쓰레드
Thread

- Thread 기본
- Pthread
- Thread 생성과 소멸
- Thread 동기화
- 공유 변수
- 상호 배제

Thread 기본

❑ Thread?

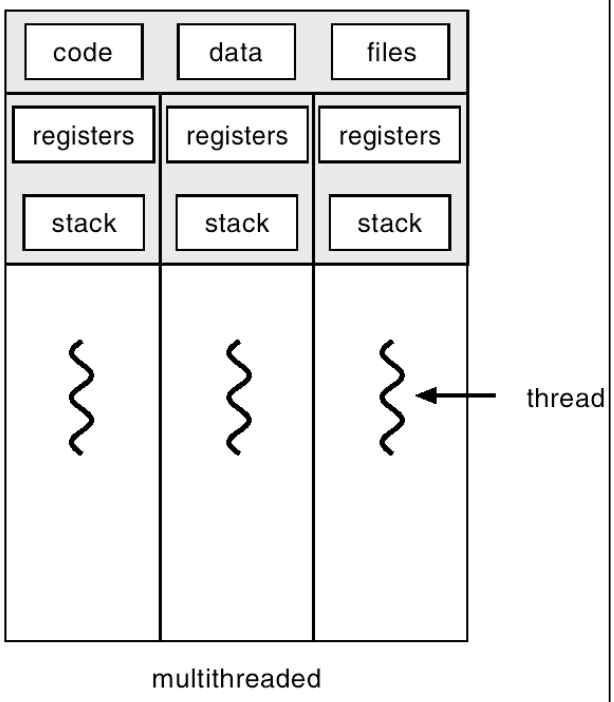
- 경량 프로세스(lightweight process: LWP)
  - 일반 프로세스는 생성 시 자신만의 메모리 영역을 할당받는다
    - PCB, code, static, heap, stack 등
  - Thread: PCB와 스택만 별도로 할당받고 나머지는 부모 프로세스와 공유
  - 생성과 전환(context switch) 시 프로세스보다 오버헤드가 적다.
- 대부분 여러 쓰레드가 하나의 프로세스가 되도록 운영
  - OS에서 운영하기 나름
    - 새로운 프로세스 생성 시 무조건 쓰레드 생성 → exec 등으로 변경 사항 있을 때만 새 메모리 할당
    - 프로세스 == 쓰레드 그룹

Thread 기본

❑ Multi-thread

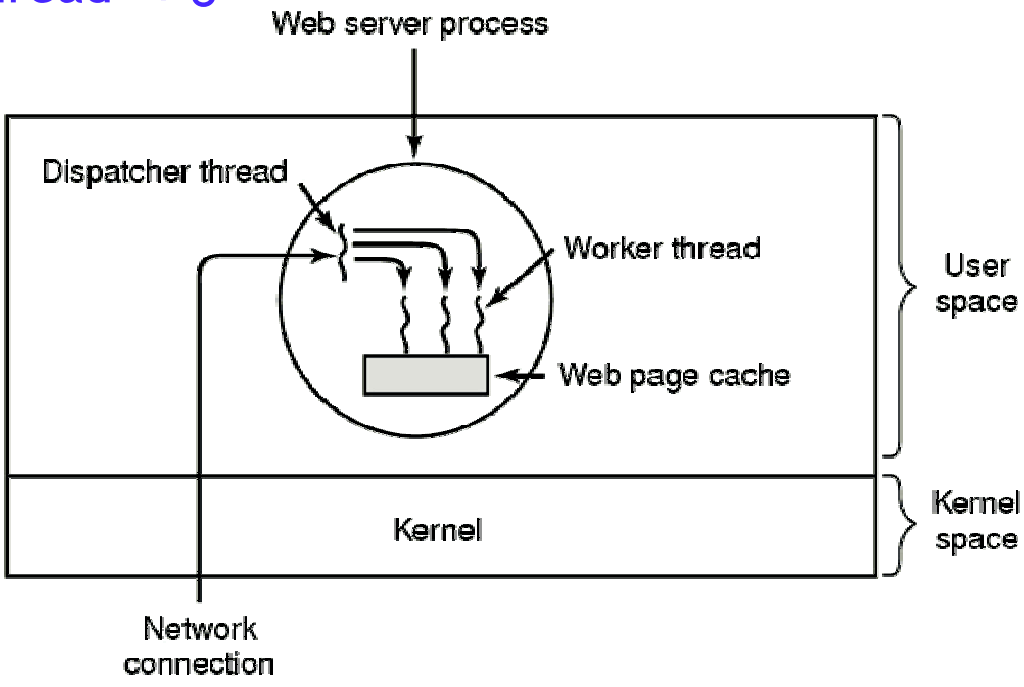
- 여러 개의 병행 프로세스처럼 독립적으로 동작
  - 별도 스케줄 가능
  - 지역 변수는 별도로 사용, 전역 변수는 공유
- 자원을 공유하므로 효율적

