# Systemy Operacyjne 1
# Laboratorium 2
# „Procesy i sygnały w Linuksie”
# (jeden tydzień)

dr inż. Arkadiusz Chrobot

6 października 2024

# Wstęp

W tej instrukcji zawarte są informacje na temat tworzenia i obsługiwania procesów przez system Linux (Unix). Pierwsza cześć zawiera opis budowy pamięci logicznej procesu użytkownika, druga zawiera opis tworzenia nowego procesu z punktu z punktu widzenia programisty aplikacji. Trzecia część przedstawia pojęcie sygnału i objaśnia jego zastosowania. Część czwarta zawiera skrócone opisy najważniejszych funkcji, które związane są z opisywanymi zagadnieniami. Instrukcja kończy się listą zadań do wykonania na laboratorium.

# 1. Budowa procesu w Linuksie

W systemach uniksowych (w tym w Linuksie) przestrzeń procesu[1] użytkownika można podzielić na dwa konteksty: kontekst użytkownika i kontekst jądra. Pierwszy z nich może być podzielony na sześć obszarów: tekstu, stałych, zmiennych zainicjowanych, zmiennych niezainicjowanych, sterty i stosu. Drugi zawiera wyłącznie dane. Obszar tekstu zawiera rozkazy maszynowe, które są wykonywane przez sprzęt. Ten obszar jest tylko do odczytu, a więc może go współdzielić kilka procesów równocześnie. Obszar stałych jest również tylko do odczytu. We współczesnych systemach uniksowych jest łączony w jeden obszar z obszarem tekstu. Obszar zmiennych zainicjowanych zawiera zmienne, którym zostały przypisane wartości początkowe, ale proces może je dowolnie modyfikować. Obszar zmiennych niezainicjowanych (bss) zawiera zmienne, które mają wartość początkową zero, a więc nie trzeba ich wartości inicjujących przechowywać w pliku programu. Sterta (ang. *heap*) i stos (ang. *stack*) tworzą w zasadzie jeden obszar - sterta służy do dynamicznego przydzielania dodatkowego obszaru w pamięci, natomiast na stosie przechowywane są ramki stosu, czyli informacje związane z wywołaniem podprogramów. Sterta rozszerza się w stronę wyższych adresów, natomiast stos w stronę niższych adresów. Proces użytkownika nie ma bezpośredniego dostępu do kontekstu jądra, który zawiera informacje o stanie tego procesu. Ten obszar może być modyfikowany tylko przez jądro. Pewne wartości w tym kontekście mogą być modyfikowane z poziomu procesu użytkownika poprzez odpowiednie wywołania systemowe.

# 2. Tworzenie nowych procesów

Proces, czyli wykonujący się program, może stworzyć proces potomny używając funkcji `fork()` udostępnianej z poziomu biblioteki standardowej języka C. W systemie Linux funkcja ta jest „opakowaniem“ na wywołanie `clone()`, które nie jest wywołaniem standardowym, tzn. nie jest dostępne w innych systemach kompatybilnych z Uniksem i nie należy go bezpośrednio stosować w programach, które mają być przenośne. W Linuksie zastosowany jest wariant tworzenia procesów określany po angielsku *copy-on-write*. Oznacza to, że po stworzeniu nowego procesu współdzieli on zarówno obszar tekstu, jak i obszar danych (tj. stertę, stos, zmienne zainicjowane i niezainicjowane) z rodzicem. Dopiero, kiedy któryś z nich będzie próbował dokonać modyfikacji danych nastąpi rozdzielenie obszaru danych (proces potomny otrzyma kopię obszaru rodziciela). Aby wykonać nowy program należy w procesie potomnym użyć jednej z funkcji `exec()`. Sterowanie z procesu potomnego do procesu rodzicielskiego nigdy bezpośrednio nie wraca, ale proces rodzicielski może poznać status wykonania procesu potomnego wykonując jedną z funkcji `wait()`. Jeśli proces rodzicielski nie wykona tej funkcji, to zakończony proces proces potomny zostaje procesem *zombie*. W przypadku, gdy proces-rodziciel zakończy się wcześniej niż proces potomny, to ten ostatni jest „adoptowany” przez proces `init`, którego PID (identyfikator procesu) wynosi `1` lub inne procesy należące do jego grupy procesu rodzicielskiego.

