Selfie Threads

Operating Systems

UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA

CS3015 - 2024I
PROF.: [jgonzalez, rubaldo, mpinto]@utec.edu.pe

Objectives

- **Recall** the **similarities** and **differences** between threads and processes
- **Make an overview** of the Selfie Threads assignment
- Propose **two alternatives** for the shared low memory requirement
- **Explain** one of the Selfie's tests

Outline

Introduction

pthread_create()

pthread_join()

pthread_exit()

Final remarks

Assignments (recall)

- For our course, Selfie has **5 assignments**:
  - **Intro**
    - ~~print your name (L00)~~
  - **Processes**
    - ~~processes (L01)~~
  - **Syscalls**
    - ~~fork-wait (L02-1)~~
    - ~~fork-wait-exit (L02-2)~~
  - **Mutex**
    - ~~lock (L03)~~
  - **Threads**
    - threads (L04)

Processes vs Threads

Threads describe **another process model**, a different abstraction created by the OS.

Processes

- "Start Private" approach
  - Inter-Process Communication
  - Ex. Give processes access to same memory

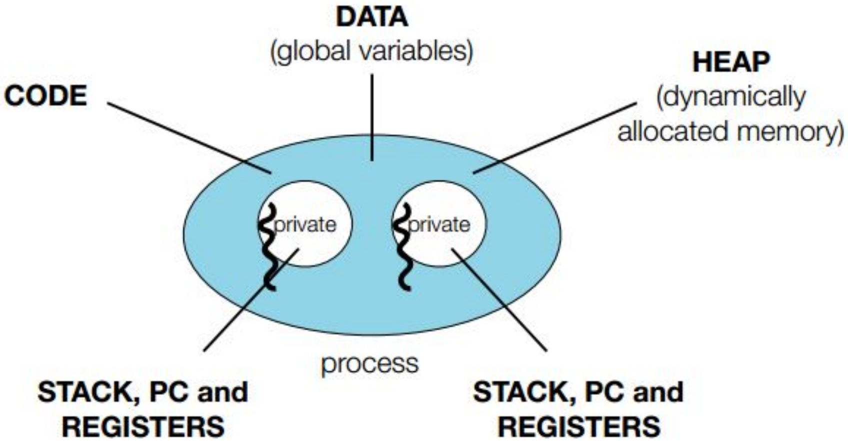
Threads

- "Start Shared" approach
- Share almost everything from the start
  - Communicate through shared memory

Memory layout

Threads **share process state**:

- Parts of memory (address space)
  - **Do share** code, data and heap segments.
  - **Do not share** stack, registers, PC
- Resources
  - File descriptors, hardware access, etc.



Outline

Introduction

pthread_create()

pthread_join()

pthread_exit()

Final remarks

pthread_create()

You will implement the **pthread_create()** syscall. When called, a new thread is created for the current process.

- **Very similar to fork:**
  - Takes no arguments
  - **Returns** pid of child or zero
  - Thread continues execution on next instruction
  - Identifiers should start at zero
- **Key differences:**
  - Code, data and heap segments are shared.
  - All threads exit on process exit.

Thread relationships

Threads generally run **as part of a process.** Therefore threads only share memory with threads **within the same process.**

How do we know **which process is related to a thread**?

- Parent-Children relationships?
- Thread Groups?
- **However you wish to implement it!**

Two Ideas for a solution

- Idea 1: Use unique address spaces for each thread and synchronize their shared segments.
  - Stacks can be allocated normally.
  - Reads, Writes and allocations to shared memory must be replicated for all threads.

- Idea 2: Use a shared address space for all threads of a process, and only allocate separate stacks.
  - Define a policy to allocate stacks below each other.
  - Threads use the same page table, no need for replication of reads and writes to shared memory:
    - **Be careful!** Extensions of the program break **need to be replicated for all threads.**

