CS 35L- Software Construction Laboratory

Fall 18

TA: Guangyu Zhou

Course Information

- Presentation Schedule

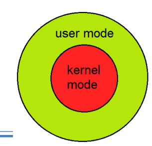
11/7	Daisy Chen		TENGs	Michael Warren		Using photonics for quantum computng	Iris Gur	Karthik Rajagopalan	Mind-Mimicking Machine: The SpiNNaker
11/12						Veteren Day, no class			
11/14	Nihar	Zhenghao Li	Android energy	Varun Narayanan	Varun Pidugu	https://github.com/mit-pdos/biscuit	Jessica Cheng		using neural networks to find medication complaints in social networks

System call programming and debugging

Week 5

Review: Processor Modes

- To understand system calls, first we need to distinguish between **supervisor (kernel) mode** and **user mode** of a CPU
  - **Mode** bit used to distinguish between execution on behalf of OS & behalf of user.
- Modern operating system supports these two modes
  - Supervisor (Kernel) mode: processor executes every instruction in it's hardware repertoire
  - User mode: can only use a subset of instructions



Review: Kernel vs User Mode

Kernel Mode

- When CPU is in **kernel mode**, the code being executed can access any memory address and any hardware resource.
- Hence kernel mode is a very privileged and powerful mode.
- If a program crashes in kernel mode, the entire system will be halted.

User Mode

- When CPU is in **user mode**, the programs don't have direct access to memory and hardware resources.
- In user mode, if any program crashes, only that particular program is halted.
- That means the system will be in a safe state even if a program in user mode crashes.
- Hence, most programs in an OS run in user mode.

Review: The Kernel

- Code of the OS **executes** in **supervisor** state
- Trusted software
  - manages hardware resources (CPU, memory, and I/O)
  - Implements protection mechanisms that could not be changed through actions of untrusted software in user space
- **System call interface** is a safe way to expose privileged functionality

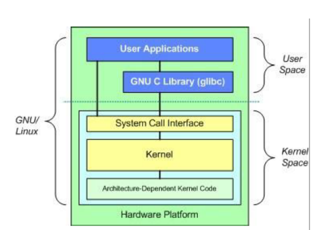
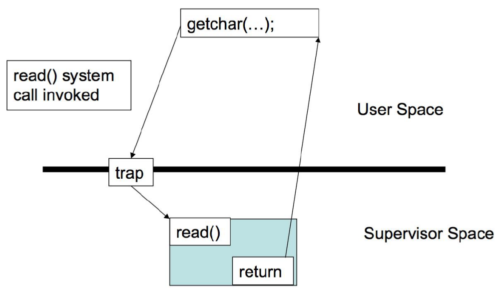


Image by: Tim Jones (IBM)

Reminder of how System calls work



Trap: System call causes a switch from user mode to kernel mode

Reminder of how System calls work

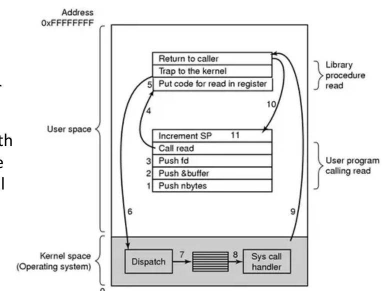
- 1. program get to the system call in the user's code
  int res = sys_call(a_few_params)
- 2. puts the parameters on the stack
- 3. performs a system 'trap' -- hardware switch
  now in system mode
- 4. operating system code may copy large data structures into system memory
- 5. starts operation…

Reminder of how System calls work (cont.)

- 6. operation complete!
- 7. if necessary copies result data structures back to user program's memory
  return to user mode
- 8. user program puts return code into res(the return value from the system call)
- 9. program recommences

More detailed steps for making a system call:

- 1-3: push parameters on stack
- 4: invoke the system call
- 5: put code for system call on register
- 6: trap to the kernel
- 7-10: since a number is associated with each system call, system call interface invokes/dispatch intended system call in OS kernel and return status of the system call and any return value
- 11: increment stack pointer



System calls and Library calls usage

- Library calls
  - executed in the **user program**
  - may perform **several task**
  - may call system calls
- System calls
  - executed by **the operating system**
  - perform simple **single** operations

Types of system calls

System calls can be roughly grouped into five major categories:

1. Process Control
   - load
   - execute
   - end, abort
   - create process (for example, fork on Unix-like systems)
   - terminate process
   - get/set process attributes
   - wait for time, wait event, signal event
   - allocate, free memory
2. File Management
   - create file, delete file
   - open, close
   - read, write, reposition
   - get/set file attributes

Types of system calls

3. Device Management
   - request device, release device
   - read, write, reposition
   - get/set device attributes
   - logically attach or detach devices
4. Information Maintenance
   - get/set time or date
   - get/set system data
   - get/set process, file, or device attributes
5. Communication
   - create, delete communication connection
   - send, receive messages
   - transfer status information
   - attach or detach remote devices

