Biblioteka wątków użytkownika `libpwuthread`.

Paweł Wieczorek

16 września 2014

Spis treści

1. Wstęp.

Dokument stanowi opis realizacji biblioteki wątków użytkownika. Celem biblioteki było pokonanie wyzwania programistycznego polegającego na zrealizowaniu wątków użytkownika z wywłaszczeniami bez użycia wstawek jęza maszynowego. Na zaimplementowany model miał wpływ ustandaryzowany interfejs biblioteki wątków - PTHREAD [1].

1.1. Budowa.

Przed zbudowaniem biblioteki należy użyć skryptu `configure.sh` w celu zbudowania pliku budującego `Makefile`. Skrypt zadaje użytkownikowi kilka pytań, a następnie tworzy plik podstawiając odpowiednie wartości w danych z pliku `Makefile.in`. Poprawność odpowiedzi użytkownika nie jest weryfikowana.
Przykładowe użycie skryptu na systemie Solaris:

```
$ ./configure.sh
Select compiler (GCC, SUNCC): SUNCC
Select context mechanism (setjmp, ucontext): ucontext
Select stack mechanism (ansi, unix): unix
```

Znaczenia pytań i odpowiedzi są następujące:

- `compiler` - zestaw budujący
  - `GCC` - kompilator `gcc`
  - `SUNCC` - kompilator *Sun Studio*
- `context mechanism` - mechanizm obsługujący kontekst procesora
  - `setjmp` - plik `support/unix_ctx_setjmp.c` - mechanizm opierający się o wywołanie `sigaltstack` oraz procedury `sigsetjmp`, `siglongjmp`
  - `ucontext` - plik `support/unix_ctx_ucontext.c` - mechanizm operający się o interfejs `ucontext` ze standardu POSIX.
- `stack mechanism` - mechanizm obsługi stosu wątków
  - `ansi` - używający pary procedur `malloc`, `free`
  - `unix` - używający UNIX-owych wywołań `mmap`, `munmap`

Jeżeli wartości zostały wybrane prawidłowo to biblioteka oraz przykłady zostaną zbudowane po wydaniu polecenia `make`. Wynikiem końcowym budowania jest biblioteka dzielona `libpwuthread.so.1` oraz przykłady w katalogu `examples`.

1.2. Użycie.

Program używający bibliotekę musi załączyć plik nagłówkowy `pwuthread.h` oraz dołączyć bibliotekę `libpwuthread.so.1`. Przykład kompilacji na systemie Linux, gdzie `ipath` to katalog w którym znajduje się nagłówek, a `lpath` to katalog w którym znajduje się biblioteka.

```
$ gcc -Iipath -o program program.c -Llpath -lpwuthread
```

Jeżeli użytkownik zdefiniuje zmienną środowiskową `PWUTHREAD_LOG` to na terminal będą wypisywane wewnętrzne komunikaty biblioteki.

1.3. Przykłady.

Wraz z biblioteką są dostarczone przykłady wykorzystania:

1. `ex01` - przykład stworzenia wątku i dołączenia się do niego
2. `ex03` - przykład używania kluczy oraz usypiania wątku
3. `ex04` - przykład użycia synchronizacji
4. `ex05` - przykład użycia wartości warunkowej, implementacja blokującej kolejki

Przykładowe programy do uruchomienia wymagają dostępu do dzielonej biblioteki. System operacyjny zwykle szuka bibliotek dzielonych w ustalonych katalogach np.: `/usr/lib`, `/usr/local/lib` itp. Obecne zestawy narzędzi budujących dołączają do programu informacje, aby system przeszukiwał także obecny katalog w poszukiwaniu bibliotek. Stąd istotne jest aby uruchamiać przykład będąc w katalogu z biblioteką. Czy biblioteka może być znaleziona można sprawdzić programem `ldd`, przykład testu na systemie FreeBSD:

```
$ ldd ./ex01
./ex01:
        libpwuthread.so.1 => not found (0x0)
        libc.so.7 => /lib/libc.so.7 (0x800647000)
$ cd ..
$ ldd ./examples/ex01
./examples/ex01:
        libpwuthread.so.1 => ./libpwuthread.so.1 (0x800647000)
        libc.so.7 => /lib/libc.so.7 (0x800752000)
```

Na teście widać, że przykładowy program nie może być uruchomiony z katalogu `examples` bo nie ma w nim biblioteki.

1.4. Przetestowane platformy i przenośność.

Lista przetestowanych platform, platformy mające przedrostek *qemu* oznaczają środowisko emulatora *qemu*, uruchomionego na komputerze *żubr*.

