Operating Systems (02340123)
Functions Reference - Spring 2025

Razi & Yara

July 26, 2025

Contents

Part I

Functions Reference

Topic 1: Process Management

---

fork

Declaration: `pid_t fork();`
Usage/Explanation: Copies the parent process to the child process and returns with the two processes.

Same code, Same memory, Same environment (files, etc.). But they are separate processes with separate memory spaces and different PIDs.
Parameters: None
Return Values: Returns 0 to child, PID of child to parent, -1 on error.
Additional:

---

wait

Declaration: `pid_t wait(int *wstatus);`
Usage/Explanation: Waits until **any** child process ends, suspending the calling process until a child terminates
Parameters: `wstatus`: Pointer to the variable where the exit status of the child will be stored. If NULL, no status is returned.

To get the value of the status, you can use macros like `WEXITSTATUS(*wstatus)` which return the second byte of the variable, where the exit code of the child is stored.
Return Values: If there are no children or all children have already terminated and waited for, it returns -1. Else waits until a child process ends and returns its PID.
Additional: Can only wait for direct child!

---

waitpid

Declaration: `pid_t waitpid(pid_t pid, int *wstatus, int options);`
Usage/Explanation: Wait until a specific child process ends.
Parameters:

- `pid`: The PID of the child process to wait for. If -1, waits for any child process.

- `wstatus`: Pointer to an integer where the exit status of the child will be stored. If NULL, no status is returned.
- `options`: Options for waiting, such as WNOHANG (do not block if no child has exited). default is 0, which blocks until the child exits.

Return Values: Returns the PID of the child that exited, or -1 on error. If WNOHANG is set and no child has exited, it returns 0.
Additional:

---

exit

Declaration: `void exit(int status);`
Usage/Explanation: Terminates the calling process and releases all of its recourses. The process becomes *zombie* until its parent process requests to check its termination (e.g. wait()) and then clears completely.
Parameters: `status`: The exit status of the process which is returned to the parent process when checked.
Return Values: No return value. The process is terminated immediately, and will never fail.
Additional: The `main()` is not in fact the main function of the process, it is wrapped by `int __libc_start_main()` who collects the return value of `main()` and calls `exit()` with it. This is why we don't use `exit()` in `main()` but rather return from it usually $\implies$ `exit` is always called.

---

execv

Declaration: `int execv(const char *filename, char *const argv[]);`
Usage/Explanation: Replaces the current running process code with a new program. (same PID,PPID but different code and memory).
Parameters:

- `filename`: The path to the file containing the program to execute.
- `argv`: An array of pointers to null-terminated strings containing the parameters to pass to the new program. The first element is the name of the process, i.e. `argv[0]=filename`. The last argument must be `NULL` to indicate the end of the array.

Return Values: Returns -1 on error, and does not return on success as the current process is replaced by the new program.
Additional: The `execv()` function is one of the *exec* family of functions, which replace the current process image with a new process image. It does not create a new process; it replaces the current one. The `v` stands for the array of arguments, `p` is for searching in the `PATH` environment variable for the filename.

---

getpid, getppid

Declaration: `pid_t getpid();`
`pid_t getppid();`
Usage/Explanation: `getpid()` returns the PID of the calling process (in Linux, this is the TGID). `getppid()` returns the PID of the parent process.
Parameters: None
Return Values: Returns the PID of the calling process or its parent accordingly.
Additional:

---

gettid

Declaration: `pid_t gettid();`
Usage/Explanation: Returns the thread ID (TID) of the calling thread. In Linux, each thread has a unique process ID (PID) which is its TID.
Parameters: None
Return Values: Returns the TID of the calling thread.
Additional: This is a Linux-specific system call. `getpid()` returns the thread group ID (TGID), which is the same for all threads in a process, while `gettid()` returns the unique ID for each thread.

---

ptrace

Declaration: `long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);`
Usage/Explanation: Provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.
Parameters:

- `request`: Specifies the ptrace action to be performed (e.g., `PTRACE_ATTACH`, `PTRACE_DETACH`).
- `pid`: Specifies the process ID of the tracee.
- `addr`: Specifies an address in the tracee's memory space.
- `data`: Specifies data to be written to the tracee's memory or registers.

Return Values: On success, the return value depends on the request. On error, -1 is returned, and `errno` is set.

Additional: A process being traced becomes the child of the tracer process (its ‘parent‘ field in the PCB points to the tracer). The original parent is stored in ‘real_parent‘.

Topic 2: Signals

---

kill

Declaration: `int kill(pid_t pid, int sig);`
Usage/Explanation: sends signam num. `sig` to the process with PID `pid`.
Parameters:

- `pid`: The PID of the process to send the signal to.
  If `pid` is 0, the signal is sent to all processes in the same process group as the calling process.

- `sig`: The signal number to send. (e.g. `SIGKILL`, `SIGTERM`, etc.).

Return Values: Returns 0 on success, -1 on error. e.g. if the process doesn't exist
Additional: Since there is no signal with the number 0, it is used to check if the process exists or not. If the process exists, it returns 0, else it returns -1. (e.g. `kill(<pid>,0)`).

---

signal

Declaration: `sighandler_t signal(int signum, sighandler_t handler);`
`typedef sighandler_t void (*sighandler_t)(int);`
Usage/Explanation: Sets a signal handler for the specified signal `signum`.
Parameters:

- `signum`: The signal number to set the handler for. (e.g. `SIGINT`, `SIGTERM`, etc.).

