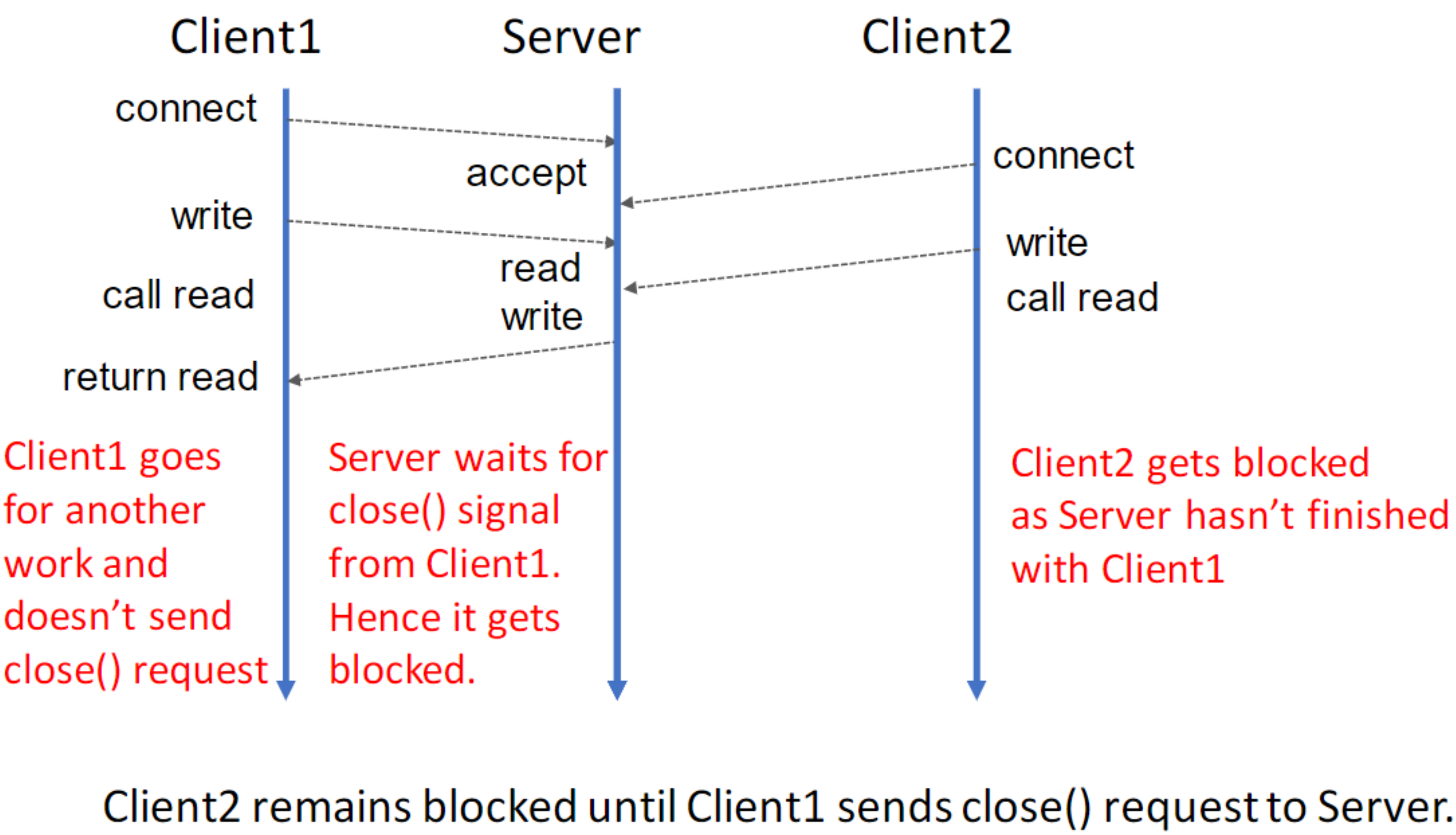
Concurrent Programming

Concurrent Programming

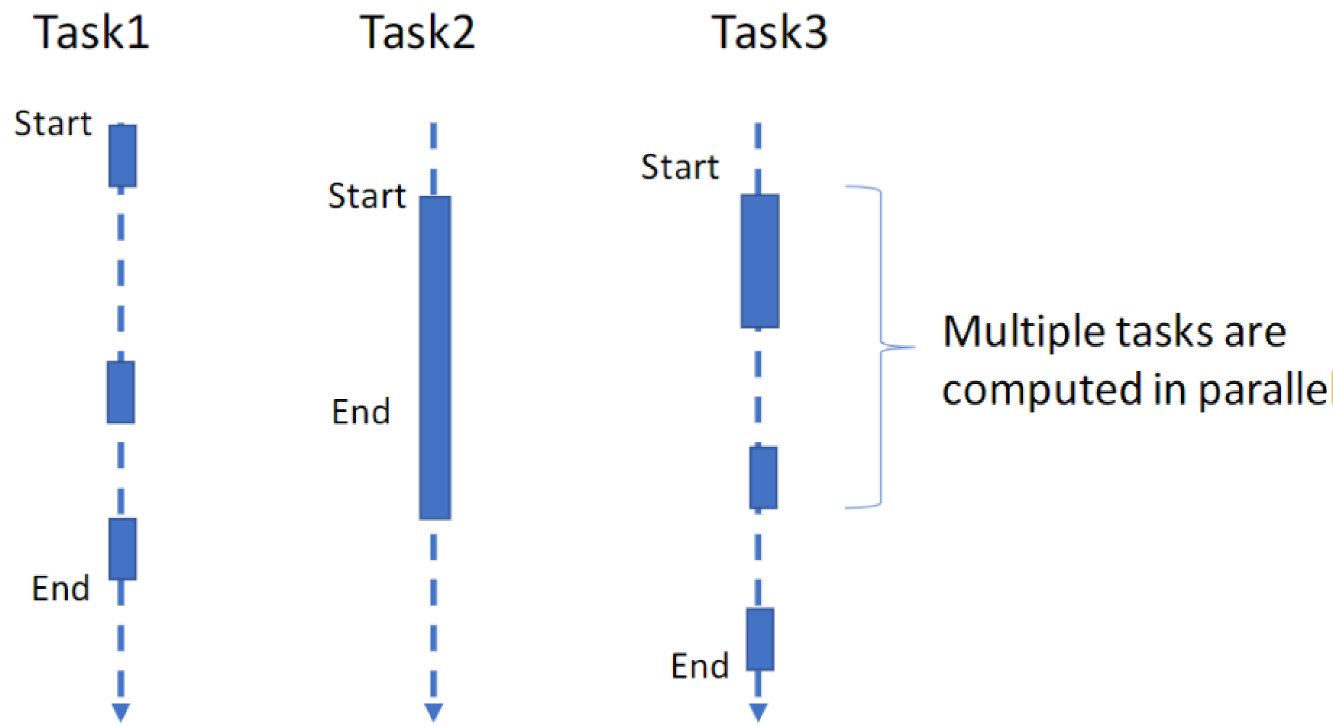
Example of Sequential Server



Solution, use concurrent server to server multiple clients so a client "cannot" block another

