Λογισμικό και Προγραμματισμός Συστημάτων Υψηλής Επίδοσης

«Multithreading – POSIX threads»

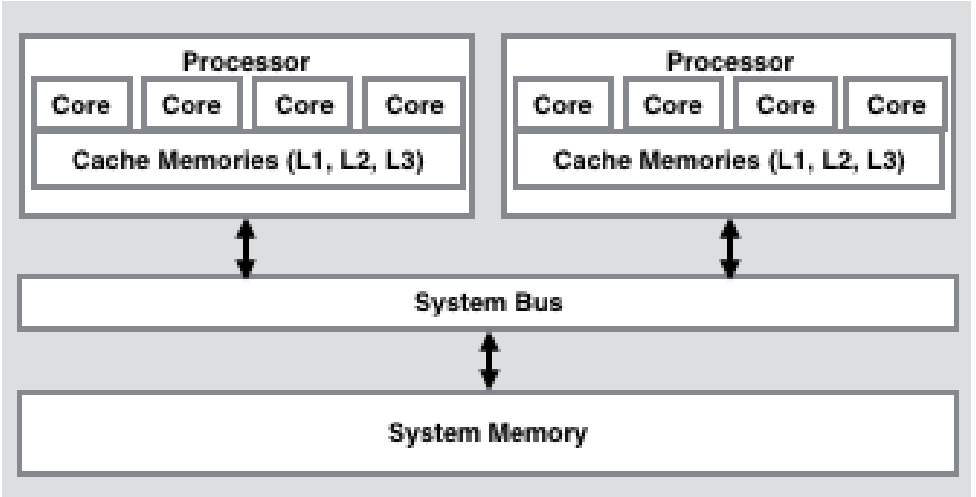
Παναγιώτης Χατζηδούκας

Outline

- Processes and Threads
- POSIX Threads API
  - Thread management
  - Synchronization with mutexes
- Deadlock and thread safety

Terminology

- Parallelism in Hardware:
  - multiple cores and memory
- Parallelism in Software:
  - **process**: execution sequence within the OS, a running program
  - **thread**: can execution sequence within a process, all threads of the same process share the application data (memory)



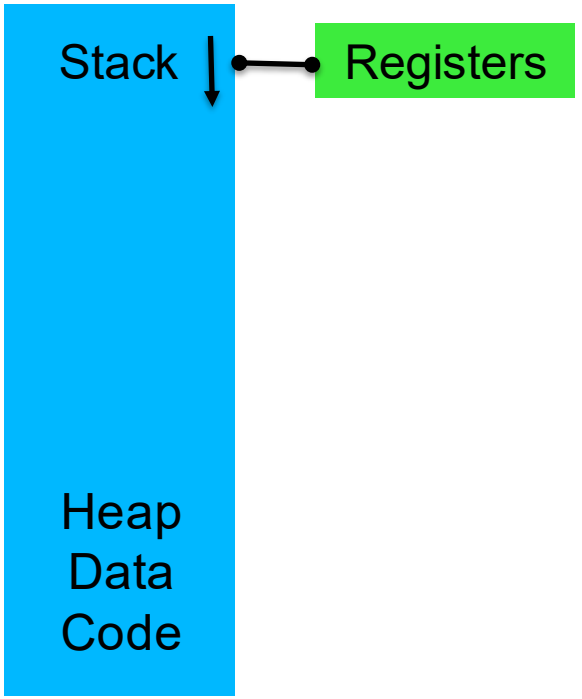

```
int a[1000];

int main( int argc, char** argv )
{
  for(int i =   0; i <  500; i++ ) a[i] = 1;
  for(int i = 500; i < 1000; i++ ) a[i] = 2;

  return 0;
}
```

Processes

- A process consists of the following:
  - Address space: text segment (code), data segment, heap and stack
  - Information maintained by the operating system (process state, priority, resources, statistics)
- Process state: snapshot where the above information has specific values
  - Memory state: state of the address space
  - Processor state: register values



