Lecture 12

Multithreading

리뷰

- 동적 메모리 할당 `malloc()`
  - 동작
    - 작은 메모리 요청: `brk`, `sbrk`를 호출하여 힙을 조정함
    - 큰 메모리 요청: `mmap`, `munmap`을 호출하여 OS에 메모리 블록을 요청하거나 반환함
  - 구현
    - Tracking: implicit(묵시적)/explicit(명시적) 가용 리스트
    - Placement: first-fit, next-fit, best-fit
    - Splitting: 가용 블록 전체 할당, 둘로 나눠 할당
    - Coalescing: 경계 태그 활용, doubly linked list 활용
  - 메모리 할당 관련 이슈
    - 단편화
    - 메모리 누수

리뷰

- 쓰레기 수집기
  - C 언어에서 쓰레기 수집기는 구현되지 않음
  - 필요 시 쓰레기 수집기를 구현할 수 있음
    - 참조 카운팅(reference counting)
    - Mark-and-Sweep
    - 부수적 쓰레기 수집기(conservative garbage collector)

병렬 컴퓨팅

- 동시에 많은 계산을 하는 연산의 한 방법
  - 명령어 수준 병렬
    - 명령어 파이프라인(instruction pipeline): 하나의 명령어가 실행되는 도중에 다른 명령어 실행을 시작하는 식으로 동시에 여러 개의 명령어를 실행하는 기법
    - N 스테이지 파이프라인은 N 개 만큼의 다른 명령어들을 다른 완료된 단계에서 가질 수 있음
  - 자료 병렬 처리
    - 프로그램 루프에 내재된 병렬화
    - 프로그램 루프는 병렬로 처리된 다른 컴퓨팅 노드들의 자료를 분산시키는데 초점을 맞춤
    - 루프의 중속성은 병렬화 루프에 의해 막힘
  - 작업 병렬 처리
    - 프로그램을 구성하는 여러 개의 작업을 독립적으로 나누어, 각각을 병렬적으로 수행함

병렬 컴퓨팅

- 동시에 많은 계산을 하는 연산의 한 방법
  - 자료 병렬 처리

병렬화 가능

```
float params[10];
int k;
for (k=0; k<10; k++)
   func_1(params[k]);
```

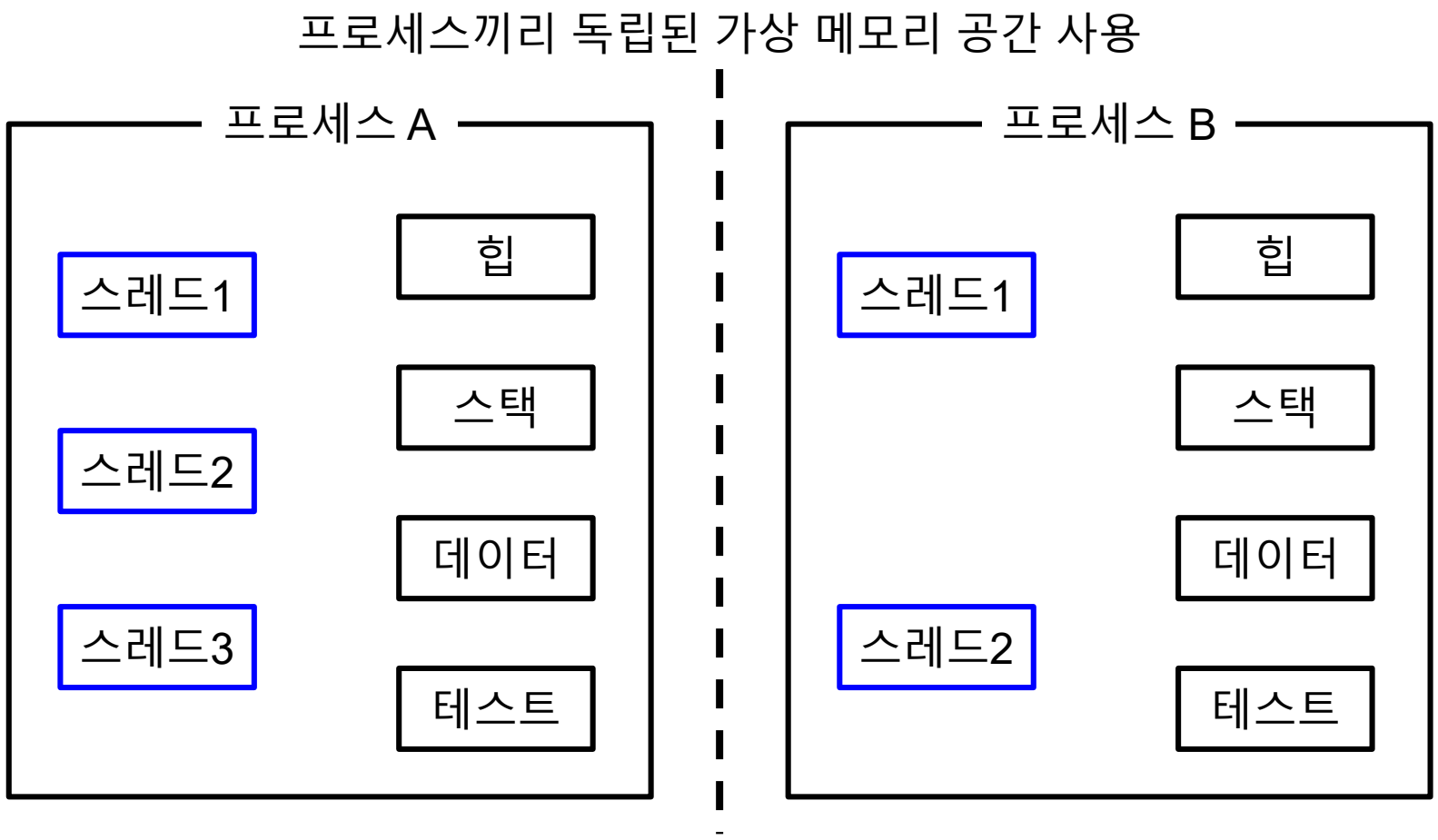
병렬화 불가능

```
float params[10];
float prev=0;
int k;
for (k=0; k<10; k++)
   func_2(params[k],prev);
```

Process vs. Thread

- 프로세스(process)
  - 운영체제로부터 자원을 받아 메모리에 올라와 실행 중인 프로그램
  - 각 프로세스는 독립적인 실행 환경을 가지며, 다른 프로세스와 격리되어 자신의 가상 메모리 공간, 레지스터, 파일 디스크립터 등을 소유함
  - 프로세스 간 통신
    - 공유 메모리
    - 네트워크
    - 파이프(pipe), 큐(queue)
- 스레드(thread)
  - 프로세스 내에서 실행되는 실행 흐름의 단위
  - 하나의 프로세스는 하나 이상의 스레드를 가질 수 있음
  - 각 스레드는 고유한 프로그램 카운터(PC), 레지스터 세트, 스택을 갖고 있지만, 동일한 프로세스 내의 다른 스레드와 코드, 데이터, 힙 영역을 공유함

