DEVLINUX
ACADEMY

LINUX THREAD

“Code Your Embedded Future”

https://devlinux.vn

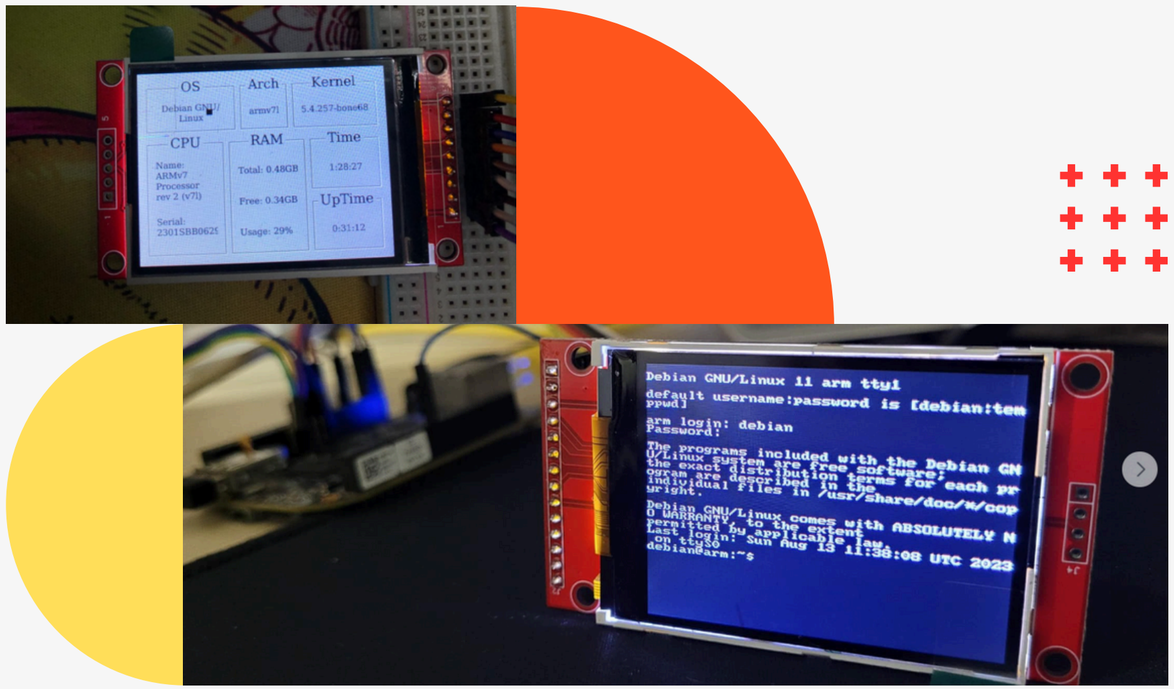
Cộng đồng lập trình Nhúng & Vi Mạch

SUBSCRIBE @devlinux097

DEVLINUX ACADEMY

"Code Your Embedded Future"

AGENDA



- I. Introduction
- II. Compare Between Process & Thread
- III. Operations on Thread
- IV. Thread Management
- V. Thread Synchronization
- VI. Thread Sync - Mutex Lock
- VII. Thread Sync - Conditional Variable

https://devlinux.vn

Cộng đồng lập trình Nhúng & Vi Mạch

SUBSCRIBE @devlinux097

DEVLINUX
ACADEMY

INTRODUCTION

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

“Code Your Embedded Future”

What is Thread?

Similar to processes, threads are created for the purpose of processing multiple tasks at the same time (mutil-task).

Thread

A thread is a light-weight process that can be managed independently by the scheduler.

Process

A process is a program that is executing and using system resources.

Thread excuting

On a mutil-core system. Multiple threads can work in parallel.

If one thread is blocked, the other threads will still work normally.

Every time a thread is created, they are placed in stack segments.

