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

Ivan Cibrario Bertolotti

IEIIT-CNR / Politecnico di Torino

Academic Year 2006-2007

Outline

The POSIX Standard

- The first version of the POSIX (**P**ortable **O**perating **S**ystem **I**nterface for Computing Environments) standard was published between 1988 and 1990. It specifies a standard way for applications to interact with the operating system.
- Nowadays, POSIX is a set of over 30 standardisation documents, which span from the definition of the basic services an operating system shall provide, up to the specification of analysis methods to check whether a given operating system complies with the standard.
- Among these documents, of special interest for us is the **System Interfaces** (XSH) volume of **IEEE Std 1003.1-2004**, which specifies a C language interface between applications and the operating system, and includes **real-time** extensions.
- The standard defines the semantics of the system services, the way of using them by means of the C language, including notes on portability and error detection and recovery.

POSIX and Real-Time

Real-Time Extensions to the Standard

Since 1993, the POSIX standard was extended to make it more suitable and useful for real-time applications. In particular, the following extension were adopted:

Standard	Description
1003.1b	Basic real-time extensions; first published in 1993
1003.1c	Threads extensions; published in 1995
1003.1d	Additional real-time extensions; published in 1999
1003.1j	Advanced real-time extensions; published in 2000

These extensions, originally published as amendments to the standard, have then been incorporated into the standard itself with IEEE Std 1003.1-2001 and -2004. We will focus on the contents of 1003.1b and 1003.1c, because these extensions are older and thus more widely implemented.

Where is the Standard?

The Open Group

The main volumes of IEEE Std 1003.1-2004, also known as the "Single UNIX Specification, Version 3", are available for download (free registration required), at the following URL:

`http://www.unix.org/version3/online.html`

Dynamic vs. Static Process Creation

- In some real-time applications the set of processes needed to carry out the job is known in advance, and remains the same for the whole life of the system.
- Accordingly, several real-time operating systems support the **static** configuration of the set of processes to be executed, and can take advantage from the additional information they have available about them.
- Instead, POSIX provides a set of functions to **dynamically** create new processes at will. In particular, `fork` duplicates the calling process and `exec` replaces the current process image with a new one.
- In addition, the `exit` function terminates the calling process, `wait` blocks the calling process until one of its child processes terminates, and `kill` sends a signal to a process.

Process Hierarchy

In POSIX, processes are organized as a hierarchical **tree**.

- When the operating system is booted, it crafts the ancestor of all other processes, traditionally named `init`.
- When a process creates another process, by invoking `fork`, it becomes its **parent**.
- Each process has exactly **one** parent, and **zero or more children**.
- Each process has its own **process identifier**, that is guaranteed to **uniquely identify** it during all its lifetime.
- The process identifier is used whenever it is necessary to make a reference to the process, for example when sending it a signal.

Semantics of Fork

```
pid_t fork(void);
```

- `fork` creates a new process that is **almost** an **exact copy** of the calling process. After a successful `fork`, both processes execute concurrently the statements that follow the `fork` call.
- The two processes mainly differ for the following:
  - The child process has its own, unique process identifier.
  - The return value of `fork` is the reserved value **0** in the child process, and the **process identifier** of the child in the parent.
  - The child process has its own copy of the parent's descriptors, but they reference the same underlying objects.
- The reserved value **-1** is returned by `fork` to inform the caller that an error occurred. In this case, as for many other functions, the `errno` variable gives more information about the error.

An Example of Fork

Process X (parent)

```
1.  ... A ...
2.  r = fork();
3a. ... B ...
```

Process Y (child)

```
    ... A ...
    r = fork();
3b. ... B ...
```

- The parent executes the statement `A`, and then `fork`.
- `fork` creates the child process.
- Both processes concurrently execute the statement `B`.
- The parent and the child process have different values of `r` after the `fork`:
  - `r` is 0 in the child process.
  - `r` is the child process identifier (always $\neq 0$) in the parent process.
- For example, in `B` we may evaluate `if (r) ...` to distinguish between the parent and the child.

An Example of Exec

Process X

```
1. ... A ...
2. x = execve("newp",...); » » » main() (of "newp")
   ... B ...
```

- "`exec`" is the generic name of a group of functions; `execve` is one of them.
- Process X executes the statement `A`, then `execve()`.
- The statement `B`, that follows the `execve()` call, is **never** executed, unless `execve()` fails.
- Instead, the execution continues at the entry point of the new executable image, that has been loaded from file `newp`.
- The process identifier does not change.

Process Termination

```
void exit(int status);
```

- The `exit` function **explicitly** terminates the calling process when invoked.
- The function makes the low-order 8 bits of the **`status`** argument available to the parent process.
- The same result can also be obtained **implicitly**, by using `return` from `main()`. In this case, the behavior is the same as calling `exit` with the returned value. Moreover, reaching the end of `main()` is the same as performing `exit(0)`.
- In any case, a number of cleanup activities are carried out before terminating, for example:
  - Call the cleanup functions registered with `atexit`, in LIFO order.
  - Flush and close all open streams.
  - Unlink all temporary files.