Idea 1: Unique synchronized address space

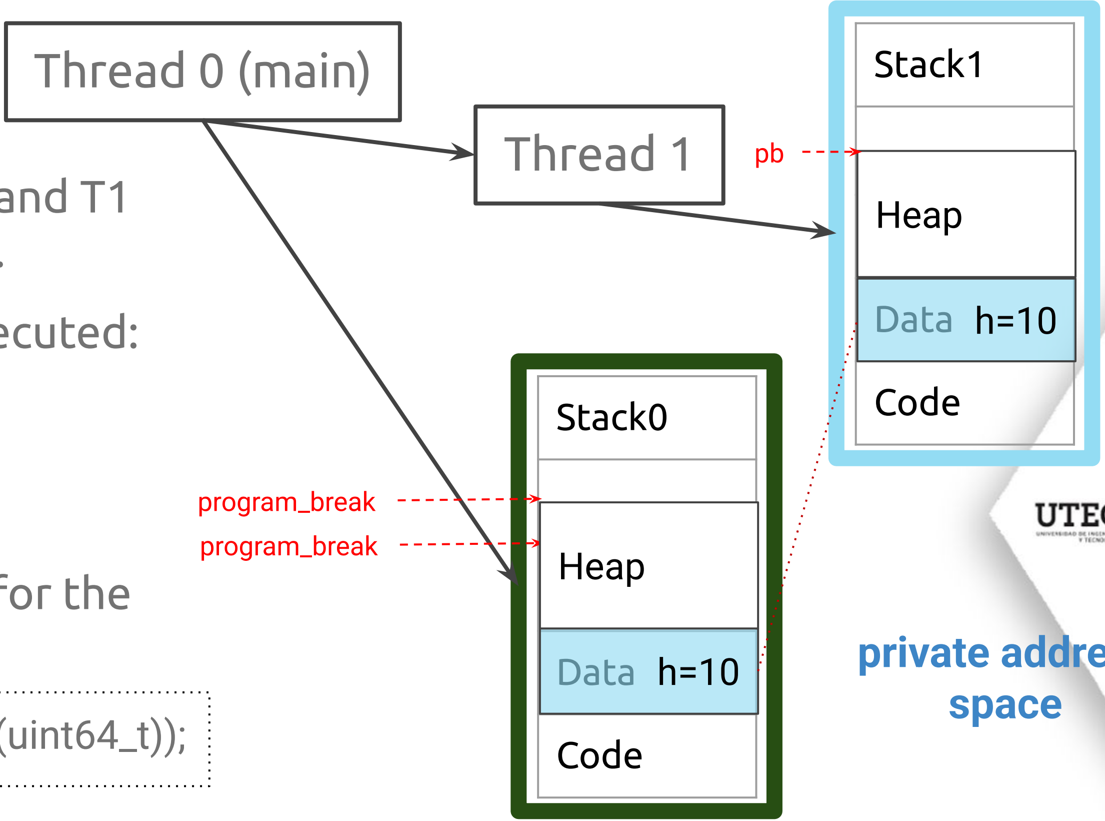
The **low memory** between T0 and T1 should be **synchronized** when.

- An **store** instruction is executed:

```
global h = 0;
...
h = 10;
```

- **New memory** is allocated for the **heap**

```
heap_variable = malloc(sizeof(uint64_t));
```



