CSCI 350
Ch. 3 Programmer's Interface

Mark Redekopp

Getting on the Same Page

- What is a thread?

- What is a process?

- What is user mode vs. kernel mode?

Getting on the Same Page

- What is a thread?
  - Independently scheduled task
  - PC + state (i.e. registers, stack, etc.)
- What is a process?
  - Instance of an executable program
  - Threads + independent address space
- What is user mode vs. kernel mode?
  - User mode = non-priviledged execution environment
  - Kernel mode = privileged execution environment where the OS executes

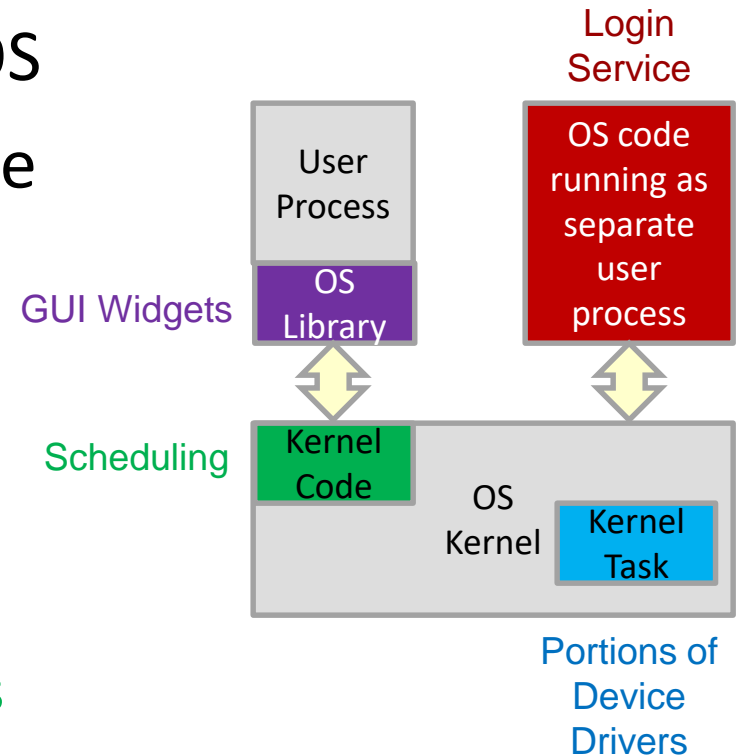Chapter 3 of "Operating Systems…", Anderson and Dahlin

PROGRAMMING INTERFACE

Overview

- OS must provide services to a user application
  - Process and thread management
  - Input/output (file systems, network, etc.)
  - Memory management
  - Authentication and security
 - Graphics and window management

Where Services Might Execute

- Many services must be provided to user level processes by the OS

- Options exist for where to locate those services
  - Directly in the user process by linking in a user-level OS library
  - Separate user level processes
  - Part of the kernel executed in the control flow of the calling process
  - Part of the kernel running as a separate kernel task

Login Service

OS code running as separate user process

User Process

GUI Widgets

OS Library

Scheduling

Kernel Code

OS Kernel

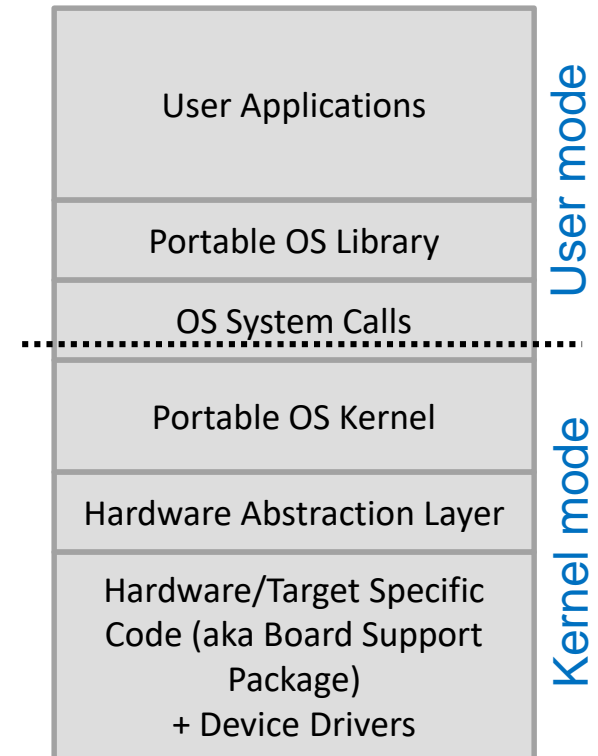Kernel Task

Portions of Device Drivers

Considerations

- Evaluate where services should be located based on:
  - **Safety**, **Flexibility**, **Performance**
- Pros of locating services in a user-level process
  - Safety/Isolation from other kernel structures (i.e. a bug cannot bring down the rest of the kernel)
  - Flexibility: Easier to update and provide new versions of functionality (no recompilation of kernel needed)
- Cons of location services in a user-level process
  - Performance: Overhead of moving data and switching contexts
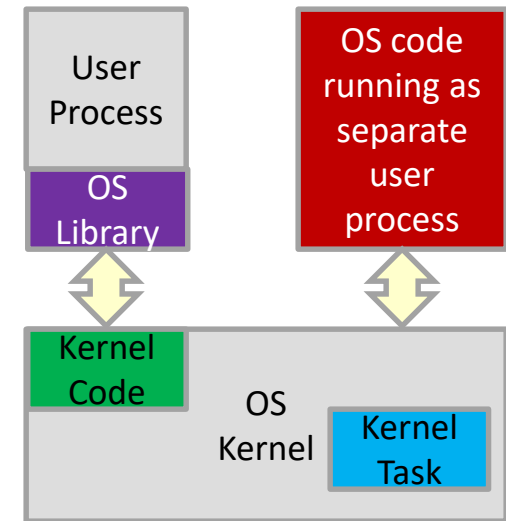
Common OS Layers/Architecture

- System calls are the "interface" between user applications and the kernel
  - OS may provide libraries that provide some services and abstraction above the system call level
  - Changing the interface of system calls has a major impact on software
    - Likely requires recompile/update of software applications

User mode
User Applications
Portable OS Library
OS System Calls

Kernel mode
Portable OS Kernel
Hardware Abstraction Layer
Hardware/Target Specific Code (aka Board Support Package) + Device Drivers

Meet the syscalls

- Process management
  - fork() : new_pid
  - exec(char* exec, char** args)
  - wait(pid)
  - exit()
- I/O
  - open(name) : fd
  - read(fd, buffer, size) : int
  - write(fd, buffer, size) : int
  - close(fd)
  - pipe(fd[2])
  - select(fd_array[], fd_array_size) : fd
  - dup2(fromFd, toFd)

Some definitions
- **Address space** = Protected (hardware checked) memory ranges accessible only to a single process (and possibly the kernel)
- **pid** = Process ID (unique identifier of a process)
- **fd** = File descriptor (identifier to lookup data structure holding the state of I/O access to a file, network socket, "pipe", etc.)
  - Often just an integer (index) into some table of descriptors in the kernel
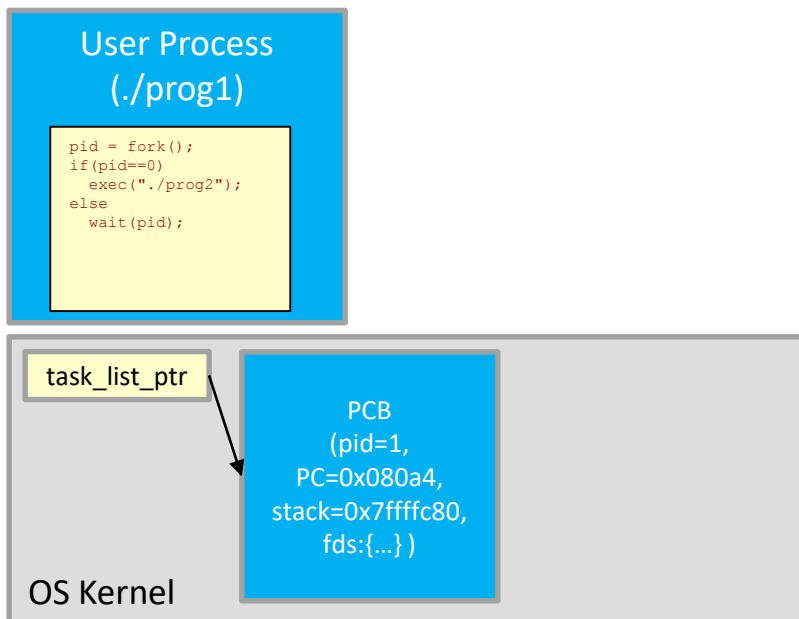
fork, exec, wait, exit

