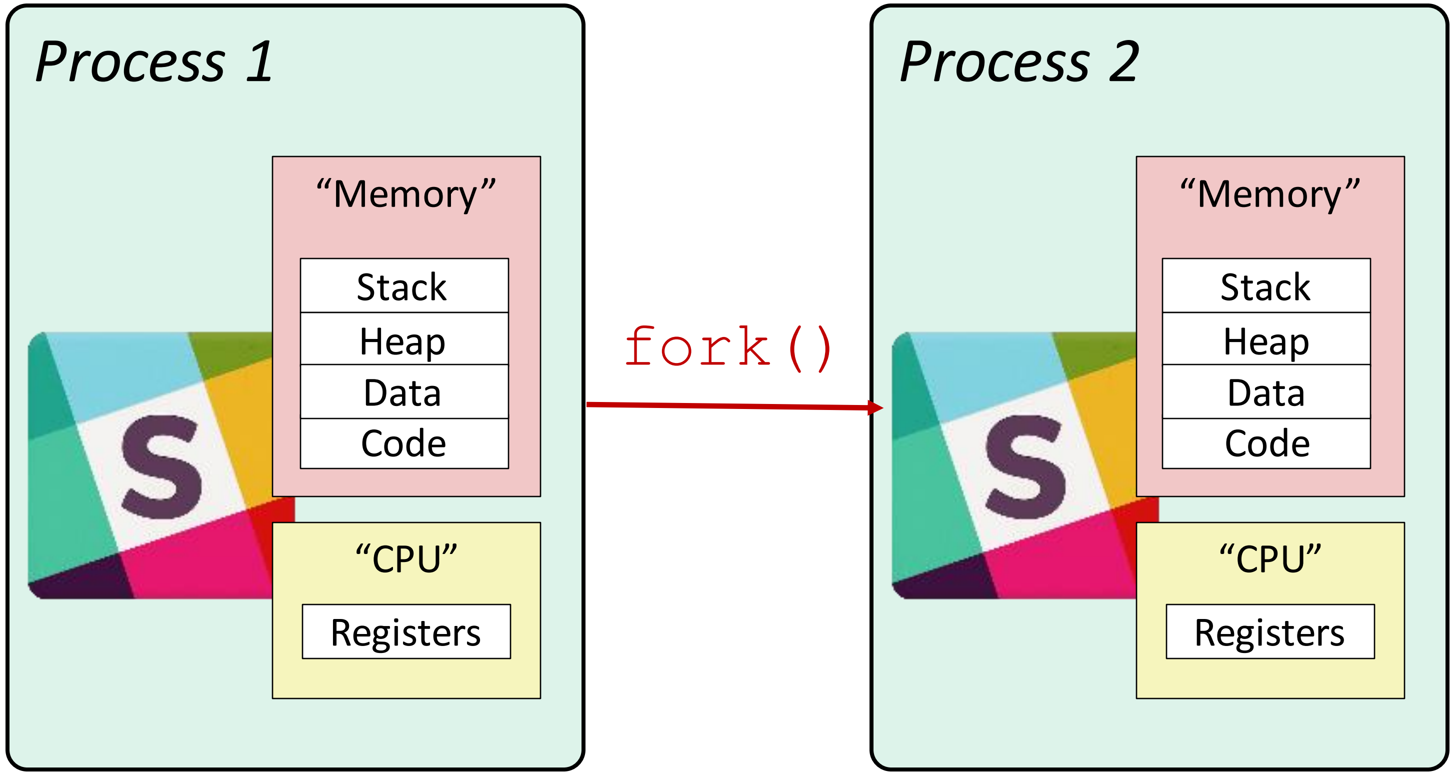
Lecture 4

JAN 10, 2024

Process creation

1. In POSIX compliant operating systems, a new processes can be created with the **fork()** system call.
2. The process that invoked fork is called the **parent** and the newly created "forked" process is called the **child**.
3. **fork()** returns to both parent and child. However, its return value is 0 (zero) to the child and the new process ID number (PID) is returned to the parent. Consequently, looking at the return value allows the child to identify itself
4. Other than that, the child is "born" with all the knowledge of its parent. New data sections for global variables and a heap are created and a new stack is created for the child. However, the initial contents of these new memory spaces contain the identical content for both parent and child.


Process 1
"Memory"
Stack
Heap
Data
Code
"CPU"
Registers
fork()
Process 2
"Memory"
Stack
Heap
Data
Code
"CPU"
Registers

What happens during a fork?

