CS 354 - Machine Organization & Programming
Tuesday April 23 , and Thursday April 25th, 2024

Homework hw7: DUE on or before Monday Apr 21st

Homework hw8: DUE on Monday Apr 29th

Homework hw9: DUE on Wednesday May 1st

Project p6: Due on last day of classes, May 3rd. **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 a small message from the kernel (usually an int)

Linux: 30 signals exist

$kill -l lists the 30 signals

signal(7) man signal

Why?

- so the kernel can notify us of events
  1. "hardware" error, ie divide by 0 (low-level)
  2. setting an alarm,
     eg ask user for input, wait for that value, set up signal handler, set alarm to go off in 10s (high-level)
- communicate among processes
- implement high-level exceptional flow control

Examples

1. divide by zero

exception #0 interrupts to kernel handler

- kernel signals user proc with SIGFPE #8

2. illegal memory reference

exception #13 interrupts to kernel handler

- kernel signals user proc with SIGSEGV #11

3. keyboard interrupt #1 — 8259 programmable interrupt controller

- ctrl-c interrupts to kernel handler which

  generates SIGINT #2 — default to kill the program

- ctrl-z interrupts to kernel handler which

  generates SIGSTP #20 — default to SUSPEND the program (in background and not running)

kill -9 is an unabortable kill
kill -2 equivalent to sending Ctrl-C signal to other window
kill %<JOB ID>

Three Phases of Signaling

Sending

- when the kernel is always involved in sending

- kill -9 pID — directed at a process (or process group)

Delivering

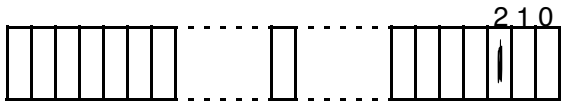
when the kernel records the signal for a process

pending signal are signals that are delivered but not received

- each process has a bit vector

bit vectors are 32-bit ints

if type ctrl-c, set bit 2 to 1 since SIGINT is #2



- k-th bit is set for the k-th signal

Receiving

when the kernel calls your signal handler

- when is returning control to your process

- will call signal handler instead

blocking prevent signals from being received

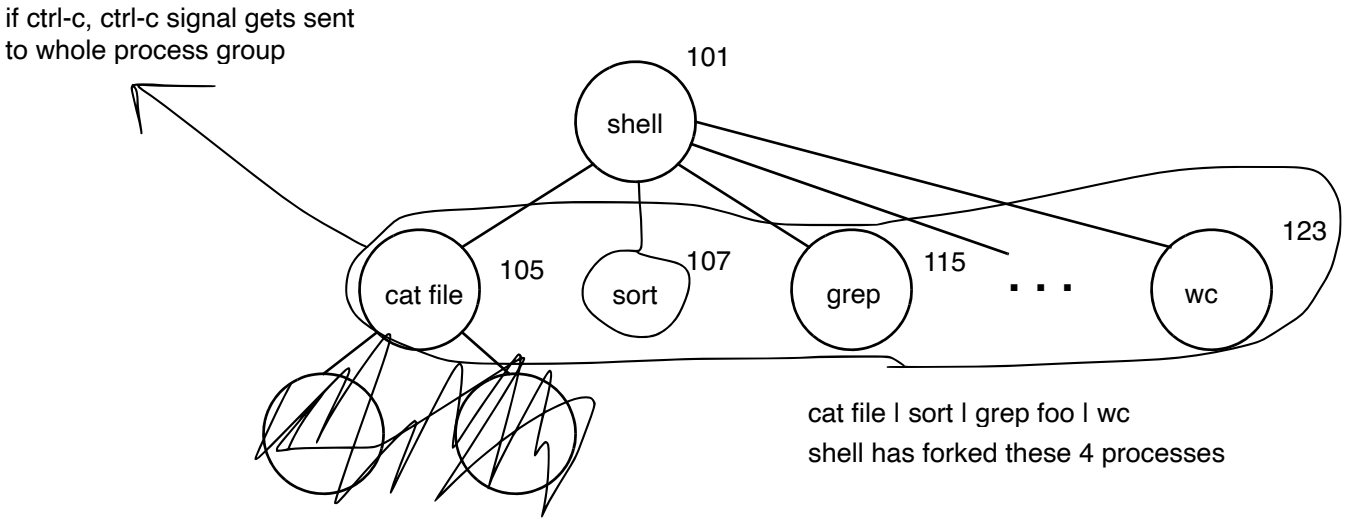
- control what signals are delivered

- another bit vector — if k-th bit is 1 then k-th interrupt is blocked

Process IDs and Groups

What? Each process

- has a process id pid (32bit number) — unique identifier for that process running
- process group — process belongs to only one group at a time



Why?

