CS24: Introduction to Computing Systems

Spring 2015

Lecture 20

Last Time: UNIX Process Model

- Began covering the UNIX process model and API
- Information associated with each process:
  - A PID (process ID) to identify the process' context
  - Also, a parent process ID and a process group ID
  - Three states: Running, Stopped, Terminated
- Standard API calls to work with processes:
  - **`fork()`** creates a new process
  - **`exit()`** terminates a running process
  - **`wait()`**, **`waitpid()`** reap terminated ("zombie") child processes
  - **`execve()`** loads and runs a program in a process
  - **`kill()`** sends a signal to another process
- The kernel provides all of these operations

Last Time: The `init` Process

- Can build powerful facilities using process model
- `init` is the ancestor of all processes
  - Started as the last step of kernel boot sequence
  - PID of `init` is 1
- Purpose of `init` is to manage other processes in the operating system
- Switches between different runlevels
  - Each runlevel has a different (possibly overlapping) set of processes that are running
  - e.g. single-user, multi-user networking, X11
- Also handles some process-termination scenarios
  - (more on this today…)

Last Time: Zombie Processes

- A child process doesn't immediately go away when it terminates
  - Child process terminates with some status value…
  - Parent process may need to find out the child's status
- A terminated child process is called a <u>zombie</u>
  - The process is dead, but it hasn't yet been reaped
- Parent processes reap zombie children by calling:

  `pid_t wait(int *status)`
    - Waits for some child process to terminate

  `pid_t waitpid(pid_t pid, int *status, int options)`
    - Waits for a specific child process to terminate
    - Can also wait on children in a process-group, or all children
  - Both report an error if calling process has no children
  - Helpful functions for extracting details from **`status`**

NOTIFICATIONS FROM ZOMBIE CHILDREN

- When a child process terminates, the kernel also sends a **`SIGCHLD`** signal to its parent
  - Child process calls back to kernel when it terminates, so kernel can inform the process' parent
  - Default behavior is to ignore this signal
- Parent can set up a signal handler for **`SIGCHLD`** to reap the zombie child process

PARENT/CHILD PROCESS EXAMPLE

```
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#include <stdio.h>

/* Handle SIGCHLD signals. */
void handle_sigchld(int sig) {
  pid_t pid;
  int status;

  pid = wait(&status);

  /* NOT REENTRANT!      */
  /* Avoid in practice! */
  printf("Reaped child %d\n",
         pid);
  sleep(1);
}
```

```
int main() {
  int i;
  signal(SIGCHLD, handle_sigchld);

  for (i = 0; i < 3; i++) {
    if (fork() == 0) {
      /* Child-process code. */
      printf("Hello from child %d\n",
             getpid());
      sleep(5);
      return 0; /* Terminate child */
    }
  }

  /* Parent-process code. */
  while (1)   /* Wait for children */
    pause();  /* to terminate.     */

  return 0;
}
```

Parent/Child Process Example (2)

- Save, compile, and run from the command line:

```
[user@host:~]> ./reaped
Hello from child 1099!
Hello from child 1101!
Hello from child 1100!
```

  - *(5 seconds pass)*

```
Reaped child 1101
```

  - *(1 second passes)*

```
Reaped child 1100
```

  - *(…and then, nothing else…)*
- Hmm, last child process doesn't get reaped.
  - **ps** reports that process 1099 is a zombie ☹
  - **1099 ttys000   00:00:00 reaped <defunct>**

Our Example's Source Code

```c
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#include <stdio.h>

/* Handle SIGCHLD signals. */
void handle_sigchld(int sig) {
  pid_t pid;
  int status;

  pid = wait(&status);

  /* NOT REENTRANT!      *
   * Avoid in practice! */
  printf("Reaped child %d\n",
         pid);
  sleep(1);
}
```

Reaps one zombie per **SIGCHLD** received

```c
int main() {
  int i;
  signal(SIGCHLD, handle_sigchld);

  for (i = 0; i < 3; i++) {
    if (fork() == 0) {
      /* Child-process code. */
      printf("Hello from child %d\n",
             getpid());
      sleep(5);
      return 0; /* Terminate child */
    }
  }

  while (1)
    pause();

  return 0;
}
```

- Parent starts 3 child processes.
- Children terminate after five seconds.
- Kernel sends three **SIGCHLD** signals to the parent process.

Buggy Signal Handler!

