9. 스레드 프로그래밍

- 멀티스레드 프로그래밍 소개 : Posix 표준 스레드
- 스레드 생성, 동기화 처리, 뮤텍스, 조건변수, 세마포어, 스레드 시그널

9.1 스레드 개요

스레드

- 프로세스처럼 독립적으로 수행되는 프로그램 코드
- 경량 프로세스라고도 함
- “프로세스 내에서 독립적으로 수행되는 제어의 흐름”이라고도 함
- 한 프로세스내의 스레드들은 전역 변수를 공유
  - 스레드간에 데이터 공유가 편리
  - 복잡한 프로세스간 통신(IPC) 기능을 사용하지 않아도 됨

스레드 생성과 종료

- pthread_create()
  - **스레드 생성**

```
int pthread_create (
  pthread_t *thread,                    // 생성된 스레드 ID가 리턴됨
  pthread_attr_t *attr,                 // 스레드 속성을 지정하는 인자
  void *(*start_routine) (void *),      // 스레드 시작함수(start routine)
  void *arg)                            // 스레드 시작 함수의 인자
```

  - start_routine **인자로 지정한 함수가 실행 (스레드 시작 함수)**
  - **스레드가 성공적으로 생성되면 스레드** ID**를** thread**로 리턴**
    - **프로그램에서는** pthread_t **타입의 스레드** ID**를 통해 접근 가능**
  - attr**은 생성되는 스레드의 속성을 지정**
    - **디폴트 값을 사용하려면** NULL**로 지정**
  - **성공시** 0**을 리턴, 에러 발생시 에러코드의 정수 값을 리턴**
    - **전역변수** errno**는 여러 스레드가 공유하므로 사용하지 않음**
  - **스레드별로** errno **사용이 가능**
    - **오버헤드가 높음**
    - **컴파일시** –D_REENTRANT **옵션 사용**

스레드 생성과 종료

- pthread_self()
  - **자신의 스레드** ID**를 얻음**

```
pthread_t pthread_self(void);
```

- pthread_exit()
  - **스레드 종료**

```
void pthread_exit(void *retval);
```

- **스레드 종료**
  - **프로세스와 마찬가지로** return**을 만나면 종료**
  - pthread_exit() **호출시 종료**
  - main() **함수**
    - **일종의 스레드로** init **스레드라 함**
    - main()**에서** return**을 만나면** main**에서 생성한 스레드들도 모두 종료**
    - main()**에서** pthread_exit()**를 호출하면** main **스레드만 종료하고 자식 스레드는 종료되지 않음** (**프로세스는 자식 스레드가 모두 종료할 때까지 기다림**)

스레드 생성과 종료

- pthread_join()
  - **자신이 생성한 자식 스레드가 종료할 때까지 기다림**

```
int pthread_join(pthread_t thrd, void **thread_return);
```

    - thrd
      - **종료를 기다릴 스레드의** ID
    - thread_return
      - **자식 스레드의 종료 상태가 저장됨**
      - NULL**로 하면 종료 상태를 받지 않음**
      - **자식 스레드의 종료값은** return, pthread_exit()**로 남길 수 있음**
        - » **자식스레드가 다른 스레드에 의해 종료되었다면 종료 값은** PTHREAD_CANCELED**가 된다**.
  - **자식 스레드가 종료되기 전에 부모 스레드가 먼저 종료되지 않게 할 때 주로 사용**
    - **부모 스레드가 종료하면 자식 스레드가 함께 종료되며, 자식 스레드의 작업이 모두 이루어 지지 않은 상태일 수 있음**
  - **자식 스레드의 종료 시점을 정확히 파악하여 어떤 작업을 하고 싶거나 종료 상태 값을 얻기 위해 사용**

Posix 스레드 라이브러리

- **스레드 프로그램을 작성하려면 스레드 라이브러리가 필요**
  - Poxis **라이브러리에 포함되어 있는** pthread **라이브러리를 사용**
- **시스템에 설치된** Posix **라이브러리 버전을 확인하는 예**

