

ASSIGNMENT 1 : HANDSHAKE FORK U

Purpose of Assignment 1: Handshake

The primary goal is to master three core operating system concepts:

1. **Process Creation (fork)**: Understanding how a single program splits into two separate execution paths.
2. **Pipes**: Learning how to create unidirectional data channels in the kernel. Because a pipe is one-way, you need **two** pipes to achieve a two-way "handshake."
3. **Synchronization**: Ensuring the parent and child wait for each other appropriately (e.g., the child waits for the parent's data, and the parent waits for the child to finish).

ICT1012 – Operating Systems



xv6labs-w3

Handshake (using system calls: pipe and fork)

TIME	PARENT PROCESS (PID 2)	PIPE CHANNELS	CHILD PROCESS (PID 3)
	1. pipe(p2c) & pipe(c2p)	[][][][]	(Not yet created)
	2. fork()	----->	3. Child created
	4. close unused ends		5. close unused ends
	6. write(p2c[1], "x") ----->	[][x][][]	7. read(p2c[0]) (BLOCKS)
	8. read(c2p[0]) (BLOCKS)	[][][][]	9. read(p2c[0]) (WAKES)
		[][][][]	10. "received x from parent"
		[][][][x] <-----	11. write(c2p[1], "x")
	12. read(c2p[0]) (WAKES)	[][][][]	13. exit(0)
	14. "received x from child"		
	15. wait(0) <-----		
V	16. exit(0)		

SOLUTION Line-by-Line Code Explanation

Here is the breakdown of your `handshake.c` logic:

1. Initial Setup and Argument Check

```
if (argc != 2){
    fprintf(2, "Usage: handshake <byte>\n");
    exit(1);
}
char byte_to_send = argv[1][0]; // Extracts the first character from input
```

- `argc != 2`: Ensures the user provided exactly one argument (the byte to send).

- `argv[1][0]` : Grabs the first character of the first command-line argument to use as our "handshake" token.

2. Creating the Communication Channels

```
int p1[2], p2[2]; // Arrays to hold file descriptors (0=read, 1=write)
pipe(p1);         // Creates Pipe 1 (Parent -> Child)
pipe(p2);         // Creates Pipe 2 (Child -> Parent)
```

- `pipe(p1)` : Creates the first pipe. `p1[0]` is the **read end**, and `p1[1]` is the **write end**.
- **Why two?** Pipes are unidirectional. To have a conversation, you need one pipe for "Parent → Child" (`p1`) and another for "Child → Parent" (`p2`).
- Why `int p1[2], p2[2]` specifically?
 - **The Array Structure:** The `pipe()` system call is designed to return **two** values at once. Since a function in C can't easily return two separate integers, you pass it an array of size 2. The system call then "fills in" those two slots.

3. Splitting the Process

```
int pid = fork();
```

- `fork()` : This is where the magic happens. The OS creates a copy of the current process.
- In the **Child**, `pid` will be `0`.
- In the **Parent**, `pid` will be the actual Process ID of the child.

4. The Child's Logic (`pid == 0`)

```
char buf;
read(p1[0], &buf, 1);
printf("%d: received %c from parent\n", getpid(), buf);
write(p2[1], &buf, 1);
exit(0);
```

- `read(p1[0]...)` : The child pauses here until the parent writes something into pipe 1.
- `printf` : Once it receives the byte, it prints its own PID and the character.
- `write(p2[1]...)` : It sends that same byte back to the parent through the second pipe.

5. The Parent's Logic (`else` block)

```

char buf;
write(p1[1], &byte_to_send, 1);
read(p2[0], &buf, 1);
printf("%d: received %c from child\n", getpid(), buf);
wait(0);
exit(0);

```

- `write(p1[1]...)` : The parent starts the handshake by sending the byte.
- `read(p2[0]...)` : The parent then waits for the child to send the byte back through pipe 2.
- `wait(0)` : This is crucial. It tells the parent to wait for the child to finish exiting so the system can clean up the process entry (preventing "zombie processes").

