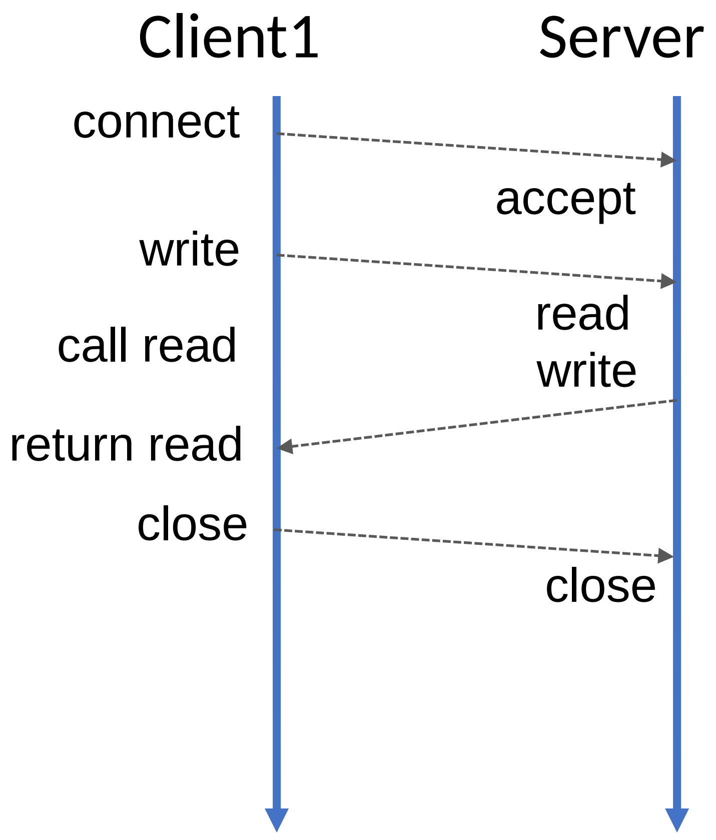
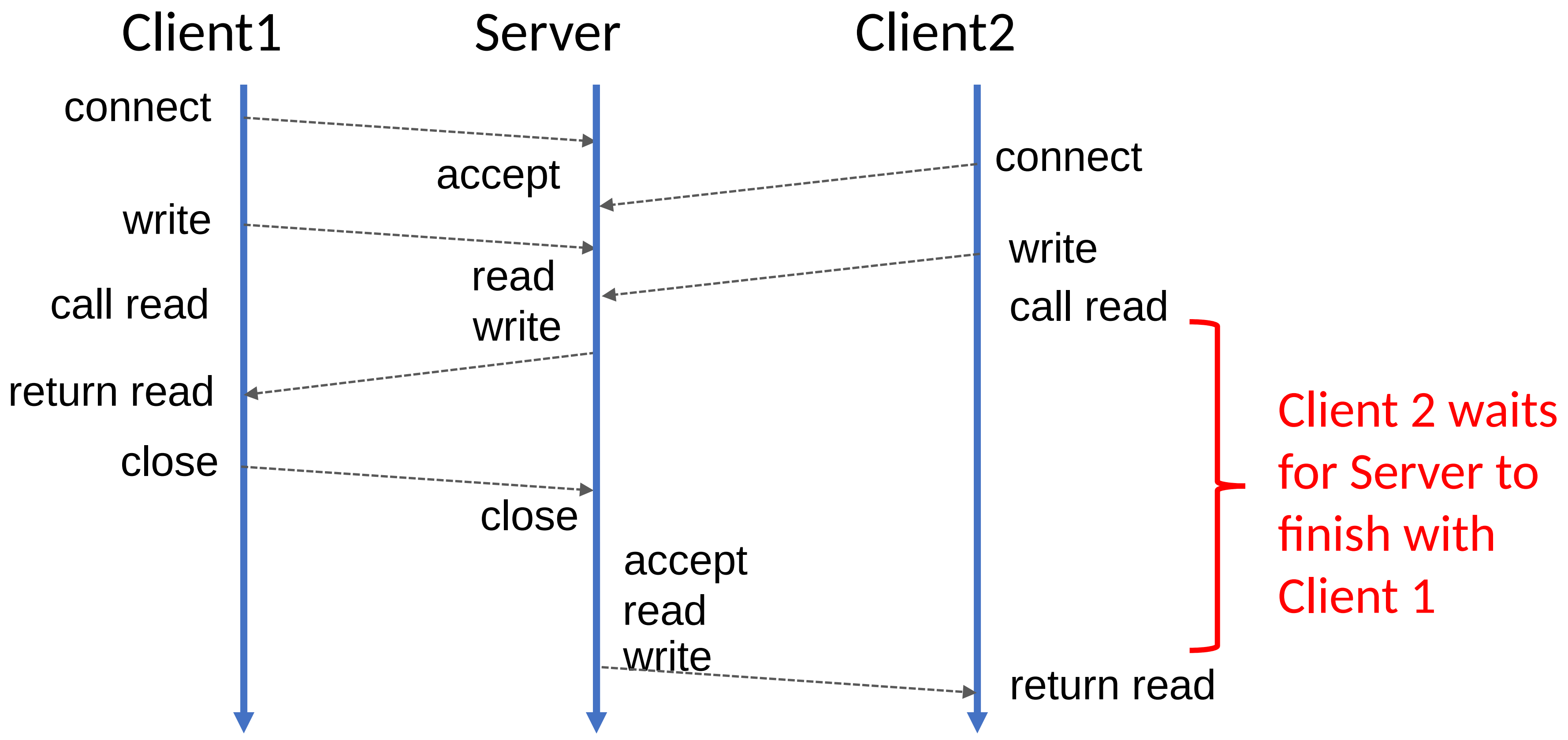
## Example of a Sequential Server



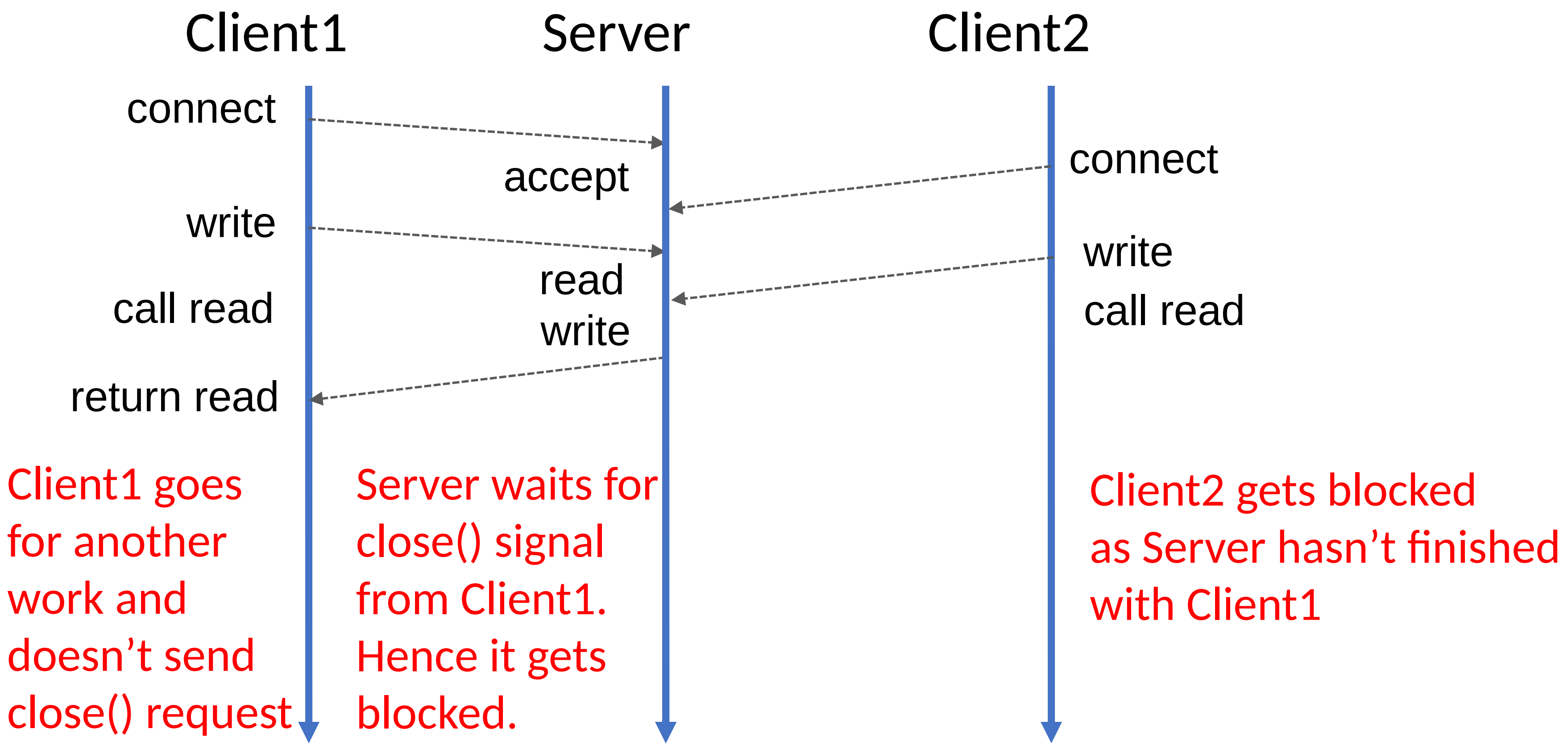
- Sequential server processes one client at a time

## Example of a Sequential Server



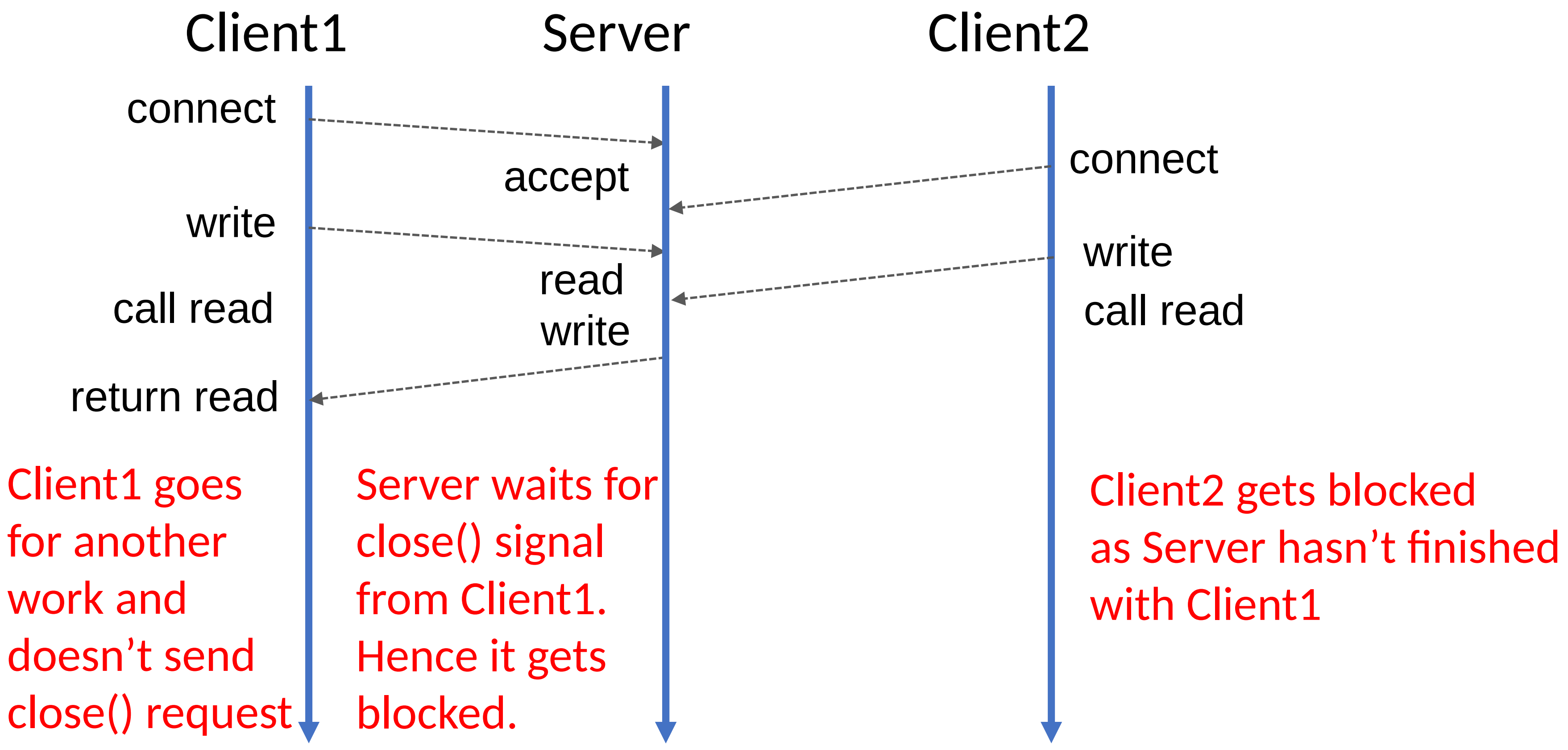
- Sequential server processes one client at a time
- In this example, Client 2 has a long waiting time

## Big problem with Sequential Server


Client1
Server
Client2
connect
accept
connect
write
read
write
write
call read
call read
return read
Client1 goes for another work and doesn't send close() request
Server waits for close() signal from Client1. Hence it gets blocked.
Client2 gets blocked as Server hasn't finished with Client1

Client2 remains blocked until Client1 sends close() request to Server.

