CSC / CPE 357

Systems Programming

Chapter 8 in Advanced Programming
in the UNIX Environment

Process Control

The UNIX system handles **process control**, including:

- Creation of new processes
- Program execution
- Process termination

Process Identifiers

- Every process has a unique **process ID** (PID)
  - Unique, non-negative integer
  - May be reused; terminated process IDs are candidates for reuse

- Special processes (implementation specific).
  - Process ID 0 is usually the scheduler process
  - Process ID 1 is usually the `init` process, responsible for initializing a system after boot
  - Process ID 2 is the `pagedaemon`; responsible for supporting the paging of the virtual memory system

- `ps` command

Library Functions

```
#include <unistd.h>

pid_t getpid(void);
```

Returns: process ID of calling process

```
pid_t getppid(void);
```

Returns: parent process ID of calling process

Process Group ID

- Every process is member of a unique process group, identified by its **process group ID**.

- When the process is created, it becomes a member of the process group of its parent.

- `int setpgid(pid_t pid, pid_t pgid);`
    - Sets the process group id of the process specified by `pid` to `pgid`

- `pid_t getpgid(pid_t pid);`

fork Function

An existing process can create a new process by calling the `fork` function.

```
#include <unistd.h>

pid_t fork(void);
```

Returns: 0 in child, process ID of child in parent, –1 on error

- New process created by `fork()` is referred to as the *child* process.
- Child and the parent processes continue executing with the instruction that follows the call to fork
  - Return value from `fork()` is 0 in child process
  - In the parent, `fork()` returns the child's process ID
- Child process gets a *copy* of the parent's data space, heap, and stack.

Example: `fork.c`

- `printf` before the `fork` is called once, but the line remains in the buffer when fork is called and buffer is copied

File Sharing

- When using fork, all file descriptors (stdin, stdout, and any others) that are open in the parent are duplicated in the child

- If both parent and child write to the same open file descriptor, without any synchronization (ie. parent waits for the child) output will be interleaved

- Typically one of these approaches is used:
  - Parent waits for the child to complete
  - After `fork()`, parent and child each close the file descriptors they don't need

Uses for `fork()`

1. Network servers that handle client requests
    a. Parent waits for a request from a client
    b. Parent calls fork, child process handle the request
    c. Parent is free to wait for next request

2. Shells
    a. Process need to execute a different program
    b. Child uses `exec()`

Normal Process Termination

- `return` from `main` function (equivalent to calling `exit()`)

- Call the `exit` function
  - Calls exit handlers registered by `atexit`
  - Closes all Standard I/O streams

- Calling the `_exit` or `_Exit` function
  - Terminate without running exit handlers or signal handlers
  - On UNIX systems, `_Exit` and `_exit` are synonymous and do not flush standard I/O streams

Abnormal Process Termination

- Calling `abort` (generates the `SIGABRT` signal)

- Process receives a signal indicating a critical, unrecoverable error
  - Referencing a memory location not within its address space
  - Attempt to divide by 0