System calls—Examples I

- ssize_t **read**(int fildes, void *buf, size_t nbyte)
  - fildes: file descriptor
  - buf: buffer to write to
  - nbyte: number of bytes to read
- ssize_t **write**(int fildes,const void *buf,size_t nbyte)
  - fildes: file descriptor
  - buf: buffer to write to
  - nbyte: number of bytes to write
- int **open**(const char *pathname,int flags,mode_t mode)
- int **close**(int fd)

System calls—Examples II

- pid_t **getpid** (void)
  - returns the process id of the calling process
- int **dup**(int fd)
  - Duplicates a file descriptor fd. Returns a second file descriptor that points to the same file table entry as fd does.
- int **fstat**(int filedes, struct stat *buf)
- Returns information about the file with the descriptor filedes to buf
- *Why are these system calls and not just regular library functions?*

System calls vs library call conventions

- Library functions often return pointers
  - FILE *fp = fopen("cs35l","r")
  - return NULL for failure
- System calls usually return an integer
  - int res=system_call_function(a_few_args)
  - Where the return value
    - res >= 0 → all is well
    - res < 0 → failure
    - See the global variable errorno for more info

More hints for Assignment 5

- **Lab 5: Programs tr2b and tr2u in 'C':**
  - Take two arguments 'from' and 'to'.
  - Transliterate every **byte** in 'from' to corresponding byte in 'to'
  - Report an error ***from*** and ***to*** are not the same length, or if ***from*** has duplicate bytes

```
./tr2b 'abc' < bigfile.txt => Error
./tr2b 'abc' 'wxyz' < bigfile.txt => Error
./tr2b 'abad' 'wxyz' < bigfile.txt => Error
```

- `tr2b`: **getchar** and **putchar**
- `tr2u`: **read** and **write**
  - ret = read(STDIN_FILENO, *buf*, *nbyte*=1);
  - ret = write(STDOUT_FILENO, *buf*, *nbyte*=1);

The following symbolic constants shall be defined for file streams:
STDIN_FILENO File number of stdin; 0.
STDOUT_FILENO File number of stdout; 1.
STDERR_FILENO File number of stderr; 2.

Homework 5

- Recall Homework 4!
- Rewrite *sfrob* using **system calls** (*sfrobu*)
- *sfrobu* should behave like *sfrob* except
  - If stdin is a regular file, it should initially allocate enough memory to **hold all data in the file all at once**
  - It outputs a line with the number of comparisons performed
- System call functions you'll need: **read, write, and fstat**

Homework 5

- Measure differences in **performance** between *sfrob* and *sfrobu* using the ***time*** command
- Estimate the number of **comparisons** as a function of the number of input lines provided to *sfrobu*
- Write a shell script "*sfrobs*" that uses tr and the sort utility to perform the same overall operation as *sfrob*
- Encrypted input -> tr (decrypt) -> sort (sort decrypted text) -> tr (encrypt) -> encrypted output

More hints for Assignment 5

- **Homework 5: Encrypted sort revisited**
- `int` **`fstat`**`(int filedes, struct stat *buf)`
  - Returns information about the file with the descriptor filedes into buf
- `sfrobu` should behave like `sfrob` except:
  - If stdin is a regular file, it should initially allocate enough memory to hold all data in the file all at once

```
/* File Information */
    ret = fstat(STDIN_FILENO, &buf);
    fileSize = buf.st_size + 1;
/* Setup Initial Buffer */
    input = (char*) malloc(sizeof(char) * fileSize);
```

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for file system I/O */
    blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

Multithreading/Parallel Processing

Week 6

Outline

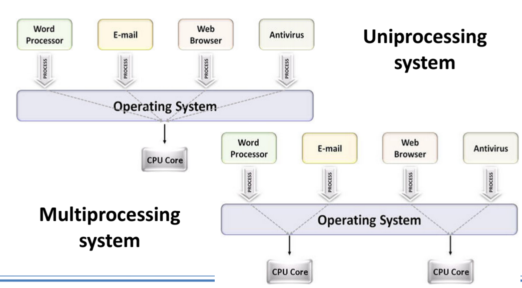
- **Multitasking**
- Multi-Thread Processing
- Synchronization

Multi-tasking (Multi-processing)

- Run multiple processes **simultaneously** to increase performance
- Processes do not share internal structures (stacks, globals, etc)
- Single core: Illusion of parallelism by switching processes quickly (**time-sharing**)
- **Multi-core: True** parallelism. Multiple processes execute **concurrently** on different CPU cores

Multitasking: Single core vs. Multi-core

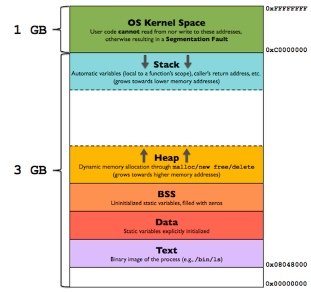
- The use of multiple CPUs/cores to run multiple tasks simultaneously



What is a process, actually?

