CS222

Introduction to Multithread Programming

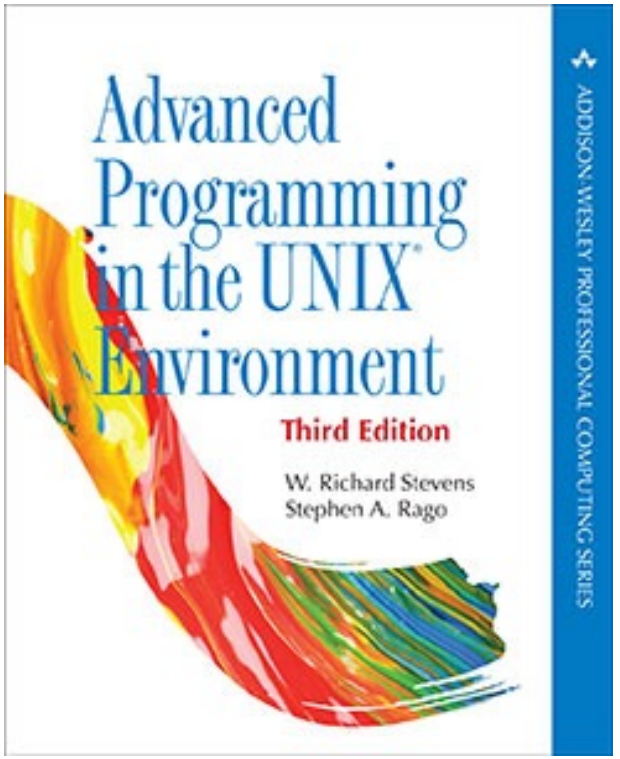
2nd Semester 2024

Kasidit Chanchio

Department of Computer Science

Textbook

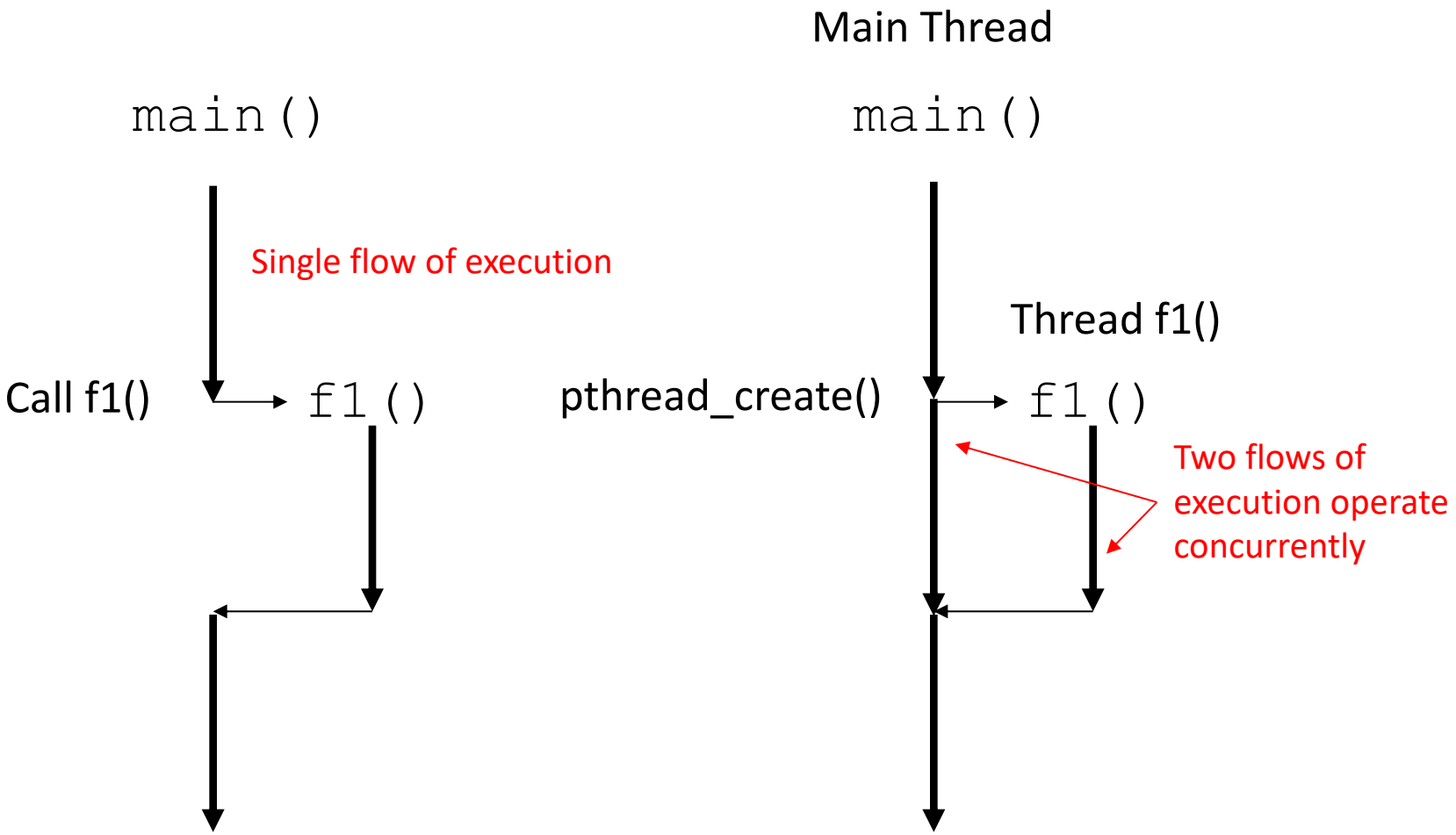
- Advanced Programming in the UNIX Environments
- http://www.apuebook.com/



Thread

- Thread คือการอนุญาตให้สร้าง ***multiple flow of control ใน*** process
- เรามักจะสร้างเทรดขึ้นมาเพื่อรองรับเหตุการณ์หลายเหตุการณ์ไปพร้อมๆกัน
- ในขณะที่โพรเซสจะแยกหน่วยความจำกัน เทรดเป็นสิ่งที่อยู่ในโพรเซสเดียวกัน มันจะแชร์หน่วยความจำของโพรเซสร่วมกัน และแชร์ไฟล์ที่โพรเซสเปิดด้วย
- เราสามารถสร้างเทรดหลายเทรดให้ประมวลผลบนหลาย CPU cores ไปพร้อมกันเพื่อให้โพรเซสผลิตผลงานได้มากขึ้น
- นอกจากนั้น เรายังสามารถสร้างเทรดหลายเทรด และกำหนดให้แต่ละเทรดทำหน้าที่ต่างกันเช่น เทรดหนึ่งติดต่อผู้ใช้ อีกเทรดหนึ่งประมวลผล อีกเทรดหนึ่งอ่านเขียนข้อมูลกับอุปกรณ์ I/O เป็นต้น

Process and Threads



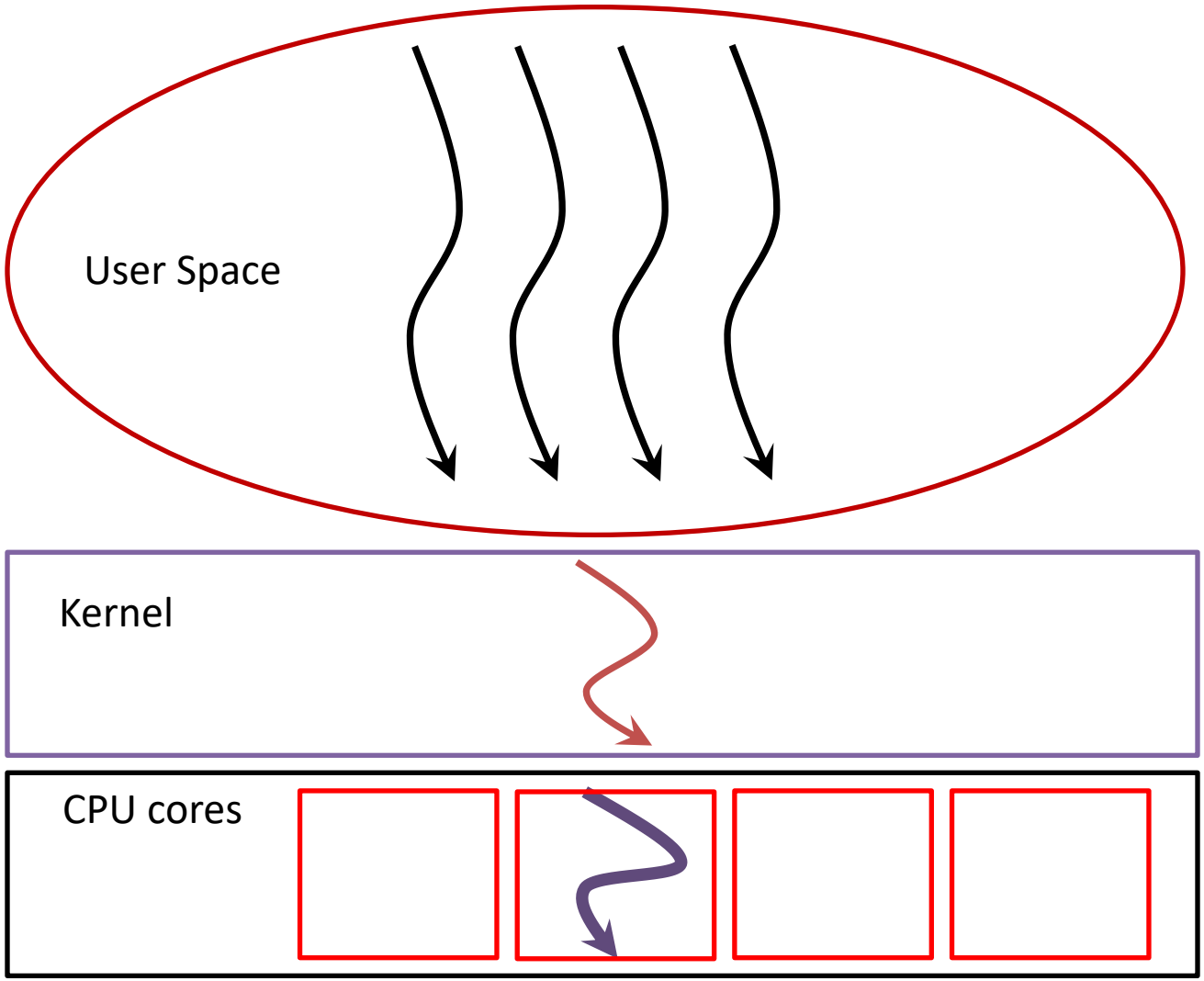
Types of Thread

- There are 3 types:
  - ***User-Level Thread,***
  - ***Kernel-Level Thread***, and
  - *Light-weighted Process (Many to Many Model)*
- Remind that process is preemptive in multiprogramming OS.
- ***Process context switching is costly*** due to the isolation principle where each process has separate memory space.
- Process's context switching costs include:
  - Save/restore register contents
  - Changing page table, caching, virtual memory
  - TLB flushing

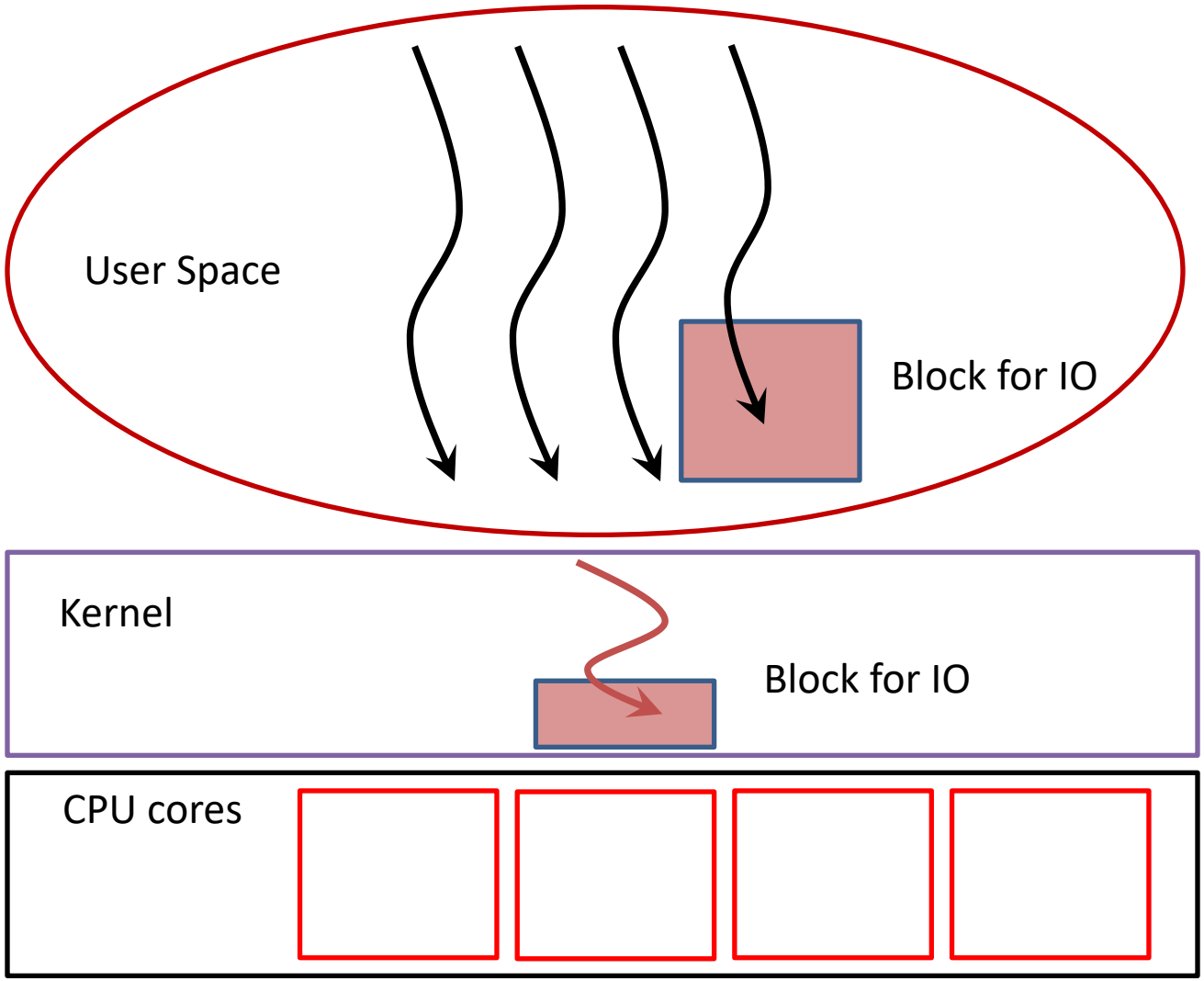
1. User-Level Thread

- ***Thread scheduling is implemented in process user space***. Threading is emulated in process.
- Thus, context switching is low:
  - ***Context switching is performed in user space***: no need to change CPU execution mode to access kernel's memory
- Disadvantages:
  - A blocking I/O suspends the entire process, even though another thread is ready to run
  - Cannot exploit multi-processing capability of HW

