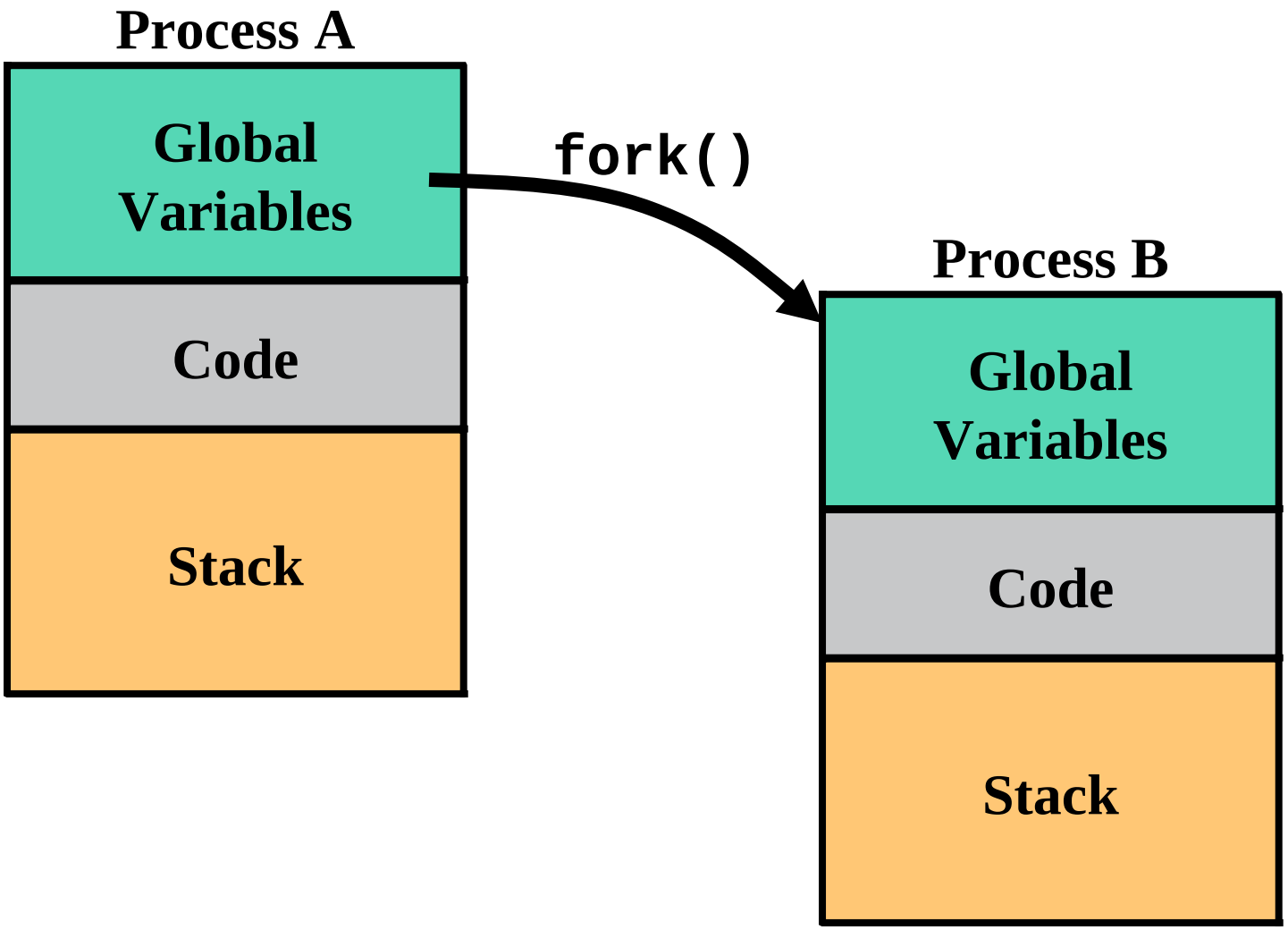
