Today

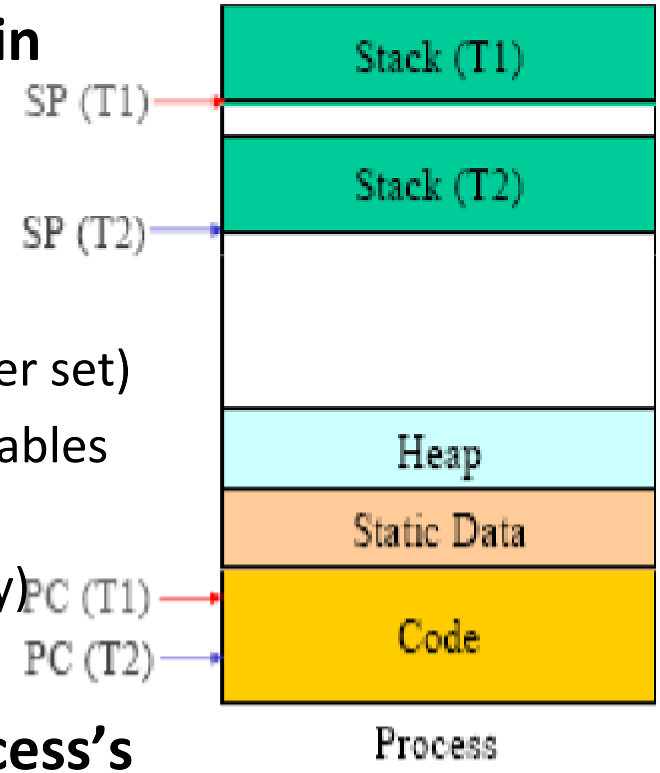
- Threads vs Processes
- Thread Features and types
- Hello world with threads
- Comparison and applications
- Concurrent Programming – Intro
- Sharing among threads

Motivation

- **Processes are expensive to create**
  - Address space for code and data pages
  - OS descriptors of resources allocated
  - State
  - Registers, etc …
- **It takes quite a bit of time to switch between processes**
- **Communicating between processes is costly because most communication goes through the OS**
  - Files
  - Pipes
  - Message Queues
  - Shared memory

