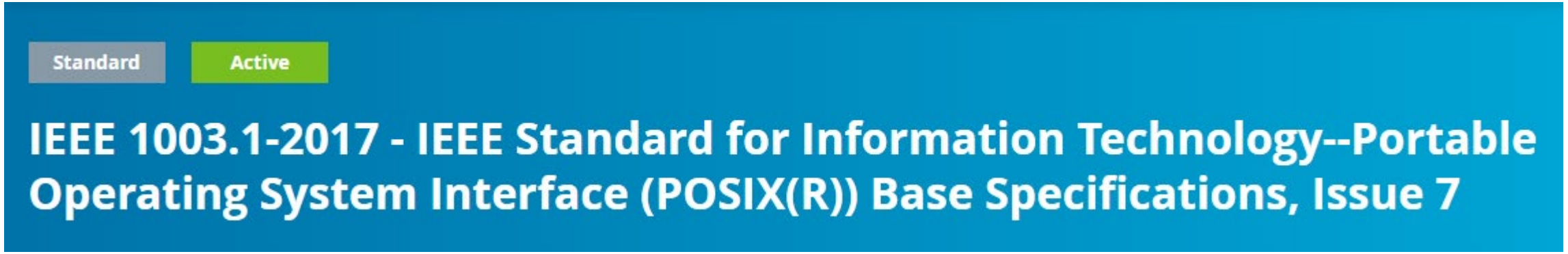
pthreads (POSIX)

implementacji równoległości – poprzez wątki w architekturach wieloprocesorowych z pamięcią współdzieloną
przenośność – problem programistyczny, gdy dostawcy sprzętu wdrażali własne wersje wątków
unix – standardowy interfejs programowania w języku C został określony przez standard IEEE POSIX 1003.1-x

Implementacje zgodne z tym standardem są nazywane **wątkami POSIX** lub **Pthreads**.



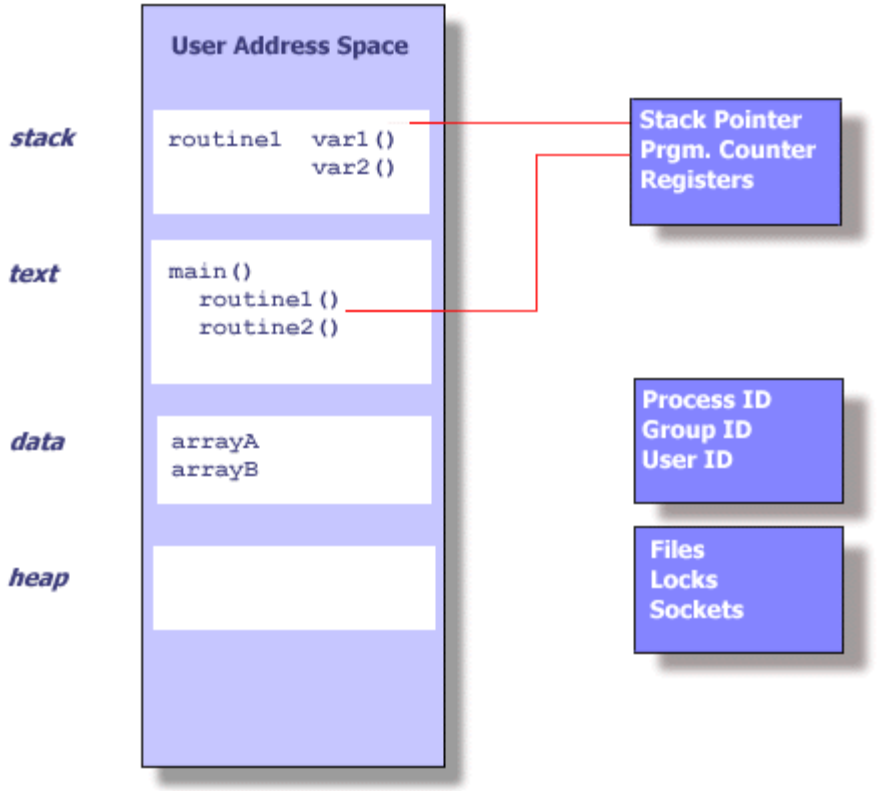
STANDARD https://standards.ieee.org/standard/1003_1-2017.html
POSIX FAQ http://www.opengroup.org/austin/papers/posix_faq.html
WIKIPEDIA https://en.wikipedia.org/wiki/POSIX_Threads

Co to jest wątek i proces

- wątek – niezależny strumień instrukcji, które mogą być zaplanowane do uruchomienia przez system operacyjny
- wątek jako pojęcie dla programistów – koncepcja „procedury” działającej niezależnie od programu głównego
- program „wielowątkowy” – program główny (a.out), który zawiera wiele procedur, wszystkie te procedury mogą być zaplanowane do równoczesnego uruchamiania, i / lub niezależnie przez system operacyjny

Proces UNIX. Proces jest tworzony przez system operacyjny i ma sporo „kosztów ogólnych”. Procesy zawierają informacje o zasobach programu i stanie wykonywania programu, w tym:

- Identyfikatory: procesu, grupy procesów, użytkownika i grupy
- Środowisko
- Katalog roboczy
- Instrukcje programu
- Rejestry
- Stos
- Sterta
- Deskryptory plików
- Akcje sygnałowe
- Współdzielone biblioteki
- Narzędzia komunikacji międzyprocesowej (kolejki komunikatów, potoki, semafory lub pamięć współdzielona)



Co to jest wątek

Wątek UNIX. Wątki używają i istnieją w zasobach procesowych, ale mogą być planowane przez system operacyjny i uruchamiane jako niezależne jednostki, głównie dlatego, że duplikują tylko podstawowe zasoby, które umożliwiają ich istnienie jako kod wykonywalny.

Niezależny przepływ sterowania jest realizowany, ponieważ wątek utrzymuje swój własny:

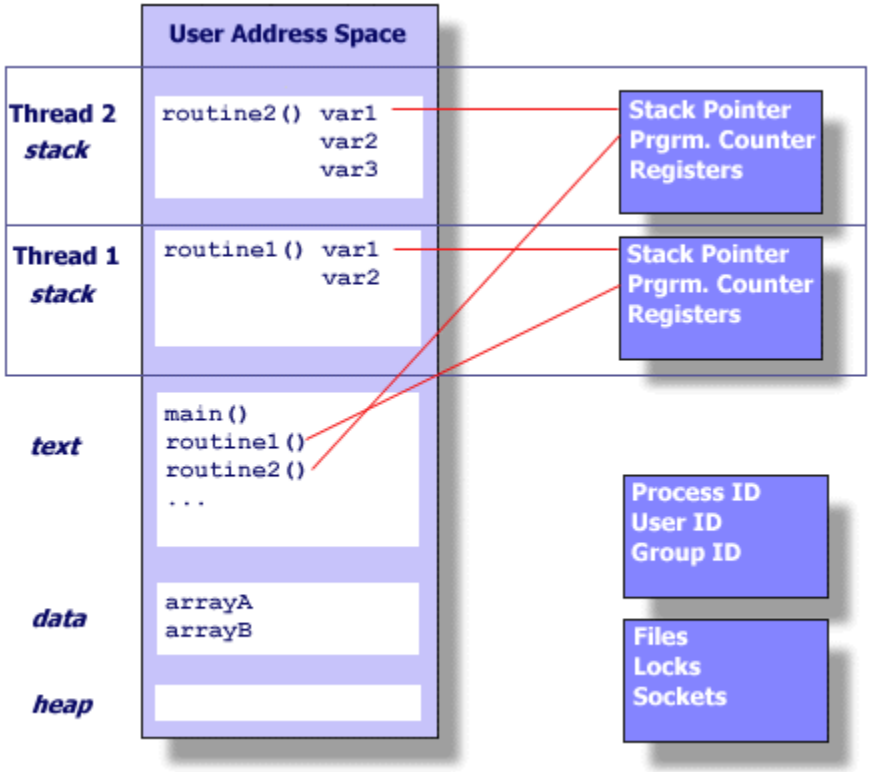
- Wskaźnik stosu
- Rejestry
- Planowanie właściwości (strategia, priorytet)
- Zestaw oczekujących i zablokowanych sygnałów
- Dane specyficzne dla wątku

W środowisku UNIX wątek:

- Występuje w procesie i wykorzystuje zasoby procesu
- Ma swój własny niezależny przepływ sterowania, o ile istnieje proces nadrzędny i system operacyjny go obsługuje
- Duplikuje tylko podstawowe zasoby, które musi mieć aby być zaplanowany i wykonywany niezależnie
- Może udostępniać zasoby procesu innym wątkom, które działają równie niezależnie (lub zależnie)
- Umiera, jeśli proces macierzysty umrze
- Jest „lekki”, ponieważ większość kosztów ogólnych została już osiągnięta dzięki stworzeniu procesu

Ponieważ wątki w tym samym procesie współdzielą zasoby:

- Zmiany dokonane przez jeden wątek na współużytkowane zasoby systemowe (np. zamknięcie pliku) są widoczne dla wszystkich innych wątków
- Dwa wskaźniki o tej samej wartości wskazują te same dane
- Odczyt i zapis do tych samych lokalizacji pamięci jest możliwy i dlatego wymaga synchronizacji przez programistę



Czym są pthreads?

- Historycznie, dostawcy sprzętu wdrażali własne wersje wątków. Te implementacje różniły się znacznie od siebie, co utrudniało programistom tworzenie przenośnych aplikacji wielowątkowych.
- Pełne wykorzystanie możliwości oferowane przez wątki – wymagany standardowy interfejs programowania.
  - W UNIX interfejs został określony przez standard IEEE POSIX 1003.1c (1995)
  - Implementacje zgodne z tym standardem są nazywane wątkami POSIX lub Pthreads.
  - Większość dostawców sprzętu oferuje Pthreads jako dodatek do zastrzeżonych API.
- Standard POSIX ewoluował i podlega zmianom, w tym specyfikacje Pthreads.

- Pthreads są zdefiniowane jako zestaw typów programowania w języku C i wywołań procedur, zaimplementowanych z nagłówkiem pthread.h (plikiem dołączanym) i biblioteką wątków - chociaż ta biblioteka może być częścią innej biblioteki, takiej jak `libc`, w niektórych implementacjach.

Dlaczego pthreads?

- W porównaniu z kosztami tworzenia i zarządzania procesem można utworzyć wątek o znacznie mniejszym obciążeniu systemu operacyjnego. Zarządzanie wątkami wymaga mniej zasobów systemowych niż zarządzanie procesami.
- Na przykład poniższa tabela porównuje wyniki czasowe dla podprogramu `fork()` i podprogramu `pthread_create()`. Czasy odzwierciedlają 50 000 kreacji procesu / wątku, zostały wykonane za pomocą biblioteki `time`, jednostki są w sekundach, bez flag optymalizacji.

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

- Czasy: systemowy (sys) i użytkownika (user) nie sumują się do czasu rzeczywistego (real), ponieważ obliczenia wykonano na maszynach SMP (Symmetric MultiProcessing) z wieloma procesorami / rdzeniami, pracującymi nad problemem równolegle. Wyniki są przybliżonymi estymatami, działającymi na lokalnych maszynach.

Komunikacja / wymiana danych

- Motywacją do użycia Pthreads w środowisku obliczeniowym o wysokiej wydajności jest osiągnięcie optymalnej wydajności. W szczególności, jeśli aplikacja używa MPI do komunikacji przez węzły, istnieje możliwość, że wydajność może zostać poprawiona poprzez użycie Pthreads.
- Biblioteki MPI zazwyczaj implementują komunikację zadań na węzłach za pośrednictwem pamięci współdzielonej, co oznacza co najmniej jedną operację kopiowania pamięci (od procesu do procesu).
- W przypadku Pthreads nie jest wymagana kopia pamięci, ponieważ wątki współdzielą tę samą przestrzeń adresową w ramach jednego procesu. Nie ma transferu danych, co najwyżej przekazanie wskaźnika.
- W najgorszym przypadku komunikacja Pthread staje się bardziej problemem przepustowości pamięci: cache-to-CPU lub memory-to-CPU. Prędkości te są znacznie wyższe niż w przypadku komunikacji w pamięci współdzielonej MPI. Na przykład:

Platform	MPI Shared Memory Bandwidth (GB/sec)	Pthreads Worst Case Memory-to-CPU Bandwidth (GB/sec)
Intel 2.6 GHz Xeon E5-2670	4.5	51.2
Intel 2.8 GHz Xeon 5660	5.6	32
AMD 2.3 GHz Opteron	1.8	5.3
AMD 2.4 GHz Opteron	1.2	5.3
IBM 1.9 GHz POWER5 p5-575	4.1	16
IBM 1.5 GHz POWER4	2.1	4
Intel 2.4 GHz Xeon	0.3	4.3
Intel 1.4 GHz Itanium 2	1.8	6.4

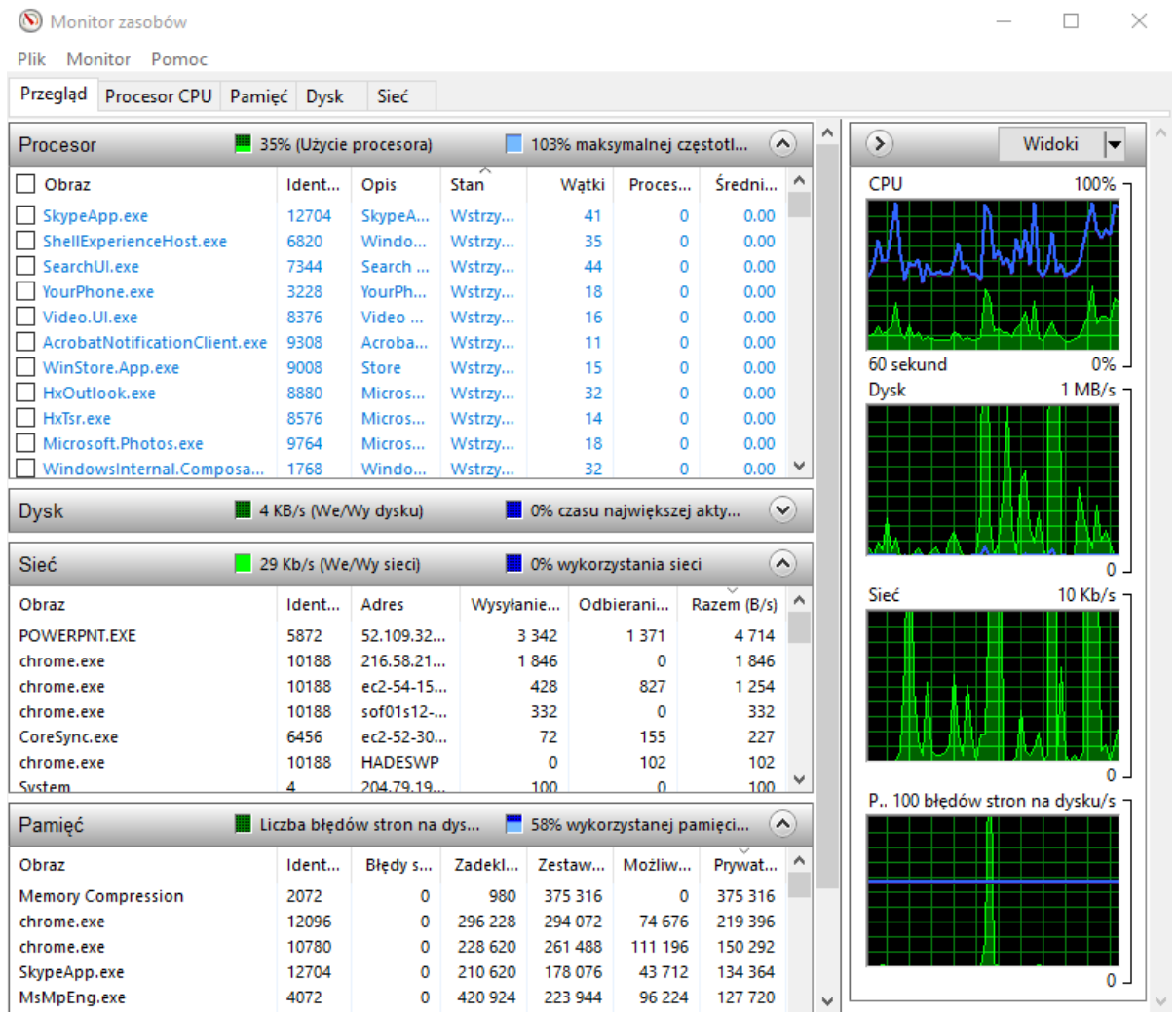
Inne powody używania Pthreads

Aplikacje wielowątkowe oferują potencjalny wzrost wydajności i przewagę nad aplikacjami jednowątkowymi:

- Równoległe działanie CPU i operacji We/Wy (I/O): program może mieć sekcje, w których wykonuje długą operację I/O. Podczas gdy jeden wątek czeka na zakończenie operacji I/O, procesor może być wykorzystany przez inne wątki.
- Priorytet / planowanie w czasie rzeczywistym: zadania, które są ważniejsze, mogą być zaplanowane do zastąpienia lub przerwania zadań o niższym priorytecie.
- Obsługa zdarzeń asynchronicznych: zadania, które obsługują zdarzenia o nieokreślonej częstotliwości i czasie trwania, mogą się przeplatać. Np. serwer WWW może zarówno przesyłać dane z poprzednich żądań, jak i zarządzać nadejściem nowych żądań.

Doskonałym przykładem jest przeglądarka internetowa, w której wiele równoległych zadań może się wykonywać w tym samym czasie, zadania mogą mieć też różne priorytety.

Innym dobrym przykładem jest nowoczesny system operacyjny, który w szerokim zakresie wykorzystuje wątki. Po prawej: zrzut ekranu systemu operacyjnego Windows (Monitor zasobów).



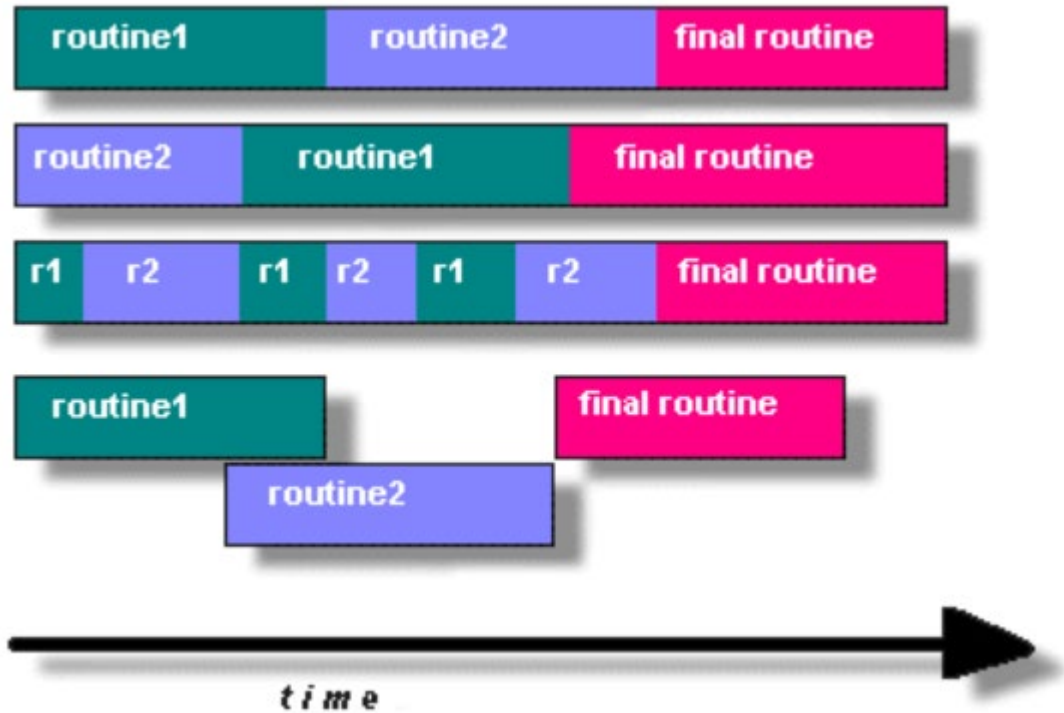
Projektowanie programów wielowątkowych

W nowoczesnych, wielordzeniowych maszynach pthreads idealnie nadają się do programowania równoległego, a cokolwiek dotyczy programowania równoległego, dotyczy również równoległych programów pthreads.

Istnieje wiele rozważań dotyczących projektowania programów równoległych:

- Jaki typ modelu programowania równoległego ma być używany?
- Problem z partycjonowaniem
- Równoważenie obciążenia
- Komunikacja
- Zależności danych
- Synchronizacja i "race conditions"
- Problemy z pamięcią
- Problemy z I/O
- Złożoność programu
- Wysiłek programisty / koszty / czas

Ogólnie jednak, aby program mógł korzystać z Pthreads, musi być on zorganizowany w dyskretne, niezależne zadania, które mogą być wykonywane jednocześnie. Na przykład, jeśli routine1 i routine2 mogą być wymieniane, przeplatane i / lub nakładane w czasie rzeczywistym, są kandydatami do realizacji wielowątkowej.



Pthreads w programach

Programy o następujących cechach mogą być dobre do wykorzystania wątków pthreads:

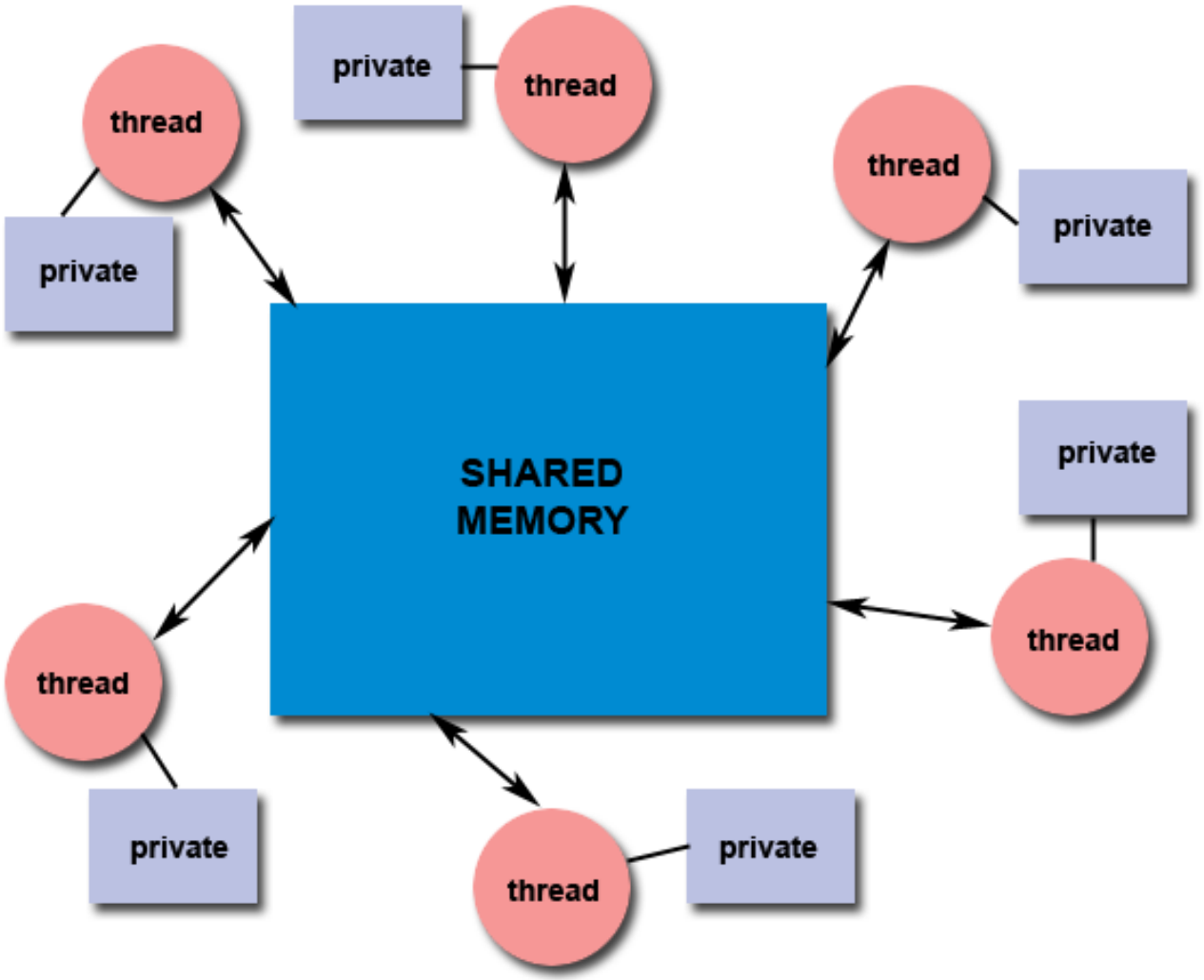
- Praca, którą można wykonać, lub dane, które mogą być obsługiwane przez wiele zadań jednocześnie
- Zadanie, które jest potencjalnie zablokowane przez długie oczekiwania na operacje I/O
- Użycie wielu cykli procesora w niektórych miejscach, a w innych nie
- Musi reagować na zdarzenia asynchroniczne
- Niektóre prace są bardziej priorytetowe niż inne (przerwania priorytetowe)

Kilka popularnych modeli dla programów wielowątkowych:

- **Menedżer/pracownik**: pojedynczy wątek, **menedżer** przypisuje pracę do innych wątków, **pracowników**. Zazwyczaj menedżer obsługuje wszystkie dane wejściowe i rozdziela pracę do innych zadań. Wspólne są co najmniej dwie formy modelu menedżera/pracownika: statyczna i dynamiczna pula pracowników.
- **Pipeline**: zadanie jest podzielone na serię podoperacji, z których każda jest obsługiwana szeregowo, ale jednocześnie współbieżnie przez inny wątek, np. linia montażowa samochodów.
- **Peer**: podobny do modelu menedżera/pracownika, ale po tym, jak główny wątek stworzy inne wątki, również uczestniczy w pracy.

Model pamięci współdzielonej

- Wszystkie wątki mają dostęp do tej samej globalnej pamięci współdzielonej
- Wątki mają również swoje prywatne dane
- Programiści są odpowiedzialni za synchronizowanie dostępu (chronienie) globalnie udostępnionych danych.

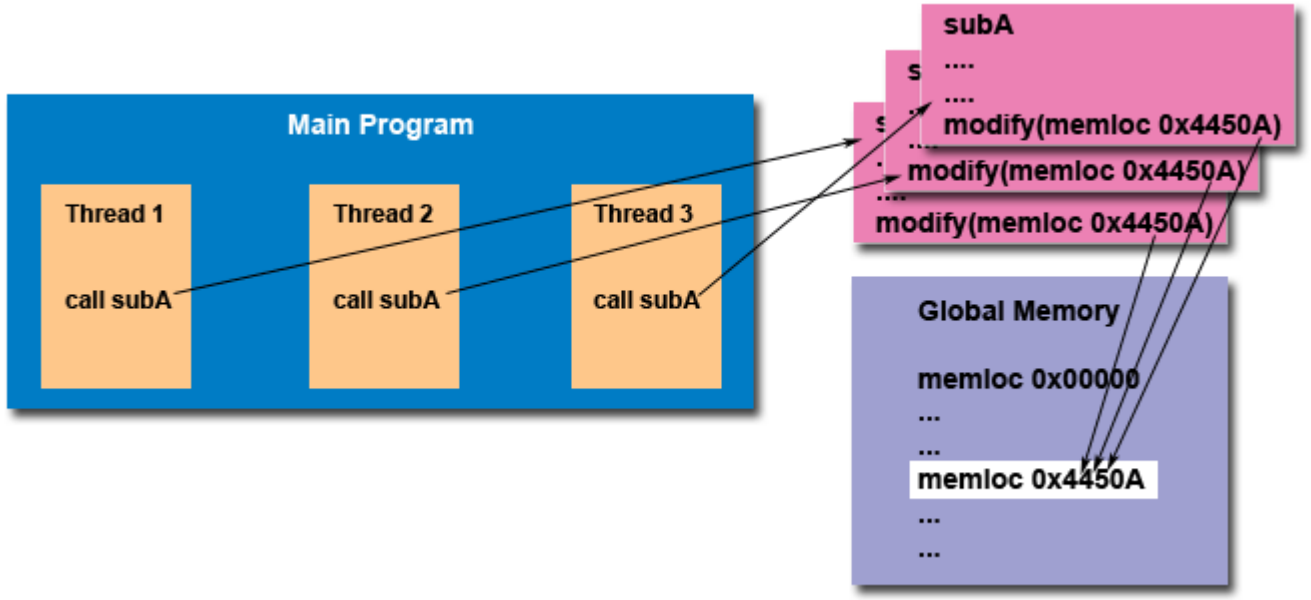


Bezpieczeństwo wątków

- Odnosi się do zdolności aplikacji do jednoczesnego wykonywania wielu wątków bez nadpisywania udostępnionych danych lub tworzenia warunków „wyścigu” (data race).
- Niech aplikacja tworzy kilka wątków, z których każdy wywołuje tę samą procedurę biblioteczną:
- Ta procedura biblioteczna uzyskuje dostęp do / modyfikuje globalną strukturę w pamięci.
- Ponieważ każdy wątek wywołuje tę procedurę, możliwe jest, że spróbują zmodyfikować tę globalną lokalizację struktury / pamięci w tym samym czasie.
- Jeśli procedura nie wykorzystuje jakiegoś rodzaju konstrukcji synchronizujących, aby zapobiec uszkodzeniu danych, nie jest bezpieczna dla wątków.

Chociaż API Pthreads jest standardem ANSI / IEEE, implementacje mogą i zazwyczaj się zmieniają w sposób nie określony przez standard.
Z tego powodu program działający poprawnie na jednej platformie może zawieść lub spowodować błędne wyniki na innej platformie.

- Implikacje dla użytkowników zewnętrznych procedur bibliotecznych: jeśli nie jesteś w 100% pewien, że procedura jest bezpieczna dla wątków, ryzykujesz problemy, które mogą się pojawić.
- Zalecenie: Zachowaj ostrożność, jeśli aplikacja używa bibliotek lub innych obiektów, które nie gwarantują bezpieczeństwa wątków. W razie wątpliwości załóż, że nie są bezpieczne dla wątków, dopóki nie udowodni się inaczej. Można to zrobić, „serializując” wywołania do niepewnej procedury itp.



API Pthreads

Oryginalny interfejs API Pthreads został zdefiniowany w normie ANSI / IEEE POSIX 1003.1 - 1995.
Standard POSIX ewoluował i podlega zmianom, w tym specyfikacji Pthreads.
Procedury, które zawierają API Pthreads, można ująć w cztery główne grupy:

- **Zarządzanie** wątkami**:** Procedury działające bezpośrednio na wątkach - tworzenie, odłączanie, łączenie itp. Obejmują one również funkcje do ustawiania /odpytania atrybutów wątków.
- **Mutexes:** procedury, które zajmują się synchronizacją, nazywane „mutexami", („wzajemne wykluczanie"). Funkcje Mutex umożliwiają tworzenie, niszczenie, blokowanie i odblokowywanie muteksów. Są one uzupełniane przez funkcje atrybutów mutex, które ustawiają lub modyfikują atrybuty związane z muteksami.
- **Zmienne warunkowe:** procedury adresujące komunikację między wątkami, które współdzielą muteks, na podstawie określonych warunków programisty. Ta grupa zawiera funkcje do tworzenia, niszczenia, oczekiwania i sygnalizowania na podstawie określonych wartości zmiennych. Dołączone są również funkcje ustawiania / sprawdzania atrybutów zmiennych warunkowych.
- **Synchronizacja:** procedury, które zarządzają blokadami odczytu i zapisu i barierami.

Konwencje nazewnictwa: Wszystkie identyfikatory w bibliotece wątków zaczynają się od **pthread_** .
Interfejs API Pthreads zawiera około 100 procedur.
Plik nagłówkowy pthread.h powinien być dołączony do każdego pliku źródłowego przy użyciu biblioteki Pthreads.
Bieżący standard POSIX jest zdefiniowany tylko dla języka C.

API Pthreads – przykłady

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects.
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys
pthread_rwlock_	Read/write locks
pthread_barrier_	Synchronization barriers

Kompilacja programów z Pthreads

Kilka przykładów komend kompilacyjnych używanych dla kodów pthreads znajduje się w poniższej tabeli.

Compiler / Platform	Compiler Command	Description
INTEL Linux	`icc -pthread`	C
	`icpc -pthread`	C++
PGI Linux	`pgcc -lpthread`	C
	`pgCC -lpthread`	C++
GNU Linux, Blue Gene	`gcc -pthread`	GNU C
	`g++ -pthread`	GNU C++
IBM Blue Gene	`bgxlc_r / bgcc_r`	C (ANSI / non-ANSI)
	`bgxlC_r, bgxlc++_r`	C++

Zarządzanie wątkami

Tworzenie i kończenie wątków

Sprawdzanie / ustawianie
```
ulimit -a
ulimit -Hu
ulimit -u <wartość>
```

```
pthread_create (thread,attr,start_routine,arg)

pthread_exit (status)

pthread_cancel (thread)

pthread_attr_init (attr)

pthread_attr_destroy (attr)
```

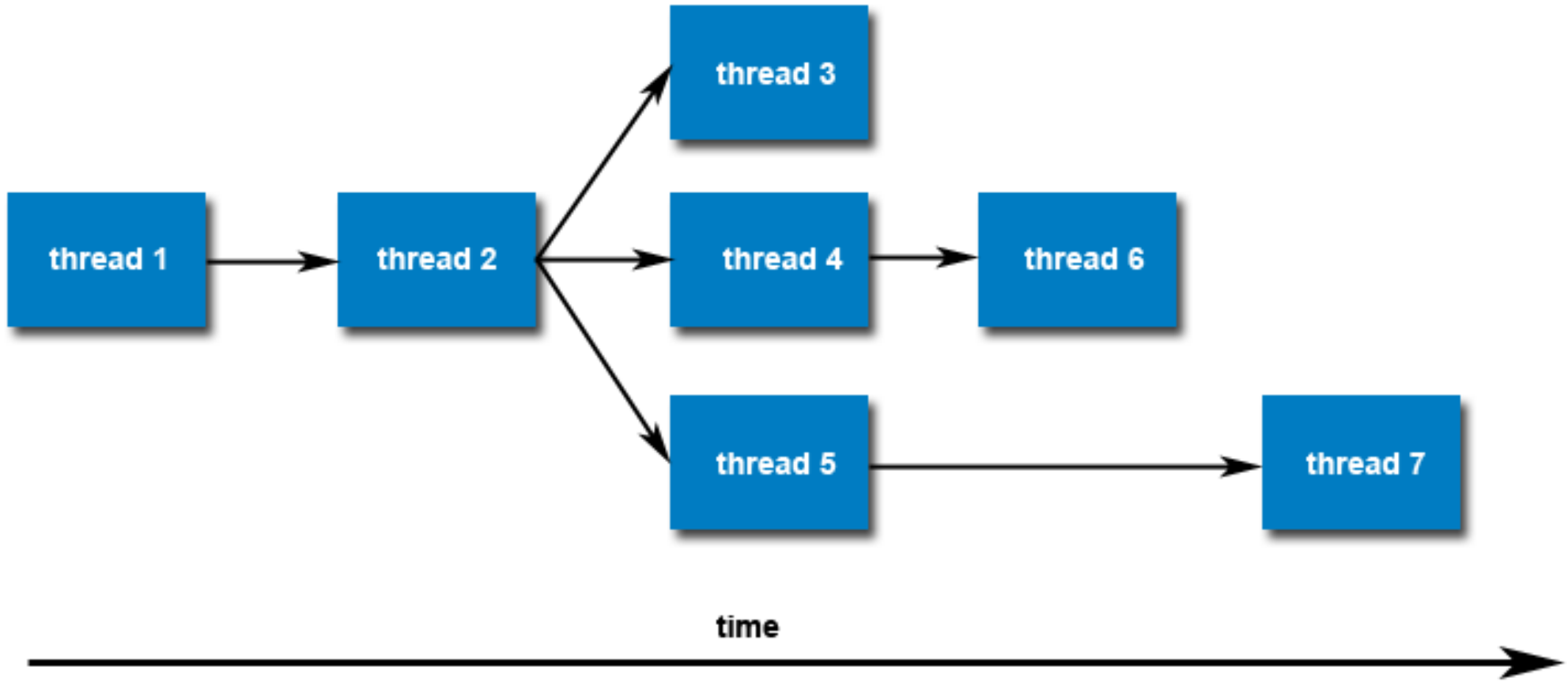
Program `main()` zawiera pojedynczy domyślny wątek. Wszystkie inne wątki muszą być jawnie utworzone przez programistę.

