Recitation 9
Shell Lab

Grant Skudlarek

News

- Welcome back
- Shell lab due next Thursday, at 11:59 PM

Agenda

- **Problem 1 and 5**
- Processes and Signals
- Shell lab

Problem 1

A.Consider the C function below:

```
int func( int x )
{
   int y;
   y=(x<<31)>>31;
   return y;
}
```

When “func(7)” is called (i.e.x=7) on IA32, what is the return value? **-1**

Problem 1

B. The expression x*x >= 0 holds uniformly for

(a) both signed and unsigned integers

(b) signed integers, but not for unsigned integers

(c) unsigned integers, but not for signed integers

(d) neither signed nor unsigned integers

C.What is the evaluation result of expression 1110_2ˆ1010_2?

(a) 1111_2

(b) 1010_2

(c) 0100_2

(d) 0110_2

Problem 1

D. Assume that you are working on a machine with 8-bit ints and arithmetic right shifts. Further assume that variable x is a signed integer represented in two's complement. Match each of the descriptions on the left with 0, 1, or more snippets of code on the right. Write the letter of each matching snippet in the blank under the description. Some code snippets may not match any of the descriptions.

(1) 13*x **(F)**
(2) Absolute value of x. (**H)**
(3) Round x down to nearest power of 2. (**NONE)**
(4) Round x to a multiple of 16 **(E)**
(5) x < 0 **(B) (C)**
(6) Swap most significant and least significant **four bits** of x. **(D)**

(a) x & (x - 1)
(b) -((x | MAX_INT) >> 7)
(c) ((˜x & MIN_INT) == 0
(d) (x << 4) | ((x >> 4) & 0x0F)
(e) (x >> 4) << 4
(f) (x << 3) + (x << 2) + x
(g) (0x80 >> 4) & x
(h) x*(1 | (x >> 7))

Problem 5

Structure layout.

```
struct a {
   float *f;
   char c;
   int x;
   char z[4];
   double d;
   short s;
};

struct b {
   struct a a1;
   int y;
   struct a a2;
};
```

Problem 5

A									
0x0	f	f	f	f	c	#	#	#	
0x8	x	x	x	x	z	z	z	z	
0x10	d	d	d	d	d	d	d	d	
0x18	s	s	#	#					28 bytes
0x20									
B									
0x0	f	f	f	f	f	f	f	f	
0x8	c	#	#	#	x	x	x	x	
0x10	z	z	z	z	#	#	#	#	
0x18	d	d	d	d	d	d	d	d	
0x20	s	s	#	#	#	#	#	#	40 bytes

Problem 5

C

Struct a1: 28 bytes

Int: 4 bytes

Struct a2: 28 bytes

60 bytes

B

Struct a1: 40 bytes

Int: 4 bytes

4 bytes padding (struct a needs 8 byte alignment)

Struct a2: 40 bytes

88 bytes

Agenda

- Problem 1 and 5
- **Processes and Signals**
- Shell lab

Processes

- What is a program?
  - Written according to a specification that tells users what it is supposed to do
  - A bunch of data and instructions stored in an executable binary file
  - Stateless since binary file is static

Processes

- What is a process?
  - A running **instance** of a program in execution
  - One of the most profound ideas in CS

- A fundamental abstraction provided by the OS
  - Single thread of execution (linear control flow) ….
  - … until you create more threads (later in the course)
  - **Stateful:**
    - Full set of **private** address space and registers
    - Other state like open file descriptors and etc.

Processes

- Four basic process control functions
  - fork()
  - exec*() and other variants such as execve()
    - But they all fundamentally do the same thing
  - exit()
  - wait()

Standard on all UNIX-based systems

Don't be confused:
Fork(), **E**xit(), **W**ait() are all wrappers provided by CSAPP

Processes

- fork()
  - Creates or spawns a child process
  - OS creates an exact duplicate of parent's state:
    - Virtual address space (memory), including heap and stack
    - Registers, except for the return value (%eax/%rax)
    - File descriptors **but files are shared**
  - **Result** → Equal but **separate** state
  - Returns 0 for child process but child's PID for parent

