Killing Zombies, Working, Sleeping, and Spawning Children

CS 532
Prof. Karavanic

(c) 2020 Karen L. Karavanic

An Operating System…

- **Is a program that controls the execution of application programs**
  - OS must relinquish control to user programs and regain it safely and efficiently
  - OS tells the CPU when to execute other programs and what specific program to execute next
- **How to provide the illusion of unlimited CPU availablity?**
  - Virtualization.

(c) 2020 Karen L. Karavanic

The Process Model

- How should we represent the running code / program?
- What does the OS do?
  - Protect system resources
    - E.g. from user programs, from malicious code
  - Allocate memory
  - Allocate processor time
  - Handle errors
  - Accounting
  - Given this list of tasks, what is the best *abstraction* for the running code?
    - Proposal #1: function
      - How can we describe a running function?

The Process Model

- How should we represent the running code / program?
- What does the OS do?
  - Protect system resources
    - E.g. from user programs, from malicious code
  - Allocate memory
  - Allocate processor time
  - Handle errors
  - Accounting
  - Given this list of tasks, what is the best *abstraction* for the running code?
    - Proposal #1: function
    - How can we describe a running function?
    - Proposal #2: “whole program”

The Process Model

- A Process = a program in execution
- A process includes:
  - program counter (PC): what instruction to execute next?
  - Stack: one stack frame per function called
  - Register values: what values currently held in hardware registers?
- Program is passive entity, process is active
  - Program becomes process when executable file loaded into memory

The Process Model

- The Process Address Space includes:
  - **text section:** the program code
  - **program counter**: next instruction to execute
  - **Values of processor registers**: smallest, fastest memory
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time
  - Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program

Processes

- *process == task*
  - key idea: the OS *virtualizes* the CPU: we can have more than one program executing but each one is written as if it has the CPU to itself
  - unit of execution. in timeshare system, each user had a separate shell. in multiprocessor server or multicore chip, user has > 1 task running concurrently
  - process has **context**: structures in system that constitute that process

(c) 2020 Karen L. Karavanic

Process Control Block (PCB)

Information associated with each process includes:

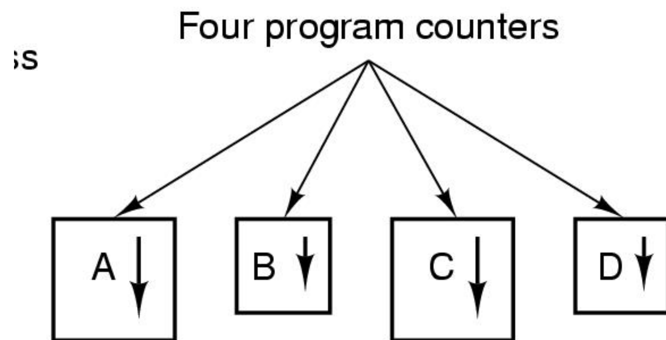
- Process state
- Program counter value
- CPU register values
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

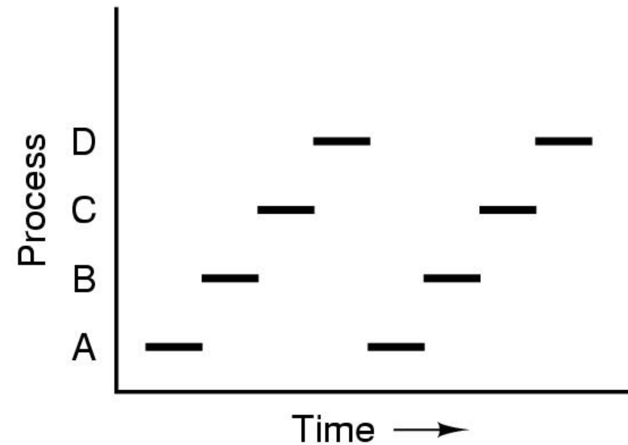
Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues

Processes

The Process Model





- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes
- Only one program active at any instant

(c) 2020 Karen L. Karavanic

Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a **context switch**.
- **Context** of a process represented in the PCB
- Context switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB -> longer the context switch
- Context switch time is dependent on hardware support
  - Ex: Some hardware provides multiple sets of registers per CPU -> multiple contexts loaded at once

