Threads

Real-Time Operating Systems Programming (RTOSP)
Master in Critical Computing Systems Engineering (MCCSE)

2022/23

Paulo Baltarejo Sousa and Cláudio Maia
{pbs,crr}@isep.ipp.pt

Disclaimer

Material and Slides

Some of the material/slides are adapted from various:

- Presentations found on the internet;
- Books;
- Web sites;
- ...

Outline

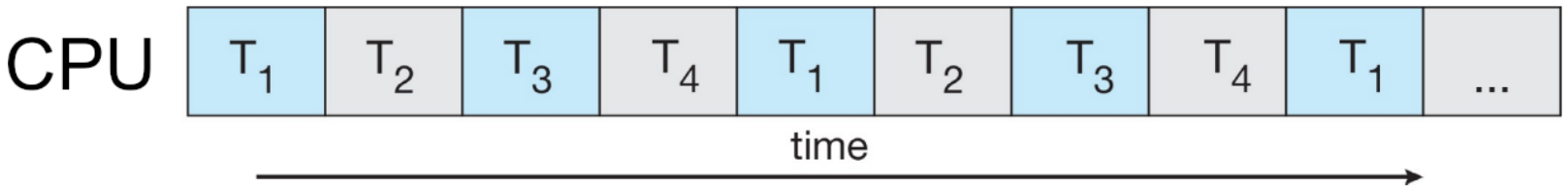
Concurrency and Parallelism

Concurrency *vs* Parallelism (I)

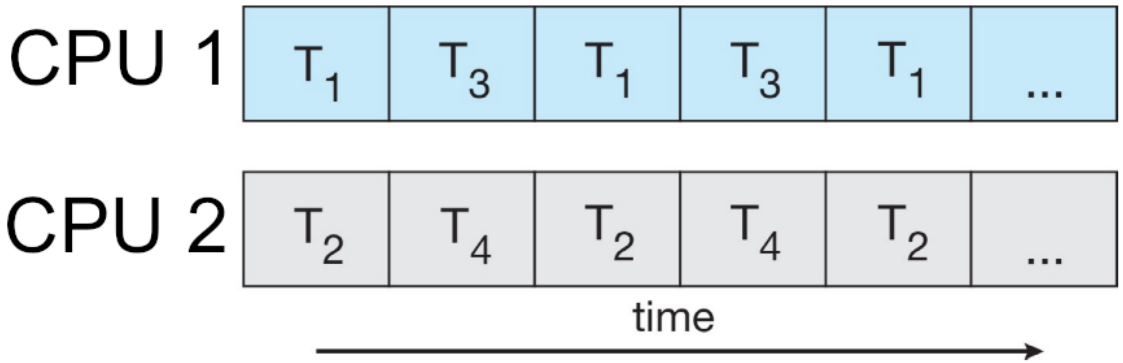
- A question you might ask is whether processes or threads **can run at the same time**.
- The answer is: **It depends**.
  - On a system with **multiple processors or CPU cores**, **multiple processes or threads can be executed in parallel**.
  - On a **single processor**, though, **it is not possible to have processes or threads truly executing at the same time**.
    - In this case, the **CPU is shared among running processes or threads** using a process scheduling algorithm that divides the CPU's time and yields the illusion of parallel execution.
    - The time given to each task is called a **time slice**.
    - The switching back and forth between tasks happens so fast it is usually not perceptible.

Concurrency *vs* Parallelism (II)

- **Concurrent** execution on single-processor system



- **Parallelism** on a multiple processor system:



Threads

Why Threads?