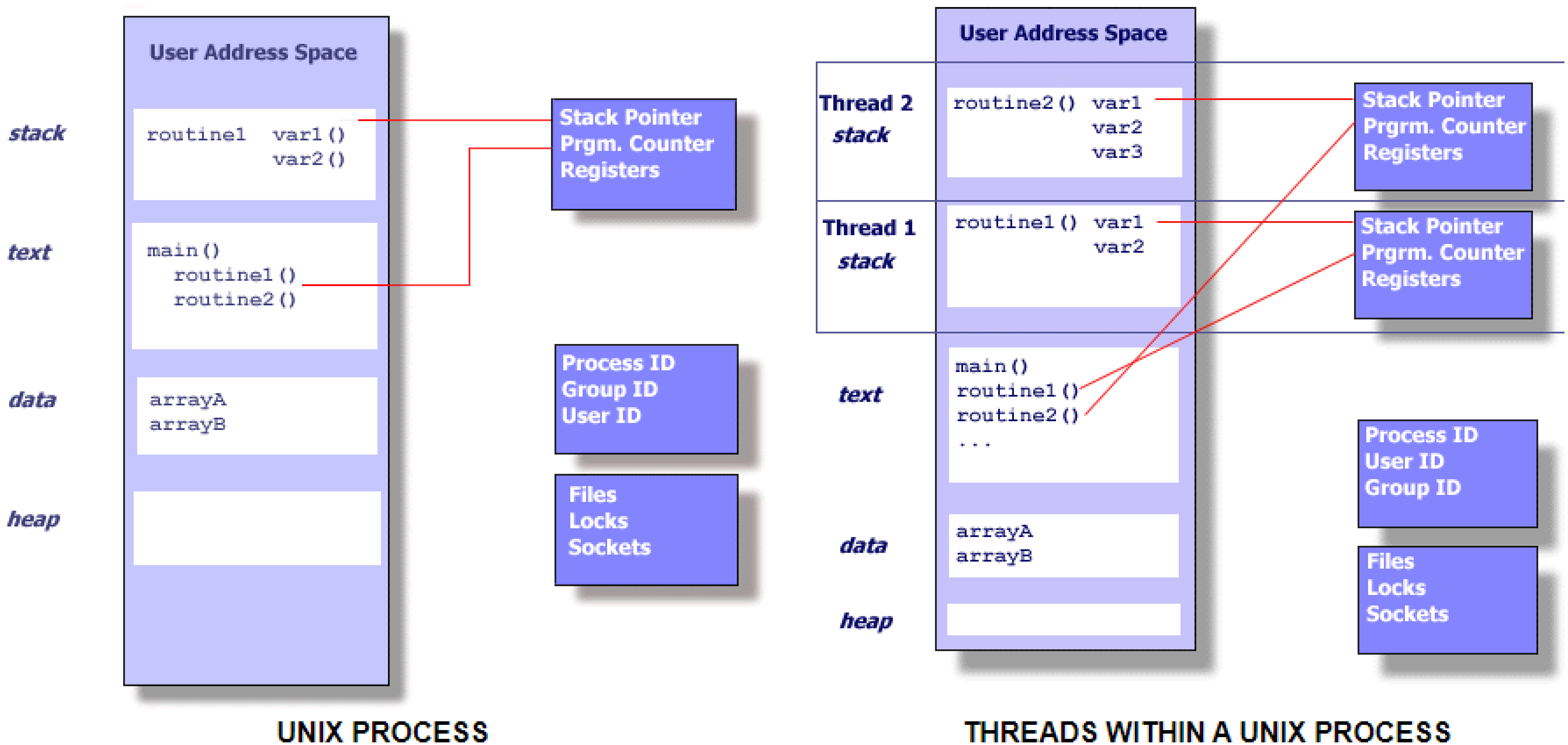
Threads

- **A thread is the smallest schedulable unit in multithreading**
- **A thread in execution works with**
  - Thread ID
  - Registers (program counter and working register set)
  - Stack (for procedure call parameters, local variables etc.)
  - Scheduling properties (such as policy or priority)
  - Set of pending and blocked signals
- **A thread shares with other threads a process's (to which it belongs to)**
  - Instruction and Data Sections
  - Permissions
  - Other resources, such as files



