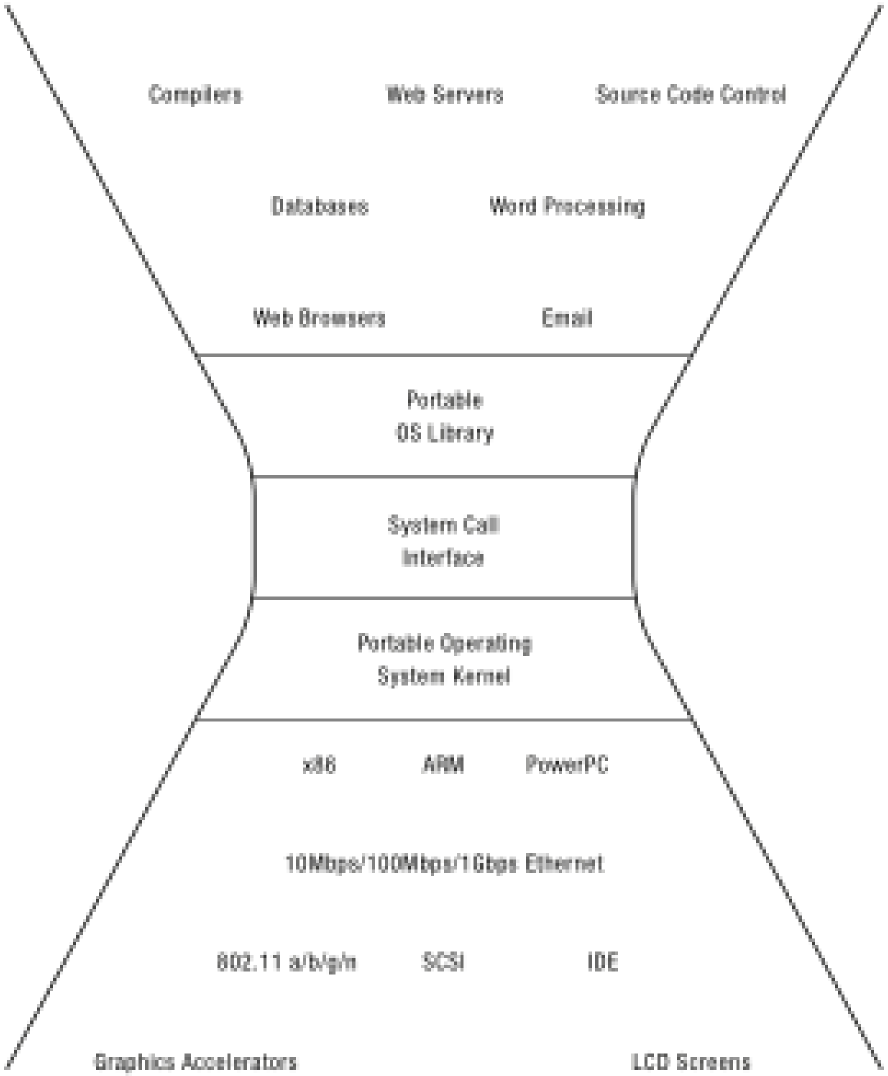
