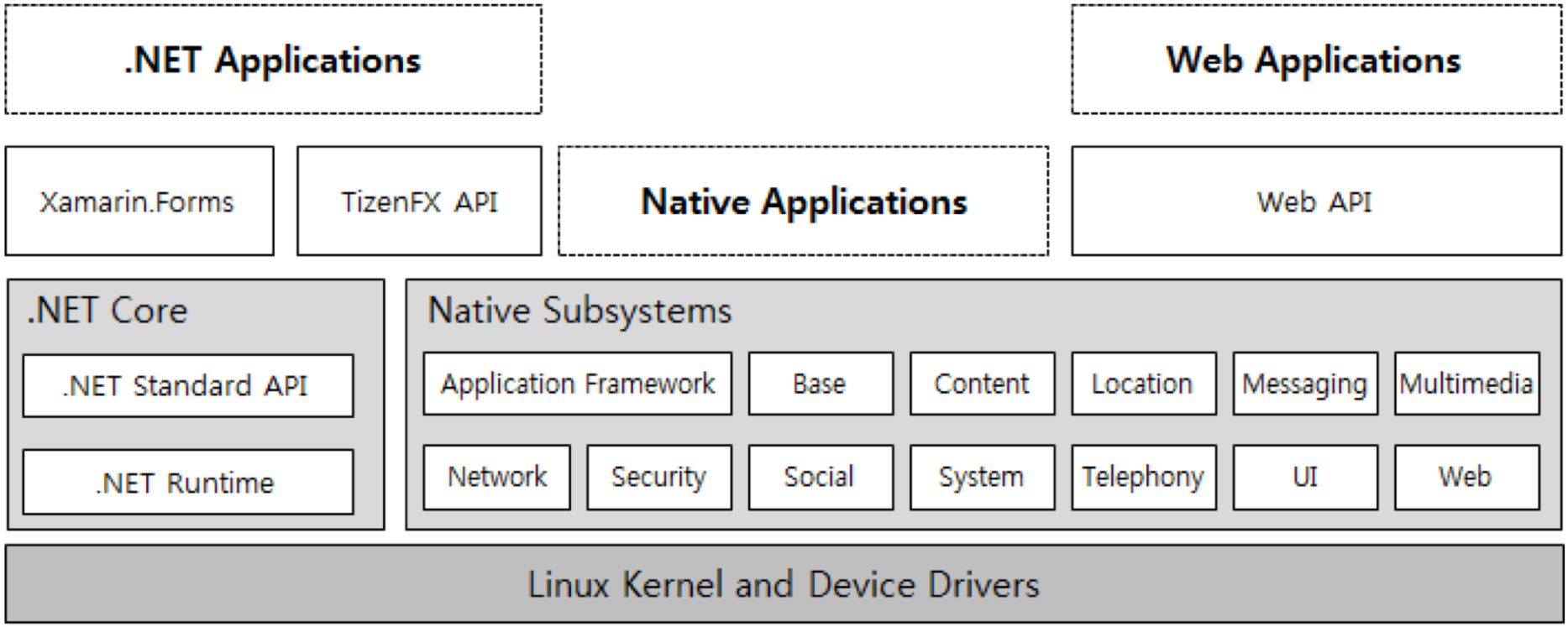
The Programming Interface

March 14, 2018
Byung-Gon Chun

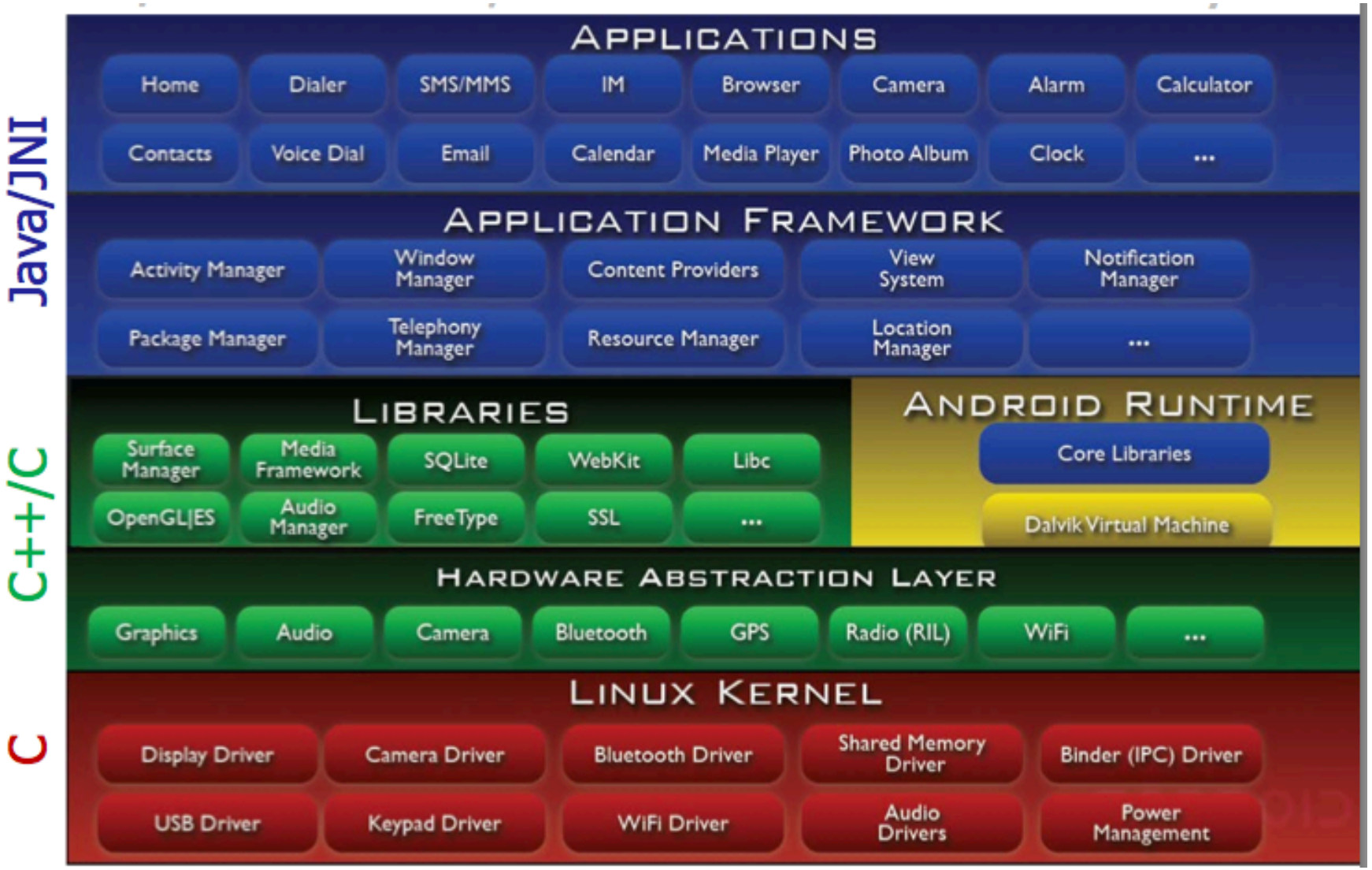
Acknowlegments. Slides and/or picture in the following are adapted from UW, Colombia, UC Berkeley class slides

