

---

# Parallel programming

## Shared-memory

**Salvatore Orlando**

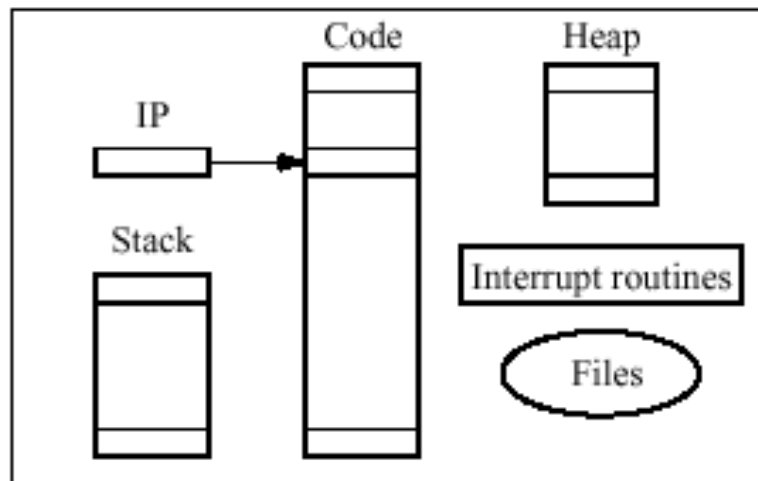
# Shared Memory Programming Models

---

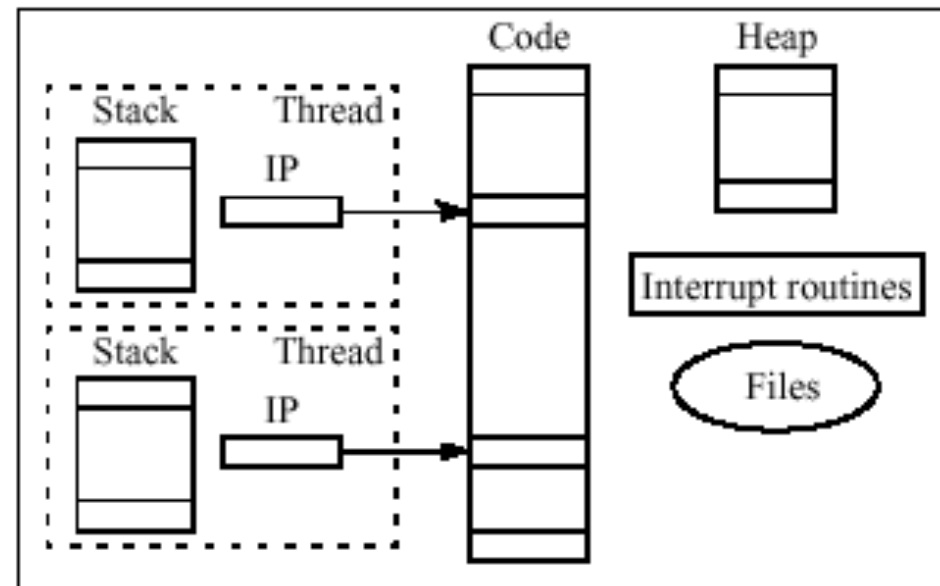
- Shared-Memory Programming models provide support for expressing concurrency and synchronization.
- **Process based models** assume that all data associated with a process is private, by default, unless otherwise specified
  - distinct page tables per each process that map virtual addresses to distinct physical memory pages
  - processes can include one or more threads
  - processes are **units of resource ownership**
    - program code, program counter, heap memory, program code, program counter, stack memory, stack pointer, file descriptors, virtual memory table, signal table, etc
- **Thread based models** assume that all memory is global
  - threads **share** the same physical address space (same page table)
  - at least a single thread per each process (usually more threads)
  - threads are **units of scheduling**

# Threads

- Each thread
  - has a separate flow of execution
  - is however the owner of
    - **private** program counter, stack memory, stack pointer, signal table pointer, signal table



