SPIS TREŚCI

1. Funkcje C

- **pid_t fork (void) [#include <unistd.h>]**
  - ☞ działanie funkcji
    - funkcja tworzy nowy proces
    - nowy proces jest możliwie dokładną kopią procesu wywołującego $\mathrm{fork}()$ (w sensie stosu, sterty, instrukcji) ale :
      - → mają różne ID procesu (*process ID*)
      - → mają różne ID procesu rodzica (ten nowy ma wpisany ID wywołującego $fork()$)
      - → proces potomny ma własną kopię deskryptorów plików rodzica
        - ale odwołują się do tych samych plików co deskryptory rodzica
      - → proces potomny otrzymuje nowe niezależne zasoby
    - w momencie wywołania $fork()$ proces potomny wykonuje się współbieżnie z rodzicem od miejsca wywołania $fork()$
      - → zatem do funkcji $fork()$ wchodzi tylko jeden proces (rodzic) a wychodzą z niej dwa procesy (rodzic i proces potomny) z różnymi wartościami zwracanymi przez $fork()$
  - ☞ wartość zwracana
    - PID procesu potomnego → w procesie rodzica
    - 0 → w procesie potomnym
    - -1 → jeśli nie zostanie utworzony proces potomny, ustawia errno
      - → EAGAIN - system nie miał zasobów do stworzenia nowego procesu lub istnieje za dużo procesów w tym momencie
      - → ENOMEM - niewystarczająca ilość pamięci

- **pid_t getpid (void) [#include < unistd.h>]**
  - ☞ działanie funkcji
    - funkcja pobiera process ID (*PID*) wywołującego ją procesu
  - ☞ wartość zwracana
    - PID wywołującego procesu → zawsze powinna mieć szczęśliwe zakończenie bez błędów

---

- **pid_t wait (int *status) [#include<sys/wait.h>]** dodatkowe materiały
  - ☞ działanie funkcji
    - funkcja zawiesza działanie wywołującego ją procesu do czasu, gdy :
      - → informacja o statusie jednego z zakończonych procesów potomnych jest dostępna
      - → dostanie sygnał którego zadaniem jest zakończyć proces
    - w skrócie funkcja czeka na zakończenie procesu potomnego

- ☞ argumenty :
  - *int* status*
    - → jeżeli == NULL → nic się nie dzieje
    - → jeżeli != NULL → w *status* zapisywana jest przyczyna zakończenia procesu potomnego
      - 0 jeśli :
        - ► proces potomny zwrócił 0 z $main()$
        - ► proces wywołał $exit()$ lub $\_exit()$ z argumentem 0
        - ► proces został zakończony ponieważ ostatni wątek procesu został zakończony
- ☞ informacja zawarta w *status* może być zinterpretowana za pomocą makr [#include<sys/wait.h>]
  - $WIFEXITED\ (\ status\ )$
    - → niezerowa wartość, gdy potomek zostanie zakończony normalnie
  - $WEXITSTATUS\ (\ status\ )$
    - → jeśli wartość $WIFEXITED\ (\ status\ )$ jest niezerowa, to makro odwołuje się do 8 mniej ważnych bitów argumentu $exit()$ lub $\_exit()$ wywołanego przez proces potomny lub wartości zwracanej z $main()$ procesu potomnego
  - $WIFSIGNALED\ (\ status\ )$
    - → niezerowa wartość dla procesu potomnego, który został zakończony z powodu nie wyłapanego wygnału
  - $WTERMSIG\ (\ status\ )$
    - → jeśli $WIFSIGNALED\ (\ status\ )$ zwróci niezerową wartość, to makro odwołuje się do numeru sygnału, który spowodował zakończenie procesu potomnego
  - $WIFSTOPPED\ (\ status\ )$
    - → niezerowa wartość dla procesu potomnego, który jest aktualnie zatrzymany [*stopped*]
  - $WSTOPSIG\ (\ status\ )$
    - → jeśli $WIFSTOPPED\ (\ status\ )$ jest niezerowe, to makto odwołuje się do numeru sygnału, który spowodował zatrzymanie procesu potomnego
  - $WIFCONTINUED\ (\ status\ )$
    - → niezerowa wartość dla procesu potomnego, który jest kontynuowany przez *job control*
- ☞ wartość zwracana
  - PID zakończonego procesu potomnego → jeśli $wait()$ kończy się, ponieważ proces potomny się zakończył
  - -1 → w przypadku błędu
    - → ECHILD → proces wywołujący $wait()$ nie ma istniejących procesów potomnych
    - → EINTR → funkcja została przerwana przez sygnał, wartość sygnału przechowana w $status$

