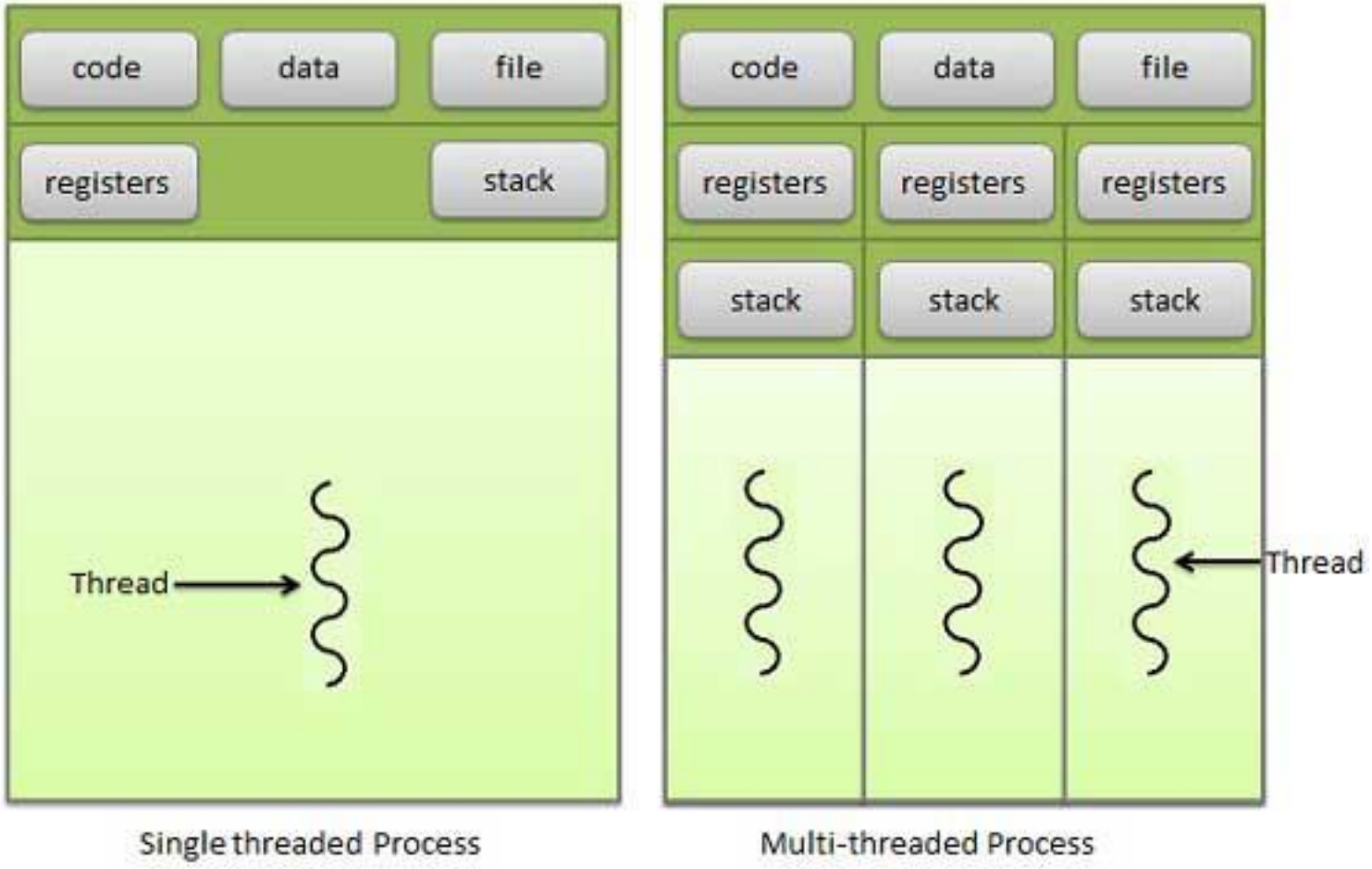
Operating Systems

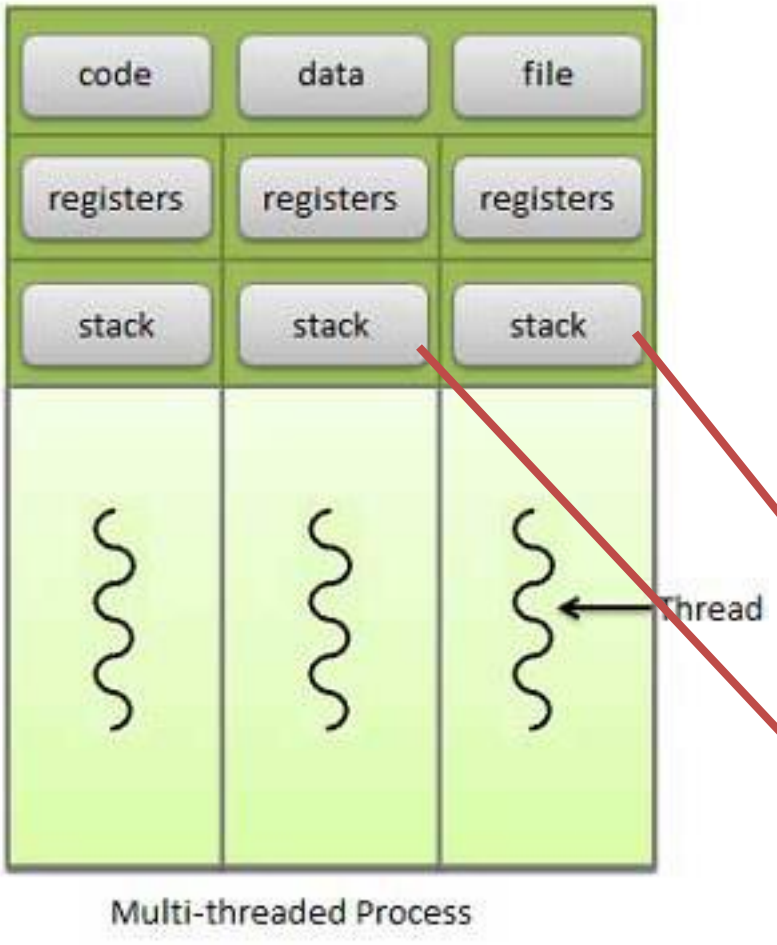
Eric Lo

