National University of Singapore
CS2106 Operating System
Midterm Summary Notes

Dong Shaocong A0148008J

March 15, 2018

1 Basic Idea

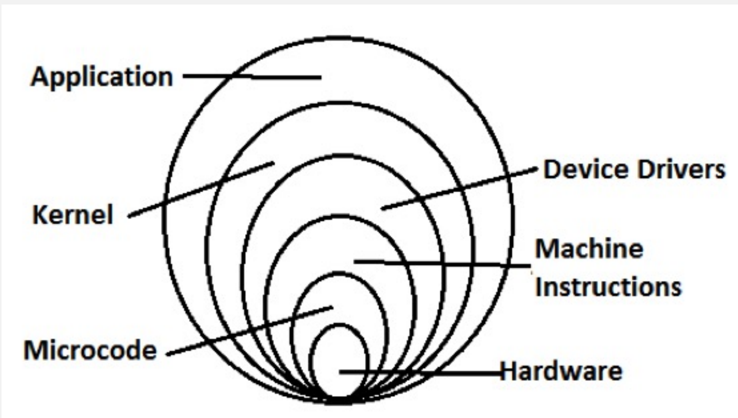
Definition 1.1. Operating System is a suite (i.e. a collection) of specialised software that:

- Gives you access to the hardware devices like disk drives, printers, keyboards and monitors.
- Controls and allocate system resources like memory and processor time.
- Gives you the tools to customise your and tune your system.

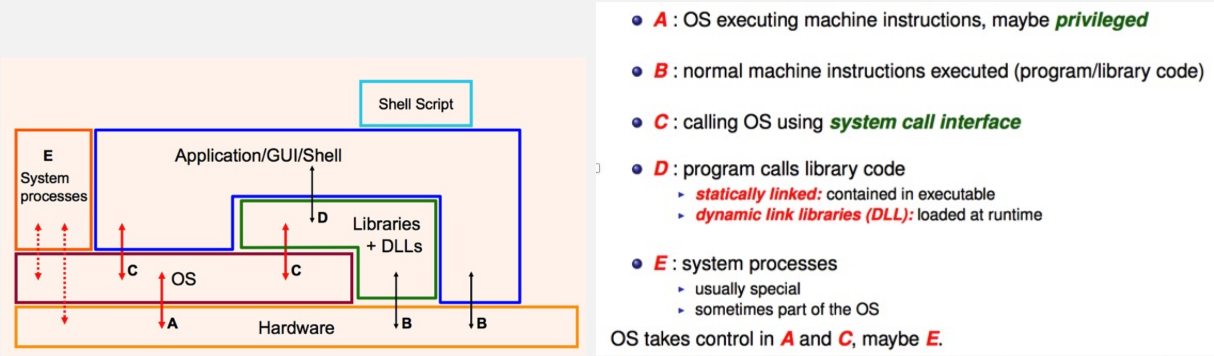
Example 1.1. LINUX, OS X (or MAC OS, a variant of UNIX), Windows 8

What are Operating System? It usually consists of several parts. (Onion Model)

- Bootloader First program run by the system on start-up. Loads remainder of the OS kernel.
  - On Wintel systems this is found in the Master Boot Record (MBR) on the hard disk.
- Kernel The part of the OS that runs almost continuously.
- System Programs Programs provided by the OS to allow:
  - Access to programs.
  - Configuration of the OS.
  - System maintenance, etc.



Abstraction Layer & Opearting System Structure



Definition 1.2. Boostrapping

- The **OS is not present in memory** when a system is cold started.
  - When a system is first started up, memory is completely empty.
- We start first with a **bootloader** to get an operating system into memory.
  - Tiny program in the first (few) sector(s) of the hard-disk.
  - The first sector is generally called the boot sector or master boot record for this reason.
  - Job is to load up the main part of the operating system and start it up.

Definition 1.3. Core CPU units that can execute processes, because we have much more number of processes than the number of cores, we have to do **context switching** to share a core very quickly between different processes.