- **pid_t waitpid (pid_t pid , int* status, int options)**
  - ☞ działanie funkcji :
    - zawiesza działanie procesu aż do zakończenia procesu potomnego
    - z argumentami $pid = -1, options = 0$ działa jak $\boldsymbol{wait()}$
  - ☞ argumenty :
    - $pid\_t\ pid$ → określa zbiór procesów potomnych, na które oczekuje
      - → -1 → jakikolwiek proces potomny
      - → > 0 → dokładny proces potomny o $PID = pid$
      - → 0 → jakikolwiek proces potomny o GID równym GID rodzica
      - → < -1 → jakikolwiek proces potomny o $GID = |pid|$
    - $int * status$ → zapisany status – przyczyna zakończenia procesu potomnego, podobnie jak w $\boldsymbol{wait()}$
    - $int\ options$ → flaga, opcja działania funkcji
      - → WNOHANG → natychmiastowy powrót (nie wstrzymuje procesu wywołującego) jeśli potomek nie zakończył w tym momencie jeszcze działania
      - → WUNTRACED → oznacza zakończenie także dla dzieci, które się zatrzymały, a których status jeszcze nie został ogłoszony
  - ☞ wartość zwracana
    - PID zakończonego procesu potomnego → $wait()$ kończy się, bo proces potomny się kończy
    - 0 → gdy $WNOHANG$ ustawione a nie ma żadnego potomka
    - -1 → w przypadku błędu , errno :
      - → ECHILD → proces określony przez pid nie istnieje lub nie jest procesem potomnym → gdy grupa określona przez pid nie istnieje lub nie ma żadnego procesu, który byłby procesem potomnym procesu wywołującego $waitpid()$
      - → EINTR → funkcja została przerwana przez sygnał
      - → EINVAL → argument $options$ jest nieprawidłowy

- **unsigned int sleep (unsigned int seconds) [#include <unistd.h>]**
  - ☞ działanie funkcji :
    - wstrzymanie działania procesu na czas $seconds$ lub do odebrania sygnału
    - zawieszenie działania może być dłuższe ze względu na planowanie działań w systemie
  - ☞ argumenty :
    - $seconds$ → liczba sekund, przez które proces ma być wstrzymany
  - ☞ wartość zwracana :
    - 0 → jeśli minęło $seconds$ sekund
    - >0 → jeśli $sleep()$ przerwane przez sygnał, zwraca ilość „nieprzespanych” sekund

- **int sigaction (int sig, const struct sigaction* act, struct sigaction * oact) [#include < signal.h>]**

☞ działanie funkcji (**dodatkowe materiały**)

– funkcja pozwala procesowi, który ją wywołał, badać i określić akcję jaka będzie związana z otrzymanym określonym sygnałem

– struktura sigaction określa akcję podejmowaną w przypadku podania sygnału sig

→ $void(*)(int)\ sa\_handler$ → wskaźnik na funkcję wyłapującą sygnały

→ $sigset\_t\ sa\_mask$ → zbiór sygnałów blkowanych podczas wykonania funkcji $sa\_handler$

→ $int\ sa\_flags$ → podaje zbiór flag, które modyfikują zachowanie procesu obsługi sygnałów, zbiór wartości OR :

