14. Thread

Hyunchan, Park

http://oslab.jbnu.ac.kr

Division of Computer Science and Engineering

Jeonbuk National University

학습내용

- Parallel Programming
- Thread

Parallel Programming

병렬 프로그래밍

- 공동의 목적을 달성하기 위해, 다수의 실행 주체가 동시에 작업을 수행하는 방식
  - 실행 주체: Process or Thread
  - 사실 컴퓨터 시스템에서 프로그램을 실행하는 주체는 CPU (or Core)!!
  - 프로세스나 쓰레드는 여러 프로그램이 CPU를 공유해서 사용하기 위한 추상적 단위
- 병렬 프로그래밍의 필요성
  - 최근 컴퓨터 시스템은 대부분 다수의 CPU 를 탑재하고 있으며,
  - 여러 CPU를 최대한 활용하여 성능을 높이기 위해서는 병렬 프로그래밍이 필수

MODEL	CORES/ THREADS
AMD Ryzen™ 9 5950X	16C/32T
AMD Ryzen™ 9 5900X	12C/24T
AMD Ryzen™ 7 5800X	8C/16T
AMD Ryzen™ 5 5600X	6C/12T

3rd Generation Intel Xe				
Model	Family	L3 Cache	Cores	Threads
8380HL	Platinum	38.50	28	56
8380H	Platinum	38.50	28	56
8376HL	Platinum	38.50	28	56
8376H	Platinum	38.50	28	56
8354H	Platinum	24.75	18	36
8353H	Platinum	24.75	18	36
6348H	Gold	33.00	24	48
6328HL	Gold	22.00	16	32
6328H	Gold	22.00	16	32
5320H	Gold	27.50	20	40
5318H	Gold	24.75	18	36

소켓 프로그래밍에서는?

- 일반적으로 서버는 여러 클라이언트에 대해 동시에 서비스를 제공함
  - 예) 포털에 여러 사람이 동시에 접근하여 각자 서로 다른 서비스를 이용함
- 반복 서버
  - 하나의 프로세스가 동작하며 모든 클라이언트의 요청을 순서대로 처리
  - 클라이언트는 연결 수립을 위해
    이전 클라이언트의 요청이 모두 처리될 때까지 대기하여야 함
- 동시 동작 서버 (Parallel or Concurrently running or Multi-user Server)
  - 여러 요청을 동시에 처리해 서비스할 수 있는 서버
  - 서버를 병렬로 동작하도록 구성
  - 일반적인 서버 역할 분담
    - Door-keeper process: 문지기 역할. 접속을 대기하다, 요청이 오면 연결을 수립함. 그리고 수립된 각각의 연결에 대해 서비스 프로세스를 할당함. 이를 반복함.
    - Service thread: 각 연결에 대해 1:1로 실제 서비스를 제공하는 쓰레드

반복 서버의 예

- 지난 소켓에서의 서버-클라이언트 예제 동작 방식
  - Server: listen()→ accept() → send() → recv() → close()
  - Client: connect() → recv() → send() → close()
- 서버 프로세스 수정: 반복 처리 서버
  - 하나의 클라이언트 처리가 종료된 후, 다시 accept()로 돌아와서 대기
  - 위 동작을 3번 반복하여, 3개의 클라이언트 요청을 처리하게 함
- 만약 클라이언트가 send()를 하지 않는다면?
  - 서버 프로세스는 recv() 에서 무한 대기하며,
    해당 클라이언트로부터 패킷이 전송되길 기다릴 것
- 그때 만약 다른 클라이언트가 동시에 접속을 요청한다면?
  - 서버가 accept()를 대기하고 있기 않기 때문에, 접속 처리를 할 수 없음

[예제 1] 반복 서버: Server (1/2)

