Threads & Concurrency

- Spring 2022
- Chi-Sheng Shih/Chung-Wei Lin
- National Taiwan University

國立臺灣大學

施吉昇

NEWS Lab
嵌入式系統暨無線網路實驗室

Outline

- Overview - Multitasking a Process
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples

Outline

- Overview - Multitasking a Process
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples

Multi-tasking a Process?

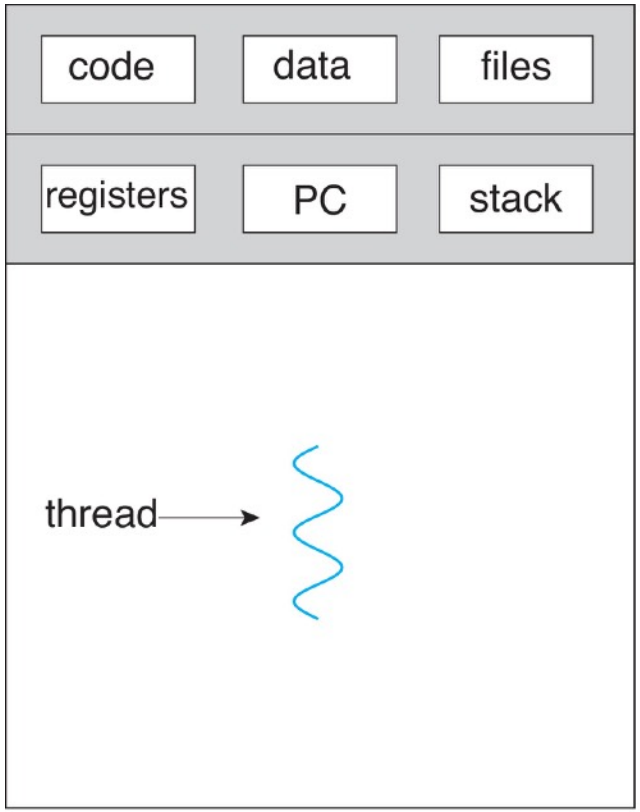
Can the system execute different functions of one process in time-sharing manners?

Motivation - Better Than Multiple Processes

- One process can have many *tasks* to accomplish, including
  - User interface,
  - Storage,
  - Computing, etc.
- Sequencing all the tasks can lead to poor responsiveness and performance.
- They are not all sequential and can be conducted in parallel. However, it is expensive to use multiple process including
  - demanding memory use,
  - unnecessary isolation among the sibling processes, and
  - competing resources with other applications/processes.
- Threading is designed to tackle aforementioned pitfalls. In particular, process creation is heavy-weight while thread creation is light-weight.
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request

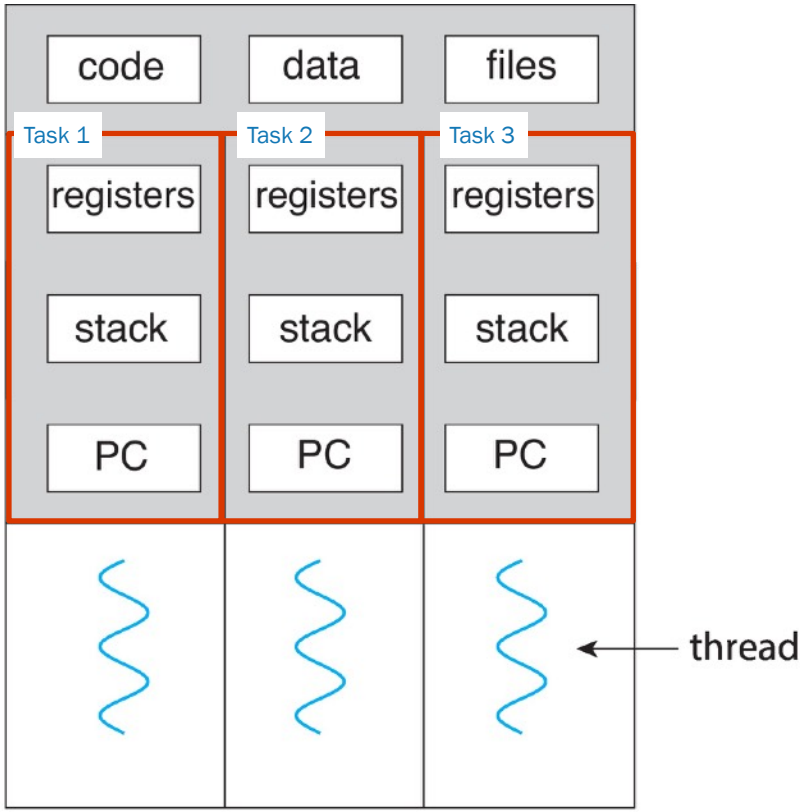
Multitasking in a Process

Sequential Process



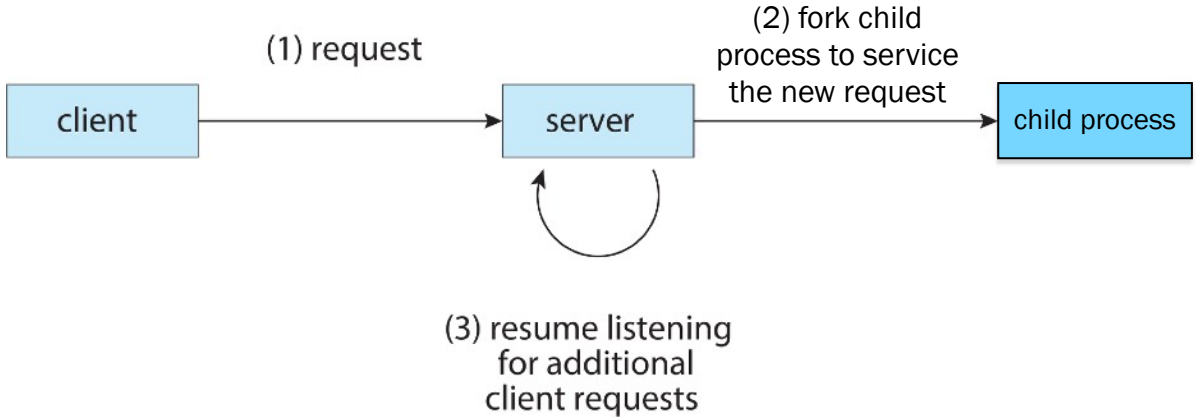
single-threaded process

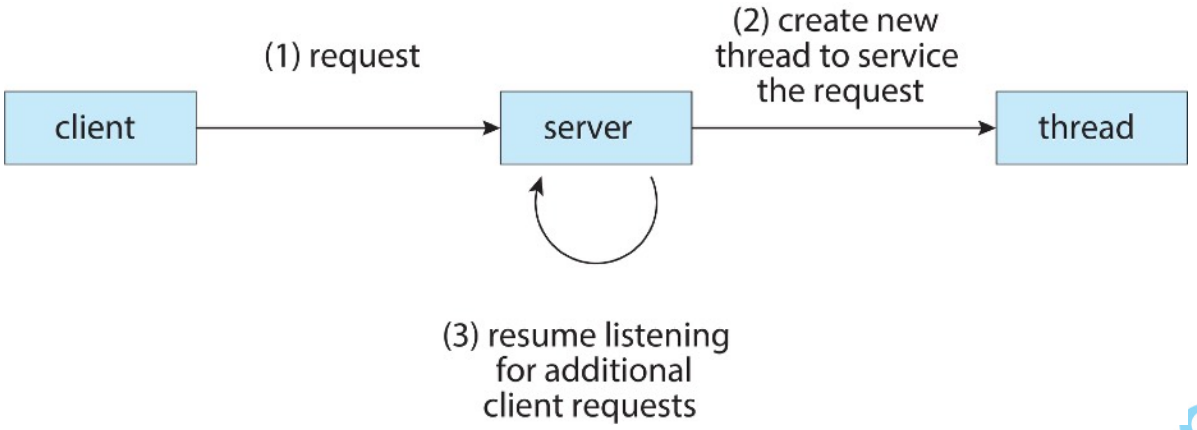
Concurrent Process



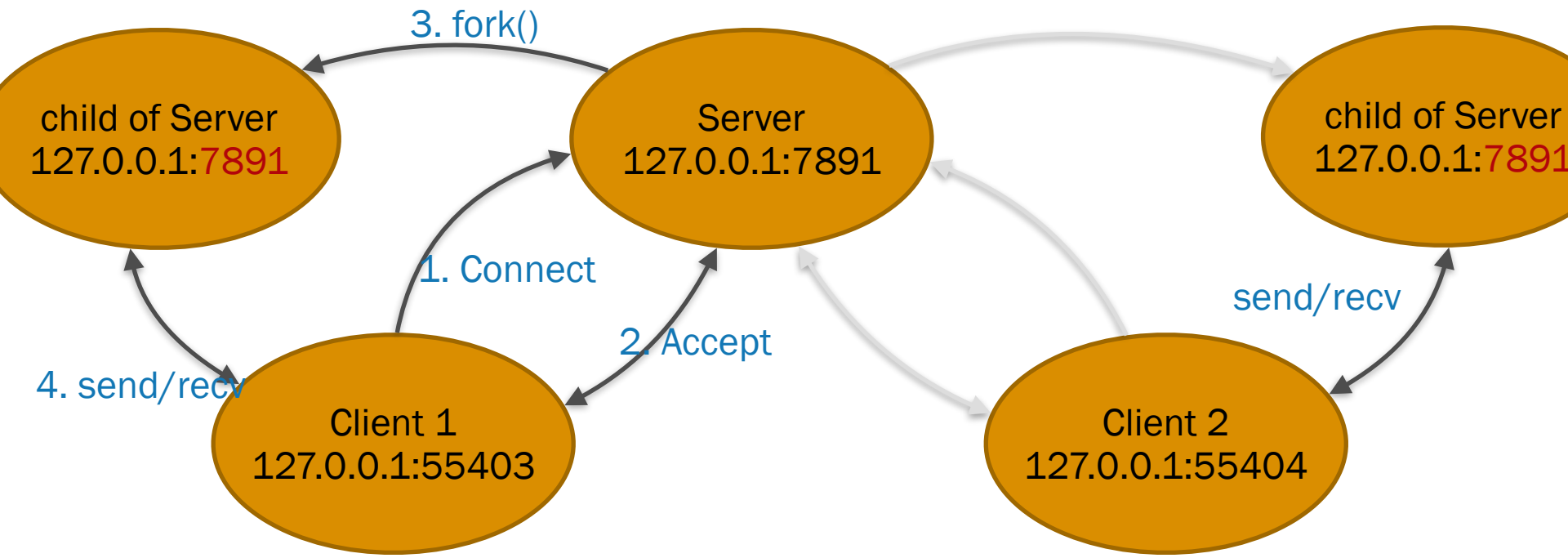
multithreaded process

Multiprocessed vs. Multithreaded Socket Server





Client - Server Process Recap



child of Server
127.0.0.1:55403

```
45 if(!fork()){
46   …
47
52   while(nBytes!=0){
53   …
59   }
60   printf("Close one socket.\n");
61   close(newSocket);
62   exit(0);
63
```

Server
127.0.0.1:7891

```
41   while(1){
42     newSocket =   …
68   }
69
70   return 0;
71 }
```

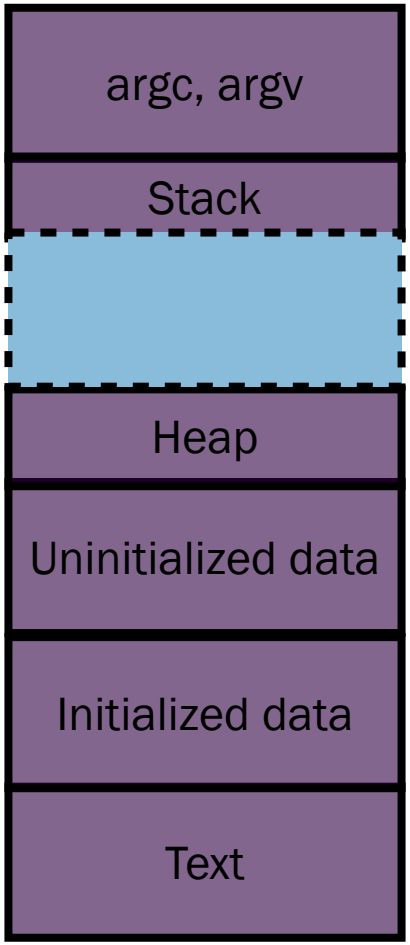
child of Server
127.0.0.1:55404

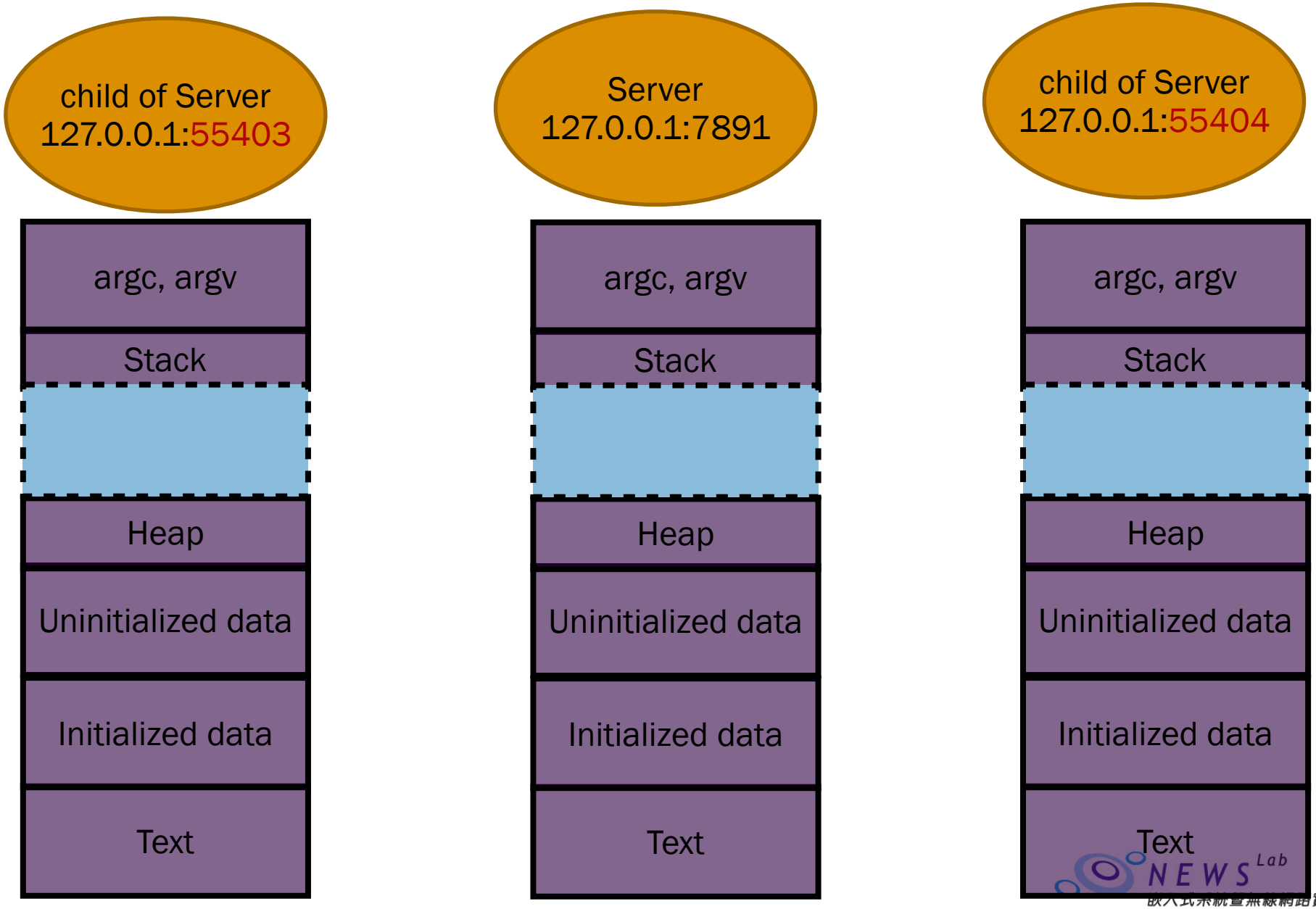
```
45 if(!fork()){
46   …
47
52   while(nBytes!=0){
53   …
59   }
60   printf("Close one s
61   close(newSocket);
62   exit(0);
63
```

