SOP LABORATORIA 3

SPIS TREŚCI

1. Wątki

- Definicja
  - pthreads = POSIX threads
  - pojedynczy proces może zawierać wiele wątków, które wykonują ten sam program
    → wątki mogą wykonywać różne części tego programu
  - wątki dzielą pamięć globalną, ale każdy z nich ma też swój własny stos
  - wątki umożliwiają programowi realizację więcej niż jednej operacji na raz
    → na maszynach wieloprocesorowych mogą być wykonywane jednocześnie
    → jądro Linux szereguje wątki asynchronicznie
  - istnieją wewnątrz procesów
- Współdzielona pamięć
  - PID [Process ID]
  - PPID [Parent Process ID]
  - GPID [Group Process ID]
  - Kontrolujący terminal
  - ID użytkownika i grupy [UID i GID]
  - otwarte deskryptory plików
  - akcje dla sygnałów
  - maskę tworzenia pliku [umask]
  - aktywny folder [chdir]
  - folder roota [chroot]
  - priorytet [nice value]
  - limit zasobów [setrlimit]
- Własna pamięć wątku
  - id wątku [thread ID, typ pthread_t]
  - maska sygnałów [pthread_sigmask]
  - zmienna errno
  - stos sygnałów [alternate signal stack]
- Funkcje pthreads
  - Większość z nich zwraca 0 w przypadku powodzenia i numer błędu w razie niepowodzenia
  - Nie ustawiają errno
- ID wątku
  - Każdy wątek ma unikalny identyfikator [Thread ID]
    → jego unikalność zagwarantowana tylko w ramach procesu
    → jeśli wątek zostanie zakończony, jego Thread ID może być nadane innemu wątkowi
  - Typ zmiennej przechowującej Thread ID to pthread_t
  - uzyskuje się :
    → wartość zwracana z $pthread\_create()$
    → każdy wątek może pobrać funkcją $pthread\_self()$

- Funkcje Thread-safe
  - to takie funkcje, które mogą być bezpiecznie wywoływane z różnych wątków w tym samym czasie
  - Lista takich funkcji :

- $asctime()$
- $basename()$
- $catgets()$
- $crypt()$
- $ctermid()$
- $ctime()$
- $dbm\_clearerr()$
- $dbm\_close()$
- $dbm\_delete()$
- $dbm\_error()$
- $dbm\_fetch()$
- $dbm\_firstkey()$
- $dbm\_nextkey()$
- $dbm\_open()$
- $dbm\_store()$
- $dirname()$
- $dlerror()$
- $drand48()$
- $ecvt()$
- $encrypt()$
- $endgrent()$
- $endpwent()$
- $endutxent()$
- $fcvt()$
- $ftw()$
- $gcvt()$
- $getc\_unlocked()$
- $getchar\_unlocked()$
- $getdate()$
- $getenv()$
- $getgrent()$
- $getgrgid()$
- $getgrnam()$
- $gethostbyaddr()$
- $gethostbyname()$
- $gethostent()$
- $getlogin()$
- $getnetbyaddr()$
- $getnetbyname()$
- $getnetent()$
- $getopt()$
- $getprotobyname()$
- $getprotobynumber()$
- $getprotoent()$
- $getpwent()$
- $getpwnam()$
- $getpwuid()$
- $getservbyname()$
- $getservbyport()$
- $getservent()$
- $getutxent()$
- $getutxid()$
- $getutxline()$
- $gmtime()$
- $hcreate()$
- $hdestroy()$
- $hsearch()$
- $inet\_ntoa()$
- $l64a()$
- $lgamma()$
- $lgammaf()$
- $lgammal()$
- $localeconv()$
- $localtime()$
- $lrand48()$
- $mrand48()$
- $nftw()$
- $nl\_langinfo()$
- $ptsname()$
- $putc\_unlocked()$
- $putchar\_unlocked()$
- $putenv()$
- $pututxline()$
- $rand()$
- $readdir()$
- $setenv()$
- $setgrent()$
- $setkey()$
- $setpwent()$
- $setutxent()$
- $strerror()$
- $strsignal()$
- $strtok()$
- $system()$
- $tmpnam()$
- $ttyname()$
- $unsetenv()$
- $wcrtomb()t$
- $wcsrtombs()$
- $wcstombs()$
- $wctomb()$

