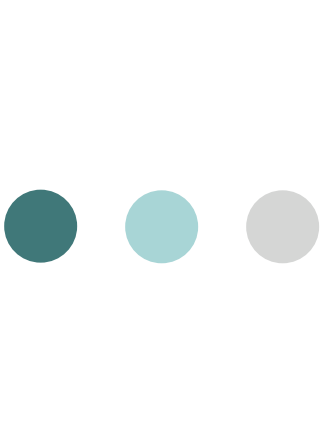


POSIX API

Ricardo.Sanz@etsii.upm.es
http://aslab.org/~sanz

Curso 2005-2006

The Standard

- **IEEE Std 1003.1-2003 IEEE Standard for Information Technology— Portable Operating System Interface(POSIX®)— Part 1: System Application Program Interface (API)**

- **Abstract:** This standard is part of the POSIX series of standards for applications and user interfaces to open systems. It defines the applications interface to system services for synchronization, memory management, time management, and thread management. This standard is stated in terms of its C language binding.
- **keywords:** API, application portability, C (programming language), data processing, information interchange, open systems, operating system, portable application, POSIX, programming language, realtime, system configuration computer interface.

API Releases

- POSIX.1
  - Standard API
- POSIX.1b
  - Real time Extensions
- POSIX.1c
  - Thread extensions

- Last one: POSIX.1j (More real-time)

POSIX.1

- Process Creation and Control
- Signals
- Floating Point Exceptions
- Segmentation Violations
- Illegal Instructions
- Bus Errors
- Timers
- File and Directory Operations
- Pipes
- I/O Port Interface and Control

POSIX.1b
Real time extensions

- Priority Scheduling
- Real-Time Signals
- Clocks and Timers
- Semaphores
- Message Passing
- Shared Memory
- Asynch and Synch I/O
- Memory Locking

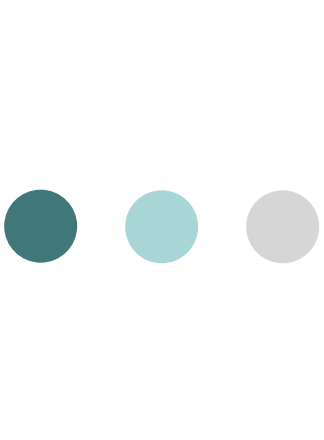
POSIX.1c
Threads extensions

- Thread Creation, Control, and Cleanup
- Thread Scheduling
- Thread Synchronization
- Signal Handling

Interesting site

- The Open Group **Single Unix Specification** website:

  http://www.unix.org/



Example 1

Simple timer using POSIX API

sleep()

NAME

sleep - suspend execution for an interval of time

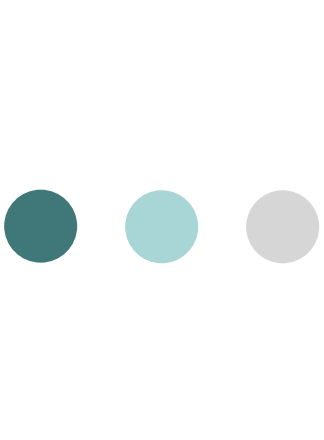
SYNOPSIS

```
#include <unistd.h>

   unsigned sleep(unsigned seconds);
```

DESCRIPTION

The *sleep*() function shall cause the calling thread to be suspended from execution until either the number of realtime seconds specified by the argument *seconds* has elapsed or a signal is delivered to the calling thread and its action is to invoke a signal-catching function or to terminate the process. The suspension time may be longer than requested due to the scheduling of other activity by the system.

```
int main (int argc, char *argv[])
{
int sec;
char line[128];
char msg[64];
while (1) {
  printf ("Alarm> ");
  if (fgets (line, sizeof (line), stdin) == NULL)
     exit (0);
  if (strlen (line) <= 1) continue;
  if (sscanf (line, "%d %64[^\n]", &sec, msg) < 2)
    {
      fprintf (stderr, "Bad command\n");
    }
  else
    {
      sleep (sec);
      printf ("(%d) %s\n", sec, msg);
    }
}
}
```

