Lista 2

tags: SO

Zadanie 1

Zadanie 1. Na podstawie rysunku 4.15 z §4.6 przedstaw **stany procesu** w systemie l
zdarzenia wyzwalające zmianę stanu. Które przejścia mogą być rezultatem działań pode
systemu operacyjnego, kod sterowników, proces użytkownika? Wyjaśnij różnice między
i **nieprzerywalnym**. Czy proces może **zablokować** lub **zignorować** sygnał «SIGKIL

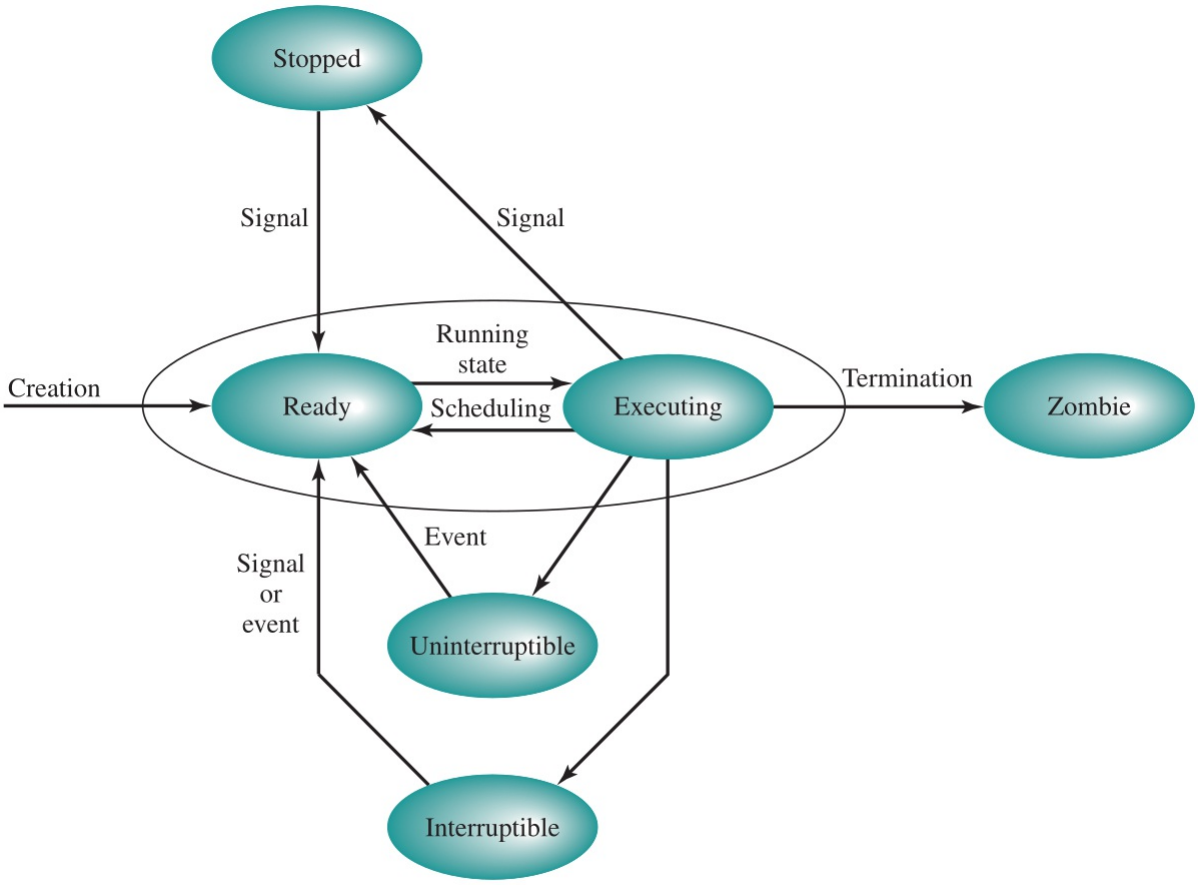

Figure 4.16 Linux Process/Thread Model

- **Running:** This state value corresponds to two states. A Running process is either executing or it is ready to execute.
- **Interruptible:** This is a blocked state, in which the process is waiting for an event, such as the end of an I/O operation, the availability of a resource, or a signal from another process.
- **Uninterruptible:** This is another blocked state. The difference between this and the Interruptible state is that in an Uninterruptible state, a process is waiting directly on hardware conditions and therefore will not handle any signals.
- **Stopped:** The process has been halted and can only resume by positive action from another process. For example, a process that is being debugged can be put into the Stopped state.
- **Zombie:** The process has been terminated but, for some reason, still must have its task structure in the process table.

