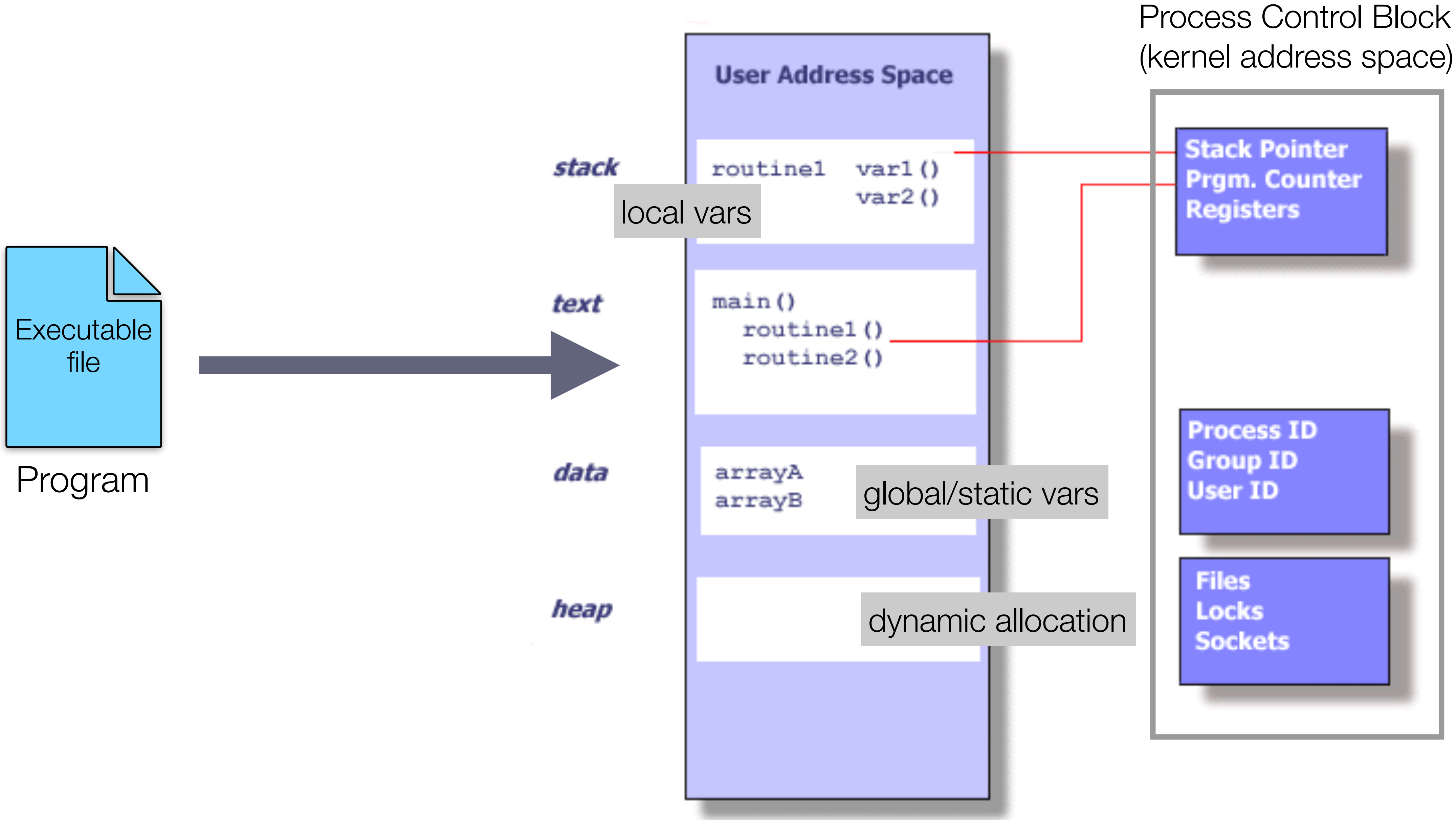
2110313 Operating Systems
Multithreaded Programming

Asst.Prof. Natawut Nupairoj, Ph.D.
Department of Computer Engineering
Chulalongkorn University
natawut.n@chula.ac.th

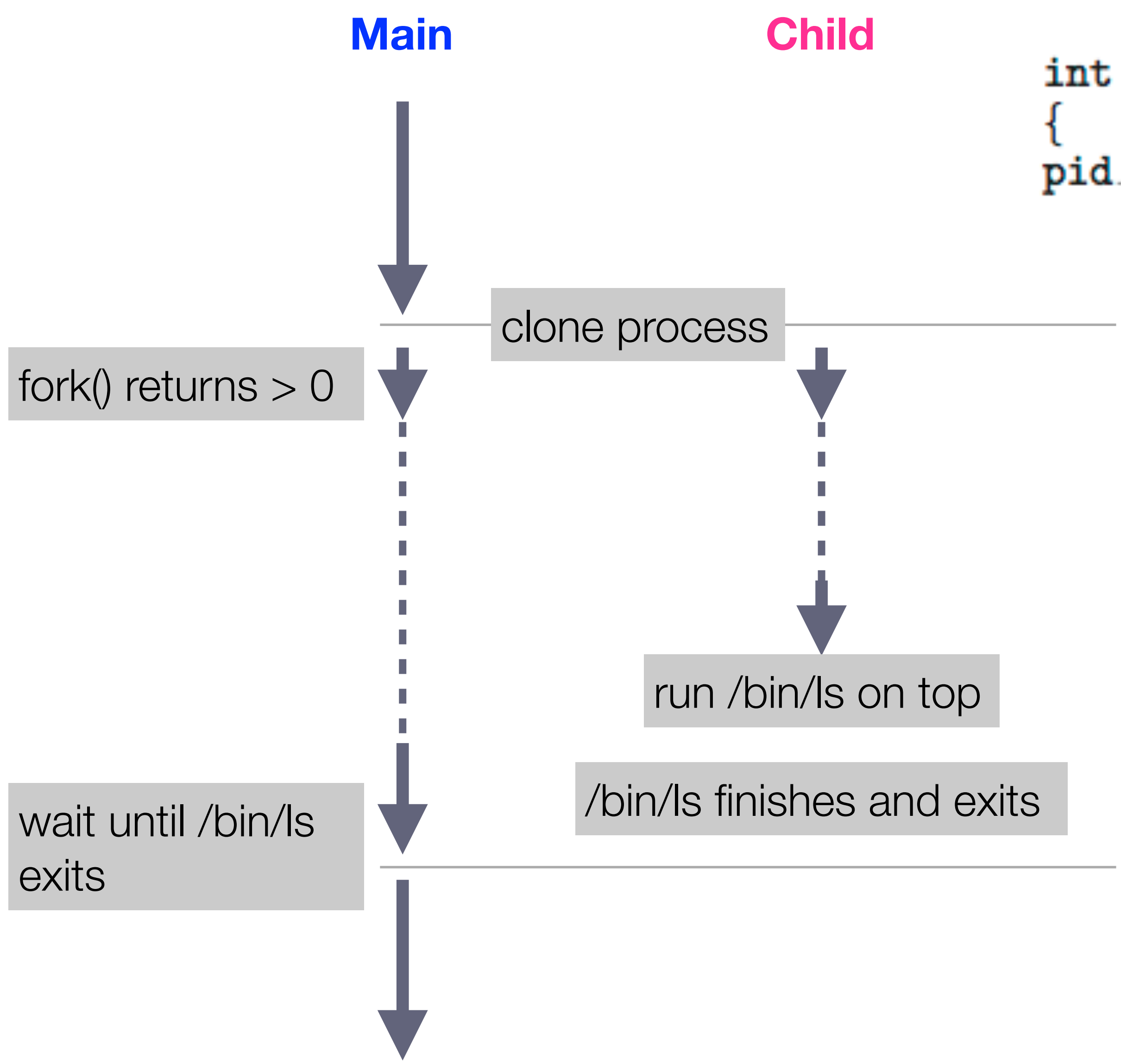
CHULA ENGINEERING
Foundation toward Innovation