Example of Concurrent Server

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

7

Concurrency using Threads

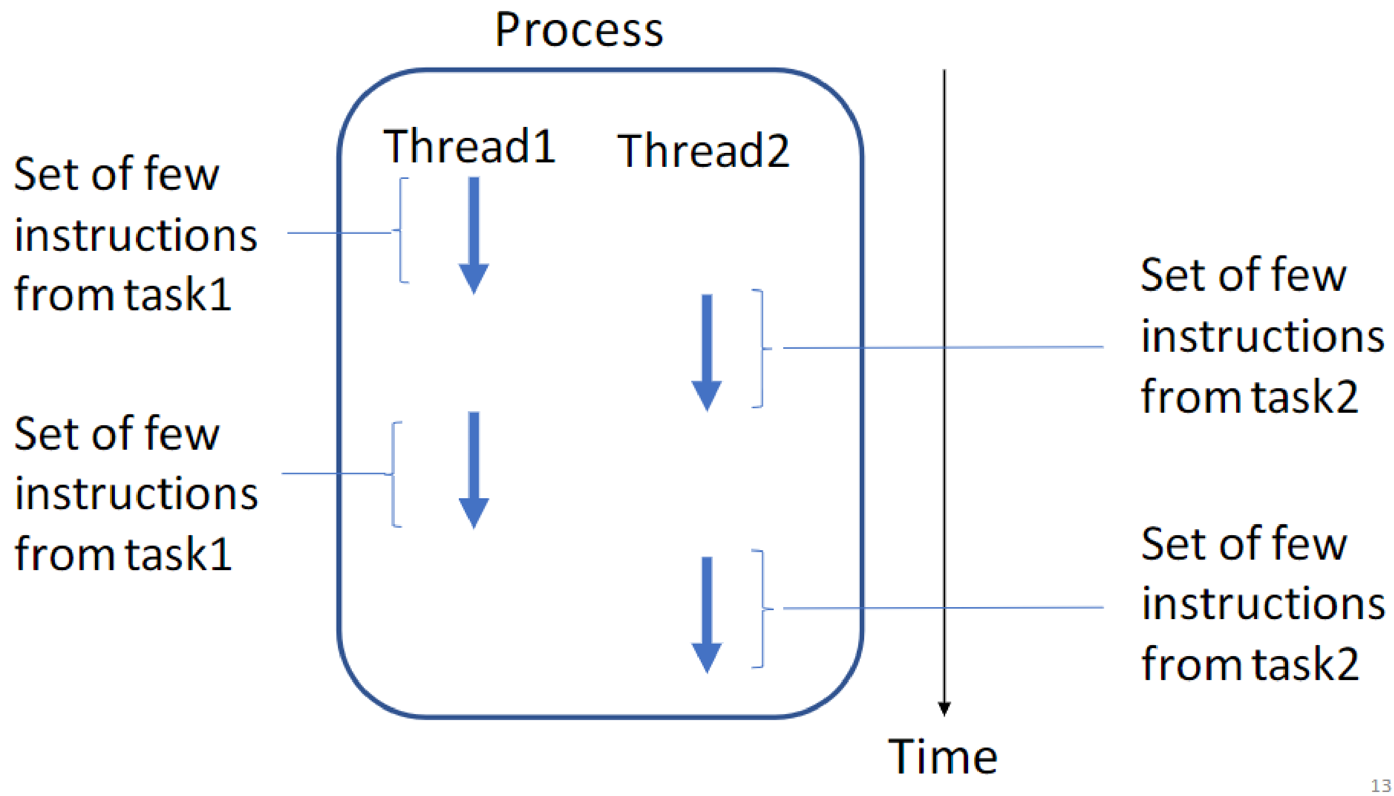
```
#include <stdio.h>

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 the above as a *sequence of instructions* executed serially.
In a concurrent perspective, we view program as a collections of tasks and if it is possible to execute some tasks at the same time, result is unchanged but overall execution time reduced

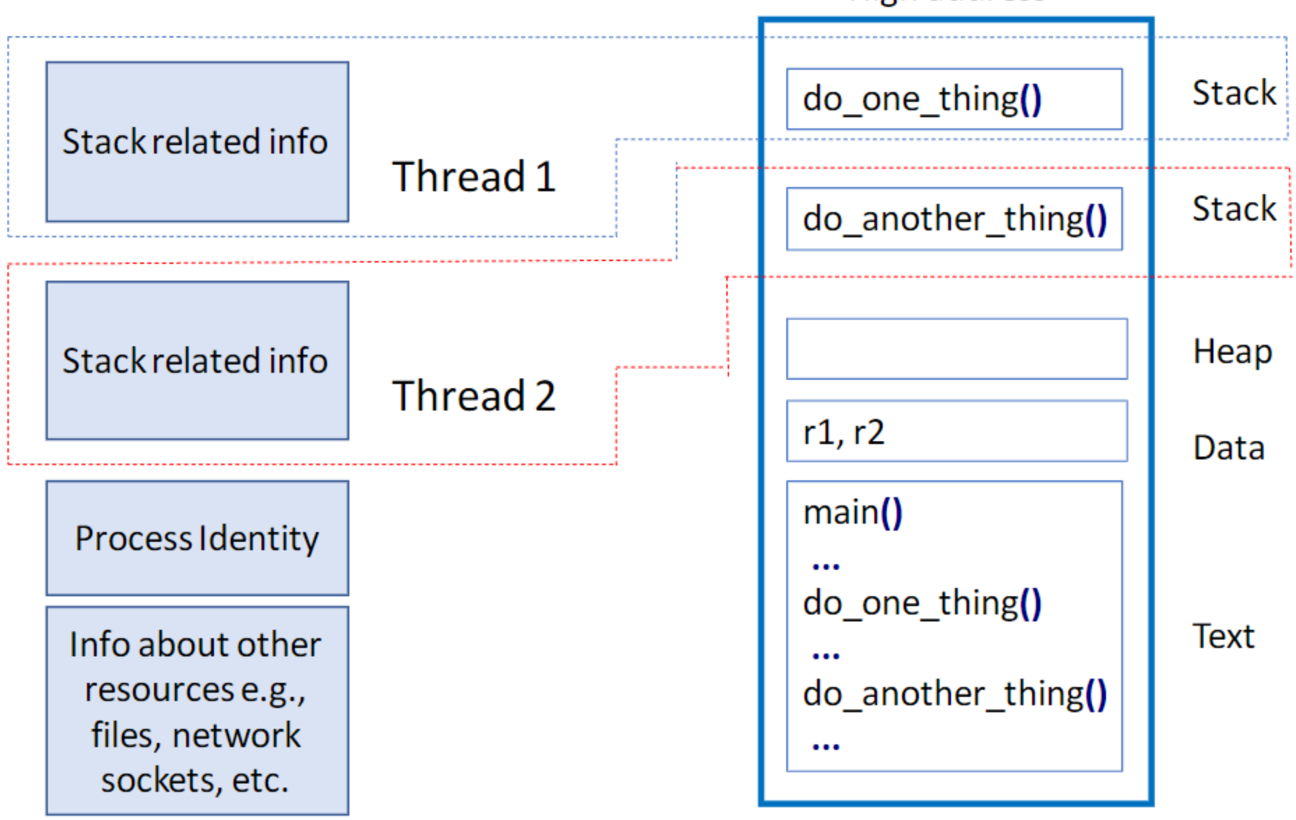
In the example, `do_one_thing` and `do_another_thing` can be viewed as a task.

Threads

> A thread is the smallest sequence of programmed instructions that can be independently managed by a scheduler



