CS3281 / CS5281

Concurrent Programming

Will Hedgecock
Sandeep Neema
Bryan Ward

**Some lecture slides borrowed and adapted from CMU's "Computer Systems: A Programmer's Perspective"*

Motivation

- Idea: Use multiple processes to "collaborate" to accomplish work faster
  - Divide up work between two or more threads
    - Single-core hardware performance advances slowly, but we are seeing more cores and parallel architectures
  - Run concurrently on multiple cores
    - Fun fact: it's actually possible to get "super-linear" speedups!
  - A process may be able to run while another waits for some event
    - e.g., waiting on a socket for a network packet (as we will see later)
- What challenges might we face?

Terminology

- Concurrent programming: a programming paradigm in which multiple tasks are executed in overlapping periods
  - Enables multiple operations to be executed out-of-order without adversely affecting final outcome

- Benefits
  - Speed & responsiveness
  - Better resource utilization
  - Scalability in modern architectures

Traditional View of a Process

- Process = process context + code, data, and stack

Process context

Program context:
- Data registers
- Condition codes
- Stack pointer (SP)
- Program counter (PC)

Kernel context:
- VM structures
- Descriptor table
- brk pointer

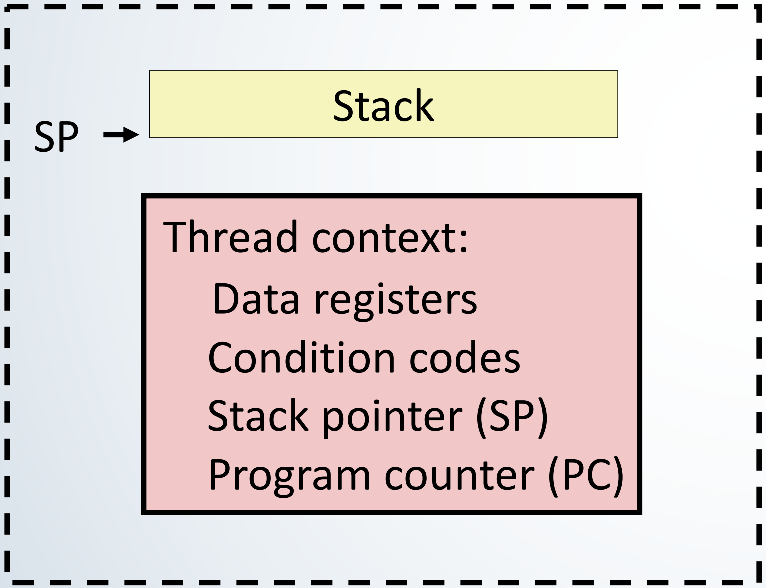
Code, data, and stack

	Stack
SP →	
	Shared libraries
brk →	
	Run-time heap
	Read/write data
PC →	Read-only code/data
0	

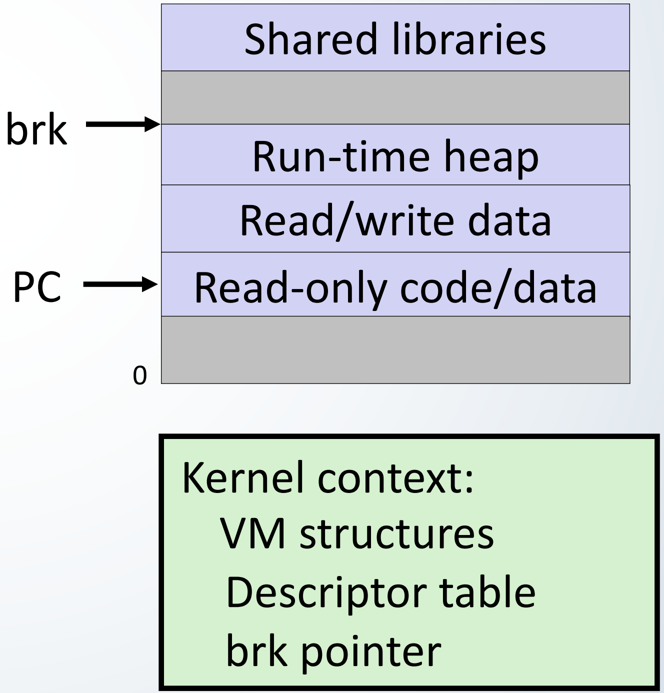
Alternate View of a Process

- Process = thread + code, data, and kernel context

Thread (main thread)