Process with 2 threads

Threads in the VM

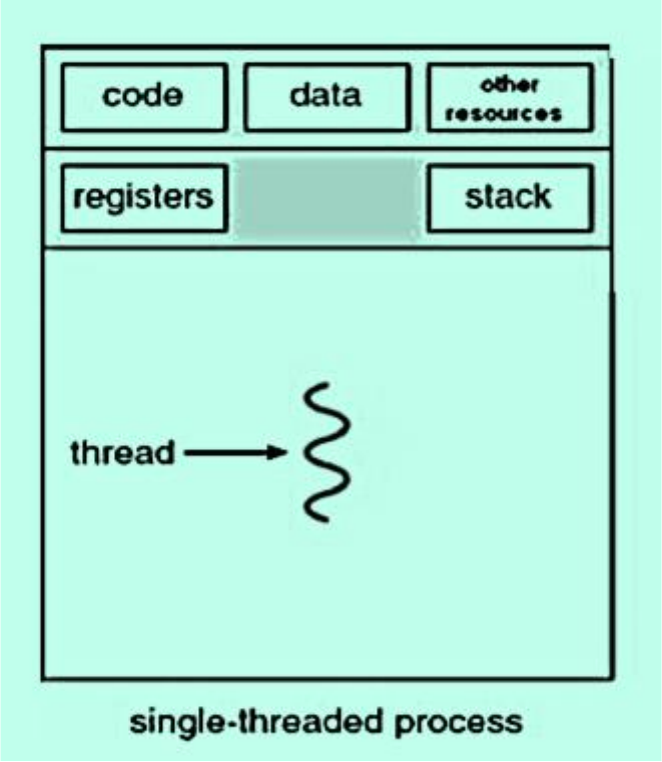


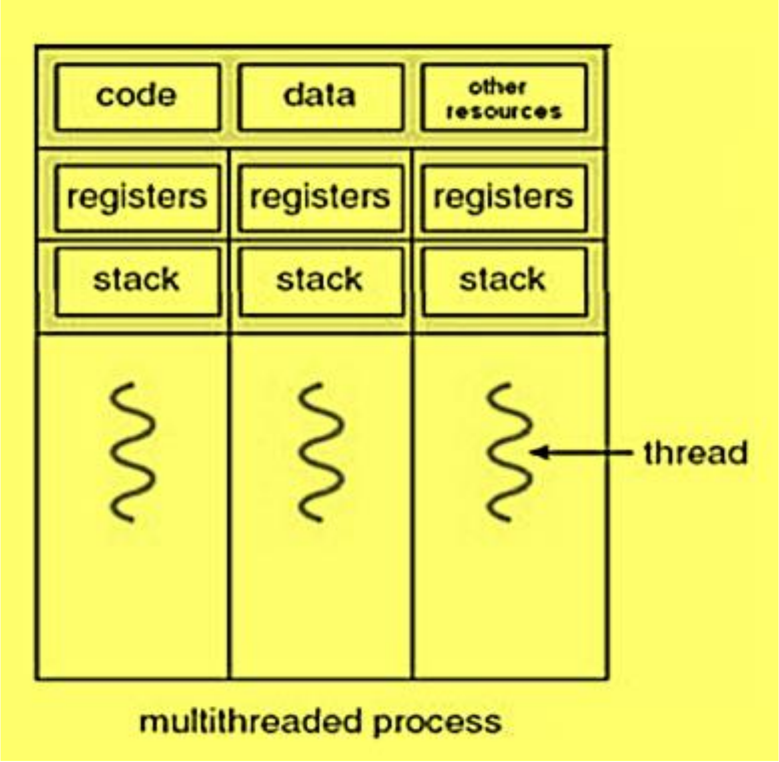
https://computing.llnl.gov/tutorials/pthreads/

Threads vs. Processes

- **How threads and processes are similar**
  - Each has its own logical control flow
  - Each can run concurrently with others (possibly on different cores)
  - Each is context switched
- **How threads and processes are different**
  - Threads share all code and data (except local stacks)
    - Processes (typically) do not
  - Threads are somewhat less expensive than processes
    - Process control (creating and reaping) twice as expensive as thread control
    - Linux numbers:
      - ~20K cycles to create and reap a process
      - ~10K cycles (or less) to create and reap a thread

Difference between Single vs Multithread Process





- **A process by itself can be viewed a single thread and is traditionally known as a heavy weight process**

Today

- Threads vs Processes
- **Thread Features and types, functions**
- Hello world with threads
- Comparison and applications
- Concurrent Programming – Intro
- Sharing among threads

Thread Features

1. Exists within a process and uses the process resources
2. Has its own independent flow of control as long as its parent process exists and the OS supports it
3. Independently “schedulable“
4. May share the process resources with other threads
5. Dies if the parent process dies - or something similar
6. Is "lightweight" because most of the overhead has already been accomplished through the creation of its process.

