Real-Time Operating Systems (0_KRI)
The POSIX Standard II

Ivan Cibrario Bertolotti

IEIIT-CNR / Politecnico di Torino

Academic Year 2006-2007

Outline

Multithreading

IEEE Std 1003.1-2004 supports **multithreaded** applications. It defines the following classes of functions:

- Creation, termination and wait for thread termination.
- Forced termination **(cancellation)**.
- Scheduler control.
- Synchronization.
- Thread-specific data.
- Other functions.

Availability and Restrictions

- These functions are suitable for **soft** real-time applications, and are available on most modern real-time and general-purpose operating systems.
- Barring proprietary extensions to the standard, it is **not** possible to specify execution deadlines, periodic tasks, interrupt handlers and other items of interests for **hard** real-time systems.

Creating and Terminating a Thread

The most basic functions to create, terminate and wait for the termination of a thread are:

Nome	Funzione
`pthread_create`	Create a new thread
`pthread_exit`	Terminate the calling thread
`pthread_join`	Wait for thread termination
`pthread_cancel`	Forcibly terminate a thread

These function are similar to the Unix system calls `fork`, `exit`, `wait`, and `kill`, but handle threads instead of processes.

Creating a Thread

The following function creates a new thread:

```
int pthread_create(
  pthread_t *TH,
  pthread_attr_t *ATTR,
  void * (*START_ROUTINE)(void *),
  void * ARG);
```

- It creates a new thread that starts executing concurrently with the calling thread.
- The new thread executes the function **`START_ROUTINE`** with argument **`ARG`**.
- The **`ATTR`** attribute specifies the attributes of the new thread; a `NULL` value denotes a default set of attributes.
- When successful, this function stores into the location pointed by **`TH`** the identifier of the new thread and returns zero. Otherwise, it returns a non-zero error code.

Voluntary Thread Termination

A thread can **voluntarily** terminate its execution in two ways:

1. **implicitly**, by performing a `return` from its `START_ROUTINE`;
2. **explicitly**, by calling the `pthread_exit` function from anywhere in the code.

In the first case, the effect is the same as an explicit call to `pthread_exit` with an exit code set to the value returned by `START_ROUTINE`.

Thread Attributes (I)

The `ATTR` argument, specified during the creation of a new thread, allows the caller to specify the attributes of the thread being created. Some attributes deal with thread scheduling, and will be described later. Other interesting attributes are, for example:

Attribute	Value
`detachstate`	`PTHREAD_CREATE_JOINABLE` or `PTHREAD_CREATE_DETACHED` (default)
`stackaddr`	Address of the thread stack
`stacksize`	Size of the thread stack (> `PTHREAD_STACK_MIN`)
`stack`	Address & size of the thread stack
`guardsize`	Size of the guard region surrounding the thread stack

Thread Attributes (II)

- To set the attributes of a thread, one must first of all initialize a `pthread_attr_t` object that will hold them, by means of the function:

```
int pthread_attr_init(
  pthread_attr_t *ATTR);
```

- Then, it is possible to get and set each attribute through the pair of functions:

```
int pthread_attr_getattr(
    pthread_attr_t *ATTR, ...);

int pthread_attr_setattr(
    pthread_attr_t *ATTR, ...);
```

where **attr** is the name of the attribute to get/set.

Thread Attributes (III)

For example, the functions:

```
int pthread_attr_getdetachstate(
  pthread_attr_t *ATTR, int *STATE);

  int pthread_attr_setdetachstate(
  pthread_attr_t *ATTR, int STATE);
```

get and set the **`detachstate`** attribute.

Cleanup e Finalization

A thread voluntarily terminates by executing `return` from its `START_ROUTINE` or by calling `void pthread_exit(void *RETVAL)`.

In both cases, immediately before terminating, the thread still executes several "cleanup" actions previously associated with the thread itself, namely:

- It invokes the **cleanup handlers** registered by `pthread_cleanup_push`, in LIFO order.
- It invokes the **finalizers** associated with all thread-specific data items that are not `NULL`.

`RETVAL` is the exit code of the thread. It can be passed to another thread when it executes the `pthread_join` function. `pthread_exit` **never** returns to the caller.

Waiting for Thread Termination

The function:

```
int pthread_join(
  pthread_t TH,
  void **RETURN);
```

- Blocks the calling thread until the thread specified by **`TH`** terminates, either voluntarily or after a cancellation request.
- If **`RETURN`** is not `NULL`, this function also stores the exit code of the thread into the location pointed by `RETURN`.
- The exit code is set to the reserved value `PTHREAD_CANCELED` if the thread terminated after a cancellation request.
- This function returns zero when successful, a non-zero value on error.

Detached Threads and pthread_join

- By the standard, multiple `pthread_join` calls with the same target thread are not allowed and produce unspecified results. Moreover, the target thread must not be **detached**.
- When a non-detached thread terminates, the system keeps part of its resources, e.g. the thread descriptor, until another thread executes a `pthread_join` on it.
- To avoid memory leaks, be sure to execute `pthread_join` on all non-detached threads. Otherwise, detach all threads for which there are no good reasons to perform a join.

