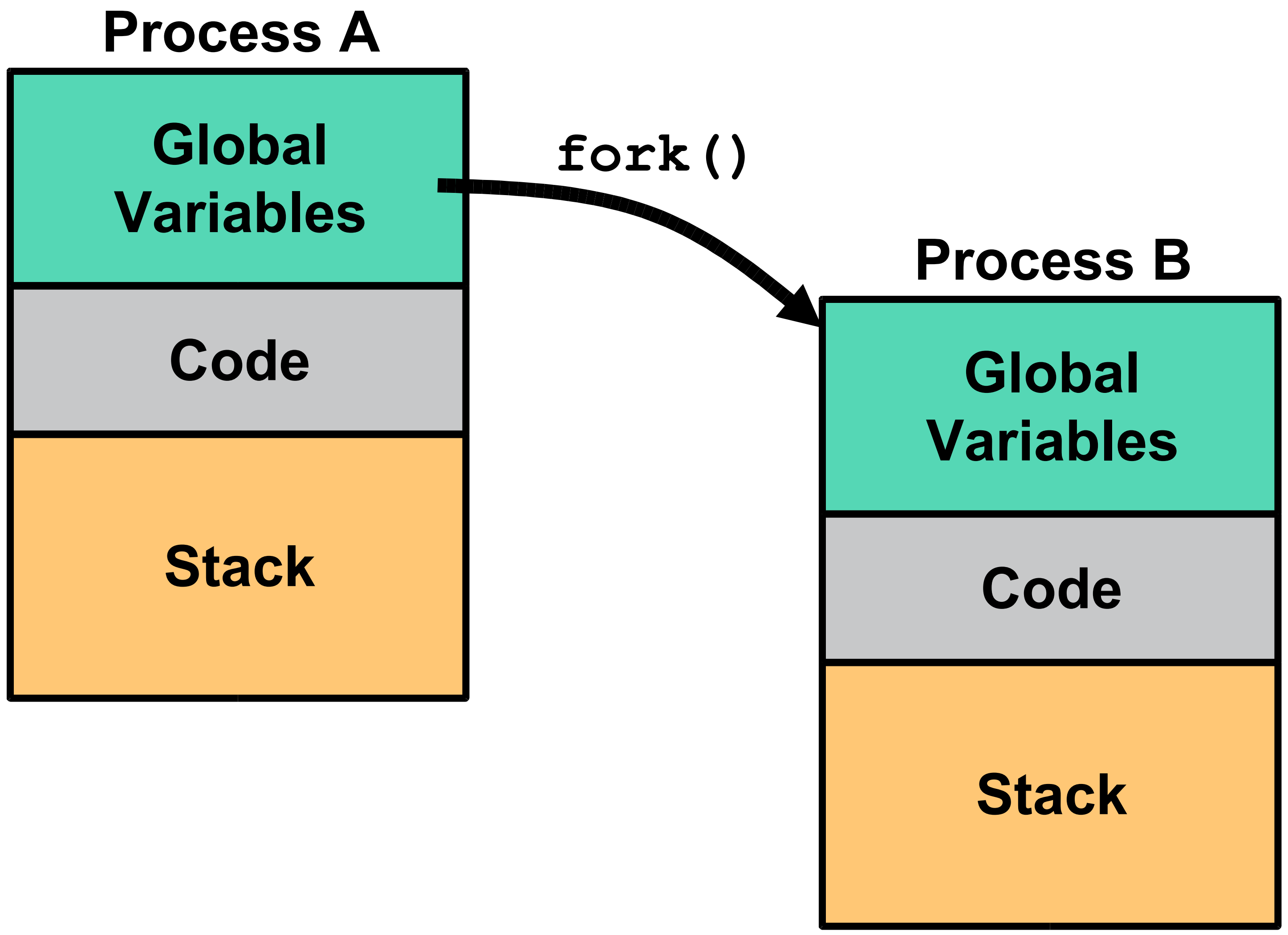
Threads Programming

Threads vs. Processes