Overall code

```

#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main (int argc, char *argv[]){
    // check input argument first
    if (argc != 2){
        fprintf(2, "Usage: handshake <byte>\n");
        exit(1);
    }

    char byte_to_send = argv[1][0];
    int p1[2], p2[2];

    // need to create 2 pipes
    pipe(p1);
    pipe(p2);

    int pid = fork();

    if (pid < 0){
        fprintf(2, "fork failed\n");
        exit (1);
    }

    if (pid == 0){
        // child process
        char buf;
        read(p1[0], &buf, 1); // read from the parent
        printf("%d: received %c from parent\n", getpid(), buf);
    }
}

```

```

        write(p2[1], &buf, 1); // sending to parent
        exit(0);
    }
    else{
        // parent process
        char buf;
        write(p1[1], &byte_to_send, 1); // send to child
        read(p2[0], &buf, 1); // read from child
        printf("%d: received %c from child\n", getpid(), buf);
        wait(0); // clean up the child process, reap the child
        exit(0);
    }
}

```

ASSIGNMENT 2: SNIFFER

The Purpose of the "Sniffer" Assignment

The goal of this task is to demonstrate a **Side-Channel Attack** caused by a kernel bug.

In a secure OS, when a process (like `secret`) finishes and frees its memory, the kernel should "zero out" that memory before giving it to a new process (like `sniffer`). This ensures Process B cannot see what Process A was doing.

Because this specific version of xv6 has the `memset` (zeroing) calls removed, it is "leaking" the contents of physical RAM. Your `sniffer` program is designed to exploit this by requesting a large chunk of memory and "sifting" through the leftovers of the previous process to find sensitive information.

In normal xv6:

- When memory is allocated:
`memset(mem, 0, size) --> clears memory`
- When memory is freed:
`memset(page, garbage) --> overwrites old data`

BUT IN THIS LAB:

Those `memset()` calls are REMOVED

Effect:

- Freed memory is NOT erased

- Newly allocated memory may contain OLD DATA
 - Memory reused between processes leaks information
-

WHAT "secret.c" DOES

Process A (victim)

Steps:

1. secret.c gets a string from command line
2. Stores the secret string in its heap memory
3. Exits
4. Memory is freed BUT NOT CLEARED

Example:

```
$ secret ict1012
```

Memory now contains:

"ict1012" (still there, but process is gone)

WHAT "sniffer.c" DOES

Process B (attacker)

Goal:

- Read memory that USED TO belong to secret.c

How?

1. Call sbrk() to allocate heap memory
2. OS gives previously freed pages
3. Because memory is not cleared, old data is still inside
4. sniffer scans memory byte-by-byte
5. Finds printable characters
6. Prints the secret

Example:

```
$ sniffer
ict1012
```

WHY THIS IS A SECURITY VULNERABILITY

This violates:

- Process isolation
- Confidentiality
- Secure memory management

Real-world impact:

- Passwords
- Tokens
- Cryptographic keys
- Personal data

All could leak if OS forgets to wipe memory

WHAT YOU ARE EXPECTED TO DO

YOU MUST:

- Modify ONLY user/sniffer.c
- Do NOT modify kernel or secret.c
- Use sbrk() to allocate memory
- Scan allocated memory
- Print the secret EXACTLY

NOT ALLOWED:

- Kernel changes
 - Hardcoding the secret
 - Communicating directly with secret.c
-