Code, data, and kernel context



A Process with Multiple Threads

- Multiple threads can be associated with a process
  - Each thread has its own logical control flow
  - Each thread shares the same code, data, and kernel context (unlike processes)
  - Each thread has its own stack for local variables
    - but not protected from other threads – why?
  - Each thread has its own thread id (TID)

Thread 1 (main thread)

stack 1

Thread 1 context:
Data registers
Condition codes
SP1
PC1

Thread 2 (peer thread)

stack 2

Thread 2 context:
Data registers
Condition codes
SP2
PC2

Shared code and data

shared libraries

run-time heap
read/write data
read-only code/data

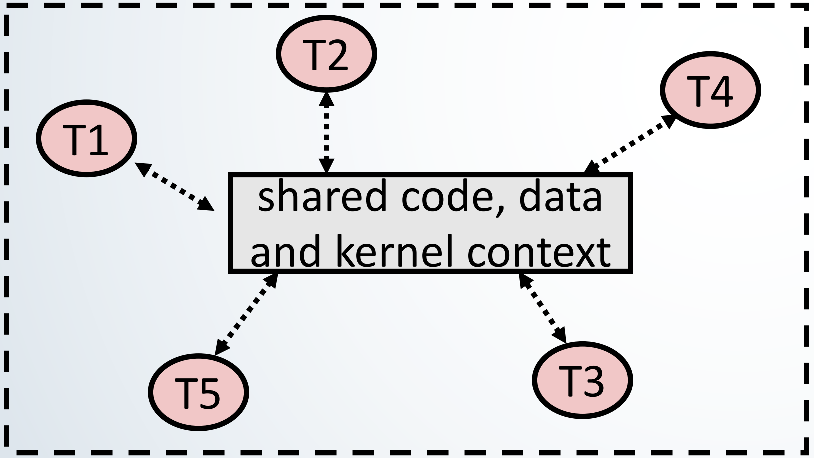
0

Kernel context:
VM structures
Descriptor table
brk pointer

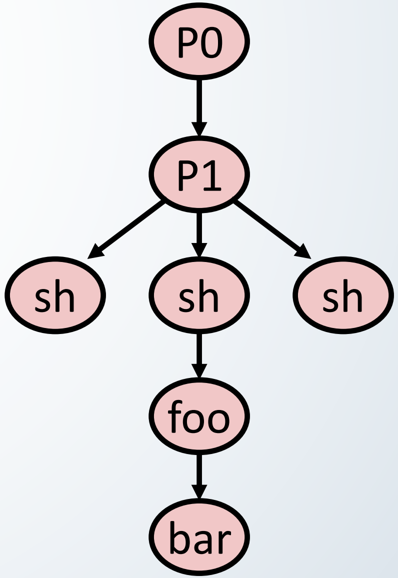
Logical View of Threads

- Threads associated with process form a pool of peers
  - Unlike processes, which form a tree hierarchy

