Lecture 5

Jan 11, 2024

fork-exec model:

`fork()` creates a copy of the current process

`exec*()` replaces the current process' code and address space with the code for a different program

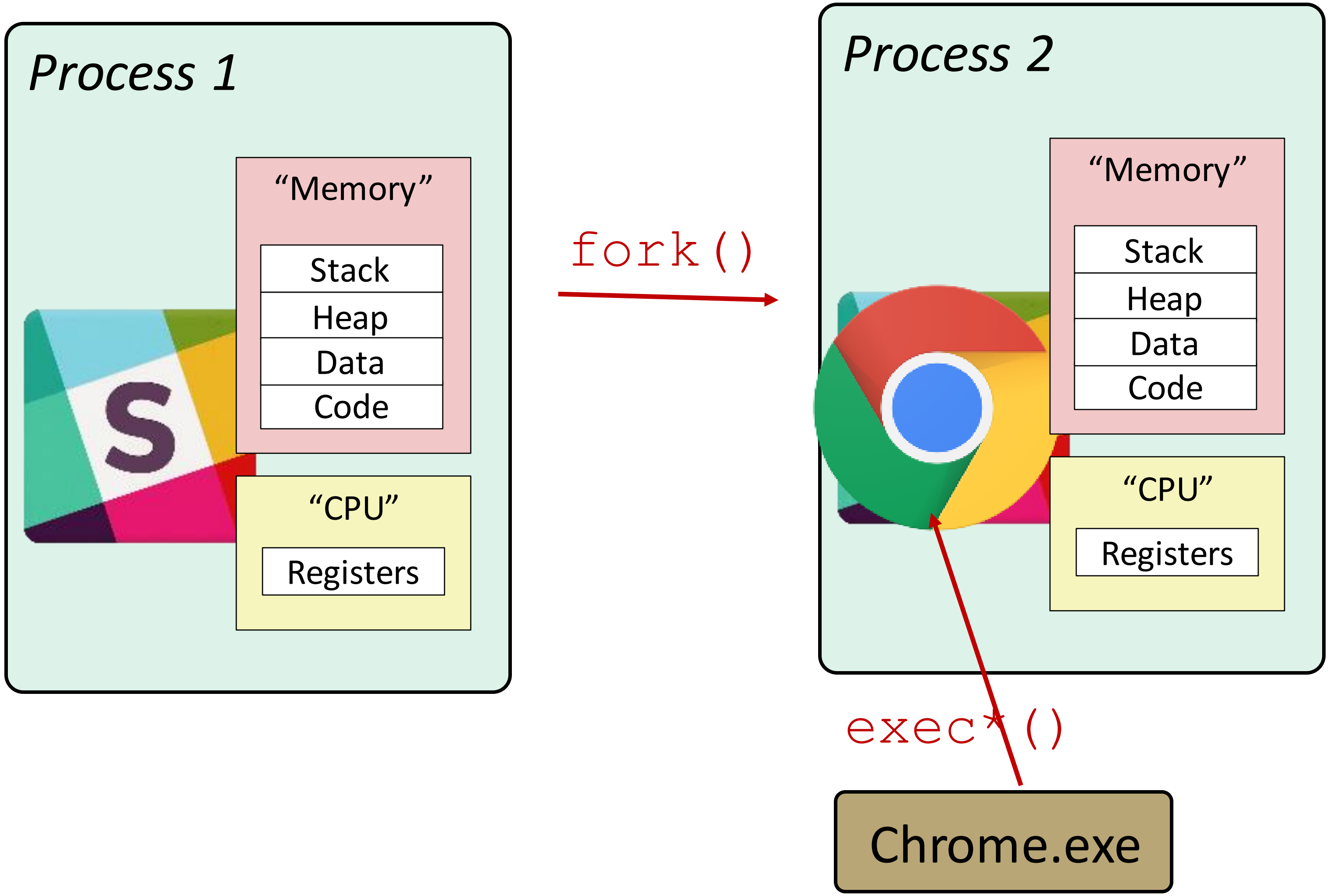
- Whole family of `exec` calls – see **`exec(3)`** and **`execve(2)`**

```
// Example arguments: path="/usr/bin/ls",
//      argv[0]="/usr/bin/ls", argv[1]="-ahl", argv[2]=NULL
void fork_exec(char *path, char *argv[]) {
   pid_t pid = fork();
   if (pid != 0) {
      printf("Parent: created a child %d\n", pid);
   } else {
      printf("Child: about to exec a new program\n");
      execv(path, argv);
   }
   printf("This line printed by parent only!\n");
}
```

What happens during exec?