C n1.c × C n2.c

hw12 > C n1.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define BUFFSIZE 4096
#define SERVERPORT 7799

int main(void) {
    int i, s_sock, c_sock;
    struct sockaddr_in server_addr, client_addr;
    socklen_t c_addr_size;
    char buf[BUFFSIZE] = {0};
    char hello[] = "Hello~I am Server!\n";

    s_sock = socket(AF_INET, SOCK_STREAM, 0);

    int option = 1;          // SO_REUSEADDR 의 옵션 값을 TRUE 로
    setsockopt(s_sock, SOL_SOCKET, SO_REUSEADDR, &option, sizeof(option));

    bzero(&server_addr, sizeof(server_addr));

    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(SERVERPORT);
    server_addr.sin_addr.s_addr = inet_addr("10.0.0.249");

    if (bind(s_sock, (struct sockaddr *) &server_addr, sizeof(server_addr)) == -1) {
        perror("[S] Can't bind a socket");
        exit(1);
    }

    listen(s_sock,1);
    c_addr_size = sizeof(struct sockaddr);

```

```
ubuntu@41983:~/hw12$ ./n1
[S] Can't bind a socket: Address already in use
```

```
ubuntu@41983:~/hw12$ netstat -al | grep 7799
tcp        0      0 41983:7799              41983:38752             TIME_WAIT
tcp        0      0 41983:7799              41983:38756             TIME_WAIT
tcp        0      0 41983:38754             41983:7799              TIME_WAIT
```

- 프로그램 종료 후, 다시 실행시켰을 때, bind() 에서 “already in use” 오류가 나는 것을 무시하기 위해 사용 (참고)

[예제 1] 반복 서버: Server (2/2)

```
35      for(i=0; i<3; i++) {
36          printf("[S] waiting for a client..#%02d\n", i);
37          c_sock = accept(s_sock, (struct sockaddr *) &client_addr, &c_addr_size);
38          if (c_sock == -1) {
39              perror("[S] Can't accept a connection");
40              exit(1);
41          }
42
43          printf("[S] Connected: client IP addr=%s port=%d\n", inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));
44
45          //1. say hello to client
46          if (send(c_sock, hello, sizeof(hello)+1, 0) == -1) {
47              perror("[S] Can't send message");
48              exit(1);
49          }
50
51          printf("[S] I said Hello to Client!\n");
52
53          //2. recv msg from client
54          if (recv(c_sock, buf, BUFFSIZE, 0) == -1) {
55              perror("[S] Can't receive message");
56              exit(1);
57          }
58
59          printf("[S] Client says: %s\n", buf);
60
61          close(c_sock);
62      }
63      close(s_sock);
64
65      return 0;
66  }
```

[예제 1] 반복 서버: Client (1/2)

hw12 > C n2.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define BUFFSIZE 4096
#define SERVERPORT 7799

int main(void) {
    int c_sock;
    struct sockaddr_in server_addr, client_addr;
    socklen_t c_addr_size;
    char buf[BUFFSIZE] = {0};
    char hello[] = "Hi~I am Client!!\n";

    c_sock = socket(AF_INET, SOCK_STREAM, 0);

    bzero(&server_addr, sizeof(server_addr));

    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(SERVERPORT);
    server_addr.sin_addr.s_addr = inet_addr("10.0.0.249");


    printf("[C] Connecting...\n");

    if (connect(c_sock, (struct sockaddr *) &server_addr, sizeof(server_addr)) == -1) {
        perror("[C] Can't connect to a Server");
        exit(1);
    }

    printf("[C] Connected!\n");
```

[예제 1] 반복 서버: Client (2/2)

hw12 > C n2.c

```c
    printf("[C] Connecting...\n");

    if (connect(c_sock, (struct sockaddr *) &server_addr, sizeof(server_addr)) == -1) {
        perror("[C] Can't connect to a Server");
        exit(1);
    }

    printf("[C] Connected!\n");

    //1. recv msg from server (maybe it's "hello")
    if (recv(c_sock, buf, BUFFSIZE, 0) == -1) {
        perror("[C] Can't receive message");
        exit(1);
    }

    printf("[C] Server says: %s\n", buf);

    //2. say hi to server

    if (send(c_sock, hello, sizeof(hello)+1, 0) == -1) {
        perror("[C] Can't send message");
        exit(1);
    }

    printf("[C] I said Hi to Server!!\n");

    //printf("[C] I am going to sleep...\n");
    //sleep(10);

    close(c_sock);

    return 0;
}
```