1. User-level Thread



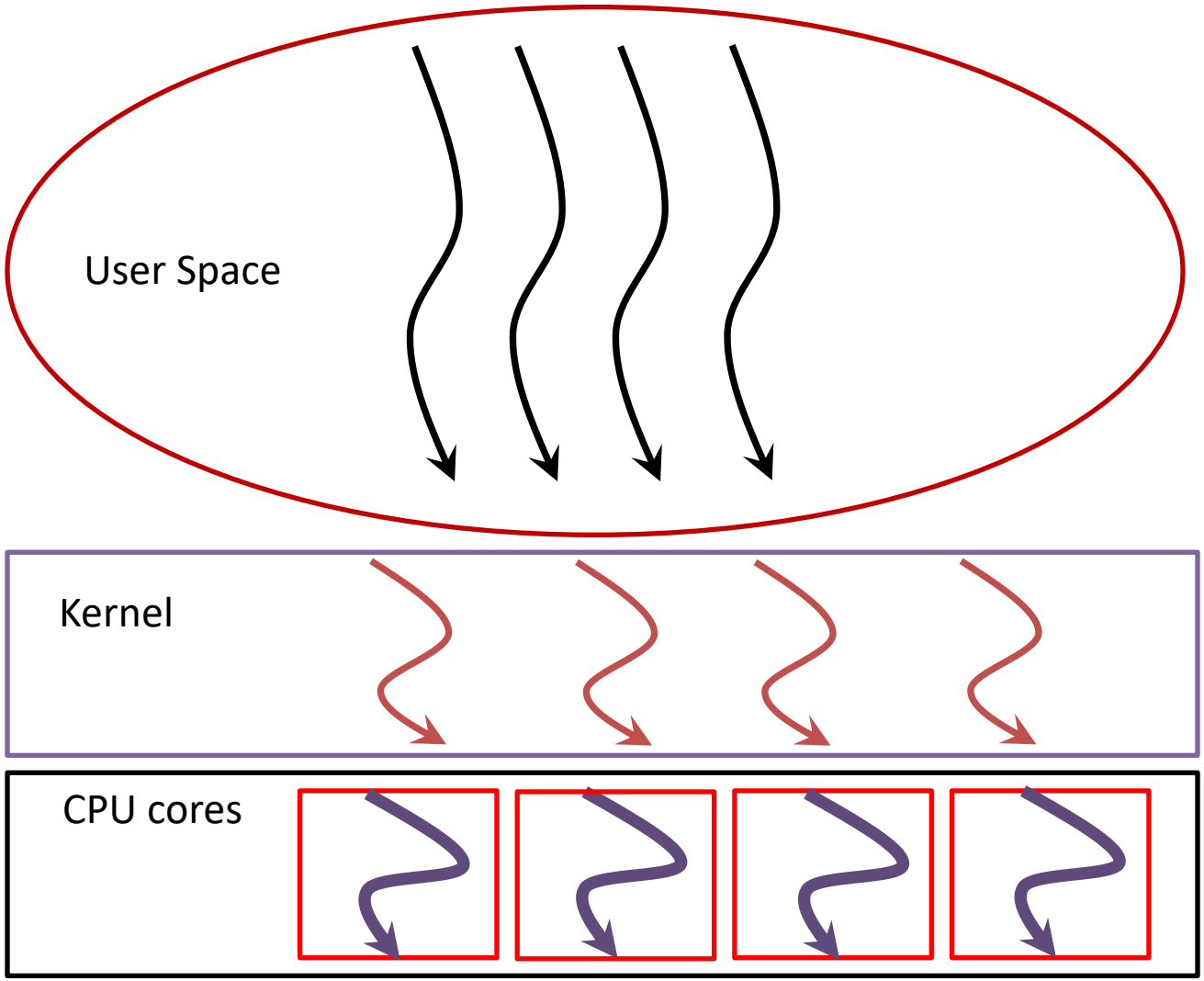
1. User-level Thread



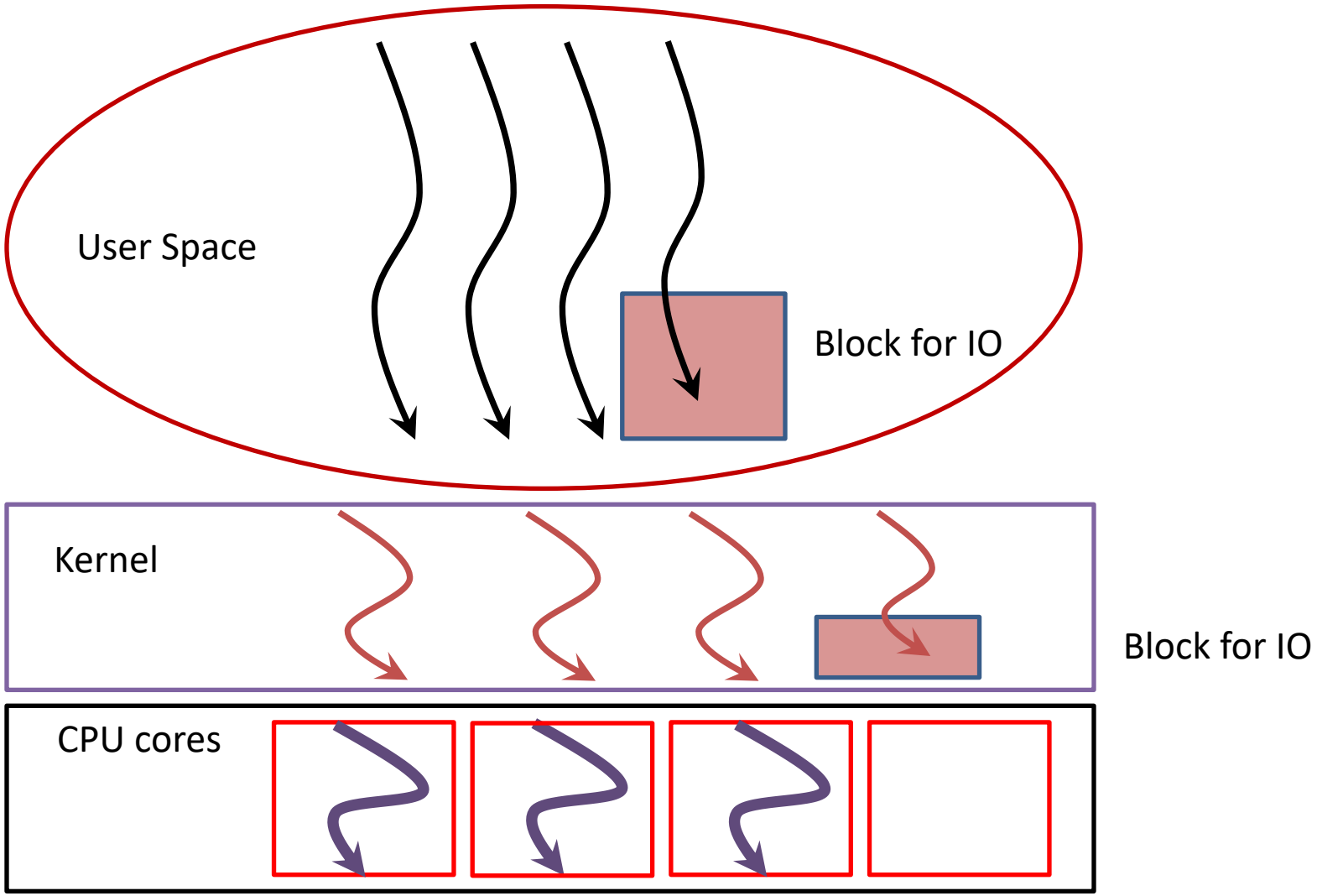
2. Kernel-Level Thread

- *Thread scheduling is implemented in kernel.*
- Disadvantages:
  - ***context switching is cost is high*** but not as much as process's context switching.
- Advantages:
  - *When a thread is blocked for I/O, another thread can run*
  - *Multiple threads on the sane process can run concurrently on different CPU cores or processors*

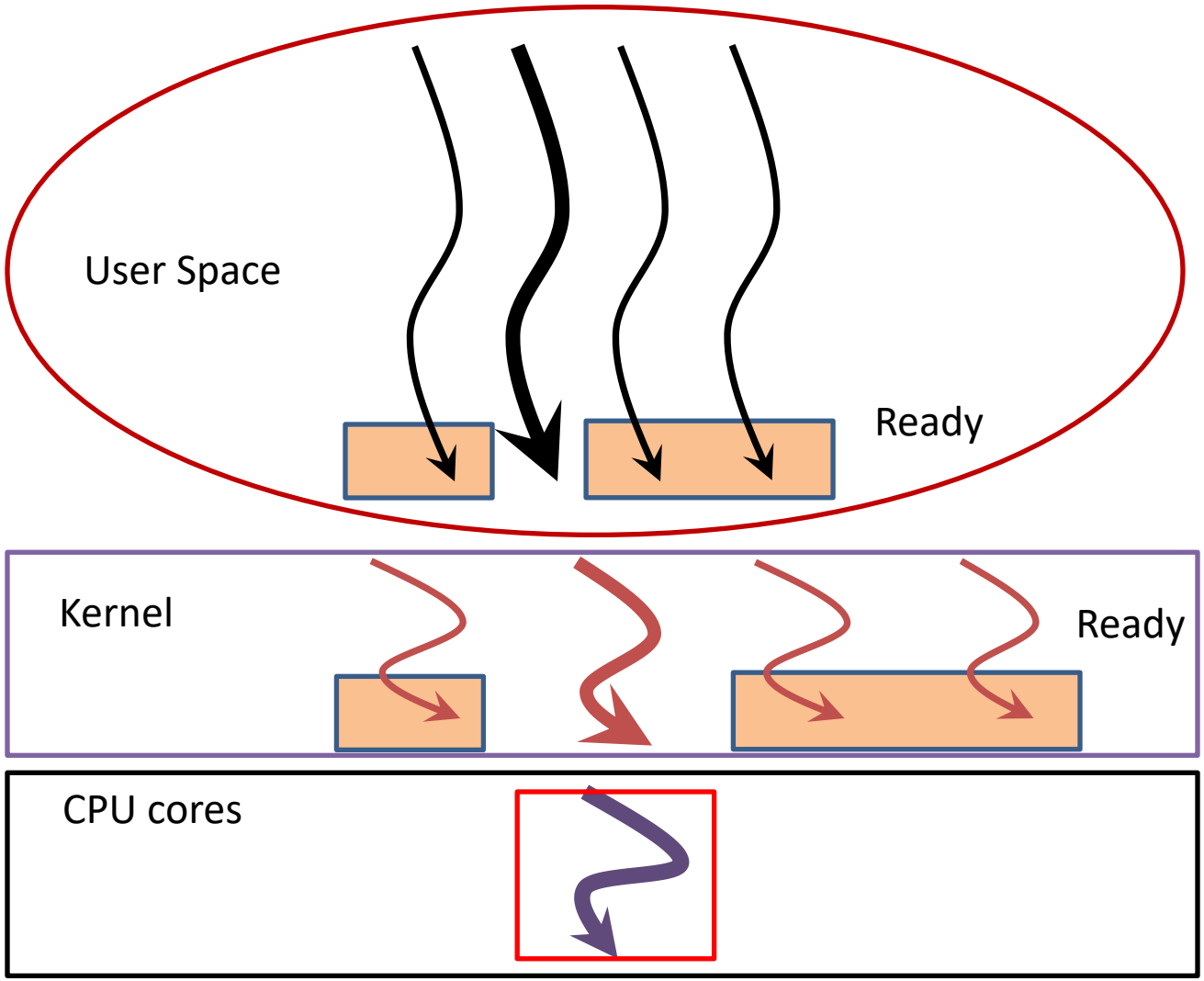
2. Kernel-level Thread



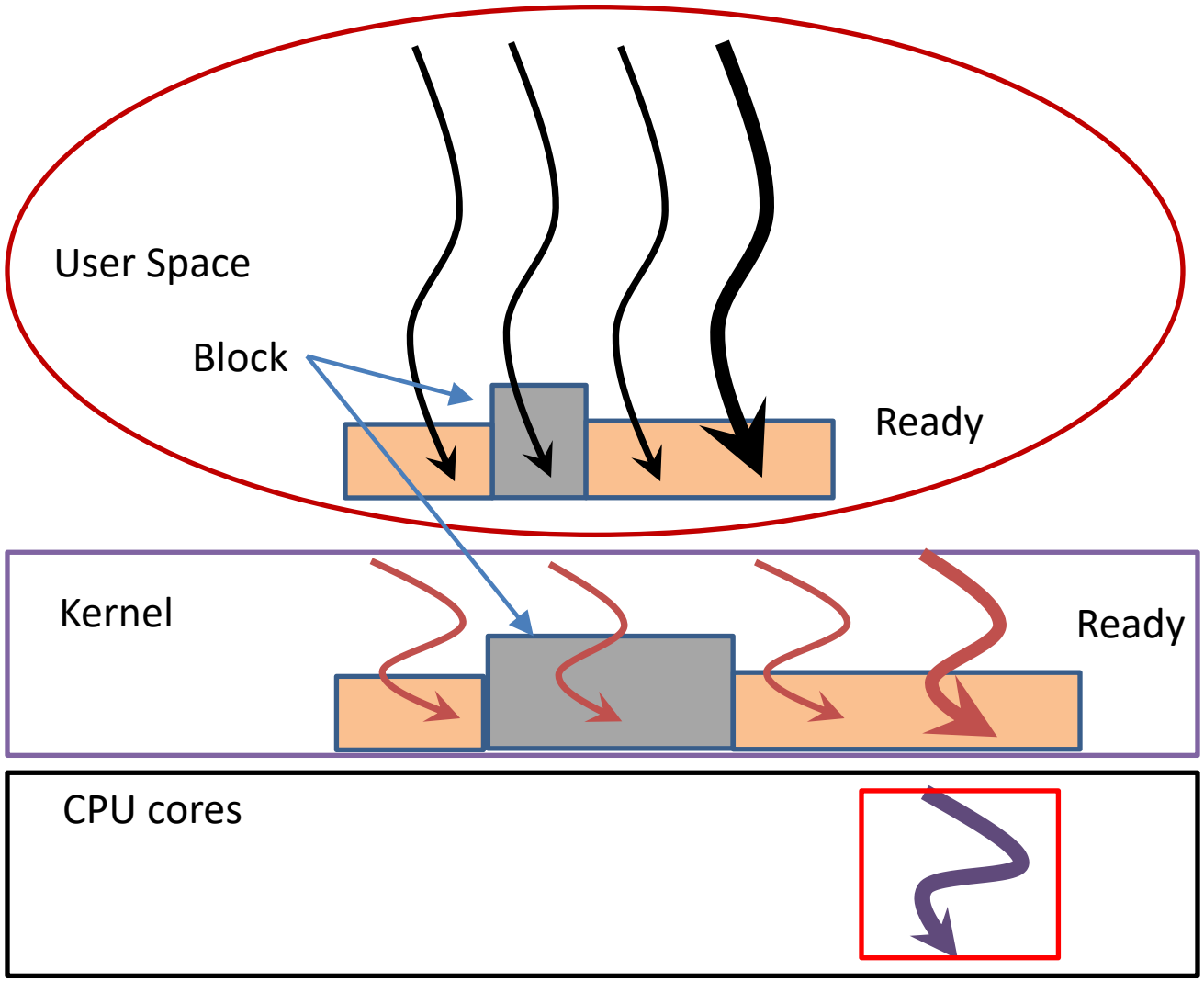
2. Kernel-level Thread



2. Kernel-level Thread



2. Kernel-level Thread



Linux Threads

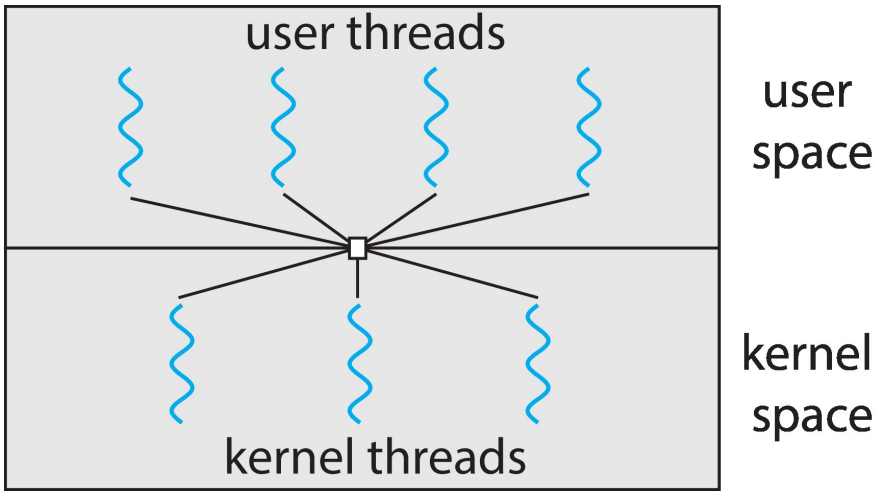
- Linux refers to them as ***tasks*** rather than ***threads***
- Thread creation is done through **clone()** system call
- **clone()** allows a child task to share the address space of the parent task (process)
  - Flags control behavior

