Frequently asked questions from the previous class survey

- Cores: Since CPU clock speeds have tapered off significantly, do we rely on the kernel to do things or ... [threads/parallel programming]
- What is the executable image?
- Processes: Can a process have multiple parents?
- Do you have to recompile the Kernel? NO!
- PCB: Nothing fancy about this data structure?

Frequently asked questions from the previous class survey

- Memory? [Main memory, RAM, Physical memory, DRAM]
- Is there one giant Stack and Heap for ALL processes? NO!
- How is the stack and heap connected to main memory?
- FSMs: How can a process go from Waiting for I/O to Ready without using the CPU?
- Lots of Memory Management Questions
  - Access limits, Paging, etc

Topics covered in this lecture

- Operations on processes
  - Creation
  - Termination
- Process groups
- Buffer Overflows
  - One of the greatest security violations of all time

Process creation in UNIX

- Process created using \texttt{fork()}
  - \texttt{fork()} copies parent's memory image
  - Includes copy of parent's address space
- Parent and child continue execution \textbf{at instruction after} \texttt{fork()}
  - Child: Return code for \texttt{fork()} is 0
  - Parent: Return code for \texttt{fork()} is the non-ZERO process-ID of new child
fork() results in the creation of 2 distinct processes

Parent

- Child will execute from here
- id = 0 here
- Parent PID
- id = fork()

Child

- id = 0 here
- Child PID
- id = fork()

Results in

id = xyz here
id = 0 here

Child will execute from here

Simple example:

```c
#include <stdio.h>
#include <unistd.h>

int main(void) {
    int x;
    x = 0;
    fork();
    x = 1;
    ...
}
```

Both parent and child execute this after returning from fork()

Another example

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("Hello World\n");
    fork();
    printf("Hello World\n");
}
```

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("Hello World\n");
    if (fork() == 0) {
        printf("Hello World\n");
    }

    printf("Hello World\n");
    if (fork() == 0) {
        printf("Hello World\n");
    }
    ...
}
```

What happens when fork() fails?

- No child is created
- fork() returns -1 and sets errno
- errno is a global variable in errno.h

If a system is short on resources OR if limit on number of processes breached

- fork() sets errno to EAGAIN
- Some typical numbers for Solaris
  - maxusers: 2 less than number of MB of physical memory up to 1024
    - Set up to 2048 manually in /etc/system file
  - mx_nprocs: Default: 16 x maxusers + 10
    - min = 138, max = 30,000

Take different paths depending on what happens with fork()
Creating a chain of processes

```c
for (int i=1; i < 4; i++) {
    if (childid = fork()) {
        break;
    }
}
```

For each iteration:
- Parent has non-ZERO childid
- So it breaks out
- Child process
- Parent in NEXT iteration

Value of `i` when process leaves loop

Creating a process fan

```c
for (int i=1; i < 4; i++) {
    if ((childid = fork()) <= 0) {
        break;
    }
}
```

Newly created process breaks out
Original process continues

Value of `i` when process leaves loop

Creation of a process tree

```c
int i=0;
for (i=1; i < 4; i++) {
    if ((childid = fork()) == -1) {
        break;
    }
}
```

Original process has a 0 label
- Value of `i` when created
- Lower case letters: Process created with same `i`
- Both parent and child go on to create processes in the next iteration

Replacing a process's memory space with a new program

- Use `exec()` after the `fork()` in one of the two processes
- `exec()` does the following:
  1. Destroys memory image of program containing the call
  2. Replaces the invoking process's memory space with a new program
  3. Allows processes to go their separate ways

Replacing a process's memory space with a new program

- **TRADITION:**
  - Child executes new program
  - Parent executes original code

Launching programs using the shell is a two-step process

- **Example:** user types `sort` on the shell
  1. Shell `forks` off a child process
  2. Child executes `sort`
But why is this the case?

- Allows the child to manipulate its file descriptors
  - After the fork()
  - But before the exec()
- Accomplish redirection of standard input, standard output, and standard error

A parent can move itself from off the ready queue and await child’s termination

- Done using the wait() system call.
- When child process completes, parent process resumes

wait/waitpid allows caller to suspend execution till a child’s status is available

- Process status availability
  - Most commonly after termination
  - Also available if process is stopped
- waitpid(pid, *stat_loc, options)
  - pid == -1 : any child
  - pid > 0 : specific child
  - pid == 0 : any child in the same process group
  - pid < -1 : any child in process group abs(pid)

Process creation in Windows

- CreateProcess handles
  1. Process creation
  2. Loading in a new program
- Parent and child’s address spaces are different from the start

CreateProcess takes up to 10 parameters

- Program to be executed
- Command line parameters that feed program
- Security attributes
- Bits that control whether files are inherited
- Priority information
- Window to be created?

