6. 쓰레드와 동기화

6.1 `pthread_create()`, `pthread_join()`

개요

리눅스에서 C 언어로 **멀티스레드 프로그래밍**을 하기 위해서는 **POSIX 스레드(POSIX thread, pthread)** 라이브러리를 사용한다.

`pthread_create()` 함수는 새 스레드를 생성하고, `pthread_join()` 함수는 특정 스레드가 종료될 때까지 호출한 스레드(일반적으로 메인 스레드)가 기다리게 한다.

이 두 함수는 멀티스레드 프로그램의 **기본 구성 요소**이다.

스레드와 프로세스

- 프로세스(Process)는 독립적인 실행 단위로 **독립된 메모리 공간**을 가진다.
- 스레드(Thread)는 프로세스 내에서 실행되는 흐름으로, **프로세스의 자원을 공유**한다. (코드 영역, 데이터 영역, 힙, 열린 파일 디스크립터 등)

스레드는 커널에서 스케줄링되므로 진정한 동시 실행(멀티코어 CPU에서 병렬 실행 가능)을 지원한다.

---

pthread_create()

함수 원형

```
#include <pthread.h>

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

매개변수 설명

매개변수	설명
`thread`	생성된 스레드의 ID를 반환할 변수 (pthread_t 타입)
`attr`	스레드 속성 (기본값 사용 시 `NULL`)
`start_routine`	새 스레드에서 실행할 함수의 포인터
`arg`	`start_routine`에 전달할 인자 (`void *` 타입)

반환값

- `0` → 성공
- `에러 코드` → 실패 시 에러 코드 반환

`start_routine` 함수 형식

```
void *start_routine(void *arg);
```

- 스레드 함수는 반드시 `void *` 반환형, `void *` 인자 형식을 가져야 한다.
- 함수 종료 시 `pthread_exit()` 호출하거나 return으로 반환값을 줄 수 있다.

---

pthread_join()

함수 원형

```
#include <pthread.h>

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

매개변수 설명

매개변수	설명
`thread`	기다릴 스레드의 pthread_t 값
`retval`	스레드 종료 시 반환한 값 포인터 (필요 없으면 `NULL`)

반환값

- `0` → 성공
- `에러 코드` → 실패 시 에러 코드 반환

설명

- `pthread_join()` 호출 시 해당 스레드가 종료될 때까지 **현재 스레드가 블록(block)** 된다.
- 스레드 종료 시 반환한 값을 `retval`로 받을 수 있다.

기본 예제

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

void *thread_func(void *arg) {
    int num = *(int *)arg;
    printf("Thread received arg: %d\n", num);
    int result = num * 2;
```

```
    return (void *)(long)result;  // 반환값을 void*로 변환
}

int main() {
    pthread_t tid;
    int arg = 10;
    void *retval;

    // 스레드 생성
    if (pthread_create(&tid, NULL, thread_func, &arg) != 0) {
        perror("pthread_create");
        return 1;
    }

    // 스레드 종료까지 대기
    if (pthread_join(tid, &retval) != 0) {
        perror("pthread_join");
        return 1;
    }

    printf("Thread returned: %ld\n", (long)retval);
    return 0;
}
```

실행 결과 예시

```
Thread received arg: 10
Thread returned: 20
```

주의사항

- `pthread_join()` 을 호출하지 않으면 해당 스레드는 **detached 상태로 전환되지 않으면 자원 누수** 발생 가능.
- 일반적으로 **생성한 모든 joinable thread는 반드시 join하거나 detach** 해야 함.
- `pthread_create()` 로 생성한 스레드는 **프로세스와 동일한 주소 공간**에서 실행됨 → 전역 변수, 힙 메모리 공유 주의 필요.
- `pthread_join()` 을 **반드시 한번만 호출**해야 함. 동일한 스레드에 대해 여러 번 `pthread_join()` 호출 시 undefined behavior 발생.

에러 처리

함수	주요 에러 코드	의미
pthread_create	EAGAIN	시스템 자원 부족 (스레드 생성 실패)
pthread_create	EINVAL	잘못된 스레드 속성
pthread_create	EPERM	권한 부족

함수	주요 에러 코드	의미
pthread_join	ESRCH	해당 스레드 존재하지 않음
pthread_join	EINVAL	스레드가 joinable 상태가 아님
pthread_join	EDEADLK	교착 상태 발생

정리

기능	사용 함수
스레드 생성	`pthread_create()`
스레드 종료 대기	`pthread_join()`

실습 아이디어

- **멀티스레드 웹 크롤러**: URL 리스트를 여러 스레드로 나누어 병렬 다운로드
- **병렬 합계 계산기**: 배열을 여러 구간으로 나누어 스레드별 합산 후 통합
- **멀티스레드 서버**: 클라이언트마다 스레드 생성하여 요청 처리

6.2 뮤텍스(`pthread_mutex_t`)

개요

뮤텍스(Mutex, Mutual Exclusion) 는 **상호 배제**를 의미한다.
멀티스레드 환경에서는 **여러 스레드가 동시에 공유 자원**(전역 변수, 파일, 메모리 등)에 접근하면 **경쟁 상태(Race Condition)** 가 발생할 수 있다.

이를 막기 위해 **임계 영역(Critical Section)** 을 정의하고, 한 시점에 **오직 한 스레드만** 해당 영역을 실행하도록 강제하는 것이 뮤텍스의 목적이다.

---

pthread_mutex_t

POSIX 스레드 라이브러리에서는 **pthread_mutex_t 타입**을 사용해 뮤텍스를 제공한다.

뮤텍스는 다음과 같은 상태 전이를 가진다:

```
Unlocked → Locked → Unlocked → ...
```

한 스레드가 **Lock** 하면 다른 스레드는 **Block(대기)** 상태가 되며, Lock 해제 후에야 접근 가능하다.

주요 함수

pthread_mutex_init()

```
#include <pthread.h>

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

- 뮤텍스 초기화
- `attr`는 일반적으로 `NULL` → 기본 속성 사용
- 반환값: 0(성공), 에러코드(실패)

pthread_mutex_destroy()

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- 뮤텍스 제거
- 뮤텍스가 **사용 중**이면 undefined behavior 발생

pthread_mutex_lock()

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

- 뮤텍스를 **Lock** → 이미 다른 스레드가 Lock 중이면 **Block** 상태로 대기
- 반환값: 0(성공), 에러코드(실패)

pthread_mutex_unlock()

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

- 뮤텍스를 **Unlock** → 다른 대기 중인 스레드에게 Lock 권한 부여
- 반환값: 0(성공), 에러코드(실패)

pthread_mutex_trylock()

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

- 즉시 Lock 시도 → 성공 시 0 반환
- 이미 Lock 중이면 **즉시 EBUSY 반환**(Block하지 않음)

기본 예제

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

#define NUM_THREADS 5

pthread_mutex_t lock;
int counter = 0;

void *thread_func(void *arg) {
    int i;
    for (i = 0; i < 10000; i++) {
        pthread_mutex_lock(&lock);

        // 임계 영역 시작
        counter++;
        // 임계 영역 끝

        pthread_mutex_unlock(&lock);
    }
    return NULL;
}

int main() {
    pthread_t threads[NUM_THREADS];
    int i;

    pthread_mutex_init(&lock, NULL);

    for (i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, thread_func, NULL);
    }

    for (i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    pthread_mutex_destroy(&lock);

    printf("Final counter value: %d\n", counter);
    return 0;
}
```

실행 결과 예시

```
Final counter value: 50000
```

설명

- 5개의 스레드가 `counter`를 1씩 증가 (10000번씩 → 총 50000)
- **뮤텍스 없이 실행하면 counter 값이 틀어짐** → Race Condition 발생
- **뮤텍스 사용 시 정확한 결과** 보장

뮤텍스의 주요 특징

특징	설명
상호 배제	한 번에 한 스레드만 임계 영역 접근 허용
재진입 불가	기본 뮤텍스는 동일 스레드라도 중복 Lock 시 Deadlock 발생 가능
공정성 보장 아님	POSIX 기본 뮤텍스는 스레드가 Lock 요청한 순서를 보장하지 않음
경량화	커널 컨텍스트 스위칭 없이 사용자 공간에서 처리 가능 (경량화된 경우)

사용 패턴

올바른 패턴

```
pthread_mutex_lock(&lock);
... 임계 영역 ...
pthread_mutex_unlock(&lock);
```

잘못된 패턴

- Unlock 없이 return / exit 하면 Lock 상태 유지 → Deadlock 발생
- 반드시 **try-finally 패턴** 또는 **goto-cleanup 패턴**으로 관리하는 것이 좋다.

뮤텍스 속성 (고급)

pthread_mutexattr_t 사용

- `pthread_mutexattr_settype()` 로 뮤텍스의 **타입** 설정 가능
- 주요 타입

타입	의미
PTHREAD_MUTEX_NORMAL	기본 (재귀적 Lock 불가, Deadlock 발생 가능)
PTHREAD_MUTEX_ERRORCHECK	오류 체크 가능 (중복 Lock 시 에러 반환)
PTHREAD_MUTEX_RECURSIVE	재귀적 Lock 허용 (동일 스레드가 여러 번 Lock 가능)
PTHREAD_MUTEX_DEFAULT	구현 정의 (보통 NORMAL과 동일)

예제

```
pthread_mutexattr_t attr;
pthread_mutexattr_init(&attr);
pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);

pthread_mutex_init(&lock, &attr);
```

뮤텍스와 Deadlock

Deadlock 발생 조건 (Coffman 조건)

1. 상호 배제
2. 점유와 대기
3. 비선점
4. 순환 대기

Deadlock 예방 전략

- **Lock 순서 고정**
  항상 동일한 순서로 Lock 획득
- **trylock 사용 후 Backoff**
- **Timeout 기반 Lock 시도**

정리

동작	함수
초기화	`pthread_mutex_init()`
제거	`pthread_mutex_destroy()`
Lock	`pthread_mutex_lock()`
Unlock	`pthread_mutex_unlock()`
비차단 Lock 시도	`pthread_mutex_trylock()`

실습 아이디어

- **은행 계좌 시뮬레이터**: 여러 스레드가 입금/출금 작업 → 뮤텍스 없이 실행 시 오류 발생
- **멀티스레드 로그 파일 기록기**: 뮤텍스 없이 로그 파일 기록 시 꼬임 발생 → 뮤텍스로 보호
- **생산자-소비자 문제 (다음 6.3과 연계)**: 큐에 대한 Push/Pop 시 뮤텍스로 보호

6.3 조건 변수, `pthread_cond_t`

개요

멀티스레드 환경에서는 종종 **특정 조건이 만족될 때까지 기다리는** 동기화 패턴이 필요하다.

- 뮤텍스는 **상호 배제**만 보장한다.
- 조건 변수(Condition Variable)는 **상태 변화 조건**을 기다렸다가 조건이 만족되면 다른 스레드를 깨우는 용도로 사용된다.

조건 변수 + 뮤텍스는 **고전적 동기화 패턴**(생산자-소비자 문제 등)을 구현할 때 핵심적이다.

조건 변수와 임계 영역

조건 변수는 자체적으로 **상호 배제를 제공하지 않는다.**
따라서 항상 **뮤텍스와 함께 사용**해야 한다.

패턴:

```
pthread_mutex_lock(&mutex);
while (조건이 만족되지 않음) {
    pthread_cond_wait(&cond, &mutex);
}
... 조건이 만족된 상태에서 작업 수행 ...
pthread_mutex_unlock(&mutex);
```

주요 함수

pthread_cond_init()

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);
```

- 조건 변수 초기화
- `attr`는 일반적으로 `NULL` 사용
- 반환값: 0(성공), 에러코드(실패)

pthread_cond_destroy()

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

- 조건 변수 제거

pthread_cond_wait()

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

- 호출 스레드는 `mutex`를 잠근 상태여야 함.
- 함수는 다음을 수행:
    1. `mutex`를 **자동으로 Unlock**.
    2. 조건 변수가 signal/broadcast될 때까지 **Block** 상태로 대기.
    3. 깨운 후 다시 `mutex`를 **Lock** 한 상태로 복귀.

pthread_cond_signal()

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

- 조건 변수를 기다리는 스레드 중 **하나**를 깨움.

pthread_cond_broadcast()

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

- 조건 변수를 기다리는 **모든** 스레드를 깨움.

기본 예제

생산자-소비자 문제 (단순 버전)

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

pthread_mutex_t lock;
pthread_cond_t cond;
int data_ready = 0;

void *producer(void *arg) {
    sleep(1); // 데이터 생성 지연 시뮬레이션
    pthread_mutex_lock(&lock);
    data_ready = 1;
    printf("Producer: data ready, signaling consumer\n");
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&lock);
    return NULL;
}

void *consumer(void *arg) {
    pthread_mutex_lock(&lock);
    while (data_ready == 0) {
```

```
        printf("Consumer: waiting for data\n");
        pthread_cond_wait(&cond, &lock);
    }
    printf("Consumer: data received, processing\n");
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main() {
    pthread_t prod_thread, cons_thread;

    pthread_mutex_init(&lock, NULL);
    pthread_cond_init(&cond, NULL);

    pthread_create(&cons_thread, NULL, consumer, NULL);
    pthread_create(&prod_thread, NULL, producer, NULL);

    pthread_join(prod_thread, NULL);
    pthread_join(cons_thread, NULL);

    pthread_cond_destroy(&cond);
    pthread_mutex_destroy(&lock);

    return 0;
}
```

실행 예시

```
Consumer: waiting for data
Producer: data ready, signaling consumer
Consumer: data received, processing
```

사용 패턴

왜 while 루프를 쓰는가?

```
while (조건이 만족되지 않음) {
    pthread_cond_wait(&cond, &mutex);
}
```

이유:

- **Spurious wakeup** (가짜 깨움) 가능성 존재 → 반드시 **조건을 재확인**해야 안전하다.
- 여러 스레드가 대기 중일 때 특정 스레드가 신호를 받았더라도 조건이 이미 다른 스레드에 의해 변경됐을 수 있다.

Signal vs Broadcast

함수	효과
pthread_cond_signal()	**하나**의 대기 스레드 깨움
pthread_cond_broadcast()	**모든** 대기 스레드 깨움

일반적으로:

- 단일 소비자 → `pthread_cond_signal()`
- 다수 소비자 → `pthread_cond_broadcast()` 고려

Deadlock 방지

- 반드시 **뮤텍스 잠금 상태**에서 `pthread_cond_wait()` 호출
- 조건 변수 대기 전후로 **뮤텍스 해제/획득 패턴을 정확히 준수**해야 Deadlock을 예방할 수 있다.

정리

동작	함수
초기화	`pthread_cond_init()`
제거	`pthread_cond_destroy()`
대기	`pthread_cond_wait()`
신호 보내기	`pthread_cond_signal()`
전체 깨우기	`pthread_cond_broadcast()`

실습 아이디어

- **생산자-소비자 문제 확장** → Circular Queue 구현 + 다수 생산자/소비자
- **Barrier(장벽) 구현** → 모든 스레드가 특정 시점까지 대기 → 조건 만족 시 동시에 진행
- **Thread Pool 작업 대기 큐** → 작업 없을 때 조건 변수로 대기

고급: 시간제한 대기

pthread_cond_timedwait()

```
int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                           const struct timespec *abstime);
```

- 특정 시간까지만 조건 변수 대기
- 타임아웃 발생 시 `ETIMEDOUT` 반환

활용 예:

- **서버 timeout 처리**
- **UI 응답성 개선**

6.4 데드락/경쟁 조건 시뮬레이션

개요

멀티스레드/멀티프로세스 환경에서는 **동기화 문제**가 가장 큰 설계/구현상의 어려움이다.
대표적 문제는:

- **Race Condition (경쟁 조건)**: 여러 스레드가 동시 접근 시 예상치 못한 결과 발생
- **Deadlock (교착 상태)**: 두 개 이상의 스레드가 서로 Lock을 기다리며 영원히 Block 상태가 되는 문제

이들은 모두 **잘못된 동기화 설계**에서 발생한다.

1️⃣ Race Condition (경쟁 조건)

개념

- Race Condition은 **타이밍에 의존한 오류**다.
- 여러 스레드가 **임계 영역 보호 없이** 공유 자원에 접근 시 발생한다.

증상

- 데이터 손상
- 예측 불가능한 실행 결과
- 실행 시마다 다른 결과 발생

예제

뮤텍스 없이 counter 증가

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

#define NUM_THREADS 5
#define NUM_INCREMENTS 100000

int counter = 0;  // 공유 변수

void *thread_func(void *arg) {
    for (int i = 0; i < NUM_INCREMENTS; i++) {
        counter++;  // 경쟁 발생
    }
    return NULL;
```

```
}

int main() {
    pthread_t threads[NUM_THREADS];

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, thread_func, NULL);
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("Final counter value: %d\n", counter);
    return 0;
}
```

