Network Programming: Ch. 26: Threads

Li-Hsing Yen
NYCU

Ver. 1.0.0

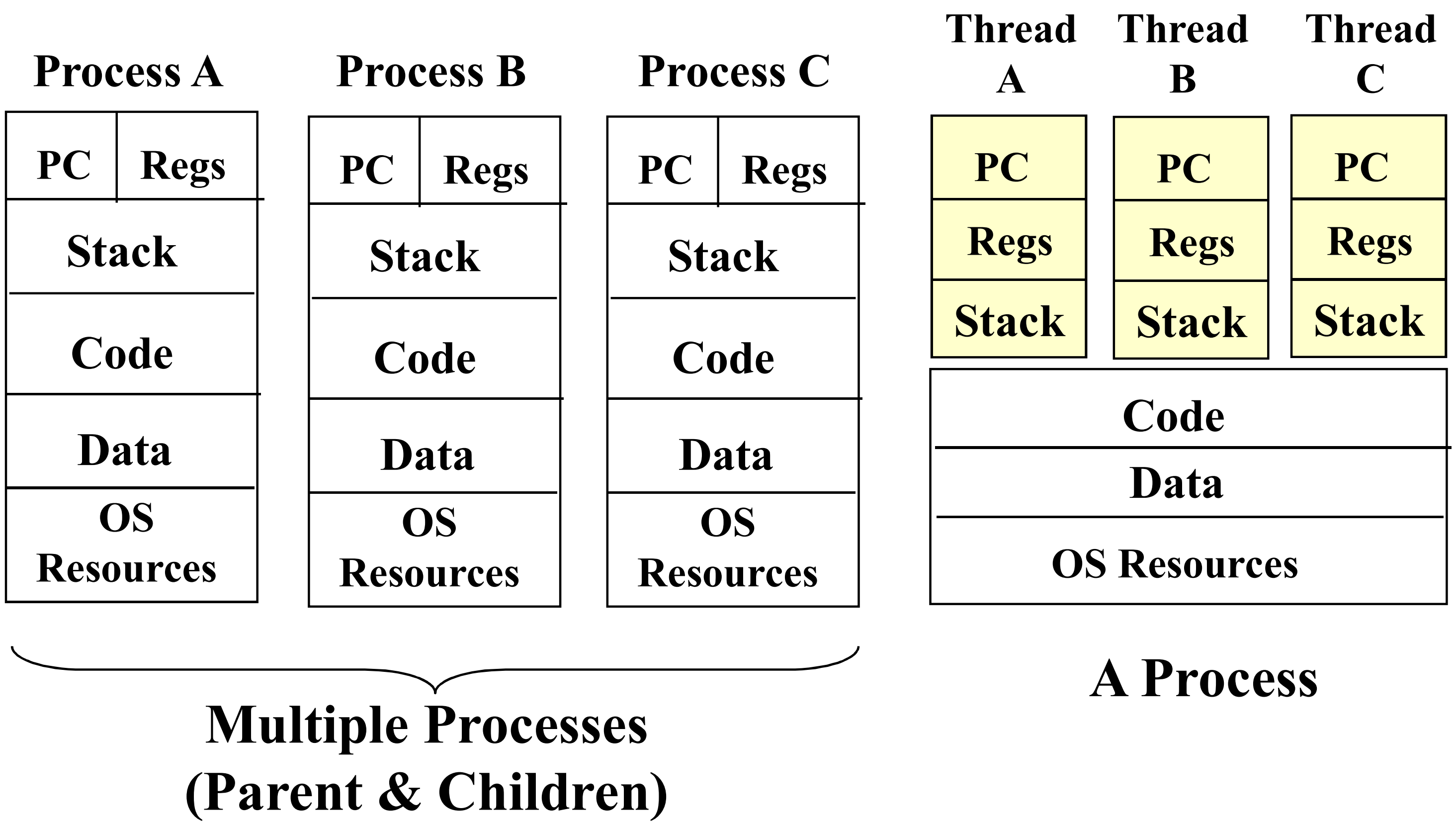
Threads

- Introduction
- Basic Thread Functions: Creation and Termination
- **`str_cli`** Function Using Threads
- TCP Echo Server Using Threads
- Thread-Specific Data
- Web-Client and Simultaneous Connections

Problems With `fork`

- `fork` is expensive
  - Memory copy, descriptor duplication, etc.
- IPC is required to pass information between parent and child
  - Passing info from parent to child before `fork` is easy
  - Returning info from child to the parent is not

Multiple Processes vs. Threads

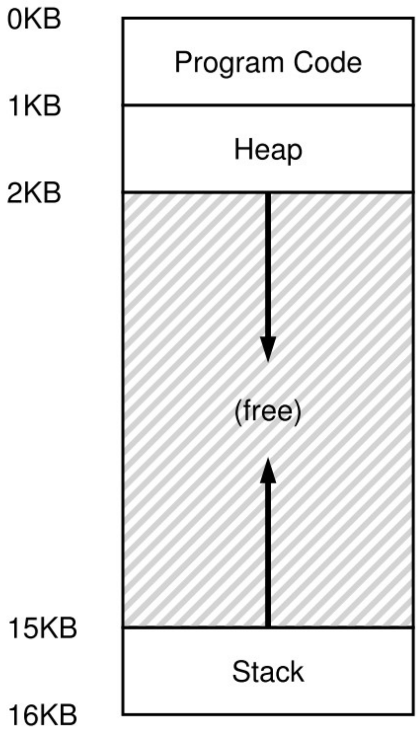


Threads

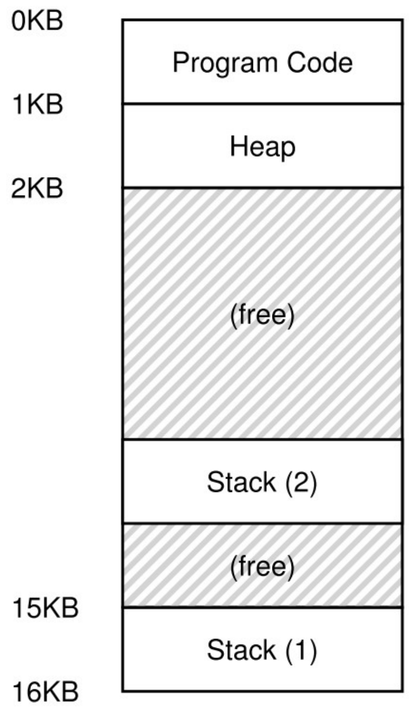
- All threads within a process share the same global memory, code, open files, user ID
- Each thread has its own
  - Thread ID
  - Set of registers (including PC and stack counter)
  - **`errno`**
  - Signal mask
  - priority

Address Spaces

- A single-thread process



- A Process with Two Threads



Why Threads?

- Parallelism
  - performance gain in multicore and multiprocessor system
- to avoid blocking program progress due to slow I/O
  - overlap of I/O with other activities within a single program
  - help in structuring clients and servers
- implementing a large application
  - a way of modulation

Advantage 1: possibility to exploit parallelism

- Possible when executing the program on a multiprocessor or multicore system
- each thread can be assigned to a different CPU or core
- shared data are stored in shared main memory

Advantage 2: Non-blocking

- A single-threaded process as a whole is blocked whenever a blocking system call (e.g., I/O operation) is executed
- On the other hand, a multi-threaded process will not be entirely blocked simply because one threaded is executing a system call (e.g., waiting for user input)

Advantage 3: for implementing a large application

- Two options for such an implementation
  - As a collection of cooperating programs (each to be executed by a separate process)
  - As a program with multiple threads

Downside of Threads

- OS does not directly provide protections among threads
  - threads share an address space so it's easier to share data
  - needs additional intellectual efforts
- Compared with multiple processes
  - processes are a more sound choice for logically separate tasks

Thread Creation

- When a program is started, a single thread (called initial thread or main thread) is created
- Additional threads are created by

Return 0 if OK, positive *Exxx* value on error

```
#include <pthread.h>
int Pthread_create (pthread_t *tid, const pthread_attr *attr,
                    void *(*func)(void *), void *arg);
```

tid: pointer to ID of the created thread

attr: pointer to the attribute of the thread; NULL to take the default

func: pointer to the function for the created thread to execute

arg: pointer to the data passed to *func*

pthread_self Function

- A thread fetches the thread ID for itself

```
#include <pthread.h>
pthread_t pthread_slef (void);
                                  Return: thread ID of calling thread
```

Process		Thread
getpid	⟷	pthread_self

Simple Thread Creation Codes

```
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

void *mythread(void *arg) {
  printf("%s\n", (char *) arg);
  return NULL;
}
```

```
int main(int argc, char *argv[]) {
  pthread_t p1, p2;
  int rc;
  printf("main: begin\n");
  Pthread_create(&p1, NULL, mythread, "A");
  Pthread_create(&p2, NULL, mythread, "B");
  // join waits for the threads to finish
  Pthread_join(p1, NULL);
  Pthread_join(p2, NULL);
  printf("main: end\n");
  return 0;
}
```

What Really Happens

The new thread

```
void *mythread(void *arg) {
  printf("%s\n", (char *) arg);
  return NULL;
}
```

The main thread

```
main(int argc, char *argv[]) {
  pthread_t p1, p2;
  int rc;
  printf("main: begin\n");
  Pthread_create(&p1, NULL, mythread, "A");
…
```

The main thread continues without waiting

What if we have more than one datum to pass?

