CS310 Operating Systems

Lecture 14: Threads – Examples

Ravi Mittal

IIT Goa

Acknowledgements !

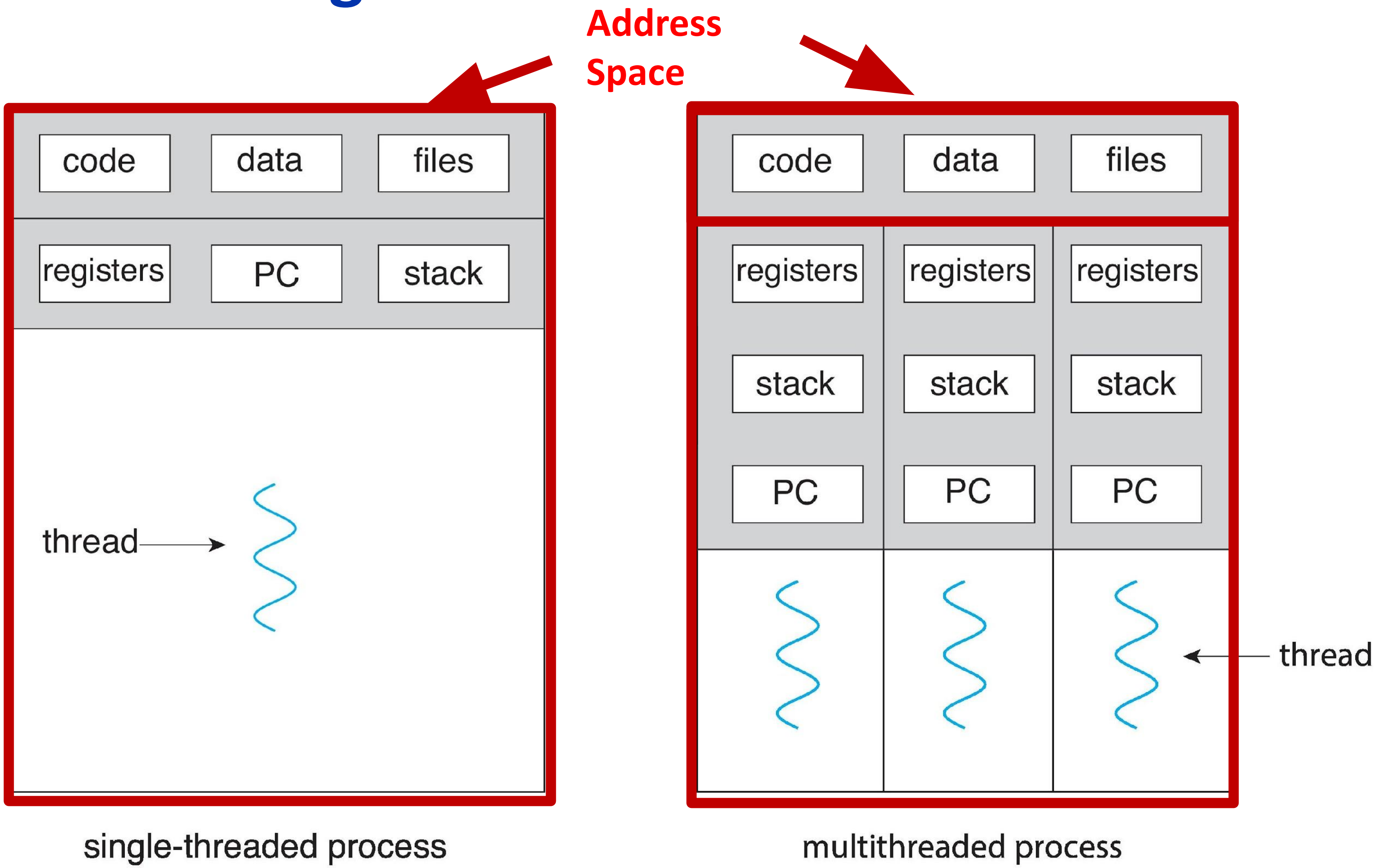
- Contents of this class presentation has been taken from various sources. Thanks are due to the original content creators:
  - CS162, Operating System and Systems Programming, Profs. Natacha Crooks and Anthony D. Joseph, University of California, Berkeley
  - CS240 Computer Systems, Univ of Illinois, Prof. Wade Fagen-Ulmschneider
  - Operating Systems: Three Easy Pieces, by Remzi and Andrea Arpaci-Dusseau,
available for free online
  - Book: Modern Operating Systems, Andrew Tenenbaum, and Herbert Bos, 4th Edition, Pearson

Reading

- Book: Operating Systems: Principles and Practice: Thomas Anderson and Michael Dahlin, Part 2, Chapter 4
- CS162, Operating Systems and Systems Programming, University of California, Berkeley
- CS4410, Operating Systems, Course, Cornell University, Spring 2019, Lecture on Threads
- Operating Systems: Three Easy Pieces, by Remzi and Andrea Arpaci-Dusseau,
  available for free online

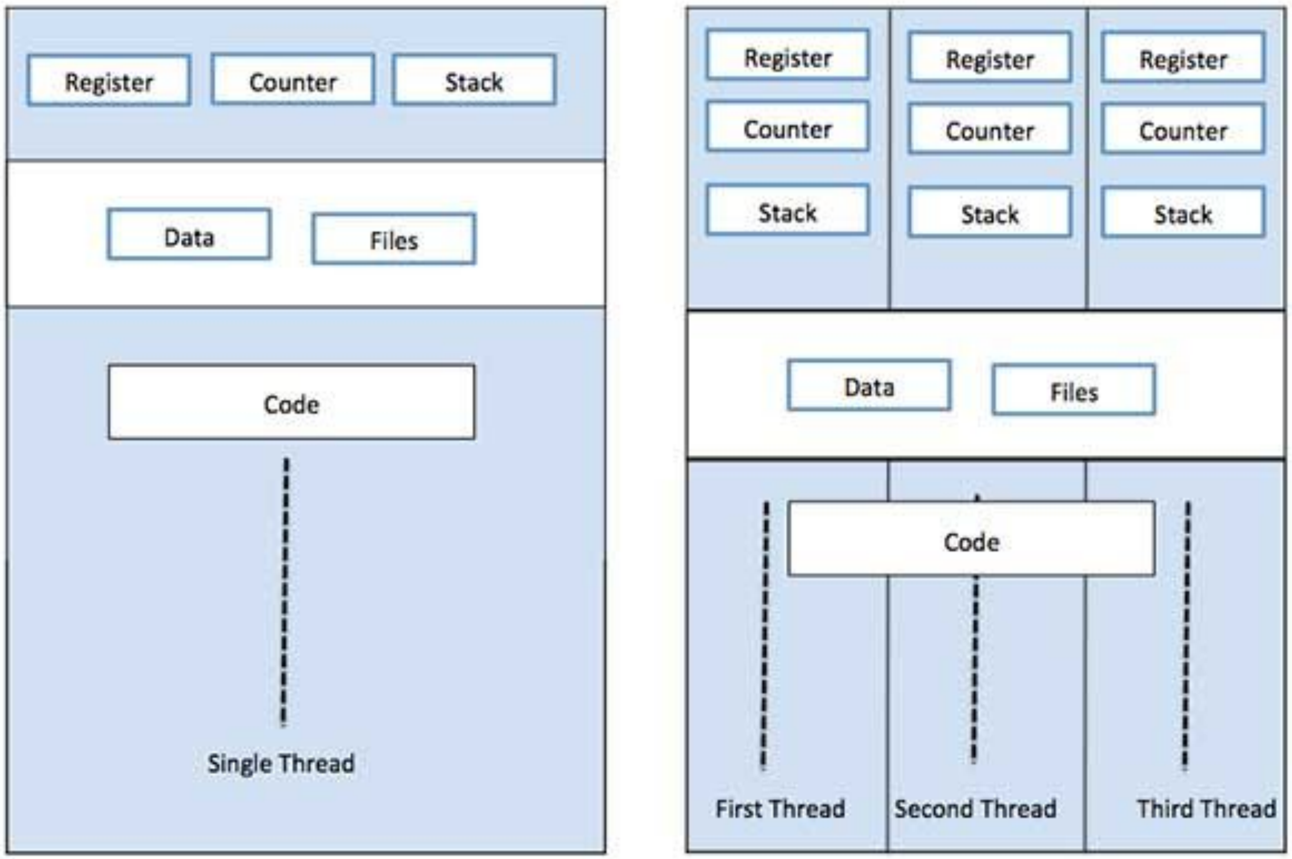
We have learnt so far ..

