POSIX Real-Time

Mário
de Sousa

msousa@fe.up.pt

POSIX Real-Time

- The POSIX standards

- POSIX for RT Applications
  - Concurrency + Scheduling
  - Mutual exclusion synchronisation
  - Signal/wait synchronisation
  - Asynchronous notification
  - Message passing
  - Timing services
  - Memory management

POSIX Standard

- POSIX → **Portable Operating System Interface for UNIX**
  - Describes the services that the OS must provide,
  - Describes the syntax and semantics of their interfaces (data types and function prototypes)
  - Interfaces defined at source code level => portable source code. Binary level portability is outside the scope of the standard.
  - Implementation of those services is not specified by the standard (left open for each OS vendor to decide how to achieve it)
  - Developed by IEEE, first version in 1988. Most recent version from 2017.

POSIX Standards

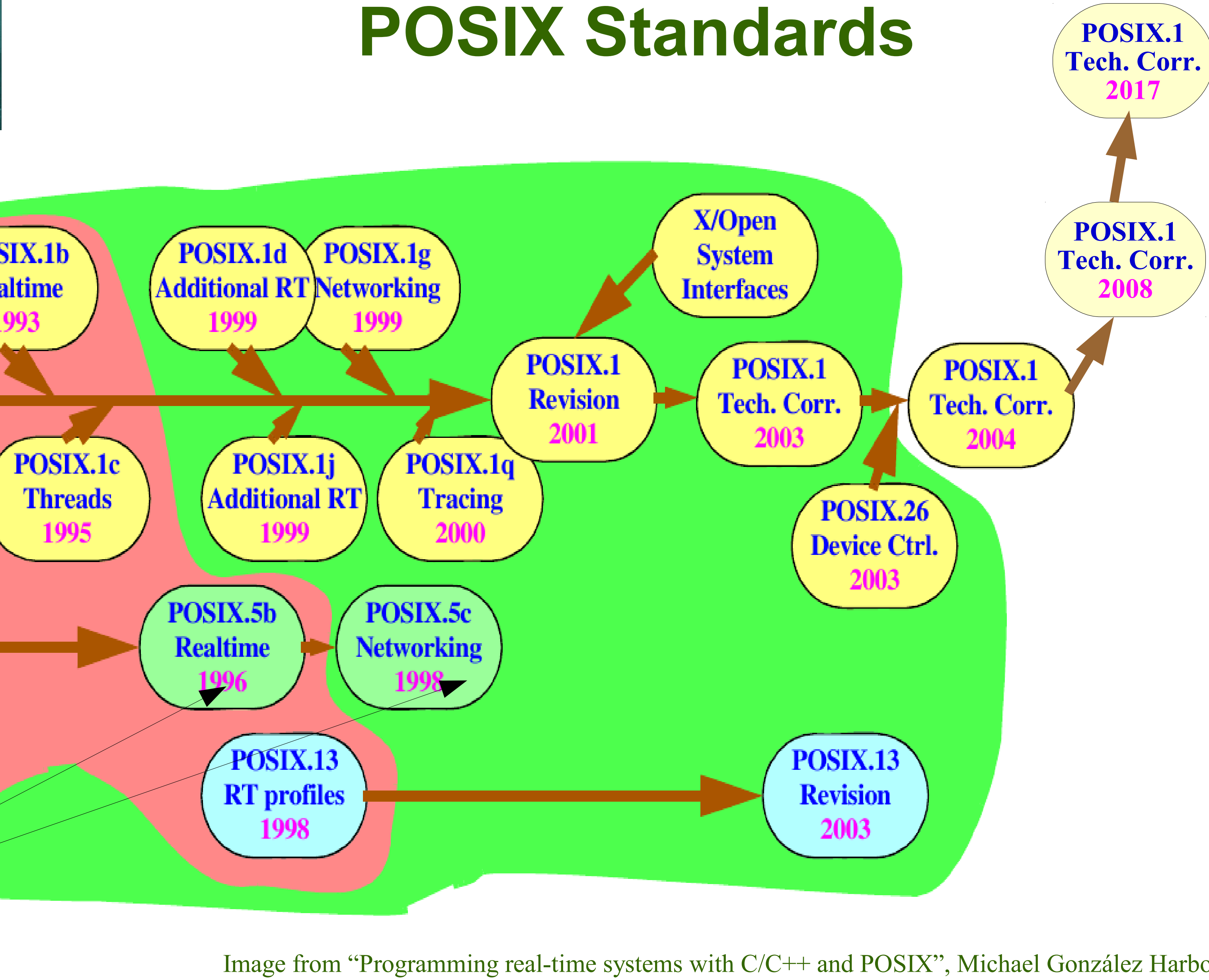


Image from "Programming real-time systems with C/C++ and POSIX", Michael González Harbour