- pthread_create tworzy nowy wątek i czyni go wykonywalnym. Tę procedurę można wywołać dowolną liczbę razy z dowolnego miejsca w kodzie.
- Argumenty pthread_create :
  - thread : unikalny identyfikator nowego wątku zwróconego przez procedurę
  - attr : obiekt atrybutu, który może być używany do ustawiania atrybutów wątku. Możesz określić obiekt atrybutów wątku lub NULL dla wartości domyślnych.
  - start_routine : procedura C, którą wątek wykona po utworzeniu.
  - arg : Pojedynczy argument, który może zostać przekazany do start_routine. Musi być przekazany jako wskaźnik (rzutowanie do) typu void. NULL może być użyty, jeśli nie ma żadnego argumentu.

Maksymalna liczba wątków, które mogą zostać utworzone przez proces, zależy od implementacji. Programy, które próbują przekroczyć limit, mogą zawieść lub dać błędne wyniki.

Zależności między wątkami

Po utworzeniu wątki są równorzędne i mogą tworzyć inne wątki. Nie ma domniemanej hierarchii ani zależności między wątkami.



Przykład: tworzenie i kończenie wątku

Prosty przykładowy kod tworzy 5 wątków procedurą `pthread_create()`. Każdy wątek drukuje "Hello World!", a następnie kończy się wywołaniem `pthread_exit()`.

Przykładowy wynik:

```
In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
Hello World! It's me, thread #0!
Hello World! It's me, thread #1!
In main: creating thread 3
Hello World! It's me, thread #2!
In main: creating thread 4
Hello World! It's me, thread #3!
Hello World! It's me, thread #4!
```

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0;t<NUM_THREADS;t++){
     printf("In main: creating thread %ld\n", t);
     rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
     if (rc){
       printf("ERROR; return code from pthread_create() is %d\n", rc);
       exit(-1);
       }
     }

   pthread_exit(NULL);
}
```

Przekazywanie argumentów do wątków

Funkcja `pthread_create()` pozwala programiście przekazać jeden argument do procedury uruchamiania wątku. W przypadkach, w których trzeba przekazać wiele argumentów, ograniczenie to można łatwo pokonać, tworząc strukturę zawierającą wszystkie argumenty, a następnie przekazując wskaźnik do tej struktury w procedurze `pthread_create()`.

Wszystkie argumenty muszą być przekazywane po rzutowaniu na `(void*)`.

W jaki sposób można bezpiecznie przekazywać dane do nowo utworzonych wątków, biorąc pod uwagę ich niedeterministyczne uruchamianie i planowanie?
Upewnij się, że wszystkie przekazane dane są bezpieczne dla wątków - nie można ich zmienić przez inne wątki.

Przekazywanie argumentów do wątków: przykład

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 8

char *messages[NUM_THREADS];

void *PrintHello(void *threadid)
{
   long taskid;

   sleep(1);
   taskid = (long) threadid;
   printf("Thread %d: %s\n", taskid, messages[taskid]);
   pthread_exit(NULL);
}
```

```c
int main(int argc, char *argv[])
{
pthread_t threads[NUM_THREADS];
long taskids[NUM_THREADS];
int rc, t;

messages[0] = "English: Hello World!";
messages[1] = "French: Bonjour, le monde!";
messages[2] = "Spanish: Hola al mundo";
messages[3] = "Klingon: Nuq neH!";
messages[4] = "German: Guten Tag, Welt!";
messages[5] = "Russian: Zdravstvuyte, mir!";
messages[6] = "Japan: Sekai e konnichiwa!";
messages[7] = "Latin: Orbis, te saluto!";

for(t=0;t<NUM_THREADS;t++) {
  taskids[t] = t;
  printf("Creating thread %d\n", t);
  rc = pthread_create(&threads[t], NULL, PrintHello, (void *) taskids[t]);
  if (rc) {
    printf("ERROR; return code from pthread_create() is %d\n", rc);
    exit(-1);
    }
  }

pthread_exit(NULL);
}
```

```
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Creating thread 5
Creating thread 6
Creating thread 7
Thread 0: English: Hello World!
Thread 2: Spanish: Hola al mundo
Thread 3: Klingon: Nuq neH!
Thread 5: Russian: Zdravstvuyte, mir!
Thread 1: French: Bonjour, le monde!
Thread 4: German: Guten Tag, Welt!
Thread 6: Japan: Sekai e konnichiwa!
Thread 7: Latin: Orbis, te saluto!
```

Ten fragment kodu pokazuje, jak przekazać prostą liczbę całkowitą do każdego wątku. Wątek wywołujący używa unikalnej struktury danych dla każdego wątku, zapewniając, że argument każdego wątku pozostanie nienaruszony w całym programie.

Przekazywanie argumentów do wątków: przykład

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 8

char *messages[NUM_THREADS];

struct thread_data
{
   int  thread_id;
   int  sum;
   char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)
{
   int taskid, sum;
   char *hello_msg;
   struct thread_data *my_data;

   sleep(1);
   my_data = (struct thread_data *) threadarg;
   taskid = my_data->thread_id;
   sum = my_data->sum;
   hello_msg = my_data->message;
   printf("Thread %d: %s  Sum=%d\n", taskid, hello_msg, sum);
   pthread_exit(NULL);
}
```

Ten przykład pokazuje, jak skonfigurować / przekazać wiele argumentów poprzez strukturę. Każdy wątek otrzymuje unikalną instancję struktury.

```c
int main(int argc, char *argv[])
{
pthread_t threads[NUM_THREADS];
int *taskids[NUM_THREADS];
int rc, t, sum;

sum=0;
messages[0] = "English: Hello World!";
messages[1] = "French: Bonjour, le monde!";
messages[2] = "Spanish: Hola al mundo";
messages[3] = "Klingon: Nuq neH!";
messages[4] = "German: Guten Tag, Welt!";
messages[5] = "Russian: Zdravstvytye, mir!";
messages[6] = "Japan: Sekai e konnichiwa!";
messages[7] = "Latin: Orbis, te saluto!";

for(t=0;t<NUM_THREADS;t++) {
  sum = sum + t;
  thread_data_array[t].thread_id = t;
  thread_data_array[t].sum = sum;
  thread_data_array[t].message = messages[t];
  printf("Creating thread %d\n", t);
  rc = pthread_create(&threads[t], NULL, PrintHello, (void *)
       &thread_data_array[t]);
  if (rc) {
    printf("ERROR; return code from pthread_create() is %d\n", rc);
    exit(-1);
    }
  }
pthread_exit(NULL);
}
```

```
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Creating thread 5
Creating thread 6
Creating thread 7
Thread 0: English: Hello World!  Sum=0
Thread 1: French: Bonjour, le monde!  Sum=1
Thread 2: Spanish: Hola al mundo  Sum=3
Thread 3: Klingon: Nuq neH!  Sum=6
Thread 4: German: Guten Tag, Welt!  Sum=10
Thread 5: Russian: Zdravstvytye, mir!  Sum=15
Thread 6: Japan: Sekai e konnichiwa!  Sum=21
Thread 7: Latin: Orbis, te saluto!  Sum=28
```

Przekazywanie argumentów do wątków: przykład

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS     8

void *PrintHello(void *threadid)
{
   long taskid;
   sleep(1);
   taskid = *(long *)threadid;
   printf("Hello from thread %ld\n", taskid);
   pthread_exit(NULL);
}
```

Ten przykład wykonuje niepoprawny argument. Przekazuje adres zmiennej t , która jest przestrzenią pamięci współdzielonej i widoczną dla wszystkich wątków. W miarę iteracji pętli zmienia się wartość tej lokalizacji pamięci, prawdopodobnie zanim utworzone wątki będą miały do niej dostęp.

```c
int main(int argc, char *argv[])
{
pthread_t threads[NUM_THREADS];
int rc;
long t;

for(t=0;t<NUM_THREADS;t++) {
  printf("Creating thread %ld\n", t);
  rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &t);
  if (rc) {
    printf("ERROR; return code from pthread_create() is %d\n", rc);
    exit(-1);
    }
   }

pthread_exit(NULL);
}
```

```
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Creating thread 5
Creating thread 6
Creating thread 7
Hello from thread 8
Hello from thread 8
Hello from thread 8
Hello from thread 8
Hello from thread 8
Hello from thread 8
Hello from thread 8
Hello from thread 8
```

Zarządzanie wątkami

Łączenie i odłączanie wątków

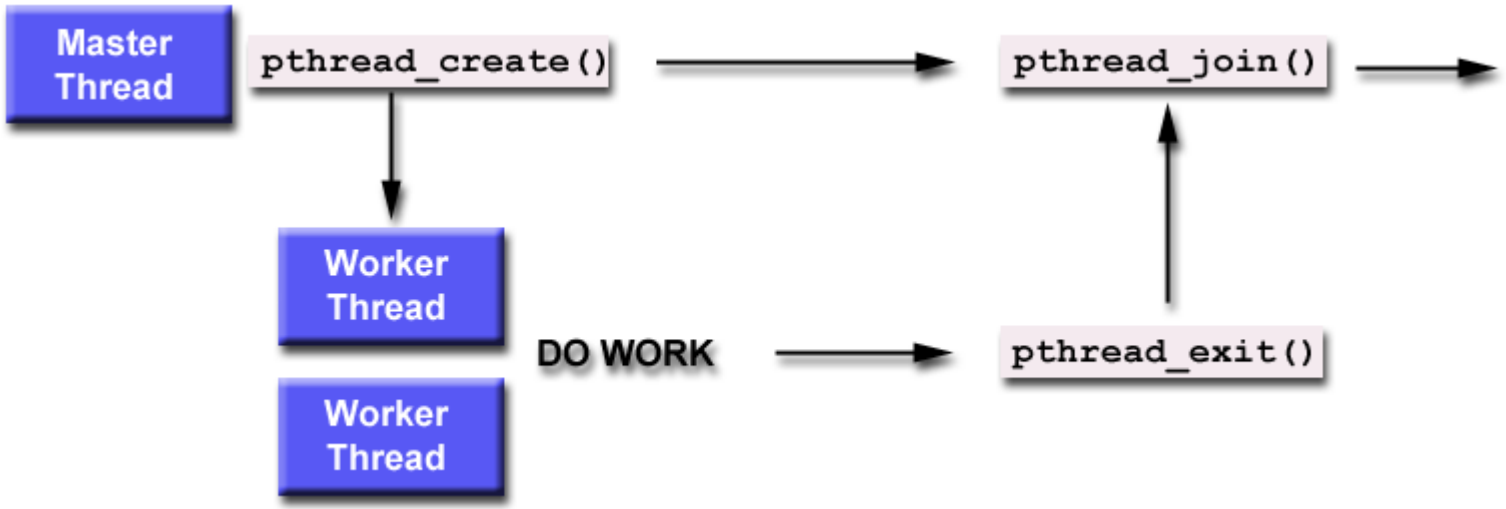
```
pthread_join (threadid,status)

pthread_detach (threadid)

pthread_attr_setdetachstate (attr,detachstate)

pthread_attr_getdetachstate (attr,detachstate)
```

„Łączenie” to jeden ze sposobów synchronizacji wątków. Na przykład:



- Podprogram `pthread_join()` blokuje wywołujący wątek do momentu zakończenia określonego wątku `threadid`.
- Programista może uzyskać status powrotu zakończenia wątku docelowego, jeśli został określony w wywołaniu docelowego wątku na `pthread_exit()`.
- Łączący wątek może pasować do jednego wywołania `pthread_join()`.
  Logicznym błędem jest próba wielu połączeń w tym samym wątku.