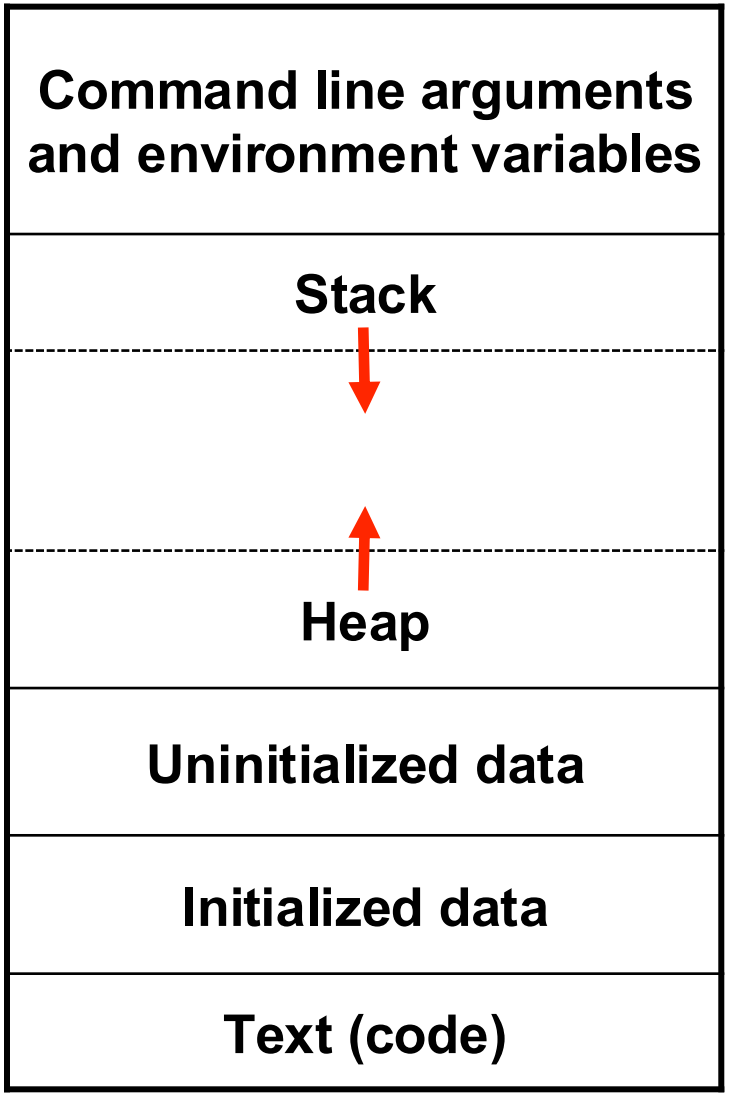
Process Switching

- Before execution, the processor state of a process must be loaded first to the specific processor
- During execution, the processor state of the process changes
- Context switching: a running process stops and another one starts (or resumes)
  - The processor state of the current process is stored
  - The processor state of the next process is loaded

Process Memory

- Each process has its own (private) memory space
  - A process cannot access the memory of another process
  - This provides basic safety in a multi-user environment
- The operating system has full access to the memory of all processes
- Communication between processes is important
  - When they cooperate to solve a single problem
- Operating systems implement several mechanisms for interprocess communication
  - signals, files, pipes, sockets
  - shared memory

Process Memory Layout



growth directions

Memory Organization (C/C++)

- Text segment
  - Instruction executed by the processor
  - Can be shared between multiple processes
  - Read-only segment
- Initialized data segment
  - Global variables with initial value:

```
double Pi = 3.1415;
static char message[] = "hello world!";
```

Memory Organization (C/C++)

- Uninitialized data segment
  - Global variables without initial value

```
int result;
double Matrix[512][512];
```

  - The operating system initializes these variables to zero before the execution of the program
- Stack
  - Local variables, function parameters, returned value
- Heap
  - Dynamic memory management (`malloc`, `calloc`, `new`, …)

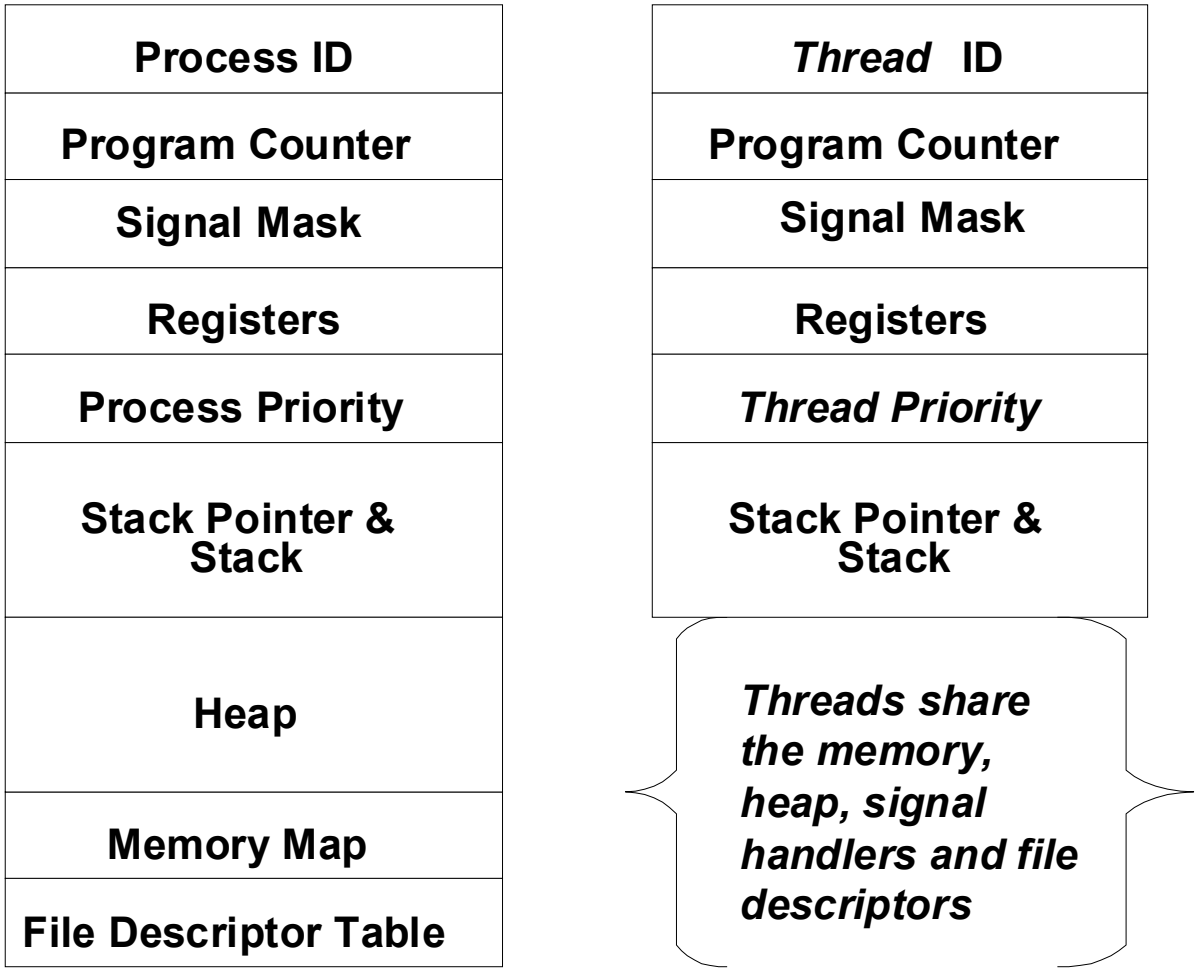
Threads

- Thread: an independent stream of instructions that can be scheduled to run as such by the operating system
  - execution sequence within the process
- A process can create multiple threads
  - each thread executes a specific user-defined function
  - main() is the first (primary) thread
- Threads
  - share the memory space of the process they belong to
  - have their own state and some private memory (stack)
  - are cheap to create but difficult to use correctly
  - can run on different processors

