POSIX threads
Andrzej Janaszek (ITE)
13.11.2025

ZARZĄDZANIE WĄTKAMI I ICH PRACĄ

biblioteka dla wątków posix:
`#include <pthread.h>`

Tworzenie wątku:

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

pthread_create(id wątku, atrybuty, funkcja startowa, argumenty)

Tworzy nowy wątek zaczynający pracę od funkcji startowej (start_routine) i zwraca inta, sukces = 0, albo wartość != 0 czyli numer błędu.
Wątkowi można podać atrybuty i argumenty.

Oczekiwanie na wykonanie wątku (join):

```
int pthread_join(pthread_t thread, void **retval);
```

pthread_join(id wątku, wynik zwracany)

Czeka na zakończenie wątku (w nieskończoność - bez określonego czasu zakończenia). Możemy również uzyskać wartość zwracaną przez wątek jako wynik zwracany (retval).

Przerwanie pracy wątku (cancel):

```
int pthread_cancel(pthread_t thread);
```

pthread_cancel(id wątku)

Funkcja **pthread_cancel** wysyła sygnał żądania przerwania pracy do wątku. Sygnał rozpatrywany jest w zależności od stanu i typu przerywalności (eng. cancelability *state* and *type*).

Stan przerywalności (**state**):

Ustawiany przez **pthread_setcancelstate**(), może być włączony (enabled) lub wyłączony (disabled). Jeżeli wątek ustawić stan na wyłączony, to żądanie przerwania pracy będzie zakolejkowane do czasu ustawienia stanu na włączony.

Typ przerywalności (**type**):

Ustawiany przez **pthread_setcanceltype**(), może być asynchroniczny (asynchronous) lub odroczony (deferred), co jest wartością domyślną.

Typ asynchroniczny oznacza, że wątek może zostać przerwany w dowolnym momencie (zazwyczaj od razu ale nie jest to gwarantowane przez system).

Typ odroczony oznacza, że przerwanie wątku będzie odroczone aż do wywołania funkcji, która może być punktem przerwania (cancelation point).

Kod 1. Funkcja wątku ze zmianą stanu przerwania

```
void* zadanie_watku (void * arg_wsk){
      pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
      // kod - brak możliwości zabicia wątku
      pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);

      // funkcja punktu przerwania (cancelation point)
      pthread_testcancel();

      return(NULL);
}
```

Kod 2. Próba wywołania przerwania z wątku głównego

```
// Wysłanie sygnału zabicia wątku (tid - id procesu)
pthread_cancel(tid);
```

Zakończenie pracy wątku (exit):

void **pthread_exit**(void *retval);
Wywołanie funkcji **pthread_exit**() kończy pracę wątku. Przez argument funkcji możemy zwrócić, która będzie dostępna przy wywołaniu **pthread_join** (jeżeli wątek nie jest odłączony).

Atrybuty wątku:

Możemy określić atrybuty wątku, w tym celu musimy:

- utworzyć obiekt klasy pthread_attr_t
- zainicjować go funkcją **pthread_attr_init**(pthread_attr_t *attr)
- ustawić atrybuty
- utworzyć wątek
- *usunąć atrybuty funkcją **pthread_attr_destroy**(pthread_attr_t *attr)

Przykładowe atrybuty możliwe do ustawienia:

- Odłączalność wątku (detach)
  int **pthread_attr_setdetachstate**(pthread_attr_t *attr, int detachstate);
- Rozmiar stosu
  pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);

Kod 3. Przykładowy kod przedstawiający ustawienie atrybutów wątków

```
pthread_t t_id;
pthread_attr_t attr;

// inicjalizacja obiektu atrybutów
pthread_attr_init(&attr);

// ustawienie rozmiaru stosu
size_t rozmiar = 16*1024*1024;
pthread_attr_setstacksize(&attr, rozmiar);
```

```
// ustawienie odłączania wątku
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

pthread_create(&t_id, &attr, thread_fun, NULL);

// usunięcie atrybutów
pthread_attr_destroy(&attr);

pthread_join(t_ids, NULL);
```

Przesyłanie argumentów do wątku:

Do wątku możemy przesłać argumenty podając referencje do nich jako ostatni argument w funkcji tworzenia wątku.