Dołączanie wątków

- Gdy tworzony jest wątek, jeden z jego atrybutów określa, czy można go łączyć, czy odłączać. Można łączyć tylko wątki utworzone jako łączone. Jeśli wątek jest tworzony jako odłączony, nigdy nie można go połączyć.
- Projekt standardu POSIX określa, że wątki powinny być tworzone jako łączone.

- Aby jawnie utworzyć wątek jako łączony lub odłączany, używany jest argument attr w procedurze `pthread_create()`. Typowy 4-etapowy proces to:
  - Zadeklaruj zmienną atrybutu pthread typu danych `pthread_attr_t`
  - Zainicjuj zmienną atrybutu `pthread_attr_init()`
  - Ustaw status `detached` za pomocą `pthread_attr_setdetachstate()`
  - Kiedy skończysz, zwolnij zasoby biblioteki używane przez atrybut za pomocą `pthread_attr_destroy()`

Odłączanie
Procedura `pthread_detach()` może być użyta do jawnego odłączenia wątku, nawet jeśli został utworzony jako łączony. Nie ma odwrotnej procedury.

Zalecenia:
- Jeśli wątek wymaga dołączenia, rozważ jawne utworzenie go jako łączonego. Zapewnia to przenośność, ponieważ nie wszystkie implementacje mogą tworzyć wątki jako domyślnie dołączane.
- Jeśli z góry wiesz, że wątek nigdy nie będzie musiał łączyć się z innym wątkiem, rozważ utworzenie go w stanie odłączonym. Niektóre zasoby systemowe mogą zostać zwolnione.

Dołączanie wątków: przykład

```
Main: creating thread 0
Main: creating thread 1
Main: creating thread 2
Thread 0 starting...
Thread 1 starting...
Main: creating thread 3
Thread 2 starting...
Thread 3 starting...
Thread 2 done. Result = -3.153838e+06
Thread 0 done. Result = -3.153838e+06
Thread 3 done. Result = -3.153838e+06
Thread 1 done. Result = -3.153838e+06
Main: completed join with thread 0 having a status of 0
Main: completed join with thread 1 having a status of 1
Main: completed join with thread 2 having a status of 2
Main: completed join with thread 3 having a status of 3
Main: program completed. Exiting.
```

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 4

void *BusyWork(void *t)
{
   int i;
   long tid;
   double result=0.0;
   tid = (long)t;
   printf("Thread %ld starting...\n",tid);
   for (i=0; i<1000000; i++)
   {
      result = result + sin(i) * tan(i);
   }
   printf("Thread %ld done. Result = %e\n",tid, result);
   pthread_exit((void*) t);
}

int main (int argc, char *argv[])
{
   pthread_t thread[NUM_THREADS];
   pthread_attr_t attr;
   int rc;
   long t;
   void *status;

   int rc;
   long t;
   void *status;

   /* Initialize and set thread detached attribute */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   for(t=0; t<NUM_THREADS; t++) {
      printf("Main: creating thread %ld\n", t);
      rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
      if (rc) {
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
         }
      }

   /* Free attribute and wait for the other threads */
   pthread_attr_destroy(&attr);
   for(t=0; t<NUM_THREADS; t++) {
      rc = pthread_join(thread[t], &status);
      if (rc) {
         printf("ERROR; return code from pthread_join() is %d\n", rc);
         exit(-1);
         }
      printf("Main: completed join with thread
                %ld having a status of %ld\n",t,(long)status);
      }

printf("Main: program completed. Exiting.\n");
pthread_exit(NULL);
}
```

- Ten przykład pokazuje, jak „czekać” na ukończenie wątku za pomocą procedury łączenia Pthread.
- Ponieważ niektóre implementacje Pthreads nie mogą tworzyć wątków w stanie możliwym do połączenia, wątki w tym przykładzie są jawnie tworzone w stanie możliwym do połączenia, dzięki czemu można je połączyć później.

Zarządzanie rozmiarem stosu

Procedury (dla stosu)

```
pthread_attr_getstacksize (attr, stacksize)
pthread_attr_setstacksize (attr, stacksize)
pthread_attr_getstackaddr (attr, stackaddr)
pthread_attr_setstackaddr (attr, stackaddr)
```

Zapobieganie problemom ze stosem

- Standard POSIX nie narzuca rozmiaru stosu wątku (zależne od implementacji i różni się)
- Przekroczenie domyślnego limitu stosu skutkuje: zakończeniem programu i/lub uszkodzeniem danych.
- Bezpieczne i przenośne programy nie są zależne od domyślnego rozmiaru stosu, ale zamiast tego jawnie przydzielają wystarczającą wielkość stosu dla każdego wątku, używając procedury `pthread_attr_setstacksize`.
- Procedury `pthread_attr_getstackaddr` i `pthread_attr_setstackaddr` mogą być używane przez aplikacje w środowisku, w którym stos dla wątku musi być umieszczony w określonym miejscu pamięci.

Zarządzanie rozmiarem stosu - przykład

- Ten przykład demonstruje sposób zapytania i ustawiania rozmiaru stosu wątku.

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NTHREADS 4
#define N 1000
#define MEGEXTRA 1000000

pthread_attr_t attr;

void *dowork(void *threadid) {
    double A[N][N];
    int i,j;
    long tid;
    size_t mystacksize;

    tid = (long)threadid;
    pthread_attr_getstacksize (&attr, &mystacksize);
    printf("Thread %ld: stack size = %li bytes \n", tid, mystacksize);
    for (i=0; i<N; i++)
      for (j=0; j<N; j++)
        A[i][j] = ((i*j)/3.452) + (N-i);
    pthread_exit(NULL);
}
```

```c
int main(int argc, char *argv[]) {
    pthread_t threads[NTHREADS];
    size_t stacksize;
    int rc;
    long t;

    pthread_attr_init(&attr);
    pthread_attr_getstacksize (&attr, &stacksize);
    printf("Default stack size = %li\n", stacksize);
    stacksize = sizeof(double)*N*N+MEGEXTRA;
    printf("Amount of stack needed per thread = %li\n",stacksize);
    pthread_attr_setstacksize (&attr, stacksize);
    printf("Creating threads with stack size = %li bytes\n",stacksize);
    for(t=0; t<NTHREADS; t++){
       rc = pthread_create(&threads[t], &attr, dowork, (void *)t);
       if (rc){
          printf("ERROR; return code from pthread_create() is %d\n", rc);
          exit(-1);
       }
    }
    printf("Created %ld threads.\n", t);
    pthread_exit(NULL);
}
```

Mutex (mutual exclusion)

- Mutex to skrót od „wzajemne wykluczenie”. Zmienne *muteksowe* są jednym z podstawowych sposobów implementacji synchronizacji wątków i ochrony współdzielonych danych, gdy może wystąpić wiele zapisów do nich.
- Zmienna mutex działa jak „blokada” chroniąca dostęp do współdzielonego zasobu danych. Podstawową koncepcją muteksu stosowaną w Pthreads jest to, że tylko jeden wątek może zablokować (lub posiadać) zmienną mutex w danym momencie. Zatem nawet jeśli kilka wątków próbuje zablokować muteks, tylko jeden wątek zdoła założyć blokadę. Żaden inny wątek nie może posiadać tego muteksu, dopóki wątek będący właścicielem nie odblokuje go. Wątki muszą „na zmianę” (taking turns) uzyskiwać dostęp do chronionych danych.
- Muteksy mogą być używane do zapobiegania sytuacjom „wyścigu” (race).

- Przykład sytuacji wyścigu związanego z transakcją bankową. Mutex powinien zostać użyty do zablokowania „stanu konta” (balance), podczas gdy wątek używa tego współdzielonego zasobu danych.

Thread 1	Thread 2	Balance
Read balance: $1000		$1000
	Read balance: $1000	$1000
	Deposit $200	$1000
Deposit $200		$1000
Update balance $1000+$200		$1200
	Update balance $1000+$200	$1200

Mutex - charakterystyka

- Często akcja wykonywana przez wątek posiadający mutex polega na aktualizacji zmiennych globalnych. Jest to bezpieczny sposób na zagwarantowanie, że gdy kilka wątków aktualizuje tę samą zmienną, końcowa wartość jest taka sama, jak gdyby był tylko jeden wątek przeprowadzający aktualizację. Aktualizowane zmienne należą do „sekcji krytycznej” (critical section).
- Typowa sekwencja użycia muteksu jest następująca:
  - Utwórz i zainicjuj zmienną mutex
  - Kilka wątków próbuje zablokować muteks
  - Tylko jednemu się powiedzie i ten wątek jest właścicielem muteksu
  - Wątek właściciela wykonuje pewien zestaw działań
  - Właściciel odblokowuje muteks
  - Inny wątek nabywa muteks i powtarza proces
  - W końcu mutex zostaje zniszczony
- Gdy kilka wątków współzawodniczy o muteks, przegrane wątki są zablokowane na tym wywołaniu – dlatego istnieje też wywołanie nieblokujące: `trylock` zamiast `lock`.
- Podczas ochrony udostępnianych danych obowiązkiem programisty jest upewnienie się, że robi to każdy wątek, który musi użyć muteksu. Na przykład, jeśli 4 wątki aktualizują te same dane, ale tylko jeden używa muteksu, dane mogą być nadal uszkodzone.

Tworzenie i niszczenie mutexów

Procedury

```
pthread_mutex_init (mutex,attr)

pthread_mutex_destroy (mutex)

pthread_mutexattr_init (attr)

pthread_mutexattr_destroy (attr)
```

- Zmienne Mutex muszą być zadeklarowane przy użyciu typu `pthread_mutex_t` i muszą być zainicjowane, zanim będą użyte. Istnieją dwa sposoby zainicjowania zmiennej mutex:
  - Statycznie, gdy zostanie zadeklarowane. Na przykład:
    `pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;`
  - Dynamicznie dzięki procedurze `pthread_mutex_init()`.
    Ta metoda pozwala na ustawienie atrybutów obiektu mutex, `attr`.
- Muteks jest początkowo **odblokowany**.

Tworzenie i niszczenie mutexów

- Obiekt `attr` służy do ustanawiania właściwości obiektu mutex i musi być typu `pthread_mutexattr_t`, jeśli jest używany (może być określony jako NULL, aby zaakceptować wartości domyślne).

  Standard Pthreads definiuje trzy opcjonalne atrybuty mutex:
  - Protocol: Określa protokół używany do zapobiegania inwersji priorytetów dla muteksu.
  - Prioceiling: Określa pułap priorytetu muteksu.
  - Process-shared: Określa proces współużywający muteksu.

  Należy zauważyć, że nie wszystkie implementacje mogą udostępniać trzy opcjonalne atrybuty mutex.

- Funkcje `pthread_mutexattr_init()` i `pthread_mutexattr_destroy()` są używane do tworzenia i niszczenia obiektów atrybutów mutex.
- `pthread_mutex_destroy()` powinien być używany do zwolnienia obiektu mutex, który nie jest już potrzebny.