child of Server
127.0.0.1:55403

Server
127.0.0.1:7891

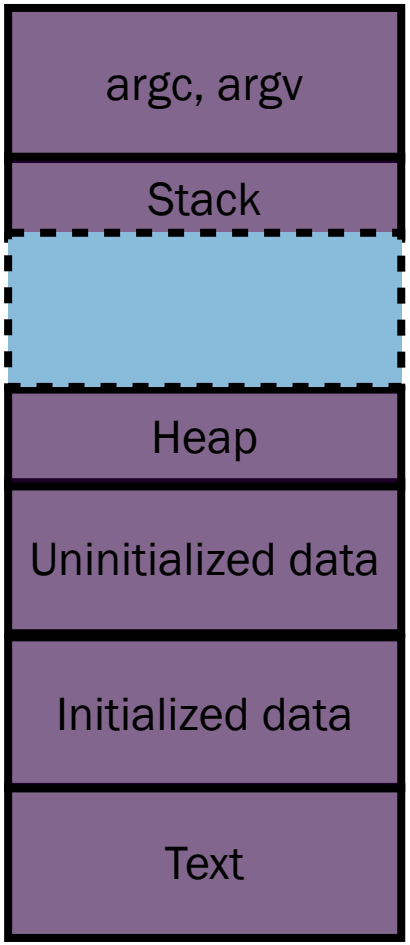
child of Server
127.0.0.1:55404

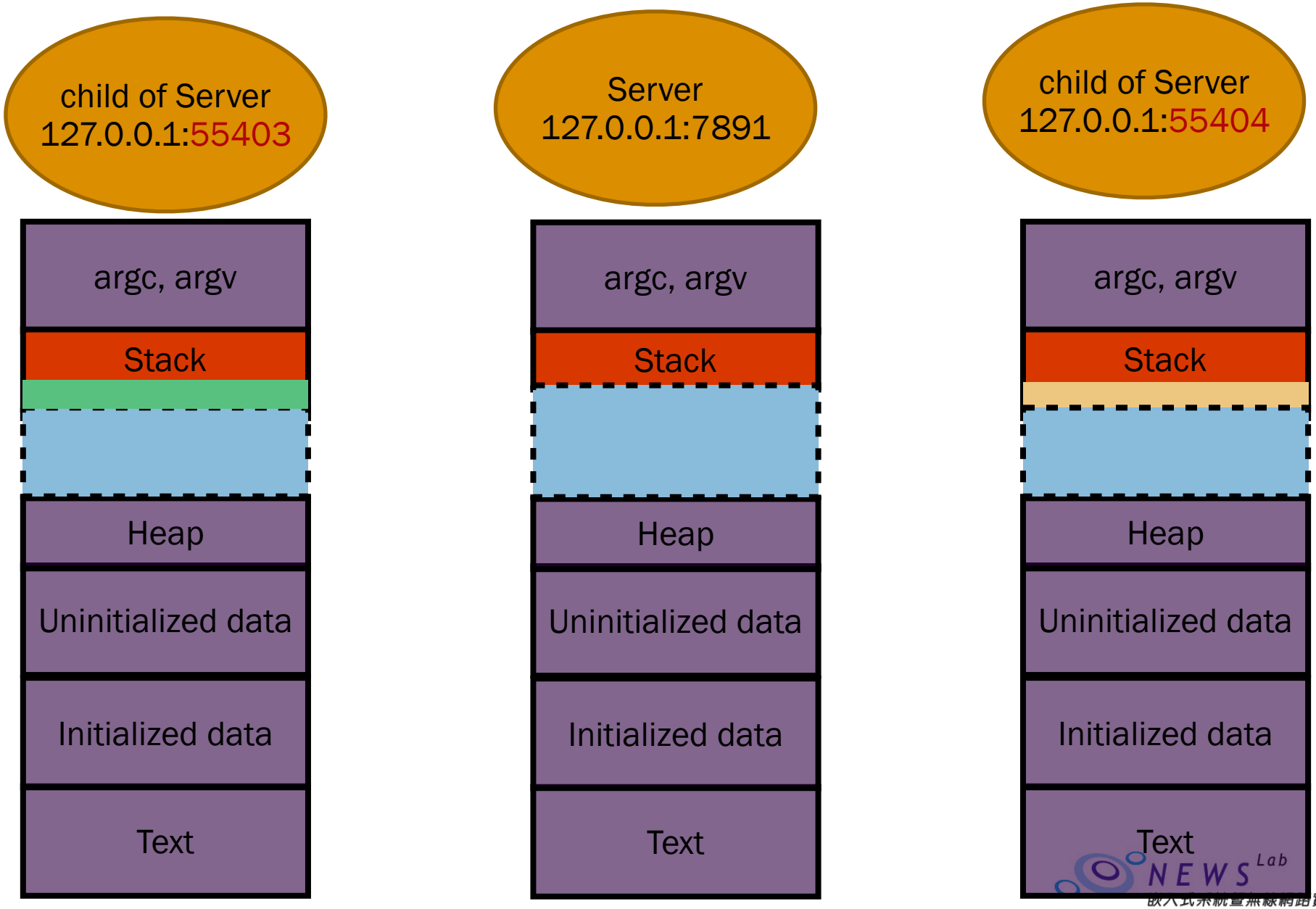



child of Server
127.0.0.1:55403
Server
127.0.0.1:7891
child of Server
127.0.0.1:55404
argc, argv
Stack
Heap
Uninitialized data
Initialized data
Text
argc, argv
Stack
Heap
Uninitialized data
Initialized data
Text
argc, argv
Stack
Heap
Uninitialized data
Initialized data
Text

How to achieve multithread on single processor?

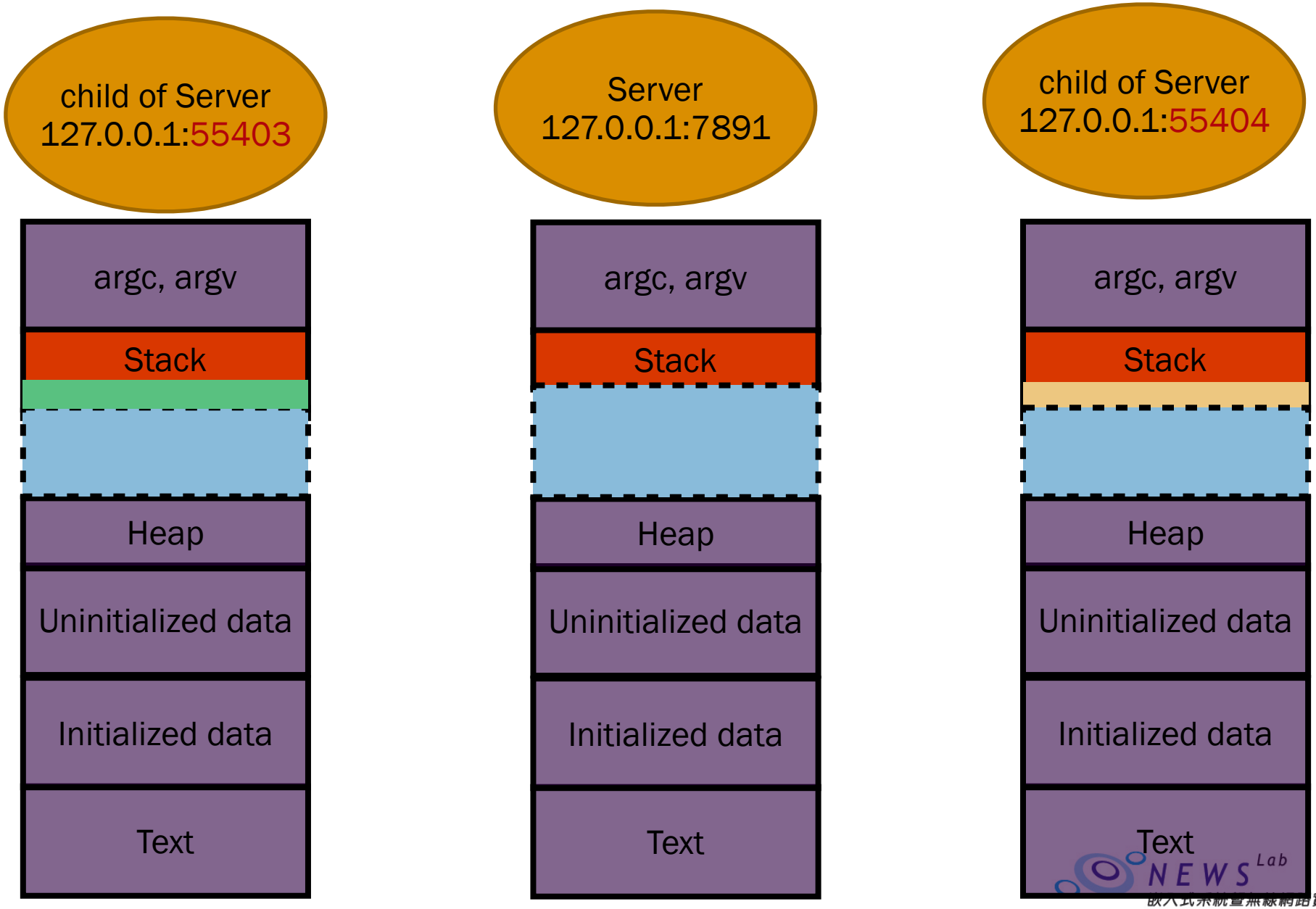
- What do we need to run these concurrent tasks in a process? From memory point of views,
  - Argc/argv: no need to change.
  - Stack: need to support concurrently access to different stack area. Each thread can only access its stack.
  - Heap: shared by the threads and need to avoid access conflict among threads.
  - Data: shared by the threads and need to avoid access conflict among threads.
  - Text: read-only and can be shared without extra efforts.

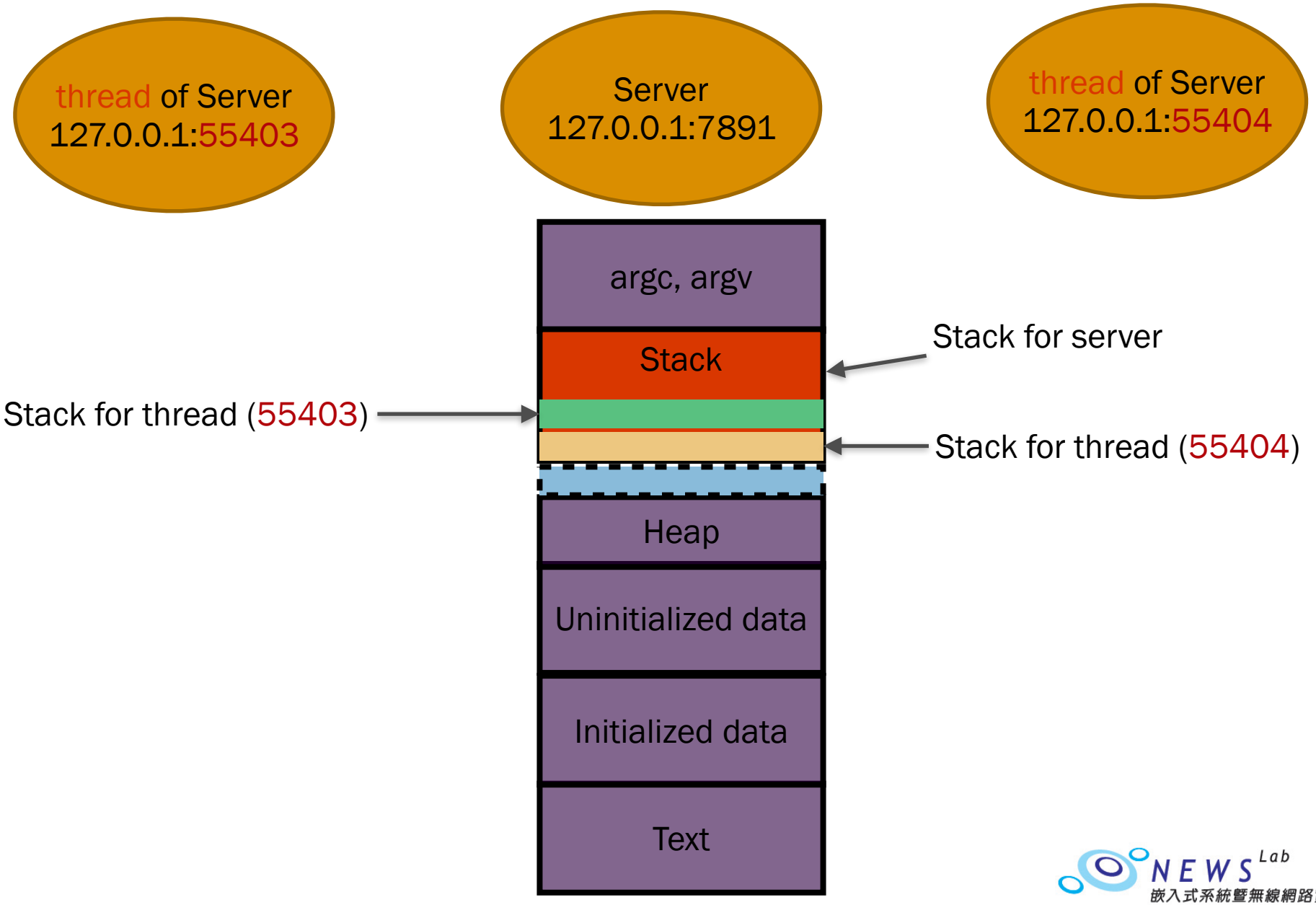


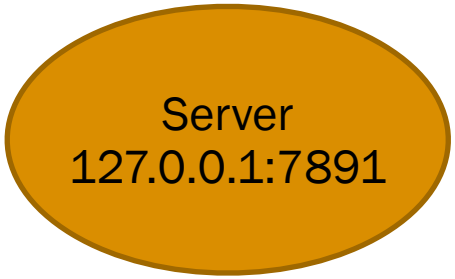

child of Server
127.0.0.1:55403
Server
127.0.0.1:7891
child of Server
127.0.0.1:55404
argc, argv
Stack
Heap
Uninitialized data
Initialized data
Text
argc, argv
Stack
Heap
Uninitialized data
Initialized data
Text
argc, argv
Stack
Heap
Uninitialized data
Initialized data
Text

How to achieve multithread on single processor?

- From CPU point of views:
  - Registers for each thread: increase the number of registers on CPU or limit the use on register for each thread.
  - Cache: reserve the cache for each thread (but how?)
- From OS point of views:
  - The threads of one process can share resources: physical and logical.
  - Reduce the frequency and overhead of context switch.


child of Server
127.0.0.1:55403
Server
127.0.0.1:7891
child of Server
127.0.0.1:55404
argc, argv
Stack
Heap
Uninitialized data
Initialized data
Text
argc, argv
Stack
Heap
Uninitialized data
Initialized data
Text
argc, argv
Stack
Heap
Uninitialized data
Initialized data
Text


thread of Server
127.0.0.1:55403
Server
127.0.0.1:7891
thread of Server
127.0.0.1:55404
argc, argv
Stack
Stack for server
Stack for thread (55403)
Stack for thread (55404)
Heap
Uninitialized data
Initialized data
Text



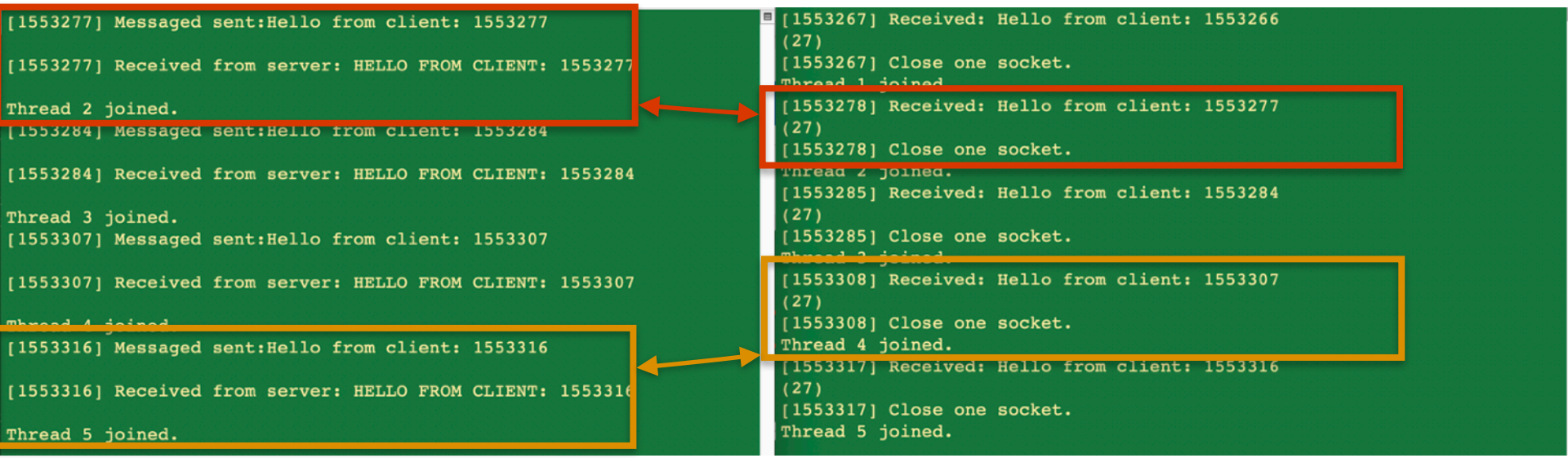

```
75    newSocket = accept(welcomeSocket, …
80    if( pthread_create(&tid[i++],…
81        printf("Failed to create thread\n");
82      }
83
```

```
16 void * socketThread(void *arg) {
…
30    /* loop while connection is live */
31    while ( nBytes!=0 ) {
32        if ( nBytes = recv(newSocket, …
33      printf("[%llu] Received: %s\n", …
```

```
16 void * socketThread(void *arg) {
…
30    /* loop while connection is liv
31    while ( nBytes!=0 ) {
32        if ( nBytes = recv(newSocke
33      printf("[%llu] Received: %s\r
```

```
86      while(i < nThread) {
87    pthread_join(tid[i++],NULL);
88    printf("Thread %d joined.\n", i);
89      }
9
```