Threads associated with process foo



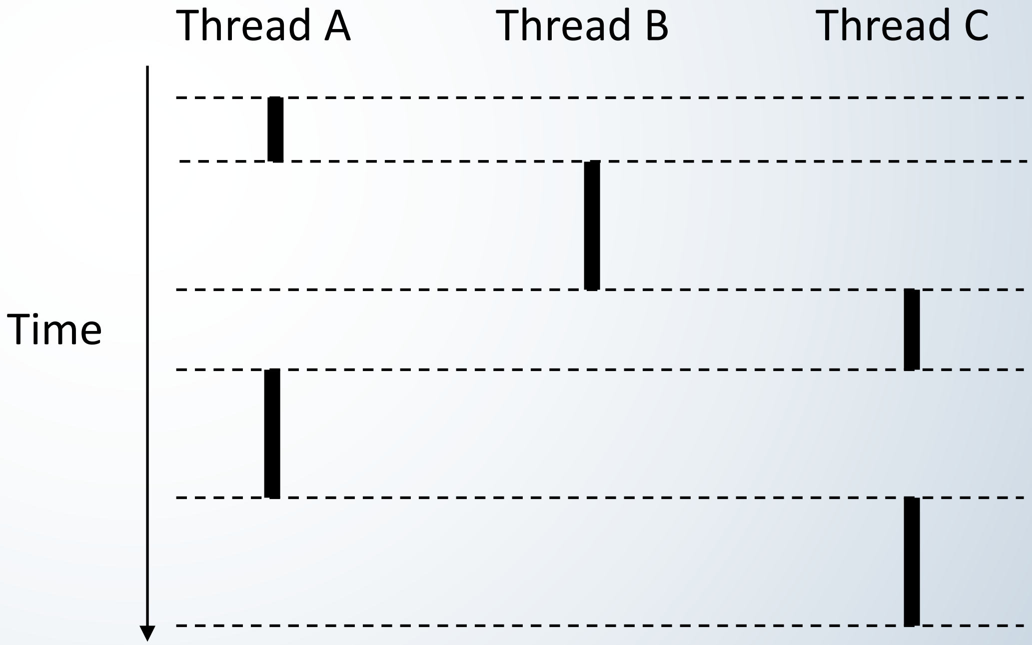
Process hierarchy



Concurrent Threads

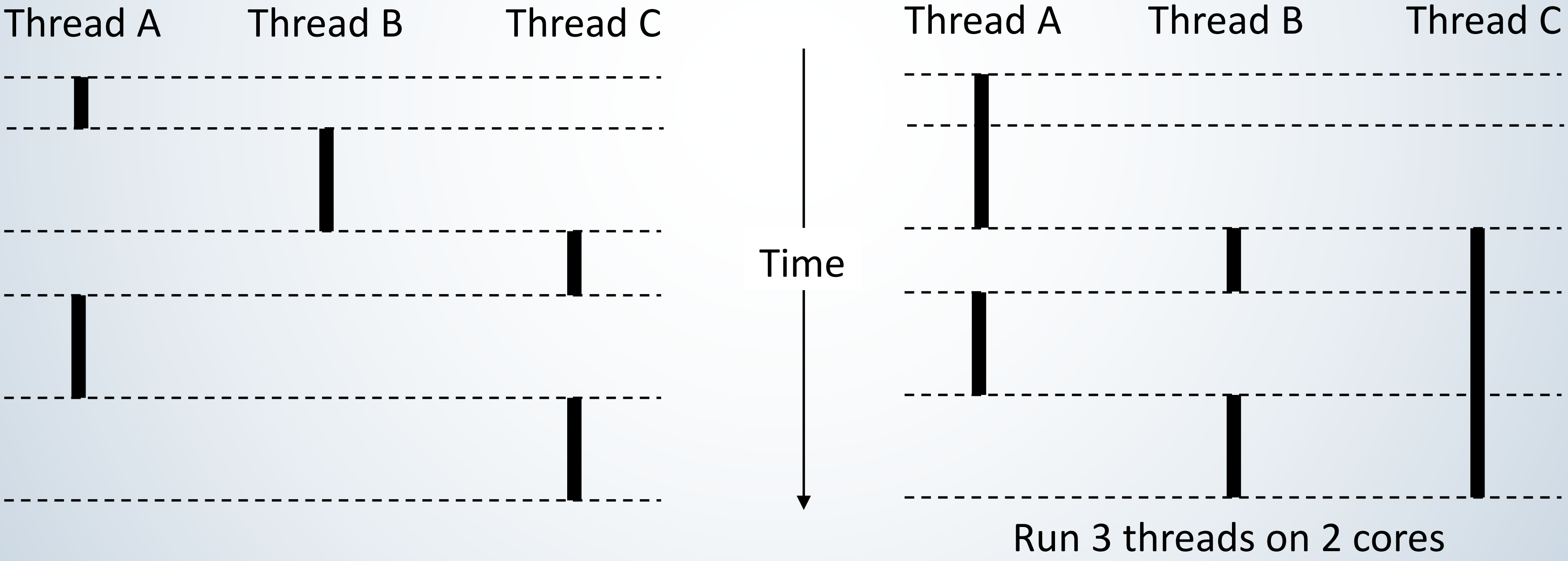
- Two threads are *concurrent* if their flows overlap in time
- Otherwise, they are sequential

