DCS216 Operating Systems

Lecture 10
Threads (1)

Mar 27th, 2024

Instructor: Xiaoxi Zhang

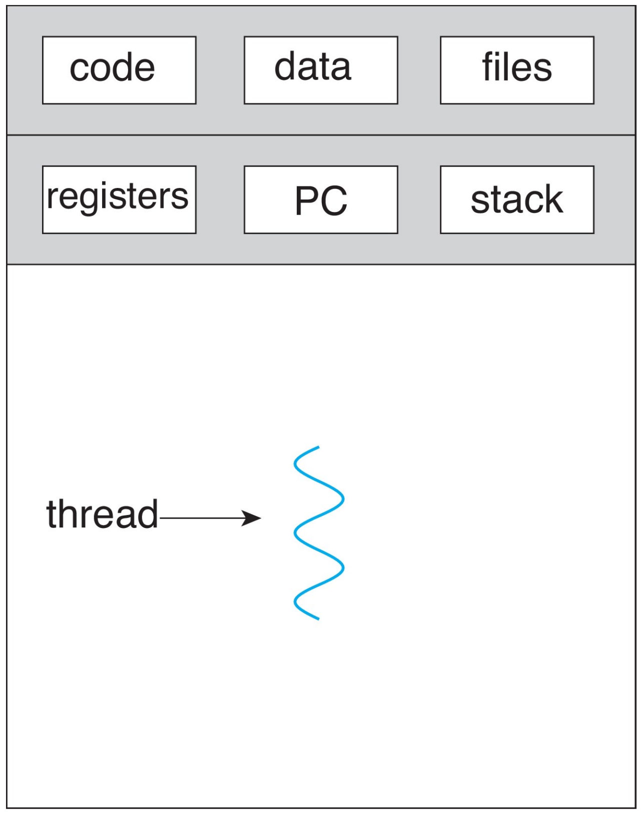
Sun Yat-sen University

Content

- **Content**

Overview

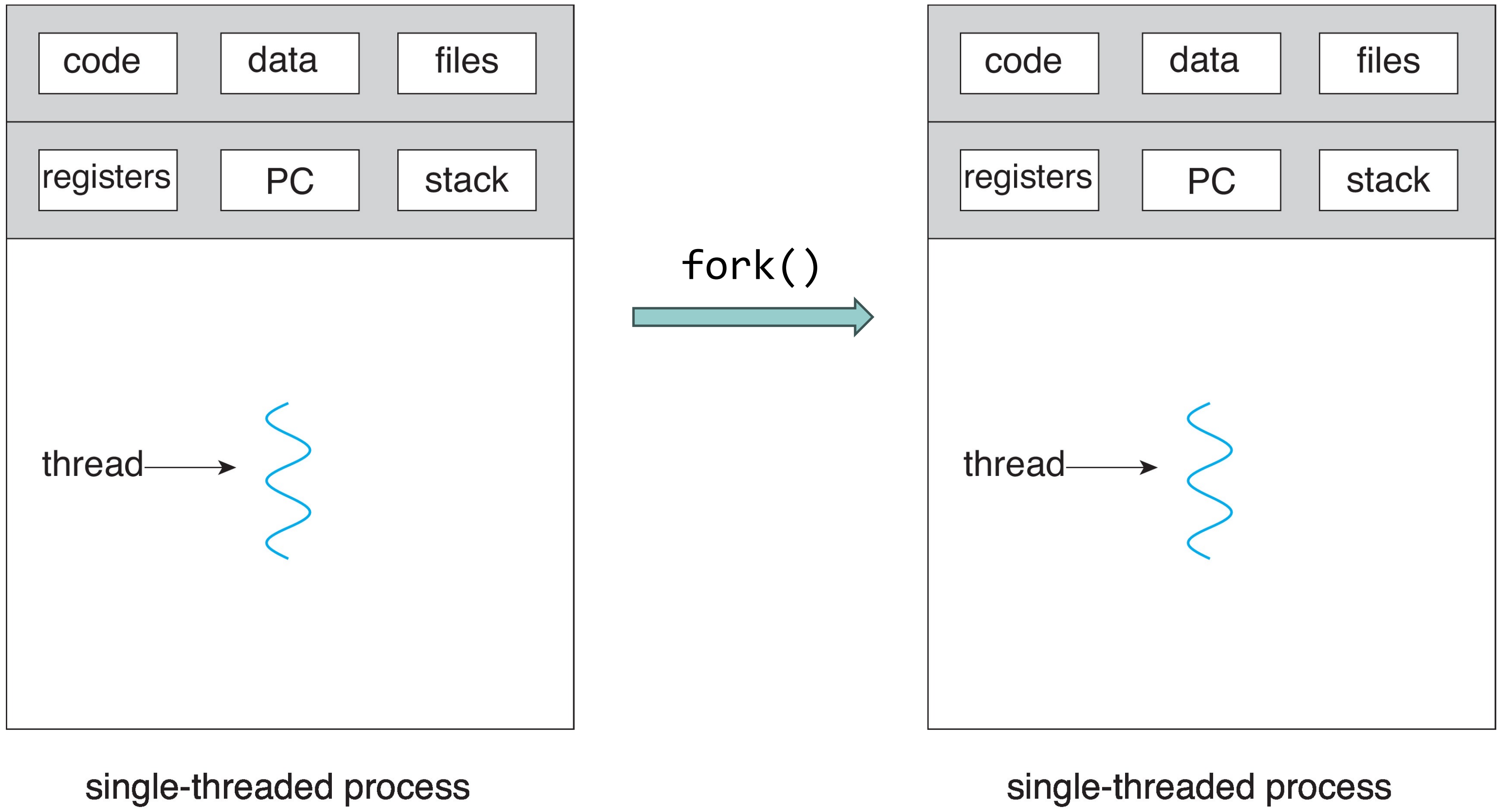
- **Single-threaded Process**
  - So far, we have only considered a process to have a single unique sequence of execution context, aka., single-threaded process.



single-threaded process

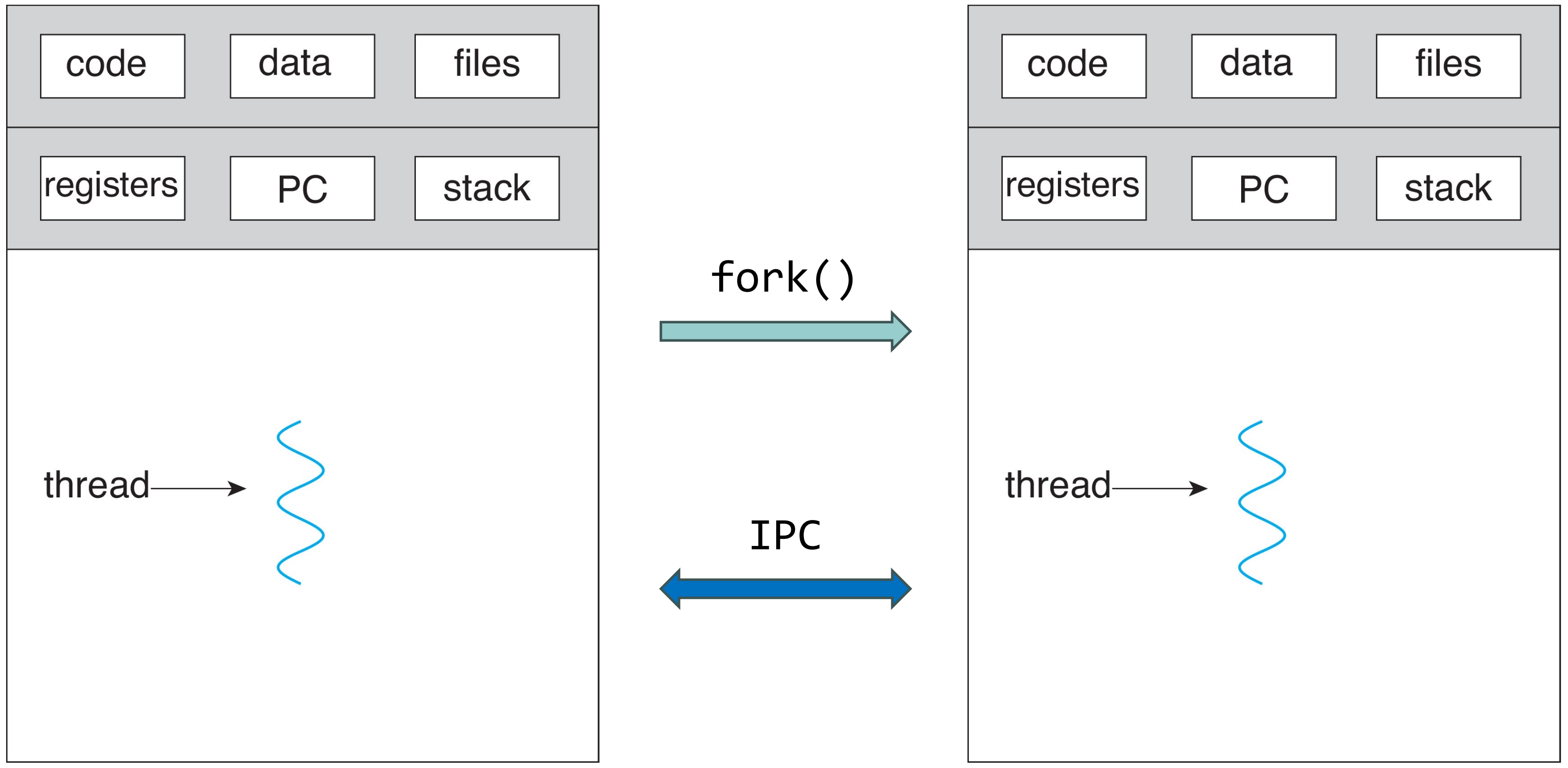
Overview

- **Single-threaded Process**
  - In cases where we have to perform a separate task, we `fork()` and `exec()` another process



Overview

- **Single-threaded Process**
  - In cases where we have to perform a separate task, we `fork()` and `exec()` another process, possibly communicating via IPC.