To Get The ID of the Created Tread

- Function *func* is called with a single pointer argument *arg*
  - If we need multiple arguments to the function, pack them into a structure and pass the address
- function *func* returns a generic **`(void *)`** pointer
- function *func* terminates either explicitly (by calling **`pthread_exit`**) or implicitly (letting the function return)

The Function to Be Executed

- Function *func* is called with a single pointer argument *arg*
  - If we need multiple arguments to the function, pack them into a structure and pass the address
- function *func* returns a generic **`(void *)`** pointer
- function *func* terminates either explicitly (by calling **`pthread_exit`**) or implicitly (letting the function return)

Thread: Joinable or Detached

- A thread is either *joinable* (by default) or *detached*
  - When a joinable thread terminates, its thread ID and exit status are retained until another thread calls **`pthread_join`**
  - When a detached thread terminates, all its resources are released (we cannot wait it)
- If one thread needs to know when another thread terminates, it is best to leave the thread *joinable*

pthread_detach Function

- Change the specified thread to detached

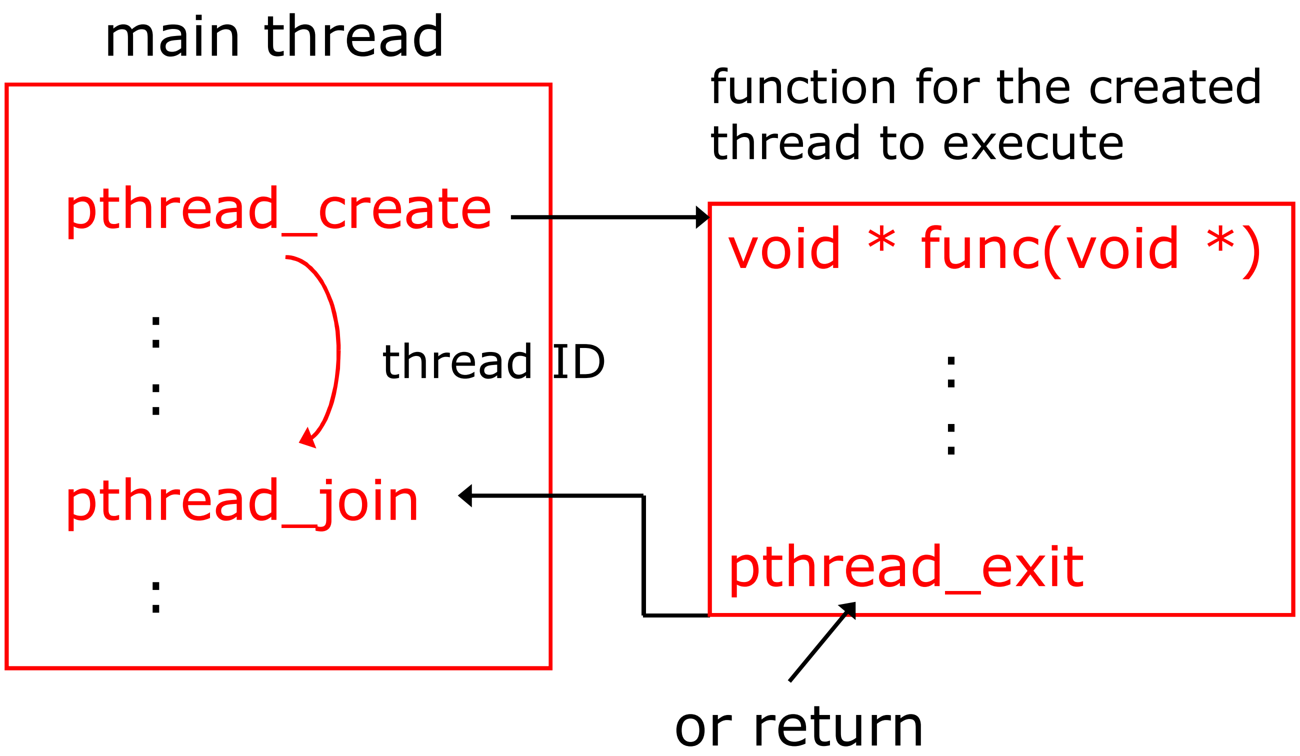
```
#include <pthread.h>
int pthread_detach (pthread_t tid);
```

Return 0 if ok, positive *Exxx* value on error

This function is commonly called by the thread that wants to detach itself, as in

```
pthread_detach(pthread_self());
```

Joinable Thread



`pthread_join` Function

- Wait for a given thread to terminate

Return 0 if OK, positive *Exxx* value on error

#include <pthread.h>

int pthread_join (pthread_t *tid*, void *******status*);

tid: thread ID

status: if non-null, the return value from the thread (which is a void pointer) is stored in the location pointed to by *status*

Process		Thread
fork	⟺	pthread_create
waitpid	⟺	pthread_join

pthread_self Function

- A thread fetches the thread ID for itself

```
#include <pthread.h>
pthread_t pthread_slef (void);
```

Return: thread ID of calling thread

Process		Thread
getpid	⟷	pthread_self

pthread_exit Function

- One way for a thread to terminate

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

Does **not** return to caller

If the thread is not detached, its thread ID and exit status are retained for a latter **pthread_join** by some other thread

Pointer *status* must not point to an object that is local to the calling thread (the object no longer exists after **pthread_exit**)

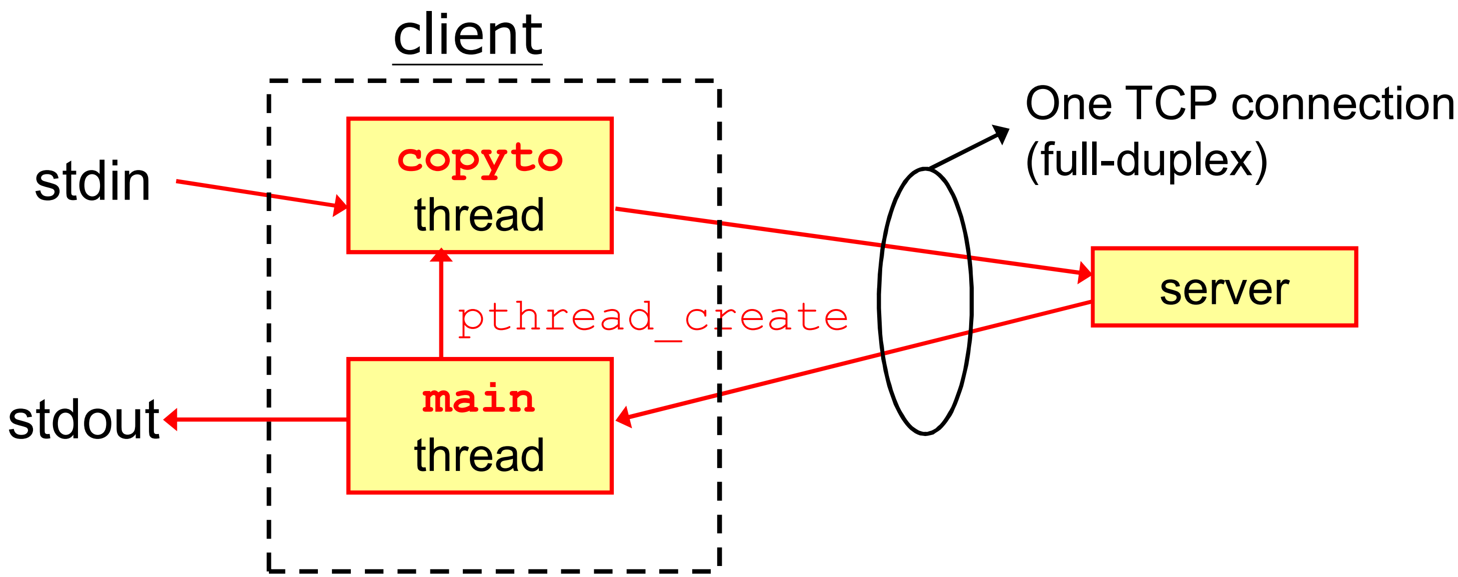
Getting Thread Exit Status

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

int main() {
  pthread_t t;
  void *ret;
  int input = 5;

  pthread_create(&t, NULL, child, (void*) &input);
  …
  pthread_join(t, &ret);
  int *resultp = (int *) ret;
  printf("%d\n", *(resultp));
}
```

```
void *child(void *arg) {
   int *inputp = (int *) arg;
   int *resultp = malloc(sizeof(int) * 1);
   *(resultp) = *(inputp) + 10;
   pthread_exit((void *) resultp);
}
```

str_cli Function Using Threads

- 將Fig. 16.10兩個processes的版本改為兩個threads的版本



main thread

```
void
str_cli(FILE *fp_arg, int sockfd_arg)
{
	char	recvline[MAXLINE];
	pthread_t	tid;

	sockfd = sockfd_arg;	/* copy arguments */
	fp = fp_arg;

	Pthread_create(&tid, NULL, copyto, NULL);

	while (Readline(sockfd, recvline, MAXLINE) > 0)
		Fputs(recvline, stdout);
}
```

另個thread要執行的function

copyto thread

```
void *
copyto(void *arg)
{
	char	sendline[MAXLINE];

	while (Fgets(sendline, MAXLINE, fp) != NULL)
		Writen(sockfd, sendline, strlen(sendline));

	Shutdown(sockfd, SHUT_WR);	/* EOF */

	return(NULL);
		/* return (i.e., thread terminates) */
}
```

These two threads do not communicate

Thread Terminations

- When a process terminates, all threads in the process are also terminated
- When `str_cli` returns, the main function terminates by calling `exit`
  - So all threads are terminated
  - Normally, `copyto` will have already terminated
  - If not, `copyto` will be terminated now

TCP Echo Server Using Threads

- One thread per client (instead of one child process per client)
  - Call `pthread_create` instead of `fork`
  - Creating a new thread does not affect the reference counts for open descriptors

  ⇒ Main thread **must not** close the connected socket

  ⇒ Created thread **must** close the connected socket

main thread

```
int main(int argc, char **argv)
{
	int                 listenfd, connfd;
	pthread_t           tid;
	socklen_t           addrlen, len;
	struct sockaddr     *cliaddr;

	…
	cliaddr = Malloc(addrlen);

	for ( ; ; ) {
		len = addrlen;
		connfd = Accept(listenfd, cliaddr, &len);
		Pthread_create(&tid, NULL, &doit, (void *) connfd);
	}
}
```

cast connfd to a void pointer

function for the thread to run

doit thread