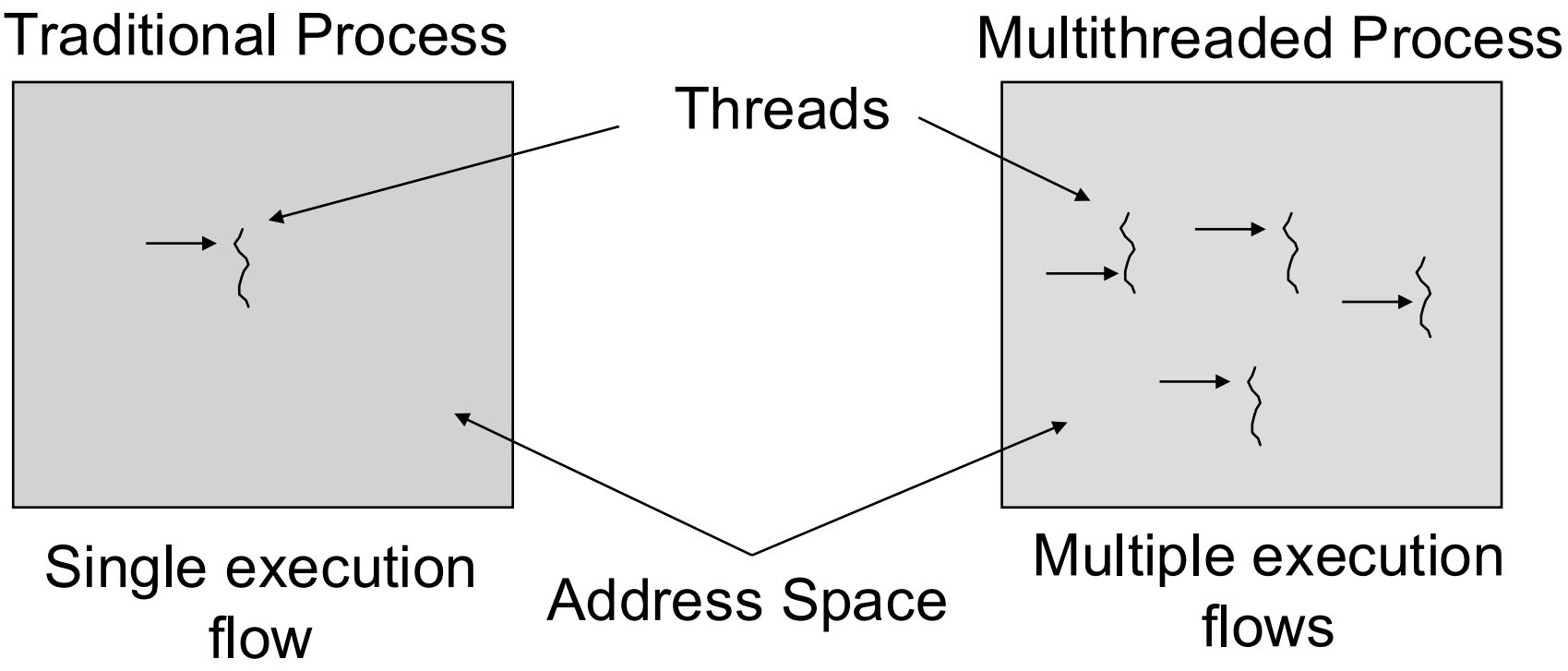
Threads

- Threads: also known as lightweight processes
  - Process: memory, instructions, program counter, stack pointer, registers, file descriptors, …
  - Thread: program counter, stack pointer and stack, registers
- The threads of a process share:
  - Program instructions, most data, open files, signal handlers, current working directory, user and group id
- The threads of a process do not share:
  - Thread id, registers (program counter, stack pointer), stack, errno, signal mask, priority
- POSIX Threads (pthreads): Application Programming Interface (API) defined by the IEEE POSIX.1c standard