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

int main() {
     long version = sysconf( _SC_VERSION );
     printf("Posix version : %ld\n", version);
     if ( version >= 199506L)
        printf("이 시스템에서는 Posix 라이브러리 사용이 가능합니다\n");
     else
        printf("이 시스템에서는 Posix1003.1c 스레드를 지원하지 않습니다\n");
     return 0;
}
```

- **실행결과**

```
$ posix_ver
Posix version : 199506
이 시스템에서는 Posix 라이브러리 사용이 가능합니다
```

REENTRANT

- errno **를 전역변수가 아니라 스레드별로 고유의 변수로 사용할 때의 컴파일 옵션**(-D_REENTRANT)
  - #define _REENTRANT**를 선언한 효과**
  - **일반 함수들이 멀티스레드 환경에 적합하게 동작하도록 함**
  - **일반 함수 호출 시에** errno **변수나** perror() **함수를 그대로 사용할 수 있음**

스레드의 생성과 종료 예(thread_basic.c)

- pthread_create()와 pthread_self() 함수를 사용하여 스레드를 생성하고 생성된 스레드의 ID를 구하는 예
  - 컴파일 옵션 –lpthread는 pthread 라이브러리(libpthreaqd.so)를 링크시킴
  - 실행 결과

```
$ thread_basic
(child's main) Process ID = 20530
(child's main) Init thread ID = 1074008704
(parent's main) Process ID = 20529
(parent's main) Init thread ID = 1074008704
(parent' thread routine) Process ID = 20529
(parent' thread routine) Thread ID = 1082399936

(parent)[1082399936] 스레드가 종료했습니다
(child' thread routine) Process ID = 20530
(child' thread routine) Thread ID = 1082399936

(child)[1082399936] 스레드가 종료했습니다
```

스레드의 취소

- **다른 스레드의 실행을 종료시키는 것**
  - **스레드 종료를 위해서 취소요청을 전송**
  - **취소 요청을 수신한 스레드의 동작**
    - **취소요청 수용, 취소요청 무시**
  - **취소요청 수용**
    - **즉시 취소 : 취소 요청을 받는 즉시 스레드를 취소**
    - **지연 취소 : 처리중인 작업을 마치고 취소**
  - cannellcation point
    - **취소 요청을 받았을 때 즉시 처리할 수 없는 함수(코드)를 실행중인 상태**
    - **취소요청을 즉시 처리할 수 있는 함수(코드)를 수행중인 상태**
  - **스레드에서 취소 요청을 받아 들이는 방법**
    - Cancellation point**에서 취소요청을 받으면 항상 즉시 종료**
    - **즉시 취소** : cancellation point**가 아니어도 즉시 종료**
    - **지연 취소** : cancellation point**에 도달할 때까지 기다린 후 종료**
  - Cancellation point **함수 (즉시취소)**
    - read()/write(), open()/close(), fcntl(), sleep(), wait(), waitpid(), pthread_join()

스레드의 취소 관련 함수

- **스레드의 취소와 관련된 함수**
  - **취소요청의 수용 여부를 설정하는 함수**
  - **취소요청을 수용한 경우**
    - **지연취소** : canncellation point**까지 도달할 때까지 지연**
    - **즉시취소** : **취소 요청을 즉시 종료**
  - **취소요청의 수용여부를 설정하는 함수**

```
int pthread_setcancelstate(int state, int *oldstate)
```

    - state
      - PTHREAD_CANCEL_ENABLE : **취소요청 수용**
      - PTHREAD_CANCEL_DISABLE : **취소요청 무시**
    - oldstate
      - **이전의 취소 상태 값을 저장하기 위한 변수**
      - NULL **설정시 취소 상태값을 저장하지 않음**
    - **성공하면** 0**을 반환**

스레드의 취소 관련 함수

- **취소요청을 수용할 경우** cancellation point**까지 요청을 연기할 것인지 아니면 즉시 종료할 것인지 설정하는 함수**

