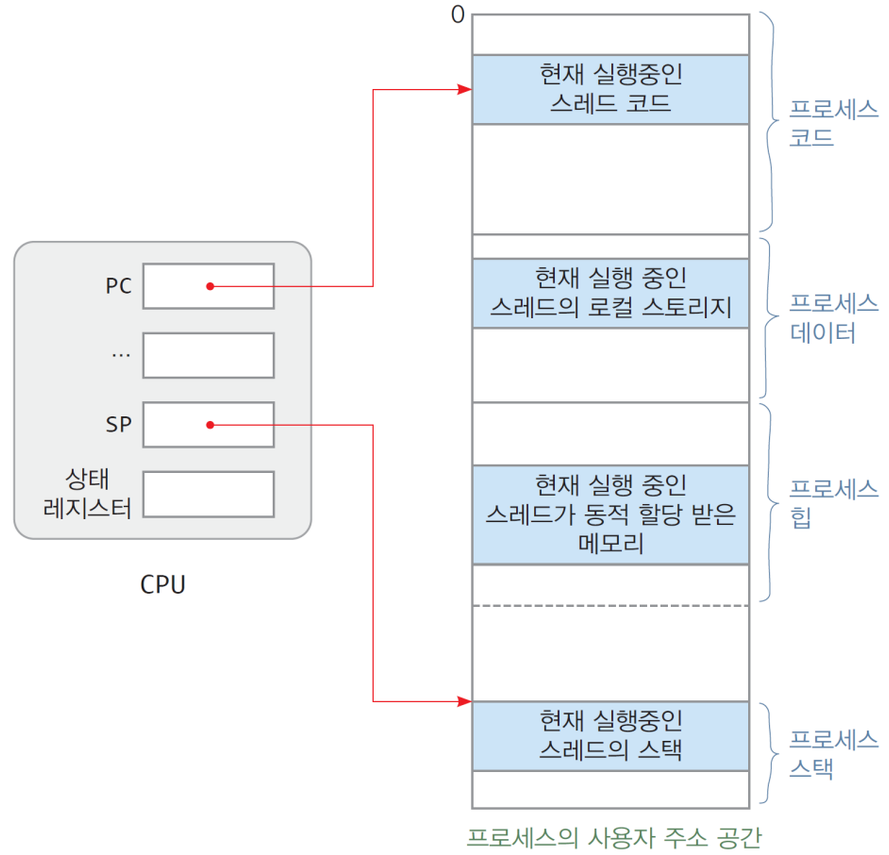


- **쓰레드 양보 (yield)**
  - 쓰레드가 자발적으로 yield()와 같은 함수 호출을 통해 자신의 실행을 중단하고 다른 쓰레드를 스케줄하도록 지시

Thread context

- **쓰레드의 실행중인 상태 정보들은 TCB에 저장됨**
  - **Thread control block**
    - 쓰레드가 생성될 때 커널에 의해 만들어짐
    - 쓰레드가 소멸되면 함께 사라짐
  - 각종 CPU 레지스터들의 값을 관리!
    - PC: 실행 중인 코드 주소
    - SP/BP: 실행 중인 함수의 스택 주소
    - Flag: 현재 CPU의 상태 정보
    - etc...
  - 나머지 메모리들은 어차피 공유하므로,
    레지스터들만 저장해두었다가
    필요할 때 CPU에 복귀하면
    이전에 실행하던 상태로 돌아갈 수 있음!