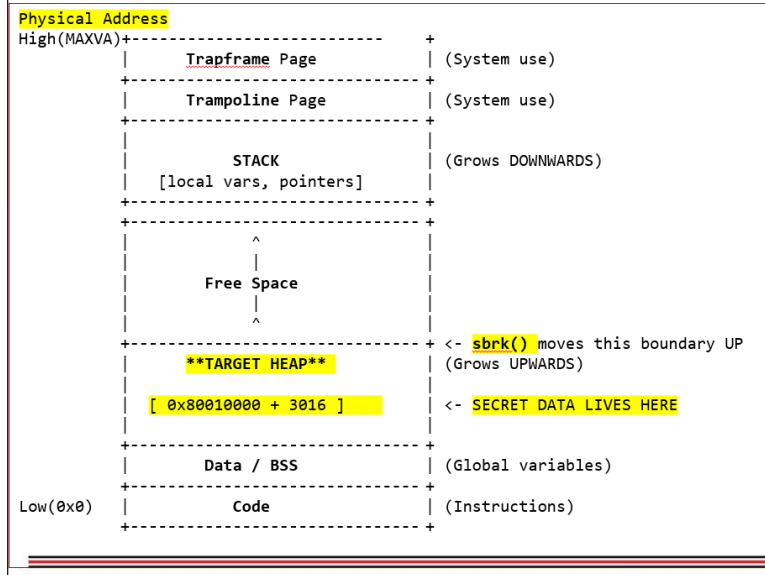
SOLUTION

clues:

ICT1012 – Operating Systems

xv6labs-w3

sniffer



```
#include "kernel/types.h"
#include "kernel/fcntl.h"
#include "user/user.h"
#include "kernel/riscv.h"

#define DATASIZE (8*4096)

char data[DATASIZE];

int
main(int argc, char *argv[])
{
    if(argc != 2){
        printf("Usage: secret the-secret\n");
        exit(1);
    }

    strcpy(data, "this may help.");

    strcpy(data + 16, argv[1]);

    exit(0);
}
```



Specified or
randomly generated

Secret = MyPassword123

VIRTUAL ADDRESS	OFFSET	DATA (Hex)	DATA (ASCII)	ROLE
0x4000 + 0BB8	3000	54 68 69 73	T h i s	ANCHOR START
0x4000 + 0BBC	3004	20 6D 61 79	_ m a y	(Pattern used by
0x4000 + 0BC0	3008	20 68 65 6C	_ h e l	sniffer to scan
0x4000 + 0BC4	3012	70 2E 00 00	p . \0 \0	ANCHOR END
0x4000 + 0BC8	3016	4D 79 50 61	M y P a	SECRET START
0x4000 + 0BCC	3020	73 73 77 6F	s s w o	(The Password/
0x4000 + 0BD0	3024	72 64 31 32	r d 1 2	sensitive
0x4000 + 0BD4	3028	33 00 00 00	3 \0 \0 \0	SECRET END

2. How the Code Works

```
#define SEARCH_SIZE (64*4096)
```

- **What it does:** Defines a search area of 64 pages (each page in xv6/RISC-V is 4096 bytes). (can search larger area like 128 as well)
- **Why:** You need to request enough memory to increase the chances that the kernel gives you the exact physical memory addresses previously used by the `secret` program.

Step 1: Claiming the "Dirty" Memory

```
char *data = sbrk(SEARCH_SIZE);
```

- **What it does:** Calls the `sbrk` system call to expand the heap.
- **The "Trick":** Since the kernel bug prevents zeroing, `data` now points to memory that still contains whatever `secret` wrote there while ago. (as long as the xv6 is still running, the virtual RAM will be there)
- In your xv6 environment (QEMU), the memory holds its state until:
 - **Another process overwrites it:** If you ran 10 other programs between `secret` and `sniffer`, those programs might have claimed that RAM and written their own data

over the secret.

- **The system reboots:** If you close the xv6 shell or restart QEMU, the "virtual" RAM is wiped clean.

Step 2: The Marker (The Anchor)

```
char *marker = "This may help.";
int marker_len = strlen(marker);
```

- **What it does:** Defines a known string to look for.
- **Why:** Searching for a "secret" is hard if you don't know what it looks like. However, if you know the `secret.c` source code, you know it places the secret right next to this "anchor" string. You find the anchor first to find the location of the secret.
 - our anchor string is "This may help."

