Lecture 3 : Process Management

INSTRUCTOR : DR. BIBHAS GHOSHAL (BIBHAS.GHOSHAL@IIITA.AC.IN)

Lecture Outline

- What is Process?
- Process States
- Process Creation
- Process Address Space
- Creating a process by Cloning
- Process Termination
- Zombie Process
- Orphans Process
- Process Address Map in xv6

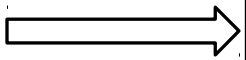
Process

- Program in Execution

```
#include<stdio.h>
int main(){
Char str[] = “Hello World!\n”;
Printf(“%s”,str);
}
```

$ gcc hello.c

ELF Executable(a.out)



Process

$ ./a.out

- Program
  - Code + static & global data
  - One program can create several processes

≠

Process

- Dynamic instantiation of code + data + heap + stack + process state
- A process is unique isolated entity

Process Comprises of:

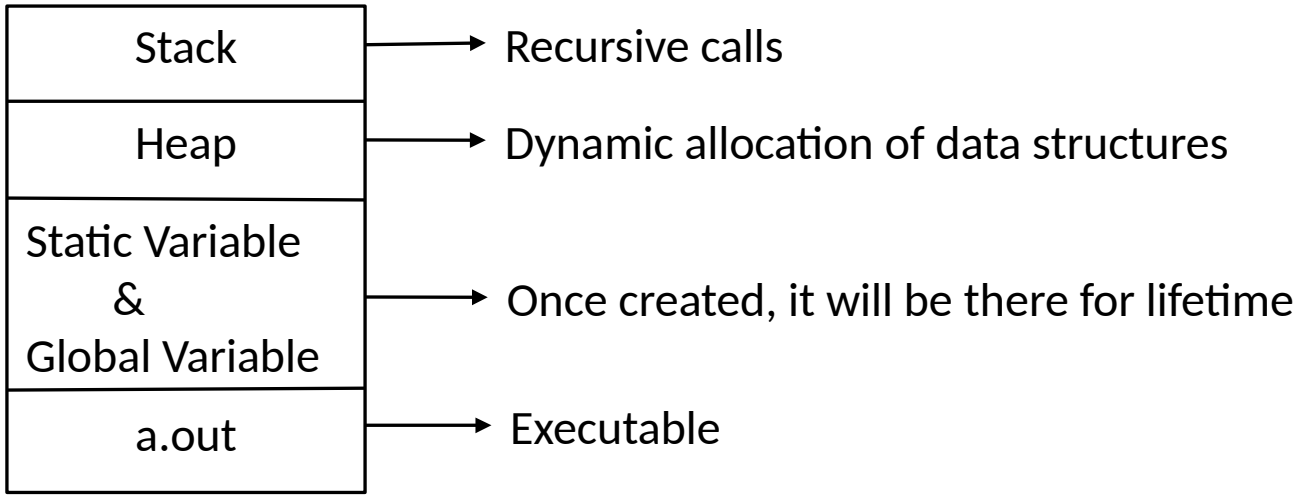
- Code
- Data
- Stack
- Heap

} In User space of process

- State in OS
- Kernel stack

} In Kernel space of process