Nazwa	**System**	**Platforma**	**Wersja**	`setjmp`	`ucontext`
cvs	Linux	x86_64	2.6.26-2	OK	OK
hera	Linux	x86	2.6.26-2	OK	OK
macbeth	Darwin	x86	10.2.0	OK	OK
qemu1	NetBSD	qemu-x86	5.0.1	OK	OK
qemu2	NetBSD	qemu-sparc	4.0	NIE	OK
tryglaw	Solaris	x86	5.10	OK	OK
trzask	FreeBSD	x86	8.0	OK	OK
żubr	FreeBSD	x86_64	8.0	OK	OK

Niewykluczone, że inne systemy oraz platformy będą działać. Należy uważać na działanie mechanizmu `setjmp` ponieważ początek stosu wątku może być modyfikowany przez mechanizm dostarczający sygnały. Przykładowo, autorowi sprawiło wiele trudności odkrycie co modyfikuje zmienne lokalne procedury krótkiego skoku (patrz opis mechanizmu `setjmp`) na systemie Solaris. Natomiast dla emulowanej platformy sparc nie udało się znaleźć przyczyny nieprawidłowego działania tego mechanizmu na systemie NetBSD-4.0.

1.5. Maszyny z wieloma procesorami.

Ponieważ wątki są implementowane po stronie użytkownika to system operacyjny nie wie o ich istnieniu. Dlatego program mimo wątków może używać tylko jednego procesora. W celu użycia wielu procesorów należy stworzyć wątki widziane po stronie jądra. Jednak ich tworzenie wymaga użycia specyficznych dla danego systemu wywołań, przez co biblioteka nie mogłaby być przenośna - nawet między systemami UNIX-owymi. Lepiej się posłużyć w takim wypadku systemową biblioteką wątków implementującą standard PTHREAD[1] gdyż interfejs jest przenośny, a biblioteka dostarczona przez system użyje jego wewnętrznych mechanizmów.

1.6. Co można jeszcze zrobić

Przy tworzeniu biblioteki brano pod uwagę inny model wątkowania - kooperatywny, gdzie każdy wątek jawnie oddaje sterowanie innemu wątkowi. Taki model nie opierający się o wywłaszczenia mógłby posłużyć do implementacji np leniwych generatorów znanych w językach C# i Python (patrz słowo kluczowe `yield`). Ograniczenia czasowe są powodem nie zaimplementowania tego mechanizmu, lecz wszystkie niezbędne elementy do jego tworzenia są gotowe [1].

Dodatkową modyfikacją jaką można wykonać byłaby modułowość programu planisty, wtedy biblioteka mogłaby służyć do testowania różnych algorytmów zarządzania zadaniami - programista by pisał wtyczkę, która by kontrolowała działające wątki.

Do samego zarządzania wątkami opartego o wywłaszczenia należy wykonać jeszcze procedury anulowania wątku (patrz `pthread_cancel` w standardzie).

Bibliotekę można uczynić bardziej funkcjonalną dodajać dodatkowe procedury realizujące wejście-wyjście. Ponieważ obecnie mogą one zostać przerwane przed dostarczenie sygnału. Realizacja takich nakładek wydaje się być prosta - powinna korzystać z nieblokujących deskryptorów oraz dodać dodatkowe zadanie administracyjne planisty sprawdzające operacje wej-wyj.

UWAGA! Zgodnie z najnowszym standardem POSIX:2008 niektóre funkcje otrzymują status *przestarzałe* [5]. Co oznacza, że wraz z przyszłym rozwojem systemów i standardu mogą one zniknąć, a w tym używane przez bibliotekę `setitimer` oraz `gettimeofday`. Obsługa czasu jest przygotowana na modułowość, zatem obok obecnego pliku `unix_time_gettimeofday.c`, można dodać `unix_time_clock.c` używającą nowszego mechanizmu `clock_gettime`. Do obsługi czasomierza należałoby jednak wprowadzić interfejs, który mógłby być implementowany przez obecnie używane `setitimer` oraz dodatkowy moduł, używający nowszego interfejsu `timer_create`. Dzięki modułowości biblioteka mogłaby działać na starych oraz przyszłych systemach.

2. Zarządzanie wątkami.

Niniejszy rozdział opisuje interfejs biblioteki, każda omawiana stała i funkcja ma przedrostek `pwuthread_` lub `PWUTHREAD_`. Ich odpowiedniki w standardzie PTHREAD[1], jeżeli istnieją, to mają przedrostki `pthread_` lub `PTHREAD_`. W przytaczanych fragmentach kodów przedrostki nie są obcinane.

[1] Ten model jest prostszy niż zrealizowany opierający się o wywłaszczenia.

2.1. Tworzenie wątków.

```
int pwuthread_create(pwuthread_t *tdp, pwuthread_attr_t *attr, func, void *arg);
```

Każdy wątek jest związany z obiektem `attr_t` opisującym jego atrybuty. Obecnie w implementacji jedyne ustawialne atrybuty to segment stosu, stąd też nie będą one omówione.