Waiting for Process Termination

```
pid_t wait(int *status);
```

- This function returns to the calling process the `status` information about one of its terminated children, waiting for child termination if necessary.
- Upon successful completion, `wait` return value is the process identifier of the child for which it is giving the status information.
- In this case, `status` contains the exit code of the child in its low-order 8 bits, and other status information in the others.
- `wait` returns to the caller prematurely, with the reserved return value **-1**, if an error occurs.
- An extended function, `waitpid`, allows the caller to be more specific about which children it is interested in, and to set additional options.

Order of Termination and Wait

The standard does not make any assumption on the relative **order** between the child termination and the parent waiting for it.

- A child can terminate **before** its parent waits for it. In this case, barring some special circumstances, it is transformed into a *zombie* process, that is:
  - its process identifier remains allocated, and
  - its status information remains available for later retrieval.
- A parent can also terminate when some of its children are still alive, or have become zombies. In this case, all these processes are inherited by an implementation-defined system process (that is often `init` itself), which is also responsible of reaping them as appropriate.

Killing a Process

Voluntary and involuntary termination

Besides **voluntary** termination, a POSIX process can also be terminated **involuntarily**, as a consequence of catching a **signal** raised against it by another process.

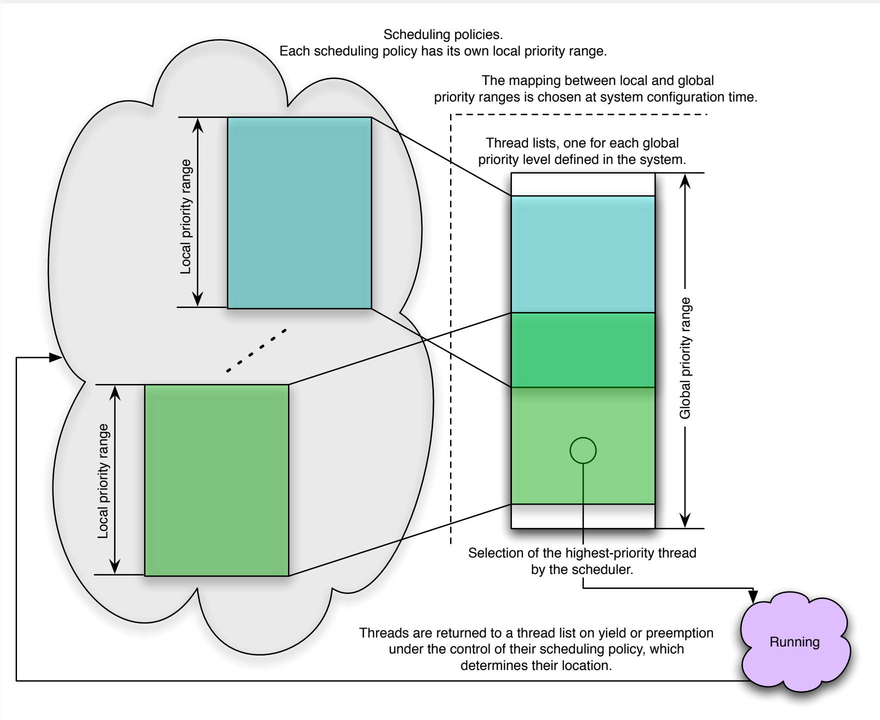
- The involuntary termination of a process is prone to several nasty side effects, especially when the process is involved in concurrent programming activities.
- As a consequence, except for several special signals, each process has some freedom to decide **whether** to catch signals or not, and **how** to react to them.
- Signal handling behavior is quite complex and related to multithreading, hence it will be discussed in more detail later.

Syntax and Semantics of Kill

```
int kill(pid_t pid, int sig);
```

- The `kill` function sends a signal to the process whose process identifier is **`pid`**.
- The **`sig`** specifies which signal shall be sent, among the signal types supported by the system.
- Like many other POSIX functions, `kill` returns to the caller an **integer value** that is zero if it was executed succesfully, and is -1 otherwise.
- Several special values of `pid` are used to indicate that the signal shall be sent to a **group** of processes.
- The right for a process to send a signal to another process is subject to some security-related restrictions, related to their owning users and to their relative position in the process hierarchy.

Scheduling Model (I)



Scheduling Model (II)

- The abstract POSIX scheduling model foresees an ordered thread list for each global priority level defined in the system. It contains all **ready** threads for that priority.
- To assign a CPU, the scheduler **extracts** the thread from the head of the highest-priority, non-empty thread list (like the fixed priority scheduling with priority classes does). That thread becomes **running** and is removed from the list.
- Each thread in the system is under the control of a **scheduling policy**, that decides how it is **inserted** into the thread lists, and how it is **moved** among them.
- Each scheduling policy works inside a local priority range assigned to it (and comprising at least 32 distinct priorities).

