

Concurrency

Benjamin Brewster

Except as noted, all images copyrighted with Creative Commons licenses,
with attributions given whenever available

Concurrency

- Concurrency (an informal term): doing multiple things at the same time
 - Program decomposition into order-independent chunks
- On UNIX concurrency is easy
 - Multiple processes *can* be running simultaneously
 - Multiple copies of the same program can be running
 - CPU time can be split and shared
- Concurrency is very powerful
 - Greatly increases the efficiency of an OS: while one process is waiting for I/O, another process can use the CPU



Definitions: System Calls vs. Library Functions

- System Calls

- A request for service that causes the normal operation of a process to be *interrupted* and control passed to the OS
- Typically, the process is now blocked and won't do anything else until the system call returns
- `read()`, `write()`, etc.

- C Library Functions

- Faster, as they have no permissions or blocking issues
- `sqrt()`, `printf()`, etc.

Illusions of Simultaneous Execution

- Multiprogramming
 - More than one process can be ready to execute
 - System calls trigger “context switches”, which let the next process run
 - The process will not execute again until its system call returns
- Timesharing
 - CPU time split between multiple processes
 - Gives illusion that many processes are running at once

Processes can also communicate
amongst themselves - more on this later

Multiprocessing

- Executing multiple processes at the actual same time is called **multiprocessing**
- Today's CPUs have 4, 6, 8, and 10 cores, with more coming
 - Each core acts like a mini-CPU, each of which can do **multiprogramming** AND **timesharing**

Possible Complications

- Concurrently running processes can share data and/or resources
- What if multiple processes access the same resource at the same time?
- This is most likely a disaster
 - aka, “Race Condition”, “Oops”, or “Aw carp”



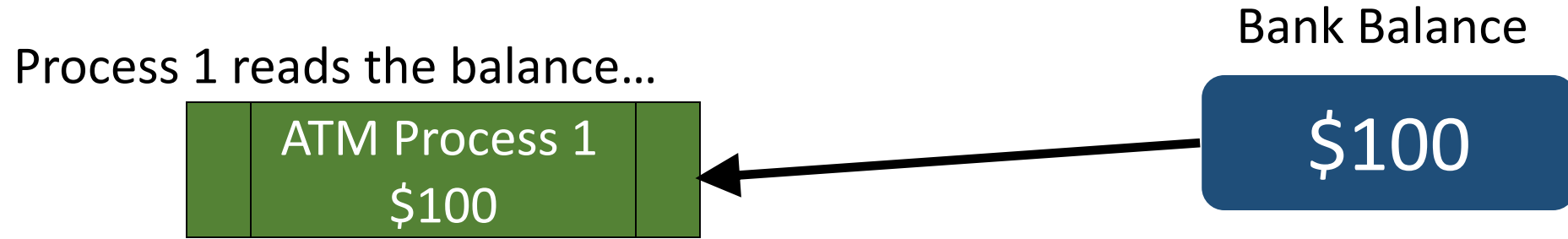
\$#!@7

The Classic Example

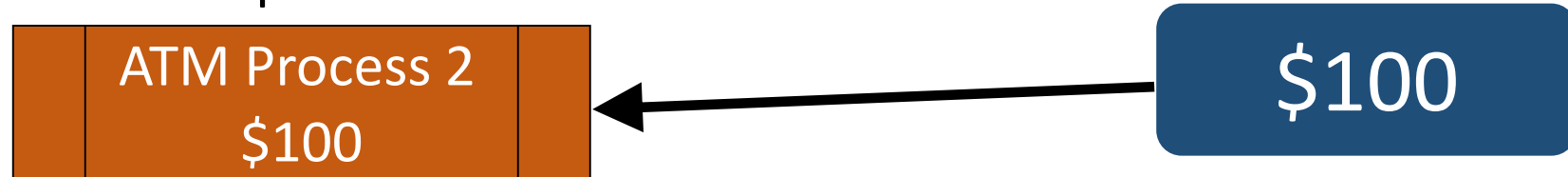
- Two ATM machines each withdraw \$20 from the same account
- To update bank account balance:
 - Read current balance into memory
 - Subtract \$20
 - Write new balance to the bank account



