Aplikasi PThread

Kuliah#6 TSK617 Pengolahan Paralel - TA 2011/2012

Eko Didik Widianto

Teknik Sistem Komputer - Universitas Diponegoro

Tentang Kuliah

- Sebelumnya dibahas tentang: pustaka Posix Thread
  - Pustaka Pthread
  - API Pthread
  - Mengkompile program multithread
  - Mengelola Thread
  - Variabel Mutex
  - Variabel Kondisional
- Dalam kuliah ini akan dibahas tentang aplikasi program menggunakan pustakan PThread

Kompetensi Dasar

- Setelah mempelajari bab ini, mahasiswa akan mampu:
  1. [C3] Mahasiswa akan mampu mengaplikasikan rutin-rutin pustaka pthread untuk memprogram paralel berbasis thread
  2. [C4] Mahasiswa akan mampu memprogram suatu aplikasi berbasis pthread sesuai dengan kebutuhan desain
- Link
  - Website: http://didik.blog.undip.ac.id/2012/02/25/kuliah-tsk-617-pengolahan-paralel-2011/
  - Email: didik@undip.ac.id

Acknowledment

- Materi dan gambar didapat dari:
  - POSIX Threads Programming di https://computing.llnl.gov/tutorials/pthreads/
  - B. Lewis, D.J Berg, Pthreads Primer: A Guide to Multithreaded Programming, 1996. Terutama bab 4 ttg Thread Lifecycle
  - N. Matthew, R. Stones, Beginning Linux Programming 3rd Edition, 2004. Terutama bab 12 ttg Posix Thread
  - Mark Mitchell, Jeffrey Oldham, and Alex Samuel, Advanced Linux Programming , 2001. Terutama bab 4 ttg Posix Thread
  - http://docs.oracle.com/cd/E19963-01/html/821-1601

Bahasan

API PThread

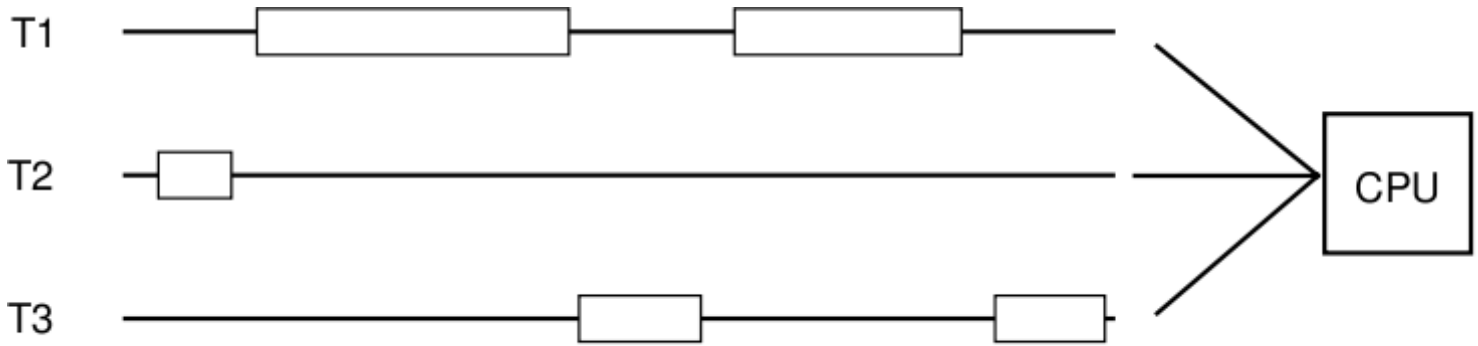
- Pthreads API was defined in the ANSI/IEEE POSIX 1003.1 - 1995 standard
- The subroutines which comprise the Pthreads API can be informally grouped into:
  1. **Thread management**: Routines that work directly on threads - creating, detaching, joining, etc
     - Also include functions to set/query thread attributes (joinable, scheduling etc.)
  2. **Mutexes**: Routines that deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion"
     - Provide for creating, destroying, locking and unlocking mutexes
     - Supplemented by mutex attribute functions that set or modify attributes associated with mutexes
  3. **Condition variables**: Routines that address communications between threads that share a mutex.
     - Based upon programmer specified conditions
     - Includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included
  4. **Synchronization**: Routines that manage read/write locks and barriers