SYSCALLS FOR PROCESS MANAGEMENT
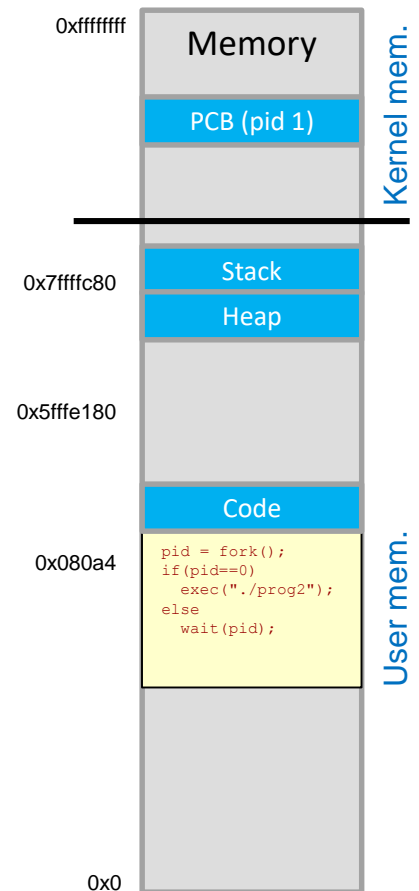
Process Management

- Most OSs allow one user process to create another through various system calls
  - Rather than the kernel initiating all process creations
- Windows approach
  - Perform process creation, environment setup, and program startup through a **<u>single</u>** system call
    - Single `createProcess(…)` system call
- Unix approach
  - Separate process creation, environment setup, and program startup with **<u>separate</u>** system calls
    - Create process with `fork()` system call
    - Setup environment with possible calls to `open(), close(), dup2()`
    - Load program image and start execution with `exec(…)` system call

Unix Process Syscalls

- Suppose a process (pid=1) is running and wants to start another

Abstract View

User Process
(./prog1)

```
pid = fork();
if(pid==0)
  exec("./prog2");
else
  wait(pid);
```

task_list_ptr

PCB
(pid=1,
PC=0x080a4,
stack=0x7ffffc80,
fds:{...})

OS Kernel

Process 1 AS

0xffffffff — Memory

PCB (pid 1)

Kernel mem.

0x7ffffc80 — Stack

Heap

0x5fffe180

Code

0x080a4
```
pid = fork();
if(pid==0)
  exec("./prog2");
else
  wait(pid);
```

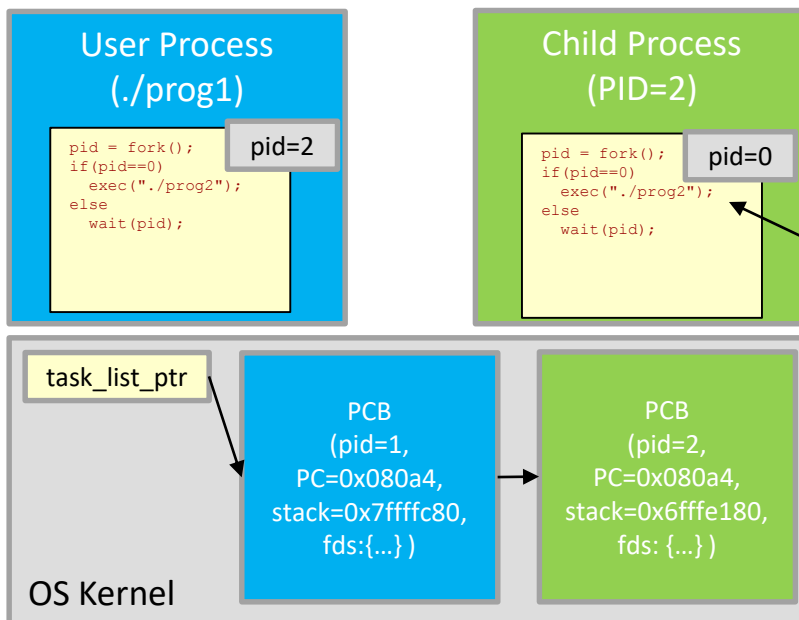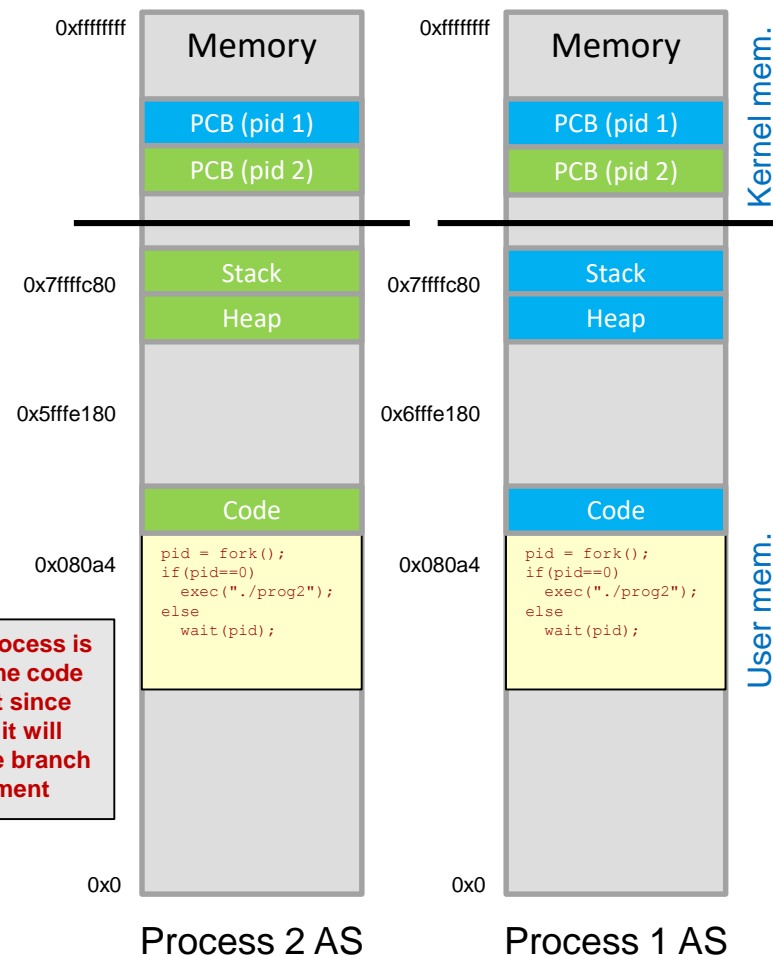User mem.