single-threaded process

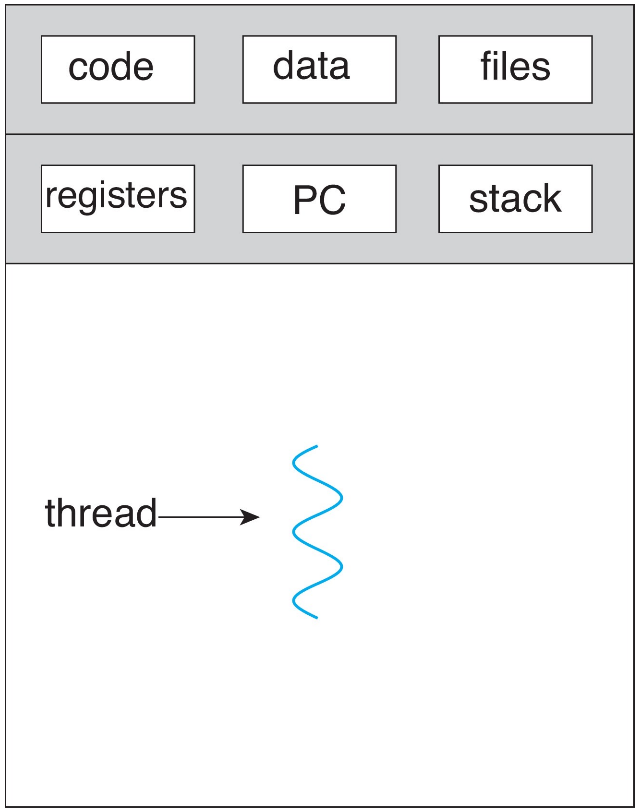
single-threaded process

Overview

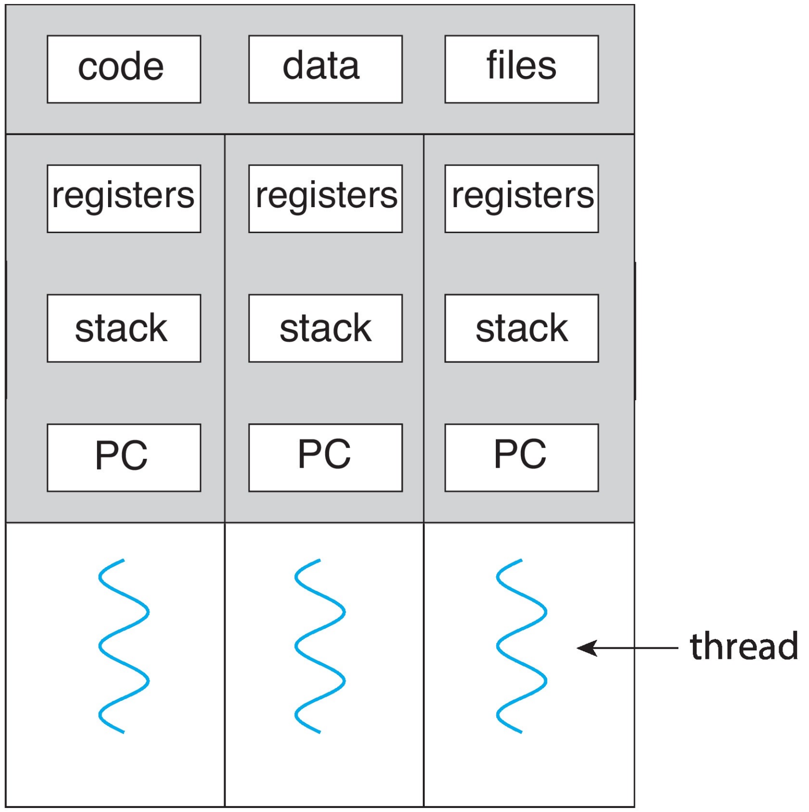
- **Single-threaded Process**
  - In cases where we have to perform a separate task, we `` `fork()` `` and `` `exec()` `` another process, possibly communicating via IPC.
    - Creating and managing processes is generally considered **expensive**
      - E.g., `fork()` system call
    - Context switching **between processes** is pure overhead, and **expensive**
      - Store PCB(old) and Load PCB(new)
    - IPC between processes is **expensive**
      - Setup Shared Memory (system calls)
      - Message Passing (system calls for every `read()` and `write()`)
      - Explicit coordination and management required from programmers
  - Is there a better (cheaper) way to simply spawn a new task?
    - Multi-threaded Process

Overview

- **Single-threaded and Multi-threaded Processes**
  - An oversimplified View: Process ~= Address Space + Threads
  - Each thread represents single unique execution context
    - Program Counter, Registers, Execution Flags, Stack



single-threaded process



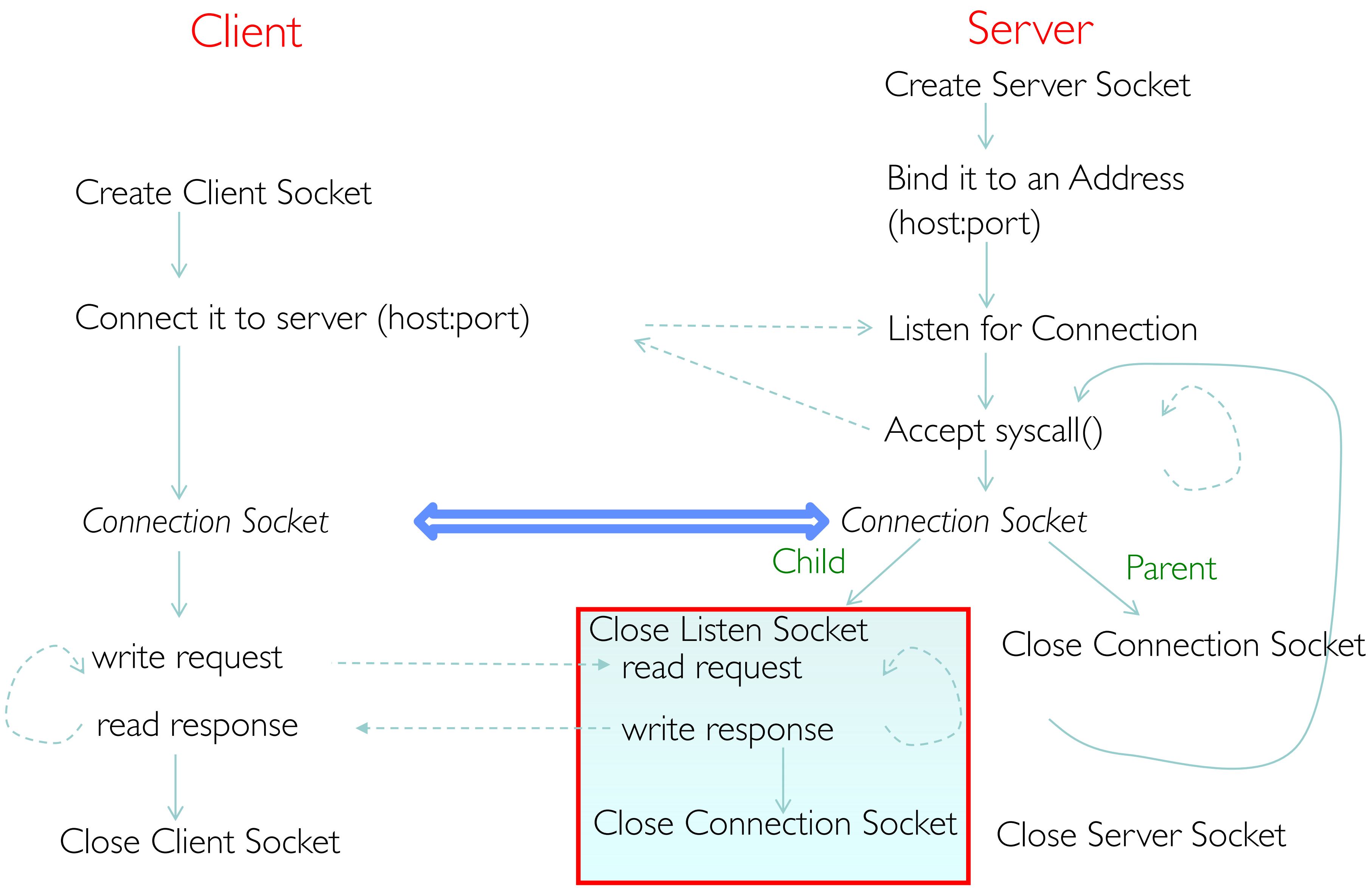
multithreaded process

Overview

- **Single-threaded and Multi-threaded Processes**
  - An oversimplified View: Process ~= Address Space + Threads
  - Each thread represents single unique execution context
    - Program Counter, Registers, Execution Flags, Stack
  - Use threads to spawn a new task (vs. Processes)
    - Creating and managing threads is less expensive than processes
      - No need to allocate a separate memory space or duplicate resources since they are shared among threads within a process
    - Context switching between threads is less expensive
      - Store TCB(old) and Load TCB(new)
        - TCB == **T**hread **C**ontrol **B**lock contains way less info than **PCB**
    - Communication between threads is cheaper than IPC
      - Threads share the same address space → direct communication
  - Okay! But there is no such thing as a free lunch. What's the catch?
    - Well, with Threads, you sacrificed Protection provided by processes.

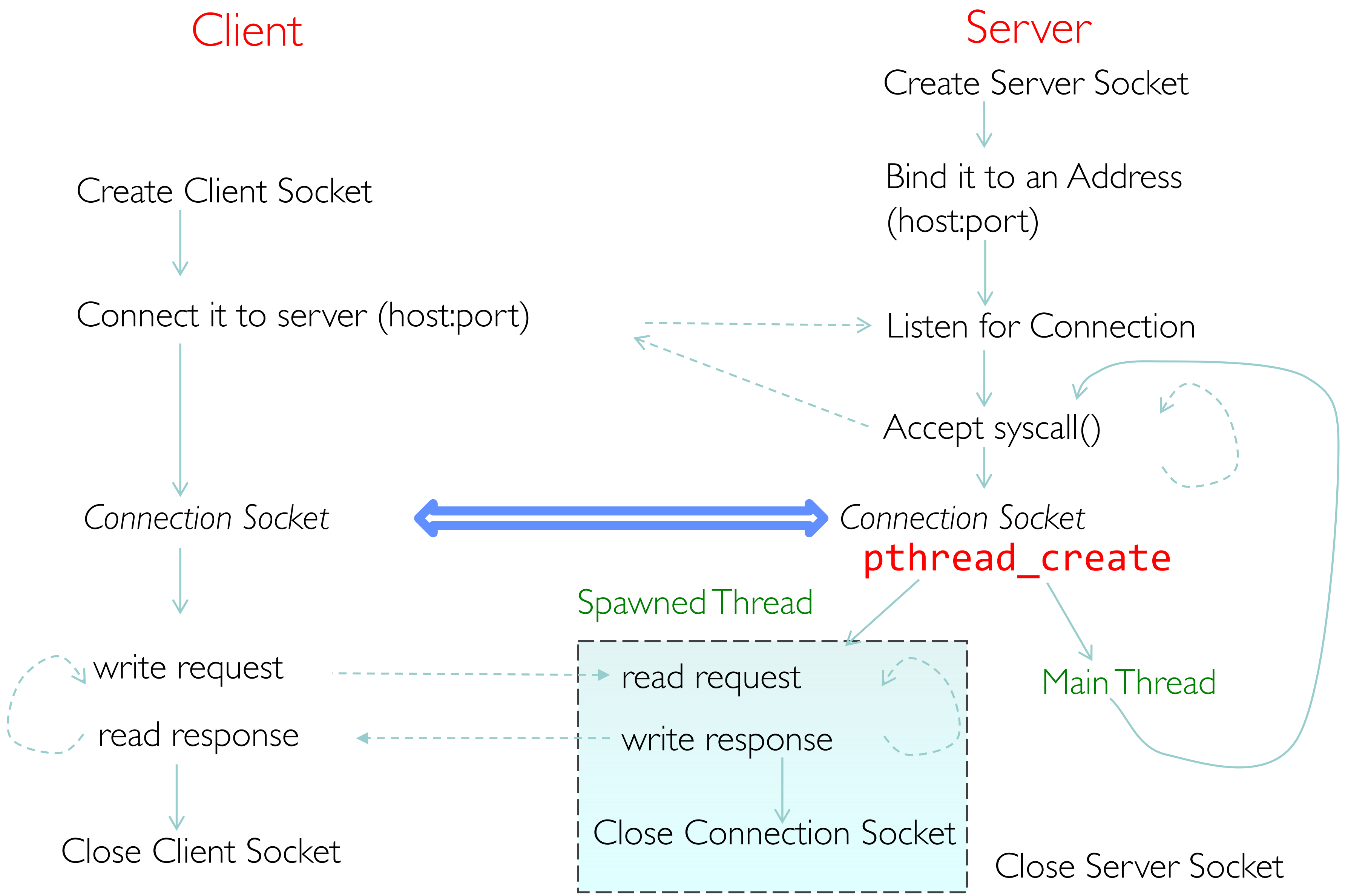
Recall: Sockets

Sockets with Protection and Concurrency



Recall: Sockets

- **Sockets with Concurrency, without Protection**



Overview

Motivation for Threads

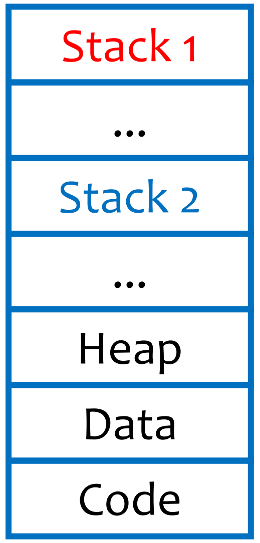
- Most modern applications are multithreaded
  - Threads run within application
- Multiple tasks with the application can be implemented by several threads, e.g., a **word processor**, such as Microsoft Word.app:
  - Update display
  - Responding to keystrokes from the user
  - Spell checking
  - ...
- Process creation is heavy-weight.
- Thread creation is light-weight.
- Can simplify code, improve efficiency
- Kernels are generally multithreaded