Za tworzenie wątku odpowiedzialna jest procedura `create`, która przyjmuje cztery argumenty. Pierwszym jest referencja do deskryptora wątku, który zostanie przypisany do tworzonego wątku. Drugim argumentem jest obiekt atrybutów, wartość tego argumentu może wynosić `NULL` jeżeli chcemy użyć atrybutów domyślnych. Jak wspomniano biblioteka nie udostępnia modyfikacji żadnych praktycznych atrybutów, stąd też korzystanie z domyślnych atrybutów jes zalecane. Ostatnie dwa argumenty to adres i argument podprogramu wątku.

2.2. Prywatne dane wątku.

```
int pwuthread_key_create(pwuthread_key_t *key, pwuthread_key_dtor_t *f);
int pwuthread_key_destroy(pwuthread_key_t key);
void *pwuthread_getspecific(pwuthread_key_t key);
void pwuthread_setspecific(pwuthread_key_t key, const void *v);
```

Standard opisuje możliwość tworzenia kluczy, za pomocą których każdy wątek może zapisywać i pobierać prywatne wartości.

Klucze są opisywane przez typ `key_t`, a wartości muszą być wskaźnikami.

Druga opcja funkcji `key_create` służy jedynie dla podobieństwa do standardu PTHREAD, nie jest używana wewnątrz biblioteki. Procedurami `getspecific` i `setspecific` można odpowiednio pobierać i zapisywać wartość kojarzoną z kluczem przez właśnie działający wątek. Przykład wykorzystania znajduje się w `examples/ex03.c` oraz w następnym podrozdziale.

UWAGA! Należy zwrócić uwagę, że nie istnieje możliwość aby wątek pobrał wartość kojarzoną z danym kluczem przez inny wątek, zatem należy uważać na wycieki pamięci! Przykładowo załóżmy, że z danym kluczem kojarzymy wskaźnik na dynamicznie przydzieloną tablicę i jest to jedyne miejsce przechowywania tego wskaźnika. Jeżeli wątek zakończy działanie to aplikacja gubi bezpowrotnie wskaźnik i nie będzie mogła już nigdy zwolnić tej pamięci. Należy pilnować aby wątek przed swoją śmiercią zwolnił prywatne zasoby.

2.3. Synchronizacja.

```
int pwuthread_mutex_init(pwuthread_mutex_t *mtx, const void *);
int pwuthread_mutex_lock(pwuthread_mutex_t *mtx);
int pwuthread_mutex_unlock(pwuthread_mutex_t *mtx);
int pwuthread_mutex_trylock(pwuthread_mutex_t *mtx);
int pwuthread_mutex_destroy(pwuthread_mutex_t *mtx);

int pwuthread_cond_init(pwuthread_cond_t *cnd, const void *);
int pwuthread_cond_destroy(pwuthread_cond_t *cnd);
int pwuthread_cond_wait(pwuthread_cond_t *cnd, pwuthread_mutex_t *mtx);
int pwuthread_cond_signal(pwuthread_cond_t *cnd);
int pwuthread_cond_broadcast(pwuthread_cond_t *cnd);
```

Biblioteka dostarcza dwóch mechanizmów synchronizacji - zamki typu `mutex`[2] oraz zmienne warunkowe.

[2]MUTual EXclude - wzajemne wykluczenie

Drugi parametr procedur `mutex_init` oraz `cond_init` służy dla zachowania podobieństwa do standardu PTHREAD.

Zamki można wyobrazić sobie jako semafory binarne, które dodatkowo cechuje właściciel. Wątek zamykający zamek staje się jego właścicielem i tylko on może go otworzyć. Próba zamknięcia zamkniętego zamka uśpi dany wątek, który zostanie obudzony przez bibliotekę gdy będzie mógł stać się właścicielem zamka.

Zmienna warunkowa to pomocniczy mechanizm służący do implementacji zdarzeń. Jeżeli wątek jest właścicielem pewnego zamka, to może za pomocą procedury `cond_wait` odblokować go i zasnąć w oczekiwaniu na spełnienie warunku symbolizowany przez daną zmienną warunkową. Jeden oczekujący na zdarzenie wątek można kazać obudzić za pomocą procedury `cond_signal`. Biblioteka obudzi wątek dopiero wtedy, gdy ten będzie mógł stać się ponownie właścicielem zamka, który zwolnił idąc spać. Za pomocą procedury `cond_broadcast` można kazać obudzić wszystkie oczekujące na zdarzenie wątki.

Przykładowe wykorzystanie kluczy i synchronizacji. Poniższe procedury pozwalają nazywać wątek, każdy wątek może pobrać swoją nazwę za pomocą procedury `get_my_name`. Przykłady wykorzystania synchronizacji znajdują się także w plikach `ex04.c` oraz `ex05.c`.

```
pwuthread_key_t name_key;
pwuthread_mutex_t name_mtx;

const char *get_my_name() {
    static int i = 0;
    char *b = (char*) pwuthread_getspecific(name_key);
    if (b == NULL) {
        pwuthread_mutex_lock(&mtx);
        b = (char*)malloc(200);
        snprintf(b, 200, "thr%04u", i);
        pwuthread_setspecific(name_key, b);
        i++;
        pwuthread_mutex_unlock(&mtx);
    }
    return b;
}

void free_my_name() {
    const char *n = get_my_name();
    free( (void*)n );
}
```

