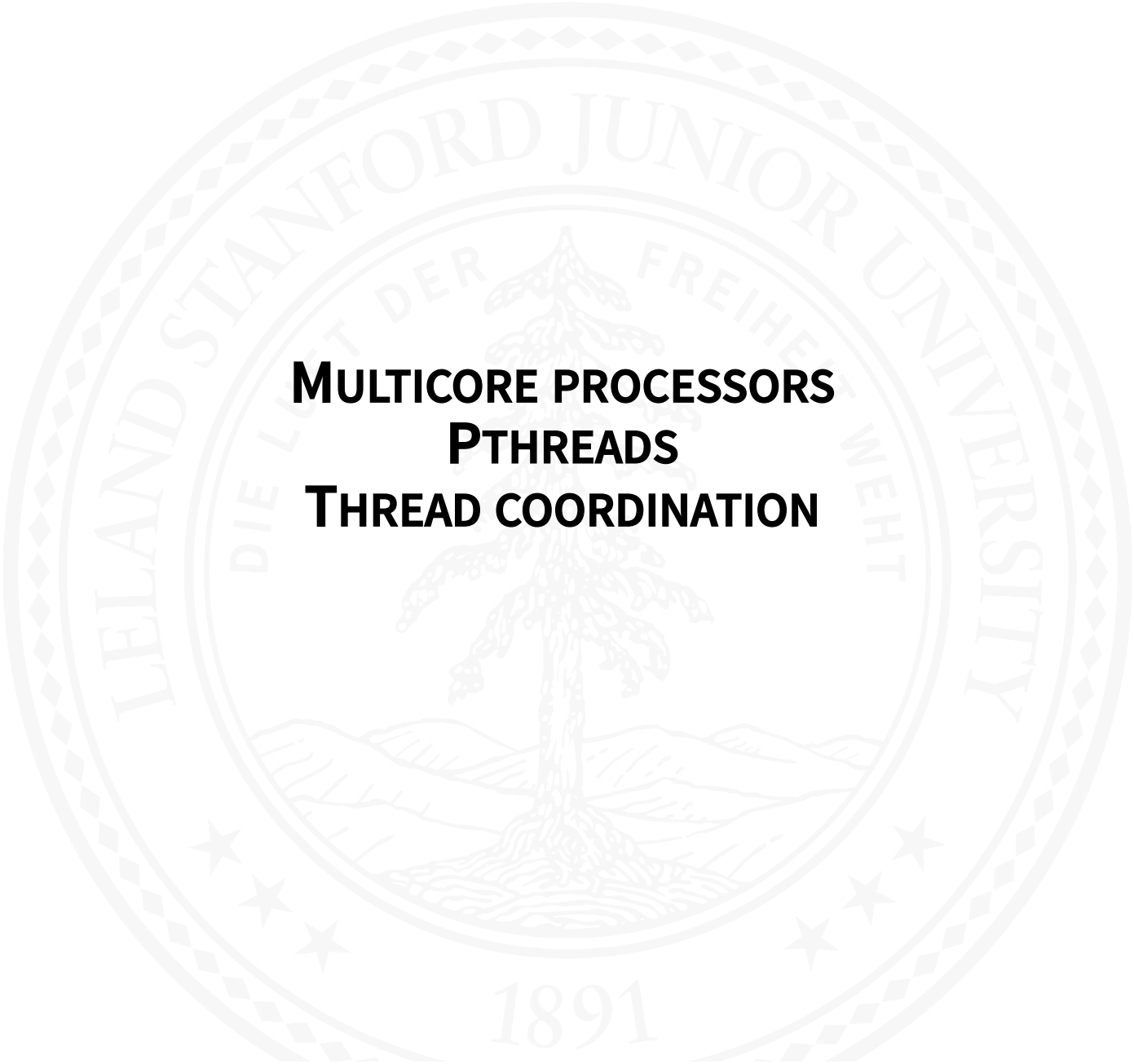
CME 213

SPRING 2012-2013

Eric Darve

Stanford University

Multicore processors
Pthreads
Thread coordination




LELAND STANFORD JUNIOR UNIVERSITY
DIE LUFT DER FREIHEIT WEHT
1891

MUTEXES

MUTEX FUNCTIONS

```
int pthread_mutex init (pthread_mutex t *mutex,
                        const pthread_mutexattr_t *attr)
```

Initialization of mutex; choose **`NULL`** for **`attr`**.

```
int pthread_mutex_destroy (pthread_mutex_t *mutex)
```

Destruction of mutex

```
int pthread_mutex_lock (pthread_mutex_t *mutex)
```

Locks a mutex; blocks if another thread has locked this mutex and owns it.

```
int pthread_mutex_unlock (pthread_mutex_t *mutex)
```

Unlocks mutex; after unlocking, other threads get a chance to lock the mutex.