Recall: Single and Multithreaded Processes



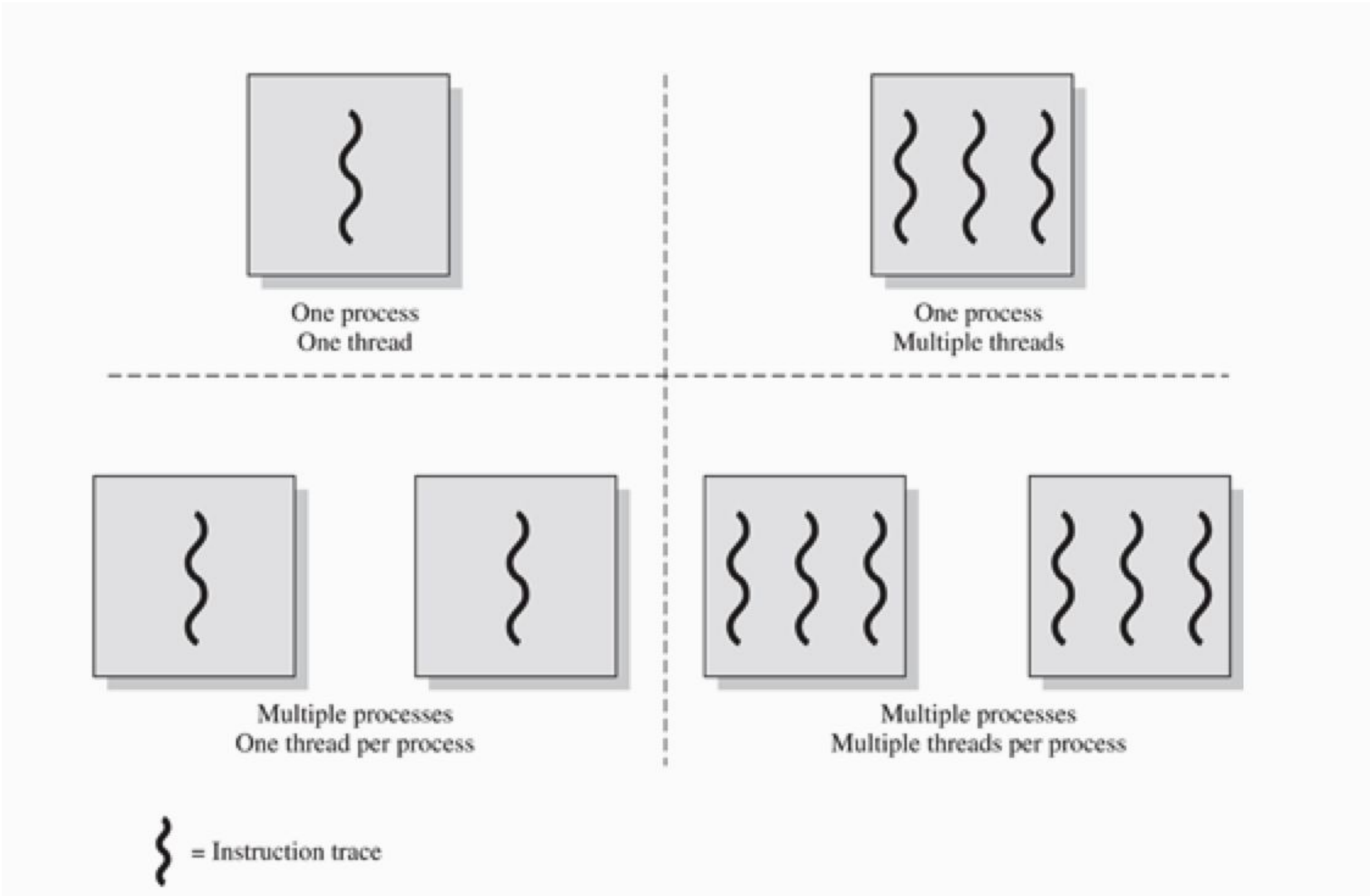
- Address spaces encapsulate protection: Passive Part
- Threads share code, data, files, heap

Thread Abstraction



Single Process P with single thread

Single Process P with three threads

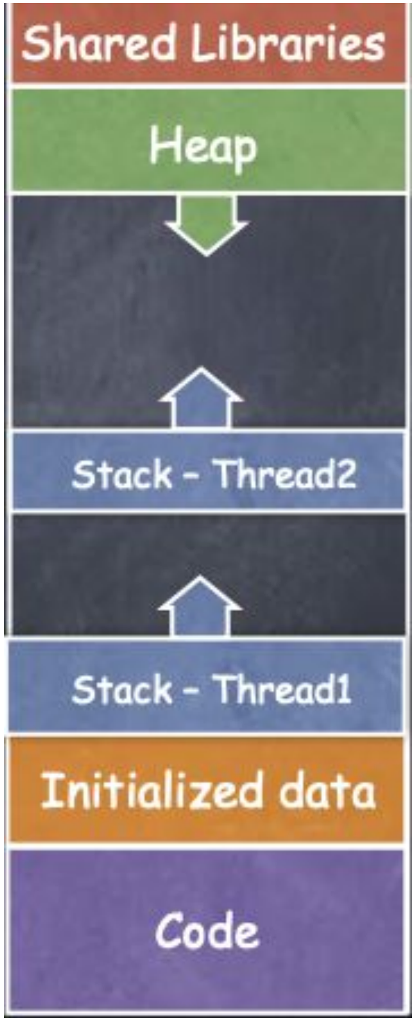


Source: OS book by Stallings

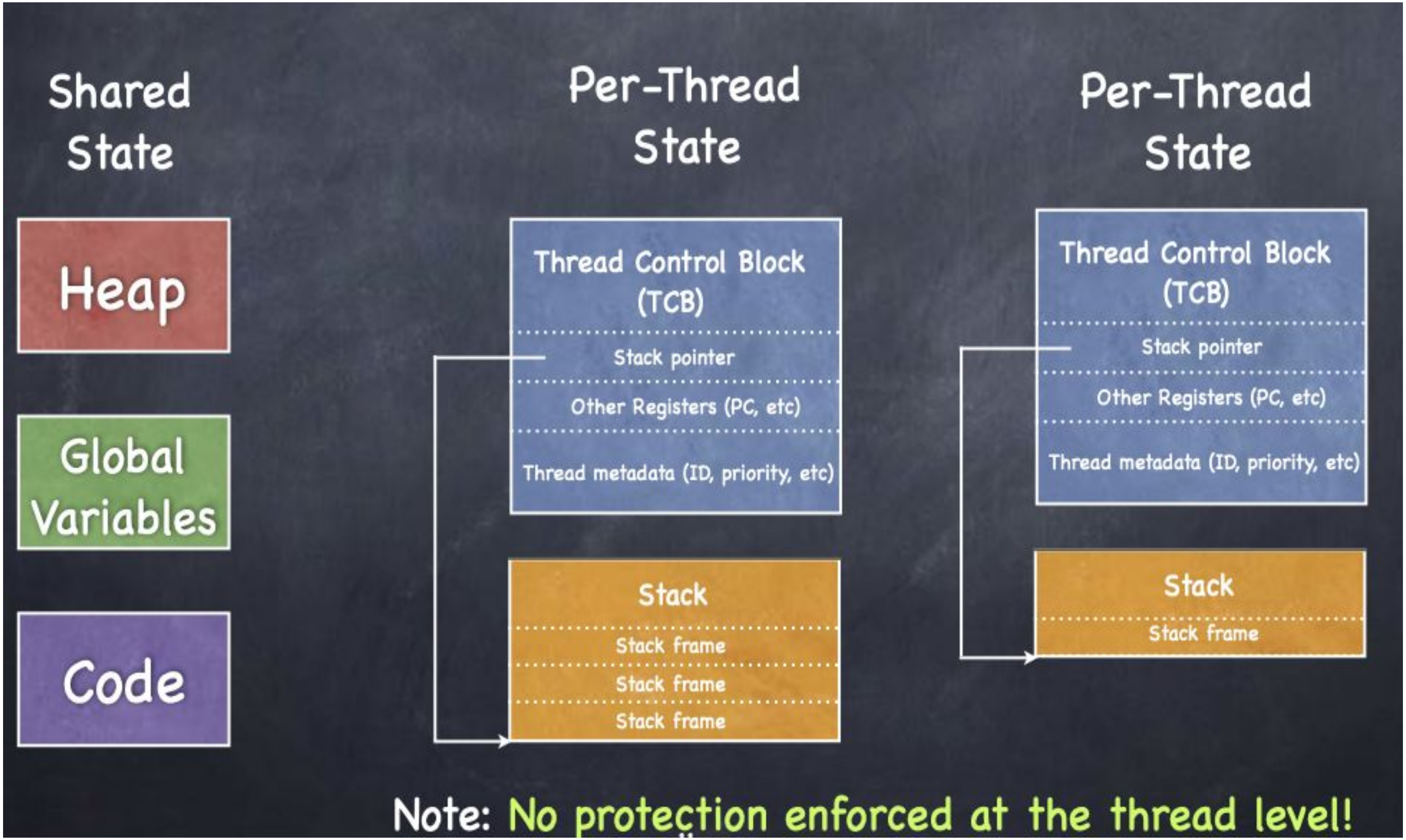
Threads

- All threads within a process share
  - Heap
  - Global/static data
  - Libraries
- Each thread has a separate
  - Program Counter
  - Stack
  - registers