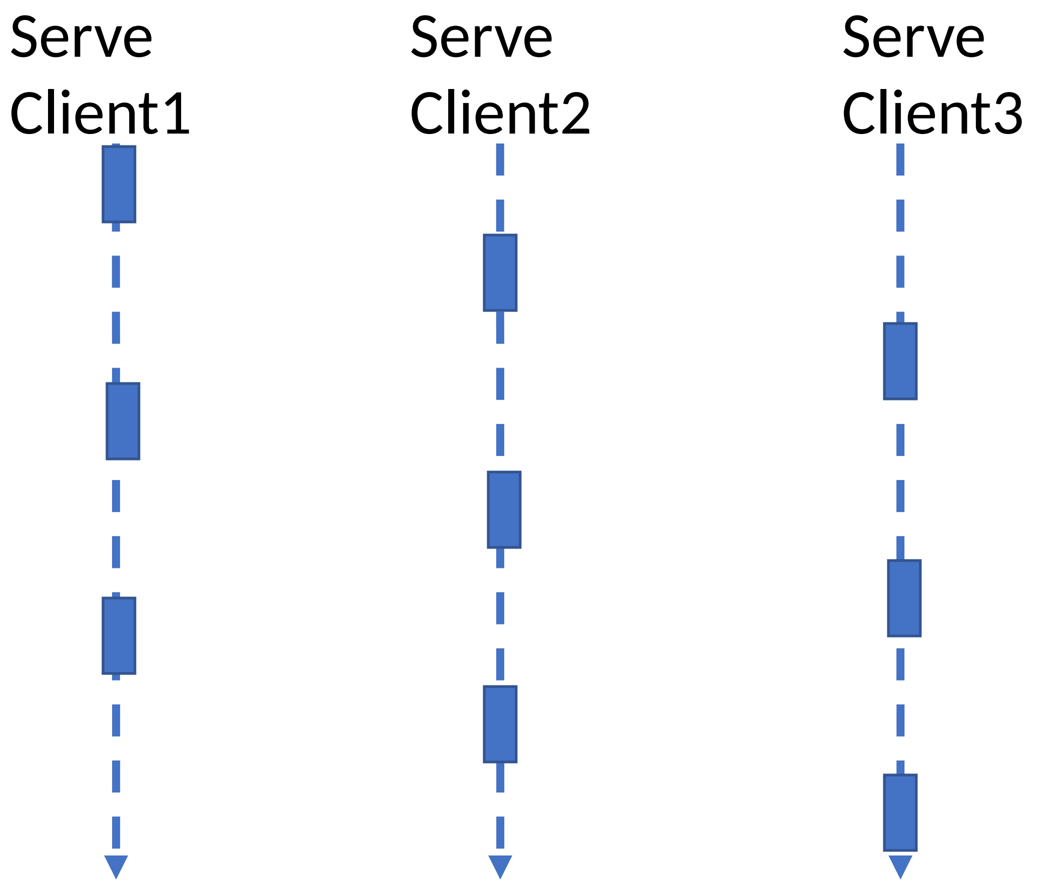
# Big problem with Sequential Server


Client1
Server
Client2
connect
accept
connect
write
write
read
call read
call read
write
return read
Client1 goes for another work and doesn't send close() request
Server waits for close() signal from Client1. Hence it gets blocked.
Client2 gets blocked as Server hasn't finished with Client1

Client2 remains blocked until Client1 sends close() request to Server.

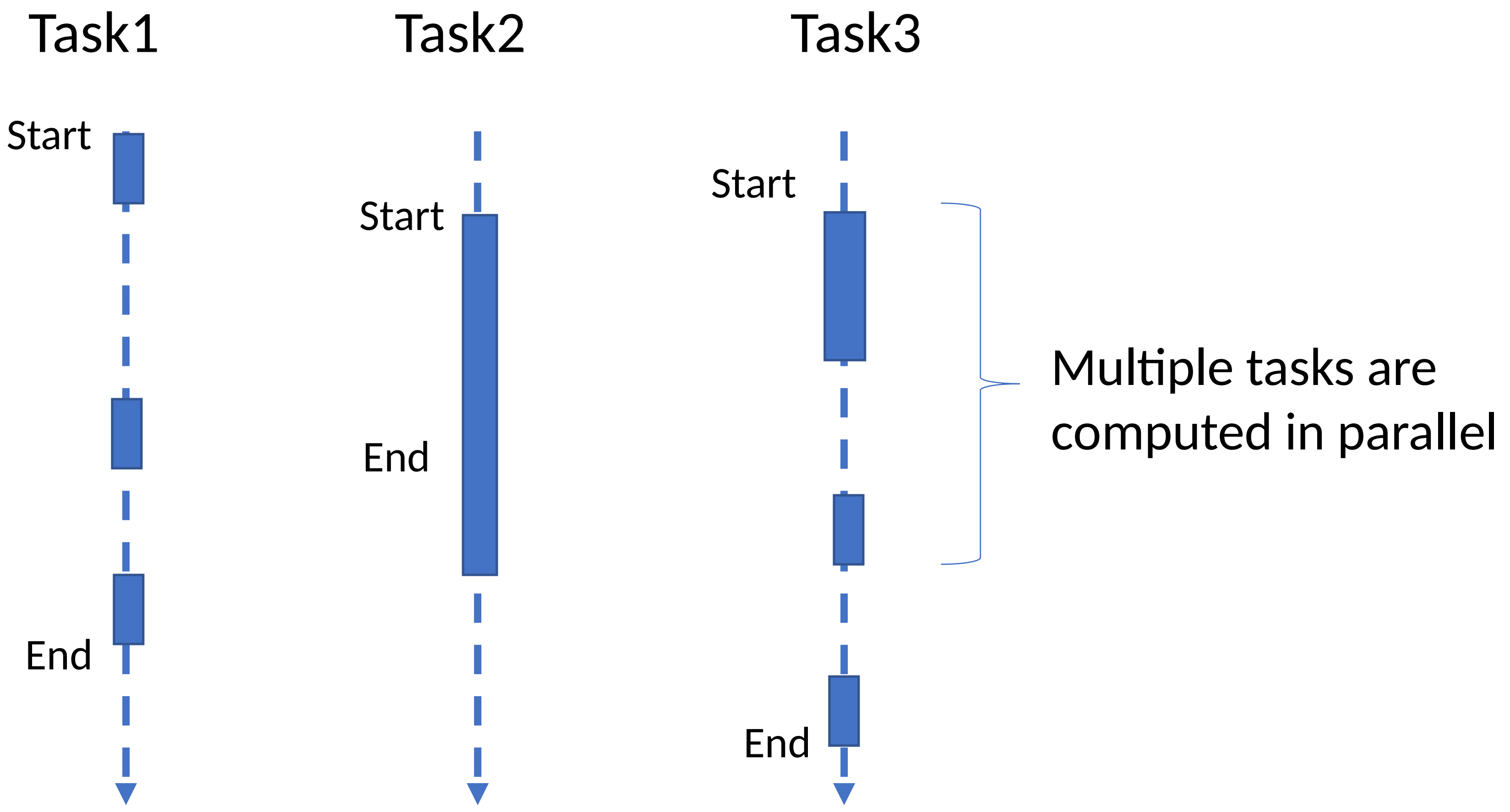
**Solution**: Use a concurrent server to serve multiple clients concurrently. Thus, a client cannot block another client.

## Concurrent Server



- All clients get served by the server.
- Even if one client causes blocking, the other clients do not have to wait.

## Concurrent vs Parallel



- Task2 is **parallel** to Task1 and Task3
- Parallel tasks are always concurrent.
- Concurrent tasks may not be parallel (Task1 and Task3)
- So, 'concurrency' is a more general term

## Concurrency using Threads

## Program execution: Sequential perspective

```
#include <stdio.h>
void do_one_thing(int *);
void do_another_thing(int *);
void do_wrap_up(int, int);

int r1 = 0, r2 = 0;
int main(void)
{
	do_one_thing(&r1);
	do_another_thing(&r2);
	do_wrap_up(r1, r2);
	return 0;
}
```

In a serial system, we see this program as a *sequence of instructions* which are executed one-by-one.

- It starts from main()
- then it executes the instructions that are inside do_one_thing()
- after that do_another_thing()
- and finally do_wrap_up()

## Program execution: Concurrent perspective

```
#include <stdio.h>
void do_one_thing(int *);
void do_another_thing(int *);
void do_wrap_up(int, int);

int r1 = 0, r2 = 0;
int main(void)
{
	do_one_thing(&r1);
	do_another_thing(&r2);
	do_wrap_up(r1, r2);
	return 0;
}
```

We look at our program (our big task) as a collection of subtasks.

Example:
Subtask1 is do_one_thing()
Subtask2 is do_another_thing()
Subtask3 is do_wrap_up()

**IDEA:** If it is possible to execute some of these subtasks at the same time with no change in final result (i.e., correctness), then we can **reduce overall time** by executing these subtasks **concurrently**.

```
C = A + B
D = C - E
F = D - K
```

```
C = A + B
D = A - B
F = D - K
```

## Program execution: Concurrent perspective

```
#include <stdio.h>
void do_one_thing(int *);
void do_another_thing(int *);
void do_wrap_up(int, int);

int r1 = 0, r2 = 0;
int main(void)
{
	do_one_thing(&r1);
	do_another_thing(&r2);
	do_wrap_up(r1, r2);
	return 0;
}
```

We look at our program (our big task) as a collection of subtasks.

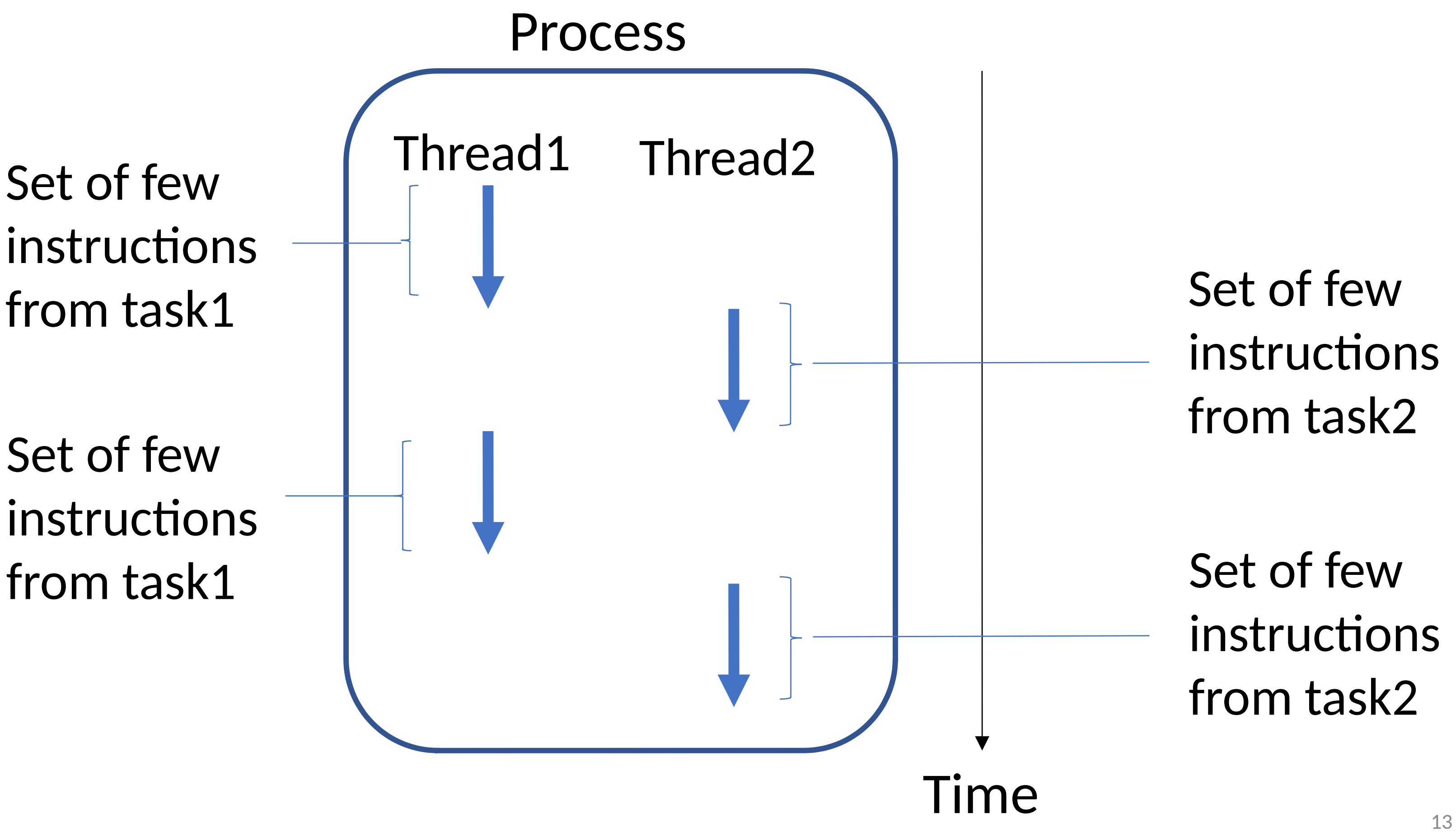
Example:
Subtask1 is do_one_thing()
Subtask2 is do_another_thing()
Subtask3 is do_wrap_up()

