Part 02. 리눅스 시스템 프로그래밍

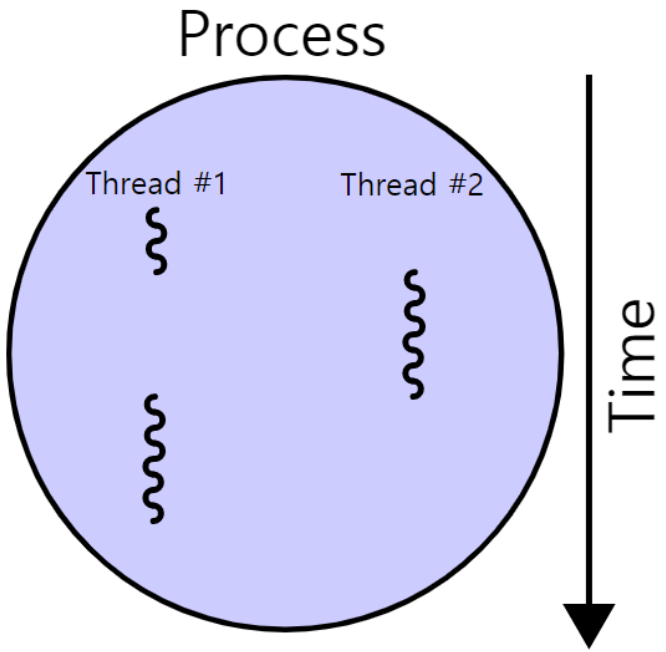
Chapter 08. 스레드(Thread)

진행 순서

Chapter 08_01 스레드 개요
Chapter 08_02 스레드 생성
Chapter 08_03 스레드 종료
Chapter 08_04 스레드 대기
Chpater 08_05 스레드 동기화
Chapter 08_06 스레드 실습

Chapter 08_01 스레드 개요

스레드(Thread) 란?
어떤 프로그램(프로세스) 내에서 실행되는 흐름의 단위,
일반적으로 한 프로그램은 하나의 스레드를 가지지만, 둘 이상의 스레드를 동시에 실행할 수도 있다.
(멀티스레드, Multithread)



Chapter 08_01 스레드 개요

예를 들어 어떤 신호나 데이터 입력을 기다리다가 입력이 들어오면 입력 데이터를 분석하여,
분석의 결과를 특정 파일에 기록하는 프로그램을 작성한다고 가정하자.

절차식 언어인 C 프로그램의 특성 상 main() 함수의 흐름대로 입력을 기다리고 해당 입력 데이터를 읽어서(read),
분석의 결과를 특정 파일에 기록(write) 루틴이 끝나면 다시 루틴의 처음으로 가서 입력 데이터를 기다리게 될 것이다.(read)
하지만 처리하는 도중에 입력이 들어올 경우 해당 입력은 읽어들이기 전까지 대기하게 된다.

이럴때 여러 입력을 병렬적으로 처리할 수 있는 방법중에 하나가 스레드(Thread) 이다.
입력 데이터를 읽어서 처리하는 스레드가 존재하면,
입력 데이터를 인지할 때마다 이후 처리를 스레드에게 맡기고, 다시 입력 데이터를 인지할 수 있다.
즉, 여러 입력 데이터를 처리하는 스레드가 동시에 여러개가 동작하는 것이다.

```
main() {
   while (true) {
      if (입력 데이터 존재)
         처리 쓰레드(work) 생성; /* 생성 되자마자 다시 루프 */
   }
}

work(입력 데이터) { /* 별도 독립적으로 동작 */
   입력 데이터 분석;
   데이터 처리;
   파일에 기록;
}
```

Chapter 08_01 스레드 개요

다중 프로세스와 다중 스레드 차이

다중 프로세스의 경우 프로세스간 데이터 공유를 위하여 IPC 등의 매커니즘을 이용하여 데이터 공유
다중 스레드의 경우 프로세스 내에서 전역 변수 등 데이터 공유

다중 프로세스의 경우 프로세스간의 컨택스트 스위칭 시 다중 스레드보다 많은 비용을 소모하므로,
다중 스레드가 좀 더 빠를 수 있고 메모리 공유로 인하여 메모리를 절약할 수 있다.

프로세스 컨택스트 스위칭 이란?

특정 CPU에서 A라는 프로세스가 돌아가다가 B라는 프로세스 작업을 처리해야하는 경우
A 프로세스를 위해 가지고 있던 가상 메모리 공간, 버퍼 등을 비우고 B 프로세스를 위한
가상 메모리 공간, 버퍼 등을 준비해야 한다. 이 처리를 컨택스트 스위칭이라 한다.

주의할 점

동일한 프로세스의 다중 스레드의 경우 공유된 리소스에 동시 접근할 경우 동기화에 주의해야 한다.
경쟁상태에서 데이터의 원자성이 깨지거나, 동기화 락 사용 시 데드락 등에 주의해야 한다.