Cancellation (I)

- The **cancellation** mechanism allows a thread to request the **forced**, involuntary termination of another thread.
- The following function raises a cancellation request:

```
int pthread_cancel(
  pthread_t TH);
```

- When receiving a cancellation request, the target thread can react by:
  1. **ignoring** the request completely;
  2. terminate **immediately**;
  3. terminate as soon as it reaches a **cancellation point**.

Cancellation (II)

- When a thread terminates after a cancellation, the effect is the same as the invocation of `pthread_exit(PTHREAD_CANCELED)`, including the activation of its cleanup handlers and finalizers.
- There are many cancellation points:
  - `pthread_join`, `pthread_cond_wait`, `pthread_cond_timedwait`, `pthread_testcancel`, `sem_wait`.
  - Many system calls, for example `read` ans `write`.
  - All library functions that, either directly or indirectly, invoke these system calls.

Cancellation State and Type

- The function:

  ```
  int pthread_setcancelstate(
    int STATE,
    int *OLDSTATE);
  ```

  allows the calling thread to **ignore** (`STATE == PTHREAD_CANCEL_DISABLE`) or **honor** (`STATE == PTHREAD_CANCEL_ENABLE`) the cancellation requests it receives.
- The function:

  ```
  int pthread_setcanceltype(
    int TYPE,
    int *OLDTYPE);
  ```

  allows the calling thread to choose whether the cancellation requests it receives are honored **immediately** (`TYPE == PTHREAD_CANCEL_ASYNCHRONOUS`) or are **deferred** until the next cancellation point (`TYPE == PTHREAD_CANCEL_DEFERRED`).

Cleanup handlers – Registration

- Each thread can register through the function:

```
void pthread_cleanup_push(
   void (*ROUTINE) (void *),
   void *ARG);
```

  one or more functions, the **cleanup handlers**.
- `ROUTINE` is the function to be executed, with argument `ARG`.
- These functions will be called in LIFO order, when a thread is about to terminate (either voluntarily or involuntarily).
- Their main purpose is to ensure that any resource allocated by the thread during its life is correctly released upon termination, especially when the termination is involuntary.

Cleanup handlers – Removal

- It is possible to remove a cleanup handler even if the thread is not about to terminate, by means of the function:

```
void pthread_cleanup_pop(
   int EXECUTE);
```

- This function removes the cleanup handler that has been registered last. If the `EXECUTE` flag is not zero, it invokes the cleanup handler being removed, too.

Both `pthread_cleanup_push` and `pthread_cleanup_pop` may be implemented as macros. Hence, their use is subject to several **restrictions** described in the standard.

Scheduler Control

The thread **attributes** related with scheduler control are:

- `schedpolicy`: `SCHED_OTHER` (default, not real-time), `SCHED_RR` (round robin, real-time), `SCHED_FIFO` (first-in, first-out, real-time), ...
- `schedparam`: Scheduling parameters (priority, ...).
- `inheritsched`:
  - `PTHREAD_EXPLICIT_SCHED`: the thread is scheduled according to `schedpolicy` and `schedparam` (default);
  - `PTHREAD_INHERIT_SCHED`: the thread **inherits** its scheduling parameters from its father.
- `scope`:
  - `PTHREAD_SCOPE_SYSTEM`: the thread priority is relative to all other threads in the **system** (default);
  - `PTHREAD_SCOPE_PROCESS`: the thread priority is relative to the other threads belonging to the same **process**.

Mutex Devices

Definition

A *mutex* is a specialized semaphore, whose initial (and maximum) value is 1.

- It is useful to implement mutual exclusion in the accesses to shared data.
- In other words, a mutex can be in one of two possible states:

  unlocked: no threads “own” the mutex.
  locked: a thread “owns” the mutex.

When a thread tries to acquire a mutex owned by another thread, it shall wait.

Mutex Initialization and Destruction

- The function:
```
int pthread_mutex_init(
   pthread_mutex_t *MUTEX,
   const pthread_mutexattr_t *MUTEXATTR);
```
  initializes the mutex pointed by **`MUTEX`**.
- The **`MUTEXATTR`** parameters allows the caller to specify which "flavor" of mutex it wants, that is, its attributes. Several attributes, related with the priority inversion problem, will be described later.
- The function:
```
int pthread_mutex_destroy(
         pthread_mutex_t *MUTEX);
```
  destroys a mutex.

Mutex Lock/Unlock

The functions:

```
int pthread_mutex_lock(pthread_mutex_t *MUTEX);
  int pthread_mutex_unlock(pthread_mutex_t *MUTEX);
```

allow a thread to acquire and release the mutex **`MUTEX`**, respectively.

- `pthread_mutex_lock` blocks the calling thread if the mutex is currently owned by another thread.
- `pthread_mutex_unlock` restarts one of the threads waiting for the mutex, if any...
- ...otherwise, it unlocks the mutex.

Polling and Timed Wait

The following functions allow the caller to exercise a finer control on the wait:

- ```
  int pthread_mutex_trylock(
    pthread_mutex_t *MUTEX);
  ```

  works like `pthread_mutex_lock`, but never blocks the caller and immediately returns with an error code if the mutex is owned by another thread.
- ```
  int pthread_mutex_timedlock(
    pthread_mutex_t *MUTEX,
    const struct timespec *ABSTIME);
  ```

  has an additional parameter **`ABSTIME`** that poses an upper bound on the wait time.

