CSCI-3753: Operating Systems Fall 2019

Abigail Fernandes

Department of Computer Science

University of Colorado Boulder

Week 6
Threads
PThreads
Shared Memory (IPC)

Visual Metaphor

A thread is like a worker in a toy shop!

Worker in a toy shop

- Is an active entity
  - Executing a unit of toy order
- Works simultaneously with others
  - Many workers completing toy orders
- Requires co-ordination
  - Sharing of tools, parts and workstations

Visual Metaphor

A thread is like a worker in a toy shop!

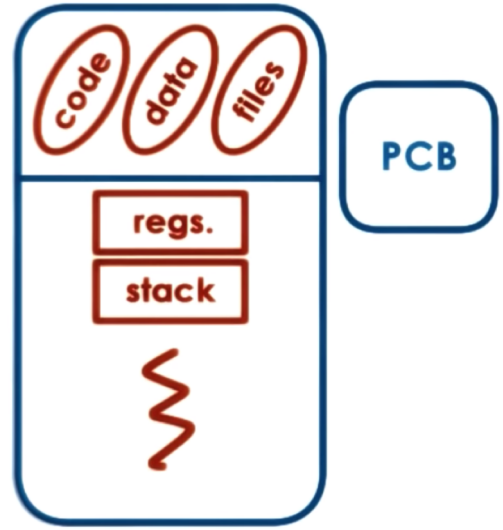
Thread

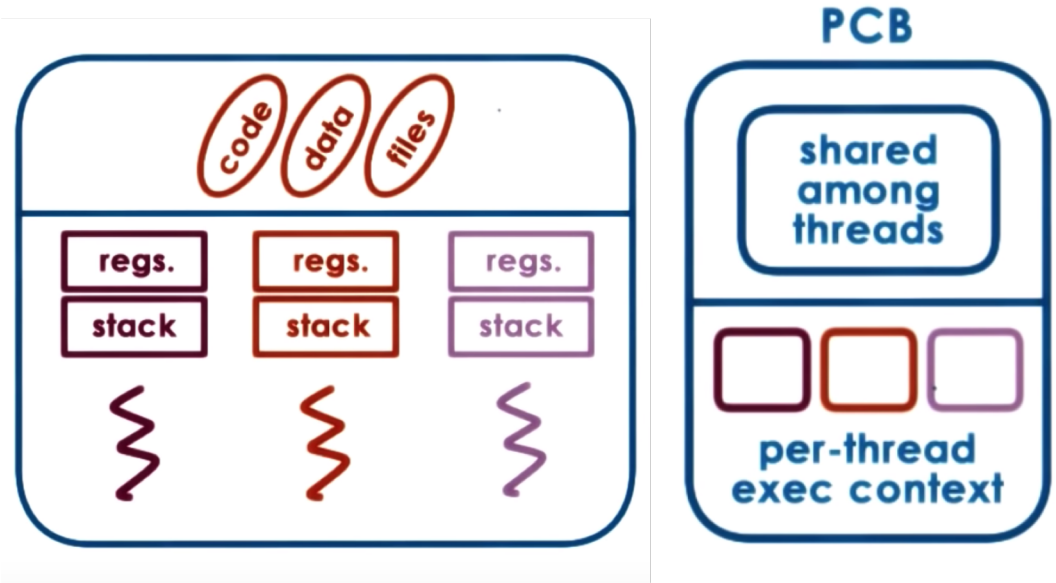
- Is an active entity
  - Executing a unit of process
- Works simultaneously with others
  - Many threads executing
- Requires co-ordination
  - Sharing I/0 devices, CPUs, memory

Worker

- Is an active entity
  - Executing a unit of toy order
- Works simultaneously with others
  - Many workers completing toy orders
- Requires co-ordination
  - Sharing of tools, parts and workstations