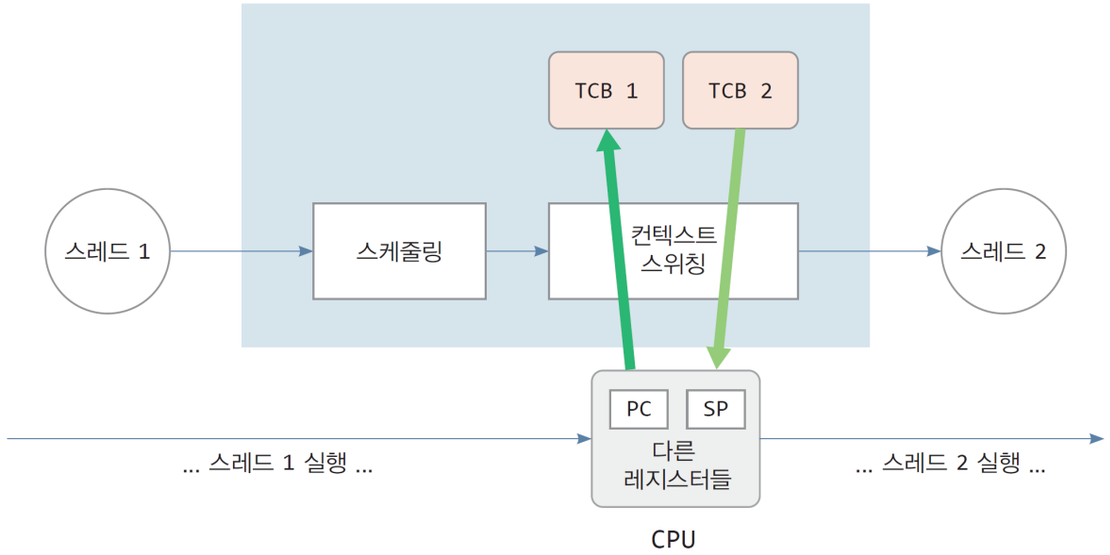
구분	요소	설명
스레드 정보	tid	스레드 ID. 스레드가 생성될 때 부여된 고유 번호
	state	스레드의 상태 정보. 실행(Running), 준비(Ready), 블록(Blocked), 종료(Terminated)
컨텍스트	PC	CPU의 PC 레지스터 값. 스레드가 다음에 실행할 명령의 주소
	SP	CPU의 SP 레지스터 값. 스레드 스택의 톱 주소
	다른 레지스터들	스레드가 중지될 때의 여러 CPU 레지스터 값들
스케줄링	우선순위	스케줄링 우선순위
	CPU 사용 시간	스레드가 생성된 이후 CPU 사용 시간
관리를 위한 포인터들	PCB 주소	스레드가 속한 프로세스 제어 블록(PCB)에 대한 주소
	다른 TCB에 대한 주소	프로세스 내 다른 TCB들을 연결하기 위한 링크
	블록 리스트/준비 리스트 등	입출력을 대기하고 있는 스레드들을 연결하는 TCB 링크, 준비 상태에 있는 스레드들을 연결하는 TCB 링크(스레드 스케줄링 시 사용) 등