- `handler`: The function to call when the signal is received. If `handler` is `SIG_IGN`, the signal is ignored. If `handler` is `SIG_DFL`, the default action for the signal is restored.

Return Values: On success, returns the previous signal handler for the specified signal. If there was no previous handler, it returns `SIG_DFL` or `SIG_IGN`. On error, it returns `SIG_ERR`.
Additional: `sigaction` is preferred over `signal` for portability and more features.

---

sigaction

Declaration: `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);`
Usage/Explanation: Examines and changes the action associated with a specific signal. It is more robust and portable than `signal()`.
Parameters:

- `signum`: The signal number.
- `act`: Pointer to a `sigaction` structure specifying the new action. If NULL, the action is not changed.
- `oldact`: Pointer to a `sigaction` structure where the old action is stored. If NULL, the old action is not saved.

Return Values: Returns 0 on success and -1 on error.
Additional: The `sigaction` structure contains fields for the handler, a signal mask to apply during handler execution, and flags to modify behavior.

---

sigprocmask

Declaration: `int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);`
Usage/Explanation: Examines and/or changes the signal mask of the calling thread. The signal mask is the set of signals whose delivery is currently blocked for the caller.
Parameters:

- `how`: Specifies how the signal mask is to be changed. Can be `SIG_BLOCK` (add signals to mask), `SIG_UNBLOCK` (remove signals), or `SIG_SETMASK` (replace mask).
- `set`: Pointer to a set of signals.
- `oldset`: If not NULL, the previous value of the signal mask is stored here.

Return Values: Returns 0 on success and -1 on error.
Additional: Used to prevent race conditions by temporarily blocking signals during critical sections.

---

alarm

Declaration: `unsigned int alarm(unsigned int seconds);`
Usage/Explanation: Arranges for a `SIGALRM` signal to be delivered to the process in `seconds` seconds.
Parameters: `seconds`: The number of seconds to wait before sending the signal. If 0, any pending alarm is canceled.

Return Values: Returns the number of seconds remaining until any previously scheduled alarm was due to be delivered, or zero if there was no previously scheduled alarm.
Additional:

---

setitimer

Declaration: `int setitimer(int which, const struct itimerval *new_value, struct itimerval *old_value);`
Usage/Explanation: Sets the value of the interval timer specified by `which`. This can be used to generate signals at regular intervals.
Parameters:

- `which`: The type of timer (`ITIMER_REAL` for `SIGALRM`, `ITIMER_VIRTUAL` for `SIGVTALRM`, `ITIMER_PROF` for `SIGPROF`).
- `new_value`: Specifies the new timer value (interval and initial value).
- `old_value`: If not NULL, stores the previous timer value.

Return Values: Returns 0 on success, -1 on error.
Additional: More flexible than `alarm()`, allowing for periodic timers.

---

setrlimit

Declaration: `int setrlimit(int resource, const struct rlimit *rlim);`
Usage/Explanation: Sets resource limits for a process. For example, it can set the maximum CPU time a process can consume.
Parameters:

- `resource`: The resource to limit (e.g., `RLIMIT_CPU`).
- `rlim`: A pointer to a `rlimit` structure that specifies the soft and hard limits for the resource.

Return Values: Returns 0 on success, -1 on error.
Additional: Exceeding the soft limit for CPU time results in a `SIGXCPU` signal. Exceeding the hard limit results in a `SIGKILL` signal.

Topic 3: Threads

---

pthread_create

Declaration: `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);`
Usage/Explanation: Creates a new thread within a process.
Parameters:

- `thread`: Pointer to a `pthread_t` variable that will be set to the ID of the new thread.
- `attr`: Pointer to attributes for the new thread (e.g., stack size). If NULL, default attributes are used.
- `start_routine`: The function that the new thread will execute.
- `arg`: The argument to be passed to the `start_routine`.

Return Values: Returns 0 on success, and a non-zero error code on failure.
Additional: The new thread shares the same memory space, file descriptors, etc., with the creating thread, but has its own stack and registers.

---

pthread_self

Declaration: `pthread_t pthread_self(void);`
Usage/Explanation: Returns the thread ID of the calling thread.
Parameters: None
Return Values: Returns the thread ID of the calling thread.
Additional: This ID is used to identify the thread in other pthread functions.

---

pthread_exit

Declaration: `void pthread_exit(void *retval);`
Usage/Explanation: Terminates the calling thread and makes a return value available to

any thread that joins it.
Parameters: `retval`: A pointer to the return value of the thread. This value can be obtained by another thread calling `pthread_join()`.
Return Values: This function does not return.
Additional: Calling `exit()` from any thread terminates the entire process, while `pthread_exit()` only terminates the calling thread.

---

pthread_cancel

Declaration: `int pthread_cancel(pthread_t thread);`
Usage/Explanation: Sends a cancellation request to the specified thread.
Parameters: `thread`: The ID of the thread to be canceled.
Return Values: Returns 0 on success, and a non-zero error code on failure.
Additional: Whether and when the target thread reacts to the cancellation request depends on its cancellation state and type.

---

pthread_join

Declaration: `int pthread_join(pthread_t thread, void **retval);`
Usage/Explanation: Waits for the specified thread to terminate. This is analogous to `wait()` for processes.
Parameters:

- `thread`: The ID of the thread to wait for.
- `retval`: A pointer to a location where the exit status of the terminated thread will be stored.