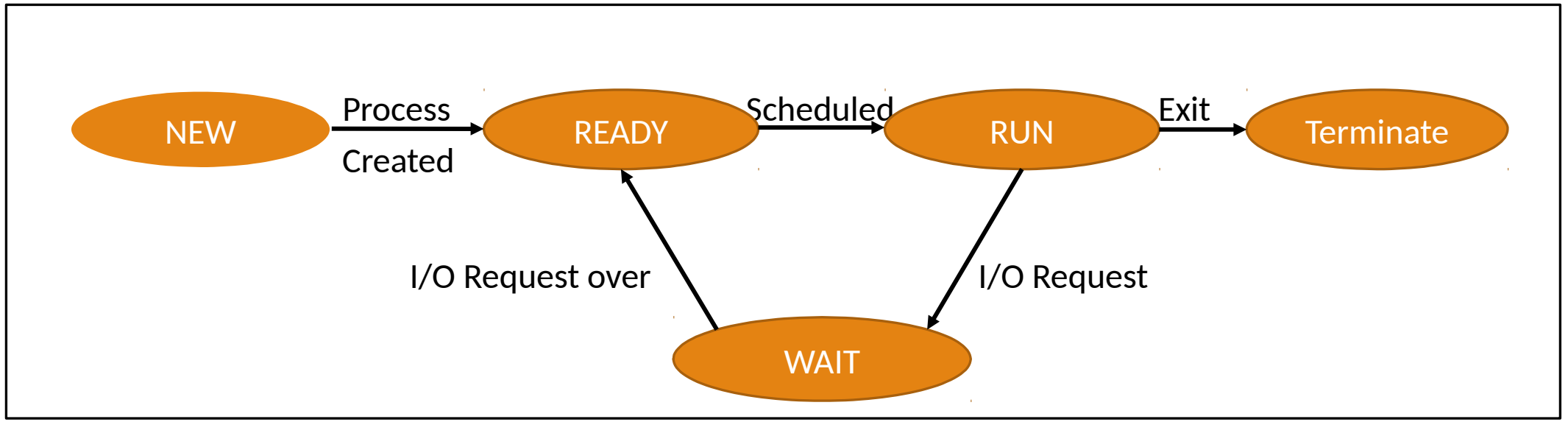
Data Structure of Process



- While executing a process, you are restricted to process boundary or else segmentation fault will occur.
- Heap grows upwards while Stack grows downwards.

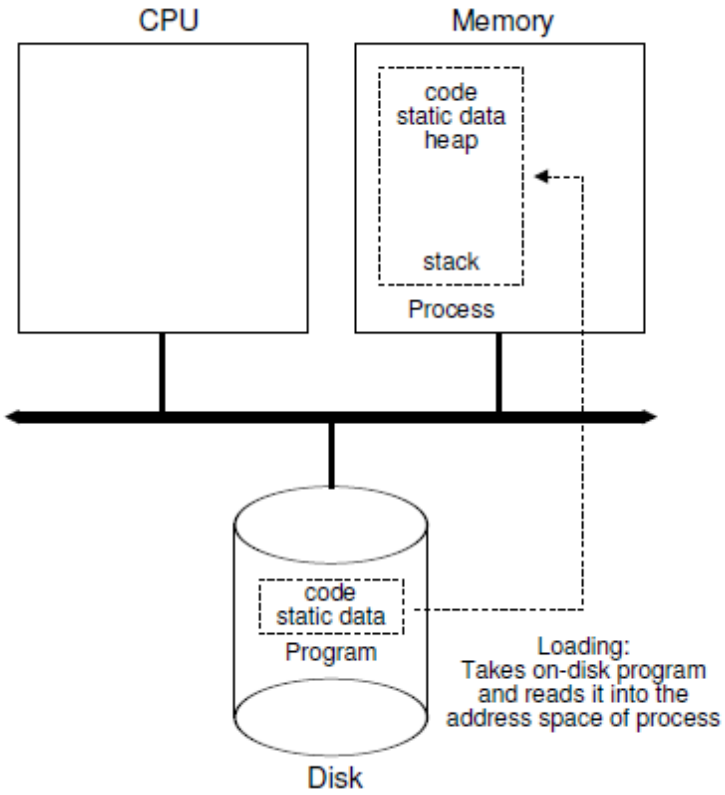
- Attributes of a Process
  - Process Id
  - Program counter
  - Process State
  - Priority
  - General Purpose Register
  - List of open files
  - List of open devices
  - Protection

Process State



Running: In the running state, a process is running on a processor. This means it is executing instructions.
Ready: In the ready state, a process is ready to run but for some reason the OS has chosen not to run it at this given moment.
Wait: In the blocked state, a process has performed some kind of operation that makes it not ready to run until some other event takes place. A common example: when a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.

