

CS 610: POSIX Threads

Swarnendu Biswas

Semester 2020-2021-I

CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

Advantages of Multithreading

Overlap compute while waiting for I/O

Handle asynchronous events

Allows for implementing priority via threads

Can be advantageous even on uniprocessor systems

Multithreading with C/C++

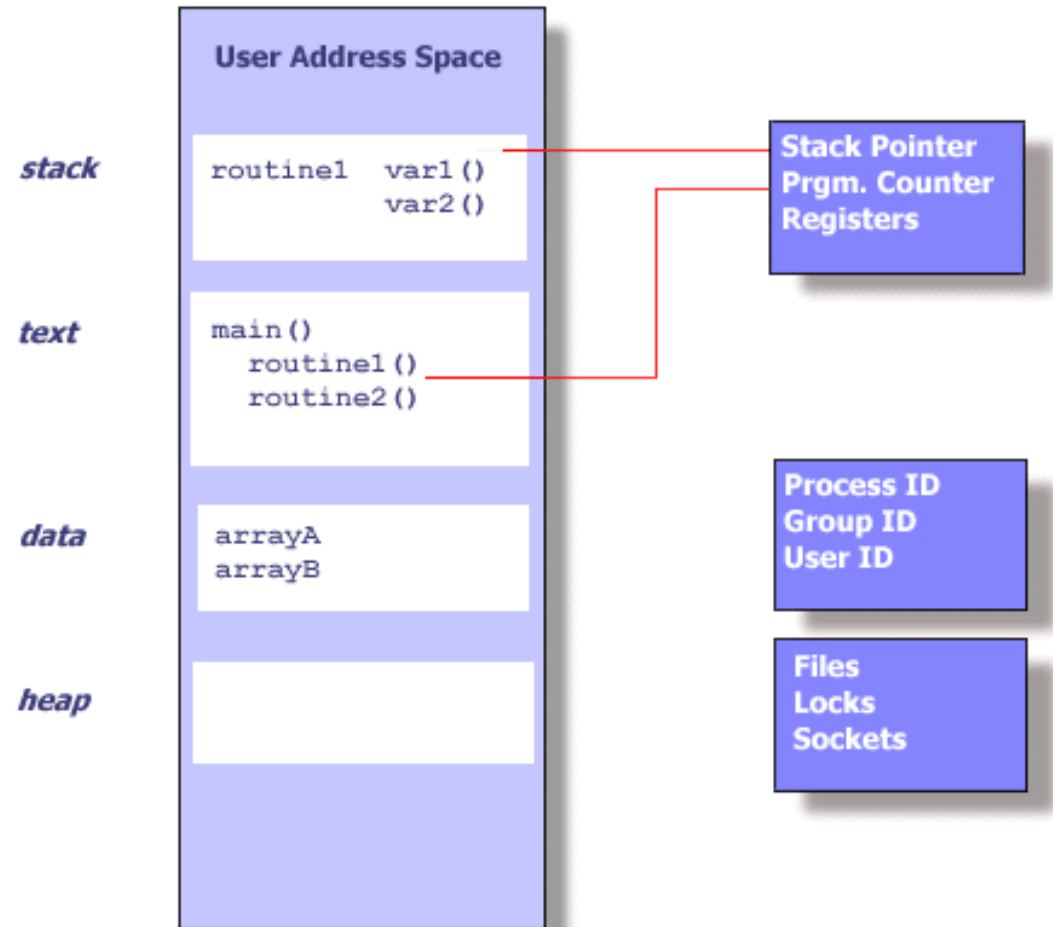
C/C++ languages do not provide built-in support for threads

Several thread libraries have been proposed

- Pthreads – low-level API with fine-grained control
- OpenMP – higher-level abstraction, cross-platform
- Intel TBB – high-level library for task-based programming

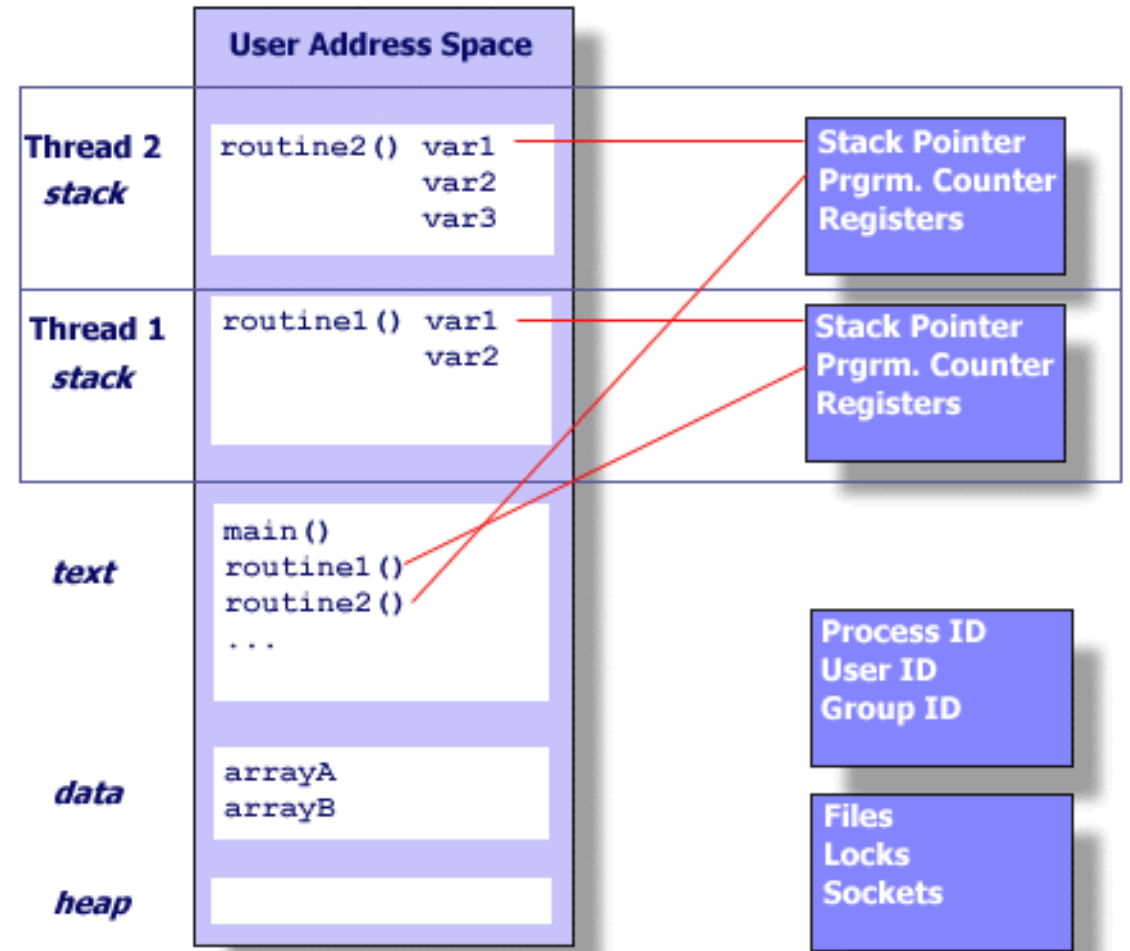
Unix Process

- Process id, process group id, user id, parent id, group id, etc.
- Working directory
- Program instructions
- Registers, stack, heap
- File descriptors
- Shared libraries
- IPC



