2.3. Process Management

PROFESSORS:

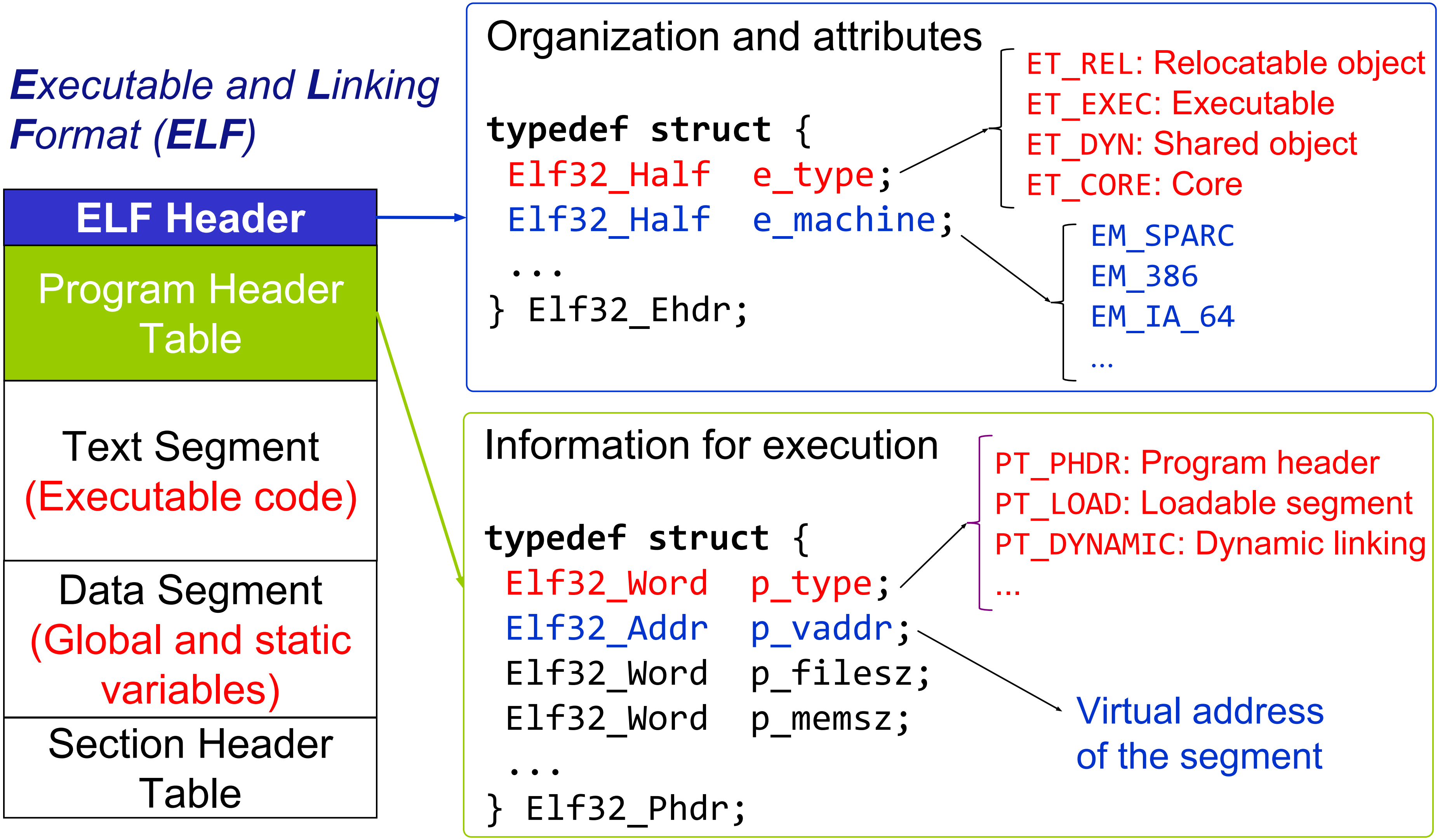
Eduardo Huedo Cuesta
Rubén Santiago Montero

OTHER AUTHORS:

Ignacio Martín Llorente
Juan Carlos Fabero Jiménez

Structure of a Program

- Set of machine instructions and data, stored in a executable image on disk
- A program is a passive entity



***Executable** and **Linking Format** (**ELF**)*

Organization and attributes

```
typedef struct {
 Elf32_Half  e_type;
 Elf32_Half  e_machine;
 ...
} Elf32_Ehdr;
```

ET_REL: Relocatable object
ET_EXEC: Executable
ET_DYN: Shared object
ET_CORE: Core

EM_SPARC
EM_386
EM_IA_64
...

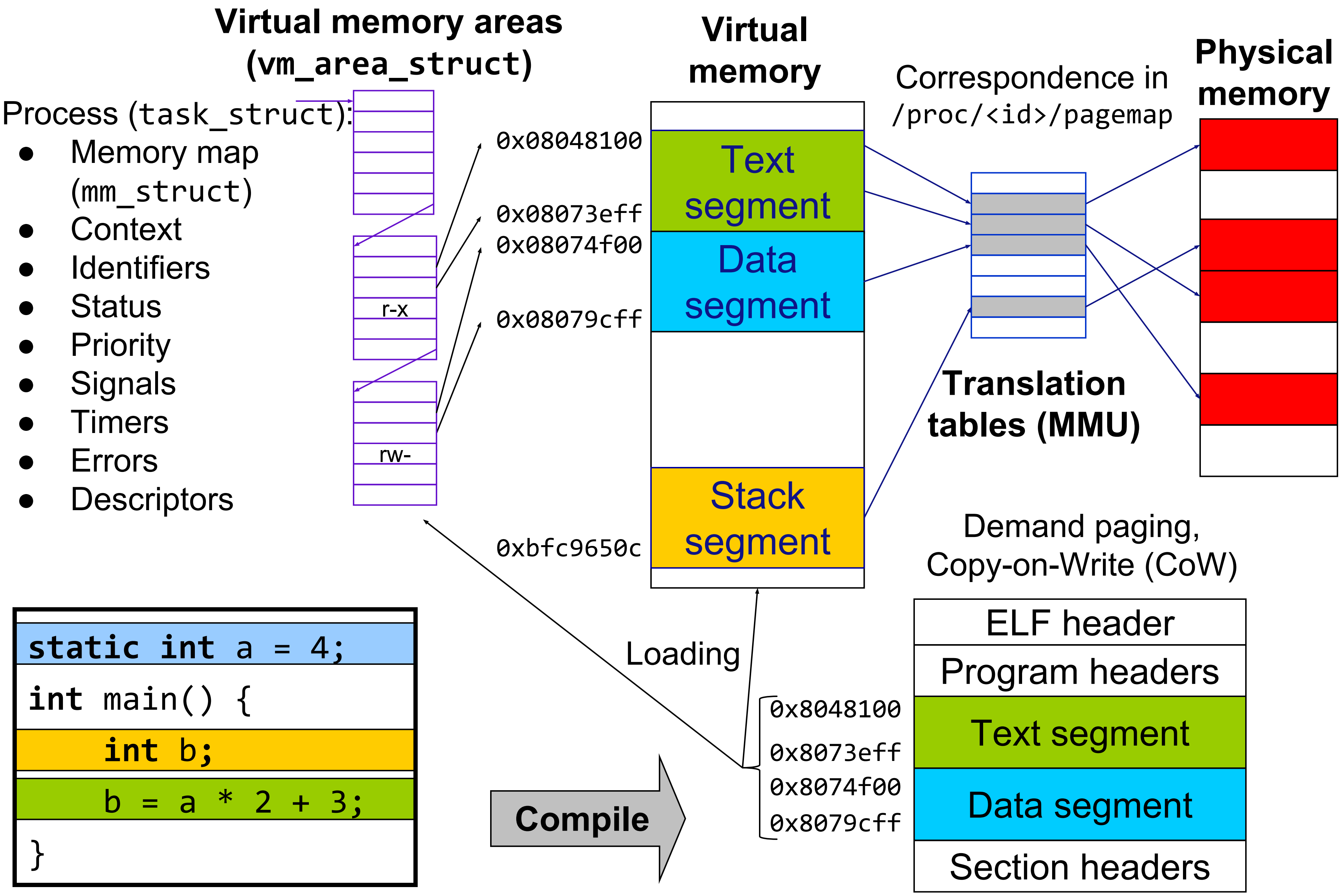
Information for execution

```
typedef struct {
 Elf32_Word  p_type;
 Elf32_Addr  p_vaddr;
 Elf32_Word  p_filesz;
 Elf32_Word  p_memsz;
 ...
} Elf32_Phdr;
```

