Wątki

dr inż. Sławomir Samolej
Katedra Informatyki i Automatyki
Politechnika Rzeszowska

Program przedmiotu oparto w części na materiałach opublikowanych na:
http://wazniak.mimuw.edu.pl/

oraz
na materiałach opracowanych przez
dr inż. Jędrzeja Ułasiewicza:
jedrzej.ulasiewicz.staff.iiar.pwr.wroc.pl

Potrzeba wprowadzania wątków

- W pewnych przypadkach tworzenie programów złożonych z wielu „ciężkich” procesów uważane jest za mało wydajne.
- Oczekuje się możliwości tworzenia aplikacji współbieżnych, które w bardziej naturalny sposób współdzielą pamięć i zasoby.
- Niektóre systemy operacyjne nie dysponują tak szerokim zestawem mechanizmów programowania współbieżnego opartych na współpracy procesów jak Unix i udostępniają mechanizmy programowania współbieżnego tylko na bazie wątków.

Definicja wątku

- **Sekwencja działań, która może wykonywać się równolegle z innymi sekwencjami działań w kontekście danego procesu (programu).**

Tworzenie wątku a instrukcja fork

- Po uruchomieniu funkcji fork tworzony jest osobny proces z własnym zestawem zmiennych (odziedziczonych po rodzicu) i własnym identyfikatorem proces (PID).
- Po utworzeniu nowego wątku w procesie, otrzymuje on nowy stos (zmienne lokalne) ale współdzieli z innymi zmienne globalne, deskryptory plików, funkcje przechwytujące sygnały i stan bieżącego katalogu i procesu.

Zalety stosowania wątków

- Mniejszy nakład obliczeniowy ze strony systemu operacyjnego
- Możliwość tworzenia aplikacji, które mają wiele wątków pracujących na wspólnych danych lub zasobach (serwer bazy danych, serwer WWW, przetwarzanie w tle tekstu, zapis stanu jednostek w grach strategicznych itp.)
- Możliwość wydzielenia w aplikacji osobnych wątków pobierających dane, wysyłających dane i przetwarzających dane. Przetwarzanie może się odbywać, gdy operacje odczytu/zapisu są zablokowane.
- Przełączanie pomiędzy wątkami wymaga znacznie mniejszego nakładu obliczeniowego od SO

Wady stosowania wątków

- Tworzenie programów wielowątkowych wymaga precyzyjnego projektowania, ponieważ łatwo nieprzewidywalne wyniki spowodowane dostępem do zmiennych w czasie i przestrzeni
- Utrudnione jest śledzenie takich programów
- Aplikacja wielowątkowa na jednoprocesorowym komputerze **nie** musi działać szybciej niż jej jednowątkowa wersja.

Uwagi do tworzenia aplikacji wielowątkowych

- Należy włączyć bibliotekę pthread do opcji konsolidacji
- Należy włączyć makro: _REENTRANT (fragmenty kodu typu re-entrant mogą być równocześnie wywoływane przez wiele wątków w programie i nie zaszkodzi to ich spójności)

Tworzenie wątku

```
#include <pthread.h>
int pthread_create(pthread_t *thread, pthread_attr_t *attr, void
*(*start_routine)(void *), void *arg);
```

Atrybuty:
thread – identyfikator wątku
attr – atrybuty wątku (można podać NULL)
start_routine – wskaźnik na funkcję, od której zaczyna pracę wątek
Wartość zwracana:
0 – sukces, inna liczba – kod będu

Kończenie pracy wątku

```
#include <pthread.h>
void pthread_exit(void *retval);
```

Atrybuty:
retval – identyfikator wątku (można się jeszcze do niego odwołać, choć wątek zakończył działanie)
Uwaga: Jako parametr retval nie należy podawać zmiennych lokalnych.

Oczekiwanie na zakończenie wątku

```
#include <pthread.h>
int pthread_join(pthread_t th, void **thread_return);
```

Atrybuty:
th – identyfikator wątku, na który oczekujemy
thread_return – wskaźnik na wskaźnik pokazujący na wartość zwracaną przez wątek
Funkcja zwraca:
0 – sukces, lub kod błędu.

Pierwszy program wielowątkowy

```
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>

void *thread_function(void *arg);

char message[] = "Hello World";

int main() {
   int res;
   pthread_t a_thread;
   void *thread_result;
   res = pthread_create(&a_thread, NULL,
thread_function, (void *)message);
   if (res != 0) {
      perror("Thread creation failed");
      exit(EXIT_FAILURE);
   }
   printf("Waiting for thread to finish...\n");
   res = pthread_join(a_thread, &thread_result);
```

```
if (res != 0) {
      perror("Thread join failed");
      exit(EXIT_FAILURE);
   }
   printf("Thread joined, it returned %s\n", (char *)
                                thread_result);
   printf("Message is now %s\n", message);
   exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
   printf("thread_function is running. Argument
was %s\n", (char *)arg);
   sleep(3);
   strcpy(message, "Bye!");
   pthread_exit("Thank you for the CPU time");
}
```

```
$ ./thread1
Waiting for thread to finish...
thread_function is running. Argument was Hello World
Thread joined, it returned Thank you for the CPU time
Message is now Bye!
```

Jak to działa?

- thread_function jest funkcją, która rozpocznie działanie w nowym wątku
- W funkcji powoływany jest nowy wątek, obsługiwany przez thead_function, wątek przyjmuje domyślne parametry (NULL), a do funkcji wykonującej jako parametr zostaje wysłana tablica message
- Program główny kontynuuje swoje obliczenia i oczekuje na zakończenie utworzonego wątku (pthread_join)
- Wątek potwierdza odebranie danych przez parametr wywołania funkcji, modyfikuje stan współdzielonej zmiennej i kończy swoje działanie (pthread_exit).
- Następuje zakończenie programu macierzystego.

Sprawdzenie, czy wątki się przełączają?

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

void *thread_function(void *arg);
int run_now = 1;

int main() {
   int res;    pthread_t a_thread;
   void *thread_result;
   int print_count1 = 0;
   res = pthread_create(&a_thread, NULL,
thread_function, (void *)message);
   if (res != 0) {
      perror("Thread creation failed");
      exit(EXIT_FAILURE);
   }

   while(print_count1++ < 20000000) {
      if (run_now == 1) {
         printf("1");
         run_now = 2;    } }
```

```
   printf("\nWaiting for thread to finish...\n");
   res = pthread_join(a_thread, &thread_result);
   if (res != 0) {
      perror("Thread join failed");
      exit(EXIT_FAILURE);
   }
   printf("Thread joined\n");
   exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
   int print_count2 = 0;

   while(print_count2++ < 20000000) {
      if (run_now == 2) {
         printf("2");
         run_now = 1;     }}
   sleep(3);
}
```

```
$ ./thread2
12121212121212121212
Waiting for thread to finish...
Thread joined
```

Problem aktywnego czekania…

- Każdy z wątków aplikacji sprawdza stan zmiennej run_now.
- Jeśli przyjmuje ona zadaną wartość, program wypisuje tę wartość i zmienia na drugą z możliwych
- Oba wątki wykonują niekorzystne z punktu wydajności systemu **aktywne czekanie** i próbkowanie stanu zmiennej
- Większość mocy obliczeniowej procesora zużywana jest na sprawdzanie stanu zmiennej w każdym z wątków.

Synchronizacja z zastosowaniem semaforów

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

void *thread_function(void *arg);
sem_t bin_sem;

#define WORK_SIZE 1024
char work_area[WORK_SIZE];

int main() {
   int res;
   pthread_t a_thread;
   void *thread_result;

   res = sem_init(&bin_sem, 0, 0);
   if (res != 0) {
      perror("Semaphore initialization failed");
      exit(EXIT_FAILURE); }
   res = pthread_create(&a_thread, NULL,
thread_function, NULL);
   if (res != 0) {
      perror("Thread creation failed");
      exit(EXIT_FAILURE); }
```

```
printf("Input some text. Enter 'end' to finish\n");
   while(strncmp("end", work_area, 3) != 0) {
      fgets(work_area, WORK_SIZE, stdin);
      sem_post(&bin_sem);
   }
   printf("\nWaiting for thread to finish...\n");
   res = pthread_join(a_thread, &thread_result);
   if (res != 0) {
      perror("Thread join failed");
      exit(EXIT_FAILURE);
   }
   printf("Thread joined\n");
   sem_destroy(&bin_sem);
   exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
   sem_wait(&bin_sem);
   while(strncmp("end", work_area, 3) != 0) {
      printf("You input %d characters\n",
strlen(work_area) -1);
      sem_wait(&bin_sem);
   }
   pthread_exit(NULL);
}
```