0x0

Fork

- A call to **fork** copies the state (exact replica) of the current (**parent**) process to a new **child** process
    - Returns new pid (e.g. 2) to the parent
    - Returns 0 to the child process (it can retrieve its own pid with another system call if it needs that info)
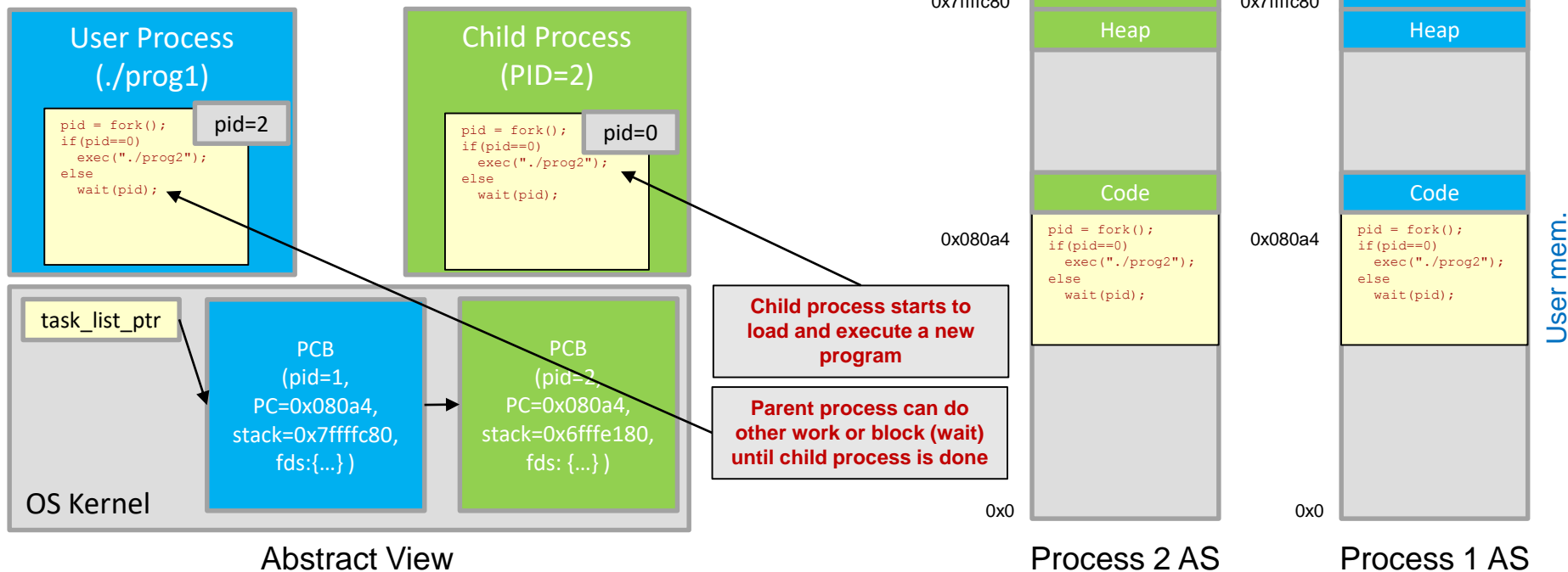
User Process (./prog1)

```
pid = fork();
if(pid==0)
  exec("./prog2");
else
  wait(pid);
```
pid=2

Child Process (PID=2)

```
pid = fork();
if(pid==0)
  exec("./prog2");
else
  wait(pid);
```
pid=0

task_list_ptr

PCB
(pid=1,
PC=0x080a4,
stack=0x7ffffc80,
fds:{...})

PCB
(pid=2,
PC=0x080a4,
stack=0x6fffe180,
fds: {...})

OS Kernel

Abstract View

Notice the child process is executing the same code as the parent but since 'pid' is different it will execute a separate branch of the 'if' statement

Kernel mem.

User mem.

0xffffffff Memory
PCB (pid 1)
PCB (pid 2)

0x7ffffc80 Stack
Heap

0x5fffe180

Code

0x080a4
```
pid = fork();
if(pid==0)
  exec("./prog2");
else
  wait(pid);
```

0x0

Process 2 AS

0xffffffff Memory
PCB (pid 1)
PCB (pid 2)

0x7ffffc80 Stack
Heap

0x6fffe180

Code

0x080a4
```
pid = fork();
if(pid==0)
  exec("./prog2");
else
  wait(pid);
```
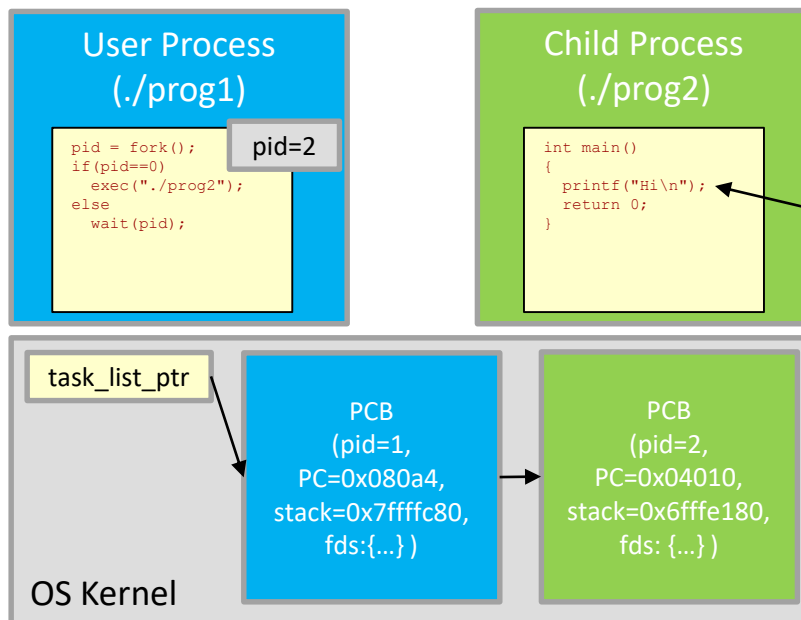
0x0

Process 1 AS

Exec 1

- A call to **exec** now loads a new program (e.g. ./prog2) with its own code, sets up its global data, stack, heap, etc.
  - **exec** returns only if an error occurs
- Parent can continue execution or block until the child process finishes by calling **wait**

Child process starts to load and execute a new program

Parent process can do other work or block (wait) until child process is done

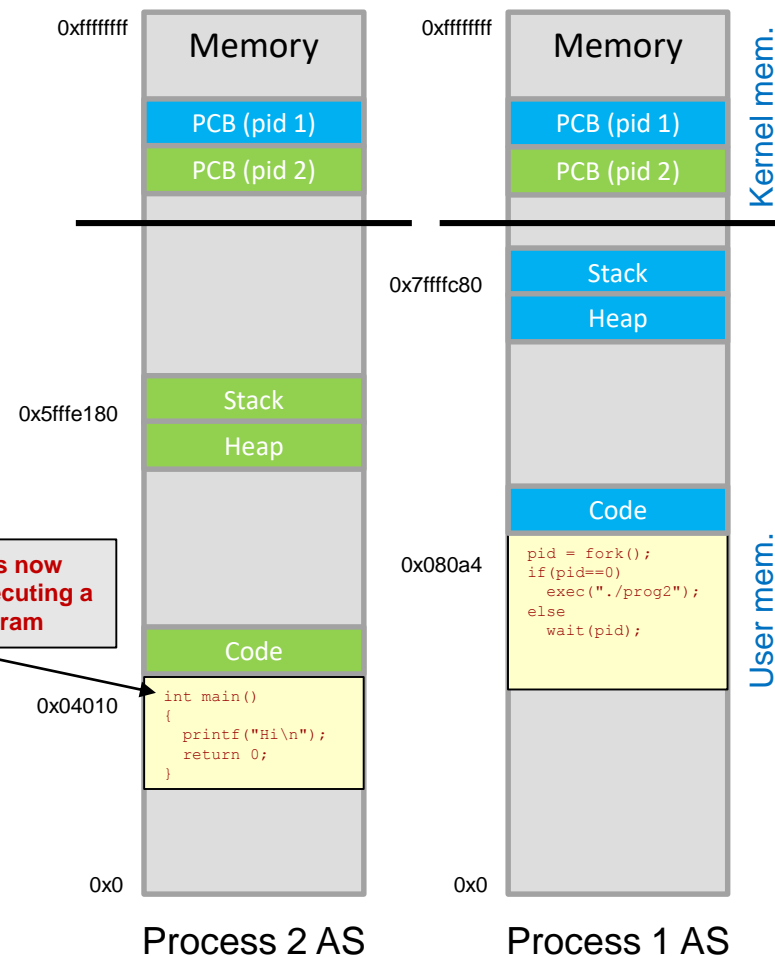Abstract View

Process 2 AS

Process 1 AS

Exec, Wait, and Exit

- The effect of a call to **exec** is a separate program image is loaded and begins execution