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

- **struct task_struct** points to process data structures (shared or unique)

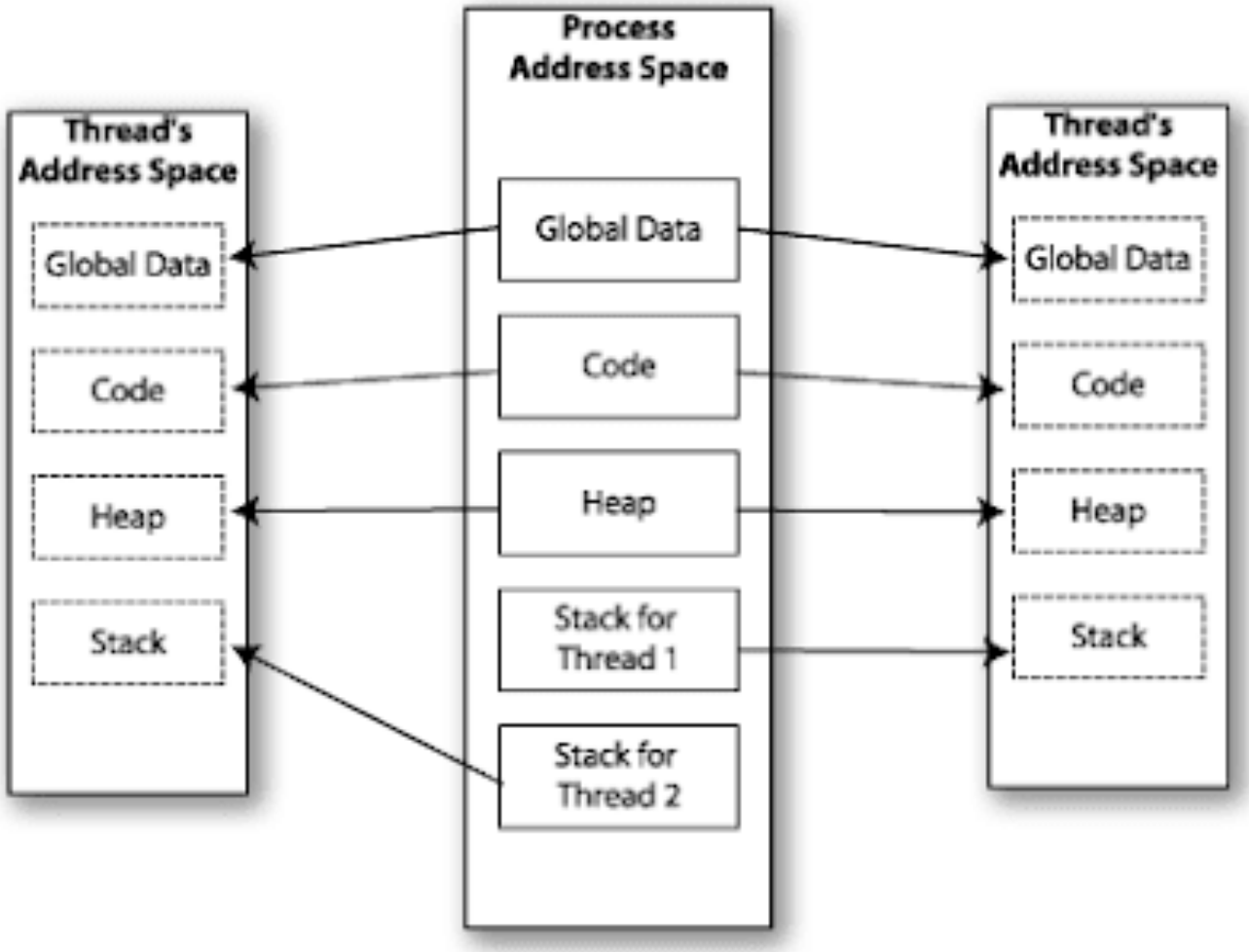
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Windows with the *ThreadFiber* package
- Otherwise not very common



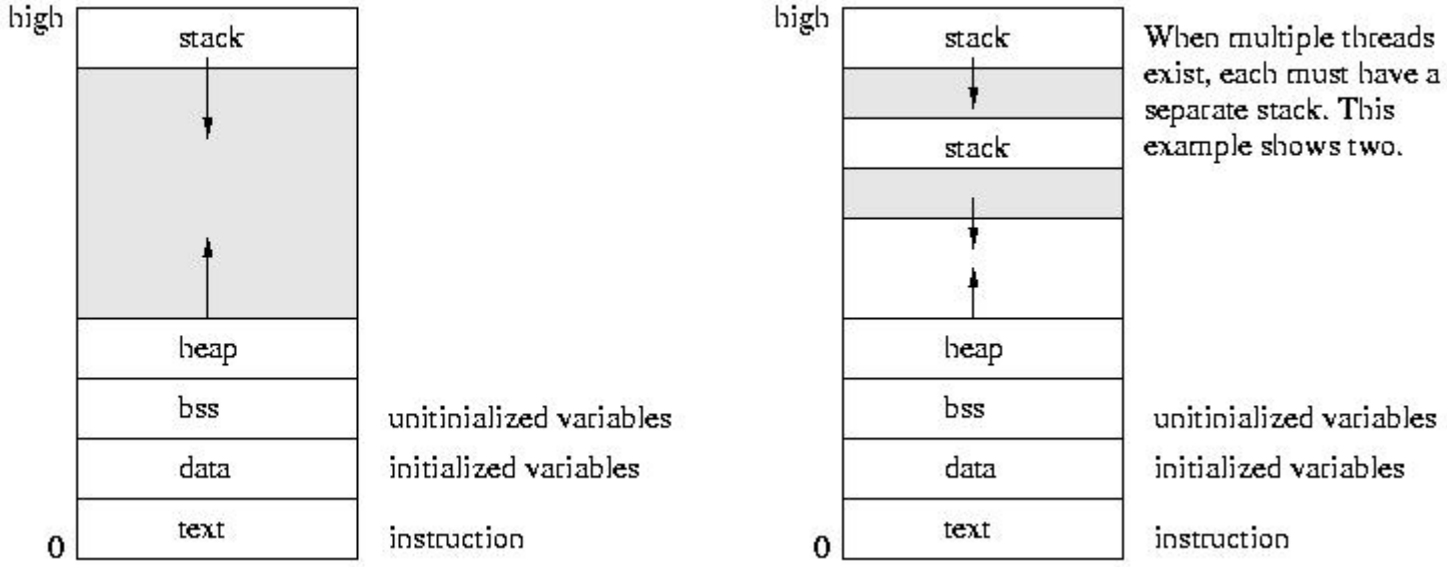
Memory Layout

- Share global, code, heap
- Divide stack



Process Memory Layout

- Threads share global data, code, and heap segments of a process.
- Each thread has a separate stack segment.



Reference: http://www.andrew.cmu.edu/course/15-310/applications/ln/lecture17.html

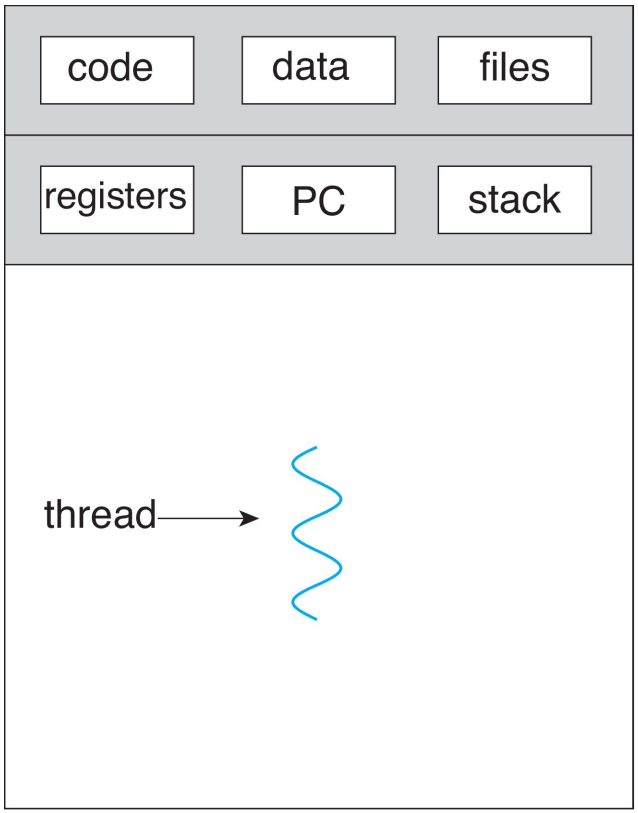
Global variables

- Can introduce subtle and confusing bugs!
- Limit use of global variables to situations in which they're really needed.
  - Shared variables.

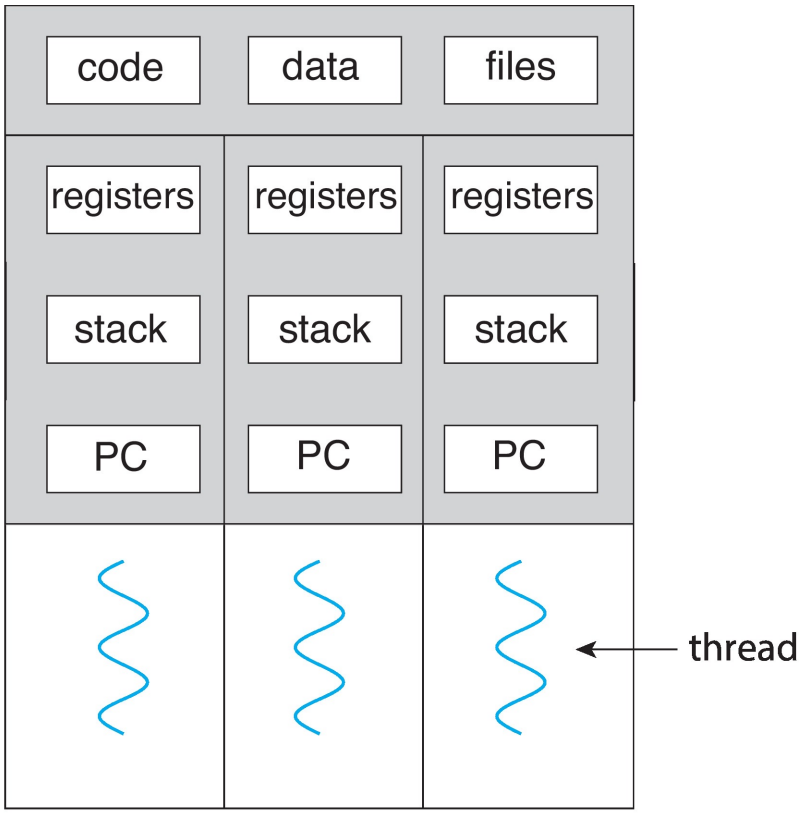
Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded
  - ใน OS เช่น FreeBSD หลังจากโปรเซส 0 สร้างโปรเซส 1 (init) มันจะสร้าง threads ขึ้นจำนวนหนึ่งเพื่อดูแลการใช้งานระบบคอมพิวเตอร์

Single and Multithreaded Processes

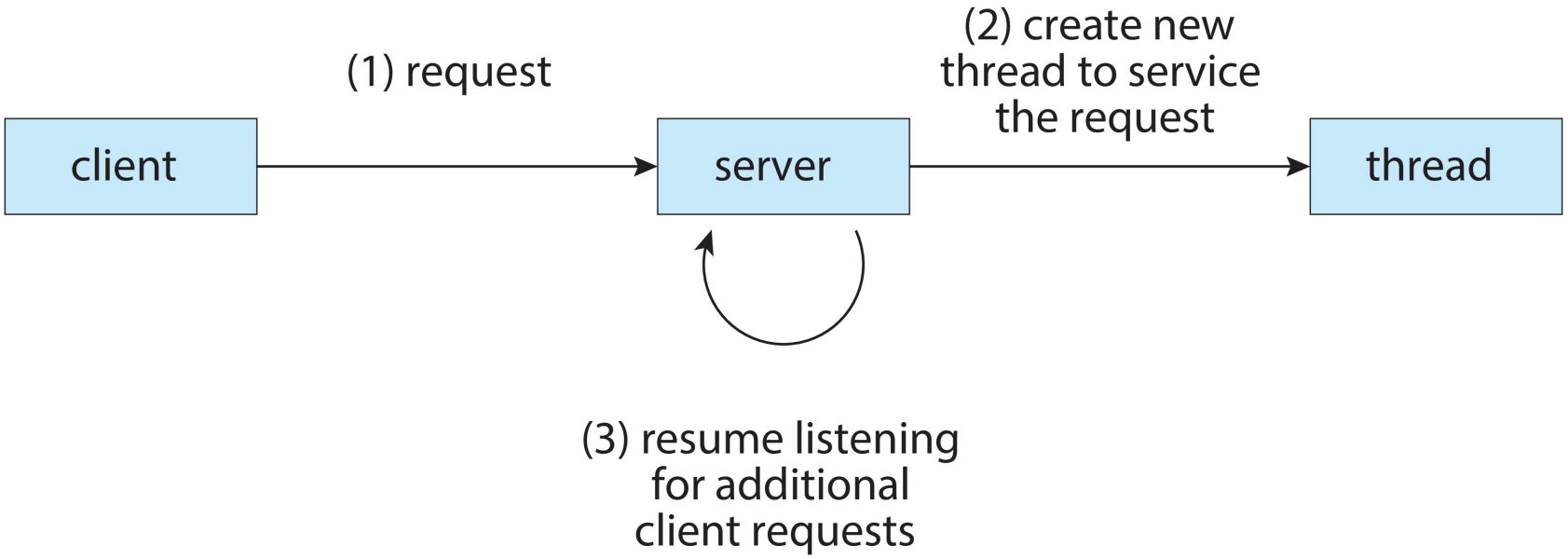


single-threaded process



multithreaded process

Multithreaded Server Architecture



Benefits

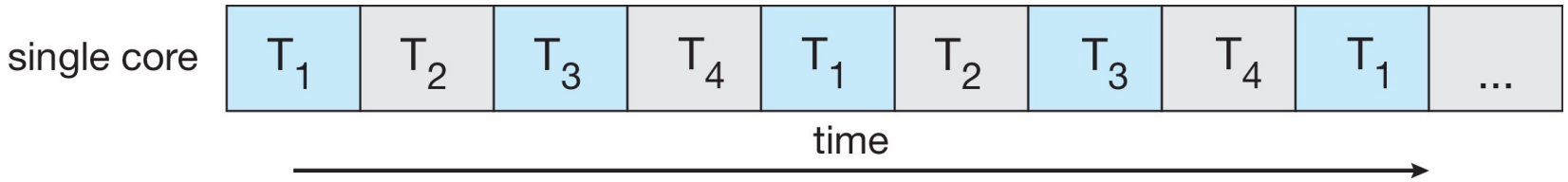
- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing** – threads share resources of process, easier than shared memory or message passing
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching
- **Scalability** – process can take advantage of multicore architectures

