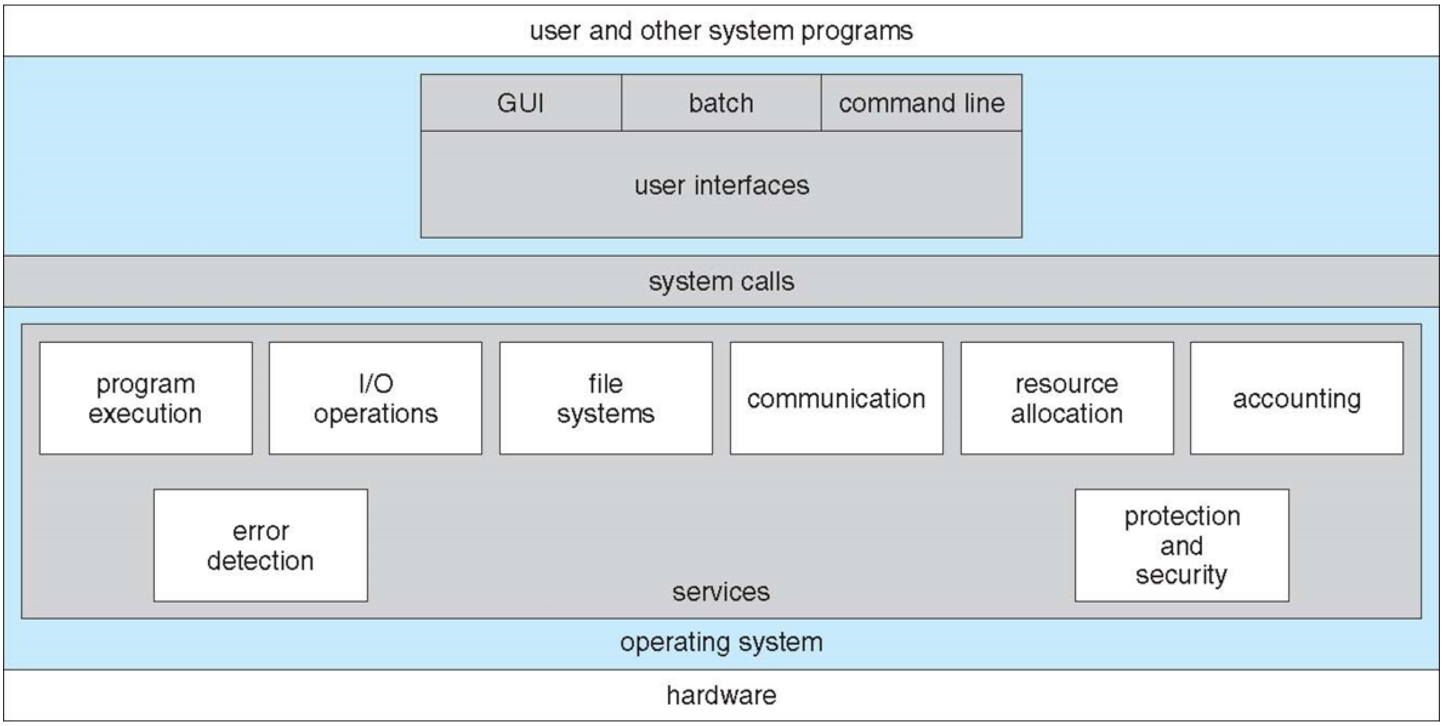
Tutorial 2

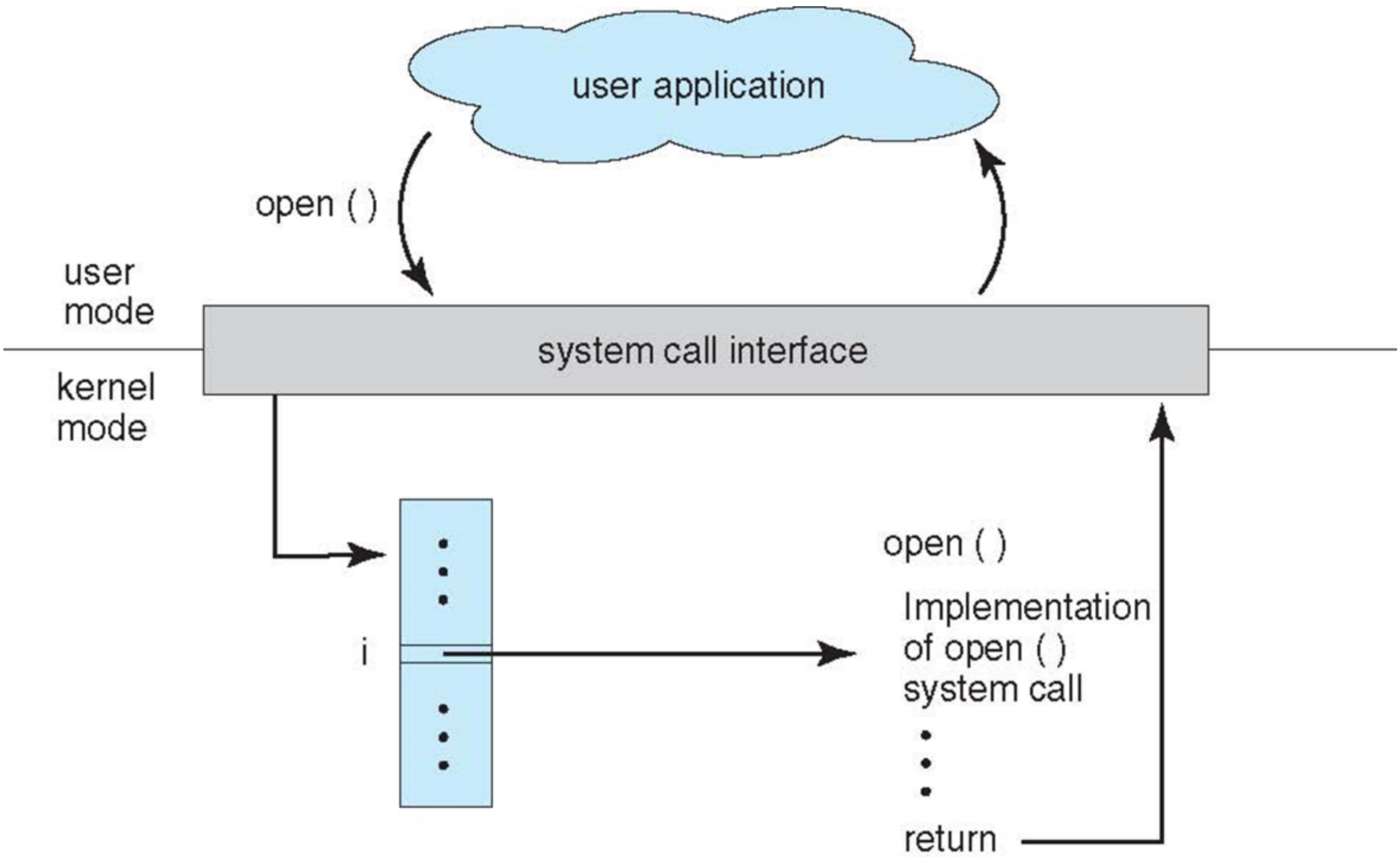
Recap

System Calls

- **Programming interface to the services provided by the OS**



- **Typically written in a high-level language (C++)**
- **Mostly accessed by programs via a high-level Application Programming Interface (API) rather than direct system call use**
- **Three most common system call APIs are**
  - **Win32 API for Windows,**
  - **POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and**