Kod 4. Przesłanie prymitywu i struktury jako argument wątku

```
// Utworzenie prymitywu i struktury
double prymityw = 42;
struct Pub struktura;

// ... reszta kodu ...

// przesłanie prymitywu jako argument
pthread_create(t_id1, NULL, f_watku, &prymityw);

// przesłanie referencji do struktury pub jako argument
pthread_create(t_id2, NULL, f_watku, &struktura);
```

W wątku możemy odnieść się do przesłanego argumentu na dwa sposoby:

- wykonać lokalną kopie wartości
- operować na wskaźniku

Kod 5. Przykład dostępu do argumentu w sposób wskaźnikowy i na wartości lokalnej

```
void* f_watku(void * args){
    // WSKAŹNIK
    // rzutujemy jedynie typ wskaźnika na typ wskaźnikowy naszej struktury
    struct Pub* p = (struct Pub*)args;

    // KOPIA LOKALNA WARTOŚCI
    // lub kopiujemy strukturę lokalnie - dodatkowo stosujemy dereferencje
    struct Pub kopia = *( (struct Pub*)args );

    // reszta kodu
}
```

Oczywiście zmiany na kopii lokalnej nie będą widoczne poza funkcją wątku, ponieważ są wykonywane na lokalnej kopii. Natomiast operacje na wskaźniku będą widoczne poza funkcją (w wątku wywołującym, jak i w innych wątkach jeżeli przekażemy do nich ten sam

obiekt/referencje). Dlatego trzeba wziąć pod uwagę sytuację wyścigu (race condition) podczas dostępu do tych danych przez więcej niż jeden wątek na raz.

WSPÓŁDZIAŁANIE WĄTKÓW - WSPÓŁDZIELENIE ZASOBÓW - MUTEX

Podczas próby dostępu do tych samych zasobów przez więcej niż jeden wątek może powstać sytuacja wyścigu (race condition). Aby uniknąć takiej sytuacji możemy zastosować dwie strategie:

- mutex (od angielskiego mutual exclusion - wzajemne wykluczenie). Blokujemy dostęp do współdzielonego zasobu, tak aby tylko jeden wątek na raz miał do niego dostęp
- Tworzymy tablicę zasobów (każdy wątek ma swój zasób), następnie po wykonaniu wszystkich wątków ja np. agregujemy. Tutaj właściwie nie rozwiązujemy problemu dostępu do **jednego** wspólnego zasobu, lecz rozdzielamy problem na mniejsze niekolidujące.

MUTEX:

Mutex pozwala zablokować dostęp do zasobu dzięki czemu korzystać z niego będzie tylko jeden wątek na raz. Wykorzystanie mutexu (w wątku) wiąże się zablokowaniem (lock), wykonaniem operacji w tzw. sekcji krytycznej oraz ze zwolnieniem zasobu (mutex unlock).

Funkcje mutexu:

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

Kod 6. Stworzenie, zainicjalizowanie i usunięcie mutexa w funkcji głównej programu

```
// globalnie dostępny / widoczny dla wątków
pthread_mutex_t mutex;

pthread_mutex_init(&mutex, NULL);

// kod progamu - tworzenie wątków i oczekiwanie na zakończenie

pthread_mutex_destroy(&mutex);
```

Kod 7. Użycie mutex w funkcji wątku

```
void* f_watku(void * args){
    pthread_mutex_lock(&mutex);
    // operacja w sekcji krytycznej
    pthread_mutex_unlock(&mutex);

    // reszta kodu
}
```

Funkcja **pthread_mutex_lock**() czeka na możliwość zablokowania zasobu.
Funkcja **pthread_mutex_trylock**() pozwala na próbę zablokowania zasobu, dla sukcesu zwracana jest wartość 0 oraz zasób jest zablokowany. Dzięki temu możemy sprawdzić czy wątek może zablokować zasób, jeżeli nie, to może wykonać inne działanie i sprawdzić ponownie. Dzięki temu zamiast czekać na możliwość wykorzystania zasobu możemy przeprowadzić inne działania -> lepiej wykorzystać zasoby obliczeniowe.

