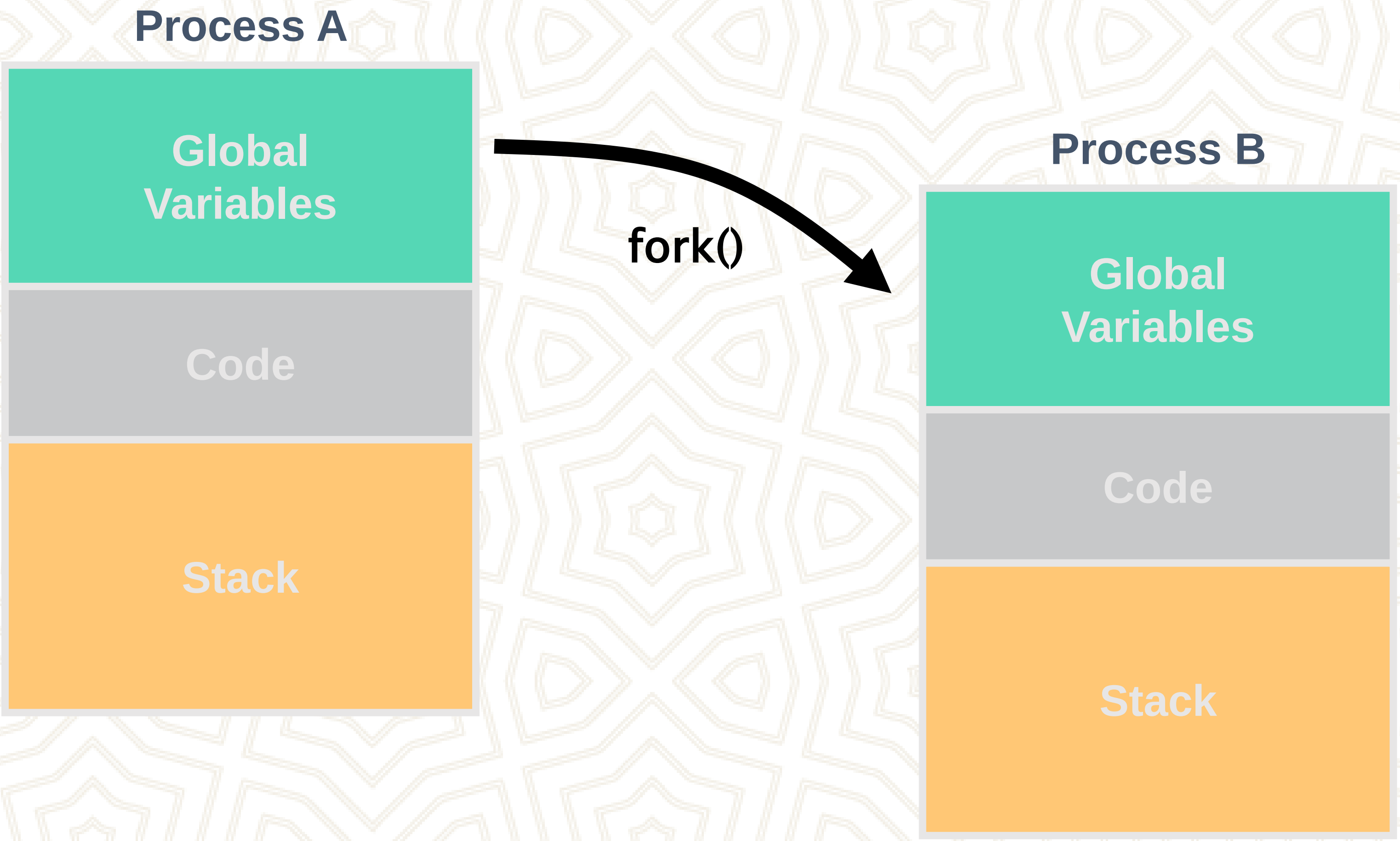
Threads vs. Processes

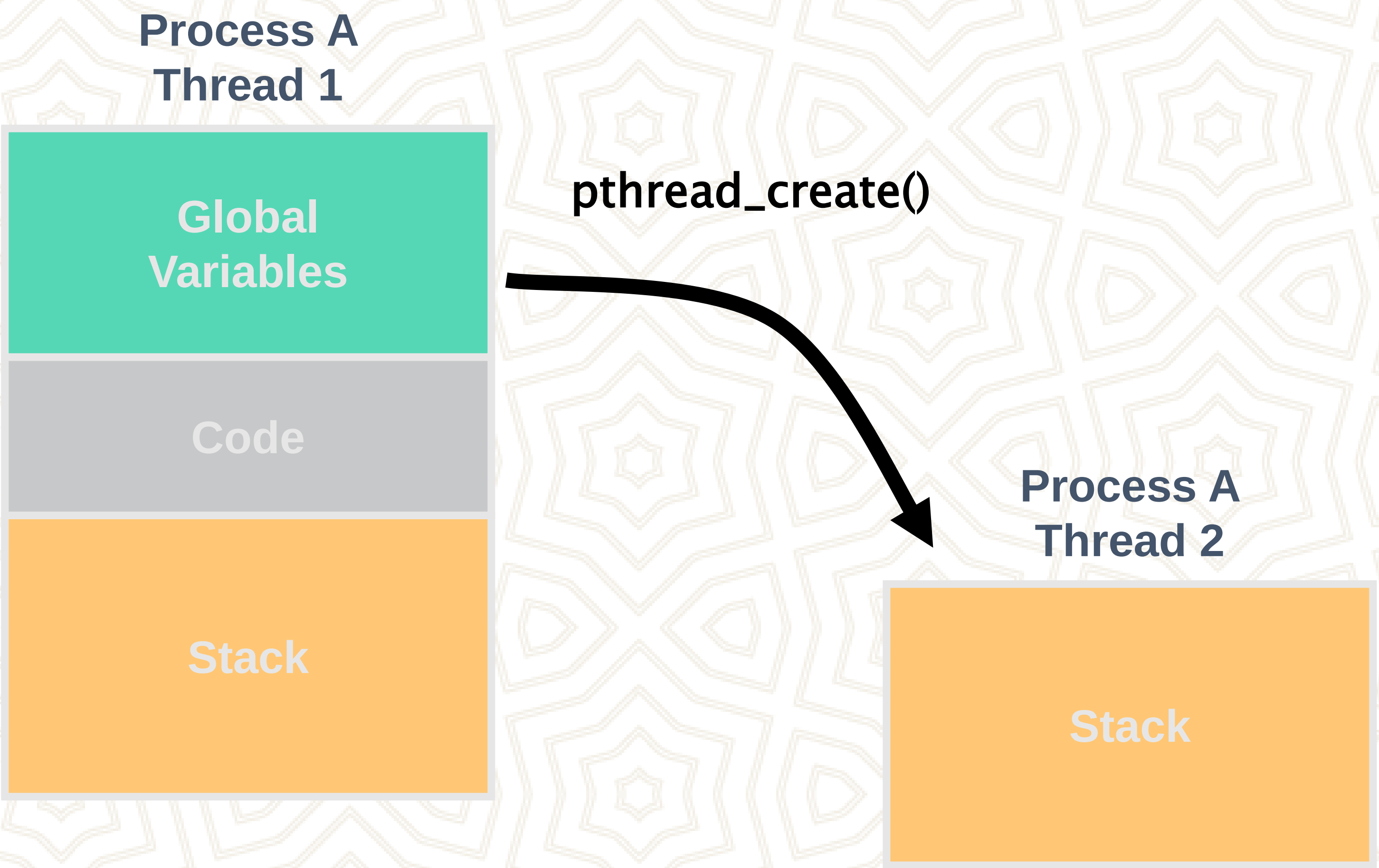
- Creation of a new process using fork is *expensive* (time & memory).

- 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 its own
  - Thread ID
  - 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**”

- 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.
  - positive error number on error.

- Does not set errno !!!

- Thread ID is returned in **tid**

pthread_create()

Creates a new thread executing a start routine (callback). function.

```
#include <pthread.h>

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

On success, the ID of the created thread will be stored here.

What does this mean?

Return type of the function

Name of function pointer

Type of parameter to the function

```
void * ( * start_routine ) ( void * )
```

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);
}
```

Pthread workflow

`main()`

When a process starts execute in the `main()` function, there is one thread of control.

`pthread_create()`

Creates a new thread and gives it a start routine (callback) to execute.

`start_routine()`

The new thread execute the start routine (callback).

`pthread_exit()`

The new threads calls `pthread_exit()` to terminate.

`pthread_join()`

Wait for the new thread to terminate.

