Advanced System Programming

Agenda

- Thread Management
- IPC

Threads

- Threads are mechanisms to do more than one job at a time.

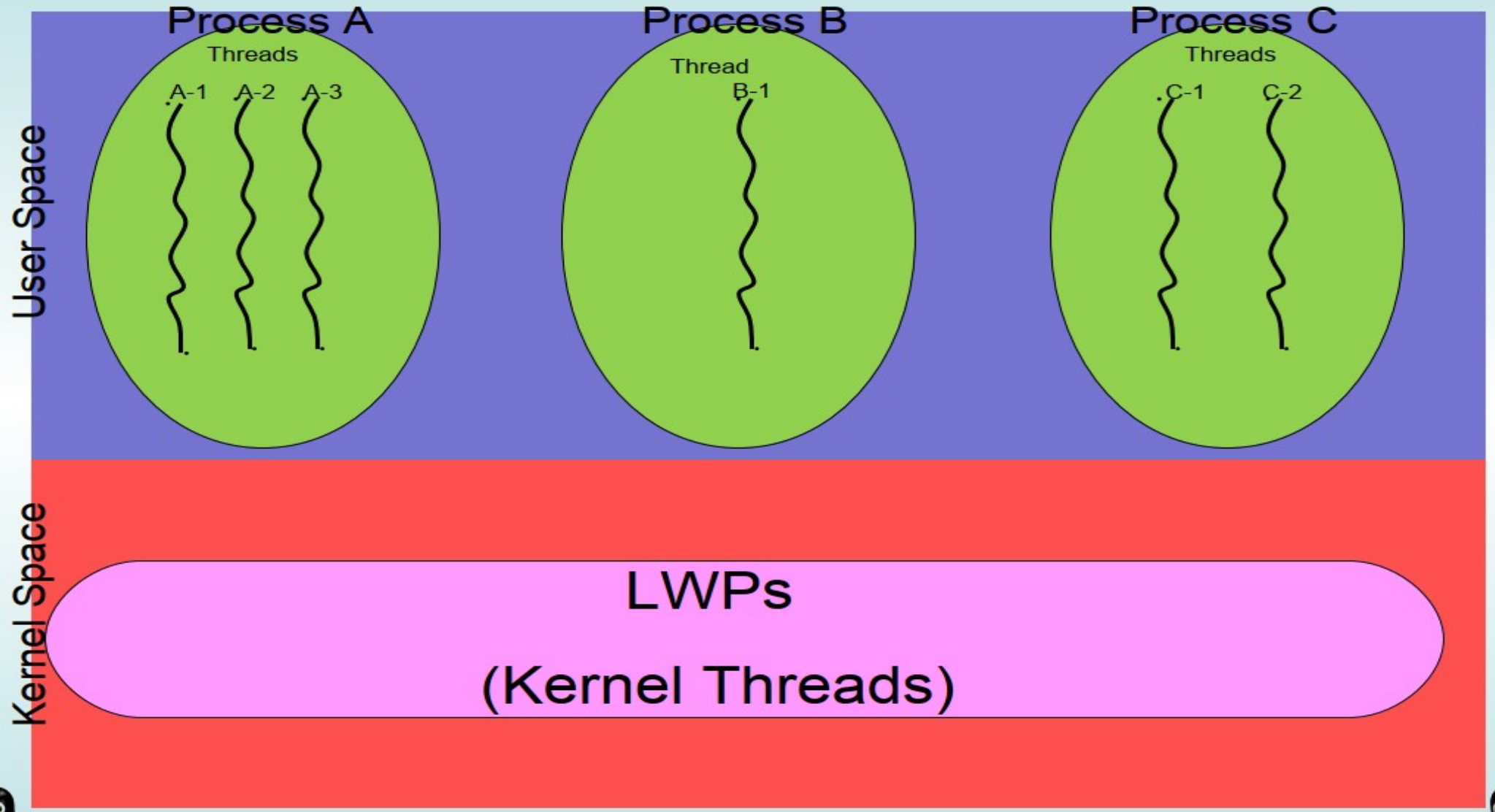
- Threads are finer-grained units of execution.

- Threads, unlike processes, share the same address space and other resources.

- POSIX standard thread API is not included in standard C library, they are in *libpthread.so*.

- In Linux, threads are handled by LWPs.

Thread Management()



Thread Management()

The primary motivation behind Pthreads is improving program performance.

Can be created with much less OS overhead.

Needs fewer system resources to run.