API PThread

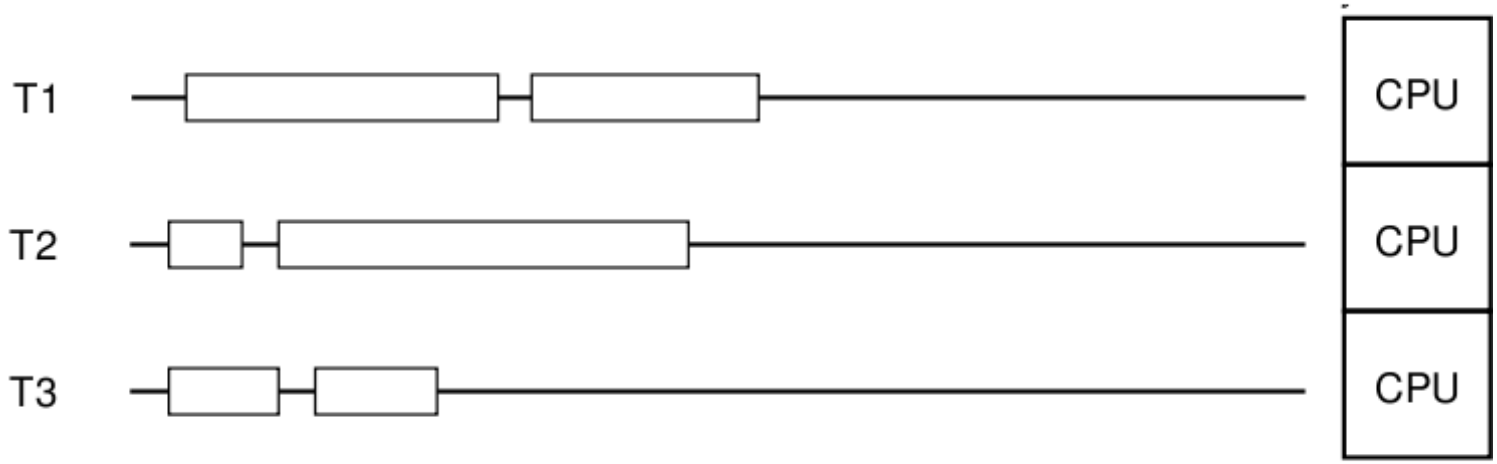
Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects.
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific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

Referensi: http://docs.oracle.com/cd/E19963-01/html/821-1601

Recall Thread Execution



Three Threads Running Concurrently on One CPU



Three Threads Running in Parallel on Three CPUs

Rutin Manajemen Thread

- *pthread_create (thread, attr, start_routine, arg)*
- *pthread_exit (status)*
- *pthread_cancel (thread)*
- *pthread_attr_init (attr)*
- *pthread_attr_destroy (attr)*

