Posix threads (pthreads -wątki)

pthreads

- Standardy:
  - POSIX 1003.1c 1995 r
  - ISO/IEC 9945-1 1996 r
  - 1003.4a - DCE
  - ANSI / IEEE 1003.1
- Dostępne w większości systemów Unix (Linux)
- Często implementowane na poziomie kernela
- Bardziej przenośne niż inne implementacje wątków (np. Boost)
- Silnie “zintegrowane” ze środowiskiem Unix – głównie w zakresie obsługi sygnałów

dlaczego wątki?

- Współbieżne procesy:
  - fork() jest kosztowny
  - komunikacja jest trudna
  - powielanie kodu
  - zajętość pamięci operacyjnej
  - wyczerpywanie zasobów jądra
  - spowalnianie
  - kłopoty z synchronizacją
  - “migotanie” zawartości cache procesora

wątki

- **Wątek – odrębny tok wykonania w pocesie**
- Wątki współdzielą większość danych procesu, z wyjątkiem:
  - identyfikatora wątku, priorytetu
  - stosu i rejestrów
  - kodu powrotu z funkcji systemowej; errno
  - maski obsługi sygnałów
- Dostęp do wspólnych danych wymaga synchronizacji, dane wspólne:
  - dane globalne
  - sterta (malloc(), calloc(), także niejawnie przydzielane)
  - Koncepcja: “MT-safe” (multi-thread safe)
    - niektóre funkcje systemowe posiadają zmienne typu stattic: srand(), strtok() -> strtok_r()
    - niektóre funkcje systemowe nie są MT-safe, np. resolver (gethostbyname(), itd.)

podstawowe funkcje wątków

```
#include <pthread.h>

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

- Tworzy nowy wątek
- Zwraca 0 lub kod błędu:
  - [EAGAIN] brak zasobów lub przekroczenie limitu liczby wątków PTHREAD_THREADS_MAX
  - [EINVAL] niepoprawny argument
  - [EPERM] brak uprawnień (dot. szeregowania)
- Zakończenie wątku:
  - `void pthread_exit(void *value_ptr);`
- Porównanie deskryptorów wątków:
  - `int pthread_equal(pthread_t t1, pthread_t t2);`

tworzenie wątków

```
#define _REENTRANT
#include <pthread.h>
#include <stdio.h>
void *print_message_function(void
*ptr);

int main(int argc, char**argv) {
  pthread_t thread1, thread2;
  char *message1 = "Hello";
  char *message2 = "World";

  pthread_create(&thread2, NULL,
(void *) &print_message_function,
(void *) message2);

  pthread_create(&thread1, NULL,
(void *) &print_message_function,
(void *) message1);
  return 0;
}
```

```
void *print_message_function(
   void *ptr )
{
  char *message;
  message=(char *) ptr;
  printf("%s ", message);
  pthread_exit(NULL);
}
```

synchronizacja wątków

```
int main( int argc, char**argv ) {
  char *message1 = "Hello";
  char *message2 = "World";

  pthread_create(&thread2, NULL, (void *) &print_message_function,
     (void *) message2);
  pthread_create(&thread1, NULL,  (void *) &print_message_function,
     (void *) message1);
  pthread_join(thread2, NULL);
  return 0;
}

