Rozdział 1

Wątki w Linuxie

1.1 Autorzy

Przemyslaw Ciok
Piotr Michalczyk

1.2 Wstęp

1.2.1 Co to są wątki?

Wątki (*threads*) są nazywane tzw. "lekkimi procesami" w przeciwieństwie do normalnych, tzw. "ciężkich" procesów. Różnica bierze się z narzutu systemowego potrzebnego do przełączania kontekstu oraz komunikacji między procesami: "ciężkie" procesy składają się z kilku części: kodu programu, danych, stosu, tablicy plików i tablicy sygnałów. Przy wywoł aniu funkcji `fork`, przekazywany jest właściwie tylko kod procesu. Za każdym razem trzeba więc zapamiętać wartości wszystkich atrybutów procesu oraz stan procesora. Kolejną wadą "ciężkich" procesów jest konieczność użycia specjalny ch mechanizmów (np. pamięci dzielonej) przy dostępie do wspólnych zmiennych, co zwiększa narzut czasowy. W przeciwieństwie do typowych, "ciężkich" procesów, wątki redukują narzut systemowy przez współdzielenie swoich części (pamięć, tablica plików). Właściwie przy tworzeniu nowego wątku nie jest przekazywany jedynie stan rejestrów i stos. Zasadniczo, ze względu na sposób implementacji, wyróżniamy dwa typy wątków: **wątki poziomu użytkownika** (*user-space threads*) i **wątki poziomu jądra** (*kernel-space threads*).

1.2.2 Wątki poziomu użytkownika (`1-n`)

Wątki poziomu użytkownika same zarządzają procesem wykonywania. Unixowy proces staje się "nadzorcą" i "tworzy" kolejne "wątki", między którymi będzie się odbywać przekazywanie sterowania (jednemu procesowi unixemu odpo wiada `n` wątków). Zazwyczaj każdy wątek "rezygnuje" z wykonywania poprzez samodzielne oddanie procesora (wysłanie odpowiedniego sygnału) albo zostaje "wywłaszczony" (po odebraniu odpowiedniego sygnału). Z założen ia operacja taka jest szybsza niż wymiana z poziomu jądra. Efektem ubocznym jest to, że jądro nie może traktować poszczególnych wątków "osobno", gdyż nie wie o ich istnieniu. Możliwe jest więc że jeden zablokowany wątek (np. przy blokuj ącym wywołaniu systemowym) zablokuje cały proces. Innym mankamentem jest brak możliwości przypisania wątków różnym procesorom (w systemie wieloprocesorowym), co uniemożliwia równoległe wykonywanie. Różne biblioteki w różny sposób próbują obchodzić te pro blemy np. poprzez nadpisywanie blokujących funkcji systemowych.

1.2.3 Wątki poziomu jądra (1-1)

Realizowane poprzez jądro - każdy wątek jest osobnym unixowym procesem (stąd oznaczenie - jednemu procesowi unixowemu odpowiada jeden wątek), dzieli przestrzeń adresową z innymi wątkami (jest to możliwe poprzez odpowiednie parametry wywołania funkcji t̂ clone()). Za przełączanie kontekstu między wątkami odpowiedzialne jest jądro analogicznie do zarządzania "normalnymi" procesami. Umożliwia to efektywne wydziedziczanie wątków, rozwiązuje problemy z operacjami wejścia - wyjścia. Możl iwe jest również korzystanie z równoległości w systemach wieloprocesorowych. Mechanizm przełączania kontekstu jest wiec intuicyjnie "płynniejszy" poprzez zastosowanie mechanizmu przełączania kontekstu typowego dla jądra. Niestety konieczność przełączania procesora w tryb jądra powoduje ze narzut czasowy jest większy, mniej efektywne jest pod tym względem także tworzenie wątków. Obsługa wątków jest wbudowana w jądro od czasów 2.1 (umożliwia tworzenie procesów współdzieląc ych pewne atrybuty, ale nie zapewnia mechanizmów synchronizacji między wątkami) Wątki poziomu jądra są wspomagane już od wersji `1.3.56`. Obecnie obsługa wątków jest w pełni zintegrowana z jądrem, a standardem stał się pakiet *LinuxThreads*.

1.2.4 Wątki pośrednie (m-n)

Spotykane są tez implementacje będące kombinacja obu rozwiązań: jądro tworzy "lżejsze" od "ciężkich" procesy, które z kolei zawierają w sobie wątki poziomu użytkownika. W przypadku blokady którejś grupy w ramach jedneg o procesu, może być tworzony nowy "lekki" proces pozwalający na jej ominięcie. Taka implementacja umożliwia również efektywniejsze niż. "czyste" wątki użytkownika wykorzystanie wielu procesorów. Ceną za te nowe możliwo ści jest dłuższy czas przełączania kontekstu niż przy tradycyjnych watkach użytkownika. Przy tym rozwiązaniu, `m` procesom unixowym odpowiada `n` wątków (`m`<`n`)