- Punkty anulowania
  - [cancellation points]
  - wątek może być zakończony w wyniku
    - → zakończenia funkcji wątku
    - → wywołania funkcji $pthread\_exit()$
    - → w wyniku anulowania
  - wątek jest anulowany jeśli inny wątek wywoła $pthread\_cancel()$
  - anulowany wątek zwraca jako wynik specjalną wartość PTHREAD_CANCELED
  - podział wątków
    - → asynchronicznie anulowalne - mogą być przerwane w dowolnej chwili
    - → synchronicznie anulowalne - żądania anulowania są kolejkowane i sprawdzane w tzw. punktach anulowania
    - → nie anulowalne - próby anulowania takiego wątku są ignorowane
  - Domyślnie wątki są synchronicznie anulowalne
  - zmiana typu anulowania wątku za pomocą funkcji $pthread\_setcanceltype()$

– Punkty anulowania :

- $accept()$
- $aio\_suspend()$
- $clock\_nanosleep()$
- $close()$
- $connect()$
- $creat()$
- $fcntl()\ F\_SETLKW$
- $fdatasync()$
- $fsync()$
- $getmsg()$
- $getpmsg()$
- $lockf()\ F\_LOCK$
- $mq\_receive()$
- $mq\_send()$
- $mq\_timedreceive()$
- $mq\_timedsend()$
- $msgrcv()$
- $msgsnd()$
- $msync()$
- $nanosleep()$
- $open()$
- $openat()$
- $pause()$
- $poll()$
- $pread()$
- $pselect()$
- $pthread\_cond\_timedwait()$
- $pthread\_cond\_wait()$
- $pthread\_join()$
- $pthread\_testcancel()$
- $putmsg()$
- $putpmsg()$
- $pwrite()$
- $read()$
- $readv()$
- $recv()$
- $recvfrom()$
- $recvmsg()$
- $select()$
- $sem\_timedwait()$
- $sem\_wait()$
- $send()$
- $sendmsg()$
- $sendto()$
- $sigpause()$
- $sigsuspend()$
- $sigtimedwait()$
- $sigwait()$
- $sigwaitinfo()$
- $sleep()$
- $system()$
- $tcdrain()$
- $usleep()$
- $wait()$
- $waitid()$
- $waitpid()$
- $write()$
- $writev()$

---

2. Mutexy (na pods. **wikipedia**)

- ogólnie
  - Mutex [Mutual Exclusion] → wzajemne wykluczanie
  - mutex to zasób systemowy, który może być w jednym z dwóch stanów :
    - → oblokowany (unlocked) → nie będący w posiadaniu żadnego wątku
    - → zablokowany (locked) → będący w posiadaniu wątku
  - blokada, którą może uzyskać tylko jeden wątek
  - mutexy służą głównie do realizacji sekcji krytycznych
    - → bezpieczny dostęp do zasobów współdzielonych
    - → sekcja krytyczna (**wiki**)
      - ☞ fragment kodu programu, w którym korzyta się z zasobu dzielonego
      - ☞ w danej chwili może być wykorzystywany przez co najwyżej jeden wątek
      - ☞ system operacyjny dba o synchronizację – jeśli więcej wątków żąda wykonania kodu sekcji krytycznej, dopuszczany jest tylko jeden wątek, pozostałe są wstrzymywane
      - ☞ powinna mieć krótki kod (wykonywać się szybko)
      - ☞ zwykle używana, gdy program wielowątkowy musi uaktualnić wiele powiązaych zmiennych, tak żeby inny wątek nie dokonał szkodliwych zmian tych danych
      - ☞ może być użyta, by zagwarantować, że wspólny zasób (np. drukarka) jest używana tylko przez jeden proces w określonym czasie
  - Schemat działania na mutexach :
    - → pozyskanie blokady
    - → modyfikacja lub odczyt współdzielonego obiektu
    - → zwolnienie blokaty
  - Mutex opisywany w $pthreads$ przez strukturę typu $pthread\_mutex\_t$
  - Atrybuty muetxu opisywane przez strukturę typu $pthread\_mutexattr\_t$
- Inicjalizacja i zwalnianie mutexu
  - inicjalizacja :