- Examples:
  - Concurrent: A & B, A&C
  - Sequential: B & C



Concurrent Thread Execution

- Single Core Processor
  - Simulate parallelism by time slicing
- Multi-Core Processor
  - Can have true parallelism



Threads vs. Processes

- How threads and processes are similar
  - Each has its own logical control flow
  - Each can run concurrently with others (possibly on different cores)
  - Each is context switched
- How threads and processes are different
  - Threads share all code and data (except local stacks)
    - Processes (typically) do not
  - Threads are somewhat less expensive than processes
    - Process control (creating and reaping) twice as expensive as thread control
    - Linux numbers:
      - ~20K cycles to create and reap a process
      - ~10K cycles (or less) to create and reap a thread
    - Switching between threads of the same process is faster than switching between processes

Posix Threads (pthreads) Interface (not xv6)

- *Pthreads:* Standard interface for ~60 functions that manipulate threads from C programs
  - Creating and reaping threads
    - `pthread_create()`
    - `pthread_join()`
  - Determining your thread ID
    - `pthread_self()`
  - Terminating threads
    - `pthread_cancel()`
    - `pthread_exit()`
    - `exit()` [terminates all threads] , `RET` [terminates current thread]
  - Synchronizing access to shared variables
    - `pthread_mutex_init`
    - `pthread_mutex_[un]lock`

Tel (615) 343-7472 | Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu

The pthreads "Hello, World" Program

```c
// hello.c - Pthreads "hello, world" program
#include "csapp.h"
void *thread(void *vargp);

int main()
{
    pthread_t tid;
    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}
```

hello.c

Thread ID

Thread attributes (usually NULL)

