Files and Processes (review)

Learning Objectives

- Review of files in standard C versus using system call interface for files
- Review of buffering concepts
- Review of process memory model
- Review of bootup sequence in Linux and Microsoft Windows
- Review of basic system calls under Linux: `fork`, `exec`, `wait`, `exit`, `sleep`, `alarm`, `kill`, `signal`
- Review of similar basic system calls under MS Windows

Files

- Recall how we write a file copy program in standard C.

```
#include <stdio.h>
FILE *fopen(const char *path, const char *mode);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
int fclose(FILE *fp);
```

- We can also use character-based functions such as:

```
#include <stdio.h>
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
```

- With either approach, we can write a C program that will work on any operating system as it is in standard C.

Standard C File Copy

- Uses fread and fwrite.
- files-processes/stdc-mycp.c

POSIX/Unix Files

- "On a UNIX system, everything is a file; if something is not a file, it is a process."
- A directory is just a file containing names of other files.
- Programs, services, texts, images, and so forth, are all files.
- Input and output devices, and generally all devices, are considered to be files.

File Types

- Regular files: text files, executables, input for or output from a program, etc.
- Directories: files that are lists of other files.
- Special files: the mechanism used for input and output. Most special files are in /dev.
  - Character device: allow users and programs to communicate with hardware peripheral devices.
  - Block device: similar to character devices. Mostly govern hardware such as hard drives, memory, etc.
- Links: a system to make a file or directory visible in multiple parts of the system's file tree.
- (Domain) sockets: a special file type, similar to TCP/IP sockets, provides inter-process networking protected by the file system's access control.
- Named pipes: like sockets, provide a way for local processes to communicate with each other.

File Types (2)

- Can use `ls -l` to determine the file type. The first character displays the type.

Symbol	Meaning
-	Regular file
d	Directory
l	Link
c	Character device
s	Socket
p	Named pipe
b	Block device

POSIX/Unix File Interface

- The system call interface for files in POSIX systems like Linux and MacOSX.
- A *file* is a named, ordered stream of bytes.
  - `open(..)` Open a file for reading or writing. Also allows a file to be locked providing exclusive access.
  - `close(..)`
  - `read(..)` The read operation is normally *blocking*.
  - `write(..)`
  - `lseek(..)` Seek to an arbitrary location in a file.
  - `ioctl(..)` Send an arbitrary control request (specific to a device). e.g. rewinding a tape drive, resizing a window etc.
  - See **`man 2 <function-name>`** for documentation.
- Let's rewrite the file copy program using the POSIX system call interface.

POSIX System File Copy

- Uses read and write.
- files-processes/mycp.c

Buffering


NETFLIX
36%
Buffering


Everyday
I'm buffering
YouTube
smosh.com
0:26 / 3:45

Effect of buffer size on I/O speed

- Buffering helps adjust the data rate between two entities to avoid overflow.
- Buffering can also improve performance of systems by allowing I/O to happen ahead of time or to have I/O happen in parallel with computing.
- *Buffering is a widely used concept in Computer Science.*
- Observe the effect of buffer size on the speed of the copying. Experiment using the file copy program with different buffer sizes on a large file and time the copy.
- Script for testing effects of buffering: files-processes/test-mycp.sh

Effect of Buffering on File I/O (System Calls)

The following times are for the file copy program with varying buffer sizes. All times are in seconds. Total speedup due to buffering is 1455!

buffer size	elapsed	user	system
1	36.387	1.565	34.398
2	17.783	0.757	16.974
4	9.817	0.400	9.393
8	4.603	0.180	4.375
16	2.289	0.093	2.190
32	1.142	0.047	1.091
64	0.581	0.017	0.562
128	0.299	0.012	0.286
256	0.158	0.007	0.150
512	0.090	0.003	0.084
1024	0.054	0.001	0.051
2048	0.035	0.001	0.032
4096	0.025	0.000	0.024

Do we get a similar improvement with the standard C program? Why or why not?

Effect of Buffering on File I/O (Standard C)

The following times are for the file copy program with varying buffer sizes. All times are in seconds. Total speedup is around 40.

buffer size	elapsed	user	system
1	0.965	0.945	0.018
2	0.502	0.475	0.026
4	0.267	0.244	0.021
8	0.156	0.136	0.019
16	0.082	0.053	0.028
32	0.056	0.032	0.023
64	0.042	0.014	0.027
128	0.034	0.012	0.021
256	0.033	0.010	0.022
512	0.029	0.008	0.021
1024	0.029	0.006	0.022
2048	0.028	0.004	0.023
4096	0.024	0.001	0.022

Why does the standard C program behave differently?

