Approach #3 : Thread-based Servers

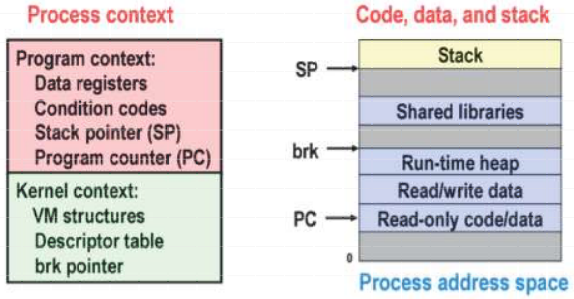
+) Background Idea

→ DRAM은 power가 charging 된 상태여야만, 메모리 셀의 데이터를 유지하는 Volatile memory

→ 프로세스 P1가 생성되면, 메모리에 적재 'code-data-stack-heap' 03 구분.

Traditional View of a Process

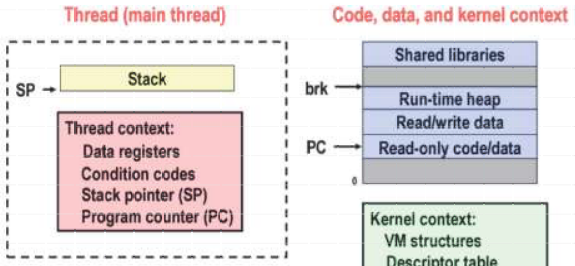
Process = Process context + Code/Data/Stack
(Program context + Kernel context)



Alternate view of a Process

Stack을

Process = Thread + Code/Data/Kernel Context = Thread (Program Context + Stack) + Code/Data + Kernel Context

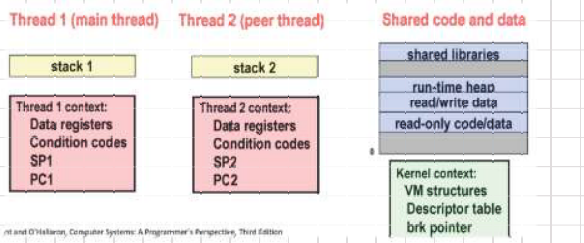


* stack이 다른 thread로부터 보호받지는 X.

stack 공간을 나누는 것.

↓

그래서 나와 같이 local variable 참조하면 안됨.



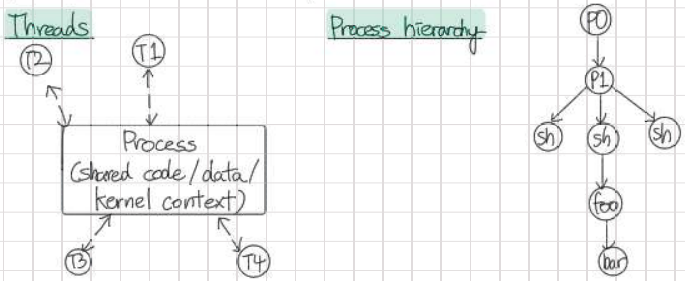
nt and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Logical view of Threads

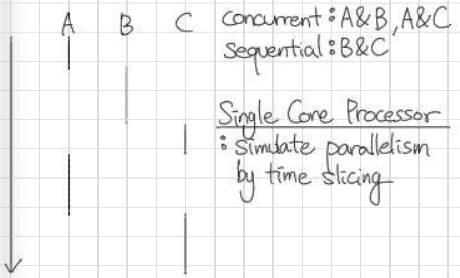
→ Thread는 자신의 process에 여러 동등한 peer들과 연결되어 있음.

↕

process에서는 child들의 hierarchy를 구축했었음.

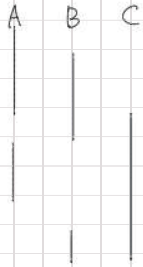


Concurrent Threads



concurrent : A&B, A&C
sequential : B&C

Single Core Processor
: Simulate parallelism by time slicing



Multi-Core Processor
: can have true parallelism

Core가 N개면 최대 N개의 Thread 동시에 수행가능

Threads vs Processes

◦ 어떻게 비슷한가?

→ own logical flow 가지고 있음.

→ can run concurrently with others (possibly on different cores)

→ context switch에 영향 받음.

◦ 어떻게 다른가.