Step 3: The Search Loop

```
for (int i = 0; i < SEARCH_SIZE - 64; i++) {
    if (memcmp(&data[i], marker, marker_len) == 0) {
```

- **What it does:** It slides through every byte of the newly allocated `data` and compares it to your `marker`.
- **memcmp:** This checks if the memory at the current index `i` matches "This may help".
 - **The Comparison:** The code uses `memcmp(&data[i], marker, marker_len)`. It checks: "Does the memory starting at position `i` look like 'T-h-i-s- -m-a-y-...?'"
 - **The Lock-on:** If the answer is **No**, the loop moves to `i + 1` and checks again. It does this thousands of times.
 - **The Discovery:** When it finally finds the Anchor, the variable `i` represents the **exact starting address** of that "Big Red Statue" in memory
 - **The Calculation:** Now that we know where the statue is (`i`), we use the "map" provided by the `secret.c` source code (which told us the secret is 16 bytes away).
 - **Address of Secret = Address of Anchor (`i`) + 16**

Step 4: Extracting the Prize

```
char *secret_candidate = &data[i + 16];
```

- **What it does:** Once the marker is found at index `i`, it jumps 16 bytes forward.

- **Why 16?** This matches the memory alignment used in the `secret.c` program. It's the "offset" where the actual secret argument is stored.
- The string "This may help." contains **14 characters**. In C, a string also needs a "Null Terminator" (`\0`) to mark the end, bringing the total to **15 bytes**.
 - If the programmer had used `+ 15`, the secret would start at offset **3015**. However:
 - **Alignment:** 3015 is an odd number. CPUs usually prefer even numbers (specifically powers of 2, like 4, 8, or 16).
 - **Predictability:** By jumping 16 bytes, the programmer ensures the secret starts at a nice, "clean" memory address (like `0x4000 + 0x0BC8` in the image).
- In a **64-bit system** (like the version of xv6), the "word size" is **8 bytes**.
- Compilers often "pad" memory to make sure important data starts on an 8-byte boundary.
- By using `+ 16`, the `secret.c` program effectively skipped **two 8-byte blocks** ($2 \times 8 = 16$) to place the secret.

Secret = **MyPassword123** 

VIRTUAL ADDRESS	OFFSET	DATA (Hex)	DATA (ASCII)	ROLE
<code>0x4000 + 0BB8</code>	3000	54 68 69 73	T h i s	ANCHOR START
<code>0x4000 + 0BBC</code>	3004	20 6D 61 79	_ m a y	(Pattern used by
<code>0x4000 + 0BC0</code>	3008	20 68 65 6C	_ h e l	sniffer to scan
<code>0x4000 + 0BC4</code>	3012	70 2E 00 00	p . \0 \0	ANCHOR END
<code>0x4000 + 0BC8</code>	3016	4D 79 50 61	M y P a	SECRET START
<code>0x4000 + 0BCC</code>	3020	73 73 77 6F	s s w o	(The Password/
<code>0x4000 + 0BD0</code>	3024	72 64 31 32	r d 1 2	sensitive
<code>0x4000 + 0BD4</code>	3028	33 00 00 00	3 \0 \0 \0	SECRET END

Step 5: Validation and Output

```
if (secret_candidate[0] >= 'a' && secret_candidate[0] <= 'z') {
    printf("%s\n", secret_candidate);
}
```

```
    exit(0);  
}
```

- **What it does:** It checks if the first character looks like a valid lowercase letter (to avoid printing random binary junk). If it looks correct, it prints the string and terminates.
or

```
if (secret_candidate[0] >= '!' && secret_candidate[0] <= '~') {  
    printf("%s\n", secret_candidate);  
    exit(0);  
}
```

- This broader range is better because it allows the sniffer to detect any printable ASCII character—including uppercase letters, numbers, and symbols—while still effectively filtering out non-human-readable binary "junk" from the dirty memory.

