

chapter12. POSIX Threads

objectives

1. Learn basic thread concepts
2. Experiment with POSIX thread calls
3. Explore threaded application design → multi threads
4. Use threads in unsynchronized applications
5. Understand thread-safety and error handling

Threads

- file descriptors를 monitoring 하는 방법에 대해 생각해보자.
 1. a separete process : 독립적, concurrent한 process로 fd를 monitor하면 좋음
→ but, 자식 process는 어떤 변수든 공유가 불가하여 IPC 사용해야 함(OS 도움 필요)
 2. select(), poll() : 함수를 이용하여 monitor
→ but, sequential하게 이용하면 fd 여러 개 monitor 불가(하나가 block 되면 check 불가) → 제한적
 3. Nonblocking I/O with polling : I/O(nonblocking) 이용하여 읽은 곳이 있는지 check
→ but, I/O check를 위해 timing에 대한 hard-coding이 필요함 → 어려움
 4. POSIX asynchronous I/O
→ but, handler는 오직 async-signal-safe func만 사용
- ★ ⇒ A separete thread : 다른 것보다 간단한 해결책!
→ concurrent 가능
- 실행 단위 threads : a unit of execution → stack과 CPU states(ex. registers) 포함
 - program counter를 thread마다 가지고 있음
 - program counter : 다음에 실행할 instruction의 주소값
 - why?
 1. asynchronous event 효율적으로 관리
 2. shared-memory multiprocessor에서 parallel performance 가능

→ OS에게 큰 도움이 됨

(but, 무작정 많이 만드는 것이 좋지는 않음, overhead가 더 커질 수 있음)

- two process in UNIX : 오직 OS(IPC) 통해서만 communicate 가능(ex. files, pipes, sockets)

★ two threads in a task : 공유 메모리를 통해서 communicate 가능

- thread들이 simultaneously 하게 실행하는 것처럼 보임
 - 각각의 thread가 자신만의 CPU를 가지고 있는 것처럼 보임(같은 메모리를 모두 공유)

Multitasking

1. single processor

- a. multithreading : time-division multiplexing (as in multitasking)
 - i. processor : 다른 thread들을 switch하면서 진행 → context switching
 - ii. context switching : user가 동시에 실행되는 것처럼 느낄 정도로 빠르게 진행

2. multi processor or multi-core system

- a. threads(tasks) 실제로 동시에 실행 → processor 혹은 core가 thread 각각 실행
↳ 각각 자신의 thread, task 실행 ↳ 대부분 내용은 multi core

• processes vs. threads

- process : 'heaviest' unit of kernel scheduling

1. 독립적 : process들이 사용하는 address는 OS가 독립적으로 메모리 할당

a. memory, file handles, sockets, device handlers

b. address space나 file resources 공유하지 x → 각각 address space 가짐
(상속, memory 따로 사용, 같은 file은 예외)

2. process의 information 양이 훨씬 많음 ↔ thread 간의 switch : 공유 메모리 사용

a. process 간의 context switch → 기존 process의 상태 저장/load 반복의 overhead ∝ Data 양

3. IPC를 통해서만 interact → OS의 도움 필요

⇒ preemptively multitasked

- threads

scheduling에 따라서 CPU에서 쫓겨나고
ready queue 우선순위에 의해 scheduling 될 수 있음.

- threads : 'lightest' unit of kernel scheduling → process마다 적어도 하나의 thread

- multiple threads ↔ process

1. share the state information of a process, memory, and other resources directly

2. context switching : ^{overhead ↓} faster than process

3. do not own resources : 자체적으로 resource 소유 X

(stack, a copy of the registers(PC), thread-local storage는 예외)

4. kernel threads / user threads

kernel이 관리

userspace에서 의해 관리

⇒ preemptively multitasked (if the OS process scheduler가 preemptive)