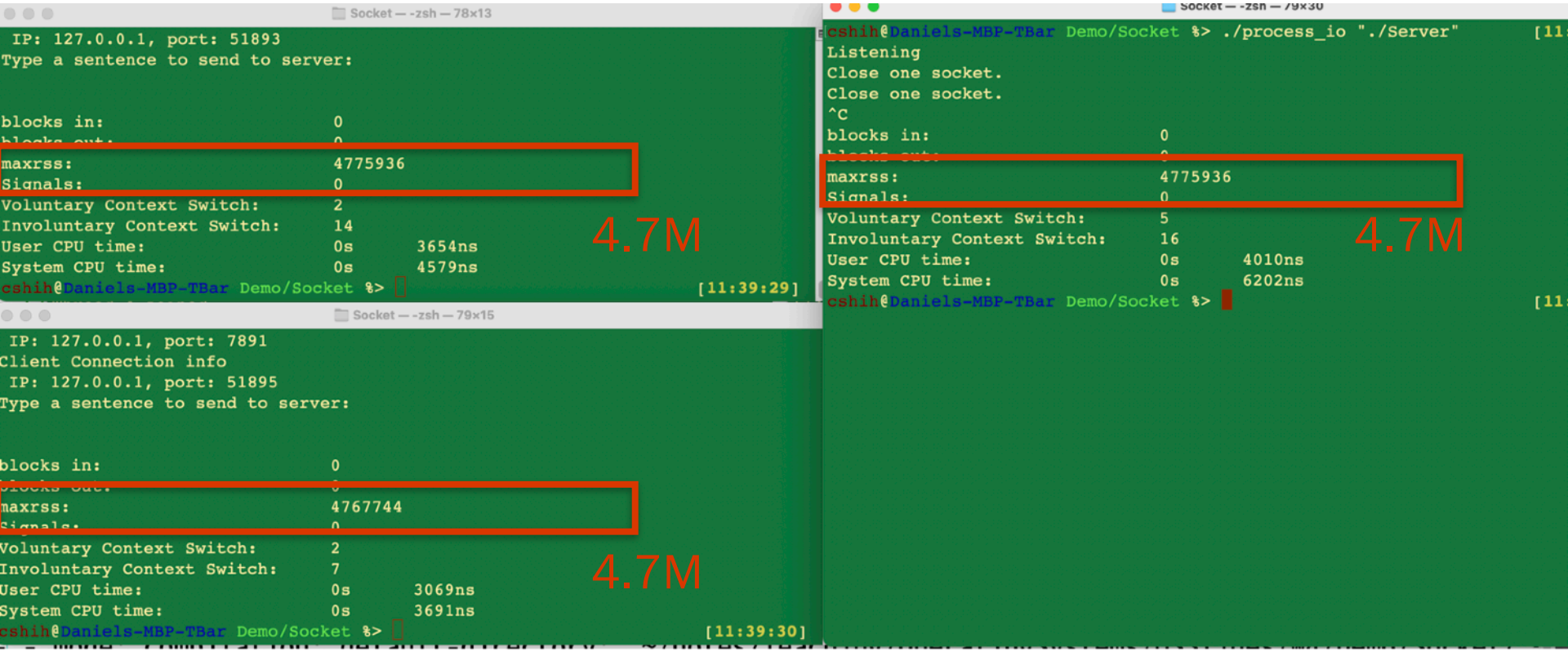
Multiple Thread Version for Socket Programming



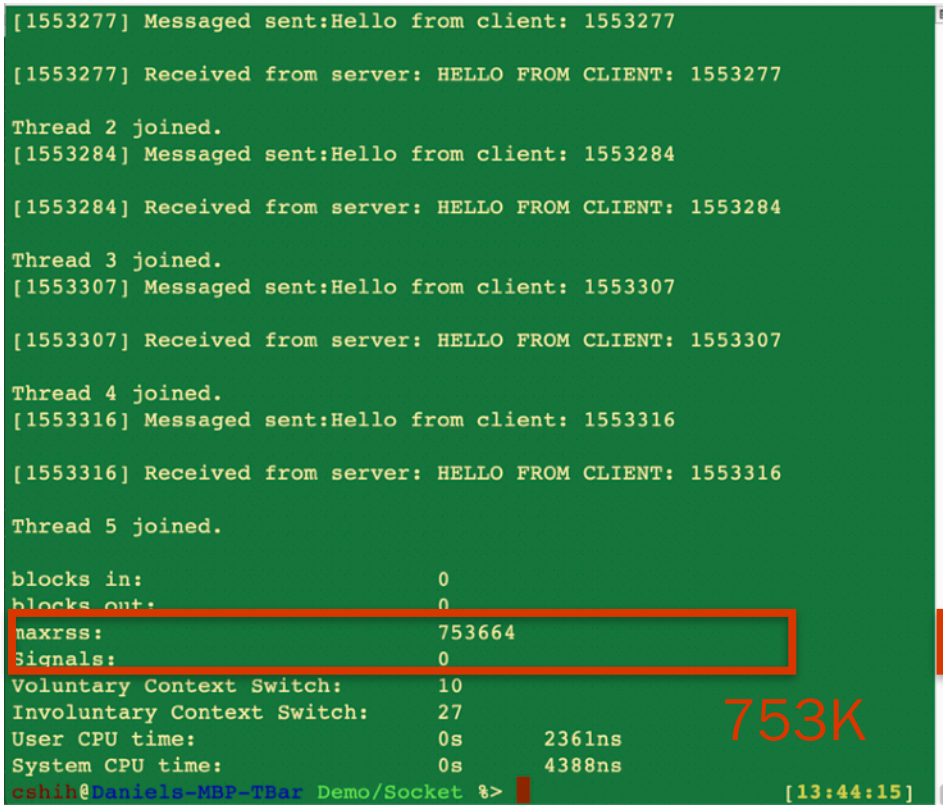
Client

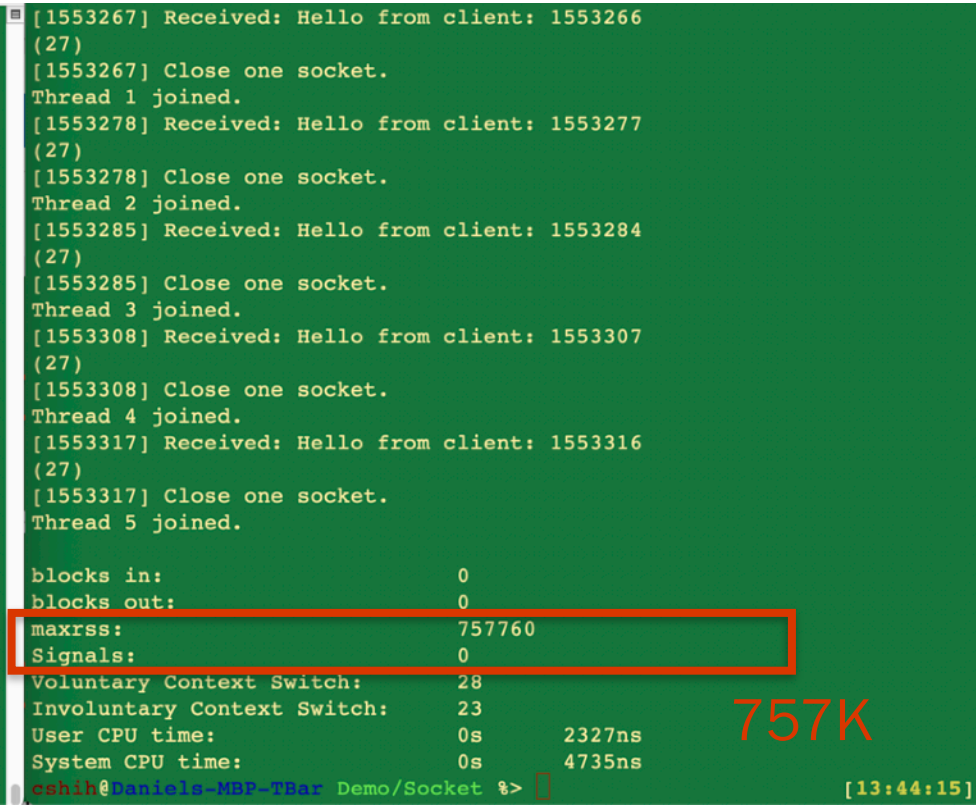
Server

Multiple Process Version - 2 Clients



Multiple Thread Version - 5 Clients





6 times less per process

Socket in Multiple Thread

Socket - Server

```
48 int main(){
49     int welcomeSocket, newSocket, portNum, clientLen, nBytes, i;
50     char buffer[1024];
51     struct sockaddr_in serverAddr;
52     struct sockaddr_storage serverStorage;
53     socklen_t addr_size;
54     pthread_t tid[60];
55
56     welcomeSocket = socket(PF_INET, SOCK_STREAM, 0);
57     portNum = 7891;
58     memset(&serverAddr, '\0', sizeof(serverAddr));
59     serverAddr.sin_family = AF_INET;
60     serverAddr.sin_port = htons(portNum);
61     serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
62     memset(serverAddr.sin_zero, '\0', sizeof serverAddr.sin_zero);
63     bind(welcomeSocket, (struct sockaddr *) &serverAddr, sizeof(serverAddr));
64
65     if(listen(welcomeSocket, 50)==0)
66   printf("Listening\n");
67     else
68   printf("Error\n");
69
70     addr_size = sizeof serverStorage;
```

Socket - Server (Cont.)

```
72    /*loop to keep accepting new connections*/
73    i = 0;
74    while (i < nThread){
75   newSocket = accept(welcomeSocket, (struct sockaddr *) &serverStorage, &addr_size);
76
77   // for each client request creates a thread and assign the client request
78   // to it to process
79   // so the main thread can entertain next request
80   if( pthread_create(&tid[i++], NULL, socketThread, &newSocket) != 0 )
81       printf("Failed to create thread\n");
82      }
83
84      // Collecting thread
85      i = 0;
86      while(i < nThread) {
87   pthread_join(tid[i++],NULL);
88   printf("Thread %d joined.\n", i);
89      }
90      return 0;
91 }
```

Socket - Server (Cont.)

```
16 void * socketThread(void *arg) {
17   int newSocket = *((int *)arg);
18   char buffer[1024];
19   int nBytes, i;
20   struct sockaddr_in childAddr;
21   uint64_t threadID;
22   pthread_threadid_np(NULL, &threadID);
23
24   printf("Starting thread: %llu\n", threadID);
25   memset(&childAddr, '\0', sizeof(childAddr));
26   int len = sizeof(childAddr);
27   nBytes = 1;
28   memset(&buffer, '\0', sizeof(buffer));
29
30   /* loop while connection is live */
31   while ( nBytes!=0 ) {
32       if ( nBytes = recv(newSocket, buffer, 1024, 0) > 0)
33     printf("[%llu] Received: %s(%d)\n", (unsigned int) threadID, buffer, strlen(buffer));
34       else
35     break;
36
37       for (i=0 ; i < strlen(buffer)-1 ; i++){
38     buffer[i] = toupper(buffer[i]);
39       }
40       send(newSocket, buffer, strlen(buffer), 0);
41       memset(&buffer, '\0', sizeof(buffer));
42   }
43   printf("[%llu] Close one socket.\n", (unsigned int) threadID);
44   close(newSocket);
45   pthread_exit(NULL);
46 }
47
```

Socket - Client

```
55 int main(){
56     int i = 0;
57     pthread_t tid[nThread];
58
59     while (i < nThread) {
60   if( pthread_create(&tid[i], NULL, clientThread, NULL) != 0 )
61       printf("Failed to create thread\n");
62   sleep(1);
63   i++;
64     }
65     i = 0;
66     while (i < nThread) {
67   pthread_join(tid[i++],NULL);
68   printf("Thread %d joined.\n", i);
69     }
70     return 0;
71 }
```

Socket - Client (Cont.)

```
11 int nThread = 5;
12
13 void * clientThread(void *arg) {
14     char message[1024];
15     int clientSocket;
16     struct sockaddr_in serverAddr;
17     socklen_t addr_size;
18     uint64_t threadID;
19     pthread_threadid_np(NULL, &threadID);
20
21     printf("Starging thread: %llu\n", threadID);
22     // Create the socket.
23     clientSocket = socket(PF_INET, SOCK_STREAM, 0);
24
25     // Configure settings of the server address
26     // Address family is Internet
27     serverAddr.sin_family = AF_INET;
28     serverAddr.sin_port = htons(7891);
29     serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
30
31     memset(serverAddr.sin_zero, '\0', sizeof serverAddr.sin_zero);
32     addr_size = sizeof serverAddr;
33     connect(clientSocket, (struct sockaddr *) &serverAddr, addr_size);
34
```

Socket - Client (Cont.)

```
35      sleep(5);
36      sprintf(message, "Hello from client: %llu\n", threadID);
37      if( send(clientSocket, message , strlen(message), 0) < 0)
38    printf("Send failed\n");
39      else
40    printf("[%llu] Messaged sent:%s\n", threadID, message);
41
42      // Read the message from the server into the buffer
43      if (recv(clientSocket, message, 1024, 0) < 0) {
44    printf("[%llu] Receive failed\n", threadID);
45      }
46      else // Print the received message
47    printf("[%llu] Received from server: %s\n", threadID, message);
48
49      send(clientSocket, "", 0, 0);
50      close(clientSocket);
51
52      pthread_exit(NULL);
53  }
```

How to achieve multithread on single processor?

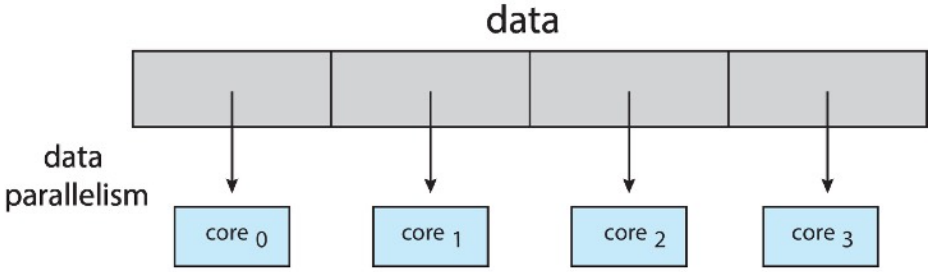
- From CPU point of views:
  - Registers for each thread: increase the number of registers on CPU or limit the use on register for each thread.
  - Cache: reserve the cache for each thread and how
- From OS point of views:
  - The threads of one process can share resources: physical and logical.
  - Reduce the frequency and overhead of context switch.
- In summary, concurrently running different functions, called multithreading, can be achieved by software, by using complier, system software, and operating systems.

How about multi-core processors?

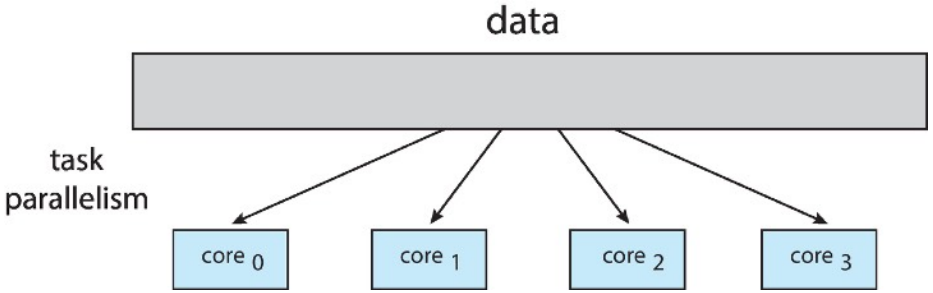
- Each core can be considered one processor.
- The operating system can treat the system as a distributed system and assign
  - one thread for each core or
    - each core can run a single processor private operating system.
  - multiple threads for each core.
    - each core can run a multiple threading private operating system.

How about multi-core processors?