Return Values: Returns 0 on success, and a non-zero error code on failure.
Additional: A thread that is joined is automatically detached, and its resources are cleaned up.

---

clone

Declaration: `int clone(int (*fn)(void *), void *child_stack, int flags, void *arg, ...);`
Usage/Explanation: Creates a new child process, in a manner similar to `fork()`. Unlike `fork()`, `clone()` allows the child process to share parts of its execution context with the calling process, such as the memory space, the table of file descriptors, and the table of signal handlers.
Parameters:

- `fn`: Pointer to the function to be executed by the child process.
- `child_stack`: Pointer to the top of the stack for the child process.
- `flags`: A bitmask that specifies what is shared between the parent and child (e.g., `CLONE_VM`, `CLONE_FILES`, `CLONE_THREAD`).
- `arg`: Argument to be passed to the function `fn`.

Return Values: On success, the thread ID of the child process is returned in the parent's thread of execution. On failure, -1 is returned.
Additional: This is the underlying system call used by `fork()` and `pthread_create()` in Linux.

Topic 4: Synchronization

---

mutex_init

Declaration: `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);`
Usage/Explanation: Initializes a mutex object.
Parameters:

- `mutex`: A pointer to the mutex object to be initialized.
- `attr`: A pointer to a mutex attributes object. If NULL, default attributes are used.

Return Values: Returns 0 on success, and a non-zero error code on failure.
Additional: A mutex must be initialized before it can be used.

---

mutex_lock

Declaration: `int pthread_mutex_lock(pthread_mutex_t *mutex);`
Usage/Explanation: Locks a mutex. If the mutex is already locked by another thread, the calling thread blocks until the mutex becomes available.
Parameters: `mutex`: A pointer to the mutex object.
Return Values: Returns 0 on success, and a non-zero error code on failure.
Additional:

---

mutex_trylock

Declaration: `int pthread_mutex_trylock(pthread_mutex_t *mutex);`
Usage/Explanation: Attempts to lock a mutex without blocking. If the mutex is available, it is locked. If it is already locked, the function returns immediately with an error.
Parameters: `mutex`: A pointer to the mutex object.
Return Values: Returns 0 if the lock was acquired, and a non-zero error code otherwise (e.g., `EBUSY` if the mutex is already locked).
Additional:

---

mutex_unlock

Declaration: `int pthread_mutex_unlock(pthread_mutex_t *mutex);`
Usage/Explanation: Unlocks a mutex. This allows another thread that is waiting for the mutex to proceed.
Parameters: `mutex`: A pointer to the mutex object.
Return Values: Returns 0 on success, and a non-zero error code on failure.
Additional: Only the thread that locked a mutex should unlock it.

---

mutex_destroy

Declaration: `int pthread_mutex_destroy(pthread_mutex_t *mutex);`
Usage/Explanation: Destroys a mutex object, freeing any resources it might hold. The mutex must be unlocked.
Parameters: `mutex`: A pointer to the mutex object to be destroyed.
Return Values: Returns 0 on success, and a non-zero error code on failure.
Additional:

---

cond_wait

Declaration: `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`
Usage/Explanation: Atomically unlocks the mutex and waits for the condition variable `cond` to be signaled. The thread re-acquires the mutex before returning.
Parameters:

- `cond`: A pointer to the condition variable.
- `mutex`: A pointer to the associated mutex, which must be locked by the calling thread.

Return Values: Returns 0 on success, and a non-zero error code on failure.
Additional: Should be called in a loop to protect against spurious wakeups: `while(!condition) cond_wait(...);`

---

cond_signal

Declaration: `int pthread_cond_signal(pthread_cond_t *cond);`
Usage/Explanation: Wakes up at least one thread that is currently waiting on the specified condition variable.
Parameters: `cond`: A pointer to the condition variable.

Return Values: Returns 0 on success, and a non-zero error code on failure.
Additional:

---

cond_broadcast

Declaration: `int pthread_cond_broadcast(pthread_cond_t *cond);`
Usage/Explanation: Wakes up all threads that are currently waiting on the specified condition variable.
Parameters: `cond`: A pointer to the condition variable.
Return Values: Returns 0 on success, and a non-zero error code on failure.
Additional:

---

sem_init

Declaration: `int sem_init(sem_t *sem, int pshared, unsigned int value);`
Usage/Explanation: Initializes an unnamed semaphore.
Parameters:

- `sem`: A pointer to the semaphore object.
- `pshared`: If 0, the semaphore is shared between threads of a process. If non-zero, it is shared between processes.
- `value`: The initial value of the semaphore.

Return Values: Returns 0 on success, -1 on error.
Additional:

---

sem_wait

Declaration: `int sem_wait(sem_t *sem);`
Usage/Explanation: Decrements (locks) the semaphore. If the semaphore's value is greater than zero, the decrement proceeds, and the function returns immediately. If the semaphore currently has the value zero, the call blocks until it becomes possible to perform the decrement.
Parameters: `sem`: A pointer to the semaphore object.
Return Values: Returns 0 on success, -1 on error.
Additional: This operation is also known as P, down, or wait.

---

sem_post

Declaration: `int sem_post(sem_t *sem);`
Usage/Explanation: Increments (unlocks) the semaphore. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a `sem_wait()` call will be woken up and proceed to lock the semaphore.
Parameters: `sem`: A pointer to the semaphore object.
Return Values: Returns 0 on success, -1 on error.
Additional: This operation is also known as V, up, or signal.

---

sem_destroy

Declaration: `int sem_destroy(sem_t *sem);`
Usage/Explanation: Destroys an unnamed semaphore, freeing any resources it might hold.
Parameters: `sem`: A pointer to the semaphore object.
Return Values: Returns 0 on success, -1 on error.
Additional:

Topic 5: Scheduling

---

sched_yield

Declaration: `int sched_yield(void);`
Usage/Explanation: Causes the calling thread to relinquish the CPU. The thread is moved to the end of the queue for its static priority and a new thread gets to run.
Parameters: None
Return Values: Returns 0 on success, -1 on error.
Additional: A process voluntarily gives up the CPU.

---

sched_setscheduler

Declaration: `int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param);`
Usage/Explanation: Sets both the scheduling policy and parameters for the thread whose ID is specified in `pid`.
Parameters:

- `pid`: The process/thread ID. If 0, the scheduler of the calling thread is set.
- `policy`: The scheduling policy (e.g., `SCHED_OTHER`, `SCHED_FIFO`, `SCHED_RR`).
- `param`: Pointer to a structure containing the scheduling parameters (e.g., priority).

Return Values: Returns 0 on success, -1 on error.
Additional: Allows for setting realtime scheduling policies, which requires appropriate privileges.

---

nice

Declaration: `int nice(int inc);`
Usage/Explanation: Adds `inc` to the nice value for the calling thread. A higher nice value means a lower priority.

Parameters: `inc`: The value to add to the current nice value.
Return Values: On success, the new nice value is returned. On error, -1 is returned.
Additional: The range for user nice values is -20 (highest priority) to 19 (lowest priority).

Topic 6: File & I/O Operations

---

open

Declaration: `int open(const char *pathname, int flags, mode_t mode);` (mode_t is optional)
Usage/Explanation: Opens the requested file by `pathname` for access with the properties specified by `flags` and permissions specified by `mode`.
Parameters:

- `pathname`: The path to the file (or device) to open.
- `flags`: Flags that specify how the file should be opened. Must include one of the following:
  - `O_RDONLY`: Open for reading only.
  - `O_WRONLY`: Open for writing only.
  - `O_RDWR`: Open for reading and writing.

  Additional flags can be combined using the bitwise OR operator (`|`), such as:
  - `O_CREAT`: Create the file if it does not exist.
  - `O_TRUNC`: Truncate the file to zero length if it already exists.
  - `O_APPEND`: Append data to the end of the file.
- `mode`: Optional parameter that specifies the permissions of the file if it is created. It is used only if the `O_CREAT` flag is set, and is a must! It is a bitwise OR of the following permission bits:
  - `S_IRUSR`: Read permission for the owner.
  - `S_IWUSR`: Write permission for the owner.
  - `S_IXUSR`: Execute permission for the owner.
  - `S_IRGRP`: Read permission for the group.
  - `S_IWGRP`: Write permission for the group.
  - `S_IXGRP`: Execute permission for the group.
  - `S_IROTH`: Read permission for others.

- `S_IWOTH`: Write permission for others.
- `S_IXOTH`: Execute permission for others.

Return Values: In case of success, returns a file descriptor (an integer) that refers to the opened file. The given FD is the lowest available index in the FDT of the process. If the file cannot be opened, it returns -1 and sets `errno` to indicate the error.
Additional:

---

close

Declaration: `int close(int fd);`
Usage/Explanation: Closes the file descriptor `fd`, releasing the resources associated with it.
Parameters: `fd`: The file descriptor to close. It must be a valid file descriptor that was previously opened using `open()` or similar functions.
Return Values: Returns 0 on success, or -1 on error. If the file descriptor is invalid or already closed, it returns -1 and sets `errno` to indicate the error.
Additional: After closing an FD, you can no longer use it to access the file.

---

read

Declaration: `ssize_t read(int fd, void *buf, size_t count);`
Usage/Explanation: Reads up to `count` bytes from the file descriptor `fd` into the buffer pointed to by `buf`.
Parameters:

- `fd`: The file descriptor to read from.
- `buf`: A pointer to the buffer where the read data will be stored.
- `count`: The maximum number of bytes to read from the file descriptor.

Return Values: In case of success, returns the number of bytes read (which can be less than `count` if fewer bytes are available). If the end of the file is reached, it returns 0. On error, it returns -1 and sets `errno` to indicate the error.
Additional: `read()` is a blocking call by default, meaning it will wait until data is available to read.
Note that the seek pointer of the `fd` is advanced by the number of bytes read, so subsequent reads will continue from where the last read left off.

---

write

Declaration: `ssize_t write(int fd, const void *buf, size_t count);`
Usage/Explanation: Writes up to `count` bytes from the buffer pointed to by `buf` to the file descriptor `fd`.
Parameters:

- `fd`: The file descriptor to write to.
- `buf`: A pointer to the buffer containing the data to write.
- `count`: The number of bytes to write from the buffer.

Return Values: In case of success, returns the number of bytes written (which can be less than `count` if fewer bytes can be written). On error, it returns -1 and sets `errno` to indicate the error.
Additional: As with `read()`, `write()` is a blocking call by default, meaning it will wait until the data can be written. And the seek pointer of the `fd` is advanced by the number of bytes written.