::: warning

- **running** – proces jest wykonywany lub gotowy do wykonania.
- **interruptible** – (**sen przerywalny**) stan zablokowany. Proces czeka na zdarzenie (*event*), koniec operacji I/O, dostępność zasobu albo sygnał od innego procesu.
- **uninterruptible** – (**sen nieprzerywalny**) stan zablokowany. Podobny do **interruptible** z tą różnicą, że tutaj proces oczekuje bezpośrednio na określone warunki nałożone przez sprzęt, np. na jakieś urządzenie, dlatego nie obsługuje żadnych sygnałów.
- **stopped** – proces został zatrzymany i może zostać wznowiony przez inny proces. Przykładem takie sytuacji jest debuggowanie procesu (zostaje on wtedy zatrzymany) lub CTRL+Z i `fg`.
- **zombie** – proces został zabity, ale w tablicy procesów (*process table*) wciąż znajduje się jego struktura (*task structure*). :::

Akcje wyzwalające zmianę stanu

- **jądro**
  - terminacja procesu (np. SIGSEGV)
  - przerwanie
- **proces użytkownika**
  - wyjście programu (`exit()`)

    - otrzymanie sygnału (np. SIGKILL)
- **kod sterownika**
    - zdarzenia lub sygnały I/O

Czy proces może zablokować lub zignorować `SIGKILL` lub `SIGSEGV`?

::: info

- **blokowanie sygnału** – program nie otrzymuje przychodzącego sygnału. Nieodebrany sygnał oczekujewtedy na odebranie (*pending*). Blokowanie sygnału odbywa się poprzez modyfikację maski procedurą `int sigprocmask(int how, const sigset_t *set, sigset *oldset)` .
- **ignorowanie sygnału** – program otrzymuje sygnał, ale nie podejmuje w związku z tym żadnej akcji. :::

Sygnału `SIGKILL` nie można zablokować, obsłużyć lub zignorować.

Sygnał `SIGSEGV` jest blokowalny, ale nie ma to większego sensu jako, że program nie będzie mógł się dalej wykonywać. (https://www.gnu.org/software/libc/manual/html_node/Program-Error-Signals.html)

Zadanie 2

Zadanie 2. Wyjaśnij różnice w tworzeniu procesów w systemie Linux (§10.3.3) i Win
przebieg najważniejszych akcji podejmowanych przez jądro w trakcie obsługi funkcji
Załóżmy, że system posiada wywołanie `spawn`, o takich samych argumentach jak exe
wywołań `fork` i `execve`, a realizuje takie samo zadanie. Dlaczego w takim przypadk
z dodaniem do powłoki obsługi **przekierowania** standardowego wejścia/wyjścia odpo
łączenia dowolnych procesów **potokami**?

Zadanie 3

Zadanie 3. Na podstawie dokumentacji `fork(2)` (§8.3) i `execve(2)` (§8.10) wymi
procesu, które są (a) dziedziczone przez proces potomny (b) przekazywane do nowego
do przestrzeni adresowej. Czemu przed wywołaniem `fork` należy opróżnić bufory bibliot
robi w trakcie wywołania `execve` z konfiguracją **zainstalowanych** procedur obsługi

::: success **Zasoby procesu dziedziczone przez proces potomny (fork)**

- deskryptory plików otwartych w rodzicu
- przestrzeń danych rodzica,
- stos rodzica,
- sterta rodzica,
- współdzielony segment `text`,
- real user ID, real group ID, effective user i group ID,
- pomocnicze group ID,
- controlling terminal (/dev/tty),
- process group ID,
- session ID,
- obecny katalog roboczy,
- root directory,
- maska file mode creation,
- signal mask i dyzpozycje
- środowisko,
- mapowanie pamięci,
- limity zasobów,
- flaga close-on-exec dla każdego deskryptora otwartego pliku. :::

:::success ***Zasoby procesu przekazywane do nowego programu załadowanego do przestrzeni adresowej (exec)***