Overview

- **Motivation for Threads**
  - Operating Systems must handle **m**ultiple **t**hings **a**t **o**nce (MTAO)
    - Processes, interrupts, background system maintenance
  - Networked servers must handle MTAO
    - Multiple connections handled simultaneously
  - Parallel programs must handle MTAO
    - To achieve better performance
  - Programs with user interface often must handle MTAO
    - To achieve user responsiveness while doing computation
  - Network and disk bound programs must handle MTAO
    - To hide network/disk latency
    - Sequence steps in access or communication

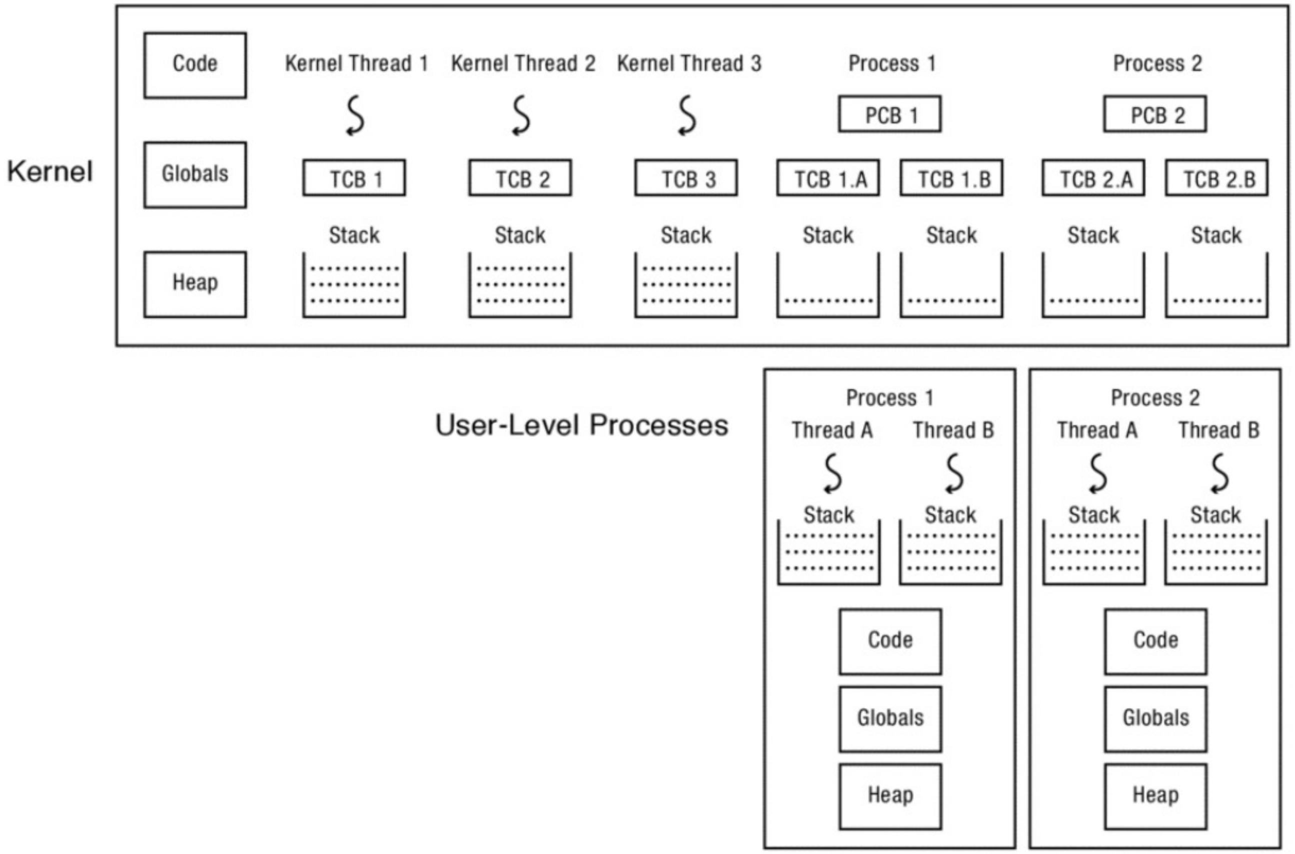
Overview

- **Memory Layout with Two Threads**
  - Within the same Address Space of a Process
  - Two sets of CPU registers
  - Two sets of Stacks



Overview

- **Memory Layout with Two Threads**
  - Within the same Address Space of a Process
  - Two sets of CPU registers
  - Two sets of Stacks



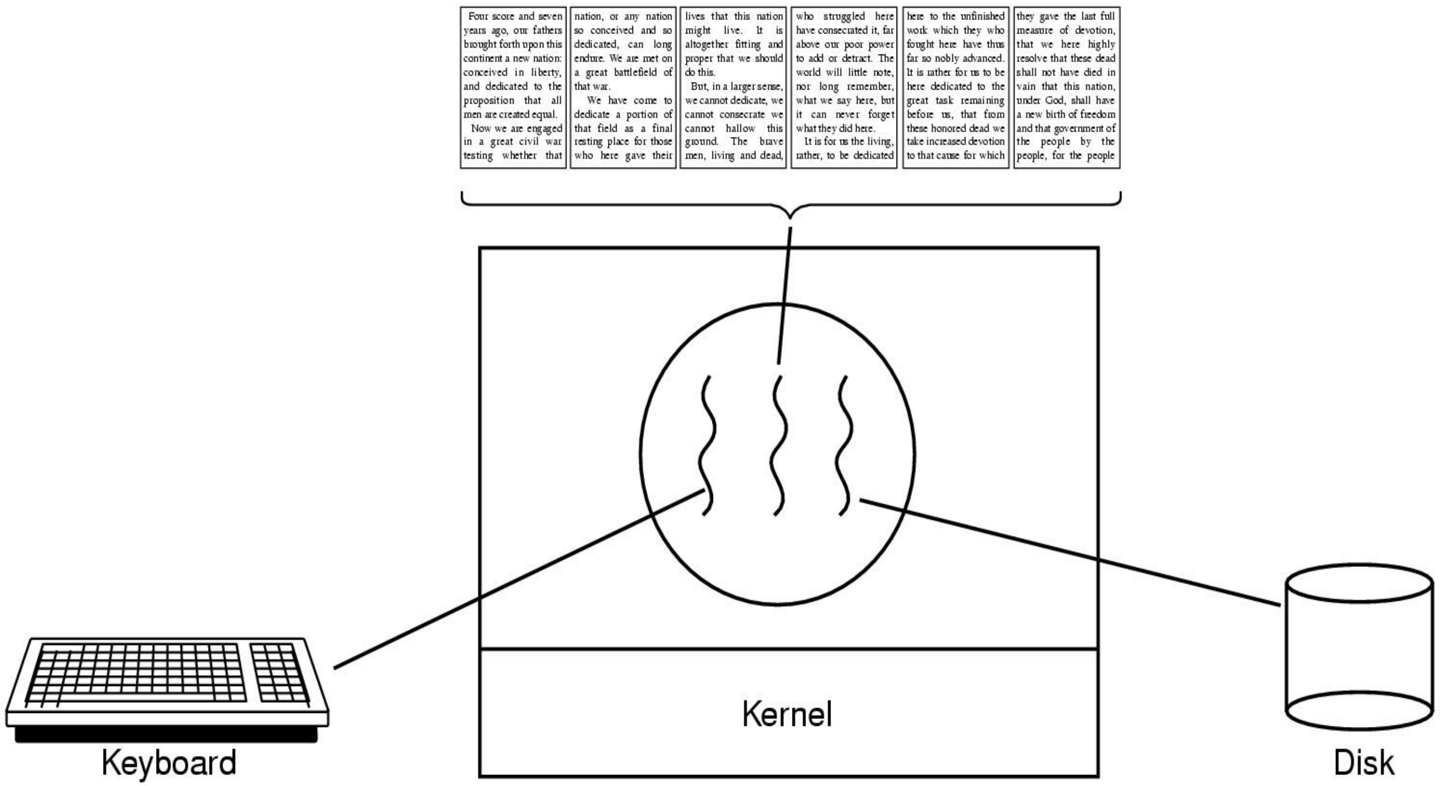
Overview

- **Benefits of Threads**
  - **Responsiveness (响应能力)**
    - may allow continued execution if part of the process is blocked, especially important for user interfaces
  - **Resource Sharing (资源共享)**
    - threads share resources of process, easier than IPC such as shared memory or message passing
    - Inter-thread communication and synchronization is faster than IPC
  - **Economy (经济性/低系统开销)**
    - cheaper than process creation (no need to create new address space)
    - thread switching lower overhead than process context switching
  - **Scalability (可扩展性/可伸缩性)**
    - process can take advantage of multicore architectures, i.e., multiple threads running in parallel on different cores.

Overview

Benefits of Threads

- Example 1: **Word Processor**
  - One thread displays UI and texts
  - A second thread reads user inputs from keyboard (*usually blocks*)
  - A third thread perform disk I/O (*usually blocks*)


Four score and seven years ago, our fathers brought forth upon this continent a new nation: conceived in liberty, and dedicated to the proposition that all men are created equal.

Now we are engaged in a great civil war testing whether that nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battlefield of that war.

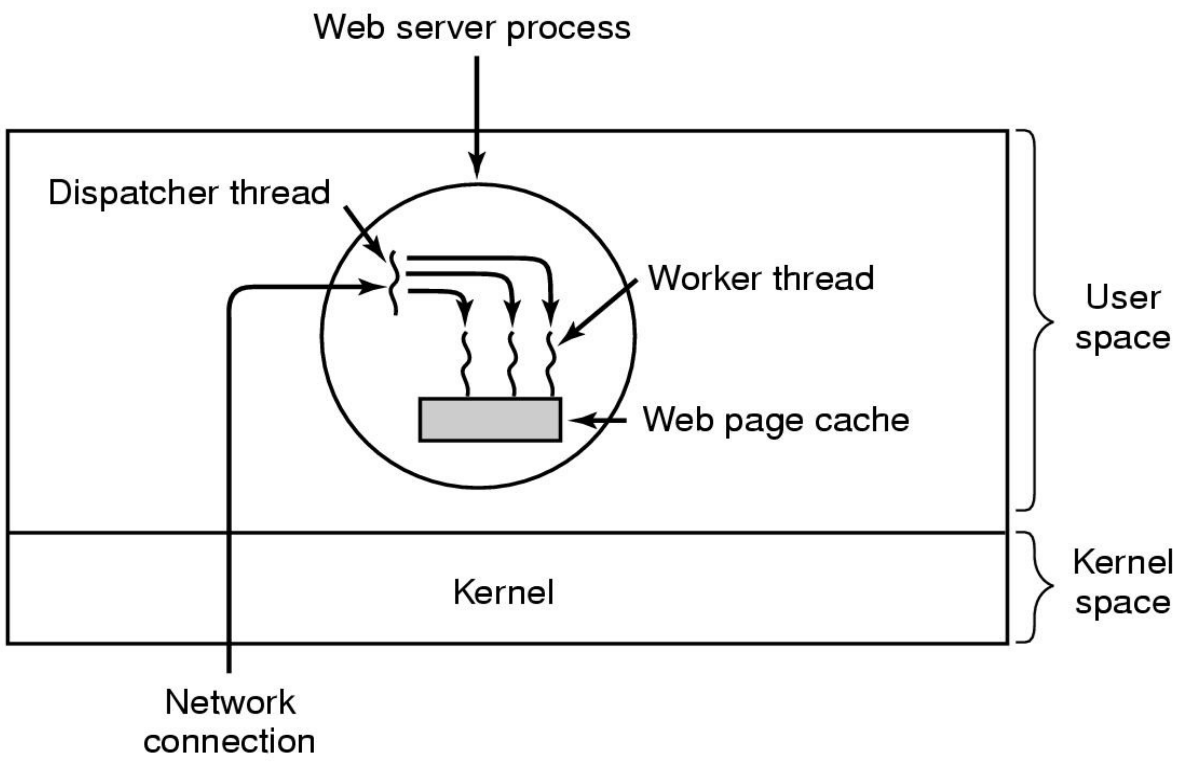
We have come to dedicate a portion of that field as a final resting place for those who here gave their lives that this nation might live. It is altogether fitting and proper that we should do this.

