Operating System Project System Calls

POORIA NAJI ZAVAR GARMROUDI

S300006

PAYAMREZA POURREZA

S299859

MINA SAMADIAN

S289898

PROFESSOR: STEFANO DI CARLO

CONTENT

- DEFINITION OF SYSTEM CALL

SYSTEM CALL

System calls are implemented as a mechanism for user programs to request services from the operating system kernel.
System calls allow user programs to interact with the underlying operating system and perform various tasks such as file operations, process management, and I/O operations.
In OS/161, the system call interface is defined by a set of functions that user programs can invoke to make requests to the kernel.
These functions are typically provided as part of a C library that is linked with user programs.

SOME EXAMPLES OF SYSTEM CALLS IN DIFFERENT CATEGORIES:

1. File System Operations:
2. Process Control:
3. Memory Management:
4. Network Operations:
5. Device I/O:

THE “ getpid “ SYSTEM CALL

- Responsible for retrieving the process ID (PID) of the current process. The PID is a unique identifier assigned by the operating system to each running process.
- The getpid system call plays a crucial role in process identification and process-specific operations. It provides a way for processes to obtain their unique identifier and utilize it for various purposes, such as interprocess communication, process management, and accessing process-specific attributes.

THE “getpid” SYSTEM CALL

- Purpose: the primary purpose of getpid is to allow processes to identify themselves and perform process-specific operations. By retrieving the PID, a process can distinguish itself from other processes and use this information for various purposes.
- Process identification: the pid serves as a unique identifier for a process within the operating system. It distinguishes one process from another, enabling the operating system and other processes to interact with or manipulate a specific process.
- Interprocess communication: processes often need to communicate or coordinate with other processes. The PID obtained from getpid is used to establish communication channels, send signals, or perform other interprocess communication mechanisms.

THE “getpid” SYSTEM CALL

- Process-Specific Operations: Various operations may require process-specific information. For example, a process may need to access or modify its own attributes, such as changing the working directory, terminating itself, or monitoring its resource usage. The PID obtained through getpid is used as a reference to perform such process-specific operations.
- Parent-Child Relationships: When a process forks and creates a child process, the child process receives a new PID, which is distinct from the parent's PID. The parent process can use getpid to obtain its own PID and fork to retrieve the child's PID. This way, the parent-child relationship can be established and maintained.

THE “write” SYSTEM CALL

The “write” system call is a fundamental component of an operating system's input/output (I/O) functionality. It enables a process to write data to a file or an output device. Understanding and correctly implementing the write system call is essential for handling output operations in an operating system.

THE “write” SYSTEM CALL

- Purpose: The primary purpose of the write system call is to allow processes to output or write data to a file or an output device, such as the console, printer, network socket, or any other writable resource.
- Syntax: The write system call typically takes three arguments:
  - File descriptor: A unique identifier representing the file or output device.
  - Buffer: A pointer to the data that the process wants to write.
  - Size: The number of bytes to write from the buffer.

THE “write” SYSTEM CALL

- Writing Data: When a process invokes the write system call, it requests the operating system to write data from the buffer to the specified file or output device. The operating system handles the low-level details of managing the I/O operation, including buffering, data transfer, and synchronization.

- Return Value: Upon completion, the write system call returns the number of bytes successfully written to the file or output device. If an error occurs, a negative value is returned, indicating the type of error encountered.

THE “write” SYSTEM CALL

Error Handling: The write system call can encounter various errors, such as a full disk, write permission issues, or a disconnected network socket. It is crucial to handle these errors appropriately to ensure the reliability of the output operations.

Asynchronous Operations: Depending on the operating system, the write system call may support synchronous or asynchronous behavior. In synchronous mode, the process is blocked until the write operation is completed. In asynchronous mode, the process can continue execution while the write operation is being handled by the operating system.

THE “write” SYSTEM CALL

- File Descriptor (fd):
  - The fd parameter represents the file descriptor, which is a unique identifier associated with an open file or output device. It specifies the destination where the data should be written.The file descriptor can be a standard input/output/error (stdin/stdout/stderr) or a custom file descriptor obtained through previous calls such as open or socket.
- Buf and buflen:
  - The buf parameter is a user pointer that points to the data buffer from which the write system call retrieves the data to be written.The buflen parameter indicates the size of the data buffer, specifying the number of bytes to be written to the file or output device.

THE “write” SYSTEM CALL

- Error Handling:
  - The sys_write system call should handle various error conditions that may occur during the write operation.
  - Common errors include insufficient disk space, write permission denied, a disconnected network socket, or invalid file descriptors.
  - The function should appropriately return an error code or indicate an error condition to the caller.