☺ 하나의 프로그램 내에서 여러 개의 함수가 병행하여 실행되는 것으로 이해



Thread 기본

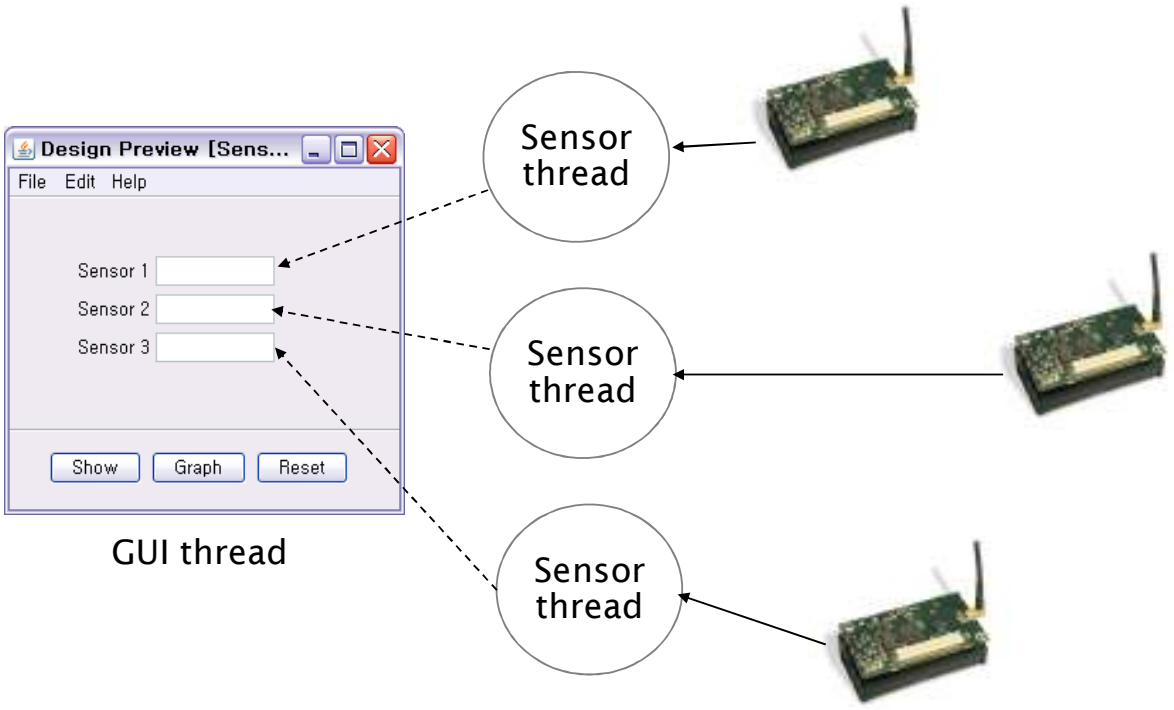
❑ Thread 사용 예



A multithreaded web server

Thread 기본

❑ 쓰레드 사용 예



Pthread

❑ POSIX thread

- IEEE POSIX 1003.1c (1995)에서 표준 thread API 제정
  - 실제 구현은 하드웨어 및 OS에 따라 다름
- 60개가 넘는 함수로 구성됨