Scheduling Model (III)

- The mapping of each **local** priority range, one for each scheduling policy, into the **global** priority range is defined during system configuration, and cannot change thereafter.
- Partial or total overlaps between different local priority ranges when they are mapped into the global priority range are allowed.
- The standard specifies the following scheduling policies:

`SCHED_FIFO`:	First in, first out
`SCHED_RR`:	Round robin
`SCHED_SPORADIC`:	Sporadic server
`SCHED_OTHER`:	Other policy (usually time-sharing)

SCHED_FIFO

- When a running thread is preempted, that thread is placed at the **head** of the thread list it belongs to.
- When a thread transitions from blocked to ready, it is placed at the **tail** of the thread list it belongs to.
- When the priority of a running or ready thread is modified, it is placed at the tail of the thread list that corresponds to its new priority, except when `pthread_setschedprio` has been used.
- In the latter case, the placement depends on the sign of the priority change, namely:
  - If the priority has been increased, it is placed at the tail.
  - If the priority is unchanged, the thread does not change place.
  - If the priority has been decreased, it is placed at the head.
- When a thread voluntarily yields the CPU, it is placed at the tail of the thread list it belongs to.

SCHED_RR, SCHED_OTHER

- The `SCHED_RR` scheduling policy is quite similar to `SCHED_FIFO`, but:
  - When the system detects that a thread has been running for a full **quantum**, it is forced to return to the ready state, and placed at the tail of the thread list it belongs to, thus forcing a rescheduling.
  - The length of the quantum is a system configuration parameter and cannot be changed dynamically.
- The `SCHED_OTHER` scheduling policy is the default scheduling policy of the operating system, and **may be unsuitable for real-time**.
- Implementations are allowed to implement additional scheduling policies, but their use is inherently not portable.

Scheduler Control

The **process-level** functions for scheduler control are:

Nome	Scopo
`sched_get_priority_max`	Maximum priority of a policy
`sched_get_priority_min`	Minimum priority of a policy
`sched_getparam`	Get scheduling parameters
`sched_getscheduler`	Get scheduling policy
`sched_rr_get_interval`	Round robin quantum length
`sched_setparam`	Set scheduling parameters
`sched_setscheduler`	Set scheduling policy and its parameters
`sched_yield`	Voluntary yield

Include `sched.h` before using any of these functions.

Priority Range

These functions return the maximum and minimum priority value allowed for a given scheduling policy:

```
int sched_get_priority_max(
    int POLICY);

int sched_get_priority_min(
    int POLICY);
```

Both functions have **`POLICY`** as argument, to uniquely identify a scheduling policy, and return an integer priority value. The symbolic constants to be used for `POLICY` are defined in `sched.h`.

Scheduling Parameters (I)

These functions get and set the scheduling parameters of a process:

```
int sched_getparam(
    pid_t PID,
    struct sched_param *PARAM);

int sched_setparam(
    pid_t PID,
    const struct sched_param *PARAM);
```

- **`PID`** is the process identifier of the process of interest. If `PID` is zero, the functions act on the calling process.
- **`PARAM`** is a pointer to a data structure, (declared in `sched.h`) that will receive (get) or contains (set) the scheduling parameters.

Scheduling Parameters (II)

- Simple policies, like `SCHED_FIFO` and `SCHED_RR`, have only one scheduling parameter, namely `.sched_prio`. It represents the priority of the process.
- More complex policies, for example `SCHED_SPORADIC`, have more parameters.
- Both functions return zero on success, a non-zero value on error.

Scheduling Policy

These functions get and set both the scheduling policy and the scheduling parameters of a process:

```
int sched_getscheduler(
    pid_t PID);

int sched_setscheduler(
    pid_t PID,
    int POLICY,
    const struct sched_param *PARAM);
```

- `sched_getscheduler` returns an **integer** that represents the current scheduling policy of process **`PID`**.
- `sched_setscheduler` atomically sets both a new scheduling policy (**`POLICY`**), and a new set of scheduling parameters (**`PARAM`**) for process **`PID`**. It returns zero on success, a non-zero value on error.

Other Functions

- The function:
  ```
  int sched_rr_get_interval(
      pid_t PID,
      struct timespec *INTERVAL);
  ```
  stores into the data structure pointed by **`INTERVAL`** the round-robin quantum length for process **`PID`**. It returns zero on success, a non-zero value on error.
- The function:
  ```
  int sched_yield(
      void);
  ```
  allows a process to voluntarily relinquish the CPU, in favour of another ready process. Albeit the standard specifies that this function shall return a non-zero value on error, it does not specify any error condition.