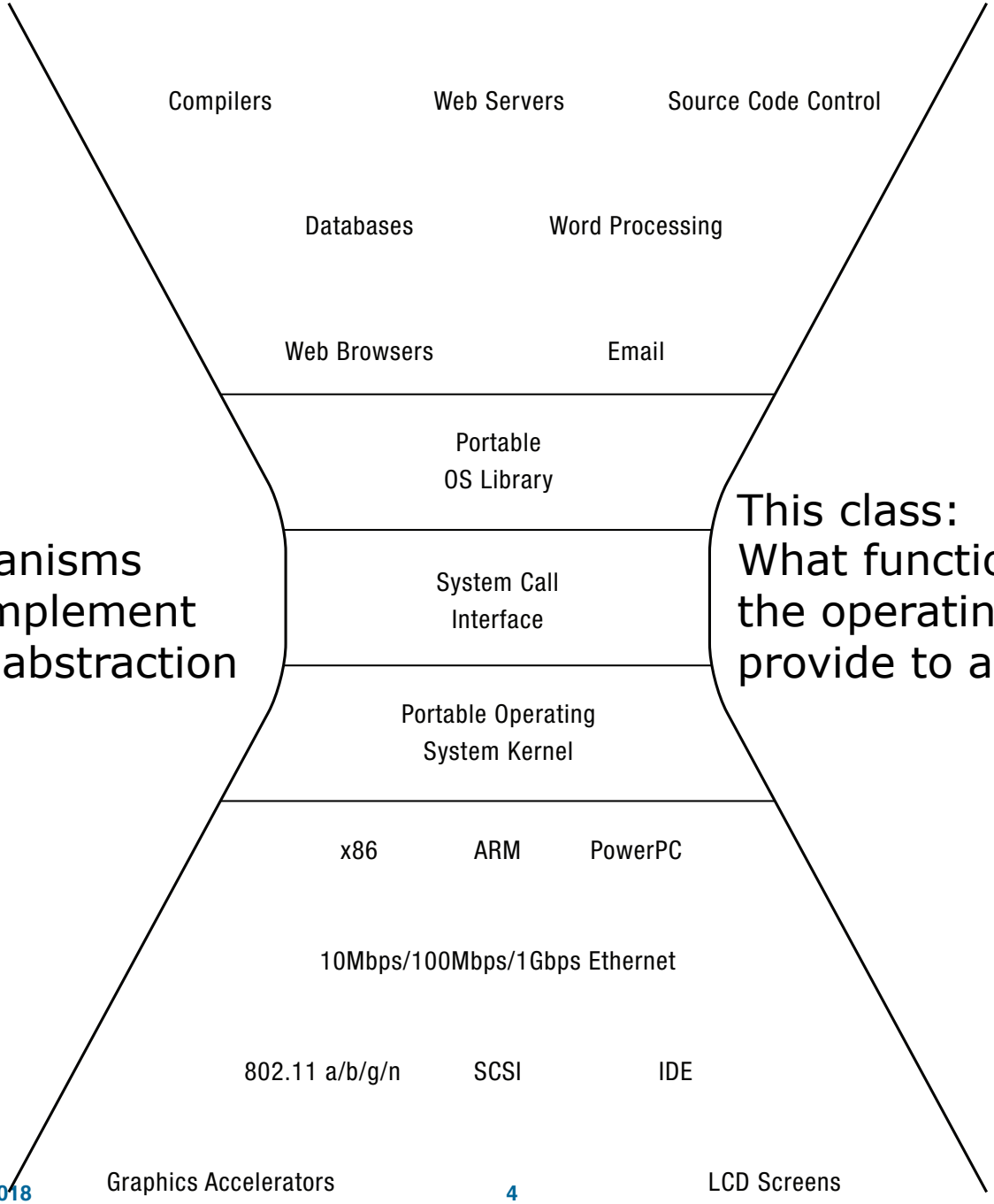
Tizen 3.0 architecture



(Ref. https://developer.tizen.org/development/training/overview)

Android architecture





Last class:
Some mechanisms needed to implement the process abstraction

This class:
What functionality does the operating system kernel provide to applications

Programming Interface

- Process management: creating and managing processes
  - fork, exec, wait
- Input/output: Performing I/O
  - open, read, write, close
- Thread management
- Memory management
- File systems and storage
- Communicating between processes
  - pipe, dup, select, connect
- Authentication and security
- Example: implementing a shell

Process Management

- In early batch processing systems, the kernel was in control.

  Users submitted jobs, and the operating system took it from there, instantiating the process when it was time to run the job.

Shell

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

- Example: to compile a C program

  cc –c sourcefile1.c

  cc –c sourcefile2.c

  ln –o program sourcefile1.o sourcefile2.o

Question

- If the shell runs at the user level, what system calls does it make to run each of the programs?
  - Ex: cc, ln

Windows CreateProcess

- System call to create a new process to run a program
  A simplified form:
  boolean CreateProcess(char *prog, char *args);

