R E P O R T

2. 파일처리, 프로세스 및 프로세스간 통신

과목명	오픈소스 프로젝트
담당교수	김성우 교수님
학과	컴퓨터소프트웨어공학과
학번	20212979
이름	임진호

동의대학교
DONG-EUI UNIVERSITY

목 차

Ⅰ. 실습에 필요한 준비사항 (이론) ····· 3~36

1. Rust 요약 정리 ····· 3~13
2. 파일 처리 함수 정리 ····· 14~16
3. 입출력 장치 파일 ····· 16~19
4. 프로세스 및 시그널 ····· 19~26
5. 리눅스의 프로세스 모델과 시그널/쓰레드와의 관계 ····· 26~27
6. 프로세스 간 통신 ····· 27~34
7. 리눅스에서 DBUS 통신 기법 ····· 34~36

Ⅱ. 실습 결과 ····· 37~74

1. github 저장소에 업로드 ····· 37
2. 두 행렬을 더하는 프로그램 (Rust) ····· 38~40
3. 전화번호부 및 검색 프로그램 ····· 41
4. 파일 및 디렉토리와 관련된 함수 ····· 42~46
5. “ls -R” 명령 ····· 47~48
6. 타자 연습 프로그램 ····· 49~51
7. 프로세스 및 시그널과 관련된 함수 ····· 54~57
8. 프로세스 간 통신 함수 ····· 58~62
9. 텍스트 기반의 채팅 프로그램(파이프) ····· 63~65
10. 텍스트 기반의 채팅 프로그램(메시지큐) ····· 66~68
11. 한 파일을 다른 파일로 복사하는 프로그램 ····· 68~69
11. 알림 메시지를 전달하는 서버-클라이언트 프로그램(D-BUS) ····· 70~74

Ⅱ. 검토 ····· 74

1. Rust 에 대해서 조사하고 요약하여 보고서에 정리하여 본다.

Rust
-모질라 리서치에서 개발한 범용 프로그래밍 언어 -인터넷 서버 및 클라이언트 프로그램 개발 목표로 설계 -안전성과 병행 프로그래밍, 메모리 관리의 직접 제어에 초점 -객체 지향 프로그래밍 스타일을 모두 지원한다
Rust의 특징
-메모리 안전성: 소유권(Ownership) 시스템을 통한 메모리 오류 (예: 매달린 포인터) 컴파일 시점 방지. -성능: C++와 유사한 수준의 고성능을 목표로 함 -동시성: 데이터 경합(race) 없는 안전한 병렬 프로그래밍 지원.

Rust의 안정성
러스트의 안정성은 여러 메모리 안전성 문제를 해결하는 데 중점을 두고 있다. 여기에는 매달린 포인터, 데이터 경합, 버퍼 오버플로, 반복자 무효화와 같은 문제들이 포함된다.
1. 매달린포인터(dangling pointer) -> 이미 해제된 메모리를 가리키고 있는 포인터를 의미한다. 이러한 포인터는 프로그램이 해당 메모리에 접근하려 할 때 예기치 않은 동작을 초래할 수 있다. 러스트는 소유권 시스템을 통해 이 문제를 방지한다. 소유자가 범위를 벗어나면 메모리가 자동으로 해제되며, 이때 해당 포인터는 더 이상 유효하지 않게 된다. 러스트의 컴파일러는 매달린 포인터가 발생하지 않도록 강력한 검사를 수행하여 안정성을 높인다.
2. 데이터경합(race) -> 두 개 이상의 포인터가 동시에 동일한 데이터에 접근할 때 발생한다. 특히, 그 중 하나라도 데이터를 수정하는 경우, 결과는 예측할 수 없게 된다. 러스트는 대여(borrowing) 시스템을 통해 이 문제를 해결한다. 불변 참조는 여러 개 허용되지만, 가변 참조는 하나만 허용되므로, 동시에 데이터를 수정하는 상황을 방지하고, 이를 통해 데이터 경합을 예방한다.
3. 버퍼 오버플로 (Buffer Overflow) -> 배열이나 버퍼의 경계를 넘어서는 데이터를 쓰는 경우 발생한다. 이는 메모리 손상이나 프로그램 충돌을 초래할 수 있기 때문에, 러스트는 안전한 배열 접근을 제공하며, 인덱스가 유효한지 컴파일 타임 및 런타임을 검사한다. 이를 통해 버퍼 오버플로 문제를 예방.
4. 반복자 무효화 (Iterator Invalidations) -> 컬렉션이 변경될 때 반복자가 더 이상 유효하지 않게 되는 상황을 의미한다. 예를 들어, 컬렉션에서 요소를 추가하거나 제거할 경우, 기존의 반복자는 그 상태를 반영하지 않게 된다. 러스트에서는 반복자가 무효화되지 않도록 설계된 컬렉션을 사용하여 이 문제를 해결한다. 또한, 반복자를 안전하게 사용할 수 있도록 컴파일 타임 검사를 통해 안정성을 보장

주요 러스트 관리 프로그램	
파일	설명
rustup	러스트 인스톨로
cargo	러스트 패키지 관리 프로그램
rustc	러스트 컴파일러
rustdoc	러스트 문서 제작

Rustup

- rustup은 러스트의 공식 인스톨러(installer)이자 툴체인(toolchain) 관리 도구입니다. 이 프로그램을 통해 개발자는 여러 버전의 러스트 컴파일러(안정형, 베타형 등)를 손쉽게 설치 및 관리할 수 있으며, 개발 환경에 필요한 패키지를 갱신하는 역할을 담당합니다

Cargo

- Cargo는 러스트의 공식 패키지 관리자(Package Manager)이자 빌드 시스템입니다. 프로젝트의 생성, 빌드, 테스트 실행, 문서화 등 개발에 필요한 수많은 기능을 자동화합니다. 특히 외부 라이브러리(Crate)의 의존성을 관리하고 프로젝트를 실행 가능한 바이너리로 만드는 핵심적인 역할을 수행합니다.

Rustc

- rustc는 LLVM 컴파일러를 기반으로 하는 실제 러스트 컴파일러입니다. 개발자가 작성한 Rust 소스 코드를 기계어로 번역하여 실행 파일을 생성하며, 이 과정에서 Rust의 강력한 안전성 검사 및 코드 최적화를 수행합니다.

Rustdoc

- rustdoc은 러스트의 공식 문서 제작 도구입니다. 소스 코드 내의 주석을 자동으로 분석하여 일관된 형식의 API 문서를 생성해 줍니다. 이는 프로젝트의 유지보수성과 다른 개발자들의 코드 이해를 돕는 데 중요한 역할을 합니다.

Rust 설치

- curl-sSfhttps://sh.rustup.rs|sh 명령어를 입력 후 엔터키를 누르면 설치 완료

Cargo 패키지 관리자의 주요 명령

명령	설명	명령	설명
init	프로젝트생성	add	Manifest 파일에 의존성 추가
build	프로젝트 빌드	remove	Manifest 파일에 의존성 삭제
run	프로젝트 실행	update	Cargo.lock의 의존성 갱신
doc	문서화	search	크레이트 레지스트리 검색
test	테스트 실행	install	러스트실행 파일을 설치

함수를 사용한 프로그램 코드 사례

```
fn add(a: i32, b: i32)-> i32 {
 a + b
 }
 fn main() {
 let a = 1;
 let b: i32 = 2;
 let c = add(am b);
 println!("1 + 2 = {}", c);
 }
```

함수 형식

-fn: 새로운 함수를 정의하는 키워드

-인자는 콤마(,)로 구분하고 각 인자 자료형은 콜론(:) 뒤에 명시

-반환값 자료형은 -> 뒤에 표시하고 반환문은 세미콜론(;) 생략

-함수내용은 { } 로 기술

```
fn add(a: i32, b: i32) -> i32 ;
```

함수 이름 변수자료형 변수자료형 반환값 자료형

1. 자료형
러스트에서 모든 값들은 어떤 자료형을(type)을 가짐
단일 자료형 -정수형 i8, i16, i32, i64, i128: 부호 있는 정수형 -부동소수점 실수형 => f64: 64비트 부동 소수점 -bool 형 => bool: 참(true) 또는 거짓(false) 값을 가질 수 있는 자료형 -문자 => char: 하나의 유니코드 스칼라 값을 나타내며, 4바이트 크기
복합 자료형 -> 복합 자료형은 여러 값을 하나로 묶을 수 있는 자료형입니다. -튜플 -> 여러 다른 자료형을 포함할 수 있는 고정 크기의 자료형. -배열 -> 동일한 자료형의 고정 크기 리스트
사용자 정의 자료형 -구조체 => 서로 다른 자료형의 값들을 이름(필드)으로 묶어 관리하는 자료형 -열거형 => 여러 종류의 값들 중 하나가 될 수 있음을 표현하는 자료형. -트레이트 객체 => 특정 트레이트(공유 동작)를 구현한 객체를 동적으로 가리키는 참조자 -함수와 클로저 => 실행 가능한 코드를 묶어 정의하고 호출할 수 있는 자료형

2. 제어문
if 문은 가장 기본적인 조건 분기문이다. - 주요 기능: 주어진 조건(condition)의 true/false 여부에 따라 다른 코드 블록을 실행 - 다중 조건: else if를 사용하여 여러 조건을 연쇄적으로 검사할 수 있다 - 표현식으로 사용: if 문은 표현식(expression)이므로, let 구문에서 변수에 값을 할당하는 데 사용할 수 있다.
loop문 (무한반복) - loop 키워드는 명시적으로 중단(break)되기 전까지 코드를 무한히 반복하는 루프를 생성한다.
while 문 (조건부 반복) - while 문은 특정 조건이 참(true)인 동안에만 루프를 반복해서 실행합니다. 조건이 false가 되면 루프가 종료된다.
for 문 (컬렉션 반복) -for 문은 배열, 벡터 등 컬렉션의 모든 요소를 순차적으로 반복(iterate)할 때 사용됩니다. 예를 들어 a.iter()는 배열의 각 요소를 순회하는 반복자를 반환하여 for 문에서 사용할 수 있게 한다.

3. 소유권
-> 러스트의 핵심 개념중 하나 원칙 - 각각의 값은 소유 객체 또는 변수인 소유자를 가짐 - 한 번에 하나의 소유자만 존재 - 소유자가 파괴될 때 값도 해제(dropped) - 값을 변수에 대입하거나 함수로 전달하거나 반환하면 이동할 수 있다

예시

{
letp=Box::new((1,2));
lets=format!("{:?}",p);
…
} 여기서부터 더 이상 p,s가 유효하지 않는다.

소유권 이동

주의 사항
let s = String.from("hello");
let t = s;
println!("{}",s);
-> t에 s를 이동시키고 s는 더 이상 유효하지 않는다.

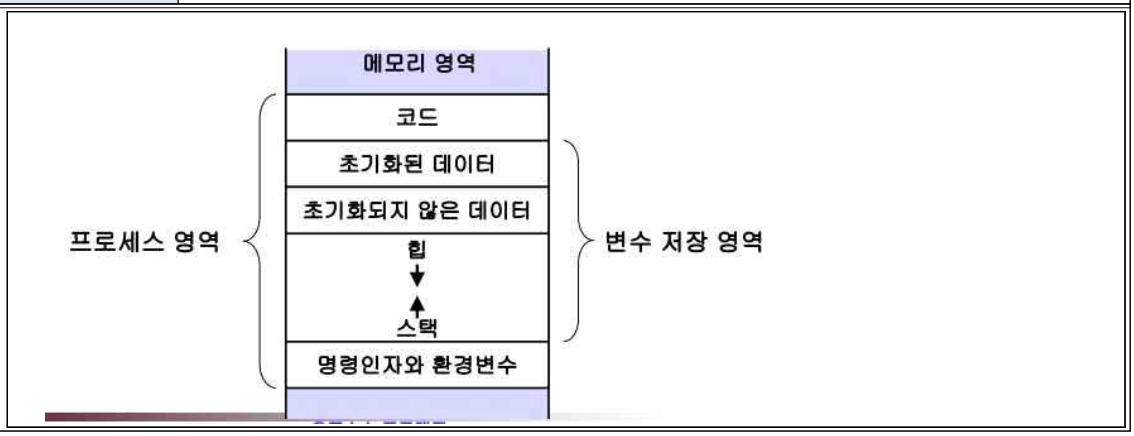
소유권의 경직성을 해소하는 방법
- 이동 활용 - 참조자를 통해 빌림 - 참조 카운트를 활용한 스마트 포인터 -Rc,Arc등 공유 소유권 객체

4. 공유 소유권
- 참조 카운트(reference count)를 사용하여 값의 수명 관리 - Arc는 스레드 안전성을 가져 스레드 간 공유 가능

use std::rc::Rc:
let s: Rc<String> = Rc::new(String::from("hello"));
let t: Rc<String> = s.clone();
let u: Rc<String> = s.clone();
-> Rc가 소유한 값은 변경할 수 없음 → 따라서 순환 구조를 만들 수 없다.

5. 변수 저장 공간

공간	기능
데이터	정적(static)변수및상수,컴파일시에크기가결정
힙(heap)	동적할당공간,할당및해제알고리즘이알고리즘이필요하며느림
스택(stack)	지역변수할당공간,push및pop연산으로할당해제가빠름



포인터 형 -> 메모리 주소를 표현하는 자료형

종류	기능
참조자(reference)	소유권을갖지않는러스트의기본포인터형. 절대로null이 될 수 없으며 소유권과 수명 추적&를 사용하여 지정(&T 형과&mut T 형)
원시(raw)포인터	메모리주소값을가지며,기존C/C++의포인터와동일 * 를 사용하여 지정(*const T 형과*mut T 형)안전성이 떨어져 unsafe 블록 내에서 사용
스마트포인터	메모리주소외에부가적인정보포함 Box<T>, Rc<T>, RefCell<T> 동적 모듈에 반대되는 독 립적 수행 모델 생성

6. 참조자 (레퍼런스, reference)

- 소유권을 가지지 않는 러스트의 포인터 형
- 다른 변수의 값을 가리키는 포인터 이다.

공유 참조자 - &T형이며 대상을 읽을 수 있지만 수정 불가능

```
let x = 1;
let r = &x;
 //&x 는 공유 참조자
Assert!(*r == 10); // r을 명시적으로 역참조
```

가변참조자 - &mutT (레프뮤트T)형이며 대상을 읽거나 수정 할 수 있다. 특정한 범위에서 가변 참조자는 하나만 만들 수 있다.

```
let mut y = 2;
let m = &mut y; //&mut y 는 가변 참조자
*m += 1; // y 값 참조를위해m을명시적으로역참조
assert!(*m == 3); // 수정된 값 확인
```

참조자 안전성

-변수의 수명은 참조자의 수명을 포함해야 한다.
-참조자를 통해 빌린다.

```
 {
 let r;              //-----'b
 {                   //    |
 letx=1;             //---+-'a
 r=&x;               //     ||
 }                   // -- +|
 assert_eq!(*r,1); // ---+
 }
```

-> x가 수명을 다하여 오류가 발생

참조자 수명
- 모든 참조자는 수명을 가지며 암묵적으로 추론된다 - 따옴표를 활용하여 수명을 명시

정적 변수와 수명
-정적 변수는 unsafe 블록으로 사용 ex) 코드

```
staticmutSTREF:&i32;
 fnf(p:&i32){STREF=p;}
```

-> 컴파일 되지 않음

```
 staticmutSTREF:&i32=&100;
 fnf(p:&'statici32){
 unsafe{
 STREF=p;}}
```

-> 다음고 같이 정적 수명 매개변수 static을 사용

7. 오류 처리
- 복구 가능한 오류와 복구 불가능한 오류로 구분 - 복구 불가능한 오류는 panic! - 복구 가능한 오류는 Result<T,E> 활용
복구 가능한 오류 -> 일반적으로 발생할 수 있는 오류로, 프로그램이 이를 처리하고 계속 실행할 수 있다. 예를 들어, 파일을 열 때 파일이 존재하지 않는 경우
복구 불가능한 오류 -> 프로그램의 실행을 중단해야 하는 심각한 오류 예를 들어, 배열의 인덱스 범위를 넘어가는 경우

복구 불가능한 오류 (패닉) -> 프로그램 자체의 버그로 인해 발생하는 문제
예) 배열 범위 밖의 요소를 참조하거나 0으로 나누는 등

```
n main() {
 let v = vec![1, 2, 3];
 v[4];
 }
```

```
fn main() {
 panic!("panic error!");
 }
```

-> 명시적으로 panic!() 매크로 함수를 호출하는 것도 가능.

복구 불가능한 오류 (백트레이스) -> 복구불가능한 오류에 대해 백트레이스 획득가능
-> RUST_BACKTRACE환경변수를 1로 설정하고 실행
예) 0으로 나눔 오류

```
fnmain(){
 letdiv=|a,b|a/b;
 println!("err={}",div(1,0));
 }
```

```
$cargorun
 thread'main'panickedatsrc/main.rs:6:21:
 attempt to divide by zero
 note: run with `RUST_BACKTRACE=1`
 environment variable to display a backtrace
```

-> 환경변수를 1로 설정

복구 가능한 오류 Result<T,E> 열거형으로오류처리
-> Result 형은 실패 가능성을 암시함. 즉, 성공하면 Ok 아니면 Err 반환
-> 수많은 시스템 함수들이 Result 형으로 반환됨

```
usestd::fs::File;
 fnmain(){
 let f=File::open("hello.txt");
 let f=matchf{
 Ok(file)=>file,
 Err(error)=>{
 panic!("Therewasaproblemopeningthefile:{:?}",error)
 }, }; }
```

Result 형 메소드 (자주 사용하는 메소드 목록)
-> unwrap() 과 expect()는 자주 사용됨

함수	설명
is_ok(), is_err()	Result 결과가 성공인지 오류인지 bool 반환
ok()	성공값이 있을 경우 Option<T>로 반환
err()	오류값이 있을 경우 Option<E>로 반환
unwrap_or(fallback)	셩공 결과일 경우 성공값을 반환하고, 그렇지 않으면 fallback을 반환
unwrap_or_else(fallbackfn)	위와 동일하지만 값이 아니라 함수나 클로저를 받음
unwrap()	성공 결과일 경우 성공값을 반환하고, 오류일 경우 패닉에 빠짐
expect(message)	unwrap()과 동일하지만 패닉에 빠질 때 메시지를 지정
as_ref()	Result<T,E> 형을 Result<&T,&E>로 변환
as_mut()	Result<T,E> 형을 Result<&mut T,&mut E> 로 변환

unwrap() 사례

```
usestd::fs::File;
 fnmain(){
 let f=File::open("hello.txt").unwrap();
 }
```

expect() 사례

```
 usestd::fs::File;
 fn main(){
 let f=File::open("hello.txt").expect
("Failedtoopenhello.txt"); }
```

8. 구조체

- 서로 다른 자료형들을 하나의 자료형으로 묶음
- 튜플과 비슷하지만 각 요소들의 이름으로 접근 가능하고 c/c++의 구조체, 파이썬/자바의 클래스와 비슷하다.