```
int pthread_setcanceltype(int type, int *oldtype)
```

  - type
    - PTHREAD_CANCEL_ASYNCHRONOUS : **즉시 취소**
    - PTHREAD_CANCEL_DEFERRED : cancellation point**까지 지연**
  - oldstate
    - **이전의 취소** type**을 저장하기 위한 변수**
    - NULL **설정시 이전의** type **값을 저장하지 않음**
  - **성공하면** 0**을 반환**

스레드의 취소 관련 함수

- pthread_testcancel()

```
void pthread_testcancel()
```

  - **지연 취소 결정 후** cancellation point**를 만나지 못하면 계속 지연**
  - **현재 도착한 취소요청이 있는지를 검사**
    - **취소요청이 있었으면 스레드를 종료시킴**
  - cancellation point**가 없는 코드에 강제로** cancellation point**를 넣어주는 효과를 가짐**

```
// cancellation point가 없는 코드
for ( ; ; )
  count++;
```

```
// cancellation point를 추가한 코드
for ( ; ; )
{
  count++;
  if (count%100 == 0)
    pthread_testcancel();
}
```

취소요청을 무시하는 예제 프로그램(thcancel_dis.c)

- **스레드의 취소 환경을** PTHREAD_CANCEL_DISABLE**로 설정하고 2천만번의 루프를 실행**
  - **매** 5**십만 번 마다** pthread_testcancel()**로 취소요청을 확인**
    - PTHREAD_CANCEL_DISABLE**이므로 취소요청을 무시**
  - cancel_and_join()**은 취소 요청을 보내고 스레드의 종료를 기다린다.**

```
Cancel_and_join()
{
        pthread_cancel(tid);          //최소 요청을 보냄
        pthread_join(tid, &result);   // 해당 스레드가 종료하기를 기다림
}
```

  - **실행 결과**

```
$ threadcancel_dis
** PTHREAD_CANCEL_DISABLE
[Thread ID=1082399936] thread is not canceled
총 20000000 번의 루프중 20000000 번의 루프를 돌았음
```

지연 취소

- thcancel_dis.c **에서 취소 요청을 지연하도록 수정**

```
void *thrfunc(void *arg)
{
    curthd = pthread_self();
    // 취소 요청을 허용하도록 설정
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);

    // 취소 요청을 calcellation point까지 지연
    pthread_setcanceltype(PTHREAD_CANCEL_DEFFERED, NULL);
    // 500,000 번 마다 취소 요청이 도착했는지 검사
    for(cnt=1; cnt<max_loop; cnt++)
        if(cnt%500000 === 0)  pthread_testcancel();
}
```

- **실행결과**

```
$ thcancel_def
** PTHREAD_CANCEL_ENABLE
[Thread ID=1082399936] thread is canceled
총 20000000 번의 루프중 650000 번의 루프를 돌았음
```

즉시 취소

- thcancel_dis.c **에서 즉시 요청을 취소하도록 수정**

```
void *thrfunc(void *arg)
{
    curthd = pthread_self();
    // 취소 요청을 허용하도록 설정
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);

    // 취소 요청을 즉시 수용
    pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
}
```

- **실행결과**