- If the parent calls **wait** then it will block until the child process calls **exit**
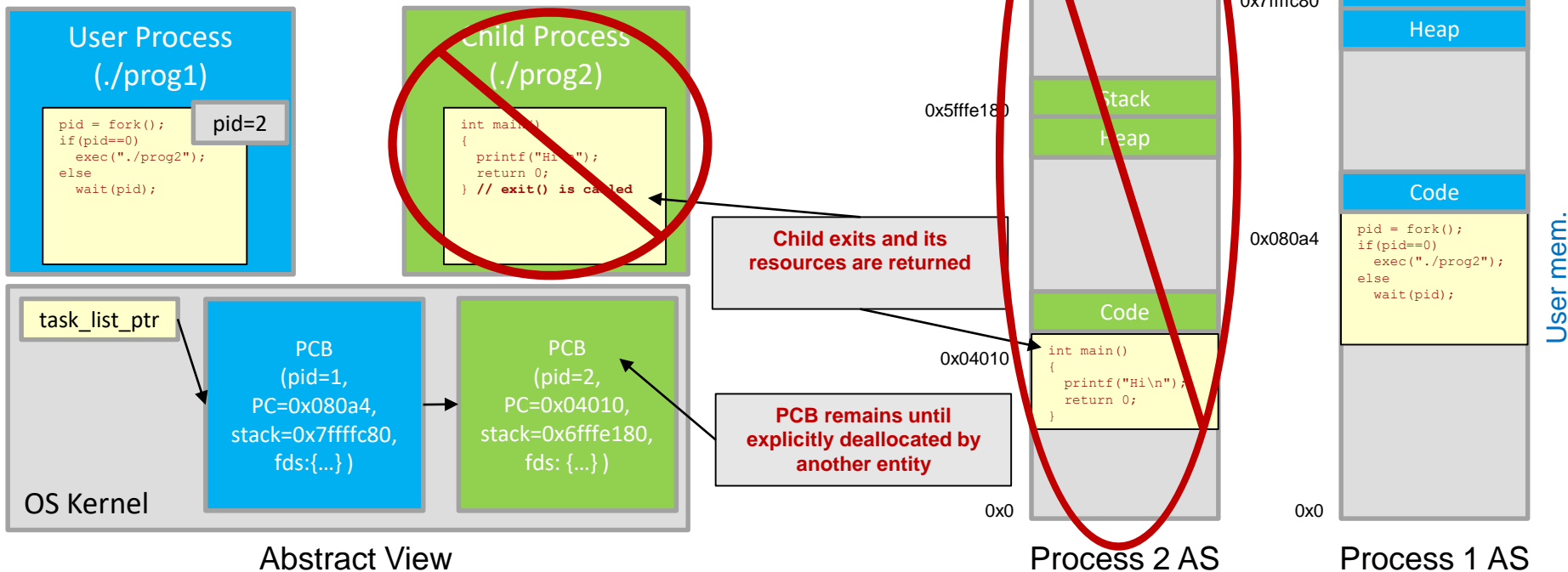 - **exit** is generally called when the program finishes main()

User Process (./prog1)

```
pid = fork();
if(pid==0)
  exec("./prog2");
else
  wait(pid);
```
pid=2

Child Process (./prog2)

```
int main()
{
  printf("Hi\n");
  return 0;
}
```

Child process is now independently executing a separate program

task_list_ptr

PCB
(pid=1,
PC=0x080a4,
stack=0x7ffffc80,
fds:{…})

PCB
(pid=2,
PC=0x04010,
stack=0x6fffe180,
fds: {…})

OS Kernel

Abstract View

Kernel mem.

User mem.

0xffffffff Memory   0xffffffff Memory

PCB (pid 1)   PCB (pid 1)
PCB (pid 2)   PCB (pid 2)

                        0x7ffffc80   Stack
                                     Heap

0x5fffe180   Stack

             Heap

                        0x080a4   Code
                                     ```
                                     pid = fork();
                                     if(pid==0)
                                       exec("./prog2");
                                     else
                                       wait(pid);
                                     ```

             Code
0x04010   ```
          int main()
          {
            printf("Hi\n");
            return 0;
          }
          ```

0x0                      0x0

Process 2 AS            Process 1 AS

# Exit

- **exit** deletes the current process and its resources (i.e. address space, etc.)

- PCB is not deallocated until parent or other entity specifically release it
  - This allows the parent or other task to examine the child process' exit status, etc.



User Process
(./prog1)

```
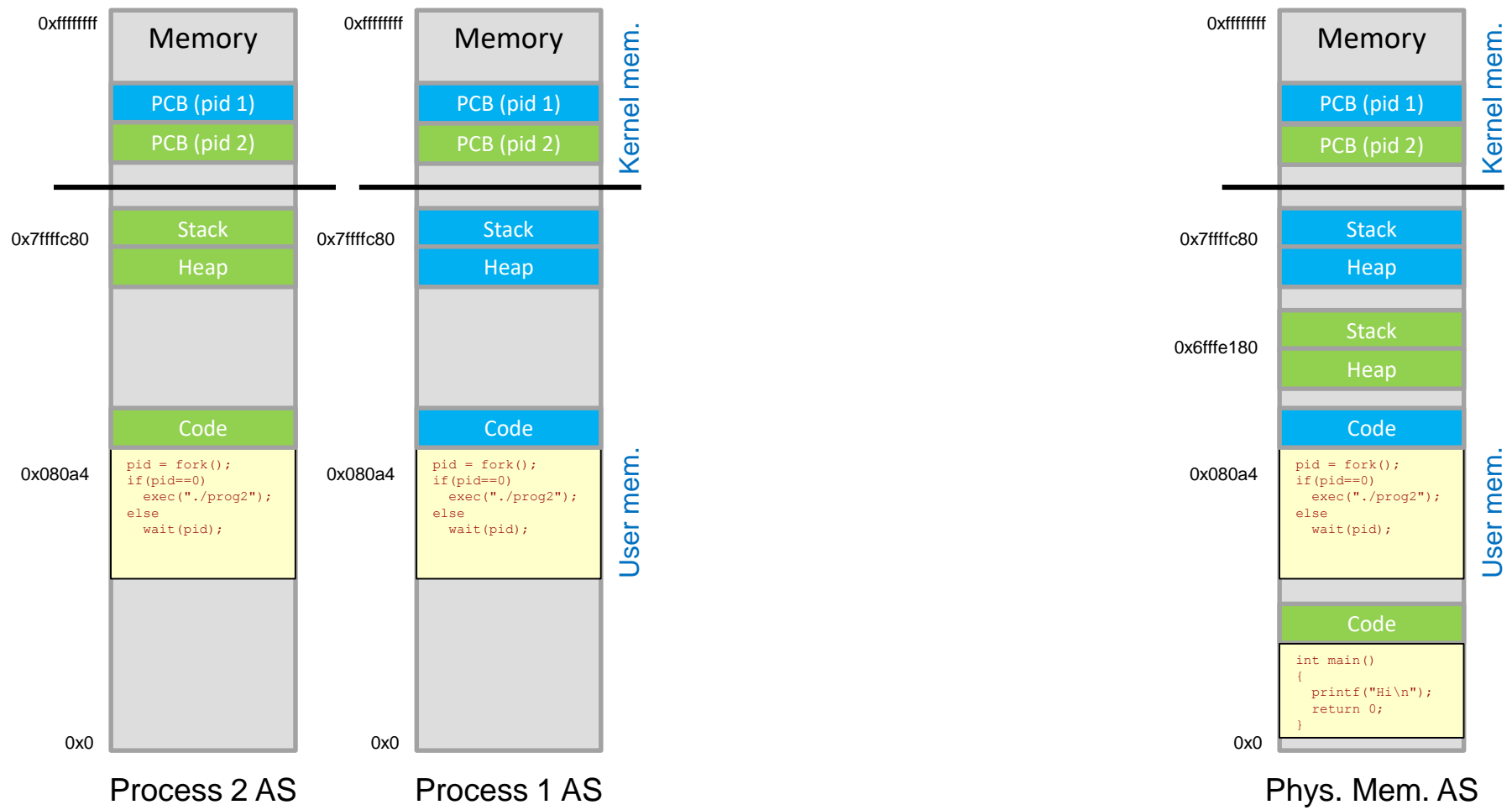pid = fork();
if(pid==0)
  exec("./prog2");
else
  wait(pid);
```