  - 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

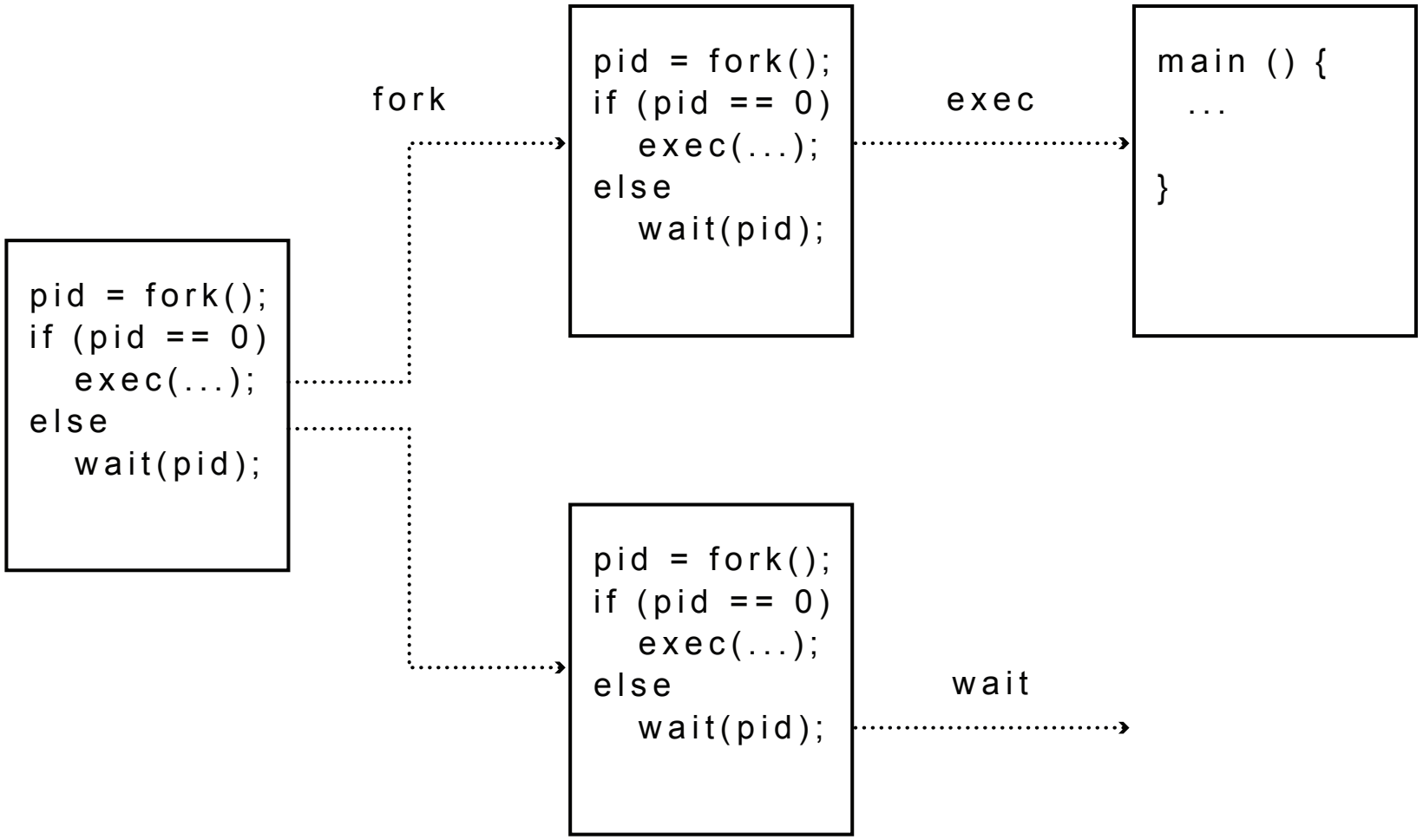
Windows CreateProcess API (simplified)

```
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

- UNIX fork – system call to create a copy of the current process, and start it running
  - No arguments!
- UNIX exec – system call to change the program being run by the current process
- UNIX wait – system call to wait for a process to finish
- UNIX signal – system call to send a notification to another process

UNIX Process Management



fork: Creating new processes

- **`int fork(void)`**
  - creates a new process (child process) that is identical to the calling process (parent process)
  - returns 0 to the child process
  - returns child's pid to the parent process

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

- `fork()` is interesting (and often confusing) because it is called once but returns twice

Understanding fork

Process n

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = m

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Child Process m

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = 0

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

`hello from parent` *Which one is first?* `hello from child`

Fork Example #1

- Both parent and child can continue forking

