Threads

Juan José Costa and Alejandro Pajuelo

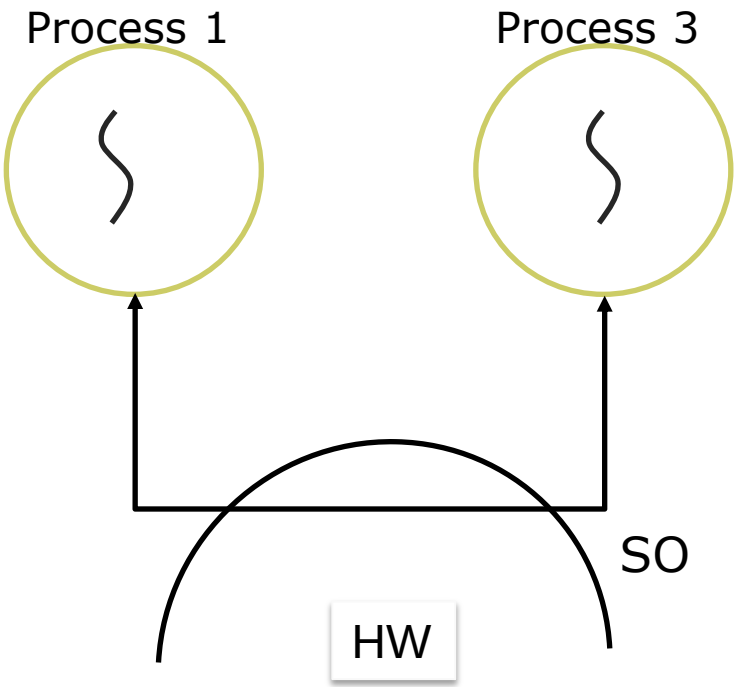
Escola Tècnica Superior de Telecomunicacions de Barcelona (ETSETB)

Universitat Politècnica de Catalunya (UPC)

2021-2022Q2

Problem

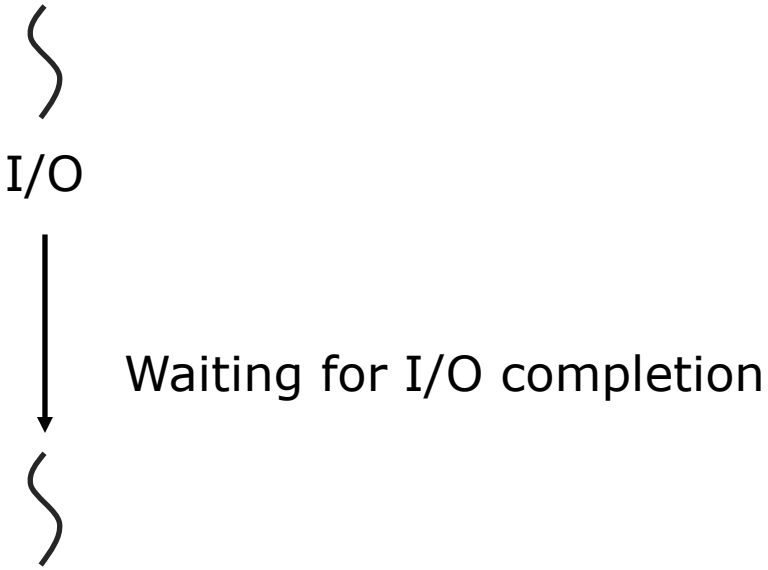
- Cooperative processes send information to each other using system calls since they do not share memory



- Introduces overhead
- Parallel tasks can become sequential
- Solution: shared memory

