The Programming Interface

*Throughout the course we will use overheads that were adapted from those distributed from the textbook website. Slides are from the book authors, modified and selected by Jean Mayo, Shuai Wang and C-K Shene.

I'm afraid that the following syllogism may be used by some in the future.

Turing believes machines think
Turing lies with men
Therefore machines do not think

Alan Turing

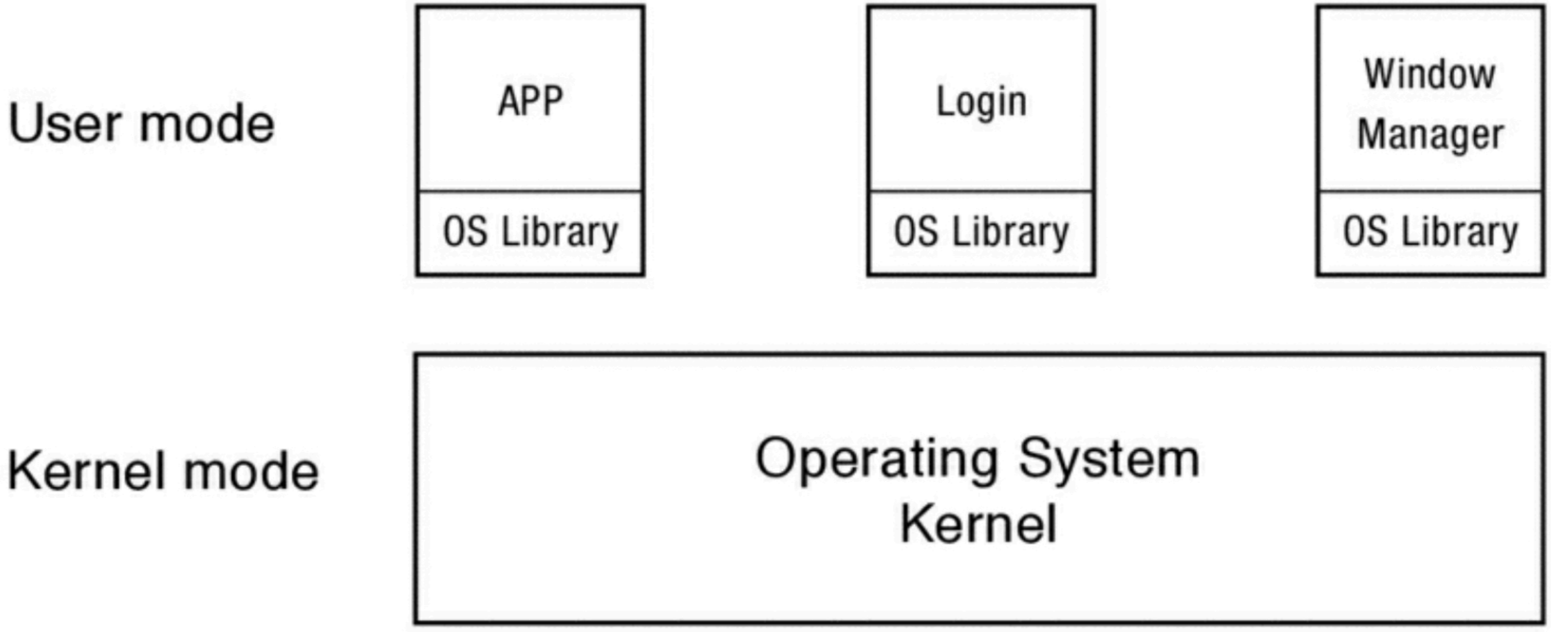
Spring 2019

What Functions an OS Can Provide Applications?

- **Process Management**: fork, wait, exec
- **Performing I/O**: open, read, write, close
- **Thread Management**: create, terminate, join, etc.
- **Memory Management**: Can a process ask for more (or less) memory space? Can it share the same physical memory region with other processes?
- **File System and Storage**: How does the user name and organize their data? How does a process store the user's data persistently?
- **Networking and Distributed Systems**: How do processes communicate with processes on other computer?
- **Graphics and Window Management**
- **Authentication and Security.**

