Thread Abstraction & Concurrency

2023/24 COMP3230B

Concurrency

- "Why are we studying this in OS class?"
  - "History" is the one-word answer.
  - Simply put, the OS was the first concurrent program, and thus most of these techniques arose due to the need for them within the OS. Later, as multi-threaded programs became popular, application programmers also had to consider such things.

Contents

- What are threads?
  - Why multithreading?
- POSIX Threads (Pthreads)
- Concurrency Issue
  - Race Condition
  - Critical Section
  - Mutual Exclusion
  - Synchronization

Related Learning Outcomes

- ILO 2a - explain how OS manages processes/threads
- ILO 2c - explain the underlying causes of concurrency issues
- ILO 4 - demonstrate knowledge in applying system software and tools available in modern operating system for software development

Readings & References

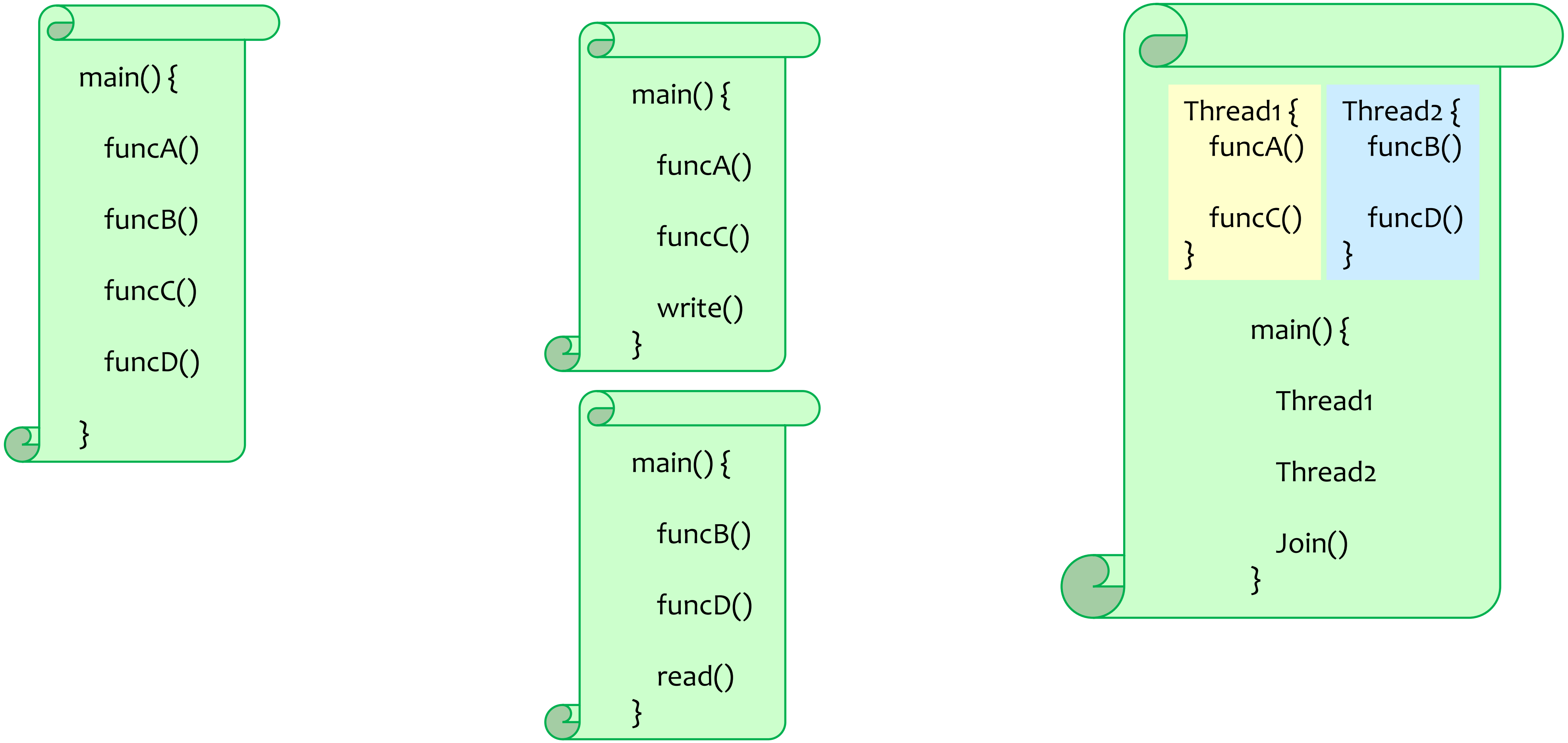
- Required Reading
  - Chapter 26 – **Concurrency: An Introduction**
    - http://pages.cs.wisc.edu/~remzi/OSTEP/threads-intro.pdf
  - Chapter 27 – **Interlude: Thread API**
    - http://pages.cs.wisc.edu/~remzi/OSTEP/threads-api.pdf

What are threads?