Semaphores

The following functions of IEEE Std 1003.1-2004 are related with **semaphores**:

Nome	Scopo
`sem_init`	Initialize an unnamed semaphore
`sem_destroy`	Destroy an unnamed semaphore
`sem_open`	Create/open a named semaphore
`sem_close`	Close a named semaphore
`sem_unlink`	Remove a named semaphore
`sem_wait`	$P()$ on a semaphore
`sem_post`	$V()$ on a semaphore
`sem_trywait`	Non-blocking $P()$ (polling)
`sem_getvalue`	Get current semaphore value

The `SEM_VALUE_MAX` macro, defined in `semaphore.h`, gives the **maximum value** allowed for semaphores.

Initialization

The following function creates a fresh, unnamed semaphore:

```
int sem_init(
   sem_t *SEM,
   int PSHARED,
   unsigned int VALUE);
```

The function:

- Initializes the data structure, pointed by **`SEM`**, that will represent the semaphore.
- Sets the initial value of the semaphore to **`VALUE`**.
- The **`PSHARED`** argument, when not zero, indicates that the semaphore may be shared among multiple **processes** (the semaphore is **always** shared among all **threads** belonging to the same process).

Destruction

The following function destroys an unnamed semaphore:

```
int sem_destroy(
   sem_t * SEM);
```

- When destroying a semaphore, all process blocked on it are **unblocked** immediately.
- These processes get to know that the semaphore has been destroyed because, in this case, `sem_wait` returns a non-zero error code.

Both `sem_init` and `sem_destroy` return a non-zero value on error.

Semaphore Sharing

- `sem_init` creates unnamed semaphores, hence they can be used only by the processes that know their descriptor.
- The descriptor (when stored into a global variable) is implicitly shared among all threads belonging to the same process, because they share the same address space.
- By contrast, **copying** a descriptor does **non** produce a valid descriptor. Hence, the usual address space inheritance mechanism put in place by `fork()` does **not** produce valid descriptors.
- To share a descriptor, it is necessary to store it into a shared memory segment, and map it into the address spaces of all the processes interested in it.

Named semaphores can be shared in a simpler, albeit less efficient, way.

Named Semaphores – Open

A process can gain access to a named semaphore by invoking:

```
sem_t *sem_open(
   const char *NAME, int OFLAG, ...);
```

Where:

- **NAME** is the name of the semaphore.
- **OFLAG** contains a set of flags that change several aspects of the function behavior (like with `open()`).
- The **additional arguments** depend on the value of `OFLAG`.

OFLAG

`OFLAG` is the inclusive or of a set of **flags** that change several aspects of the behavior of `sem_open`. The most important ones are:

`O_CREAT` when set allows `sem_open` to create the semaphore if it does not exist yet.

`O_EXCL` when set together with `O_CREAT`, makes `sem_open` fail if the semaphore already exists.

The value of `OFLAG` determines which additional arguments must follow `OFLAG` itself in the argument list. For example, if `OFLAG` contains `O_CREAT` two additional arguments are required, to specify the **protection** attributes of the semaphore and its **initial value**. `sem_open` returns a null pointer on error.

Named Semaphores – Close/Unlink

- The function:
```
int sem_close(
   sem_t *SEM);
```
cuts the link between the calling process and the semaphore **SEM**, and returns a non-zero value on error. No more operations on `SEM` are allowed after a successful invocation of `sem_close`.
- The function:
```
int sem_unlink(
   const char *NAME);
```
asks the system to delete the semaphore **NAME** as soon as the number of processes linked to it drops to zero. It returns a non-zero value on error.

P() and Polling

A thread executes a *P*() on a semaphore (either blocking or non-blocking), by means of the functions:

```
int sem_wait(sem_t *SEM);
   int sem_trywait(sem_t *SEM);
```

- **`sem_wait`** is equivalent to the abstract synchronization primitive *P*() and is a **cancellation point**. It returns a non-zero value on error: in particular, the destruction of an unnamed semaphore while one or more processes are waiting on it is reported as an error condition.
- **`sem_trywait`** is the non-blocking variant of `sem_wait`. If it is unable to immediately conclude the *P*() (becase the semaphore value is zero at the moment), `sem_trywait` immediately returns to the caller a non-zero value instead of waiting.
- For both functions **SEM** points to a semaphore descriptor, obtained from either `sem_init` or `sem_open`.

Timed *P*()

The function:

```
int sem_timedwait(
    sem_t *SEM,
    const struct timespec *ABS_TIMEOUT);
```

is the timed variant of `sem_wait`, where **`ABS_TIMEOUT`** represents an absolute time reference. When it is unable to conclude the *P*() before that time, this function returns a non-zero error code.

- When the function fails, the value of `errno` gives more information about the exact nature of the error (it may actually be a timeout, or something else).
- Specifying an **absolute** time reference, instead of a **relative** reference, is useful to contain the uncertainty of time measurements in the system.