Process Creation

Principal events that cause process creation:

- System initialization
- Execution of a process creation system
- User request to create a new process
- Initiation of a batch job

(c) 2020 Karen L. Karavanic

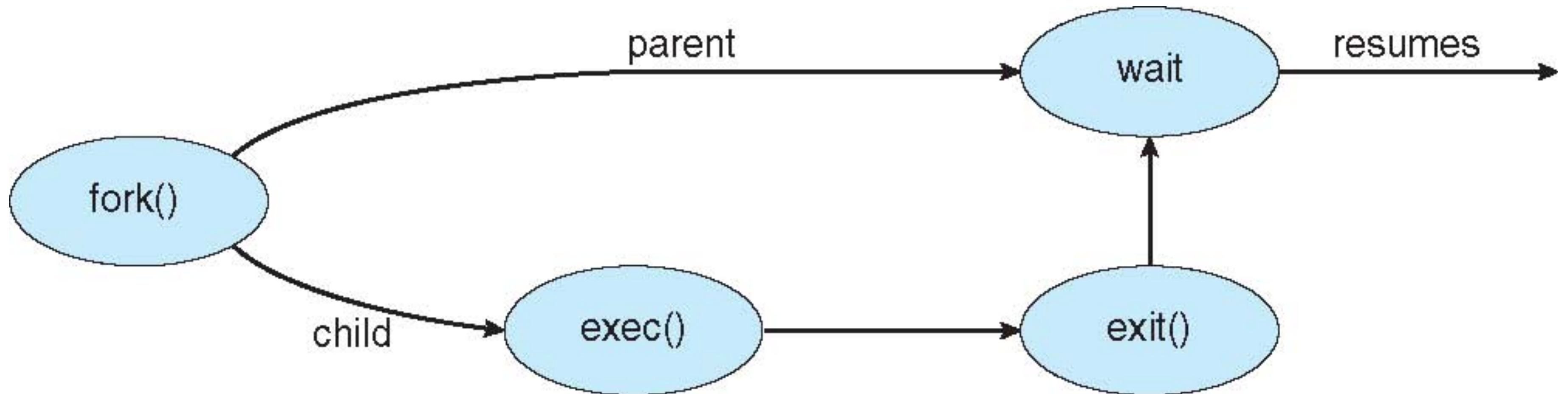
Process Creation

- **Parent** process creates **child** processes, which, in turn create grandchild processes, forming a tree of processes
- Generally, process identified and managed via **a process identifier** (**pid**)
- Resource sharing – different models
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation (Cont.)

- Address space
  - Child is created as a *duplicate* of parent:
    - Same program loaded into text segment
    - Same values in data segment
    - Same stack
    - Same register values
    - Same PC value
- UNIX/Linux examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program

Process Creation



Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system

- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
      - All children terminated - **cascading termination**

Process Termination

Conditions which terminate processes

1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (involuntary)
4. Killed by another process (involuntary)

   The kill command:

   kill [pid]

   can any process kill any other process?

(c) 2020 Karen L. Karavanic

Unix System Calls for Process Management

System call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, opts)	Wait for a child to terminate
s = execve(name, argv, envp)	Replace a process' core image
exit(status)	Terminate process execution and return status

