Concurrency: An Introduction

Operating System (OS)

안인규

강의자료는 KAIST 원유집 교수님 OS 강의 자료를 참고했습니다.

Recap: Virtualization

- Virtualization of CPU and Memory
  - Transform a physical resource into a more general, powerful and easy-to-use virtual form of itself
  - System calls, Scheduling, Context Switching, Paging, …
    - Create the illusion of a large, private virtual memory for each process
    - Turn a single physical CPU into multiple virtual CPUs
      (이미 concurrency이 보장되는 것이 아닌가?)

- Virtualization → Concurrency (동시성)

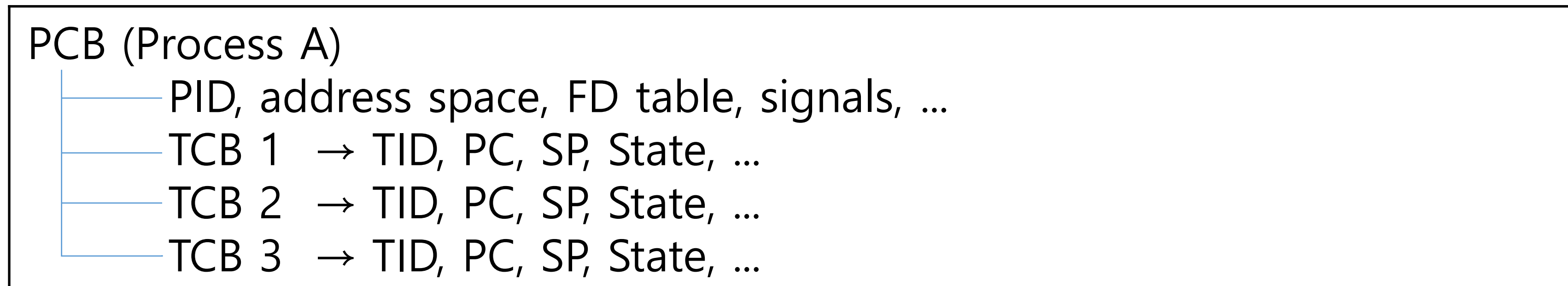
Thread

- A new abstraction for a single running process

- Multi-threaded program
  - A multi-threaded program has more than one point of execution.
  - Multiple PCs (Program Counter)
  - They share the share the same address space
    - 따라서, thread끼리 메모리 상의 데이터를 쉽게 주고받을 수 있음

Context switch between threads

- Each thread has its own program counter and set of registers
  - One or more **thread control blocks(TCBs)** are needed to store the state of each thread



- When switching from running one (T1) to running the other (T2),
  - The register state of T1 be saved
  - The register state of T2 restored
  - The address space remains the same
    - Page table을 그대로 유지 가능 → TLB entry도 그대로 유효
    - 만약, process를 바꿔서 page table이 바뀐다면, 기존 TLB entry는 더이상 유효하지 않음 (TLB flush 필요) → 이를 완화하기 위해 ASID를 활용