Listing 1 przedstawia prosty program, w którym za pomocą wywołania funkcji `fork()` tworzony jest proces potomny. Jeśli ta funkcja zwróci wartość `-1`, to znaczy, że nie udało się jej utworzyć potomka. Opis wyjątku, który spowodował to niepowodzenie możemy uzyskać na ekranie dzięki funkcji `perror()`. Jak jej argument wystarczy przekazać ciąg znaków będący nazwą funkcji, ale w przypadku gdy ta funkcja jest używana kilkukrotnie w obrębie, a nawet kilku plików, jeśli program jest podzielony na osobne jednostki kompilacji, to można jeszcze dodać do tej nazwy numer wiersza i nazwę pliku, w którym ta funkcja jest wywoływana. Po zakończeniu swojej pracy proces potomny kończy się wywołując funkcję `exec()`. Proces

[1] Proces, to program, który został uruchomiony i się wykonuje.

rodzicielski czeka na jego zakończenie wywołując funkcję `wait()`. Ponieważ funkcja `exec()` nie zwraca kodu zakończenia, to nie jest on badany, ale ta czynność wykonywana jest w przypadku funkcji `wait()`.

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>

void do_child_work(void)
{
        puts("Jestem potomkiem.");
        exit(0);
}

void do_parent_work(void)
{
        puts("Jestem rodzicem.");
        if(wait(0)<0)
                perror("wait");
}

int main(void)
{
        int pid = fork();
        if(pid<0) {
                perror("fork");
                return 0;
        }
        if(pid==0)
                do_child_work();
        else
                do_parent_work();
        return 0;
}
```

Listing 1: Tworzenie pojedynczego procesu potomnego

Rozważmy teraz bardziej skomplikowany przypadek. Spróbujmy utworzyć kilka procesów potomnych. Najprostsze rozwiązanie, jakie odruchowo przychodzi na myśli przedstawia listing 2. Czy jest ono poprawne? To zależy, co chcemy osiągnąć. Proszę zauważyć, że z każdą iteracją pętli wykonywane są dwie czynności: tworzony jest potomek i proces rodzicielski czeka na jego zakończenie. W ten sposób nie powstanie kolejny nowy potomek, zanim nie zakończy się jego poprzednik. Zatem procesy potomne wykonują się współbieżnie[2] z procesem rodzicielskim, a sekwencyjnie względem siebie.

[2] Współbieżne (ang. *concurrent*) wykonanie procesów na komputerze z jednym procesorem oznacza, że ten procesor jest dzielony, a więc przełączany między tymi procesami, a więc są one wykonywane „fragmentami" i nie zawsze da się przewidzieć w jakie kolejności. W przypadku komputera z wieloma procesami, wszystkie lub niektóre procesy, w zależności od ich liczby i liczby procesorów, wykonują się na osobnych procesorach, czyli *równolegle*. Angielski termin oznaczający współbieżność pochodzi stąd, że procesy *rywalizują* ze sobą o dostęp do procesora.

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>

#define NUMBER_OF_CHILDREN 10

void do_child_work(void)
{
        puts("Jestem potomkiem.");
        exit(0);
}

void do_parent_work(void)
{
        puts("Jestem rodzicem.");
        if(wait(0)<0)
                perror("wait");
}

int main(void)
{
        int i;
        for(i=0;i<NUMBER_OF_CHILDREN;i++) {
                int pid = fork();
                if(pid<0) {
                        perror("fork()");
                        break;
                }
                if(pid==0)
                        do_child_work();
                else
                        do_parent_work();
        }

        return 0;
}
```

Listing 2: Tworzenie sekwencyjnych procesów potomnych

Inną możliwość przedstawia listing 3. W tym programie funkcja `generate_child_processes()` tworzy w pętli określoną jej argumentem wywołania liczbę procesów potomnych. Proces potomny natychmiast po powstaniu wykonuje funkcję `do_child_work()`, a proces rodzicielski wykonuje następną iterację pętli, tworząc nowego potomka. W ten sposób wszyscy potomkowie mogą się wykonywać współbieżnie względem siebie i procesu rodzicielskiego. Funkcja `wait_for_children()` jest wykonywana wyłącznie przez proces rodzicielski. Procesy potomne wykonują jedynie `do_child_work()`, gdyż wywołuje ona na koniec funkcję `exit()`, która kończy działanie procesu. W funkcji `wait_for_children()` proces rodzicielski, w pętli wywołuje funkcję `do_parent_work()`, której zadaniem jest oczekiwania na zakończenie procesu potomnego. W tej sposób żaden z nich po zakończeniu procesu macierzystego nie zostanie procesem *zombie*.