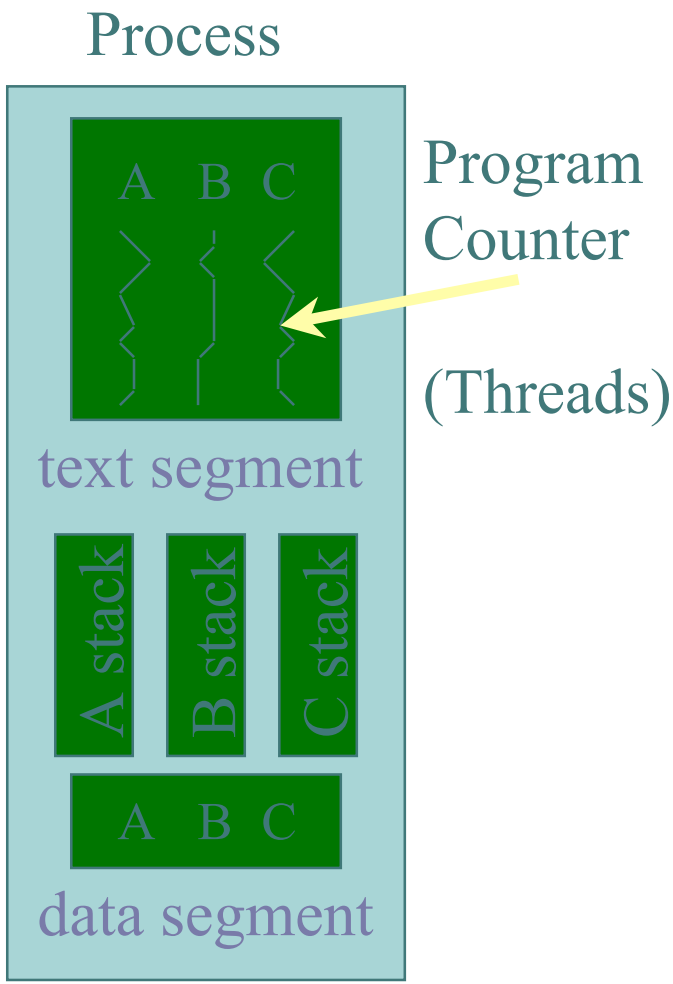


Threads

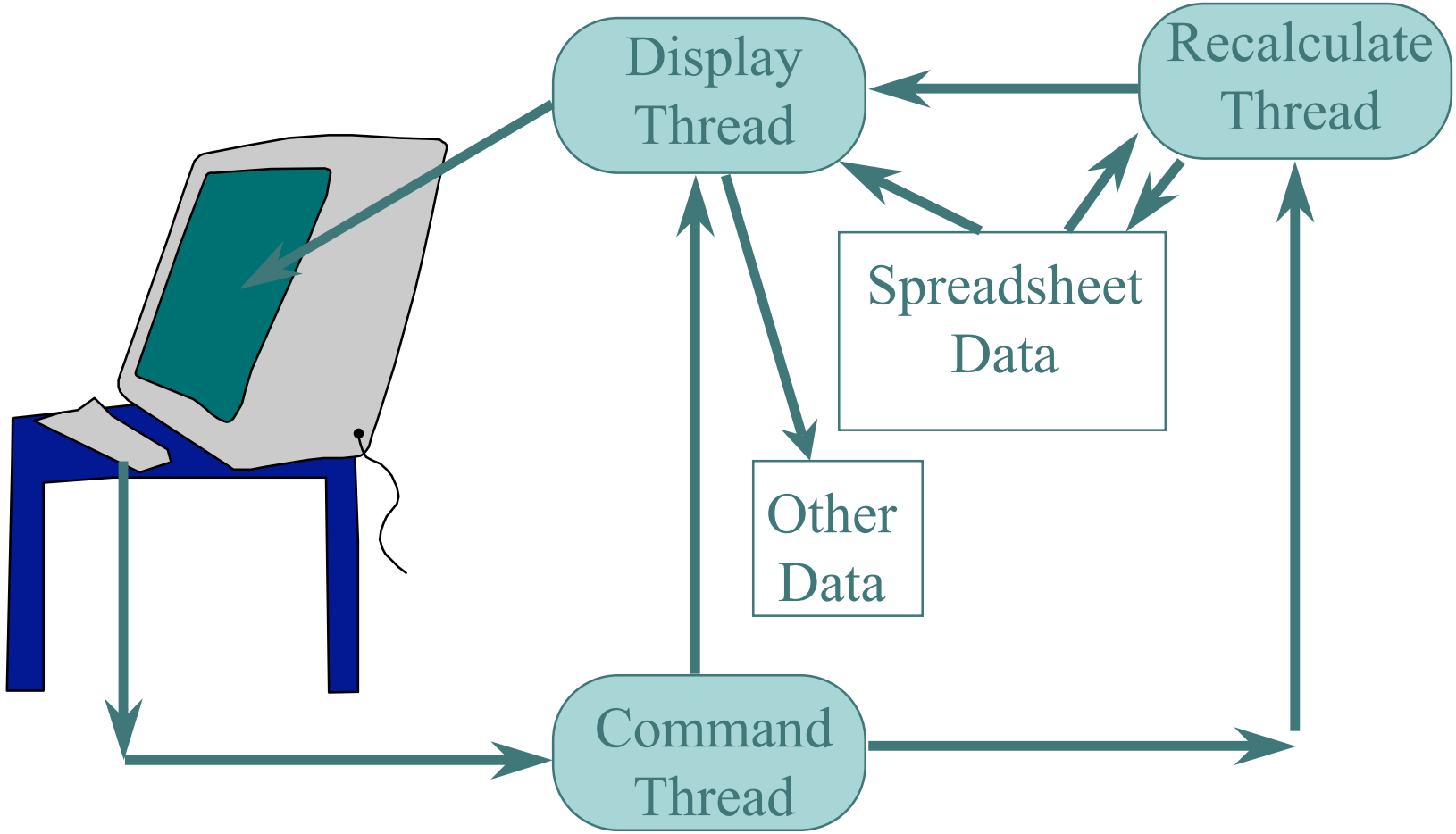
Software Multi-Processors

Threads

- Basic unit of CPU utilization
- Own
  - program counter
  - register set
  - stack space
- Shares
  - code section
  - data section
  - OS resources



Example: A Threaded Spreadsheet



What to Thread?

- Independent tasks
  - ex: debugger needs gui, program, perf monitor…
  - especially when blocking for I/O!
- Single program, concurrent operation
  - Servers
    - ex: file server, web server
  - OS kernels
    - concurrent requests by multiple users -- no protection needed in kernel

Thread Benefits

- "What about just using processes with shared memory?"
  - fine
  - debugging tougher (more thread tools)
  - processes slower
    - 30 times slower to create on Solaris
    - slower to destroy
    - slower to context switch among
  - processes eat up memory
    - few thousand processes not ok
    - few thousand threads ok

Threads Standards

- POSIX threads (pthreads)
  - Common API
  - Almost all Unix's have thread library
- Win32 and OS/2
  - very different from POSIX, tough to port
  - commercial POSIX libraries for Win32
  - OS/2 has POSIX option
- Solaris
  - started before POSIX standard
  - likely to be like POSIX

POSIX Threads

```
#include <pthread.h>
int  pthread_create(
  pthread_t       *thread,
  pthread_attr_t *attr,
  void *(*start_routine)(void *),
  void            *arg);
```

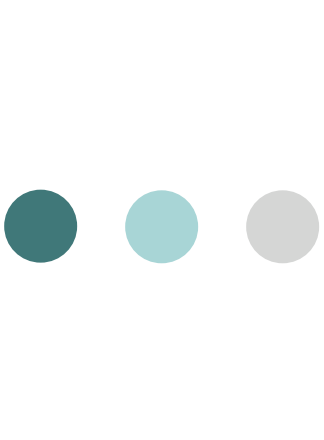