Process Creation

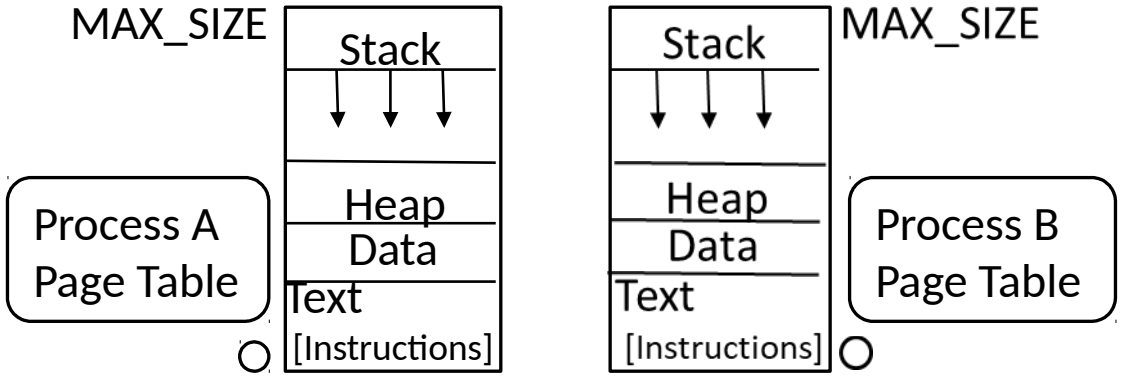


- The first thing that the OS must do to run a program is to **load** its code and any static data (e.g., initialized variables) into memory, into the address space of the process. Programs initially reside on **disk** (or, in some modern systems, **flash-based SSDs**) in some kind of **executable format**; thus, the process of loading a program and static data into memory requires the OS to read those bytes from disk and place them in memory somewhere.
- Once the code and static data are loaded into memory, there are a few other things the OS needs to do before running the process. Some memory must be allocated for the program's **run-time stack** (or just **stack**).
- The OS allocates this memory and gives it to the process. The OS may also allocate some memory for the program's **heap**. In C programs, the heap is used for explicitly requested dynamically-allocated data.
- The OS will also do some other initialization tasks, particularly as related to input/output (I/O). For example, in UNIX systems, each process by default has three open **file descriptors**.

Process Address Space

- Virtual Address Map
  - All memory a process can address
  - Large contiguous array of address from 0 to MAX_SIZE
  - Each Process has different address space
  - This is achieved by use of virtual memory

  i.e. 0 to MAX_SIZE are virtual memory addresses
- Advantages of Virtual Address Space
  - Isolation (Private address space)
  - Relocatable (Data & Code within the process is relocatable)
  - Size (Processes can be much larger than physical memory)



- Process Stack
  - User Space Stack (used when using user code)
  - Kernel Space Stack (used when using kernel code)
  - Advantages: Kernel can execute even when stack is corrupted (Attacks such as buffer overflow will not affect Kernel)
- Each Process has PCB (Process Control Block) : Holds process specific information
- PID : Process Identifier(No. incremented sequentially)

Virtualizing The CPU

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>
#include "common.h"

int
main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: cpu <string>\n");
        exit(1);
    }
    char *str = argv[1];
    while (1) {
        Spin(1);
        printf("%s\n", str);
    }
    return 0;
}
```

Code that loops and prints (loop.c)

```
prompt> ./cpu A & ./cpu B & ./cpu C & ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
...
```

Result of program cpu.c : Running many programs at once

Turning a single CPU (or small set of them) into a seemingly infinite number of CPUs and thus allowing many programs to seemingly run at once is what we call **virtualizing the CPU.**

Virtualizing The Memory

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "common.h"
5
6  int
7  main(int argc, char *argv[])
8  {
9      int *p = malloc(sizeof(int));                    // a1
10     assert(p != NULL);
11     printf("(%d) address pointed to by p: %p\n",
12            getpid(), p);                             // a2
13     *p = 0;                                          // a3
14     while (1) {
15         Spin(1);
16         *p = *p + 1;
17         printf("(%d) p: %d\n", getpid(), *p);        // a4
18     }
19     return 0;
20 }
```