- •After fork, parent and child are running same code
- –Not too useful!
- •A process can run exec() to load another executable to its memory image
- –So, a child can run a different program from parent
- •Variants of exec(), e.g., to pass command line arguments to new executable
- On Linux, there are six variants of exec(): execl, execlp(), execle(), execv(), execvp(), and execvpe().
- Read the man pages to learn more.

Exec()

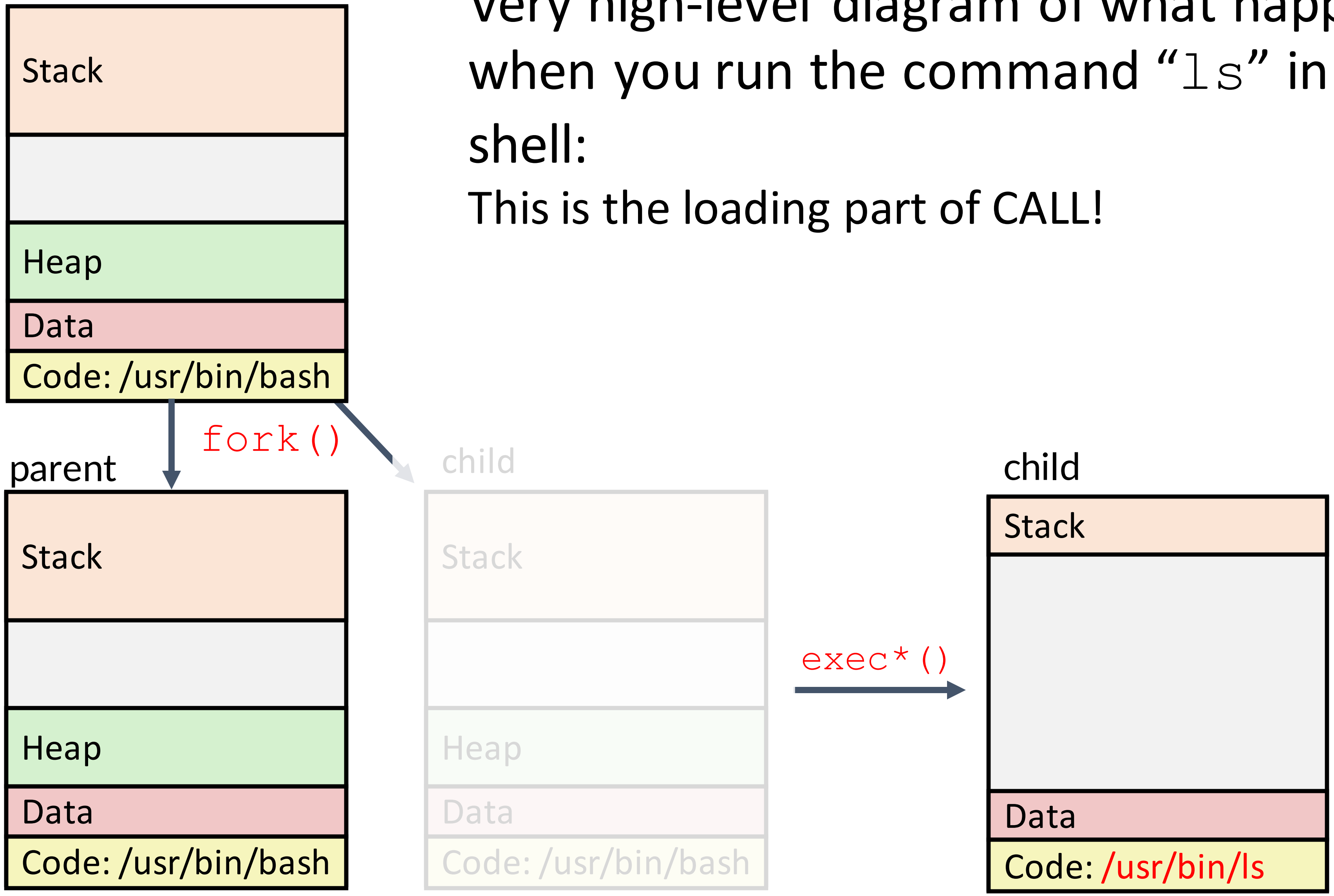

Process 1
"Memory"
Stack
Heap
Data
Code
"CPU"
Registers
fork()
Process 2
"Memory"
Stack
Heap
Data
Code
"CPU"
Registers
exec*()
Chrome.exe