```
$ thcancel_async
** PTHREAD_CANCEL_ENABLE
[Thread ID=1082399936] thread is canceled
총 20000000 번의 루프중 650000 번의 루프를 돌았음
```

스레드의 상태

- **스레드의 상태**
  - **준비**(ready) : **스레드가 실행될 수 있는 상태**
  - **실행**(running) : CPU**의 서비스를 받고 있는 상태**
  - **블록**(block) : sleep(), read(), **세마포어 연산 등으로 기다리는 상태**
  - **종료**(terminated) : **스레드가 종료 또는 취소된 상태**

스레드의 상태

- 준비 상태
  - 스레드가 처음 생성되면 준비 상태로 감
  - CPU의 서비스를 받을 수 있는 상태
- 실행 상태
  - 운영체제의 스케줄링에 의해 CPU의 서비스를 받고 있는 상태
  - 스케줄링에 의해 준비상태로 갈 수 있음
- 블록 상태
  - 실행 상태에서 즉시 처리할 수 없는 작업을 만나는 경우
    - read(), wait(), sleep(), 세마포어/뮤텍스/조건변수를 기다리는 경우
  - 원하는 조건이 만족되면 준비 상태가 됨
- 종료 상태
  - 스레드 시작함수에서 return 또는 pthread_exit()가 호출된 경우
  - 사용하던 자원의 대부분을 시스템에 반납
  - 종료 상태 값을 부모 스레드에게 넘겨주기 위한 자원은 부모 스레드에서 pthread_join()을 통해 읽어가면 완전히 삭제

분리된 스레드와 joinable 스레드

- 분리된 스레드
  - 데몬 프로세스처럼 부모 스레드와 분리된 스레드로 실행되는 스레드
  - pthread_detach()가 호출된 경우
  - 이 스레드에 대해서 pthread_join()을 호출할 수 없음
  - 종료될 경우 종료 상태에 남아있지 않고 리소스를 모두 반납
- Joinable 스레드
  - 디폴트 스레드
  - 부모 스레드는 pthread_joing()을 호출하여 이 스레드의 종료를 기다릴 수 있음
  - Joinable 스레드에서 pthread_detach()를 호출하면 분리된 스레드가 됨

스레드의 동기화 문제

- **동기화 문제**
  - **여러 스레드가 공유 데이터를 액세스할 때 공유 데이터의 값을 정확히 예측할 수 없는 문제**
- **해결 방법**
  - **스레드들이 서로 배타적으로 공유데이터에 접근하도록 함**
  - mutex(mutual+exclusion)**를 사용**

멀티스레드 실행환경에서 발생하는 동기화 문제 예

- race.c 에서 두 개의 스레드를 생성
  - 각 스레드는 prn_data(pthread_self())를 반복해서 호출
  - prn_data()에서는 공유변수 who_run에 자신의 스레드 ID를 저장
  - 공유변수에 저장된 스레드 ID와 자신의 스레드 ID를 변경되었는 지를 비교
  - 한 스레드가 prn_data()를 호출하여 수행하는 동안 다른 스레드가 prn_data()를 호출하면 who_run이 바뀔 수 있다.
- 실행결과

```
$ race
Error : 1082399936스레드 실행중 who_run=-1
Error : 1082399936스레드 실행중 who_run=-1
Error : 1082399936스레드 실행중 who_run=1090788416
Error : 1090788416스레드 실행중 who_run=-1
Error : 1090788416스레드 실행중 who_run=1082399936
Error : 1090788416스레드 실행중 who_run=-1
Error : 1082399936스레드 실행중 who_run=-1
Error : 1082399936스레드 실행중 who_run=-1
.....
```

9.2 뮤텍스

뮤텍스

- 커널이 제공하는 일종의 플래그
- 멀티스레드 프로그램에서 동기화문제를 해결
- 뮤텍스외에도 세마포어 사용이 가능
  - 뮤텍스 사용이 간편하여 널리 사용

뮤텍스 사용 방법

- 크리티컬 영역(공유데이터)에 들어가기 전에 뮤텍스를 잠금
- 크리티컬 영역에서 나오면서 뮤텍스 잠금을 해제
- 다른 스레드들은 뮤텍스 잠금이 해제될 때까지 대기
- 뮤텍스 사용 절차

```
pthread_mutex_t mutex;                  // 뮤텍스 선언
          ...                           // 뮤텍스 초기화 작업
pthread_mutex_lock(mutex);              // 뮤텍스 잠금
          ...                           // 공유데이터(크리티컬 영역) 액세스
pthread_mutex_unlock(mutex);            // 뮤텍스 잠금 해제
```