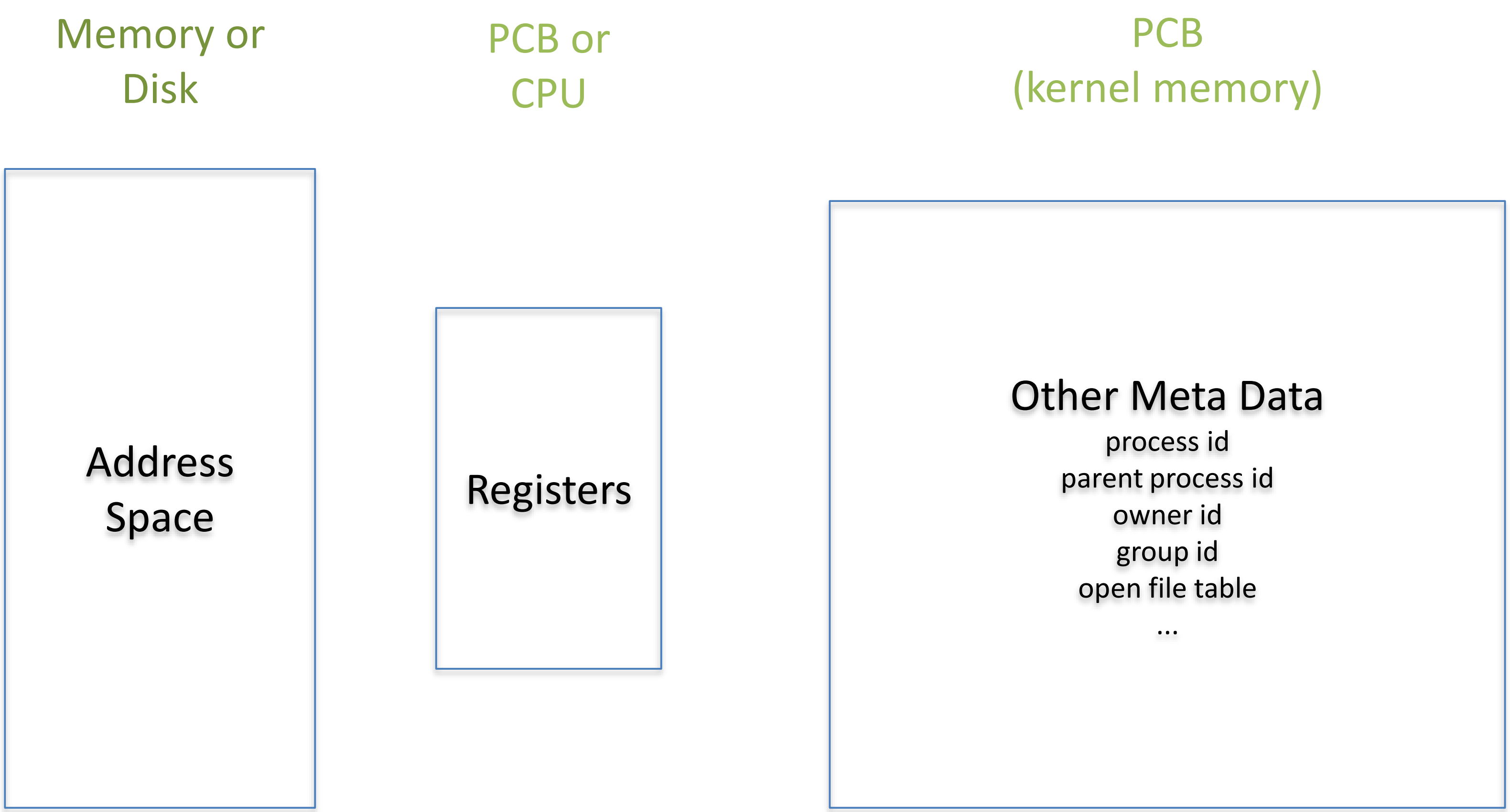
Module 3
The Programming Interface

