Fall 2015 COMP 3511
Operating Systems

Lab #3

Outline

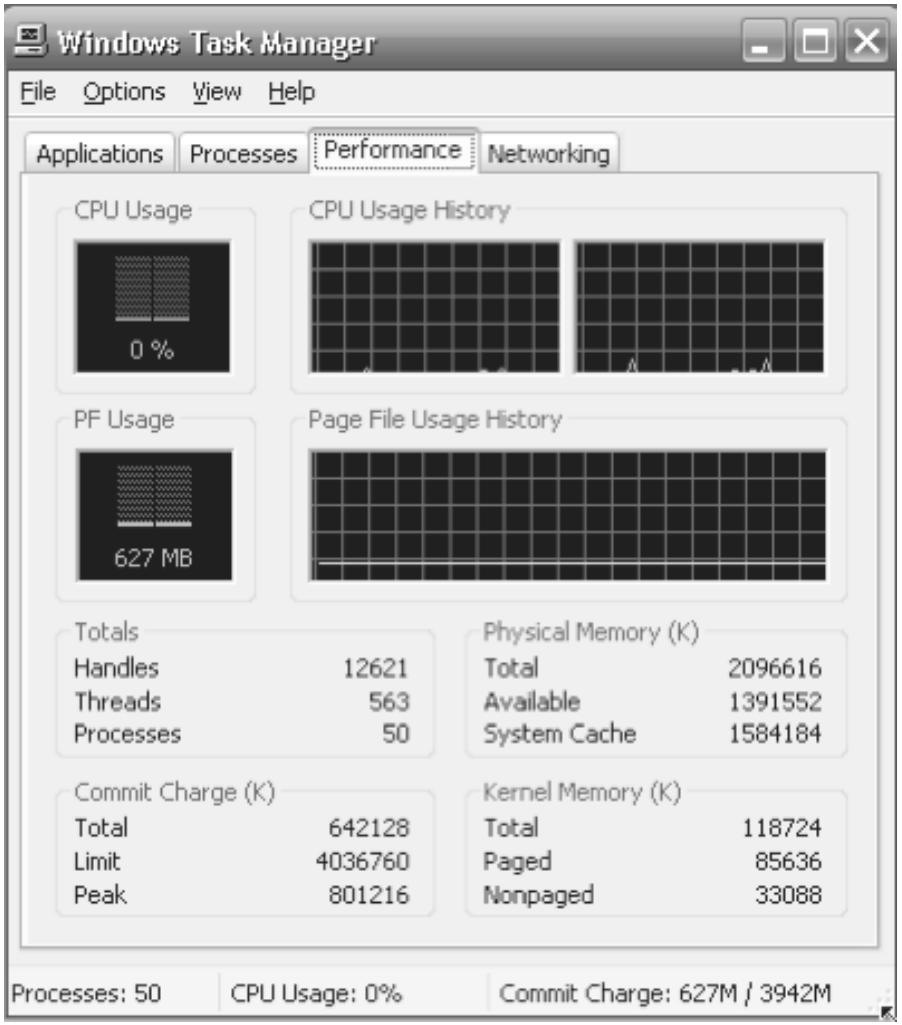
- Operating System Debugging, Generation and System Boot
- Review Questions
- Process Control
- UNIX fork() and Examples on fork()
- exec family: execute a program

Operating-System Debugging

- **Debugging** is finding and fixing errors, or **bugs**
- Usually the OS generates **log files** containing error information
- Failure of an application can generate **core dump** file capturing memory of the process
- Operating system failure can generate **crash dump** file containing kernel memory
- Beyond crashes, performance tuning can optimize system performance
  - Sometimes using ***trace listings*** of activities, recorded for analysis
  - **Profiling** is periodic sampling of instruction pointer to look for statistical trends

Performance Tuning

- Improve performance by removing bottlenecks
- OS must provide means of computing and displaying measures of system behavior
- For example, “top” program or Windows Task Manager



Operating System Generation

- Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site

- **SYSGEN** program obtains information concerning the specific configuration of the hardware system
  - Used to build system-specific compiled kernel or system-tuned
  - Can general more efficient code than one general kernel

System Boot

- When power initialized on system, execution starts at a fixed memory location
  - **Firmware ROM** used to hold initial boot code
- Operating system must be made available to hardware so hardware can start it
  - Small piece of code – **bootstrap loader**, stored in **ROM** or **EEPROM** locates the kernel, loads it into memory, and starts it
  - Sometimes two-step process where **boot block** at fixed location loaded by ROM code, which loads bootstrap loader from disk
- Common bootstrap loader, **GRUB**, allows selection of kernel from multiple disks, versions, kernel options
- Kernel loads and system is then **running**