- SA_NOCLDSTOP → jeśli $sig = SIGCHLD$ nie odbieraj powiadomienia o zatrzymaniu procesu potomnego (tj. gdy proces potomny otrzyma jeden z : SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU)
- SA_ONSTACK → wywołaj $sa\_handler$ na zastępczym stosie sygnałów dostarczonym przez $sigaltstack()$.
  → jeżeli stos zastępczy nie jest dostępny, użyty będzie stos domyślny
- SA_RESETHAND → przy wejściu do procedury obsługi sygnału jest przywracana domyślna (SID_DFL) dyspozycja dla tego sygnału
- SA_RESTART → wywołania systemowe przerwane przez ten sygnał są automatycznie powtarzane
- SA_SIGINFO → używaj $sa\_sigaction$ a nie $sa\_handler$
- SA_NOCLDWAIT → jeśli $sig = SIGCHLD$ to procesy potomne po ich zakończeniu nie będą Zombie
- SA_NODEFER → taki sygnał po przechwyceniu nie zostanie automatycznie zablokowany przez system w czasie wykonania procedury obsługi sygnału

→ $void(*)(int, siginfo\_t *, void *)\ sa\_sigaction$ → wskaźnik na funkcję wyłapującą sygnały

- $siginfo\_t$
  - ► $int\ si\_signo$ → numer sygnału
  - ► $int\ si\_errno$ → wartość errno
  - ► $int\ si\_code$ → dodatkowe informacje, zależne od sygnału
  - ► $pid\_t\ si\_pid$ → identyfikator procesu wysyłającego sygnał
  - ► $uid\_t\ si\_uid$ → rzeczywisty identyfikator użytkownika procesu wysyłającego sygnał
  - ► $inne\ pola...$
- trzeci argument może być obiektem typu $ucontext\_t$

☞ argumenty :

– $int\ sig$ → określa sygnał, akceptowane wartości określone w $< signal.h >$

– $struct\ sigaction * act$ → określa akcję podejmowaną dla sygnału sig

→ jeśli jest $NULL$ → obsługa sygnału nie zmienia się

→ jeśli nie jest $NULL$ → określa obsługę sygnału

– $struct\ sigaction * oact$ → określa akcję podejmowaną wcześniej dla sygnału

→ jeśli nie jest NULL → będzie tam zapisana akcja podejmowana wcześniej dla sygnału sig

☞ wartość zwracana

– 0 → w przypadku zakończenia powodzeniem

– -1 → w przypadku zakończenia niepowodzeniem, ustawia errno na :

→ EINVAL → sig nie jest odpowiednim numerem sygnału, lub próbowano przechwycić sygnał, którego nie można przechwytywać

- **int nanosleep (const struct timespec* rqtp, struct timespec* rmtp) [#include<time.h>]**
  - ☞ działanie funkcji :
    - wstrzymuje aktualny wątek dopóki nie upłynie czas określony przez $rqtp$ lub dopóki nie zotał dostarczony sygnał
    - wstrzymanie wątku może być dłuższe z powodu planowania zadań w systemie
    - poza przypadkiem z sygnałem czas wstrzymania nie powinien być krótszy niż $rqtp$
    - użycie $nanosleep()$ nie ma wpływu na blokowanie sygnałów
    - $struct\ timespec$
      - → $time\_t$ tv_sec → sekundy
      - → $long$ tv_nsec → nanosekundy ∈ $[1,999\ 999\ 999]$
  - ☞ argumenty :
    - $struct\ timespec\ rqtp$ → czas wstrzymania
    - $struct\ timespec\ rmtp$
      - → jeśli != NULL → czas pozostały w przypadku przerwania działania funkcji
      - → jeśli == NULL → NULL ☺
  - ☞ wartość zwracana :
    - 0 → funkcja zakończona powodzeniem (odczekała $rqtp$ czasu)
    - -1 → funkcja zakończona niepowoedzniem, ustawia errno :
      - → EINTR → przerwana przez sygnał
      - → EINVAL → wartość $rqtp$ nie należy do przedziału $[1,999\ 999\ 999]$