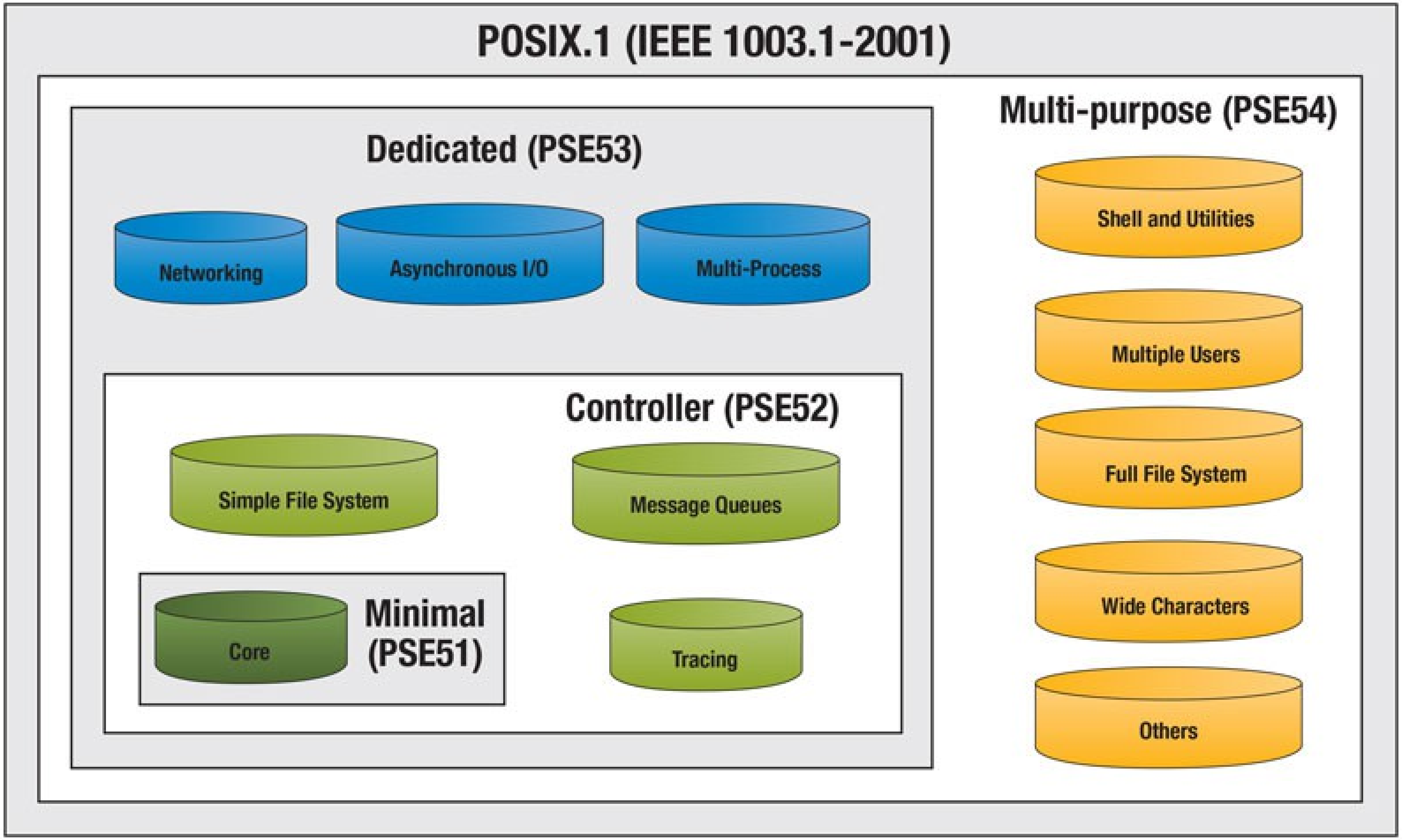
from "The Use of POSIX in Real-time Systems" Kevin M. Obenland