1.2.5 Popularne biblioteki z obsługą wątków

W powyższej tabeli zostały zebrane informacje na temat kilkunastu dostępnych bibliotek obsługujących wątki[1]. Wszystkie podane wyżej linki zostały sprawdzone i wskazują na źródła bądź też na stronę z dokumentacj ą. Obecnie najpopularniejszą implementacją jest biblioteka wątków jądra *LinuxThreads*, która praktycznie stała się standardem (także w wydziałowym laboratorium). Biblioteka ta jest w pełni zgodna z aktualną wersją biblioteki `libc`. Spośród bibliotek wątków użytkownika najpopularniejszą wydaje się być *PCThreads*. Obie są zgodne ze standardem *IEEE Posix 1003.c* [2], więc z wyjątkiem nieprzenośnych funkcji nie różnią się interfejsem. W dalszej części zostaną opisane struktury danych i algorytmy charakterystyczne dla tej pierwszej biblioteki.

[1] opracowane na podstawie ThreadsFAQ
[2] Specyfikacja jest, niestety, dostępna za odpłatnością

Tablica 1.1: Biblioteki wątków dostępne dla Linuxa

Biblioteka	Rodzaj	Skąd ściągnąć?
Bare-Bones Threads	jądra	ftp://caliban.physics.utoronto.ca/pub/linux/bb_threads.tar.gz
CLThreads	jądra	ftp://lin.fsid.cvut.cz/pub/linux/clthreads-0.1.1.tgz
FSU PThreads	użytkownika	http://www.cs.fsu.edu/ mueller/projects.html
LinuxThreads	jądra	http://pauillac.inria.fr/ xleroy/linuxthreads/
LWP	użytkownika	ftp://gummo.doc.ic.ac.uk/rex/lwp.tar.gz
NThreads	użytkownika	ftp://ftp.tu-chemnitz.de/pub/Local/informatik/linux/nthreads.tgz
PC Threads	użytkownika	http://www.aa.net/ mtp/PCThreads.hmtl

1.2.6 Trudności związane z programowaniem wielowątkowym

- **Problem z obsługą sygnałów** - nie ma możliwości ustawienia osobnej obsługi sygnałów dla każdego z wątków. Wysyłanie sygnałów do poszczególnych wątków jednak wymaga stworzenia specjalnego dodatkowego wątku, który w sposób synchroniczny oczekiw ałby na sygnał (`sigwait`) i dostarczał obsługę wszystkich sygnałów.

- **Problem z bezpieczeństwem i wielowejściowością** - aby kod był **bezpieczny** (*MT-safe*), powinien zapewniać przyzwoitą współbieżność. **Wielowejściowość** (*reentrant*) polega na tym, że bezpieczne musi być wielokrotne wchodzenie do tego samego kodu - kod nie może używać zmiennych globalnych bądź statycznych bez odpowiednich środków bezpieczeństwa (sekcja krytyczna; operacje atomowe). Standardowe funkcje nie zawsze to zapewniają. Najprostszym rozwiązaniem jest otoczenie kodu mute xem, który zapewni, ze tylko jeden watek na raz będzie wykonywał dany kod. Rozwiązanie to jest niesamowicie nieefektywne, dlatego tworzone są biblioteki dostarczające standardowych funkcji dla wątków (bezpieczne; zapewniające wielowejściowość) rozróżnion e od tradycyjnych przez dodanie sufiksu "`_r`" (od reentrant).

- **Problem z rozpoznawaniem wątków** - przy synchronicznym kończeniu wątków nie jest możliwe sprawdzanie relacji ojciec-syn, która zachodzi dla zwykłych procesy. Konieczne jest trzymanie informacji o liczbie aktywnych wątków lub tworzenie wątków char'¨rozdzielonych" (detached threads) i zmuszenie ich do zmniejszania licznika tuz przed końcem ich wykonania.

- **Problem przy wywołaniu `fork()`** - mogą powstać problemy związane z tym ze w procesie potomnym istnieje jedynie watek, który wywołał ta funkcje (a zawartość pamięci łącznie z informacja o mutexach jest kopiowana zawartością pamięci rodzica). Jeśli mutex byłby zajęty przez inny watek niż ten który wywołał `fork()`, w wątku potomnym mutex ten nigdy nie zostałby odblokowany. Dlatego przed wywołaniem funkcji `fork()` należy zadbać o odpowiednie operacje na mutexach.

- **Ograniczenia systemowe** - należy pamiętać o stałej `NR_TASKS` która ogranicza ilość wątków (w architekturze `x86` nie może przekraczać `4090`). Mogą także pojawić się problemy przy przydzielaniu pamięci na stos, gdyż ilość pamięci przeznaczonej na stosy procesów również jest ograniczona.