Chapter 08_01 스레드 개요

리눅스에서는 Pthread API를 통해 스레드를 지원하고 있다.
일반적으로 Pthread라고 하는 POSIX 스레드는 표준 POSIX.1c, 스레드 확장 (IEEEstd 1003.1c-1995)에 정의된 API.

Pthread API는 <pthread.h> 파일에 정의되어 있으며, API의 모든 함수는 pthread_로 시작한다.

Pthread API가 코드에 추가된 경우 gcc를 통해 컴파일할 때는 –pthread 플래그를 통하여
libpthread 라이브러리를 링크해 주어야 한다.

```
# gcc -pthread hello.c –o hello
```

다음으로 스레드 생성, 종료, 대기, 동기화 하기 위한 API들을 알아보자.

Chapter 08_02 스레드 생성

```
#include <pthread.h>

int pthread_create (pthread_t *thread,
                    const pthread_attr_t *attr,
                    void *(*start_routine) (void *),
                    void *arg);
```

pthread_create()는 새로운 스레드를 생성한다.
생성되는 새로운 스레드는 arg를 인자로하는 start_routine 함수를 수행한다.
thread 인자는 NULL이 아니라면, 새로 만든 스레드를 나타내는 스레드 ID를 저장한다.
attr 인자는 스레드 생성 시 속성을 변경하기 위한 값이며, NULL일 경우 기본 속성을 따른다.

성공할 경우 0 리턴, 실패할 경우 0이 아닌 에러 코드를 직접 리턴.

errno
EAGAIN 새로운 스레드를 만들기 위한 리소스 부족
EINVAL 유효하지 않은 attr 속성 값
EPERM 권한 오류

Chapter 08_02 스레드 생성

void * start_routine (void *arg);

start_routine() 함수는 상기와 같은 형식을 가진다.
fork()와 유사하게 새로 생성된 스레드는 부모 스레드로부터 대부분의 속성과 기능, 상태 등을 상속받는다.
하지만 프로세스와는 다르게 스레드는 부모 스레드의 리소스를 공유한다.

생성된 스레드는 pthread_exit()을 호출하거나 start_routine에서 return할 경우 종료된다.

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

void *start_func(void *arg)
{
        int i, num = *(int *)arg;
        for (i = 0; i < num; i++)
                printf("%d\n", i);
        pthread_exit((void *)0);
}
```

```
int main()
{
        int num = 10;
        pthread_t thread_t;
        int ret, status;

        ret = pthread_create(&thread_t, NULL, start_func, (void *)&num);
        if (ret) {
                printf("pthread_create error: %s\n", strerror(ret));
                return -1;
        }

        pthread_join(thread_t, (void **)&status);
        printf("Thread returned: %d\n", status);
        return 0;
}
```

Chapter 08_03 스레드 종료

```
#include <pthread.h>

void pthread_exit (void *retval);
```

pthread_exit()는 현재 실행중인 스레드를 종료시키기 위해 호출한다.
retval은 해당 스레드가 종료되기를 기다리는 다른 스레드에게 전달할 값이다.

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

void *start_func(void *arg)
{
        int i, num = *(int *)arg;
        for (i = 0; i < num; i++)
                printf("%d\n", i);
        pthread_exit((void *)0);
}
```

Chapter 08_03 스레드 종료

```
#include <pthread.h>

int pthread_cancel (pthread_t thread);
```

pthread_cancel() 함수를 통해 다른 스레드를 취소시켜 종료할 수 있다.
인자 thread로 표현된 스레드 ID를 가진 스레드에 취소 요청을 보낸다.

성공 시 0 리턴, 실패 시 thread가 유요하지 않다는 ESRCH 리턴

취소 요청을 받은 스레드는 pthread_exit(PTHREAD_CANCELED) 를 수행한다.

pthread_create()를 통해 만들어 지는 스레드는 별다른 설정이 없을 경우
PTHREAD_CANCEL_ENABLE, PTHREAD_CANCEL_DEFERRED 상태로 만들어 진다.
이 의미는 취소가 가능하며, 취소 시점은 취소 요청이 들어올 경우 안전한 시점에 종료를 한다는 의미이다.

```
int pthread_setcancelstate (int state, int *oldstate);
int pthread_setcancelgype (int type, int *oldtype);
```
상기 함수를 통해 취소 상태와 타입 변경이 가능

Chapter 08_03 스레드 종료

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

void *start_func(void *arg)
{
        printf("start_func() just wait...\n");
        while (1) {
                sleep(1);
        }

        return NULL;
}
```

```
int main()
{
        pthread_t thread_t;
        int ret;

        ret = pthread_create(&thread_t, NULL, start_func, NULL);
        if (ret) {
                printf("pthread_create error: %s\n“, strerror(ret));
                return -1;
        }

        sleep(5);

        ret = pthread_cancel(thread_t);
        if (ret) {
                perror("pthread_cancel error: ");
                return -1;
        }

        printf("Thread canceled\n");
        return 0;
}
```

```
[root@localhost ch08]# gcc -g pthread_cancel_example.c -o pthread_cancel_example -pthread
[root@localhost ch08]# ./pthread_cancel_example
start_func() just wait...
Thread canceled
```

Chapter 08_04 스레드 대기

#include <pthread.h>

int pthread_join (pthread_t thread, void **retval);

pthread_join()은 thread 인자로 명시한(스레드 ID를 가진) 스레드가 종료될 때까지 대기하도록 한다.
해당 스레드가 이미 종료되었다면 pthread_join()은 즉시 리턴된다.

retval은 NULL 이 아닐 경우 종료한 스레드가 리턴한 값이다.

정상 종료 시 0 리턴, 에러 발생 시 errno 리턴
EDEADLK 데드락 감지
EINVAL thread는 조인 불가한 스레드
ESRCH thread 인자가 유효하지 않다