Process vs Threads





Threads

- Threads share the same address space
  - Code
  - Data
  - Other OS resources
    - Heap segments
    - Open file descriptors
    - Signal and signal handlers
    - Current working directory
    - User and group ids
- Each thread has its own unique
  - Thread ID
  - Stack
  - Set of registers
  - Program counter

Common Pitfalls with Multithreading

- Race Conditions
- Thread Safe Code
- Mutex deadlock

Common Pitfalls with Multithreading

- **Race Conditions**

  While the code may appear on the screen in the order you wish the code to execute, threads are scheduled by the operating system and are executed at random

- Thread Safe Code
- Mutex deadlock

Common Pitfalls with Multithreading

- Race Conditions
- **Thread Safe Code**
  - There should be no static or global variables which other threads may clobber or read assuming single threaded operation.
  - Many non-reentrant functions return a pointer to static data. → Thread-unsafe functions
- Mutex deadlock

Common Pitfalls with Multithreading

- Race Conditions
- Thread Safe Code
- **Mutex deadlock**
  - Occur when a mutex is applied but then not "unlocked".
  - Cause program execution to halt indefinitely.

Week 6

Threads

PThreads

Shared Memory (IPC)

PThreads

- PThreads == **POSIX** Threads
- POSIX = Portable Operating Systems Interface
- POSIX Threads
  - POSIX version of Birell's API
  - **Specifies** syntax and semantics of the operations
  - API for thread creation and synchronization
- May be provided as user level or kernel level

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

The PThreads API

- **Thread management:** The first class of functions work directly on threads - creating, terminating, joining, etc.
- **Mutexes:** provide for creating, destroying, locking and unlocking mutexes.
- Semaphores: provide for create, destroy, wait, and post on semaphores.
- Condition variables: include functions to create, destroy, wait and signal based upon specified variable values.

Thread Creation

Birell's Mechanism

- Thread
- Fork (proc, args)
  - Thread creation
- Join (thread)

PThreads

```
pthread_t aThread; // type of thread
```

```
int pthread_create (thread, attr, start_routine, arg)
```

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

Thread Creation

int **pthread_create** (tid, attr, start_routine, arg)

- It returns the new thread ID via the ***tid*** argument.
- The ***attr*** parameter is used to set thread attributes, NULL for the default values.
- The ***start_routine*** is the C routine that the thread will execute once it is created.
- A single argument may be passed to ***start_routine*** via ***arg***. It must be passed by reference as a pointer cast of type void.
- Returns status information that indicates if creation was successful or not.

Thread Termination and Join

int pthread_exit (value)

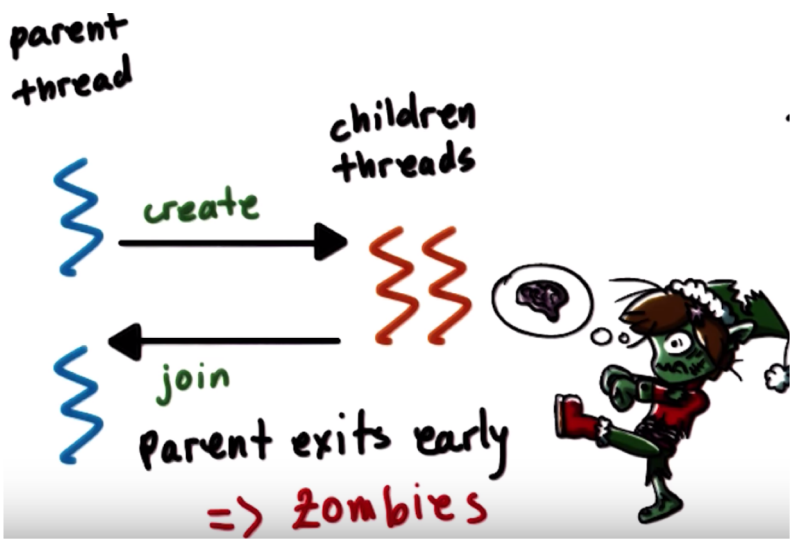
- This function is used by a thread to terminate
- The return value is passed as a pointer.