실행 결과 예시

```
Final counter value: 423598
(기대값은 5 * 100000 = 500000)
```

→ 매번 실행 시 결과가 다르게 나온다 → **Race Condition 발생**

2️⃣ Deadlock (교착 상태)

개념

- Deadlock은 **스레드들이 서로가 소유한 Lock을 기다리는 상황**이다.
- 조건:
  - 상호 배제
  - 점유와 대기
  - 비선점
  - 순환 대기

증상

- 프로그램이 **멈춤(Hang)**
- CPU 사용량 감소 / 스레드가 영원히 Block 상태

예제

Lock 순서 반전으로 Deadlock 발생

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

pthread_mutex_t lock1;
```

```c
pthread_mutex_t lock2;

void *thread1_func(void *arg) {
    pthread_mutex_lock(&lock1);
    printf("Thread 1 acquired lock1\n");
    sleep(1);  // Deadlock 유도

    pthread_mutex_lock(&lock2);
    printf("Thread 1 acquired lock2\n");

    pthread_mutex_unlock(&lock2);
    pthread_mutex_unlock(&lock1);
    return NULL;
}

void *thread2_func(void *arg) {
    pthread_mutex_lock(&lock2);
    printf("Thread 2 acquired lock2\n");
    sleep(1);  // Deadlock 유도

    pthread_mutex_lock(&lock1);
    printf("Thread 2 acquired lock1\n");

    pthread_mutex_unlock(&lock1);
    pthread_mutex_unlock(&lock2);
    return NULL;
}

int main() {
    pthread_t t1, t2;

    pthread_mutex_init(&lock1, NULL);
    pthread_mutex_init(&lock2, NULL);

    pthread_create(&t1, NULL, thread1_func, NULL);
    pthread_create(&t2, NULL, thread2_func, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    pthread_mutex_destroy(&lock1);
    pthread_mutex_destroy(&lock2);

    return 0;
}
```

실행 결과 예시

```
Thread 1 acquired lock1
Thread 2 acquired lock2
(이후 멈춤 - Deadlock 발생)
```

→ 두 스레드가 서로가 가진 Lock을 기다리며 영원히 Block → **Deadlock 발생**

Deadlock 예방 전략

전략	설명
Lock 순서 고정	항상 동일한 순서로 Lock 획득
Try-lock 사용	`pthread_mutex_trylock()` 으로 실패 시 Backoff
타임아웃 사용	`pthread_mutex_timedlock()` 또는 `pthread_cond_timedwait()` 활용
Lock 분해 설계	가능한 한 Lock 사용 범위를 최소화

안전한 Lock 순서 예시

```
pthread_mutex_lock(&lock1);
pthread_mutex_lock(&lock2);

... 작업 수행 ...

pthread_mutex_unlock(&lock2);
pthread_mutex_unlock(&lock1);
```

모든 스레드에서 동일한 순서로 lock1 → lock2 사용 → Deadlock 발생 방지

경쟁 조건 해결

- 반드시 **뮤텍스 또는 원자적 연산**으로 임계 영역 보호 필요
- 예시:

```
pthread_mutex_lock(&lock);
counter++;
pthread_mutex_unlock(&lock);
```

또는

```
__sync_fetch_and_add(&counter, 1);  // GCC 원자적 연산 built-in
```

정리

문제	발생 원인	증상	해결 방법
Race Condition	보호되지 않은 임계 영역	데이터 손상, 불안정한 결과	뮤텍스 또는 원자적 연산 사용

문제	발생 원인	증상	해결 방법
Deadlock	Lock 순서 반전, 순환 대기 발생	프로그램 멈춤, Hang	Lock 순서 고정, Trylock 사용, 타임아웃 도입

실습 아이디어

- 경쟁 조건: 여러 스레드로 Bank Account 입금/출금 시뮬레이션 → Lock 유무 비교 실험
- Deadlock: 3개 이상의 Lock을 순서 없이 획득하는 코드 → Deadlock 발생 확인
- Deadlock 예방: Trylock으로 Backoff 구현

6.5 CPU 바인딩 (`sched_setaffinity()`)

개요

멀티코어 CPU 환경에서는 **스레드/프로세스를 특정 CPU 코어에 고정(바인딩, Pinning)** 하는 것이 가능하다.
이 기능을 **CPU affinity(affinity = 친화도, 고정)** 라고 부른다.

`cpu affinity`를 사용하면:

- **캐시 효율성** 증가
  스레드가 동일 코어에서 반복 실행 시 **CPU cache locality** 향상
- **특정 코어 전용 작업 실행**
  실시간 처리나 고우선순위 작업을 특정 코어에 고정 가능
- **성능 튜닝**
  NUMA 아키텍처에서 메모리 접근 효율 향상

Linux에서는 `sched_setaffinity()` 와 `sched_getaffinity()` 시스템 콜로 구현된다.

---

sched_setaffinity()

함수 원형

```
#define _GNU_SOURCE
#include <sched.h>

int sched_setaffinity(pid_t pid,
                      size_t cpusetsize,
                      const cpu_set_t *mask);
```

매개변수 설명

매개변수	설명
`pid`	대상 프로세스/스레드 ID 0 → 호출한 자신 (`getpid()` or `gettid()` 사용 가능)
`cpusetsize`	CPU 셋의 크기 (보통 `sizeof(cpu_set_t)`)

매개변수	설명
`mask`	어떤 CPU에 바인딩할지 지정한 마스크 (`cpu_set_t` 타입)

반환값

- `0` → 성공
- `-1` → 실패 (`errno` 설정됨)

CPU_SET 매크로

Linux에서는 `cpu_set_t` 라는 CPU 집합 구조체를 사용하며, 다음과 같은 매크로로 조작한다:

매크로	설명
`CPU_ZERO(&set)`	CPU 집합 초기화 (비움)
`CPU_SET(cpu, &set)`	특정 CPU를 set에 추가
`CPU_CLR(cpu, &set)`	특정 CPU를 set에서 제거
`CPU_ISSET(cpu, &set)`	특정 CPU가 set에 포함되어 있는지 확인

---

sched_getaffinity()

함수 원형