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>

#define NUMBER_OF_CHILDREN 10

void do_child_work(void)
{
        puts("Jestem potomkiem.");
        exit(0);
}

void do_parent_work(void)
{
        puts("Jestem rodzicem.");
        if(wait(0)<0)
                perror("wait");
}

void generate_child_processes(unsigned int number_of_children)
{
        int i;
        for(i=0;i<number_of_children;i++) {
                int pid = fork();
                if(pid<0)
                        perror("fork()");
                if(pid==0)
                        do_child_work();
        }
}

void wait_for_children(unsigned int number_of_children)
{
        int i;
        for(i=0;i<number_of_children;i++)
                do_parent_work();
}

int main(void)
{
        generate_child_processes(NUMBER_OF_CHILDREN);
        wait_for_children(NUMBER_OF_CHILDREN);

        return 0;
}
```

Listing 3: Tworzenie współbieżnych procesów potomnych

## 2.1. Przykład

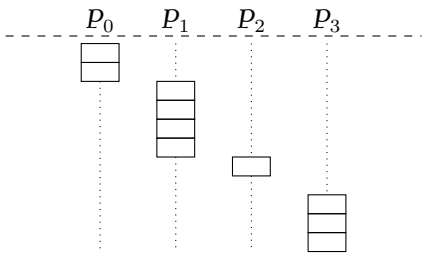
Różnicę między procesami potomnymi wykonującymi się sekwencyjnie i współbieżnie można zaprezentować na przykładzie problemu, w którym każdy z utworzonych procesów musi wejść w stan uśpienia

```c
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<sys/wait.h>

#define NUMBER_OF_CHILDREN 4

void do_child_work(int number_of_seconds)
{
        sleep(number_of_seconds);
        printf("%d\n",number_of_seconds);
        exit(0);
}

void do_parent_work(void)
{
        if(wait(0)<0)
                perror("wait");
}

int main(void)
{
        int array[NUMBER_OF_CHILDREN] = {2,4,1,3};

        for(int i=0; i<NUMBER_OF_CHILDREN; i++) {
                int pid = fork();
                if(pid==-1)
                        perror("fork");
                if(pid==0)
                        do_child_work(array[i]);
                else
                        do_parent_work();
        }

        return 0;
}
```

Listing 4: Sekwencyjnie procesy potomne, które wypisują liczby

na określoną liczbę sekund, a następnie wypisać tę liczbę na ekranie i zakończyć działanie.

Listing 4 prezentuje rozwiązanie tego problemu z użyciem procesów potomnych działających względem siebie sekwencyjnie. Każdy proces potomny realizuje funkcję `do_child_work()` (wiersze 8–13), która otrzymuje jeden argument. Jest nim liczba sekund przez którą ten proces pozostaje w uśpieniu, a następnie ją wypisuje. Jest ona pobierana z tablicy `array` (wiersz nr 23). Ta tablica ma cztery elementy, czyli dokładnie tyle ile w tym programie jest tworzonych procesów. Wartość z jej pierwszego elementu otrzymuje pierwszy utworzony proces potomny, z drugiego drugi, itd. Po wykonaniu programu na ekranie będą wyświetlone w kolumnie liczby 2, 4, 1, 3, dokładnie w takiej kolejności, w jakiej są umieszczone w tablicy `array`. Rysunek 4 przedstawia koncepcyjnie sposób działania programu. Każdy prostokąt sym-




Rysunek 1: Ilustracja sposobu działania programu z listingu 4

bolizuje jedną sekundę czasu. Najpierw pierwszy proces ($P_0$) jest tworzony i usypia na 2 sekundy zanim

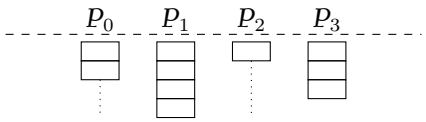
wypisze 2 na ekranie i się zakończy. Potem tworzony jest proces drugi ($P_1$), który śpi przez 4 sekundy, wypisuje 4 na ekranie i się kończy, itd.

Listing 5 przedstawia rozwiązanie problemu z użyciem procesów potomnych tworzonych współbieżnie.

