Multithreading Basics and Programming

Prof. Yung-Pin Cheng

2017 03

History

- In the past (< 1996), when there was no multithreading support from O.S., how a server (such as telnet, BBS server) is implemented?
- The era of multi-process programming

Unix's fork() revisited

Process management	
Call	**Description**
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

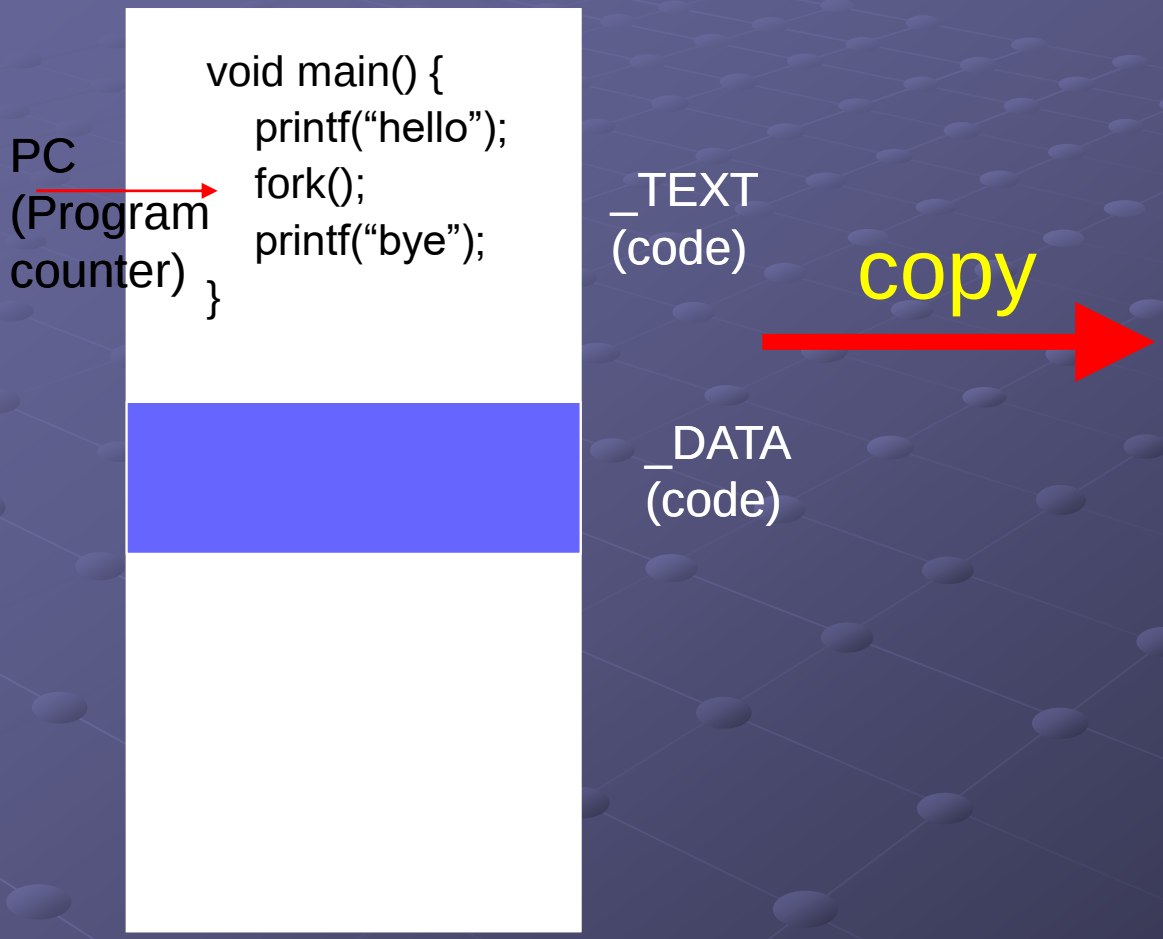
An example of Unix fork()

```
void main() {
    printf("hello");
    fork();
    printf("bye");
}
```

The output

```
hello
bye
bye
```

How fork() is implemented in Unix?



Process's Image in Memory

fork() example again

```
void main() {
   if (fork() == 0)
     printf(“ in the child process”);
   else
     printf(“ in the parent process”);
}
```

An old-fashioned concurrent server

```
main() {
// create a TCP/IP socket to use
s =socket(PF_INET,SOCK_STREAM ,0);

// bind the server address
Z = bind(s, (struct sockaddr *)&adr_srvr,
     len_inet);

// make it a listening socket
Z = listen(s, 10);

// start the server loop
for (;;) {
    // wait for a connect
    c = accept(s, (struct sockaddr*)
     &adr_clnt,& len_int);
    PID = fork();
    if (PID > 0) {
            // parent process
        close(c);
        continue ;
    }
    // child process
    rx = fdopen(c,"r");
    tx = fdopen(dup(c), "w");

    // process client's request
    ……..
    fclose(tx); fclose(rx);
    exit(0);
}
```

Problems with old-fashioned multi-processes programming

- Context switching overhead is high
- Communications between forked processes also has high cost
  - typical mechanism – via IPC (interprocess communication) such as shared memory, semaphore, message queue
  - these are cross address space communication
  - invoking of IPC cause mode switch – may cause a process to block.