Process vs. Thread



Multithreading

- 프로세스(process)
  - 현재까지 실습했던 프로그램들은 main 스레드만 갖고 있음
  - 별도의 스레드 생성 없이, main 스레드가 종료되면 자동으로 종료됨
- 밀티스레딩(multithreading)
  - 하나의 프로세스 내에서 여러 스레드가 동시에 실행되는 것을 의미함
  - Main 스레드 외에 여러 개의 스레드를 생성해 병렬로 실행할 수 있음
  - 스레드 간의 통신이 가능하며 프로세스 간의 통신보다 간단함
  - ☺ **장점**: 자원 공유 용이성, 응답성 향상, 효율적인 자원 활용, 경량화
  - ☹ **단점**: 동기화 문제, 디버깅의 어려움, 교착 상태(deadlock) 위험

Multithreading

```
int balance=500;
void deposit(int sum) {
   int cur_balance=balance; // read
   ...
   cur_balance+=sum;
   balance=cur_balance; // write
}
void withdraw(int sum) {
   int cur_balance=balance; // read
   if (cur_balance>0) cur_balance-=sum;
   balance=cur_balance; // write
}
...
deposite(100); // thread 1 (T1)
...
withdraw(50); // thread 2 (T2)
...
```

- 시나리오 1: T1(read) → T2(read,write) → T1(write), balance=600
- 시나리오 2: T2(read) → T1(read,write) → T2(write), balance=450

→ 전역/전적 변수 사용을 최소화해야 함

Multithreading

- C 언어에서 여러 개의 방법으로 멀티스레딩 프로그래밍이 가능함
  - POSIX C의 pthread 라이브러리
  - OpenMP
  - Intel의 스레딩 빌딩 블록
  - Cilk
  - Grand central dispatch (GCD)
  - CUDA (GPU)
  - OpenCL (GPU/CPU)

POSIX 스레드(pthread)

- pthread의 주요 기능
  - 스레드 생성과 종료
    - `pthread_create()`: 새로운 스레드 생성
    - `pthread_exit()`: 스레드 종료
  - 스레드 동기화
    - 뮤텍스(mutex): 상호 배제를 통해 공유 자원의 동시 접근을 제어
    - 조건 변수(condition variable): 특정 조건이 만족될 때까지 스레드를 대기시키고, 조건 충족 시 깨우는 기능
  - 스레드 관리
    - `pthread_join()`: 특정 스레드가 종료될 때까지 대기
    - `pthread_detach()`: 스레드를 독립적으로 실행시켜 종료 시 자원 자동 반환
  - 스레드 속성 제어
    - 우선순위, 스택 크기, 스케줄링 정책 등 조정 가능
- GCC와의 컴파일: `gcc infile.c -o outfile  -pthread`

스레드 생성 및 종료

- `int pthread_create(pthread_t *thread,`
  `const pthread_attr_t *attr,`
  `void *(*start_routine)(void *),`
  `void *arg);`
  - `attr`로 지정한 속성을 갖는 스레드를 생성함
  - `attr=NULL`일 경우 기본 속성을 갖는 스레드 생성
  - 성공 시 새로 생성된 스레드는 `thread`에 저장됨
  - 스레드가 실행할 함수는 `start_rountine(arg)` 임
  - 리턴 값: 성공 시 `0`, 실패 시 `0`이 아닌 오류 코드
- `void pthread_exit(void *retval);`
  - 스레드가 실행할 함수가 끝날 때 자동 불어오게 됨
  - `exit()` 함수와 비슷함

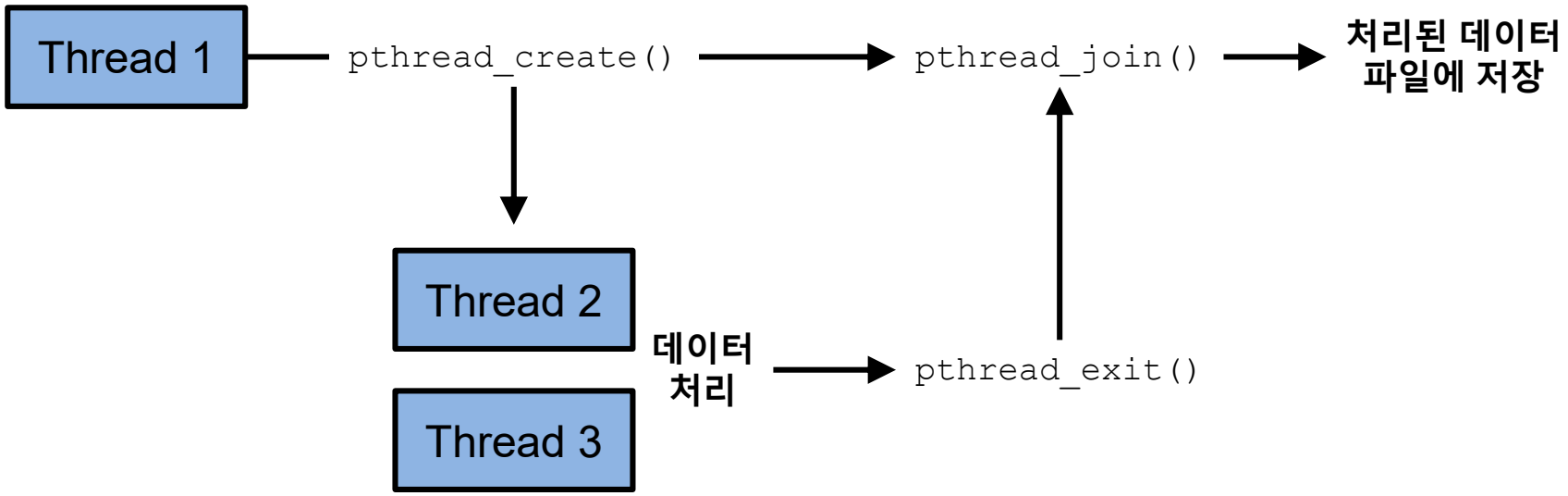
스레드 생성 및 종료


```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *printHello(void *threadID) {
   long tid;
   tid = (long)threadID;
   printf("Hello from thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for (t=0; t<NUM_THREADS; t++) {
      printf("In main: creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, printHello, (void*)t);
      if (rc) {
         printf("ERROR: %d\n", rc);
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