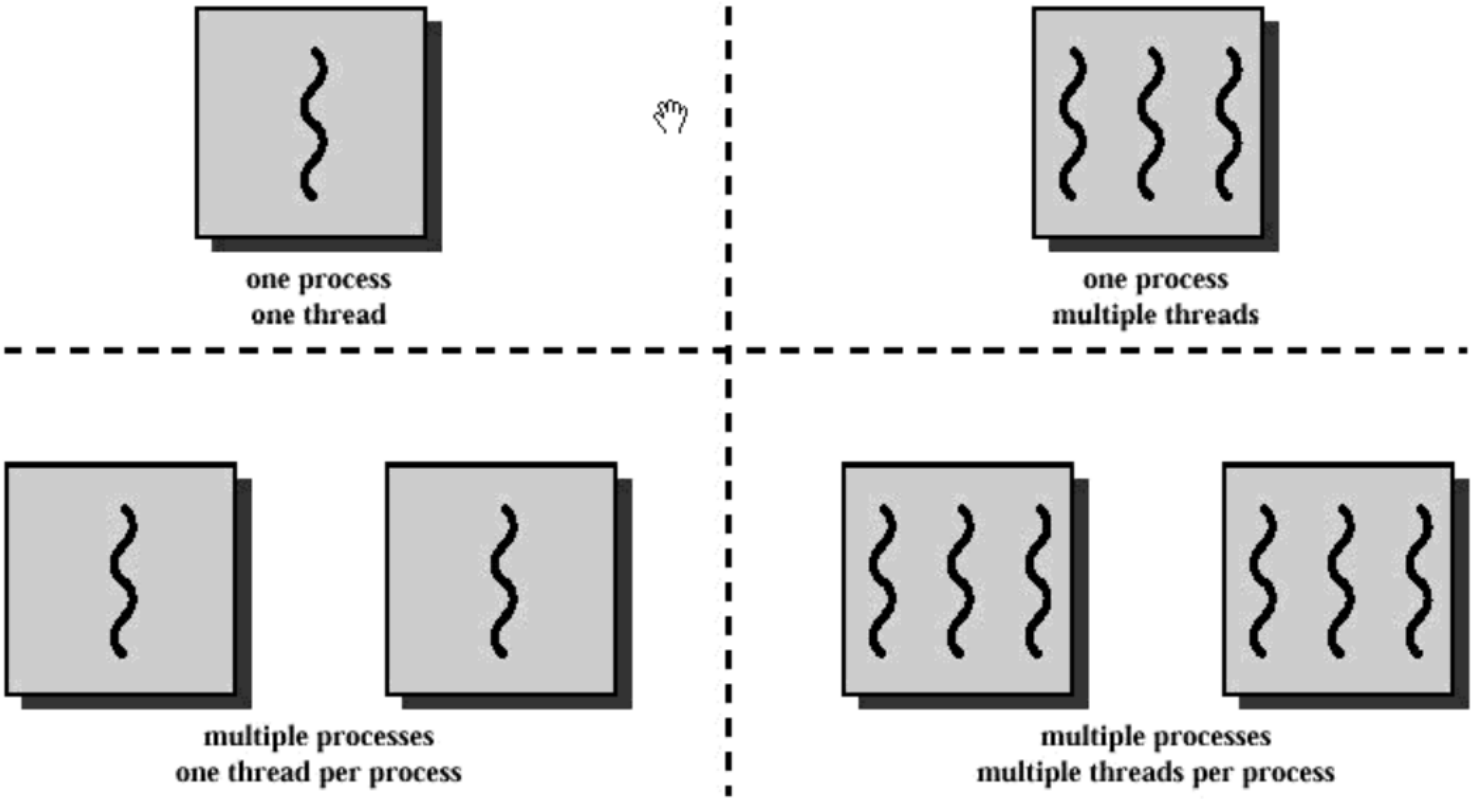
- Threads are very useful **whenever a process has multiple tasks to perform independently of the others**.
- This is particularly true when **one of the tasks may block**, and it is desired to allow the other tasks to proceed without blocking.
  - For example in a word processor, a background thread may check spelling and grammar while a foreground thread processes user input (keystrokes), while yet a third thread loads images from the hard drive, and a fourth does periodic automatic backups of the file being edited.
  - Another example is a web server, where multiple threads allow for multiple requests to be satisfied simultaneously.

What are?