→ przypisanie makro PTHREAD_MUTEX_INITIALIZER;
☞ $pthread\_mutex\_t\ mutex = PTHREAD\_MUTEX\_INITIALIZER;$
→ wywołanie funkcji $pthread\_mutex\_init()$ która umożliwia podanie atrybutów blokady
– każdy muteź musi zostać zwolniony funkcją $pthread\_mutex\_destroy()$

- Pozyskiwanie i zwolnienie blokady
  – pozyskiwanie blokady :
  → $pthread\_mutex\_lock()$
  → $pthread\_mutex\_trylock()$
  → $pthread\_mutex\_timedlock()$
  – wątek musi zwolnić blokadę funkcją $pthread\_mutex\_unlock()$
- Typy mutexów
  – typ mutexu to jeden z jego atrybutów
  – typy :
  → zwykły [normal]
  → rekursywny [recursive]
  → bezpieczny [error check]
  – ponowne blokowanie
  → sytuacja jak na obrazku :

```
pthread_mutex_lock(&mutex);     // (1)
pthread_mutex_lock(&mutex);     // (2) - w
funkcji pomocniczej
/* ... */
pthread_mutex_unlock(&mutex)    // (3) - w
funkcji pomocniczej
pthread_mutex_unlock(&mutex)    // (4)
```

→ zachowanie :
☞ mutex zwykły → (1) się wykona, na (2) wątek się zatrzyma
→ następuje zakleszczenie [deadlock]
→ sterowanie nie dojdzie do (3) ani (4) nigdy
☞ mutex rekursywny → wykonają się wszystkie funkcje, bo posiada dodatkowy licznik zagnieżdżeń, który z każdym wywołaniem $pthread\_mutex\_lock()$ zwiększa się, a po wykonaniu $pthread\_mutex\_unlock()$ zmniejsza się
→ po osiągnięciu wartości 0 przez licznik blokada jest zwalniana
☞ mutex bezpieczny → drugie wywołanie $pthread\_mutex\_lock()$ zwróci EDEADLK, oznaczający, że wątek już posiada blokadę