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.

pthreads_create_exit_null_join.c

This program creates four threads and wait for all of them to terminate.

```
$ ./bin/pthreads_create_exit_null_join
main() - before creaing new threads
  thread 0 - hello
  thread 1 - hello
  thread 2 - hello
  thread 3 - hello
main() - thread 0 terminated
main() - thread 1 terminated
main() - thread 2 terminated
main() - thread 3 terminated
main() - all new threads terminated
$
```

Ex_1_pthread1.c

```
void* hello(void* arg) {
  int i = *(int*) arg;
  printf("  thread %d - hello\n", i);
  pthread_exit(NULL);
}
```

This is the start routine each of the threads will execute.

Every start routine must take `void*` as argument and return `void*`.

When creating a new thread we will use a pointer to an integer as argument, pointing to an integer with the thread number.

Here we first cast from `void*` to `int*` and then dereference the pointer to get the integer value.

Terminate the thread by calling `pthread_exit(NULL)`. Here `NULL` means we don't specify a termination status.

```c
/* An array of thread identifiers, needed by
   pthread_join() later. */
pthread_t tid[NUM_OF_THREADS];

/* An array to hold argument data to the hello()
   start routine for each thread. */
int arg[NUM_OF_THREADS];

/* Attributes (stack size, scheduling information
   etc) for the new threads. */
pthread_attr_t attr;

/* Get default attributes for the threads. */
pthread_attr_init(&attr);
```

Declaration of arrays used to store thread IDs and arguments for each threads start routine, the `hello()` function.

Use default attributes when creating new threads.

```c
/* Create new threads, each executing the
    hello() function. */
for (int i = 0; i < NUM_OF_THREADS; i++) {
   arg[i] = i;
   pthread_create(&tid[i], &attr, hello, &arg[i]);
}
```

1) Pass in a pointer to `tid_t`. On success `tid[i]` will hold the thread ID of thread number `i`.

2) Pass a pointer to the default attributes.

3) The start routine (a function pointer).

4) A pointer to the argument for the start routine for thread number `i`.

```c
/* Wait for all threads to terminate. */
for (int i = 0; i < NUM_OF_THREADS; i++){
   if (pthread_join(tid[i], NULL) != 0) {
     perror("pthread_join");
     exit(EXIT_FAILURE);
   }
   printf("main() - thread %d terminated\n", i);
}

printf("main() - all new threads terminated\n");
```

1
2

Ex_2_pthread2.c

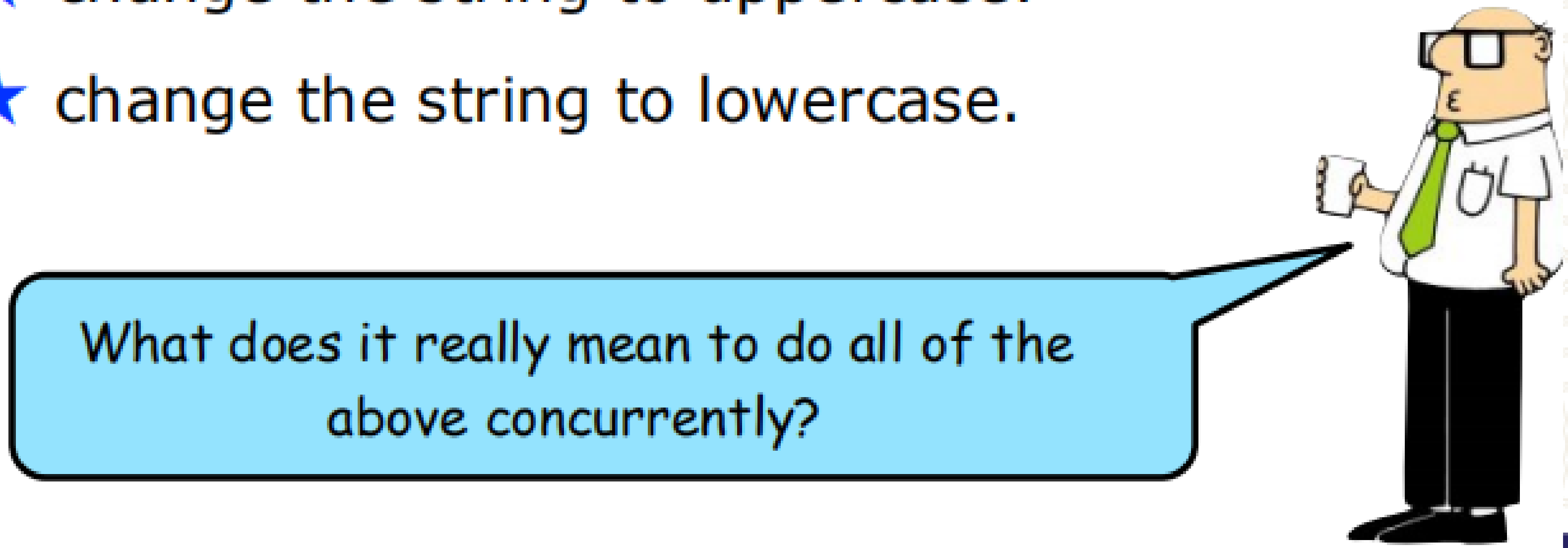
1) Wait for thread with thread ID `tid[i]` to terminate.