Condition variables

Definition

A **condition variable** is a synchronization device useful to perform a passive wait until a predicate on a set of shared variable is satisfied.

- It is always associated with a **mutex**, to avoid a race condition between the evaluation of the predicate and the execution of the wait, when another thread signals the condition in between.
- Unlike with monitors (which are a **programming language** construct), the programmer is responsible of ensuring a correct association in this case.

Initialization and Destruction

- The function:

```
int pthread_cond_init(
  pthread_cond_t *COND,
  pthread_condattr_t *CONDATTR);
```

  initializes the condition variable pointed by **COND**. The CONDATTR argument allows the caller to specify which attributes the condition variable shall have but, for the sake of simplicity, we will not describe them.
- The function:

```
int pthread_cond_destroy(
  pthread_cond_t *COND);
```

  destroys the condition variable pointed by **COND**, provided that no threads are waiting on it. Otherwise, it fails and returns an error code to the caller without destroying the condition variable.

Wait on a Condition Variable

The function:

```
int pthread_cond_wait(
  pthread_cond_t *COND,
  pthread_mutex_t *MUTEX);
```

atomically releases the mutex **MUTEX** and waits for the condition variable **COND** to be signaled.

To avoid race conditions, this function **re-acquires** **MUTEX** before returning to the caller after the wait.

Signalling a Condition Variable

The functions:

```
int pthread_cond_signal(
    pthread_cond_t *COND);
int pthread_cond_broadcast(
    pthread_cond_t *COND);
```

restart **one** or **all** threads that are waiting on the condition variable `COND`, respectively. They have no effect if no threads are waiting on `COND`.

- Both functions assume that the calling thread owns `MUTEX`, to avoid race conditions.
- If more than one thread is waiting on `COND`, `pthread_cond_signal` restarts one of them, but it is not specified **which one**.

Timed Wait and Cancellation

The function:

```
int pthread_cond_timedwait(
  pthread_cond_t *COND,
  pthread_mutex_t *MUTEX,
  const struct timespec *ABSTIME);
```

works like `pthread_cond_wait` but bounds the maximum duration of the wait by means of the **`ABSTIME`** argument.

Both `pthread_cond_wait` and `pthread_cond_timedwait` are cancellation points: if a thread accepts a cancellation request while waiting on a condition variable, it resumes execution (to perform its cleanup and finalization functions) after **re-acquiring `MUTEX`**. Keeping this peculiarity in mind is important to write good cleanup handlers.

Thread-Specific Data (TSD)

- All threads belonging to the same process share the same (virtual) address space.
- Hence, only **automatic** variables (usually allocated on the thread stack) are private to each thread.
- All other variables, either **global** or **static**, are shared among all threads.

Thread Specific Data (TSD)

It is often convenient to have variables that are globally accessible (like **global** and **static** variables), but have **private** contents nevertheless.

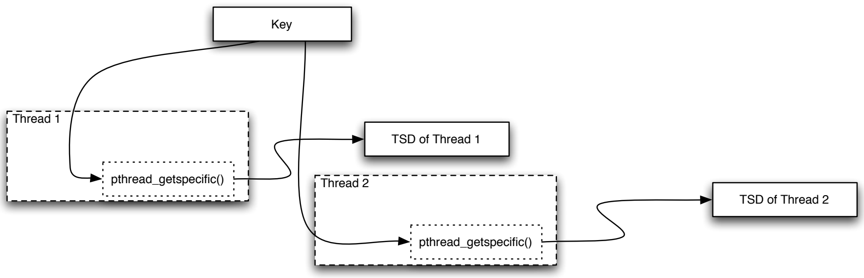
TSD Management and Access

- The following functions can be used to manage and access TSD:

Nome	Scopo
`pthread_key_create`	Create a new TSD key
`pthread_key_delete`	Destroy a TSD key
`pthread_setspecific`	Set the per-thread TSD value
`pthread_getspecific`	Get the per-thread TSD value

- Each process can create up to `PTHREAD_KEYS_MAX` **keys** to represent different sets of Thread-Specific Data. Each key has a `void *` pointer value that is **distinct** for each thread and that each thread can get and set autonomously.

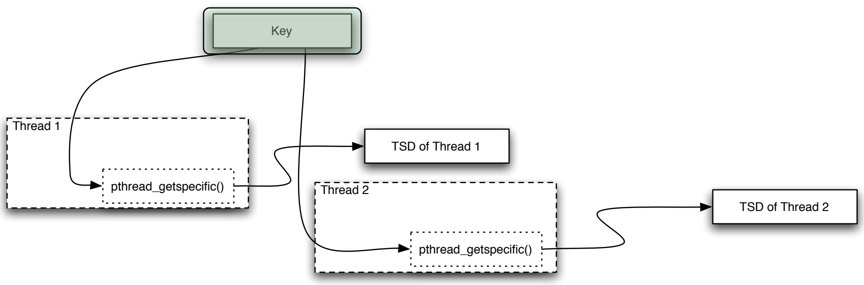
TSD Usage



The picture shows how TSD are typically used:

- First of all, the **shared** TSD key must be created by means of the `pthread_key_create` function.
- Then, each thread can set and get its own **private** pointer associated with the key by means of the `pthread_setspecific` and `pthread_getspecific` functions.
- In the picture, two threads access their TSD by invoking `pthread_getspecific` on the shared key.