  - **Java API for the Java virtual machine (JVM)**



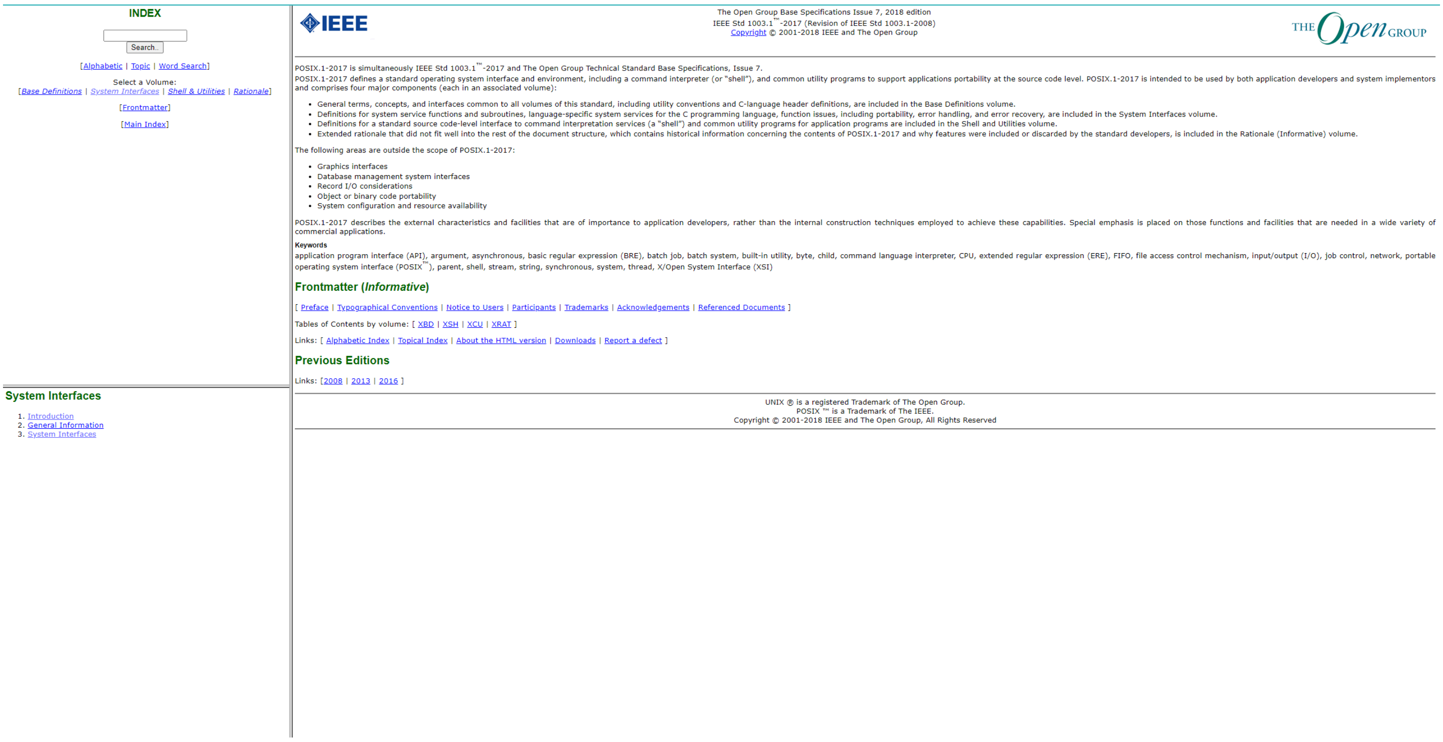
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
    - Just needs to obey API and understand what OS will do as a result call
    - Most details of OS interface hidden from programmer by API
        - Managed by run-time support library (set of functions built into libraries included with compiler)
- Typically, a number associated with each system call
    - **System-call interface** maintains a table indexed according to these numbers

System Call Programming Introduction

Reading Material

The Open Group Base Specifications Issue 7, 2018 edition

The System Interfaces volume of POSIX



INDEX

Search..

[Alphabetic | Topic | Word Search]

Select a Volume:
[Base Definitions | System Interfaces | Shell & Utilities | Rationale]

[Frontmatter]

[Main Index]

System Interfaces

1. Introduction
2. General Information
3. System Interfaces

IEEE

The Open Group Base Specifications Issue 7, 2018 edition
IEEE Std 1003.1™-2017 (Revision of IEEE Std 1003.1-2008)
Copyright © 2001-2018 IEEE and The Open Group

THE Open GROUP

POSIX.1-2017 is simultaneously IEEE Std 1003.1™-2017 and The Open Group Technical Standard Base Specifications, Issue 7.
POSIX.1-2017 defines a standard operating system interface and environment, including a command interpreter (or "shell"), and common utility programs to support applications portability at the source code level. POSIX.1-2017 is intended to be used by both application developers and system implementors and comprises four major components (each in an associated volume):

- General terms, concepts, and interfaces common to all volumes of this standard, including utility conventions and C-language header definitions, are included in the Base Definitions volume.
- Definitions for system service functions and subroutines, language-specific system services for the C programming language, function issues, including portability, error handling, and error recovery, are included in the System Interfaces volume.
- Definitions for a standard source code-level interface to command interpretation services (a "shell") and common utility programs for application programs are included in the Shell and Utilities volume.
- Extended rationale that did not fit well into the rest of the document structure, which contains historical information concerning the contents of POSIX.1-2017 and why features were included or discarded by the standard developers, is included in the Rationale (Informative) volume.

The following areas are outside the scope of POSIX.1-2017:

- Graphics interfaces
- Database management system interfaces
- Record I/O considerations
- Object or binary code portability
- System configuration and resource availability

POSIX.1-2017 describes the external characteristics and facilities that are of importance to application developers, rather than the internal construction techniques employed to achieve these capabilities. Special emphasis is placed on those functions and facilities that are needed in a wide variety of commercial applications.

Keywords
application program interface (API), argument, asynchronous, basic regular expression (BRE), batch job, batch system, built-in utility, byte, child, command language interpreter, CPU, extended regular expression (ERE), FIFO, file access control mechanism, input/output (I/O), job control, network, portable operating system interface (POSIX™), parent, shell, stream, string, synchronous, system, thread, X/Open System Interface (XSI)

Frontmatter (*Informative*)

[Preface | Typographical Conventions | Notice to Users | Participants | Trademarks | Acknowledgements | Referenced Documents]

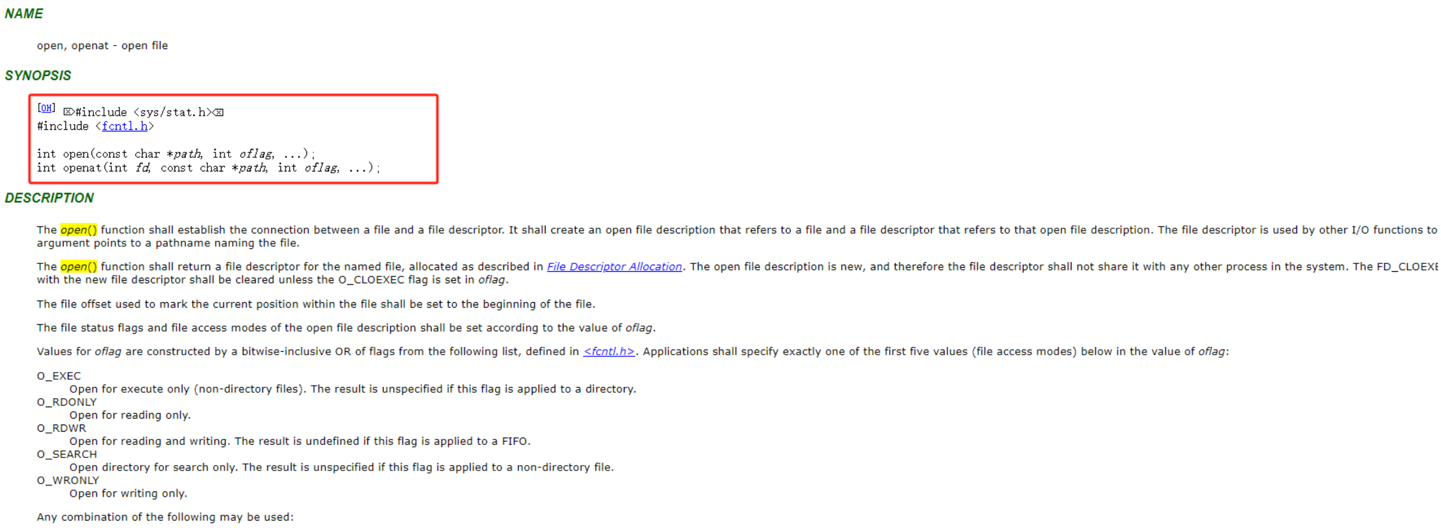
Tables of Contents by volume: [XBD | XSH | XCU | XRAT]

Links: [Alphabetic Index | Topical Index | About the HTML version | Downloads | Report a defect]

Previous Editions

Links: [2008 | 2013 | 2016]

Take POSIX System call open() as an example



NAME

open, openat - open file

SYNOPSIS

```
[OH] ⌦#include <sys/stat.h>⌫
#include <fcntl.h>

int open(const char *path, int oflag, ...);
int openat(int fd, const char *path, int oflag, ...);
```

DESCRIPTION

The *open()* function shall establish the connection between a file and a file descriptor. It shall create an open file description that refers to a file and a file descriptor that refers to that open file description. The file descriptor is used by other I/O functions to argument points to a pathname naming the file.

The *open()* function shall return a file descriptor for the named file, allocated as described in *File Descriptor Allocation*. The open file description is new, and therefore the file descriptor shall not share it with any other process in the system. The FD_CLOEXE with the new file descriptor shall be cleared unless the O_CLOEXEC flag is set in *oflag*.

The file offset used to mark the current position within the file shall be set to the beginning of the file.

The file status flags and file access modes of the open file description shall be set according to the value of *oflag*.

Values for *oflag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in *<fcntl.h>*. Applications shall specify exactly one of the first five values (file access modes) below in the value of *oflag*:

O_EXEC
    Open for execute only (non-directory files). The result is unspecified if this flag is applied to a directory.
O_RDONLY
    Open for reading only.
O_RDWR
    Open for reading and writing. The result is undefined if this flag is applied to a FIFO.
O_SEARCH
    Open directory for search only. The result is unspecified if this flag is applied to a non-directory file.
O_WRONLY
    Open for writing only.

Any combination of the following may be used:

Practice:

Manage Files

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

ssize_t read(int fd, void buf[.count], size_t count);

