14. Concurrency and Threads

▼ Threads

▼ Multi-threaded program

- single thread와 state : process와 유사하게 독립적인 context information 가짐
  - 각 thread in multi thread
    1. PC
    2. set of registers(private)
    3. stack : thread마다 존재
       - thread들이 각각의 context를 가져야 함 ↔ single thread

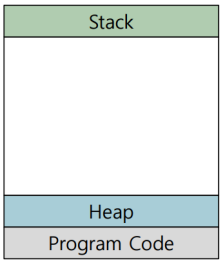
       ⇒ 호출되는 함수의 내용이 별도로 저장되어야 함
- 하나의 process 내부의 thread는 모두 같은 메모리 공간을 공유(heap 공유)
  - 같은 data에 접근할 수 있음
- **context switch**
  - thread control block(TCB) : TCB별로 context switch
    - in Linux : pcb와 tcb 구분을 따로 하지 않고 thread 단위로 switch
  - 하나의 process 내부에서 같은 메모리 공간 공유

    → 사용 중인 page table도 switch할 필요가 없다!

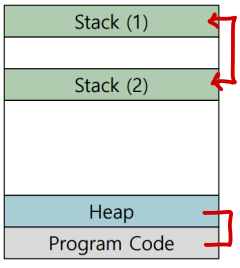
    (PDBR 바뀌지 X)
    - 같은 process 내부에서 thread context switch → 같은 address space

- single VS multi

• Single-threaded address space



• Multi-threaded address space



thread별로 저장되어야 함.

같은 process 내부라면 공유

*heap : race condition 일어나지 않도록 잘 관리

*code : read only, PC가 내부에서 각각 다른 부분 가리키도록!

▼ thread를 사용하는 이유

1. Parallelism : mutliple Cpus → 병렬성 증가
2. Avoiding blocking
   a. slow I/O
   b. 하나의 thread가 program을 기다리고 있을 때 CPU scheduler가 다른 thead 실행 가능

⇒ 많은 현대의 서버 기반 응용 프로그램에서 thread 사용 중 ex) web server, database manage

▼ Thread Creation

```c
void *mythread(void *arg) {
  printf("%s\n", (char *) arg);
  return NULL;
}
int main(int argc, char *argv[]) {
  pthread_t p1, p2;
  int rc;
①printf("main: begin\n");
②rc = pthread_create(&p1, NULL, mythread, "A");
  assert(rc == 0); ⇒ 이게 return되어야 create 성공적으로 완료된것
③rc = pthread_create(&p2, NULL, mythread, "B");
  assert(rc == 0);
④,⑥ // join waits for the threads to finish
  rc = pthread_join(p1, NULL); assert(rc == 0);
  rc = pthread_join(p2, NULL); assert(rc == 0);
⑧printf("main: end\n");
  return 0;
}
```

*CPU core가 하나라고 가정 후 실행한 예시

main	Thread 1 (T1)	Thread 2 (T2)
① prints "main: begin"		
② creates Thread 1		
③ creates Thread 2		
④ waits for T1		
	⑤ prints "A"	
⑥ waits for T2		
		⑦ prints "B"
⑧ prints "main: end"		

ex1 — mythread, ready, running, ready, running

main	Thread 1 (T1)	Thread 2 (T2)
① prints "main: begin"		
② creates Thread 1		
	⑤ prints "A"	
③ creates Thread 2		
		⑦ prints "B"
④ waits for T1		
⑥ waits for T2		
⑧ prints "main: end"		

ex2

2중 더 복잡한 예시!

▼ Shared Data

* 여러개의 thread가 어떤 순서로 발생할지 모르기에 발생되는 문제 살펴보자!

```
static volatile int counter = 0;
void * mythread(void *arg) {
  int i;
  printf("%s: begin\n", (char *) arg);
  for (i = 0; i < 1e7; i++) {
    counter = counter + 1;
  }
  printf("%s: done\n", (char *) arg);
  return NULL;
}
int main(int argc, char *argv[]) {
  pthread_t p1, p2;
  printf("main: begin (counter = %d)\n", counter);
  pthread_create(&p1, NULL, mythread, "A");
  pthread_create(&p2, NULL, mythread, "B");
  pthread_join(p1, NULL);
  pthread_join(p2, NULL);
  printf("main: done with both (counter = %d)\n", counter);
  return 0;
}
```

→ 서로다른 thread에서도 같은 memory 사용함.