```
#define _GNU_SOURCE
#include <sched.h>

int sched_getaffinity(pid_t pid,
                      size_t cpusetsize,
                      cpu_set_t *mask);
```

- 현재 프로세스/스레드의 CPU affinity를 읽어옴
- 반환값 및 사용법은 `sched_setaffinity()` 와 동일

기본 예제

```
#define _GNU_SOURCE
#include <stdio.h>
#include <sched.h>
#include <unistd.h>

int main() {
    cpu_set_t set;
    int cpu;
```

```
    // 현재 프로세스의 CPU affinity 조회
    CPU_ZERO(&set);
    if (sched_getaffinity(0, sizeof(set), &set) == -1) {
        perror("sched_getaffinity");
        return 1;
    }

    printf("Current CPU affinity: ");
    for (cpu = 0; cpu < CPU_SETSIZE; cpu++) {
        if (CPU_ISSET(cpu, &set))
            printf("%d ", cpu);
    }
    printf("\n");

    // CPU affinity 설정: CPU 0번만 사용하도록 고정
    CPU_ZERO(&set);
    CPU_SET(0, &set);

    if (sched_setaffinity(0, sizeof(set), &set) == -1) {
        perror("sched_setaffinity");
        return 1;
    }

    printf("CPU affinity set to CPU 0\n");

    while (1) {
        // CPU 0에서 무한 루프 실행
    }

    return 0;
}
```

실행 결과 예시

```
Current CPU affinity: 0 1 2 3
CPU affinity set to CPU 0
```

→ 이후 프로세스는 **CPU 0에서만 실행**됨 → 다른 CPU로 스케줄링되지 않음.

멀티스레드에서 사용

스레드 별 affinity 설정:

- Linux에서는 **스레드마다 독립적인 affinity 설정 가능**하다.
- `pthread_self()` → `gettid()` 필요:
    - Linux의 **스레드 ID는 `gettid()`로 구해야 정확함** → `getpid()`는 전체 프로세스의 PID 반환.

```
#define _GNU_SOURCE
#include <stdio.h>
```

```c
#include <pthread.h>
#include <unistd.h>
#include <sched.h>
#include <sys/syscall.h>

#define gettid() syscall(SYS_gettid)

void *thread_func(void *arg) {
    cpu_set_t set;
    CPU_ZERO(&set);
    CPU_SET(1, &set);  // CPU 1로 고정

    if (sched_setaffinity(gettid(), sizeof(set), &set) == -1) {
        perror("sched_setaffinity");
        return NULL;
    }

    while (1) {
        // CPU 1에서 반복 실행
    }

    return NULL;
}

int main() {
    pthread_t thread;

    pthread_create(&thread, NULL, thread_func, NULL);

    pthread_join(thread, NULL);

    return 0;
}
```

→ 위 예제에서 스레드는 **CPU 1로 고정**되어 실행된다.

주의사항

- 프로세스/스레드 affinity는 **커널 스케줄러 힌트**일 뿐 → 강제적 완전 고정은 아님 (시그널 핸들링 등으로 이동 가능성 존재).
- **NUMA 시스템**에서는 CPU affinity와 **메모리 affinity**도 같이 고려해야 최적 성능 가능.
- **잘못된 affinity 설정**(예: 모든 스레드를 CPU 0으로 고정)은 성능 저하를 유발할 수 있다.
- affinity 설정은 **실시간 시스템, 고성능 네트워크 처리, 데이터베이스 튜닝** 등에서 유용하다.

정리

기능	함수
CPU affinity 설정	`sched_setaffinity()`
CPU affinity 조회	`sched_getaffinity()`
CPU set 관리	`CPU_ZERO()`, `CPU_SET()`, `CPU_CLR()`, `CPU_ISSET()`

실습 아이디어

- **멀티스레드 서버 성능 튜닝**: 특정 스레드를 고정 CPU에 바인딩 → **스레드간 캐시 경합(Cache Thrashing) 감소**
- **실시간 미디어 처리 스레드**만 고정 CPU 할당 → latency 감소
- **멀티큐 NIC(Network Interface Card)** → IRQ affinity와 연계한 CPU affinity 튜닝

실습

다중 쓰레드 파일 다운로드 시뮬레이터

목표

멀티스레드 프로그래밍에서 자주 등장하는 패턴 중 하나가 **파일을 여러 부분으로 나누어 동시에 다운로드**하는 것이다.
실제 HTTP Range 요청과는 다르지만, 여기서는 **파일을 블록 단위로 나누어 동시에 읽어와 병합**하는 형태의 **시뮬레이터**를 C 언어로 구현해보자.

- 각 쓰레드는 **파일의 일부분**만 읽는다.
- 쓰레드는 해당 블록을 **공유 버퍼에 기록**한다.
- 최종적으로 전체 파일이 병합된다.

구조