Process



Thread

- Alternative name: “lightweight process”
- Besides **pthread** (**POSIX**), there exist other libraries
  - Solaris threads, Linux threads, DCE threads, Win32 e OS/2 threads, GNU Portable threads

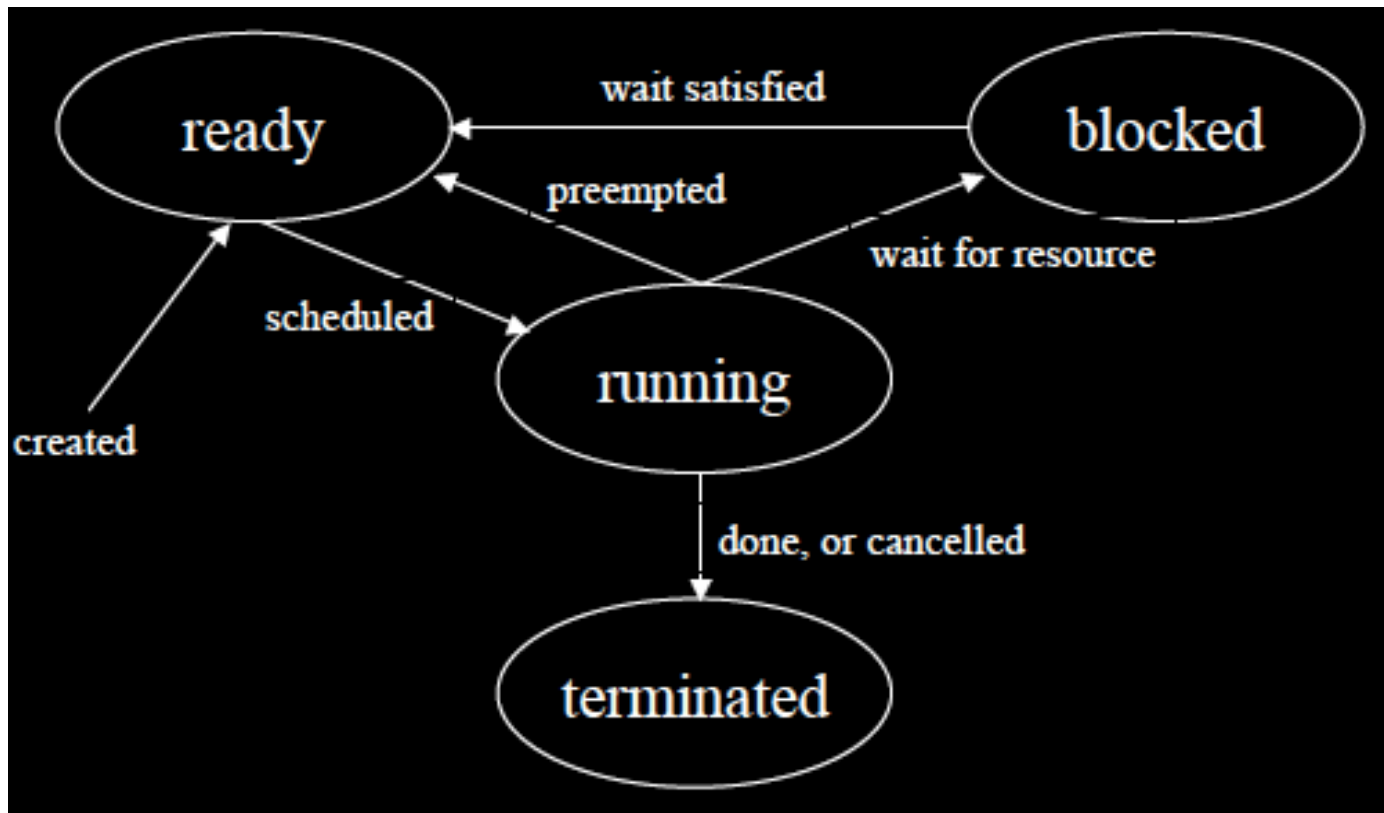
# pthread

---

- **POSIX.1c standard**
  - C language interface
- **All threads in a process**
  - Are peers
  - No explicit parent-child model
  - Exception: “main thread” holds process information
- **Advantages over processes**
  - Not swapped out for thread context switch
  - Threads read/write to shared variables for communication
    - cost of two memory accesses, without system call
  - Faster *context-switch*

