9. 신호와 예외 처리

9.1 `signal()`, `sigaction()` 사용법

개요

Signal(시그널) 은 리눅스/유닉스에서 **비동기적인 이벤트 알림 메커니즘**이다.
→ 커널 또는 다른 프로세스가 **현재 프로세스에 특정 이벤트가 발생했음을 알릴 때** 사용한다.

주요 용도

- 프로세스 종료 요청 (ex: `SIGTERM`, `SIGKILL`)
- 사용자 인터럽트 (ex: `SIGINT` → Ctrl+C)
- 알람/타이머 이벤트 (ex: `SIGALRM`)
- 자식 프로세스 종료 알림 (ex: `SIGCHLD`)
- 사용자 정의 시그널 (ex: `SIGUSR1`, `SIGUSR2`)

기본 흐름

- 특정 시그널이 프로세스에 전달 → 커널이 프로세스의 시그널 핸들러 호출 → 지정 동작 수행.

1️⃣ Signal 종류 예시

Signal	번호	기본 동작	설명
SIGINT	2	Terminate	Ctrl+C 입력 시 발생
SIGTERM	15	Terminate	일반 종료 요청
SIGKILL	9	Terminate (강제)	강제 종료 (무시 불가)
SIGSTOP	19	Stop	프로세스 일시 중지 (무시 불가)
SIGCONT	18	Continue	중지된 프로세스 실행 재개
SIGALRM	14	Terminate	`alarm()` 함수에 의해 발생
SIGCHLD	17	Ignore	자식 프로세스 종료 시 부모에 전달
SIGUSR1	10	Terminate	사용자 정의 시그널 1
SIGUSR2	12	Terminate	사용자 정의 시그널 2

2️⃣ 시그널 처리 방법

1️⃣ Default handler

- 아무 설정 안 하면 시그널의 **기본 동작**이 실행됨.
  - 대부분 Terminate / Ignore.

2️⃣ Custom handler

- 프로그래머가 직접 핸들러 함수 지정 가능.
- 대표 API:
  - `signal()`
  - `sigaction()` (더 안전하고 기능 강력함 → 권장)

3️⃣ signal()

원형

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

handler 인자

- `handler` 함수는 다음과 같이 정의:

```
void handler(int signum);
```

- `signum` → 수신한 시그널 번호.

예제

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

void my_handler(int signum) {
    printf("Received signal %d\n", signum);
}

int main() {
    signal(SIGINT, my_handler);

    while (1) {
        printf("Running...\n");
        sleep(1);
    }

```

```
    return 0;
}
```

→ 실행 후 **Ctrl+C** 입력 시:

```
Running...
Running...
Received signal 2
```

4️⃣ sigaction()

원형

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

struct sigaction

```
struct sigaction {
    void     (*sa_handler)(int);         // 간단 핸들러 등록
    void     (*sa_sigaction)(int, siginfo_t *, void *); // 고급 핸들러
    sigset_t sa_mask;                    // 시그널 블록 마스크
    int      sa_flags;                   // 옵션 설정
    void     (*sa_restorer)(void);       // obsolete
};
```

→ 기본적으로 `sa_handler` 사용.

sa_flags 주요 옵션

옵션	설명
SA_RESTART	시그널 처리 후 자동으로 interrupted system call 재시도
SA_SIGINFO	고급 핸들러 사용 (`sa_sigaction`) 활성화

기본 사용 예제 (sa_handler)

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

void my_handler(int signum) {
    printf("SIGACTION: received signal %d\n", signum);
}

int main() {
```

```
    struct sigaction sa;
    sa.sa_handler = my_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;

    sigaction(SIGINT, &sa, NULL);

    while (1) {
        printf("Running...\n");
        sleep(1);
    }

    return 0;
}
```

→ 실행 후 **Ctrl+C** 입력 시:

```
Running...
Running...
SIGACTION: received signal 2
```

고급 핸들러 사용 예제 (sa_sigaction)

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

void my_handler(int signum, siginfo_t *info, void *context) {
    printf("SIGACTION(SIGINFO): received signal %d from PID %d\n", signum, info->si_pid);
}

int main() {
    struct sigaction sa;
    sa.sa_sigaction = my_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_SIGINFO;

    sigaction(SIGUSR1, &sa, NULL);

    while (1) {
        printf("Waiting for SIGUSR1...\n");
        sleep(1);
    }

    return 0;
}
```

```
$ ./siginfo_example
# 새 터미널에서
$ kill -SIGUSR1 <PID>
```

출력 예:

```
Waiting for SIGUSR1...
SIGACTION(SIGINFO): received signal 10 from PID 12345
```

---

5 signal() vs sigaction() 차이점

항목	signal()	sigaction()
API 단순성	간단	복잡 (유연성 제공)
재진입 처리	시스템마다 구현 차이 있음	SA_RESTART 옵션으로 제어 가능
고급 핸들러	미지원	`sa_sigaction` 지원
마스크 처리	미지원	지원 (`sa_mask`)
호환성	POSIX에서 deprecated 취급 → 과거 호환용	현재 표준, 권장 API

→ **실전에서는 항상 `sigaction()` 사용 권장**.

---

6 시그널 블록 마스크

- `sa_mask` 를 이용해서 **시그널 핸들러 실행 중 블록할 시그널 설정 가능**.

예:

```
sigemptyset(&sa.sa_mask);
sigaddset(&sa.sa_mask, SIGTERM);  // SIGINT 처리 중 SIGTERM 블록
```

→ SIGINT 핸들러 실행 중에는 SIGTERM이 블록됨 → 핸들러 종료 후 처리.

---

7 정리

API	기능	권장 사용 여부
signal()	간단한 시그널 핸들러 등록	❌ (구형 코드 호환용만)
sigaction()	안전하고 유연한 시그널 핸들러 등록	✅ (실전에서는 sigaction 사용)

핵심 패턴:

```c
struct sigaction sa;
sa.sa_handler = handler_func; // 또는 sa_sigaction
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART;  // or SA_SIGINFO
sigaction(signum, &sa, NULL);
```

결론

- 시그널은 **리눅스/유닉스 프로세스 제어의 기본 요소**다.
- 실전에서는:
  - `SIGCHLD` 처리 (fork() 이후 자식 프로세스 종료 감지)
  - `SIGUSR1`, `SIGUSR2` 를 이용한 프로세스 간 제어
  - `SIGTERM` 처리 → graceful shutdown 구현
  - `SIGALRM` 으로 타이머 기능 구현
- **sigaction() 를 표준으로 사용**하는 습관을 반드시 들여야 한다 → signal()은 의도치 않은 호환성 문제가 발생할 수 있다.

9.2 시그널 블로킹과 마스크

개요

시그널 블로킹(signal blocking) 은:

- 프로세스가 특정 시그널을 **일시적으로 수신하지 않도록 막는 기능**이다.
- → 블록된 동안 시그널은 **pending 상태로 유지** → 블록이 해제되면 처리됨.

주요 사용 이유

- **Critical section 보호** (중간에 시그널로 방해받으면 안 되는 코드 구간)
- **시그널 동기화 제어** (처리 타이밍 직접 제어)
- **일부 시그널 지연 처리** (특정 구간에서만 처리 허용)

1️⃣ 관련 API

함수	기능
sigprocmask()	현재 프로세스의 시그널 마스크 변경
pthread_sigmask()	현재 스레드의 시그널 마스크 변경 (멀티스레드 안전)
sigpending()	현재 블록된 상태에서 pending 된 시그널 확인
sigset_t 관련 함수	시그널 집합 조작 (sigemptyset, sigfillset, sigaddset, sigdelset)

---

2 시그널 집합(sigset_t) 조작

기본 패턴

```
sigset_t set;

sigemptyset(&set);              // 시그널 집합 초기화 (빈 집합)
sigaddset(&set, SIGINT);        // SIGINT 추가
sigaddset(&set, SIGTERM);       // SIGTERM 추가
sigdelset(&set, SIGTERM);       // SIGTERM 제거
sigfillset(&set);               // 모든 시그널 추가
```

→ **sigset_t** 는 시그널 번호들의 집합을 표현.

---

3 sigprocmask()

원형

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

매개변수	설명
how	SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK
set	적용할 시그널 집합
oldset	이전 시그널 마스크 저장 (NULL 가능)

동작 방식

how 값	동작
SIG_BLOCK	시그널 블록 추가
SIG_UNBLOCK	시그널 블록 해제
SIG_SETMASK	현재 시그널 마스크를 set로 교체

---

4 예제 1 기본 블로킹/해제

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

void handler(int signum) {
    printf("Received signal %d\n", signum);
}

```

```c
int main() {
    struct sigaction sa;
    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigaction(SIGINT, &sa, NULL);

    sigset_t block_set;
    sigemptyset(&block_set);
    sigaddset(&block_set, SIGINT);

    printf("Blocking SIGINT for 5 seconds...\n");
    sigprocmask(SIG_BLOCK, &block_set, NULL);

    sleep(5);

    printf("Unblocking SIGINT now.\n");
    sigprocmask(SIG_UNBLOCK, &block_set, NULL);

    // Wait to observe if SIGINT is delivered now
    while (1) {
        sleep(1);
    }

    return 0;
}
```

실행 흐름

```
$ ./sigblock_example
Blocking SIGINT for 5 seconds...
# Ctrl+C 눌러도 반응 없음 (SIGINT pending 상태)
Unblocking SIGINT now.
# 즉시 "Received signal 2" 출력됨
```

→ SIGINT는 pending 상태였다가 unblock 시 처리됨.

5️⃣ sigpending()

원형

```c
int sigpending(sigset_t *set);
```

- 현재 프로세스에 대해 pending 상태인 시그널 집합을 조회.

예제 2 pending 시그널 확인

```
sigset_t pending_set;
sigpending(&pending_set);

if (sigismember(&pending_set, SIGINT)) {
    printf("SIGINT is pending\n");
}
```

→ SIGINT가 블록 상태에서 발생한 경우 pending 상태로 유지됨 → 확인 가능.

---

6 pthread_sigmask()

원형

```
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset);
```

- **멀티스레드 프로세스에서 스레드별 시그널 마스크 설정 가능**.
- → 쓰레드별로 시그널 블록 상태를 별도로 설정해야 하는 경우 사용 (POSIX 표준).