- Problem:
  - Parent process is sent three **`SIGCHLD`** signals in a row
  - Each **`SIGCHLD`** signal takes <u>one second</u> to handle
    - Due to the **`sleep(1)`** in the signal handler
  - First **`SIGCHLD`** received:
    - Parent process handles the **`SIGCHLD`** signal
    - Kernel <u>blocks</u> other **`SIGCHLD`** signals while handler is running!
  - Second **`SIGCHLD`** received:
    - Parent can't receive it yet since it's still in its handler
    - Kernel records the pending **`SIGCHLD`** signal…
  - Third **`SIGCHLD`** received:
    - Parent is still in its **`SIGCHLD`** handler for the first signal
    - Since the process already has a pending **`SIGCHLD`** signal, the third **`SIGCHLD`** is discarded
- When parent's **`SIGCHLD`** handler returns, kernel delivers pending **`SIGCHLD`**
  - Third, dropped **`SIGCHLD`** is <u>never</u> delivered

A Better Signal Handler

- The kernel does not queue up signals for delivery!
  - Only has a **pending** bit-vector to track next signals to deliver
- Instead, handler should reap as many zombies as it can, each time it's invoked

```
void handle_sigchld(int sig) {
  pid_t pid;
  int status;

  while (1) {
    pid = waitpid(-1, &status, WNOHANG);
    if (pid <= 0)  /* No more zombie children to reap. */
      break;

    /* NOT REENTRANT!  Avoid in practice! */
    printf("Reaped child %d\n", pid);
  }
  sleep(1);
}
```

(CS:APP Figs 8.32-33 don't use **WNOHANG**, but they should…)

SIGNAL HANDLERS

- Signals cannot be used to count the number of occurrences of an event
- Signals indicate that *at least one* event of that type has occurred

- If expecting multiple signals of a particular type, write the handler to do as much work as possible
  - When your handler is invoked, you know that *at least one* signal of the type was received…

Process Termination Notes

- Every process has a parent process
  - The parent must be a currently running process
- If a parent process dies before its children do:
  - The kernel makes **`init`** (process 1) the parent of the orphaned children
  - When child processes die, **`init`** receives a **`SIGCHLD`** and then reaps them
- If a process terminates while it still has zombie children, the **`init`** process reaps the zombies too
  - The child processes become children of **`init`**…
  - Then, **`init`** is informed that it has zombie children
  - **`init`** reaps the zombie processes

Loading and Running Programs

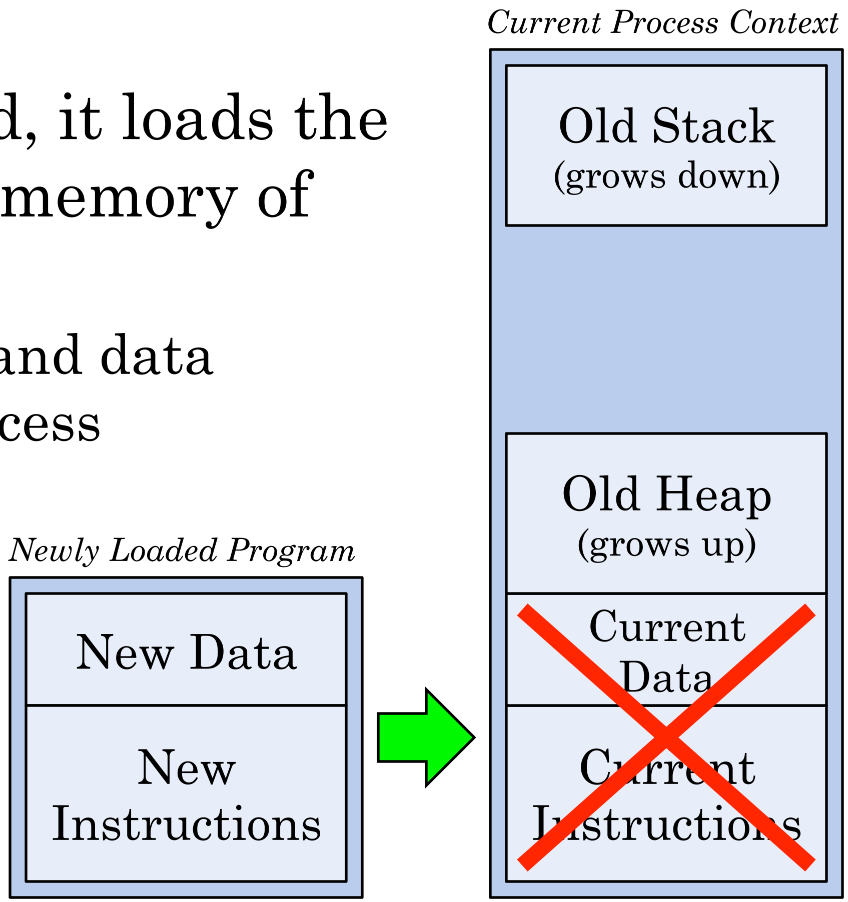
- The **`execve()`** function is used to load and run a program in the current process context

```
int execve(char *filename,
           char *argv[], char *envp[])
```