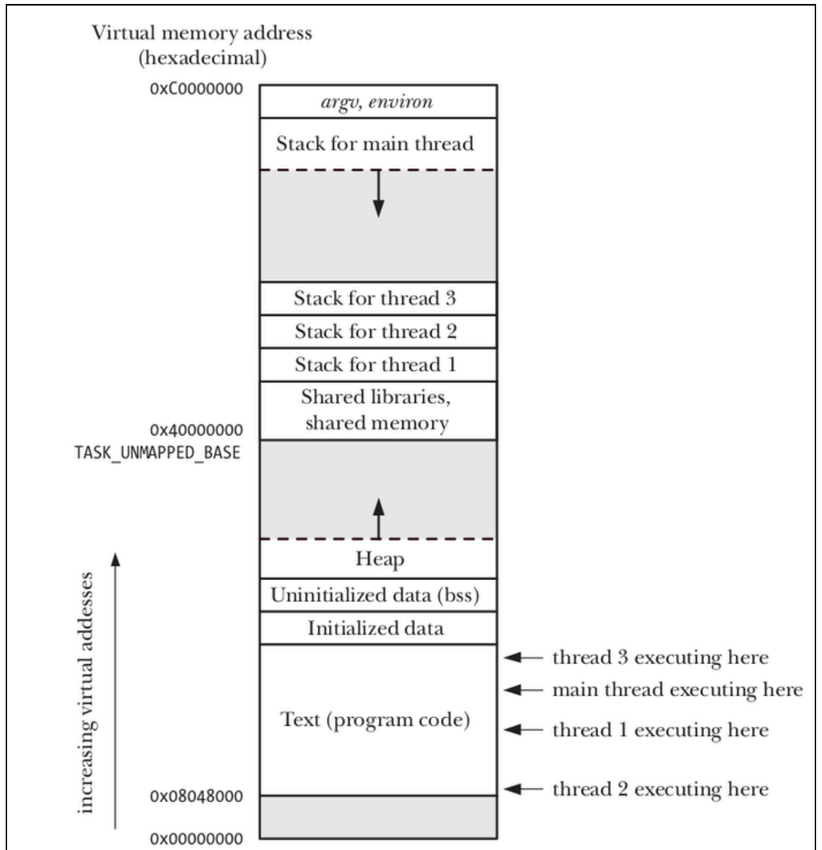


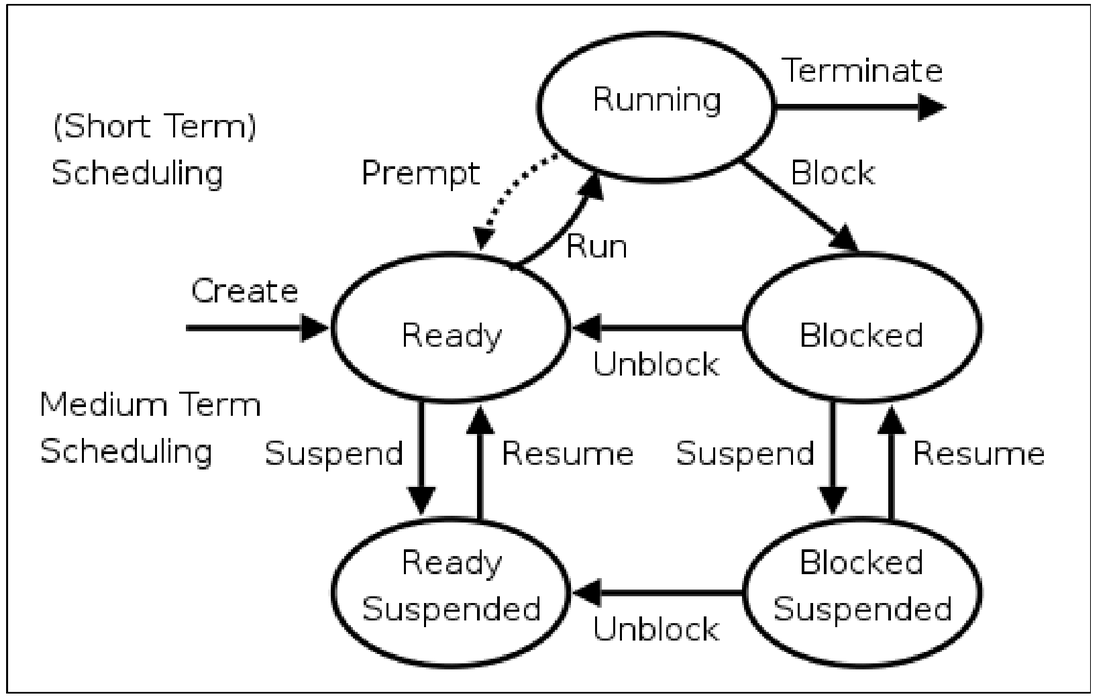
Figure 29-1: Four threads executing in a process (Linux/x86-32)

