Announcements

- Assignment 9
  - Writing your own shell
  - Due 11:59 p.m., Monday, December 8
- Next week
  - Monday
    - Quiz on processes (Lec 32-33)
    - Lecture on floating-point number representation
  - Wednesday
    - Review session: 2-4 p.m. MECC 127
  - Thursday
    - Final exam: comprehensive

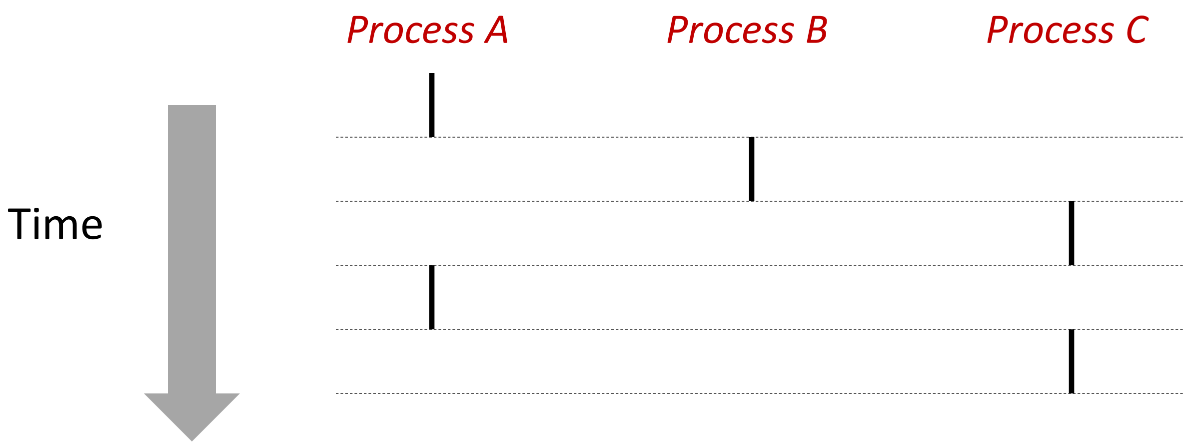
Lecture 34

Concurrency

CPSC 275
Introduction to Computer Systems

Concurrent Processes

- Two processes *run concurrently* if their flows overlap in time
- Otherwise, they are *sequential*
- Examples (running on single core):
  - Concurrent: A & B, A & C
  - Sequential: B & C



Concurrent Programming is Hard!

- The human mind tends to be sequential
- The notion of time is often misleading
- Thinking about all possible sequences of events in a computer system is error prone and frequently impossible

Example

```
printf("1");
if ((pid = fork()) == 0) {
   printf("3");
} else {
   if ((pid = fork()) == 0)
      printf("4");
   else {
      wait(NULL);
      printf("2");
   }
}
printf("5");
```

Possible output

```
13425 14325 13245 14235 13542
```

Concurrent Programming is Hard!

- Classical problems of concurrent programming:
  - *Races*: outcome depends on arbitrary scheduling decisions elsewhere in the system
  - *Deadlock*: improper resource allocation prevents forward progress
  - *Starvation* / *Fairness*: external events and/or system scheduling decisions can prevent sub-task progress
- Many aspects of concurrent programming are beyond the scope of this course.
  - CPSC 375: High Performance Computing, Spring 2026

