System Calls

Library Functions

- Compilers typically ship with a “standard library” of functions, types, etc.
- This is for simplifying:
  - I/O
  - string manipulation
  - math
  - etc.
- E.g. the C standard library contains functions like `printf`, `strlen`, `sqrt`, `rand`, `srand` etc.

Library Functions

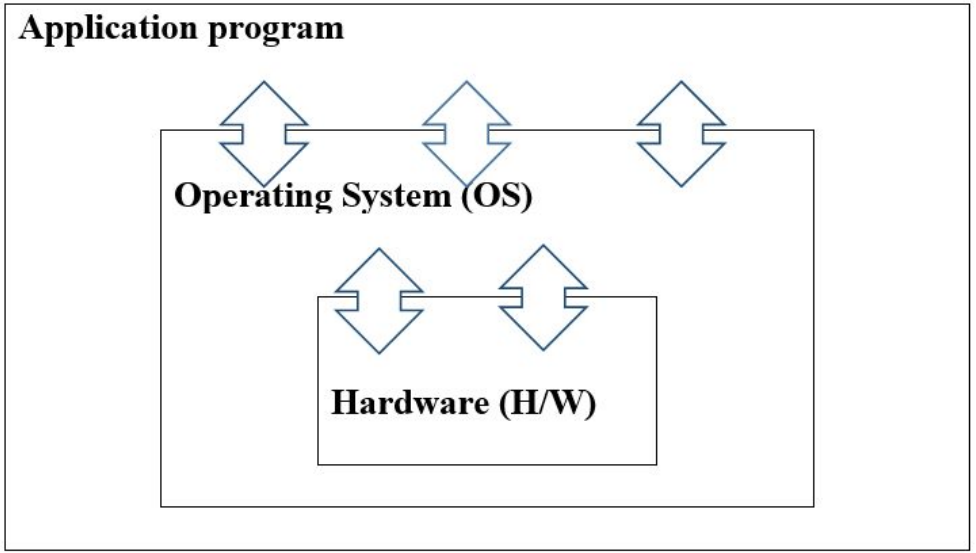
- In C, there is nothing special about these functions (other than being commonly needed). They themselves are written in C
- In C, even `malloc` and `free` are just library functions
- In C, a program can be (and usually is) linked to the needed portions of the standard library – sometimes called the “C runtime”
    - Some functions, such as `malloc` and `free`, required the runtime
- Aside: when you run such a program, it actually starts inside the C runtime. The runtime does some setup (e.g. of the heap) and then calls main

O/S Responsibilities

- The responsibilities of an operating system (O/S) include:
  - Managing system hardware
    - (e.g. I/O devices)
  - Managing system resources
    - (e.g. CPU time, memory)
  - Sometimes providing an environment for users
    - (e.g. a GUI)
  - **Providing an environment for programs**
    - **(a system call interface)**

O/S Responsibilities

- Here, we are concerned with the last point
- Diagram: system layers, showing H/W, O/S kernel, user programs and the system call interfaces (SCI)



O/S Responsibilities

- The arrows are SCIs, each is a doorway by which the two levels can communicate. Only a limited number are shown on the diagram, but in reality there are many more
- E.g. what happens when a user program needs to write a character to an output device?
- **Note:** The O/S manages all I/O devices.
    - On modern O/S's regular programs are not usually allowed to access I/O hardware directly – they must invoke the O/S
- What happened on computers before O/S existed?

O/S Responsibilities

- Some possibilities:
    1. if allowed, the program could deal directly with the output device hardware
    2. the program could invoke a “write” system call, and it would then deal with the output device hardware
    3. the program could invoke a library function (e.g. printf or putchar), and it would invoke the system call indirectly

O/S Responsibilities

- **Diagram:** levels at which program can work
- What are the pros and cons of each?

Level	Pros	Cons
Low level (H/W)	– Total control	– Must know all details for the H/W specific to this computer – Non-portable, limited to this device
System calls	– More generic – Less detailed knowledge needed	– Less control – Limited to this OS
Library	– H/W & system independent – Very portable	– Limited to what the library provides, i.e. buffered input

O/S Responsibilities

- How has this been covered in the degree
  - In 1701/1633/2631/2633 all programs work at the library level
  - In 2655 the programs worked at the system call level
  - In 2659 the programs will eventually work at the low level hardware level

System Calls in Assembly

- Consider the Atari ST (with its 68000 CPU and TOS O/S).
- TOS supports several system calls
  - e.g. `Cconout` for doing console output.
- The 68000 supports system calls using the `trap` instruction
  - details on what it actually does will come later

System Calls in Assembly

- Under TOS, system calls take their input parameters on the stack.
- E.g. this code writes a to the screen:

```
move.w  #'a',-(sp)      ; push input parameter
move.w  #2,-(sp)    ; Cconout is system call #2
trap    #1          ; call O/S
adda.l  #4,sp           ; correct stack
```

- Realize that system call code interacts with system resources
- Under TOS, output parameters (if any) are usually returned in a register, e.g. `D0`

System Calls in C

- Various library functions (standard, therefore portable) wrap O/S calls
- E.g. to write a character:

```
#include <stdio.h>

int main() {
    putchar('a');   /* wraps Cconout call */
    return 0;
}
```

System Calls in C

- E.g. alternatively:

```
#include <stdio.h>

int main() {
    printf("%c", 'a');  /* probably wraps putchar! */
    return 0;
}
```

System Calls in C

- However, system calls can be invoked more directly from C!
- E.g. this works:

```
#include <osbind.h>     /* Atari specific */

int main() {
    Cconout('a');       /* Atari specific */
    return 0;
}
```

System Calls in C

- ... but wait, how does the compiler know to generate a `trap` instruction instead of a regular `jsr`?
- It does not. `Cconout` is a library function implemented in asm. The library function turns around and does the `trap`.
- In other words, on each platform, the C standard library is extended with O/S specific system call wrappers, implemented in asm

System Calls in C

- Beware of mixing standard I/O library function calls and system calls in the same program
- What does this code output?

```
#include <stdio.h>
#include <osbind.h>

int main() {
        printf("%c", 'a');
        putchar('b');
        Cconout('c');

        printf("\n");

        return 0;
}
```

System Calls in C

- It outputs `cab`.
- Why?
- Because `printf` and `putchar` are buffered, they are NOT put to the screen until the buffer is flushed, which only occurs when a newline is output (or the program terminates).
- `Cconout` is not buffered so is immediately output. And since the buffer is only flushed AFTER the `Cconout` is executed it occurs first.

TOS Organization

- TOS is unusual, in that it has several levels of system call:
  - GEMDOS — a generic, hardware-independent SCI
  - BIOS and XBIOS — hardware-dependent, ST-specific SCI
  - LINE A — primitive graphics (and mouse) operations
  - GEM — GUI layer – not discussed here

TOS Organization

- GEMDOS is invoked using trap #1. It contains basic I/O, file system, date/time, ... routines.
- BIOS is invoked using trap #13. These are lower level.
- XBIOS is invoked using trap #14. These are also low-level, and ST-specific (e.g. routines for controlling sound chip).
- LINE A is actually invoked using a different technique, not discussed here. It contains routines for plotting pixels, lines, etc.