int pthread_join(pthread_t thread_id, void ** status)

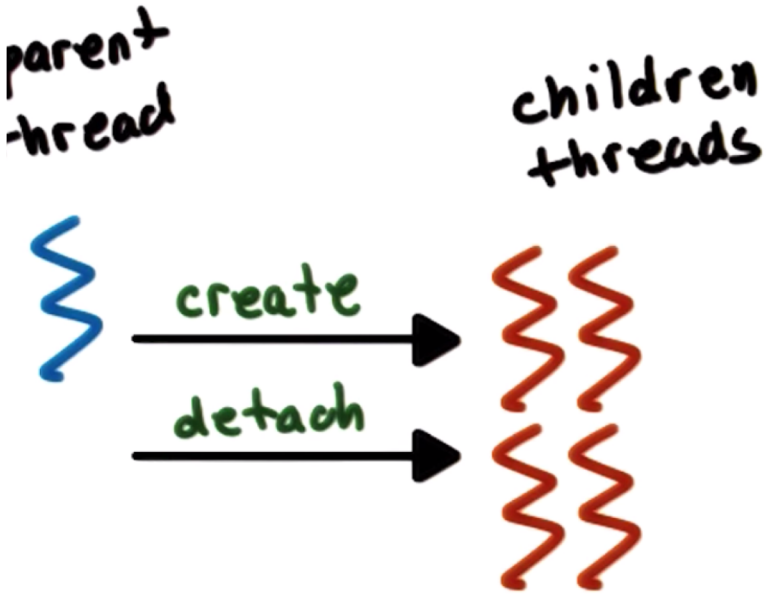
- This subroutine blocks the calling thread until the specified ***thread_id*** thread terminates
- Returns 0 on success, -1 on failure.
- The returned value is a pointer returned by reference.
- If you do not care about the return value, you can pass NULL for the second argument

Detatching PThreads

Default: Joinable Threads



Detached Threads



Compiling PThreads

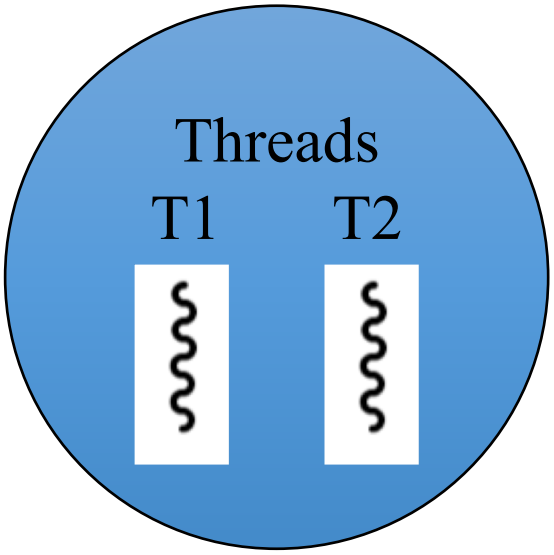
1. `#include <pthread.h>` in main file.

2. Compile source with –lpthread or -pthread

```
Intro to OS ~ ==> gcc -o main main.c -lpthread
Intro to OS ~ ==> gcc -o main main.c -pthread
```

