Answering the Project Questions :

OS project report LAB2

Mohammad Taha Majlesi

Alireza karimi

Pooria Mahdian

Question 1

User-level libraries in xv6 are used to facilitate communication between user programs and the kernel. These libraries include functions that use system calls to access hardware and software resources of the operating system. By analyzing the files specified in the `ULIB` variable in xv6, explain how these libraries utilize system calls. Also, discuss the reasons for using these system calls and their impact on program performance and portability.

Answer:

In xv6, the `ULIB` variable in the `Makefile` typically includes the following user-level library files:

```
ULIB = ulib.o usys.o printf.o umalloc.o
```

These correspond to the source files `ulib.c`, `usys.S`, `printf.c`, and `umalloc.c`. Here's how these libraries utilize system calls:

1. `usys.S`:

    - This assembly file contains the system call stubs for user programs.
    - It defines a series of wrapper functions for system calls using the `SYSCALL` macro.
    - For example, `SYSCALL(fork)` creates a function `fork()` that sets up the system call number in the `eax` register and triggers a software interrupt (`int $64`) to enter kernel mode.
2. **`ulib.c`**:
    - Provides user-level implementations of utility functions.
    - May include wrapper functions that simplify system call usage.
    - Functions in `ulib.c` often call system calls directly or indirectly through the stubs defined in `usys.S`.
3. **`printf.c`**:
    - Implements the `printf` function for formatted output.
    - Uses the `write` system call to output characters to file descriptors.
    - `printf` abstracts away the details of the `write` system call, providing a higher-level interface.
4. **`umalloc.c`**:
    - Provides dynamic memory allocation functions (`malloc`, `free`).
    - Uses the `sbrk` system call to adjust the program's data segment size, requesting more memory from the kernel when needed.

How These Libraries Utilize System Calls:

- **Abstraction**: The libraries provide higher-level functions that abstract the low-level details of system calls. For example, instead of directly invoking `int $64`, a user program calls `fork()` defined in `usys.S`.
- **Convenience**: They offer convenient interfaces for common tasks, such as printing to the console or allocating memory, which internally use system calls.
- **System Call Wrappers**: By defining wrapper functions, the libraries manage the setup of system call numbers and arguments, reducing the burden on the programmer.

Reasons for Using System Calls and Their Impact:

- **Access to Kernel Services**: System calls are the primary mechanism for user programs to request services from the kernel, such as I/O operations, process management, and memory allocation.
- **Performance Impact**:
  - **Overhead**: Each system call involves a context switch from user mode to kernel mode, which introduces overhead.
  - **Optimization**: By using efficient wrapper functions, the libraries can minimize this overhead and improve performance.
- **Portability Impact**:
  - **Abstraction Layer**: The libraries abstract away architecture-specific details, such as system call numbers and calling conventions.
  - **Maintainability**: Changes to system call implementations or numbers only require updates in the library, not in every user program.
  - **Portability**: Programs become more portable across different architectures and versions of the operating system.

In summary, the user-level libraries in xv6 utilize system calls by providing abstraction layers and wrapper functions that simplify interactions with the kernel. This approach improves program portability, maintainability, and can positively impact performance by optimizing system call usage.

Question 2

System calls are not the only method for user programs to interact with the kernel. What other methods exist in Linux that allow user-level programs to access the kernel? Briefly explain each of these methods.

Answer:

In Linux, user programs can interact with the kernel through several mechanisms besides system calls:

1. **Procfs and Sysfs (Pseudo-filesystems)**:
   - `/proc` **Filesystem (procfs)**:

    - A virtual filesystem that presents information about processes and other system information in a hierarchical file-like structure.
    - Programs can read files like `/proc/cpuinfo` or `/proc/meminfo` to get system information.
  - `/sys` **Filesystem (sysfs)**:
    - Provides a view of the kernel's device model.
    - Used to export information about devices and drivers from the kernel to user space.
2. `ioctl` **Calls**:
   - Although `ioctl` is a system call, it allows for device-specific commands that aren't covered by standard system calls.
   - Used to perform operations on device files that require passing control information or data structures.
3. **Netlink Sockets**:
   - A socket-based IPC mechanism between the kernel and user space.
   - Commonly used for networking-related communication, such as configuring network interfaces or managing routing tables.
4. **Signals**:
   - Asynchronous notifications sent to a process to notify it of events like segmentation faults (`SIGSEGV`), interrupts (`SIGINT`), or termination requests (`SIGTERM`).
   - Processes can define handlers for signals to manage or clean up resources.
5. **Shared Memory and Memory Mapping**:
   - Using `mmap`, processes can map files or devices into memory, facilitating communication between processes or accessing device memory.
   - Shared memory segments allow multiple processes to access the same memory region.
6. `/dev` **Files and Device Nodes**:

- Interaction with hardware devices via device files in `/dev`.
- Reading from or writing to these files communicates with device drivers in the kernel.