실전 예

- main thread만 시그널 수신하도록 설정 → worker thread에서는 SIGINT 블록.

---

7 정리 패턴

시그널 블로킹 기본 패턴

```
sigset_t set;
sigemptyset(&set);
sigaddset(&set, SIGINT);
sigprocmask(SIG_BLOCK, &set, NULL);
```

시그널 블로킹 해제

```
sigprocmask(SIG_UNBLOCK, &set, NULL);
```

시그널 블로킹 전체 교체

```
sigprocmask(SIG_SETMASK, &set, NULL);
```

---

8 시그널 블로킹의 실전 활용

용도	활용 방식
Critical section 보호	시그널 블록 → 코드 실행 → unblock
시그널 동기적 처리	특정 시점에서만 unblock
멀티스레드 시그널 처리	main thread만 시그널 unblock 상태 유지, 나머지는 block
SIGCHLD 핸들러 설치 시	설치 전 SIGCHLD block → handler 등록 후 unblock (race condition 방지)

---

9 정리

API	기능	비고
sigprocmask()	프로세스 시그널 마스크 변경	일반 프로세스용
pthread_sigmask()	스레드별 시그널 마스크 변경	멀티스레드 환경에서 사용
sigpending()	pending 시그널 확인	debugging / sync 제어용
sigset_t 관련 함수	시그널 집합 구성/변경	필수 유틸 함수들

결론

- **시그널 블로킹과 마스크는 고급 시그널 처리에서 매우 중요한 기술**이다.
- 실전에서는:
  - **중요 코드 구간 보호**
  - **race condition 방지**
  - **멀티스레드 시그널 처리 제어**
- → 반드시 `sigaction()` + `sigprocmask()` / `pthread_sigmask()` 를 함께 쓸 줄 알아야 안정적인 시그널 기반 코드를 작성할 수 있다.

9.3 `alarm()`, `pause()`, `kill()`

개요

이 세 가지 함수는 **시그널 기반 프로세스 제어와 동기화에서 자주 사용하는 유틸리티 함수**다.

함수	기능
`alarm()`	일정 시간 후 `SIGALRM` 시그널 발생 예약
`pause()`	시그널을 기다리며 **현재 프로세스를 블록**
`kill()`	다른 프로세스(또는 자신)에게 시그널 전송

1️⃣ `alarm()`

원형

```
unsigned int alarm(unsigned int seconds);
```

설명

- `seconds` 후에 커널이 **SIGALRM 시그널을 현재 프로세스에 보냄**.
- `alarm(0)` 호출 시 **예약된 알람 취소**.
- 반환값: **기존 예약이 남아있던 초 수** (없으면 0).

사용 예

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

void alarm_handler(int signum) {
    printf("Alarm triggered! Received signal %d\n", signum);
}

int main() {
    signal(SIGALRM, alarm_handler);

    printf("Setting alarm for 3 seconds...\n");
    alarm(3);

    while (1) {
        pause();  // 시그널이 올 때까지 대기
    }

    return 0;
}
```

→ 실행 결과:

```
Setting alarm for 3 seconds...
Alarm triggered! Received signal 14
```

→ **SIGALRM = 시그널 번호 14**.

2️⃣ `pause()`

원형

```
int pause(void);
```

설명

- 프로세스를 **시그널이 수신될 때까지 블록**.
- 시그널 핸들러 실행 후 `pause()`가 반환됨 → 항상 -1 반환, `errno`에 `EINTR` 설정됨.

사용 예

- 일반적으로 **시그널만 기다리는 메인 루프**에 사용:

```
while (1) {
    pause();
}
```

→ 주의: 반드시 **시그널 핸들러 등록 후 사용**.

3️⃣ `kill()`

원형

```
#include <signal.h>

int kill(pid_t pid, int sig);
```

설명

- `pid` 프로세스에 `sig` 시그널을 전송.

pid 값	의미
> 0	해당 pid의 프로세스에 시그널 전송
0	현재 프로세스 그룹 전체에 시그널 전송
-1	권한 있는 모든 프로세스에 시그널 전송
< -1	특정 프로세스 그룹(-pgid)에 시그널 전송

주의

- **kill()는 프로세스를 "죽인다"는 뜻이 아님** → 시그널을 "보낸다"는 의미임.
- 실제 종료 여부는 시그널 핸들링에 따라 달라짐.
- → SIGKILL(9) 전송 시 무조건 종료.

예제 1 다른 프로세스에 SIGUSR1 보내기

```
$ kill -SIGUSR1 <PID>
```

→ C 코드로는:

```
kill(target_pid, SIGUSR1);
```

예제 2 자신에게 시그널 보내기

```
kill(getpid(), SIGUSR1);
```

→ 현재 프로세스에 시그널 전송 → 핸들러가 실행됨.

---

4 실전 예제: `alarm()` + `pause()` 조합

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

void alarm_handler(int signum) {
    printf("Alarm fired! SIGALRM received.\n");
}

int main() {
    signal(SIGALRM, alarm_handler);

    printf("Setting alarm for 5 seconds...\n");
    alarm(5);

    printf("Waiting for alarm...\n");
    pause();  // SIGALRM 올 때까지 대기

    printf("Alarm was handled, program will exit.\n");

    return 0;
}
```

실행 흐름

```
Setting alarm for 5 seconds...
Waiting for alarm...
(5초 후)
Alarm fired! SIGALRM received.
Alarm was handled, program will exit.
```

→ 매우 많이 쓰이는 **타이머 기반 구조의 기본 패턴**.