Process Management on Windows

- Win 32 has about 100 other functions
  - Managing & Synchronizing processes
Process Groups

August 30, 2018

CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University

Process group is a collection of processes
Each process has a process group ID
Process group leader?
- Process with pid == pgid
kill treats negative pid as pgid
- Sends signal to all constituent processes

Process Group IDs:
When a child is created with fork()

1. Inherit parent’s process group ID
2. Parent can change group ID of child by using setpgid
3. Child can give itself new process group ID
   - Set process group ID = its process ID

Windows has no concept of a process hierarchy

- The only hint of a hierarchy?
  - When a process is created, parent is given a special token (called handle)
    - Use this to control the child
  - However, parent is free to pass this token to some other process
    - Invalidates hierarchy

Process Terminations

August 30, 2018

CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University
Process terminations

- Normal exit (voluntary)
  - E.g., successful compilation of a program
- Error exit (voluntary)
  - E.g., trying to compile a file that does not exist

Process terminations: This can be either normal or abnormal

- OS deallocates the process resources
  - Cancel pending timers and signals
  - Release virtual memory resources and locks
  - Close any open files
  - Updates statistics
    - Process status and resource usage
  - Notifies parent in response to a wait()

On termination a UNIX process DOES NOT fully release resources until a parent waits for it

- When the parent is not waiting when the child terminates?
  - The process becomes a zombie
- Zombie is an inactive process
  - Still has an entry in the process table

Zombies and termination

- When a process terminates, its orphaned children and zombies are adopted
  - This special system process is init
- Some more about init
  1. Has a pid of 1
  2. Periodically waits for children
  3. Eventually orphaned zombies are removed

Normal termination of processes

- Return from main
- Implicit return from main
  - Function falls off the end
  - Call to exit, _Exit or _exit

CS370: Operating Systems [Fall 2018]
Dept. Of Computer Science, Colorado State University
Abnormal termination

- Call abort
- Process signal that causes termination
  - Generated by an external event: keyboard Ctrl-C
  - Internal errors: Accessing illegal memory location
- Consequences
  - Core dump
  - User-installed exit handler not called

Protection and Security

- Control access to system resources
  - Improve reliability
- Defend against use (misuse) by unauthorized or incompetent users
- Examples
  - Ensure process executes within its own space
  - Force processes to relinquish control of CPU
  - Device-control registers accessible only to the OS
  - E.g. Why the Security of USB Is Fundamentally Broken
    https://www.wired.com/2014/07/usb-security/

Buffer overflows:

- When? Program copies data into a variable for which it has not allocated enough space

```c
char buf[80];
printf("Enter your first name:");
scanf("%s", buf);
```

If user enters string > 79 bytes?

- The string AND string terminator do not fit.

Automatic variables (local variables)

- Allocated/deallocated automatically when program flow enters or leaves the variable's scope
- Allocated on the program stack
- Stack grows from high-memory to low-memory

Buffer Overflows:
Fixing the example problem

```c
char buf[80];
printf("Enter your first name:");
scanf("79%s", buf);
```

Program now reads at most 79 characters into buf
A process in memory

- **Stack**
  - Function parameters, return addresses, and local variables
- **Heap**
  - Memory allocated dynamically during runtime
- **Data**
  - (Global variables)
- **Text**
  - (Program code)

A rough anatomy of the program stack

- **Base**
  - (Return address)
- **Top**
  - (Local variables)

A function that checks password: Susceptible to buffer overflow

```c
int checkpass(void) {
    int x;
    char a[9];
    x = 0;
    printf("Enter a short word: ");
    scanf("%s", a);
    if (strcmp(a, "mypass") == 0) {
        x = 1;
        return x;
    }
}
```

Stack layout for our unsafe function

- **Base**
  - Return address
- **Top**
  - A long password may overwrite this too

Problems with buffer overflow

- Function will try to return to an address space **outside** the program
- Segmentation fault or core dump
- Programs may lose unsaved data
- In the OS, such a function can cause the OS to crash!

One of the greatest security violations of all time: November 2, 1988

- Exploited 2 bugs in Berkeley UNIX
- Worm: Self replication program
- Bought down most of the Sun and VAX systems on the internet within a few hours
Worm had two programs

1. Bootstrap (99 lines of C, l1.c)
2. Worm proper

Both these programs compiled and executed on the system under attack

Synopsis of the worm’s modus operandi

1. Spread the bootstrap to machines
2. Once the bootstrap runs:
   - Connects back to its origins
   - Download worm proper
   - Execute worm
3. Worm then attempts to spread bootstrap

Infecting new machines: Method 1 & 2

Violate trust

- Method 1: Run the remote shell rsh
  - Machines used to trust each other, and would willingly run it
  - Use this to upload the worm
- Method 2: sendmail

Method 3: Buffer overflow in the finger daemon (finger name@site)

- finger daemon runs all the time on sites, and responds to queries
- The worm called finger with a handcrafted 536-byte string as a parameter.
  - Overflowed daemon’s buffer & overwrote its stack
  - Daemon did not return to main(), but to a procedure in the 536-bit string on stack
  - Next try to get a shell by executing /bin/sh

Far too many worms can grind things to a halt

- Break user passwords
- Check for copies of worm on machine
  - Exit if there is a copy 6 out of 7 times
    - This is in place to cope with a situation where sys admin starts fake worm to fool the real one
  - Use of 1 in 7 caused for too worms
    - Machines ground to a halt

Consequences

- $10K fine, 3 years probation and 400 hours community service
- Legal costs $150,000
The contents of the slide-set are based on the following references: