Threads

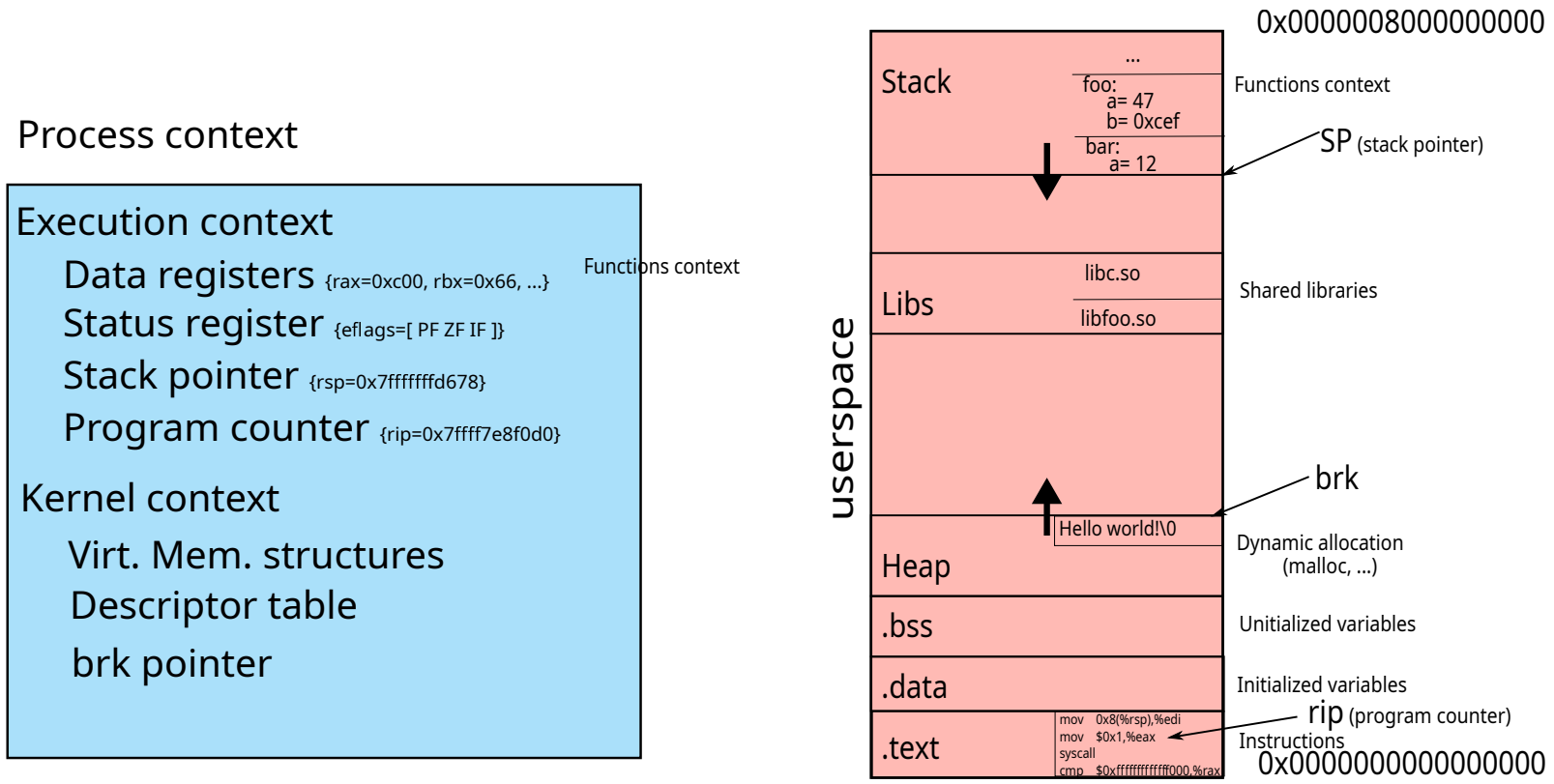
François Trahay

CSC4508 – Operating Systems

2022–2023

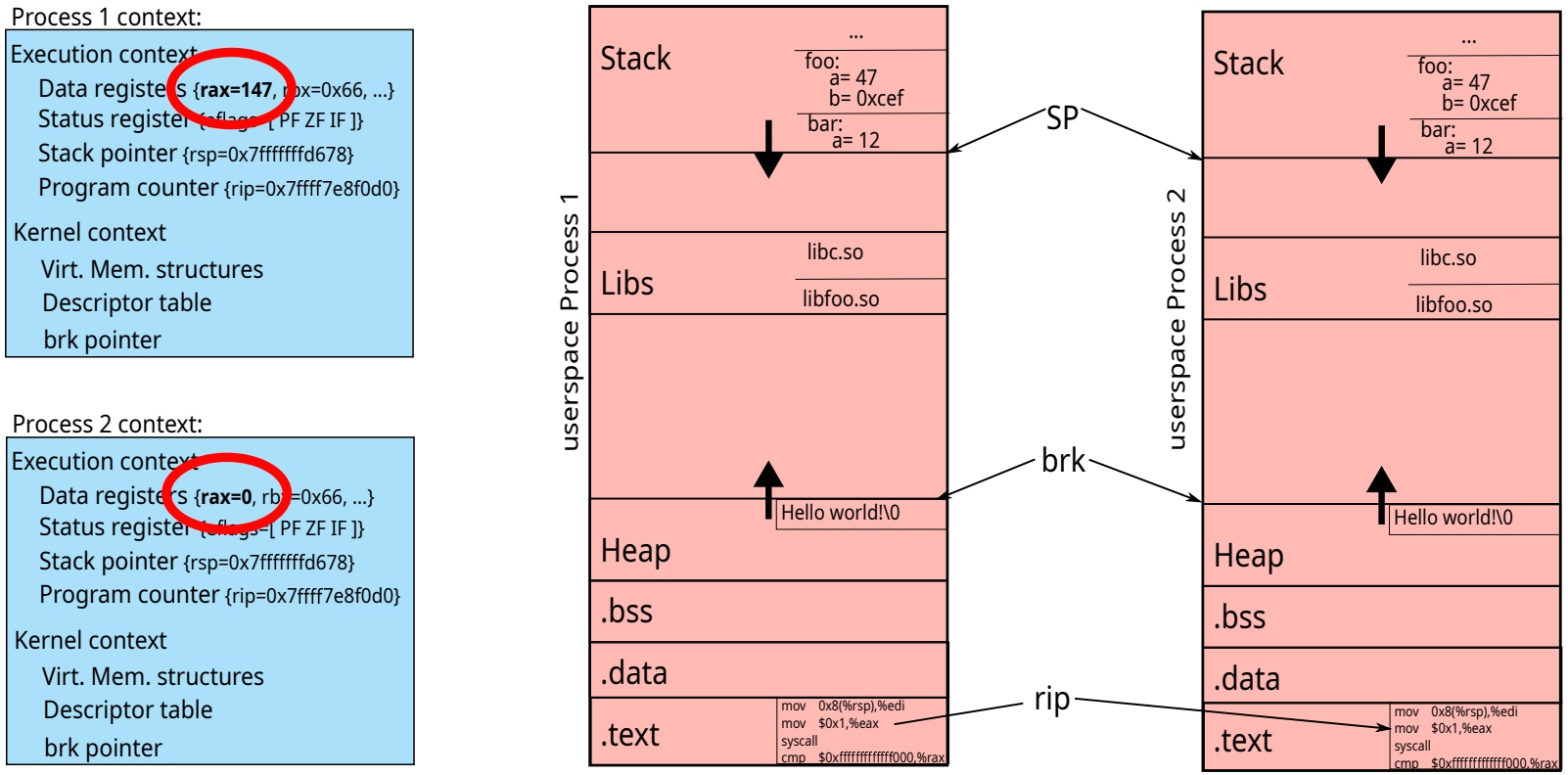
1 Execution context of a process

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



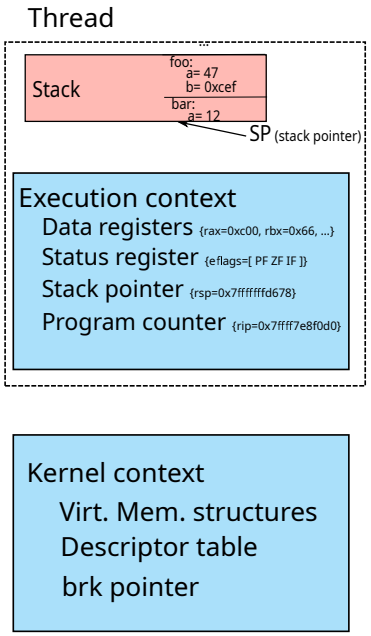
1.1 Duplicating a process

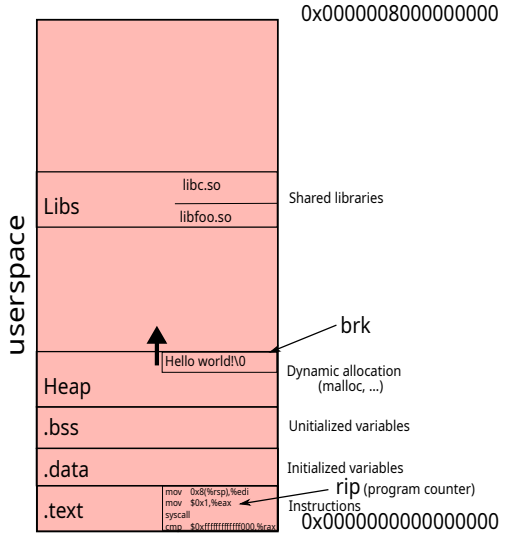
- Fork creates a new process and duplicates
  - Context: execution context + kernel context
  - except for the eax register