View comparison of forking processes to using a pthreads_create subroutine. Timings reflect 50,000 processes/thread creations.

Threads vs Forks

PLATFORM	fork()			pthread_create()		
	REAL	USER	SYSTEM	REAL	USER	SYSTEM
AMD 2.4 GHz Opteron (8cpus/node)	41.07	60.08	9.01	0.66	0.19	0.43
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.24	30.78	27.68	1.75	0.69	1.1
IBM 1.5 GHz POWER4 (8cpus/node)	104.05	48.64	47.21	2.01	1	1.52
INTEL 2.4 GHz Xeon (2 cpus/node)	54.95	1.54	20.78	1.64	0.67	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.54	1.07	22.22	2.03	1.26	0.67

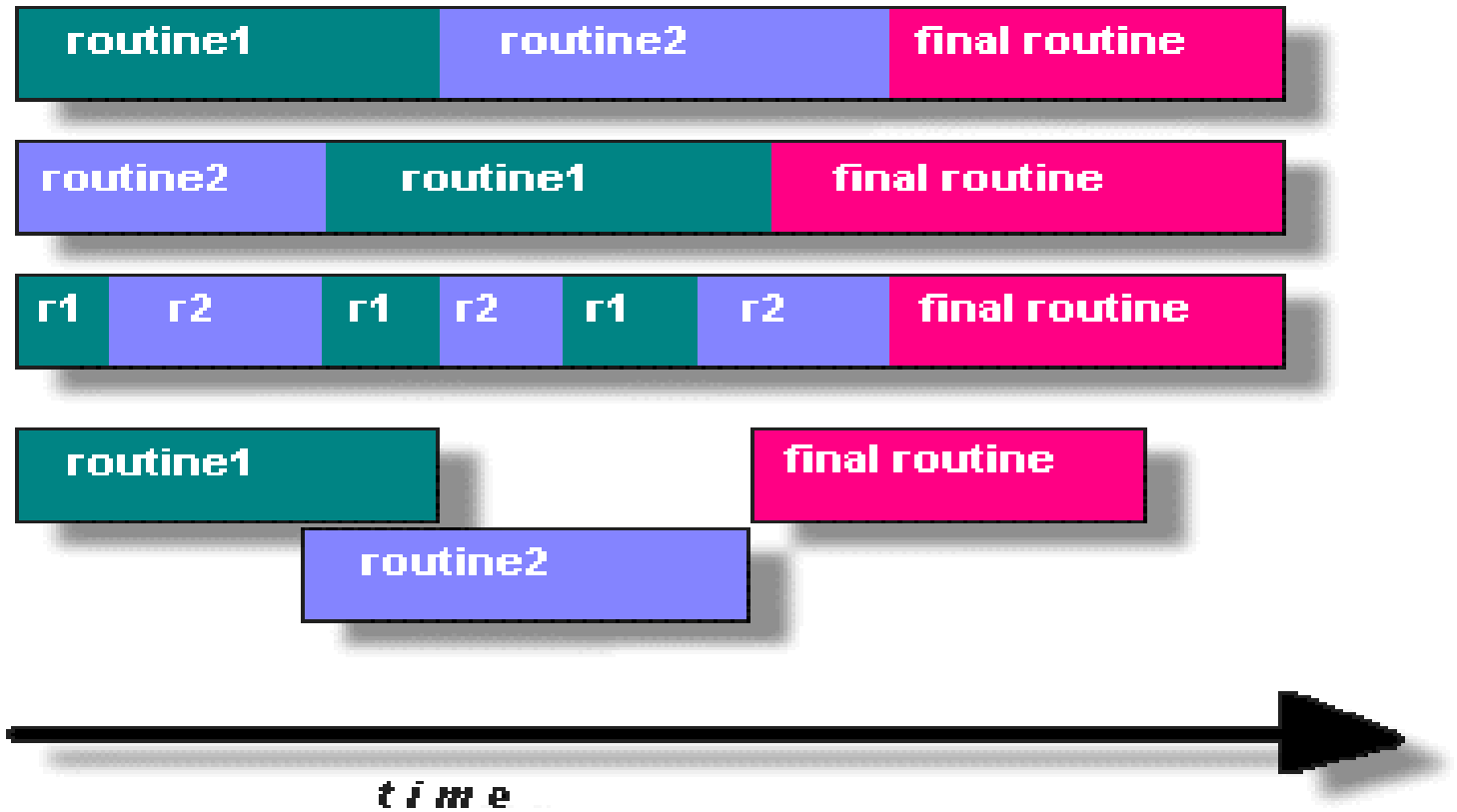
Thread Management()

Designing Pthreads Programs

Pthreads are best used with programs that can be organized into discrete, independent tasks which can execute concurrently.