Caveats

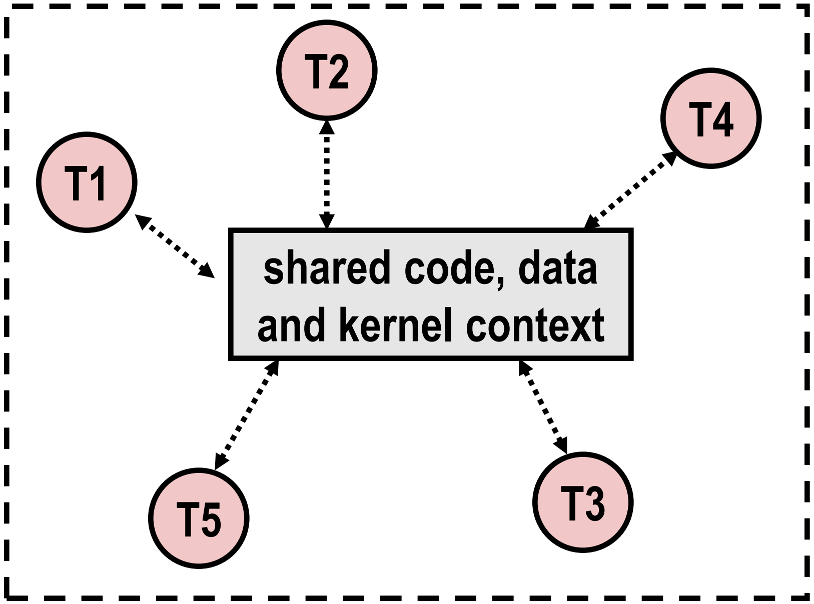
Because threads within the same process share resources:

- **Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.**
- **Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.**
- **If one of the threads crash, whole process may crash**

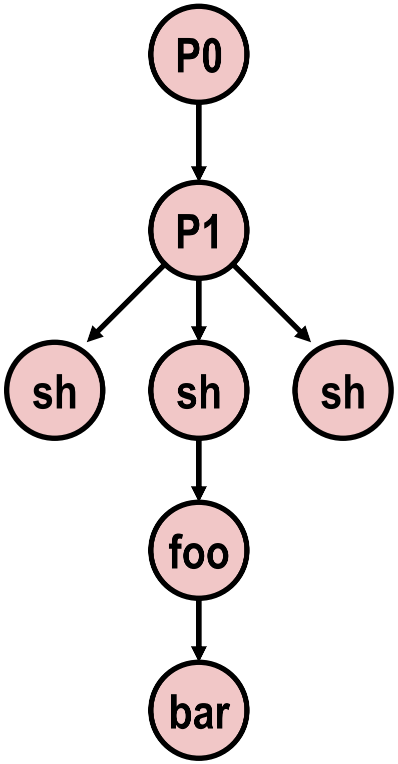
Logical View of Threads

- **Threads associated with process form a pool of peers**
  - Unlike processes which form a tree hierarchy

Threads associated with process foo



Process hierarchy



Implementation

- **Threads can be implemented in**

  - ***Kernel-space:*** *Involves system calls, and library support*
  - ***User-space:*** *API functions implemented in user level*

  - **POSIX PThreads** - may be provided as either a user or kernel library, as an extension to the POSIX standard.
  - PThreads are available on Solaris, Linux, Mac OSX, Tru64 UNIX, and via public domain shareware for Windows.

Linux Threads – Kernel Level System Call

- Created through:
  - **clone (fn, stack, flags)**

- Where
  - **fn** specifies function to be executed by the new thread or process
  - **stack** points to the stack it will use
  - **flags** is a set of flags specifying various options (**CLONE_VM or CLONE_THREAD** for threads. If it is missing a regular process will be created)
    - fork() also calls clone internally
  - Threads are also called lightweight processes in kernel space .