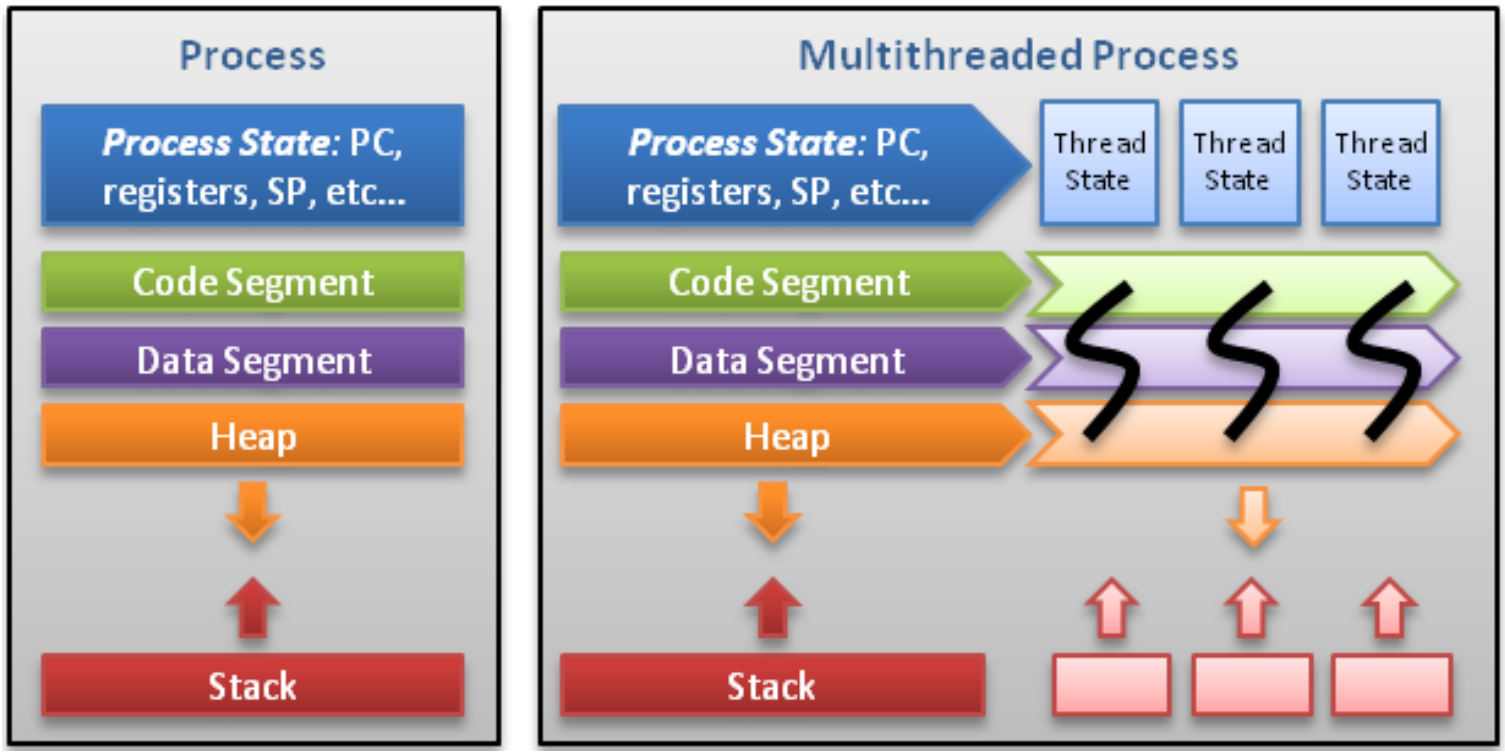
- A **computer program becomes a process when it is loaded** from some store into the computer's memory and begins execution.
- A process **can be executed by a processor or a set of processors**.
- A process description in memory contains vital information such as the program counter which keeps track of the current position in the program (i.e. which instruction is currently being executed), registers, variable stores, file handles, signals, and so forth.
- A thread **is a sequence of such instructions within a program that can be executed independently of other code**.

Multithreading *vs* Single threading

- Models for threads and processes.



Threads (I)



- Threads contain only necessary information, such as **stack** (for local variables, function arguments, return values) a copy of the **registers**, **program counter** and any **thread-specific data** to allow them to be scheduled individually.
- Other data **is shared within the process among all threads**

Threads (II)

- A thread is a semi-process that has its own stack, and **executes a given piece of code**.
- Unlike a real process, the thread normally shares its memory with other threads (where as for processes we usually have a different memory area for each one of them).
- A **thread group is a set of threads all executing inside the same process**.
- They all share the same memory, and thus can access the same global variables, same heap memory, same set of file descriptors, etc.
- All these **threads execute in parallel** (i.e. using time slices, or if the system has several processors, then really in parallel).