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.

Threads have their own *stacks* but *share heap, global data and opened files or sockets*

PThreads

To use pthreads on posix systems, use `#include <pthread.h>` and to spawn a thread, use `pthread_create()` which has the function signature:

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

`pthread_create()` returns 0 if creation successful, otherwise nonzero value returns (indicates error)

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

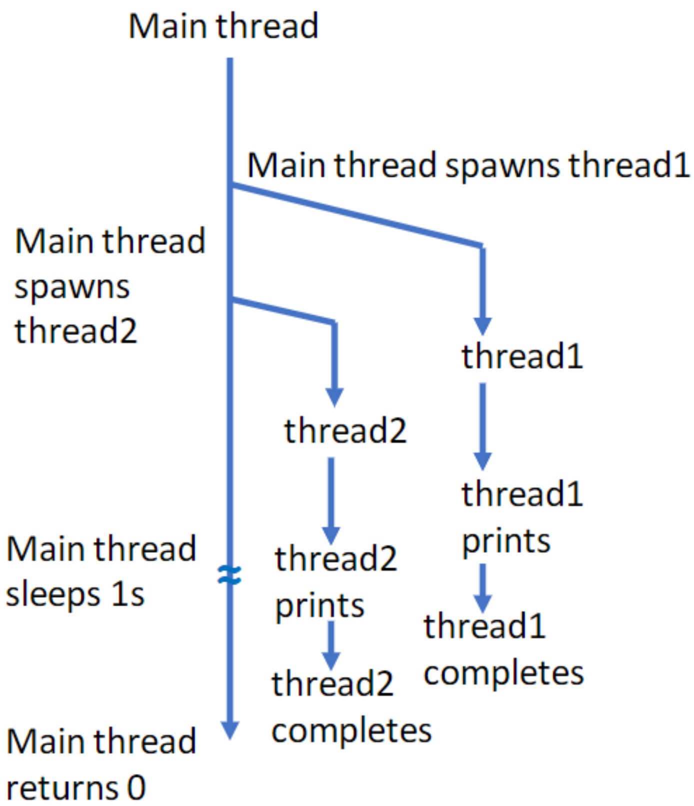
  T *foo (T *, T *);

  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);
}
```