7 – Multi-threading

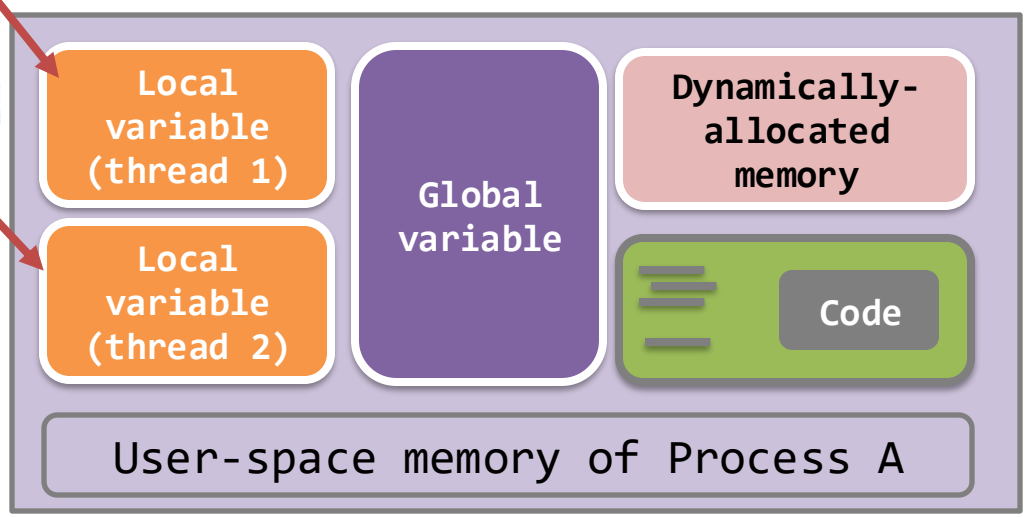
Single-threaded process vs. multi-threaded



Multi-threading



- Threads share the same code
  - From the same program
- Threads share the same address space
  - Threads can then share via global variables
- Stacks of non-parent threads are located somewhere in the memory-map segment



Process vs. Thread

Process	Thread
Heavy weight	Light weight
Costly creation	Cheap creation
Process can't access the memory area of other processes	Threads share the same memory area
Costly process switching	Cheap thread switching
One process multiple threads	
Different programs	Same program (different thread functions)

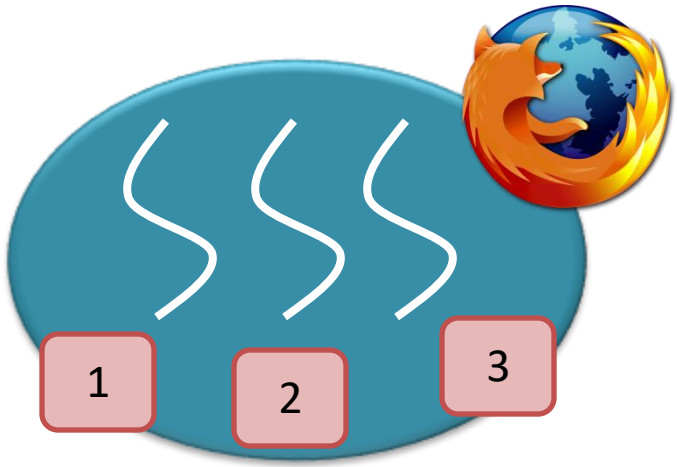
While thread "sounds" much better than a process technically

Don't put the cart before the horse

You won't imagine the same program "ls" has one thread doing the job of "ls" and another thread doing the job of "less" → they should be 2 different programs

Multi-thread examples: pros & cons

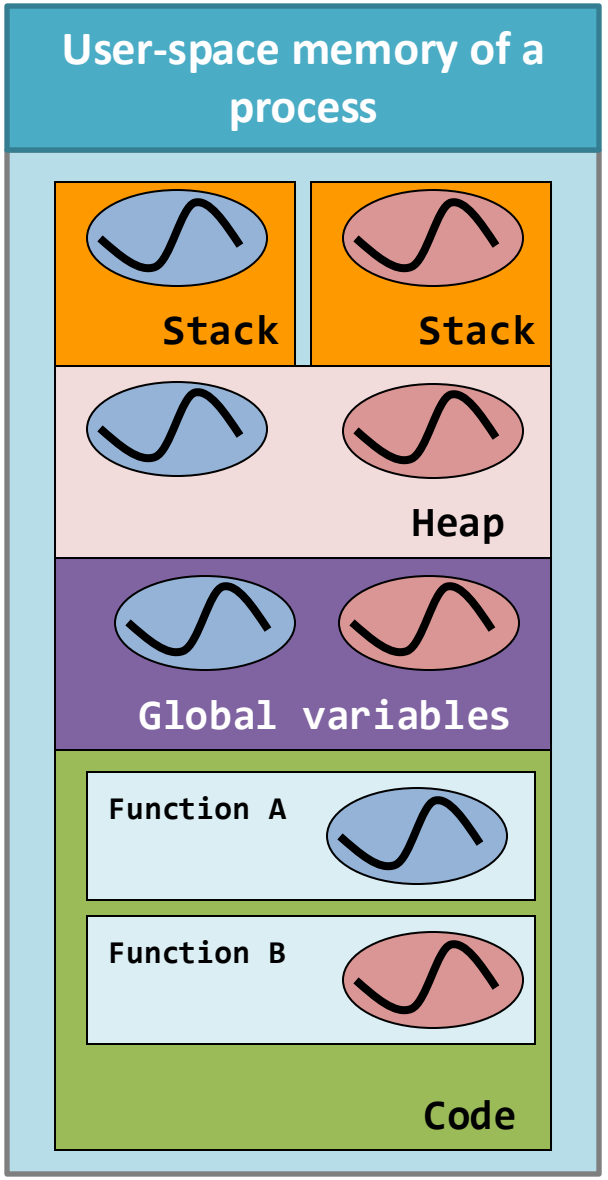
- Old Firefox
  - Single-process
  - Multi-threaded
    - Thread 1: Tab1
    - Thread 2: Tab2
  - Faster
  - Crash one? Crash all!
- Chrome
  - 1 tab 1 process



