CS304 Operating Systems

DR GAYATHRI ANANTHANARAYANAN

gayathri@iitdh.ac.in

Materials in these slides have been borrowed from textbooks and existing operating systems courses

Processes and Process Management

What is a **Process**?

How are **processes represented by OS**?

How are **multiple concurrent processes are managed by OS**? [Particularly when multiple physical processes share a **single physical platform**]

Process Definition

Instance of an executing program [Synonymous with "task" or "Job"]

Recall : OS manages H/W on behalf of applications

Application = Program on a disk/flash memory/cloud.. **[Static Entity]**

Process = State of the program when executing [Gets loaded in memory & starts execution] **[Active Entity]**

Same program launched more than once => Multiple processes will get created => But each one of them may be in a different state {??}

Program	Process
code + static and global data	Dynamic instantiation of code + data + heap + stack + process state
One program can create several processes	A process is unique isolated entity

Process Management

When you run an exe file, the OS creates a process = a running program

• OS timeshares CPU across multiple processes: virtualizes CPU

• OS has a CPU scheduler that picks one of the many active processes to execute on a CPU

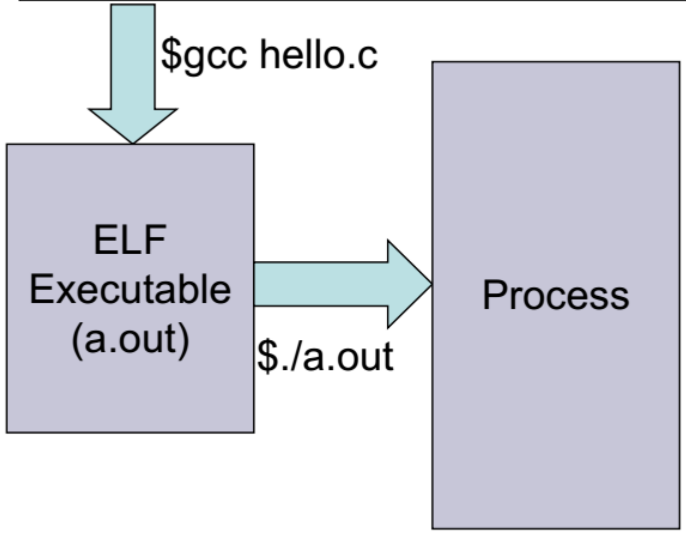
– Policy: which process to run

– Mechanism: how to “context switch” between processes

What does a Process look like?

```c
#include <stdio.h>

int main(){
  char str[] = "Hello World\n";
  printf("%s", str);
}
```



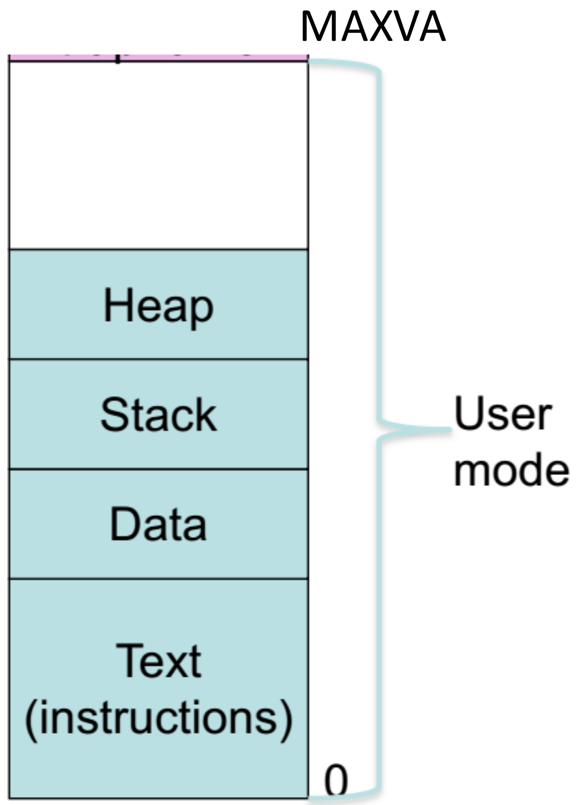
- **Process**
  - A program in execution
  - Most important abstraction in an OS
  - Comprises of A unique identifier (PID)
    - Code
    - Data

      from ELF
    - Stack
    - Heap

      In the user space of process
    - State in the OS
    - Kernel stack

      In the kernel space
  - State contains : registers, list of open files, related processes, etc.