```
1  #include<stdio.h>
2  #include<unistd.h>
3  #include<stdlib.h>
4  #include<sys/wait.h>
5  #include<assert.h>
6
7  #define NUMBER_OF_CHILDREN 4
8
9  void do_child_work(unsigned int number_of_seconds)
10 {
11         sleep(number_of_seconds);
12         printf("%d\n",number_of_seconds);
13         exit(0);
14 }
15
16 void do_parent_work(void)
17 {
18         if(wait(0)<0)
19                 perror("wait");
20 }
21
22 void generate_child_processes(unsigned int number_of_children)
23 {
24         int array[NUMBER_OF_CHILDREN] = {2,4,1,3};
25         assert(number_of_children<=NUMBER_OF_CHILDREN);
26         for(int i=0; i<number_of_children; i++) {
27                 int pid = fork();
28                 if(pid<0)
29                         perror("fork");
30                 if(pid==0)
31                         do_child_work(array[i]);
32         }
33 }
34
35 void wait_for_children(unsigned int number_of_children)
36 {
37         for(int i=0;i<number_of_children;i++)
38                 do_parent_work();
39 }
40
41 int main(void)
42 {
43         generate_child_processes(NUMBER_OF_CHILDREN);
44         wait_for_children(NUMBER_OF_CHILDREN);
45         return 0;
46 }
```

Listing 5: Współbieżne procesy potomne, które wypisują liczby

Tablica `array` została w tym programie przeniesiona do funkcji `generate_child_processes()` (wiersz nr 24). Makro `assert` sprawdza, czy liczba procesów potomnych przekazywana do tej funkcji jest mniejsze lub równa stałem `NUMBER_OF_CHILDREN`, która używana jest również do określania liczby elementów w tablicy. Jeśli tak nie jest, to działanie programu jest przerywane, a w przeciwnym przypadku kontynuowane. Każdy utworzony proces potomny jest, tak jak w poprzednim programie, usypiany na tyle sekund, ile określa liczba umieszczona w odpowiadającym mu elemencie tablicy `array`, a w tym czasie proces macierzysty tworzy kolejnych potomków. Po zakończeniu programu na ekranie pojawią się w kolumnie liczby 1, 2, 3, 4, czyli w kolejności rosnącej. Z tego względu algorytm, który realizuje ten program nazywany jest w języku angielskim „*sleep sort*”, chociaż w rzeczywistości on nie sortuje tablicy, a jedynie wyświetla jej zawartość w posortowanej kolejności. Dodatkowo jest on nieefektywny i działa wyłącznie dla liczb naturalnych. Rysunek 2 przedstawia koncepcyjnie sposób działania tego programu.




Rysunek 2: Ilustracja sposobu działania programu z listingu 5

# 3. Sygnały

Sygnały można uznać za prostą formę komunikacji między procesami, ale przede wszystkim służą one do powiadomienia procesu, że zaszło jakieś zdarzenie, stąd też nazywa się je przerwaniami programowymi. Sygnały są asynchroniczne względem wykonania procesu (nie można przewidzieć kiedy się pojawią). Mogą być wysłane z procesu do procesu lub z jądra do procesu. Programista ma do dyspozycji funkcję `kill()`, która umożliwia wysłanie sygnału do procesu o podanym `PID`. Z każdym procesem jest związana struktura, w której umieszczone są adresy procedur obsługi sygnałów. Jeśli programista nie napisze własnej funkcji obsługującej dany sygnał, to wykonywana jest procedura domyślna, która powoduje natychmiastowe zakończenie procesu lub inne, zależne od konfiguracji zachowanie. Część sygnałów można ignorować, lub zablokować je na określony czas. Niektórych sygnałów nie można samemu obsłużyć, ani zignorować, ani zablokować (np. `SIGKILL`).