---

5 실전 활용

함수	주요 활용 사례
alarm()	타임아웃 구현, 프로세스 watchdog 타이머
pause()	이벤트 기반 메인 루프 구성
kill()	프로세스 간 제어 (start/stop/reload 등) 구현, SIGUSR1/2 활용

6 고급 활용 패턴

alarm()으로 타임아웃 구현 예제

```
alarm(10);  // 10초 후 timeout 처리 예정

ssize_t n = read(fd, buf, size);  // read가 블록됨

// read가 10초 내에 끝나지 않으면 SIGALRM 발생 → interrupted system call
```

kill()으로 자식 프로세스 제어

```
pid_t child_pid = fork();

if (child_pid == 0) {
    // child
    while (1) {
        pause();
    }
} else {
    sleep(2);
    kill(child_pid, SIGTERM);  // 자식 종료 요청
    wait(NULL);
}
```

7 정리

API	기능	주요 특징
alarm()	n초 후 SIGALRM 예약	간단한 타이머 구현에 유용
pause()	시그널 수신까지 블록	시그널 기반 메인 루프에서 유용
kill()	시그널 전송	프로세스 제어, IPC 대체로 자주 사용

결론

- `alarm()` + `pause()` + `kill()` 는 시그널 기반 프로세스 제어의 **기본 도구**다.
- 실전에서는:
  - **타임아웃 처리**
  - **이벤트 기반 루프 구성**
  - **프로세스 간 제어 / IPC 대체**
    등에 매우 많이 사용된다.
- → 반드시 **sigaction()** + `alarm()` + `pause()` 조합으로 안정적 패턴 구성하는 습관을 들이는 것이 좋다.

9.4 커스텀 시그널 핸들러 작성

개요

커스텀 시그널 핸들러는 특정 시그널을 수신했을 때 **프로그램이 원하는 동작을 직접 구현**하는 기능이다.

활용 예:

- **SIGINT (Ctrl+C) 무시 또는 graceful shutdown 처리**
- **SIGUSR1 / SIGUSR2 로 유저 정의 이벤트 처리**
- **SIGTERM 으로 서버 종료 시 리소스 정리**
- **SIGCHLD 로 자식 프로세스 종료 감지**

핸들러는 다음과 같은 구조로 작성한다:

```
void handler(int signum);
```

or (고급):

```
void handler(int signum, siginfo_t *info, void *context);
```

1️⃣ sigaction 기반 커스텀 핸들러 작성

기본 패턴

```
struct sigaction sa;
sa.sa_handler = my_handler;  // 또는 sa_sigaction 사용
sigemptyset(&sa.sa_mask);    // 핸들러 실행 중 블록할 시그널 집합 (없으면 비움)
sa.sa_flags = SA_RESTART;    // 시스템 콜 자동 재시도

sigaction(SIGINT, &sa, NULL);
```

핸들러 함수 예

```
void my_handler(int signum) {
    printf("Received signal %d\n", signum);
}
```

2️⃣ 고급 핸들러 (siginfo_t 사용)

패턴

```
struct sigaction sa;
sa.sa_sigaction = my_handler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_SIGINFO;

sigaction(SIGUSR1, &sa, NULL);
```

핸들러 함수 예

```
void my_handler(int signum, siginfo_t *info, void *context) {
    printf("Received signal %d from PID %d\n", signum, info->si_pid);
}
```

→ 고급 핸들러는 다음 정보를 받을 수 있음:

정보	설명
signum	시그널 번호
info->si_pid	시그널을 보낸 프로세스의 PID
info->si_uid	시그널을 보낸 프로세스의 UID
기타 info 필드	시그널 타입별 추가 정보

3️⃣ 실전 예제 1️⃣ SIGINT graceful shutdown

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

volatile sig_atomic_t stop = 0;

void sigint_handler(int signum) {
    printf("SIGINT received, preparing to exit...\n");
    stop = 1;
}
```

```c

int main() {
    struct sigaction sa;
    sa.sa_handler = sigint_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    sigaction(SIGINT, &sa, NULL);

    while (!stop) {
        printf("Working...\n");
        sleep(1);
    }

    printf("Cleanup complete, exiting.\n");

    return 0;
}
```

흐름

- Ctrl+C 입력 시 **SIGINT** → stop flag 설정 → loop 종료 → cleanup 후 정상 종료.

4️⃣ 실전 예제 2️⃣ SIGUSR1 고급 핸들러

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

void sigusr1_handler(int signum, siginfo_t *info, void *context) {
    printf("SIGUSR1 received from PID %d, UID %d\n", info->si_pid, info->si_uid);
}

int main() {
    struct sigaction sa;
    sa.sa_sigaction = sigusr1_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_SIGINFO;
    sigaction(SIGUSR1, &sa, NULL);

    while (1) {
        printf("Waiting for SIGUSR1...\n");
        sleep(5);
    }

    return 0;
}
```