⇒ 계속 1000만 씩 더함.

- Nondeterministic Results

```
main: done with both (counter = 20000000)
main: done with both (counter = 19345221)
main: done with both (counter = 19221041)
```

2000만보다 적은 횟수가 짝수로 발생

⇒ **race condition** 발생(실행 순서에 따라 결과가 달라짐)

→ 같은 변수에 동시에 접근하는 걸 방지해야 함

▼ Race Condition

실행 순서에 따라서 결과가 달라짐

▼ why? → example

```
counter = counter + 1;
기계어 code                    → counter 주소
100 mov 0x8049a1c, %eax        ⇒ memory → counter
105 add $0x1, %eax             ⇒ ++
108 mov %eax, 0x8049a1c        ⇒ counter → memory
memory 번지
```

같은 process 내에 있음. ⇒ address 공유

OS	Thread 1	Thread 2	(after instruction) PC	%eax	counter
			100	0	50
	mov		105	50	50
	add		108	51	50
Context switch					
			100	0	50
		mov	105	50	50
		add	108	51	50
		mov	113	51	51
Context switch					
			108	51	51
	mov		113	51	51

memory에 반영되지 않고 context switch

add 2번 했는데 1만 증가

▼ Critical Section

shared resource에 접근하는 code 영역 → 동시에 여러 thread가 접근하지 않도록!

- **Mutual exclusion**
  - 하나의 thread가 critical section 내에서 실행 중일 때 다른 thread가 접근하는 것을 막는다 →이를 보장

▼ Atomicity (원자성)

machine instruction의 특성

→ instruction 중간에 interrupted 되어서는 x

- interrupt가 일어났을 때 instruction이 모두 실행되지 않았거나 아예 종료되었거나 둘 중 하나의 상태에만 존재함, 이 사이 어중간하게 존재 x ☆
- 동기화를 어떻게 support?
  - atomic instructions
    - atomic memory add

- atomic update of B-tree → no(여러 r개의 thread가 동시에 update)

- OS : 이러한 instruction에 따라 일반적인 synchoronization privitives 구축

▼ Mutex

```
int pthread_mutex_lock(pthread_mutex_t *m)
int pthread_mutex_unlock(pthread_mutex_t *m)

int pthread_mutex_trylock(pthread_mutex_t *m)
int pthread_mutex_timedlock(pthread_mutex_t *m,
                            struct timespec *abs_timeout)
```

나time 정해서

```
//example
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER; //static init
int rc = pthread_mutex_init(&lock, NULL) //dynamic init
```

ex)

```
pthread_mutex_lock(&lock);
counter = counter + 1; // critical section
pthread_mutex_unlock(&lock);

 pthread mutex destroy() //destroy
```

▼ Condition Variables

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
int pthread_cond_signal(pthread_cond_t *cond)
```

mutex와 다르게 내가 원하는 상태가 되었을 때 다른 thread가 나에게 signal을 보내서 깨우는 것까지 가능

① synchoronizing two threads → 상당히 비효율적, 불안정한 구현 방법

```c
//thread 1
while (ready == 0)
        ; // spin -> cpu 낭비

//thread 2
ready = 1;
```

- error prone
  - 현대의 hw는 memory consistency model에 weak
  - compiler optimization

② 더 나은 사용법 ☆

```c
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_lock(&lock);
while (ready == 0)
  pthread_cond_wait(&cond, &lock);
pthread_mutex_unlock(&lock);
```

```c
pthread_mutex_lock(&lock);
ready = 1;
pthread_cond_signal(&cond);
pthread_mutex_unlock(&lock);
```

cond system call 사용 시 → mutex lock을 반드시 사용해야 함

```
#include <pthread.h>
prompt> gcc -o main main.c -Wall -pthread
```

compiling