Listing 6 zawiera kod źródłowy prostego programu, który zmienia, przy pomocy funkcji `sigaction()`, obsługę sygnału `SIGUSR1`, wysyła do siebie ten sygnał, a następnie przywraca jego oryginalną obsługę i ponownie przesyła ten sygnał do siebie. Twórcy mechanizmu sygnałów przeznaczyli sygnały `SIGUSR1` i `SIGUSR2` do dyspozycji programisty, dlatego nie mają one z góry określonego znaczenia. Ich domyślna obsługa, tak jak w większości innych sygnałów, polega na zakończeniu działania procesu.

W wierszach 8–18 opisywanego programu zdefiniowana jest funkcja, która zastąpi tę domyślną obsługę w przypadku sygnału `SIGUSR1`. Nazywa się ją *procedurą obsługi sygnału*. Jest to jeden z dwóch rodzajów procedur obsługi, który może zarejestrować funkcja `sigaction()`. Drugi jest prostszy, ale może być także zarejestrowany przez funkcję `signal()`.

Funkcja `handler()` ma trzy parametry. Przez pierwszy otrzymuje numer sygnału, który spowodował jej wykonanie, drugi jest wskaźnikiem na strukturę typu `siginfo_t`, która przechowuje bardziej szczegółowe informacje o tym sygnale, a trzeci jest wskaźnikiem, przez który przekazywany jest adres struktury zawierającej informacje o kontekście procesu, w trakcie otrzymania sygnału. Ten adres nie jest używany przez większość procedur obsługi sygnałów. Funkcja `handler()` również go nie potrzebuje, dlatego w wierszu nr 10 jest on rzutowany na typu `void`. Następnie wypisywany jest numer sygnału przekazany funkcji w pierwszym parametrze. Potem (wiersz nr 13) funkcja sprawdza, czy pole `si_code` struktury typu `siginfo_t` ma wartość stałej `SI_USER`. W tym polu przechowywany jest tzw. *kod sygnału*, czyli informacja określająca *przyczynę powstania sygnału*. Jeśli jego wartość wynosi `SI_USER`, to znaczy, że obsługiwany sygnał został wysłany przez proces użytkownika.

Po sprawdzeniu kodu sygnału funkcja `handle()` może wypisać wartości innych pól tej struktury typu `siginfo_t`, które przechowują informacje związane z procesem, który wysłał sygnał. Są to identyfikator tego procesu, przechowywany w polu `si_pid` (wiersz nr 15) oraz rzeczywisty identyfikator użytkownika, z którego uprawnieniami ten proces jest wykonywany, odczytany z pola `si_uid` (wiersz nr 16).

Struktura typu `siginfo_t` ma wiele innych pól. Jej dokładna budowa oraz możliwe kody sygnału są opisane na stronie podręcznika `man` funkcji `sigaction()` (`man sigaction`).

W wierszu nr 22 programu tworzone są dwie zmienne lokalne będące strukturami typu `struct sigaction`. Budowa tej struktury również jest opisana na stronie podręcznika `man` dla funkcji `sigaction()`. Pozwala ona określić sposób rejestracji procedury obsługi sygnału. W opisywanym programie zmieniane są wartości tylko dwóch jej pól: `sa_flags` i `sa_sigaction`. Pierwsze pole zawiera *flagi*, czyli wartości definiujące zachowanie funkcji `sigaction()` podczas rejestracji procedury obsługi funkcji. W wierszu nr 24 do struktury typu `struct sigaction`, o nazwie `new_action` jest przypisywana wartość stałej `SA_SIGINFO`, która informuje funkcję `sigaction()`, że adres procedury obsługi jest zapisany w polu `sa_sigaction` tej struktury, a nie w `sa_handler`. Tego ostatniego pola należy użyć, jeśli mamy zamiar zarejestrować prostszą procedurę obsługi sygnału. Adres procedury obsługi zdefiniowanej w programie jest przypisywany do pola `sa_sigaction` w wierszu nr 25.

W wierszu nr 26 wywoływana jest funkcja `sigaction()`, celem rejestracji nowej obsługi sygnału

