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 a small message sent to process via kernel

Linux: hardware (HW) has 30 standard signal types, each with unique non-negative ID#

$kill -l see a list of signal names and numbers

signal(7) see man 7 signal

Why?

- So kernel can notify user process about:
  1. low-level HW exceptions 0-31
  2. high-level evens in kernel or user processes
- enable user processes to communicate with each other
- to implement high-level software form of exception handling

Examples

1. divide by zero

exception #0 interrupts to kernel handler

- kernel signals user proc with SIGFPE #8 (floating point exception)

2. illegal memory reference

exception #13 interrupts to kernel handler

- kernel signals user proc with SIGSEGV #11

3. keyboard interrupt

- ctrl-c interrupts to kernel handler which signals SIG INT #2
  terminates foreground process (fg) by default
- ctrl-z interrupts to kernel handler which signals SIGTSTP #20 (suspend)
  suspends foreground process by default

Three Phases of Signaling

Sending

- when the kernel exception handler runs in response to an exceptional event or signal from user process

- is directed to a destination process

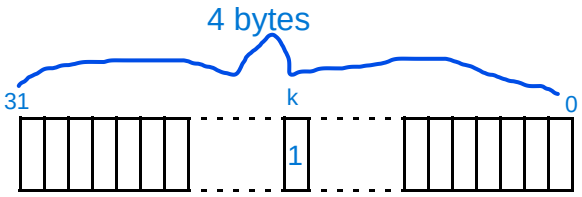
Delivering

when the kernel records a sent signal for its destination process

pending signal is delivered but not received

- each process has a bit vector for recording pending signals

bit vectors kernel data structure where each bit has a distinct meaning



- bit k is set to 1 when that signal is pending (delivered)
  bit k is set to 0 when that signal is received

Receiving

when the kernel causes destination process to react to pending signal

- Happens when kernel transfers control back to process

- multiple pending signals are in order from low to high signal numbers
  lower numbers have higher priority

blocking prevents a signal from being received

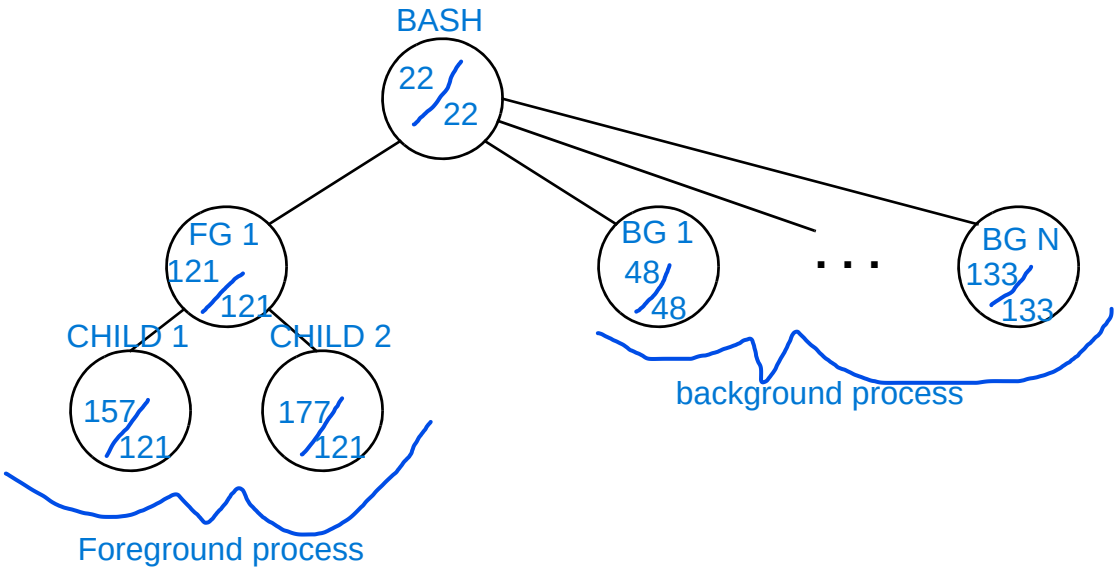
- enables process to control which signals it pays attention to

- each process has a second bit vector for blocking signals
  4 bytes 1 = block

Process IDs and Groups

What? Each process

- is identified by a PID (a process ID number)
- and belongs to one group identified by PGID (process group ID number)



Why?

numbers pid are easier to manages than names
pgid

How?