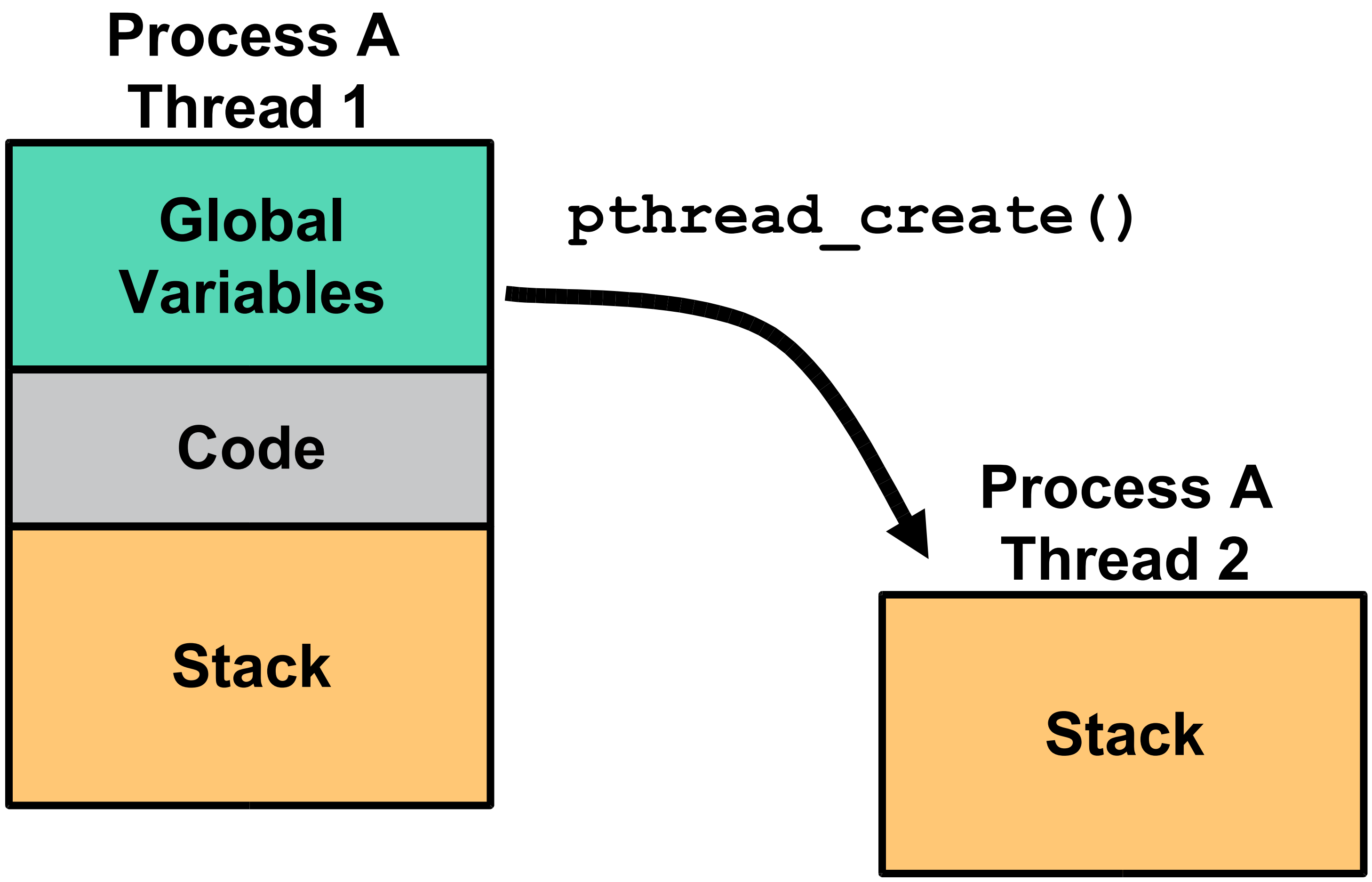
Creation of a new process using fork is *expensive* (time & memory).