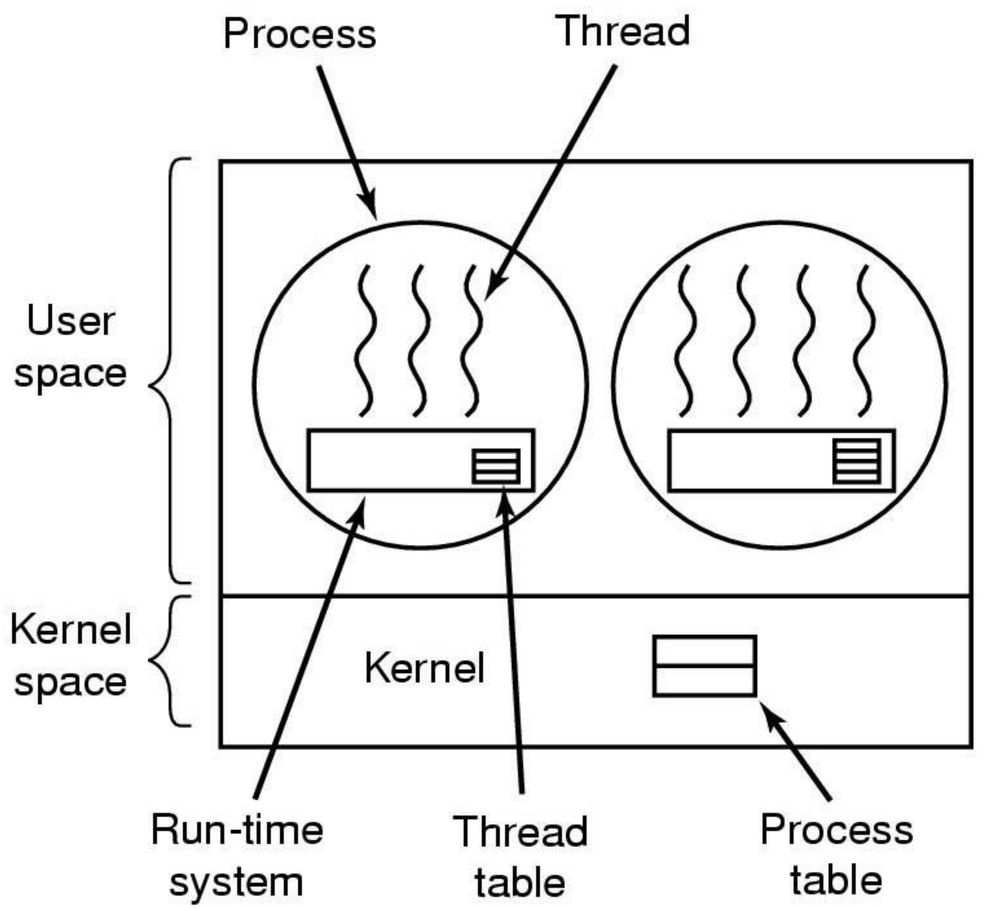
Process Address Space



Implementing Thread Abstraction



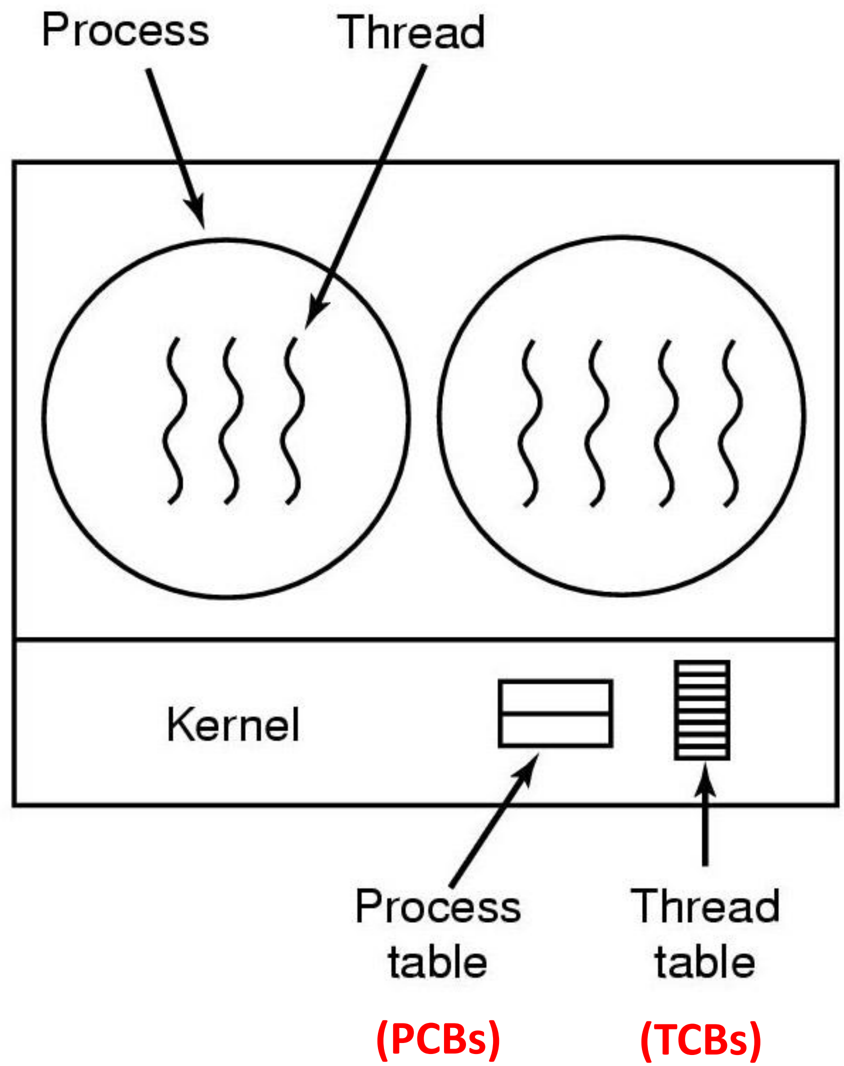
Implementing Thread in User Space (1)



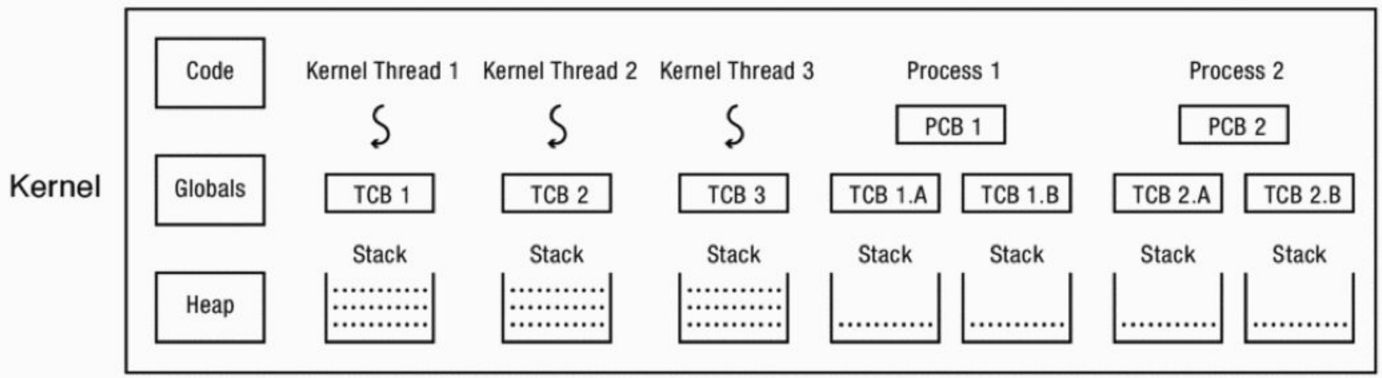
- Threads run on top of run-time system
    - Collection of procedures to manage threads
- Each process needs own private thread table (Thread control Block)

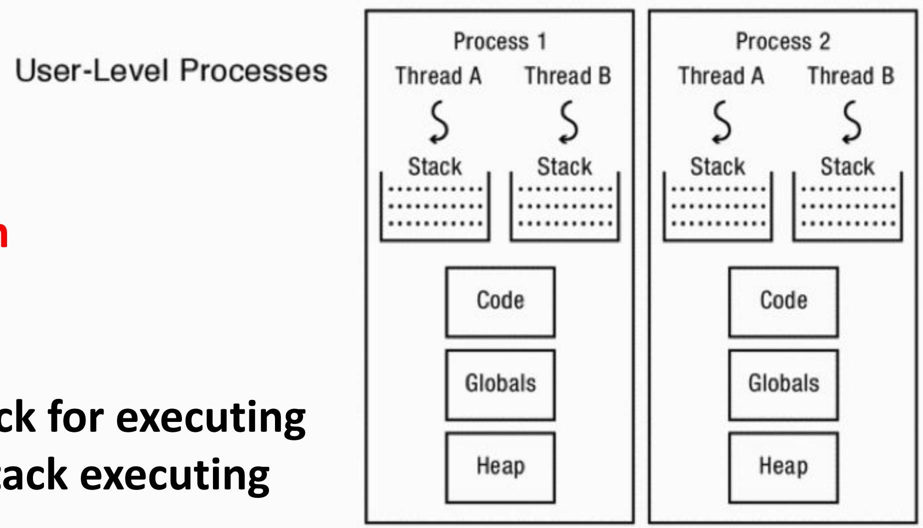
Thread Management in Kernel (1)

- Kernel knows how to manage threads in a user process
  - No run-time system required
  - No thread table in each process
- When a thread wants to create a new thread or destroy an existing thread, makes a system call
  - Kernel creates or destroys thread as per request.
  - Updates kernel thread table



Thread Management in Kernel





Three Kernel threads and
Two user level processes – each
process with two threads

