CS 354 - Machine Organization & Programming
Tuesday April 25 , and Thursdat April 27th, 2023

Homework hw7: DUE on or before Monday Apr 24th

Homework hw8: DUE on Monday May 1st

Homework hw9: DUE on Wednesday May 3rd

Project p6: Due on last day of classes, May 5th. **Please complete p6 by Friday of this week as labs are very busy last week of classes**.
If you do plan on getting help during last week of classes, be sure to bring your own laptop in case there is no workstation available.

Last Week

Pointers Function Pointers Buffer Overflow & Stack Smashing Flow of Execution Exceptional Events Kinds of Exceptions	Transferring Control via Exception Table Exceptions/System Calls in IA-32 & Linux Processes and Context User/Kernel Modes Context Switch Context Switch Example

This Week

Meet Signals Three Phases of Signaling Processes IDs and Groups Sending Signals Receiving Signals	Issues with Multiple Signals Forward Declaration Multifile Coding Multifile Compilation Makefiles
Next Week: Linking and Symbols B&O 7.1 Compiler Drivers 7.2 Static Linking 7.3 Object Files 7.4 Relocatable Object Files 7.5 Symbols and Symbols Tables 7.6 Symbol Resolution 7.7 Relocation	

Meet Signals

❋ *The Kernel uses signals to notify User processes of exceptional events.*

What? A *signal* is

Linux:

$kill -l

signal(7)

Why?

- 1.
  2.
-
-

Examples

1. divide by zero

exception interrupts to kernel handler

- kernel signals user proc with

2. illegal memory reference

exception interrupts to kernel handler

- kernel signals user proc with

3. keyboard interrupt

- ctrl-c interrupts to kernel handler which

- ctrl-z interrupts to kernel handler which

Three Phases of Signaling

Sending

- when the kernel

-

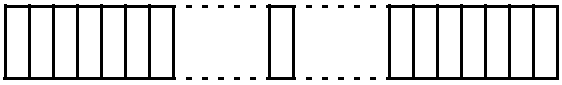
Delivering

when the kernel

pending signal

-

bit vectors



-

Receiving

when the kernel

-

-

blocking

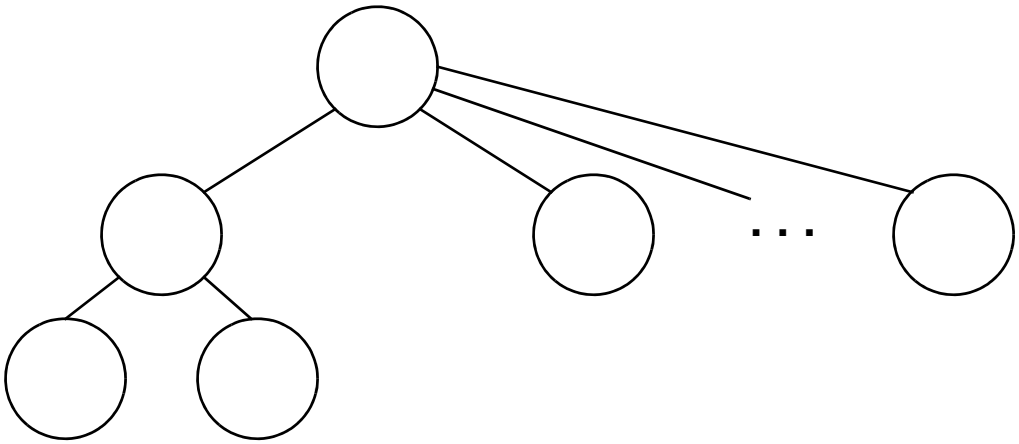
-

-

Process IDs and Groups

What? Each process

-
-



Why?

How?

Recall:ps

jobs

getpid(2)getpgrp(2)

```
#include

pid_t getpid(void)

pid_t getpgrp(void)
```

Sending Signals

What? A signal is sent by the kernel or a user process via the kernel

How? Linux Command

kill(1)

kill -9 <pid>

→ What happens if you kill your shell?

How? System Calls

kill(2)

killpg(2)

```
#include <sys/types.h>
#include

int kill (pid_t pid, int sig)
```

alarm(2)

```
#include

unsigned int alarm(unsigned int seconds)
```

Receiving Signals

What? A signal is received by its destination process

How? Default Actions

- Terminate the process
- Terminate the process and dump core
- Stop the process
- Continue the process if it's currently stopped
- Ignore the signal

How? Signal Handler

1.
   -
   -
2.
   -

   ~~signal(2)~~
   sigaction(2)

Code Example

```
#include <signal.h>
#include ...
#include <string.h>

void handler_SIGALRM() { ... }

int main(...) {
```

Issues with Multiple Signals

What? Multiple signals of the same type as well as those of different types

Some Issues

→ Can a signal handler be interrupted by other signals?

❋ *Block any signals*

→ Can a system call be interrupted by a signal?

slow system calls

→ Does the system queue multiple standard signals of the same type for a process?

❋ *Your signal handler shouldn't assume*

Real-time Signals

-
- Multiple signals of same type
- Multiple signals of different types

Forward Declaration

What? Forward declaration

❋ *Recall, C requires that an identifier*

Why?

-
-
-

Declaration vs. Definition

declaring

variables:

functions:

defining

variables:

functions:

❋ *Variable declarations*

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

❋ *A variable is proceeded with*

Multifile Coding

What? Multifile coding

Header File (finename.h) - “public” interface

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__
```

- *An identifier*

#include guard:

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

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() {. . . }
```

Multifile Compilation

gcc Compiler Driver

preprocessor

compiler

assembler

linker

Object Files

relocatable object file (ROF)

executable object file (EOF)

shared object file (SOF)

Compiling All at Once

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

Compiling Separately

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

- *Compiling separately is*

Makefiles

What? Makefiles are

-
-

Why?

-
-

Rules

Example

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

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
```