Thread routine

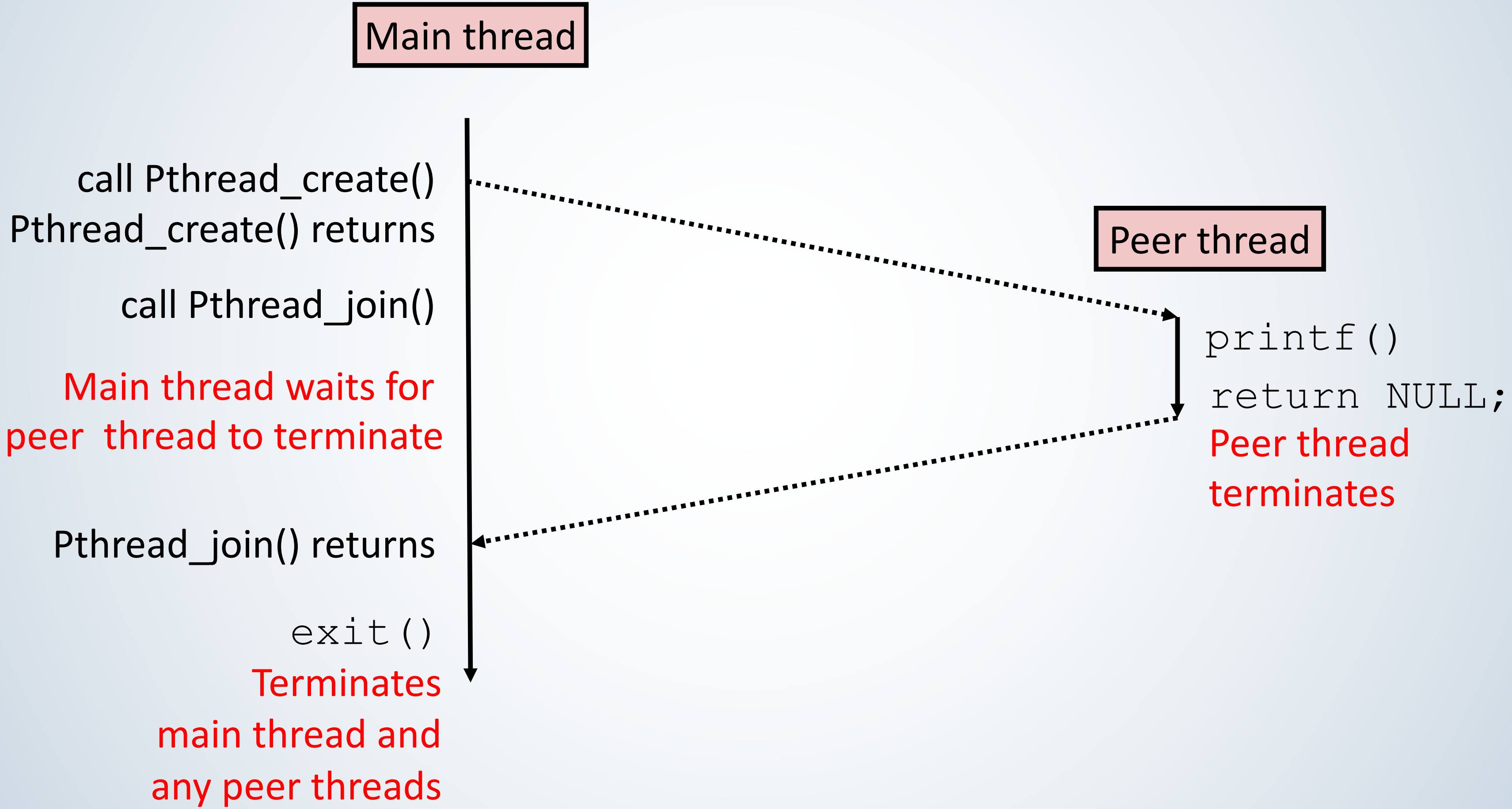
*Thread arguments (void *p)*

*Return value (void **p)*

```c
void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!\n");
    return NULL;
}
```

hello.c

Execution of Threaded "Hello, World"



Issues with Threads

- Must run "detached" to avoid memory leak
  - At any point in time, a thread is either *joinable* or *detached*
  - *Joinable* thread can be reaped and killed by other threads
    - must be reaped (with `pthread_join`) to free memory resources
  - *Detached* thread cannot be reaped or killed by other threads
    - resources are automatically reaped on termination
  - Default state is joinable
    - use `pthread_detach(pthread_self())` to make detached
- Must be careful to avoid unintended sharing
  - For example, passing pointer to main thread's stack
    - `pthread_create(&tid, NULL, thread, (void *)buffer);`
- All functions called by concurrent thread must be *thread-safe*
  - (next lecture)

Pros and Cons of Thread-Based Designs

- + Easy to share data structures between threads
  - e.g., logging information, file cache
- + Threads are more efficient than processes

- – Unintentional sharing can introduce subtle and hard-to-reproduce errors!
  - The ease with which data can be shared is both the greatest strength and the greatest weakness of threads
  - Hard to know which data should be shared vs. private
  - Hard to detect by testing
    - Probability of bad "race" outcome very low
    - But nonzero!
  - Future lectures

ISIS
Tel (615) 343-7472 | Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu
VANDERBILT UNIVERSITY