Why do we need fork() and exec()

- In a basic OS, the init process is created after initialization of hardware
- The init process spawns a shell like bash
- Shell reads user command, forks a child, execs the command executable, waits for it to finish, and reads next command
- Common commands like ls are all executables that are simply exec'ed by the shell

Very high-level diagram of what happens when you run the command "`ls`" in a Linux shell:

This is the loading part of CALL!

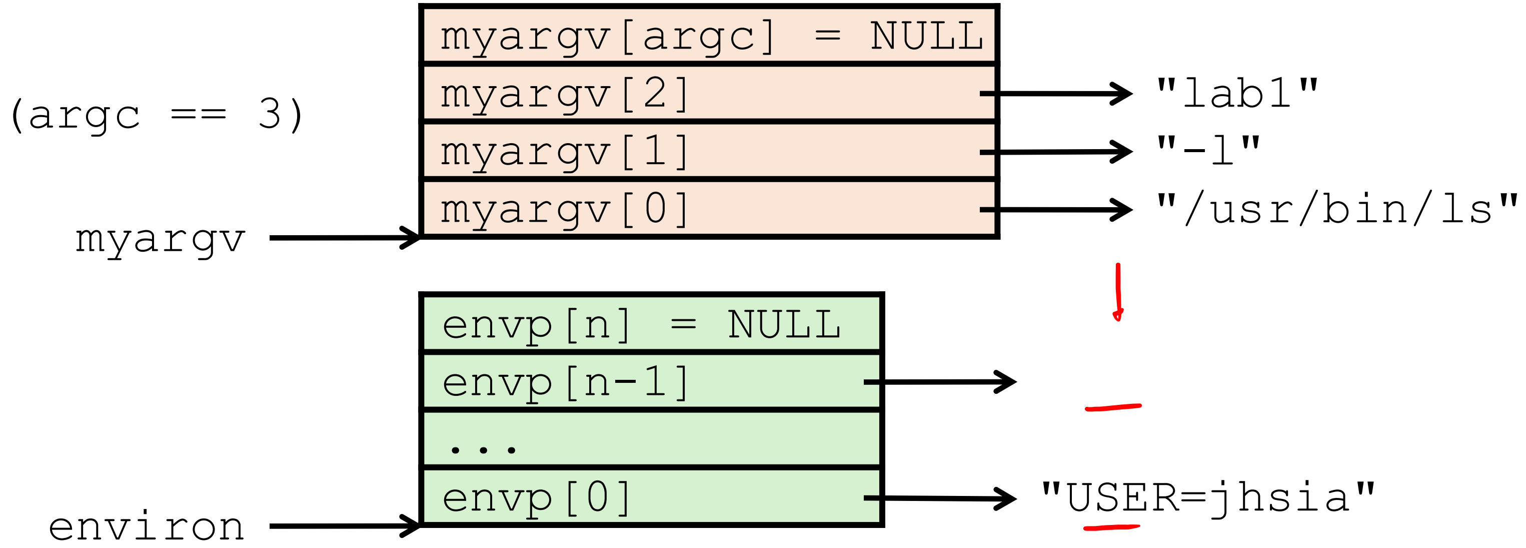


```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {         // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) { // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");   // program: "wc" (word count)
    myargs[1] = strdup("p3.c"); // argument: file to count
    myargs[2] = NULL;           // marks end of array
    execvp(myargs[0], myargs);  // runs word count
    printf("this shouldn't print out");
  } else {              // parent goes down this path (main)
    int rc_wait = wait(NULL);
    printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
            rc, rc_wait, (int) getpid());
  }
  return 0;
}
```