Multicore Programming

- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

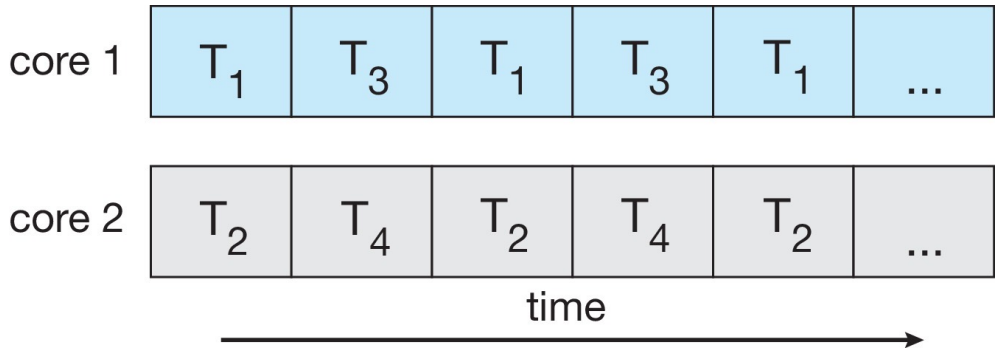
Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**



  - ทุกโปรเซสมีความคืบหน้า

- **Parallelism on a multi-core system:**

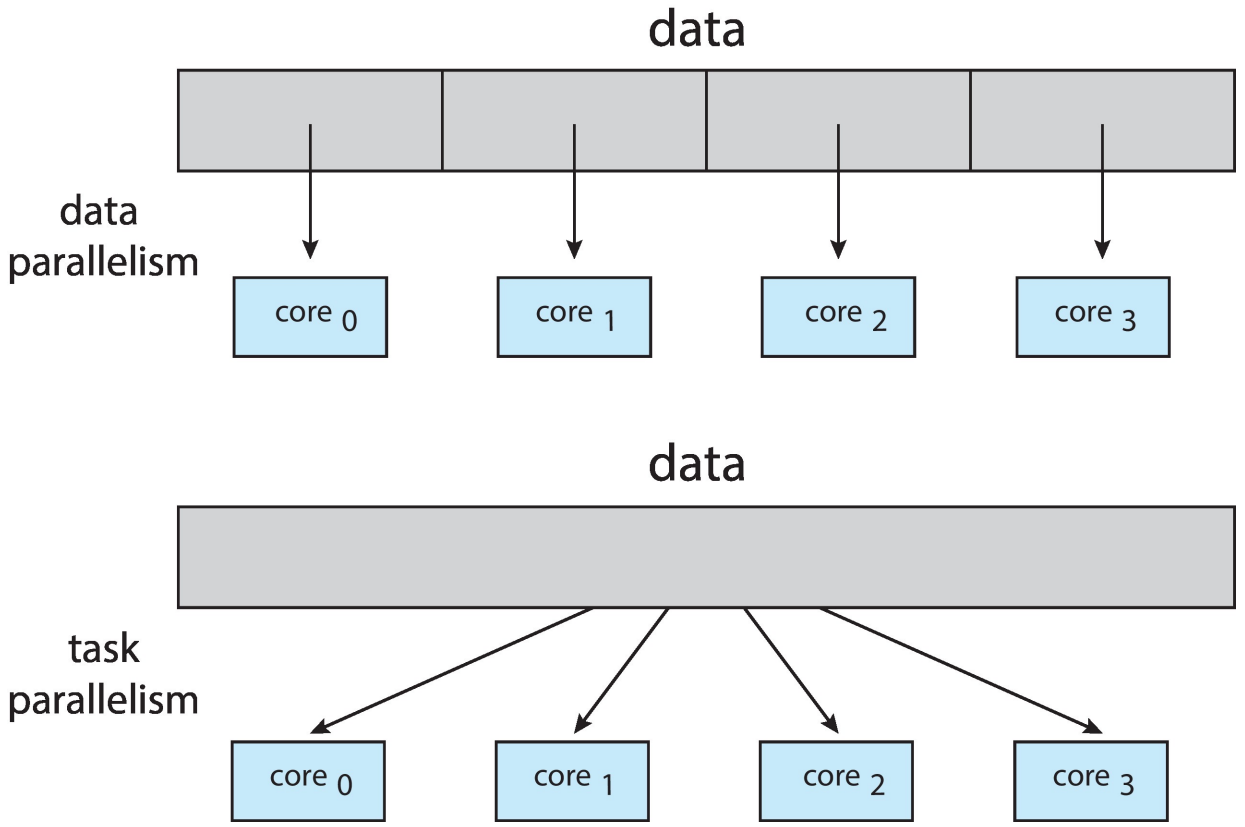


  - หลายโปรเซสทำงานไปพร้อมกันจริงๆ

Multicore Programming

- Types of parallelism
  - Data parallelism - distributes subsets of the same data across multiple cores, same operation on each
  - Task parallelism - distributing threads across cores, each thread performing unique operation

Data and Task Parallelism



ธรรมชาติของการประมวลผลแบบขนาน (parallel)

- สมมุติว่า นศ มีบริษัทก่อสร้าง มีคนงาน 10 คน
- บริษัทสร้างบ้านหนึ่งหลัง ใช้เวลา 1 เดือน
- เปรียบเหมือน นศ มี CPU cores 10 cores (หรือ 10 processors)
- มีงาน W work และใช้เวลา T time จึงทำงานเสร็จ
- ถ้ามีคนงานเร่งอยากจะได้บ้านเร็วมาก แล้ว นศ เริ่มคำนวณ
- สร้างบ้าน 1 หลัง คนงาน 10 คน ใช้เวลา 720 ชม (30 วัน)
  - แสดงว่างานรวมถ้าทำคนเดียวต้องใช้เวลา 720x10 = 7200 ชม
- ดังนั้น ถ้าอยากให้บ้านเสร็จใน 1 ชม ก็ต้องใช้คน 7200 คน

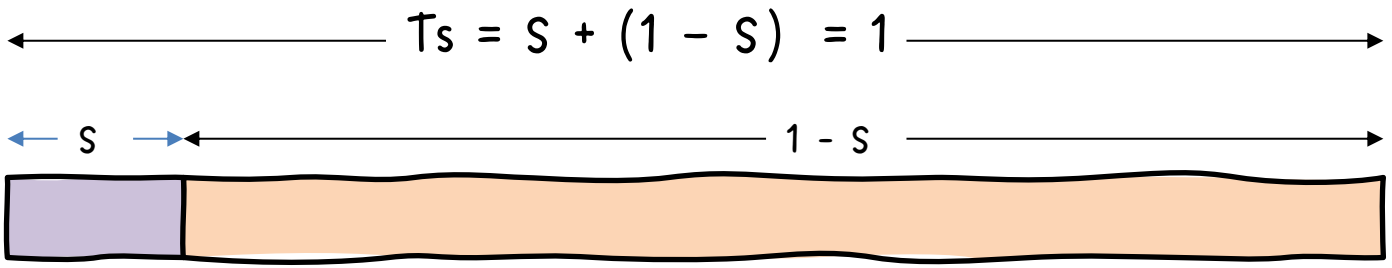
ธรรมชาติของการประมวลผลแบบขนาน (parallel)

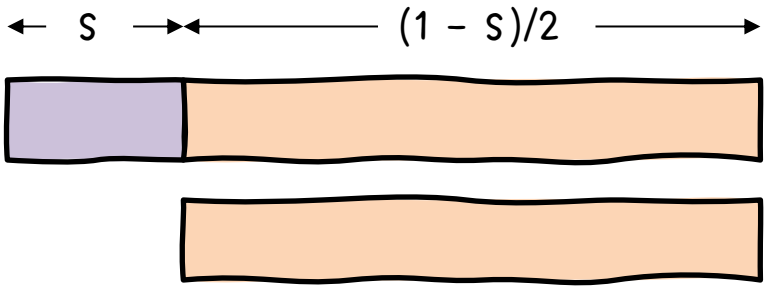
- ถ้าอยากให้บ้านเสร็จภายใน 1 นาที ต้องใช้ 7200x60 คน ????
- เหมือน นศ มีงานกลุ่มที่ทำ 3 คนเสร็จใน 1 อาทิต ถ้าเพิ่มคน 1000 คนจะเสร็จเร็วขึ้นไหม
- จะเห็นว่าเป็นไปไม่ได้ เพราะ ในการทำงานมีงานสองประเภท
  - งานที่แบ่งกันไปทำได้
  - งานที่แบ่งกันไม่ได้ต้องทำงานคนเดียว
- ในการทำงานแบบขนานทำให้เกิดงานแบบใหม่ (ไม่มีในการทำงานคนเดียว)
  - การแบ่งงานและบริหารจัดการ
  - การรอกัน
  - การสื่อสารประสานงาน
  - การรวบรวมผลงาน

Amdahl's Law

- สมมุติว่าเวลาที่ใช้ประมวลผลแบบ sequential คือ 1
- สมมุติว่างานที่แบ่งไม่ได้คือ S