Bahasan

Membuat Thread Baru

```
#include <pthread.h>

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

- main() program comprises a single, default thread. All other threads **must be explicitly** created by the programmer
- Argument:
  - *thread*: An opaque, unique identifier for the new thread returned by the subroutine
  - *attr*: An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values
  - *start_routine*: A pointer to the C routine that the thread will execute once it is created
  - *arg*: A single argument that may be passed to start_routine. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed
- Return: 0 for success or an error number if anything goes wrong

Atribut Thread

- Can be set when creating a thread
- Or initialize thread attribute in main thread: *pthread_attr_init* and *pthread_attr_destroy*
- Attributes:
  - Detached or joinable state
  - Scheduling inheritance
  - Scheduling policy
  - Scheduling parameters
  - Scheduling contention scope
  - Stack size
  - Stack address
  - Stack guard (overflow) size

Terminasi Thread

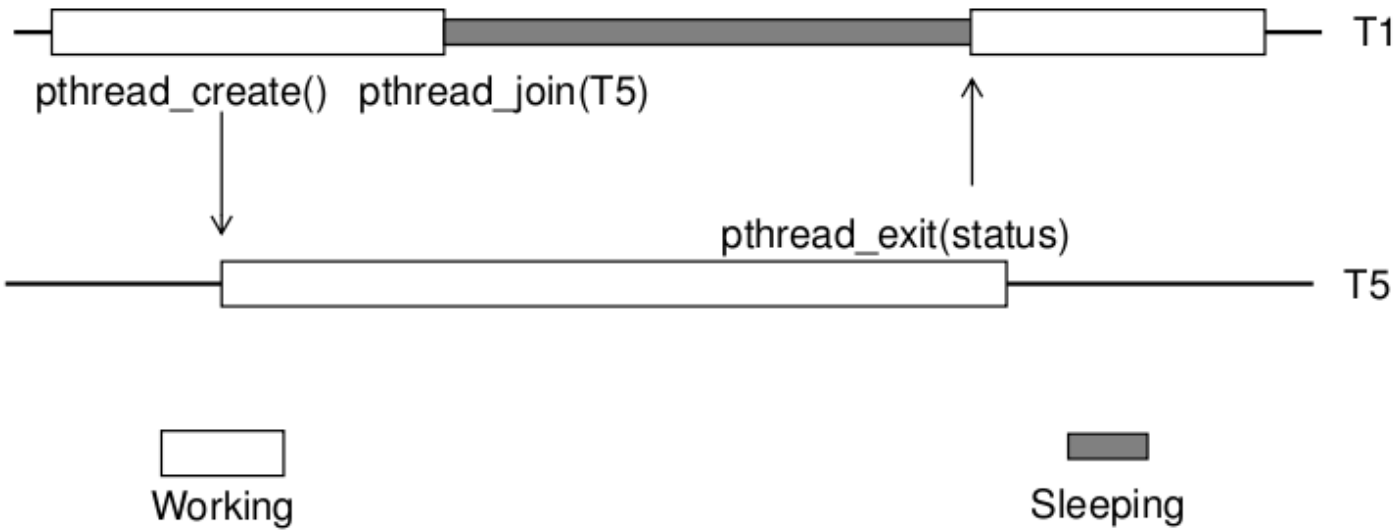
- A thread may be terminated:
  - The thread **returns** normally from its starting routine. It's work is done
  - The thread makes a call to the ***pthread_exit*** subroutine - whether its work is done or not
  - The thread is canceled by another thread via the ***pthread_cancel*** routine
  - The entire process is terminated due to making a call to either the exec() or exit()
  - If main() finishes first, without calling pthread_exit explicitly itself
- Optional termination status can be specified
  - Typically returned to threads joining the terminated thread
- Cleanup: the pthread_exit() does not close files
  - Any files opened inside the thread will remain open after the thread is terminated

Rutin pthread_exit

```
#include <pthread.h>

void *status;
pthread_exit(status); /* exit with status */
```

- Return: The calling thread terminates with its exit status set to the contents of status. Called by *pthread_join*

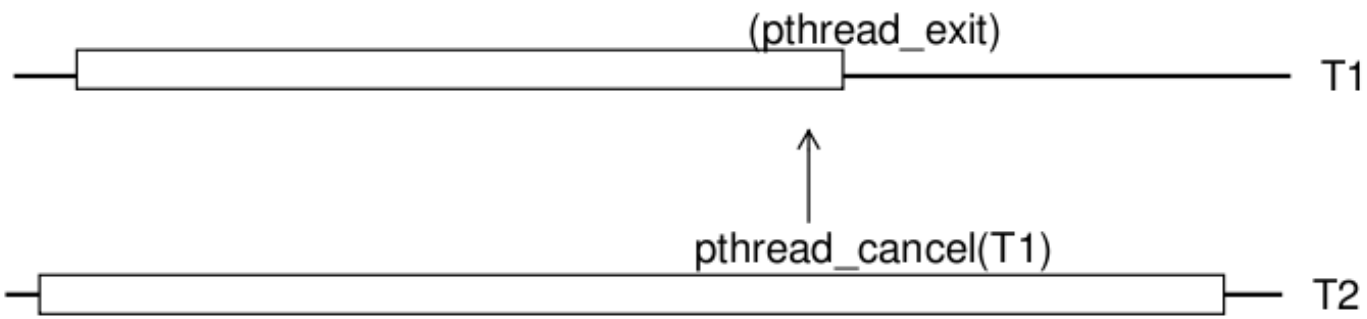


Rutin pthread_cancel

```
#include <pthread.h>

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

- There is no relationship between the threads
  - Maybe T2 created T1, maybe T3 created both of them, maybe something else