```c
#include<stdio.h>
#include<signal.h>
#include<sys/types.h>
#include<unistd.h>

#define LENGTH 50

void handler(int singal_number, siginfo_t *signal_information, void *context)
{
    (void)context;
    printf("Numer sygnału: %d.\n", singal_number);
    printf("Numer sygnału ze struktury sig_info: %d\n",signal_information->si_signo);
    if(signal_information->si_code == SI_USER) {
        puts("Sygnał został wysłany przez proces użytkownika.");
        printf("PID procesy wysyłającego ze struktury sig_info: %d.\n",signal_information->si_pid);
        printf("Rzeczywisty identyfikator użytkownika procesu wysyłającego:
        ↪  %d.\n",signal_information->si_uid);
    }
}

int main(void)
{
    struct sigaction new_action, old_action;
    char function_and_location[LENGTH];
    new_action.sa_flags = SA_SIGINFO;
    new_action.sa_sigaction = handler;
    if(sigaction(SIGUSR1,&new_action,&old_action)<0) {
        snprintf(function_and_location,LENGTH,"sigaction %d",__LINE__-1);
        perror(function_and_location);
    }
    if(kill(getpid(),SIGUSR1)<0)
        perror("kill");
    if(sigaction(SIGUSR1,&old_action,0)<0) {
        sprintf(function_and_location,"sigaction %d",__LINE__-1);
        perror(function_and_location);
    }
    if(kill(getpid(),SIGUSR1)<0)
        perror("kill");
    puts("Nieosiągalny fragment kodu.");
    return 0;
}
```

Listing 6: Prosty przykład definiujący obsługę sygnału

`SIGUSR1`. Jako pierwszy argumenty przekazywana jest jej stała określająca numer tego sygnału, jako drugi adres struktury `new_action`, która definiuje nową obsługę, która będzie wykonywana przez funkcję `handle()`. Trzecim argumentem funkcji `sigaction()` jest adres struktury `old_action`, która również jest typu `struct sigaction`. W tej strukturze funkcja `sigaction()` zapisze bieżący (czyli przed podmianą) sposób obsługi sygnału `SIGUSR1`. Będzie ona potrzebna do tego, by program przywrócił domyślna obsługę tego sygnału.

Proszę zwrócić uwagę, że funkcja `sigaction()` wywoływana jest w programie dwukrotnie. Za pierwszym razem (wiersz nr 26) rejestruje nową obsługę sygnału, a za drugim (wiersz nr 32) przywraca starą. W obu przypadkach program sprawdza, czy wykonała się ona poprawnie. Jeśli nie, to wypisuje na ekranie komunikat o wyjątku przy pomocy funkcji `perror()`. Problem w tym, że `perror()` jako argument przyjmuje nazwę funkcji, której dotyczy defekt, ale w tym wypadku to może nie wystarczyć do jednoznacznego ustalenia *gdzie* on powstał, z uwagi na dwukrotne użycie `sigaction()`. Dlatego do nazwy funkcji dołączany jest numer wiersza, w którym ta funkcja jest wywoływana. Ustalany jest on przy pomocy wyrażenia `__LINE__-1`. Makro `__LINE__` w trakcie kompilacji zamieniane jest na numer wiersza, w którym jest użyte, ale funkcja `sigaction()` wywoływana jest wiersz wcześniej, stąd trzeba odjąć jedynkę. Nazwa funkcji i numer wiersza łączone są w jeden ciąg znaków przy pomocy funkcji `snprintf()`. Jej wynik zapisywany jest w tablicy znaków `functon_and_location`, zadeklarowanej w wierszu nr 23.

Po zarejestrowaniu nowej obsługi sygnału `SIGUSR1` program wysyła go do siebie przy pomocy funkcji

`kill()` (wiersz nr 30). Jako pierwszego argumentu ta funkcja potrzebuje identyfikatora procesu, który program uzyskuje przy pomocy innej funkcji, czyli `getpid()`. Po otrzymaniu sygnału program wykona funkcję `handler()`.