```
static void *
doit(void *arg)
{
        Pthread_detach(pthread_self());
        str_echo((int) arg);
        Close((int) arg);
        return(NULL);
}
```

connfd的值被當作void pointer傳進來成為arg的值

Call str_echo; Cast arg to int

created thread must close the connected socket

Potential Problem in This Version

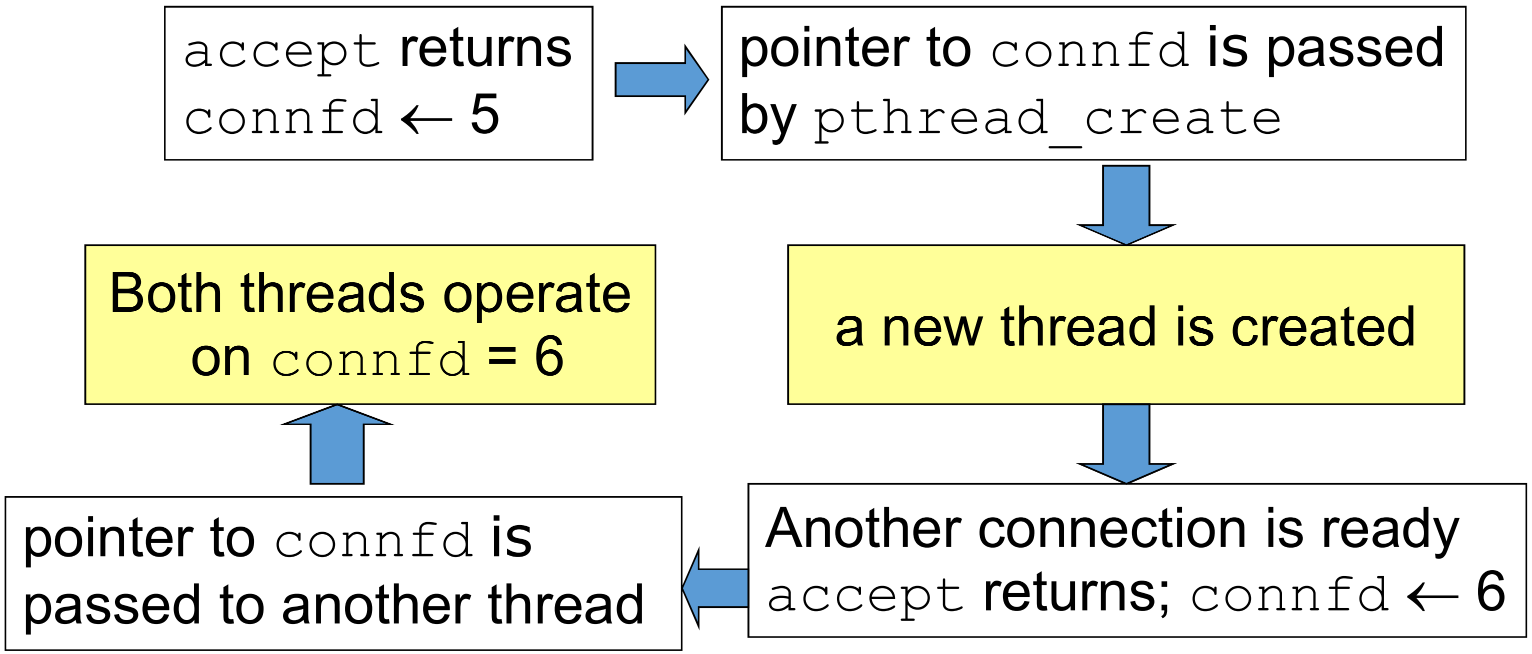
- casting an integer (`connfd`) to a void pointer may not work on all systems
- How about passing the address of `connfd`?

```
int main(int argc, char **argv)
{
        int  listenfd, connfd;
        …
        for ( ; ; ) {
                len = addrlen;
                connfd = Accept(listenfd, cliaddr, &len);
                Pthread_create(&tid, NULL, &doit, &connfd);
        }
}
```

Not good

Problem Caused by Shared Variables

- threads in the same process share variables



A Better Solution

- give each thread its own copy of `connfd`

```
int main(int argc, char **argv)
{
        int listenfd, *iptr;
        …
        for ( ; ; ) {
                len = addrlen;
                iptr = Malloc(sizeof(int));
                *iptr = Accept(listenfd, cliaddr, &len);
                Pthread_create(&tid, NULL, &doit, iptr);
        }
}
```

Another Part of the Solution

- the storage for `connfd` is freed

```
static void *
doit(void *arg)
{
        int connfd;
        connfd = *((int *) arg);
        free(arg);
        Pthread_detach(pthread_self());
        str_echo(connfd);
        Close(connfd);
        return(NULL);
}
```

Nonre-entrant Functions

- Historically, `malloc` and `free` are nonre-entrant functions
  - calling either function from a thread while another thread is in the middle of `malloc`/`free` is a disaster
  - Because these two functions manipulate static data structures
- These two functions (as well as many others; including all ANSI C functions) must be *thread-safe* (re-entrant)

Re-entrant (Thread-Safe) Function

pthread_create(.., fun1,..)
:
:
pthread_create(.., fun2,..)
:
:
pthread_create(.., fun3,..)

當每個function只被一個thread執行時，沒有re-entrant的問題

loop
:
pthread_create(.., fun1,..)
:
or :
pthread_create(.., fun3,..)
間接呼叫fun1

當某個function被一個以上的thread執行時，必須是re-entrant

Sharing Data Among Threads

Communication Between Threads

- By sharing variables
  - sockfd in the TCP Echo Client example is a shared variable
- Or by passing variables from one thread to another
  - connfd in the TCP Echo Server example is a passing variable
- Thread switching can sometimes be done entirely in user space (no context switching between user-level threads)
- Much faster

Static vs. Stack-Dynamic Data

```
#include <stdio.h>

int sum;

int fun1(int a, float b)
{
     static int cnt;
     int i, j;
     …
}
```

static

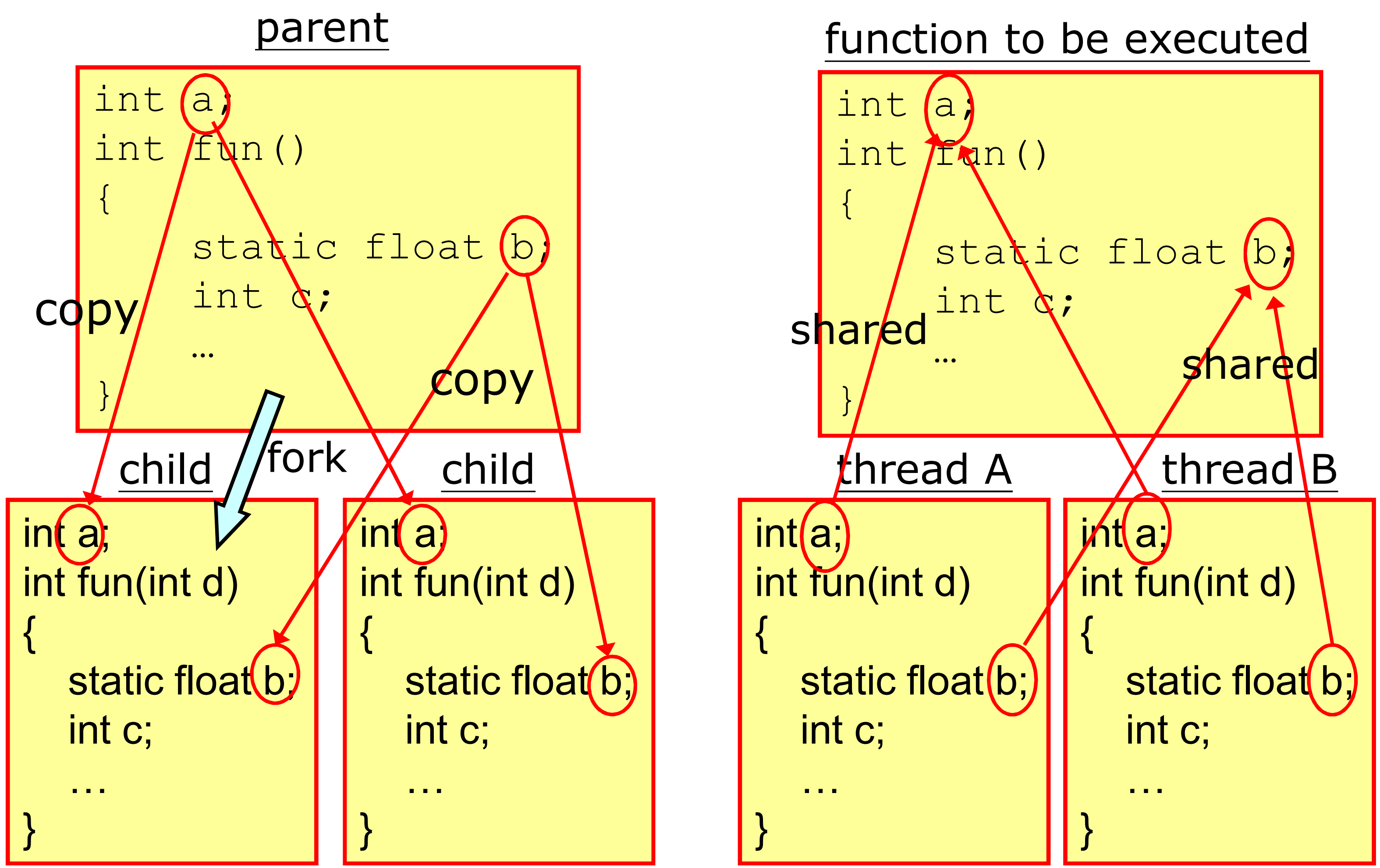
stack-dynamic

static

stack-dynamic

- Static data
  - variables bound to memory cells before execution begins and remains bound to the same memory cell throughout execution
- Stack-dynamic
  - storage bindings are created for variables when their declaration statements are elaborated.
  - cannot be history sensitive

Static Data in Sub-processes & Threads



Static Data

儲存位址在程式執行過程中是不變的

- Static data is a common problem when making a function thread-safe
  - For example, functions that keep state in a private buffer---multiple threads cannot use the buffer to hold different things at the same time