Process

- Process = Program in Execution



Source: https://computing.llnl.gov/tutorials/pthreads/



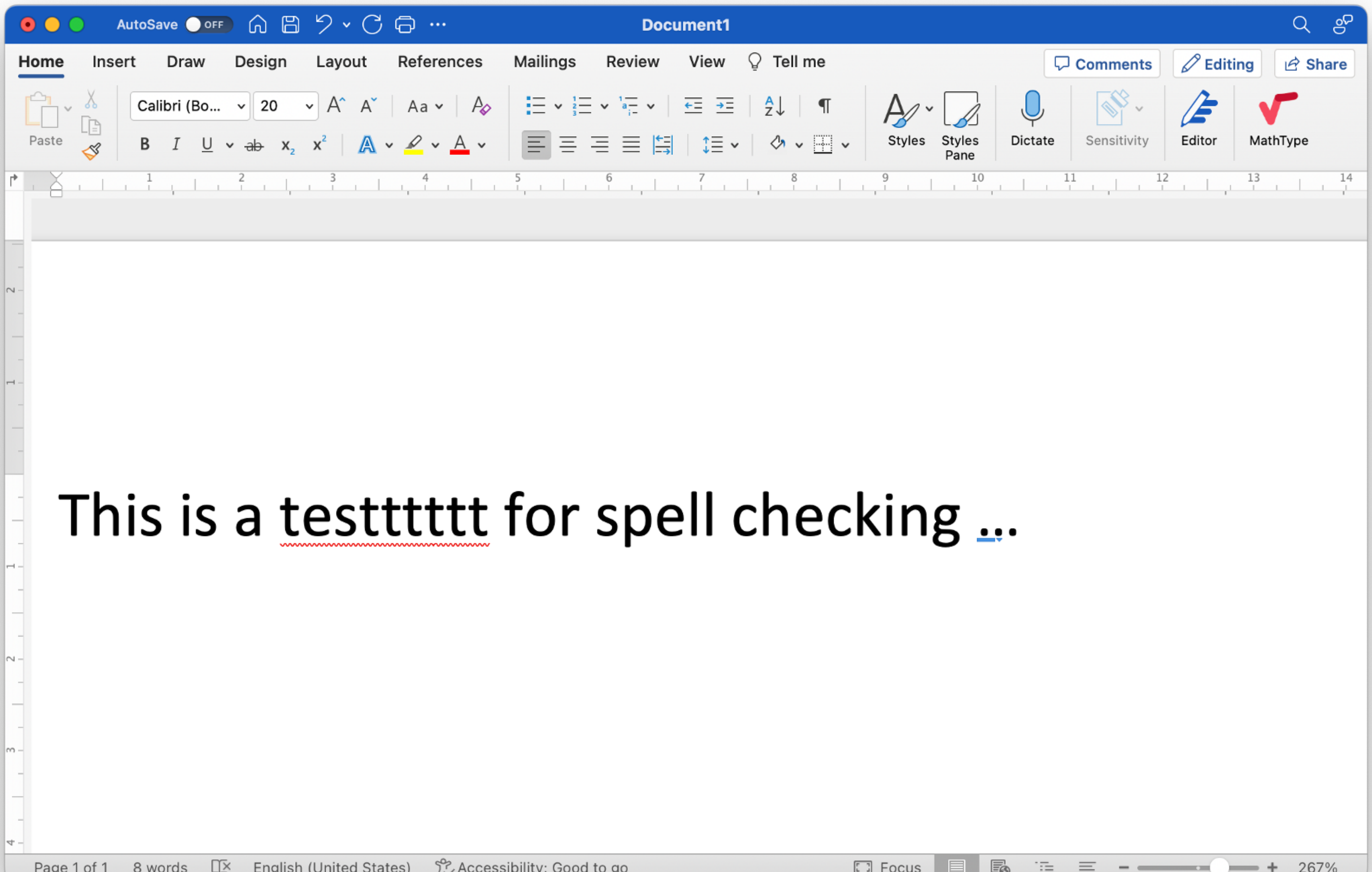
fork() returns 0

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Introduction to Thread

- 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

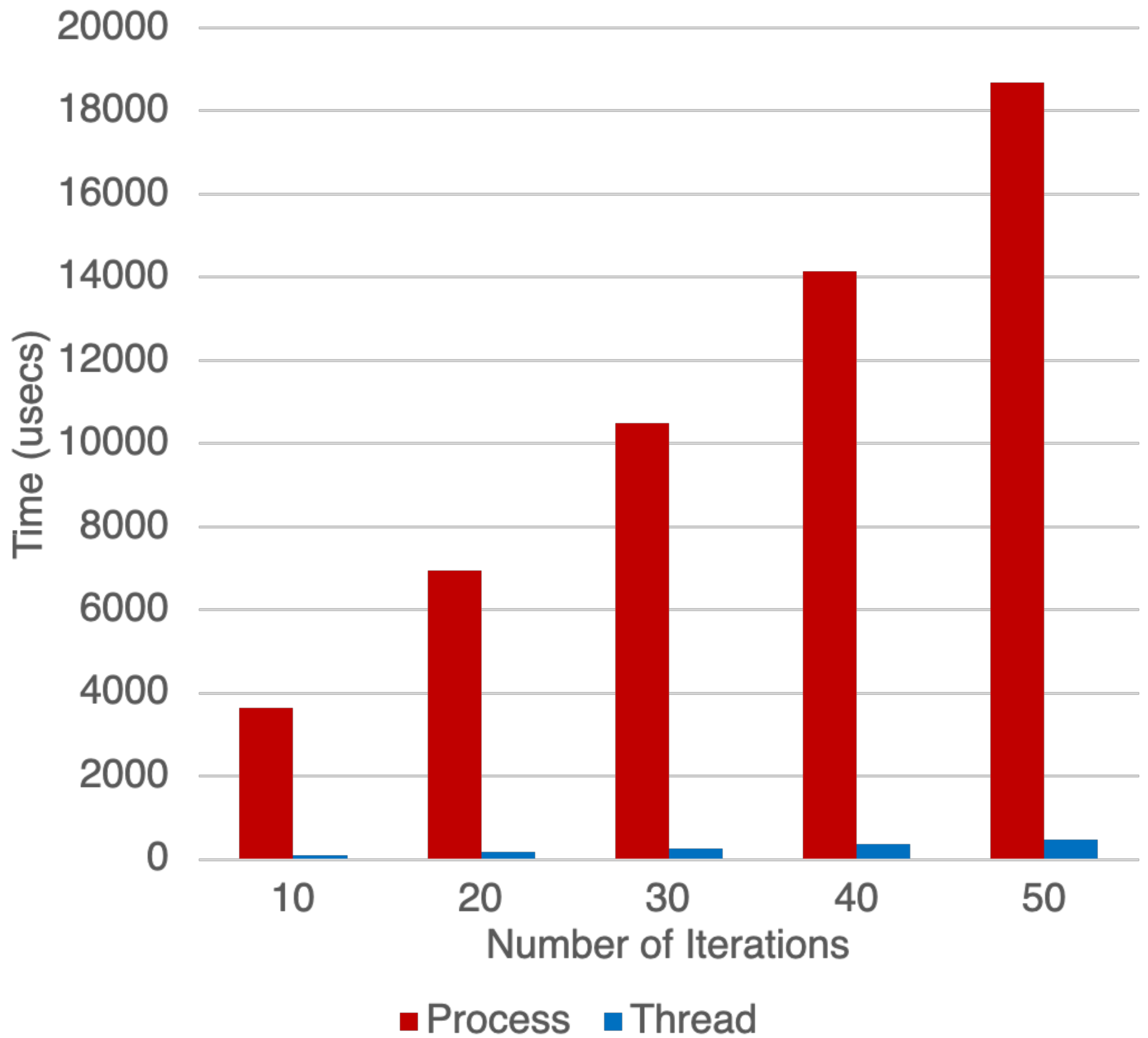
Microsoft Word - Spell Checking



Introduction to Thread

- Thread = Light-Weight Process
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Process/Thread Creation Overheads



Iterations	Process	Thread
10	3633.80	87.80
20	6947.80	168.20
30	10501.00	269.80
40	14135.00	359.40
50	18693.00	470.60

```
Type as fast as you can, press q to quit
The pressed key is
```

Count number of key pressed in a second

Keyhit Rating – No Thread

```
key_hit_count = 0
last_second_count = 0
t0 = current timestamp
While not stop:
      Check keyboard or timeout in 9,5 seconds
      If there is a key hit:
            increase key_hit_count by one
      t1 = current timestamp
      if t1 - t0 >= 1 second:
            key_hit_last_second = key_hit_count - last_second_count
            rate = key_hit_last_second / (t1 - t0)
            print rate
            last_second_count = key_hit_count
            t0 = t1
```