- Thread of execution
  - A **sequence of instructions** that performs a task (within the application process)
  - We can view traditional process as a process with one thread of execution

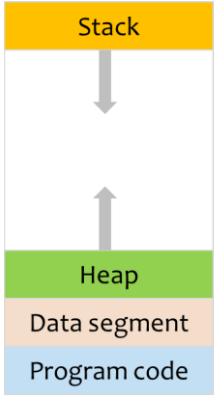
- Multithreaded process
  - A process has multiple threads of execution
  - A thread is an entity within a process
  - Multiple threads within a process
    - **can execute concurrently**
    - **share the process's address space** and other global info

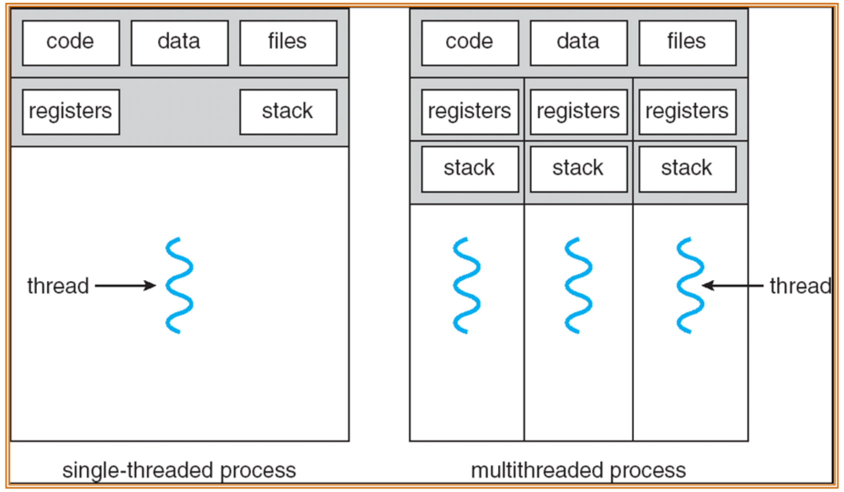
What are threads?

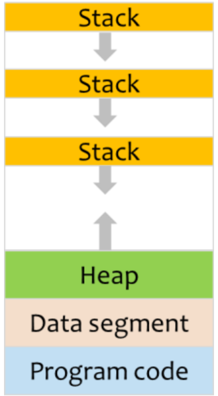


What are threads?

- To support multiple threads within a process, the system must provide each thread with its own
  - program counter; private set of registers
  - stack (& kernel stack)
  - and its own control block – **Thread Control Block** (TCB)
- Like process, each thread transits among a series of discrete thread states: new, running, ready, blocked, and terminated







Why Multithreading?

- Make inherently parallel tasks simpler to express in code
- For performance
  - Less set up is needed, it takes less time to create/terminate a thread than a process
  - A process with multiple threads could continue running even if one of its threads is blocked
    - A traditional process will be moved to blocked queue if it calls a blocking system call
  - Can make use of underlying multicores
    - One process with multiple threads; each thread can run on a core
  - Less overhead in switching between threads of the same process
    - thread switching as compared to process switching

POSIX Threads - Pthreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization

- Why use POSIX threads?
  - Common in UNIX operating systems (Solaris, Linux, Mac OS X)

- API specifies behavior of the thread library, implementation is up to development of the library
  - POSIX states that processor registers, stack and signal mask are maintained individually for each thread
  - POSIX specifies how operating systems should deliver signals to pthreads in addition to specifying several thread-cancellation modes

Thread Creation

- pthread_create() creates a new thread and makes it executable.
  - Typically, threads are first created from within main() inside a single process.

```
#include <pthread.h>
int pthread_create(pthread_t * thread,
                   const pthread_attr_t attr,
                   void *(*thrfunc)(void *),
                   void *args);
```

- 4 arguments:
  - thread: a pointer to a structure of type pthread_t, which becomes the handler of a thread
    - we can pass this handler to various thread operations
  - attr: used to set thread attributes
    - in most case, **set it to NULL** to use the default setting
  - thrfunc: a function pointer, which **points to the C function** that the thread will start executing once it is created
  - args: a single argument to be passed to the thrfunc function.