- **pthread_create** creates a new thread of control that executes concurrently with the calling thread.

POSIX Threads

```
#include <pthread.h>
int pthread_join(
  pthread_t th,
  void      **thread_return);
```

- **pthread_join** suspends the execution of the calling thread until the thread identified by th terminates

```
#include <pthread.h>

void *hello (void *arg)
{
 printf ("Hello world\n");
 return NULL;
}

int main (int argc, char *argv[])
{
 pthread_t hello_id;
 int status;

 status = pthread_create (&hello_id, NULL, hello, NULL);
 if (status != 0) exit (1);

 status = pthread_join (hello_id, NULL);
 if (status != 0) exit (2);

 return 0;
}
```



Processes

Those who Live in Computers

Process Management

- A Unix process is the execution of an image of a virtual machine
- This virtual machine remains in memory until its displaced by a higher priority process or terminated by an exit system call or kill signal
- An image is characterized by:
  - Memory in use
  - General register values
  - Status of files opened
  - Default (current) directory

Process Creation & Initialization

- In Unix, process 0 is assigned to the scheduler and is created as part of the system boot process
- Every other process is created as the result of a *fork* or *vfork* system call
- The *fork* and *vfork* system calls split a process into two processes
- The process that calls *fork/vfork* is the *parent process*
- The newly created process is known as the *child process*