But, in a larger sense, we cannot dedicate, we cannot consecrate we cannot hallow this ground. The brave men, living and dead, who struggled here have consecrated it, far above our poor power to add or detract. The world will little note, nor long remember, what we say here, but it can never forget what they did here.

It is for us the living, rather, to be dedicated here to the unfinished work which they who fought here have thus far so nobly advanced. It is rather for us to be here dedicated to the great task remaining before us, that from these honored dead we take increased devotion to that cause for which they gave the last full measure of devotion, that we here highly resolve that these dead shall not have died in vain that this nation, under God, shall have a new birth of freedom and that government of the people by the people, for the people

Overview

- **Benefits of Threads**
  - Example 2: **A File/Web server**
    - The server needs to handle requests over a short period. Hence it is more efficient to create (and destroy) a single thread for each request.
    - On a SMP machine, multiple threads can possibly be executed simultaneously on multiple CPU cores.



Overview

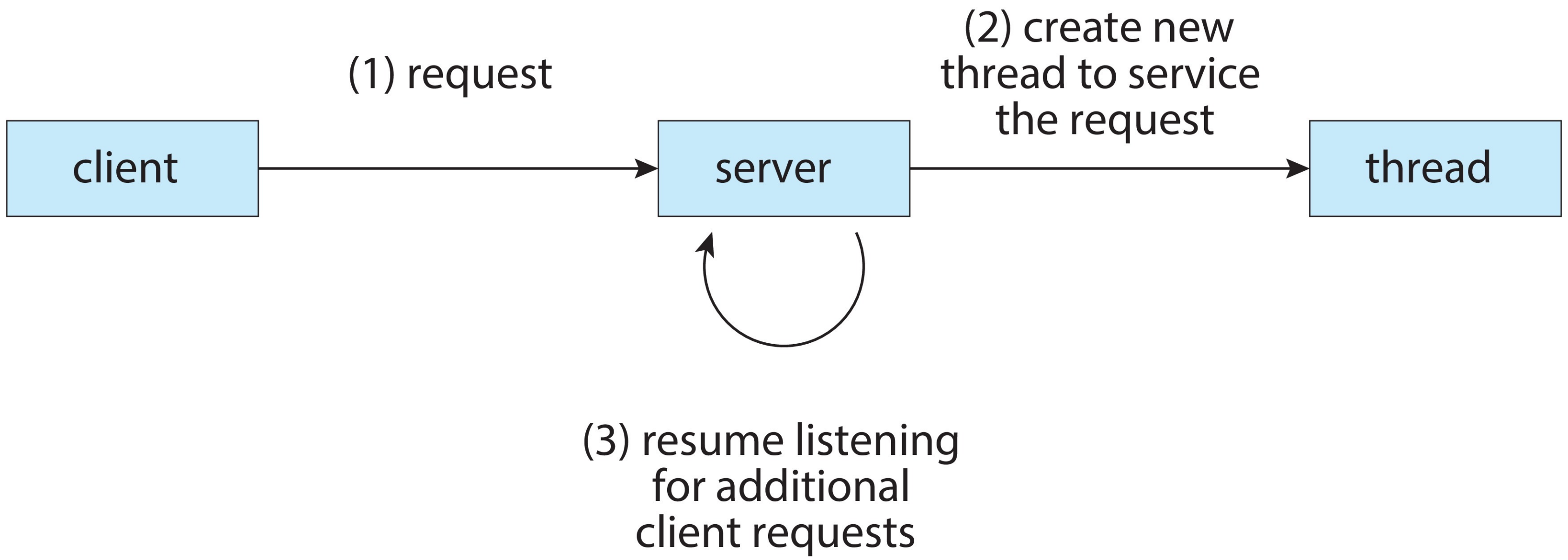
- **Benefits of Threads**
  - Example 2: **A File/Web server**
    - **Dispatcher** thread (main thread):

```
while (true) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

    - **Worker** thread

```
while (true) {
    wait_for_work(&buf);
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

Multithreaded Server Architecture

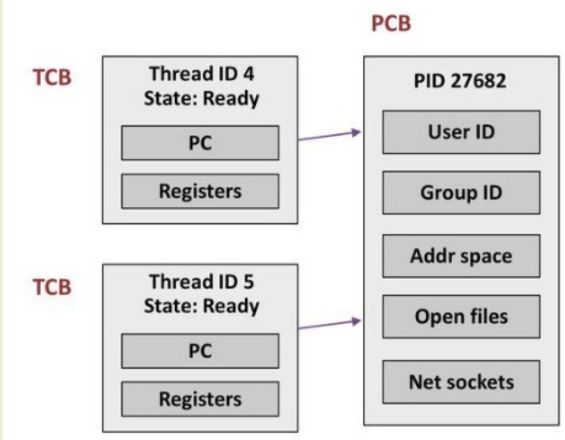

client
(1) request
server
(2) create new thread to service the request
thread
(3) resume listening for additional client requests

Overview

Benefits of Threads

- Two important implications
  - Threaded applications often run faster than non-threaded applications
    - context switching between kernel and user-space are avoided

```
// Simplified pseudocode for process context switching
void context_switch_process(Process *current, Process *next) {
    // Save current process state to its PCB
    save_state_to_pcb(&current->pcb);
    // Switch memory address space to the next process
    switch_memory_address_space(next->memory_address_space);
    // Restore the state of the next process from its PCB
    restore_state_from_pcb(&next->pcb);
    // Update system scheduler and process pointers
    current_process = next;
}
// Simplified pseudocode for thread context switching within the same process
void context_switch_thread(Thread *current, Thread *next) {
    // Save current thread state to its TCB
    save_state_to_tcb(&current->tcb);
    // Restore the state of the next thread from its TCB
    restore_state_from_tcb(&next->tcb);
    // Update system scheduler and thread pointers
    current_thread = next;
}
```



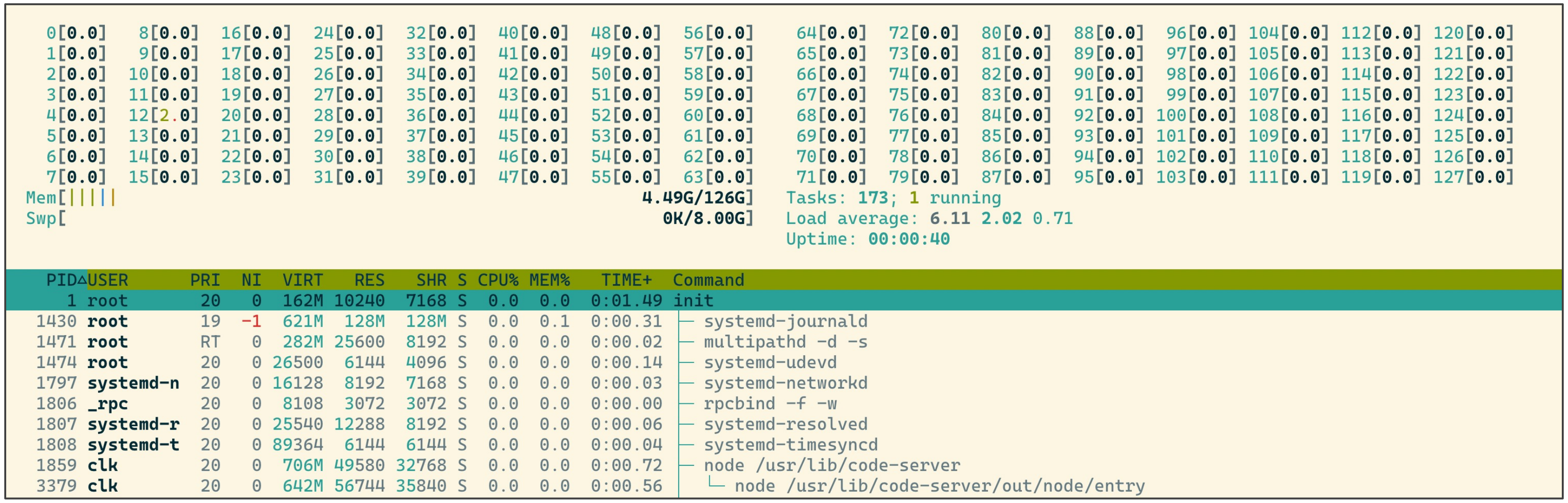
Overview

- **Benefits of Threads**
  - Two important implications
    - Threaded applications often run faster than non-threaded applications
      - context switching between kernel and user-space are avoided
    - Threaded applications are harder to develop
      - although simple, clean designs can help
  - Additionally, the assumption is that the development environment provides a Threads Library for developers to use.
    - Most modern OSes do

Multicore Programming

Multicore Systems

- Multicore architectures place multiple cores on a single chip.
  - Each core appears as a separate processor to the OS
  - Whether the cores appear across CPU chips or within CPU chips, we call these systems multicore or multiprocessor systems.
  - Almost all systems are multicore nowadays.
    - E.g., your phone has **>6 cores**; our code-server has **128 cores**!

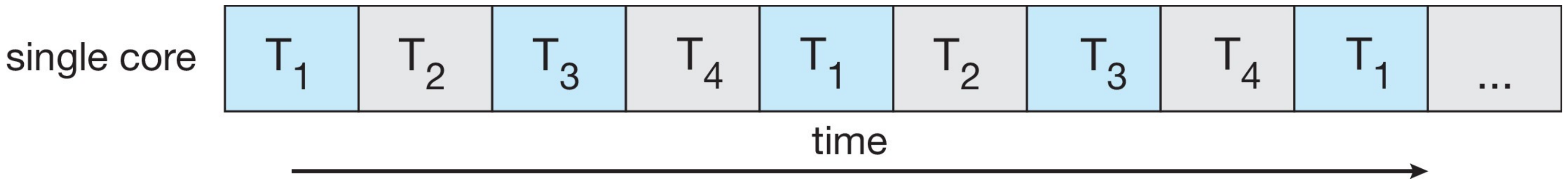


```
   0[0.0]   8[0.0]  16[0.0]  24[0.0]  32[0.0]  40[0.0]  48[0.0]  56[0.0]    64[0.0]  72[0.0]  80[0.0]  88[0.0]  96[0.0] 104[0.0] 112[0.0] 120[0.0]
   1[0.0]   9[0.0]  17[0.0]  25[0.0]  33[0.0]  41[0.0]  49[0.0]  57[0.0]    65[0.0]  73[0.0]  81[0.0]  89[0.0]  97[0.0] 105[0.0] 113[0.0] 121[0.0]
   2[0.0]  10[0.0]  18[0.0]  26[0.0]  34[0.0]  42[0.0]  50[0.0]  58[0.0]    66[0.0]  74[0.0]  82[0.0]  90[0.0]  98[0.0] 106[0.0] 114[0.0] 122[0.0]
   3[0.0]  11[0.0]  19[0.0]  27[0.0]  35[0.0]  43[0.0]  51[0.0]  59[0.0]    67[0.0]  75[0.0]  83[0.0]  91[0.0]  99[0.0] 107[0.0] 115[0.0] 123[0.0]
   4[0.0]  12[2.0]  20[0.0]  28[0.0]  36[0.0]  44[0.0]  52[0.0]  60[0.0]    68[0.0]  76[0.0]  84[0.0]  92[0.0] 100[0.0] 108[0.0] 116[0.0] 124[0.0]
   5[0.0]  13[0.0]  21[0.0]  29[0.0]  37[0.0]  45[0.0]  53[0.0]  61[0.0]    69[0.0]  77[0.0]  85[0.0]  93[0.0] 101[0.0] 109[0.0] 117[0.0] 125[0.0]
   6[0.0]  14[0.0]  22[0.0]  30[0.0]  38[0.0]  46[0.0]  54[0.0]  62[0.0]    70[0.0]  78[0.0]  86[0.0]  94[0.0] 102[0.0] 110[0.0] 118[0.0] 126[0.0]
   7[0.0]  15[0.0]  23[0.0]  31[0.0]  39[0.0]  47[0.0]  55[0.0]  63[0.0]    71[0.0]  79[0.0]  87[0.0]  95[0.0] 103[0.0] 111[0.0] 119[0.0] 127[0.0]
 Mem[|||||                                                    4.49G/126G]   Tasks: 173; 1 running
 Swp[                                                           0K/8.00G]   Load average: 6.11 2.02 0.71
                                                                            Uptime: 00:00:40

   PID△USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
     1 root       20   0  162M 10240  7168 S  0.0  0.0  0:01.49 init
  1430 root       19  -1  621M  128M  128M S  0.0  0.1  0:00.31 ├─ systemd-journald
  1471 root       RT   0  282M 25600  8192 S  0.0  0.0  0:00.02 ├─ multipathd -d -s
  1474 root       20   0 26500  6144  4096 S  0.0  0.0  0:00.14 ├─ systemd-udevd
  1797 systemd-n  20   0 16128  8192  7168 S  0.0  0.0  0:00.03 ├─ systemd-networkd
  1806 _rpc       20   0  8108  3072  3072 S  0.0  0.0  0:00.00 ├─ rpcbind -f -w
  1807 systemd-r  20   0 25540 12288  8192 S  0.0  0.0  0:00.06 ├─ systemd-resolved
  1808 systemd-t  20   0 89364  6144  6144 S  0.0  0.0  0:00.04 ├─ systemd-timesyncd
  1859 clk        20   0  706M 49580 32768 S  0.0  0.0  0:00.72 ├─ node /usr/lib/code-server
  3379 clk        20   0  642M 56744 35840 S  0.0  0.0  0:00.56 │  └─ node /usr/lib/code-server/out/node/entry
```