PT_PHDR: Program header
PT_LOAD: Loadable segment
PT_DYNAMIC: Dynamic linking
...

Virtual address of the segment

Structure of a Process



Process (task_struct):

- Memory map (mm_struct)
- Context
- Identifiers
- Status
- Priority
- Signals
- Timers
- Errors
- Descriptors

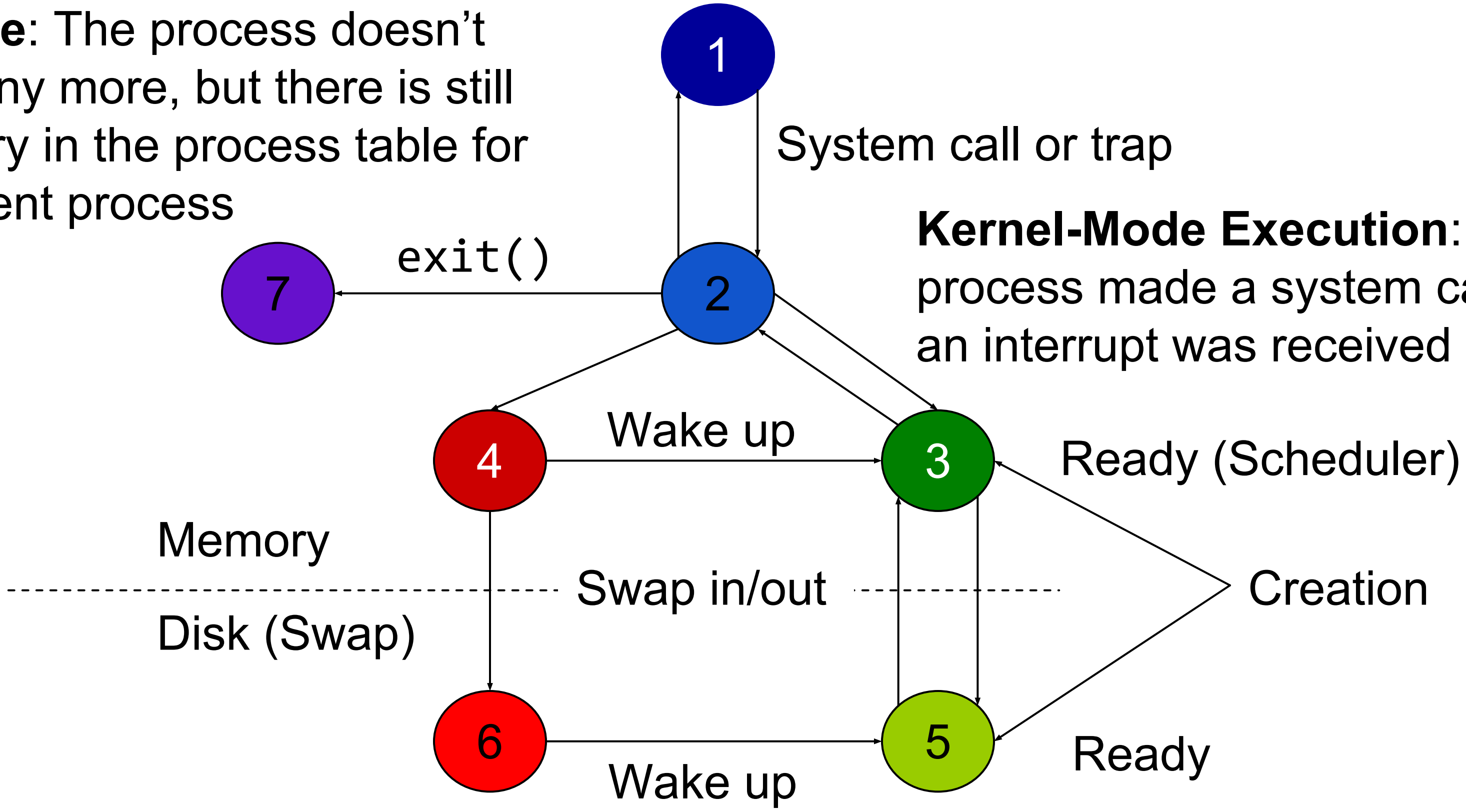
```
static int a = 4;
int main() {
    int b;
    b = a * 2 + 3;
}
```

States of a Process

User-Mode Execution: The process is active and executing user code

Zombie: The process doesn’t exist any more, but there is still an entry in the process table for its parent process

Kernel-Mode Execution: The process made a system call or an interrupt was received

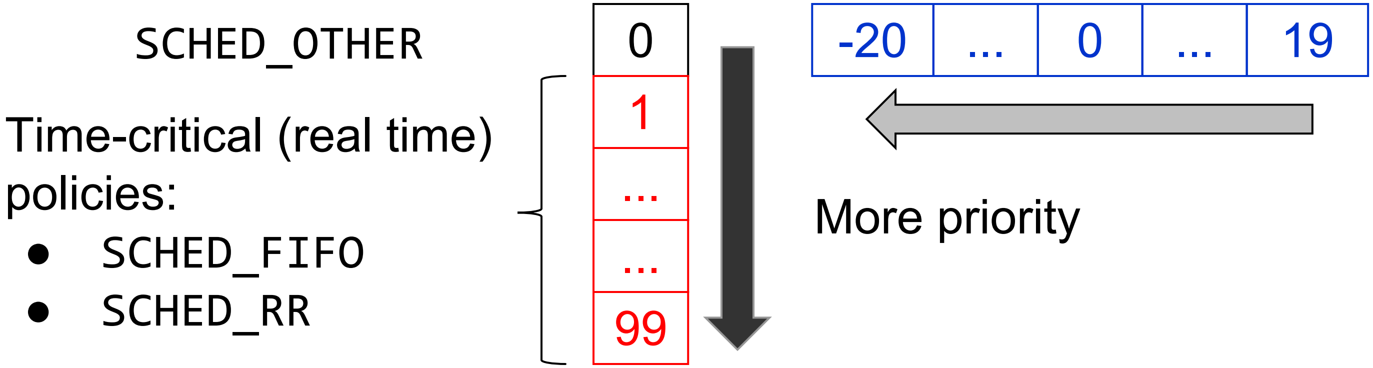


Waiting: The process is waiting for an event (e.g. a I/O operation to complete) or resource