ssize_t write(int fd, const void buf[.count], size_t count);
```

open()

The **open()** system call opens the file specified by pathname. If the specified file does not exist, it may optionally (if O_CREAT is specified in flags) be created by open().

DESCRIPTION

The *open*() function shall establish the connection between a file and a file descriptor. It shall create an open file description that refers to a file and a file descriptor used by other I/O functions to refer to that file. The *path* argument points to a pathname naming the file.

The *open*() function shall return a file descriptor for the named file, allocated as described in *File Descriptor Allocation*. The open file description is new, and therefor the system. The FD_CLOEXEC file descriptor flag associated with the new file descriptor shall be cleared unless the O_CLOEXEC flag is set in *oflag*.

The file offset used to mark the current position within the file shall be set to the beginning of the file.

The file status flags and file access modes of the open file description shall be set according to the value of *oflag*.

Values for *oflag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in *<fcntl.h>*. Applications shall specify exactly one of the first five v

O_EXEC
: Open for execute only (non-directory files). The result is unspecified if this flag is applied to a directory.

O_RDONLY
: Open for reading only.

O_RDWR
: Open for reading and writing. The result is undefined if this flag is applied to a FIFO.

O_SEARCH
: Open directory for search only. The result is unspecified if this flag is applied to a non-directory file.

O_WRONLY
: Open for writing only.

Any combination of the following may be used:

O_APPEND
: If set, the file offset shall be set to the end of the file prior to each write.

O_CLOEXEC
: If set, the FD_CLOEXEC flag for the new file descriptor shall be set.

O_CREAT
: If the file exists, this flag has no effect except as noted under O_EXCL below. Otherwise, if O_DIRECTORY is not set the file shall be created as a regular file; t the process; the group ID of the file shall be set to the group ID of the file's parent directory or to the effective group ID of the process; and the access permi the value of the argument following the *oflag* argument taken as type **mode_t** modified as follows: a bitwise AND is performed on the file-mode bits and the c creation mask. Thus, all bits in the file mode whose corresponding bit in the file mode creation mask is set are cleared. When bits other than the file permissic following the *oflag* argument does not affect whether the file is open for reading, writing, or for both. Implementations shall provide a way to initialize the file' Implementations may, but need not, provide an implementation-defined way to initialize the file's group ID to the effective group ID of the calling process.

O_DIRECTORY
: If *path* resolves to a non-directory file, fail and set *errno* to [ENOTDIR].

O_DSYNC
: [SIO] ⌦ Write I/O operations on the file descriptor shall complete as defined by synchronized I/O data integrity completion. ⌫

O_EXCL
: If O_CREAT and O_EXCL are set, *open*() shall fail if the file exists. The check for the existence of the file and the creation of the file if it does not exist shall be the same filename in the same directory with O_EXCL and O_CREAT set. If O_EXCL and O_CREAT are set, and *path* names a symbolic link, *open*() shall fail ar symbolic link. If O_EXCL is set and O_CREAT is not set, the result is undefined.

O_NOCTTY
: If set and *path* identifies a terminal device, *open*() shall not cause the terminal device to become the controlling terminal for the process. If *path* does not ider

O_NOFOLLOW
: If *path* names a symbolic link, fail and set *errno* to [ELOOP].

O_NONBLOCK
: When opening a FIFO with O_RDONLY or O_WRONLY set:

The return value of open() is a file descriptor, a small, nonnegative integer that is an index to an entry in the process's table of open file descriptors. The file descriptor is used in subsequent system calls (read(2), write(2), lseek(2), fcntl(2), etc.) to refer to the open file. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

read()

```
ssize_t read(int fd, void buf[.count], size_t count);
```

read() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*. On files that support seeking, the read operation commences at the file offset, and the file offset is incremented by the number of bytes read. If the file offset is at or past the end of file, no bytes are read, and **read**() returns zero.

If *count* is zero, **read**() *may* detect the errors described below. In the absence of any errors, or if **read**() does not check for errors, a **read**() with a *count* of 0 returns zero and has no other effects.

write()