```
// race.c에서 동기화 문제를 뮤텍스를 사용하여 해결하는 방법
// 뮤텍스 선언 및 초기화
pthread_mutext_t lock = PTHREAD_MUTEX_INITIALIZER;

void *thrfunc(void *arg) {
  while (1) {
    pthread_mutex_lock(&lock);          // 뮤텍스 잠금
    prn_data(pthread_self());           // 공유데이터 액세스
    pthread_mutex_unlock(&lock)         // 뮤텍스 잠금 해제
  }
  return NULL;
}
```

뮤텍스 선언 및 초기화

- **뮤텍스 변수 타입** : pthread_mutex_t
- **여러 스레드들이 사용하므로 전역 변수로 선언**
- **반드시 초기화를 해야 사용 가능**
- **초기화 방법**
  - **함수를 통한 초기화**
  - **매크로를 통한 초기화**

```
// 매크로를 이용한 초기화
// 기본 속성만 갖는 뮤텍스를 만들 경우 사용
pthread_mutext_t mutex = PTHREAD_MUTEX_INITIALIZER;

// 함수를 이용한 초기화
// attr 인자를 사용하여 다양한 속성을 갖는 뮤텍스를 만들 경우 사용
// attr에 NULL을 지정할 경우 기본 속성의 뮤텍스 생성
pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
```

뮤텍스 선언 및 초기화

- attr **인자**
  - pthread_mutexattr_settype() **사용**

```
// 뮤텍스에 속성을 지정하는 순서
pthread_mutex_t mutex;                      // 뮤텍스 변수 선언
pthread_mutexattr_t attr;                   // 뮤텍스 속성 변수 선언

pthread_mutexattr_settype(&attr, type);     // 뮤텍스 타입을 type에 지정

pthread_mutex_init(&mutex, &attr);          // 속성을 뮤텍스에 적용하여 초기화
```

  - type
    - PTHREAD_MUTEX_TIMED_NP
      - timed **타입**
    - PTHREAD_MUTEX_RECURSIVE_NP
      - recursive **타입**
    - PTHREAD_MUTEX_ERRORCHECK_NP
      - error checking **타입**

뮤텍스의 종류

- 기본 뮤텍스
  - 뮤텍스 초기화에 특별한 타입을 지정하지 않은 뮤텍스
  - 단 한 번의 잠금만 허용
    - 스레드 A가 잠금을 하고 있는 상황에서 스레드 B가 잠금을 시도하면 B는 A의 잠금이 해제될 때까지 기다림
    - A에 의해 잠긴 뮤텍스에 대해 A가 다시 잠금을 시도하면 데드락이 발생하여 A는 영원히 블록 상태가 됨
    - 이러한 현상을 피하기 위해 pthread_mutex_trylock()을 사용
      - 뮤텍스를 얻지 못하는 경우 블록되지 않고 바로 리턴되고 EBUSY 에러 발생
  - 다른 스레드가 뮤텍스의 잠금 해제가 가능
  - 잠금되어 있는 않은 뮤텍스에 잠금해제를 시도하면 아무 동작을 하지 않음
- Timed 타입 뮤텍스
  - pthread_mutex_lock()은 뮤텍스를 얻을 때까지 블록
  - Timed 타입 뮤텍스는 pthread_mutex_timedlock() 함수를 사용
  - 지정한 시간 동안만 블록됨
  - 리눅스의 기본 뮤텍스는 timed 타입 뮤텍스

뮤텍스의 종류

- Recursive **타입 뮤텍스**
  - **한 스레드가 한 번 이상 잠금이 가능**
  - pthread_mutex_lock()**을 호출한 수만큼** pthread_mutex_unlock() **을 호출해야 함**
  - **다른 스레드가 잠금 해제를 시도하면 에러 발생**
- Error check **타입 뮤텍스**
  - **기본 뮤텍스에서 한 스레드가 두번 잠금을 시도하면 무한 블록**
  - Error check **타입 뮤텍스는 두번 잠금을 시도하면 에러가 발생하면서 프로세스 종료**
  - **잠금되어 있지 않은 뮤텍스에 잠금 해제를 시도하면 에러가 발생하면서 프로세스 종료**

뮤텍스의 삭제

- 더 이상 뮤텍스를 사용하지 않을 때 삭제
- 잠금 상태의 뮤텍스를 삭제하려하면 에러 발생
  - EBUSY 에러 발생
- 뮤텍스의 현재 상태(잠금, 잠금해제) 확인 함수는 없음
- 안전하게 뮤텍스를 삭제하기 위해서는 삭제전에 무조건 뮤텍스를 해제해야 함