2) Pass `NULL` here means we don't care about the exit status of the terminated thread.

pthreads_unsynchronized_concurrency.c

Given a string, write a program using Pthreads to concurrently:

- ★ calculate the length of the string.
- ★ calculate the number of spaces in the string.
- ★ change the string to uppercase.
- ★ change the string to lowercase.



Header files and global data Start routines (1)

```c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h> // sleep()

#define NUM_OF_THREADS 4

/* A global string for the threads to work on. */
char STRING[] = "The string shared among the threads.";

/* Global storage for results. */
int LENGTH;
int NUM_OF_SPACES;
```

```c
void* length(void *arg) {
  char *ptr = (char*) arg;
  int i = 0;
  while (ptr[i]) i++;
  LENGTH = i;
}

void* num_of_spaces(void *arg) {
  char *ptr = (char*) arg;
  int i = 0;
  int n = 0;

  while (ptr[i]) {
    if (ptr[i] == ' ') n++;
    i++;
  }
  NUM_OF_SPACES = n;
}
```

The implementation details of these functions are not important for the purpose of this exercise.

But, note that to for Pthreads to be able to use these functions as start routines for the threads, they must all be declared **void*** and take a single argument of type **void***.

main() - step 1

```
int main(int argc, char *argv[]) {
  /* An array of thread identifiers, needed by pthread_join() later... */
  pthread_t tid[NUM_OF_THREADS];

  /* Attributes (stack size, sche
  pthread_attr_t attr;

  /* Get default attributes for the thre    . */
  pthread_attr_init(&attr);

pthread_create(&tid[i], &attr, length, STRING);
```

We could simply call **pthread_create()** four times using the four different string functions:

- ★ `length()`
- ★ `num_of_spaces()`
- ★ `to_upppercase()`
- ★ `to_lowercase()`

, for example like this.

But, it is more practical (and fun) to collect pointers to all the functions in an array.

main() - step 2

```
int main(int argc, char *argv[]) {
  /* An array of thread identifiers, needed by pthread_join() later... */
  pthread_t tid[NUM_OF_THREADS];

  /* An array of pointers to the callback functions. */
  void* (*callback[NUM_OF_THREADS]) (void* arg) =
    {length,
     to_uppercase,
     to_lowercase,
     num_of_spaces};

  /* Attributes (stack size, scheduling information) for the threads. */
  pthread_attr_t attr;

  /* Get default attributes for the threads. */
  pthread_attr_init(&attr);

  /* Create one thread running each of the callbacks. */
  for (int i = 0; i < NUM_OF_THREADS; i++) {
    pthread_create(&tid[i], &attr, *callback[i], STRING);
  }

  /* Wait for all threads to terminate. */
  for (int i = 0; i < NUM_OF_THREADS; i++){
    pthread_join(tid[i], NULL);
  }

  /* Print results. */
  printf("        lenght(\"%s\") = %d\n", STRING, LENGTH);
  printf("num_of_spaces(\"%s\") = %d\n", STRING, NUM_OF_SPACES);
}
```

Test runs

```
Terminal — a.out — 74×17
karl ~/Documents/Teaching/OS/2011/lab1/tutorial: gcc -std=c99 pthreads.c
karl ~/Documents/Teaching/OS/2011/lab1/tutorial: ./a.out
      lenght("tHE STRING SHared among the threads.") = 36
num_of_spaces("tHE STRING SHared among the threads.") = 5
karl ~/Documents/Teaching/OS/2011/lab1/tutorial: ./a.out
      lenght("THE STRING SHARED AMONG THE THREADS.") = 36
num_of_spaces("THE STRING SHARED AMONG THE THREADS.") = 5
karl ~/Documents/Teaching/OS/2011/lab1/tutorial: ./a.out
      lenght("THE STRING SHARED among the threads.") = 36
num_of_spaces("THE STRING SHARED among the threads.") = 5
karl ~/Documents/Teaching/OS/2011/lab1/tutorial: ./a.out
      lenght("THE STRING SHARED AMONG THE THREADS.") = 36
num_of_spaces("THE STRING SHARED AMONG THE THREADS.") = 5
karl ~/Documents/Teaching/OS/2011/lab1/tutorial: ./a.out
      lenght("tHe string shared amOng the threads.") = 36
num_of_spaces("tHe string shared amOng the threads.") = 5
karl ~/Documents/Teaching/OS/2011/lab1/tutorial: 
```

Because the threads execute and operate on the same data concurrently, the result of **to_uppercase()** and **to_lowercase()** will be unpredictable due to **data races**.