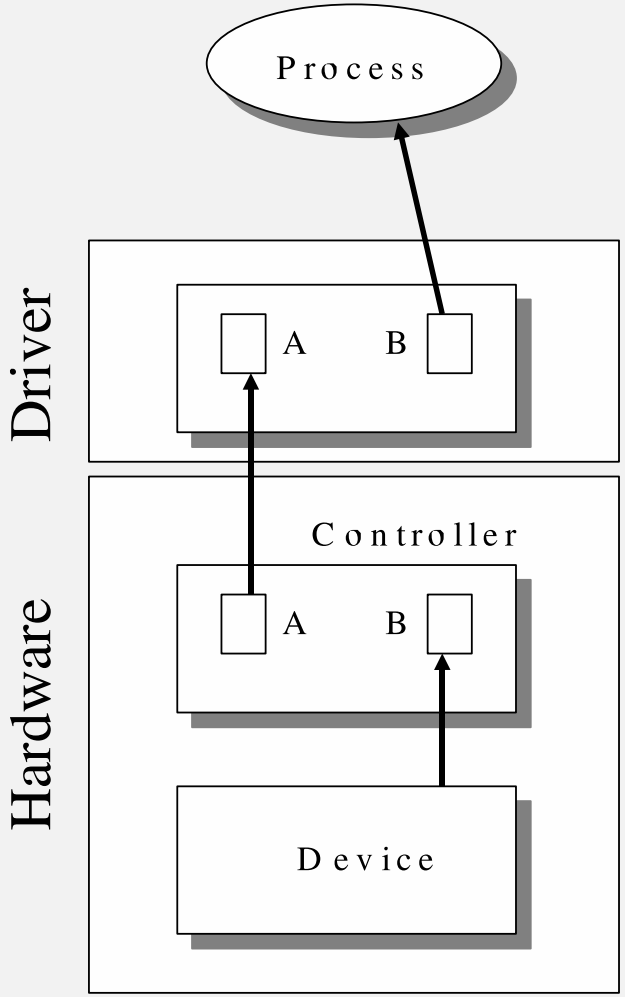
Examples of Buffering from “Real-Life"

- Ice cube trays. You have one tray that you get ice cubes from and another full tray that is not used. When the first tray is empty, you refill that tray and let it freeze while you get ice cubes from the other tray.
- Shock absorbers in car, truck or mountain bike.
- Ski lift is a circular buffer
- Two parents buffer a child’s demand for attention.
- Multiple elevators in a hotel lobby. An escalator might be considered a circular buffer.
- Traffic lights at an intersection buffer the flow of traffic through the limited resource that is an intersection. A round-about is a circular-buffer solution to the same problem.

Buffering

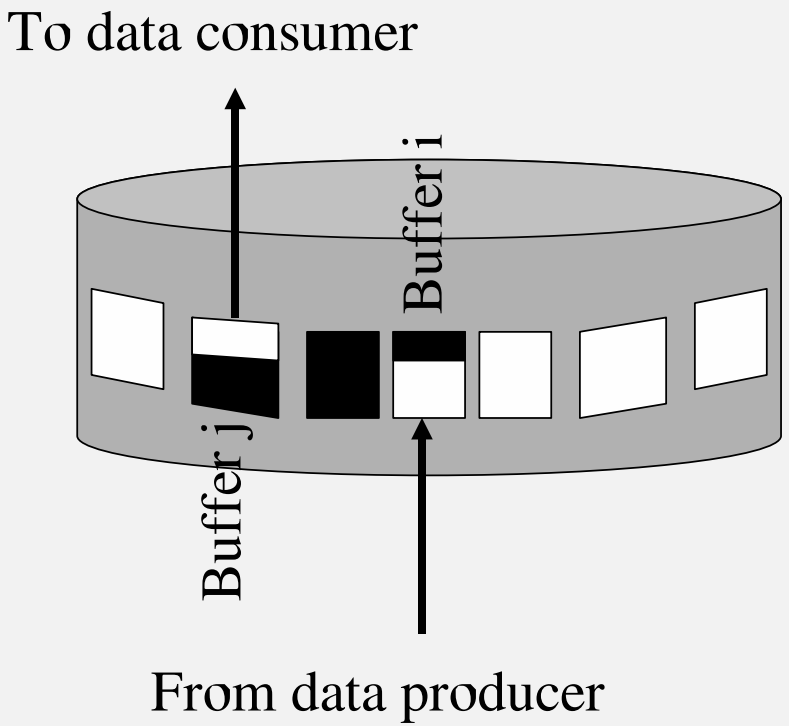
- Buffering improves I/O performance by allowing device managers to keep slower I/O devices busy when process do not need I/O.
- Single buffering.
- Double buffering.
- Circular buffering.

Double Buffering



- Double buffering is used in software and hardware.

Circular Buffering



- Similar to maintaining a variable-sized queue in a fixed size circular array.
- See example of queue implemented as a circular array in device-management/ArrayQueue.c

Presenting Experimental Results

- Always give details of the CPU on the machine along with the clock speed, cache size, amount of main memory and swap size and disk speed (seek time, latency, buffer size) (if swapping is an issue).
- Remember that you will get a better time by running the application several times because of caching.
- Always mention what compiler was used (including the version), what compiler flags were set (whether you used the optimization flag or not), and under what operating system was the experiment carried out.
- Remember that elapsed time depends on how many processes are using the CPU at the time the timing was done. The *user time* (a.k.a. CPU time) and the *system time* are pretty much independent of the number of processes active on the system.

MS Windows File Interface

A *file* is a named, ordered stream of bytes.