Ready: The process is ready to run, waiting for a free CPU

Scheduler

- The scheduler is the kernel component that decides which runnable task will be executed by the CPU next, based on the task priority and the scheduling policy
- Scheduling is preemptive and the scheduling policy only determines the ordering within the list of runnable processes with equal priority
- **Scheduling policies** (see `/usr/include/bits/sched.h`):
  - `SCHED_OTHER`: Standard round-robin time-sharing policy with priority 0. It also considers the *nice* value (usually, from -20 to 19 and 0 as default)
  - `SCHED_FIFO`: “Real-time” first-in, first-out policy with priorities in the range 1 to 99. A task of this policy will always preempt any lower priority task, and will run until it is blocked by an I/O request, it is preempted by a higher priority task or it yields the CPU
  - `SCHED_RR`: As the previous policy, except that each task is allowed to run only for a maximum time quantum in a round-robin fashion



`SCHED_OTHER`

Time-critical (real time) policies:

- `SCHED_FIFO`
- `SCHED_RR`

0
1
...
...
99

-20	...	0	...	19

More priority

Scheduler

`<sched.h>`

POSIX

- Set scheduling policy and parameters:

```
int sched_setscheduler(pid_t pid, int policy,
      const struct sched_param *p);
```

  - `pid` is a PID (a value of 0 refers to the current process)
  - `policy` selects the scheduling policy
  - p is used to set the new priority:

```
struct sched_param {
    int sched_priority;
    ...
};
```

- Get scheduling policy

```
int sched_getscheduler(pid_t pid);
```

- These calls actually affect the thread, as the scheduler manages threads (or tasks)
  - In multithreaded programs, calls have a `pthread` equivalent
- Scheduling attributes are inherited through `fork()` calls

Scheduler

`<sched.h>`

POSIX

- Set and get scheduling parameters:

```
int sched_setparam(pid_t pid, const struct sched_param *p);
int sched_getparam(pid_t pid, struct sched_param *p);
```

  - `pid` is a PID (a value of 0 refers to the current process)
  - p is used to set the new priority or to get the current one

- Get priority range:

```
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

  - `policy` selects the scheduling policy

- Error codes:
  - `EINVAL`: Invalid arguments: `pid` is negative, `param` is NULL, `policy` is not one of the recognized policies or `p` does not make sense for the specified policy
  - `EPERM`: The calling thread does not have appropriate privileges, the UID doesn't match, or non-root user trying to set "real time" policies
  - `ESRCH`: The thread whose ID is `pid` could not be found

- `chrt` (change real-time) command provides access to these calls

Scheduler

`<sys/resource.h>`
SV+BSD

- Get and set the *nice* value:

```
int getpriority(int which, int who);
int setpriority(int which, int who, int prio);
```

  - `which` is one of `PRIO_PROCESS`, `PRIO_PGRP`, or `PRIO_USER`
  - `who` is a PID for `PRIO_PROCESS`, a PGID for `PRIO_PGRP`, or a UID for `PRIO_USER`
    - 0 denotes the calling process, the process group of the calling process or the real UID of the calling process, respectively
  - `prio` is the new *nice* value in the range -20 (highest) to 19 (lowest)
- Error codes:
  - `ESRCH`: No process was located using the `which` and `who` values specified
  - `EINVAL`: Invalid value for `which`
  - `EPERM`: A process was located, but its effective user ID did not match either the effective or the real user ID of the caller, and was not privileged
  - `EACCES`: The caller attempted to lower a process priority, but did not have the required privilege
- `nice` and `renice` commands provide access to these calls (`ps` to get the *nice* value)

Process Attributes

`<unistd.h>`
SV+BSD+POSIX

- Get process identifiers:

```
pid_t getpid(void);
pid_t getppid(void);
```

  - Each process has a unique PID (Process ID) and registers which process created it in its PPID (Parent PID)

- Set and get process group:

```
int setpgid(pid_t pid, pid_t pgid);
pid_t getpgid(pid_t pid);
```

  - Processes belong to a process group, which is mainly used for signal distribution
    - The PGID (Process Group ID) is equal to the PID of the process group leader

- Error codes:
  - `EINVAL`: pgid < 0
  - `EPERM`: Permission violation
  - `ESRCH`: The process does not exist

Process Attributes

`<unistd.h>`
SV+POSIX

- Set or get session IDs:

```
pid_t setsid(void);
pid_t getsid(pid_t pid);
```

  - Process groups can be grouped in sessions, with a SID (Session ID) equal to the PID of the session leader
  - Sessions are used to manage access to the system, for example:
    - The `login` process creates a session
    - All user processes and process groups belong to that session
    - On disconnection, all processes of the session receive the `SIGHUP` signal
  - A process can create a new session if it is not a process group leader
    - A new process group is also created, with only this process
    - The calling process will become the leader of the new session and the new process group, so its PGID and SID are set to its PID
- Command `setsid` runs a program in a new session

Process Resources: Limits

`<unistd.h>`
`<sys/types.h>`
`<sys/resource.h>`

SV+BSD

- Get and set resource limits to processes:

```
int getrlimit(int resource, struct rlimit *rlim);
int setrlimit(int resource, const struct rlimit *rlim);
```

  - `resource` can be:
    - `RLIMIT_CPU`: Maximum CPU time (seconds)
    - `RLIMIT_FSIZE`: Maximum file size (bytes)
    - `RLIMIT_DATA`: Maximum heap size (bytes)
    - `RLIMIT_STACK`: Maximum stack size (bytes)
    - `RLIMIT_CORE`: Maximum core file size (bytes)
    - `RLIMIT_NPROC`: Maximum number of processes
    - `RLIMIT_NOFILE`: Maximum number of open file descriptors
  - `rlim` specifies the limit

```
struct rlimit {
    int rlim_cur; /* Soft limit */
    int rlim_max; /* Hard limit (ceiling for rlim_cur) */
};
```

  - The value `RLIM_INFINITY` denotes no limit

- The shell builtin command `ulimit` provides access to this functionality

Process Resources: Usage

<sys/time.h>
<sys/resource.h>

SV+BSD+POSIX

- Get resource usage:

```
int getrusage(int who, struct rusage *usage);
```

  - who can be:
    - RUSAGE_CHILDREN: only by its children
    - RUSAGE_SELF: by the process and its children
  - usage will contain the resource usage

