Shared-Memory Programming (w/ Pthreads) Part 1

Prof. Hung-Pin (Charles) Wen
Dept. Electrical & Computer Engr, Nat'l Chiao Tung University

Courtesy of

Prof. C.T. Yang, Tonghai University &

Prof. P. Pacheco, Introduction to Parallel Programming

Outline

- Shared-Memory Programming
  - Process vs. Threads

- POSIX Threads
  - Basics
  - Critical Sections

History

- In the past (>10 years w/o multithreading support from O.S., how a server (such as telnet, BBS server) is implemented?

  ⇒ multi-process programming

- Shared-memory multiprocessor system
  - any memory location can be accessible by any processor
  - exist before multi-core systems
- Shared-memory programming
  - store data in the shared memory
  - not use message passing
  - more convenient

Approaches to Program Shared-Memory Multiprocessors

- Using heavyweight processes
- Using threads, ex: Pthreads, Java threads
- Using a completely new programming language for parallel programming, ex: Ada $\Rightarrow$ not popular
- Modifying existing sequential programming languages to create a parallel programming language, ex: UPC
- Supplemented with compiler directives and libraries for specifying parallelism, ex: OpenMP

Using Heavyweight Processes

- Operating systems often based upon notion of a process.
- Processor time shares between processes
  - switches from one process to another
  - may occur at regular intervals or when an active process becomes delayed
  - de-schedule processes blocked from proceeding, ex: waiting for an I/O operation
- Concept can be used for parallel programming
  - not much used because of overhead
  - but fork/join concepts used elsewhere

https://computing.llnl.gov/tutorials/pthreads/#WhyPthreads - Windows Internet Explorer

Why Pthreads?

- The primary motivation for using Pthreads is to realize potential program performance gains.
- When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.

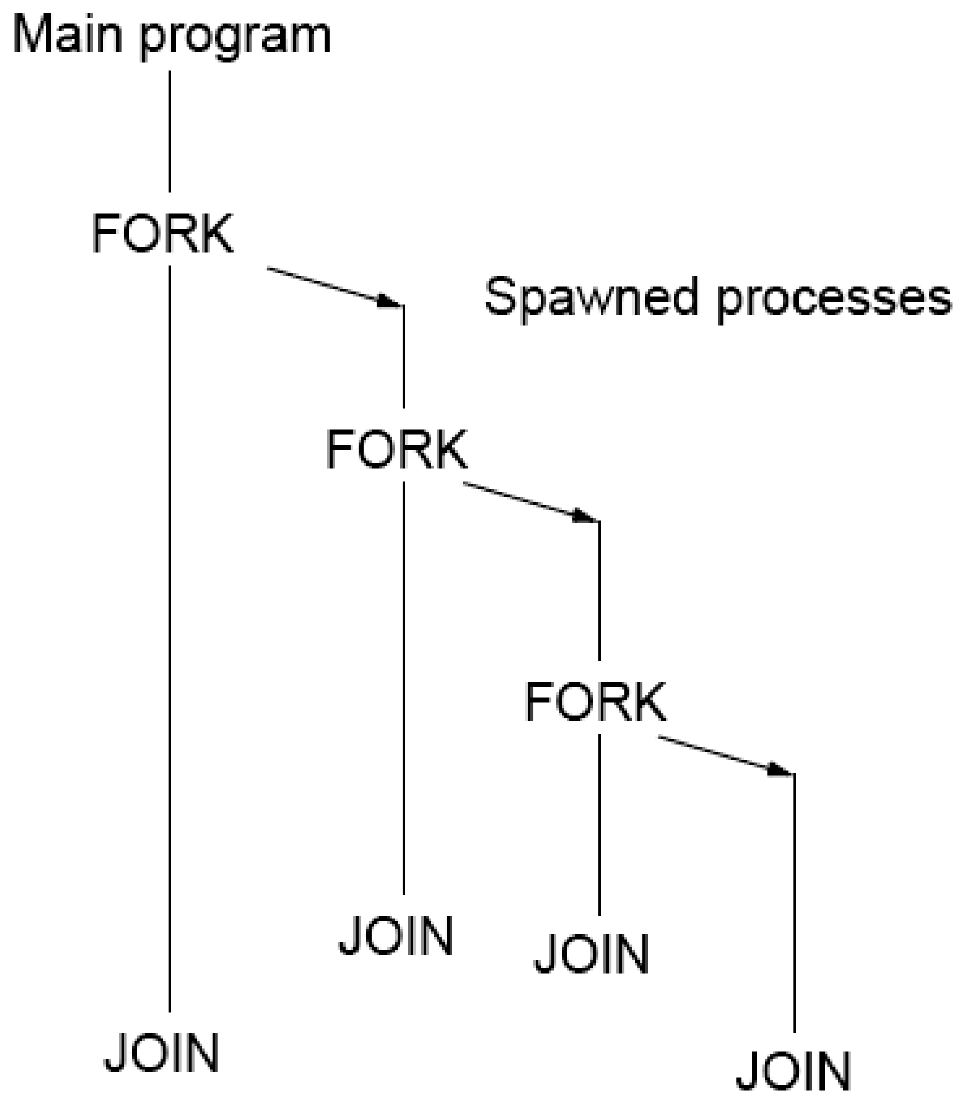
  For example, the following table compares timing results for the `fork()` subroutine and the `pthreads_create()` subroutine. Timings reflect 50,000 process/thread creations, were performed with the `time` utility, and units are in seconds, no optimization flags.

  Note: don't expect the sytem and user times to add up to real time, because these are SMP systems with multiple CPUs working on the problem at the same time. At best, these are approximations run on local machines, past and present.

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
AMD 2.3 GHz Opteron (16cpus/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8cpus/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

Source fork_vs_thread.txt

Multi-Process Programming: **FORK/JOIN**



UNIX System Calls

No join routine - use exit() and wait()

SPMD model

```
    .
    .
pid = fork();                      /* fork */
  Code to be executed by both child and parent
if (pid == 0) exit(0); else wait(0);/* join */
    .
    .
    .
```

Example (1) - Unix fork()

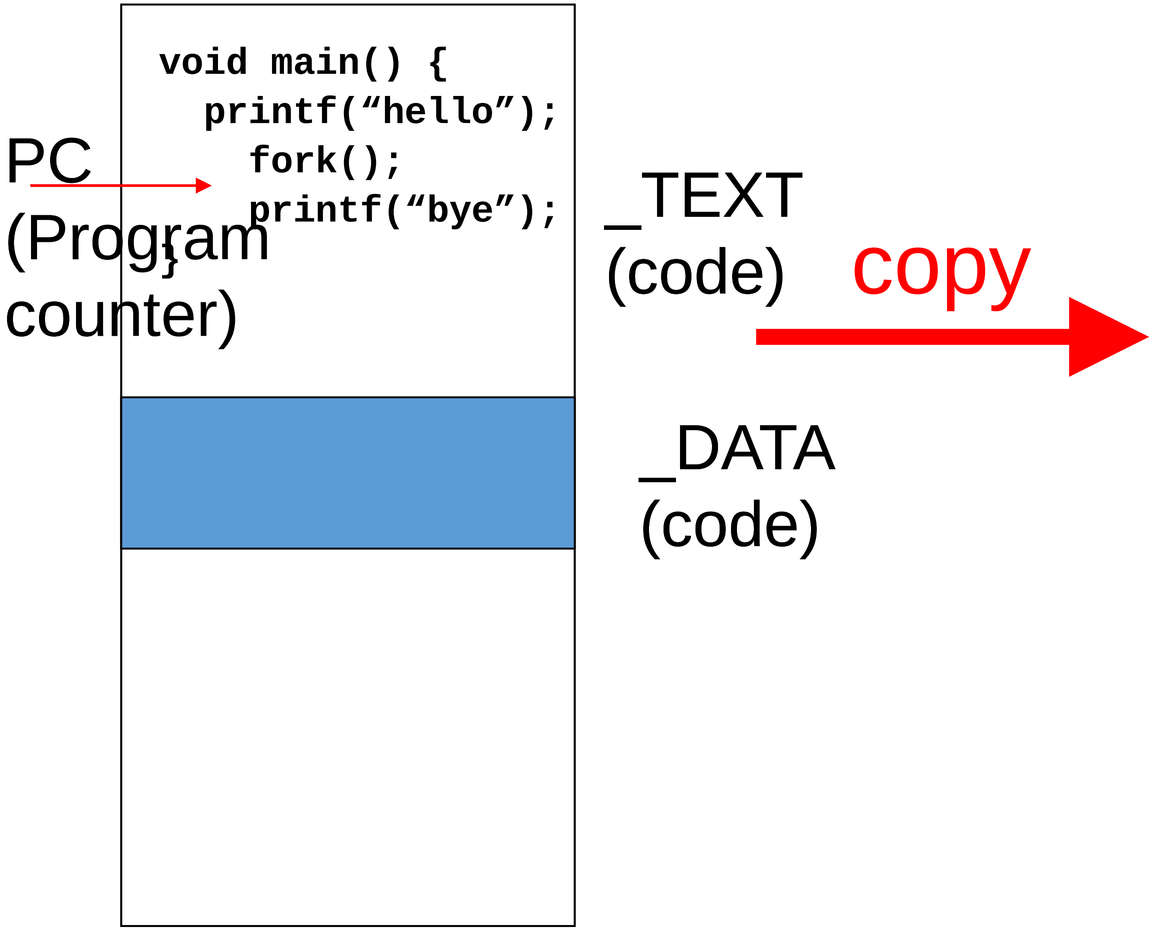
fork1.c

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

Output

```
hello
bye
bye
```

How fork() is implemented in Unix?



