Contents

1 Info

- course website: website
- slides: slides
- notes: notes

1.1 grading scheme

- final exam 40%
- quizzes 30%

  - 6 quizzes, with 10 pts each

- assignment 30%

2 Tools

- ltrace: to trace the library call
- strace: to trace the system call
- ldd: to show the library used in the programs
- insmod: to insert program into the kernel
- /dev/null: abandon output
- /dev/full: cannot write
- /dev/random: random file
- pmap: to show the memory information for a specific program

3 Intro

3.1 system call v. library call

- system call is expensive

3.2 requirement of OS

- An operating system should manage resources in a way that avoids shortages or overload conditions
- An operating system should be efficient and introduce little overhead
- An operating system should be robust against malfunctioning application

programs

- Data and programs should be protected against unauthorized access and hardware failures

3.3 services for application

- Loading of programs
- Execution of programs (management of processes)
- High-level input/output operations
- Logical file systems (open(), write(), . . .)
- Control of peripheral devices
- Interprocess communication primitives
- Support of basic communication protocols
- Checkpoint and restart primitives
- . . .

3.4 services for system operations

- User identification and authentication
- Access control mechanisms
- Support for cryptographic operations and the management of keys - Control functions (e.g., forced abort of processes)
- Testing and repair functions (e.g., file systems checks)
- Monitoring functions (observation of system behavior)
- Logging functions (collection of event logs)
- Accounting functions (collection of usage statistics)
  - useful for cloud application
- System generation and system backup functions
- Software management functions
- . . .

3.5 to trace the library call

'ltrace'

- our hello worlds appers to use 'putc', it is done in the compile time

3.6 to trace the system call

'strace'

- it produces write(1, "hello, world")

3.7 to show library used in the problem

'ldd'

- if you use compiler flag -static, it will produces a bigger executable.

3.8 printf v. write

- use printf, it is buffered. Using the buffer we can reduce the # of system call.

3.9 return value of the main function

Should be 0 if success, other for failure

3.10 what happened in the hello world program

- c library: 'printf()', 'puts()', 'fflush(), all this function will make a system call 'write()'
  - **user mode**
- OS kernel: $\text{sys}_{\text{write}}()$
  - **kernel mode** [1]
  - in the kernel, there is a table indexed by the system call number

[1] crossing the boundry between user mode and kernel mode is expensive

3.11 kernel space

- the equivalent of ‘printf‘ is ‘printk‘
- running program in the keneral is different from running program in the user space
- to push a module into the kernel, use ‘sudo insmod hello/hello.ko‘
- kernel have the highest priviledge, running on Ring 0

3.12 types of operating system

3.12.1 Batch Processing Operating Systems

- Characteristics:
  - Batch jobs are processed sequentially from a job queue
  - Job inputs and outputs are saved in files or printed
  - No interaction with the user during the execution of a batch program
- Batch processing operating systems were the early form of operating systems.
- Batch processing functions still exist today, for example to execute jobs on super computers.

3.12.2 General Purpose Operating Systems

- Characteristics:
  - Multiple programs execute simultaneously (multi-programming, multi-tasking)
  - Multiple users can use the system simultaneously (multi-user)
  - Processor time is shared between the running processes (time-sharing)
  - Input/output devices operate concurrently with the processors
  - Network support but no or very limited transparency
- Examples:
  - Linux, BSD, Solaris
  - Windows, MacOS

3.12.3 Parallel Operating Systems

- Characteristics:
  - Support for a very large number of tightly integrated processors(1000s)
  - Symmetrical: Each processor has a full copy of the operating system
  - Asymmetrical: Only one processor carries the full operating system, Other processors are operated by a small operating system stub to transfer code and tasks
- Massively parallel systems are a niche market and hence parallel operating systems are usually very specific to the hardware design and application area.

3.12.4 Distributed Operating Systems

- Characteristics:
  - Support for a medium number of loosely coupled processors
  - Processors execute a small operating system kernel providing essential communication services
  - Other operating system services are distributed over available processors
  - Services can be replicated in order to improve scalability and availability
  - Distribution of tasks and data transparent to users (single system image)
- Examples:
  - Amoeba (Vrije Universiteit Amsterdam)
  - Plan 9 (Bell Labs, AT&T)

3.12.5 Real-time Operating Systems

- Characteristics:
  - Predictability
  - Logical correctness of the offered services

  - Timeliness of the offered services
  - Services are to be delivered not too early, not too late
  - Operating system executes processes to meet time constraints
- Examples:
  - QNX
  - VxWorks
  - RTLinux, RTAI, Xenomai
  - Windows CE

3.12.6 Embedded Operating Systems

- Characteristics:
  - Usually real-time systems, sometimes hard real-time systems
  - Very small memory footprint (even today!)
  - No or limited user interaction
  - 90-95 % of all processors are running embedded operating systems
- Examples:
  - Embedded Linux, Embedded BSD
  - Symbian OS, Windows Mobile, iPhone OS, BlackBerry OS, Palm OS
  - Cisco IOS, JunOS, IronWare, Inferno
  - Contiki, TinyOS, RIOT