- A new process is created by making a copy of parent’s memory image
- The new process is added to the OS process list and scheduled
- Parent and child start execution just after fork(with different return values)
- Parent and child execute and modify the memory data independently

fork: Creating New Processes

`pid_t fork(void)`

- Creates a new "child" process that is *identical* to the calling "parent" process, including all state (memory, registers, etc.)
- Returns 0 to the child process
- Returns child's process ID (PID) to the parent process

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Child is almost identical to parent:

- Child gets an identical (but separate) copy of the parent's virtual address space
- Child has a different PID than the parent

`fork` is unique (and often confusing) because it is called once but returns "twice"

Understanding fork

Process X (parent)

```c
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = Y

```c
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
hello from parent
```

Process Y (child)

```c
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = 0

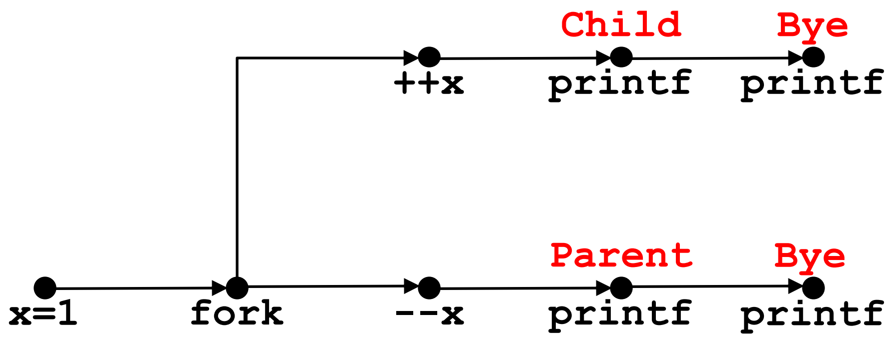
```c
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
hello from child
```

```
void fork1() {
    int x = 1;
    pid_t pid = fork();
    if (pid == 0)
       printf("Child has x = %d\n", ++x);
    else
       printf("Parent has x = %d\n", --x);
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

- Both processes continue/start execution after `fork`
  - Child starts at instruction after the call to `fork` (storing into `pid`)
- Can’t predict execution order of parent and child
- Both processes start with `x=1`
  - Subsequent changes to `x` are independent
- Shared open files: stdout is the same in both parent and child

```
void fork1() {
    int x = 1;
    pid_t pid = fork();
    if (pid == 0)
        printf("Child has x = %d\n", ++x);
    else
        printf("Parent has x = %d\n", --x);
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```



```
Parent has x= 0
Bye from process 7881 with x = 0
Child has x= 2
Bye from process 7882 with x = 2
```

Calling fork()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
  } else {
    // parent goes down this path (main)
    printf("hello, I am parent of %d (pid:%d)\n",
            rc, (int) getpid());
  }
  return 0;
}
```

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

Homework

```c
void nestedfork() {
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

Process termination and wait()

Process termination scenarios–By calling exit() (exit is called automatically when end of main is reached)

–OS terminates a misbehaving process

–Terminated process exists as a zombie

•When a parent calls wait(), zombie child is cleaned up or “reaped”

•wait() blocks in parent until child terminates(non-blocking ways to invoke wait exist)

•What if parent terminates before child? Init process adopts orphans and reaps them

Process Termination

- Voluntary : exit(status)
  - OS passes exit status to parent via wait(&status)
  - OS frees process resources

- Involuntary : kill(pid, signal)
  - Signal can be sent by another process or by OS
  - pid is for the process to be killed
  - signal a signal that the process needs to be killed
    - Examples : SIGTERM, SIGQUIT (ctrl+\), SIGINT (ctrl+c), SIGHUP

Zombies

- When a process terminates it becomes a zombie (or defunct process)
  - PCB in OS still exists even though program no longer executing
  - Why? So that the parent process can read the child’s exit status (through wait system call)
- When parent reads status,
  - zombie entries removed from OS… process reaped!
- Suppose parent does’nt read status
  - Zombie will continue to exist infinitely … a resource leak
  - These are typically found by a reaper process

Zombies

A terminated process still consumes system resources

- Various tables maintained by OS
- Called a “zombie” (a living corpse, half alive and half dead)