```
struct rusage {
    struct timeval ru_utime; /* user CPU time used */
    struct timeval ru_stime; /* system CPU time used */
    long ru_maxrss;      /* maximum resident set size */
    ... /* see /usr/include/bits/resource.h */
}
```

- Program time -v provides resource usage information
  - Since time is also a shell keyword, use /usr/bin/time or command time

Process Resources: Working Directory

<unistd.h>
POSIX

- Get current working directory:

```
char *getcwd(char *buf, size_t size);
```

  - The working directory is used for all relative paths in the process
  - This function returns a null-terminated string containing an absolute pathname that is the current working directory of the calling process
  - The pathname is also returned in `buf`, if present
  - If the length of the pathname, including the terminating NULL byte, exceeds `size` bytes, NULL is returned, and `errno` is set to `ERANGE`

- Change working directory:

```
int chdir(const char *path);
```

Process Resources: Environment

`<stdlib.h>`

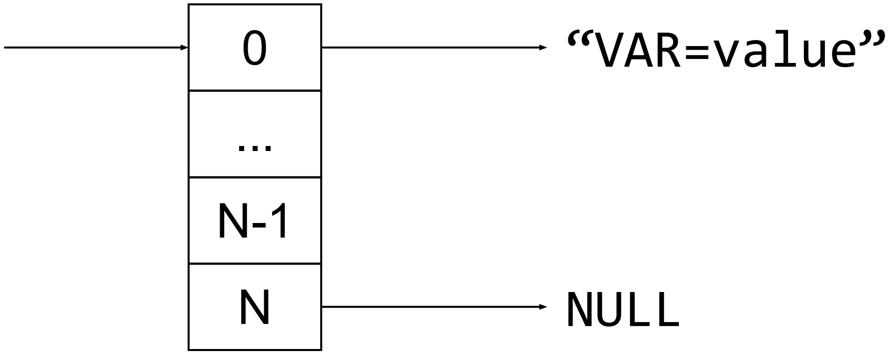
SV+BSD+POSIX

- Get and set environment variables:

```
char *getenv(const char *name);
int setenv(const char *name, const char *value, int overwrite);
int unsetenv(const char *name);
```

  - Processes are executed in a given environment, which is in general inherited from the parent process
    - Some applications, e.g. `sudo` or the shell, limit or control the environment passed to the processes or the way it is initialized
  - The environment is a set of strings in the form `VARIABLE=value`
    - By convention, variable names are capitalized

```
extern char **environ;
```



HOME, USER, PATH, PWD, SHELL…

Process Creation: Program Execution

`<stdlib.h>`
ANSI C+POSIX

- Execute a shell command:

```
int system(const char *command);
```

  - Executes the command specified in `command` by calling `/bin/sh -c`
  - The call returns after the command has been completed (unless it is executed in background)
  - If there is no error in the call, it returns the termination status of the program
  - If `command` is NULL, it returns nonzero if the shell is available, and zero if not

Process Creation: Program Execution

`<unistd.h>`
POSIX

- Execute a file:

```
int execve(const char *path, char *const argv[],
        char *const envp[]);
```

- The following library functions are front-ends for the `execve(2)` system call:

```
int execl(const char *path, const char *a0, ...);
int execlp(const char *path, const char *a0, ...);
int execle(const char *path, const char *a0, ...,
        char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *path, char *const argv[]);
```

  - The first argument must be the program and the last element must be NULL

- The `exec()` family of functions, executes a program in the current process, replacing its image with a new process image

	Absolute path	Using PATH	Set env.
Argument list	`execl()`	`execlp()`	`execle()`
Argument vector	`execv()`	`execvp()`	`execve()`

Process Creation: Fork

`<unistd.h>`

SV+POSIX+BSD

- Create a child process:

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

  - This function returns:
    - -1: Failure (no child process is created, and `errno` is set appropriately)
    - 0: In the child process
    - >0: In the parent process, the value is the child’s PID
  - Errors:
    - `EAGAIN`: The caller's `RLIMIT_NPROC` resource limit was encountered
    - `ENOMEM`: Not enough memory

- After creation, the new process executes the same code as the parent process

- Both processes have the same user context:
  - The child has its own unique PID
  - The child inherits copies of the parent's set of open file descriptors
  - The child's set of pending signals is initially empty
  - The child does not inherit record locks from its parent
  - The child does not inherit timers from its parent

Process Creation: Fork

```
int main() {
  pid_t mypid, pid;

  pid = fork();
  mypid = getpid();

  switch (pid) {
    case -1:
      perror("fork");
      exit(-1);

    case 0:
      printf("New process %i (parent: %i)\n", mypid, getppid());
      break;

    default:
      printf("Parent process %i (new: %i)\n", mypid, pid);
      break;
  }

  return 0;
}
```

Process Termination

`<unistd.h>`
SV+POSIX+BSD

- Cause normal process termination:

```
void _exit(int status);
```

  - `status` is the exit status, which is a number below 255
    - By convention, 0 means success and 1 means error (don't return `errno` or -1)
    - 8 bits accessible in the shell via `$?` or in the parent process via `wait()`
- A process can finalize because:
  - It voluntarily calls `exit` (or `return` from `main()`)
  - It receives a signal (multiple causes)

Process Termination

sys/types.h sys/wait.h
SV+POSIX+BSD

- Wait for process termination (or some state change):

```
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

info about exit status (8 bits)	exit status (8 bits)

- `pid` can be:
  - > 0: wait for the child whose PID is equal to the value of `pid`
  - 0: wait for any child process whose PGID is equal to that of the calling process
  - -1: wait for any child process (same as `wait()`)
  - < -1: wait for any child process whose PGID is equal to `-pid`
- `options` is an OR of zero or more of the following constants:
  - `WNOHANG`: return immediately if no child has exited
  - `WUNTRACED`: return if a child has stopped (but not traced via `ptrace(2)`)
  - `WCONTINUED`: return if a stopped child has been resumed with `SIGCONT`
- Some useful macros to check `status`:
  - `WIFEXITED(status)`: the child terminated normally, calling `exit()`
  - `WEXITSTATUS(status)`: returns the exit status (least significant 8 bits)
  - `WIFSIGNALED(status)`: the child process was terminated by a signal
  - `WTERMSIG(status)`: number of the signal that caused the process to terminate

Inter-Process Communication. Signals and Pipes

IPC Mechanisms

- Synchronization between processes:
    - Process/threads executed on the same system:
        - **Signals**
        - **File locks**
        - Mutex and condition variables (for threads within a process)
        - Semaphores (System V IPC)
        - Message queues (System V IPC)
    - Processes executed on different systems
        - Based on **sockets** (message passing, message queues…)
- Data sharing between processes:
    - Processes executed on the same system
        - **Unnamed pipes** (pipes)
        - **Named pipes** (FIFOs)
        - Shared memory (System V IPC)
        - Message queues (System V IPC)
        - Based on **files**
    - Processes executed on different systems
        - Based on **sockets**