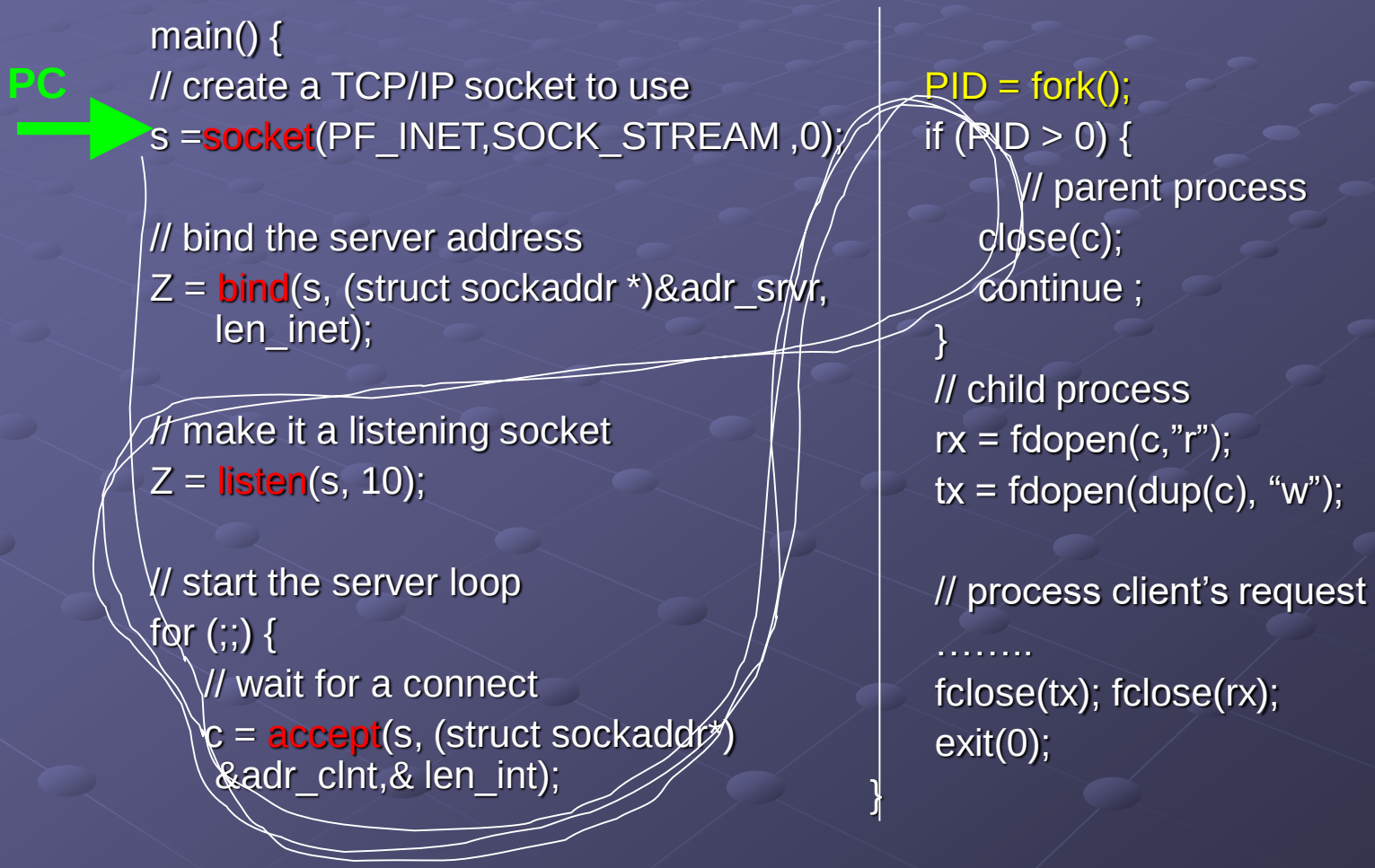
The concept of a ***process***

- However, processes are still the very basic element in O.S.
- UNIT of resources ownership
  - Allocated with virtual address space + control of other resources such as I/O, files….
- Unit of dispatching
  - An execution paths (may interleaved with other processes)
  - An execution state, dispatching priority.
  - Controlled by OS

Now, What is a thread?

- A thread is an execution path in the code segment
- O.S. provide an individual Program Counter (PC) for each execution path

An example


PC
main() {
// create a TCP/IP socket to use
s =socket(PF_INET,SOCK_STREAM ,0);
// bind the server address
Z = bind(s, (struct sockaddr *)&adr_srvr,
len_inet);
// make it a listening socket
Z = listen(s, 10);
// start the server loop
for (;;) {
// wait for a connect
c = accept(s, (struct sockaddr*)
&adr_clnt,& len_int);
PID = fork();
if (PID > 0) {
// parent process
close(c);
continue ;
}
// child process
rx = fdopen(c,"r");
tx = fdopen(dup(c), "w");
// process client's request
……..
fclose(tx); fclose(rx);
exit(0);
}

Comments

- Traditional program is one thread per process.
- The main thread starts with main()
- Only one thread (or, program counter) is allowed to execute the code segment
- To add a new PC, you need to fork() to have another PC to execute in another process address space.

Multithreading

- The ability of an OS to support multiple threads of execution within a single process (many program counters in a code segment)
- Windows support multithreading earlier (mid 1990)
- SunOS, Linux came late

The example again

```
main() {
// create a TCP/IP socket to use
s =socket(PF_INET,SOCK_STREAM ,0);

// bind the server address
Z = bind(s, (struct sockaddr *)&adr_srvr,
      len_inet);

// make it a listening socket
Z = listen(s, 10);

// start the server loop
for (;;) {
     // wait for a connect
     c = accept(s, (struct sockaddr*)
      &adr_clnt,& len_int);
     hThrd = createThread(Threadfunc,…)
     close(c);
     continue ;
    }
}
ThreadFunc() {
     // child process
     rx = fdopen(c,”r”);
     tx = fdopen(dup(c), “w”);

     // process client’s request
     ……..
     fclose(tx); fclose(rx);
     exit(0);
}
```