...
Profit!
...
\$#!@7



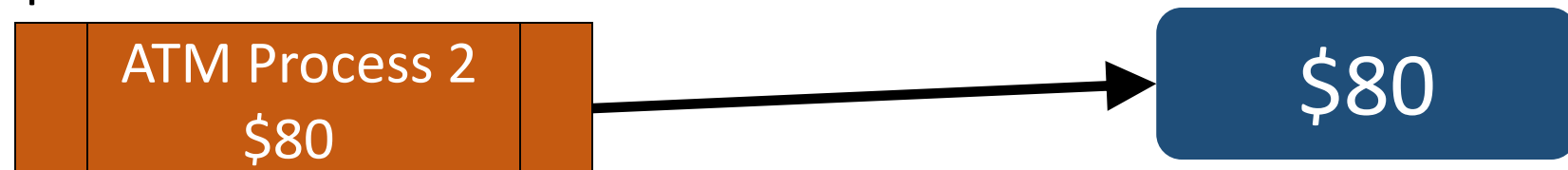
At the same time process 2 reads the balance



So process 1 subtracts and writes...



As does process 2



Race conditions

- Why are race conditions hard to detect?
 - They may only ever show up once, in a particularly strange set of conditions
- Another way of saying it: “A race hazard (or race condition) is a flaw in a system or process where the output exhibits unexpected critical **dependence** on the relative timing of events.”

-Wikipedia



The Seriousness of Software Engineering

- Most infamous race condition:
 - http://en.wikipedia.org/wiki/Therac_25
- People could outrun the system
 - The system was counting on a normally slow human, and didn't take into account people learning to use the system faster
- 3 people died, and 3 were injured as a result of this software engineering disaster



<http://hackaday.com/2015/10/26/killed-by-a-machine-the-therac-25>

The Lesson - Provide Access Control

- Concurrent update situation
 - 2+ processes accessing resource concurrently
 - At least one process might write
- **Must provide access control**
 - If one process is writing, no other process should access (read OR write) the resource
- "Locks" solve these problems
 - Only process owning the lock may access (r/w) the resource
 - Many ways to do locking
 - Locking usually requires support from the OS
 - But you can do it in software, too



Access Control with a Lock File

```
do
{
    lock_fd = open(lock_file_path, O_WRONLY | O_CREAT | O_EXCL, 0644);

    if (lock_fd == -1)
    {
        if (errno == EEXIST)
        {
            // File already exists - wait a while, then try again
            sleep(1);
        }
        else
        {
            // An unexpected error - bail out
            perror("Couldn't open lock file\n");
            return(-1);
        }
    }
}
while (lock_fd == -1);
```