- *Work = 1*
- *S* is serial portion
- *N* processing cores

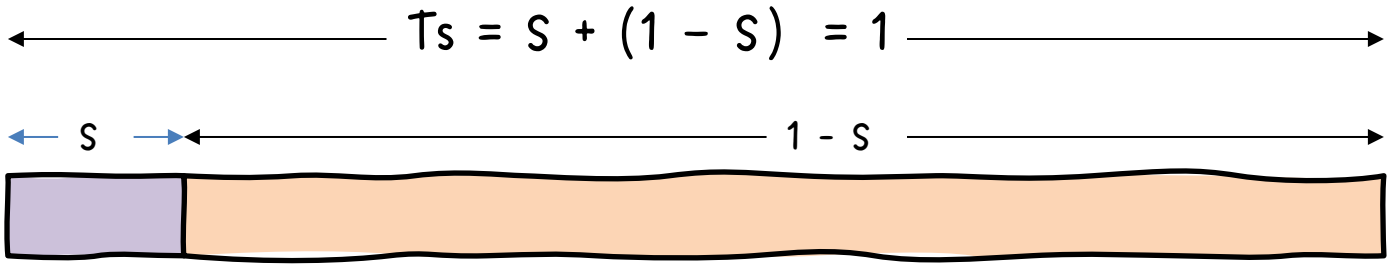




$$Tp = S + (1 - S)/N = 1$$

https://en.wikipedia.org/wiki/Amdahl%27s_law

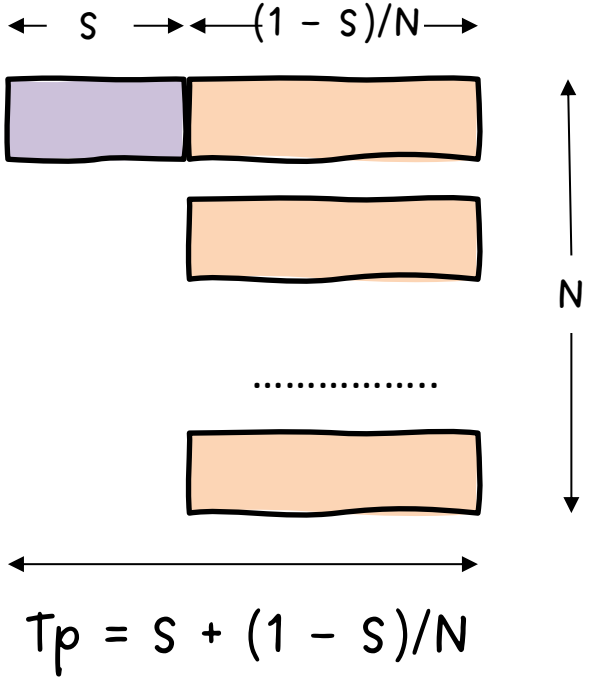
Amdahl's Law



- Speedup S(N) เมื่อใช้ N processors แล้วเร็วขึ้นกี่เท่า
- S(N) = Ts / Tp

$$= \frac{1}{S+\frac{(1-s)}{N}}$$

- เมื่อ N สูง S(N) → 1/S



Amdahl's Law

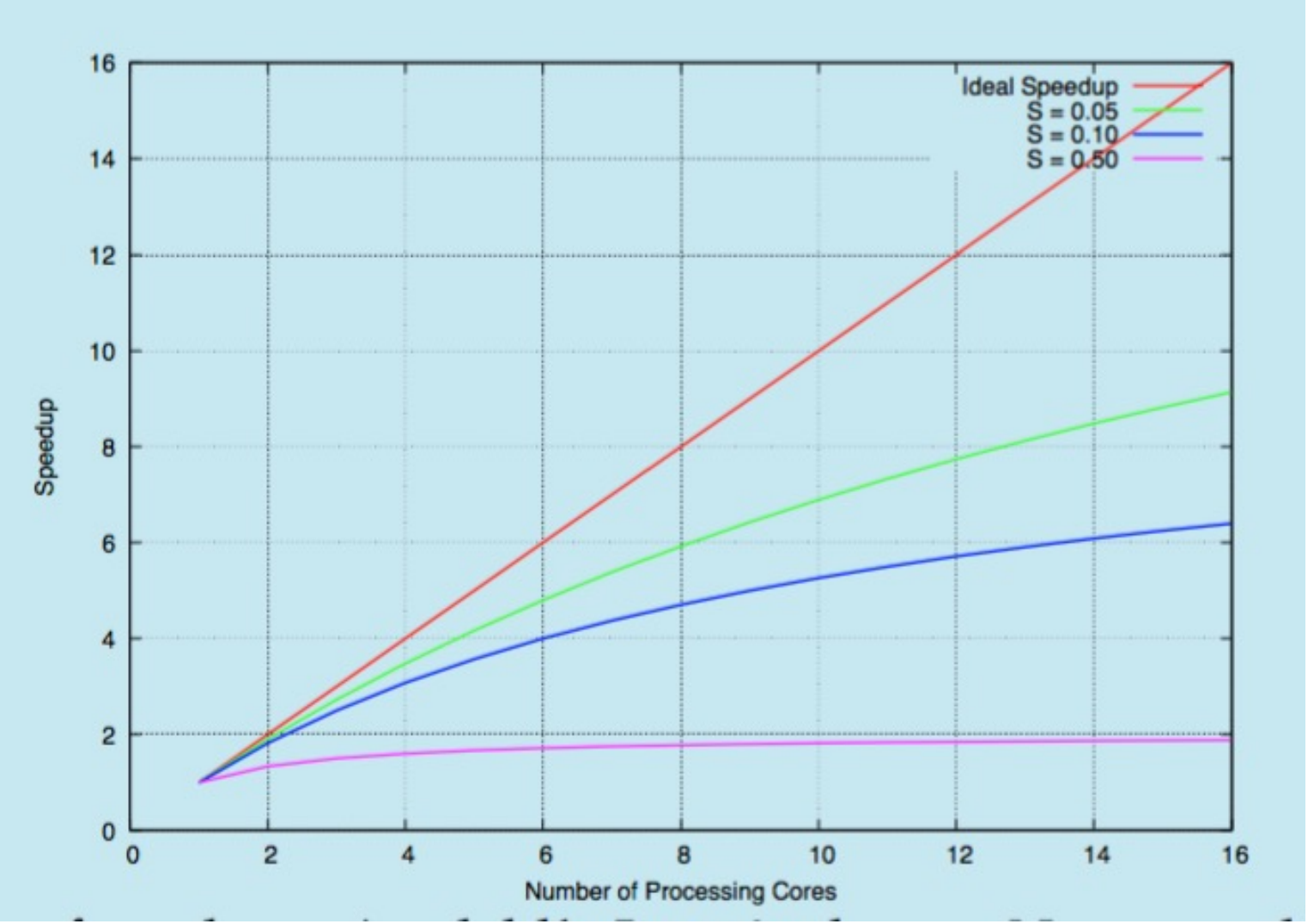
- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- *S* is serial portion
- *N* processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As *N* approaches infinity, speedup approaches 1 / *S*

Serial portion of an application has disproportionate effect on performance gained by adding additional cores

Amdahl's Law


Ideal Speedup
S = 0.05
S = 0.10
S = 0.50
Speedup
Number of Processing Cores
0
2
4
6
8
10
12
14
16

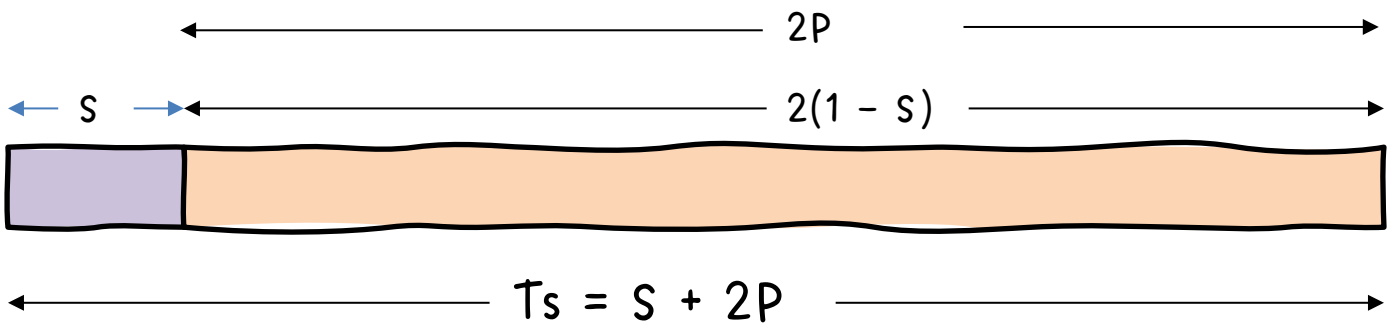
ธรรมชาติของการประมวลผลแบบขนาน (parallel)

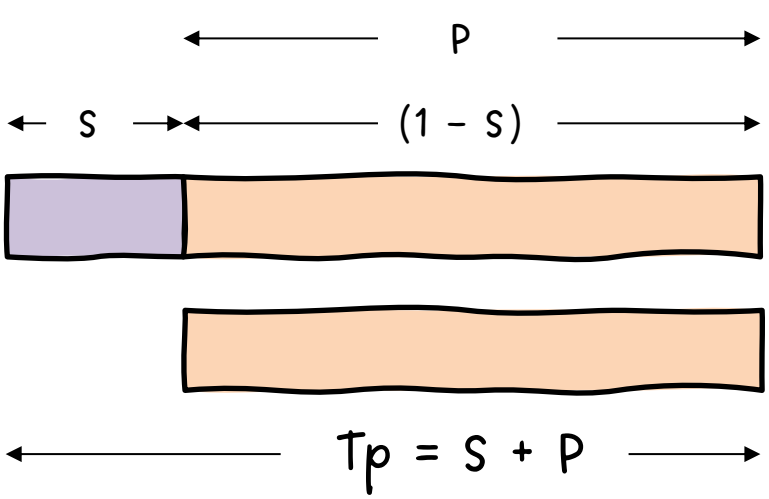
- จากกฎของ Amdahl การเพิ่มจำนวน processors ไม่ได้ช่วยให้ประมวลผลงานได้เร็วขึ้นได้เสมอไป
- กฎนี้ทำให้ จำนวน processors ที่ถูกพัฒนาขึ้นใน parallel computers ในช่วงปี 1980 – 1990 มีจำนวน processor ไม่เกิน 256 processors
- ทำไมในปัจจุบัน Supercomputer ถึงมีจำนวน processors เป็น แสน processors และทำไมจำนวน cores บน CPU ระบบ server เพิ่มขึ้นตลอด
- ในช่วงต้นทศวรรต 1990 มี **Gustafson's law** เกิดขึ้นและเกิดมาตรวัดใหม่เรียกว่า **Scalability**
- หลักการคือ “จะเพิ่ม processor ก็ต้องเพิ่มขนาดของปัญหาด้วย ถึงจะคุ้ม”
- นศ ไม่เพิ่มคนงานจาก 10 คน เป็น 1000 เพื่อสร้างบ้านหลังเดียว แต่เพิ่มขึ้นให้สร้างหมู่บ้านจัดสรรที่มีบ้านเป็น หลายสิบหรือหลายร้อยหลัง
- https://en.wikipedia.org/wiki/Gustafson%27s_law

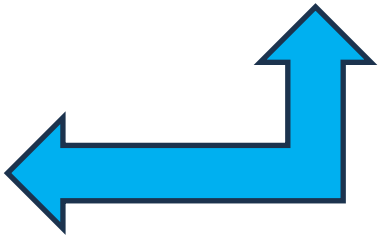
Gustafson's Law

- สมมุติว่าเวลาที่ใช้ประมวลผลแบบ sequential คือ 1
- สมมุติว่างานที่แบ่งไม่ได้คือ S
- ให้ P = (1 – S)

- *Work = 1*
- *S is serial portion*
- *N processing cores*

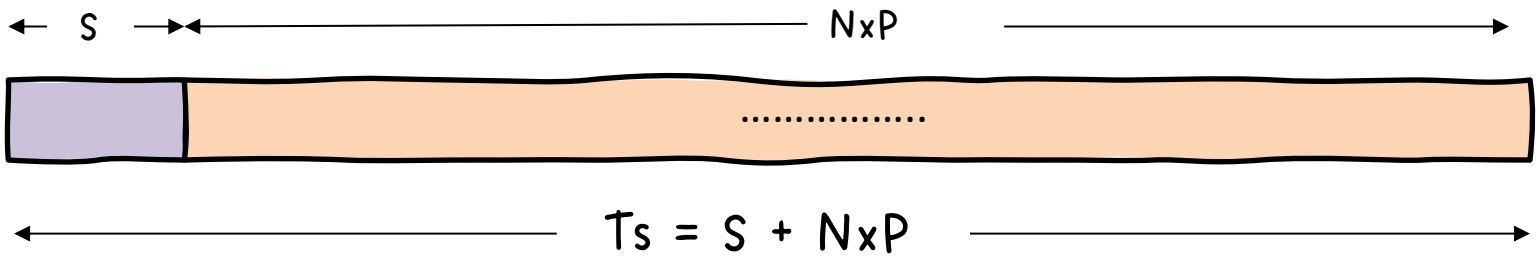


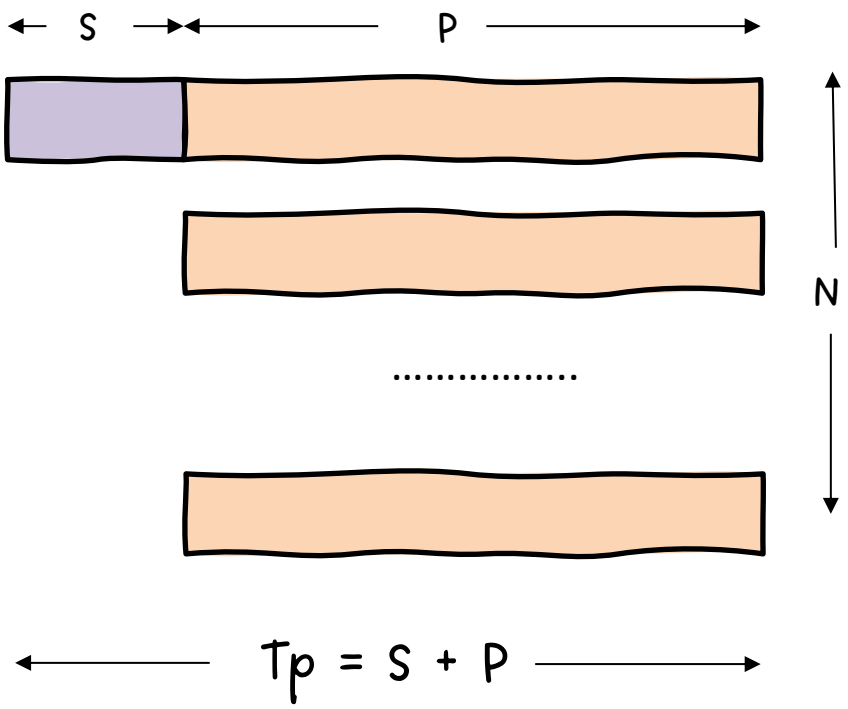




2 processing cores

Gustafson's Law





- Speedup S(N) เมื่อใช้ N processors แล้วเร็วขึ้นกี่เท่า
- S(N) = Ts / Tp

$$= \frac{S+NP}{S+P} = \frac{S+NP}{S+(1-S)}$$

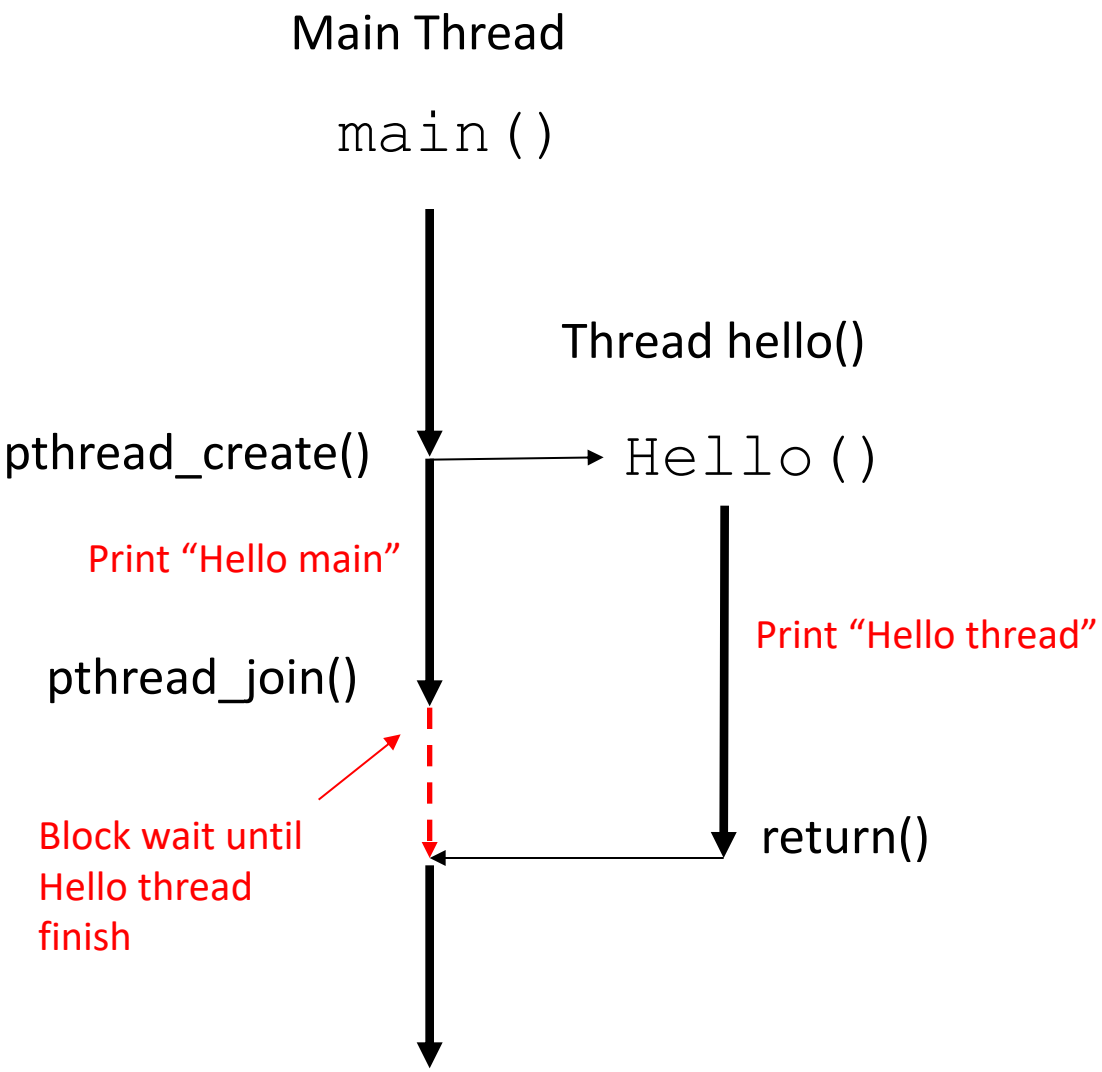
$$= S + NP$$

- เมื่อ N สูง S(N) สูงตามแบบ linear
  - (P เป็น slope)

Phread programming

- ภาษา C มี Library สำหรับสร้าง Thread เรียกว่า pthread library
- เราใช้คำสั่ง pthread_create เพื่อสร้างเทรด
- เราใช้คำสั่ง pthread_join เพื่อรอให้เทรดใดเทรดหนึ่งในโปรแกรมจบการประมวลผล
- เราใช้คำสั่ง pthread_exit เพื่อจบการประมวลผลของเทรด
- เราใช้คำสั่ง pthread_detach เพื่อประกาศว่าเทรดใดเทรดหนึ่งเป็นอิสระจะจบไปเองและไม่ต้องใช้ pthread_join รอ

./pth_hello1



Thread Creation

- Create a thread

```
#include <pthread.h>

int pthread_create(pthread_t *restrict tidp,
                   const pthread_attr_t *restrict attr,
                   void *(*start_rtn)(void *), void *restrict arg);
```

Returns: 0 if OK, error number on failure

Thread Creation Parameters

- ***arg*** is a typeless pointer ***that can point to a structure if want to pass more than one args***
- The **pthread functions** usually return an error code when they fail

Thread Termination

A single thread can exit in three ways, thereby stopping its flow of control, without terminating the entire process.