3. Funkcje

- $pthread\_create()$ $[\#include < pthread.h >]$
  - **sygnatura:**
    $int\ pthread\_create\ (pthread\_t *\ thread, const\ pthread\_attr\_t\ attr,$
    $void *(* start\_routine\ )(void *), void *\ arg))$
  - **działanie :**
    → funkcja tworzy nowy wątek, który wykonuje się współbieżnie z wątkiem wywołującym funkcję
    → nowy wątek rozpoczyna działanie od funkcji $start\_routine$
    → now wątek kończy działanie poprzez wyjście ze $start\_routine$ lub wywołanie $pthread\_exit()$
    → funkcja kończy działanie natychmiast i wątek wywołujący wznawia działanie od następnej instrukcji
      ☞ w międzyczasie nowy wątek rozpoczyna realizację funkcji wątku
    → wątki przełączane są asynchronicznie, tak więc kod programu nie może zależeć od kolejności wykonania wątków
  - **argumenty**
    → $pthread\_t *\ thread$ → wskaźnik na zmienną, w której będzie zapamiętany thread ID nowego wątku
    → $const\ pthrea\_attr\_t\ attr$ → wskaźnik a zmienną stanowiącą atrybuty wątku lub wartość NULL
      ☞ w przypadku NULL użyte będą domyślne atrybuty
      ☞ obiekt atrybutów jest używany jedynie podczas tworzenia nowego wątku
        1. może być usunięty zaraz po jego utworzeniu lub ponownie użyty do utworzenia kolejnego wątku
      ☞ modyfikowanie obiektu atrybutów nie ma wpływu na atrybuty wcześniej utworzonych wątków
      ☞ obsługiwane atrybuty :
        1. PTHREAD_CREATE_JOINABLE → wątek może być wcielony, po jego zakończeniu zasoby wątku nie są zwalniane aż do chwili, kiedy zostanie jawnie wcielony przez $pthread\_join()$
           → wątek po zakończeniu będzie w stanie przypominającym proces zombie
        2. PTHREAD_CREATE_DETACHED → wątek jest oderwany i nie może być wcielony
           → jego zasoby są zwalniane po jego zakończeniu
    → $void *(* start\_routine)(void *)$ → wskaźnik na funkcję wątku
    → $void *\ arg$ → wartość argumentu wątku
      ☞ przekazywana do funkcji wątku jako jej argument
      ☞ często jest to wskaźnik na pewną strukturę, która zawiera parametry oczekiwane przez funkcję wątku
  - **wartość zwracana**
    → 0 → w prypadku powodzenia
    → $error\ number$ → w przypadku niepowodzenia , ustawia errno (nie ma EINTR) :
      ☞ EAGAIN - system nie miał wystarczająco zasobów do utworzenia nowego wątku

- $pthread\_join()$ $[\#include < pthread.h >]$
  - **sygnatura :**
    $int\ pthread\_join(pthread\_t\ thread, void\ **\ value\_ptr)$
  - **działanie funkcji :**
    → wstrzymuje wykonanie wywołującego ją wątku dopóki $thread$ się nie zakończy
    ☞ jeśli jeszcze nie jest zakończony
    ☞ $thread$ może zakończyć się np.
    1. przez wywołanie $pthread\_exit()$
    2. w punkcie anulowania **(cancellataion point)**

    → wątek na który oczekuje funkcja musi być wątkiem włączalnym
    ☞ nie może być wątkiem oderwanym $pthread\_detach()$
    ☞ nie może być wątkiem utworzonym $pthread\_create()$ z $PTHREAD\_CREATE\_DETACHED$

    → zasoby pamięciowe kończonego wątku (deskryptor wątku oraz stos) nie są usuwane do chwili, aż inny wątek go nie wcieli przez $pthread\_join()$

    → dla każdego nieoderwanego wątku powinna być wywołana raz funkcja $pthread\_join()$ aby uniknąć wycieków pamięci
  - **argumenty :**
    → $pthread\_t\ thread$ → thread ID wątku, na który funkcja ma oczekiwać
    → $void\ **\ value\_ptr$ → zostanie ustawiony na wartość zwracaną przez zakończony wątek
    ☞ może zawierać adres struktury zawierającej wyniki działania wątku
    ☞ może być rzutowana na zmienną określonego małego typu
    ☞ jeśli podamy NULL to nic nie zawiera
    ☞ jest to wartość przekazywana przez kończący się wąte do funkcji $pthread\_exit()$ lub $PTHREAD\_CANCELED$ jeśli wątek został przerwany
  - **wartość zwracana :**
    → 0 → w przypadku powodzenia
    → $error\ number$ → w przypadku niepowodzenia

- $rand\_r()$ $[\#include < stdlib.h >]$
  - **sygnatura :**
    $int\ rand\_r\ (unsigned\ *\ seed);$
  - **działanie funkcji :**
    → pseudolosowy generator liczb z przedziału $[0, RAND\_MAX]$
    → wywoływana z tą samą wartością początkową $seed$ i nie modyfikuje $seed$
    → działa jak rand, ale ma argument (ziarno jak $srand()$)
  - **argumenty :**
    → $unsigned * seed$ → wartość początkowa dla generatora
  - **wartość zwracana :**
    → zwraca pseudolosową liczbę całkowitą z przedziału $[0, RAND\_MAX]$