```
pthread_mutex_unlock(&mutex);
pthread_mutex_destory(&mutex);
```

뮤텍스와 데드락

- 데드락
  - 두 개의 스레드가 두 개의 뮤텍스를 사용
  - 각 스레드가 뮤텍스를 하나씩 잠그고 있는 상태에서 상대방의 뮤텍스 해제를 기다리는 현상
  - 데드락이 발생하면 프로그램은 영원히 블록
- 데드락을 피하는 방법
  - 뮤텍스를 잠그는데 일정한 순서를 정하여 사용
- 성능
  - 뮤텍스를 많이 사용하면 프로그램의 성능이 저하됨
  - 뮤텍스를 얻은 상태에서 처리하는 작업의 량(크리티컬 영역)을 최소화 해야 함
    - 뮤텍스가 잠긴 동안 다른 스레드가 블록될 가능성이 높으며 크리티컬 영역이 길수록 전체 프로그램 성능이 저하될 수 있음

뮤텍스 사용예(mutextest.c)

- **뮤텍스를 사용하여 한 스레드가** counting **변수를 접근하는 동안 다른 스레드가 그 변수에 접근하지 못하는 것을 확인**
  - Main**에서 두개의 스레드를 만들고 스레드 시작 함수에서는 뮤텍스를 이용하여** 1**코 간격으로** counting **값을 출력**
- **실행결과**

```
$mutextest

[1082399936 스레드] 뮤텍스 잠금
[1082399936 스레드] counting = 0
[1082399936 스레드] 뮤텍스 해제

[1082399936 스레드] 뮤텍스 잠금
[1082399936 스레드] counting = 1
[1082399936 스레드] 뮤텍스 해제

[1082399936 스레드] 뮤텍스 잠금
[1082399936 스레드] counting = 2
[1082399936 스레드] 뮤텍스 해제
.....
```

스레드의 Cleanup 핸들러

- **스레드의 종료**
  - **스레드 시작함수에서** return
  - pthread_exit() **호출**
  - **다른 스레드에 의해 취소**
- cleanup **핸들러**
  - **스레드 종료 직전에 어떤 작업을 처리하도록 하는 방법**
  - **프로세스의** atexit() **함수와 유사**
    - **프로세스 종료시 자동으로 수행시킬 함수를 등록**
- cleanup **핸들러가 필요한 경우**
  - **스레드의 종료전 반드시 뮤텍스의 잠금을 해제해야 하는 경우**

cleanup 핸들러 등록

- **핸들러 등록과 해제 함수**

```
void pthread_cleanup_push(void (*routine) (void*), void *arg);
void pthread_cleanup_pop(int execute);
```

  - pthread_cleanup_push()
    - routine : cleanup **핸들러 함수**
    - arg : cleanup **핸들러로 넘겨주는 인자**
  - pthread_cleanup_pop()
    - execute**가 0이면 핸들러를 제거하는 동작만 수행**
    - execute**가 0이 아니면 핸들러를 한번 수행한 후 제거**
- **뮤텍스를 잠근 상태에서 스레드가 종료되는 경우를 대비한 코드**