  - **`filename`** is the binary file (or script) to load and run
    - Scripts must start with "**`#!`** ***`interpreter`*** **`[`*`args`*`]`**" line
  - Program arguments are specified by **`argv`**
    - These are the command-line arguments
    - Last element in **`argv`** must be **`NULL`**
  - The program's environment is specified in **`envp`**
    - Each environment variable is a "NAME=VALUE" pair
    - Last element in **`envp`** must be **`NULL`**
- On success, loaded program *replaces* the current process' context and starts running from its start
  - On success, **`execve()`** does not return!

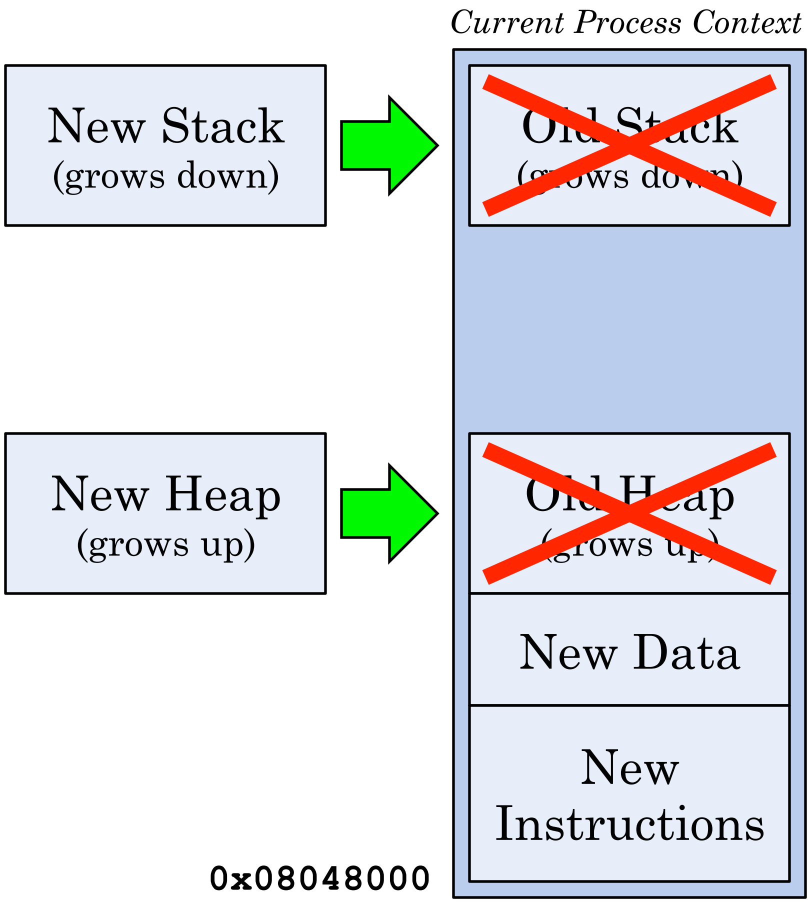
Loading and Running Programs (2)

- **`execve()`** is an operating system function
  - Kernel has total control over what happens in processes
- When **`execve()`** is invoked, it loads the specified program into the memory of the current process
  - Overwrites the instruction and data segments of the current process