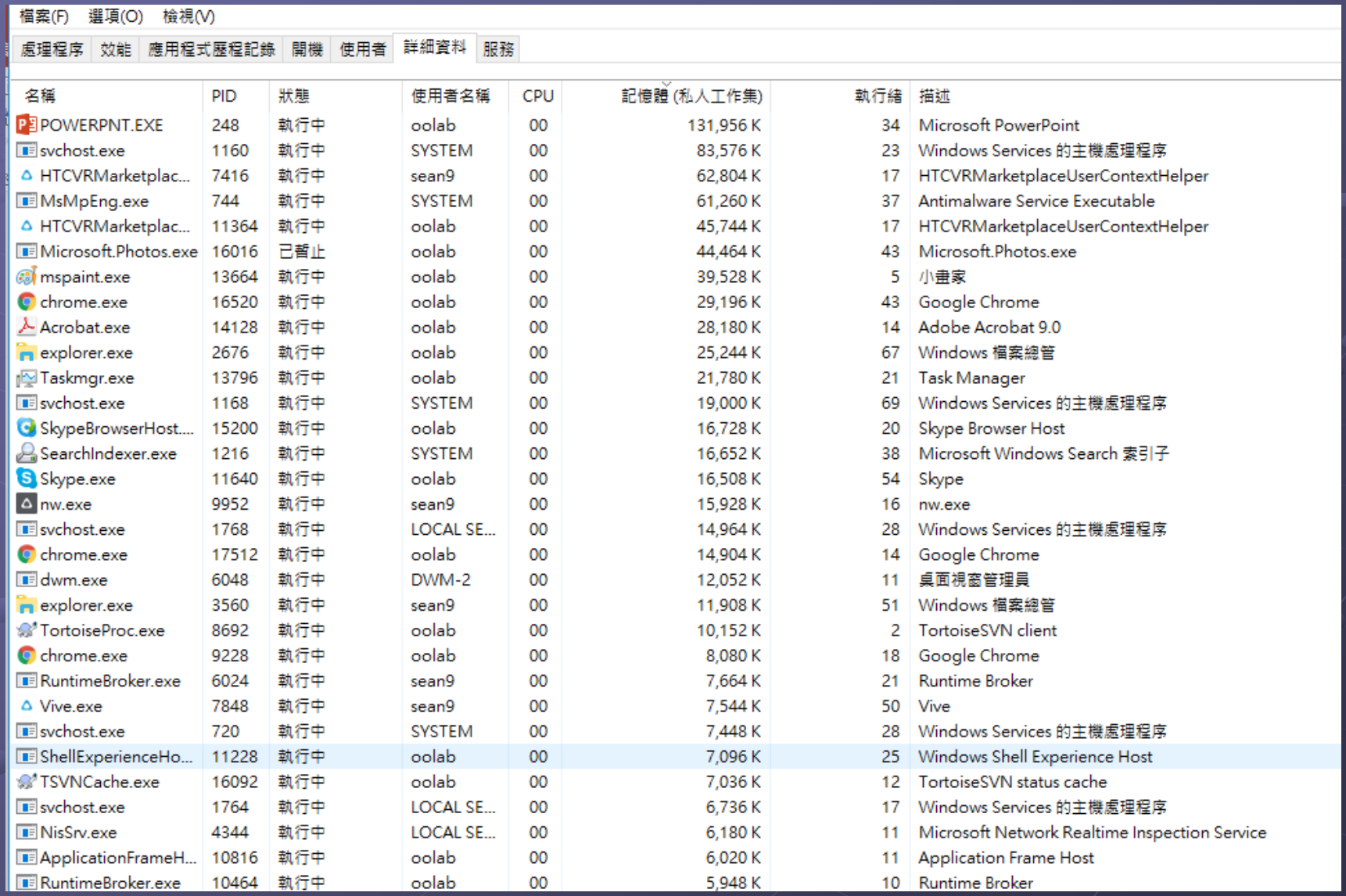
PC

PC

PC

PC

Most APPs are Multithreaded NOW!!



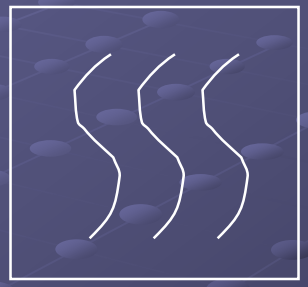
檔案(F) 選項(O) 檢視(V)

處理程序 效能 應用程式歷程記錄 開機 使用者 詳細資料 服務

名稱	PID	狀態	使用者名稱	CPU	記憶體 (私人工作集)	執行緒	描述
POWERPNT.EXE	248	執行中	oolab	00	131,956 K	34	Microsoft PowerPoint
svchost.exe	1160	執行中	SYSTEM	00	83,576 K	23	Windows Services 的主機處理程序
HTCVRMarketplac...	7416	執行中	sean9	00	62,804 K	17	HTCVRMarketplaceUserContextHelper
MsMpEng.exe	744	執行中	SYSTEM	00	61,260 K	37	Antimalware Service Executable
HTCVRMarketplac...	11364	執行中	oolab	00	45,744 K	17	HTCVRMarketplaceUserContextHelper
Microsoft.Photos.exe	16016	已暫止	oolab	00	44,464 K	43	Microsoft.Photos.exe
mspaint.exe	13664	執行中	oolab	00	39,528 K	5	小畫家
chrome.exe	16520	執行中	oolab	00	29,196 K	43	Google Chrome
Acrobat.exe	14128	執行中	oolab	00	28,180 K	14	Adobe Acrobat 9.0
explorer.exe	2676	執行中	oolab	00	25,244 K	67	Windows 檔案總管
Taskmgr.exe	13796	執行中	oolab	00	21,780 K	21	Task Manager
svchost.exe	1168	執行中	SYSTEM	00	19,000 K	69	Windows Services 的主機處理程序
SkypeBrowserHost....	15200	執行中	oolab	00	16,728 K	20	Skype Browser Host
SearchIndexer.exe	1216	執行中	SYSTEM	00	16,652 K	38	Microsoft Windows Search 索引子
Skype.exe	11640	執行中	oolab	00	16,508 K	54	Skype
nw.exe	9952	執行中	sean9	00	15,928 K	16	nw.exe
svchost.exe	1768	執行中	LOCAL SE...	00	14,964 K	28	Windows Services 的主機處理程序
chrome.exe	17512	執行中	oolab	00	14,904 K	14	Google Chrome
dwm.exe	6048	執行中	DWM-2	00	12,052 K	11	桌面視窗管理員
explorer.exe	3560	執行中	sean9	00	11,908 K	51	Windows 檔案總管
TortoiseProc.exe	8692	執行中	oolab	00	10,152 K	2	TortoiseSVN client
chrome.exe	9228	執行中	oolab	00	8,080 K	18	Google Chrome
RuntimeBroker.exe	6024	執行中	sean9	00	7,664 K	21	Runtime Broker
Vive.exe	7848	執行中	sean9	00	7,544 K	50	Vive
svchost.exe	720	執行中	SYSTEM	00	7,448 K	28	Windows Services 的主機處理程序
ShellExperienceHo...	11228	執行中	oolab	00	7,096 K	25	Windows Shell Experience Host
TSVNCache.exe	16092	執行中	oolab	00	7,036 K	12	TortoiseSVN status cache
svchost.exe	1764	執行中	LOCAL SE...	00	6,736 K	17	Windows Services 的主機處理程序
NisSrv.exe	4344	執行中	LOCAL SE...	00	6,180 K	11	Microsoft Network Realtime Inspection Service
ApplicationFrameH...	10816	執行中	oolab	00	6,020 K	11	Application Frame Host
RuntimeBroker.exe	10464	執行中	oolab	00	5,948 K	10	Runtime Broker

Possible combination of thread and processes

One process one thread



One process multiple thread

Multiple processes
One thread per process

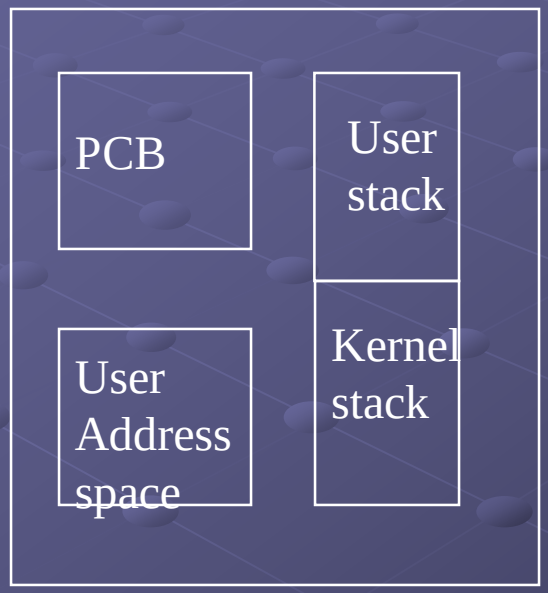
Multiple processes multiple
Threads per process

Process is still there, what's new for thread?