pid=2

Child Process
(./prog2)

```
int main()
{
  printf("Hi\n");
  return 0;
} // exit() is called
```

task_list_ptr

PCB
(pid=1,
PC=0x080a4,
stack=0x7ffffc80,
fds:{…})

PCB
(pid=2,
PC=0x04010,
stack=0x6fffe180,
fds: {…})

Child exits and its resources are returned

PCB remains until explicitly deallocated by another entity

OS Kernel

Abstract View

0xffffffff

Memory

PCB (pid 1)

PCB (pid 2)

0x5fffe180

Stack

Heap

Code

0x04010

```
int main()
{
  printf("Hi\n");
  return 0;
}
```

0x0

Process 2 AS

0xffffffff

Memory

PCB (pid 1)

PCB (pid 2)

0x7ffffc80

Stack

Heap

Code

0x080a4

```
pid = fork();
if(pid==0)
  exec("./prog2");
else
  wait(pid);
```

0x0

Process 1 AS

Kernel mem.

User mem.

Virtual Memory (Backup)

- Virtual Memory allows multiple "virtual" address spaces to be mapped to a single physical address space

Summary

- **Fork** causes the kernel to
    - Create an initialize the PCB entry for a new process
    - Create a new address space
    - Initialize the new address space with a complete copy of the parent's address space
    - Inherit the execution context of the parent (open files, etc.)
    - Inform the scheduler that the new process is ready to run
- **Exec** causes the kernel to
    - Load a new program image into the current address space
    - Copy arguments into the address space
    - Initialize the hardware context (PC + stack) to the "start" entry point of the program image

open, close, read, write, select, dup2

SYSCALLS FOR INPUT/OUTPUT

Problem with I/O

- There are a vast number of input/output devices that may be connected to a computer and many possible interfaces
  - A magnetic disk may want a group of values (think struct) that indicate the location (sector, cylinder, etc.) and amount of data to read synchronously
  - A keyboard device may asynchronously supply a byte of data at a time
- Is there a single API that can generally fit for all these different devices?

Byte / Block oriented

Synchronous/ Asynchronous

Speed

Common I/O Syscall Characteristics

- Unix and many other OSs try to interact with all I/O devices with a common set of syscalls

- Most OSs I/O syscall characteristics
  - Accessed via **file descriptors**
    - Files, network sockets, pipes (more in a moment), etc. are all identified with a file descriptor and a common set of syscalls
    - **file descriptor** is a handle or index (usually just an int) to the internal bookkeeping information and state of the I/O connection
  - **Byte oriented**: Whether you need to pass a single byte, an array of bytes, or a struct the API is just a pointer to the start byte and total size
  - **Open/close**: Open/close allows kernel to setup internal bookkeeping (file position, etc.) and access control to a device before read/writes are allowed
    - Open returns a **file descriptor**

```
struct DiskOp
{
  uint8_t flags;
  uint8_t padding[3];
  uint32_t sector;        Login
  uint32_t cylinder;      Service
};
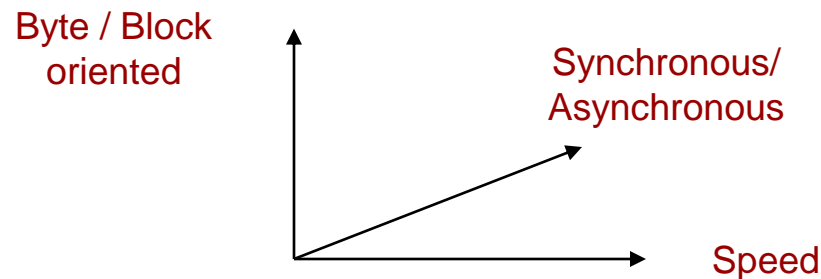
struct DiskOp myop;
```

flags	padding
sector	
cylinder	

Memory View

```
// Call OS syscall
write(fd,
      &myop,
      sizeof(DiskOp));
```

0	1	2	3
file: /dev/ttyS0 fileOffset: 0	file: /home/user/hi.txt fileOffset: 4	file: /tmp/myfifo fileOffset: 8	file: /dev/sda1

Common I/O Syscall Characteristics

- Unix and many other OSs try to interact with all I/O devices with a common set of syscalls

- Most OSs I/O syscall characteristics

  - **Kernel buffering**: Input data is internally buffered in the kernel before the user process can read. Output data is buffered in the kernel as well and then passed to the I/O device from the kernel

  - Provides decoupling of speed (flow control)

0xffffffff

Memory

Kernel I/O Buffer

Kernel mem.

0x7ffffc80

Stack

Heap

0x6fffe180

Code

0x080a4

```
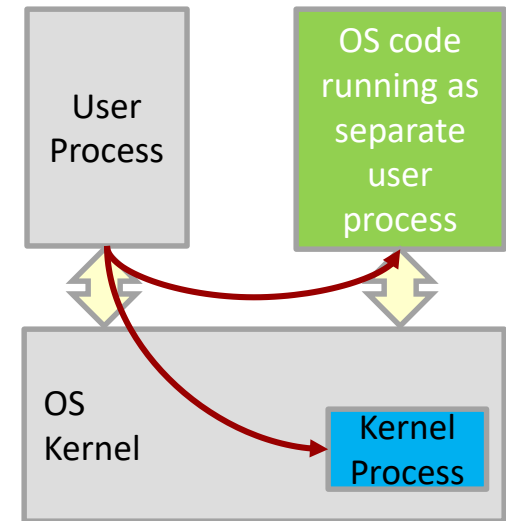// other work
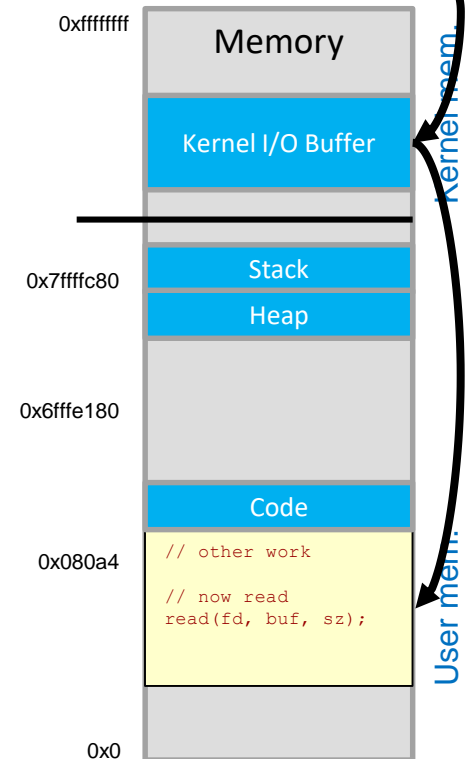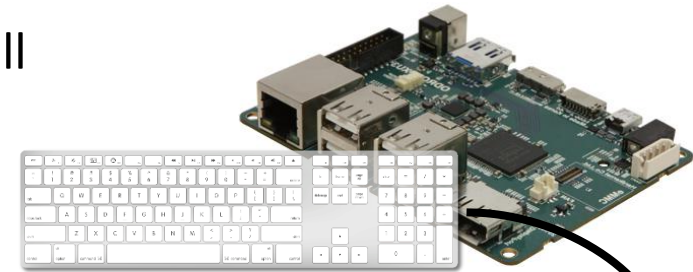
// now read
read(fd, buf, sz);
```

0x0

User mem.