형식	코드

struct구조체이름{ 요소이름: 자료형,
(바디코드)
}
	structRectangle{
	center:(i32, i32),
	size:(u32,u32)
	}

멤버 접근
- 구조체 멤버는 비공개가 기본이며 선언된 모듈 및 하위 모듈에서 접근 가능하다. - pub를 붙이면 모듈 외부에 공개된다. ex) pub center: (i32, i32)
메소드
-함수와 비슷하지만 구조체 내에서 정의되고 호출됨 - 메소드의 첫째 인자는 self 이며 호출하는 구조체의 인스턴스 - impl 블록으로 정의하며 함수들의 집합체

```
impl Rectangle {
 fn area(&self)-> u32 {
 self.size[1] * self.size[2]
 }
 }
```

```
fn main() {
let rect1 = Rectangle { (0, 0), (10, 20) }
println!("the area of rect1 is {:?}", rect1.area());}
```

-> p1.method(&p2); 와(&p1).method(&p2); 는 동일

8. 열거형	
- 여러 자료형 값들을 나열 하여 하나의 자료형으로 표현 - c의 공용체와 비슷하지만 훨씬 유연하고 안전성을 가진다.	
형식	코드

enum열거형이름{요소이름또는요소이름(자료형),
(바디코드)
}
	enum Shape{
	Point,
	Line,
	Triangle,
	Rectangle,
	Circle
	}

-> 열거형 식별자 및 :: 에 의해 열거형값 (variants 또는생성자) 구분	

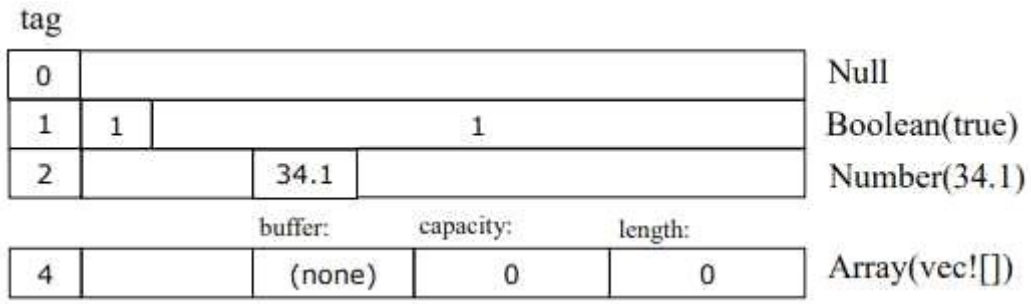
```
let r = Shape::Rectangle;
 Let c = Shape::Circle;
 fn area(sh_type: Shape) { … }
```

데이터를 가지는 열거형

-> 열거형 베리언트로 튜플, 구조체 등 어떤 값도 가질 수 있다.

```
enum Json {
 Null,
 Boolean(bool),
 Number(f64),
 String(String),
 Array(Vec<Json>),
 Object(Box<HashMap<String, Json>>)
```

메모리 구조



제네릭 열거형 -> 대표적으로 많이 쓰는 Option<T> 와 Result<T, E> 형

```
enumOption<T>{
Some(T),
None, }
letsome_number=Some(5);
letsome_string=Some("astring");
letabsent_number:Option<i32>=None;
```

```
enumResult<T,E>{
 Ok(T),
 Err(E),
 }
```

Match 연산자

- 일련의 패턴에 대해 어떤 값을 비교한 뒤 적절한 코드 수행
- 열거형, 튜플, 배열, 슬라이스, 구조체, 참조자 패턴 가능

```
enumCoin{
 Penny,
 Nickel,
 Dime,
 Quarter, }
 fnvalue_in_cents(coin:Coin)->u32{
 matchcoin{
 Coin::Penny=>{
 println!("Luckypenny!");
 1 },
 Coin::Nickel=>5,
 Coin::Dime=>10,
 Coin::Quarter=>25,} }
```

```
fnplus_one(x:Option<i32>)->Option<i32>{
matchx{
None=>None,
Some(i)=>Some(i+1),
}
}
let five=Some(5);
letsix=plus_one(five);
letnone=plus_one(None);;
```

다형성

-같은 코드로 여러 가지 자료형의 값을 처리할 수 있는 프로그래밍 기법

러스트의 다형성 기법

1. 제네릭을 통해 여러 자료형들에 대해 추상화
2.
3. 트레이트를 통해 공유하는 여러 동작들에 대해 제네릭 방식으로 정의

트레이트	
- 인터페이스나 추상클래스의 역할을 한다. - 여러 자료형의 공유 동작을 정의	
형식	코드

trait 트레이트이름{
fn 함수이름(&self, 인자)-> 반환형;
…
}
	trait Shape {
	fn area(&self)-> f64;
	}

형식	구현

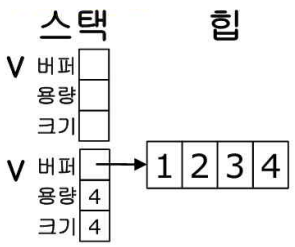
impl 트레이트이름for 자료형{
함수구현부
… }
	struct Rectangle {
	width: f64,
	height: f64 }
	impl Shape for Rectangle {
	fn area(&self)-> f64 {
	self.width * self.height } }

10. 컬렉션 -> 가변 길이의 객체들을 처리하는 다양한 자료구조 모음
러스트 자원 자료형

러스트 유형	설명	C++	Java	Python
Vec<T>	크기 확장 가능 배열	vector	ArrayList	List
Vecdeque<T>	양방향 큐	deque	ArrayDeque	
LinkedList<T>	양방향 연결 리스트	list	LinkedList	
BinaryHeap<T>	최대 힙	priority_queue	PriorityQueue	
HashMap<K,V>	키-값 해시 테이블	unordered_map	HashMap	Dict
BTreeMap<K,V>	키-값 정렬 테이블	map	TreeMap	
HashSet<T>	순서없는 해시 집합	unordered_set	HashSet	set
BTreeSet<T>	정렬된 집합	set	TreeSet	

Vec<T> 자료형
크기 조절이 가능한 T형 배열이며 힙에 할당 - 각 데이터 원소들이 연속 공간에 할당되며, 임의 요소 접근이 빠르다 - 임의 위치에서 원소 삽입 및 삭제가 가능 하지만 처리 속도 느림
vec! 매크로를 사용하여 생성
3가지 값으로 구성된 팻 포인터 - 힙에 할당해 소유하는 버퍼 포인터 - 버퍼 저장 요소의 개수인 용량(capacity) - 현재 실제로 가진 요소 개수 길이(length)
메모리 확보 및 재할당 기능 - 용량보다 크게 삽입이 일어나면 메모리 재할당(reallocation)이 일어나며, 이러한 재할당은 vector를 사용할 때 성능 저하의 요인이 됨

빈 벡터 생성
let mutv : Vec<i32>=Vec::new(); let mutv = Vec::with_capacity(8);
초기 값으로 생성
let v=vec![1,2,3,4,5];//매크로를사용하여생성



요소 추가

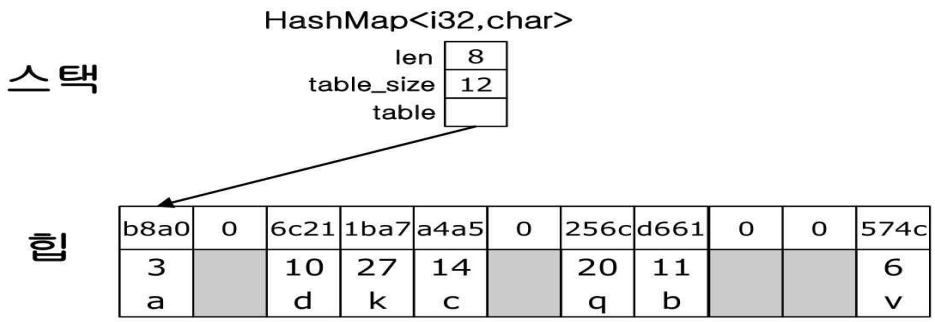
let mutv=Vec::new();
v.push(1);//요소추가
v.push(2);

임의의 위치에서 삽입

let mutv = vec![1, 2, 3];
v.insert(1, 4);
v.insert(0, 5);

마지막에서 삭제
let mutv = vec![1, 2, 3, 4]; v.pop();
임의의 위치에서 삭제 remove
v.remove(1);
전체 삭제 clear
v.clear();

Map
- map은 키(key)와 데이터(data)를 연결하여 키를 이용해서 데이터를 빠르게 찾을 수 있는컬렉터 - 키는 유일(unique)해야 하며, 숫자, 문자열등 순서가 있는 자료형이면 모두 키가 될 수 있음



-

2. 파일 처리 함수에 대하여 조사하고 이해한다.

파일 처리 함수의 종류

- 저수준 파일 처리 함수 UNIX/LINUX 에서사용하는기본방법
-> 정수파일 디스크립터(descriptor) 번호 사용
- ANSI C 표준 입출력 함수
-> 정수형 파일 기술자 대신 FILE구조를 이용

기능	함수원형
파일열기/닫기	FILE*fopen(constchar*path,constchar*mode); intfclose(FILE*stream);

파일읽기/쓰기	intfgetc(FILE*stream);
	intfputc(intc,FILE*stream);
	char*fgets(char*s,intsize,FILE*stream);
	int fputs(const char *s, FILE *stream);
	intfscanf(FILE*stream,constchar*format,…);
	intfprintf(FILE*stream,constchar*format,…);

파일위치재배치	intfseek(FILE*stream,longoffset,intwhence);
	long ftell(FILE *stream);
	voidrewind(FILE*stream);

저수준 파일 처리 함수

-UNIX 파일 접근 프리미티브
->프로그램 안에서 파일들을 다루기 위해 UNIX가 제공하는 기본적인 프리미티브들
->UNIX 커널에 의해 제공되는 I/O 장치에 직접 접근을 제공하는 시스템 호출 들의 작은 집합

이름	의 미
open	읽거나 쓰기 위해 파일을 열거나, 또는 빈 파일을 생성
creat	빈 파일을 생성한다
close	열려진 파일을 닫는다
read	파이로부터 정보를 추출한다
write	파일에 정보를 기록한다
lseek	파일 안의 지정된 바이트로 이동한다.
unlink	파일을 제거한다.
remove	파일을 제거하는 다른 방법
fcntl	한 파일에 연관된 속성을 제어한다.

open 시스템 호출 -> 기존의 파일을 읽거나 쓰기 전에 항상 파일 개방

사용법

```
#include <sys/types.h>
 #include <sys/stat.h>
 #include <fcntl.h>
 int open(const char *pathname, int flags,
[mode_t mode]);
```

pathname : 개방될 파일의 경로

Flags : 접근 방식 지정

Mode : 보안과 연관, 생략가능

close 시스템 호출 -> open의 역, 개방중인 파일을 닫음

사용법

```
#include <unistd.h>
 int close(int filedes);
```

➢filedes : 닫혀질 파일기술자

파일 생성시 접근 권한 지정

사용자 권한		그룹 권한		기타 사용자 권한	
S_IRWXU	읽기, 쓰기, 실행 가능	S_IRWXG	읽기, 쓰기, 실행 가능	S_IRWXO	읽기, 쓰기, 실행 가능
S_IRUSR	읽기 가능	S_IRGRP	읽기 가능	S_IROTH	읽기 가능
S_IWUSR	쓰기 가능	S_IWGRP	쓰기 가능	S_IWOTH	쓰기 가능
S_IXUSR	실행 가능	S_IXGRP	실행 가능	S_IXOTH	실행 가능

creat 시스템 호출 -> 파일을 생성하는 대안적 방법

사용법

```
#include <sys/types.h>
 #include <sys/stat.h>
 #include <fcntl.h>
 int creat(const char *pathname, mode_t mode);
```

pathname : 개방 될 파일의 경로

Mode : 필요한 접근 허가 제시, 생략가능

read 시스템 호출 ->

파일로 부터 임의의 문자들 또는 바이트들을 호출 프로그램의 제어하에 있는 버퍼로 복사

```
#include <unistd.h>
 ssize_t read(int filedes, void *buffer, size_t n)
```

filedes : 이전의 open 또는 creat로 부터 얻은 파일기술자

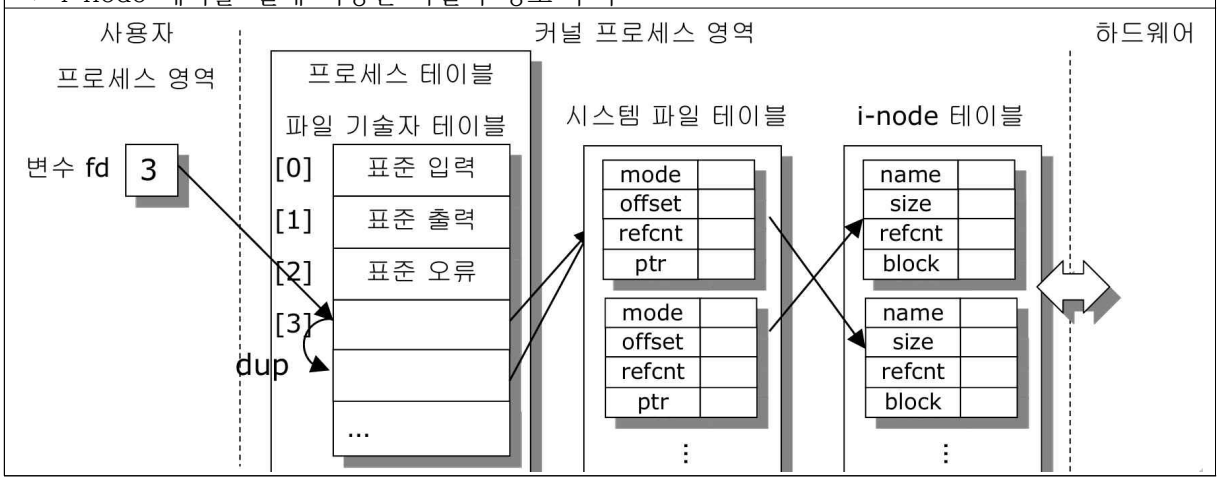
buffer : 자료가 복사 되어 질문자 배열의 포인터

n : 파일로부터 읽혀질 바이트의 수
write 시스템 호출 -> 문자 배열인 프로그램 버퍼로부터 외부 파일로 자료를 복사
사용법 #include <unistd.h> ssize_t write(int filedes, const void *buffer, size_t n); filedes : 이전의 open 또는 creat로 부터 얻은 파일기술자 buffer : 자료가 복사 되어 질문자 배열의 포인터 n : 파일로 쓰여질 바이트의 수

lseek 와 임의 접근
- 읽기 쓰기 포인터의 위치 (다음에 읽거나 쓸 바이트 위치) 변경 - 파일에 대한 임의 접근가능
사용법 #include <sys/types.h> #include <unistd.h> off_t lseek(int filedes, off_t offset, int start_flag); filedes : 개방 되어 있는 파일의 파일기술자 offset : 읽기-쓰기 포인터의 새 위치 결정 start_flag : 읽기-쓰기 포인터가 파일의 어느지점을 시작으로 할지 결정

3. 입출력 장치 파일에 대하여 조사하고 요약하여 보고서에 정리하여 본다.

파일의 표현
시스템의 파일 관리 -파일 기술자 테이블, 시스템 파일 테이블, i-node 테이블을 통해 각 프로세스가 사용하는 파일관리 -> 시스템 파일 테이블-시스템의 모든 파일에 대한정보(상태, 현재 오프셋등) 수록 -> i-node 테이블-실제 저장된 파일의 정보 수록



파일 기술자 복사 함수
-파일 기술자를 새로운 항목으로 복사 -> dup 함수는 소스 파일 기술자를 사용하지 않는 가장 작은 파일 기술자로 복사 -> dup2 함수는 소스 파일 기술자를 대상 파일 기술자로 복사 - 복사되기 전 후의 기술자들은 같은 시스템 파일 항목 공유 -> 락, 파일위치포인터, 플래그공유

사용법

```
#include <unistd.h>
 int dup (int filedes);
 int dup2 (int filedes, int filedes2);
```

filedes : 존재하는 소스 파일 기술자 항목
filedes2: 복사할 대상 파일 기술자 항목
반환되는 값은 새로운 파일 기술자 번호, 복사가 실패하면 -1 반환

파일 상태
- 파일 유형 등의 상태 정보 처리

사용법

```
#include <sys/types.h>
 #include <sys/stat.h>
 #include <unistd.h>
 int stat(const char *filename, struct stat *buf);
 int fstat(int filedes, struct stat *buf);
 int lstat(const char *filename, struct stat *buf);
```

stat() : filename이 주어지면 해당 파일에 대한 정보 획득
fstat() : 파일 기술자를 인자로 파일 정보 획득
lstat() : stat()와 비슷하나 해당 파일이 심볼릭 링크인 경우 링크의 정보 획득
buf : 파일 정보를 정의한 stat 구조체 포인터

파일 유형을 테스트하는 매크로들 /<sys/stat.h> 에 정의

매크로	의 미
S_ISREG()	Regular file (일반적인 파일 형태)
S_ISDIR()	Directory file (다른 파일을 가지는 파일 형태)
S_ISCHR()	Character special file (문자 장치에 사용)
S_ISBLK()	Block special file (블록 장치에 사용)
S_ISFIFO()	FIFO pipe (프로세스간 IPC 에 사용)
S_ISLNK()	Symbolic link file (다른 파일에 대한 링크 파일 형태)
S_ISSOCK()	socket file (네트워크 통신에 사용)

access 시스템 호출

-실제 사용자 ID와 실제 그룹ID에 기반해서 파일에 대한 접근성 검사

사용법

```
#include <unistd.h>
 int access(const char *pathname, int mode);
```

pathname : 접근성 검사할 파일에 대한 경로

mode : 검사할 파일의 모드, 다음 매크로 들을 bitwise OR

umask 시스템 호출

-파일 모드 생성 매스크를 설정하고 이전 값을 반환

사용법

```
#include <sys/types.h>
 #include <sys/stat.h>
 mode_t umask(mode_t mask);
```

mask : 파일모드생성매스크

디렉토리관련함수

-디렉토리 관련 동작수행

사용법

```
#include <sys/types.h>
 #include <dirent.h>
 DIR *opendir(const char *name);
 int closedir(DIR *dir);
 struct dirent *readdir(DIR *dir);
 void rewinddir(DIR *dir);
```

getcwd 함수

- 작업 디렉토리에 대한 정보 읽기

사용법

```
#include <unistd.h>
 char *getcwd(char *buf, size_t size);
```

buf : 현재 작업 디렉토리의 절대 경로를 복사할 버퍼 포인터

size : 경로 이름이 초과할 경우NULL이 반환

파일링크관련함수

- 파일에 대한 링크와 관련된 동작 수행

사용법

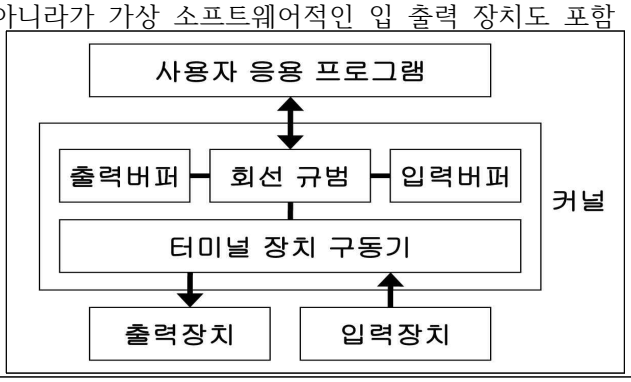
```
#include <unistd.h>
 int link(const char *oldpath, const char *newpath);
 int unlink(const char *pathname);
 int symlink(const char *oldpath, const char *newpath);
```

link() : oldpath 파일에 대한 newpath 하드링크를 생성
unlink() : 파일 시스템에서 pathname 파일삭제. 파일이 심볼릭 링크이면 링크를 삭제
symlink() : oldpath 파일에 대한 newpath 심볼릭 링크를 생성