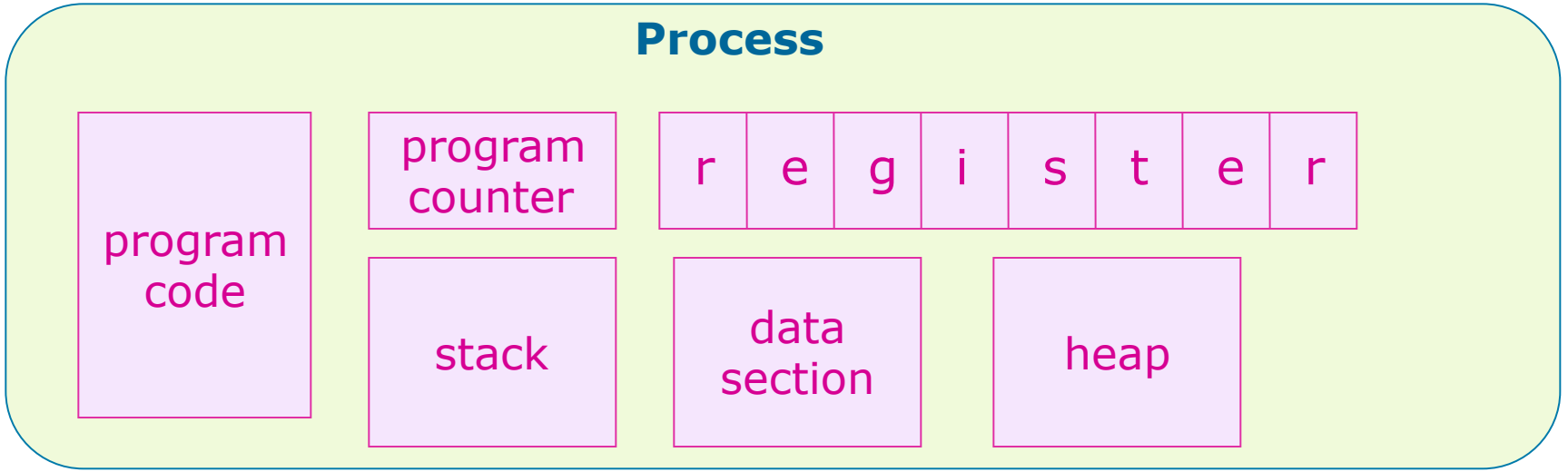
Q. 1

- What is the main difference between a **program** and a **process**?

A program is static (lines of codes stored)