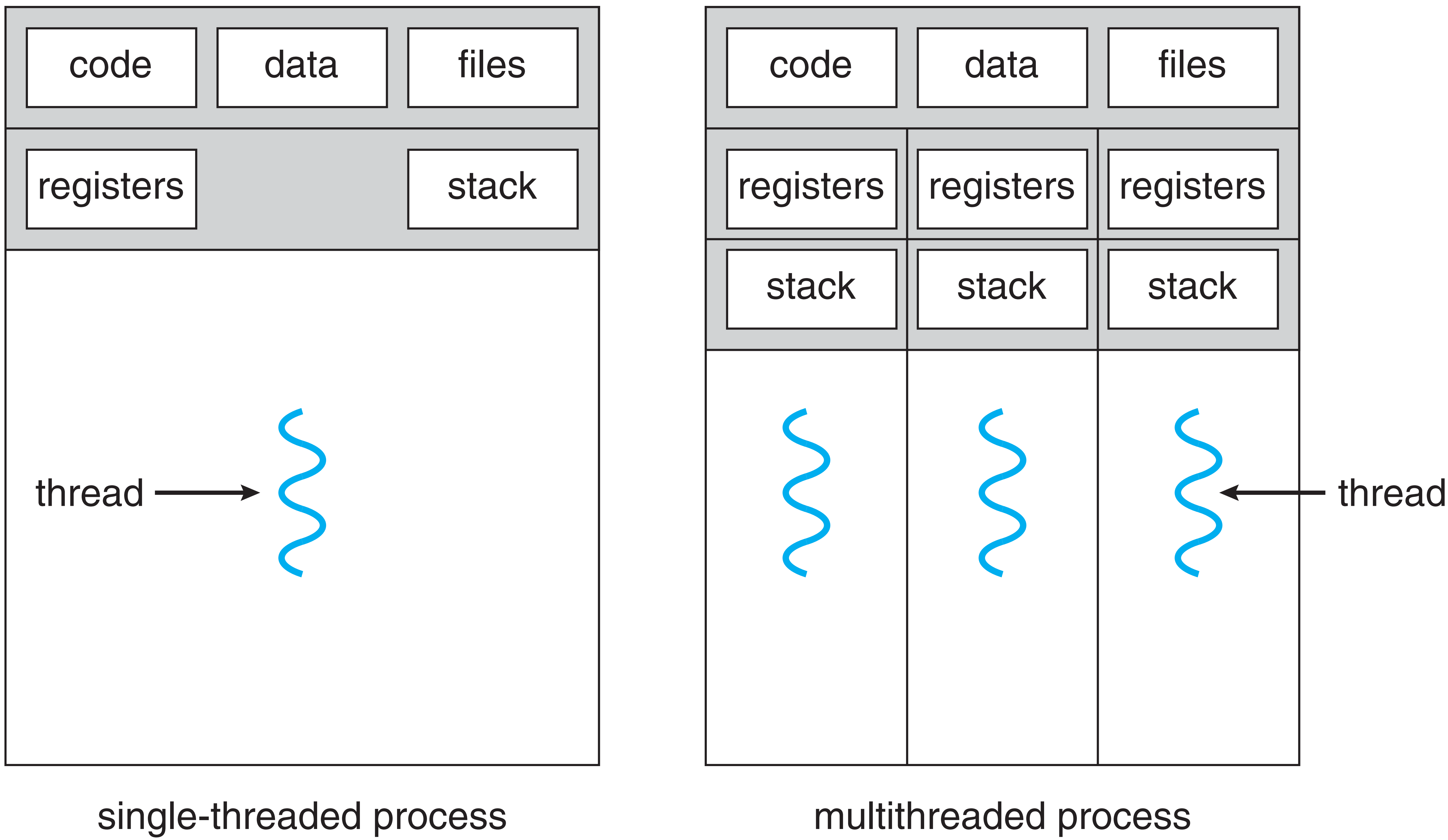
Keyhit Rating – Two Threads

```
Key_count_thread:
      While not stop:
            Wait for keyhit
            increase key_hit_count by one
Rating_thread:
      While not stop:
            sleep for 1 second
            key_hit_last_second = key_hit_count - last_second_count
            rate = key_hit_last_second / (t1 - t0)
            print rate
            last_second_count = key_hit_count
Main:
      key_hit_count = 0
      last_second_count = 0
      Start Rating_thread
      Start Key_count_thread
```

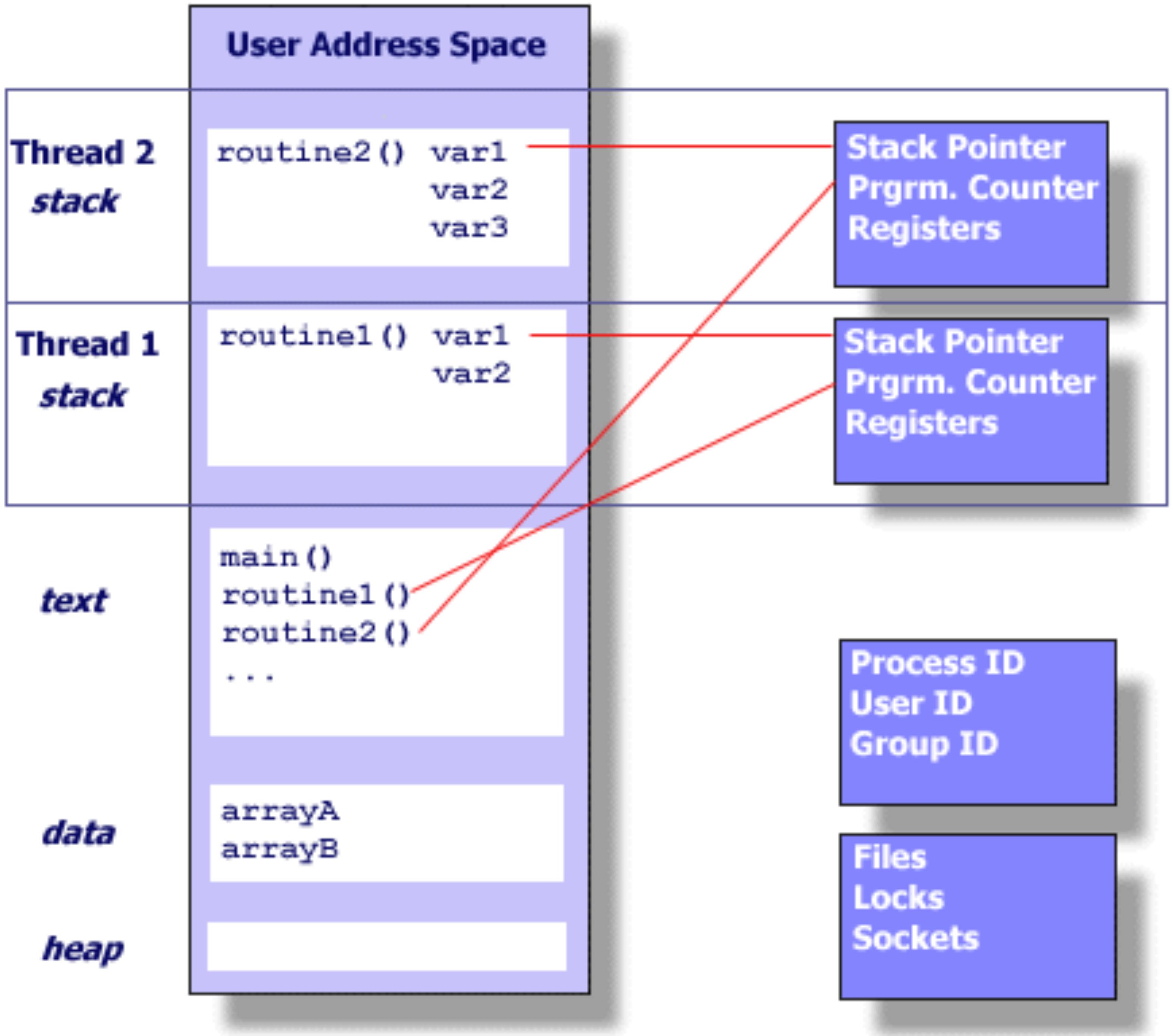
Inside Thread

- Each thread has its program counter and stack (local variables)
- All threads share code, data (global / static variables), heap (dynamic allocation)
- Each thread executes at its own pace