Example: routine 1 and routine 2 can be interchanged, interleaved and/or overlapped in real time.

Candidates for Pthreads

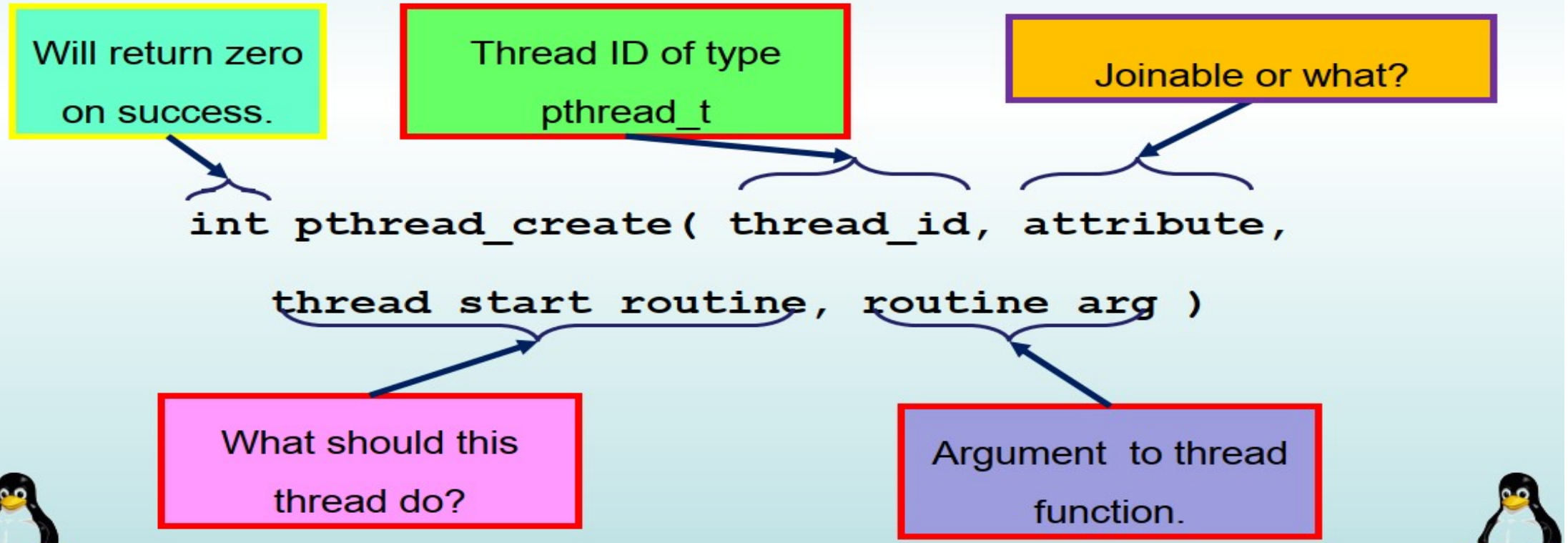


Thread Management() – Creation()

Creating threads

- Like processes, each thread has its own Thread-ID of type *pthread_t.*

- You can create a thread bye calling the *pthread_create* function.



Thread Management() – Creation()

Creating threads

- *pthread_create* returns immediately and the specified thread will do its job separately.

- If one of the threads in a program, call *exec* the whole process image will be replaced.

- The argument passed to the thread routine is a *void* * .

- You can pass more data in a structure of type *void* *.

Thread Management() – Join()

Joining threads

- You can wait for a thread to finish its job using *pthread_join*.

- *pthead_join* is something similar to *wait* function in processes.

- Using *pthread_join*, you can also take the return value of a thread.

- A thread, can not call *pthread_join* to wait for itself, you can use *pthread_self* function to get the TID of running thread and deciding what to do.

Thread Management() – Join()

Joining threads

- Like processes, you can wait for a thread to finish its job…

```
int pthread_join( pthread_t thread_id, void ** return_value )
```

Will return zero on success.

Thread ID which you want to wait for.

The return value of thread will be put here.

Thread Management() – exit()

Several ways to terminate a thread:

- The thread is complete and returns
- The pthread_exit() method is called
- The pthread_cancel() method is invoked
- The exit() method is called

The pthread_exit() routine is called after a thread has completed its work and it no longer is required to exist.