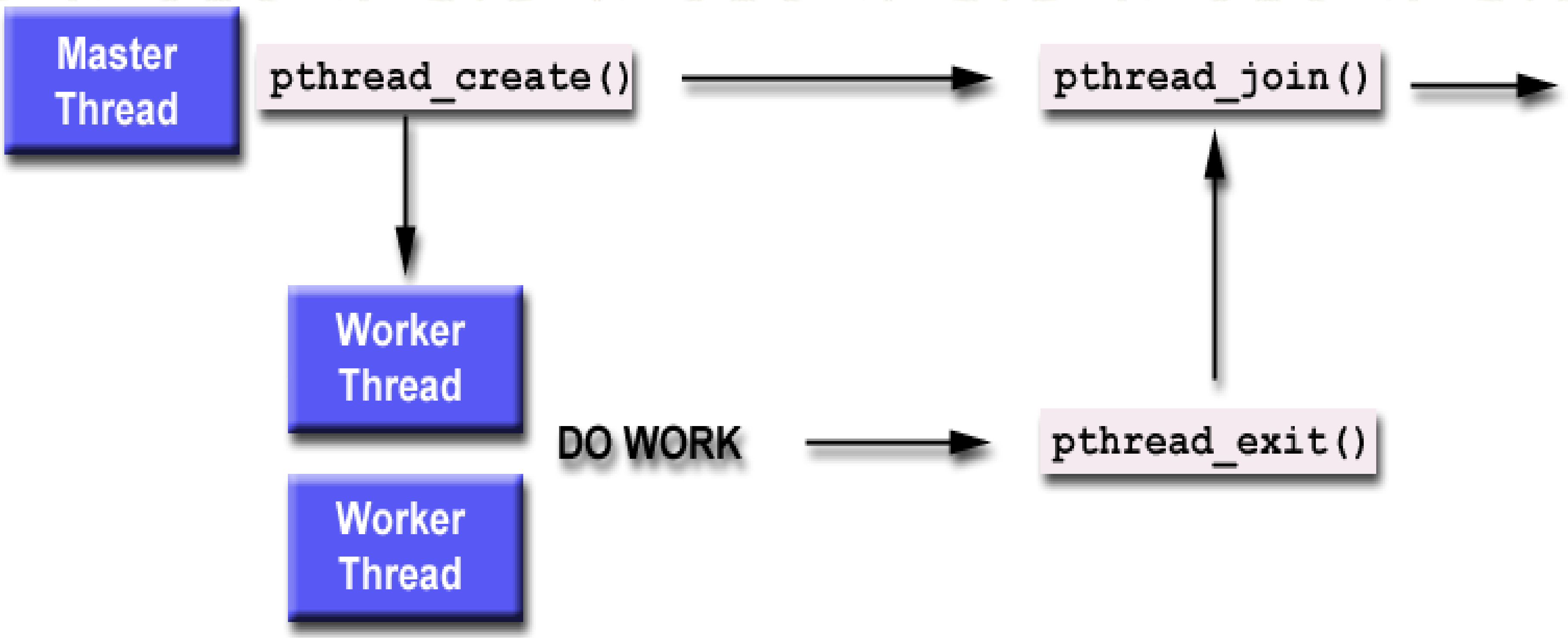
Ex_3_pthread3.c

Detached vs. Joinable

Ex_4_pthread4.c

- Each thread can be either **joinable** or **detached**.
- **Joinable:** on thread termination the thread ID and exit status are saved by the OS.
- **Detached:** on termination all thread resources are released by the OS. A detached thread cannot be joined.

Detached vs. Joinable (Contd.)



Howto detach

Ex_5_pthread5.c

```c
#include <pthread.h>

pthread_t      tid;  // thread ID
pthread_attr_t attr; // thread attribute

// set thread detachstate attribute to DETACHED
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

// create the thread
pthread_create(&tid, &attr, start_routine, arg);
...
```

Shared Global Variables

- Possible problems
  - Global variables
- Avoiding problems
- Synchronization Methods
  - Mutexes
  - Condition variables

Possible problems

- 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**!!!