터미널 처리

- 사용자가 데이터를 입출력하는 장치
- 고전적으로는 키보드와 CRT 터미널과 같은 컴퓨터 하드웨어
- 리눅스에서는 텍스트 입출력장치 뿐만 아니라가 가상 소프트웨어적인 입 출력 장치도 포함
- 회선규범(line discipline) 제어 기능

1. 특수 문자 및 키 설정,
2. 에코 기능
3. 버퍼링 기능
4. 회선 속도 결정

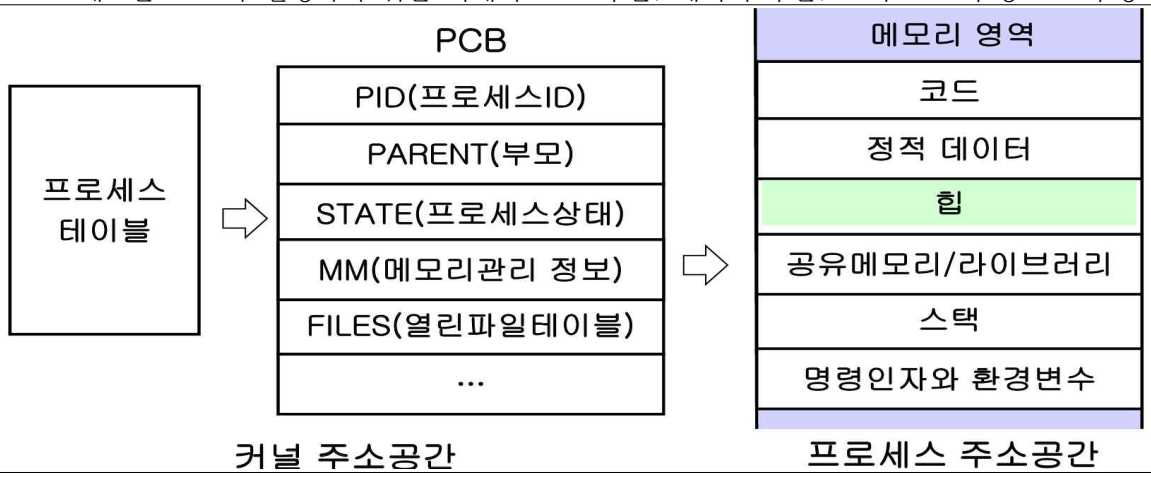


4. 교재에서 프로세스 및 시그널에 대한 내용을 읽고 이해한다.

프로세스 제어

- 프로그램 : 디스크 상에 저장되어 실행 가능한파일
- 프로세스 : 보통 디스크로부터 주메모리에 적재되어, 실행중인 프로그램을 의미

-> 쉘은 하나의 명령을 수행하기 위해 어떤 프로그램을 시작 할 때 마다 새로운 프로세스를 생성
-> 프로세스는 CPU가 실행하기 위한 기계어 코드 부분, 데이터 부분, 그리고 스택 등으로 구성

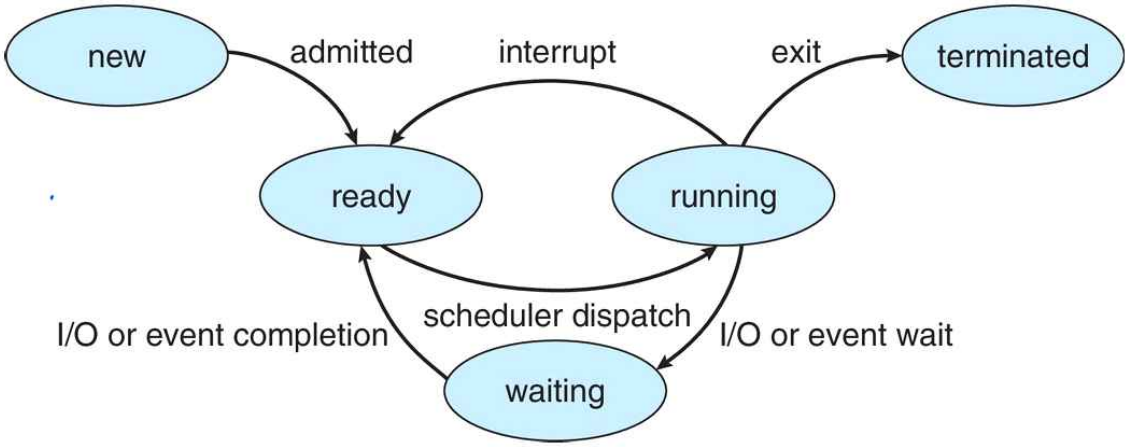


프로세스 생명 주기

- 운영체제는 여러개의 프로세스를 생성하고 스케줄링 하면서 번갈아 실행시키는 멀티프로그래밍 방식적용

프로세스 상태

상태	의 미
생성	프로그램이 실행되어 프로세스가 생성
준비	커널이 프로세스를 실행하기 위해 준비
실행	프로세스가 커널로부터 CPU를 할당받아 실행 중
대기	입출력 작업을 처리하기 위해 대기
종료	프로세스를 종료



프로세스 생성

- 시스템이 부팅되고 난 후에 커널이 커널 스레드/프로세스들과 최초의 사용자 프로세스를 생성

-> 각 프로세스는 자식 프로세스를 생성 할 수 있다

- UNIX 프로세스환경은 디렉토리 트리처럼 계층적인 구조

-> 0번 ID를 가진 프로세스부터 생성

-> pstree 명령으로 확인 가능

프로세스 시스템 호출들

이름	의미
fork	호출프로세스와똑같은새로운프로세스를생성
exec	한프로세스의기억공간을새로운프로그램으로대체
wait	프로세스동기화제공.연관된다른프로세스가끝날때까지기다림
exit	프로세스를종료

fork 시스템 호출

- 기본 프로세스 생성 함수
- 성공적으로 수행되면 호출 프로세스(부모프로세스)와 똑같은 새로운 프로세스(자식프로세스) 생성

사용법

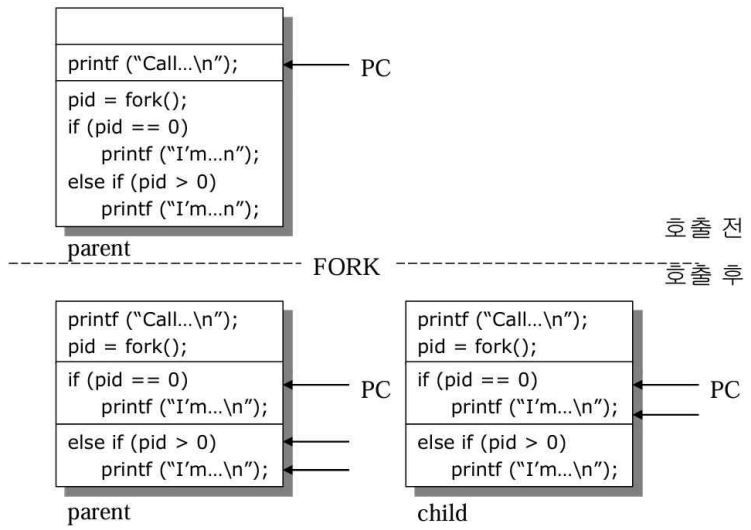
```
#include <sys/types.h>
 #include <unistd.h>
 pid_t fork(void);
```

반환값(return value)

->정상실행

- 실행(부모) 프로세스는: 생성(자식) 프로세스의 프로세스 ID반환
- 생성(자식) 프로세스는: 0을 반환

-> 이상 실행: 음수 값을 반환



-> fork() 호출 후에는 부모·자식 프로세스 모두에서 그 다음 명령문이 계속 실행됨

-> pid 값으로 부모/자식을 구분하여 각 역할을 달리할 수 있음

-> 이 구조로 멀티 프로세싱이나 병렬 실행 등이 구현됨

실행결과

```
Calling fork
I'm the child process
I'm the parent process
```

프로세스 종료

exit 시스템 호출 -> 프로세스 종료

사용법

```
#include <stdlib.h>
 void exit (int status);
```

status : 종료 상태를 나타냄

반환값: 정상적 종료: 0, 그렇지 않을 경우: 0 이 아닌 값

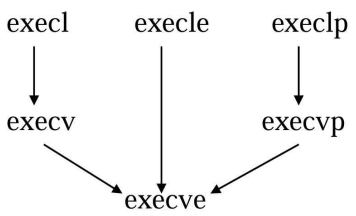
-> 모든 개방된 파일 기술자들을 닫는다

프로세스 동기화
- wait 시스템 호출 -> 자식 프로세스들 중 하나가 수행을 마치기를 기다린다
사용법 #include <sys/types.h> #include <sys/wait.h> pid_t wait(int *status); status : wait가 복귀 될 때 유용한 상태 정보(퇴장상태) 때로 부모 프로세스가 fork로 자식을 만든 직후에 호출
waitpid 함수는 특정 자식 프로세스가 끝나기를 기다릴 때 사용한다.

좀비 프로세스 (너무 이른퇴장)
- 부모 프로세스가 wait를 수행하지 않고 있는 상태에서 자식이 퇴장 할때 -> 퇴장 프로세스는 좀비 프로세스가 된다. -> 부모 프로세스가 wait를 수행할 때 삭제 된다. - 하나 이상의 자식 프로세스가 수행 중인 상태에서 부모가 퇴장할때 -> 자식 프로세스는 init 프로세스(리눅스최초수행프로세스)에게 맡겨진다. ※ 부모가 wait하면서 자식의 자원을 수거해주지 않는다면 자원 낭비가 생긴다.

프로그램 수행 -> exec 계열 함수
- 새로운 프로그램의 수행 시작

```
#include<unistd.h>
/*execl호출계열은NULL끝나는리스트가인수로주어져야한다*/
/*execl은수행가능파일에대한유효한경로이름이주어져야한다*/
intexecl(constchar *path,const char*arg0, ...,constchar*argn,(char
*)0);
intexecle(constchar*path,constchar*arg0,...,constchar*argn, (char *)0,
char *const envp[]);
/*execlp은단지수행가능파일의파일이름을필요로한다*/
intexeclp(constchar*file,constchar*arg0,...,constchar*argn, (char *) 0);
/* execv 호출 계열은 반드시 인수의 배열이 주어져야 한다*/int
execv(const char *path, char *const argv[]);
intexecve(constchar*path,char*constargv[],char*constenvp[]);
/* execvp는 단지 수행가능 파일의 파일 이름을 필요로 한다*/
int execvp(const char *file, char *const argv[]);
```



반환값

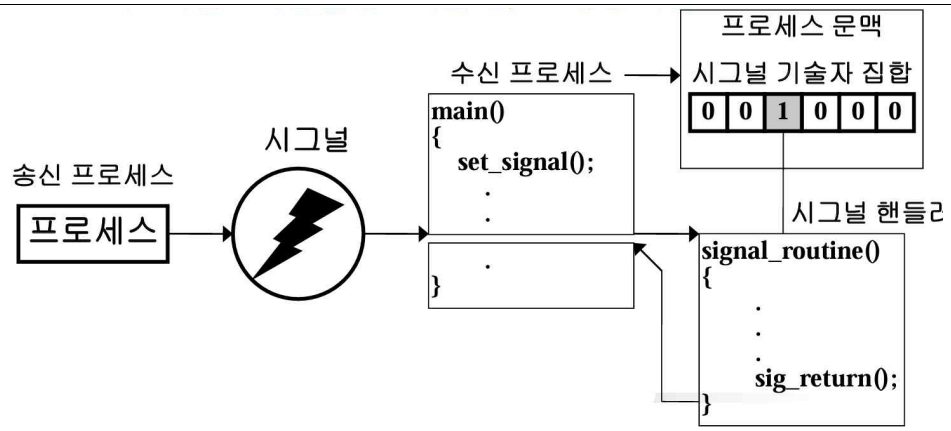
1. 정상적으로 수행한 경우 : 실행 프로세스로 복귀하지 않는다.(fork와는다름)
2. 오류 : 음수 값으로 반환

프로세스 속성 -> 프로세스 식별 번호
- 각 프로세스는 고유의 식별 번호를 가진다 -> 한 시점에서 유일한 프로세스 번호 -> 프로세스0 : 스케줄러, 즉스와퍼(swapper) 프로세스 -> 프로세스1 : init 초기화 프로세스

시그널
- 다른 프로세스에게 이벤트 발생을 알리는 소프트웨어 인터럽트 -> 시그널이 발생하면 수신 프로세스는 현재 문맥을 저장하고 시그널 핸들러 루틴을 실행하고 다시 복귀 - 시그널 발생 예 -> 프로그램 실행시 ctrl + c를 눌려 강제종료 시킬 때 -> 백그라운드 작업을 종료 시킬 때, kill 명령 사용

시그널 동작

1. 기본적으로 설정한 동작을 수행한다.
2. 시그널을 무시하고 프로그램의 수행을 계속한다.
3. 미리 정해진 일을 수행한다.



시그널 종류

이름	의 미
SIGHUP	hangup. 단말기 연결이 끊어졌을 때 그 단말기에 연결된 모든 프로세스에 보냄. 이것을 받으면 종료.
SIGINT	interrupt. 사용자가 인터럽트키를 칠 때 단말기와 연결된 모든 프로세스에 보냄. 수행중인 프로그램을 중지시키는 일반적인 방법
SIGQUIT	quit. SIGINT 와 마찬가지로 사용자가 단말기에서 종료(quit)키를 칠 때 보냄. 종료키의 일반적인 값은 ASCII FS 또는 CTRL-\
SIGILL	illegal Instruction. 비정상적인 명령 수행 시 보냄
SIGTRAP	trace trap. ptrace 시스템과 함께 sdb 또는 adb 등의 디버거에 의해 사용되는 특별한 시그널.
SIGABRT	abort. 현재 프로세스가 abort 함수를 호출할 때 보냄. 비정상적인 종료(abnormal termination) 가 됨. 이것을 받으면 코어 덤프하고 종료.
SIGFPE	floating-point exception. 오버플로우나 언더풋로우 같은 부동 소숫점 오류가 발생했을 때 보냄
SIGKILL	kill. 프로세스로부터 다른 프로세스를 종료시키기 위해 보냄.
SIGUSR1 SIGUSR2	SIGTERM 과 마찬가지로 이것들은 커널에 의해 사용되는 것이 아니라 사용자가 원하는 목적을 위하여 사용 가능.
SIGSEGV	segmentation violation. 프로세스가 유효하지 않은 메모리 주소에 접근할 때 보냄. 이것을 받으면 비정상적 종료.

이름	의 미
SIGPIPE	write on a pipe with no-one to read it. 종료한 파이프나 소켓에 쓸 때 보냄. (파이프는 또다른 프로세스간 통신 방식)
SIGALRM	alarm clock. 타이머가 만료되었을 때 프로세스에 보냄. alarm 함수로 이루어짐.
SIGTERM	software termination. 프로세스를 종료시키기 위해 사용자에 의해 사용.
SIGSTKFLT	stack fault. 스택 오류일 때 보냄.
SIGCHLD	child status has changed. 자식 프로세스가 종료하거나 중단될 때 부모 프로세스에 보냄. 이 시그널을 받으면 무시.
SIGCONT	continue. 이 시그널을 받으면 중단된 프로세스의 경우 계속 실행하고, 실행중일 때는 무시.
SIGSTOP	stop, unblockable. 프로세스를 중단시키기 위해 보냄.
SIGTSTP	keyboard stop. 사용자가 일시 중지 키 (Ctrl-Z)을 칠 때 보냄. SIGSTOP와 비슷
SIGTTIN	백그라운드 프로세스가 단말기로부터 읽기를 시도할 때 보냄.
SIGTTOU	백그라운드 프로세스가 단말기로 쓰기를 시도할 때 보냄.
SIGURG	프로세스에게 네트워크 연결에 긴급하거나 대역을 벗어난 자료가 수신되었음을 알림.
SIGSYS	bad arguments to a system call. 이것은 잘못된 시스템 호출을 보냈을 때 보냄.

시그널 집합 - 시그널을 원소로 하는 집합
- 관련 시스템 호출 -> 시그널 집합 관련 기능 수행 -> 각각의 시그널을 따로 처리하는 것보다 편리

사용법
 	
#include <signal.h>
/* 초기화 */
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
/* 조작 */
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
 	
sigemptyset - 비어있는 시그널 집합 생성
sigfillset - 모든 시그널을 포함 하는 집합 생성
sigaddset - 특정 시그널번호를 집합에 추가
sigdelset - 특정 시그널번호를 집합에서 제거

시그널 처리 ->
- sigaction 시스템호출 -> 프로세스가 특정 시그널에 대해 다음 세가지 행동중 하나를 지정 1. 프로세스는 종료하고 코어 덤프 2. 시그널을 무시 3. 시그널 핸들러에 지정된 함수를 실행
사용법 #include <signal.h> int sigaction(int signo, const struct sigaction *act, struct sigaction *oact)' signo: signal number act : sigaction 구조체 oact: 이전 설정값

시그널 전송
-다른 프로세스에게 특정 시그널 전송
사용법 #include <sys/types.h> #include <signal.h> int kill(pid_t pid, int sig); sig : 전송한 시그널 pid : 시그널 sig를 받을 프로세스ID -> 양수: 해당 프로세스 식별번호를 가진 프로세스 -> 음수 : 음수: 프로세스 그룹 식별번호가 pid의 절대값과 같은 모든 프로세스에게 전송

pause 시스템호출 -> 시그널이 도착할 때까지 프로세스 실행을 중단하고 대기

사용법

```
#include <unistd.h>
 int pause(void);
```

raise 시스템호출 -> 현재 프로세스에게 시그널 전송

사용법

```
#include <unistd.h>
 int raise(int sig);
```

시그널 차단

sigprocmask 시스템호출 -> 특정 시그널을 차단하도록 허용

사용법

```
#include <signal.h>
 int sigprocmask(int how, const sigset_t *set, sigset_t *oset;
```

how: sigprocmask가 어떻게 동작할 것인지를 결정

set: 선택할 시그널들의 집합

oset: 이전에 설정된 시그널들의 마스크값

시그널로부터 이전 상태 복귀

sigsetjmp 와 siglongjmp 시스템호출

-sigsetjmp 함수는 현재 프로그램 상태를 저장

-siglongjmp 함수는 저장된 위치로 복귀

사용법

```
#include <setjmp.h>
 int sigsetjmp(sigjmp_buf env, int savesigs);
 void siglongjmp(sigjmp_buf env, int val);
```

sigsetjmp 함수는 현재의 스택과 시그널 마스크 저장

-> siglongjmp가 호출되어 저장한 위치로 복귀 할 때 이들을 복구

sigsetjmp 함수의 반환값

-> 직접 반환 되면 0을 반환

-> Siglongjmp에 의해 반환 되면 0 이아닌값(val) 반환

데몬 프로세스 -> 빽그라운드로 돌면서 여러 작업을 하는 프로그램

1. 프로세스를 백그라운드로 만든다 (fork 호출 및 부모프로세스종료)
2. 제어 터미널이 없는 세션과 그룹 리더로 설정(setsid() 호출)
- 제어 터미널이 있으면 터미널 문자등으로 죽을 수 있음
3. 현재 작업 디렉토리를 루트 디렉토리(“/”)로 설정
4. umask(0)로 변경하여 파일 접근 권한 미리 설정
5. 열고 있는 모든 파일들을 닫는다
- 표준입력, 표준출력, 표준오류는/dev/null 로설정
6. 오류 메시지를 남기기위해 로그 파일설정

daemon 함수 활용

```
#include <unistd.h>
 int daemon(int nochdir, int noclose);
```

NOCHDIR가 0이면, 현재 디렉토리를/로바꿈
NOCLOSE가 0이면, 표준입력, 표준출력, 표준 오류 파일을 닫음

5. 리눅스의 프로세스 모델과 시그널/쓰레드와의 관계에 대하여 조사하고 요약하여 보고서에 정리하여 본다.