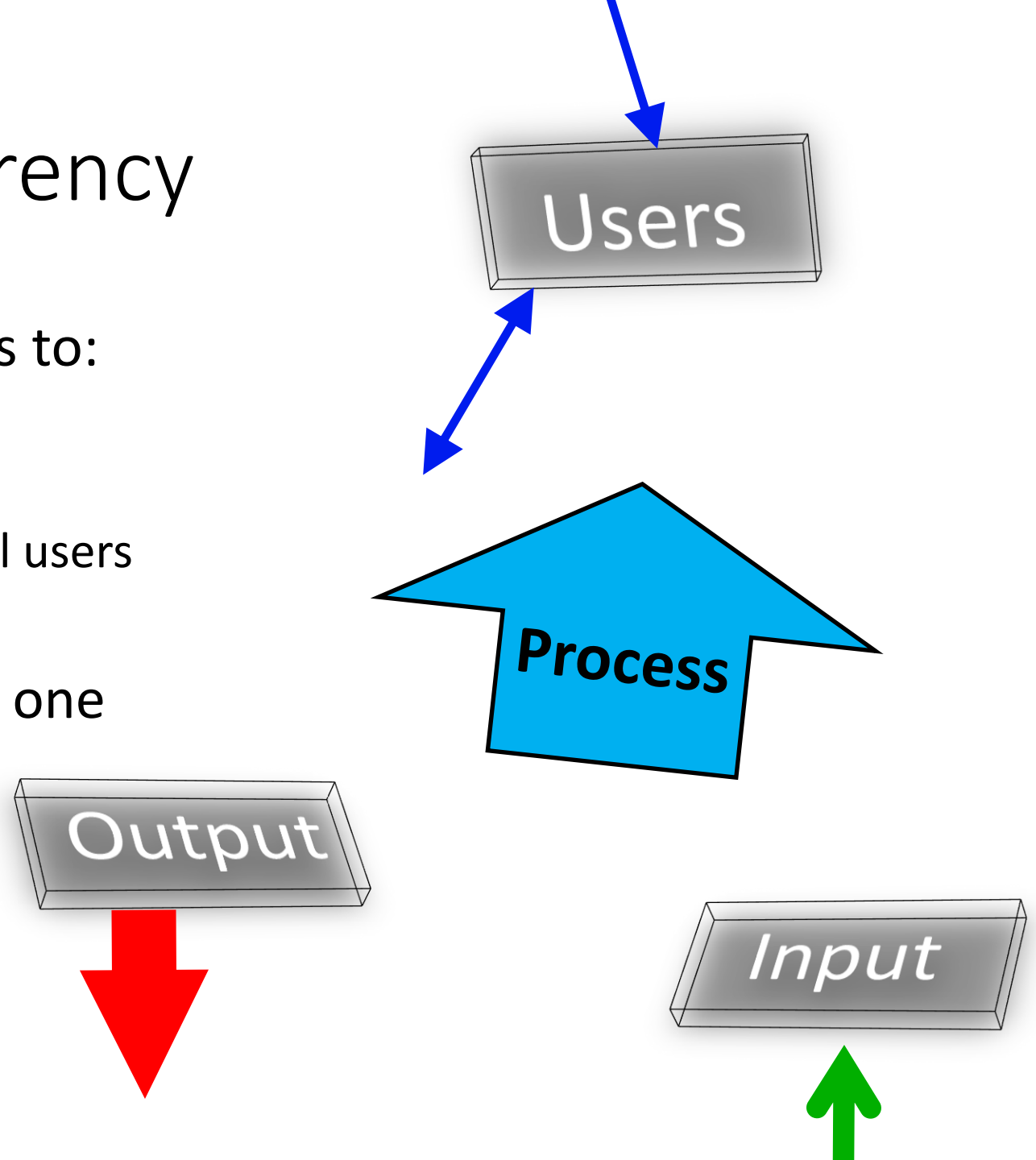
Deeper Definitions

- Kernel
 - The central part of an Operating System
 - Manages hardware (and drivers)
 - Provides the Scheduler
 - Not interacted with by users: system calls are requests to the kernel
- Scheduler
 - Distributes prioritized and/or fair CPU time