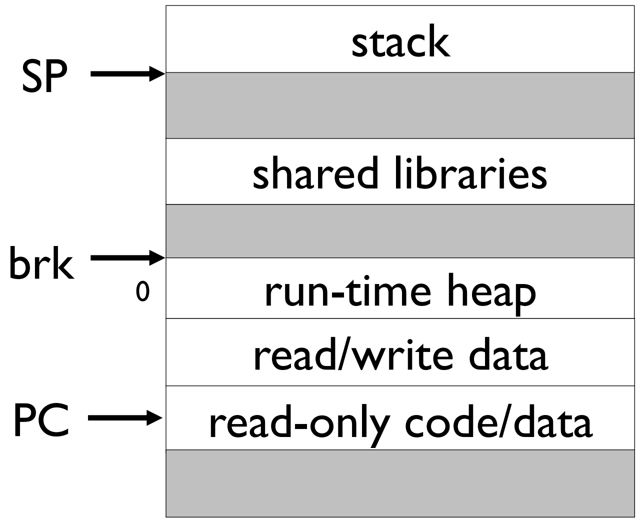
Traditional View of a Process

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

Process context

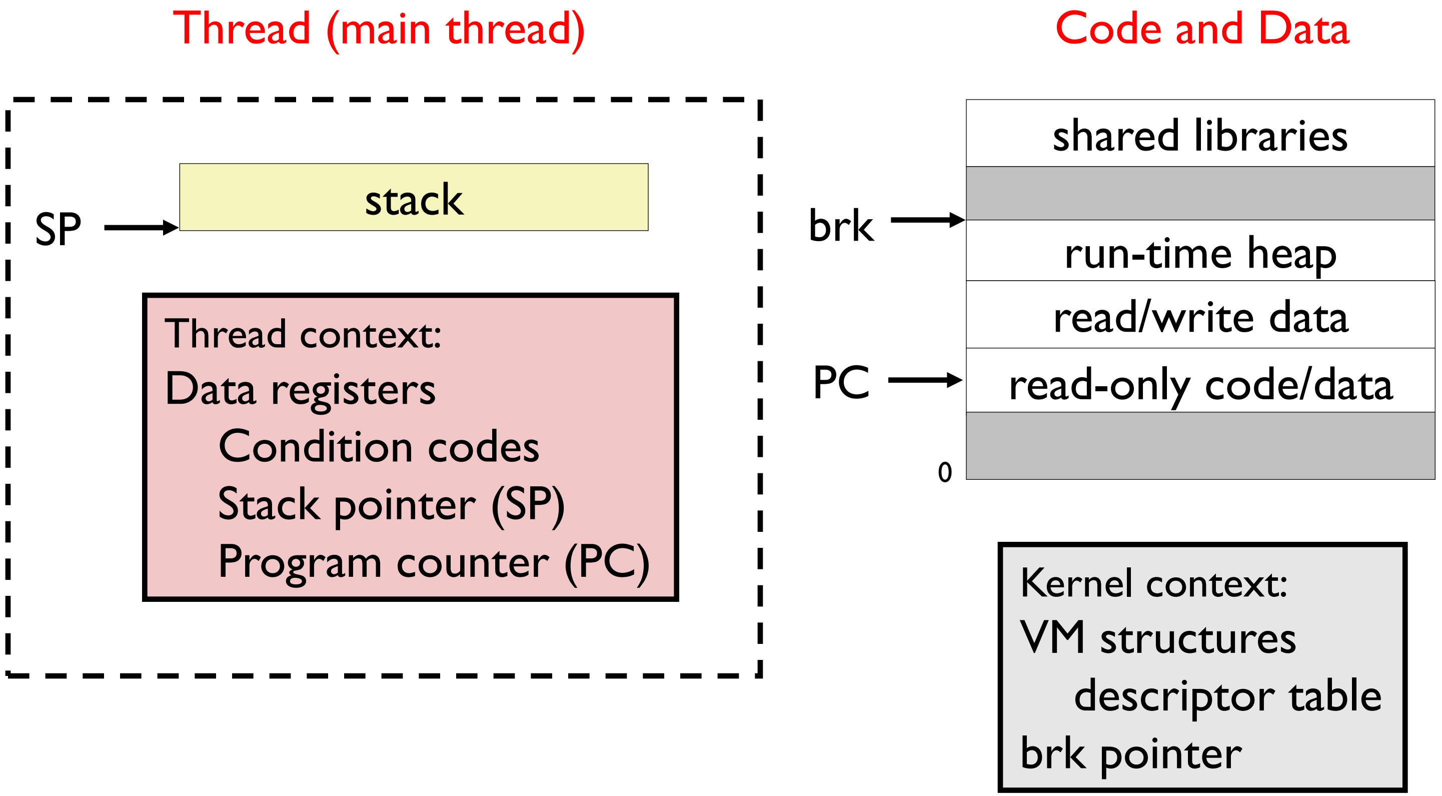
Program context:
    Data registers
    Condition codes
    Stack pointer (SP)
    Program counter (PC)
Kernel context:
VM structures
    Descriptor table
brk pointer