Single-Threaded vs. Multithreaded Processes

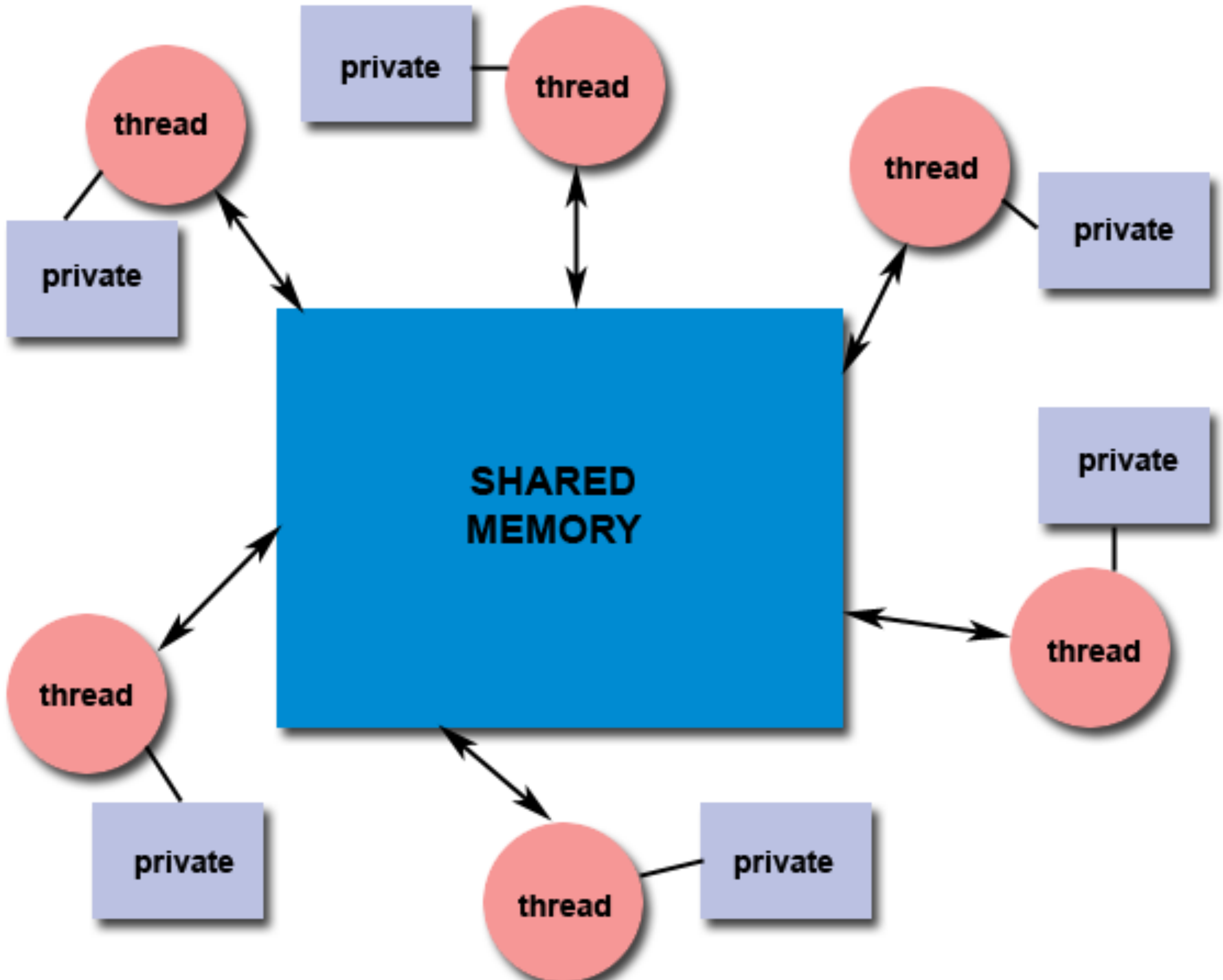


Threads and Process Memory Spaces



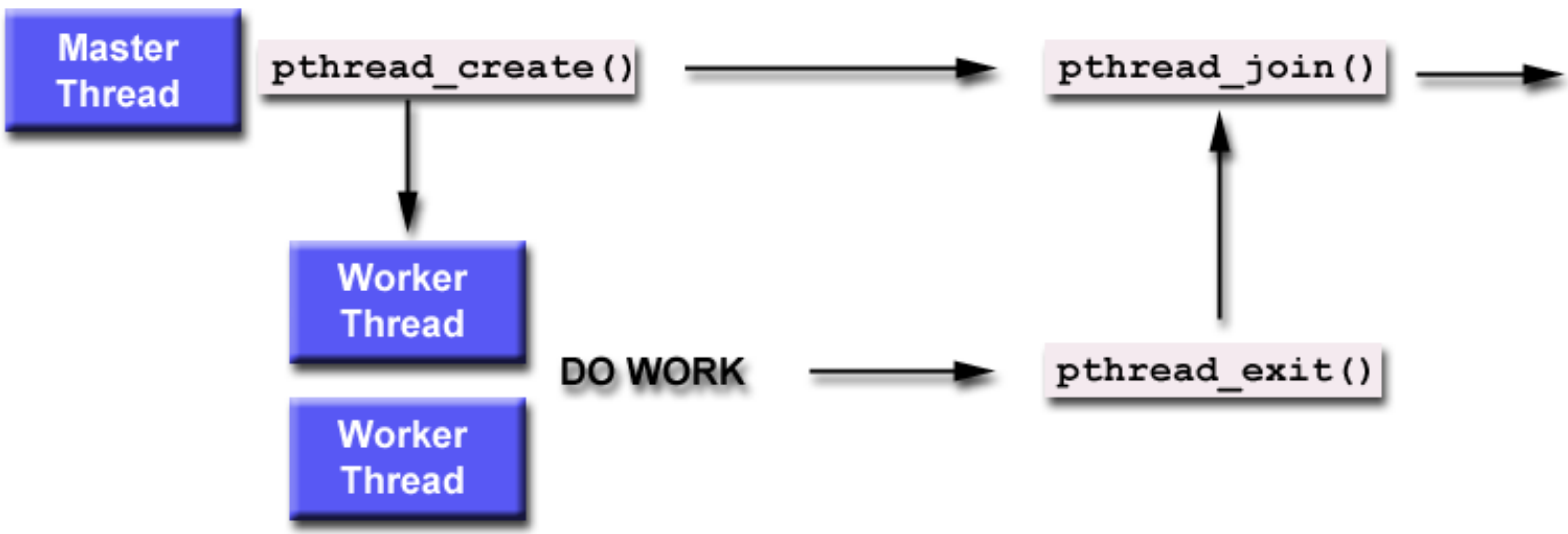
Source: https://computing.llnl.gov/tutorials/pthreads/

Shared Memory Model



Source: https://computing.llnl.gov/tutorials/pthreads/

Thread Creation / Termination



Source: https://computing.llnl.gov/tutorials/pthreads/

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NUM_THREADS     4

void *BusyWork(void *t)
{
   int i;
   long tid;
   double result=0.0;
   tid = (long)t;
   printf("Thread %ld starting...\n",tid);
   for (i=0; i<1000000; i++)
   {
      result = result + sin(i) * tan(i);
   }
   printf("Thread %ld done. Result = %e\n",tid, result);
   pthread_exit((void*) t);
}
```

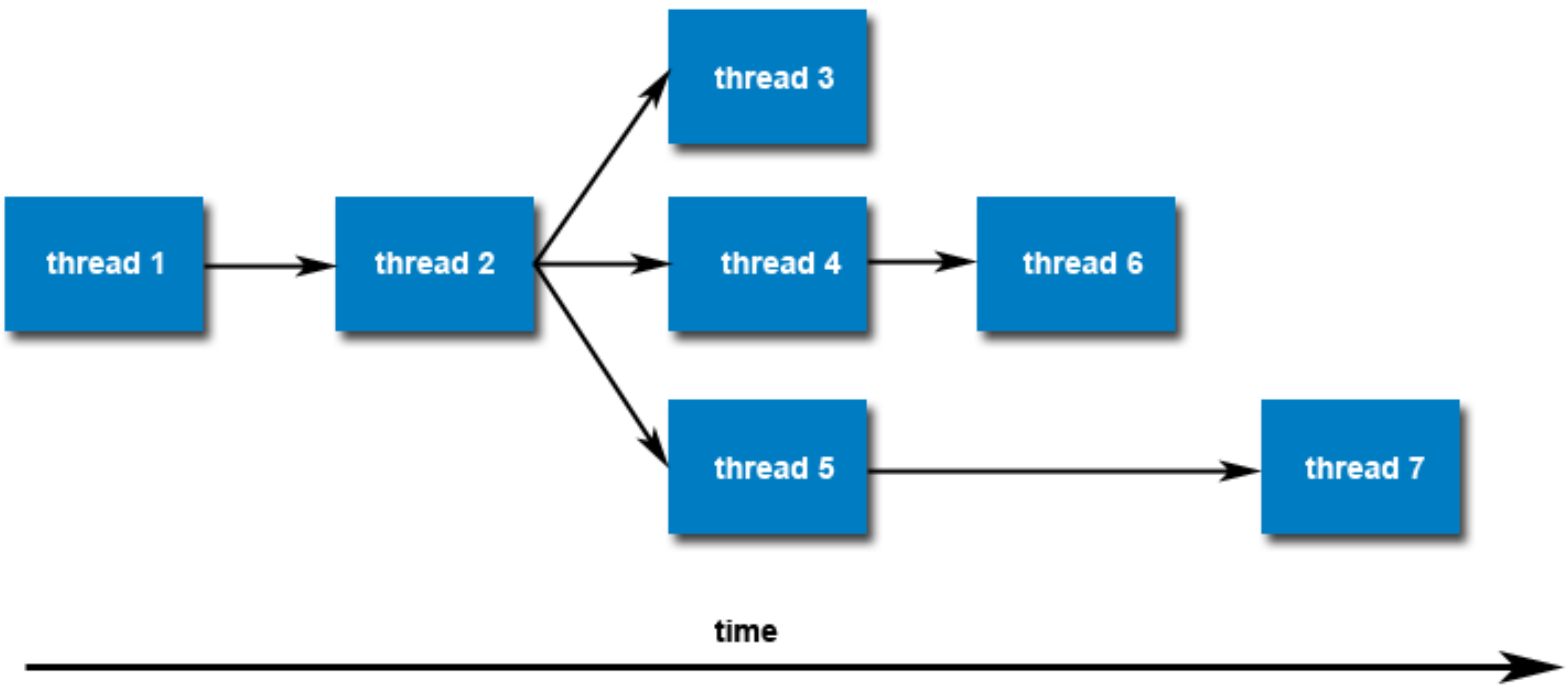
Source: https://computing.llnl.gov/tutorials/pthreads/