Διεργασίες και Νήματα

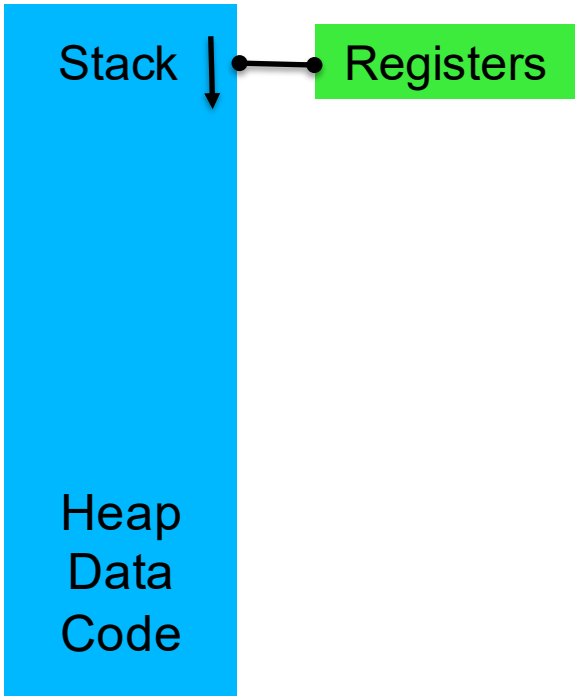


Processes and Threads



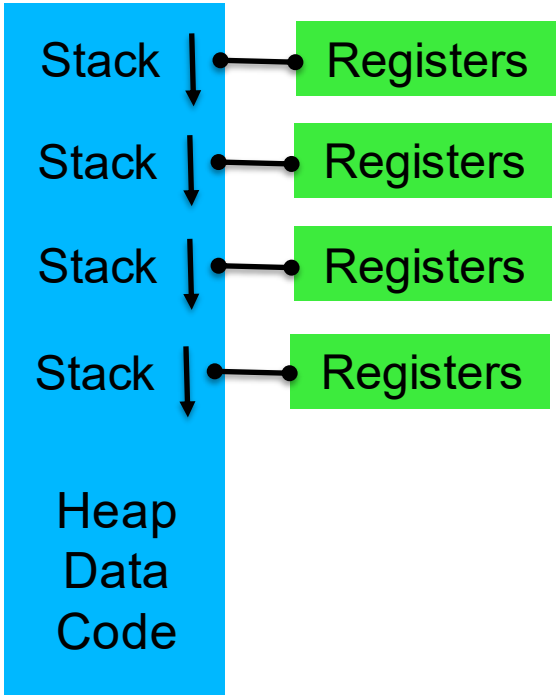
Processes and Threads

Traditional Process



Single instruction execution flow

Multithreaded Process



Multiple instruction execution flows

Processes and Threads

- Advantages/disadvantages of multithreading

  [+] Lower creation and management overhead

  [+] Simpler and cheaper communication between threads than processes

  [-] Error-prone programming

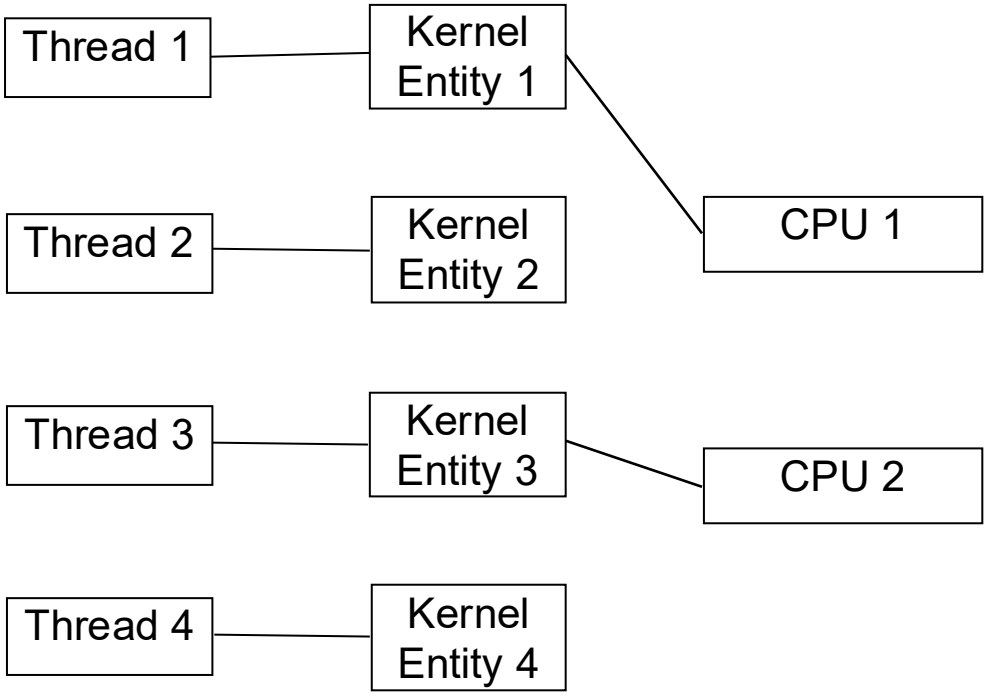
- Thread implementations
  - user level threads
  - system/kernel level threads
- It depends on whether the operating system is aware of the existence of application threads or not

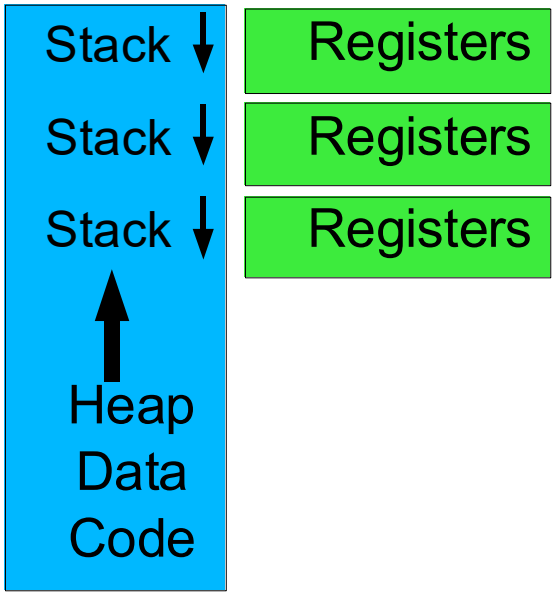
Kernel-level threads

- Implemented as the OS level
  - Each thread is a lightweight process
  - Thread management is based on system calls
- Scheduled by the OS, similarly to processes
  - Straightforward parallel execution of threads on multicore hardware
  - If a threads blocks (at a system call), the rest of the threads (of the same process) continue their execution
- Practically all POSIX Threads implementations follow the specific model

