System Calls

Exec

Daemons

System Calls

- UNIX system provides several system calls
    - To create and end programs
    - To send and receive software interrupts
    - To allocate memory
- Process system calls
    - fork(), the "exec" family, wait(), exit()

Running UNIX Commands from C

- Use the function `int system(char *string)`
    - A unix utility
    - An executable shell script
    - User program
- System returns the exit status of the shell
- System is prototyped in <stdlib.h>
- Run commands in a C program as if from command line
- Save time and hassle using proven programs, scripts etc.
- Gather scripts created for system management together

Process Management

- Command `ps` is used to view running processes
  - `ps -aef | grep netid` searches for the netID
- Command `kill` used to end process with a process ID
- Header files `<stdlib.h>` and `<unistd.h>`
- `int getpid()`
  - Returns the process ID of the calling process
  - Used to generate unique temp filenames by routines
- `int getppid()`
  - Returns the parent process ID of child by `fork()`

Process Management

- `int wait(status)`
    - Status a pointer to integer where system stores value
    - Returns the process ID of the process that ended
    - Control the execution of child processes
    - Forces parent suspend execution until child finished
    - Returns process ID of a child process that finished
    - Child finishes before, immediately return child's pID
    - Fails:
        - Process has no children
        - Process ended by calling the exit() system call

Process Management

- `waitpid(pid,status,options)`
  - Calling process pause until specified process finishes
  - Used for inter-process synchronization
- `void exit(status)`
  - Status is an integer between 0 and 255
  - Ends a process and returns a value to it parent
  - Status 0 means it did not encounter any problems
  - Status non-zero status means it did have problems
  - Library routine that calls system exit()
  - Buffered output not written out is flushed

Fork()

- To create a new process `int fork()`
- New process called the "child process"
- Child process is a copy of the parent process
- Child process has a new process ID
- Child process contents identical to the parent process
- Parent and child process exist on the system
- Parent and child have different address space

Fork Create 2 Child Processes

```c
int main (void) {
   pid_t cp1, cp2;
   cp1 = fork();
   cp2 = fork();
   if (cp1 == 0) {
      printf("I am the first child.\n");
   }
   else if (cp2 == 0 ) {
      printf("I am the second child.\n");
   }
   else {
      printf("I am the parent.\n");
   }
   return 0;
}
```

Problems Encountered

- Memory Issues
  - Modifying shared resources lead to inconsistencies
  - Synchronization mechanisms need to be implemented
- Resource Leaks
  - Child process does not close unnecessary resources
- Process Zombies
  - Completed process still has an entry in process table
  - Parent did not get their exit status
  - Can fill up process table, leading to system instability

Problems Encountered

- Error Handling
  - fork() function returns twice, can lead to confusion
  - fork() fails due to lack of system resources
- Thread Safety
  - fork() can lead to issues because it is not thread-safe
  - Child inherits the thread control blocks leading to problems

Fork Bomb Notes and Avoidance Techniques

- Yes, they're hilarious until ....
- Under no circumstances should you attempt this
- Warning signs your program will consume all system resources available and lock you and everyone else out:
    - You've written a loop that calls fork()
    - Code where child process creates a child process
    - Code in which child process is starting up a loop
- Set a flag in loop and have a separate function call fork()
- Add an extra condition to a loop to abort if 50 forks

Exec() Family

- Calling one of the exec() family
    - Terminate the currently running program / process
    - Execute new code specified in parameters of exec
    - The process id is not changed
- Provides some flexibility in how arguments are passed in and how the program is looked up.
- `execl()`, `execv()`, `execle()`
- `execlp()`, `execvp()`

Execl()

- `int execl(char *path,char *arg0,...,char *argn,0)`
    - Path a pointer the name of a file holding a command
    - *arg0* a pointer to a string same as path
    - *arg1 ... argn* are pointers to arguments for command
    - NULL or 0 simply marks end of the list of arguments
- Creation of a child replace the parent process
- Child replaces the address space of the parent process

execl() Program

```c
#include <unistd.h>

int main() {

  // Replace the current process with myprog
  execl("/path/to/myprog", "myprog",
   "ARG1", "ARG2", NULL);

  // If execl returns, there was an error
  perror("execl");

  return 1;
}
```

Daemons

- A process that runs in the background
- Any output is logged in a file
- Not associated with any terminal
    - Output doesn't end up in another session
    - Terminal generated signals (^C) not received
- Unix systems typically have many daemon processes
- Most servers run as a daemon process
- Almost no user interaction is required
- Used for long running services begin at start up

Daemons

- Equivalent to a Microsoft Windows Services
- Manipulation at startup or in a config files
- init daemon is the "parent" of all processes
- Observes network activity
- Logs any suspicious communication

Finding Daemons

- `ps -aef` lists the current processes
- `kill -9 ##` stops the process where a pID is provided

Common Daemons

- Super-Daemon (inetd)
- Scheduler (crond)
- Dynamic Host (dhcpd)
- File Transfer Protocol (ftpd)
- Web Server (httpd)
- Line Printer (lpd)
- Domain Name (named)
- Secure Shell (sshd)
- Logging Requests (syslogd)
- Replaces init (systemd)

Daemon Process Design

- Create a normal process (Parent process)
- Create a child process from within the above parent process
- The process hierarchy at this stage looks like:
    - Terminal
        - Parent Process
            - Child Process
                - Child Process

Daemon Process Design

- Terminate the the first child process
- Second Child now becomes orphan
- Control is taken over by the init process
- Becomes a daemon process without a controlling terminal
- Let the logic of daemon process run
- Parent (init) will be informed at completion
- Resources and table entry will be released

Zombie Process

- A process completed execution not reaped by parent
- Process entry held in the process table and PID
- Kill command has not effect on a zombie process
- Memory and resources associated are not deallocated
- Similar to memory leak
- Usually a sign of invalid software behavior

Prevent Zombie Process

- init works to look for processes left behind
- Process table is finite
- System will come to a standstill
- System signal show parent processes ignorance
    - SIG_IGN - signal to ignore child
    - SA_NOCLDWAIT - flag set to ignore child

Questions?