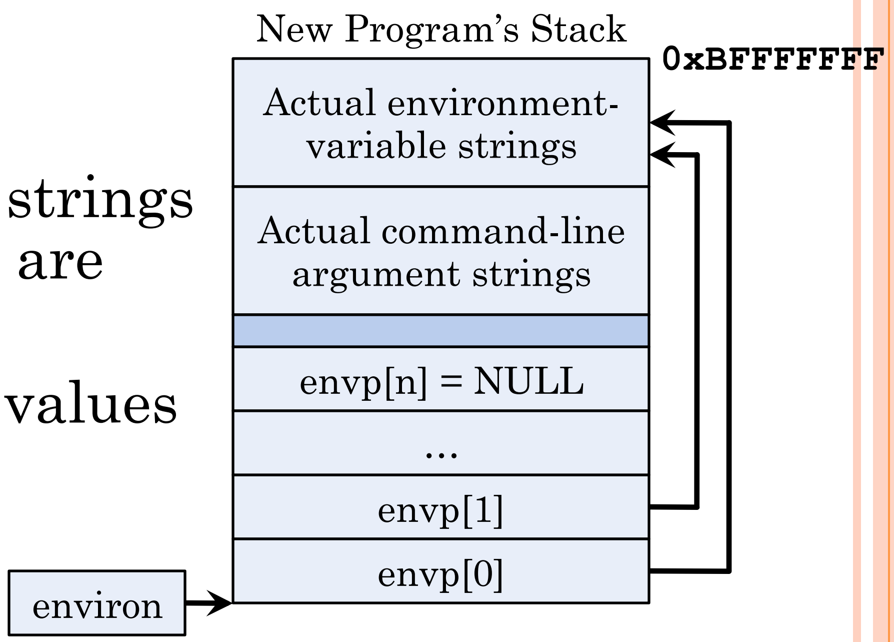
Loading and Running Programs (3)

- Next, **`execve()`** calls the program's entry point
  - For Linux on IA32, always at address 0x08048000
- For C/C++ programs, the entry point sets up the program's stack and heap
- New stack includes:
  - The new program's environment variables (passed to **`execve()`**)
  - The program's command-line arguments (also passed to **`execve()`**)
  - Arguments to the program's **`main()`** function



New Program's Stack

- First, program's environment strings and command-line arguments are copied to top of stack
- Next, program's environment values are pushed onto stack
  - Pointers reference the strings higher up in the stack
  - **`unistd.h`** defines an external **`environ`** pointer for accessing these values