```
ssize_t write(int fd, const void buf[.count], size_t count);
```

write() writes up to *count* bytes from the buffer starting at *buf* to the file referred to by the file descriptor *fd*.
The number of bytes written may be less than *count* if, for example, there is insufficient space on the underlying physical medium, or the **RLIMIT_FSIZE** resource limit is encountered (seesetrlimit(2)), or the call was interrupted by a signal handler after having written less than *count* bytes. (See also pipe(7).)

For a seekable file (i.e., one to which lseek(2) may be applied, for example, a regular file) writing takes place at the file offset, and the file offset is incremented by the number of bytes actually written. If the file was open(2)ed with **O_APPEND**, the file offset is first set to the end of the file before writing.
The adjustment of the file offset and the write operation are performed as an atomic step.

Map files

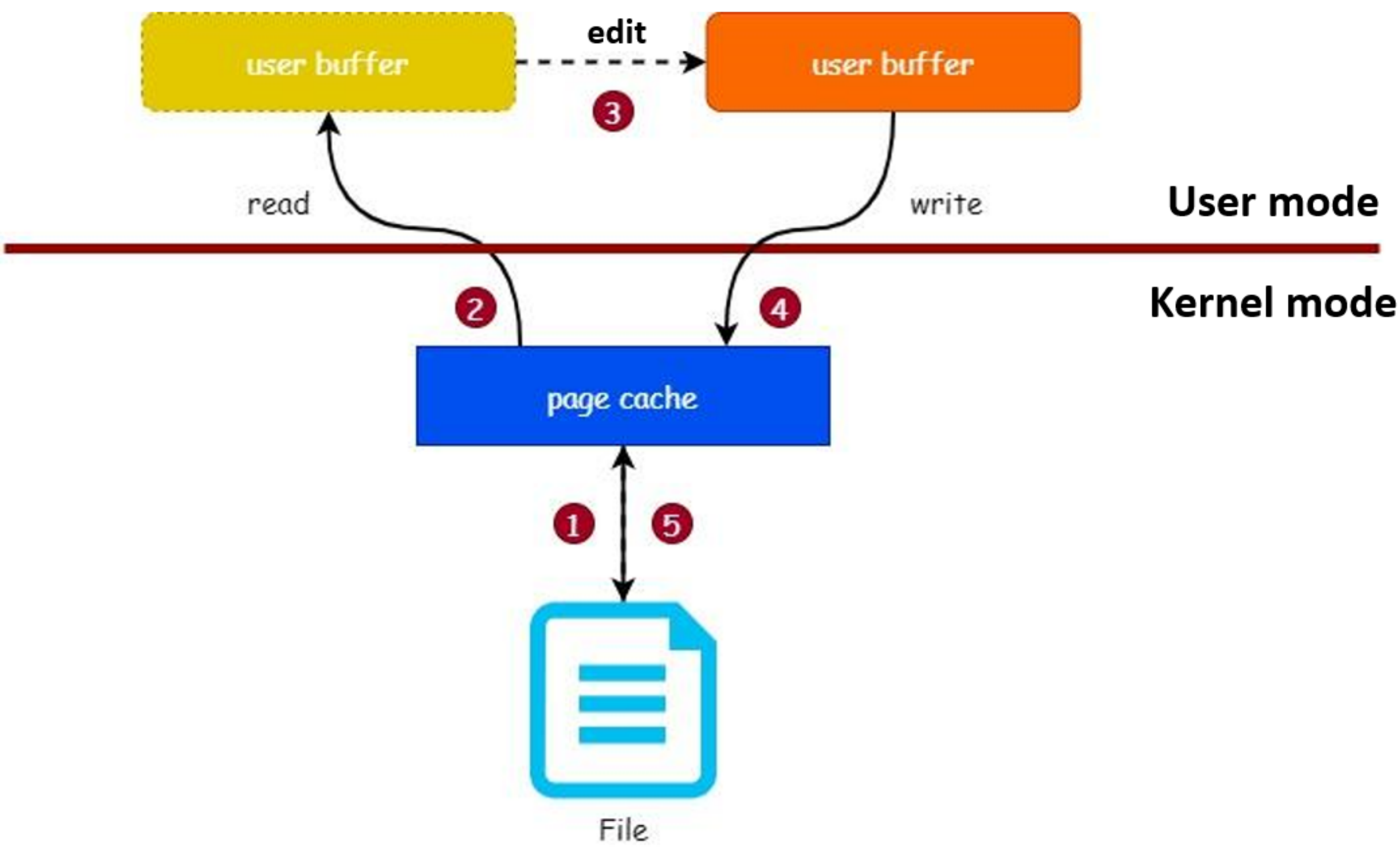
mmap() creates a new mapping in the virtual address space of the calling process. (Why?)

Concept of memory mapping

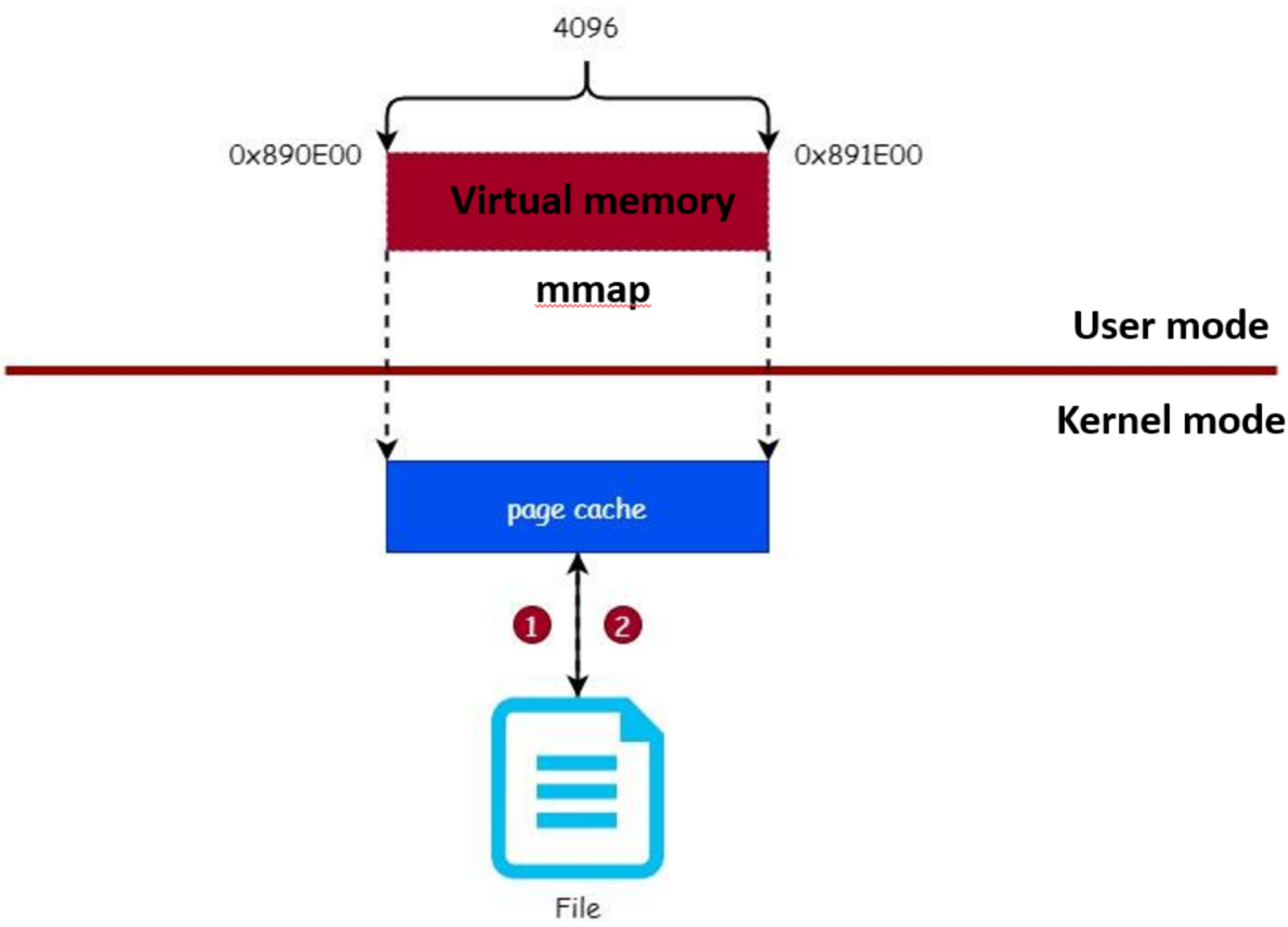
- If the virtual memory sub-system is integrated with the file-system, it enables a simple and efficient mechanism to load programs and data into memory

- If disk I/O requires the transfer of **large amounts of data (one or more pages)**, mmap significantly speeds up I/O by mapping a disk file directly into user-space memory
  - It does not suffer the overhead of syscalls like read/write
  - User-process has direct access to kernel disk cache

System Interface Read/write (for each process):



MMap (can be shared by multiple processes):




