POSIX Threads
The POSIX IEEE standard

✓ Specifies an operating system interface similar to most UNIX systems
  
  – *It extends the C language with primitives that allows the specification of the concurrency*

✓ POSIX distinguishes between the terms process and thread
  
  – "A **process** is an address space with one or more threads executing in that address space"
  
  – "A **thread** is a single flow of control within a process (a unit of execution)"

✓ Every process has at least one thread
  
  – the “**main()**” (aka "**master**") thread; its termination ends the process
  
  – All the threads **share** the same address space, and have a **private** stack
The pthread primitives are usually implemented into a pthread library

All the declarations of the primitives cited in these slides can be found into sched.h, pthread.h and semaphore.h
  - Use man to get on-line documentation

When compiling under gcc & GNU/Linux, remember the -lpthread option!
PThread creation, join, end
A (P)thread is identified by a C function, called body:

```c
void *my_pthread_fn(void *arg)
{
    ...
}
```

A thread starts with the first instruction of its body.

The threads ends when the body function ends:
- It's not the only way a thread can die.
Thread creation

✓ Thread can be created using the primitive

```c
int pthread_create ( pthread_t *ID,
                   pthread_attr_t *attr,
                   void *(*body)(void *),
                   void * arg
);
```

✓ `pthread_t` is the type that contains the thread ID
✓ `pthread_attr_t` is the type that contains the parameters of the thread
✓ `arg` is the argument passed to the thread `body` when it starts
Thread attributes

✓ Thread attributes specifies the characteristics of a thread
  – Stack size and address
  – Detach state (joinable or detached)
  – Scheduling parameters (priority, …)

✓ Attributes must be initialized and destroyed
  – int pthread_attr_init(pthread_attr_t *attr);
  – int pthread_attr_destroy(pthread_attr_t *attr);
Thread termination

✓ A thread can terminate itself calling

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

✓ When the thread body ends after the last “}”, `pthread_exit()` is called implicitly

✓ Exception: when `main()` terminates, `exit()` is called implicitly
Thread IDs

✓ Each thread has a unique ID

✓ The thread ID of the current thread can be obtained using

```c
pthread_t pthread_self(void);
```

✓ Two thread IDs can be compared using

```c
int pthread_equal(   pthread_t thread1,
                    pthread_t thread2   );
```
A thread can wait the termination of another thread using

```c
int pthread_join ( pthread_t th,
                 void **thread_return);
```

It gets the return value of the thread or `PTHREAD_CANCELED` if the thread has been killed

By default, every thread must be joined
- The join frees all the internal resources
- Stack, registers, and so on
A thread which does not need to be joined has to be declared as detached

2 ways to have it:
- While creating (in father thread) using `pthread_attr_setdetachstate()`
- The thread itself can become detached calling in its body `pthread_detach()`

Joining a detached thread returns an error
Example

- Filename: `ex_create.c`
- The demo explains how to create a thread
  - the `main()` thread creates another thread (called `body()`)
  - the `body()` thread checks the thread IDs using `pthread_equal()` and then ends
  - the `main()` thread joins the `body()` thread

- Credits to PJ
Pthread cancellation
A thread can be killed calling

```c
int pthread_cancel(pthread_t thread);
```

When a thread dies its data structures will be released

- By the join primitive if the thread is joinable
- Immediately if the thread is detached
- Why?

Killing a thread
✓ Specifies how to react to a kill request
✓ There are two different behaviors:
  – deferred cancellation
    when a kill request arrives to a thread, the thread does not die. The thread will die only when it will execute a primitive that is a cancellation point. This is the default behavior of a thread.
  – asynchronous cancellation
    when a kill request arrives to a thread, the thread dies. The programmer must ensure that all the application data structures are coherent.
The user can set the cancellation state of a thread using:

```c
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
```

The user can protect some regions providing destructors to be executed in case of cancellation

```c
int pthread_cleanup_push(void (*routine)(void *), void *arg);
int pthread_cleanup_pop(int execute);
```
The cancellation points are primitives that can potentially block a thread

When called, if there is a kill request pending the thread will die
- void pthread_testcancel(void);
- sem_wait, pthread_cond_wait, printf and all the I/O primitives are cancellation points
- pthread_mutex_lock, is NOT a cancellation point
  • Why?

A complete list can be found into the POSIX Standard
The user must guarantee that when a thread is killed, the application data remains coherent.