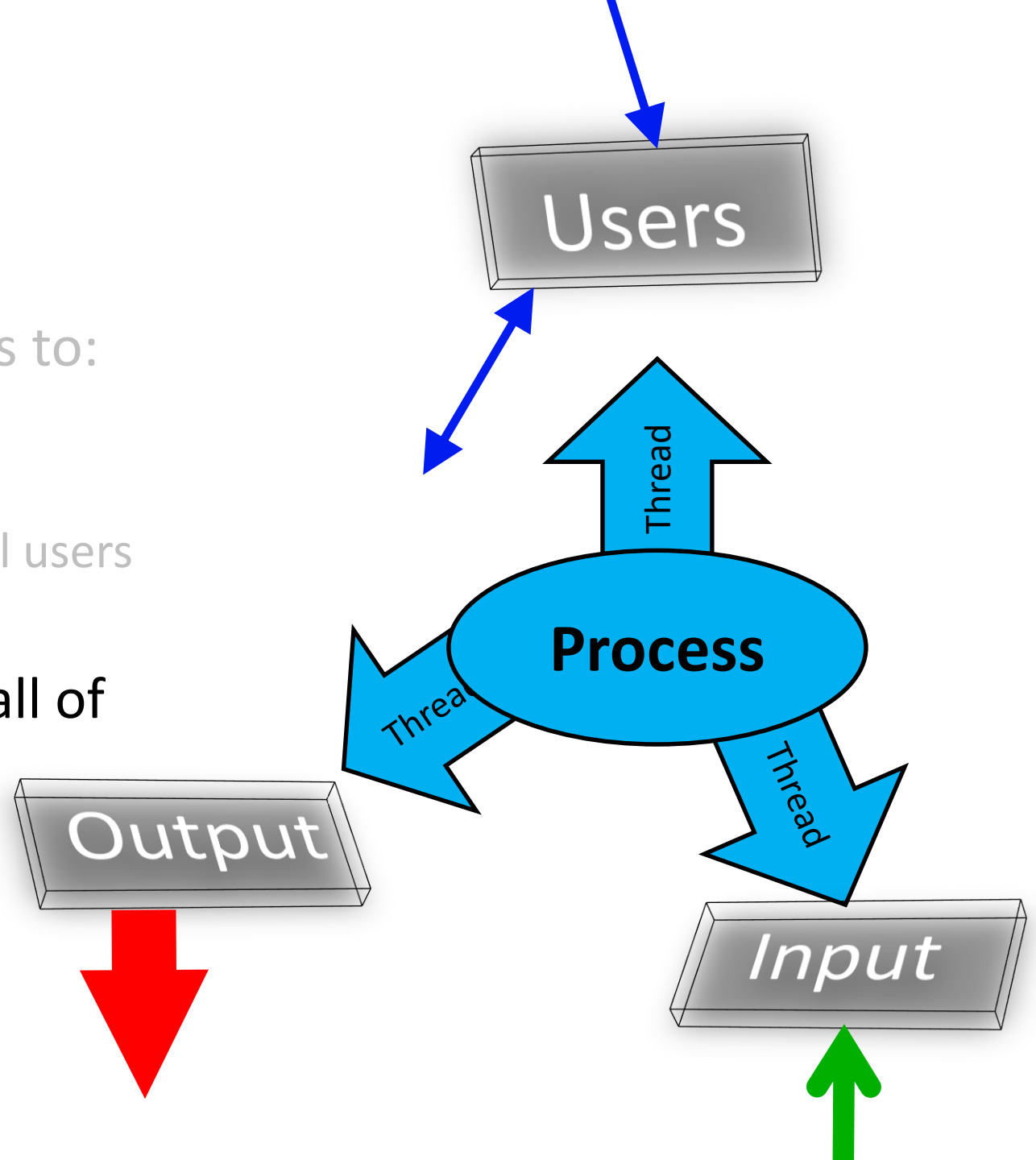
Process-Level Concurrency

- Imagine a chat server that needs to:
 - Watch for users connecting
 - Send chat output to all users
 - Receive chat input from individual users
- But a single **process** can only do one of these things at a time!



Enter Threads

- Imagine a chat server that needs to:
 - Watch for users connecting
 - Send chat output to all users
 - Receive chat input from individual users
- **Threads** allows a process to do all of these things at the same time



Thread Advantages over Processes



- Communication between threads is vastly simpler than IPC:
 - Threads share the following:
 - Code, Heap, Data
 - They each have their own Stack, but they can access the Stacks of other threads(!)
- The CPU switches between executing threads much faster than switching between processes, because of what's shared above
 - On a multi-core CPU, the CPU can run each kernel-level thread on a separate core, yielding true concurrence

Thread Disadvantages



- Shared resources means increased possibilities of:
 - Race conditions
 - Resource contention, including file access
 - Leaking of sensitive data across threads

Types of Threads

- Kernel-Level Threads
 - Controlled by the kernel, given time by the scheduler
 - Akin to a mini-process
- User-Level Threads
 - The kernel doesn't know about these – they are entirely within a process and do not involve the scheduler; switching between them is done cooperatively by the protocol of the software itself to manage the task
 - Really just *emulation* of threading
 - Sometimes called green threads





Thread Type Comparison

- Kernel-Level Threads
 - Controlled by the kernel, given time by the scheduler
 - Simple to create with built-in libraries in UNIX
 - Generally considered the better choice in almost all circumstances
- User-Level Threads
 - Switching between user-level threads is even faster than switching between kernel-level threads, because it doesn't require swapping memory protection to the in-kernel scheduler and back to the process
 - The entire process can be pre-empted by the scheduler
 - A blocking system call blocks the entire process, and thus all of the threads
 - More difficult to use; libraries aren't built-in to UNIX

Thread Implementation in UNIX

- Implemented with the POSIX Threads API in UNIX

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

Compile with `-lpthread` option in gcc



Creating a Thread

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

- **thread**: points to the variable in which the ID of the new thread is written into; depending on OS implementation, sometimes this is an int, sometimes it's a struct

Creating a Thread

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

- **attr**: points to a `pthread_attr_t` struct that contains option flags (NULL if none)

Creating a Thread

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

- **start_routine**: points to a function (in the current program) that will be the start point of execution for the *new* thread that copies this one



Similar to `fork()`, which we'll cover later

Creating a Thread

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

- **arg**: points to the sole argument that is passed into `start_routine` (NULL if none); if multiple arguments are desired, pass a struct