---

lseek

Declaration: `off_t lseek(int fd, off_t offset, int whence);`
Usage/Explanation: Repositions the file offset of the open file description associated with the file descriptor `fd` to the argument `offset` according to the directive `whence`.
Parameters:

- `fd`: The file descriptor.
- `offset`: The new offset.
- `whence`: The directive for the new offset (`SEEK_SET` from the beginning, `SEEK_CUR` from the current position, `SEEK_END` from the end of the file).

Return Values: Upon successful completion, `lseek()` returns the resulting offset location as measured in bytes from the beginning of the file. On error, the value (off_t) -1 is returned.
Additional:

---

pread

Declaration: `ssize_t pread(int fd, void *buf, size_t count, off_t offset);`
Usage/Explanation: Reads up to `count` bytes from file descriptor `fd` at `offset` into the buffer `buf`. The file offset is not changed.
Parameters:

- `fd`: The file descriptor.
- `buf`: The buffer to store the data.
- `count`: The number of bytes to read.
- `offset`: The offset in the file to start reading from.

Return Values: On success, the number of bytes read is returned. On error, -1 is returned.
Additional: Useful for multi-threaded applications where multiple threads read from the same file descriptor without interfering with each other's file offset.

---

pwrite

Declaration: `ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);`
Usage/Explanation: Writes up to `count` bytes from the buffer `buf` to the file descriptor `fd` at `offset`. The file offset is not changed.
Parameters:

- `fd`: The file descriptor.
- `buf`: The buffer containing the data.
- `count`: The number of bytes to write.
- `offset`: The offset in the file to start writing to.

Return Values: On success, the number of bytes written is returned. On error, -1 is returned.
Additional: Similar to `pread`, it's an atomic operation that doesn't affect the file's current offset.

---

ioctl

Declaration: `int ioctl(int fd, unsigned long request, ...);`
Usage/Explanation: Manipulates the underlying device parameters of special files. The third argument is an untyped pointer to memory.
Parameters:

- `fd`: The file descriptor of the device.
- `request`: A device-dependent request code.
- `...`: An optional untyped pointer to memory, used to pass data to/from the device driver.

Return Values: Usually, on success 0 is returned. A few `ioctl` requests use the return value as an output parameter. On error, -1 is returned.
Additional: Allows adding additional functionality to a device driver beyond the standard read/write operations.

---

pipe

Declaration: `int pipe(int filedes[2]);`
Usage/Explanation: Creates a unidirectional data channel *(pipe)* with two FDs: one for reading and one for writing.
Parameters:

- `filedes`: An array of two integers that the syscall will fill with:

  1. `filedes[0]`: The file descriptor for reading from the pipe.
  2. `filedes[1]`: The file descriptor for writing to the pipe.

Return Values: Return 0 if successful, otherwise -1.
Additional: Pipes reside in memory, not on disk, and are used for IPC between related processes (e.g., parent and child).

---

dup, dup2

Declaration: `int dup(int oldfd);`
`int dup2(int oldfd, int newfd);`
Usage/Explanation: Creates a copy of the file descriptor `oldfd` and returns a new file descriptor that refers to the same open file description.
Parameters:

- `oldfd`: The file descriptor to duplicate. Must be an open file descriptor.

- `newfd`: The desired new file descriptor (only for `dup2()`). If `newfd` is already open, it will be closed before being reused.

Return Values: For `dup()`: Returns the lowest numbered unused file descriptor.
For `dup2()`: Returns `newfd` if successful, or -1 on error.
Additional: If we want to close a file we need to close all of its FDs, which they all point to the same *File Object*

---

sync

Declaration: `void sync(void);`
Usage/Explanation: Causes all buffered modifications to file metadata and data to be written to the underlying filesystems.
Parameters: None
Return Values: This function always succeeds.
Additional: This is a system-wide sync. `fsync` is preferred for syncing a single file.

---

fsync

Declaration: `int fsync(int fd);`
Usage/Explanation: Transfers ("flushes") all modified in-core data of (i.e., modified buffer cache pages for) the file referred to by the file descriptor `fd` to the disk device.
Parameters: `fd`: The file descriptor of the file to be synced.
Return Values: Returns 0 on success, -1 on error.
Additional: Ensures that data is physically written to the storage device, providing data integrity.

Topic 7: Filesystem & Directories

---

mkfifo

Declaration: `int mkfifo(const char *pathname, mode_t mode);`
Usage/Explanation: Creates a named pipe (FIFO) with the specified `pathname` and permissions `mode`.
Parameters:

- `pathname`: The path where the FIFO will be created.
- `mode`: The permissions for the FIFO, similar to those used in the `open()` syscall. 0777 means XRW for all users.

Return Values: 0 on success, -1 on error.
Additional: FIFO are shown as files in the filesystem, but they are not saved on disk. It is a bidirectional communication channel between processes.
Note: The FIFO must be opened by at least one process before any other process can write to it, i.e. it's blocking.
If opened for read & write it will be non-blocking

---

creat

Declaration: `int creat(const char *pathname, mode_t mode);`
Usage/Explanation: Creates a new file or overwrites an existing one. It is equivalent to `open(pathname, O_CREAT|O_WRONLY|O_TRUNC, mode)`.
Parameters: `pathname`: The path of the file to create. `mode`: The permissions for the new file.
Return Values: Returns a new file descriptor for the file, or -1 on error.
Additional: This function is considered obsolete; `open()` is preferred.