The user can protect the application code using cleanup handlers:
- A cleanup handler is an user function that cleans up the application data.
- They are called when the thread ends and when it is killed.
Cleanup handlers (2)

```c
void pthread_cleanup_push (void (*routine)(void *),
    void *arg);
void pthread_cleanup_pop (int execute);
```

- They are pushed and popped as in a stack
- If `execute!=0` the cleanup handler is called when popped
- The cleanup handlers are called in LIFO order
Example

✓ Filename: ex_cancellation.c
✓ Highlights the behavior of:
  – Asynchronous cancellation
  – Deferred cancellation
✓ Explains the cleanup handlers usage
POSIX semaphores
Semaphores

✓ A semaphore is a counter managed with a set of primitives
✓ It is used for
  – Synchronization
  – Mutual exclusion
✓ POSIX Semaphores can be
  – Unnamed (local to a process)
  – Named (shared between processed through a file descriptor)
Unnamed semaphores

- Mainly used with multithread applications
- Operations permitted:
  - initialization / destruction
  - blocking wait / nonblocking wait
    - counter decrement
  - post
    - counter increment
  - counter reading
    - simply returns the counter
Initializing a semaphore

✔ The `sem_t` type contains all the semaphore data structures

```c
int sem_init(sem_t *sem, int pshared, unsigned int value);
```
- `pshared` is 0 if `sem` is not shared between processes

```c
int sem_destroy(sem_t *sem);
```
- It destroys the `sem` semaphore
Semaphore waits

```c
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
```

✓ Under the hood..
✓ If the counter is greater than 0 the thread does not block
  - `sem_trywait` never blocks
✓ `sem_wait` is a cancellation point
Other semaphore primitives

```c
int sem_post(sem_t *sem);
```

- It increments the semaphore counter
- It unblocks a waiting thread

```c
int sem_getvalue(sem_t *sem, int *val);
```

- It simply returns the semaphore counter
Example

✓ Filename: `ex_sem.c`

✓ In this example, semaphores are used to implement mutual exclusion in the output of a character in the console.
PThread mutexes
What is a POSIX mutex?

✓ Like a *binary semaphore used for mutual exclusion*
  – But.. a mutex can be unlocked *only* by the thread that locked it

✓ Mutexes also support some RT protocols
  – Priority inheritance
  – Priority ceiling
  – They are not implemented under a lot of UNIX OS

✓ Out of scope for this course
Mutex attributes

✓ Mutex attributes are used to initialize a mutex

```c
int pthread_mutexattr_init (pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy (pthread_mutexattr_t *attr);
```

✓ Initialization and destruction of a mutex attribute
Mutex initialization

✓ Initialize a mutex with a given mutex attribute

```c
int pthread_mutex_init (pthread_mutex_t *mutex,
                      const pthread_mutexattr_t *attr);
```

✓ Destroys a mutex

```c
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```
This primitives implement the blocking lock, the non-blocking lock and the unlock of a mutex

The mutex lock is NOT a cancellation point

```c
int pthread_mutex_lock(pthread_mutex_t *m);
int pthread_mutex_trylock(pthread_mutex_t *m);
int pthread_mutex_unlock(pthread_mutex_t *m);
```
Example

✓ Filename: `ex_mutex.c`
✓ This is prev. example written using mutexes instead of semaphores.
pthread condition variables
What is a POSIX condition variable?

✔ Used to enforce synchronization between threads
  – A thread into a mutex critical section can wait on a condition variable
  – When waiting, the mutex is automatically released and locked again at wake up
  – The synchronization point have to be checked into a loop!
✓ Attributes are used to initialize a condition variable

```c
int pthread_condattr_init (pthread_condattr_t *attr);
int pthread_condattr_destroy (pthread_condattr_t *attr);
```

✓ These functions initialize and destroy a condition variable
 ✓ In order to be used, a condition variable must be initialized

```
int pthread_cond_init (pthread_cond_t *cond,
                     const pthread_condattr_t *attr)
```

 ✓ ...and destroyed when it is no more needed

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

✓ Every condition variable is implicitly linked to a mutex  
  – given a condition variable, the mutex parameter must always be the same
✓ The condition wait must always be called into a loop protected by a cleanup handler!!!
Cancellation and mutexes

- Mutexes are not cancellation points

- The condition wait is a cancellation point

- When a thread is killed while blocked on a condition variable, the mutex is locked again before dying
  - The mutex is left locked, and no thread can use it anymore!
  - We must protect the thread from a cancellation
  - A cleanup function that releases the mutex
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);

✓ These functions wakes up at least one (signal) or all (broadcast) the thread blocked on the condition variable
  – Nothing happens if no thread is blocked on the condition variable

✓ The thread should lock the associated mutex when calling these functions
Example

✓ Filename: `ex_cond.c`
✓ This is prev. examples written using simulated semaphores obtained using mutexes and condition variables
✓ A simulated semaphore is composed by a counter, a mutex and a condition variable
✓ The functions lock the mutex to work with the counter and use the condition variable to block