**Pthreads**

- Low-level API
- Implemented in C
- `pthread_create`
- `pthread_join`
- `pthread_exit`
See

hello_pthread_bug_1.c
hello_pthread_bug_2.c
What's the error in bug_1.c?

e type of result is wrong

The type of p_thread_result is wrong

result is a local variable

result no longer exist after the thread terminates
Line 49
Line 51
Line 52

Start the presentation to activate live content
If you see this message in presentation mode, install the add-in or get help at PollEv.com/app
The result is determined. 

What happens if you delete line 49 in bug_2.c.

If you see this message in presentation mode, install the add-in or get help at PoIiev.com/app.
EXAMPLE: MATRIX MATRIX PRODUCT

- $C = A \times B$
  - where $A$ and $B$ are square matrices.
- See `matrix_prod.c`
Why are we casting work to void* in line 179?

So MatrixMult can be a general function

Because this is the type that pthread_create expects

Because this is required by the signature of MatrixMult.
THREAD COORDINATION
THE RISKS OF MULTI-THREADED PROGRAMMING

• Let us assume that a well-known bank company has asked you to implement a multi-threaded code to perform bank transactions.
• You start with the modest goal of allowing deposits.
• Clients deposit money and the amount gets credited to their accounts.
• As a result of having multiple threads running concurrently the following can happen:
## A Parallel Bank Deposit

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client requests a deposit</td>
<td>Client requests a deposit</td>
<td>$1000</td>
</tr>
<tr>
<td>Check current balance = $1000</td>
<td>Check current balance = $1000</td>
<td></td>
</tr>
<tr>
<td>Ask for deposit amount = $100</td>
<td>Ask for deposit amount = $300</td>
<td></td>
</tr>
<tr>
<td>Compute new balance = $1100</td>
<td>Compute new balance = $1300</td>
<td></td>
</tr>
<tr>
<td>Write new balance to account</td>
<td>Write new balance to account</td>
<td>$1300</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$1100</td>
</tr>
</tbody>
</table>
• Although the correct balance should be $1,400, it is $1,100.
• The problem is that many operations “take time” and can be interrupted by other threads attempting to modify the same data.
• This is called a race condition: the final result depends on the precise order in which the instructions are executed.
**Race condition**

- This issue is addressed using mutexes (mutex): mutual exclusion.
- They ensure that certain common pieces of data are accessed and modified by a single thread.
- This problem typically occurs when you have a sequence like: READ/WRITE, or WRITE/READ performed by different threads.
Thread 0 wants to add new to-do item.

Thread 0 closes lock. No other thread can open the lock.

Thread 0 is done with the to-do list. It opens the lock.

Thread 1 wants to open the lock. It has to wait.

Thread 1 can close the lock and access the to-do list.
**Mutex**

- A mutex can only be in two states: locked or unlocked.
- Once a thread locks a mutex:
  - Other threads attempting to lock the same mutex are blocked
  - Only the thread that initially locked the mutex has the ability to unlock it.
- This allows to protect regions of code.
**Typical usage**

Mutex use:

- Create and initialize a mutex variable.
- Threads attempt to lock the mutex.
- Only one succeeds, and that thread owns the mutex.
- The owner thread performs some set of actions.
- The owner unlocks the mutex.
- Another thread acquires the mutex and repeats the process.
**Pizza Restaurant**

**Pizza Cook**
- Receives all the orders.
- Prepares all the pizzas.
- Ask delivery team to deliver pizzas to customers

**Pizza Delivery Team**
- Checks the addresses for customers
- Delivers pizza

Go back; check if there are orders left.
PIZZA RESTAURANT CODE

See

mutex_demo.c
SUMMARY OF KEY FUNCTIONS

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

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.
Should I move line 76 (mutex_lock) to before line "global_task_list = global_task_list->next")

Yes because the thread is not modifying global_task_list until line 86.

No because of line 79.

It depends on what threads are doing.
It makes no difference to activate live content.

If you see this message in presentation mode, install the add-in or get help at Poliev.com/app.

No this will make the code go slower.

Yes this is safer.
**Deadlock**

Another strange case (but common!) 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`
- 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:** threads always lock mutexes in the same order.
CONDITION VARIABLES
**Are we there yet?**

- There is a common situation in computing: you have a pool of threads that are ready to perform tasks, but there is nothing to do yet. So they have to wait until some work becomes available.
- Instead of constantly checking the to-do list, they can simply wait and be awakened when a certain condition is met.
WAITING ON A CONDITION

• Application: pizza delivery boys waiting on pizza cook.
• This is the producer-consumer model.

Pizza cook

lock(&mutex);
insert new order
cond_signal(&cond);
unlock(&mutex);

Notify delivery team
when new order has
been placed

Wait for
next order

Pizza delivery team

lock(&mutex);
while (delivery == empty)
    cond_wait(&cond, &mutex);
check delivery order
unlock(&mutex);

deliver to correct address

Check for next
delivery
CONDITION VARIABLE EXAMPLE

cond_var.c
Is the mutex unlocked when the thread enters `pthread_cond_wait`?
Happens to the mutex after a thread receives a cold signal?

- It is unlocked
- It is locked
- Nothing

Start the presentation to activate live content
CONDITION VARIABLE

• Requires a condition variable and a mutex.
• The mutex is used to protect the access to the condition variable.

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

Recommended scenario

```c
pthread_mutex_lock(&mutex);
while (!condition_ready())
    pthread_cond_wait(&cond, &mutex);
access_modify_shared_data();
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**

```c
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.
PARALLEL PATTERNS AND OTHER FEATURES OF PTHREADS
The producer–consumer model distinguishes between producer threads and consumer threads. The producer thread can only store data elements into the buffer, if this is not full. Consumer threads retrieve data elements generated into the buffer, if they are not empty.

For the transfer of data from producer threads to consumer threads, producer threads produce data which are used as input by consumer threads. Producer threads produce data which are used as input by consumer threads.

### Producer consumer

- **Producer threads** store data into the buffer.
- **Consumer threads** retrieve data from the buffer, if it is not empty.

### Task pool

- Tasks are stored in a task pool.
- Tasks are retrieved by consumer threads.
- Consumer threads can be threads of a single application or threads of different applications.

### Client server

- **Server** coordinate tasks between clients and tasks.
- **Clients** send requests to the server, which then assigns tasks to the server.
- The server distributes tasks to the tasks pool, which are then retrieved by the tasks.

### Pipeline

- Data flows through stages of a pipeline, where each stage performs a specific operation.
- The output of one stage is the input of the next stage.
- The pipeline model is well-suited for data-intensive applications, where the data can be processed one after another.

### Flexibility

- Tasks can be generated dynamically at any point during execution.
- Tasks can be assigned to different threads or processes.
- The actual task pool data structure could be provided by the specific system.
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