Threads (III)

- There are four major categories of benefits to multi-threading:
  - **Responsiveness**: One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.
  - **Resource sharing**: By default threads share common code, data, and other resources, which allows multiple tasks to be performed simultaneously in a single address space.
  - **Economy**: Creating and managing threads (and context switches between them) is much faster than performing the same tasks for processes.
  - **Scalability**, i.e. Utilization of multiprocessor architectures: A **single threaded process can only run on one CPU**, no matter how many may be available, whereas the execution of a multi-threaded application may be split amongst available processors.

Processes *vs* Threads (I)

- Remember that `fork` produces a second copy of the calling process.
  - The **parent and the child are completely independent**, each with its own address space, with its own copies of its variables, which are completely independent of the same variables in the other process.
- Threads **share a common address space**, thereby avoiding a lot of the inefficiencies of multiple processes.
  - The OS does not need to make a new independent copy of the process memory space, file descriptors, etc.
    - This saves a lot of CPU time, making **thread creation ten to a hundred times faster than a new process creation**.
    - **Less time to terminate** a thread than a process.
    - **Context switching between threads is much faster** then context switching between processes (context switching means that the system switches from running one thread or process, to running another thread or process)

Processes *vs* Threads (II)

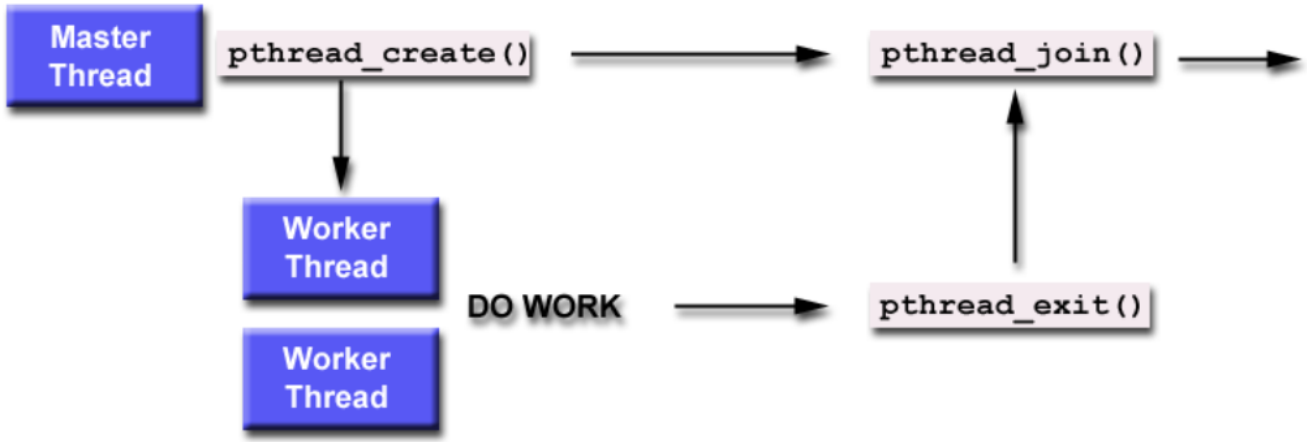
Processes	Threads
Processes are heavyweight operations.	Threads are lighter-weight operations.
Each process has its own memory space.	Threads use the memory of the process they belong to.
Inter-process communication is slow as processes have different memory addresses.	Inter-thread communication can be faster than inter-process communication because threads of the same process share memory with the process they belong to.
Context switching between processes is more expensive.	Context switching between threads of the same process is less expensive.
Processes don't share memory with other processes.	Threads share memory with other threads of the same process.

POSIX Thread APIs

Introduction

- Portable Operating System Interface (POSIX) defines set of Application Programming Interface(API), function, header file for threaded programming known as POSIX threads or Pthreads.
- The pthread library is a standard based thread API for C/C++.
- When you are using the functions of the pthread library, you must ensure the compiler links the pthread library into your executable, through the option `-lpthread`.
  - `gcc -o <obj> <source>.c -lpthread`

Lifecycle



- The master/main thread creates a pool of child/worker threads by invoking through `pthread_create` function.
- Then, the main thread calls `pthread_join` for waiting to child threads termination.
  - by terminating the execution of the routine
  - By invoking `pthread_exit` function (in this case it is possible to specify a termination status)
- When child threads finish their work the main thread continues.