Multi-thread – Code and thread function

Code

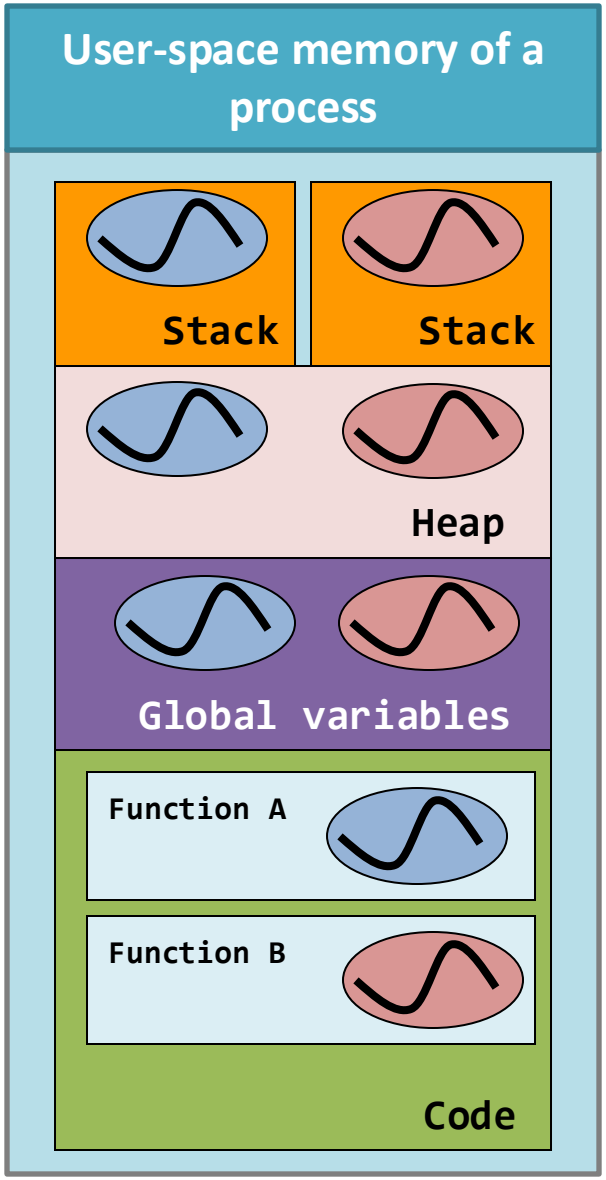
- All threads share the same code
- A thread starts with **one specific function**
  - the **thread function**
  - E.g., Functions A & B in the diagram.
- Of course, the thread function can invoke other functions or system calls.
- But, a thread **never returns to the caller of the thread function**



Multi-thread – Local Variables

Local variables

- Each thread has **its own stack** for the local variables.
- i.e., thread function's () {
  int x;
  }
- Thread 1 updates her x != updating Thread 2's x