Kernel-level threads

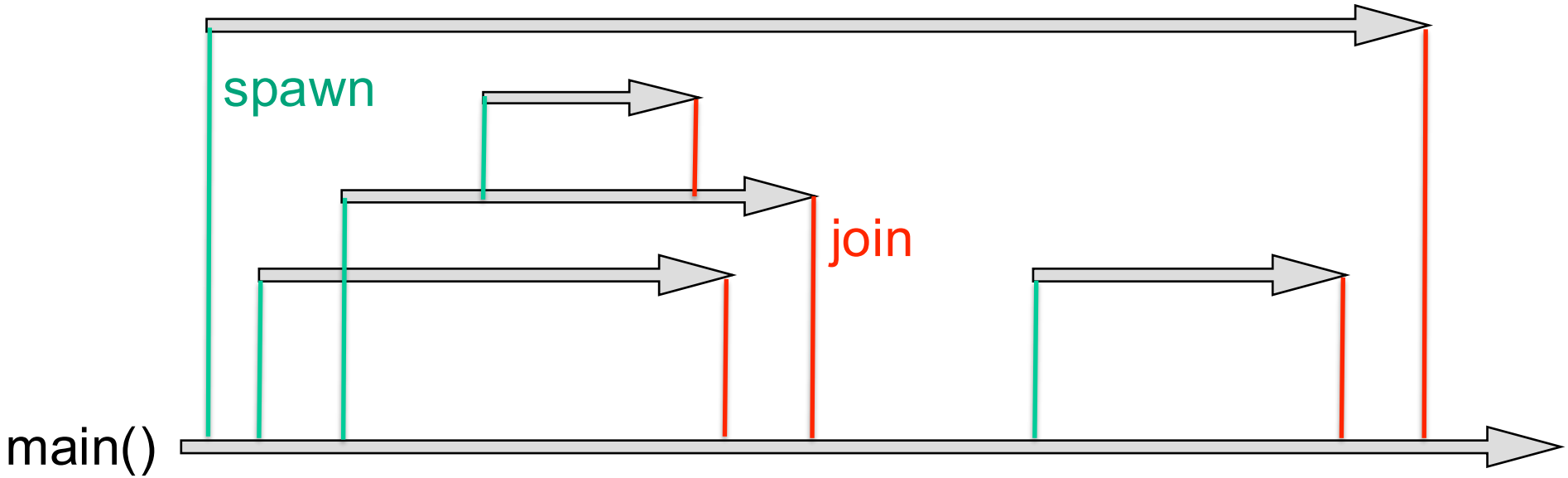
1:1 (one-to-one) model



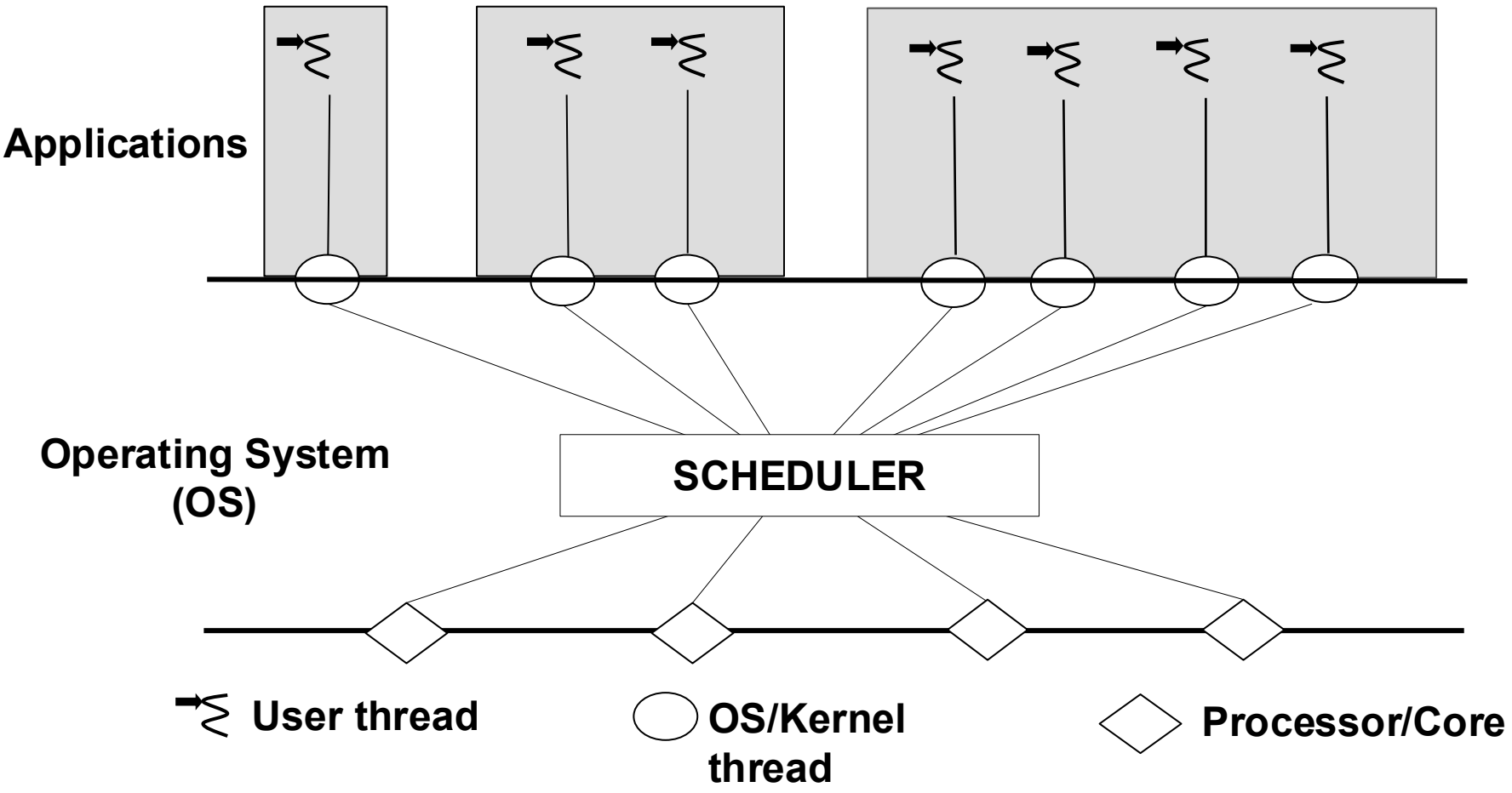


Spawning and Joining Threads

- During the execution of a multithreaded program threads get spawned and joined dynamically



General View



POSIX Threads (Pthreads)

- Standardized C language threads programming interface
  - http://pubs.opengroup.org/onlinepubs/9699919799/

- Header file:

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

- Compilation

```
$ gcc -pthread -o hello hello.c
```

- Execution

```
$ ./hello
```

Processes vs POSIX Threads

	Processes	**POSIX Threads**
Identifier	pid_t getpid()	pthread_t pthread_self()
Creation	fork()	pthread_create()
Start	exec()	
Wait	wait()	pthread_join()
Exit	exit()	pthread_exit()