USC Viterbi
School of Engineering

Common I/O Syscalls

- open(name) : fd
  - Returns a new descriptor to the device with path "name"
  - In Unix all devices map to a filename (i.e. a USB serial connection lives at /dev/tty.usbmodemXXX)
- read(fd, buffer, size) : int
  - Reads at most size bytes of data into buffer from the device specified by fd, returning the number of bytes *actually* read
- write(fd, buffer, size) : int
  - Writes size bytes from buffer to the device specified by fd
- close(fd)
- select(fd_array[], fd_array_size) : fd
  - Waits for data to be available for reading on any of the fds in fd_array and returns the first fd that can be read
- dup2(fromFd, toFd)
  - Copies the state (position/status) of the file specified by fromFd to the descriptor located at toFd
  - Often used to redirect stdin and stdout of a process
- pipe(fd[2])
  - Creates a pipe for unidirectional communication between two processes (fd[0] is the read side of the pipe and fd[1] the write side)

Inter-Process Communication (IPC)

- A core service a kernel must provide is a method of inter-process communication (IPC)

- If OS services run as separate processes then we need a way to communicate between those processes

- Common models
  - Producer-consumer
  - Client/server
  - Regular files

Pipes

- A pipe is a unidirectional buffer for communication between two processes
  - Uses two file descriptors (one for the write side and one for the read side)
  - Acts as a queue/stream
  - Can be "named" so processes know who to connect to
    - Mapped to the file system as in `/tmp/myfifo`
- Sample code
  - http://man7.org/linux/man-pages/man2/pipe.2.html

Producer

User Process

Consumer

OS code running as separate user process

OS Kernel

Pipe

Client/Server

- 2-way communication:
  - Client requests
  - Server responds
- Some methods
  - Message queues, sockets, shared memory, etc.
- select() syscall
  - Blocks until activity on 1 of N descriptors
  - ```
    int select(int nfds, fd_set *readfds,
    fd_set *writefds, fd_set *exceptfds,
    struct timeval *timeout);
    ```
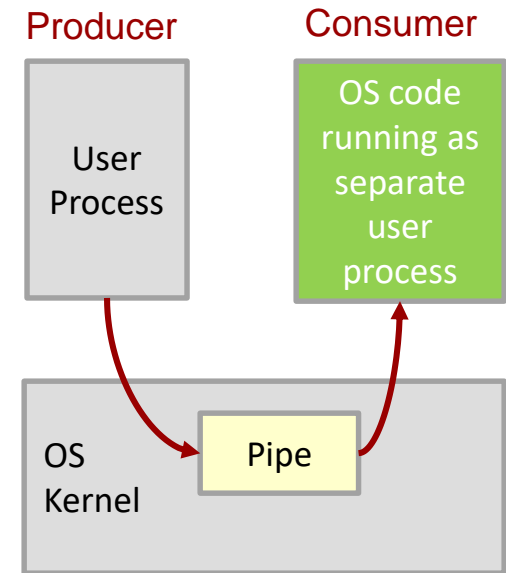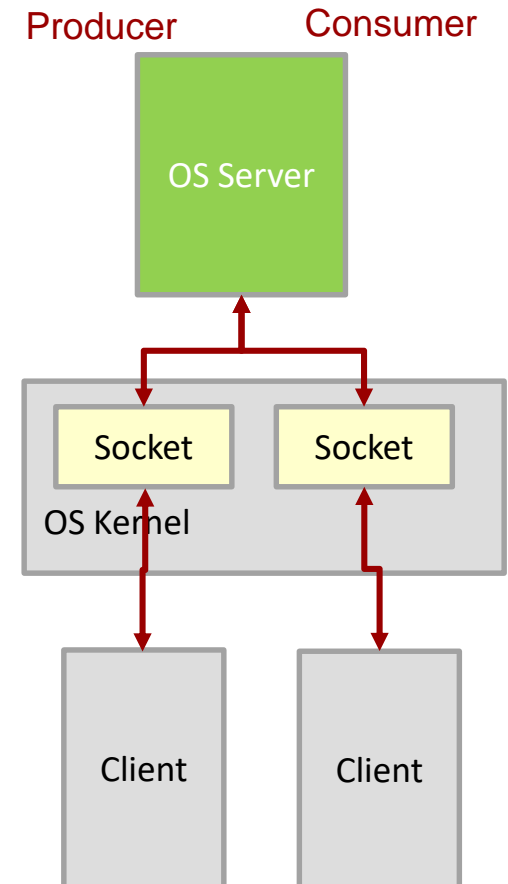
Producer   Consumer

OS Server

Socket   Socket

OS Kernel

Client   Client

Socket, Select() Example

- Select waits for any event to occur on a set of descriptors
  - Can be file, socket, pipe, etc. descriptors

```
int listener = _srvsocket->getSockDesc();
fd_set read_fds;
FD_ZERO(&read_fds);
FD_SET(listener,&read_fds);
int fdmax = listener;

while(1) {
  memcpy(&read_fds, &master, sizeof(master));
  if( select(fdmax +1,&read_fds,NULL,NULL,NULL) == -1){
    cerr << "Error: select" << endl;
    return 1;
  }
  for(int i = 0; i<=fdmax;i++){
    if(FD_ISSET(i,&read_fds)){
#ifdef DEBUG
      cerr << "FD " << i << " is set"<< endl;
#endif
      // A new connection is being made
      if(i == listener){
          int new_sock = accept(listener, ...);
          fd_max = max(listener, new_sock);
          FD_SET(int new_sock,&read_fds);
      }
      // Current connection, message event
      else {
          read(i, ...);
      }
    }
  }
}
```

CASE STUDY 1: SHELLS

Shells

- The shell/terminal/command line is just a user process that reads inputs that indicate other programs to run (i.e. command lines)
  - $ ./prog1 hello 5
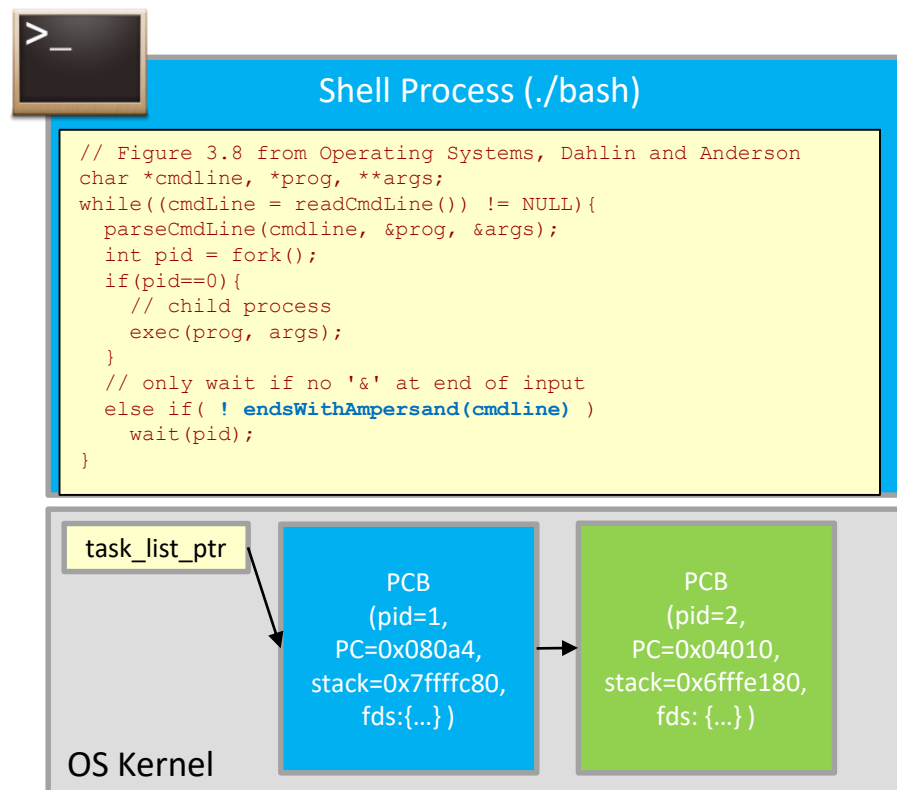- The shell then forks a new process and execs the specified program

Shell Process (./bash)