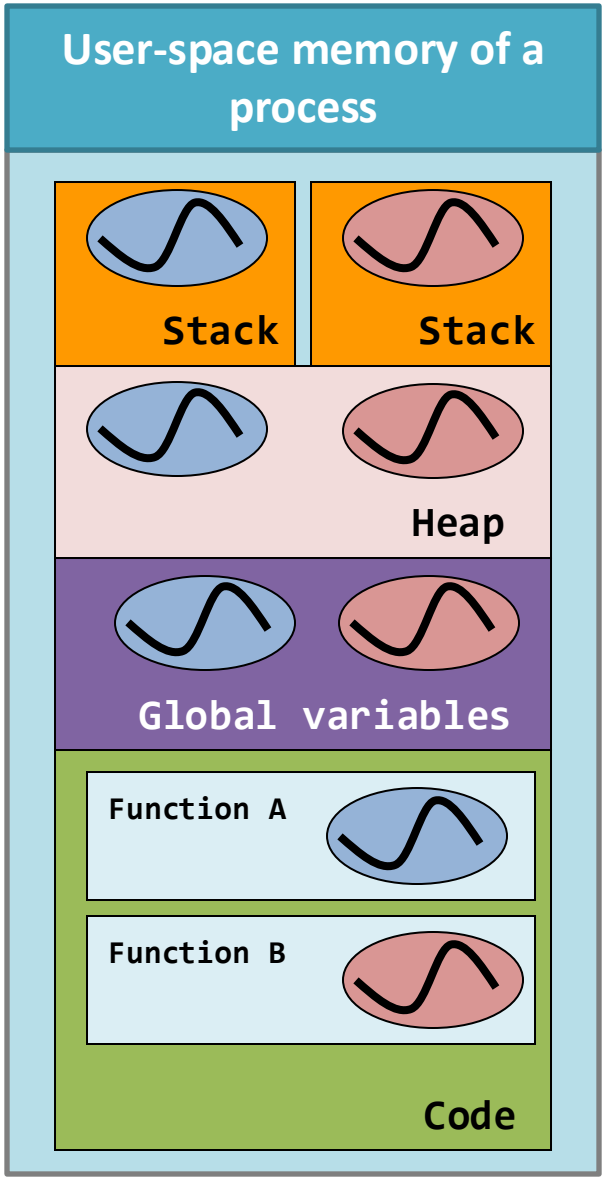


Multi-thread – Global Variables + Heap

Heap

Global variables

- All threads share the **global variable zone** and the **heap**
- i.e.,
- int g;
- thread function's () {
  int x;
  }
- Thread 1 updates g ➔ Thread 2 see that change



Multi-threading

- Introduction.
- Basic Programming.

The Pthread library

- POSIX `thread` library

	Process	Thread
Creation	`fork()`	`pthread_create()`
I.D. Type	`PID, an integer`	`“pthread_t”, a structure`
Who am I?	`getpid()`	`pthread_self()`
Termination	`exit()`	`pthread_exit()`
Wait for child termination	`wait() or waitpid()`	`pthread_join()`
Kill	`kill()`	`pthread_kill()`

Thread creation – `pthread_create()`

Thread Function

```
1  void * hello( void *input ) {
2      printf("%s\n", (char *) input);
3      pthread_exit(NULL);
4  }
```

Main Function

```
5  int main(void) {
6      pthread_t tid;
7      pthread_create(&tid, NULL, hello, "hello world");
8      pthread_join(tid, NULL);
9      return 0;
10 }
```

This sets the thread function of the to-be-created thread as: **`hello()`**.

```
[examples@3150] cat pthread_hello_world.c
```

Thread creation – pthread_create()

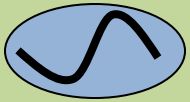
Thread Function

```
1  void * hello( void *input ) {
2      printf("%s\n", (char *) input);
3      pthread_exit(NULL);
4  }
```

Main Function

```
5  int main(void) {
6      pthread_t tid;
7      pthread_create(&tid, NULL, hello, "hello world");
8      pthread_join(tid, NULL);
9      return 0;
10 }
```

At the beginning, there is only one thread running: **the main thread**.