**These subtasks can be executed as 'threads' within a single program**

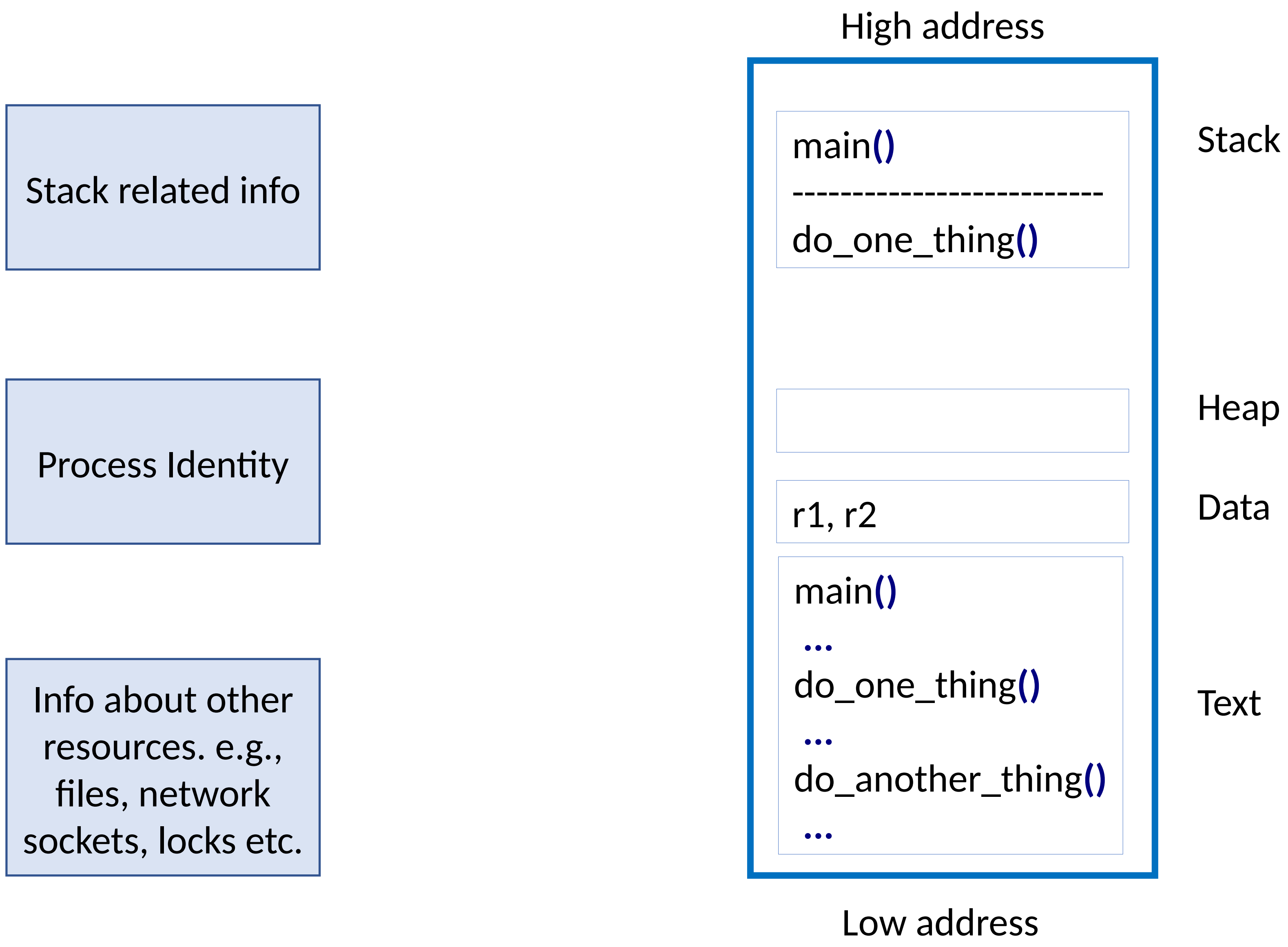
**IDEA:** If it is possible to execute some of these subtasks at the same time with no change in final result (i.e., correctness), then we can **reduce overall time** by executing these subtasks **concurrently**.

## What is a thread?

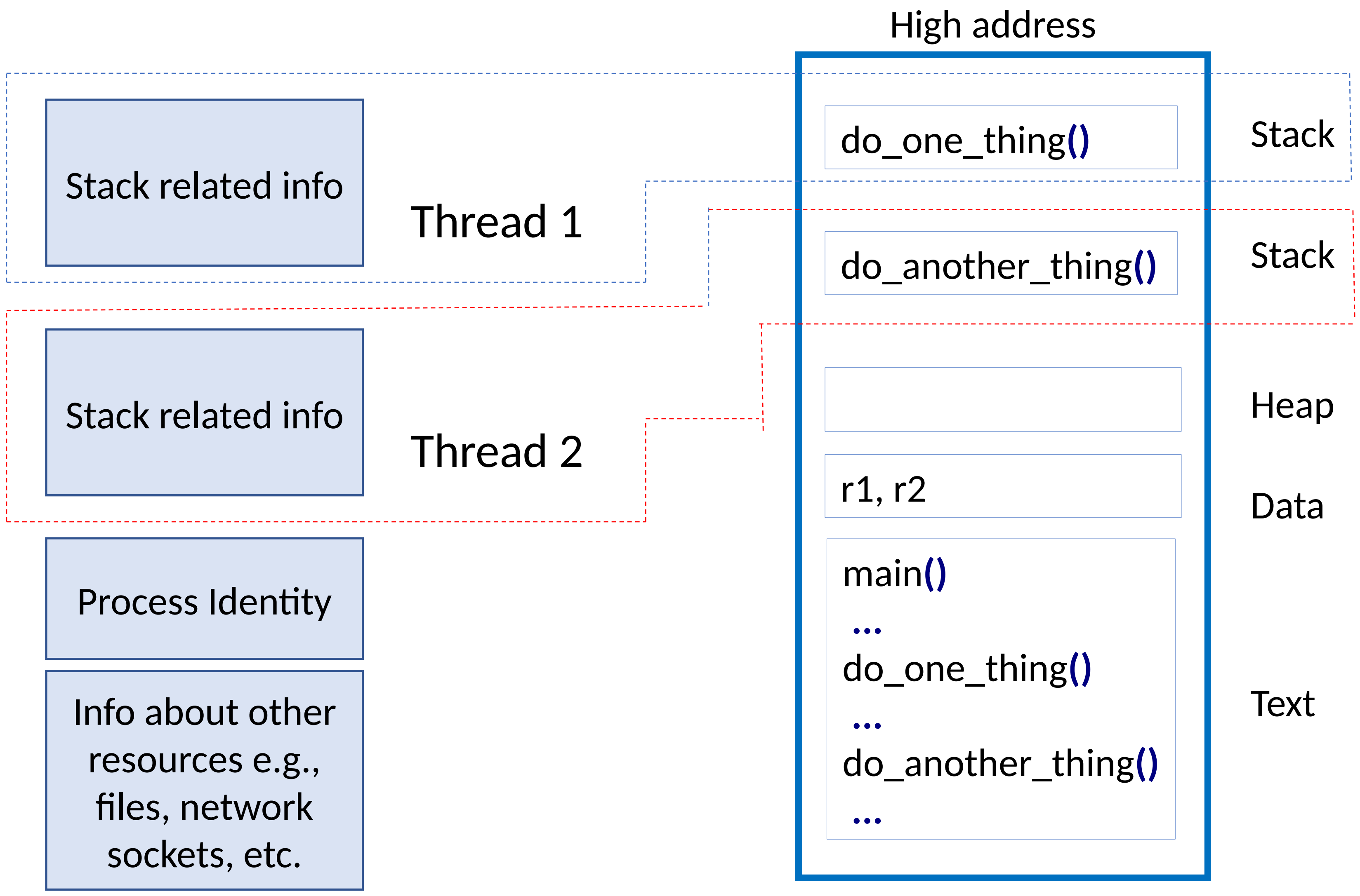
A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.


Process
Thread1
Thread2
Set of few instructions from task1
Set of few instructions from task1
Set of few instructions from task2
Set of few instructions from task2
Time

# Program as a single process with no threads



# Program as a single process with two threads



C program as a process with two threads, each executing a function (or task) concurrently in one of the two stack regions. Each thread has its own copy of stack related info. Both threads can refer to common Heap, global Data and other resources such as opened files, sockets etc.

## Using Pthreads library in C

- In your C program you need to #include <pthread.h>
- There are many functions related to pthreads.
- On a UNIX-based system, a list of these functions can typically be obtained with the command man -k pthread
- To know about a specific function see the man-page.
- When linking, you must link to the pthread library using compilation flag -lpthread
  Example: gcc -lpthread file.c
  or
  ```
  gcc file.c -lpthread
  ```

## Spawning a thread using pthread_create()

- A thread is created using pthread_create()
  The syntax is:

```
int pthread_create(
   pthread_t *thread_id,   // ID number for thread
   const pthread_attr_t *attr,   // controls thread attributes
   void *(*function)(void *),   // function to be executed
   void *arg   // argument of function
);
```

- pthread_create( ) returns 0 if thread creation is successful
- Otherwise it returns a nonzero value to indicate an error.

## Spawning a thread using pthread_create()

- We will use default attributes set by the system
  So, *attr gets replaced by **NULL**
  The easier syntax is:

```
int pthread_create(
   pthread_t *thread_id,   // ID number for thread
   NULL,   // we set it to default thread attributes
   void *(*function)(void *),   // function to be executed
   void *arg   // argument of function
);
```

## Spawning a thread using pthread_create()

- A thread is created using pthread_create()
  The syntax is:

```
int pthread_create(
   pthread_t *thread_id,   // ID number for thread
   const pthread_attr_t *attr,   // controls thread attributes
   void *(*function)(void *),   // function to be executed
   void *arg   // argument of function
);
```

Example of functions that can be passed to pthread_create()

```
void *foo1();            ✓
void foo2(int *);        ✓
int *foo3(int *);        ✓
int *foo4(int *, int*);  ✗
```

## Note: functions with multiple arguments

- Consider the function with multiple arguments

  T *foo (T *, T *);

  where T represents any data type.