Deadlock

Another strange case of parallel computing.

Let's assume:

Thread 0 Locks *mutex0*

Thread 1 Locks *mutex1*

Thread 0 Does some work

Thread 1 Does some work

Thread 0 Locks *mutex1*

Thread 1 Locks *mutex0*

The code will never reach this point → Thread 0 Work requiring lock on both mutexes

Thread 1 Work requiring lock on both mutexes

Thread 0 Unlocks all mutexes

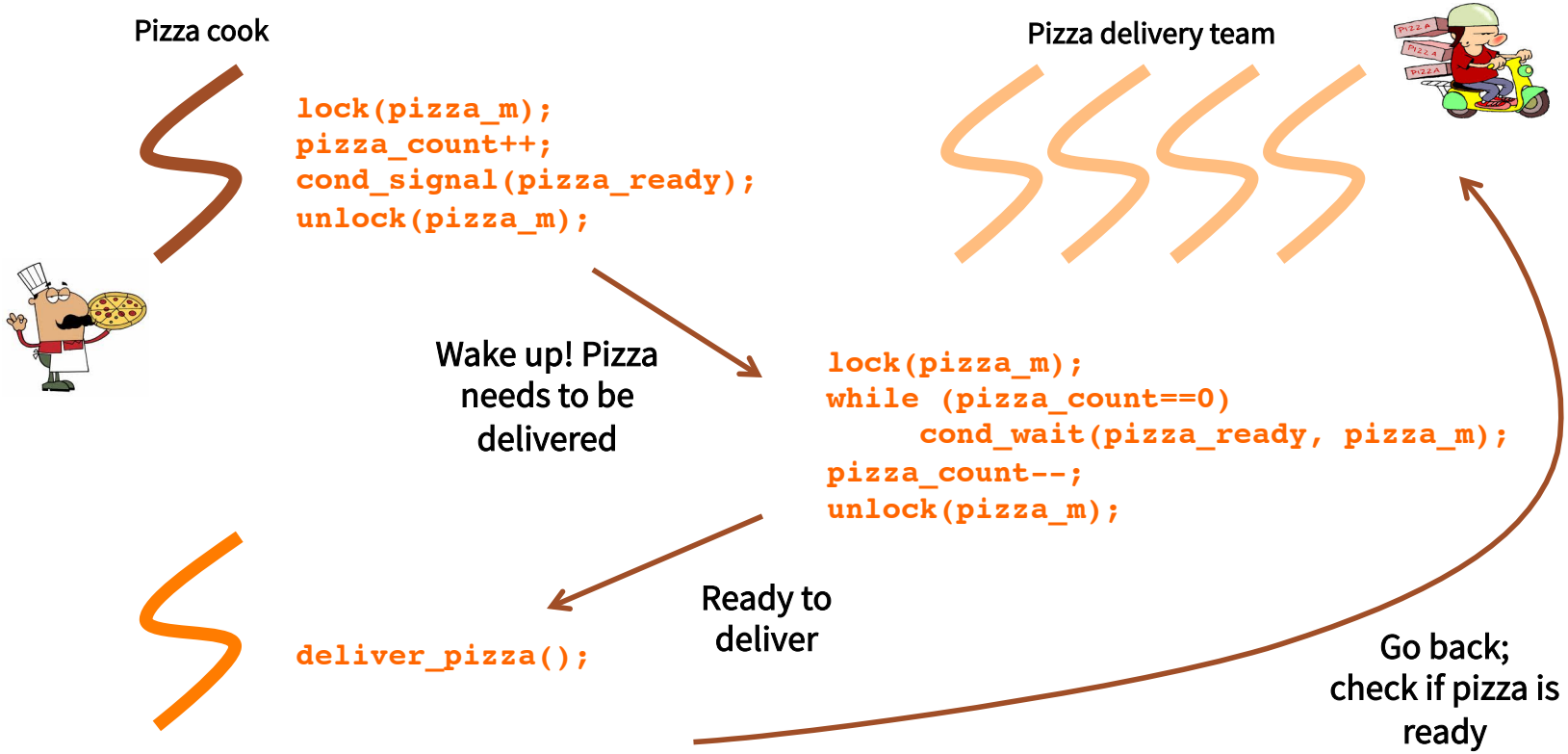
Thread 1 Unlocks all mutexes

Solution: lock mutexes always in the same order.

