Лекція 11-12. Signaling – Part 1

1. The Signal Mechanism in Brief

Сигнали — це один із базових способів взаємодії між процесами (Inter-Process Communication, IPC) в Unix-подібних ОС. Це асинхронні повідомлення, які надсилаються процесам ядром або іншими процесами для повідомлення про події (наприклад, натискання клавіш, помилки тощо).

Ключові особливості:

- Сигнали мають числові імена (наприклад, SIGINT — сигнал переривання, зазвичай через Ctrl+C).
- Вони можуть надійти в будь-який момент, що робить їх асинхронними.
- Кожен сигнал має дію за замовчуванням (kill, ignore, stop, continue тощо).
- Процес може перехоплювати або ігнорувати деякі сигнали.

2. The Standard or Unix Signals

Unix визначає стандартний набір сигналів, який включає:

Назва	Номер	Опис
SIGINT	2	Переривання з клавіатури (Ctrl+C)
SIGTERM	15	Запит на завершення процесу
SIGKILL	9	Неможливо перехопити, примусове завершення
SIGSTOP	19	Призупинення виконання процесу
SIGCONT	18	Продовження виконання процесу

SIGCHLD	17	Надсилається батьківському процесу при завершенні дочірнього
SIGSEGV	11	Сегментаційна помилка

Примітка: сигнали SIGKILL і SIGSTOP не можна перехопити або ігнорувати.

3. Handling Signals

Обробка сигналів включає:

- Встановлення обробника сигналу (signal handler), який викликається, коли процес отримує сигнал.
- Використання функцій signal() або більш сучасної sigaction().

Простий приклад:

```
#include <signal.h>

#include <stdio.h>


void handle_sigint(int sig) {

   printf("Caught signal %d\n", sig);

}


int main() {

   signal(SIGINT, handle_sigint);

   while (1) {} // нескінченний цикл

}
```

Тут обробляється SIGINT — натискання Ctrl+C не завершить програму, а виведе повідомлення.

4. A Simple C Program that Handles a Couple of Signals

```
#include <signal.h>

#include <stdio.h>

#include <unistd.h>

void handle_sigint(int sig) {

   printf("Caught SIGINT (Ctrl+C)\n");

}

void handle_sigterm(int sig) {

   printf("Caught SIGTERM, exiting...\n");

   _exit(0);

}

int main() {

   signal(SIGINT, handle_sigint);

   signal(SIGTERM, handle_sigterm);

   printf("PID: %d\n", getpid());

   while (1) {

      sleep(1);

   }

}
```

Ця програма:

- Виводить свій PID.
- Реагує на SIGINT повідомленням.
- Завершується при отриманні SIGTERM.

5. Masking Signals

Іноді потрібно заблокувати (замаскувати) сигнали, щоб тимчасово відкласти їх обробку (наприклад, під час виконання критичної секції коду).

Для цього використовуються функції:

- sigprocmask()
- sigemptyset()
- sigaddset()

Приклад маскування:

```
sigset_t set;

sigemptyset(&set);

sigaddset(&set, SIGINT);

sigprocmask(SIG_BLOCK, &set, NULL); // блокуємо SIGINT
```

Сигнал буде доставлений лише після розблокування.

6. Reentrant Safety and Signalling

Оскільки обробники сигналів виконуються асинхронно, вони можуть перервати будь-яку функцію. Тому важливо:

- Уникати небезпечних (небезпечних для повторного входу) функцій в обробнику сигналів.
- Бажано використовувати лише асинхронно-безпечні функції, наприклад: _exit(), write(), signal().

Не можна використовувати: malloc(), printf(), printf() у більшості випадків, fork(), sleep(), strtok() тощо — вони не є thread-safe або reentrant.

7. Sigaction Flags

Функція sigaction() — більш контрольований спосіб установки обробників. Дає змогу:

- Встановити маску сигналів під час обробки.
- Вказати прапори (flags), що змінюють поведінку.