1. The thread can simply return from the start routine. The return value is the thread's exit code.

2. The thread can be canceled by another thread in the same process.

3. The thread can call `pthread_exit`.

Thread Termination

```
#include <pthread.h>

void pthread_exit(void *rval_ptr);
```

- The ***rval_ptr pointer*** will be available to **other threads** by calling the **pthread_join()** function

pthread_join()

```
#include <pthread.h>

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

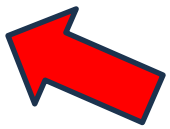
Returns: 0 if OK, error number on failure

- **The calling thread will block** until the specified thread
  - calls pthread_exit,
  - returns from its start routine, or
  - is canceled.
- If the thread simply returned from its start routine, **rval_ptr will contain the return code**.

- If the **thread was canceled**, the memory location specified by **rval_ptr is set to PTHREAD_CANCELED**.
- By calling pthread_join, we automatically place the thread with which we're joining in the **detached state** so that its resources can be recovered (เมื่อเทรดนั้นจบการประมวล).
- Return **If the thread was already in the detached state, pthread_join can fail**ing EINVAL
- If rval_ptr == NULL, **pthread_join** will wait but not return status info

example1.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4 #include <pthread.h>
 5
 6 void * foo(void * arg){
 7    printf("Hello World\n");
 8    return(NULL);
 9 }
10
11 int main(void){
12
13    pthread_t mtid, ntid;
14
15    mtid = pthread_self();
16    pthread_create(&ntid, NULL, foo, NULL);
17
18    printf("main thread id = %lu foo thread id = %lu\n",
19                                  (unsigned long) mtid,
20                                  (unsigned long) ntid);
21
22 }
```



ถ้า main จบก่อน เทรด foo ก็จะไม่เห็นผลลัพธ์ของ foo

example2.c

```
#include <pthread.h>

pthread_t ntid;

void
printids(const char *s)
{
    pid_t       pid;
    pthread_t   tid;

    pid = getpid();
    tid = pthread_self();
    printf("%s pid %lu tid %lu (0x%lx)\n", s, (unsigned long)pid,
      (unsigned long)tid, (unsigned long)tid);
}

void *
thr_fn(void *arg)
{
    printids("new thread: ");
    return((void *)0);
}
```

```
int
main(void)
{
    int     err;

    err = pthread_create(&ntid, NULL, thr_fn, NULL);
    if (err != 0)
        err_exit(err, "can't create thread");
    printids("main thread:");
    sleep(1);
    exit(0);
}
```

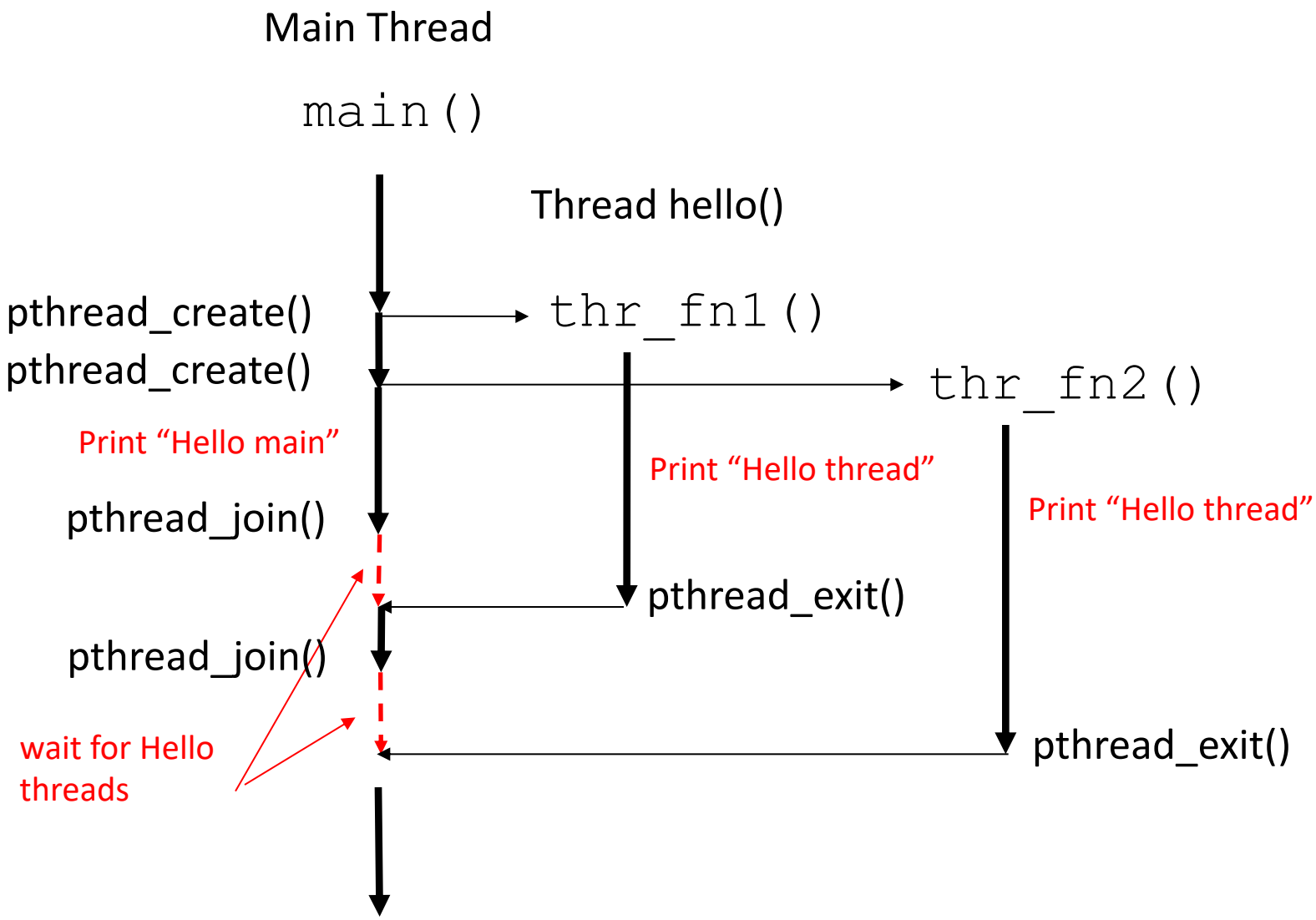
example3.c

```
#include <pthread.h>

void *
thr_fn1(void *arg)
{
    printf("thread 1 returning\n");
    return((void *)1);
}

void *
thr_fn2(void *arg)
{
    printf("thread 2 exiting\n");
    pthread_exit((void *)2);
}

int
main(void)
{
    int          err;
    pthread_t    tid1, tid2;
    void         *tret;

    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (err != 0)
        err_exit(err, "can't create thread 1");
    err = pthread_create(&tid2, NULL, thr_fn2, NULL);
```

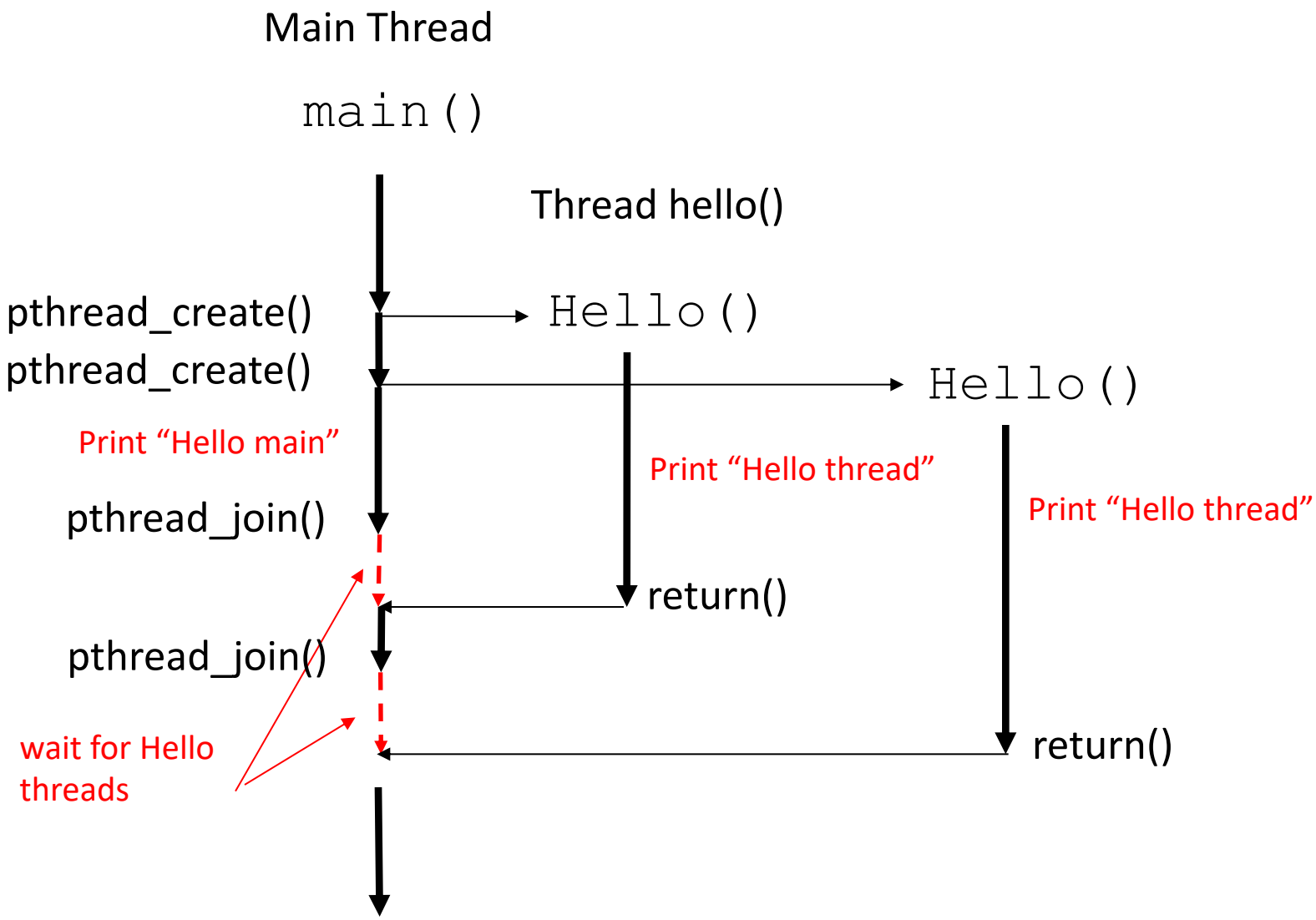
```
    if (err != 0)
        err_exit(err, "can't create thread 2");
    err = pthread_join(tid1, &tret);
    if (err != 0)
        err_exit(err, "can't join with thread 1");
    printf("thread 1 exit code %ld\n", (long)tret);
    err = pthread_join(tid2, &tret);
    if (err != 0)
        err_exit(err, "can't join with thread 2");
    printf("thread 2 exit code %ld\n", (long)tret);
    exit(0);
}
```

```
$ ./a.out
thread 1 returning
thread 2 exiting
thread 1 exit code 1
thread 2 exit code 2
```

./a.out



ตัวอย่าง ./hello 2



Hello World! (1)

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

/* Global variable:  accessible to all threads */
int thread_count;

void *Hello(void* rank);  /* Thread function */

int main(int argc, char* argv[]) {
   long       thread;  /* Use long in case of a 64-bit system */
   pthread_t* thread_handles;

   /* Get number of threads from command line */
   thread_count = strtol(argv[1], NULL, 10);

   thread_handles = malloc (thread_count*sizeof(pthread_t));
```

declares the various Pthreads functions, constants, types, etc.

Allocate memory to store thread handle objects.

Hello World! (2)

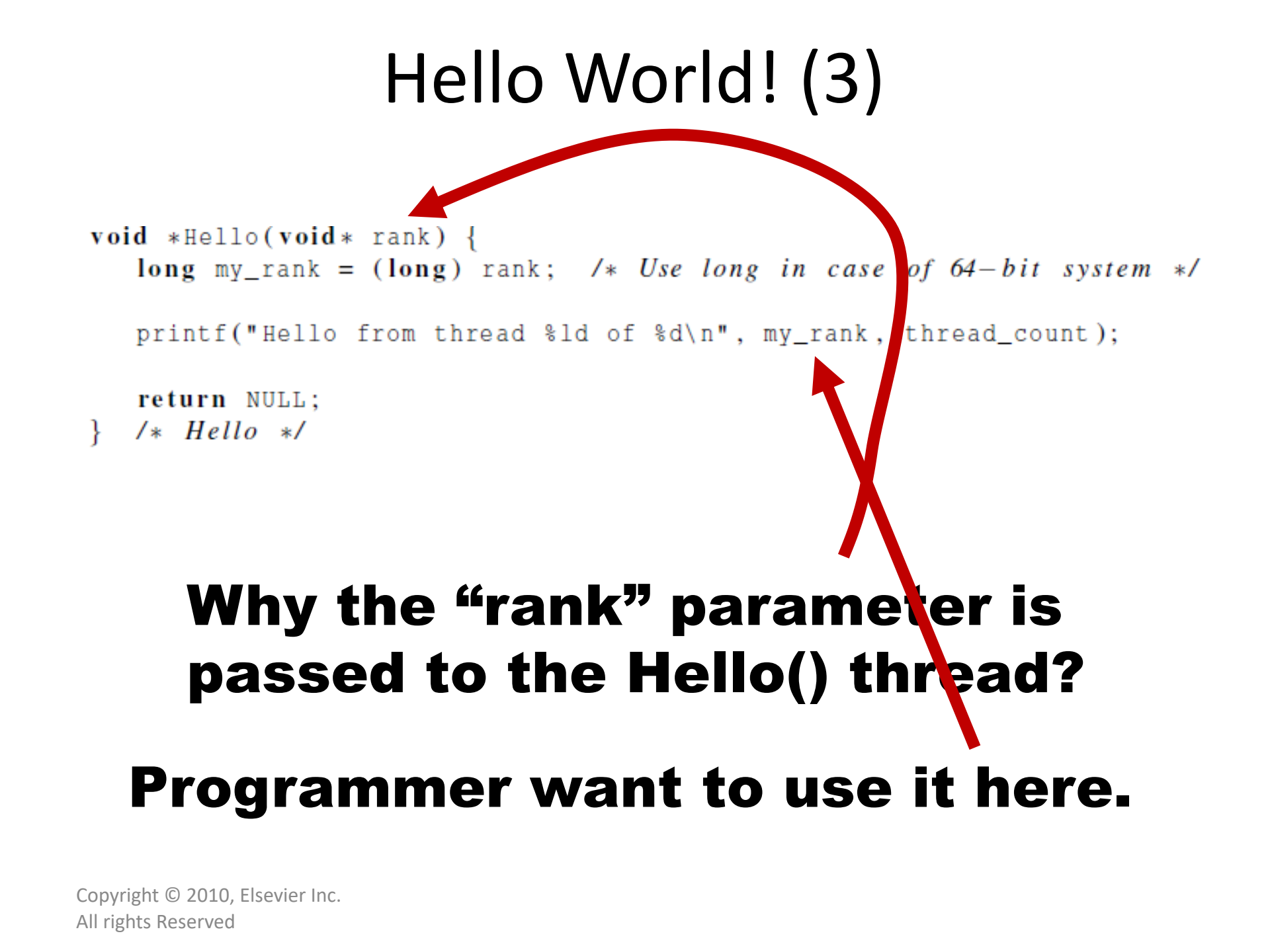
Repeatedly create a number of threads. Each running a “Hello()” function