- 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



Concurrent Programming

- Single Process on Single Processor
- Multiple Processes on Single Processor
- Single Instruction Single Data
- Multiple Instruction Multiple Data (MIMD)
  - Multi-core Processor
  - (HW) Hyper-Threading: virtualize the processing core
- Single Instruction Multiple Data (SIMD)
- **Zero** Instruction Multiple Data: Computation in Memory

Impact to Operating Systems

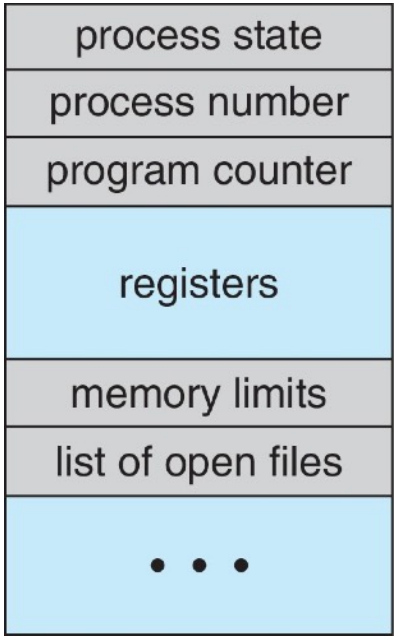
How about Process Control Block (PCB)?

- What should be changed to share the memory space?
  - Process ID
  - Process state
  - Program Counter
  - CPU Registers
  - CPU Scheduling information
  - Accounting information
  - I/O status information

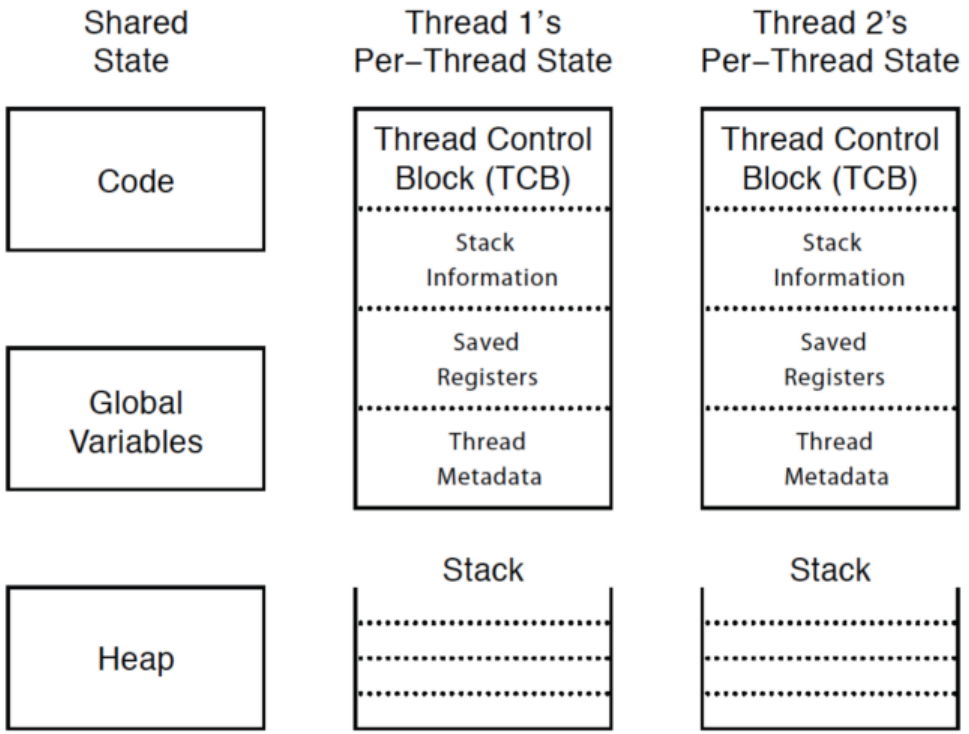
Thread Control Block

- Thread control block (TCB) includes five essential information below:
  - Current state (Runnable or Exited).
  - Registers
  - Status (EFLAGS)
  - Program Counter (EIP)
  - Stack

From PCB to TCB



PCB



TCB

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

How to estimate the speed-up

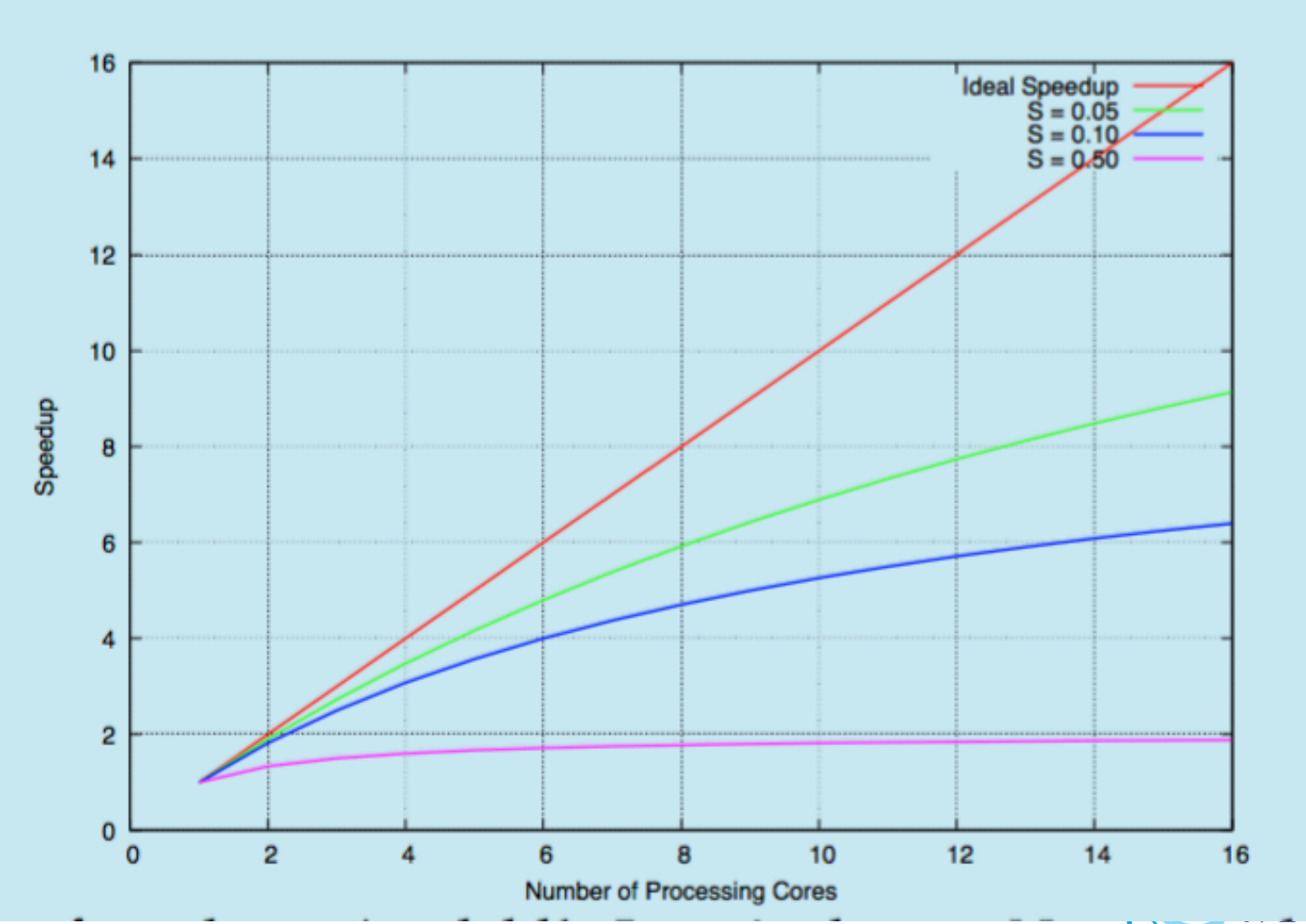
Amdahl's Law

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

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

  - For example: if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times (S=0.25, N=2)
- As N approaches infinity, speedup approaches 1 / S
  - Serial portion of an application has disproportionate effect on performance gained by adding additional cores
- But does the law take into account contemporary multicore systems? For example, GPU.

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

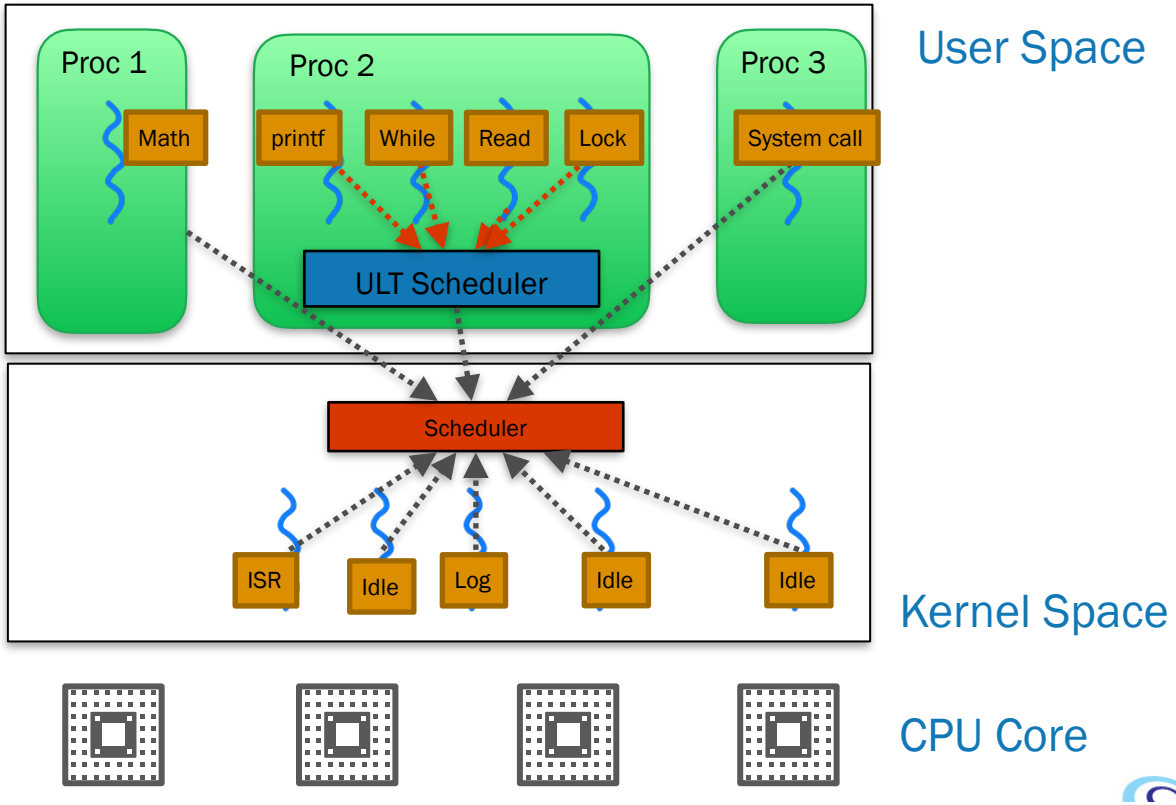
Multithreading Model

User Threads and Kernel Threads

- User threads - management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Windows threads
  - Java threads
- Kernel threads - Supported by the Kernel
- Examples – virtually all general -purpose operating systems, including:
  - Windows
  - Linux
  - Mac OS X
  - iOS
  - Android

Relationship between User Level Thread (ULT) and Kernel Level Thread (KLT)

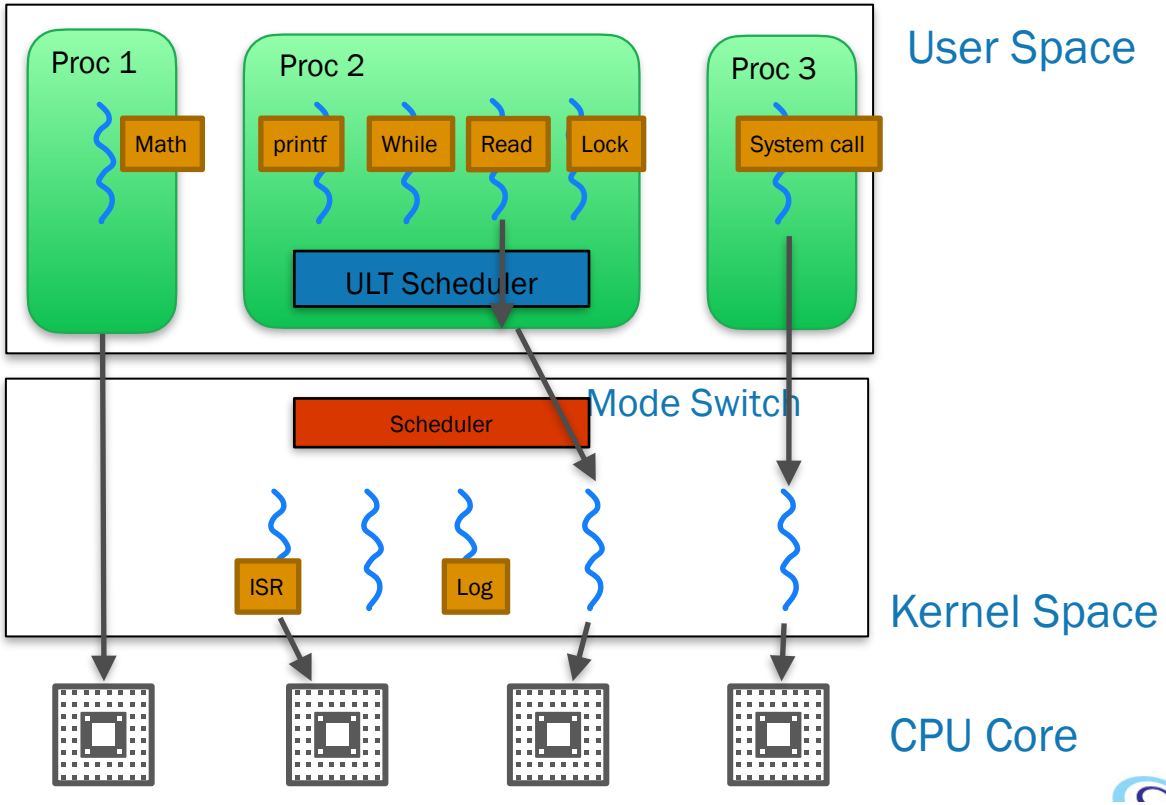
- On CPU, the KLT is the entity to be executed kernel mode instructions.
- Mode switch requires privilege change and thread switching (between ULT and KLT).
  - In some cases, it requires lock certain kernel resources and takes significant overhead.



Reference: Eykholt, Joseph R., et al. "Beyond Multiprocessing: Multithreading the SunOS Kernel." *USENIX Summer*. 1992.,

Relationship between User Level Thread (ULT) and Kernel Level Thread (KLT)

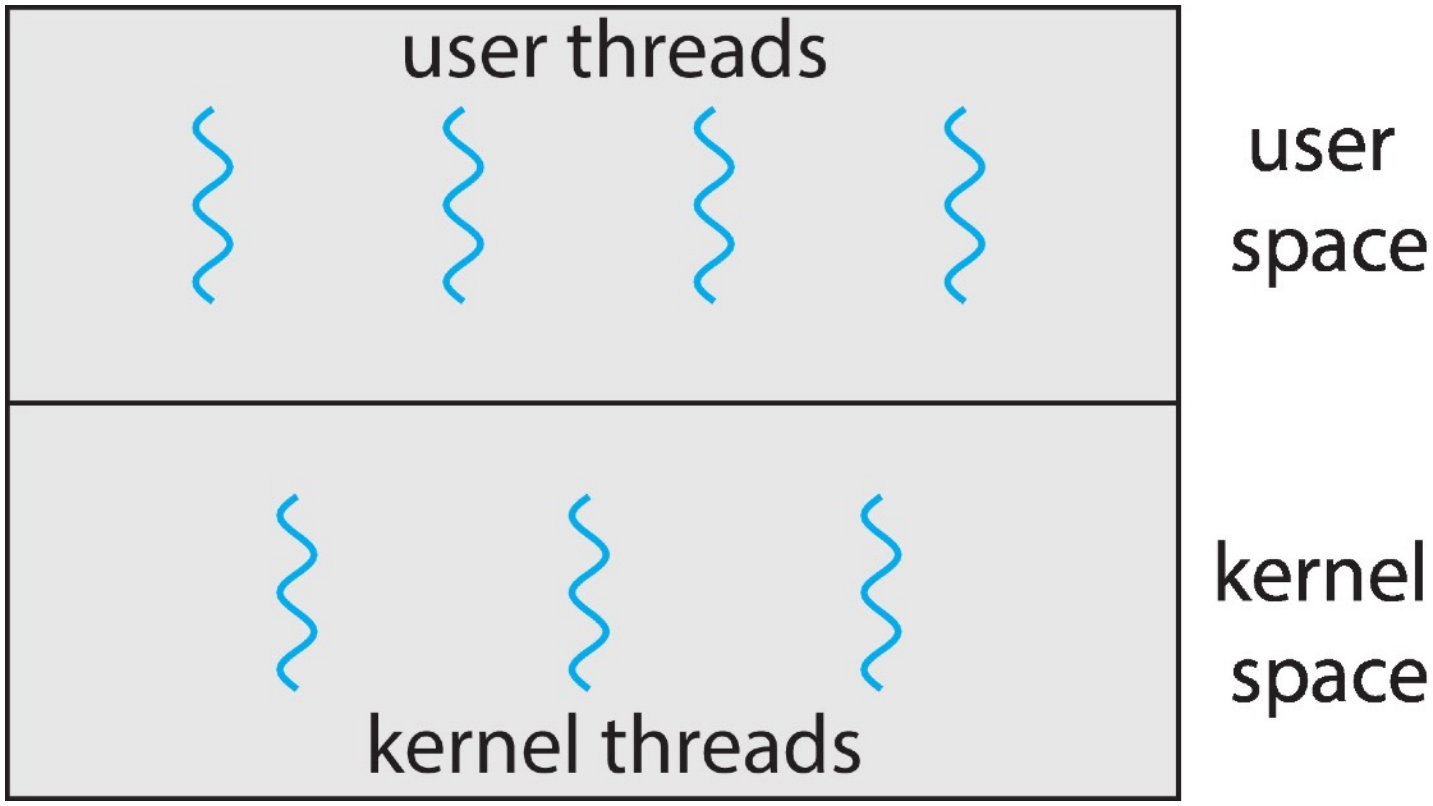
- On CPU, the KLT is the entity to be executed kernel mode instructions.
- Mode switch requires privilege change and thread switching (between ULT and KLT).
  - In some cases, it requires locking certain kernel resources and takes significant overhead.



