CS 354 - Machine Organization & Programming
Tuesday, December 3, 2019

Project p6 (4.5%): DUE at 10 pm on Saturday, December 14th

Homework hw8 (1.5%): Due at 10 pm on Friday, December 6th

Last Time

Meet Signals
Three Phases of Signaling
Processes IDs and Groups
Sending Signals
Receiving Signals

Today

Issues with Multiple Signals

Forward Declaration
Multifile Coding
Multifile Compilation
Makefiles

Next Time

Bring your devices to fill out online course evaluation
Linking and Symbols
Read: B&O 7.3 - 7.6

Issues with Multiple Signals

What? Multiple signals of the same type as well as those of different types
can be sent at the same period that other signals are sent
and even where a signal handler is running.

Some Issues

→ Can a signal handler be interrupted by other signals?
yes, but linear signal of same type as running handler don't interrupt.
instead, they become pending

❋ *Block any signals* you don't want to interrupt your handler
sigemptyset(&sa.sa_mask); // block signals set to 1 in sa_mask.

→ Can a system call be interrupted by a signal? yes, two ----

slow system calls take a potentially long time read(), printf()

such sys calls return immediately with an error.

sa.sa_flags = SA_RESTART;

Note: sleep() can not be restarted.

→ Does the system queue multiple standard signals of the same type for a process? No
bit vector can't keep a count of duplicates
Duplicates signals that are pending are ignored.

❋ *Your signal handler shouldn't assume* that a signal was sent only once.

Real-time Signals

Linux has 33 additional app define signals.

- They can include an integer or pointer in their message.
- Multiple signals of same type are queued in order received.
- Multiple signals of different types are received.
from low to high signal number

Forward Declaration

What? Forward declaration tells the compiler about certain attributes of an identifier before it is fully defined.

❋ *C requires that an identifier* be declared before it is used.

Why?

- one pass compilers (gcc) can then ensure the identifier exists and is correctly used.
- large programs can be divided into seperate functional units that can be independently compiled
- mutual recuration is possible

Declaration vs. Definition

<u>declaring</u> tells the compiler about

variables: name & type

functions: return type, name, param types

<u>defining</u> provides the full details

variables: where in mem it's located

functions: function's body

❋ *Variable declarations* usually both declare & define

```
void f(){
   int i = 11;
   static int j;
```

(`int i = 11;`: declare, define, & initialized. `static int j;`: - - - - - - -, no initialization.)

❋ *A variable is proceeded with* extern is not defined (alloc's mem)

Multifile Coding

What? Multifile coding divides a program into functional units, each coded with its own header file and source file.

Header File (finename.h) - "public" interface

contain things you intend to share mainly func declarations. and also declaration and definations of types, constants, macros.

recall **heapAlloc.h** from project p3:

```
#ifndef __heapAlloc_h__
#define __heapAlloc_h__

int   initHeap(int sizeOfRegion);
void* allocHeap(int size);
int   freeHeap(void *ptr);
void  dumpMem();

#endif // __heapAlloc_h__
```

public Func, defn

- *An identifier* can be defined only in the global scope

#include guard: prevent multiple inclusions of same header file without it a header file with definitions would result in linker errors

Source File (filename.c) - "private" implementation

must includes definitions of things declared in its header file also contains additional things you don't intend to share.

recall **heapAlloc.c** from project p3:

```
#include <unistd.h>
. . .
#include "heapAlloc.h"

typedef struct blockHeader {
   int size_status;
} blockHeader;

blockHeader *heapStart = NULL;

void* allocHeap(int size) { . . . }
int   freeHeap(void *ptr) { . . . }
int   initHeap(int sizeOfRegion) { . . . }
void  dumpMem() {. . . }
```

source file includes its header

private type decl & defn

private global var decl & defn.

"private" Func Defns.

note: Funcs can now be in any order since declared in

Multifile Compilation

gcc Compiler Driver

directs all the tools needed to create an executive from source code

main.c → preprocessor	cpp	remove comments, does preprocessor directives	
main.i → compiler	cc	translate source code to the assembly,	
main.s → assembler	as	translate to machine code, rof	
main.o → linker	ld	combine ROFs & SOFs into an executable EOF	
↳ main			

Object Files

contain binary code & binary data

relocatable object file (ROF) produce by assembler
can be combined with other ROF's to produce EOF.

executable object file (EOF) produced by linker
can be loaded into mem and run

shared object file (SOF) produced by assembler can be loaded into mem and linked dynamically during load or run time.

Compiling All at Once

```
gcc align.c heapAlloc.c  -o align
```

cpp → cc → as → ld
to produce EOF named align

Compiling Separately

```
gcc -c align.c
gcc -c heapAlloc.c
gcc align.o heapAlloc.o -o align
```

cpp → cc → as to produce ROF named align.o

heapAlloc.o

ld to produce EOF named align.

❋ *Compiling separately is* more efficiently and easier to manage.

Makefiles

What? Makefiles are

- text files named makefile that have rules
- used with "make" command

Why?

- convenience - specifies how to build a prog
- efficiency - only builds what's necessary using rules & file dates.

Rules form

<target> : <files target depends on>
<TAB> <command(s) for making target>

Example

```
#simplified p3 Makefile
align: align.o heapAlloc.o
<TAB>gcc align.o heapAlloc.o -o align
align.o: align.c
<TAB>gcc -c align.c
heapAlloc.o: heapAlloc.c heapAlloc.h
<TAB>gcc -c heapAlloc.c
clean:
<TAB>rm *.o
<TAB>rm align
```

] Rule 1: how to make EOF
] Rule 2: how to make align.o EOF
] Rule 3: how to make heapAlloc.o EOF.
] Rule 4: cleans up directly

Using

```
$ls
align.c  Makefile  heapAlloc.c  heapAlloc.h
$make
gcc -c align.c
gcc -c heapAlloc.c
gcc align.o heapAlloc.o -o align
$ls
align  align.c  align.o  Makefile  heapAlloc.c  heapAlloc.h  heapAlloc.o
$rm heapAlloc.o
rm: remove regular file 'heapAlloc.o'? y
$make
gcc -c heapAlloc.c
gcc align.o heapAlloc.o -o align
$make heapAlloc.o
make: 'heapAlloc.o' is up to date.
$make clean
rm *.o
rm align
$ls
align.c  Makefile  heapAlloc.c  heapAlloc.h
```

Does R1 by default
no align.o so fire R2
no heapAlloc.o so fire R3
now R1 can fire

] make only does what's needed to build align.