- $pthread\_mutex\_destroy()$ $[\#include < pthread.h >]$
  - **sygnatura :**
    $int\ pthread\_mutex\_destroy\ (\ pthread\_mutex\_t\ *\ mutex\ )$
  - **działanie funkcji :**
    → usuwa obiekt mutexa, zwalniając ewentualnie zajmowane przez mutex zasoby
    → usuwany mutex musi być w stanie odblokowanym
    ☞ w przeciwnym razie niezdefiniowane zachowanie
    → w efekcie obiekt mutexa jest niezainicjalizowany
    ☞ w niektórych implementacjach może być $invalid\ value$
  - **argumenty**
    → $pthread\_mutex\_t\ *\ mutex$ → usuwany mutex
  - **wartość zwracana :**
    → 0 → w przypadku powodzenia
    → $error\ number$ → w przypadku niepowodzenia

- $pthread\_mutex\_lock()$ $[\#include < pthread.h >]$
  - **sygnatura**
    $int\ pthread\_mutex\_lock\ (pthread\_mutex\_t\ *\ mutex)$
  - **działanie funkcji :**
    → blokuje wskazany mutex
    → jeśli mutex jest aktualnie wolny, zostanie zajęty przez wątek wywołujący funkcje i stanie się własnością tego wątku
    ☞ funkcja powróci niezwłocznie
    → jeśli mutex jest już zajęty przez inny wątek, funkcja zatrzyma wywołujący wątek do chwili, aż mutex zostanie zwolniony i będzie możliwe jego zajęcie
    → jeśli mutex jest już zajęty przez wątek wywołujący to zachowanie funkcji zależy od **typu mutexa**

Mutex Type	Robustness	Relock	Unlock When Not Owner
NORMAL	non-robust	deadlock	undefined behavior
NORMAL	robust	deadlock	error returned
ERRORCHECK	either	error returned	error returned
RECURSIVE	either	recursive (see below)	error returned
DEFAULT	non-robust	undefined behaviort	undefined behaviort
DEFAULT	robust	undefined behaviort	error returned

  - **argumenty :**
    → $pthread\_mutex\_t\ *\ mutex$ → zajmowany mutex
  - **wartość zwracana :**
    → 0 → w przypadku powodzenia
    → $error\ number$ → w przypadku niepowodzenia

- $pthread\_mutex\_unlock()$ $[\#include < pthread.h >]$
  - **sygnatura :**
    $int\ pthread\_mutex\_unlock\ (pthread\_mutex\_t\ *\ mutex\ )$
  - **działanie funkcji :**
    → oblokowuje mutex (przeczytaj działanie **pthread_mutex_lock()**)
  - **argumenty :**
    → $pthread\_mutex\_t\ *\ mutex$ → mutex, który ma zostać zwolniony
  - **wartość zwracana :**
    → 0 → w przypadku powodzenia
    → $error\ number$ → w przypadku niepowodzenia

- $pthread\_detach()$ $[\#include < pthread.h >]$
  - **sygnatura**
    $int\ pthread\_detach\ (\ pthread\_t\ thread)$
  - **działanie funkcji :**
    → odłącza wątek (przełącza wątek w stan odłączony)
    → zasoby wątku (stos) zostaną natychmiast zwolnione po zakończeniu wątku
    → odłączenie uniemożliwia innym wątkom synchronizację poprzez wywołanie $\boldsymbol{pthread\_join()}$ na zakończenie tego wątku
    → po poprawnym wykonaniu funkcji kolejne wykonania $pthread\_join()$ na tym wątku zawiodą
    → jeżeli inny wątek już wciela (oczekuje na zakończenie wątku poprzez $pthread\_join()$) wątek próbujący się odłączyć pozostanie w stanie nie odłączonym a funkcja nie wykona żadnej operacji
  - **argumenty**
    → $pthread\_t\ thread$ → wątek, który ma zostać odłączony
  - **wartość zwracana :**
    → 0 → w przypadku powodzenia
    → $error\ number$ → w przypadku niepowodzenia