Process’s Image in Memory

Example (2) - fork() (1/2)

fork2.c

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

Try fork3.c

```
> ps -el
```

- PID: unique process id
- PPID: process id of parent
- UID: user id of process owner
- Priority: execution priority
- State: state of the process, e.g. running, sleep, zombie, …
- …

Example (2) - fork() (2/2)

See fork3.c

- Creates a child process that is identical to its parent process
- Child has its own PID
- Sets PPID of child to PID of parent
- Fork returns different return values to parent and child
  - Returns PID of child to the parent
  - Returns 0 to child
  - In this way we can determine if we are the parent or the child
- Forked child typically calls exec
- Parent waits for child to complete
  - otherwise child is zombie:
    - completed execution but has not delivered status to parent
  - wait return status encoded as integer
    - See macros in man 2 wait, e.g. WEXITSTATUS

Old-Fashioned Concurrent Server

```
void 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);
 } // end of for
 }
```

Problems w/ Multi-Process Programming

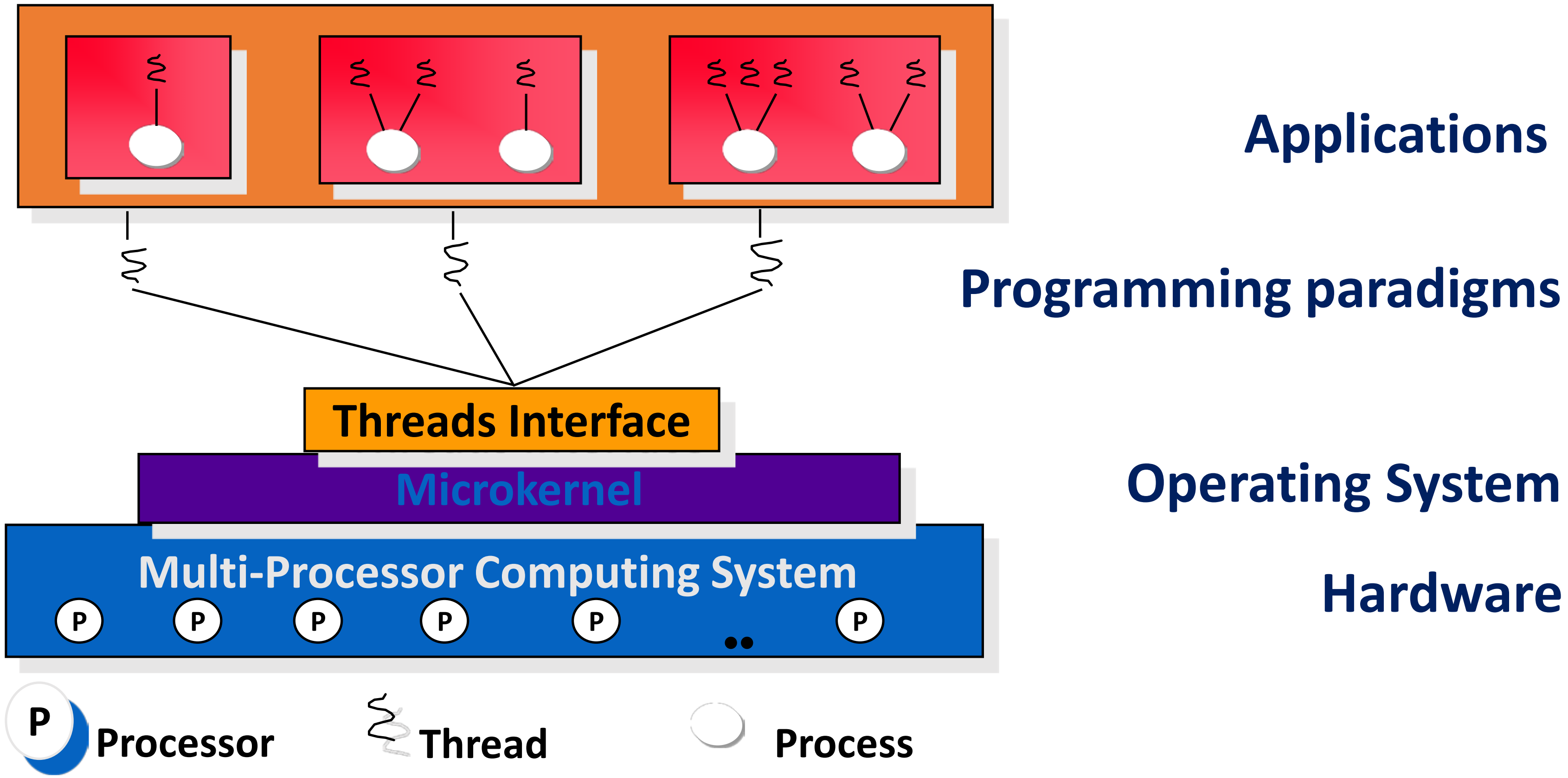
- Context switching overhead is high
- Communication between forked processes also has high cost
  - typical via IPC (interprocess communication)
  - ex: shared memory, semaphore, message queue
  - cross-address space communication (due to different processes)
  - IPC cause mode switch $\Rightarrow$ may cause a process to block

Concept of Process

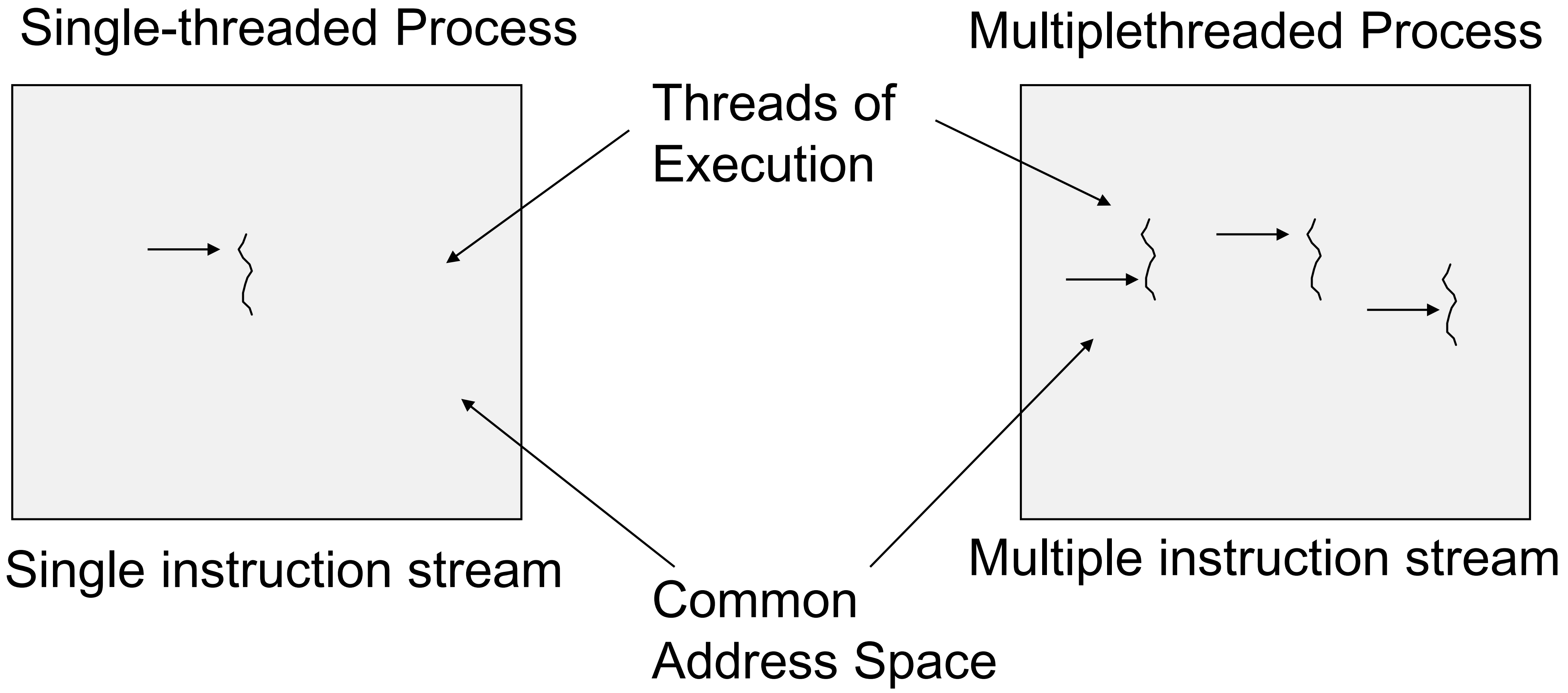
- Process: basic element in O.S, running program
  - UNIT of resources ownership
    - ✓ allocated with virtual address space
    - ✓ control of other resources
    - ✓ ex: I/O, files and etc
  - Unit of dispatching
    - ✓ execution paths (may interleaved with other processes)
    - ✓ execution state + dispatching priority
    - ✓ controlled by OS