A program that accesses memory (mem.c)

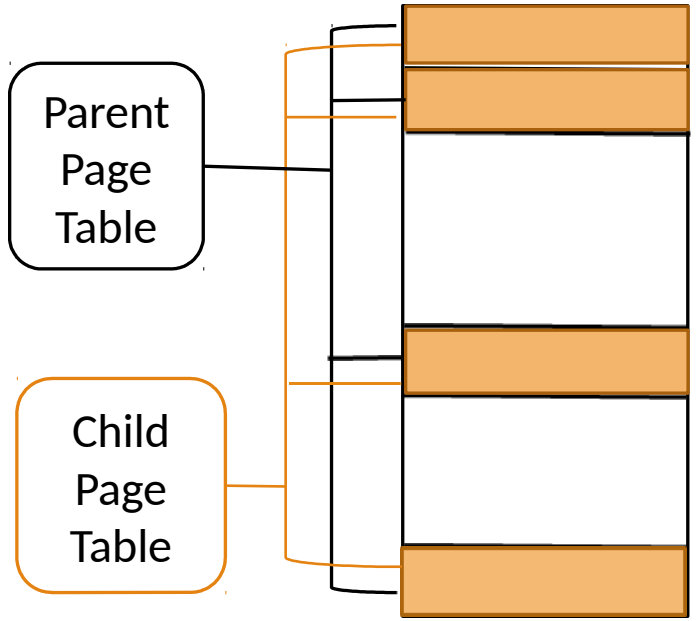
```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) address pointed to by p: 0x200000
(24114) address pointed to by p: 0x200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
...
```

Result of program mem.c : Running the memory program multiple times

Each running program has allocated memory at the same address (0x200000), and yet each seems to be updating the value at 0x200000 independently! It is as if each running program has its own private memory, instead of sharing the same physical memory with other running programs.

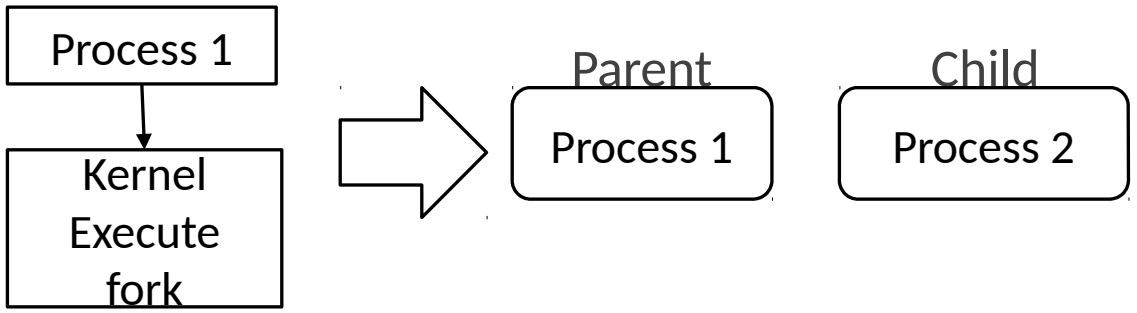
Virtual Memory Advantage

- Easy to make copies of process
- Making a copy of process is called forking
  - Parent is the original
  - Child (New Process)
- When fork() is invoked
  - Child is exact copy of Parent
  - All pages are shared between Parent & Child
  - Easily done by copying the parent's page tables



Creating A Process by Cloning

- Cloning : Child Process is expect replica of Parent
- ***fork()*** system call



- Operation on Process
  - Creation
  - Scheduling
  - Execution
  - Killing/delete

- In Parent process fork() return child process id
- In Child process fork() return process id of exit child
- ***wait()*** : If there is at least one child process running, caller is blocked until of its child exist & then caller resumes.