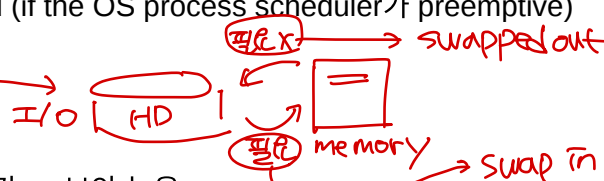
- OS의 virtual memory 분류 방법

1. Kernel space : kernel이 실행되는 공간 → 보안 높음

- a. 절대 swapped out to disk X

2. User space : 모든 user mode의 application이 사용하는 공간

- a. 필요한 경우에 swapped out 가능 ★



POSIX thread library → Pthread

- POSIX thread function : EINTR return X → 중단된 경우 다시 시작할 필요 X
- pthread_t : unsigned long type (in Linux)
- 대부분의 function
 - return 0 → successful
 - return nonzero error → unsuccessful

unsigned long

```
#include <pthread.h>
pthread_t pthread_self(void);
//thread ID return
```

```
pthread_t pthread_equal(pthread_t t1, pthread_t t2);
//t1과 t2가 같은지 check
```

```
//return nonzero value -> equal
//return 0 -> false
```

```
int pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr,
void *(*start_routine)(void *), void *restrict arg);
//자동으로 runnable한 thread 생성 -> 생성 완료되면 실행까지 요청
```

두개의 thread 같은지 판단

```

//thread : 새로 생성된 thread의 ID 반환
//attr : default가 아닌 지정한 속성으로 thread 생성
//start_routine : 함수 pointer -> 새로 만들어진 thread가 수행할 함수 -> 간접적으로 실행
//arg : start_routine 함수의 parameter(input parameter) -> 필요 없으면 NULL

//return 0 -> successful ⇒ 0점을 성공적으로 받아들였다는 뜻, 연연가 실행 할 것이라는 의미!
//return nonzero error -> unsuccessful

int pthread_detach(pthread_t thread);
//'thread'를 detach thread로 만들
//-> thread가 종료될 때 바로 resource 회수할 수 있도록 내부 옵션 설정

//detach thread : 다른 thread가 해당 thread의 종료를 기다릴 수 X -> 상태를 report X
//      사용했던 resource 바로 release
//      <-> joinable thread : thread의 default. 다른 thread가 종료 기다릴 수 있음
//      사용했던 resource 바로 release X -> 종료 후 반환값 있을 수 있음

//return 0 -> successful
//return nonzero error -> unsuccessful thread가 공간생성 → p+에 pointer의 위치를 남겨줌

int pthread_join(pthread_t thread, void **value_ptr);
//'thread'가 종료될 때까지 호출한 thread를 suspend → 기다리는 thread로부터 return 값을 받기 위함.
//joinable thread의 resource :
//      다른 thread가 pthread_join을 호출 or 전체 process가 종료될 때까지 release X

//thread : a target thread
//value_ptr : return status의 pointer 위치
//      -> NULL : 호출한 함수가 상태를 반환 X

//return 0 -> successful
//return nonzero error -> unsuccessful

//ex. pthread_join(pthread_self()); -> 나의 종료를 내가 기다림 -> deadlock

```

```

//example - creation/joining : 여러 thread를 monitoring

void monitorfd(int fd[], int numfds) { /* create threads to monitor fds */
    int error, i;
    pthread_t *tid;
    ① 각 pthread type은 memory 등적할당

    if ((tid = (pthread_t *)calloc(numfds, sizeof(pthread_t))) == NULL) {
        perror("Failed to allocate space for thread IDs");
        return;
    }

    for (i = 0; i < numfds; i++) /* create a thread for each file descriptor */
        if (error = pthread_create(tid + i, NULL, processfd, (fd + i))) {
            fprintf(stderr, "Failed to create thread %d: %s\n", i, strerror(error));
            tid[i] = pthread_self(); ← 피 저장
        }

    for (i = 0; i < numfds; i++) {
        if (pthread_equal(pthread_self(), tid[i]))
            continue;
        if (error = pthread_join(tid[i], NULL))
            fprintf(stderr, "Failed to join thread %d: %s\n", i, strerror(error));
        return 할게 없다면 시니로 설정
    }
}