→ Thread는 모두 code, data 영역 공유 (local stack 제외) → +) kernel context

→ Thread는 process보다 less expensive, 2배 더.

process create 하고 reaping 하는데 ~20K cycles

thread create 하고 reaping 하는데 ~10K cycles

05/20

Posix Threads (Pthreads) Interface

creating & reaping threads
- pthread_create() — fork와 비슷
- pthread_join() — waitpid와 비슷

determining your thread ID
- pthread_self() — getpid와 비슷

terminating threads
- pthread_cancel()
- pthread_exit()
- exit()

synchronizing access to shared variables
- pthread_mutex_init
- pthread_mutex_[un]lock

The Pthreads "hello, world" Program

```c
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"
void *thread(void *vargp);

int main()
{
    pthread_t tid;
    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}
```

hello.c

Thread ID

Thread attributes (usually NULL)

Thread routine

Thread arguments (void *p)

Return value (void **p)

```c
void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!\n");
    return NULL;
}
```

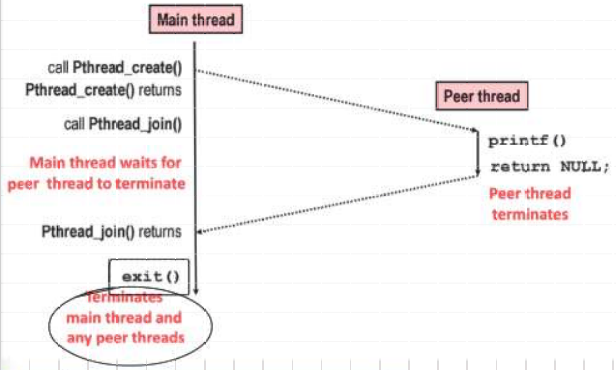
hello.c

→ main thread(process)가 있고, pthread_create 이용해 새로운 peer thread 생성

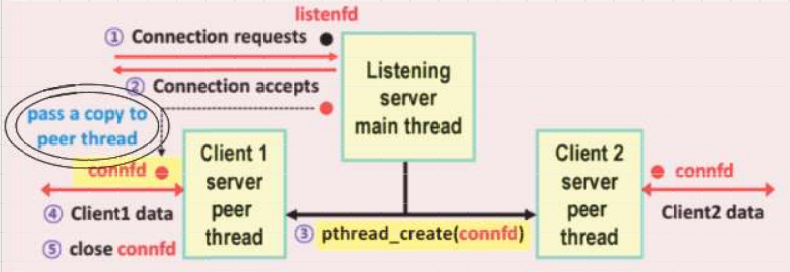
→ pthread_create(TID 변수 주소값, 일반적으로 NULL, Thread Routine, Routine의 인자);

→ pthread_join (TID 변수값, return값)

Execution of Threaded "hello, world"



Thread-based Server Execution Model



→ main thread는 listening 역할

→ peer threads는 connection 및 service 역할

→ 각 client는 개별의 peer thread가 handling

→ main과 peer thread는 TID 제외 all process state 공유

→ 각 thread는 local variable 위한 separate stack 가짐.

+) code / data / kernel context

data
- → 전역변수
- → heap memory
- → static variables

kernel context
- → file descriptor table
- → VM structures
- → brk pointer
- → signal handler
- =

claude 피셜 바뀌어야할 점
1) stack 변수 주소를 넘기지 마라
2) thread 간 공유할 데이터는 힙이나 전역변수에 저장하고 mutex로 보호

잘못된 case (자료 받은 거)

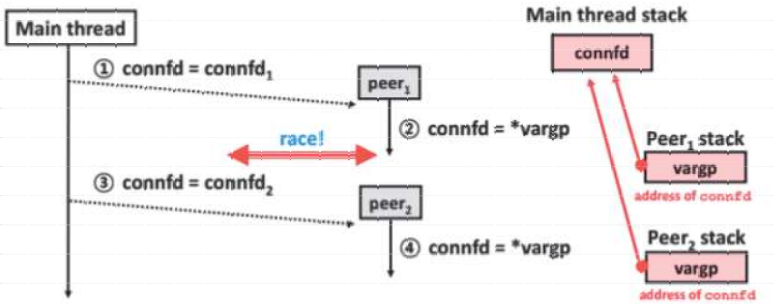
```c
int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;
    pthread_t tid;

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen=sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        Pthread_create(&tid, NULL, thread, &connfd);
    }
}
```

Spawn a new thread for each client

Pass the address of connection file descriptor (connfd) to the client

```c
/* Thread routine */
void *thread(void *vargp)
{
    int connfd = *((int *)vargp);
    echo(connfd);
    Close(connfd);
    return NULL;
}
```

Any problems with this approach?