Code, data, and stack



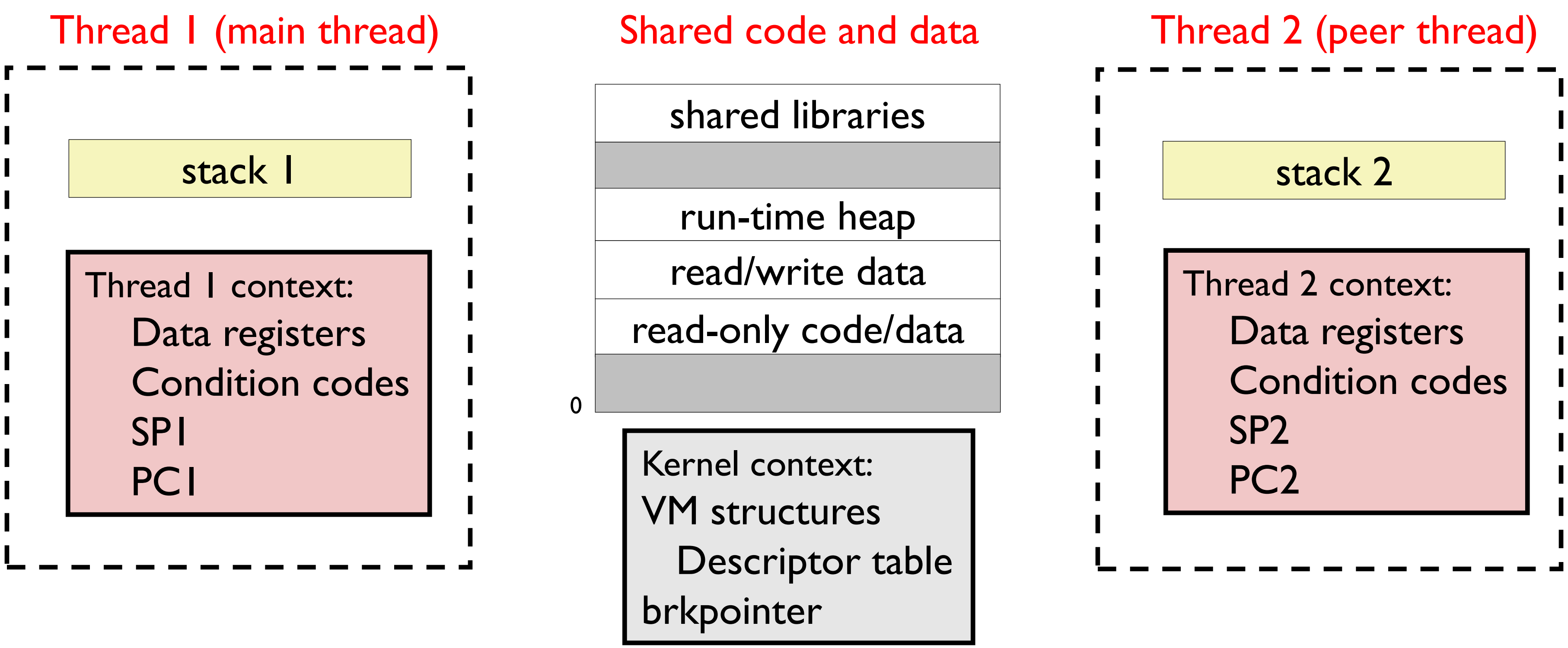
Alternate View of a Process

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