3. Check return values of common functions

Example 1



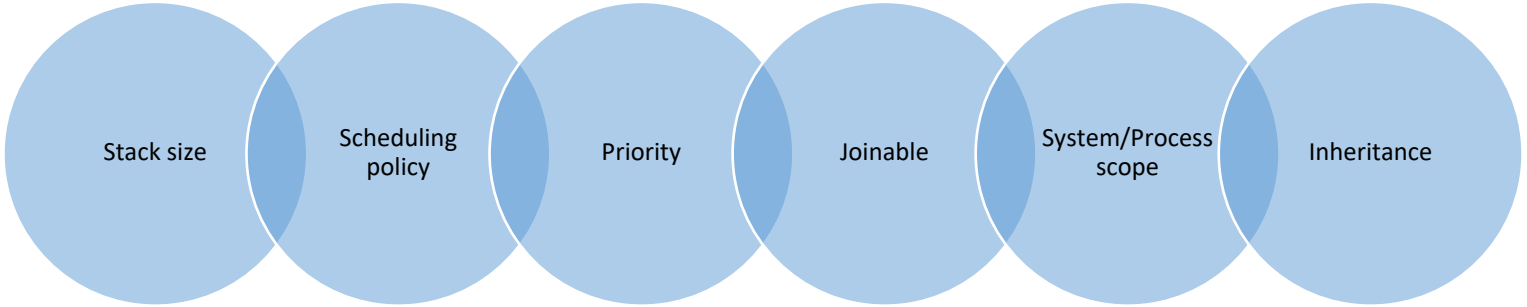
- One main thread creates two new threads with **default** parameters.
- **Main thread:**
  - Write whole numbers in a file called ***whole_num***
- **New thread 1:**
  - Execute odd() function that writes odd numbers in a file called ***odd_num***
- **New thread 2:**
  - Execute even() function that writes even numbers in a file called ***even_num***

PThread Attributes

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

pthread_attr_t

- Specified in pthread_create
- Defines features of the new thread
- Has default behavior with NULL in pthread_create



PThread Attributes

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);

int pthread_attr_setstacksize(pthread_attr_t *attr,
size_t sz);
int pthread_attr_getstacksize(const pthread_attr_t
*attr, size_t *sz);

int pthread_attr_setstack(pthread_attr_t *attr, void
*addr, size_t stacksize);
int pthread_attr_getstack(const pthread_attr_t *attr,
void **addr, size_t *stacksize);
```

Thread Creation (Attributes)

- Thread's SET SCOPE
  - Scope values
    - PTHREAD_SCOPE_PROCESS (unbound, default)
      - The thread competes for resources with all other threads in the same process.
      - They are scheduled relative to one another according to their scheduling policy and priority.
    - PTHREAD_SCOPE_SYSTEM (bound)

```
int pthread_attr_setscope(pthread_attr_t *attr, int
scope);
int pthread_attr_getscope(const pthread_attr_t
*attr, int *scope);
```

Thread Creation (Attributes)

- Thread scheduling attributes
  - Inherit values
    - PTHREAD_INHERIT_SCHED (default)
      - Threads that are created using `attr` inherit scheduling attributes from the creating thread.
    - PTHREAD_EXPLICIT_SCHED
      - Threads that are created using `attr` take their scheduling attributes from the values specified by the attributes object.

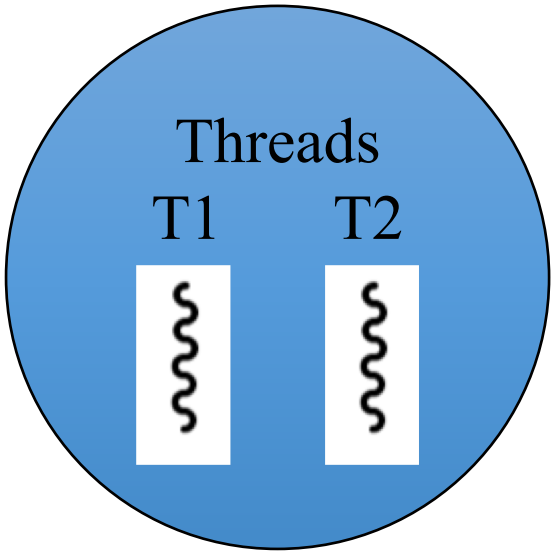
```
int pthread_attr_setinheritsched(pthread_attr_t *attr,
int inherit);
int pthread_attr_getinheritsched(const pthread_attr_t
*attr, int *inherit);
```

Thread Creation (Attributes)

- Thread scheduling attributes
  - Policy values
    - SCHED_FIFO (First in-first out scheduling)
    - SCHED_RR (Round-robin scheduling)
    - SCHED_OTHER (Default Linux time-sharing scheduling)