```
In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
In main: creating thread 3
In main: creating thread 4
Hello from thread #1!
Hello from thread #2!
Hello from thread #0!
Hello from thread #3!
Hello from thread #4!
```

동기화: pthread_join

- `int pthread_join(pthread_t thread, void **retval);`
  - 호출한 스레드를 멈춰주고, 대상 스레드가 종료될 때까지 기다림
  - 스레드들이 정해진 순서로 실행되도록 강제할 수 있음
- 다른 동기화 방법: 뮤텍스, 조건 변수



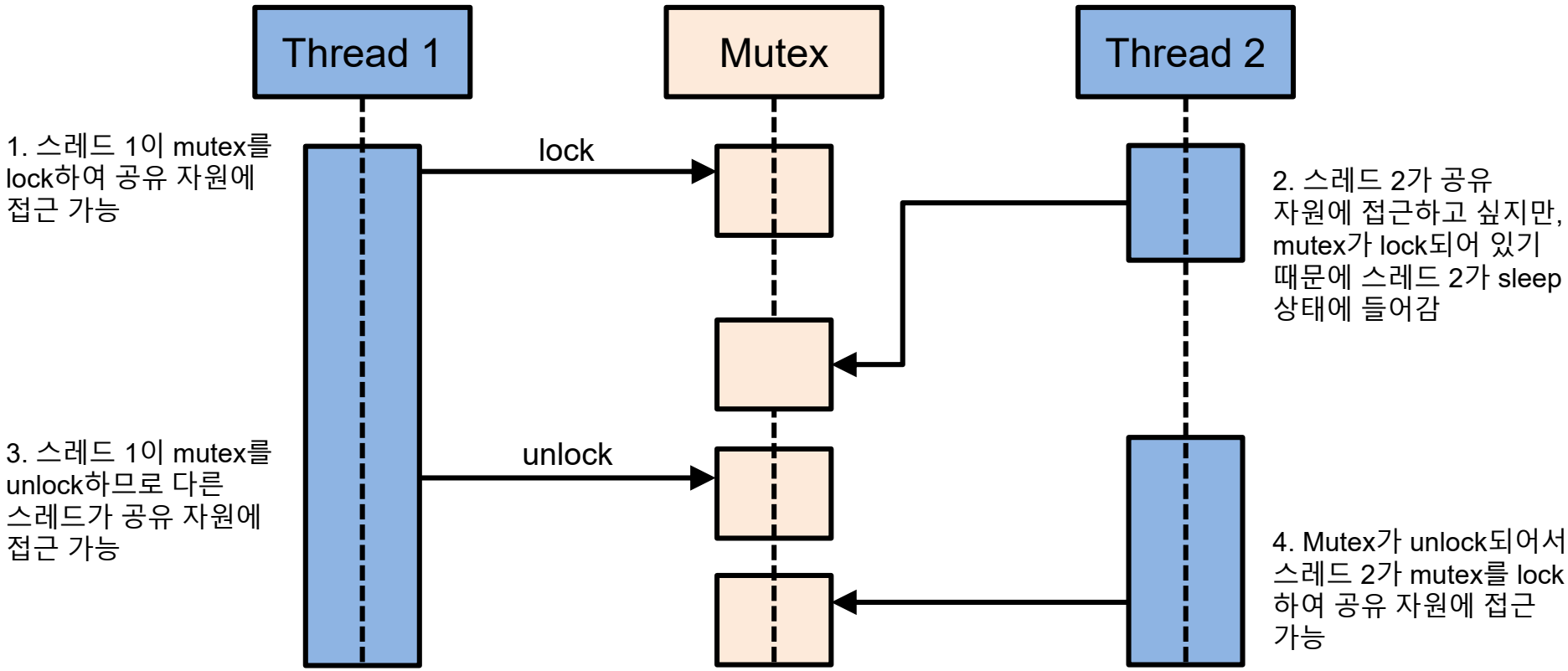
동기화: pthread_join


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

void *worker(void *arg) {
    int num = *((int*)arg);
    printf("Thread(%d) completed\n", num);
    int *result = malloc(sizeof(int));
    *result = num * num;
    pthread_exit((void*)result);
}

int main() {
    pthread_t tid1, tid2;
    int a = 2, b = 3;
    int *res1, *res2;

    pthread_create(&tid1, NULL, worker, &a);
    pthread_create(&tid2, NULL, worker, &b);

    // 동기화: 스레드 종료까지 기다림
    pthread_join(tid1, (void**)&res1); // tid1 스레드가 끝날 때까지 메인 스레드 대기
    pthread_join(tid2, (void**)&res2); // tid2 스레드가 끝날 때까지 메인 스레드 대기

    printf("Result 1 = %d\n", *res1);
    printf("Result 2 = %d\n", *res2);

    free(res1);
    free(res2);

    return 0;
}
```

```
Thread(2) completed
Thread(3) completed
Result 1 = 4
Result 2 = 9
```

동기화: 뮤텍스(mutex)

- Mutex: 실행 중 공유 자원에 접근할 때 충돌 방지 방법



동기화: 뮤텍스(mutex)

- Mutex 사용의 일반적인 흐름
  - 초기화

```
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);
// PTHREAD_MUTEX_INITIALIZER 매크로를 사용해도 됨
```

  - 임계 구역 진입 전 잠금

```
pthread_mutex_lock(&lock);
// 공유 자원(전역 변수, 파일, 버퍼 등)에 안전하게 접근
```

  - 임계 구역 종료 후 해제

```
pthread_mutex_unlock(&lock);
```

  - 사용 종료 시 mutex 해제

```
pthread_mutex_destroy(&lock);
```

동기화: 뮤텍스(mutex)

- `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);`
  - `mutex`를 `attr` 속성으로 초기화함
  - `attr=NULL`일 경우 기본 속성은 선택됨
  - 성공 시 `0`, 실패 시 오류 코드를 리턴함
- `int pthread_mutex_destroy(pthread_mutex_t *mutex);`
  - 사용이 끝난 `mutex`를 해제함
  - 성공 시 `0`, 실패 시 오류 코드를 리턴함
- `int pthread_mutex_lock(pthread_mutex_t *mutex);`
  - `mutex`를 잠그고 다른 스레드 접근을 막음
  - 성공 시 `0`, 실패 시 오류 코드를 리턴함

동기화: 뮤텍스(mutex)

- `int pthread_mutex_unlock(pthread_mutex_t *mutex);`
  - `mutex`를 해제하여 다른 스레드가 접근할 수 있게 함
  - 성공 시 `0`, 실패 시 오류 코드를 리턴함
- `int pthread_mutex_trylock(pthread_mutex_t *mutex);`
  - `mutex` 잠금을 시도함
  - 이미 잠겨 있으면 `EBUSY`, 성공 시 `0`, 그 외 오류 시 오류 코드를 리턴함