A thread (sometimes called a *lightweight process*) does not require lots of memory or startup time.

`fork()`


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

`pthread_create()`



Multiple Threads

Each process can include many threads.

All threads of a process share:

- memory (program code and global data)
- open file/socket descriptors
- signal handlers and signal dispositions
- working environment (current directory, user ID, etc.)

Thread-Specific Resources

Each thread has it’s own:

- Thread ID (integer)
- 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

We will focus on *Posix Threads* - most widely supported threads programming API.

You need to link with “**`-lpthread`**”

```
    gcc -threads
```

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 args example

```
struct { int x,y } 2ints;

void *blah( void *arg) {
  struct 2ints *foo = (struct 2ints *) arg;
  printf("%u sum of %d and %d is %d\n",
     pthread_self(), foo->x, foo->y,
     foo->x+foo->y);
  return(NULL);
}
```

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 be terminated 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.

Just because you don't see a problem when running your code doesn't mean it can't and won't happen!!!!

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!

Example Problem

A server creates a thread for each client. No more than *n* threads (and therefore *n* clients) can be active at once.

How can we have the main thread know when a child thread has terminated and it can now service a new client?

`pthread_join()` doesn’t help

`pthread_join` (which is sort of like **`wait()`**) requires that we specify a thread id.

We can wait for a specific thread, but we can't wait for "the next thread to exit".

Use a global variable?

When each thread starts up:

– acquires a lock on the variable (using a mutex)
– increments the variable
– releases the lock.

When each thread shuts down:

– acquires a lock on the variable (using a mutex)
– decrements the variable
– releases the lock.

What about the main loop?

```
active_threads=0;
// start up n threads on first n clients
// make sure they are all running
while (1) {
   // have to lock/relase active_threads
   if (active_threads < n)
        // start up thread for next client
   busy_waiting(is_bad);
}
```

Condition Variables

pthreads 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

Revised strategy

Each thread decrements **`active_threads`** when terminating and calls **`pthread_cond_signal`** to wake up the main loop.

The main thread increments **`active_threads`** when each thread is started and waits for changes by calling **`pthread_cond_wait`**

Revised strategy

All changes to **`active_threads`** must be inside the lock and release of a mutex.

If two threads are ready to exit at (nearly) the same time – the second must wait until the main loop recognizes the first.

We don’t lose any of the condition signals.

Global Variables

```
// global variable the number of active
// threads (clients)
int active_threads=0;

// mutex used to lock active_threads
pthread_mutex_t at_mutex =
  PTHREAD_MUTEX_INITIALIZER;

// condition var. used to signal changes
pthread_cond_t at_cond =
  PTHREAD_COND_INITIALIZER;
```

Child Thread Code

```
void *cld_func(void *arg) {
  . . .
  // handle the client
  . . .
  pthread_mutex_lock(&at_mutex);
  active_threads--;
  pthread_cond_signal(&at_cond);
  pthread_mutex_unlock(&at_mutex);
  return();
}
```

Main thread

```
// no need to lock yet
active_threads=0;
while (1) {
  pthread_mutex_lock(&at_mutex);
  while (active_threads < n ) {
     active_threads++;
     pthread_start(…)
  }
  pthread_cond_wait( &at_cond, &at_mutex);
  pthread_mutex_unlock(&at_mutex);
}
```

IMPORTANT! Must happen while the mutex lock is held.

Other pthread functions

Sometimes a function needs to have thread specific data (for example, a function that uses a static local).

Functions that support thread specific data:

`pthread_key_create()`

`pthread_once()`

`pthread_getspecific()`

`pthread_setspecific()`

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!

Thread Summary

Threads are awesome, but dangerous. You have to pay attention to details or it's easy to end up with code that is incorrect (doesn't always work, or hangs in deadlock).

Posix threads provides support for mutual exclusion, condition variables and thread-specific data.