Thread Programming in C

What are threads?

- A lightweight process within a program
- Independent stream of instructions executed simultaneously
- Execute concurrently within the process
- Allow efficient parallel execution
- Allow efficient utilization of multiprocessor systems

How threads and processes are similar

- Each has its own logical control flow
- Each can run concurrently
- Each is context switched

Thread Properties

- Exists within a process and uses the process resources
- Have own stacks, registers, code conditions
- Has its own independent flow of control
- Duplicates only the essential resources
- May share the process resources with other threads
- All threads within a process share same address space
- Changes made by one thread affect other threads

Process

- An execution path for one or more programs

Unit of resource ownership / Process

- a virtual address space to hold the process image
- protected access to processor or processors
- control of some resources (files, I/O devices...)

Unit of dispatching / Threads

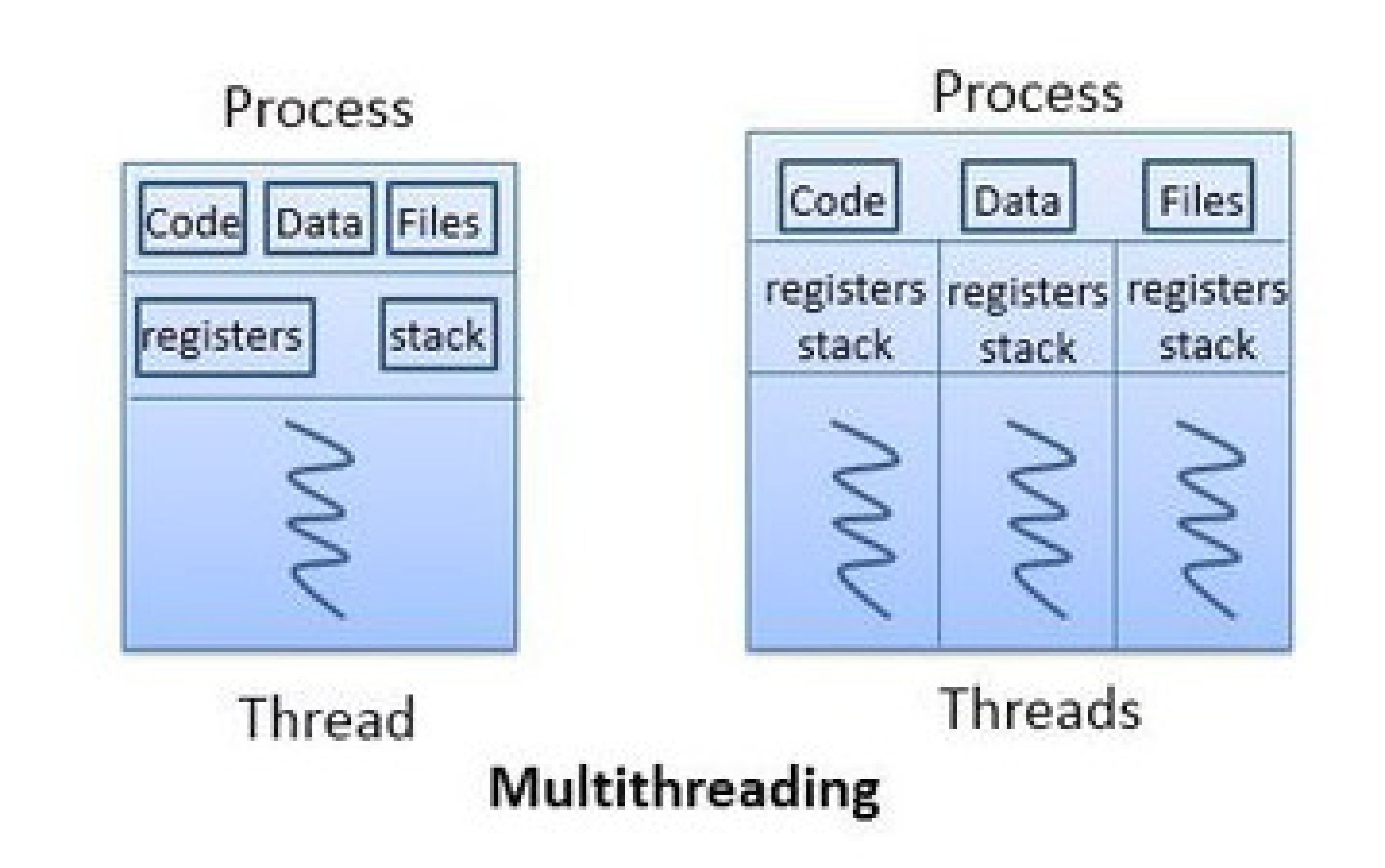
- process has an execution state and a dispatching priority
- saved context, per-thread stack and storage
- access to process resources

Advantages of Threads vs. Processes

- Fast thread creation and termination
- Fast context switch
- Fast communication across threads
- Share memory and other resources of the parent process
- Efficient communication and data sharing
- Faster context switching across threads