- manage group like chained commands

How?

when in shell and wants to run new program, eg sleep,
shell forks process, fork() returns twice — exact duplicate copy of running program, pid of newly or 0, child is return value of 0

Recall:ps ps -u | grep sleep

jobs looks for suspended processes

```
pid = fork()
if (pid == 0) {
  // in child
} else {
  // in parent
}
```

getpid(2)getpgrp(2)

```
#include

pid_t getpid(void)

pid_t getpgrp(void)
```

shell forks, makes 2 copies, then executes execve()
eventually SIGCHLD delivered to parent to signal child has gone away but not entirely gone, if parent interested in exit code of child, little bit of state of running program such a (zombie program)

Sending Signals

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

can send by cmd line or via sys call

How? Linux Command

kill(1)

kill -9 <pid> unconditional unblockable kill

→ What happens if you kill your shell?

How? System Calls

kill(2) man 2 kill

killpg(2)

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

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

needs permission to send signal

alarm(2) man 2 alarm — schedules an alarm or call to signal handler in certain number of seconds

```
#include

unsigned int alarm(unsigned int seconds)
```

want to wake up in the future

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

if SIGINT, sets bit handler, kernel will call signal handler, if another SIGINT,
kernel will queue this up, waits to deliver same type
if diff type, kernel will interrupt

Some Issues

→ Can a signal handler be interrupted by other signals? YES

marked as pending

❋ *Block any signals* where you don't want the handler called

struct sigaction sa
sa.sa_mask is used to tell kernel which signal to be blocked/enabled
sigemptyset(&sa.sa_mask), blocks all signals
sigfillset(&sa.sa_mask), enables all signals
sigaddset(&sa.sa_mask, signum) and sigdelset(&sa.sa_mask, signum)

can set empty to block all and add those signums i do not want to block

→ Can a system call be interrupted by a signal? YES

slow system calls

result = read() ==> result = -1 tells us there is an error
errno = EINTR (read failed because of interrupted)

sa.sa_flags = SA_RESTART; (will restart system calls that can be restarted)
sleep cannot be restarted

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

NO, bit vector cant count

❋ *Your signal handler shouldn't assume* that it will be called only once

Real-time Signals

Linux has 33 additional siganls

- can take *ptr, int as input
- Multiple signals of same type will be delivered
- Multiple signals of different types devliered but received low to high

kernel uses queue instead of bit vector here

Forward Declaration

What? Forward declaration tells the compiler about something it needs

wouldnt know what printf is without include std.io, printf is forward declared

❋ *Recall, C requires that an identifier* be declared before use

Why?

- C is a one pass compiler
- want to break up a program — linux kernel is about 100,00 .c files
- recursion

funcA -> funcB -> funcA problem!! funcB wont know about funcA

forward declaration eg: int foo (char *);

Declaration vs. Definition

declaring telling compiler about something

variables: name and type

functions: name, return type, params

defining

variables: where in memory it exists

functions: is the body of the function

❋ *Variable declarations* define and declare variables at the same time

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

on stack frame somewhere and points to address which has value 11

j can only exist in function, but telling compiler there is only one j reference => in .bss

❋ *A variable is proceeded with* extern int errno;

Multifile Coding

What? Multifile coding modularization

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

will contain declarations of functions

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__
```

#pragma once does same thing as guard

- *An identifier*

#include guard: prevents multiple inclusions

header file contain declarations, source file contains defintions

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

definitions

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

- text files that describe how to build sth
- make Makefile

Why?

- convenience
- efficiency

Rules

target

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
```

dependencies

tab

if target is older than dependencies,
need rebuild since dependencies are newer

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
```