Blokowanie i odblokowanie mutexów

Procedury

```
pthread_mutex_lock (mutex)

pthread_mutex_trylock (mutex)

pthread_mutex_unlock (mutex)
```

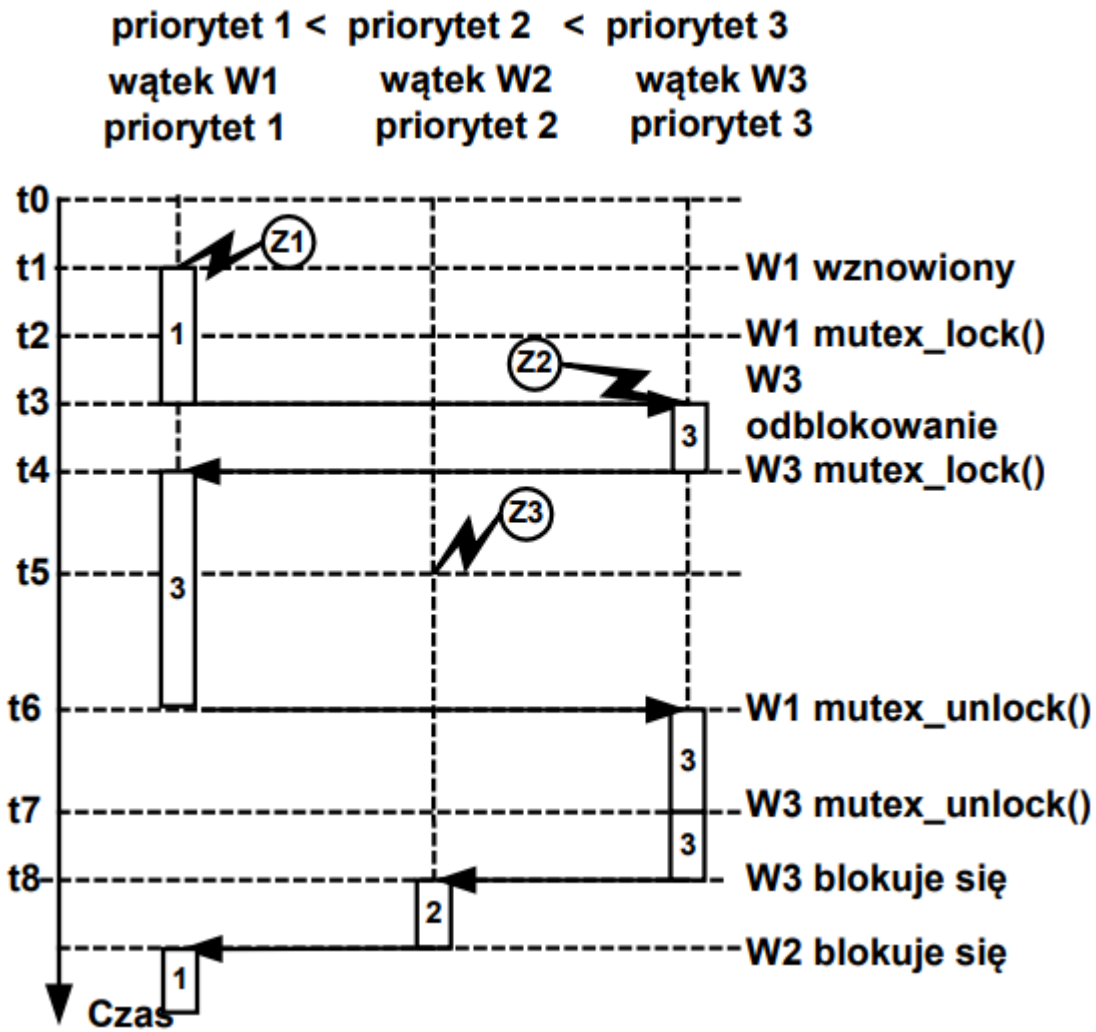
- Procedura `pthread_mutex_lock()` jest używana przez wątek do uzyskania blokady określonej zmiennej mutex. Jeśli mutex jest już zablokowany przez inny wątek, wywołanie to zablokuje wątek wywołujący, dopóki mutex nie zostanie odblokowany.
- `pthread_mutex_trylock()` podejmie próbę zablokowania muteksu. Jeśli jednak mutex jest już zablokowany, procedura natychmiast powróci z kodem błędu „zajęty”. Ta procedura może być przydatna w zapobieganiu zakleszczeniom, jak i w sytuacji inwersji priorytetów.
- `pthread_mutex_unlock()` odblokuje mutex, jeśli zostanie wywołany przez wątek będący właścicielem. Wywołanie tej procedury jest wymagane po tym, jak wątek zakończył korzystanie z chronionych danych, jeśli inne wątki mają uzyskać muteks do pracy z chronionymi danymi. Błąd zostanie zwrócony, jeśli:
  - jeśli mutex był już odblokowany
  - jeśli mutex jest własnością innego wątku

Inwersja priorytetów

Inwersja priorytetów (*ang. Priority Inversion*)

Inwersja priorytetów – zjawisko polegające na wykonywaniu się wątku o niższym priorytecie mimo że wątek o wyższym priorytecie pozostaje gotowy. Inwersja priorytetów może się pojawić gdy wątki o różnych priorytetach używają wspólnego muteksu lub podobnego mechanizmu synchronizacyjnego.

- Gdy dwa lub więcej wątki o różnych priorytetach używają wspólnego zasobu chronionego przez pewien mechanizm zapewnienia wzajemnego wykluczania (np. muteks) może dojść zjawiska nazywanego inwersją priorytetów.
- Przykład: Dwa wątki, W3 o priorytecie wyższym i W1 o priorytecie niższym używają zabezpieczonego muteksem wspólnego zasobu. Scenariusz: 1. Jako pierwszy startuje wątek W1 (o niższym priorytecie) a następnie wątek W1 zajmuje muteks. 2. Startuje wątek W3. Muteks jest już zajęty, wątek W3 mimo że posiada wyższy niż W1 priorytet nie będzie wykonywany. 3. Pojawia się wątek W2 o priorytecie pośrednim wywłaszczy on wątek W1 który nie będzie mógł zwolnić muteksu. Wątek W1 pozostanie wciąż zablokowany, a wykonywał się będzie wątek W2 o priorytecie niższym niż W3.



Dziedziczenie priorytetu

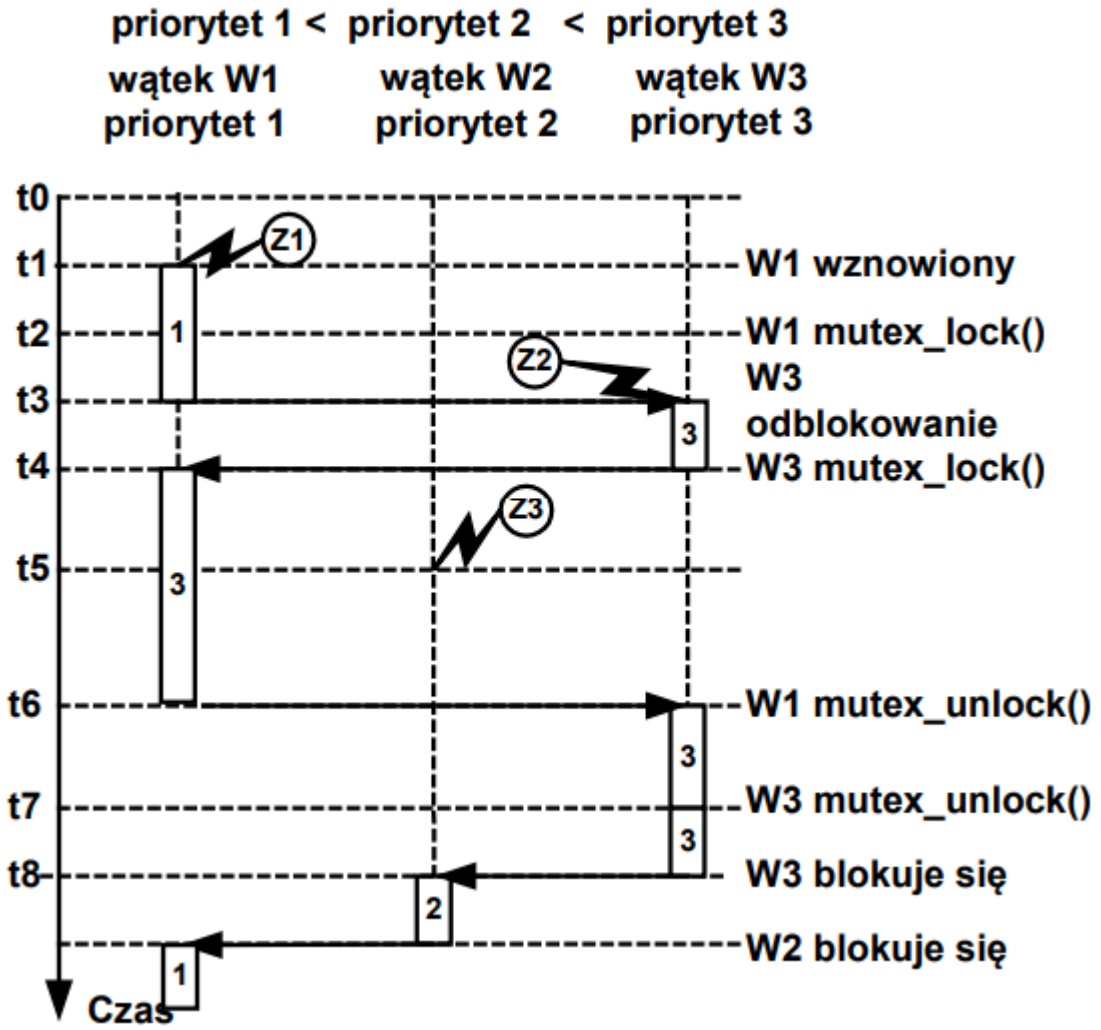
Dziedziczenie priorytetu (*ang. priority inheritance*)

Dziedziczeniem priorytetu – tymczasowe zwiększenie priorytetu wątku posiadającego zasób do najwyższego priorytetu z priorytetów wątków ubiegających się o zajęcie tego zasobu. Po zwolnieniu zasobu wątkowi przywracany jest początkowy priorytet.