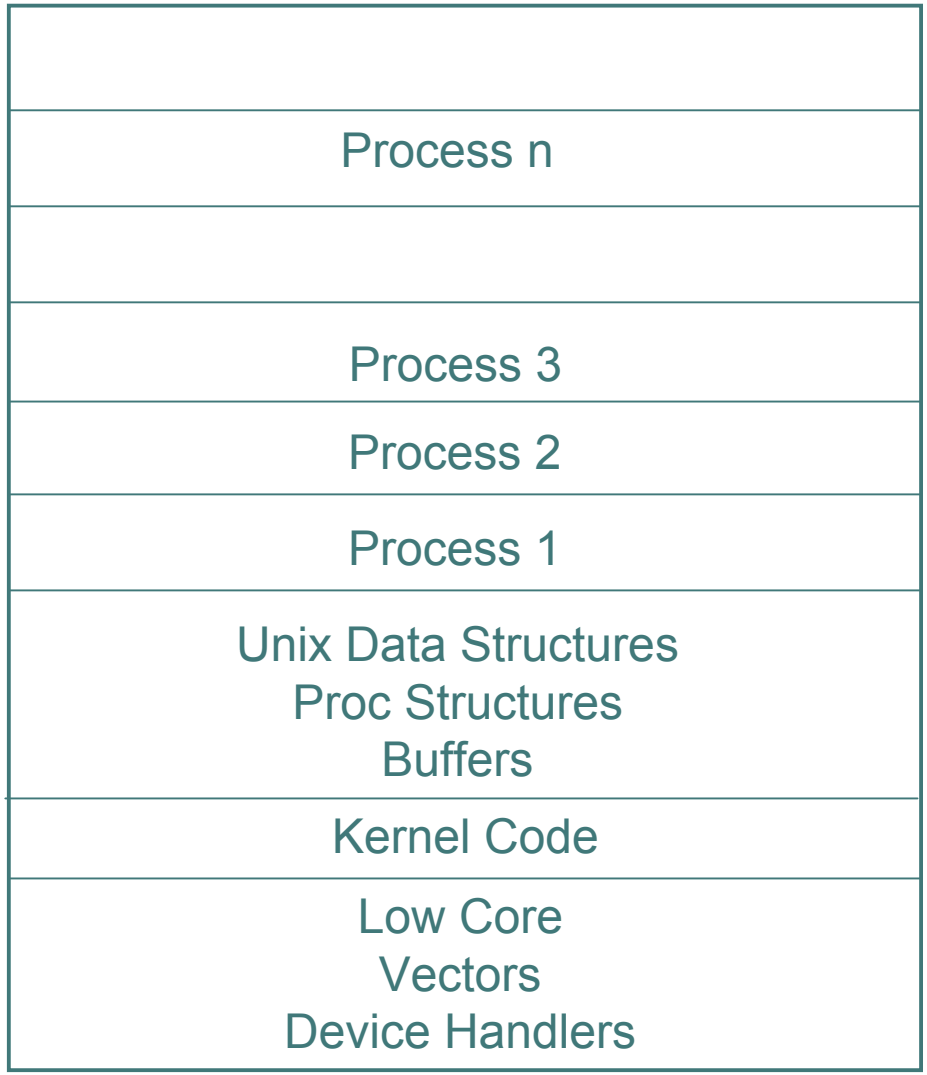
The *fork* System Call

- The *fork* system call creates a child process in the image of the parent process
- The image includes:
  - Shared text (code)
  - Data
  - User stack
  - User structure
  - Kernel stack
- A combination of *fork* and *exec* system calls is used to create a new process and start another program under the new process

Virtual Machine

- Unix implements a collection of *virtual machines* for managing process execution
- These virtual machines are generally similar to the base hardware except some hardware dependent and potentially hazardous instructions are not available
- The virtual machines share a number of hardware resources
  - Memory
  - Disk drives
  - I/O ports
  - The CPU

Unix System Memory Map



Process Table

- Process table entries are defined in /usr/include.sys/proc.h
- The *proc structure*, of which there is one per process, is used by the kernel to determine:
  - Priorities
  - Scheduling states
  - Required resources for a process to run

Proc Structure Components

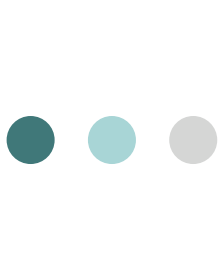
- Process state (sleeping, running, ready-to-run)
- Process flags (in-core, being swapped out, cannot be swapped out)
- Process priority and priority adjustments (nice)
- Scheduling parameters
- Pending signals
- Name of highest level process in group hierarchy and the parent ID
- Address and size of swappable image
- Pointer to user structure
- Pointer to linked list of running processes