동기화: 뮤텍스(mutex)


```
int balance=500;
void deposit(int sum) {
   int cur_balance=balance; // read
   ...
   cur_balance+=sum;
   balance=cur_balance; // write
}
void withdraw(int sum) {
   int cur_balance=balance; // read
   if (cur_balance>0) cur_balance-=sum;
   balance=cur_balance; // write
}
...
deposite(100); // thread 1 (T1)
...
withdraw(50); // thread 2 (T2)
...
```

- 시나리오 1: T1(read) → T2(read,write) → T1(write), balance=600
- 시나리오 2: T2(read) → T1(read,write) → T2(write), balance=450

동기화: 뮤텍스(mutex)


```
int balance=500;
pthread_mutex_t mbalance=PTHREAD_MUTEX_INITIALIZER;
void deposit(int sum) {
    pthread_mutex_lock(&mbalance);
    int cur_balance=balance; // read
    ...
    cur_balance+=sum;
    balance=cur_balance; // write
    pthread_mutex_unlock(&mbalance);
}
void withdraw(int sum) {
    pthread_mutex_lock(&mbalance);
    int cur_balance=balance; // read
    if (cur_balance>0) cur_balance-=sum;
    balance=cur_balance; // write
    pthread_mutex_unlock(&mbalance);
}
...
deposite(100); // thread 1 (T1)
...
withdraw(50); // thread 2 (T2)
...
```

어떤 시나리오에 걸리든 balance=550

동기화: 조건 변수

- 스레드가 어떤 조건이 만족될 때까지 대기하거나, 조건이 만족되면 다른 스레드가 신호를 보내 깨움
- 레이스 컨디션(race condition) 방지를 위해 뮤텍스와 함께 사용
- 조건 변수 사용의 일반적인 흐름
  - 공유 자원 접근 전 `mutex` 잠금
  - 원하는 조건이 만족하지 않으면 `pthread_cond_wait()` 호출 → 자동으로 `mutex` 해제 후 대기
  - 다른 스레드가 조건을 만족시키면 `pthread_cond_signal()`이나 `pthread_cond_broadcast()`로 깨움
  - 깨어난 스레드는 다시 `mutex`를 획득하고 조건 검사 후 작업 실행

동기화: 조건 변수

- `int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);`
  - 조건 변수 `cond`를 속성 `attr`(`NULL`이면 기본 속성)로 초기화함
  - 성공 시 `0`, 실패 시 오류 코드를 리턴함
- `int pthread_cond_destroy(pthread_cond_t *cond);`
  - 조건 변수 `cond`를 파괴하여 사용을 종료함
  - 성공 시 `0`, 실패 시 오류 코드를 리턴함
- `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`
  - 조건 변수 `cond`가 신호를 받을 때까지 스레드를 대기시키며, 대기 중에는 보호용 뮤텍스 `mutex`를 자동으로 잠금 해제하고 깨어나면 다시 잠금
  - 성공 시 `0`, 실패 시 오류 코드를 리턴함

동기화: 조건 변수

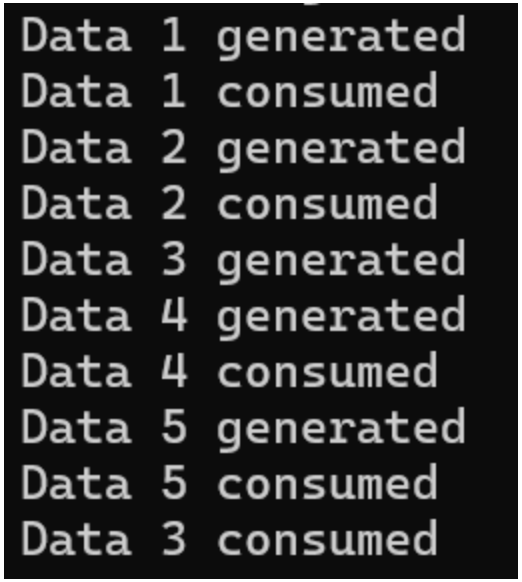
- `int pthread_cond_timewait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);`
  - 조건 변수 `cond`를 기다리되, 보호용 뮤텍스 `mutex`를 해제한 채 절대 시간 `abstime`까지 대기하고, 시간이 지나면 자동으로 반환됨
  - 성공 시 `0`, 시간 초과 시 `ETIMEDOUT`, 그 외 실패 시 오류 코드를 리턴함
- `int pthread_cond_signal(pthread_cond_t *cond);`
  - 조건 변수 `cond`에서 대기 중인 스레드 중 하나를 깨움
  - 성공 시 `0`, 실패 시 오류 코드를 리턴함
- `int pthread_cond_broadcast(pthread_cond_t *cond);`
  - 조건 변수 `cond`에서 대기 중인 모든 스레드를 깨움
  - 성공 시 `0`, 실패 시 오류 코드를 리턴함

동기화: 조건 변수


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

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int count = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_produce = PTHREAD_COND_INITIALIZER;
pthread_cond_t cond_consume = PTHREAD_COND_INITIALIZER;

void* producer(void* arg) {
   for (int i = 1; i <= BUFFER_SIZE; i++) {
      pthread_mutex_lock(&mutex);
      // 버퍼가 꽉 차 있으면 기다린다
      while (count == BUFFER_SIZE) {
         pthread_cond_wait(&cond_produce, &mutex);
      }
      buffer[count++] = i;
      printf("Data %d generated\n", i);
      // 소비자에게 신호
      pthread_cond_signal(&cond_consume);
      pthread_mutex_unlock(&mutex);
      sleep(1);
   }
   return NULL;
}
```

```c
void* consumer(void* arg) {
   for (int i = 1; i <= 5; i++) {
      pthread_mutex_lock(&mutex);
      // 버퍼가 비어 있으면 기다린다
      while (count == 0) {
         pthread_cond_wait(&cond_consume, &mutex);
      }
      int data = buffer[--count];
      printf("Data %d consumed\n", data);
      // 생산자에게 신호
      pthread_cond_signal(&cond_produce);
      pthread_mutex_unlock(&mutex);
      sleep(2);
   }
   return NULL;
}

int main() {
   pthread_t t1, t2;
   pthread_create(&t1, NULL, producer, NULL);
   pthread_create(&t2, NULL, consumer, NULL);
   pthread_join(t1, NULL);
   pthread_join(t2, NULL);
   pthread_mutex_destroy(&mutex);
   pthread_cond_destroy(&cond_produce);
   pthread_cond_destroy(&cond_consume);
   return 0;
}
```

동기화: 조건 변수


Data 1 generated
Data 1 consumed
Data 2 generated
Data 2 consumed
Data 3 generated
Data 4 generated
Data 4 consumed
Data 5 generated
Data 5 consumed
Data 3 consumed

예시: 다운로드 시뮬레이션

- 개요
  - 여러 개의 스레드가 동시에 파일 조각을 다운로드한다고 가정함
  - 각 스레드는 일정 시간이 걸리는 다운로드 작업을 수행하고, 완료하면 공유 변수 progress를 업데이트함
  - 메인 스레드는 모든 다로운드가 끝날 때까지 대기함
- 참고용 프로그램의 구조