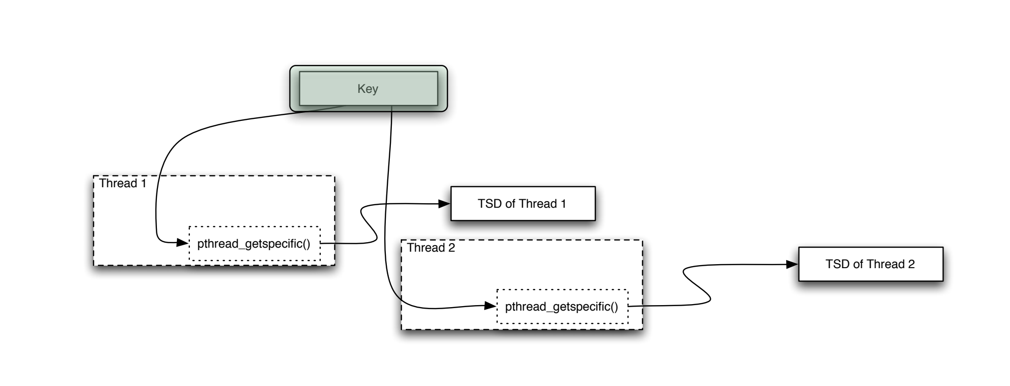
TSD Usage



The picture shows how TSD are typically used:

- First of all, the **shared** TSD key must be created by means of the `pthread_key_create` function.
- Then, each thread can set and get its own **private** pointer associated with the key by means of the `pthread_setspecific` and `pthread_getspecific` functions.
- In the picture, two threads access their TSD by invoking `pthread_getspecific` on the shared key.

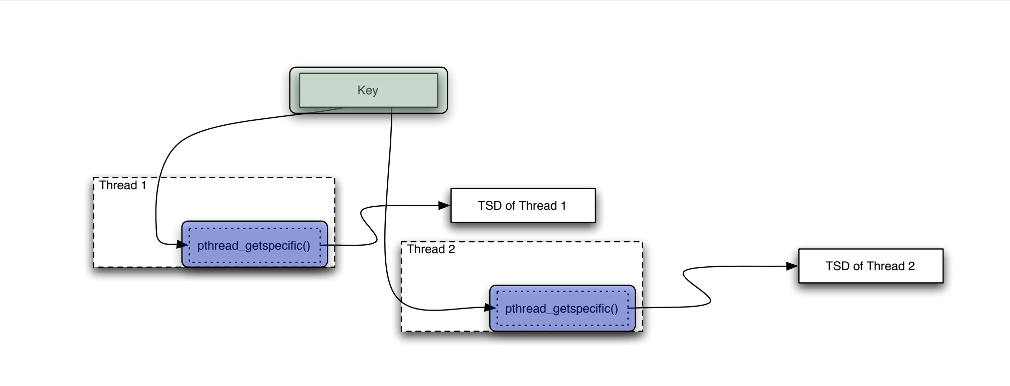
TSD Usage



The picture shows how TSD are typically used:

- First of all, the **shared** TSD key must be created by means of the `pthread_key_create` function.
- Then, each thread can set and get its own **private** pointer associated with the key by means of the `pthread_setspecific` and `pthread_getspecific` functions.
- In the picture, two threads access their TSD by invoking `pthread_getspecific` on the shared key.

TSD Usage



The picture shows how TSD are typically used:

- First of all, the **shared** TSD key must be created by means of the `pthread_key_create` function.
- Then, each thread can set and get its own **private** pointer associated with the key by means of the `pthread_setspecific` and `pthread_getspecific` functions.
- In the picture, two threads access their TSD by invoking `pthread_getspecific` on the shared key.

Creation of a New TSD Key

The following function creates a new TSD key:

```
int pthread_key_create(
  pthread_key_t *KEY,
  void (*FINALIZER) (void *));
```

- This function creates a new TSD key and stores it into the location pointed by **KEY**.
- The **FINALIZER** function will be invoked to destroy the TSD associated with the key by a thread when that thread terminates, either voluntarily or after a cancellation.
- When invoked, the FINALIZER gets a pointer to the TSD to be destroyed as argument.
- The FINALIZER is **not** called when a thread explicitly executes a pthread_setspecific or when the key itself is destroyed by means of pthread_key_delete.

Deletion of a TSD Key

The following function deletes a TSD key:

```
int pthread_key_delete(pthread_key_t KEY);
```

Its effect is to destroy the given **KEY**, **without** calling its finalizer.

Both pthread_key_create and pthread_key_delete return a non-zero integer on error.

TSD Access

- Each thread sets its own pointer, associated with a TSD key `KEY`, with the function:

  ```
  int pthread_setspecific(
    pthread_key_t KEY,
    const void *POINTER);
  ```

  where **`POINTER`** is the new TSD pointer to be associated with **`KEY`** for the calling thread. It returns a non-zero value on error, and does **not** invoke the finalizer for the old TSD pointer being replaced.