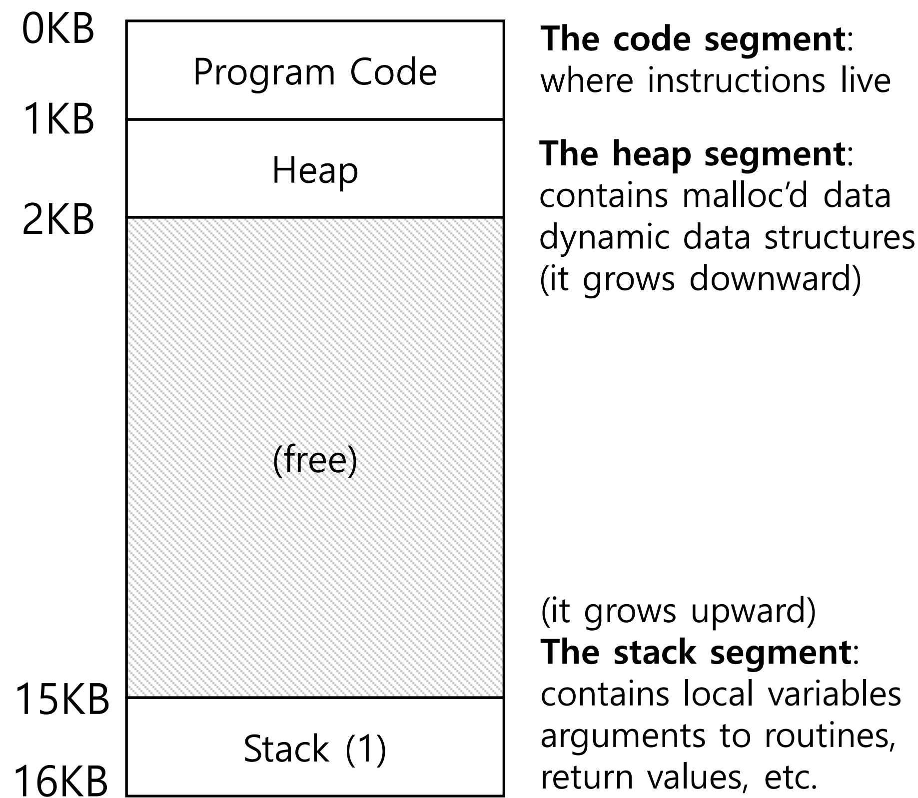
Context switch between threads

- Context switch from T1 to T2
  1. Kernel mode 진입 (Timer Interrupt / System call / page fault …)
  2. Register(CPU)를 T1's kernel stack에 저장
  3. Scheduler가 다음 process / thread 선택 → T2
  4. Switch from T1 to T2
     - A의 SP, PC 등(OS 마다 조금씩 다름)을 A's TCB에 저장
     - B의 TCB로부터 SP, PC 등을 복원
  5. 이제 CPU는 B's Kernel stack 위에서 실행 중

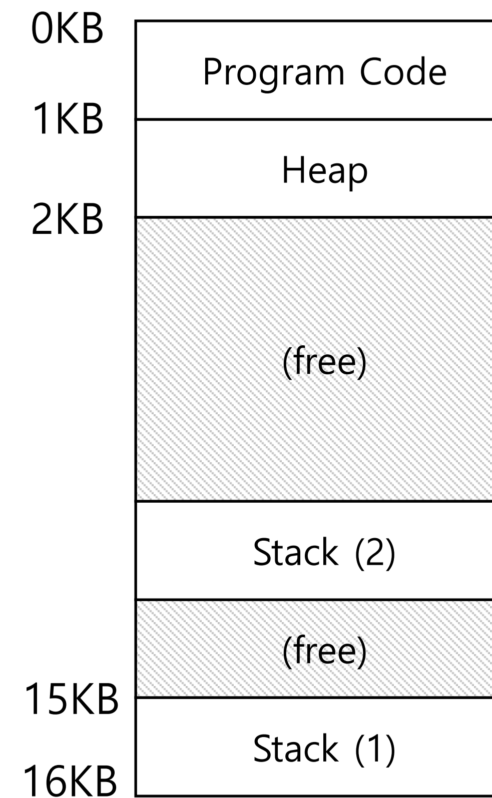
단계	Thread Switch (같은 process)	Process Switch
User register → Kernel stack 저장	O	O
Kernel SP 교체	O	O
Page table base 교체	X	O
FD table 등 교체	X (ASID 같음)	O (ASID 전환)
TLB 영향	X	O

The stack of the relevant thread

- There will be one stack per thread



A Single-Threaded Address Space



Two threaded Address Space

Why Use Threads?