Signals

- A signal is an asynchronous notification sent to a process in order to notify it of an event that occurred
- Signals are generated by processes or by the kernel
- A process can elect one of the following behaviors (signal dispositions) to occur on delivery of the signal:
  - Ignore the signal
  - Block the signal
  - Perform the default action
  - Catch the signal with a signal handler, which is a programmer-defined function that is automatically invoked when the signal is delivered
- Signal types:
  - Process termination
  - Exceptions
  - System calls
  - Generated by process
  - Terminal control
  - Process trace
- Heavily dependent on the system (check `signal.h`)

Examples

- `SIGHUP`: Hangup detected on controlling terminal or death of controlling process (sent to all processes in a session) → Terminate
- `SIGINT`: Interrupt from keyboard (Ctrl+C) → Terminate
- `SIGQUIT`: Quit from keyboard (Ctrl+\) → Core dump
- `SIGILL`: Illegal instruction (erroneous function pointers) → Core dump
- `SIGTRAP`: Trace/breakpoint trap → Core dump
- `SIGKILL` (9): Kill signal (can’t be ignored) → Terminate
- `SIGBUS`: Bus error (bad alignment or invalid address) → Core dump
- `SIGSEGV`: Invalid memory reference → Core dump
- `SIGPIPE`: Broken pipe: write to pipe with no readers → Terminate
- `SIGALRM`: Timer signal from `alarm(2)` → Terminate
- `SIGTERM`: Termination signal (can be captured) → Terminate
- `SIGUSR1`: User-defined signal 1 → Terminate
- `SIGUSR2`: User-defined signal 2 → Terminate
- `SIGCHLD`: Child stopped or terminated → Ignore
- `SIGCONT`: Continue if stopped → Continue
- `SIGSTOP`: Stop process → Stop
- `SIGPROF`: Profiling timer expired → Terminate
- `SIGURG`: Urgent condition on socket (4.2BSD) → Ignore

`signal(7)`

Sending a Signal

<signal.h>
SV+BSD+POSIX

- Send signal to a process:

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

  - `pid` identifies the process(es) receiving the signal:
    - >0: The process with PID equal to `pid`
    - 0: All processes in the process group
    - -1: All processes (from highest to lowest), but `init`
    - <-1: All processes in the process group with PGID equal to `-pid`
  - `signal` is the signal to send (if 0, no signal is sent, but error checking is done)
- The `kill` command provides access to this call
- Equivalent calls:

<signal.h>
ANSI-C

<stdlib.h>
SV+BSD+POSIX

```
int raise(int signal);
int abort(void);
```

  - `raise(signal)` ⇒ `kill(getpid(), signal)`
  - `abort()` ⇒ `kill(getpid(), SIGABRT)`

Example

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

int main() {
    kill(getpid(), SIGABRT);
    return 0;
}
```

```
> ./abort_self
Aborted (core dumped)
```

Signal Sets

`<signal.h>`

POSIX

- POSIX signal set operations:

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(sigset_t *set, int signum);
```

  - These functions allow the manipulation of POSIX signal sets, mainly to define signal masks
  - `sigemptyset()` initializes `set` to empty, excluding all signals
  - `sigfillset()` initializes `set` to full, including all signals
  - `sigaddset()` adds signal `signum` to `set`
  - `sigdelset()` deletes signal `signum` from `set`
  - `sigismember()` tests whether signal `signum` is a member of `set`
- A signal set is an opaque type that depends on the system
  - Usually, implemented as a bitmap (see `/usr/include/bits/sigset.h`)

Signal Mask

`<signal.h>`

POSIX

- Examine and change blocked signals:

```
int sigprocmask(int how, const sigset_t *set,
         sigset_t *oldset);
```

  - The signal mask is the set of signals whose delivery is currently blocked for the caller (e.g. to protect code regions)
  - how defines the behaviour:
    - `SIG_BLOCK`: The signals in `set` are added to the current signal mask
    - `SIG_UNBLOCK`: The signals in `set` are removed from the current signal mask (it is permissible to unblock a signal which is not blocked)
    - `SIG_SETMASK`: The signal mask is set to the argument `set`
  - If `oldset` is non-NULL, the previous value of the signal mask is stored
  - If `set` is NULL, then the signal mask is unchanged

- Examine pending signals:

```
int sigpending(const sigset_t *set);
```

  - Pending signals (i.e. signals which have been raised while blocked) are returned in `set`
  - `sigismember()` can be used to determine the signal and `sigprocmask()`, to handle it

Example

```c
#include <signal.h>

int main() {
   sigset_t blk_set;

   sigemptyset(&blk_set);
   sigaddset(&blk_set, SIGINT);
   sigaddset(&blk_set, SIGQUIT);

   sigprocmask(SIG_BLOCK, &blk_set, NULL);

   /* Protected code */

   sigprocmask(SIG_UNBLOCK, &blk_set, NULL);
}
```

Signal Handling

<signal.h>

POSIX

- Examine and change a signal action:

```
int sigaction(int signum,
              const struct sigaction *act,
              struct sigaction *oldact);
```

  - It is used to change the action taken by a process on receipt of a specific signal
  - signum specifies the signal (except SIGKILL and SIGSTOP)
  - If act is non-NULL, the new action for the signal is installed
  - If oldact is non-NULL, the previous action is saved

```
struct sigaction {
    void     (*sa_handler)(int);
    sigset_t   sa_mask;
    int        sa_flags;
    ...
}
```

Signal Handling

- `sa_handler` specifies the action to be associated with `signum`, and may be:
  - `SIG_DFL` for the default action
  - `SIG_IGN` to ignore this signal
  - A pointer to a signal-handling function:

```
void handler(int signum);
```

- `sa_mask` is a set of signals that should be blocked during execution of the signal handler (plus the signal being handled, unless `SA_NODEFER` is used)
- `sa_flags` modifies the behavior of the signal handling:
  - `SA_RESTART`: For BSD compatibility, make certain system calls restartable across signals (otherwise they terminate with `errno` set to `EINTR`)
  - `SA_RESETHAND`: Restore the signal action to the default upon entry to the signal handler
  - `SA_SIGINFO`: The signal-handling function (`sa_sigaction` instead of `sa_handler`) receives additional arguments

Signal Handling

- The process suspends its execution and calls the handler, restoring the execution in the point where the signal was delivered
- Some precautions need to be taken in the handler:
  - Don’t use `extern` or `static` variables
  - Don’t use non-reentrant functions, like `malloc()`, `free()` or functions from the `stdio` library
  - Save and restore the value of `errno`
- As a general rule, do as little as possible in the handler
  - Usually, just set some flag (declared as `volatile`) and exit
- Take always into account that signals are asynchronous

Signal Handling

<signal.h>

POSIX

- Wait for a signal:

  ```
  int sigsuspend(const sigset_t *mask);
  ```

  - This temporarily replaces the signal mask of the calling process with the mask given by `mask` and then suspends the process until delivery of a signal whose action is to invoke a signal handler or to terminate a process
  - If the signal terminates the process, then the call does not return
  - If the signal is caught, then the call returns after the signal handler returns, and the signal mask is restored to the state before the call
  - It always returns -1, with `errno` set to indicate the error (normally, `EINTR`)