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

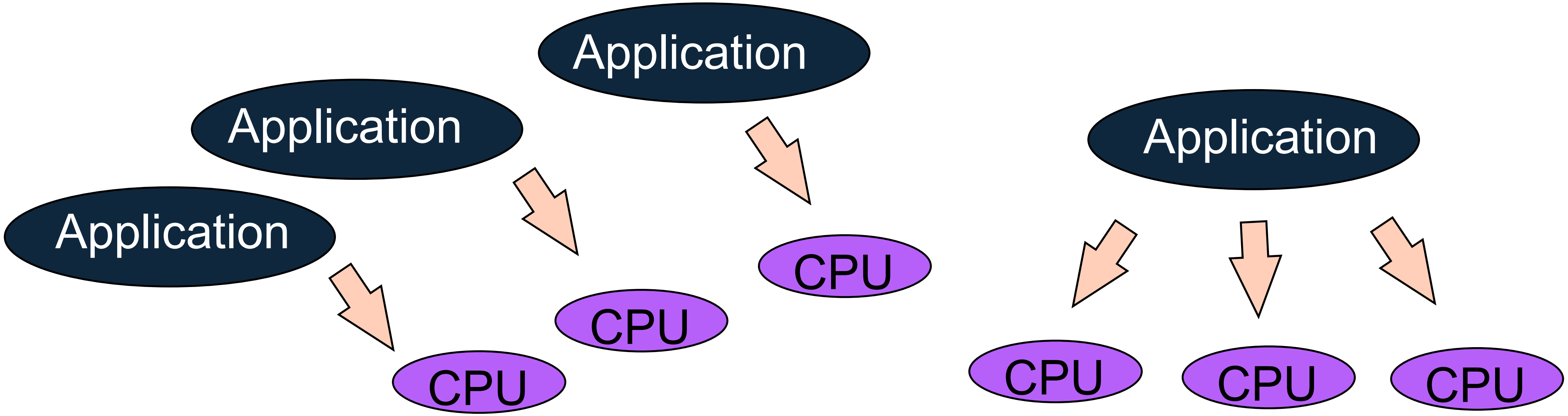


Single vs. Multithreaded Processes



Multi-Processing & Multi-Threaded OS

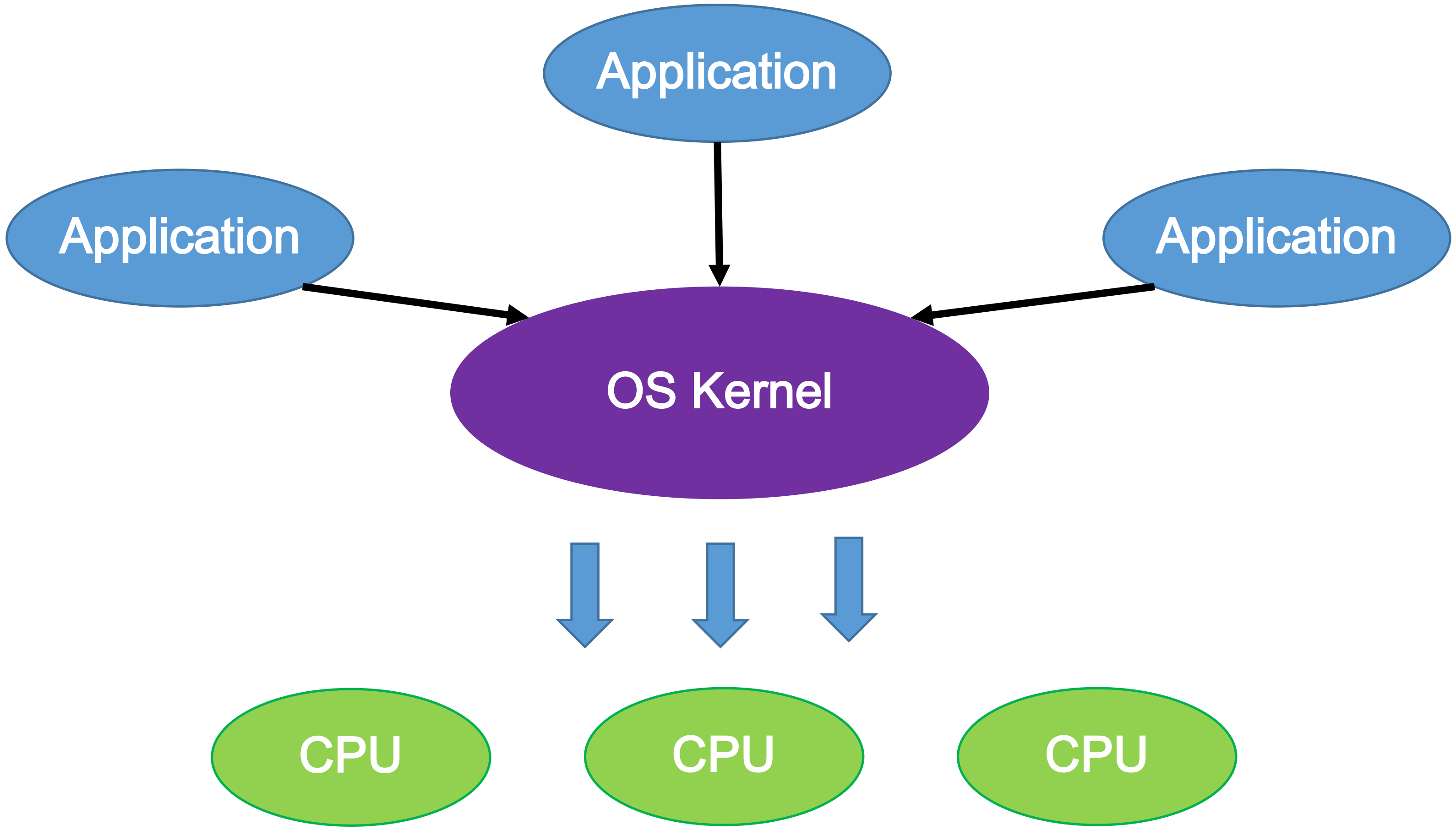
- Threaded Libraries, Multi-threaded I/O