---

unlink

Declaration: `int unlink(const char *pathname);`
Usage/Explanation: Deletes a name from the filesystem. If that name was the last link to a file and no processes have the file open, the file is deleted and the space it was using is made available for reuse.
Parameters: `pathname`: The path of the file name to delete.
Return Values: Returns 0 on success, -1 on error.
Additional: Used to remove files and symbolic links.

---

rmdir

Declaration: `int rmdir(const char *pathname);`
Usage/Explanation: Deletes a directory, which must be empty.
Parameters: `pathname`: The path of the directory to delete.
Return Values: Returns 0 on success, -1 on error.
Additional:

---

rename

Declaration: `int rename(const char *oldpath, const char *newpath);`
Usage/Explanation: Renames a file, moving it between directories if required.
Parameters:

- `oldpath`: The current path of the file.
- `newpath`: The new path for the file.

Return Values: Returns 0 on success, -1 on error.
Additional: This operation is atomic.

---

stat, fstat, lstat

Declaration: `int stat(const char *pathname, struct stat *statbuf); int fstat(int fd, struct stat *statbuf); int lstat(const char *pathname, struct stat *statbuf);`
Usage/Explanation: Retrieve information about the file pointed to by `pathname` or `fd`.
Parameters:

- `pathname/fd`: The file to get information about.
- `statbuf`: A pointer to a buffer where the file information will be stored.

Return Values: Returns 0 on success, -1 on error.
Additional: `stat` follows symbolic links, `lstat` provides information about the link itself, and `fstat` operates on an open file descriptor.

---

chmod, fchmod

Declaration: `int chmod(const char *pathname, mode_t mode); int fchmod(int fd, mode_t mode);`
Usage/Explanation: Changes the permissions of a file.
Parameters:

- `pathname/fd`: The file to change permissions for.
- `mode`: The new permission bits.

Return Values: Returns 0 on success, -1 on error.
Additional:

---

chown, fchown

Declaration: `int chown(const char *pathname, uid_t owner, gid_t group); int fchown(int fd, uid_t owner, gid_t group);`
Usage/Explanation: Changes the ownership of a file.
Parameters:

- `pathname/fd`: The file to change ownership for.
- `owner`: The new user ID.
- `group`: The new group ID.

Return Values: Returns 0 on success, -1 on error.
Additional:

---

mkdir

Declaration: `int mkdir(const char *pathname, mode_t mode);`
Usage/Explanation: Creates a new directory.
Parameters: `pathname`: The path of the new directory. `mode`: The permissions for the new directory.
Return Values: Returns 0 on success, -1 on error.
Additional:

---

symlink

Declaration: `int symlink(const char *target, const char *linkpath);`
Usage/Explanation: Creates a symbolic link named `linkpath` which contains the string `target`.
Parameters:

- `target`: The path that the symbolic link will point to.
- `linkpath`: The path of the symbolic link itself.

Return Values: Returns 0 on success, -1 on error.
Additional:

---

readlink

Declaration: `ssize_t readlink(const char *pathname, char *buf, size_t bufsiz);`
Usage/Explanation: Places the contents of the symbolic link `pathname` in the buffer `buf`, which has size `bufsiz`.
Parameters:

- `pathname`: The path of the symbolic link.
- `buf`: The buffer to store the target path.
- `bufsiz`: The size of the buffer.

Return Values: On success, returns the number of bytes placed in `buf`. On error, -1 is returned.
Additional:

---

mount

Declaration: `int mount(const char *source, const char *target, const char *filesystemtype unsigned long mountflags, const void *data);`
Usage/Explanation: Attaches the filesystem specified by `source` to the directory specified by `target`.
Parameters:

- `source`: The device or resource containing the filesystem.
- `target`: The mount point directory.
- `filesystemtype`: The type of the filesystem (e.g., "ext4").
- `mountflags`: Flags controlling the mount operation.

- `data`: Filesystem-specific data.

Return Values: Returns 0 on success, -1 on error.
Additional: This is how filesystems become part of the main directory tree.

Topic 8: Networking

---

socket

Declaration: `int socket(int domain, int type, int protocol);`
Usage/Explanation: Creates an endpoint for communication and returns a file descriptor that refers to that endpoint.
Parameters:

- `domain`: The communication domain (e.g., `AF_INET` for IPv4).
- `type`: The communication semantics (e.g., `SOCK_STREAM` for TCP, `SOCK_DGRAM` for UDP).
- `protocol`: The protocol to be used. Usually 0 to select the default for the given type.

Return Values: Returns a file descriptor for the new socket, or -1 on error.
Additional: This is the first step in network communication.

---

bind

Declaration: `int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`
Usage/Explanation: Assigns the address specified by `addr` to the socket referred to by the file descriptor `sockfd`.
Parameters:

- `sockfd`: The socket file descriptor.
- `addr`: A pointer to a `sockaddr` structure containing the address (IP and port) to be bound.
- `addrlen`: The length of the address structure.

Return Values: Returns 0 on success, -1 on error.
Additional: Typically used on the server side to assign a well-known port.

---

listen

Declaration: `int listen(int sockfd, int backlog);`
Usage/Explanation: Marks the socket referred to by `sockfd` as a passive socket, that is, as a socket that will be used to accept incoming connection requests using `accept()`.
Parameters:

- `sockfd`: The socket file descriptor.
- `backlog`: The maximum length to which the queue of pending connections for `sockfd` may grow.

