Multithreading

Operating Systems II
Viktor Iakovlev (Victor Yacovlev)

Threads

- Thread (aka lightweight process) - task to be controlled by scheduler
- Each task has it’s own stack and processor state
- One process might have several threads

Main Thread

- Function `_start` starts execution in main thread.
- Main thread might start other threads
- Any thread can start new thread
- All threads are equal

exit

- System call `_exit` terminates the process: all threads
- System call `pthread_exit` terminates current thread only
- It is possible to terminate main thread but keep other threads working

Stack

Память процесса:

0x7fffffffffff - 8 Mb	**Стек** ↓
0x03106000	↑ **Куча**
	.bss
	.data
	.rodata
	.text
0x00400000 0x00000000	**Заполнено нулями**

Память потока:

0x7fffffffffff - 8 Mb	**Стек** ↓

0x7fffff7fffff - 8 Mb	**Стек** ↓

0x7ffffeffffff - 8 Mb	**Стек** ↓

Stack areas might be mixed with dynamic library areas

Stack

- Thread stack size has the same size as main thread by default
- POSIX allows to specify stack size for newly created threads
- Guard Page below the stack

- **No memory isolation: any thread is allowed to access any thread's stack memory**

Process v.s. Thread

- Just a task from system scheduler's point of view
- System call `sched_yield` affects current thread; in case of single-threaded programs - Main thread
- Process count limit for Linux - limit for maximum number of threads but not processes

Process v.s. Thread

Process

- Complete isolation from each other
- Process crash has no effect to other processes
- Processes might have different privileges
- Inter-Process Communication is complex in general

Thread

- Threads share the same memory address space
- Signal (in case of errors too) affect all threads
- All threads work within the same user
- Simple communication using common variables

Thread and Process Attributes

Common to All Threads

- Process ID
- Parent process ID
- Group ID/Session ID
- Associated Terminal
- UID/GID
- File Descriptors
- File Locks
- Signal Handlers
- umask, cwd, root dir
- Limits

Distintc

- Thread ID
- Thread Stack Memory
- Signal Handler Stack
- Scheduling Priority
- Signal Mask
  pthread_sigmask
- errno Value

<errno.h>

- «Global Variable» behaviour
- Implemented by macro in modern systems

```
/* The error code set by various library
functions.  */
extern int
  *__errno_location (void)
  __THROW __attribute_const__;
# define errno (*__errno_location ())
```

Threading API

- POSIX (Linux, *BSD, Mac)
  <pthread.h>
  pthread_create(...)

- WinAPI (Windows)
  <Windows.h>
  CreateThread(...)

WinAPI Threads

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES   lpThreadAttributes,
  SIZE_T                  dwStackSize,
  LPTHREAD_START_ROUTINE  lpStartAddress,
  LPVOID                   lpParameter,
  DWORD                   dwCreationFlags,
  LPDWORD                 lpThreadId
);
```

Minimum stack size - 64Kb

C++11 Threads

- Common interface for all platforms
  ```
  <thread>
  std::thread(...)
  ```

- Covers only common features

C11 Threads

- Common interface for all platforms

```
<threads.h>
void call_once(
        once_flag *flag,
        void (*func)(void)
        );
```

- **At present there is no any implementation**

Threading and fork

- All threads are equal → it is possible to call `fork()` from any thread
- Process creation implies memory copy, including stacks of all threads
- Newly created process **will have just one task**, that continnues execution of thread that called `fork()`

Threading and fork

- Might have locked resources → deadlock
- In case of using `fork`, it is required to call `pthread_atfork` to register a function that releases all possible resoures

Threading and fork + exec

- System call `exec` completely replaces address space by another program
- System call `fork` keeps only one task running
- Combination of `fork+exec` is safe in general

Signal Mask

(each thread has it's own)

Signlas to be processed →	0	0	0	0	0	1	0
Mask of [blocked] signals →	1	0	1	1	1	1	1
	SIGHUP	SIGINT	SIGQUIT	SIGILL	SIGABRT	SIGFPE	SIGKILL
	0	1	0	0	0	1	0

Process 1
`kill(SIGINT, PID)`

Process 2
`kill(SIGINT, PID)`

Process 3
`kill(SIGFPE, PID)`

Signals Processing

- Signal Mask is a **process** but not thread attribute
- Signal handlers are available for all threads
- Signal handlers are to be executed by **arbitary** thread
- It is possible to block signals handling for distinct threads (`pthread_sigmask`)

Thread Cancelation

- Avoid using this
- Tho ways to perform:
  - default: thread might be canceled at cancelation point (mostly system calls)
  - hard way: force cancelation eveon on code like this: `while (1) {}`.
    Threads can't be canceled immediately.

Thread Cancelation

- Function `pthread_cancel` "cancels" specified thread: sends a request to stop
- Thread might disallow these requests:

```
pthread_setcancelstate(
        PTHREAD_CANCEL_DISABLE)
```

Thread Cancelation

- Two ways of cancelation:
  - at cancelation point (`man 7 pthreads`)
  - by sending a signal

- Thread might specify cancelation type `pthread_setcancelype`

Thread Cancelation

- On thread canceled:
  - semaphores and mutexes might be locked
  - unclosed file descriptors
  - lost memory pointers
  - just important work not complete
- Avoid thread cancelation!

Thread-Local Storage

- Global variables are available for all threads
- Language keywords `_Thread_local` (C11) and `thread_local` (C++) make variables to be local to threads
- Values of TLS-variables are not isolated

Global Variables

- May lead to data race
- Might be changed indirectly via standard functions
- Many functions have marked as Thread-Safe: they are implemented with locks (with drawback of performance loss)
- Example: `getc` v.s. `getc_unlocked`

How to Avoid Data Race

- Use atomic values
  <stdatomic.h> (C11)

```
typedef _Atomic _Bool atomic_bool;
typedef _Atomic char atomic_char;
typedef _Atomic signed char atomic_schar;
typedef _Atomic unsigned char atomic_uchar;
typedef _Atomic short atomic_short;
typedef _Atomic unsigned short atomic_ushort;
typedef _Atomic int atomic_int;
```

It is required to use atomic operations

- Use locks: semaphores and mutexes

The Key Problem of Multithreading

- There are a lot of single-threaded programs accumulated due computes history
- The simpliest way make them parallel-safe is to add locks
- The simpliest way is not the most effective

Example: Python

- Thread start is `pthread_create` + initialization of internal structure `PyThreadState`
- Each Python Thread is a system Thread

but...

- All of them uses the single interpreter instance (implemented as `PyInterpreterState` structure)

Global Interpreter Lock (GIL)

Example: Python. How to Avoid GIL

- Do not use Python :)
- Use `multiprocessing` instead of `multithreading` in case to make program really parallel

In-Kernel Threads

- Goal: make Kernel Task to run in parallel
- All of them runs in Kernel address space
- Linux assigns virtual ProcessID to all in-kernel threads like processes
- Examples:
  - kswapd - manages swap
  - kworker - [in most cases] handles I/O operations

Giant Kernel Lock

- Global lock for the whole Kernel
- Processes might block another processes during system calls
- Problem origins from legacy device drivers and subsystems
- The problem firstly eliminated for BSD at 2009 (FreeBSD 8.0)
- The problem eliminated for Linux at 2011 году (Kernel version 2.6.39)

BeOS / HaikuOS

https://www.haiku-os.org/

- Operating System that have support for multiple-cores by design
- Each Kernel subsystem works in it's own thread