Recitation

By yzhuang, sseshadr
Agenda

• Debugging practices
  – GDB
  – Valgrind
  – Strace

• Errors and Wrappers
  – System call return values and wrappers
  – Uninitialization
  – `malloc()` related bugs

• System IO
Debugging Practices
General Debugging

• printfs are a good start, but won’t solve everything

• Remember printfs CHANGE your code
  – And how it’s compiled
  – And how it runs
    • Especially for races

• A lot of the debugging tools should be used with the –g compiler flag
GDB

- From bomblab / buflab
- You WILL need it for malloc
- Demo?
Valgrind

• Memory related issues
• Lots of options
• `man valgrind`
• `valgrind --leak-check=full ./a.out`
• Demo?
strace

• From the man page
  – “In the simplest case strace runs the specified command until it exits. It intercepts and records the system calls which are called by a process and the signals which are received by a process. The name of each system call, its arguments and its return value are printed on standard error or to the file specified with the -o option.”

• Cool for debugging!
Errors and Wrappers
System Call Error Handling

• Always handle errors for every system call
  – #include <errno.h>
  – Failed system calls almost always return -1
  – Global integer error number: errno
  – Getting error description: strerror(errno)

• We deduct style points for not handling system call errors
Wrappers

• If a system call is frequently used, create a wrapper for it. For example:

```c
pid_t Fork(void){
    pid_t pid;
    if ( (pid = fork() ) < 0 ){ //error handling }
    return pid;
}
```

• Proclab: always handle errors
  – You can choose whether to use wrappers
malloc

- #include <stdlib.h>
- void *malloc(size_t size);
- Allocates size bytes of memory
- A pointer is returned
- Returned memory **uninitialized!!**
  ```c
  p=(struct cacheline*) malloc( sizeof(cacheline) );
  P->valid = `?
  -- Cachelab: using uninitizlied valid bit (very bad)!
  ```
callloc

• With malloc
  – Either initialize
  – Or use calloc

• void * calloc ( size_t num, size_t size );
  – Allocate num * size bytes of memory
  – Initialized to 0

• Caveat: what if num * size causes an overflow? Check before calling calloc.
free

- Free memory allocated by malloc/calloc

- **Common mistakes:**
  - Freeing memory already freed
  - Freeing memory not allocated
  - Writing to memory already freed
  - Index-out-of-bound accesses of allocated array
  - Not freeing allocated memory
I/O
System I/O Basics

• Four basic operations:
  – open
  – close
  – read
  – write

• What’s a file descriptor?
  – Returned by open.
  – int fd = open("/path/to/file", O_RDONLY);
    • fd is some positive value or -1 to denote error
System I/O Basics

• **Every** process starts with 3 open file descriptors
  - 0 - STDIN
  - 1 - STDOUT
  - 2 - STDERR

• Can I **close** these file descriptors?
  – Yes!
  – But you shouldn’t... this next example is just for illustrative purposes
Sample Code

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char ** argv){

    int fd = atoi(argv[1]);
    argc = argc; /* Keep GCC Happy */

    fprintf(stdout, "STDOUT: close(%d) = %d\n", fd, close(fd));
    fprintf(stderr, "ERROR: close(%d) = %d\n", fd, close(fd));