리눅스 운영체제는 동시에 많은 작업을 수행하기 위해 프로세스와 스레드를 활용하며, 이들 사이의 비동기적 이벤트 전달을 위해 시그널(Signal)을 사용한다.

기본 개념 소개

1. 프로세스(Process)
-프로세스는 실행 중인 프로그램으로, 독립된 메모리 공간을 갖는다(코드, 데이터, 힙, 스택).
운영체제는 각 프로세스를 PCB(Process Control Block)로 관리한다.

2. 스레드(Thread)
- 스레드는 프로세스 내부의 실행 흐름 단위로, 프로세스의 메모리 공간(code/data/heap)을 공유하지만 자신만의 스택과 레지스터 문맥을 가진다.

3. 시그널(Signal)
- 시그널은 비동기 이벤트를 프로세스(또는 스레드)에 전달하기 위한 소프트웨어 인터럽트이다.

프로세스와 스레드의 관계

- 하나의 프로세스는 여러 개의 스레드를 가질 수 있으며, 이 스레드들은 같은 주소 공간(code, data, heap)과 파일 디스크립터를 공유한다. 이로 인해 메모리 사용 효율은 높아지지만,
공유 자원을 동시에 접근할 때 경쟁 상태(race condition) 가 발생할 수 있어 뮤텍스, 세마포어 등의 동기화 기법이 필요하다. 서로 다른 프로세스는 완전히 독립된 메모리 공간을 갖기 때문에 자원 충돌은 발생하지 않지만, 대신 프로세스 간 통신(IPC)이 필요하다.

프로세스와 시그널의 관계

- 시그널은 기본적으로 프로세스 단위로 전달되는 비동기 알림이다. 시그널이 도착하면 해당 프로세스 전체가 시그널을 받은 것으로 간주되며, 프로세스는 등록된 시그널 핸들러를 실행하거나 기본 동작(종료·중단·무시)을 수행한다.
즉, 시그널은 프로세스의 흐름을 변경하거나 특정 동작을 강제로 수행하도록 하는 운영체제의 제어 수단으로 사용된다.

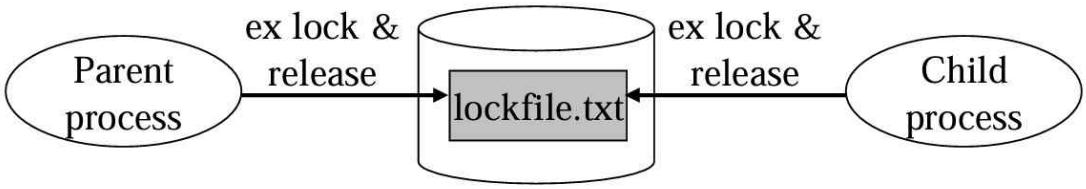
스레드와 시그널의 관계
리눅스/POSIX 환경에서는 시그널이 프로세스에 전달되지만, 실제 처리는 프로세스 내 스레드 중 하나가 담당한다. 핵심 동작은 다음과 같다. 시그널은 프로세스 전체로 들어옴 프로세스 내부에서 해당 시그널을 차단(block)하지 않은 스레드들 중 하나가 선택되어 시그널을 처리 필요할 경우, 특정 스레드만 시그널을 처리하도록 설정 가능 (pthread_sigmask(), pthread_kill() 활용) 즉, 시그널은 프로세스에 보내지지만, 실제 처리는 스레드 단위에서 이루어지는 구조이다.

6. 교재에서 프로세스 간 통신에 대한 내용을 읽고 이해한다.

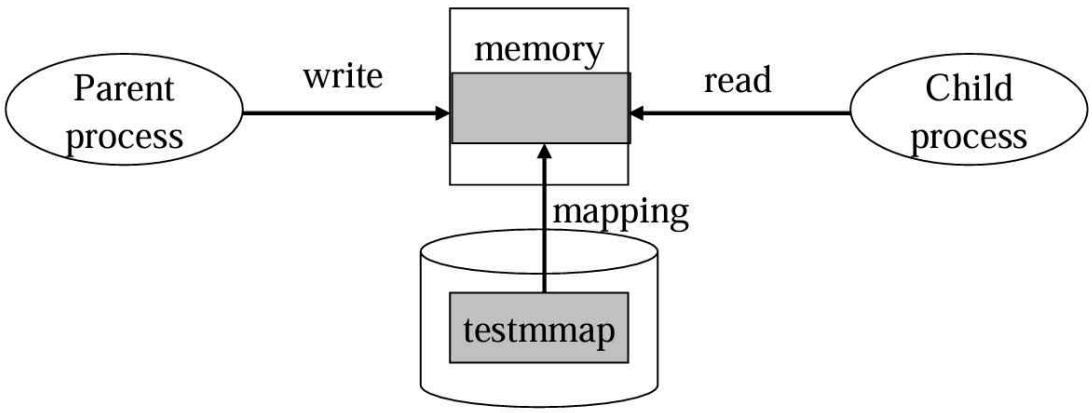
프로세스간 통신 기법
시그널, 파일잠금, 파이프, 메시지큐, 세마포어, 공유메모리, 소켓등
파일을 이용한 잠금(Lock) – 레코드 잠금(fcntl)과 flock(파일잠금)함수 활용 종류 - 공유적(shared)잠금 - -> 다른 프로세스 들이 해당영역에 공유잠금(읽기)은 가능 하지만 배제잠금(쓰기)은 불가능 - 배제적(exclusive) 잠금 -> 다른 프로세스들이 해당영역에 접근 불가
– 레코드 잠금(recordlocking) –> 프로세스가 특정 파일의 일부 레코드에 대하여 잠금 기능 설정 –> 다른 프로세스로 하여금 이 파일에 접근 하지 못 하게 한다.
- 메모리 맵 파일 공유 -> 특정 파일 영역을 프로세스 메모리 공간으로 매핑

파일을 이용한 레코드 잠금
fcntl 시스템호출 -> 파일제어: fd가 가리키는 파일을 cmd명령에 따라 제어한다.
사용법 #include <fcntl.h> int fcntl(int fd, int cmd, struct flock *lock); cmd: fcntl 이 어떻게동작할것인지결정 - F_GETLK : 레코드 잠금 정보 획득, 정보는 셋째 인자 lock에 저장 - F_SETLK : 파일에 레코드 잠금 적용, 불가하면 즉시-1 반환 - F_SETLKW : 파일에 레코드 잠금 적용, 불가하면 잠금해제를 기다림 -> 성공적인 호출에 대하여, 반환 값은 동작에 달려 있다

파일을 이용한 잠금
flock 시스템호출 -> ➢ fd가 가리키는 파일을 operation에 따라 잠금 또는 해제한다.
사용법 #include <sys/file.h> int flock(int fd, int operation); operation: flock이 어떻게 동작할 것인지 결정 - LOCK_SH : 공유 잠금 적용, 한 프로세스 이상 공유 가능 - LOCK_EX : 배제 잠금 적용, 한 프로세스만 잠금 가능 - LOCK_UN : 잠금 해제



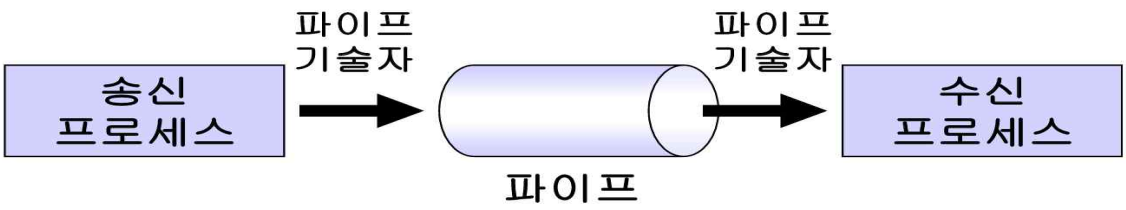
-> 파일 잠금 예제 프로그램 동작 과정

메모리맵 관련 시스템 호출
-메모리맵 영역 생성 및 삭제
사용법 #include <sys/mman.h> void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset); Int munmap(void *addr, size_t length) mmap()은 프로세스 가상공간에 매핑생성 - Prot: PROT_READ (읽기), PROT_WRITE(쓰기), PROT_EXEC(실행), PROT_NONE(접근불가) 설정 - Flags: MAP_SHARED(공유), MAP_PRIVATE(개별) 설정 mmap() 반환 값은 성공적인 호출에 대하여, 매핑 영역 주소를 반환하고, 실패시-1 반환 munmap()은 메모리 매핑 삭제 munmap 반환 값은 성공시 0을 반환, 실패시-1 반환
   -> 메모리 맵 파일 예제 프로그램 동작 과정
파일 잠금 및 메모리맵 파일의 단점 - 파일을 이용 - 추가적인 자원낭비 및 비효율적 - 경쟁 문제 - 외부에서 파일에 접근할 때의 보안 문제

파이프

-UNIX의 고유한 프로세스간 통신 방식

- 한 프로세스를 다른 프로세스와 연결 시켜주는 단방향 채널



파이프를 이용한 프로그래밍

pipe 시스템호출

- 파이프 생성
- 파이프를 가리키는 파일 기술자 쌍을 생성하고, 이를 filedes에 저장

사용법

```
#include <unistd.h>
 int pipe (int filedes[2]);
```

filedes[0]은 읽기 위한 파이프

filedes[1]은 쓰기 위한 파이프

반환값

- 정상적 종료: 0,

그렇지 않을 경우: 0이 아닌 값

➢오류번호(errno)

봉쇄(blocking) 읽기/쓰기

- 파이프로부터 자료를 읽거나 쓸 때까지 프로세스가 대기

비봉쇄(nonblocking) 읽기/쓰기

- 파이프로부터 자료를 읽거나 쓸 때 프로세스가 대기하지 않음

select 를 이용한 파이프 프로그래밍

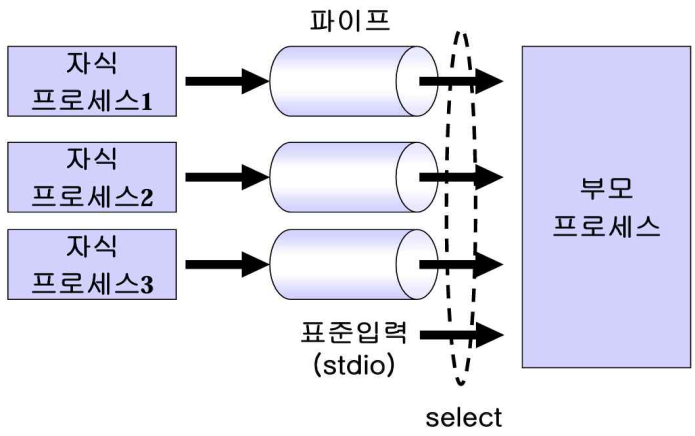
select 시스템 호출

- 다중 입출력 관리
- 지정된 파일 기술자 집합 중 읽기와 쓰기가 준비된 것이 있는지, 또 오류가 있는지 등을 검사

사용법

```
#include <sys/time.h>
 int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct
timeval *timeout);
 void FD_CLR(int fd, fd_set *set);
 void FD_ISSET(int fd, fd_set *set);
 void FD_SET(int fd, fd_set *set);
 void FD_ZERO(fd_set *set);
```

select 첫째 인자n은 검사할 파일 기술자 번호 최대값 +1
- FD_SET : 파일 기술자를 집합에 추가
- FD_CLR : 파일 기술자를 집합에서 뺀다
- FD_ISSET : 파일 기술자가 집합의 일부분 인지 아닌지를 검사
- FD_ZERO : 파일 기술자 집합을 초기화



고급 프로세스간 통신

고급 IPC개요

-UNIX와 LINUX에서 지원하는 고급 IPC 자원
1. 메시지 큐를 통하여 프로세스들간에 메시지 전달
2. 세마포어를 통하여 프로세스들간의 동기화
3. 프로세스 간 메모리 영역 공유
- 자원 요청시 동적으로 커널 데이터 구조 생성
-> 명시적으로 삭제 하지 않으면 시스템이 종료 될 때까지 메모리에 존재
-> 독립적인 프로세스 간 통신에 사용가능
- 종류
1, System V IPC (또는XSI IPC)
2, POSIX IPC

System V IPC

-관련 함수, 파일, 라이브러리

구분	메시지 큐	세마포어	공유메모리
헤더파일	sys/msg.h	sys/sem.h	sys/shm.h
생성, 개방 함수	msgget	semget	shmget
제어 함수	msgctl	semctl	shmctl
IPC 동작 함수	msgsnd msgrcv	semop	shmat shmdt

IPC 자원의 구별

-Key(32bit)

-> 파일 경로와 비슷하며 프로그래머가 자유롭게 선택 하거나 ftok로 생성

-IPC 설비 식별자(32bit)

-> 커널에 의해 자원에 부여 되며, 시스템내에서 고유한 값

- ftok함수 - 프로젝트ID로부터 key 값으로 반환

```
#include <sys/types.h>
 #include <sys/ipc.h>
 key_t ftok(const char *pathname, int proj_id);
```

고급 IPC 자원의 사용

IPC 자원의 생성

- 임의의 IPC키와 생성함수를 통하여 IPC 자원과 IPC 식별자를 생성
- 생성함수 -msgget, semget, shmget 함수
- IPC 식별자를 통한 자원 접근과 프로세스 간 통신
- 생성시 오류

오류코드	설명
EACCESS	프로세스가 적절한 접근 권한을 가지지 않는다.
EEXIST	프로세스가 이미 존재하는 키를 가진 **IPC** 자원의 생성을 시도
EIDRM	자원이 삭제된 것으로 표시되어 있다.
ENOENT	요청한 키의 **IPC** 자원이 존재하지 않고, 프로세스가 생성을 요구하지 않음
ENOMEM	추가 **IPC** 자원에 대한 공간이 남아 있지 않다.
ENOSPC	**IPC** 자원 수에 대한 최대 제한을 초과하였다.

-> 생성 함수의 마지막 인자: permflags

IPC 자원의 상태 정보

- 해당 자원에 관련된 데이터구조를 만들어 각종 정보 수록
- 허가 구조 포함: ipc_perm

```
 ushort cuid; /*IPC 객체 생성자의 사용자 id */
 ushort cgid; /* 생성자의 그룹 id */
ushort uid; /* 유효 사용자 id */
ushort gid; /* 유효 그룹-id */
mode_t mode; /* 허가권 비트 마스크 */
 ushort seq; /* slot usage sequence number */
```

- IPC 자원의제어

-> 자원의 상태 정보를 얻거나 제어하는데 사용

-> 제어함수 - msgctl, semctl, shmctl

- IPC 자원의 동작

-> 실제IPC 자원을 동작 시킨다

-> **동작함수**

1. msgsnd 와msgrcv 함수: 메시지큐를 통해 메시지를 전달 하거나 받는다
2. semop 함수: 세마포어를 얻거나 준다
3. shmat 와shmdt 함수: 공유 메모리를 프로세스의 주소 공간에 붙이거나 땐다

고급IPC 자원을 위해 쉘에서 제공하는 명령

ipcs - 전체 시스템에서 사용하고 있는 고급 IPC 자원 정보 출력

```
[linux@seps ipcs]$ ipcs------Shared Memory Segments -------
key        shmid      owner      perms      bytes      nattch     status
 0x00000049  1146880        linux        666                1024             0
------Semaphore Arrays -------
key        semid      owner      perms      nsems      status
0x00000049 65536       linux     666         1          ------Message Queues
-------
key        msqid      owner      perms      used-bytes   messages
0x00000049 0          linux     666        0            0
```

ipcrm -시스템으로부터 특정한 고급 IPC 자원을 제거

ipcrm	
일반형식	ipcrm [msg|sem|shm] id
주요옵션	msg : 메시지 큐를 삭제하는 경우 sem : 세마포어를 삭제하는 경우 shm : 공유 메모리를 삭제하는경우

Sys V 메시지큐

메시지 큐

- 프로세스 간에 문자나 바이트열로 이루어진 메시지를 전달하기위한 일종의 버퍼와같은 큐
- 메시지 큐와 파이프의 차이점

1. 파이프와 달리 메시지의 크기가 제한적
2. select 등을 사용할 수 없음
3. 우선순위 등에 따라 메시지 관리 가능

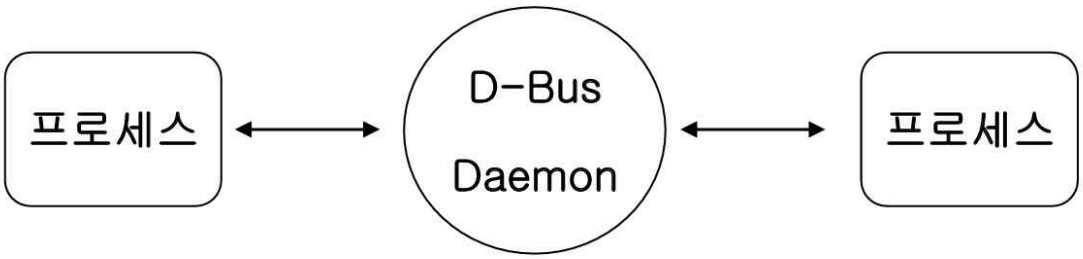
Sys V 메시지큐
msgget 시스템 호출 -> 메시지 큐 생성
사용법 #include <sys/msg.h> int msgget(key_t key, int permflags); -> key : 생성할 메시지큐에대한 고유번호 -> permflags : 생성 시 메시지큐 사용 허가권 지정 - 반환 값 -> 성공적인 호출에 대하여 메시지 큐 식별자 반환 -> 오류시에는 -1이 반환되고, errno 가 적절히 설정

메시지 큐 연산 함수 -> msgsnd()
#include <sys/msg.h> int msgsnd(int msqid, strucg msgbuf *msgp, size_t msgsz, int msgflg);
msgsnd : msqid가 가리키는 메시지큐에 메시지를 추가 한다.
메시지 큐 제어 함수 -> mshctl()
#include <sys/msg.h> ssize_t msgrcv (int msqid, struct msgbuf *msgp, size_t msgsz, long msgtyp, int msgflg);
-> 프로세스들간의 공유 자원 접근 제공(상호배제요구조건만족) -> 파이프나 메시지큐는 대규모 자료 전송에는 적합 하지만 자원 낭비가 크므로 동기화 수단으로는 적합하지 않다.