If the main program finishes before the thread(s) do, the other threads will continue to execute if a pthread_exit() method exists.

The pthread_exit() method does not close files; any files opened inside the thread will remain open, so cleanup must be kept in mind.

Thread Management() – pthread_attributes()

Thread attributes

- Second parameter in *pthread_create* is the thread attribute.

- Most useful attribute of a thread is *joinability*.

- If a thread is *joinable*, it is not automatically cleaned up.

- To clean up a *joinable* like a child process, you should call *pthread_join* .

- A *detached* thread, is automatically cleaned up.

- A joinable thread may be turned into a detached one, but can not be made joinable again.

- Using *pthread_detach* you can turn a joinable thread into detached.

Thread Management() – pthread_attributes()

Thread attributes

- If you do not clean up the joinable thread, it will become something like zombie.

- To assign an attribute to a thread, you should:
  - Create a *pthread_attr_t* object.
  - Call *pthread_attr_init* to initialize the attribute object.
  - Modify the attributes.
  - Pass a pointer to *pthread_create*.
  - Call *pthread_attr_destroy* to release the attribute object.

Thread Management() – pthread_cancel()

Thread cancelation

- A thread might be terminated by finishing its job or calling *pthread_exit* or by a request from another thread.

- The latter case is called “Thread Cancelation”.

- You can cancel a thread using *pthread_cancel*.

- If the canceled thread is not detached, you should join it after cancelation, otherwise it will become zombie.

- You can disable cancelation of a thread using *ptherad_setcancelstate()*.

Thread Management() – pthread_cancel()

Thread cancelation

- There are two cancel state:

- **PTHREAD_CANCEL_ASYNCHRONOUS:** Asynchronously cancelable (cancel at any point of execution)

- **PTHREAD_CANCEL_DEFERRED:** Synchronously cancelable (thread checks for cancellation requests)

- There are two cancelation types:

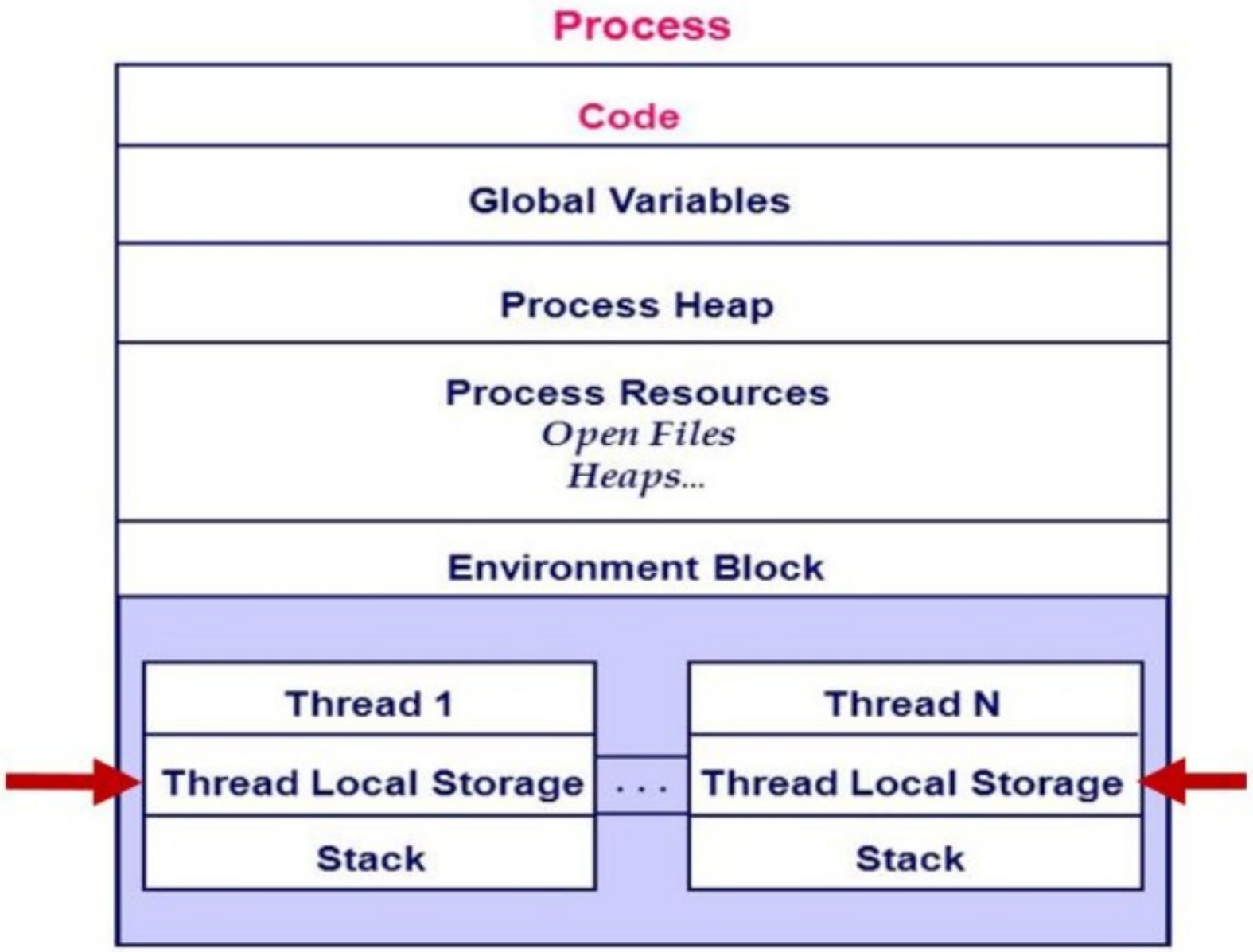
- **PTHREAD_CANCEL_DISABLE** and **PTHREAD_CANCEL_ENABLE.**