- `OpenFile() or CreateFile(..)` Open or create a file for reading or writing. Returns a HANDLE (reference) to a file structure used to identify the file for other system calls.
- `CloseHandle(..)`
- `ReadFile(..)` The read operation is normally *blocking*.
- `WriteFile(..)`
- `SetFilePointer(..)` Seek to an arbitrary location in a file.

Processes

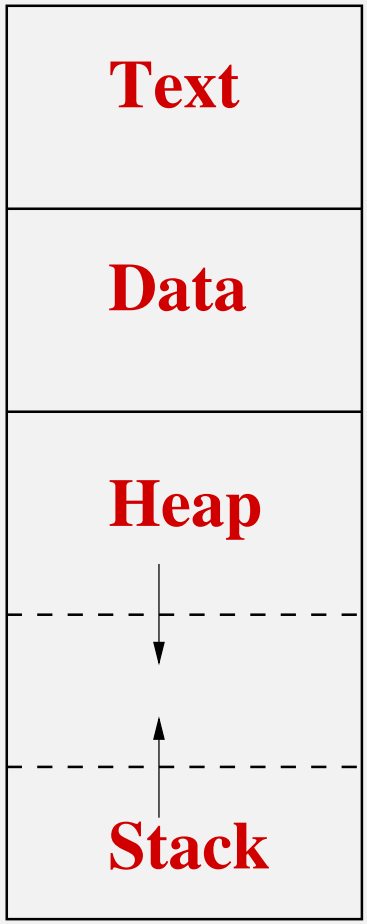
Components of a process:

- the executable code (also known as program text or text segment)
- the data on which the program will execute
- status of the process
- resources required by the process: e.g. files, shared libraries etc.

The data associated with a process is divided into several segments:

- Global and static variables: data segment
- Dynamically allocated variables: heap segment
- Local variables, function/method arguments and return values: stack segment

The Linux/UNIX Process model



0x00000000

Program Binary

Global/Static variables

Dynamically allocated variables

Local variables, function/method arguments

0xFFFFFFFF

Memory Quiz (part 1)

Where is Neo? Where is Morpheus? Somewhere in the Matrix. Based on the following code, determine in which segment are the specified variables (on the next slide) allocated.

```
#define BODY_BIT_SIZE 1000000
int A[BODY_BIT_SIZE];
extern void transfer();

void booth(char *xyz)
{
    int i;
    static int neo[BODY_BIT_SIZE];
    int *morpheus = (int *) malloc(sizeof(int) * BODY_BIT_SIZE);
    for (i = 0; i < BODY_BIT_SIZE; i++)
        morpheus[i] = neo[i];
    morpheus[0] = xyz;
    transfer();
}
int main(int argc, char *argv[])
{
    char *xyz = (char *) malloc(sizeof(char) * BODY_BIT_SIZE);
    printf("Hello?\n"); scanf("%s", xyz)
    booth(xyz);
}
```

Memory Quiz (part 2)

1.	`A[100]`	Data	Heap	Stack
2.	`i`	Data	Heap	Stack
3.	`morpheus`	Data	Heap	Stack
4.	`morpheus[0]`	Data	Heap	Stack
5.	`neo[10]`	Data	Heap	Stack
6.	argc	Data	Heap	Stack
7.	`xyz` (in booth(...))	Data	Heap	Stack