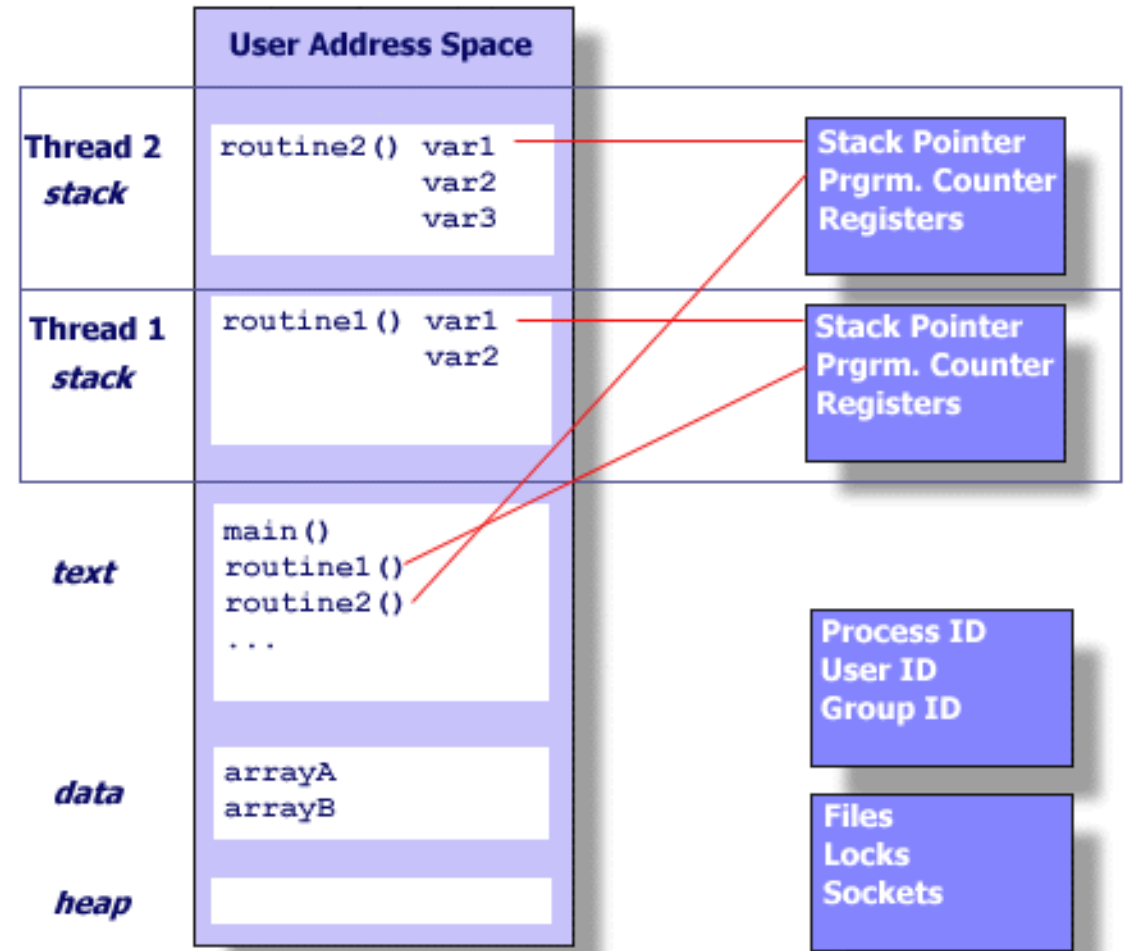
Threads in Unix

- Part of the process and reuses resources
- Software analog of cores
- Maintains its own SP, PC, registers, scheduling properties, any thread-specific data, ...
- All threads share the process heap and the global data structures



Threads in Unix

- Runtime system schedules threads to cores
- If there are more threads than cores, the runtime will time-slice threads on to the cores



POSIX Threads (Pthreads)

- POSIX: Portable Operation System Interface for Unix
 - Standardized programming interface by IEEE POSIX 1003.1c for Unix-like systems
- **Pthreads**: POSIX threading interface
 - Provides system calls to create and manage threads
 - Contains ~100 subroutines

When to use Pthreads?

- Pthreads provide good performance on shared-memory single-node systems
 - Compare with MPI on a single node
 - No need for memory copies, no overhead from data transfer
- Ideal for shared-memory parallel programming
- Heuristic: # threads == # cores

Groups in Pthreads API

Thread management

- Create, detach, join threads

Mutexes

- Support mutual exclusion

Condition variables

- Communicate between threads via mutexes

Synchronization

- Other forms with read/write locks and barriers

Pthread Routines

Call Prefix	Functional Group
pthread_	Thread management
pthread_attr_	Thread attributes objects
pthread_mutex	Mutexes
pthread_mutexattr_	Mutex attribute objects
pthread_cond_	Condition Variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys
pthread_rwlock_	Read/write locks
pthread_barrier_	Synchronization barriers

Compile Pthread Programs

- `-pthread` defines few library macros during preprocessing
- `-lpthread` only links

- GNU GCC

- `gcc/g++ <options> <file_name(s)> -pthread`

- Clang

- `clang/clang++ <options> <file_name(s)> -pthread`

- Intel C/C++ Compiler

- `icc/icpc <options> <file_names(s)> -pthread`

Creating Threads

- Program begins execution with the **main** thread

```
#include <pthread.h>
```

```
int pthread_create(pthread_t* thread_handle,  
                  const pthread_attr_t* attribute,  
                  void* (*thread_function) (void*),  
                  void* arg);
```

Thread Creation Example

```
errcode = pthread_create(&tid, &attribute, &thread_function,  
                        &fun_args);
```

- A pthread with handle “tid” is created
- Thread will execute the code defined in `thread_function` with optional arguments captured in `fun_args`
- `attribute` captures different thread features
 - Default values are used if you pass `NULL`
- `errcode` will be **nonzero** if thread creation fails

Thread Creation Example

```
errcode = pthread_create(&tid, &attribute, &thread_function,  
                        &fun_args);
```