Kod 8. Przykład wykorzystania funkcji pthread_mutex_trylock()

```
int success = 0;
do{
    if(pthread_mutex_trylock(&mutex) == 0){
        // wykonywanie operacji w sekcji krytycznej
        success = 1;  // operacja wykonana
    }
    else{
        // brak możliwości dostępu do zasobu
        // wykonanie innej czynności przez wątek
    }
}while(success == 0);
```

Tablica zasobów:

Strategia wygląda następująco:

- Tworzymy tablicę zasobów
- przekazujemy referencje do konkretnego zasobu każdemu wątkowi
- wątki wykonują swoją pracę
- agregujemy wyniki

Porównanie mutex i tablicy zasobów:

Dla tablicy wątki wykonują swoją pracę równolegle i dopiero podczas agregacji następuje łączenie wyników. Łączy się to z alokacją większej liczby zasobów -> tablicy. Oraz z czasem agregacji tych zasobów po wykonaniu pracy wątków.

Dla mutexa nie alokujemy dodatkowych zasobów, ponieważ pracujemy na jednym wspólnym, natomiast wiąże się to z możliwością czekania na zwolnienie zasobu. Nie ma tutaj natomiast etapu agregacji wyników.
Dobór i szybkość metody zależy od danego przypadku.

Typy dekompozycji pętli dla wątków:

Wyróżnione zostaną 3 typy:

- cykliczny
- blokowy
- domenowy (podział domeny)

Możliwe oczywiście są typy mieszane.

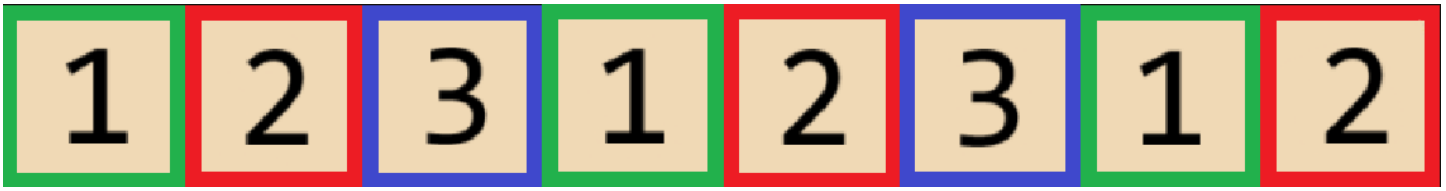
Podział cykliczny:

Najłatwiej pokazać to na przykładzie. Załóżmy że mamy 3 wątki o id 1,2,3 oraz 8 zadań. Każde zadanie musi zostać wykonane przez jakiś wątek. Przydzielanie zadań wykonujemy w następujący sposób:
Iterujemy po wątku aż do wyczerpania zadań (od 1 do ostatniego wątku i od nowa) i przydzielamy dane zadanie do wątku.

