SOPY NOTATKI

Zabijanie Zombie, w każdym kodzie - sigchld handler

```
void sigchld_handler(int sig)
{
   pid_t pid;
   while (1)
   {
      pid = waitpid(0, NULL, WNOHANG);
      if (pid == 0)
         return;
      if (pid <= 0)
      {
         if (errno == ECHILD)
            return;
         ERR("waitpid");
      }
   }
}
```

Każdy kod musi posiadać, to jeżeli forkujemy

Wysyłanie/Odbieranie sygnałów

potrzebny jest sigchld_handler, ponadto dla wysyłanych sygnałów np. przez dziecko, potrzebne jest handlowanie tymi sygnałami (SIGUSR1).

1. w main definujemy handlowanie przez rodzica

```
sethandler(sigchld_handler, SIGCHLD);
sethandler(SIG_IGN, SIGUSR1);
sethandler(SIG_IGN, SIGUSR2)
```

2. w create_children, własne handlowanie

```
sethandler(sig_handler, SIGUSR1);
sethandler(sig_handler, SIGUSR2);
```

ponadto definujemy własne sig_handler

```
void sig_handler(int sig)
{
   printf("[%d] received signal %d\n", getpid(), sig);
   last_signal = sig;
}
```

3. Daj maskę sigprocmask, dla tych sygnałów, to zablokuje sygnały

```
sigset_t mask, oldmask;
sigemptyset(&mask);
sigaddset(&mask, SIGUSR1);
sigaddset(&mask, SIGUSR2);
sigprocmask(SIG_BLOCK, &mask, &oldmask);
```

4. Wówczas w parent/child_work zależy, który odbiera, powinieneś, na chiwle wyłączać maske i czekać na sygnał.

```
while (last_signal != SIGUSR2)
   sigsuspend(&oldmask);
```

Wówczas, gdybyś czekał z sleepem()/albo wczytywał coś, to sygnały które przyjdą wtedy nie przerwą tego

5. Nie zawsze musi wyjść z pętli, może się skleić z SIGSUR1, wtedy najlepiej count++ w odzielnym handlerze dla SIGUSR2 i dodatkowa zmienna globalna.

```
while (last_signal != SIGUSR2)
   sigsuspend(&oldmask);
count++
```

5. Korzystaj z nanosleep, nie z sleep, bo SIGALARM posiada sleep!!
4. parent_work, child_work, jeden wysyła drugi odbiera
5. W jaki sposób zabijać proces potomny, wyslij mi sygnał, który on ma ustawiony do handlowania na default (to może sie nie udać, dlatego wyślij sygnał, inny od tego używanego do komunikacji).
6. Jeżeli funkcje open/read robimy kiedy jaki proces wysyła sygnały, to sigsuspend nie działa. Proponowane, jest to, żeby dodać funkcje bulk_read, bulk_write, które będą wypełałniały bufor. Makro TEMP_FAILURE_RETRY, pozwala, że w sytuajci kiedy open jest przerwany przez sygnał, to można dalej kontynuać wczytywanie na tych samych parametrach, aż do przeczytania wszystkiego (kiedy np. open przerwany to errno = EINTR, wówczas makro przywraca działanie, jeżeli errno ==błąd to nie)
7. Zaimplementuj z kodu na stronie.
8. SA_RESTART ma pomagać z EINTR, ale wówczas musielibyśmy zdefiniować wszystkie sygnały, na które uważać, a tego przewidzieć nie możemy.