Creation of Processes

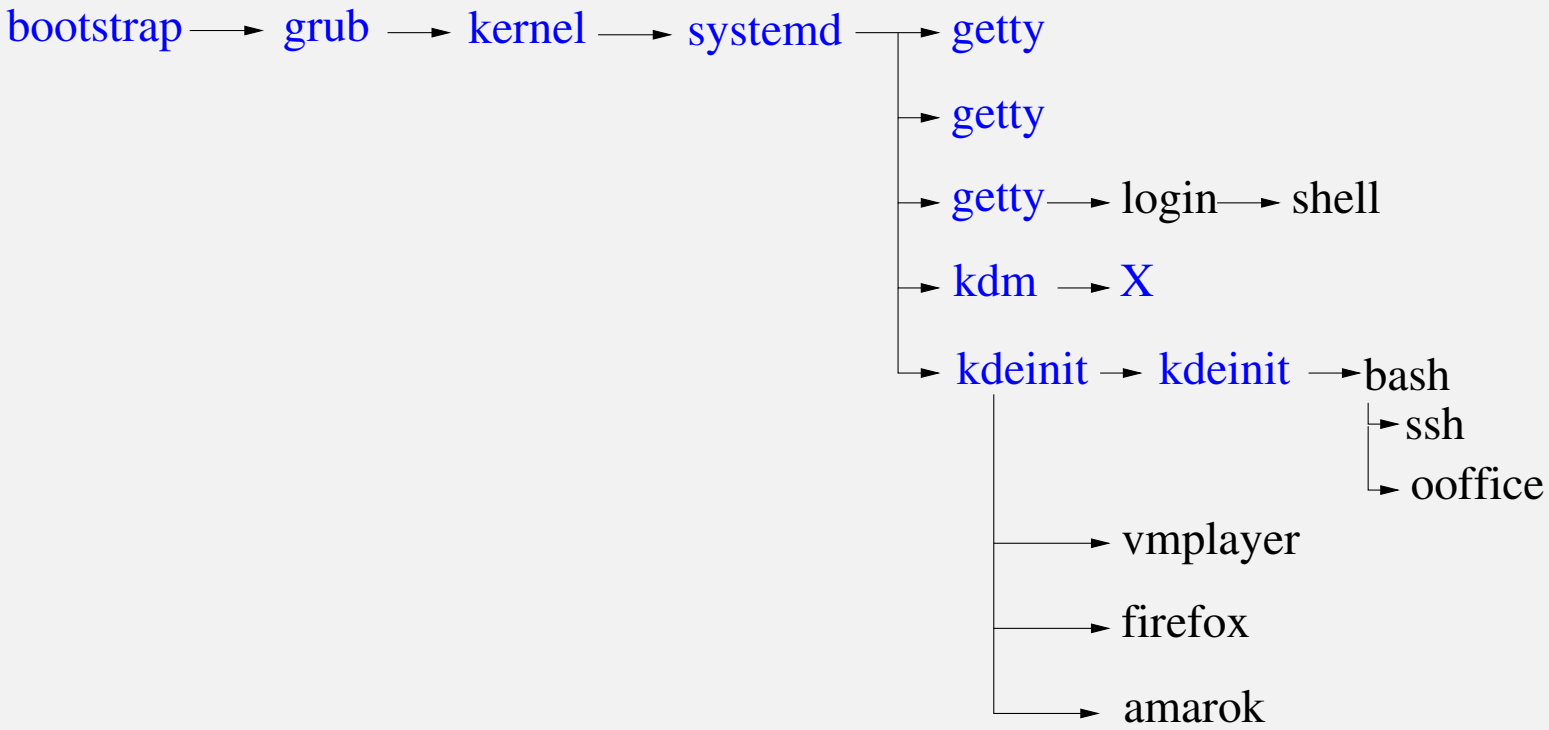
- To the user, the system is a collection of processes. Some of them are part of the operating system, some perform other supporting services and some are application processes.
- Why not just have a single program that does everything?
- How is a process created?
- How is the operating system created at bootup? Let's examine the bootup of Linux and Microsoft Windows.

Initializing the Operating System

"Booting" the computer.

```
Main Entry: bootstrap
Function: noun
Date: 1913
1 plural: unaided efforts -- often used in the phrase
          by one's own bootstraps
2 : a looped strap sewed at the side or the rear top
          of a boot to help in pulling it on.
```