```
int fun()
{
    static char buf[10];
    static int a;
    …
}
```

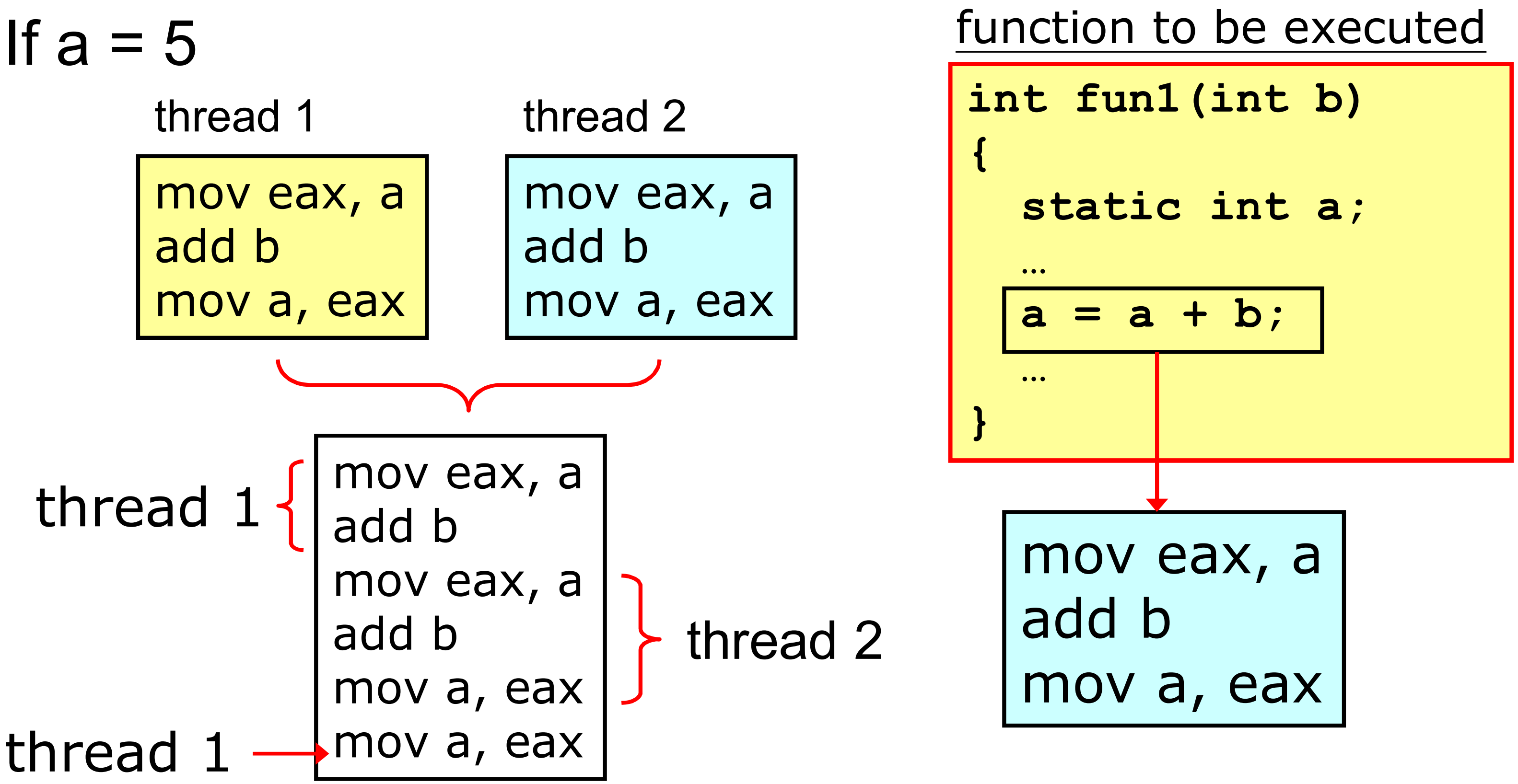
只有一個唯一的儲存空間。當某個thread將其內容變更時，其它thread的執行會受影響

Needs for Different Types of Data

- If data are to be shared among threads
  - use static data
  - need protection to avoid concurrent accesses (for data consistency)
- If data are specific to threads and history insensitive
  - use stack-dynamic data
- If data are specific to threads and history sensitive
  - Use heap-dynamic data (e.g., calling `malloc()` and `free()`) and the scheme provided by threads (covered later)

Race Condition Between Two Threads

- If a = 5



Data Inconsistency Errors

- Occur when multiple threads update a shared static variable simultaneously
- Occur rarely
- Hard to duplicate
- The same code works on one system but not on another
  - The hardware instruction might or might not be atomic (i.e., its execution is uninterruptable)

Critical Section and Mutual Execution

- To avoid data inconsistency, we need critical session
- A critical section is a piece of code that accesses a shared variable (or more generally, a shared resource)
- A critical section must not be concurrently executed by more than one thread.
- Mutual exclusion
  - A property guarantees that if one thread is executing within the critical section, the others will be prevented from doing so.

How to achieve mutual execution?

- Powerful atomic instruction

```
mov eax, a
add 1
mov a, eax
```

→

```
memory-add a, 1
```

executes atomically

either not run at all or
run to completion

When we have to update a general structure

- e.g., a concurrent B-tree
- Atomic instructions are not enough
- We only need a few useful instructions to build a set of synchronization primitives (such as locks and semaphores)

Covered later

Calling Non-reentrant Functions

Re-entrant Function: An Example

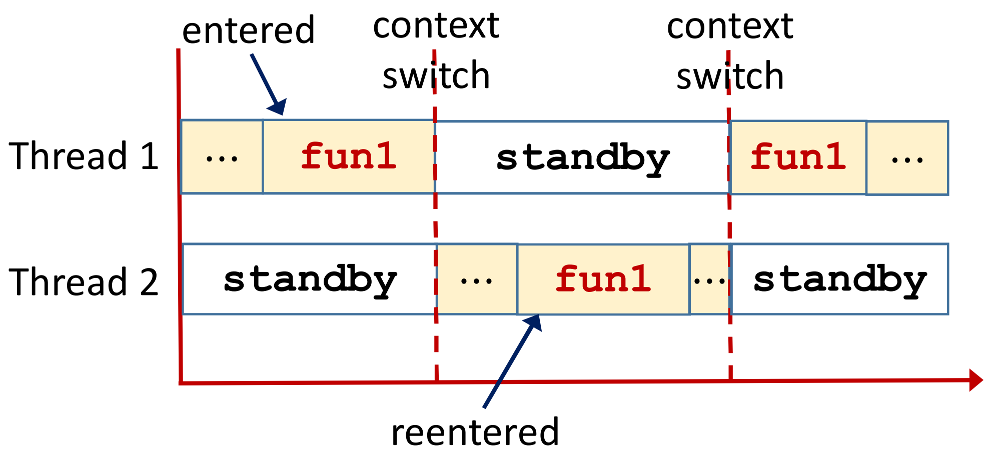
Thread 1

```
void *mythreadA(void *arg)
{
   …
   fun1(2);
   …
}
```

Thread 2

```
void *mythreadB(void *arg)
{
   …
   fun1(1);
   …
}
```

```
int fun1(int b)
{
   static int a;
   …
   a = a + b;
   …
}
```



Static Data in Re-entrant Function

- If a = 5

thread 1

```
mov eax, a
add b
mov a, eax
```

thread 2

```
mov eax, a
add b
mov a, eax
```

thread 1

```
mov eax, a
add b
```

thread 2

```
mov eax, a
add b
mov a, eax
```

thread 1

```
mov a, eax
```

function to be executed

```
int fun1(int b)
{
  static int a;
  …
  a = a + b;
  …
}
```

```
mov eax, a
add b
mov a, eax
```

Problem With Re-entrant Function

- manipulating static data structures in a re-entrant function could be a disaster
- This happens when a thread calls the function while another thread is in the middle of it
- The function writer could avoid the potential problem by
  - Not using static data in a re-entrant function
  - Let the re-entrant function use synchronization primitives to maintain the consistency of static data
- What if the re-entrant function is a library function?

Consider An Example

Thread 1

```
void *mythreadA(void *arg)
{
  char *str;
  …
  str = malloc(sizeof(char)*200);
  …
  free(str);
}
```

Thread 2