Context Switching

When a process is removed from access to the processor, sufficient information on its current operating state must be stored. Because when it is again scheduled to run on the processor it can resume its operation from an identical position.

This operational state data is known as its context. The context of a process includes its address space, stack space, virtual address space, register set image (e.g. Program Counter (PC), Stack Pointer (SP).

The act of removing the process of execution from the processor (and replacing it with another) is known as a process switch or context switch.



DEVLINUX
ACADEMY

COMPARE BETWEEN THREAD & PROCESS

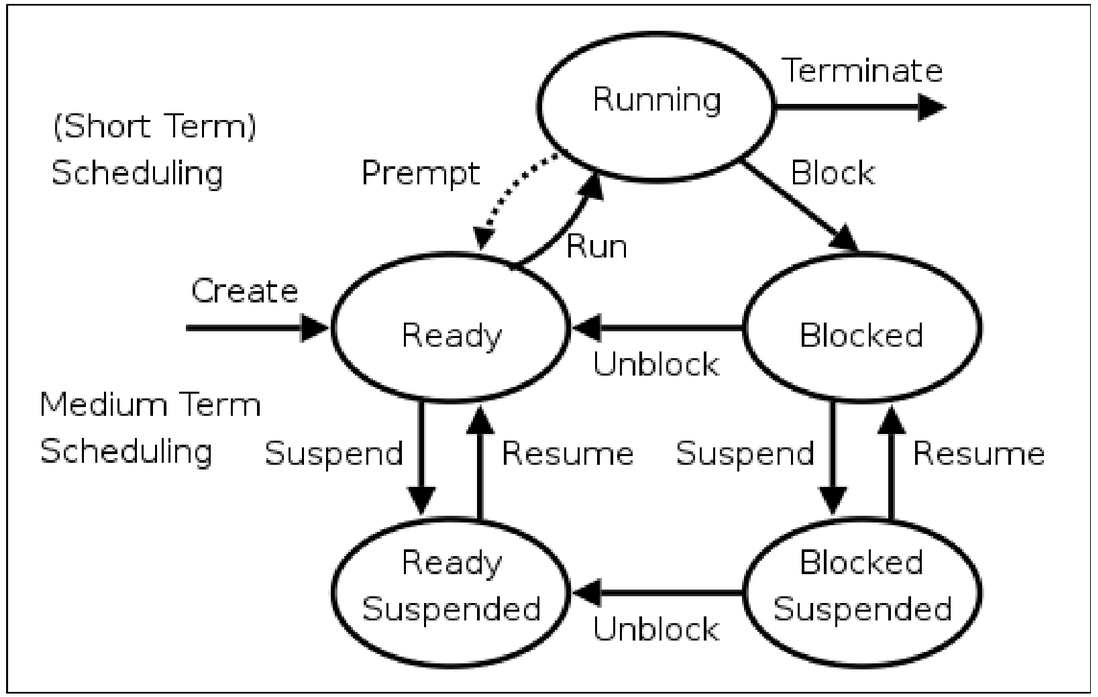
https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

“Code Your Embedded Future”

Context Switching Time

Processes take longer because they use more resources.

Threads need less time because they are lighter than processes.



Shared Memory

When creating a process with fork(), the process and child process reside in two different allocated memory areas. Sharing data between them becomes more difficult.

Sharing data between threads in a process is faster and easier because they reside in the memory address space of process.

shared memory

Sharing data between processes is more difficult, through the IPC mechanism
Threads in a process share data with each other quickly.

Crashed

When creating a process with fork(), the process and its child are on two different allocated memory areas, operating independently of each other. When one process fails, the other process still executes normally.