private address space

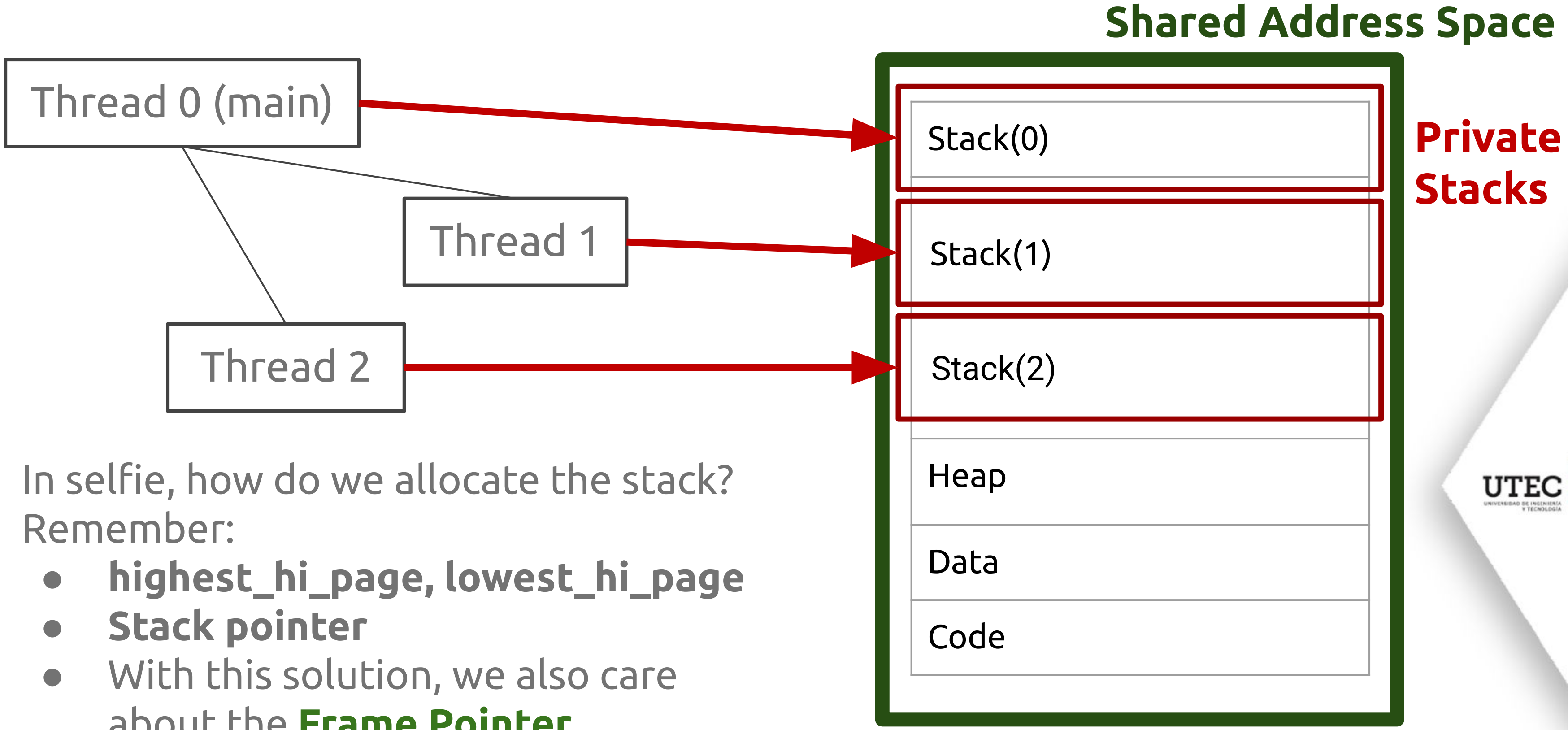
Idea 1: Unique synchronized address space

Functions to take into account:

- do_store()
- implement_brk()
- handle_page_fault

Be careful! Only **code**, **data** and **heap** segments should be synchronized within threads. **Stack** should remain **private**.

Idea 2: Shared address space



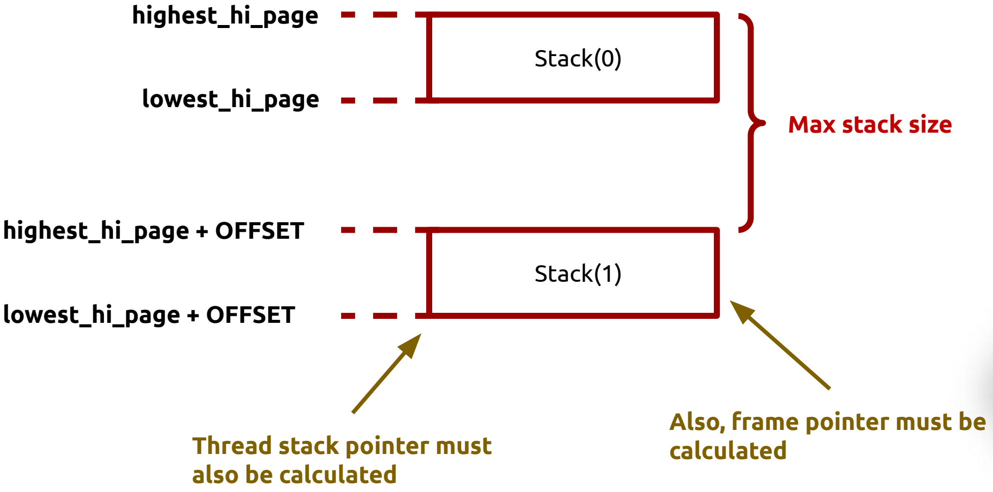
In selfie, how do we allocate the stack? Remember:

- **highest_hi_page, lowest_hi_page**
- **Stack pointer**
- With this solution, we also care about the **Frame Pointer**

Idea 2: Shared address space

Functions to take into account:

- implement_brk()
- try_brk()



Thread stack pointer must also be calculated

Also, frame pointer must be calculated

Outline

Introduction

pthread_create()

pthread_join()

pthread_exit()

Final remarks

pthread_join(wstatus)

You will implement the **pthread_join()** syscall. When called, a parent thread will wait until **one of it children threads exits**.