Process Events

- There are two process events that can affect a process
  - Interrupts
  - Signals

Interrupts

- Interrupts are asynchronous events that are generally caused by a hardware condition
  - May be an indication that something needs attention or that something is now available or ready
    - A printer is ready for more data or a disk drive has the requested block available
- There are two types of interrupts handled by the kernel
  - Device interrupts
  - Hardware traps



Interrupts

- Hardware traps are usually a result of some kind of CPU error condition
  - Invalid bus access
  - Divide by zero
- Device interrupts often indicate the completion of an I/O request
- Interrupt service often results in a process switch

Signals

- *Signals* are a software mechanism that are similar to a message of some sort
- They can be trapped and handled or ignored
- Signals operate through two different system calls
  - The *kill* system call
  - The *signal* system call

kill() System Call

- The *kill* system call sends a signal to a process
- *kill* is generally used to terminate a process
- It requires the PID of the process to be terminated and the signal number to be sent as arguments

```
#include <sys/types.h>
#include <signal.h>

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

The Signal System Call

- The *signal* system call is much more diverse
- When a signal occurs, the kernel checks to see if the user had executed a signal system call and was therefore expecting a signal
  - If the call was to ignore the signal, the kernel returns
  - Otherwise, it checks to see if it was a *trap* or *kill* signal
  - If not, it processes the signal
  - If it was a *trap* or *kill* signal, the kernel checks to see if core should be dumped and then calls the exit routine to terminate the user process

signal() system call

- The signal() system call installs a new signal handler for the signal

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum,
  sighandler_t handler);
```

Common Unix Signals

- SIGHUP Hang-up
- SIGINT Interrupt (Ctrl-C)
- SIGQIT Quit
- SIGINS Illegal Instruction
- SIGTRAP Trace Trap
- SIGKILL Kill
- SIGSYS Bad argument to system call
- SIGPIPE Write on pipe with no one to read it
- SIGTERM Software termination signal from kill
- SIGSTOP Stop signal
- See /usr/include/sys/signal.h

Signal Acceptance

- There are several possible actions to take when a signal occurs
  - Ignore it (not SIGKILL nor SIGSTOP)
  - Catch it
  - Let the default action apply
- The superuser can send signals to any process
- Normal users can only send signals to their own processes

raise() function

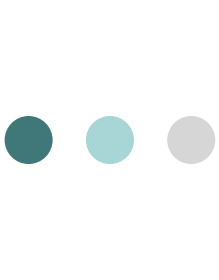
- The raise() function sends a signal to the current process.

```
#include <signal.h>
  int raise(int sig);
```

- It is equivalent to:
  `kill(getpid(),sig);`

Process Termination

- A process is terminated by executing an *exit* system call or as a result of a *kill* signal
- When a process executes an exit system call, it is first placed in a *zombie* state
- In this state, it doesn't exist anymore but may leave timing information and exit status for its parent process
- A zombie process is removed by executing a *wait* system call by the parent process



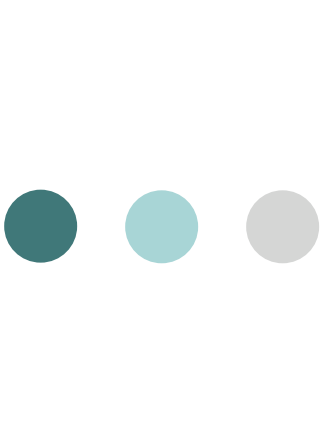
Process Cleanup

- The termination of a process requires a number of cleanup actions
- These actions include:
  - Releasing all memory used by the process
  - Reducing reference counts for all files used by the process
  - Closing any files that have reference counts of zero
  - Releasing shared text areas, if any
  - Releasing the associated process table entry, the *proc structure*
    - This happens when the parent issues the *wait* system call, which returns the terminated child's PID

