Systems Programming

Processes

Byoungyoung Lee

Seoul National University

byoungyoung@snu.ac.kr

https://lifeasageek.github.io

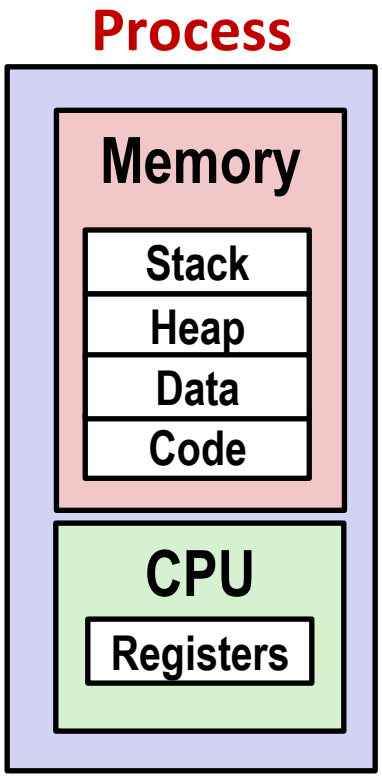
Lecture slides are prepared based on materials provided by CSAPP authors.

Today

- **Processes**
- **Process Control**

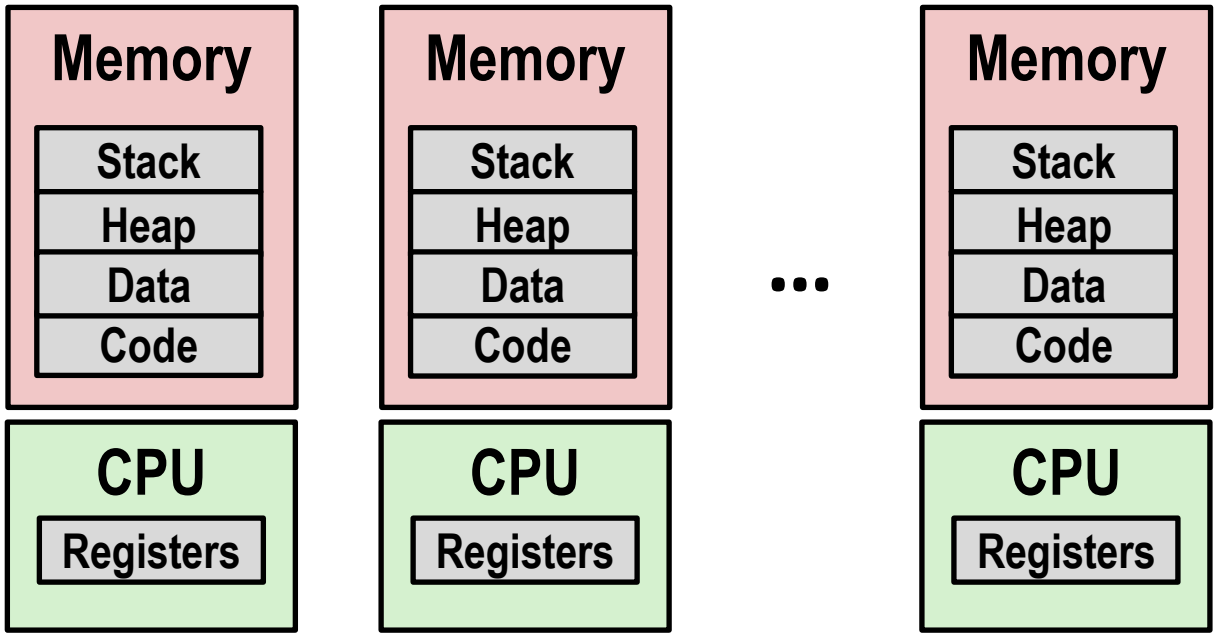
Processes

- **Definition: A *process* is an instance of a running program.**
  - Not the same as “program” or “processor”



- **Process provides two key abstractions:**
  - ***Logical control flow***
    - Each process seems to have exclusive use of the CPU
    - Provided by kernel mechanism called ***context switching***
  - ***Private address space***
    - Each process seems to have exclusive use of main memory.
    - Provided by kernel mechanism called ***virtual memory***

Multiprocessing: The Illusion



- **Computer runs many processes simultaneously**
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

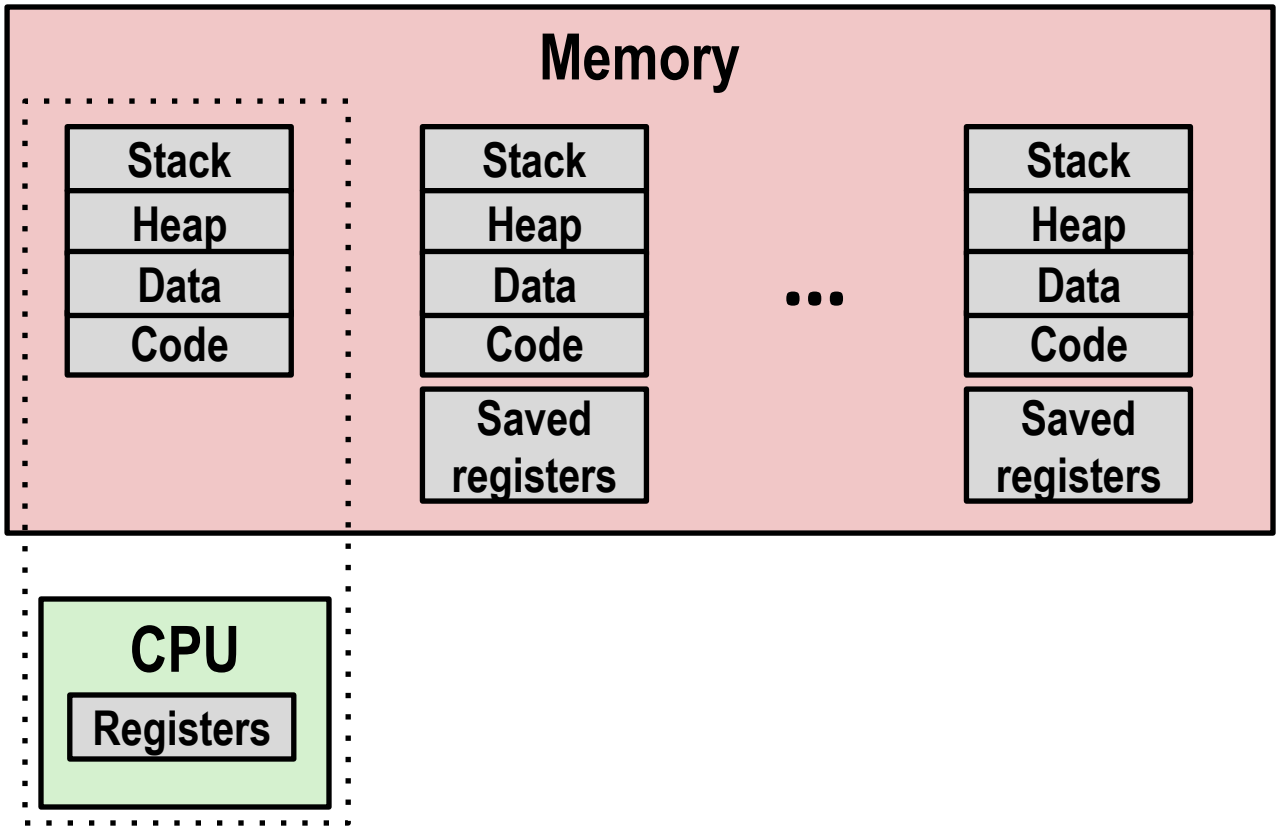
Multiprocessing Example

```
top - 00:51:39 up 121 days,  9:29,  2 users,  load average: 0.06, 0.02, 0.00
Tasks: 262 total,   1 running, 171 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 16343056 total,  5315620 free,  1694380 used,  9333056 buff/cache
KiB Swap:  4194300 total,  4191984 free,     2316 used. 14304932 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
13961 yoochan   20   0 1913772  72668  39552 S   1.0  0.4   1271:49 VBoxHeadless
 6278 blee      20   0   40628   3840   3120 R   0.3  0.0   0:00.07 top
    1 root      20   0  226284   9816   6596 S   0.0  0.1   7:24.52 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:01.89 kthreadd
    4 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 kworker/0:0H
    6 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 mm_percpu_wq
    7 root      20   0       0      0      0 S   0.0  0.0   0:01.48 ksoftirqd/0
    8 root      20   0       0      0      0 I   0.0  0.0  67:14.21 rcu_sched
    9 root      20   0       0      0      0 I   0.0  0.0   0:00.00 rcu_bh
   10 root      rt   0       0      0      0 S   0.0  0.0   0:00.13 migration/0
   11 root      rt   0       0      0      0 S   0.0  0.0   0:15.41 watchdog/0
```