```
+------------+------------+------------+------------+
|  Thread 1  |  Thread 2  |  Thread 3  |  Thread N  |
+------------+------------+------------+------------+
        \           \            \           \
         \           \            \           \
          +-------- Shared Buffer / Output File --------+
```

- **Mutex** 사용: 여러 쓰레드가 공유 버퍼나 출력 파일에 동시에 접근하지 않도록 보호
- **조건 변수(옵션)**: 특정 상황에서는 쓰레드간 동기화도 가능

예제 코드 (단일 파일, 여러 블록 읽기 시뮬레이터)

전제

- `input.dat`라는 파일이 있다고 가정한다 (대용량 파일이면 효과가 더 뚜렷함).
- 쓰레드가 나눠서 읽고 `output.dat`에 동일한 순서로 쓰는 예제.

코드

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

#define NUM_THREADS 4
#define BLOCK_SIZE  1024 * 1024  // 1MB

typedef struct {
    int thread_id;
    off_t start_offset;
    size_t size;
    const char *input_filename;
    FILE *output_file;
    pthread_mutex_t *output_lock;
} thread_arg_t;

void *download_block(void *arg) {
    thread_arg_t *targ = (thread_arg_t *)arg;
    char *buffer = malloc(targ->size);
    if (!buffer) {
        perror("malloc");
        return NULL;
    }

    FILE *input_file = fopen(targ->input_filename, "rb");
    if (!input_file) {
        perror("fopen input");
        free(buffer);
        return NULL;
    }

    fseeko(input_file, targ->start_offset, SEEK_SET);
    fread(buffer, 1, targ->size, input_file);
    fclose(input_file);

    // 출력 파일에 쓰기 (뮤텍스 보호 필요)
    pthread_mutex_lock(targ->output_lock);
    fseeko(targ->output_file, targ->start_offset, SEEK_SET);
    fwrite(buffer, 1, targ->size, targ->output_file);
```

```
    pthread_mutex_unlock(targ->output_lock);

    printf("Thread %d: copied offset %ld (%zu bytes)\n", targ->thread_id, targ->start_offset, targ->size);

    free(buffer);
    return NULL;
}

int main() {
    pthread_t threads[NUM_THREADS];
    thread_arg_t thread_args[NUM_THREADS];
    pthread_mutex_t output_lock;

    struct stat st;
    if (stat("input.dat", &st) == -1) {
        perror("stat");
        return 1;
    }
    off_t file_size = st.st_size;
    printf("Input file size: %ld bytes\n", file_size);

    FILE *output_file = fopen("output.dat", "wb+");
    if (!output_file) {
        perror("fopen output");
        return 1;
    }

    // output.dat 파일 사이즈 미리 확보
    ftruncate(fileno(output_file), file_size);

    pthread_mutex_init(&output_lock, NULL);

    for (int i = 0; i < NUM_THREADS; i++) {
        off_t start = i * (file_size / NUM_THREADS);
        size_t size = (i == NUM_THREADS - 1)
                        ? (file_size - start)
                        : (file_size / NUM_THREADS);

        thread_args[i].thread_id = i;
        thread_args[i].start_offset = start;
        thread_args[i].size = size;
        thread_args[i].input_filename = "input.dat";
        thread_args[i].output_file = output_file;
        thread_args[i].output_lock = &output_lock;

        pthread_create(&threads[i], NULL, download_block, &thread_args[i]);
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

```

```
    pthread_mutex_destroy(&output_lock);
    fclose(output_file);

    printf("Download simulation complete.\n");
    return 0;
}
```

실행 흐름

1. `input.dat` 파일 크기 확인
2. NUM_THREADS 개수만큼 쓰레드 생성
3. 각 쓰레드는 지정된 **오프셋(start_offset)** 부터 자신의 블록을 읽음
4. 읽은 데이터를 **뮤텍스 보호 하에 output.dat 에 기록**
5. 모든 쓰레드 완료 후 프로그램 종료

실행 예시

```
$ ls -lh input.dat
-rw-r--r-- 1 user user 16M input.dat

$ gcc -pthread download_sim.c -o download_sim
$ ./download_sim
Input file size: 16777216 bytes
Thread 0: copied offset 0 (4194304 bytes)
Thread 1: copied offset 4194304 (4194304 bytes)
Thread 2: copied offset 8388608 (4194304 bytes)
Thread 3: copied offset 12582912 (4194304 bytes)
Download simulation complete.
```

→ output.dat가 input.dat와 **동일한 내용으로 완성**됨

개선 아이디어

- **I/O 스케줄링**: 쓰레드간 작업 배치 최적화
- **I/O Priority 조절** (`ionice` 사용 가능)
- **Buffer 크기 튜닝**
- **동적 작업 할당**: 각 쓰레드가 유휴 상태일 때 다음 블록 할당받기 (Work stealing)

정리

기능	구현
블록 분할	쓰레드별 `start_offset`, `size` 계산
병렬 다운로드	`pthread_create()` 사용

기능	구현
출력 보호	`pthread_mutex_t` 사용
파일 병합	모든 블록을 output.dat에 순서대로 기록

실습 아이디어

- **진짜 HTTP Range 요청과 연계**: curl/libcurl + multi-thread + output merge
- **TCP 서버에서 블록별 수신** 후 병합
- **디스크 I/O 벤치마크 툴 제작**: IOPS 측정

데드락 상황 생성 및 해결

1️⃣ 데드락(Deadlock)이란?

데드락이란 **둘 이상의 스레드(또는 프로세스)가 서로가 보유한 자원을 기다리면서 영원히 Block 상태**가 되는 현상이다.

발생 조건 (Coffman 조건 4가지)

조건	설명
상호 배제	자원은 한 번에 하나의 스레드만 사용 가능
점유와 대기	자원을 점유한 채 다른 자원을 요청
비선점	점유한 자원을 강제로 빼앗을 수 없음
순환 대기	스레드들이 자원을 순환 구조로 기다림

→ 이 4가지 조건이 모두 만족되면 데드락 발생 가능.

2️⃣ 데드락 상황 생성 예제

시나리오

- `Thread 1` → `lock1` 획득 후 `lock2` 획득 시도
- `Thread 2` → `lock2` 획득 후 `lock1` 획득 시도

→ **Lock 순서 반전**으로 Deadlock 발생 가능.

코드 예제

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

pthread_mutex_t lock1;
pthread_mutex_t lock2;
```

```
7
8  void *thread1_func(void *arg) {
9      pthread_mutex_lock(&lock1);
10     printf("Thread 1 acquired lock1\n");
11     sleep(1);  // 의도적 지연
12
13     pthread_mutex_lock(&lock2);
14     printf("Thread 1 acquired lock2\n");
15
16     pthread_mutex_unlock(&lock2);
17     pthread_mutex_unlock(&lock1);
18     return NULL;
19 }
20
21 void *thread2_func(void *arg) {
22     pthread_mutex_lock(&lock2);
23     printf("Thread 2 acquired lock2\n");
24     sleep(1);  // 의도적 지연
25
26     pthread_mutex_lock(&lock1);
27     printf("Thread 2 acquired lock1\n");
28
29     pthread_mutex_unlock(&lock1);
30     pthread_mutex_unlock(&lock2);
31     return NULL;
32 }
33
34 int main() {
35     pthread_t t1, t2;
36
37     pthread_mutex_init(&lock1, NULL);
38     pthread_mutex_init(&lock2, NULL);
39
40     pthread_create(&t1, NULL, thread1_func, NULL);
41     pthread_create(&t2, NULL, thread2_func, NULL);
42
43     pthread_join(t1, NULL);
44     pthread_join(t2, NULL);
45
46     pthread_mutex_destroy(&lock1);
47     pthread_mutex_destroy(&lock2);
48
49     return 0;
50 }
```

실행 결과 예시

```
1  Thread 1 acquired lock1
2  Thread 2 acquired lock2
3  (이후 멈춤 - Deadlock 발생)
```

→ 각 스레드가 서로의 Lock을 기다리면서 **영원히 Block 상태 진입** → 데드락 발생.

3 데드락 해결 방법

원칙: 데드락 회피 또는 예방

방법	설명
Lock 순서 고정	모든 스레드가 동일한 순서로 Lock 획득
Try-Lock 사용	`pthread_mutex_trylock()` 사용 → 실패 시 Backoff or Retry
Timeout 도입	`pthread_mutex_timedlock()` 사용 (POSIX 옵션)
Lock 수 줄이기	가능한 Lock 영역 최소화
순환 대기 차단	자원 요청 시 전체 순서 명확히 지정

해결 1: Lock 순서 고정

모든 스레드가 lock1 → lock2 순서로만 Lock 획득

```
void *thread1_func(void *arg) {
    pthread_mutex_lock(&lock1);
    printf("Thread 1 acquired lock1\n");
    sleep(1);

    pthread_mutex_lock(&lock2);
    printf("Thread 1 acquired lock2\n");

    pthread_mutex_unlock(&lock2);
    pthread_mutex_unlock(&lock1);
    return NULL;
}

void *thread2_func(void *arg) {
    pthread_mutex_lock(&lock1);  // 순서 고정
    printf("Thread 2 acquired lock1\n");
    sleep(1);

    pthread_mutex_lock(&lock2);
    printf("Thread 2 acquired lock2\n");

    pthread_mutex_unlock(&lock2);
    pthread_mutex_unlock(&lock1);
    return NULL;
}
```

→ 데드락 발생하지 않음.

해결 2: Try-Lock + Backoff 패턴

```
void *thread2_func(void *arg) {
    while (1) {
        pthread_mutex_lock(&lock2);
        printf("Thread 2 acquired lock2\n");

        if (pthread_mutex_trylock(&lock1) == 0) {
            printf("Thread 2 acquired lock1\n");

            pthread_mutex_unlock(&lock1);
            pthread_mutex_unlock(&lock2);
            break;
        } else {
            printf("Thread 2 failed to acquire lock1, releasing lock2 and retrying\n");
            pthread_mutex_unlock(&lock2);
            usleep(100000);  // Backoff (100ms)
        }
    }
    return NULL;
}
```

→ **trylock** 실패 시 lock2 해제 후 재시도 → Deadlock 회피 가능.

해결 3: Timeout 기반 Lock

- `pthread_mutex_timedlock()` 사용 → 일정 시간 내에 Lock 실패 시 timeout 발생 (glibc 지원 필요).
- 실시간 시스템에 유용.

```
struct timespec ts;
clock_gettime(CLOCK_REALTIME, &ts);
ts.tv_sec += 1;  // 1초 timeout

if (pthread_mutex_timedlock(&lock1, &ts) == ETIMEDOUT) {
    printf("Thread failed to acquire lock1 (timeout)\n");
}
```

---

4 정리

방법	장점	단점
Lock 순서 고정	간단, 효과적	코드 전체 일관성 필요
Try-Lock + Backoff	Deadlock 회피	구현 복잡성 증가, 성능 저하 가능
Timeout Lock	실시간성 확보	플랫폼/호환성 문제, 복잡성 증가

실습 아이디어

- **3개 이상의 Lock으로 Deadlock 발생 테스트**
- Try-Lock으로 **Adaptive Backoff** 구현 (Backoff 시간 점진적 증가)
- Thread Pool에서 Lock 순서 고정 설계 연습

결론

- Deadlock은 **설계단계에서 예방**하는 것이 최선이다.
- 항상:
  - **Lock 순서 일관성 유지**
  - **Lock 최소화**
  - 필요시 **Try-Lock/Timeout** 도입
- 성능 최적화보다 **안정성 우선 설계**가 중요.

생산자-소비자 문제 C로 구현

개요

생산자-소비자 문제(Producer-Consumer Problem) 는 멀티스레드 동기화의 **고전적 예제**다.

구성

- **생산자(Producer)**: 데이터를 생성해서 **버퍼에 추가**함
- **소비자(Consumer)**: 버퍼에서 데이터를 **소비(제거)** 함
- 버퍼는 **유한 크기**를 가지므로:
  - 가득 찼으면 생산자는 **대기**
  - 비었으면 소비자는 **대기**

동기화 필요성

- **뮤텍스(Mutex)**: 버퍼 접근 시 상호 배제 보장
- **조건 변수(Condition Variable)**:
  - 버퍼 **가득 참/비어 있음** 상태를 스레드에게 알림

1️⃣ 구현 개요

버퍼 구조