V() and Semaphore Value

- A thread executes a *V*() on semaphore `SEM` through the function:

  ```
  int sem_post(sem_t *SEM);
  ```

  `sem_post` never blocks the caller, and is not a **cancellation point**. It is forbidden to increment a semaphore beyond the maximum value `SEM_VALUE_MAX`.
- It is possible to get the current value of semaphore `SEM` and store it into the location pointed by `SVAL` with the function:

  ```
  int sem_getvalue(sem_t *SEM, int *SVAL);
  ```

Is sem_getvalue useful?

`sem_getvalue` is not as useful as it seems, because it is **not** executed **atomically** with respect to any other synchronization primitive.

For example, this is **not** a valid substitute for `sem_trywait`:

```
sem_t sem;
   int sval, wait_result;
   ...
   sem_getvalue(&sem, &sval);
   if(sval > 0)  wait_result = sem_wait(&sem);
   ...
```

Another process may change the semaphore value between the invocation of `sem_getvalue` and `sem_wait`.

Shared Memory

The main functions dealing with **shared memory** are:

Nome	Scopo
`shm_open`	Open/create a shared memory segment
`close`	Close a shared memory segment
`shm_unlink`	Remove a shared memory segment
`mmap`	Map a shared memory segment into the caller's address space
`munmap`	Remove a mapping made by `mmap`

Include `sys/mman.h` before using any of these functions.

Open/Close/Unlink (I)

The functions:

```
int shm_open(const char *NAME, int OFLAG,
    mode_t MODE);
int close(int FD);
int shm_unlink(const char *NAME);
```

do the same as their counterparts for named semaphores, but work on shared memory segments. Each shared memory segment is represented by a **file descriptor**, that is, an integer value.

Open/Close/Unlink (II)

- Opening a shared memory segment does **not** automatically perform any mapping of the segment into the address space of the calling process. Hence, `sem_open` does not make the shared memory segment addressable in any way.
- The standard does not specify whether the contents of shared memory segments are preserved across system bootstraps.
- `shm_open` returns either a **file descriptor** (never negative), or a negative value (on error). Both `close` and `shm_unlink` return a non-zero value on error.

Mapping – mmap()

The function:

```
void *mmap(
    void *ADDR, size_t LEN,
    int PROT, int FLAGS,
    int FILDES, off_t OFF);
```

maps a portion of the shared memory segment **FILDES** starting from the given offset **OFF** into the caller's address space.

- **LEN** represents the size of the mapping, in bytes.
- **PROT** denotes how the mapped region can be accessed. It is the bitwise inclusive or of: `PROT_READ`, `PROT_WRITE`, and `PROT_EXEC`.
- **ADDR** allows the caller to "suggest" to the system where in the address space the mapped region should be placed.
- **FLAGS** indicates how the mapped region will be manipulated (see the next slide).

mmap() Flags

The `FLAGS` argument of `mmap()` is the bitwise inclusive or of a set of flags. The most important ones are:

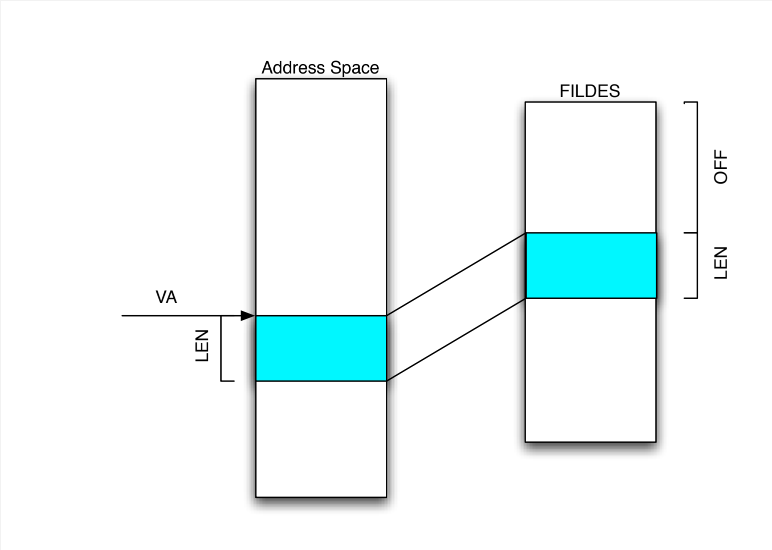
`MAP_SHARED`: Any update made to the mapped region will be **global**, hence it will be seen by any other process.

`MAP_PRIVATE`: The updates will be kept **private** to each process (copy on write).

`MAP_FIXED`: Forces the system to obey the suggestion given by `ADDR`. Its use requires a deep knowledge of the address space organization adopted by the operating system, to avoid damaging it.

