Concurrency: An Introduction

Prof. Yongtae Kim

Computer Science and Engineering
Kyungpook National University

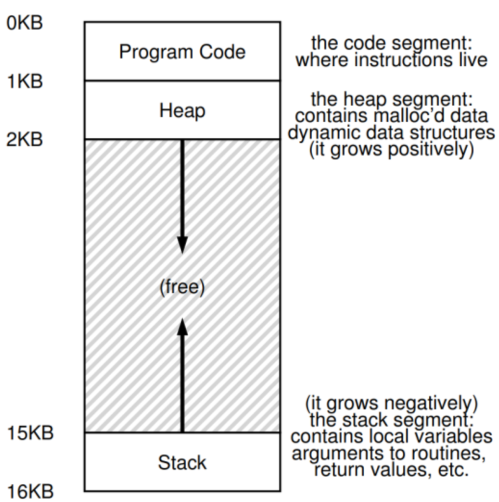
Thread: Concept

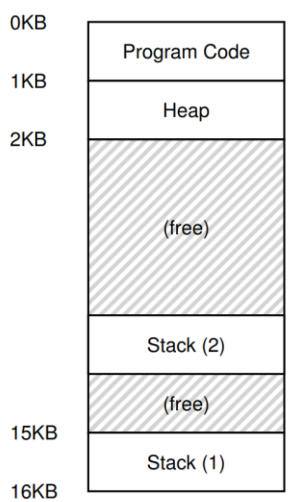
PCB

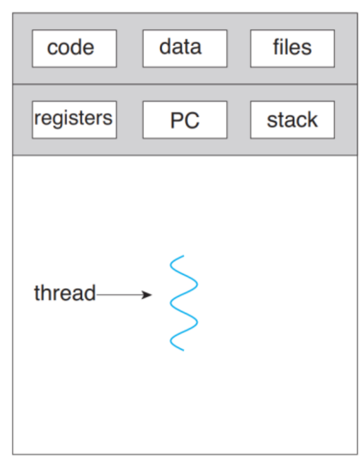
PC가 여러개

- **Thread is a new abstraction for a single running process**
  - 각각의 PC가 존재함.
  - A multi-threaded program has more than one point of execution (multiple PCs)
  - Thread is much like a separate process but the same address space is shared
- **The state of a single thread is very similar to that of a process**
  - Each thread has its own program counter (PC), private set of registers, thread control block (TCB) ➔ Switching between threads requires a context switch
  - The address space remains the same (i.e. no need to switch for page table)
  - Because each thread runs independently, there will be one stack per thread, sometimes called thread-local storage

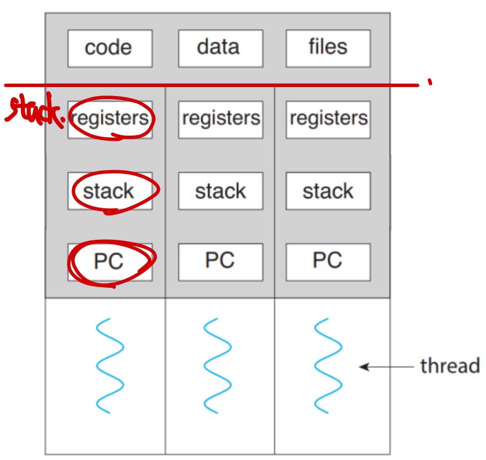
쓰레드는 page table을 switch할 필요가 없다.







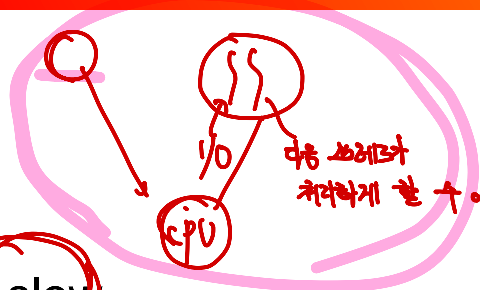
single-threaded process



multithreaded process

Why Use Threads?

병렬화!



