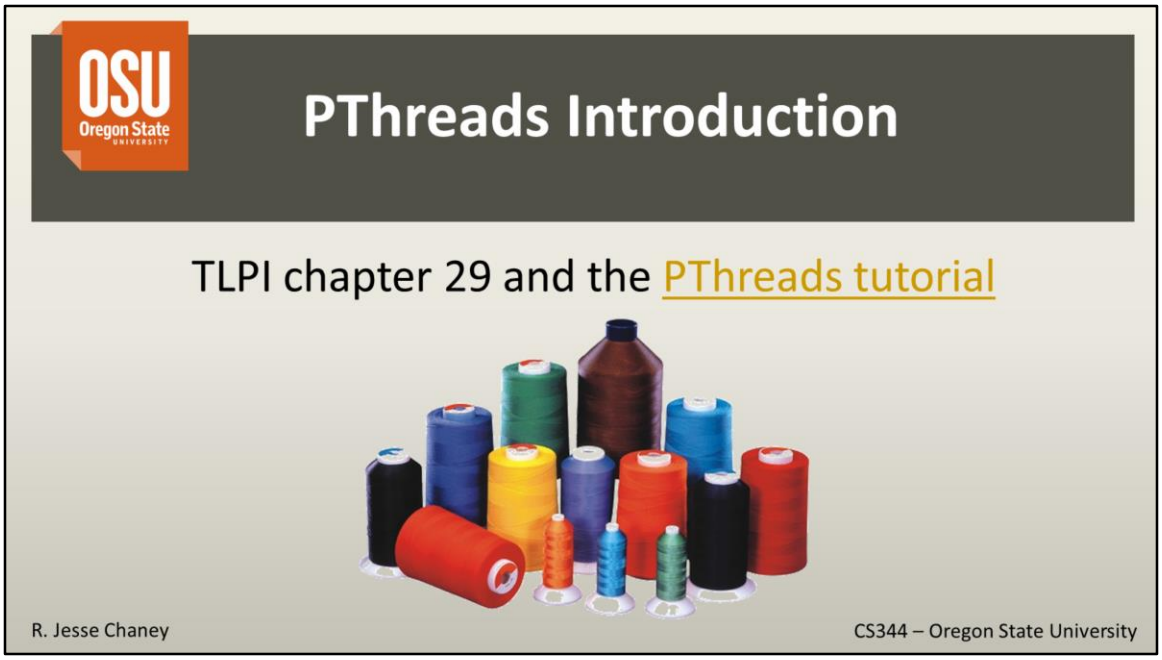
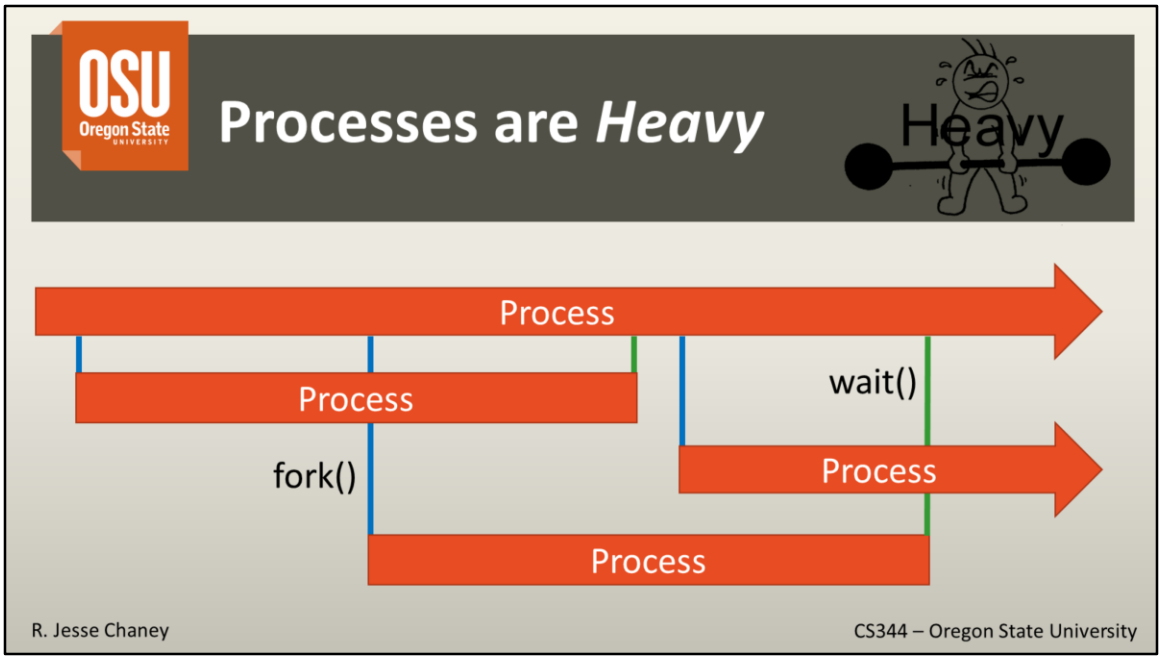