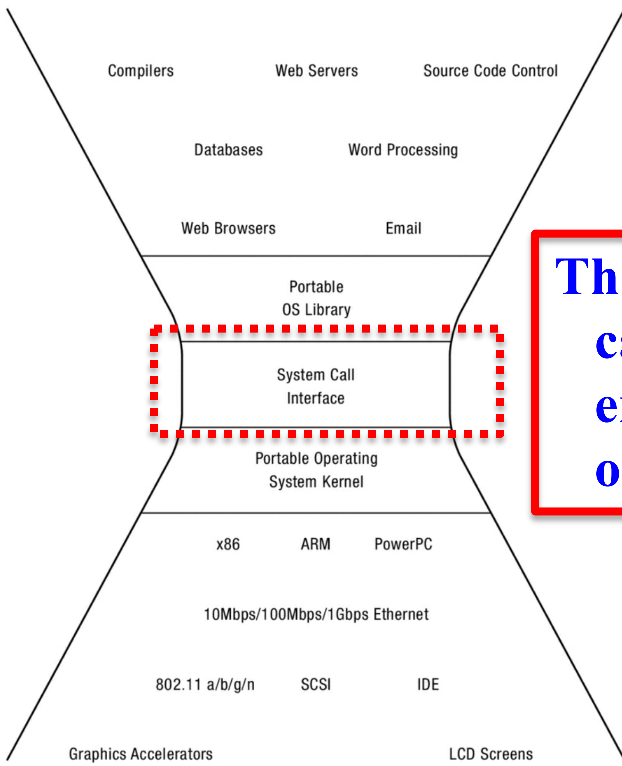
Operating System Functionality

- A functional interface for process management and I/O can be described with a dozen system calls, and the rest of the system call interface with another dozen.



Tradeoff

- As long as the OS provides an interface, where each function is implemented is up to the OS, based on a tradeoff between **flexibility**, **reliability**, **performance**, and **safely**.



The kernel system call interface can be seen as a "thin waist." enabling independent evolution of applications and hardware.

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**
- **Each command issued create a process to execute it.**
- **Example: to compile a C program**
  - `cc -c file1.c`
  - `cc -c file2.c`
  - `ln -o program file1.o file2.o`
- **Three processes are created, one after the other: for the 1st `cc`, the 2nd `cc`, and the `ln`.**

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

- 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

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
- **Inherit** the execution context of the parent (e.g., any open files)
- Inform the scheduler that the new process is ready to run

`fork()` Return Values

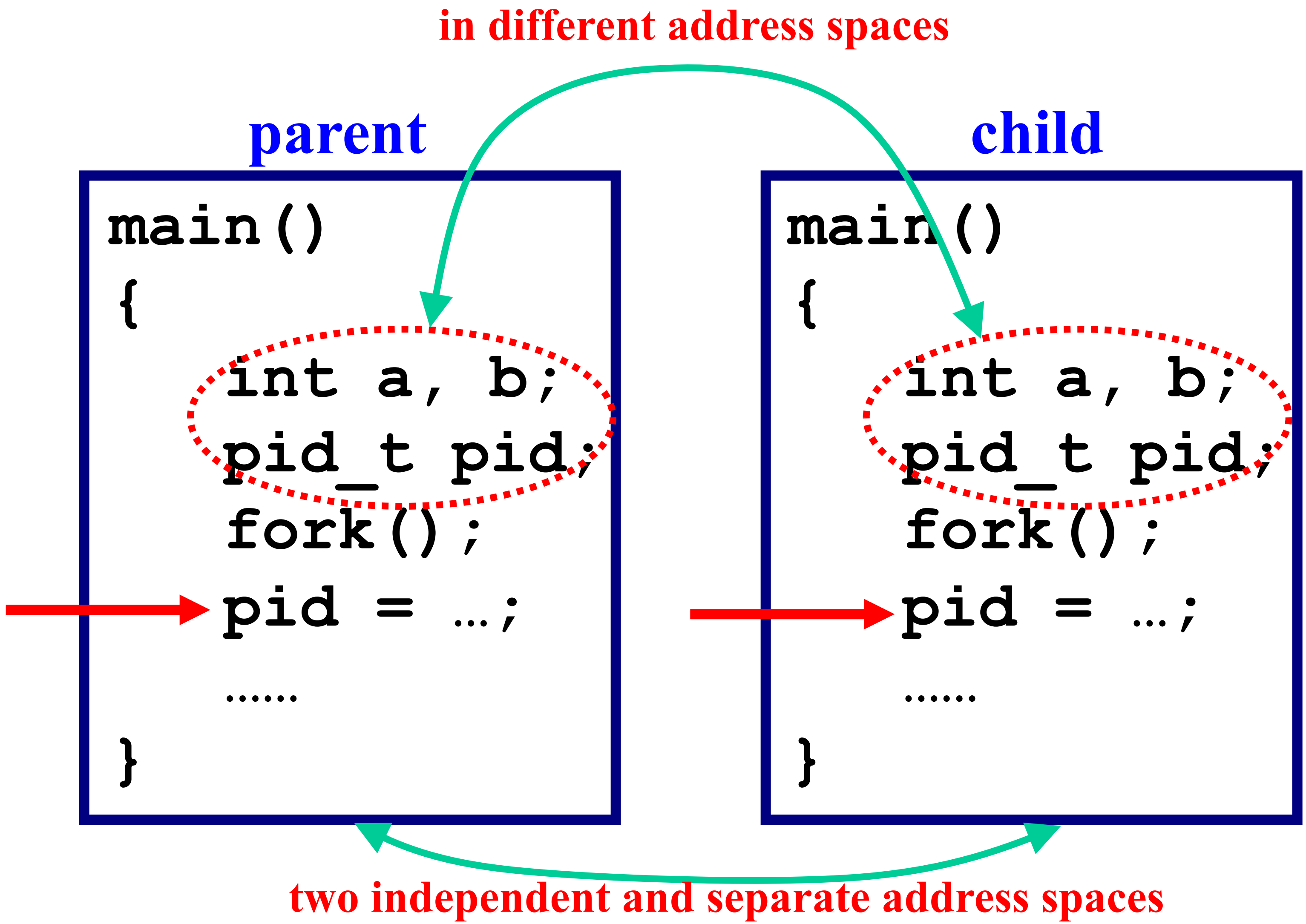
- **A negative value means the creation of a child process was unsuccessful.**
- **A zero means the process is a child.**
- **Otherwise, `fork()` returns the process ID of the child process. The ID is of type `pid_t`.**
- **Function `getpid()` returns the process ID of the caller.**
- **Function `getppid()` returns the parent's process ID. If the calling process has no parent, `getppid()` returns `1`.**