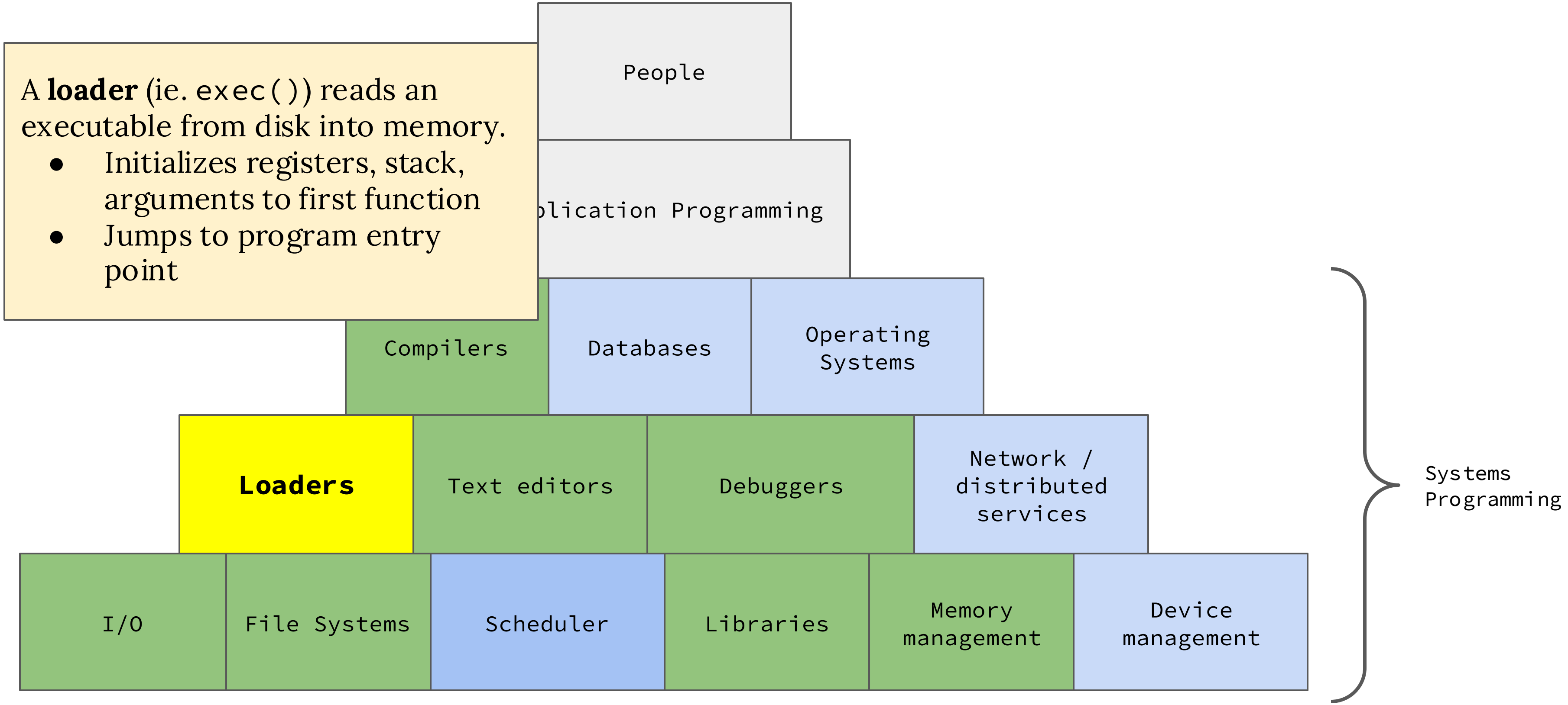
wait / waitpid system calls

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

pid_t wait(int *status);

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

Wait for state changes in a child of the calling process, and obtain information about the child whose state has changed

A **loader** (ie. `exec()`) reads an executable from disk into memory.

- Initializes registers, stack, arguments to first function
- Jumps to program entry point



exec Functions

- The fork function can be used to create a new process (the child) that then causes another program to be executed by calling one of the exec functions.

- When a process calls one of the exec functions, the program is completely *replaced* by the new program.
    - Process ID does not change

exec Functions

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );

int execv(const char *pathname, char *const argv[]);

int execle(const char *pathname, const char *arg0, ...
           /* (char *)0, char *const envp[] */ );

int execve(const char *pathname, char *const argv[], char *const envp[]);

int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );

int execvp(const char *filename, char *const argv[]);

int fexecve(int fd, char *const argv[], char *const envp[]);
```

All seven return: –1 on error, no return on success

exec Functions

Decoding function names:

- `p` - the function takes a filename argument and uses the `PATH` environment variable to find the executable file
- `l` - the function takes a list of arguments
- `v` - takes an `argv[]` vector (mutually exclusive with `l`)
- `e` - the function takes an `envp[]` array instead of using the current environment

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );

int execv(const char *pathname, char *const argv[]);

int execle(const char *pathname, const char *arg0, ...
           /* (char *)0, char *const envp[] */ );

int execve(const char *pathname, char *const argv[], char *const envp[]);

int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );

int execvp(const char *filename, char *const argv[]);

int fexecve(int fd, char *const argv[], char *const envp[]);
```