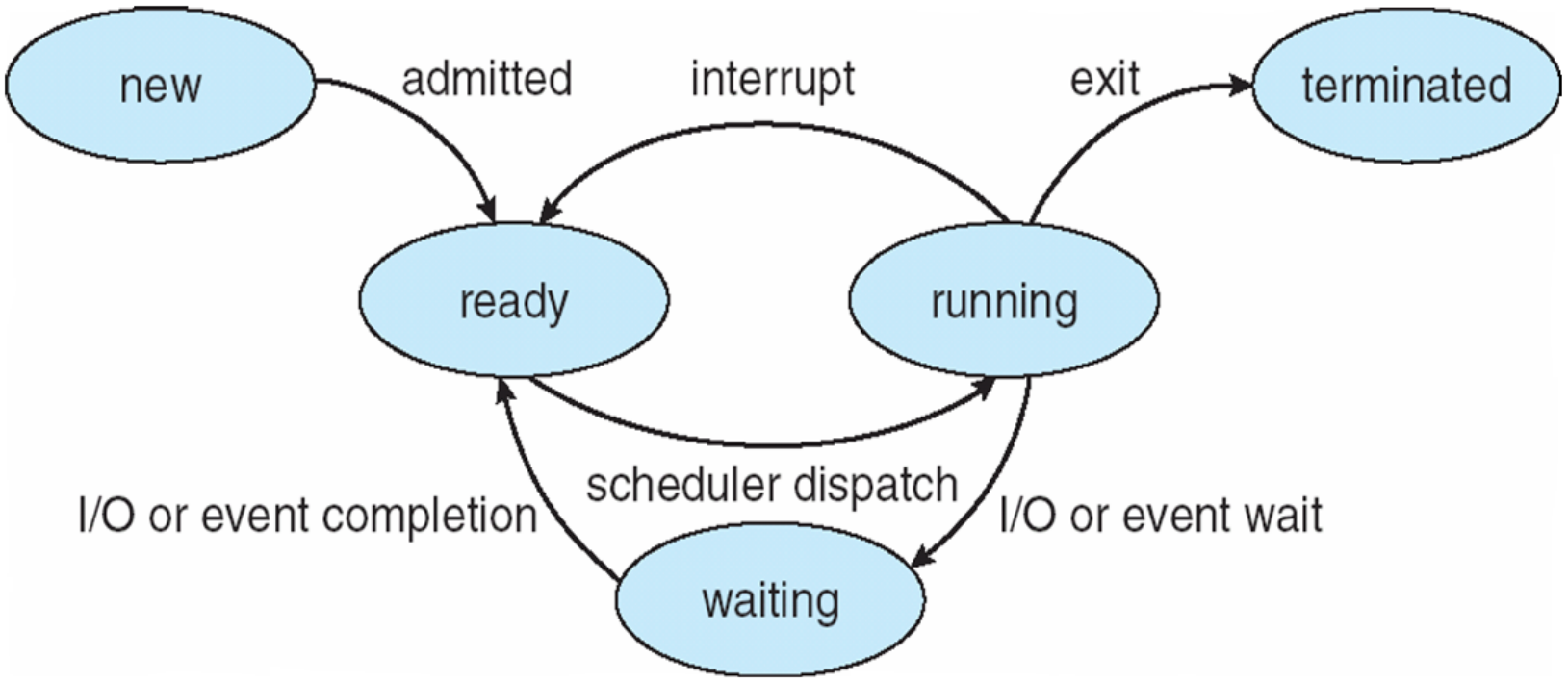
A process is active in execution,

which has a life cycle and can be in different states



Q. 2

- Briefly describe the process lifecycle with different states.



Q. 3

- When a process creates a child process, what are the four tasks that need to be done?

Creates a new PCB for the child process

allocate address space for the child process

copy data from the parent process

copy I/O state if any

Q. 4

- What are the two possibilities in terms of the address space of a newly created process?

A. The child process has a new program loaded into it

B. The child process has an exact copy of the address space of the parent process including program

C. The child process has an exact copy of the address space of the parent process including data

D. The child process has an exact copy of the address space of the parent process including program and data

Cont.

- What are the two possibilities in terms of the address space of a newly created process?

A. The child process has a new program loaded into it

B. The child process has an exact copy of the address space of the parent process including program

C. The child process has an exact copy of the address space of the parent process including data

D. The child process has an exact copy of the address space of the parent process including program and data

Q. 5

- Describe the differences between **short-term** and **long-term scheduling**.
  - Short-term scheduling (CPU scheduler) selects which process should be executed next and allocates CPU
  - Long-term scheduling (job scheduler) determines which processes should be brought into the ready queue

Cont.

- The primary difference is in the frequency of their execution
  - Short-term scheduling must select a new process quite often
  - Long-term scheduling is used much less often since it handles placing jobs in the system and may wait a while for a job to finish before it admits another one

Fork() example in C

```
int main(void)
{
    pid_t pid = fork();
    if (pid == -1 ) {
                    /* when fork() return -1, an error occurred */
         fprintf(stderr, "Fork Failed");
         exit(EXIT_FAILURE);
    }
    else if (pid == 0) {
                    /* when fork() return 0, we are in the child
 process */
         printf("Hello from the child process!");
                    _exit(EXIT_SUCCESS);
               }

           else {
         /* when fork() return a positive integer, we are in the
 parent process */
                    /* the return value of the process id of the
 newly created child process */
                    int status;
         (void) waitpid(pid, &status, 0);
               }
                    return EXIT_SUCCESS;
```

Q. 6

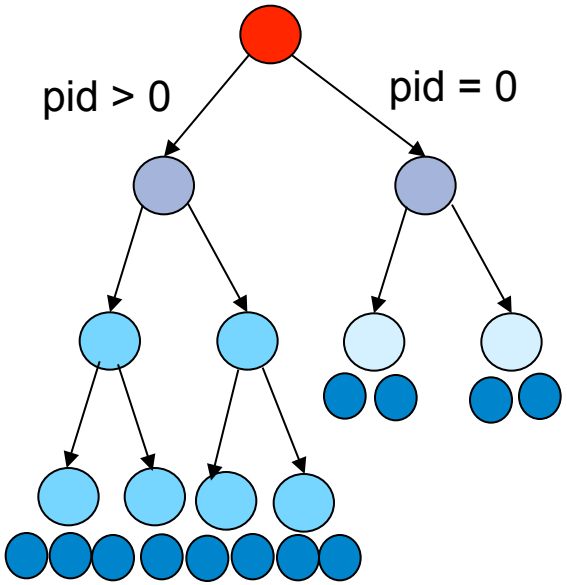
- Consider the following code segment:

```
pid_t pid;
pid = fork();
if (pid == 0)  {
        fork();
 }
 if (pid > 0)  {
        fork(); fork();
 }
  fork();
```

- Q: How many distinct child processes will be generated?