Before Executing `fork()`

parent

```
main()
{
    int a, b;
    pid_t pid
    fork();
    pid = …;
    ……
}
```

After Executing `fork()`



fork() : A Typical Use

```
main(void)
{
  pid_t pid;

  if ((pid=fork()) < 0)
    printf("Oops!");
  else if (pid == 0)
    child();
  else // pid > 0
    parent();
}
```

```
void child(void)
{
  int i;
  for (i=1; i<=10; i++)
    printf(" Child:%d\n", i);
  printf("Child done\n");
}

void parent(void)
{
  int i;
  for (i=1; i<=10; i++)
    printf("Parent:%d\n", i);
  printf("Parent done\n");
}
```

we use `printfs` here to save space.

Before the Execution of `fork()`

parent

```
main(void)          pid = ?
{
  pid = fork();
  if (pid == 0)
    child();
  else
    parent();
}

void child(void)
{ ...... }

void parent(void)
{ ...... }
```

After the Execution of `fork()` 1/2

parent

pid=123

```
main(void)
{
  pid = fork();
  if (pid == 0)
    child();
  else
    parent();
}

void child(void)
{ ...... }

void parent(void)
{ ...... }
```

child

pid = 0

```
main(void)
{
  pid = fork();
  if (pid == 0)
    child();
  else
    parent();
}

void child(void)
{ ...... }

void parent(void)
{ ...... }
```

in two different address spaces

After the Execution of `fork()` 2/2

parent

pid=123

```
main(void)
{
  pid = fork();
  if (pid == 0)
    child();
  else
    parent();
}

void child(void)
{ ...... }

void parent(void)
{ ...... }
```

child

pid = 0

```
main(void)
{
  pid = fork();
  if (pid == 0)
    child();
  else
    parent();
}

void child(void)
{ ...... }

void parent(void)
{ ...... }
```

