

Process

Adapted from slides by Ian Hartwig

Processes

■ What is a *program*?

- A bunch of data and instructions stored in an executable binary file
- Written according to a specification that tells users what it is supposed to do
- Stateless since binary file is static

Processes

- Definition: A *process* is an instance of a running program.
- Process provides each program with two key abstractions:
 - Logical control flow
 - Each program seems to have exclusive use of the CPU
 - Private virtual address space
 - Each program seems to have exclusive use of main memory
 - Gives the running program a *state*
- How are these Illusions maintained?
 - Process executions interleaved (multitasking) or run on separate cores
 - Address spaces managed by virtual memory system
 - Just know that this exists for now; we'll talk about it soon

Processes

■ Four basic States

- Running
 - Executing instructions on the CPU
 - Number bounded by number of CPU cores
- Runnable
 - Waiting to be running
- Blocked
 - Waiting for an event, maybe input from STDIN
 - Not runnable
- Zombie
 - Terminated, not yet reaped

Processes

■ Four basic process control function families:

- `fork()`
- `exec()`
 - And other variants such as `execve()`
- `exit()`
- `wait()`
 - And variants like `waitpid()`

■ Standard on all UNIX-based systems

■ Don't be confused:

Fork(), Exit(), Wait() are all wrappers provided by CS:APP

Processes

■ `int fork(void)`

- creates a new process (child process) that is identical to the calling process (parent process)
- OS creates an exact duplicate of parent's state:
 - Virtual address space (memory), including heap and stack
 - Registers, except for the return value (%eax/%rax)
 - File descriptors but files are shared
- **Result → Equal but separate state**
- Fork is interesting (and often confusing) because it is called *once* but returns *twice*

Processes

■ `int fork(void)`

- returns 0 to the child process
- returns child's `pid` (process id) to the parent process
- Usually used like:

```
pid_t pid = fork();

if (pid == 0) {
    // pid is 0 so we can detect child
    printf("hello from child\n");
}

else {
    // pid = child's assigned pid
    printf("hello from parent\n");
}
```

Processes

■ `int exec()`

- Replaces the current process's state and context
 - But keeps PID, open files, and signal context
- Provides a way to load and run **another** program
 - Replaces the current running memory image with that of new program
 - Set up stack with arguments and environment variables
 - Start execution at the entry point
- Never returns on successful execution
- The newly loaded program's perspective: as if the previous program has not been run before
- More useful variant is `int execve()`
- More information? `man 3 exec`

Processes

■ `void exit(int status)`

- Normally return with status 0 (other numbers indicate an error)
- Terminates the current process
- OS frees resources such as heap memory and open file descriptors and so on...
- Reduce to a zombie state
 - Must wait to be reaped by the parent process (or the init process if the parent died)
 - Signal is sent to the parent process notifying of death
 - Reaper can inspect the exit status

Processes

■ `int wait(int *child_status)`

- suspends current process until one of its children terminates
- return value is the pid of the child process that terminated
 - When wait returns a pid > 0, child process has been reaped
 - All child resources freed
- if `child_status != NULL`, then the object it points to will be set to a status indicating why the child process terminated
- More useful variant is `int waitpid()`
- For details: `man 2 wait`

Process Examples

```
pid_t child_pid = fork();

if (child_pid == 0){
    /* only child comes here */

    printf("Child!\n");

    exit(0);
}
else{

    printf("Parent!\n");
}
```

- What are the possible output (assuming fork succeeds) ?
 - Child!
Parent!
 - Parent!
Child!
- How to get the child to always print first?

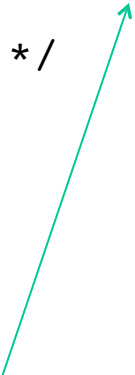
Process Examples

```
int status;
pid_t child_pid = fork();

if (child_pid == 0){
    /* only child comes here */

    printf("Child!\n");

    exit(0);
}
else{
    waitpid(child_pid, &status, 0);
    printf("Parent!\n");
}
```



- Waits til the child has terminated.
Parent can inspect exit status of child using 'status'
 - WEXITSTATUS(status)
- Output always:
Child!
Parent!

Process Examples

```
int status;  
pid_t child_pid = fork();  
char* argv[] = {"/bin/ls", "-l",  
NULL};  
char* env[] = {..., NULL};
```

```
if (child_pid == 0){  
    /* only child comes here */  
  
    execve("/bin/ls", argv, env);  
  
    /* will child reach here? */  
}  
else{  
    waitpid(child_pid, &status, 0);  
  
    ... parent continue execution...  
}
```

- An example of something useful.
- Why is the first arg “/bin/ls”?
- Will child reach here?