Reference: Eykholt, Joseph R., et al. "Beyond Multiprocessing: Multithreading the SunOS Kernel." *USENIX Summer*. 1992.,

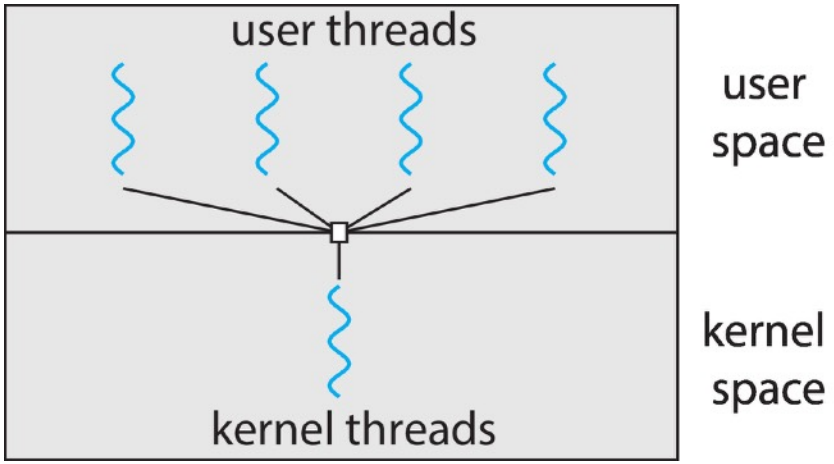
Multithreading Models

Mapping between User and Kernel Threads



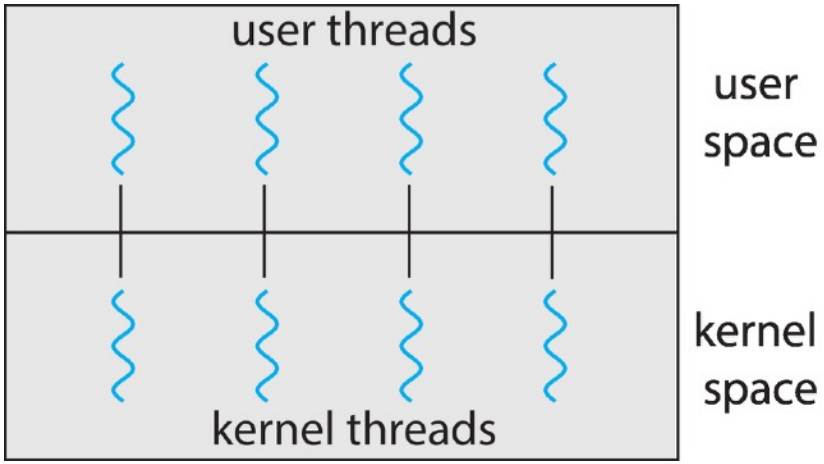
Many-to-One (N:1)

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



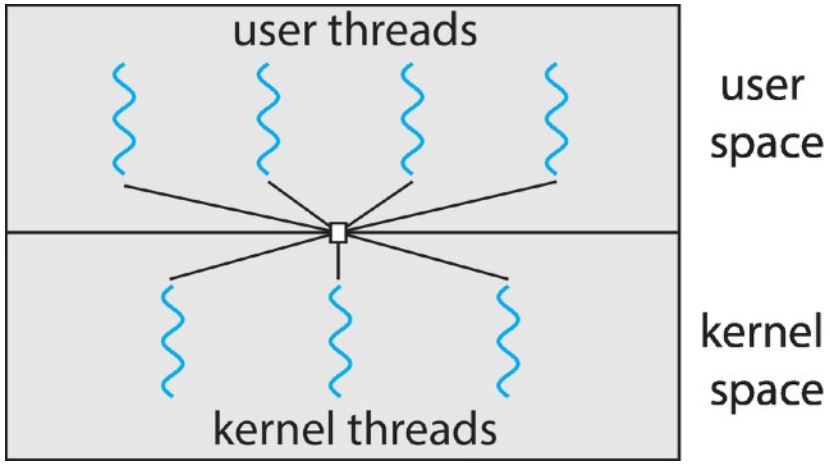
One-to-One (1:1)

- 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
  - Linux



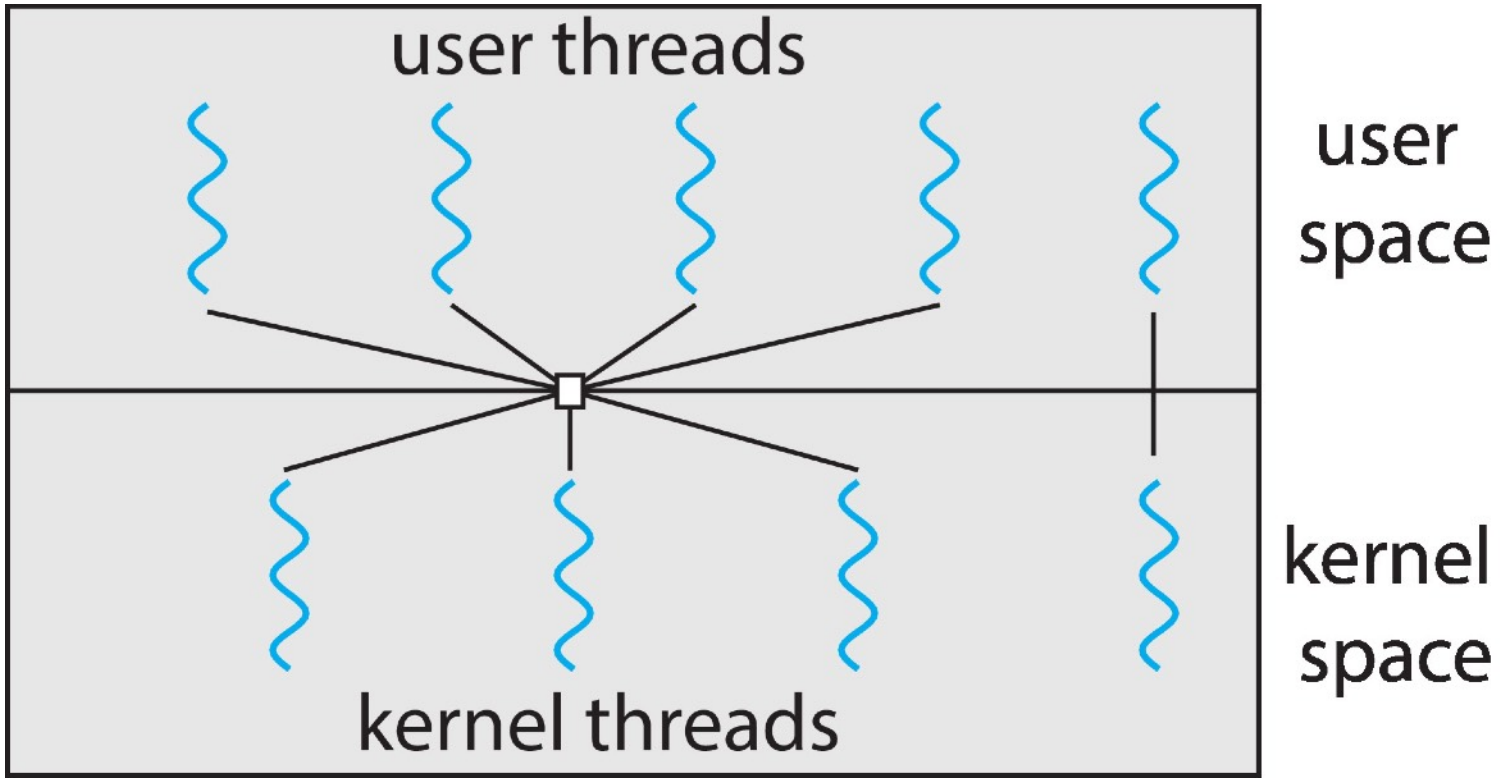
Many-to-Many Model (N:M)

- 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



Two-level Model

- Similar to N:M, except that it allows a user thread to be bound to kernel thread



Remarks

- Although the many-to-many model appears to be the most flexible of the models discussed, in practice it is difficult to implement.
- In addition, with an increasing number of processing cores appearing on most systems, limiting the number of kernel threads has become less important. As a result, most operating systems now use the one-to-one model.

Recap

- Below definitions greatly depend on architecture and OS implementation.
- From operating systems points of view,
  - **Process** (including Light Weight Process): the entity to allocate and manage resource.
  - **Thread**: the entity to execute instruction, called task in linux kernel.
    - User thread: an entity to execute instructions in user mode, i.e., user CPU time.
    - Kernel thread: an entity to execute instruction in kernel mode, i.e., system CPU time.
  - **Context Switch**: between execution entity
    - Process Context Switch:
    - Thread Context Switch:
  - **Mode Switch**: between user mode and kernel model

Implicit Threading - Threading Model

Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness is more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Five methods explored
  - Thread Pools
  - Fork-Join
  - OpenMP
  - Grand Central Dispatch
  - Intel Threading Building Blocks

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
  - Allows the number of threads in the application(s) to be bound to the size of the pool
  - 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
- Windows API supports thread pools:

```
DWORD WINAPI PoolFunction(AVOID Param) {
    /*
    * this function runs as a separate thread.
    */
}
```

Worker Thread

- Worker threads are created before it is requested and wait for task assignment.
  - When no assignment, it waits on signals.
  - When signaled (unlocked), it executes the function of the assigned task.
  - When done, it waits for next assignment.
  - One worker thread may execute different functions.
- Worker threads require explicitly free and shutdown.

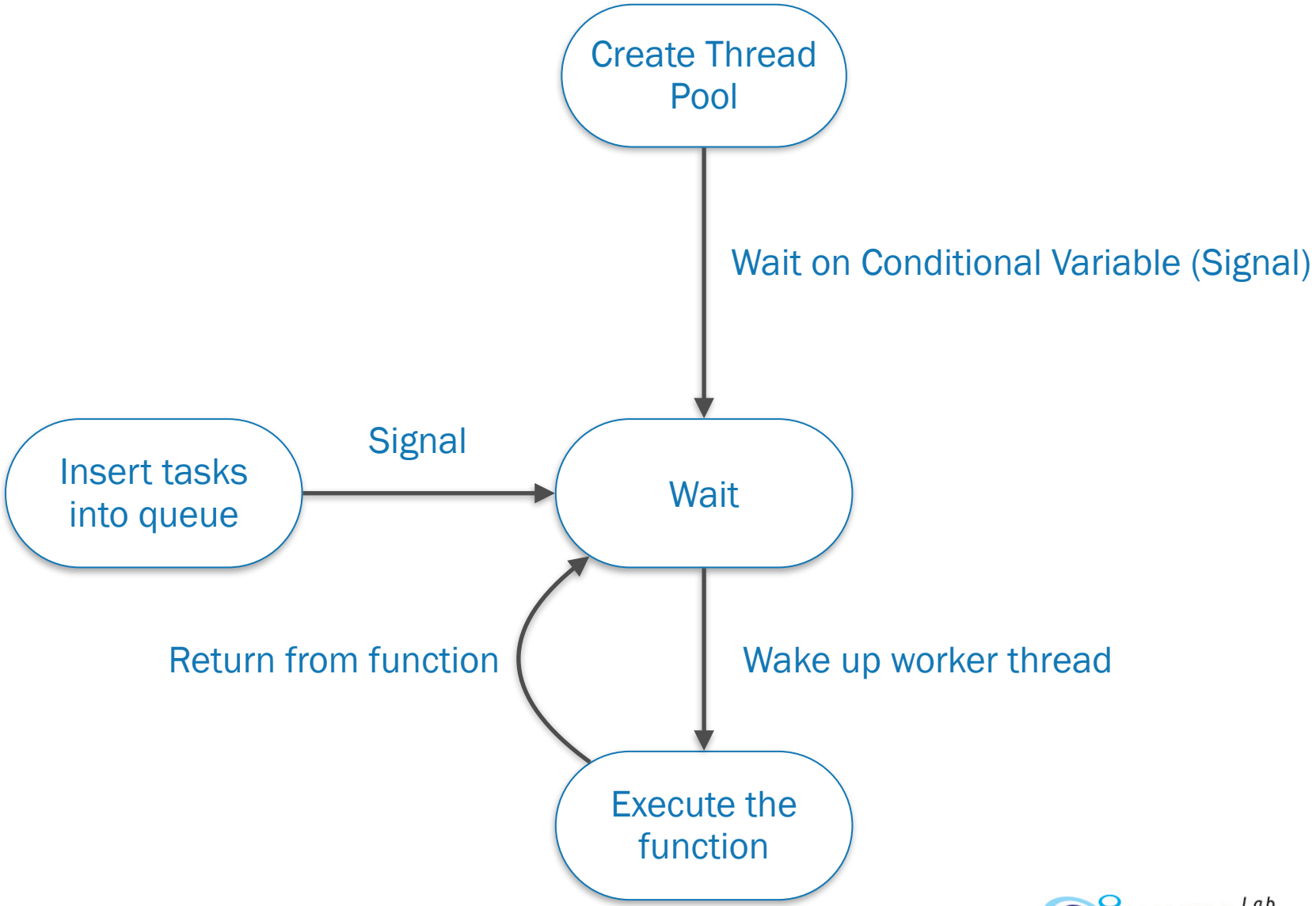
Threadpool for Socket Server

Example Code of Thread Pool

Socket Server

```
 53 int main(){
 54     int welcomeSocket, newSocket, portNum, clientLen, nBytes, i;
 55     char buffer[1024];

.  .  .

 72     assert((pool = threadpool_create(THREAD, QUEUE, 0)) != NULL);

.  .  .

 85     while (i < nThread){
 86  newSocket = accept(welcomeSocket, (struct sockaddr *) &serverStorage, &addr_size);
 87  i++;
 88  printf("Accept %d connection.\n", i);
 89
 90  // for each client request creates a thread and assign the client request
 91  // to it to process
 92  // so the main thread can entertain next request
 93  if (threadpool_add(pool, socketThread, &newSocket, 0) != 0)
 94      printf("Failed to add thread\n");
 95     }
 96

.  .  .
```

threadpool


Create Thread Pool
Wait on Conditional Variable (Signal)
Insert tasks into queue
Signal
Wait
Return from function
Wake up worker thread
Execute the function