# Life of a thread

---



# pthread management

---

- **pthread\_create**
  - create a thread
  - start execution of a function mapped to thread
- **pthread\_join**
  - wait for thread to finish
  - retrieve exit code from joined thread
- **pthread\_self**
  - returns the thread ID of the calling thread
- **pthread\_exit**
  - halt execution of calling thread
  - report exit code

# Hello world

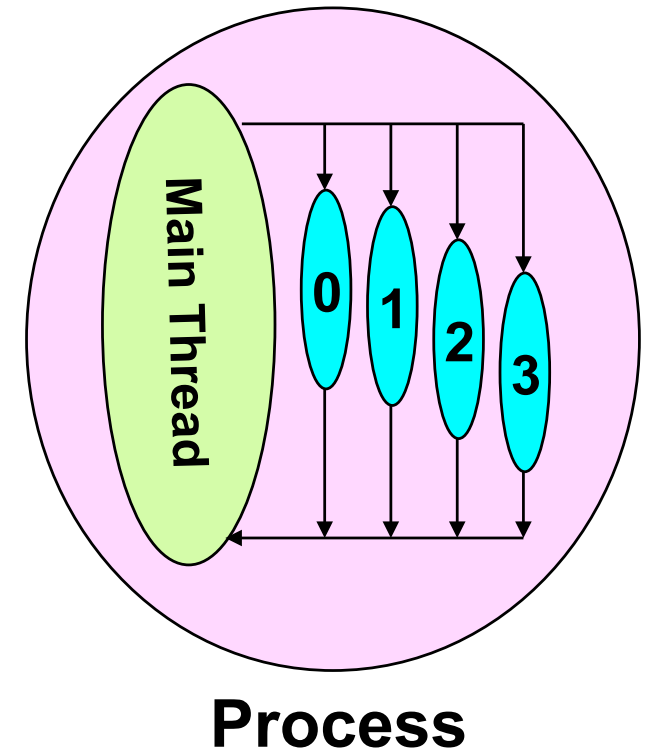
---

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 4

main()
{
    int i;
    pthread_t tid[NUM_THREADS];

    for (int i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], NULL, hello, NULL);
    for (int i = 0; i < NUM_THREADS; i++)
        pthread_join(&tid[i], NULL);
}

hello()
{
    printf("Hello, World\n");
}
```



**In this case, one may think of the thread as an instance of a function that returns before the function has finished executing**

# Parallelizing with pthreads

---

- Identify tasks for threading
- Identify the algorithms to use
- Identify how data will be accessed
  - What is logically global?
  - What is logically private/local to threads?
  
- Create function to encapsulate computation
  - may be function that already exists
  - use single parameter (C structure for multiple arguments)
  - follow `pthread_create` template for types

Example: `void *solve (void *arg)`

# Parallelizing with pthreads

---

- **Static Task Allocation**
  - Computations/Data are divided equally
  - based on number of threads and thread ID
  
- **Dynamic Task Allocation (work pool)**
  - Single thread “generates” tasks to be worked on [**Boss/Master thread**]
  - Other threads request new task when done with previous [**Worker threads**]
  - Boss sends kill signal at end
  - Workers terminate gracefully
  - Good model for unequal amounts of computation between tasks

# Data Parallelism (static approach)

---

- **Array sum (sequential):**

```
int array_sum(int n, int data[])
{
    int mid;
    int low_sum, high_sum;

    mid = n/2;
    low_sum = 0; high_sum = 0;

    // The following two "for" loops are independent
    for (int i = 0; i < mid; i++)
        low_sum = low_sum + data[i];
    for (int j = mid; j < n; j++)
        high_sum = high_sum + data[j];

    return low_sum + high_sum;
}
```

# Data Parallelism (static approach)