Better Response Times in Multiple Application Environments

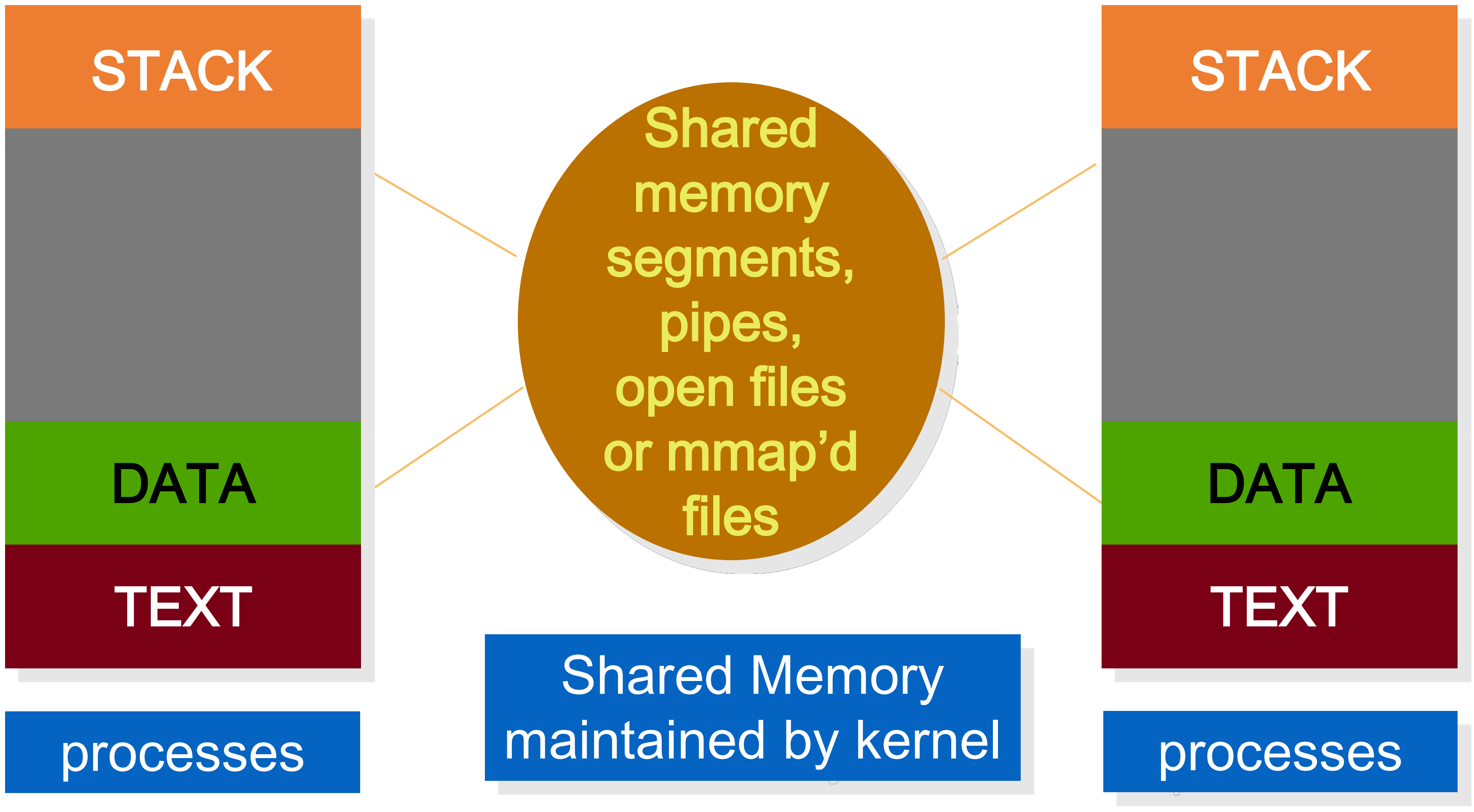
Higher Throughput for Parallelizeable Applications

Multi-threaded OS enables parallel, scalable I/O



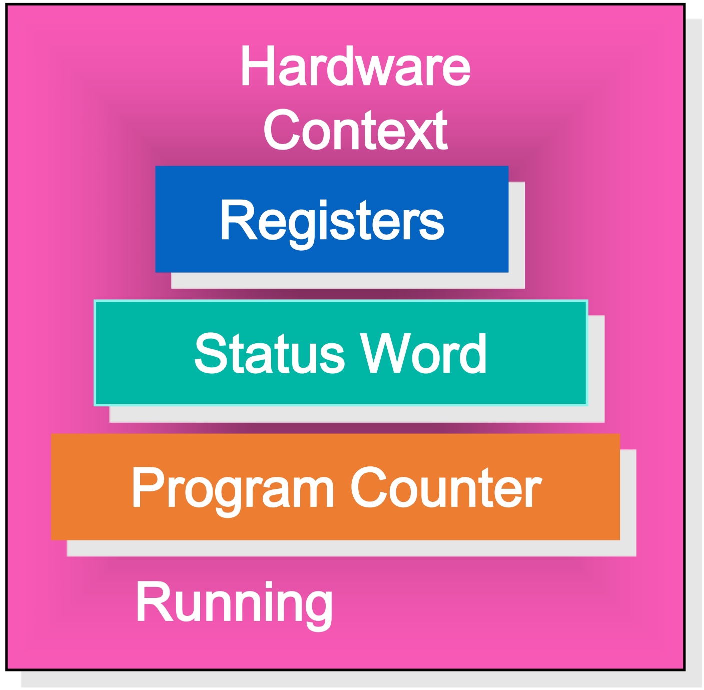
Multiple, independent I/O requests can be satisfied simultaneously because all the major disk, tape, and network drivers have been multi-threaded, allowing any given driver to run on multiple CPUs simultaneously.

Basic Process Model



What Are Threads?

- thread: a piece of code can execute in concurrence with other threads
  - a schedule entity on a processor



- Local state
- Global/ shared state
- PC
- Hard/Software Context

Thread Object

Sample Run (1)

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
  - main thread starts with main()
  - only one thread (or one program counter (PC)) 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 (PCs) in one code segment
- Windows support multithreading earlier (mid 1990)
- SunOS, Linux came late

Sample Run (1)

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);
        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

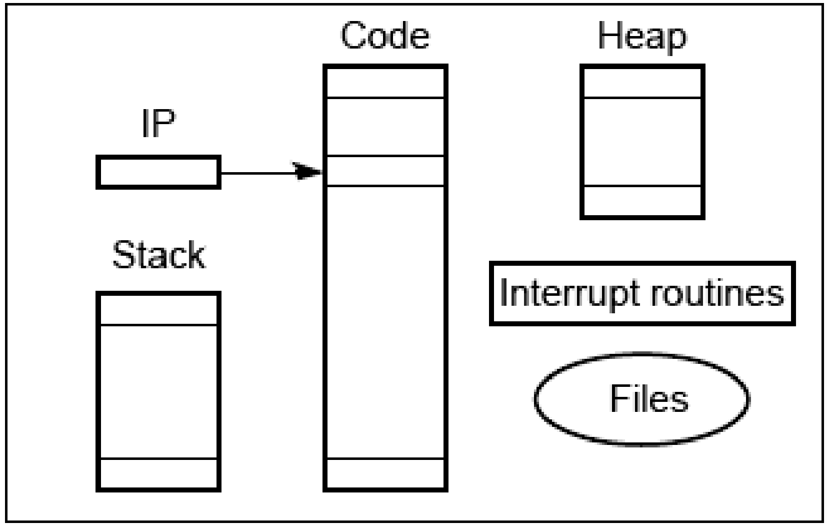
What’s New For Threads?

- 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

Differences: Process vs. Threads

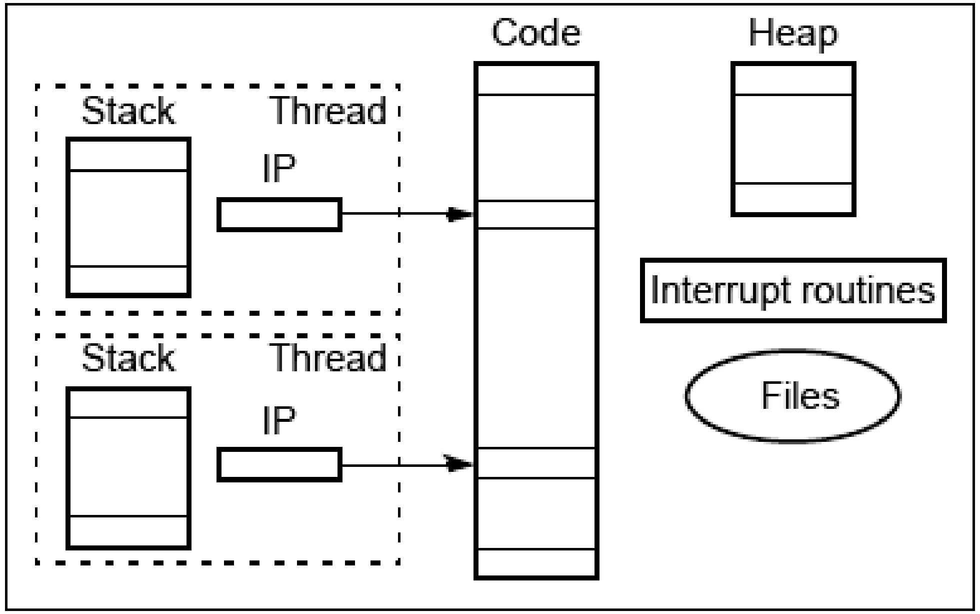
"heavyweight" process - completely separate program with its own variables, stack, and memory allocation.