7. **Sysctl Interface**:
   - Allows querying and modifying kernel parameters at runtime.
   - Accessible via the `/proc/sys` directory or the `sysctl` command.
8. **FUSE (Filesystem in Userspace)**:
   - Allows user-space programs to implement a fully functional filesystem.
   - The kernel communicates with the user-space filesystem via a device file and messages.
9. **Event Tracing and Monitoring Interfaces**:
   - Tools like `epoll`, `inotify`, and `fanotify` allow user programs to monitor filesystem events or file descriptor readiness.
10. **eBPF (Extended Berkeley Packet Filter)**:
    - A kernel technology that allows user-space applications to execute code in the kernel safely.
    - Used for performance monitoring, networking, and security applications.
11. **Netfilter Queue**:
    - Allows user-space programs to inspect and modify network packets before the kernel processes them.

These methods provide mechanisms for user programs to interact with the kernel beyond traditional system calls, enabling rich and flexible communication for various purposes such as configuration, monitoring, and direct hardware access.

Question 3

Is it possible to activate other traps with the access level `DPL_USER`? Why or why not?

Answer:

In the x86 architecture, each entry in the Interrupt Descriptor Table (IDT) has a Descriptor Privilege Level (DPL) that specifies the minimum privilege level required to invoke that interrupt or trap.

- **Possible to Activate Other Traps with `DPL_USER`**:
  - Technically, it is possible to set the DPL of any IDT entry to `DPL_USER` (privilege level 3), allowing user-mode code to invoke it using the `int` instruction.
- **Why It Is Generally Not Done**:
  - **Security Concerns**: Allowing user-mode code to invoke arbitrary traps or interrupts can compromise system security and stability.
  - **Restricted Access**: Critical system traps and interrupts are intended to be managed solely by the kernel to prevent unauthorized access to privileged operations.
  - **Kernel Integrity**: Exposing sensitive traps to user mode could allow malicious code to interfere with kernel operations or hardware devices.
- **Exception for System Calls**:
  - The system call trap is deliberately set with `DPL_USER` to allow user programs to request services from the kernel in a controlled manner.
  - In xv6, the system call trap (`T_SYSCALL`) has its DPL set to `DPL_USER`, enabling user programs to invoke system calls via `int $64`.

Conclusion:

While it is possible to activate other traps with `DPL_USER` by configuring the IDT entries accordingly, it is not advisable due to security and stability reasons. Only specific traps designed for user interaction, like the system call trap, are exposed to user mode.

Question 4

If the access level does not change, `ss` and `esp` are not pushed onto the stack. Why is this the case?

Answer:

In the x86 architecture, when a trap or interrupt occurs, the processor may need to switch from the current stack to a new stack appropriate for the new privilege level. Here's what happens:

- **Privilege Level Change (e.g., from Ring 3 to Ring 0)**:
  - The processor switches to a stack defined in the Task State Segment (TSS) for the target privilege level.
  - It saves the current stack segment (`ss`) and stack pointer (`esp`) onto the new stack so that it can restore them when returning from the interrupt.
- **No Privilege Level Change**:
  - If the trap occurs within the same privilege level (e.g., from kernel code to kernel code), the processor continues using the current stack.
  - There is no need to switch stacks or save `ss` and `esp` because the stack remains valid and appropriate for the current privilege level.
  - Pushing `ss` and `esp` would be unnecessary and wasteful.

Therefore, when the access level does not change during a trap or interrupt:

- **No Stack Switch**: The processor does not perform a stack switch, so the stack segment and pointer remain unchanged.
- **No Need to Save `ss` and `esp`**: Since the stack context is the same, there is no need to save and restore these registers.
- **Efficiency**: Omitting unnecessary operations improves performance and reduces stack usage.

Question 5

Briefly explain the functions used to access system call parameters. Why does `argptr()` check address ranges? What security issues can arise from exceeding valid address ranges? Provide an example where failing to check ranges in `sys_read()` can cause problems in system execution.