- dekryptory plików bez flag close-on-exec
- PID, PPID,
- RUID, RGID,
- dodatkowe group ID,
- process group ID,
- session ID,
- controlling terminal,
- czas pozostały do alarmu,
- obecny katalog roboczy,
- root directory,
- maska file mode creation.
- blokady plików (file locks),
- process signal mask,
- pending signals,
- limity zasobów,
- nice value,
- tms_utime, tms_stime, tms_cutime, tms_cstime (czas użytkownika i systemu oraz czas użytkownika i systemu dla dziecka) :::

```
close-on-exec -- flaga dla deskryptora pliku zapewniająca, że deskryptor zostanie zamknięty po udanym execve
```

Czemu przed wywołaniem `fork` należy opróżnić bufory biblioteki `stdio(3)`

Standardowa biblioteka I/O umieszcza dane w buforze. Użycie jedej z jej procedur (np. `printf`) przed forkiem spowoduje kopiowanie buforu do dziecka. Jeśli bufor nie był wcześniej wyczyszczony (np. `fflush(stdout)`) to jego zawartość wypisze się ponownie przy forkowaniu.

Co robi jądro w trakcie wywołania `execve` z konfiguracją zainstalowanych procedur obsługi sygnałów?

Obsługa wszystkich sygnałów jest przywracana do konfiguracji domyślnej. Zarejestrowane handlery łapanych sygnałów zostają zastąpione domyślnymi.

Zadanie 4

Zadanie 4. Uruchom program «xeyes» po czym użyj na nim polecenia «kill
Który sygnał jest wysyłany domyślnie? Przy pomocy kombinacji klawiszy «CTRL+Z»
«SIGTSTP», a następnie wznów jego wykonanie. Przeprowadź inspekcję pliku «/proc/
maskę **sygnałów oczekujących** na dostarczenie. Pokaż jak będzie się zmieniać,
wstrzymanemu procesowi kolejno: «SIGUSR1», «SIGUSR2», «SIGHUP» i «SIGINT». C
pliku «status» dotyczące sygnałów? Który sygnał zostanie dostarczony jako pierwszy

```
xeyes kill $(pidof xeyes) kill -l $(echo $?)
```

```
xeyes pkill xeyes kill -l $(echo $?)
```

```
xeyes xkill kill -l $(echo $?)
```

::: warning

kill, pkill - SIGKILL xkill - SIGHUB (komenda służy do rozłączenia usługi X serwera od procesu)

wysyłanie wybranego sygnału (tutaj HUP) `pkill -HUP xeyes`

czytanie maski sygnałów oczekujących `cat /proc/$(pidof xeyes)/status | grep ShdPnd`

- `SIGUSR1(10)` , `SIGUSR2(12)` – sygnały definiowane przez użytkownika
- `SIGHUP(1)` – sygnalizuje śmierć procesu kontrolującego (hang up)
- `SIGINT(2)` – wysyłany przez ctrl+C :::

::: info

```
man proc
```

(szesnastkowe maski)

- SigPnd – oczekujące sygnały dla wątku
- ShdPnd – oczekujące sygnały dla całego procesu
- SigBlk – sygnały blokowane
- SigIgn – sygnały ingnorowane
- SigCgt – sygnały łapane :::