- **Very similar to wait:**
  - Returns pid of child exited
  - Save the child exit code in **wstatus**
    - But: there is no need to multiply by 256
  - Thread continues execution on next instruction

Outline

Introduction

pthread_create()

pthread_join()

pthread_exit()

Final remarks

pthread_exit(status)

You will implement the **pthread_exit()** syscall. When called, a thread will exit with **status**.

- **Very similar to exit:**
  - **Notify** a parent (if exists) my **pid** and **exit code**
  - **Remove** from **used_context** if I don't have children and I'm not the main thread
- **Key differences:**
  - Main thread continues execution on next instruction

Outline

Introduction

pthread_create()

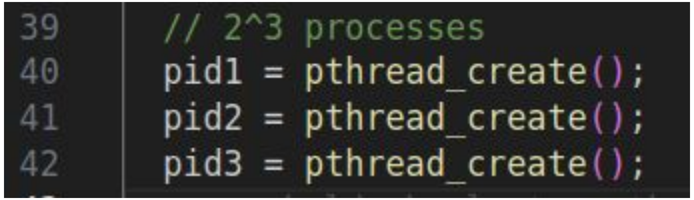
pthread_join()

pthread_exit()

Final remarks

Test syscalls.c in detail

Each thread has the following variables with different values:

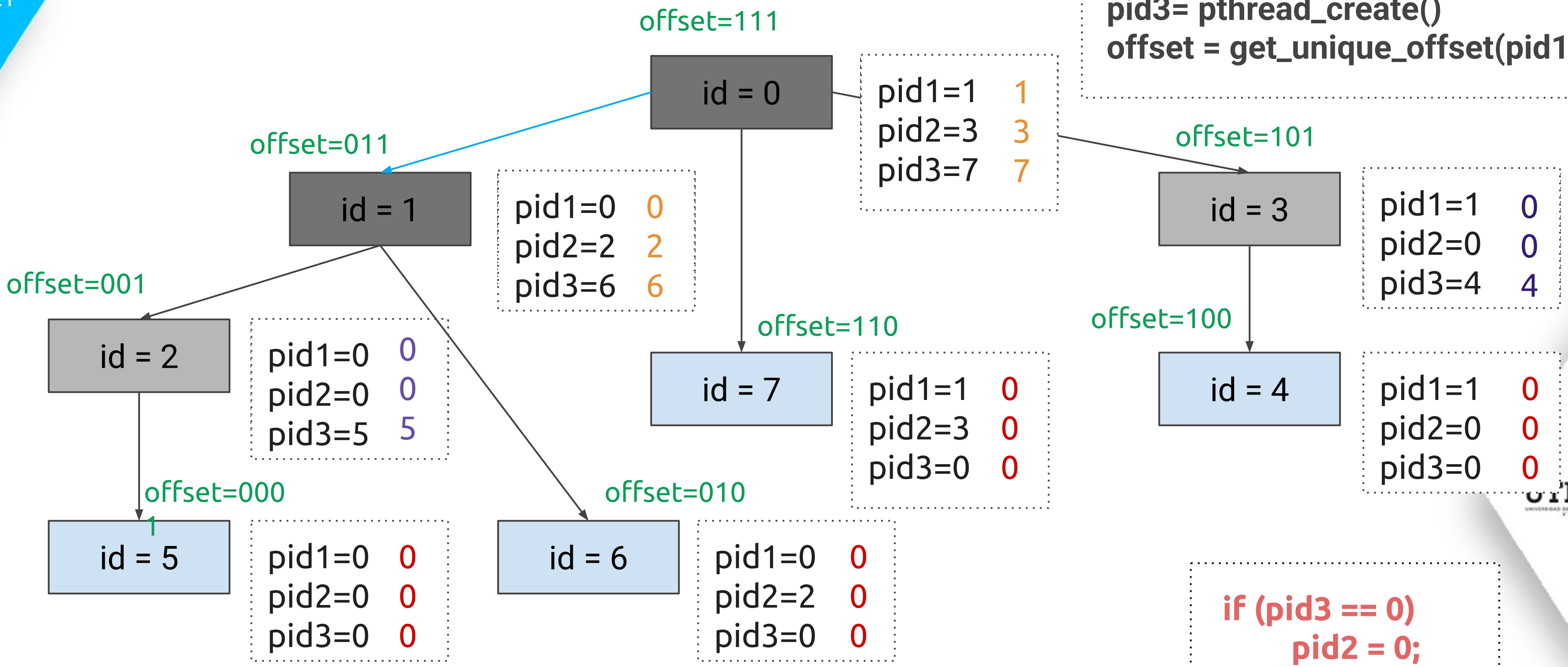


```
// 2^3 processes
pid1 = pthread_create();
pid2 = pthread_create();
pid3 = pthread_create();
```

At the beginning **just one** context with id=0 exists. This context calls pthread_create() and creates a new thread context with id=1. Then both contexts call pthread_create() again and again.