Po wykonaniu obsługi `SIGUSR1` przy pomocy funkcji `handler()` program przywróci jego domyślną obsługę. W tym celu ponownie wywoła `sigaction()` (wiersz nr 32), tym razem jako pierwszy argument przekazując jej adres struktury `old_action`, a jako `0` (można użyć także `NULL`). Zapamiętanie bieżącej obsługi sygnału nie jest konieczne przed rejestracją domyślnej, stąd taka wartość ostatniego argumentu tej funkcji.

Po przywróceniu poprzedniej obsługi sygnału, program ponownie wysyła go do siebie (wiersz nr 36). Tym razem program zakończy swoje działanie, bo na tym właśnie polega domyślna obsługa `SIGUSR1`. Dowodem tego jest to, że nie dojdzie do wywołania funkcji `puts()`, która wyświetliłaby na ekranie napis *Nieosiągalny fragment kodu.*. Więcej na temat sygnałów i jej domyślnej obsługi można dowiedzieć się wydając polecenie: `man 7 signal`.

# 4. Opis ważniejszych funkcji

`fork()` - stwórz proces potomny. Funkcja ta nie ma argumentów i zwraca dwie wartości: dla procesu macierzystego - `PID` potomka, dla procesu potomnego `0`. Jeśli jej wywołanie się nie powiedzie, to zwraca wartość `-1`. Kodu programu z oprogramowanym zachowaniem potomka i rodzica został zaprezentowany na listingu 1.
Szczegóły: `man fork`
`clone()` - funkcja specyficzna dla Linuksa, służy do tworzenia nowego procesu.
Szczegóły: `man clone`
`getpid()` i `getppid()` - funkcje zwracają odpowiednio: `PID` procesu bieżącego i `PID` jego rodzica. Nie przyjmują żadnych argumentów.
Szczegóły: `man getpid`
`sleep()` - służy do „uśpienia" procesu (zawieszenia jego działania) na określoną liczbę sekund, którą przyjmuje przez argument. Jeśli w trakcie uśpienia proces otrzyma sygnał, to `sleep()` przerywa uśpienie i zwraca liczbę sekund, które pozostały do zakończenia zadanego czasu oczekiwania. Jeśli uśpienie zakończy się w normalny sposób, to `sleep()` zwraca zero.
Szczegóły: `man 3 sleep`
`wait` - nie jest to jedna funkcja, ale rodzina funkcji (`wait()`, `waitpid()`, `wait3()`, `wait4()`). Powodują one, że proces macierzysty czeka na zakończenie procesu potomnego. Status zakończenia procesu możemy poznać korzystając z odpowiednich makr. W przypadku funkcji `wait()` status ten zapisywany jest w zmiennej typu `int`, której adres jest argumentem funkcji. Funkcja ta zwraca PID procesu potomnego, który się zakończył, lub $-1$ w przypadku pojawienia się wyjątku.
Szczegóły: `man 2 wait`.
`exit()` - funkcja kończąca wykonanie procesu. Nic nie zwraca, a jako argument przyjmuje status zakończenia procesu, w postaci liczby typu `int`. Istnieje kilka innych podobnych funkcji.
Szczegóły: `man 3 exit`.
`exec` - rodzina funkcji (`execl()`, `execlp()`, `execle()`, `execv()`, `execv()`), które zastępują obraz w pamięci aktualnie wykonywanego procesu obrazem nowego procesu, odczytanym z pliku.
Szczegóły: `man 3 exec`.
`kill()` - funkcja powodująca wysłanie sygnału o określonym numerze do procesu o określonym `PID`. Obie wartości przyjmuje jako argumenty (pierwszy PID, drugi numer sygnału). Zwraca 0 w przypadku powodzenia lub $-1$ w przeciwnym przypadku. Szczegóły: `man 2 kill`.
`signal()` - funkcja pozwala określić zachowanie procesu, po otrzymaniu odpowiedniego sygnału. Z tą funkcją powiązane są funkcje `sigblock()` i `sigsetmask()`. Współcześnie zalecane jest stosowanie `sigaction()` i `sigprocmask()` zamiast `signal()`.
Szczegóły: `man signal`, `man sigblock`, `man sigsetmask`, `man sigaction`, `man sigprocmask`.
`pause()` - funkcja wstrzymuje działanie procesu do momentu, aż otrzyma on sygnał. Nie przyjmuje ona żadnych argumentów.
Szczegóły: `man pause`.
`alarm()` - wysyła do procesu sygnał `SIGALRM` po ustalonym czasie. Jako argument przyjmuje liczbę sekund, po których ma wysłać sygnał. Jeśli po wywołaniu tej funkcji zostanie ona ponownie wywołana