```
typedef struct {  
    int n, *data, mid;  
    int *high_sum, *low_sum;  
} args_block;
```

```
void sum_0(args_block *args)  
{  
    for (int i = 0; i < args->mid; i++)  
        *(args->low_sum) = *(args->low_sum) +  
            (args->data[i]);  
}
```

```
void sum_1(args_block *args)  
{  
    for (int j = args->mid; j < args->n; j++)  
        *(args->high_sum) = *(args->high_sum) +  
            args->data[j];  
}
```

```
int array_sum(int n, int data[]) {  
    int i, mid;  
    int low_sum, high_sum, *retval;  
    args_block args;  
    pthread_t threads[2];  
    mid = n/2;  
    args.n = n; args.data = data; args.mid = mid;  
    args.low_sum = &low_sum; args.high_sum = &high_sum;  
    pthread_create(&threads[0], NULL, (void *) sum_0, (void *) &args);  
    pthread_create(&threads[1], NULL, (void *) sum_1, (void *) &args);  
  
    for (i = 0; i < 2; i++) /* wait for threads to complete */  
        pthread_join(threads[i], (void **) &retval);  
  
    return low_sum + high_sum;  
}
```

attributes

Routine to execute

Thread args

# Mutex

---

- **Sharing data**
  - on read/write shared data we can have race condition
  - *mutual exclusion over a critical section*
  - `pthread_mutex_lock(MUTEX)`
  - `pthread_mutex_unlock(MUTEX)`
  - **MUTEX are binary semaphores**
  - **At any point of time, only one thread can lock a MUTEX. A lock is an atomic operation.**
- **Each thread entering a critical segment has first to try to get a lock. It goes ahead when the lock is granted**

```
pthread_mutex_t printlock =PTHREAD_MUTEX_INITIALIZER;  
...
```

```
pthread_mutex_lock(&printlock);  
< critical section >  
pthread_mutex_unlock(&printlock);
```

# Mutex is insufficient

---

- Often a thread has to check within a critical section (reading shared data) whether a given **event** happened or not
  - this event occurrence may depend on another thread
  - the other thread should access the same shared data in critical section => to signal (write) the occurrence the event
  - e.g. 2 threads: producer/consumer
    - the **waited event** for the consumer is the **presence of at least one data item** to consume in the shared buffer
    - the **waited event** for the producer is the **presence of at least one free slot** in the shared buffer

# Mutex is insufficient

---

- Naive solution with **active waiting**:
  - the thread continues accessing the critical section, with the hope that in the while the other thread is able to enter the same critical section thus signaling the occurrence of the waited event
- Optimal solution:
  - the thread *blocks* its execution if the event is not yet happened
  - the thread that generates the event awake the threads blocked and waiting for the event occurrence

# Example of Prod/Cons with only Mutex Lock

---

- **Constraints of the Producer/Consumer:**
  - the *producer thread* has not to rewrite shared slots of of the shared buffer which are not yet consumed
  - the *consumer thread* cannot extract data from any buffer slots until this is empty
  - producers and consumers has to work one at a time (mutual exclusion) on the shared buffer to void inconsistencies
  
- **Naive** example, with a single position buffer, that only exploits Mutex Locks

```
pthread_mutex_t task_queue_lock;  
int task_available;  
...  
main() {  
    ....  
    task_available = 0;  
    pthread_mutex_init(&task_queue_lock, NULL);  
    ....  
}
```

# Example of Prod/Cons with Mutex Lock

---

```
void *producer(void *producer_thread_data) {
    ....
    while (create_task(&my_task) != NULL) {
        while (1) {
            pthread_mutex_lock(&task_queue_lock);
            if (task_available == 0) {
                insert_into_queue(my_task);
                task_available = 1;
                pthread_mutex_unlock(&task_queue_lock);
                break;
            }
            else
                pthread_mutex_unlock(&task_queue_lock);
        }
    }
}
```

# Example of Prod/Cons with Mutex Lock

---