Skeleton

```
void *func(void *arg)
{
   /* define local data */
   - - - - - - - - - - - -
   - - - - - - - - - - - -
   - - - - - - - - - - - -
   return (void *)&result;
}

main()
{
   pthread_t tid;
   int exit_value;
   - - - - - - - - - - - -
   pthread_create (&tid, NULL, func, NULL);
   - - - - - - - - - - - -
   pthread_join (tid, &exit_value);
   - - - - - - - - - - - -
}
```

```
/* function code */
```

equivalently:

```
pthread_exit(&result);
```

Thread Creation

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

- `thread`: unique identifier for the new thread returned by the subroutine
- `attr`: used to set thread attributes. If NULL, the default values are used.
- `routine`: the C routine that the thread will execute once it is created.
- `arg`: single argument that may be passed to start_routine. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.
- if there are no errors, it returns 0

pthread_create

```
#include <pthread.h>

pthread_t tid;
extern void *func(void *arg);
void *arg;

int res = pthread_create(&tid, NULL, func, arg);
```

Passing Multiple Arguments

```
struct data {
  int i;
  float f;
};

void *routine(void *arg) {
  struct data *d = (struct data *) arg;
  int local_i = d->i;
  d->f = 5.0;
  return NULL;
}

int main() {
  pthread_t tid;
  struct data main_data;
  main_data.i = 6;

  pthread_create(&tid, NULL, routine, (void *) &main_data);
  //...
}
```

Thread Joining

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

- `pthread_join()` blocks the calling thread until the specified thread terminates
- The value returned by the thread function is stored in the memory location specified by `status`
- if there are no errors, it returns 0

pthread_join

```
#include <pthread.h>

pthread_t tid;
int result;


pthread_join(tid, (void *)&result);

pthread_join(tid, NULL);
```

Hello World

```
void *work(void *arg)
{
  pthread_t me = pthread_self();
  printf("Hello world from thread %ld!\n", (long)me);
  return NULL;
}

int main(int argc, char **argv)
{
  long i = 1;
  pthread_t thread;

  printf("main thread %ld!\n", (long)pthread_self());

  pthread_create(&thread, NULL, work, (void *)i);
  pthread_join(thread, NULL);

  printf("Child ended, exiting\n");
  return 0;
}
```

ID of calling thread

Spawning and Joining Threads

```
void *func(void *arg)
{
   sleep(1);
   return NULL;
}

int main(int argc, char * argv[])
{
   pthread_t id[4];

   for (long i = 0; i < 4; i++)    {
       pthread_create(&id[i], NULL, func, NULL);
   }

   for (long i = 0; i < 4; i++)    {
       pthread_join(id[i], NULL);
    }

   return 0;
}
```

Creating and Joining Threads

```
void * func(void * arg)
{
   long sec = (long) arg + 1;
   sleep((long) sec);
   return arg;     /* pthread_exit(arg); */
}

int main(int argc, char * argv[])
{
   pthread_t id[4];
   long result;

   for (long i = 0; i < 4; i++)    {
       pthread_create(&id[i], NULL, func, (void *) i);
   }

   for (long i = 0; i < 4; i++)    {
       pthread_join(id[i], (void *) &result);
       /* result == i */;
   }

   return 0;
}
```

fix: *((long *) arg) +1;

what if we pass &i and also apply the above fix

Creating and Joining Threads

```
long x=5;

void *func(void * arg)
{
   long sec = *((long *)arg);
   sleep(x+sec);
   return NULL;
}

int main(int argc, char * argv[])
{
   pthread_t id[4];
   long a[4];

   for (long i = 0; i < 4; i++) {
       a[i] = i;
       pthread_create(&id[i], NULL, func, (void *) &a[i]);
   }

   for (long i = 0; i < 4; i++) {
       pthread_join(id[i], NULL);
   }

   return 0;
}
```

Synchronization

- Consider the following code
  - `next_ticket` is a global variable initialized to 0
  - `ticket` is a local variable, private to each thread

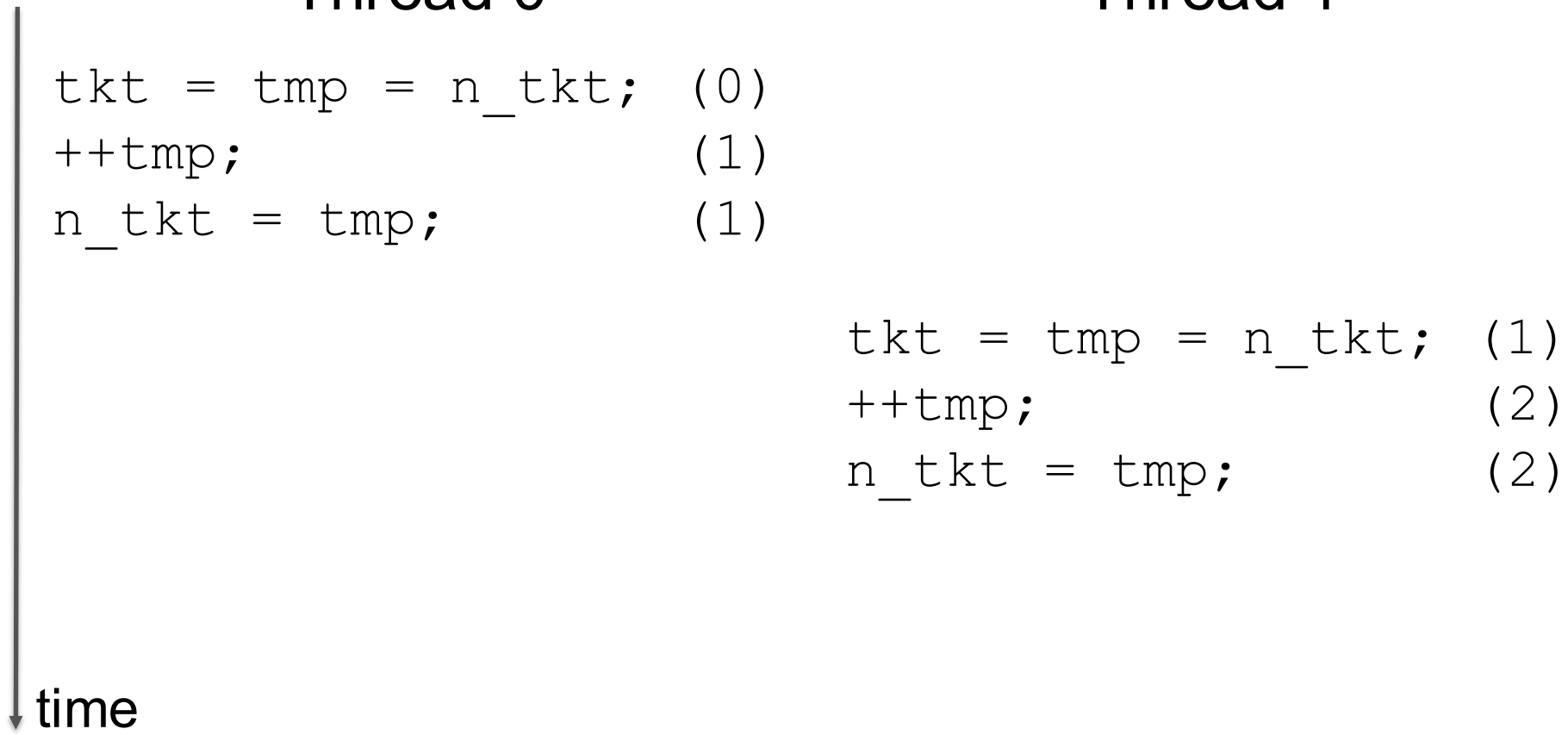
```
ticket = next_ticket++;          /* 0 ⇒ 1 */
```