❑ Pthread 사용한 프로그래밍

- 소스 내

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

- 컴파일 옵션

```
$ gcc -pthread [other options] source
```

Pthread

❑ 함수 분류

- 쓰레드 관리: pthread_ , pthread_attr_
  - 생성, 분리, 결합
  - 속성 관리
- 상호 배제: pthread_mutex_ , pthread_mutexattr_
  - 쓰레드 간 동기화, 공유 변수 관리에 사용
  - Mutex 영역 생성, 소멸, lock/unlock
  - Mutex 관련 속성의 설정 또는 변경
- 조건 변수: pthread_cond_, pthread_condattr_
  - mutex를 공유하는 쓰레드 간 통신에 사용
  - 고급 동기화 구조 생성
- 기타: pthread_kill, pthread_sigmask, …

쓰레드 관리 함수

함수	의미
pthread_create	쓰레드 생성
pthread_exit	쓰레드 종료
pthread_attr_init	쓰레드 속성 초기화
pthread_attr_destroy	쓰레드 속성 제거
pthread_join	특정 쓰레드가 종료하여 결합할 때까지 대기
pthread_detach	쓰레드 분리
pthread_attr_setdetachstate	분리상태에 대한 속성 설정
pthread_attr_getdetachstate	분리상태에 대한 속성 질의
pthread_self	자신의 쓰레드 ID 반환

쓰레드 생성과 소멸

❑ 쓰레드 생성과 소멸

```
#include <pthread.h>

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

thread	생성이 성공할 경우 생성된 쓰레드의 ID
attr	속성 객체. 기본 속성을 사용하려면 NULL로 설정
start_routine	쓰레드로 분기하여 실행할 함수
arg	쓰레드 함수에 전달할 인자. 사용 시 필요한 타입으로 변환
반환값	호출이 성공하면 0, 실패하면 0이 아닌 에러 코드 반환
retval	쓰레드 종료 시 반환값의 포인터.

쓰레드 생성과 소멸

```
void *thread (void *data)
{
   int id = *(int *) data;
   sleep (id);
   printf("Hello! I'm thread #%d\n", id);
   pthread_exit(NULL);
}
```

```
int main (void)
{
   pthread_t threads[5];
   int rc, t;

   for (t = 0; t < 5; t++) {
       printf("In main: creating thread %d\n", t);
       rc = pthread_create(&threads[t], NULL, thread, (void *)&t);
       if (rc) { printf("Thread creation error!\n"); exit(1); }
   }
   pthread_exit(NULL);
}
```

```
juyoon@linda: ~/test/system
juyoon@linda:~/test/system$ pth3
In main: creating thread 0
Hello! I'm thread #0!
In main: creating thread 1
In main: creating thread 2
In main: creating thread 3
In main: creating thread 4
Hello! I'm thread #1!
Hello! I'm thread #2!
Hello! I'm thread #3!
Hello! I'm thread #4!
juyoon@linda:~/test/system$
```

쓰레드 생성과 소멸

❑ 인자 전달

- 어떤 타입이든 가능 - void * 로 만들어 전달
- 인수 시점에 주의해야 한다.