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
     29     107    1030 p3.c
hello, I am parent of 29384 (rc_wait:29384) (pid:29383)
prompt>
```

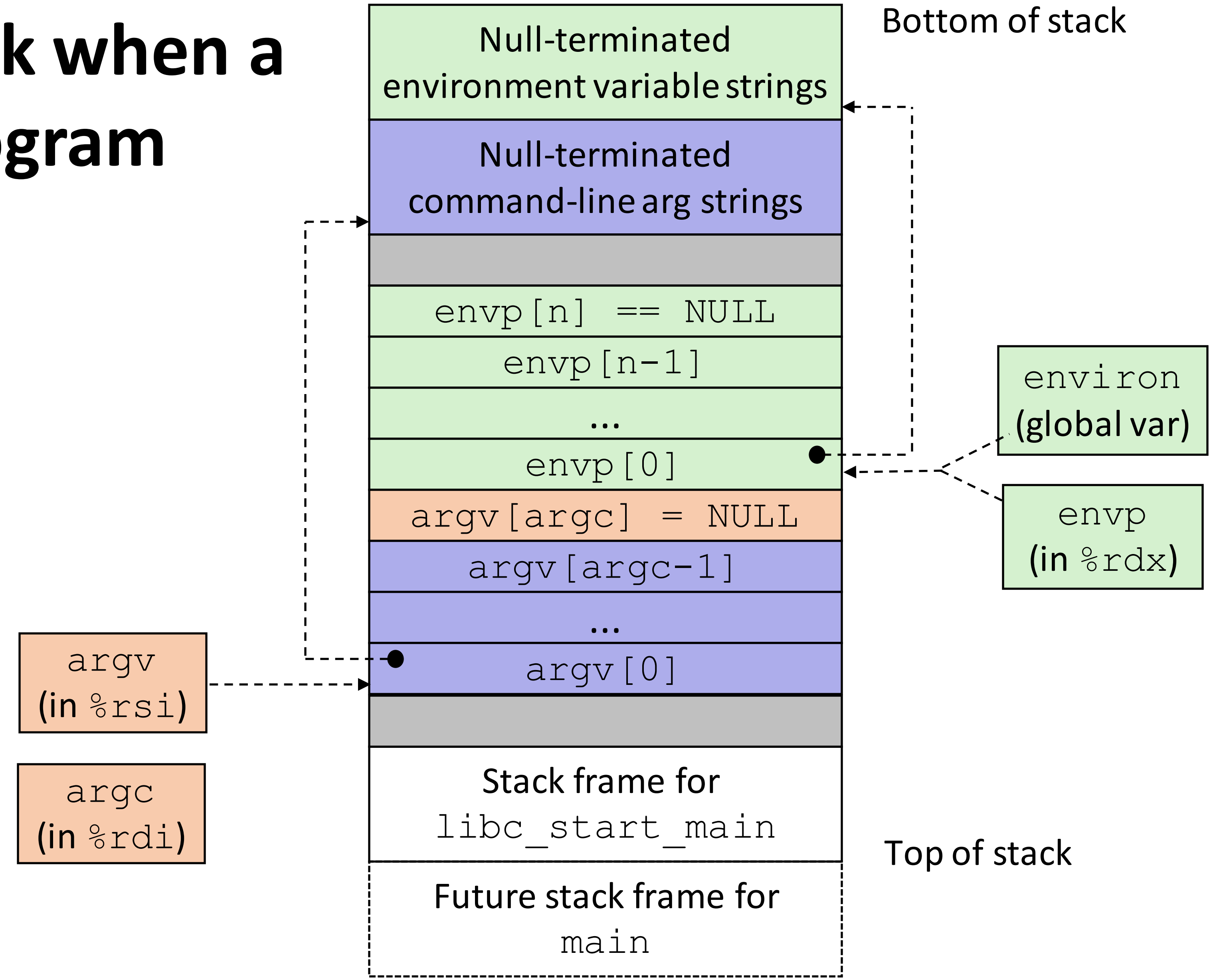
Execute `"/usr/bin/ls -l lab1"` in child process using current environment:




```
if ((pid = fork()) == 0) {    /* Child runs program */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: Command not found.\n", myargv[0]);
        exit(1);
    }
}
```

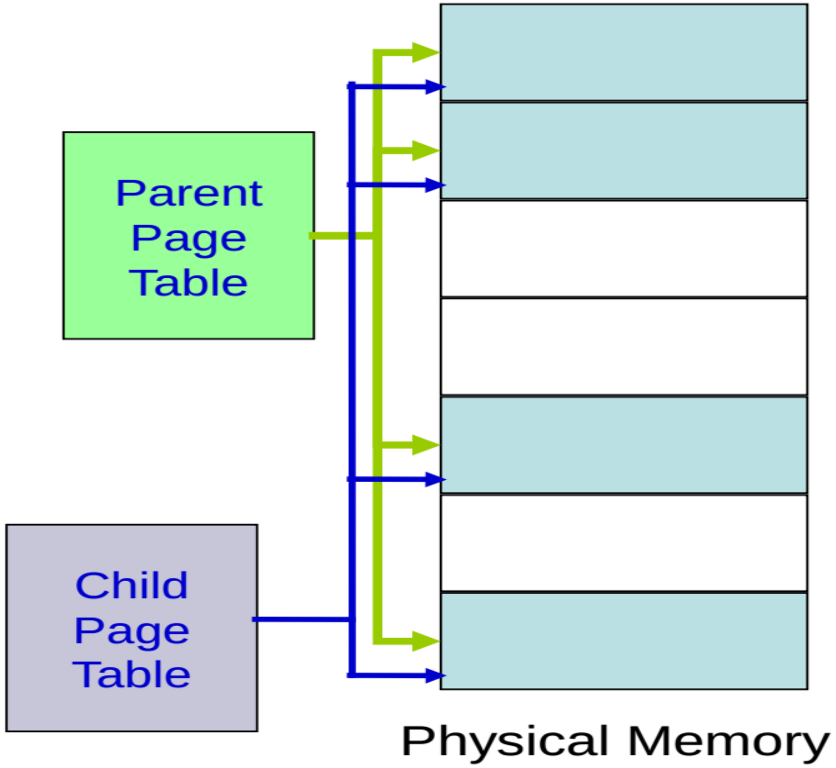
Run the `printenv` command in a Linux shell to see your own environment variables

Structure of the Stack when a new program starts



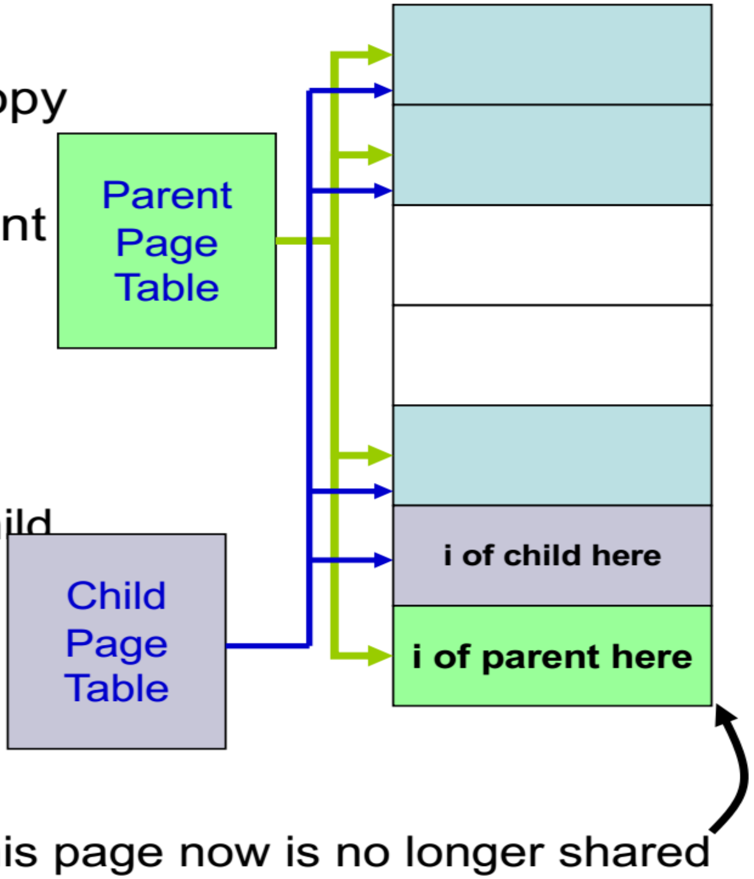
Virtual Addressing Advantage (easy to make copies of a process)

- Making a copy of a process is called forking.
  - Parent (is the original)
  - child (is the new process)
- When fork is invoked,
  - child is an exact copy of parent
    - When fork is called all pages are shared between parent and child
    - Easily done by copying the parent's page tables



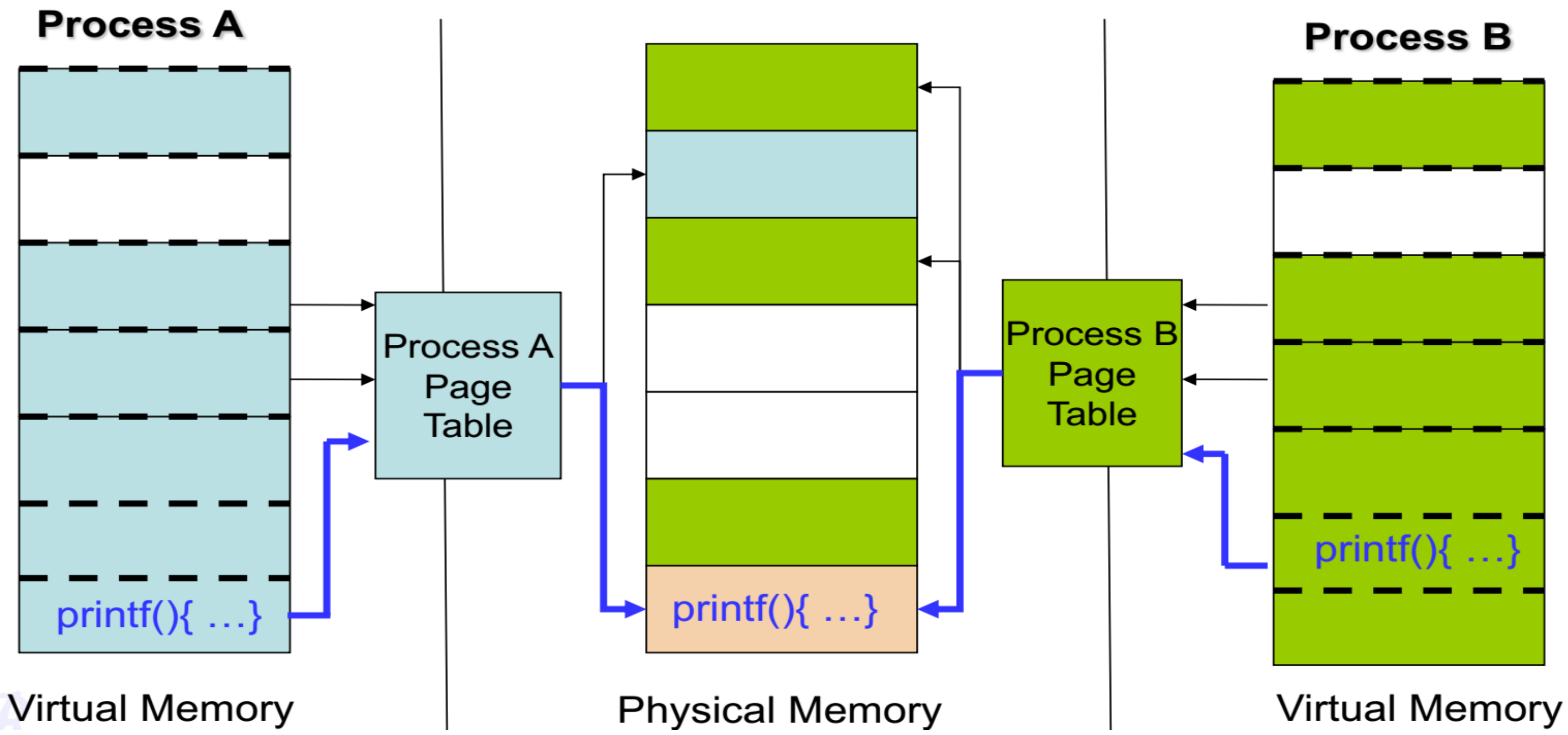
Copy on Write (CoW)

- When data in any of the shared pages change, OS intercepts and makes a copy of the page.
- Thus, parent and child will have different copies of this page
- Why?
  - A large portion of executables are not used.
  - Copying each page from parent and child would incur significant disk swapping.. huge performance penalties.
  - Postpone coping of pages as much as possible thus optimizing performance



Virtual Addressing Advantages (Shared libraries)

- Many common functions such as *printf* implemented in shared libraries
- Pages from shared libraries, shared between processes



How COW works??

- When forking,
- – Kernel makes COW pages as read only
- – Any write to the pages would cause a page fault
- – The kernel detects that it is a COW page and duplicates the page

More details on Shell

- Shell can manipulate the child in strange ways.
- Suppose you want to redirect output from a command to a file
- **`prompt>ls > foo.txt`**
- Shell spawns a child, rewires its standard output to a file, then calls exec on the child

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  int rc = fork();
  if (rc < 0) {
    // fork failed
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child: redirect standard output to a file
    close(STDOUT_FILENO);
    open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);

    // now exec "wc"...
    char *myargs[3];
    myargs[0] = strdup("wc");   // program: wc (word count)
    myargs[1] = strdup("p4.c"); // arg: file to count
    myargs[2] = NULL;           // mark end of array
    execvp(myargs[0], myargs);  // runs word count
  } else {
    // parent goes down this path (main)
    int rc_wait = wait(NULL);
  }
  return 0;
}
```

```
prompt> ./p4
prompt> cat p4.output
      32     109     846 p4.c
prompt>
```