```
// Figure 3.8 from Operating Systems, Dahlin and Anderson
char *prog, **args;
while(readAndParseCmdLine(&prog, &args)){
  int pid = fork();
  if(pid==0) {
    // child process
    exec(prog, args);
  }
  else
    wait(pid);  // parent process
}
```

Child Process (./prog)

```
int main()
{
  printf("Hi\n");
  return 0;
}
```

task_list_ptr

PCB
(pid=1,
PC=0x080a4,
stack=0x7ffffc80,
fds:{…})

PCB
(pid=2,
PC=0x04010,
stack=0x6fffe180,
fds: {…})

OS Kernel
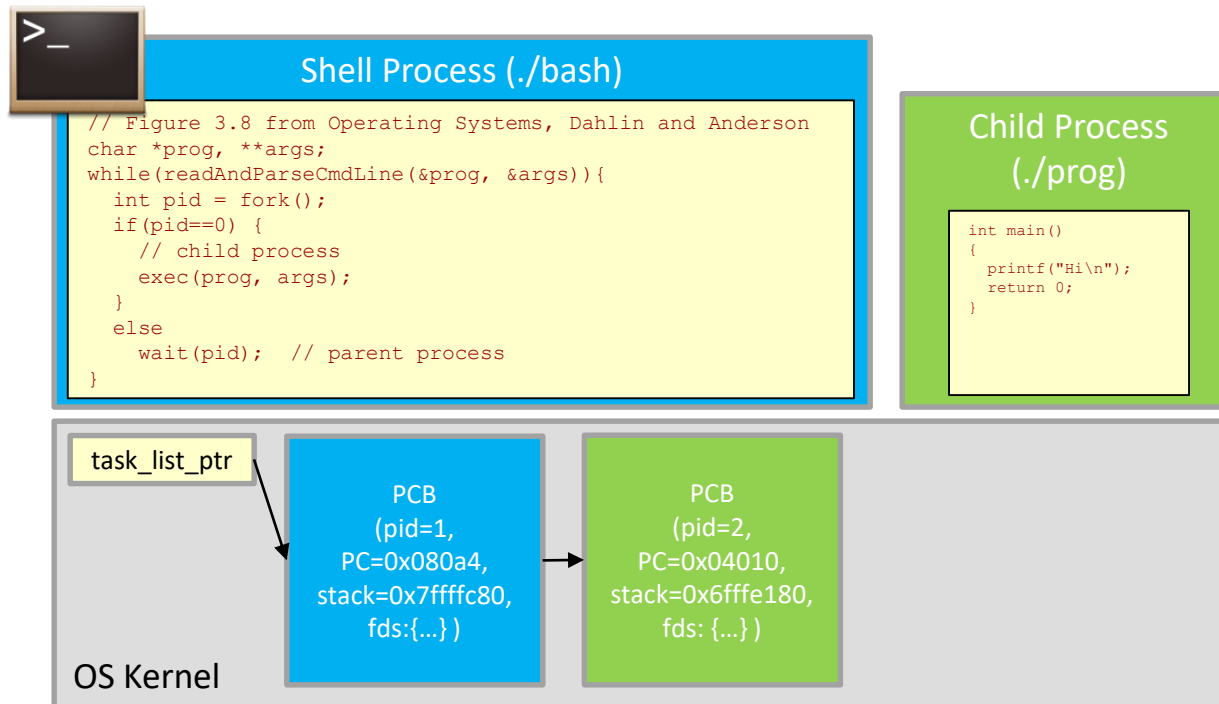
Abstract View

stdin and stdout

- Processes generally have two given file descriptors
  - **stdin (fd=0)**: read input from the system (generally the keyboard)
  - **stdout (fd=1)**: write output to the system (generally the terminal/screen)
- Shells usually allow redirection of stdin and stdout

Process Management Review

- Recall Unix allows the parent process to perform child process setup before it execs the program
  - Separate process creation, environment setup, and program startup with **<u>separate</u>** system calls
    - Create process with `fork()` system call
    - <u>Setup environment with possible calls to `open(), close(), dup2()`</u>
    - Load program image and start execution with `exec(…)` system call
- Shells can use this capability to perform I/O redirection and/or piping
  - $ ./prog1 > output.txt
    - Redirects stdout to output.txt file
  - $ ./prog1 < input.txt
    - Redirects stdin to come from input.txt
  - $ ./prog1 < input.txt > output.txt
    - Redirects both input and output
  - $ ./prog1 | ./prog2
    - Pipe output (stdout) of prog1 as the input (stdin) of prog2

To Wait or Not

- Recall how many shells allow you to continue processing new shell commands while the exec'd program runs by placing '&' at the end of the command line
  - $ ./prog1 hello 5 &
- This just causes the shell to skip the call to **wait**

Shell Process (./bash)