```

```

}

free(tid);
return;
}

```

```

#include <pthread.h>
void pthread_exit(void* value_ptr);
//causes the calling thread to terminate.
//Q. exit()와 다른 점 -> 'return'과 동일함. return은 implicitly하게 pthread_exit() 호출

//value_ptr : available to successful pthread_join

int pthread_cancel(pthread_t thread);
//'thread'가 동작 취소되도록 요청하는 함수
//취소 요청이 cancel된 option으로 생성된 thread는 cancel 불가

//return 0 -> successful
//return nonzero error -> unsuccessful

//result는 target thread의 state와 type에 따라 결정됨
//1. PTHREAD_CANCEL_ENABLE: receives the request → 요청 받아들임
//2. PTHREAD_CANCEL_DISABLE: the request is held pending → 요청이 pending

int pthread_setcancelstate(int state, int *oldstate);
//호출한 thread의 cancellability 상태를 변경하는 함수

//state : PTHREAD_CANCEL_ENABLE or PTHREAD_CANCEL_DISABLE
//oldstate : 기존 thread cancellability 상태 저장

//return 0 -> successful
//return nonzero error -> unsuccessful

int pthread_setcanceltype(int type, int *oldtype);
//취소할 시점을 설정하는 함수
//이전 시점을 저장하는 output parameter

void pthread_testcancel(void);
//호출한 함수에서 취소할 지점을 설정하는 함수

//cancel -> 만약 thread가 release 되어야만 하는 resource를 가지고 있을 경우
//exit handler에서 resource를 release하는게 항상 가능하지 x → cancel 시점과 type을 설정해야 함

//cancellation type
//1. PTHREAD_CANCEL_ASYNCHRONOUS(default) : 요청을 받은 순간 cancel
//2. PTHREAD_CANCEL_DEFERRED : 내가 지정한 지점에서만 cancel

```

- example → program 12.5

```

//copyfilemalloc.c
//file copy해서 copy된 byte수를 return
↳ 동적할당 메모리를 사용해서 return (↔ static storage에 있을 경우 하나의 thread에서만 사용 가능)

#include <stdlib.h>
#include <unistd.h>
#include "restart.h"

```

```

void *copyfilemalloc(void *arg) { /* copy infd to outfd with return value */
    int *bytesp;
    int infd;
    int outfd;

    infd = *((int *)arg);
    outfd = *((int *)arg) + 1;
    if ((bytesp = (int *)malloc(sizeof(int))) == NULL)
        return NULL;
    *bytesp = copyfile(infd, outfd);
    r_close(infd);
    r_close(outfd);
    return bytesp;
}

```

↙역참조

→ local 변수 사용 X
pointer를 지정한 뒤 동작할당해야
return 되더라도 heap에서 잊어지지 X

```

#include <errno.h>
#include <fcntl.h>
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#define PERMS (S_IRUSR | S_IWUSR)
#define READ_FLAGS O_RDONLY
#define WRITE_FLAGS (O_WRONLY | O_CREAT | O_TRUNC)

void *copyfilemalloc(void *arg);

int main (int argc, char *argv[]) { /* copy fromfile to tofile */
    int *bytesp;
    int error;
    int fds[2];
    pthread_t tid;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s fromfile tofile\n", argv[0]);
        return 1;
    }
    if (((fds[0] = open(argv[1], READ_FLAGS)) == -1) ||
        ((fds[1] = open(argv[2], WRITE_FLAGS, PERMS)) == -1)) {
        perror("Failed to open the files");
        return 1;
    }
    if (error = pthread_create(&tid, NULL, copyfilemalloc, fds)) {
        fprintf(stderr, "Failed to create thread: %s\n", strerror(error));
        return 1;
    }
    if (error = pthread_join(tid, (void **)&bytesp)) {
        fprintf(stderr, "Failed to join thread: %s\n", strerror(error));
        return 1;
    }
    printf("Number of bytes copied: %d\n", *bytesp);
    return 0;
}

```

→ source file

→ target file

① pthread 생성

② suspend : tid가 종료될 때까지 현재 thread가 기다림
→ tid의 return값

→ free!

```
ccslab@ccslab-linux:~/programs/usp_all/chapter12$ callcopymalloc README README2
Number of bytes copied: 538
```