```
#define BUFFER_SIZE 10
int buffer[BUFFER_SIZE];
int count = 0;
int in = 0;
int out = 0;
```

- `count` → 현재 버퍼에 있는 데이터 개수
- `in` → 생산자가 데이터를 추가할 위치
- `out` → 소비자가 데이터를 가져올 위치

동기화 도구

```
pthread_mutex_t mutex;
pthread_cond_t not_full;
pthread_cond_t not_empty;
```

- `mutex` → 임계 영역 보호
- `not_full` → 버퍼가 가득 차지 않았을 때 생산자 대기 해제
- `not_empty` → 버퍼가 비어 있지 않을 때 소비자 대기 해제

2️⃣ 전체 코드 예제

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

#define BUFFER_SIZE 10
#define PRODUCE_ITEMS 20

int buffer[BUFFER_SIZE];
int count = 0;
int in = 0;
int out = 0;

pthread_mutex_t mutex;
pthread_cond_t not_full;
pthread_cond_t not_empty;

void *producer(void *arg) {
    for (int i = 1; i <= PRODUCE_ITEMS; i++) {
        pthread_mutex_lock(&mutex);

        // 버퍼가 가득 차 있으면 대기
        while (count == BUFFER_SIZE) {
            pthread_cond_wait(&not_full, &mutex);
        }

        // 버퍼에 데이터 추가
        buffer[in] = i;
        in = (in + 1) % BUFFER_SIZE;
        count++;

        printf("Producer: produced item %d (count = %d)\n", i, count);

```

```
        // 소비자에게 알림
        pthread_cond_signal(&not_empty);

        pthread_mutex_unlock(&mutex);

        usleep(100000);  // 0.1초 sleep → 생산 속도 조절
    }
    return NULL;
}

void *consumer(void *arg) {
    for (int i = 1; i <= PRODUCE_ITEMS; i++) {
        pthread_mutex_lock(&mutex);

        // 버퍼가 비어 있으면 대기
        while (count == 0) {
            pthread_cond_wait(&not_empty, &mutex);
        }

        // 버퍼에서 데이터 소비
        int item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;

        printf("Consumer: consumed item %d (count = %d)\n", item, count);

        // 생산자에게 알림
        pthread_cond_signal(&not_full);

        pthread_mutex_unlock(&mutex);

        usleep(150000);  // 0.15초 sleep → 소비 속도 조절
    }
    return NULL;
}

int main() {
    pthread_t prod_thread, cons_thread;

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&not_full, NULL);
    pthread_cond_init(&not_empty, NULL);

    pthread_create(&prod_thread, NULL, producer, NULL);
    pthread_create(&cons_thread, NULL, consumer, NULL);

    pthread_join(prod_thread, NULL);
    pthread_join(cons_thread, NULL);

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&not_full);
    pthread_cond_destroy(&not_empty);

```

```
    return 0;
}
```

3 실행 결과 예시

```
Producer: produced item 1 (count = 1)
Consumer: consumed item 1 (count = 0)
Producer: produced item 2 (count = 1)
Producer: produced item 3 (count = 2)
Consumer: consumed item 2 (count = 1)
...
Producer: produced item 20 (count = 2)
Consumer: consumed item 20 (count = 1)
```

→ 생산자와 소비자가 **상호 협력**하여 버퍼를 적절히 채우고 비움.

4 동작 흐름

1. 생산자는 버퍼가 가득 차면 `not_full` 조건 변수에서 대기
2. 소비자가 아이템을 소비 → `pthread_cond_signal(&not_full)` 로 생산자 깨움
3. 소비자는 버퍼가 비면 `not_empty` 조건 변수에서 대기
4. 생산자가 아이템을 추가 → `pthread_cond_signal(&not_empty)` 로 소비자 깨움
5. 반복

5 핵심 포인트

요소	사용 이유
`pthread_mutex_t mutex`	버퍼 접근 시 상호 배제
`pthread_cond_t not_full`	버퍼가 가득 찼을 때 생산자 대기
`pthread_cond_t not_empty`	버퍼가 비었을 때 소비자 대기
`while` 루프 사용	**Spurious wakeup** 대응 → 항상 조건 재확인 필요

6 실습 아이디어

- **다수 생산자/다수 소비자** 구현해보기
- 생산/소비 속도를 다르게 조정해 race condition 실험하기
- `pthread_cond_broadcast()` 사용 실험
- `pthread_cond_timedwait()` 사용해 timeout 처리해보기

7️⃣ 결론

- **생산자-소비자 문제**는 멀티스레드 동기화 설계의 대표적 예제.
- 반드시:
  - **뮤텍스 + 조건 변수** 조합 사용
  - while 조건 확인 패턴 적용 → **깨운다고 바로 조건이 만족되는 것 아님!**
- 실제 응용: **Thread Pool**, **I/O Buffering**, **Pipeline Architecture** 등에 핵심적으로 사용된다.