- **unsigned alarm (unsigned seconds) [#include <unisdt.h>]**
  - ☞ działanie funkcji :
    - wywołuje SIGALRM dla procesu po upływie $seconds$ sekund
    - jeżeli istnieje już wcześniej zainicjowany licznik, jego wartość zostaje nadpisana
  - ☞ argumenty :
    - $unsigned\ seconds$ → liczba sekund do wysłania SIGALRM
      - → jeśli == 0 to anuluje wywołanie SIGALRM, nie będzie alarmu
  - ☞ wartość zwracana :
    - $\neq 0$ → jeśli była wcześniej wywołana funkcja $alarm()$ z podanym czasem
      - → wtedy zwraca ilość sekund, która została do wysłania sygnału
    - $= 0$ → w przeciwnym przypadku

- **void* memset (void* s, int c, size_t n) [#include<string.h>]**
  - ☞ działanie funkcji :
    - funkcja wypełnia pamięć s podanym bajtem c (n razy)
  - ☞ argumenty :
    - $void * s$ → wskaźnik na obszar do wypełnienia
    - $int\ c$ → bajt wypełnienia (konwertowany na unsigned char)
    - $size\_t\ n$ → długość obszaru do wypełnienia
  - ☞ wartość zwracana :
    - zwraca wskaźnik na s
    - nie ma wartości wskazujących na error, nie ustawia errno

- **int kill (pid_t pid, int sig) [#include<signal.h>]**
  - ☞ działanie funkcji :
    - funkcja wysyła sygnał procesowi (lub grupie procesów) określonych wartością pid
    - sygnał jest określony w bibliotece $< signal.h >$
    - funkcja kończy się powodzeniem, jeśli proces wysyłający sygnał ma uprawnienia do wysłania sygnału sig do procesów określonych przez pid
  - ☞ argumenty :
    - $pid\_t\ pid$ → proces (lub grupa procesów) do których wysyłany jest sygnał
      - → > 0 → sygnał wysyłany do konkretnego procesu $PID = pid$
      - → $= 0$ → sygnał wysyłany do wszystkich procesów, których GID jest równy GID procesu wysyłającego sygnał
      - → $= -1$ → sygnał wysyłany do wszystkich procesów, do których ma uprawnienia, żeby być wysłanym
      - → < -1 → sygnał wysyłany do wszystkich procesów, których $GID = |pid|$
    - $int\ sig$ → numer sygnału
  - ☞ wartość zwracana :
    - 0 → funkcja zakończona powodzeniem
    - -1 → funkcja zakończona niepowodzeniem, ustawia errno :
      - → EINVAL → wartość sig nie jest prawidłowa
      - → EPERM → brak uprawnień do wysłania sygnału
      - → ESRCH → nie ma procesów spełniających pid

- **int sigsuspend (const sigset_t* set) [#include<signal.h>]**
  - ☞ działanie funkcji :
    - funkcja zamienia aktualną maskę sygnałów wywołującego ją wątku na maskę set
    - następnie wstrzymuje wątek aż do otrzymania sygnału
    - nie można zablokować SIGKILL i SIGSTOP (próby takie są ignorowane)
    - jeśli sygnał ma zakończyć proces, $sigsuspend()$ nic nie zwraca
    - jeśli sygnał ma wykonać funkcję wyłapującą sygnały wtedy $sigsuspend()$ zwraca to, co zwraca ta funkcja wyłapująca
      - → ustawia maskę sygnałów procesu spowrotem na wartość sprzed wywołania $sigsuspend()$
  - ☞ argument :
    - $const\ sigset\_t * set$ → maska blokowanych sygnałów
  - ☞ zwracana wartość
    - -1 i ustawia errno :
      - → EINTR → proces wywołujący $sigsuspend()$ otrzymał sygnał i zwrócił wartość z funkcji przechwytującej sygnały

---

- **pid_t getppid (void) [#include<unistd.h>]**
  - ☞ działanie funkcji :
    - – funkcja pobiera PID rodzica procesu, który ją wykonał
  - ☞ zwracana wartość :
    - – zawsze powinien być to PID rodzica
    - – nie przewiduje wystąpienia błędu