```
pthread_cleanup_push(pthread_mutex_unlock, (void *)&mutex);
pthread_mutex_lock(&mtuex);
//크리티컬 영역 처리
pthread_mutex_unlock(&mutex);
pthread_cleanup_pop(0);
```

- **여러 개의** Cleanup **핸들러 등록이 가능**
  - **가장 나중에 등록된 핸들러가 가장 먼저 수행**

9.3 스레드의 조건변수

조건변수 사용 방법

- 조건변수
  - 공유데이터에 대한 정보를 스레드간의 통신을 위해 사용
- 조건변수의 필요성
  - 큐를 사용하는 경우 예
    - 생산자 스레드는 큐에 메시지 쓰기 작업을 수행
    - 소비자 스레드는 큐에서 메시지 읽기 작업을 수행
    - 운영체제(스케줄려)는 생산자와 소비자 스레드가 교대로 실행되는 것을 보장하지 않음
    - 큐가 비어있을 때 소비자 스레드가 수행하면 메시지를 읽을 수 없음
      - 큐에 메시지가 기록된 사실을 소비자에게 알려주는 조건알림(signal)
    - 큐가 차있을 때 생산자 스레드가 수행하면 메시지 쓰기를 할 수 없음
      - 큐에 빈공간이 있음을 소비자에게 알려주는 조건알림(signal)

조건변수의 동작

- **조건변수는 반드시 뮤텍스와 함께 사용되야 함**
- **조건변수를 기다리는 함수**

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

  - cond : **조건변수, 조건알림이 발생할 때까지 대기**
  - mutex : **이** mutex**에 의해 조건변수** cond**가 제어됨**
  - **동작**
    - **뮤텍스 해제** : **생산자가 큐에 메시지를 기록하게 함**
    - wait **함수의 블록 상태**
    - **조건알림을 받음** : **생산자가 큐에 데이터를 기록한 경우**
    - wait **함수가 깨어나면서 뮤텍스를 얻음(잠금)**

- pthread_cond_timedwait()
  - **지정된 시간동안만 블록 상태에서 조건 알림을 기다림**

조건변수의 동작

- **조건 알림을 보내는 함수**

```
int pthread_cond_signal(pthread_cond_t *cond);
```

  - **지정한 조건 변수(cond)에 대해 조건 알림이 발생하기를 기다리는 스레드에게 조건 알림을 보냄**

- **하나의 생산자 스레드와 여러개의 소비자 스레드인 경우**
  - pthread_cond_signal() **함수를 사용하면 단 하나의 스레드만 깨우고 나머지 스레드는 영원히 기다릴 가능성이 있음**
  - pthread_cond_broadcast()**를 사용**
    - **여러 스레드를 동시에 깨움**
    - **깨어난 스레드들 중 공유 데이터를 위한 뮤텍스를 얻은 스레드만이 공유 데이터에 접근**

조건변수의 생성과 삭제

- **조건변수의 생성**
  - **조건변수를 사용하기 전에 초기화**
  - **매크로를 이용한 초기화와 함수를 이용한 초기화 방법이 있음**

```
// 매크로를 이용한 초기화 방법
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

// 함수를 이용한 초기화 방법
pthread_cond_t cond;
pthread_cond_init(&cond, NULL);//첫번째 인수는 조건변수, 두번째는 조건변수의 속성
                               // NULL 이면 기본속성으로 초기화
```

- **조건변수의 삭제**
  - **더 이상 사용하지 않는 조건변수는 삭제**

```
pthread_cond_destory(&cond);
```

조건변수의 사용 예(cond.c)

- **조건변수를 사용하여 두 스레드가 정보를 교환하는 예**
  - **부모스레드**(main)**는 자식 스레드를 생성한 후 공유데이터**(data)**가** 1**이 되기를 기다린다**.
  - **자식 프로세스는 주어진 시간 동안** sleep() **후** data**를** 1**로 변경 후 부모 스레드에게 조건알림을 보낸다**.
  - **편의상 뮤텍스, 조건변수, 공유데이터를 정의한 구조체를 사용**

```
typedef struct _complex {
    pthread_mutex_t mutex;     // 뮤텍스
    pthread_cond_t cond ;      // 조건변수
    int value;                 // 공유데이터
} thread_control_t;
```

- **실행결과**

```
$ cond 5
Condtion wait time out
$ cond 1
Wait on Condition.....
Condition was signaled.
```

스레드의 세마포어

- **스레드간의 동기화**
  - **뮤텍스와 조건변수 이용**
  - **세마포어 이용**
    - **프로세스간 통신의 세마포어와 유사하나 사용법은 다름**
- **스레드용 세마포어**
  - sem_t**를 사용**
  - **스레드용 세마포어 초기화는** sem_init() **함수를 사용**