void *print_message_function(void *ptr) {
  char *message;
  message=(char *) ptr;
  if( pthread_equal(pthread_self(),thread1)==0 )
      pthread_join( thread1, NULL );
  printf("%s ", message);
  pthread_exit(NULL);
}
```

pthread_join()

- `int pthread_join(pthread_t thread, void **value_ptr);`
- Zawiesza wątek do momentu zakończenia wskazanego wątku
- value_ptr odbiera kod powrotu wątku zakończonego
- Zwraca 0 jeśli poprawne zakończenie lub kod błędu

mutex

- Mutex – odpowiednik semafora binarnego (lub wielo wartościowego), podstawowy obiekt synchronizacyjny dla wątków
- obiekt atrybutów mutexu
  - `pthread_mutexattr_init(pthread_mutex_t *);`
  - `pthread_mutexattr_destroy(pthread_mutex_t *);`
- inicjowanie
  - `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER`
  - `pthread_mutex_init(pthread_mutex_t *);`
- likwidowanie pthread_mutex_destroy()
- blokowanie
  - `pthread_mutex_lock(pthread_mutex_t *);`
  - `pthread_mutex_trylock(pthread_mutex_t *);`
  - `pthread_mutex_unlock(pthread_mutex_t *);`

mutex

- Mutex domyślnie jest binarny
- Problem - co się stanie gdy ten sam wątek kilkukrotnie będzie zajmować mutex?
- Zachowanie mutaxa można zmienić:
  - PTHREAD_MUTEX_NORMAL - ponowne zajęcie powoduje deadlock
  - PTHREAD_MUTEX_ERRORCHECK - mutex binarny, ponowne zajęcie powoduje błąd
  - PTHREAD_MUTEX_RECURSIVE - mutex wielowartościowy
  - PTHREAD_MUTEX_DEFAULT - zachowanie przy ponownym zajęciu nieokreślone

- ustalenie typu:
  - `pthread_mutexattr_settype(pthread_mutex_t *, int type);`

mutex

```
#define _REENTRANT
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char **argv) {
  pthread_t reader, writer;
  pthread_create(&reader, NULL, (void *) &reader_f, NULL);
  pthread_create(&writer, NULL, (void *) &writer_f, NULL);
  pthread_join(writer, NULL);
  return 0;
}

void *writer_f(void *ptr) {
  char ch;  ch='a';
  while(1)  {
    pthread_mutex_lock( &mutex );
    if(buffer_has_item == 0)    {
      buffer = ch;
      buffer_has_item = 1;
     }
    pthread_mutex_unlock( &mutex );
```

mutex

```
    ch++;
    if(ch>'z') ch='a';
    usleep(1);
    }
  pthread_exit(NULL);
}


void *reader_f(void *ptr)
{
  while(1)  {
    pthread_mutex_lock( &mutex );

    if(buffer_has_item == 1)    {
      printf("%c ", buffer);
      fflush(stdout);
      buffer_has_item = 0;
      }

    pthread_mutex_unlock( &mutex );
    usleep(1);
    }
  pthread_exit(NULL);
}
```

zmienne warunkowe - cond

- Mutex nie jest wystarczający jeśli oczekujemy na jakieś zdarzenie

  np: czekanie na akcje/zakończenie *jakiegoś* wątku

  – aktywne oczekiwanie jest (prawie) zawsze złym rozwiązaniem
- cond zawsze używany w parze z mutexem
- **cond_wait**: usypia wątek w oczekiwaniu na *signal* i zwalnia mutex
- **cond_signal**: wysyła sygnał
- typowa sekwencja:
  – T1: L(m); n++; S(c); U(m);
  – T2: L(m); while (*pred*(n)) W(c,m);

cond

- obiekt atrybutów zmiennych warunkowych
  - `pthread_condattr_init()`
  - `pthread_condattr_destroy()`
- inicjowanie zmiennych warunkowych
  - `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`
  - `pthread_cond_init()`
- likwidowanie zmiennych warunkowych
  - `pthread_cond_destroy()`
- blokowanie na zmiennych warunkowych
  - `pthread_cond_wait(&cond, &mutex)`
  - `pthread_cond_timedwait(&cond, &mutex)`
  - *`int pthread_condattr_setclock(pthread_condattr_t *attr, clockid_t clock_id);`*
  - `pthread_cond_signal(&cond)`
  - `pthread_cond_broadcast(&cond)`

cond

```
char buffer;
int buffer_has_item = 0;
pthread_mutex_t mutex_w = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex_r = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_reader =PTHREAD_COND_INITIALIZER;
pthread_cond_t cond_writer =PTHREAD_COND_INITIALIZER;

void *writer_f(void *ptr)
{ int i;  char ch = 'a';

  while(1)  {
    pthread_mutex_lock(&mutex_w);
    while(buffer_has_item != 0)
     pthread_cond_wait(&cond_writer, &mutex_w);
    pthread_mutex_unlock(&mutex_w);

    buffer = ch++;
    if(ch>'z') ch='a';
    buffer_has_item = 1;
    pthread_mutex_lock(&mutex_r);
    pthread_cond_signal(&cond_reader);
    pthread_mutex_unlock(&mutex_r);
    }
  pthread_exit(NULL);
}
```

cond

