POSIX Threads (pthreads)

Chen Chen

chenchen@shanghaitech.edu.cn

What is pthreads?

- POSIX Threads, usually referred to as pthreads, is **an execution model** that exists independently from a language, as well as **a parallel execution model**. It allows a program to control multiple different flows of work that overlap in time. Each flow of work is referred to as a thread, and creation and control over these flows is achieved by making calls to the **POSIX Threads API**.
- Defined by POSIX, implemented by different operating systems (Linux, BSD…), a standard.
- The pthreads library on different platforms are the user space interface of multi-threading implementations.

Sample usage

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
void worker(int* tid) {
    printf("Hello from thread %d\n", *tid);
}

int main () {
    int num_threads = 10;
    int* tids = (int*)malloc(sizeof(int)*num_threads);
    pthread_t* threads = (pthread_t*)malloc(sizeof(pthread_t)*num_threads);
    for (int i=0; i<num_threads; i++) {
        tids[i] = i;
        pthread_create(&threads[i], NULL, (void*)worker, &tids[i]);
    }
    return 0;
}
```

```
Hello from thread 0
Hello from thread 1
Hello from thread 2
Hello from thread 3
Hello from thread 4
Hello from thread 5
Hello from thread 6
Hello from thread 7
Hello from thread 8
Hello from thread 9
```

fork() and pthread_create()

- They both create new “context of execution” (or new task)
- fork():
  - Creates a **new child process** which is a copy of current process.
  - The new process is a child node of parent process in the process tree.
  - Their context is generally the same and dedicated.
- pthread_create():
  - creates a **new thread** within the current process.
  - They could be executed in parallel with the context of current process, and the memory address space are shared between threads.
  - Each thread has its own context (the stack and the heap), but everything is still in the processes’ memory space).

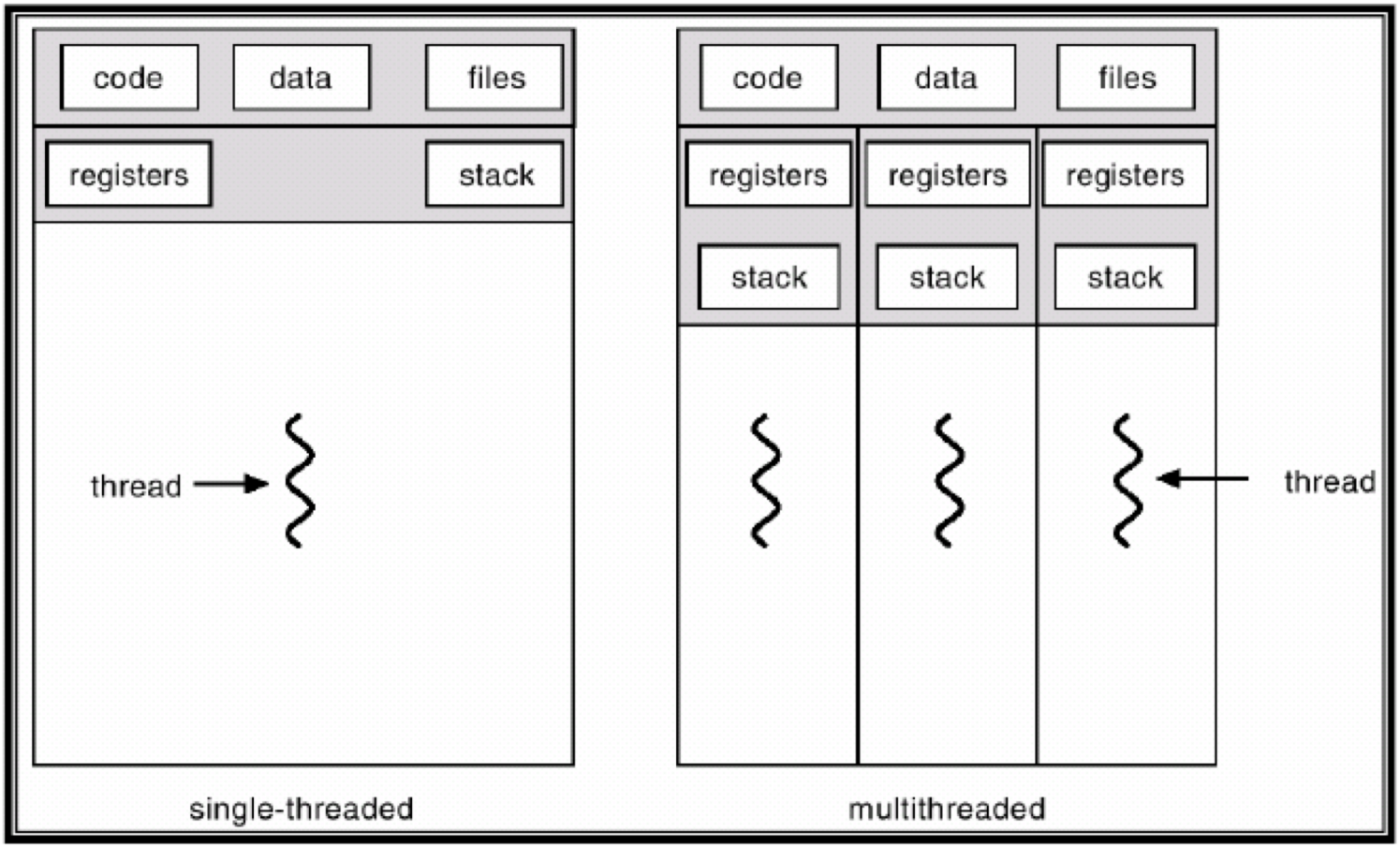
Process and Thread

- They are both “context of execution”, but they apply to different scenarios.
- Processes
  - Multi-tasking, concurrency.
  - Different processes are highly isolated.
  - Different memory address spaces, different page tables, different PIDs, different file descriptors (and more).
  - Sometimes parent and child processes may share the page table, and the child may duplicate the table when make changes (COW).
  - Process synchronizations are highly dependent on the kernel. Maybe use pipe, kernel semaphore, MPI and other methods.

Process and Thread

- Threads
  - Multi-processing, parallelism.
  - Different threads have different execution context, but still within the context of the process.
  - The PID of different threads in modern Linux are the same. But in Linux system with ancient libc, the PID may be different. It highly depends on the implementation of POSIX threads.
  - The page table, memory address space, file descriptors and many other things are shared. The OS won't give protection on threads' memory isolation. The isolation is handled by the process itself.
  - Threads synchronized with shared memory, or the built-in synchronization APIs provided by pthreads.

Process and Thread



Process and Thread

- Process is heavier, thread is lighter, as fork() introduces overhead in memory allocation and process management. It also introduces barriers in synchronization.
- The underlying implementation of threads is the clone() function. Since glibc 2.3.3, the fork() function won't invoke the kernel's fork() system call. It is provided as part of the NPTL threading implementation invokes clone() function.

What's the output?

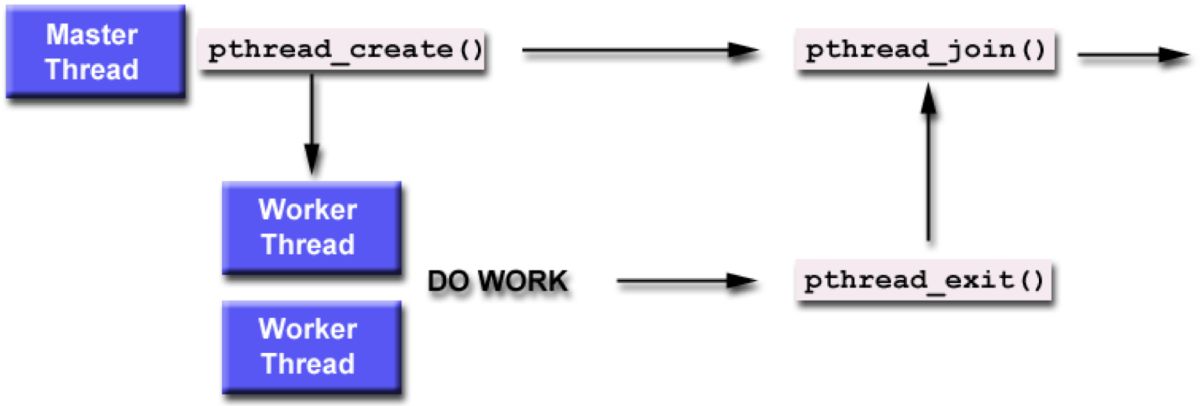
```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void worker(int* tid) {
    if (*tid == 8) {
        pid_t p = fork();
        if (p == 0) {
            int i = 0;
            while (i++<2147483647) {}
            char* const argv[] = { "/bin/ls", "/opt" };
            execv("/bin/ls", argv);
        }
    }
    printf("Hello from thread %d\n", *tid);
}