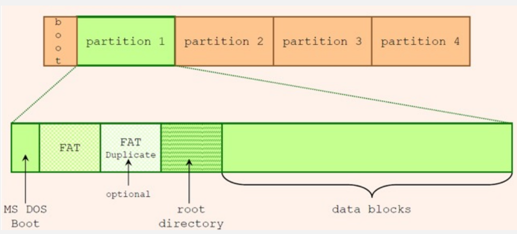
- Entire sharing must be transparent.
- Processes can be suspended and resumed arbitrarily.

Definition 1.4. Context switching

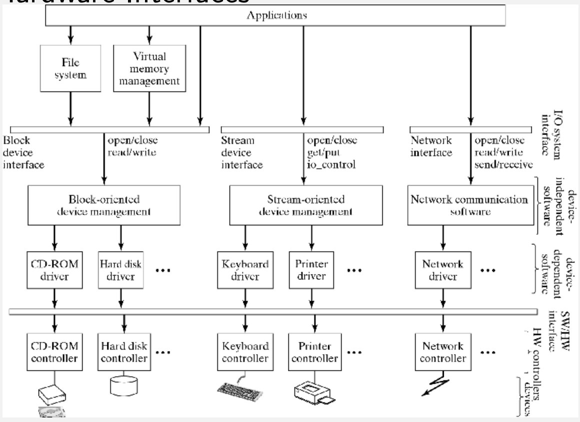
1. Save the context of the process to be suspended.
2. Restore the context of the process to be (re)started.
3. Issues of scheduling to decide which process to run.

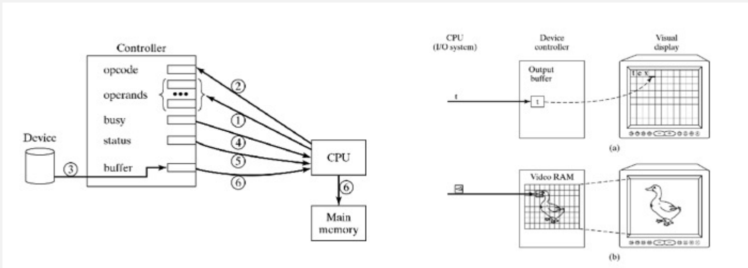
Definition 1.5. File system A set of data structures on disk and within the OS kernel memory to organise persistent data.

How OS file system works?



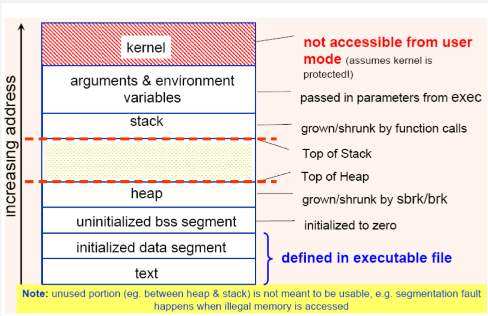
Hardware Interfaces





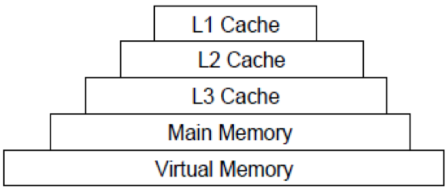
Definition 1.6. Memory static/dynamic (new, delete, malloc, free). Memory to store instructions Memory to store data.

Memory Management



Definition 1.7. Virtual Memory management

- For cost/speed reasons memory is organized in a hierarchy:



- The lowest level is called "virtual memory" and is the slowest but cheapest memory.
  - Actually made using hard-disk space!
  - Allows us to fit much more instructions and data than memory allows!

Definition 1.8. OS security

- Data (files): Encryption techniques, Access control lists
- Resources: Access to the hardware (biometric, passwords, etc), Memory access, File access, etc.

Writing an OS (BSD Unix)

Machine independent

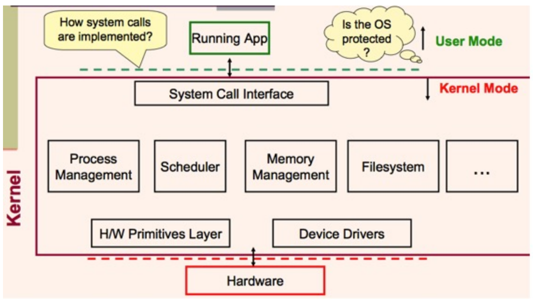
- 162 KLOC
- 80% of kernel
- headers, init, generic interfaces, virtual memory, filesystem, networking+protocols, terminal handling