Problem

- I/O cannot be parallelized with just one flow of code execution



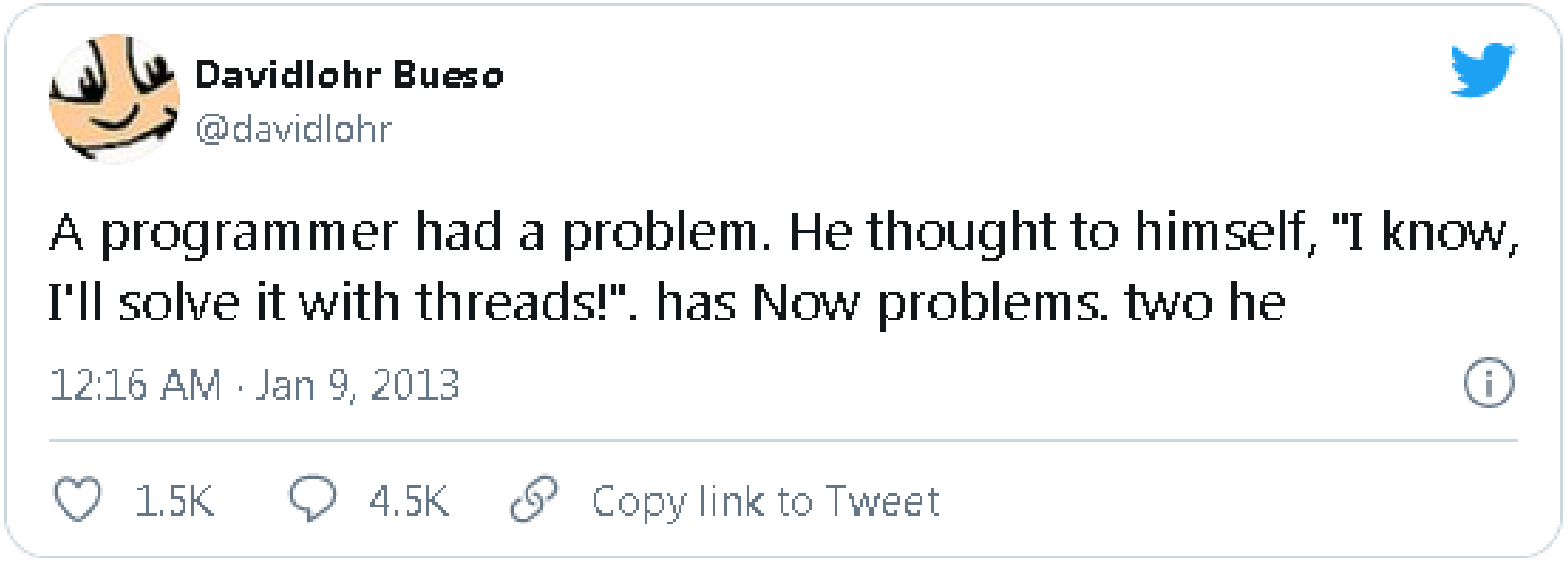
- Maybe, some computations could be done while waiting for I/O

Threads - Why?

- When and why they are used…
  - Parallelism exploitation (code and hardware resources)
  - Encapsulate tasks (modular coding)
  - I/O efficiency (specific Threads for I/O)
  - Pipelining of service requests (to maintain QoS of services)

- Advantages
  - Creation/termination/context switches of threads are faster than with processes
  - With memory sharing inside the process, threads can share information without system calls

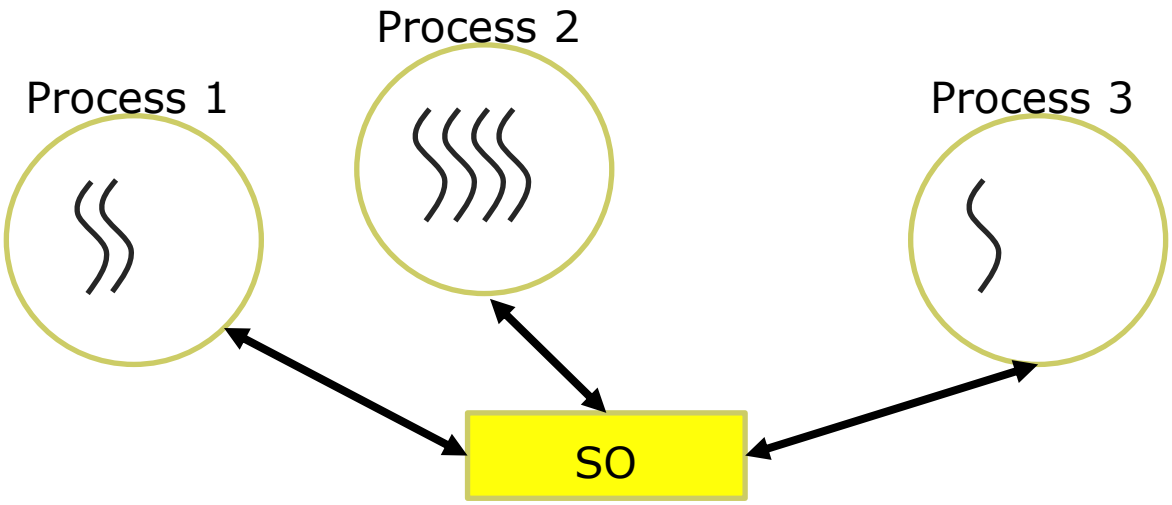
Why not?