Sys V 세마포어
세마포어 - 프로세스 간 효율적인 동기화 방식 - 열쇠와 비슷하여, 어떤 프로세스가 세마포어를 획득하면 공유 자원에 접근하거나 동작을 수행하도록 허용하고, 그렇지 않으면 세마포어가 해제 되기를 기다리며 대기 **기능** - 프로세스간의 공유 자원 접근 제공 - 파이프나 메시지 큐는 대규모 자료 전송에는 적합하지만 자원 낭비가 크므로 동기화 수단으로는 적합하지 않다.

Sys V 공유메모리
공유 메모리 - 둘 이상의 프로세스가 특정 메모리 영역을 공유하여 자료에 접근 - 세가지 고급 IPC 기법 중에서 가장 유용 기능 -> 프로세스들 간의 자료 공유
공유 메모리 사용 - IPC 공유 메모리영역(shared memory region)에 대한 자료 구조 - 프로세스 접근을 위해 프로세스 주소 공간에 공유 메모리 영역 추가 - 사용이 끝나면 주소 공간에서 공유 메모리를 제거

7. 리눅스에서 DBUS 통신 기법에 대하여 조사하고 요약하여 보고서에 정리하여

D-BUS
리눅스 환경에서 응용 프로그램 간 메시지 버스(message bus) -독립적인 메시지 통신이 아니라 단일 가상 채널을 통해 프로세스 그룹 간 모든 통신을 수집하는 소프트웨어 추상화 제공 **-로컬 UNIX 도메인 소켓 기반 메시지 통신** - 리눅스의 많은 시스템 서비스 (예 udev, NetworkManager, logind) 활용
구성요소 -Bus Daemon - 중앙 메시지 중계기 - 응용 프로그램 -D-Bus 라이브러리를 이용하여 메시지를 송수신

버스유형 - System Bus - OS 전역 서비스용(예: udev, NetworkManager, logind) - Session Bus - 특정 사용자 세션 내 응용 간 통신 (예: 브라우저)

D-BUS개념
- 버스/ 서비스 이름-서비스를 제공하는 엔드포인트의 고유한 이름 -> 보통 도메인 주소의 역순으로 구성(예, org.freedesktop.Notifications) - 타입 시스템 - 서비스와 상호 작용하는 다양한 자료형 (객체포함) -> 객체를 나타내는 경로 필요(예, /org/freedesktop/Notifications) - 인터페이스 - 객체 타입에 구현된 API -> 메소드, 속성, 시그널 등 가능
C D-BUS 라이브러리
-libdbus -> 가장 보편적인 저수준 라이브러리. 클라이언트용으로는 부적합 하다. **- GDBus** -> Glib 기반의 라이브러리. 예제 풍부하지만 의존성 있음 **- sd-bus** ->Systemd 기반의 라이브러리. 코드기 간결하고 클라이언트용 가능하다.

SD-BUS -> 경량D-BUS IPC 클라이언트 라이브러리
D-BUS 프로그램구성
- D-BUS 서버 프로그램 -> 세션 버스 연결 -> D-BUS 서비스 이름 요청 -> 객체 경로 및 인터페이스 등록 -> 적절한 자료구조를 통해 호출 메소드, 속성, 시그널 정의 -> 루프를 돌며 버스 메시지처리 또는 시그널 전송 - D-BUS 클라이언트 프로그램 -> 세션 버스 연결 -> D-BUS 서비스 수행 1. 원격 메소드 호출 2. 속성 획득 및 설정 3. 루프를 돌며 버스 메시지 처리 및 시그널 수신

Busctl -> Systemd에서 제공하는 D-BUS 접속 유틸리티
시스템 버스 목록
$ busctl NAME PID PROCESS USER CO> … org.freedesktop.NetworkManager 689 NetworkManager root :1>
사용자 버스 목록 ->
$ busctl--user list NAME PID PROCESS USER CON> :1.0 835 systemd linux:1.> … org.freedesktop.DBus 835 systemd linux->

서비스 목록 검사 ->

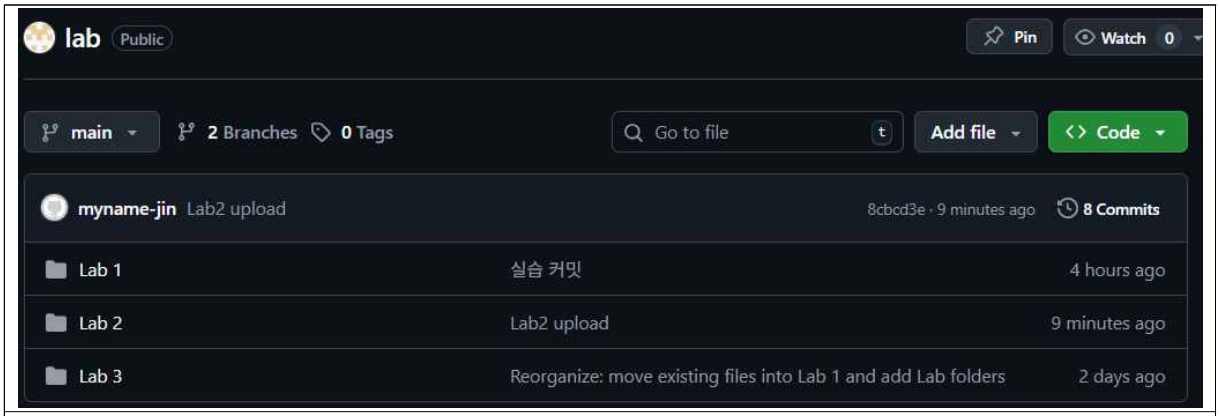
```
$ busctl tree org.freedesktop.Dbus
└─/org/freedesktop/DBus
```

메소드 호출

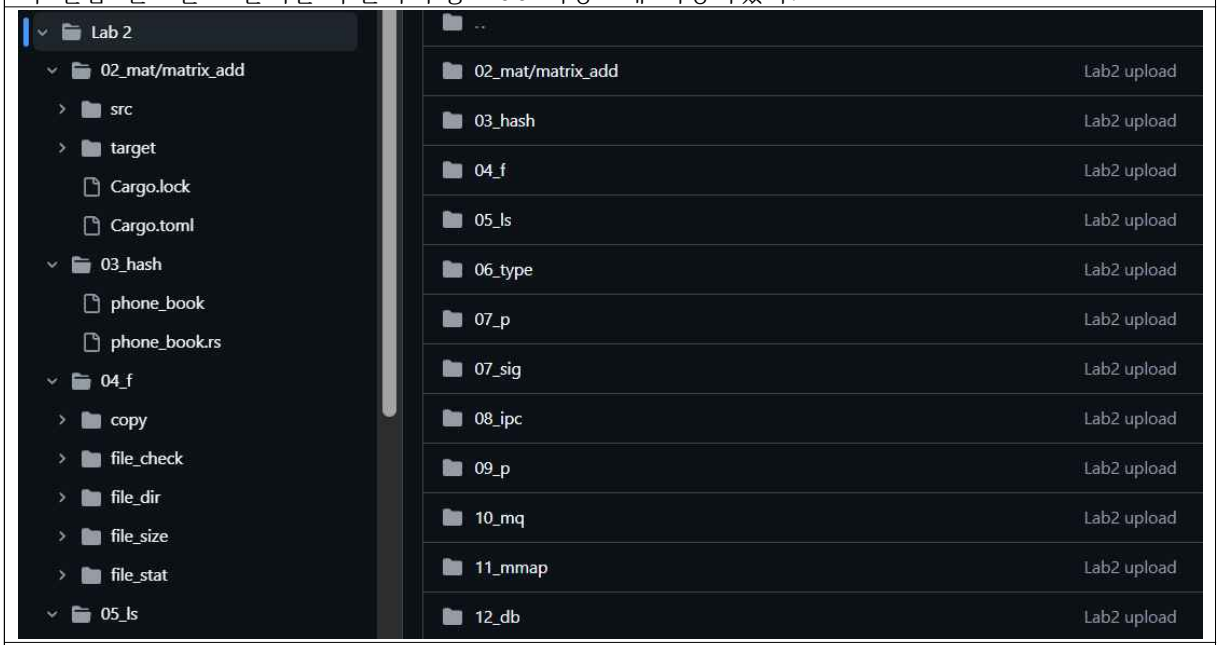
```
$ busctlcall org.freedesktop.DBus/org/freedesktop/DBusorg.freedesktop.DBusListNames
as 90 "org.freedesktop.DBus" ":1.7" ":1.569" "org.freedesktop.timesync1" ":1.8“ …
```

<실습문제>

1. 자신의 github 저장소에 lab2 프로젝트를 생성하고 아래의 모든 과제 프로그램을 업로드한다



각 실습 번호별로 폴더를 구분하여 github 저장소에 저장하였다.



URL https://github.com/myname-jin/lab

2. 두 행렬의 크기를 입력받아서 필요한 만큼의 메모리를 동적으로 할당하여 행렬의 내용을 저장하고 두 행렬을 더하는 프로그램을 Rust 의 Vec<T> 자료형을 를 활용하여 구현하시오.

2번 실습 문제를 해결하기 위해 cargo new matrix_add 명령어로 새 프로젝트를 생성한다.

```
02_mat/
 └─ matrix_add/
      ├─ Cargo.toml
      └─ src/
           └─ main.rs
```

-> 다음과 같은 경로로 파일이 생성된다.

cargo build 및 run

```
jinho@localhost:~/lab2/02_mat/matrix_add/src$ cargo build
   Compiling matrix_add v0.1.0 (/home/jinho/lab2/02_mat/matrix_add)
    Finished dev [unoptimized + debuginfo] target(s) in 0.98s
jinho@localhost:~/lab2/02_mat/matrix_add/src$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.04s
     Running `/home/jinho/lab2/02_mat/matrix_add/target/debug/matrix_add`
```

main.rs 코드

- 사용자에게 행/열 개수 같은 “양의 정수”를 안전하게 입력받는 함수

```rust
use std::io::{self, Write};

fn read_usize(prompt: &str) -> usize {
    loop {
        print!("{prompt}");
        // print! 출력이 바로 보이도록 flush
        io::stdout().flush().unwrap();

        let mut line = String::new();
        io::stdin().read_line(&mut line).expect("입력 오류");

        match line.trim().parse::<usize>() {
            Ok(v) if v > 0 => return v,
            _ => {
                println!("양의 정수를 입력하세요.");
            }
        }
    }
}
```

- 입력된 행·열 크기에 맞게 행렬 원소를 받아 Vec<Vec<i32>> 형태의 2차원 벡터로 생성하는 함수

```rust
fn read_matrix(rows: usize, cols: usize, name: &str) -> Vec<Vec<i32>> {
    println!("{name} 행렬의 원소를 입력하세요 (각 행마다 {cols}개씩):");
    // rows 개의 행을 담기 위해 미리 capacity를 지정
    let mut mat: Vec<Vec<i32>> = Vec::with_capacity(rows);
    for r in 0..rows {
        loop {
            print!("행 {}/{}: ", r + 1, rows);
            io::stdout().flush().unwrap();

            let mut line = String::new();
            io::stdin().read_line(&mut line).expect("입력 오류");

            let nums: Vec<i32> = line
                .split_whitespace()
                .filter_map(|s| s.parse().ok())
                .collect();
            if nums.len() == cols {
                mat.push(nums);
                break;
            } else {
                println!("정확히 {cols}개의 정수를 입력해야 합니다. 다시 입력하세요.");
            }
        }
    }
    mat
}
```

- 두 행렬의 같은 위치 원소를 더해 새로운 결과 행렬을 만들어 반환하는 함수

```rust
fn add_matrices(a: &Vec<Vec<i32>>, b: &Vec<Vec<i32>>) -> Vec<Vec<i32>> {
    let rows = a.len();
    let cols = if rows > 0 { a[0].len() } else { 0 };

    let mut c: Vec<Vec<i32>> = Vec::with_capacity(rows);

    for r in 0..rows {
        let mut row: Vec<i32> = Vec::with_capacity(cols);
        for c_idx in 0..cols {
            row.push(a[r][c_idx] + b[r][c_idx]);
        }
        c.push(row);
    }

    c
}
```

-2차원 벡터로 저장된 행렬을 보기 좋게 출력하는 함수

```rust
fn print_matrix(mat: &Vec<Vec<i32>>, name: &str) {
    println!("{name} 행렬:");
    for row in mat {
        for val in row {
            print!("{:4}", val);
        }
        println!();
    }
}
```

- 전체 프로그램의 흐름을 제어하며 입력 → 계산 → 출력 단계를 차례로 수행하는 함수

```rust
fn main() {
    println!("=== 두 행렬의 합 (Rust Vec<T> 이용) ===");

    let rows = read_usize("행의 개수를 입력하세요: ");
    let cols = read_usize("열의 개수를 입력하세요: ");

    let a = read_matrix(rows, cols, "첫 번째");
    let b = read_matrix(rows, cols, "두 번째");

    let c = add_matrices(&a, &b);

    println!();
    print_matrix(&a, "A");
    print_matrix(&b, "B");
    print_matrix(&c, "A + B");
}
```

실행결과

```
=== 두 행렬의 합 (Rust Vec<T> 이용) ===
행의 개수를 입력하세요: 2
열의 개수를 입력하세요: 3
첫 번째 행렬의 원소를 입력하세요 (각 행마다 3개씩):
행 1/2: 1 2 3
행 2/2: 4 5 7
두 번째 행렬의 원소를 입력하세요 (각 행마다 3개씩):
행 1/2: 7 8 9
행 2/2: 10 11 12

A 행렬:
   1   2   3
   4   5   7
B 행렬:
   7   8   9
  10  11  12
A + B 행렬:
   8  10  12
  14  16  19
```

-> 입력된 크기에 따라 Rust의 Vec<T> 구조를 사용해 행렬을 유연하게 구성하고 값을 저장한다. 이후 각 원소를 순서대로 더해 새로운 행렬을 생성하여 출력함으로써 동적 자료구조 기반의 행렬 연산을 구현

3. Rust 의 HashMap<K,V> 자료형을 활용하여 전화번호부(이름, 전화번호) 를 만들고 검색하는 프로그램을 구현하시오.

전화번호부 프로그램

```
use std::collections::HashMap;
use std::io;
fn main() {
    let mut phone_book = HashMap::new();

    phone_book.insert("홍길동".to_string(), "010-1111-1111".to_string());
    phone_book.insert("김철수".to_string(), "010-2222-2222".to_string());
    phone_book.insert("이영희".to_string(), "010-3333-3333".to_string());
    println!("--- Rust 전화번호부 ---");
    loop {
        println!("\n검색할 이름을 입력하세요 (종료하려면 '종료'):");
        let mut input = String::new();
        // 5. 키보드로 입력을 받습니다. (panic 대신 match로 수정)
        match io::stdin().read_line(&mut input) {
            Ok(_) => { // 입력 성공. 다음으로 넘어갑니다.
            }
            Err(error) => {// 입력 실패!
                println!("! 입력을 읽는 중 에러가 발생했습니다: {}", error);
                // 프로그램이 죽는 대신, 루프의 처음으로 돌아갑니다.
                continue;
            }
        }
        // 6. 입력받은 문자열의 앞뒤 공백과 줄바꿈(\n) 문자를 제거합니다.
        let name = input.trim();
        if name == "종료" {  // 7. "종료"라고 입력하면 loop를 빠져나갑니다.
            println!("프로그램을 종료합니다.");
            break;
        }
        match phone_book.get(name) { // 8. get 함수로 HashMap에서 이름을 검색합니다.
            Some(phone_number) => {
                println!(" -> {} 님의 전화번호는 {} 입니다.", name, phone_number);
            }
            None => {
                println!(" -> '{}' 님은 등록되어 있지 않습니다.", name);
            }}}
```

컴파일 및 테스트

```
jinho@localhost:~/lab2/03_hash$ rustc phone_book.rs
jinho@localhost:~/lab2/03_hash$ ./phone_book
--- Rust 전화번호부 ---

검색할 이름을 입력하세요 (종료하려면 '종료'):
홍길동
 -> 홍길동 님의 전화번호는 010-1111-1111 입니다.

검색할 이름을 입력하세요 (종료하려면 '종료'):
임진호
 -> '임진호' 님은 등록되어 있지 않습니다.

검색할 이름을 입력하세요 (종료하려면 '종료'):
이영희
 -> 이영희 님의 전화번호는 010-3333-3333 입니다.

검색할 이름을 입력하세요 (종료하려면 '종료'):
종료
프로그램을 종료합니다.
```

4. 파일 및 디렉토리와 관련된 함수들을 사용하여 프로그램을 작성하고 실행하여 보고, 익숙해지도록 사용해 본다.

