# Threads

Dr. Yingwu Zhu

## Processes are expensive

- Recall that a process includes many things
  - An address space (defining all the code and data pages)
  - OS resources (e.g., open files) and accounting information
  - Execution state (PC, SP, regs, etc.)
- Creating a new process is costly because of all of the data structures that must be allocated and initialized
  - FreeBSD: 81 fields, 408 bytes
  - …which does not even include page tables, etc.
- Communicating between processes is costly because most communication goes through the OS
  - Overhead of system calls and copying data

## Parallel programs

- E.g.: A multi-processing web server that forks off copies of itself to handle multiple simultaneous requests
  - Or any parallel program that executes on a multiprocessor
- To execute these programs we need to
  - Create several processes that execute in parallel
  - Cause each to map to the same address space to share data
    - They are all part of the same computation
- Have the OS schedule these processes in parallel (logically or physically)
- This situation is very inefficient
  - Space: PCB, page tables, etc.
  - Time: create data structures, fork and copy addr space, etc.

# Rethinking Processes

- What is similar in these cooperating processes?
  - They all share the same code and data (address space)
  - They all share the same privileges
  - They all share the same resources (files, sockets, etc.)
- What don't they share?
  - Each has its own execution state: PC, SP, and registers
- Key idea: Why don't we separate the concept of a process from its execution state?
  - Process: address space, privileges, resources, etc.
  - Execution state: PC, SP, registers
- Exec state also called thread of control, or thread

## Threads

- Modern OSes (Mach, Chorus, NT, modern Unix) separate the concepts of processes and threads
  - The thread defines a sequential execution stream within a process (PC, SP, registers)
  - The process defines the address space and general process attributes (everything but threads of execution)
- A thread is bound to a single process
  - Processes, however, can have multiple threads
- Threads become the unit of scheduling
  - Processes are now the containers in which threads execute
  - Processes become static, threads are the dynamic entities

# Multi-threaded Models

A thread is a subset of a process:

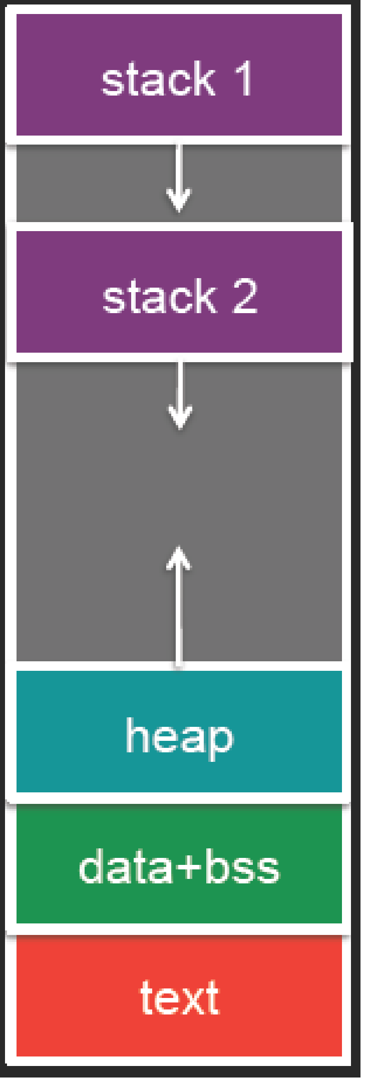
- A process contains one or more kernel threads

Share memory and open files

- BUT:
  separate program counter, registers, and stack
- Shared memory includes the heap and global/ static data
- No memory protection among the threads

Preemptive multitasking:

- Operating system preempts & schedules threads



## Sharing

- The items that are shared among threads within a process are:
  - Text segment (instructions)
  - Data segment (static and global data)
  - BSS segment (uninitialized data)
  - Open file descriptors
  - Signals
  - Current working directory
  - User and group IDs