```
void *thread (void *data)
{
   int id;
   sleep (1);
   id = *(int *) data;
   printf("Hello! I'm thread #%d\n", id);
   pthread_exit(NULL);
}
```

```
juyoon@linda: ~/test/system
juyoon@linda:~/test/system$ pthw
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Thread #5
Thread #5
Thread #5
Thread #5
Thread #5
```

- 쓰레드 함수가 인자를 전달받기 전에 main이 종료하면 쓰레드 실행에 문제가 생긴다.

쓰레드 생성과 소멸

❑ 속성 (Attributes)

- default 속성 사용: NULL 지정
- 별도의 attr 객체 사용
  - attr 객체를 생성하고 default로 초기화
    pthread_attr_init (pthread_attr_t *attr)
  - attr 객체 제거
    pthread_attr_destroy (pthread_attr_t *attr)
- 속성의 종류
  - 분리 상태: PTHREAD_CREATE_JOINABLE, PTHREAD_CREATE_DETACHED
  - 스케줄링 정책: PTHREAD_INHERIT_SCHED, PTHREAD_EXPLICIT_SCHED
  - 쓰레드 영역: PTHREAD_SCOPE_SYSTEM, PTHREAD_SCOPE_PROCESS
  - 스택 주소
  - 스택 크기
  - 그 외

쓰레드 생성과 소멸

pthread_exit

- 쓰레드 함수 마지막에 써 준다.
  - cleanup 함수가 정의되어 있을 경우 그를 실행하고 종료 (pthread_cleanup_push 함수 사용하여 정의)
  - 쓰레드 종료 시에는 개방된 파일, 동적 할당된 메모리에 대한 처리를 시스템이 자동으로 해 주지 않는다. 프로그래머 책임!
- main 함수 (쓰레드를 생성한 함수) 마지막에 써 준다.
  - main이 종료해도 생성한 쓰레드가 종료하지 않도록 한다.

쓰레드 ID

쓰레드 ID

- 생성 시 시스템이 부여해 주는 고유 번호
- 부모 쓰레드는 pthread_create의 인자로 ID를 받는다.
- 생성된 쓰레드 자신은 다음 함수를 호출

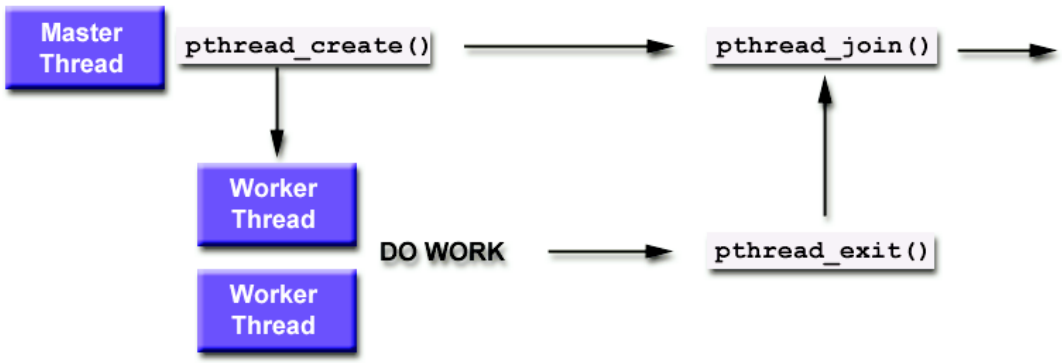
```
#include <pthread.h>

pthread_t pthread_self (void);
```

반환값	호출한 쓰레드의 ID를 반환

쓰레드 동기화

❑ 쓰레드 종료 대기



#include <pthread.h> int pthread_join (pthread_t thread, void **thread_return);	
thread	기다릴 쓰레드의 ID
thread_return	쓰레드가 종료 시 반환하는 값을 받아 올 포인터
반환값	호출이 성공하면 0, 실패하면 0이 아닌 에러 코드 반환

쓰레드 동기화

juyoon@linda: ~/test/system

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

void *thread (void *data)
{
    int num = *(int*) data;
    printf("num %d₩n", num);
    sleep(1);
    return (void *)(num*num);
}

int main (void)
{
    pthread_t th;
    int tid;
    int status;
    int a = 100;

    tid = pthread_create(&th, NULL, thread, (void *)&a);
    pthread_join (th, (void*)&status);
    printf("Thread join: %d₩n", status);
    return 0;
}
```

juyoon@linda: ~/test/system

```
juyoon@linda:~/test/system$ pthj
num 100
Thread join: 10000
juyoon@linda:~/test/system$
```