```
void *mythreadB(void *arg)
{
  int *vec;
  …
  vec = malloc(sizeof(int)*100);
  …
  free(vec);
}
```

Calling Non-reentrant Functions

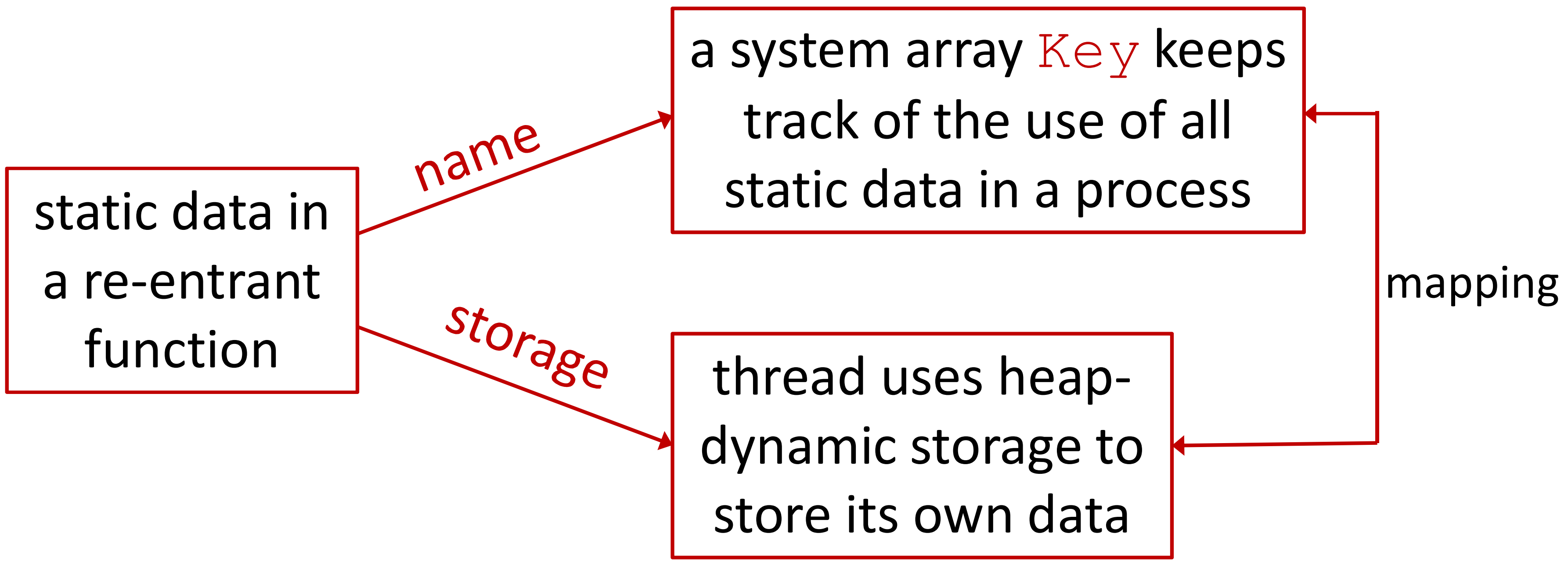
- Historically, `malloc` and `free` are non-reentrant functions
  - Because these two functions manipulate static data structures
  - calling either function from a thread while another thread is in the middle of `malloc`/`free` is a disaster
- These two functions (as well as many others; including all ANSI C functions) must be *thread-safe* (re-entrant)

Writing Your Own Thread-Safe Function

- 重點：共用同一個function但每個thread要有各自存取的資料
- Three possible ways
  - Avoid any static variables (i.e., using only local variables): not always viable (效能可能變差)
  - The caller packs all the arguments (and stores static variable) into a structure
  - Use thread-specific data: nontrivial, works only on systems with threads support

Providing Thread-Specific Data

- 要達成thread-safe的function需避免所有threads共用static data

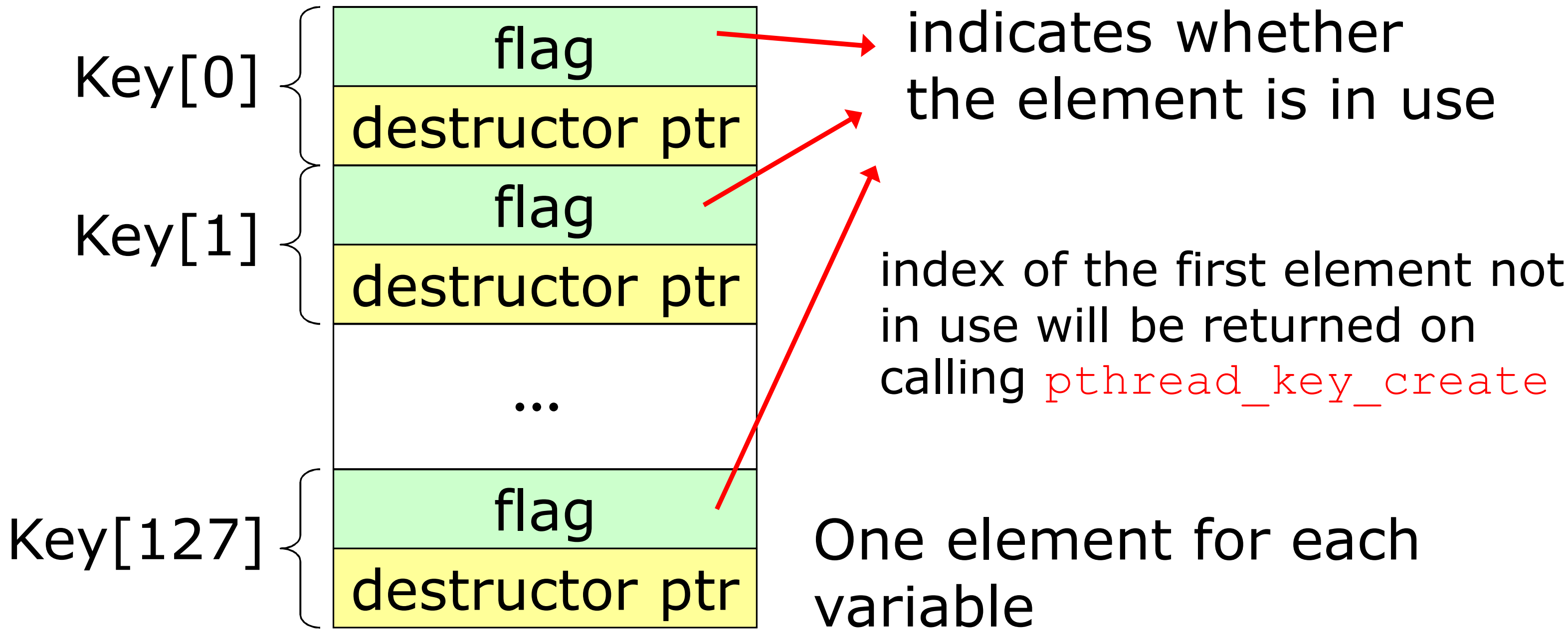


Using Thread-Specific Data

- 要達成thread-safe的function需避免使用static data，改呼叫 pthread_key_create，得到一個尚未用過Key的index (例如1)取代變數名
- 呼叫function的thread呼叫malloc取得記憶體，用以儲存其thread-specific data。然後呼叫pthread_setspecific將function取得的Key的index對應到此記憶體位址
- 在function中以pthread_getspecific取得Key的index對應到的不同thread的data

Name (Index) For Thread-Specific Data

- The kernel maintains one array of structures (Key structure) for each process (每個process一個)



pthread_key_create Function