- Solution: Pack all arguments into a compound object and create a wrapper function which takes the compound argument and then unpacks inside before passing the unpacked arguments to foo()

```
typedef struct Compound{
     T *a, *b;
} Compound;

T * foo_wrapper(Compound *c){// This can be passed to pthread_create
     T  *d;
     d=foo(c->a, c->b);
}
```

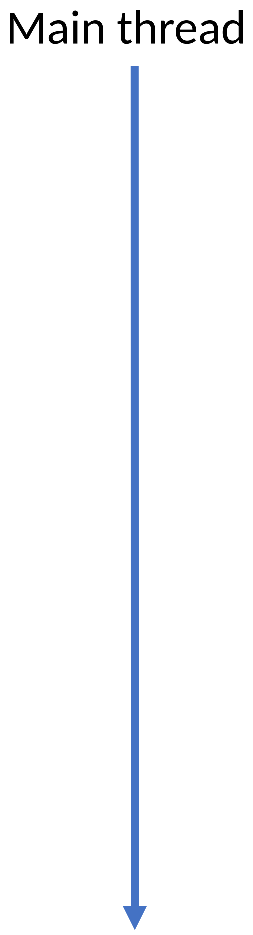
## Example: program with threads

## File pthread0.c

```
void do_one_thing()
{
  int i, j, x;

  for (i = 0; i  <  200;  i++) {
    printf("doing one thing\n");
  }

}
void do_another_thing()
{
  int i, j, x;

  for (i = 0;    i  <  200;  i++) {
    printf("doing another \n");
  }
}
```

```
int main(void)
{
  pthread_t thread1,  thread2;

  pthread_create(&thread1,
          NULL,
          (void *) do_one_thing,
          NULL);

  pthread_create(&thread2,
          NULL,
          (void *) do_another_thing,
          NULL);
  sleep(1);      // sleeps 1s
  return 0;
}
```

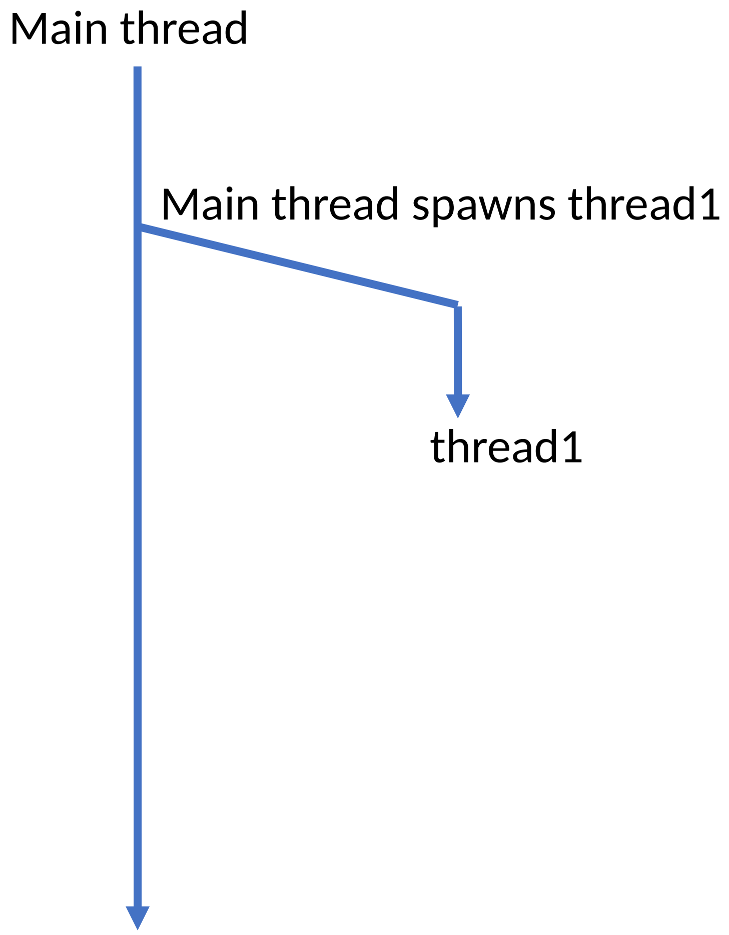
## Program flow thread0.c



```
int main(void)
{
  pthread_t thread1,  thread2;

  pthread_create(&thread1,
         NULL,
         (void *) do_one_thing,
         NULL);

  pthread_create(&thread2,
         NULL,
         (void *)
do_another_thing,
         NULL);
  sleep(1);     // sleeps 1s
  return 0;
}
```

1. Initially, only the main thread is present.

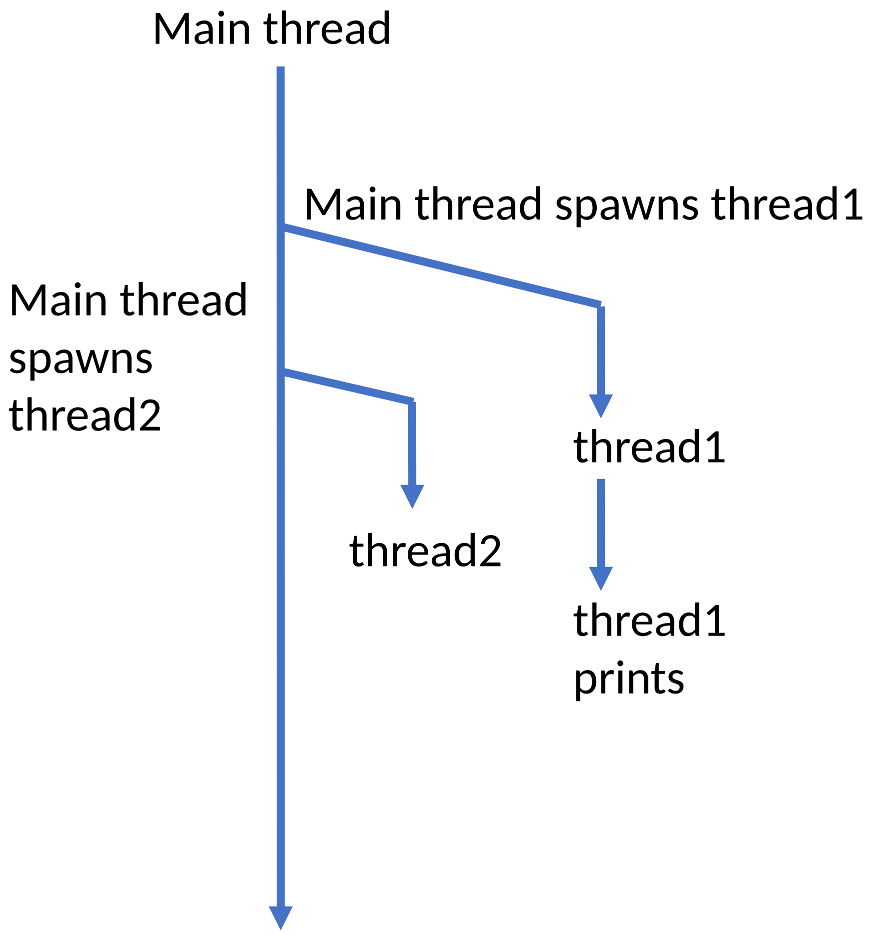
## Program flow thread0.c



```
int main(void)
{
  pthread_t thread1,  thread2;

  pthread_create(&thread1,
        NULL,
        (void *) do_one_thing,
        NULL);

  pthread_create(&thread2,
        NULL,
        (void *)
do_another_thing,
        NULL);
  sleep(1);     // sleeps 1s
  return 0;
}
```

2. Main thread spawns thread1

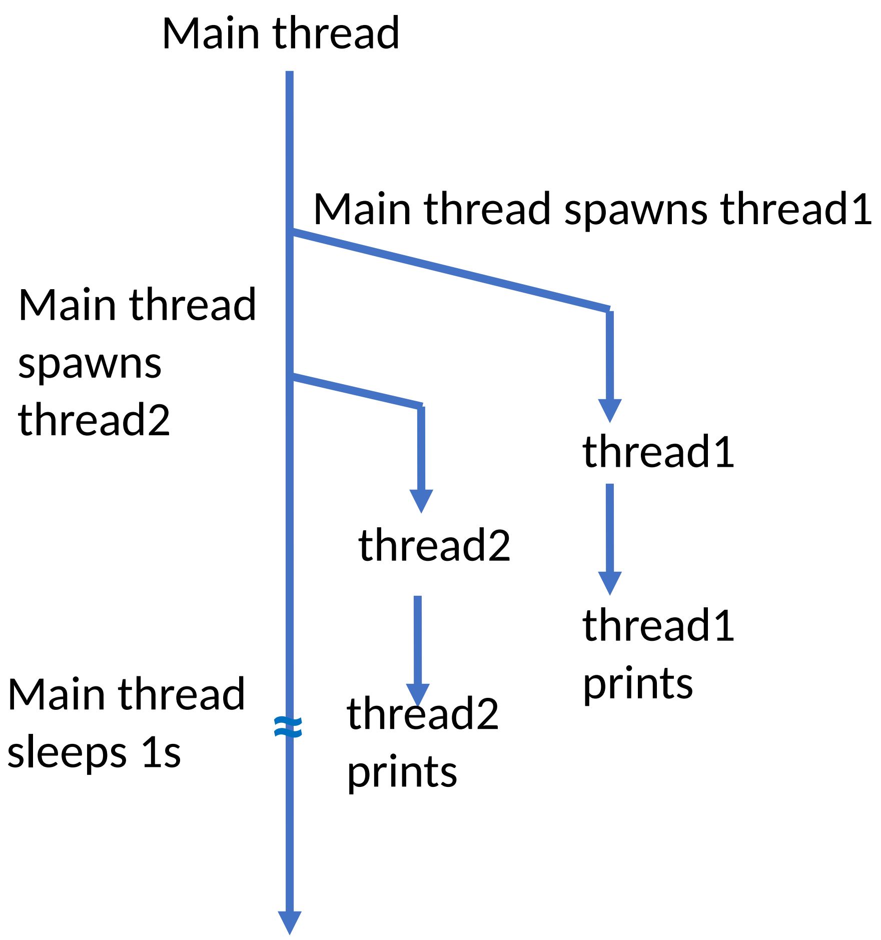
## Program flow thread0.c



```
int main(void)
{
  pthread_t thread1,  thread2;

  pthread_create(&thread1,
        NULL,
        (void *) do_one_thing,
        NULL);

  pthread_create(&thread2,
        NULL,
        (void *)
do_another_thing,
        NULL);
  sleep(1);     // sleeps 1s
  return 0;
}
```

3. Main thread spawns thread2.
4. It might happen that thread1 has started printing

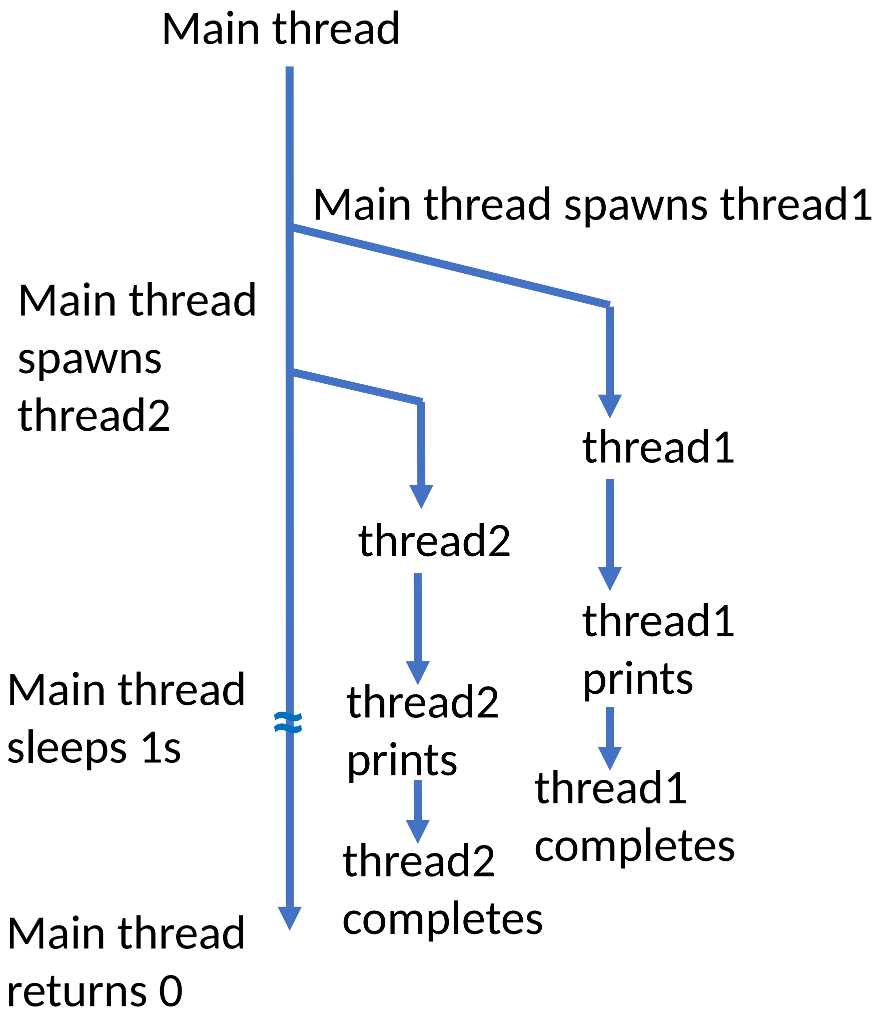
# Program flow thread0.c



```
int main(void)
{
  pthread_t thread1,  thread2;

  pthread_create(&thread1,
        NULL,
        (void *) do_one_thing,
        NULL);

  pthread_create(&thread2,
        NULL,
        (void *)
do_another_thing,
        NULL);
  sleep(1);     // sleeps 1s
  return 0;
}
```

5. While main thread goes for a sleep of 1s,
   the other two threads continue with printing and finally completes.

## Program flow thread0.c



```
int main(void)
{
  pthread_t thread1,  thread2;

  pthread_create(&thread1,
          NULL,
          (void *) do_one_thing,
          NULL);

  pthread_create(&thread2,
          NULL,
          (void *)
do_another_thing,
          NULL);
  sleep(1);      // sleeps 1s
  return 0;
}
```

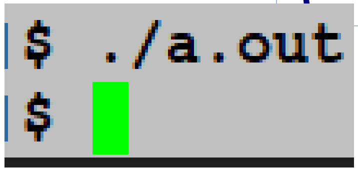
6. Finally the main thread finishes and returns.