Machine dependent

- 39 KLOC
- 20% of kernel
- 3 KLOC in asm
- machine dependent headers, device drivers, VM

Definition 1.9. Kernel

- Monolithic Kernel (Linux, MS Windows)
  - All major parts of the OS-devices drivers, file systems, IPC, etc, running in "kernel space" (an elevated execution mode where certain privileged operations are allowed).
  - Bits and pieces of the kernel can be loaded and unloaded at runtime (e.g. using "modprobe" in Linux)



- MicroKernel (Mac OS)
  - Only the "main" part of the kernel is in "kernel space" (Contains the important stuff like the scheduler, process management, memory management, etc.)
  - The other parts of the kernel operate in "user space" as system services: The file systems, USB device drivers, Other device drivers.

External View of an OS

- The kernel itself is not very useful. (Provides key functionality, but need a way to access all this functionality.)
- We need other components:
  - System libraries (e.g. stdio, unistd, etc.)
  - System services (creat, read, write, ioctl, sbrk, etc.)
  - OS Configuration (task manager, setup, etc.)
  - System programs (Xcode, vim, etc.)
  - Shells (bash, X-Win, Windows GUI, etc.)
  - Admin tools (User management, disk optimization, etc.)
  - User applications (Word, Chrome, etc).

Definition 1.10. System Calls calls made to the Application Program Interface or API of the OS.

- UNIX and similar OS mostly follow the POSIX standard. (Based on C. Programs become more portable.) *POSIX: portable operating system interface for UNIX, minimal set of system calls for application portability between variants of UNIX.*
- Windows follows the WinAPI standard. (Windows 7 and earlier provide Win32/Win64, based on C. Windows 8 provide Win32/Win64 (based on C) and WinRT (based on C++).)

Example 1.2. User mode + Kernel mode

- Programs (process) run in user mode.
- During system calls, running kernel code in kernel mode.
- After system call, back to user mode.

> How to switch mode? Use privilege mode to switching instructions:
>
> - syscall instruction
> - software interrupt - instruction which raises specific interrupt from software.

Example 1.3. LINUX system call

- User mode: (outside kernel)
  - C function wrapper (eg. **getpid()**) for every system call in C library.
  - assembler code to setup the system call no, arguments
  - trap to kernel
- Kernel mode: (inside kernel)
  - dispatch to correct routine
  - check arguments for errors (eg. invalid argument, invalid address, security violation)
  - do requested service
  - return from kernel trap to user mode
- User mode: (Outside kernel)
  - returns to C wrapper - check for error return values

2 Process Management

Definition 2.1. Program consists of: Machine instructions (and possibly source code) and Data. A program exists as a file on the disk. (e.g. command.exe, MSword.exe)

Definition 2.2. Process consists of Machine instructions (and possibly source code), Data and Context. It exists as instructions and data in memory, **may** be executing on the CPU.

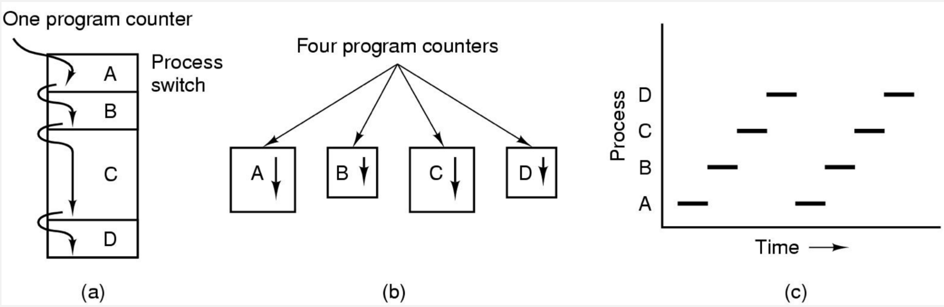
> Program vs. Process
> A single program can produce multiple processes. (e.g. chrome.exe is a single program, but every tab in Chrome is a new process!)