Example

- Catch signals
  - SIGINT
  - SIGTSTP
- Can be generated from the keyboard
  - Crtl-C
  - Ctrl-Z
- SIGTSTP ≠ SIGSTOP

```c
#include <signal.h>

static void sig_handler(int signum)
{
    if (signum == SIGINT)
        printf("received SIGINT\n");
    else if (signum == SIGTSTP)
        {printf("received SIGTSTP\n"); exit(0);}
    else
        printf("ERROR: received signal %d\n",
   signum);
    return;
}

int main(void)
{
    if (signal(SIGINT, sig_handler) == SIG_ERR)
        printf("ERROR: can't catch SIGINT");
    if (signal(SIGTSTP, sig_handler) == SIG_ERR)
        printf("ERROR: can't catch SIGINT");

    for(;;) pause();
}
```



File IO

Writing and reading data from files

POSIX and C

- There are functions for file IO both in the
  - Standard C library
    - printf(), scanf(), putc(), getc()
  - POSIX API
    - creat(), write(), read()
- Both can be used to handle files

Some interesting functions

- *creat():* create file
- *open():* open file
- *close():* close file
- *read():* read data
- *write():* write data
- *lseek():* reposition head

open(), creat() - open a file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
```

DESCRIPTION

- The **open()** system call is used to convert a pathname into a file descriptor (a small, non-negative integer for use in subsequent I/O as with read, write, etc.).
- This call creates a new open file with the minimum available number.

close() - close a file descriptor

SYNOPSIS

```
#include <unistd.h>

int close(int fd);
```

DESCRIPTION

- Closes a file descriptor, so that it no longer refers to any file and may be reused.

write() - write to a file

SYNOPSIS

```
include <unistd.h>

ssize_t write(int fd, const void *buf,
size_t count);
```

DESCRIPTION

- **write()** writes up to **count** bytes to the file referenced by the file descriptor **fd** from the buffer starting at **buf**.
- It returns the number of bytes written

read() - read from a file

SYNOPSIS

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

DESCRIPTION

- **read()** attempts to read up to **count** bytes from file descriptor **fd** into the buffer starting at **buf**.
- If **count** is zero, read() returns zero and has no other results.
- POSIX requires that a **read()** which can be proved to occur after a **write()** has returned returns the new data.
- Note that not all file systems are POSIX conforming.

lseek() - reposition file offset

SYNOPSIS

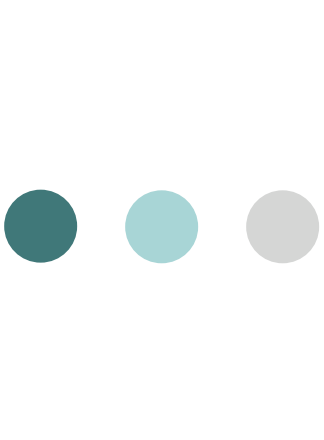
```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int fildes, off_t offset, int whence);
```

DESCRIPTION

- The lseek() function repositions the offset of the file descriptor fildes to the argument offset according to the directive whence as follows:
  - SEEK_SET: The offset is set to offset bytes.
  - SEEK_CUR: The offset is set to its current location plus offset bytes.
  - SEEK_END: The offset is set to the size of the file plus offset bytes.

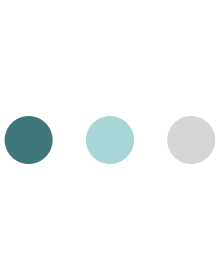
Example

```
#include    <sys/stat.h>
#include    <fcntl.h>
#include    <unistd.h>

/* default file access permissions for new file */
#define FILE_MODE   (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

char    buf[] = "abcdefghij";

int main(void)
{
    int     fd;

    if ( (fd = creat("file.test", FILE_MODE)) < 0)
        {printf("creat error");exit(1);}