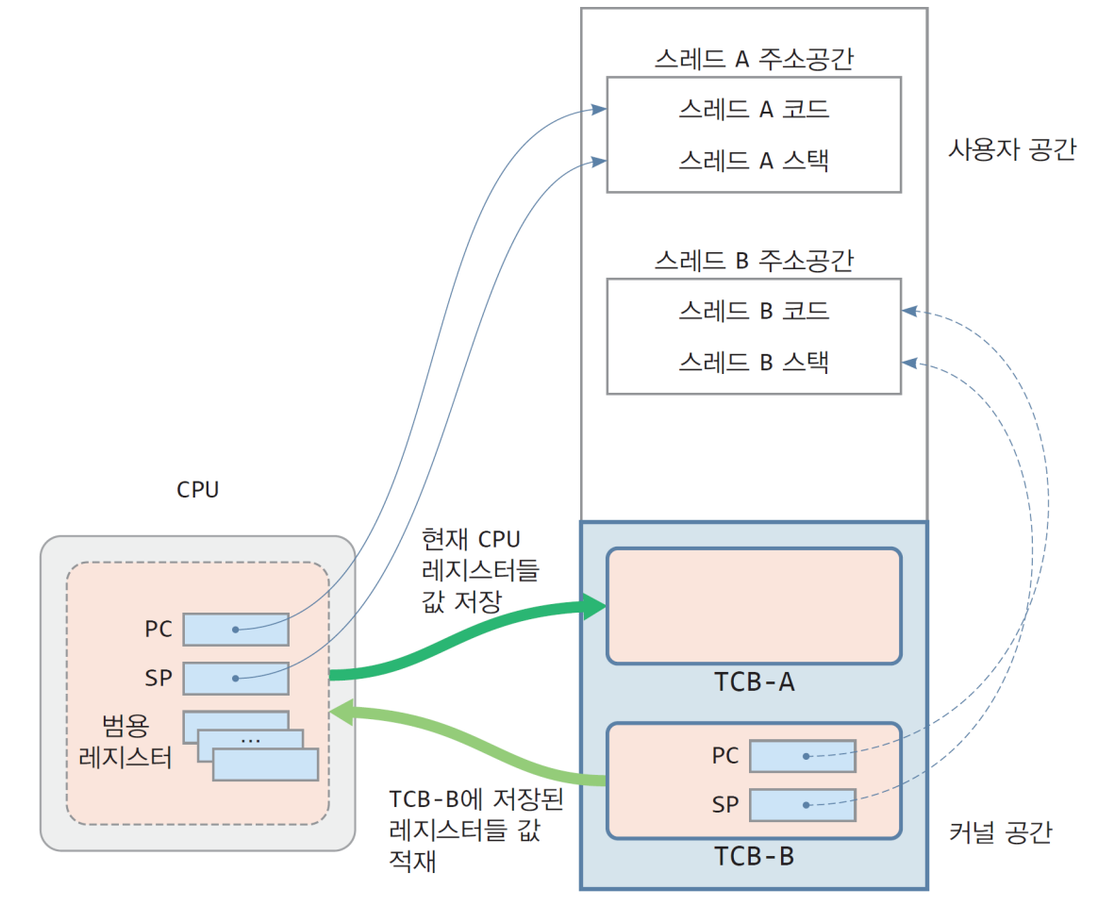
프로세스의 사용자 주소 공간

Thread context switching, a.k.a, thread switching

- 현재 실행중인 쓰레드를 중단시키고, 다른 쓰레드에게 CPU 할당, 현재 CPU 컨텍스트를 TCB에 저장하고, 다른 TCB에 저장된 컨텍스트를 CPU에 적재



- **1) CPU 레지스터 저장 및 복귀**
  - 현재 실행 중인 쓰레드 A의 컨텍스트를 TCB-A에 저장
  - TCB-B에 저장된 쓰레드 B의 컨텍스트를 CPU에 적재
  - CPU는 쓰레드 B가 이전에 중단된 위치에서 실행 재개 가능
  - SP 레지스터를 복귀함으로서 자신의 이전 스택을 되찾게 됨
  - 스택에는 이전 중단될 때 실행하던 함수의 매개변수나 지역변수들이 그대로 저장되어 있음
- **2) 커널 정보 수정**
  - TCB-A와 TCB-B에 쓰레드 상태 정보와 CPU 사용 시간 등 수정
  - TCB-A를 준비 리스트나 블록 리스트로 옮김
  - TCB-B를 준비 리스트에서 분리



Overhead in context switching

- **Context switching은 상당히 비싼 연산중 하나!**
  - 작업을 바꾸기 위해 작업을 하는 작업...(?)
  - CPU가 본래 할 일은 못하고 다른 작업에 리소스를 빼앗긴다! → CPU 시간 소모
    - Context switching의 시간이 길거나, 잦은 경우 컴퓨터 처리율이 심각하게 저하 될 수 있다!
- **동일한 프로세스의 다른 쓰레드로 스위칭되는 경우**
  - 1) 컨텍스트 저장 및 복귀
    - 현재 CPU의 컨텍스트(PC,PSP, 레지스터) TCB에 저장
    - TCB로부터 쓰레드 컨텍스트를 CPU에 복귀
  - 2) TCB 리스트 조작
  - 3) 캐시 Flush와 채우기 시간
- **다른 프로세스의 쓰레드로 스위칭하는 경우**
  - 다른 프로세스로 교체되면, CPU가 실행하는 주소 공간이 바뀌는 큰 변화로 인한 추가적인 오버헤드 발생
  - 1) 추가적인 메모리 오버헤드
    - 시스템 내에 현재 실행 중인 프로세스의 매핑 테이블을 새로운 프로세스의 매핑 테이블로 교체
  - 2) 추가적인 캐시 오버헤드
    - 프로세스가 바뀌기 때문에, CPU 캐시에 담긴 코드와 데이터 무력화
    - 새 프로세스의 쓰레드가 실행을 시작하면 CPU 캐시 미스 발생, 캐시가 채워지는데 상당한 시간 소요

쓰레드 모델

멀티쓰레딩 모델

Type of thread

- **Kernel-level thread**
  - 운영체제가 커널에서 관리하는 쓰레드.