사용법

```
$ ./sigusr1_handler
# 다른 터미널에서:
$ kill -SIGUSR1 <PID>
```

출력 예:

```
SIGUSR1 received from PID 12345, UID 1000
```

---

5 커스텀 시그널 핸들러 작성 시 주의사항

재진입 안전성 (reentrancy)

- 시그널 핸들러 내에서는 **비동기 안전 함수만 사용해야 함**.
- 예를 들어 `printf()`는 technically 비동기 안전이 아님 → 실전에서는 `write()` 사용 권장.

```
const char msg[] = "SIGINT received\n";
write(STDOUT_FILENO, msg, sizeof(msg) - 1);
```

허용되는 안전 함수 목록:

→ man 7 signal → "Async-signal-safe functions" 섹션 참조.

글로벌 상태 사용

- 핸들러 내에서는 **volatile sig_atomic_t 변수 사용 권장**.

```
volatile sig_atomic_t stop = 0;
```

- 멀티스레드 프로그램에서는 주의 → 시그널과 스레드 동기화가 매우 중요해짐 → `pthread_sigmask()` 사용 병행 추천.

---

6 커스텀 핸들러 주요 활용 예시

시그널	커스텀 핸들러 활용 사례
SIGINT	Ctrl+C 시 graceful shutdown 구현
SIGTERM	프로세스 종료 시 리소스 정리 (서버, daemon 등)
SIGCHLD	자식 프로세스 종료 감지 후 wait() 호출
SIGUSR1 / SIGUSR2	프로세스 간 사용자 정의 이벤트 처리 (IPC 대체 패턴)
SIGALRM	타이머 기반 처리

7️⃣ 정리

시그널 처리 방법	권장 사용
signal()	❌ (호환성용)
sigaction(sa_handler)	기본 핸들링 시 사용
sigaction(sa_sigaction)	고급 핸들링 시 사용 (추천)

핸들러 작성 시 주의:

- **비동기 안전 함수만 사용**.
- **Global flag** 로 동작 전달.
- 멀티스레드에서는 `pthread_sigmask()` 사용 고려.

결론

- 커스텀 시그널 핸들러는 **리눅스 시스템 프로그래밍의 필수 패턴**이다.
- 실전에서는 반드시 **sigaction() + 고급 핸들러(sa_sigaction)** 사용법까지 숙지하는 것이 좋다.
- 핸들러 작성 시 반드시:
  - 비동기 안전 함수 사용
  - 글로벌 flag 기반으로 안전하게 구현
  - 시그널 블로킹 마스크 적절히 설정 → **race condition 방지**.

9.5 시그널을 통한 상태 전이 구현

개요

상태 전이(State transition) 란:

- 시스템이 여러 상태 중 하나에 있으며,
- **이벤트(여기서는 시그널)가 발생하면 다른 상태로 전이**하는 모델을 말한다.

시그널 기반 상태 전이는:

- 시그널을 통해 프로세스 외부에서 **현재 상태를 변경하거나 동작을 제어**할 수 있게 해준다.

→ 이벤트 기반 서버, Daemon, Interactive process, Simulation 등에 많이 사용됨.

1️⃣ 기본 구성

상태 머신 설계

```
enum state {
    STATE_IDLE,
    STATE_RUNNING,
    STATE_PAUSED,
```

```
    STATE_TERMINATING
};
```

→ 글로벌 상태 변수로 현재 상태 저장:

```
volatile sig_atomic_t current_state = STATE_IDLE;
```

→ 각 시그널 → 특정 상태로 전이:

시그널	상태 전이
SIGUSR1	IDLE → RUNNING
SIGUSR2	RUNNING → PAUSED → RUNNING (토글)
SIGTERM	모든 상태 → TERMINATING

→ 이와 같은 매핑으로 커스텀 핸들러 작성.

---

2 시그널 핸들러 작성

핸들러 내부에서:

- **전역 상태 변수 변경**만 수행 (주의: 비동기 안전성 보장 필요).
- 복잡한 로직은 메인 루프에서 상태에 따라 수행.

---

3 실전 예제: 상태 전이 구현

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

enum state {
    STATE_IDLE,
    STATE_RUNNING,
    STATE_PAUSED,
    STATE_TERMINATING
};

volatile sig_atomic_t current_state = STATE_IDLE;

void signal_handler(int signum) {
    switch (signum) {
        case SIGUSR1:
            current_state = STATE_RUNNING;
            break;
        case SIGUSR2:
            if (current_state == STATE_RUNNING)
                current_state = STATE_PAUSED;
```