Create the threadpool

```
 96 threadpool_t *threadpool_create(int thread_count, int queue_size, int flags)
 97 {
 98     if(thread_count <= 0 || thread_count > MAX_THREADS || queue_size <= 0 || queue_size > MAX_
 99         return NULL;
100     }
101
102     threadpool_t *pool;
103     int i;
104
105     if((pool = (threadpool_t *)malloc(sizeof(threadpool_t))) == NULL) {
106         goto err;
107     }
108
109     /* Initialize */
110     pool->thread_count = 0;
111     pool->queue_size = queue_size;
112     pool->head = pool->tail = pool->count = 0;
113     pool->shutdown = pool->started = 0;
114
115     /* Allocate thread and task queue */
116     pool->threads = (pthread_t *)malloc(sizeof(pthread_t) * thread_count);
117     pool->queue = (threadpool_task_t *)malloc
118         (sizeof(threadpool_task_t) * queue_size);
119
120     /* Initialize mutex and conditional variable first */
121     if((pthread_mutex_init(&(pool->lock), NULL) != 0) ||
122        (pthread_cond_init(&(pool->notify), NULL) != 0) ||
123        (pool->threads == NULL) ||
124        (pool->queue == NULL)) {
125         goto err;
126     }
127
128     /* Start worker threads */
129     for(i = 0; i < thread_count; i++) {
```

```
111     pool->queue_size = queue_size;
112     pool->head = pool->tail = pool->count = 0;
113     pool->shutdown = pool->started = 0;
114
115     /* Allocate thread and task queue */
116     pool->threads = (pthread_t *)malloc(sizeof(pthread_t) * thread_count);
117     pool->queue = (threadpool_task_t *)malloc
118         (sizeof(threadpool_task_t) * queue_size);
119
120     /* Initialize mutex and conditional variable first */
121     if((pthread_mutex_init(&(pool->lock), NULL) != 0) ||
122        (pthread_cond_init(&(pool->notify), NULL) != 0) ||
123        (pool->threads == NULL) ||
124        (pool->queue == NULL)) {
125         goto err;
126     }
127
128     /* Start worker threads */
129     for(i = 0; i < thread_count; i++) {
130         if(pthread_create(&(pool->threads[i]), NULL,
131                           threadpool_thread, (void*)pool) != 0) {
132             threadpool_destroy(pool, 0);
133             return NULL;
134         }
135         pool->thread_count++;
136         pool->started++;
137     }
138
139     return pool;
140
141 err:
142     if(pool) {
143         threadpool_free(pool);
144     }
145     return NULL;
146 }
```

threadpool - an array of Thread

```
58 /**
59  *  @struct threadpool
60  *  @brief The threadpool struct
61  *
62  *  @var notify       Condition variable to notify worker threads.
63  *  @var threads      Array containing worker threads ID.
64  *  @var thread_count Number of threads
65  *  @var queue        Array containing the task queue.
66  *  @var queue_size   Size of the task queue.
67  *  @var head         Index of the first element.
68  *  @var tail         Index of the next element.
69  *  @var count        Number of pending tasks
70  *  @var shutdown     Flag indicating if the pool is shutting down
71  *  @var started      Number of started threads
72  */
73 struct threadpool_t {
74   pthread_mutex_t lock;
75   pthread_cond_t notify;
76   pthread_t *threads;
77   threadpool_task_t *queue;
78   int thread_count;
79   int queue_size;
80   int head;
81   int tail;
82   int count;
83   int shutdown;
84   int started;
85 };
86
```

Assign task to the queue

```
148 int threadpool_add(threadpool_t *pool, void (*function)(void *),
149                     void *argument, int flags)
150 {
151     int err = 0;
152     int next;
153
154     if(pool == NULL || function == NULL) {
155         return threadpool_invalid;
156     }
157
158     if(pthread_mutex_lock(&(pool->lock)) != 0) {
159         return threadpool_lock_failure;
160     }
161
162     next = pool->tail + 1;
163     next = (next == pool->queue_size) ? 0 : next;
164
 . ..
190
191     if(pthread_mutex_unlock(&pool->lock) != 0) {
192         err = threadpool_lock_failure;
193     }
194
195     return err;
196 }
```

Assign task to the queue

```
162     next = pool->tail + 1;
163     next = (next == pool->queue_size) ? 0 : next;
164
165     do {
166         /* Are we full ? */
167         if(pool->count == pool->queue_size) {
168             err = threadpool_queue_full;
169             break;
170         }
171
172         /* Are we shutting down ? */
173         if(pool->shutdown) {
174             err = threadpool_shutdown;
175             break;
176         }
177
178         /* Add task to queue */
179         pool->queue[pool->tail].function = function;
180         pool->queue[pool->tail].argument = argument;
181         pool->tail = next;
182         pool->count += 1;
183
184         /* pthread_cond_broadcast */
185         if(pthread_cond_signal(&(pool->notify)) != 0) {
186             err = threadpool_lock_failure;
187             break;
188         }
189     } while(0);
190
```

Worker Thread

```
265 static void *threadpool_thread(void *threadpool)
266 {
267     threadpool_t *pool = (threadpool_t *)threadpool;
268     threadpool_task_t task;
269
270     for(;;) {
271         /* Lock must be taken to wait on conditional variable */
272         pthread_mutex_lock(&(pool->lock));
273
274         /* Wait on condition variable, check for spurious wakeups.
275            When returning from pthread_cond_wait(), we own the lock. */
276         while((pool->count == 0) && (!pool->shutdown)) {
277             pthread_cond_wait(&(pool->notify), &(pool->lock));
278         }
279
280         if((pool->shutdown == immediate_shutdown) ||
281            ((pool->shutdown == graceful_shutdown) &&
282             (pool->count == 0))) {
283             break;
284         }
285
286         /* Grab our task */
287         task.function = pool->queue[pool->head].function;
288         task.argument = pool->queue[pool->head].argument;
289         pool->head += 1;
290         pool->head = (pool->head == pool->queue_size) ? 0 : pool->head;
291         pool->count -= 1;
292
293         /* Unlock */
294         pthread_mutex_unlock(&(pool->lock));
295
296         /* Get to work */
297         (*(task.function))(task.argument);
298     }
299
300     pool->started--;
301
302     pthread_mutex_unlock(&(pool->lock));
303     pthread_exit(NULL);
304     return(NULL);
305 }
```

Multithread Client

```
56 int main(){
57     int i = 0;
58     pthread_t tid[nThread];
59
60     while (i < nThread) {
61   sleep(i*2);
62   if( pthread_create(&tid[i], NULL, clientThread, NULL) != 0 )
63       printf("Failed to create thread\n");
64   i++;
65     }
66     i = 0;
67     while (i < nThread) {
68       pthread_join(tid[i++],NULL);
69       printf("Thread %d joined.\n", i);
70     }
71     return 0;
72 }
```

Redundant Worker Threads

```
cshih@MacPro2507 Demo/threadpool %> ./ServerThreadpool
Pool started with 10 threads and queue size of 256
Listening
Accept 1 connection.
Starting server thread: 412144
Accept 2 connection.
Starting server thread: 412147
[412144] Received: Hello from client: 412203
(26)
[412144] Close one socket.
[412147] Received: Hello from client: 412235
(26)
[412147] Close one socket.
Accept 3 connection.
Starting server thread: 412145
[412145] Received: Hello from client: 412318
(26)
[412145] Close one socket.
Accept 4 connection.
Starting server thread: 412146
[412146] Received: Hello from client: 412460
(26)
[412146] Close one socket.
Accept 5 connection.
Starting server thread: 412148
[412148] Received: Hello from client: 412637
(26)
[412148] Close one socket.
```

```
cshih@MacPro2507 Demo/threadpool %> ./ClientThreadpool                [0]
Starging thread: 412203
Starging thread: 412235
[412203] Messaged sent:Hello from client: 412203

[412203] Received from server: HELLO FROM CLIENT: 412203

[412235] Messaged sent:Hello from client: 412235

[412235] Received from server: HELLO FROM CLIENT: 412235

Starging thread: 412318
[412318] Messaged sent:Hello from client: 412318

[412318] Received from server: HELLO FROM CLIENT: 412318

Starging thread: 412460
[412460] Messaged sent:Hello from client: 412460

[412460] Received from server: HELLO FROM CLIENT: 412460

Starging thread: 412637
Thread 1 joined.
Thread 2 joined.
Thread 3 joined.
Thread 4 joined.
[412637] Messaged sent:Hello from client: 412637

[412637] Received from server: HELLO FROM CLIENT: 412637

Thread 5 joined.
cshih@MacPro2507 Demo/threadpool %>                                   [0]
```

Each connection uses different worker thread to send/recv messages.

```
cshih@MacPro2507 Demo/threadpool %> ../Socket/process_io "./ServerThreadpool"
Pool started with 10 threads and queue size of 256
Listening
Accept 1 connection.
Starting server thread: 421538
Accept 2 connection.
Starting server thread: 421540
[421538] Received: Hello from client: 421610
(26)
[421538] Close one socket.
[421540] Received: Hello from client: 421667
(26)
[421540] Close one socket.
Accept 3 connection.
Starting server thread: 421541
[421541] Received: Hello from client: 421787
(26)
[421541] Close one socket.
Accept 4 connection.
Starting server thread: 421542
[421542] Received: Hello from client: 421938
(26)
[421542] Close one socket.
Accept 5 connection.
Starting server thread: 421539
[421539] Received: Hello from client: 422139
(26)
[421539] Close one socket.
Start to destroy threadpool.

blocks in:                      0
blocks out:                     0
maxrss:                         831488
Signals:                        0
Voluntary Context Switch:       15
Involuntary Context Switch:     49
User CPU time:                  0s      2282ns
System CPU time:                0s      3965ns
cshih@MacPro2507 Demo/threadpool %>
```

```
cshih@MacPro2507 Demo/threadpool %> ../Socket/process_io "./ClientThreadpool"
Starging thread: 421610
Starging thread: 421667
[421610] Messaged sent:Hello from client: 421610

[421610] Received from server: HELLO FROM CLIENT: 421610

[421667] Messaged sent:Hello from client: 421667

[421667] Received from server: HELLO FROM CLIENT: 421667

Starging thread: 421787
[421787] Messaged sent:Hello from client: 421787

[421787] Received from server: HELLO FROM CLIENT: 421787

Starging thread: 421938
[421938] Messaged sent:Hello from client: 421938

[421938] Received from server: HELLO FROM CLIENT: 421938

Starging thread: 422139
Thread 1 joined.
Thread 2 joined.
Thread 3 joined.
Thread 4 joined.
[422139] Messaged sent:Hello from client: 422139

[422139] Received from server: HELLO FROM CLIENT: 422139

Thread 5 joined.

blocks in:                      0
blocks out:                     0
maxrss:                         761856
Signals:                        0
Voluntary Context Switch:       10
Involuntary Context Switch:     32
User CPU time:                  0s      2121ns
System CPU time:                0s      4201ns
cshih@MacPro2507 Demo/threadpool %>                                     [0]
```

Less Worker Thread

```
cshih@MacPro2507 Demo/threadpool %> ./ServerThreadpool
Pool started with 3 threads and queue size of 256
Listening
Accept 1 connection.
Starting server thread: 410569
Accept 2 connection.
Starting server thread: 410571
[410569] Received: Hello from client: 410607
(26)
[410569] Close one socket.
[410571] Received: Hello from client: 410653
(26)
[410571] Close one socket.
Accept 3 connection.
Starting server thread: 410570
[410570] Received: Hello from client: 410758
(26)
[410570] Close one socket.
Accept 4 connection.
Starting server thread: 410569
[410569] Received: Hello from client: 410948
(26)
[410569] Close one socket.
Accept 5 connection.
Starting server thread: 410571
[410571] Received: Hello from client: 411100
(26)
[410571] Close one socket.
```

```
cshih@MacPro2507 Demo/threadpool %> ./ClientThreadpool                [0]
Starging thread: 410607
Starging thread: 410653
[410607] Messaged sent:Hello from client: 410607

[410607] Received from server: HELLO FROM CLIENT: 410607

[410653] Messaged sent:Hello from client: 410653

[410653] Received from server: HELLO FROM CLIENT: 410653

Starging thread: 410758
[410758] Messaged sent:Hello from client: 410758

[410758] Received from server: HELLO FROM CLIENT: 410758

Starging thread: 410948
[410948] Messaged sent:Hello from client: 410948

[410948] Received from server: HELLO FROM CLIENT: 410948

Starging thread: 411100
Thread 1 joined.
Thread 2 joined.
Thread 3 joined.
Thread 4 joined.
[411100] Messaged sent:Hello from client: 411100

[411100] Received from server: HELLO FROM CLIENT: 411100

Thread 5 joined.
cshih@MacPro2507 Demo/threadpool %>                                   [0]
```

Some connections have to re-use worker threads to send/recv messages.
Which threads are assigned for more than once?

410569 and 410571

Less Worker Thread

```
cshih@MacPro2507 Demo/threadpool %> ../Socket/process_io "./ServerThreadpool"
Pool started with 3 threads and queue size of 256
Listening
Accept 1 connection.
Starting server thread: 424727
Accept 2 connection.
Starting server thread: 424729
[424727] Received: Hello from client: 424786
(26)
[424727] Close one socket.
[424729] Received: Hello from client: 424820
(26)
[424729] Close one socket.
Accept 3 connection.
Starting server thread: 424728
[424728] Received: Hello from client: 424966
(26)
[424728] Close one socket.
Accept 4 connection.
Starting server thread: 424727
[424727] Received: Hello from client: 425140
(26)
[424727] Close one socket.
Accept 5 connection.
Starting server thread: 424729
[424729] Received: Hello from client: 425436
(26)
[424729] Close one socket.
Start to destroy threadpool.

blocks in:                      0
blocks out:                     0
maxrss:                         774144
Signals:                        0
Voluntary Context Switch:       15
Involuntary Context Switch:     27
User CPU time:                  0s      2378ns
System CPU time:                0s      3628ns
cshih@MacPro2507 Demo/threadpool %>
```

```
cshih@MacPro2507 Demo/threadpool %> ../Socket/process_io "./ClientThreadpool"
Starging thread: 424786
Starging thread: 424820
[424786] Messaged sent:Hello from client: 424786

[424786] Received from server: HELLO FROM CLIENT: 424786

[424820] Messaged sent:Hello from client: 424820

[424820] Received from server: HELLO FROM CLIENT: 424820

Starging thread: 424966
[424966] Messaged sent:Hello from client: 424966

[424966] Received from server: HELLO FROM CLIENT: 424966

Starging thread: 425140
[425140] Messaged sent:Hello from client: 425140

[425140] Received from server: HELLO FROM CLIENT: 425140

Starging thread: 425436
Thread 1 joined.
Thread 2 joined.
Thread 3 joined.
Thread 4 joined.
[425436] Messaged sent:Hello from client: 425436

[425436] Received from server: HELLO FROM CLIENT: 425436

Thread 5 joined.

blocks in:                      0
blocks out:                     0
maxrss:                         770048
Signals:                        0
Voluntary Context Switch:       10
Involuntary Context Switch:     38
User CPU time:                  0s      2516ns
System CPU time:                0s      6357ns
cshih@MacPro2507 Demo/threadpool %>                                    [0]
```

Compare Resource Usage

10 worker threads

```
blocks in:                      0
blocks out:                     0
maxrss:                         831488
Signals:                        0
Voluntary Context Switch:       15
Involuntary Context Switch:     49
User CPU time:                  0s      2282ns
System CPU time:                0s      3965ns
cshih@MacPro2507 Demo/threadpool %>
```

```
Thread 5 joined.

blocks in:                      0
blocks out:                     0
maxrss:                         761856
Signals:                        0
Voluntary Context Switch:       10
Involuntary Context Switch:     32
User CPU time:                  0s      2121ns
System CPU time:                0s      4201ns
cshih@MacPro2507 Demo/threadpool %>                                    [0]
```

3 worker threads

```
blocks in:                      0
blocks out:                     0
maxrss:                         774144
Signals:                        0
Voluntary Context Switch:       15
Involuntary Context Switch:     27
User CPU time:                  0s      2378ns
System CPU time:                0s      3628ns
cshih@MacPro2507 Demo/threadpool %>
```

```
Thread 5 joined.

blocks in:                      0
blocks out:                     0
maxrss:                         770048
Signals:                        0
Voluntary Context Switch:       10
Involuntary Context Switch:     38
User CPU time:                  0s      2516ns
System CPU time:                0s      6357ns
```

What if 100 threads are created?