`fexecve`

```
int fexecve(int fd, char *const argv[], char *const envp[]);
```

- `fexecve()` performs the same task as `execve()`, with the difference that the file to be executed is specified via a file descriptor `fd` rather than via a path name

- File descriptor `fd` must be opened read-only
  - Caller must have permission to execute the file that it refers to.

`wait`

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);
```

`waitpid`

The `waitpid()` system call suspends execution of the calling process until a child specified by `pid` argument has changed state (terminated, by default)

The value of `pid` can be:

`< -1`	Wait for any child process whose process group ID is equal to the absolute value of pid.
`-1`	Wait for any child process
`0`	Wait for any child process whose process group ID is equal to that of the calling process
`> 0`	Wait for the child whose process ID is equal to the value of pid

`waitpid` options

By default, `waitpid()` waits only for terminated children, but this behavior is modifiable via the options argument:

`WNOHANG`	Return immediately if no child has exited
`WUNTRACED`	Return if child has stopped
`WCONTINUED`	Return if a stopped child has been stopped and resumed by delivery of `SIGCONT`

`wait` vs `waitpid`

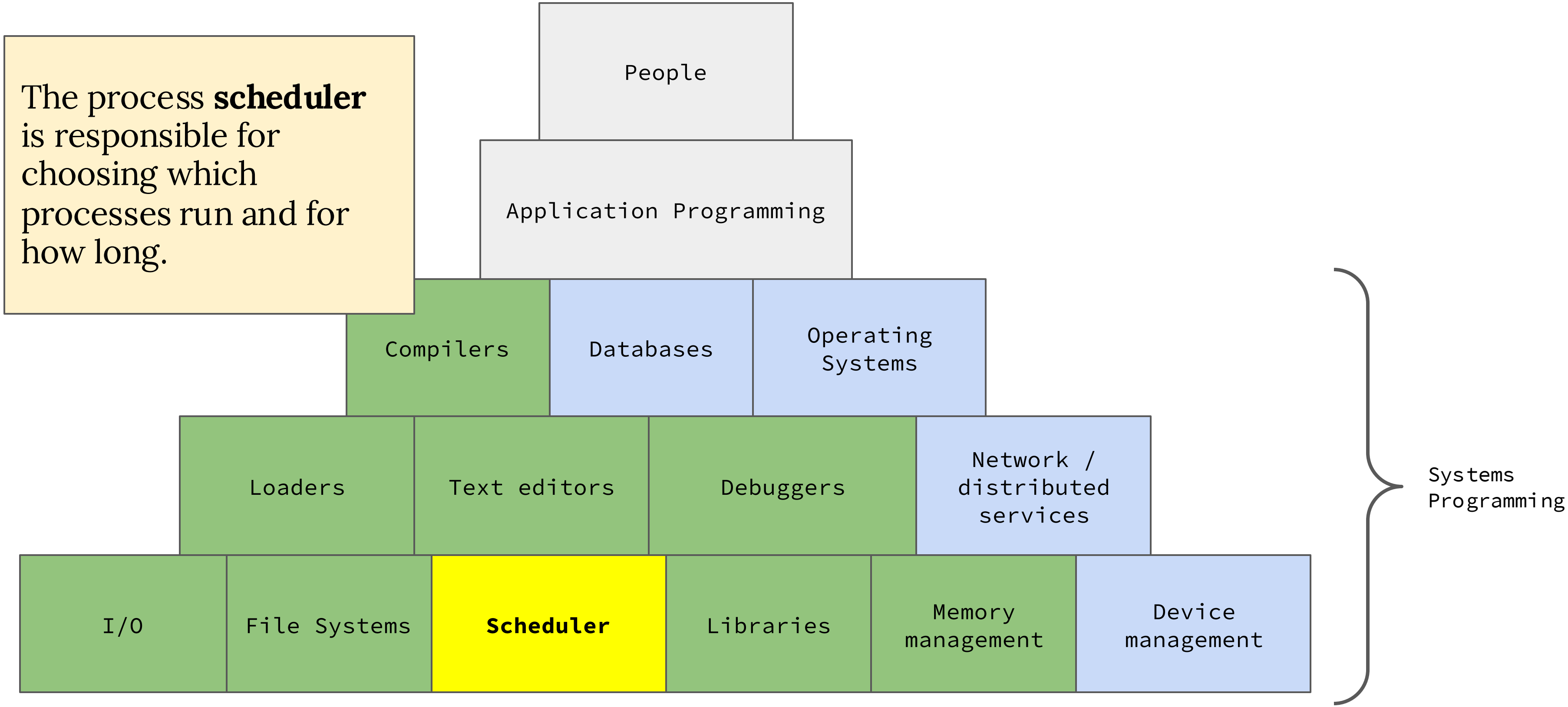
The `waitpid` function provides three features that aren't provided by the `wait` function.

1. The `waitpid` function lets us wait for one particular process, whereas the wait function returns the status of *any* terminated child.

2. The `waitpid` function provides a nonblocking version of `wait`.

3. The `waitpid` function provides support for job control with the `WUNTRACED` and `WCONTINUED` options.

The process **scheduler** is responsible for choosing which processes run and for how long.



Race Conditions

A race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run.

Process Scheduling

- A process may choose to run with a different priority by adjusting its **nice value**

- Nice values range from `0` to `(2*NZERO)-1`
    - `NZERO` is defined as the default nice value of a system (20 in Linux)

- Higher nice value indicates lower scheduling priority

nice Function