Table 1: POSIX Standards

Standard	Name	Description
1003.1a	OS Definition	Basic OS interfaces; includes support for: (single process, multi process, job control, signals, user groups, file system, file attributes, file device management, file locking, device I/O, device specific, system database, pipes, FIFO, and C language
1003.1b	Real-time Extensions	Functions needed for real-time systems; includes support for: real-time signals, priority scheduling, timers, asynchronous I/O, prioritized I/O, synchronized I/O, file sync, mapped files, memory locking, memory protection, message passing, semaphores, and shared memory
1003.1c	Threads	Functions to support multiple threads within a process; includes support for: thread control, thread attributes, priority scheduling, mutexes, mutex priority inheritance, mutex priority ceiling, and condition variables
1003.1d	Additional Real-time Extensions	Additional interfaces; includes support for: new process create semantics (spawn), sporadic server scheduling, execution time monitoring of processes and threads, I/O advisory information, timeouts on blocking functions, device control, and interrupt control.
1003.1j	Advanced Real-time Extensions	More real-time functions including support for: typed memory, nanosleep improvements, barrier synchronization, reader/writer locks, spin locks, and persistent notification for message queues
1003.21	Distributed Real-time	Functions to support real-time distributed communication; includes support for: buffer management, send control blocks, asynchronous and synchronous operations, bounded blocking, message priorities, message labels, and implementation protocols
1003.1h	High Availability	Services for Reliable, Available, and Serviceable Systems (SRASS); includes support for: logging, core dump control, shutdown/reboot, and reconfiguration

Table 2: POSIX 1003.13 Profiles

Profile	Number of Processes	Threads	File System
54	Multiple	Yes	Yes
53	Multiple	Yes	No
52	Single	Yes	Yes
51	Single	Yes	No



from “The Use of POSIX in Real-time Systems” Kevin M. Obenland

from “POSIX – 25 Years of Open Standard APIs” Arun Subbarao, LynuxWorks

POSIX 1003.1b Real-Time Extensions

- Timers
  Periodic timers, delivery is accomplished using POSIX signals
- Priority scheduling
  Fixed priority preemptive scheduling (minimum of 32 priority levels)
- Real-time signals
  Additional signals with multiple levels of priority
- Semaphores
  Named and memory counting semaphores
- Message queues
- Shared memory
- Memory locking
  Prevent virtual memory swapping of physical memory pages (mlockall() ...)

from "The Use of POSIX in Real-time Systems" Kevin M. Obenland

POSIX 1003.1c Threads

- Thread control
  Creation, deletion and management of individual threads
- Priority scheduling
  POSIX RT scheduling extended to scheduling on a per thread basis
- Mutexes
  Used to guard critical sections of code
  (include support for priority inheritance and priority ceiling protocols)
- Condition variables
  Used with mutexes, are used to create a synchronization point
- Signals
  Ability to deliver signals to individual threads

from "The Use of POSIX in Real-time Systems" Kevin M. Obenland

POSIX Processes and Threads

- The POSIX standards

- POSIX for RT Applications
  - Concurrency + Scheduling
  - Mutual exclusion synchronisation
  - Signal/wait synchronisation
  - Asynchronous notification
  - Message passing
  - Timing services
  - Memory management

Concurrency + Scheduling

- Concurrency
  - Processes or Threads
- Scheduling Algorithms...
  - SCHED_OTHER → Not Fixed Priority. No good for RT!!

Fixed Priority

  - SCHED_FIFO (FIFO for threads/processes of same priority)
  - SCHED_RR (Like FIFO, but with max. quantum execution time)
  - SCHED_SS (Sporadic Server → good for aperiodic tasks)

Support is optional.
Config. Param.:
- replenishment_period
- budget
- high priority
- low priority
- max pending replenishments

All algorithms are compatible, and may co-exist!