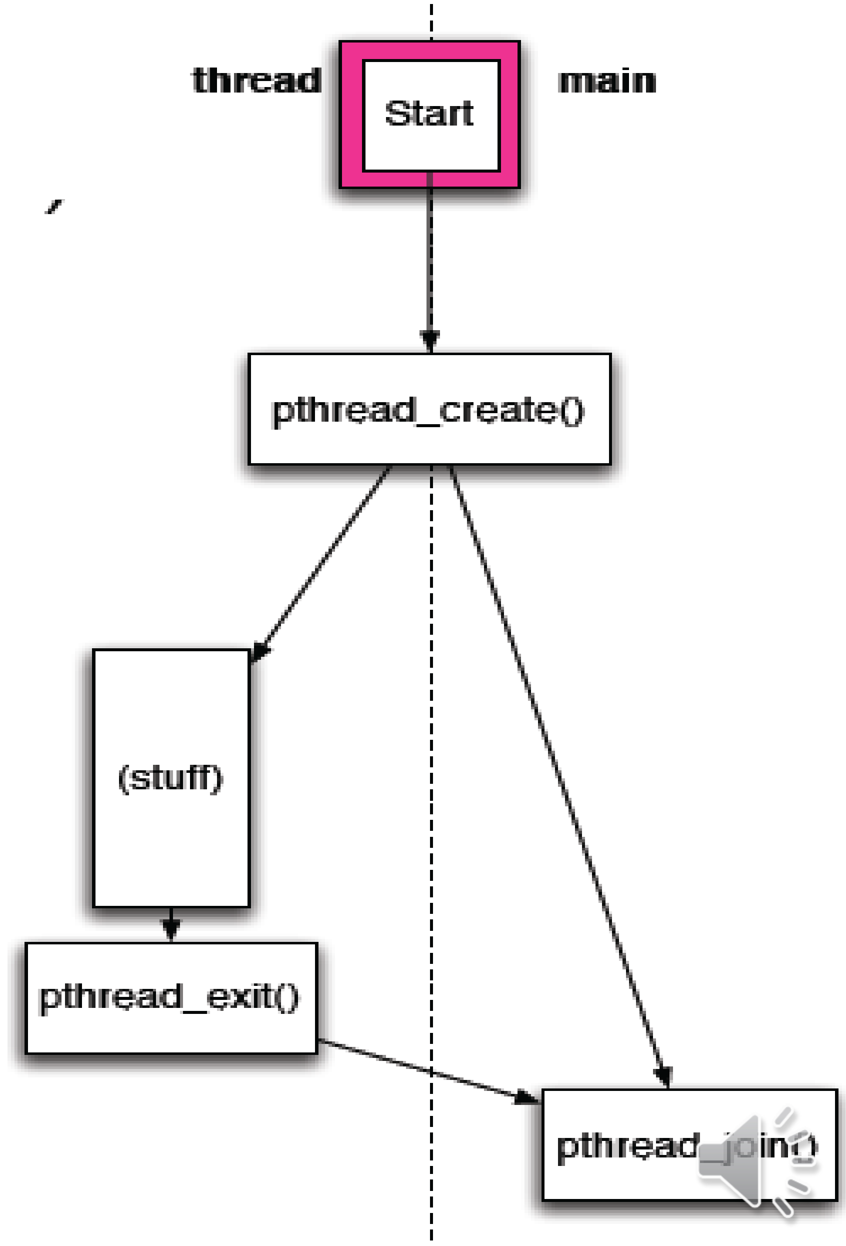
Linux Threads – User Level with Kernel Support

- POSIX threads, or ***PThreads***, started as pure user-level threads managed by the POSIX thread library
- IEEE standard, supported by most vendors
  - Emerged as standard thread API
- Gained later ***some kernel support***
- Ported to various Unix and Windows systems
- Function names start with **pthread_**
- Calls tend to have a complex syntax

Posix Threads (Pthreads) Interface

- ***Pthreads:*** **Standard interface for ~60 functions that manipulate threads from C programs**
  - Creating and reaping threads
    - `pthread_create()`
    - `pthread_join()`
  - Determining your thread ID
    - `pthread_self()`
  - Terminating threads
    - `pthread_cancel()`
    - `pthread_exit()`
    - `exit()` [terminates all threads]
  - Synchronizing access to shared variables (we will use semaphores instead)
    - `pthread_mutex_init`
    - `pthread_mutex_[un]lock`

Thread lifecycle



Main

✓ **pthread_create()** (create thread)

✓ wait for thread() to finish via **pthread_join()**

Thread

✓ begins at function pointer

✓ runs until **pthread_exit()**

pthread_create()

- The pthread_create function starts a new thread in the calling process.