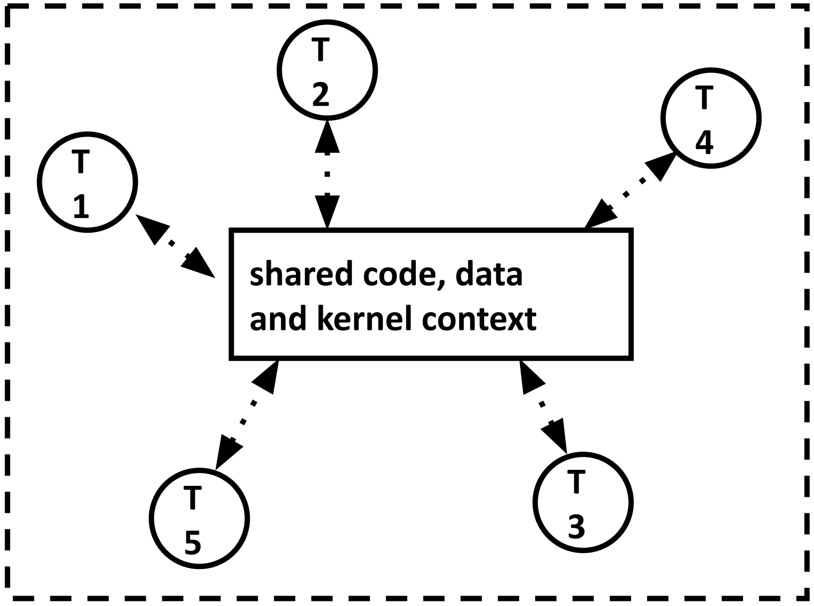
Each user thread has a user stack for executing code and a Kernel (interrupt) stack executing interrupts and system calls

Thread management is done in Kernel

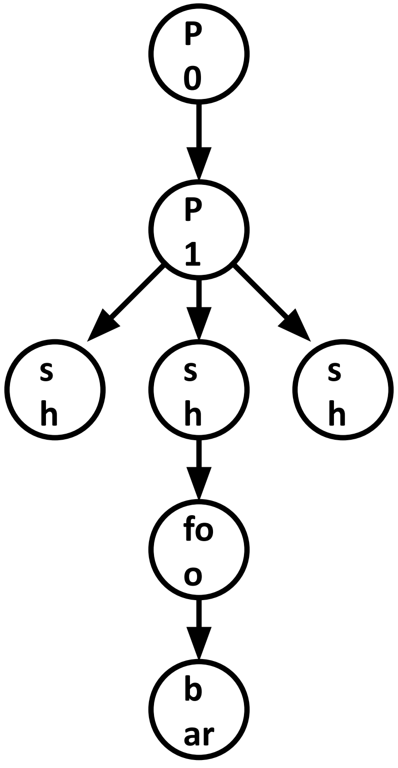
Logical View of Threads

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

Threads associated with process foo



Process hierarchy



Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
    - All code and data structures for the library exist in user space with no kernel support
  - Kernel-level library supported by the OS
    - Code and data structures exist in Kernel Space
- Invoking a function in the API for the library typically results in a system call to the kernel

Today we will study ..

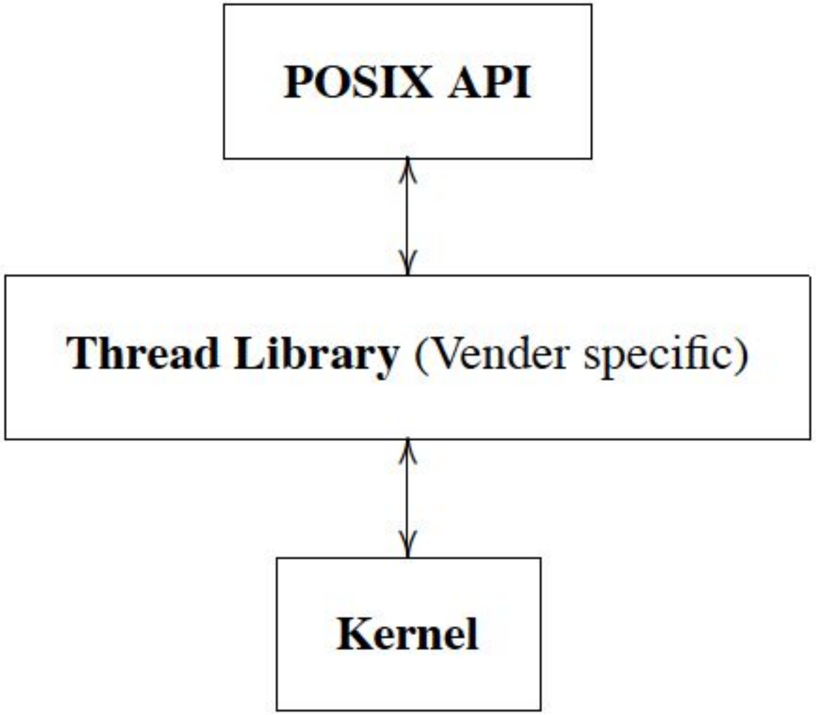
- Posix Thread library
- Thread Example 1

Thread Libraries

- Three main Thread Libraries in use today
  - POSIX Pthread
    - User or Kernel level
  - Windows
    - Kernel-level
  - Java
    - Using Windows APIs
- Linux, Unix, MacOS
  - Pthreads

Thread Library: pthread

Thread Implementation: Layers of Abstraction



What are pthreads?

- POSIX standard IEEE 1003.1c defines a thread interface
  - `pthreads`
- Unix/Linux provides `pthread` library
  - APIs to create and manage threads
- Implementation is up to development of the library
- Simply a collection of C functions
- Standard interface for ~60 functions that manipulate threads from C programs
- Primary way of doing threading in Linux is `pthread`
- Since 2003 (Kernel 2.6), Linux implements POSIX threads as kernel-scheduled threads

pthread APIs

- Creating and reaping threads
  - `pthread_create, pthread_join`
- Determining your thread ID
  - `pthread_self`
- Terminating threads
  - `pthread_cancel, pthread_exit`
- Synchronizing access to shared variables
  - `pthread_mutex_init, pthread_mutex_[un]lock`
  - `pthread_cond_init, pthread_cond_[timed]wait`

Aside: Pointers to Functions in C (self reading)

- A function can take many types of arguments including the address of another function
- Instructions of a program are stored in MM locations
- Instruction too have addresses
  - Like data
- So, we can point to an individual instruction
  - Pointing to an individual instruction is not useful
- Point to the first instruction of a function useful
- Function Pointer :
  - a variable or parameter that points at a function
- Note that a function name is a pointer to it
  - Helps to initialize such variables or parameters

Aside: Pointers to Functions in C (self reading)

- Function name is actually starting address of the code in the memory (think Computer Architecture – Machine instructions)
- Start of the function code or the address of a function is a function pointer
- Function pointer is different from other pointers since you do not allocate or deallocate memory with them
- Function pointers can be passed as arguments to other functions or return from functions
- Unfortunately function pointers have complicated syntax and therefore are not widely used

Aside: Pointers to Functions in C (self reading)

- Provides late binding
  - deciding the proper function during runtime instead of compile time
- Ex: Determine sorting function based on type of data at run time
- Technically speaking, the name of any function is a pointer to that function
- However, when we refer to a function pointer, we mean a variable or parameter that points to a function

Aside: Pointers to Functions in C (self reading)

```
int (*fn)(int,int) ;
```

- Here we define a function pointer `fn`, that can be initialized to any function that takes two integer arguments and return an integer
- Consider a function:

```
int sum(int x, int y) {
    return (x+y);
}
```

Now to initialize `fn` to the address of the `sum`, we can do the following:

`fn = &sum`   /* make `fn` points to the address of `sum` */

`int x = (*fn)(12,10);`   /* call to the function through a pointer */

Pthread_create

```
int pthread_create(
      pthread_t   *thread,        /* thread struct */
   const pthread_attr_t *attr, /* usually NULL */
   void *(*start_routine) (void *),  /* start func */
   void *arg                    /* thread start arg */
   );
```

- This function takes 4 parameters
- The first parameter is a pointer to structure of type pthread_t which describes a thread
  - Pthread_create function fills in this structure with information about the newly created thread
  - We can use type pthread_t to refer to the thread in other library calls
  - The thread parameter points to the thread ID of the newly created thread
- The second argument lets you specify particular attribute of the thread
  - Usually Null. The new thread has default attributes.

Pthread_create

```
int pthread_create(
      pthread_t  *thread,        /* thread struct */
   const pthread_attr_t *attr, /* usually NULL */
   void *(*start_routine) (void *), /* start func */
   void *arg              /* thread start arg */
   );
```

- The third argument specifies the entry point to the new thread
  - Unlike the entire program, which starts at main, there is no pre-defined entry point for other threads
  - The thread that creates the new thread must specify the function in which the new thread must start
  - This function is specified by passing a function pointer which must have type `void*(*)(void *)`
    - A pointer to a function that takes a `void*` and returns a `void*`

Pthread_create

```
int pthread_create(
      pthread_t  *thread,      /* thread struct */
   const pthread_attr_t *attr, /* usually NULL */
   void *(*start_routine) (void *), /* start func */
   void *arg                  /* thread start arg */
   );
```

- arg parameter specifies what to pass into this function as its arguments when the function is called on the newly spawned thread
- It returns 0 on success; non-zero error code on error
- The thread created by pthread_create is runnable without requiring a separate start code. It begins execution on the first line of the function routine provided in the third argument.