z liczbą 0 jako argumentem, to anuluje uruchomione odliczanie. Funkcja `alarm()` zwraca 0 jeśli odliczanie się zakończy lub liczbę pozostałych sekund do zakończenia, jeśli zostanie anulowane.
Szczegóły: `man alarm`.
`perror()` - wypisuje na ekranie komunikat związany z ostatnim napotkanym wyjątkiem pochodzącym od funkcji systemowej, po której jest wywoływana. Jako argument przyjmuje ciąg znaków będący nazwą funkcji systemowej, której wykonanie zakończyło się wyjątkiem i nic nie zwraca.
Szczegóły: `man 3 perror`.

# 5. Zadania

UWAGA! PROGRAMY NALEŻY NAPISAĆ Z PODZIAŁEM NA FUNKCJE Z PARAMETRAMI. KAŻDY PROGRAM MUSI SPRAWDZAĆ, CZY FUNKCJA, Z TYCH, KTÓRE SĄ OPISANE WYŻEJ NIE SYGNALIZUJE WYJĄTKU PODCZAS WYKONANIA, O ILE Z JEJ DOKUMENTACJI WYNIKA, ŻE MOŻE COŚ TAKIEGO ROBIĆ.

1. Napisz program, który utworzy proces potomny. Proces rodzicielski powinien wypisać swoje PID i PID potomka, natomiast proces potomny powinien wypisać swoje PID i PID rodzica.
2. Zademonstruj w jaki sposób mogą powstać w systemie procesy *zombie*.
3. Napisz program, który stworzy proces potomny. Proces macierzysty powinien poczekać na wykonanie procesu potomnego i zbadać status jego wyjścia.
4. Napisz program, który w zależności od wartości argumentu podanego w wierszu poleceń wygeneruje odpowiednią liczbę procesów potomnych, które będą się wykonywały współbieżnie. Każdy z procesów potomnych powinien wypisać 4 razy na ekranie swój PID, PID swojego rodzica oraz numer określający, którym jest potomkiem rodzica (1, 2, 3 ...), a następnie usnąć na tyle sekund, ile wskazuje ten numer (pierwszy - 1 sekunda, 2 - dwie sekundy, trzeci - 3 sekundy, ...). Proces macierzysty powinien poczekać na zakończenie wykonania wszystkich swoich potomków.
5. Napisz dwa programy. Program pierwszy stworzy proces potomny, a następnie zastąpi jego program drugim programem.
6. Napisz program, który wyśle do siebie sygnał SIGALRM i obsłuży go.
7. Napisz program, który stworzy dwa procesy. Proces rodzicielski wyśle do potomka sygnał SIGINT (można go wysłać „ręcznie“ naciskając na klawiaturze równocześnie `Ctrl + c`). Proces potomny powinien ten sygnał obsłużyć za pomocą napisanej przez Ciebie funkcji. Do jej rejestracji użyj `signal()`.
8. Napisz cztery osobne programy. Każdy z nich powinien obsługiwać wybrany przez Ciebie sygnał. Pierwszy z procesów będzie co sekundę wysyłał sygnał do drugiego procesu, drugi proces po odebraniu sygnału powinien wypisać na ekranie komunikat, a następnie przesłać sygnał do procesu trzeciego. Proces trzeci powinien zachowywać się podobnie jak drugi, a proces czwarty powinien jedynie wypisywać komunikat na ekranie. Odliczanie czasu w pierwszym procesie należy zrealizować za pomocą SIGALRM.
9. Napisz program, który udowodni, że obszar danych jest współdzielony między procesem potomnym i macierzystym do chwili wykonania modyfikacji danych przez jednego z nich.
10. Ze względów bezpieczeństwa zaleca się, aby w ramach funkcji obsługującej sygnał wykonywane były tylko proste czynności, jak np. ustawienie flagi informującej o otrzymaniu sygnału, a skomplikowane czynności żeby były wykonywane w osobnym kodzie. Przedstaw schemat takiego rozwiązania stosując proces macierzysty i potomny.
11. Pokaż w jaki sposób sygnały mogą być przez proces blokowane lub ignorowane.
12. Aby procesy potomne nie stawały się procesami zombie wystarczy, żeby proces macierzysty ignorował sygnał SIGCHLD. Napisz program, który sprawdzi, czy rzeczywiście tak się dzieje i co w takim przypadku zwraca `wait()` lub `waitpid()` po zakończeniu potomka.