- File Descriptor Validation:
  - Before performing the write operation, the sys_write system call should validate the provided file descriptor (fd) to ensure it is a valid and open file descriptor.
  - This step prevents writing to invalid or closed file descriptors and helps maintain the integrity of the system's file and I/O operations.

THE “write” SYSTEM CALL

- Buffer Validation:
  - The buf and buflen parameters need to be validated to ensure they are valid and accessible memory regions.
  - This validation prevents issues such as buffer overflows, accessing invalid memory, or writing to read-only memory regions.

- User I/O Setup:
  - The sys_write system call may need to set up additional configurations or perform specific operations to handle user I/O. For example, it might involve setting up buffers, managing I/O modes, or handling encryption/decryption if required.

THE “write” SYSTEM CALL

- Post-Write Updates:
  - After completing the write operation, the sys_write system call may need to update internal data structures, such as file pointers or file size information.
  - These updates ensure the system maintains accurate bookkeeping and enables subsequent operations on the written data.

The sys_write system call is responsible for handling the writing of data from a buffer to a specified file or output device. It involves validating input parameters, performing the write operation, handling errors, and updating relevant data structures.

THE “read” SYSTEM CALL

The read system call is a crucial component of an operating system's file handling mechanism.

- Purpose: The read system call allows a process to read data from a file specified by a file descriptor. It is an essential mechanism for input operations in an operating system.
- Function Signature: The function signature of the read system call may vary depending on the operating system and programming language, but it generally includes the following parameters:

THE “read” SYSTEM CALL

- File Descriptor (fd): The file descriptor representing the file from which data will be read.
- Buffer (buf) and buflen: The buffer where the read data will be stored, and the length of the buffer indicating the maximum number of bytes to read.
- Error Handling: The read system call handles various error conditions that may occur during the read operation. Common errors include reaching the end of the file, reading from a closed file descriptor, or permission-related issues. Error codes or indicators are returned to the caller to indicate any failures during the process.

THE “read” SYSTEM CALL

- Kernel Buffer Allocation: When a process invokes the read system call, the operating system typically allocates a kernel buffer to temporarily store the data read from the file. This buffer ensures efficient and secure data transfer between the file and the user buffer.
- Read Setup: The read system call sets up the necessary data structures and configurations to perform the read operation. This includes maintaining the file offset to keep track of the current position from where data should be read.

THE “read” SYSTEM CALL

- Data Reading: The read system call reads the requested data from the file into the allocated kernel buffer. It involves low-level operations, such as accessing the file system, fetching the data, and transferring it to the kernel buffer.
- Bytes Read Calculation: After reading the data into the kernel buffer, the read system call calculates the number of bytes successfully read. This value is then returned to the caller to indicate the amount of data retrieved.

THE “read” SYSTEM CALL

- File Offset Update: Once the data is read, the read system call updates the file offset, allowing subsequent read operations to continue from the appropriate position in the file.
- Data Copy to User Buffer: Finally, the read system call copies the requested portion of the data from the kernel buffer to the user buffer (buf) provided by the calling process. This ensures that the process can access the read data in its own memory space.

By implementing the read system call correctly, an operating system ensures proper data reading, error handling, and efficient file handling for processes.

THE “read” SYSTEM CALL

The open system call is indeed a fundamental operation in an operating system that allows processes to open files. It provides access to file resources and enables various file operations. Understanding its inner workings is crucial for proper file management and resource utilization.

- Function Signature: The function signature of the open system call may vary depending on the operating system and programming language, but it generally includes the following parameters:
  - const char *filename: The name or path of the file to be opened.
  - int flags: Flags specifying the mode of opening the file, such as read-only, write-only, or read-write.
  - int32_t *fd: A pointer to an integer where the file descriptor will be stored.

THE “read” SYSTEM CALL

- Error Handling: The open system call handles various error conditions that may occur during the file opening process. Common errors include file not found, insufficient permissions, or invalid flags. Error codes or indicators are returned to the caller to indicate any failures during the process.

- Validating Flags and Filename: Before opening the file, the open system call validates the provided flags and filename. It ensures that the flags are valid and compatible with the file's access mode. Additionally, the system checks if the filename is accessible and exists.

- Opening the File: The open system call opens the specified file using the validated filename and flags. This involves interacting with the file system to obtain the necessary metadata and allocate resources to track the opened file.

THE “read” SYSTEM CALL

- Managing File Table: The operating system maintains a file table or a similar data structure to keep track of open files. The open system call updates this table by creating an entry for the newly opened file, associating it with the file descriptor (fd), and storing relevant information such as the file's current position or permissions.

- Returning the File Descriptor: After successfully opening the file and updating the file table, the open system call returns the file descriptor (fd) to the calling process. The file descriptor is an integer that uniquely identifies the opened file and is used in subsequent file-related operations.

By implementing the open system call correctly, an operating system enables processes to open files, gain access to file resources, and perform various file operations such as reading, writing, or seeking within the file. Proper error handling, validation, and file table management are essential for efficient file handling and resource utilization.

THE “close” SYSTEM CALL

The close system call is a crucial operation for managing file descriptors in an operating system. It allows a process to release a file descriptor and free associated resources.

- Function Signature: The function signature of the close system call may vary depending on the operating system and programming language, but it generally includes the following parameters:
  - int fd: The file descriptor to be closed.
  - int32_t *result: A pointer to an integer where the result or status of the close operation will be stored.

THE “close” SYSTEM CALL

- File Descriptor Validation: Before performing the close operation, the close system call typically validates the provided file descriptor (fd) to ensure it is a valid and open file descriptor. This step helps maintain the integrity of the system's file handling and prevents closing invalid or closed file descriptors.
- Closing the File Descriptor: Once the file descriptor is validated, the close system call proceeds to close the associated file or resource. This involves releasing any resources associated with the file descriptor, such as file buffers or locks. The operating system updates its internal data structures to reflect the closed file descriptor.
- Result or Status: After the close operation is completed, the close system call stores the result or status of the operation in the result variable pointed to by the caller. This status can indicate the success or failure of the close operation, allowing the calling process to handle any errors or confirm the successful release of resources.

THE “close” SYSTEM CALL

- Clearing Filename Buffer: In some cases, the operating system may maintain a filename buffer associated with the file descriptor being closed. The close system call may also involve clearing or resetting this buffer to ensure that sensitive information is not exposed or leaked.

Proper implementation of the close system call is crucial for efficient file descriptor management and resource utilization. By releasing file descriptors and associated resources, the close system call ensures that processes can properly handle file operations, maintain system stability, and avoid resource leaks.

THE “exit” SYSTEM CALL

- The exit system call allows a process to terminate itself.
- It sets the exit status, representing the outcome of the process execution.
- The process is marked as exited and completed.
- The exit system call can perform cleanup and resource management before termination.
- The exit status can be returned to the caller or used for additional functionality.
- Proper implementation of the exit system call ensures proper termination and provides exit status information.

THE “exit” SYSTEM CALL

- The sys_exit system call has a function signature of int sys_exit(int status).
- It is invoked by a process to terminate itself.
- The process sets the exit status, indicating the outcome of its execution.
- Before termination, an assertion check may be performed or resources cleaned up.
- The exit status is stored and can be returned to the caller or used for further processing.
- The sys_exit system call can trigger additional functionality or actions, such as releasing resources or notifying other processes.

THE “getcwd” SYSTEM CALL

The getcwd system call retrieves the current working directory of a process. It allows processes to obtain information about their current location in the file system. By implementing the getcwd system call correctly, processes can retrieve their current working directory path, enabling efficient file path management within the operating system.

THE “getcwd” SYSTEM CALL

The getcwd system call retrieves the current working directory of a process. It allows processes to obtain information about their current location in the file system. By implementing the getcwd system call correctly, processes can retrieve their current working directory path, enabling efficient file path management within the operating system.

- Function Signature: int sys___getcwd(char *buf, size_t buflen, int32_t *retval);

THE “dup2” SYSTEM CALL

dup2 system call is used to duplicate a file descriptor in an operating system.

- It allows a process to associate a new file descriptor with an existing one, pointing to the same file or resource.
- By correctly implementing the sys_dup2 system call, we enable processes to efficiently work with file descriptors.dup2 system call

THE “dup2” SYSTEM CALL

- Function Signature:int sys_dup2(int oldfd, int newfd, int32_t *fd)
- Oldfd
- Newfd
- Fd
- Checking Validity
- Duplication Process

THE “lseek” SYSTEM CALL

The lseek system call is used to change the current position within a file. It allows seeking to a specific position or offset relative to the current position or the end of the file.

- Function Signature: The function signature of the lseek system call may vary depending on the operating system and programming language, but it generally includes the following parameters:
  - int fd: The file descriptor representing the file on which the seek operation will be performed.
  - off_t pos: The position or offset to seek within the file.
  - int whence: The parameter indicating the reference point for the seek operation (current position, start of the file, or end of the file).
  - int32_t *retval: A pointer to an integer that will hold the result or status of the lseek operation.
  - SEEK_CUR: The whence parameter includes different options, such as SEEK_CUR, SEEK_SET, and SEEK_END, which determine the reference point for the seek operation. SEEK_CUR seeks relative to the current position, SEEK_SET seeks relative to the start of the file, and SEEK_END seeks relative to the end of the file.