    return 1;
}

What are the outputs when run with ./a.out 0 , ./a.out 1 , and ./a.out 2?
Sample Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char ** argv){
    int fd = atoi(argv[1]);
    argc = argc; /* Keep GCC Happy */
    fprintf(stdout, "STDOUT:close(%d) = %d\n", fd, close(fd));
    fprintf(stderr, "STDERR:close(%d) = %d\n", fd, close(fd));
    return 1;
}
>> ./a.out 0
STDOUT:close(0) = 0
STDOUT:close(0) = -1
>> ./a.out 1
STDERR:close(1) = -1
Why -1 on the second time??
Why no STDOUT output? And why -1 for close return value?
>> ./a.out 2
STDOUT:close(2) = 0
Why no STDERR output? And why 0 for close return value this time?
```
Some Real Stuff

From Lecture:

Descriptor table
[one table per process]

Open file table
[shared by all processes]

v-node table
[shared by all processes]

File A (terminal)

File B (disk)

Info in stat struct
Some Real Stuff

From Lecture:

CALL TO fork()
Some Real Stuff

From Lecture:

Descriptor table
[one table per process]

Open file table
[shared by all processes]

v-node table
[shared by all processes]

Diagram:

- **Parent**:
  - fd 0
  - fd 1
  - fd 2
  - fd 3
  - fd 4

- **File A (terminal)**:
  - File pos
  - refcnt=2

- **Child**:
  -fd 0
  - fd 1
  - fd 2
  - fd 3
  - fd 4

- **File B (disk)**:
  - File pos
  - refcnt=2

- **File access**
- **File size**
- **File type**
Dup

• What is file redirection?
  – Redirection vs. Pipes
    • Redirection has one “input” of a file, pipes can be between tasks
    • Ex: `cat < filename`
    • Ex: `find . | cat`
  – What is `dup2`?
    • It switches which file a file descriptor is pointing to!
  – What is `dup`?
    • It initializes another file descriptor to point to an already existing file!
Some Real Stuff

From Lecture:

- **Descriptor table** [one table per process]
- **Open file table** [shared by all processes]
- **v-node table** [shared by all processes]

Diagram:
- `stdin` and `stdout` with `fd 0` and `fd 1` respectively.
- `stderr` with `fd 2` to `fd 4`.
- `File A (terminal)` and `File B (disk)`
- `File access`, `File size`, `File type`, and `refcnt=1` are shown.
- `Info in stat struct`
Some Real Stuff

From Lecture:

CALL TO
dup2 (4, 1)
Dup

From Lecture

Descriptor table
[one table per process]

Open file table
[shared by all processes]

v-node table
[shared by all processes]

stdin  fd 0
stdout fd 1
stderr fd 2
fd 3
fd 4

File A
- File pos
- refcnt=0

File B
- File pos
- refcnt=2

File access
File size
File type

File access
File size
File type
**Dup**

- So what is `dup`, and what is `dup2`?
  - `int fd1 = open(…);`
  - `int fd2 = dup(fd1);`

- `int fd1 = open(…);`
  - `int fd2; dup2(fd1, fd2);`

- Are these the same??
  - NO!
  - The first is OK, the second uses an uninitialized variable!
    Remember, it’s not
    `dup2(fd1, &fd2);`
File writing example

• Questions
  – What are all of the possible contents of the file after running this code??
  
  – What is wrong with the style of this code?
  
  – How would you close these file descriptors?
#include <stdio.h>
#include <unistd.h>

int main()
{
    int fd1, fd2, fd3, parent = 0;
    char *fname = "filename";

    fd1 = open(fname, O_CREAT|O_TRUNC|O_RDWR, S_IRUSR|S_IWUSR);
    write(fd1, "A", 1);

    fd3 = open(fname, O_APPEND|O_WRONLY, 0);
    write(fd3, "BBB", 3);

    if((parent = fork()))
        fd2 = dup(fd1);
    else
        fd2 = fd3;

    write(fd2, "C", 1);
    write(fd3, "D", 1);

    if(parent) waitpid(-1, NULL, 0);

    return 0;
}
File writing example

• Answers
  – Possible outputs:
    • ACBBDCD
    • ACBBCDD
  – What is wrong with the style of this code?
    • Didn’t check error codes, didn’t close anything, no comments.
  – How would you close these file descriptors?
    • fd2 sometimes needs close() ’d, sometimes doesn’t!
    • So: don’t do both \( fd2 = fd3; \) and \( fd2 = \text{dup}(f1); \)
Practice!!

• Tons of practice available in past exam #2’s
• Very likely to be an I/O question on the next test!