Thread Termination

- pthread_exit() terminates the execution of the calling thread
  - Typically, a thread calls this after it has completed its work

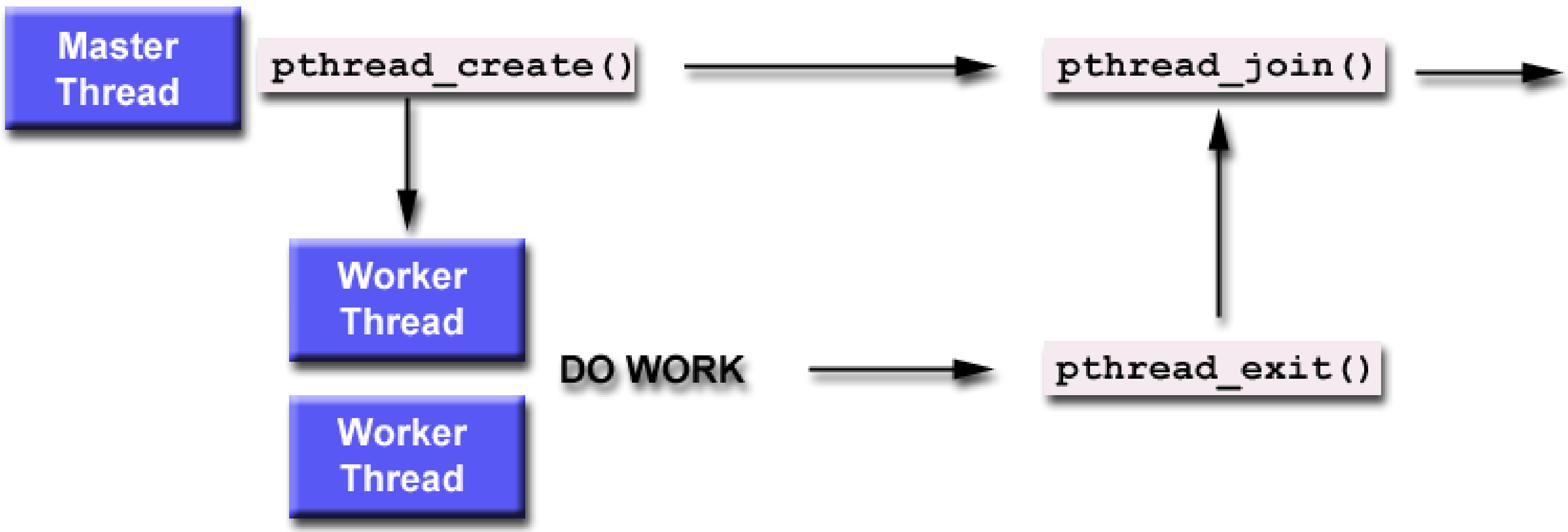
```
#include <pthread.h>
void pthread_exit (void *retval)
```

- retval: return value of the thread
  - It can be retrieved by another thread using pthread_join()

- It does not close files; any files opened inside the thread will remain open after the thread is terminated

Waiting for a thread to terminate

- “Joining” is one way for a thread (especially main/master thread) to **wait for** other threads to exit. For example:



Thread Join

- The pthread_join() **blocks the calling thread until** the specific thread terminates

```
#include <pthread.h>
int pthread_join (pthread_t thread_id,
                  void ** value_ptr);
```

- thread_id: specify which thread in which this calling thread will wait for

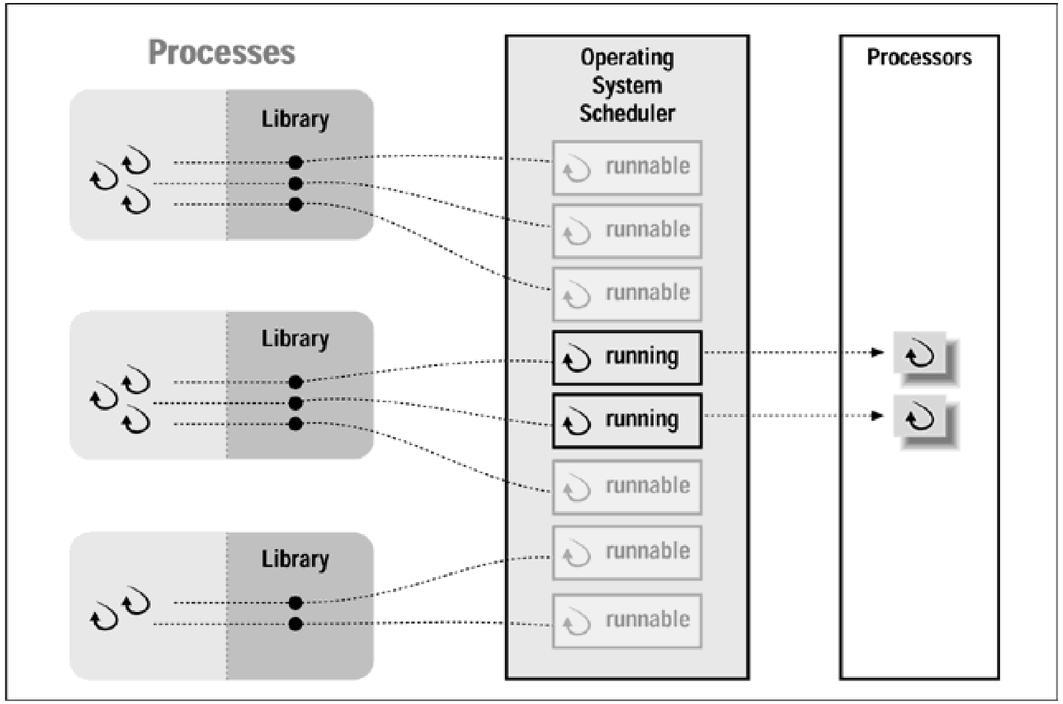
- value_ptr: a return value you expect to get back from the target thread (via a pointer to void pointer)
  - If you don't care, set it to NULL