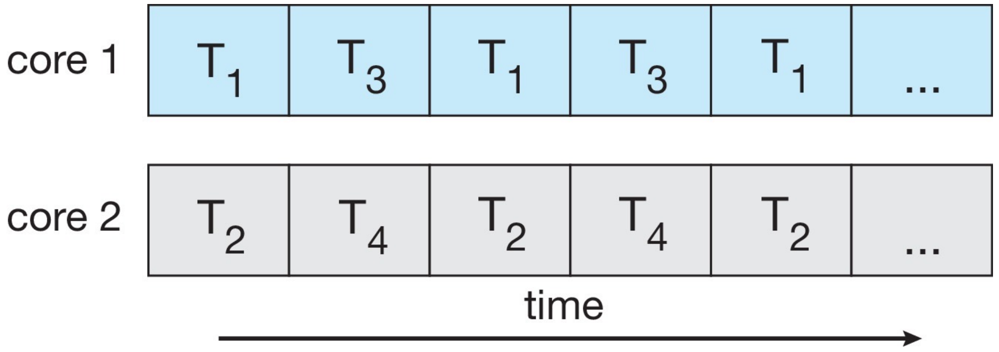
- Multithreaded programming provides a mechanism for more **efficient use** of these multiple computing cores and improved **concurrency**.

Multicore Programming

- **Multicore Systems**
  - **Concurrent** execution on single-core system:
    - execution of threads are **interleaved** over time



  - **Parallelism** on a multi-core system:
    - some threads can run **in parallel**.



Multicore Programming

- **Concurrency is not Parallelism**
  - Concurrency is about handling multiple things at once (**MTAO**)
  - Parallelism is about doing multiple things **simultaneously**

  - It is possible to have concurrency without parallelism.

    Example: Two threads on a single-core system
    - ... execute concurrently ...
    - ... but not in parallel

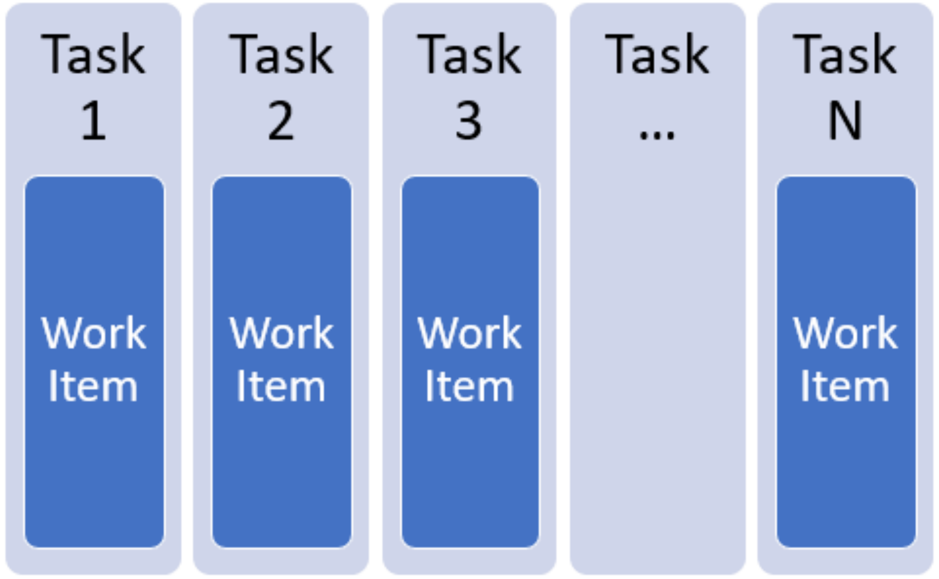
  - Each thread handles or manages a separate thing or task
  - But those tasks are not necessarily executing simultaneously

Multicore Programming

- Challenges in programming for Multicore systems:
  - Identifying Tasks
  - Balance
  - Data Splitting
  - Data Dependency
  - Testing and debugging

Multicore Programming

- **Challenges in programming for Multicore systems:**
  - **Identifying Tasks**
    - This involves examining applications to find areas that can be divided into separate, concurrent tasks. Ideally, tasks are independent of one another and thus can run in parallel on individual cores.
  - Balance
  - Data Splitting
  - Data Dependency
  - Testing and debugging

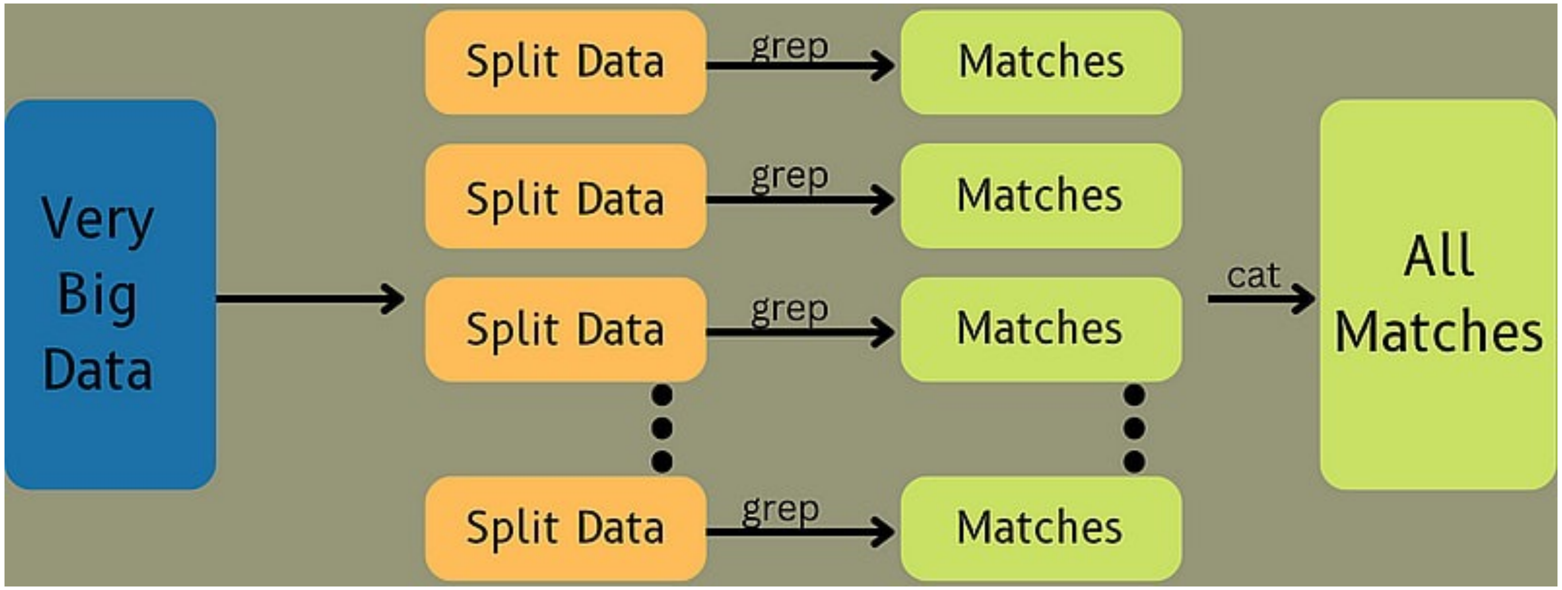


Multicore Programming

- **Challenges in programming for Multicore systems:**
  - Identifying Tasks
  - **Balance**
    - While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value. In some instances, a certain task may not contribute as much value to the overall process as other tasks. Using a separate execution core to run that task may not be worth the cost.
  - Data Splitting
  - Data Dependency
  - Testing and debugging

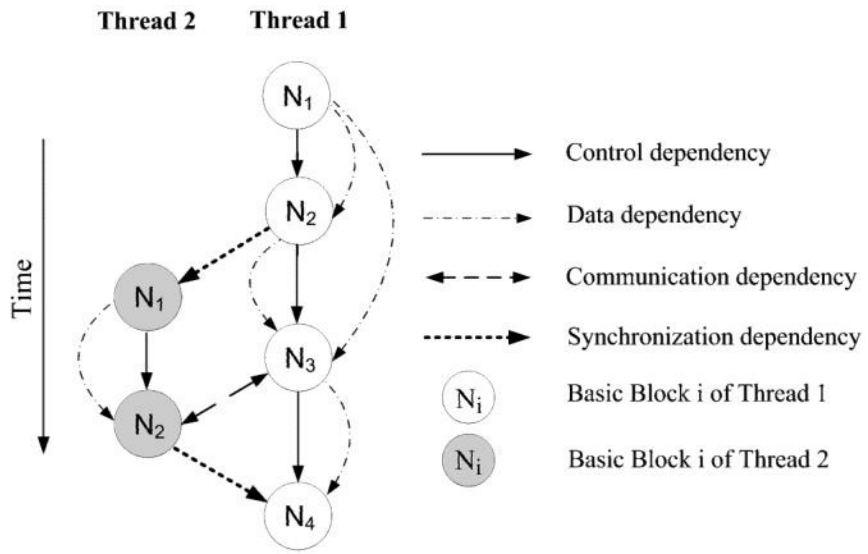
Multicore Programming

- **Challenges in programming for Multicore systems:**
  - Identifying Tasks
  - Balance
  - **Data Splitting**
    - Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.
  - Data Dependency
  - Testing and debugging

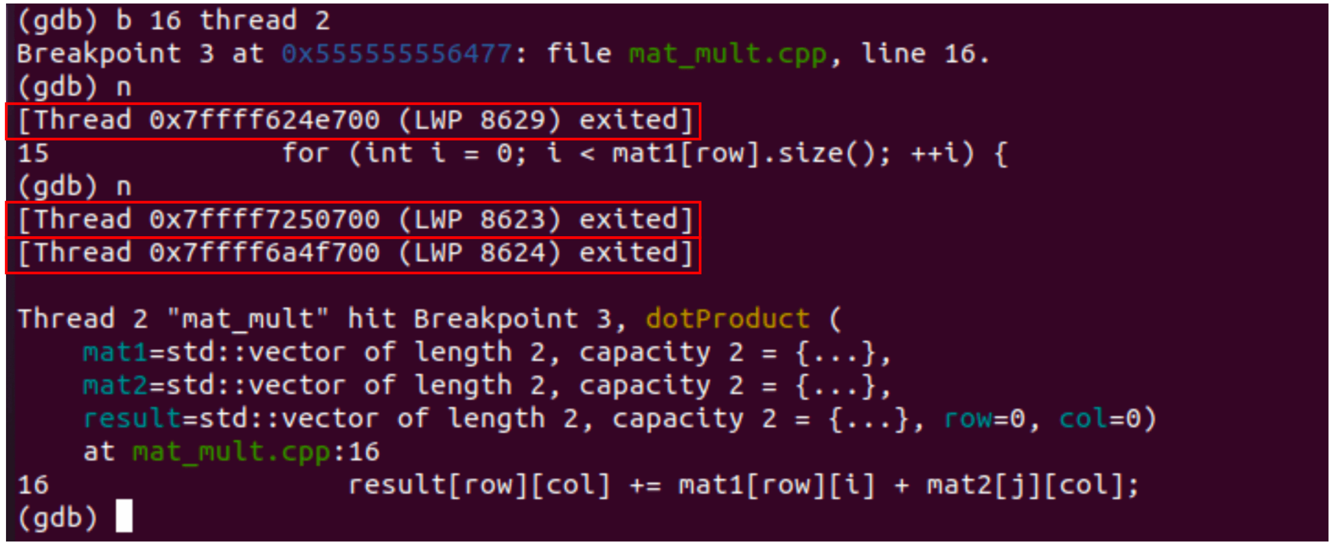


Multicore Programming

- **Challenges in programming for Multicore systems:**
  - Identifying Tasks
  - Balance
  - Data Splitting
  - **Data Dependency**
    - The data accessed by the tasks must be examined for dependencies between two or more tasks. When one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency.
  - Testing and debugging



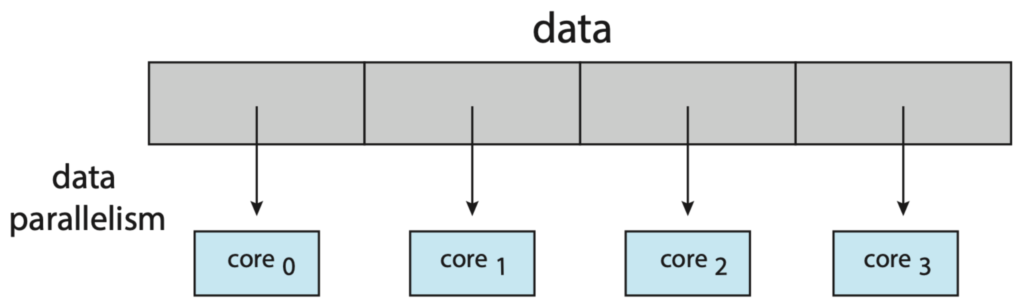
- **Challenges in programming for Multicore systems:**
  - Identifying Tasks
  - Balance
  - Data Splitting
  - Data Dependency
  - **Testing and debugging**
    - When running in parallel on multiple cores, many different execution paths are possible. Testing and debugging such concurrent programs is inherently more difficult than on single-threaded applications.



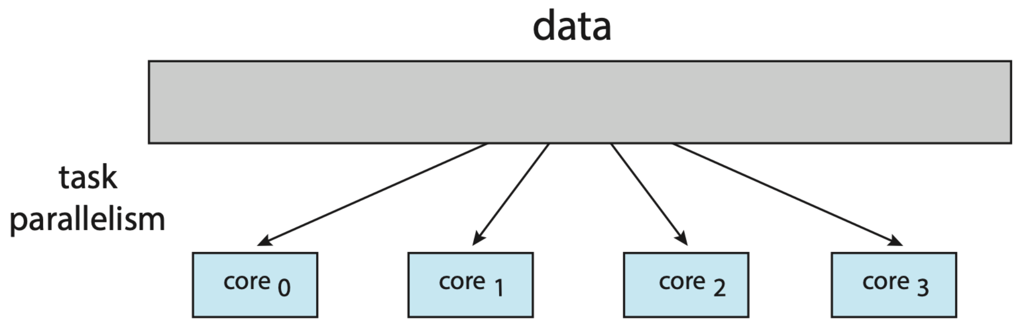
```
(gdb) b 16 thread 2
Breakpoint 3 at 0x555555556477: file mat_mult.cpp, line 16.
(gdb) n
[Thread 0x7ffff624e700 (LWP 8629) exited]
15              for (int i = 0; i < mat1[row].size(); ++i) {
(gdb) n
[Thread 0x7ffff7250700 (LWP 8623) exited]
[Thread 0x7ffff6a4f700 (LWP 8624) exited]

Thread 2 "mat_mult" hit Breakpoint 3, dotProduct (
    mat1=std::vector of length 2, capacity 2 = {...},
    mat2=std::vector of length 2, capacity 2 = {...},
    result=std::vector of length 2, capacity 2 = {...}, row=0, col=0)
    at mat_mult.cpp:16
16                  result[row][col] += mat1[row][i] + mat2[j][col];
(gdb) 
```

Multicore Programming

- **Types of Parallelism**
  - **Data** Parallelism
    - focuses on distributing subsets of the **same** data across multiple computing cores and performing the **same** operation on each core



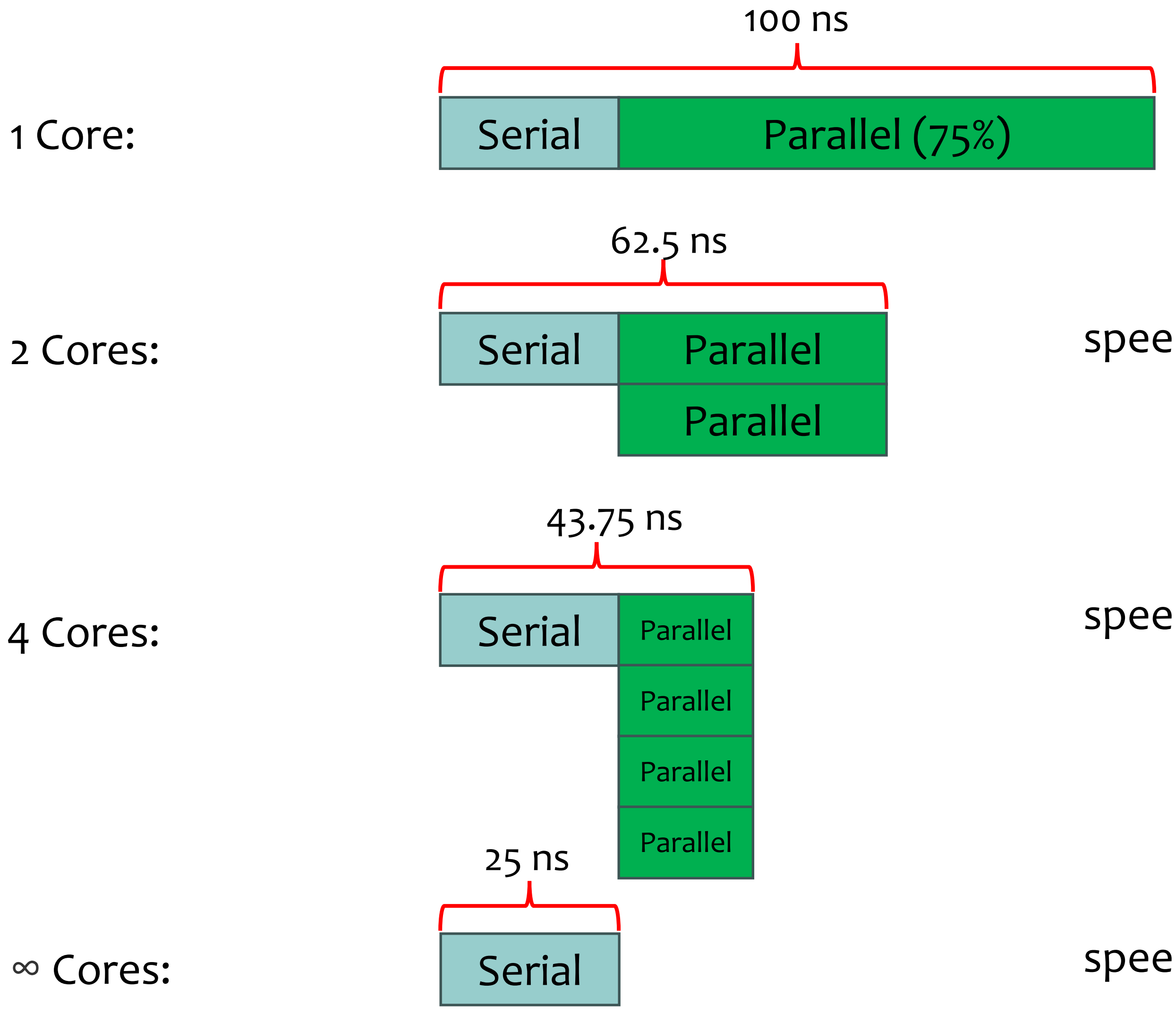
  - **Task** Parallelism
    - involves distributing not data but **tasks** (threads) across multiple computing cores. Each thread is performing a unique operation.
    - Different threads may be operating on the same data, or different data



Multicore Programming

- **Amdahl's Law**
  - Identifies performance gains from adding additional cores to an application that has both serial (non-parallel) and parallel components
  - **S** is the serial portion; **N** processing cores
  - $speedup \leq \frac{1}{S+\frac{(1-S)}{N}}$
  - For example, if an application is 75% parallel / 25% serial, moving from 1 to 2 cores results in a speedup of $\frac{1}{0.25+\frac{0.75}{2}} = 1.6$
  - As N approaches infinity, speedup approaches $\frac{1}{S}$
    - For example, if 50% is serial, then the maximum speedup is 2.0 times.
  - Serial portion of an application has disproportionate effect on performance gained by adding additional cores.