Kode: Pembuatan dan Terminasi Thread

- See code: simple_thread.c

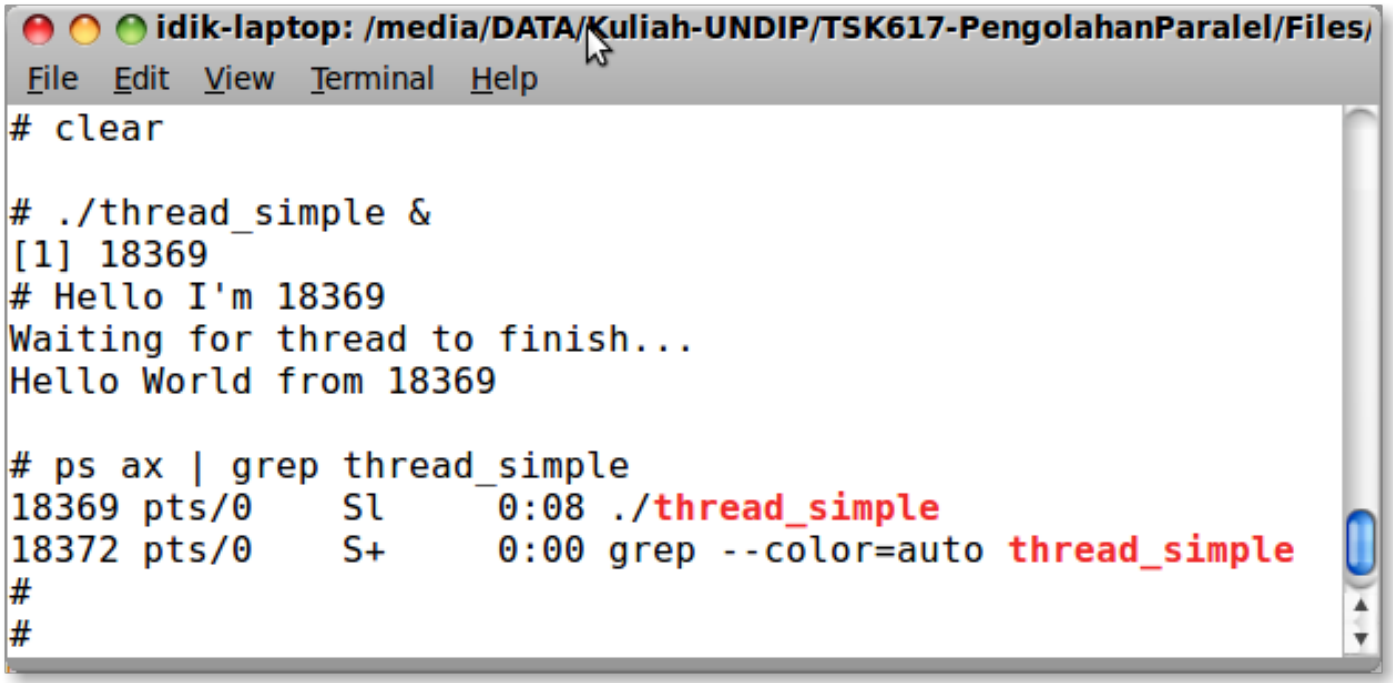
```
#include <pthread.h>

void *thread_function(void *arg);
int main() {
    pthread_t a_thread;
    void *thread_result;

    res = pthread_create(&a_thread, NULL, thread_function, NULL);
    ...
    res = pthread_join(a_thread, &thread_result);
printf("Thread joined, it returned %s\n", (char *)thread_result);
    ...
}

void *thread_function(void *arg) {
    printf("Hello World\n"); //Thread tasks
    pthread_exit("Thank you for the CPU time\n");
}
```

Eksekusi Kode



Bahasan

API PThread

Manajemen Thread

Pembuatan dan Terminasi Thread

Passing Argumen ke Thread

Joining dan Detaching Thread

Manajemen Stack

Rutin Lainnya

Variabel Mutex

Variabel Kondisional

Umpan Balik

Lisensi

Passing Argumen ke Thread

```
#include <pthread.h>

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

- The pthread_create() routine permits the programmer to pass one argument to the thread_start routine
  - To pass multiple arguments, create a structure which contains all of the arguments, and then passing a pointer to that structure in the pthread_create() routine
- All arguments must be passed by reference and cast to (void *).

Passing Argumen Tunggal

- See code: simple_thread_arg1.c