- In the general case, this is equivalent to the following:

```
ticket = temp = next_ticket; /* 0 */
++temp;                      /* 1 */
next_ticket = temp;          /* 1 */
```

```
R1 ← next_ticket (load)
ticket ← R1      (store)
R1 ← R1+1        (inc)
next_ticket ← R1 (store)
```

Execution with 2 Threads



Another Possible Case

Thread 0		Thread 1	
`tkt = tmp = n_tkt;`	(0)		
		`tkt = tmp = n_tkt;`	(0)
`++tmp;`	(1)		
		`++tmp;`	(1)
`n_tkt = tmp;`	(1)		
		`n_tkt = tmp;`	(1)

time

What we observe here is a race condition
The update of the shared variable is a critical section and must be protected

Synchronization - Mutexes

- POSIX Threads include several synchronization mechanisms
  - Mutexes, Condition variables, Barriers, Reader-Writer locks, Spinlocks, Semaphores
- A Mutex (*Mut*ual *Ex*clusion) is a mechanism that allows multiple threads to synchronize their access to shared resources (e.g. variables)
  - A mutex has two states: *locked* and *unlocked*
  - Only one thread can lock the mutex
  - Once a mutex is locked, any other thread that tries to lock that mutex will suspend its execution, until the thread unlocks the mutex
  - At that point, one of the waiting threads acquires the mutex and continues its execution

Mutex Management

- Declaration and static initialization of a mutex:

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

- Declaration and dynamic initialization of a mutex:

```
pthread_mutex_t mymutex;
pthread_mutex_init(&mymutex, NULL);
```

- Locking (acquiring) the mutex:

```
pthread_mutex_lock(&mymutex);
```

- Unlocking (releasing) the mutex after the critical section:

```
pthread_mutex_unlock(&mymutex);
```

Dynamic Allocation of Mutex

- Pointer to mutex

```
pthread_mutex_t *mymutex;
```

- Memory allocation:

```
mymutex = (pthread_mutex_t *)
          malloc(sizeof(pthread_mutex_t));
```

- Mutex intialization:

```
pthread_mutex_init(mymutex, NULL);
```

- Locking and unlocking as before
- Destroying mutex:

```
pthread_mutex_destroy(mymutex);
```

Usage

```
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
double global_sum = 0.0;

void *work(void *arg)
{
   ...
  pthread_mutex_lock(&mutex);
  global_sum += local_sum;
  pthread_mutex_unlock(&mutex);
   ...
}
```

Check and Lock

```
int pthread_mutex_trylock(pthread_mutex_t *m);
```

- Allows a thread to try to lock a mutex
- If the mutex is available then the thread locks the mutex
- If the mutex is locked then the function informs the user by returning a special value (EBUSY)
- This approach allows for implementations of spinlocks

```
while (pthread_mutex_trylock(&mutex) == EBUSY)
    /*sched_yield()*/;
```

Mutexes - Recap

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

pthread_mutex_t static_mutex = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_t *dynamic_mutex;

dynamic_mutex = (pthread_mutex_t *)
                malloc(sizeof(pthread_mutex_t));
pthread_mutex_init(&dynamic_mutex, NULL);



pthread_mutex_destroy(&dynamic_mutex);
free(dynamic_mutex);
```

Example: Thread Number

```
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int temp_id = 0;

void *work(void *arg)
{
  int id;
  ...
  pthread_mutex_lock(&mutex);
  id = temp_id;
  temp_id = temp_id+1;
  pthread_mutex_unlock(&mutex);
  ...
}
```

Example: try lock

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

void * func(void *arg) {
   int res;
   res = pthread_mutex_trylock(&mutex1); /* EBUSY */;
   return 0;
}

int main() {
   pthread_t t;
   pthread_mutex_lock(&mutex1);
   pthread_create(&t, NULL, func, NULL);
   sleep(2);
   pthread_mutex_unlock(&mutex1);
   pthread_join(t, NULL);
   return 0;
}
```