- Each thread can get its own pointer associated with a TSD key `KEY`, with the function:

  ```
  void *pthread_getspecific(pthread_key_t KEY);
  ```

  This function returns either the TSD pointer, or a `NULL` pointer on error.

It is impossible for a thread to get the TSD pointer belonging to another thread.

Initialization Code (I)

In a multithreaded process, there is often the need of executing a fragment of code, typically dealing with initialization activities, **exactly once**. The following function comes to help:

```
int pthread_once(
  pthread_once_t *ONCE,
  void (*INIT) (void));
```

Initialization Code (II)

- The shared variable pointed by **`ONCE`** shall be statically initialized to the constant `PTHREAD_ONCE_INIT`, defined in `pthread.h`.
- The very first invocation of `pthread_once(`**`ONCE, INIT`**`)` atomically executes the **`INIT`** function and updates the variable pointed by **`ONCE`**.
- Any further invocation of `pthread_once(ONCE, INIT)` with the same `ONCE` has no effect.
- It can be used, for example, to create a TSD key when a thread attempts to use the TSD pointer for that key for the first time.

What is Exception Handling?

Definition

The terms **exception** and **exception handling** express an approach to programming which attempts to detect, handle and contain error situations.

- As a consequence, most programming languages provide facilities which enable at least some exceptions to be handled.
- The degree of sophistication of the exception handling models and mechanisms increased considerably in recent years.
- Unfortunately, the C programming language, even when used within the framework of the POSIX standard, is well behind the state of the art in this respect.

Error Detection

Error detection techniques can be partitioned into two main classes:

1. **Environmental detection**: these errors are detected by the environment in which a program is executed. They include, for example:
   - Hardware-detected errors, for example the execution of an illegal instruction, a protection violation or a power failure.
   - Errors detected by the language run-time support system or by the operating system, for example an invalid argument passed to a system call.
2. **Application detection**: these errors are detected by the application itself as a result of self-checking. For example:
   - Replication checks (perform a computation N times, using different techniques, and compare the results).
   - Assertion checks (evaluate a logical expression which is true if no error is detected).
   - Timing checks (verify that the timing of a critical process function stays within its deadline).

Synchronous and Asynchronous Reporting

When an error is detected, it is reported to the offending process by means of an **exception**.

Depending on the nature of the error, and on the delay in detecting it, the exception can be raised either synchronously or asynchronously:

- A **synchronous** exception is raised as the immediate result of a fragment of code attempting to perform an illegal operation, for example a system call invoked with an invalid parameter.
- An **asynchronous** exception has no direct relation with the operation being performed by the process when it is raised, for example an exception raised as a result of a power failure.

Exception Classes

Overall, there are therefore four classes of exceptions:

1. Synchronous exceptions, detected by the environment: for example, a divide by zero.
2. Synchronous exceptions, detected by the application: for example, an assertion failure.
3. Asynchronous exceptions, detected by the environment: for example, a power failure.
4. Asynchronous exceptions, detected by the application: for example, a violation of a timing constraint.

Reserved Return Values

Returning an **unusual**, or **reserved** value from a function to denote an exception is one of the most primitive exception handling mechanisms.

- In POSIX, error conditions are indicated by the return of either a **`NULL` pointer**, or a **non-zero integer value**, depending on the function.

+ It is simple, and does not require any new language mechanism to be implemented.

– The exception handling code is obtrusive, because it is intertwined with the regular application code.

– It is not clear how to handle errors detected by the environment and not explicitly checked for by the process. For example, what about protection violations?

Exceptions and Signals

POSIX uses a **signal**, to be discussed soon, to inform a process of an exception which it is not explicitly checking for.

- All these exceptions are detected by the environment, but they may be detected either synchronously or asynchronously. For example:
  - A divide by zero is (usually) detected synchronously.
  - A power failure is detected asynchronously.
- POSIX uses two distinct mechanisms for exception handling, and they have no direct relation to exception classes.

Exception Handling in C

Unfortunately, C does not define any exception-handling mechanism within the language. However, two POSIX facilities, `setjmp` and `longjmp`, combined with the macro facility of the language, can be used to provide some form of non-trivial exception handling.

- The `setjmp` routine saves the calling thread status and returns zero.
- `longjmp` restores the status previously saved by `setjmp`, so that the thread abandons its current execution flow and restarts from the position where `setjmp` was called.
- The second time, however, `setjmp` returns the value of an argument passed to `longjmp` instead of zero.

Setjmp

```
int setjmp(jmp_buf env);
```

- `setjmp` saves the calling thread status into **`env`** and returns zero.
- The size and format of the data structure that holds the thread status depend on the execution environment, but POSIX defines an opaque data type, **`jmp_buf`** for it, so that the application is still portable.
- After a `longjmp` is performed on the same `env`, `setjmp` “appears” to return to the caller a second time, with a non-zero return value.

Longjmp

```
void longjmp(jmp_buf env, int val);
```

- This function restores the thread status saved into **`env`** by the most recent invocation of `setjmp`.
- Hence, the thread proceeds “as if” that `setjmp` had just returned the value **`val`** instead of zero.
- It is not possible to call `longjmp` when the saved context is no longer valid, for example after the routine which invoked `setjmp` has returned to the caller.
- It is not possible to restore a context saved by a thread from a different thread.
- These error conditions might not always be detectable.