3.13 evolution of os

- Vacuum Tubes: no operating system
- Transistors: batch systems automatically process job queues
- Integrated Circuit: Spooling (Simultaneous Peripheral Operation On Line), Multiprogramming and Time-sharing
- VLSI: Personal computer (CP/M, MS-DOS, Windows, Mac OS, Unix), Network operating systems (Unix), Distributed operating systems (Amoeba, Mach, V)

3.14 Monolithic and Modular Operating Systems

- Example: Linux
- Modules can be platform independent
- Easier to maintain and to develop
- Increased reliability / robustness
- All services are in the kernel with the same privilege level
- May reach high efficiency Example: Linux

3.15 Monolithic and Layered Operating Systems

- Easily portable, significantly easier to maintain
- Often reduced efficiency because of the need to go through many layered interfaces
- Rigorous implementation of the stacked virtual machine perspective
- Services offered by the various layers are important for the overall performance
- Example: THE (Dijkstra, 1968)

3.16 Virtual Machines

- Virtualization of the hardware
- Multiple operating systems can execute concurrently
- Separation of multi-programming from other operating system services
- Examples: IBM VM/370 ('79), VMware (1990s), XEN (2003)

3.17 micro kernel

- bring up the driver, just implement the minimum of a kernel

4 Hardware

4.1 Common Computer Architecture

- sequencer
- ALU
- Registers
- buses:
  - control bus
  - address bus
  - data bus

4.1.1 CPU registers

- Typical set of registers:
  - Processor status register
  - Instruction register (current instruction)
  - Program counter (current or next instruction) - Stack pointer (top of stack)
  - Special privileged registers
  - Dedicated registers
  - Universal registers
- Privileged registers are only accessible when the processor is in privileged mode
- Switch from non-privileged to privileged mode via traps or interrupts

4.1.2

4.2 I/O Systems and Interrupts

4.2.1 I/O Devices

- I/O devices are essential for every computer
- Typical classes of I/O devices:

  - clocks, timers
  - user-interface devices
  - document I/O devices (scanner, printer, ...) - audio and video equipment
  - network interfaces
  - mass storage devices
  - sensors and actuators in control applications

- Device drivers are often the biggest component of general purpose operating system kernels

4.2.2 Basic I/O Programming

- **Status driven**: the processor polls an I/O device for information
  - Simple but inefficient use of processor cycles
- **Interrupt driven**: the I/O device issues an interrupt when data is available or an I/O operation has been completed
  - Program controlled: Interrupts are handled by the processor directly
  - Program initiated: Interrupts are handled by a DMA-controller and no processing is performed by the processor (but the DMA transfer might steal some memory access

  cycles, potentially slowing down the processor)
  - Channel program controlled: Interrupts are handled by a dedicated channel device, which is usually itself a micro-processor

4.2.3 Interrupts

- Interrupts can be triggered by hardware and by software
  - clock interrupts
  - keyboard
  - timer
- Interrupt control:
  - grouping of interrupts

  - encoding of interrupts
  - prioritizing interrupts
  - enabling / disabling of interrupt sources
- Interrupt identification:
  - interrupt vectors, interrupt states
- Context switching:
  - mechanisms for CPU state saving and restoring

4.2.4 Interrup Servie Routines

- minimal hardware support (supplied by the CPU)
  - save essential CPU registers
  - jump to the vectorized interrupt servie routine
  - restore essential CPU register to return
- minimal wrapper (supplied by the OS)
  - Save remaining CPU registers
  - Save stack-frame
  - Execute interrupt service code
  - Restore stack-frame
  - Restore CPU registers

4.3 Memory

4.3.1 memory size and access time

- trade-off between memory speed and memory size
  - register is fast, but super small
  - L1, L2, L3 cache increases in size, decrease in speed
  - main memory, big
  - disk

Table 1: Different Segments in Memory

Segments	Description
text	machine instructions
data	static variable and constants, may be further devided into initialized and uninitialized da
heap	dynamically allocated data structures
stack	automatically allocated local variables, management of function calls

4.3.2 memory segments

- Memory used by a program is usually partitioned into different segments that serve different purposes and may have different access rights

4.3.3 stack frame

- arguments pushed onto the stack in a reverse order
- buttom is arguments for the function calls
- then return address(instruction pointer)
- frame pointer
- stack locals

4.3.4 stack smashing attack

- unchecked boundry will lead to stack overflow

4.3.5 caching

- caching is A general technique to speed up memory access by introducing smaller and faster memories which keep a copy of frequently / soon needed data
- cache hit: A memory access which can be served from the cache memory
- cache miss: A memory access which cannot be served from the cache and requires access to slower memory
- cache write through: A memory update which updates the cache entry as well as the slower memory cell