- **User-level thread**
  - User-space에서 관리하는 쓰레드.

Kernel-level thread

- **커널 쓰레드:** 커널이 직접 생성하고 관리하는 쓰레드
  - 응용프로그램이 시스템 호출을 통해 커널 레벨 쓰레드 생성
  - 커널이 쓰레드에 대한 정보(TCB)를 커널 공간에 생성하고 소유 → 커널에 의해 스케줄
  - **쓰레드 주소 공간(쓰레드 코드와 데이터): 사용자 공간에 존재**
  - main 쓰레드는 커널 쓰레드
    - 응용프로그램이 적재되어 프로세스가 생성될 때 자동으로 커널은 main 쓰레드 생성

- **순수 커널 레벨 쓰레드(pure kernel-level thread)**
  - 부팅 때부터 커널의 기능을 돕기 위해 만들어진 쓰레드
  - 커널 코드를 실행하는 커널 쓰레드
  - **쓰레드의 주소 공간은 모두 커널 공간에 형성**
  - 커널 모드에서 작동, 사용자 모드에서 실행되는 일은 없음

User-level thread

- **사용자 쓰레드:** 라이브러리에 의해 구현된 일반적인 쓰레드
  - 응용프로그램이 라이브러리 함수를 호출하여 사용자 레벨 쓰레드 생성
  - 쓰레드 라이브러리가 쓰레드 정보(U-TCB)를 사용자 공간에 생성하고 소유
    - 쓰레드 라이브러리는 사용자 공간에 존재
    - 쓰레드 라이브러리에 의해 스케쥴
  - 커널은 사용자 레벨 쓰레드의 존재에 대해 알 수 없음 → **하나의 프로세스로만 인식**
  - 쓰레드 주소 공간(쓰레드 코드와 데이터) : 사용자 공간에 존재

