Threads

Adapted by Joseph Lunderville
from slides by Dr. Brian Fraser
and course material by Dr. Steve Ko

Topics

- How do “threads of execution” differ from processes?
  - How do two threads share memory space?
- How do we start and work with threads?

What is a Thread?

- *A **thread** is a unit of execution*
  - Like a process, in terms of scheduling…
  - But lighter weight, from the kernel's perspective: everything else shared within a process
  - Sometimes called a "lightweight process"
- ***Main thread***
  - A process always has at least one thread, called a **main thread:** this is where `main()` is initially called

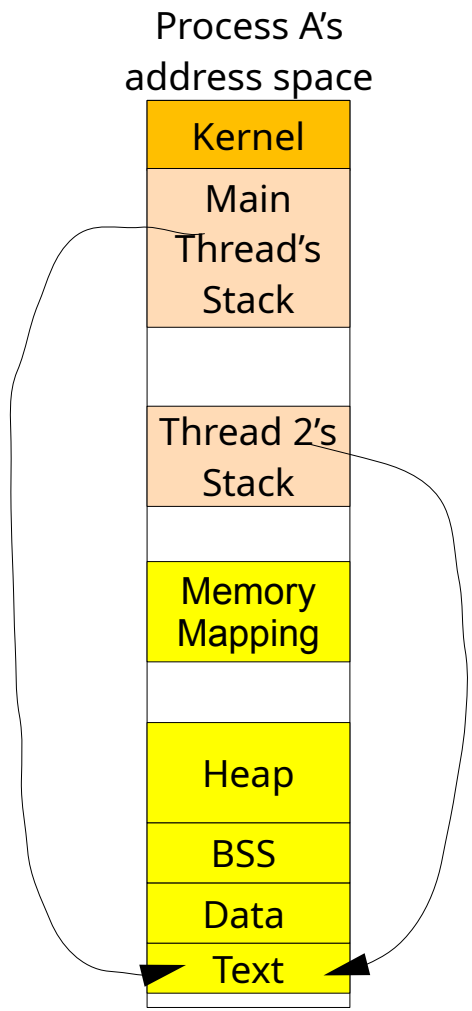
Details

- Can find more info in OSTEP book (more depth than we require)
  - Chapter 26 Concurrency: An Introduction
    https://pages.cs.wisc.edu/~remzi/OSTEP/threads-intro.pdf
  - Chapter 27 Interlude: Thread API
    https://pages.cs.wisc.edu/~remzi/OSTEP/threads-api.pdf

Threads vs. Processes

Thread vs Process

- If **threads** and **processes** both execute concurrently, how are they different?
  - Processes have independent access to kernel resources
  - Most obviously: *separate (virtual) address spaces*
  - fork() creates a child *process* with its own address space
  - Threads share the same address space, and data changed by other threads is immediately visible
- Each thread gets its own execution state
  - Stack, registers: local variables
  - Program counter: which function
  - Special “thread-local” data, e.g. `errno`



Threads and Processes - Tradeoffs

- Benefit of a thread
  - Threads in a process share the same address space, data sharing is convenient and usually fast
    - Any thread can read from or write to a global variable
    - Pointers point to the same code and data structures, and are valid to share
  - Usually faster to create (lighter weight)
- Benefit of a process
  - Memory isolation (just one, but it’s big)

POSIX Threads

man pthreads

- *man pthreads*
  - Description (what's shared and what's not!)
  - Return values (and `errno`)
  - Thread IDs
  - Thread-safe functions

```
pthreads(7)              Miscellaneous Information Manual              pthreads(7)

NAME
       pthreads - POSIX threads

DESCRIPTION
       POSIX.1 specifies a set of interfaces (functions, header files) for
       threaded programming commonly known as POSIX threads, or Pthreads.  A sin-
       gle process can contain multiple threads, all of which are executing the
       same program.  These threads share the same global memory (data and heap
       segments), but each thread has its own stack (automatic variables).
```

Common Functions

- *`pthread_create()` (read along on `man` page)*
  - `pthread_t`: this is the type used for thread IDs
  - Function pointer to a thread function
    - Unlike with `fork()`, we specify which function to execute
  - `void * arg`, passed in and returned
    - `void *` can be cast to any pointer type (or `intptr_t` in a pinch!)
    - Use a struct to pass multiple arguments
  - `pthread_attr_t` specifies various attributes of the new thread
- *`pthread_exit()`*
  - Terminates the calling thread
  - Happens implicitly if thread function returns

```
return NULL;  // Leaving thread function!
```

Common Functions

- `pthread_self()`
  - Returns the caller's thread id
  - Hint: use `gettid()` to be able to print Linux thread ID as a number
- `pthread_join()`
  - Waits until that thread terminates
  - Receive thread return value via `void * * retval`
- `pthread_detach()`
  - Lets the calling thread just run (why don’t we have the zombie problem?)
  - Can use this when return value not used

Audience Participation - pthreads

- Each thread gets its own...

a) Stack

b) Heap

c) Text (Code segment)

d) stdout

Audience Participation - `pthread_create()`

- Which of the following is true about `pthread_create()`?

a) It creates a new process running the provided thread start function.

b) It passes nothing to the function (void).

c) It waits until the spawned thread finishes.

d) It stores the thread_id for later user.

