CSci 4061
Introduction to Operating Systems

Programs in C/Unix
Structure of a C program

- A C program consists of a collection of C functions, structs, arrays, typedefs.

- One function must be called `main`:
  - `int main (int argc, char *argv[])`
  - `argc` is # of command-line args (>= 1)
  - `argv` is an array of `argc` “strings” (incl. program)

- There is no string type in C! These are “close”
  - `typedef char *string;`
  - `typedef char [] string;`
Structure of a C program (cont’d)

• To run a program you simply type its executable name
  • To pass arguments you provide them on the command-line
• I have an exec. program called mine
• In my login shell, I type:
  • `shell> mine -c 10 2.0 (.%/mine if paranoid)`

| argv  | \n\hline
| 0     | \texttt{'m'} \texttt{'i'} \texttt{'n'} \texttt{'e'} \texttt{\textbackslash 0}' \hline
| 1     | \texttt{\textbackslash 'c'} \texttt{\textbackslash 0}' \hline
| 2     | \texttt{\textbackslash '1'} \texttt{\textbackslash 0'} \texttt{\textbackslash 0}' \hline
| ...   | \hline

C arrays start at 0
\texttt{argc} = ?
Simple Exercise

• What will be the value of argc and argv inside the program argtest:
  
  ./argtest “–x $PRINTER a b c x”

• Why are command-line args useful?
  • Without them, how would you get inputs=>program?

• Really useful call:
  
  x = atoi (argv[i]); // string to int
Structure of a C program (cont’d)

• Functions may come from multiple source files and libraries
  • (e.g. /lib or /usr/lib)

• Types/constants/prototypes are usually defined in header files (.h)

• Analogous to class definitions in C++ or Java
Program Structure: Style #1

• A C program contains a set of “modules”
  • Separate files, separately compiled
  • Each contains functions
  • Common types, data-structures, function prototypes are in header files

```c
foo.h
#define MaxTokens 10
int sortit(char a[100]);

foo.c
#include <foo.h> // like a macro
...
int sortit(char a[100]); {
  ...
}
```

```c
other.c
#include <foo.h>
int main() {
  ...
  y = sortit(...);
  ...
}
```

Link in foo.o (object file)
Program Scoping: Global

// allocated and available only to the file containing
// this declaration

static int foo;

// allocated, global and exportable

int bar;

// allocated elsewhere, must be linked in

extern int baz;

Global variables get deallocated when?
Program Scoping: Local

int main (...) {
    int a; // allocated new on the stack each call
    static int b=0; // allocated once, value stays!

    b++;
    b++;
    ...
    ...
}

Local variables get deallocated when?
What about statics?
Process Environment

• Way to embed system-specific info into prog.
  • e.g. program to always write to user’s HOME directory; send output to user’s default PRINTER
  • <name=value> pairs defined by your SHELL

char *getenv (const char *name);

getenv ("HOME") -> /home/fac04/jon
Libraries and Include Files

• When you invoke a function, the compiler needs a prototype for it
  • e.g. if you want to use fopen

```c
#include <stdio.h>
FILE *f;
F = fopen ("/usr/jon/f.dat", "r");
```
• Function prototype is in `<stdio.h>`
• Usually functions themselves are in standard libraries, if NOT you must use:

  `-l<library-name>` when you compile

For example, `-lpthread`

`stdio libraries (and others) linked in by default (libc.a)`
Compiling

• On most Unix systems, the compiler is gcc
  gcc -o foo foo.c
• Compiles into a single executable named foo
  To run, shell>foo

• Multiple modules
  gcc -c foo1.c (produces foo1.o)
  gcc -c foo2.c (produces foo2.o)
  gcc -o foo foo1.o foo2.o -lpthread
  gcc -v -o foo foo1.o foo2.o // verbose
  gcc -o foo foo1.c foo2.c // ok, too
Error Handling: Style #2

```c
#include <unistd.h>

// -1 returned if failure; sets errno (extern int)
int close (int fildes);

if (close (fildes) == -1)
    perror ("close failed ..."); // uses errno
```

GOOD style to check for errors in system calls!
The Ubiquity of 0

• In C and Unix, 0 is used a lot:
  • NULL and 0 are used interchangeable
  • #define NULL 0

• Boolean not
  • #define FALSE 0
Memory Allocation

• The primary dynamic allocation function
  • `void *malloc (size_t size)`
  • Allocates size bytes, returns ptr (address) or NULL if memory not available

```c
ptr1 = malloc (5);
ptr2 = (my_t *) malloc (sizeof (my_t));
```

Casting: keeps compiler happy

Handy! Returns size of a variable or type in bytes

Release allocated memory
`void free (void *ptr_var);`

VERY error-prone!
Memory Leakage

• Your program leaks if its memory usage grows w/o bound
  • For what kind of program is this a problem?

• Happens if you forget to free memory not needed anymore

• Moral: don’t lose ptr to allocated memory!
  \[
  a = \text{malloc} \ (100000); \\
  a = 10;
  \]

• On program exit, OS reclaims memory
Buffer Overflow (Attack)

• Buffer overflow

```c
void func (char *buffer, ...) {
    char local[5];
    ...
    strcpy (local, buffer);
    ...
}
```

You call it with a big string:

```c
func ("sjfh28&54NASTY_CODEw992385jsdh8");
```
Buffer Overflow (cont’d)

• You will clobber the stack
  • This will overwrite local variables and possibly the return address of the call!
  • If you are lucky the program just dies

• It can get worse ... attack
  • Suppose overwrite causes return address to be a location that now contains NASTY_CODE!

• Solutions?
  • Use strncpy and/or check length
  • Use non-ptr language (Java)
C crashes

• C program crash
• Segmentation violation
  • Program attempts to access memory outside its boundary

  ```c
  int a[10];
  A[10] = 3;
  A[-2] = 5;
  To catch this you can run splint
  ```

• Illegal instruction
  • Program attempted to execute an undefined or privileged machine instruction
Unix/C tools: Makefiles

• Make builds programs by processing a dependency tree
  • It is a set of rules that describes dependencies and how to resolve them
  • Uses time-stamps
    foo.o: foo.c foo.h
    gcc -c foo.c

• Each action line begins with a TAB
• Default makefile is called makefile
Make is your friend

• Be aggressive with recompilation

• Strange bugs can be resolved by recompilation
  
  make clean
  make all
Debugging

• Debugging 101: the `printf` and debugging levels

```c
#ifdef DEBUG
    printf (stderr, "A=%d\n", A);
#endif
```

```
gcc -o foo foo.c -DDEBUG
```

Can set multiple levels: DEBUG1, DEBUG2, ...

Several preprocessor directives:

```c
#include, #define, #ifdef, #ifndef
```
Unix/C tools: Debugging

• Use gdb: GNU debugger
  • There are many others
  • Set breakpoints, look at vars, step, trace
  • Recommend you learn this!

```bash
gcc -g -o crash crash.c
```
Unix/C tools: System Monitoring

- **ps**: tells you the state of a running program
  - **R**: running, if always R, maybe an infinite loop
  - `ps -lu <uid>`
    - Shell commands have many flag options

- **top**: shows complete information and dynamically updates
Next Time

• We will look at processes

• Read Chapter 2, 3 R&R, opt: Chapter 2 MOS or Chapter 3 S&G