Processes

- exec*()
  - Replaces the current process's state and context
  - Provides a way to load and run **another** program
    - Replaces the current running memory image with that of new program
    - Set up stack with arguments and environment variables
    - Start execution at the entry point
  - The newly loaded program's perspective: as if the previous program has not been run before
  - It is actually a family of functions
    - man 3 exec

Processes

- exit()
  - Terminates the current process
  - OS frees resources such as heap memory and open file descriptors and so on…
  - Reduce to a zombie state
    - Must wait to be **reaped** by the parent process (or the **init** process if the parent died)
    - Reaper can inspect the exit status

Processes

- wait()
  - Waits for a child process to change state
  - If a child terminated, the parent “reaps” the child, freeing all resources and getting the exit status
  - Child fully “gone”
  - For details: man 2 wait

Processes (Concurrency)

```
pid_t child_pid = fork();

if (child_pid == 0){
   /* only child comes here */

   printf("Child!\n");

   exit(0);
}
else{
   printf("Parent!\n");
}
```

- What are the possible output (assuming fork succeeds) ?
  - Child!, Parent!
  - Parent!, Child!
- How to get the child to always print first?

Processes (Concurrency)

```
int status;
pid_t child_pid = fork();

if (child_pid == 0){
   /* only child comes here */

   printf("Child!\n");

   exit(0);
}
else{
   waitpid(child_pid, &status, 0);

   printf("Parent!\n");
}
```

- Waits til the child has terminated.
  Parent can inspect exit status of
  child using 'status'
  - WEXITSTATUS(status)

- Output always: Child!, Parent!

Processes (Concurrency)

```
int status;
pid_t child_pid = fork();
char* argv[] = {"ls", "-l", NULL};
char* env[] = {…, NULL};

if (child_pid == 0){
   /* only child comes here */

   execve("/bin/ls", argv, env);

   /* will child reach here? */
}
else{
   waitpid(child_pid, &status, 0);

   … parent continue execution…
}
```

- An example of something useful.
- Why is the first arg "ls"?
- Will child reach here?

Processes

- Four basic States
  - Running
    - Executing instructions on the CPU
    - Number bounded by number of CPU cores
  - Runnable
    - Waiting to be running
  - Blocked
    - Waiting for an event, maybe input from STDIN
    - Not runnable
  - Zombie
    - Terminated, not yet reaped

Signals

- Primitive form of interprocess communication
- Notify a process of an event
- Asynchronous with normal execution
- Come in several types
  - man 7 signal
- Sent in various ways
  - Ctrl+C, Ctrl+Z
  - kill()
  - kill utility

Signals

- Handling signals
  - Ignore
  - Catch and run signal handler
  - Terminate, and optionally dump core
- Blocking signals
  - sigprocmask()
- Waiting for signals
  - sigsuspend()
- Can’t modify behavior of SIGKILL and SIGSTOP
- **Non-queuing**

Signals

- Signal handlers
  - Can be installed to run when a signal is received
  - The form is void handler(int signum){ …. }
  - **Separate** flow of control in the same process
  - Resumes normal flow of control upon returning
  - Can be called **anytime** when the appropriate signal is fired

Signals (Concurrency)

```
….install sigchld handler…

pid_t child_pid = fork();

if (child_pid == 0){
   /* child comes here */

   execve(……);
}
else{

   add_job(child_pid);

}
```

What could happen here?

```
void sigchld_handler(int signum)
{
    int status;

    pid_t child_pid =
      waitpid(-1, &status, WNOHANG);

    if (WIFEXITED(status))
       remove_job(child_pid);
}
```

How to solve this issue?
Block off SIGCHLD signal at the appropriate places. You'd have to think of it yourself.

Agenda

- Problem 1 and 5
- Processes and Signals
- **Shell lab**

Shell lab

- Read the code we've given you
  - There's a lot of stuff you don't need to write yourself
  - It's a good example of the code we expect from you

Shell lab

- Do not use sleep() to avoid race conditions. This is incorrect, we will dock performance points for it.
- Only use sleep() for performance to avoid executing useless instructions (like sitting in a while loop) Your code should still work if sleep calls are removed

Shell lab

- Hazards
  - Race conditions
    - Hard to debug so start early
  - Reaping zombies
    - Race conditions
    - Fiddling with signals
  - Waiting for foreground job
    - One of the only places where sleep is acceptable, though not really needed

Q & A

- Thank you