So, should we rewrite programs for each OS?

- POSIX API: a standard set of system calls that an OS must implement
- –Programs written to the POSIX API can run on any POSIX compliant OS
- –Most modern
- OSes are POSIX compliant–Ensures program portability
- Program language libraries hide the details of invoking system calls
- –The printf function in the C library calls the write system call to write to screen
- –User programs usually do not need to worry about invoking system calls

Summary

- Processes
  - At any given time, system has multiple active processes
  - On a one-CPU system, only one can execute at a time, but each process appears to have total control of the processor
  - OS periodically "context switches" between active processes
- Process management
  - `fork`: one call, two returns
  - `execve`: one call, usually no return
  - `wait` or `waitpid`: synchronization
  - `exit`: one call, no return

`fork` makes two copies of the same process (parent & child)

Returns different values to the two processes

`exec*` replaces current process from file (new program)

Two-process program:

- First `fork()`
- **if** (pid == 0) { */* child code */* } **else** { */* parent code */* }

Two different programs:

- First `fork()`
- **if** (pid == 0) { execv(…) } **else** { */* parent code */* }

`wait` or `waitpid` used to synchronize parent/child execution and to reap child process

Process Execution

- **OS creates a process entry for the process in a process list**

•Allocates memory and creates memory image

- –Code and data (from executable)
- –Stack and heap
- **• Points CPU program counter to current instruction**
- –Other registers may store operands, return values etc.
- **• After setup, OS is out of the way and process executes directly on CPU**

A simple function call

- A function call translates to a jump instruction
- A new stack frame pushed to stack and stack pointer (SP)updated
- Old value of PC (return value) pushed to stack and PC updated
- Stack frame contains return value, function arguments etc.

How is a system call different?

- CPU hardware has multiple privilege levels
  - One to run user code: user mode
  - One to run OS code like system calls: kernel mode
  - Some instructions execute only in kernel mode
- Kernel does not trust user stack
  - Uses a separate kernel stack when in kernel mode
- Kernel does not trust user provided addresses to jump to
  - Kernel sets up Interrupt Descriptor Table (IDT) at boot time
  - IDT has addresses of kernel functions to run for system calls and other events

Mechanism of system call: trap instruction

- When system call must be made, a special trap instruction is run(usually hidden from user by libc)
- Trap instruction execution
  - –Move CPU to higher privilege level
  - –Switch to kernel stack
  - –Save context (old PC, registers) on kernel stack
  - –Look up address in IDT and jump to trap handler function in OS code

More on trap instruction

- Trap instruction is executed on hardware in following cases:

  –System call (program needs OS service)

  –Program fault (program does something illegal, e.g., access memory it doesn’t have access to)

  –Interrupt (external device needs attention of OS, e.g., a network packet has arrived on network card)

- Across all cases, the mechanism is: save context on kernel stack and switch to OS address in IDT
- IDT has many entries: which to use?

  –System calls/interrupts store a number in a CPU register before calling trap, to identify which IDT entry to use