- Dziedziczeniem priorytetu polega na tym, że gdy wątek W3 o wyższym priorytecie próbuje zająć muteks zajęty już przez wątek W1 o priorytecie niższym, to system podwyższa chwilowo priorytet wątku W1 zajmującego muteks do wysokości priorytetu wątku W3. Dzięki podwyższonemu priorytetowi wątek W1 szybciej wykona swe zadanie i zwolni muteks. Po zwolnieniu muteksu wątkowi W1 zostaje przywrócony pierwotny priorytet.
- Protokół dziedziczenia priorytetów działa prawidłowo w przypadku użycia jednego typu zasobu. Gdy używana jest większa liczba zasobów może dojść do różnych niekorzystnych zjawisk jak:
  - **blokowanie przechodnie**
  - **zakleszczenie**

Dziedziczenie priorytetu

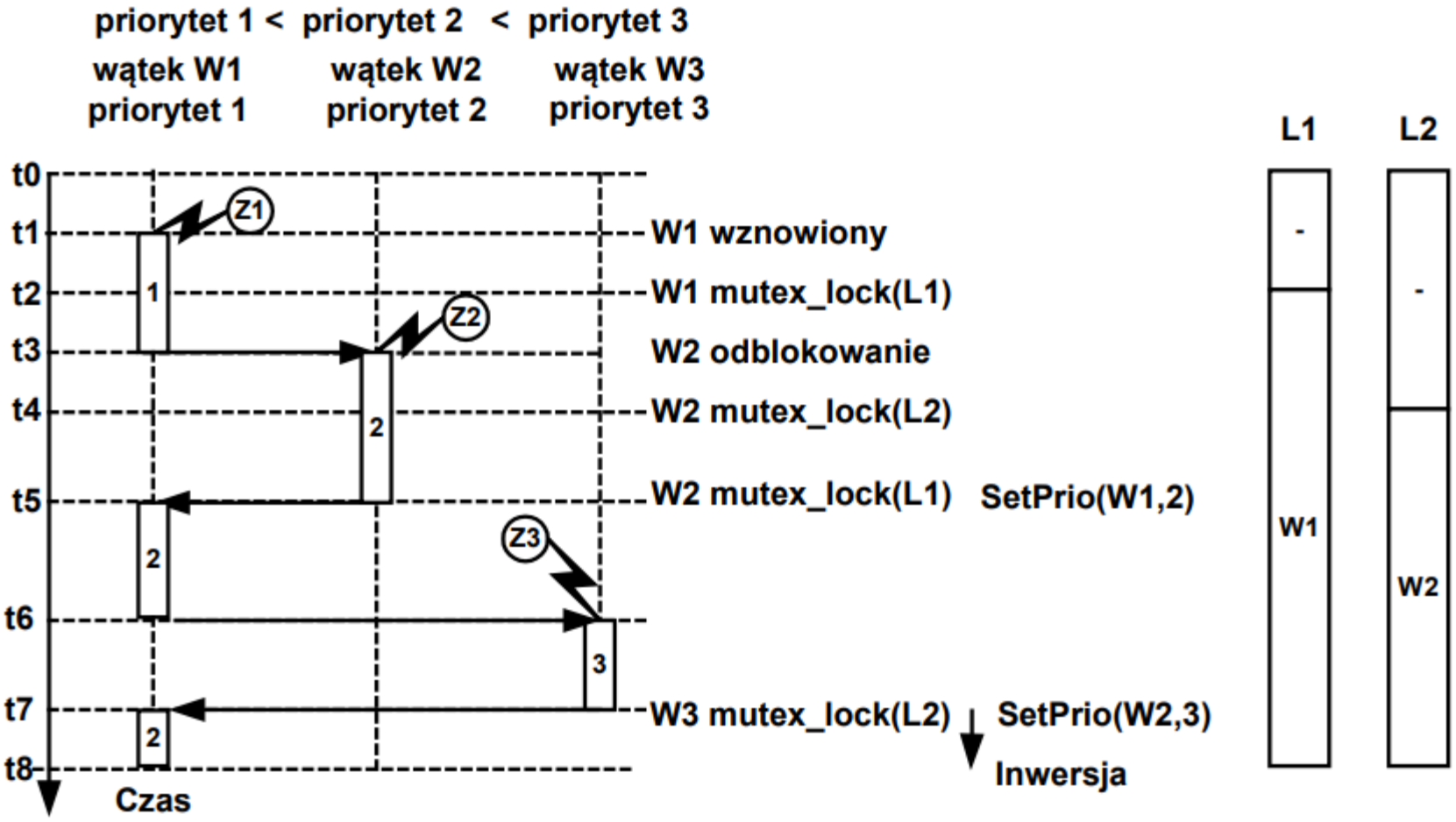
t1 Zdarzenie Z1 odblokowuje wątek W1
t2 Wątek W1 zajmuje muteks
t3 Zdarzenie Z2 odblokowuje wątek W3. Ma on priorytet wyższy od W1 i go wywłaszcza.
t4 Wątek W3 próbuje zająć muteks. Muteks jest zajęty a więc W3 blokuje się ale wątek W1 dziedziczy priorytet wątku W3 (z 1 wzrósł do 3).
t5 Zdarzenie Z3 odblokowuje wątek W2. Nie powoduje to wywłaszczenia W1 gdyż W2 ma niższy priorytet.
t6 Wątek W1 zwalnia muteks. Dopiero teraz wątek W3 staje się gotowy i wywłaszcza W1.
t7 Wątek W3 zwalnia muteks
t8 Wątek W3 blokuje się i dopiero teraz W2 może być wykonywany.



Blokowanie przechodnie

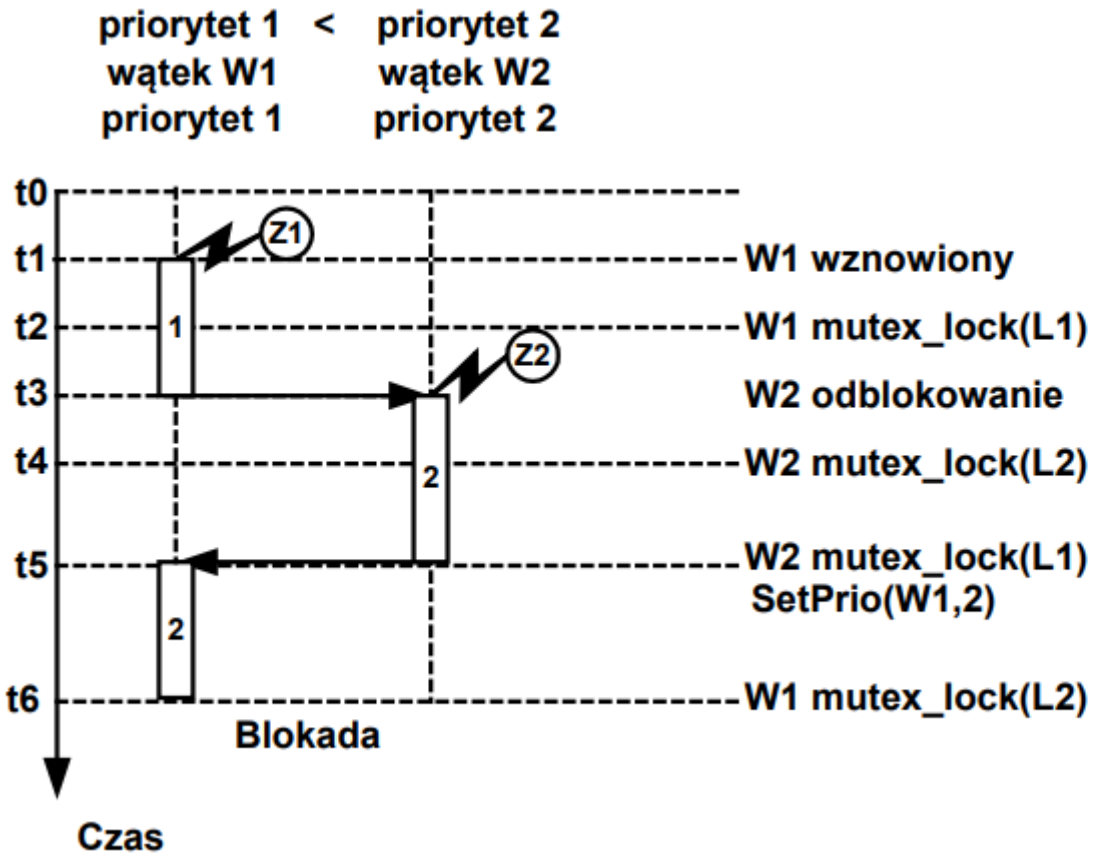
(ang. transitive blocking)

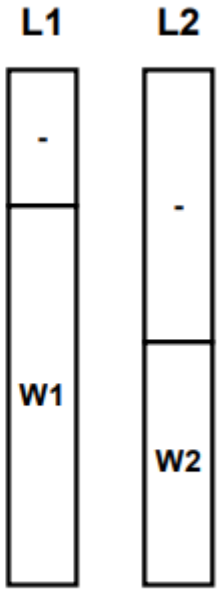
- W3 jest pośrednio blokowany przez W1 ale priorytet W1 nie jest podnoszony do priorytetu W3. Wady tej pozbawiony jest protokół wykorzystujący pułap priorytetów.



Zakleszczenie

Protokół dziedziczenia priorytetów posiada istotny defekt. Gdy wątki potrzebują dwóch różnych zasobów może dojść do ich zakleszczenia. Może się tak zdarzyć gdy wątek W1 zablokuje zasób potrzebny wątkowi W2 a wątek W2 zajmie zasób potrzebny wątkowi W1.





Protokół z pułapem priorytetów

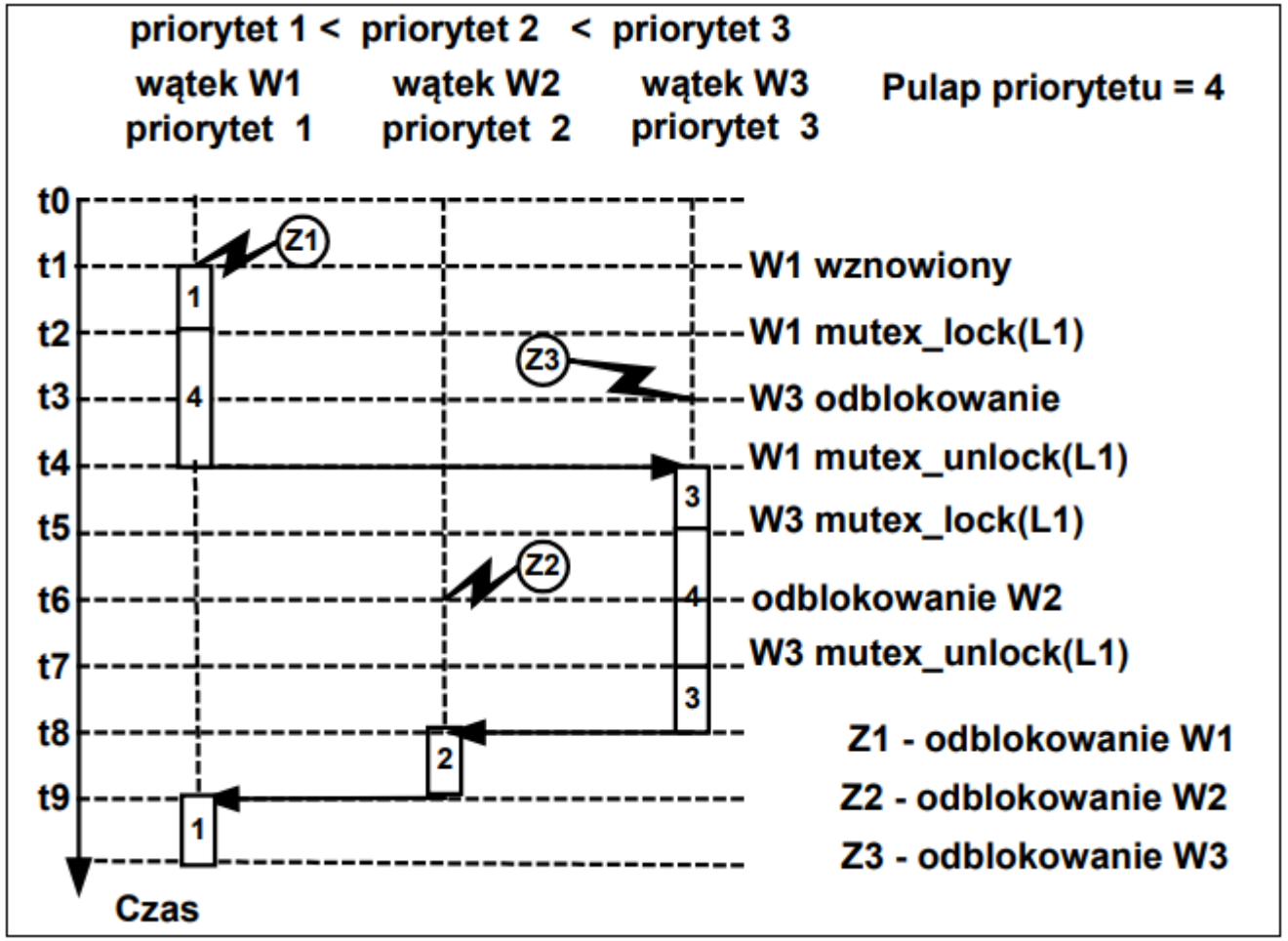
Protokół z pułapem priorytetu (*ang. priority ceiling protocol*)

W protokole z pułapem priorytetu następuje tymczasowe zwiększenie priorytetu wątku usiłującego zająć zasób do pewnego ustalonego priorytetu (wyższego od priorytetu jakiegokolwiek wątku konkurującego o zasób). Wszystkim wątkom konkurujące o zasób zostaje tymczasowo nadany ten jednakowy priorytet.