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

- Where,
  - ► thread is a pthread library structure holding thread info
  - ► attr is a set of attributes to apply to the thread(Usually NULL)
  - ► start_routine is the thread function pointer
  - ► arg is an opaque data pointer to pass to thread
- Returns 0 on success

`pthread_join()`

- The pthread_join function waits for the thread specified by thread to terminate.

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

- Where,
  - thread is a pthread library structure holding thread info
  - *retval* is a double pointer return value. Usually NULL if not will return the exit status

- Will return 0 if successful
- Similar to waitpid for fork. There is no wait alternative for Pthreads

pthread_exit()

- The pthread_exit function terminates the calling thread and returns a value

```
void pthread_exit(void *retval);
```

- Where,
  - retval is a pointer to a return value

    Note: better be dynamically allocated because the thread stack will go away when the thread exits

Today

- Threads vs Processes
- Thread Features and types
- **Hello world with threads**
- Comparison and applications
- Concurrent Programming – Intro
- Sharing among threads

The Pthreads "hello, world" Program

```c
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"
void *thread(void *vargp);

int main()
{
   pthread_t tid;
   Pthread_create(&tid, NULL, thread, NULL);
   Pthread_join(tid, NULL);
   exit(0);
}
```

Thread ID

Thread attributes (usually NULL)