Pthread_create

- When `pthread_create` is called a few things happen (that are different from we have seen before)
- A new stack is created that is independent of caller's stack
  - Frames can be created and destroyed on this stack – which is independent of frame creation and destruction on other stacks (for other threads)
  - The stack with a frame is created for the function requested (as the third argument) passing in the arguments
- Now second execution arrow (corresponding to the current value of PC) is created at the start of the function that we specify as the entry point of the new thread
  - You can imagine execution arrow as a point in execution of the thread
- Concurrent execution of both threads begins

pthreads – Completion

- A Thread exits
  - By returning from the function it started in, or
  - By calling `pthread_exit`
  - It's return value (either what the function returned or argument passed to `pthread_exit` must be kept available for another thread to retrieve
- `pthread_join`
  - Thread calling pthead_join waits for the specified thread to exit and obtaining the return value
  - When one thread calls `pthread_joins` - it blocks until the thread it is joining terminates
    - Resources associated with the thread that is ended

Pthread - Completion

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

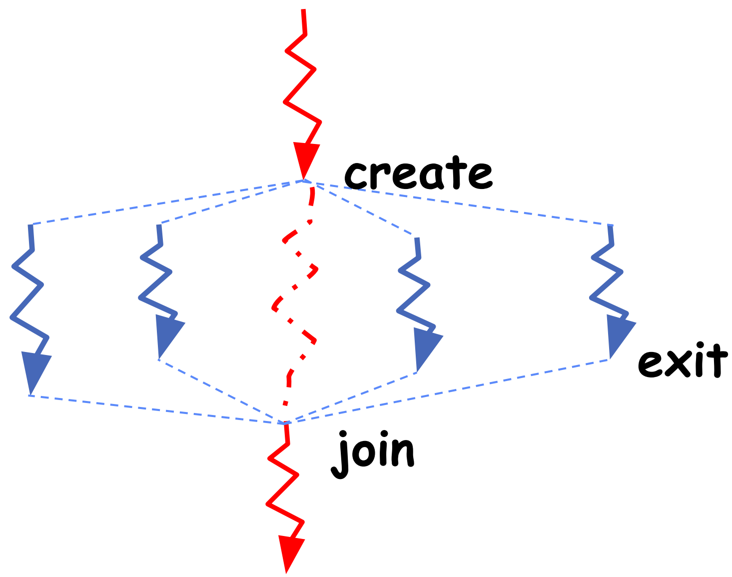
- The first argument is of type `pthread_t` and it specifies the thread to wait for
- The second argument is a pointer to the return value you expect to get back
  - It ignores the return value by passing in NULL for the second argument
- When one thread calls `pthread_join` it stops advancing – until the thread it is joining terminates

- Joining a thread also allows for the pthread library to release resources associated with the thread
- A zombie thread is a joinable thread which has terminated, but which hasn't been joined.

pthread_detach; Main terminating

- `Pthread-detach`
  - A thread tells the `pthread` library that is will never join another thread with `pthread_detach`
  - This allows library to release resources as soon as the thread exits
- The Main thread (in main function) behaves differently
  - If main returns, the entire process exits – thus terminating all threads inside it
  - Note that main was not called by pthread library. It was called by C library

Fork-Join Parallelism



- Main thread *creates* (forks) collection of sub-threads passing them args to work on…
- … and then *joins* with them, collecting results
- Data may be safely shared between parent and child
  - It is written by parent before child starts
  - It is written by the child and read by the parent after the join

Thread - Example

th1.c

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

void * sqr(void *arg){
int * ptr = arg;
int n = *ptr;
for (int i = 0; i < n ; i++){
int fsq = (i * i);
printf("squaare: %d\n", fsq);
}
return NULL;
}
int main(void) {
pthread_t thread;
int x = 3;
pthread_create(&thread, NULL, sqr, &x);
sleep(1);
for (int i = 0; i < 100; i++){
printf("main: %d\n", i);

}
pthread_join(thread, NULL);
printf("join completed \n");
return EXIT_SUCCESS;

}
```

th1.c

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

void * sqr(void *arg){
int * ptr = arg;
int n = *ptr;
for (int i = 0; i < n ; i++){
int fsq = (i * i);
printf("squaare: %d\n", fsq);
}
return NULL;
}
int main(void) {
pthread_t thread;
int x = 3;
pthread_create(&thread, NULL, sqr, &x);
sleep(1);
for (int i = 0; i < 100; i++){
printf("main: %d\n", i);

}
pthread_join(thread, NULL);
printf("join completed \n");
return EXIT_SUCCESS;

}
```

Thread 0

main

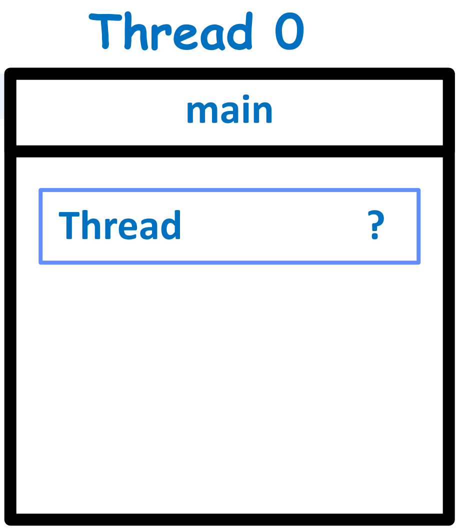
th1.c

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

void * sqr(void *arg){
int * ptr = arg;
int n = *ptr;
for (int i = 0; i < n ; i++){
int fsq = (i * i);
printf("squaare: %d\n", fsq);
}
return NULL;
}
int main(void) {
pthread_t thread;
int x = 3;
pthread_create(&thread, NULL, sqr, &x);
sleep(1);
for (int i = 0; i < 100; i++){
printf("main: %d\n", i);

}
pthread_join(thread, NULL);
printf("join completed \n");
return EXIT_SUCCESS;

}
```



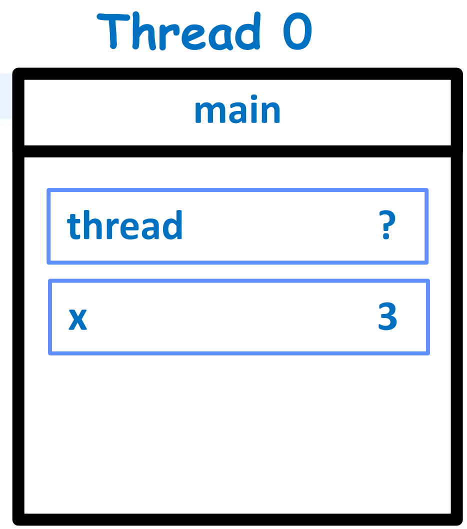
th1.c

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

void * sqr(void *arg){
int * ptr = arg;
int n = *ptr;
for (int i = 0; i < n ; i++){
int fsq = (i * i);
printf("squaare: %d\n", fsq);
}
return NULL;
}
int main(void) {
pthread_t thread;
int x = 3;
pthread_create(&thread, NULL, sqr, &x);
sleep(1);
for (int i = 0; i < 100; i++){
printf("main: %d\n", i);

}
pthread_join(thread, NULL);
printf("join completed \n");
return EXIT_SUCCESS;

}
```



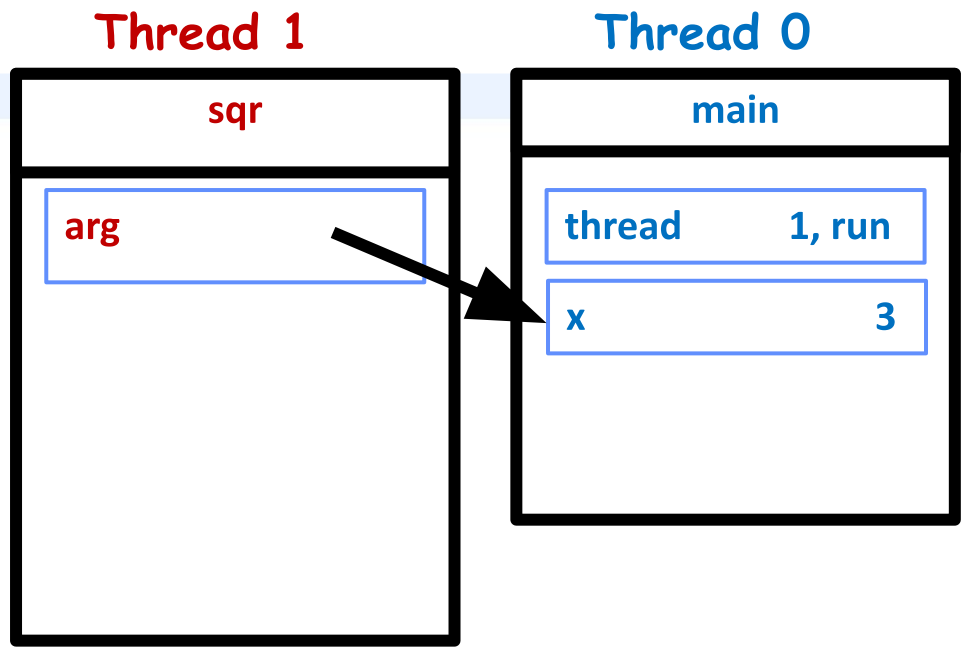
th1.c

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

void * sqr(void *arg){
int * ptr = arg;
int n = *ptr;
for (int i = 0; i < n ; i++){
int fsq = (i * i);
printf("squaare: %d\n", fsq);
}
return NULL;
}
int main(void) {
pthread_t thread;
int x = 3;
pthread_create(&thread, NULL, sqr, &x);
sleep(1);
for (int i = 0; i < 100; i++){
printf("main: %d\n", i);

