

Function pointers, Signals and low level file I/O

COMP2017/COMP9017

Dr. John Stavrakakis

FACULTY OF
ENGINEERING



THE UNIVERSITY OF
SYDNEY



- › So far all variables are exposed as having an address

- › Compiled binary code is no different

```
if ( != 0 ) {  
    execute statement;  
} else {  
    execute other statement;  
}
```

```
int x = 33;  
if (x == 33)  
{  
    x = 480+7;  
}  
x += 768;
```



```
movl $33, -4(%rbp)  
cmpl $33, -4(%rbp)  
jne .L2  
movl $487, -4(%rbp)  
.L2:  
    addl $768, -4(%rbp)
```

- › JUMP!
- › Same with loops
- › rbp is the stack frame pointer on x86_64

```
int x = 33;  
if (x == 33)  
{  
    x = 480+7;  
    foo();  
}  
x += 768;
```



```
movl $33, -4(%rbp)  
cmpl $33, -4(%rbp)  
jne .L4  
movl $487, -4(%rbp)  
movl $0, %eax  
call foo  
.L4:  
    addl $768, -4(%rbp)
```

› Call? If not a jump, how do we get back?

```
int x = 33;  
if (x == 33)  
{  
    x = 480+7;  
    foo();  
}  
x += 768;
```



```
movl $33, -4(%rbp)  
cmpl $33, -4(%rbp)  
jne .L4  
movl $487, -4(%rbp)  
movl $0, %eax  
call foo  
.L4:  
    addl $768, -4(%rbp)
```

- › Call? If not a jump, how do we get back?
- › Stack is being managed here. Callee or caller will setup and teardown the stack

```
int (*fptr)() = foo;
```

```
int x = 33;  
if (x == 33) {  
    x = 480+7;  
    fptr();  
}  
x += 768;
```



```
subq $16, %rsp  
leaq foo(%rip), %rax  
movq %rax, -16(%rbp)  
movl $33, -4(%rbp)  
cmpl $33, -4(%rbp)  
jne .L4  
...
```

- › If we jump, or call, all we need is an address

```
int (*fptr)() = foo;
```

```
int x = 33;  
if (x == 33)  
{  
    x = 480+7;  
    fptr();  
}  
x += 768;
```



```
...  
movl $33, -4(%rbp)  
cmpl $33, -4(%rbp)  
jne .L4  
movl $487, -4(%rbp)  
movq -16(%rbp), %rdx  
movl $0, %eax  
call *%rdx  
.L4:  
    addl $768, -4(%rbp)
```

- › Call a function, jump to address is (almost) the same process

- › A function pointer is an address that refers to an area of memory with executable code
- › Typically the first instruction of the function call[^]
- › Are useful for conventional programming patterns
- › Examples
 - Do something, and when you are finished call this function
 - Do something, and if it goes wrong, call this function
 - I am a data source, give me an function to send new bits of data to
 - I want to sort a list of objects, here is the address of a function to perform comparison of two elements

[^] Depends on callee/caller conventions



- › The declaration of the function pointer parameter looks like:
type (*f)(param declaration...)
 - › and the call of the function looks like:
f(params...)
-

Function pointer: Example

- › Call functionA if x is true, or functionB otherwise

```
if (x)
    funcA();
else
    funcB();
```

Function pointer: Example

- › Call functionA if x is true, or functionB otherwise

```
if (x)
    funcA();
else
    funcB();
```

- › What if we don't know what funcA and funcB are at compile time?

```
void do_process(int x, funcA?, funcB?) {
    if (x)
        funcA(x); // print X
    else
        funcB(x); // delete elem X
}
```

Function pointer: Example

- > What if we don't know what funcA and funcB are at compile time?

```
void deleteX(int x);  
void printX(int x);
```

```
void do_process(int x,  
    void __funcA__(int),  
    void __funcB__(int))  
{  
    if (x)  
        funcA(x); // print X  
    else  
        funcB(x); // delete elem X  
}
```

Function pointer: Example

- > What if we don't know what funcA and funcB are at compile time?

```
void deleteX(int x);  
void printX(int x);  
  
void do_process(int x,  
    void (*funcA)(int),  
    void (*funcB)(int))  
{  
    if (x)  
        funcA(x); // print x  
    else  
        funcB(x); // delete elem x  
}
```

Function pointer: Example

- Write less code. Allow option to change implementation choices at runtime. E.g. heuristics, look and feel, plugins

```
void printX_1(int x) { printf("%d", x); }  
void printX_2(int x) { printf("%d\n", x); }  
void printX_3(int x) { printf("x: %d\n", x); }
```

```
// delegate which fn pointer  
if (user_style == PRETTY)  
    print_style = printX_3;
```

```
...
```

```
// generic code  
do_process(value1, print_style, remove_style);  
do_process(value2, print_style, remove_style);  
do_process(value3, print_style, remove_style);
```

Signals

COMP2017/COMP9017

FACULTY OF
ENGINEERING



THE UNIVERSITY OF
SYDNEY



- › a process can communicate with another using a *signal*
 - › these are a form of *software interrupt*
 - › execution is interrupted and a function call is made at that point to a user specified function
 - › when the function returns, execution is resumed
-

- › signals can be generated by one process to another using the *kill* system call
 - › signals are also generated by the operating system, eg when an access outside memory bounds is attempted (Segmentation Fault)
-



man 7 signal for standard numbers

SIGHUP	1	Hangup	SIGFPE	8	Arithmetic Exception
SIGINT	2	Interrupt	SIGKILL	9	Kill
SIGQUIT	3	Quit	SIGUSR1	10	User Signal 1
SIGILL	4	Illegal Instruction	SIGSEGV	11	Segmentation Fault
SIGTRAP	5	Trace or Breakpoint Trap	SIGUSR2	12	User Signal 2
SIGABRT	6	Abort	SIGPIPE	13	Broken Pipe
SIGIOT	6	Input/Output Trap	SIGALRM	14	Alarm Clock
SIGBUS	7	Bus Error	SIGTERM	15	Terminated
SIGEMT	-	Emulation Trap (non x86	... many more!		

- › You can send a signal to a running process from the command line using the `kill` command

- › Eg `kill -9 12345`

Will send the SIGKILL signal to process 12345.

- › Some signals can be *caught* and handled by a user supplied function
 - › Some signals (such as SIGKILL) cannot be caught and caused the process to be terminated
-

- > You can send a signal to a running process using the kill system call function

```
#include <sys/types.h>
```

```
#include <signal.h>
```

```
int kill (pid_t pid, int sig);
```

Where pid is the process ID of the process to be signaled and sig is the signal to be sent.

Catching Signals

- › You can “catch” a signal by specifying a function that is called when the signal is received
- › This is done using the signal function:

```
#include <signal.h>
```

```
sighandler_t signal(int signum, sighandler_t handler);  
void (*signal(int sig, void (*catch)(int)))(int);
```

This complicated looking declaration means that `signal` is called with 2 arguments: the first is the signal to catch, the second is a pointer to the function that will be called when the signal is received. The `signal` function returns a pointer to the function that previously caught the signalpew.

Signal: Catch SIGINT

```
volatile int interrupted = 0;  
→ void impatient(int sigval) {  
    interrupted = 1;  
}
```

```
int main() {  
→   signal(SIGINT, impatient);  
   printf("Now we wait...\n");  
   while (!interrupted)  
       usleep(10);  
   printf("Oh..you didn't like waiting\n");  
   printf("Program terminated\n");  
   return 0;  
}
```

Does it work?



errno

- › Most C functions report errors via return values, or their parameters
 - › However, there is still an error reporting mechanism using a global variable called `errno`
 - › Failed system calls typically set `errno` to be an integer value representing the type of error.
 - › A companion function, `strerror` and `perror`, will print a textual description of the `errno` code.
-