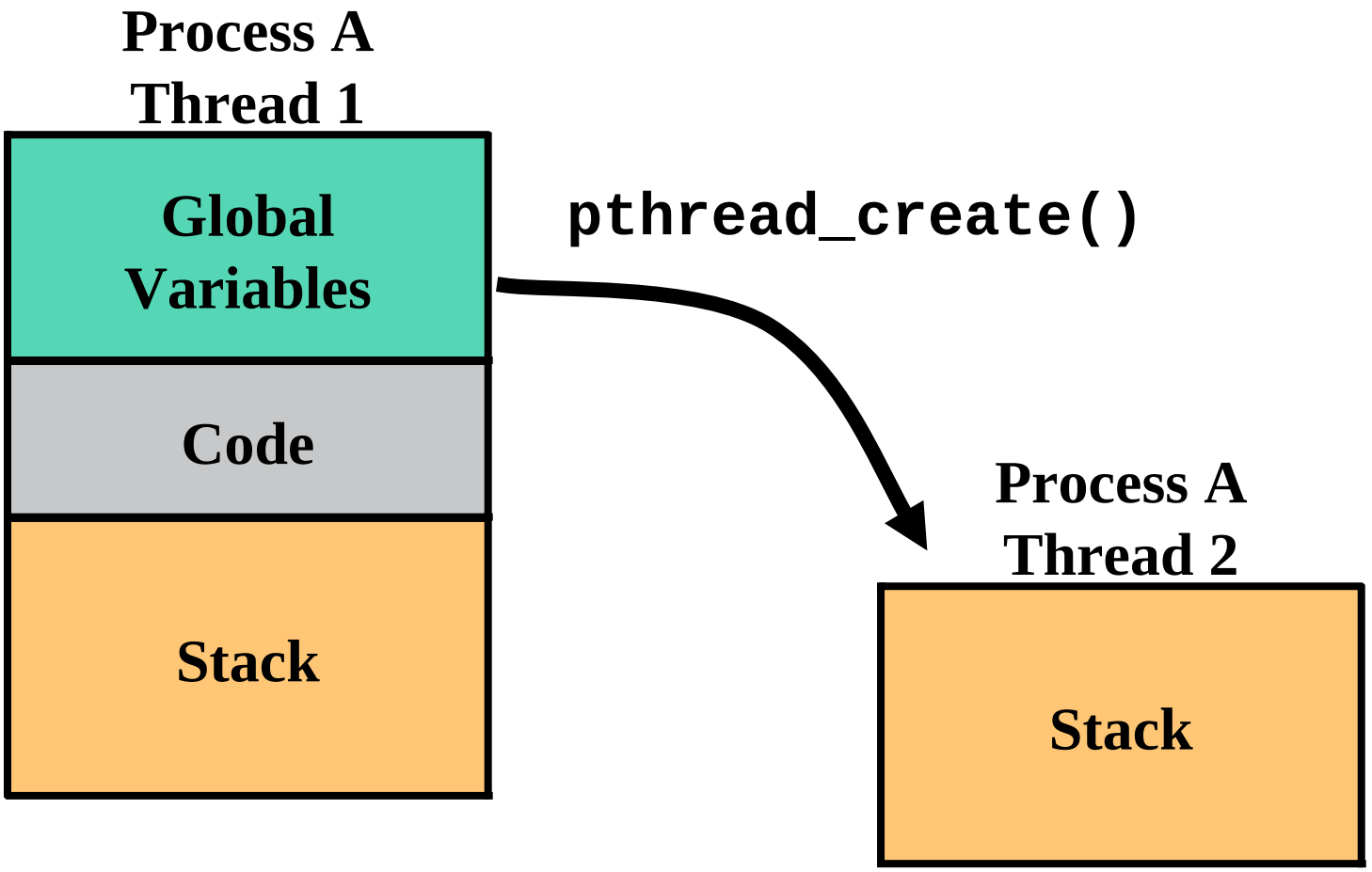
Threads Programming