①→②→③→④가 아닌
①→③→②→④라면?

왜?

ex) client 1 연결 → connfd=5
Main thread가 &connfd를 thread 1에 넘김.
thread 1에서 int connfd = *((int*) vargp); 실행 전에
main thread가 loop 돌아서, client 2 연결하고 connfd 변수에 7 저장.
이제 thread 1이 *((int*)vargp) 실행하면 5가 아닌 7을 읽게 됨. → thread 1이 client 1이 아닌 client 2에 응답.

올바른 case

```c
int main(int argc, char **argv)
{
    int listenfd, *connfdp;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;
    pthread_t tid;

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen=sizeof(struct sockaddr_storage);
        connfdp = Malloc(sizeof(int));
        *connfdp = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        Pthread_create(&tid, NULL, thread, connfdp);
    }
}
```

echoserve

```c
/* Thread routine */
void *thread(void *vargp)
{
    int connfd = *((int *)vargp);
    Pthread_detach(pthread_self());
    Free(vargp);
    echo(connfd);
    Close(connfd);
    return NULL;
}
```

type casting

main에서 allocate
thread에서 free

→ connfdp는 포인터 변수 (main thread의 스택에 있는)
*connfdp는 heap에 존재, thread에 heap 공간을 넘기는 것

→ detached mode
- 다른 thread와 독립적으로 run.
- terminate하면 kernel이 reap automatically

☆ Free(vargp) 해야함.) peer thread.
☆ Close connfd
↓
메인 thread에서 따로 닫을 필요없음.

하나의 프로세스 안에서 thread들은 같은 file descriptor table 공유
∴ listenfd는 main thread에서만 닫고
connfd는 peer thread에서만 닫자.

Issues With Thread-Based Servers

- 반드시 run "detached"해서, memory leak 방지.
  - → thread는 joinable 또는 detached
  - → Joinable thread는
    다른 thread (main이던 peer thread던)가 reap, kill 가능
    memory resources를 free하기 위해서는 pthread_join으로 reap해야함.
  - → Detached thread는
    다른 thread가 reap 또는 kill 불가.
    OS Kernel이 알아서 reaping 해줌.
  - → default state가 joinable이기에, pthread_detach(pthread_self()) 사용해서 detached로 변경
- unintended sharing에 주의.
  OS scheduling, context switch에 의한 Race 발생
  ex) Pthread_create(&tid, NULL, thread, (void*)&connfd);
- thread가 call하는 모든 function은 thread-safe 해야함.

Pros & Cons of Thread-Based Designs

+) thread간의 자료구조 공유 easy
  ex) logging information, file cache
+) Process based에 비해 efficient (성능이 우수)

-) unintentional sharing은 subtle & hard-to-reproduce errors 야기 가능
  data의 쉬운 공유가 가장 큰 strength이면서 weakness
  어떤 data가 공유되었는지 알기 어려움.
  corruption일 경우 알아차리기 어려움.
  race가 매우 비정하고 가끔 발생해서 알아차리기 어려움.

Summary: Approaches to Concurrency

overhead
event based < thread based < process based

① Process-based

→ resource 공유 어려움 ⇒ unintended sharing 피하기 쉬움.
→ client 추가 및 제거 시 high overhead

② Event-based

→ tedious & low-level ⟶ 내가 직접 이벤트 핸들링 & 상태관리 다 해야함. (개 귀찮음)
→ total control over scheduling
→ very low overhead
→ cannot create as fine grained a level of concurrency ⟶ 세밀한 수준의 동시성 어렵다
  ↳ 이벤트루프가 한번 도는 동안 하나의 클라이언트 요청을 완전히 처리 못하고 쪼개서 해야함.
→ does not make use of multicore

③ Thread-based

→ easy to share resources: perhaps too easy
→ medium overhead
→ not much control over scheduling policies
→ difficult to debug
  ↳ event ordering not repeatable ⟶ 쓰레드 실행순서가 실행할때마다 다르게 나와서