CONDITION VARIABLES

WAITING ON A CONDITION

- Race conditions are not the only problem.
- Assume we have a producer-consumer algorithm, e.g., a pizza guy makes pizzas and delivery boys deliver pizzas to customers.



CONDITION VARIABLE

- Requires a condition variable and a mutex.
- The mutex is also used to check that no one else is modifying the condition while the thread is checking on it.

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

Recommended scenario (deviate at your own risk)

```
pthread_mutex_lock(&mutex);
while (!condition_ready())
    pthread_cond_wait(&cond, &mutex);
compute_something();
pthread_mutex_unlock(&mutex);
```

When **`pthread_cond_wait`** is called:

- The thread releases the mutex **`mutex`**
- The thread waits until a signal is sent
- Upon receiving a signal, the thread locks **`mutex`** and proceeds

CONDITION SIGNAL

```
int pthread_cond_signal (pthread_cond_t *cond)
```

- Wakes up a single thread waiting on the variable **`cond`**.
- Typically a mutex is used around **`pthread_cond_signal`** to protect the evaluation of the condition.

CONDITION VARIABLE EXAMPLE

- Thread 1 and 2 increment a counter.
- Thread 0 waits until the counter is equal to 12.
- When this happens, thread 0 adds 125 to the counter.
- Thread 1 and 2 then continue incrementing the counter until it reaches 145. The program exits.

When either thread 1 or 2 finds that the counter is equal to 12, it sends a condition signal to thread 0 so thread 0 can add 125.

```
…
#include <pthread.h>

#define NUM_THREADS     3
#define TCOUNT          10
#define COUNT_LIMIT     12

int count = 0;
pthread_mutex_t count_mutex;
pthread_cond_t threshold_condvar;

int main(int argc, char *argv[]) {
    int i, rc;
    long t1=1, t2=2, t3=3;
    pthread_t threads[3];

    /* Initialize mutex and condition variable objects */
    pthread_mutex_init(&count_mutex, NULL);
    pthread_cond_init (&threshold_condvar, NULL);

    pthread_create(&threads[0], NULL, watch_count, (void *)t1);
    pthread_create(&threads[1], NULL, inc_count, (void *)t2);
    pthread_create(&threads[2], NULL, inc_count, (void *)t3);

    /* Wait for all threads to complete */
    for (i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    printf("Main(): waited and joined with %d threads. Final value of count = %d. Done.\n",
           NUM_THREADS, count);

    /* Clean up and exit */
    pthread_mutex_destroy(&count_mutex);
    pthread_cond_destroy(&threshold_condvar);
    pthread_exit (NULL);
}
```

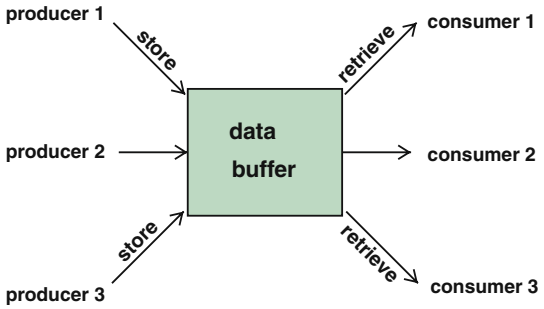
```
void *inc_count(void *t) {
    int i;
    long my_id = (long)t;

    for (i=0; i < TCOUNT; i++) {
        pthread_mutex_lock(&count_mutex);

        count++;

        /* Check the value of count and signal waiting thread when condition is
         reached. */
        if (count == COUNT_LIMIT) {
            printf("inc_count(): thread %ld, count = %d. Threshold reached... ",
                   my_id, count);
            pthread_cond_signal(&threshold_condvar);
            printf("Signal was sent.\n");
        } else
            printf("inc_count(): thread %ld, count = %d.\n",
                   my_id, count);

        pthread_mutex_unlock(&count_mutex);

        /* Do some “work” */
        sleep(1);
    }
    pthread_exit(NULL);
}
```

```
void *watch_count(void *t)
{
    long my_id = (long)t;

    printf("watch_count(): thread %ld. Waiting on condition...\n", my_id);

    pthread_mutex_lock(&count_mutex);
    while (count < COUNT_LIMIT)
        pthread_cond_wait(&threshold_condvar, &count_mutex);

    count += 125;
    printf("watch_count(): thread %ld. Signal received.");
    printf(" Added 125 to count = %d\n", my_id, count);
    pthread_mutex_unlock(&count_mutex);
    pthread_exit(NULL);
}
```

