Concurrency & Debugging

18-345 Spring 2015
Don’t Forget!

• Project deadline!
  – Fri Feb 6th, 11:59 PM

• HW1 deadline
  – Today, Fri Jan 30th, 11:59 PM
  – Submit on Piazza

• Quiz
  – Mon Feb 2nd, last 30 min. of class
  – Lectures 1-5
Announcements

• Office Hours
  – Prof. Peter Steenkiste
    • GHC 9107, Thu 11:00 – 12:00 AM
  – Antonio Rodrigues
    • PH B33, Tue 5:00 – 6:00 PM
Outline

• Concurrency
  – Forking
  – select()

• Debugging tools
  – General tips
  – Debugging macros
  – Debugging tools
Flashback!

- `accept()` blocks
- `read()` and `write()` block
- How to implement concurrent connections?

### Server
- `socket()`
- `bind()`
- `listen()`
- `accept()`
- Blocks
- `read()`
- `write()`
- `close()`

### Client
- `socket()`
- `connect()`
- Data
- `write()`
- `read()`
- `close()`
Solution 1 - Forking

- Main server blocks on accept()
- Accept incoming connection
- fork() child process for each connection
- fork() method:
  - Creates a duplicate copy of the parent process
  - Different process IDs (PIDs)
  - Returns 0 in child process
  - Returns PID in parent process
Forking

Server

socket()
bind()
listen()
accept()
close(newsockfd)
close(sockfd)

Client

socket()
connect()
fork()
write()
read()
write()
close()
exit()
Pseudo-Code

• Loop
  – New socket = accept()
  – Pid = fork() // Replicate current process
  – If (pid < 0)
    • Error
  – If (Pid == 0) //We are in child process
    • Close listening socket
    • Do stuff
    • Exit
  – If (Pid > 0) //We are in parent process
    • Close created socket
Forking - Drawbacks

• fork() is expensive
  – Memory copied from parent to child
  – All descriptors duplicated, etc.
• IPC required to pass information between parent and child (after fork())
• E.g. ‘threads’ can be created 10-100 times faster and share global memory
Solution 2 - \texttt{select()}

- Event driven programming!
- Single process that \texttt{multiplexes} all requests.
- Caveat
  - Programming is not so transparent!
  - Server no longer acts like it has only one client!
How to use `select()`?

- Sockets are treated just like files:
  - `socket()` returns a file descriptor
  - `read()` & `write()`: same interface as for files

- Give `select` a set of sockets/file descriptors.
How to use `select()`?

- `select()` blocks till **something** happens.
  - Data coming in on some socket.
  - Able to write to a socket.
  - Exception at the socket.
- Once woken up, check for the event and **service** it the way the server would do.
select()

#include <sys/select.h>

int select ( 
    int nfds,
    fd_set*  readfds, //
    fd_set*  writefds,
    fd_set*  exceptfds,
    struct timeval *timeout);

select() Parameters

- The FDs between 0 to nfds-1 are checked.
- Check for reading in readfds.
- Check for writing in writefds.
- Check for exception in exceptfds.
- These fd_sets can be NULL.
- timeout
  - NULL – blocking
  - else how long to wait for the required condition before returning to the caller.
fd_set Datastructure

- Remember, file descriptor is just an integer!
- Helper macros:

  ```c
  FD_ZERO(fd_set* fdset); /* initializes fdset to have 0s for all fds */
  FD_SET(int fd, fd_set* fdset); /* sets the bit for fd in fdset */
  FD_CLR(int fd, fd_set* fdset); /* clears the bit for fd in fdset */
  FD_ISSET(int fd, fd_set* fdset); /* returns non-0 if fd is set, else 0 */
  ```
Return Value, Error States

- **Success** – number of ready descriptors.
  - `readfds`, `writefds` and `exceptfds` are modified to reflect the file descriptors which are ready
- **Time expired** – returns 0 (errno set to `EINTR`)
- **Failure** – returns -1
  - `EBADF`, `EINTR`, `EINVAL`, `ENOMEM`
Pseudo-code of Usage (1/2)

- \( nfds = 0 \)

- Initialize readfds, writefds, exceptfds using FD_ZERO

- Add the listener socket to readfds using FD_SET and update nfds

- For each active connection
  - If connection has available read buffer, add fd to readfds (FD_SET)
  - If connection has available write buffer, add to writefds (FD_SET)
  - Add to exceptfds (FD_SET) – not really needed for this project.
  - Update nfds to ensure that the fd falls in the range
Pseudo-code of Usage (2/2)

- `select_return = select(nfds, readfds, writefds, exceptfds, NULL)`
- If `select_return > 0`
  - Handle exceptions if any fd in `exceptfds` is set to 1 `(FD_ISSET)`
  - Read data from connections for which fd in `readfds` is set to 1 `(FD_ISSET)`
  - Write data from connections for which fd in `writefds` is set to 1 `(FD_ISSET)`
  - If listener socket is set to read, `accept` and handle new connection.
- Else handle error states
Other Solutions!