- **Debugging wątków** - z powodu istnienia nierównoważnych implementacji wątków, nie powstał dotąd żaden program umożliwiający bezpośrednie obserwowanie działania poszczególnych wątków. Poza tradycyjna metoda "printf", pozostaje jeszcze wykorzystanie narzędzi takich jak gdb (wątki na razie ze względu na napotkane trudności implementacyjne nie dzielą PID z rodzicem). Można także próbować badać zrzut pamięci (plik `core`)

1.3 Biblioteka Linux Threads

1.3.1 Funkcje obsługujące wątki

Poniżej przedstawione są główne funkcje operujące na wątkach, przydatne do właściwego korzystania z tego mechanizmu. Przytoczone opisy tych funkcji mają jedynie nakierować na to co właściwie one robią. Dla głębszego ich poznania polecam podręcznik MAN. Podstawowe algorytmy wykorzystane w ważniejszych funkcjach zostaną przedstawione na dalszych stronach.

Tworzenie wątku

```
int pthread_create ( pthread_t *thread, pthread_attr_t *attr,
                     void *(*start_routine)(void *) , void *arg);
```

Opis: Funkcja tworzy nowy wątek, ktory działa równolegle z aktualnym wątkiem.)

Ustawianie atrybutów wątku

```
int pthread_attr_init (pthread_attr_t *attr);
int pthread_attr_destroy (pthread_attr_t *attr);
```

Opis: Inicjalizuje/usuwa obiekt `attr`.

```
int pthread_attr_setdetachstate (pthread_attr_t *attr, int detachstate);
int pthread_attr_getdetachstate (const pthread_attr_t *attr, int *detachstate);
```

Opis: Ustawia/Odczytuje atrybut `detachstate`, który określa czy wątek ma być utworzony w stanie umożliwiajacym dołączenie czy odłącznym.

```
int pthread_attr_setschedpolicy (pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy (const pthread_attr_t *attr, int *policy);
```

Opis: Ustawia/Odczytuje atrybut `schedpolicy`, który, umożliwia wybór sposobu przełączania dla wątku. Mozna wybrać:

- SCHED_OTHER (normalne, szeregowanie nie w czasie rzeczywistym)
- SCHED_RR (w czasie rzeczywistym, round-robin)
- SCHED_FIFO (w czasie rzeczywistym, FIFO).

Wartością domyslną jest `SCHED_OTHER`. Sposoby `SCHED_RR` i `SCHED_FIFO` są dostępne jedynie dla procesów z uprawnieniami superużytkownika.

```
int pthread_attr_setschedparam (pthread_attr_t *attr,
                                const struct sched_param *param);
int pthread_attr_getschedparam (const pthread_attr_t *attr,
                                struct sched_param *param);
int pthread_setschedparam (pthread_t target_thread, int policy, const struct
                           sched_param *param);
int pthread_getschedparam (pthread_t target_thread, int *policy, struct
                           sched_param *param);
```

Opis: Ustawia/Odczytuje atrybut `schedparam`, który, zawiera parametry do przełączania wątku (priorytet brany pod uwage tylko gdy wątek działa w trybie rzeczywistym, czyli szeregowanie `SCHED_RR` lub `SCHED_FIFO`).

Przyłączanie, odłączanie, zakończenie wątku

Aby można było korzystać z wyników wyliczonych przez jeden wątek wewnątrz drugiego wątku istnieje możliwosc dołączania wątku przez przez inny wątek. Aby wątek mógł zostać przyłączony musi mieć w polu `p_detached` (w wewnętrznej strukturze `__pthread_descr_struct`) ustawioną możliwosc przyłączania. Jeśli wątek posiadający możliwość przyłączania zakończy działanie zasoby tego wątku nie zostają zwolnione lecz czekają na przyłączenie przez inny wątek. Dlatego też, wszystkie wątki utworzone z opcją `PTHREAD_CREATE_JOINABLE` powinny zostać przyłączone.

```
int pthread_join (pthread_t th, void **thread_return);
```

Opis: Przyłącza wątek (opis algorytmu w dalszej części). Oczekiwanie przez wywołanie funkcji `pthread_join` jest tzw. `cancellation-point`.

```
int pthread_detach (pthread_t th);
```

Opis: Funkcja `pthread_detach` odłącza wołający wątek, co gwarantuje zwolnienie zasobów pamięci natychmiast po zakończeniu działania wątku. Funkcja ta wykona się poprawnie jeśli na wątek nikt nie czeka.

```
void pthread_exit (void *retval);
```

Opis: Funkcja `pthread_exit` kończy działanie wołającego wątku. W `retval` ustawiona zostaje wartość dla `join`.

Funkcje czyszczące po zakończonym wątku (cleanup handlers)

Funkcje te są wołane gdy wątek kończy swoje działanie zarowno przez `pthread_exit` jak i przez `cancellation`. Zadaniem ich jest zwolnienie zasobów jakie mogą być w posiadaniu wątku w chwili gdy jest przerywany (np. gdy wątek zamknie `mutex` i przed jego zwolnieniem zostanie przerwany to inne wątki wiszące na tym mutexie zostaną zablokowane, aby tego uniknąć należy przed zamknięciem mutexa zainstalować `cleanup handler`, który zwolni mutex gdy wątek zostanie przerwany.

```
void pthread_cleanup_push ( void (*routine) (void *), void *arg);
```

Opis: Instaluje funkcje `routine` z argumentem `arg` jako `cleanup handler`. Jeśli wiele `cleanup handlers` jest aktywnych wołane sa one w kolejnosci LIFO.