```
void *consumer(void *consumer_thread_data) {
    struct task my_task;    /* local data structure declarations */
    . . .
    while (!done()) {
        while (1) {
            pthread_mutex_lock(&task_queue_lock);
            if (task_available == 1) {
                extract_from_queue(&my_task);
                task_available = 0;
                pthread_mutex_unlock(&task_queue_lock);
                break;
            }
            else
                pthread_mutex_unlock(&task_queue_lock);
        }
        done = process_task(my_task);
    }
}
```

# Types of Mutexes

---

- Pthreads supports three types of mutexes - *normal*, *recursive*, and *error-check*.
- A **normal** mutex deadlocks if a thread that already has a lock tries a second lock on it.
- A **recursive** mutex allows a single thread to lock a mutex as many times as it wants.
  - It simply increments a count on the number of locks. A lock is relinquished by a thread when the count becomes zero.
- An **error check** mutex reports an error when a thread with a lock tries to lock it again (as opposed to deadlocking in the first case, or granting the lock, as in the second case)
- The type of the mutex can be set in the attributes object before it is passed at time of initialization.

# Overheads of Locking

---

- **Locks represent serialization points since critical sections must be executed by threads one after the other**
- **Encapsulating large segments of the program within locks can lead to significant performance degradation**
- **It is often possible to reduce the idling overhead associated with locks using an alternate function:**

```
int pthread_mutex_trylock (  
    pthread_mutex_t *mutex_lock);
```
- **It differs from a normal lock because it returns immediately an EBUSY error check if the the mutex is already locked**

# Condition Variables for Synchronization

---

- A **condition variable** allows a thread to block itself until specified data reaches a predefined state, which is checked by a **predicate**
- A condition variable is associated with this predicate. When the predicate becomes true, the condition variable is used to signal one or more threads waiting on the condition
- A single condition variable may be associated with more than one predicate
- A **condition variable** always has a **mutex associated** with it.
  - A thread locks this mutex and tests the predicate defined on the shared variable
  - If the predicate is not true, the thread waits on the condition variable associated with the predicate using the function `pthread_cond_wait`

# Condition Variables for Synchronization

- **Pthreads provides the following functions for condition variables:**

```
int pthread_cond_wait(pthread_cond_t *cond,  
pthread_mutex_t *mutex);
```

```
int pthread_cond_signal(pthread_cond_t *cond);
```

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

```
int pthread_cond_init(pthread_cond_t *cond,  
const pthread_condattr_t *attr);
```

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

**a blocked thread has to release the critical section**

**semantics: block +  
unlock(mutex)**

**unblocks at least one of  
the threads**

**semantics: the  
unblocked thread returns  
from its call to  
pthread\_cond\_wait()**

**The thread reobtain the  
mutex with which it called  
pthread\_cond\_wait()**

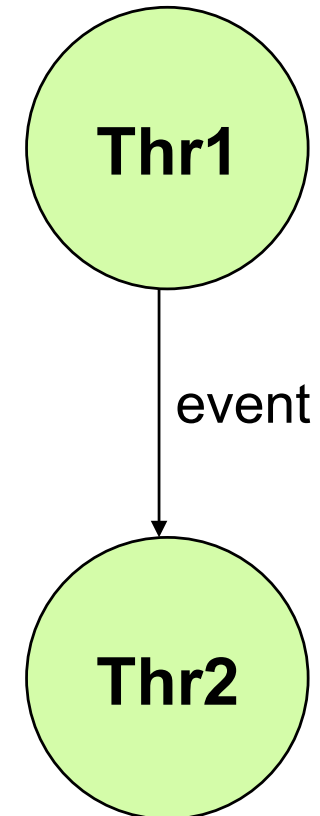
**If no sleeping threads to  
catch a signal exist, the  
signal is lost**

# Example with condition variables

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;  
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;  
...
```

```
thr1()  
{  
    pthread_mutex_lock(&mtx);  
    comp1(); event=1;  
    pthread_cond_signal(&cond);  
    pthread_mutex_unlock(&mtx);  
}
```

```
thr2()  
{  
    pthread_mutex_lock(&mtx);  
    while (! event)  
        pthread_cond_wait(&cond, &mtx);  
    comp2();  
    pthread_mutex_unlock(&mtx);  
}
```