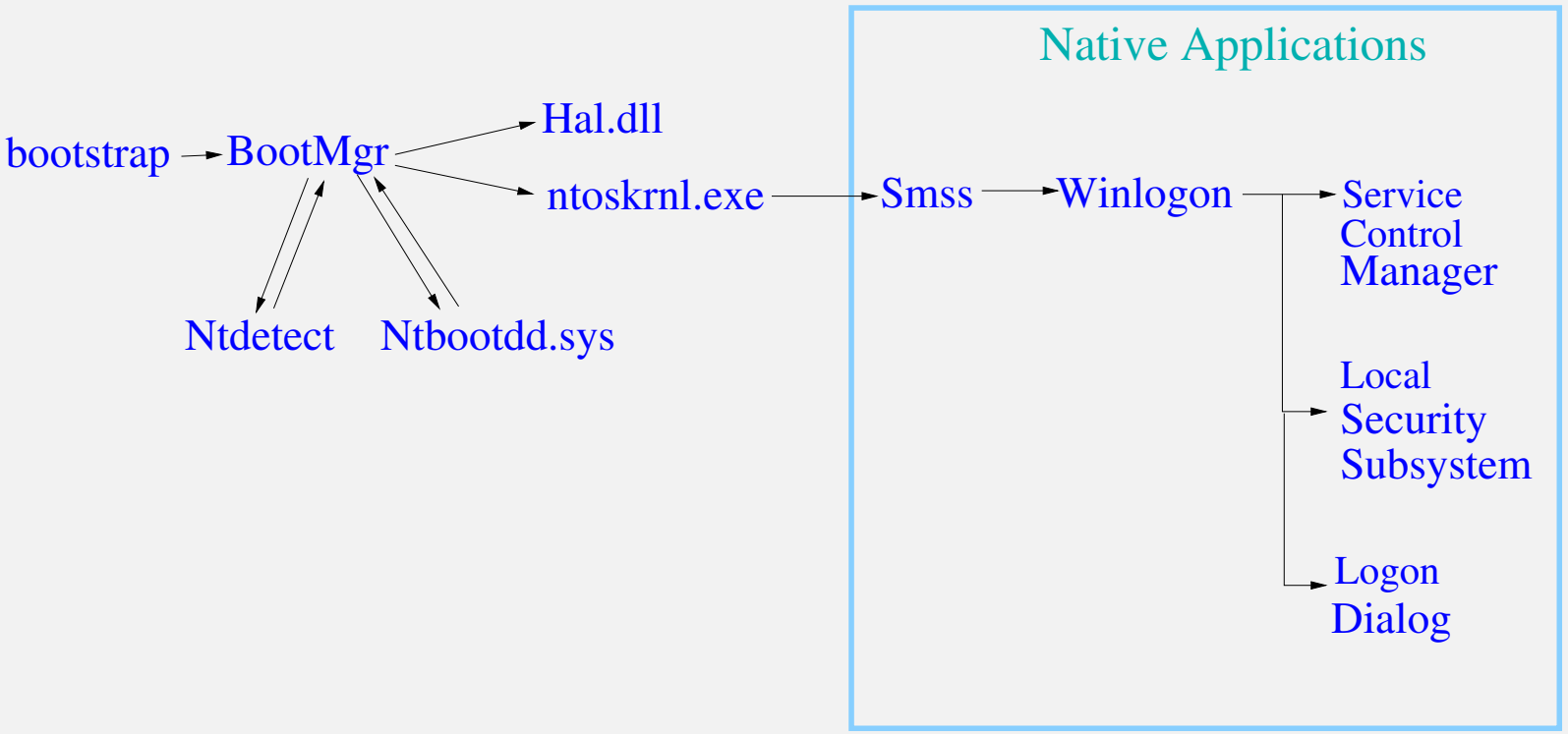
The Linux Bootup Process



systemd has replaced init process that was used traditionally in Linux/Unix systems

Try the pstree command!

Microsoft Windows Bootup Process



Smss: Session Manager SubSystem

Executing Computations

- The Unix model (followed by most operating systems) is to create a new process every time to execute a new computation. The system at any time looks like a tree of processes, with one process being the ancestor of all other processes.
- *What's the advantage of creating a process each time we start a new computation?*

The fork() system call

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

pid_t fork(void);
```

- The fork() system call creates a child process that is a clone of the parent. The stack, data and heap segments are the same at the moment of creation. The program text is also logically copied (but may be physically shared).
- The child process differs from the parent process only in its process id and its parent process id and in the fact that resource utilization is set to zero.
- One easy way to communicate between a parent and a child is for the parent to initialize variables and data structures before calling `fork()` and the child process will inherit the values.
- *The `fork()` is called once but it returns twice!* (one in the parent process and once in the child process)

Poetry and Aphorisms

Two roads diverged in a wood, and I–
I took the one less traveled by,
and I got lost!

When you come to a fork in the road, take it!
–Yogi Berra

Forked Processes?

Forked Processes?

Forked Processes?

The `fork` system call

- Example: files-processes/fork-hello-world.c

wait and waitpid system calls

- Used to wait for a state change in a child process.

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

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

- The `waitpid()` system call suspends the calling process until one of its child terminates or changes states. Using -1 for `pid` in `waitpid()` means to wait for any of the child processes. Otherwise, `pid > 0` provides the specific process id to wait for.
- It is possible to do a non-blocking wait using the `WNOHANG` option, as shown below where the call will return immediately if no child is done or changed state:

```
waitpid(-1, &status, WNOHANG);
```

- The `wait()` system call suspends execution of the calling process until one of its children terminates. The call `wait(&status)` is equivalent to:

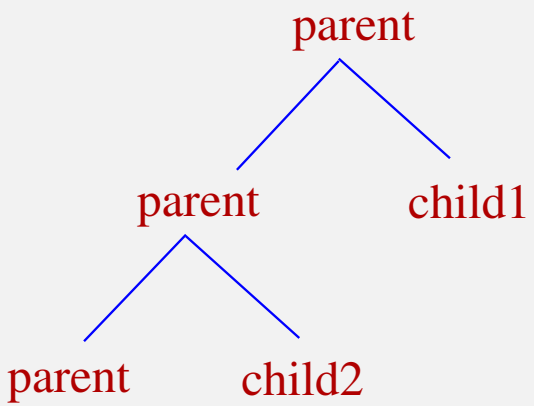
```
waitpid(-1, &status, 0);
```

- Check the man page on how to check the `status` variable to get more information about the child process whose pid was returned by `waitpid()` or `wait()` call.

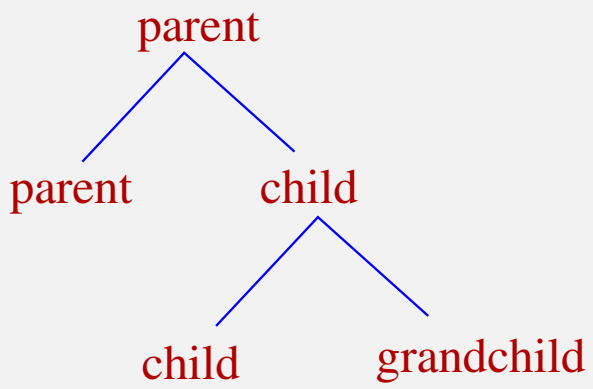
- Example: files-processes/fork-and-wait.c
- Example: files-processes/fork-child-grandchild.c

Process Hierarchy

- We can draw a process family tree to make it easier to understand the flow of control.
- For example, see below for the process tree for the last two examples:



fork–hello–world



fork–child–grandchild

In-Class Exercises (1)

- **Fork Exercise 1**. How many **new** processes are created by the following code?

```
fork();
fork();
```

- **Fork Exercise 2**. How many **new** processes are created by the following code?