저수준 파일처리 함수(파일 복사 프로그램)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>
#define MAX_READ 10

int main(int argc, char *argv[])
{
    int src_fd;
    int dst_fd;
    char buf[MAX_READ];
    ssize_t rcnt;
    ssize_t tot_cnt = 0;
    mode_t mode = S_IRUSR | S_IWUSR |
        S_IRGRP | S_IROTH;

    if (argc < 3) {
        fprintf(stderr,"Usage: file_copy src_file \
dest_file\n");
        exit(1);
}
    if ((src_fd = open(argv[1], O_RDONLY)) == -1 ){
        perror("src open");
        exit(1);
    }
    if ((dst_fd = creat(argv[2], mode)) == -1 ){
        perror("dst open");
        exit(1);
    }
    while ( (rcnt = read(src_fd, buf, MAX_READ)) > 0) {
        tot_cnt += write(dst_fd, buf, rcnt);
    }
    if (rcnt < 0) {
        perror("read");
        exit(1);
    }
    printf("total write count = %ld\n", tot_cnt);
    close(src_fd);
    close(dst_fd);
}
```

컴파일 후 테스트 파일을 만들어 실행

```
jinho@localhost:~/lab2/03_f/copy$ gcc -o file_copy file_copy.c
jinho@localhost:~/lab2/03_f/copy$ echo "This is a test file for low-level I/O." > source.txt
jinho@localhost:~/lab2/03_f/copy$ ./file_copy source.txt destination.txt
total write count = 39
```

파일 열기/닫기 프로그램

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h> /* exit()를 위해 추가 */
int main(int argc, char *argv[])
{     int fd;
      if (argc < 2) {
            fprintf(stderr,"Usage: file_check filename\n");
            exit(1);
      }
      if ((fd = open(argv[1], O_RDONLY)) == -1) {
            perror("open");
            exit(1);
      }
      printf("File \"%s\" found...\n", argv[1]);

      close(fd);
}
```

컴파일 후 파일실행 file_check.c를 열어보도록 한다.

```
jinho@localhost:~/lab2/03_f/file_check$ gcc -o file_check file_check.c
jinho@localhost:~/lab2/03_f/file_check$ ./file_check file_check.c
File "file_check.c" found...
```

저수준 파일 처리 함수(파일크기 알아내기 프로그램)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int fd;
    off_t size;
    if (argc < 2) {
        fprintf(stderr,"Usage: file_size filename\n");
        exit(1);
    }
    if ((fd = open(argv[1], O_RDONLY)) == -1){
        perror("open");
        exit(1);
    }
    size = lseek(fd, 0, SEEK_END);
    if (size < 0) {
        perror("lseek");
        exit(1);
    }

    /* off_t 타입은 long int일 수 있으므로 %ld 사용 */
    printf("%s's size = %ld\n", argv[1], size);

    close(fd);
}
```

컴파일 후 실행

```
jinho@localhost:~/lab2/03_f/file_size$ gcc -o file_size file_size.c
jinho@localhost:~/lab2/03_f/file_size$ ./file_size file_size.c
file_size.c's size = 667
```

ls -l 명령어로 결과물 검증

```
jinho@localhost:~/lab2/03_f/file_size$ ls -l file_size.c
-rw-r--r-- 1 jinho jinho 667 Nov 17 00:02 file_size.c
```

디렉토리 관련 함수(디렉토리 내용 보기 프로그램)

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>
#include <stdlib.h> /* exit()를 위해 추가 */
#include <errno.h>  /* perror()를 위해 추가 */

int main(int argc, char *argv[])
{
    DIR *pdir;
    struct dirent *pde;
    int i = 0;
    if (argc < 2) {
        fprintf(stderr,"Usage: file_dir dirname\n");
        exit(1);
    }
    if ((pdir = opendir(argv[1])) == NULL) {
        perror("opendir");
        exit(1);
    }
    while ((pde = readdir(pdir)) != NULL) {
        printf("%20s ", pde->d_name);
        if (++i % 3 == 0)
            printf("\n");
    }
    printf("\n");
    closedir(pdir);
}
```

컴파일 후 실행하기

```
jinho@localhost:~/lab2/03_f/file_dir$ gcc -o file_dir file_dir.c
jinho@localhost:~/lab2/03_f/file_dir$ ./file_dir .
                   ..            file_dir.c              file_dir
                    .
```

ls -a로 검증하기

```
jinho@localhost:~/lab2/03_f/file_dir$ ls -a
.   ..   file_dir   file_dir.c
```

저수준 파일 처리 함수(파일 형태 출력 프로그램)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdlib.h> /* exit()를 위해 추가 */
#include <errno.h>  /* perror()를 위해 추가 */

int main(int argc, char *argv[])
{
    struct stat buf;
    char *msg;
    if (argc < 2) {
        fprintf(stderr,"Usage: file_stat filename\n");
        exit(1);
    }
    if (lstat(argv[1], &buf) < 0) {
        perror("lstat");
        exit(1);
    }
    if (S_ISREG(buf.st_mode)) msg = "regular file";
    else if (S_ISDIR(buf.st_mode)) msg = "directory";
    else if (S_ISCHR(buf.st_mode)) msg = "character special file";
    else if (S_ISBLK(buf.st_mode)) msg = "block special file";
    else if (S_ISFIFO(buf.st_mode)) msg = "pipe of FIFO";
    else if (S_ISLNK(buf.st_mode)) msg = "symbolic link";
    else if (S_ISSOCK(buf.st_mode)) msg = "socket";
    else msg = "unknown";

    printf("\"%s\" is %s\n", argv[1], msg);
}
```

컴파일,테스트용 디렉토리 생성, 테스트용 심볼릭 링크 생성

```
jinho@localhost:~/lab2/03_f/file_stat$ gcc -o file_stat file_stat.c
jinho@localhost:~/lab2/03_f/file_stat$ mkdir test_dir
jinho@localhost:~/lab2/03_f/file_stat$ ln -s file_stat.c test_link
```

일반 파일 테스트, 디렉토리 테스트, 심볼릭 링크 테스트

```
jinho@localhost:~/lab2/03_f/file_stat$ ./file_stat file_stat.c
"file_stat.c" is regular file
jinho@localhost:~/lab2/03_f/file_stat$ ./file_stat test_dir
"test_dir" is directory
jinho@localhost:~/lab2/03_f/file_stat$ ./file_stat test_link
"test_link" is symbolic link
```

ls -l file_stat.c test_dir test_link 로 결과 검증

```
jinho@localhost:~/lab2/03_f/file_stat$ ls -l file_stat.c test_dir test_link
-rw-r--r-- 1 jinho jinho  996 Nov 17 00:19 file_stat.c
lrwxrwxrwx 1 jinho jinho   11 Nov 17 00:19 test_link -> file_stat.c
```

5. 주어진 디렉토리 내에 존재하는 파일과 디렉토리를 나열하고, 디렉토리의 경우 재귀적으로 방문해서 그 디렉토리 내에 존재하는 파일과 디렉토리를 나열하는 프로그램을 작성하시오. 즉, “ls -R” 명령과 동일한 결과를 보이도록 하시오.

파일 처리 (재귀적 디렉토리 탐색)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>      //  stat 함수와 함께 사용
#include <dirent.h>      // opendir, readdir, closedir
#include <string.h>      // strcmp, snprintf
#include <errno.h>       // perror

// 함수 선언
void list_directory(const char *path);

int main(int argc, char *argv[]) {
    const char *start_path = "."; // 기본 경로는 현재 디렉토리

    // 명령행 인자로 디렉토리 경로를 제공받으면 그 경로로 시작
    if (argc > 1) {
        start_path = argv[1];
    }

    // 디렉토리 나열 시작
    list_directory(start_path);
    return 0;
}

void list_directory(const char *path) {
    DIR *dir;
    struct dirent *entry;

    // 1. 디렉토리를 열기
    if ((dir = opendir(path)) == NULL) {
        perror("opendir");
        return;
    }

    printf("\n%s:\n", path); // 현재 디렉토리 경로 출력

    // 2. 디렉토리 내용을 읽으며 파일/디렉토리 이름 출력
    while ((entry = readdir(dir)) != NULL) {
        // 숨김 파일 및 현재/상위 디렉토리 제외
        if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0) {
            continue;
        }
        printf("%s  ", entry->d_name); // 파일/디렉토리 이름 출력
    }
    printf("\n");
```

```
    // 3. 다시 디렉토리 스트림을 읽기 위해 포인터를 처음으로 되돌림
    rewinddir(dir);

    // 4. 하위 디렉토리를 찾기 위해 다시 반복
    while ((entry = readdir(dir)) != NULL) {
        char sub_path[1024];

        // 숨김 파일 및 현재/상위 디렉토리 제외
        if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0) {
            continue;
        }
        // 전체 경로 생성 (예: "./test_dir")
        snprintf(sub_path, sizeof(sub_path), "%s/%s", path, entry->d_name);

        struct stat statbuf;

        // 5. 파일 정보를 가져옴
        if (stat(sub_path, &statbuf) == 0) {
            // 6. 항목이 디렉토리인지 확인
            if (S_ISDIR(statbuf.st_mode)) {
                // 7. 재귀 호출: 하위 디렉토리 탐색
                list_directory(sub_path);
            }
        } else {
            perror("stat");
        }
    }
    // 8. 디렉토리 닫기
    closedir(dir);
}
```

컴파일 및 테스트용 파일,폴더 생성

```
jinho@localhost:~/lab2/04_ls$ gcc -o ls_R ls_R.c
jinho@localhost:~/lab2/04_ls$ mkdir -p dir_A/sub_dir_A
jinho@localhost:~/lab2/04_ls$ mkdir dir_B
jinho@localhost:~/lab2/04_ls$ touch dir_A/file1.txt
jinho@localhost:~/lab2/04_ls$ touch dir_A/sub_dir_A/file2.txt
jinho@localhost:~/lab2/04_ls$ touch dir_B/file3.txt
```

실행

```
jinho@localhost:~/lab2/04_ls$ ./ls_R

.:
dir_B  dir_A  ls_R  ls_R.c

./dir_B:
file3.txt

./dir_A:
sub_dir_A  file1.txt

./dir_A/sub_dir_A:
file2.txt
```

ls -R 명령으로 검증

```
jinho@localhost:~/lab2/04_ls$ ls -R
.:
dir_A  dir_B  ls_R  ls_R.c

./dir_A:
file1.txt  sub_dir_A

./dir_A/sub_dir_A:
file2.txt

./dir_B:
file3.txt
```

-> 파일순서가 다른 것은 출력방식이 다른 것이다.

6. 몇 개의 문장을 타자하도록 하여 잘못 타이핑한 횟수와 평균 분당 타자수를 측정하는 타자 연습 프로그램을 구현하여 보시오.

여러 문장 타자 연습

```
#include <stdio.h>
#include <termios.h>     // tcgetattr, tcsetattr
#include <unistd.h>      // STDIN_FILENO
#include <wchar.h>        // wchar_t, wcslen, fgetwc, wprintf, putwchar
#include <locale.h>      // setlocale
#include <sys/time.h>    // gettimeofday
#include <stdlib.h>      // exit

// 1. 연습할 문장들 (한글 + 영문)
#define SENTENCE_COUNT 3
const wchar_t *sentences[SENTENCE_COUNT] = {
    L"동해물과 백두산이 마르고 닳도록",
    L"하느님이 보우하사 우리 나라 만세",
    L"Java와 Linux를 배우는 나는 동의대학교 임진호"
};

int main(void)
{
    int fd = STDIN_FILENO;
    struct termios init_attr, new_attr;
    wchar_t wch;

    int total_errors = 0;
    int total_chars = 0;

    struct timeval start, end;
    double elapsed_time;
    double wpm;

    setlocale(LC_ALL, "");

    // --- 터미널 설정 ---
    if (tcgetattr(fd, &init_attr) != 0) {
        perror("tcgetattr"); exit(1);
    }
    new_attr = init_attr;
    new_attr.c_lflag &= ~ICANON;
    new_attr.c_lflag &= ~ECHO;
    new_attr.c_cc[VMIN] = 1;
    new_attr.c_cc[VTIME] = 0;
    if (tcsetattr(fd, TCSANOW, &new_attr) != 0) {
        perror("tcsetattr"); exit(1);
    }

    wprintf(L"--- 타자 연습 시작 (총 %d개 문장) ---\n\n", SENTENCE_COUNT);
    fflush(stdout);

    gettimeofday(&start, NULL);
```

```
for (int i = 0; i < SENTENCE_COUNT; i++)
{
    const wchar_t *current_sentence = sentences[i];
    int len = wcslen(current_sentence);
    int current_pos = 0;

    wprintf(L"  %ls\n> ", current_sentence);
    fflush(stdout);

    // 4. 한 문장이 끝날 때까지 한 글자씩 입력받기
    while (current_pos < len)
    {
        wch = fgetwc(stdin);

        if (wch == WEOF) {
            perror("fgetwc"); break;
        }

        // --- 해결 1: Enter 키 입력 무시 ---
        // Enter 키가 눌리면 아무것도 안 하고 다음 입력을 기다림
        if (wch == L'\n') {
            continue;
        }

        // 4-1. 올바르게 입력한 경우
        if (wch == current_sentence[current_pos]) {
            putwchar(wch);
        }
        // 4-2. 오타인 경우
        else {
            // --- 해결 2: 영문/한글 구분하여 별표 출력 ---
            wchar_t correct_char = current_sentence[current_pos];

            // 원본 글자가 ASCII(영문, 숫자, 1칸 기호)인지 확인
            if (correct_char < 128) {
                putwchar(L'*'); // 1칸짜리 별표
            } else {
                putwchar(L'＊'); // 2칸짜리 (한글용) 별표
            }
            total_errors++;
        }

        fflush(stdout);
        current_pos++;
        total_chars++;
    }
    wprintf(L"\n\n");
}

gettimeofday(&end, NULL);
tcsetattr(fd, TCSANOW, &init_attr);
```

```
    // --- 최종 결과 계산 ---
    long seconds = end.tv_sec - start.tv_sec;
    long microseconds = end.tv_usec - start.tv_usec;
    elapsed_time = seconds + microseconds * 1e-6;

    wpm = (total_chars / 5.0) / (elapsed_time / 60.0);

    wprintf(L"--- 타자 연습 완료 ---\n");
    wprintf(L"총 소요 시간: %.2f초\n", elapsed_time);
    wprintf(L"총 오타 횟수: %d\n", total_errors);
    wprintf(L"평균 분당 타자수 (WPM): %.2f\n", wpm);

    return 0;
}
```

컴파일 및 실행

```
jinho@localhost:~/lab2/05_type$ vi typing_multi.c
jinho@localhost:~/lab2/05_type$ gcc -o typing_practice typing_multi.c
jinho@localhost:~/lab2/05_type$ ./typing_practice
```

실행 결과

```
--- 타자 연습 시작 (총 3개 문장) ---

  동해물과 백두산이 마르고 닳도록
> *해물과 백두산이 마르고 닳도록

  하느님이 보우하사 우리 나라 만세
> 하느님이 보*하***리 나라 만세

  Java와 Linux를 배우는 나는 동의대학교 임진호
> Java**Lin*x를 배우는 나는 동*대학교 ***호

--- 타자 연습 완료 ---
총 소요 시간: 134.75초
총 오타 횟수: 11
평균 분당 타자수 (WPM): 5.70
```

핵심 문제 해결

1. 직전줄에서 Enter 키를 입력하면 다음 줄에도 Enter 가 입력되어 오답 처리가 되는 문제점을
-> if (wch == L'\n') continue; 코드로 Enter 키를 무시하게 만들어 해결했습니다.

2. 칸 밀림 문제: 영어(1칸)와 한글(2칸)의 오타 표시(*) 크기가 안 맞는 문제를
-> if (correct_char < 128) 코드로 글자 크기를 감지해 1칸/2칸짜리 별표를 구분해서 출력하도록 해결했습니다.

7. 프로세스와 관련된 함수들을 사용하여 프로그램을 작성하고 실행하여 보고, 익숙해지도록 사용해 본다.

프로세스 생성 함수 사용 fork

```
#include <stdio.h>     /* printf()를 위해 추가 */
#include <sys/types.h>
#include <unistd.h>    /* fork()를 위해 */

int main()
{
    pid_t pid;

    printf("Calling fork\n");

    pid = fork();

    if (pid == 0)
        printf("I'm the child process\n");
    else if (pid > 0)
        printf("I'm the parent process\n");
    else
        printf("fork failed\n");
}
```

컴파일 후 실행결과

```
jinho@localhost:~/lab2/06_p/forkprocess$ gcc -o forkprocess forkprocess.c
jinho@localhost:~/lab2/06_p/forkprocess$ ./forkprocess
Calling fork
I'm the parent process
I'm the child process
```

프로세스 종료 함수 사용 exit() -> 소멸함수

```
#include <stdio.h>  /* printf(), scanf()를 위해 추가 */
#include <stdlib.h> /* exit()를 위해 */

int main()
{
    int exit_status;

    printf("enter exit status: ");
    scanf("%d", &exit_status);
    exit(exit_status);
}
```

컴파일 및 실행 // echo $? 명령으로 종료 상태 확인

```
jinho@localhost:~/lab2/06_p/forkprocess$ gcc -o exitprocess exitprocess.c
jinho@localhost:~/lab2/06_p/forkprocess$ ./exitprocess
enter exit status: 0
jinho@localhost:~/lab2/06_p/forkprocess$ echo $?
0
jinho@localhost:~/lab2/06_p/forkprocess$ ./exitprocess
enter exit status: 2
jinho@localhost:~/lab2/06_p/forkprocess$ echo $?
2
```

프로세스 동기화 함수

```
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h> /* printf(), perror()를 위해 추가 */

int main()
{
    pid_t pid;
    int status, exit_status;

    if ((pid = fork()) < 0) {
        perror("fork failed");
    }
    if (pid == 0) {
        /* 자식 코드 */
        sleep(4); /* 4초 동안 중단 */
        exit(5);  /* 5라는 종료 상태를 남기고 종료 */
    }
    /* 자식이 퇴장(exit)할 때까지 대기 */     /* --- 여기서부터 부모 코드 --- */
    if ((pid = wait(&status)) == -1) {
        perror("wait failed");
        exit(2);
    }
    /* 자식이 정상적으로 종료되었는지 확인 */
    if (WIFEXITED(status)) {
        /* 자식이 남긴 종료 상태(5)를 추출 */
        exit_status = WEXITSTATUS(status);
        printf("Exit status from %d was %d\n",
            pid, exit_status);
    }
    exit(0);
}
```

컴파일 및 실행

```
jinho@localhost:~/lab2/06_p/waitprocess$ gcc -o waitprocess waitprocess.c
jinho@localhost:~/lab2/06_p/waitprocess$ ./waitprocess
Exit status from 2112 was 5
jinho@localhost:~/lab2/06_p/waitprocess$
```

-> 4초동안 대기후 Exit status from 2112 was 5 출력

fork 와exec를 활용한 쉘(Shell)의 기본 동작 원리를 보여주는 프로그램

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>    /* printf(), perror()를 위해 추가 */
#include <stdlib.h>  /* exit()를 위해 추가 */
#include <sys/wait.h> /* wait()를 위해 추가 */

int main()
{
    pid_t pid;
    pid = fork();

    if (pid == 0) {
        /* 자식 프로세스 */
        execl("/bin/ls", "ls", "-l", (char *) 0);

        /* execl()이 성공하면 이 아래 코드는 실행되지 않음 */
        perror("execl failed");

    } else if (pid > 0) {
        /* 부모 프로세스 */
        wait((int *) 0); /* 자식이 끝날 때까지 대기 */
        printf("ls completed\n");
        exit(0);

    } else { /* fork 실패 */
        perror("fork failed");
    }
}
```

실행 결과

```
jinho@localhost:~/lab2/06_p/forkexecls$ gcc -o forkexecls forkexecls.c
jinho@localhost:~/lab2/06_p/forkexecls$ ./forkexecls
total 20
-rwxr-xr-x 1 jinho jinho 16176 Nov 17 13:27 forkexecls
-rw-r--r-- 1 jinho jinho   749 Nov 17 13:27 forkexecls.c
ls completed
```

-> 이 프로그램은 fork()로 자신과 똑같은 자식 프로세스를 하나 생성한다.

자식은 execl()을 통해 자신을 /bin/ls 명령어로 교체해 실행하고, 부모는 wait()로 자식이 끝날 때까지 기다렸다가 종료한다.

7-2 . 시그널과 관련된 함수들을 사용하여 프로그램을 작성하고 실행하여 보고, 익숙해지도록 사용해 본다

시그널 핸들러가 인터럽트 시그널을 받는 프로그램

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h> // exit()를 위해 추가

void handler(int signo);

int main()
{
    struct sigaction act;
    int i = 0;

    /* 1. 핸들러 함수 지정 */
    act.sa_handler = handler;

    /* 2. 모든 시그널을 일단 차단하도록 설정 (핸들러 실행 중) */
    sigfillset(&(act.sa_mask));

    /* 3. SIGINT(Ctrl+C)에 대해 act 설정 적용 */
    sigaction(SIGINT, &act, NULL);

    printf("SIGINT on\n");

    while(1) {
        sleep(1);
        printf("sleep for %d sec(s).\n", ++i);
    }
}
/* 시그널 핸들러 함수 */
void handler(int signo)
{
    printf("handler: signo=%d\n", signo);
}
```