Main Thread

```
[examples@3150] cat pthread_hello_world.c
```

Thread creation – pthread_create()

Thread Function

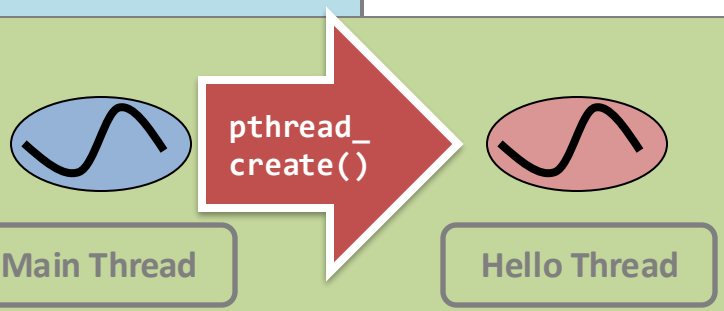
```
1  void * hello( void *input ) {
2      printf("%s\n", (char *) input);
3      pthread_exit(NULL);
4  }
```

Main Function

```
5  int main(void) {
6      pthread_t tid;
7      pthread_create(&tid, NULL, hello, "hello world");
8      pthread_join(tid, NULL);
9      return 0;
10 }
```

The hello thread is created!

It is running "*together*" with the main thread.




```
[examples@3150] cat pthread_hello_world.c
```

Thread creation – `pthread_create()`

Thread Function

```
1  void * hello( void *input ) {
2      printf("%s\n", (char *) input);
3      pthread_exit(NULL);
4  }
```

Main Function

```
5  int main(void) {
6      pthread_t tid;
7      pthread_create(&tid, NULL, hello, "hello world");
8      pthread_join(tid, NULL);
9      return 0;
10 }
```

Remember **`wait()`** and **`waitpid()`**?

`pthread_join()` performs similarly.


```
[examples@3150] cat pthread_hello_world.c
```

Thread creation – pthread_create()

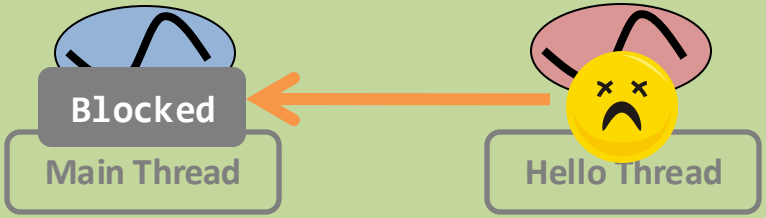
Thread Function

```
1  void * hello( void *input ) {
2      printf("%s\n", (char *) input);
3      pthread_exit(NULL);
4  }
```

Main Function

```
5  int main(void) {
6      pthread_t tid;
7      pthread_create(&tid, NULL, hello, "hello world");
8      pthread_join(tid, NULL);
9      return 0;
10 }
```

Termination of the target thread causes `pthread_join()` to return.




```
[examples@3150] cat pthread_hello_world.c
```

Shared address space

Thread Function

```
 1  void * do_your_job( void *input ) {
 2      printf("child = %d\n", *( (int *) input) );
 3      *((int *) input) = 20;
 4      printf("child = %d\n", *( (int *) input) );
 5      pthread_exit(NULL);
 6  }
```

Main Function

```
 7  int main(void) {
 8      pthread_t tid;
 9      int input = 10;
10      printf("main = %d\n", input);
11      pthread_create(&tid, NULL, do_your_job, &input);
12      pthread_join(tid, NULL);
13      printf("main = %d\n", input);
14      return 0;
15  }
```

```
$ ./pthread_sharing
main = 10
child = 10
child = 20
main = 20
$
```

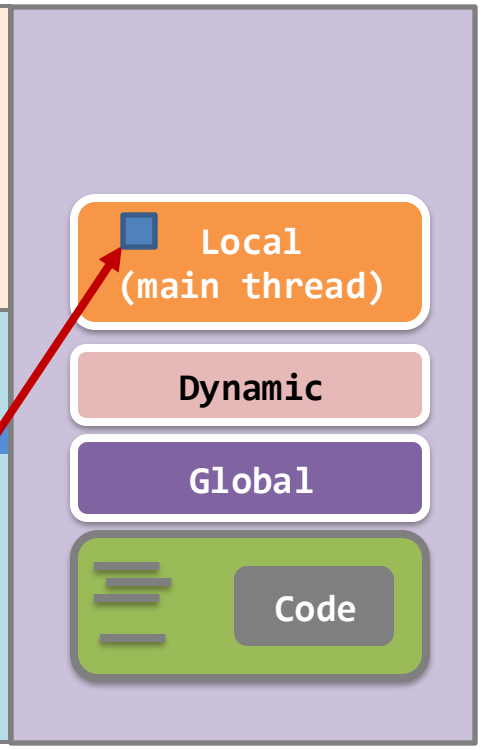
```
[examples@3150] cat pthread_sharing.c
```