(a) Process

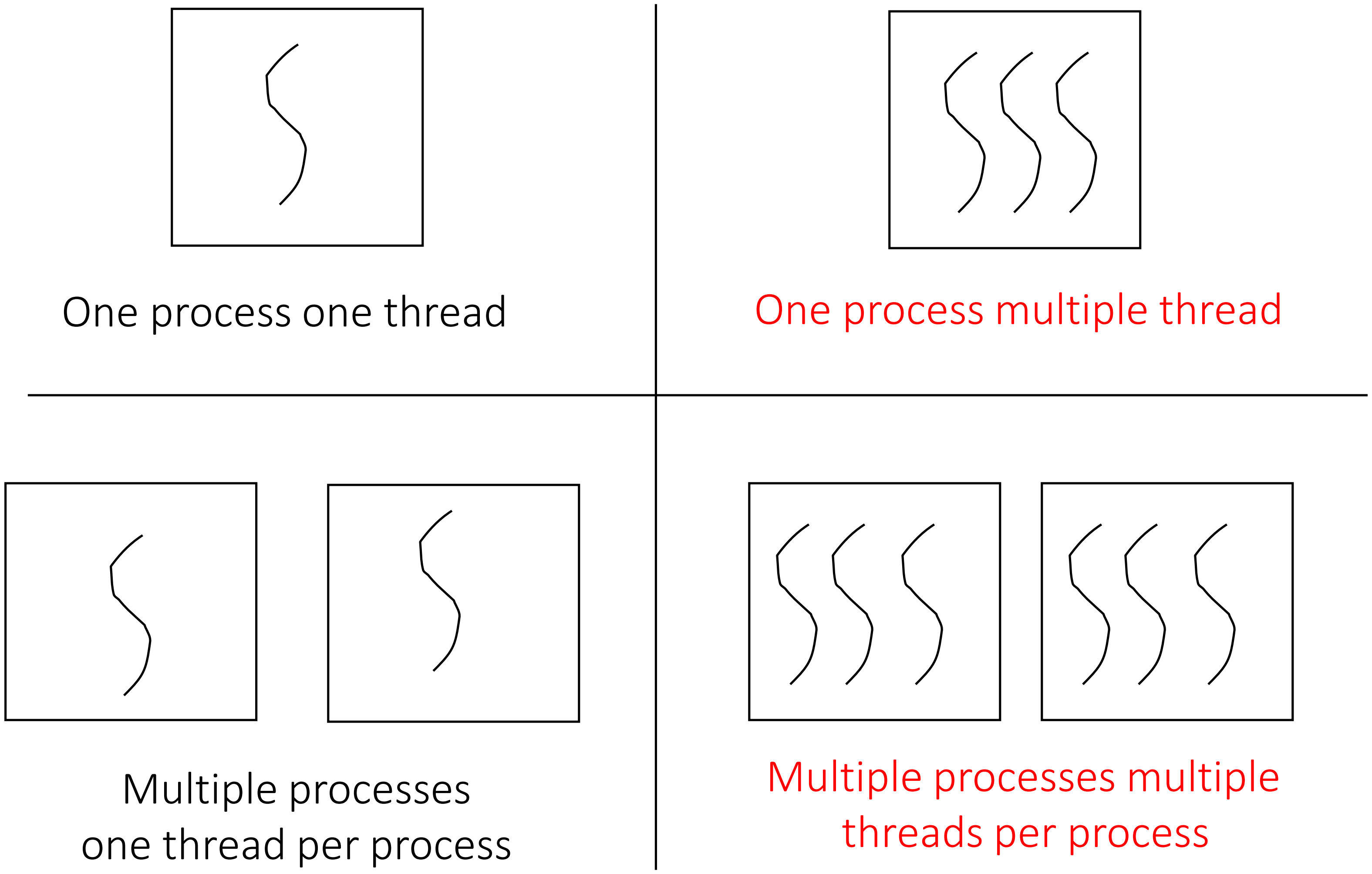


Threads - shares the same memory space and global variables between routines.