What about Accessible Objects?

After a `longjmp`...

- Since the thread status does **not** include the address space, all accessible objects (e.g. global variables) have values, and all other components of the abstract machine (e.g. open files) have state, as of the time `longjmp` (**not** `setjmp`) was called.
- POSIX also specifies that some processor state item, for example the floating-point status flags, may not be properly restored.
- Mainly due to compiler optimizations, the values of **automatic** objects that are local to the function containing the `setjmp` invocation and have been changed between the `setjmp` and the `longjmp` calls may have an **undefined** value, unless they are qualified as **`volatile`**.

The Termination Model for Exception Handling

In this model, also known as the **escape** model, when an exception has been raised and handled, control does **not** return to the point where the exception occurred. Instead, control is passed to the **calling** block or procedure.

- Other models exist, for example the **resumption** or **notify** model in which execution resumes from the point where the exception occurred after the exception has been handled. POSIX supports the resumption model for exceptions it conveys through a signal.
- The **hybrid** model leaves the exception handler free to decide whether to resume the operation in which the exception was raised, or terminate it.

Implementing the Termination Model

```
#include <setjmp.h>
#define EXC_FIRST 1
#define EXC_...   ...

int f(void) {
    jmp_buf save_area;
    int exc;

    if((exc = setjmp(save_area)) == 0) {
      ... The guarded block ...
      if(... Exception EXC_FIRST has been identified ...)
        longjmp(save_area, EXC_FIRST);
      ... End of the guarded block ...
    }

    else {
      ... Exception handling code ...
      switch(exc) {
        case EXC_FIRST:
          ... Handle EXC_FIRST ...
          break;
        case ...
      }
    }
}
```

Each exception has its own, unique identifier: for example, `EXC_FIRST` represents an exception. Identifiers cannot be zero.

Implementing the Termination Model

```
#include <setjmp.h>
#define EXC_FIRST 1
#define EXC_...   ...

int f(void) {
    jmp_buf save_area;
    int exc;

    if((exc = setjmp(save_area)) == 0) {
      ... The guarded block ...
      if(... Exception EXC_FIRST has been identified ...)
        longjmp(save_area, EXC_FIRST);
      ... End of the guarded block ...
    }

    else {
      ... Exception handling code ...
      switch(exc) {
        case EXC_FIRST:
          ... Handle EXC_FIRST ...
          break;
        case ...
      }
    }
}
```

A status save area is defined inside the block (function `f` in this case) containing the code to be guarded.

Implementing the Termination Model

```
#include <setjmp.h>
#define EXC_FIRST 1
#define EXC_...   ...

int f(void) {
    jmp_buf save_area;
    int exc;

    if((exc = setjmp(save_area)) == 0) {
      ... The guarded block ...
      if(... Exception EXC_FIRST has been identified ...)
        longjmp(save_area, EXC_FIRST);
      ... End of the guarded block ...
    }

    else {
      ... Exception handling code ...
      switch(exc) {
        case EXC_FIRST:
          ... Handle EXC_FIRST ...
          break;
        case ...
      }
    }
}
```

The status is saved on entry into the guarded block. The first time `setjmp` returns zero, thus leading to the execution of the guarded block.

Implementing the Termination Model

```
#include <setjmp.h>
#define EXC_FIRST 1
#define EXC_...   ...

int f(void) {
    jmp_buf save_area;
    int exc;

    if((exc = setjmp(save_area)) == 0) {
      ... The guarded block ...
      if(... Exception EXC_FIRST has been identified ...)
        longjmp(save_area, EXC_FIRST);
      ... End of the guarded block ...
    }

    else {
      ... Exception handling code ...
      switch(exc) {
        case EXC_FIRST:
          ... Handle EXC_FIRST ...
          break;
        case ...
      }
    }
}
```


In the guarded block, `longjmp` is called whenever an exception is identified. The exception identifier is passed to `longjmp` as an argument, in order to propagate it to the exception handling code.

Implementing the Termination Model

```
#include <setjmp.h>
#define EXC_FIRST 1
#define EXC_...   ...

int f(void) {
    jmp_buf save_area;
    int exc;

    if((exc = setjmp(save_area)) == 0) {
      ... The guarded block ...
      if(... Exception EXC_FIRST has been identified ...)
        longjmp(save_area, EXC_FIRST);
      ... End of the guarded block ...
    }

    else {
      ... Exception handling code ...
      switch(exc) {
        case EXC_FIRST:
          ... Handle EXC_FIRST ...
          break;
        case ...
      }
    }
}
```

The call to `longjmp` leads to the execution of the exception handling code. It can distinguish one exception from another by means of the exception identifier.

Implementing the Termination Model

```
#include <setjmp.h>
#define EXC_FIRST 1
#define EXC_...   ...

int f(void) {
    jmp_buf save_area;
    int exc;

    if((exc = setjmp(save_area)) == 0) {
      ... The guarded block ...
      if(... Exception EXC_FIRST has been identified ...)
        longjmp(save_area, EXC_FIRST);
      ... End of the guarded block ...
    }

    else {
      ... Exception handling code ...
      switch(exc) {
        case EXC_FIRST:
          ... Handle EXC_FIRST ...
          break;
        case ...
      }
    }
}
```


According to the termination model, after exception handling control is returned to the caller of `f`.

Comments to the Code

– The method cannot always be used to handle the exceptions that are conveyed through a signal, because on some systems `longjmp` cannot be used while handing a signal.
– The code is still difficult to understand, due to the various calls to `setjmp` and `longjmp`. Moreover, it relies on programmer's discipline to work correctly, because the compiler is unable to perform any check on it.
+ It is possible to define a set of `cpp` macros to help structure the program.
+ The exception handling code is less obtrusive, because it is not longer intertwined with the application code dealing with normal processing.
+ It resembles the approach taken by more sophisticated languages, for example Ada.

Asynchronous Event Notification

The POSIX standard specifies three different ways to notify a process about the occurrence of an asynchronous event (depending on the value of the `.sigev_notify` field of the `struct sigevent`):

1. No notification: the process explicitly waits for events to occur by invoking a function, for example `mq_receive` on a message queue.
2. Execution of a notification **function**, specified by the `.sigev_notify_function` field. The function is executed by its own thread, whose attributes are taken from `.sigev_notify_attributes`.
3. Generation of an asynchronous, real-time **signal** as specified by the `.sigev_signo` field, tagged with the value of `.sigev_value`.

The execution of a notification function requires multithreading support.

Signals

Signals were already specified by the **ISO C** standard as a way for the operating system to convey information to a process or thread even if it is not necessarily waiting for that information. The **IEEE 1003.1** standard extended the mechanism for real-time. The system raises a signal:

- when an error occurs during process execution, for example a memory reference through an invalid pointer (synchronous signal);
- when a hardware or operating system error occurs, for example a power outage (asynchronous signal);
- when the signal is explicitly raised by another process;
- ...
- to notify the process about an asynchronous event.

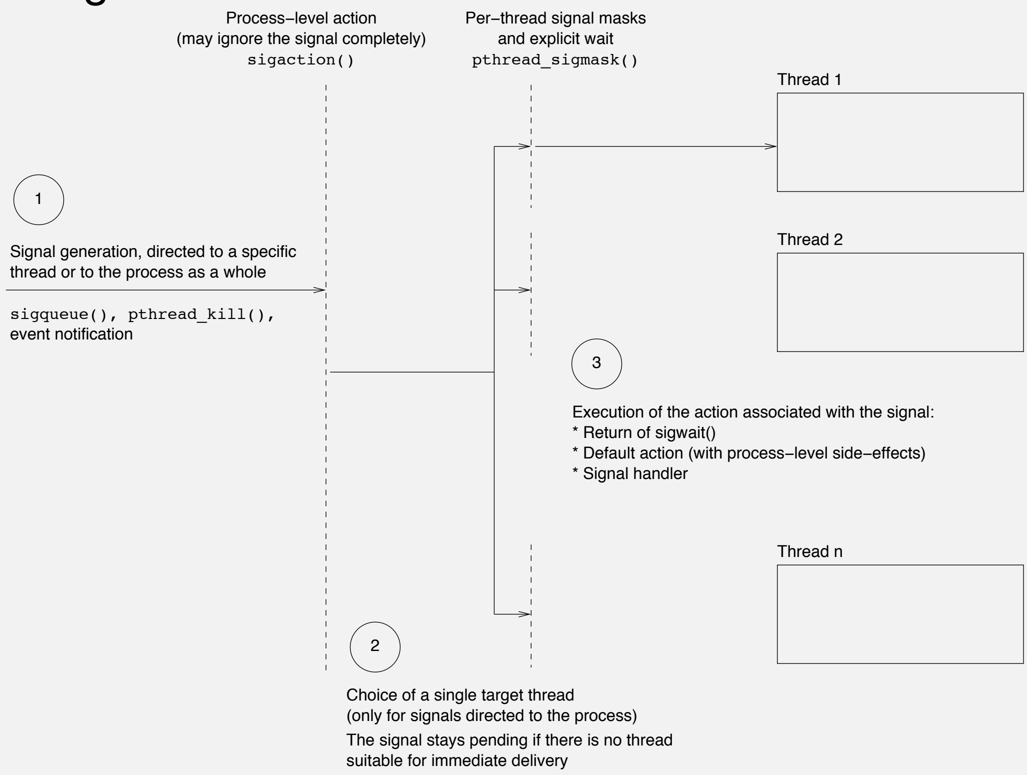
ISO Signal Versus POSIX Signals

- In ISO C each kind of signal has a unique integer value (its *signal number*), but when multiple signals of different kind are pending their service order is unspecified. Instead, POSIX establishes a priority hierarchy for a subset of signal numbers, from `SIGRTMIN` to `SIGRTMAX`.
- In ISO C there is no provision for signal queuing. Hence, if multiple signals of the same kind are raised before the target process had a chance of handling them, all signals but the first one are lost. POSIX associates a FIFO queue to each kind of signal and provides the ability to tag each signal request with an information item (`union sigval`).
- Signal handling and multithreading (if implemented) are fully integrated.