```
void *reader_f(void *ptr)
{ int i;

  while(1)  {
    pthread_mutex_lock( &mutex_r );
    while(buffer_has_item == 0)
     pthread_cond_wait(&cond_reader, &mutex_r);
    pthread_mutex_unlock( &mutex_r);
    printf("%c ", buffer);
    fflush(stdout);
    buffer_has_item = 0;
    pthread_mutex_lock(&mutex_w);
    pthread_cond_signal(&cond_writer);
    pthread_mutex_unlock(&mutex_w);

    }
   pthread_exit(NULL);
}
```

rd/wr lock

- `int` **`pthread_rwlock_wrlock`**`(pthread_rwlock *rwlock);`
- `int` **`pthread_rwlock_rdlock`**`(pthread_rwlock *rwlock);`
- `int` **`pthread_rwlock_unlock`**`(pthread_rwlock *rwlock);`

- Blokada typu czytelnicy / pisarz
- pthread_rwlock inicjalizowany podobnie jak mutexy
- Dostępne też funkcje: **`pthread_rwlock_trywrlock`**, **`pthread_rwlock_init`**, **`pthread_rwlockattr_init`**

Kończenie wątków

- Żądanie zakończenia `pthread_cancel(thread)`
  - dla wątku określone: “cancel state” i “cancel type”
  - cancel state – czy można “zdalnie” zakończyć wątek
  - cancel type – jak wątek jest kończony: natychmiast, przy określonych operacjach
  - int pthread_setcanceltype(int type, int *oldtype);
  - `int **pthread_setcancelstate**(int state, int *oldstate); state: PTHREAD_CANCEL_ENABLE, PTHREAD_CANCEL_DISABLE`
  - `int **pthread_setcanceltype**(int type, int *oldtype); state: PTHREAD_CANCEL_DEFERRED, PTHREAD_CANCEL_ASYNCHRONOUS`
- punkty zakończenia:
  - `pthread_join()`
  - `pthread_cond_wait()`
  - `pthread_cond_timedwait()`
  - `pthread_testcancel()`
  - `sem_wait()`
  - `sig_wait()`
  - `read()`
  - `write()`

Atrybuty wątków

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

- Funkcja inicjalizuje/niszczy obiekt atrybutów wątku
- Poszczególne atrybuty można ustawiać indywidualnie
- Zniszczenie obiektu nie ma wpływu na watki nim zainicjowane
- Typowe domyślne atrybuty wątku:
  - `Detach state = PTHREAD_CREATE_JOINABLE`
  - `Scope = PTHREAD_SCOPE_SYSTEM`
  - `Inherit scheduler = PTHREAD_INHERIT_SCHED`
  - `Scheduling policy = SCHED_OTHER`
  - `Scheduling priority = 0`
  - `Guard size = 4096 bytes`
  - `Stack address = 0x.....`
  - `Stack size = 0x.....`

Atrybuty wątków c.d.

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

- Pozwala/zakazuje operacji join: PTHREAD_CREATE_DETACHED, PTHREAD_CREATE_JOINABLE

```
int int pthread_attr_setscope(pthread_attr_t *attr, int
    scope);
```

- Określa “pulę” wątków schedulera: PTHREAD_SCOPE_SYSTEM – watek należy do systemowej puli, PTHREAD_SCOPE_PROCESS – wątek należy do puli procesu
- Linux obsługuje tylko pulę systemową!

```
int pthread_attr_setguardsize(pthread_attr_t *attr, size_t
    guardsize);
```

- Ustanawia dodatkowy obszar na końcu stosu wątku, nadpisanie tego obszaru generuje SIGSEGV
- Rozmiar powinien być > rozm. strony

Atrybuty wątków c.d.