Implementing UNIX `exec`

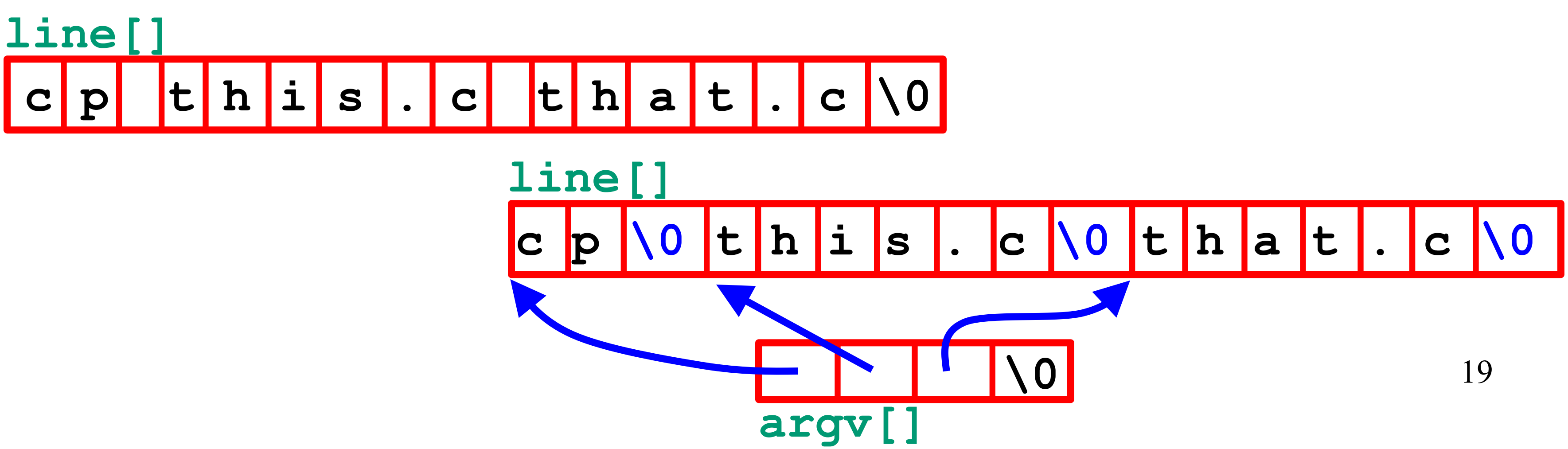
- **Steps to implement UNIX `exec`**
  - **Load the program into the current address space**
  - **Copy arguments into memory in the address space**
  - **Initialize the hardware context to start execution at ``start''**

The `exec()` System Calls

- A newly created process may run a different program rather than that of the parent.
- This is done using the `exec` system calls. We will only discuss `execvp()`:
- `int execvp(char *file, char *argv[]);`
  - `file` is a `char` array that contains the name of an executable file. Depending on your system settings, you may need the `./` prefix for files in the current directory.
  - `argv[]` is the argument passed to your main program
  - `argv[0]` is a pointer to a string that contains the program name
  - `argv[1]`, `argv[2]`, … are pointers to strings that contain the arguments

A Mini-Shell: 1/3

```
void parse(char *line, char **argv)
{
    while (*line != '\0') { // not EOLN
        while (*line == ' ' || *line == '\t' || *line == '\n')
            *line++ = '\0';    // replace white spaces with 0
        *argv++ = line;        // save the argument position
        while (*line != '\0' && *line ! =' '
                             && *line!='\t' && *line != '\n')
            line++;            // skip the argument until ...
    }
    *argv = '\0';              // mark the end of argument list
}
```



A Mini-Shell: 2/3

```
void execute(char **argv)
{
   pid_t pid;
   int status;
   if ((pid = fork()) < 0) { // fork a child process
      printf("*** ERROR: forking child process failed\n");
      exit(1);
   }
   else if (pid == 0) {      // for the child process:
      if (execvp(*argv, argv) < 0) { // execute the command
         printf("*** ERROR: exec failed\n");
         exit(1);
      }
   }
   else {                    // for the parent:
      while (wait(&status) != pid) // wait for completion
         ;
   }
}
```

A Mini-Shell: 3/3

```
void main(void)
{
   char line[1024];   // the input line
   char *argv[64];    // the command line argument

   while (1) {        // repeat until done ....
      printf("Shell -> "); // display a prompt
      gets(line);     // read in the command line
      printf("\n");
      parse(line, argv);   // parse the line
      if (strcmp(argv[0], "exit") == 0) // is it an "exit"?
         exit(0);     // exit if it is
      execute(argv); // otherwise, execute the command
   }
}
```

Don't forget that `gets()` is risky! Use `fgets()` instead.

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 the 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
    - …

The End