Thread Creation and Termination

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_THREADS 2
void *do_work( void *ptr ){
  int i;
  double result=0.0;
  for (i=0; i<1000000; i++){
    result = result + (double)random();
  }
  pthread_exit((void *) 0);
}
int main (int argc, char *argv[]){
   pthread_t thread[NUM_THREADS];
   int t, status;
   for(t=0; t<NUM_THREADS; t++){
     pthread_create(&thread[t], NULL, do_work, NULL);
   }
   for(t=0; t<NUM_THREADS; t++){
     pthread_join(thread[t], (void **)&status);
     printf("Completed join with thread %d status= %d\n",t, status);
   }
   pthread_exit(NULL);
  }
}
```

pthread_create

- `int pthread_create(pthread_t * th_id, const pthread_attr_t * attr, void * (*start_routine)(void *), void *arg)`
  - `pthread_t *th_id`: Each thread is identified by id known as thread identifier, so this variable(pointer) points to address which holds the identifier for the new created thread.
  - `pthread_attr_t *attr`: The `attr` argument points to `pthread_attr_t` structure whose contents are used at the time of creation of thread.
    - It `attr` is `NULL`, then the thread is created with default values.
  - `void * (*start_routine)`: Pointer to the function to be threaded.
    - Function has a single argument: `void *`.
  - `void * arg`: Pointer to argument of function.
    - To pass multiple arguments, send a pointer to a structure.
  - Return value: It returns 0 on success and an error number of failure.

Check: `https://man7.org/linux/man-pages/man3/pthread_create.3.html`

pthread_exit

- `void pthread_exit(void *retval);`
  - `retval`: Return value of thread.
- This routine kills the thread.
- The `pthread_exit` function never returns.
- If the thread is not detached, the return value may be examined from another thread by using `pthread_join`.
  - The return pointer `*retval`, must not be of local scope otherwise it would cease to exist once the thread terminates.

Check: `https://man7.org/linux/man-pages/man3/pthread_exit.3.html`

pthread_join

- Allows the calling thread to wait for the ending of the target thread.
- `int pthread_join(pthread_t th_id, void **retval)`
  - `pthread th_id`: It is the thread to wait for that is the thread identifier filled while creating `pthread_create`.
  - `void **retval`: If `retval` is not `NULL`, then `pthread_join` copies the exit status of the target thread that is the value given by `pthread_exit` into the location pointed by `*retval`.
  - Return value: It returns 0 in success and error number in the case of failure.

Check: `https://man7.org/linux/man-pages/man3/pthread_join.3.html`

Attributes (I)

- Attributes are a way to specify behavior that is different from the default.
- When a thread is created with `pthread_create` or when a synchronization variable is initialized, an attribute object can be specified.
  - Attributes are specified only at thread creation time; they cannot be altered while the thread is being used.

Attributes (II)

- For using attributes, four functions are usually called:
  - Thread attibute intialisation: `pthread_attr_init` initialize a default `pthread_attr_t` struct variable;
  - Thread attribute value change (unless defaults appropriate): a variety of `pthread_attr_*` functions are available to set individual attribute values for the `pthread_attr_t` struct variable.
  - Thread creation: A call to `pthread_create` with appropriate attribute values set in a `pthread_attr_t` struct variable.
  - Attribute destroy: When a thread attributes struct variable is no longer required, it should be destroyed using the `pthread_attr_destroy` function

Check: `https://man7.org/linux/man-pages/man3/pthread_attr_init.3.html`

Attributes (IV)

```
...
 void *BusyWork(void *null){
   ...
 }
 int main (int argc, char *argv[])
 {
  pthread_t thread[NUM_THREADS];
  pthread_attr_t attr;
  int t, status;
  /* Initialize and set thread detached attribute */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  for(t=0; t<NUM_THREADS; t++){
    pthread_create(&thread[t], &attr, BusyWork, NULL);
  }
  /* Free attribute and wait for the other threads */
  pthread_attr_destroy(&attr);
  for(t=0; t<NUM_THREADS; t++){
    pthread_join(thread[t], (void **)&status);
    printf("Completed join with thread %d status= %d\n",t, status);
  }
  pthread_exit(NULL);
 }
```

Check: https://www.ibm.com/docs/en/i/7.3?topic=ssw_ibm_i_73/apis/rzah4mst.html