Multithreading models

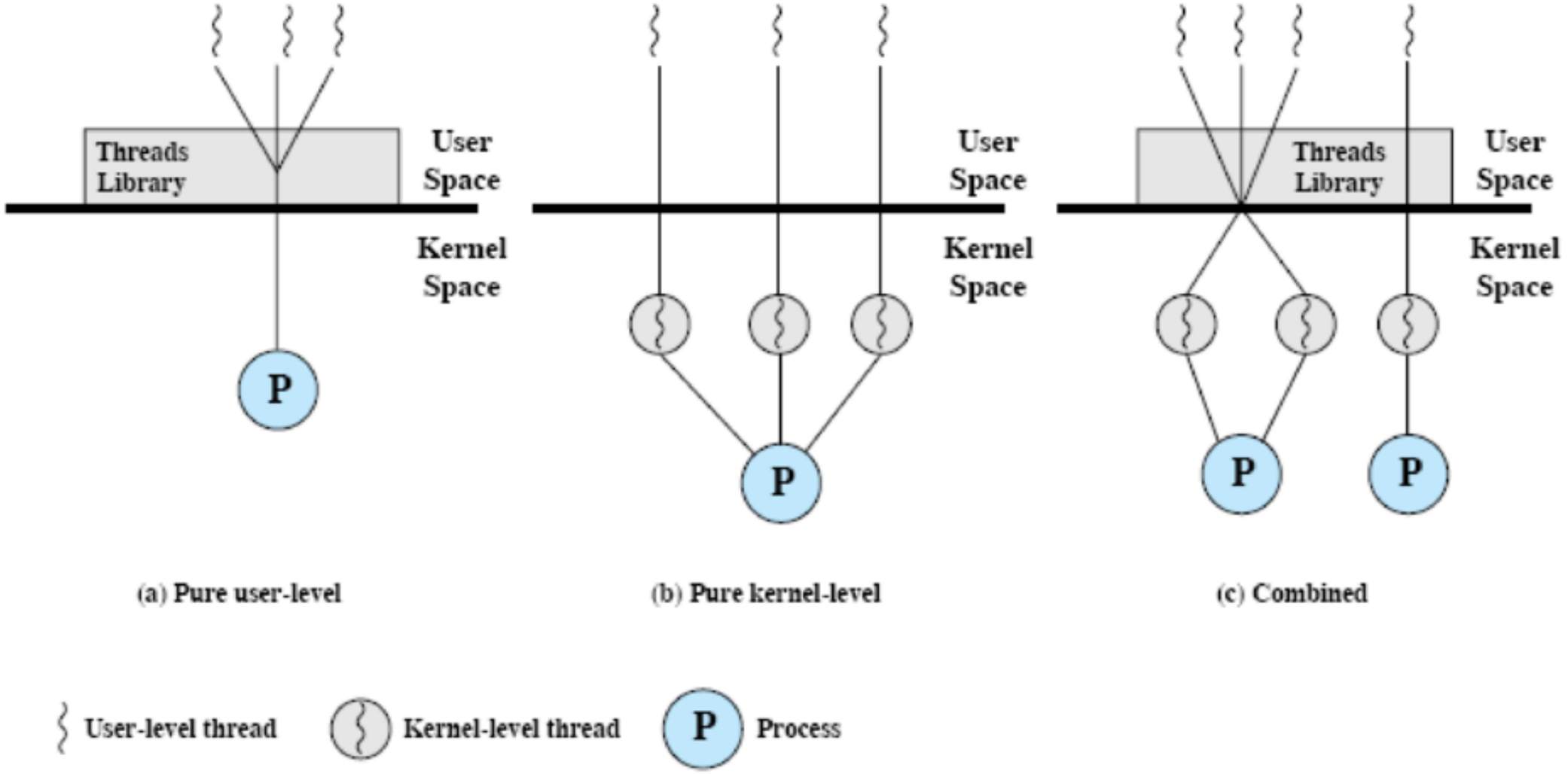
- 멀티쓰레드의 구현
  - 응용프로그램에서 작성한 쓰레드가 시스템에서 실행되도록 구현하는 방법
    - 사용자가 만든 쓰레드가 시스템에서 스케줄되고 실행되도록 구현하는 방법
    - 쓰레드 라이브러리와 커널의 시스템 호출의 상호 협력 필요