TASK1

```
#define _GNU_SOURCE
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <time.h>
#include <unistd.h>
#include <signal.h>

#define ERR(source) \
   (fprintf(stderr, "%s:%d\n", __FILE__, __LINE__), perror(source), kill(0, SIGKIL
L), exit(EXIT_FAILURE))
```

```
volatile sig_atomic_t last_signal;
volatile sig_atomic_t got_signal;
volatile sig_atomic_t count_signal = 0;

void sigchld_handler(int sig)
{
   pid_t pid;
   while (1)
   {
      pid = waitpid(0, NULL, WNOHANG);

      if (pid == 0)
         return;

      if (pid <= 0)
      {
         if (errno == ECHILD)
            return;
         ERR("waitpid");
      }
   }
}

void sethandler(void (*f)(int), int sigNo)
{
   struct sigaction act;
   memset(&act, 0, sizeof(struct sigaction));
   act.sa_handler = f;

   if (-1 == sigaction(sigNo, &act, NULL))
      ERR("sigaction");
}

void sig_handler(int sig)
{
   last_signal = sig;
```

```
}

void sigusr1_handler(int sig)
{
   got_signal = 1;
   count_signal ++;
}

ssize_t bulk_write(int fd, char *buf, size_t count)
{
   ssize_t c;
   ssize_t len = 0;

   do
   {
      c = TEMP_FAILURE_RETRY(write(fd, buf, count));
      if (c < 0)
         return c;

      buf += c;
      len += c;
      count -= c;
   } while (count > 0);

   return len;
}

void child_work(int n, int s)
{
   alarm(1);
   sigset_t mask, oldmask, suspend_mask;

   sigemptyset(&mask);
   sigaddset(&mask, SIGUSR1);

   if (sigprocmask(SIG_BLOCK, &mask, &oldmask) < 0)
```

```
    ERR("sigprocmask");

suspend_mask = oldmask;
sigdelset(&suspend_mask, SIGUSR1);

int out;
char *buf = malloc(s);
if (!buf)
    ERR("malloc");

char digit_char = '0' + n;
/* Sprawdzenie poprawności cyfry dla bezpieczeństwa */
if (n < 0 || n > 9) digit_char = '?';

for (int i = 0; i < s; i++)
    buf[i] = digit_char;

char filename[32];
snprintf(filename, sizeof(filename), "%d.txt", getpid());
if ((out = TEMP_FAILURE_RETRY(open(filename,
            O_WRONLY | O_CREAT | O_TRUNC | O_APPEND, 0777))) < 0)
    ERR("open");

while (last_signal != SIGALRM)
{
    while (!got_signal && last_signal != SIGALRM)
        sigsuspend(&suspend_mask);

    if (last_signal == SIGALRM)
        break;

    got_signal = 0;
    for(int i =0 ; i< count_signal; i++){
        if (bulk_write(out, buf, s) < 0)
            ERR("write");
    }
```

```
        count_signal = 0;
    }

    if (close(out))
        ERR("close");
    sigprocmask(SIG_UNBLOCK, &mask, &oldmask);
    free(buf);
}

/* ZMIANA 1: Funkcja przyjmuje argv, losowanie i parsowanie n wewnątrz pętli
*/
void create_children(int child_count, char **argv)
{
    srand(getpid()); // Ziarno dla rodzica

    for (int i = 0; i < child_count; i++)
    {
        int s = 10 + rand() % 91;    // Każde dziecko ma losowy rozmiar
        int n = atoi(argv[i + 1]);   // Każde dziecko bierze swój parametr

        switch (fork())
        {
            case 0:
                // Potomek powinien przelosować ziarno, żeby uniezależnić się od r
odzica
                srand(getpid());
                sethandler(sigusr1_handler, SIGUSR1);
                sethandler(sig_handler, SIGALRM);
                child_work(n, s);
                exit(EXIT_SUCCESS);
            case -1:
                perror("Fork:");
                exit(EXIT_FAILURE);
        }
    }
}
```

```
/* ZMIANA 2: Pętla ograniczona do 100 iteracji (1 sekunda) */
void parent_work()
{
    sethandler(sigchld_handler, SIGCHLD);
    struct timespec tk = {0, 10000000}; // 10ms

    // 100 * 10ms = 1000ms = 1s
    for (int i = 0; i < 100; i++)
    {
        nanosleep(&tk, NULL);
        if (kill(0, SIGUSR1) < 0)
            ERR("kill");
    }
}

void usage(char *name)
{
    fprintf(stderr, "USAGE: %s n1 n2 ...\n", name);
    exit(EXIT_FAILURE);
}

int main(int argc, char **argv)
{
    // srand(time(NULL)); // Przeniesione/obsłużone w create_children
    int child_count;

    if (argc < 2)
        usage(argv[0]);

    child_count = argc - 1;

    sethandler(SIG_IGN, SIGUSR1);

    /* ZMIANA 3: Przekazujemy argv, usuwamy n = atoi(argv[1]) */
    create_children(child_count, argv);
```

```
    parent_work();

    while (wait(NULL) > 0)
    {
    }
    return EXIT_SUCCESS;
}
```

TASK2

```

```