- **int sigprocmask (int how, const sigset_t* set, sigset_t* oset) [#include<signal.h>]**
  - ☞ działanie funkcji :
    - – funkcja modyfikuje bieżącą maskę procesu
  - ☞ argumenty :
    - – how → określa w jaki sposób zmieniana jest maska
      - → SIG_BLOCK → sygnały z set zostaną dodane do maski sygnałowej
      - → SIG_UNBLOCK → sygnały z set są usuwane z bieżącego zestawu sygnałów blokowanych
        → można próbować usuwać niezablokowane sygnały
      - → SIG_SETMASK → maska sygnałowa zamieniana na set
    - – set → maska sygnałów modfikująca aktualną maskę sposobem how
    - – $oset$ → jeśli nie jest $NULL$ to zapisywana jest tu maska sygnałów przed zmianą
  - ☞ wartość zwracana :
    - – 0 → funkcja zakończona powodzeniem
    - – -1 → funkcja zakończona niepowodzeniem, ustawia errno :
      - → EINVAL → wartość how jest nieprawidłowa

- **int sigaddset (sigset_t* set, int signo) [#include<signal.h>]**
  - ☞ działanie funkcji
    - – dodaje sygnał do zbioru sygnałów set
  - ☞ argumenty :
    - – $sigset * set$ → zbiór sygnałów, do którego dodawany jest sygnał
    - – $int\ signo$ → sygnał, który ma być dodany do zbioru
  - ☞ wartość zwracana :
    - – 0 → funkcja zakończona powodzeniem
    - – -1 → funkcja zakończona niepowodzeniem, ustawia errno :
      - → EINVAL → wartość $signo$ jest nieprawidłowa

- **int sigemptyset (sigset_t* set) [#include<signal.h>]**
  - ☞ działanie funkcji :
    - inicjalizuje zbiór sygnałów set jako pusty (nie zawiera standardowych sygnałow)
  - ☞ argumenty :
    - $sigset\_t * set$ → zbiór, który ma być zainicjalizowany
  - ☞ wartość zwracana :
    - 0 → funkcja zakończona powodzeniem
    - -1 → funkcja zakończona niepowodzeniem

- **int open (const char* path, int oflag, …) [#include<sys/stat.h> #include<fcntl.h>]**
  - ☞ działanie funkcji :
    - funkcja tworzy deskryptor pliku będącego pod ścieżką $path$
    - otwiera plik lub urządzenie
  - ☞ argumenty :
    - $const\ char * path$ → ścieżka pliku, który ma być otwarty
    - $int\ oflag$ → flagi otwarcia (OR)
      - → O_EXEC → otwiera tylko do wykonania, nie dla katalogów
      - → O_RDONLY → otwiera tylko do odczytu
      - → O_RDWR → otwiera do odczytu i zapisu
      - → O_SEARCH → otwiera do wyszukiwania
      - → O_WRONLY → otwiera tylko do zapisu
    - $oflag$ może być łączony z flagami :
      - → O_APPEND → ustawia odczyt na koniec pliku (dopisywanie)
      - → O_CLOEXEC → jeśli ustawiona, FD_CLOEXEC jest ustawiane dla nowego deskryptora pliku
        - FD_CLOEXEC [*close-on-exec*] zamyka deskryptor jeśli wykona się jakakolwiek funkcja z rodziny $exec$
      - → O_CREAT → jeśli plik nie istnieje to będzie stworzony
      - → O_DIRECTORY → jeśli $path$ odnosi się nie do katalogu, funkcja kończy się niepowodzeniem i ustawia $errno = ENOTDIR$
      - → O_SYNC → wszystkie zapisy na utworzonym deskryptorze pliku będą blokować proces wołający aż do fizycznego zapisania danych na sprzęcie
      - → O_EXCL → w połączeniu z O_CREAT, jeśli plik już istnieje, $open()$ się nie powiedzie
      - → O_NOCTTY → jeżeli $path$ odnosi się do terminala, nie stanie się on terminalem sterującym procesu, nawet jeśli proces takiego nie ma
      - → O_NOFOLLOW → jeśli $path$ to symlink, $open()$ kończy się niepowodzeniem i $errno = ELOOP$
      - → O_NONBLOCK → plik otwierany w trybie nieblokującym, ani $open()$ ani kolejne operacje na tym deskryptorze nie spowodują blokowania procesu (zatrzymania w oczekiwaniu danych itp.)
      - → O_TRUNC → jeśli plik istnieje i jest regularnym plikiem i jest otwarty z poowdzeniem O_RDWR lub O_RWONLY, jego długość jest skracana do 0 (nadpisanie pliku)
  - ☞ zwracana wartość
    - nieujemna liczba całkowita → w przypadku powodzenia, reprezentuje najniższy numer nieużywanego deskryptora pliku (czyli nowy deksryptor)
    - -1 → w przypadku niepowodzenia
      - → Może ustawiać dużo różnych errno (kilka opisanych **na tej stronie**)

