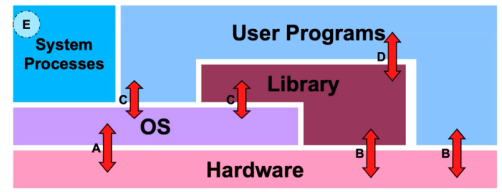
CS2106 Lec2b: System call

$_/|/|U_Ch@NgRu!$

May 10, 2021

1 System call interface



- **A**: OS executing machine instructions
- **B**: normal machine instructions executed (program/library code)
- **C**: calling OS using ***system call interface***
- **D**: user program calls library code
- **E**: system processes
  - Provide high level services, usually part of OS

1.1 The interface between the OS and the (user programs and user library) are called system interface

1.1.1 Library calls are programming language dependent, while system calls are dependent only on the operating system

1.2 Application Program Interface(API) to OS

1. provide way of calling facilities/service in kernel
2. Not the same as normal function call (have to change from user mode to kernel mode)

1.3 Different OS has different APIs

1.3.1 Unix variants:

mostly follow POSIX standards

small number of calls:≈100

1.3.2 Windows variants:

Uses Win API across different Windows versions
New version of windows usually adds more calls
Huge number of calls ≈ 1000

2 In C/C++ program, system call can be invoked almost directly

2.1 Majority of the system calls have a library version with the same name and the same parameters

The library version act as a function wrapper

2.2 a few library functions present a more user friendly version to the programmer(e.g. printf)

lesser number of parameters, more flexible parameter values etc
The library version acts as a function adapter

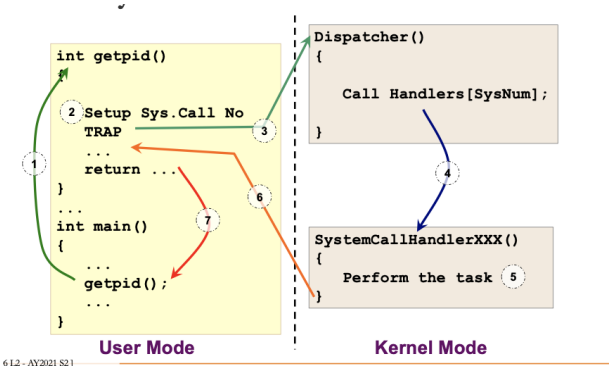
2.3 A universal wrapper for any system call

```
long syscall(long number, ...);
//the number represents the type of system call, and the latter parameters are the some
    parameters for the system call
```

2.4 Example of system call

```
#include <unistd.h> // header for system call
#include <stdio.h>
int main() {
int pid;
/* get Process ID */
pid = getpid(); // getpid() is a system call
printf("process id = %d\n", pid); //printf is a friendly function for calling write() system
    call
return 0; }
```

3 System call mechanism



3.1 User program invokes the library call

Using the normal function call mechanism as discussed
as shown in (1) in the above image

3.2 Library call (usually in assembly code) places the system call number in a designated location(E.g. Register)

as shown in (2) in the above image

3.3 Library call executes a special instruction to switch from user mode to kernel mode

That instruction is commonly know as TRAP(it can raise an exception and transfer control to kernel)
Where transfer from user mode to kernel mode
as shown in (3) in the above image

3.4 Now in kernel mode, the appropriate system call handler is determined

Using the system call number as index
This step is usually handled by a dispatcher
as shown in (4) in the above image

3.4.1 Syscall dispatcher

It receive the Sysnum, and there is a table contains pointers(function pointer) pointing to some code. The Sysnum can be a offset and corresponding systemCallHandler will be selected

3.5 System call handler is executed

Carry out the actual request
as shown in (5) in the above image

3.6 System call handler ended

Control return to the library call
Switch from kernel mode to user mode
as shown in (6) in the above image

3.7 Library call return to the user program

via normal function return mechanism
as shown in (7) in the above image

4 Exception

4.1 Executing a machine level instruction can cause exception

They cannot come at any time but certain scene, so it's predictable e.g
Arithmetic Errors: Overflow, Underflow, Division by Zero
Memory Accessing Errors: Illegal memory address, Mis-aligned memory access

4.2 Exception is Synchronous

occur due to program execution
They cannot come at any time

4.3 Effect of interrupt

Have to execute a exception handler
Similar to a forced function call

5 Interrupt

5.1 External events can interrupt the execution of a program

5.2 Usually hardware related

Timer, Mouse Movement, Keyboard Pressed etc

5.3 Interrupt is asynchronous

Events that occurs independent of program execution

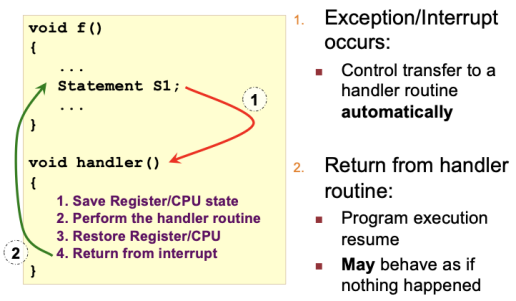
5.4 Different interrupt has different priority

nested interrupt: when a interrupt is executing, the system get the interrupt again