```
#include <unistd.h>

int nice(int incr);
```

Returns: new nice value − NZERO if OK, −1 on error

- A process can retrieve and change its nice value with the `nice` function
- Can't affect the nice value of any other process
- The `incr` argument is added to the nice value of the calling process.

getpriority / setpriority Function

```
#include <sys/resource.h>

int getpriority(int which, id_t who);
```

Returns: nice value between −NZERO and NZERO−1 if OK, −1 on error

- Alternative function `setpriority` can be used to change nice value
- Also permits retrieving/setting nice value for a group of related processes
- `which` argument can be:
  - `PRIO_PROCESS` to indicate a process,
  - `PRIO_PGRP` to indicate a process group, or
  - `PRIO_USER` to indicate a user ID

Process Timing

Any process can call the `times` function to obtain timing information for itself and any terminated children

```
#include <sys/times.h>

clock_t times(struct tms *buf);
```

Returns: elapsed wall clock time in clock ticks if OK, –1 on error

```
struct tms {
  clock_t  tms_utime;  /* user CPU time */
  clock_t  tms_stime;  /* system CPU time */
  clock_t  tms_cutime; /* user CPU time, terminated children */
  clock_t  tms_cstime; /* system CPU time, terminated children */
};
```

Signals

- **Signals** are software interrupts

- Provide a way of handling asynchronous events. For example:
    - User typing Ctrl+C to interrupt program execution
    - Unexpected termination of a process
    - Job control (stop/resume)

- Most nontrivial application programs need to deal with signals

Signal Names

- Every signal has a name, beginning with: `SIG`

- A few examples:
  - `SIGQUIT - terminal quit character`
  - `SIGSEGV - invalid memory reference`
  - `SIGILL - illegal instruction`
  - `SIGPWR - power fail/restart`

Signal Handlers

The `signal()` function allows us to register custom signal handlers.

```
sighandler_t signal(int signum, sighandler_t handler);
```

`sighandler_t` is defined as a function that accepts a single int argument and has a return type of void