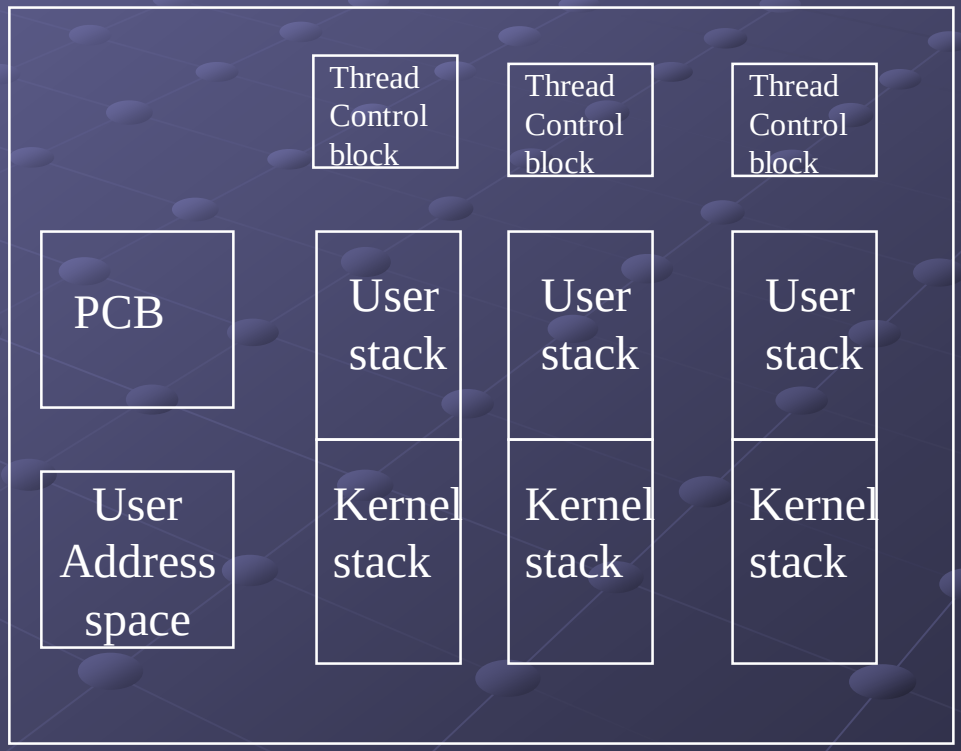
- **With process**
  - Virtual address space (holding process image)
  - Protected access to CPU, files, and I/O resources
- **With thread (each thread has its own..)**
  - Thread execution state
  - Saved thread context (an independent PC within a process)
  - An execution stack
  - Per-thread static storage for local variable
  - Access to memory and resource of its process, shared with all other threads in that process

Single threaded and multithreaded model

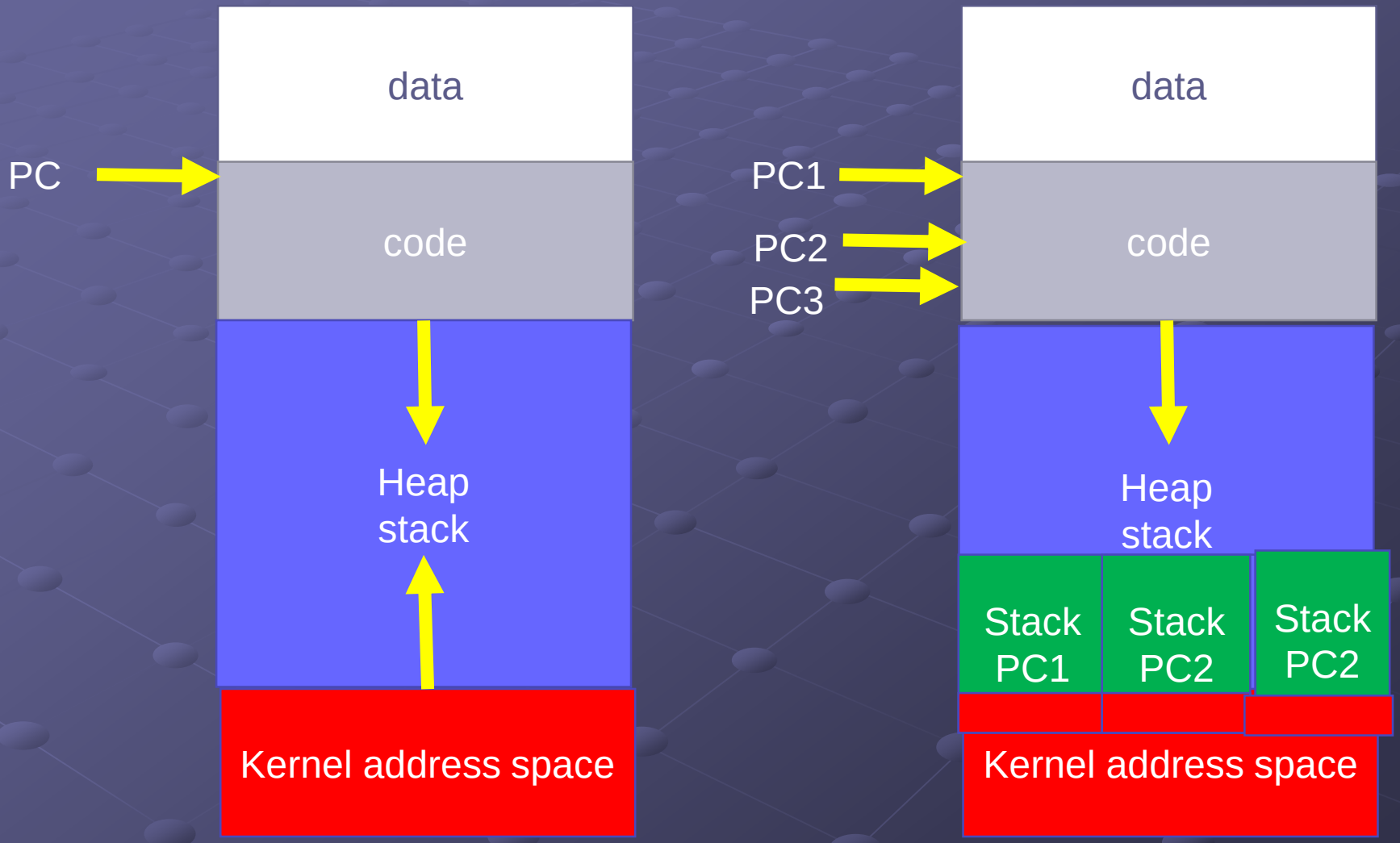
Single-threaded process



Multithread Process model



The meaning of each thread has its own stack but share address space?