Threads on a process work concurrently. When one thread fails, other threads will terminate.

Crashed

If one process fails, other processes continue to execute normally.
If one thread fails, the remaining threads in the process will stop immediately.

DEVLINUX
ACADEMY

OPERATION ON THREAD

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

“Code Your Embedded Future”

Thread ID

Just as a process is identified by the process ID, a thread in the process is identified by the thread ID.
Some points need to be clarified:

- Process IDs are unique throughout the system, where thread IDs are unique within a process.
- The process ID is an integer value but the thread ID is a structure.
- Process ID can be obtained very easily while thread ID cannot.

The thread ID will be represented by type pthread_t.
To compare two thread IDs with each other. We need a function that can do the job (For an integer process ID the comparison is simple than).

- pthread_self()
- pthread_equal()

Thread ID

pthread_equal()

```c
#include <pthread.h>

/*
 * @return Trả về 0 nếu tid1 khác tid2, khác 0 nếu tid1 = tid2.
 */
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

pthread_self()

```c
#include <pthread.h>

/*
 * @return Trả về thread ID của thread đang gọi pthread_self().
 */
pthread_t pthread_self(void);
```

Create Thread

To create a new thread, we use the pthread_create() function.

Every process has at least one thread. The thread containing the main function is called the main thread.

pthread_create()

```c
/*
 * @return Trả về 0 nếu thành công, trả về một số dương nếu lỗi.
 */
int pthread_create(pthread_t *restrict thread,
                   const pthread_attr_t *restrict attr,
                   typeof(void *(void *)) *start_routine,
                   void *restrict arg);
```

Terminate Thread

Thread ends normally.

Thread terminates when calling pthread_exit() function.

Thread is canceled when calling pthread_cancel() function.

Any thread that calls the exit() function, or the main thread terminates, all remaining threads terminate immediately.

pthread_exit()