Kod 9. Pseudokod przydzielania cyklicznego zadań wątkom (indeksowanie od 0)

```
for (i = 0; i < liczba_zadań; i++)
    dla wątku nr. (i % liczba_wątków)
    przydziel zadanie nr. i
```

Rys. 1. Ilustracja podziału cyklicznego. Cyfry to numery wątków, a pola to zadania

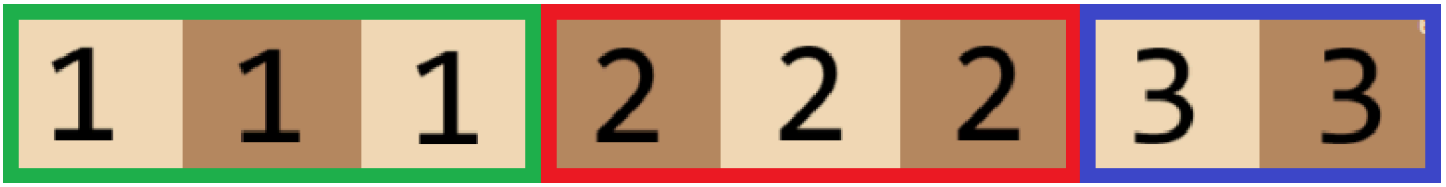


Dla podziału cyklicznego maksymalna różnica między ilością zadań dla wątków wynosi 1. Jeżeli ilość zadań jest wielokrotnością liczby wątków, każdy wątek otrzyma tyle samo zadań do wykonania. W przeciwnym wypadku część wątków będzie miała 1 zadanie więcej.
Np. dla 5 zadań i 4 wątków:
wątek 1 będzie miał 2 zadania, a pozostałe wątki tylko 1.

Podział blokowy:

Dzielimy liczbę zadań przez liczbę wątków i zaokrąglamy w górę, w ten sposób otrzymujemy liczbę zadań na wątek. Do wątków przydzielamy zadaną liczbę zadań dopóki są one dostępne.

Rys. 2. Ilustracja podziału blokowego. Cyfry to numery wątków, a pola to zadania



Kod 10. Pseudokod przydzielania blokowego zadań wątkom (indeksowanie od 0)

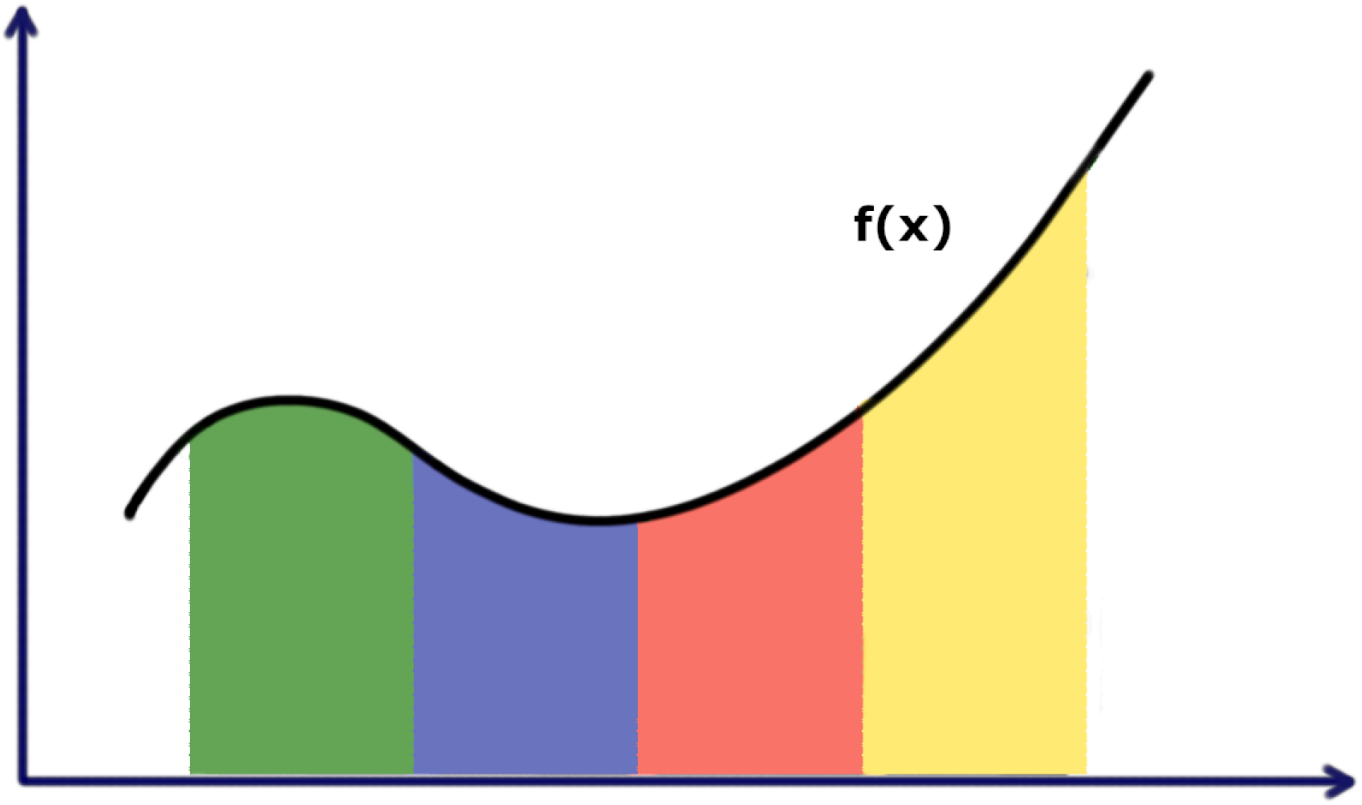
```
zadania_per_watek = ceil(liczba_zadań / liczba_watków)
for(i = 0; i < liczba_watkow; i++){
    // dla wątku przydziel zadania:
    poczatek = i * zadania_per_watek
    koniec = (i+1) * zadania_per_watek

    if koniec >= (liczba_zadań-1)
        koniec = liczba_zadań-1
        // Zadania się skończyły
        // do pozostałych wątków nie przypisuj zadań
        break;
}
```