<unistd.h>

POSIX

- Sleep for the specified number of seconds:

  ```
  unsigned int sleep(unsigned int sec);
  ```

  - Suspends execution until `sec` seconds have elapsed or a signal arrives which is not ignored
  - Simpler way to suspend a process

<unistd.h>

POSIX

- Wait for signal:

  ```
  int pause(void);
  ```

Alarms

`<unistd.h>`

SV+BSD+POSIX

- Set an alarm clock for delivery of a signal:

```
unsigned int alarm(unsigned int sec);
```

  - A timer (`ITIMER_REAL`) is used to arrange for a `SIGALRM` signal to be delivered to the calling process in `sec` seconds (if zero, no new alarm is scheduled)
  - In any case, any previously set alarm is canceled
  - It returns the number of seconds remaining until any previously scheduled alarm was due to be delivered, or zero if there was no alarm
  - A handler must be set first
  - Do not mix it with calls to `sleep(3)` or any other function using the same timer, like `setitimer()`
  - Alarms are not inherited by `fork()`, but they are by `exec()`

Alarms

<sys/time.h>
SV+BSD

- Get or set value of an interval timer:

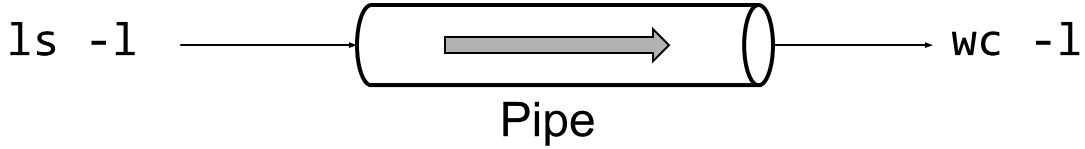
```
int getitimer(int which, struct itimerval *value);
int setitimer(int which, struct itimerval *value,
        struct itimerval *old_value);
```

  - Argument `which` selects the timer:
    - `ITIMER_REAL`: Real (wall clock) time. At each expiration, a `SIGALRM` signal is generated
    - `ITIMER_VIRTUAL`: CPU time consumed by the process in user mode. At each expiration, a `SIGVTALRM` signal is generated
    - `ITIMER_PROF`: CPU time consumed by the process in user and system mode. At each expiration, a `SIGPROF` signal is generated
  - Arguments `value` and `old_value` contain:

```
struct itimerval {
    struct timeval it_interval; /*Interval for periodic timer*/
    struct timeval it_value;    /*Time until next expiration*/
}
```

Unnamed Pipes

- Support for **unidirectional communication** between two processes
- Managed by the system as **files:**
  - Inode
  - File descriptor
  - System and process file table
  - Typical I/O operations
  - Inherited from parent to child processes
- **Synchronization** is done by the **kernel**
- **FIFO** (*first-in-first-out*) access
- The pipe is stored on **main memory**

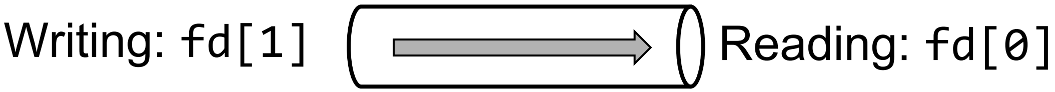


Unnamed Pipes

`<unistd.h>`
SV+BSD+POSIX

- Create a pipe:

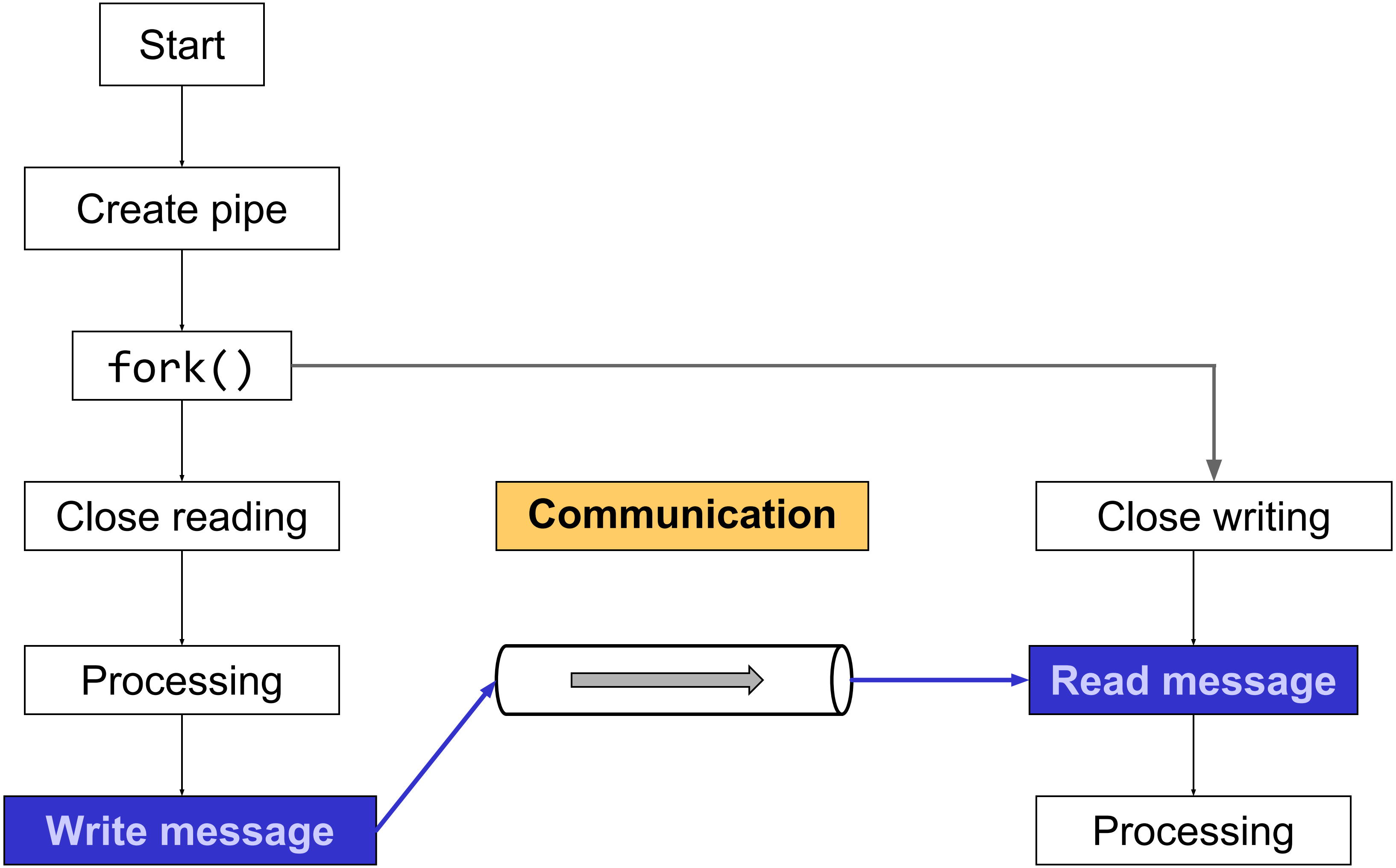
```
int pipe(int fd[2]);
```



Writing: `fd[1]` Reading: `fd[0]`

  - If a process attempts to write to a full pipe, then `write(2)` will block until sufficient data has been read from the pipe to allow the write to complete
  - Errors:
    - `EMFILE`: Too many file descriptors used by the process
    - `ENFILE`: Too many open files in the system
    - `EFAULT`: `fd` is not valid