- **int close (int fildes) [#include<unistd.h>]**
  - ☞ działanie funkcji :
    - funkcja zwalnia deksryptor pliku
    - deskryptor dzięki temu może być ponownie wykorzystany
    - jeżeli $fildes$ jest ostatnią kopią deskryptora pliku, zasoby z nim związane zostają zwolnione
    -
  - ☞ argumenty :
    - $int\ fildes$ → deksryptor pliku, który ma zostać zwolniony
  - ☞ wartość zwracana :
    - 0 → w przypadku powodzenia
    - -1 → w przypadku niepowodzenia, ustawia errno :
      - → EBADF → $fildes$ nie jest otwartym deskryptorem
      - → EINTR → funkcja $close()$ została przerwana przez sygnał

- **ssize_t read (int fildes, void* buf, size_t nbyte) [#include<unistd.h>]**
  - ☞ działanie funkcji :
    - funkcja czyta $nbyte$ bajtów z pliku i zapisuje w buforze buf
    - pozycja w pliku jest przesuwana o $nbyte$
      - → jeśli plik kończy się wcześniej, to może być mniej niż $nbyte$ i wtedy funkcja zwraca liczbę bajtów, o którą pozycja się zmieniła
  - ☞ argumenty :
    - $int\ fildes$ → deskryptor pliku, który chcemy czytać
    - $void * buf$ → bufor pamięci, do którego czytamy plik
    - $size\_t\ nbyte$ → ilość bajtów do przeczytania
      - → jeśli $nbyte = 0$ to
  - ☞ wartość zwracana :
    - > 0 → liczba bajtów przeczytanych – w przypadku powodzenia
    - -1 → w przypadku niepowodzenia, ustawia errno :
      - → EBADF → $fildes$ nie jest deskryptorem otwartym do czytania
      - → EIO → błąd I/O
      - → EISDIR → $fildes$ odnosi się do katalogu
      - → EINVAL → $fildes$ wskazuje na obiekt nieodpowiedni do odczytu

---

- **size_t write (int fildes, const void* buf, size_t nbyte) [#include<unistd.h>]**
  - ☞ działanie funkcji :
    - zapisuje do pliku wskazywanego przez $fildes$ maksymalnie $nbyte$ bajtów
    - zapis z bufora buf
  - ☞ argumenty :
    - $int\ fildes$ → deskryptor pliku, do którego chcemy pisać
    - $const\ void * buf$ → bufor pamięci, z którego czytamy kolejne bajty do pliku
    - $size\_t\ nbyte$ → maksymalna ilość bajtów, które piszemy z buf do pliku
  - ☞ wartość zwracana :
    - >0 → ilość zapisanych bajtów
    - 0 → jeśli $nbyte = 0$ a $fildes$ odnosi się do zwykłego pliku
    - -1 → w przypadku błędu , errno :
      - → EBADF → $fildes$ nie jest deskryptorem otwartym do zapisu
      - → ENOSPC → urządzenie zawierające plik $fildes$ nie ma miejsca na dane
      - → EINTR → wywołanie $write()$ przerwane sygnałem