Answer:

Functions Used to Access System Call Parameters:

In xv6, the following functions are used to retrieve system call arguments:

- `argint(int n, int *ip)`:
  - Retrieves the nth 32-bit integer argument from the user stack.
  - Copies the value into the integer pointed to by `ip`.
- `argptr(int n, char **pp, int size)`:
  - Retrieves a pointer (address) argument from the nth position.
  - Ensures that the pointer refers to a valid user-space memory region of the specified size.
  - Stores the pointer in `pp` if valid.
- `argstr(int n, char **pp)`:
  - Retrieves a string argument from the nth position.
  - Validates that the string is null-terminated and resides entirely within the user address space.
  - Stores the pointer to the string in `pp` if valid.

Why Does `argptr()` Check Address Ranges?

- **Memory Safety**:
  - To ensure that the kernel does not read from or write to invalid or kernel memory addresses.
  - Prevents accidental crashes due to invalid memory access.
- **Security**:
  - Protects the kernel from malicious attempts by user programs to access or modify kernel memory.
  - Ensures that user programs cannot exploit system calls to perform unauthorized actions.

Security Issues from Exceeding Valid Address Ranges:

- **Kernel Memory Corruption**:

    - If the kernel reads from or writes to invalid memory addresses, it may corrupt its own data structures.
    - Could lead to system instability or crashes.
- **Unauthorized Access**:
    - A malicious user program could attempt to pass a kernel address as a pointer argument.
    - Without proper checks, the kernel might inadvertently expose sensitive data or alter critical information.

Example in `sys_read()` Without Checking Ranges:

Suppose `sys_read(int fd, void *buf, int n)` is implemented without checking if `buf` points to a valid user-space address.

- **Scenario**:
    - A user program deliberately passes a kernel address or an invalid address as the `buf` argument.
    - The kernel attempts to write data into `buf` without validation.
- **Consequences**:
    - **Kernel Data Overwrite**: Critical kernel data structures might be overwritten, leading to unpredictable behavior.
    - **Security Breach**: Sensitive information could be leaked or modified.
    - **System Crash**: Accessing invalid memory could cause a page fault, crashing the kernel.

Therefore, functions like `argptr()` are essential to validate user-supplied pointers and protect the kernel from potential security vulnerabilities and stability issues.

Implementing System Calls

Below are the implementations of the required system calls along with testing procedures and explanations.

Sending System Call Arguments

Implementing `void create_palindrome(int num)`

Implementation Steps:

1. **Define System Call Number in `syscall.h`:**

   ```
   #define SYS_create_palindrome 22
   ```

2. **Declare System Call in `user.h`:**

   ```
   int create_palindrome(int num);
   ```

3. **Add System Call Stub in `usys.S`:**

   ```
   SYSCALL(create_palindrome)
   ```

4. **Add System Call to `syscall.c`:**

   ```
   extern int sys_create_palindrome(void);

   [SYS_create_palindrome] sys_create_palindrome,
   ```

5. **Implement System Call Handler in `sysproc.c`:**

   ```
   int
   sys_create_palindrome(void)
   {
       int num;

       if (argint(0, &num) < 0)
           return -1;

       int reversed = 0, temp = num;
       while (temp != 0) {
   ```

```
        reversed = reversed * 10 + temp % 10;
        temp /= 10;
    }

    int digits = 0;
    temp = reversed;
    if (temp == 0) {
        digits = 1;
    } else {
        while (temp != 0) {
            temp /= 10;
            digits++;
        }
    }

    int multiplier = 1;
    for (int i = 0; i < digits; i++)
        multiplier *= 10;

    int palindrome = num * multiplier + reversed;

    cprintf("Palindrome: %d\\n", palindrome);

    return 0;
}
```

6. **Rebuild the Kernel**:

```
$ make clean
$ make
```

7. **Write User-Level Program** `test_palindrome.c`:

```
#include "types.h"
#include "stat.h"
```

```
#include "user.h"

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf(2, "Usage: test_palindrome <number>\\n");
        exit();
    }

    int num = atoi(argv[1]);
    create_palindrome(num);

    exit();
}
```

8. **Compile and Test**:
   - Add `_test_palindrome` to `UPROGS` in `Makefile`.
   - Run the test:

     ```
     $ test_palindrome 123
     ```

   - Expected Output:

     ```
     Palindrome: 123321
     ```

---

1. Implementing `int move_file(const char* src_file, const char* dest_dir)`

Implementation Steps:

1. **Define System Call Number in `syscall.h`**:

   ```
   #define SYS_move_file 23
   ```

2. **Declare System Call in `user.h`**:

```
int move_file(const char *src_file, const char *dest_dir);
```