```
void *mmap(void addr[.length], size_t length, int prot, int flags,int fd, off_t
 offset);
int munmap(void addr[.length], size_t length);
```

The starting address for the new mapping is specified in *addr*. The *length* argument specifies the length of the mapping (which must be greater than 0).

For details of arguments, see mmap(2) - Linux manual page

Process

```
pid_t fork(void);
pid_t wait(int *_Nullable wstatus);
pid_t waitpid(pid_t pid, int *_Nullable wstatus, int options);
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

Why create a new process?

- Scenario 1: Program wants to run an additional instance of itself
    - E.g., web server receives request; creates additional instance of itself to handle the request;
    - **Original instance continues listening for requests..**
- Scenario 2: Program wants to run a different program
    - E.g., shell receives a command; creates an additional instance of itself;
    - additional instance overwrites itself with requested program to handle command;
    - **Original instance continues listening for commands...**
- How to create a new process? :
    - A "parent" process forks a "child" process;
    - (Optionally) child process overwrite itself with a new program.

fork()

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

fork() creates a new process by duplicating the calling process.
The new process is referred to as the *child* process. The calling process is referred to as the *parent* process.
The child process and the parent process run in separate memory spaces. At the time of **fork**() both memory spaces have the same content.

Memory writes, file mappings (mmap(2)), and unmappings (munmap(2)) performed by one of the processes do not affect the other.
The child process is an exact duplicate of the parent process except for the following points:

- The child has its own unique process ID, and this PID does not match the ID of any existing process group (setpgid(2)) or session.
- The child's parent process ID is the same as the parent's process ID.
- The child does not inherit its parent's memory locks (mlock(2), mlockall(2)).

For more details, see fork(2) - Linux manual page

wait()

```
pid_t wait(int *_Nullable wstatus);
```

All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed.

A state change is considered to be:

- the child terminated;
    - In the case of a terminated child, **performing a wait allows the system to release the resources associated with the child**; if a wait is not performed, then the terminated child remains in a "zombie" state.
    - What if do no 'wait'?
        - A child that terminates, but has not been waited for becomes a **"zombie". (We'll discuss this later).**
        - The kernel maintains a minimal set of information about the zombie process (PID, termination status, resource usage information) in order to allow the parent to later perform a wait to obtain information about the child. **As long as a zombie is not removed from the system via a wait, it will consume a slot in the kernel process table, and if this table fills, it will not be possible to create further processes.**

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

wait() and waitpid()