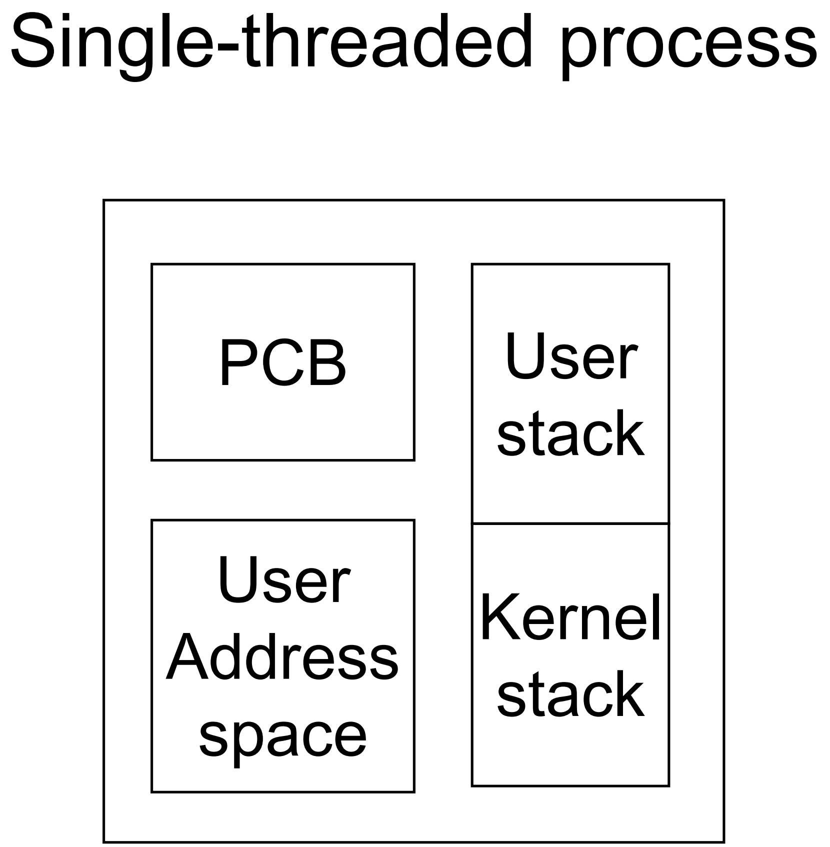
(b) Threads

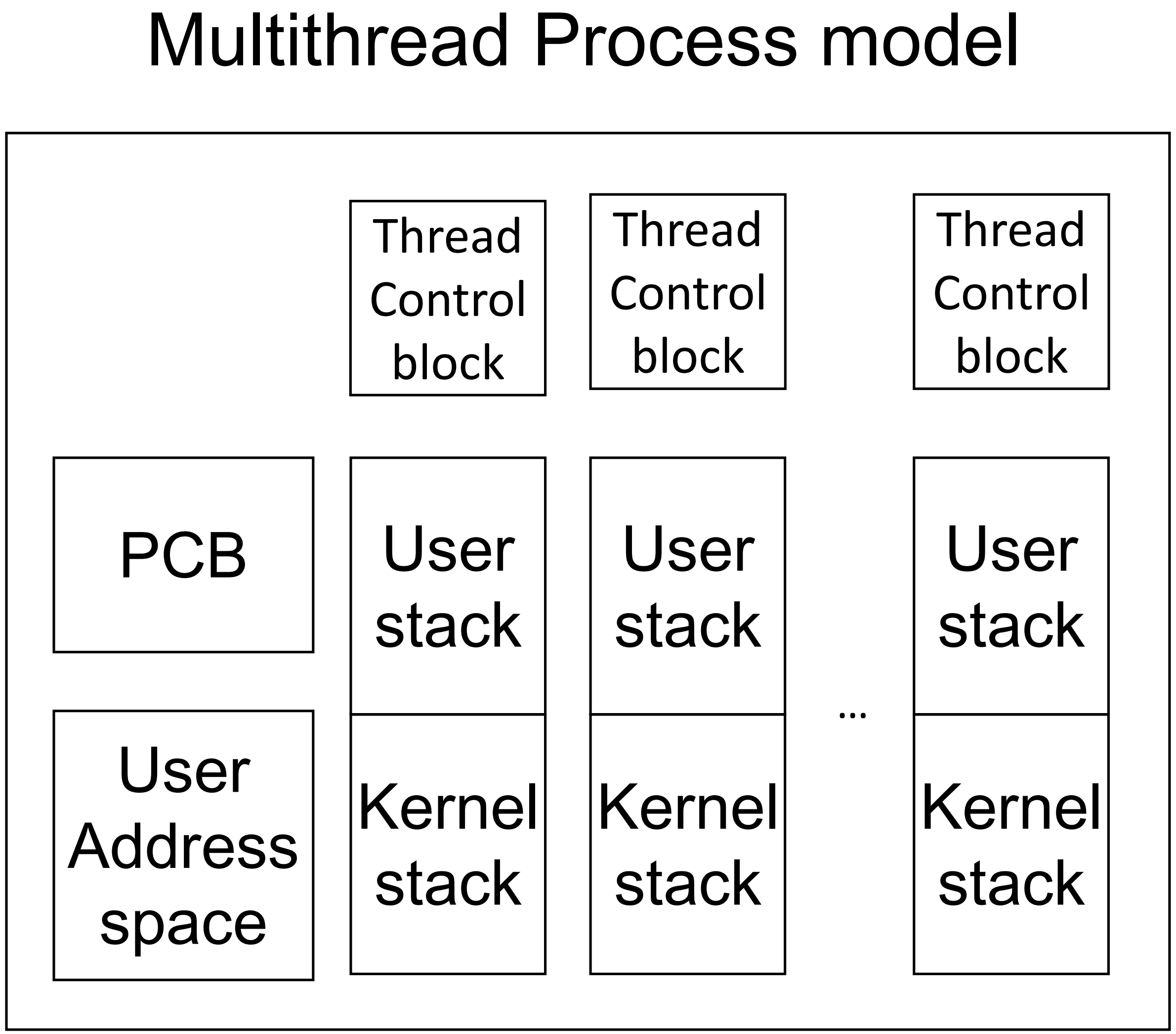


Possible Thread + Processes



Single Threaded & Multithreaded Model





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
- Ex: MACH (a OS kernel by CMU) shows a factor of 10

Outline

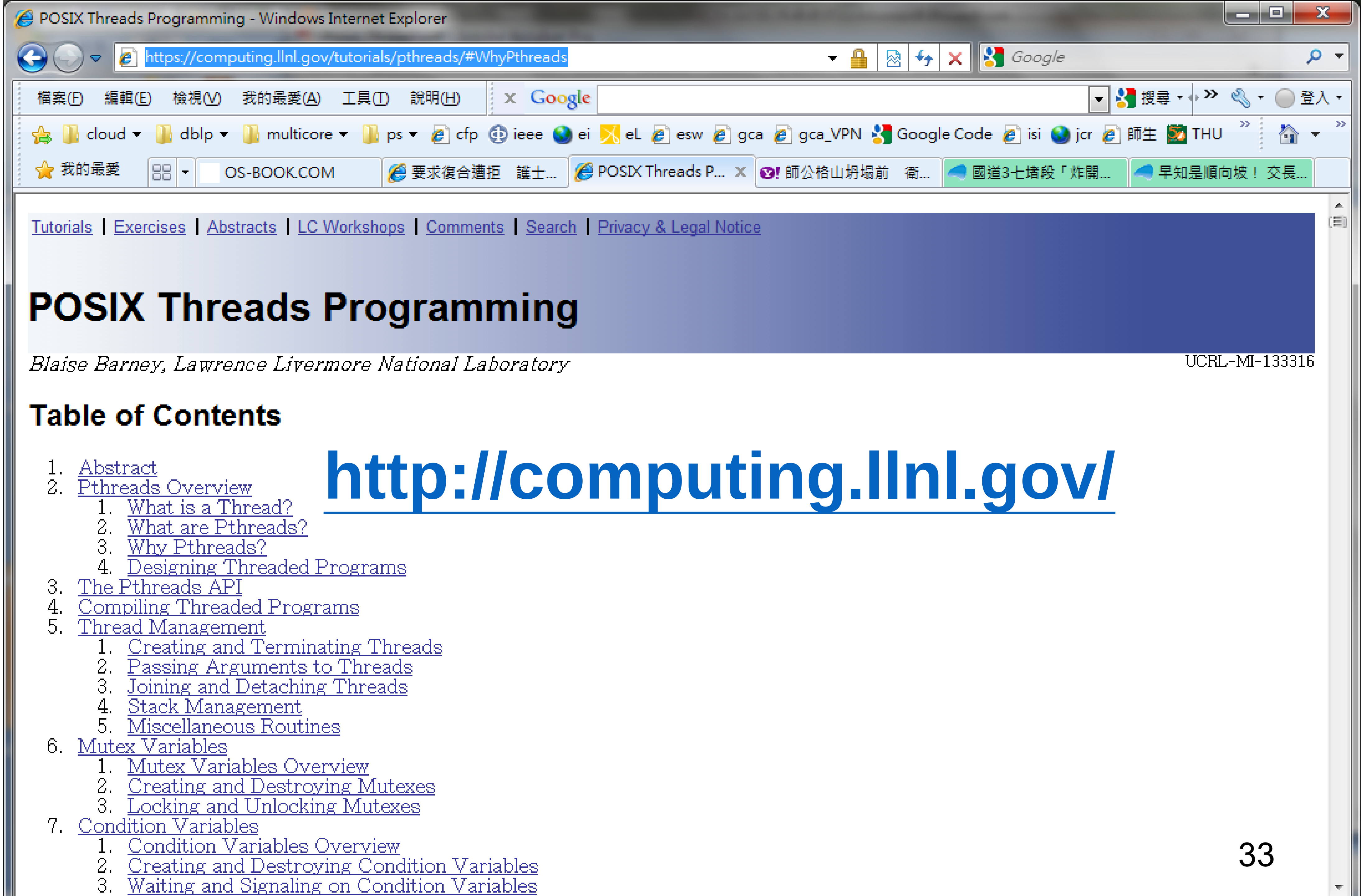
- Shared-Memory Programming
  - Process vs. Threads

- POSIX Threads
  - Basics
  - Critical Sections

POSIX® Threads Basics

Portable Operating System Interface (X??)

- IEEE standards, approved in 1995 (~20yrs)
- Also known as Pthreads
- A library that can be linked with C programs (header pthread.h)
- Specifies an application programming interface (API) for multi-threaded programming
- Only available on Unix-like POSIX® systems – Linux, MacOS X, …



POSIX Threads Programming - Windows Internet Explorer

https://computing.llnl.gov/tutorials/pthreads/#WhyPthreads

[Tutorials](#) | [Exercises](#) | [Abstracts](#) | [LC Workshops](#) | [Comments](#) | [Search](#) | [Privacy & Legal Notice](#)

POSIX Threads Programming

Blaise Barney, Lawrence Livermore National Laboratory

UCRL-MI-133316

Table of Contents

http://computing.llnl.gov/

Pthreads (1): Hello World! (1/3)

declares various Pthreads functions, constants, types, etc.

pth_hello.c

```
#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));
```

Pthreads (1): Hello World! (2/3)

```
   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 */
```

Pthreads (1): Hello World! (3/3)

```
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 */
```

