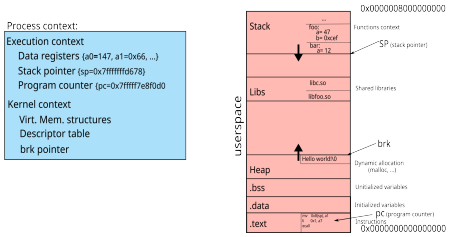
Threads

François Trahay

Contents

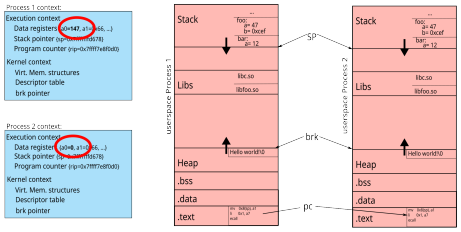
Execution context of a process

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



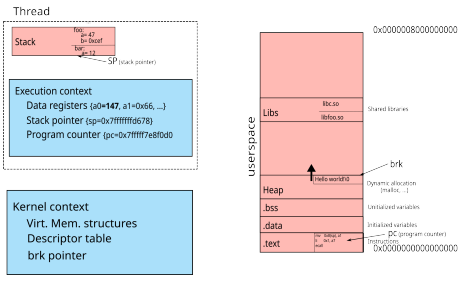
Duplicating a process

- 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



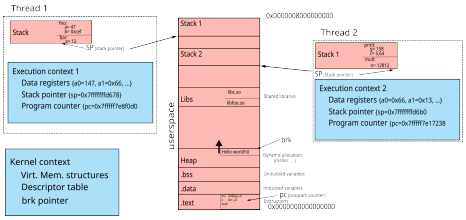
Execution flows

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



Multithreaded process

- Several execution flows
- Shared resources



In a multi-threaded process, each thread has a context (registers + stack). The rest of the memory (code, data, etc.) and resources (open files, etc.) are shared between threads.

The stacks of the different threads are located in memory so that they can grow. However, if a thread's stack grows too much, it might overflow onto the stack of another thread. To avoid this problem, the size of the stack is limited (the command `ulimit -s` gives the maximum stack size). This size limit can be changed using command line (by example `ulimit -s 32768`), or from a program (in using the `setrlimit` function).

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

We present here the Pthread API (POSIX thread) which is the most used in C. The C11 standard defines another interface for manipulating threads. However, there are only few implementations of this interface. The *de facto* standard therefore remains Pthread.

Unlike the creation of processes which generates a hierarchy (ie. each process has a parent process), there is no hierarchy between threads.

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 —

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

Technically, all the memory space is shared between the threads. It is therefore possible to share all the variables, including 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

Example: `strtok`

Another example of a non-reentrant function is the `char *strtok(char * str, char * delim)` function. This function extracts substrings from a string.

For example, the following code displays the different directories of the `PATH` variable:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void extract_path() {
  char* string = getenv("PATH");
```

```
  printf("Parsing '%s'\n", string);


  for(char* token = strtok(string, ":") ;
      token ;
      token = strtok(NULL, ":") ){
    printf("\t %s\n", token);
  }
}

int main(int argc, char**argv) {
  extract_path();
  return 0;
}
```

Here is an example of result obtained with this program:

```
Parsing '/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games'
         /usr/local/bin
         /usr/bin
         /bin
         /usr/local/games
         /usr/games
```

The `strtok` function is not reentrant because it is based on a previous state (a pointer to the last character tested in the string). Thus, in this example, the processing applied to each `token` cannot use `strtok`. For example:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void extract_path() {
  char* string = getenv("PATH");
  printf("Parsing '%s'\n", string);
  // string should contain a list of directories separated with :
  // eg. /usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games

  // Extract the directories
  // eg. /usr/local/bin, /usr/bin, /bin, /usr/local/games, /usr/games
  for(char* token = strtok(string, ":") ;
      token ;
      token = strtok(NULL, ":") ){
    // token contains a directory (eg. /usr/local/bin)
    printf("\t %s contains: ", token);

    // Extract the subdirectories
    // eg. usr, local, bin
```

```
    for(char* word = strtok(token, "/ ") ;
        word ;
        word = strtok(NULL, "/") ){
      printf("%s ", word);
    }
    printf("\n");
  }
}

int main(int argc, char**argv) {
  extract_path();
  return 0;
}
```

Will result in:

```
Parsing '/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games'
         /usr/local/bin contains: usr local bin
```

Here the first `token` (`/usr/local/bin`) is split into words (`usr`, `local`, `bin`) by successive calls to `strtok` which modify the previous state of `strtok`, which prevents subsequent calls to `token = strtok (NULL, ":")` to iterate over the `string` string.

Making a function reentrant

It is possible to make a non-reentrant function reentrant by adding a parameter corresponding to the state of the function. For example, the reentrant version of `char* strtok(char *str, const char *delim);` is `char* strtok_r(char *str, const char *delim, char **saveptr );`

Thus, the previous program can be corrected:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void extract_path() {
  char* string = getenv("PATH");
  char* saveptr = NULL;
  printf("Parsing '%s'\n", string);


  for(char* token = strtok_r(string, ":", &saveptr) ;
      token ;
      token = strtok_r(NULL, ":", &saveptr) ){
    printf("\t %s contains: ", token);
```

```
    char* saveptr_word = NULL;
    for(char* word = strtok_r(token, "/ ", &saveptr_word) ;
        word ;
        word = strtok_r(NULL, "/", &saveptr_word) ){
      printf("%s ", word);
    }
    printf("\n");
  }
}

int main(int argc, char**argv) {
  extract_path();
  return 0;
}
```

Which will result in:

```
Parsing '/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games'
         /usr/local/bin contains: usr local bin
         /usr/bin contains: usr bin
         /bin contains: bin
         /usr/local/games contains: usr local games
         /usr/games contains: usr games
```

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;`

TLS variables in C99

- Before the C11 standard, using thread-local storage was supported by some compilers using compiler-dependant keywords:
  - in C99 with gcc: `__thread int variable = 0;`
  - in C99 with Visual studio: `__declspec(thread) int variable = 0;`

`pthread_key`

- Another way (more portable, but much more painful to write) to declare a TLS variable is to use a `pthread_key`:
  - creation:

    - `int pthread_key_create(pthread_key_t *key, void (*destructor)(void*));`
  - terminaison:
    - `int pthread_key_delete(pthread_key_t *key););`
  - usage:
    - `void *pthread_getspecific(pthread_key_t key);`
    - `int pthread_setspecific(pthread_key_t key, const void *value);`
  - initialisation:
    - `int pthread_once(pthread_once_t *once_control, void (*init_routine) (void));`

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)

The following program illustrates the problem of simultaneous access to shared variables. Here, two threads each increment 1 000 000 000 times the same variable:

```c
/*
 * compteurBOOM.c
 *
 * Synchronization problem
 *
 *
 */

#include <error.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

/* INT_MAX / 2 */
#define NBITER 1000000000
```

```c
int counter = 0;

void *start_routine(void *arg) {
  int i;

  for (i = 0; i < NBITER; i++) {
      /* OOPS: WRONG ! Access to an unprotected shared variable */
      counter ++;
    }
  pthread_exit(NULL);
}

int main (int argc, char *argv[]) {
  int rc;
  pthread_t thread1, thread2;

  rc = pthread_create(&thread1, NULL, start_routine, NULL);
  if (rc)
    error(EXIT_FAILURE, rc, "pthread_create");

  rc = pthread_create(&thread2, NULL, start_routine, NULL);
  if (rc)
    error(EXIT_FAILURE, rc, "pthread_create");

  rc = pthread_join(thread1, NULL);
  if (rc)
    error(EXIT_FAILURE, rc, "pthread_join");
  rc = pthread_join(thread2, NULL);
  if (rc)
    error(EXIT_FAILURE, rc, "pthread_join");

  if (counter != 2 * NBITER)
    printf("BOOM! counter = %d\n", counter);
  else
    printf("OK counter = %d\n", counter);

  exit(EXIT_SUCCESS);
}
```

While the counter should be `2 * 1 000 000 000 = 2 000 000 000`, running this program gives another result, for example:

```
$ ./compteurBOOM
BOOM! compteur = 1076588402
```

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);`

Using a mutex, we can correct the BOOM counter program by ensuring that the counter increments are done in mutual exclusion:

```c
/*
 * compteurBOOM.c
 *
 * Synchronization problem
 *
 *
 */

#include <error.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

/* INT_MAX / 2 */
#define NBITER 1000000000

int counter = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *start_routine(void *arg) {
  int i;

  for (i = 0; i < NBITER; i++) {
    pthread_mutex_lock(&mutex);
    counter ++;
    pthread_mutex_unlock(&mutex);
```

```
  }
  pthread_exit(NULL);
}

int main (int argc, char *argv[]) {
  int rc;
  pthread_t thread1, thread2;

  rc = pthread_create(&thread1, NULL, start_routine, NULL);
  if (rc)
    error(EXIT_FAILURE, rc, "pthread_create");

  rc = pthread_create(&thread2, NULL, start_routine, NULL);
  if (rc)
    error(EXIT_FAILURE, rc, "pthread_create");

  rc = pthread_join(thread1, NULL);
  if (rc)
    error(EXIT_FAILURE, rc, "pthread_join");
  rc = pthread_join(thread2, NULL);
  if (rc)
    error(EXIT_FAILURE, rc, "pthread_join");

  if (counter != 2 * NBITER)
    printf("BOOM! counter = %d\n", counter);
  else
    printf("OK counter = %d\n", counter);

  exit(EXIT_SUCCESS);
}
```

While the result is correct, the use of a mutex significantly slows down the program (144s with mutex, against 4.1s without 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;`

We can fix the `counterBOOM` program by using atomic operations. To do this, all we have to do is declare the counter like `_Atomic int`. The counter increment then uses the atomic operation `atomic_fetch_add`.

```c
/*
 * compteurBOOM.c
 *
 * Synchronization problem
 *
 *
 */

#include <error.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

/* INT_MAX / 2 */
#define NBITER 1000000000

_Atomic int counter = 0;

void *start_routine(void *arg) {
  int i;

  for (i = 0; i < NBITER; i++) {
    counter ++;
  }
  pthread_exit(NULL);
}

int main (int argc, char *argv[]) {
  int rc;
  pthread_t thread1, thread2;

  rc = pthread_create(&thread1, NULL, start_routine, NULL);
  if (rc)
    error(EXIT_FAILURE, rc, "pthread_create");

  rc = pthread_create(&thread2, NULL, start_routine, NULL);
  if (rc)
    error(EXIT_FAILURE, rc, "pthread_create");

  rc = pthread_join(thread1, NULL);
```

```
  if (rc)
    error(EXIT_FAILURE, rc, "pthread_join");
  rc = pthread_join(thread2, NULL);
  if (rc)
    error(EXIT_FAILURE, rc, "pthread_join");

  if (counter != 2 * NBITER)
    printf("BOOM! counter = %d\n", counter);
  else
    printf("OK counter = %d\n", counter);

  exit(EXIT_SUCCESS);
}
```

Here, the result is correct and the program is much faster than when using a mutex:

- without synchronization: 4.1s
- with a mutex: 144s
- with an atomic operation: 35s