

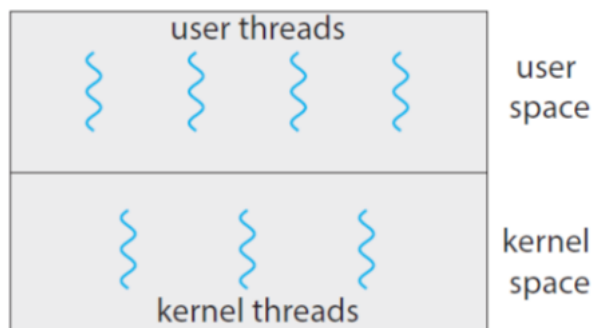


Chapter 4. Thread & Concurrency (Part 2)

4.3 Multithreading Model

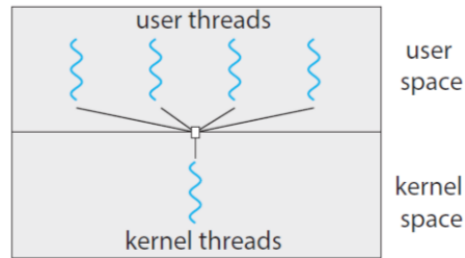
Thread의 두 가지 종류

- **User threads** : 지금까지 배운 쓰레드 > java로 쓰레드를 만든 예제들
 - kernel 위에서 threading함.
 - kernel의 간섭없이 수행
- **Kernel threads** : 운영체제의 쓰레드 > cpu core에서 직접 threading을 하며 os에서 직접 관리함
 - 운영체제가 직접 관리하는 쓰레드

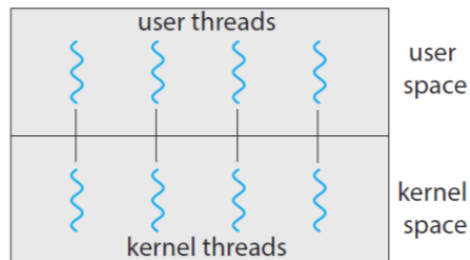


User thread와 Kernel thread의 관계

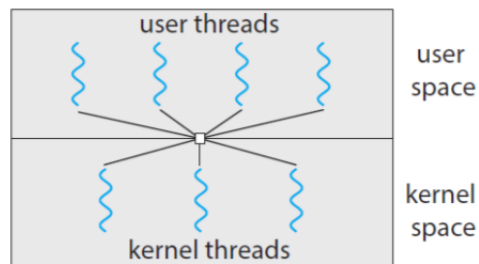
1. **Many-to-One Model** : 많은 user 쓰레드가 하나의 kernel 쓰레드의 서비스를 받음 → 가장 기본적인 모델



2. **One-to-One Model** : 하나의 user 쓰레드가 하나의 kernel 쓰레드의 서비스를 받음



3. **Many-to-Many Model** : 많은 user쓰레드가 많은 kernel 쓰레드의 서비스를 받음



4.4 Thread Libraries

Thread Library

- thread를 생성하고 관리하는 API를 제공

많이 사용하는 Thread 라이브러리

1. **POSIX Pthreads** : Linux, unix 기반
2. **Window threads**
3. **java threads**

Pthreads

- 스레드 동작에 대한 사양일 뿐, 구현이 되어있는 건 아니다.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int sum;

void * runner(void *param); //java의 public void run() 메소드와 같음

int main(int argc, char *argv[])
{
    pthread_t tid;
    pthread_attr_t attr;