- Q: Now that we have created a thread, when and where will the thread be scheduled to run?
- `attribute` captures different thread features
 - Default values are used if you pass `NULL`
- `errcode` will be **nonzero** if thread creation fails

```

#include <cstdint>
#include <iostream>
#include <pthread.h>

#define NUM_THREADS 1

void *thr_func(void *thread_id) {
    uint32_t id = (intptr_t)thread_id;
    std::cout << "Hello World from
Thread " << id << "\n";
    pthread_exit(NULL);
}

```

```

int main() {
    pthread_t threads[NUM_THREADS];
    int errcode;
    uint32_t id;
    for (id = 0; id < NUM_THREADS; id++) {
        std::cout << "In main: creating thread: " << id <<
"\n";
        errcode =
pthread_create(&threads[id], NULL, thr_func, (void
*)(intptr_t)id);
        if (errcode) {
            std::cout << "ERROR: return code from
pthread_create() is " << errcode
<< "\n";
            exit(-1);
        }
    }

    pthread_exit(NULL);
}

```

Swarnendu Biswas

```
#include <cstdint>
#include <iostream>
#include <pthread.h>
```

```
int main() {
    pthread_t threads[NUM_THREADS];
    int errcode;
    uint32_t id;
```

```
#define NUM_THREADS 1
```

```
void *thread_func(void *arg)
```

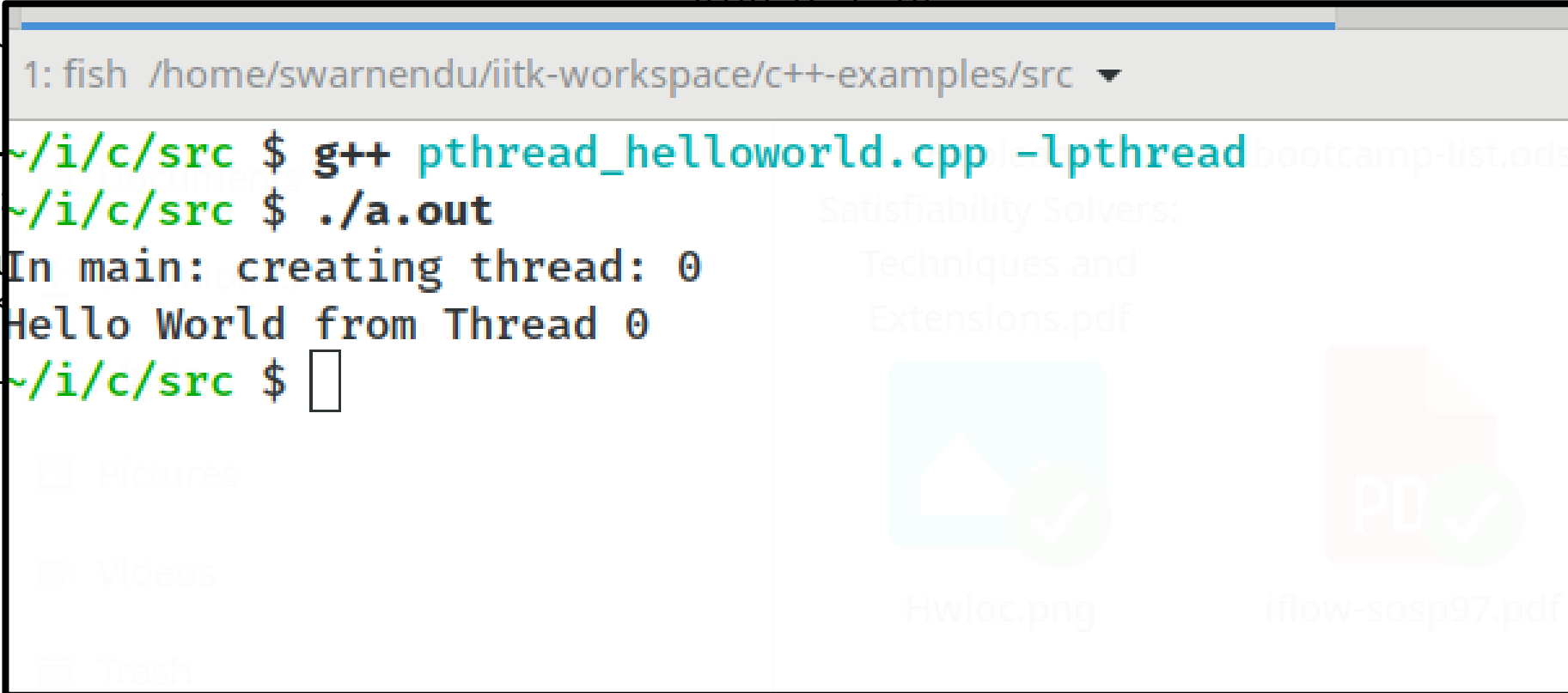
```
uint32_t id;
```

```
std::cout << "Thread " << id << "\n";
```

```
pthread_exit(NULL);
```

```
return NULL;
```

```
}
```



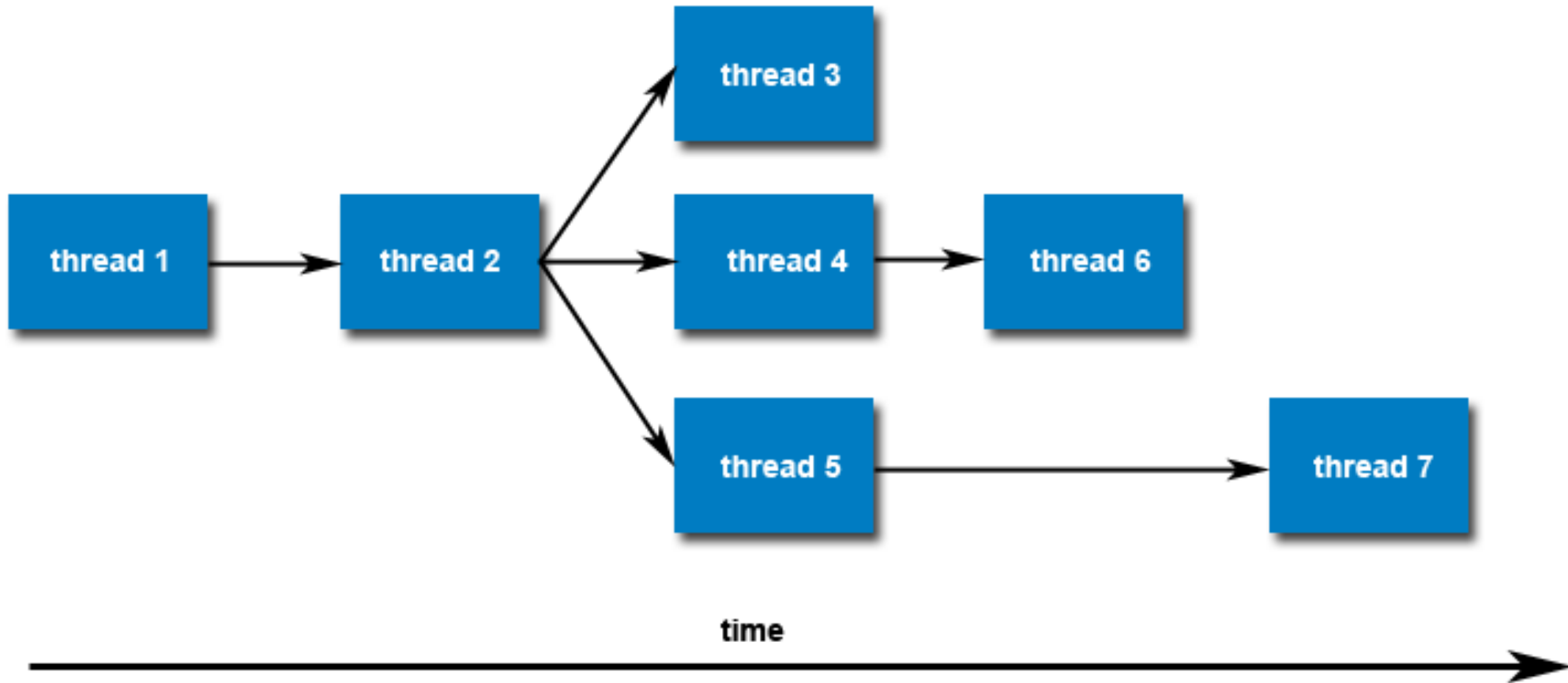
```
}
```

```
pthread_exit(NULL);
```

```
}
```