}
pthread_join(thread, NULL);
printf("join completed \n");
return EXIT_SUCCESS;

}
```



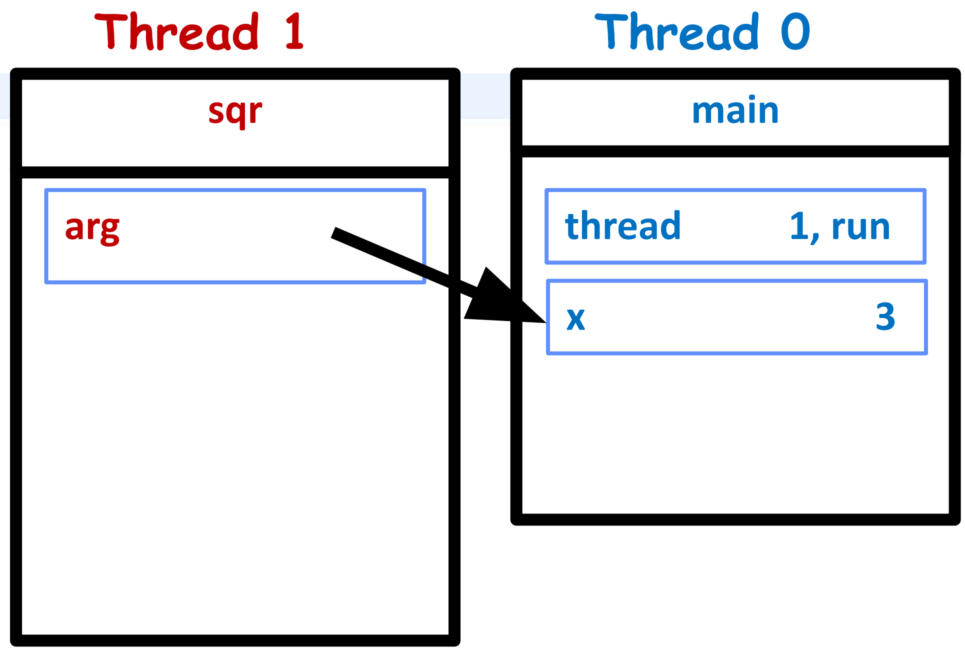
th1.c

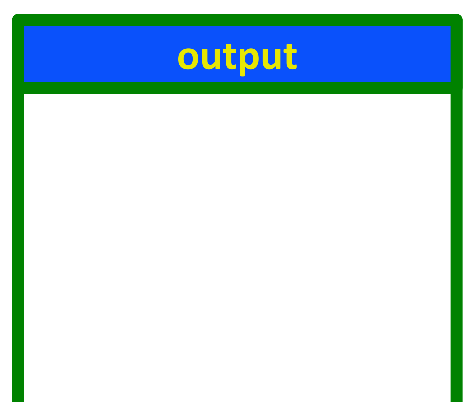
```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

void * sqr(void *arg){
int * ptr = arg;
int n = *ptr;
for (int i = 0; i < n ; i++){
int fsq = (i * i);
printf("squaare: %d\n", fsq);
}
return NULL;
}
int main(void) {
pthread_t thread;
int x = 3;
pthread_create(&thread, NULL, sqr, &x);
sleep(1);
for (int i = 0; i < 100; i++){
printf("main: %d\n", i);

}
pthread_join(thread, NULL);
printf("join completed \n");
return EXIT_SUCCESS;

}
```





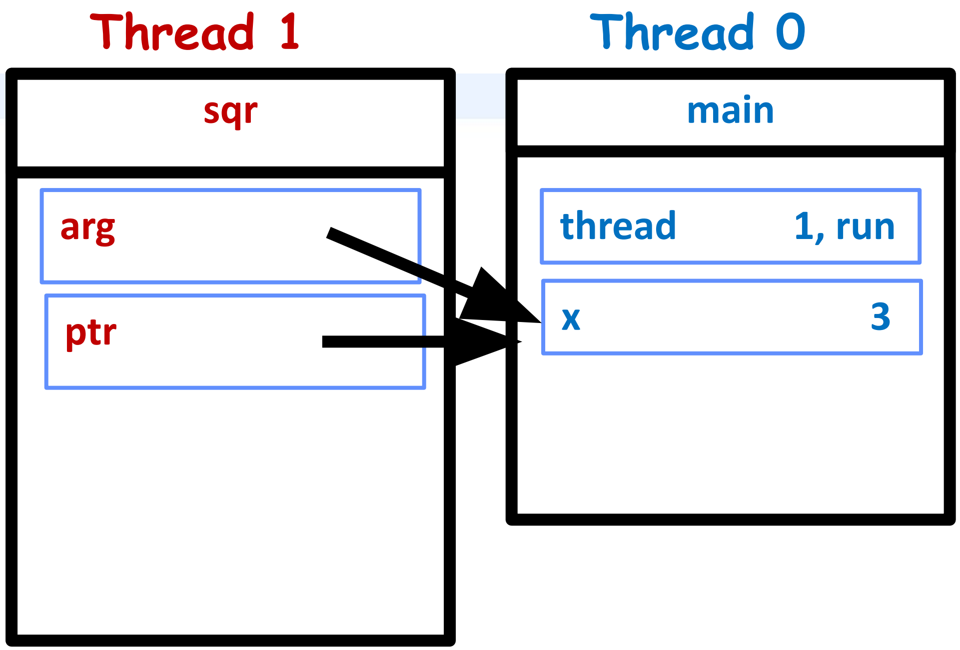
th1.c

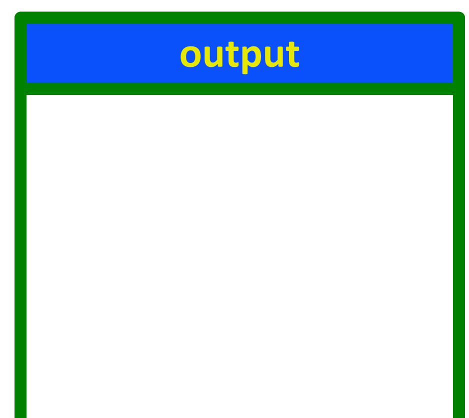
```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

void * sqr(void *arg){
int * ptr = arg;
int n = *ptr;
for (int i = 0; i < n ; i++){
int fsq = (i * i);
printf("squaare: %d\n", fsq);
}
return NULL;
}
int main(void) {
pthread_t thread;
int x = 3;
pthread_create(&thread, NULL, sqr, &x);
sleep(1);
for (int i = 0; i < 100; i++){
printf("main: %d\n", i);

}
pthread_join(thread, NULL);
printf("join completed \n");
return EXIT_SUCCESS;

}
```





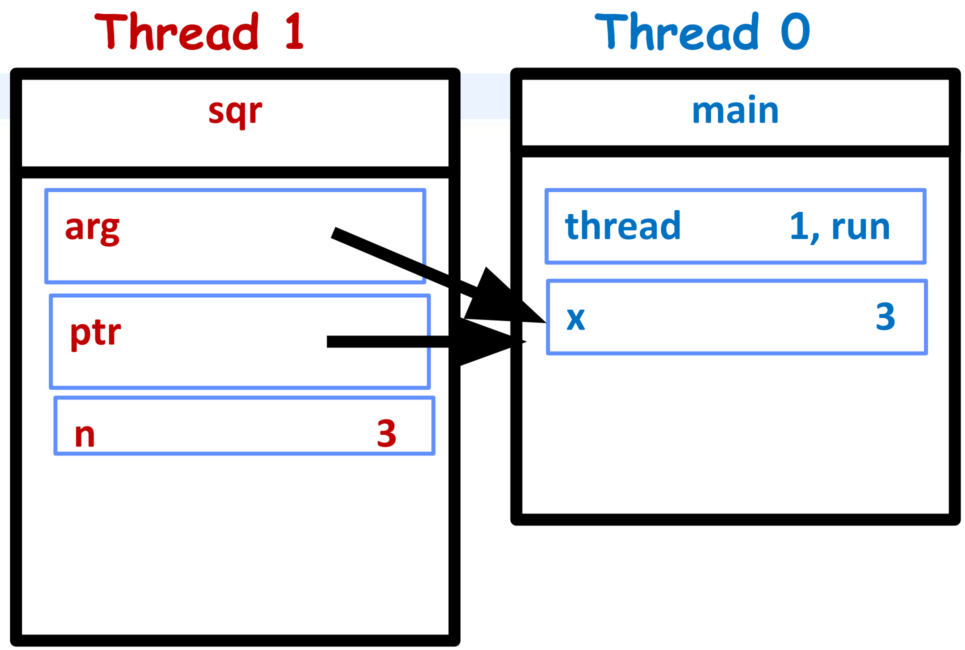
th1.c

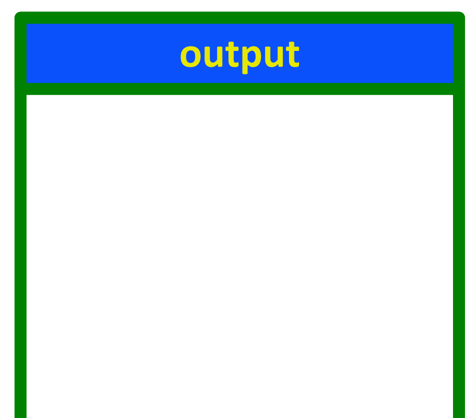
```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

void * sqr(void *arg){
int * ptr = arg;
int n = *ptr;
for (int i = 0; i < n ; i++){
int fsq = (i * i);
printf("squaare: %d\n", fsq);
}
return NULL;
}
int main(void) {
pthread_t thread;
int x = 3;
pthread_create(&thread, NULL, sqr, &x);
sleep(1);
for (int i = 0; i < 100; i++){
printf("main: %d\n", i);