쓰레드 동기화

❑ 쓰레드 분리

#include <pthread.h> int pthread_detach (pthread_t thread);	
thread	분리할 쓰레드의 ID
반환값	호출이 성공하면 0, 실패하면 0이 아닌 에러 코드 반환

- 분리된 쓰레드는 join할 수 없다.
  - 독립적으로 수행 후 종료
- pthread_join 또는 pthread_detach가 필요한 이유
  - 쓰레드가 사용한 자원을 <명시적으로> OS에 되돌려 준다.
- 기본은 join?
  - 속성이 PTHREAD_CREATE_JOINABLE로 된 경우
  - 시스템에 따라 다를 수 있다.

쓰레드 동기화

```
void *work (void *null)
{
    int i;
    double result=0.0;
    for (i = 0; i<100000; i++)
        result += (double)random();
    printf("result = %e\n", result);
    pthread_exit((void*) 0);
}

int main (void)
{
    pthread_t thread[3];
    pthread_attr_t attr;
    int rc, t;
    void *status;

    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    for (t=0; t<3; t++) {
        printf("Creating thread %d\n", t);
        rc = pthread_create(&thread[t], &attr, work, NULL);
    }
    pthread_attr_destroy(&attr);
    for (t=0; t<3; t++) {
        rc = pthread_join(thread[t], &status);
        printf("Completed join with thread %d status = %ld\n",
                t, (long)status);
    }
    pthread_exit(NULL);
}
```

```
juyoon@linda: ~/test/system
juyoon@linda:~/test/system$ ptha
Creating thread 0
result = 1.073459e+14
Creating thread 1
Creating thread 2
Completed join with thread 0 status = 0
result = 1.074103e+14
Completed join with thread 1 status = 0
result = 1.074218e+14
Completed join with thread 2 status = 0
juyoon@linda:~/test/system$
```

공유 변수

❑ 쓰레드의 공유 변수

- 각 쓰레드는 전역 변수와 static 변수를 공유한다.

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

void *thread(void *data);
char **ptr;

int main (void)
{
    int i;
    pthread_t tid;
    char *msgs[2] = {"Hello from 1", "Hello from 2"};

    ptr = msgs;
    for (i=0; i<2; i++)
        pthread_create(&tid, NULL, thread, (void *)i);
    pthread_exit(NULL);
}

void *thread(void *data)
{
    int myid = *(int *) data;
    static int cnt = 0;
    printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
}
```

```
juyoon@linda: ~/test/system
juyoon@linda:~/test/system$ pth
[0]: Hello from 1 (cnt=1)
[1]: Hello from 2 (cnt=2)
juyoon@linda:~/test/system$
```


공유 변수

❑ 공유 변수 사용 시의 문제점

```c
#define NITERS 100000000
unsigned int cnt = 0;
void *count (void *data);

int main (void)
{
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, count, NULL);
    pthread_create(&tid2, NULL, count, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    if (cnt != (unsigned) NITERS*2)
        printf("BOOM! cnt=%d\n", cnt);
    else
        printf("OK! cnt=%d\n", cnt);
    return 0;
}

void *count (void *data)
{
    int i;
    for (i=0; i<NITERS; i++)
        cnt++;
    return NULL;
}
```