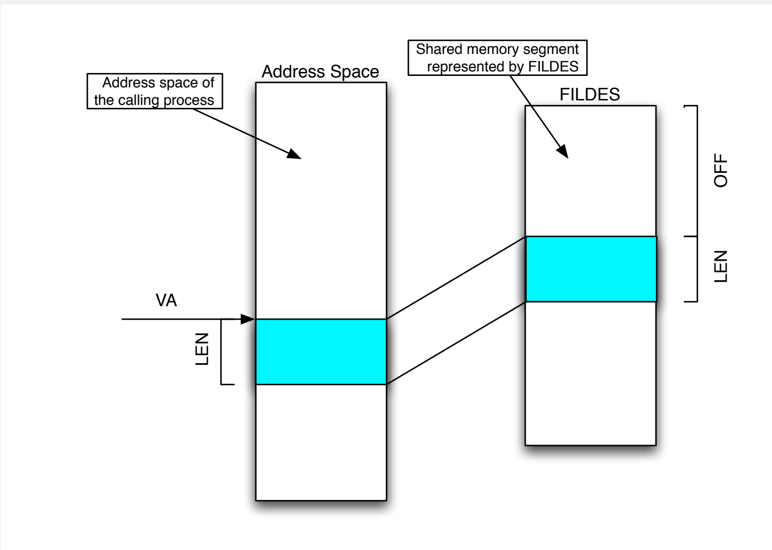
Mapping – Summary

`mmap` returns the starting address `VA` of the mapped region inside the caller's address space. It returns a null pointer on error.



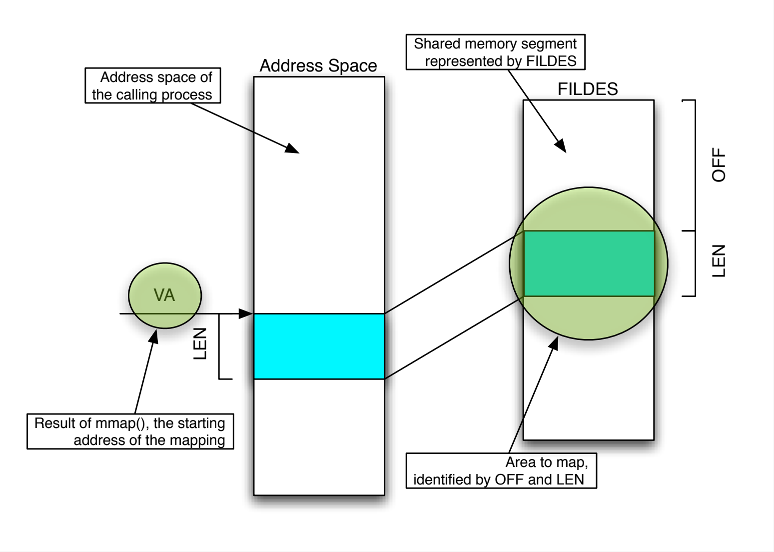
Mapping – Summary

`mmap` returns the starting address `VA` of the mapped region inside the caller's address space. It returns a null pointer on error.



Mapping – Summary

`mmap` returns the starting address `VA` of the mapped region inside the caller's address space. It returns a null pointer on error.



Mapping – munmap()

The function:

```
int munmap(
   void *VA,
   size_t LEN);
```

removes any mapping (made by `mmap`) from the portion of the caller's address space that starts at address **`VA`**, for **`LEN`** bytes.

- `munmap` returns a negative value on error.
- Both `mmap` and `munmap` work only if addresses are correctly aligned to a multiple of the memory page size in use by the system.
- For the sake of portability, the page size can be obtained through the `sysconf` function (with argument `_SC_PAGESIZE`).

Message Queues

The main functions dealing with **message queues** are:

Nome	Scopo
mq_open	Open/create a message queue
mq_close	Close a message queue
mq_receive	Receive a message
mq_send	Send a message
mq_notify	Asynchronous notification
mq_timedreceive	Timed variant of mq_receive
mq_timedsend	Timed variant of mq_send
mq_getattr	Get message queue attributes
mq_setattr	Set message queue attributes
mq_unlink	Remove a message queue

Include mqueue.h before using any of these functions.

Open/Close/Unlink

The functions:

```
mqd_t mq_open(const char *NAME, int OFLAG, ...);
   int mq_close(mqd_t MQDES);
   int mq_unlink(const char *NAME);
```

do the same as their counterparts for named semaphores, but operate on message queues, represented by an opaque data type, mqd_t. When mq_open is invoked to create a new message queue, its **optional arguments** are used to denote its attributes.

struct mq_attr

The `mq_attr` structure represents the attributes of a message queue, and contains the following fields:

`long mq_flags`: Flags (O_NONBLOCK).
`long mq_maxmsg`: Maximum number of messages.
`long mq_msgsize`: Maximum size of each message.
`long mq_curmsgs`: Number of messages currently in the mailbox.

It is possible to get the attributes of a message queue after its creation, by means of `mq_getattr`, and set `mq_flags` by means of `mq_setattr`.