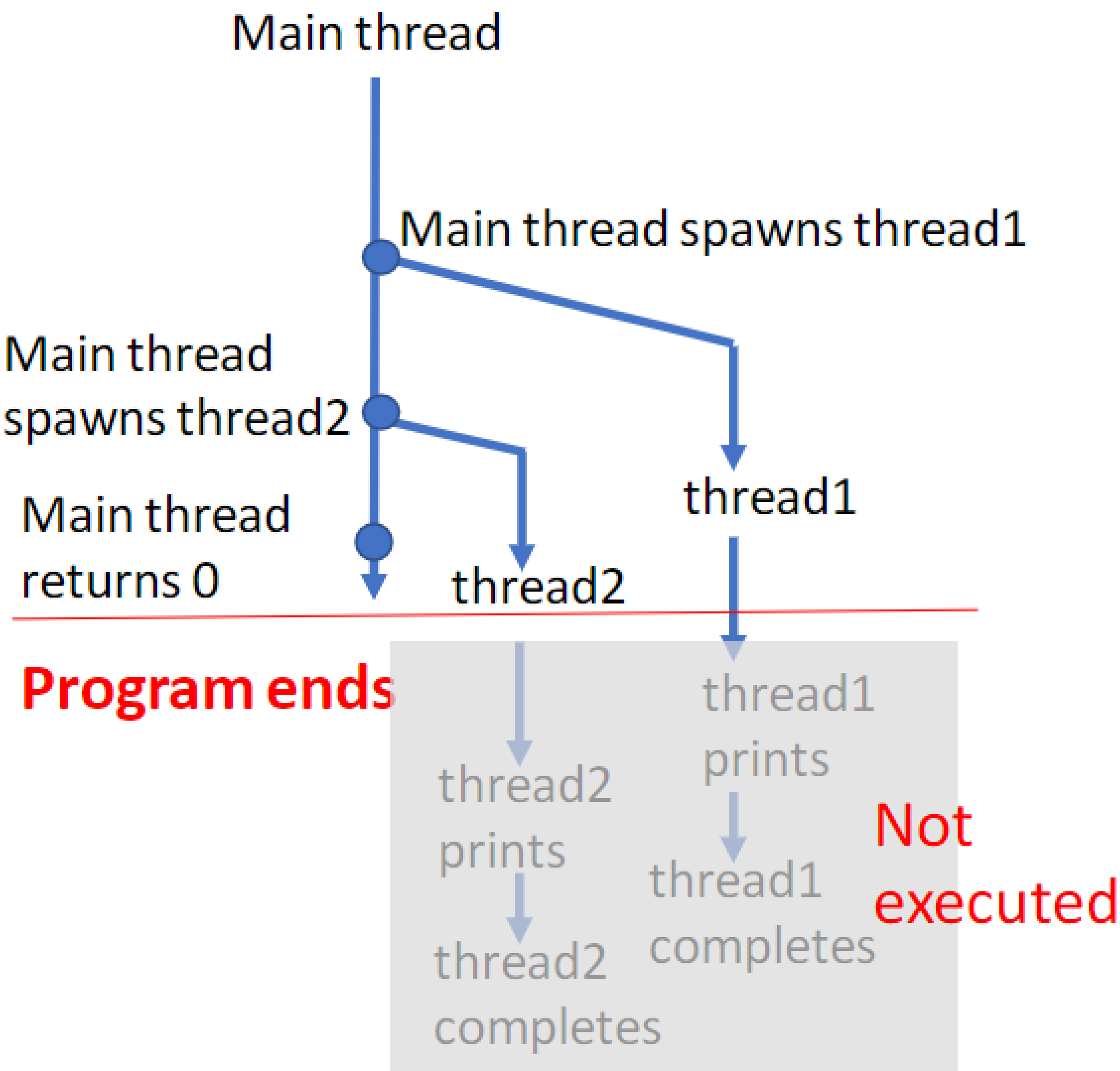
Program flow thread0.c




```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.

If main thread does not `sleep(1)` then most likely does not print anything since main thread would return before thread1 and thread2 could finish executing their own functions.



Shared data objects in concurrent system

Cooperation of threads leads to sharing of:

- *Global data objects*
- *Heap objects*
- *Files*

Lack of synchronization leads to wrong calculations and potential undefined behaviour
Synchronization makes sure some events happen in order

Synchronization methods in Pthreads

Pthread lib provides three sync mechanisms:

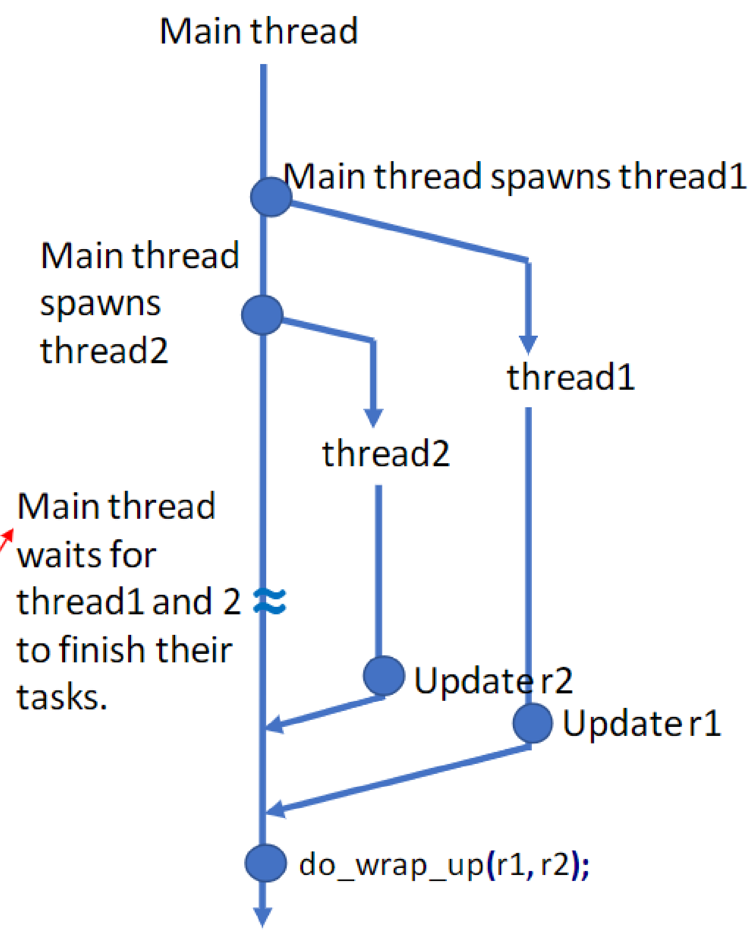
- *Joins*
- *Mutual exclusions (Mutexes)*
- *Condition variables*

Using `pthread_join()`

`pthread_join()` is a blocking function and syntax is:

```c
int pthread_join(
        pthread_t thread_id,     //ID of thread
        void** value_pntr        //address of function return value
);
```

```c
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;
}
```



There will be issues when threads update shared data

Example: Several threads update a shared data

```c
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:

counter=0

thread1: counter++; → counter=1

thread2: counter++; → counter=1

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

Program prints:
1
1

Data inconsistencies due to race conditions

Race conditions and prevention

- A race condition occurs when multiple threads perform operations on shared data but results depend on order they are performed in.
- Problem solved using *mutual exclusion*, so threads get exclusive access to shared data in turn
- Pthread lib offers *mutex* objects to enforce this for a variable or set of variables

Mutual exclusion in Pthread

- Syntax for mutex objects are:

```c
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
```

- Generally, mutexes are *global*
- To use a mutex for a set of variables, enclose the variable with a lock and unlock as:

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

The section between a lock and unlock is called the critical region

Deadlocks

```
//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

Using `pthread_mutex_trylock()`

- Syntax is:

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

- Tries to lock mutex and if available, locks it and returns 0

- Otherwise, returns nonzero but will not wait for mutex to be freed

Example

- 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);
...
```

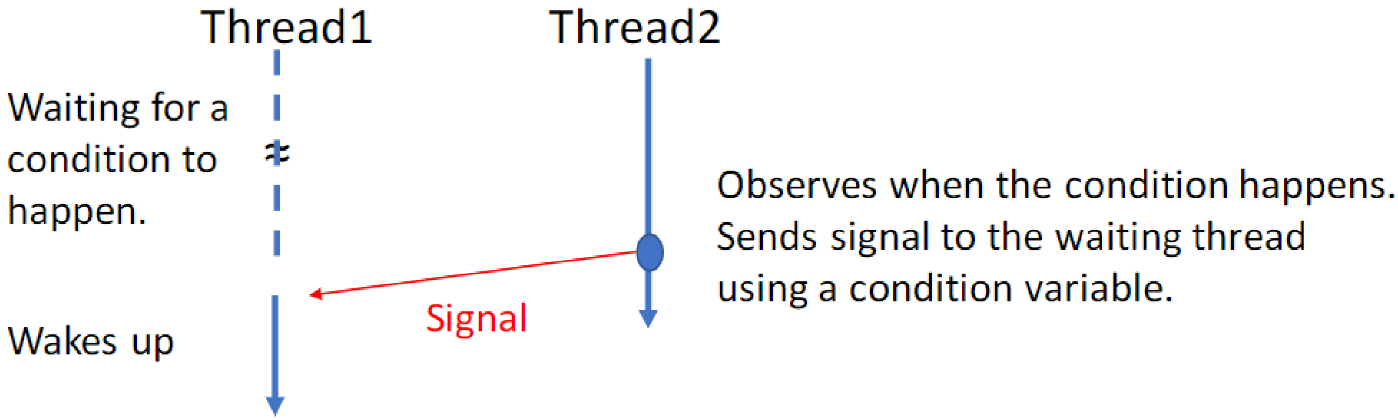
Using condition variables

- Condition variables can be used to synchronize threads based on value

- One thread waits until data reaches particular value or certain event occurs

- Another active thread sends signal when event occurs

- Receiving the signal, waiting thread wakes up



- Condition variable is of type `pthread_cond_t`

- Syntax is:

```
pthread_cond_t condition_cond = PTHREAD_COND_INITIALIZER;
```

- Thread goes to waiting state based on condition to happen by:

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

`pthread_cond_wait` takes two args, condition variable and mutex variable

- Waking thread based on condition:

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

An example would be:

```
int count = 0;
#define COUNT_DONE 10
#define COUNT_HALT1 3
#define COUNT_HALT2 6
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;
        }
}

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;
        }

}
```