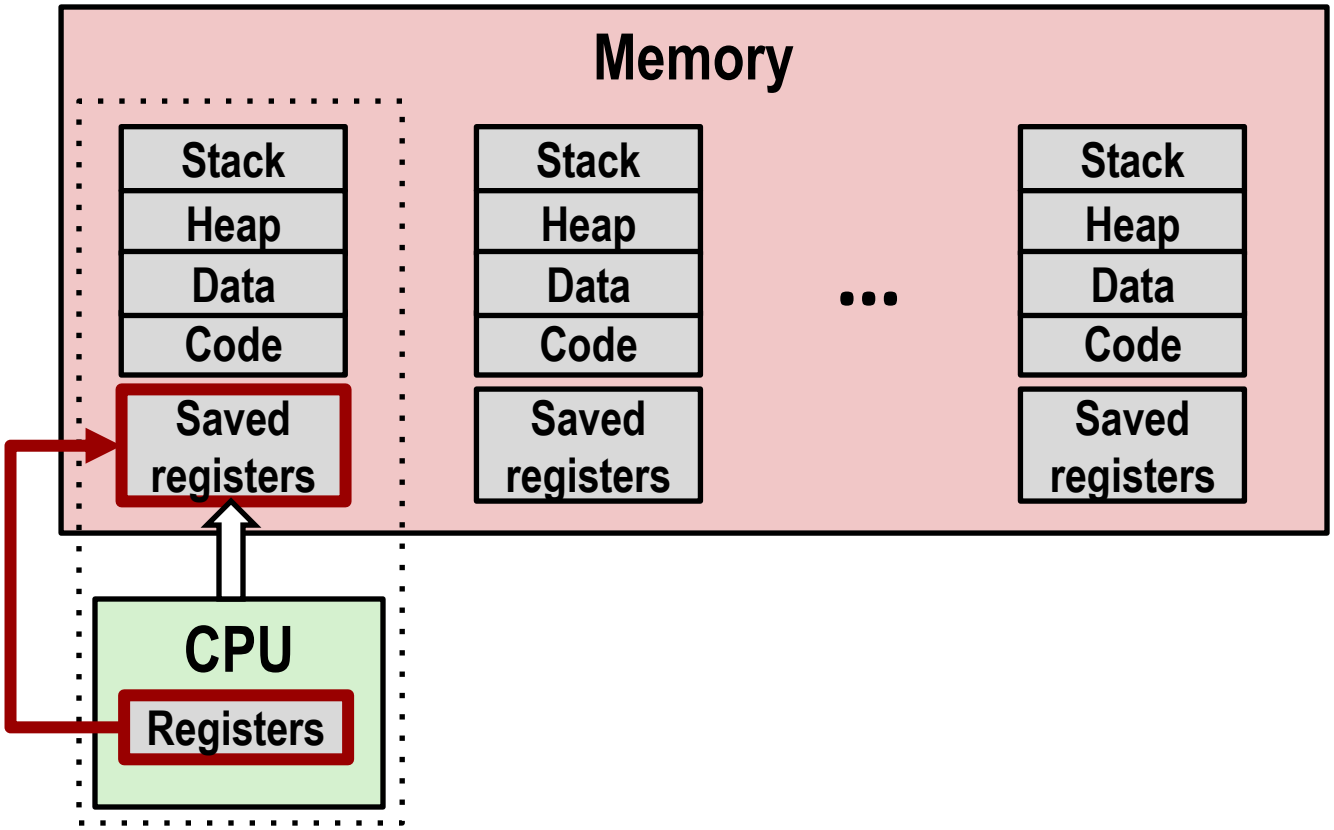
- **Running program “top” on Linux**
  - System has 262 processes
  - Identified by Process ID (PID)

Context Switch: CPU/Memory Perspectives



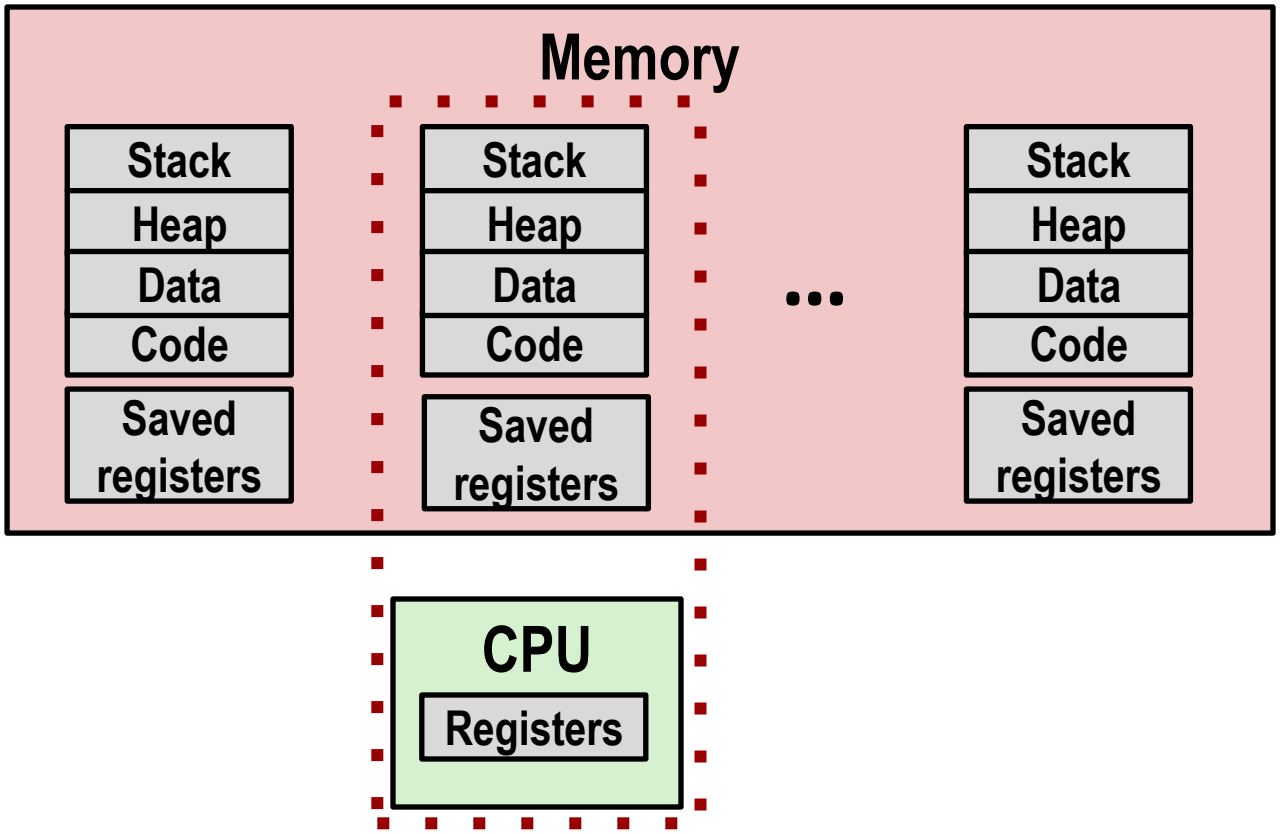
- **Single processor executes multiple processes concurrently**
  - Process executions are interleaved (multi-tasking)
  - Address spaces are managed by virtual memory system
  - Register values of non-executing processes are saved in memory

Context Switch: CPU/Memory Perspectives



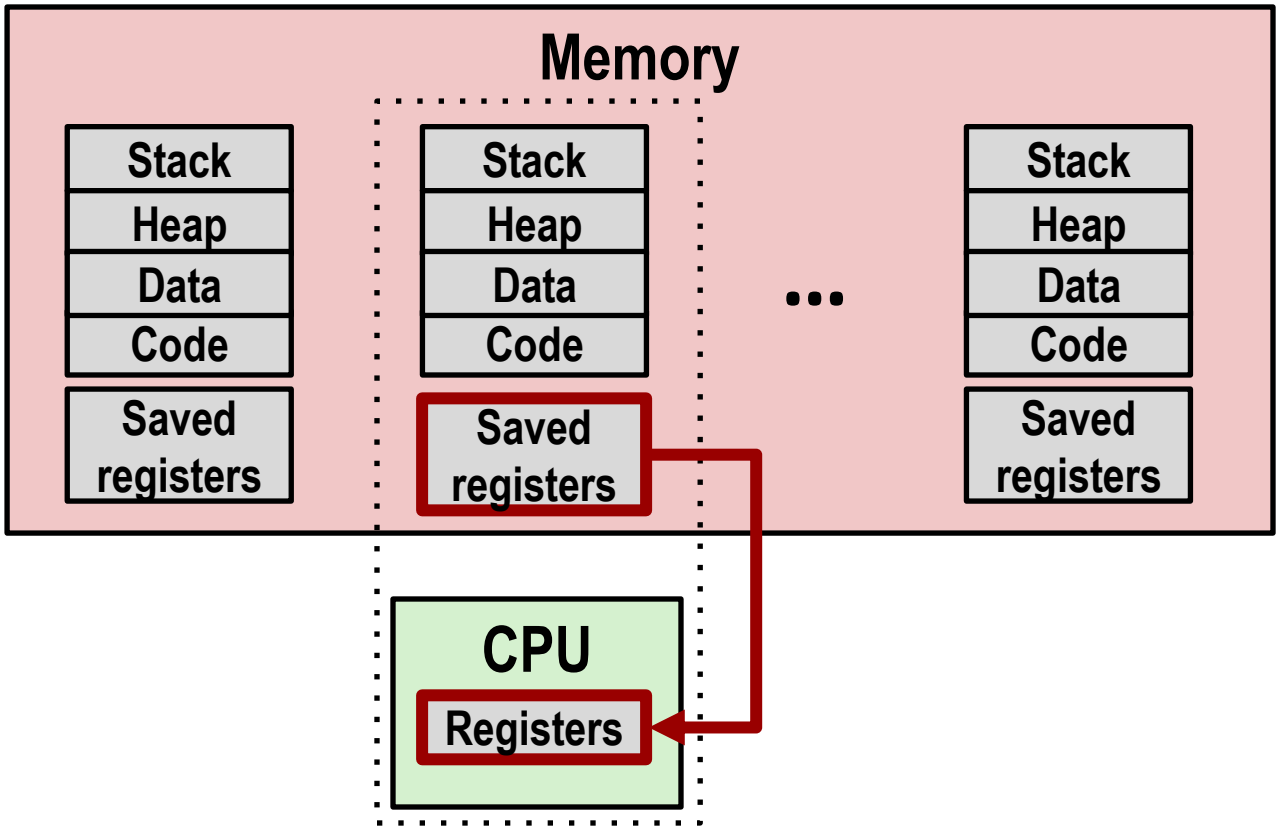
- **Context switch**
  - Step #1. Save current registers in memory

Context Switch: CPU/Memory Perspectives



- **Context switch**
  - Step #2. Schedule next process for execution

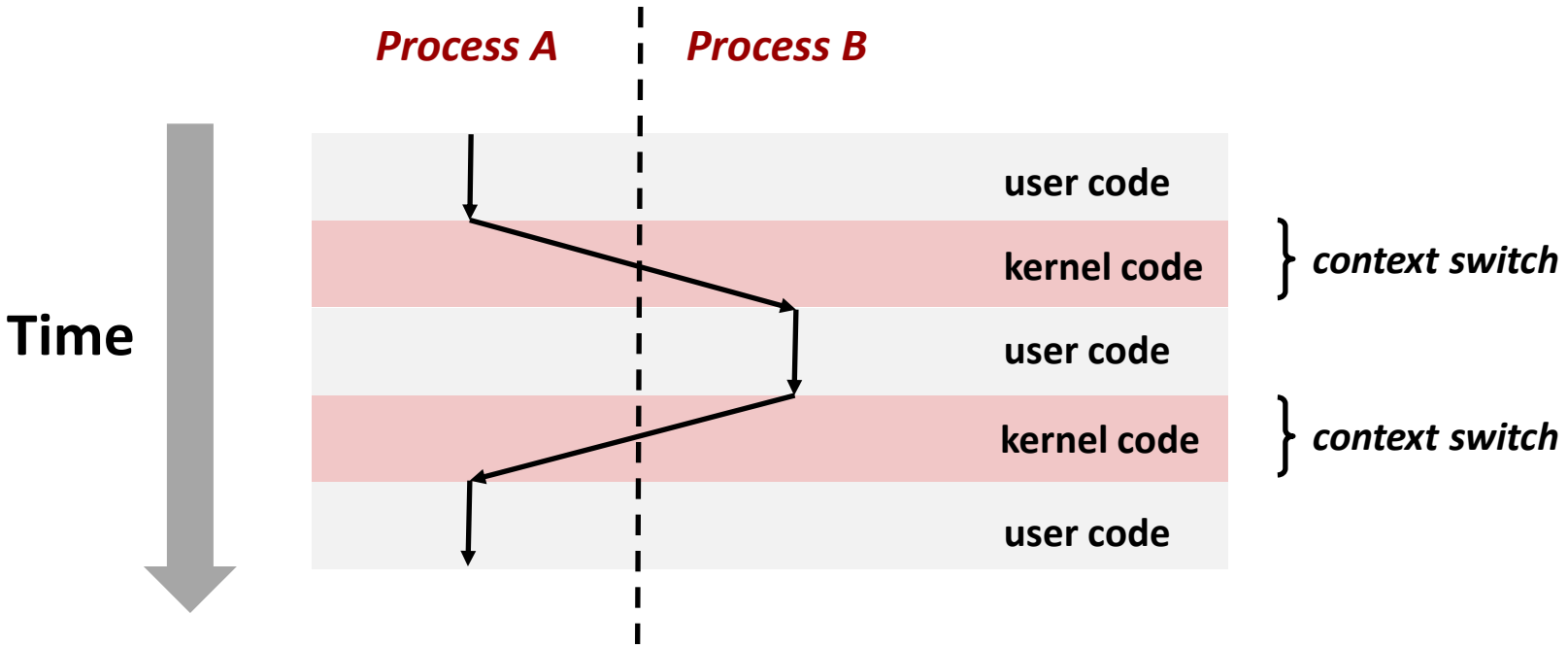
Context Switch: CPU/Memory Perspectives:



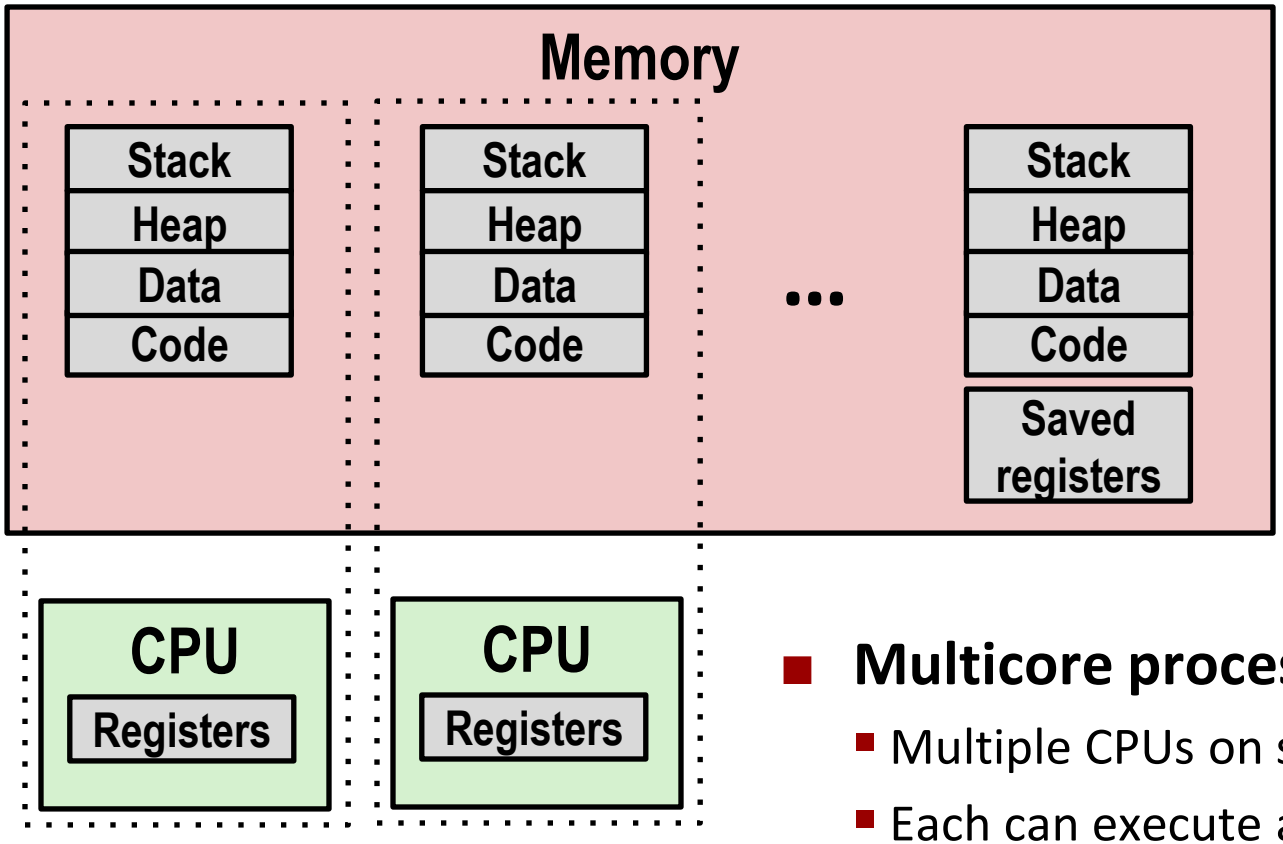
- **Context switch:**
  - Step #3. Load saved registers and switch address space

Context Switch: User/Kernel Perspectives

- Processes are managed by the *kernel*
- Control flow passes from one process to another via a *context switch*



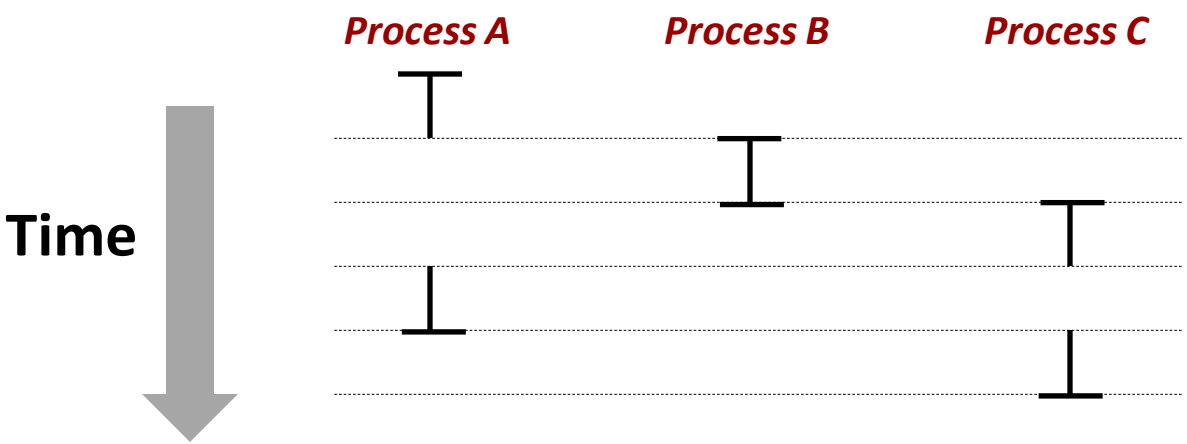
Multiprocessing with Multicore processors



- **Multicore processors**
  - Multiple CPUs on single chip
  - Each can execute a separate process
    - Scheduling of processors onto cores done by kernel

Concurrent Processes

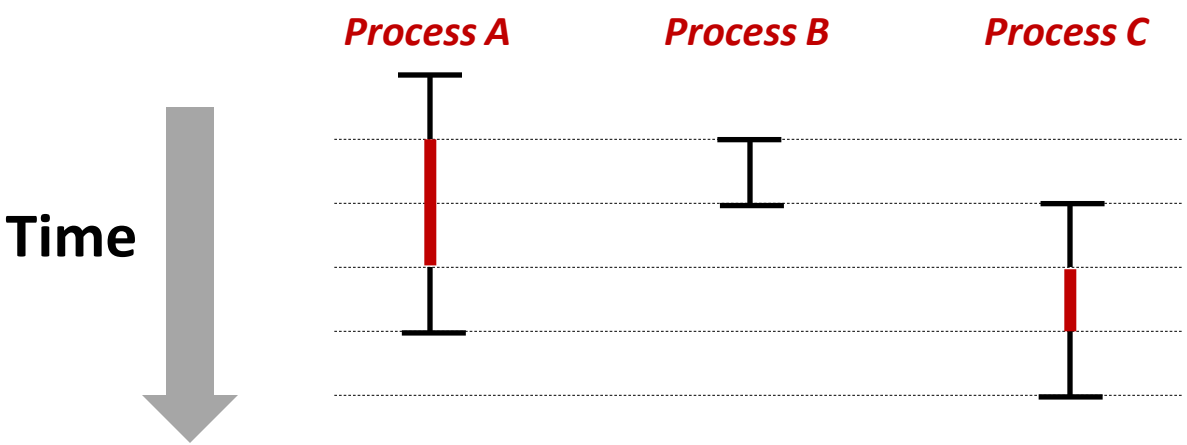
- **Each process is a logical control flow.**
- **Two processes *run concurrently* if their flows overlap in time**
- **Otherwise, they are *sequential***
- **Examples (running on single core):**
  - Concurrent: A & B, A & C
  - Sequential: B & C



User View of Concurrent Processes

- Control flows for concurrent processes are physically disjoint in time

- However, we can think of concurrent processes as running in parallel with each other

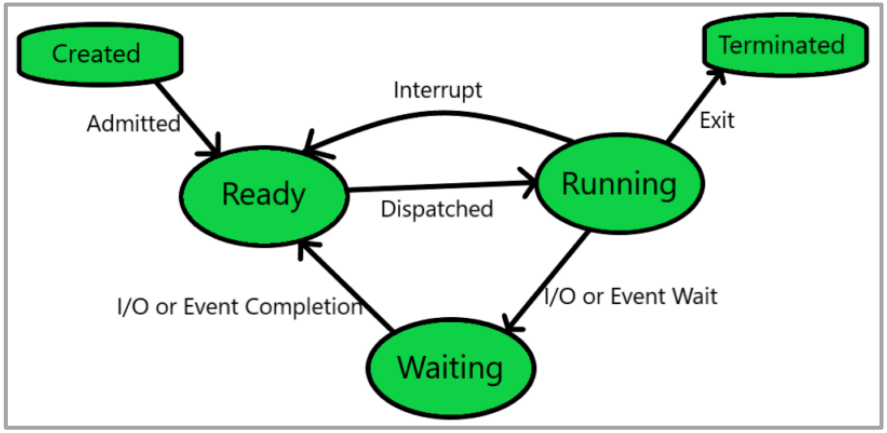


Today

- Processes
- **Process Control**

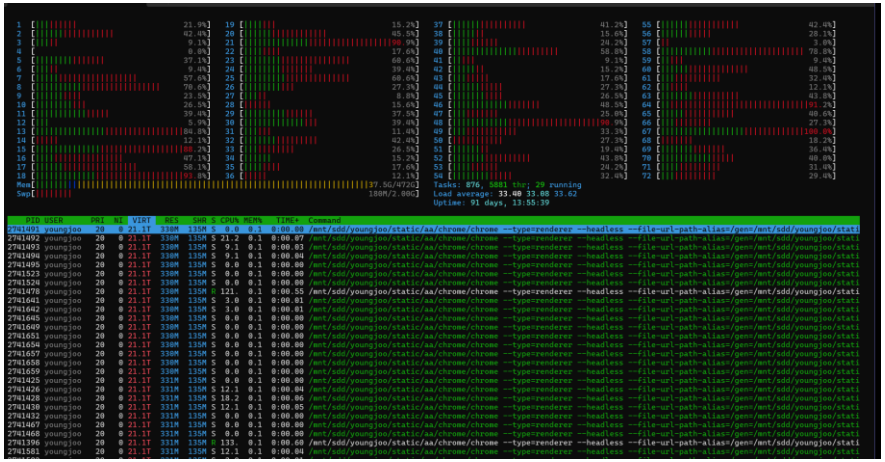
Lifecycle of a process

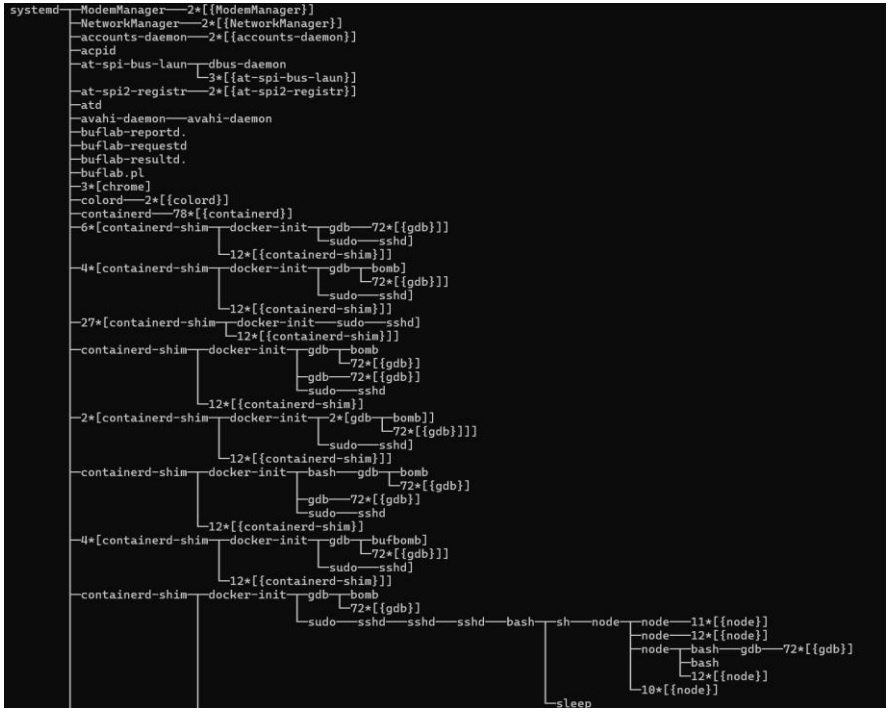
- **Ready**
  - Process is ready to be scheduled by the kernel
- **Running**
  - Process is executing
- **Waiting**
  - Waiting for I/O or events to be completed
- **Terminated**
  - Process is stopped permanently



Obtaining Process IDs

- `pid_t getpid(void)`
  - Returns PID of current process
- `pid_t getppid(void)`
  - Returns PID of parent process





Terminating Processes

- **Process becomes terminated for one of three reasons:**
  - Receiving a signal whose default action is to terminate (next lecture)
  - Returning from the **`main`** routine
  - Calling the **`exit`** function

- **`void exit(int status)`**
  - Terminates with an *exit status* of **`status`**
  - Convention: normal return status is 0, nonzero on error

- **`exit` is called once but never returns.**

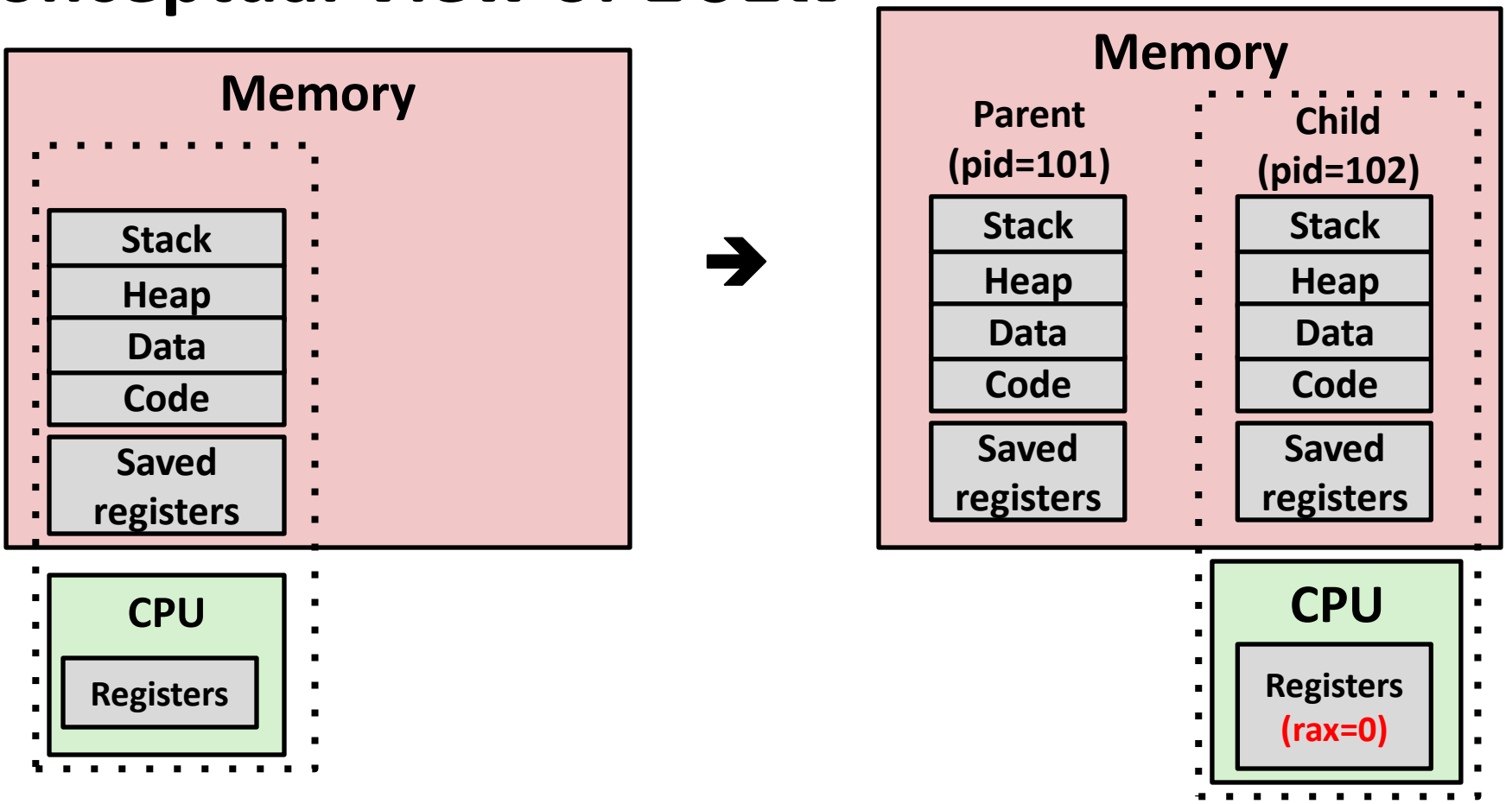
Creating Processes

- ***Parent process*** **creates a new running** ***child process*** **by calling `fork`**

- `int fork(void)`
  - Returns
    - 0 to the child process
    - child's PID to parent process
  - Child is *almost* identical to parent:
    - Childs get an identical copy of the parent's virtual address space.
    - Child gets identical copies of the parent's open file descriptors
    - Child has a different PID than the parent

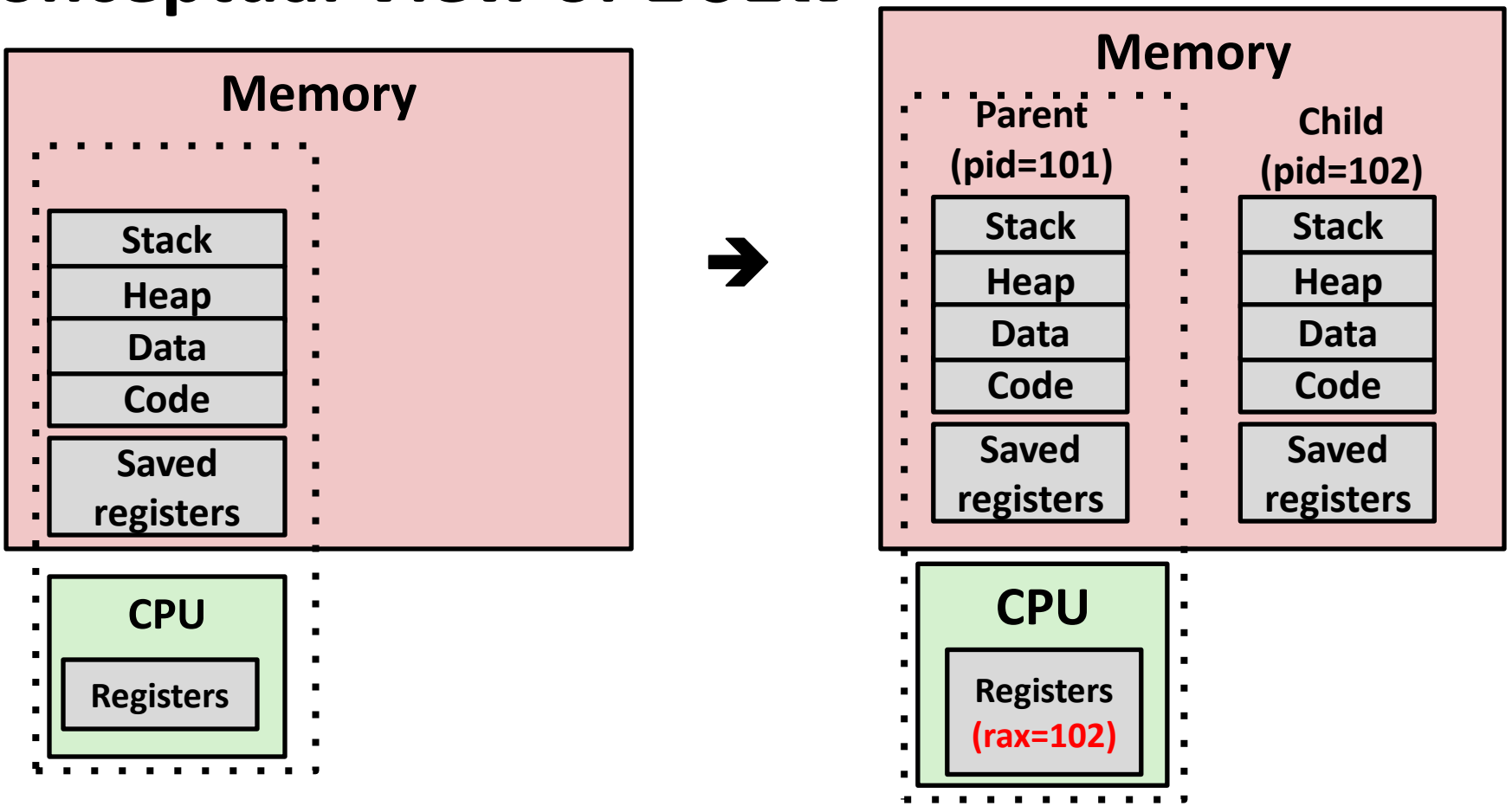
- **`fork` is interesting (and often confusing) because it is called** ***once*** **but returns** ***twice***

Conceptual View of `fork`



- **Make complete copy of execution state**
  - Designate one as parent and one as child
  - Resume execution of either parent or child

Conceptual View of `fork`



- **Make complete copy of execution state**
  - Designate one as parent and one as child
  - Resume execution of either parent or child

The `fork` Function Revisited

- **Virtual memory is the key for `fork` to provide private address space for each process.**
- **To create virtual address for new process:**
  - Create exact copies of current page tables.
- **On return of fork(), two processes have the same virtual memory.**

- **Copy-on-write (COW)**
  - Should the kernel keep individual physical copies of memory of these two identical processes?
  - COW: Let's copy only when required (i.e., written)
  - To implement COW, all pages in both processes are flagged as read-only
  - Any write creates new pages using copy-on-write (COW).

`fork` Example

```
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    } else {         /* Parent */
        printf("parent: x=%d\n", --x);
        return 0;
    }
}
```

```
$ ./fork
Q. what would be printed??
```

- **Call once, return twice**
- **Concurrent execution**
  - Can't predict execution order of parent and child
- **Duplicate but separate address space**
  - `x` has a value of 1 when fork returns in parent and child
  - Subsequent changes to `x` are independent
- **Shared open files**
  - `stdout` is the same in both parent and child

Modeling `fork` with Process Graphs

- **A *process graph* is a useful tool for capturing the partial ordering of statements in a concurrent program:**
  - Each vertex is the execution of a statement
  - a -> b means `a` happens before b
  - Each graph begins with a vertex with no in-edges

- **Any *topological sort (ordering)* of the graph corresponds to a feasible total ordering.**
  - Total ordering of vertices where all edges point from left to right
  - This makes easier to understand
    - all feasible task ordering
    - non-feasible task ordering

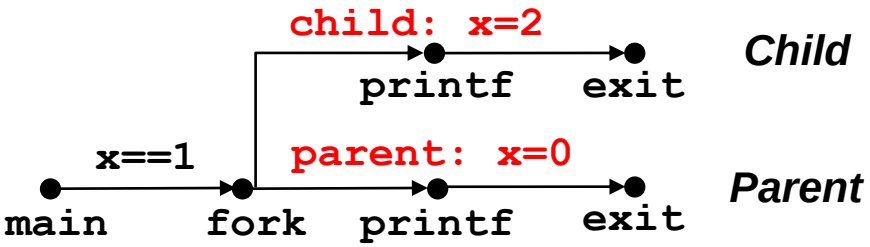
Process Graph Example

```
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

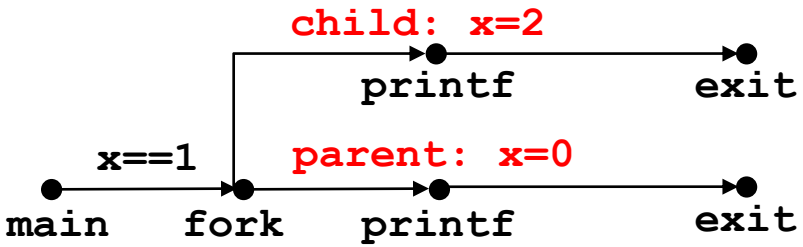
    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```

fork.c

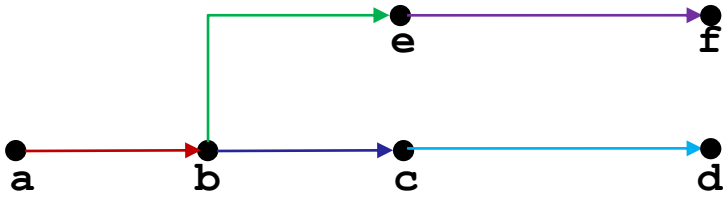


Interpreting Process Graphs

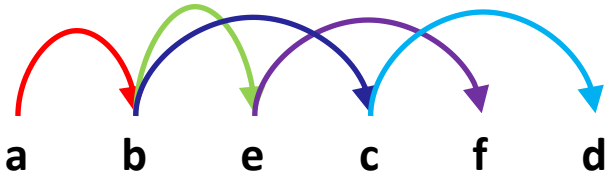
- Original graph:



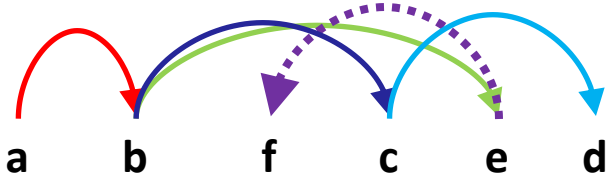
- Re-labled graph:



Feasible total ordering:

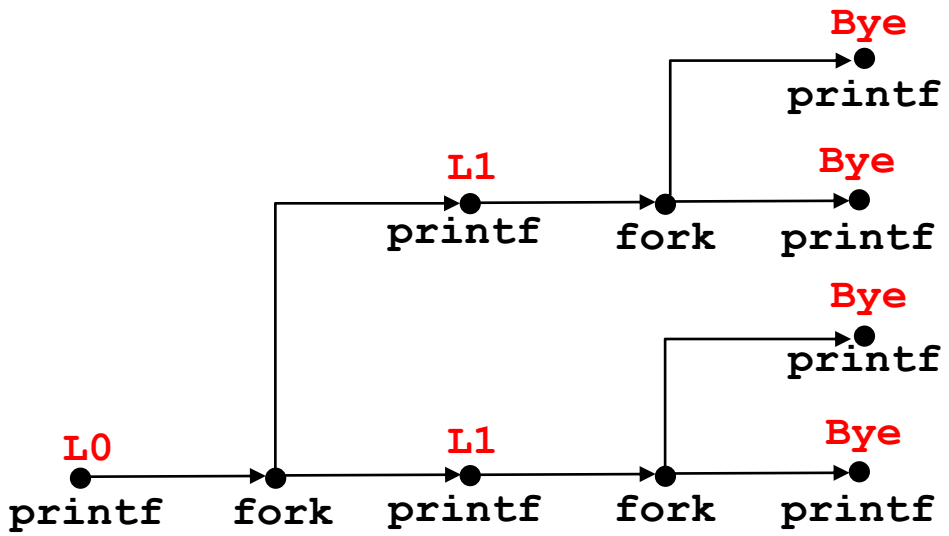


Q. Feasible or Infeasible?



`fork` Example: Two consecutive `fork`s

```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```
forks.c



Feasible output:

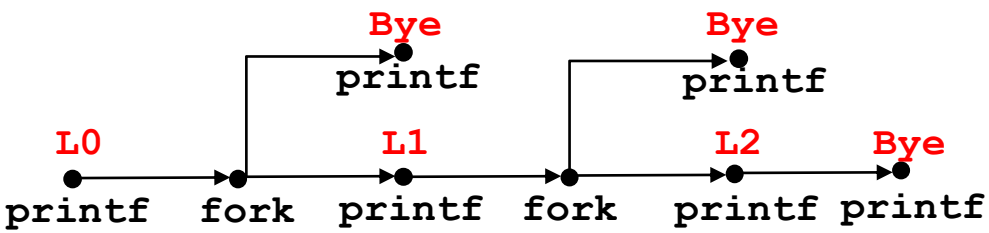
```
L0
L1
Bye
Bye
L1
Bye
Bye
```

Infeasible output:

```
L0
Bye
L1
Bye
L1
Bye
Bye
```

`fork` Example: Nested `forks` in parent

```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```
forks.c



Feasible or Infeasible?

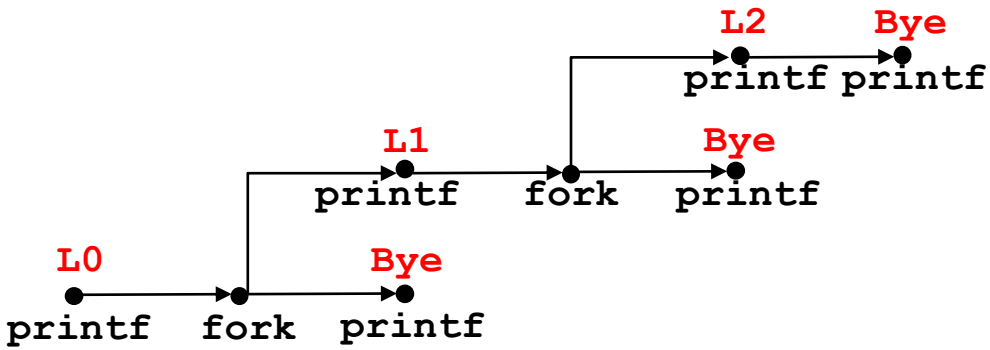
L0
Bye
L1
Bye
Bye
L2

Feasible or Infeasible?

L0
L1
Bye
Bye
L2
Bye

`fork` Example: Nested `forks` in children

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```
forks.c



Feasible or Infeasible?

L0
Bye
L1
Bye
Bye
L2

Feasible or Infeasible?

L0
Bye
L1
L2
Bye
Bye

Reaping Child Processes

- **Idea**
  - When process terminates, it still consumes system resources
    - Examples: Exit status, various OS tables
  - Called a “zombie”
    - Living corpse, half alive and half dead
- **Reaping**
  - Performed by parent on terminated child (using `wait` or `waitpid`)
  - Parent is given exit status information
  - Kernel then deletes zombie child process
- **What if parent doesn’t reap?**
  - If any parent terminates without reaping a child, then the orphaned child should be reaped by **`init`** process (pid == 1)
  - So, only need explicit reaping in long-running processes
    - e.g., shells and servers



Zombie Example

```
void fork7() {
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n", getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

forks.c

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640

linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps

linux> kill 6639
[1]    Terminated

linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

- **`ps`** shows child process as "defunct" (i.e., a zombie)
- Killing parent allows child to be reaped by **`init`**

Non-terminating Child Example

```
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}
```

```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676

linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps

linux> kill 6676

linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

- Child process still active even though parent has terminated
- Must kill child explicitly, or else will keep running indefinitely

`wait`: Synchronizing with Children

- **Parent reaps a child by calling the `wait` syscall**

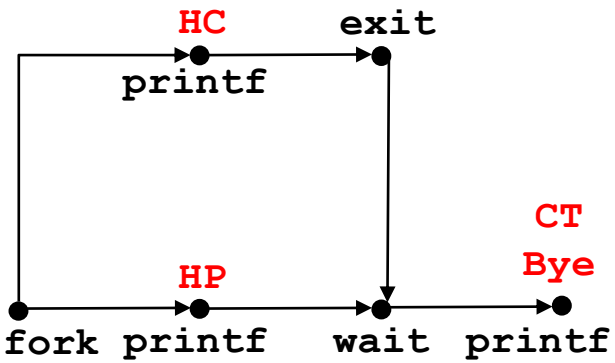
- **`int wait(int *child_status)`**
  - Suspends current process until one of its children terminates
  - Return value is the **`pid`** of the child process that terminated
  - If **`child_status != NULL`**, then it will be set to a value indicating reason the child terminated and the exit status:

`wait`: Synchronizing with Children

```
void fork9() {
    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");
}
```

forks.c



Feasible output(s):

HC	HP
HP	HC
CT	CT
Bye	Bye

Infeasible output:

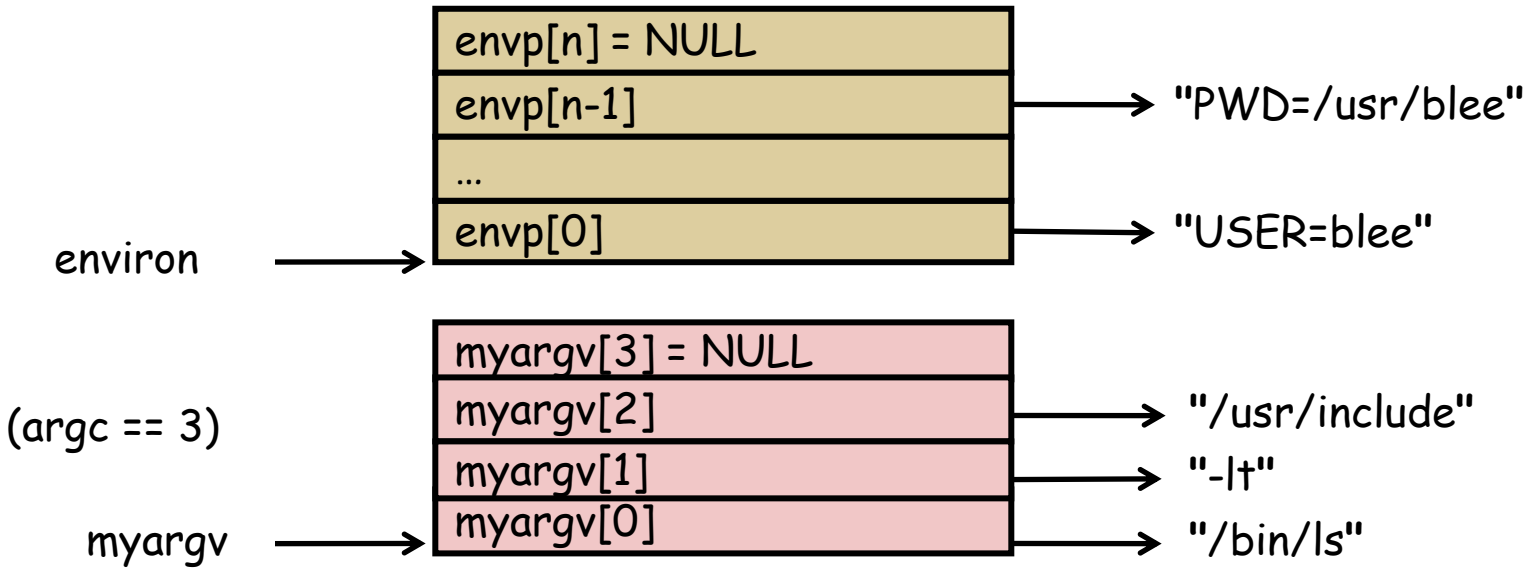
HP
CT
Bye
HC

`execve`: Loading and Running Programs

- **`int execve(char *filename, char *argv[], char *envp[])`**
  - Load and run the executable in the current process:
    - Executable file **`filename`**
    - With argument list **`argv`**
      - By convention **`argv[0]==filename`**
    - Environment variable list **`envp`**
      - "name=value" strings (e.g., `USER=blee`)
  - Overwrite code, data, and stack
    - Retains PID, open files and signal context
  - Called once and never returns
    - …except if there is an error to load/run a program

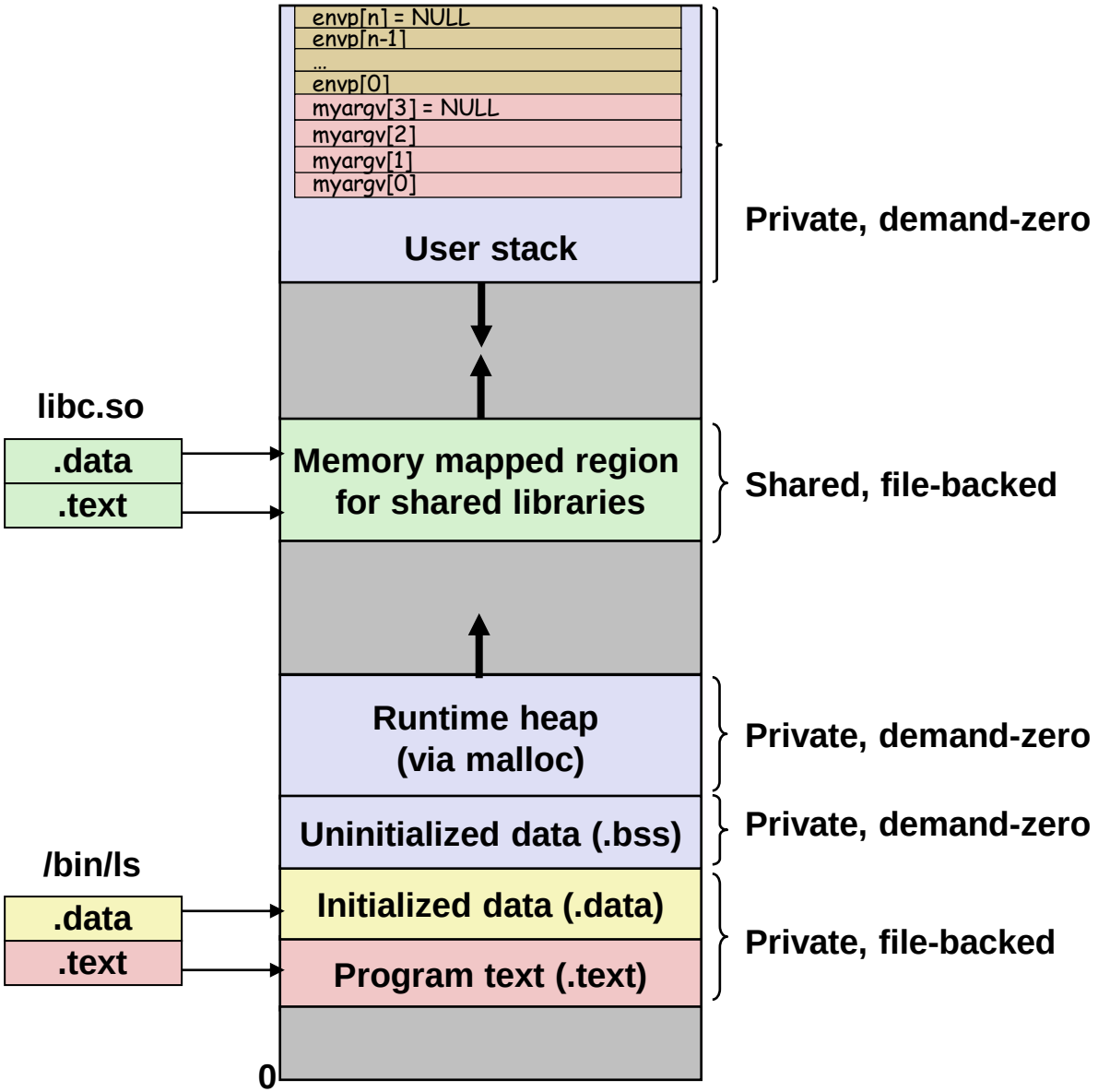
execve Example

- **Execute** `"/bin/ls -lt /usr/include"` **in child process:**



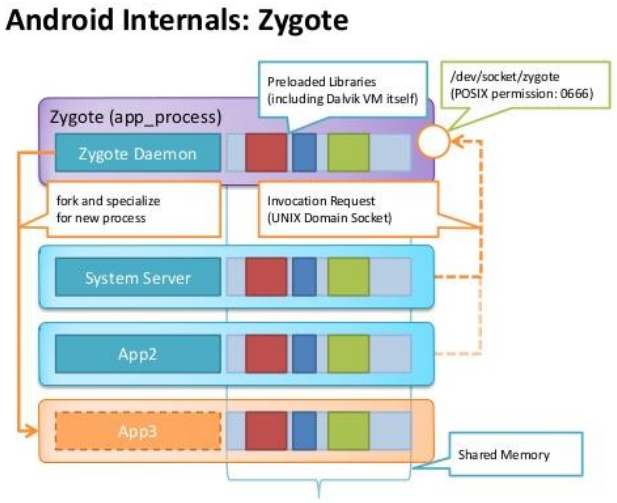