    if (write(fd, buf1, 10) != 10)
        {printf("buf1 write error");exit(2);}

   if (close(fd) != 0)
        {printf("close error");exit(3);}

   exit(0);
}
```



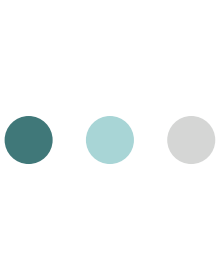
Pipes

Inter-process Communication



IPC

- How does one process communicate with another process?
  - semaphores -- signal notifies waiting process
  - software interrupts -- process notified asynchronously
  - message passing -- processes send and receive messages.
  - **pipes** -- unidirectional stream communication

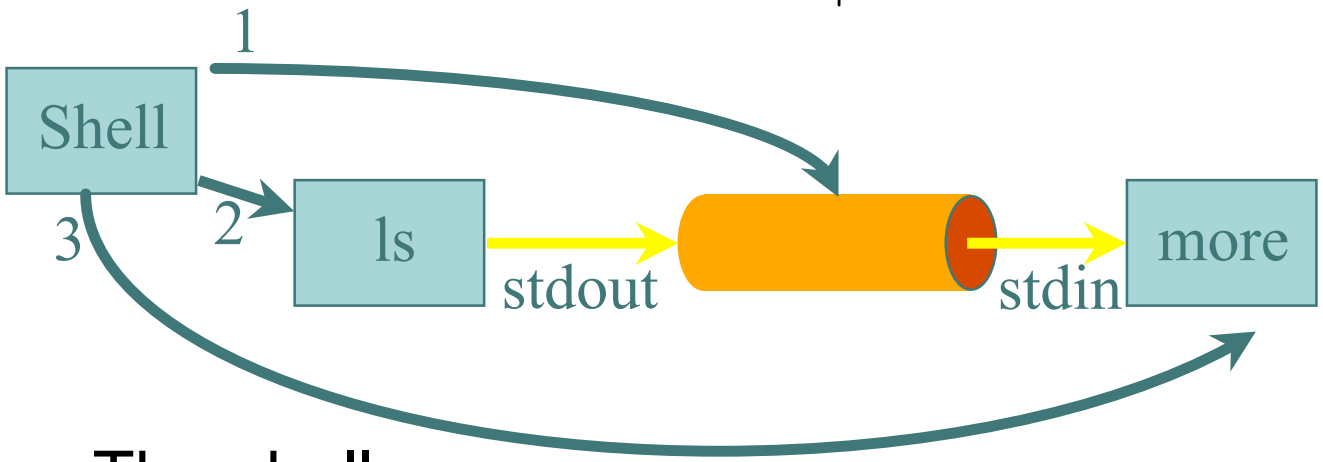


Pipes

- Pipes are a way to allow processes to communicate with each other
- Pipes are uni-directional
  - They can only transfer data in one direction
  - If you want two processes to have a two-way conversation, you must use two pipes

Shell Pipes

- Shell command: `% ls | more`



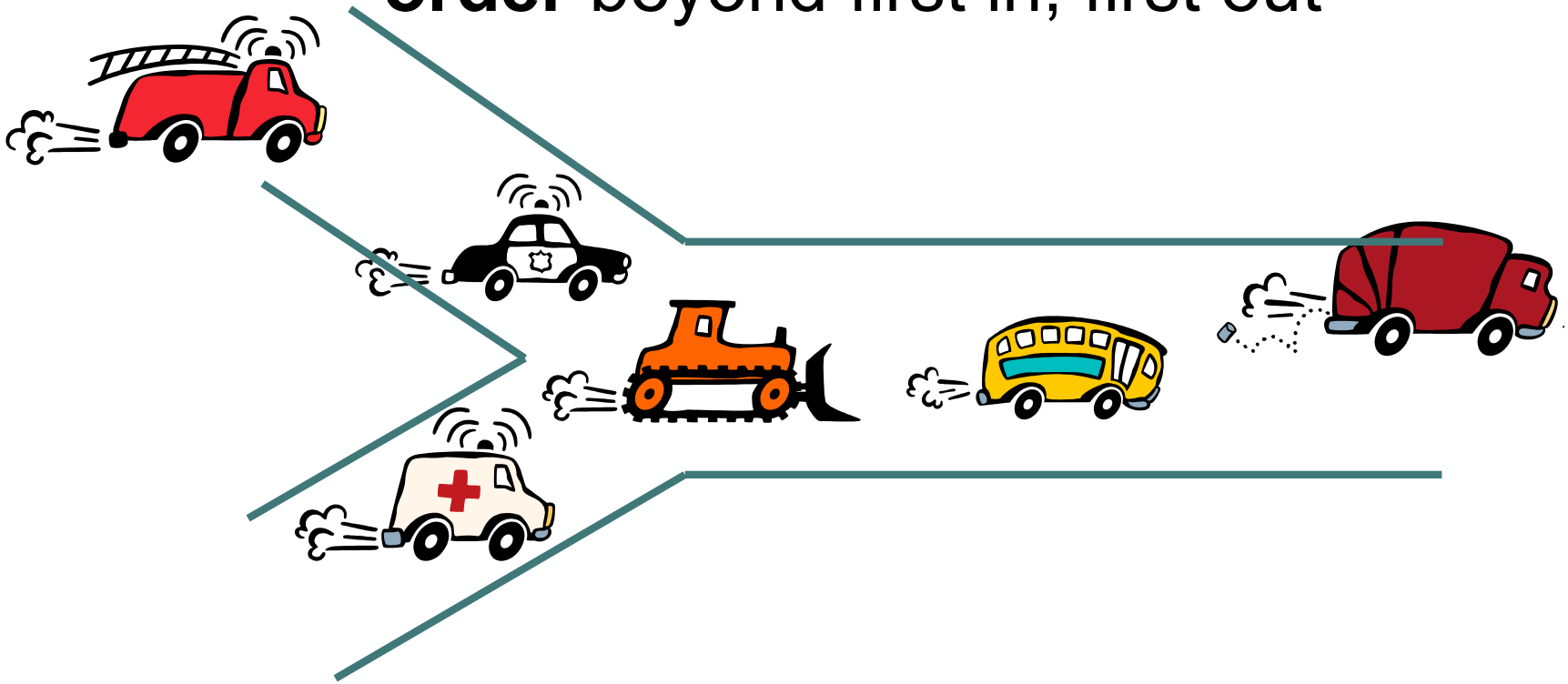
- The shell:
  - creates a pipe
  - creates a process for **ls** command, setting **stdout** to write side of pipe
  - creates a process for **more** command, setting **stdin** to read side of pipe

Pipes Implement a FIFO

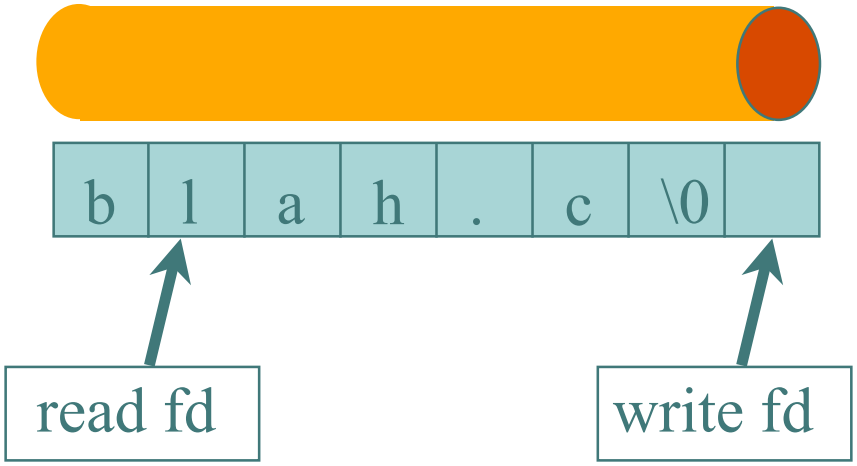
- A FIFO (First In, First Out) buffer is like a queue or a line at a movie theater
- Elements are added at one end of the queue and exit the other end in the same order
- There is no way for any individual element to move ahead of another