```
#include <pthread.h>

char message[] = "Hello World";
void *thread_function(void *arg);
int main() {
   pthread_t a_thread;
   void *thread_result;

   res = pthread_create(&a_thread, NULL, thread_function, (void
*)message);
   ...
   res = pthread_join(a_thread, &thread_result);
printf("Thread joined, it returned %s\n", (char *)thread_result);
   ...
}

void *thread_function(void *arg) {
   printf("Message:%s\n", (char *)arg); //Print Argument
   pthread_exit("Thank you for the CPU time\n");
}
```

Passing Beberapa Argumen

- See code: simple_thread_arg2.c

```
#include <pthread.h>

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

int main() {
    char messages[] = "Hello World";
    void *thread_result;
    thread_data_array.thread_id = t;
    thread_data_array.message = messages;
    ...
    res = pthread_create(&a_thread, NULL, thread_function,
&thread_data_array);
    ...
    res = pthread_join(a_thread, &thread_result);
printf("Thread joined, it returned %s\n", (char *)thread_result);
    ...
}

void *thread_function(void *arg) {
```

Bahasan

API PThread

Manajemen Thread

Pembuatan dan Terminasi Thread

Passing Argumen ke Thread

Joining dan Detaching Thread

Manajemen Stack

Rutin Lainnya

Variabel Mutex

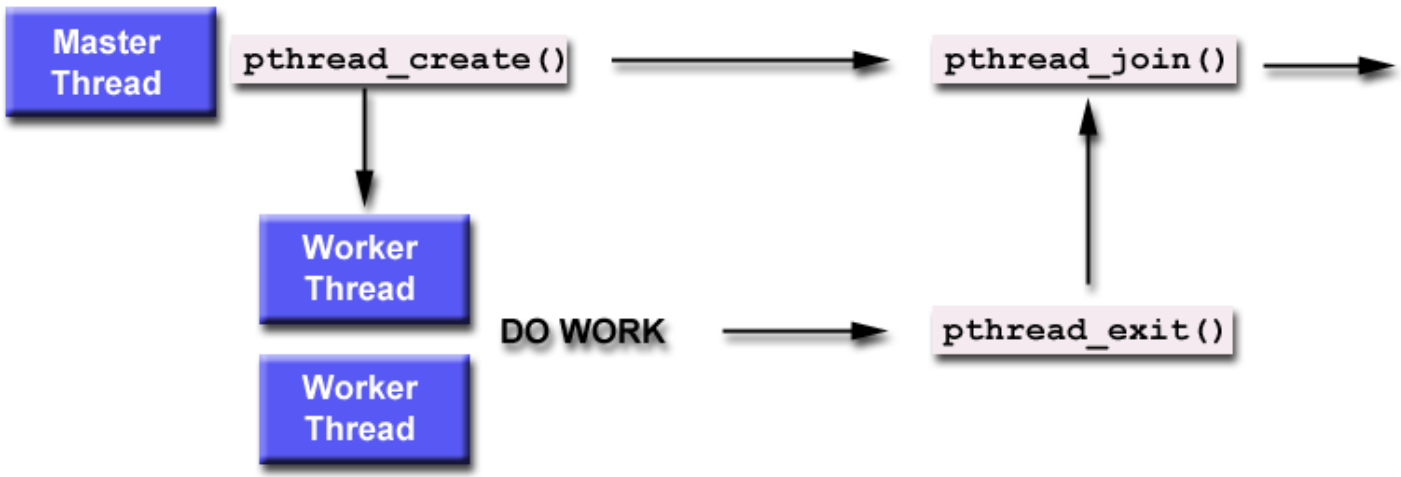
Variabel Kondisional

Umpan Balik

Lisensi

Joining

- Joining is one way to accomplish synchronization between threads
- The pthread_join() subroutine blocks the calling thread until the specified threadid thread terminates
- Two other synchronization methods, mutexes and condition variables
  - Will be discussed later



Setting Atribut Thread: Joinable atau Detached

- When a thread is created, one of its attributes defines whether it is joinable or detached
  - Only threads that are created as joinable can be joined
  - If a thread is created as detached, it can never be joined
- The final draft of the POSIX standard specifies that threads **should be** created as joinable.
- To explicitly create a thread as joinable or detached, the attr argument in the pthread_create() routine is used. The typical 4 step process is:
  1. Declare a pthread attribute variable of the *pthread_attr_t* data type
  2. Initialize the attribute variable with *pthread_attr_init()*
  3. Set the attribute detached status with *pthread_attr_setdetachstate()*
  4. When done, free library resources used by the attribute with *pthread_attr_destroy()*
- The pthread_detach() routine can be used to explicitly detach a thread even though it was created as joinable

Rutin pthread_join dan pthread_detach