s: an error code

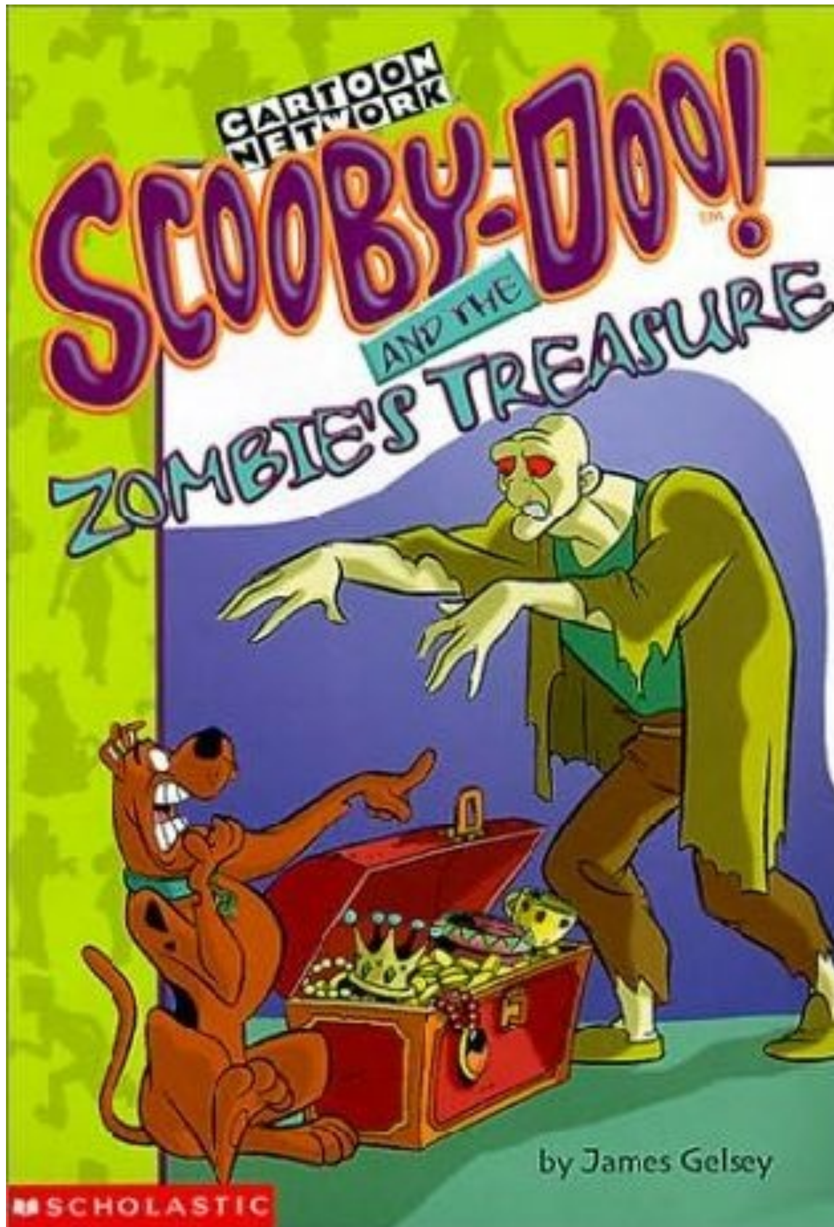
pid: a process ID

status: *Did the process exit normally*?

Yes:

No:

(c) 2020 Karen L. Karavanic



What is a zombie process??

(c) 2020 Karen L. Karavanic

POSIX signals - simple IPC

- IPC - interprocess communication
- Also used for kernel to process communication
- software interrupts for application
  - you arrange for asynchronous call to a function that will *handle* the signal
- signals are *events* – we can’t predict what line of code will be executing when they occur
- can be caught, sent, or ignored (SIG_IGN)
- use kill(1) and kill(2) to send them (some of them)

(c) 2020 Karen L. Karavanic

Example Signal Functions

- SIGINT - catch control-C, exit program
- SIGSEGV - o.s. (mmu) forces program to exit because of memory violation
- SIGHUP - signal sent to daemon to tell it to exit system cleanly (shutdown) or to reread control file
- SIGSTOP, SIGCONT, job control of processes (SIGSTOP goes to background)
- SIGKILL - go away NOW ... % kill -9 <pid>

(c) 2020 Karen L. Karavanic

POSIX Signals

Signal	Cause
SIGABRT	Sent to abort a process and force a core dump
SIGALRM	The alarm clock has gone off
SIGFPE	A floating-point error has occurred (e.g., division by 0)
SIGHUP	The phone line the process was using has been hung up
SIGILL	The user has hit the DEL key to interrupt the process
SIGQUIT	The user has hit the key requesting a core dump
SIGKILL	Sent to kill a process (cannot be caught or ignored)
SIGPIPE	The process has written to a pipe which has no readers
SIGSEGV	The process has referenced an invalid memory address
SIGTERM	Used to request that a process terminate gracefully
SIGUSR1	Available for application-defined purposes
SIGUSR2	Available for application-defined purposes