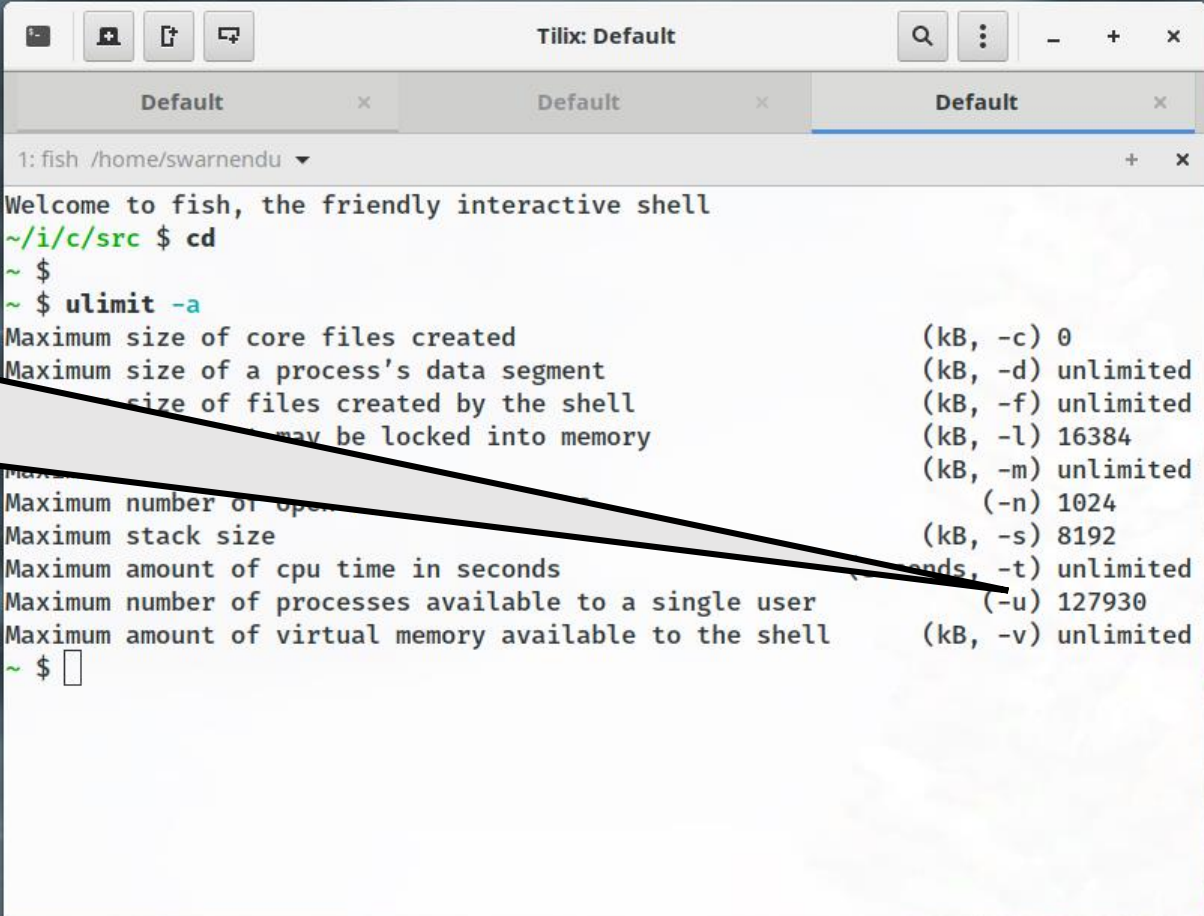
Swarnendu Biswas

No Implied Hierarchy Between Threads



Number of Pthreads

The limit is implementation-dependent, and can be changed.



```
Tilix: Default
Default x Default x Default x
1: fish /home/swarnendu
Welcome to fish, the friendly interactive shell
~/i/c/src $ cd
~ $
~ $ ulimit -a
Maximum size of core files created (kB, -c) 0
Maximum size of a process's data segment (kB, -d) unlimited
Maximum size of files created by the shell (kB, -f) unlimited
Maximum size of files that may be locked into memory (kB, -l) 16384
Maximum size of memory used by child processes (kB, -m) unlimited
Maximum number of open files (-n) 1024
Maximum stack size (kB, -s) 8192
Maximum amount of cpu time in seconds (seconds, -t) unlimited
Maximum number of processes available to a single user (-u) 127930
Maximum amount of virtual memory available to the shell (kB, -v) unlimited
~ $
```