THE “lseek” SYSTEM CALL

- Seekability Check: Before performing the seek operation, the lseek system call may check if the file is seekable. Some files, such as pipes or sockets, may not support seeking, and the operation would not be allowed.
- File Positioning: The lseek system call determines the new position within the file based on the pos and whence parameters. It calculates the absolute position to which the file pointer should be moved.
- Validity Check: The lseek system call performs a validity check to ensure that the seek operation does not exceed the file size or result in an invalid position within the file.

THE “lseek” SYSTEM CALL

- Update File Position: After performing the validity check, the lseek system call updates the file position, moving the file pointer to the desired position within the file.

- Return Value: The lseek system call updates the retval variable to indicate the success or failure of the seek operation. It may return the new file position or a specific value indicating an error condition.

By correctly implementing the lseek system call, an operating system ensures accurate file positioning and enables seek operations within files.

THE “chdir” SYSTEM CALL

The chdir system call is responsible for changing the current working directory of a process in an operating system. It allows processes to navigate and operate on files and directories effectively.

- Function Signature: The function signature of the chdir system call is typically int sys_chdir(userptr_t pathname). It takes a user-provided pathname as a parameter.
- Pathname: The pathname parameter represents the new directory path to which the process wants to change its current working directory.
- kpath Buffer: The chdir system call typically uses a kernel buffer (kpath buffer) to safely handle the user-provided pathname.

THE “chdir” SYSTEM CALL

- Copying User Pathname: The chdir system call copies the user-provided pathname from the process's address space into the kpath buffer. This step ensures that the kernel can safely access and manipulate the pathname.
- Error Handling: The chdir system call includes error handling to check for possible issues, such as an invalid or inaccessible directory path. It returns an appropriate error code or indicator to the caller if any errors occur during the process
- Cleanup: After successfully changing the current working directory, the chdir system call performs any necessary cleanup operations, such as releasing resources or updating internal data structures related to the process's directory management.

By correctly implementing the chdir system call, an operating system enables processes to change their current working directory, facilitating effective file and directory navigation. Proper error handling and cleanup operations ensure reliable and secure directory management for processes.

THE “execv” SYSTEM CALL

- Execv system call is responsible for executing a new program in the
- Current process.
- It loads the specified program into a new address space, sets up the
- Argument stack, and transfers control to the new program.
- Proper error handling and memory management are crucial for the
- Correct execution of the system call.Execv system call

THE “execv” SYSTEM CALL

- Function signature:int sys_execv(const char *program, char **args,
- Int32_t *retval)
- Program and args
- Create and destroy
- Entrypoint and stackptr
- Process execution process
- Error handling

THE “waitpid” SYSTEM CALL

The waitpid system call is a crucial part of process management in an operating system. It allows a parent process to wait for a specific child process to terminate and obtain its exit status.

- Function Signature: The function signature of the waitpid system call typically includes the following parameters:
  - pid_t pid: The process ID of the specific child process the parent process is waiting for.
  - int *status: A pointer to an integer where the exit status of the child process will be stored.
  - int options: Additional options to control the behavior of the waitpid call.
  - int32_t *retval: A pointer to an integer that will hold the result or status of the waitpid operation.

THE “waitpid” SYSTEM CALL

- Initialization: The waitpid system call initializes the process and resources needed for waiting for the child process to terminate.
- Validate Process ID: The waitpid system call validates the provided process ID (pid) to ensure it corresponds to a valid child process of the current parent process.
- Wait for Child Process to Exit: The waitpid system call causes the parent process to suspend its execution and wait until the specified child process terminates. During this waiting period, the operating system may schedule other processes to execute.

THE “fork” SYSTEM CALL

The fork system call is crucial for process creation in an operating system. It allows a process to create a new child process that is an identical copy of the parent process.

- Function Signature: The fork system call typically has a function signature like int sys_fork(struct trapframe *tf, int32_t *ret). It takes a trapframe pointer and a return value pointer as parameters.

- Source (Parent) and New Process (Child): The fork system call creates a new child process by duplicating the parent process. The child process inherits the same memory contents, file descriptors, and execution state from the parent.

THE “fork” SYSTEM CALL

- Creation Process: The fork system call initiates the process creation by duplicating the parent process's resources and creating a new child process. This includes duplicating memory pages, file descriptors, and other process-related data structures.
- Set the Return Value: After the fork system call completes, it sets the return value to indicate the result of the operation. The return value may differ between the parent and child processes, allowing them to identify their respective process IDs.
- Update the Trapframe: The fork system call updates the trapframe of the child process, reflecting the new process's execution context. This includes updating the program counter, stack pointer, and other relevant registers.

By correctly implementing the fork system call, an operating system enables processes to create new child processes that share the same code and resources, allowing for parallel execution and multitasking.