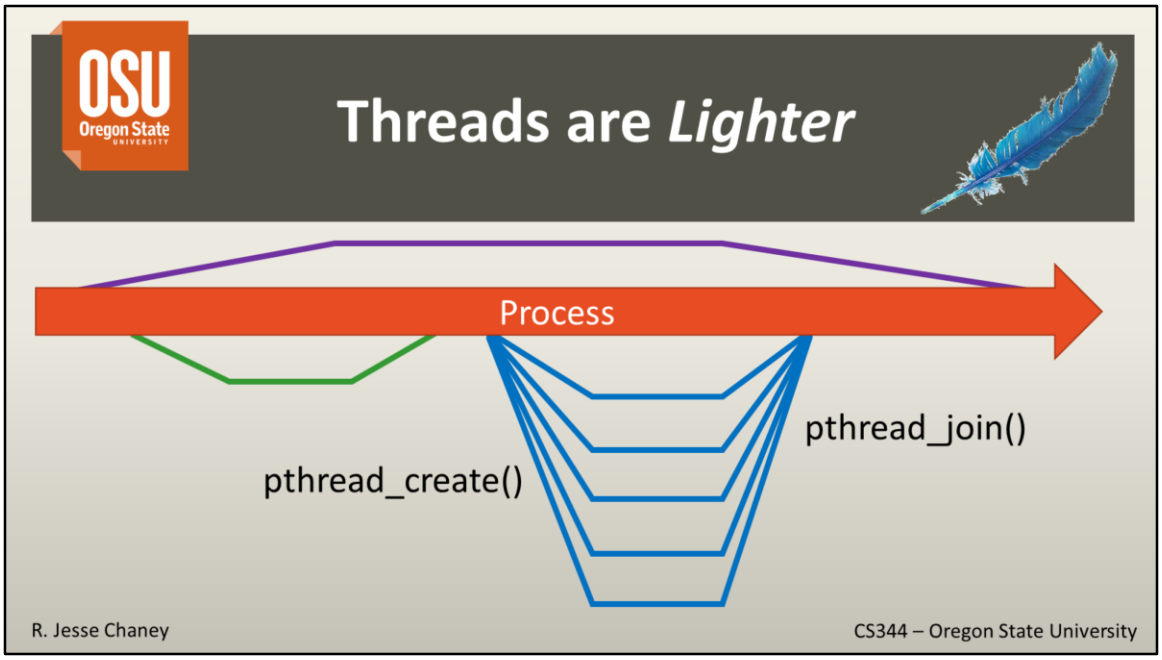
- Like multiple processes, threads represent streams of execution that can be scheduled to run by the operating system.
- Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.
- Technically, a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system.
- To the software developer, the concept of a "*procedure*" that runs independently from its main program may best describe a thread.
- To go one step further, imagine a main program that contains a number of procedures. Then imagine all of these procedures being able to be scheduled to run simultaneously and/or independently by the operating system. That would describe a "*multi-threaded*" program.

Threads History

- Historically, hardware vendors have implemented their own proprietary versions of threads.
- These implementations differed substantially from each other.
- This makes it difficult for programmers to develop portable threaded applications.
- For UNIX systems, a standardized programming interface has been specified by the IEEE POSIX 1003.1c standard.
- Implementations which adhere to this standard are referred to as POSIX threads, or **PThreads**.
- Most hardware vendors now offer PThreads in addition to their proprietary API's.



Processes are heavy. Each time a fork() is called, a new memory address space is created into which a program will be loaded.