}
pthread_join(thread, NULL);
printf("join completed \n");
return EXIT_SUCCESS;

}
```





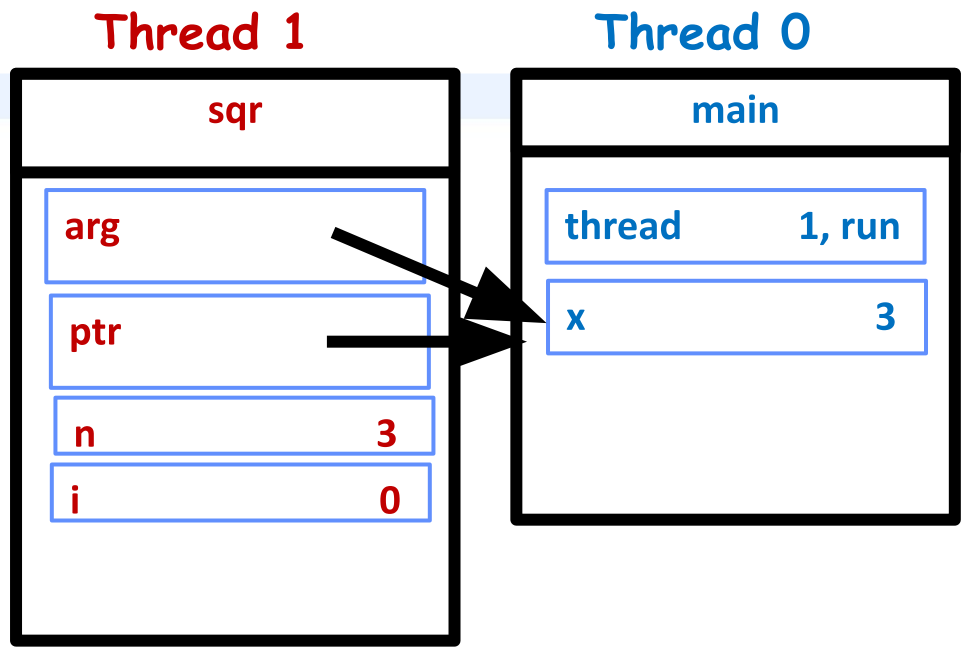
th1.c

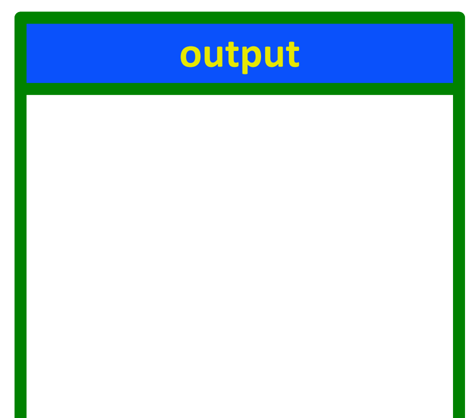
```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

void * sqr(void *arg){
int * ptr = arg;
int n = *ptr;
for (int i = 0; i < n ; i++){
int fsq = (i * i);
printf("squaare: %d\n", fsq);
}
return NULL;
}
int main(void) {
pthread_t thread;
int x = 3;
pthread_create(&thread, NULL, sqr, &x);
sleep(1);
for (int i = 0; i < 100; i++){
printf("main: %d\n", i);

}
pthread_join(thread, NULL);
printf("join completed \n");
return EXIT_SUCCESS;

}
```





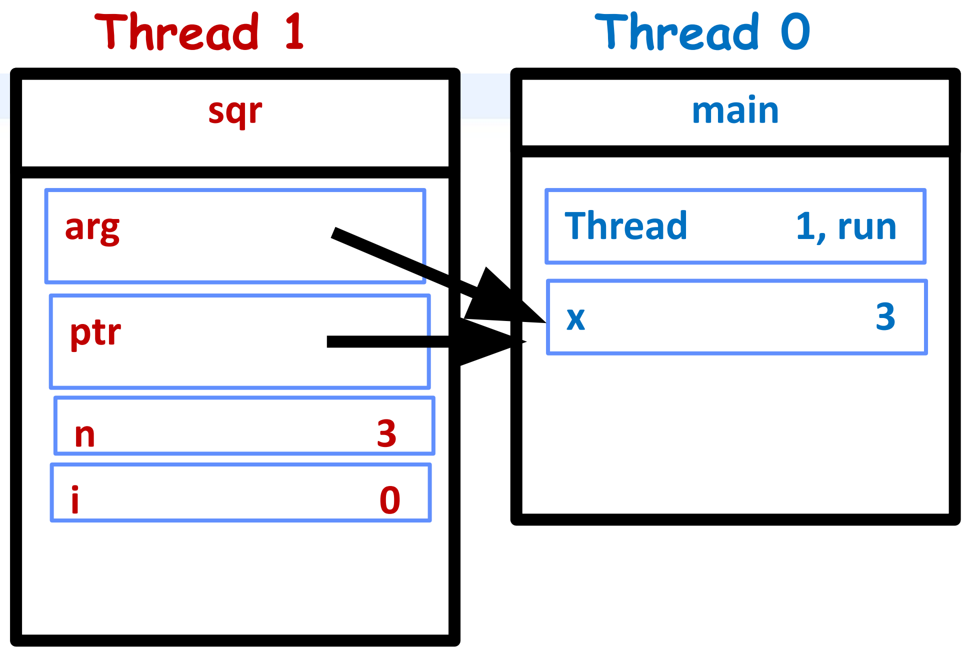
th1.c

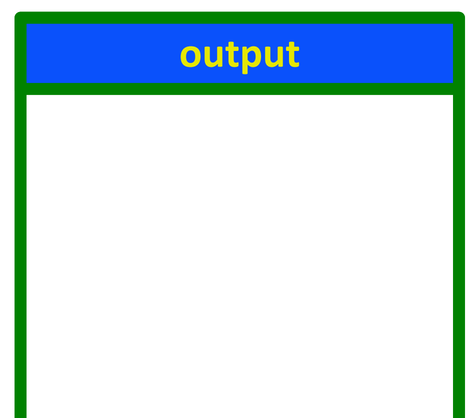
```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

void * sqr(void *arg){
int * ptr = arg;
int n = *ptr;
for (int i = 0; i < n ; i++){
int fsq = (i * i);
printf("squaare: %d\n", fsq);
}
return NULL;
}
int main(void) {
pthread_t thread;
int x = 3;
pthread_create(&thread, NULL, sqr, &x);
sleep(1);
for (int i = 0; i < 100; i++){
printf("main: %d\n", i);

}
pthread_join(thread, NULL);
printf("join completed \n");
return EXIT_SUCCESS;

}
```





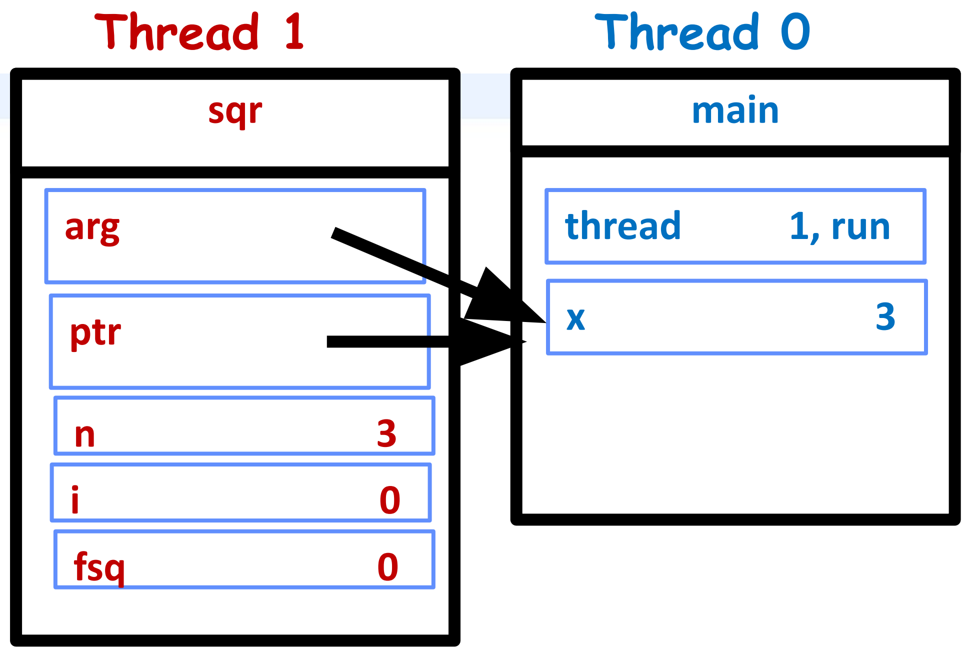
th1.c

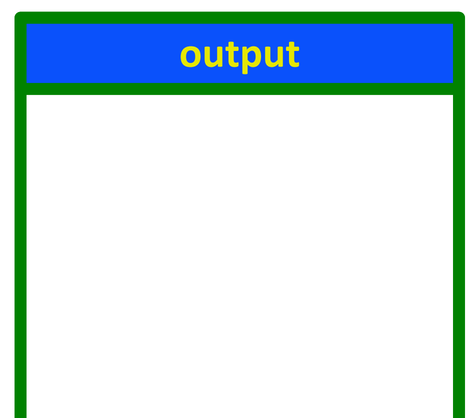
```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