# Prod/Cons with Condition Variables

---

```
pthread_cond_t cond_queue_empty, cond_queue_full;
pthread_mutex_t task_queue_cond_lock;
int task_available;
/* other data structures here */
main() {
    /* declarations and initializations */
    task_available = 0;
    pthread_init();

    /* producers wait for empty buffer */
    pthread_cond_init(&cond_queue_empty, NULL);

    /* consumers wait for full buffer */
    pthread_cond_init(&cond_queue_full, NULL);

    pthread_mutex_init(&task_queue_cond_lock, NULL);
    /* create and join producer and consumer threads */
    . . . . .
}
```

# Prod/Cons with Condition Variables

---

```
void *producer(void *producer_thread_data) {
    while (!done()) {
        create_task();
        pthread_mutex_lock(&task_queue_cond_lock);
        while (task_available == 1)
            pthread_cond_wait(&cond_queue_empty, &task_queue_cond_lock);
        insert_into_queue();
        task_available = 1;
        pthread_cond_signal(&cond_queue_full);
        pthread_mutex_unlock(&task_queue_cond_lock);
    }
}
```

# Prod/Cons with Condition Variables

---

```
void *consumer(void *consumer_thread_data) {
    while (!done()) {
        pthread_mutex_lock(&task_queue_cond_lock);
        while (task_available == 0)
            pthread_cond_wait(&cond_queue_full, &task_queue_cond_lock);
        my_task = extract_from_queue();
        task_available = 0;
        pthread_cond_signal(&cond_queue_empty);
        pthread_mutex_unlock(&task_queue_cond_lock);
        process_task(my_task);
    }
}
```

# Composite Synchronization Constructs

---

- Higher-level synchronization constructs can be built using basic synchronization constructs, e.g. barrier
- As in MPI, a barrier holds a thread until all threads participating in the barrier have reached it.
- Barriers can be implemented using a counter, a mutex and a condition variable.
- A single integer is used to keep track of the number of threads that have reached the barrier.
- If the count is less than the total number of threads **N**, the threads execute a condition **wait**.
- The last thread entering (and setting the count to the number of threads) wakes up all the threads using a **condition broadcast**.

# Barrier

---

```
typedef struct {
    pthread_mutex_t count_lock;
    pthread_cond_t ok_to_proceed;
    int count;
} mylib_barrier_t;

void mylib_init_barrier(mylib_barrier_t *b) {
    b -> count = 0;
    pthread_mutex_init(&(b -> count_lock), NULL);
    pthread_cond_init(&(b -> ok_to_proceed), NULL);
}
```

# Barrier

---

```
void mylib_barrier (mylib_barrier_t *b, int num_threads) {  
  
    pthread_mutex_lock(&(b -> count_lock));  
    b -> count ++;  
  
    if (b -> count == num_threads) {  
        b -> count = 0;  
        pthread_cond_broadcast(&(b -> ok_to_proceed));  
    }  
    else  
        pthread_cond_wait(&(b -> ok_to_proceed), &(b -> count_lock));  
    pthread_mutex_unlock(&(b -> count_lock));  
}
```

# Barrier

---

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
int pthread_barrier_destroy(pthread_barrier_t *barrier);
int pthread_barrier_init(pthread_barrier_t *restrict barrier,
                        const pthread_barrierattr_t *restrict attr,
                        unsigned count);
int pthread_barrierattr_destroy(pthread_barrierattr_t *attr);
int pthread_barrierattr_init(pthread_barrierattr_t *attr);
```

- ... with the usual semantic ...
- **unsigned count**: is used to specify how many thread must be waiting at the barrier before this is released.

## Further details

---

- <https://computing.llnl.gov/tutorials/pthreads/>