```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```

Fork Example #2

- Both parent and child can continue forking

```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
       printf("L1\n");
       if (fork() != 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```

Process creation

- The *fork* system call is used to create a new process.
  - Identical to parent except ...
  - execution state
  - process ID
  - parent process ID.
  - other data is either copied or made copy on write (like process address space).

- Copy on write allows data to be shared as long as it is not modified, but each task gets its own copy when one task tries to modify the data.

Implementing UNIX fork

Steps to implement UNIX fork

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

Implementing UNIX exec and wait

- Steps to implement UNIX exec
  - 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''

- wait
  - Pauses the parent until the child finishes, crashes, or is terminated
  - It is parametrized with the process ID of the child

UNIX I/O

- Uniformity
  - All operations on all files, devices use the same set of system calls: open, close, read, write
- Open before use
  - Open returns a handle (file descriptor) for use in later calls on t he file
- Byte-oriented
- Kernel-buffered read/write
- Explicit close
  - To garbage collect the open file descriptor

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?
```

```
// used in a setuid program

if (access("file", W_OK) != 0)
{
  exit(1);
}
fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

Victim

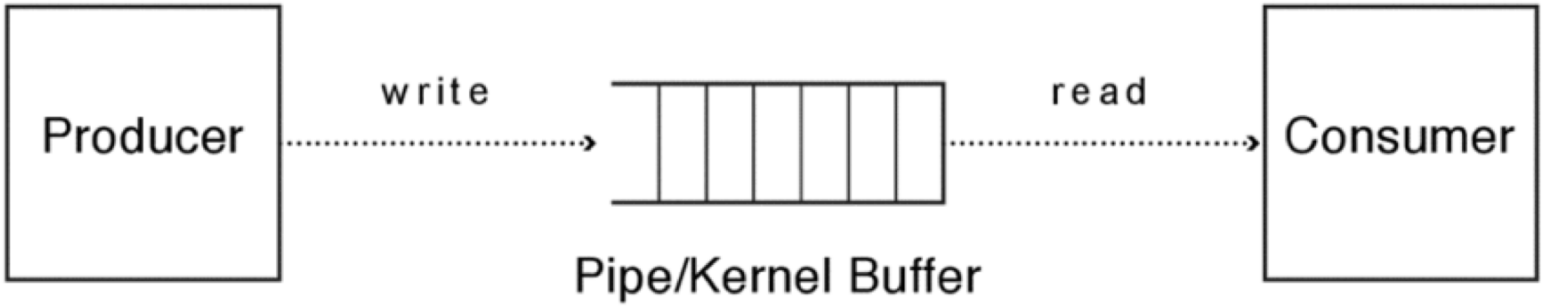
```
if (access("file", W_OK) != 0)
{
  exit(1);
}


fd = open("file", O_WRONLY);
// Actually writing over /etc/passwd
write(fd, buffer, sizeof(buffer));
```

Attacker