Terminating Threads

- A thread is terminated with

```
void pthread_exit(void* retval);
```

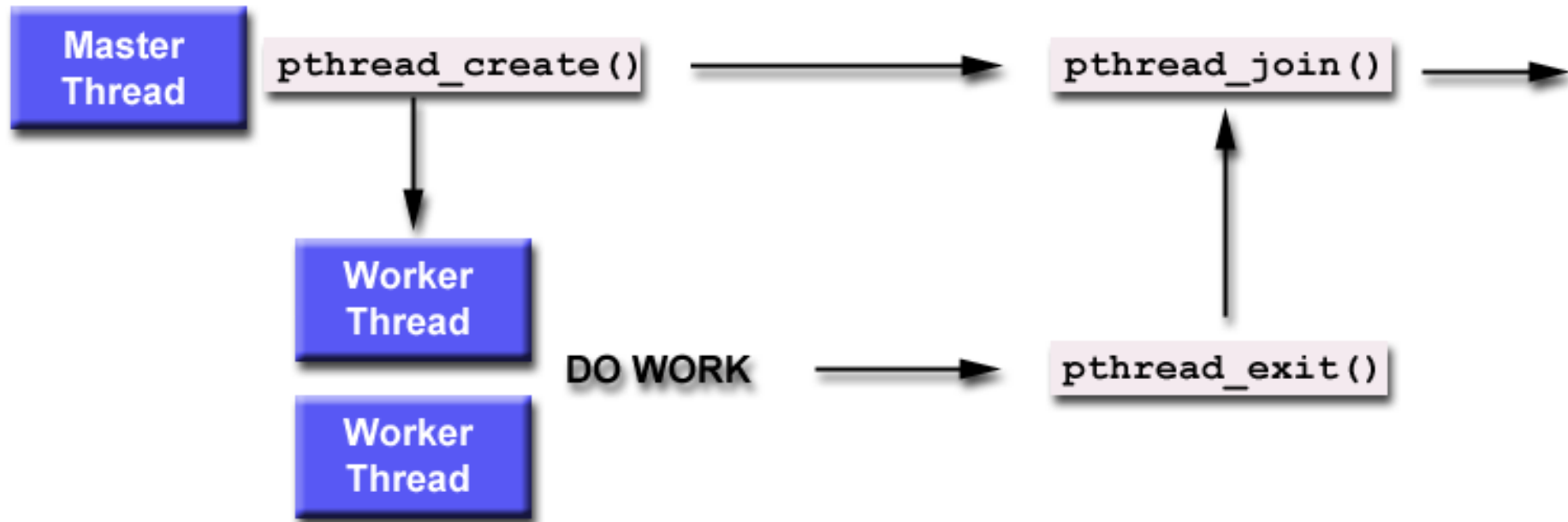
- Process-shared resources (e.g., mutexes, file descriptors) are not released
- Process terminates after the last thread terminates
 - Like calling `exit()`
 - Shared resources are released
- Child threads will continue to run if **called** from main thread

Other Ways to Terminate

- Thread completes executing `thr_func()`
- Thread calls `pthread_exit()`
- Thread is canceled by another thread via `pthread_cancel()`
- Entire process is terminated by `exit()`
- If main thread finishes first without calling `pthread_exit()` explicitly

Joining Threads

```
int pthread_join(pthread_t thread, void ** value_ptr);
```



Subtle Issues to Keep in Mind

- Only threads that are created as “joinable” can be joined
 - If a thread is created as “detached”, it can never be joined
- A joining thread can match one `pthread_join()` call
 - It is a logical error to attempt multiple joins on the same thread
- If a thread requires joining, it is recommended to explicitly mark it as **joinable**
 - Provides portability as not all implementations may create threads as joinable by default

Other Thread Management Routines

```
pthread_t pthread_self(void);
```

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

```

#define NUM_THREADS 10

uint32_t counter;

struct thr_args {
    uint16_t id;
};

void *thrBody(void *arguments) {
    struct thr_args *tmp =
static_cast<struct thr_args
>(arguments);
    for (uint32_t i = 0; i < 1000; i++) {
        counter += 1;
    }
    pthread_exit(NULL);
}

```

```

int main() {
    int i = 0;
    int error;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    struct thr_args args[NUM_THREADS] = {0};

    while (i < NUM_THREADS) {
        args[i].id = i;
        error = pthread_create(&tid[i], &attr, thrBody,
args + i);
        i++;
    }

    pthread_attr_destroy(&attr);
    cout << "Value of counter: " << counter << "\n";

    // Join with child threads
    pthread_exit(NULL);
}

```



```
#define NUM_THREADS 10
```

```
uint32_t counter;
```

```
struct thr_args {
```

```
    uint16_t id;
```

```
};
```

```
void *thrBody(void *argument)
```

```
    struct thr_args *tmp =  
    static_cast<struct thr_ar  
    *>(arguments);
```

```
    for (uint32_t i = 0; i  
         counter += 1;  
    }
```

```
    pthread_exit(NULL);
```

```
}
```

```
~/i/c/src $ g++ pthread_datarace.cpp -lpthread
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 9569
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 9218
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 9636
```

```
NUM_THREADS];
```

```
thr);
```

```
NUM_THREADS] = {0};
```

```
S) {
```

```
    pthread_create(&tid[i], &attr, thrBody,
```

```
&attr);
```

```
    counter: " << counter << "\n";
```

```
reads
```

```
#define NUM_THREADS 10
```

```
uint32_t counter;
```

```
struct thr_args {
```

```
    uint16_t id;
```

```
};
```

```
void *thrBody(void *argument)
```

```
    struct thr_args *tmp =  
    static_cast<struct thr_args*>(argument);
```

```
    for (uint32_t i = 0; i < NUM_THREADS; i++)  
        counter += 1;  
}
```

```
pthread_exit(NULL);
```

```
}
```

```
~/i/c/src $ g++ pthread_datarace.cpp -pthread
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 9569
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 9218
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 9636
```



```
#define NUM_THREADS 10
```

```
uint32_t counter;
```

```
struct thr_args {
```

```
    uint16_t id;
```

```
};
```

```
void *thrBody(void *argument)
```

```
    struct thr_args *tmp
```

```
    static_cast<struct thr_
```

```
    *>(arguments);
```

```
    for (uint32_t i = 0; i
```

```
        counter += 1;
```

```
    pthread_exit(NULL);
```

```
}
```

```
~/i/c/src $ g++ pthread_datarace.cpp -pthread
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 9569
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 9218
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 10000
```

```
~/i/c/src $ ./a.out
```

```
Value of counter: 9636
```

Data race which
results in an atomicity
violation



Mutual Exclusion

Mutual exclusion (locks)

- Synchronize access to a shared data structure
- Cannot prevent bad behavior if other threads do not take or take wrong locks

Checkout `pthread_mutex_ ...`

```
...  
lock l = alloc_init()  
...  
  
Thread i  
acq(l)  
access data  
rel(l)  
...  
  
Thread i+1  
acq(l)  
access data  
rel(l)  
...
```

Creating Mutexes

```
int pthread_mutex_init(pthread_mutex_t *restrict  
mutex, const pthread_mutexattr_t *restrict attr);
```

- Mutex variables must be initialized before use
 - `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
 - `pthread_mutex_init()`

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

Using a Mutex

- 1) Create and initialize a mutex variable
- 2) Several threads attempt to lock the mutex
- 3) Only one thread wins and owns the mutex, other threads possibly block
- 4) Owner thread performs operations in the critical section
- 5) Owner unlocks the mutex
- 6) One other thread acquires ownership of the mutex
- 7) Go to Step (2) if needed
- 8) Destroy the mutex

Locking and Unlocking Mutexes

```
int pthread_mutex_lock(pthread_mutex_t *mutex);  
int pthread_mutex_trylock(pthread_mutex_t *mutex);  
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Locking and Unlocking Mutexes

```
int pthread_mutex_lock(pthread_mutex_t *mutex);  
int pthread_mutex_trylock(pthread_mutex_t *mutex);  
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

What can be
uses of a trylock?

