Threads

François Trahay



Execution context of a process

- Context: execution context + kernel context
- Address space: code, data and stack



Duplicating a process

- Fork creates a new process and duplicates
 - Context: execution context + kernel context
 - except for the a0 register (where the return value is stored)
 - On x86_64 architecture, this is the register rax
 - Address space: code, data and stack



Execution flows

- Execution flow ! = Resources
 - Execution flow (or thread) : execution context + stack
 - Resources: code, data, kernel context



Multithreaded process

- Several execution flows
- Shared resources



Creating a Pthread

- attr (in): attributes of the thread to be created
- **start_routine** (in): function to be executed once the thread is created
- arg (in): parameter to pass to the function
- thread (out): identifier of the created thread

Other Pthread functions

7

int pthread_exit(void* retval);

• Terminates the current thread with the return value retval

int pthread_join(pthread_t tid, void **retval);

• Wait for the tid thread to terminate and get its return value

Sharing data

- The memory space is shared between the threads, in particular
 - global variables
 - static local variables
 - the kernel context (file descriptors, streams, signals, etc.)
- Some other resources are not shared
 - local variables

Thread-safe source code

- thread-safe source code: gives a correct result when executed simultaneously by multiple threads:
 - No call to non *thread-safe* code
 - Protect access to shared data

Reentrant source code

- Reentrant source code: code whose result does not depend on a previous state
 - Do not maintain a persistent state between calls
 - example of a non-reentrant function: **fread** depends on the position of the stream cursor

TLS – Thread-Local Storage

- Global variable (or static local) specific to each thread
 - Example: errno
 - Declaring a TLS variable
 - o in C11: _Thread_local int variable = 0;

Synchronization

- Guarantee data consistency
 - Simultaneous access to a shared read / write variable
 - x++ is not atomic (consisting of load, update, store)
 - Simultaneous access to a set of shared variables
 - o example: a function swap(a, b) { tmp=a; a=b; b=tmp; }
- Several synchronization mechanisms exist
 - Mutex
 - Atomic Instructions
 - Conditions, semaphores, etc. (see Lecture #3)

Mutex

- Type: pthread_mutex_t
- Initialisation:
 - pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
 - int pthread_mutex_init(ptread_mutex_t *m, const pthread_mutexattr_t
 *attr);
- Usage:
 - int pthread_mutex_lock(pthread_mutex_t *mutex));
 - int pthread_mutex_trylock(pthread_mutex_t *mutex);
 - int pthread_mutex_unlock(pthread_mutex_t *mutex);
- Terminaison:
 - int pthread_mutex_destroy(pthread_mutex_t *mutex);

Atomic operations

- Operation executed atomically
- C11 defines a set of functions that perform atomic operations
 - C atomic_fetch_add(volatile A *object, M operand);
 - Bool atomic_flag_test_and_set(volatile atomic_flag *object);
- C11 defines atomic types
 - operations on these types are atomic
 - declaration: _Atomic int var; or _Atomic(int) var;