Overall code

```
#include "kernel/types.h"  
#include "user/user.h"  
  
// We search a larger area (64pages to ensure we catch the  
// physical memory used by the previous process.  
#define SEARCH_SIZE (64*4096)  
  
int  
main(int argc, char *argv[])  
{  
    // 1. Allocate memory using sbrk().  
    // Because the lab disables memset() in the kernel, this memory  
    // will contain "dirty" data from previous processes.  
    char *data = sbrk(SEARCH_SIZE);  
  
    // Basic error checking for memory allocation.  
    if (data == (char*)-1) {  
        printf("sbrk failed\n");  
        exit(1);  
    }  
  
    // 2. Define our "anchor" string.  
    // We look for this specific string to find where the secret is.  
    char *marker = "This may help.";  
    int marker_len = strlen(marker);
```

```
// 3. Scan the allocated memory for the marker.
// We stop (marker_len + 64) bytes early to prevent the loop from
// running off the end and causing a page fault crash (scause 0xd).
for (int i = 0; i < SEARCH_SIZE - 64; i++) {

    // Check if the current location in 'data' matches our marker string.
    if (memcmp(&data[i], marker, marker_len) == 0) {
        // 4. Calculate the location of the secret.
        // In secret.c, the secret was copied to (data + 16).
        char *secret_candidate = &data[i + 16];

        // 5. Filter for valid data.
        // A better check: is it a character we can actually see/print?
        if (secret_candidate[0] >= '!' && secret_candidate[0] <= '~') {
            printf("%s\n", secret_candidate);
            exit(0);
        }
    }
}
// Exit gracefully if no secret was found in the allocated memory.
exit(0);
}
```

3. How things are "done" (The Flow)

1. **Run secret ict1012**: The kernel allocates physical RAM, stores "ict1012" and the marker "This may help.", then "frees" the memory when the program ends. But the data **stays** on the RAM chips.
 2. **Run sniffer**: Your program asks for memory. The kernel says, "Here is some RAM I just got back from that last guy."
 3. **Scanning**: Your program treats that RAM like a dusty old book, flipping through pages until it sees the "marker" it recognizes.
 4. **Exfiltration**: Once the marker is found, it reads the bytes immediately following it and prints them to your screen.
-

ASSIGNMENT 3: MONITOR

PURPOSE OF THIS ASSIGNMENT

The goal of this assignment is to add a SYSTEM CALL MONITORING (TRACING) feature into the xv6 kernel.

This helps with:

- Debugging programs
- Understanding how user programs interact with the kernel
- Observing system call behavior at runtime

WHAT THIS ASSIGNMENT TEACHES

1. How USER SPACE communicates with KERNEL SPACE
 - Adding a new system call (monitor)
 - Using `ecall` to switch from user mode to kernel mode
2. How SYSTEM CALLS are implemented in xv6
 - syscall numbers
 - syscall dispatch table
 - `sys_*` kernel functions
3. How the KERNEL tracks PROCESS-SPECIFIC state
 - Each process has its own `monitor_mask`
 - Monitoring affects only the process and its children
4. How to DEBUG kernel behavior safely
 - Print syscall name and return value
 - No need to print arguments (keeps output clean)
5. How PROCESS INHERITANCE works
 - Child processes inherit monitor settings from parent
 - Implemented via copying `monitor_mask` during `fork()`

WHAT THE MONITOR SYSTEM CALL DOES

`monitor(mask):`

- `mask` is a BITMASK
- Each bit corresponds to a system call number
- If a bit is set:
 - > that system call is traced

When a traced system call is ABOUT TO RETURN:

- Print:
PID : syscall name -> return value

EXAMPLE

monitor(1 << SYS_read)

Means:

- Only trace the read system call
-

OBJECTIVE IS TO ACHIEVE:

[illegible]

SIT
SINGAPORE
INSTITUTE OF
TECHNOLOGY

monitoring “fork”

You will get this as the results:

```
3: syscall write -> 1
$ monitor 2147483647 grep hello README
4: syscall monitor -> 0
4: syscall exec -> 3
4: syscall open -> 3
4: syscall read -> 1023
4: syscall read -> 965
4: syscall read -> 437
4: syscall read -> 0
4: syscall close -> 0
$
```