Wyjście programu

```
$./thread3
Input some text. Enter ‘end’ to finish
The Wasp Factory
You input 16 characters
Iain Banks
You input 10 characters
end
Waiting for thread to finish...
Thread joined
```

Komentarz

- W programie rozwiązano problem synchronizacji z zastosowaniem semaforów, podobnie jak na poprzednim wykładzie (str. 13)
- Wartość początkowa semafora ustawiana jest na 0
- Podstawowy wątek cyklicznie odbiera od użytkownika komunikaty tekstowe i zapisuje do bufora, a następnie zwalnia semafor (sem_post).
- Wątek potomny najpierw oczekuje na zwolnienie semafora (sem_wait), a potem cyklicznie analizuje zawartość bufora z danymi (długość tekstu) i oczekuje na ustawienie semafora (sem_wait).

Zależności czasowe

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

void *thread_function(void *arg);
sem_t bin_sem;

#define WORK_SIZE 1024
char work_area[WORK_SIZE];

int main() {
   int res;    pthread_t a_thread;
   void *thread_result;
   res = sem_init(&bin_sem, 0, 0);
   if (res != 0) {
      perror("Semaphore initialization failed");
      exit(EXIT_FAILURE);    }
   res = pthread_create(&a_thread, NULL,
                  thread_function, NULL);
   if (res != 0) {   perror("Thread creation failed");
      exit(EXIT_FAILURE);    }
```

```
   printf("Input some text. Enter 'end' to finish\n");
   while(strncmp("end", work_area, 3) != 0) {
    if (strncmp(work_area, "FAST", 4) == 0) {
      sem_post(&bin_sem);
      strcpy(work_area, "Wheeee...");
    } else {
      fgets(work_area, WORK_SIZE, stdin);}
    sem_post(&bin_sem);
   }
   printf("\nWaiting for thread to finish...\n");
   res = pthread_join(a_thread, &thread_result);
   if (res != 0) {
      perror("Thread join failed");
      exit(EXIT_FAILURE); }
   printf("Thread joined\n");
   sem_destroy(&bin_sem);
   exit(EXIT_SUCCESS);
}
void *thread_function(void *arg) {
   sem_wait(&bin_sem);
   while(strncmp("end", work_area, 3) != 0) {
      printf("You input %d characters\n",
                                  strlen(work_area) -1);
      sem_wait(&bin_sem);    }
   pthread_exit(NULL);
}
```

Wyjście programu

```
$ ./thread4a
Input some text. Enter ‘end’ to finish
Excession
You input 9 characters
FAST
You input 7 characters
You input 7 characters
You input 7 characters
end
Waiting for thread to finish...
Thread joined
```

Komentarz

- Kiedy użytkownik wprowadza słowo FAST na wejście programu to następuje **zwolnienie semafora**,
- Warunek zakończenia pętli while nie jest spełniony
- Następuje rozpoznanie, że użytkownik wprowadził wyraz „FAST” , ponowne **zwolnienie semafora**, a następnie wprowadzenie do współdzielonej zmiennej tekstu „Wheeee..”
- Ponowny raz semafor **zostaje zwolniony**
- Warunek zakończenia pętli while nie jest spełniony
- Program oczekuje na wprowadzenie nowego tekstu na standardowym wejściu
- Zmiany zawartości współdzielonej zmiennej odbywają się tak szybko, że wątek obliczający długość tekstu „gubi” jedną ze zmian.
- Wątek macierzysty podczas wywoływania funkcji sem_post() zwiększa licznik.
- W konsekwencji wątek potomny, który wcześniej nie uzyskał dostępu do współdzielonej zmiennej trzykrotnie „odblokowuje się na semaforze” i wypisuje obliczoną długość tekstu.

Mutexy

- Wątki dzielą wspólny obszar danych. Stąd współbieżny dostęp do danych może naruszyć ich integralność.
- Należy zapewnić synchronizację dostępu do wspólnych danych.
- W bibliotece pthreads do zapewnienia wyłączności dostępu do danych stosuje się mechanizm muteksu (*ang. mutex*). Nazwa ta pochodzi od słów *Mutual exclusion* czyli wzajemne wykluczanie.

Funkcje do obsługi mutexów