Concurrency + Scheduling

POSIX		LINUX
- ...	→	SCHED_IDLE
- SCHED_OTHER	→	SCHED_OTHER, SCHED_BATCH
- SCHED_FIFO	→	SCHED_FIFO
- SCHED_RR	→	SCHED_RR
- SCHED_SS	→	
-	→	SCHED_DEADLINE

Fixed Priority

Implements EDF with CBS (Constant Bandwidth Server)
Config. Param.:
- runtime
- deadline
- period

Concurrency + Scheduling

- **Concurrency**
  - Processes or Threads
- **Scheduling Algorithms...**
  - ...Applied per thread or per process **-> may co-exist !**
    - System contention scope
      All threads in the system compete, regardless of the process to which they belong.
    - Process contention scope
      The scheduler works at two levels: It first chooses a process according to its priority, and then chooses the highest priority thread of that process.
    - Mixed contention scopes
      some threads have a “system” scope, and other threads have a “process” scope.

Mutual Exclusion Synchronisation

Where tasks must coordinate to atomically access a common resource

- **Mutex**
  - Protecting against Unbounded Priority Inversion...
    - No protection (Priority Inheritance - PTHREAD_PRIO_NONE)
    - immediate priority ceiling (Priority Protection - PTHREAD_PRIO_PROTECT)
      good for static systems where it is possible to determine a priority ceiling
    - priority inheritance (Priority Inheritance - PTHREAD_PRIO_INHERIT)
      useful in dynamic systems where it is impossible to assign a ceiling.
  - Implemented as a variable
    - all threads/processes accessing the mutex must be able to access the mutex var.
      => using mutexes between processes requires mutex var be placed in shared memory
      [mmap()]

Mutexes (em POSIX)

- A mutex is a variable of type pthread mutex_t

```c
#include <pthread.h>
pthread_mutex_t my_lock = PTHREAD_MUTEX_INITIALIZER;
```

```c
void thread1(void *arg) {
  /* outside critical section */
  pthread_mutex_lock(&my_lock);
  /* within critical section */
  pthread_mutex_unlock(&my_lock);
  /* outside critical section */
}
```

```c
void thread2(void *arg) {
  /* outside critical section */
  pthread_mutex_lock(&my_lock);
  /* within critical section */
  pthread_mutex_unlock(&my_lock);
  /* outside critical section */
}
```

Mutexes

```
#include <pthread.h>

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

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

NOTE 1: A mutex must be initialized before use.

```
int pthread_mutex_lock   (pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

NOTE 2: pthread_mutex_trylock() attempts to lock but does not block the calling thread if the mutex is already locked, unlike pthread mutex lock().

Mutexes

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

`pthread_mutexattr_destroy, pthread_mutexattr_init`

→ destroy and initialize the mutex attributes object

`pthread_mutexattr_getprioceiling, pthread_mutexattr_setprioceiling`

→ get and set the prioceiling attribute of the mutex attributes object (REALTIME THREADS)

`pthread_mutexattr_getprotocol, pthread_mutexattr_setprotocol`

→ get and set the protocol attribute of the mutex attributes object (REALTIME THREADS)

(PTHREAD_PRIO_NONE, PTHREAD_PRIO_INHERIT, PTHREAD_PRIO_PROTECT)

`pthread_mutexattr_getpshared, pthread_mutexattr_setpshared`

→ get and set the process-shared attribute (to allow sharing between processes)

`pthread_mutexattr_getrobust, pthread_mutexattr_setrobust`

→ get and set the mutex robust attribute (what to do if thread is terminated while holding mutex:
(PTHREAD_MUTEX_STALLED → do nothing; PTHREAD_MUTEX_ROBUST → notify next thread attempting a mutex_lock())

`pthread_mutexattr_gettype, pthread_mutexattr_settype`

→ get and set the mutex type attribute (changes semantics of pthread_lock(): allow recursive locking, ...)