# Not Shared

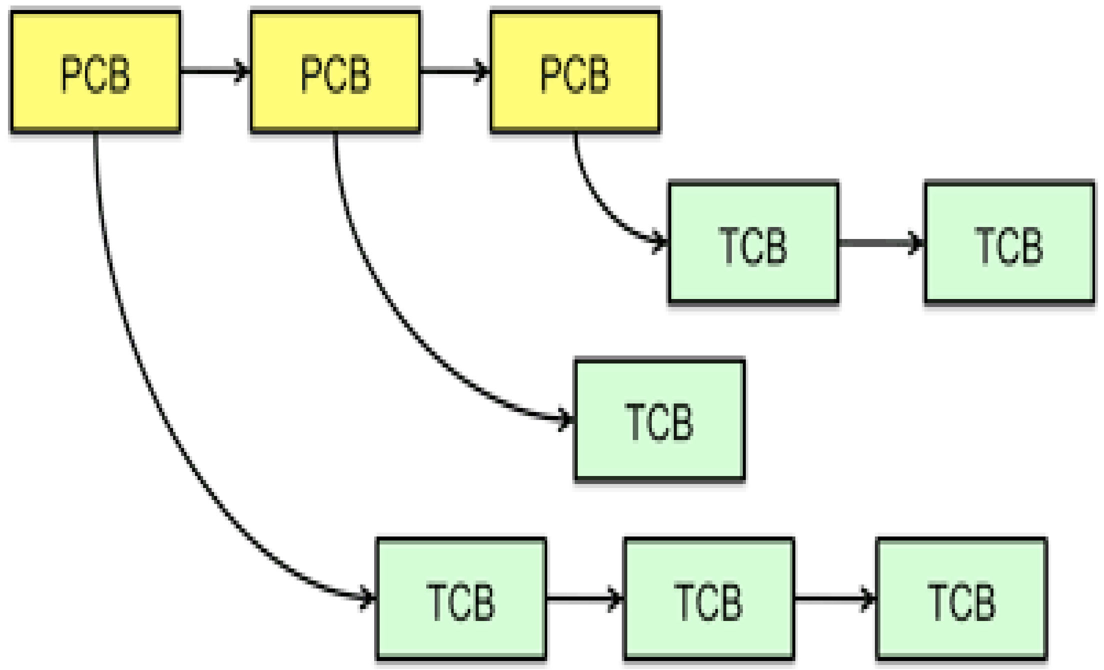
- A multithread-aware OS also needs to keep track of threads
- The items that the operating system must store that are unique to each thread are:
  - Thread ID
  - Saved registers, stack pointer, instruction pointer
  - Stack (local variables, temporary variables, return addresses)
  - Signal mask
  - Priority (scheduling information)

# Why Threads?

- Threads are more efficient
  - Much less overhead to create: no need to create a new memory map and allocate new structures to track open files & reference counts
- Sharing memory is easy (inborn)
  - Do not need IPC
- Take advantage of multiple CPUs – just like processes
  - Scale in performance as the number of processors or cores increases

# How OS manage threads?

- PCB contains one or more Thread Control Blocks (TCB):
  - Thread ID
  - Saved registers
  - Other per-thread info (signal mask, scheduling parameters)



Thread control blocks

## Scheduling

- A traditional, non-multithreaded OS scheduled processes
- A thread-aware OS schedules *threads*, not *processes*
  - A process is just a container for one or more threads

## Scheduling

- Scheduler has to realize
  - Context switch among threads of different processes is more expensive
    - Flush cache memory (or have memory with process tags)
    - Flush virtual memory TLB (or have tagged TLB)
    - Replace page table pointer in memory management unit
  - Scheduling threads onto a different CPU is more expensive
    - The CPU’s cache may have memory used by the thread cached
    - CPU affinity

# Thread Programming Pattern

- Single task thread
  - Create a thread for a specific job and the thread exits upon completion
- Worker thread
  - Specific task for each worker thread
  - Dispatch task to the thread that handles it
- Thread pools
  - Creates a number of threads upon start-up
  - All of these threads then grab work items off the same work queue
  - Wait if no thread available
  - Common model for servers

# Kernel-level threads vs. User-level threads

- Kernel level
  - Threads supported by OS
  - OS handles scheduling, creation, synchronization
- User level
  - Library with code for creation, termination, scheduling
  - Kernel sees one execution context: one process
  - May or may not be preemptive

## User-level threads & Threading library

- A threading library is responsible for handling the saving and switching of the execution context from one thread to another
  - Allocate a region of memory within the process that will serve as a stack for each thread
  - Save and swap registers and the instruction pointer as the library switches execution from one thread to another
  - Threads call the threading library to yield its use of the processor to another thread
  - Or: ask OS for a timer-based interrupt (e.g., *setitimer* system call). When the process gets the interrupt (via the *signal* mechanism), the function in the threading library that registered for the signal is called and handles the saving of the current registers, stack pointer, and stack and restoring those items from the saved context of another thread.

## User-level threads

- Advantages
  - Low cost: user level operations, no switching to the kernel
  - Scheduling algorithms can be replaced eaisly & custom to app
  - Greater portability
- Disadvantages
  - If a thread is blocked, all threads for the process are blocked
    - Every system call needs an asynchronous counterpart
  - Cannot take advantage of multiprocessing