An Example

```
#include <stdio.h>
#include <pthread.h>

void *func1 (void *arg){
   int x = *((int*)arg);
   printf("The integer passed in is %d\n", x);
   printf("Thread: Process id is %d\n", (int)getpid());
   pthread_exit(NULL);
}

int main() {
   pthread_t thread_id;
   int x = 1;

   printf("Main process: Process id is %d\n", (int)getpid());

   pthread_create(&thread_id, NULL, func1,(void*)&x);
   pthread_join(thread_id, NULL);
   return 0;
}
```

Pthreads in Linux

- In Linux, each pthread is implemented as standard process – lightweight process (LWP)
  - All LWPs in the same multithreaded application share the memory address space, the open files, global variables, heap, …
  - To enable threading, Linux uses the clone() instead of fork()
    - Clone accepts arguments that specify which resources to share with the child task
- It means the scheduler does not differentiate between a thread and a process



Concurrency

Concurrency Issue

- Suppose the value of x was originally 10, what will be the outcome after two threads executed the instruction?

Thread 1	Thread 2
`x++;`	`x++;`

- It can be 11 or 12
- Why? And how can we guarantee to always get 12?

Reasons

- Multiprocessor or multi cores
  - The 3 instructions can be executed by 2 cores at the same time

- Single core

```
Thread 1                        Thread 2

read x to reg

inc reg            TS
                          read x to reg

                          inc reg

                   TS     store reg to x

store reg to x
```

TS – thread switch

Concurrency Issue

- The main issue is **uncontrolled scheduling**
  - When will a thread be executing is not predictable
    - We cannot predict at what time and for how long a thread is being schedule to run

- **Race Condition**
  - We called the scenario that **several threads** access and **modify** a **shared data item concurrently** and the outcome of the execution depends on the particular order (**a race**) in which the accesses took place
  - This results in non-deterministic computation, where it is ***not known what the result*** will be and it is *indeed likely to be different across runs*

Critical Sections

- For multithreaded program, most code is safe to run concurrently
  - When not accessing and **modifying** shared data

- Blocks of code where a particular shared data is modified must be guarded

- We called these blocks of code – **Critical Sections**
  - We would like to have only one thread be in **its** critical section accessing the specific protected shared data **at one time**
    - it should execute as quickly as possible

Mutual Exclusion

- Therefore, shared data in critical sections must be accessed in mutually exclusive way
  - Only one thread is allowed access at one time
    - The winner thread is in-effect “locked” the shared data
  - Others must wait until the shared data is unlocked
- This is called serialized access or Mutual Exclusion

Atomic Operation

- One way to solve the race condition in "x++" is to make the operation to be ATOMIC –
  - The operation cannot be interrupted in the middle, and hardware can guarantee either has done or not done at all

- Unfortunately, critical sections may contain more complicated operations, e.g., linked list traversal
- Just having Atomic operations is not good enough

Synchronization

- Within a multithreaded process, there is another common interaction
  - One thread has to wait for another thread to work on some action before it continues
    - e.g., a worker has to wait for a task assigned by the boss
- The Crux
  - How to provide support for mutual exclusion? for synchronization? What support do we need from hardware and the OS?

Summary

- Threads have another name – light-weighted processes.
  - Each thread is a code fragment, within a process, that can be scheduled and executed independently
  - A thread has its own program counter, registers' contents, and stack, but it shares the same process's address space with other threads
  - Similar to process management, OS uses a thread control block to abstract a thread entity

- With multiple processes or threads, we have to face the concurrency issues
  - One of the responsibilities of OS is to provide mechanisms for processes/threads to synchronize and coordinate between processes/threads