```
#include <pthread.h>
pthread_t tid;
int ret;

void *status;
/* waiting to join thread "tid" with status */
ret = pthread_join(tid, &status);

/* waiting to join thread "tid" without status */
ret = pthread_join(tid, NULL);
```

- If a thread requires joining, consider explicitly creating it as joinable
  - This provides **portability as not all implementations** may create threads as joinable by default
- If you know in advance that a thread will never need to join with another thread, consider creating it in a detached state
  - Some system resources may be able to be freed

Joining dan Detaching Thread

- See code: thread_join.c (still need improvement)
- See code: thread_join_rev.c
  - Make sure all threads joined before accesing shared data

Bahasan

API PThread

Manajemen Thread

Pembuatan dan Terminasi Thread

Passing Argumen ke Thread

Joining dan Detaching Thread

Manajemen Stack

Rutin Lainnya

Variabel Mutex

Variabel Kondisional

Umpan Balik

Lisensi

Mencegah Masalah Stack

- The POSIX standard does not dictate the size of a thread's stack
  - This is implementation dependent and varies.
- Exceeding the default stack limit is often very easy to do, with the usual results: **program termination and/or corrupted data**
- Safe and portable programs do not depend upon the default stack limit
  - But instead, explicitly allocate enough stack for each thread by using the *pthread_attr_setstacksize* routine.
- The *pthread_attr_getstackaddr* and *pthread_attr_setstackaddr* routines can be used by applications in an environment where the stack for a thread must be placed in some particular region of memory

See: http://docs.oracle.com/cd/E19963-01/html/821-1601/attrib-27410.html#attrib-78533

Rutin pthread_attr_getstacksize

```
#include <pthread.h>
pthread_attr_t tattr;
size_t size;
int ret;

/* getting the stack size */
ret = pthread_attr_getstacksize(&tattr, &size);
```

Rutin pthread_attr_setstacksize

```
#include <pthread.h>
#include <limits.h>
pthread_attr_t tattr;
pthread_t tid;
int ret;
size_t size = PTHREAD_STACK_MIN + 0x4000;

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);

/* setting the size of the stack also */
ret = pthread_attr_setstacksize(&tattr, size);

/* only size specified in tattr*/
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

Ukuran Stack Default

- Default thread stack size varies greatly
  - may depend upon the number of threads per node

Node Architecture	#CPUs	Memory (GB)	Default Size (bytes)
AMD Xeon 5660	12	24	2,097,152
AMD Opteron	8	16	2,097,152
Intel IA64	4	8	33,554,432
Intel IA32	2	4	2,097,152
IBM Power5	8	32	196,608
IBM Power4	8	16	196,608
IBM Power3	16	16	98,304

Mengeset Ukuran Stack

- See code: thread_stack.c
  - How to query and set a thread's stack size

Bahasan

API PThread

Manajemen Thread

Pembuatan dan Terminasi Thread

Passing Argumen ke Thread

Joining dan Detaching Thread

Manajemen Stack

Rutin Lainnya

Variabel Mutex

Variabel Kondisional

Umpan Balik

Lisensi

Rutin pthread_self, pthread_equal

- Getting the Thread Identifier

```
#include <pthread.h>

pthread_t tid;
tid = pthread_self();
```

  - Returns the unique, system assigned thread ID of the calling thread

- Comparing Thread IDs

```
#include <pthread.h>

pthread_t tid1, tid2;
int ret;
ret = pthread_equal(tid1, tid2);
```

  - Returns a nonzero value when tid1 and tid2 are equal, otherwise, 0 is returned

- Because thread IDs are opaque objects:
  - the C language equivalence operator == should not be used to compare two thread IDs against each other, or to compare a single thread ID against another value

Rutin pthread_once

```
#include <pthread.h>

pthread_once_t once_control = PTHREAD_ONCE_INIT;
int ret;
ret = pthread_once(&once_control, init_routine);
```

- Call to pthread_once in a threaded process executes the init_routine exactly once in a process
  - The first **call to this routine by any thread** in the process executes the given init_routine, without parameters
  - Any subsequent call will have no effect.
- The init_routine routine is typically an initialization routine.
- The once_control parameter is a synchronization control structure that requires initialization prior to calling pthread_once.
  - For example: pthread_once_t once_control = PTHREAD_ONCE_INIT;