2. SYGNAŁY (man 7 signal)

- **Zadania sygnałów**
  - ☞ Term → domyślne zadanie : zakończyć proces
  - ☞ Ign → domyślne zadanie : ignorować sygnał
  - ☞ Core → domyślne zadanie : zakończyć proces i zrzut obrazu pamięci
  - ☞ Stop → domyślne zadanie : zatrzymanie procesu
  - ☞ Cont → domyślne zadanie : kontynuacja procesu jeśli jest aktualnie zatrzymany
- **Wysyłanie sygnału**
  - ☞ $raise()$ → wysłanie sygnału do samego siebie
  - ☞ $kill()$ → wysłanie sygnału do okreslonego procesu, wszystkich procesów w grupie,systemie
  - ☞ $killpg()$ → wysłanie sygnału do wszystkich procesów w grupie
  - ☞ $pthread\_kill()$ → wysłanie sygnału do określonego wątku POSIX w procesie wywołującym
  - ☞ $tgkill()$ → wysłanie sygnału do określonego wątku w określonym procesie
  - ☞ $sigqueue()$ → wysyłanie sygnałów czasu rzeczywistego
- **Czekanie na wyłapanie sygnału**
  - ☞ $pause()$ → wstrzymuje proces do momentu wyłapania jakiegokolwiek sygnału
  - ☞ $sigsuspend()$ → tymczasowo zmienia maskę sygnału i wstrzymuje process do wyłapania sygnału, który może być obsłużony
- **Signal mask**
  - ☞ sygnał może być zablokowany
    - nie będzie dostarczony dopóki nie zostanie odblokowany
    - w czasie między zablkowaniem a odblokowaniem jest sygnałem oczekującym [*pending signal*]
  - ☞ każdy wątek w procesie ma niezależną maskę sygnałów [*signal mask*]
    - wskazuje zbiór sygnałów, które wątek aktualnie blokuje
  - ☞ można zmieniać maskę sygnałów za pomocą
    - $pthread\_sigmask()$ → w przypadku wątku
    - $sigprocmask()$ → w przypadku jednowątkowej aplikacji
  - ☞ proces potomny utworzony przez $\boldsymbol{fork()}$ dziedziczy maskę sygnałów
    - jego zbiór sygnałów oczekujących jest pusty
  - ☞ wątek może uzyskać zbiór aktualnych sygnałów oczekujących za pomocą funkcji $sigpending()$
- **Standardowe sygnały (dodatkowe materiały – opisy wszystkich sygnałów)**
  - ☞ SIGINT 2 Term → Ctrl+C, wysyłany do procesów pierwszoplanowych
  - ☞ SIGQUIT 3 Core → Ctrl+\, tak jak SIGINT, ale tworzy plik core
  - ☞ SIGKILL 9 Term → niezawodna metoda natycmiastowego zakończenia danego procesu, nie może być przechwycony ani zignorowany
  - ☞ SIGALRM 14 Term → sygnał generowany w momenci upłynięcia czasu określonego w funkcji $alarm()$
  - ☞ SIGTERM 15 Term → wysyłany domyślnie przez polecenie $kill()$
  - ☞ SIGUSR1 30,10,16 Term → sygnał zdefiniowany przez użytkownika
  - ☞ SIGUSR2 31,12,17 Term → sygnał zdefiniowany przez użytkownika
  - ☞ SIGCHLD 20,17,18 Ign → wysłany do procesu macierzystego gdy proces potomny zakończy działanie lub jest zatrzymany
  - ☞ SIGSTOP 17,19,23 Stop → bezwarunkowo zatrzymuje proces, nie może być przechwycony ani zignorowany