```
int ret;

ret = pthred_join (thread, NULL);
if (ret)
    printf("pthread_join error: %s\n", strerror(ret));
    return -1;
}
```

Chapter 08_04 스레드 대기

```
#include <pthread.h>

int pthread_detach (pthread_t thread);
```

기본적으로 스레드는 조인이 가능하도록 생성되지만,
pthread_detach()를 이용하면 조인이 가능하지 않도록 하는 것도 가능하다.
조인할 생각이 없는 스레드는 디태치하면 불필요한 시스템 자원을 낭비하지 않는다.

thread(스레드 ID)를 인자로 호출에 성공하면 0을 리턴,
실패 시 thread 인자가 유효하지 않다는 의미로 ESRCH 리턴

Chapter 08_05 스레드 동기화

스레드 생성의 경우 프로세스 생성과 달리 한 프로세스 내에서 생성된 여러 스레드들은 메모리 주소 공간을 공유한다.

한 프로세스 내에서 생성된 여러 프로세스들이 동시에 특정 전역 변수를 참조하는 경우를 생각해보자.

예를 들어 예금 계좌 잔액을 나타내는 total 변수가 존재하고, 현재 잔고가 1000원이 있다고 가정할 때,
동시에 두 개의 쓰레드에서 total 변수를 참조해서 100원을 인출하는 상황을 생각해보자.

첫 번째 스레드에서 현재 잔고가 1000원이므로 100원을 빼서 900원의 값을 total에 할당하는 중이고,
두 번째 스레드도 현재 잔고가 1000원인 시점에 참조해서 100원을 빼서 900원의 값을 total에 할당한다면,
실제로는 200원을 빼서 total 변수의 최종 값이 800원이 되어야 하지만 900원인 문제 상황이 발생할 수 있다.

위와 같이 동시에 공유 자원에 동시에 접근하는 상태를 "경쟁 상태"라 하고,
동시에 참조되는 공유 자원을 참조하는 영역, 즉 스레드가 실행 중 다른 스레드가 끼어들지 말아야 하는
영역을 "크리티컬 섹션"이라고 한다.
이 "크리티컬 섹션"을 "상호 배제(Mutual Exclusion)"하는 방식으로 접근을 동기화해야 한다.

pthread에서 상호 배제를 위해 제공하는 락(lock) 매커니즘으로 뮤텍스(mutex)가 존재한다.

Chapter 08_05 스레드 동기화

```
#include <pthread.h>

/* 뮤텍스 초기화 */
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

/* 뮤텍스 락 걸기 */
int pthread_mutex_lock (pthread_mutex_t *mutex);

/* 뮤텍스 락 풀기 */
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

뮤텍스는 pthread_mutex_t 자료구조로 표현된다.
정상수행 시 0 리턴, 에러 발생 시 0이 아닌 값 리턴 후 errno 설정
일반적으로 리턴 값을 검사하지 않는 경향이 있음

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *start_func(void *arg)
{
    pthread_mutex_lock(&mutex);
    /* 크리티컬 섹션 */
    pthread_mutex_unlock(&mutex);
}
```

Chapter 08_06 스레드 실습

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

static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int total_money;

void *withdrawal(void *arg)
{
        int money = *(int *)arg;

        pthread_mutex_lock(&mutex);
        printf("withdraw $%d from total balance\n", money);
        total_money -= money;
        sleep(3);
        printf("calculation finish. (withdraw $%d)\n", money);
        pthread_mutex_unlock(&mutex);

        pthread_exit(NULL);
}
```

Chapter 08_06 스레드 실습

```
int main()
{
        pthread_t thread1, thread2;
        int money1 = 100;
        int money2 = 200;

        total_money = 1000;
        printf("Total balance is $%d\n", total_money);

        pthread_create(&thread1, NULL, withdrawal, (void *)&money1);
        pthread_create(&thread2, NULL, withdrawal, (void *)&money2);

        pthread_join(thread1, NULL);
        pthread_join(thread2, NULL);

        printf("The remained balance is $%d\n", total_money);
        return 0;
}
```

```
[root@localhost ch08]# gcc -g pthread_example.c -o pthread_example -pthread
[root@localhost ch08]# ./pthread_example
Total balance is $1000
withdraw $200 from total balance
calculation finish. (withdraw $200)
withdraw $100 from total balance
calculation finish. (withdraw $100)
The remained balance is $700
```