C# Multithread examples

All examples assume the following namespaces are imported:

```
using System;
using System.Threading;
```

```
class ThreadTest
{
  static void Main()
  {
    Thread t = new Thread (WriteY);          // Kick off a new thread
    t.Start();                               // running WriteY()

    // Simultaneously, do something on the main thread.
    for (int i = 0; i < 1000; i++) Console.Write ("x");
  }

  static void WriteY()
  {
    for (int i = 0; i < 1000; i++) Console.Write ("y");
  }
}
```

```
xxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Each thread has its own copy of stack.

```
static void Main()
{
  new Thread (Go).Start();      // Call Go() on a new thread
  Go();                         // Call Go() on the main thread
}

static void Go()
{
  // Declare and use a local variable - 'cycles'
  for (int cycles = 0; cycles < 5; cycles++) Console.Write ('?');
}

??????????
```

Threads that can share on same object

```
class ThreadTest
{
  bool done;

  static void Main()
  {
    ThreadTest tt = new ThreadTest();   // Create a common instance
    new Thread (tt.Go).Start();
    tt.Go();
  }

  // Note that Go is now an instance method
  void Go()
  {
     if (!done) { done = true; Console.WriteLine ("Done"); }
  }
}
```

Because both threads call `Go()` on the same `ThreadTest` instance, they share the `done` field. This results in "Done" being printed once instead of twice:

```
Done
```

Static variables are in data segment

Static fields offer another way to share data between threads. Here's the same example with done as a static field:

```
class ThreadTest
{
  static bool done;    // Static fields are shared between all threads

  static void Main()
  {
    new Thread (Go).Start();
    Go();
  }

  static void Go()
  {
    if (!done) { done = true; Console.WriteLine ("Done"); }
  }
}
```

Both of these examples illustrate another key concept: that of thread safety (or rather, lack of it!) The output is actually indeterminate: it's possible (though unlikely) that "Done" could be printed twice. If, however, we swap the order of statements in the Go method, the odds of "Done" being printed twice go up dramatically:

```
static void Go()
{
  if (!done) { Console.WriteLine ("Done"); done = true; }
}

Done
Done   (usually!)
```

The problem is that one thread can be evaluating the if statement right as the other thread is executing the WriteLine statement—before it's had a chance to set done to true.

Thread lock (mutex) in C# (not IPC, it is faster)

The remedy is to obtain an exclusive lock while reading and writing to the common field. C# provides the lock statement for just this purpose:

```
class ThreadSafe
{
  static bool done;
  static readonly object locker = new object();

  static void Main()
  {
    new Thread (Go).Start();
    Go();
  }

  static void Go()
  {
    lock (locker)
    {
      if (!done) { Console.WriteLine ("Done"); done = true; }
    }
  }
}
```

Key benefits of multithreading

- Less time to create a thread than a process
- Less time to terminate a thread than a process
- Less time to switch a thread
- Enhance efficiency in communication: no need for kernel to intervene
- MACH shows a factor of 10

Thread Synchronization vs Process Synchronization

A Comparison of Locking Constructs

Construct	Purpose	Cross-process?	Overhead*
`lock` (`Monitor.Enter` / `Monitor.Exit`)	Ensures just one thread can access a resource (or section of code) at a time	-	20ns
`Mutex`		Yes	1000ns
`SemaphoreSlim` (introduced in Framework 4.0)	Ensures not more than a specified number of concurrent threads can access a resource	-	200ns
`Semaphore`		Yes	1000ns
`ReaderWriterLockSlim` (introduced in Framework 3.5)	Allows multiple readers to coexist with a single writer	-	40ns
`ReaderWriterLock` (effectively deprecated)		-	100ns

*Time taken to lock and unlock the construct once on the same thread (assuming no blocking), as measured on an Intel Core i7 860.

Boosted by SMP

- If you install SMP motherboard and SMP enhanced O.S. kernel (Windows and Linux both support)
- Each thread can be assign to an individual CPU.
- The result – boosted performance

Java threads

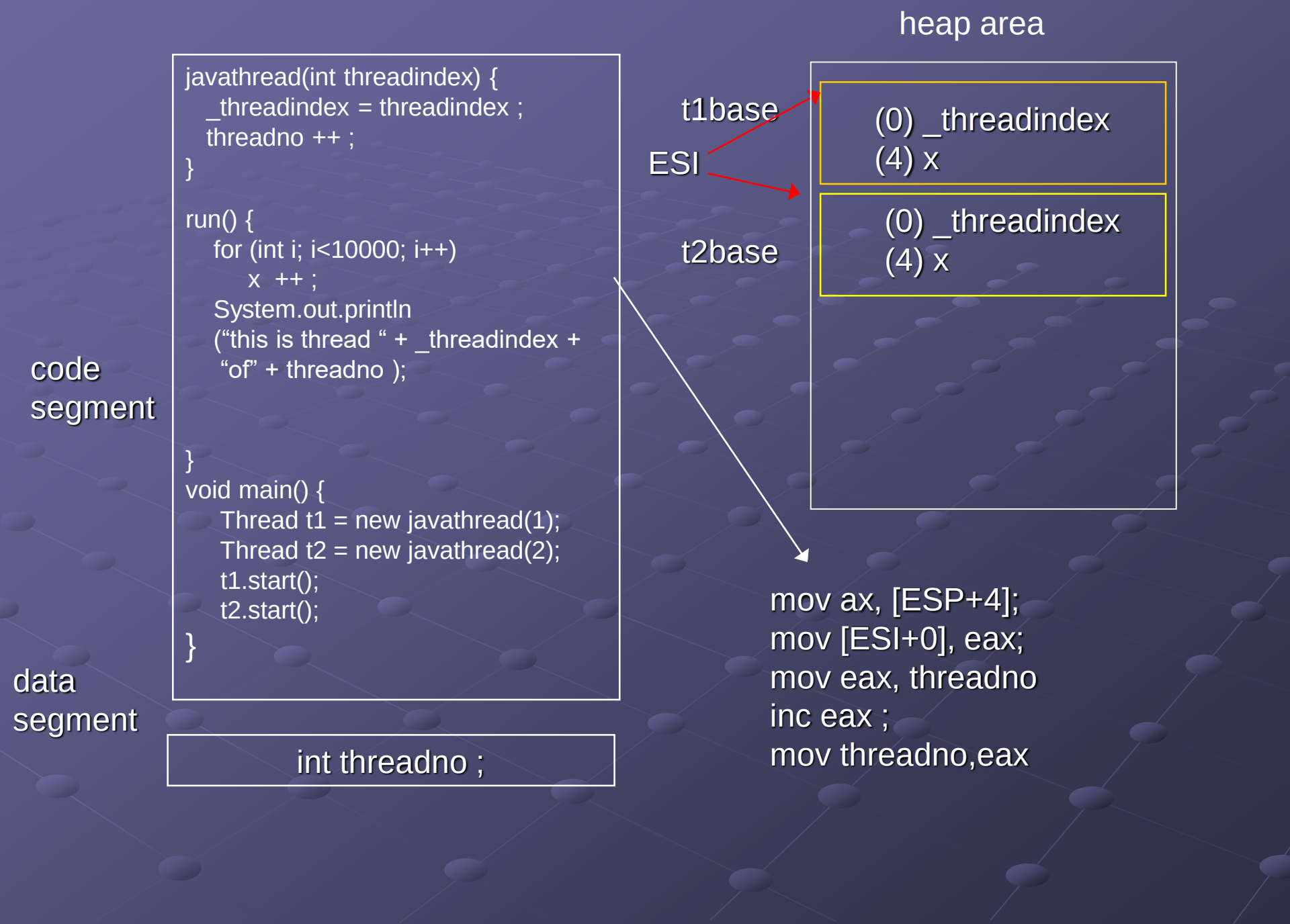
Like a global variable in C, C++