[예제 1] 정상 실행 예

```
ubuntu@41983:~/hw12$ ./n1
[S] waiting for a client..#00
[S] Connected: client IP addr=10.0.0.249 port=38970
[S] I said Hello to Client!
[S] Client says: Hi~I am Client!!

[S] waiting for a client..#01
[S] Connected: client IP addr=10.0.0.249 port=38972
[S] I said Hello to Client!
[S] Client says: Hi~I am Client!!

[S] waiting for a client..#02
[S] Connected: client IP addr=10.0.0.249 port=38974
[S] I said Hello to Client!
[S] Client says: Hi~I am Client!!

ubuntu@41983:~/hw12$
```

```
ubuntu@41983:~/hw12$ ./n2 & ./n2 & ./n2
[1] 3010698
[2] 3010699
[C] Connecting...
[C] Connected!
[C] Connecting...
[C] Connected!
[C] Connecting...
[C] Server says: Hello~I am Server!

[C] I said Hi to Server!!
[C] Connected!
[C] Server says: Hello~I am Server!

[C] I said Hi to Server!!
[C] Server says: Hello~I am Server!

[C] I said Hi to Server!!
[1]-  Done                    ./n2
[2]+  Done                    ./n2
ubuntu@41983:~/hw12$
```

[예제 1] 비정상 실행 예

- 클라이언트 소스 수정
  - Send() 하지 않고, 10초간 sleep()
  - 맨 첫 클라이언트만 수정한 코드로 실행
- 결과
  - 다른 클라이언트들은 연결 수립은 됨
    - 포트가 열려있으므로, 커널이 수락한 것
  - 그러나 recv() 에서 계속 대기하여야 함
    - 서버 프로세스가 recv()에서 대기하느라, send()를 할 수 없으므로
  - 10초 후, 첫 번째 클라이언트가 소켓을 닫음으로써 연결이 종료되고,
    이로 인해 recv() 가 취소되어 다음 동작을 수행 가능함

```
45      //2. say hi to server
46      /*
47      if (send(c_sock, hello, sizeof(hello)+1, 0) == -1) {
48          perror("[C] Can't send message");
49          exit(1);
50      }
51
52      printf("[C] I said Hi to Server!!\n");
53      */
54
55      printf("[C] I am going to sleep...\n");
56      sleep(10);
57
58      close(c_sock);
```

```
ubuntu@41983:~/hw12$ ./n1
[S] waiting for a client..#00
[S] Connected: client IP addr=10.0.0.249 port=39018
[S] I said Hello to Client!
```

```
ubuntu@41983:~/hw12$ ./n2_s & ./n2 & ./n2
[1] 3013779
[2] 3013780
[C] Connecting...
[C] Connected!
[C] Server says: Hello~I am Server!

[C] I am going to sleep...
[C] Connecting...
[C] Connected!
[C] Connecting...
[C] Connected!
```

Process-based Parallel Socket Programming

- 프로세스 기반 동시 동작 서버
  - Door-keeper process
    - 기존 서버 코드와 같이 서버 소켓을 열고 bind(), listen() 수행
    - 새로운 연결이 수립되면, child process를 만들어, 아래 service 동작을 수행하게 함
    - 서버는 fork() 이후, 즉각 다시 accept()에서 대기. 이를 3회 반복함
    - 종료 전, 생성된 child 개수만큼 wait()를 수행하여 모든 종료 상태값을 출력 후, 종료
  - Service process
    - 기존과 같이 send(), recv() 수행하고, 소켓을 닫고, 종료
- 개인과제 12-1
  - 뒤의 12-2와 함께 제출
  - 파일 명: hw12/p-server.c, hw12/p-client.c
  - 앞의 비정상 실행 예제와 같이, 중간에 sleep()하는 클라이언트와 일반 클라이언트를 혼합하여 실행한 후, 결과를 캡처하여 보고서에 기재