```c
int main (int argc, char *argv[])
{
   pthread_t thread[NUM_THREADS];
   pthread_attr_t attr;
   int rc;
   long t;
   void *status;

   /* Initialize and set thread detached attribute */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   for(t=0; t<NUM_THREADS; t++) {
      printf("Main: creating thread %ld\n", t);
      rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
      if (rc) {
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
         }
      }

   /* Free attribute and wait for the other threads */
   pthread_attr_destroy(&attr);
   for(t=0; t<NUM_THREADS; t++) {
      rc = pthread_join(thread[t], &status);
      if (rc) {
         printf("ERROR; return code from pthread_join() is %d\n", rc);
         exit(-1);
         }
      printf("Main: completed join with thread %ld having a status
            of %ld\n",t,(long)status);
      }

printf("Main: program completed. Exiting.\n");
pthread_exit(NULL);
}
```

Source: https://computing.llnl.gov/tutorials/pthreads/

Once created, threads are peers, and may create other threads. There is no implied hierarchy or dependency between threads.

ยังไม่มีการแบ่งระดับ Thread

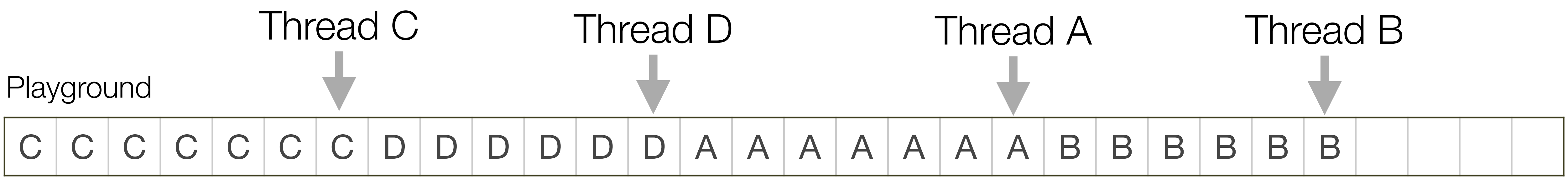


Source: https://computing.llnl.gov/tutorials/pthreads/

Thread Scheduling

- Order of starting execution of threads can be arbitrary
- CPU times are also allocated randomly similar to process scheduling
- Thus, we should write a program to NOT depend on the order of thread execution

order ตรงนี้ไม่ได้เท่ากับ order ที่สร้าง




```c
void *MyThread(void *arg)
{
    struct timeval end;

    char my_call_sign = (char)*((char *)arg);
    sleep(1);
    for(int i=0 ; i < PLAYGROUND_SIZE ; i++) {
        for(int k=0 ; k < 10000 ; k++)
            ;
        playground[i] = my_call_sign;
    }
    gettimeofday(&end, 0);
    double elapsed_thread = elapsed_time(&t_start, &end);
    printf("Thread %c ends at time %f\n", my_call_sign, elapsed_thread);
    pthread_exit(NULL);
}
```

```
(base) natawut@Natawuts-Mac-mini os-class % ./attack
main(): Created 4 threads.
Thread D ends at time 1.103172
Thread A ends at time 1.103241
Thread C ends at time 1.108047
Thread B ends at time 1.112472
[0] 218
[1] 3653
[2] 6294
[3] 75
----
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
(base) natawut@Natawuts-Mac-mini os-class %
```

Passing Arguments to Threads

Correct

```
long taskids[NUM_THREADS];

for(t=0; t<NUM_THREADS; t++)
{
   taskids[t] = t;
   printf("Creating thread %ld\n", t);
   rc = pthread_create(&threads[t], NULL, PrintHello, (void *) taskids[t]);
   ...
}
```

Incorrect

```
int rc;
long t;

for(t=0; t<NUM_THREADS; t++)
{
   printf("Creating thread %ld\n", t);
   rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &t);
   ...
}
```

Source: https://computing.llnl.gov/tutorials/pthreads/

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

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)
{
   struct thread_data *my_data;
   ...
   my_data = (struct thread_data *) threadarg;
   taskid = my_data->thread_id;
   sum = my_data->sum;
   hello_msg = my_data->message;
   ...
}

int main (int argc, char *argv[])
{
   ...
   thread_data_array[t].thread_id = t;
   thread_data_array[t].sum = sum;
   thread_data_array[t].message = messages[t];
   rc = pthread_create(&threads[t], NULL, PrintHello,
        (void *) &thread_data_array[t]);
   ...
}
```

pointer

Source: https://computing.llnl.gov/tutorials/pthreads/

Multicore Programming ให้ thread มีประสิทธิภาพ

- Multicore or multiprocessor systems putting pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging

Multicore Programming

- **Parallelism** implies a system can perform more than one task simultaneously

real multi processor/core

- **Concurrency** supports more than one task making progress

ทำสลับไปมาจนดูว่า simultaneously

  - Single processor / core, scheduler providing concurrency

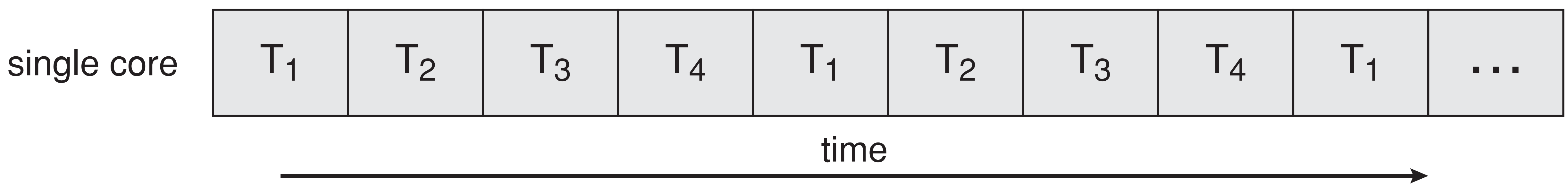
- Types of parallelism

เหมือนๆ กันบน data ต่างกัน (GPU)

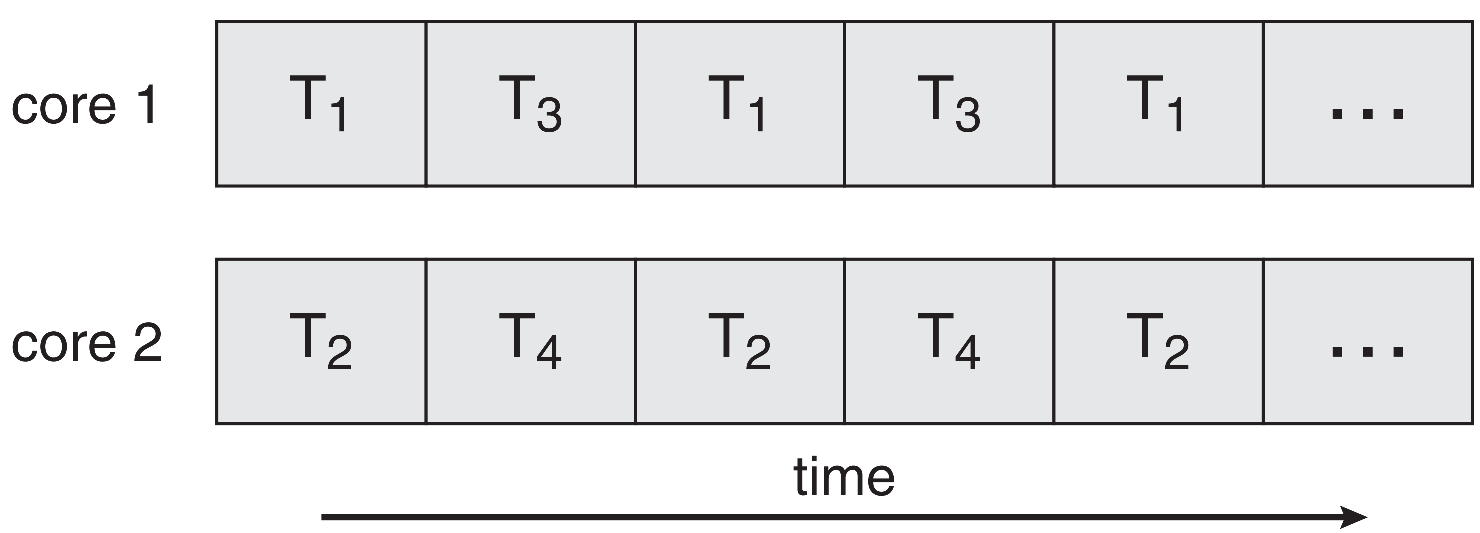
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation

Concurrency vs. Parallelism

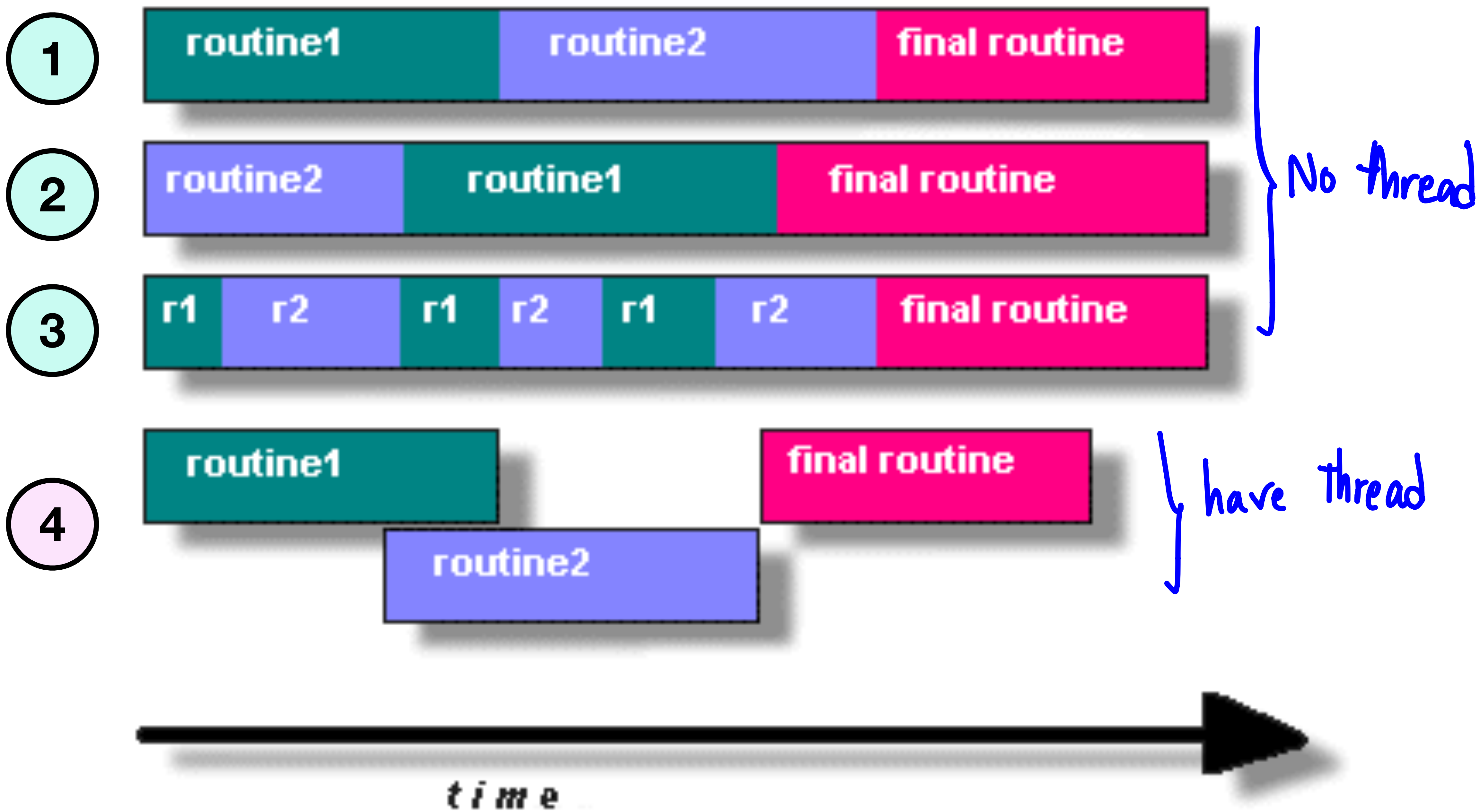
- Concurrent execution on single-core system:



- Parallelism on a multi-core system:

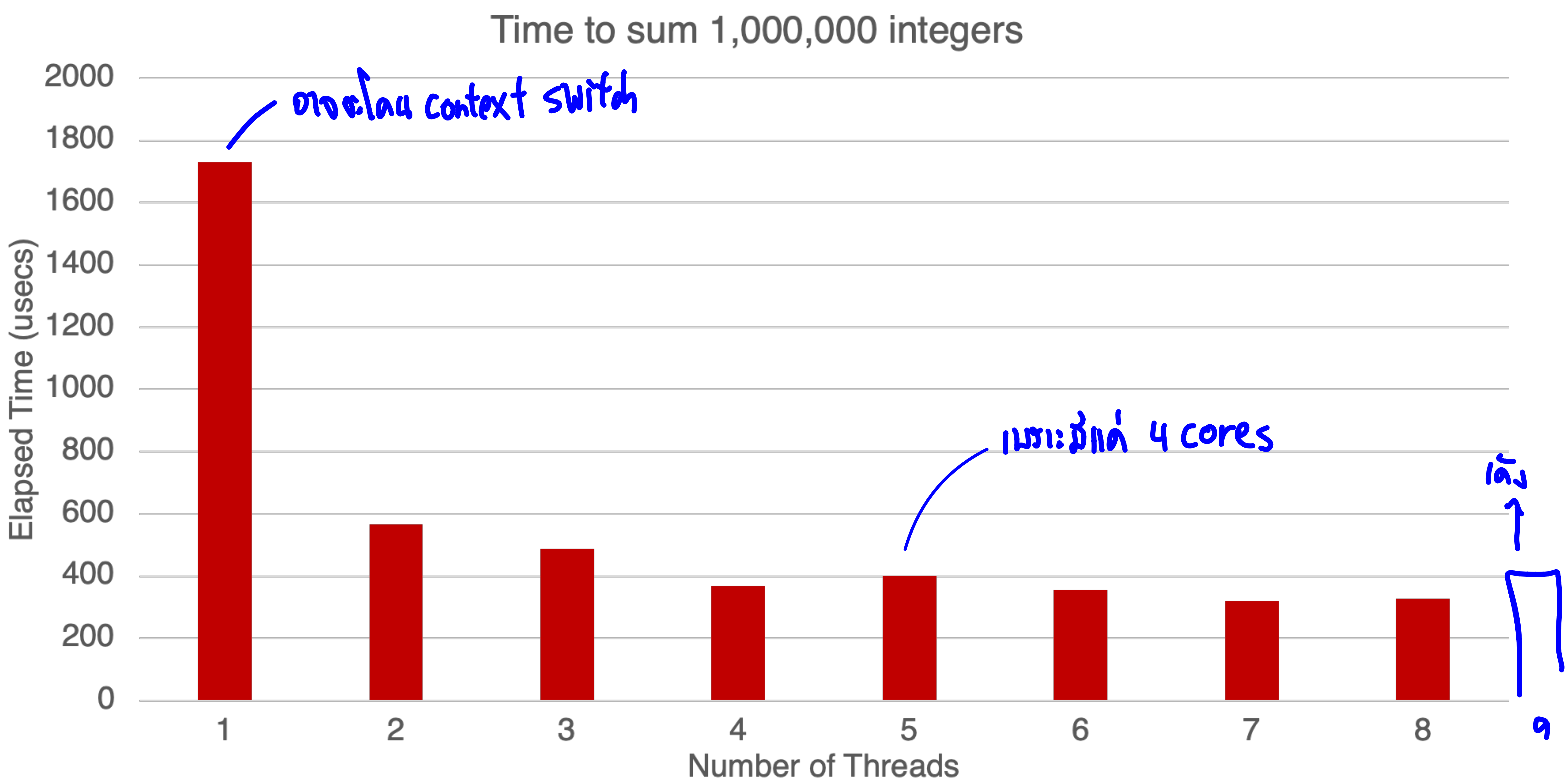


Execution Patterns of Interleave and Overlap



Source: https://computing.llnl.gov/tutorials/pthreads/

Multithreading Performance - Parallel Sum



Threading Issues

Threadที่เราเขียนProgram

Thread of OS

- User Thread vs. Kernel Thread
- Thread Libraries
- Thread Pools
- fork()/exec() and Thread
- Single and cancelation