```
class javathread extends Thread
    {
int _threadindex ;
int  x;
static int threadno ;
// constructor
javathread(int threadindex) {
   _threadindex = threadindex ;
   threadno ++ ;
}
run() {
    for (int i; i<10000; i++)
        x ++ ;
    System.out.println
    (“this is thread “ +
     _threadindex + “of”
      threadno
}
```

```
void main() {
    Thread t1 = new javathread(1);
    Thread t2 = new javathread(2);
    t1.start();
    t2.start();
    System.out.println(“This is
    main thread\n”);
}
```

POSSIBLE OUTPUT:

This is main thread !
This is thread 1 of 2
This is thread 2 of 2


heap area
javathread(int threadindex) {
_threadindex = threadindex ;
threadno ++ ;
}
run() {
for (int i; i<10000; i++)
x ++ ;
System.out.println
(“this is thread “ + _threadindex +
“of” + threadno);
}
void main() {
Thread t1 = new javathread(1);
Thread t2 = new javathread(2);
t1.start();
t2.start();
}
t1base
ESI
t2base
(0) _threadindex
(4) x
(0) _threadindex
(4) x
code segment
data segment
int threadno ;
mov ax, [ESP+4];
mov [ESI+0], eax;
mov eax, threadno
inc eax ;
mov threadno,eax

Win32 Thread – an example (Low level system calls of Windows)

```
int main() {
        HANDLE hThrd ;
        DWORD threadid ;
        int i ;
        for (i=0;i<5;i++) {
          hThrd = CreateThread(NULL,0,ThreadFunc,(LPVOID)I,0,&threadid);
          if (hThr) printf("Thread launched %d\n",i);
        }
        Sleep(2000);
        return EXIT_SUCCESS;
}
DWORD WINAPI ThreadFunc(LPVOID n) {
          int i ;
          for (i=0;i<10;i++) {
              printf("%d%d%d%d%d%d%d%d\n", n,n,n,n,n,n,n,n);
}
```

possible output

```
0000000
Thread lauched
1111111
1111111
1111111
1111111
2222222
2222222
2222222
2222222
Thread lauched
```

```
0000000
Thread lauched
1111111
1111111
1111111
1111111
2222222
2222222
2222222
2222222
Thread lauched
```

```
3333333
3333333
3333333
33334444444
4444444
4444444
4444444
3333
3333333
3333333
3333333
```

Context switch may occurs in the middle of printf command - a type of race condition

NOTES

- In multi-threading, typically you have no control of output sequences or scheduling
- Context switching may occur in any time
  - the probility may be 1/10000000 but consider how many instruction CPU may execute per second
- Your program become concurrent and nondeterministic.
  - Same input to a concurrent program may generate different output
  - beware of the probe effect

race condition

Thread 1

```
…..
AddHead(list,B)
…..
```

```
AddHead(struct List *list,
        struct Node *node){
    node->next = list->head ;
    list->head = node;
}
```

Thread 2

```
…..
AddHead(list,C)
…..
```

race condition- if you think you are smart enough, things still go wrong

```
AddHead(struct List *list,
        struct Node *node){
   while (flag !=0)  ;
   flag = 1 ;
   node->next = list->head ;
   list->head = node;
   flag = 0;
}
```