Signal-Related Functions

Funzioni	Scopo
sigqueue pthread_kill	Raise a signal
sigaction	Get/Set the process-level action
sigaltstack	Set the signal handler private stack
pthread_sigmask	Get/Set the per-thread signal mask
sigemptyset sigfillset sigaddset sigdelset sigismember	Manipulate signal masks
sigwait sigwaitinfo sigtimedwait	Wait for a signal to arrive

The Big Picture



Raising a Signal

- Most signals are raised by the operating system itself, and not by any process or thread.
- A signal may target either a **process** as a whole or, more specifically, a single **thread**;
- POSIX specifies that the signals raised by the system must be targeted as precisely as possible (depending on the kind of signal), for example:
  - memory fault → *thread*;
  - power outage → *process*;
- It is possible to synthesize a signal with the functions:
  - `sigqueue`, for signals that target a process;
  - `pthread_kill`, for signals that target a thread (belonging to the same process as the caller).

Process-Level Action

Each **process** can specify how it wants to react to each kind of signal defined in the system by means of the `sigaction` function. The reaction may consist of:

- Ignoring the signal completely.
- Carrying out a default action associated with the signal. That action is executed by the operating system on behalf of the process and may have process-wide side effects, for example process termination.
- Executing a signal handling function specified by the programmer (**signal handler** for short).

When a signal is raised, the system immediately checks the process-level action associated with the signal. If the process is ignoring the signal, it is immediately discarded, otherwise the system looks for a thread to handle it.

Signal Flags

The `sigaction` function allows the caller to set a (huge) set of **flags** for each kind of signal. For example:

`SA_SIGINFO` allows each signal request to be tagged with a limited amount of information (a `union sigval`). That information is then passed on to the signal handler through an additional argument.

`SA_RESTART` enables the automatic restart of any system call that has been interrupted by the arrival of a signal. When it is not set, the interrupted system call fails with an error indication.

`SA_ONSTACK` specifies that the signal handler must be executed on its own, private stack, set up by the `sigaltstack` function, instead of using the standard stack.

Signal Delivery and Acceptance

- A signal may be either ***delivered*** *to* or ***accepted*** *by* a thread.
- Each **thread** has its own signal mask that allows it to **block** one or more kinds of signal. It can be read and written by means of the `pthread_sigmask` function.
- The following functions manipulate a signal mask:
  - `sigemptyset`: set a mask so that it *excludes* all kinds of signal;
  - `sigfillset`: set a mask so that it *includes* all kinds of signal;
  - `sigaddset`, `sigdelset`: add/remove a kind of signal to/from a mask;
  - `sigismember`: check whether a mask includes a kind of signal or not.

Delivery

- A thread is a candidate for signal *delivery* if and only if its *signal mask* is not blocking that kind of signal.
- When a signal is successfully delivered to a **thread**, that thread executes the **process**-level action associated with the signal.
- That action may entail process-wide side effects. For example, it may terminate the process.

Acceptance

- A thread can also **explicitly** wait for signals to arrive by means of the `sigwait` function. The function has a signal mask as argument, to specify which kinds of signal the thread is interested in.
- When a signal of the right kind arrives, the thread **accepts** the signal and continues its execution after the `sigwait` call, which also returns the signal number of the signal just accepted.
- The priority hierarchy set forth for the signal numbers between `SIGRTMIN` and `SIGRTMAX` is valid for signal acceptance, too.
- When a thread wants to accept a kind of signal in this way, it must also block that kind of signal, otherwise delivery takes precedence.
- `sigwaitinfo` and `timedwait` are extended versions of `sigwait`. They allow the caller to gather the information item that optionally tags each signal request, and to place an upper bound on the waiting time.

Selection of the Victim Thread

- If a signal targets a specific **thread**, then only that thread is a candidate for delivery or acceptance.
- Is a signal targets a **process** as a whole, then all its threads are candidates. Among them, the system chooses *exactly* one thread whose signal mask does not block the signal (for delivery), or is currently executing a `sigwait` with a mask that includes the signal (for acceptance).
- If there is not any suitable candidate when the signal is raised, the signal is kept *pending*, until:
  - either its delivery or acceptance become possible, or
  - the process-level action is set to ignore the signal.

Other Functions

Nome	Scopo
`pthread_equal`	Check whether two thread IDs are equal
`pthread_self`	Return the caller thread ID
`pthread_detach`	*Detach* a thread after creation

- Each POSIX implementation may provide (and usually does provide, especially for real-time systems) additional functions besides those mandated by the standard.
- These functions shall have the suffix **`_np`** (not portable), to highlight that their use impairs the portability of the application to other execution environments.

POSIX and the Real World

IEEE Std 1003.1 is currently supported by most real-time operating systems, both commercial and experimental. For example:

- LynxOS
- pSOSystem
- QNX
- RTMX O/S
- VxWorks
- ChorosOS
- eCos
- RT-Linux/RTAI
- RTEMS

Other Standards: OSEK/VDK

- Devoted to automotive applications.
- *Static* configuration of processes and synchronization devices.
- Specialized interface, less general but more specific and powerful than POSIX's for the automotive application domain.
- Originally developed by several German automotive industries, later joined by many other partners.
- The specification is freely available (at `http://www.osek-vdx.org/`), implementations may be (and most are) proprietary.