2.4. Usypianie wątku na określony czas.

```
void pwuthread_util_sleep(int s);
void pwuthread_util_usleep(int us);
```

Wątek nie może prosić o uśpienie system operacyjny, ponieważ w zależności od działania systemu - może to albo zablokować cały proces (wszystkie wątki) albo uśpić proces jedynie do następnego wywołania programu planisty[3]. W związku z tym za usypianie poszczególnych

[3] Na systemach UNIX-owych jest domyślnie realizowany drugi scenariusz - proces jest budzony przy dostarczeniu sygnału, a wywołanie systemowe blokujące proces zwraca błąd `EINTR`.

wątek musi być odpowiedzialny program planisty - wymione procedury emulują procedury `sleep` oraz `usleep` z języka C.

3. Implementacja.

Jednym z założeń biblioteki była jej przenośność, więc bibliotekę podzielono na dwie części: wsparcie dla konkretnych mechanizmów systemu operacyjnego oraz abstrakcyjną niezależną od niego.
UWAGA! Ponieważ biblioteka obecnie nie obsługuje innych systemów niż te wywodzące się lub wzorujące się na systemie UNIX to możliwe, że mimowolnie do części abstrakcyjnej przemycono założenia narzucone przez te systemy.

- `include/pwuthread.h` - nagłówek dla użytkowników biblioteki
- `include/pwuthread/` - katalog z wewnętrznymi nagłówkami
- `lib/` - abstrakcyjny kod biblioteki
- `lib/main.c` - administracyjne procedury
- `lib/sched.c` - planista
- `lib/specific.c` - implementacja mechanizmu kluczy
- `lib/sync.c` - implementacja mechanizmów synchronizacji
- `lib/thread.c` - implementacja procedur zarządzających wątkami
- `lib/thread_attr.c` - implementacja obiektu atrybutów wątku
- `lib/util.c` - pomocnicze procedury - nakładki na wywołania systemowe.
- `support/` - wsparcie dla niskopoziomowych mechanizmów
- `support/unix_ctx_setjmp.c` - mechanizm obsługi kontekstu procesora `setjmp`
- `support/unix_ctx_ucontext.c` - mechanizm obsługi kontekstu procesora `ucontext`
- `support/unix_sched.c` - implementacja podstaw planisty
- `support/unix_stack.c` - procedury zarządzające stosem za pomocą wywołania `mmap`
- `support/ansi_stack.c` - procedury zarządzające stosem za pomocą procedury `malloc`
- `support/unix_time_gettimeofday.c` - procedury zarządzające czasem, używające wywołania `gettimeofday`

3.1. Zarządzanie kontekstem procesora.

Moduł implementujący zarządzanie kontekstem procesora musi zaimplementować poniższe procedury oraz zdefiniować strukturę `pwuthread_ctx`

```
struct pwuthread_ctx;

int pwuthread_ctx_create(pwuthread_t t, pwuthread_main_f *f, void *sa, size_t sl);
int pwuthread_ctx_switch(pwuthread_ctx_t ctx0, pwuthread_ctx_t ctx1);
int pwuthread_ctx_detach(pwuthread_ctx_t ctx);
```

Pierwsza służy do stworzenia nowego kontekstu dla wątku `t`. Procedurą startową wątku ma być `f`, a stosem o rozmiarze `sl` ma być `sa`. Druga procedura przełącza się z obecnego kontekstu `ctx0` do kontekstu `ctx1`. Ostatnia służy do zwalniania zasobów danego kontekstu.

3.1.1. Tworzenie kontekstu procesora za pomocą *setjmp*.

Mechanizm `setjmp` jest dość trudny w realizacji, jego implementacja jest w pewnym sensie *sprytna*. Idea stojąca za tym mechanizmem to użycie sygnałów do zmiany stosu oraz procedur z języka C `setjmp`, `longjmp` do manipulacji kontekstem procesora.

Procedura `setjmp` zachowuje obecny kontekst procesora w tak zwanym buforze skoku (*jump buffer*), w takim wypadku zwracaną wartością procedury jest zero. Druga procedura `longjmp` służy do wczytania wcześniej zachowanego kontekstu procesora. Po wczytaniu kontekstu procesor zacznie wykonywać kod programu od momentu powrotu procedury zachowującej kontekst. Aby umożliwić programiście rozpoznanie czy powrót z procedury zachowującej kontekst to rzeczywisty jej powrót czy efekt wczytania kontekstu umożliwia się podanie zwracanego wyniku przez `setjmp`.