```
int pthread_attr_setschedpolicy(pthread_attr_t
*attr, int policy);
int pthread_attr_getschedpolicy(const pthread_attr_t
*attr, int *policy);
```

Example 2



One main thread creates two new threads with **specific** parameters.

All 3 threads write to standard output (i.e., only monitor).

All 3 threads use

- Case 1: Local variables -> predict the execution
- Case 2: The same global variable -> predict the execution

PThread Mutexes

Solve mutual exclusion problems among concurrent threads.

Birell's Mechanism

- Mutex
- Lock (mutex) {

}

PThreads

```
pthread_mutex_t aMutex; // mutex type
```

```
int pthread_mutex_lock (pthread_mutex_t *mutex)
```

```
int pthread_mutex_unlock (pthread_mutex_t *mutex)
```

Pthread Mutexes

Birell

```
list<int> my_list;
Mutex m;
void safe_insert(int i) {
  Lock(m) {
    my_list.insert(i);
  } // unlock;
}
```

PThreads

```
list<int> my_list;
pthread_mutex_t m;
void safe_insert(int i) {
  pthread_mutex_lock(m);
  my_list.insert(i);
  pthread_mutex_unlock(m);
}
```

Other Mutex Operations

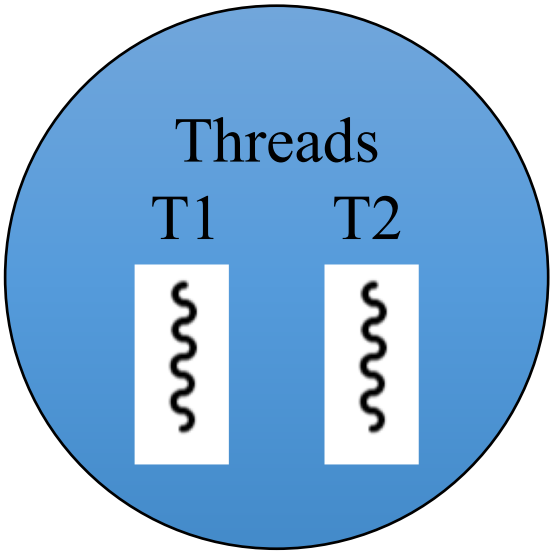
```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
// mutex attributes == specifies mutex behavior when
// a mutex is shared among processes
```

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

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

A null value of ***attr*** initializes mutex with default attributes.

Example 3 (Mutex)



- One main thread creates two new threads with **specific** parameters.

- All 3 threads write to standard output (i.e., only monitor).