Disadvantages of Threads vs. Processes

- Threads are less robust than processes
- Threads have more synchronization problems
    - Mutex, Semaphores and Barriers can
- Execution may be interleaved with other processes
- Multiple threads can read and write same memory


Process
Code
Data
Files
registers
stack
Thread
Process
Code
Data
Files
registers
stack
registers
stack
registers
stack
Threads
Multithreading

Single Threading

- There is only one thread of execution
- All code is executed sequentially in one thread
- No concept of concurrency
- The main() function runs in this single thread
- When OS does not support multi-threading concept

Multi-Threading

- Allows multiple threads of execution within a program
- Each thread has its own flow of control and stack
- Threads execute individually and concurrently

Add GCC Options

Go to the Explorer

Open .vscode folder

Open tasks.json

Add -lpthread option to gcc command

```
args": [
   "-fdiagnostics-color=always",
   "-g",
   "${file}",
   "-o",
   "${fileDirname}/${fileBasenameNoExtension}",
   "-lm",
   "-lpthread"
]
```

Multi-threading Benefits

- Improve application responsiveness
- Many independent activities defined as a thread
- Use multiprocessors more efficiently and effectively
- Numerical algorithms and applications can run faster
- Improve program structure
- More adaptive to variations in user demands
- Use fewer system resources
- Low costs for creating and maintaining state information

Threads

- Serving multiple clients
- Background tasks
- Parallel computation
- Efficient utilization of multiprocessor
- Execute truly concurrently on different CPUs
- Managed in POSIX systems which provides
    - thread creation
    - synchronization
    - other functionality

Create a thread

- Creates a new thread and makes it executable
- Pointer to store the thread identifier
- Attributes for thread creation (`NULL` for default)
- Function to be executed by the thread
- Argument passed to `start_routine`

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

Terminate the thread

- Thread returns from its starting routine
- Thread calls the pthread_exit subroutine
- Canceled by another thread
- Entire process ended using exec() or exit()
- Main finished before pthread_exit() call
- Value to be made available to any successful

```
void pthread_exit(void *value_ptr);
```

Suspend execution

- Waits for a thread to terminate
- Detaches the thread
- Returns the threads exit status, unless NULL
- Identifier of the thread to be joined
- Pointer to store the exit status of thread

```
int pthread_join(pthread_t thread, void **value_ptr);
```

C Programming

```c
#include <pthreads.h>

void* thread1_func();
void* thread2_func();

int main() {
  pthread_t t1, t2;

  // launch threads
  pthread_create(&t1, NULL, thread1_func, NULL);
  pthread_create(&t2, NULL, thread2_func, NULL);

  return 0;
}

void* thread1_func() { /* thread 1 tasks */ }
void* thread2_func() { /* thread 2 tasks */ }
```

Applications of Threads

Concurrent Server

- In a multi-threaded server, two threads will be created to process two requests received by a web server simultaneously.

Taking Advantage of Multiple CPUs

- If a program uses multiple threads, the OS will schedule a different thread in each CPU.

Interactive Applications

- Threads simplify the implementation of interactive applications that require multiple simultaneous activities.

Problems with Multiple Threads

- Failing to protect shared variables
- Relying on persistent state across multiple function calls
- Returning pointer to `static` variable
- Calling threads function causes unstable shared variable
- Race (Data Race) when program correctness depends on one thread reaching point x before another thread reaches point y.
- Deadlock when two competing processes are waiting for each other to finish.
- Starvation when a process never gains accesses to resources.
- Livelock when two threads are dependent on each other signals and respond to each others signal.

Mutex

- A mutual exclusion lock
- Only one thread can hold the lock
- Protects data or other resources from access

```
// Initialize a mutex.
int pthread_mutex_init(pthread_mutex_t *mutex,
 const pthread_mutexattr_t *attr);

// Lock a mutex.
int pthread_mutex_lock(pthread_mutex_t *mutex);

// Unlock a mutex.
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Responsibility

- Prevent Race Conditions
    - Avoid outcomes dependent on order of access
- Ensure Data Integrity and Consistency
    - Avoid corruption of data or unexpected behavior
- Concurrency Control
    - Safely shared data structures without conflicts
- Priority Inheritance
    - Ensures higher-priority do not remain blocked
- Developer Responsibility
    - Ensure the robustness of concurrent applications
- Resource Management
    - kernel resources managed to avoid locks and leaks

Responsibility

```c
#include <pthread.h>
pthread_mutex_t count_mutex;
long long count;

void increment_count() {
    pthread_mutex_lock(&count_mutex);
    count = count + 1;
    pthread_mutex_unlock(&count_mutex);
}

long long get_count() {
    long long c;

    pthread_mutex_lock(&count_mutex);
    c = count;
    pthread_mutex_unlock(&count_mutex);
    return (c);
}
```

Questions?