2019 Spring - System Programming

1. `setjmp()`, `longjmp()` 가 사용되는 예제를 하나만 드시오.

   - The `setjmp` function saves the current *calling environment* in the `env` buffer, for later use by `longjmp`, and returns 0. The calling environment includes the program counter, stack pointer, and general purpose registers.
   - The `longjmp` function restores the calling environment from the `env` buffer and then triggers a return from the most recent `setjmp` call that initialized `env`. The `setjmp` then returns with the nonzero return value `retval`.

     ```
     #include <setjmp.h>
     int setjmp(jmp_buf env);
     void longjmp(jmp_buf env, int retval);
     ```

   - Example.

     ```
     #include "csapp.h"
     jmp_buf buf;
     int error1 = 0, error2 = 1;
     void foo(void), bar(void);
     int main() {
         switch(setjmp(buf)) {
             case 0:
                 foo(); break;
             case 1:
                 printf("Detected error 1 in foo\n"); break;
             case 2:
                 printf("Detected error 2 in foo\n"); break;
             default:
                 printf("Unknown error happened.\n")
         }
         exit(0);
     }

     void foo(void) {
         if(error1) longjmp(buf, 1);
         bar();
     }

     void bar(void) {
         if(error2) longjmp(buf, 2);
     }
     ```

   - Permits an immediate return from a deeply nested function call. Good tool when a non-local jump is needed. Normal `jmp` can't jump to other functions without call/return.

2. Library interpositioning 이 compile time, link time, run time 에 일어나는 과정을 설명하시오.

   - **Library Interpositioning**: Allows you to intercept calls to shared library functions and execute your own code instead.
   - **Compile Time**: Apparent calls to user's `#define` function get macro-expanded into calls. It's very simple, but you must have access to original source and must be able to recompile this.
   - **Link Time**: Interpose when the relocatable object files are statically linked to from an executable object file.
   - **Load/Run Time**: Interpose when an executable object file is loaded into memory, dynamically linked, and then executed. Implement customized version of X', Y', and use dynamic linking to load the library functions under different names.

3. Zombie process 가 어떻게 생기는지, 이를 어떻게 방지할 수 있는지 설명하시오.

   - When a parent process dies *before* the child process, `init` will take care of the orphaned child.
   - A terminated process that has not yet been reaped is called a zombie, and this happens when the child process is terminated but not reaped by the parent process.
   - To prevent this, the parent process can use the `waitpid` function to reap zombie children.

4. TLB 의 역할을 서술하고 일반적인 cache 보다 hit ratio 가 높은 이유를 설명하시오.

   - **Translation Lookaside Buffer** is a small cache of page table entries in the memory management unit. TLB is a small, virtually addressed cache where each line holds a block consisting of a single PTE.
   - Each entry in the TLB is a adress to a page, and a page is generally 4K. Every address within a page will cause a page hit. But in normal caches, the range for a cache hit is block (word) size, which is 4 bytes. Thus TLB's hit ratio is higher.

5. File I/O 에서 short count 란 무엇인지, 언제 일어나는지 설명하시오.

   - `read` and `write` functions may transfer fewer bytes than the application requests. This is called a **short count**, and this is not an error.
   - Short counts happen in these cases.
     - Encountering EOF on reads.
     - Reading text lines from a terminal.
     - Reading and writing network sockets.

6. Executable file 이 메모리에 올라갈 때 어떤 위치에 올라가게 될 지는 미리 알 수 없다. 올라가는 위치의 주소에 따라 변수나 함수의 주소가 바뀌게 될텐데, linker 와 loader가 이를 어떻게 처리하는지 설명하시오.

   - The **linker** resolves symbols, relocates the address in each object file, and merges then into a single address space. Also, dynamic linking binds shared libraries at runtime.

- The **loader** copies the code and data in the executable object file from disk into memory and then runs the program by jumping to its first instruction, or *entry point*.

7. Global variable 을 사용했을 때 생기는 문제점과 해결방안을 제시하시오.

   - The use of global variables may cause naming conflicts. If the variable is not initialized, many source codes may refer to the same variable, which may cause nasty bugs. (Weak symbol)
   - Solutions
     - Avoid them if possible
     - Use `static` keyword
     - Initialize when defining a global variable
     - Use `extern` keyword when referencing an external global variable.

8. 지금의 Linux 에서는 여러 개의 시그널이 도착해도 이를 모두 처리하는 것은 불가능하다. 그 이유를 설명하고 어떻게 하면 모든 시그널을 받을 수 있겠는지 방법을 제시하시오.

   - Since the `pending` variable is a bit vector, it can only detect on/off. So when many signals of the same kind arrive, it will only change 1 bit of the `pending` bit vector. And this bit is set to 0 after the signal is handled. So some of the signals may be ignored.
   - If a bit in the `pending` bit vector is 1, we can only infer that at least 1 signal has arrived.
   - Thus when writing a handler, one must assume that at least 1 signal has arrived, and try to let the handler do all the work of cleaning and finding out what to do.
   - (???) One can use `sigqueue` to queue a signal and data to a process.

9. ELF file format

   - `.data`: Initialized global variables
   - `.bss`: Uninitialized global variables, no space
   - `.symtab`: Procedure and static variable names

10. Guidelines for writing safe signal handlers

    (a) Keep your handlers as simple as possible

    (b) Call only *async-signal-safe* functions in your handlers

    (c) Save and restore `errno` on entry and exit

    (d) Protect access to shared data structures by temporarily blocking all signals (in both handler and main - `sigprocmask`)

    (e) Declare global variables as `volatile`

    (f) Declare global flags as `volatile sig_atmoic_t` (Read/write happens on one un-interruptable step)

11. Relocation Algorithm

```
foreach section s {
    foreach relocation entry r {
        refptr = s + r.offset; // ptr to reference to be relocated

        // Relocate a PC-relative reference
        if(r.type == R_X86_64_PC32) {
            refaddr = ADDR(s) + r.offset; // ref's run-time address
            *refptr = (unsigned) (ADDR(r.symbol) + *refptr - refaddr);
        }

        // Relocate an absolute reference
        if(r.type == R_X86_64_32)
            *refptr = (unsigned) (ADDR(r.symbol) + *refptr);
    }
}
```

12. Unix I/O
    (a) Pros
        i. Most general and lowest overhead form of I/O
        ii. Provides functions for accessing file metadata
        iii. Functions are async-signal-safe and can be used safely in signal handlers
    (b) Cons
        i. Dealing with short counts is tricky and error prone
        ii. Efficient reading of text lines requires some form of buffering, also tricky and error prone
        iii. The above issues can be handled by standard I/O and RIO packages
13. Standard I/O
    (a) Pros
        i. Buffering increases efficiency by decreasing the number of `read`, `write` system calls
        ii. Short counts are handled automatically
    (b) Cons
        i. Provides no function for accessing file metadata
        ii. Not async-signal-safe. Not appropriate for signal handlers
        iii. Not appropriate for network sockets
14. Virtual Memory
    (a) Uses main memory efficiently
    (b) Simplifies memory management
    (c) Isolates address spaces