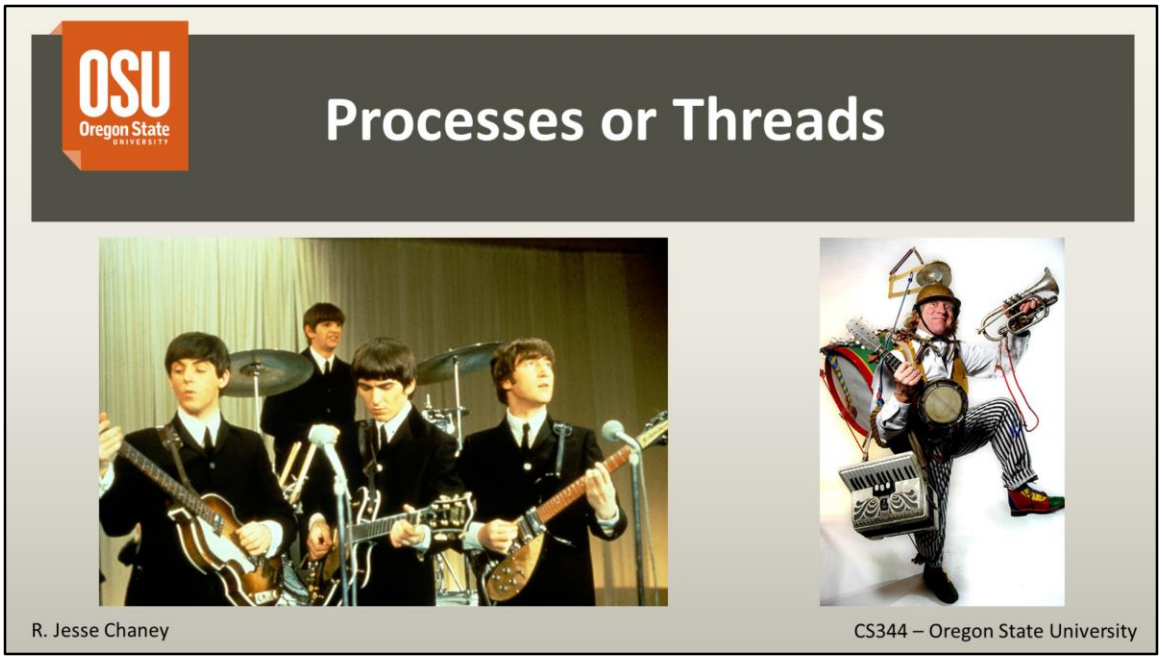


Thread creation is faster than process creation. Comparatively, process creation is slow. The pthreads tutorial has a table showing the relative speed of thread creation over process creation.

Thread creation is like a function call, that keep running after the call returns.

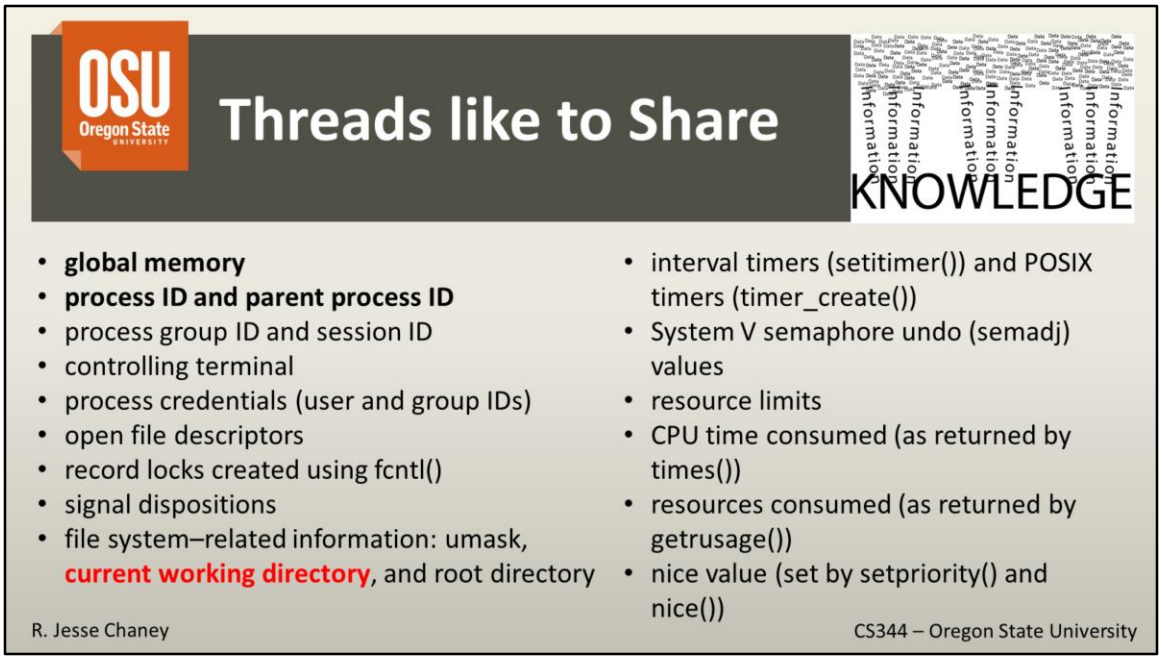
Threads exist within a single process.

All processes have at least 1 thread.



Think of a band with multiple musicians as an example of multi-processing. Each band member represents a process. The processes are all independent streams of execution.

Think of the one-man-band as a multi-threaded application. There is only one process (the single person), but there are multiple streams of execution (multiple instruments) going on with the single process.



Threads like to share a lot of things.

Sharing these among processes can be… … … complex.