Reaping is performed by parent on terminated child

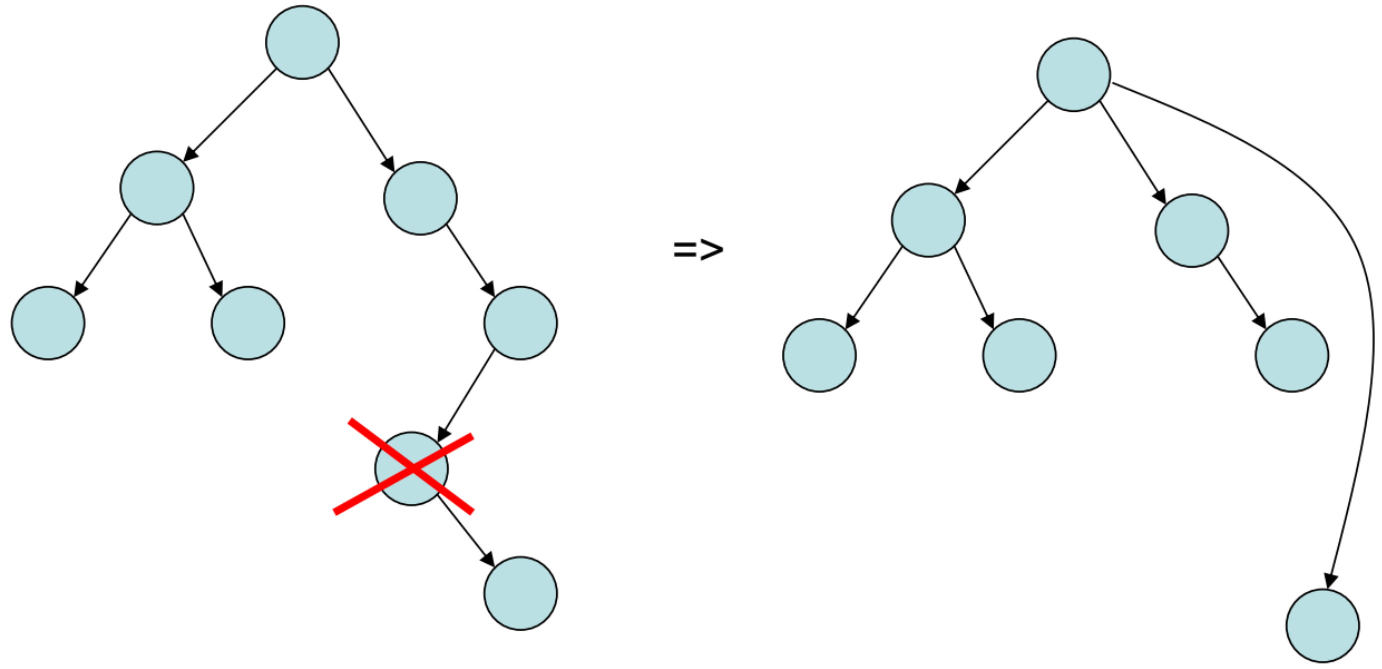
- Parent is given exit status information and kernel then deletes zombie child process

What if parent doesn’t reap?

- If any parent terminates without reaping a child, then the orphaned child will be reaped by `init` process (pid == 1)
  - **Note:** on recent Linux systems, `init` has been renamed `systemd`
- In long-running processes (e.g. shells, servers) we need *explicit* reaping

Orphans

- When a parent process terminates before its child
- Adopted by first process (/sbin/init)



- Unintentional orphans
  - When parent crashes
- Intentional orphans
  - Process becomes detached from user session and runs in the background
  - Called daemons, used to run background services
  - See nohup

`int wait(int *child_status_ptr)`

Suspends current process (*i.e.* the parent) until one of its children terminates

Return value is the PID of the child process that terminated

- *On successful return, the child process is reaped*
- argument child_*status_ptr* points to a location where wait() can store a status value.

If `child_status_ptr` = `NULL`, in which case wait() ignores the child's return status but reaps the child process

Note: If parent process has multiple children, `wait` will return when *any* of the children terminates

`waitpid` can be used to wait on a specific child process

Wait()

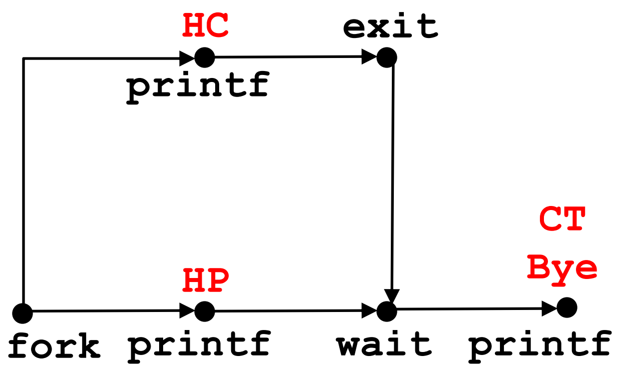
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

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());
  } 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> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (rc_wait:29267) (pid:29266)
prompt>
```