```
fork();
fork();
fork();
```

- **Fork Exercise 3**. How many **new** processes are created by the following code?

```
int i;
for (i = 0; i < n; i++)
    fork();
```

In-Class Exercises (2)

- **The Forked Ancestry**. Examine the following C program fragment. How many **new** processes are created by the following code?

```
/* process A */
/* ... */
 if (fork() == 0) {
    if (fork() == 0) {
        if (fork() == 0) {
            if (fork() == 0) {
             /* do something */
            }
        }
    }
 }
/* ... */
```

1. 5 new processes
2. 4 new processes
3. 16 new processes
4. 15 new processes
5. 8 new processes

Pedal to the metal: fork test

- Try to generate a lot of processes
- Example: files-processes/fork-test.c

The exec() system call


```
#include <unistd.h>
int execve(const char *filename, char *const argv [],
           char *const envp[]);
```

- The `execve()` executes the program pointed to by the `filename` parameter. It does not return on success, and the text, data and stack segments of the calling process are overwritten by that of the program loaded. The program invoked inherits the calling process's process id. See the man page for more details.
- The `execve()` function is *called once but it never returns on success!* The only reason to return is that it failed to execute the new program.
- The following variations are front-ends in the C library. In the first two variations, we only have to specify the name of the executable (without any '/') and the function searches for its location in the same way as the shell using the `PATH` environment variable. The last three variations specify the full path to the executable.

```
int execlp(const char *file, const char *arg, ...);
int execvp(const char *file, char *const argv[]);
int execl(const char *path, const char *arg, ...);
int execle(const char *path, const char  *arg, ...,
           char *const envp[]);
int execv(const char *path, char *const argv[]);
```

How does the shell find an executable?

- When we type the name of a program and hit enter in the shell, it searches for that executable in a list of directories specified usually by the `PATH` environment variable
- We can check the value of the PATH variable with the `echo` command:

```
[user@onyx ~]$ echo $PATH
/bin:/usr/lib64/ccache:/usr/local/bin:/usr/bin:
/home/students/user/bin:.:
```

  We get a colon separated list of directories. The search is done in order from the first to the last directory in the list and chooses the first instance of the executable it finds
- We can ask the shell which executable it will use with the `which` command. For example:

```
[user@onyx ~]$ which gcc
/bin/gcc
```

- A program can find the value of the `PATH` variable with the system call `getenv("PATH")`

Example of exec'ing a process

- Example: files-processes/fork-and-exec.c
- The exce'ed program: files-processes/print-pid.c

A simple shell

- Example: files-processes/simple-shell.c
- Need error.c and ourhdr.h to compile.

In-Class Exercises (3)

- **Drones and Drones**. Read the code below and choose the statement below that correctly describes what the code is doing.

```
int main(int argc, char **argv)
{
    pid_t pids[4], pid;
    int i, status;
    for (i=0; i<4; i++) {
        if ((pid = fork()) == 0) {
            execlp("xlogo", "xlogo", (char *) 0);
        } else if (pid > 0) {
            pids[i] = pid;
        } else {
            /* print appropriate error message */
        }
    }
    waitpid(-1, &status, 0);
    exit(0);
}
```

1. Starts four copies of the program xlogo and waits for all them to finish
2. Starts four copies of the program xlogo and waits for any one of them to finish
3. Starts four copies of the program xlogo and waits for three of them to finish
4. Starts four copies of the program xlogo and waits forever
5. Gangnam-style xlogo dance!

Signals: asynchronous events

Linux/Unix signals are a type of event. Signals are asynchronous in nature and are used to inform processes of certain events happening.
Examples:

- User pressing the interrupt key (usually `Ctl-c` or `Delete` key). Generates the `SIGINT` signal.
- User pressing the stop key (usually `Ctl-z`). Generates the `SIGTSTP` signal, which stops (suspends) the process.
- The signal `SIGCONT` can restart a process if it is stopped.
- Signals are available for alarm (`SIGALRM`), for hardware exceptions, for when child processes terminate or stop and many other events.
- Special signals for killing (`SIGKILL`) or stopping (`SIGSTOP`) a process. These cannot be ignored by a process.

POSIX signals list

Read `man signal` and `man 7 signal` for more information.

```
SIGHUP   Hangup detected on controlling terminal
         or death of controlling process
SIGINT   Interrupt from keyboard
SIGQUIT  Quit from keyboard
SIGILL   Illegal Instruction
SIGABRT  Abort signal from abort
SIGFPE   Floating point exception
SIGKILL  Kill signal (cannot be ignored)
SIGSEGV  Invalid memory reference
SIGPIPE  Broken pipe: write to pipe with no readers
SIGALRM  Timer signal from alarm
SIGTERM  Termination signal
SIGUSR1  User-defined signal 1
SIGUSR2  User-defined signal 2
SIGCHLD  Child stopped or terminated
SIGCONT  Continue if stopped
SIGSTOP  Stop process (cannot be ignored)
SIGTSTP  Stop signal from keyboard
SIGTTIN  tty input for background process
SIGTTOU  tty output for background process
```

Signals (contd.)

- For each signal there are three possible actions: default, ignore, or catch. The system call `signal()` attempts to set what happens when a signal is received. The prototype for the system call is:

```
void (*signal(int signum, void (*handler)(int)))(int);
```

- The above prototype can be made easier to read with a typedef as shown below.