Definition 2.3. Execution Modes

- Programs usually run sequentially. (Each instruction is executed one after the other.)
- Having multiple cores or CPUs allow parallel ("concurrent") execution. (Streams of instructions with no dependencies are allowed to execute together.)
- A multitasking OS allows several programs to run "concurrently". (Interleaving, or time-slicing)

Remark. we mostly assume number of processes $\geq$ number of CPU otherwise can have idle tasks. So each core must still switch between processes even for multi-cores, and we will assume a single processor with a single core.

The Process Model



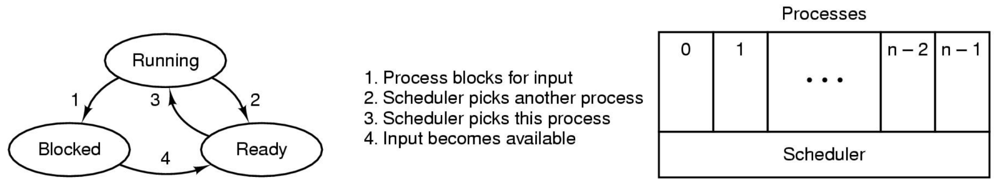
- Figure (b) shows what appears to be happening in a single processor system running multiple processes:
  - There are 4 processes each with its own program counter (PC) and registers.
  - All 4 processes run independently of each other at the same time.
- Figure (a) shows what actually happens.
  - There is only a single PC and a single set of registers.
  - When one process ends, there is a "context switch" or "process switch":
    - PC, all registers and other process data for Process A is copied to memory.
    - PC, register and process data for Process B is loaded and B starts executing, etc.
- Figure (c) illustrates how processes A to D share CPU time.

Definition 2.4. Process States there are three possible states for a process

- Running
  - The process is actually being executed on the CPU.
- Ready
  - The process is ready to run but not currently running.
  - A "scheduling algorithm" is used to pick the next process for running.
- Blocked.
  - The process is waiting for "something" to happen so it is not ready to run yet. e.g. include waiting for inputs from another process.

Definition 2.5. Process Context (values change as a process runs)

- CPU register values.
- Stack pointers.



- CPU Status Word Register
  - This maintains information about whether the previous instruction resulted in an overflow or a ”zero”, whether interrupts are enabled, etc.
  - This is needed for branch instructions assembly equivalents of ”if” statements.

The AVR Status Register – SREG – is defined as:

Bit	7	6	5	4	3	2	1	0	
0x3F (0x5F)	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

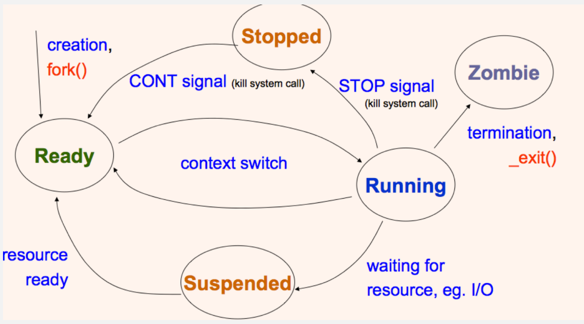
Example 2.1. Context Switching in FreeRTOS Atmega Port FreeRTOS relies on regular interrupts from Timer 0 to switch between tasks. When the interrupt triggers:

1. PC is placed onto Task As stack.
2. The ISR calls portSAVECONTEXT, resulting in Task As context being pushed onto the stack.
3. pxCurrentTCB will also hold SPH/SPL after the context save.
   - This must be saved by the kernel.
   - The kernel stores a copy of the stack pointer for each task.
4. The kernel then selects Task B to run, and copies its SPH/SPL values into pxCurrentTCB and calls portRESTORE_CONTEXT.
5. The rest of portRESTORE_CONTEXT is executed, causing Task Bs data to be loaded into R31-R0 and SREG. Now Task B can resume like as though nothing happened
6. Only Task Bs PC remains on the stack. Now the ISR exits, causing this value to be popped off onto the AVRs PC.
   - PC points to the next instruction to be executed.
   - End result: Task B resumes execution, with all its data and SREG intact!

> How can context switching be triggered?
> It can be triggered by a timer; currently running process waiting for input; currently running task blocking on a synchronisation mechanism; currently running task wants to sleep for a fixed period; higher priority task becoming READY; ...

Definition 2.6. Process Control Block maintains information about that process: Process ID (PID), Stack Pointer, Open files, Pending signals, CPU usage, ...