## Program flow thread0.c with // sleep(1)

```
int main(void)
{
  pthread_t thread1,  thread2;

  pthread_create(&thread1,
          NULL,
          (void *) do_one_thing,
          NULL);

  pthread_create(&thread2,
          NULL,
          (void *)
do_another_thing,
          NULL);
 //sleep(1); // sleeps 1s
  return 0;
}
```

## Program flow thread0.c with // sleep(1)

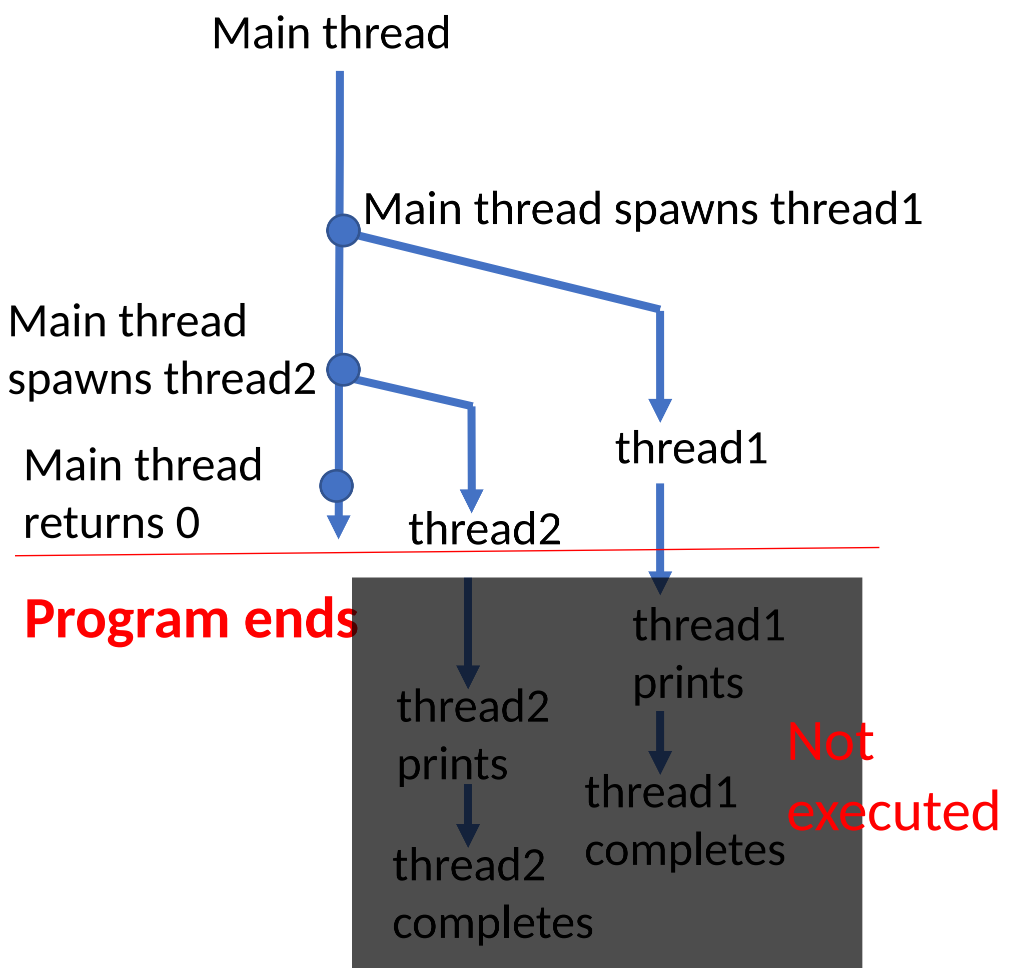
```
int main(void)
{
  pthread_t thread1,  thread2;

  pthread_create(&thread1,
        NULL,
        (void *) do_one_thing,
        NULL);

  pthread_create(&thread2,
        NULL,
        (void *)
do_another_thing,
        NULL);
 //sleep(1); // sleeps 1s
  return 0;
}
```



```
$ ./a.out
$ 
```

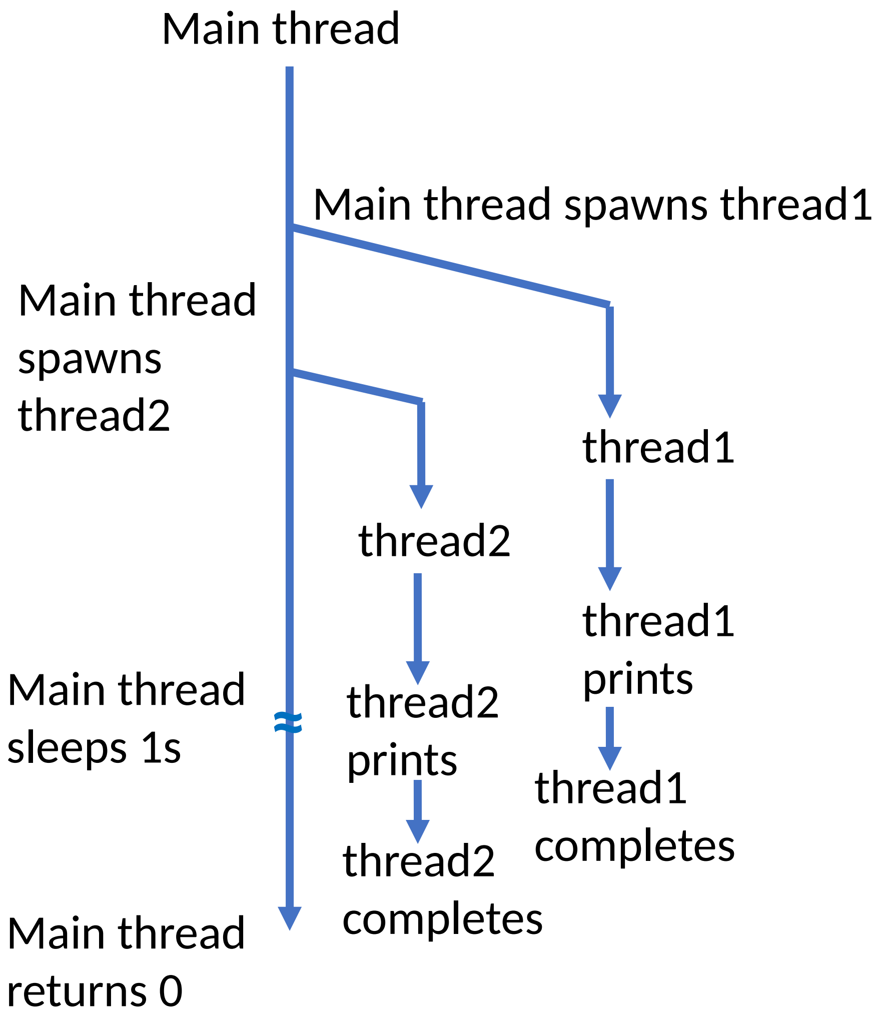
The program does not print anything!

# Program flow thread0.c with // sleep(1)



```
int main(void)
{
  pthread_t thread1,  thread2;

  pthread_create(&thread1,
        NULL,
        (void *) do_one_thing,
        NULL);

  pthread_create(&thread2,
        NULL,
        (void *)
do_another_thing,
        NULL);
 //sleep(1); // sleeps 1s
  return 0;
}
```

# Program flow thread0.c



```
int main(void)
{
  pthread_t thread1,  thread2;

  pthread_create(&thread1,
         NULL,
         (void *) do_one_thing,
         NULL);

  pthread_create(&thread2,
         NULL,
         (void *)
do_another_thing,
         NULL);
  sleep(1);     // sleeps 1s
  return 0;
}
```

We inserted a sleep(1) in main-thread and 'hoped' that Thread1 and Thread2 will finish by the time main-thread wakes up.

## Shared data objects in a concurrent system

Cooperation between concurrent threads leads to the sharing of

- Global data objects,
- Heap objects
- Files, etc.

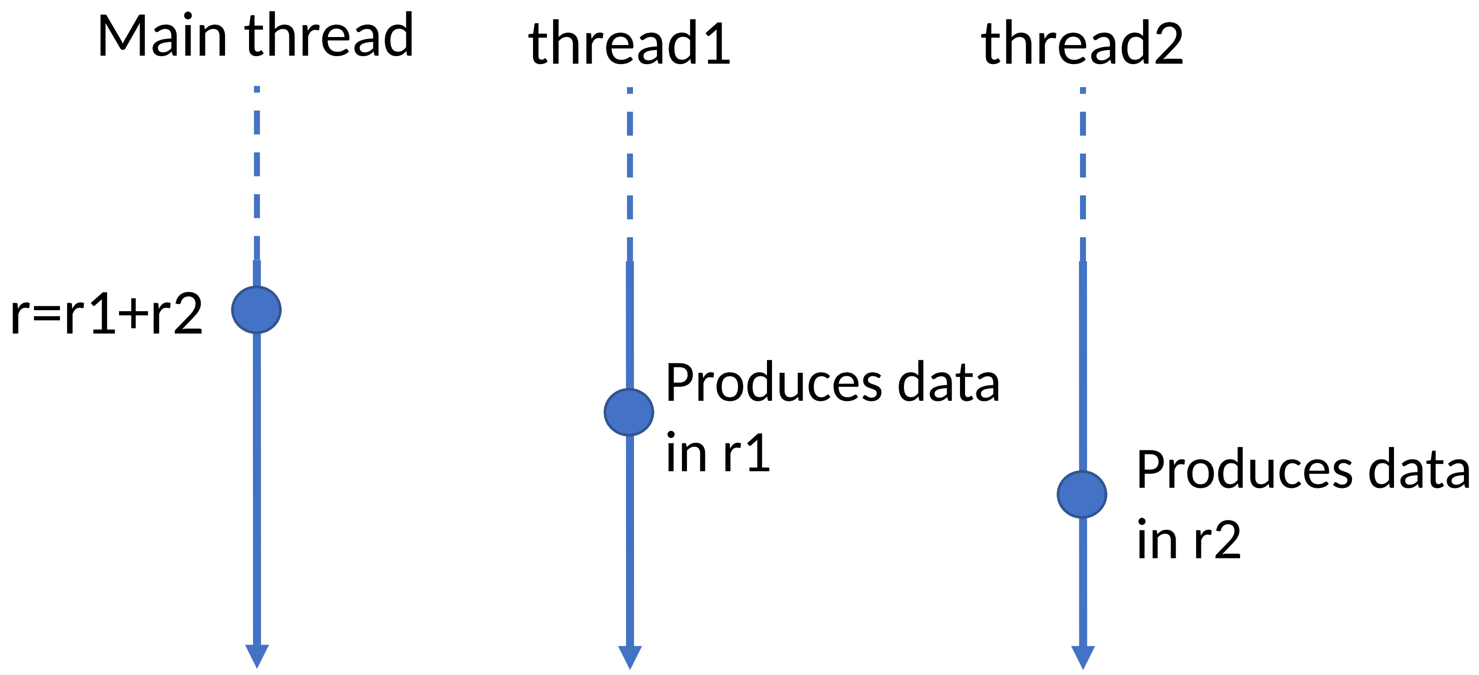
Lack of synchronization leads to chaos and wrong calculations.

```
// global variables r1 and r2
int r1 = 0, r2 = 0;
main(){     // main thread
  int r;
  ...
  Call do_one_thing() in thread1;
  Call do_another_thing() in thread2;
  r = r1 + r2;
  ...
}
```

```
do_one_thing(...){  // in thread1
  ...
  Compute result in r1;
  ...
}

do_another_thing(...){  // in thread2
  ...
  Compute result in r2;
  ...
}
```

