Outline

1. Announcements
2. Kernel Types
3. Simple OS Examples
4. System Calls
   - System Call Overview
   - Types of System Calls – §2.4, SGG
Announcements

- Week03 lab
- `strace`: program to watch system calls
- C programming tute
- ICT Learning Centre C programming intro.
The OS Kernel

- The **kernel** is the core of an OS
- It has complete control over everything that occurs in the system
- **Kernel** overview and comparison

(courtesy of Wikipedia)
Monolithic Kernel

- Older monolithic kernels were written as a mixture of everything the OS needed, without much of a structure. The monolithic kernel offers everything the OS needs: 
  - memory management, 
  - multiprogramming, interprocess communication (IPC), 
  - file systems, network protocols, etc.

- Newer monolithic kernels have a modular design, which offers run-time adding and removal of services. The whole kernel runs in "kernel mode", a processor mode in which the software has full control over the machine. The processes running on top of the kernel run in "user mode", in which programs have only access to the kernel services.

- Modern OS with monolithic kernels: Linux, FreeBSD, NetBSD, Solaris
Microkernel

- Moves as much from the kernel into “user” space to run as processes
- Communication takes place between user modules using

**Benefits**
- Easier to extend a microkernel
- Easier to port the operating system to new architectures
- More reliable (less code is running in kernel mode)
- More secure

**Disadvantages**
- Performance overhead of user space to kernel space communication

**Example microkernels:** L4, Minix, QNX
Monolithic vs. Microkernel System Structure

Monolithic Kernel

Microkernel
Hybrid Kernels

- Similar to microkernels, except that they include additional code in kernel space so that such code can run more swiftly than it would were it in user space.
- These kernels represent a compromise that was implemented by some developers before it was demonstrated that pure microkernels can provide high performance.
- Hybrid kernels should not be confused with monolithic kernels that can load modules after booting (such as Linux).
- Example OS with hybrid kernels: Microsoft Windows NT, 2000 and XP, Windows 7, DragonFly BSD.
OS Kernels Comparisons

Monolithic Kernel based Operating System
- Application
- VFS, System call
- IPC, File System
- Scheduler, Virtual Memory
- Device Drivers, Dispatcher, ...

Microkernel based Operating System
- Application
- UNIX Server
- Device Driver
- File Server
- Basic IPC, Virtual Memory, Scheduling

"Hybrid kernel" based Operating System
- Application
- Device Driver
- UNIX Server
- File Server
- Basic IPC, Virtual Memory, Scheduling

(courtesy Wikipedia)
Best current methodology for OS design involves using OOP techniques to create a **modular** kernel.

Kernel has set of *core* components and links in additional services as required – either at boot time or run time.

Uses **dynamically loadable modules**

- Solaris and 7 types of loadable modules
Apple’s Mac OS X uses a hybrid structure.

Top layers include application environments and services providing a graphical interface to applications.

Below these layers is kernel environment, which consists mainly of:

- Mach microkernel: provides memory management, thread scheduling and support for RPCs and other types of IPC.
- BSD kernel: file system support, networking.
MS-DOS

- Shell invoked when system booted
- Simple method to run program: No separate process created
- Single memory space; fig. (a)
- Loads program into memory, overwriting all but the kernel; fig. (b)
- On program exit shell is reloaded; back to (a)
FreeBSD

- Unix variant
- Multitasking
- On user login, invoke user’s choice of shell
- Shell executes `fork()` system call to create process
  - Executes `exec()` to load program into process
  - Shell waits for process to terminate or continues with user commands
- Process exits with:
  - code = 0: no error
  - code > 0: error code
Outline

1. Announcements
2. Kernel Types
3. Simple OS Examples
4. System Calls
   - System Call Overview
   - Types of System Calls – §2.4, SGG
Overview

- System call: a **request** made by any arbitrary program to the OS for which the said program does not have required permissions to execute.
- System calls are generally available as instructions and they are usually listed in the various manuals used by the assembly-language programmers.
  - Mostly accessed by programs via a high-level **Application Program Interface (API)** rather than direct system call use.
  - Managed by **run-time support library** (set of functions built into libraries included with a compiler).
Overview (contd.)

Three most common APIs are:

- **Windows** API
- **POSIX** API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
- Java API for the Java virtual machine (JVM)
Example Use of System Calls

System call sequence to copy the contents of one file to another file

Example System Call Sequence
- Acquire input file name
- Write prompt to screen
- Accept input
- Acquire output file name
- Write prompt to screen
- Accept input
- Open the input file
  - if file doesn't exist, abort
- Create output file
  - if file exists, abort
- Loop
  - Read from input file
  - Write to output file
  - Until read fails
- Close output file
- Write completion message to screen
- Terminate normally
The Linux `open()` function – a function (which uses a system call) for opening a file prior to reading / writing

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
```

**DESCRIPTION**

Given a `pathname` for a file, `open()` returns a file descriptor, a small, non-negative integer for use in subsequent system calls (`read(2)`, `write(2)`, `lseek(2)`, `fcntl(2)`, etc.). The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.
Implementing System Calls

- Typically, a **number** associated with each system call
  - System-call interface maintains a **table** indexed according to these numbers

- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values

- The caller (our program) need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)
API – System Call – OS Relationship
Standard C Library Example

```c
#include <stdio.h>
int main ()
{
    
    printf ("Greetings");
    
    return 0;
}
```
System Call Parameter Passing

- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in registers
    - ...but what if there are more parameters than registers?
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
    - Each individual system call knows how to treat the table contents
    - This approach taken by Linux and Solaris
  - Parameters placed, or pushed, onto the stack by the program and popped off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed
Links to Explore

- How to start developing your own OS kernel
- Java API
- POSIX API
- Mac OS X API
Outline

1. Announcements
2. Kernel Types
3. Simple OS Examples
4. System Calls
   - System Call Overview
   - Types of System Calls – §2.4, SGG
Categories

- Process control
- File management
- Device management
- Information maintenance
- Communications
Process Control

- end, abort
- load, execute
- create process, terminate process
- get/set process attributes
- wait for time
- wait/signal event
- allocate and free memory
File Management

- create/delete file
- open, close
- read, write, reposition
- get/set file attributes
Device Management

- request device, release device
- read, write, reposition
- get/set device attributes
- logically attach or detach devices
Information Maintenance

- get/set time or date
- get/set system data
  - number of current users
  - amount of free memory
  - (numerous other statistics)
Communication

- create, delete communication connection
- send, receive messages
- transfer status information
- attach or detach remote devices