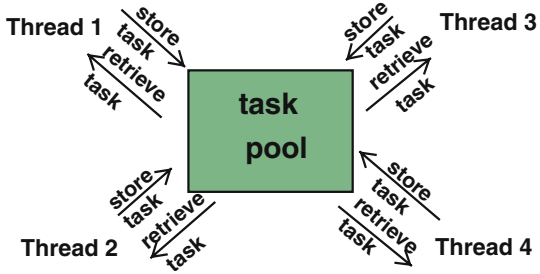
```
$ ./a.out
watch_count(): thread 1. Waiting on condition...
inc_count(): thread 3, count = 1.
inc_count(): thread 2, count = 2.
inc_count(): thread 2, count = 3.
inc_count(): thread 3, count = 4.
inc_count(): thread 2, count = 5.
inc_count(): thread 3, count = 6.
inc_count(): thread 2, count = 7.
inc_count(): thread 3, count = 8.
inc_count(): thread 2, count = 9.
inc_count(): thread 3, count = 10.
inc_count(): thread 2, count = 11.
inc_count(): thread 3, count = 12. Threshold reached... Signal was sent.
watch_count(): thread 1. Signal received. Added 125 to count = 137
inc_count(): thread 2, count = 138.
inc_count(): thread 3, count = 139.
inc_count(): thread 2, count = 140.
inc_count(): thread 3, count = 141.
inc_count(): thread 2, count = 142.
inc_count(): thread 3, count = 143.
inc_count(): thread 2, count = 144.
inc_count(): thread 3, count = 145.
Main(): waited and joined with 3 threads. Final value of count = 145. Done.
```

PARALLEL PATTERNS AND OTHER FEATURES OF PTHREADS

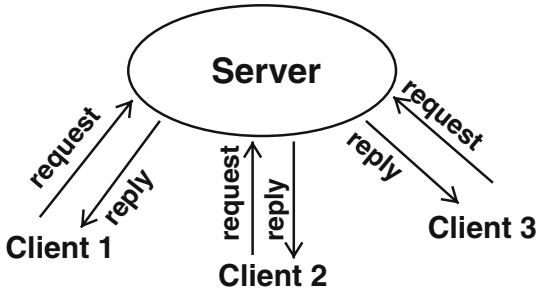
EXAMPLE OF USEFUL PARALLEL PROGRAMMING PATTERNS



Producer consumer

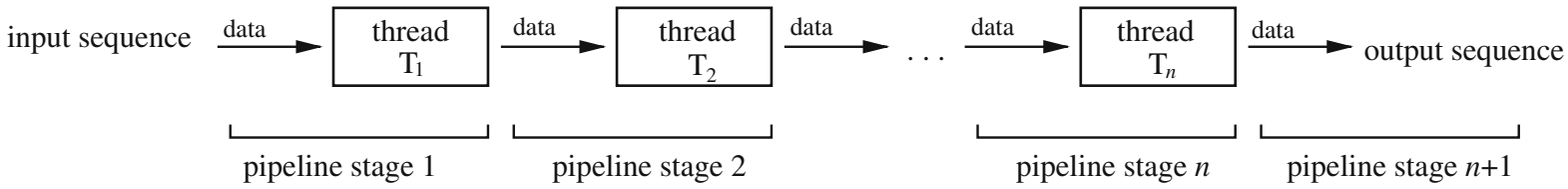


Task pool



Client server

Pipeline



Other features

- Thread Scheduling
  - Implementations will differ on how threads are scheduled to run. In most cases, the default mechanism is adequate.
  - The Pthreads API provides routines to explicitly set thread scheduling policies and priorities which may override the default mechanisms.
- Thread-specific data: keys
  - To preserve stack data between calls, you can pass it as an argument from one routine to the next, or else store the data in a global variable associated with a thread.
  - Pthreads provides another, possibly more convenient and versatile, way of accomplishing this, through keys.
- Priority inversion problems; priority ceiling and priority inheritance
- Thread cancellation
- Barriers; not always available
- Thread safety


LELAND STANFORD JUNIOR UNIVERSITY
DIE LUFT DER FREIHEIT WEHT
1891

OpenMP

OpenMP

- Great when it works
- Complicated when it does not
- OpenMP in some sense tries to simplify the complicated syntax and coding effort of Pthreads. Who wants to continuously cast `void*` ?