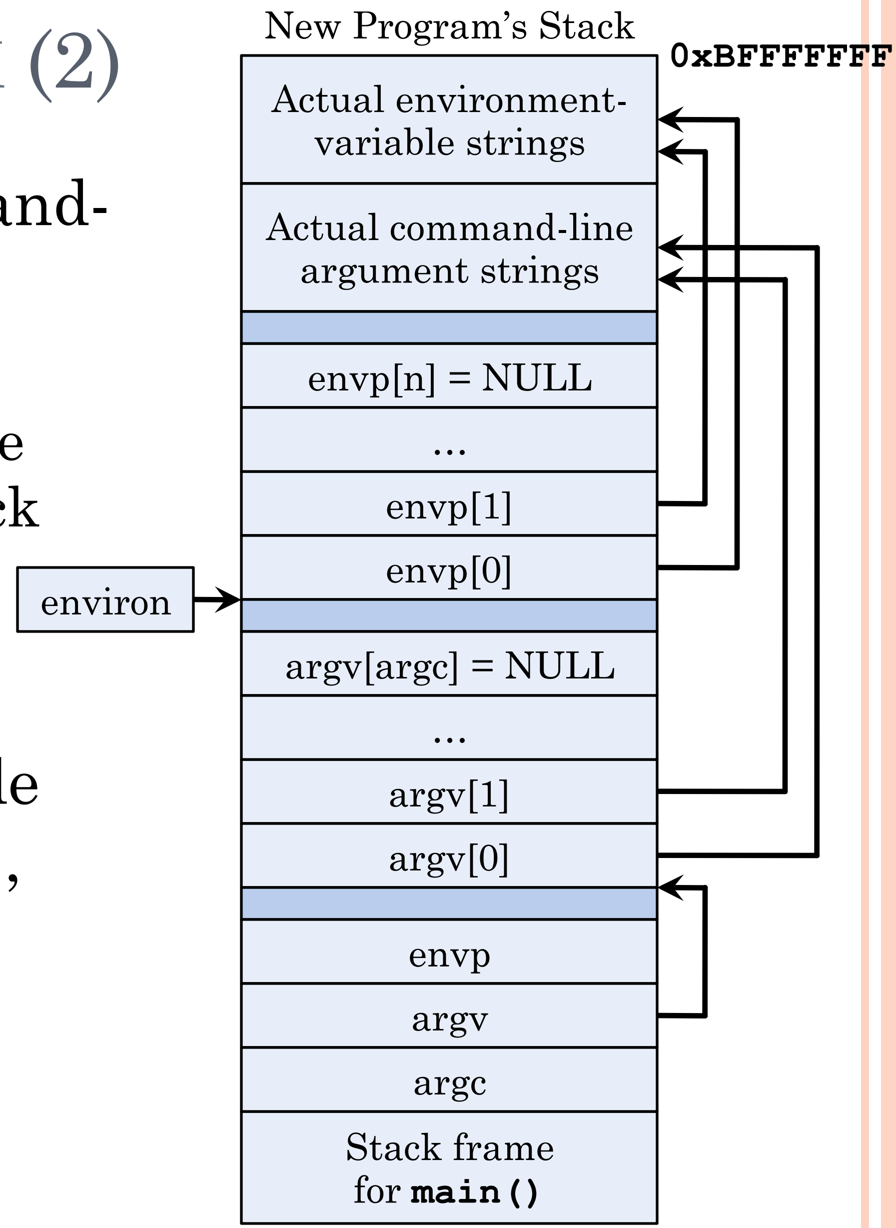


- Several functions for accessing environment variables

```
char * getenv(char *name)
int putenv(char *string)
int clearenv()
int setenv(char *name, char *value, int overwrite)
int unsetenv(char *name)
```

NEW PROGRAM'S STACK (2)

- Next, the program's command-line arguments are pushed onto the stack
  - Again, pointers reference the strings higher up in the stack
- Finally, the entry-point code sets up for a call to **`main()`**, and then calls it!



fork() AND **execve()**

- UNIX shells use a combination of **fork()** and **execve()** to run programs

```
void eval(char *cmdline) {
  char *argv[MAXARGS];
  pid_t pid;
  int status;

  parseline(cmdline, argv);        /* Tokenize command */
  pid = fork();     /* Spawn a process to run command */
  if (pid == 0) {   /* This is the child process!     */
    /* Replace the shell process with the new program */
    if (execve(argv[0], argv, environ) < 0) {
      printf("%s:  command not found!\n", argv[0]);
      exit(1);
    }
  }

  /* Shell process waits for the command to finish. */
  waitpid(pid, &status, 0);
}
```

`fork()` AND `execve()` (2)

- Basic idea:
  - **`fork()`** off a child process, then the child **`execve()`**s the new program
  - New program *completely replaces* the shell state in the child process
- Many useful variations! For example:
  - Background processes
    - If command should be run in background, shell simply doesn't **`waitpid()`** for the child to terminate
  - Built-in commands
    - Shell can check if specified command is a built-in operation
    - Update shell's internal state instead of forking off a process
  - Input and output redirection
    - Child process can change stdin and stdout file-descriptors before executing the new program

`fork()` AND `execve()` (3)

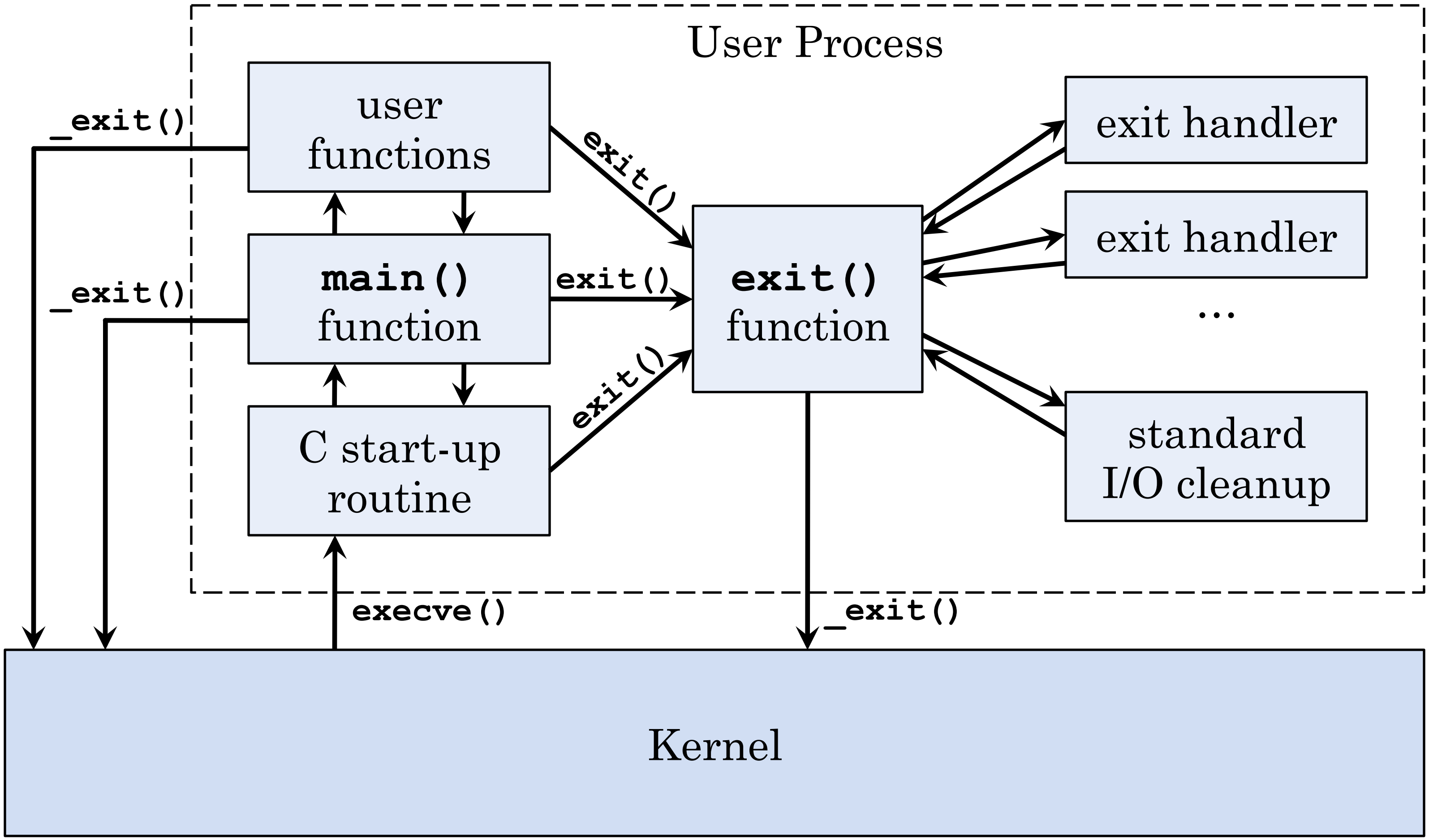
- This example code also lacks important features
  - Doesn't include any error checking on UNIX process functions, which is absolutely essential!
  - If we add support for background processes, need to reap them! (Requires a signal handler for this.)

