POSIX 1003.1b
Portable Operating System Interface (for uniX)
Real-time operating systems
POSIX

• Goals:
  – To support application portability at source-code level
  – Interface, not implementation
  – The C language
  – Minimal changes to historical implementations
  – Minimal interface + extensions

• Standards:
  – Now:
    • POSIX = IEEE Std 1003.n and parts of ISO/IEC 9945
    • POSIX.1 = IEEE Std 1003.1-2004
Real-time POSIX

- IEEE Std 1003.1b-1993 Realtime Extension (formerly POSIX.4)
- IEEE Std 1003.1c-1995 Threads
- IEEE Std 1003.1d-1999 Additional Realtime Extensions
- IEEE Std 1003.1j-2000 Advanced Realtime Extensions
- IEEE Std 1003.1q-2000 Tracing
POSIX.1 and POSIX.4
mandatory and optional parts

POSIX 1003.1
- Mandatory Parts
  - POSIX1003.1
    - POSIX.1 Base
  - POSIX 1003.1b
    - POSIX.4 Base

POSIX 1003.1b
- Optional Parts
  - POSIX Chown Restricted
  - Shared Memory
  - Memory Locking
  - SYNCH_I/O
  - ASYNCH_I/O
  - Message Queues
  - Semaphores
  - Timers
  - POSIX Saved IDS
  - Realtime Signals
  - Priority Scheduling
  - POSIX NO Trunc
Posix 1003.1b - overview

• Coordination mechanisms
  – *Realtime* signals
  – General semaphores
  – Shared memory (and memory mapped files)
  – Message queues

• *Realtime* clocks and timers

• Scheduling

• Memory locking

• Asynchronous and synchronous I/O
What’s in the box (1)

• Qualitative properties = what is implemented

```c
#include <unistd.h>
```

– Constants with `_POSIX_`, prefix e.g.
  `_POSIX_TIMERS`, `_POSIX_MEMLOCK`,
  `_POSIX_IPV6`

• Quantitative properties = how many

```c
#include <limits.h>
```

– `_MAX` constants e.g. `OPEN_MAX`, `TIMER_MAX`,
  `AIO_MAX`, `RTSIG_MAX`
• Run-time checking

#include <unistd.h>

/* Parameters valid for the whole OS */
long sysconf(int option);
e.g. sysconf(_SC_OPEN_MAX);
/* Parameters depending on directory/file */
long pathconf(char *pathname, int option);
long fpathconf(int fd, int option);
e.g. pathconf(“/home”, _PC_NAME_MAX);
POSIX 1003.1(b) – API conventions

- For `int` return type, -1 represents error, 0 (or positive number) success
- In case of error, the global variable `errno` contains the error code (`#include <errno.h>`)  
  - Each thread has its own value of `errno`
- For pointer return types, `NULL` or -1 indicates an error
- If a special type is returned, it can often be typecasted to `int` and -1 indicates an error
Example error handling

```c
int fd = open("/etc/passwd", O_RDWR);
if (fd == -1) {
    perror("/etc/passwd");
    exit(1);
}
```

- perror prints string representation of errno
- e.g.: if errno == EPERM => 
  /etc/passwd: Operation not permitted
Tips & tricks

• Always start the names of named objects with „/“, then don't use this character again in the name.

• In most times, the existence of named object at creation time indicates problems. Create named objects with \( (O\_CREAT|O\_EXCL) \) flags and in case of error check \( \text{errno} \) for detailed information.
Clocks & Timers
Clocks and timers

- Clocks are used to determine actual time
- Timers generate time intervals and periodic intervals of the same length
- Data type with nanosecond resolution
- Ability to determine resolution of each clock
- Notification by RT signal on timer expiration
- Timer overruns are detected
/* Header file */
#include <time.h>

/* Constants */
• CLOCK_REALTIME – System real-time clock
• CLOCK_MONOTONIC – Monotonic clock; cannot be set
• CLOCK_PROCESS_CPUTIME_ID – measures process CPU time
• CLOCK_THREAD_CPUTIME_ID – measures thread CPU time

/* POSIX.4 time data type */
struct timespec {
    time_t    tv_sec; /* seconds (POSIX.1 type) */
    long      tv_nsec; /* nanoseconds */
}; /* Type for time, interval, time resolution */
CLOCKS – API (2)

/* Get clock resolution */
int clock_getres(clockid_t clock_id);