3. Czekanie na sygnały

- **Używanie $pause()$**
  - ☞ funkcja $int\ \boldsymbol{pause}(\boldsymbol{void})$ **[#include <unistd.h>]**
    - – wstrzymuje program do otrzymania sygnału, który
      - → powoduje wykonanie funkcji przechwytującej sygnał [*handler function*]
      - → powoduje zakończenie procesu
    - – gdy sygnał dotrze do procesu wywołującego $pause()$ i zostanie przechwycony i obsłużony, $pause()$ zwraca -1
      - → $errno = EINTR$ → funkcja została przerwana przez sygnał
    - – gdy sygnał dotrze do procesu wywołującego $pause()$ i zakończy ten proces, $pause()$ nic nie zwraca
  - ☞ ta funkcja jest odwołaniem wątku w wielowątkowych programach
    - – powstaje problem, gdy ten wątek zajął jakieś zasoby, bo nie są one zwalniane aż do zakończenia programu

---

4. dev/random

- **podstawowe informacje**
  - ☞ wirtualne urządzenie w systemach opracyjnych z rodziny UNIX
  - ☞ pełni funkcję generatora liczb losowych
    - – losowość pochodzi ze sterowników urządzeń i innych źródeł
  - ☞ przy odczycie z $/dev/random$ wygenerowane zostaną przypadkowe bajty
  - ☞ odpowiednie wp rzypadku wymaganej wysokiej przypadkowości danych
  - ☞ dev/random nie nadaje się do generowania wielu liczb losowych w krótkim czasie
    - – lepiej radzi sobie dev/urandom
- **dev/urandom**
  - ☞ generator liczb pseudolosowych
  - ☞ mniejsza przypadkowość danych i odporność na przywidywalność kolejnych danych
  - ☞ traktowany jako plik
    - – otwiera się go funkcją $\boldsymbol{open}()$
    - – zamyka się go funkcją $\boldsymbol{close}()$
    - – czyta się go funkcją $\boldsymbol{read}()$

---

5. Stałe opisujące uprawnienia do plików ($mode\_t$)

- ☞ S_ISUID → ustawia UID na wykonywanie
- ☞ S_ISGID → ustawia GID na wykonywanie
- ☞ S_IRWXU → czytanie, pisanie lub wykonywanie przez właściciela
- ☞ S_IRUSR → czytanie przez właściciela
- ☞ S_IWUSR → pisanie przez właściciela
- ☞ S_IXUSR → wykonywanie przez właściciela
- ☞ S_IRWXG → czytanie, pisanie lub wykonywanie przez grupę
- ☞ S_IRGRP → czytanie przez grupę
- ☞ S_IWGRP → pisanie przez grupę
- ☞ S_IXGRP → wykonywanie przez grupę
- ☞ S_IRWXO → pisanie, czytanie lub wykonywanie przez innych
- ☞ S_IROTH → czytanie przez innych
- ☞ S_IWOTH → pisanie przez innych
- ☞ S_IXOTH → wykonywanie przez innych
- ☞ S_ISVTX → na folderach, ‘flaga ograniczonego usuwania’

6. Makro TEMP_FAILURE_RETRY

- **TEMP_FAILURE_RETRY (expression)**
  - to makro ocenia $expression$ raz i bada jego wartość jako *long int*
  - jeśli wartość $expression = -1$, oznacza to błąd i powinno być ustawione errno
  - wówczas TEMP_FAILURE_RETRY ocenia $expression$ ponownie do czasu aż wynik nie będzie tymczasowym błędem
  - wartość zwracana przez to makro jest wartością zwracaną przez $expression$
- żeby było dostępne, trzeba zdefiniować $\#define\ GNU\_SOURCE$
- trzeba dołączyć $\#include < unistd.h >$

---

7. Przydatne strony :

- **łapanie sygnałów**
  - **https://gist.github.com/aspyct/3462238**
    - przykład łapania sygnałów SIGKILL, SIGALRM, SIGHUP itd.
    - funkcje :
      - → $sigaction() + handle\_signal$
      - → $sigsuspend()$
      - → $alarm()$