```
int pthread_attr_setstack(pthread_attr_t *attr,
          void *stackaddr, size_t stacksize);
int pthread_attr_setstackaddr(pthread_attr_t *attr,
          void *stackaddr);
int pthread_attr_setstacksize(pthread_attr_t *attr,
          size_t stacksize);
```

- Ustala adres i rozmiary stosu dla wątku
- stos moze być dowolnym obszrem pamięci wcześniej przydzielonym

Sygnały "procesowe" i wątki

- Wątki mają odrębne maski sygnałów.
- Możliwe jest wybiórcze dostarczanie sygnałów do wątków poprzez blokowanie (wstrzymywanie) sygnałów w tych wątkach, które nie powinny otrzymywać sygnałów.
- **Uwaga**: stosowanie wątków nie zmienia koncepcji obsługi sygnałów przyjętej w modelu jednowątkowym procesu.
- “Ogólno - procesowe” skutki sygnału skutkują dla całego procesu (a nie wątku), np: SIGKILL, SIGSTOP zabija/wstrzymuje cały proces, a nie odbierający watek.
- Wątki w zakresie działań na masce sygnałów zwykle nie powinny wywoływać “procesowych” funkcji obsługi sygnałów np. sigprocmask(), a jedynie funkcje przeznaczone dla wątków: pthread_sig*()
- int pthread_kill(pthread_t thread, int sig); <- sygnał może być dostarczony do wszystkich wątków
- typowa obsługa sygnałów: wątki maja zablokowaną obsługę sygnałów, jeden wyróżniony wątek odpowiada za ich obsługę

Przypisanie wątku do obsługi sygnału

```
pthread_mutex_t stats_lock=PTHREAD_MUTEX_INITIALIZER;
int samples;
/* wątek obsługi sygnału SIGUSR1 */
void *report_stats(void *p) {
  int caught;
  sigset_t  sigs_to_catch;

/* Odziedziczyliśmy maskę z blokowaniem sygnału SIGUSR1. Ponieważ wszystkie
inne wątki również blokują ten sygnał i tylko my oczekujemy na niego za pomocą
sigwait() mamy pewność, że każde wystąpienie SIGUSR1 zostanie dostarczone do
nas. */
  sigemptyset(&sigs_to_catch);
  sigaddset(&sigs_to_catch, SIGUSR1);
  for (;;) {                              // obsługa przybycia sygnału
     sigwait(&sigs_to_catch, &caught);
     // otrzymaliśmy sygnał SIGUSR1
     pthread_mutex_lock(&stats_lock);
     printf("report_stats(): samples = %d\n", samples);
     pthread_mutex_unlock(&stats_lock);
  }
  return NULL;
}
```

Przypisanie wątku do obsługi sygnału

```
void *worker_thread(void *p) { /* wątek - robotnik */
  for (;;) {
    sleep(5);
    pthread_mutex_lock(&stats_lock);
    samples++;
    pthread_mutex_unlock(&stats_lock);
  }
  return NULL;
}

int main(int argc, char **argv) {
  int       i;
  pthread_t threads[MAX_NUM_THREADS];
  int       num_threads = 0;
  sigset_t  sigs_to_block;

// Ustawiamy blokowanie sygnału SIGUSR1. Pozostałe wątki odziedziczą tą maskę.
  sigemptyset(&sigs_to_block);
  sigaddset(&sigs_to_block, SIGUSR1);
  pthread_sigmask(SIG_BLOCK, &sigs_to_block, NULL);
```

Przypisanie wątku do obsługi sygnału

```
/* tworzymy wątek obsługujący SIGUSR1 */
  pthread_create(&threads[num_threads++], NULL, report_stats, NULL);

/* pozostałe wątki - robotnicy */
  for (i=num_threads; i<MAX_NUM_THREADS; i++) {
      pthread_create(&threads[num_threads++], NULL, worker_thread, NULL);
      }
  for (i = 0; i < num_threads; i++)
    pthread_join(threads[i], NULL);
  return 0;
}
```