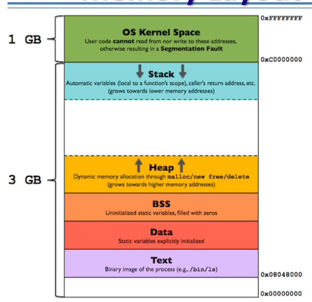
- A process is program that is running. A process is also just one instance of that computer program running.
- Processes have a lot of things at their disposal. At the start of each program you get one process, but each program can make more processes.
- In fact, your operating system starts up with only one process and all other processes are **forked** off of that -- all of that is done under the hood when booting up.

Memory Layout of a process (detailed)



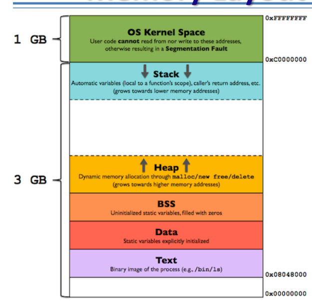
A Stack. The Stack is the place where automatic variable and function call return addresses are stored. Every time a new variable is declared, the program moves the stack pointer down to reserve space for the variable. If the stack grows too far -- meaning that it either grows beyond a preset boundary or intersects the heap -- you will get a stackoverflow most likely resulting in a SEGFAULT or something similar. **The stack is statically allocated by default meaning that there is only a certain amount of space to which one can write**

Memory Layout of a process (detailed)



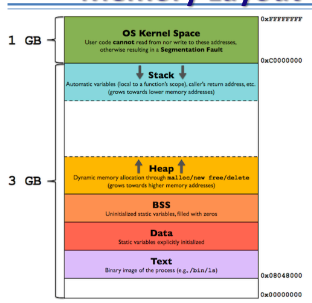
A Heap. The heap is an expanding region of memory. If you want to allocate a large object, it goes here. The heap starts at the top of the text segment and grows upward. One can run out of heap memory if the system is constrained or if you run out of addresses.

Memory Layout of a process (detailed)



A Data Segment This contains all of your globals. This section starts at the end of the text segment and is static in size because the amount of globals is known at compile time.

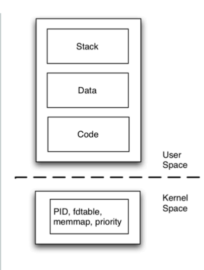
Memory Layout of a process (detailed)



A Text Segment. This is, arguably, the most important section of the address. This is where all your code is stored. Since assembly compiles to 1's and 0's, this is where the 1's and 0's get stored. The program counter moves through this segment executing instructions and moving down the next instruction. **It is important to note that this is the only Executable section of the code**. If you try to change the code while it's running, most likely you will segfault.

Get information about process

- In user space
  - Code (C functions)
  - Data (static and dynamic)
  - Stack
- In kernel space
  - Process Control Block
  - Priority
  - File descriptors
  - Memory map
  - Others



System calls for Processes

- pid_t fork(void)
  - Create a new child process, which is a copy of the current process
  - Parent return value is the PID of the child process
  - Child return value is 0
- int execl(char *name, char *arg0, ..., (char *) 0)
  - Change program image of current process
  - Reset stack and free memory
  - Start at main()
  - Also see other versions: execlp(), execv(), etc.
- pid_t wait(int *status)
  - Wait for a child process (any child) to complete
  - Also see waitpid() to wait for a specific process
- void exit(int status)
  - Terminate the calling process
  - Can also achieve with a return from main()
- int kill(pid_t pid, int sig)
  - Send a signal to a process
  - Send SIGKILL to force termination

Example of process system call

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char **argv)
{
        pid_t id;

        id = fork();

        if (id == 0) {
                /* we are in the child */
                execl("/bin/date", "date", NULL);
                printf("Child: WE DON'T SEE THIS\n");
                exit(0);
        } else {
                /* we are in the parent */
                id = wait(NULL);
                printf("Parent: child terminated\n");
        }

        return 0;
}
```

Ref: https://usfca.instructure.com/courses/1298668/pages/unix-system-calls

Multitasking: Pipeline of commands

- tr -s '[:space:]' '\n' | sort -u | comm -23 – words
- Three separate processes spawned simultaneously
  - P1 – tr
  - P2 – sort
  - P3 - comm
- Common buffers (pipes) exist between 2 processes for communication
  - 'tr' writes its stdout to a buffer that is read by 'sort'
  - 'sort' can execute, as and when data is available in the buffer
  - Similarly, a buffer is used for communicating between 'sort' and 'comm'
- Each process has its own address space
- How do these processes communicate?
  - Pipes/System Calls