```
int progress=0;
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond=PTHREAD_COND_INITIALIZER;

void *download(void *arg) {
   // 다운로드는 sleep() 함수로 시뮬레이션
   // mutex와 cond를 사용하여 progress 업데이트
}

int main() {
   // 다운로드 스레드 생성, mutex와 cond를 통해 다운로드 끝날 지 확인
}
```

어셈블리 레벨 레이스 컨디션

- 레이스 컨디션
  - 여러 개의 스레드가 동기화 없이 공유 자원을 동시에 접근할 때 발행함
  - 레이스 컨디션을 뮤텍스, 조건 변수 등을 사용하여 방지할 수 있음
  - 어셈블리 레벨에서 레이스 컨디션이 발생할 수 있음

```
unsigned int cnt=0;
void *count(void *arg) {
    int k;
    for (k=0; k<100000; k++)
        cnt++;
    return NULL;
}
int main() {
    pthread_t tids[4];
    int k;
    for (k=0; k<4; k++)
        pthread_create(&tids[k], NULL, count, NULL);
    for (k=0; k<4; k++)
        pthread_join(tids[k], NULL);
    printf("cnt=%u\n", cnt);
    return 0;
}
```

어셈블리 코드 상
- 레지스터에 cnt를 불러옴
- 레지스터 값 증가
- 레지스터 값을 cnt에 저장

동기화 방식을 사용하지 않으면 다른 스레드가 어셈블리 코드를 막아낼 수 있음

```
C:\Users\USER\Dropbox\KNUT_lectures\
cnt=243683

C:\Users\USER\Dropbox\KNUT_lectures\
cnt=129562

C:\Users\USER\Dropbox\KNUT_lectures\
cnt=145259

C:\Users\USER\Dropbox\KNUT_lectures\
cnt=132255
```

어셈블리 레벨 레이스 컨디션

- 레이스 컨디션

```
pthread_mutex_t mutex;
unsigned int cnt=0;
void *count(void *arg) {
    int k;
    for (k=0; k<100000; k++) {
        pthread_mutex_lock(&mutex);
        cnt++;
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
int main() {
    pthread_t tids[4];
    int k;
    pthread_mutex_init(&mutex, NULL);
    for (k=0; k<4; k++)
        pthread_create(&tids[k], NULL, count, NULL);
    for (k=0; k<4; k++)
        pthread_join(tids[k], NULL);
    pthread_mutex_destroy(&mutex);
    printf("cnt=%u\n", cnt);
    return 0;
}
```

```
C:\Users\USER\Dropbox\KNUT_lectures\
cnt=400000

C:\Users\USER\Dropbox\KNUT_lectures\
cnt=400000

C:\Users\USER\Dropbox\KNUT_lectures\
cnt=400000

C:\Users\USER\Dropbox\KNUT_lectures\
cnt=400000
```

세마포어(semaphore)

- 세마포어
  - 정수 값을 가지는 동기화 객체
  - 두 가지 핵심 연산
    - `wait()` 또는 `P()`: 세마포어 값 감소
      - 값이 `0`보다 크면 감소 후 진행
      - 값이 `0`이면 잠금
    - `signal()` 또는 `V()`: 세마포어 값 증가
      - 자원을 해제해서 다른 스레드가 접근할 수 있게 함
  - 초기값 = 허용 가능한 동시 전근 개수
    - 초기값 = `1` → 뮤텍스처럼 동작
    - 초기값 = `3` → 동시에 3개의 스레드가 접근 가능

세마포어(semaphore)

- 세마포어 사용의 일반적인 흐름
  - 세마포어 변수 선언

```
sem_t sem;
```

  - 초기화

```
sem_init(&sem, 0, 1);
// 0: 프로세서 내 스레드만 공유; 1: 초기값
```

  - 자원 접근 전/후 사용

```
sem_wait(&sem); // 자원 접근 전(P 연산, 잠금)
// 공유 자원 사용
sem_post(&sem); // 자원 접근 후(V 연산, 해제)
```

  - 사용 종료

```
sem_destroy(&sem);
```

세마포어(semaphore)

- `int sem_init(sem_t *sem, int pshared, unsigned int value);`
  - 세마포어 객체 `sem`을 초기화함
  - `pshared`가 `0`이면 해당 세마포어는 같은 프로세스 내의 스레드들 사이에서만 공유됨
  - `pshared`가 `0`이 아니면 프로세스 간 공유 가능
  - `value`는 세마포어 초기값으로, 동시에 접근 가능한 자원의 개수를 지정함
  - 성공 시 `0`, 실패 시 `-1`을 리턴함
- `int sem_destroy(sem_t *sem);`
  - 세마포어 객체 `sem`을 제거하고 관련된 시스템 자원을 반환함
  - 성공 시 `0`, 실패 시 `-1`을 리턴함
- `int sem_wait(sem_t *sem);`
  - 세마포어 객체 `sem`을 이용해 자원 접근을 시도함
    - 세마포어 0보다 크면, 세마포어 값을 `1` 감소시키고 자원 접근은 허용함
    - 값이 `0`이면, 다른 스레드나 프로세스가 `sem_post`로 증가시킬 때까지 대기됨
  - 성공 시 `0`, 실패 시 `-1`을 리턴함

세마포어(semaphore)

- `int sem_trywait(sem_t *sem);`
  - 세마포어 객체 `sem`을 이용해 자원 접근을 시도함
  - 세마포어 값이 `0`보다 크면, 값을 `1` 감소시키고 자원을 사용함
  - 값이 `0`이면 대기하지 않고 즉시 반환하며, 실패로 처리됨
  - 성공 시 `0`, 실패 시 `-1`을 리턴함
- `int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);`
  - 세마포어 객체 `sem`을 이용해 자원 접근을 시도함
  - 세마포어 값이 `0`보다 크면, 값을 `1` 감소시키고 자원을 사용함
  - 값이 0이면 `abs_timeout`까지 대기하며, 지정된 시간까지도 자원이 해제되지 않으면 실패
  - 성공 시 `0`, 실패 시 `-1`을 리턴함
- `int sem_post(sem_t *sem);`
  - 세마포어 객체 `sem`의 값을 `1` 증가시켜 자원 반환을 알림
  - 대기 중인 스레드/프로세스가 있으면 하나를 깨워서 진행하게 함
  - 성공 시 `0`, 실패 시 `-1`을 리턴함

세마포어(semaphore)


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

sem_t sem;

void* task(void* arg) {
    int id = *((int*)arg);
    printf("Thread %d: waiting...\n", id);
    sem_wait(&sem); // 세마포어 진입 (최대 2개까지 동시에 허용)
    printf("Thread %d: resource using\n", id);
    sleep(2); // 자원을 사용하는 것처럼 대기 (예: 파일 접근, 계산 등)
    printf("Thread %d: resource using completed\n", id);
    sem_post(&sem); // 세마포어 해제
    return NULL;
}

int main() {
    pthread_t threads[5];
    int ids[5];
    // 세마포어 초기값을 2로 설정, 동시에 2개 스레드만 접근 가능
    sem_init(&sem, 0, 2);
    for (int i = 0; i < 5; i++) {
        ids[i] = i + 1;
        pthread_create(&threads[i], NULL, task, &ids[i]);
    }
    for (int i = 0; i < 5; i++)
        pthread_join(threads[i], NULL);
    sem_destroy(&sem);
    return 0;
}
```

```
Thread 1: waiting...
Thread 1: resource using
Thread 3: waiting...
Thread 3: resource using
Thread 2: waiting...
Thread 4: waiting...
Thread 5: waiting...
Thread 1: resource using completed
Thread 3: resource using completed
Thread 2: resource using
Thread 4: resource using
Thread 2: resource using completed
Thread 4: resource using completed
Thread 5: resource using
Thread 5: resource using completed
```

멀티스레싱 관련 이슈

- 스레드 안전성(thread safety)
- 교착 상태(deadlock)
- 기아 상태(starvation)

멀티스레싱 관련 이슈

- 스레드 안전성(thread safety)
  - 여러 스레드가 동시에 같은 함수를 호출하거나 같은 자원을 사용할 때도, 프로그램이 정상적으로 동작하는 성질
  - 스레드 안전하지 않은 함수는 동시에 호출 시 레이스 컨디션, 잘못된 결과 발생 가능
  - 스레드 안전하게 만드는 방법
    - 뮤텍스/세마포어 같은 동기화 메커니즘 사용
    - 스레드마다 분리된 자원 사용
    - 원자적 연산(atomic operation) 사용

스레드 안전하지 않은 함수

```
char *unsafe_func() {
    static char buffer[100];
    sprintf(buffer, "hello");
    return buffer;
}
```

정적 변수를 사용 → 여러 스레드 동시 접근 시 문제 발생

스레드 안전한 함수

```
char *safe_func(char *buffer, size_t size) {
    snprintf(buffer, size, "hello");
    return buffer;
}
```

멀티스레싱 관련 이슈

- 스레드 안전성(thread safety)
  - 스레드 안전성을 위해 재진입 함수(**reentrant function**) 사용을 권장
  - 재진입 함수
    - 중간에 실행이 끊기더라도, 다른 실행 흐름에서 동일한 함수를 안전하게 호출할 수 있는 함수
    - 특징
      - 전역 변수나 정적 변수 사용 안 함
      - 공유 자원 사용 안 함
      - 입력 파라미터 또는 지역 변수만 사용

    ✶ **모든 재진입 함수는 스레드 안전함**

    ✶ **모든 스레드 안전 함수가 반드시 재진입 가능한 것이 아님**

멀티스레싱 관련 이슈

- 교착 상태(deadlock)
  - 두 개 이상의 스레드/프로세스가 서로 가진 자원을 기다리면서 **무한 대기 상태**에 빠지는 문제
  - 다음과 같은 4 가지 조건을 만족할 때 발생
    - 상호 배제: 자원은 한 번에 하나의 프로세스만 사용
    - 점유와 대기: 자원을 가진 상태에서 다른 자원 요청 가능
    - 비선점: 자원을 강제로 빼앗을 수 없음
    - 순환 대기: 프로세스 간에 원형 형태로 대기
  - 해결 방법
    - 자원 획득 순서 규칙: 항상 같은 순서로 잠금을 획득
    - 타임아웃 설정: 일정 시간 대기 후 실패 처리
    - 교착 상태 감지 후 회복: wait-for graph 알고리즘 사용, 교착 검사 후 해제

멀티스레싱 관련 이슈

- 교착 상태(deadlock)

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

pthread_mutex_t lock1, lock2;

void* thread_func1(void* arg) {
    pthread_mutex_lock(&lock1);
    printf("Thread 1: lock1 acquired\n");
    sleep(1); // Deadlock 발생 가능성 증가
    printf("Thread 1: lock2 try to acquire...\n");
    pthread_mutex_lock(&lock2);
    printf("Thread 1: lock2 acquired\n");
    pthread_mutex_unlock(&lock2);
    pthread_mutex_unlock(&lock1);
    return NULL;
}
```

```
Thread 1: lock1 acquired
Thread 2: lock2 acquired
Thread 1: lock2 try to acquire...
Thread 2: lock1 try to acquire...
```

```c
void* thread_func2(void* arg) {
    pthread_mutex_lock(&lock2);
    printf("Thread 2: lock2 acquired\n");
    sleep(1); // Deadlock 발생 가능성 증가
    printf("Thread 2: lock1 try to acquire...\n");
    pthread_mutex_lock(&lock1);
    printf("Thread 2: lock1 acquired\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, thread_func1, NULL);
    pthread_create(&t2, NULL, thread_func2, NULL);
    pthread_join(t1, NULL); pthread_join(t2, NULL);
    pthread_mutex_destroy(&lock1);
    pthread_mutex_destroy(&lock2);
    return 0;
}
```

멀티스레싱 관련 이슈

- 교착 상태(deadlock)
  - 뮤텍스 획득 순서를 통일시키면 교착 상태를 해결할 수 있음

```c
// 획득 순서: lock1 -> lock2
void* thread_func1(void* arg) {
    pthread_mutex_lock(&lock1);
    printf("Thread 1: lock1 acquired\n");
    pthread_mutex_lock(&lock2);
    printf("Thread 1: lock2 acquired\n");
    pthread_mutex_unlock(&lock2);
    pthread_mutex_unlock(&lock1);
    return NULL;
}
```

```c
// 획득 순서: lock1 -> lock2
void* thread_func2(void* arg) {
    pthread_mutex_lock(&lock1);
    printf("Thread 2: lock1 acquired\n");
    pthread_mutex_lock(&lock2);
    printf("Thread 2: lock2 acquired\n");
    pthread_mutex_unlock(&lock2);
    pthread_mutex_unlock(&lock1);
    return NULL;
}
```