Return Values: Returns 0 on success, -1 on error.
Additional: Used only on the server side for TCP sockets.

---

accept

Declaration: `int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);`
Usage/Explanation: Extracts the first connection request on the queue of pending connections for the listening socket, `sockfd`, creates a new connected socket, and returns a new file descriptor referring to that socket.
Parameters:

- `sockfd`: The listening socket file descriptor.
- `addr`: A pointer to a `sockaddr` structure to be filled with the address of the connecting client.
- `addrlen`: A pointer to the size of the address structure.

Return Values: Returns a new file descriptor for the connected socket, or -1 on error.
Additional: This is a blocking call; it waits until a client connects.

---

connect

Declaration: `int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`
Usage/Explanation: Connects the socket referred to by the file descriptor `sockfd` to the address specified by `addr`.
Parameters:

- `sockfd`: The socket file descriptor.
- `addr`: A pointer to a `sockaddr` structure containing the server's address.
- `addrlen`: The length of the address structure.

Return Values: Returns 0 on success, -1 on error.
Additional: Used on the client side to establish a connection with a server.

---

select

Declaration: `int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);`
Usage/Explanation: 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.
Parameters:

- `nfds`: The highest-numbered file descriptor in any of the three sets, plus one.
- `readfds`: Set of FDs to monitor for reading.
- `writefds`: Set of FDs to monitor for writing.
- `exceptfds`: Set of FDs to monitor for exceptional conditions.
- `timeout`: Maximum interval to wait. If NULL, block indefinitely.

Return Values: Returns the number of ready file descriptors, 0 if the timeout expires, and -1 on error.
Additional: Enables event-oriented programming for handling multiple I/O channels concurrently.

Topic 9: Virtual Memory

---

mmap

Declaration: `void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);`
Usage/Explanation: Creates a new mapping in the virtual address space of the calling process.
Parameters:

- `addr`: The starting address for the new mapping. If NULL, the kernel chooses the address.
- `length`: The length of the mapping.
- `prot`: Desired memory protection (`PROT_READ`, `PROT_WRITE`, `PROT_EXEC`).
- `flags`: Determines whether updates are visible to other processes (`MAP_SHARED`, `MAP_PRIVATE`) and other options (`MAP_ANONYMOUS`).
- `fd`: File descriptor for file-backed mapping. -1 for anonymous mapping.
- `offset`: Offset in the file to start mapping from.

Return Values: On success, returns a pointer to the mapped area. On error, `MAP_FAILED` is returned.
Additional: A powerful tool for file I/O, IPC, and dynamic memory allocation.

---

sbrk

Declaration: `void *sbrk(intptr_t increment);`
Usage/Explanation: Increments the program's data space by `increment` bytes. Calling `sbrk()` with an increment of 0 can be used to find the current location of the program break.
Parameters: `increment`: The number of bytes to add to the data space.
Return Values: On success, returns the previous program break. On error, (void*) -1 is returned.
Additional: The traditional way to implement `malloc`. Modern implementations often use `mmap` instead.

Topic 10: Kernel Modules

---

init_module

Declaration: `int init_module(void);`
Usage/Explanation: The entry point for a kernel module. This function is called when the module is loaded into the kernel.
Parameters: None
Return Values: Returns 0 on success, or a non-zero error code on failure.
Additional: Responsible for registering drivers, creating device files, and initializing hardware.

---

cleanup_module

Declaration: `void cleanup_module(void);`
Usage/Explanation: The exit point for a kernel module. This function is called when the module is unloaded from the kernel.
Parameters: None
Return Values: None
Additional: Responsible for unregistering drivers, deleting device files, and releasing resources.

---

module_param

Declaration: `module_param(name, type, perm);`
Usage/Explanation: A macro used to declare a module parameter that can be changed at load time.
Parameters:

- `name`: The name of the parameter.
- `type`: The data type of the parameter (e.g., int, charp).
- `perm`: The file permissions for the parameter's entry in `/sys/module/`.

Return Values: N/A (it's a macro).
Additional: Allows for flexible configuration of kernel modules without recompiling.

---

mknod

Declaration: `int mknod(const char *pathname, mode_t mode, dev_t dev);`
Usage/Explanation: Creates a filesystem node (a file, device special file, or named pipe) named `pathname`, with attributes specified by `mode` and `dev`.
Parameters:

- `pathname`: The path for the new node.
- `mode`: Specifies both the permissions to use and the type of node to be created.
- `dev`: If the node is a character or block special file, this specifies the major and minor numbers of the newly created device special file.

Return Values: Returns 0 on success, -1 on error.
Additional: Used to create device files in `/dev`.

---

register_chrdev

Declaration: `int register_chrdev(unsigned int major, const char *name, const struct file_operations *fops);`
Usage/Explanation: Registers a character device driver with the kernel.
Parameters:

- `major`: The major number to be allocated. If 0, the kernel allocates a dynamic major number.
- `name`: The name of the driver, which will appear in `/proc/devices`.
- `fops`: A pointer to the file operations structure for the driver.

Return Values: On success, returns the allocated major number. On failure, a negative error code is returned.
Additional: Connects a major number to a set of driver functions (`file_operations`).