- › The `<errno.h>` header file defines the integer variable

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

```
int main() {
    errno = 0;
    FILE *fp = fopen("doesn't exist", "r");
    printf("errno: %d\n", errno);
    return 0;
}
```

- › `errno` is set by the **last** function call that will set `errno`.
 - › There is only one `errno` value.
 - › It can be overridden by subsequent function calls
 - › It is important to:
 - Initialise it to zero before calls, and
 - save this value immediately following
-

Low level file I/O

COMP2017/COMP9017

FACULTY OF
ENGINEERING



THE UNIVERSITY OF
SYDNEY



Low Level File Descriptors

- › Low level I/O is performed on file descriptors that are integers indicating an open file
 - › When a process is started file descriptor 0 is standard input, 1 is standard output, 2 is standard error output (UNIX)
 - › System call functions operate using these file descriptors (not higher level C lib FILE struct)
-

Low Level File Descriptors

- › low level I/O functions in C wrap system calls:
 - creat, open, close
 - read, write
 - ioctl
 - umask
- › eg read 100 characters from standard input into array “buffer”
`ssize_t result = read(0, buffer, 100);`

\$ man 2 open

“If I was to change anything in Unix it would be to spell creat with an e”

Ken Thompson

> read()

On error, -1 is returned, and errno is set appropriately. In this case, it is left unspecified whether the file position (if any) changes.

- > This may be interrupted by a signal. The way to check is to use `errno`

```
ssize_t result = read(...);  
if (result < 0)  
    error_val = errno;  
if (EINTR == error_val) // reattempt
```

- > These operations are blocking. There may be a need to interrupt them upon a new event.
-

Working with read and write

> Error checking

- errno is set to an error value
- signal can be sent by operating system

```
#include <errno.h>
```

```
...
```

```
signal(SIGINT, interrupted);
```

```
char buffer[100];
```

```
ssize_t result = read(0, buffer, 100);
```

```
// check for errors
```

```
int error_val = errno;
```

```
if (0 != error_val) {
```

```
    printf("read() was interrupted by signal\n");
```

```
}
```

Does it work?

- › You can “catch” a signal by specifying a function that is called when the signal is received
- › This is done using the **sigaction()** function:

```
#include <signal.h>
```

```
int sigaction(int signum, const struct sigaction *act,  
              struct sigaction *oldact);
```

```
struct sigaction {  
    void      (*sa_handler)(int);  
    void      (*sa_sigaction)(int, siginfo_t *, void *);  
    sigset_t   sa_mask;  
    int        sa_flags;  
    void      (*sa_restorer)(void);  
};
```

Working with read and write

> Error checking

- errno is set to an error value
- signal can be sent by operating system

```
#include <errno.h>
```

```
...
```

```
// setup new handler
```

```
new_sig_int.sa_handler = interrupted;
```

```
new_sig_int.sa_flags = 0;
```

```
// install the new handler
```

```
sigaction(SIGINT, &new_sig_int, NULL);
```

```
char buffer[100];
```

```
ssize_t result = read(0, buffer, 100);
```

```
// check for errors
```

```
int error_val = errno;
```

```
if (error_val != 0) {
```

```
    printf("read() was interrupted by signal\n");
```

It works

```
}
```


Low Level File Descriptors

- › Extra attention is needed when working with files at this level
 - Buffering
 - Sharing vs exclusive access (resource locking)
 - Errors and interruptions
 - Notifications (Linux)
 - Resource limit setting
 - Performance

 - › `fcntl` - manipulate file descriptors

 - › Valuable to have very fine control of file operations
-

- ✓ Understand what is a function pointer
 - ✓ Be able to define a function pointer type
 - ✓ Understand the idea of a signal and it's interaction with a process (exceptional flow control)
 - ✓ Be able to send a signal
 - ✓ Be able to setup signal handling (receive)
 - ✓ Be able to set and check errno
 - ✓ Be able to read and write with low level file descriptors
 - ✓ Check for signal interrupts to blocking function calls
-