Przykład, w poniższym kodzie procedura `g` zapisuje kontekst procesora w buforze i wynik `setjmp` zapisuje w zmiennej `r`. Jeżeli wynikiem było zero to wiemy, że właśnie zachowaliśmy kontekst procesora i uruchamiamy procedurę `f`. Ta procedura natomiast wykonuje pętle nieskończoną, wewnątrz której wczytuje znaczki z klawiatury. Gdy wczytanym znaczkiem będzie „*q*" to wczytywany jest kontekst procesora z buforu. W takim wypadku procesor *skoczy* do momentu powrotu z procedury `setjmp`, a jako zwracaną wartość będzie podane 42.

```
jmp_buf bufor;

void f() {
    while (1) {
        c = fgetc(stdin);
        if (c == 'q') {
            longjmp(bufor, 42);
        }
    }
}

void g() {
    int r = setjmp(bufor);
    printf("r = %i\n", r);
    if (r == 0) f();
}
```

Mechanizm niestety nie pozwala zmienić stos w kontekście procesora na podany, co uniemożliwia tworzenie nowych wątków, które muszą posiadać własny stos. W tym celu wykorzystywane jest wywołanie systemowe `sigaltstack` pozwalające wskazać alternatywy stos na czas obsługi sygnału. Biblioteka zmienia stos wysyłając do procesu sygnał `SIGUSR2`, którego obsługa jest rejestrowana w następujący sposób (flaga `SA_ONSTACK` informuje system, że ma być użyty stos wskazany przez `sigaltstack`):

```
static struct sigaction shortjump_sa;

    sigemptyset(&shortjump_sa.sa_mask);
    shortjump_sa.sa_handler = shortjump_handler;
    shortjump_sa.sa_flags = SA_ONSTACK;
    sigaction(SIGUSR2, &shortjump_sa, &oldsa)
```

Obsługa sygnału przez system operacyjny nie kończy się jedynie na uruchomieniu procedury obsługi, stąd należy taką procedurę zawsze zakończyć aby mechanizm dostarczania sygnału mógł „posprzątać". W związku z tym obsługa wyznaczonego sygnału `SIGUSR2` zachowuje kontekst procesora w buforze skoku i natychmiast kończy działanie, w efekcie w buforze zostaje zachowany kontekst z wskazanym przez bibliotekę stosem nowego wątku. Przy wczytaniu takiego kontekstu kod będzie mógł rozpoznać skok i uruchomić obsługę kodu wątku. Ponieważ obsługa sygnału jest bardzo krótka to procedura została nazwana *krótkim skokiem*:

```
void shortjump_handler(int s) {
    pwuthread_t thread = shortjump_thread;
    pwuthread_log("shortjump done for %p", thread);
    if (sigsetjmp(thread->context->jb, 1)) {
        thread = pwuthread_current;
        pwuthread_work(pwuthread_current);
        pwuthread_log("thread %p should never be here", pwuthread_current);
        abort();
        while(1);
    }
    shortjump_thread = 0;
    shortjump_raised = 1;
}
```

Ponieważ standard nie definiuje czy procedury `setjmp`, `longjmp` mają zachowywać maskę sygnałów procesów to można spodziewać się różnic w zachowaniu. W tym celu zostały użyte odpowiedniki procedur `sigsetjmp` oraz `siglongjmp` których zachowywanie maski sygnałów można kontrolować.

Opis użytych mechanizmów znajduje się w [2].

3.1.2. Tworzenie kontekstu procesora za pomocą *ucontext*.

W standardzie POSIX zdefiniowane są gotowe procedury służące do zarządzania kontekstem procesora [3].

```
#include <ucontext.h>

int getcontext(ucontext_t *);
```

```
int setcontext(const ucontext_t *);
void makecontext(ucontext_t *, void (*)(void), int, ...);
int swapcontext(ucontext_t *, const ucontext_t *);
```

Pierwsza procedura służy do zapisywania obecnego kontekstu procesora, druga do wczytania wcześniej zachowanego.

Tworzenie nowego kontekstu wymaga wcześniej zachowanego kontekstu oraz przydzielonego stosu. Ostatnia wspomniana procedura służy do przełączenia się miedzy kontekstami.

Mechanizm `ucontext` w bibliotece jest jedynie opakowaniem na procedury zdefiniowane przez standard.

3.2. Planista.

Kluczowym elementem biblioteki jest program planisty, który zarządza wątkami. Planista wykorzystuje semantykę sygnałów na systemie UNIX, gdzie dostarczenie sygnału wywłaszcza działający proces i uruchamia w nim obsługę sygnału. Podczas inicjalizacji biblioteki jest instalowany czasomierz, który co bardzo mały kwant czasu wysyła do procesu sygnał `SIGALRM` [4], jego obsługa uruchamia podprogram planisty, który zmienia obecnie działający wątek.

Program planisty przypisuje wątkom ustalone stany, które głównie znaczą w jakiej kolejce znajduje się wątek. Stany jak i dodatkowe informacje są kodowane w bitach pola `sched_flags` w strukturze wątku.