`void pthread_cleanup_pop (int execute);`

Opis: Usuwa ostatnio zainstalowanego `cleanup handlera` i uruchamia go jeśli `execute<>0`

Odwoływanie innego wątku

Odwołanie (**cancellation**) jest mechanizmem, za którego pomocą jeden wątek moze zakończyć działanie innego wątku. Dokładniej wątek może wysłać zadanie odwołania do innego wątku. Zależnie od ustawień wątek, do którego wysłano takie zadanie może je zignorować, uwzględnić (zakończyć swoje działanie) natychmiast lub odwlec zakończenie aż do osiągnięcia pierwszego punktu zwanego **cancellation point**. Jeśli wątek uwzględnia zadanie odwołania, zachowuje się tak jakby wywołana była funkcja `pthread_exit`:
wszystkie funkcje czyszczące zostają uruchomione, wywoływane sa funkcje kończące i ostatecznie zwracana jest wartość `PTHREAD_CANCELED`.

`int pthread_cancel (pthread_t thread);`

Opis: Funkcja ta wysyła zadanie odwołania do wątku `thread`.

`int pthread_setcancelstate (int state, int *oldstate);`

Opis: Funkcja ta ustawia rodzaj reakcji na `cancellation` (akceptacja/ignorowanie)

`int pthread_setcanceltype (int type, int *oldtype);`

Opis: Funkcja zmienia typ reakcji na zadanie odwołania (przerywanie pracy natychmiast/ przerywanie pracy po dotarciu do **cancellation point**).
Punkty **cancellation points** sa miejscami w programie, gdzie wykonywany jest test, czy przypadkiem nie przyszło zadanie odwołania danego wątku i w razie pozytywnego rezultatu, odwołanie wątku jest wykonywane. Standard POSIX ustala następujące funkcje jako **cancellation points**:

- pthread_join()
- pthread_cond_wait()
- pthread_cond_timedwait()
- pthread_testcancel()
- sem_wait()
- sigwait()

`void pthread_testcancel(void);`

Opis: Funkcja przeprowadza test, czy nie ma zadania odwołania i odpowiednio reaguje w razie jego istnienia.

`void pthread_kill_other_threads_np(void);`

Opis: Jest to nieprzenośna funkcja z biblioteki LinuxThtreads. Wywołanie jej powoduje zakończenie wszystkich wątkow dla danego programu, poza wątkiem wywołującym funkcję.

Przechwytywanie sygnałów w wątkach

```
int pthread_sigmask(int how, const sigset_t *newmask, sigset_t *oldmask);
```

Opis: Zmienia maskę sygnałową dla wątku `oldmask` zostanie nadpisany `newmask`.

```
int pthread_kill (pthread_t thread, int signo);
```

Opis: Wysyła sygnał `signo` do wątku `thread`.

```
int sigwait (const sigset_t *set, int *sig);
```

Opis: Zawiesza wątek w oczekiwaniu na sygnał wymieniony w `set`.

Mutexy w wątkach

```
int pthread_mutex_init (pthread_mutex_t *mutex,
                        const pthread_mutexattr_t *mutexattr);
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

Opis: Funkcja inicjalizuje/usuwa mutex (w `mutexattr` atrybuty mutexa).

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
int pthread_mutex_trylock (pthread_mutex_t *mutex);
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

Opis: Funkcja (zajmuje blokująco/nieblokująco)/(zwalnia) mutex (działa w standardowy sposób)

```
int pthread_mutexattr_init (pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy (pthread_mutexattr_t *attr);
```

Opis: Tworzy/usuwa `attr` ustawiając mu domyslne wartości.

```
int pthread_mutexattr_setkind_np (pthread_mutexattr_t *attr, int *kind);
int pthread_mutexattr_getkind_np (const pthread_mutexattr_t *attr, int *kind);
```

Opis: Ustawia/Odczytuje atrybut `kind` z `attr` pobierając/wstawiając go z/na `*kind`. Domyślnie ustawiony jest na `fast`.

Zmienne Conditions w wątkach

Zmienne warunkowe są metoda synchronizacji umożliwiającą zawieszenie działania i odstąpienie czasu procesora aż do momentu, w którym pewien warunek (umieszczony w pamięci dzielonej) nie zostanie spełniony. Najprostszymi operacjami na zmiennych warunkowych sa: zasygnalizowanie spełnienia warunku i oczekiwanie na spełnienie warunku (zawieszenie działania wątku, aż do otrzymania sygnału od innego wątku). Zmienna warunkowa musi być zawsze otoczona mutexem, żeby uniknać równoczesnej próby oczekiwania i sygnalizowania na zmiennej warunkowej.

```
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);
int pthread_cond_destroy (pthread_cond_t *cond);
```

Opis: Funkcja inicjalizuje/zwalnia zmienną warunkową `cond` używając atrybutów wyspecyfikowanych w `cond_attr` lub używając domyślnych atrybutów.

```
int pthread_cond_signal (pthread_cond_t *cond);
int pthread_cond_broadcast (pthread_cond_t *cond);
```

Opis: Funkcja wznawia jeden/wszystkie z wątkow oczekujących na zmiennej warunkowej `cond`.

```
int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mutex,
                             const struct timespec *abstime);
```

Opis: Funkcja `pthread_cond_wait` w sposób atomowy zwalnia mutex i oczekuje na sygnal o spełnieniu zmiennej warunkowej cond. Wykonywanie wątku jest zawieszane i nie zajmuje czasu procesora aż do odebrania sygnału zmiennej warunkowej (w przypadku drugiej funkcji także po upłynięciu `abstime`). Mutex musi być zajęty przez wątek wołający `pthread_cond_wait`. Przed końcem działania funkcje zajmuje mutex.

```
pthread_condattr_init ( pthread_condattr_t *attr);
pthread_condattr_destroy ( pthread_condattr_t *attr);
```

Opis: Funkcje inicjalizują/usuwają obiekt typu `pthread_condattr_t`.

1.3.2 Podstawowe struktury danych

Struktura w której przekazywane są atrybuty tworzonego wątku

```
typedef struct {
int detachstate;     /* Czy może być przyłączany */
int schedpolicy      /* Rodzaj szeregowania */
struct sched_param schedparam      /*Priorytet gdy szeregowanie realtime*/
int inheritsched;       /*Czy sposób szeregowania dziedziczony od ojca */
int scope;         /* nieużywany (współzawodnictwo o procesor między
                      wątkami czy ze wszystkimi procesami) */
} pthread_attr_t;
```

Struktura w której przekazywany jest rodzaj tworzonego mutexa:

```
typedef struct {
int mutexkind;    /*może byc FAST(domyslny -standardowy), RECURSIVE(mutex może
       być ponownie zamknięty przez wątek który go posida) lub ERRORCHECKING)*/
} pthread\_mutexattr\_t;
```

Struktura w której ustawione zostaną parametry tworzonego wątku:

```
struct pthread_start_args {
void *(*start_routine)(void *);     /* funkcja wykonywana przez wątek */
void *arg;           /*argumenty tej funkcji*/
sigset_t mask;       /*maska sygnalowa dla wątku*/
```

```
int schedpolicy;      /* sposób szeregowania */
struct sched_param schedparam;/*priorytet szeregowania jeśli realtimeschedule*/
};
```

Identyfikator wątku używany w każdej procedurze:

```
typedef unsigned long pthread_t;
```

Wenętrzna struktura używana do określania wątkow:

```
struct _pthread_descr_struct {
pthread_descr p_nextlive, p_prevlive;     /* Dwukierunkowa lista aktywnych
                                             wątkow */
pthread_descr p_nextwaiting;  /* Nastepny element w kolejce do obsłużenia */
pthread_t p_tid;        /* Identyfikator wątku (ten widoczny na zewnątrz) */
int p_pid;        /* PID Unix-owego procesu */
int p_priority;     /*Priorytet wątku (== 0 jeśli nie czasu rzeczywistego) */
int *p_spinlock;      /* Blokada dla synchronicznego dostępu */
int p_signal;       /* Ostatnio otrzymany sygnał */
sigjmp_buf *p_signal_jmp;     /* co ma wykonać po otrzymaniu sygnału */
sigjmp_buf *p_cancel_jmp;/* co ma wykonać po przerwaniu go przez inny wątek */
char p_terminated;       /* true jeśli zakończony przez pthread_exit */
char p_detached;      /* true jeśli nie można go przyłączyć */
char p_exited;     /* true jeśli przerwany ze skutkiem natychmiastowym */
void *p_retval;     /* wartość zwracana przez wątek */
int p_retcode;      /* kod zwracany przez wątek (błąd lub nie) */
pthread_descr p_joining;     /* dekryptor wątku który go przyłącza lub NULL */
struct _pthread_cleanup_buffer *p_cleanup;     /* funkcja czyszcząca po
                                                  zakończeniu wątku */
char p_cancelstate;       /* czy może byc przerwany czy nie */
char p_canceltype;     /* typ reakcji na przerwanie wątku (deferred/async) */
char p_canceled;     /* czy dostał zlecenie przerwania */
int p_errno;       /* błąd zwrócony przez wywołanie systemowe */
int p_h_errno;      /* błąd zwrócony przez netdb funkcje */
struct pthread_start_args p_start_args;     /* struktura opisana wyżej
                   przechowująca argumenty przekazane przy tworzeniu wątku */
void **p_specific[PTHREAD_KEY_1STLEVEL_SIZE];/* specyficzne dane dla wątku */
};
```

Wskaźnik na poprzednią strukturę używany w większości funkcji operujących na wątkach:

```
typedef struct _pthread_descr_struct *pthread_descr;
```

Kolejka czekających wątkow:(wykorzystywana w mutex-ie i zmiennych condition)

```
struct _pthread_queue {
```

```
_pthread_descr head;     /* Pierwszy element lub null gdy pusta */
_pthread_descr tail;     /* Ostatni element lub null gdy pusta*/
};
```

Struktura mutexa:

```
typedef struct {
int m_spinlock;    /* blokada synchronicznego dostepu */
int m_count;      /* 0 jeśli wolny >0 jeśli zajety */
_pthread_descr m_owner;    /* właściciel mutexa (potrzebne dla mutexów typu
                              recursive ) */
int m_kind;    /* rodzaj mutexa */
struct _pthread_queue m_waiting;    /* Kolejka czekających wątkow */
} pthread_mutex_t;
```

Struktura zmiennych conditions:

```
typedef struct {
int c_spinlock;      /* blokada synchronicznego dostępu */
struct _pthread_queue c_waiting;       /* Kolejka czekających wątkow */
} pthread_cond_t;
```

Struktura opisująca funkcje czyszczące po zakończaniu wątku(cleanup buffers):

```
struct _pthread_cleanup_buffer {
void (*routine) __P ((void *));     /* Wywoływana funkcja */
void *arg;       /* argumenty funkcji */
int canceltype;        /* Zapamiętany typ przerywania wątku */
struct _pthread_cleanup_buffer *prev;      /* lista funkcji czyszczących */
};
```

Globalna tablica handlerów

Służąca do znajdowania deskryptora wątku `pthread_descr` na podstawie identyfikatora wątku (identyfikator `pthread_t` wątku określa miejsce w tej globalnej tablicy). Tablica ta łączy wszystkie struktury wymienione wyżej i mam nadzieję że wyjaśnia w jaki sposób na podstawie identyfikatora wątku wszystko o nim wiadomo.

```
typedef struct pthread_handle_struct __pthread_handles[PTHREAD_THREADS_MAX];

struct pthread_handle_struct {
int h_spinlock;           /* blokada synchr. dostępu */
pthread_descr h_descr;          /* Deskryptor wątku */
};
```

1.3.3 Ogólne informacje wprowadzające do budowy algorytmów

Wątki w bibliotece LinuxThreads dzialają na zasadzie klient-serwer. Przy tworzeniu pierwszego wątku automatycznie powstaje wątek zarządca który jest swoistego rodzaju serwerem dla pozostałych wątkow (to mu zlecane sa operacje tworzenia, usuwania, zakończenia wątkow i on fizycznie te operacje wykonuje i po wykonaniu budzi wątek, od którego otrzymał zlecenie). Jeśli proces stworzy N wątkow to tak naprawdę powstanie N+2 wątkow (zarządca, wątek główny od którego poprzez clone() powstają inne wątki i N zwykłych wątków).

Ograniczenia jądra Linuksa

Dla wątków korzystających ze wsparcia jadra (funkcja clone()):
- stala NR_TASKS w jadrze (sched.h) – standardowo 512
- ograniczenia architektury x86: NR_TASKS=<4090

Ograniczenia LinuxThreads

W bibliotece LinuxThreads rozmiar tablicy wątkow ograniczony jest do 1024. Zasoby rezerwowane przez wątek to miedzy innymi 2MB pamieci wirtualnej na stos, a na intelu tylko pierwsze 1GB jest na to przeznaczone. Wielkość rezerwowanego obszaru można zmienić używając atrybutu `setstackaddr`.

Funkcja clone()

Jak wiadomo wątki to praktycznie takie same procesy jak inne tylko współdzielą pewne zasoby. Dlatego tworzy się je za pomocą podobnych mechanizmów jak procesy jedynie ustawiając parametry tworzenia na dzielenie zasobów (np pamięci, tablicy obsługi sygnałów, tablicy deskryptorów), wykorzystując przy tym funkcję `do_fork`. Domyślnie funkcja `clone()` nie jest dostępna. Aby można było z niej korzystać należy skompilować jądro ze zdefiniowanym symbolem:
`CLONE_ACTUALLY_WORKS_OK`.

```
pid_t clone(void *sp, unsigned long flags)
```

- Jeśli `sp` jest różne od NULL, to wskaźnik ten jest używany jako początkowy wskaźnik stosu dziecka.
- Pole flags określa, co chcemy klonwać.

W bibliotece LinuxThreads wykorzystywana jest funkcja `__clone`. Uruchamiana jest ona przez funkcję `pthread_handle_create`. Funkcja `pthread_handle_create` tworzy nowy obiekt `pthread_descr` (jest to wskaźnik na wypisaną wyżej ogromną strukturę wątku) i przydziela nowy identyfikator wątku (dokładniejszy opis niżej).

```
__clone(int (*fn) (void *arg), void *child\_stack,int flags,void *arg);
```

- pierwszy argument jest wskaźnikiem do funkcji wykonywanej przez wątek,
- drugi wskaźnikiem do stosu w przypadku dzielenia pamięci,

- trzeci to flagi wspierające tworzenie wątkow:
  - CLONE_VM - współdzielenie danych i stosu
  - CLONE_FS - współdzielenie tablicy plików
  - CLONE_FILES - współdzielenie informacji o otwartych plikach
  - CLONE_SIGHAND - współdzielenie tablicy sygnalów
  - CLONE_PID - współdzielenie pidu
- czwarty argumenty

W praktyce jako `*child_stack` i `*arg podawany` jest deskryptor wątku czyli wskaźnik na strukturę `_pthread_descr_struct`.

Funkcja `sys_clone ()` - działanie na poziomie jądra

Wywołanie tej funkcji systemowej nastepuje w funkcji `__clone()`. Funkcja `sys_clone()` pobiera z rejestrów flagi (ebx) oraz nowy wskaźnik stosu (ecx). W przypadku gdy nowy wskaźnik stosu nie zostanie podany, używany jest stos procesu-ojca, co jest niewskazane przy dzieleniu pamięci (poprawne działanie takich procesów jest prawie niemożliwe). Następnie wywołuje `do_fork()`, który:

- kopiuje strukturę task aktualnego procesu
- wywołuje funkcje kopiujące pozostałe struktury procesu: (pamięć, parametry lekkiego procesu, pliki, sygnały), które również aktualizują strukturę task.

1.3.4 Ważniejsze algorytmy

Algorytmy dzialania zwyklych wątkow

Podczas pierwszego uruchomienia funkcji tworzacej wątek (`pthread_create()`), czyli gdy zarządca jeszcze nie istnieje, uruchamiany jest algorytm tworzenia wątku zarządcy:
`pthread_initialize_manager()`

Algorytm 1 pthread_create() - stworzenie wątku

Dane wejściowe: wskaźnik na id, atrybuty, wykonywaną funkcję i jej argumenty
Wynik: wartość 0 (gdy sie udało), id wątku we wskaźniku na id
1: **jeśli** wątek zarządca nie istnieje **to**
2: wywołaj funkcję `pthread_initialize_manager()`, która go zainicjuje.
3: **koniec jeśli**
4: Ustaw parametry nowego wątku w strukrurze `request` (atrybuty, maskę sygnałową, polecenie stworzenia wątku)
5: Wyślij zarządcy polecenie stworzenia wątku z parametrami przekazanymi w `request` i zawieś się czekając na sygnał od zarządcy, który inicjuje tworzenie wątku.

Algorytm 2 pthread_exit() - zakończenie wątku

Dane wejściowe: void
Wynik: wartość zwracaną przez wątek

Wywołaj funkcje czyszczące niepotrzebne pola wątku: `__pthread_perform_cleanup()` `__pthread_destroy_specifics()`;
Zapamiętaj zwracaną wartość w strukturze wątku i ustaw w polu wątku że się zakończył:
jeśli ktoś chce dołączyć wątek **to**
  wyślij mu sygnał `PTHREAD_SIG_RESTART` żeby go obudzić.
koniec jeśli
jeśli kończacy sie wątek jest wątkiem glownym **to**
  wyślij zarządcy zlecenie wyjścia głównego wątku i zawieś się w oczekiwaniu na sygnał od zarządcy.
koniec jeśli
exit(0);

Algorytm 3 dołączania innego wątku (pthread_join())

Dane wejściowe: id przyłączanego wątku, wskaźnik na zwracaną przez niego wartość
Wynik: wartość zwróconą przez przyłączany wątek

jeśli wątek który chcemy dołączyć nie istnieje **to**
  Zwróć błąd ESRCH
koniec jeśli
jeśli probujesz dolaczyc siebie **to**
  Zwróć błąd EDEADLK
koniec jeśli
jeśli wątek nie da się przyłączyc (jest detached) lub jest już przyłączony **to**
  zwróć błąd EINVAL
koniec jeśli
jeśli przykłączany wątek nie przestał jeszcze działać **to**
  Zapisz w dołączanym wątku, że chcesz go przyłączyć i zawieś się w oczekiwaniu na jego zakończenie (wyśle on wtedy sygnał budzący do wszystkich którzy na niego czekają)
koniec jeśli
Ustaw zwracaną wartość
Wyślij do wątku zarządcy informacje o przyłączanym wątku, aby mógł go usunąć.

Algorytm 4 pthread_handle_create (tworzy strukturę opisującą wątek używana w algorytmie zarządcy)

Dane wejściowe: atrybuty wątku,maska sygnałowa, pid ojca, funkcja do wykonywania, wskaźnik do id_wątku
Wynik: identyfikator nowego wątku w pobranym wskaźniku

Znajdź miejsce na stos
Zaalokuj pamięć na stos
Ustaw atrybuty nowego wątku (inicjalizacja struktury `pthread_descr`)
Ustawia parametry dla funkcji `pthread_start_thread`
Wywołaj `__clone` z funkcją wykonywaną `pthread_start_thread`
Wstaw wątek do dwukierunkowej listy aktywnych wątkow (tej w strukturze `pthread_descr` (pola `p_prevlive`, `p_nextlive`))

Algorytm 5 pthread_start_thread - uruchamia wątek (używana w algorytmie zarządcy)

Dane wejściowe: deskryptor wątku (wskaźnik na strukturę `_pthread_descr_struct`)
Wynik: kod błędu

Zainicjuj pid dla pewności
Zainicjuj maskę sygnałową wątku
Uruchom kod wątku (funkcję przez niego wykonywaną)
Wywołaj funkcję `pthread_exit()` z wartością zwrócona przaz wykonywaną funkcję;

Algorytm 6 pthread_initialize_manager() - inicjuje zarządcę

Dane wejściowe: void
Wynik: kod blędu

jeśli nie zainicjalizowano wątku głównego (ang. pthread_initial_thread_bos) **to**
    Zainicjuj go (wywołaj funkcję `pthread_initialize()`)
koniec jeśli
Przydziel pamięć na stos
Utworz `pipe-a` do komunikacji z innymi wątkami
Wywołaj `__clone()` (tutaj zacznie działać wątek zarządca- funkcja `__pthread_manager`)
Sprawdź wartość zwróconą przez `__clone()` i w przypadku wystąpienia błędu zwolnij pamięć na stos, zamknij łącze i zwróć -1

Algorytm 7 __pthread_manager() - praca wątku zarzadcy

Dane wejściowe: deskryptor **pipe-a** do czytania zleceń od wątków

Wynik: kod błędu