```
        xor eax, eax ;
;  while (flag !=0)
L86:
        cmp _flag, eax
        jne L86
; flag = 1
        mov eax, _list$[esp-4]
        mov ecx, _node$[esp-4]
        mov _flag, 1
; node->next = list->head
        mov edx, [eax]
        mov [ecx], edx
;   list->head
        mov [eax] , ecx
 ; flag = 0
        mov _flag,0
```

What is C runtime library – multi-threaded version

- Original C runtime library is not suitable for multithreaded program
- Original C runtime library uses many global variables and static variables -> cause multithreads to race

NOTES

- conventional C library is not made to execute under multithreaded program
- choose the right C runtime library to link
  - /ML Single-threaded
  - /MT Multi-threaded (static)
  - /MD Multi-threaded DLL
  - /MLd Debug Single-threaded
  - /MTd Debug Multithreaded (static)
  - /MDd Debug Multithreaded (DLL)

Some suggestions

- avoid using global variables among threads
- do not share GDI objects between thread
- make sure you know the status of the threads you create, do not exit without waiting your threads to terminate
- have the main thread to handle UI (user interface)

Non Multithreading-Safe Code

- When two or more threads enter your code, your code function incorrectly
  - Using global (static) variables (races !!) but programmers are not aware of it
  - Two threads share the same instance of an object (races !!) but programmers are not aware of it.

Synchronization Problems

- There are kinds multithreaded applications which do not need synchronizations among threads.
- e.g., telnet and bbs daemon.
- if your threads do not need any kind of synchronization, such as telnet, you are lucky to get away with sync horror.
- Once you need synchronization, you invite concurrency errors to your program.
  - deadlock
  - race conditions
  - starvation

The Path of NO returns

- You have races !!
- You add synchronizations and thread coordination to fix races !!
- You introduce deadlock, live lock, starvation…

Concurrency errors

- are known to be
  - hard to detect (irreproducible)
  - hard to debug (probe effect)
  - hard to get a correct fix
- Currently, there is no effective way to detect, debug
- Your better choice is
  - be careful at the beginning (spend more time in design stage)
  - use the simplest synchronization structure that cannot be wrong or proved to be correct but not so efficient
  - do not use complicated synchronization structure unless you are sure what you are doing
  - sacrifices some performance in exchange of correctness.

Why concurrency errors are hard to reproduce, debug, and fix?

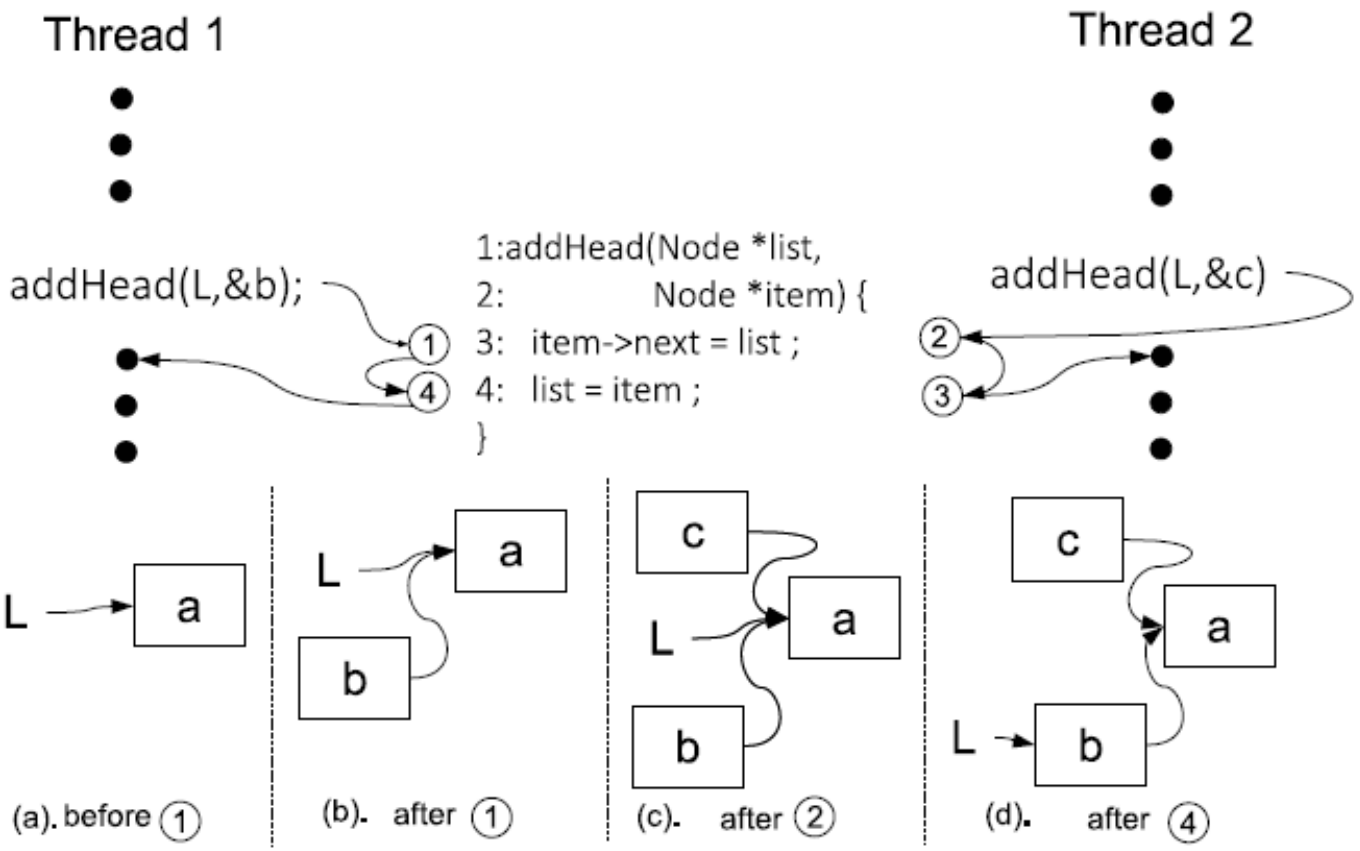


Figure 2: A linked-list with races.

Total interleaving permutation

- 1423
- 1234
- 1243
- 2143
- 2134
- 2314
- …..
- Typically exponential explosion !!
- You may finding a bug-triggering interleaving but cannot reproduce it deterministically because you have no control on O.S. scheduling and context switching
- Yea, we do have a newly developed secrete weapon. In the future, if you encounter concurrency error testing and debugging problem. Please come back to Prof. Cheng.

A Historical Review of Synchronization

Shared memory synchronization problem Arises
Support hardware sync instruction

The concept of Monitor
Never supported by O.S. system calls

Timesharing multiprocesses

O.S. supports InterProcess Communications
- semaphore (mutex lock)
- message queue
- pipe

Concurrency bugs are nightmare
Concurrency design is in general too difficult for programmers
Only kernel people play with them

Multithreading
- new thread synchronizations
- Monitor are supported in programming language (java)
And usually with a new name
Called synchronized object
- Let the expert do the synchronization In side the shared object
- Clients just use them without knowing the synchronization details

Using semaphores for solving critical section problems

- For n processes
- Initialize S.count to 1
- Then only 1 process is allowed into CS (mutual exclusion)
- To allow k processes into CS, we initialize S.count to k

```
Process Pi:
repeat
   P(S);
    CS
   V(S);
    RS
forever
```

Problems with semaphores

- semaphores provide a powerful tool for enforcing mutual exclusion and coordinate processes
- But P(S) and V(S) are scattered among several processes. Hence, difficult to understand their effects
- Usage must be correct in all the processes
- One bad (or malicious) process can fail the entire collection of processes

IMPORTANT:Suggestion and Trend

- The current trend of synchronization is called synchronized shared objects
- gather all the semaphore, mutex, critical sections in an shared object's methods
- Clients need not worry about the synchronization details inside the synchronized shared objects.
- Let the shared object implementer (usually an experienced expert in concurrent programming) worry about the synchronization

Synchronization via a shared object in Java

```
class PRODUCER
extends Thread {
buffer b ; // the shared object
PRODUCER(buffer _b)
{ b = _b ; }
run() {
    do {
        data = generate_a_data();
        b.append(data);
    } while (1);
}
```