Compile/Execte A Pthread program

- Compile

```
> gcc −g −Wall −o pth_hello pth_hello.c −lpthread
```

link Pthreads library

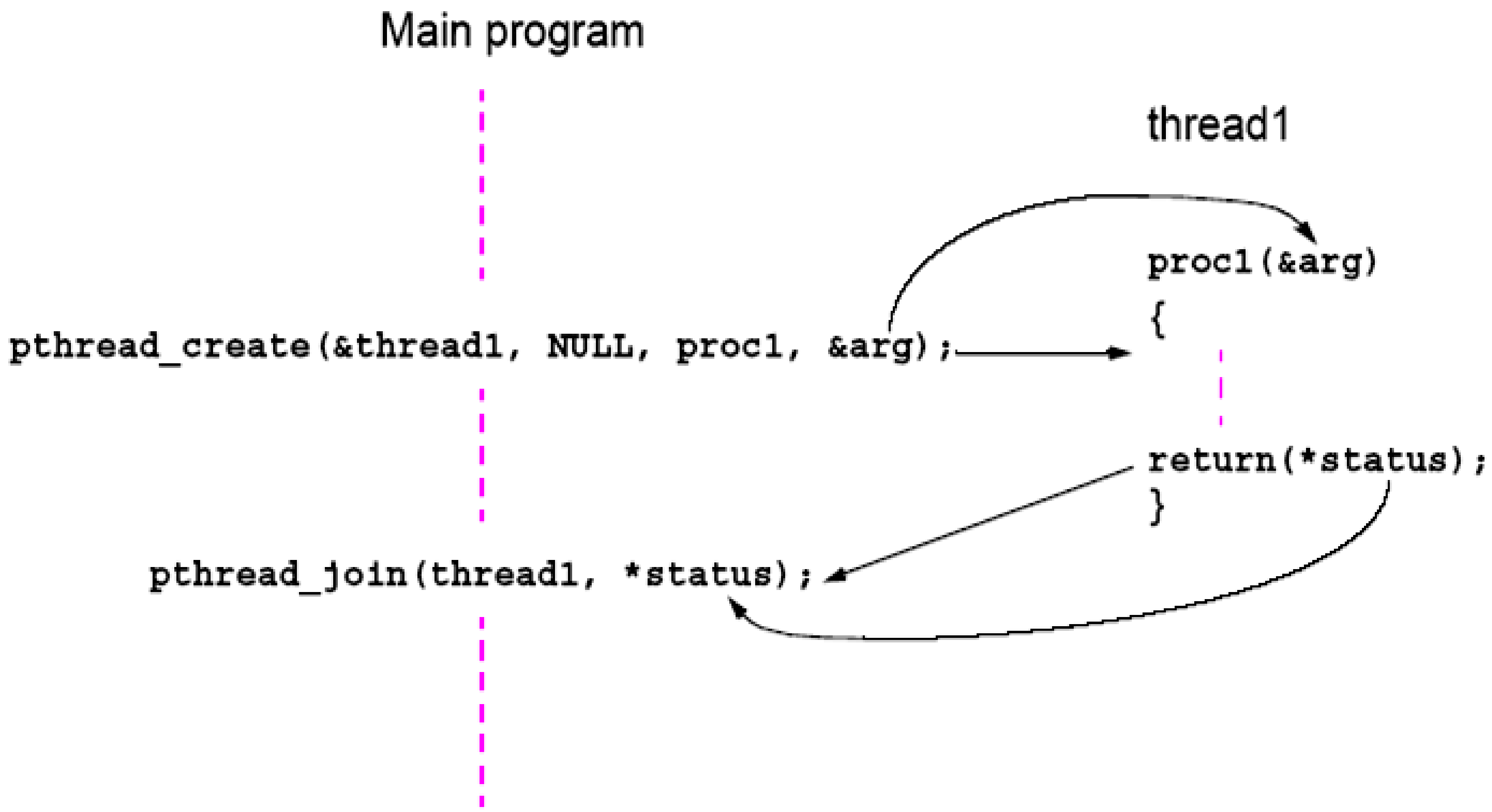
- Execute: > pth_hello <number of threads>

```
> pth_hello 1
Hello from the main thread
Hello from thread 0 of 1
>
```

```
> pth_hello 4
Hello from the main thread
Hello from thread 0 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
Hello from thread 1 of 4
>
```

Pthreads Flow

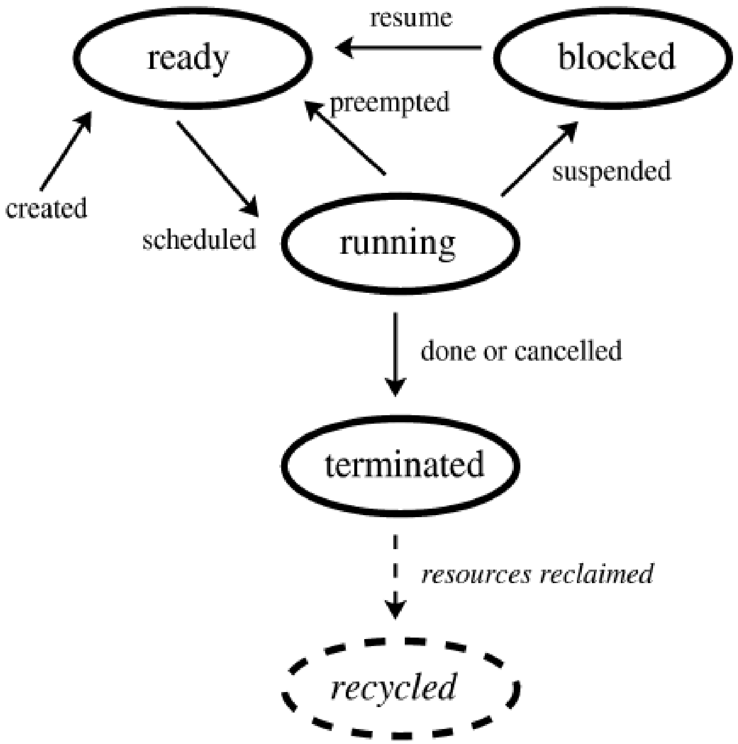
- IEEE Portable Operating System Interface, POSIX standard



*Processes in MPI are usually started by a script. But Pthreads are started by the program executable.

Pthreads Lifecycle: States

- Ready
  - able to run, waiting for processor
- Running
  - on multiprocessor possibly more than one at a time
- Blocked
  - thread is waiting for a shared resource
- Terminated
  - system resources partially released
  - but not yet fully cleaned up
    - thread's own memory is obsolete
    - can still return value
- (Recycled)
  - all system resources fully cleaned up
  - controlled by the operating system



Global Variables

- can introduce subtle and confusing bugs!
- limit use of global variables when really needed
  - shared variables

Starting Threads

pthread.h

One object for each thread

pthread_t

```
int pthread_create (
        pthread_t*  thread_p /* out */ ,
        const pthread_attr_t*  attr_p /* in */ ,
        void*  (*start_routine ) ( void ) /* in */ ,
        void*  arg_p /* in */ ) ;
```

pthread_t objects

- Opaque
- actual data that they store is system-specific
- data members are NOT directly accessible to user code
- However, the Pthreads standard guarantees that a pthread_t object does store enough information to uniquely identify the thread with which it's associated

A Closer Look

```
int pthread_create (
        pthread_t*  thread_p /* out */ ,
        const pthread_attr_t*  attr_p /* in */ ,
        void*  (*start_routine ) ( void ) /* in */ ,
        void*  arg_p /* in */ ) ;
```

allocate before calling.

not use, just pass NULL

function pointer that the thread is to run

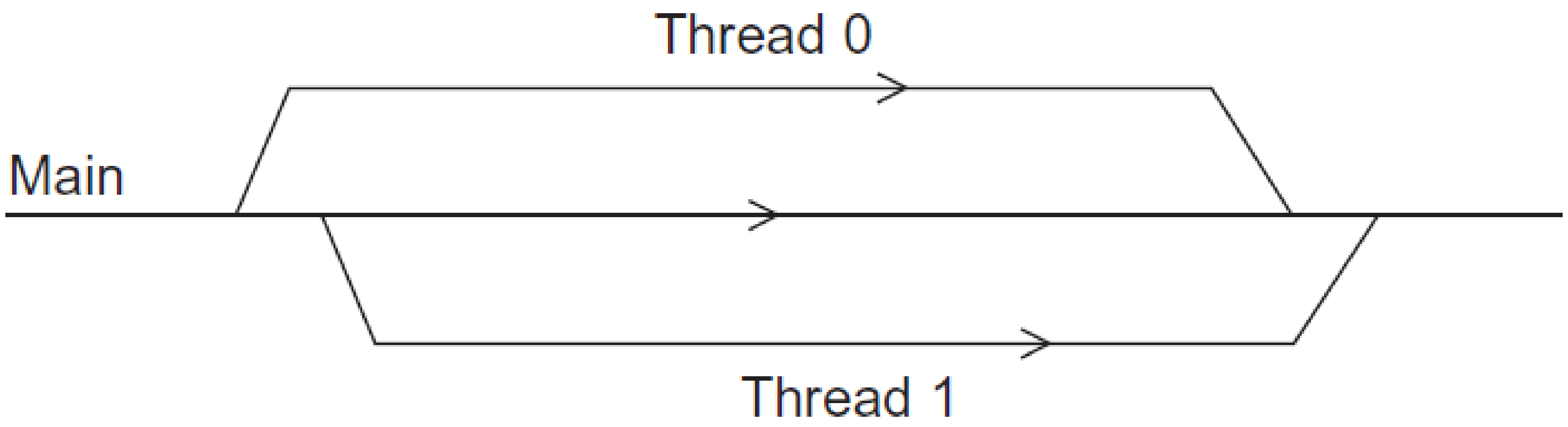
pointer to the argument that should be passed to the function start_routine

Function Started by pthread_create

- Prototype: void* thread_function (void* args_p) ;
  - void* can be cast to any pointer type in C (at two places)
  - args_p can point to a list containing one or more values needed by thread_function.
  - similarly, the return value of thread_function can point to a list of one or more values.