Creating a Thread - Example

```
int resultInt;  
pthread_t myThreadID;  
  
resultInt = pthread_create( &myThreadID,  
                            NULL,  
                            start_routine,  
                            NULL );
```



Destroying a Thread

- Threads can be killed by:
 - The thread calling `pthread_exit()`
 - The thread returns from `start_routine()`
 - The thread gets cancelled by another thread calling `pthread_cancel()`
 - Any thread in the process calls `exit()`



Identifying the Executing Thread

- Here's the function for getting the “thread ID” of an executing thread; note that this ID is not necessarily an integer:

```
pthread_t myThreadID = pthread_self();
```

- Testing for equality:

```
pthread_equal(myThreadID, unknownThreadID);
```


Thread Example – Page 1 of 3

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define NUM_THREADS      5

void* perform_work(void* argument)
{
    int passed_in_value;

    passed_in_value = *((int *) argument);
    printf("Hello World! It's me, thread with argument %d!\n", passed_in_value);
    return NULL;
}
```



Thread Example – Page 2 of 3

```
int main(void)
{
    pthread_t  threads[NUM_THREADS];
    int  thread_args[NUM_THREADS];
    int  result_code, index;

    for (index = 0; index < NUM_THREADS; ++index) {
        // create all threads one by one
        thread_args[index] = index;
        printf("In main: creating thread %d\n", index);
        result_code = pthread_create(&threads[index], NULL,
                                     perform_work, (void *) &thread_args[index]);
        assert(0 == result_code);
    }
}
```

...



Thread Example – Page 3 of 3

...

```
// wait for each thread to complete
```

```
for (index = 0; index < NUM_THREADS; ++index)
{
```

```
    result_code = pthread_join(threads[index], NULL);
```

```
    printf("In main: thread %d has completed\n", index);
```

```
    assert(0 == result_code);
```

```
}
```

```
printf("In main: All threads completed successfully\n");
```

```
exit(EXIT_SUCCESS);
```

```
}
```



Block until thread 'index' completes



Thread Example - Results

```
$ gcc -o threadtest threadtest.c -lpthread
```

```
$ threadtest
```

```
In main: creating thread 0
```

```
In main: creating thread 1
```

```
In main: creating thread 2
```

```
Hello World! It's me, thread with argument 0!
```

```
In main: creating thread 3
```

```
Hello World! It's me, thread with argument 1!
```

```
Hello World! It's me, thread with argument 2!
```

```
In main: creating thread 4
```

```
Hello World! It's me, thread with argument 3!
```

```
In main: thread 0 has completed
```

```
In main: thread 1 has completed
```

```
In main: thread 2 has completed
```

```
Hello World! It's me, thread with argument 4!
```

```
In main: thread 3 has completed
```

```
In main: thread 4 has completed
```

```
In main: All threads completed successfully
```



Mutexes

- An abbreviation for “mutual exclusion”
- Implemented as part of the POSIX `pthread` API to provide thread synchronization via “locks”
- Provides the programmer the ability to protect data from multiple reads and writes on the same files
- There are several types which check variously for errors, perform faster, etc.



The Lifespan of a Mutex

```
pthread_mutex_t myMutex = PTHREAD_MUTEX_INITIALIZER;  
pthread_mutex_destroy(myMutex);
```



Mutex Locking

- A thread acquires a lock on a mutex variable by attempting to call a special lock function:

```
pthread_mutex_t myMutex = PTHREAD_MUTEX_INITIALIZER;  
pthread_mutex_lock(myMutex);
```

- Once the mutex is locked, any other thread attempting to lock it will block!



Mutex Unlocking

- A thread unlocks a mutex variable *that it has previously locked* like so:

```
pthread_mutex_unlock(myMutex);
```

- Once unlocked, one of the other blocked threads (chosen essentially randomly) currently blocked on the mutex will unblock and gain the lock

- A non-blocking attempt to lock the mutex:

```
int resultCode = pthread_mutex_trylock(myMutex);
```



Mutexes Generalized

- A mutex is a form of *semaphore*, which come in two types:
 - Counting semaphore
 - Allow some sort of arbitrary resource count, e.g number of available buffers
 - Binary semaphore
 - Equal to 1 or 0, indicating locked/unavailable or unlocked/available
- Invented by Edsger Dijkstra in 1962 or 1963
 - More on him later!