Threads don't Share Everything

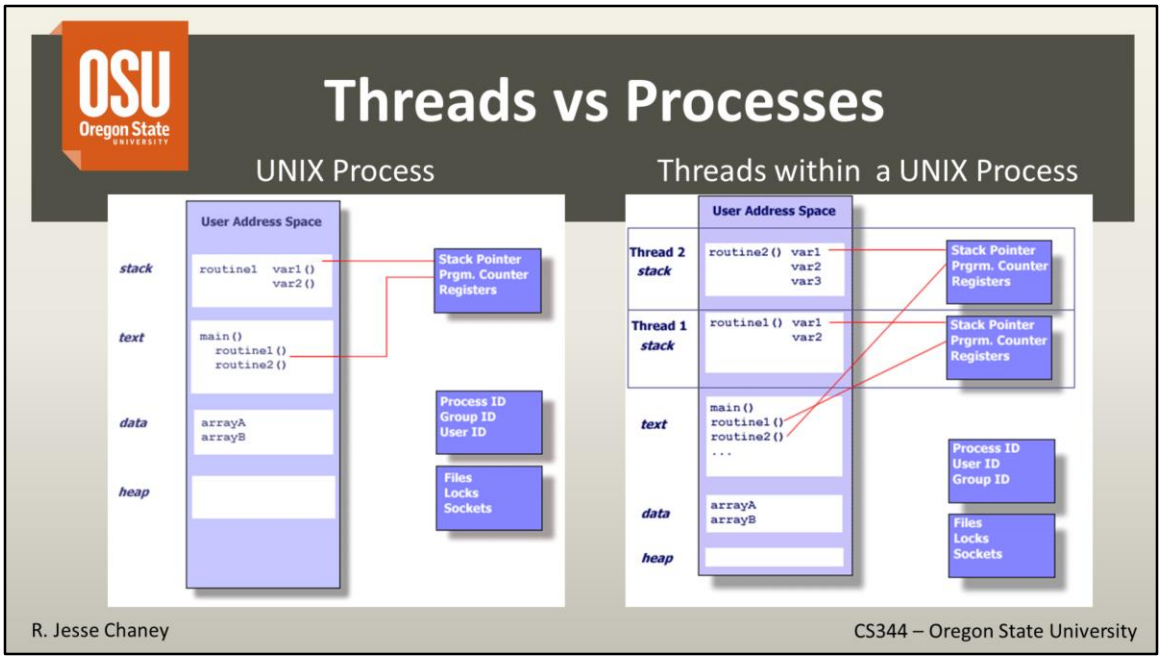
- **thread ID**
- signal mask
- thread-specific data
- alternate signal stack (sigaltstack());
- **the errno variable**
- floating-point environment
- realtime scheduling policy and priority
- CPU affinity
- capabilities
- stack (local variables and function call linkage information).

Threads vs Processes

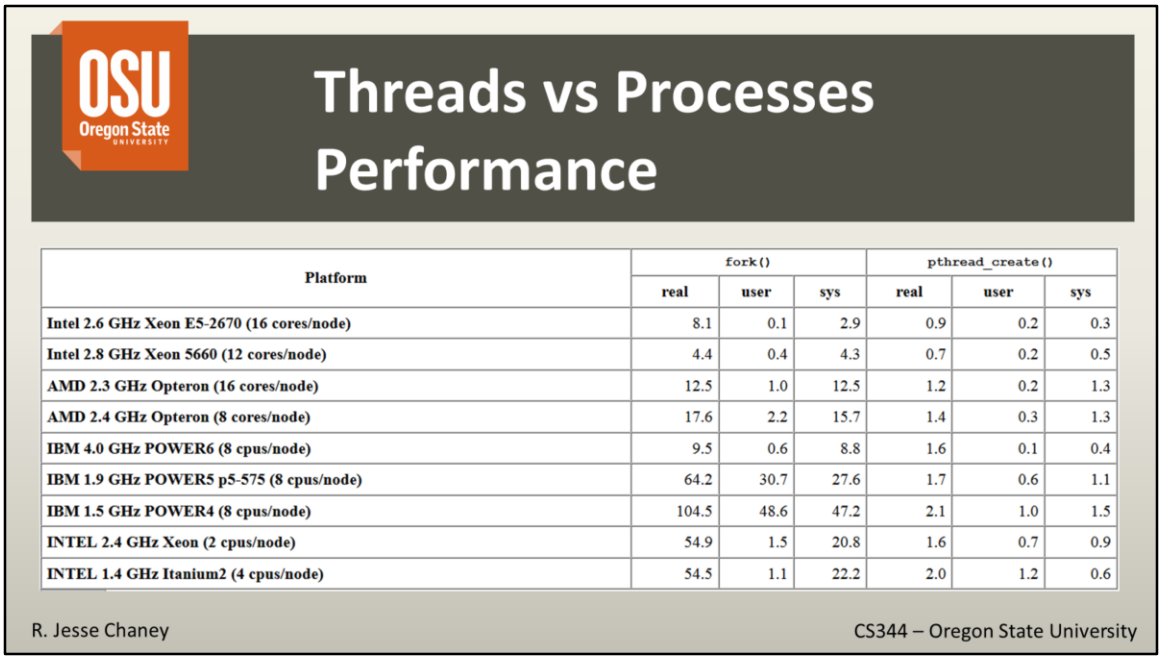
- Sharing data between threads is easy. Sharing data between processes requires work and effort (creating a shared memory segment, using a pipe, …).
- Thread creation is faster than process creation.
- Context-switch time may be lower for threads than for processes.
- Multiple threads can be run from within a single debugger.