```
// Figure 3.8 from Operating Systems, Dahlin and Anderson
char *cmdline, *prog, **args;
while((cmdLine = readCmdLine()) != NULL){
  parseCmdLine(cmdline, &prog, &args);
  int pid = fork();
  if(pid==0){
    // child process
    exec(prog, args);
  }
  // only wait if no '&' at end of input
  else if( ! endsWithAmpersand(cmdline) )
    wait(pid);
}
```

task_list_ptr

PCB
(pid=1,
PC=0x080a4,
stack=0x7ffffc80,
fds:{…})

PCB
(pid=2,
PC=0x04010,
stack=0x6fffe180,
fds: {…})

OS Kernel

I/O Redirection

- Redirection can be applied before the child process calls **exec** by replacing the stdin and stdout file descriptors with those of the specified files
   - $ ./prog1 < input.txt > output.txt
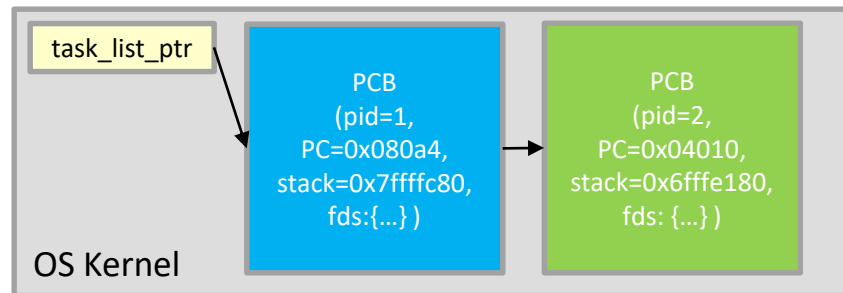
Piping

- Challenge: Consider how a shell would handle piping
  - ./prog1 | ./prog2
- High-level approach (order may vary)
  - Create a pipe which returns two fd's (for the read and write sides of the pipe)
  - Fork each process (prog1 and prog2)
  - Set the write fd of the pipe as the stdout fd of prog1
  - Set the read fd of the pipe as the stdin fd of prog2
  - Exec prog1 and wait for it to complete
  - Exec prog2 and wait for it to complete
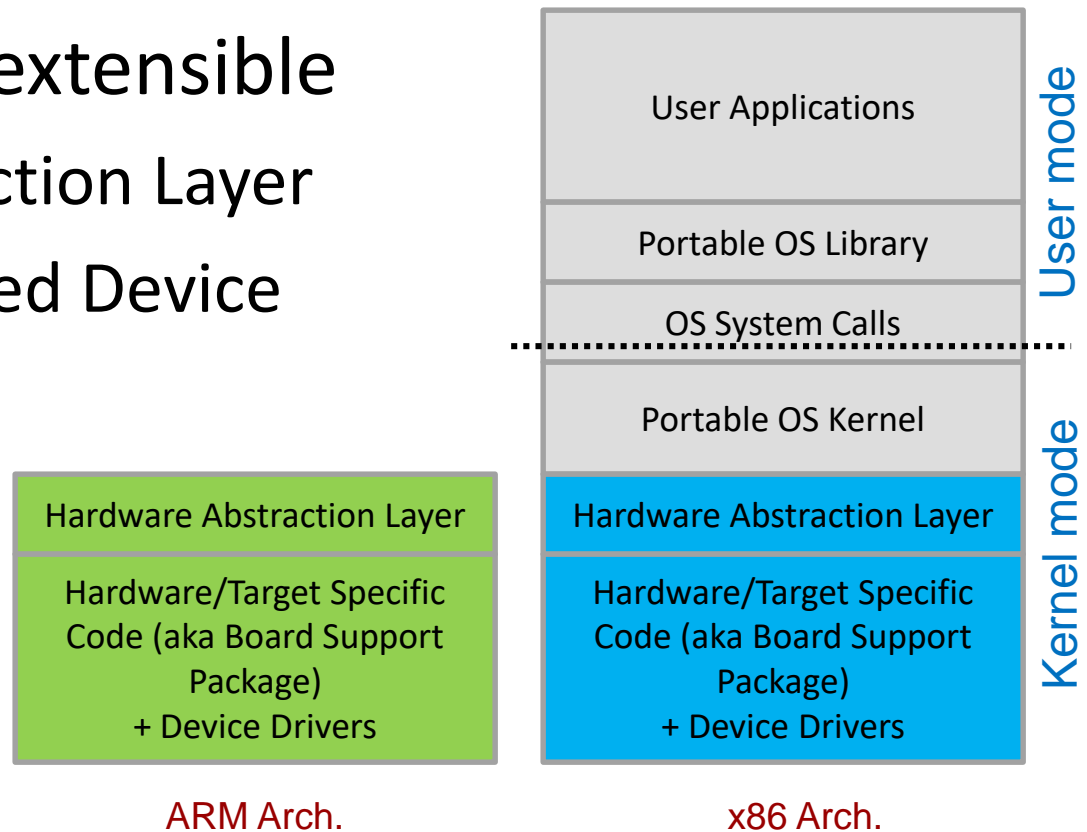
KERNEL ORGANIZATION

Approaches

- Monolithic Kernel
  - Majority of services compiled and execute as part of the kernel (not as a user service) in the same address space
  - Most commercial operating systems
    - Why? What get's optimized: safety, flexibility, performance?
  - Generally allows better performance
- Microkernel
  - Majority of services run as user-level processes (separate address spaces)
  - Allows better safety and some flexibility
  - Ex. Mach, NeXT
- Syscalls can make the differences transparent

Monolithic Kernel

MicroKernel
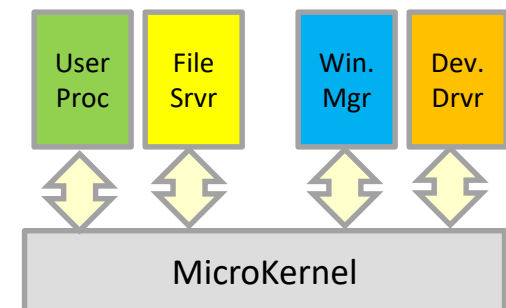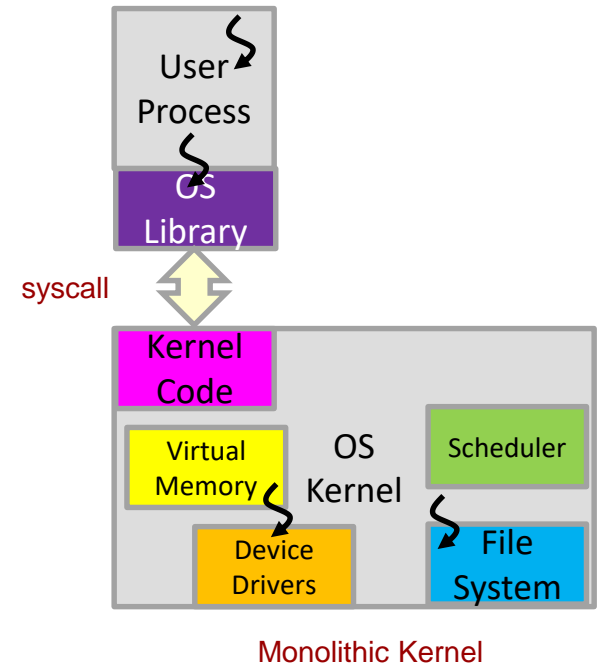
Monolithic Kernel

- General features to make them portable & extensible
  - Hardware Abstraction Layer
  - Dynamically loaded Device Drivers

User mode
User Applications
Portable OS Library
OS System Calls

Kernel mode
Portable OS Kernel
Hardware Abstraction Layer
Hardware/Target Specific Code (aka Board Support Package) + Device Drivers

ARM Arch.

Hardware Abstraction Layer

Hardware/Target Specific Code (aka Board Support Package) + Device Drivers

x86 Arch.

Hardware Abstraction Layer

- Provide functions and abstract models of devices
  - OS is written to use these functions & models
  - To support a new HW target just re-implement those functions

- Examples:
  - Interrupt handling
  - System call handling (low level)
  - Process context switches
  - Low-level VM operations like loading a page table etc.
  - Timer handling

http://elixir.free-electrons.com/linux/v4.11.3/source

Dynamically Loadable Device Drivers

- 70% of Linux source code is drivers
  - In many general purpose systems we won't know all the attached HW devices
  - So we can't provide an OS with everything built in

- Dynamically Loadable Device Drivers
  - Not compiled directly into kernel but loaded as a shared library when needed
  - On boot or hot plug, OS discovers devices and then loads appropriate drivers
  - Common interface to allow applications to be written to interface to various HW devices regardless of actual implementation

Device Driver Issues

- According to OS:PP 2nd Ed. up to 90% of system crashes are due to device drivers, not kernel
  - Bugs in 3rd party device driver code
  - OS Updates cause device drivers to not work
- Mitigations
  - Code inspection (Linux requires peer evaluation of driver code, and then driver becomes part of Linux and is actively maintained in each kernel release)
  - Bug Tracking (Reports sent to vendor upon crash)
  - User-level device drivers
    - Provide flexible syscalls so that device drivers can run in user mode and thus prevent kernel crashes
  - Virtual machine or sandboxed device drivers
    - Run device driver in a protected environment

Resources & Examples

- Process Control Block (task_struct): /usr/src/linux-headers-3.13.0-24-generic/include/linux/sched.h
  - Around line 1042
- Syscalls: http://man7.org/linux/man-pages/man2/syscalls.2.html
  - Defined in <unistd.h>