Process Life Cycle



Definition 2.7. Creating a new process - fork()

- Fork system call creates a new process by duplicating the current image into a new process, *child process*
- same code (executable image) is executed
- Child differs only in process id (PID) and parent (PPID), fork return value
- Data in child is a COPY of the parent (i.e. not shared)
- In PARENT process after fork:
  - PC is at return from fork system call
  - fork return value: new child PID
- In CHILD process after fork:
  - PC is at return from fork system call
  - fork return value: 0
  - Shares open file & signal handlers with parent, current working directory
  - Independent copy of: memory, arguments, environment variables (note: cloning example)
- fork return result is -1 if the fork failed.

Definition 2.8. The Master Process

- Every process has parent: where does it stop?
- Special initial process - init process created in kernel at the end UNIX boot process, traditionally having PID=1.
- Forking creates process tree, init is the root process.
- init watches for processes and response where needed, e.g. terminal login.
- init also manages system run levels (e.g. shutdown, power failure, single-user mode), etc. Example of a system-like process running in kernel mode.

Definition 2.9. Start/Stop a Process

- kill() system call sends signal to process
- Special process signals:
  - stopping process (SIGSTOP)
  - killing process (SIGKILL)
  - restart stopped process (SIGCONT)

Terminating a process

- system call: `void _exit(int status)`
- _exit system call used for immediate voluntary termination of process (never returns!).
- Closes all open file descriptors; children processes are inherited by init process;
- parent sent `SIGCHLD` signal (see later section on signals & IPC)
- status returned to parent using wait()
- Usually status is used to indicate errors, eg. convention is

  _exit(0) for success, 0 means no error

  _exit(1) for error, positive number for error number

- Process finished execution
- can release **most** system resources used by process are released on exit
- **BUT** some basic process resources not releasable: PID & status needed when `wait()` is called. Also process accounting info, e.g. cpu time. Note: may mean that process table entry still being used

Notes: See `wait()`, zombies

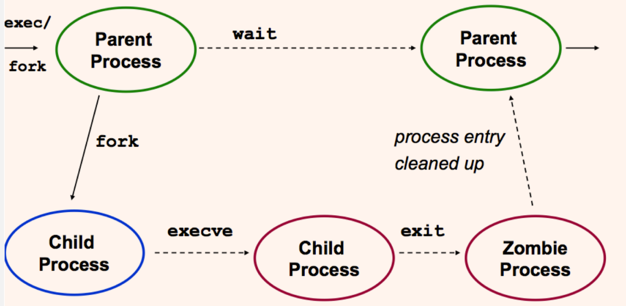
Definition 2.10. Normal Program Termination - void exit(int status) from standard C library function

- Usuall don't use _exit() but exit(), which cleans up: open streams from C stdio library (e.g., fopen, printf) are flushed and closed
- calls some exit handlers
- finally calls _exit(status) after all standard C cleanup done.

Remark. returning from main() implicitly calls exit. exec didn't actually call main directly but a startup routine. Open files also get flushed automatically.

Waiting for Child Processes to Terminate - process interaction

- system call: `pid_t wait(int *status)`
- Parent can wait until some child processes terminates (calls `exit`)
- Note: also some other conditions
- `status` gives child exit status
- cleans up remainder of child system resources – ones not removed when exit!
- Other versions of wait: (some are non blocking!) `waitpid(), wait3(), wait4()`



Note: example uses one ordering of execution, others are possible!

Zombie Process

- process is "dead" / terminated
- Recall: `wait()` means that process termination is not complete when it exits
- process goes to **zombie state**: remainder of process data structure **cleaned up** when `wait()` happens (if it happens!)
- **can't delete** process since dont know if wait from parent needs exited process info! (so it's a consequence of having a wait() operation defined!)
- **cannot kill** zombie since already exited!

2 cases:

- Parent dies before child, the `init` process becomes "pseudo" parent of child processes. Child dying sends signal to `init` process, which organizes to call `wait()` to cleanup process
- Child dies before parent but parent didn't call `wait`. Becomes a zombie process. Can fillup process table requiring reboot. Unix SVR4 can specify no zombie creation on termination
- modern Unixes have mechanisms to avoid zombies

3 Process Scheduling