Creating 100 threads

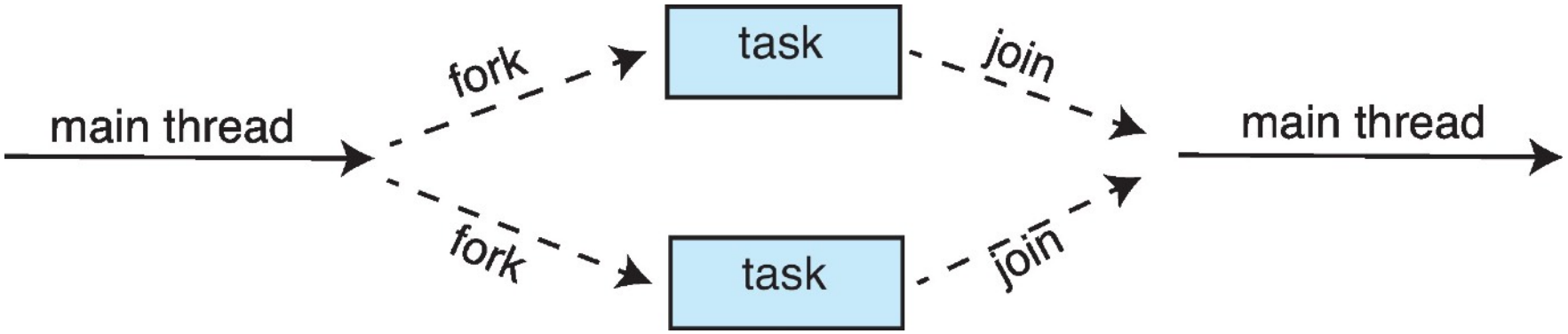
```
cshih@MacPro2507 Demo/threadpool %> ../Socket/process_io "./ServerThreadpool"
Pool started with 100 threads and queue size of 256
Listening
Accept 1 connection.
Starting server thread: 445419
Accept 2 connection.
Starting server thread: 445420
[445419] Received: Hello from client: 445622
(26)
[445419] Close one socket.
[445420] Received: Hello from client: 445662
(26)
[445420] Close one socket.
Accept 3 connection.
Starting server thread: 445422
[445422] Received: Hello from client: 445802
(26)
[445422] Close one socket.
Accept 4 connection.
Starting server thread: 445423
[445423] Received: Hello from client: 445974
(26)
[445423] Close one socket.
Accept 5 connection.
Starting server thread: 445421
[445421] Received: Hello from client: 446250
(26)
[445421] Close one socket.
Start to destroy threadpool.

blocks in:                      0
blocks out:                     0
maxrss:                         1572864
Signals:                        0
Voluntary Context Switch:       15
Involuntary Context Switch:     510
User CPU time:                  0s      3268ns
System CPU time:                0s      10694ns
cshih@MacPro2507 Demo/threadpool %>
```

```
cshih@MacPro2507 Demo/threadpool %> ../Socket/process_io "./ClientThreadpool"
Starging thread: 445622
Starging thread: 445662
[445622] Messaged sent:Hello from client: 445622

[445622] Received from server: HELLO FROM CLIENT: 445622

[445662] Messaged sent:Hello from client: 445662

[445662] Received from server: HELLO FROM CLIENT: 445662

Starging thread: 445802
[445802] Messaged sent:Hello from client: 445802

[445802] Received from server: HELLO FROM CLIENT: 445802

Starging thread: 445974
[445974] Messaged sent:Hello from client: 445974

[445974] Received from server: HELLO FROM CLIENT: 445974

Starging thread: 446250
Thread 1 joined.
Thread 2 joined.
Thread 3 joined.
Thread 4 joined.
[446250] Messaged sent:Hello from client: 446250

[446250] Received from server: HELLO FROM CLIENT: 446250

Thread 5 joined.

blocks in:                      0
blocks out:                     0
maxrss:                         761856
Signals:                        0
Voluntary Context Switch:       10
Involuntary Context Switch:     21
User CPU time:                  0s      2298ns
System CPU time:                0s      4753ns
cshih@MacPro2507 Demo/threadpool %>                                    [0]
```

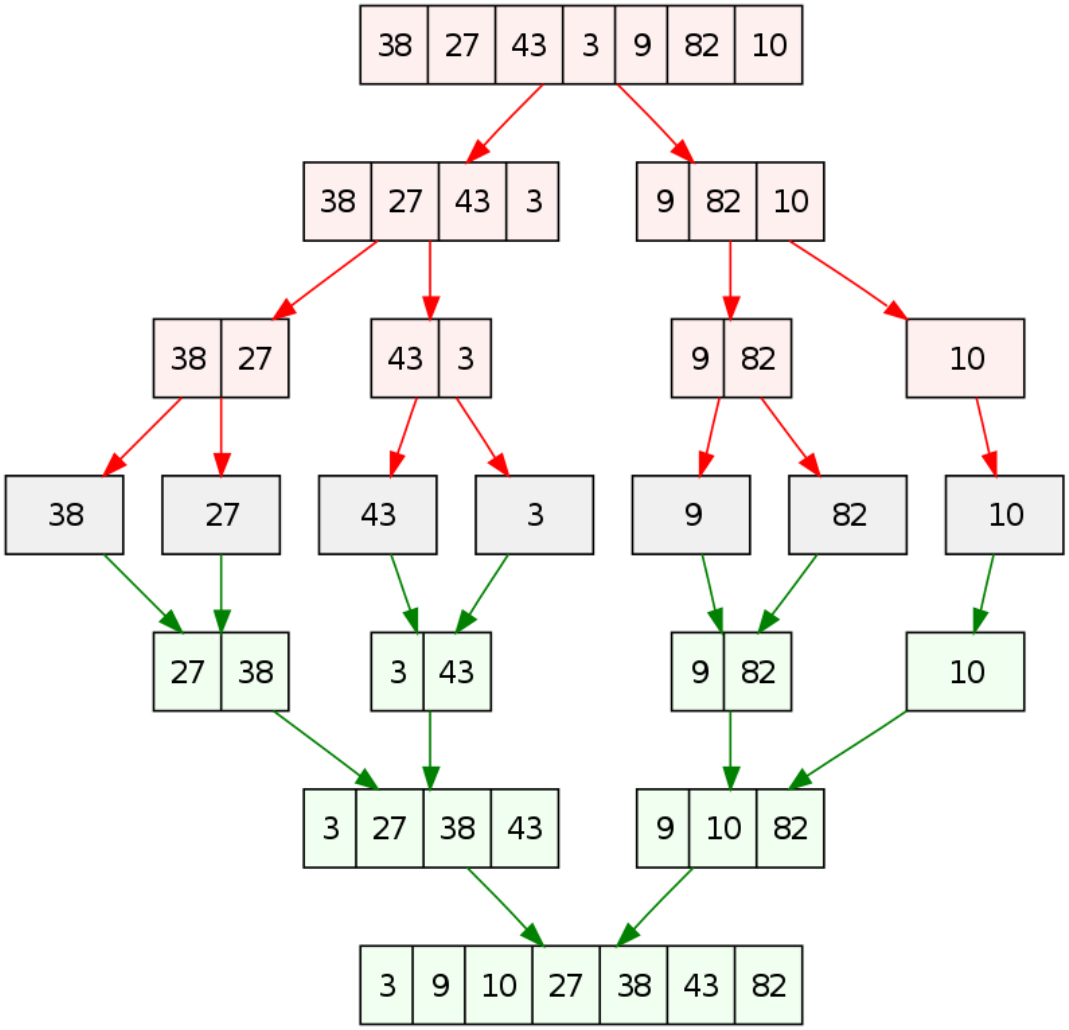
Two times of memory are used for additional threads.

Fork-Join Parallelism

- Multiple threads (tasks) are forked, and then joined.
  - Similar to the Divide-and-Conquer algorithm.



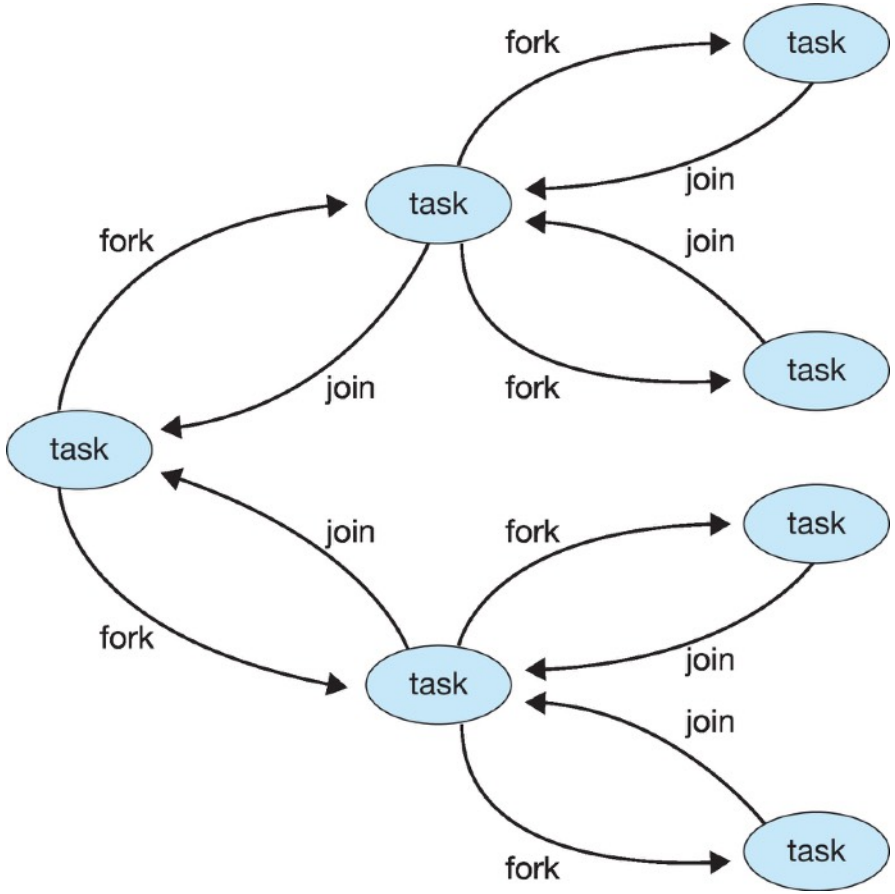
Sorting by Divide-and-Conquer



Fork-Join Parallelism

- General algorithm for fork-join strategy:

```
Task(problem)
   if problem is small enough
      solve the problem directly
   else
      subtask1 = fork(new Task(subset of problem)
      subtask2 = fork(new Task(subset of problem)

      result1 = join(subtask1)
      result2 = join(subtask2)

      return combined results
```



Fork-Join Parallelism

```
11 import java.util.concurrent.*;
12
13 public class SumTask extends RecursiveTask<Integer>
14 {
15     static final int SIZE = 10000;
16     static final int THRESHOLD = 1000;
17
18     private int begin;
19     private int end;
20     private int[] array;
21
22     public SumTask(int begin, int end, int[] array) {
23         this.begin = begin;
24         this.end = end;
25         this.array = array;
26     }
```

Fork-Join Parallelism

```
28      protected Integer compute() {
29          if (end - begin < THRESHOLD) {
30              // conquer stage
31              int sum = 0;
32              for (int i = begin; i <= end; i++)
33                  sum += array[i];
34
35              return sum;
36          }
37          else {
38              // divide stage
39              int mid = begin + (end - begin) / 2;
40
41              SumTask leftTask = new SumTask(begin, mid, array);
42              SumTask rightTask = new SumTask(mid + 1, end, array);
43
44              leftTask.fork();
45              rightTask.fork();
46
47              return rightTask.join() + leftTask.join();
48          }
49      }
50
```

Fork-Join Parallelism

```
51   public static void main(String[] args) {
52        ForkJoinPool pool = new ForkJoinPool();
53        int[] array = new int[SIZE];
54
55        // create SIZE random integers between 0 and 9
56        java.util.Random rand = new java.util.Random();
57
58        for (int i = 0; i < SIZE; i++) {
59             array[i] = rand.nextInt(10);
60        }
61
62        // use fork-join parallelism to sum the array
63        SumTask task = new SumTask(0, SIZE-1, array);
64
65        int sum = pool.invoke(task);
66
67        System.out.println("The sum is " + sum);
68   }
69 }
```

OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies parallel regions – blocks of code that can run in parallel
- #pragma omp parallel
- Create as many threads as there are cores

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  /* sequential code */

  #pragma omp parallel
  {
    printf("I am a parallel region.");
  }

  /* sequential code */

  return 0;
}
```

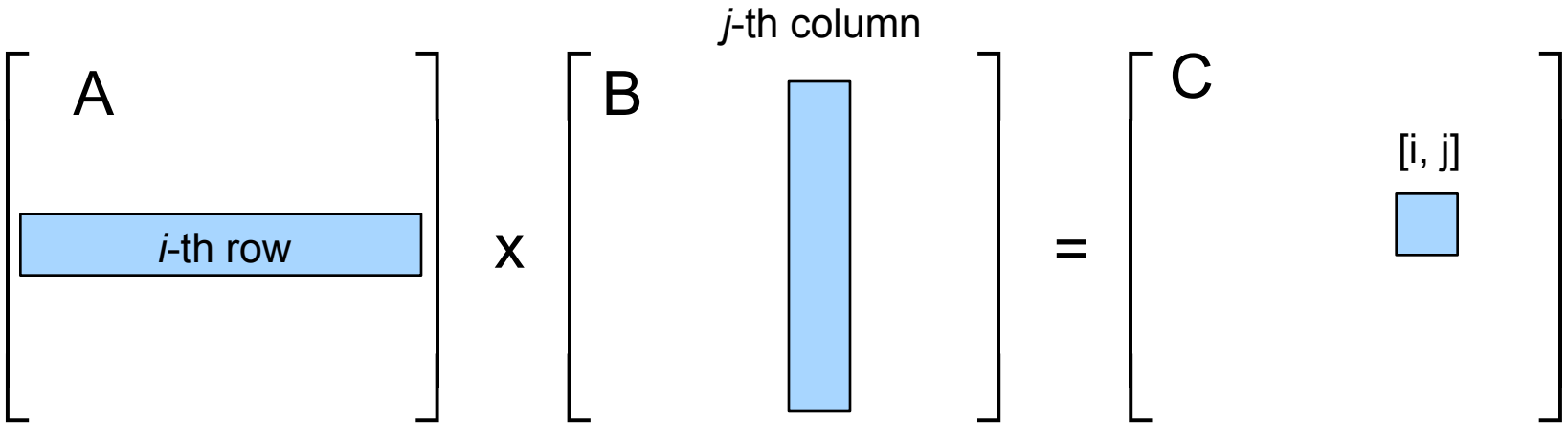
OpenMP construct

- All OpenMP constructs in C and C++ are indicated with a #pragma omp followed by parameters, ending in a newline. The pragma usually applies only into the statement immediately following it, except for the barrier and flush commands, which do not have associated statements.
- The parallel construct
  - The parallel construct starts a parallel block. It creates a team of N threads (where N is determined at runtime, usually from the number of CPU cores, but may be affected by a few things), all of which execute the next statement (or the next block, if the statement is a {…} -enclosure).
  - After the statement, the threads join back into one.
- The `for` construct splits the for-loop so that each thread in the current team handles a different portion of the loop.

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
   c[i] = a[i] + b[i];
}
```

Example of using OpenMP

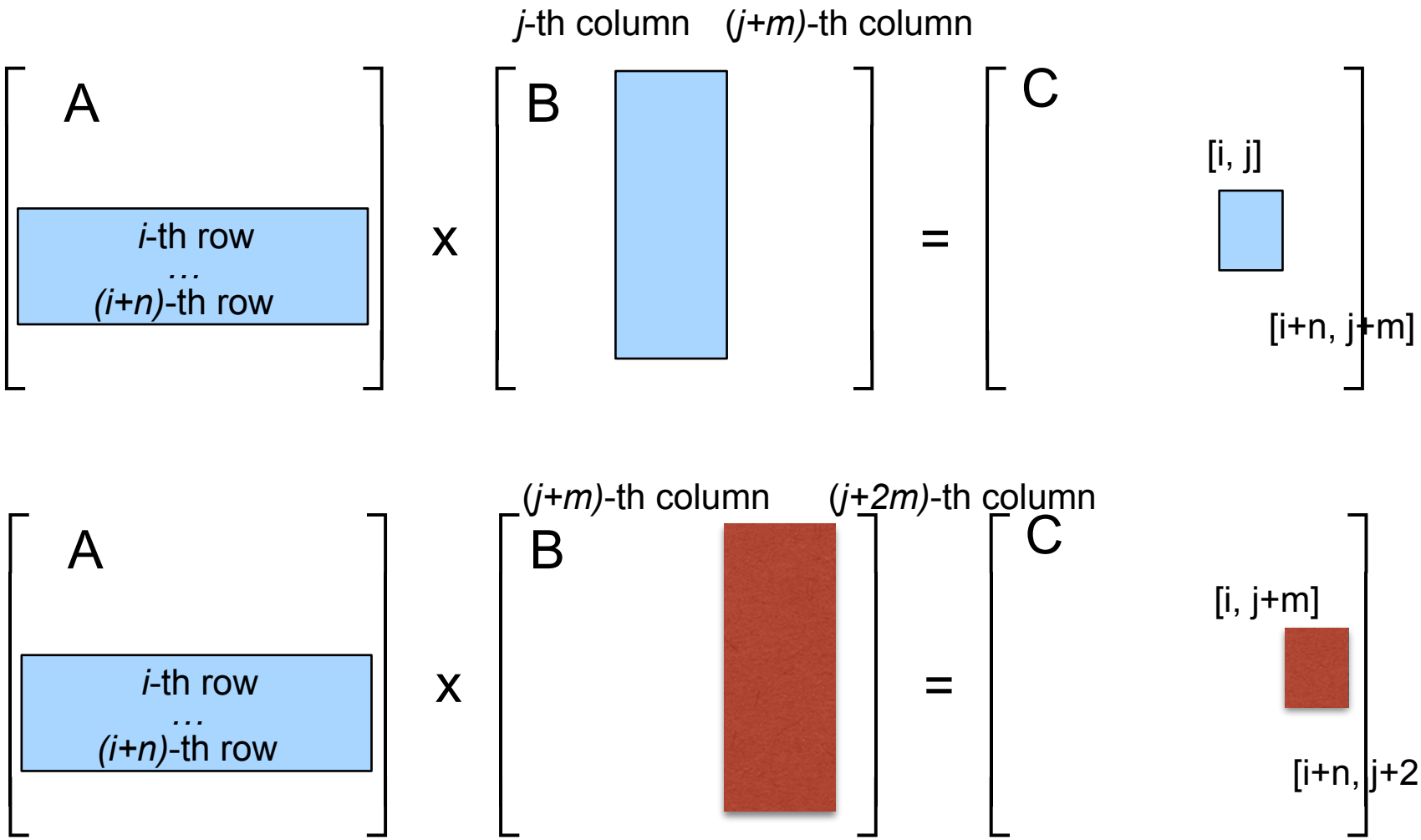
Matrix Multiplication




```
do i = 1, n
   do k = 1, l
    do j = 1, m
     c(i,k) = c(i,k) + a(i,j)*b(j,k)
    end do
  end do
end do
```