Threads vs Processes

- When programming with threads, you need to ensure that the functions you call are thread-safe or are called in a thread-safe manner. Multi-process applications don't need to be concerned with this (or less concerned).
- A bug in one thread (e.g., modifying memory via an incorrect pointer) can damage all of the threads in the process, since they share the same address space and other attributes. Multiple processes are more isolated from one another. One process can crash and burn and not take down other processes.
- Each thread is competing for use of the finite virtual address space of the host process.
- Threads can't span systems. A multi-process application can be spread across several different machines.

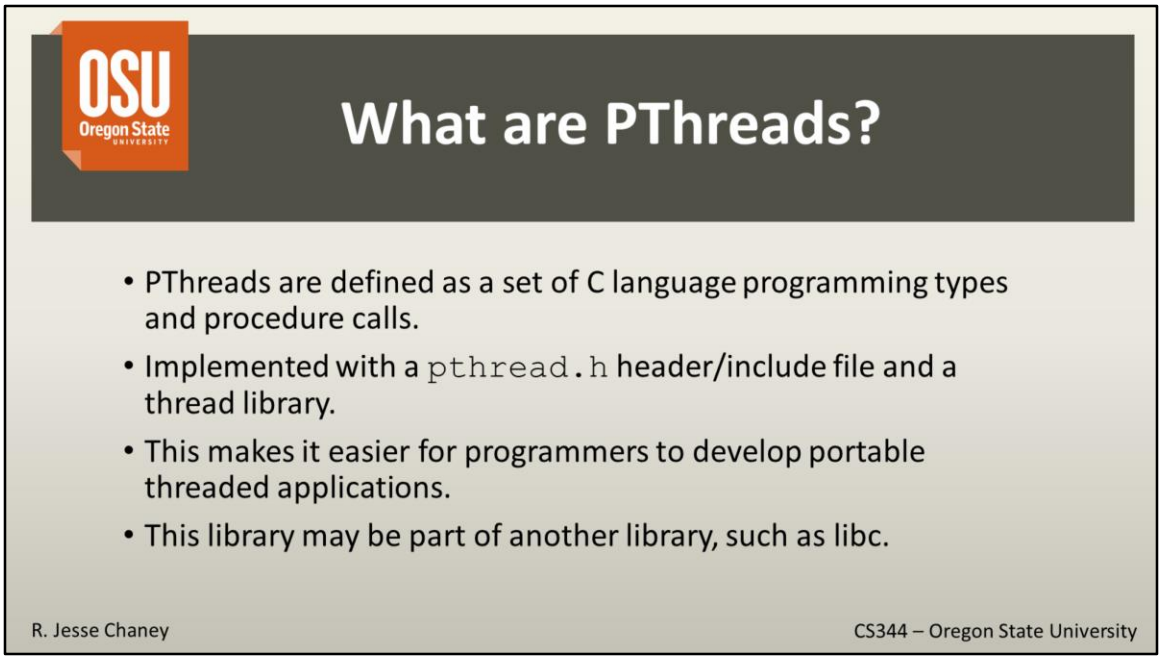


Threads use and exist within these process resources, yet are able to be scheduled by the operating system and run as independent entities largely because they duplicate only the bare essential resources that enable them to exist as executable code.



Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

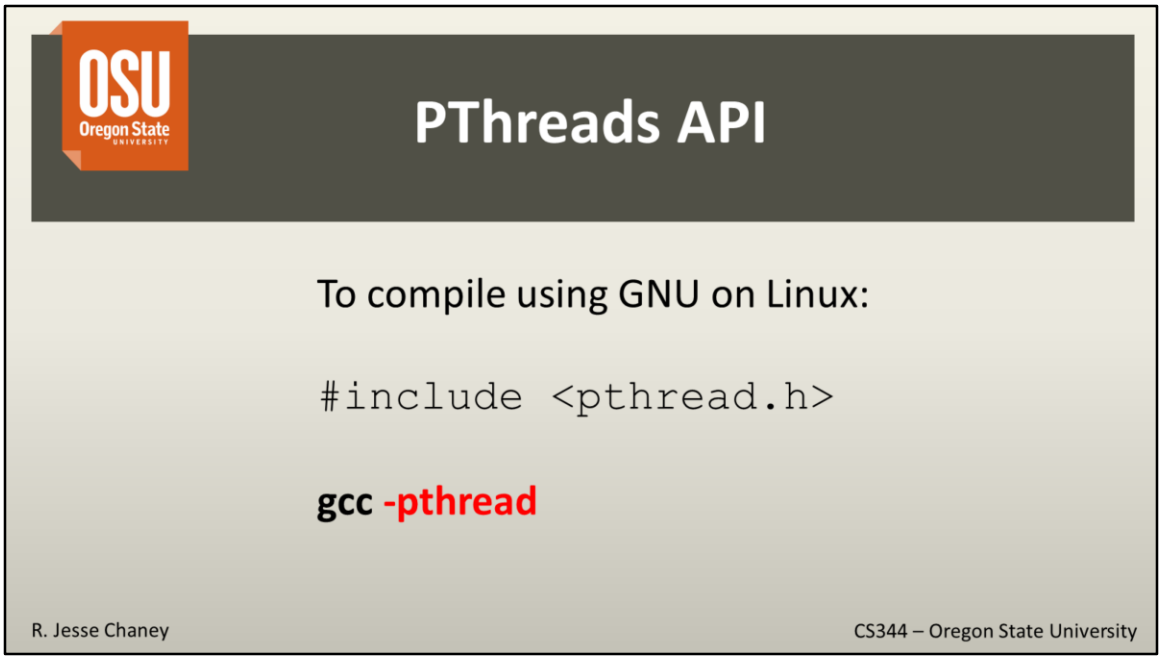
For example, the following table compares timing results for the **fork()** subroutine and the **pthread_create()** subroutine. Timings reflect 50,000 process/thread creations, were performed with the time utility, and units are in seconds, no optimization flags.

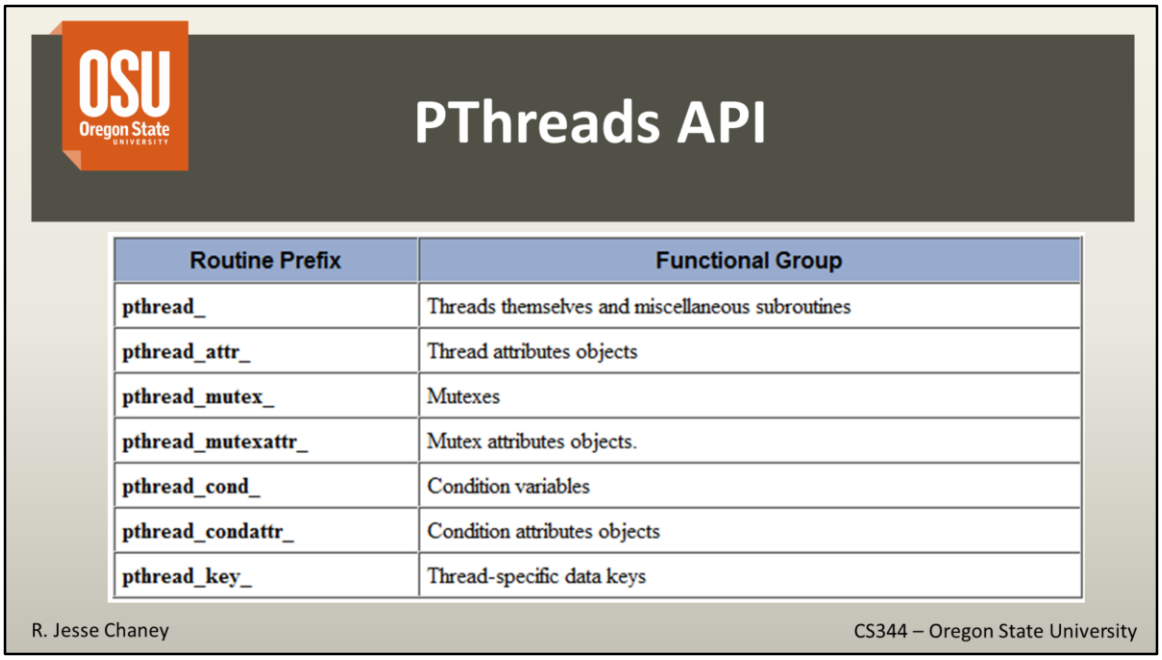


The P in Pthreads stands for POSIX.