Multiple Inputs

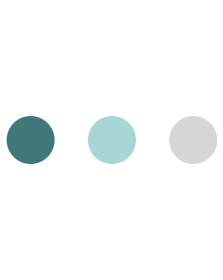
- It is possible to have multiple feeders of a pipe but there is **no guarantee of order** beyond first in, first out



The Pipe



- Bounded Buffer
  - shared buffer (Unix 4096K)
  - blocks writes to full pipe
  - blocks reads to empty pipe



The Pipe

- Process inherits file descriptors from parent
  - file descriptor 0 stdin, 1 stdout, 2 stderr
- Process doesn't know when reading from keyboard, file, or process or writing to terminal, file, or process

pipe() System call

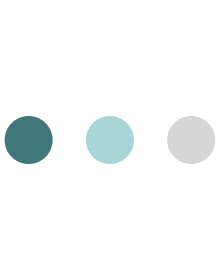
- Pipe creation:
  - `pipe(fds)` creates a pipe
  - `fds` is an array of 2 file descriptors
  - Read from `fds[0]`, write to `fds[1]`
- Read:

```
read(fds[0], buffer, nbytes)
```

- Write:

```
write(fds[1], buffer, nbytes)
```

```
int     n, fd[2];
pid_t   pid;
char    line[MAXLINE];

if (pipe(fd) < 0)
        {printf("pipe() error\n");exit(1);}

if ( (pid = fork()) < 0)
        {printf("fork() error\n");exit(2);}
else if (pid > 0) {     /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
} else {                /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
}

write(STDOUT_FILENO, "Bye.\n", 5);
```



Exercise

- Use a pipe to make a child know the PID of the father to kill() it a SIGHUP
- Use
  - pipe()
  - fork()
  - getpid()
  - write()
  - read()
  - kill()