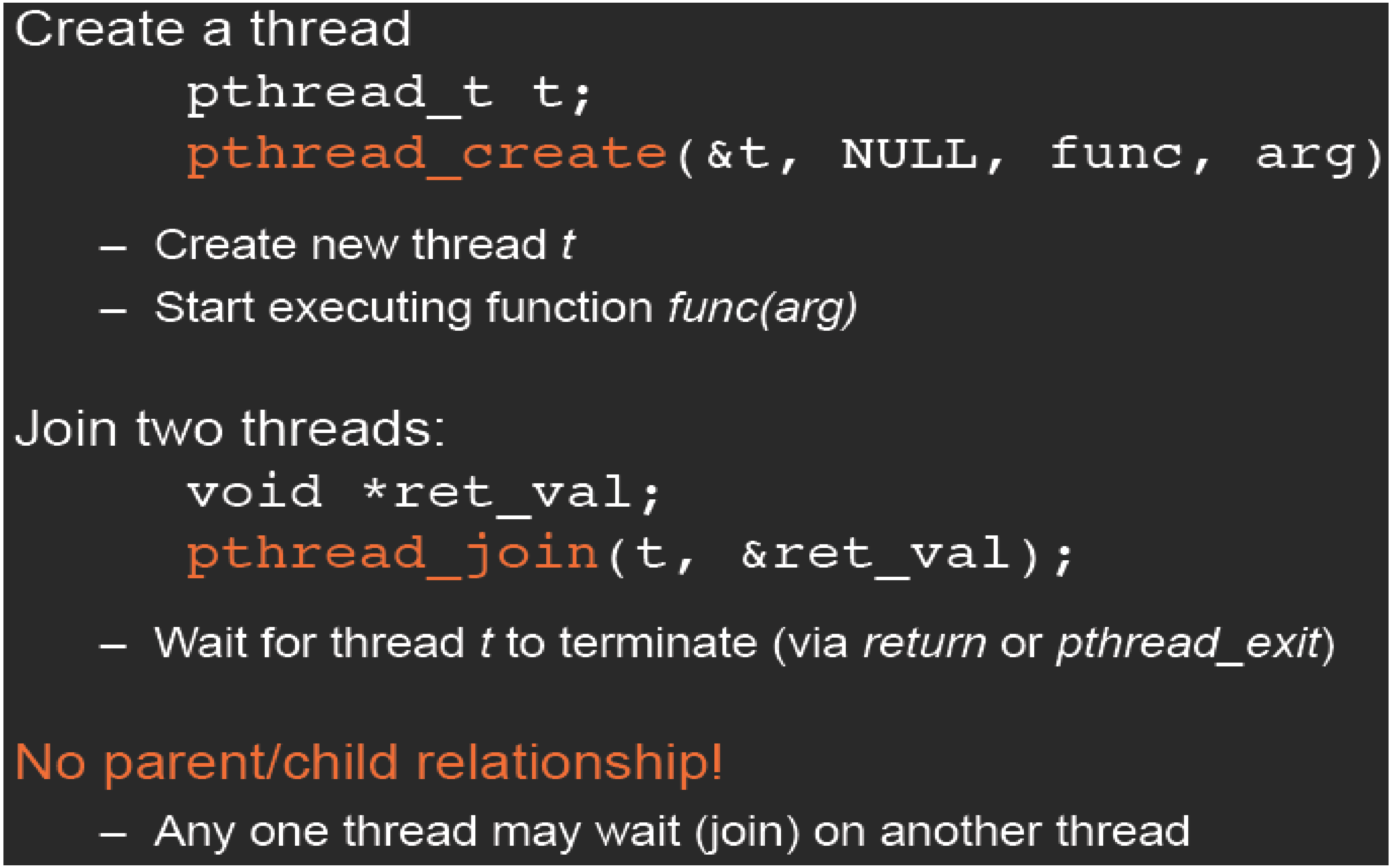
## You can have both!

- User-level thread library on top of multiple kernel threads
- 1:1 – pure kernel threads only
  - 1 user thread = 1 kernel thread
  - E.g. win32, Linux, Windows 7, FreeBSD
- N:1 – pure user threads only
  - N user threads on 1 kernel thread/process
  - E.g. Early version of Java
- N:M – hybrid threading
  - N user threads on M kernel threads

# POSIX Threads (Pthreads)

- Low-level threading libraries
- Native threading interface for Linux now
- Use kernel-level thread (1:1 model)
- Developed by the IEEE committees in charge of specifying a portable operating system interface (POSIX)
- Shared memory

# Using Pthreads



Create a thread

```
    pthread_t t;
    pthread_create(&t, NULL, func, arg)
```

- Create new thread *t*
- Start executing function *func(arg)*

Join two threads:

```
    void *ret_val;
    pthread_join(t, &ret_val);
```

- Wait for thread *t* to terminate (via *return* or *pthread_exit*)