- $pthread\_attr\_init()$ $[\#include < pthread.h >]$
  - **sygnatura :**
    $int\ pthread\_attr\_init\ (pthread\_attr\_t\ *\ attr)$
  - **działanie funkcji**
    → funkcja inicjalizuje obiekt atrybutów wątku [thread attribute object] i wypełnia go wartościami domyślnymi atrybutów
    ☞ dołączalność
    1. wątek może być włączalny lub odłączony
    2. $PTHREAD\_CREATE\_JOINABLE$ → wątek dołączalny [domyślny]
    3. $PTHREAD\_CREATE\_DETACHED$ → wątek jest odłączony

    ☞ sposób szeregowania
    1. wątek może być przełączany przez planistę (scheduler) według kilku algorytmów
    2. $SCHED\_OTHER$ → strategia równomiernego wykorzystania procesora [domyślny]
    3. $SCHED\_RR$ → strategia czasu rzeczywistego przełączania w kółko ze stałym interwałem (round-robin)
    4. $SCHED\_FIFO$ → strategia czasu rzeczywistego typu first in-first out

    ☞ priorytet szergowania
    1. strategie czasu rzeczywistego uwzględniają priorytety wątków podczas szeregowania

- **argumenty :**
  → $pthread\_attr\_t * attr$ → wskaźnik na zmienną, która ma być zainicjowana domyślnymi wartościami atrybutów wątku
- **wartość zwracana :**
  → 0 → w przypadku powodzenia
  → $error\ number$ → w przypadku niepowodzenia

- $pthread\_attr\_destroy()$ $[\#include < pthread.h >]$
  - **sygnatura :**
    $int\ pthread\_attr\_destroy(pthread\_attr\_t * attr)$
  - **działanie funkcji**
    → usuwa obiekt atrybutów
    → obiekt ten nie może być użyty bez ponownej inicjalizacji
  - **argumenty :**
    → $pthread\_attr\_t\ attr$ → wskaźnik na usuwany obiekt atrybutów
  - **zwracana wartość**
    → 0 → w przypadku powodzenia
    → $error\ number$ → w przypadku niepowodzenia

- $pthread\_attr\_setdetachstate()$ $[\#include < pthread.h >]$
  - **sygnatura :**
    $int\ pthread\_attr\_setdetachstate\ (\ pthread\_attr\_t * attr, int\ detachstate\ )$
  - **działanie funkcji :**
    → ustawia atrybut okreslający czy wątek jest oderwany
    → wątek oderwany nie może być wcielony przez ***pthread_join()***
    → domyślnie wątek jest możliwy do wcielenia (PTHREAD_CREATE_JOINABLE)
  - **argument :**
    → $pthread\_attr\_t * attr$ → wskaźnik na obiekt atrybutów wątku
    → $int\ detachstate$ → możliwe wartości :
    ☞ $PTHREAD\_CREATE\_JOINABLE$
    ☞ $PTHREAD\_CREATE\_DETACHED$
  - **wartość zwracana :**
    → 0 → w przypadku powodzenia
    → $error\ number$ → w przypadku niepowodzenia

- $pthread\_attr\_getdetachstate()$ $[\#include < pthread.h >]$
  - **sygnatura**
    $int\ pthread\_attr\_getdetachstate(const\ pthread\_attr\_t * attr, int * detachstate)$
  - **działanie funkcji**
    → funkcja pobiera atrybut określający czy wątek może być wcielony czy też jest oderwany
  - **argumenty :**
    → $pthread\_attr\_t * attr$ → wskaźnik na obiekt atrybutów wątku
    → $int * detachstate$ → wskaźnik na zmienną, w której zostanie zapisany atrybut określający, czy wątek jest oderwany
  - **wartość zwracana**
    → 0 → w przypadku powodzenia
    → $error\ number$ → w przypadku niepowodzenia