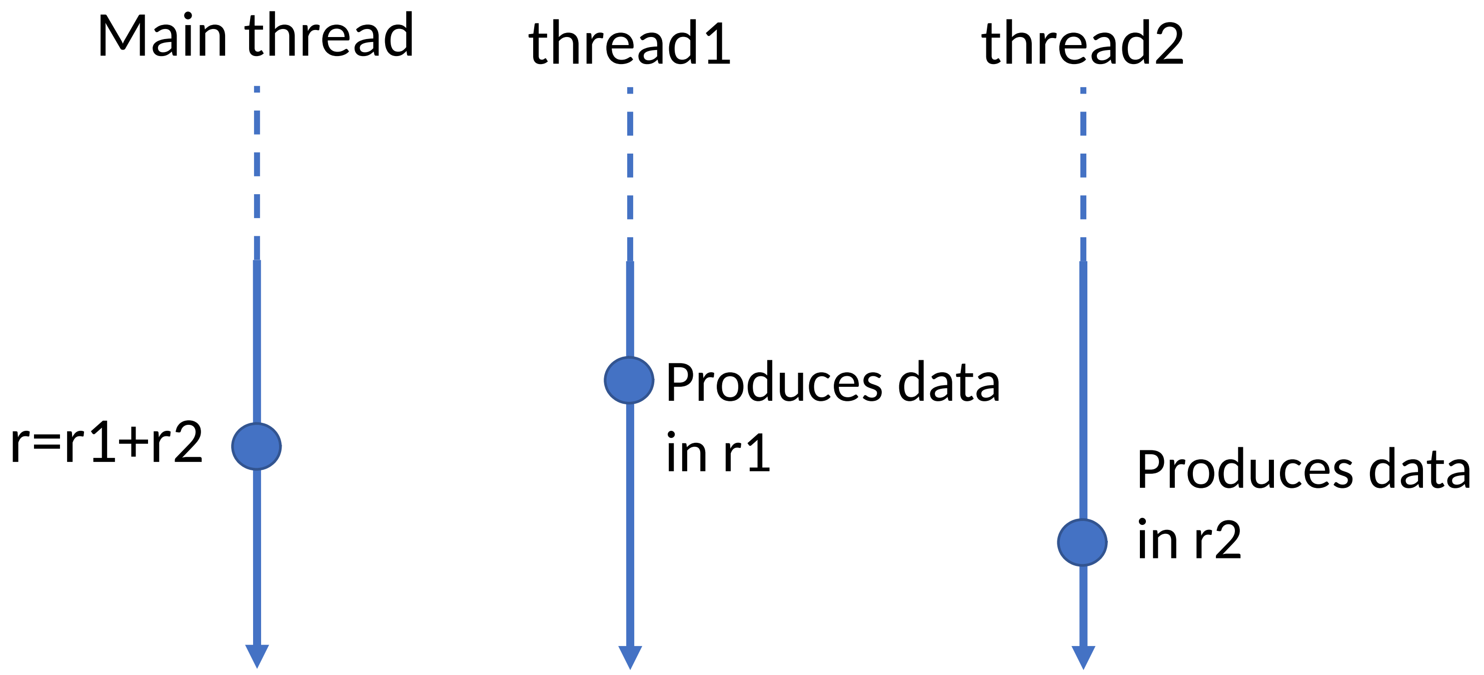
[ Assume that only thread1 uses r1, only thread2 uses r2 and only main-thread writes to r. ]

## What can go wrong?



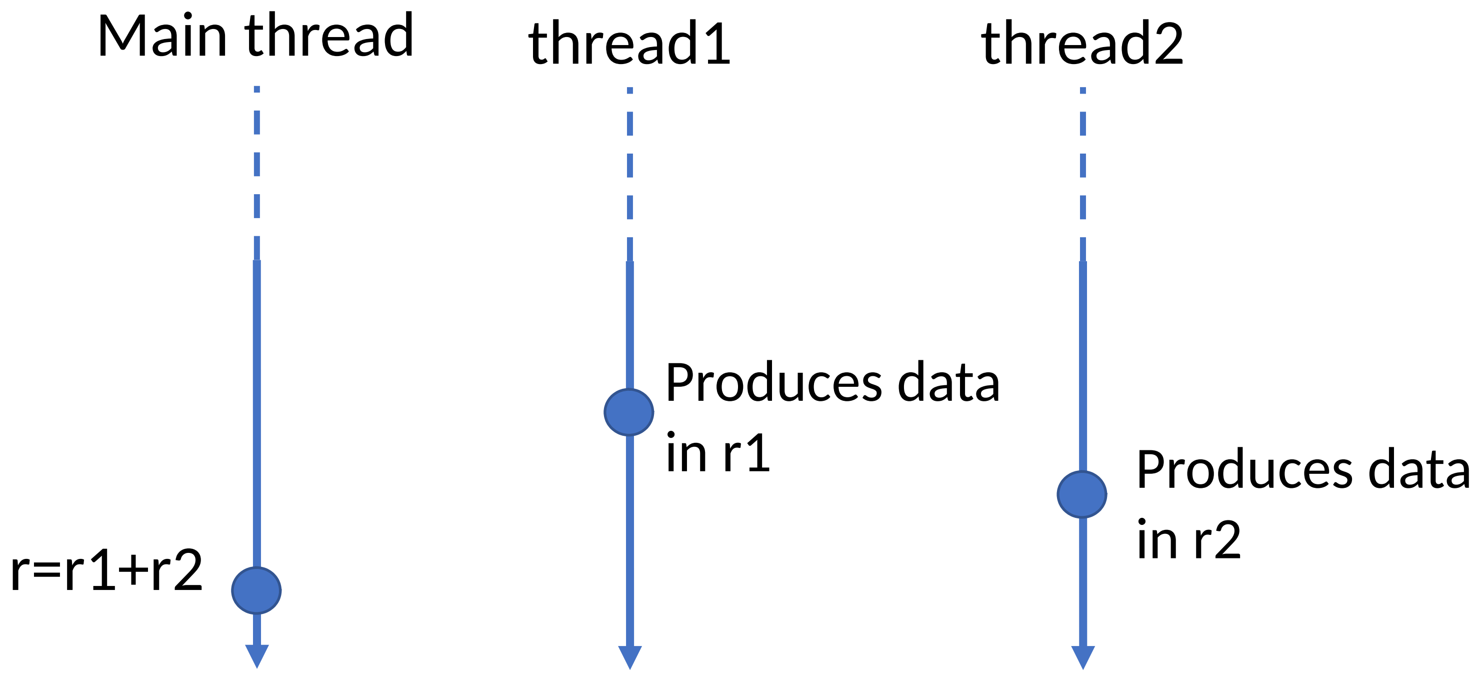
Scenario 1:
Main thread computes r1+r2 before thread1 and thread2 produces the results.
Both r1 and r2 will be wrong.

## What can go wrong?



Scenario 2:
Main thread computes r1+r2 after thread1 but before thread2 produces.
So, r1 will have correct but r2 will have wrong values.
Thus r will be wrong.

## What can go wrong?



Scenario 3:
Luckily, main thread computes r1+r2 after thread1 and thread2 produce.
Luckily, r will be correct.

## Synchronization of threads

Synchronization in threads programming is used to make sure that some events happen in order.

Image source https://theclassicalnovice.com/glossary/ensembles/

## Synchronization mechanism in Pthreads

Pthreads library provides three synchronization mechanisms:

- Joins
- Mutual exclusions
- Condition variables

## Synchronizing threads using pthread_join()

- pthread_join() is a blocking function
  The syntax is:

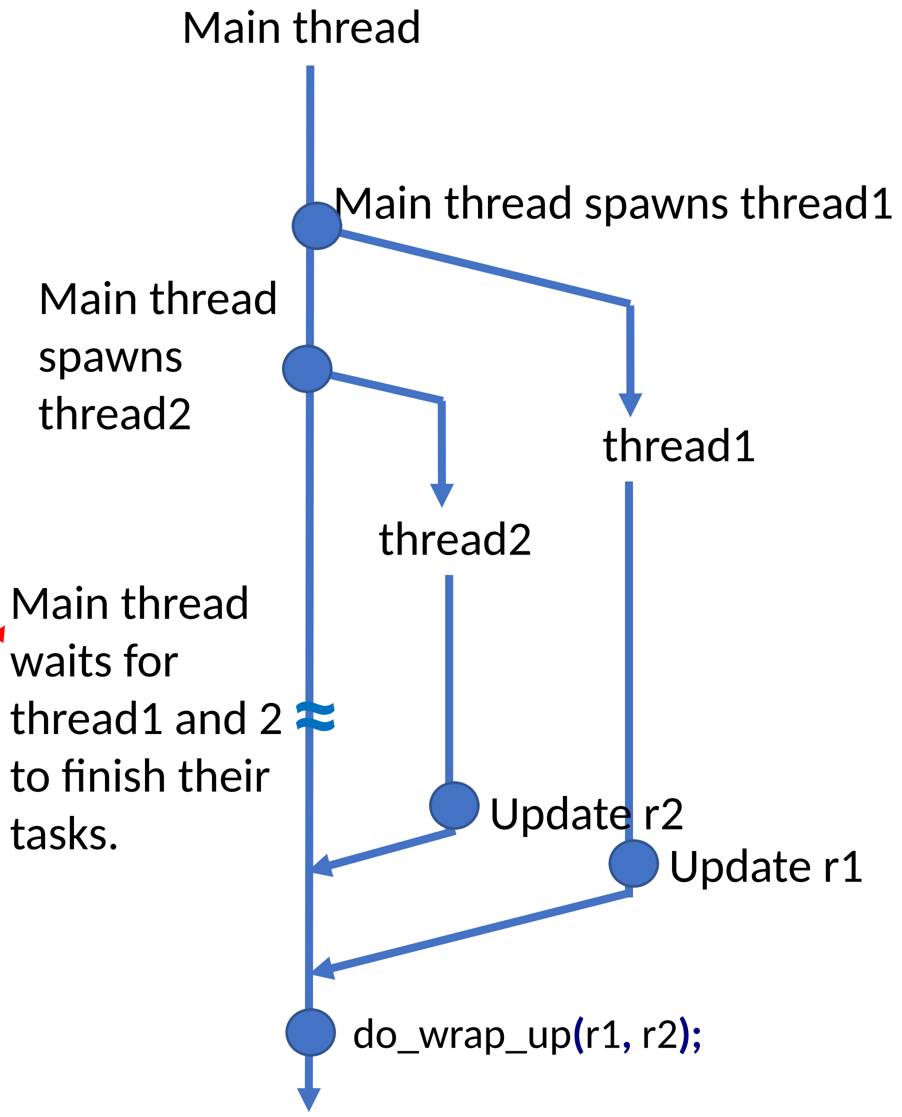
```
int pthread_join(
  pthread_t thread_id, // ID of thread to "join"
  void **value_pntr // address of function's return value
);
```

- Again, we will set value_pntr to **NULL**
  The syntax we will use is:

```
int pthread_join(
  pthread_t thread_id, // ID of thread to "join"
  NULL
);
```

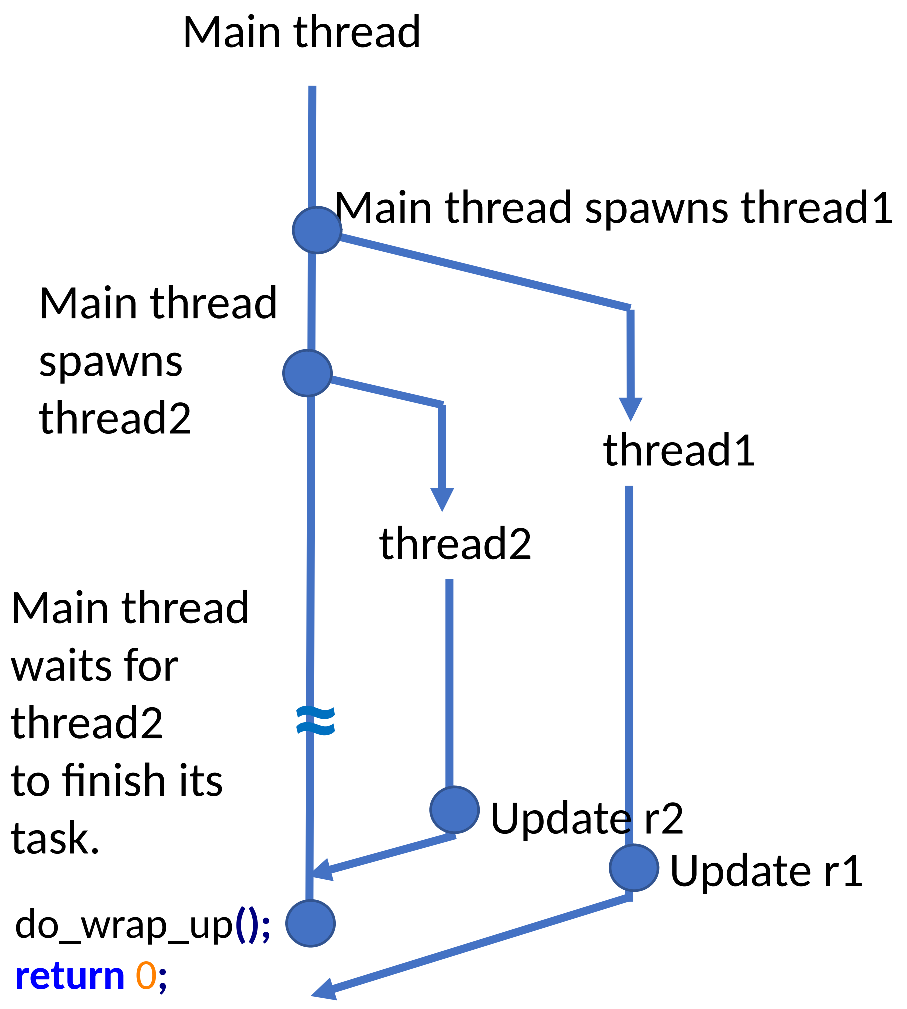
## Example: Synchronizing threads using pthread_join()

```
int r1 = 0, r2 = 0;

int main(void){
  pthread_t thread1,  thread2;

  pthread_create(&thread1,
        NULL,
        (void *) do_one_thing,
        (void *) &r1);

  pthread_create(&thread2,
        NULL,
        (void *) do_another_thing,
        (void *) &r2);

  pthread_join(thread1,  NULL);
  pthread_join(thread2,  NULL);

  do_wrap_up(r1, r2);
  return 0;
}
```


Main thread
Main thread spawns thread1
Main thread spawns thread2
thread1
thread2
Main thread waits for thread1 and 2 to finish their tasks.
≈
Update r2
Update r1
do_wrap_up(r1, r2);

## Example: Synchronizing threads using pthread_join()

```
int r1 = 0, r2 = 0;

int main(void){
  pthread_t thread1,  thread2;

  pthread_create(&thread1,
        NULL,
        (void *) do_one_thing,
        (void *) &r1);

  pthread_create(&thread2,
        NULL,
        (void *) do_another_thing,
        (void *) &r2);

  //pthread_join(thread1,  NULL);
  pthread_join(thread2,  NULL);

  do_wrap_up(&r1, &r2);
  return 0;
}
```


Main thread
Main thread spawns thread1
Main thread spawns thread2
thread1
thread2
Main thread waits for thread2 to finish its task.
≈
Update r2
Update r1
do_wrap_up();
return 0;

The main thread doesn’t wait for thread1 to finish. The result will be incorrect.