void * sqr(void *arg){
int * ptr = arg;
int n = *ptr;
for (int i = 0; i < n ; i++){
int fsq = (i * i);
printf("squaare: %d\n", fsq);
}
return NULL;
}
int main(void) {
pthread_t thread;
int x = 3;
pthread_create(&thread, NULL, sqr, &x);
sleep(1);
for (int i = 0; i < 100; i++){
printf("main: %d\n", i);

}
pthread_join(thread, NULL);
printf("join completed \n");
return EXIT_SUCCESS;

}
```





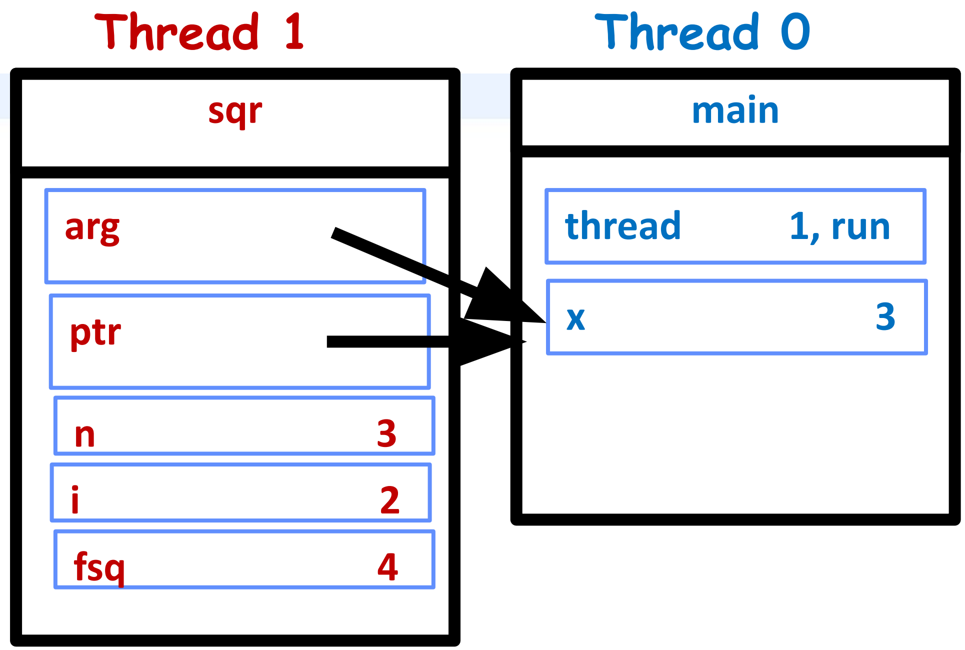
th1.c

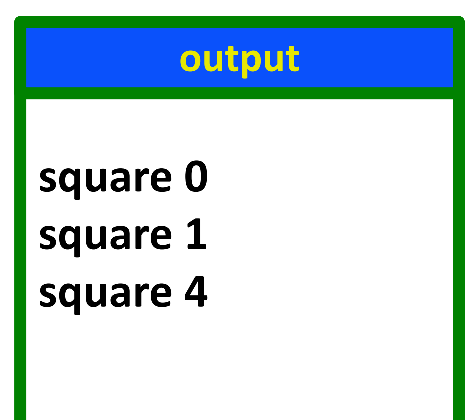
```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

void * sqr(void *arg){
int * ptr = arg;
int n = *ptr;
for (int i = 0; i < n ; i++){
int fsq = (i * i);
printf("squaare: %d\n", fsq);
}
return NULL;
}
int main(void) {
pthread_t thread;
int x = 3;
pthread_create(&thread, NULL, sqr, &x);
sleep(1);
for (int i = 0; i < 100; i++){
printf("main: %d\n", i);

}
pthread_join(thread, NULL);
printf("join completed \n");
return EXIT_SUCCESS;

}
```





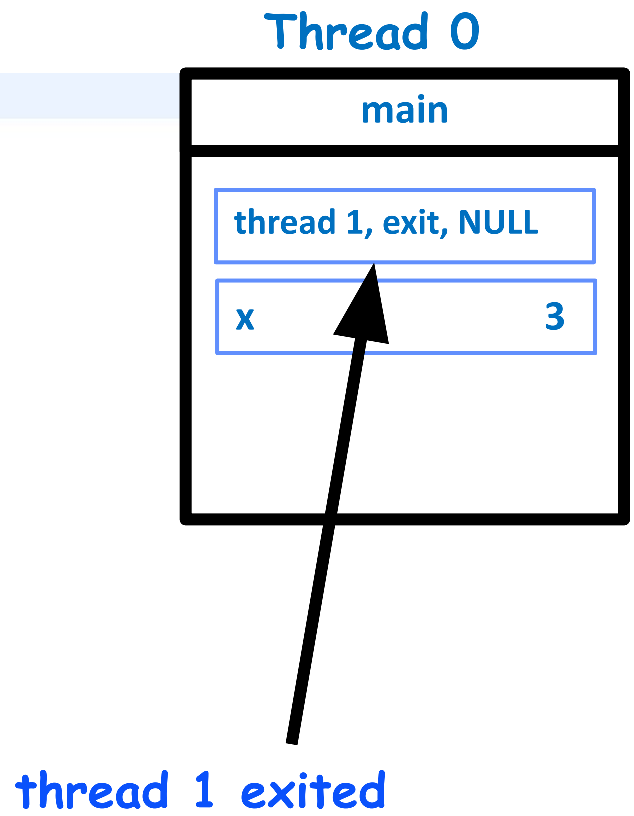
th1.c

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

void * sqr(void *arg){
int * ptr = arg;
int n = *ptr;
for (int i = 0; i < n ; i++){
int fsq = (i * i);
printf("squaare: %d\n", fsq);
}
return NULL;
}
int main(void) {
pthread_t thread;
int x = 3;
pthread_create(&thread, NULL, sqr, &x);
sleep(1);
for (int i = 0; i < 100; i++){
printf("main: %d\n", i);

}
pthread_join(thread, NULL);
printf("join completed \n");
return EXIT_SUCCESS;

}
```



output

square 0
square 1
square 4

th1.c

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

void * sqr(void *arg){
int * ptr = arg;
int n = *ptr;
for (int i = 0; i < n ; i++){
int fsq = (i * i);
printf("squaare: %d\n", fsq);
}
return NULL;
}
int main(void) {
pthread_t thread;
int x = 3;
pthread_create(&thread, NULL, sqr, &x);
sleep(1);
for (int i = 0; i < 100; i++){
printf("main: %d\n", i);

}
pthread_join(thread, NULL);
printf("join completed \n");
return EXIT_SUCCESS;

}
```

Thread 0

main	
thread	1, exit, NULL
x	3
i	0

output

main 0

th1.c

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

void * sqr(void *arg){
int * ptr = arg;
int n = *ptr;
for (int i = 0; i < n ; i++){
int fsq = (i * i);
printf("squaare: %d\n", fsq);
}
return NULL;
}
int main(void) {
pthread_t thread;
int x = 3;
pthread_create(&thread, NULL, sqr, &x);
sleep(1);
for (int i = 0; i < 100; i++){
printf("main: %d\n", i);

}
pthread_join(thread, NULL);
printf("join completed \n");
return EXIT_SUCCESS;

}
```

Thread 0

main	
thread	1, exit, NULL
x	3
i	99

output

main 0
main 1

main 99

th1.c

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

void * sqr(void *arg){
int * ptr = arg;
int n = *ptr;
for (int i = 0; i < n ; i++){
int fsq = (i * i);
printf("squaare: %d\n", fsq);
}
return NULL;
}
int main(void) {
pthread_t thread;
int x = 3;
pthread_create(&thread, NULL, sqr, &x);
sleep(1);
for (int i = 0; i < 100; i++){
printf("main: %d\n", i);

}
pthread_join(thread, NULL);
printf("join completed \n");
return EXIT_SUCCESS;

}
```

Thread 0

main	
thread	1, exit, NULL
x	3
i	99

Thread 0 unblocks

output
main 0
main 1
main 99
join competed

Lecture Summary

- We have used pthread library's APIs
  - `pthread_create`
  - `pthread_join`