Zadanie 5

Zadanie 5. Na podstawie kodu źródłowy sinit.c[1] opowiedz jakie zadania pełni minima
sinit. Jakie akcje wykonuje pod wpływem wysyłania do niego sygnałów wymieniony
Do czego służą procedury sigprocmask(2) i sigwait(3)? W jaki sposób grzebie

```c
/* See LICENSE file for copyright and license details. */
#include <sys/types.h>
#include <sys/wait.h>

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

#define LEN(x) (sizeof (x) / sizeof *(x))
#define TIMEO 30

static void sigpoweroff(void);
static void sigreap(void);
static void sigreboot(void);
static void spawn(char *const []);

static struct {
    int sig;
    void (*handler)(void);
} sigmap[] = {
    { SIGUSR1, sigpoweroff },
    { SIGCHLD, sigreap     },
    { SIGALRM, sigreap     },
    { SIGINT,  sigreboot   },
};

#include "config.h"

static sigset_t set;

int
main(void)
{
    int sig;
    size_t i;

    // Najpierw upewniamy się, czy proces jest uruchomiony jako init.
    if (getpid() != 1)
        return 1;

    // Zmieniamy katalog na root
    chdir("/");

    // Wypełniamy zbiór sygnałów "set"
    sigfillset(&set);

    /*
    Blokujemy wszystkie sygnały ze zbioru "set" używając procedury "sigprocmask":

        int sigprocmask(int how, const sigset_t *set, sigset_t *oldset), gdzie
        > [how] decyduje o tym jak należy potraktować sygnały (wartości SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK)
        > [set] to zbiór sygnałów, którym ustawiane są flagi
        > [oldset] zmienna, do której zapisany jest poprzedni stan zbioru
```

```
        > [oldset] zmienna, do której zapisany jest poprzedni stan zbioru
    */
    sigprocmask(SIG_BLOCK, &set, NULL);

    /*
    Spawn najpierw forkuje proces a następnie wykonuje execvp na jego dziecku.
    Jej argumentem jest polecenie uruchamiające proces.
    */
    spawn(rcinitcmd);



    while (1) {

        // Ustawiamy zegar, który po TIMEO sekundach dostarcza SIGALRM
        alarm(TIMEO);

        /*
        Nasłuchujemy sygnałów określonych w "set" używając procedury "sigwait":

            int sigwait(const sigset_t *set, int *sig)
            > [set] zbiór sygnałów, które są wyczekiwane
            > [sig] zmienna, przez którą zwracamy złapany sygnał
        */
        sigwait(&set, &sig);

        /*
        Po otrzymaniu sygnału przeglądamy tablicę mapującą sygnały na ich odpowiednie handlery.
        Jeśli otrzymany sygnał znajduje się w tablicy to uruchamiamy jego handlera.
        */
        for (i = 0; i < LEN(sigmap); i++) {
            if (sigmap[i].sig == sig) {
                sigmap[i].handler();
                break;
            }
        }
    }
    /* not reachable */
    return 0;
}

// Wyłącza system poleceniem "rcpoweroffcmd"
static void
sigpoweroff(void)
{
    spawn(rcpoweroffcmd);
}

/* Grzebie dziecko

    pid_t waitpid(pid_t pid, int *wstatus, int options)
    > [pid] kryterium PID dla dzieci, na które chcemy oczekiwać (-1 oznacza, że akceptujemy wszystkie)
    > [wstatus] jeśli nie jest nullem to zapisujemy do wskazanego przez niego adresu informacje o stanie wyczekiwanego procesu
    > [options] przyjmuje pewne stałe, w szczególności dla WNOHANG wraca natychmiast, jeśli żadne dziecko się nie zakończyło.
*/
static void
sigreap(void)
{
    while (waitpid(-1, NULL, WNOHANG) > 0)
        ;
    alarm(TIMEO); // ?
}

// Restartuje system poleceniem "rcrebootcmd"
static void
sigreboot(void)
{
    spawn(rcrebootcmd);
}

static void
spawn(char *const argv[])
{
    switch (fork()) {
    case 0:
        sigprocmask(SIG_UNBLOCK, &set, NULL);
        setsid();
        execvp(argv[0], argv);
        perror("execvp");
        _exit(1);
    case -1:
        perror("fork");
    }
}
```

Zadanie 6

Ściągnij ze strony przedmiotu archiwum «so21_lista_2.tar.gz», następnie rozpakuj i zapoznaj si

UWAGA! Można modyfikować tylko te fragmenty programów, które zostały oznaczone w koment

Zadanie 6. Uzupełnij program «reaper.c» prezentujący powstawanie **sierot**. Proce **żniwiarza** (ang. *reaper*) przy użyciu prctl(2). Przy pomocy procedury «spawn» syna i wnuka. Następnie osieroć wnuka kończąc działanie syna. Uruchom podproce «ps», aby wskazać kto przygarnął sierotę – przykład poniżej (zwróć uwagę na nume

```
1   PID  PPID  PGRP STAT CMD
2 24886 24643 24886 S+   ./reaper (main)
3 24888 24886 24887 S    ./reaper (grandchild)
4 24889 24886 24886 R+   /usr/bin/ps -o pid,ppid,pgrp,stat,cmd
```

Po udanym eksperymencie należy zabić wnuka sygnałem «SIGINT», a następnie po r jego **kod wyjścia**. Wysłanie «SIGINT» do procesu głównego jest zabronione! Zauwa zna numeru pid wnuka. W rozwiązaniu należy wykorzystać setpgid(2), pause(2),