```
for (thread = 0; thread < thread_count; thread++)
   pthread_create(&thread_handles[thread], NULL,
       Hello, (void*) thread);

printf("Hello from the main thread\n");

for (thread = 0; thread < thread_count; thread++)
   pthread_join(thread_handles[thread], NULL);

free(thread_handles);
return 0;
} /* main */
```

Main thread call “pthread_join” to wait until every thread finishes.

Hello World! (3)

```
void *Hello(void* rank) {
   long my_rank = (long) rank;  /* Use long in case of 64-bit system */

   printf("Hello from thread %ld of %d\n", my_rank, thread_count);

   return NULL;
}  /* Hello */
```



Why the "rank" parameter is passed to the Hello() thread?

Programmer want to use it here.

pthread_detach()

```
#include <pthread.h>

int pthread_detach(pthread_t tid);
```

Returns: 0 if OK, error number on failure

- **We can detach a thread by calling pthread_detach.**
- After a thread is detached, **we *cannot* use the pthread_join function to wait for its termination status**, because calling pthread_join for a detached thread results in undefined behavior.

process vs thread interfaces

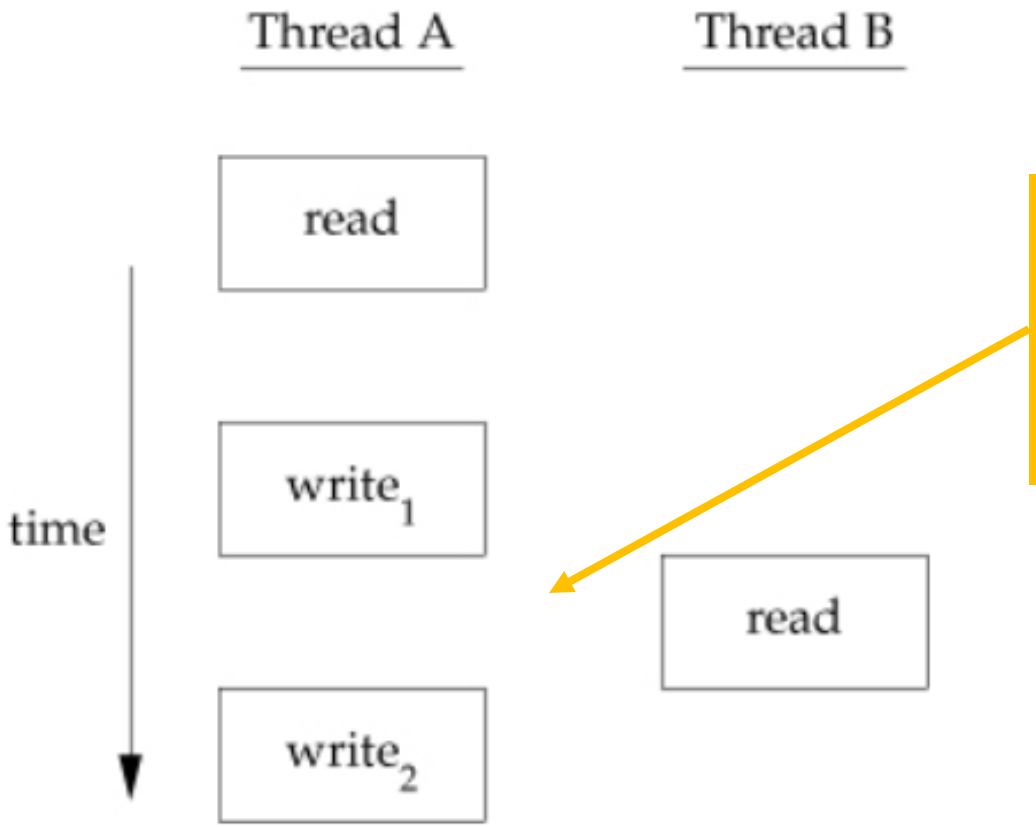
Process primitive	Thread primitive	Description
`fork`	`pthread_create`	create a new flow of control
`exit`	`pthread_exit`	exit from an existing flow of control
`waitpid`	`pthread_join`	get exit status from flow of control
`atexit`	`pthread_cleanup_push`	register function to be called at exit from flow of control
`getpid`	`pthread_self`	get ID for flow of control
`abort`	`pthread_cancel`	request abnormal termination of flow of control

Figure 11.6 Comparison of process and thread primitives

Thread Synchronization

- เทรดทุกเทรดใช้ memory ร่วมกัน
- ดังนั้นข้อมูลในตัวแปรต่างๆต้องได้รับการเข้าถึงอย่างถูกต้อง
- การ assign ค่าให้กับ variable ในภาษา C ประกอบไปด้วยคำสั่งภาษาเครื่องมากกว่า 1 คำสั่ง
- ในขณะที่เทรด 1 เทรดกำลัง assign ค่าให้กับ variable อาจมีเทรดอีกหนึ่งเทรดอ่านค่าจาก variable นั้นไปประมวลผล
  - เนื่องจากเทรดที่ 2 อาจอ่านค่าในขณะที่เทรดแรกยัง assign ไม่เสร็จก็จะทำให้เทรดที่ 2 ได้ค่าที่ผิดพลาดไปใช้งาน
- จำเป็นต้องมีการ Synchronization ระหว่างสองเทรด

ปัญหา เขียนและอ่านพร้อมกัน



เทรด A ยังเขียนข้อมูลสู่ Variable ใน memory ไม่เสร็จเลย แต่ เทรด B อ่านค่าที่อาจยังไม่สมบูรณ์ ของ variable นั้นไปใช้แล้ว

Figure 11.7 Interleaved memory cycles with two threads

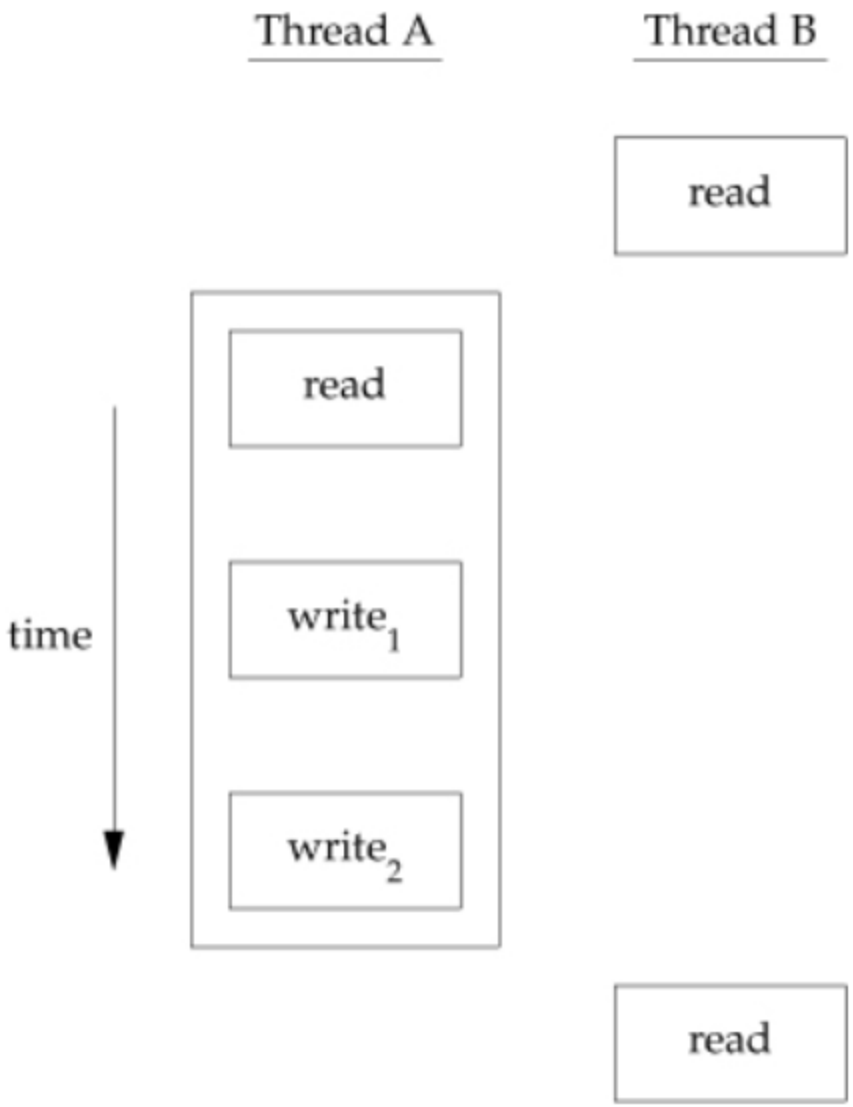
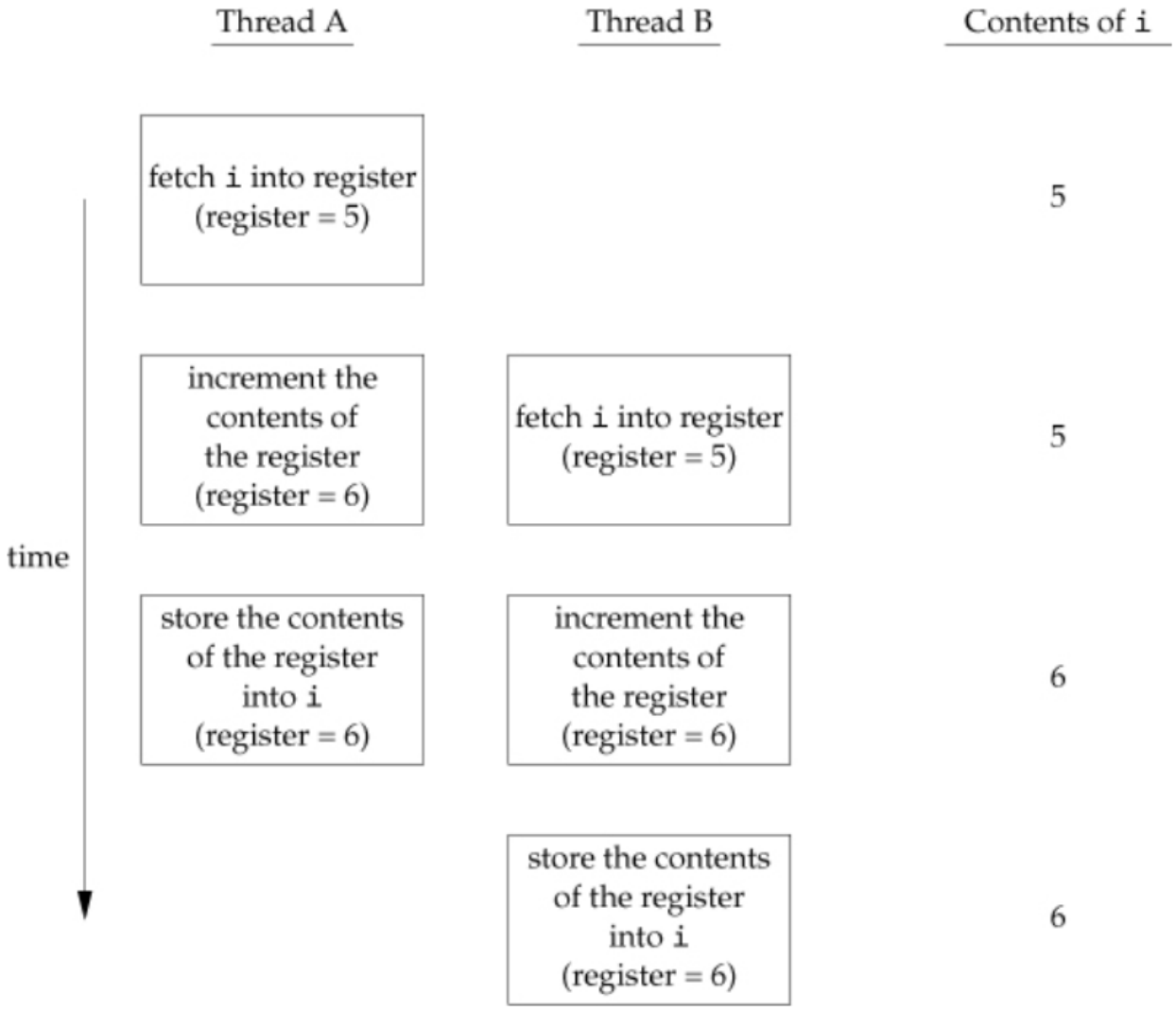


Figure 11.8 Two threads synchronizing memory access

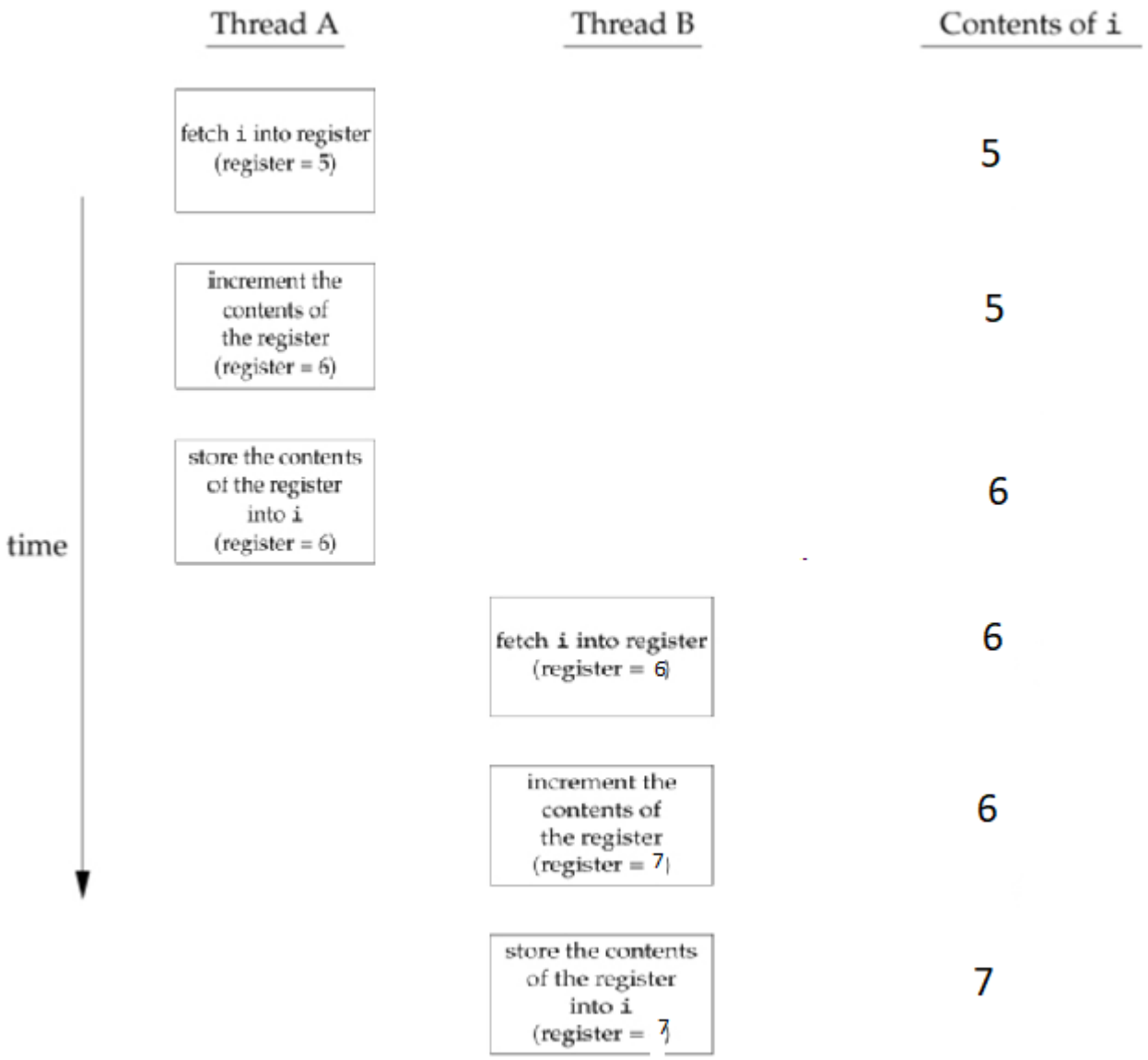
ปัญหาการ update ค่าพร้อมกัน

- อ่านค่าจากตัวแปร i มาที่ register
- เพิ่มค่าใน register + 1
- เขียนค่าจาก register สู่ตัวแปร I
- เกิดปัญหาเนื่องจากการ update ไม่เป็น atomic การ update เหลื่อมกัน



แก้ปัญหาการ update ค่าพร้อมกัน

- ต้อง synchronize ให้ เทรด A update ก่อนแล้วตามด้วย B หรือ เทรด B update ก่อนแล้วตามด้วยเทรด A จึงจะถูก



mutex

- mutex คือการล้อก (lock) การเข้าใช้งาน shared resource และปลดล้อก (unlock) เมื่อใช้งาน resource นั้นเสร็จแล้ว
- ถ้ามีเทรดหนึ่งได้ lock เทรดอื่นที่จะเข้าใช้ resource ต้องรอ (block) จนกว่าเทรดที่ได้ lock จะ unlock
- ในกรณีที่มีเทรด blocked มากกว่า 1 เทรด เมื่อมีการ unlock เทรดที่ถูก block อยู่ทุกเทรดจะเปลี่ยนสถานะเป็น runnable แต่จะมีเพียงเทรดเดียวเท่านั้นที่ได้ lock และเทรดที่เหลือต้องกลับไป block รอเหมือนเดิม

ตัวแปร Mutex variable

- เป็นตัวแทนของการทำ lock/unclock synchronization
- มีชนิดเป็น pthread_mutex_t data type.
- เราสามารถกำหนดค่าเริ่มต้นให้เป็น PTHREAD_MUTEX_INITIALIZER (สำหรับตัวแปรที่ประกาศในปรแกรม) หรือ
- กำหนดค่าเริ่มต้นด้วย pthread_mutex_init()
- เมื่อเลิกใช้งานให้เรียก pthread_mutex_destroy()

init and destroy

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Both return: 0 if OK, error number on failure

- Attr is set to NULL

Lock/Trylock/Unlock

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

All return: 0 if OK, error number on failure

mutexex1.c

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

pthread_mutex_t mutex_var;
int global_var = 100;
```