Shared address space

The local variable "**input**" is in **the stack of the main thread**.

```
 1  void * do_your_job( void *input ) {
 2      printf("child = %d\n", *( (int *) input) );
 3      *((int *) input) = 20;
 4      printf("child = %d\n", *( (int *) input) );
 5      pthread_exit(NULL);
 6  }

 7  int main(void) {
 8      pthread_t tid;
 9      int input = 10;
10      printf("main = %d\n", input);
11      pthread_create(&tid, NULL, do_your_job, &input);
12      pthread_join(tid, NULL);
13      printf("main = %d\n, input);
13      return 0;
14  }
```



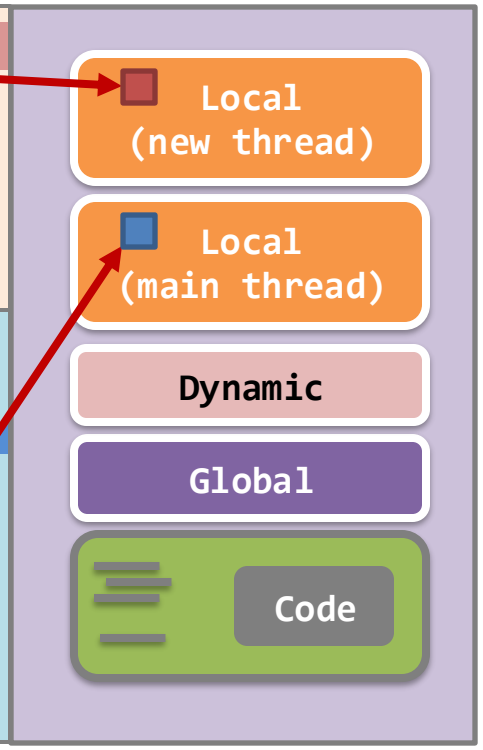

```
[examples@3150] cat pthread_sharing.c
```

Shared address space

Yet…the stack for the new thread is not on the another process, but is **on the same piece of user-space memory as the main thread**.

```
 1  void * do_your_job( void *input ) {
 2      printf("child = %d\n", *( (int *) input) );
 3      *((int *) input) = 20;
 4      printf("child = %d\n", *( (int *) input) );
 5      pthread_exit(NULL);
 6  }

 7  int main(void) {
 8      pthread_t tid;
 9      int input = 10;
10      printf("main = %d\n", input);
11      pthread_create(&tid, NULL, do_your_job, &input);
12      pthread_join(tid, NULL);
13      printf("main = %d\n, input);
13      return 0;
14  }
```



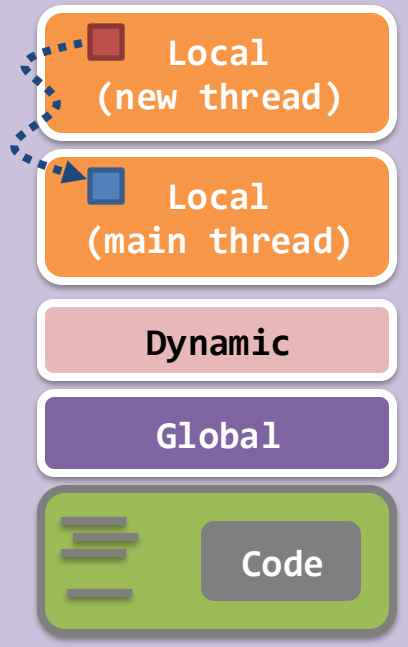

```
[examples@3150] cat pthread_sharing.c
```