- `Q_run` - kolejka wątków działających (bit `RUN`)
- `Q_wait` - kolejka wątków oczekujących na wybudzenie (bit `WAIT`)
- `Q_sleep` - kolejka wątków uśpionych na czas (bit `SLEEP`)
- `Q_detached` - kolejka martwych wątków (bit `DETACHED`)
- `Q_zombie` - kolejka żywych trupów, które zakończyły stan działania, lecz potrzebne są ich deskryptorów (bit `ZOMBIE`)

Nowo utworzone wątki trafiają do kolejki `Q_run`. Po zakończeniu działania wątek może zostać przeniesiony do dwóch kolejek, w zależności od tego czy był *odłączony*. Odłączenie informuje bibliotekę, że na wynik działania danego wątku nikt nie będzie oczekiwał i jego zasoby można zwolnić w każdej chwili, wątek można odłączyć za pomocą procedury `detach`. Kolejka `Q_detached` jest zaimplementowana w celach pomocniczych, ponieważ wątek nie może zwolnić sam swoich zasobów, bo z nich korzysta. Dlatego odpowiedzialność za ich zwalnianie została przeniesiona do zadań administracyjnych planisty.

Wątek może być tylko w jednej z kolejek `wait`, `sleep`, `detached`, `zombie` jednocześnie.

Nieodłączony wątek po swojej śmierci jest przeniesiony do kolejki żywych trupów (*zombie*). Dopiero gdy inny wątek wywoła procedurę `join` wątek zostanie odłączony.

Kolejka wątków działających jest cykliczna, planista przełącza wątki. Obecnie nie są zaimplementowane żadne priorytety, dodatkowo biblioteka aby zapewnić że kolejka `Q_run` nigdy nie będzie tworzy nic nie robiący wątek *idle*.

3.2.1. Zatrzymanie wątku.

W pewnych sytuacjach wątek za pomocą procedury `pwuthread_sched_wait` (lub odpowiednio `pwuthread_sched_timedwait` zgłasza planiście, że należy go zastopować. Trudno jest wykonywać tę operację natychmiastowo, dlatego planista posługuje się dodatkowym bitem `NEED_STOP` informującym, że przy następnym wywłaszczeniu wątek musi zostać zastopowany. Następnie procedury oczekują na wywłaszczenie.

Wątkom uśpionym na określony czas ustawia się budzik, który zawiera w sobie informację ile czasu dany wątek powinien jeszcze być uśpiony.

3.2.2. Sekcja krytyczna

Niektóre procedury uruchamiane przez wątki używają tych samych danych co program planisty, stąd tez potrzebna jest forma zapewniająca spójność danych. W bibliotece jest to rozwiązanie przez specjalną sekcję krytyczną. Wejście do sekcji krytycznej odbywa się przez procedurę `sched_lock`, a wyjście przez `sched_unlock`. Wewnątrz sekcji krytycznej ignorowane są wywłaszczenia planisty.

3.2.3. Zadania administracyjne.

Obecnie program planisty wykonuje dwa *zadania administracyjne*. Jest to, wspomniane już, niszczenie zasobów martwych odłączonych wątków oraz obsługę budzika śpiochów. Zadania są obsługiwane za każdym razem kiedy działa program planisty, czyli przy każdym wywłaszczeniu spowodowanym dostarczeniem sygnału czasomierza.

Planista oblicza ile czasu minęło od ostatniego wywłaszczenia. Następnie ta wartość jest odejmowana od budzików wszystkich wątków znajdujących się w kolejce `Q_sleep`. Przy wykryciu, że wątek śpi niemniej niż oczekiwano zostaje automatycznie wybudzany (przenoszony do kolejki `Q_run`). Mogłoby się wydawać, że obliczenia ile czasu minęło od ostatniego wywłaszczenia jest zabiegiem zbędnym ponieważ wiadomo co jaki czas następuje wywłaszczenie. Gdyby jednak nie obliczać czasu to planista byłby wrażliwy wszelkie opóźnienia. Przykładowo, gdyby zatrzymać taki proces na kilka minut, a następnie go wznowić to planista by tego nie zauważył i uznał, że wątki od ostatniego wywłaszczenia spały mały kwant czasu.

3.3. Tworzenie stosu wątku.

Każdy wątek (poza wątkiem głównym) wymaga przydzielania stosu. Zapotrzebowanie podprogramów uruchamianych w wątkach na stos może być znaczne, stąd też ustawienie domyślnego rozmiaru stosu na małą wartość może okazać się niepraktyczne. Przydzielanie natomiast dużych stosów może okazać się nieoptymalnym użyciem pamięci RAM w programie którego podprogramy mają zapotrzebowanie wynoszące tylko kilka kilobajtów.

Możliwe jest zrzucenie odpowiedzialności za optymalne użycie pamięci RAM na obsługę pamięci wirtualnej w systemie operacyjnym poprzez stworzenie *anonimowego*[4] odwzorowania pamięci. Dla tak utworzonego segmentu pamięci jądro może przydzielać strony na żądanie[5]. Domyślnie przyjęto rozmiar stosu wynoszący 32MB. Poniższa procedura jest odpowiedzialna za przydział stosu w bibliotece:

[4] Mechanizm służy do odwzorowywania plików w pamięci procesu, np bibliotek dzielonych, jeżeli przydzielany obszar nie jest związany z żadnym plikiem to nazywany jest *anonimowym*.