On modern, multi-cpu machines, pthreads are ideally suited for parallel programming

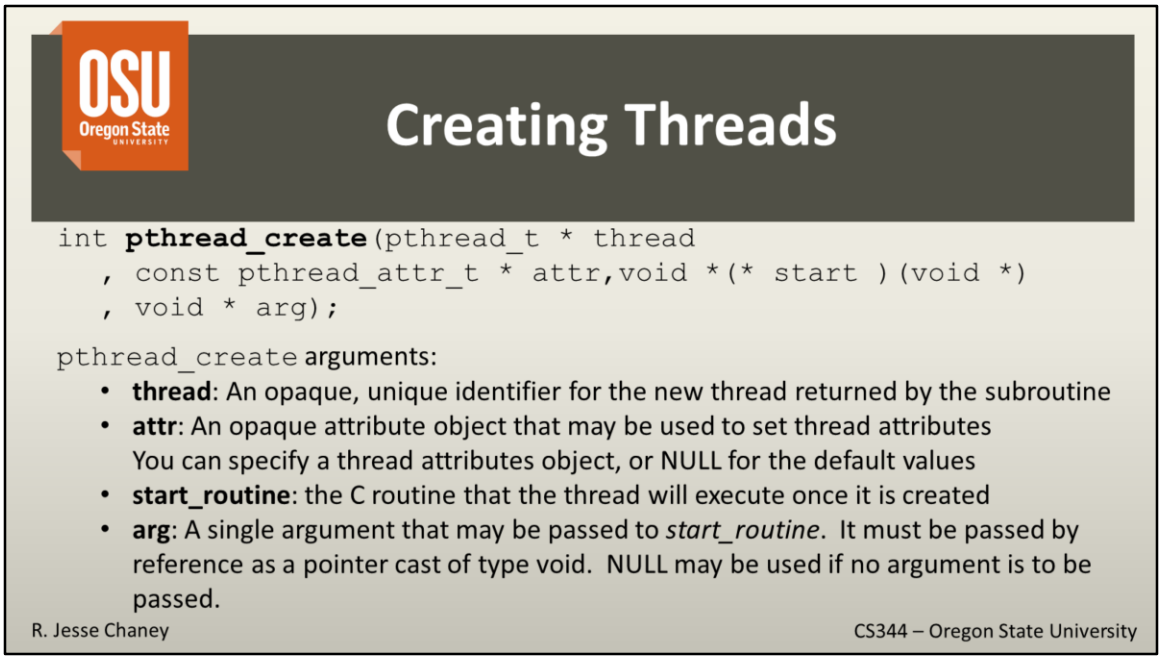
Whatever applies to parallel programming in general, applies to parallel pthreads programs


PThreads API
To compile using GNU on Linux:
#include <pthread.h>
gcc -pthread
R. Jesse Chaney
CS344 – Oregon State University



The subroutines which comprise the Pthreads API can be informally grouped into four major groups:

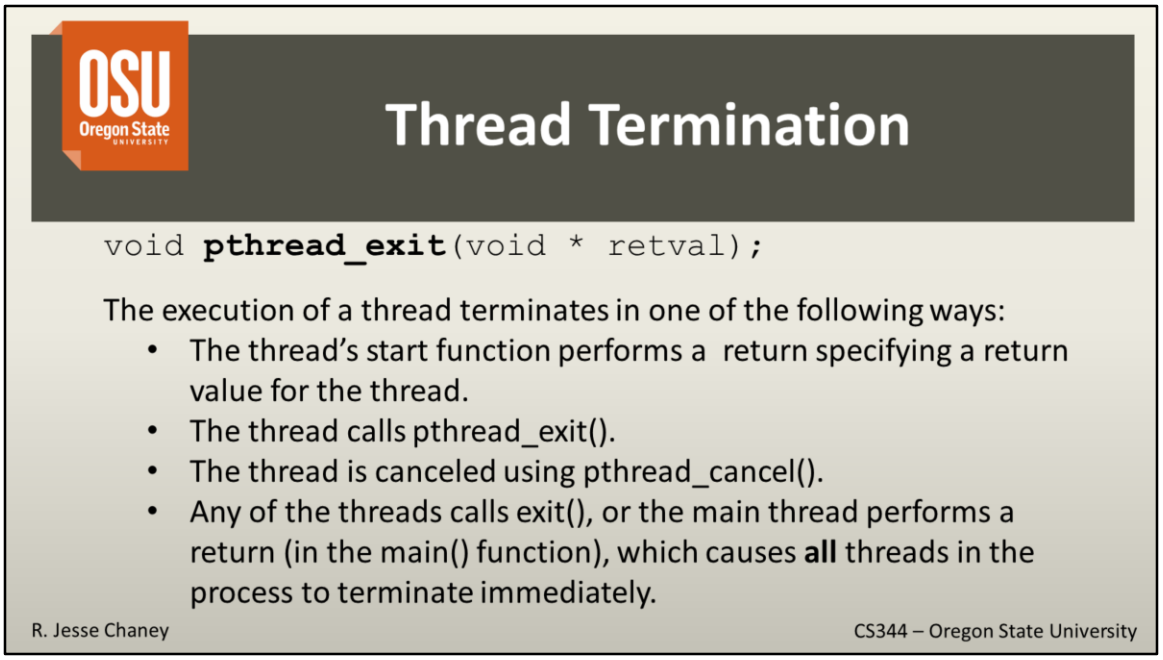
- **Thread management:** Routines that work directly on threads - creating, detaching, joining, etc. They also include functions to set/query thread attributes (joinable, scheduling etc.)
- **Mutexes:** Routines that deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes. These are supplemented by mutex attribute functions that set or modify attributes associated with mutexes.
- **Condition variables:** Routines that address communications between threads that share a mutex. Based upon programmer specified conditions. This group includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.
- **Synchronization:** Routines that manage read/write locks and barriers.