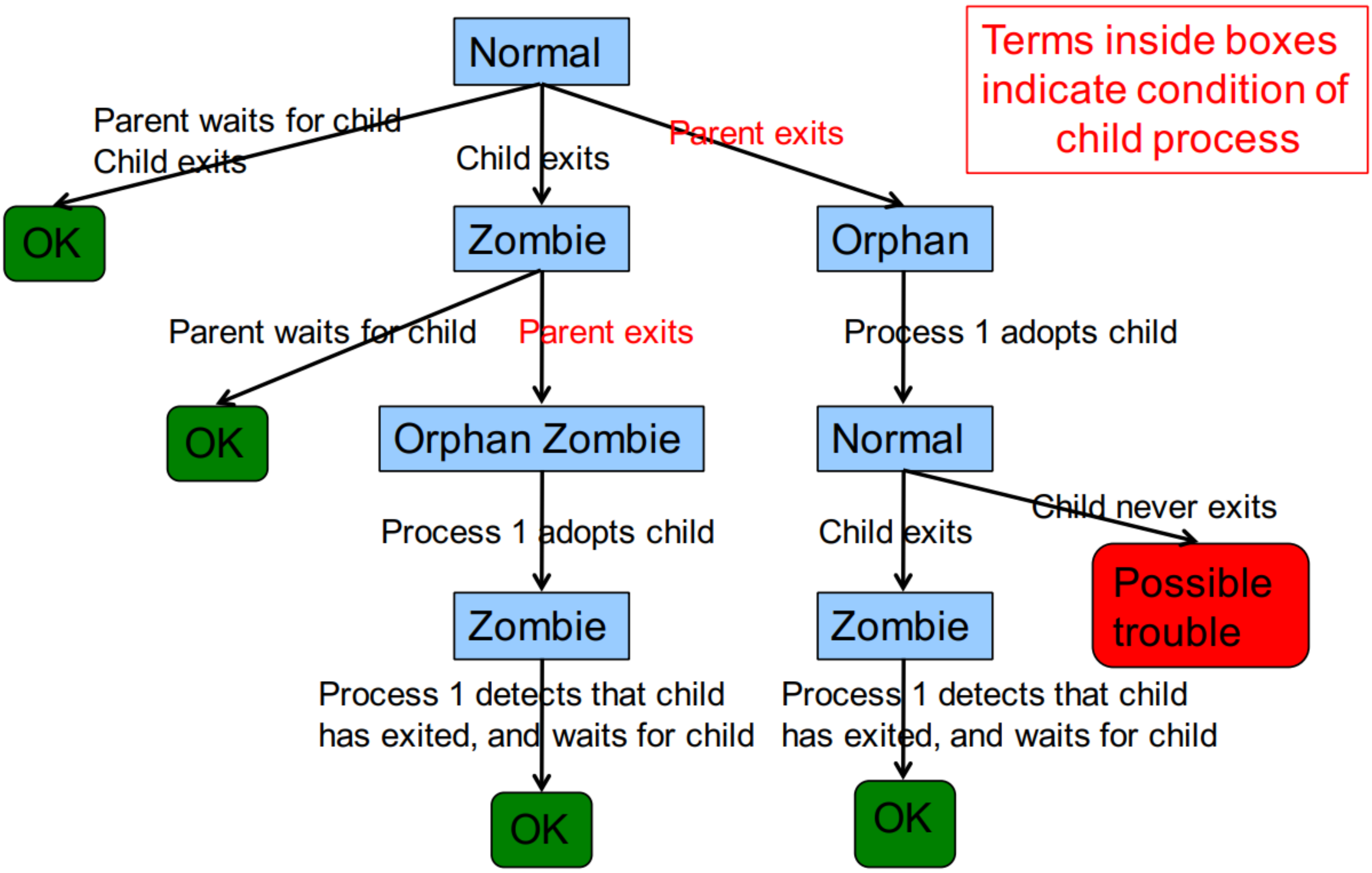
The wait() system call suspends execution of the calling thread until one of its children terminates. The call *wait(&wstatus)* is equivalent to: waitpid(-1, &wstatus, 0); The waitpid() system call suspends execution of the calling thread until a child specified by *pid* argument has changed state.

If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes shall be assigned a new parent process ID corresponding to an implementation-defined system process.

What happens if parent process does not wait for (reap/harvest) child process?

- In shell, could cause sequencing problems

        - E.g, parent process running shell writes prompt for next command before current command is finished executing
    - In general, child process becomes zombie and/or orphan
        - Orphan: A process that has no parent
        - Zombie: A process that has terminated but has not been waited for (reaped)
    - Orphans and zombies would:
        - Clutter Unix data structures unnecessarily;
        - OS maintains unnecessary PCBs
        - Can become long-running processes
        - Consume CPU time unnecessarily



Due to scheduling conflicts, **the onsite/offline tutorial for the upcoming third week will be canceled.**

Instead, the Teaching Assistant (TA) will **upload tutorial videos and materials to Blackboard (BB) during that period.**

Should you have any questions, please feel free to reach out.