- **Many-to-One 'N:1' model**
  - N개의 사용자 레벨 쓰레드를 1개의 커널 레벨 쓰레드로 매핑

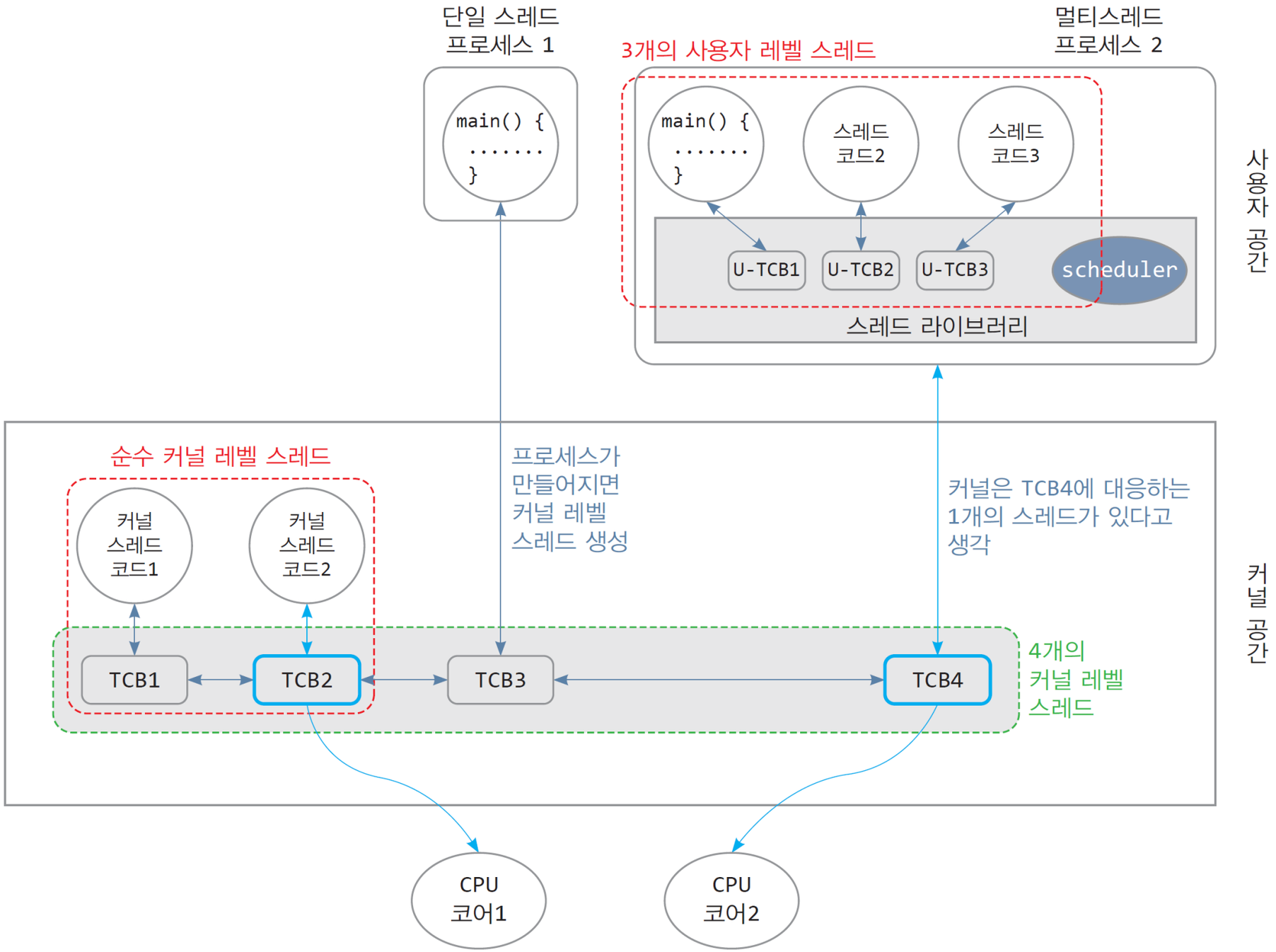
- **One-to-One '1:1' model**
  - 1개의 사용자 레벨 쓰레드를 1개의 커널 레벨 쓰레드로 매핑