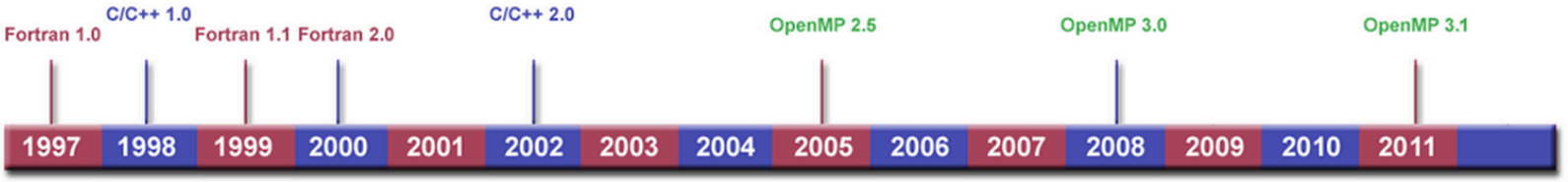
What is OpenMP?

- OpenMP is an Application Programming Interface (API), jointly defined by a group of major computer hardware and software vendors.
- OpenMP provides a portable, scalable model for developers of shared memory parallel applications.
- The API supports C/C++ and Fortran on a wide variety of architectures.

Hence it is more portable in fact than Pthreads.

- OpenMP website: openmp.org
- Wikipedia: en.wikipedia.org/wiki/OpenMP

Timeline



OpenMP 4.0 is around the corner

COMPILING YOUR CODE

- First things first
- Header file:

```
#include <omp.h>
```

- Compiler flags:

Compiler	Flag
icc icpc ifort	-openmp
gcc g++ g77 gfortran	-fopenmp

PARALLEL REGIONS

DIRECTIVES

- OpenMP is based on directives.
- Powerful because of simple syntax.
- Dangerous because you may not understand exactly what the compiler will do.

BASIC EXAMPLE

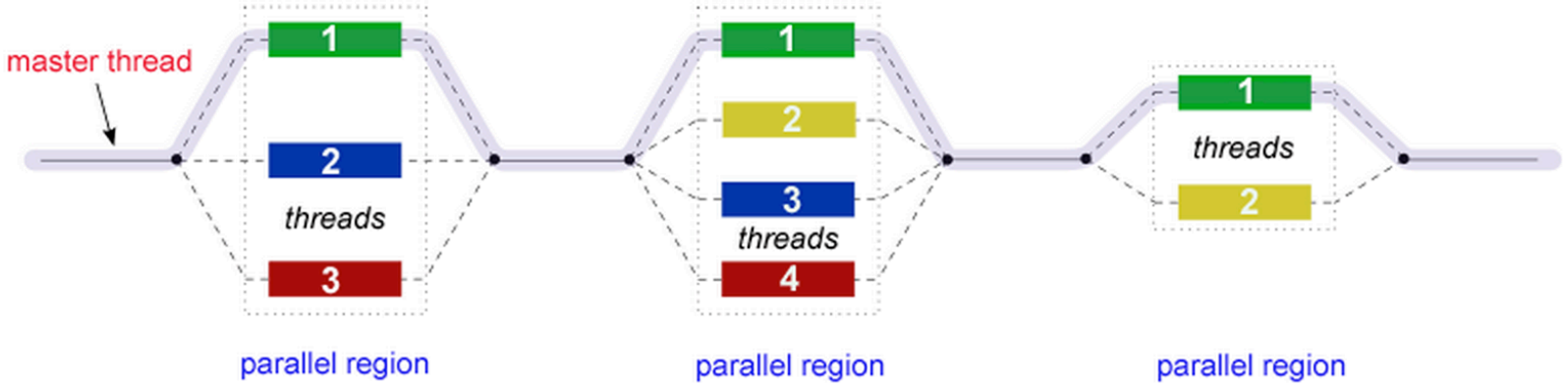
```
#include <omp.h>
#include <stdio.h>

void do_work(int tid) {
    /* busy work */
    return;
}

int main ()  {
      int nthreads, tid;

/* Fork a team of threads with each thread having a private tid variable
     */
#pragma omp parallel private(tid)
      {
            /* Obtain and print thread id */
            tid = omp_get_thread_num();
            printf("Hello World from thread = %d\n", tid);

            /* Only master thread does this */
            if (tid == 0) {
                  nthreads = omp_get_num_threads();
                  printf("Number of threads = %d\n", nthreads);
            }

        /* Pretend to do some work */
        do_work(tid);
      }  /* All threads join master thread and terminate */
      return 0;
}
```

PARALLEL REGION

- The most basic directive is

```
#pragma omp parallel
{ // structured block ... }
```

- This starts a new parallel region. OpenMP follows a fork-join model:



- Upon entering a region, if there are no further directives, a team of threads is created and all threads execute the code in the parallel region.