- Don’t settle for solutions 1 & 2...
- One-fork-per-client, preforking
- One-thread-per-client, prethreading
- Check chapter 30 of “UNIX Network Programming: The Sockets Networking API”
Remember

- Code quality
- Code documentation
- Robustness
  - Handle all errors
  - Buffer overflows
  - Connection reset by peer
Debugging

• What can go wrong?
• How can we avoid errors?
• What tools are available to debug errors?
  – Valgrind
  – GDB
  – strace
What can go wrong?

• Project 1: HTTP server, what are our components?
  – managing connections (e.g., sockets)
  – handling clients (e.g., client pool)
  – handling data (e.g., buffers and strings manipulation)
  – HTTP protocol (RFC 2616)

• What kind of errors can we have? (2 major types)
  – Logical error vs. fault (crashing)
Error Types and Project 1

• What logic errors do you need to be careful of?
  – HTTP protocol following the RFC
  – Handling socket information properly

• What faults do you need to be careful of?
  – Memory copying (e.g., buffer to buffer)
  – String manipulation (e.g., handling client messages)
  – Array accesses (e.g., your client pool)
  – Socket descriptors
Save yourself a headache!

• First and foremost: practice *smart* programming to avoid faults.

• **CHECK RETURN CODES!**
  – Bad: `read(fd, &buffer, nbtr);`
  – Good: `if((nbytes=read(fd, &buffer, nbtr))===-1)`

• Use *safe* functions: `snprintf(good)` vs. `sprintf(bad)`

• Check pointers before use: `if(clientfd!=NULL) {
      ... 
  }`
Save yourself a headache!

- Make clean: you might not have compiled a part of your code
- Check compiler warnings: always use –Wall, -Wshadow, -Wunreachable-code
- Examine the most recent changes
- Eliminate hypothesis
Outline

• What can go wrong?
• How can we avoid errors?
• What tools are available to debug errors?
  – valgrind
  – strace
  – GDB
Reality: Errors WILL Happen

• We are all human, bugs will occur
  – Goal: find and *terminate* them as fast as possible

• *Don’t*: toss printf()’s everywhere and hope for the best, this takes a long time

• *Do*: use a great set of tools for debugging
  – Saves time → saves points!
  – Saves headache → saves sanity!
DEBUG Macros

• Different levels of debug verbosity
• Control by changing a single parameter

```c
#ifndef DEBUG
    #define PRINT(arg)
#else
    #define PRINT(arg) printf arg
#endif

PRINT( ("x = %f, y = %f\n", x, y) );
```
strace

• Lists each system call, its parameters and results

• strace <executable> [args]

  socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 3
  bind(3, {sa_family=AF_UNSPEC,
           sa_data="\0\0...\0\0\0"}, 16) = 0
  exit_group(0) = ?

Process 24787 detached
GDB: GNU Project Debugger

• The **best** debugging tool for your projects!

• You can step through your program, line by line and monitor *any* memory!

• Seriously, it doesn’t get any better than this
How to use GDB

• Two major ways:
  • Read a core dump
  • Step through a program

• Getting a segfault and just want to determine where it happened?
  • Get a core file, run: `ulimit –c unlimited`
  • Core file contains in-memory state of the program when it crashed
  • Cause the program to segfault

• **MUST enable –ggdb flag when compiling**
GDB: Reading a core file

• Enable core dumping and run:
  
  $ ulimit -c unlimited
  $ ./cache_sim config.example0 < trace.example0
  
  ....
  Segmentation fault (core dumped)

• Open the core in GDB:
  
  $ gdb cache_sim core
  
  ...  
  #0  0x08049bae in memory::load (...,...) at cache_sim.cc:252
  252    if(!d_tag_store[i][index].valid) {
    (gdb) backtrace
    #0  0x08049bae in memory::load (...,...) at cache_sim.cc:252
    #1  0x0804a3e2 in handle_load_reference (... at cache_sim.cc:366
    #2  0x0804b63e in main (...,...) at cache_sim.cc:562
  
  Function where the segfault occurs (load)
  
  Line where the segfault occurs
  
  How we got there
GDB: Useful commands

• Useful commands for you to know:
  – Start the program: `run <arg1> <arg2> ...
  – Create breakpoint: `break <line> OR break <function>
  – Goto next line: `next
  – Step into a function: `step
  – Check a variable value: `print <variable name>
  – Display a variable value: `display <variable name>
Example – Buggy program

```c
int a (int *p);

int
main (void)
{}
  int *p = 0; /* null pointer */
  return a (p);
{}

int a (int *p)
{}
  int y = *p;
  return y;
{}
[Source - http://www.network-theory.co.uk/articles/gccdebug.html]
Using gdb

- $ gcc -Wall -ggdb null.c
- $ ./a.out Segmentation fault (core dumped)
- $ gdb a.out core

Core was generated by './a.out'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0 0x080483ed in a (p=0x0) at null.c:13
  13    int y = *p;
(gdb)
Using gdb

(gdb) print p
$1 = (int *) 0x0

(gdb) backtrace
#0  0x080483ed in a (p=0x0) at null.c:13
#1  0x080483d9 in main () at null.c:7