Основні поля:

```
struct sigaction {
    void     (*sa_handler)(int);
    sigset_t sa_mask;
    int      sa_flags;
};
```

Корисні прапори:

- SA_RESTART: автоматичне повторення системних викликів, які були перервані сигналом.
- SA_SIGINFO: дозволяє використовувати розширений обробник з додатковою інформацією.
- SA_NOCLDWAIT: не створювати зомбі для дочірніх процесів.
- SA_NODEFER: не блокувати сигнал під час його обробки.

8. No Zombies

Коли дочірній процес завершується, але батько не викликає wait(), дочірній стає зомбі (process in zombie state).

Щоб уникнути зомбі:

- Обробити сигнал SIGCHLD з SA_NOCLDWAIT.
- Або викликати waitpid() або wait() у батьківському процесі.

Приклад:

```
struct sigaction sa;
sa.sa_handler = SIG_IGN;
sa.sa_flags = SA_NOCLDWAIT;
sigaction(SIGCHLD, &sa, NULL);
```

9. Different Approaches to Handling Signals at High Volume

При великій кількості сигналів (наприклад, сотні SIGUSR1 за секунду), можуть виникнути проблеми:

- Сигнали не буферизуються — один тип сигналу може бути доставлений один раз, навіть якщо він був надісланий кілька разів.
- Можливі втрати сигналів.

Підходи до масштабної обробки:

- signalfd() — перетворює сигнал на файловий дескриптор (Linux-specific).
- sigqueue() — надсилає сигнал з додатковими даними (union sigval).
- RT-сигнали (SIGRTMIN..SIGRTMAX) — можуть бути черговими (queued) і зберігають порядок.
- Перехід до event-based моделей (наприклад, epoll/kqueue у поєднанні з signalfd).
- Сигнали — потужний, але потенційно небезпечний інструмент IPC.
- Важливо правильно маскувати сигнали, обирати безпечні функції та розуміти обмеження.
- У сучасних системах краще уникати сигналів для масового оповіщення — використовуються черги, дескриптори, події.

Signaling – Part 2

1. Gracefully Handling Process Crashes

Процеси можуть аварійно завершуватися через помилки, такі як:

- Сегментаційна помилка (SIGSEGV)
- Ділення на нуль (SIGFPE)
- Незаконна інструкція (SIGILL)
- Порушення доступу до пам’яті (SIGBUS)

Мета — не просто впасти, а зробити це контрольовано, наприклад:

- Зберегти лог або дамп регістрів

- Очистити ресурси (файли, сокети, пам’ять)

Підхід:

- Встановлення обробника для цих сигналів
- Використання sigaction() з прапором SA_SIGINFO для отримання додаткових даних

2. Trapping and Extracting Information from a Crash

Обробник сигналу з SA_SIGINFO отримує додаткову інформацію через siginfo_t:

Приклад:

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

void handler(int sig, siginfo_t *info, void *ucontext) {
    printf("Caught signal %d\n", sig);
    printf("Fault address: %p\n", info->si_addr);
    exit(1);
}
int main() {
    struct sigaction sa = {0};
    sa.sa_sigaction = handler;
    sa.sa_flags = SA_SIGINFO;
    sigaction(SIGSEGV, &sa, NULL);
    int *p = NULL;
    *p = 42; // викликає SIGSEGV
}
```

3. Register Dumping

Після аварії процесу можна отримати доступ до контексту процесора, зокрема до регістрів через ucontext_t.

Фрагмент:

```
#include <ucontext.h>

void handler(int sig, siginfo_t *info, void *context) {

   ucontext_t *uc = (ucontext_t *)context;

#if defined(__x86_64__)

   printf("RIP: %llx\n", (unsigned long long)uc->uc_mcontext.gregs[REG_RIP]);

#endif

}
```

Реєстри залежать від архітектури: REG_RIP, REG_EIP, REG_RSP, REG_RAX тощо.

4. Sleeping Correctly

Звичайні функції сну (sleep, usleep, nanosleep) можуть бути перервані сигналами.

Рішення:

- Повторно викликати nanosleep() з оновленим залишком
- Або використовувати sigsuspend()

Приклад:

```
struct timespec req = {1, 0}; // 1 сек

while (nanosleep(&req, &req) == -1 && errno == EINTR);
```

5. Real-Time Signals

Сигнали в діапазоні SIGRTMIN до SIGRTMAX мають такі переваги:

- Буферизовані (queued) — не втрачаються
- Можна надсилати разом з даними через sigqueue()

Надсилання RT-сигналу з даними:

```
union sigval val;
val.sival_int = 123;
sigqueue(pid, SIGRTMIN, val);
```

В обробнику:

```
void handler(int sig, siginfo_t *info, void *ctx) {
   printf("Received data: %d\n", info->si_value.sival_int);
}
```

6. Sending Signals

Сигнали можна надсилати з процесу або з shell:

Shell:

```
kill -SIGUSR1 <pid>
```

C API:

```
kill(pid, SIGTERM);
sigqueue(pid, SIGUSR1, val); // з даними
```

7. A Small Publisher-Subscriber Type of Application

У цьому прикладі:

- Підписник (subscriber) очікує сигнали
- Видавець (publisher) надсилає RT-сигнали з payload

Subscriber:

```
void handler(int sig, siginfo_t *info, void *ctx) {
    printf("Received %d from PID %d with data %d\n",
        sig, info->si_pid, info->si_value.sival_int);
}

int main() {
    struct sigaction sa = {0};
    sa.sa_flags = SA_SIGINFO;
    sa.sa_sigaction = handler;
    sigaction(SIGRTMIN, &sa, NULL);
    printf("PID: %d\n", getpid());
    while (1) pause();
}
```

Publisher:

```
int main(int argc, char *argv[]) {
    pid_t pid = atoi(argv[1]);
    union sigval val;
    val.sival_int = 42;
    sigqueue(pid, SIGRTMIN, val);
}
```

8. Alternative Signal-Handling Techniques

Замість signal()/sigaction() використовують:

- signalfd() — Linux-specific API, перетворює сигнал у файловий дескриптор
- eventfd()/epoll() — для інтеграції з event loop
- Threads + sigwaitinfo() — синхронне очікування сигналів у спеціальному потоці

9. The sigwaitinfo and the sigtimedwait System Calls

Дозволяють синхронно очікувати сигнал і отримати про нього дані (подібно до обробника, але не асинхронно).

Приклад:

```
sigset_t set;

sigemptyset(&set);

sigaddset(&set, SIGUSR1);

sigprocmask(SIG_BLOCK, &set, NULL);


siginfo_t info;

sigwaitinfo(&set, &info);


printf("Got signal %d from PID %d\n", info.si_signo, info.si_pid);
```

Переваги:

- Немає асинхронного виклику
- Безпечніше працювати з ресурсами

10. Source Code (Збірка компонентів)

```
// subscriber.c

#include <signal.h>

#include <stdio.h>

#include <unistd.h>

#include <string.h>


void handler(int sig, siginfo_t *info, void *ctx) {

   printf("Received %d from PID %d with value %d\n",

         sig, info->si_pid, info->si_value.sival_int);
```

```
}
int main() {
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_sigaction = handler;
    sa.sa_flags = SA_SIGINFO;
    sigaction(SIGRTMIN, &sa, NULL);
    printf("Subscriber PID: %d\n", getpid());
    while (1) pause();
}

// publisher.c
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
    if (argc < 2) return 1;
    pid_t pid = atoi(argv[1]);
    union sigval val;
    val.sival_int = 99;
    sigqueue(pid, SIGRTMIN, val);
}
```

Компіляція:

```
gcc subscriber.c -o subscriber
gcc publisher.c -o publisher
```

Завдання до практичних робіт

1. Напишіть програму, яка ловить сигнал SIGSEGV, зберігає fault address, поточний стек та PID у лог-файл, і намагається перезапустити себе.

2. Створіть демон, який відстежує процеси, що аварійно завершуються (через SIGFPE, SIGILL, SIGSEGV) та зберігає всю інформацію про контекст у базу даних SQLite.

3. Напишіть обробник SIGBUS, який розрізняє помилки через mmap та помилки доступу до фізичної пам’яті.

4. Реалізуйте систему логування, яка при кожному падінні програми виконує дамп усіх регістрів незалежно від архітектури.

5. Напишіть багатопоточну програму, яка виконує критичну обчислювальну задачу, і у випадку SIGSEGV відновлюється з останньої збереженої контрольної точки.

6. Розробіть програму, яка замінює стандартний механізм логування ядра при падінні процесу, використовуючи ptrace для збирання розширеної інформації.

7. Реалізуйте сигналобезпечну функцію обробки помилки, яка намагається коректно завершити роботу з усіма ресурсами, включаючи сокети, тимчасові файли та потокові буфери.

8. Напишіть програму, яка обробляє SIGINT або SIGTERM і використовує sigaltstack, щоб сигнал не завадив основному стеку.

9. Напишіть демон, який відстежує всі сигнали, що надходять до системи, і фіксує PID, UID, GID та ім’я процесу-відправника, використовуючи audit або netlink (якщо доступно).

10. Реалізуйте систему обміну повідомленнями між двома процесами, де кожне повідомлення кодується у sigval.sival_int, і процес отримує повідомлення через sigqueue.

11. Створіть додаток, де кожен потік відповідає за свій набір сигналів і використовує sigwaitinfo для синхронної обробки.

12. Розробіть систему публікації-підписки, де кілька підписників слухають сигнали з різними пріоритетами, використовуючи SIGRTMIN + N.

13. Побудуйте багатопроцесну модель, де головний процес розподіляє завдання за допомогою сигналів, а робочі повідомляють про завершення також через сигнали.

14. Напишіть програму, яка посилає реальні сигнали з таймером (timer_create) і аналізує, скільки з них реально оброблено, а скільки загублено.

15. Реалізуйте контролер, який реагує на сигнали від кількох процесів, сортує події за часом отримання та будує часову лінію у вигляді графу.

16. Напишіть програму, яка за допомогою sigpending, sigprocmask, sigaction та sigwaitinfo відстежує, які сигнали було заблоковано, доставлено, але не оброблено.

17. Створіть задачу, в якій через sigaction з SA_RESETHAND можна лише один раз перехопити сигнал — як забезпечити повторний запуск обробника без переоголошення sigaction.

18. Реалізуйте затримку виконання через sigsuspend, яка одночасно дозволяє приймати деякі сигнали, але ігнорує інші — поясніть, як це безпечно використовувати у

багатопотоковому середовищі.

19. Реалізуйте модель обміну між двома процесами, де один надсилає сигнал SIGUSR1 з таймаутом, а інший повинен відповісти SIGUSR2 — що буде, якщо обидва сигнали приходять одночасно?

20. Напишіть програму, що у випадку SIGSEGV намагається проаналізувати стек і визначити, чи можливо «відкотитися» на безпечну точку виконання.

21. Створіть фреймворк, який дозволяє будь-якому додатку підключити власний логгер падіння (core-dump обробник), не впливаючи на основну логіку.

22. Розробіть механізм, який за допомогою signalfd інтегрує сигнали у epoll-петлю подій — протестуйте на високонавантаженому сервері.

23. Напишіть декілька воркерів, які працюють паралельно й обробляють сигнали, що надходять одночасно, і порівняйте продуктивність між sigaction, sigwaitinfo та signalfd.

24. Створіть програму, яка використовує SIGALRM або timer_create() для тайм-аутів, але гарантує, що ніколи не втратить сигнал навіть під навантаженням.

25. Реалізуйте сервер, який дозволяє клієнтам «підписатися» на сигнали та отримувати повідомлення через SIGRTMIN+n, тестуючи масштабованість (наприклад, 1000 клієнтів).

26. Напишіть програму, яка генерує одночасно десятки сигналів різних типів і аналізує, в якому порядку вони будуть оброблені — обґрунтуйте результат.

27. Змоделюйте ситуацію, де під час обробки одного сигналу надходить інший — дослідіть наслідки з увімкненим та вимкненим SA_NODEFER, поясніть, які ризики виникають.