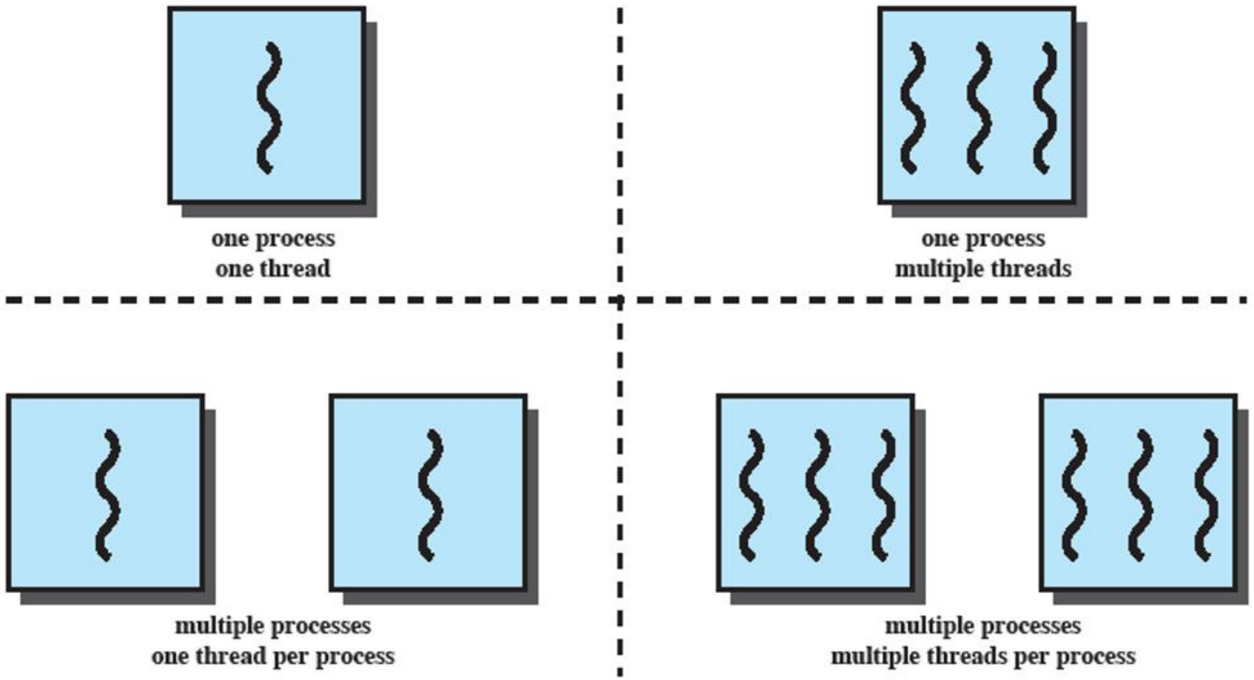


A Process with Multiple Threads

- 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
  - Each thread has its own thread id (TID)