- **The first reason to use threads is parallelism**
  - Single-threaded program: the task is straightforward but slow
  - Multi-threaded program: natural and typical way to make program run faster on modern hardware (e.g. multicore processor) → multicore processor에서 빠르게 동작하도록 thread가 필요함
  - Parallelization is the task of transforming standard single-threaded program into a program that does this sort of work on multiple CPUs
- **The second reason is to avoid blocking program progress due to slow I/O**

  " → 느린 I/O로 인해 프로그램의 수행이 차단되는 것을 막기 위함이다
  - Using threads is a natural way to avoid getting stuck due to I/O; While one thread is doing I/O, the other thread can utilize the CPU
  - Threading enables overlap of I/O with other activities within a single program

    → I/O가 발생하면 context-switch 하여 프로그램의 한 쓰레드가 대기하는 동안 다른 쓰레드를 overlap.
- **Multiple processes instead of threads can be used**
  - However, threads share an address space and thus make it easy to share data
  - Hence, they are a natural choice when constructing these types of program

↳ 독립적인 프로그램 (멀티 프로세스가 더 적합).

An Example: Thread Creation

- **Considering a program that creates two threads printing "A" and "B" independently**
  - Three threads: main thread, T1, and T2
  - The execution ordering may vary whenever it runs
  - What runs next is determined by the OS scheduler

순서가 다를 수 있음. →OS scheduler에 따라 다름

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: end\n");
    return 0;
}
```

Creating two threads

Waiting each thread to finish

끝날 때 까지 기다림.

Three different results:

Hard to know what will run when

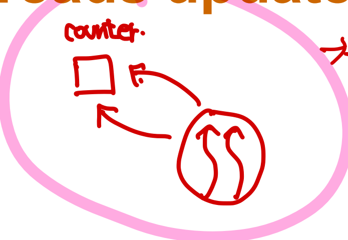
main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
		runs
		prints "B"
		returns
prints "main: end"		

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Why it Gets Worse: Shared Data → 쓰레드 share data.

- **Example where two threads update a global shared variable**



→ 공유 자원 (critical area).

```
#include <stdio.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

static volatile int counter = 0;

// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
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");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n",
            counter);
    return 0;
}
```

```
prompt> gcc -o main main.c -Wall -pthread; ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19221041)
```

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

Three different results:

Counter value may not be exactly 2e7

Not deterministic
Why does this happen?

Heart of the Problem: Uncontrolled Scheduling

- **To understand this, we must understand the code sequence that the compiler generates for the update to `counter`**
  – If counter = 50, we expect counter = 52 after two threads finished, however,

```
counter = counter + 1
```

메모리에서 로드

T1 T2

```
100 mov     0x8049a1c, %eax
105 add     $0x1,  %eax
108 mov     %eax,  0x8049a1c
```

더하고

메모리에 다시 저장.

The variable **`counter`** is assumed to locate at address **`0x8049a1c`**

OS	Thread 1	Thread 2	PC	eax	counter
			(after instruction)		
	before critical section		100	0	50
	`mov 8049a1c,%eax`		105	**50**	50
	`add $0x1,%eax`		108	**51**	50
interrupt					
save T1					
restore T2			100	0	50
		`mov 8049a1c,%eax`	105	**50**	50
		`add $0x1,%eax`	108	**51**	50
		`mov %eax,8049a1c`	113	51	**51**
interrupt					
save T2					
restore T1			108	51	51
	`mov %eax,8049a1c`		113	51	**51**

mov 이전에 context-switch 발생시 → 문제발생.

  – We see a race condition (or data race) where the results depend on the timing execution of the code → indeterminate results
- "Critical section" **is a piece of code that accesses shared variable**
  – This section must not be concurrently executed by more than once thread
  – We need mutual exclusion that guarantees that if one thread is executing within the critical section, the others will be prevented from doing so

상호 배타성

The Wish for Atomicity

→ 모든 instruction이 원자성을 만족하도록 하는건 불가능

- **What if we had a super instruction that looked like:**