The *fork()* System Call

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int main(int argc, char *argv[]) {
6    printf("hello world (pid:%d)\n", (int) getpid());
7    int rc = fork();
8    if (rc < 0) {
9      // fork failed
10     fprintf(stderr, "fork failed\n");
11     exit(1);
12   } else if (rc == 0) {
13     // child (new process)
14     printf("hello, I am child (pid:%d)\n", (int) getpid());
15   } else {
16     // parent goes down this path (main)
17     printf("hello, I am parent of %d (pid:%d)\n",
18             rc, (int) getpid());
19   }
20   return 0;
21 }
```

When you run this program (called `p1.c`), you'll see the following:

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

The *wait()* System Call

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(int argc, char *argv[]) {
7    printf("hello world (pid:%d)\n", (int) getpid());
8    int rc = fork();
9    if (rc < 0) {          // fork failed; exit
10     fprintf(stderr, "fork failed\n");
11     exit(1);
12   } else if (rc == 0) { // child (new process)
13     printf("hello, I am child (pid:%d)\n", (int) getpid());
14   } else {               // parent goes down this path (main)
15     int rc_wait = wait(NULL);
16     printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
17            rc, rc_wait, (int) getpid());
18   }
19   return 0;
20 }
```

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

The *exec()* System Call

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <sys/wait.h>
6
7  int main(int argc, char *argv[]) {
8    printf("hello world (pid:%d)\n", (int) getpid());
9    int rc = fork();
10   if (rc < 0) {          // fork failed; exit
11     fprintf(stderr, "fork failed\n");
12     exit(1);
13   } else if (rc == 0) { // child (new process)
14     printf("hello, I am child (pid:%d)\n", (int) getpid());
15     char *myargs[3];
16     myargs[0] = strdup("wc");   // program: "wc" (word count)
17     myargs[1] = strdup("p3.c"); // argument: file to count
18     myargs[2] = NULL;           // marks end of array
19     execvp(myargs[0], myargs);  // runs word count
20     printf("this shouldn't print out");
21   } else {                  // parent goes down this path (main)
22     int rc_wait = wait(NULL);
23     printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
24            rc, rc_wait, (int) getpid());
25   }
26   return 0;
27 }
```