Types of Mutexes

- **NORMAL**

- Attempt to relock a mutex by the same thread will deadlock, no deadlock detection
- Attempt to unlock an unowned or unlocked mutex results in undefined behavior

- **ERRORCHECK**

- Returns error if a thread tries to relock the same mutex
- Attempt to unlock an unowned or unlocked mutex results in an error

- **RECURSIVE**

- Allows the concept of reentrancy by maintaining a lock count
- Attempt to unlock an unowned or unlocked mutex results in an error

- **DEFAULT**

- Wrong use results in undefined behavior

```

#define NUM_THREADS 10

uint32_t counter;
pthread_mutex_t count_mutex;

struct thr_args {
    uint16_t id;
};

void *thrBody(void *arguments) {
    pthread_mutex_lock(&count_mutex);
    for (uint32_t i = 0; i < 1000; i++) {
        counter += 1;
    }
    pthread_mutex_unlock(&count_mutex);
    pthread_exit(NULL);
}

```

```

int main() {
    int i = 0;
    int error;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    struct thr_args args[NUM_THREADS] = {0};

    while (i < NUM_THREADS) {
        args[i].id = i;
        error = pthread_create(&tid[i], &attr, thrBody,
args + i);
        i++;
    }

    pthread_attr_destroy(&attr);
    cout << "Value of counter: " << counter << "\n";
    // Join with child threads
    pthread_exit(NULL);
}

```

```
#define NUM_THREADS 10
```

```
uint32_t counter;
```

```
pthread_mutex_t count_mutex;
```

```
struct thr_args {
```

```
    uint16_t id;
```

```
};
```

```
void *thrBody(void *argument)
```

```
    pthread_mutex_lock(&count_mutex);
```

```
    for (uint32_t i = 0; i < NUM_THREADS; i++)
```

```
        counter += 1;
```

```
    }
```

```
    pthread_mutex_unlock(&count_mutex);
```

```
    pthread_exit(NULL);
```

```
}
```

```
Default x
1: fish /home/swarnendu/iitk-workspace/c++-examples/src
~/i/c/src $ g++ pthread_mutex.cpp -lpthread
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $ ./a.out
Value of counter: 10000
~/i/c/src $
```



```
HREADS];
;
attr);
s[NUM_THREADS] = {0};

ADS) {

reate(&tid[i], &attr, thrBody,

y(&attr);
ounter: " << counter << "\n";

threads
```

Pthread Mutexes vs Synchronized in Java

Pthread Mutex

- Explicit calls to release or unlock the mutex
- Reentrancy is not enabled by default

Synchronized in Java

- Implicit, lock is release once out of scope
- Reentrancy is enabled since a method can be called recursively

POSIX Semaphores in Pthreads

Semaphores

- Generalize locks to allow “n” threads to access
- Useful if you have **> 1** resource units

```
#include <semaphore.h>
```

```
sem_init()
```

```
sem_wait()
```

```
sem_post()
```

```
gcc/g++ <options> <file_name(s)>  
-pthread -lrt
```