Może zajść sytuacja dla której pewne wątki będą bezczynne, a jeden będzie miał mniejszą liczbę zadań od pozostałych. Np. dla 100 zadań i 30 wątków. Liczba zadań na ątek będzie wynosiła 4, więc 25 wątków dostanie 4 zadania a pozostałe 5 nie dostanie żadnego. Jeżeli liczba zadań byłaby większa np. o 1 to 26-ty wątek dostałby 1 zadanie a pozostałe 4 brak przydziału.

Podział domeny:
Dzielimy domenę problemu na ilość wątków, a następnie każdemu przydzielamy tego poddomenę, na której wykonuje zadanie. Wyniki agregujemy albo każdy wątek robi to samodzielnie.

Przykład:
Obliczenie całki na przedziale [a;b]. Dzielimy odcinek na N mniejszych, gdzie N to ilość wątków. Każdemu wątkowi podajemy jego poddomenę (własny pododcinek odcinka [a;b]), na której wykonuje obliczenia. Następnie sumujemy wyniki dla każdego z pododcinków.

Rys 3. Ilustracja podziału domeny. Każdy kolor odpowiada osobnemu wątkowi.



Porównanie czasów wykonania programu sekwencyjnie i równolegle dla obliczania sumy.

Tabela 1. Czasy wykonania programu sekwencyjnie

ROZMIAR	OPT	sekwencyjne [s]
1000	debug	0.000008
	-O3	0.000002
1000000	debug	0.008207
	-O3	0.002271

Tabela 2. Czasy wykonania programu przy użyciu wątków z mutex’em oraz bez mutex’u.

Zestawienie dla obliczania sumy				
Ilość wątków	ROZMIAR	OPT	czas w [s]	
			mutex	no mutex
1	1000	debug	0.000620	0.000167
		-O3	0.000657	0.000146
	1000000	debug	0.008421	0.007981
		-O3	0.002945	0.002336
2	1000	debug	0.000658	0.000183
		-O3	0.000667	0.000241
	1000000	debug	0.004748	0.004165
		-O3	0.001726	0.001271
4	1000	debug	0.000823	0.000251
		-O3	0.000868	0.000234
	1000000	debug	0.003068	0.002056
		-O3	0.001389	0.000846

Tabela 3. Porównanie czasów wykonania z wątkami do sekwencyjnego.