```
void fork_wait() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
        exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}
```



Feasible output:
HC
HP
CT
Bye

Infeasible output:
HP
CT
Bye
HC

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

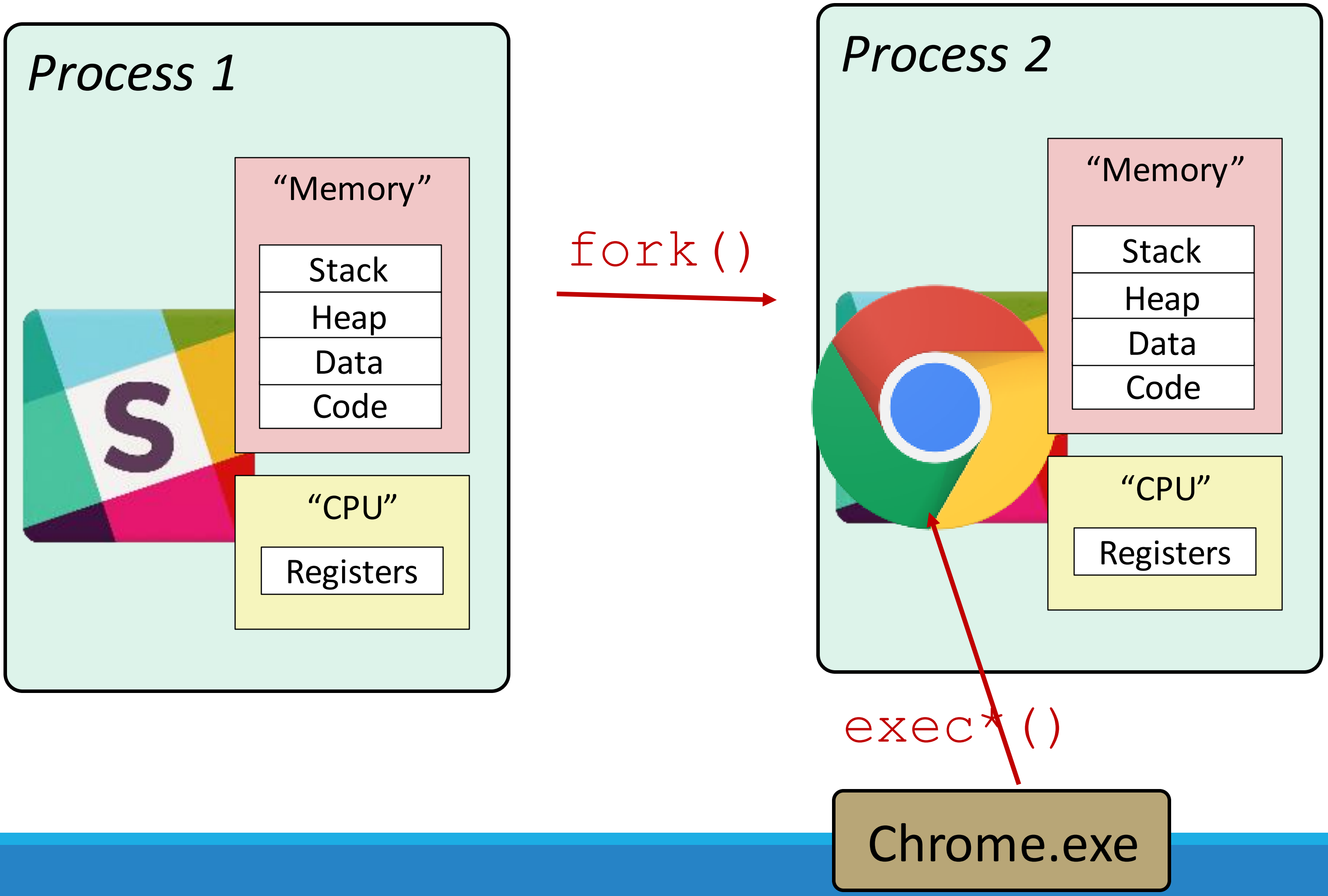