- The threads use **mutexes** to synchronize their writes in the following 2 cases:
  - **Case 1**: Local variables -> predict the execution
  - **Case 2**: The same global variable -> predict the execution

Week 6

Threads

PThreads

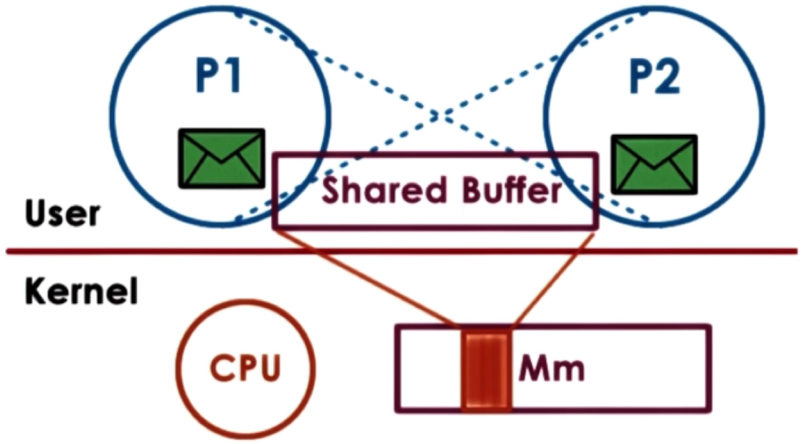
Shared Memory (IPC)

Visual Metaphor

IPC is like working together in a toy shop

- Workers share work area
  - Parts and tools on the table
- Workers call each other
  - Explicit requests and responses
- Requires synchronization
  - I'll start when you finish

Shared Memory IPC

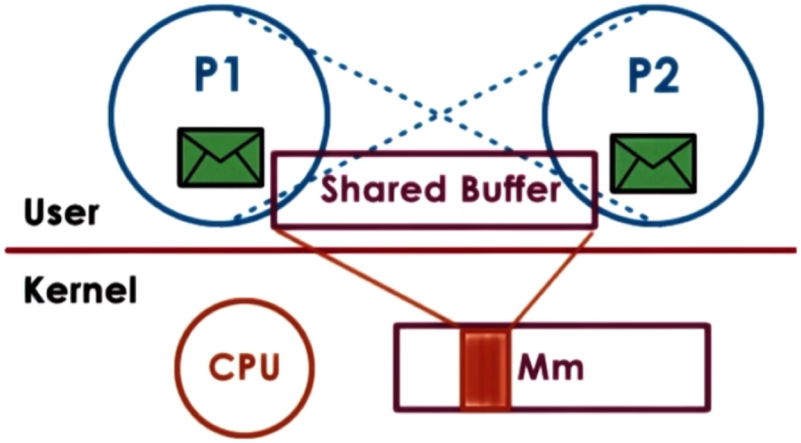


Read and write to shared memory region

OS establishes shared channel between the processes.

1. Physical pages mapped into virtual address space
2. VA(p1) and VA(p2) map to the same physical address.
3. VA(p1) != VA(p2)
4. Physical memory does not need to be contiguous

Shared Memory IPC



- Explicit synchronization
- Programmers responsibility: Communication protocol, shared buffer management
- Complicated to implement in a distributed system

- System calls only for set up, so very fast.
- Data copies are potentially reduced (but not eliminated)
- Useful for large amounts of data

Shared Memory

- Create a shared memory segment
- `int shmget(key_t key, size_t size, int shmflg);`
- Other processes (if permitted) can access.
- Library: *sys/ipc.h* and *sys/shm.h*
  - *key*: a value used to share between processes
  - *size*: size of the shared memory segment
  - *shmflg*: options for segment creation

Shared Memory

- Attach the shared segment to a process address space