LETS SAY FOR THIS (monitor 2147483647 grep hello README)

```
$ monitor 2147483647 grep hello README
4: syscall monitor -> 0
4: syscall exec -> 3
4: syscall open -> 3
4: syscall read -> 1023
4: syscall read -> 965
4: syscall read -> 437
4: syscall read -> 0
4: syscall close -> 0
```

GENERAL FORMAT FOR EACH MONITOR LINE

```
PID: syscall <name> -> <return value>
```

For example:

```
4: syscall read -> 1023
```

Meaning:

Process ID 4

called the read() system call

and read() returned 1023

SYSCALL MONITOR:

```
4: syscall monitor -> 0
```

- 4 → Process ID (grep process)
- monitor → Your custom system call

- 0 → Return value
monitor() returns 0 on success
Inference:
- Monitor syscall was successfully enabled for this process.

System read

```
4: syscall read -> 1023
```

- read(fd, buf, size)
- Return value = **number of bytes read**
Kernel read **1023 bytes** from README

```
4: syscall read -> 965
```

Read another chunk of **965 bytes**

```
4: syscall read -> 437
```

Read another chunk of **437 bytes**

```
4: syscall read -> 0
```

Important

- read() returning **0** means: END OF FILE (EOF)

Inference:

grep has finished reading README.

```
4: syscall close -> 0
```

- close(fd) returns:
 - 0 → success
 - -1 → failure

File descriptor was closed successfully.

```
Summary:  
monitor -> 0      success
```

exec	-> >=0	success (value not user-meaningful)
open	-> fd	file descriptor number
read	-> N	number of bytes read
read	-> 0	end of file
close	-> 0	success
fork	-> pid	child PID
fork	-> -1	failure

SOLUTION

Phase 1: Plumbing (The Boring Part)

Before the kernel can "trace," it needs to know the `monitor` system call even exists.

1. `kernel/syscall.h`: Add `#define SYS_monitor 22`.
2. `user/user.h`: Add the prototype `int monitor(int);`.
3. `user/usys.pl`: Add `entry("monitor");`.
4. `kernel/syscall.c`:
 - Add `extern uint64 sys_monitor(void);`.
 - Add `[SYS_monitor] sys_monitor` to the `syscalls` array.
 - Add the `syscall_names` array (as shown in your hint) so the kernel can print "fork" instead of just "1".

Phase 2: Storing the State

The kernel needs to remember which process is being watched.

1. `kernel/proc.h`: Inside `struct proc`, add `uint32 monitor_mask;`.
2. `kernel/sysproc.c`: Implement `sys_monitor()`. This function just pulls the integer argument from the user and saves it into the current process's `p->monitor_mask`.

```
uint64
sys_monitor(void){
    int mask;
    argint(0,&mask);
    struct proc *p=myproc();
    p->monitor_mask=(uint32)mask; // This saves the "spy settings" into the
                                // specific process's data structure.
```

Without

```

        // this, the kernel would "forget" what you
        // wanted to monitor as soon as the system
call
        // finished.

    return 0;
}

```

3. `kernel/proc.c`: In the `fork()` function, find where the child process (`np`) is created and add `np->monitor_mask = p->monitor_mask;`. This ensures that if you monitor a shell, everything it runs is also monitored.

1. The `fork()` modification (The "Inheritance" step)

The Point: This is the most "system-oriented" part of the assignment.

- **The Scenario:** When you run `monitor 32 grep ...`, the `monitor` program sets the mask and then starts `grep`. In `xv6`, starting a program usually involves `fork()` (creating a child) and `exec()` (replacing the child's code).
- **Purpose:** If you don't copy the mask in `fork()`, the child process starts with a blank `monitor_mask` (0). Your monitoring would immediately stop before the command even starts running!
- `np->monitor_mask = p->monitor_mask`: This ensures that "monitoring" is a family trait. If the father is being watched, the son is watched too.