```
#include <pthread.h>
int pthread_key_create (pthread_key_t *keyptr,
                        void (*destructor) (void *value));
                        Return 0 if ok, positive Exxx value on error
```

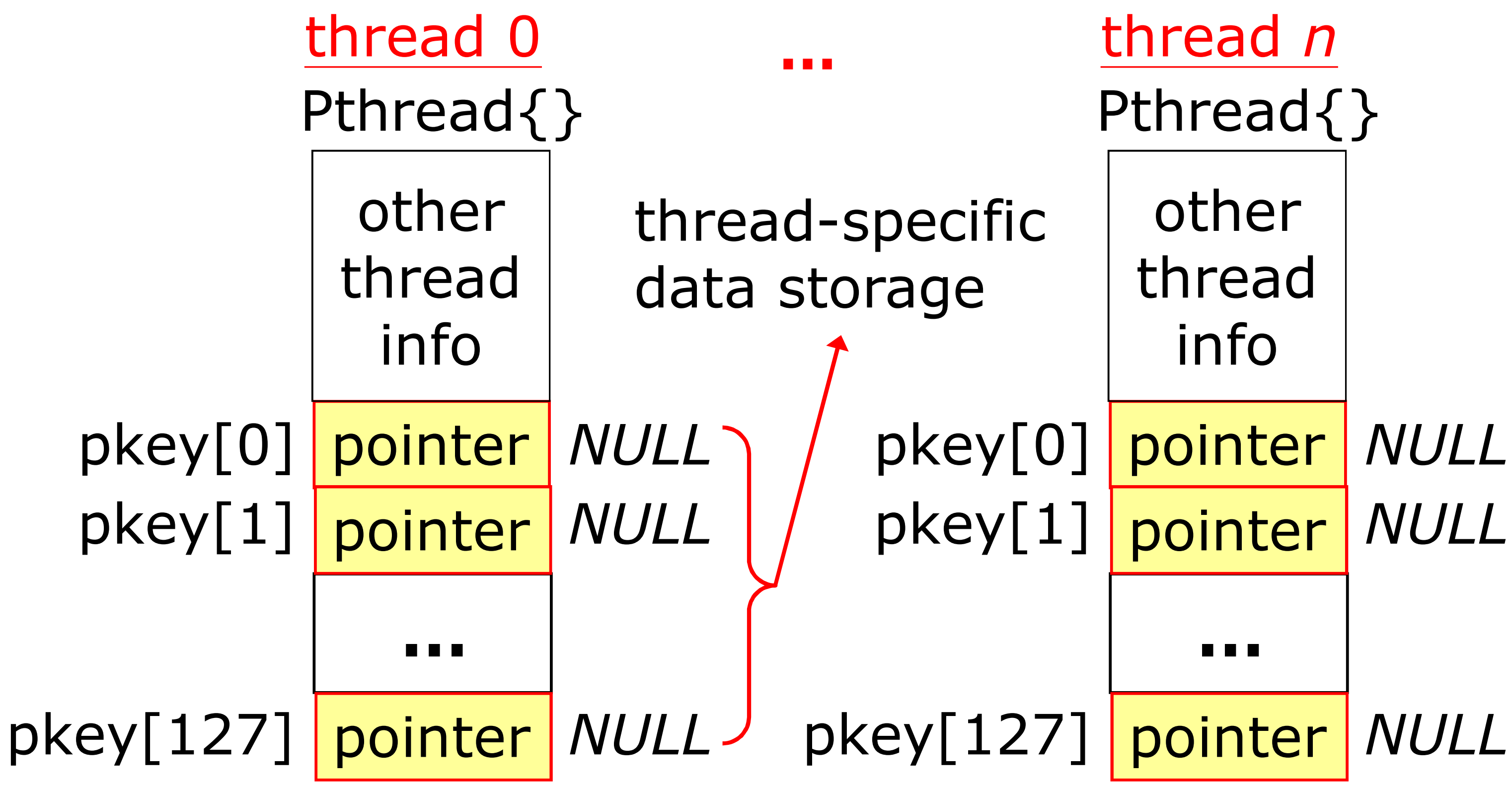
- 要求kernel給一個未用的Key index來對應thread-specific data
- 傳回的Key index放在*keyptr*
- `destructor` points to a function which will be called when a thread terminated

Storage and Mapping for Thread-Specific Data

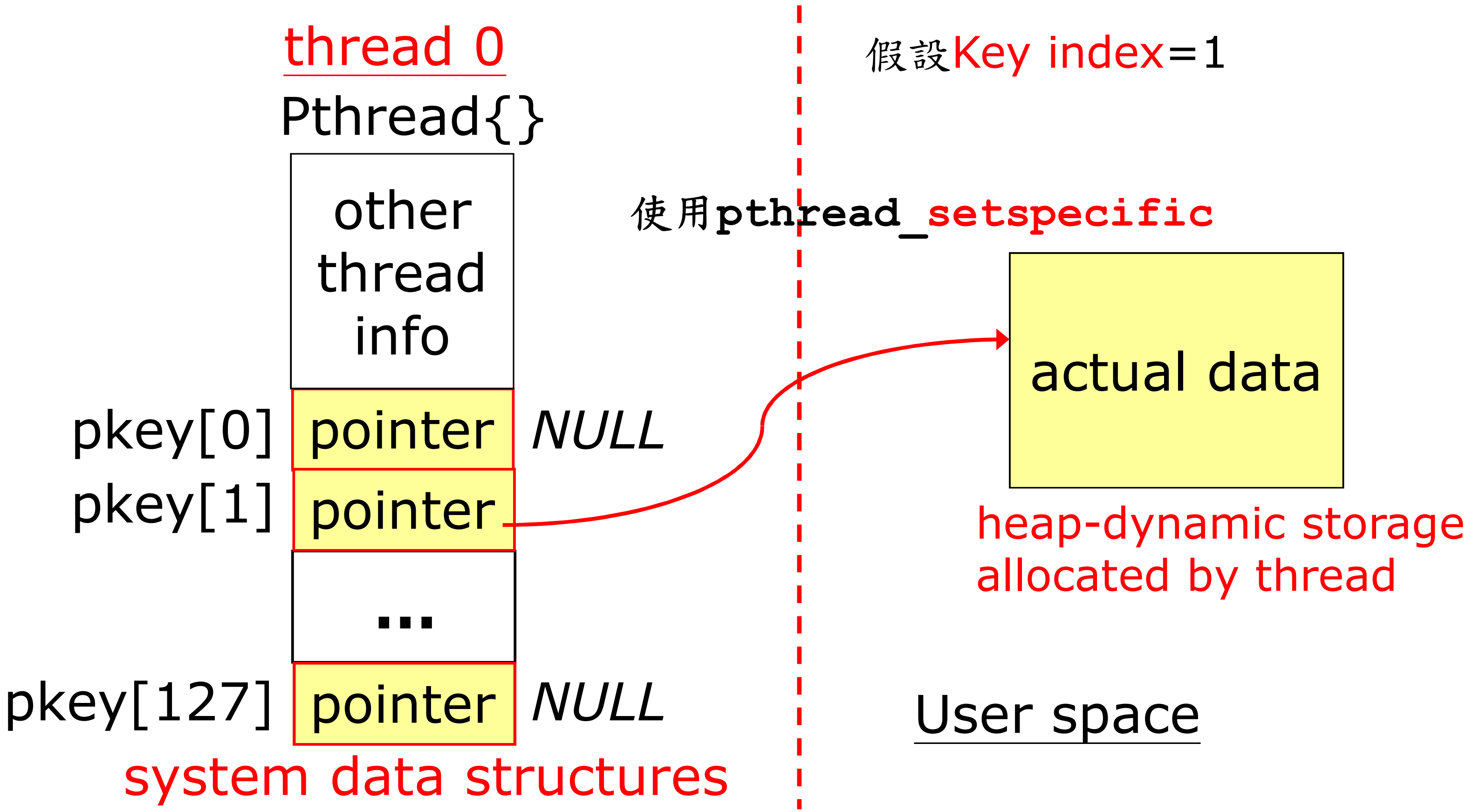
- thread呼叫 pthread_key_create，得到一個未使用的Key index (例如1)，作為static data的name
- thread呼叫malloc取得heap-dynamic storage，用以儲存static data
- thread呼叫pthread_setspecific將此storage位址對應到取得的Key index
- 在reentrant function中以pthread_getspecific取得Key index對應到的thread-specific data

Pthread Structure for the Mapping

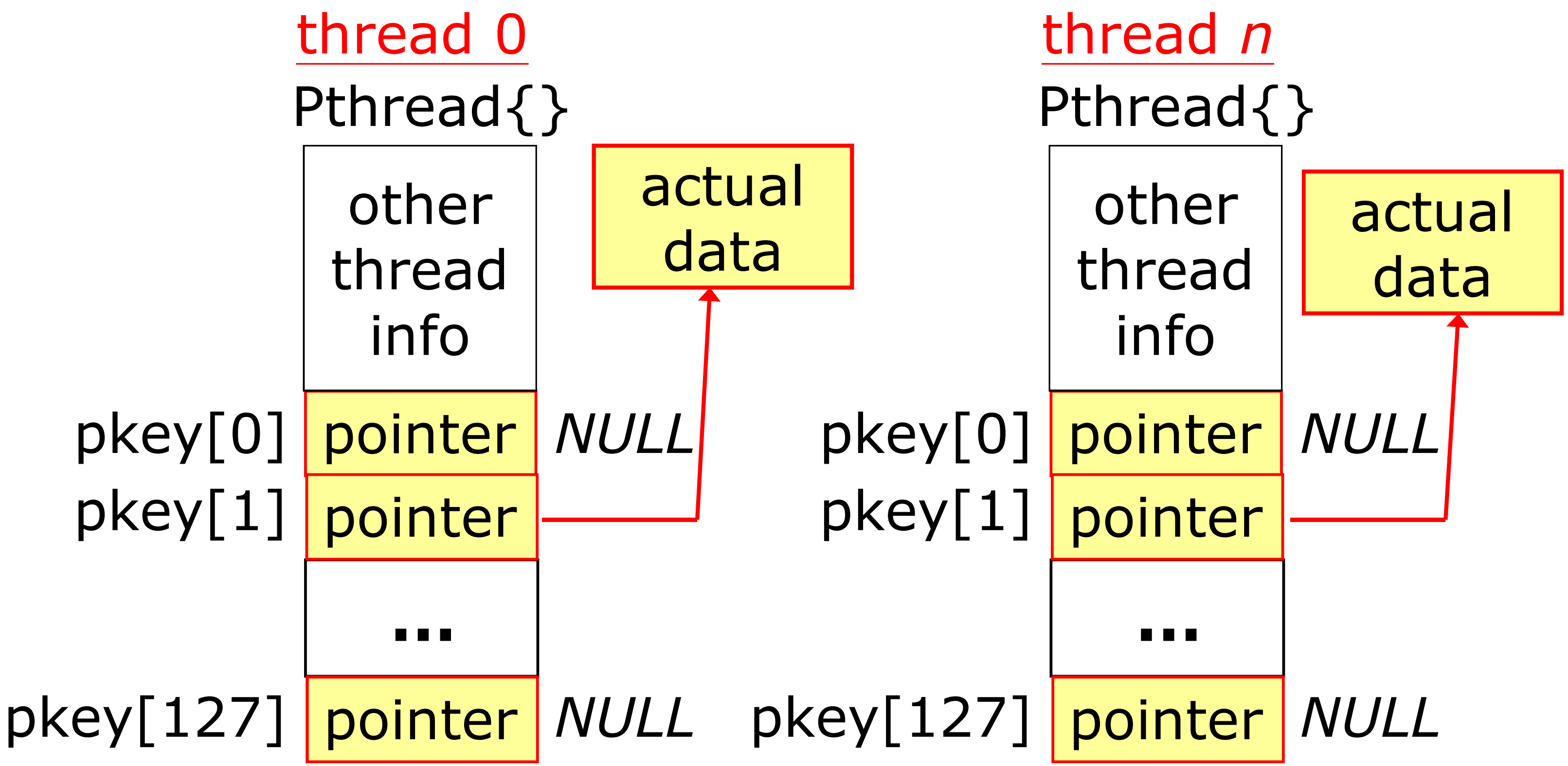
- maintained by OS; one for each thread in a process



Map Thread-Specific Data Pointer to `malloc`ed Region

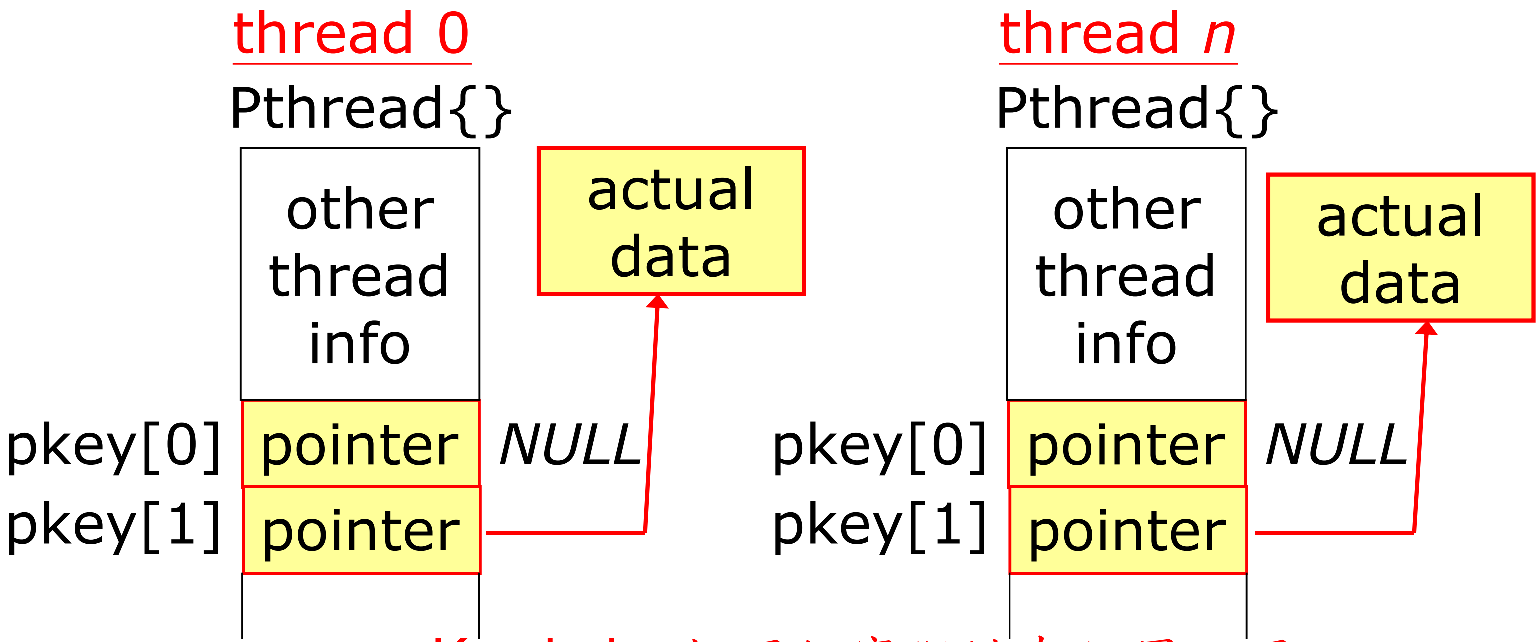


Different Threads Have Different Storages for the Same Name (Index)



One Name to Different Locations

多個thread要執行同一function access同一變數
此變數在所有thread中用相同的Key index來表示



Key index相同但實際儲存位置不同

Key index 的取得與使用

- function中任何 (變數)欲達成thread-safe要呼叫一次 pthread_key_create得到一個唯一的key index
- 不同key index供同一function中不同static data使用
- 對同一static data，由第一個執行此function的thread去請求key index即可。因為每次呼叫會傳回新的未使用的index。
- 使用pthread_once來達成此功能(對同一static data只請求一次key index)

pthread_once Function

