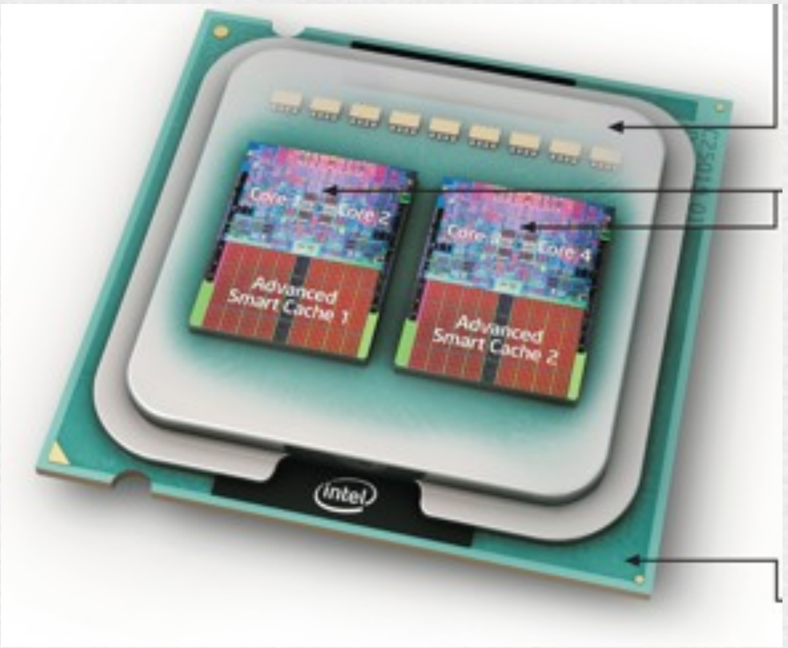
Threaded Programming

Taylan Akdogan
Bogazici University / Istanbul

ISOTDAQ 2011 - Rome

Why?

- Multicore processors are took over, *many*core is coming
- The processor is the “new transistor”
- This is a “sea change” for HW designers and especially for programmers