5.5 Effect of interrupt

Program execution is suspended
Have to execute an interrupt handler

6 Exception/Interrupt handler: illustration



1. Exception/Interrupt occurs:
   - Control transfer to a handler routine **automatically**
2. Return from handler routine:
   - Program execution resume
   - **May** behave as if nothing happened

7 Exercise

7.1 A process is currently executing the following instruction that loads a byte resideng at memory address addrA into register reg1

```
load reg1, addrA;
```

In the middle of the instruction, while the data is travelling on the bus, the **highest priority interrupt** is received. What will be the action of the operating system? a. The OS will send the data back to the memory to free up the bus for the highpriority interrupt routine.
b. The OS will save the data into a buffer next to the bus so that the transfer can continue after the high-priority interrupt is served
c. The OS will drop the data without saving it to minimize the interrupt latency.
The correct answer:none of above

There is no such buffer in b; OS is a piece of soft and it contains a set of instructions
The highest priority interrupt here is hardware
so OS cannot do any thing cannot execute an instruction to control the hardware execution of current instruction
The OS cannot control whether to write the data into a buffer
So the the OS will finishes the execution of the instruction first and then serves the interrupt(check whether need interrupt)

8 Process Creation in Unix: fork()

```
#include <unistd.h>
int fork( );
```

Returns: PID of the newly created process (for parent process) OR
0 (for child process)

8.1 Behavior

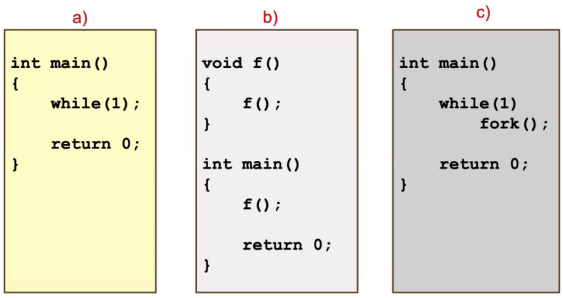
Creates a new process (known as child process)
Child process is a duplicate of the current executable image(i.e. same code, same address space etc; Data in child is a COPY of the parent (ie.not shared))
Both parent and child processes continue executing after fork( ) subsectionChild difference Process id (PID)
Parent (PPID)(Parent = The process which executed the fork())
fork() return value

8.2 distinguish parent and child

```
int var = 1234;
int result;
result = fork();
if (result != 0){
printf("Parent: Var is %i\n", var); var++;
printf("Parent: Var is %i\n", var);
}
else {
printf("Child: Var is %i\n", var); var--;
```

```
printf("Child: Var is %i\n", var);
}
```

8.3 There is risk for the fork to make stack overflow(if the fork() exponentially grows)



fork bomb

8.4 Implementation(Simplified)

1. Create address space of child process
2. Allocate p'=new PID
3. create kernel process data structure
4. copy kernel environment of parent process
5. Initialize child process context
6. Initialize child process context: PID=p', PPID=parent, id, zero CPU time
7. Copy memory regions from parent(Code, Data, Stack)(This is very expensive operation that can be optimized)
8. Acquires shared resources(Open files,current working directory, etc.)
9. Initialize hardware context for child process(Copy registers, tec. from parent process)
10. children process is ready to run

8.4.1 An optimization

Memory copy is very expensive: potentially need to copy the whole memory space, there is a observation that the child process will not access the whole memory range right away, so the optimization can be:

1. If child just read from a location(Remain unchanged; can use a shared version)
2. Only when write is perform on a location, then two independent copies are needed

9 Process Creation in Unix:exec( )

9.1 exec( ) family

execv, execl, execle, execlv, execlp, etc

9.2 execl()

To replace current executing process image with a new one

```
#include <unistd.h>
int execl( const char *path,
          const char *arg0,
          ...,
          const char *argN, NULL );
```

path: Location of the executable
arg0, ..., argN: Command Line Argument(s)
NULL: To indicate end of argument list

10 Command Line Argument in C

```
int main( int argc, char* argv[] ) {
//use argc and argv
}
```

10.1 argc

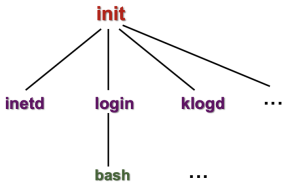
Number of command line arguments
Including the program name itself

10.2 argv[]

A char strings array
Each element in argv[] is a C character string

11 The Master Process

The ancestor of every process 1. The Master Process
2. Created in kernel at boot up time
3. Traditionally has a PID = 1
4. Watches for other processes and respawns where needed
5. fork() creates process tree:



Note: just a simple example, actual process tree varies according to Unix setup

12 Process Termination: exit()

```
#include <stdlib.h>
void exit( int status );
```

Status is returned to the parent process (more later)
Unix Convention:
0 = Normal Termination (successful execution)
!0 = To indicate problematic execution
The exit() itself **has no return**!! the "return" has

12.1 resource release

Most system resources used by process are released on exit:
for example, File descriptors(opened file in C has a file descriptor attach to it; in Java File object; in C++ File stream Object)
However, Some basic process resources not releasable

PID & status needed(not release for parent-children synchronization)

Process accounting info, e.g. cpu time Process table entry may be still needed