Shared address space

The **pthread_create()** function passes an address to the new thread in the same process space

```
 1  void * do_your_job( void *input ) {
 2      printf("child = %d\n", *( (int *) input) );
 3      *((int *) input) = 20;
 4      printf("child = %d\n", *( (int *) input) );
 5      pthread_exit(NULL);
 6  }

 7  int main(void) {
 8      pthread_t tid;
 9      int input = 10;
10      printf("main = %d\n", input);
11      pthread_create(&tid, NULL, do_your_job, &input);
12      pthread_join(tid, NULL);
13      printf("main = %d\n, input);
13      return 0;
14  }
```



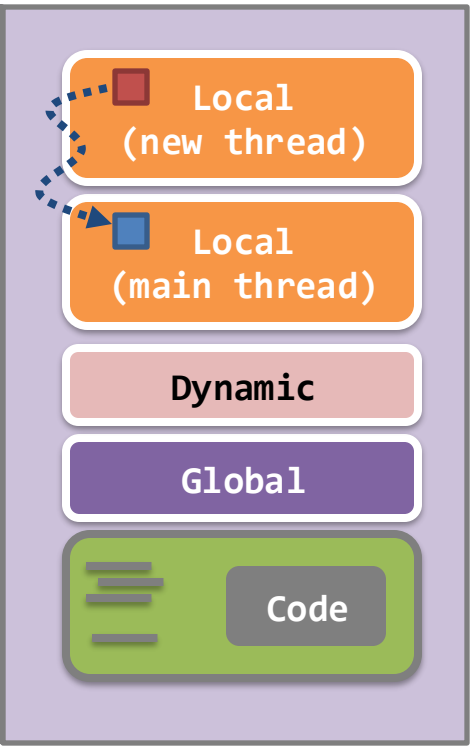

```
[examples@3150] cat pthread_sharing.c
```

Shared address space

Therefore, the new thread can change the value in the main thread, and **vice versa**.

```
 1  void * do_your_job( void *input ) {
 2      printf("child = %d\n", *( (int *) input) );
 3      *((int *) input) = 20;
 4      printf("child = %d\n", *( (int *) input) );
 5      pthread_exit(NULL);
 6  }

 7  int main(void) {
 8      pthread_t tid;
 9      int input = 10;
10      printf("main = %d\n", input);
11      pthread_create(&tid, NULL, do_your_job, &input);
12      pthread_join(tid, NULL);
13      printf("main = %d\n, input);
13      return 0;
14  }
```




```
[examples@3150] cat pthread_sharing.c
```

Thread termination – passing return value?

Thread Function

```
 1  void * do_your_job(void *input) {
 2       int *output = (int *) malloc(sizeof(int));
 3       srand(time(NULL));
 4       *output = ((rand() % 10) + 1) * (*((int *) input));
 5       pthread_exit( output );
 6  }
```

On thread exit, give pthread library the address of where the return value stored

Main Function

```
 7  int main(void) {
 8       pthread_t tid;
 9       int input = 10, *t-output;
10       pthread_create(&tid, NULL, do_your_job, &input);
11       pthread_join(tid, (void **) &t-output );
12       printf("output = %d\n", *t-output);
13       return 0;
14  }
```

Pass to the library the address of a space that the main set aside to receive the thread's return value

```
[examples@3150] cat pthread_exit.c
```

Kernel

- The Kernel itself is a multi-threaded program
- The threads created by Kernel are named
  - kernel threads
  - OS threads

Debunking SMT / Hyper-threading

- Simultaneous Multi-Threading (SMT)
  - It refers to **hardware** threading
  - **Not** pThread/OS level **software** threads
- Each core simulates two (virtual) cores
  - So, when a CPU sells you as 8 cores, look at the specification more carefully what 8 cores really are!
  - Virtual cores alternating between them on a physical cycle-by-cycle basis
    - When one hardware thread waits for memory
    - Another hardware thread can use the physical core to do computation