## What did we assume in the previous synchronization example?

```
// global var r1 and r2
int r1 = 0, r2 = 0;
main(){     // main thread
  int r;
  ...
  Call do_one_thing() in thread1;
  Call do_another_thing() in thread2;
  r = r1 + r2;
  ...
}
```

```
do_one_thing(...){  // in thread1
  ...
  Compute result in r1;
  ...
}

do_another_thing(...){  // in thread2
  ...
  Compute result in r2;
  ...
}
```

We assumed that only thread1 uses r1, only thread2 uses r2 and only main-thread writes to r after thread1 and 2 finish.

What will happen if several threads try to read/write a shared variable?

## Example: Several threads update a shared data

```
void *functionC();
int  counter = 0;

main(){
  int rc1, rc2;
  pthread_t thread1, thread2;

  // Two threads execute functionC() concurrently
  pthread_create(&thread1, NULL, &functionC, NULL);
  pthread_create(&thread2, NULL, &functionC, NULL);

  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  return 0;
}
void *functionC(){
  counter++;
  printf("Counter value: %d\n", counter);
}
```

We expect the shared data ‘counter’ to be updated by one thread at a time and thus expect the program to print:

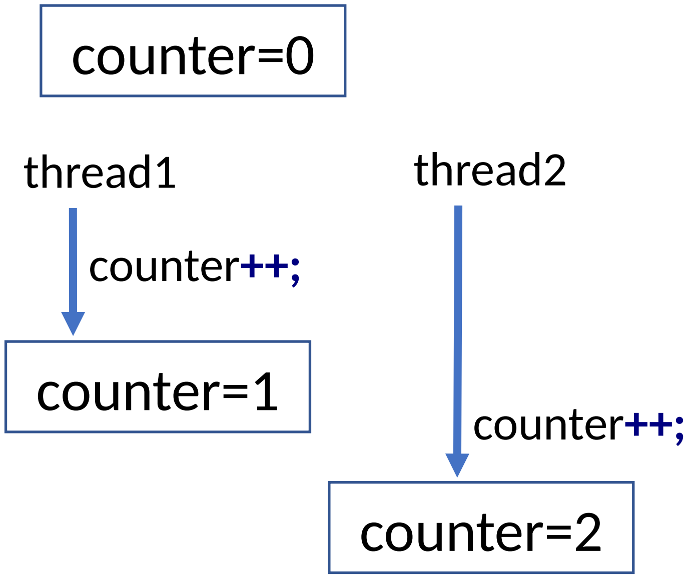
```
1
2
```

**What can go wrong?**

## Example: Several threads update a shared data

```
void *functionC();
int  counter = 0;

main(){
  int rc1, rc2;
  pthread_t thread1, thread2;

  // Two threads execute functionC() concurrently
  pthread_create(&thread1, NULL, &functionC, NULL);
  pthread_create(&thread2, NULL, &functionC, NULL);

  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  return 0;
}
void *functionC(){
  counter++;
  printf("Counter value: %d\n", counter);
}
```

**Scenario 1:**



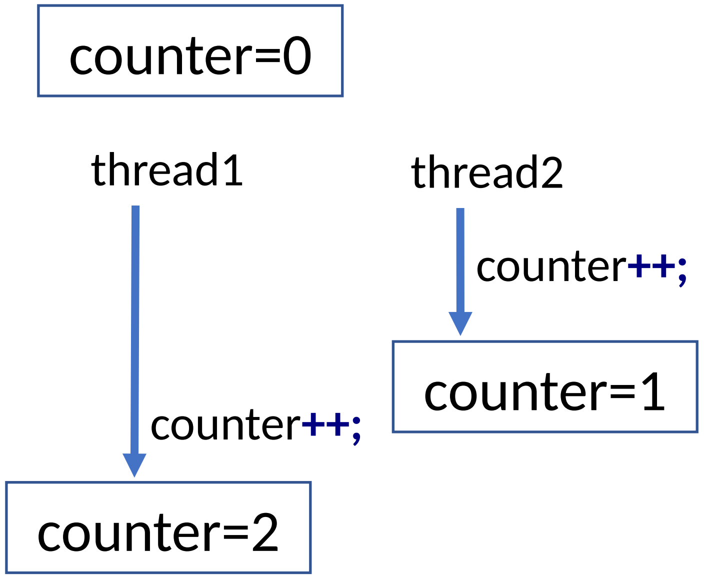
Program prints:

```
1
2
```

## Example: Several threads update a shared data

```
void *functionC();
int  counter = 0;

main(){
  int rc1, rc2;
  pthread_t thread1, thread2;

  // Two threads execute functionC() concurrently
  pthread_create(&thread1, NULL, &functionC, NULL);
  pthread_create(&thread2, NULL, &functionC, NULL);

  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  return 0;
}
void *functionC(){
  counter++;
  printf("Counter value: %d\n", counter);
}
```

**Scenario 2:**



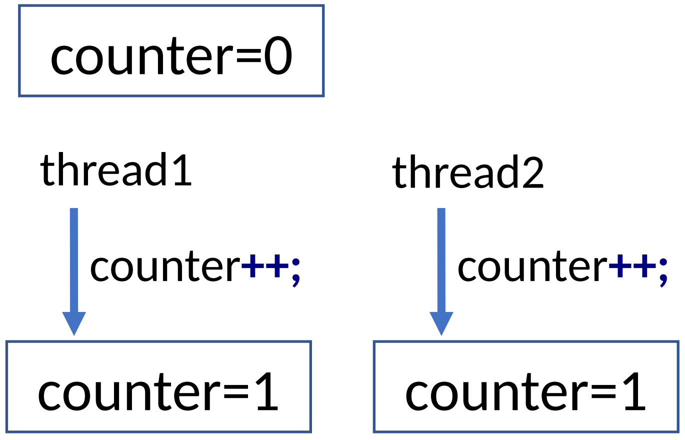
Program prints:

```
1
2
```

## Example: Several threads update a shared data

```
void *functionC();
int  counter = 0;

main(){
  int rc1, rc2;
  pthread_t thread1, thread2;

  // Two threads execute functionC() concurrently
  pthread_create(&thread1, NULL, &functionC, NULL);
  pthread_create(&thread2, NULL, &functionC, NULL);

  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  return 0;
}
void *functionC(){
  counter++;
  printf("Counter value: %d\n", counter);
}
```

**Scenario 3:**



Both threads race to update the shared data at the same time.

Program prints:

```
1
1
```

Data inconsistencies due to race conditions

## Race condition and its prevention

- A race condition often occurs when two or more threads need to perform operations on the same data, but the results of computations depend on the order in which these operations are performed.

- The problem can be solved if we can enforce **mutual exclusion**
  - Threads get exclusive access to the shared resource in turn

- The Pthreads library offers ‘**mutex**’ objects to enforce exclusive access by a thread to a variable or a set of variables.

## Mutual exclusion in Pthread: mutex

- Syntax for declaration and initialization of a mutex object is

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
```

This statement will create a mutex object ‘mutex1’ and initialize it with the default characteristics.

- Generally mutex objects are declared as global.

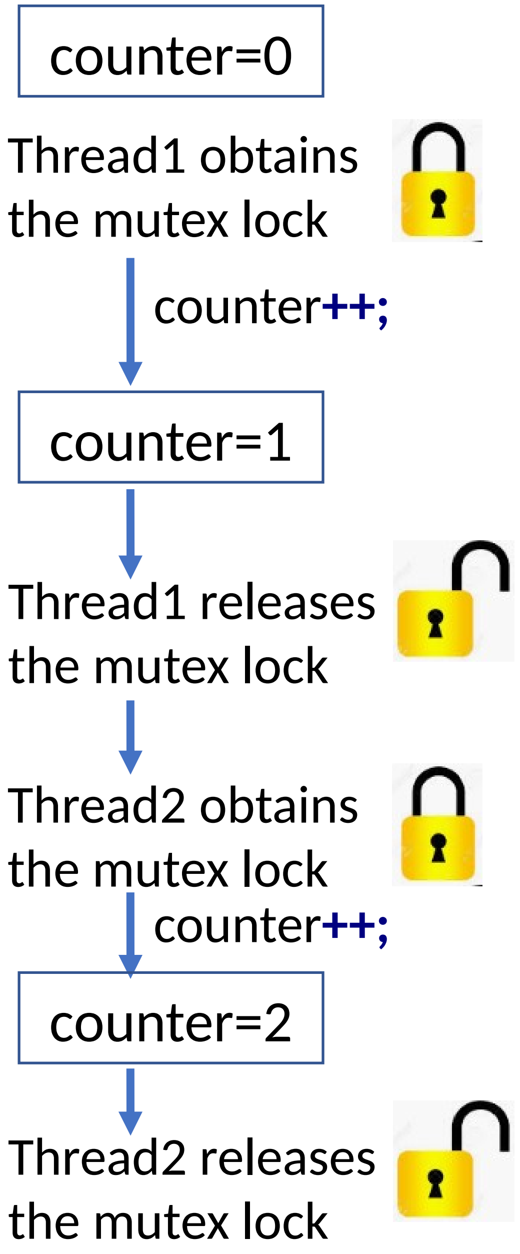
- The following shows the use of mutex for serializing access to a shared resource.

```
...
pthread_mutex_lock( &mutex1 );
counter++;
pthread_mutex_unlock( &mutex1 );
...
```

Threads access ‘counter’ serially one after another.

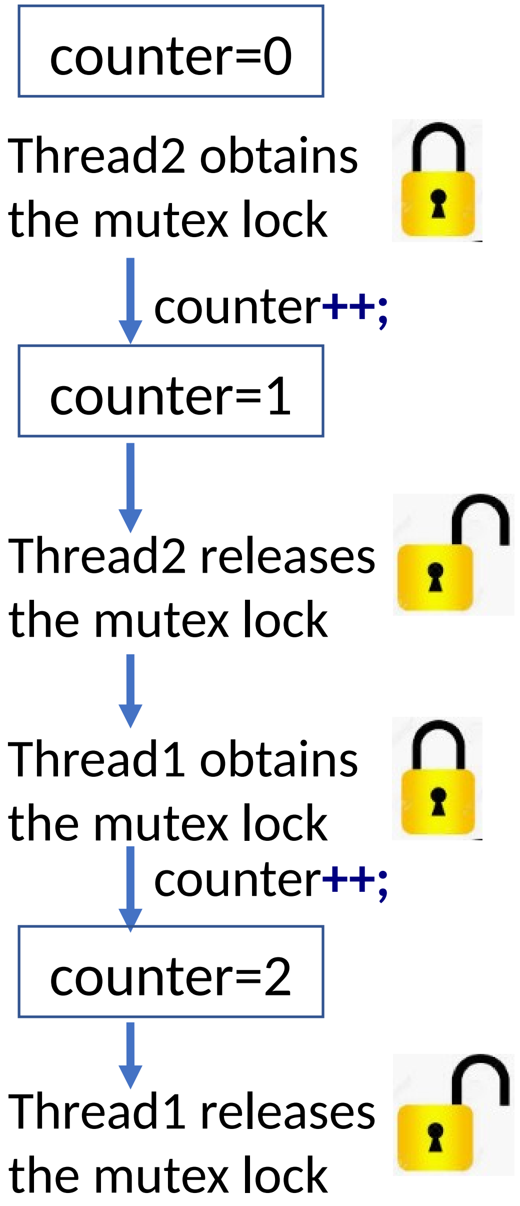
## Several threads update a shared data using mutex lock/unlock

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
void *functionC();
int  counter = 0;
main(){
  int rc1, rc2;
  pthread_t thread1, thread2;

  // Two threads execute functionC() concurrently
  pthread_create(&thread1, NULL, &functionC, NULL);
  pthread_create(&thread2, NULL, &functionC, NULL);

  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  return 0;
}
void *functionC(){
   pthread_mutex_lock( &mutex1 );
   counter++;
   printf("Counter value: %d\n",counter);
   pthread_mutex_unlock( &mutex1 );
}
```


counter=0
Thread1 obtains the mutex lock
counter++;
counter=1
Thread1 releases the mutex lock
Thread2 obtains the mutex lock
counter++;
counter=2
Thread2 releases the mutex lock

## Several threads update a shared data using mutex lock/unlock

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
void *functionC();
int  counter = 0;
main(){
  int rc1, rc2;
  pthread_t thread1, thread2;

  // Two threads execute functionC() concurrently
  pthread_create(&thread1, NULL, &functionC, NULL);
  pthread_create(&thread2, NULL, &functionC, NULL);

  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  return 0;
}
void *functionC(){
   pthread_mutex_lock( &mutex1 );
   counter++;
   printf("Counter value: %d\n",counter);
   pthread_mutex_unlock( &mutex1 );
}
```


counter=0
Thread2 obtains the mutex lock
counter++;
counter=1
Thread2 releases the mutex lock
Thread1 obtains the mutex lock
counter++;
counter=2
Thread1 releases the mutex lock

**The order may vary with time**

## Exclusive access to a set of shared objects

We can also serialize access to a set of shared objects.