- **Many-to-Many 'N:M' model**
  - N개의 사용자 레벨 쓰레드를 M개의 커널 레벨 쓰레드로 매핑

Compare

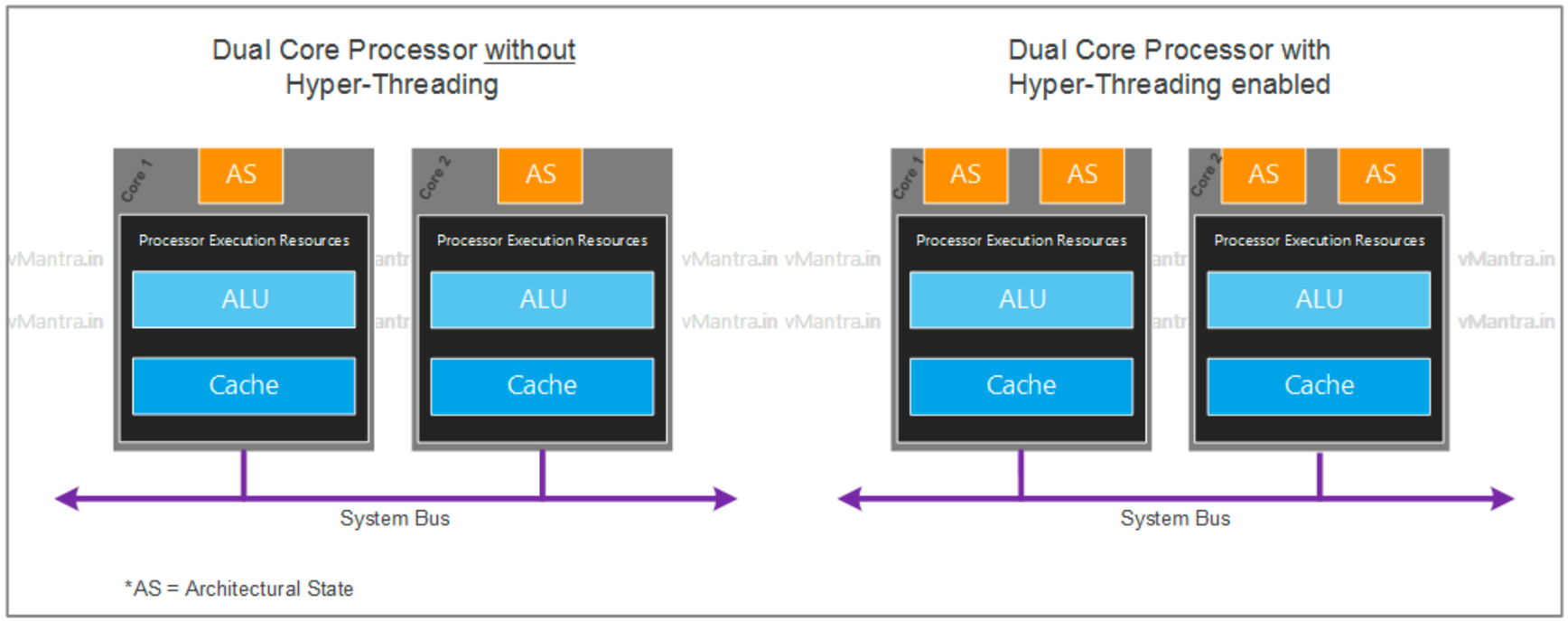

Threads Library
User Space
Kernel Space
P
(a) Pure user-level
(b) Pure kernel-level
(c) Combined
User-level thread
Kernel-level thread
Process

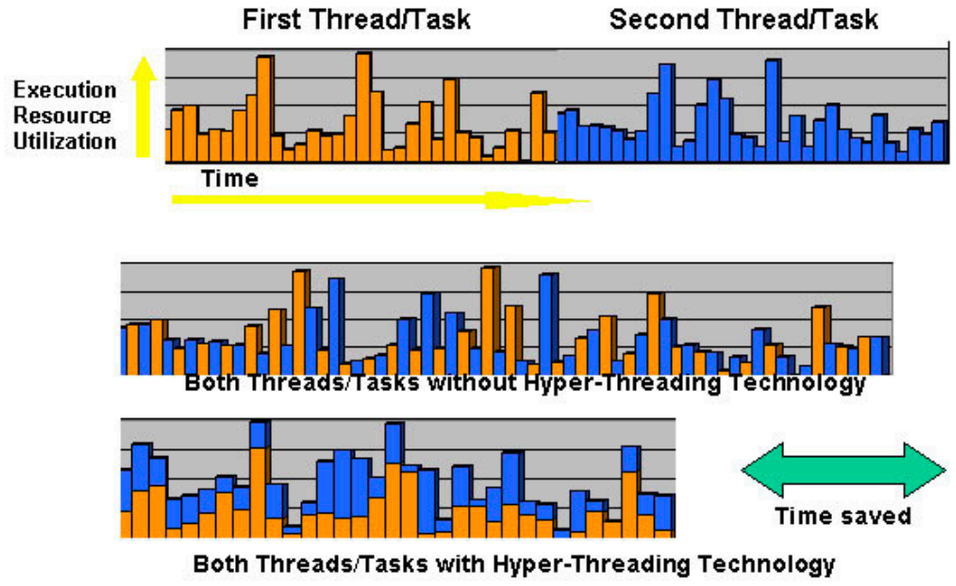
Kernel and user-level thread



c.f., Simultaneous multithreading (SMT)

- a.k.a., Hyper-threading





쓰레드 추가적인 이슈들...