- For a more detailed discussion of shell program structure, see CS:APP §8.4.6, and Problem 8.26

- For a *very* detailed discussion of process control:
  - Advanced Programming in the UNIX Environment (2nd edition) by W. Richard Stevens, Stephen Rago

EXEC/EXIT PROCESS LIFECYCLE

- Simple diagram of process lifecycle, from **`execve()`** to **`exit()`**:



Summary: UNIX Process Model

- UNIX process model is simple, but very powerful
  - API is clean and relatively straightforward
  - Can easily build up very sophisticated abstractions and systems using the simple model
- In last two lectures, we have covered:
  - How to start processes, how to handle process termination, how to load and start programs, …
- This is enough to implement basic interactive-shell support for the computer!
- Apps take these tools provided by OS, and build even more sophisticated services with them

UNIX Processes: Under the Hood

- This is how the UNIX process abstraction is exposed to user programs…
- How this model is implemented in the kernel is *much* more involved
  - Process states and state-transitions
  - Details that the kernel must keep track of
  - Process behavior and scheduling
- Many platform-specific design choices to make!
  - This will be a high-level overview of general themes

Process States

- Processes follow a specific series of state transitions
- UNIX process API exposes these states:
  - Running – the process is actively running on a CPU, or is waiting to be executed
  - Stopped – the process will not be scheduled for execution until it is resumed
  - Terminated – the process has permanently stopped executing
- Process model allows multiple running processes…
  - …but really, the number of CPUs dictates how many processes can actually <u>run</u> at any given time
  - e.g. 4 cores = 4 running processes, and that's it!
- Kernel must distinguish between processes that are running, and processes that are "ready to run"
  - Ready processes are waiting to get a turn on the CPU

Process States (2)

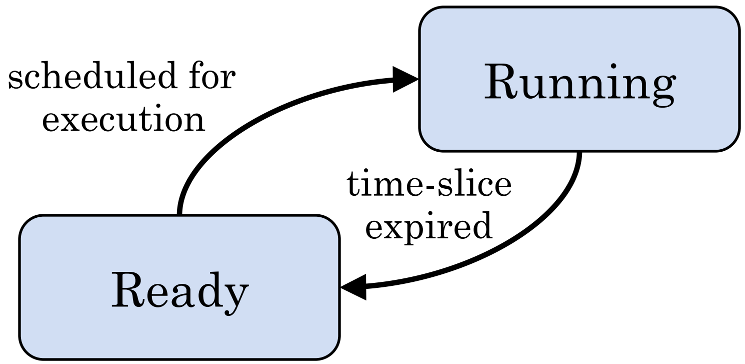
- Running processes can make slow system calls
  - Read or write a block of data to the disk
  - Read or write a data buffer to the network
  - Read user input
  - Kernel mediates these operations, so it can schedule other processes while long-running tasks complete
- Introduce another process state: blocked
  - A blocked process can't continue until some external condition changes
  - e.g. a DMA transfer from disk to memory must complete, and a hardware interrupt must occur
  - While blocked, makes no sense to schedule process!

Process States (3)

- Running processes can be stopped
  - e.g. type Ctrl-Z at terminal, which sends **`SIGTSTP`** signal to the process
  - Stopped processes are not scheduled until they are resumed, by sending a **`SIGCONT`** signal to them
- Blocked processes necessarily complicate this…
  - A blocked process can be stopped and resumed
  - While a blocked process is stopped, it can become unblocked
- Kernel needs to keep track of all of these states
- Define a state diagram for processes
  - Specifies the states and transitions for processes managed by the kernel

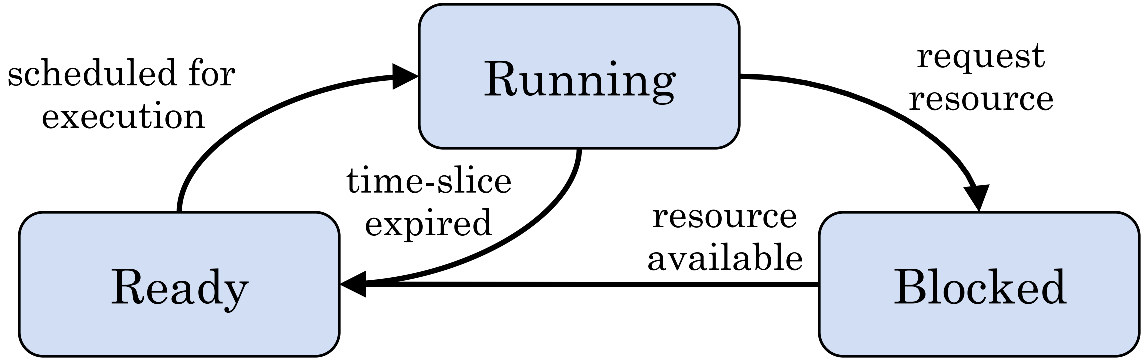
UNIX Process State Diagram (1)

- One process is actually Running, per CPU
  - Ready processes wait for access to the CPU
  - Kernel interrupts the running process to run another



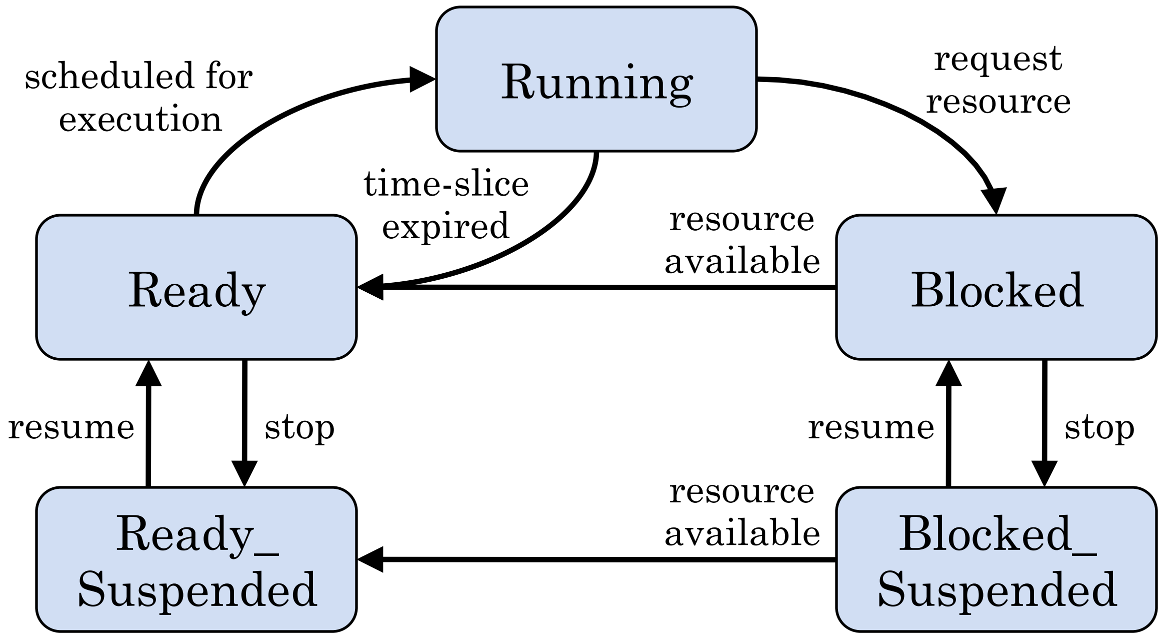
UNIX Process State Diagram (2)