```
pthread_mutex_lock( &mutex1 );
   Access shared object1;
   Access shared object2;
   …
pthread_mutex_unlock( &mutex1 );
```

Critical region

Threads will access {object1, object2, …} serially.

The code segment that resides between mutex_lock() and mutex_unlock() is called ‘critical region’.

Critical region is executed serially by the threads.

## Mutual exclusion: Pitfalls

```
//Thread1
void *do_one_thing(){
  ...
  pthread_mutex_lock(&mutex1);
  pthread_mutex_lock(&mutex2);
  ...
  ...
  pthread_mutex_lock(&mutex2);
  pthread_mutex_lock(&mutex1);
  ...
}
```

```
//Thread2
void *do_another_thing(){
  ...
  pthread_mutex_lock(&mutex2);
  pthread_mutex_lock(&mutex1);
  ...
  ...
  pthread_mutex_lock(&mutex1);
  pthread_mutex_lock(&mutex2);
  ...
}
```

Thread1 obtains mutex1

Thread1 waits to obtain mutex2

Thread2 obtains mutex2

Thread2 waits to obtain mutex1

Both threads get stalled indefinitely

Situation is known as Deadlock

## pthread_mutex_trylock()

Syntax is:

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

- Tries to lock a mutex object.
- If the mutex object is available, then it is locked and 0 is returned.
- Otherwise, the function call returns nonzero. It **will not wait** for the object to be freed.

## Avoiding mutex deadlock

- Thread tries to acquire mutex2
- and if it fails, then it releases mutex1 to avoid deadlock

```
...
pthread_mutex_lock(&mutex1);
// Now test if already locked
while ( pthread_mutex_trylock(&mutex2) ){
   // unlock resource to avoid deadlock
   pthread_mutex_unlock(&mutex1);
   ...
   // wait here for some time
   ...
   pthread_mutex_lock(&mutex1);
}
count++;
pthread_mutex_unlock(&mutex1);
pthread_mutex_unlock(&mutex2);
...
```

## Pthread Synchronization so far

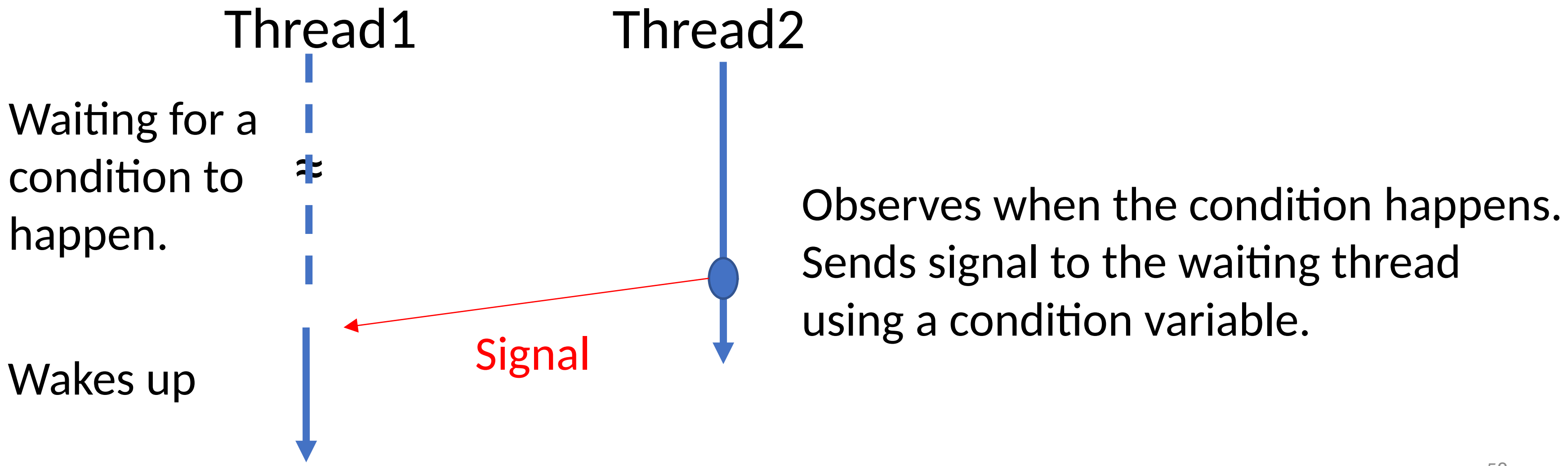
We have used the following synchronization functions

- Join: It is used to block the calling thread until the specified thread finishes.

- Mutual Exclusion: It is used to serialize access to shared data

## Condition variables

- A condition variable is used to synchronize threads based on the value of data (i.e., a condition).
- One thread waits until data reaches a particular value or until a certain event occurs.
- Then another active thread sends a signal when the event occurs.
- Receiving the signal, the waiting thread wakes up and becomes active.


Thread1
Thread2
Waiting for a condition to happen.
Observes when the condition happens. Sends signal to the waiting thread using a condition variable.
Signal
Wakes up

## Condition variables: Syntax

- A condition variable is a variable of type pthread_cond_t

- Creation and initialization

```
pthread_cond_t condition_cond = PTHREAD_COND_INITIALIZER;
```

- A thread goes to waiting state based on 'condition to happen' by:

```
pthread_cond_wait( &condition_cond, &condition_mutex );
```

Function takes two arguments: the condition variable and a mutex variable associated with the condition variable.

- Waking thread based on condition

```
pthread_cond_signal( &condition_cond );
```

```
int  count = 0;
#define COUNT_DONE  10
#define COUNT_HALT1  3
#define COUNT_HALT2  6
. . .
void *functionCount1(){
  for(;;){

     pthread_mutex_lock( &count_mutex );
     count++;
     printf("Counter value functionCount1: %d\n",count);
     pthread_mutex_unlock( &count_mutex );

     if(count >= COUNT_DONE) return(NULL);
   }
}
```

```
. . .
void *functionCount2(){
   for(;;){

      pthread_mutex_lock( &count_mutex );
      count++;
      printf("Counter value functionCount1: %d\n",count);
      pthread_mutex_unlock( &count_mutex );

      if(count >= COUNT_DONE) return(NULL);
    }
}
```

- functionCount1() and functionCount2() are run concurrently.
- They increment the shared variable ‘count’ serially
- The order in which the two functions increment ‘count’ is somewhat random.

Suppose, we want that only functionCount2() increments ‘count’ during the following condition

```
while( count >= COUNT_HALT1 && count <= COUNT_HALT2 )
```

```
pthread_mutex_t condition_mutex = PTHREAD_MUTEX_INITIALIZER;
void *functionCount1(){
  for(;;){
    pthread_mutex_lock( &condition_mutex );
    while( count >= COUNT_HALT1 && count <= COUNT_HALT2 ){
      pthread_cond_wait( &condition_cond, &condition_mutex );
    }
    pthread_mutex_unlock( &condition_mutex );

    pthread_mutex_lock( &count_mutex );
    count++;
    printf("Counter value functionCount1: %d\n",count);
    pthread_mutex_unlock( &count_mutex );

    if(count >= COUNT_DONE) return(NULL);
   }
}
```

- When functionCount1() sees for the first time that ‘count’ is in the mentioned range, it goes to wait state.
- pthread_cond_wait() **releases** the condition mutex.
- So, the condition variable can be used by the other thread now.

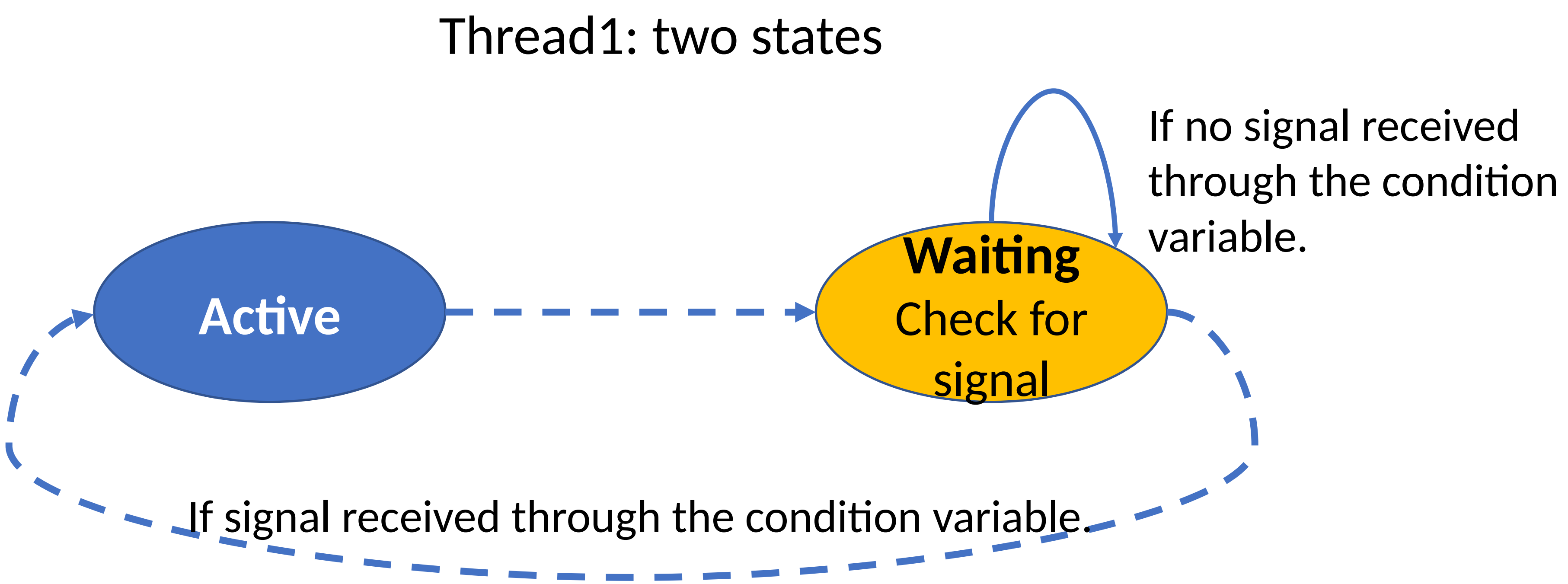
```
void *functionCount2(){
   for(;;){
      pthread_mutex_lock( &condition_mutex );
      if(count > COUNT_HALT2 ){
         pthread_cond_signal( &condition_cond );
      }
      pthread_mutex_unlock( &condition_mutex );

      pthread_mutex_lock( &count_mutex );
      count++;
      printf("Counter value functionCount2: %d\n",count);
      pthread_mutex_unlock( &count_mutex );

      if(count >= COUNT_DONE) return(NULL);
   }
}
```

Only functionCount2() increments ‘count’ from COUNT_HALT1 to COUNT_HALT2

After that, it ‘signals’ the waiting thread to wake up using the condition variable.

functionCount2() releases the condition mutex as the other thread will get it after waking up.

## Why do we need a mutex with a condition variable?


Thread1: two states
Active
Waiting
Check for signal
If no signal received through the condition variable.
If signal received through the condition variable.

- Presence of a signal is checked only in the ‘waiting’ state.
- If the signal arrives before Thread1 moves to the ‘waiting’ state, then the Thread1 will miss that
  ✉ Consequence: Thread1 will wait indefinitely

Condition mutex is used to ‘serialize’ the access of the condition variable in a proper way.