```
sem_t *sem;
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

    - sem : **세마포어 객체**
    - pshared : **다른 스레드와 공유할 것인지의 여부 (리눅스는 지원하지 않음)**
    - value : **세마포어의 초기값**
- **세마포어를 기다리는 함수**

```
// 세마포어가 0이면 블록, 1로되면 블록에서 나와 크리티컬로 진입
int sem_wait(sem_t *sem);
// 넌블록 버전, 0이어도 블록되지않고 EAGAIN 에러와 함께 즉시 리턴
int sem_trywait(sem_t *sem);
```

  - **크리티컬 영역에 진입하면서 자동으로 세마포어의 값을** 0**으로 설정**

스레드의 세마포어

- **세마포어의 값을 1 증가시키는 함수**

```
int sem_post(sem_t *sem);
```

- **현재 세마포어의 값을 읽는 함수**

```
int sem_getvalue(sem_t *sem, int *sval);
```

- **세마포어를 삭제하는 함수**

```
int sem_destory(sem_t *sem);
```

9.4 스레드의 시그널 처리

스레드 시그널 처리 개요

- 시그널
  - 프로세스 사이에 이벤트의 발생을 알려주는 수단
    - 프로세스를 대상으로 전달
    - 프로세스 내의 특정 스레드를 목적지로 하여 전달 되지는 않음
  - 멀티스레드 프로그램에서는 스레드가 시그널을 수신 가능하도록 프로그램을 작성하지 않음
    - 여러 스레드 중 어떤 스레드가 시그널을 받을지 예측할 수 없음
  - 프로세스에서 시그널 핸들러를 설정할 경우 프로세스 내의 스레드들은 시그널 핸들러를 공유
    - 어떤 스레드가 시그널을 받아 처리해도 동일한 시그널 핸들러가 적용됨
  - 시그널 블록 마스크
    - 각 스레드별로 정의 할 수 있으며 스레드마다 특정 시그널의 수신/블록을 선택할 수 있음
  - 멀티스레드 프로그램에서의 시그널 처리
    - 시그널과 관계 없는 스레드는 모든 시그널을 블록
    - 시그널 처리를 전담하는 스레드를 지정하여 시그널을 처리
  - 멀티스레드 프로그램에서는 시그널을 주의하여 사용
    - 시그널처리가 복잡
    - 예상치 못한 동작을 할 수 있음

스레드 시그널 처리 함수

- **스레드에 시그널 블록 마스크를 설정 함수**

```
int pthread_sigmask(int how, const sigset_t *netmask, sigset_t *oldmask);
```

  - how
    - SIG_SETMASK
      - **시그널 블록 마스크 인자를** newmask **값으로 설정**
    - SIG_BLOCK
      - **시그널 블록 마스크에** newmask**를 추가**
    - SIG_UNBLOCLK
      - **시그널 블록 마스크에서** newmask**를 제거**
- **스레드에서 다른 스레드에게 시그널을 보내는 함수**
  - **같은 프로세스 내의 스레드에게만 시그널을 전달 할 수 있음**
  - **다른 프로세스에게 시그널을 보내기 위해서는** kill() **함수를 사용**

```
int pthread_kill(pthread_t thread, int signo);
```

    - thread : **시그널을 수신할 스레드** ID
    - signo : **전송할 시그널 번호**

멀티스레드 에코서버(mth_echoserv.c)

- 수신스레드 echo_recv()는 클라이언트로부터 메시지를 수신하여 메시지큐에 넣는 작업을 수행
- 송신스레드 echo_resp()는 메시지큐로부터 메시지를 읽어 클라이언트에 응답
- 송신 및 수신 스레드를(5개)를 생성
  - 임의의 스레드가 랜덤하게 에코 서비스를 수행
- 수행결과

```
$ mth_echoserv 1234 9909
recv tgread ID = 1026
response thread = 6151
recv tgread ID = 3076
response thread = 6151
recv tgread ID = 5126
response thread = 6151
recv tgread ID = 2051
response thread = 6151
recv tgread ID = 4101
response thread = 6151
```