UWAGA! Użycie funkcji sleep(3) lub podobnych do właściwego uszeregowania procesów jest zab

```c
#include "include/csapp.h"

static pid_t spawn(void (*fn)(void)) {
  pid_t pid = Fork();
  if (pid == 0) {
    fn();
    printf("(%d) I'm done!\n", getpid());
    exit(EXIT_SUCCESS);
  }
  return pid;
}

static void grandchild(void) {
  printf("(%d) Waiting for signal!\n", getpid());

  /* TODO: Something is missing here! */
  pause();

  printf("(%d) Got the signal!\n", getpid());
}

static void child(void) {
  pid_t pid;

  /* TODO: Spawn a child! */
  Setpgid(getpid(), getpid());
  pid = spawn(grandchild);

  printf("(%d) Grandchild (%d) spawned!\n", getpid(), pid);
}

/* Runs command "ps -o pid,ppid,pgrp,stat,cmd" using execve(2). */
static void ps(void) {
  /* TODO: Something is missing here! */
  char *args[] = {"/bin/ps", "-o", "pid,ppid,pgrp,stat,cmd", NULL};
  execve(args[0], args, NULL);
}

int main(void) {
  /* TODO: Make yourself a reaper. */
#ifdef LINUX
  Prctl(PR_SET_CHILD_SUBREAPER, 1);
#endif
  printf("(%d) I'm a reaper now!\n", getpid());

  pid_t pid, pgrp;
  int status;

  /* TODO: Start child and grandchild, then kill child!
   * Remember that you need to kill all subprocesses before quit. */

  pid = spawn(child);
  Waitpid(pid, NULL, 0);
  pgrp = pid;

  Waitpid(spawn(ps), NULL, 0);

  Kill(-pgrp, SIGINT);
  Waitpid(-1, &status, 0);

  printf( "(%d) Grandchild exited with status %d!\n", getpid(), status);

  return EXIT_SUCCESS;
}
```

Zadanie 7

Zadanie 7. Uzupełnij program «cycle.c», w którym procesy grają w piłkę przy pom
Proces główny tworzy n dzieci. Każde z nich czeka na piłkę, a po jej odebraniu
starszego brata. Zauważ, że najstarszy brat nie zna swojego najmłodszego rodzeńs
więc należy go wciągnąć do gry! Niech tata rozpocznie grę rzucając piłkę do najmł
znudzi Ci się obserwowanie procesów grających w piłkę możesz nacisnąć «CTRL+
do całej rodziny. Możesz wprowadź do zabawy dodatkową piłkę wysyłając sygnał
«kill». Czy piłki ostatecznie skleją się w jedną? W rozwiązaniu należy wykorzy
`sigsuspend(2)` i `kill(2)`.

UWAGA! Użycie funkcji `sleep(3)` lub podobnych do właściwego uszeregowania procesów jest zab

```c
#include "include/csapp.h"

static void signal_handler(int signum, siginfo_t *info, void *data) {
  if (signum == SIGINT) {
    safe_printf("(%d) Screw you guys... I'm going home!\n", getpid());
    _exit(0);
  }
}

static void play(pid_t next, const sigset_t *set) {
  for (;;) {
    printf("(%d) Waiting for a ball!\n", getpid());

    /* TODO: Something is missing here! */
    sigsuspend(set);


    usleep((300 + random() % 400) * 1000);
    Kill(next, SIGUSR1);
    printf("(%d) Passing ball to (%d)!\n", getpid(), next);
  }
}

int main(int argc, char *argv[]) {
  if (argc != 2)
    app_error("Usage: %s [CHILDREN]", argv[0]);

  int children = atoi(argv[1]);

  if (children < 4 || children > 20)
    app_error("Give number of children in range from 4 to 20!");

  /* Register signal handler for SIGUSR1 */
  struct sigaction action = {.sa_sigaction = signal_handler};
  Sigaction(SIGINT, &action, NULL);
  Sigaction(SIGUSR1, &action, NULL);

  /* TODO: Start all processes and make them wait for the ball! */
  sigset_t blocked, set;
  sigaddset(&blocked, SIGUSR1);
  sigprocmask(SIG_BLOCK, &blocked, &set);


  pid_t pid, prev = getpid();
  for (int i = 0; i < children; i++) {
    if ((pid = Fork()) == 0)
      play(prev, &set);
    else
      prev = pid;
  }

  Kill(prev, SIGUSR1);
  play(prev, &set);


  return EXIT_SUCCESS;
}
```

Zadanie 8

Zadanie 8. Uzupełnij program «demand» o **procedurę obsługi sygnału** «SIGSEGV». demonstrować przechwytywanie **błędów stron**, których nie było w stanie obsłużyć