```
typedef void sighandler_t(int);
sighandler_t *signal(int, sighandler_t *);
```

- The header file <signal.h> defines two special dummy functions SIG_DFL and SIG_IGN for use as signal catching actions. For example:

```
signal(SIGALRM, SIG_IGN);
```

To kill or to really kill?

- The system call `kill()` is used to send a specified signal to a specified process. Given a process with id `pid`, and assuming that it is a child process or that it is owned by the user, here are some examples:

```
kill(pid, SIGTERM); // terminate process pid
kill(pid, SIGSTOP); // suspend process pid
kill(pid, SIGCONT); // restart process pid
```

- Special signals for killing (SIGKILL) or stopping (SIGSTOP) a process. These cannot be ignored by a process. The SIGTERM signal can be ignored or caught, so to really kill use SIGKILL!
- Linux has a command named `kill` that invokes the `kill()` system call.

```
kill -s signal pid
kill -l  --> list all signals
kill -9  --> send SIGKILL
```

- **In-Class Exercises (4)**. What do the following statements do?


```
kill(getpid(), SIGKILL);
kill(getpid(), SIGSTOP);
```

To kill gently....

- To kill a process use `kill -1` (SIGHUP) or `kill -15` (SIGTERM) first to give the process a chance to clean up before being killed (as those signals can be caught). If that doesn't work, then use `kill -9` to send SIGKILL signal that cannot be caught or ignored. In some circumstances, however, even SIGKILL doesn't work....

  Because I could not stop for Death,
  He kindly stopped for me;
  The carriage held but just ourselves
  And Immortality.
  ...
  Emily Dickinson

- Examples:
  - A simple signal handler example. Ignores `CTRL+c` and `CTRL+z` (and prints an annoying message): files-processes/sig-handler.c
  - A bigger example of systems programming. Sets a time limit on a process: files-processes/timeout.c

In-Class Exercises (5)

- **Autosave?** Consider the following C program sketch. Choose the right answer that explains what the code does.

```
int main() {
    /* ... */
    signal(SIGALRM, savestate);
    alarm(10);
    /* ... */
    for (;;) {
        /* do something */
    }
}
void savestate(int signo) {
    /* save the state to disk */
}
```

1. Saves the state of the program to disk every 10 seconds
2. Exits after saving the state of the program once to the disk after 10 seconds
3. Keeps running after saving the state of the program once to the disk after 10 seconds
4. Exits after saving the state of the program twice to the disk after 10 seconds
5. Never saves the state of the program to the disk

System Calls Introduced

- `exit()`
- `open()`, `creat()`, `close()`, `read()`, `write()`
- `fork()`
- `wait()`, `waitpid()`
- `execvp()`, `execlp()`
- `alarm()`
- `signal()`
- `getpid()`, `getppid()`
- `sleep()`, `kill()`

MS Windows API for Processes

- In MS Windows, the system call interface is not documented. Instead the MS Windows API is documented, which helps with portability across multiple versions of the MS Windows operating systems.
- Using the MS Windows API requires us to include `#include <windows.h>` header file.
- Creating a process gives a *handle* that is used to refer to the actual object that represents a process (or a thread). Most system calls use handles.
- CloseHandle(...). Frees the space used by the handle.
- Get detailed information from http://msdn.microsoft.com/library/

Calls in Linux versus Windows API

Linux	Windows API
`fork()`, `exec()`	`CreateProcess()` (fork and exec combined)
`exit()`	`ExitProcess()`
`wait()`, `waitpid()`	`WaitForSingleObject()`, `WaitForMultipleObjects()`
`WEXITSTATUS()` macro	`GetExitCodeProcess()`
`getpid()`	`GetCurrentProcessId()`
—	`GetCurrentProcess()` (returns process handle)
`getppid()`	No easy way to get parent process id!
`sleep()`	`Sleep()` (warning: this takes time in milliseconds!)
`kill()`	`TerminateProcess()`
`alarm()`	`CreateTimerQueueTimer()`
`signal()`	`signal()`

CreateProcess Call in MS Windows API


```
BOOL WINAPI CreateProcess(
  LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

Processes Related Calls in MS Windows API


```
WaitForSingleObject(hProcess, INFINITE);
```

```
CloseHandle(pi.hProcess);
```

```
DWORD WINAPI GetCurrentProcessId(void);
HANDLE WINAPI GetCurrentProcess(void);
```

```
VOID WINAPI ExitProcess(
  UINT uExitCode
);
```

```
BOOL WINAPI TerminateProcess(
  HANDLE hProcess,
  UINT uExitCode
);
```

```
BOOL WINAPI GetExitCodeProcess(
  HANDLE hProcess,
  LPDWORD lpExitCode
);
```

Structures Related to Processes


```
typedef struct _PROCESS_INFORMATION {
  HANDLE hProcess;
  HANDLE hThread;
  DWORD dwProcessId;
  DWORD dwThreadId;
} PROCESS_INFORMATION, *LPPROCESS_INFORMATION;
```

```
typedef struct _SECURITY_ATTRIBUTES {
  DWORD nLength;
  LPVOID lpSecurityDescriptor;
  BOOL bInheritHandle;
} SECURITY_ATTRIBUTES, *LPSECURITY_ATTRIBUTES;
```