[5] Jak wspomniano, odpowiedzialność jest zrzucona na system operacyjny, więc program nie ma kontroli nad tym czy jądro rzeczywiście przydziela strony na żądanie czy cały segment od razu.

```
plik: support/unix_stack.c

int
pwuthread_stack_alloc(void **addr, size_t len)
{
    enum {
        prot = PROT_READ|PROT_WRITE,
        flags = MAP_ANON|MAP_PRIVATE
    };
    *addr = mmap(NULL, len, prot, flags, -1, 0);
    if (*addr == MAP_FAILED) {
        int e = errno;
        pwuthread_perror(e, "maping stack");
        return e;
    }
    return 0;
}
```

Proces powinien mieć co najmniej prawa zapisu i odczytu do stosu, lecz na komputerach domowych z powodów ograniczeń sprzętowych dołączona zostanie flaga `PROT_EXEC`, przez system, pozwalająca na wykonywanie instrukcji na stosie. Może to być istotny fragment dla bezpieczeństwa programu.

3.4. Synchronizacja.

Zaimplementowane mechanizmy korzystają z wspomnianej sekcji krytycznej planisty oraz z założenia że działa jednocześnie jeden wątek (nie ma możliwości użycia większej ilości procesorów, stąd to założenie).

Zamki typu `mutex` obsługują wewnątrz kolejkę FIFO, do której są dołączane wątki oczekujące na wejście do sekcji krytycznej (na zamknięcie zamka). Użycie kolejki FIFO wyklucza możliwość, że jeden wątek oczekujący na zamek będzie oczekiwał w nieskończoność, kiedy inne będą swobodnie synchronizować się na tym samym zamku. Podczas odblokowywania zamka jest wyznaczany jego nowy właściciel - jest nim najdłużej dotąd oczekujący wątek.

Zmienne warunkowe również operują na kolejkach FIFO zapamiętując wątki oczekujące na zdarzenie. Ponieważ wątki są usypiane będąc wewnątrz sekcji krytycznej to zapamiętywany w strukturze wątku jest również `mutex` za nią odpowiedzialny. Rozkaz obudzenia wątku jest implementowany przez przeniesienie go do kolejki wątków oczekujących na jego zablokowanie. Można to zinterpretować tak, że aby powrócić do sekcji krytycznej, w której wątek został uśpiony - biblioteka emuluje wywołanie `mutex_lock` po jego wybudzeniu.

3.5. Prywatne dane wątku.

Jak wspomniano wcześniej biblioteka umożliwia zapisywanie prywatnych wartości wątkom pod publiczne klucze za pomocą procedur `getspecific` oraz `setspecific`. O ile tworzenie i niszczenie kluczy może być uznane za operacje *ciężkie* to procedury pobierania i zapisywania wartości związanej z kluczem może mieć wpływ na wydajność programu, stąd powinny być *lekkie*.

Interfejs PTHREAD nakazuje aby domyślną wartością klucza był `NULL`. Jednak bardzo trudne w realizacji jest aby ustawiać tę wartość ręcznie wszystkim działającym już wątkom

w momencie tworzenia klucza. Standard umożliwia również ograniczenie kluczy przez stałą `KEYS_MAX`, wykorzystano ten fakt do efektywnej implementacji operacji na kluczach.

Klucze i wartości są opisywane przez trzy typy danych

```
struct pwuthread_keyinfo {
    int                 used;
    int                 serial;
    pwuthread_keydtor_f *dtor;
};

struct pwuthread_key {
    int                 index;
    int                 serial;
};

struct pwuthread_value {
    int                 serial;
    pwuthread_key_t     key;
    void               *value;
};
```

Typ `keyinfo` służy do opisywania przydzielonego *slotu* na klucz, typ `key` to deskryptor klucza jakim posługuje się użytkownik biblioteki. Wartości przypisane do kluczy w poszczególnych wątkach są opisywane przez `value`.

Każdy klucz musi być związany z jakimś slotem oraz numerem seryjnym, który służy do rozpoznawania czy wartość pod danym kluczem jest aktualna.

```
void f() {
    pwuthread_setspecific(global_key, &wsk_a);
    pwuthread_util_sleep(100);
    printf("%p\n", pwuthread_getspecific(global_key));
}
```

W celu zobrazowania przydatności numerów seryjnych rozważmy sytuacje działania powyższej procedury. Wątek zapisze prywatną wartość `&wsk_a` pod klucz opisywany przez deskryptor `global_key`. Następnie uda się na sto sekundową drzemkę. W tym czasie inny wątek może wykonać poniższe instrukcje, niszczące klucz i tworzące pod tym samym deskryptorem nowy.

```
    pwuthread_key_delete(global_key)
    pwuthread_key_create(&global_key, NULL);
```

Po obudzeniu się z drzemki wątek wydrukuje wartość pobraną spod klucza. Ponieważ klucz jest nowy, to dany wątek nie powinien mieć skojarzonej z nim żadnej wartości i zgodnie ze standardem powinna zostać zwrócona wartość domyślna `NULL`.