- A Running process can become Blocked
  - Request a resource that won't be available for a while
  - When resource is available, changes back to Ready



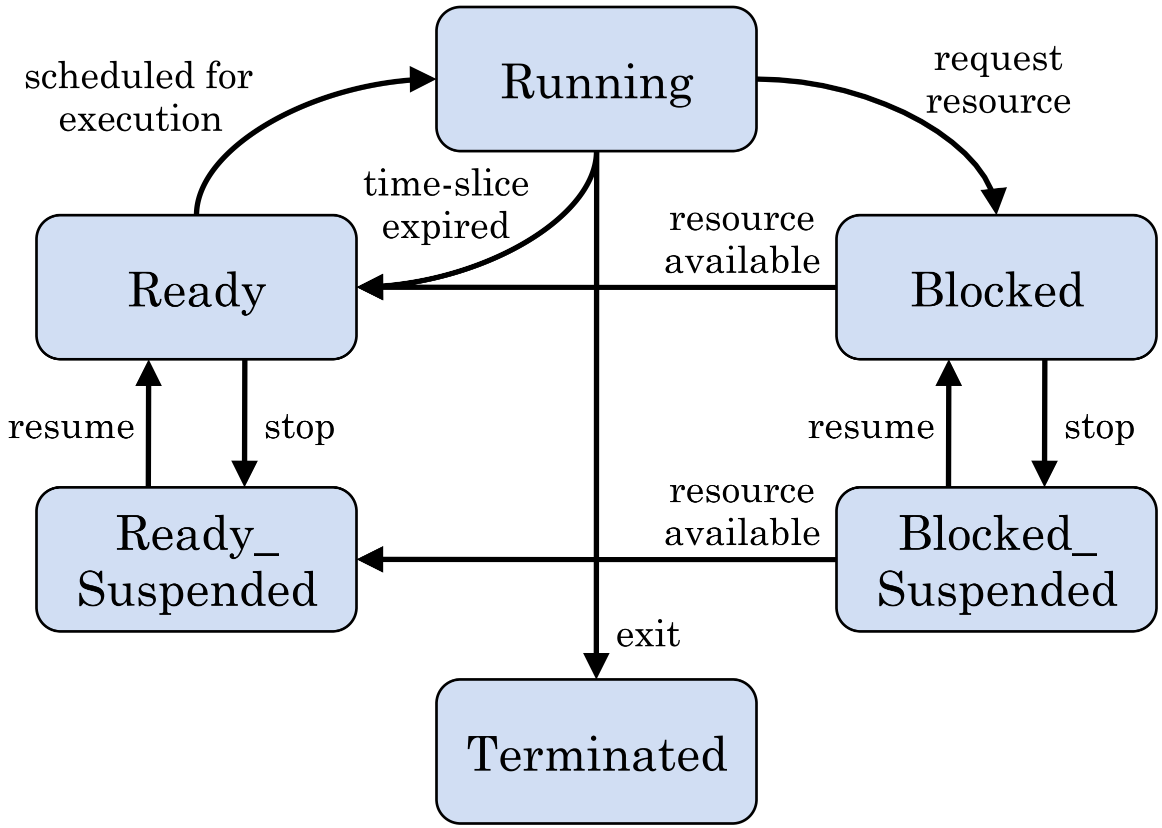
UNIX Process State Diagram (3)

- Stopping a process causes it to become suspended
  - If resource becomes available while suspended, a blocked process changes to ready/suspended



UNIX Process State Diagram (4)

- Running process may also terminate
  - Most resources can be reclaimed by kernel…
  - Needs to preserve status until the zombie is reaped



States and Processes

- At this level, only one process may be in the Running state for each CPU in the computer
  - e.g. 4 cores = 4 processes in Running state
- <u>Many</u> processes can be in the other states!
- Kernel needs to manage collections of processes in each state
  - Different strategies for managing these processes, so different kinds of collections are employed
- Next Time:
  - Data structures used by kernel to keep track of process context and state information