```
//
//
// After the access check
symlink("/etc/passwd", "file");
// Before the open, "file" points to the
password database
//
//
```

Interprocess communication

- Unix pipe : int pipe(int pipefd[2])



- Unix socket : int socket(int domain, int type, int protocol)

- Replace file descriptor : int dup2(int oldfd, int newfd)

- Wait for multiple file descriptors :
  int select(int nfds, fd_set *readfds, fd_set *writefds,
  fd_set *exceptfds, struct timeval *timeout)

Implementing a Shell

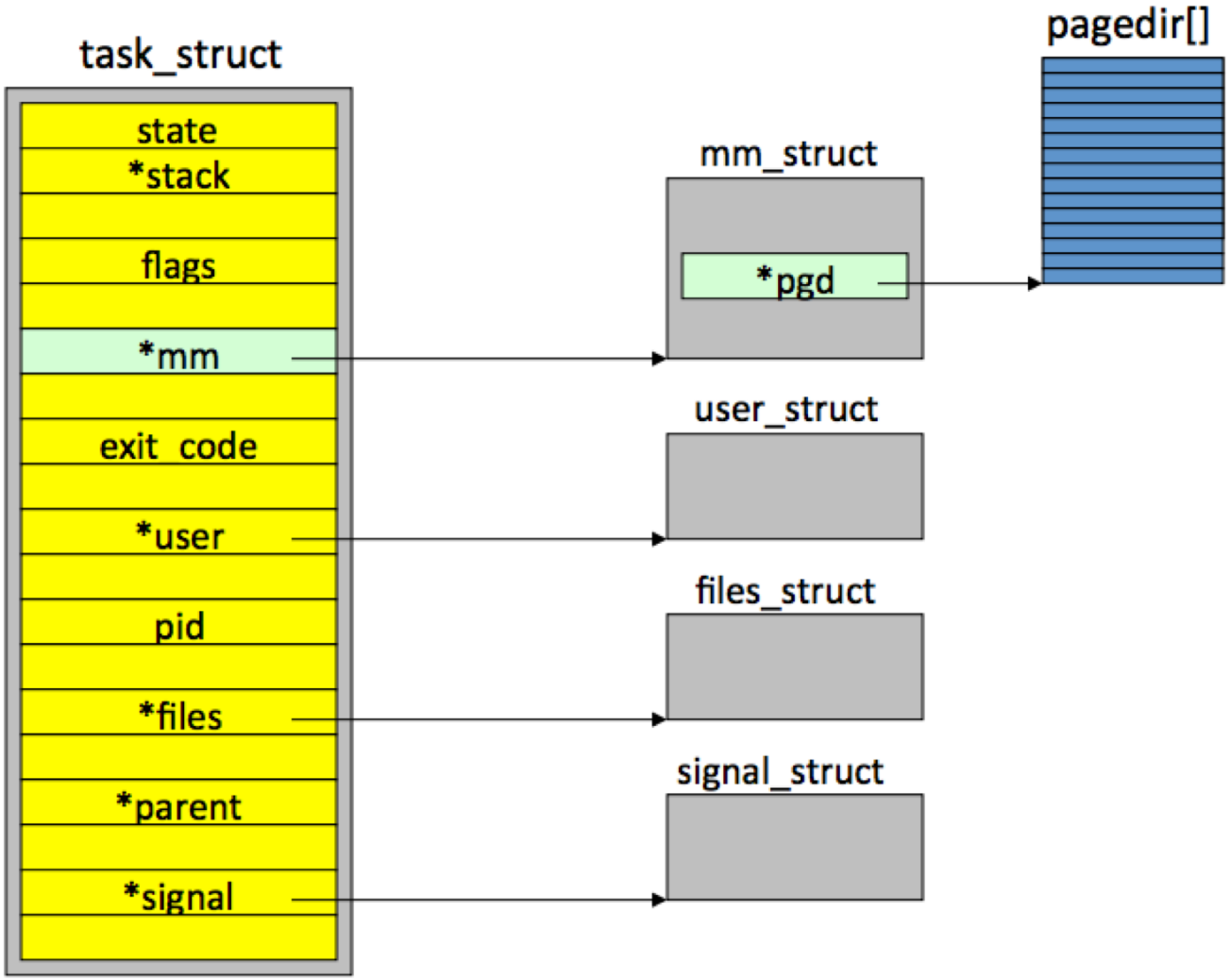
```
char *prog, **args;
int child_pid;

// 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;
    }
}
```

Linux: task_struct

Each process descriptor contains many fields

and some are pointers to other kernel structures

which may themselves include fields that point to structures



The task structure

- The task_struct is used to represent a task

- The task_struct has several sub-structures that it references
  - tty_struct – TTY associated with the process
  - fs_struct – current and root directories associated with the process
  - files_struct – file descriptors for the process
  - mm_struct – memory areas for the process
  - signal_struct – signal structures associated with the process
  - user_struct – per-user information (e.g., number of current processes)

Task relationships

- Several pointers exist between task_structs
  - parent – pointer to parent process
  - children – pointer to the linked list of my children
  - sibling – pointer to task of “next younger sibling” of the current task
    (linkage in my parent’s children list)

Linked List in Kernel

```
//include/linux/list.h

struct list_head {
        struct list_head *next, *prev;
};

struct my_data {
        ...
        struct list_head lh1;
        ...
        struct list_head lh2;
        ...
};

Many macros to get the payload, iterate, add, delete
list_entry(ptr, type, member)
list_for_each_entry(pos, head, member)
```