컴파일 및 실행

```
jinho@localhost:~/lab2/07_sig/handlesignal$ gcc -o handlesignal handlesignal.c
jinho@localhost:~/lab2/07_sig/handlesignal$ ./handlesignal
SIGINT on
sleep for 1 sec(s).
sleep for 2 sec(s).
^Chandler: signo=2
sleep for 3 sec(s).
sleep for 4 sec(s).
sleep for 5 sec(s).
^Chandler: signo=2
sleep for 6 sec(s).
^Chandler: signo=2
sleep for 7 sec(s).
^\Quit (core dumped)
jinho@localhost:~/lab2/07_sig/handlesignal$
```

alarm 함수사용

```
/*
 * 8장 시그널
 * 파일 이름: alarmsignal.c
 * 기능: alarm()을 이용한 시그널 예약
 */
#include <stdio.h>
#include <signal.h>
#include <unistd.h> // alarm(), pause()
#include <stdlib.h>

void alarm_handler(int);
int alarm_flag = 0;

int main()
{
    struct sigaction act;

    act.sa_handler = alarm_handler;
    sigaction(SIGALRM, &act, NULL);

    alarm(5); /* 5초 뒤에 SIGALRM 시그널 예약 */
    pause();  /* 시그널이 올 때까지 멈춤 */
    if (alarm_flag)
        printf("Passed a 5 secs.\n");
}
void alarm_handler(int sig)
{
    printf("Received a alarm signal.\n");
    alarm_flag = 1;
}
```

컴파일 및 실행

```
jinho@localhost:~/lab2/07_sig/alarmsignal$ gcc -o alarmsignal alarmsignal.c
jinho@localhost:~/lab2/07_sig/alarmsignal$ ./alarmsignal
Received a alarm signal.
Passed a 5 secs.
jinho@localhost:~/lab2/07_sig/alarmsignal$
```

alarm(5) 함수로 5초 뒤에 SIGALRM 시그널을 예약한 뒤 , pause() 함수를 이용해 해당 시그널이 도착할 때까지 대기하는 예제입니다

시그널을 차단 하고 해제 하는 프로그램

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h> // sleep()
int main()
{
    sigset_t set1, set2;

    /* 1. set1: 모든 시그널을 포함 */
    sigfillset(&set1);

    /* 2. set2: SIGINT(Ctrl+C)만 포함 */
    sigemptyset(&set2);
    sigaddset(&set2, SIGINT);

    printf("Critical region start.\n");
    /* 3. [모든 시그널 차단] 시작 */
    sigprocmask(SIG_BLOCK, &set1, NULL);

    sleep(5);

    printf("Less critical region start.\n");
    /* 4. [SIGINT만 차단 해제] (다른 건 여전히 차단됨) */
    sigprocmask(SIG_UNBLOCK, &set2, NULL);

    sleep(5);

    printf("Non critical region start.\n");
    /* 5. [모든 시그널 차단 해제] */
    sigprocmask(SIG_UNBLOCK, &set1, NULL);

    sleep(5);
}
```

컴파일 및 실행 0~5초 동안은 ctrl+c가 차단되고 5초가 지난후 종료가 된다.

```
jinho@localhost:~/lab2/07_sig/blocksignal$ gcc -o blocksignal blocksignal.c
jinho@localhost:~/lab2/07_sig/blocksignal$ ./blocksignal
Critical region start.
^C^C^C
^C
^C
Less critical region start.
```

강제종료인 Ctrl+\가 차단되다가 모든 차단이 풀리는 "Non critical region"에 자동 종료

```
jinho@localhost:~/lab2/07_sig/blocksignal$ ./blocksignal
Critical region start.
^\^\^\^\^\^\^\^\Less critical region start.
^\^\^\^\
Non critical region start.
Quit (core dumped)
jinho@localhost:~/lab2/07_sig/blocksignal$
jinho@localhost:~/lab2/07_sig/blocksignal$
```

8. 프로세스 간 통신 함수들을 사용하여 프로그램을 작성하고 실행하여 보고, 익숙해지도록 사용해 본다.

익명 파이프 예제 프로그램

```
#include <stdio.h>     // printf(), perror(), sprintf()
#include <stdlib.h>    // exit()
#include <unistd.h>    // pipe(), fork(), read(), write(), close()
#include <string.h>    // C언어 문자열 처리를 위해 (sprintf)
#define MSGSIZE 16
int main()
{
    char buf[MSGSIZE];
    int p[2], i;
    int pid;

    /* 1. 파이프 생성 (읽기용 p[0], 쓰기용 p[1]) */
    if (pipe(p) == -1) {
        perror("pipe call failed");
        exit(1);
    }
    pid = fork();
    if (pid == 0) {
        /* --- 자식 프로세스 (송신) --- */
        close(p[0]); // 자식은 읽기용 p[0]을 닫음

        /* "Hello" 메시지를 파이프(p[1])에 씀 */
        for (i = 0; i < 2; i++) {
            sprintf(buf, "Hello, world #%d", i+1);
            write(p[1], buf, MSGSIZE);
        }
    }
    else if (pid > 0) {
        /* --- 부모 프로세스 (수신) --- */
        close(p[1]); // 부모는 쓰기용 p[1]을 닫음

        /* 파이프(p[0])에서 메시지를 읽어와 출력 */
        for (i = 0; i < 2; i++) {
            read(p[0], buf, MSGSIZE);
            printf("%s\n", buf);
        }
    }
    else {
        perror("fork failed");
    }
}
```

컴파일 및 실행

```
jinho@localhost:~/lab2/08_ipc/pipetest$ gcc -o pipetest pipetest.c
jinho@localhost:~/lab2/08_ipc/pipetest$ ./pipetest
Hello, world #1
Hello, world #2
jinho@localhost:~/lab2/08_ipc/pipetest$
```

FIFO 수신 프로그램 코드

```
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>    // read()
#include <stdlib.h>    // exit()
#include <sys/types.h> // mkfifo()
#include <sys/stat.h>  // mkfifo()
#define MSGSIZE 64
char *testfifo = "fifo"; // 통신에 사용할 파일 이름
int main (int argc, char **argv)
{     int fd;
     char buf[MSGSIZE];

     /* 1. "fifo"라는 이름의 명명된 파이프 생성 */
     if (mkfifo(testfifo, 0666) == -1) {
          perror("mkfifo failed");
          // exit(1); //  "File exists" 오류 시에도 계속 진행하기 위해 주석 처리
     }
     /* 2. "fifo" 파일을 (읽기+쓰기) 모드로 열기 */
     if ((fd = open(testfifo, O_RDWR)) < 0) {
          perror("fifo open failed");
          exit(1);
     }
     /* 3. "fifo" 파일에서 메시지를 계속 읽어와 출력 */
     while (1) {
          if (read(fd, buf, MSGSIZE) < 0) {
               perror("fifo read failed");
               exit(1);
          }
          printf("received message: %s\n", buf);
     }
}
```

FIFO 송신 코드

```
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>    // write()
#include <stdlib.h>    // exit()
#include <string.h>    // strcpy()
#define MSGSIZE 64
char *testfifo = "fifo"; // 접속할 파일 이름
int main (int argc, char **argv)
{    int fd, i, nwrite;
     char buf[MSGSIZE];
     if (argc < 2) {
          fprintf(stderr, "Usage: ./writefifo msg ...\n");
          exit(1);
     }
     /* 1. "fifo" 파일을 (쓰기 전용) 모드로 열기 */
     if ((fd = open(testfifo, O_WRONLY)) < 0) {
          perror("fifo open failed");
          exit(1);
     }
     /* 2. 프로그램 실행 시 입력받은 인자(argv)들을 "fifo"에 쓴다 */
     for (i = 1; i < argc; i++) {
          strcpy(buf, argv[i]);
          if ((nwrite = write(fd, buf, MSGSIZE)) < 0) {
               perror("fifo write failed");
               exit(1);
          }
     }
}
```

컴파일 및 실행

```
jinho@localhost:~/lab2/08_ipc/fifo$ gcc -o readfifo readfifo.c
gcc -o writefifo writefifo.c
jinho@localhost:~/lab2/08_ipc/fifo$ ./readfifo &
[1] 3549
```

수신 파일 상태 p는 파이프 파일임을 알려준다.

```
jinho@localhost:~/lab2/08_ipc/fifo$ ls -l
total 40
prw-r--r-- 1 jinho jinho     0 Nov 17 18:12 fifo
-rwxr-xr-x 1 jinho jinho 16216 Nov 17 18:11 readfifo
-rw-r--r-- 1 jinho jinho  1091 Nov 17 18:10 readfifo.c
-rwxr-xr-x 1 jinho jinho 16296 Nov 17 18:11 writefifo
-rw-r--r-- 1 jinho jinho   927 Nov 17 18:10 writefifo.c
```

수신과 송신 두 터미널로 데이터를 전달

```
jinho@localhost:~/lab2/08_ipc/fifo$ received message
: Hello
received message: World
jinho@localhost:~/lab2/08_ipc/fifo$ received message
: 다시테스트
```

```
readfifo  readfifo.c  writefifo  writefifo.c
jinho@localhost:~/lab2/08_ipc/fifo$ ./writefifo Hello World
jinho@localhost:~/lab2/08_ipc/fifo$ ./writefifo 다시테스트
jinho@localhost:~/lab2/08_ipc/fifo$
```

Sys V 세마포어

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>    // exit(), srand(), rand()
#include <unistd.h>    // fork(), getpid(), sleep()

#define SEMKEY (key_t)0111 // 세마포어를 식별하기 위한 키

/* 함수 선언 */
void testsem(int semid);
void p(int semid);
void v(int semid);

/* * main 함수 (부모): 세마포어를 만들고,
 * 자식 3개를 생성한 뒤, 10초 후 세마포어를 제거한다.
 */
int main()
{
    int semid, i;
    union semun {
        int value;
        struct semid_ds *buf;
        unsigned short int *array;
    } arg; // semctl에 값을 전달하기 위한 공용체

    /* 1. 세마포어 1개를 생성 (없으면 IPC_CREAT) */
    if ((semid = semget(SEMKEY, 1, IPC_CREAT | 0666)) == -1) {
        perror("semget failed");
        exit(1);
    }

    /* 2. 세마포어 0번의 값을 1로 초기화 (1명만 통과 가능) */
    arg.value = 1;
    if (semctl(semid, 0, SETVAL, arg) == -1) {
        perror("semctl failed");
        exit(1);
    }

    /* 3. 자식 프로세스 3개 생성 */
    for (i = 0; i < 3; i++) {
        if (!fork()) // fork() == 0, 즉 자식 프로세스라면
            testsem(semid); // testsem 함수 실행
    }
    sleep(10); // 자식들이 모두 끝날 때까지 10초 대기
    /* 4. 세마포어 제거(IPC_RMID) */
    if (semctl(semid, 0, IPC_RMID, arg) == -1) {
        perror("semctl failed");
        exit(1);
    }
}
```

```
/* * testsem 함수 (자식): 세마포어(lock)를 획득하고,
 * 랜덤 시간 동안 "작업"한 뒤, 세마포어를 반납한다. */
void testsem(int semid)
{
    srand((unsigned int)getpid());

    p(semid); // "잠금" (P 연산)
    printf("process %d: semaphore in use\n", getpid());
    sleep(rand() % 5); // 0~4초간 "작업" (sleep)

    printf("process %d: putting semaphore\n", getpid());
    v(semid); // "해제" (V 연산)

    exit(0);
}
/* P 연산: 세마포어 값을 1 감소 (잠금 요청) */
void p(int semid)
{
    struct sembuf pbuf;
    pbuf.sem_num = 0;   // 0번 세마포어
    pbuf.sem_op = -1;  // 값을 1 감소
    pbuf.sem_flg = SEM_UNDO;
    if (semop(semid, &pbuf, 1) == -1) {
        perror("semop p failed");
        exit(1);
    }
}
/* V 연산: 세마포어 값을 1 증가 (잠금 해제) */
void v(int semid)
{
    struct sembuf vbuf;
    vbuf.sem_num = 0;
    vbuf.sem_op = 1;   // 값을 1 증가
    vbuf.sem_flg = SEM_UNDO;
    if (semop(semid, &vbuf, 1) == -1) {
        perror("semop v failed");
        exit(1);
    }
}
```

컴파일 및 실행

```
jinho@localhost:~/lab2/08_ipc/testsem$ gcc -o testsem testsem.c
jinho@localhost:~/lab2/08_ipc/testsem$ ./testsem
process 3705: semaphore in use
process 3705: putting semaphore
process 3706: semaphore in use
process 3706: putting semaphore
process 3707: semaphore in use
process 3707: putting semaphore
jinho@localhost:~/lab2/08_ipc/testsem$
```

process 3705: semaphore in use (3705번 자식이 "열쇠"를 가짐)

process 3705: putting semaphore (3705번 자식이 "열쇠"를 반납)

9. 파이프를 사용하여 텍스트 기반의 간단한 채팅 프로그램을 구현하시오.

명명된 파이프(FIFO)를 이용한 양방향 통신 프로그램

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>      // read, write, close, STDIN_FILENO
#include <string.h>      // strlen, bzero, strcmp
#include <fcntl.h>       // O_RDWR, O_WRONLY, O_NONBLOCK
#include <sys/select.h> // select(), FD_SET, FD_ZERO, FD_ISSET
#include <sys/types.h>  // mkfifo()
#include <sys/stat.h>   // mkfifo()

#define BUF_SIZE 256

const char *FIFO_A = "chat_fifo1"; // 1 -> 2
const char *FIFO_B = "chat_fifo2"; // 2 -> 1

int main(int argc, char *argv[])
{
    int read_fd, write_fd;
    char buf[BUF_SIZE];
    int nbytes;
    fd_set read_fds;
    int maxfd;

    if (argc != 2) {
        fprintf(stderr, "사용법: ./fifo_chat [사용자번호 (1 또는 2)]\n");
        exit(1);
    }

    // FIFO 파일 2개 생성
    mkfifo(FIFO_A, 0666);
    mkfifo(FIFO_B, 0666);

    // 사용자 번호에 따라 읽기/쓰기 파이프 설정
    if (strcmp(argv[1], "1") == 0) {
        printf("--- 채팅 클라이언트 1 (송신: %s, 수신: %s) ---\n", FIFO_A, FIFO_B);
        printf("상대방이 접속할 때까지 잠시 기다릴 수 있습니다...\n");

        // 1번 사용자는 A(1)에 쓰고, B(2)에서 읽음
        write_fd = open(FIFO_A, O_WRONLY);
        read_fd = open(FIFO_B, O_RDONLY); // 상대가 open할 때까지 대기(Block)

    } else if (strcmp(argv[1], "2") == 0) {
        printf("--- 채팅 클라이언트 2 (송신: %s, 수신: %s) ---\n", FIFO_B, FIFO_A);

        //  2번 사용자는 A(1)에서 먼저 읽고, B(2)에 씀
        read_fd = open(FIFO_A, O_RDONLY);
        write_fd = open(FIFO_B, O_WRONLY);
    } else {
        fprintf(stderr, "사용자 번호는 1 또는 2 여야 합니다.\n");
        exit(1);
    }
```

```
if (read_fd < 0 || write_fd < 0) {
    perror("FIFO 열기 실패");
    exit(1);
}

printf("--- 채팅이 연결되었습니다. 'quit' 입력 시 종료 ---\n");

// select()가 감시할 파일 디스크립터 최대값 설정
maxfd = (read_fd > STDIN_FILENO ? read_fd : STDIN_FILENO) + 1;

while (1) {
    // 감시 대상 초기화
    FD_ZERO(&read_fds);
    FD_SET(STDIN_FILENO, &read_fds); // 키보드(0번) 감시
    FD_SET(read_fd, &read_fds);      // 상대방 파이프 감시

    printf("나: ");
    fflush(stdout);

    // 키보드 또는 상대방 파이프에서 입력이 올 때까지 대기
    if (select(maxfd, &read_fds, NULL, NULL, NULL) < 0) {
        perror("select error");
        break;
    }

    // 키보드 입력이 감지된 경우 (내가 메시지 보냄)
    if (FD_ISSET(STDIN_FILENO, &read_fds)) {
        nbytes = read(STDIN_FILENO, buf, BUF_SIZE);
        if (nbytes > 0) {
            buf[nbytes] = '\0';
            write(write_fd, buf, nbytes); // 상대방 파이프로 전송

            if (strncmp(buf, "quit", 4) == 0) {
                break;
            }
        }
    }

    // 상대방 파이프 입력이 감지된 경우 (상대 메시지 도착)
    if (FD_ISSET(read_fd, &read_fds)) {
        nbytes = read(read_fd, buf, BUF_SIZE);

        if (nbytes <= 0) {
            printf("\r상대방이 채팅을 종료했습니다.\n");
            break;
        }

        buf[nbytes] = '\0';
        // "\r"은 "나: " 프롬프트를 덮어쓰고 줄의 맨 앞으로 이동
        printf("\r상대방: %s", buf);
```

```
            if (strncmp(buf, "quit", 4) == 0) {
                break;
            }
        }
    }

    close(read_fd);
    close(write_fd);
    printf("--- 채팅을 종료합니다. ---\n");
    return 0;
}
```

1번 사용자로 프로그램 시작 -> 2번 사용자가 접속까지 대기

```
jinho@localhost:~/lab2/09_p$ ./chat 1
--- 채팅 클라이언트 1 (송신: chat_fifo1, 수신: chat_fifo2) ---
상대방이 접속할 때까지 잠시 기다릴 수 있습니다...
```

2번 사용자의 접속

```
jinho@localhost:~/lab2/09_p$ ./chat 2
--- 채팅 클라이언트 2 (송신: chat_fifo2, 수신: chat_fifo1) ---
--- 채팅이 연결되었습니다. 'quit' 입력 시 종료 ---
나:
```

대기중인 1번 사용자와 자동 연결된다

```
jinho@localhost:~/lab2/09_p$ ./chat 1
--- 채팅 클라이언트 1 (송신: chat_fifo1, 수신: chat_fifo2) ---
상대방이 접속할 때까지 잠시 기다릴 수 있습니다...
--- 채팅이 연결되었습니다. 'quit' 입력 시 종료 ---
나:
```

1번 사용자

```
jinho@localhost:~/lab2/09_p$ ./chat 1
--- 채팅 클라이언트 1 (송신: chat_fifo1, 수신: chat_fifo2) ---
상대방이 접속할 때까지 잠시 기다릴 수 있습니다...
--- 채팅이 연결되었습니다. 'quit' 입력 시 종료 ---
나: 리눅스 수업은 잘 따라가고 있니?
상대방: 조금 어렵지만 잘 따라가고 있어
나: 오늘 춥던데 괜찮니?
상대방: 어제 패딩을 하나 구매해서 괜찮은 것 같아
나: 다행이다
상대방: 너도 하나 사도록 해!
나: 알겠어 추천 고마워!
```

2번 사용자