```
void *shmat(int shmid, const void *shmaddr,…int shmflg);
```

- Once attached, a process can read or write to the segment.
- Library: *sys/types.h* and *sys/shm.h*
  - *shmid*: id of the shared memory segment
  - *shmaddr*: address in the process space
  - *shmflg*: options for shared memory permission

Shared Memory

- Detach the shared memory segment located at the shmaddr address

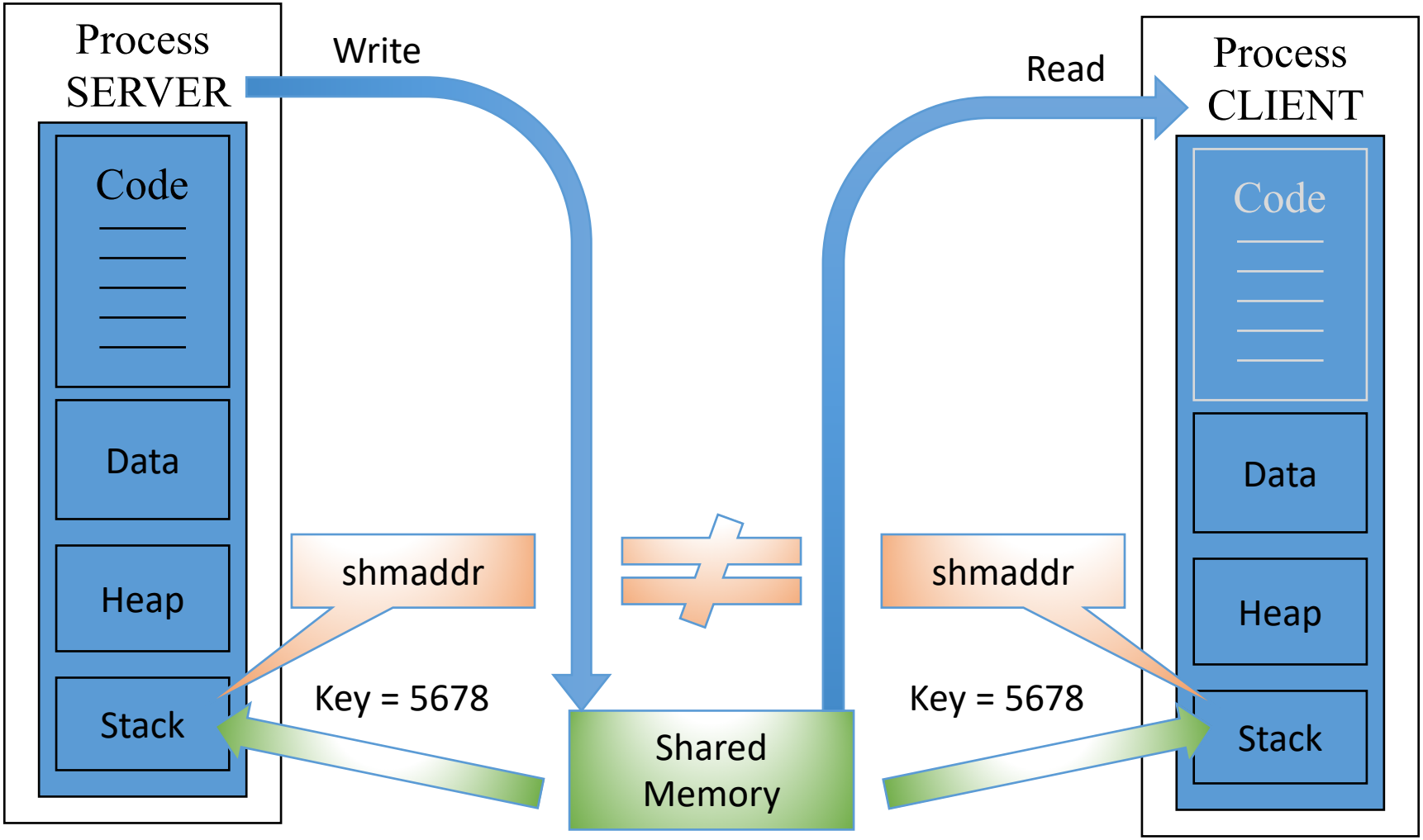
```
int shmdt(const void *shmaddr);
```

- Library: *sys/types.h* and *sys/shm.h*
  - *shmaddr*: attaching address in the process space

Shared Memory Example

- Goal: To illustrate the usage of shared memory between two processes in Linux.
- The server process creates a shared memory segment and writes some characters in it. It then waits for the first character in the shared memory to become '*', and then exits.
- The client process reads characters from this shared memory and prints them out on the terminal, writes '*' at the beginning of the shared memory segment, and then exits.

Shared Memory Example (Code Demo)



Week 6 – Checklist

- ❑ Discuss PThreads
- ❑ Reviewed examples on PThreads
- ❑ Discuss Shared Memory from IPC
- ❑ Read more about threads
- ❑ **BEGIN PA3!!!!!**