CPU context – Program counter, current operands, Stack pointer



Address Space

Address Space

Virtual Address Map

– All memory a process can address

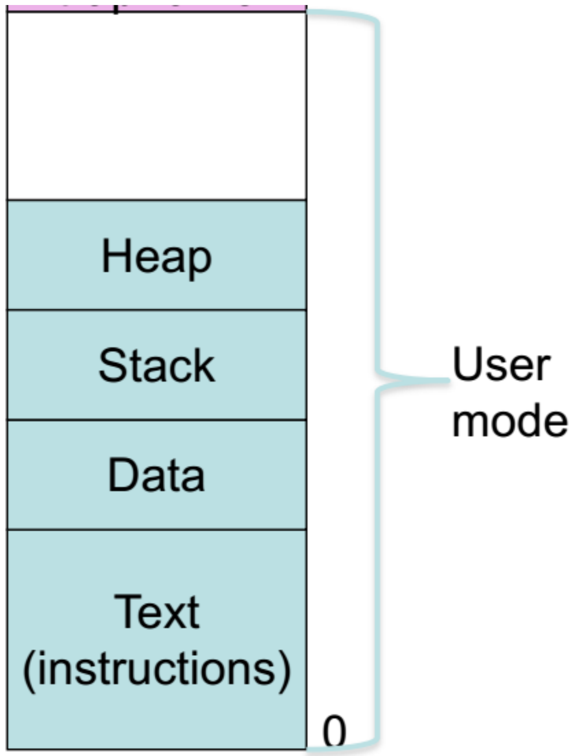
– Large contiguous array of addresses from 0 to MAXVA

Each process has a different address space

This is achieved by the use of virtual memory

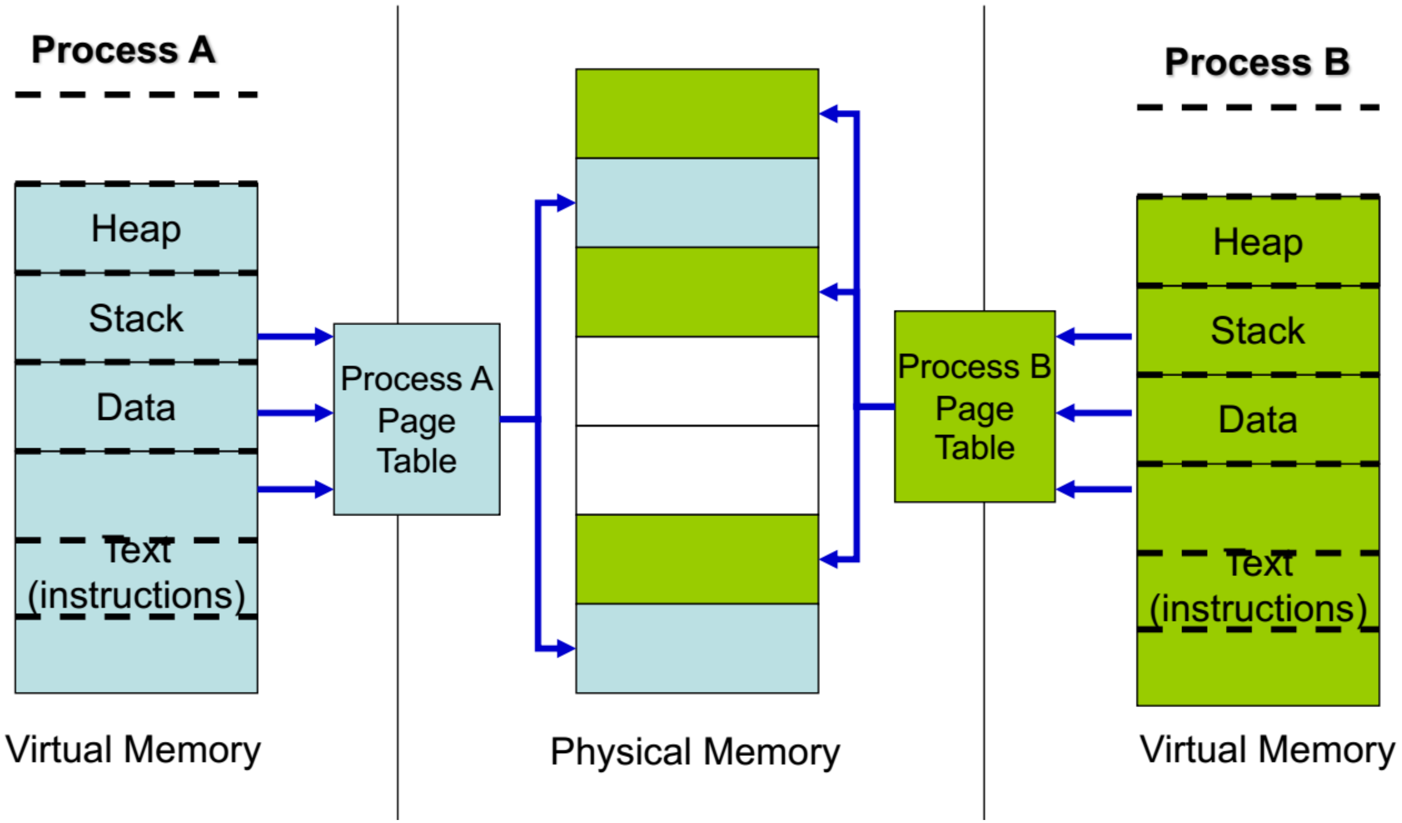
ie. 0 to MAXVA are virtual memory addresses