Serial version

Matrix Multiplication - Divide the work for parallel programming



Example: Matrix Multiplication Program

Generate matrix A in parallel

```
57 # pragma omp parallel shared ( a, b, c, n, pi, s ) private ( angle, i, j, k )
58 {
59   # pragma omp for
60   for ( i = 0; i < n; i++ )
61   {
62     for ( j = 0; j < n; j++ )
63     {
64       angle = 2.0 * pi * i * j / ( double ) n;
65       a[i][j] = s * ( sin ( angle ) + cos ( angle ) );
66     }
67   }
```

Generate matrix B in parallel

```
69 /*
70   Loop 2: Copy A into B.
71 */
72   # pragma omp for
73   for ( i = 0; i < n; i++ )
74   {
75     for ( j = 0; j < n; j++ )
76     {
77       b[i][j] = a[i][j];
78     }
79   }
80
81
```

Example: Matrix Multiplication Program

Compute C = A x B in parallel

```
81 /*
82   Loop 3: Compute C = A * B.
83 */
84   # pragma omp for
85   for ( i = 0; i < n; i++ )
86   {
87     for ( j = 0; j < n; j++ )
88     {
89       c[i][j] = 0.0;
90       for ( k = 0; k < n; k++ )
91       {
92         c[i][j] = c[i][j] + a[i][k] * b[k][j];
93       }
94     }
95   }
96
```

Results

Number of threads: 1

```
MXM_OPENMP:
  C/OpenMP version
  Compute matrix product C = A * B.

  The number of processors available = 8
  The number of threads available    = 1
  The matrix order N                 = 500
  Elapsed seconds = 0.754167
  C(100,100)  = 1

MXM_OPENMP:
  Normal end of execution.
```

Number of threads: 64

```
MXM_OPENMP:
  C/OpenMP version
  Compute matrix product C = A * B.

  The number of processors available = 8
  The number of threads available    = 64
  The matrix order N                 = 500
  Elapsed seconds = 0.228819
  C(100,100)  = 1

MXM_OPENMP:
  Normal end of execution.
```

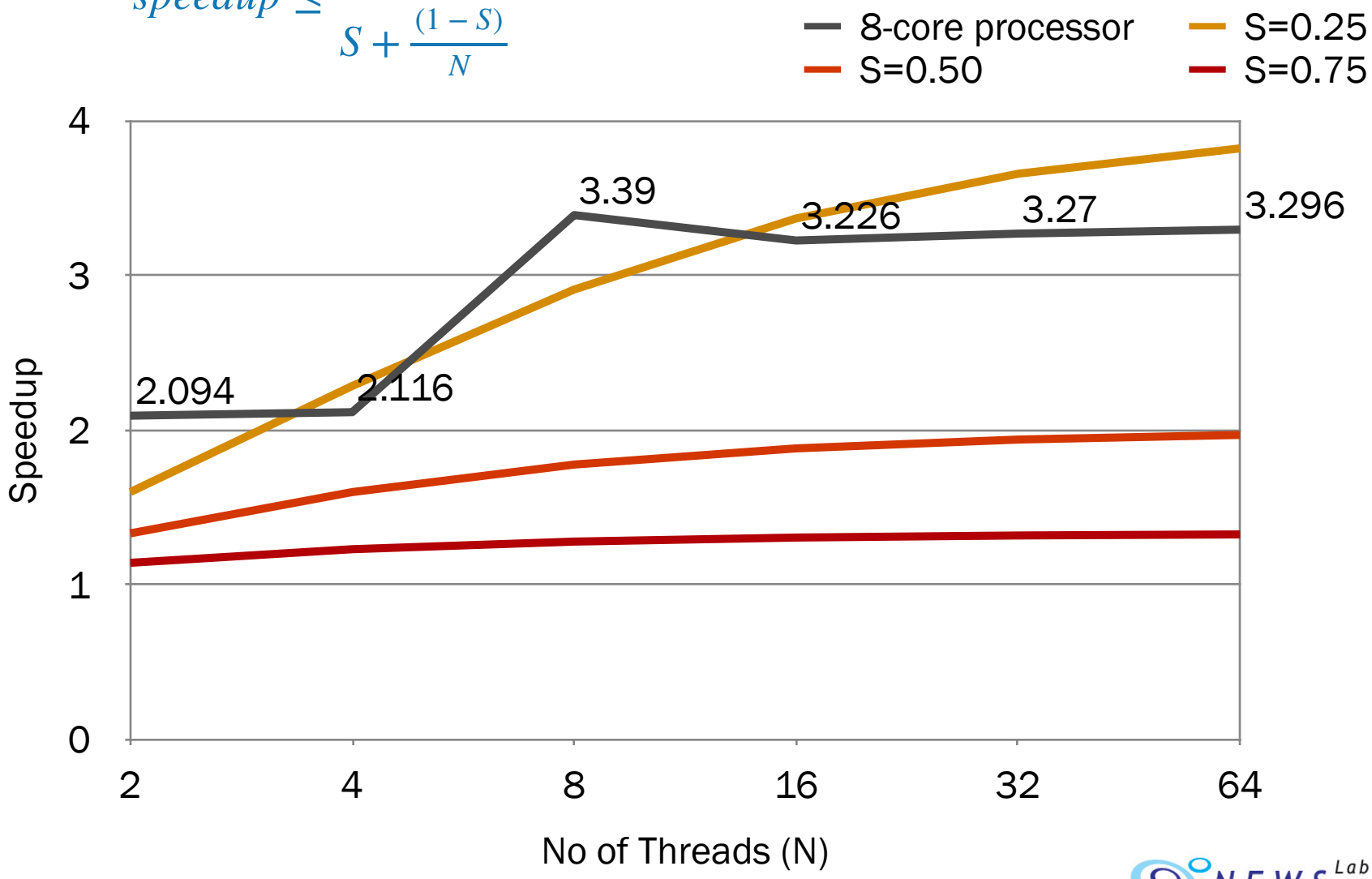
Recall the speed-up

	1.0000	2.0000	4.0000	8.0000	16.0000	32.0000	64.0000
Elapsed Time	0.7542	0.3602	0.3564	0.2225	0.2338	0.2306	0.2288
Speed-Up	S=?	2.0936	2.1163	3.3898	3.2257	3.2702	3.2959
Speed-Up	0.2500	1.6000	2.2857	2.9091	3.3684	3.6571	3.8209
Speed-Up	0.5000	1.3333	1.6000	1.7778	1.8824	1.9394	1.9692
Speed-Up	0.7500	1.1429	1.2308	1.2800	1.3061	1.3196	1.3264
Speed-Up	0.0100	1.9802	3.8835	7.4766	13.9130	24.4275	39.2638

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

Recall the speed-up

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

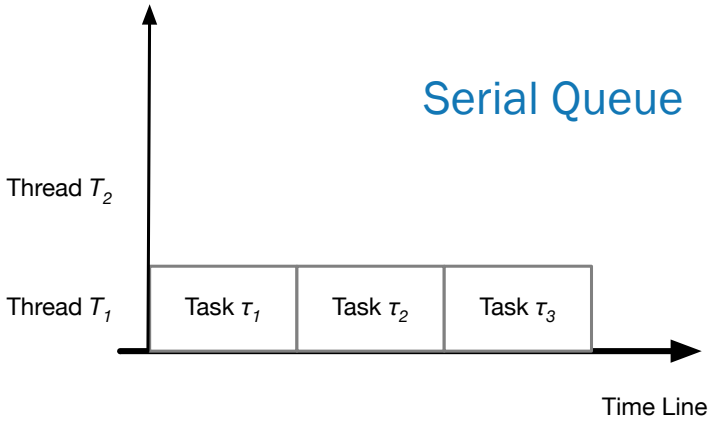


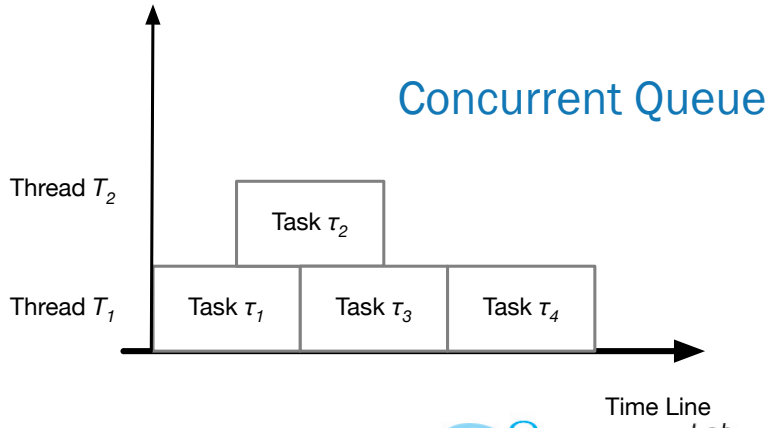
Grand Central Dispatch (GCD)

- Apple technology for macOS and iOS operating systems
- GCD's task parallelism is based on the thread pool pattern. Multiple threads are always available — waiting for tasks to be executed concurrently.
- Dispatch queues manage tasks in FIFO order.

```
class DispatchQueue
```

- There are two types of dispatch queues:
  - Serial Queue: executed on main thread
  - Concurrent Queue: executed on background thread





Grand Central Dispatch

- Two types of dispatch queues:
  - serial – blocks removed in FIFO order, queue is per process, called main queue
    - Programmers can create additional serial queues within program
  - concurrent – removed in FIFO order but several may be removed at a time
    - Four system wide queues divided by quality of service:
      - `QOS_CLASS_USER_INTERACTIVE`
      - `QOS_CLASS_USER_INITIATED`
      - `QOS_CLASS_USER_UTILITY`
      - `QOS_CLASS_USER_BACKGROUND`

C

- Manages most of the details of threading
- Block is in “^{ }” :

```
^{ printf("Hello %s!\n", argv[i]); });
```

- Blocks placed in dispatch queue
  - Assigned to available thread in thread pool when removed from queue
- For the Swift language a task is defined as a closure – similar to a block, minus the caret
- Closures are submitted to the queue using the `dispatch_async()` function:

```
14      /*
15       * Get the main serial queue.
16       * It doesn't start processing until we call dispatch_main()
17       */
18      dispatch_queue_t main_q = dispatch_get_main_queue();
19
20      for (i = 1; i < argc; i++) {
21          /* Add some work to the main queue. */
22          dispatch_async(main_q, ^{ printf("Hello %s!\n", argv[i]); });
23      }
```

Using Main thread for Serial GCD

```
 9 int
10 main(int argc, char *argv[])
11 {
12     int i;
13
14     /*
15      * Get the main serial queue.
16      * It doesn't start processing until we call dispatch_main()
17      */
18     dispatch_queue_t main_q = dispatch_get_main_queue();
19
20     if ((argc > 1) && (atoi(argv[1]) > 2)) {
21       for (i = 1; i < atoi(argv[1]); i++) {
22        /* Add some work to the queue. */
23        dispatch_async(main_q, ^{ printf("Hello object %d!\n", i); sleep(1); });
24   }
25     }
26
27     /* Add a last item to the main queue. */
28     dispatch_async(main_q, ^{
29       printf("Goodbye!\n");
30       exit(0); });
31
32     /* Start the main queue */
33     dispatch_main();
34
35     /* NOTREACHED */
36     return 0;
37 }
```

Using New thread for Serial GCD

```
9  int main(int argc, char *argv[]) {
10      int i;
11
12      /* Create a serial queue. */
13      dispatch_queue_t greeter = dispatch_queue_create("Greeter", DISPATCH_QUEUE_SERIAL);
14
15      if ((argc > 1) && (atoi(argv[1]) > 2)) {
16        for (i = 1; i < atoi(argv[1]); i++) {
17            /* Add some work to the queue. */
18            dispatch_async(greeter, ^{ printf("Hello object %d!\n", i); sleep(1); });
19        }
20      }
21
22      /* Add a last item to the queue. */
23      dispatch_sync(greeter, ^{
24        printf("Goodbye!\n");
25        exit(0); });
26
27      return 0;
28 }
```

Using New thread for Concurrent GCD

```
10 int main(int argc, char *argv[]) {
11     int i;
12
13     /* Create a serial queue. */
14     dispatch_queue_t greeter = dispatch_queue_create("Greeter", DISPATCH_QUEUE_CONCURRENT);
15
16     /*
17      * Get the main serial queue.
18      * It doesn't start processing until we call dispatch_main()
19      */
20     dispatch_queue_t main_q = dispatch_get_main_queue();
21
22     if ((argc > 1) && (atoi(argv[1]) > 2)) {
23         for (i = 1; i < atoi(argv[1]); i++) {
24             /* Add some work to the queue. */
25             dispatch_async(greeter, ^{ printf("Hello object %d!\n", i); usleep(i * 10); });
26         }
27     }
28
29     /* Add a last item to the main queue. */
30     dispatch_async(main_q, ^{
31       printf("Goodbye!\n");
32       exit(0); });
33
34     /* Start the main queue */
35     dispatch_main();
36
37     return 0;
38 }
```

Threading Issues

Threading Issues

- Semantics of fork() and exec() system calls
- Signal handling: Synchronous and asynchronous
- Thread cancellation of target thread: Asynchronous or deferred
- Thread-local storage
- Scheduler Activations

Semantics of fork() and exec()

- Does fork() duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork:
    - one that duplicates all threads and
    - another that duplicates only the thread that invoked the fork() system call.
  - exec() usually works as normal – replace the running process including all threads.

Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred.
- A signal handler is used to process signals
  - Signal is generated by particular event
  - Signal is delivered to a process
  - Signal is handled by one of two signal handlers: **default** handler or **user-defined**
- Every signal has default handler that kernel runs when handling signal
  - User-defined signal handler can override default
  - For single-threaded, signal delivered to process

Signal Handling (Cont.)

- Where should a signal be delivered for multi-threaded?
  - Deliver the signal to the thread to which the signal applies

    pthread kill(pthread_t tid, int signal);

  - Deliver the signal to **every** thread in the process
  - Deliver the signal to **certain** threads in the process: each thread can choose to block certain signals.
  - Assign a **specific** thread to receive all signals for the process

Without specified assigned, because signals need to be handled only once, a signal is typically delivered only to the first thread found that is not blocking it.

Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is target thread
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

  . . .

/* cancel the thread */
pthread_cancel(tid);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state.
- Pthreads supports **three** cancellation modes.

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

pthread_setcancelstate changes the cancellation state for the calling thread -- that is, whether cancellation requests are ignored or not.

pthread_setcanceltype changes the type of responses to cancellation requests for the calling thread: asynchronous (immediate) or deferred.

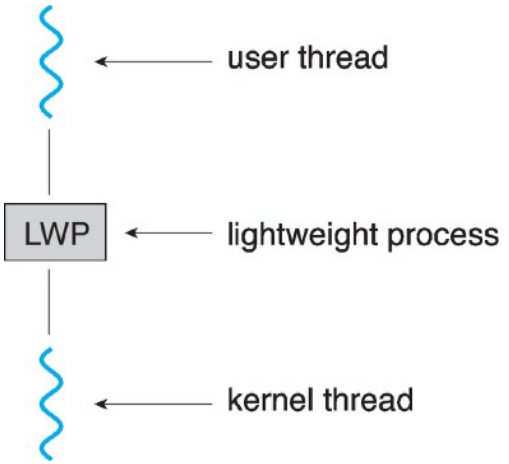
- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches cancellation point
    - i.e., pthread_testcancel()
    - Then cleanup handler is invoked
- On Linux systems, thread cancellation is handled through signals.

Thread-Local Storage

- Thread-local storage (TLS) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to static data
  - TLS is unique to each thread

Scheduler Activations

- Both M:N and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – lightweight process (LWP)
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP attached to kernel thread
- How many LWPs to create?
  - Scheduler activations provide upcalls - a communication mechanism from the kernel to the upcall handler in the thread library.
  - This communication allows an application to maintain the correct number kernel threads.



Any Questions?

See You
Next Class

End