3. **Add System Call Stub in `usys.S`:**

```
SYSCALL(move_file)
```

4. **Add System Call to `syscall.c`:**

```
extern int sys_move_file(void);

[SYS_move_file] sys_move_file,
```

5. **Implement System Call Handler in `sysfile.c`:**

```
int
sys_move_file(void)
{
    char *src_file, *dest_dir;
    char dest_path[MAXPATH];
    char filename[DIRSIZ];

    if (argstr(0, &src_file) < 0 || argstr(1, &dest_dir) <
0)
        return -1;

    if (fetch_filename(src_file, filename) < 0)
        return -1;

    safestrcpy(dest_path, dest_dir, MAXPATH);
    if (dest_path[strlen(dest_path) - 1] != '/')
        strcat(dest_path, "/");
    strcat(dest_path, filename);

    begin_op();
    if (create_link(src_file, dest_path) < 0) {
        end_op();
        return -1;
    }

    if (unlink(src_file) < 0) {
        unlink(dest_path);
        end_op();
        return -1;
    }

    end_op();

    return 0;
}
```

Helper Functions (Add to `sysfile.c`):

```
int fetch_filename(const char *path, char *filename)
{
    char *s;
    s = (char*)path + strlen(path);
    while (s >= path && *s != '/')
        s--;
    s++;
    safestrcpy(filename, s, DIRSIZ);
    return 0;
}

int create_link(const char *old, const char *new)
{
    return sys_link(old, new);
}
```

6. **Rebuild the Kernel**.

7. **Write User-Level Program** `test_move.c`:

```
#include "types.h"
#include "stat.h"
#include "user.h"

int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf(2, "Usage: test_move <source_file> <destina
tion_dir>\\n");
        exit();
    }

    if (move_file(argv[1], argv[2]) < 0) {
        printf(2, "move_file failed\\n");
        exit();
    } else {
        printf(1, "File moved successfully\\n");
    }

    exit();
}
```

8. **Compile and Test**:

- Add `_test_move` to `UPROGS` in `Makefile`.
- Run the test:

```
$ echo 'Hello' > src.txt
$ mkdir dest
$ test_move src.txt dest
```

- Verify that `src.txt` is now in `dest` directory and removed from the current directory.

2. Implementing `int sort_syscalls(int pid)`

Implementation Steps:

1. **Modify Process Structure in `proc.h`:**

```
#define MAX_SYSCALLS 100

struct proc {

    int syscalls[MAX_SYSCALLS];
    int syscall_count;
};
```

2. **Record System Calls in `syscall()` Function in `syscall.c`:**

```
void
syscall(void)
{
    int num;
    struct proc *curproc = myproc();

    num = curproc->tf->eax;
    if (num > 0 && num < NELEM(syscalls) && syscalls[num])
{
        curproc->tf->eax = syscalls[num]();

        if (curproc->syscall_count < MAX_SYSCALLS)
            curproc->syscalls[curproc->syscall_count++] =
num;

    } else {
        cprintf("%d %s: unknown sys call %d\\n", curproc->
pid, curproc->name, num);
        curproc->tf->eax = -1;
    }
}
```

3. **Define System Call Number in `syscall.h`:**

```
#define SYS_sort_syscalls 24
```

4. **Declare System Call in `user.h`:**

```
int sort_syscalls(int pid);
```

5. **Add System Call Stub in `usys.S`:**

```
SYSCALL(sort_syscalls)
```

6. **Add System Call to `syscall.c`:**

```
extern int sys_sort_syscalls(void);

[SYS_sort_syscalls] sys_sort_syscalls,
```

7. **Implement System Call Handler in `sysproc.c`:**

```
int
sys_sort_syscalls(void)
{
    int pid;
    struct proc *p;

    if (argint(0, &pid) < 0)
        return -1;

    acquire(&ptable.lock);
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->pid == pid) {
            int n = p->syscall_count;
            for (int i = 0; i < n - 1; i++) {
                for (int j = 0; j < n - i - 1; j++) {
                    if (p->syscalls[j] > p->syscalls[j +
1]) {
                        int temp = p->syscalls[j];
                        p->syscalls[j] = p->syscalls[j +
1];
                        p->syscalls[j + 1] = temp;
                    }
                }
            }
            release(&ptable.lock);
            return 0;
        }
    }
    release(&ptable.lock);
    return -1;
}
```