MAXVA



Address
Space

Virtual Address Mapping



How does OS create a process?

Allocates memory and creates memory image

– Loads code, static data from disk [exe] in to memory [Addr. Space]

– Creates runtime stack[local var, func.paramaters, ret.Addr], heap[malloc]

Opens basic files [Initializes basic I/O]

– STD IN, OUT, ERR [Three open file descriptors] (read from terminal, print output to screen]

Initializes CPU registers [main() - entry point]

– PC points to first instruction

Each process has 2 stacks

– **User space stack** Used when executing user code

– **Kernel space stack** Used when executing kernel code (for eg. during system calls)

– Advantage : Kernel can execute even if user stack is corrupted (Attacks that target the stack, such as buffer overflow attack, will not affect the kernel)

Trivia Time

If two processes P1 and P2 are running at the same time, what are the virtual address space ranges they will have ?

A) P1: 0 to 32000; P2: 32001 to 64000

B) P1: 0 to 64000; P2: 0 to 64000

C) P1: 32001 to 64000; P2: 0 to 32000

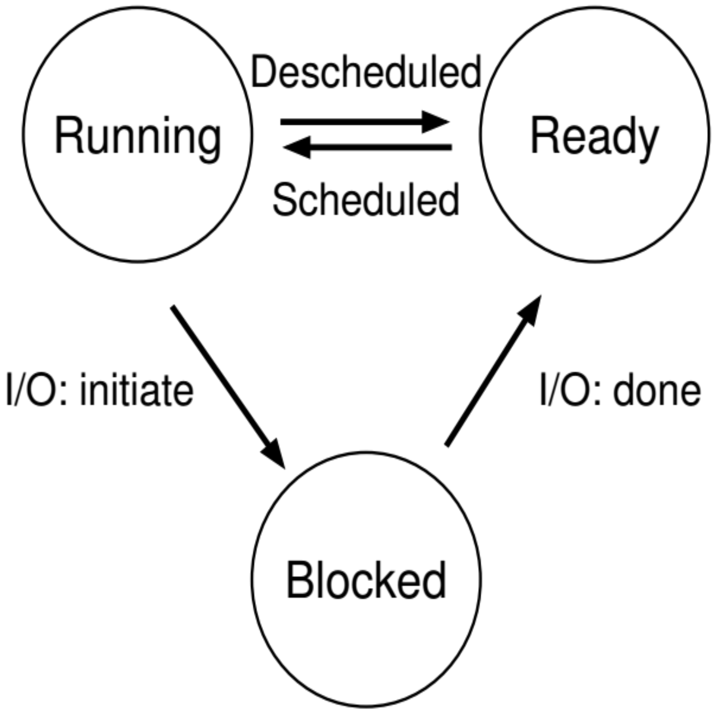
Process States

Running: A process is running on a processor. This means it is executing instructions.