- $pthread\_sigmask()$ $[\#include < signal.h >]$
  - **sygnatura**

    $int\ pthread\_sigmask(\ int\ how, cons\ sigset\_t * set, sigset\_t * oset)$
  - **działanie funkcji :**
    - → funkcja zmienia maskę sygnałów dla wywołującego ją wątku
    - → zmienia zgodnie z argumentami how i set
    - → jeżeli $oset$ nie jest NULL, to poprzednia maska sygnałów jest zapisywana w $oset$
  - **argumenty :**
    - → int how → określa w jaki sposób ustawiana 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 → w przypadku powodzenia
    - → $error\ number$ → w przypadku niepowodzenia

- $sigprocmask\ ()$ $[\#include < signal.h >]$
  - **sygnatura**

    $int\ sigprocmask\ (\ int\ how, const\ sigset\_t * set, sigset\_t * oset)$
  - **działanie funkcji**
    - → funkcja modyfikuje bieżącą maskę procesu
  - **arugmenty :**
    - → $int\ how$ → 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
    - → $const\ sigset\_t * set$ → maska sygnałów modfikująca aktualną maskę sposobem how
    - → $sigset\_t\ oset$ → 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

- $sigwait()$ $[\#include < signal.h >]$
  - **sygnatura**
    $int\ sigwait\ (const\ sigset\_t\ *\ set, int\ *\ sig\ )$
  - **działanie funkcji**
    → wstrzymuje wykonanie wątku do chwili aż jeden z sygnałów w masce set nie zostanie dostarczony do tego wątku
    → zachowuje numer otrzymanego sygnału w zmiennej sig i powraca
    → sygnały w zestawie muszą być zablokowane i nie mogą być ignorowane przed wywołaniem $sigwait()$
    → jeśli dostarczony sygnał posiada obsługujący go dołączony handler, nie jest on wywoływany
    → aby funkcja działała poprawnie, sygnały na które czeka muszą być zablokowane we wszystkich wątkach, nie tylko w wywołującym wątku
    ☞ w przeciwnym przypadku nie ma gwarancji, że wątek otrzyma sygnał
    → jeśli kilka wątków czeka na ten sam sygnał, nie więcej niż jeden z nich powinien wrócić z $sigwait()$ z numerem tego sygnału
    ☞
  - **argumenty**
    → $const\ sigset\_t\ *\ set$ → maska sygnałów, na które oczekuje funkcja
    → $int\ *\ sig$ → wskaźnik na zmienną, w której ma być umieszczony numer sygnału
  - **wartość zwracana**
    → 0 → w przypadku powodzenia
    → $error\ number$ → w przypadku niepowodzenia
    ☞ EINVAL → set zawiera błędną maskę sygnałów

- $pthread\_cancel()$ $[\#include < pthread.h >]$
  - **sygnatura :**
    $int\ pthread\_cancel(\ pthread\_t\ thread)$
  - **działanie funkcji**
    → żądanie anulowania wątku $thread$
    → czy anulowanie się uda zależy od typu i stanu wątku
    → gdy dochodzi do anulowania są uruchamiane handlery [cleanup hadnlers]
    ☞ po ich wykonaniu wywoływany jest destruktor wątku i jest on kończony
  - **argumenty**
    → $pthread\_t\ thread$ → wątek, który ma być anulowany
  - **wartość zwracana :**
    → 0 → w przypadku powodzenia
    → $error\ number$ → w przypadku niepowodzenia