Mutexes

Ex_6_pthread6.c
Ex_7_pthread7.c

- 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!

Lock & Unlock

- 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** 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);**

Ex_8_pthread7.c

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

Semaphores

- A semaphore is a data structure that is shared by several processes.
- Semaphores are most often used to synchronize operations, when multiple processes access a common, non-shareable resource.
- By using semaphores, we attempt to avoid other multi-programming problems such as:
  - Starvation
  - Deadlock

POSIX Semaphores

- POSIX semaphores allow processes and threads to synchronize their actions.

- A semaphore is an integer whose value is **never allowed** to fall below **zero**.

- POSIX semaphores come in **two forms**:
  - named semaphores
  - unnamed semaphores.

Named Semaphores

- A named semaphore is identified by a name of the form /somename; that is, a null-terminated string
- Two processes can operate on the same named semaphore by passing **the same name** to sem_open().
- Named semaphore functions
  - sem_open()
  - sem_post()
  - sem_wait(), sem_timedwait(), sem_trywait()
  - sem_close()
  - sem_unlink()

Unnamed Semaphores

- An unnamed semaphore does not have a name.
  - The semaphore is placed in a region of memory that is shared between multiple threads or processes.
- A thread-shared semaphore
  - a global variable.
- A process-shared semaphore
  - must be placed in a shared memory region
    - POSIX or System V shared memory segment

Unnamed Semaphores

- Unnamed semaphore functions
  - **sem_init()**
  - sem_post()
  - sem_wait(), sem_timedwait(), sem_trywait()
  - **sem_destroy()**

A simple semaphore example

```c
//create & initialize semaphore
mutex = sem_open(SEM_NAME,O_CREAT,0644,1);
if(mutex == SEM_FAILED)  {
    perror("unable to create semaphore");
    sem_unlink(SEM_NAME);
    exit(-1);
}

while(i<10)  {

      sem_wait(mutex);
      t = time(&t);
      printf("Process A enters the critical section at %d \n",t);
      t = time(&t);
      printf("Process A leaves the critical section at %d \n",t);
      sem_post(mutex);
      i++;
      sleep(3);
}
sem_close(mutex);
sem_unlink(SEM_NAME);
```

```c
//create & initialize existing semaphore
mutex = sem_open(SEM_NAME,0,0644,0);
if(mutex == SEM_FAILED)  {
    perror("reader:unable to execute semaphore");
    sem_close(mutex);
    exit(-1);
}

while(i<10)  {

      sem_wait(mutex);
      t = time(&t);
      printf("Process B enters the critical section at %d \n",t);
      t = time(&t);
      printf("Process B leaves the critical section at %d \n",t);
      sem_post(mutex);
      i++;
      sleep(2);
}

sem_close(mutex);
```

```
lucid@ubuntu:~$ ./PB
Process B enters the critical section at 1376420556
Process B leaves the critical section at 1376420556
Process B enters the critical section at 1376420558
Process B leaves the critical section at 1376420558
Process B enters the critical section at 1376420560
Process B leaves the critical section at 1376420560
Process B enters the critical section at 1376420562
Process B leaves the critical section at 1376420562
Process B enters the critical section at 1376420564
Process B leaves the critical section at 1376420564
Process B enters the critical section at 1376420566
Process B leaves the critical section at 1376420566
Process B enters the critical section at 1376420568
Process B leaves the critical section at 1376420568
Process B enters the critical section at 1376420570
Process B leaves the critical section at 1376420570
Process B enters the critical section at 1376420572
```

```
lucid@ubuntu: ~
File  Edit  View  Terminal  Help
lucid@ubuntu:~$ ./PA
Process A enters the critical section at 1376420554
Process A leaves the critical section at 1376420554
Process A enters the critical section at 1376420557
Process A leaves the critical section at 1376420557
Process A enters the critical section at 1376420560
Process A leaves the critical section at 1376420560
Process A enters the critical section at 1376420563
Process A leaves the critical section at 1376420563
Process A enters the critical section at 1376420566
```

Ex_3_semA.c

Ex_3_semB.c

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.
- IHOP serves breakfast 24 hours a day!

References

- https://github.com/uu-os-2019/
- Getting Started With POSIX Threads by Tom Wagner & Don Towsley Department of Computer Science University of Massachusetts at Amherst July 19, 1995