mutexex1.c

```
 8
 9 void * foo(void * arg){
10
11     pthread_mutex_lock(&mutex_var);
12     global_var += 25;
13     printf("foo() update global_var to %d\n", global_var);
14     pthread_mutex_unlock(&mutex_var);
15
16     pthread_exit((void *)NULL);
17 }
```

mutexex1.c

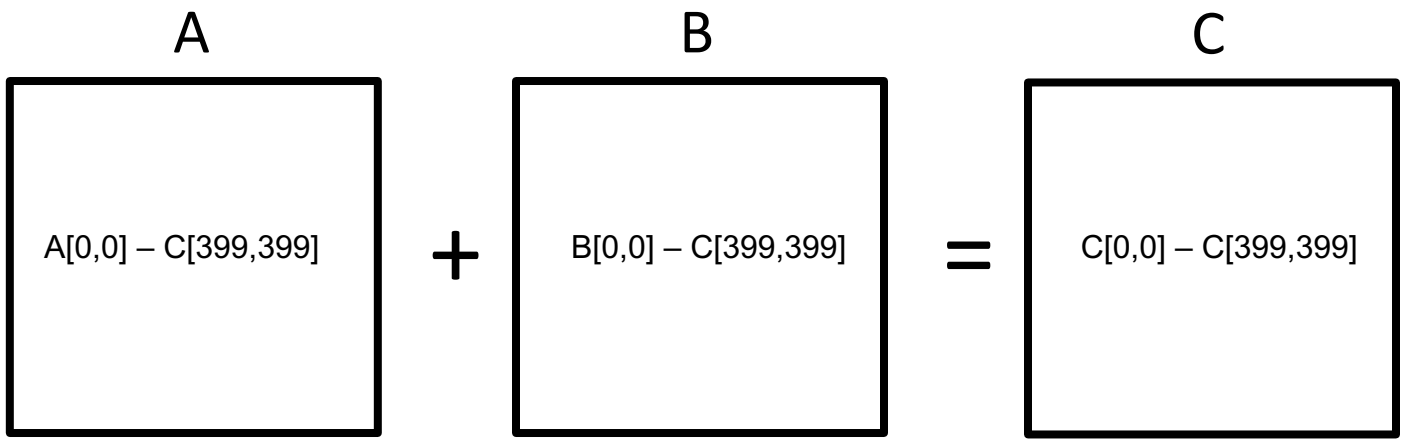
```
18
19 int main(void){
20
21     pthread_t mtid, ntid;
22     void *tret;
23
24     pthread_mutex_init(&mutex_var, NULL);
25
26     pthread_create(&ntid, NULL, foo, NULL);
27
28     pthread_mutex_lock(&mutex_var);
29     global_var += 75;
30     printf("main() update global_var to %d\n", global_var);
31     pthread_mutex_unlock(&mutex_var);
32
33     pthread_join(ntid,&tret);
34
35 }
```

mutex

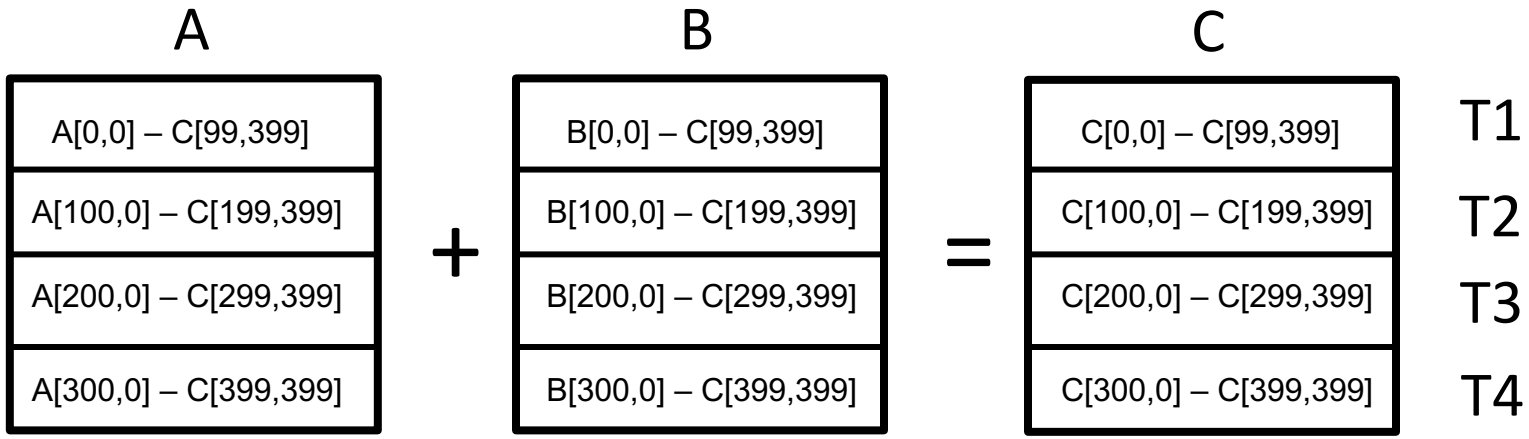
	Main thread	Foo thread
	Lock(&mutex_var)	Lock(&mutex_var)
	Global_var += 75;	
	Printf(“….”); // 175	
	Unlock(&mutex_var)	
		Global_var += 25;
		Printf(“….”); // 200
		Unlock(&mutex_var)

สรุป

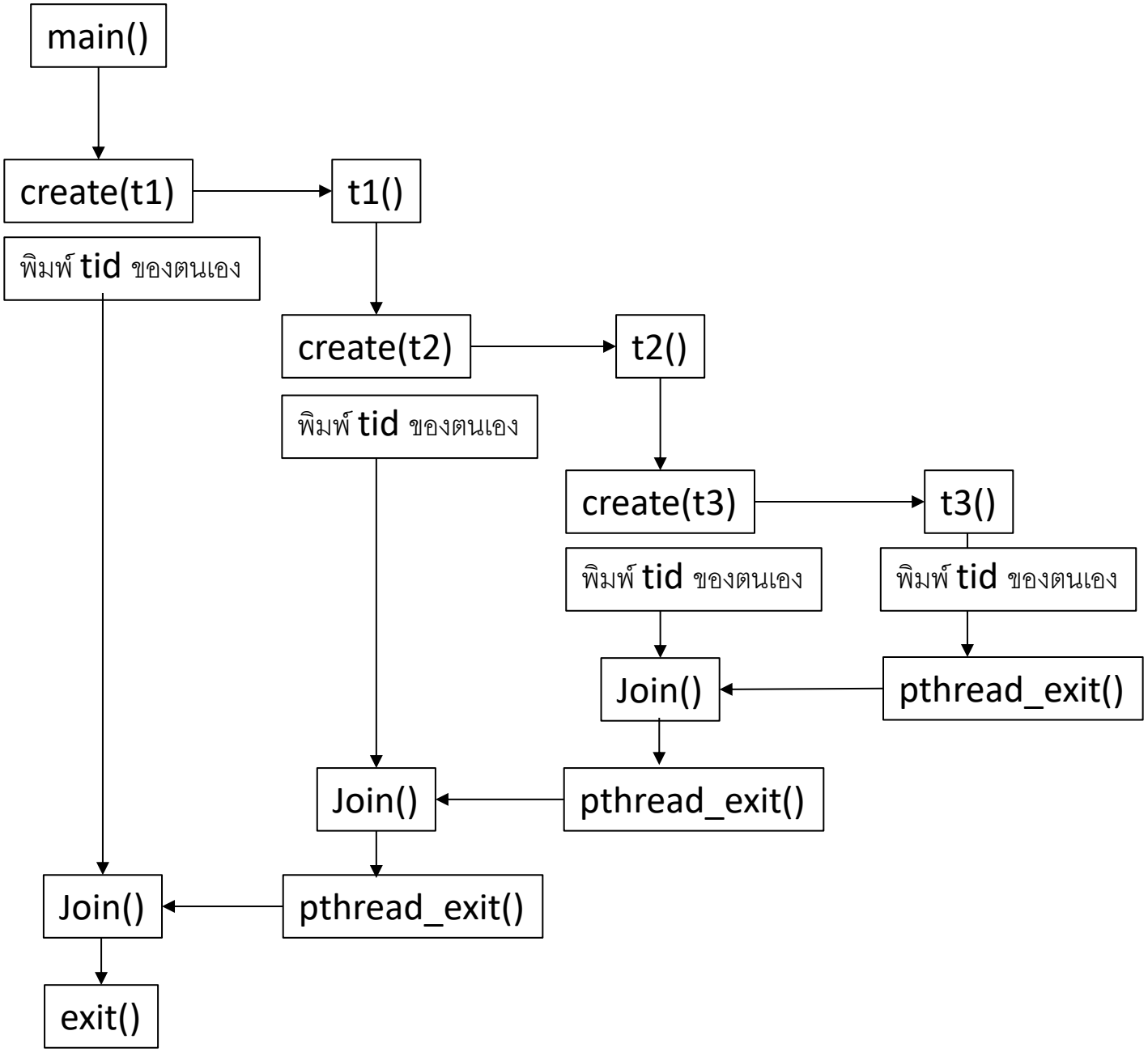
- แนะนำ Thread
- ชนิดของ Thread
- ทฤษฎีเกี่ยวกับ parallel processing: Amdahl’s Law
- การสร้าง Thread และจบการประมวลผลของ Thread
- แนะนำ Thread Synchronization ด้วยตัวแปร Mutual Exclusion


A
B
C
A[0,0] – C[399,399]
+
B[0,0] – C[399,399]
=
C[0,0] – C[399,399]

(ก)


A
B
C
A[0,0] – C[99,399]
A[100,0] – C[199,399]
A[200,0] – C[299,399]
A[300,0] – C[399,399]
+
B[0,0] – C[99,399]
B[100,0] – C[199,399]
B[200,0] – C[299,399]
B[300,0] – C[399,399]
=
C[0,0] – C[99,399]
C[100,0] – C[199,399]
C[200,0] – C[299,399]
C[300,0] – C[399,399]
T1
T2
T3
T4

(ข)


main()
create(t1)
t1()
พิมพ์ tid ของตนเอง
create(t2)
t2()
พิมพ์ tid ของตนเอง
create(t3)
t3()
พิมพ์ tid ของตนเอง
พิมพ์ tid ของตนเอง
Join()
pthread_exit()
Join()
pthread_exit()
Join()
pthread_exit()
exit()