Summary Picture 1



Summary Picture 2



N.B. We're assuming one thread per process at this point.

Module Main Points

- Creating and managing processes
  - fork, exec, wait
- Performing I/O
  - open, read, write, close
- Communicating between processes
  - pipe, dup, select, connect
- Example: implementing a shell

Shell

- A shell is a job control system
  - Allows programmer to create and manage a set of programs to do some task
  - Windows, MacOS, Linux all have shells
  - (The desktop is also a job control system)

- Example: to compile a C program

  cc –c sourcefile1.c *# compile but don't link*

  cc –c sourcefile2.c

  ln –o program sourcefile1.o sourcefile2.o *# link*

Question

- The shell runs at user-level. Can user level code create a new process?

- What system calls does the shell make to run each of the programs?
  - Ex: cc, ln

- *(How does the shell find the cc and ln executable files?)*

Windows `CreateProcess`

- System call to create a new process to run a program
  - Create and initialize the process control block (PCB) in the kernel
  - Create and initialize a new address space
  - Load the program into the address space
  - Copy arguments into memory in the address space
  - Initialize the hardware context to start execution at ``start''
  - Inform the scheduler that the new process is ready to run

Windows CreateProcess API (simplified)

```
if (!CreateProcess(
    NULL,           // No module name (use command line)
    argv[1],        // Command line
    NULL,           // Process handle not inheritable
    NULL,           // Thread handle not inheritable
    FALSE,          // Set handle inheritance to FALSE
    0,              // No creation flags
    NULL,           // Use parent's environment block
    NULL,           // Use parent's starting directory
    &si,            // Pointer to STARTUPINFO structure
    &pi )           // Pointer to PROCESS_INFORMATION structure
)
```

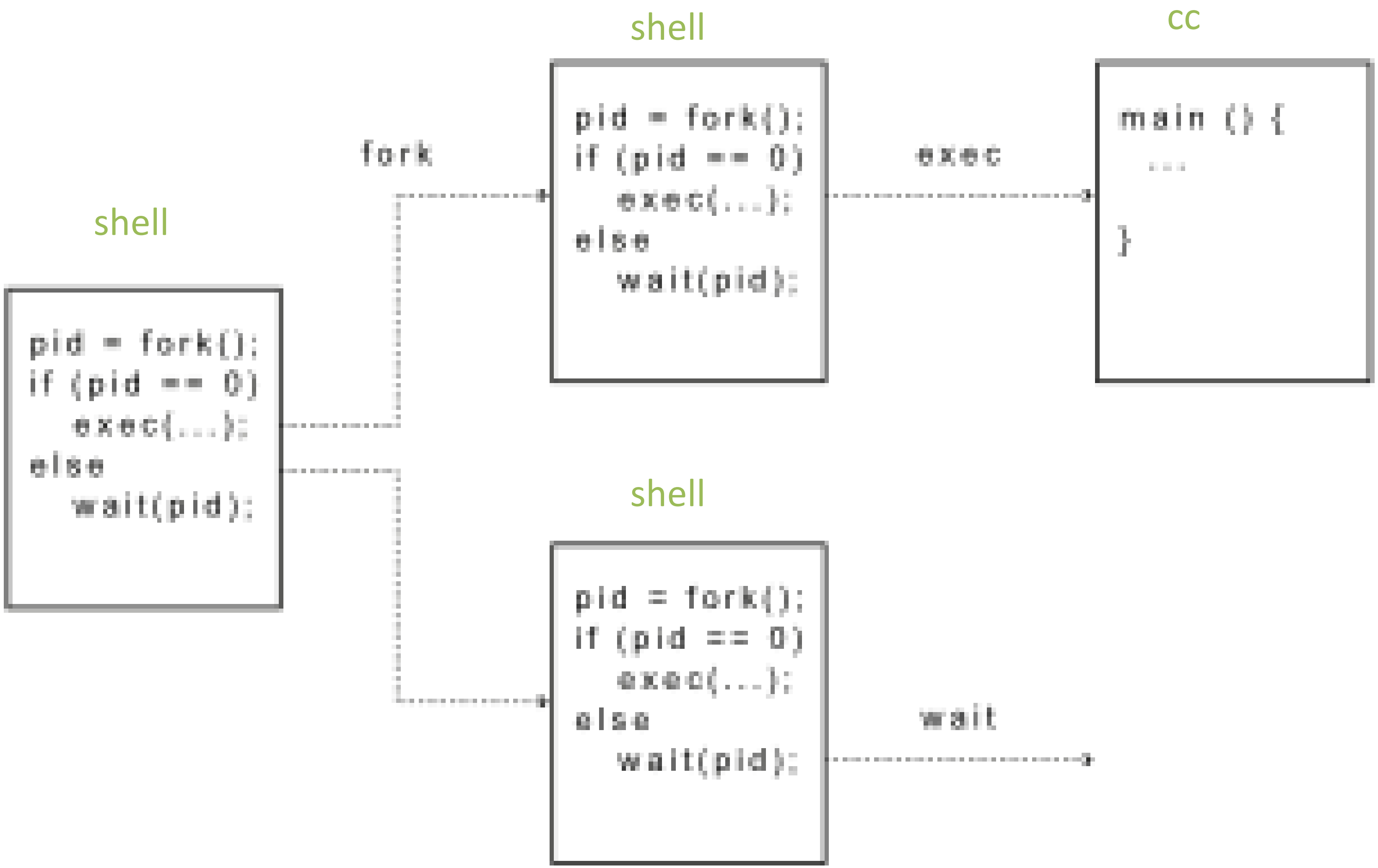
UNIX Process Management

- `fork` – system call to create a copy of the current process, and start it running
  - No arguments!
- `exec` – system call to change the program being run by the current process
  - What are the arguments?