---

unregister_chrdev

Declaration: `void unregister_chrdev(unsigned int major, const char *name);`
Usage/Explanation: Unregisters a character device driver from the kernel.
Parameters:

- `major`: The major number of the driver to unregister.
- `name`: The name of the driver.

Return Values: None
Additional: Called from the module's cleanup function.

Part II

Cheat Sheet

Function Summary Cheat Sheet

Function	Summary
`fork()`	Creates a child process identical to the parent. Returns 0 to child, PID to parent, -1 on error.
`wait()`	Suspends the process until any child ends. Stores exit status in provided pointer.
`waitpid()`	Waits for a specific child to finish. Can use options like `WNOHANG`.
`exit()`	Terminates the calling process with a status code. Becomes a zombie until parent collects status.
`execv()`	Replaces the current process image with a new one using the provided filename and argument array.
`getpid()`	Returns the PID (TGID in Linux) of the current process.
`getppid()`	Returns the PID of the parent process.
`gettid()`	Returns the thread ID (TID) of the calling thread (Linux-specific).
`ptrace()`	Allows a tracer process to observe and control a tracee process. Used for debugging.
`kill()`	Sends a signal to a process by PID. Signal 0 used to check process existence.
`signal()`	Sets a simple signal handler for a given signal number. (`sigaction` is preferred).
`sigaction()`	Sets a detailed signal handler, allowing control over masks and flags.
`sigprocmask()`	Examines and/or changes the signal mask of the calling thread.
`alarm()`	Sends `SIGALRM` after a specified number of seconds.
`setitimer()`	Sets an interval timer that can be periodic (`SIGALRM`, `SIGVTALRM`, `SIGPROF`).
`setrlimit()`	Sets resource limits (e.g., CPU time) for a process.
`pthread_create()`	Creates a new thread.
`pthread_self()`	Returns the ID of the calling thread.
`pthread_exit()`	Terminates the calling thread.
`pthread_cancel()`	Sends a cancellation request to a thread.
`pthread_join()`	Waits for a thread to terminate and retrieves its exit status.
`clone()`	The underlying Linux syscall for creating threads and processes with shared resources.
`mutex_init()`	Initializes a mutex.
`mutex_lock()`	Locks a mutex, blocking if necessary.
`mutex_trylock()`	Attempts to lock a mutex without blocking.

Function	Summary
`mutex_unlock()`	Unlocks a mutex.
`mutex_destroy()`	Destroys a mutex.
`cond_wait()`	Atomically unlocks a mutex and waits on a condition variable.
`cond_signal()`	Wakes up one thread waiting on a condition variable.
`cond_broadcast()`	Wakes up all threads waiting on a condition variable.
`sem_init()`	Initializes an unnamed semaphore.
`sem_wait()`	Decrements (waits on) a semaphore, blocking if its value is zero.
`sem_post()`	Increments (signals) a semaphore, waking a waiting thread if any.
`sem_destroy()`	Destroys an unnamed semaphore.
`sched_yield()`	Causes the calling thread to relinquish the CPU.
`sched_setscheduler`	Sets the scheduling policy and parameters for a thread.
`nice()`	Adjusts the scheduling priority of a process by changing its nice value.
`open()`	Opens or creates a file, returning a file descriptor.
`close()`	Closes a file descriptor.
`read()`	Reads up to `count` bytes from a file descriptor into a buffer.
`write()`	Writes up to `count` bytes from a buffer to a file descriptor.
`lseek()`	Repositions the read/write file offset.
`pread()/pwrite()`	Reads/writes from/to a given offset without changing the file's current offset.
`ioctl()`	Performs device-specific control operations.
`pipe()`	Creates a unidirectional pipe with two FDs: one for reading, one for writing.
`dup()/dup2()`	Duplicates a file descriptor.
`sync()`	Schedules all buffered file data and metadata to be written to disk.
`fsync()`	Flushes all data and metadata for a specific file descriptor to disk.
`mkfifo()`	Creates a named FIFO (pipe) with given pathname and mode.
`creat()`	Obsolete function to create a file; use `open()` instead.
`unlink()`	Deletes a name (hard link) from the filesystem.
`rmdir()`	Removes an empty directory.
`rename()`	Renames or moves a file.
`stat()/lstat()`	Get file status (metadata). `lstat` does not follow symlinks.
`chmod()`	Changes file permissions.
`chown()`	Changes file ownership.

Function	Summary
`mkdir()`	Creates a directory.
`symlink()`	Creates a symbolic link.
`readlink()`	Reads the value of a symbolic link.
`mount()`	Attaches a filesystem to the directory tree.
`socket()`	Creates a communication endpoint (socket).
`bind()`	Assigns an address (IP/port) to a socket.
`listen()`	Puts a TCP socket in listening mode for incoming connections.
`accept()`	Accepts an incoming connection on a listening socket.
`connect()`	Establishes a connection to a server.
`select()`	Monitors multiple file descriptors for I/O readiness.
`mmap()`	Maps files or devices into memory.
`sbrk()`	Changes the data segment size (used for heap allocation).
`init_module()`	Entry point function when a kernel module is loaded.
`cleanup_module()`	Exit point function when a kernel module is unloaded.
`module_param()`	Macro to declare a kernel module parameter.
`mknod()`	Creates a special or ordinary file (e.g., device files).
`register_chrdev()`	Registers a character device driver.
`unregister_chrdev`	Unregisters a character device driver.