Sending a Message

The function:

```
int mq_send(
  mqd_t MQDES,
  const char *MSG_PTR, size_t MSG_LEN,
  unsigned MSG_PRIO);
```

sends the message pointed by **MSG_PTR**, of **MSG_LEN** bytes, to the message queue **MQDES** and returns a non-zero **integer** on error.

- The **MSG_PRIO** argument is the message priority, between `0` e `MQ_PRIO_MAX` (defined in `mqueue.h`).
- `mq_timedsend` is the timed variant of `mq_send` and allows the caller to specify an absolute time limit to complete the send.

Receiving a Message

The function:

```
ssize_t mq_receive(
    mqd_t MQDES,
    char *MSG_PTR, size_t MSG_LEN,
    unsigned *MSG_PRIO);
```

receives the oldest message with the highest priority currently residing in the message queue **MQDES**, stores it starting at address **MSG_PTR** up to a maximum of **MSG_LEN** bytes, and removes it from the message queue.

- The function returns either the actual **size** of the message just received, or a negative value on error.
- It also stores into the location pointed by **MSG_PRIO** the priority of the message just received.
- `mq_timedreceive` is the timed variant of `mq_receive` and allows the caller to specify an absolute time limit to complete the reception.

Blocking in mq_send and mq_receive

mq_send

If the message queue is **full**, the behavior depends on the value of the `O_NONBLOCK` flag in the `mq_flags` of the queue:

- if the flag is set, the function fails immediately;
- else, it waits until there is enough space in the queue to perform the send.

mq_receive

If the message queue is **empty**, the behavior depends on the value of the `O_NONBLOCK` flag in the `mq_flags` of the queue:

- if the flag is set, the function fails immediately;
- else, it waits until a message becomes available.

Asynchronous Notification

The function:

```
int mq_notify(
   mqd_t MQDES,
   const struct sigevent *NOTIFICATION);
```

allows the calling process to register for an asynchronous notification about the availability of messages in the message queue **MQDES**. The notification will be performed as specified by the **NOTIFICATION** argument, and will be carried out when the queue transitions from an **empty** to a **non-empty** state. At the same time, the registration will be removed.

The notification is performed as specified in the `.sigev_notify` field of `struct sigevent`.

Notification Mechanisms

The notification can be performed in three different ways, depending on the value of the `.sigev_notify` field of `struct sigevent`:

1. **No notification**: any pending registration is removed. In the future the process will explicitly wait for a message to arrive (for example, by calling `mq_receive`).
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`.

Memory Management

Nome	Scopo
`mlock`	“Lock” a range of virtual range and force it to reside in main memory.
`mlockall`	Perform `mlock` on the whole address space of the calling process.
`mprotect`	Set the protection attributes of a range of virtual addresses with respect to read, write and execute operations.
`munlock`	“Unlock” a range of virtual addresses.
`munlockall`	“Unlock” the whole address space of the calling process.

Include `sys/mman.h` before using any of these functions.

Asynchronous I/O

- The usual semantics of the *read* and *write* system calls is **synchronous**: the calling process waits until the operation it requested has been (at least partially, for *write*) executed.
- For real-time applications, it is often not a good idea to tie so closely a process with I/O timings.
- General-purpose applications may benefit from having the ability of controlling multiple, concurrent I/O operations from a single process, too.

Asynchronous I/O – Functions

The following functions deal with **asynchronous I/O**:

Nome	Scopo
`aio_read`	Read request
`aio_write`	Write request
`aio_cancel`	Cancel a pending I/O request
`aio_error`	Get the result (success/failure) of an operation
`aio_return`	Get the status (bytes read/written) of an operation
`aio_fsync`	Asynchronous synchronization **(!)**
`aio_suspend`	Wait for asynchronous I/O to complete

Include `aio.h` before using any of these functions.

Asynchronous I/O Requests

Each asynchronous I/O request is specified and represented by means of a `struct aiocb` data structure. It contains the following fields:

Tipo	Nome	Scopo
`int`	`aio_fildes`	File descriptor
`off_t`	`aio_offset`	File offset
`volatile void *`	`aio_buf`	I/O buffer pointer
`size_t`	`aio_nbytes`	Transfer size
`int`	`aio_reqprio`	Priority
`struct sigevent`	`aio_sigevent`	Notification mechanism