Bahasan

Mutex: Mutual Exclusion

- One of the primary means of implementing **thread synchronization** and for **protecting shared data** when multiple writes occur
  - A mutex variable acts like a "lock"
  - **Only one thread** can lock (or own) a mutex variable at any given time
    - Thus, even if several threads try to lock a mutex only one thread will be successful
    - No other thread can own that mutex until the owning thread unlocks that mutex
    - Threads must "take turns" accessing protected data
  - Can be used to prevent “race” conditions
    - a thread owning a mutex often do updating of global variables
    - The shared variables being updated belong to a "**critical section**"

See: http://docs.oracle.com/cd/E19963-01/html/821-1601/sync-83092.html

Kondisi Race

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

- A mutex should be used to lock the “Balance” while a thread is using this shared data resource.

Typical Mutex Sequence

1. Create and initialize a mutex variable
2. Several threads attempt to lock the mutex
   - When several threads compete for a mutex, the losers block at that call
   - An unblocking call is available with "trylock" instead of the "lock" call.
3. Only one succeeds and that thread owns the mutex
4. The owner thread performs some set of actions
5. The owner unlocks the mutex
6. Another thread acquires the mutex and repeats the process
7. Finally the mutex is destroyed

Bahasan

Inisialisasi dan Menghapus Mutex

```
#include <pthread.h>

pthread_mutex_t mp = PTHREAD_MUTEX_INITIALIZER;
pthread_mutexattr_t mattr;
int ret;

/* initialize a mutex to its default value */
ret = pthread_mutex_init(&mp, NULL);

/* initialize a mutex */
ret = pthread_mutex_init(&mp, &mattr);
```

```
#include <pthread.h>

pthread_mutex_t mp;
int ret;
ret = pthread_mutex_destroy(&mp); /* mutex is destroyed */
```

Setting Atribut Mutex

- The attr object is used to establish properties for the mutex object

```
#include <pthread.h>

pthread_mutexattr_t mattr;
int ret;
/* initialize an attribute to default value */
ret = pthread_mutexattr_init(&mattr);
```

```
#include <pthread.h>

pthread_mutexattr_t mattr;
int ret;
/* destroy an attribute */
ret = pthread_mutexattr_destroy(&mattr);
```

Pthreads mutex attributes

1. Protocol: Specifies the protocol used to prevent priority inversions for a mutex. Default value: PTHREAD_PRIO_NONE
2. Prioceiling: Specifies the priority ceiling of a mutex. Default value: -
3. Process-shared: Specifies the process sharing of a mutex. Default value: PTHREAD_PROCESS_PRIVATE
   - Only those threads created by the same process can operate on the mutex

- Note that not all implementations may provide the three optional mutex attributes.

Bahasan

Locking Mutex

```
#include <pthread.h>

pthread_mutex_t mutex;
int ret;
ret = pthread_ mutex_lock(&mp); /* acquire the mutex */
```

- When pthread_mutex_lock() returns, the mutex is locked
  - The calling thread is the owner
- If the mutex is already locked and owned by another thread, the calling thread blocks until the mutex becomes available

Mutex Type

- PTHREAD_MUTEX_NORMAL , deadlock detection is not provided
  - Attempting to relock the mutex causes deadlock
  - If a thread attempts to unlock a mutex not locked by the thread or a mutex that is unlocked, undefined behavior results.
- PTHREAD_MUTEX_ERRORCHECK , then error checking is provided
  - If a thread attempts to relock a mutex that the thread has already locked, an error is returned
  - If a thread attempts to unlock a mutex not locked by the thread or a mutex that is unlocked, an error is returned.
- PTHREAD_MUTEX_RECURSIVE , then the mutex maintains the concept of a lock count
  - When a thread successfully acquires a mutex for the first time, the lock count is set to 1
  - Every time a thread relocks this mutex, the lock count is incremented by 1
  - Every time the thread unlocks the mutex, the lock count is decremented by 1
  - When the lock count reaches 0, the mutex becomes available for other threads to acquire

Non-blocking Locking

```
#include <pthread.h>

pthread_mutex_t mutex;
int ret;
ret = pthread_mutex_trylock(&mutex); /* try to lock the mutex */
```

- Will attempt to lock a mutex
  - if the mutex is already locked, the routine will return immediately with a "busy" error code
  - may be useful in preventing deadlock conditions