```
            else if (current_state == STATE_PAUSED)
                current_state = STATE_RUNNING;
            break;
        case SIGTERM:
            current_state = STATE_TERMINATING;
            break;
        default:
            break;
    }
}

void print_state() {
    switch (current_state) {
        case STATE_IDLE:
            printf("[STATE] IDLE\n");
            break;
        case STATE_RUNNING:
            printf("[STATE] RUNNING\n");
            break;
        case STATE_PAUSED:
            printf("[STATE] PAUSED\n");
            break;
        case STATE_TERMINATING:
            printf("[STATE] TERMINATING\n");
            break;
    }
}

int main() {
    struct sigaction sa;
    sa.sa_handler = signal_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;

    sigaction(SIGUSR1, &sa, NULL);
    sigaction(SIGUSR2, &sa, NULL);
    sigaction(SIGTERM, &sa, NULL);

    printf("PID: %d\n", getpid());
    printf("Send SIGUSR1 to start, SIGUSR2 to pause/resume, SIGTERM to terminate.\n");

    while (current_state != STATE_TERMINATING) {
        print_state();

        if (current_state == STATE_RUNNING) {
            // Simulated work
            printf("Doing work...\n");
            sleep(1);
        } else {
            sleep(1);
        }
    }

```

```
    printf("Exiting gracefully.\n");

    return 0;
}
```

---

4 사용법

```
$ ./state_machine
PID: 12345
Send SIGUSR1 to start, SIGUSR2 to pause/resume, SIGTERM to terminate.
```

→ 다른 터미널에서:

```
$ kill -SIGUSR1 12345
$ kill -SIGUSR2 12345
$ kill -SIGTERM 12345
```

흐름 예시 출력:

```
[STATE] IDLE
[STATE] IDLE
[STATE] RUNNING
Doing work...
Doing work...
[STATE] PAUSED
[STATE] PAUSED
[STATE] RUNNING
Doing work...
Doing work...
[STATE] TERMINATING
Exiting gracefully.
```

---

5 고급 활용 패턴

고급 상태 머신 설계

상태	허용 시그널	전이 가능 상태
IDLE	SIGUSR1	RUNNING
RUNNING	SIGUSR2, SIGTERM	PAUSED, TERMINATING
PAUSED	SIGUSR2, SIGTERM	RUNNING, TERMINATING
TERMINATING	N/A	-

구현 시 주의사항

- **핸들러 내에서는 반드시 단순한 상태 전이만 구현**.
- 상태 전이 후 실제 동작(파일 I/O, 동적 메모리 등)은 메인 루프에서 처리.
- 복잡한 상태 머신일 경우 → `switch-case` + transition table 사용 권장.

6️⃣ 정리

기술 요소	역할
sigaction() + 커스텀 핸들러	시그널 수신 시 상태 전이 구현
volatile sig_atomic_t 전역 상태 변수	현재 상태 표현
메인 루프	상태 기반으로 동작 수행
시그널 → 상태 매핑 표	명확하게 설계 필요

결론

- **시그널 기반 상태 전이 구현은 이벤트 기반 프로세스 설계의 핵심 패턴**이다.
- 실전에서는:
  - **서버 graceful reload/stop**
  - **데몬 프로세스 상태 변경**
  - **단순 IPC 대체** (시그널로 상태 전환)
- 반드시:
  - sigaction() + 글로벌 상태 설계
  - 핸들러 내 동작 최소화
  - 메인 루프와 상태 기반 논리 분리
    → 이 원칙을 지키는 것이 중요하다.

📎 실습

Ctrl+C 무시, 로그 저장 종료 시뮬레이터

이 예제는 **SIGINT 무시**, **SIGTERM 수신 시 로그 저장 및 정상 종료**를 목표로 한다.

✅ 요구사항

항목	설명
✅ SIGINT (Ctrl+C)	무시
✅ SIGTERM	로그 파일에 메시지 기록 후 종료

항목	설명
✅ 로그 파일	`shutdown.log` 에 저장
✅ 백그라운드 실행 고려	로그 저장을 눈에 보이게 출력 (tail 등 활용 가능)

전체 코드 예제

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <time.h>
#include <string.h>

#define LOGFILE "shutdown.log"

void sigint_handler(int signum) {
    // Ctrl+C 무시
    printf("SIGINT (Ctrl+C) ignored.\n");
}

void sigterm_handler(int signum) {
    FILE *log = fopen(LOGFILE, "a");
    if (log == NULL) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }

    // 현재 시각 저장
    time_t now = time(NULL);
    char *timestamp = ctime(&now); // 자동 개행 포함
    if (timestamp == NULL) {
        timestamp = "UNKNOWN TIME\n";
    }

    fprintf(log, "[TERMINATED] Received SIGTERM at %s", timestamp);
    fclose(log);

    printf("SIGTERM received, shutting down gracefully.\n");
    exit(0);
}

int main() {
    // SIGINT 무시
    struct sigaction sa_int;
    sa_int.sa_handler = sigint_handler;
    sigemptyset(&sa_int.sa_mask);
    sa_int.sa_flags = SA_RESTART;
    sigaction(SIGINT, &sa_int, NULL);

```

```
    // SIGTERM 처리
    struct sigaction sa_term;
    sa_term.sa_handler = sigterm_handler;
    sigemptyset(&sa_term.sa_mask);
    sa_term.sa_flags = SA_RESTART;
    sigaction(SIGTERM, &sa_term, NULL);

    printf("PID: %d\n", getpid());
    printf("Running... Try Ctrl+C or `kill -TERM %d`\n", getpid());

    // 무한 루프
    while (1) {
        printf("Still alive...\n");
        sleep(2);
    }

    return 0;
}
```