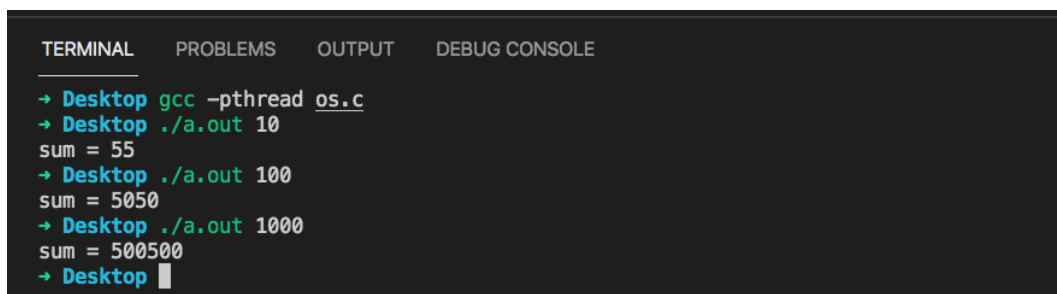
    pthread_attr_init(&attr);
    pthread_create(&tid, &attr, runner, argv[1]);
    pthread_join(tid, NULL); // main 스레드는 wait

    printf("sum = %d\n", sum);
}

void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;
    for(i = 0 ; i <= upper ; i++)
        sum += i;
    pthread_exit(0);
}
```

- `pthread_create()` == java에서 `new Thread()` 와 같다.
- 위 코드를 gcc로 실행할 때, `-pthread` 옵션을 붙여주어야 한다.
 - `$gcc -pthread 4.11.pthread.c`

실행결과



```
TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE
→ Desktop gcc -pthread os.c
→ Desktop ./a.out 10
sum = 55
→ Desktop ./a.out 100
sum = 5050
→ Desktop ./a.out 1000
sum = 500500
→ Desktop █
```

예제 1

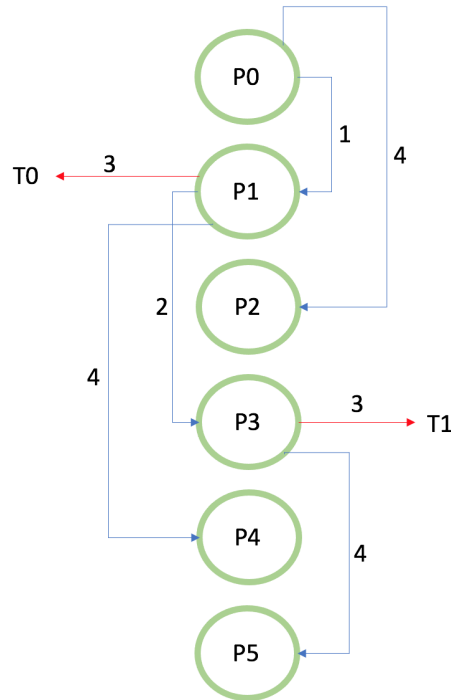
```
pid_t pid;
```

```

pid = fork(); //1
if (pid == 0){
    fork(); //2
    thread_create(..); //3
}
fork(); // 4

```

- process의 갯수는? 6개
- thread의 갯수는? 8개 (thread_create() 로 만들어지는 thread는 2개)



예제 2

```

#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/wait.h>

int value = 0;
void * runner(void *param);

int main(int argc, char *argv[])
{
    pid_t pid;
    pthread_t tid;
    pthread_attr_t attr;

    pid = fork(); //[1] : p1

    if(pid == 0) { //child process
        pthread_attr_init(&attr);

```

```

pthread_create(&tid, &attr, runner, NULL); //[2] : T1생성
pthread_join(tid, NULL); // T0는 wait, T1은 exec
printf("CHILD: value = %d\n", value);
}
else if (pid > 0 ) { //parent process
    wait(NULL);
    printf("PARENT: value = %d\n", value);
}
}
void *runner(void *param)
{
    value = 5;
    pthread_exit(0);
}

```

- 결과