–So, a child can run a different program from parent

- 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

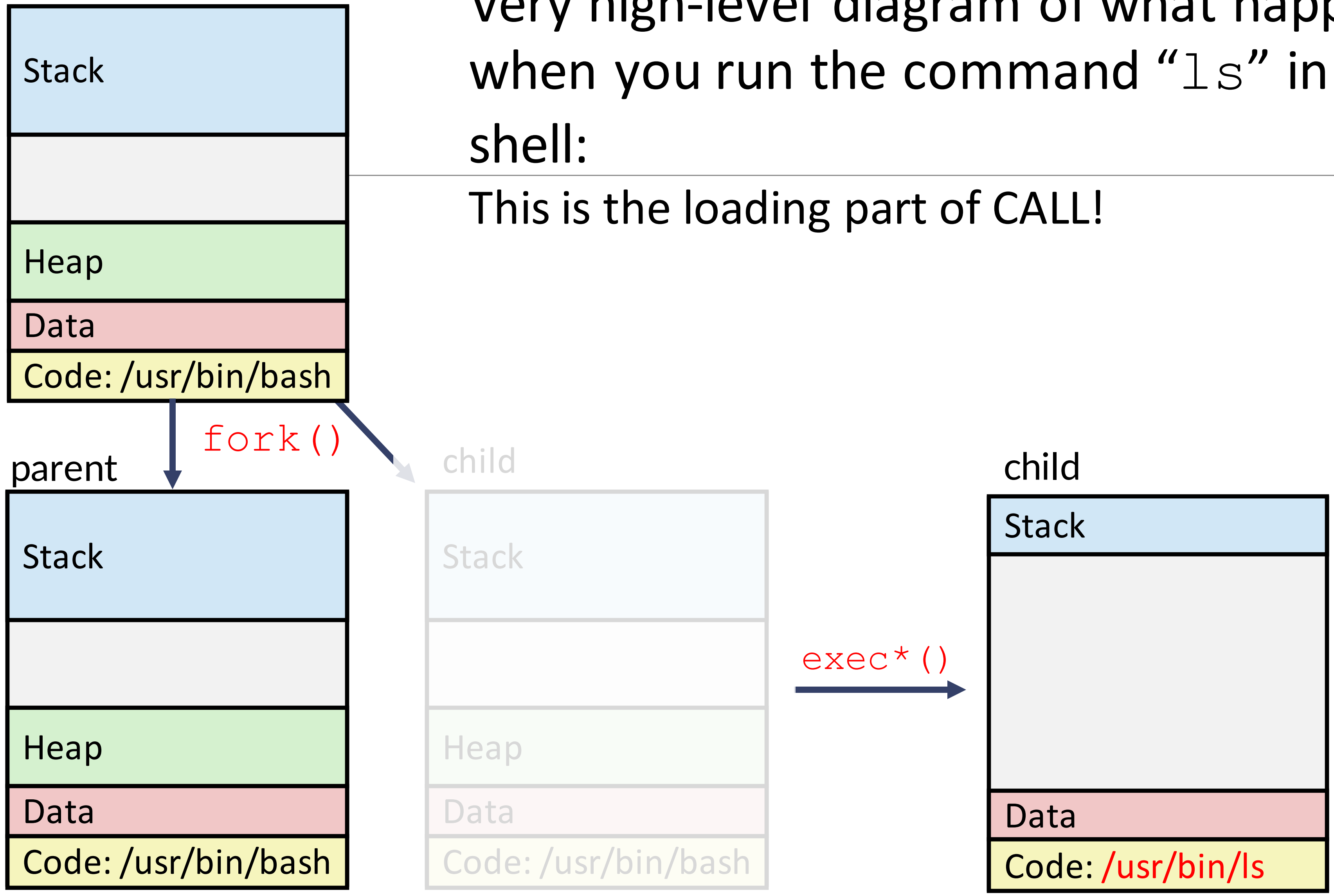
Why do we need fork() and exec()

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!

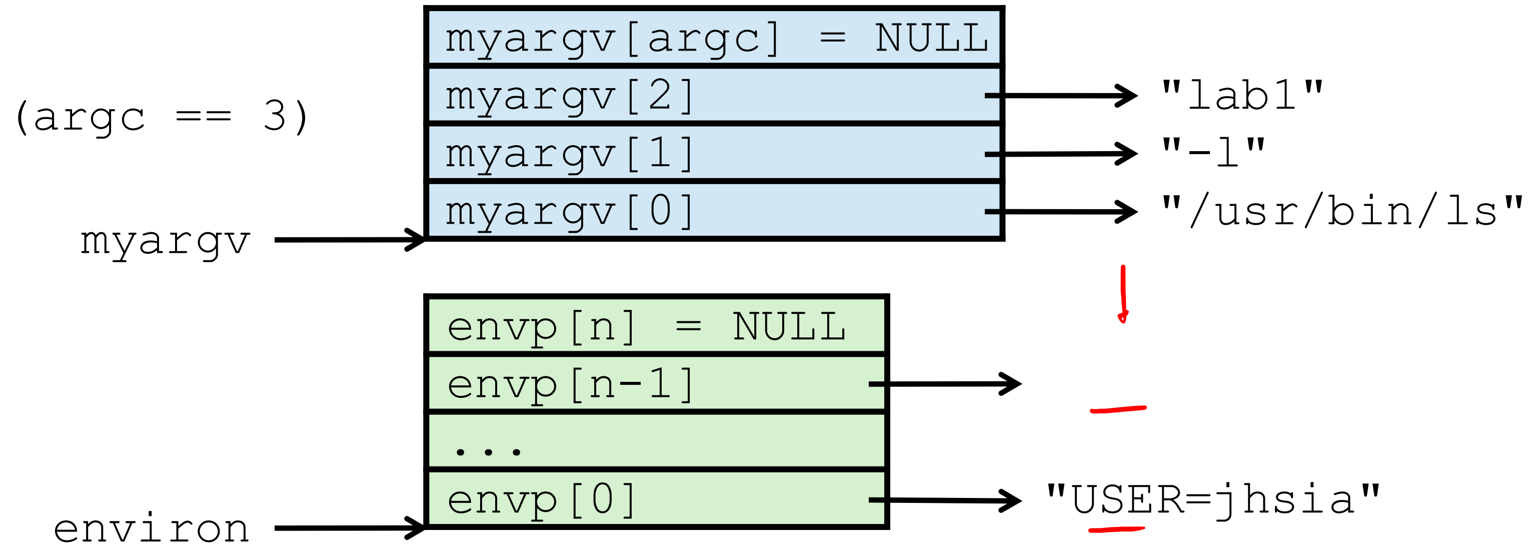