fork()


Process A
Global Variables
Code
Stack
fork()
Process B
Global Variables
Code
Stack

pthread_create()



Thread-Specific Resources

Each thread has it's own:

- Thread ID (thread_t)
- Stack, Registers, Program Counter
- **`errno`** (if not - **`errno`** would be useless!)

Threads within the same process can communicate using shared memory.

Must be done carefully!

Posix Threads

You need to link with **"`-lpthread`"**

```
gcc -pthread
```

On many systems this also forces the compiler to link in *re-entrant* libraries (instead of plain vanilla C libraries).

Thread Creation

```
pthread_create(
   pthread_t *tid,
   const pthread_attr_t *attr,
   void *(*func)(void *),
   void *arg);
```

`func` is the function to be called.

When **`func()`** returns the thread is terminated.

`pthread_create()`

- The return value is 0 for OK.
- Thread ID is returned in **`tid`**
- Thread attributes can be set using **attr**,
  - including detached state and scheduling policy
  - NULL gets the system defaults.

Thread IDs

Each thread has a unique ID, a thread can find out it's ID by calling **`pthread_self().`**

Thread IDs are of type **`pthread_t`** which is usually an unsigned int. When debugging, it's often useful to do something like this:

```
printf("Thread %u:\n",pthread_self());
```

Thread Arguments

When **`func()`** is called the value **`arg`** specified in the call to **`pthread_create()`** is passed as a parameter.

`func` can have only 1 parameter, and it can't be larger than the size of a **`void *`**.

Thread Arguments (cont.)

Complex parameters can be passed by creating a structure and passing the address of the structure.

The structure can't be a local variable (of the function calling pthread_create)!!

- threads have different stacks!

Thread Lifespan

Once a thread is created, it starts executing the function **`func()`** specified in the call to **`pthread_create()`.**

If **`func()`** returns, the thread is terminated.

A thread can also terminate by calling **`pthread_exit()`.**

If **`main()`** returns or any thread calls **`exit()`**all threads are terminated.

Detached State

Each thread can be either *joinable* or *detached*.

Detached: on termination all thread resources are released by the OS. A detached thread cannot be joined.

No way to get at the return value of the thread. (a pointer to something: **`void *`**).

Joinable Thread

Joinable: on thread termination the thread ID and exit status are saved by the OS.

One thread can "join" another by calling **`pthread_join`** - which waits (blocks) until a specified thread exits.

```
int pthread_join( pthread_t tid,
                  void **status);
```

Shared Global Variables

```
int counter=0;
void *pancake(void *arg) {
  counter++;
  printf("Thread %u is number %d\n",
       pthread_self(),counter);
}
main() {
  int i; pthread_t tid;
  for (i=0;i<10;i++)
    pthread_create(&tid,NULL,pancake,NULL);
}
```

DANGER! DANGER! DANGER!

Sharing global variables is dangerous

- Two threads may attempt to modify the same variable at the same time.

So:

- Avoid using globals, unless necessary.
- Allocate variable on stack (scalars) or use malloc()/free()

Avoiding Problems

pthreads includes support for *Mutual Exclusion* primitives that can be used to protect against this problem.

The general idea is to *lock* something before accessing global variables and to *unlock* as soon as you are done.

Shared socket descriptors should be treated as global variables

pthread_mutex

A global variable of type **pthread_mutex_t** is required:

```
pthread_mutex_t counter_mtx=
  PTHREAD_MUTEX_INITIALIZER;
```

Initialization to **PTHREAD_MUTEX_INITIALIZER** is required for a static variable

Locking and Unlocking

- To lock use:

```
pthread_mutex_lock(pthread_mutex_t *);
```

- To unlock use:

```
pthread_mutex_unlock(pthread_mutex_t *);
```

- Both functions are blocking

Condition Variables

pthreads also support *condition variables*, which allow one thread to wait (sleep) for an event generated by any other thread.

This allows us to avoid the *busy waiting* problem.

```
pthread_cond_t foo =
  PTHREAD_COND_INITIALIZER;
```

Condition Variables (cont.)

A condition variable is **always** used with mutex.

```
pthread_cond_wait(pthread_cond_t *cptr,
                  pthread_mutex_t *mptr);

pthread_cond_signal(pthread_cond_t
  *cptr);
```

don't let the word signal confuse you - this has nothing to do with Unix signals

Condition Variables (cont.)

- Lock the mutex and then wait on condvar.
- Upon wakeup the lock is held

```
pthread_mutex_lock (&mutex);
pthread_cond_wait (&cond, &mutex);
…
pthread_mutex_unlock (&mutex);
```

Other Sync. Functions

- Posix Semaphores

```
#include <semaphore.h>
sem_t s;
sem_init(sem_t *s, int shared, uint_t value);
sem_wait(sem_t *s);
sem_post(sem_t *s);
sem_getvalue(sem_t *s);
sema_destroy(sem_t *s);
```

Thread Safe library functions

- You have to be careful with libraries.
- If a function uses any static variables (or global memory) it’s not safe to use with threads!