```
juyoon@linda: ~/test/system
juyoon@linda:~/test/system$ pths
BOOM! cnt=155810007
juyoon@linda:~/test/system$ pths
BOOM! cnt=154370499
juyoon@linda:~/test/system$ pths
BOOM! cnt=154471233
juyoon@linda:~/test/system$ pths
BOOM! cnt=156678495
juyoon@linda:~/test/system$ pths
BOOM! cnt=154721524
juyoon@linda:~/test/system$
```

공유 변수

- 왜 정확한 값이 나오지 않을까?
  - C 프로그램 상에서는 문제가 없다.
  - CPU가 일하는 수준으로 내려가면...
    - 하나의 C 문장이 여러 단계의 기계어로 나누어진다.
    - CPU는 한 번에 하나의 쓰레드만 실행
      - 일정 시간 수행 후 교체
    - 명령 실행의 중간 단계에 실행 쓰레드가 교체되면 값이 보존되지 않을 수도 있다! - race condition

- 공유 변수에 대한 보호막 필요!

  → Mutual Exclusion (상호 배제)

상호 배제

- 상호 배제 관련 함수

함수	의미
pthread_mutex_init	mutex 객체 초기화
pthread_mutex_destroy	mutex 객체 제거
pthread_mutexattr_init	mutex 속성 객체 초기화
pthread_mutexattr_destroy	mutex 속성 객체 제거
pthread_mutex_lock	mutex 영역에 들어가기 위한 lock 요청 (block 상태에서 대기)
pthread_mutex_trylock	mutex 영역에 들어가기 위한 lock 요청 (대기하지 않음)
pthread_mutex_unlock	mutex 영역에서 빠져나올 때 lock 돌려 주기

상호 배제

❑ mutex 객체 초기화

- mutex를 이용해 임계 영역(critical region)을 구현하려면 먼저 mutex 객체를 선언하고 초기화해야 한다.

```
#include <pthread.h>

int pthread_mutex_init (pthread_mutex_t *mutex,
                        const pthread_mutexattr_t *attr);
```

mutex	mutex 객체
attr	mutex 속성 객체. default로 설정하려면 NULL
반환값	호출이 성공하면 0, 실패하면 0이 아닌 에러 코드 반환

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

==

```
pthread_mutex_t mutex;
pthread_mutex_init (&mutex, NULL);
```


상호 배제

❑ mutex 객체 삭제

```
#include <pthread.h>

int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

mutex	mutex 객체
반환값	호출이 성공하면 0, 실패하면 0이 아닌 에러 코드 반환

- lock 상태의 mutex 객체는 삭제해서는 안 된다.

상호 배제

❑ 임계 영역 출입

```
#include <pthread.h>

int pthread_mutex_lock (pthread_mutex_t *mutex);
int pthread_mutex_trylock (pthread_mutex_t *mutex);
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

mutex	mutex 객체
반환값	호출이 성공하면 0, 실패하면 0이 아닌 에러 코드 반환

- lock: 임계영역에 들어갈 권리(lock)를 요청한다. 누군가 lock해 놓은 상태에서는 수행을 중단하고 기다린다.
- unlock: 임계영역에서 작업을 끝내고 lock을 해제한다. 대기 중인 다른 쓰레드가 lock을 얻어 수행할 수 있다.
- trylock: lock을 요청한다. lock을 획득할 수 없으면 대기하지 않고 돌아온다.

상호 배제

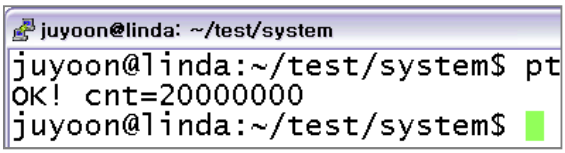
❑ mutex 예제

- 정확한 값 출력
- 시간이 더 걸린다.

```
pthread_mutex_t mutex;
…

int main (void)
{
    …
    pthread_mutex_init (&mutex, NULL);
    …
    pthread_mutex_destroy (&mutex);
    …
}

void *count (void *data)
{
    for ( i=0; i<NITERS; i++) {
        pthread_mutex_lock(&mutex);
        cnt++;
        pthread_mutex_unlock(&mutex);
    }
}
```



Thread!

- 효율적인 프로그램을 작성할 수 있다!
  - 서버나 GUI에서 일상적으로 사용된다.
- 주의 사항
  - 쓰레드 종료 시 자동 청소가 어렵다 ← 부모가 책임지도록 프로그램해야 한다.
  - 공유 변수에 동시에 변경을 가할 경우 오류가 생기지 않도록 상호 배제를 구현해야 한다.