Activity - pthreads

- *Write a program that creates a thread (15m)*
  - Main thread
    - create another thread
    - wait until thread terminates
    - print out the return value
  - New thread
    - accept a string as its argument
    - print out the argument and its own ID (use gettid())
    - return the length of the received string
  - Compile with `-pthread` compiler option
    - i.e., `clang -pthread example.c`

Start simple! Make a thread and print “hi”!

Thread function can return a number:

```
return (void*) 42;
```

main() can get the number:

```
void* ret_val = 0;
pthread_join(...);
printf(“%ul”, (uint64_t) ret_val);
```

Data Race

Data Race Activity

- *Write a program (10m)*
  - Declare a global variable

    ```
    int count = 0;
    ```

  - Create two threads (besides the main thread)
    - Each new thread adds 1 to `count`, 10 million times
  - Main thread `pthread_join()`'s the two threads
  - Print `count`
- **Run multiple times, observe output**
  - Extra credit: compare compiling with `-O0` and `-O2`

Determinism

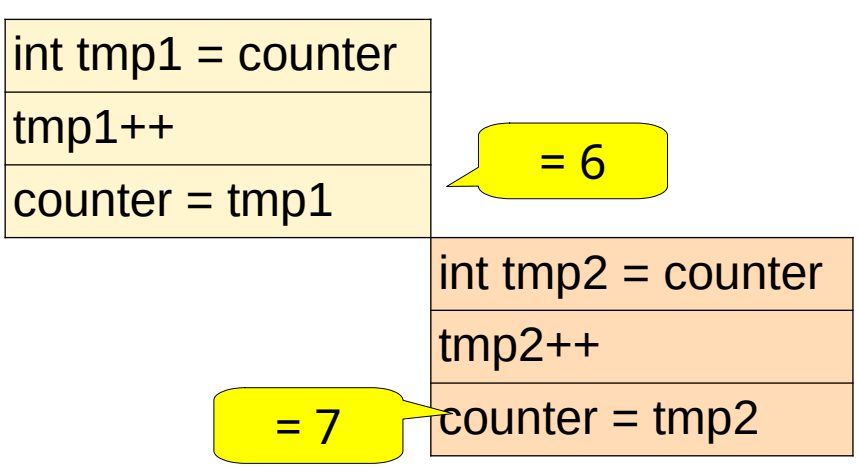
- *Deterministic behaviour*
  - Program output is the same every time, given the same input
  - Nice! Makes testing and debugging easier
- *Races*
  - Order of code execution between multiple threads and processes is not always the same
    - This is just one source of non-determinism! Determinism and repeatability can be subtle
  - When timing variation leads to visible non-determinism, that's a **race condition**
  - Term usually connotes a bug
  - Sometimes non-determinism is designed in, e.g. real-time and distributed systems – algorithm design is important here!

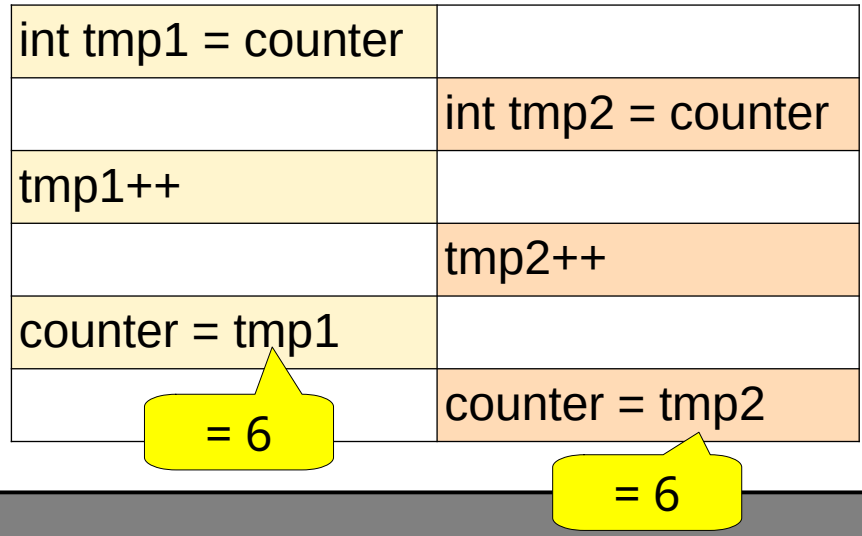
Data Race Problem

- *Consider the statement, `++count`*
  - It seems like `++count` is one operation
  - In reality,

```
int tmpRegister = counter; // Load from memory
tmpRegister++;             // Change value
counter = tmpRegister;     // Store value to memory
```

  - What happen if this runs on 2 threads? (assume counter = 5)





Race Condition

- ***Data race:***
  - This is called the data race problem
  - Different threads **race** to update data and overwrite each other's result
  - Often indicates a bug, but “data race” is a descriptive term: can be benign
- ***Race condition:***
  - The correctness of a program depends on the timing and/or order of operations
  - More general in that it *does not identify a mechanism,* but more specific in that it *identifies a problem*
- *More about the distinction*
  - Interesting and important if you are doing threaded or distributed programming at systems level, but beyond course scope for now

    https://blog.regehr.org/archives/490

Summary

- *Threads*
  - Lightweight processes that share a memory space
  - Always have main thread
- *pthreads*
  - POSIX library/API for threads
  - `pthread_create()`, `pthread_join()`, …
- *Data Race*
  - When two threads may access the same data at the same time