Unnamed Pipes



Start

Create pipe

fork()

Close reading

Processing

Write message

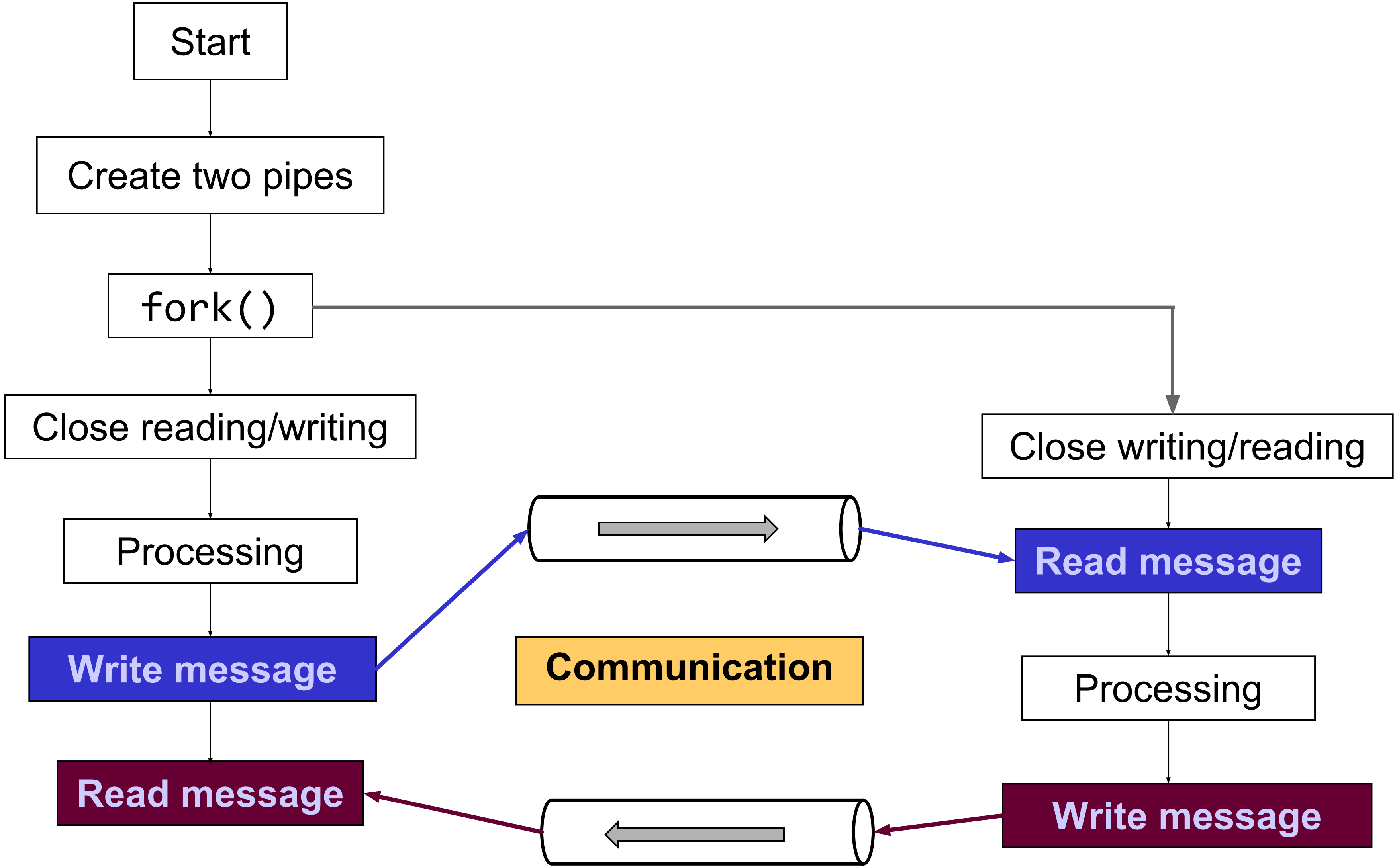
Communication

Close writing

Read message

Processing

Unnamed Pipes



Start

Create two pipes

`fork()`

Close reading/writing

Processing

Write message

Read message

Communication

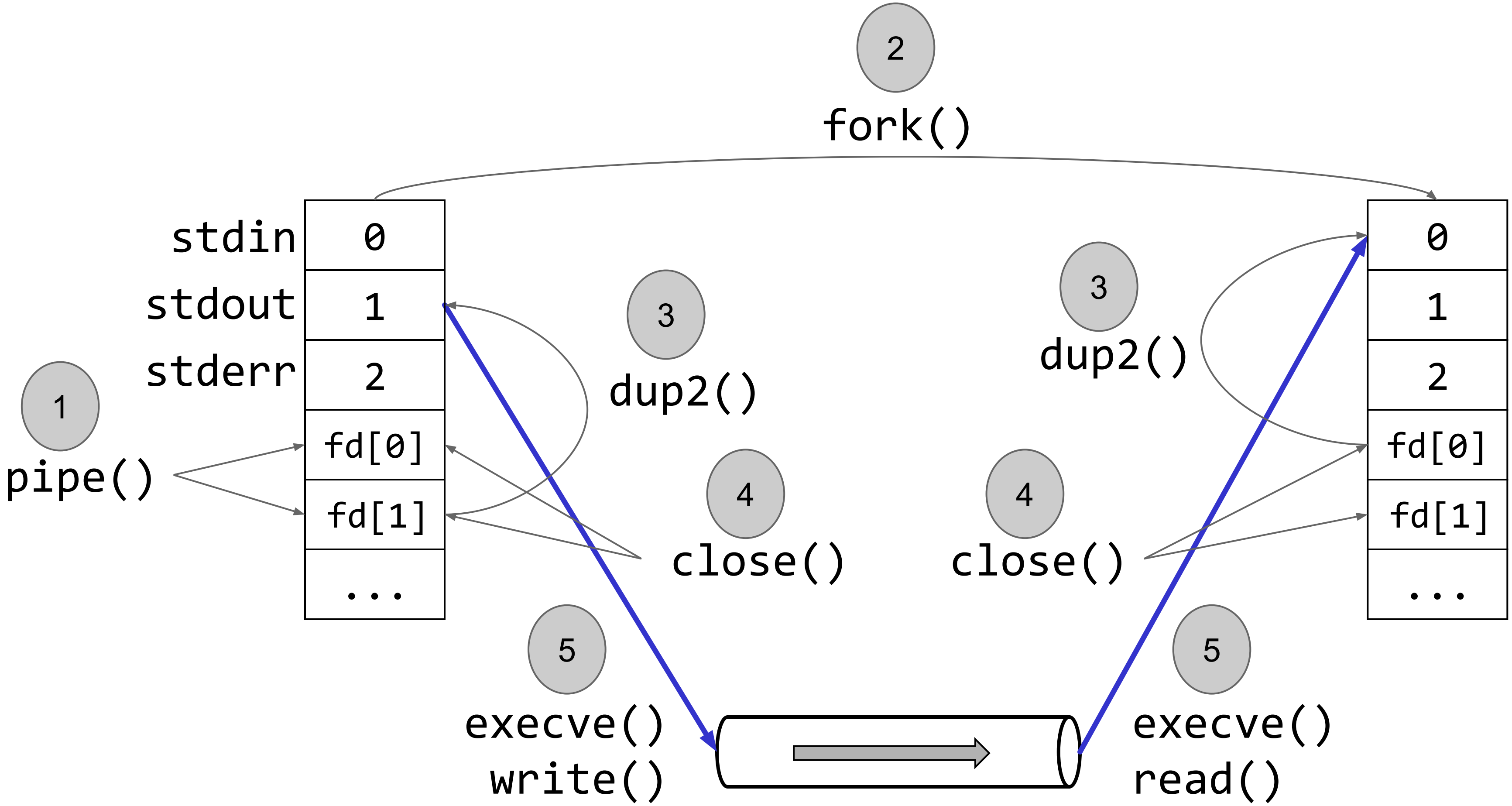
Close writing/reading

Read message

Processing

Write message

Unnamed Pipes



Communication

Named Pipes

- Communication through unnamed pipes can be done only between related processes
- A **named pipe** or a **FIFO special file** is similar to an unnamed pipe, except that it is accessed as part of the file system
  - The file system entry merely serves as a reference point so that processes can access the pipe with `open()` using a name
  - The kernel passes all data internally without writing it to the file system
  - The pipe can be opened by multiple processes for reading or writing, and it must be opened on boths ends (reading and writing) before data can be passed, since opening the pipe blocks until the other end is opened also
    - When opening for reading, this can be modified with the `O_NONBLOCK` flag

I/O Operation	**Condition**	**Result**
Read	Empty pipe, with writer	Blocks
Write	Full pipe, with reader	Blocks
Read	Empty pipe, no writer	Returns 0 (EOF)
Write	No reader	Receives `SIGPIPE`

Named Pipes

```
<sys/types.h>
<sys/stat.h>
<fcntl.h>
<unistd.h>
```

SV+BSD

- Create a special file:

```
int mknod(const char *filename, mode_t mode,
          dev_t dev);
```

  - `filename` is the name of the special file to be created
  - `mode` includes the permissions (modified by process `umask`) and the type of the file to be created, as a logical OR, where type can be:
    - `S_IFREG`: Regular file
    - `S_IFCHR`: Character device (`dev` = major,minor)
    - `S_IFBLK`: Block device (`dev` = major,minor)
    - **`S_IFIFO`**: Named pipe

- The following command provides access to this call:

```
mknod [-m permissions] name type
```

  - `name` is the name of the special file to be created
  - `type` can be:
    - `b`: block device
    - `c`: character device
    - **`p`**: FIFO

Named Pipes

```
<sys/types.h>
<sys/stat.h>
```

POSIX

- Create a named pipe:

```
int mkfifo(const char *filename, mode_t mode);
```

  - `filename` is the name of the pipe to be created
  - `mode` includes permissions of the pipe (modified by process `umask`)

- The following command provides access to this call:

```
mkfifo [-m permissions] name
```

I/O Synchronization

- When a process manages several I/O channels (pipe, socket or terminal), it should not block indefinitely on one of them while others are ready to perform an operation
- Alternatives:
  - Non-blocking I/O: Option `O_NONBLOCK`
    - Instead of blocking, the operation fails with `EAGAIN`
    - It is like I/O polling and needlessly consumes CPU time, since the process never blocks, even when no operation is possible on any file descriptor
  - Signal-driven I/O: Option `O_ASYNC`
    - The process receives a signal (`SIGIO` by default) when the file descriptor becomes ready to perform the operation