Checking Errors in System Calls

- `DWORD GetLastErrorCode(void)`. Retrieves the calling thread's last-error code value. The last-error code is maintained on a per-thread basis. Multiple threads do not overwrite each other's last-error code. This function should be called right after a system call returns an error (usually we know that from a negative return value from the system call).
- To obtain an error string for system error codes, use the `FormatMessage` function.

```
DWORD FormatMessage(
  DWORD dwFlags,
  LPCVOID lpSource,
  DWORD dwMessageId,
  DWORD dwLanguageId,
  LPTSTR lpBuffer,
  DWORD nSize,
  va_list* Arguments
);
```

Sample Error Code


```
void ErrSys(char *szMsg)
{
    LPVOID lpMsgBuf;

    // Try to format the error message from the last failed call
    //  (returns # of TCHARS in message -- 0 if failed)
    if (FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER | // source and processing options
        FORMAT_MESSAGE_FROM_SYSTEM |
        FORMAT_MESSAGE_IGNORE_INSERTS,
        NULL,                              // message source
        GetLastError(),                    // message identifier
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),  // language (Default)
        (LPTSTR) &lpMsgBuf,                          // message buffer
        0,    // maximum size of message buffer
              // (ignored with FORMAT_MESSAGE_ALLOCATE_BUFFER set)
        NULL  // array of message inserts
    ))
    {
      // Display the formatted string with the user supplied string at front.
      fprintf(stderr, "%s: %s\n", szMsg, (LPSTR)lpMsgBuf);
      LocalFree(lpMsgBuf); // Free the buffer.
    } else {
        fprintf(stderr, "%s: Could not get the error message!\n",  szMsg);
    }
    fflush(NULL); /* flush all output streams */
    ExitProcess(1); /* exit abnormally */
}
```

Using MS Visual Studio (1)

- Visual Studio is available via the Dream Spark program from the college.
- Start up Visual Studio. Choose *New Project* → *Visual C++* → *Win32* → *Win32 Console Project*.
- In the Wizard window, choose *Application Settings* → *Empty Project* → *Finish*.
- Right click on the project in the right pane (*Solution Explorer*) and then choose *Add* → *Add Existing Item...*. Note that this doesn't copy the file into the Visual Studio project folder.
- To copy the file, right click on the project name in the Solution Explorer and choose the *Open Folder in File Explorer*. Then copy files into the project folder under the solution folder. Back in Visual Studio, select *Project* → *Show All Files*. Then go to the Solution Explorer pane, right click on the file(s) you have added and choose *Include File in Project* option.

Using MS Visual Studio (2)

- Also note that, Visual Studio uses Unicode by default. For now, we will simply turn this off. Press *ALT+F7* to open the project properties, and navigate to *Configuration Properties* → *General*. Switch *Character Set* to *Multi-Byte Character Setting* from the drop-down menu. The examples provided will not work unless you choose the multi-byte setting!
- If you have several small programs in one folder, we can create a single solution with multiple projects in it to keep it more organized. After creating an empty solution, simply add new projects to it by right-clicking on the solution name in the Solution Explorer pane.
- Note that with multiple projects in one solution, we have to choose which one is the startup project. Right-click on the project name in the Solution Explorer and choose the option *Set as Startup Project*.
- **Tip**. If you want to know definition of MS Windows API typedefs, right-click on the type (e.g. LPVOID) and select "go to definition" from the drop down menu.
- **Tip**. `Ctrl-Space` does code completion (just like in Eclipse)

MS Windows API Examples

- `lab/ms-windows/files-processes/fork-and-wait.c`
- `lab/ms-windows/files-processes/fork-and-exec.c`
- `lab/ms-windows/files-processes/fork-hello-world.c`
- `lab/ms-windows/files-processes/fork-test.c`
- `lab/ms-windows/files-processes/shell1.c`
- `lab/ms-windows/files-processes/alarm-test.c`
- `lab/ms-windows/files-processes/timeout.c`
- and others in the ms-windows/files-processes examples folder....

Microsoft PowerShell

Powershell is a shell, for Microsoft platforms, with a command-line and a built-in scripting language.

- Aliases are built-in for common commands used in bash with Unix/Linux/Mac OSX systems. For example, TAB is used for command completion and aliases exist for `ls`, `cp`, `man`, `date` etc.
- Pipes are also supported but they pass objects instead of unstructured text streams.
- Includes a dynamically typed scripting language with .NET integration. Here is a simple example of a loop:

  ```
  while ($true) {.\fork-hello-world; echo ""}
  ```

- The following shows how to time a command or script in powershell:

  ```
  Measure-Command {sleep 2}
  ```

- Powershell script files are text files with a `.ps1` extension. By default, you cannot run scripts unless they are signed. To enable it, use the following command:

  ```
  Set-ExecutionPolicy RemoteSigned
  ```

  Use the command Get-Help About_Signing to learn more about signing.