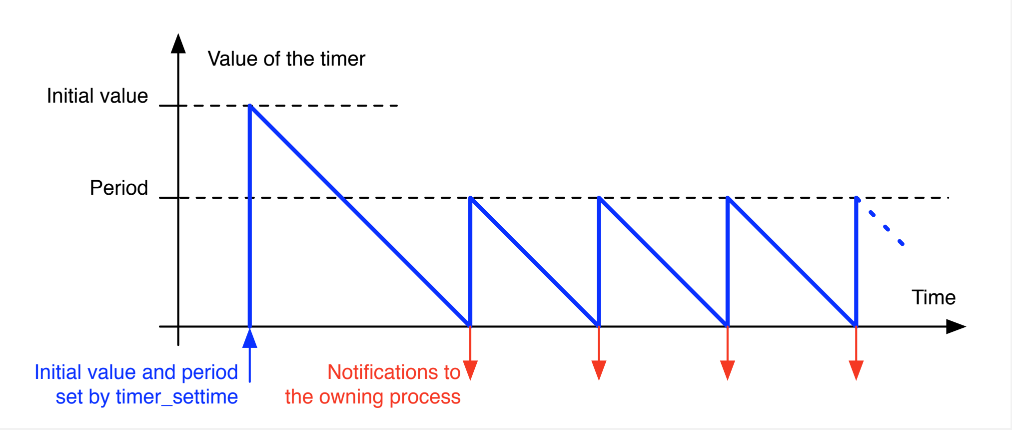
Like with message queues, processes can either wait for an asynchronous I/O request to complete by means of an explicit `aio_suspend`, or associate an asynchronous notification request (to be carried out through either a function or a signal) with each I/O request, by means of the `.aio_sigevent` field.

Clocks & Timers

On POSIX systems, time is expressed and measured by means of:

- One or more *time bases*, or **clocks**, with known resolution and of which it is possible to get the value on request. All systems shall implement, at least, `CLOCK_REALTIME`.
- Zero or more per-process timers, using a specific clock as their timing reference.
- Each timer has its own current value and, optionally, a period, also known as *reload value*.

How are Timers Updated?



- The system decrements each timer with a non-zero value using its clock as a reference.
- A timer expires when its value becomes zero. When a timer expires, the system notifies the owning process.
- If the timer period is not zero, the value of the timer is reloaded from the period whenever the timer expires.

Clock Manipulation

It is possible to get the resolution and the current value of a clock, as well as setting its value (privileged processes only) by means of:

Nome	Scopo
`clock_getres`	Get the resolution of a clock
`clock_gettime`	Get the current value of a clock
`clock_settime`	Set the value of a clock

Time Representation

Both the resolution and the value of a clock are represented through a `struct timespec` with the following fields:

Type	Name	Purpose
`time_t`	`tv_sec`	Seconds
`long`	`tv_nsec`	Nanoseconds

Clock values are relative to an absolute time reference known as the **epoch**. For historic reasons, on Unix systems the reference has been set to January 1st, 1970, 00:00 UTC.

Include `time.h` before using any of these functions.

Timer Manipulation

Each process can create and manage its **timers** by means of the following functions:

Nome	Scopo
`timer_create`	Create a timer
`timer_delete`	Destroy timer
`timer_gettime`	Get the value and period of a timer
`timer_settime`	Set the value and period of a timer
`timer_getoverrun`	Read the *overrrun count*

Include `time.h` before using any of these functions.

Timer Creation

The function:

```
int timer_create(
  clockid_t CLOCKID, struct sigevent *EVP,
            timer_t *TIMERID);
```

creates a fresh timer, local to the calling process, using the clock **`CLOCKID`** as reference, and stores its descriptor into the location pointed by **`TIMERID`**. It returns a non-zero value on error.

- Like with message queues, **`EVP`** specifies how the process shall be notified of expirations. Usually, it is not a good idea to specify “no notification”.
- After creation, the timer is inactive: it must be given a value and (optionally) a period by means of `timer_settime`.

Timer deletion

The function:

```
int timer_delete(
  timer_t TIMERID);
```

destroys the timer **TIMERID** and returns a non-zero value on error.

Timer Value and Period

- The function:

  ```
  int timer_settime(timer_t TIMERID, int FLAGS,
    const struct itimerspec *VALUE,
    struct itimerspec *OVALUE);
  ```

  atomically sets the timer **TIMERID** to a **new** value and period, as specified by **VALUE**, and stores its **old** value and period into the location pointed by **OVALUE**.
- Instead, the function:

  ```
  int timer_gettime(timer_t TIMERID,
    struct itimerspec *VALUE);
  ```

  simply stores into the location pointed by **VALUE** the current value and period of **TIMERID**.

struct itimerspec

The `struct itimerspec` has the following fields:

Type	Name	Purpose
`struct timespec`	`it_value`	Timer value
`struct timespec`	`it_interval`	Period

- Setting the timer value to zero disables the timer.
- A period of zero states that the timer is aperiodic.
- The `FLAGS` argument of `timer_settime` allows the caller to specify whether `it_value` holds a value that is **relative** to the execution time of the function, or an **absolute** value.

Overrun counter

- The standard specifies that each timer can have **no more than one** pending notification, otherwise the amount of memory needed to keep track of them would be unbounded.
- Hence, if a timer expires again while the previous notification is still pending, the new notification is lost.
- However, the function:

```
int timer_getoverrun(timer_t TIMERID);
```

if called while handling a notification, allows the caller to know how many notifications have been lost for `TIMERID` since the last notification that has been handled successfully in the past.