8. **Rebuild the Kernel.**
9. **Write User-Level Program** `test_sort_syscalls.c`:

```
#include "types.h"
#include "stat.h"
#include "user.h"

int main(void) {
    int pid = getpid();

    // Make some system calls
    printf(1, "Testing sort_syscalls\\n");
    getpid();
    sleep(10);
    write(1, "Hello\\n", 6);
    // Sort the system calls
    if (sort_syscalls(pid) < 0) {
        printf(2, "Failed to sort system calls\\n");
        exit();
    }

    // Exit
    exit();
}
```

---

3. Implementing `int get_most_invoked_syscall(int pid)`

Implementation Steps:

1. **Define System Call Number in `syscall.h`:**

   ```
   #define SYS_get_most_invoked_syscall 25
   ```

2. **Declare System Call in `user.h`:**

   ```
   int get_most_invoked_syscall(int pid);
   ```

3. **Add System Call Stub in `usys.S`:**

   ```
   SYSCALL(get_most_invoked_syscall)
   ```

4. **Add System Call to `syscall.c`:**

   ```
   extern int sys_get_most_invoked_syscall(void);

   [SYS_get_most_invoked_syscall] sys_get_most_invoked_syscal
   l,
   ```

5. **Implement System Call Handler in `sysproc.c`:**

```
int
sys_get_most_invoked_syscall(void)
{
    int pid;
    struct proc *p;

    if (argint(0, &pid) < 0)
        return -1;

    acquire(&ptable.lock);
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->pid == pid) {
            if (p->syscall_count == 0) {
                cprintf("No system calls invoked by proces
s %d\\n", pid);
                release(&ptable.lock);
                return -1;
            }

            int counts[NELEM(syscalls)] = {0};
            for (int i = 0; i < p->syscall_count; i++) {
                counts[p->syscalls[i]]++;
            }
            int max_syscall = 0;
            int max_count = 0;
            for (int i = 0; i < NELEM(syscalls); i++) {
                if (counts[i] > max_count) {
                    max_syscall = i;
                    max_count = counts[i];
                }
            }
            cprintf("Most invoked system call: %d, invoked
%d times\\n", max_syscall, max_count);
            release(&ptable.lock);
```

```
                return 0;
            }
        }
    release(&ptable.lock);
    cprintf("Process %d not found\\n", pid);
    return -1;
}
```

6. **Rebuild the Kernel**.
7. **Write User-Level Program** `test_get_most_invoked.c`:

```
#include "types.h"
#include "stat.h"
#include "user.h"

int main(void) {
    int pid = getpid();

    printf(1, "Testing get_most_invoked_syscall\\n");
    for (int i = 0; i < 5; i++)
        getpid();
    for (int i = 0; i < 3; i++)
        sleep(1);

    if (get_most_invoked_syscall(pid) < 0) {
        printf(2, "Failed to get most invoked system call
\\n");
        exit();
    }

    exit();
}
```

---

4. Implementing `int list_all_processes()`

Implementation Steps:

1. **Define System Call Number in `syscall.h`:**

```
#define SYS_list_all_processes 26
```

2. **Declare System Call in `user.h`:**

```
int list_all_processes(void);
```

3. **Add System Call Stub in `usys.S`:**

```
SYSCALL(list_all_processes)
```

4. **Add System Call to `syscall.c`:**

```
extern int sys_list_all_processes(void);

[SYS_list_all_processes] sys_list_all_processes,
```

5. **Implement System Call Handler in `sysproc.c`:**

```
int
sys_list_all_processes(void)
{
    struct proc *p;

    acquire(&ptable.lock);
    cprintf("PID\\tSyscall Count\\n");
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->state != UNUSED && p->state != ZOMBIE) {
            cprintf("%d\\t%d\\n", p->pid, p->syscall_coun
t);
        }
    }
    release(&ptable.lock);
    return 0;
}
```

6. **Rebuild the Kernel**.
7. **Write User-Level Program** `test_list_processes.c`:

```
#include "types.h"
#include "stat.h"
#include "user.h"

int main(void) {
    printf(1, "Listing all processes:\\n");
    list_all_processes();
    exit();
}
```

Additional Notes

- **Testing**: Each system call should be thoroughly tested using user-level programs.
- **Error Handling**: Ensure that appropriate error messages are displayed when operations fail.
- **Kernel Synchronization**: When accessing or modifying process tables, ensure that locks are properly acquired and released to prevent race conditions.
- **Code Organization**: Keep the kernel code organized by placing system call handlers in appropriate files (`sysproc.c` for process-related calls, `sysfile.c` for file-related calls).