- It's a good idea to set the state to *Uncancelable* when entering critical section…

Thread-Local Storage

- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to `static` data
  - TLS is unique to each thread

Thread Management() – pthread_cancel()



IPC

- Inter Process Communication (IPC) is transfer of data between processes.

- In Linux there are some methods of IPC:

  - Shared Memory.
  - Mapped Memory.
  - Pipe (Named and Unnamed).
  - Socket (Remote, Local).

Pipes

- A pipe is a communicational device that permits unidirectional communication.

- The first data written into pipe is the first one that is read.

- If the writer, writes faster than the reader and pipe is full, the writer blocks.

- If the reader reads tries to read an empty pipe, it blocks.

- You can create pipes using *pipe()*.

Pipes

On success, 0 is returned and on error, -1 is returned and errno is set.

An array of size 2, which contains read and write file descriptors.

```
int  pipe ( int filedes[2] );
```

- *pipe()* stores the reading file descriptor in array position 0 and the writing file descriptor in position 1.

- Read and write file descriptors are available only in calling process and its children.

- You can use pipes to communicate between threads in a process.

IPC - Pipe

Pipes

The file descriptor related to created process stdin (or stdout).

The command you wish to execute.

Might be “w” or “r” as for writing or reading.

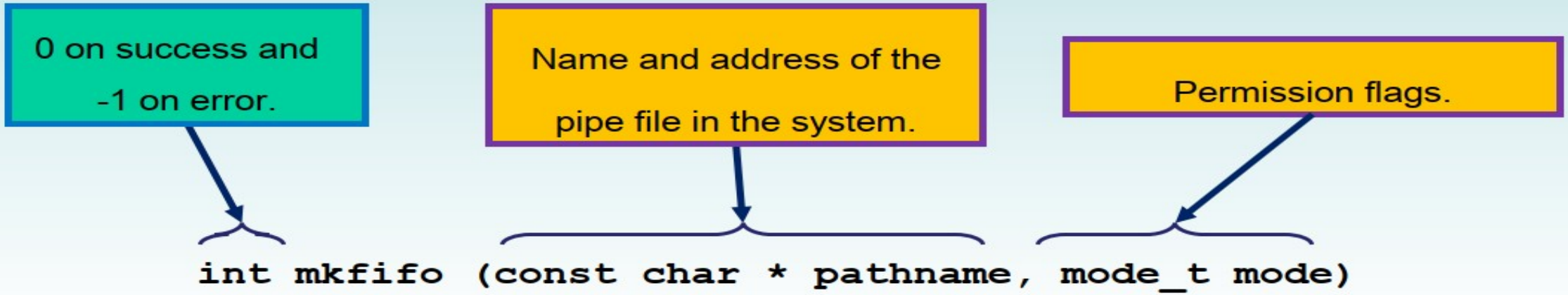
```
FILE * popen( const char * command, const char * type)
```

- You can use *popen()* to send data to or receive data from a program running in a subprocess.

- After closing the stream (using *pclos()*), *pclose()* waits for the child process to terminate.

IPC – Named Pipe

Named Pipes



- You can access a named pipe like an ordinary file.

- One program must open it for writing and another for reading.

Thank you