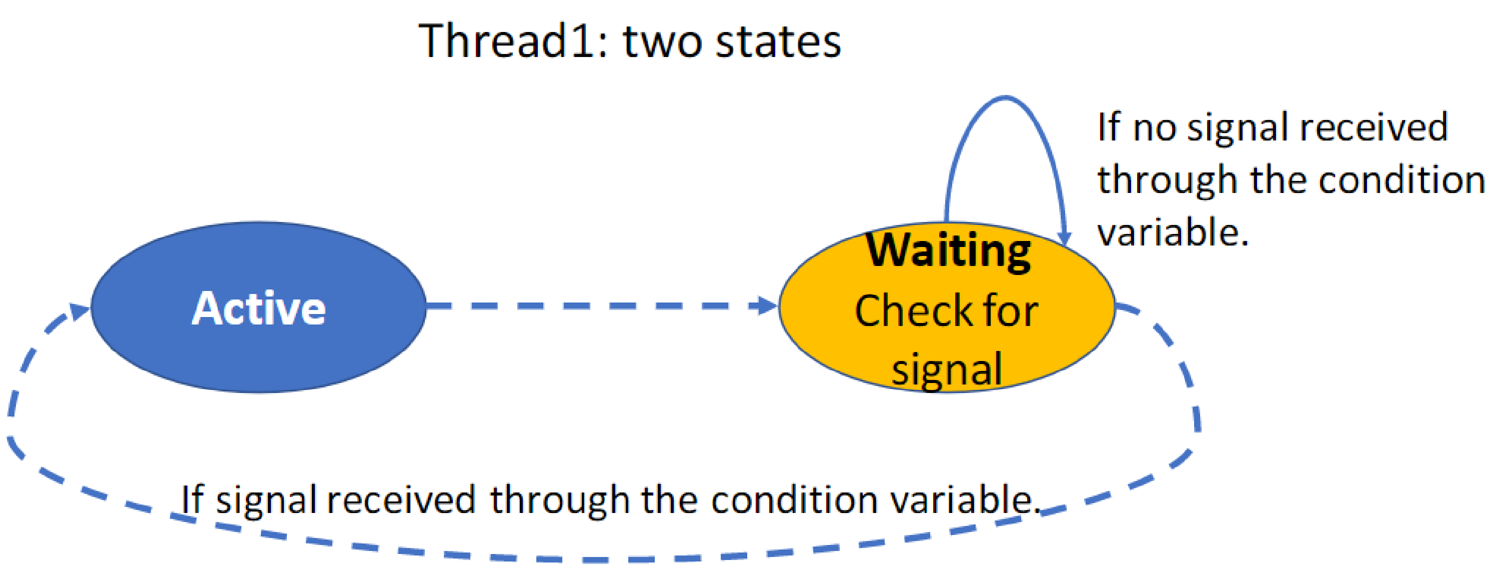
When functionCount1() sees count for the first time, out of range so goes to wait state

pthread_cond_wait() releases condition mutex so condition variable can be used by other thread

Only functionCount2() increments count from COUNT_HALT1 to COUNT_HALT2 and after signals the waiting thread to wake up using the condition variable.

functionCount2() releases the condition mutex

Need to mutex condition variable



- Presence of a signal is checked only in 'waiting' state.

- If signal arrives before Thread1 moves to 'waiting', then Thread1 will miss that so Thread1 will wait indefinitely

Condition mutex *serializes* access to condition variable properly

Concurrent operations on shared linked list

Consider sorted linked list with operations:

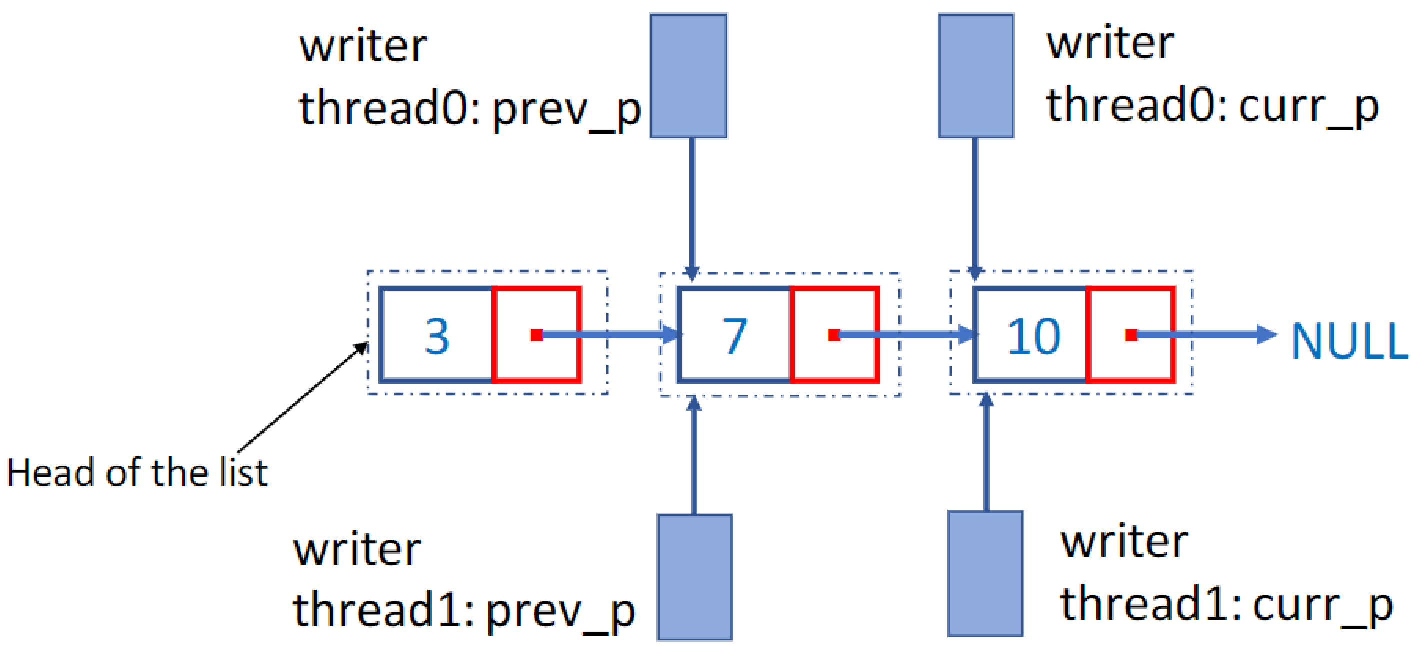
- Insert
- Delete
- Member

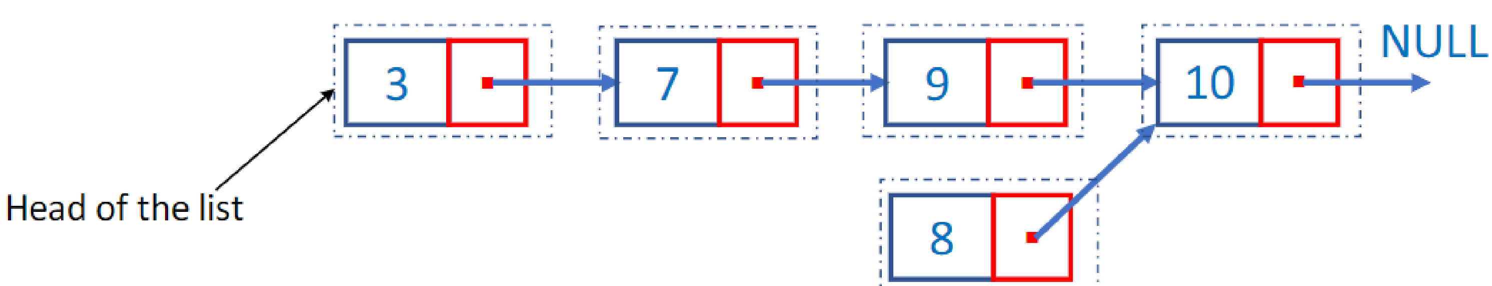
For concurrent threads to perform operations on a shared linked list, reads can occur conccurrently but writes cannot occur concurrently

---

Two concurrent operations:

- Thread0 wants to insert node value 8 in the list.
- Thread1 wants to insert node value 9 in the list.





Simply only allowing one thread access will mean reads fail to exploit parallelism but writes would be fine. *Defeats the purpose of multi-threading*

Locking each node (Granular access) would make solution complicated and slower. *major performance problem and complication*

Pthreads provide another solution, read-write locks

Using read-write locks

- Read-write locks used as:

```
pthread_rwlock_t lock = PTHREAD_RWLOCK_INITIALIZER;
```

- RW lock similar to a mutex excepts provides two lock functions:
    - for reading:

      ```
      pthread_rwlock_rdlock()
      ```

    - for read-write access:

      ```
      pthread_rwlock_wrlock()
      ```

- RW locks have only one unlock function:

  ```
  pthread_rwlock_unlock();
  ```

Goal is to allow multiple threads to read, but only one to write at a time

```
pthread_rwlock_rdlock(){
        If no other thread holds the lock, then get the lock.
        Else if other threads hold the read-lock, then get the lock.
        Else if another thread holds the write-lock, then wait.
}
```

```
pthread_rwlock_wrlock(){
        If no other threads hold the read or write lock, then get the lock.
        Else, wait for the lock.
}
```