Zestawienie dla obliczania sumy				
Ilość wątków	ROZMIAR	OPT	czas w odniesieniu do sekwencyjnego wykonania	
			mutex	no mutex
1	1000	debug	8270.00%	2220.00%
		-O3	29211.11%	6477.78%
	1000000	debug	102.60%	97.24%
		-O3	129.70%	102.89%
2	1000	debug	8770.00%	2440.00%
		-O3	29655.56%	10722.22%
	1000000	debug	57.85%	50.75%
		-O3	76.02%	55.98%
4	1000	debug	10976.67%	3346.67%
		-O3	38577.78%	10411.11%
	1000000	debug	37.38%	25.05%
		-O3	61.18%	37.24%

Wnioski z tabeli 3. (porównawczej):

- dla małego rozmiaru tablicy (rozmiar = 1000) czasy względne (wykonanie z użyciem wątków w porównaniu do sekwencyjnego) są bardzo duże, co wynika z narzutu związanego z utworzeniem wątków i/lub obsługą mutexów.
- Różnice w czasach dla wersji bez optymalizacji (debug) oraz z optymalizacją -O3 mogą wynikać z tego że kompilator jest w stanie lepiej zoptymalizować program sekwencyjny. Dlatego dla wersji z optymalizacją względne czasy wykonania są dłuższe.
- Dla 2 oraz 4 wątków można zauważyć przewidywane przyśpieszenie pracy programu. Dla wersji bez mutexu jest ono prawie idealnie odwrotnie proporcjonalne do liczby wątków, natomiast dla wykonania z użyciem mutexów wyraźnie widać narzut z tym związany. Najprawdopodobniej wykonanie każdego wątku trwa bardzo zbliżony czas i operacja zapisu przez użycie mutexu wykonuje się sekwencyjnie.

Porównanie dokładności obliczania całki sekwencyjnie w zależności od kroku całkowania (dx)

Tabela 4. Tabela przedstawiająca wyliczenia sekwencyjne wartości całki, błąd bezwzględny oraz błąd względny dla danych kroków całkowania (dx)

dx	0.1	0.01	0.0001	0.00001	0.000001
wartość całki	1.99839336097014	1.99998342215375	1.99999999833334	1.99999999998338	1.99999999999975
błąd bezwzględny	0.00160663902986	0.00001657784625	0.00000000166666	0.00000000001662	0.00000000000025
błąd względny	0.08033195149%	0.00082889231%	0.00000008333%	0.00000000083%	0.00000000001%

Zgodnie z założeniami z instrukcji dla dx = 0.000001 błąd jest mniejszy niż 12 miejsc znaczących.

Źródła:

[1] pthread_create - https://man7.org/linux/man-pages/man3/pthread_create.3.html
[2] pthread_join - https://man7.org/linux/man-pages/man3/pthread_join.3.html
[3] pthread_cancel- https://www.man7.org/linux/man-pages/man3/pthread_cancel.3.html
[4] cancelation points - https://www.man7.org/linux/man-pages/man7/pthreads.7.html
[5] pthread_exit - https://man7.org/linux/man-pages/man3/pthread_exit.3.html
[6] pthread_attr_init- https://man7.org/linux/man-pages/man3/pthread_attr_init.3.html
[7] pthread_attr_setstacksize - https://man7.org/linux/man-pages/man3/pthread_attr_setstacksize.3.html
[8] pthread_attr_setdetachstate - https://man7.org/linux/man-pages/man3/pthread_attr_setdetachstate.3.html
[9] pthread_mutex_init - https://man7.org/linux/man-pages/man3/pthread_mutex_lock.3.html
[10] wykłady profesora Banasia - https://galaxy.agh.edu.pl/~kbanas/PR/PR.html