Unlocking Mutex

```
#include <pthread.h>

pthread_mutex_t mutex;
int ret;
ret = pthread_mutex_unlock(&mutex); /* release the mutex */
```

- releases the mutex object referenced by mutex

Kode: Penggunaan Mutex

- See code: thread_mutex.c
  - Computes the dot product of two vectors (skalar product)
  - Implementing data parallelisme
- Compare to dotprod_serial.c

Bahasan

Variabel Kondisional

- Condition variables provide another way for threads to synchronize
  - While mutexes implement synchronization by controlling thread access to data
  - Condition variables allow threads to synchronize based upon the actual value of data
- Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met
  - This can be very resource consuming since the thread would be continuously busy in this activity
  - A condition variable is a way to achieve the same goal without polling
- A condition variable is always used in conjunction with a mutex lock

Bahasan

Operasi di Thread Main

Main Thread

- Declare and initialize global data/variables which require synchronization (such as "count")
- Declare and initialize a condition variable object
- Declare and initialize an associated mutex
- Create threads A and B to do work

Operasi di Thread A dan B

Thread A

- Do work up to the point where a certain condition must occur (such as "count" must reach a specified value)
- Lock associated mutex and check value of a global variable
- Call *pthread_cond_wait()* to perform a blocking wait for signal from Thread-B. Note: a call to *pthread_cond_wait()* automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B
- When signalled, wake up. Mutex is automatically and atomically locked
- Explicitly unlock mutex
- Continue

Thread B

- Do work
- Lock associated mutex
- Change the value of the global variable that Thread-A is waiting upon.
- Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A using *pthread_cond_signal()*
- Unlock mutex.
- Continue

Main Thread

- Join / Continue

Bahasan

Rutin pthread_mutex_init()

- Initialize the condition variable

```
#include <pthread.h>

pthread_cond_t cv;
pthread_condattr_t cattr;
int ret;

/* initialize a condition variable to its default value */
ret = pthread_cond_init(&cv, NULL);

/* initialize a condition variable */
ret = pthread_cond_init(&cv, &cattr);
```

Rutin pthread_cond_wait()

```
#include <pthread.h>

pthread_cond_t cv;
pthread_mutex_t mp;
int ret;

/* wait on condition variable */
ret = pthread_cond_wait(&cv, &mp);
```

- pthread_cond_wait() blocks the calling thread until the specified condition is signalled
- This routine should be called while mutex is locked, and it will automatically release the mutex while it waits
- After signal is received and thread is awakened, mutex will be automatically locked for use by the thread
  - The programmer is then responsible for unlocking mutex when the thread is finished with it

Rutin pthread_cond_signal()

```
#include <pthread.h>

pthread_cond_t cv;
int ret;

/* one condition variable is signaled */
ret = pthread_cond_signal(&cv);
```

- Used to signal (or wake up) another thread which is waiting on the condition variable
  - It should be called after mutex is locked, and must unlock mutex in order for *pthread_cond_wait()* routine to complete

Kode: Variabel Kondisional

- See code: thread_varcond.c
  - The main thread creates three threads
    - Two of those threads increment a "count" variable, while the third thread watches the value of "count"
    - When "count" reaches a predefined limit, the waiting thread is signaled by one of the incrementing threads
    - The waiting thread "awakens" and then modifies count
    - The program continues until the incrementing threads reach TCOUNT
  - The main program prints the final value of count.

Umpan Balik

- Yang telah kita pelajari hari ini:
  - tbd
- Yang akan kita pelajari di pertemuan berikutnya adalah <tbd>
  - Pelajari: tbd

Lisensi

Creative Common Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0)

- Anda bebas:
  - untuk **Membagikan** — untuk menyalin, mendistribusikan, dan menyebarkan karya, dan
  - untuk **Remix** — untuk mengadaptasikan karya
- Di bawah persyaratan berikut:
  - **Atribusi** — Anda harus memberikan atribusi karya sesuai dengan cara-cara yang diminta oleh pembuat karya tersebut atau pihak yang mengeluarkan lisensi.
  - **Pembagian Serupa** — Jika Anda mengubah, menambah, atau membuat karya lain menggunakan karya ini, Anda hanya boleh menyebarkan karya tersebut hanya dengan lisensi yang sama, serupa, atau kompatibel.
- Lihat: Creative Commons Attribution-ShareAlike 3.0 Unported License