User Thread vs. Kernel Thread

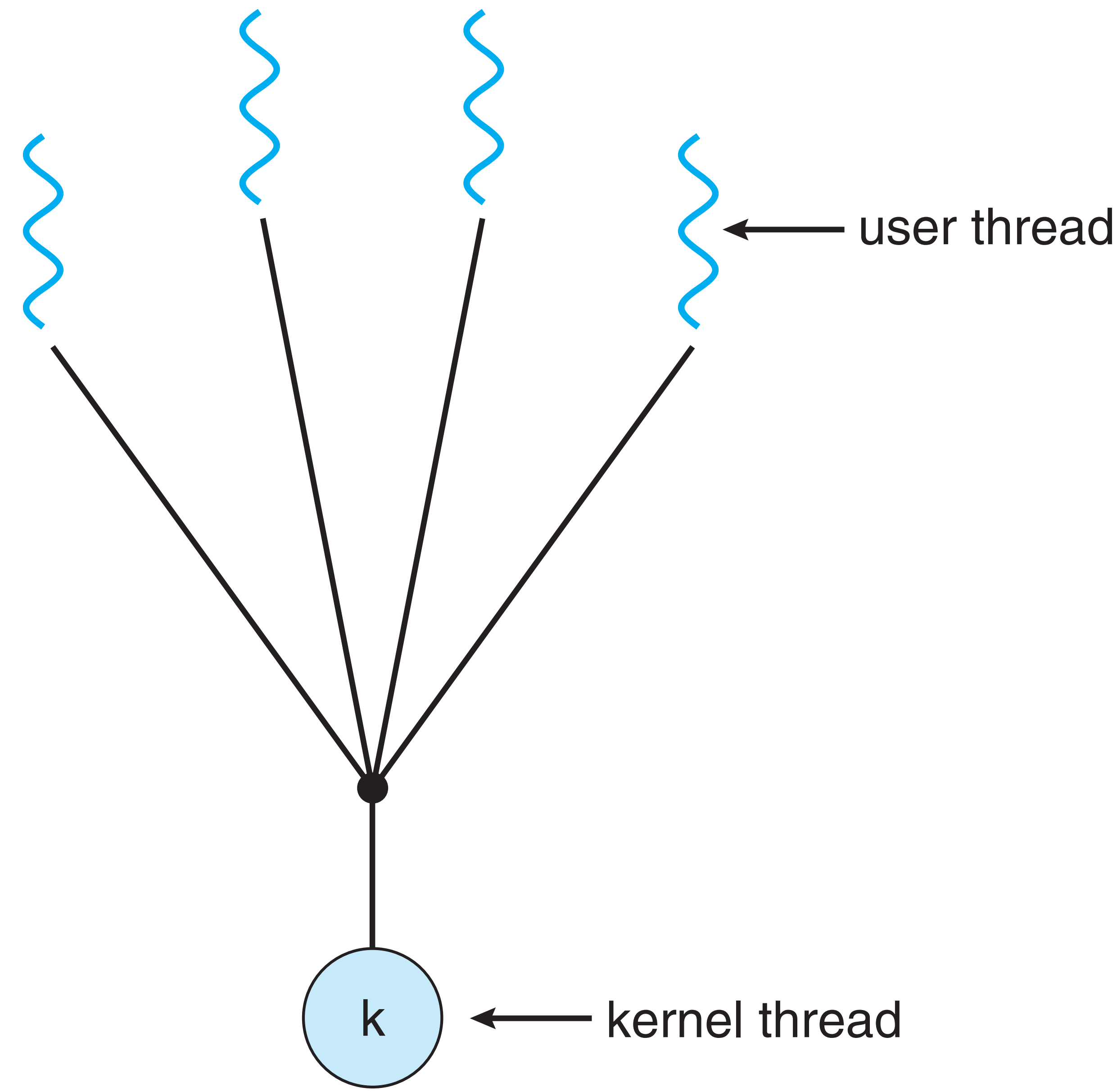
- User threads
  - management done by user-level threads library
- Example thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads
  - Python threads

User Thread vs. Kernel Thread

- Kernel threads - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Linux
  - MacOS
- How are user threads mapped to kernel threads?

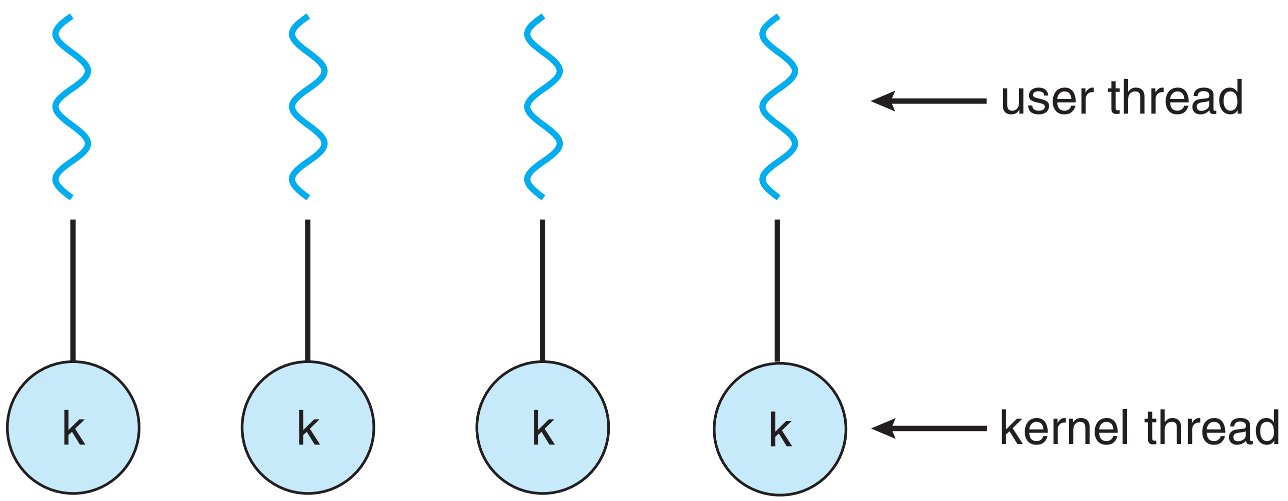
Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Examples:
  - Old OS (e.g. Solaris)
  - GNU Portable Threads



One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples:
  - Windows NT/XP/2000
  - Linux

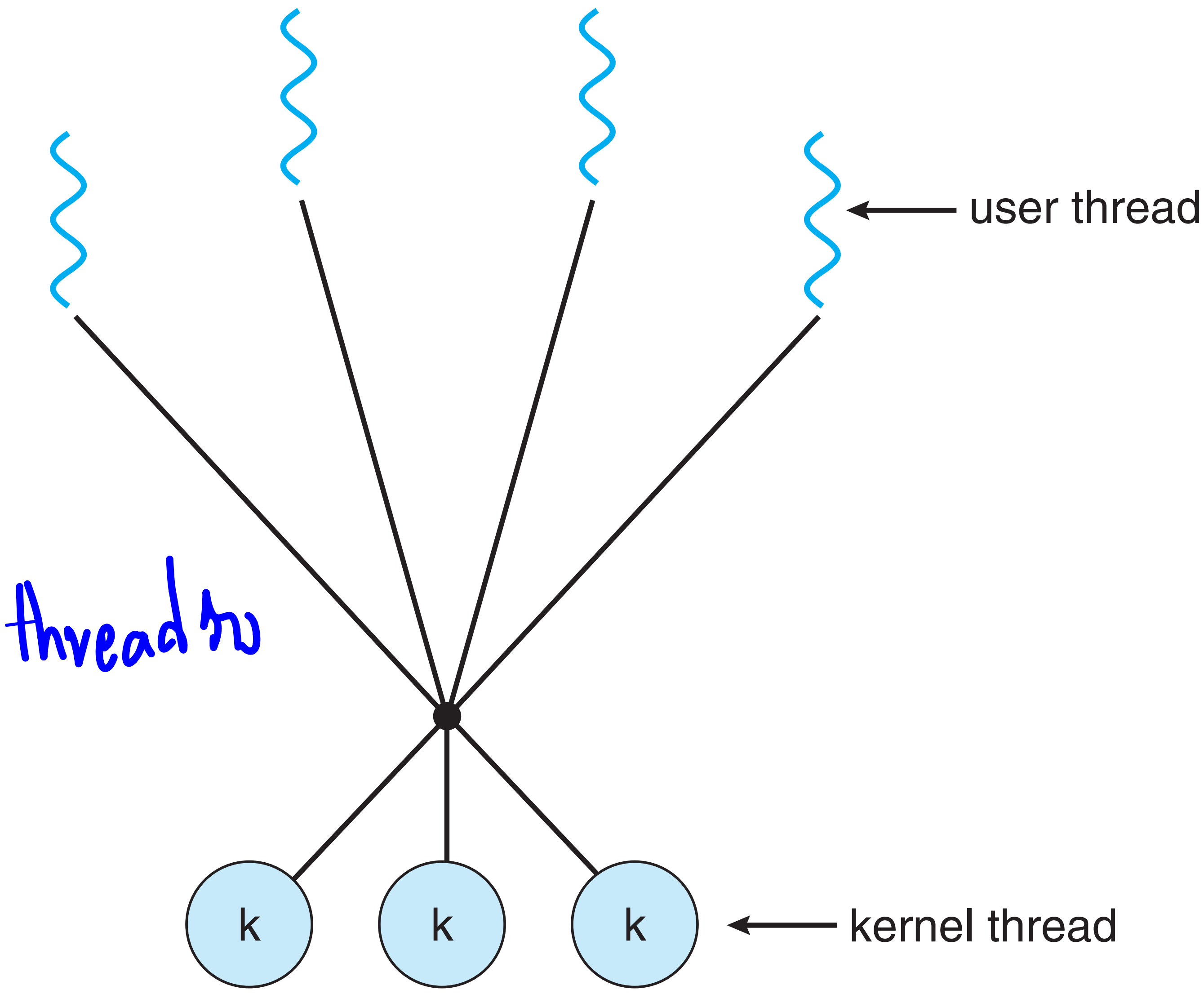


Many-to-Many

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Similar to Thread Pool

ต้องรอ kernel thread ว่าง

- Examples:
  - Old OS (Solaris < version 9)
  - Windows NT/2000 with the ThreadFiber package
- Variation: Two-level model



Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread เรียกใช้ไวกว่าสร้างใหม่
  - Allows the number of threads in the application(s) to be bound to the size of the pool ถ้า require เกิน → เข้า queue
  - Separating task to be performed from mechanics of creating task allows different strategies for running task
    - i.e.Tasks could be scheduled to run periodically

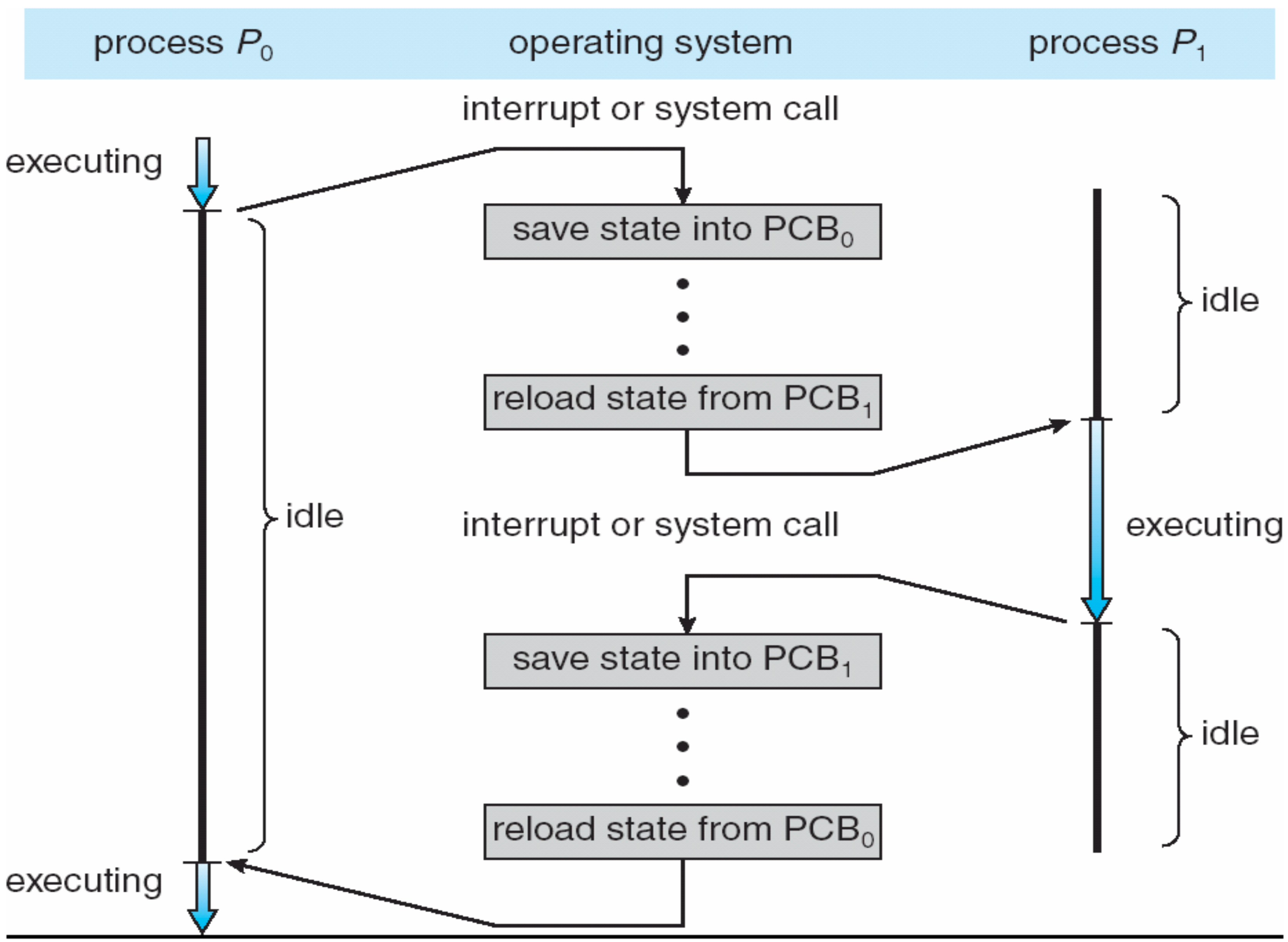
fork()/exec() and Thread

- Does fork() duplicate only the calling thread or all threads? No, process ลูกไม่ clone thread
- Exec() usually works as normal – replace the running process including all threads

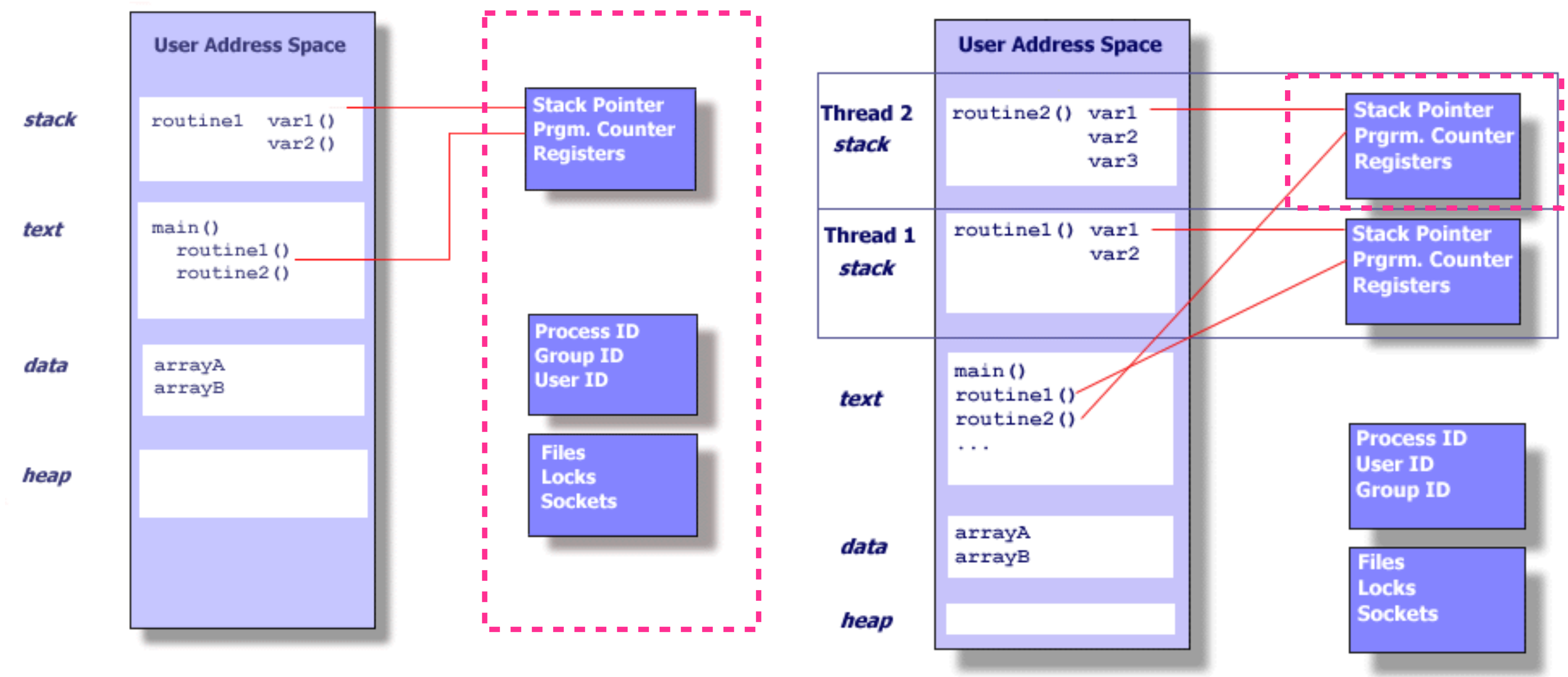
Benefits of Using Threads

- Light Weight
  - Lesser overheads to create a new thread than a new process
  - More efficient inter-thread vs. inter-process communication
  - Lesser overheads for context switching (no need to deal with memory)

Context Switching Overheads



Context Switching vs. Thread Switching



Benefits of Using Threads

- Effective Usages of Multi-Core/Multi-Thread CPU with Parallel Programming
  - Utilize more CPU cores / threads
  - Overlapping of executions (CPU vs. CPU, CPU vs. I/O)
- Simplify Program
  - Tasks can be divided among threads
  - Each thread is responsible for its own job
  - Threads work on the same data using shared memory
  - Note that synchronization between threads is needed