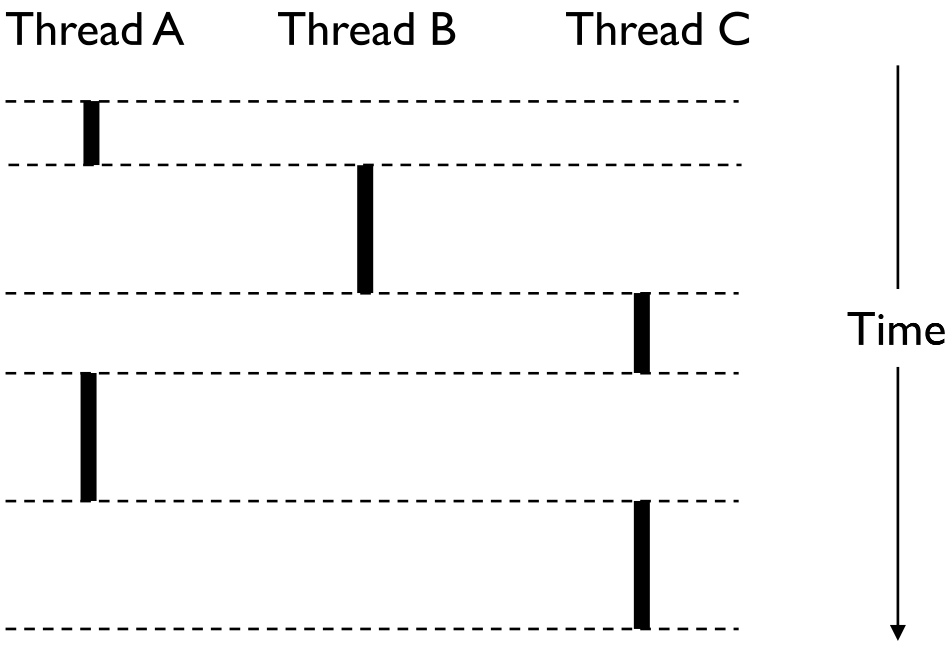
Processes and Threads



Thread Execution

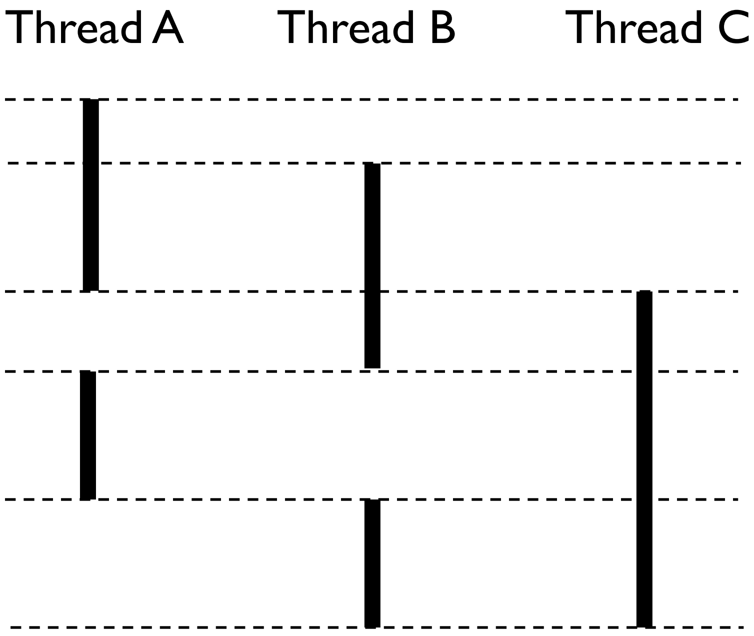
Single Core Processor

- Simulate concurrency by *time slicing*



Multi-Core Processor

- Can have true concurrency



Run 3 threads on 2 cores

How do you write code for multi-core processors?

POSIX Threads

- A standard for Unix-like operating systems known as *Pthreads*
- A library that can be linked with C programs.
- Specifies an application programming interface (API) for multi-threaded programming.

Pthreads Interface

- creating and *reaping* threads
  - `pthread_create()`
  - `pthread_join()`
- determining thread ID
  - `pthread_self()`
- terminating threads
  - `pthread_cancel()`
  - `pthread_exit()`
  - `exit()` [terminates all threads]
- *synchronizing* access to shared variables
  - `pthread_mutex_init`
  - `pthread_mutex_[un]lock`
  - `pthread_cond_init`
  - `pthread_cond_[timed]wait`

A Simple Example

Compile with:

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

link to *Pthreads* library

Run with:

$ ./pth_hello *nthreads*

Pointers to Functions in C

- C doesn't require that pointers point only to *data*; it's also possible to have pointers to *functions*.
- Functions occupy memory locations, so every function has an address.
- We can use function pointers in much the same way we use pointers to data.
- Passing a function pointer as an argument is fairly common.