Pthreads Barriers

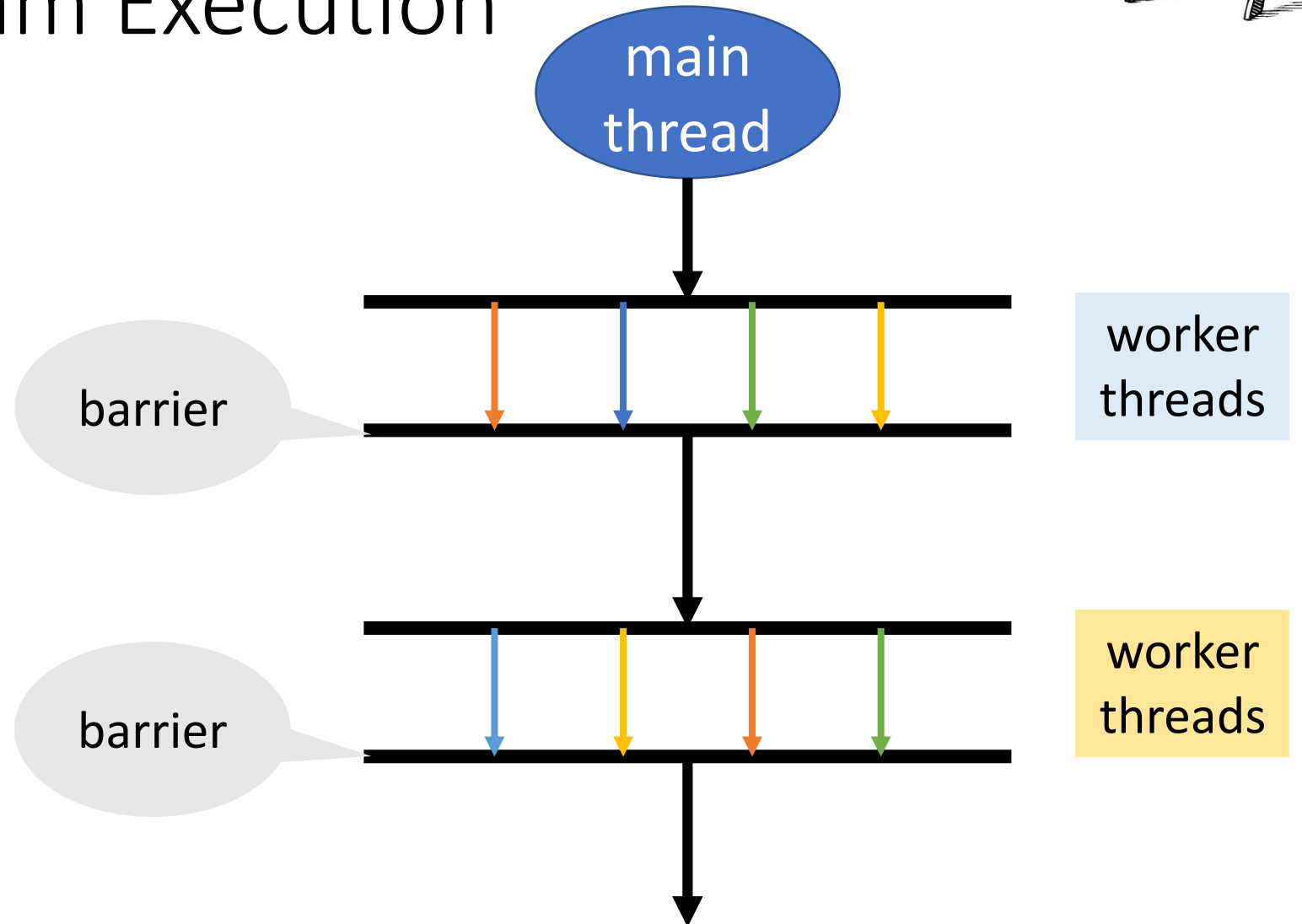
Barrier

- Form of global synchronization
- Commonly used on GPUs, graph analytics

Checkout `pthread_barrier_...`

```
...  
dowork()  
barrier  
...  
  
domorework()  
barrier()  
...
```

Phased Program Execution



Remember this Java Snippet?

```
Object X = null;  
boolean done= false;
```

Thread T1

```
X = new Object();  
done = true;
```

Thread T2

```
while (!done) {}  
X.compute();
```



```
#define NUM_THREADS 2

volatile int i = 0;

void *thr1Body(void *arguments) {
    while (i == 0) {};
    cout << "Value of i has changed\n";
    pthread_exit(NULL);
}
```

```
void *thr2Body(void *arguments) {
    sleep(1000);
    i = 42;
    pthread_exit(NULL);
}
```

```
int main() {
    pthread_t tid1, tid2;

    pthread_create(&tid1, NULL, thr1Body, NULL);
    pthread_create(&tid2, NULL, thr2Body, NULL);

    pthread_exit(NULL);
}
```

```

#define NUM_THREADS 2

volatile int i = 0;

void *thr1Body(void *arguments) {
    while (i == 0) {};
    cout << "Value of i has changed\n";
    pthread_exit(NULL);
}

void *thr2Body(void *arguments) {
    sleep(1000);
    i = 42;
    pthread_exit(NULL);
}

int main() {
    pthread_t tid1, tid2;

    pthread_create(&tid1, NULL, thr1Body, NULL);
    pthread_create(&tid2, NULL, thr2Body, NULL);

    pthread_exit(NULL);
}

```

Busy waiting leads to wasted work

- Often used idiom when we need to synchronize on the **data value**

```
#define NUM_THREADS 2
```

```
volatile int i = 0;
```

```
void *thr1Body(void *arguments) {
```

```
    while (i == 0) {};
```

```
    cout << "Value of i has changed\n";
```

```
    pthread_exit(
```

```
}
```

```
void *thr2Body(void *arguments) {
```

```
    sleep(1000);
```

```
    i = 42;
```

```
    pthread_exit(
```

```
}
```

```
int main() {
```

```
    pthread_t tid1, tid2;
```

```
    pthread_create(&tid1, NULL, thr1Body, NULL);
```

```
    pthread_create(&tid2, NULL, thr2Body, NULL);
```

```
    pthread_exit(NULL);
```

Can you think of situations where busy waiting is actually advantageous?

Can you think of compiler optimizations that can break the code?

Condition Variables

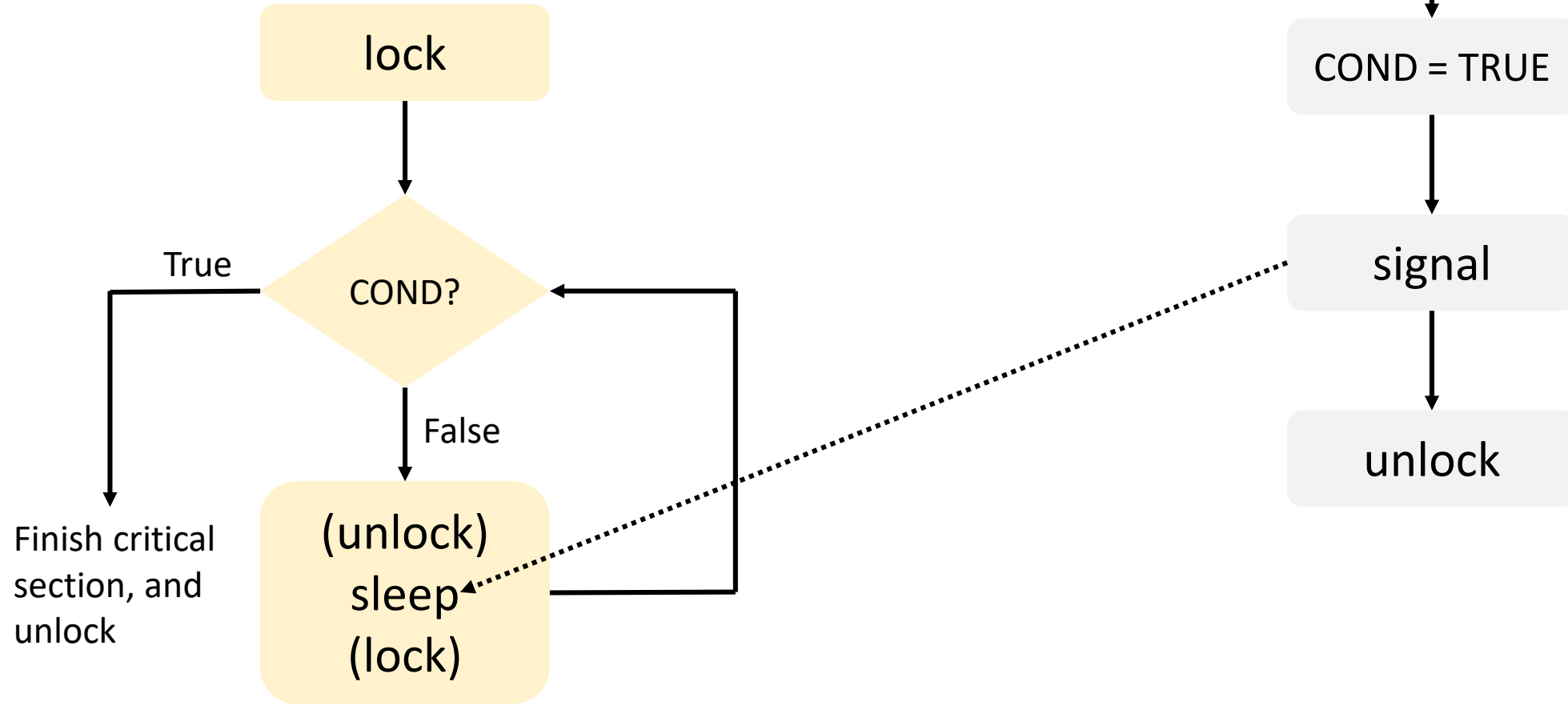
- A condition variable allows a thread to suspend execution until a certain event or condition occurs
- When the event or condition occurs another thread can signal the thread to “wake up”