• Program 12.6

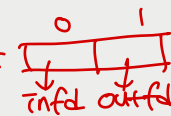
```
//copyfilepass.c
//copyfilemalloc.c의 개선된 버전
//이전 예제 : thread가 single integer를 가지고 있기위해 불필요한 동적할당 공간 사용함
```


```
//alternative approach :
//fd의 idx를 하나 더 생성하여 copy된 byte수를 추가된 idx에 담음
//join 호출해서 return 할 필요 x → 해당 fd에서 byte수 가져옴!
```

```
#include <unistd.h>
#include "restart.h"
```

```
void *copyfilepass(void *arg) {
    int *argint;

    argint = (int *)arg;
    argint[2] = copyfile(argint[0], argint[1]);
    r_close(argint[0]);
    r_close(argint[1]);
    return argint + 2;
}
```

argint 

→ element 증가 ⇒ 

```
#include <errno.h>
#include <fcntl.h>
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#define PERMS (S_IRUSR | S_IWUSR)
#define READ_FLAGS O_RDONLY
#define WRITE_FLAGS (O_WRONLY | O_CREAT | O_TRUNC)
void *copyfilepass(void *arg);

int main (int argc, char *argv[]) {
    int *bytesptr;
    int error;
    int targs[3]; → argument 개수
    pthread_t tid;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s fromfile tofile\n", argv[0]);
        return 1;
    }

    if (((targs[0] = open(argv[1], READ_FLAGS)) == -1) ||
        ((targs[1] = open(argv[2], WRITE_FLAGS, PERMS)) == -1)) {
        perror("Failed to open the files");
        return 1;
    }

    ① thread 생성
    if (error = pthread_create(&tid, NULL, copyfilepass, targs)) {
        fprintf(stderr, "Failed to create thread: %s\n", strerror(error));
        return 1;
    }
}
```

parameter

② +12 종료될 때까지 기다림

```
if (error = pthread_join(tid, (void **)&bytesptr)) {
    fprintf(stderr, "Failed to join thread: %s\n", strerror(error));
    return 1;
}
printf("Number of bytes copied: %d\n", *bytesptr);
printf("Number of bytes copied targs[2]: %d\n", targs[2]);
return 0;
}
```

```
ccslab@ccslab-linux:~/programs/usp_all/chapter12$ callcopypass README README3
Number of bytes copied: 538
Number of bytes copied targs[2]: 538
```

- program 12.9 : wrong parameter passing → multiple threads를 만들 때 thread가 parameter 접근이 완전히 끝났는지 확실할 때까지 thread의 parameter를 reuse 하면 X

```
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#define NUMTHREADS 10

static void *printarg(void *arg) {
    fprintf(stderr, "Thread received %d\n", *(int *)arg);
    return NULL;
}
```

```
int main (void) { /* program incorrectly passes parameters to
threads */
    int error;
    int i;
    int j;
    pthread_t tid[NUMTHREADS];
```

```
    for (i = 0; i < NUMTHREADS; i++)
        if (error = pthread_create(&tid[i], NULL, printarg, (void *)&i)) {
            fprintf(stderr, "Failed to create thread: %s\n", strerror(error));
            tid[i] = pthread_self();
        }
    }
    for (j = 0; j < NUMTHREADS; j++) {
```

```
        if (pthread_equal(pthread_self(), tid[j]))
            continue;
```

```
        if (error = pthread_join(tid[j], NULL))
            fprintf(stderr, "Failed to join thread: %s\n", strerror(error));
    }
```

```
    printf("All threads done\n");
    return 0;
}
```

pthread_create ⇒ delay 발생
 현재 thread create 완료되지 않았지만 다른 thread는 계속 진행 → C++

```
ccslab@ccslab-linux:~/programs/usp_all/chapter12$ badparameters
Thread received 2
Thread received 3
Thread received 3
Thread received 4
Thread received 7
Thread received 8
Thread received 7
Thread received 9
Thread received 8
Thread received 10
All threads done
```

→ 한 번에
 ⇒ 실행할 때마다
 갱신이 다름.

```
ccslab@ccslab-linux:~/programs/usp_all/chapter12$ badparameters
Thread received 0
Thread received 1
Thread received 2
Thread received 3
Thread received 4
Thread received 5
Thread received 6
Thread received 7
Thread received 8
Thread received 9
All threads done
ccslab@ccslab-linux:~/programs/usp_all/chapter12$
```

+ 다른 solution : 함수의 r를 직접 결과에서 넘겨줌

ex) C++ ⇒ C

r=1, r=2 ... (0) → 하나씩 직접 선언

Thread safety

- a thread-safe function : 충돌문제 발생을 방지
 - 여러 thread가 동시에 호출해도 문제 X
 - ↔ async-safe function : 충돌 문제 safe → 다중 thread가 아니더라도 충돌 문제 발생 가능

+) async-safe, thread safe : 둘 다 safe인 함수 가능
↓
만들기 더 쉬움.