```
Thread 1: lock1 acquired
Thread 1: lock2 acquired
Thread 2: lock1 acquired
Thread 2: lock2 acquired
```

멀티스레싱 관련 이슈

- 기아 상태(starvation)
  - 특정 스레드/프로세스 자원을 요청했는데, 다른 스레드를 때문에 계속 우선순위에서 밀려나 실행 기회를 못 얻는 문제
  - 교착 상태와 달리 프로그램은 계속 실행되지만, 일부 스레드는 영원히 기다림
  - 해결 방법
    - 공정한 스케줄링 정책 적용
    - 우선순위가 낮은 스레드도 일정 시간이 지나면 우선순위 상승
    - 세마포어나 뮤텍스에서 fair lock 사용

멀티스레싱 관련 이슈

- 기아 상태(starvation)

```c
pthread_mutex_t lock;
int shared_resource = 0;

void* greedy_thread(void* arg) {
    while (1) {
        pthread_mutex_lock(&lock);
        // 자원을 오래 붙잡고 거의 놓지 않음
        shared_resource++;
        printf("Greedy Thread: resource = %d\n", shared_resource);
        usleep(1000); // 아주 짧게만 쉼
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}

void* starving_thread(void* arg) {
    while (1) {
        pthread_mutex_lock(&lock);
        // 실행 기회를 거의 못 얻는 스레드
        shared_resource++;
        printf("Starving Thread: resource = %d\n",  shared_resource);
        sleep(1); // 실제 자원을 사용하려고 일정 시간 점유
        pthread_mutex_unlock(&lock);
        sleep(1); // 다시 시도
    }
    return NULL;
}
```

```c
int main() {
    pthread_t t1, t2;
    pthread_mutex_init(&lock, NULL);
    pthread_create(&t1, NULL, greedy_thread, NULL);
    pthread_create(&t2, NULL, starving_thread, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_mutex_destroy(&lock);
    return 0;
}
```

```
Greedy Thread: resource = 1
Greedy Thread: resource = 2
Greedy Thread: resource = 3
Greedy Thread: resource = 4
Greedy Thread: resource = 5
Starving Thread: resource = 6
Greedy Thread: resource = 7
Greedy Thread: resource = 8
Greedy Thread: resource = 9
Greedy Thread: resource = 10
Greedy Thread: resource = 11
Greedy Thread: resource = 12
Greedy Thread: resource = 13
```

멀티스레싱 관련 이슈

- 기아 상태(starvation)
  - Fair lock 사용하는 해결 방법

```
#include <stdatomic.h> // C11 원자적 연산

typedef struct {
   atomic_int ticket;
   atomic_int turn;
} fair_lock_t;

void fair_lock_init(fair_lock_t *lock) {
   atomic_init(&lock->ticket, 0);
   atomic_init(&lock->turn, 0);
}

void fair_lock_acquire(fair_lock_t *lock) {
   int my_ticket = atomic_fetch_add(&lock->ticket, 1);
   while (atomic_load(&lock->turn) != my_ticket) {
      // busy-wait (다른 방법: usleep 넣어서 CPU 낭비 줄일 수 있음)
      sched_yield();
   }
}

void fair_lock_release(fair_lock_t *lock) {
   atomic_fetch_add(&lock->turn, 1);
}
```

멀티스레싱 관련 이슈

- 기아 상태(starvation)
  - Fair lock 사용하는 해결 방법

```
fair_lock_t lock;
int shared_resource = 0;

void* greedy_thread(void* arg) {
    while (1) {
        fair_lock_acquire(&lock);
        shared_resource++;
        printf("Greedy Thread: resource = %d\n", shared_resource);
        usleep(1000); // 작업 후 잠시 대기
        fair_lock_release(&lock);
        usleep(500); // CPU 양보
    }
    return NULL;
}

void* starving_thread(void* arg) {
    while (1) {
        fair_lock_acquire(&lock);
        shared_resource++;
        printf("Starving Thread: resource = %d\n", shared_resource);
        usleep(2000); // 자원을 조금 오래 사용
        fair_lock_release(&lock);
        usleep(1000); // 다시 시도
    }
    return NULL;
}
```