Obsługujemy zakres adresów od «ADDR_START» do «ADDR_END». Pod losowo wyb z podanego przedziału zostanie podpięta **pamięć wirtualna** w trybie tylko do odcz wygeneruje do zadanego przedziału adresów zapisy, które zakończą się naruszeniem

Po wyłapaniu sygnału «SIGSEGV», korzystając z procedur «mmap_page» i «mprote zmapuj brakującą stronę (błąd «SEGV_MAPERR») i odblokuj zapis do strony (błąd «S do adresów spoza ustalonego zakresu powinien skutkować zakończeniem programu właściwy kod wyjścia tak, jakby proces został zabity sygnałem!

```
1 ...
2 Fault at rip=0x55cb50d54389 accessing 0x10003fc0! Make page at 0x10003000 wr
3 Fault at rip=0x55cb50d54389 accessing 0x10007bb0! Map missing page at 0x1000
4 ...
5 Fault at rip=0x55cb50d5439c accessing 0x10010000! Address not mapped - termi
```

W procedurze obsługi sygnału można używać tylko procedur **wielobieżnych** (ang w podręczniku ich listę. Możesz wykorzystać procedurę «safe_printf», będącą ok Czemu można ją bezpiecznie wywołać w wnętrza «sigsegv_handler»?

Adres powodujący błąd strony i rodzaj błędu znajdziesz w argumencie «sigs «siginfo_t», który opisano w podręczniku sigaction(2). Wskaźnik instrukcji, k strony, można przeczytać ze struktury przechowującej kontekst procesora «uc->uc_m definicje znajdują się w pliku nagłówkowym «/usr/include/x86_64-linux-gnu/

```c
#include "include/csapp.h"

/* First address of handled region. */
#define ADDR_START ((void *)0x10000000)
/* Last address of handled region (not inclusive). */
#define ADDR_END ((void *)0x10010000)

static size_t pagesize;

/* Maps anonymouse page with `prot` access permissions at `addr` address. */
static void mmap_page(void *addr, int prot) {
  Mmap(addr, pagesize, prot, MAP_ANONYMOUS | MAP_PRIVATE | MAP_FIXED, -1, 0);
}

/* Changes protection bits to `prot` for page at `addr` address. */
static void mprotect_page(void *addr, int prot) {
  Mprotect(addr, pagesize, prot);
}

static void sigsegv_handler(int signum, siginfo_t *info, void *data) {
  ucontext_t *uc = data;
  intptr_t rip = uc->uc_mcontext.gregs[16];

  /* TODO: You need to get value of instruction pointer register from `uc`.
   * Print all useful data from `info` and quit in such a way that a shell
   * reports program has been terminated with SIGSEGV. */

  void* addr = info->si_addr;

  safe_printf("Fault at rip=%lx accessing %lx! ", (long unsigned int)rip, (long unsigned int)addr);

  if(ADDR_START > addr || addr >= ADDR_END){
    safe_printf("Address not mapped - terminating!\n");
    _exit(128 + SIGSEGV);
  }

  int errcode = info->si_code;
  void* page = addr - (intptr_t)addr % pagesize;

  if(errcode == SEGV_MAPERR) {
    safe_printf("Map missing page at %lx\n", (long unsigned int)page);
    mmap_page(page, PROT_READ);
  }

  if (errcode == SEGV_ACCERR){
    safe_printf("Make page at %lx writable.\n", (long unsigned int)page);
    mprotect_page(page, PROT_WRITE);
  }
}

int main(int argc, char **argv) {
  pagesize = sysconf(_SC_PAGESIZE);

  /* Register signal handler for SIGSEGV */
  struct sigaction action = {.sa_sigaction = sigsegv_handler,
                             .sa_flags = SA_SIGINFO};
  sigaction(SIGSEGV, &action, NULL);

  /* Initially all pages in the range are either not mapped or readonly! */
  for (void *addr = ADDR_START; addr < ADDR_END; addr += pagesize)
    if (random() % 2)
      mmap_page(addr, PROT_READ);

  /* Generate lots of writes to the region. */
  volatile long *array = ADDR_START;
  long nelems = (ADDR_END - ADDR_START) / sizeof(long);

  for (long i = 0; i < nelems * 2; i++) {
    long index = random() % nelems;
    array[index] = (long)&array[index];
  }

  /* Perform off by one access - triggering a real fault! */
  array[nelems] = 0xDEADC0DE;

  return EXIT_SUCCESS;
}
```