```
#include <pthread.h>
int pthread_once (pthread_once_t *onceptr, void (*init) (void));
                    Return 0 if ok, positive Exxx value on error
```

- 需準備型態為pthread_once_t的變數，初值設為PTHREAD_ONCE_INIT，將其位址作為第一個參數傳入。
  - 此變數供kernel判斷並記錄是否為第一次呼叫
- 如為第一次呼叫，kernel會執行init，第二次以後呼叫就不會

```
void init(void) {
… }
```

pthread_once只呼叫一次pthread_key_create

- 不能直接將呼叫pthread_once的init函數指標指向pthread_key_create
  - ☞因型態不一致。要用間接的方式

```
ssize_t readline(…)
{
    …
    pthread_once(&r1_once, readline_once);
}
```

初值為PTHREAD_ONCE_INIT的global變數

```
void readline_once(void)
{
    pthread_key_create(&r1_key, readline_dest);
}
```

destructor

global，存傳回的Key index

destructor Function

- 如果某個thread有對某個key存資料(使用 pthread_setspecific)，當此thread terminates時，系統會呼叫此key的destructor
  - 傳進destructor的是指向資料的pointer

前頁的destructor名稱為readline_dest

```
void readline_dest(void *ptr)
{
    free(ptr);
}
```

釋放當初用malloc 要來的記憶體空間

對某個Key存取資料

```
#include <pthread.h>
void *pthread_getspecific (pthread_key_t key);
                    Return pointer to thread-specific data (possibly null)
int pthread_setspecific (pthread_key_t key, const void *value);
                    Return: 0 if ok, possitive Exxx on error
```

- 第一個函數傳回與第二個函數傳入的都是void pointer
- void pointer指過去的空間才是真正放thread-specific data (type自訂)的地方

Key Data Access Example

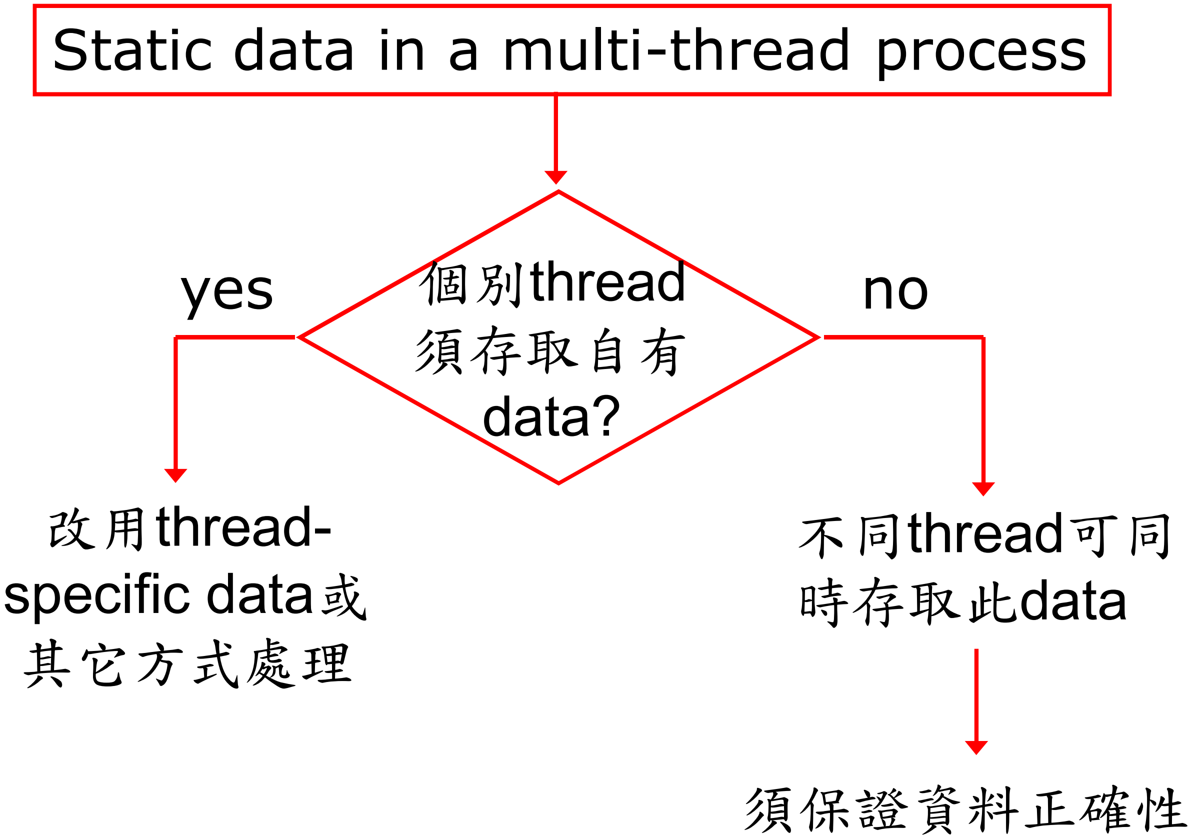
```
#include “unpthread.h”
static pthread_key_t   r1_key;
static pthread_once_t r1_once = PTHREAD_ONCE_INIT;
ssize_t readline( …_
    …
    pthread_once(&r1_once, readline_once);
    if ((ptr = pthread_getspecific (r1_key)) == NULL) {
        ptr = Malloc( … );
        pthread_setspecific (r1_key, ptr);
    }
}
```

只要一次key

確認pkey中此key的pointer是null

設定pkey中此key的pointer

So Far We Know …



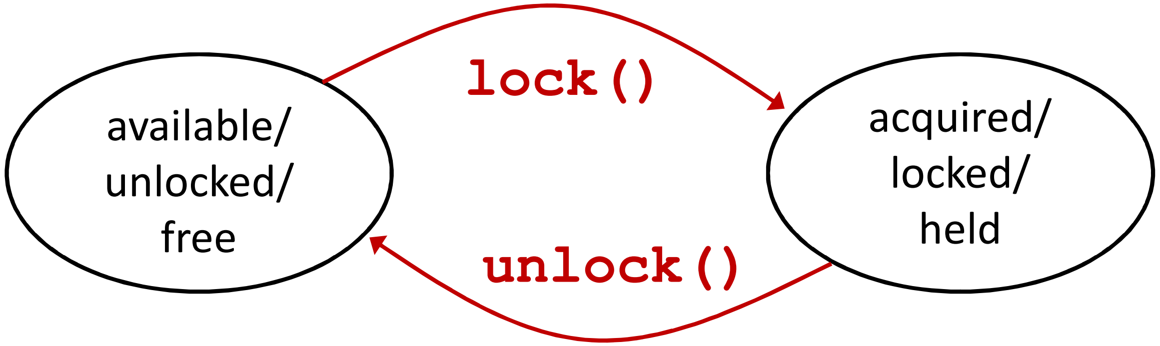
Summary

- Threads provide parallelism, avoid blocking program progress due to slow I/O, and provide a way of modulation to implement a large application
- Threads are more efficient than process but OS does not directly provide protections among threads
- We have shown how to implements threads in TCP clients and servers
- Sharing data among threads may lead to inconsistent results, calling for synchronization primitive to prevent simultaneous data modifications

Lock Usage in POSIX

What is a lock (mutex)?

- A lock (mutex) is just a variable used for mutual exclusion
- must declare a lock variable of some kind (**lock_t**)
- Possible values and operations



a lock declared by user

```
lock_t mutex;
…
lock(&mutex)
(critical session)
unlock(&mutex)
```

lock() Operation

- invoked by a thread trying to acquire the lock
- if no other thread holds the lock (i.e., it is free), the thread will acquire the lock and enter the critical section
  - this thread becomes the owner of the lock
- Otherwise (the lock is held by another thread), the thread blocks waiting for the lock becoming free

```
lock_t mutex;
…
lock(&mutex)   ⇐
(critical session)
unlock(&mutex)
```

`unlock()` Operation

- Once the owner of the lock calls **`unlock()`**, the lock is now available (free) again
- If no other threads are waiting for the same lock (i.e., no other thread has called **`lock()`** on the same lock and is stuck therein), the state of the lock is simply changed to free.
- Otherwise, one of the waiting threads will (eventually) acquire the lock and enter the critical section

```
lock_t mutex;
…
lock(&mutex)
(critical session)
unlock(&mutex)
```

Pthread Locks (in POSIX)

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread_mutex_lock(&lock); // wrapper; exits on failure
a = a + 1;
Pthread_mutex_unlock(&lock);
```

mutual exclusion

Mutexes: Mutual Exclusion

- A mutex is a variable of type `pthread_mutex_t`

- We can lock (by `pthread_mutex_lock`) or unlock (by `pthread_mutex_unlock`) a mutex

- If we try to lock a mutex that is already locked by some other thread, we are blocked until the mutex is unlocked

Using Mutexes

- We can use a mutex to protect a shared variable from being updated simultaneously

thread A		thread B	
lock mutex	← ②	lock mutex	← ①
count++		count--	
unlock mutex		unlock mutex	

(thread A locks the mutex first)	(thread B locks the mutex first)
$register_1 = \texttt{count}$	$register_2 = \texttt{count}$
$register_1 = register_1 + 1$	$register_2 = register_2 - 1$
(context switch)	(context switch)
(thread B blocks in ①)	(thread A blocks in ②)