[예제 2] 동시 동작 서버 수행 예제

```
ubuntu@41983:~/hw12$ ./server
[S] waiting for a client..#00
[S] Connected: client IP addr=10.0.0.249 port=42254
[S] waiting for a client..#01
[SS] Service: I am your child! pid=3052751
[SS] I said Hello to Client!
[S] Connected: client IP addr=10.0.0.249 port=42256
[S] waiting for a client..#02
[SS] Service: I am your child! pid=3052752
[SS] I said Hello to Client!
[SS] Client says: Hi~I am Client!!

[S] Connected: client IP addr=10.0.0.249 port=42258
[S] Child #00 is finished with 0
[SS] Service: I am your child! pid=3052753
[SS] I said Hello to Client!
[SS] Client says: Hi~I am Client!!

[S] Child #01 is finished with 0
[SS] Client says:
[S] Child #02 is finished with 0
ubuntu@41983:~/hw12$
```

```
ubuntu@41983:~/hw12$ ./n2_s & ./n2 & ./n2
[2] 3052748
[3] 3052749
[C] Connecting...
[C] Connected!
[C] Server says: Hello~I am Server!

[C] I am going to sleep...
[C] Connecting...
[C] Connected!
[C] Server says: Hello~I am Server!

[C] I said Hi to Server!!
[1]   Done                    ./n2_s
[C] Connecting...
[C] Connected!
[C] Server says: Hello~I am Server!

[C] I said Hi to Server!!
[3]+  Done                    ./n2
ubuntu@41983:~/hw12$
```