- Disadvantages
  - Difficult coding and debugging due to the shared memory
    - Problems of synchronization and mutual exclusion
      - Incoherent executions, wrong results, deadlocks, …

Threads

- A process has one thread when execution starts
- A process can have several threads
  - High performance videogames **>50 threads**;
  - Firefox/Chrome **>80 threads**
- In the next Figure: Process 1 has 2 threads; Process 2 has 4 threads; Process 3 has 1 thread
- Process management with several threads depends on the OS support
  - **User Level Threads** vs **Kernel Level Threads**



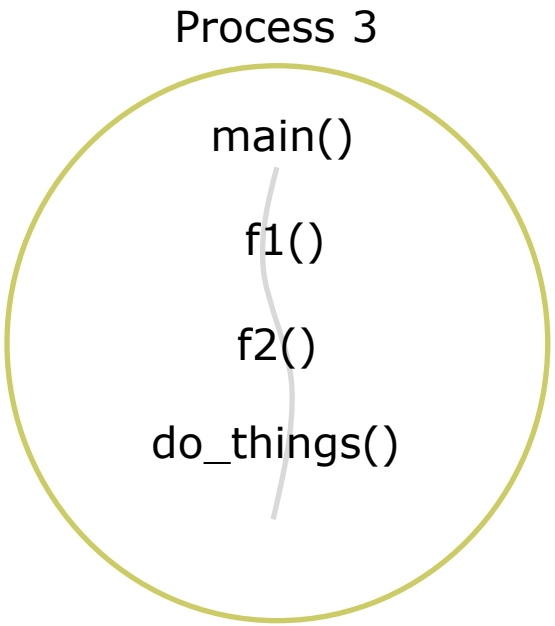
What are threads?

- Concept of a process:
  - OS representation of a program in execution
  - Resource container

- A thread is one of the resources of a process
  - Minimal scheduling unit
    - Every independent part of the code can be assigned to a thread
  - Thread resources
    - Identifier (Thread ID: TID)
    - Stack Pointer
    - Program Counter
    - Register File
    - Errno
  - Threads share the resources of the process they belong to

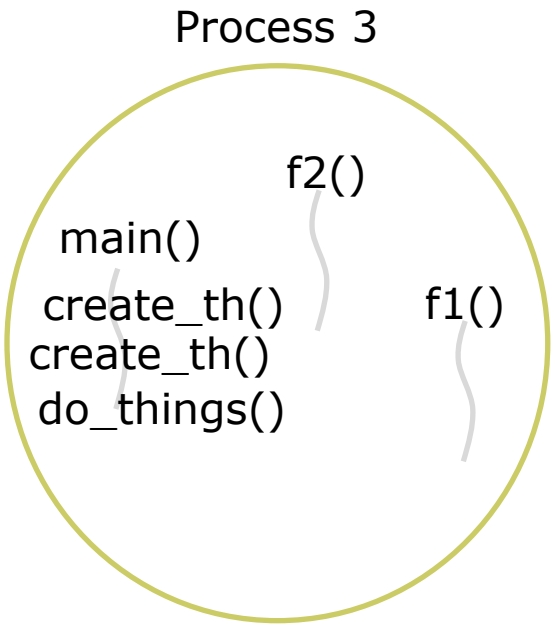
Thread sample

```
int global = 2;
int f1(void) {
        …
}
int f2(void) {
        …
}
int main() {
        f1()
        f2()
        do_things();
}
```