Compute Pi - Sequential Version

```
long num_steps = 100000;
double step;

int main()
{
  double x, pi, sum = 0.0;

  step = 1.0/(double) num_steps;

  for (int i=0; i <num_steps; i++){
    x = (i-0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }

  pi = step * sum;

  return 0;
}
```

POSIX Threads Version

```
#define NUM_THREADS 2
pthread_t thread[NUM_THREADS];
pthread_mutex_t Mutex;
long num_steps = 100000;
double step;
double global_sum = 0.0;

void *Pi (void *arg) {
  long start;
  double x, sum = 0.0;

  start = (long) (*(int *) arg);
  step = 1.0/(double) num_steps;

  for(long i=start; i<num_steps;
           i+=NUM_THREADS) {
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }
  pthread_mutex_lock (&Mutex);
  global_sum += sum;
  pthread_mutex_unlock(&Mutex);

  return 0;
}
```

```
int main()
{
  double pi;

  int Arg[NUM_THREADS];

  for(int i=0; i<NUM_THREADS; i++)
    Arg[i] = i;

  pthread_mutex_init(&Mutex, NULL);

  for (int i=0; i<NUM_THREADS; i++)
    pthread_create(&thread[i], NULL,
                   Pi, &Arg[i]);

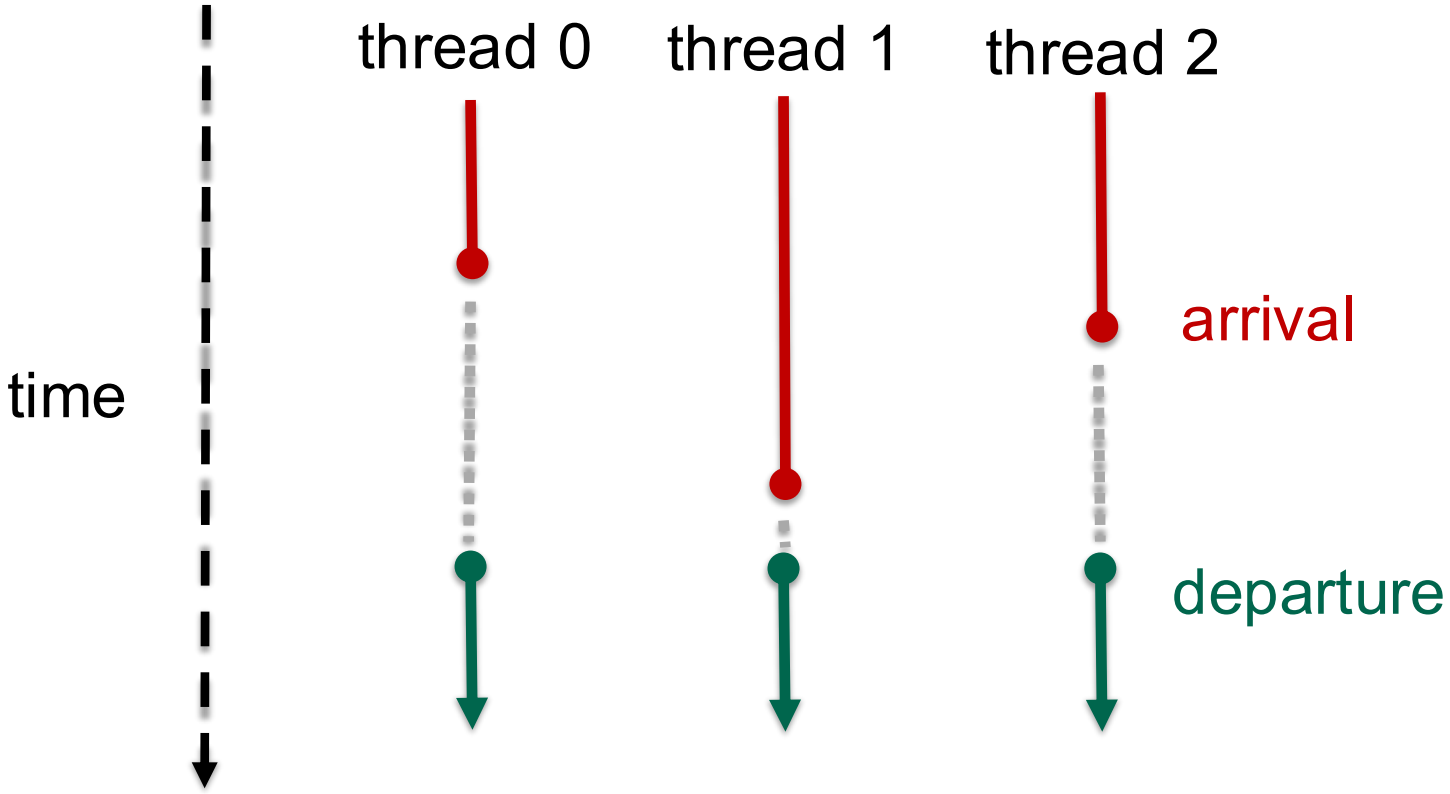
  for (int i=0; i<NUM_THREADS; i++)
    pthread_join(thread[i], NULL);

  pi = global_sum * step;

  return 0;
}
```

Barriers

- Barrier: synchronization mechanism that make sures all calling threads in a program
- No thread can cross the barrier until all the threads have reached it



Barriers

```
int pthread_barrier_init(pthread_barrier_t * bar,
                         const pthread_barrierattr_t*attr,
                         usigned int count);
int pthread_barrier_wait(pthread_barrier_t *bar);
int pthread_barrier_destroy(pthread_barrier_t *bar);
```

- pthread_barrier_t: data type
- count: number of threads that must call the barrier

Barrier Example

```
pthread_barrier_t bar;

void *func(void *arg)
{
    int id = *((int *)arg);
    printf("Hello world from thread %d!\n", id);

    pthread_barrier_wait(&bar);
    sleep(id);

    printf("Goodbye world from thread %d!\n", id);

    return NULL;
}

int main(int argc, char * argv[])
{
    ...
    pthread_barrier_init(&bar, NULL, 4);
    ...
```