Recall:ps list running processes with pid & pgid

jobs list processes using simple job number

getpid(2) getpgrp(2)

```
#include <unistd.h>

pid_t getpid(void)
```
returns pid

```
pid_t getpgrp(void)
```
returns pgid

Sending Signals

What? A signal is sent by the kernel or a user process via the kernel from command line
or in a program using system calls

How? Linux Command

kill(1) man 1 kill sends signal from command line
to specific pid

kill -l

kill -9 <pid> 9 SIGKILL 2 SIGINT 20 SIGTSTP

→ What happens if you kill your shell? log out, same as exit

How? System Calls with a program

kill(2) man 2 kill send sig from calling to another process

killpg(2) send signal to all processes in process group

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

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

pid - process id of recipient & signal
sig - signum

returns 0 on success
-1 on error

alarm(2) man 2 alarm
sets alarm that will deliver signal SIGALRM

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds)
```

until alarm is triggered

sec remaining if prev alarm
otherwise returns 0

Receiving Signals

What? A signal is received by its destination process by doing default action
or executing code specified by sig handler

How? Default Actions

- Terminate the process SIGINT #2 ctrl-c
- Terminate the process and dump core SIGSEGV #11
- Stop the process SIGSTP #20 ctrl-z
- Continue the process if it's currently stopped SIGCONT #18
- Ignore the signal SIGWINCH #28

(not the only signals
with these default behaviors)

How? Signal Handler

1. code a signal handler
   - looks like a regular function but is called by kernel
   - should not make unsafe system calls (like printf)
2. register the signal handler
   - catch one or more signals

~~signal(2)~~ DON'T USE!
sigaction(2) POSIX for examining and changing a signal's action

Code Example

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

void handler_SIGALRM() { ... }

int main(...) {
```

code to handle alarm

function pointer

```
    ...
    //Register our signal handler
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = handler_SIGALRM;
    if (sigaction (SIGALRM, &sa, NULL) != 0) {
        printf("Error binding SIGALRM handler\n");
        exit(1);
    }
```

Issues with Multiple Signals

What? Multiple signals of the same type as well as those of different types
can be sent during same period that other signals are sent
and even while sig. handler is running

Some Issues

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

❋ *Block any signals* that you don't want to interrupt your sig handler
sigemptyset(&sa.sa_mask); // blocks all
sigfullset(&sa.sa_mask); // enable all (unblock all)
sigaddset, sigdelset, sigismember(&sa.sa_mask, signum);

→ Can a system call be interrupted by a signal? yes, for...

slow system calls which potentially take a long time

such system calls return immediately EINTR

sa.sa_flags = SA_RESTART; //attemt to restart

NOTE: sleep() can not be restarted

→ Does the system queue multiple standard signals of the same type for a process? NO

the bit vector indicates pending or NOT pending, THERE IS NO COUNT

❋ *Your signal handler shouldn't assume* that a signal was sent only one time (except in p6)

Real-time Signals

Linux has 33 additional application defined signals

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

//all necessary content for p6 now covered, also in textbook

Forward Declaration

What? Forward declaration tells compiler about certain attributes of identifier before it's fully defined

- *Recall, C requires that an identifier* be declared before used

Why? gcc is a

- one pass compiler (gcc) can ensure identifier exists and is correctly used
- large programs can be divided into separate functional units and compile separately
- mutual recursion is possible. A calls B
  B calls A

Declaration vs. Definition

declaring tells compiler about

variables: name and type extern int g;

functions: return type, name, parameter types & order

defining provides full details

variables: where in memory it's allocated

functions: function body is defined

- *Variable declarations* usually both declare and define

```
void f(){
   int i = 11;    // declare, define, & initialize
   static int j; // declaration and definition (not initialized)
```

- *A variable is proceeded with* extern is NOT DEFINED

extern char * Title; //declaration only

Multifile Coding

What? Multifile coding divides a program into functional units

each unit has its own header and source file

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

contains thing intend to share

mainly function declarations

but also can have types, constants, macros

p3Heap.h

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__
```

function declaration

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

#include guard: prevents multiple inclusions of the same header file

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

must include definitions of things in header file

p3Heap.c

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() {. . . }
```

"private" defns

Multifile Compilation

gcc Compiler Driver directs tools needed to build executable

Object Files binary code & data segment

relocatable object file (ROF) can be combined by the linker with other ROFs and SOFs to produce an EOF

executable object file (EOF) can be loaded into mem and "run"

shared object file (SOF) can be loaded into memory (DLL) or statically linked into EOF

Compiling All at Once

-m32 -Wall -std=gnu99

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

src

EOF

Compiling Separately

```
gcc -c align.c
```
compile to align.o (ROF)

```
gcc -c heapAlloc.c
```
compile o heapAlloc.o (ROF)

```
gcc align.o heapAlloc.o -o align
```
link to align (EOF)

- *Compiling separately is* more efficient and easier to manage

Makefiles SKIPPED FOR TIME

What? Makefiles are

-
-

Why?

-
-

Rules

Example

```
    #simplified p3 Makefile
rule 1 align: align.o heapAlloc.o
        gcc align.o heapAlloc.o -o align
rule 2 align.o: align.c
        gcc -c align.c
rule 3 heapAlloc.o: heapAlloc.c heapAlloc.h
        gcc -c heapAlloc.c
rule 4 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
```