```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

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>
```

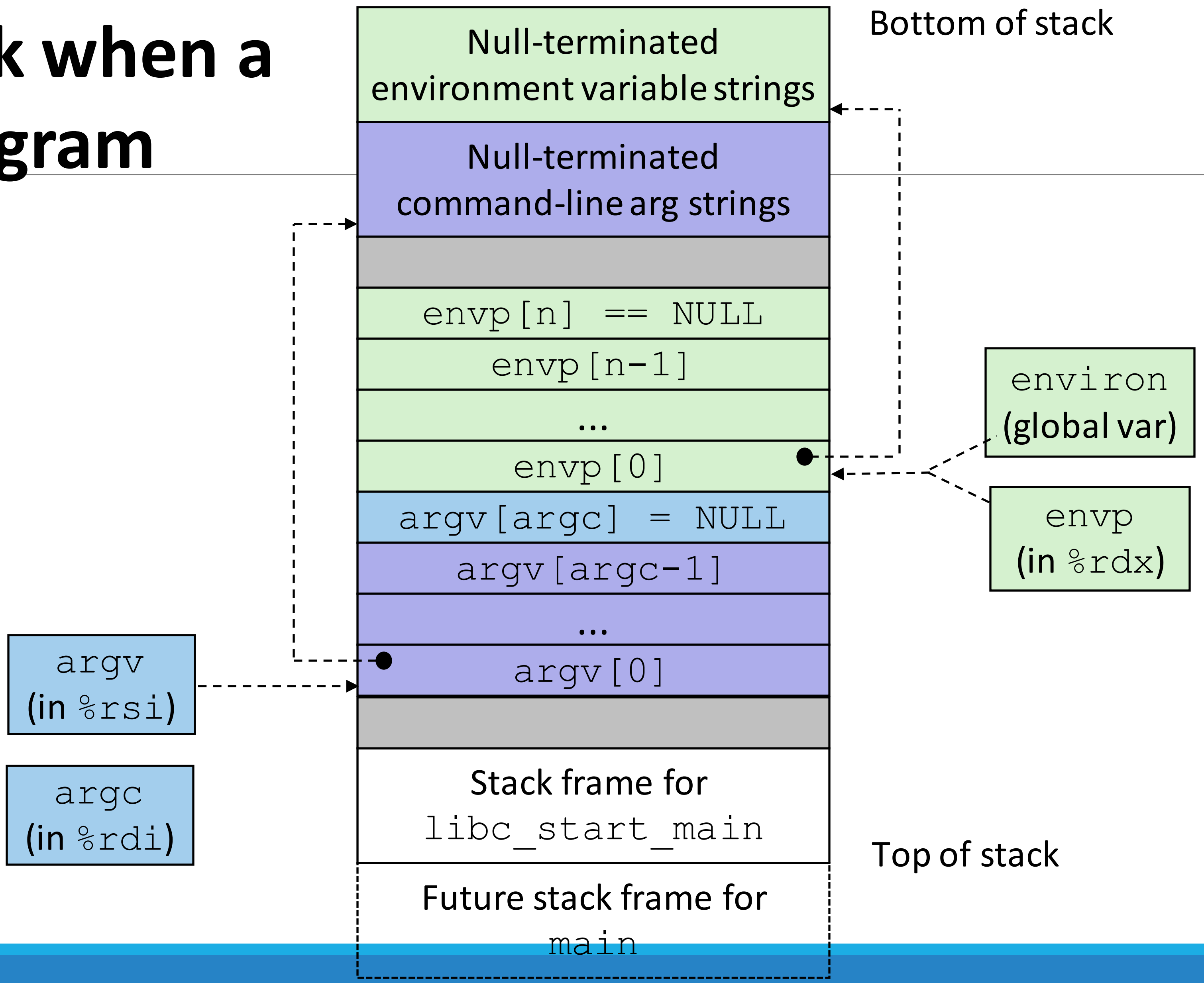
Execute `"/usr/bin/ls -l lab1"` in child process using current environment:



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

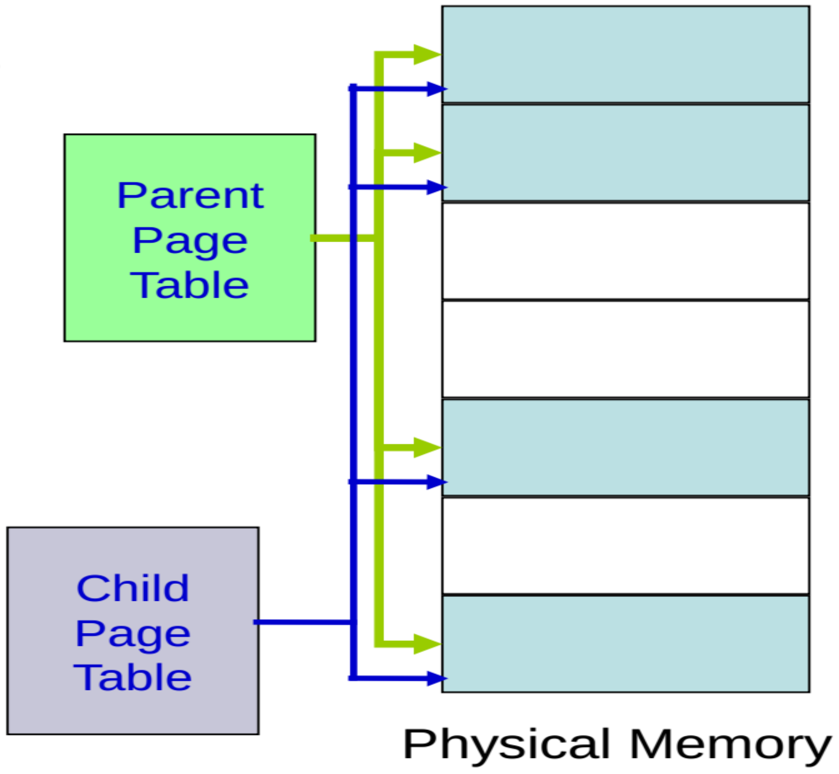
Run the `printenv` command in a Linux shell to see your own environment variables

Structure of the Stack when a new program starts



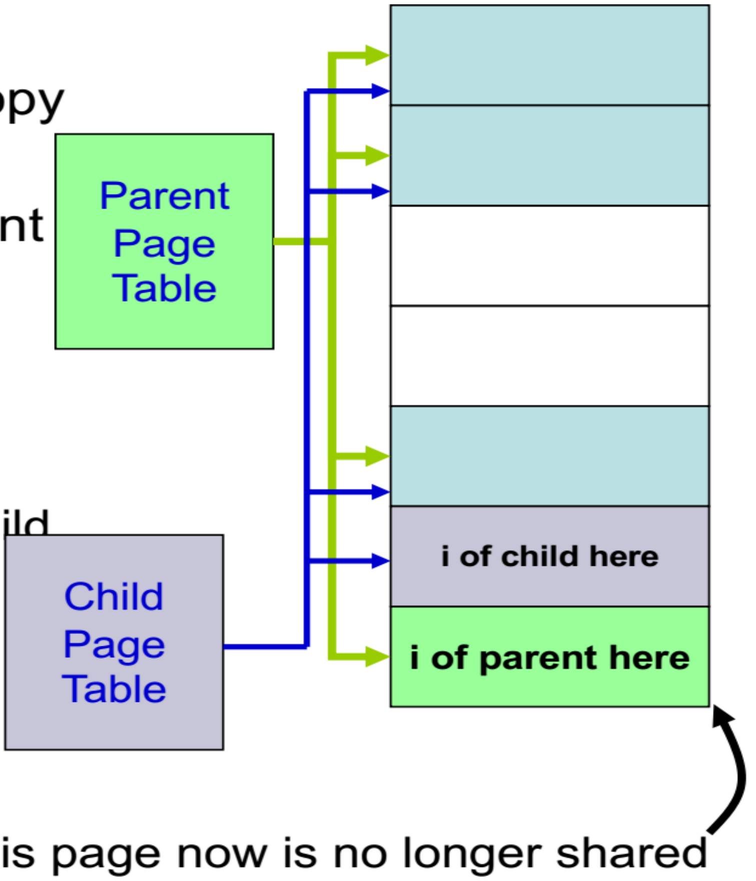