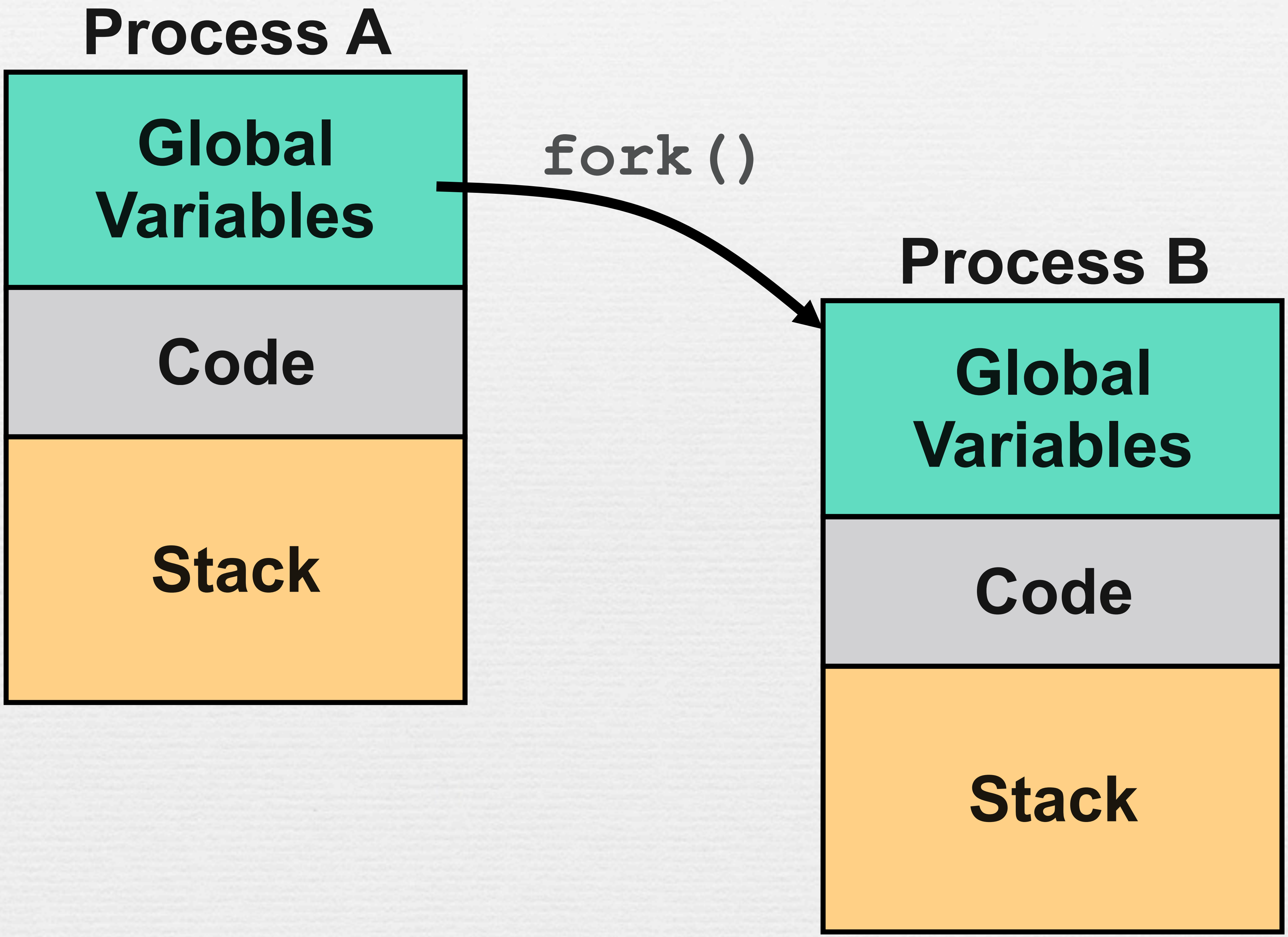


Threads vs Processes

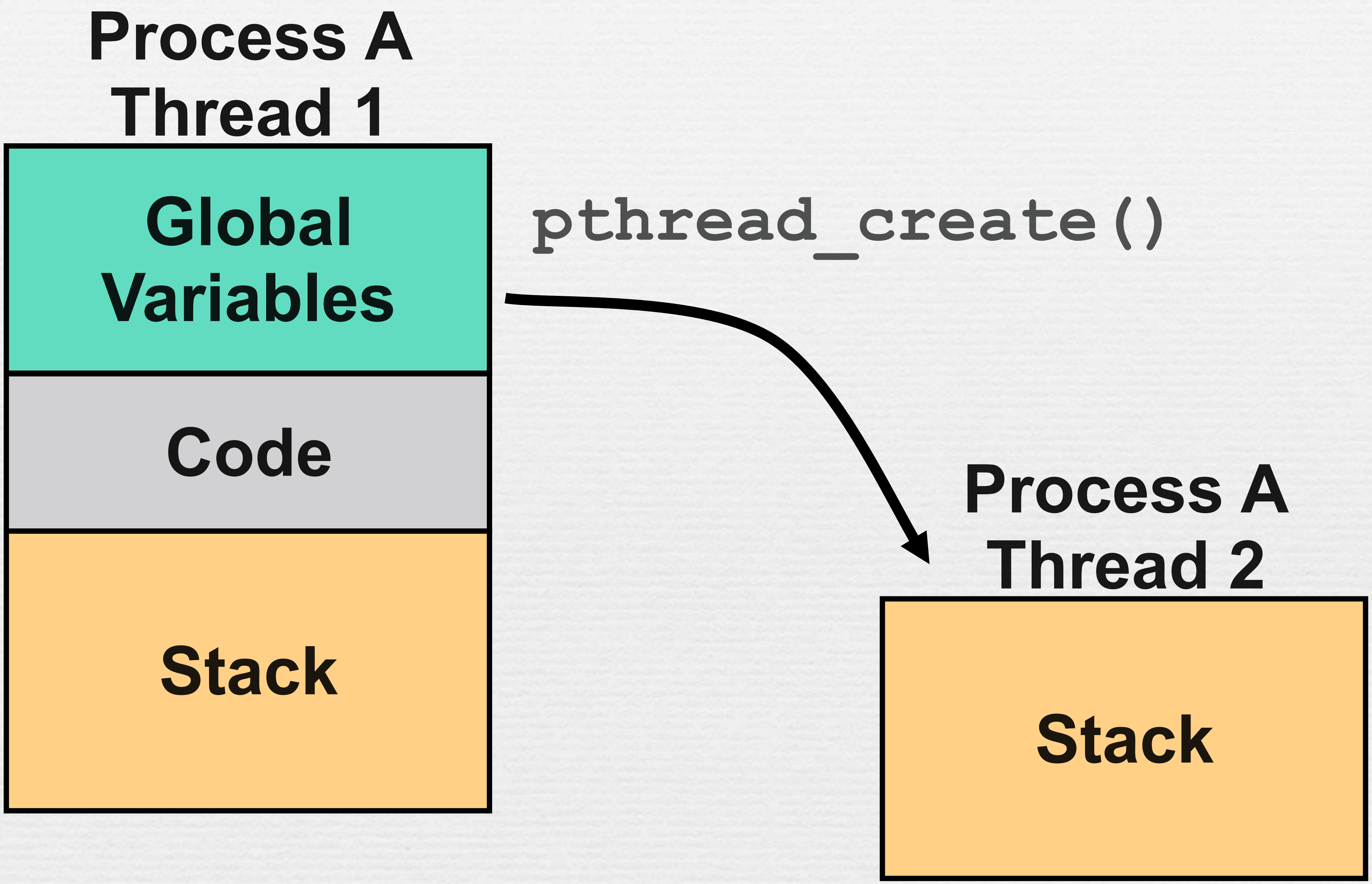
Creation of a new process using fork is *expensive* (time & memory).

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.)

Threads-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.

Unix (Unix/Darwin) - 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_t *tid`

Thread attributes can be set using **`attr`**, including detached state and scheduling policy. You can specify NULL and get the system defaults.

Threads 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());
```

Threads 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 *`**.

Joinable Threads

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);
```

Example 1

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.

Thread Arguments (cont.)

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

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

- Threads have different stacks!
- Use globals or dynamic variables (new/malloc)

Example 2

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 *`**).

Shared Global Variables

Danger / Tehlike / Il pericolo / Gefahr

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!!!!

Danger / Tehlike / Il pericolo / Gefahr

Example 3

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 (which is lockable only once) 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:

Static initilization:

```
pthread_mutex_t counter_mtx=
        PTHREAD_MUTEX_INITIALIZER;
```

Dynamic initialization:

```
pthread_mutexattr_t mattr;
pthread_mutex_t counter_mtx mutex;
pthread_mutexattr_init(&mattr);
pthread_mutex_init(&mutex, &mattr);
```

Note that, by default, mutex is created in unlocked state.

Locking and Unlocking

- To lock (blocking):

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

- To lock (nonblocking):

```
pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- To unlock:

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

Note: **semaphore** of IPC corresponds to a mutex protected variable.

Example 3
(cont.)

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/release 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 cond = PTHREAD_COND_INITIALIZER;

Condition Variables (cont.)

A condition variable is <u>always</u> used with mutex.

```
pthread_cond_wait(pthread_cond_t *cond,
             pthread_mutex_t *mutex);

pthread_cond_signal(pthread_cond_t *cond);
pthread_cond_broadcast(pthread_cond_t *cond)
```

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

IMPORTANT!
Must happen while the mutex lock is held. mutex will be unlocked atomically , at the start of wait.

```
// 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);
}
```

Other Thread Functions

Posix Threads solve almost all the problems you may encounter when you design a multi threaded application for a shared memory system.

It is usually the basis for other interfaces such as the threads of Root. (TThread)

See "`man pthread`" for more information.

Homework

1. Write a multithreaded program that accepts two arguments:
   - The number of threads n
   - A long integer number N.
2. It will calculate π with the following series (Gregory-Leibniz series):

$$\pi \approx 4\sum_{k=0}^{N}\frac{(-1)^k}{2k+1} = 4\left(\frac{1}{1}-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-\frac{1}{9}\cdots+\frac{(-1)^N}{2N+1}\right)$$

   with at least 9 correct significant figures.
3. Time your program with n=1 and n=2,...
4. Make sure that n="number of cores" runs number of cores times faster compared to n=1 case.

Do not attempt to use some other series that does the job faster, like Madhava series (N=27 yields 15 significant figures):

$$\pi = \sqrt{12}\sum_{k=0}^{\infty}\frac{(-3)^{-k}}{2k+1} = \sqrt{12}\left(1-\frac{1}{3\cdot 3}+\frac{1}{5\cdot 3^2}-\frac{1}{7\cdot 3^3}+\dots\right)$$

We want something slower to test the timing

Homework (cont.)

```
long long counter_max; // Upper limit determined by main function
long long counter = 1; // Initial value of the loop
pthread_mutex_t counter_mutex = PTHREAD_MUTEX_INITIALIZER; // for the counter
double *sum_res; // To return the partial sum

void *pi_thread(void *arg) {
  while (1) {
    ....
    pthread_mutex_lock(&counter_mutex);
    begin = counter;
    counter += 1000000;
    pthread_mutex_unlock(&counter_mutex);
    ....
  }
  ....
}

int main(int argn, char *arg[]) {
  ...
  sum_res = (double *)malloc(n*sizeof(double));
  tid = (pthread_t *)malloc(n*sizeof(pthread_t));
  id = (int *)malloc(n*sizeof(int));
  ...
  for (i=0;i<n;i++)
    pthread_create(tid+i, NULL, pi_thread, id+i);
  ...
  // wait the threads to exit

  pi = 0.0;
  for (i=0;i<n;i++)
    pi += sum_res[i];
  ...
```

Homework (cont.)

Seeing is believing...
(demo)

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!
- Make sure that the functions you use are Posix thread-safe, before you use in your threaded application…

Summary

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

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

MPI is another way of making your program multithreaded (and multi-noded)