```
Zablokuj wszystkie sygnały oprócz PTHREAD_SIG_RESTART
dopóki true wykonuj
  Ustaw czas oczekiwania na zgłoszenie na 2 sekundy
  jeśli został zakończony główny wątek to
    Wyślij sygnał końca do wszystkich wątków pthread_kill_all_threads(SIGKILL,0)
  koniec jeśli
  jeśli zakończył sie jakis zwykły wątek to
    Zwolnij po nim/nich pamięć funkcją pthread_reap_children()
  koniec jeśli
  jeśli pojawiło sie zadanie od innego wątku to
    jeśli zadanie==REQ_CREATE to
      Wywołaj funkcję pthread_handle_create, która utworzy wątek:
      Obudź nowo utworzony wątek
    koniec jeśli
    jeśli zadanie==REQ_FREE to
      Wywołaj funkcję pthread_handle_free(), zwalniająca zasoby wątku
    koniec jeśli
    jeśli zadanie==REQ_PROCESS_EXIT to
      Wywołaj pthread_handle_exit , która wysyła sygnał CANCEL do wszystkich działających wątków łącznie z głównym wątkiem ale nie do wątka, z którego przyszło polecenie (ten wątek musi zakończyć wyjście i zwolnić pamięć )
    koniec jeśli
    jeśli zadanie==REQ_MAIN_THREAD_EXIT to
      Jeśli pozostałe wątki się zakończą obudź główny wątek
    koniec jeśli
  koniec jeśli
koniec dopóki
```

Algorytm 8 zakonczenie wątku zarzadcy

Dane wejściowe:

Wynik:

```
wątek główny zgłasza, że chce się zakończyć- zlecenie REQ_MAIN_THREAD_EXIT
Zarzadca czeka na zakończenie innych wątkow i budzi wątek główny
```

1.4 Zadania

1. Pytanie: Jakich wątków najlepiej użyć aby zapewnić optymalną realizację nastepujących zadań na maszynie jednoprocesorowej:
   - (a) algorytmu producentów i konsumentów z dużym buforem dyskowym
   - (b) algorytmu pointer jumping na długiej liście
2. Pytanie: Jak myślisz, dlaczego akurat wątki poziomu jądra stały się standardem?
3. Pytanie: Powiedz dlaczego w bibliotece LinuxThreads proces może stworzyć tylko określona przez stala liczbę watków?
4. Pytanie: Wyjaśnij w jaki sposób informacje o wątku pozyskiwane są na podstawie jego identyfikatora.
5. Pytanie: W którym miejscu następuje uruchomienie nowo stworzonego wątku?

1.5 Rozwiązania zadań

1. - (a) Wątków poziomu jądra, z uwagi na używanie blokujących funkcji systemowych (nawet przy założeniu, że zostały nadpisane, efektywniejsze powinny być funkcje standardowe).
   - (b) Wątków poziomu użytkownika z uwagi na częste zmiany kontekstu których narzut czasowy jest mniejszy niż w przypadku wątków poziomu jądra.
2. Od czasu, kiedy jądro linuksa ma wbudowany mechanizm obsługi wątków, wątki użytkownika znajdują się w odwrocie. Nie dość, że wątki poziomu jądra unikają wszystkich wad implementacji użytkownika, to jeszcze stosowane są pewne mechanizmy które umożliwiają skrócenie czasu przełączania kontekstu między wątkami. Przewaga wątków poziomu użytkownika na tym polu staje się coraz mniejsza.
3. Odpowiedzią jest globalna tablica przechowująca informacjje o wątkach.
4. Na podstawie identyfikatora znajdowany jest w tablicy handlerów obiekt typu pthread_handle_struct przechowujący deskryptor wątku, czyli wskaznik na strukturę opisującą wątek
   (_pthread_descr_struct).
5. Funkcja wykonywana przez wątek zostaje uruchomiona przez pthread_start_thread, która jest funkcją wykonywalną dla funkcji clone (funkcja clone jest uruchamiana przez pthread_handle_create, a ta z kolei przez managera) Własciwe uruchomienie wątku następuje dopiero po obudzeniu go przez managera.