Mutex Example

```
#include "unpthread.h"
int count;
pthread_mutex_t count_mutex = PTHREAD_MUTEX_INITIALIZER;

void *do_it(void *vptr)
{
   …
   Pthread_mutex_lock(&count_mutex);
   count = count + 1;
   Pthread_mutex_unlock(&count_mutex);
}
```

static mutex variable
一定要設的初值

Mutex Usage: separated reading from/writing to a shared variable

`a` is a shared variable;

`b` is stack-dynamic

```
b = a;
c = …
b = b + c;
a = b;
```

```
b = a;
c = …
b = b – c;
a = b;
```

```
lock mutex
b = a;
c = …
b = b + c;
a = b;
unlock mutex
```

```
lock mutex
b = a;
c = …
b = b – c;
a = b;
unlock mutex
```

Minimize The Scope of Lock

```
lock mutex
b = a;
c = ...
b = b + c;
a = b;
unlock mutex
```

```
lock mutex
b = a;
c = ...
b = b − c;
a = b;
unlock mutex
```

If the value of `c` does not depends on `a` or `b`

```
c = ...
lock mutex
b = a;
b = b + c;
a = b;
unlock mutex
```

```
c = ...
lock mutex
b = a;
b = b − c;
a = b;
unlock mutex
```

Minimize the Scope of Lock

```
int List_Insert(list_t *L, int k
  pthread_mutex_lock(&L->lock);
  node_t *new = malloc(sizeof(no
  if (new == NULL) {
    perror("malloc");
    pthread_mutex_unlock(&L->loc
    return -1; // fail
  }
  new->key = key;
  new->next = L->head;
  L->head = new;
  pthread_mutex_unlock(&L->lock)
  return 0; // success
}
```

```
void List_Insert(list_t *L, int key) {
  // synchronization not needed
  node_t *new = malloc(sizeof(node_t));
  if (new == NULL) {
    perror("malloc");
    return;
  }
  new->key = key;

  // just lock critical section
  pthread_mutex_lock(&L->lock);
  new->next = L->head;
  L->head = new;
  pthread_mutex_unlock(&L->lock);
}
```

assuming malloc() is thread-safe

Minimize the Number of Unlocks

```
int List_Lookup(list_t *L, int key) {
  pthread_mutex_lock(&L->lock);
  node_t *curr = L->head;
  while (curr) {
    if (curr->key == key) {
      pthread_mutex_unlock(&L->lock);
      return 0; // success
    }
    curr = curr->next;
  }
  pthread_mutex_unlock(&L->lock);
  return -1; // failure
}
```

```
int List_Lookup(list_t *L, int key) {
  int rv = -1;
  pthread_mutex_lock(&L->lock);
  node_t *curr = L->head;
  while (curr) {
    if (curr->key == key) {
      rv = 0;
      break;
    }
    curr = curr->next;
  }
  pthread_mutex_unlock(&L->lock);
  return rv;
}
```

Mutex Usage: Testing Shared Variable

- 多個threads可以使用mutex來變更同一個shared variable而不會產生不正確的結果
- 如果有thread要測試此shared variable的值進行不同的動作，則也要用mutex保證執行正確

count的值可能在測試>0後do something前變為0

```
if (count > 0)
   // do something;
```

count的值在測試是否>0及do something期間不變

```
lock mutex
if (count > 0)
   // do something;
unlock mutex
```

Multiple Threads Working Together …

```
lock mutex
count++
unlock mutex
```

```
lock mutex
count--
unlock mutex
```

```
lock mutex
if (count > 0)
   // do something;
unlock mutex
```

任何一個thread搶先lock mutex，都可以阻止在此thread未unlock mutex前其它threads對同一變數count作讀或寫的動作

⇩

serialization

Reading Shared Simple Variables

- 如果thread僅僅是單純讀取某shared simple variable的值，則不一定要用mutex

```
...
printf (“%d\n”,count)
...
```

count的值可能在讀取後印出前改變，但不妨礙程式正確性

Mutex Usage: Calling Non-Reentrant Functions

Thread 1

```
void *mythreadA(void *arg)
{
  char *str;
  …
 Pthread_mutex_lock(&mutex);
  str = malloc(sizeof(char)*200);
 Pthread_mutex_unlock(&mutex);
  …
 Pthread_mutex_lock(&mutex);
  free(str);
 Pthread_mutex_unlock(&mutex);
}
```

Thread 2

```
void *mythreadB(void *arg)
{
  int *vec;
  …
 Pthread_mutex_lock(&mutex);
  vec = malloc(sizeof(char)*100);
 Pthread_mutex_unlock(&mutex);
  …
 Pthread_mutex_lock(&mutex);
  free(vec);
 Pthread_mutex_unlock(&mutex);
}
```

Condition Variables

Polling (Busy Waiting)

- 如果有thread要等到某個shared variable的值變為特定值時才進行後續動作，則要持續讀取(polling)且測試

```
L1:
  lock mutex
  if (count > 0)
     goto L2;
  unlock mutex
  goto L1
L2: // do something
  unlock mutex
```

Busy waiting,
waste CPU time

condition variable可以讓欲測試的thread sleep變成blocked等候別的thread的通知 ⇒ 不會浪費CPU時間

Another need for condition variable

- Waits for the completion of another thread

```
volatile int done = 0;

void *child(void *arg) {
  printf("child\n");
  done = 1;
  return NULL;
}
```

```
int main(int argc, char *argv[]) {
  printf("parent: begin\n");
  pthread_t c;
  Pthread_create(&c, NULL, child, NULL);
    // create child
  while (done == 0)
    ; // spin
  printf("parent: end\n");
  return 0;
}
```

Wastes CPU time

Condition Variables

- A condition variable is an explicit queue that
  - threads can put themselves on when some state of execution (i.e., some condition) is not as desired (by waiting on the condition)
  - some other thread can wake one (or more) of those waiting threads
- In POSIX, a condition variable is of type `pthread_cond_t`
  - should be initialized with `PTHREAD_COND_INITIALIER`
  - used with `pthread_cond_wait` (等待) and `pthread_cond_signal` (喚醒)

pthread_cond_wait

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
cond: pointer to a pthread_cond_t variable with initial
value PTHREAD_COND_INITIALIZER;
mutex: pointer to a pthread_mutex_t variable with initial
value PTHREAD_MUTEX_INITIALIZER;
```

- It puts the calling thread to sleep (blocked)
- It also releases the lock (mutex) when putting the caller to sleep
- waits for some other thread to signal it

Condition Variable Example: Wait

```
#include "unpthread.h"
int count = 0;
pthread_mutex_t count_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t count_cond = PTHREAD_COND_INITIALIZER;
…
Pthread_mutex_lock(&count_mutex);
while (count == 0)
    Pthread_cond_wait(&count_cond, &count_mutex);
// do something
Pthread_mutex_unlock(&count_mutex);
```

unlock count_mutex並進入blocked state等待count_cond之signal

The Need for Mutex In Wait

If not protected by `count_mutex`, two or more threads may enter here

```
Pthread_mutex_lock(&count_mutex);
while (count == 0)
    Pthread_cond_wait(&count_cond, &count_mutex);
// do something
Pthread_mutex_unlock(&count_mutex);
```

If not protected by `count_mutex`, a thread may detect that `count == 0` and then get interrupted. The value of `count` no longer `== 0` after the thread comes back but the thread goes to sleep anyway.

The Need for Atomic Unlock In Wait

pthread_cond_wait若無 “unlock count_mutex”動作，則mutex被鎖住，其它thread無法更新count值並signal此thread ⇒ block forever

```
Pthread_mutex_lock(&count_mutex);
while (count == 0)
    Pthread_cond_wait(&count_cond, &count_mutex);
// do something
Pthread_mutex_unlock(&count_mutex);
```

“unlock count_mutex”與 “進入sleep mode等待對應之signal”為atomic (unbreakable)，否則的話其它thread可能在中途介入造成問題

pthread_cond_signal

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);
```

- Awake a thread (by sending a signal to it) that is waiting on condition variable `*cond`
- The awaken thread will be ready for running

Condition Variable Example: Signal

其它thread變更count值後會喚醒前頁的thread進行檢查

```
Pthread_mutex_lock(&count_mutex);
count = 1;
Pthread_cond_signal(&count_cond);
Pthread_mutex_unlock(&count_mutex);
```

①signal

④recheck

②此signal會喚醒此thread

```
while (count == 0)
    Pthread_cond_wait(&count_cond, &count_mutex);
```

③在此thread從Pthread_cond_wait return之前，
Pthread_cond_wait會重新lock count_mutex

The Need for The Lock Before Returning From Wait

Thread 1

```
while (count == 0)
    Pthread_cond_wait(&count_cond, &count_mutex);
// do something
```

Thread 2

```
Pthread_mutex_lock(&count_mutex);
count = 1;
Pthread_cond_signal(&count_cond);
Pthread_mutex_unlock(&count_mutex);
```

If Thread 1 awakes but `count_mutex` is not locked before returning from the wait, Thread 2 may get in and set `count` to 1.

The Need for the Variable Count

Thread 1

```
Pthread_mutex_lock(&count_mutex);
[redacted]
    Pthread_cond_wait(&count_cond, &count_mutex);
// do something
Pthread_mutex_unlock(&count_mutex);
```

Thread 2

```
Pthread_mutex_lock(&count_mutex);
[redacted]
Pthread_cond_signal(&count_cond);
Pthread_mutex_unlock(&count_mutex);
```

If Thread 2 calls `signal()` before Thread 1 calls `wait()`, Thread 1 will be stuck in wait() forever.

Awakening Multiple Threads

- `pthread_cond_signal` awakens one thread that is waiting on the condition variable
- `pthread_cond_broadcast` will wake up all threads that are blocked on the condition variable
- `pthread_cond_timewait` lets a thread place a limit (absolute time) on how long it will block

Summary

- Creating threads is normally faster than creating new processes (using fork)
- All threads in a process share global variables
- The sharing introduces synchronization problem, which calls for mutexes and condition variables
- We show how to let a function thread-safe by using thread-specific data