  - Address space: code, data and stack



1.2 Execution flows

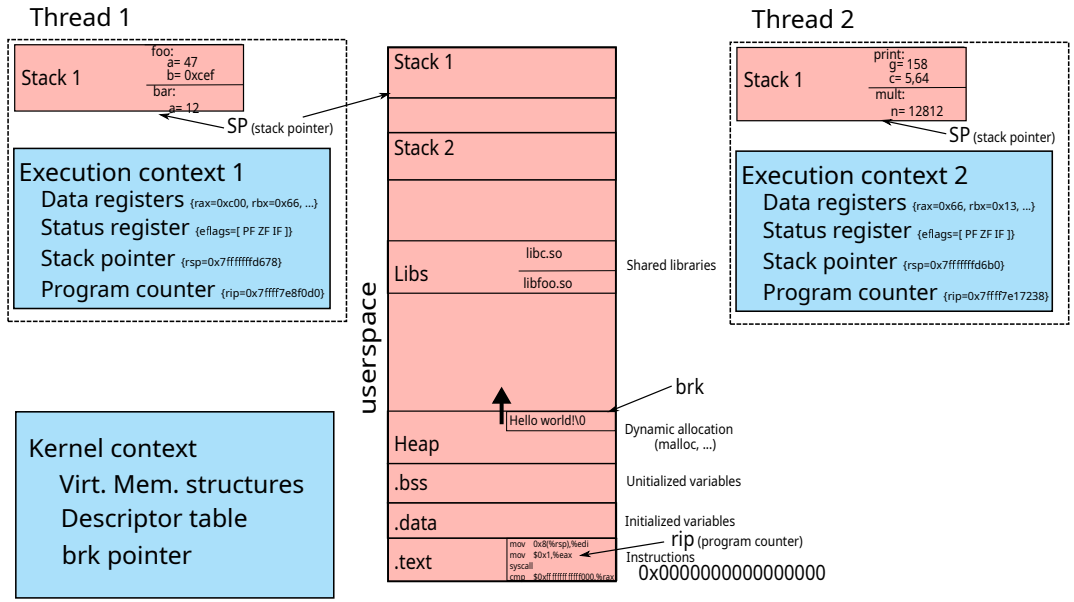
- Execution flow ! = Resources
  - Execution flow (or thread) : execution context + stack
  - Resources: code, data, kernel context





1.3 Multithreaded process

- Several execution flows
- Shared resources



1.4 Creating a Pthread

Creating a pthread

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

1.5 Other Pthread functions

- `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

2 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

2.1 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

2.2 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

2.3 TLS – Thread-Local Storage

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

3 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
    - 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)

3.1 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);`

3.2 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;`