(PTHREAD_MUTEX_NORMAL, MUTEX_ERRORCHECK, MUTEX_RECURSIVE, MUTEX_DEFAULT)

Mutexes

Semantics of: `pthread_mutex_lock()`
`pthread_mutex_unlock()`

Mutex Type	Robustness	Relock When Owner	Unlock When Not Owner
NORMAL	non-robust	deadlock	undefined
NORMAL	robust	deadlock	error returned
ERRORCHECK	either	error returned	error returned
RECURSIVE	either	recursive (counting lock)	error returned
DEFAULT	non-robust	undefined	undefined
DEFAULT	robust	undefined	error returned

POSIX standard: IEEE 1003

Signal/Wait Synchronisation

Where tasks must synchronise the execution of actions

- Counting Semaphores
- Condition Variables
  - Used in conjunction with a mutex

    Allows checking of complex synchronisation conditions while mutex is held

Semaphores

- A semaphore is a variable of type sem_t

```
#include <semaphore.h>
sem_t my_semaphore;

int sem_init(sem_t *sem, int pshared,
                unsigned int value);
int sem_destroy(sem_t *sem);

int sem_wait(sem_t *sem);     /* down */
int sem_trywait(sem_t *sem);  /* down, no wait */
int sem_timedwait(sem_t *sem, … abstime);
int sem_post(sem_t *sem);     /* up */
int sem_getvalue(sem_t *sem, int *sval); /* get current value */
```

Asynchronous Notification

Where tasks must be asynchronously notified of event occurrence

Signals

- When issued, a signal handler function is executed
- Signal is sent to any of the threads interested in that signal.
  Best is to have a single thread interested in each signal.

Sending a signal:

kill: `kill(pid_t pid, int sig)`
→ send a signal to a process or a group of processes

killpg: `killpg(pid_t pgrp, int sig);`
→ send a signal to a process group

pthread_kill: `pthread_kill(pthread_t thread, int sig)`
→ send a signal to a thread

sigqueue: `sigqueue(pid_t pid, int signo, union sigval value)`
→ send a signal to a process, including a value

Asynchronous Notification

Where tasks must be asynchronously notified of event occurrence

Receiving a signal:

sigwait: `sigwait(const sigset_t *restrict set, int *restrict sig)`

→ wait for queued signals (only waits for signals specified in sigset)

sigaction: `sigaction(int sig, const struct sigaction *restrict act, struct sigaction *restrict oact)`

→ specify action to take when receiving a signal

```
struct sigaction

    void(*) (int)                        sa_handler
```
→ Pointer to a signal-catching function or one of the macros SIG_IGN or SIG_DFL.

```
    sigset_t                             sa_mask
```
→ set of signals to be blocked during execution of signal-catching function.

```
    int                                  sa_flags
```
→ Special flags to affect behavior of signal.

```
    void(*) (int, siginfo_t *, void *)   sa_sigaction
```
→ Pointer to a signal-catching function.

Message Passing

Asynchronously pass data between tasks, using message queues

- Message Queues
  - Support variable sized messages
  - Supports
    - polling for message availability, or
    - block while waiting for message (may have timeout)
  - Message passing between processes or threads

Timing Services

Synchronise with Time!

- **Clocks**
  - CLOCK_REALTIME
    Represents the oficial time - subject to changes
    (e.g. setting the clock, adjusting by clock synchronisation services, ...)
  - CLOCK_MONOTONIC
    Monotonic, at constant rate
    (like 'realtime' but not subject to any adjustments)
  - Execution time clocks
    Based on execution time of system, process, thread, etc...
    CLOCK_PROCESS_CPUTIME_ID, CLOCK_THREAD_CPUTIME_ID

Timing Services

Synchronise with Time!

- Clocks

  Available services based on these clocks:
  - Sleep until a clock reaches an absolute time
  - Sleep for some (relative) time interval
  - Create a timer (software entity) that will notify...
    - notify when clock reaches an absolute time
    - notify when an interval has elapsed.
    - expire periodically.

Execution time clocks may be used to monitor the CPU usage from a given thread, and make sure it does not produce overload situation

Memory management

Manage unpredictable memory management

- **Swapping to disk**
  - Allows locking memory into RAM (mlockall())
  - Should also consider allocating all required memory at startup (malloc)
    Not POSIX specific!!

POSIX Processes and Threads

- The POSIX standards

- POSIX for RT Applications
  - Concurrency + Scheduling
  - Mutual exclusion synchronisation
  - Signal/wait synchronisation
  - Asynchronous notification
  - Message passing
  - Timing services
  - Memory management

- Examples...

Process creation in POSIX

```
#include <unistd.h>
pid_t fork(void); /*clones the calling process*/
```

- Create a new child process
  - Child process is a clone of parent process (executes the same program, with almost identical state, as parent)
  - Child process starts at instruction following the call to **`fork()`**
- Child proess has its own:
  - pid – Process Identifier)
  - Memory area (and variables stored there) Changes to the parent's memory area are not visible in the child's memory, and vice-versa

Process creation in POSIX

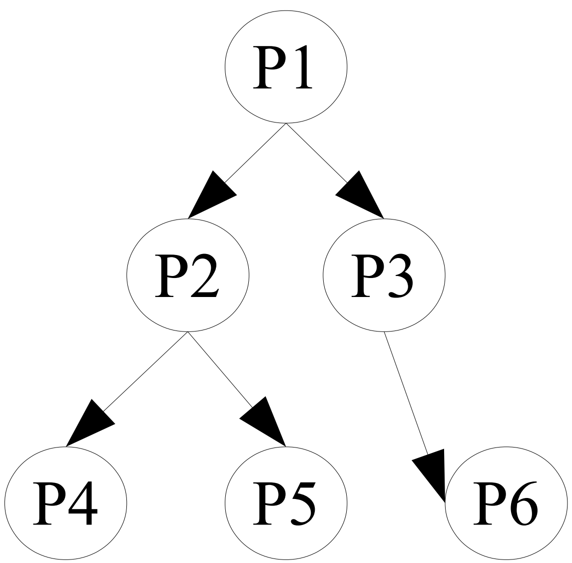
```
#include <unistd.h>

(...)
pid_t pid = fork();
if (pid == 0) {
  processo_filho();
} else {
  processo_pai();
}
```

- The call to fork() returns:
  - Child process: 0
  - Parent process: the child's pid

Process Hierarchy

- In POSIX there is a relationship between:
  - Parent process and its children
  - Processes with a common parent (process group)




```
$pstree
 init-+-apmd
      |-atd
      |-bdflush
      |-crond
      |-kapmd
      |-kdeinit-+-artsd
      |         |-evolution-alarm
      |         |-kdeinit---bash---pstree
      |         |-kdeinit---bash---ssh
      |         `-soffice.bin---
soffice.bin
      |-12*[kdeinit]
 (...)
```

Program Execution

- **Problem**: How can a child process execute a program that is different to parent's?
- **Solution**: use the system call **`execve()`**

```
#include <unistd.h>
int execve( const char *file,
                char *const argv[],
                char *const envp[]);
```

- After call, process will execute program stored in **`file`**
  - **`argv`** and **`envp`** specify the arguments to pass the new program's **`main()`** function

Process Termination

- A process may terminate by:
  - Its own initiative, by calling:
    `void _exit(int status)`
    (May also be called indirectly - by executing return in **`main()`**)
  - Executing an invalid instruction
    (e.g.: division by zero, attemted access to another process' memory area, …)
  - Another process, that calls:
    `int kill(pid_t pid, int sig)`
  - Decision of the operating system
    (insufficient resources, ...)

More System Calls...

```
#include <unistd.h>
void _exit(int status)
pid_t getpid(void);
pid_t getppid(void);
```

```
#include <stdlib.h>
void exit(int status)
```

- `getpid()` returns `pid` of calling process
- `getppid()` returns `pid` of parent process
- `_exit()` terminates calling process
- `exit()` is a C library function
  - Calls all functions registered through `at_exit()`
  - Calls the system call `_exit()`

Process Synchronisation

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

- Suspends the process execution until:
  - Any child process terminates (`wait()`)
  - The child process identified by `pid` terminates (`waitpid()`)
- `status` may be used to determine the status value the child process passed to the function `void _exit(int status)`
- `options` defines the optional semantics of `waitpid()`

Process Synchronisation example...

```
...
int status;
pid_t child_pid;
...

if ((child_pid = wait(&status)) != -1) {
  if (WIFEXITED(status) != 0)
    printf(“Process %d exited with status %d\n”,
           pid, WEXITSTATUS(status));
  else
    printf(“Process %d exited abnormally\n”, pid);
}
```

POSIX Thread Interface: libpthreads

```
int pthread_create( pthread_t *id,
                    const pthread_att_t attr,
                    void *(*start_fn)(void *),
                    void *arg)
```

- *id
  - initialized with the new thread's identifier;
- *attr
  - data structure used to configure pthread creation semantics.
  - May be initialized with default values

    ```
    int pthread_attr_init(pthread_attr_t *attr)
    ```

POSIX Thread Interface: libpthreads

```
int pthread_create( pthread_t *id,
                    const pthread_att_t attr,
                    void *(*thr_fun)(void *),
                    void *arg)
```

- *thr_fun
  - The function to be called
  - Must have the following prototype: `void *thr_fun(void *)`
- *arg
  - The data value to pass the function thr_fun().

POSIX Thread Interface: libpthreads

```
#include <pthread.h>
void *my_fun(void *arg) {
      ...
}

int main(void) {
   pthread_attr_t attr;
   pthread_t tid;
   int my_arg = 42;
      /* Initialize attr with default values */
   pthread_attr_init(&attr);
      /* create thread... */
   pthread_create(&tid, &attr, my_fun, &my_arg);
   ...
```

POSIX Thread Interface: libpthreads

```
void pthread_exit(void *value ptr)
```

Terminate the thread

```
int pthread_join(pthread t thread, void **value prt)
```

Wait until thread identified by `thread` terminates.

```
int pthread_detach(pthread_t thread);
```

Resources are released back to the system without the need for another thread to join with the terminated thread.

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

Compares two thread identifiers.

POSIX Thread Interface: libpthreads

```
/* initialize thread attributes structure */
int pthread_attr_init(pthread_attr_t *attr);

/* destroy thread attributes structure */
int pthread_attr_destroy(pthread_attr_t *attr);
```

```
/* Set/Get thread detach state attribute */
int pthread_attr_setdetachstate(pthread_attr_t *attr,
                                int detachstate);

int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                                int *detachstate);
```

`detachstate:`

`PTHREAD_CREATE_DETACHED` → **created threads in a detached state.**
i.e. thread destroys all local data when it terminates
Without waiting pthread_join() to be called.

`PTHREAD_CREATE_JOINABLE` → **created threads in a joinable state.**
i.e. possible to call pthread_join() on the thread.

POSIX Thread Interface: libpthreads

```
/* Set/Get thread scheduling policy */
int pthread_attr_setschedpolicy(pthread_attr_t *attr,
                                int policy);

int pthread_attr_getschedpolicy(const pthread_attr_t *attr,
                                int *policy);
```

`policy:`

- `SCHED_FIFO` → **created threads using FIFO scheduling algorithm**
- `SCHED_RR` → **created threads using RR scheduling algorithm**
- `SCHED_OTHER` → **created threads using OTHER scheduling algorithm**

```
/* Set thread scheduling priority */
int pthread_setschedprio(pthread_t thread, int prio);
```

POSIX Thread Interface: libpthreads

```
/* Set/Get thread scheduling policy and priority */
int pthread_setschedparam(pthread_t thread,
                          int policy,
                          const struct sched_param *param);

int pthread_getschedparam(pthread_t thread,
                          int *policy,
                          struct sched_param *param);


struct sched_param {
    int sched_priority;/* Scheduling priority */
};
```

POSIX Thread Interface: libpthreads

```
/* Set/Get thread scheduling contention scope */
int pthread_attr_setscope(pthread_attr_t *attr, int scope);

int pthread_attr_getscope(const pthread_attr_t *attr, int *scope);
```

`scope`:

- `PTHREAD_SCOPE_SYSTEM` → created thread uses scheduling contention using system scope
- `PTHREAD_SCOPE_PROCESS` → created thread uses scheduling contention using process scope

POSIX Thread Interface: libpthreads

```
/* Set/Get thread stack size attribute */
int pthread_attr_setstacksize(pthread_attr_t *attr,
                              size_t stacksize);

int pthread_attr_getstacksize(const pthread_attr_t *attr,
                              size_t *stacksize);
```