Thread routine

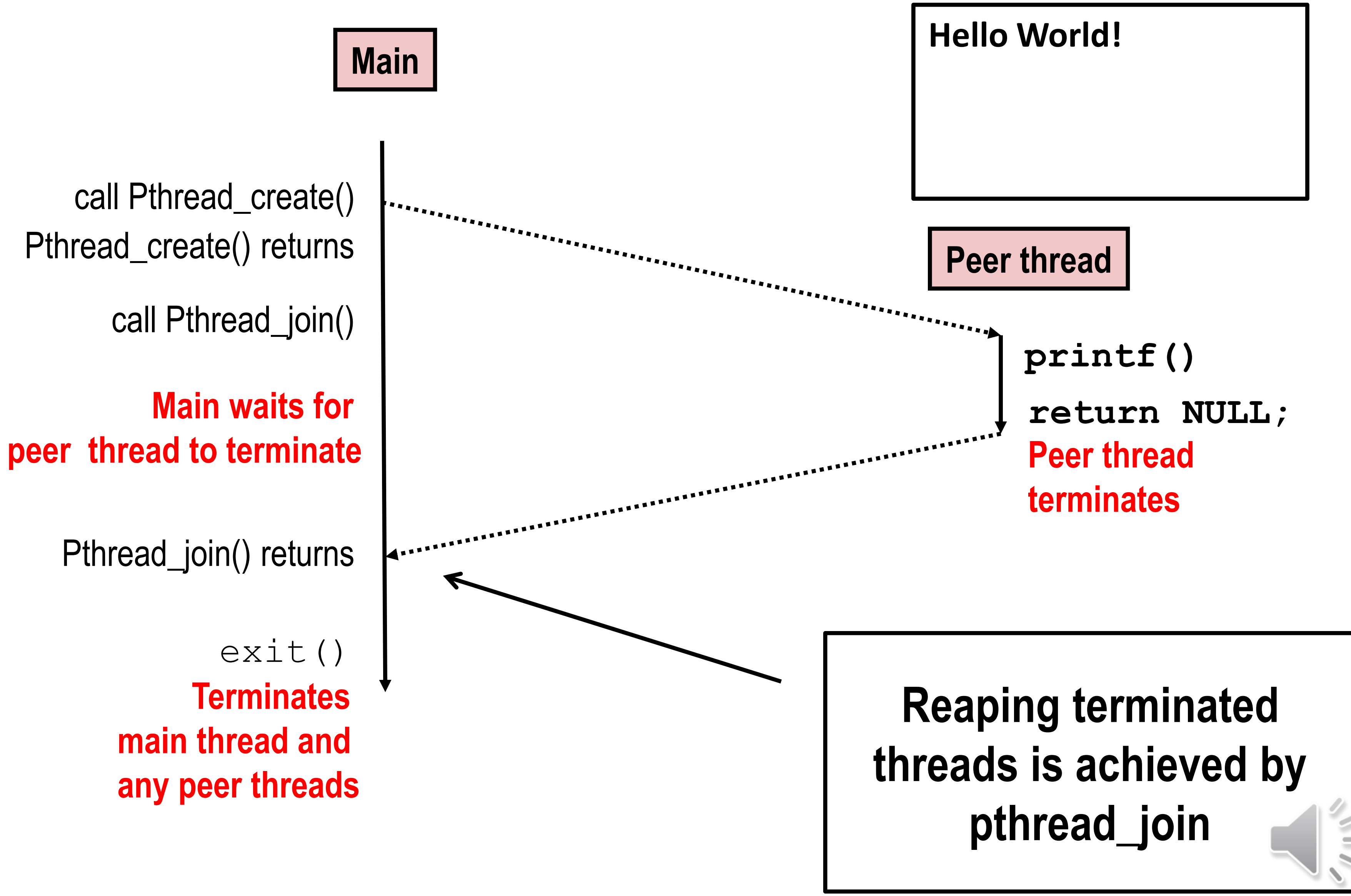
*Thread arguments (void *p)*

*Return value (void **p)*

```c
void *thread(void *vargp) /* thread routine */
{
   printf("Hello, world!\n");
   return NULL;
}
```

hello.c

Execution of Threaded “hello, world”



Putting it all together ...

```
typedef struct {
    int num;
    const char *str;
} MY_STRUCT;

void * thread_function( void * arg ) {
    MY_STRUCT *val = (MY_STRUCT *)arg; // Cast to expected type
    printf( "Thread %lx has vaules %x,%s]\n", pthread_self(), val->num, val->str );
    pthread_exit( &val->num );
}

int main( void ) {
    MY_STRUCT v1 = { 0x12345, "Val 1" };
    MY_STRUCT v2 = { 0x54312, "Val 2" };
    pthread_t t1, t2;
    printf( "Starting threads\n" );
    pthread_create( &t1, NULL, thread_function, (void *)&v1 );
    pthread_create( &t2, NULL, thread_function, (void *)&v2 );
    pthread_join( t1, NULL );
    pthread_join( t2, NULL );
    printf( "All threads returned\n" );
    return( 0 );
}
```

Putting it all together ...

```
typedef struct {
    int num;
    const char *str;
} MY_STRUCT;

void * thread_function( void * arg ) {
    MY_STRUCT *val = (MY_STRUCT *)arg; // Cast to expected type
    printf( "Thread %lx has vaules %x,%s]\n", pthread_self(), val->num, val->str );
    pthread_exit( &val->num );
}

int main( void ) {
    MY_STRUCT v1 = { 0x12345, "Val 1" };
    MY_STRUCT v2 = { 0x54312, "Val 2" };
    pthread_t t1, t2;
    printf( "Starting threads\n" );
    pthread_create( &t1, NULL, thread_function, (void *)&v1 );
    pthread_create( &t2, NULL, thread_function, (void *)&v2 );
    pthread_join( t1, NULL );
    pthread_join( t2, NULL );
    printf( "All threads returned\n" );
    return( 0 );
}
```

```
$ ./concurrency
Starting threads
Thread 7f51c3e05700 has vaules 54312,Val 2]
Thread 7f51c4606700 has vaules 12345,Val 1]
All threads returned
$
```

Today

- Threads vs Processes
- Thread Features and types
- Hello world with threads
- **Comparison and applications**
- Concurrent Programming – Intro
- Sharing among threads

When to use processes

- **When fault tolerance needed: If child process crashes parent process is unaffected. Heavily used in web-servers & databases to improve resilience**
  - If a single thread crashes, it may crash all other threads due to shared memory

- **Security: Parent and child can have different security settings, can run as different users.**
  - Threads are designed to share.