```
Greedy Thread: resource = 1
Starving Thread: resource = 2
Greedy Thread: resource = 3
Starving Thread: resource = 4
Greedy Thread: resource = 5
Starving Thread: resource = 6
Greedy Thread: resource = 7
Starving Thread: resource = 8
Greedy Thread: resource = 9
Starving Thread: resource = 10
Greedy Thread: resource = 11
Starving Thread: resource = 12
Greedy Thread: resource = 13
Starving Thread: resource = 14
Greedy Thread: resource = 15
Greedy Thread: resource = 16
Starving Thread: resource = 17
Greedy Thread: resource = 18
Greedy Thread: resource = 19
Starving Thread: resource = 20
Greedy Thread: resource = 21
Starving Thread: resource = 22
```

소켓(socket) 및 비동기식 I/O

- 소켓
  - 네트워크 통신을 위한 엔드포인트(endpoint)
  - 파일 디스크립터(file descriptor)로 표현되며, 다른 `read()`/`write()` 호출과 동일한 방식으로 I/O가 가능함
  - C 언어에서 `<sys/socket.h>` 헤더에 소켓 관련 함수와 자료구조가 제공됨
- `int socket(int domain, int type, int protocol);`
  - 지정한 주소 체계 `domain`, 소켓 종류 `type`, 프로토콜 번호 `protocol`을 사용하여 새로운 소켓을 생성함
  - 성공 시 소켓 디스크립터, 실패 시 `-1`을 리턴함
- `int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`
  - 소켓 디스크립터 `sockfd`를 지정된 주소 구조체 `addr`에 연결하며, 주소의 길이는 `addrlen`로 지정함
  - 성공 시 `0`, 실패 시 `-1`을 리턴함

소켓(socket) 및 비동기식 I/O

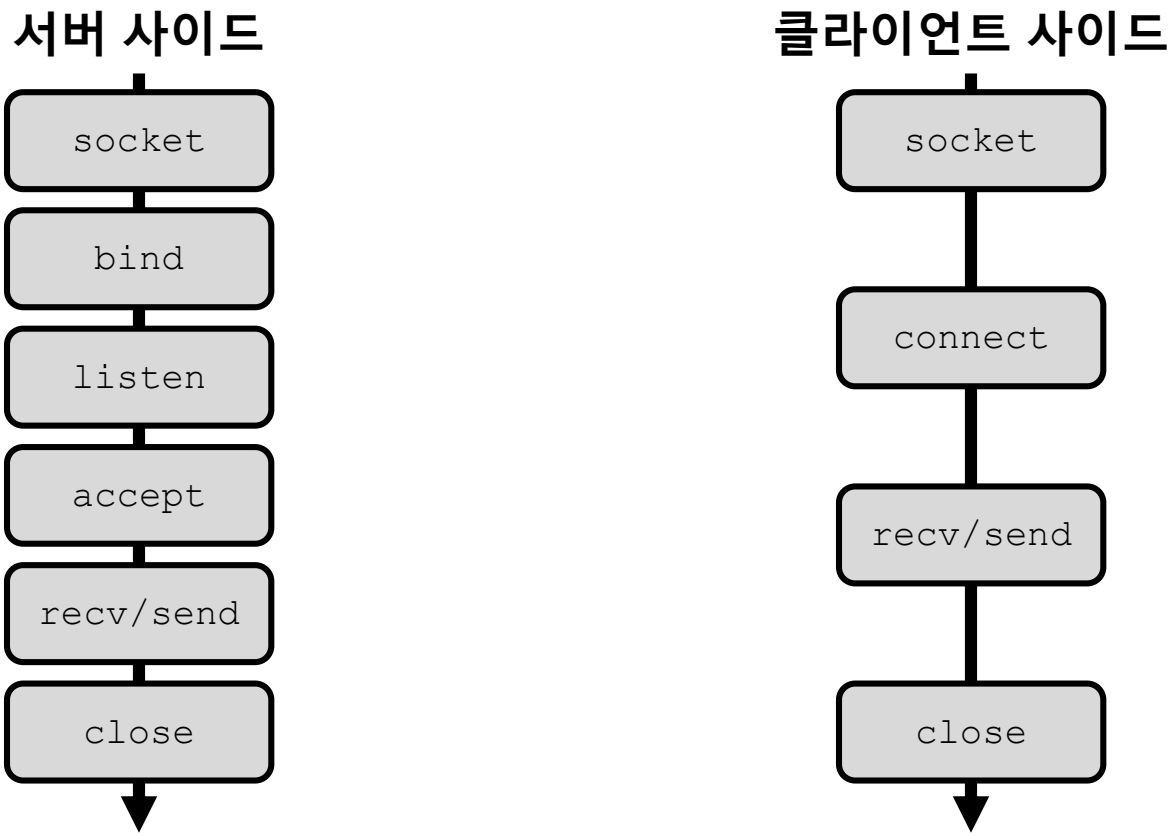
- `int listen(int sockfd, int backlog);`
  - 지정한 소켓 `sockfd`를 연결 요청 대기 상태로 전환하고, 최대 대기 큐(queue) 크기를 `backlog`로 설정함
  - 성공 시 `0`, 실패 시 `-1`을 리턴함
- `int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);`
  - 연결 요청을 기다리던 소켓 `sockfd`에서 새로운 클라이언트 연결을 수락하며, 클라이언트 주소 정보를 `addr`에 저장하고 그 크기를 `addrlen`에 기록함
  - 성공 시 새로운 소켓 디스크립터, 실패 시 `-1`을 리턴함
- `int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`
  - 클라이언트 소켓 `sockfd`를 서버 주소 `addr`과 주소 길이 `addrlen`을 이용하여 서버에 연결함
  - 성공 시 `0`, 실패 시 `-1`을 리턴함

소켓(socket) 및 비동기식 I/O

- `ssize_t send(int sockfd, const void *buf, size_t len, int flags);`
  - 소켓 `sockfd`를 통해 버퍼 `buf`에 저장된 데이터를 길이 `len`만큼 플래그 `flags` 옵션을 적용하여 전송함
  - 성공 시 전송한 바이트 수, 실패 시 `-1`을 리턴함
- `ssize_t recv(int sockfd, void *buf, size_t len, int flags);`
  - 소켓 `sockfd`에서 데이터를 수신하여 버퍼 `buf`에 최대 `len` 바이트까지 저장하며, 수신 방식 옵션을 flags로 지정함
  - 성공 시 수신한 바이트 수, 실패 시 `-1`을 리턴함
- `int close(int sockfd);`
  - 소켓 디스크립터 `sockfd`를 닫아 연결을 종료하고 관련된 자원을 해제함
  - 성공 시 `0`, 실패 시 `-1`을 리턴함

소켓(socket) 및 비동기식 I/O

- 일반적인 소켓 사용 흐름



소켓(socket) 및 비동기식 I/O

- 비동기식 I/O
  - 프로그램이 I/O를 기다리며 멈추지 않고, 다른 작업을 계속 수행할 수 있는 방식
  - CPU 시간을 낭비하지 않고 효율적 처리 가능
- I/O 멀티플렉싱
  - 여러 I/O 파일 디스크립터(FD)를 동시에 감시하여, 어느 FD가 읽기/쓰기 가능 상태가 되었는지 알려주는 기법
  - 대표적으로 서버에서 많은 클라이언트 연결을 동시에 처리할 때 활용
  - 관련 함수
    - `select()`
    - `poll()`
    - `epoll()`

소켓(socket) 및 비동기식 I/O

- `int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);`
  - 파일 디스크립터 `0`부터 `nfds-1`까지에서, 읽기 감시 집합 `readfds`, 쓰기 감시 집합 `writefds`, 예외 감시 집합 `exceptfds` 중 준비된 FD가 있는지 검사하고, 검사 최대 시간은 타임아웃 구조체 `timeout`으로 지정함
  - 성공 시 준비된 FD 수, 실패 시 `-1`을 리턴함
- `int poll(struct pollfd *fds, nfds_t nfds, int timeout);`
  - `fds` 배열에 들어있는 `nfds` 개의 FD에 대해 읽기/쓰기/에러 이벤트를 검사하며, 타임아웃 시간은 `timeout`으로 지정함
  - 성공 시 발생한 FD 수, 타임아웃 시 `0`, 실패 시 `-1`을 리턴함

소켓(socket) 및 비동기식 I/O

- `int epoll_create(int flags);`
  - 커널에 새로운 epoll 인스턴스를 생성하고, 생성된 epoll 객체를 참조할 FD를 반환하며, 추가 옵션을 `flags`로 지정함
  - 성공 시 epoll FD, 실패 시 `-1`을 리턴함
- `int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);`
  - epoll 인스턴스 `epfd`에 대해, 대상 파일 디스크립터 `fd`를 `op` 동작으로 제어하며 이벤트 조건은 `event`로 지정함
  - 성공 시 `0`, 실패 시 `-1`을 리턴함
- `int epoll_wait(int epfd, struct epoll_event *event, int maxevents, int timeout);`
  - epoll 인스턴스 `epfd`에서 등록된 FD들의 이벤트를 검사하고, 준비된 이벤트를 최대 `maxevents`개만큼 `events` 배열에 채워넣으며, 대기시간은 `timeout`로 지정함
  - 성공 시 준비된 이벤트 개수, 타임아웃 시 `0`, 실패 시 `-1`을 리턴함