Returns 0 on success, or a positive error number on error.

The new thread commences execution by calling the function identified by start with the argument arg (i.e., start(arg)). The thread that calls pthread_create() continues execution with the next statement that follows the call.

The thread id is unique to a process, but not necessary within a system.



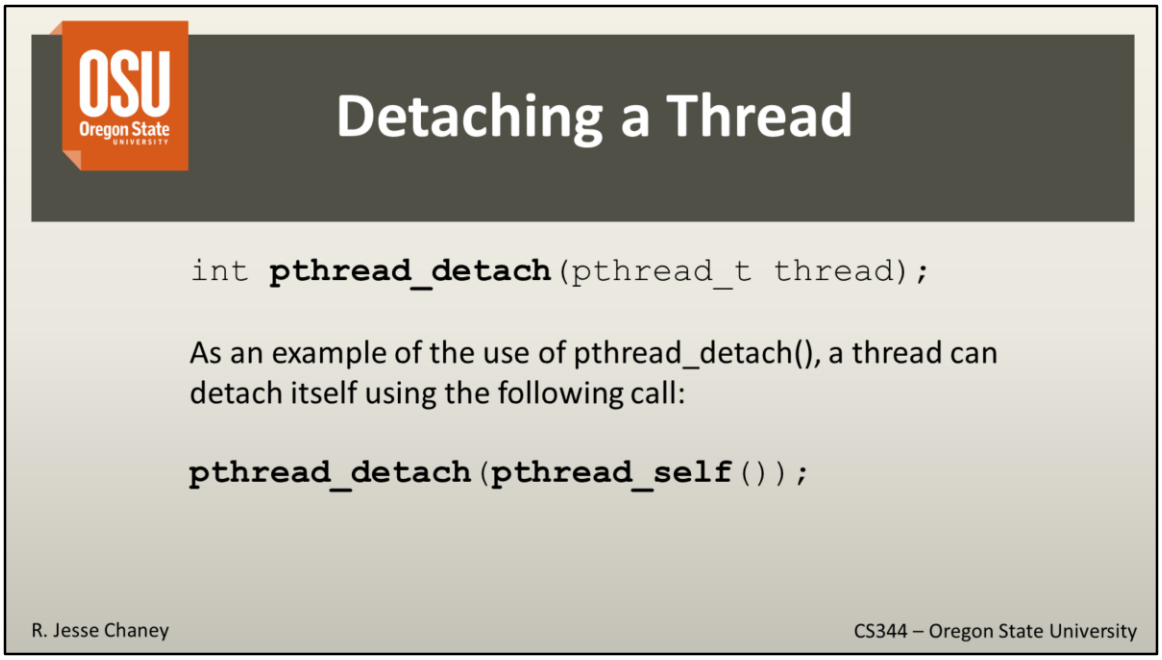
- Returns nonzero value if t1 and t2 are equal, otherwise 0

- Calling pthread_exit() is equivalent to performing a return in the thread's start function, with the difference that pthread_exit() can be called from any function that has been called by the thread's start function.

- The retval argument specifies the return value for the thread. The value pointed to by retval should not be located on the thread's stack, since the contents of that stack become undefined on thread termination.

Returns 0 on success, or a pos.itive error number on error.

If retval is a non- NULL pointer, then it receives a copy of the terminated thread's return value—that is, the value that was specified when the thread performed a return or called pthread_exit().



Returns 0 on success, or a positive error number on error.

By default, a thread is joinable, meaning that when it terminates, another thread can obtain its return status using pthread_join(). Sometimes, we don't care about the thread's return status; we simply want the system to automatically clean up and remove the thread when it terminates. In this case, we can mark the thread as detached, by making a call to pthread_detach() specifying the thread's identifier in thread.

Once a thread has been detached, it is no longer possible to use pthread_join() to obtain its return status, and the thread can't be made joinable again.