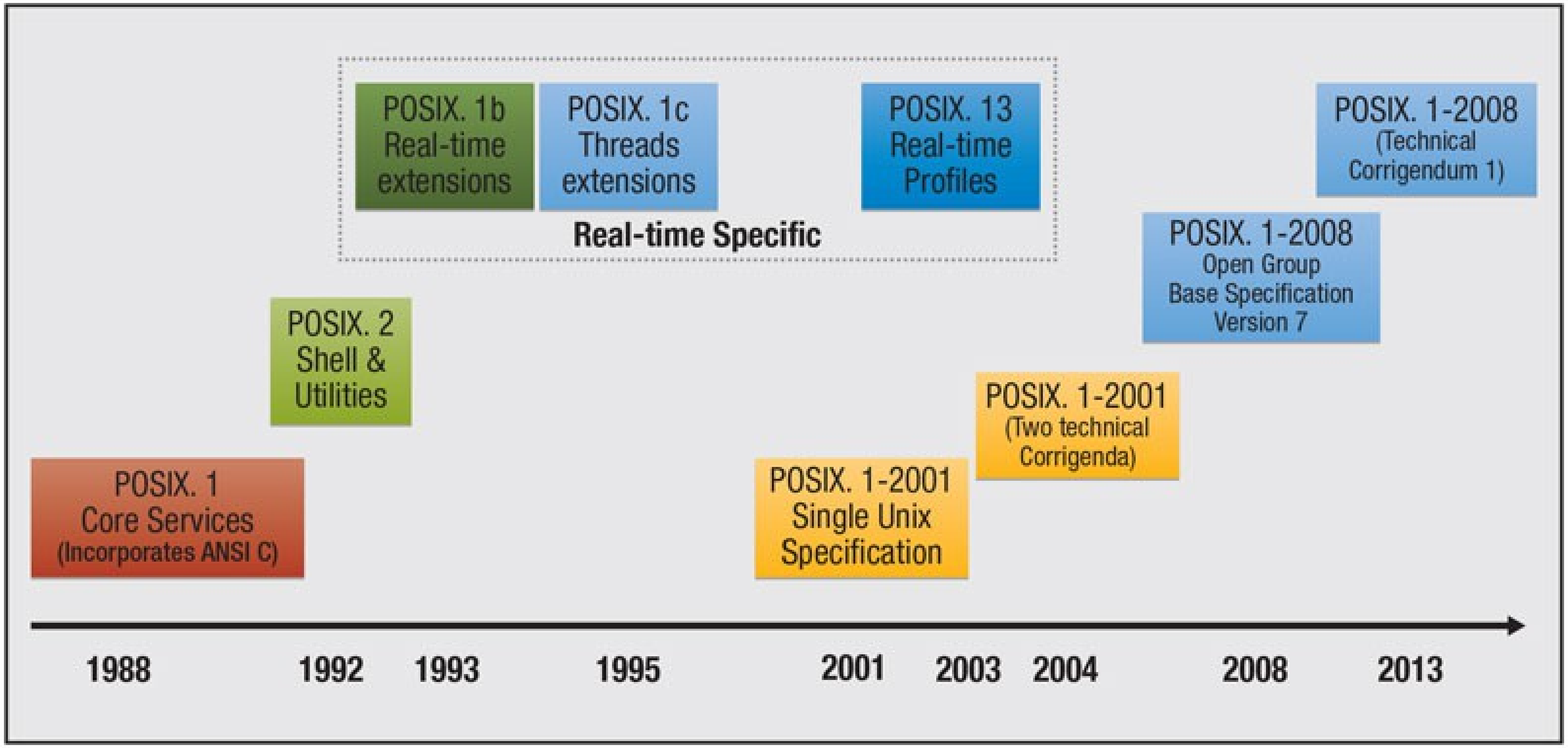
`stacksize` → minimum size in bytes

```
/* Set/Get thread stack address */int
pthread_attr_setstackaddr(pthread_attr_t *attr,
                          void *stackaddr);

int pthread_attr_getstackaddr(const pthread_attr_t *attr,
                              void **stackaddr);
```


JAPAN!

Important POSIX Standards for RT



from "POSIX – 25 Years of Open Standard APIs" Arun Subbarao, LynuxWorks