```
#include <pthread.h>
//Inicjalizacja:
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);

// Zamknięcie dostępu do sekcji krytycznej:
int pthread_mutex_lock(pthread_mutex_t *mutex));

// Otwarcie dostępu do sekcji krytycznej:
int pthread_mutex_unlock(pthread_mutex_t *mutex);

// Zniszczenie mutex'a
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Uwagi:
Funkcje korzystają ze wcześniej zadeklarowanego obiektu typu:
pthread_mutex_t

Przykład – ochrona współdzielonych danych z zastosowaniem mutex'a

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

void *thread_function(void *arg);
pthread_mutex_t work_mutex;
#define WORK_SIZE 1024
char work_area[WORK_SIZE]; int time_to_exit = 0;

int main() {
   int res;    pthread_t a_thread;    void *thread_result;
   res = pthread_mutex_init(&work_mutex, NULL);
   if (res != 0) {       perror("Mutex initialization failed");
      exit(EXIT_FAILURE);}
   res = pthread_create(&a_thread, NULL,
                                thread_function, NULL);
   if (res != 0) { perror("Thread creation failed");
                                exit(EXIT_FAILURE);}
   pthread_mutex_lock(&work_mutex);
   printf("Input some text. Enter 'end' to finish\n");
   while(!time_to_exit) {
      fgets(work_area, WORK_SIZE, stdin);
      pthread_mutex_unlock(&work_mutex);
```

```
while(1) {
        pthread_mutex_lock(&work_mutex);
        if (work_area[0] != '\0') {
          pthread_mutex_unlock(&work_mutex);
          sleep(1);   } else {   break;    }    }    }
   pthread_mutex_unlock(&work_mutex);
   printf("\nWaiting for thread to finish...\n");
   res = pthread_join(a_thread, &thread_result);
   if (res != 0) {       perror("Thread join failed");
                                exit(EXIT_FAILURE); }
   printf("Thread joined\n");
   pthread_mutex_destroy(&work_mutex);
   exit(EXIT_SUCCESS);}

void *thread_function(void *arg) { sleep(1);
   pthread_mutex_lock(&work_mutex);
   while(strncmp("end", work_area, 3) != 0) {
    printf("You input %d characters\n", strlen(work_area) -
1);   work_area[0] = '\0';
      pthread_mutex_unlock(&work_mutex); sleep(1);
      pthread_mutex_lock(&work_mutex);
      while (work_area[0] == '\0' ) {
        pthread_mutex_unlock(&work_mutex); sleep(1);
        pthread_mutex_lock(&work_mutex); } }
   time_to_exit = 1;    work_area[0] = '\0';
   pthread_mutex_unlock(&work_mutex);
   pthread_exit(0);}
```

Wyjście programu