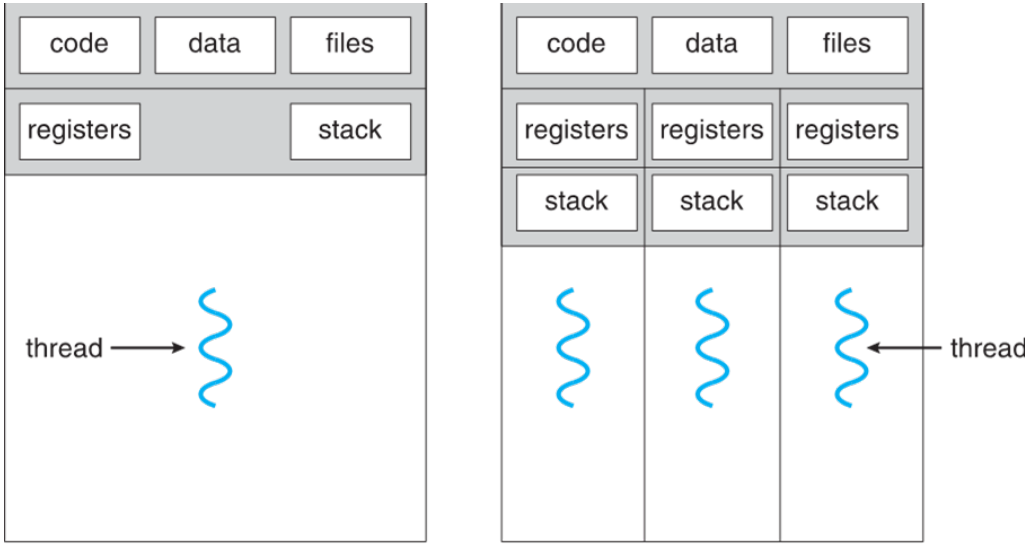
Thread

Thread

- 두 가지 종류의 실행 주체(execution unit or entity)
  - Process: 독립적인 메모리 공간을 가진 수행 주체
    - 각각 독립적인 메모리 공간을 생성해주고, 관리해주어야 하기 때문에
    - 생성/제거에 시간이 오래 걸리고, 시스템 자원을 많이 사용함
  - Thread: 프로세스의 단점을 보완하기 위해 새로이 정의된 실행 주체
- Thread: "The execution unit in a process"
  - Thread = CPU Register set + Independent Stack
    - 하나의 수행 흐름이 다른 수행 흐름과 분리되기 위해 필수적인 요소들만 모아 Thread로 정의
  - 하나의 프로세스 내에 여러 쓰레드가 함께 동작할 수 있음
    - 쓰레드들은 프로세스의 자원을 공유해서 사용: code, data, heap 영역, File descriptor 등
  - 장점
    - 프로세스보다 가볍다 (a.k.a. LWP: light-weight process)
      - 여러 쓰레드가 동시에 동작할 때, 여러 프로세스의 경우보다 효율적임 (성능 높음)
    - IPC 가 불필요함: 하나의 프로세스 내에서 메모리 공간을 공유하기 때문
  - 단점: 데이터 공유에 따른 동기화 문제, 오류 전파 (error propagation) 등

Thread

- Single-threaded Process vs. Multi-threaded Process
  - STP: 전통적인 프로세스의 구조와 동일함
    - 내부에서 명시적으로 쓰레드 관련 서비스를 사용하지 않더라도,
    - 코드가 수행되는 하나의 흐름을 쓰레드라는 개념으로 구체화하였음
    - 프로세스 생성이 완료되면, 기본 쓰레드가 코드를 순차적으로 수행하기 시작함
  - MTP: 쓰레드 개념이 등장하면서 가능해진 구조
    - 여러 쓰레드가 하나의 프로세스 내에서 프로세스의 자원을 공유하며 동시에 수행됨
  - "프로세스" 는 더 이상 실행 주체 라기보다, 실행 주체인 쓰레드를 담는 그릇과 같은 역할



single-threaded process　　multithreaded process

Pthread 서비스

- Pthread (POSIX Thread)
  - POSIX 는 운영체제 API 표준으로, 여러 OS에서 지원하고 있음
  - pthread 라이브러리: POSIX 에서 쓰레드를 관리하기 위한 여러가지 API를 구현한 것
    - 쓰레드 관련 시스템 콜들을 대신 호출하고, 관련 자료 구조를 관리함
  - 가장 일반적으로 널리 이용되는 쓰레드 라이브러리
  - 생성, 종료, 완료 대기, 동기화 관리 (lock) 등의 기능을 제공
  - gcc 컴파일 시, “-l pthread” 옵션을 사용해 명시적으로 라이브러리 지정을 해주어야 함

	프로세스	쓰레드
생성	fork()	pthread_create()
완료 대기	wait()	pthread_join()
수행 코드 변경	exec()	-

쓰레드 생성: pthread_create()

```
#include <pthread.h>

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

Compile and link with -pthread.
```

- thread: 생성된 쓰레드를 가리키는 구조체. 쓰레드를 가리키기 위한 ID로 사용
  - TID: Thread ID. 숫자가 아니라 구조체로 표현됨
  - pthread_self()를 통해 확인 가능. pthread_equal()을 통해 비교 가능
- attr: struct pthread_attr_t 로 정의된 쓰레드의 속성들. NULL 인 경우, 기본 속성
- start_routine: 쓰레드로 수행할 코드의 함수 포인터
  - void * start_thread (void *arg);
- arg: start_routine 에 전달할 인자
- Return value
  - On success: 0 , On error: error number (not zero)

쓰레드의 종료

- 쓰레드를 종료하는 네 가지 방법
  - 쓰레드 생성 시 지정된 start_routine 에서 return 하는 경우
  - pthread_exit() : 자기 자신을 종료
    - 상태 종료값을 retval 로 전달. retval 은 동적으로 할당하여 사용
  - pthread_cancel() : 인자로 지정된 쓰레드를 종료
  - 전체 프로세스가 종료되는 경우
    - main() 함수의 return
    - (어느 thread 에서든) exit() 호출
    - exec() 시리즈로 새로운 코드가 로드된 경우

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

```
int pthread_cancel(pthread_t thread);
```