As such in `kfork()` variable we:

```

int
kfork(void)
{
    int i, pid;
    struct proc *np;
    struct proc *p = myproc();

    // Allocate process.
    if((np = allocproc()) == 0){
        return -1;
    }

    // Copy user memory from parent to child.
    if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
        freeproc(np);
        release(&np->lock);
        return -1;
    }
    np->sz = p->sz;
    np->monitor_mask = p->monitor_mask; // WE ADD THIS HERE!!
}

```

```

// copy saved user registers.
*(np->trapframe) = *(p->trapframe);

// Cause fork to return 0 in the child.
np->trapframe->a0 = 0;

// increment reference counts on open file descriptors.
for(i = 0; i < NOFILE; i++)
    if(p->ofile[i])
        np->ofile[i] = filedup(p->ofile[i]);
np->cwd = idup(p->cwd);

safestrcpy(np->name, p->name, sizeof(p->name));
pid = np->pid;
release(&np->lock);
acquire(&wait_lock);
np->parent = p;
release(&wait_lock);
acquire(&np->lock);
np->state = RUNNABLE;
release(&np->lock);

return pid;
}

```

Phase 3: The "Spy" Logic

The magic happens in `kernel/syscall.c` inside the `syscall(void)` function. This function is the "bottleneck" that every system call passes through.

Logic to add:

```

void

syscall(void)

{
    int num;
    struct proc *p = myproc();
    num = p->trapframe->a7;

    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        // Use num to lookup the system call function for num, call it,

```

```

// and store its return value in p->trapframe->a0
p->trapframe->a0 = syscalls[num]();

if((p->monitor_mask >> num) & 1){
    // FIX 1: Use syscall_names[num] instead of syscalls[num]
    // FIX 2: Use %ld for the return value (uint64)
    printf("%d: syscall %s -> %ld\n", p->pid, syscall_names[num], p-
>trapframe->a0);

}

} else {
    printf("%d %s: unknown sys call %d\n",
        p->pid, p->name, num);
    p->trapframe->a0 = -1;
}
}

```

Also, The `syscall_names[]` array (The "Translation" step)

The Point: Computers love numbers; humans hate them.

- **Purpose:** If you didn't do this, your output would look like `3: syscall 5 -> 1024`. You wouldn't know if `5` meant `read`, `write`, or `exit`. This array allows the kernel to look up the ID and say "Oh, 5 is 'read'."
- **Crucial Detail:** This maps the index (the ID) directly to a string. `syscall_names[SYS_read]` returns `"read"`.

Hence we add:

```

static const char *syscall_names[] = {
[SYS_fork]      "fork",
[SYS_exit]      "exit",
[SYS_wait]      "wait",
[SYS_pipe]      "pipe",
[SYS_read]      "read",
[SYS_kill]      "kill",
[SYS_exec]      "exec",
[SYS_fstat]     "fstat",
[SYS_chdir]     "chdir",
[SYS_dup]       "dup",
[SYS_getpid]    "getpid",
[SYS_sbrk]      "sbrk",
[SYS_pause]     "pause",
[SYS_uptime]    "uptime",
[SYS_open]      "open",

```

```
[SYS_write]    "write",
[SYS_mknod]    "mknod",
[SYS_unlink]   "unlink",
[SYS_link]     "link",
[SYS_mkdir]    "mkdir",
[SYS_close]    "close",
[SYS_monitor]  "monitor",
};
```

Next, Creating the User Tool

You need to create `user/monitor.c` . This program's job is to:

1. Take the first argument (the mask) and call `monitor(mask)` .
2. Take the remaining arguments and use `exec()` to run them.

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "kernel/fcntl.h"
#include "user/user.h"

int main(int argc, char *argv[]) {
    if(argc < 3){
        fprintf(2, "Usage: monitor mask command [args...]\n");
        exit(1);
    }
    if (monitor(atoi(argv[1])) < 0) {
        fprintf(2, "monitor failed\n");
        exit(1);
    }
    exec(argv[2], &argv[2]);
    exit(0);
}
```