When to use threads

- **When sharing and/or collaboration needed**
- **For performance critical applications**
  - To gain speed through in multi-processor, multi-core systems. In multiprocessor system threads are more effective.

- **If security is not critical**
  - Buffer overflow will enable accessing to the stack of other threads

Today

- Threads vs Processes
- Thread Features and types
- Hello world with threads
- Comparison and applications
- **Concurrent Programming – Intro**
- Sharing among threads

Concurrent Programming is Hard!

- The human mind tends to be sequential
- The notion of time is often misleading
- Thinking about all possible sequences of events in a computer system is at least error prone and frequently impossible

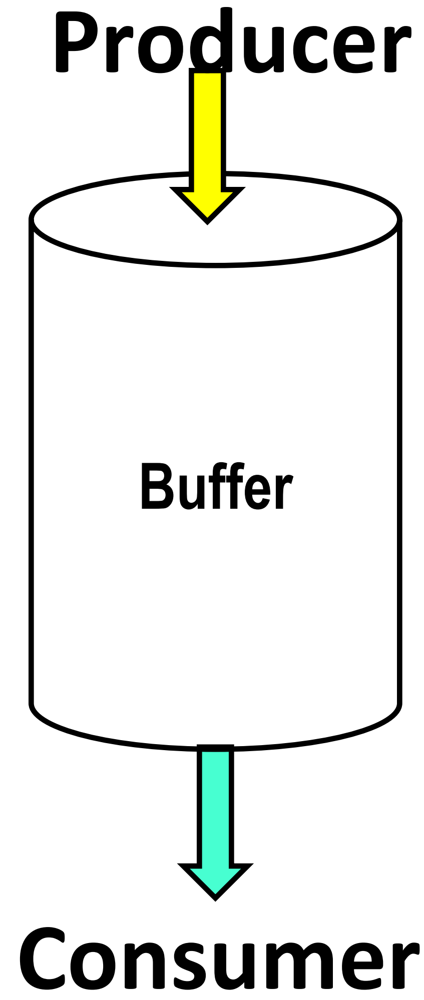
Concurrent Programming is Hard!

- **Classical problems of concurrent programs:**
  - ***Races:*** outcome depends on arbitrary scheduling decisions elsewhere in the system. There is no natural way of knowing if another thread is already working with same data.

- **Requires mutual exclusion**
- **Hardware Level Instructions**
  - test-and-set, compare_and_swap
- **OS support**
  - Mutex Locks, Semaphores
- **Higher level software support**
  - Monitors

Race condition example on a Bounded Buffer

```
while (true) {
        /* produce an item in next produced */
        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```




```
while (true) {
        while (counter == 0); /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in next consumed */
}
```

Race Condition

- `counter++` could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- `counter--` could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

Assume initial value of counter = 5

- Adding one to the buffer
- Removing one from buffer

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter`
S1: producer execute `register1 = register1 + 1`
S2: consumer execute `register2 = counter`
S3: consumer execute `register2 = register2 – 1`
S4: producer execute `counter = register1`
S5: consumer execute `counter = register2`

register1 = 5
register1 = 6
register2 = 5
register2 = 4
counter = 6
counter = 4

Any other possible results?

Race Condition

- **We would arrive this incorrect state because we allowed both processes to manipulate the variable counter concurrently.**
- **A situation like this is called <u>race condition</u>**
- **We need to ensure that only one at a time can manipulate variable counter**
  - To make such a guarantee, we need to synchronize processes in some way

- **Synchronization Brings other problems on the table**

Concurrent Programming is Hard!

- **Synchronization brings other problems to the table:**
  - ***Deadlock:*** improper resource allocation prevents forward progress
    - Example: traffic gridlock
  - ***Starvation / Fairness***: external events and/or system scheduling decisions can prevent sub-task progress
    - Example: people always jump in front of you in line