Amdahl's Law



$$\text{speedup} = \frac{1}{0.25+\frac{0.75}{2}} = 1.6$$

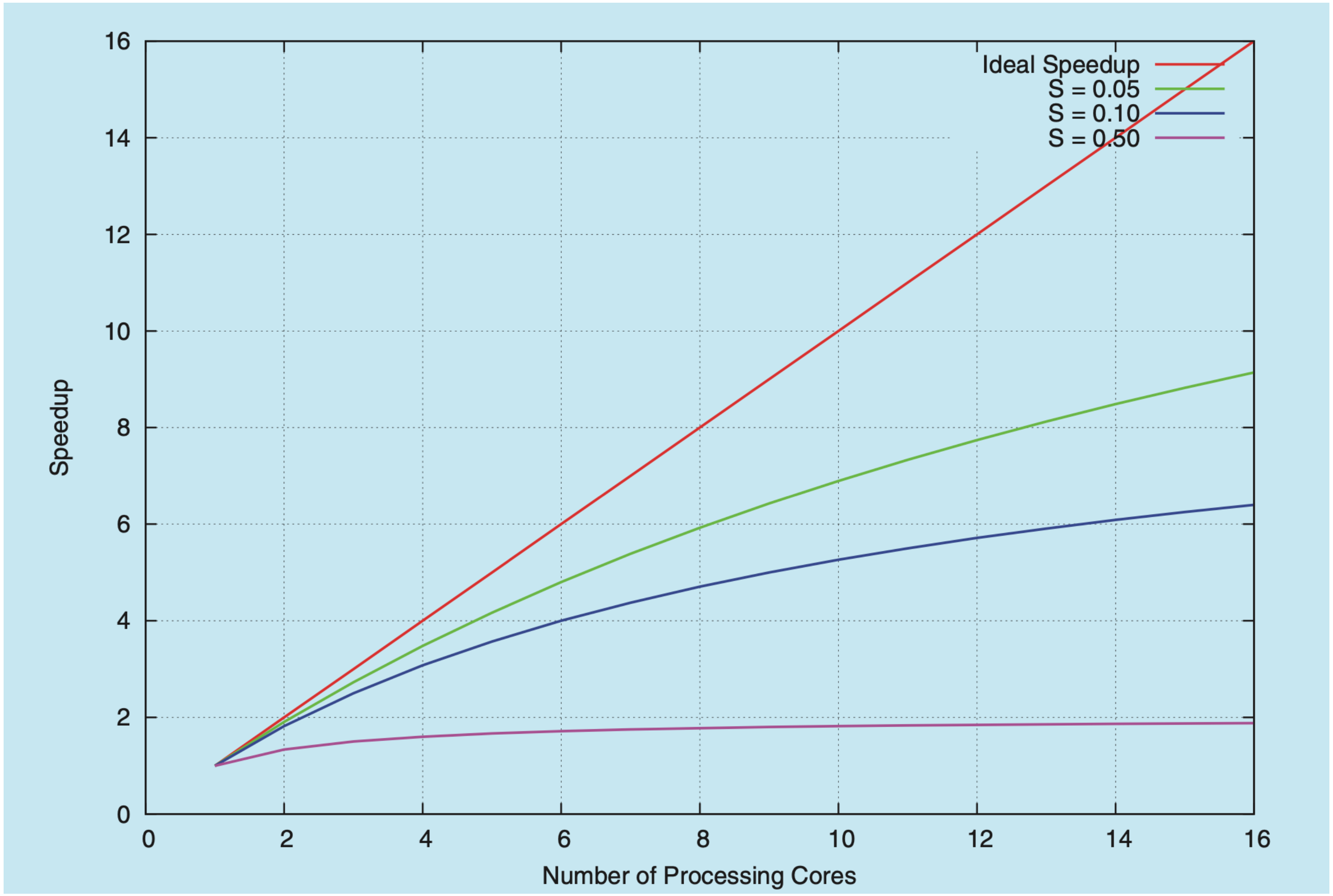
$$\text{speedup} = \frac{1}{0.25+\frac{0.75}{4}} = 2.286$$

$$\text{speedup} = \frac{1}{0.25+\frac{0.75}{\infty}} = 4$$

Multicore Programming

Amdahl's Law

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

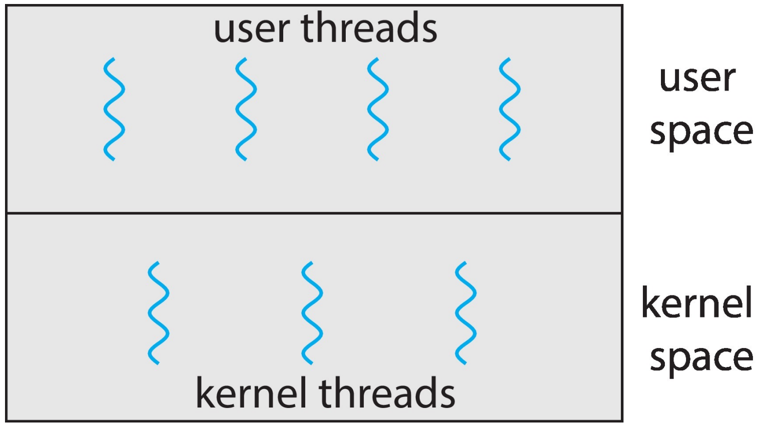


User Threads and Kernel Threads

- **User Threads**
  - Management done by user-level threads library
  - The kernel is unaware of the existence of User Level Threads
    - When one thread makes a **blocking** syscall, the whole process will **block**
  - Thread switching does not require kernel mode privileges
  - Scheduling is application specific
- **Kernel Threads**
  - Supported and managed directly by the kernel
  - Kernel maintains context information for the process and threads
  - Switching between threads requires kernel intervention

Multithreading Models

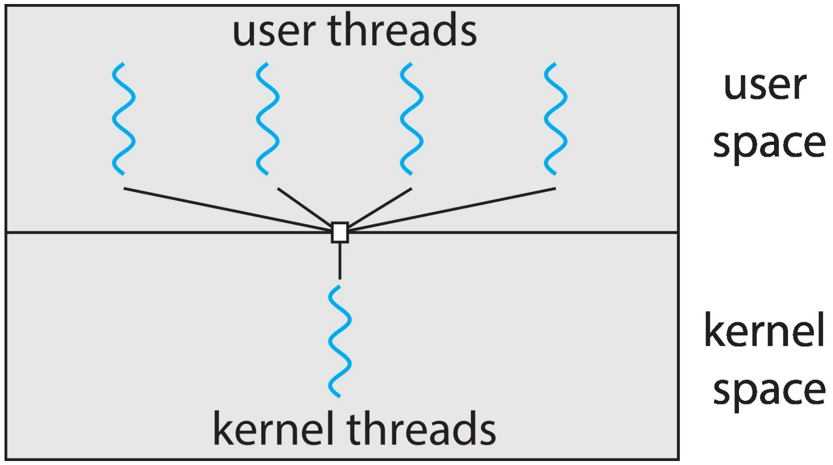
- **Multithreading Models**
  - Support for threads may be provided either at the user level, for user level threads (ULTs), or by the kernel, for kernel level threads (KLT).
    - **User threads** are supported above the kernel and are managed without kernel support
    - **Kernel threads** are supported and managed directly by the OS



  - Relationship between user threads and kernel threads:
    - **Many-to-One** model
    - **One-to-One** model *(most common)*
    - **Many-to-Many** model

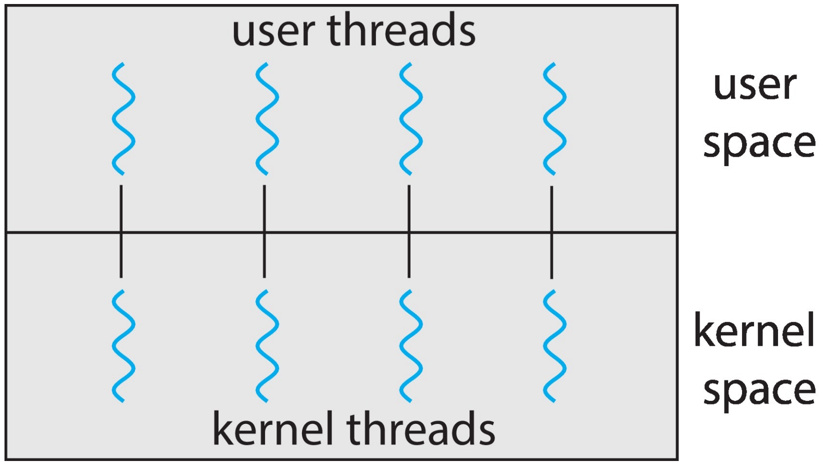
Multithreading Models

- **Many-to-One Model**
  - Many user-level thrads mapped to a single kernel-level thread
    - The kernel is unaware of user-level threads
  - One thread blocking causes all threads to block
  - Multiple threads may not run in parallel on multicore system because only one may be in the kernel at a time
  - *Few systems currently use this model*
  - Examples:
    - Solaris Green Threads
    - GNU Portable Threads



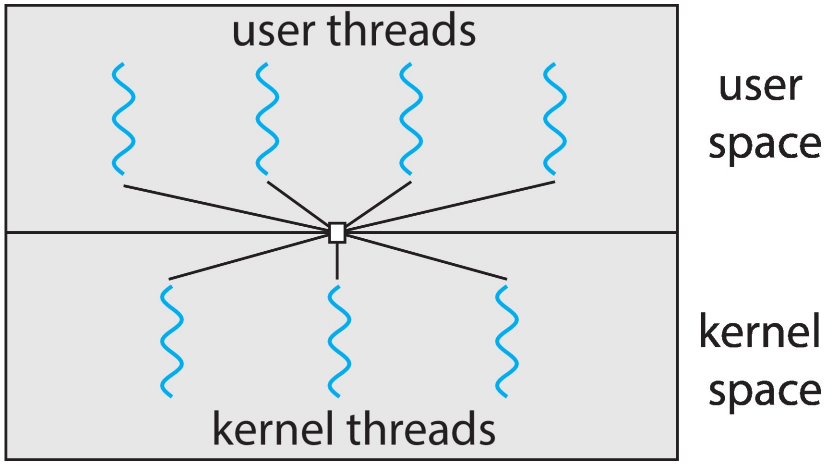
Multithreading Models

- **One-to-One Model**
  - 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 often restricted due to overhead
  - Examples
    - Linux
    - Windows



Multithreading Models

- **Many-to-Many Model**
  - Allows many user level threads to be mapped to many kernel threads
  - Allows the OS to create a sufficient number of kernel threads
  - More flexible
  - Much more difficult to implement than other models
  - Examples
    - Windows with the *ThreadFiber* package
    - Otherwise not very common



Thread Libraries

- **Thread Libraries**
  - Thread libraries provide programmers with API for creating and managing threads
  - Two primary ways of implementing threads:
    - Library entirely in user space
      - no system calls
    - Kernel-level library supported by the OS
      - code and data structures for the library exist in kernel space
      - Invoking an API function results in a system call to the kernel

Thread Libraries

- **Thread Libraries**
  - Thread libraries provide programmers with API for creating and managing threads
  - Two primary ways of implementing threads:
    - Library entirely in user space
      - no system calls
    - Kernel-level library supported by the OS
      - code and data structures for the library exist in kernel space
      - Invoking an API function results in a system call to the kernel
  - Three main thread libraries are in use today:
    - **POSIX Pthreads**
    - Windows Thread Library
    - Java Thread API

Pthreads

- **Pthreads**
  - A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
  - May be provided either as user-level or kernel-level
  - **Specification**, not **implementation**
  - API specifies behavior of thread library, implementation is up to development of the library
    - Linux uses Native POSIX Thread Library (NPTL) as their pthread impl
    - FreeBSD primarily uses `` `libthr` `` as its pthread library impl
  - Common in UNIX operating system (Linux, macOS)

Pthreads

Pthreads API

- Thread Creation:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);

/* Returns 0 on success, or a positive error number on error */
```

- Thread Completion:

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

/* Returns 0 on success, or a positive error number on error */
```

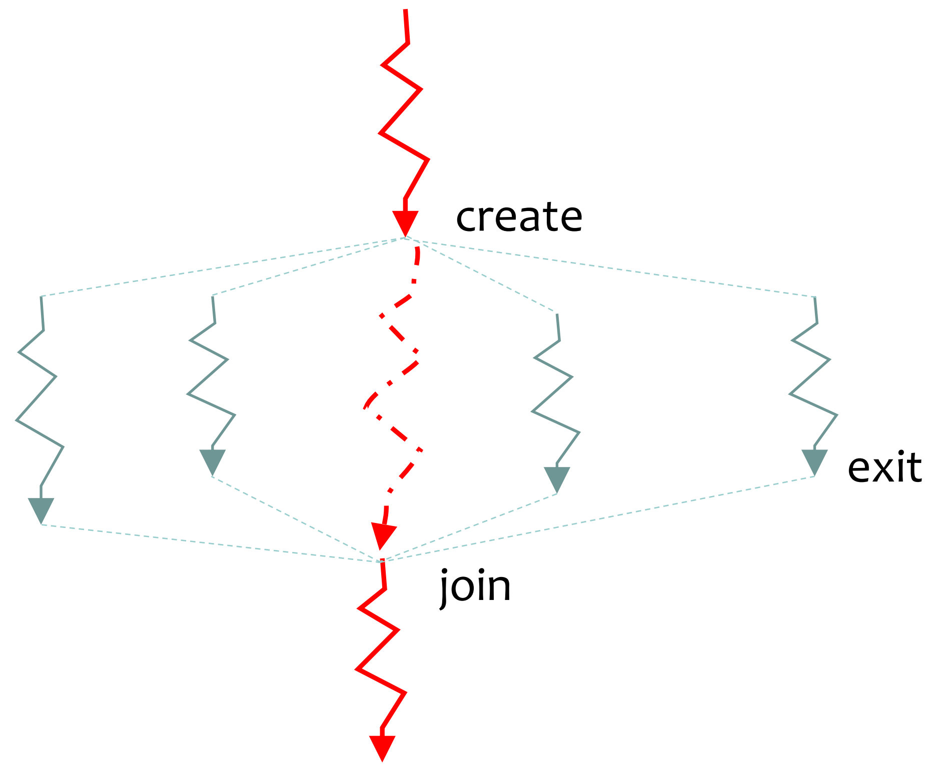
- Others:

```
void pthread_exit(void *retval);      /* Terminates calling thread */

int pthread_kill(pthread_t thread, int sig); /* Send sig to thread */

pthread_t pthread_self(void);         /* Return current thread ID */

int pthread_detach(pthread_t thread); /* Detach thread, no need to join */
```

Pthreads

Pthreads: Fork-Join Pattern

- Main thread creates (forks) multiple threads by passing them `args` to work on
- ... and then `joins` with them, collecting results.