Thread sample

```
int global = 2;
int f1(void) {
        …
}
int f2(void) {
        …
}
int main() {
        create_tread( f1 )
        create_thread( f2 )
        do_things();
}
```



Basic services (UNIX)

Servicio	Llamada a sistema
Thread creation	`pthread_create`
Thread finalization	`pthread_exit`
Wait for a thread	`pthread_join`
Identification	`pthread_self`
Scheduling	`pthread_yield`

- Do not forget to include the header:

```
#include <phtread.h>
```

- And link with `-pthread`

Thread creation

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

- `pthread_t *thread`: identifier of the created thread
- `pthread_attr_t *attr`: initial attributes to create the thread
- `void *(*start_routine)(void *):` function with the thread code
- `void *arg`: argument to pass to thread

```
start_routine(arg);
```

- Thread identifiers are only valid inside the process (`pthread_self`)
- Don’t forget to link with `-lpthread`

Thread creation: example

```
void * func(void *arg)
{
}

int main()
{
        …
        pthread_t id;
        pthread_create(&id, NULL, func, NULL);
        …
}
```

Thread creation: example

- Creation of 10 threads

```
void * func(void *arg)
{
}

int main()
{
        …
        int i;
        pthread_t id[10];
        for (i=0; i<10; i++)
        {
                pthread_create(&id[i], NULL, func, NULL);
        }
        …
}
```

Thread creation: example

- Argument passing: creation order

```
void * func(void *arg)
{
        int order = (int)arg;
}

int main()
{
        …
        int i;
        pthread_t id[10];
        for (i=0; i<10; i++)
        {
                pthread_create(&id[i], NULL, func, (void*)i);
        }
        …
}
```

Thread termination

```
void pthread_exit(void *retval);
```

- `void *retval`: termination code

- The compiler **implicitly** adds a call to `pthread_exit` in case the programmer does not put it in the code

Thread termination: example

```
void * func(void *arg)
{
        pthread_exit(0);
}

int main()
{
        …
        pthread_t id;
        pthread_create(&id, NULL, func, NULL);
        …
}
```

Wait for a thread

```
int pthread_join(pthread_t thread, void **retval);
```

- `pthread_t thread`: identifier of the thread to wait for
- `void **retval`: termination code of the thread

- **Blocking** call
- pthread_t thread is the identifier returned when the thread was created
- void **retval is the value used in pthread_exit

Wait for a thread: example

```
void * func(void *arg)
{
        pthread_exit(0);
}

int main()
{
        …
        pthread_t id;
        int a;
        pthread_create(&id, NULL, func, NULL);
        pthread_join(id, (void**)&a);
        …
}
```

Wait for 10 threads (parallel): example

```
void * func(void *arg){
        int order = (int)arg;
        pthread_exit((void*)order);
}

int main(){
        …
        int i;
        pthread_t id[10];
        int ret_val;
        for (i=0; i<10; i++){
                pthread_create(&id[i], NULL, func, (void*)i);
        }
        for (i=0; i<10; i++){
                pthread_join(id[i], (void**)&ret_val);
        }
        …
}
```

Wait for 10 threads (sequential): example

```
void * func(void *arg) {
        int order = (int)arg;
        pthread_exit((void*)order);
}

int main(){
        …
        int i;
        pthread_t id[10];
        int ret_val;
        for (i=0; i<10; i++){
                pthread_create(&id[i], NULL, func, (void*)i);
                pthread_join(&id[i], (void**)&ret_val);
        }
        …
}
```

Specialization: example

- Create 1 thread that executes a CPU-intensive workload and overlap this computation with I/O

```
void * CPU_thread(void *arg)
{
        int order = (int)arg;
        Do_a_nice_computation_here();
        pthread_exit((void*)order);
}
```

```
void * IO_thread(void *arg)
{
        int order = (int)arg;
        Do_a_nice_IO_here();
        pthread_exit((void*)order);
}
```

```
int main()
{
        …
        pthread_t id[2];
        int ret_val;
        pthread_create(&id[0], NULL, CPU_thread, (void*)1);
        pthread_create(&id[1], NULL, IO_thread, (void*)2);
        pthread_join(&id[0], (void**)&ret_val);
        pthread_join(&id[1], (void**)&ret_val);
        …
}
```