Test syscalls.c in detail

```
pid1= pthread_create()
pid2= pthread_create()
pid3= pthread_create()
offset = get_unique_offset(pid1-3);
```



offset=111

id = 0

pid1=1 1
pid2=3 3
pid3=7 7

offset=011

id = 1

pid1=0 0
pid2=2 2
pid3=6 6

offset=101

id = 3

pid1=1 0
pid2=0 0
pid3=4 4

offset=001

id = 2

pid1=0 0
pid2=0 0
pid3=5 5

offset=110

id = 7

pid1=1 0
pid2=3 0
pid3=0 0

offset=100

id = 4

pid1=1 0
pid2=0 0
pid3=0 0

offset=000

1

id = 5

pid1=0 0
pid2=0 0
pid3=0 0

offset=010

id = 6

pid1=0 0
pid2=2 0
pid3=0 0

```
if (pid3 == 0)
        pid2 = 0;
if (pid2 == 0)
        pid1 = 0;
```

Test syscalls.c in detail (cont.)

```
uint64_t sumChilds(uint64_t offset, uint64_t pid, uint64_t acc) {
 if (pid) {
         acc = acc + pid + pthread_join(status + offset);
         acc = acc + *(status + offset);
 }
 return acc;
```

```
T7:    = sumChilds(110, 0, sumChilds(110, 0, sumChilds(110, 0, 0)));
       = sumChilds(110, 0, sumChilds(110, 0, 0));
       = sumChilds(110, 0, 0);
       = 0
```

```
T6:    = sumChilds(010, 0, sumChilds(010, 0, sumChilds(010, 0, 0)));
       = 0
```

```
T5:    = sumChilds(000, 0, sumChilds(000, 0, sumChilds(000, 0, 0)));
       = 0
```

```
T4:    = sumChilds(100, 0, sumChilds(100, 0, sumChilds(100, 0, 0)));
       = 0
```

Test syscalls.c in detail (cont.)

```
uint64_t sumChilds(uint64_t offset, uint64_t pid, uint64_t acc) {
 if (pid) {
        acc = acc + pid + pthread_join(status + offset);
        acc = acc + *(status + offset);
 }
 return acc;
```

```
T3:   = sumChilds(101, 0, sumChilds(101, 0, sumChilds(101, 4, 0)));
      = 0 + 4 + pthred_join(101) + status[101]
      = 0 + 4 + pthred_join(T4) + status[T4]
      = 0 + 4 + 4 + 0
      = 8
```

```
T2:   = sumChilds(001, 0, sumChilds(001, 0, sumChilds(001, 5, 0)));
      = 0 + 5 + pthred_join(T5) + status[T5]
      = 0 + 5 + 5 + 0
      = 10
```

Test syscalls.c in detail (cont.)

```
uint64_t sumChilds(uint64_t offset, uint64_t pid, uint64_t acc) {
 if (pid) {
         acc = acc + pid + pthread_join(status + offset);
         acc = acc + *(status + offset);
 }
 return acc;
```

T1: = **sumChilds**(011, 0, **sumChilds**(011, 2, **sumChilds**(011, 6, 0)));
= 0 + [(0 + 6 + **pthred_join(T6)** + status[T6]) + 2 + **pthred_join(T2)** + status[T2]]
= 0 + [(0 + 6 + 6 + 0) + 2 + 2 + 10]
= 26

T0: = **sumChilds**(111, 1, **sumChilds**(111, 3, **sumChilds**(111, 7, 0)));
= [(0 + 7 + **pthred_join(T7)** + status[T7]) + 3 + **pthred_join(T3)** + status[T3]] + 1 +
pthred_join(T1) + status[T1]
= [(0 + 7 + 7 + 0) + 3 + 3 + 8] + 1 + 1 + 26
= 56

Activity

- Laboratory guide in **Canvas** **Week10 > L04-threads.pdf**

Selfie Threads

Operating Systems

CS3015 - 2024I

PROF.: [jgonzalez, rubaldo, mpinto]@utec.edu.pe