Ready: A process is ready to run but for some reason the OS has chosen not to run it at this given moment.

Blocked: A process has performed some kind of operation that makes it not ready to run until some other event takes place.

Eg. when a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor. When is it unblocked? Disk issues an interrupt when data is ready

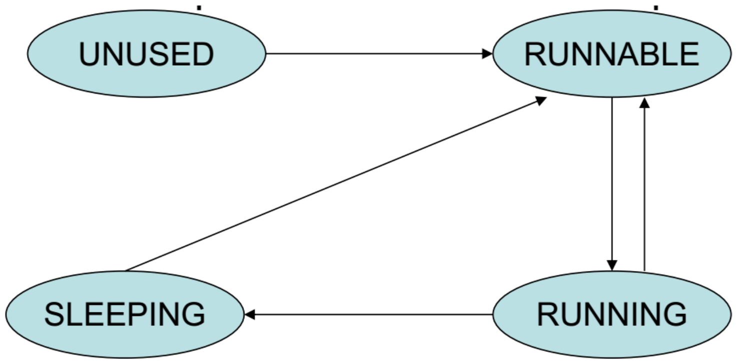


Process States

New: being created, yet to run

Dead: terminated

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked,
5	Blocked	Running	so $Process_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	$Process_1$ now done
9	Running	–	
10	Running	–	$Process_0$ now done



UNUSED → Unused PID
RUNNABLE → Ready to run
RUNNING → Currently executing
SLEEPING → Blocked for an I/O
Other states ZOMBIE (later)

Process Control Block

OS maintains a data structure (e.g., list) of all active processes

- Information about each process is stored in a process control block (PCB)

– Process identifier

– Process state

– Pointers to other related processes (parent)

– CPU context of the process (saved when the process is suspended)

– Pointers to memory locations

– Pointers to open files

```
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  struct proc *parent;         // Parent process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Bottom of kernel stack for this process
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // Page table
  struct trapframe *tf;        // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

What API does the OS provide to user programs?

API = Application Programming Interface= functions available to write user programs

- API provided by OS is a set of "system calls"

–System call is a function call into OS code that runs at a higher privilege level of the CPU

–Sensitive operations (e.g., access to hardware) are allowed only at a higher privilege level

–Some "blocking" system calls cause the process to be blocked and descheduled(e.g.,read from disk)

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

11th Jan, 2021

HOW TO CREATE AND CONTROL PROCESSES

What interfaces should the OS present for process creation and control?

How should these interfaces be designed to enable powerful functionality, ease of use, and high performance?

Process related system calls (in Unix)

fork()creates a new child process

fork ( )

–All processes are created by forking from a parent

–The init process is ancestor of all processes → Parent

- exec()makes a process execute a given executable
- exit()terminates a process
- wait()causes a parent to block until child terminates
- Many variants exist of the above system calls with different arguments

The first Process

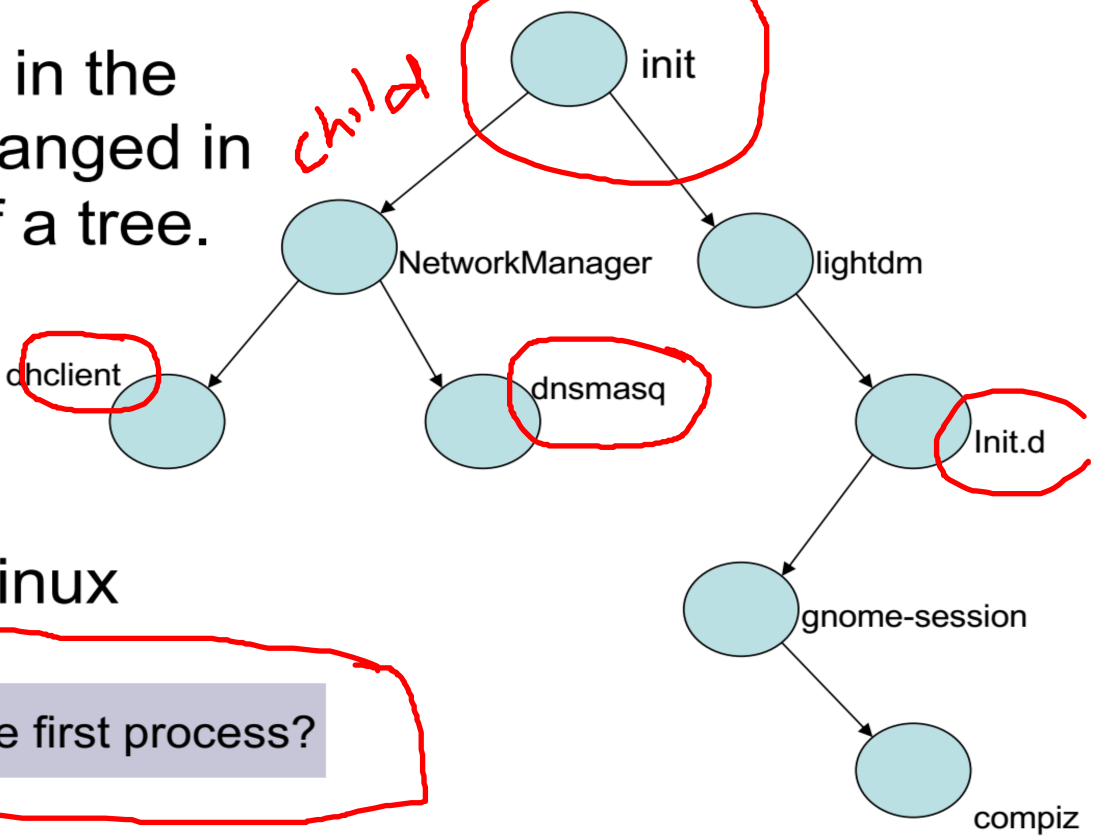
Unix : /sbin/init

- Unlike the others, this is created by the kernel during boot
- **Super parent.**
  - Responsible for forking all other processes
  - Typically starts several scripts present in /etc/init.d in Linux

Process Tree

Processes in the system arranged in the form of a tree.

child



pstree in Linux

Who creates the first process?

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

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;
}
```

P1.C

P1

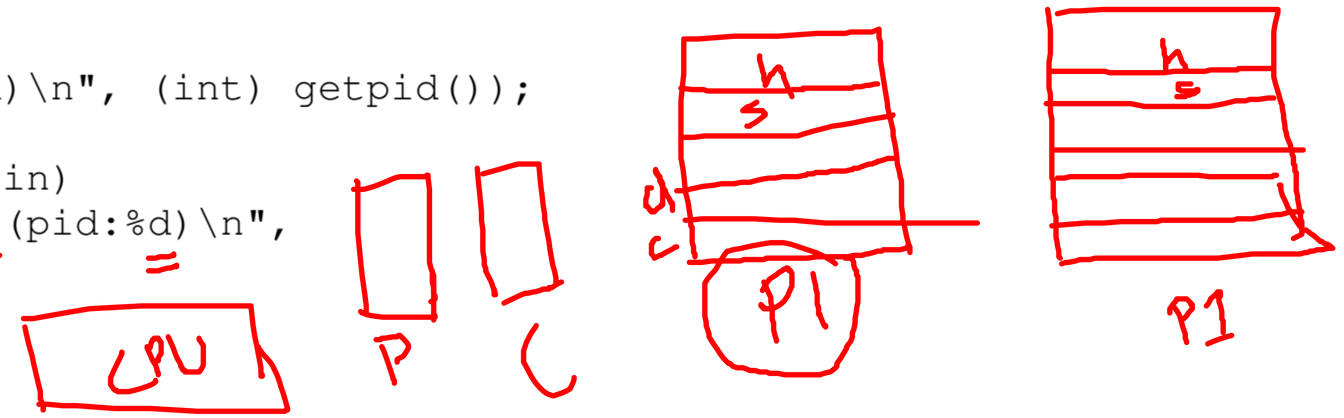
→ new

PID

rc = 0 child

= PID P

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



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

cleaning

Process Termination

- Voluntary : exit(status)
  - OS passes exit status to parent via wait(&status)
  - OS frees process resources → Reaping
- Involuntary : kill(pid, signal)

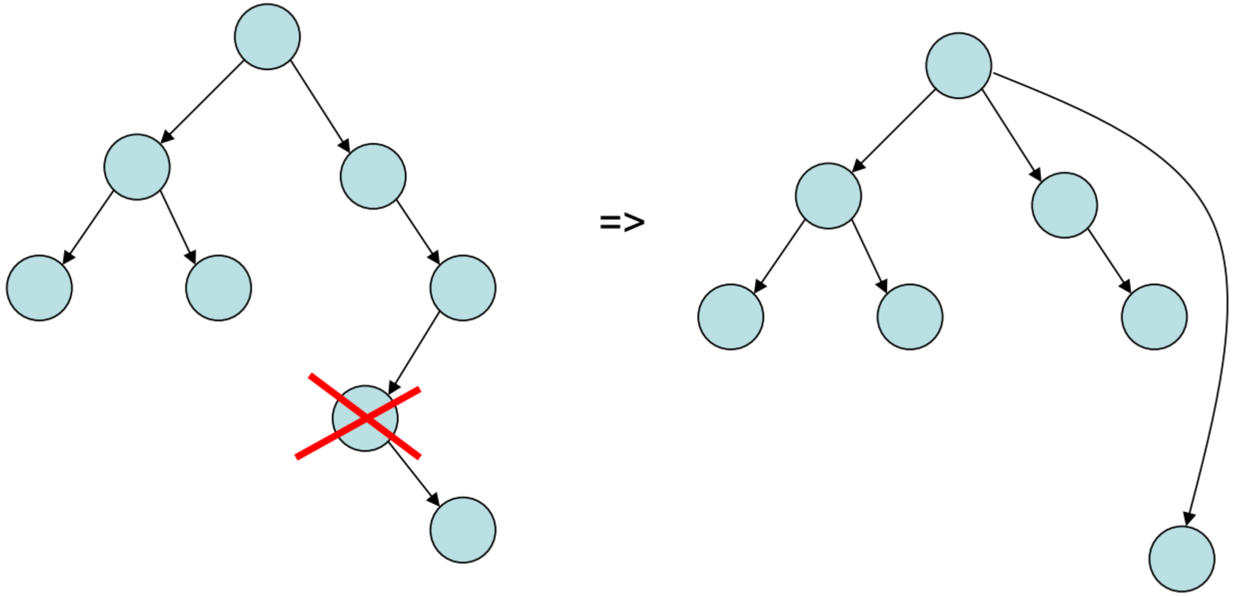
  SIGINT
  SIGTSTP
  - Signal can be sent by another process or by OS
  - pid is for the process to be killed
  - signal a signal that the process needs to be killed
    - Examples : SIGTERM, SIGQUIT (ctrl+\), SIGINT (ctrl+c), SIGHUP

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

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

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>
```

Trivia Time

```
int pid;
pid = fork();
if (pid > 0){
    printf("Parent : child PID = %d", pid);
    pid = wait();
    printf("Parent : child %d exited\n", pid);
} else{
   printf("In child process");
   exit(0);
}
```

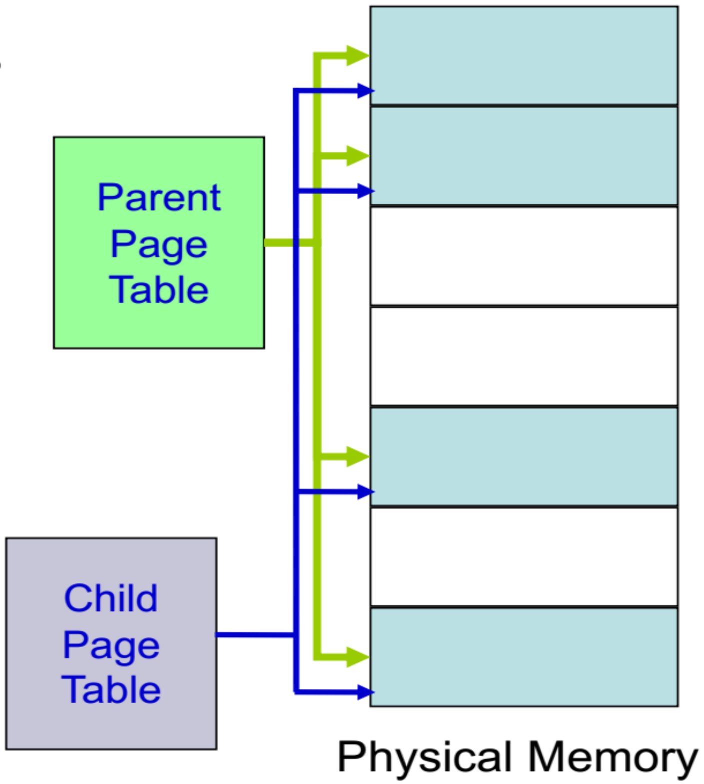
2

pid = 0

= PID

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

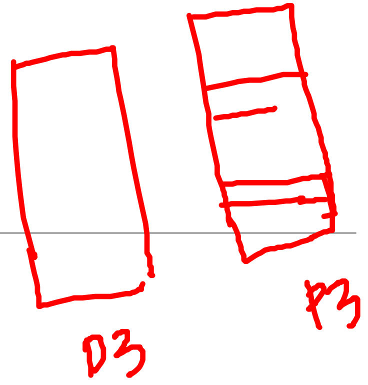


```c
int i=0, pid;
pid = fork();
if (pid > 0){
        sleep(1);
        printf("parent : %d\n", i);
        wait();
} else{
        i = i + 1;
        printf("child : %d\n", i);
}
```

Parent: 0

Child: 1

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. man

```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>
```

Exec()

exec system call

– Load into memory and then execute

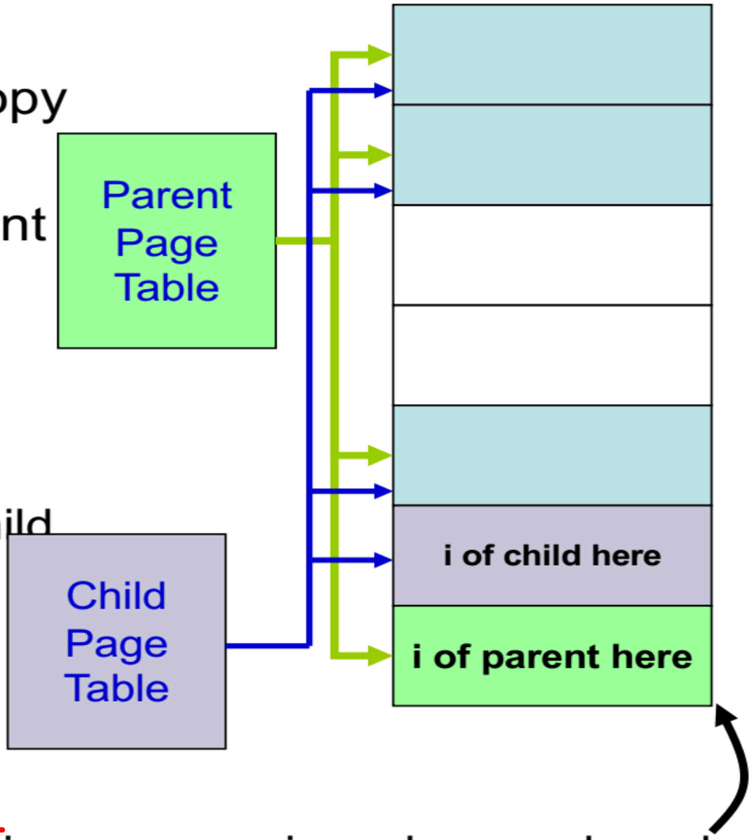
- COW big advantage for exec

– Time not wasted in copying pages.

– Common code (for example shared libraries) would continue to be shared

Copy on Write (CoW)

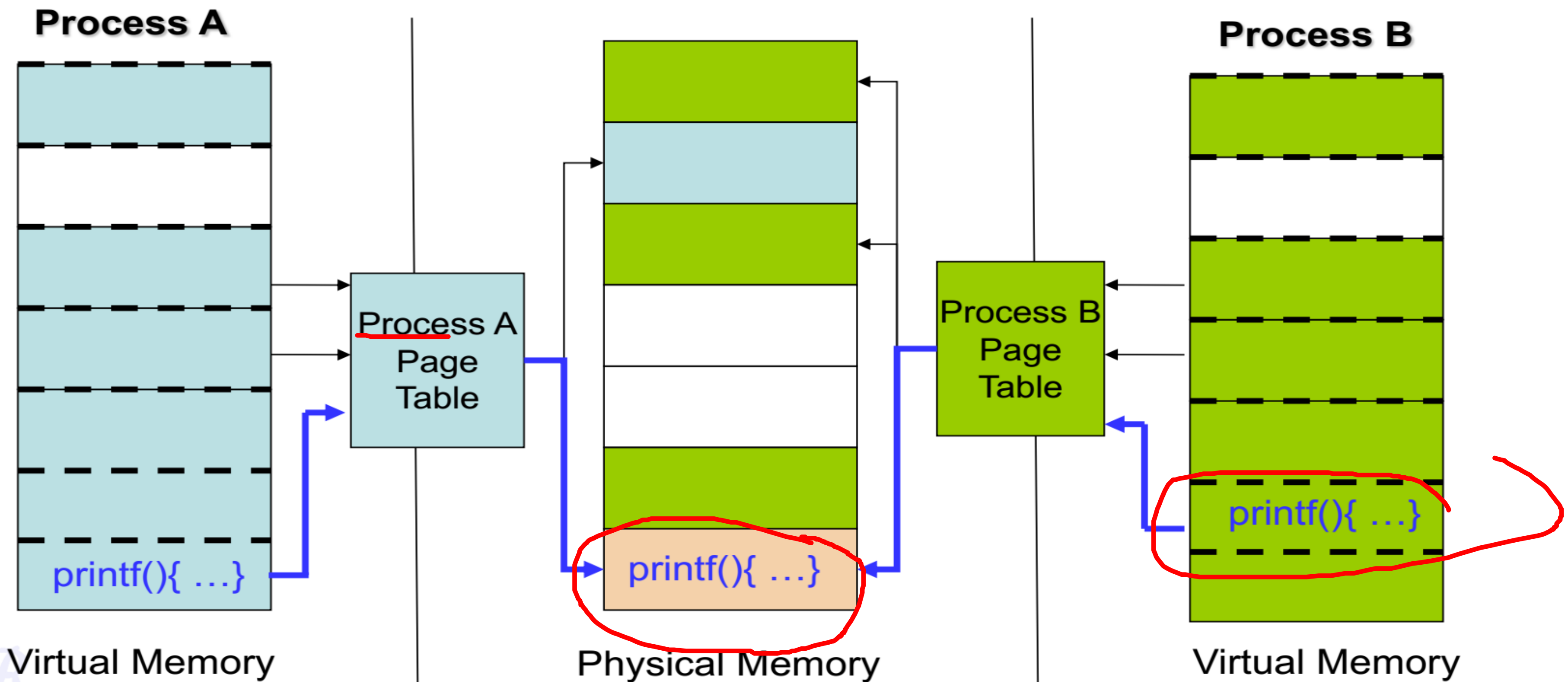
- When data in any of the shared pages change, OS intercepts and makes a copy of the page.
- Thus, parent and child will have different copies of this page
- Why?
  - A large portion of executables are not used.
  - Copying each page from parent and child would incur significant disk swapping.. huge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



This page now is no longer shared

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

COPY

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

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>
```