Function Pointers as Arguments

- A function named `integrate` that integrates a mathematical function `f` can be made as general as possible by passing `f` as an argument.

```
double integrate(double (*f)(double,double),
                 double a, double b);
```

Here, the parentheses around `*f` indicate that `f` is a pointer to a function.

Function Pointers as Arguments

- A call of `integrate` that integrates the `sin` (sine) function from 0 to $\pi/2$:

```
result = integrate(sin, 0.0, PI/2);
```

- Within the body of `integrate`, we can call the function that `f` points to:

```
y = (*f)(a, b);
```

Starting the Threads

For each thread, call

```
int pthread_create(
      pthread_t *thread,        /* thread handler */
      pthread_attr_t *attr, /* thread attributes */
      void * (*start_routine) (void *), /* thread function*/
      void *arg); /* arguments for thread function*/
```

Starting the Threads

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

Allocate before calling.

We won't be using it, so we just pass NULL.

The function (task) that the thread is to run.

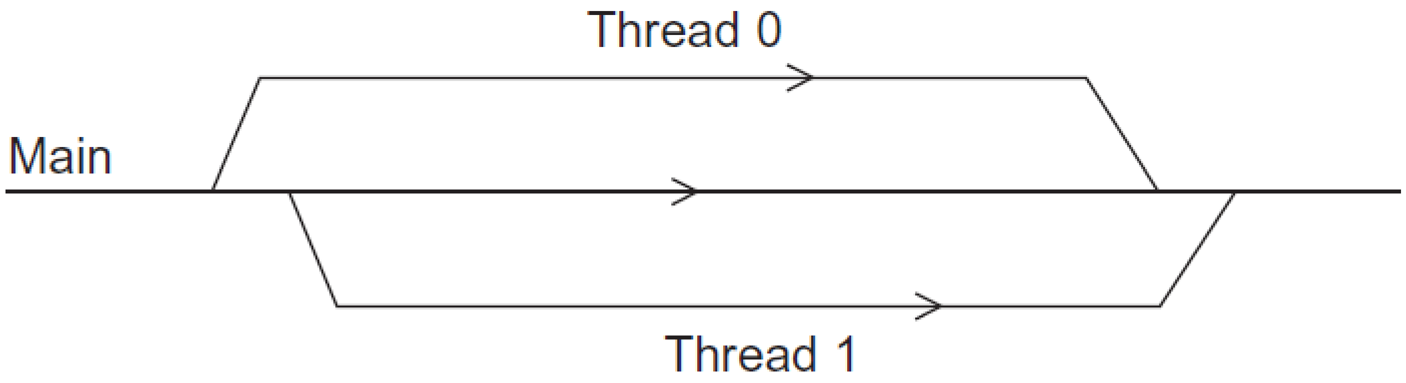
Pointer to the argument that should be passed to the function *start_routine*.

Function started by `pthread_create`

`void *`*start_routine* `(void *`*arg*`);`

- `void *` can be cast to any pointer type in C.
- *arg* can point to a list containing one or more values needed by *start_routine*.

Running Threads



Main thread forks and *joins* two threads.

Stopping Threads

- We call the function `pthread_join` once for each thread.
- A single call to `pthread_join` will wait for the thread associated with the `pthread_t` object to complete.

Demo

- `pth_hello.c` – Printing "`hello`" using multiple threads s
  :

  ```
  ./pth_hello 10
  ```

- `pth_add.c` – Computing the sum of the first *N* positive integers using multiple threads:

  ```
  ./pth_add 100 8
  ```