```
class buffer {
    ?
    }
```

```
class CONSUMER
extends Thread {
buffer b ; // the shared object
CONSUMER(buffer _b)
{ b = _b ; }
run() {
    do {
        b.take(data);
        output data ;
    } while (1);
}
```

```
main() {
   buffer rb = new buffer() ;
   Thread p = new PRODUCER(rb);
   Thread c = new CONSUMER(rb);
   p.start();
   c.start();
```

Let’s do it with hand

Appendix – Win32 Synchronization

- The followings are the Win32 synchronization.
- New thread synchronization please refer to the books.

CloseHandle

```
int main() {
      HANDLE hThrd ;
      DWORD threadid ;
      int i ;
      for (i=0;i<5;i++) {
        hThrd =
          CreateThread(NULL,0,ThreadFunc,(LPVOID)I,0,&threadid);
        if (hThr) {
                printf("Thread launched %d\n",i);
                CloseHandle(hThrd)
      }
      Sleep(2000);
      return EXIT_SUCCESS;
}
```

CloseHandle

- Thread is a kernel object
- A kernel object can be owned by several owners
- Each thread when created is owned by two owner- the creator and the thread itself. So, the reference count is two.
- When reference count is down to zero, windows destroy the object.
- if a process generate many threads but do not close the handle, this process may own many kernel thread object -> resource leaks

Misc Win32 function

- BOOL GetExitCodeThread
  - A non-blocked system call to acquire the status of a thread
- VOID ExitThread
- DWORD WaitForSingleObject
  - avoid busy waiting
  - e.g. WaitForSingleObject(hThrd, INFINITE);
- DWORD WaitForMultipleObjects
  - same as WaitForSingleObject

Synchronization mechanism in Win32

- Critical sections
- Mutexes
- Semaphore

Critical Sections

- VOID InitializeCriticalSection
- VOID DeleteCriticalSection
- VOID EnterCriticalSection
- VOID LeaveCriticalSection

```
typedef struct _Node {
      struct  _Node *next ;
      int data ;
} Node ;
typedef struct _List {
      Node *head ;
      CRITICAL_SECTION
    critical_sec ;
} List ;
```

```
List *CreateList() {
    List *plist = malloc(sizeof(pList));
    pList-> head = NULL ;
    InitializeCriticalSection(&pList->critical_sec);
    return pList ;
}
```

```
AddHead(struct List *plist,
          struct Node *node){
    EnterCriticalSection(&plist->critical_sec);
    node->next = list->head ;
    list->head = node;
    LeaveCriticalSection(&plist->critical_sec):
}
```

Thread 1

…..
AddHead(list,A)
……

Thread 2

……
AddHead(list,B)
……

```
AddHead(struct List *plist,
        struct Node *node){
    EnterCriticalSection(&plist->critical_sec);
    node->next = list->head ;
    list->head = node;
    LeaveCriticalSection(&plist->critical_sec):
}
```

Continued

- There is at most one thread can manipulate the linked list
- do not use sleep() or Wait..() in a critical section -> cause deadlock
- if a thread enter a critical section and exit, the critical section is locked by Window NT

Deadlock

```
void SwapLists(List *list1, List *list2) {
   List *tmp_list ;
   EnterCriticalSection(list1-> critical_sec);
   EnterCriticalSection(list2->critical_sec);
   tmp_list = list1->head
   list1->head = list2->head
   list2->head = tmp_list ;
   LeaveCriticalSection(list1->critical_sec);
   LeaveCriticalSection(list2->critical_sec);
}
```

- Holding a resouce and then request another resource
  -> possible deadlock

Thread 1

……

Swap_lists

……

……

```
void SwapLists(List *list1, List *list2) {
   List *tmp_list ;
   EnterCriticalSection(list1-> critical_sec);
   EnterCriticalSection(list2->critical_sec);
   tmp_list = list1->head
   list1->head = list2->head
   list2->head = tmp_list ;
   LeaveCriticalSection(list1->critical_sec);
   LeaveCriticalSection(list2->critical_sec);
}
```

Thread 2

……

……

Swap_lists

…….

…….

Mutex (MUTual EXclusion)

- Similar to critical section
- cause 100 times of time to lock a process/thread
  - CS is executed in user mode (only within a process)
  - mutex is executed in kernel mode , it can cross process (interprocess communication)

Comparision of CS/Mutex

- VOID InitializeCriticalSection
- VOID EnterCriticalSection
- VOID LeaveCriticalSection
- VOID DeleteCriticalSection

- CreateMutex
- OpenMutex
- WaitForSingleObject()
- WaitForMultipleObject()
- MsgWaitForMultipleObject
- ReleaseMutex
- CloseHandle

```
typedef struct _Node {
      struct  _Node *next ;
      int data ;
} Node ;
typedef struct _List {
      Node *head ;
      HANDLE hMutex ;
} List ;
```

```
List *CreateList() {
    List *plist = malloc(sizeof(pList));
    pList-> head = NULL ;
    plist->hMutex = CreateMutex(NULL,false,NULL)
    return pList ;
}
```

```
void SwapLists(List *list1, List *list2) {
   List *tmp_list ;
   HANDLE arrhandles[2] ;
   arrhandles[0] = list1-> hMutex ;
   arrhandles[1] = list2-> hMutex ;
   WaitForMultipleObjects(2, arrhandles, TRUE, INFINITE);
   tmp_list = list1->head
   list1->head = list2->head
   list2->head = tmp_list ;
   ReleaseMutex(arrhandles[0]);
   ReleaseMutex(arrhandles[1]);
}
```

NOTES

- When a mutex is created, no thread/process have a lock on it, it is ready to trigger
  - So, a call to Wait..() will immediately return, otherwise a call to Wait…() will be blocked.
- mutex is exactly the same as a semaphore with semaphore value 1.

Semaphore

- Synchronization tool (provided by the OS) that do not require busy waiting
- A semaphore S is an integer variable that, apart from initialization, can only be accessed through 2 atomic and mutually exclusive operations:
  - P(S), down(S), wait(S)
  - V(S), up(S), signal(S)
- To avoid busy waiting: when a process has to wait, it will be put in a blocked queue of processes waiting for the same event

Semaphores in Win32

- HANDLE CreateSemaphore
- void WaitForSingleObject (like P())
- BOOL ReleaseSemaphore (like V())

- They are the same as the semaphore taught in O.S.
- use Wait…() to decrease a semaphore or block a thread when its value is zero.

Semaphore implementation

```
P(S):
  S.count--;
  if (S.count<0) {
    block this process
    place this process in S.queue
  }

V(S):
  S.count++;
  if (S.count<=0) {
    remove a process P from S.queue
    place this process P on ready list
  }
```

S.count must be initialized to a nonnegative value (depending on application)