Virtual Addressing Advantage (easy to make copies of a process)

- 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



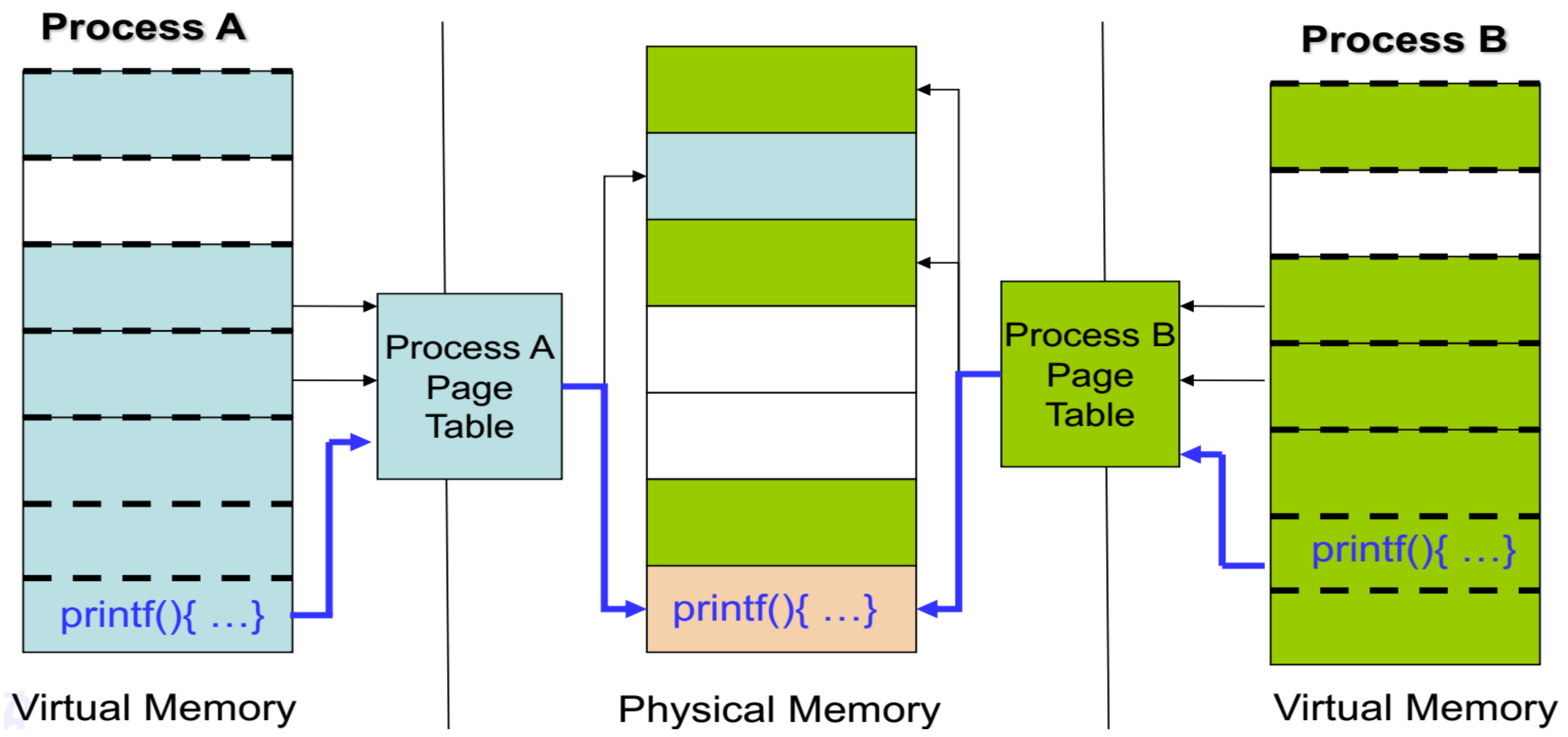
Copy on Write (CoW)

- 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