int main () {
    int num_threads = 10;
    int* tids = (int*)malloc(sizeof(int)*num_threads);
    pthread_t* threads = (pthread_t*)malloc(sizeof(pthread_t)*num_threads);
    for (int i=0; i<num_threads; i++) {
        tids[i] = i;
        pthread_create(&threads[i], NULL, (void*)&worker, &tids[i]);
    }
    while (1) {}
    return 0;
}
```

Thread joining

- "Joining" is one way to accomplish synchronization between threads.
- The pthread_join() blocks the calling thread until the specified thread terminates.



Synchronization

- Mutex
  - int pthread_mutex_init(pthread_mutex_t *mp, const pthread_mutexattr_t *attr): Initialize a new mutex
  - pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER: Declare a new mutex lock
  - int pthread_mutex_lock(pthread_mutex_t *mp): Lock a mutex (blocking)
  - int pthread_mutex_trylock(pthread_mutex_t *mp): Lock a mutex (non-blocking)
  - int pthread_mutex_unlock(pthread_mutex_t *mp): Unlock a mutex
  - int pthread_mutex_destroy(pthread_mutex_t *mp): Destroy a mutex
- Condition Variables
  - int pthread_cond_init(pthread_cond_t *cond,pthread_condattr_t *cond_attr);
  - int pthread_cond_wait(pthread_cond_t *cond,pthread_mutex_t *mutex);
  - int pthread_cond_timewait(pthread_cond_t *cond,pthread_mutex *mutex,const timespec *abstime);
  - int pthread_cond_destroy(pthread_cond_t *cond);
  - int pthread_cond_signal(pthread_cond_t *cond);
  - int pthread_cond_broadcast(pthread_cond_t *cond);

More APIs…

- pthread_exit(): Terminate current thread
- pthread_kill(): Send a kill signal to the specified thread
- pthread_equal(): Compare two pthread_t
- …
- Read the manual when needed