Shared Memory

- Threads:
  - Share the memory of the process they belong to
  - In general, they share all the resources assigned to the process
- Processes:
  - Don’t share memory
  - Every process has its own memory

Shared memory with threads: example

```
int a=0;
void * Thread_1(void *arg)
{
          while (1) a=1;
}

void * Thread_2(void *arg)
{
          while (1) a=2;
}

int main()
{
          …
          pthread_t id[2];
          int ret_val;
          pthread_create(&id[0], NULL, Thread_1, NULL);
          pthread_create(&id[1], NULL, Thread_2, NULL);
          pthread_join(&id[0], (void**)&ret_val);
          pthread_join(&id[1], (void**)&ret_val);
          …
}
```

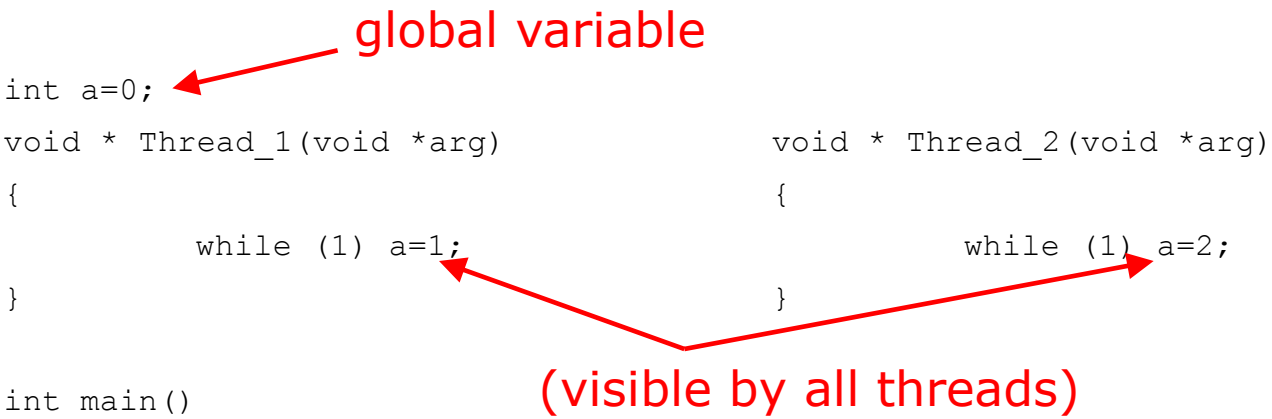
Shared memory with threads: example


```
int a=0;
void * Thread_1(void *arg)
{
          while (1) a=1;
}

int main()
{
          …
          pthread_t id[2];
          int ret_val;
          pthread_create(&id[0], NULL, Thread_1, NULL);
          pthread_create(&id[1], NULL, Thread_2, NULL);
          pthread_join(&id[0], (void**)&ret_val);
          pthread_join(&id[1], (void**)&ret_val);
          …
}
```

```
void * Thread_2(void *arg)
{
          while (1) a=2;
}
```

Shared memory with threads: example




```
int main()
{
        …
        pthread_t id[2];
        int ret_val;
        pthread_create(&id[0], NULL, Thread_1, NULL);
        pthread_create(&id[1], NULL, Thread_2, NULL);
        pthread_join(&id[0], (void**)&ret_val);
        pthread_join(&id[1], (void**)&ret_val);
        …
}
```

Shared memory with threads: example

global variable

```
int a=0;
void * Thread_1(void *arg)
{
        while (1) a=1;
}

int main()
{
        …
        pthread_t id[2];
        int ret_val;
        pthread_create(&id[0], NULL, Thread_1, NULL);
        pthread_create(&id[1], NULL, Thread_2, NULL);
        pthread_join(&id[0], (void**)&ret_val);
        pthread_join(&id[1], (void**)&ret_val);
        …
}
```

```
void * Thread_2(void *arg)
{
        while (1) a=2;
}
```

(visible by all threads)

What happens if I put a loop here printing var ‘a’ value?