```
Input some text. Enter ‘end’ to finish
Whit
You input 4 characters
The Crow Road
You input 13 characters
end
Waiting for thread to finish...
Thread joined
```

Jak to działa? (1)

- W obszarze zmiennych globalnych zadeklarowano nowe zmienne work_mutex i time_to_exit
- W głównej funkcji programu zainicjalizowano mutex
- Powołano nowy wątek który:
  - próbuje zamknąć mutex (jeśli jest zamknięty, to oczekuje na jego otwarcie),
  - sprawdza, czy spełniony jest warunek zakończenia wątku, jeśli tak, ustawia zmienną time_to_exit na 0 i pierwszy element tablicy to odblokowuje work_area na 0, otwiera mutex
  - jeśli nie, to obliczana jest długość tekstu i ustawiana pierwszy element tablicy work_area na 0, następuje tez otwarcie mutex'a (ustawienie pierwszego elementu tablicy na 0 oznacza, że wątek zakończył swoje przetwarzanie współdzielonej zmiennej)
  - po odczekaniu 1 sekundy wątek próbuje zamknąć mutex
  - jeśli zamknięcie się powiedzie, to cyklicznie sprawdza, czy pierwszy element tablicy jest w dalszym ciągu 0, zwalnia mutex, odczekuje 1 sekundę, próbuje zamknąć mutex
  - jeśli zawartość tablicy ulegnie zmianie, to pętla sprawdzająca jest przerywana, mutex pozostaje zamknięty

Jak to działa? (2)

- W wątku macierzystym :
  - następuje próba zamknięcia mutex'a
  - kiedy zamknięcie się powiedzie w pętli następuje odczytywanie tekstów wprowadzanych przez użytkownika (tekst „end" kończy działanie programu) i odblokowanie mutexa
  - w wewnętrznej pętli następuje sprawdzenie, czy tekst nie został przetworzony, sprawdzenie następuje z zachowaniem wzajemnego wykluczania w dostępie do współdzielonej zmiennej (próba zamknięcia a potem otwarcie mutex'a)
  - po wykryciu odebrania danych przez wątek przetwarzający następuje ponownie próba przjecia dostępu do danych współdzielonych i wpisanie do nich nowego tekstu
  - po wpisaniu do tablicy współdzielonej tekstu „end" nstępuje zamknięcie wątku macierzystego i potomnego.
- Uwagi:
  - Oba wątki stosują swego rodzaju pooling do wykrywania zmiany stanu zmiennej dzielonej, w profesjonalnym rozwiązaniu należy zastosować semafor lub zmienną warunkową (nowość) do synchronizacji

Argumenty wątków (wybrane)

```
#include <pthread.h>
// Inicjalizacja możliwości ustlania atrybutów wątków
int pthread_attr_init(pthread_attr_t *attr);

// Ustalenie sposobu zakończenia wątku:
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);

// Sprawdzenie, w jaki sposób wątek będzie kończony
int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);

// Ustalenie sposobu szeregowania wątku:
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);

// Sprawdzenie w jaki sposób wątek będzie szeregowany:
int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int *policy);

//
//...
//
```

Kończenie wątku bez oczekiwania na wspólne zakończenie

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

void *thread_function(void *arg);

char message[] = "Hello World";
int thread_finished = 0;

int main() {
   int res;
   pthread_t a_thread;
   void *thread_result;
   pthread_attr_t thread_attr;

   res = pthread_attr_init(&thread_attr);
   if (res != 0) {
      perror("Attribute creation failed");
      exit(EXIT_FAILURE);
   }
   res = pthread_attr_setdetachstate(&thread_attr,
PTHREAD_CREATE_DETACHED);
   if (res != 0) {
      perror("Setting detached attribute failed");
      exit(EXIT_FAILURE);    }
```

```
res = pthread_create(&a_thread, &thread_attr,
thread_function, (void *)message);
   if (res != 0) {
      perror("Thread creation failed");
      exit(EXIT_FAILURE);
   }
   (void)pthread_attr_destroy(&thread_attr);
   while(!thread_finished) {
      printf("Waiting for thread to say it's finished...\n");
      sleep(1);
   }
   printf("Other thread finished, bye!\n");
   exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
   printf("thread_function is running. Argument was
%s\n", (char *)arg);
   sleep(4);
   printf("Second thread setting finished flag, and exiting
now\n");
   thread_finished = 1;
   pthread_exit(NULL);
}
```