```

TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE  > zsh +
→ Desktop gcc -pthread os.c
→ Desktop ./a.out
CHILD: value = 5
PARENT: value = 0

```



P0 : main proces

P1 : [1]에 의해 생성된 process (P0를 복제 > 다른 메모리 공간을 사용)

T0 : P1의 main thread

T1 : [2]에 의해 생성된 thread (T0, T1은 P1 안에서 실행 > 같은 메모리 공간 사용)

⇒ 다른 process는 메모리 공간을 공유하지 않는다.

⇒ 같은 process안의 여러 개의 thread는 같은 공간을 공유한다.

4.5 Implicit Threading

Implicit Threading

- concurrent + parallel 응용프로그램을 설계하는 것
 - = multicore system + multithreading 을 설계하는 것
 - = 너무 어려운 일
- 그래서, 위의 작업을 **compiler** , **libraries** 가 대신 수행할 수 있다.
 - ↳ **complier** : **openMP** (open Multi-Processing) : 공유 메모리 다중 처리 API
 - ↳ **library** : **java.util.concurrent.*** package : 동기화 관련 다양한 클래스 제공

Implicit Threading 방법

1. **Thread pools** : 여러 개의 thread를 미리 만들어 두고 (pool에 두고) 필요할 때, 가져다 사용한다.
↳ 유저가 new Thread()를 하는 것은 프로그램에 부하를 줄 수 있다.
2. **Fork & Join** == create & wait
↳ 명시적(explicit) 쓰레딩도 어떤 프로그램에 있어서는 implicit threading에 사용 될 수 있다.
3. **OpenMP** : C/C++에서 쉽게 병렬처리 할 수 있게 해둔 API
4. **Grand Central Dispatch (GCD)** : Apple에서 개발한 것 (MacOS, iOS)

OpenMP

- parallel region을 지정해주면, 알아서 작업을 수행
1. parallel하게 수행해야 하는 소스코드에 컴파일러 지시어를 적어준다.
 2. OpenMP가 그 region을 알아서 parallel하게 수행해준다.

omp 예제1

- OS.C

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[]){
    #pragma omp parallel // complier directive
    {
        printf("I'm a parallel region.\n");
    }
    return 0;
}
```

- 실행

```
$ gcc -fopenmp os.c
$ ./a.out
```

- 결과

```
I'm a parallel region.
I'm a parallel region.
I'm a parallel region.
I'm a parallel region.
I'm a parallel region.
...
```

>병렬로 실행됨

omp 예제2

- OS.C

```
#include <omp.h>
#include <stdio.h>

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

    omp_set_num_threads(4); //4개 thread 만들

    #pragma omp parallel // complier directive
    {
        printf("OpenMP thread : %d\n", omp_get_thread_num());
    }
    return 0;
}
```

- 결과

```
OpenMP thread : 0
OpenMP thread : 1
OpenMP thread : 3
OpenMP thread : 2
```

>랜덤으로 출력됨

>병렬로 수행됨

omp 예제3

```
#include <omp.h>
#include <stdio.h>

#define SIZE 1000000000

int a[SIZE], b[SIZE], c[SIZE];

int main(int argc, char *argv[]){
    int i;
    for(i = 0 ; i < SIZE; i++)
        a[i] = b[i] = i;

    #pragma omp parallel for
    for(i = 0 ; i < SIZE ; i++)
        c[i] = a[i] + b[i];

    return 0;
}
```

- 병렬로 돌린 코드와 아닌 코드의 시간을 비교했을때,

```
joonion@joonionpc:~/VSCode/OperatingSystemConcepts$ time ./sum_not_parallel
```

```
real    0m0.586s
user    0m0.364s
sys     0m0.223s
```

```
joonion@joonionpc:~/VSCode/OperatingSystemConcepts$ time ./sum_with_openmp
```

```
real    0m0.423s
user    0m1.091s
sys     0m0.441s
```

⇒ real 시간만 비교했을 때, parallel한 코드가 더 짧게 걸린 것을 확인할 수 있다.

- `sum_not_parallel`의 경우, user + sys를 더한 값이 병렬처리한 시간이고
- `sum_with_openmp`의 경우, sys 에서 대부분 병렬처리를 수행하여, real 시간이 더 짧게 나온다.
 - user : thread switching 이 일어나는 시간 포함?