```
jinho@localhost:~/lab2/09_p$ ./chat 2
--- 채팅 클라이언트 2 (송신: chat_fifo2, 수신: chat_fifo1) ---
--- 채팅이 연결되었습니다. 'quit' 입력 시 종료 ---
상대방: 리눅스 수업은 잘 따라가고 있니?
나: 조금 어렵지만 잘 따라가고 있어
상대방: 오늘 춥던데 괜찮니?
나: 어제 패딩을 하나 구매해서 괜찮은 것 같아
상대방: 다행이다
나: 너도 하나 사도록 해!
상대방: 알겠어 추천 고마워!
```

10. 메시지 큐를 사용하여 텍스트 기반의 간단한 채팅 프로그램을 구현하시오

메시지 큐를 이용한 양방향 채팅 프로그램 (요청-응답 모델)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>

#define BUFSIZE 1024 // 안전하게 버퍼 크기를 1024로 설정
#define MSG_TYPE_A 1 // 사용자 1의 메시지 유형
#define MSG_TYPE_B 2 // 사용자 2의 메시지 유형

// 메시지 구조체
struct msg_buffer {
    long msg_type;
    char text[BUFSIZE];
};

int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("사용법: %s [송신 유형] [수신 유형]\n", argv[0]);
        printf("예: ./mq_chat 1 2 (유형 1로 보내고 유형 2를 받음)\n");
        return 1;
    }
    // 1. 송신/수신 메시지 유형 설정
    long send_type = atol(argv[1]);
    long receive_type = atol(argv[2]);

    // 2. 메시지 큐 Key 생성 (ftok)
    // 현재 디렉토리(".")의 inode와 '65'를 조합해 Key를 만듭니다.
    key_t key = ftok(".", 65);

    // 3. 메시지 큐 생성 및 접근 (IPC_CREAT | 0666)
    int msg_id = msgget(key, 0666 | IPC_CREAT);
    if (msg_id == -1) {
        perror("msgget failed");
        return 1;
    }
    struct msg_buffer msg;
    char message[BUFSIZE];

    printf("--- 메시지 큐 채팅 시작 ---\n");
    printf("현재 사용자: 송신 유형 %ld, 수신 유형 %ld\n", send_type, receive_type);
    printf("종료하려면 'exit'을 입력하세요.\n");

    while (1) {
        // --- 메시지 전송 (키보드 입력) ---
        printf("나> ");
        fflush(stdout);
```

```
        // 4. 키보드 입력
        if (fgets(message, sizeof(message), stdin) == NULL) break;
        message[strcspn(message, "\n")] = '\0'; // 개행 문자 제거

        // 5. 종료 확인 및 전송
        if (strcmp(message, "exit") == 0) {
            // 상대방이 기다리고 있으므로, 종료 메시지를 보낸 후 break
            msg.msg_type = send_type;
            strncpy(msg.text, message, sizeof(msg.text));
            msgsnd(msg_id, &msg, strlen(msg.text) + 1, 0);
            printf("채팅을 종료합니다.\n");
            break;
        }
        // 6. 메시지 전송 (msgsnd)
        msg.msg_type = send_type;
        strncpy(msg.text, message, sizeof(msg.text));
        if (msgsnd(msg_id, &msg, strlen(msg.text) + 1, 0) == -1) {
            perror("msgsnd failed");
            break;
        }
        // --- 메시지 수신 (상대방 응답 대기) ---
        // 7. 수신 (msgrcv) - 상대방이 보낼 때까지 멈춰서 기다림
        if (msgrcv(msg_id, &msg, sizeof(msg.text), receive_type, 0) == -1) {
            if (errno == EIDRM) { // 상대방이 큐를 삭제하고 종료했을 경우
                printf("상대방이 채팅을 종료했습니다.\n");
                break;
            }
            perror("msgrcv failed");
            break;
        }
        // 8. 수신된 메시지 출력
        printf("상대방> %s\n", msg.text);

        // 9. 상대방이 'exit'을 보냈다면 종료
        if (strcmp(msg.text, "exit") == 0) {
            printf("상대방이 채팅을 종료했습니다.\n");
            break;
        }
    }
    // 10. 1번 사용자(send_type=1)가 종료할 때만 메시지 큐를 삭제
    if (send_type == 1) {
        printf("메시지 큐(ID %d)를 시스템에서 삭제합니다.\n", msg_id);
        msgctl(msg_id, IPC_RMID, NULL);
    }
    return 0;
}
```

컴파일 및 IPC 시스템을 위한 준비 작업

```
jinho@localhost:~/lab2/10_mq$ touch chat_key
jinho@localhost:~/lab2/10_mq$ gcc -o mq_chat mq_chat.c
```

1번 사용자	2번 사용자
jinho@localhost:~/lab2/10_mq$ ./mq_chat 1 2 --- 메시지 큐 채팅 시작 --- 현재 사용자: 송신 유형 1, 수신 유형 2 종료하려면 'exit'을 입력하세요. 나> 안녕하세요 저는 동의대학생 입니다. 상대방> 안녕하세요 저는 동아대생 입니다. 나> 같은 부산지역 학생이네요!? 상대방> 저희 동생도 동의대학생 입니다 나>	jinho@localhost:~/lab2/10_mq$ ./mq_chat 2 1 --- 메시지 큐 채팅 시작 --- 현재 사용자: 송신 유형 2, 수신 유형 1 종료하려면 'exit'을 입력하세요. 나> 안녕하세요 저는 동아대생 입니다. 상대방> 안녕하세요 저는 동의대학생 입니다. 나> 저희 동생도 동의대학생 입니다 상대방> 같은 부산지역 학생이네요!?

11. 공유 메모리와 mmap을 사용하여 한 파일을 다른 파일로 복사하는 프로그램을 작성하시오.

mmap을 이용한 고속 파일 복사

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/mman.h> // mmap, munmap
#include <sys/stat.h> // fstat
#include <sys/types.h>

void fatal(const char *s) {
    perror(s);
    exit(1);
}
int main(int argc, char *argv[])
{
    int fd_in, fd_out;
    void *src, *dst;
    struct stat statbuf;
    size_t length;

    if (argc != 3) {
        fprintf(stderr, "사용법: %s <원본 파일> <대상 파일>\n", argv[0]);
        exit(1);
    }
```

```
    /* 1. 원본 파일 열기 및 크기 확인 */
    if ((fd_in = open(argv[1], O_RDONLY)) < 0)
        fatal("원본 파일 열기 실패");

    if (fstat(fd_in, &statbuf) < 0) // fstat으로 파일 정보(크기) 획득
        fatal("fstat 실패");

    length = statbuf.st_size;

    /* 2. 대상 파일 생성 및 크기 설정 */
    if ((fd_out = open(argv[2], O_RDWR | O_CREAT, 0666)) < 0)
        fatal("대상 파일 생성 실패");

    if (ftruncate(fd_out, length) == -1) // 대상 파일 크기를 원본과 동일하게 설정
        fatal("ftruncate 실패");

    /* 3. 메모리 매핑: 원본 파일 (읽기 전용) */
    src = mmap(0, length, PROT_READ, MAP_SHARED, fd_in, 0);
    if (src == MAP_FAILED)
        fatal("원본 mmap 실패");
    close(fd_in);

    /* 4. 메모리 매핑: 대상 파일 (쓰기 전용) */
    dst = mmap(0, length, PROT_READ | PROT_WRITE, MAP_SHARED, fd_out, 0);
    if (dst == MAP_FAILED)
        fatal("대상 mmap 실패");
    close(fd_out);

    /* 5. 복사: 메모리에서 메모리로 (단 한 번의 memcpy) */
    memcpy(dst, src, length);

    /* 6. 메모리 매핑 해제 및 종료 */
    if (munmap(src, length) == -1 || munmap(dst, length) == -1)
        fatal("munmap 실패");

    printf("파일 복사 완료: %s -> %s (크기: %zu 바이트)\n", argv[1], argv[2], length);

    return 0;
}
```

컴파일, 테스트파일 생성

```
jinho@localhost:~/lab2/11_mmap$ vi mmap_copy.c
jinho@localhost:~/lab2/11_mmap$ gcc -o mmap_copy mmap_copy.c
jinho@localhost:~/lab2/11_mmap$ echo "Hello, Memory Mapping World!" > source_file.txt
```

실행 및 검증

```
jinho@localhost:~/lab2/11_mmap$ ./mmap_copy source_file.txt destination_file.txt
파일 복사 완료: source_file.txt -> destination_file.txt (크기: 29 바이트)
jinho@localhost:~/lab2/11_mmap$ cat destination_file.txt
Hello, Memory Mapping World!
jinho@localhost:~/lab2/11_mmap$ diff source_file.txt destination_file.txt
jinho@localhost:~/lab2/11_mmap$
```

diff 명령어로 두 파일 차이를 비교

12. DBUS 를 사용하여 알림 메시지를 전달하는 서버-클라이언트 프로그램을 구현하시오.

D-BUS를 이용한 서버 코드

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <systemd/sd-bus.h>

// --- 오류 해결을 위해 함수 프로토타입을 미리 선언 ---
static int hello_method(sd_bus_message *m, void *userdata, sd_bus_error *error);
static int get_count_property(sd_bus *bus, const char *path, const char *interface,
                const char *property, sd_bus_message *reply, void *userdata, sd_bus_error *error);
// -------------------------------------------------------

#define BUS_NAME "org.example.NotificationService"
#define OBJECT_PATH "/org/example/NotificationService"
#define INTERFACE_NAME "org.example.NotificationService"

// 서버의 상태를 유지할 변수
static uint32_t count = 0;
static sd_bus *bus = NULL;

// 서버의 메서드 및 속성을 정의하는 테이블
static const sd_bus_vtable service_vtable[] = {
    SD_BUS_VTABLE_START(0),
    SD_BUS_METHOD("Hello", "s", "s", hello_method, SD_BUS_VTABLE_UNPRIVILEGED),
    SD_BUS_PROPERTY("Count", "u", get_count_property, 0, SD_BUS_VTABLE_PROPERTY_CONST),
    SD_BUS_VTABLE_END
};
// --- 함수 정의 ---
static int hello_method(sd_bus_message *m, void *userdata, sd_bus_error *error)
{
    const char *client_name;
    char reply[100];
    int ret;

    ret = sd_bus_message_read(m, "s", &client_name);
    if (ret < 0) return ret;

    sprintf(reply, "Hello %s! Server has been called %d times.", client_name, ++count);

    return sd_bus_reply_method_return(m, "s", reply);
}
static int get_count_property(sd_bus *bus, const char *path, const char *interface,
                const char *property, sd_bus_message *reply, void *userdata, sd_bus_error
*error)    {
    // 현재 count 값을 클라이언트에게 'u'(uint32_t) 형태로 반환
    uint32_t *pcount = (uint32_t *)userdata;
    return sd_bus_message_append(reply, "u", *pcount);
}
```

```
int main(void)
{
    sd_bus_slot *slot = NULL;
    int ret;
    uint32_t tick_count = 0;

    ret = sd_bus_default_user(&bus);
    if (ret < 0) {
        fprintf(stderr, "Failed to connect to dbus-daemon: %s\n", strerror(-ret));
        goto finish;
    }
    ret = sd_bus_add_object_vtable(bus, &slot, OBJECT_PATH, INTERFACE_NAME, service_vtable, &count);
    if (ret < 0) {
        fprintf(stderr, "Failed to add object interface: %s\n", strerror(-ret));
        goto finish;
    }
    ret = sd_bus_request_name(bus, BUS_NAME, 0);
    if (ret < 0) {
        fprintf(stderr, "Failed to acquire name: %s\n", strerror(-ret));
        goto finish;
    }
    printf("--- DBUS 서버 시작 (%s) ---\n", BUS_NAME);
    printf("알림 (Tick Signal)을 1초마다 전송합니다.\n");

    // --- 메인 루프: 알림 송신 및 버스 메시지 처리 ---
    while (1) {
        ret = sd_bus_process(bus, NULL);
        if (ret < 0) break;
        if (ret == 0) {
            sd_bus_wait(bus, (uint64_t)(1000 * 1000)); // 1초 대기
        }
        tick_count++;
        ret = sd_bus_emit_signal(bus, OBJECT_PATH, INTERFACE_NAME, "Tick", "u", tick_count);
        if (ret < 0) {
            fprintf(stderr, "Failed to emit signal: %s\n", strerror(-ret));
            break;
        }
        fprintf(stderr, "[Server] Tick Sent (Count=%u)\n", tick_count);
    }
finish:
    sd_bus_slot_unref(slot);
    sd_bus_unref(bus);
    return ret < 0 ? 1 : 0;
}
```

D-BUS를 이용한 클라이언트 코드

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <systemd/sd-bus.h>

#define BUS_NAME "org.example.NotificationService"
#define OBJECT_PATH "/org/example/NotificationService"
#define INTERFACE_NAME "org.example.NotificationService"

// 1. Tick 시그널이 도착했을 때 호출될 핸들러 함수
static int on_tick_signal(sd_bus_message *m, void *userdata, sd_bus_error *ret_error)
{
    uint32_t count = 0;
    int ret;

    ret = sd_bus_message_read(m, "u", &count);
    if (ret < 0) return ret;

    printf("[Client] 알림 수신: Tick (Count=%u)\n", count);
    return 0;
}
int main(void)
{
    sd_bus_error error = SD_BUS_ERROR_NULL;
    sd_bus_message *m = NULL;
    sd_bus *bus = NULL;
    sd_bus_slot *sigslot = NULL;
    const char *reply_string;
    uint32_t property_count = 0;
    int ret;

    ret = sd_bus_default_user(&bus);
    if (ret < 0) {
        fprintf(stderr, "Failed to connect to dbus-daemon: %s\n", strerror(-ret));
        goto finish;
    }
    printf("--- DBUS 클라이언트 시작 ---\n");

    // 2. Hello 메서드 호출
    ret = sd_bus_call_method(bus, BUS_NAME, OBJECT_PATH, INTERFACE_NAME, "Hello",
                             &error, &m, "s", "Client A");
    if (ret < 0) {
        fprintf(stderr, "Failed to issue Hello method call: %s\n", error.message);
        goto finish;
    }

    // 3. 응답 메시지를 읽어서 return 값 획득
    ret = sd_bus_message_read(m, "s", &reply_string);
    if (ret < 0) goto finish;
    printf("[Client] 서버 응답: %s\n", reply_string);
```

```
    // 4. Count 속성 획득
    ret = sd_bus_get_property_trivial(bus, BUS_NAME, OBJECT_PATH, INTERFACE_NAME,
                                      "Count", &error, 'u', &property_count);
    if (ret < 0) goto finish;
    printf("[Client] 서버 속성 확인: Count = %u\n", property_count);

    // 5. Tick Signal 수신을 위한 매치 등록
    ret = sd_bus_match_signal(bus, &sigslot, BUS_NAME, OBJECT_PATH, INTERFACE_NAME, "Tick",
on_tick_signal, NULL);
    if (ret < 0) {
        fprintf(stderr, "Failed to add match: %s\n", strerror(-ret));
        goto finish;
    }
    printf("Tick 알림을 수신 대기합니다 (10초)...\n");

    // 6. 10초 동안 알림 메시지 대기 루프
    for (int i = 0; i < 10; i++) { // 1초씩 10번 대기
        ret = sd_bus_process(bus, NULL);
        if (ret < 0) break;
        if (ret == 0) {
            // 메시지가 없으면 1초 대기
            sd_bus_wait(bus, (uint64_t)(1000 * 1000));
        }
    }

finish:
    sd_bus_error_free(&error);
    sd_bus_message_unref(m);
    sd_bus_unref(bus);
    return ret < 0 ? 1 : 0;
}
```

컴파일 -> systemd 라이브러리를 사용하므로, 컴파일 시 반드시 -lsystemd 옵션

```
jinho@localhost:~/lab2/12_db$ gcc -o dbusserver dbusserver.c -lsystemd
jinho@localhost:~/lab2/12_db$ gcc -o dbusclient dbusclient.c -lsystemd
```

서버에서 ./dbusserver를 실행하여 알림을 1초마다 전송

```
jinho@localhost:~/lab2/12_db$ ./dbusserver
--- DBUS 서버 시작 (org.example.NotificationService) ---
알림 (Tick Signal)을 1초마다 전송합니다.
[Server] Tick Sent (Count=1)
[Server] Tick Sent (Count=2)
[Server] Tick Sent (Count=3)
[Server] Tick Sent (Count=4)
[Server] Tick Sent (Count=5)
[Server] Tick Sent (Count=6)
[Server] Tick Sent (Count=7)
[Server] Tick Sent (Count=8)
[Server] Tick Sent (Count=9)
[Server] Tick Sent (Count=10)
```

클라이언트에서 ./dbusclient를 실행

```
jinho@localhost:~/lab2/12_db$ ./dbusclient
--- DBUS 클라이언트 시작 ---
[Client] 서버 응답: Hello Client A! Server has been calle
d 1 times.
[Client] 서버 속성 확인: Count = 1
Tick 알림을 수신 대기합니다 (10초)...
[Client] 알림 수신: Tick (Count=8)
[Client] 알림 수신: Tick (Count=9)
[Client] 알림 수신: Tick (Count=10)
```

-> 접속 시 서버의 Hello 호출 횟수와 현재 Tick 번호(Count)를 실시간으로 확인하여, 서버의 현재 알림 상태부터 이어서 수신하는 것을 보여줍니다. 즉, 현재 8,9,10을 출력중일 때 클라이언트가 접속하면 동기화 되어 8,9,10을 출력

검토

이번 실습을 통해 운영체제에서 파일 I/O, 프로세스 통신, 자원 동기화가 어떤 방식으로 이루어지는지 체계적으로 이해할 수 있었다. 특히 mmap을 활용한 고속 파일 복사 실습에서는 기존의 read/write 기반 I/O가 가지는 다중 복사 오버헤드를 줄이기 위해, 파일을 프로세스 주소 공간에 직접 매핑하는 접근 방식이 사용된다는 점을 확인할 수 있었다. 이러한 메모리 매핑 방식은 대용량 파일 처리나 고성능 시스템에서 자주 활용되는 효율적인 I/O 기법임을 실감했다.

또한 파이프, FIFO, 메시지 큐를 이용한 IPC 실습을 통해 프로세스 간 통신의 다양한 특성과 용도를 비교할 수 있었다. 파이프는 부모-자식 간 단방향 통신에 적합하고, FIFO는 서로 독립적인 프로세스 사이에서도 사용할 수 있다는 점에서 유용했다. 메시지 큐는 구조화된 데이터 교환이 가능하지만, 송수신이 큐의 상태에 따라 블로킹된다는 특성이 있어 요청-응답 구조에 적합하다는 점을 확인하였다. 이러한 IPC 기법들은 복잡한 시스템에서 프로세스 간 데이터 교환 문제를 해결하는 도구로 사용할 수 있었다.

마지막으로 세마포어와 DBus 실습을 통해 운영체제가 여러 프로세스를 어떻게 조율하는지를 깊이 있게 이해할 수 있었다. 세마포어는 여러 프로세스가 동일 자원에 접근할 때 경쟁 상태가 발생하는 문제를 해결하고, DBus는 시스템 전반의 프로세스들이 상태 및 이벤트를 실시간으로 공유할 수 있게 해주는 고급 통신 방식임을 확인하였다. 이러한 동기화 및 통신 기법들은 실제 리눅스 데스크톱 환경, 서버 관리, IoT 시스템 등 다양한 분야에서 필수적으로 사용 되는 것을 알게 되었다.