Shared memory with processes: example

- Processes do not share memory!!!!

```
int a=0;
void * Proc1_1()
{
          while (1) a=1;
}

int main()
{
          …
          if (fork()==0) Proc_1();
          if (fork()==0) Proc_2();
          …
}
```

```
void * Proc_2()
{
          while (1) a=2;
}
```

Shared memory with processes: example

- Processes do not share memory!!!!

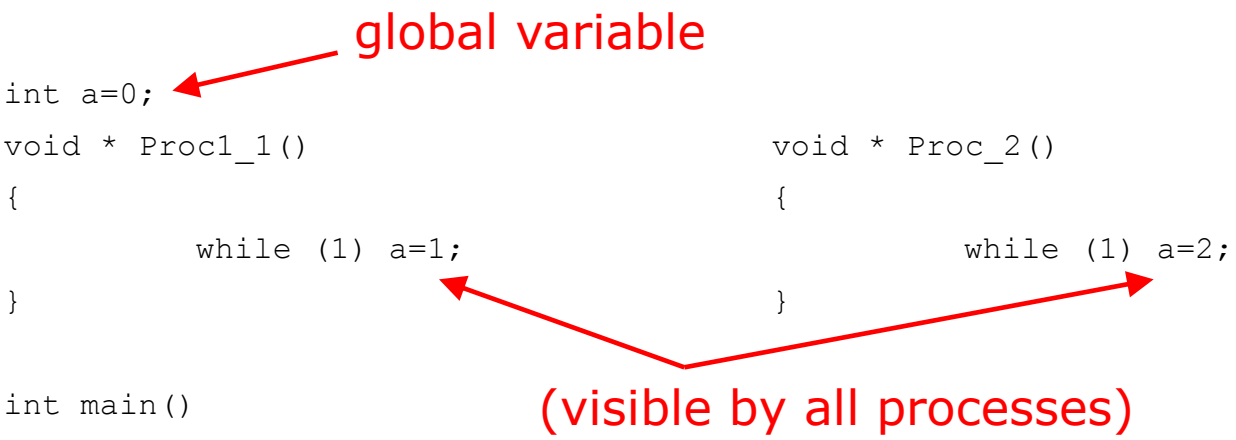

```
int a=0;
void * Proc1_1()
{
          while (1) a=1;
}

int main()
{
          …
          if (fork()==0) Proc_1();
          if (fork()==0) Proc_2();
          …
}
```

```
void * Proc_2()
{
          while (1) a=2;
}
```

Shared memory with processes: example

- Processes do not share memory!!!!



Shared memory with processes: example

- Processes do not share memory!!!!

global variable

```
int a=0;
void * Proc1_1()
{
          while (1) a=1;
}

int main()
{
          …
          if (fork()==0) Proc_1();
          if (fork()==0) Proc_2();
          …
}
```

```
void * Proc_2()
{
          while (1) a=2;
}
```

(visible by all processes)

What happens if I put a loop here printing var ‘a’ value?

Scheduler

- If an OS supports threads, the scheduler must be modified:
  - Processes are not longer scheduled since they are resource containers
- Two ways:
  - 1 level scheduling: thread scheduling
    - All the threads in the system are considered
  - 2 level scheduling:
    - The next thread to execute is chosen among the ready threads of the current process
    - In case that no thread is candidate, select one thread from another process
- Algorithms and scheduling policies are the same

Relinquish the CPU

- There are times where the CPU is not needed and you may voluntarily free the CPU for others threads to execute.

```
int pthread_yield( void );
```

- Relinquishes the CPU and another thread is scheduled to run.
- (For processes the `sched_yield` is also available)

Summary

- Thread concepts
- Services
  - Creation → `pthread_create`
  - Finalization → `pthread_exit, pthread_join`
  - Identification → `pthread_self`
  - Scheduling → `phtread_yield`