멀티쓰레드와 fork()와 exec()

- **멀티쓰레드에서 fork() 시스템콜을 실행한다면 어떻게 될까??**
  - 여러 쓰레드 중 한 쓰레드가 fork()를 호출 → 새로운 프로세스는 fork()를 **'호출한 쓰레드 만'** 복제
  - 한 쓰레드가 exec()를 호출 → 프로세스 전체가 사라 짐

- **그러면 fork가 모든 쓰레드를 복제한다면 어떨까?**
  - c.f., 몇몇 시스템은 전체 쓰레드를 복제하는 fork도 지원함... 그러나,
  - fork() 이후 exec() 를 호출하면, 모든 쓰레드를 복제할 이유가 없다.
  - fork() 이후 exec()가 호출되지 않는다면, 모든 쓰레드의 복제가 의미가 있다.

자원 동기화 문제: thread-safe 개념

- 하나의 자원에 대해 여러 쓰레드가 동시에 접근 → 공유 데이터 훼손 문제

- Thread-safe
  - 멀티 쓰레드 프로그래밍에서 일반적으로 어떤 함수나 변수, 혹은 객체가 여러 쓰레드로부터 동시에 접근이 이루어져도 프로그램의 실행에 문제가 없음을 뜻함

- Thread-safe 를 지키기 위해서
  - **1. Re-entrancy**
    - 어떤 함수가 한 쓰레드에 의해 호출되어 실행 중일 때, 다른 쓰레드가 그 함수를 호출하더라도 그 결과가 각각에게 올바르게 주어져야함
  - **2. Thread-local storage**
    - 공유 자원의 사용을 최대한 줄여 각각의 쓰레드에서만 접근 가능한 저장소들을 사용함으로써 동시 접근을 막음
    - 이 방식은 동기화 방법과 관련되어 있고, 또한 공유상태를 피할 수 없을 때 사용
  - **3. Mutual exclusion**
    - 공유 자원을 꼭 사용해야 할 경우 해당 자원의 접근을 세마포어 등의 락으로 통제
  - **4. Atomic operations**
    - 공유 자원에 접근할 때 원자 연산을 이용하거나 '원자적'으로 정의된 접근 방법을 사용함으로써 상호 배제를 구현

쓰레드 정리!

프로세스와 쓰레드

- **프로세스는 쓰레드들의 공유 공간이다.**
  - 쓰레드의 주소 공간이 형성되고 공유됨

- **프로세스는 운영체제가 응용프로그램을 적재하는 단위이고, 쓰레드는 실행 단위이다.**
  - PCB에 저장된 정보는 환경 컨텍스트,
  - TCB에 저장된 정보는 실행 컨텍스트

- 다른 프로세스에 속한 쓰레드로의 스위칭보다

  동일한 프로세스에 속한 쓰레드 스위칭은 속도가 빠르다.

- **프로세스에 속한 모드 쓰레드가 종료할 때 프로세스가 종료한다.**

쓰레드로 프로그램을 작성하면...

- 병렬 실행 덕택에 실행 성능이 좋아져요!
- 우수한 응답성을 지닙니다!
  - 한 쓰레드가 블록되어도 다른 쓰레드를 통해 사용자 인터페이스 가능
  - → 서버 프로그램 등을 운영하는데 용이!
- 시스템 자원 사용의 효율성도 좋아집니다
  - 쓰레드는 프로세스에 비해 생성 유지 시 메모리나 자원 적게 사용
- 응용프로그램 구조의 단순화할 수 있어요!
  - 응용프로그램을 작업 기준으로 여러 함수로 분할
  - 각 함수별로 쓰레드 만들어 동시 실행
  - 새로운 기능 추가 용이, 프로그램의 높은 확장성
- 작성이 쉽고 효율적인 통신

Summary

- 프로세스의 컨텍스트 스위칭 문제 → 쓰레드 개념 등장
  - 오늘날 쓰레드가 실질적인 실행 단위; 프로세스는 쓰레드의 컨테이너
  - 함수 하나가 쓰레드 하나라 보면 쉬움! (main도 함수)

- 쓰레드 주소 공간은 프로세스의 주소 공간 내에 형성
  - 일반 함수가 수직적인 관계라하면, 쓰레드는 side-by-side 형태 (주의! 말이 그렇다는 거임)

- 쓰레드 관련 연산
  - 쓰레드 라이프 사이클과 동작: 생성, 종료, 조인, 양보 ~ Context switching
  - Thread control block (TCB)

- Multi-threading model
  - User-level thread, Kernel-level thread