- Drugą strategią zapobiegania inwersji priorytetów jest przyjęcie protokołu wykorzystującego pułap priorytetów. Każdemu chronionemu zasobowi (w tym przypadku jest to muteks) przypisuje się pewien określony statyczny priorytet. Priorytet ten powinien być wyższy od najwyższego priorytetu z tych wątków które o dany zasób będą konkurowały. Gdy jakiś wątek będzie próbował zająć zasób to zostanie mu tymczasowo przydzielony priorytet związany z tym zasobem. Po zwolnieniu zasobu priorytet wątku wróci do wielkości wyjściowej.
- Dzięki temu że wątek zajmujący zasób zyskuje chwilowo priorytet wyższy niż jakiekolwiek inny wątek konkurujący o zasób – ma on szansę zakończyć operację na zasobie bez wywłaszczenia.

Protokół z pułapem priorytetów

- Zalety:
  - Protokół zapobiega powstawaniu zakleszczeń.
  - Zapewnia dobry czas oczekiwania na zasób (dla najgorszego przypadku) poprzez wątek o najwyższym priorytecie. Czas ten równy jest długości najdłuższej sekcji krytycznej wątków o niższym priorytecie.
- Wady:
  - Należy z góry wyznaczyć zbiór wszystkich wątków, które będą konkurowały o zasób i jako pułap priorytetu przyjąć najwyższy priorytet z zadań z tego zbioru + 1 . Może to być czasochłonne lub nawet niemożliwe.
  - Posiada zły średni czas odpowiedzi z związku z narzutami na implementację.



Obliczenia sekwencyjne (iloczyn skalarny)

Przykład programu liczącego iloczyn skalarny dwóch wektorów. Obliczenia wykonane sekwencyjnie.

```
Sum =  100000.000000

real    0m9.419s
user    0m0.004s
sys     0m0.000s
```

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct
{
  double      *a;
  double      *b;
  double     sum;
  int    veclen;
} DOTDATA;

#define VECLEN 100000
  DOTDATA dotstr;

void dotprod() {

   int start, end, i;
   double mysum, *x, *y;

   start=0;
   end = dotstr.veclen;
   x = dotstr.a;
   y = dotstr.b;

   mysum = 0;
   for (i=start; i<end ; i++) {
      mysum += (x[i] * y[i]);
   }
   dotstr.sum = mysum;
}
```

```c
int main (int argc, char *argv[]) {
    int i,len;
    double *a, *b;

    len = VECLEN;
    a = (double*) malloc (len*sizeof(double));
    b = (double*) malloc (len*sizeof(double));

    for (i=0; i<len; i++) {
        a[i]=1;
        b[i]=a[i];
    }

    dotstr.veclen = len;
    dotstr.a = a;
    dotstr.b = b;
    dotstr.sum=0;

    dotprod();

    printf("Sum =  %f \n", dotstr.sum);
    free(a);
    free(b);
}
```

Obliczenia równoległe (iloczyn skalarny)

Przykład programu liczącego iloczyn skalarny dwóch wektorów. Obliczenia wykonane równolegle.

- Główne dane są udostępniane wszystkim wątkom za pośrednictwem globalnie dostępnej struktury.
- Każdy wątek działa na innej części danych.
- Główny wątek czeka na zakończenie wszystkich obliczeń przez wszystkie wątki, a następnie drukuje wynikową sumę.

```
Thread 1 did 100000 to 200000:  mysum=100000.000000 global sum=100000.000000
Thread 3 did 300000 to 400000:  mysum=100000.000000 global sum=200000.000000
Thread 0 did 0 to 100000:  mysum=100000.000000 global sum=300000.000000
Thread 2 did 200000 to 300000:  mysum=100000.000000 global sum=400000.000000
Sum =  400000.000000

real    1m17.055s
user    0m0.008s
sys     0m0.008s
```

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

typedef struct  {
   double      *a;
   double      *b;
   double      sum;
   int     veclen;
} DOTDATA;

#define NUMTHRDS 4
#define VECLEN 100000
   DOTDATA dotstr;
   pthread_t callThd[NUMTHRDS];
   pthread_mutex_t mutexsum;

void *dotprod(void *arg) {
   int i, start, end, len ;
   long offset;
   double mysum, *x, *y;
   offset = (long)arg;

   len = dotstr.veclen;
   start = offset*len;
   end   = start + len;
   x = dotstr.a;
   y = dotstr.b;

   mysum = 0;
   for (i=start; i<end; i++) {
      mysum += (x[i] * y[i]);
   }

   pthread_mutex_lock (&mutexsum);
   dotstr.sum += mysum;
   printf("Thread %ld did %d to %d:  mysum=%f
   global sum=%f\n",offset,start,end,mysum,dotstr.sum);
   pthread_mutex_unlock (&mutexsum);

   pthread_exit((void*) 0);
}
```

```c
int main (int argc, char *argv[]) {
    long i;
    double *a, *b;
    void *status;
    pthread_attr_t attr;

    a = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
    b = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));

    for (i=0; i<VECLEN*NUMTHRDS; i++) {
        a[i]=1;
        b[i]=a[i];
    }

    dotstr.veclen = VECLEN;
    dotstr.a = a;
    dotstr.b = b;
    dotstr.sum=0;

    pthread_mutex_init(&mutexsum, NULL);

    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    for(i=0;i<NUMTHRDS;i++) {
        pthread_create(&callThd[i], &attr, dotprod, (void *)i);
    }

    pthread_attr_destroy(&attr);

    for(i=0;i<NUMTHRDS;i++) {
        pthread_join(callThd[i], &status);
    }

    printf ("Sum =  %f \n", dotstr.sum);
    free(a);
    free(b);
    pthread_mutex_destroy(&mutexsum);
    pthread_exit(NULL);
}
```

Zmienne warunkowe

- Zmienne warunkowe (condition variables) zapewniają jeszcze inny sposób synchronizowania wątków. Podczas gdy muteksy realizują synchronizację poprzez kontrolowanie dostępu wątku do danych, zmienne warunkowe umożliwiają synchronizację wątków w oparciu o rzeczywistą wartość danych.
- Bez zmiennych warunkowych programista musiałby ciągle odpytywać wątki (np. w krytycznej sekcji), aby sprawdzić, czy warunek jest spełniony. To bardzo obciążałoby wątek. Zmienna warunkowa jest sposobem na osiągnięcie tego samego celu bez odpytywania.
- Zmienna warunkowa jest zawsze używana w połączeniu z blokadą mutex.

Zmienne warunkowe - ilustracja

- Reprezentatywna sekwencja użycia zmiennych warunkowych jest przedstawiona poniżej.

Główny wątek
Zadeklaruj i zainicjuj dane globalne / zmienne wymagające synchronizacji (np. „licznik”)
Zadeklaruj i zainicjuj obiekt zmiennej warunkowej
Zadeklaruj i zainicjuj powiązany muteks
Utwórz wątki A i B, aby wykonać pracę

Wątek A	**Wątek B**
Wykonuj czynności do momentu, w którym musi wystąpić pewien warunek (np. „licznik” musi osiągnąć określoną wartość) Zablokuj skojarzony muteks i sprawdź wartość zmiennej globalnej Wywołaj `pthread_cond_wait()`, aby wykonać blokowanie, poczekaj na sygnał z Wątek-B. Zauważ, że wywołanie `pthread_cond_wait()` automatycznie i atomowo odblokowuje powiązaną zmienną mutex, aby mogła być używana przez Wątek-B. Po zasygnalizowaniu obudź się. Mutex jest automatycznie i atomowo zablokowany. Jawnie odblokuj mutex. Kontynuuj.	Wykonaj pracę Zablokuj skojarzony muteks Zmień wartość zmiennej globalnej, na którą czeka Wątek-A. Sprawdź wartość tej zmiennej globalnej. Jeśli spełniony jest żądany warunek, zasygnalizuj to do Wątku-A. Odblokuj mutex. Kontynuuj.

Główny wątek
Dołącz (join) / Kontynuuj

Tworzenie i niszczenie zmiennych warunkowych

Procedury

```
pthread_cond_init (condition,attr)

pthread_cond_destroy (condition)

pthread_condattr_init (attr)

pthread_condattr_destroy (attr)
```

- Zmienne warunkowe muszą być zadeklarowane przy użyciu typu `pthread_cond_t` i muszą być zainicjowane przed ich użyciem. Dwa sposoby inicjalizacji:
  - statycznie, gdy zostanie zadeklarowane:
    `pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;`
  - dynamicznie, z procedurą `pthread_cond_init()`. Identyfikator utworzonej zmiennej warunkowej jest zwracany do wątku wywołującego poprzez parametr warunku.
- Opcjonalny obiekt `attr` służy do ustawiania atrybutów zmiennych warunkowych. Dla zmiennych warunkowych zdefiniowany jest tylko jeden atrybut: współdzielony proces (proces-shared), który pozwala, aby zmienna warunkowa była widoczna przez wątki w innych procesach. Obiekt atrybutu musi być typu `pthread_condattr_t` (może być określony jako NULL, aby zaakceptować wartości domyślne).
- Procedury `pthread_condattr_init()` i `pthread_condattr_destroy()` są używane do tworzenia i niszczenia obiektów atrybutów zmiennych warunkowych.
- `pthread_cond_destroy()` należy użyć do zwolnienia zbędnej zmiennej warunkowej.

Zmienne warunkowe – oczekiwanie i sygnalizowanie

Procedury

```
pthread_cond_wait (condition,mutex)

pthread_cond_signal (condition)

pthread_cond_broadcast (condition)
```

- `pthread_cond_wait()` blokuje wywołujący wątek, dopóki nie zostanie zasygnalizowany określony warunek. Ta procedura powinna zostać wywołana, gdy mutex jest zablokowany, i automatycznie zwolni muteks podczas oczekiwania. Po odebraniu sygnału i przebudzeniu wątku mutex zostanie automatycznie zablokowany do użycia przez wątek. Programista jest odpowiedzialny za odblokowanie muteksu po zakończeniu wątku.
- Zalecenie: Użycie pętli WHILE zamiast instrukcji IF w celu sprawdzenia stanu oczekiwania może pomóc w rozwiązaniu kilku potencjalnych problemów, takich jak:
  - Jeśli kilka wątków czeka na ten sam sygnał budzenia, będą na zmianę zdobywać muteks i każdy z nich może następnie zmodyfikować warunek, na który wszyscy czekali.
  - Jeśli wątek otrzymał sygnał w wyniku błędu programu
  - Biblioteka Pthreads może wydawać fałszywe przebudzenia oczekującego wątku bez naruszania standardu.

Zmienne warunkowe - uwagi

- Funkcja `pthread_cond_signal()` służy do sygnalizowania (lub budzenia) innego wątku, który czeka na zmienną warunkową. Powinna zostać wywołana po zablokowaniu muteksu i musi odblokować mutex, aby procedura `pthread_cond_wait()` została zakończona.
- Procedura `pthread_cond_broadcast()` powinna być używana zamiast `pthread_cond_signal()`, jeśli więcej niż jeden wątek jest w stanie oczekiwania na blokowanie.
- Logicznym błędem jest wywołanie `pthread_cond_signal()` przed wywołaniem `pthread_cond_wait()`.
- Właściwe blokowanie i odblokowywanie powiązanej zmiennej mutex jest niezbędne podczas korzystania z tych procedur:
  - Nieudane zablokowanie muteksu przed wywołaniem `pthread_cond_wait()` może spowodować, że NIE zostanie on zablokowany.
  - Nieudane odblokowanie muteksu po wywołaniu `pthread_cond_signal()` może uniemożliwić wykonanie procedury `pthread_cond_wait()` (pozostanie zablokowany).