Signaling mechanism

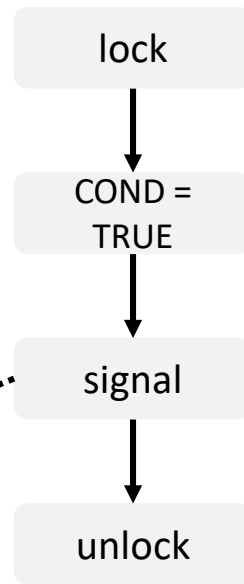
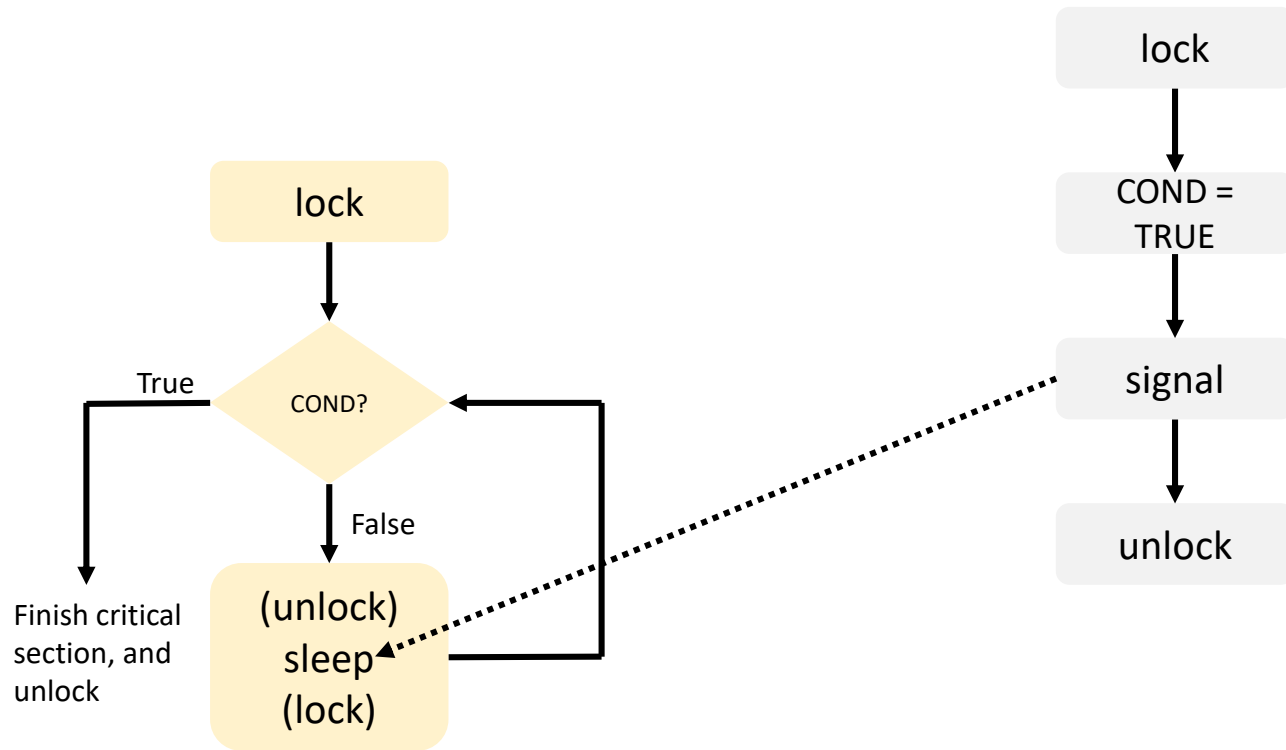
- Always **used along with a mutex lock**

Why?

Condition Variables



Condition Variables



```
lock mutex
if condition has occurred {
    signal thread(s)
} else {
    unlock mutex and block
    // When thread is unblocked,
    the mutex is relocked
}
unlock mutex
```

Using Condition Variables

```
...  
pthread_mutex_lock(&lock);  
while (!COND) {  
    pthread_cond_wait(&cond,  
    &lock);  
}  
// Check COND is true  
  
...  
pthread_mutex_unlock(&lock);  
  
...
```

```
...  
pthread_mutex_lock(&lock);  
...  
// Set COND  
// Wake up one or more threads  
pthread_cond_signal(&cond);  
...  
pthread_mutex_unlock(&lock);  
...
```

Condition Variables

lock

lock mutex

```
if condition has occurred {
```

- When a thread performs a condition wait, it takes itself off the runnable list – it does not use any CPU cycles until it is woken up
- In contrast, a mutex lock consumes CPU cycles as it polls for the lock

k
locked,

True

Finish critical
section, and
unlock

sleep-
(lock)

Lost Wakeup Problem

```
pthread_cond_signal();
```

```
Check condition
```

```
pthread_cond_wait();
```

- Broadcast to all waiting threads, waiting thread should test the condition upon wakeup
- Use timed waits

Condition Variables

Signaling mechanism

- Always **used along with a mutex lock** which protects accesses to shared data

Checkout `pthread_cond_ ...`

Slightly more
involved usage

Ways to Implement a Barrier

Mutex

Condition variables

Semaphores

Nuances of using Pthreads

- **Low-level abstraction**
- Pthreads scheduler may not be well-suited to manage large number of threads
 - Can lead to load imbalance
- OpenMP is commonly used in scientific computing
 - Compiler extensions
 - Higher level of abstraction
- Other abstractions like Transactional Memory

Pitfalls with Multithreading

- Thread scheduling – Do not assume that threads will get executed in the same order as they were created
 - In general, never assume anything about the relative order or speed of execution
- Incorrect synchronization – **Avoid data races**
- Thread safety – Ensure the called library routines are thread safe
- Be careful about other concurrency bugs
 - Deadlocks, atomicity and order violations

References

- James Demmel and Katherine Yelick – CS 267: Shared Memory Programming: Threads and OpenMP
- Keshav Pingali – CS 377P: Programming Shared-memory Machines, UT Austin.
- Blaise Barney, LLNL. POSIX Threads Programming, <https://computing.llnl.gov/tutorials/pthreads>.
- Blaise Barney, LLNL. Introduction to Parallel Computing, https://computing.llnl.gov/tutorials/parallel_comp/
- Peter Pacheco – An Introduction to Parallel Programming.