Running/Stopping Threads

- Main thread first forks and then joins two threads



- Call function pthread_join once for each thread
  - wait for the thread associated with the pthread_t object to complete
  - what if no call pthread_join!? ⇨ try comment out pth_hello.c

Exercise: Matrix-Vector Multiplication

- matrix A (m×n) multiplies vector X (n×1) = vector Y (m×1)

$$
\begin{array}{|c|c|c|c|}
\hline
a_{00} & a_{01} & \cdots & a_{0,n-1} \\
\hline
a_{10} & a_{11} & \cdots & a_{1,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\
\hline
\end{array}
\quad
\begin{array}{|c|}
\hline
x_0 \\
x_1 \\
\vdots \\
x_{n-1} \\
\hline
\end{array}
=
\begin{array}{|c|}
\hline
y_0 \\
\hline
y_1 \\
\hline
\vdots \\
\hline
y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\
\hline
\vdots \\
\hline
y_{m-1} \\
\hline
\end{array}
$$

Serial pseudo-code

- Key computation: row x column

$$y_i = \sum_{j=0}^{n-1} a_{ij} x_j$$

- In C,

```
/* For each row of A */
for (i = 0; i < m; i++) {
   y[i] = 0.0;
   /* For each element of the row and each element of x */
   for (j = 0; j < n; j++)
      y[i] += A[i][j]* x[j];
}
```

Using 3 Pthreads

Thread	Components of y
0	y[0], y[1]
1	y[2], y[3]
2	y[4], y[5]

thread 0

```
y[0] = 0.0;
for (j = 0; j < n; j++)
   y[0] += A[0][j]* x[j];
```

general case

```
y[i] = 0.0;
for (j = 0; j < n; j++)
   y[i] += A[i][j]*x[j];
```

Pthreads Matrix-Vector Multiplication

pth_mat_vect.c

```
void *Pth_mat_vect(void* rank) {
   long my_rank = (long) rank;
   int i, j;
   int local_m = m/thread_count;
   int my_first_row = my_rank*local_m;
   int my_last_row = (my_rank+1)*local_m - 1;

   for (i = my_first_row; i <= my_last_row; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
          y[i] += A[i][j]*x[j];
   }

   return NULL;
}  /* Pth_mat_vect */
```

20-Minute Exercise: Primality Check

- Ask the user to input a number and check if the number is prime
  - (Def) A prime number (or a prime) is a natural number greater than 1 that has no positive divisors other than 1 and itself
  - start with pth_prime.c

- Develop both your own series and Pthreads version
  - try 524,287 and 433,494,437
  - observe the time you use by “> time (your_program…)”

Critical Sections

- Estimating Pi

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots\right)$$

- Serial C/C++ code

```
double factor = 1.0;
double sum = 0.0;
for (i = 0; i < n; i++, factor = -factor) {
   sum += factor/(2*i+1);
}
pi = 4.0*sum;
```

Using a Dual-Core Processor

- Note that as n increases, the estimate with one thread gets better and better

	n			
	10^5	10^6	10^7	10^8
π	3.14159	3.141593	3.1415927	3.14159265
1 Thread	3.14158	3.141592	3.1415926	3.14159264
2 Threads	3.14158	3.141480	3.1413692	3.14164686

A Thread Function to Compute Pi

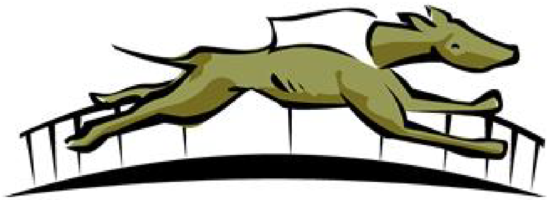
pth_pi.c

Anything wrong?!

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)  /* my_first_i is even */
      factor = 1.0;
   else  /* my_first_i is odd */
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      sum += factor/(2*i+1);
   }

   return NULL;
}  /* Thread_sum */
```

Possible Race Condition

Time	Thread 0	Thread 1
1	Started by main thread	
2	Call `Compute()`	Started by main thread
3	Assign `y = 1`	Call `Compute()`
4	Put `x=0` and `y=1` into registers	Assign `y = 2`
5	Add 0 and 1	Put `x=0` and `y=2` into registers
6	Store 1 in memory location x	Add 0 and 2
7		Store 2 in memory location x



Anything wrong? How do we avoid this problem?!

Busy Waiting

- A thread repeatedly tests a condition, but, effectively, does no useful work until the condition has the appropriate value.

- Beware of optimizing compilers, though!

```
y = Compute(my_rank);
while (flag != my_rank);
x = x + y;
flag++;
```

⇨ Busy waiting

**flag initialized to 0 by main thread

Be aware of “;” at the end of **while

Pthreads with Busy Waiting

pth_pi_busy1.c

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      while (flag != my_rank);
      sum += factor/(2*i+1);
      flag = (flag+1) % thread_count;
   }

   return NULL;
}  /* Thread_sum */
```

for what?!

Can you improve?!

Pthreads with Busy Waiting (2)

pth_pi_busy2.c

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor, my_sum = 0.0;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor)
      my_sum += factor/(2*i+1);

   while (flag != my_rank);
   sum += my_sum;
   flag = (flag+1) % thread_count;

   return NULL;
}  /* Thread_sum */
```

What's different?!

Mutexes

- A thread that is busy waiting may continually use the CPU accomplishing nothing
- mutex (mutual exclusion) is a special type of variable that can be used to restrict access to a critical section to a single thread at a time
  - used to guarantee that one thread "excludes" all other threads while it executes the critical section
- Pthreads standard includes a special type for mutexes: pthread_mutex_t.

```
int pthread_mutex_init(
      pthread_mutex_t*              mutex_p   /* out */
      const pthread_mutexattr_t*   attr_p    /* in  */);
```

More about Mutexes

- In order to gain access to a critical section a thread calls

```
int pthread_mutex_lock(pthread_mutex_t* mutex_p  /* in/out */);
```

- When a thread is finished executing the code in a critical section, it should call

```
int pthread_mutex_unlock(pthread_mutex_t* mutex_p  /* in/out */);
```

- When a Pthreads program finishes using a mutex, it should call

```
int pthread_mutex_destroy(pthread_mutex_t* mutex_p  /* in/out */);
```

Global Sum by mutex

pth_pi_mutex.c

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;
   double my_sum = 0.0;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      my_sum += factor/(2*i+1);
   }
   pthread_mutex_lock(&mutex);
   sum += my_sum;
   pthread_mutex_unlock(&mutex);

   return NULL;
}  /* Thread_sum */
```

use mutex

Compare Busy-Waiting w/ Mutex

Threads	Busy-Wait	Mutex
1	2.90	2.90
2	1.45	1.45
4	0.73	0.73
8	0.38	0.38
16	0.50	0.38
32	0.80	0.40
64	3.56	0.38

$$\frac{T_{\text{serial}}}{T_{\text{parallel}}} \approx \texttt{thread\_count}$$

Run-times (in seconds) of Pi programs using n = 108 terms on a system with two four-core processors.

Random Sequence of Busy Waiting

- Possible sequence of events with busy-waiting and more threads than cores

Time	flag	Thread				
		0	1	2	3	4
0	0	crit sect	busy wait	susp	susp	susp
1	1	terminate	crit sect	susp	busy wait	susp
2	2	—	terminate	susp	busy wait	busy wait
⋮	⋮			⋮	⋮	⋮
?	2	—	—	crit sect	susp	busy wait

END