```
if ((pid = fork()) == 0) {    /* Child runs program */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: Command not found.\n", myargv[0]);
        exit(1);
    }
}
```

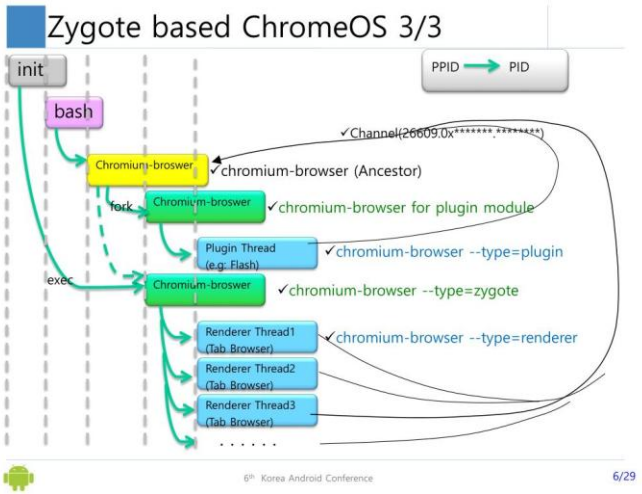
The `execve` Function Revisited

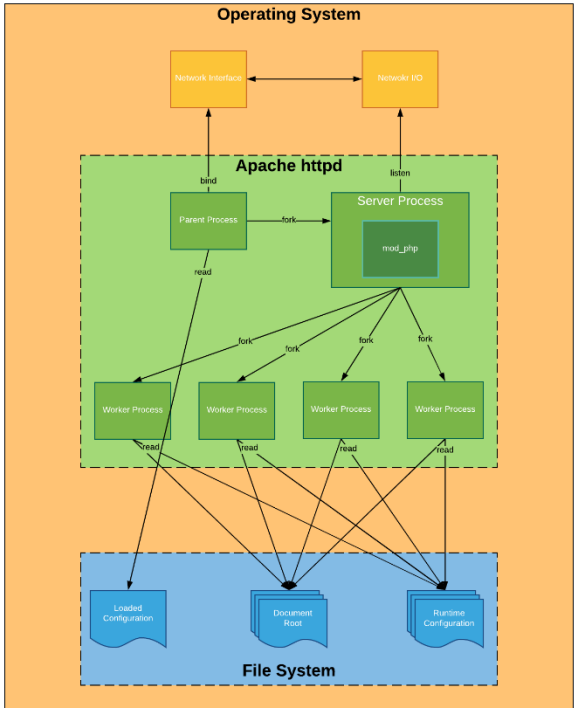


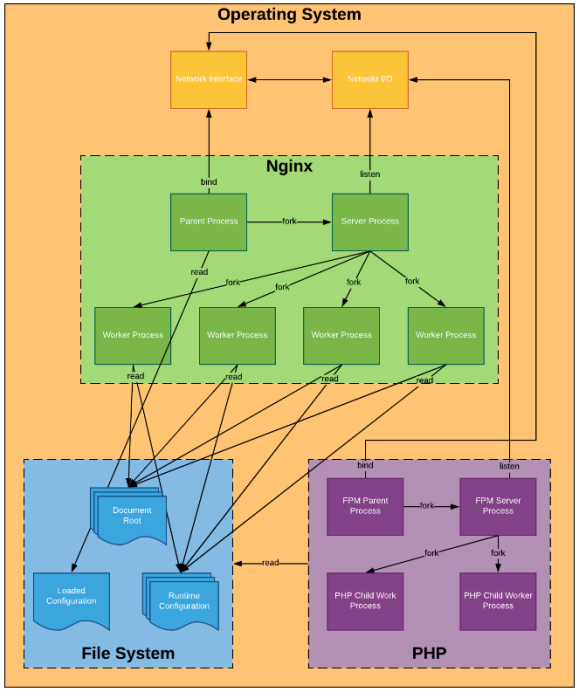
- **To load and run a new program `/bin/ls` in the current process using `execve`:**
- **Setup memory layout**
  - Initialize a page table according to the program headers
- **Set PC to entry point in `.text`**

Fork in Real-world



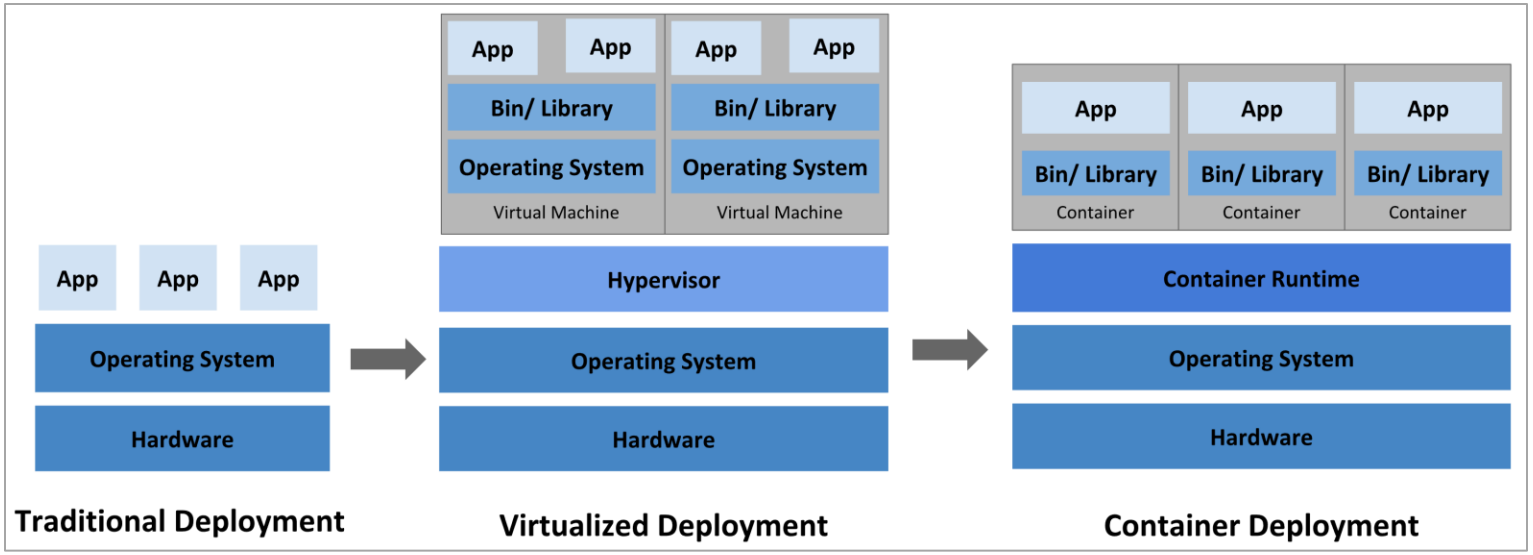






Process in Real-world

- Do you run your process on your machine?
  - No! We run everything on cloud!
  - Your process runs within a container!



https://kubernetes.io/docs/concepts/overview/

Summary

- **Processes**
  - At any given time, system has multiple active processes
  - Only one can execute at a time on any single core
  - Each process appears to have total control of processor + private memory space
- **Spawning processes**
  - Call fork
- **Process completion**
  - Call exit
- **Reaping and waiting for processes**
  - Call wait
- **Loading and running programs**
  - Call execve (or variant)