- $pthread\_cleanup\_push()$ $[\#include < pthread.h >]$
  - **sygnatura**
    $void\ pthread\_clenup\_push\ (\ void\ (*routine)(void\ *), void\ *arg)$
  - **działanie funkcji**
    → dołącza wskazaną funkcję $routine$ z argumentem arg jako cleanup handler
    → od tego miejsca, aż do odłączenia handlera poprzez $pthread\_cleanup\_pop()$ funkcja $routine$ będzie wywoływana kiedy wątki będą kończone lub przerywane (anulowane)
    → jeżeli aktywnych jest kilka handlerów, będą one wywoływane w kolejności LIFO
      ☞ ostatni podłączony handler będzie wywołany jako pierwszy
  - **argumenty**
    → $void(*routine)(void\ *)$ → funkcja, która staje się handlerem
    → $void\ *\ arg$ → argument funkcji $routine$

- $pthread\_cleanup\_pop()$ $[\#include < pthread.h >]$
  - **sygnatura :**
    $void\ pthread\_cleanup\_pop(int\ execute)$
  - **działanie funkcji**
    → usuwa ze stostu handlerów wątku pierwszą funkcję (ostatnio dodaną)
    → jeśli $execute$ jest niezerowe, to opcjonalnie wykonuje tą funkcję
  - **argumenty**
    → $int\ execute$ → argument opcjonalny, jeśli $execute \neq 0$ to wykonywany jest usuwany ze stosu handler

---

- $clock\_getres()$ $[\#include < time.h >]$
  - **sygnatura**
    $int\ clock\_getres\ (\ clockid\_t\ clock\_id, struct\ timespec\ *res)$
  - **działanie funkcji**
    → funkcja pobiera rozdzielczość (dokładność) zegara $clock\_id$
      ☞ jeśli res nie jest zerowa to zapisuje ją w res
    → rozdzielczości zegarów zależą od implementacji
      ☞ nie mogą być ustawiane przez poszczególne procesy
  - **argumenty**
    → $clockid\_t\ clock\_id$ → identyfikator zegara
      ☞ $CLOCK\_REALTIME$ → globalny zegar czasu rzeczywistego
      ☞ $CLOCK\_MONOTONIC$ → zegar, które nie może być ustawiony, reprezentujący czas od pewnej niezdefiniowanej chwili
      ☞ $CLOCK\_PROCESS\_CPUTIME\_ID$ → wysokorozdzielczy zegar z CPU o zasięgu procesu
      ☞ $CLOCK\_THREAD\_CPUTIME\_ID$ → zegar CPU o zasięgu wątku
    → struct timespec* res → wskaźnik na bufor, w którym zostanie zapisana rozdzielczość zegara
  - **wartość zwracana**
    → 0 → w przypadku powodzenia
    → -1 → w przypadku niepowodzenia, ustawia errno :
      ☞ EINVAL → $clock\_id$ nieprawidłowy
      ☞ EOVERFLOW → liczba sekund nie pasuje to obiektu typu $time\_t$

- $clock\_settime()$ $[\#include < time.h >]$
  - **sygnatura**
    $int\ clock\_settime(\ clockid\_t\ clock\_id, const\ struct\ timespec\ *\ tp\ )$
  - **działanie funkcji :**
    → ustawia czas określonego zegara $clock\_id$
  - **argumenty :**
    → $clockid\_t\ clock\_id$ → zegar do ustawienia (patrz $\boldsymbol{clock\_getres()}$)
    → $struct\ timespec\ *\ tp$ → czas do ustawienia
  - **wartość zwracana**
    → 0 → w przypadku powodzenia
    → -1 → w przypadku niepowodzenia

- $clock\_gettime()$ $[\#include < time.h >]$
  - **sygnatura**
    $int\ clock\_gettime(\ clockid\_t\ clock\_id, struct\ timespec\ *\ tp\ )$
  - **działanie funkcji**
    → pobiera czas określonego zegara $clock\_id$
  - **argumenty :**
    → $clockid\_t\ clock\_id$ → identyfikator zegara (patrz $\boldsymbol{clock\_getres()}$)
    → $struct\ timespec\ *\ tp$ → struktura, w której zostanie odłożony aktualny czas zegara
  - **wartość zwracana**
    → 0 → w przypadku powodzenia
    → -1 → w przypadku niepowodzenia