UNIX Process Management

- `wait` – system call to wait for a process to finish
  - Arguments?
- `signal` – system call to send a notification (event) to another process
  - Arguments?

UNIX Process Management



Question: What does this code print?

```
int child_pid = fork();
if (child_pid == 0) {          // I'm the child process
    printf("I am process #%d\n", getpid());
    return 0;
} else {                       // I'm the parent process
    printf("I am parent of process #%d\n", child_pid);
    return 0;
}
```

Question: What is wrong with this code?

Questions

- Can UNIX fork() return an error? Why?
- Can UNIX exec() return an error? Why?
- Can UNIX wait() ever return immediately? Why?

Implementing UNIX **fork**

Steps to implement UNIX fork

- Create and initialize the process control block (PCB) in the kernel
  - Initialize using what data?
- *Inherit the execution context of the parent (e.g., any open files)*
- Create a new address space
- Initialize the address space with a copy of the entire contents of the address space of the parent
- Inform the scheduler that the new process is ready to run

Implementing UNIX **exec**

- Steps to implement UNIX fork
  - Load the program into the current address space
  - Copy arguments into memory in the address space
  - Initialize the hardware context to start execution at ``start‘’ (the “entry point”)

Topic 2: UNIX I/O

- Uniformity
  - All operations on all files, devices use the same set of system calls: open, close, read, write
    - Files (file systems), devices, sockets, pipes
- Open before use
  - Open returns a handle (file descriptor) for use in later calls on the file
    - Open files are part of process meta-data (in PCB)
  - Why?

UNIX I/O

- Byte-oriented
  - read/write byte buffer
  - Example alternative: read/write line of text
- Kernel-buffered read/write
  - kernel may read more bytes than asked for
  - kernel may delay writing bytes to device
- Explicit close
  - To garbage collect the open file descriptor

Aside: (UNIX) Open Files

- A file handle is an integer
  - An index into the open file table
- There file handles are special:
  - 0: stdin
  - 1: stdout
  - 2: stderr
- We'll talk about how they're initialized in a bit...

UNIX File System Interface

- UNIX file open is a Swiss Army knife:
  - Open the file, return file descriptor
  - Options:
    - if file doesn’t exist, return an error
    - If file doesn’t exist, create file and open it
    - If file does exist, return an error
    - If file does exist, open file
    - If file exists but isn’t empty, nix it then open
    - If file exists but isn’t empty, return an error
    - …

Interface Design Question

- Why not separate syscalls for open/create/exists?

```
if (!exists(name))
    create(name);   // can create fail?
fd = open(name);   // does the file exist?
```

Implementing a Shell

```
char *prog, **args;
int child_pid;

// Read and parse the input a line at a time
while (readAndParseCmdLine(&prog, &args)) {
  child_pid = fork();     // create a child process
   if (child_pid == 0) {
     exec(prog, args);      // I'm the child process.  Run program
     // NOT REACHED
   } else {
     wait(child_pid);      // I'm the parent, wait for child
     return 0;
   }
}
```

Shell Input/Output Redirection

$ ./prog <inputFile

```
while (readAndParseCmdLine(&prog, &args)) {
  child_pid = fork();     // create a child process
   if (child_pid == 0) {
      --- open inputFile as file descriptor 0 (stdin) ---
     exec(prog, args);      // I'm the child process.  Run program
    // NOT REACHED
   } else {
     wait(child_pid);      // I'm the parent, wait for child
     return 0;
   }
}
```

Other Shell Operations

- *./prog >outfile*
- *./prog &*
- *./prog >>logfile*
- *./prog >outfile 2>&1*

Topic 3: Interprocess Communicaiton

- Suppose processes want to share information
  - producer-consumer
    - output of one process is input to another (running at the same time)
  - client-server
    - general message passing between two processes
  - file system
    - tends to be producer consumer, but no need for simultaneous execution

Producer-consumer Communiction

- UNIX pipes
  - gcc test.c 2>&1 | grep –i error
- What is a “pipe”
  - Where is it located?
- How is the producer connected to the pipe?
  - The consumer?