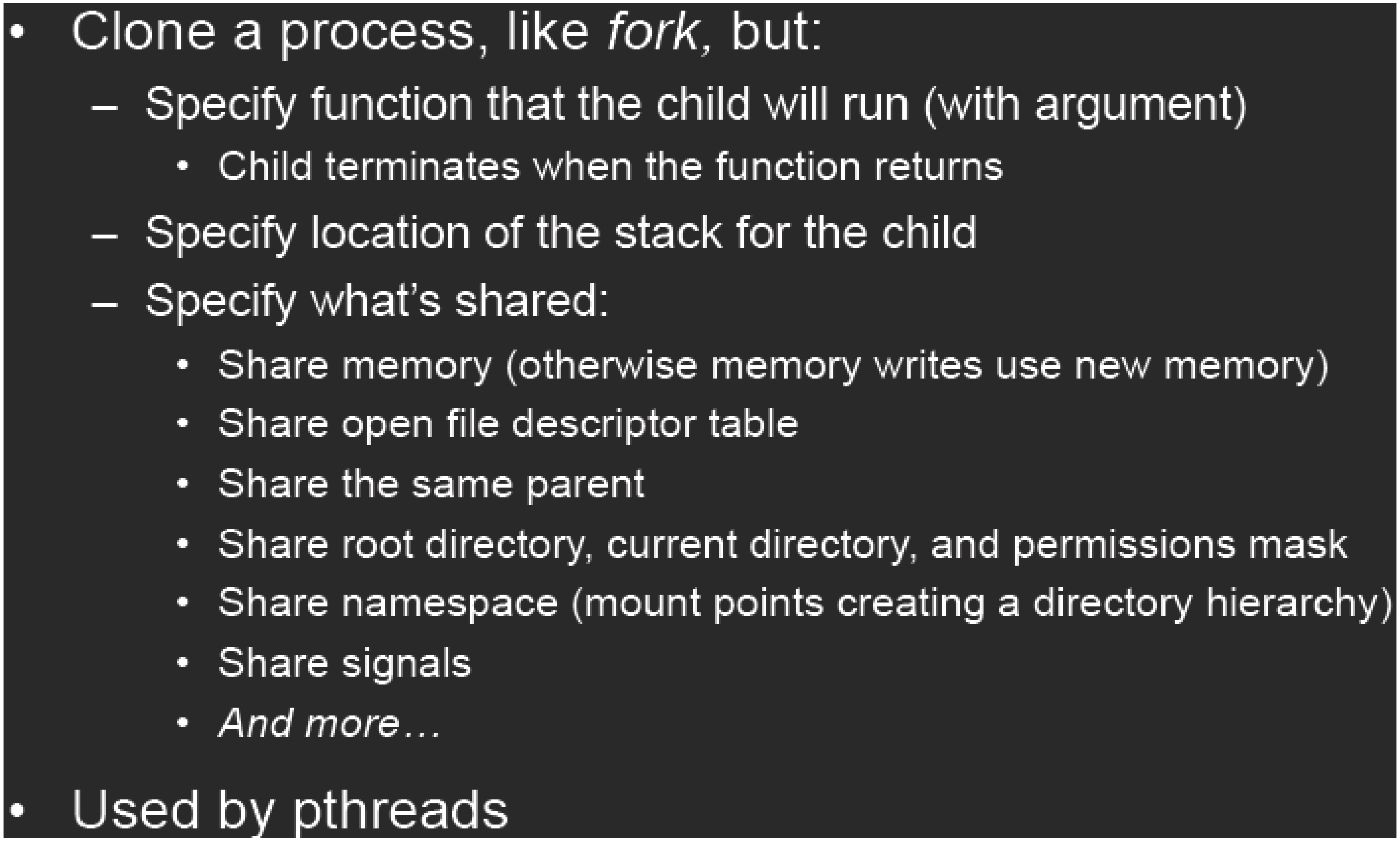
No parent/child relationship!

- Any one thread may wait (join) on another thread

## Example

```
#include <pthread.h>
#include <stdio.h>
void * entry_point(void *arg) {
    printf("Hello world!\n");
    pthread_exit(NULL);        //return NULL;
}
int main(int argc, char **argv) {
     pthread_t   thr;
     if(pthread_create(&thr, NULL, &entry_point, NULL)) {
          printf("Could not create thread\n");
          return -1; }
      if(pthread_join(thr, NULL)) {
          printf("Could not join thread\n");
          return -1; }
     return 0;
}
```

# Linux *clone()* system call



- Clone a process, like *fork*, but:
  - Specify function that the child will run (with argument)
    - Child terminates when the function returns
  - Specify location of the stack for the child
  - Specify what's shared:
    - Share memory (otherwise memory writes use new memory)
    - Share open file descriptor table
    - Share the same parent
    - Share root directory, current directory, and permissions mask
    - Share namespace (mount points creating a directory hierarchy)
    - Share signals
    - *And more…*
- Used by pthreads

# Acknowledgement

- Some slides are adapted from Dr. Paul Krzyzanowski