Biblioteka posługuje się tablicą `keyinfo` o `KEYS_MAX` elementach w celu zarządzania slotami. Przy tworzeniu klucza brany jest wolny slot tablicy i zwiększany jego numer seryjny. Następnie informacje o slocie i kluczu seryjnym są kopiowane do deskryptora jakim posługuje się użytkownik.

3.6. Microsoft Windows

Niestety autor nie umiał wykorzystać interfejsu systemu Windows aby zaimplementować niskopoziomowe podstawy tej biblioteki. Nie znany jest odpowiednik wywołania `sigaltstack` ze standardu POSIX pozwalającego wykorzystać sygnały do zmiany stosu. Dodatkowo, planista opiera się na semantyce, z wymienionego standardu, mechanizmu dostarczającego sygnały do procesu - czasomierz generuje sygnał `SIGALRM`, którego dostarczenie wywłaszcza proces. Na systemie Windows nie jest obecny mechanizm wywłaszczania przy dostarczaniu sygnału - procedura obsługująca zdarzenia czasomierza jest uruchamiana w nowym wątku.

Wraz z pojawieniem się systemu Windows 2000 został wprowadzony interfejs do tworzenia kooperatywnych wątków użytkownika o nazwie Fiber[6]

4. Inne implementacje wątków użytkownika.

PTH - Rozwijana do dziś biblioteka wątków. Implementuje interfejs *PTHREAD* oraz własny. Jej autorom udało się zapewnić działanie na bardzo wielu platformach, a w tym także na systemie Microsoft Windows. Biblioteka obsługuje koopertywny model zarządzania wątkami [6].

PTL - Nie rozwijana dziś biblioteka wątków, implementująca interfejs *PTHREAD*. Bibliotekę cechowała duża ilość nakładek na wywołania systemowe, co czyniło ją bardzo praktyczną. Biblioteka posiadała innowacyjny debugger do wątków, mianowicie program używający biblioteki stawał się także serwerem. Przygotowany program mógł się do niego podłączyć i monitorować stan wątków. Biblioteka obsługuje model wątków, opierający się o wywłaszczenia [7].

BSD - W niektórych systemach wywodzących się z BSD dostępnych jest wiele różnych bibliotek implementujących interfejs *PTHREAD*. Są one dostosowane pod konkretny system operacyjny, a ich źródła są dostępne w dystrybucji systemu w katalogu */usr/src/lib*. Na systemach *FreeBSD* oraz *NetBSD* jest to biblioteka *libc_r*, a na *OpenBSD* jest to *libpthread*[7]. Planista zarządzający wątkami w tych bibliotekach opiera się o wywłaszczenia, a wraz z załadowaniem biblioteki wszystkie procedury odpowiedzialne za blokujące wywołania systemowe są zastępowane przez nakładki.

Literatura

[1] IEEE Std 1003.1-2004 (POSIX:2004). Threads
`http://www.opengroup.org/onlinepubs/000095399/functions/xsh_chap02_09.html`
`http://www.opengroup.org/onlinepubs/000095399/basedefs/pthread.h.html`

[2] IEEE Std 1003.1-2004 (POSIX:2004). Signals
`http://www.opengroup.org/onlinepubs/000095399/functions/xsh_chap02_04.html`
`http://www.opengroup.org/onlinepubs/009695399/functions/sigaction.html`
`http://www.opengroup.org/onlinepubs/000095399/functions/sigaltstack.html`

[6] Nazwa jest analogią do słowa *thread*, które w języku angielskim znaczy nić, słowo *fiber* znaczy włókno.

[7] Nowocześniejsze biblioteki wykorzystujące wątki po stronie jądra nazwane są odpowiednio: na *FreeBSD* - *libthr*, *libpthread*; na *NetBSD* - *libpthread*, na *OpenBSD* - *librthread*

[3] IEEE Std 1003.1-2004 (POSIX:2004). Ucontext
http://www.opengroup.org/onlinepubs/009695399/basedefs/ucontext.h.html

[4] IEEE Std 1003.1-2004 (POSIX:2004). Interval timer
http://www.opengroup.org/onlinepubs/009695399/functions/getitimer.html

[5] IEEE Std 1003.1-2008 (POSIX:2008). Rationale for System Interface. Introduction.
http://www.opengroup.org/onlinepubs/9699919799/xrat/V4_xsh_chap01.html

[6] The GNU Portable Thread
http://www.gnu.org/software/pth/

[7] Portable Thread Library
http://www.media.osaka-cu.ac.jp/~k-abe/PTL/