/* Get clock's actual time */
int clock_gettime(clockid_t clock_id, struct timespec *current_time);

/* Set clock's actual time */
int clock_settime(clockid_t clock_id, struct const timespec *new_time);

/* Delay the process execution */
int nanosleep(const struct timespec *interval, struct timespec *remaining_time);

/* High-resolution sleep with specifiable clock and absolute/relative timeout */
int clock_nanosleep(clockid_t clock_id, int flags, const struct timespec *request, struct timespec *remain);
Timers – API (3)

/* Create/delete timer */
int timer_create(clockid_t clock, struct sigevent *sigev, timer_t *created_timer);
int timer_delete(timer_t timer_to_delete);

/* Set/get timer interval */
int timer_settime(timer_t timer, int flags, const struct itimerspec *val, struct itimerspec *old);
int timer_gettime(timer_t timer, struct itimerspec *oldvalue);

/* Get number of timer overruns */
int timer_getoverrun(timer_t timerid);
Memory locking
Memory locking

• Prevents non-predictable delays caused by page faults (e.g. swapping memory to disk)
• This is not the same as “locking” a mutex!
• Useful for time critical processes
• Basic variant – lock all process memory
• Extended variant – lock a specified part of memory
• Use of extended variant depends on the compiler/linker
/* Header file */
#include <sys/mman.h>

/* Lock/unlock all process memory */
int mlockall(int flags);
int munlockall(void);

/* Lock/unlock memory area */
int mlock(void *address, size_t length);
intmunlock(void *address, size_t length);
Synchronous and asynchronous I/O
Synchronous and asynchronous I/O

• When are data really stored to the disk?
  – synchronized I/O gives program control over it

• Why I have to wait for all I/O?
  – asynchronous I/O allows execution of I/O in parallel with the process execution
  – Solves the problem of waiting for multiple I/O operations
Synchronous I/O – API (1)

/* Header file */
#include <unistd.h>

/* Constants */
• F_GETFL  - determine the sync. file mode (POSIX.1)
• F_SETFL  - change the sync. file mode (POSIX.1)
• O_NONBLOCK - read/write operations doesn't blocks the process (POSIX.1)
• O_DSYNC  - synchronization during write
• O_SYNC   - O_DSYNC  + sync. Information stored in inodes
• O_RSYNC  - synchronize inode information for reading (O_SYNC read equivalent)
Synchronous I/O – API (2)

/* Change file mode (POSIX.1) */
int fcntl(int fd, int oper, ...);

/* Write data and metadata (file size etc.) to the file */
int fsync(int fd);

/* Write only data (without metadata) to the file (faster, possible problems after system crash.) */
int fdatasync(int fd);
Asynchronous I/O – API (1)

/* Header file */
#include <aio.h>

/* AIO control block */
struct aiocb {
    int    aio_fildes; /*I/O device/file FD */
    off_t  aio_offset; /* Offset in the file */
    volatile void *aio_buf; /* read/write buffer */
    struct sigevent aio_sigevent; /*notif. signal*/
    int    aio_lio_opcode; /* Requested operation */
    int    aio_rqsprio; /* AIO priority */
};
Asynchronous I/O – API (2)

/* Asynchronous input */
int aio_read(struct aiocb *read_aiocb);
/* Asynchronous output */
int aio_write(struct aiocb *write_aiocb);
/* Cancel asynchronous operation */
int aio_cancel(struct aiocb *cancel_aiocb);
/* Get (error) state of finished AIO operation */
ssize_t aio_return(struct aiocb *cancel_aiocb);
/* Get actual state of running/finished AIO operation, it can be called repeatedly */
int aio_error(struct aiocb *cancel_aiocb);
Asynchronous I/O – API (3)

/* Wait for completion of multiple AIO */
int aio_suspend(struct aiocb *laiocb[], int nent, const struct timespec *timeout);

/* Constants */
• LIO_READ, LIO_WRITE, LIO_NOP - typ operace
• LIO_WAIT, LIO_NOWAIT - blokující/neblokující chování

/* More AIO reads/writes in one call */
int aio_listio(int wait_or_not, struct aiocb *const laiocb[], int nent, struct sigevent *notification);
Threads

- TODO
- Thread cancelation (see POSIX:2008 2.9.5)
Tracing

• TODO
Literature

• Linux manpages
• Michael González Harbour: "REAL-TIME POSIX: AN OVERVIEW"