```
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>
```

Process Termination

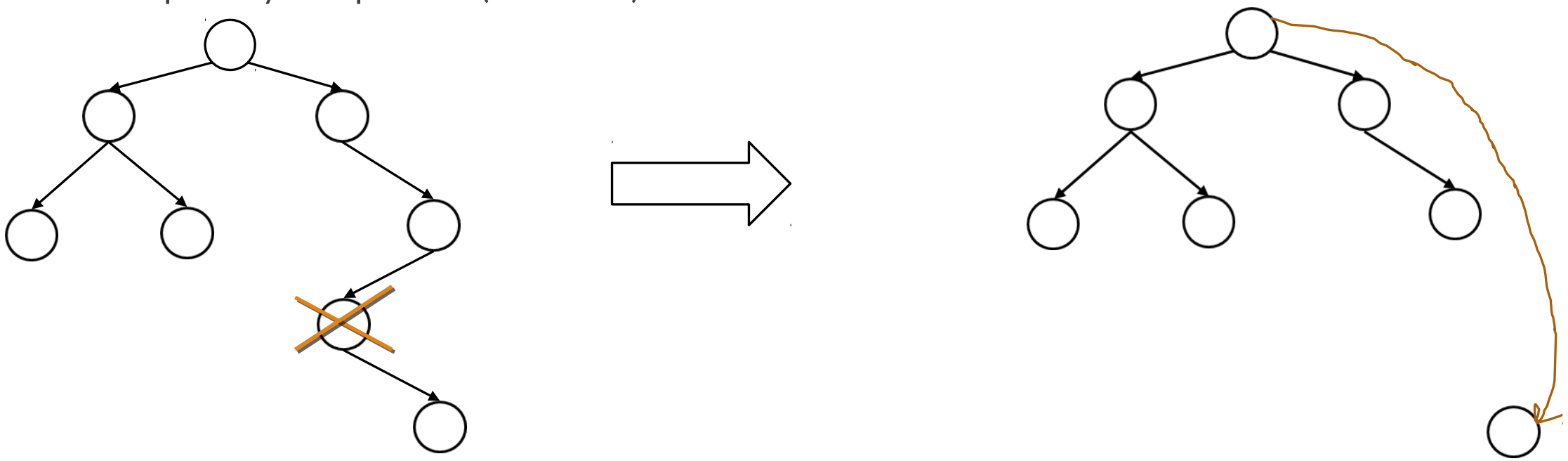
- Voluntary : exit(status)
  - OS passes exit status to parent via wait(&status)
  - OS frees process resources
- Involuntary : kill(pid, signal)
  - Signal can be sent by another process or by OS
  - pid is for the process to be killed
  - A signal that the process needs to be killed
  - For example : SIGTERM, SIGQUIT(ctrl+\), SIGINT(ctrl+c), SIGHUP

Zombie Process

- Child terminates before Parent
- The OS stores some info about the terminated child (PID, termination reason etc.)
- The Parent of the terminated child accesses this info using the *wait()* system call.
- If a parent did not call wait(), it becomes a zombie
- Process has no allocated resources but its entry in the process table (PCB)
- A zombie process can be noticed using ***ps*** command, which prints on the state column of the process (the letter ‘z’)
- PCB in OS still exist even though program no longer executing
- When parent reads status
  - zombie entries removed from OS… process reaped!
- When parent doesn’t read status
  - zombie will continue to exist infinitely… a resource leak
  - These are typically found by a reaper process

Orphan Process

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



Orphans contd.

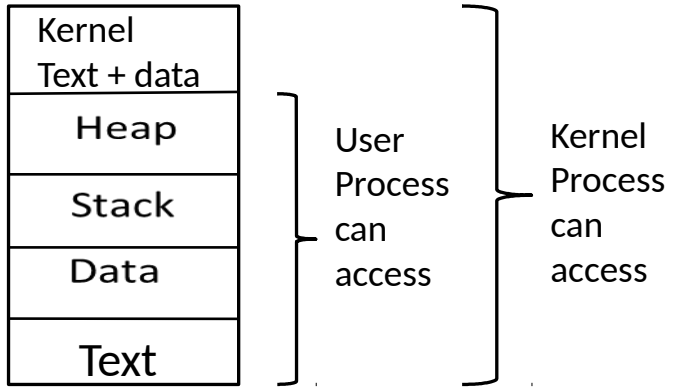
- Unintentional orphans
  - When parent crashes
- Intentional orphans
  - Process becomes detached from user session and runs in the background
  - Called Daemons process, used to run background services
  - See nohup

Process Address Map in xv6

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                  // Start of process memory
  uint sz;                    // Size of process memory
  char *kstack;               // Bottom of kernel stack
                              // for this process
  enum proc_state state;      // Process state
  int pid;                    // Process ID
  struct proc *parent;        // Parent process
  void *chan;                 // If !zero, sleeping on chan
  int killed;                 // If !zero, has been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd;          // Current directory
  struct context context;     // Switch here to run process
  struct trapframe *tf;       // Trap frame for the
                              // current interrupt
};
```



Entire Kernel mapped into every process address space

⇩

Easy Switching from user code to kernel code (during system calls) : No change of Page table

- Context Pointer
  - Contains register used for ***Context Switches***
  - Registers in context: %edi, %esi, %ebx, %ebp, %eip
  - Stored in Kernel Space

The xv6 Proc Structures