Pthreads Example

```c
/* t0.c */
#include <pthread.h>
#include <stdio.h>

void *threadfun(void *arg) {
    printf("Thread started working...\n");
    long sum = 0;
    long n = (long)arg;
    for (int i = 1; i <= n; i++) {
        sum = sum + i;
    }
    printf("sum: %ld\n", sum);
    printf("Thread finished its work.\n");
    return NULL;
}
int main(int argc, char **argv) {
    pthread_t tid;

    printf("main(): creating new thread...\n");
    pthread_create(&tid, NULL,
                   threadfun, (void *)10);
    pthread_join(tid, NULL);
    printf("main(): Thread returned.\n");

    return 0;
}
```

```
$ ./t0
main(): creating new thread...
Thread started working...
sum: 55
Thread finished its work.
main(): Thread returned.
```

Pthreads

Pthreads Example

```
/* t0.c */
#include <pthread.h>
#include <stdio.h>

void *threadfun(void *arg) {
    printf("Thread started working...\n");
    long sum = 0;
    long n = (long)arg;
    for (int i = 1; i <= n; i++) {
        sum = sum + i;
    }
    printf("sum: %ld\n", sum);
    printf("Thread finished its work.\n");
    return NULL;
}
int main(int argc, char **argv) {
    pthread_t tid;

    printf("main(): creating new thread...\n");
    pthread_create(&tid, NULL,
                   threadfun, (void *)10);
    pthread_join(tid, NULL);
    printf("main(): Thread returned.\n");

    return 0;
}
```

```
$ ./t0
main(): creating new thread...
Thread started working...
sum: 55
Thread finished its work.
main(): Thread returned.
```

Why is void pointer used as argument?

Answer: It allows us to pass in ***any*** type of argument. We have to explicitly cast the `(void *)arg` into the desired type. (in the case of `t0.c`, the `long` type).

Pthreads Example

```
/* t0.c */
#include <pthread.h>
#include <stdio.h>

void *threadfun(void *arg) {
    printf("Thread started working...\n");
    long sum = 0;
    struct data *obj = (struct data *)arg;
    for (int i = 1; i <= n; i++) {
        sum = sum + i;
    }
    printf("sum: %ld\n", sum);
    printf("Thread finished its work.\n");
    return NULL;
}
int main(int argc, char **argv) {
    pthread_t tid;
    struct data obj;
    printf("main(): creating new thread...\n");
    pthread_create(&tid, NULL,
                   threadfun, (void *)&obj);
    pthread_join(tid, NULL);
    printf("main(): Thread returned.\n");

    return 0;
}
```

```
$ ./t0
main(): creating new thread...
Thread started working...
sum: 55
Thread finished its work.
main(): Thread returned.
```

Why is `void` pointer used as argument?

Answer: It allows us to pass in ***any*** type of argument. We have to explicitly cast the `(void *)arg` into the desired type. (in the case of `t0.c`, the `long` type).

When passing a compound data type, such as `struct`, the pointer to `struct` must be used.

Pthreads Example

```c
/* t0.c */
#include <pthread.h>
#include <stdio.h>

void *threadfun(void *arg) {
    printf("Thread started working...\n");
    long sum = 0;
    long n = (long)arg;
    for (int i = 1; i <= n; i++) {
        sum = sum + i;
    }
    printf("sum: %ld\n", sum);
    printf("Thread finished its work.\n");
    return NULL;
}
int main(int argc, char **argv) {
    pthread_t tid;

    printf("main(): creating new thread...\n");
    pthread_create(&tid, NULL,
                   threadfun, (void *)10);
    pthread_join(tid, NULL);
    printf("main(): Thread returned.\n");

    return 0;
}
```

```
$ ./t0
main(): creating new thread...
Thread started working...
sum: 55
Thread finished its work.
main(): Thread returned.
```

Recall that threads in the same process share many resources, most notably the global variables.

Pthreads Example

```c
/* t0.c */
#include <pthread.h>
#include <stdio.h>

void *threadfun(void *arg) {
    printf("Thread started working...\n");
    long sum = 0;
    long n = (long)arg;
    for (int i = 1; i <= n; i++) {
        sum = sum + i;
    }
    printf("sum: %ld\n", sum);
    printf("Thread finished its work.\n");
    return NULL;
}
int main(int argc, char **argv) {
    pthread_t tid;

    printf("main(): creating new thread...\n");
    pthread_create(&tid, NULL,
                   threadfun, (void *)10);
    pthread_join(tid, NULL);
    printf("main(): Thread returned.\n");

    return 0;
}
```

```
$ ./t0
main(): creating new thread...
Thread started working...
sum: 55
Thread finished its work.
main(): Thread returned.
```

Recall that threads in the same process share many resources, most notably the global variables.

Pthreads

Pthreads Example

```c
/* t1.c */
#include <pthread.h>
#include <stdio.h>
long sum = 0;
void *threadfun(void *arg) {
    printf("Thread started working...\n");
    long sum = 0;
    long n = (long)arg;
    for (int i = 1; i <= n; i++) {
        sum = sum + i;
    }
    printf("sum: %ld\n", sum);
    printf("Thread finished its work.\n");
    return NULL;
}
int main(int argc, char **argv) {
    pthread_t tid;

    printf("main(): creating new thread...\n");
    pthread_create(&tid, NULL,
                   threadfun, (void *)10);
    pthread_join(tid, NULL);
    printf("main(): Thread returned.\n");
    printf("sum: %ld\n", sum);
    return 0;
}
```

```
$ ./t1
main(): creating new thread...
Thread started working...
Thread finished its work.
main(): Thread returned.
sum: 55
```

Recall that threads in the same process share many resources, most notably the global variables.

Pthreads

Pthreads Example

```c
/* t1.c */
#include <pthread.h>
#include <stdio.h>
long sum = 0;
void *threadfun(void *arg) {
    printf("Thread started working...\n");

    long n = (long)arg;
    for (int i = 1; i <= n; i++) {
        sum = sum + i;
    }

    printf("Thread finished its work.\n");
    return NULL;
}
int main(int argc, char **argv) {
    pthread_t tid;

    printf("main(): creating new thread...\n");
    pthread_create(&tid, NULL,
                   threadfun, (void *)10);
    pthread_join(tid, NULL);
    printf("main(): Thread returned.\n");
    printf("sum: %ld\n", sum);
    return 0;
}
```

```
$ ./t1
main(): creating new thread...
Thread started working...
Thread finished its work.
main(): Thread returned.
sum: 55
```

Recall that threads in the same process share many resources, most notably the global variables.

Pthreads Example

```c
/* t2.c */
#include <pthread.h>
#include <stdio.h>

void *threadfun(void *arg) {
    printf("Thread started working...\n");
    long sum = 0;
    long n = (long)arg;
    for (int i = 1; i <= n; i++) {
        sum = sum + i;
    }

    printf("Thread finished its work.\n");
    return (void *)sum;
}
int main(int argc, char **argv) {
    pthread_t tid;
    void *res;
    printf("main(): creating new thread...\n");
    pthread_create(&tid, NULL,
                   threadfun, (void *)10);
    pthread_join(tid, &res);
    printf("main(): Thread returned.\n");
    printf("sum: %ld\n", (long)res);
    return 0;
}
```

```
$ ./t2
main(): creating new thread...
Thread started working...
Thread finished its work.
main(): Thread returned.
sum: 55
```

Recall that threads in the same process share many resources, most notably the global variables.

Communication between threads can also be achieved by passing the return value to the main thread via

```c
pthread_join(pthread_t thread,
              void **retval);
```

Pthreads Example

```c
/* t2.c */
#include <pthread.h>
#include <stdio.h>

void *threadfun(void *arg) {
    printf("Thread started working...\n");
    long sum = 0;
    long n = (long)arg;
    for (int i = 1; i <= n; i++) {
        sum = sum + i;
    }

    printf("Thread finished its work.\n");
    return (void *)sum;
}
int main(int argc, char **argv) {
    pthread_t tid;
    void *res;
    printf("main(): creating new thread...\n");
    pthread_create(&tid, NULL,
                   threadfun, (void *)10);
    pthread_join(tid, &res);
    printf("main(): Thread returned.\n");
    printf("sum: %ld\n", (long)res);
    return 0;
}
```

```
$ ./t2
main(): creating new thread...
Thread started working...
Thread finished its work.
main(): Thread returned.
sum: 55
```

Recall that threads in the same process share many resources, most notably the global variables.

Communication between threads can also be achieved by passing the return value to the main thread via

```c
pthread_join(pthread_t thread,
             void **retval);
```

Note that `retval`'s type is `void **`, a pointer to a generic pointer. Explicit cast is required to interpret `retval`.

Pthreads Example

```c
/* t2.c */
#include <pthread.h>
#include <stdio.h>

void *threadfun(void *arg) {
    printf("Thread started working...\n");
    long sum = 0;
    long n = (long)arg;
    for (int i = 1; i <= n; i++) {
        sum = sum + i;
    }

    printf("Thread finished its work.\n");
    pthread_exit((void *)sum);
}
int main(int argc, char **argv) {
    pthread_t tid;
    void *res;
    printf("main(): creating new thread...\n");
    pthread_create(&tid, NULL,
                   threadfun, (void *)10);
    pthread_join(tid, &res);
    printf("main(): Thread returned.\n");
    printf("sum: %ld\n", (long)res);
    return 0;
}
```

```
$ ./t2
main(): creating new thread...
Thread started working...
Thread finished its work.
main(): Thread returned.
sum: 55
```

When exiting the thread, we can also **explicitly** call

```c
void pthread_exit(void *retval);
```

Pthreads

Pthreads Example

```c
/* t3.c */
#include <pthread.h>
#include <stdio.h>
void *threadfun(void *arg) {
    printf("Thread %ld started working...\n",
           pthread_self());
    long sum = 0;
    long n = (long)arg;
    for (int i = 1; i <= n; i++) {
        sum = sum + i;
    }
    printf("sum: %ld\n", sum);
    printf("Thread finished its work.\n");
    return NULL;
}
int main(int argc, char **argv) {
    pthread_t tid[10];
    printf("main(): creating new thread...\n");
    for (int t = 0; t < 10; t++)
        pthread_create(&tid[t], NULL,
                  threadfun, (void *)(t*10));
    for (int t = 0; t < 10; t++)
        pthread_join(tid[t], NULL);
    printf("main(): Thread returned.\n");
    return 0;
}
```

```
$ ./t3
main(): creating new thread...
Thread 138983810004544 started working...
sum: 0
Thread finished its work.
Thread 138983801611840 started working...
sum: 55
Thread finished its work.
Thread 138983793219136 started working...
sum: 210
Thread finished its work.
Thread 138983784826432 started working...
sum: 465
Thread finished its work.
Thread 138983776433728 started working...
sum: 820
Thread finished its work.
...
```

We often use an array of pthread_t to store the tid of multiple threads.

Thank you!