🧪 실습 방법

1. 컴파일

```
gcc -o ignore_sigint_logger ignore_sigint_logger.c
```

2. 실행

```
./ignore_sigint_logger
```

3. Ctrl+C 입력

```
^C
SIGINT (Ctrl+C) ignored.
```

4. SIGTERM 보내기 (다른 터미널에서)

```
kill -TERM <PID>
```

5. 로그 확인

```
cat shutdown.log
```

```
[TERMINATED] Received SIGTERM at Mon Jun 10 01:24:13 2025
```

🛠️ 고급 확장 아이디어

기능	설명
`SIGQUIT` 시 강제 종료	Ctrl+\ 눌렀을 때만 강제 종료 허용
`shutdown.log`에 추가 정보 기록	사용자 UID, PID 등
`syslog()`로 시스템 로그 연동	`/var/log/syslog`에 기록
백그라운드 데몬으로 실행	`fork()` 후 detach (추후 구현 가능)

🧩 정리

기술 요소	적용 내용
`sigaction()`	시그널 처리 설정
`SIGINT`	핸들러에서 무시 메시지 출력
`SIGTERM`	로그 파일 기록 + 정상 종료
`ctime()`	현재 시간 문자열 생성
`fopen()` / `fprintf()`	파일 기록

시그널 기반 IPC (사용자 정의 시그널)

개요

- **IPC(Inter-Process Communication)**: 프로세스 간에 정보를 주고받는 기술.
- 시그널 기반 IPC는 **시그널(SIGNAL)을 통해 프로세스 간에 이벤트/명령을 전달**하는 간단한 IPC 방식.
- 주로 **SIGUSR1 (10)** 와 **SIGUSR2 (12)** 를 사용자 정의 시그널로 사용.

특징

장점	단점
구현이 매우 간단	데이터 전달은 매우 제한적 (값 X, 상태만 전달)
프로세스간 동기화 가능	데이터량 전달 불가 (단, sigqueue 이용 시 약간 가능)
시스템 리소스 거의 없음	신뢰성 낮음 (시그널 유실 가능성 존재)

활용 사례

- 프로세스 상태 변경 요청
- **서버 wake-up, reload, pause 등 이벤트 알림**
- **간단한 프로세스 제어용 커맨드 인터페이스 구현**
- **데몬 관리** → `kill -USR1 pid` / `kill -USR2 pid`

1️⃣ SIGUSR1 / SIGUSR2 를 이용한 기본 IPC 예제

설계

- **서버 프로세스**: SIGUSR1, SIGUSR2 수신 → 상태 변경
- **클라이언트 프로세스**: `kill -USR1 pid` / `kill -USR2 pid` 로 이벤트 전송

2️⃣ 서버 코드 예제

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

volatile sig_atomic_t usr1_count = 0;
volatile sig_atomic_t usr2_count = 0;

void sigusr_handler(int signum) {
    if (signum == SIGUSR1) {
        usr1_count++;
    } else if (signum == SIGUSR2) {
        usr2_count++;
    }
}

int main() {
    struct sigaction sa;
    sa.sa_handler = sigusr_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;

    sigaction(SIGUSR1, &sa, NULL);
    sigaction(SIGUSR2, &sa, NULL);

    printf("Signal IPC server running. PID: %d\n", getpid());
    printf("Send SIGUSR1 or SIGUSR2 to this process.\n");

    while (1) {
        printf("SIGUSR1 count: %d, SIGUSR2 count: %d\n", usr1_count, usr2_count);
        sleep(2);
    }

```

```
    return 0;
}
```

3 클라이언트 사용법

```
$ ./signal_ipc_server
Signal IPC server running. PID: 12345
Send SIGUSR1 or SIGUSR2 to this process.
SIGUSR1 count: 0, SIGUSR2 count: 0

# 다른 터미널에서:

$ kill -SIGUSR1 12345
$ kill -SIGUSR1 12345
$ kill -SIGUSR2 12345
```

서버 출력 변화 예:

```
SIGUSR1 count: 2, SIGUSR2 count: 1
```

4 고급: sigqueue() 로 데이터 전달

sigqueue() 를 이용하면 **정수 값 하나(4byte) 정도는 함께 전달 가능**.

API

```
int sigqueue(pid_t pid, int sig, const union sigval value);
```

- value.sival_int → int 값 전달 가능.

고급 서버 예제

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

void sigusr_handler(int signum, siginfo_t *info, void *context) {
    if (signum == SIGUSR1) {
        printf("Received SIGUSR1 with value: %d\n", info->si_value.sival_int);
    }
}

int main() {
    struct sigaction sa;
    sa.sa_sigaction = sigusr_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_SIGINFO;

```

```
    sigaction(SIGUSR1, &sa, NULL);

    printf("Signal IPC server with data. PID: %d\n", getpid());

    while (1) {
        pause();
    }

    return 0;
}
```

클라이언트 사용법 (C에서):

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

int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("Usage: %s <pid> <value>\n", argv[0]);
        return 1;
    }

    pid_t pid = atoi(argv[1]);
    int val = atoi(argv[2]);

    union sigval sval;
    sval.sival_int = val;

    sigqueue(pid, SIGUSR1, sval);

    return 0;
}
```

실행 흐름:

```
$ ./signal_ipc_server_with_data
Signal IPC server with data. PID: 12345

# 클라이언트에서:
$ ./signal_ipc_client 12345 999

# 서버에서 출력:
Received SIGUSR1 with value: 999
```

5️⃣ 정리

시그널 기반 IPC 방식	특징
SIGUSR1 / SIGUSR2	상태 전이, 이벤트 알림에 적합
sigqueue() + SIGUSR1	상태 전이 + 간단한 데이터 전달 가능
주의사항	시그널 유실 가능성 있음 (시그널 재전송 시 중복 가능성 주의)

주요 활용 영역:

- **서버 프로세스 제어 (reload / stop / pause 등)**
- **데몬 관리**
- **간단한 이벤트 알림 시스템**
- **Test, Debug 시 빠른 커맨드 인터페이스**

결론

- 시그널 기반 IPC는 매우 가볍고 빠르지만 **복잡한 데이터 교환용으로는 적합하지 않다**.
- 주로:
  - **프로세스 상태 전이**
  - **간단한 "wake up" 이벤트**
  - **정상 종료 요청**
  - **hot reload 등 명령어 인터페이스**
- 로 사용하면 유용하다.
- **sigqueue()** 를 통해 기본적인 정수 값은 함께 전달 가능 → 고급 핸들링 시 사용 가능.

타이머 기반 종료 조건 처리기

이번 예제는 **일정 시간(예: 10초) 후 SIGALRM 시그널을 발생시키고 → 커스텀 핸들러에서 종료 처리**하는 패턴이야.

주요 패턴

- `alarm()` 으로 N초 후 SIGALRM 예약
- `sigaction()` 으로 SIGALRM 핸들러 등록
- 핸들러에서 **종료 조건 처리** + 종료 로그 출력
- 메인 루프는 정상 동작 중 (alarm 발생하면 종료)

전체 코드 예제: 타이머 기반 종료 조건 처리기

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>

volatile sig_atomic_t stop = 0;

void sigalrm_handler(int signum) {
    // 타이머 종료 처리
    printf("Timer expired! Shutting down.\n");
    stop = 1;
}

void log_shutdown() {
    FILE *log = fopen("timer_shutdown.log", "a");
    if (log == NULL) {
        perror("fopen");
        return;
    }

    time_t now = time(NULL);
    char *timestamp = ctime(&now);
    if (timestamp == NULL) {
        timestamp = "UNKNOWN TIME\n";
    }

    fprintf(log, "[TIMER_SHUTDOWN] %s", timestamp);
    fclose(log);
}

int main() {
    struct sigaction sa;
    sa.sa_handler = sigalrm_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;

    sigaction(SIGALRM, &sa, NULL);

    printf("Program will automatically terminate after 10 seconds.\n");

    // 10초 후 SIGALRM 발생 예약
    alarm(10);

    // 메인 루프
    while (!stop) {
        printf("Running...\n");
        sleep(1);
    }

```

```
    // 종료 처리
    log_shutdown();
    printf("Shutdown complete. Exiting.\n");

    return 0;
}
```

🧪 실행 흐름

```
$ gcc -o timer_shutdown timer_shutdown.c
$ ./timer_shutdown
Program will automatically terminate after 10 seconds.
Running...
Running...
Running...
...
Timer expired! Shutting down.
Shutdown complete. Exiting.
```

→ 이후:

```
$ cat timer_shutdown.log
[TIMER_SHUTDOWN] Mon Jun 10 01:45:12 2025
```

📌 구조 설명

구성 요소	설명
`alarm(10)`	10초 후 SIGALRM 예약
`sigaction(SIGALRM)`	SIGALRM 수신 시 `sigalrm_handler()` 호출
`volatile sig_atomic_t stop`	안전한 종료 조건 변수
`log_shutdown()`	종료 시 타임스탬프 기록 (파일 기록)
메인 루프	stop flag가 1 되기 전까지 1초 단위 작업 수행

🛠️ 고급 확장 아이디어

기능	설명
SIGUSR1 수신 시 타이머 리셋	`alarm()` 재호출로 재타이머 가능
SIGUSR2 수신 시 타이머 취소	`alarm(0)` 호출
사용자 지정 종료 메시지 기록	SIGUSR1 등으로 종료 메시지 지정 후 log 기록

기능	설명
동적 타이머 시간 설정	CLI 인자로 N초 설정 받아서 alarm(N) 사용

🚀 실전에서 활용되는 패턴

- **Watchdog timer**: 서버/디바이스가 일정 시간 이상 응답 없을 시 자동 종료.
- **Idle timeout**: 서비스가 일정 시간 동안 activity 없으면 종료.
- **Test timeout**: 자동 테스트 시 일정 시간 이상 걸리면 timeout fail 처리.

결론

- `alarm()` + `sigaction()` + stop flag 조합은 **타이머 기반 종료 처리의 표준 패턴**이다.
- 반드시:
  - **핸들러는 단순화** (stop flag 설정만 담당)
  - 메인 루프에서 **정상 종료 경로를 구현** → "graceful shutdown" 가능.
- 실전에서는:
  - SIGUSR1/SIGUSR2로 **타이머 제어 인터페이스 추가** → 더 유연하게 만들 수 있음.