```c
#include <pthread.h>

/*
 * @param[out] retval Giá trị trả về khi kết thúc thread.
 */
void pthread_exit(void *retval);
```

DEVLINUX
ACADEMY

THREAD MANAGEMENT

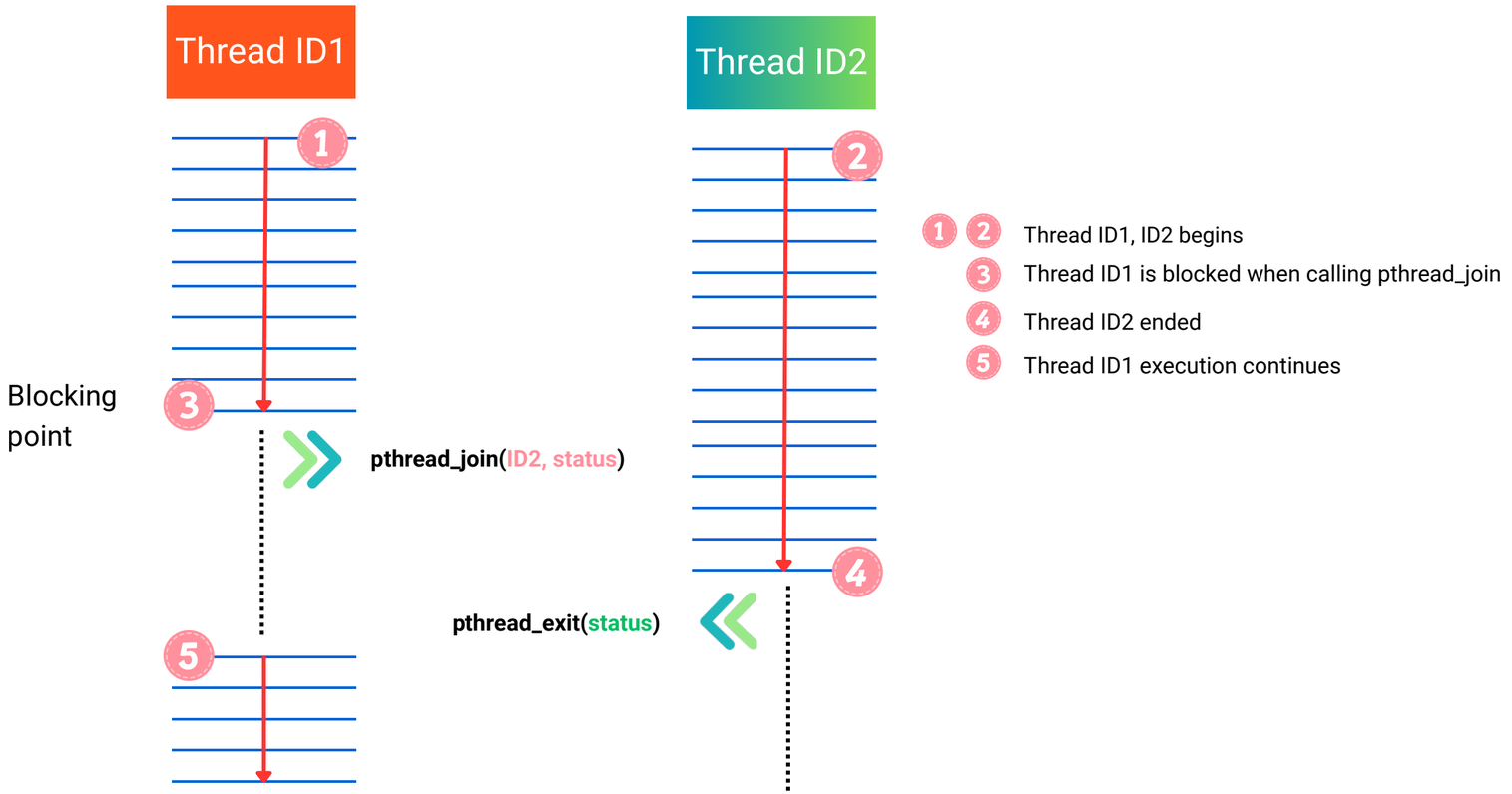
https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

“Code Your Embedded Future”

Thread joining

To get return value of another thread, we use pthread_join().

This activity is called joining.



pthread_join()

int pthread_join(pthread_t thread, void **retval)

pthread_join() will block until a thread terminates. If the thread has already terminated, pthread_join will return immediately.

When the thread terminates, it will be treated similarly to a zombie process. If the number of zombie threads increases too much, at some point the system will not be able to create new threads anymore. The role of pthread_join() is similar to the waitpid() function for processes.

Parameters:

thread: ID of the specific thread to wait for.

**retval: If retval is not NULL, it will receive the return value from the thread's pthread_exit()

Thread Detaching

Detached Thread

By default, a thread is joinable, meaning that when a thread terminates, another thread can take the return value of that thread through the pthread_join() function.

However, in many cases, we do not need to care about the return value of the thread, but only need the system to automatically release the thread after it terminates.

In this case, we can put the thread into detached state by calling the pthread_detach() function.

pthread_detach()

int pthread_detach(pthread_t thread);

Once a thread has been detached, pthread_join() cannot be used to retrieve the return value of that thread, and the thread cannot return to the joinable state.

Parameters:

- thread: The ID of the specific thread.

Returns 0 on success, less than 0 on failure.

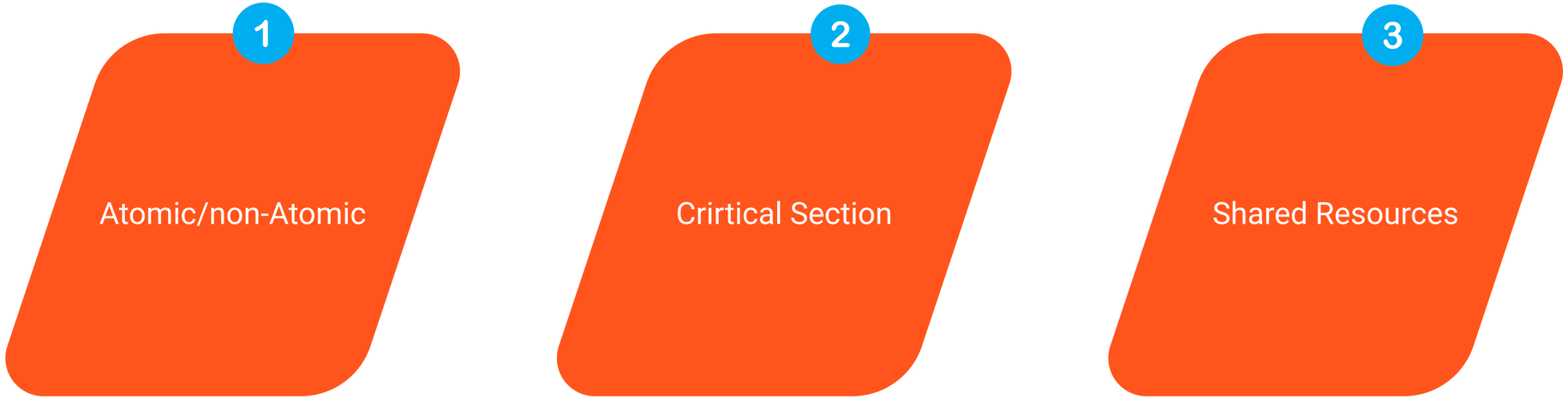
DEVLINUX
ACADEMY

THREAD SYNCHRONIZTION

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

“Code Your Embedded Future”

Related Concepts



- Atomic: Only 1 thread can access the shared data area at a time
- Non-Atomic: Multiple threads can access the shared data area at the same time

Shared data area usage time

Data area shared between two or more processes/threads

DEVLINUX
ACADEMY

THREAD SYNC - MUTEX LOCK

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

“Code Your Embedded Future”

Mutex Lock

Mutual Exclusion

Mutex is a technique used to ensure that only one thread can access a shared resource at a time.

There are three steps to implementing a mutex:

- Initialize the muxtex lock