    - Dealing with asynchronous signals changes the program logic
  - Asynchronous I/O: Functions `aio_read()`, `aio_write()`...
    - The POSIX AIO interface allows a process to initiate several operations that are performed asynchronously (i.e., in the background)
    - The process can be notified of completion of the operation by delivery of a signal or by instantiation of a thread
    - It is complex, but works on files
  - **Synchronous I/O multiplexing**: Functions `select()`, `poll()` or `epoll()`
    - The process monitors multiple file descriptors and selects those that are ready to perform a specific operation

Sychronous I/O Multiplexing

<sys/select.h>

POSIX+BSD

- Synchronous I/O multiplexing:

```
int select(int nfds, fd_set *readfds,
      fd_set *writefds, fd_set *exceptfds,
      struct timeval *timeout);
```

  - This allows a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become "ready" for some class of I/O operation
  - `nfds` is the highest-numbered file descriptor in any of the three sets, plus 1
  - `readfds` is the set of file descriptors that will be watched to see if a read will not block
  - `writefds` is the set of file descriptors that will be watched to see if a write will not block
  - `exceptfds` is the set of file descriptors that will be watched for exceptions
    - For example, the availability of urgent (out-of-band, OOB) data for reading from a TCP socket
  - `timeout` specifies the interval that `select()` should block waiting for a file descriptor to become ready
    - If both fields of the `timeval` structure are zero, then `select()` returns immediately (useful for polling)
    - If it is NULL (no timeout), `select()` can block indefinitely

Sychronous I/O Multiplexing

<sys/select.h>

POSIX+BSD

- Macros for set manipulation:

```
void FD_ZERO(fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_CLR(int fd, fd_set *set);
int FD_ISSET(int fd, fd_set *set);
```

  - `FD_ZERO` clears a set
  - `FD_SET` adds a file descriptor to a set
  - `FD_CLR` removes a file descriptor from a set
  - `FD_ISSET` tests to see if a file descriptor is part of a set, which is useful after `select()` returns

- `select()` returns the number of changes (i.e. file descriptors contained in the three returned sets) and the sets are modified in place to indicate which file descriptors actually changed status
  - In case of timeout, it returns zero

- On error, the file descriptor sets are unmodified, and `timeout` becomes undefined
  - `EBADF`: An invalid file descriptor was given in one of the sets
  - `EINTR`: A signal was caught
  - `EINVAL`: `nfds` is negative or exceeds the `RLIMIT_NOFILE` resource limit

Sychronous I/O Multiplexing

```
...
fd_set set;

FD_ZERO(&set);
FD_SET(0, &set);

timeout.tv_sec = 2;
timeout.tv_usec = 0;

changes = select(1, &set, NULL, NULL, &timeout);

if (changes == -1)
    perror("select()");
else if (changes) {
    read(0, buffer, 80);
    printf(“New data: %s\n”, buffer);
} else {
    printf("No new data in 2 seg.\n");
}
...
```