- Parallelism
  - Single-threaded program: the task is straightforward, but slow.
  - Multi-threaded program: natural and typical way to make programs run faster on modern hardware.
  - **Parallelization**: The task of transforming standard **single-threaded** program into a program that does this sort of work on multiple CPUs.
- Avoid blocking program progress due to slow I/O.
  - Threading enables **overlap** of I/O with other activities within a single program.
  - It is much like **multiprogramming** did for processes across programs.
- Process 대신 Thread를 쓰는 이유?
  - 여러 프로세스가 메모리를 주고 받는 것이 복잡
  - 다만, 논리적으로 완전히 분리된 작업은 프로세스가 효과적 (isolation 보장)

An Example: Thread Creation

- Simple Thread Creation Code (`t0.c`)

```
1    #include <stdio.h>
2    #include <assert.h>
3    #include <pthread.h>
4    #include "common.h"
5    #include "common_threads.h"
6
7    void *mythread (void *arg) {
8        printf ("%s\n", (char *) arg);
9        return NULL;
10   }
11
12   int main (int argc, char *argv[]) {
13       pthread_t p1, p2;
14       int rc;
15       printf("main: begin\n");
16       Pthread_create(&p1, NULL, mythread, "A");
17       Pthread_create(&p2, NULL, mythread, "B");
18       // join waits for thre threads to finish
19       Pthread_join(p1, NULL);
20       Pthread_join(p2, NULL);
21       printf("main: end\n");
22       return 0;
23   }
```

Thread Trace (1)

main	**Thread 1**	**Thread 2**
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"		

Thread Trace (2)

main	Thread 1	Thread 2
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”		

Thread Trace (3)

main	Thread 1	Thread 2
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"		

Thread의 순서는 어떻게 정하지?

An Example: Sharing Data

- Sharing Data (`t1.c`)

```
1   static volatile int counter = 0;

2   void *mythread(void *arg) {
3       printf("%s: begin\n", (char *) arg);
4       for (int i = 0; i < 10000000; i++) {
5           counter = counter + 1;
6       }
7       printf("%s: done\n", (char *) arg);
8       return NULL;
9   }

10  int main(int argc, char *argv[]) {
11      pthread_t p1, p2;
12      printf("main: begin (counter = %d)\n", counter);
13      Pthread_create(&p1, NULL, mythread, "A");
14      Pthread_create(&p2, NULL, mythread, "B");
15      Pthread_join(p1, NULL);
16      Pthread_join(p2, NULL);
17      printf("main: done with both (counter = %d)\n", counter);
18      return 0;
19  }
```

An Example: Sharing Data

- Sharing Data (`t1.c`)

```
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

→ Race condition

Race condition

- Increasing a value of a variable
  - counter = counter + 1

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

Race condition

- Increasing a value of a variable
  - counter = counter + 1

```
105     mov 0x8049a1c, %eax     → counter 값을 메모리에서 레지스터로 읽어옴
108     add $0x1, %eax          → 레지스터에 1을 더함
113     mov %eax, 0x8049a1c     → 결과를 다시 메모리에 저장
```

Race condition

- Example with two threads
  - counter = counter + 1 (default is 50)
  - We expect the result is 52. However,

OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	before critical section		100	0	50
	`mov 0x8049a1c, %eax`		105	50	50
	`add $0x1, %eax`		108	51	50
`interrupt`					
`save T1's state`					
`restore T2's state`			100	0	50
		`mov 0x8049a1c, %eax`	105	50	50
		`add $0x1, %eax`	108	51	50
		`mov %eax, 0x8049a1c`	113	51	51
`interrupt`					
`save T2's state`					
`restore T1's state`			108	51	50
	`mov %eax, 0x8049a1c`		113	51	**51**

Thread 중간에 context switching이 발생

A few terminologies

- Race condition:
  - the results depend on the timing execution of the code.
  - Result is indeterminate
- Critical section
  - A piece of code that accesses a shared variable and must not be concurrently executed by more than one thread.
  - Multiple threads executing critical section can result in a race condition.
  - Need to support **atomicity (원자성-쪼갤 수 없는 성질)** for critical sections (**mutual exclusion**)

The wish for atomicity

- Ideal approach; make the increment as a single assembly instruction

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

→ *메모리의 counter 값에 1을 더함*

- In general, we do not have such instruction. Instead, we use lock.
- Ensure that any such critical section executes as if it were a single atomic instruction (**execute a series of instructions atomically**).

```
1    lock_t mutex;
2    . . .
3    lock(&mutex);
4    balance = balance + 1;
5    unlock(&mutex);
```

→ Critical section (line 4)