```
memory-add 0x8049a1c, $0x1
```

  - Assuming this instruction adds a value to a memory location atomically
  - =atomic
  - There is no in-between state; run or not run at all (all or nothing)
  - Unfortunately, hardware doesn't support this kind of instruction

- **Instead, hardware can support some useful instructions for synchronization primitives**
  - Then, we can build multi-threaded code that accesses critical sections in a synchronized and controlled manner

T1 T2

done.

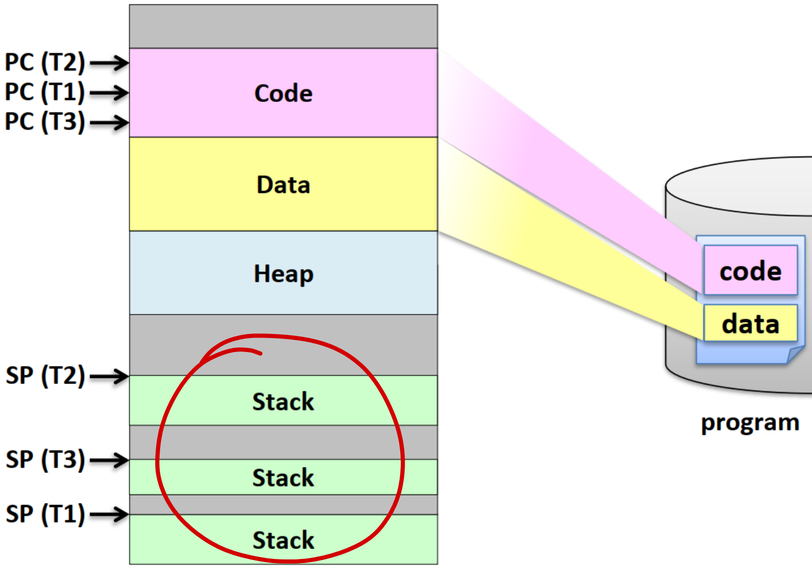
- **One more problem is waiting for another**
  - When a process performs a I/O and is put to sleep → when I/O completes, the process needs to be roused from its slumber so it can continue
  - We will also deal with this problem (conditional variables)

순서 제어

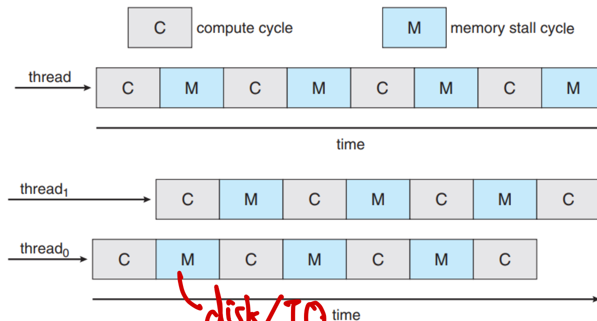
sync.

Summary

- **Thread is a new abstraction for a single running process**
  - A thread is a basic unit of CPU utilization and comprises a thread ID, a program counter (PC), a register set, and a stack
  - In modern OS, thread is a unit of scheduling
  - A process can have multiple threads and they share an address space
  - Multi-threading allows a program to effectively utilize the multi-core architecture



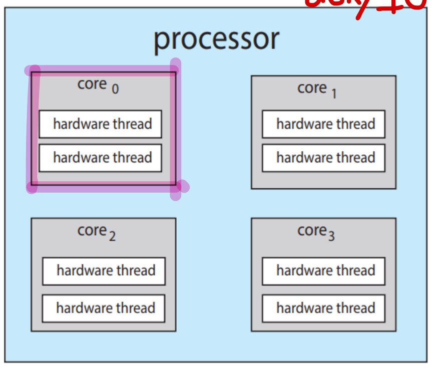
cpu가 1개인데
thread로 2개 처리

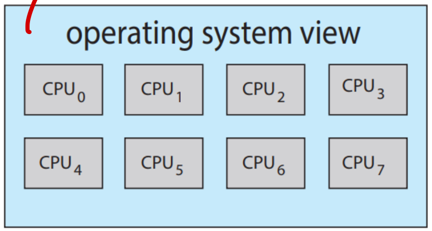


disk/IO

"hyper threading"

내부적으로 8개로 봄





Courtesy of Prof. Jin-Soo Kim @ SNU