- Implement the mutex lock for the shared resource before accessing the critical section
- Release the mutex after the access is complete

Allocate Mutex

The mutex lock is a pthread_mutex_t data type. Before using it, we always have to initialize the mutex lock.

The mutex lock can be allocated statically or dynamically:

- pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
- int pthread_mutex_init(pthread_mutex_t *mutex, constpthread_mutexattr_t *attr);

When no longer in use, we have to destroy the mutex using the pthread_mutex_destroy() function. Static initialization does not require calling this function.

Locking & Unlocking Mutex

int pthread_mutex_lock(pthread_mutex_t *mutex);

When the mutex is unlocked, pthread_mutex_lock() returns immediately. Conversely, if the mutex is locked by another thread, pthread_mutex_lock() is blocked until the mutex is unlocked.

Parameters:

- *mutex: pointer to the mutex object.

  Returns 0 if successful, and less than 0 if unsuccessful.

Locking & Unlocking Mutex

int pthread_mutex_unlock(pthread_mutex_t *mutex);

Function to unlock a mutex

Parameters:

- *mutex: pointer to the mutex object.

Returns 0 if successful, and less than 0 if failed.

DEVLINUX
ACADEMY

THREAD SYNC - CONDITIONAL VARIABLE

https://devlinux.vn
Cộng đồng lập trình Nhúng & Vi Mạch
SUBSCRIBE @devlinux097

“Code Your Embedded Future”

Conditional Variable

Conditional Variable

A mutex is a technique used to ensure that only one thread has access to a shared resource at a time.

A condition variable is used to notify another thread when there is a change in the shared resource and allows another thread to be blocked until it receives that notification.

Allocate Conditional Variable

Conditional Variable

Condition variable is a data type pthread_cond_t. Before using, we must always initialize the condition variable.

Condition variables can be statically or dynamically allocated:

- pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
- int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);

When no longer in use, we need to destroy the condition variable using the pthread_cond_destroy() function. If the condition variable is statically initialized, this function does not need to be called.

Signaling & Waiting

Signaling & Waiting

The two main operations of condition variables are signal and wait.

```c
#include <pthread.h>


int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

DEVLINUX
ACADEMY

Thanks for Your Attention!

Liên hệ với chúng tôi

https://devlinux.vn

Cộng đồng lập trình Nhúng & Vi Mạch

SUBSCRIBE @devlinux097