Wyjście programu

```
$ ./thread5
Waiting for thread to say it’s finished...
thread_function is running. Argument was Hello World
Waiting for thread to say it’s finished...
Waiting for thread to say it’s finished...
Waiting for thread to say it’s finished...
Second thread setting finished flag, and exiting now
Other thread finished, bye!
```

Komentarz

- W programie została zainicjalizowana zmienna typu pthread_attr_t
- Przed uruchomieniem wątku wykonana została funkcja pthread_attr_setdetachstate(&thread_attr, PTHREAD_CREATE_DETACHED);
- funkcja zmodyfikowała zmienną thread_attr w taki sposób, że, jeśli zostanie utworzony wątek z tą zmienną jako atrybut, to po jego zakończeniu wątek macierzysty nie będzie oczekiwał na wątek potomny

Wielo-wielowątkowa aplikacja

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

#define NUM_THREADS 6

void *thread_function(void *arg);

int main() {
   int res;
   pthread_t a_thread[NUM_THREADS];
   void *thread_result;
   int lots_of_threads;

   for(lots_of_threads = 0; lots_of_threads <
NUM_THREADS; lots_of_threads++) {

      res = pthread_create(&(a_thread[lots_of_threads]),
NULL, thread_function, (void *)&lots_of_threads);
      if (res != 0) {
         perror("Thread creation failed");
         exit(EXIT_FAILURE);
      }
      sleep(1);
   }
```

```
printf("Waiting for threads to finish...\n");
   for(lots_of_threads = NUM_THREADS - 1;
lots_of_threads >= 0; lots_of_threads--) {
      res = pthread_join(a_thread[lots_of_threads],
&thread_result);
      if (res == 0) {
         printf("Picked up a thread\n");
      }
      else {
         perror("pthread_join failed");
      }
   }
   printf("All done\n");
   exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
   int my_number = *(int *)arg;
   int rand_num;

   printf("thread_function is running. Argument was
%d\n", my_number);
   rand_num=1+(int)(9.0*rand()/(RAND_MAX+1.0));
   sleep(rand_num);
   printf("Bye from %d\n", my_number);
   pthread_exit(NULL);
}
```

Wyjście programu

```
$ ./thread8
thread_function is running. Argument was 0
thread_function is running. Argument was 1
thread_function is running. Argument was 2
thread_function is running. Argument was 3
thread_function is running. Argument was 4
Bye from 1
thread_function is running. Argument was 5
Waiting for threads to finish...
Bye from 5
Picked up a thread
Bye from 0
Bye from 2
Bye from 3
Bye from 4
Picked up a thread
Picked up a thread
Picked up a thread
Picked up a thread
Picked up a thread
All done
```

Jak to działa?

- W pętli następuje wygenerowanie NUM_THREADS wątków.
- Każdy z wątków ustala losowy czas, kiedy ma zakończyć swoje obliczenia
- Główny program „zbiera” zakończone wątki (uwaga: wątki kończą si w różnej kolejności, ale „zbieranie” odbywa się po kolei zgodnie z numerami wątków zapisanymi w tablicy a_thread)

Interesujące zjawisko

- Interesujące wyniki programu otrzymuje się, gdy zrezygnuje się w nim z wywołań funkcji sleep:
  - okazuje się, że kilka wątków „startuje” z tą samą wartością przekazywaną im jako parametr wywołania
  - problematyczną linią programu jest:
    for(lots_of_threads = 0; lots_of_threads < NUM_THREADS; lots_of_threads++) {res = pthread_create(&(a_thread[lots_of_threads]), NULL, thread_function, **(void *)&lots_of_threads**); …
  - do wątku przekazywane jest wskazanie na zmienną, pomimo, że zmienna zmienia się w pętli, komórka pamięci, w której powinien być przechowywany jej stan nie jest aktualizowana
  - Rozwiązanie problemu:
    res = pthread_create(&(a_thread[lots_of_threads]), NULL, thread_function, **(void *)lots_of_threads**); …
  - Modyfikacji musi też ulec fragment funkcji obsługującej wątek:

```
void *thread_function(void *arg) {
int my_number = (int)arg; …
```