- Delayed write: A memory update which updates the cache entry while the slower memory cell is updated at a later point in time

4.3.6 Locality

- Cache performance is relying on:
  - Spatial locality: Nearby memory cells are likely to be accessed soon
  - Temporal locality: Recently addressed memory cells are likely to be accessed again soon
- Iterative languages generate linear sequences of instructions (spatial locality)
- Functional / declarative languages extensively use recursion (temporal locality)
- CPU time is in general often spend in small loops/iterations (spatial and temporal locality)
- Data structures are organized in compact formats (spatial locality)

5 Processes and Thread

5.1 Fundamental Concepts

5.1.1 Separation of Mechanisms and Policies

- An important design principle is the separation of policy from mechanism.
- Mechanisms determine how to do something.
- Policies decide what will be done.
- The separation of policy and mechanism is important for flexibility, especially since policies are likely to change.

5.1.2 User Mode

- the processor executes machine instructions of (user space) processes;

- the instruction set of the processor is restricted to the so called unprivileged instruction set;
- the set of accessible registers is restricted to the so called unprivileged register set;
- the memory addresses used by a process are typically mapped to physical memory addresses by a memory management unit;
- direct access to hardware components is protected by using hardware protection where possible;
- direct access to the state of other concurrently running processes is restricted.

5.1.3 System Mode

- the processor executes machine instructions of the operating system kernel;
- all instructions of the processor can be used, the so called privileged instruction set;
- all registers are accessible, the so called privileged register set;
- direct access to physical memory addresses and the memory address mapping tables is enabled;
- direct access to the hardware components of the system is enableds;
- the direct manipulation of the state of processes is possible.

5.1.4 Entering the Operating System Kernel

- System calls
  - Synchronous to the running process
  - Parameter transfer via registers, the call stack or a parameter block
- Hardware traps
  - Synchronous to a running process (devision by zero)
  - Forwarded to a process by the operating system

- Hardware interrupts
  - Asynchronous to the running processes
  - Call of an interrupt handler via an interrupt vector
- Software interrupts
  - Asynchronous to the running processes

5.2 Processes

5.2.1 Characteristics

- an instance of a program under execution
- A process uses/owns resources (CPU, memory, file) and is characterized by the following:
  1. A sequence of machine instructions which determines the behavior of the running program (control flow)
  2. The current state of the process given by the content of the processor's registers, the contents of the stack, and the contents of the heap (internal state)
  3. The state of other resources (e.g., open files or network connections, timer, devices) used by the running program (external state)
- Processes are sometimes also called tasks.

5.2.2 State Machine View of Processes

- new: just created
- ready: ready to running
- running: executing
- blocked: not ready to run, wait for resources
- terminated: just finished, not yet removed

5.2.3 Queueing Model View of Processes

- Processes are enqueued if resources are not readily available or if processes wait for events
- Dequeuing strategies have strong performance impact
- Queueing models can be used for performance analysis

5.2.4 Process Control Block

- process are internally represented by a process control block (PCB)
  - process identification
  - process state
  - saved registers during context switches
  - scheduling information
  - assigned memory regions
  - open files or network connections
  - accounting info (# of CPU cycles, IO operations)
  - pointers to other PCBs

5.2.5 Process Lists

- PCBs are often organized in doubly-linked lists or tables
- can be queued by pointer operations
- run queue length of the CUP is a good load indicator
- The system load often defined as the exponentially smoothed average of the run queue length over 1, 5 and 15 minutes
  - usually count when there is a interrupt

5.2.6 Process Creation

- The fork() system call creates a new child process
  - which is an exact copy of the parent process
  - except that the result of the system call differs
- exec() system call replaces the current process image with a new process image

5.2.7 Process Tree

- the first process is created when the system is initialized (init / systemd)
- All other processes are created using fork(), which leads to a process tree
- PCBs often contain pointers to parent PCBs

5.2.8 Process Termination

- Processes can terminate themself by calling exit()
- The wait() system call allows processes to wait for the termination of a child process
- Terminating processes return a numeric result code
- shell application:
  - first fork
  - in the child we execute the operation
  - and the parent wait() for the child

5.2.9 POSIX API (fork, exec)

```
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
pid_t fork(void);
int execve(const char  *filename, char *const argv [],
           char *const envp[]);
extern char **environ;
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const  char  *path,  const  char  *arg, ...,
           char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

5.2.10 POSIX API (wait, exit)

```
#include <stdlib.h>
void exit(int status);
int atexit(void (*function)(void));
#include <unistd.h>
void _exit(int status);
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options); // wait for a process
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>
pid_t wait3(int *status, int options, struct rusage *rusage);
pid_t wait4(pid_t pid, int *status, int options, struct rusage *rusage);
```