쓰레드 종료 대기: pthread_join()

```
#include <pthread.h>

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

- thread 의 종료를 대기하고, retval 에 종료 상태값을 전달함
  - 프로세스에서의 wait() 와 유사한 역할
- Return value
  - On success: 0 , On error: error number (not zero)

- pthread_detach

```
int pthread_detach(pthread_t thread);
```

  - 만약 쓰레드를 join 하지 않기로 결정했다면,
  - detach 를 수행하여 쓰레드 종료 시, 즉각 쓰레드를 destroy 하도록 함
  - 예) 프로세스의 경우, wait()를 수행하지 않으면 좀비 프로세스가 됨
    - 쓰레드는 detach 를 시킨 경우, 다른 쓰레드가 join 을 수행하지 않아도 무방함

[예제 3] 단순 숫자 세기 쓰레드

w12 > C thread.c

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

#define MAX_THREADS (10)

pthread_t tid[MAX_THREADS];

void* counting (void *arg) {
    int i, indent = *((int *)arg), *ret;
    char buf[80];

    srand((unsigned int) indent);
    sleep(rand()%3);

    for(i=0; i<indent; i++) buf[i]='\t';

    for(i=0; i<5; i++) {
        printf("%s%d...\n", buf, i);
        sleep(1);
    }

    printf("%sFINISHED!\n", buf);

    ret = (int *)malloc(sizeof(int));
    *ret = indent;
    pthread_exit(ret);
}
```

```c
int main(int argc, char *argv[]) {
    int i, threads, *status, tcounts = 0, args[MAX_THREADS];

    if(argc != 2) {
        printf("Usage: ./thread <a number of thread>\n");
        return 1;
    }

    threads = atoi(argv[1]);

    if(threads >= 1 && threads > MAX_THREADS) {
        printf("Max. number of thread is %d! Your input: %d\n", MAX_THREADS, threads);
        return 2;
    }

    printf("Your input: %d\n", threads);

    for(i=0; i<threads; i++, tcounts++) {
        args[i] = i;
        if( pthread_create(&tid[i], NULL, counting, &args[i]) != 0 ) {
            perror("Failed to create thread");
            goto exit;
        }
    }

exit:
    for(i=0; i<tcounts; i++) {
        pthread_join(tid[i], (void **) &status);
        printf("Thread no.%d ends: %d\n", i, *status);
    }

    return 0;
}
```

[예제 3] 수행 결과

```
ubuntu@41983:~/hw12$ ./thread 5
Your input: 5
                0...
                        0...
0...
        0...
                1...
                                0...
                        1...
        1...
                2...
1...
                                1...
                        2...
                3...
        2...
2...
                                2...
                        3...
        3...
                4...
3...
                                3...
                        4...
                FINISHED!
                                4...
4...
        4...
                        FINISHED!
                                FINISHED!
FINISHED!
Thread no.0 ends: 0
        FINISHED!
Thread no.1 ends: 1
Thread no.2 ends: 2
Thread no.3 ends: 3
Thread no.4 ends: 4
```

개인 과제 12-2: Multi-threaded File Transfer Service

- 프로그램 2개 작성: server.c and client.c
  - 개인 과제 11과 동일한 내용의 파일 전송 서버-클라이언트 프로그램
  - 쓰레드를 사용하여 여러 전송 요청을 동시에 처리하여야 함
    - 최대 10개
  - 여러 전송 요청이 동시에 처리됨을 확인할 수 있도록 실험을 구성하고, 결과를 기재
  - 추가 점수 (최대 50%)
    - 12-1 의 프로세스를 이용한 예제와 비교하여, 성능, 메모리 사용량 등을 비교
- 주의 사항
  - /home/ubuntu/hw12 에서 위 파일명으로 새로운 파일 생성하여 작업할 것
- 제출 기한
  - 12/16 (수) 23:59 (기한 이후 제출 불가)
  - 12-1, 12-2 과제를 함께 압축하여 LMS 제출
    - 동작 설명과 결과가 포함된 간단한 보고서(PDF!!!) & 소스 파일들