(c) 2020 Karen L. Karavanic

Unix shells

- Shell is not part of the kernel, but makes heavy use of it
- User logs in over terminal, network, gets shell command line
- Each command is executed in a child process
- Each process at startup has 3 open file descriptors: stdin, stdout, stderr (fd=0, 1, 2)
- Child inherits these from parent

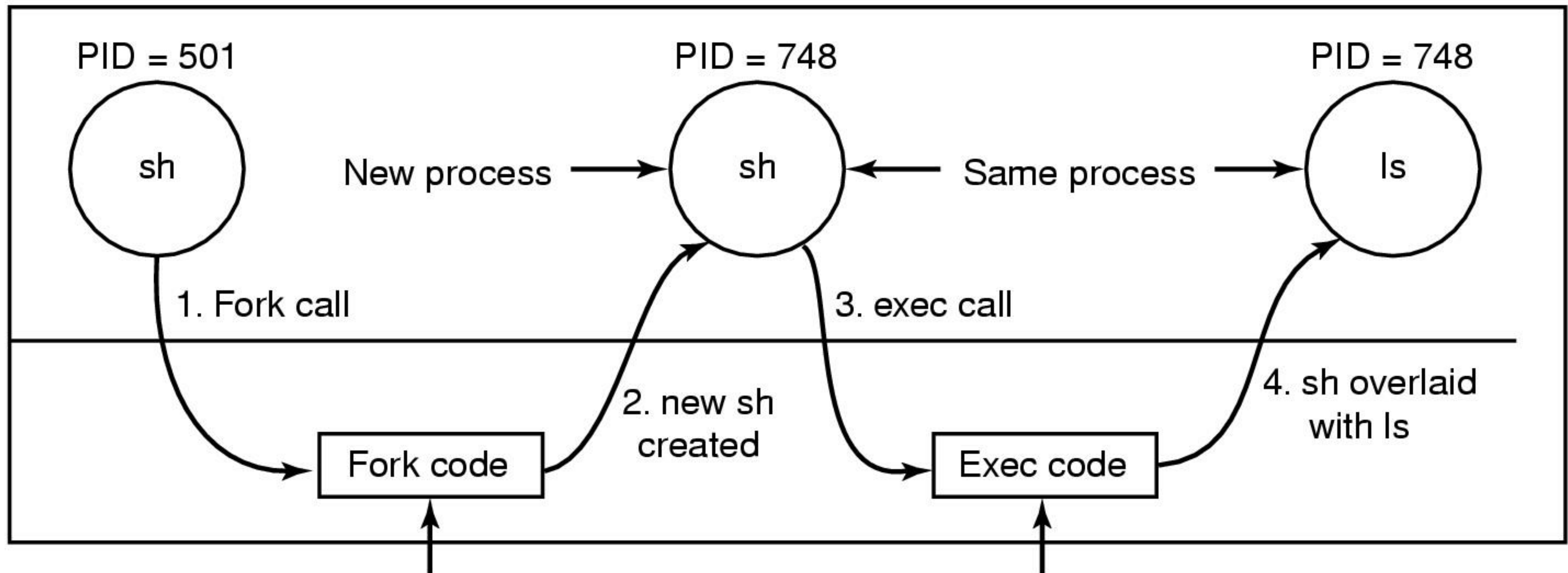
(c) 2020 Karen L. Karavanic

shellgame …

- shells handles redirection and pipes with certain metacharacters (>,<,|)
- `date > file`
- `sort < file1 > file2`
- `ls | more`
- can execute programs in background too
- `ls &`

(c) 2020 Karen L. Karavanic

Steps in executing *ls* at the Command line



Allocate child's process table entry
Fill child's entry from parent
Allocate child's stack and user area
Fill child's user area from parent
Allocate PID for child
Set up child to share parent's text
Copy page tables for data and stack
Set up sharing of open files
Copy parent's registers to child

Find the executable program
Verify the execute permission
Read and verify the header
Copy arguments, environ to kernel
Free the old address space
Allocate new address space
Copy arguments, environ to stack
Reset signals
Initialize registers

(c) 2020 Karen L. Karavanic