Q. 6

- Consider the following code segment:
- pid_t pid;
- pid = fork();
- if (pid == 0) {
- fork();
- }
- if (pid > 0) {
- fork(); fork();
- }
- fork();



- Q: How many distinct child processes will be generated?

Process control

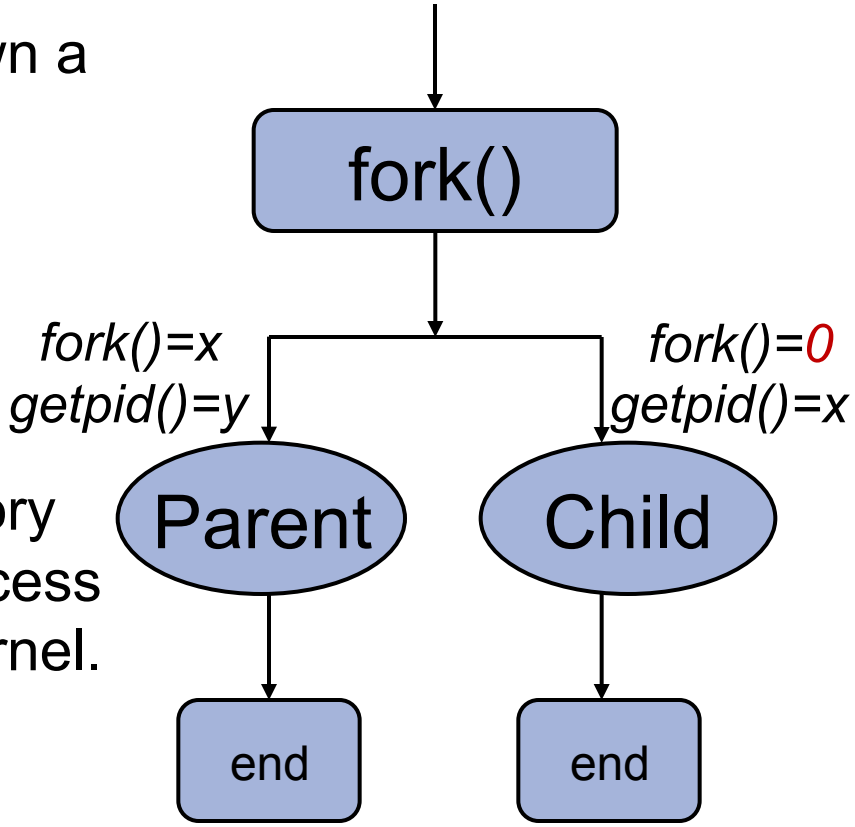
- When UNIX runs a process, it gives each process a **unique** number called **process ID**, or **pid**.
  - may be a "system" program (e.g., login, csh)
  - or a program initiated by the user (e.g., textedit, dbxtool or a user written one).

Process control

- The UNIX command "**ps**" will list all current processes running on your machine with their pid.
- The C function int **getpid**() will return the process id of process that called this function.

fork(): create a new process

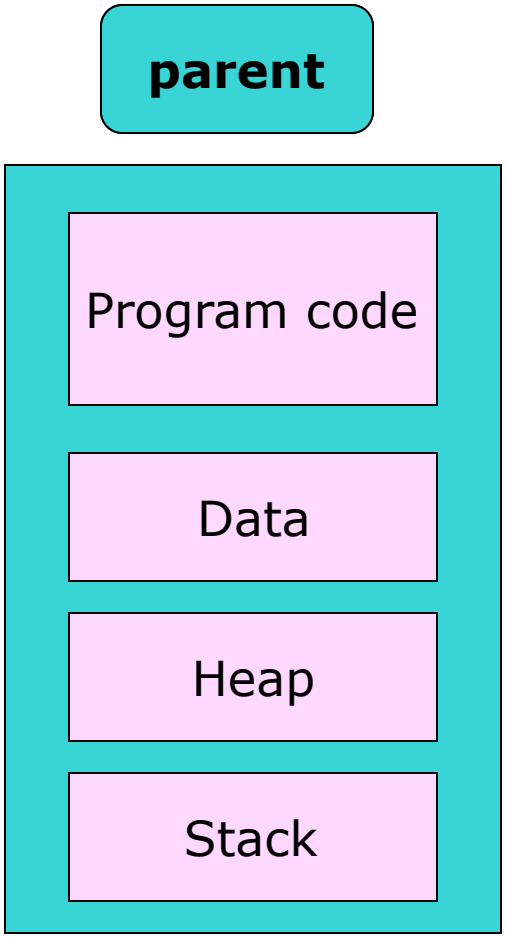
- The fork() system call will spawn a new child process, which is an identical process to the parent except that has a new system process ID.
- The process is copied in memory from the parent and a new process structure is assigned by the kernel.



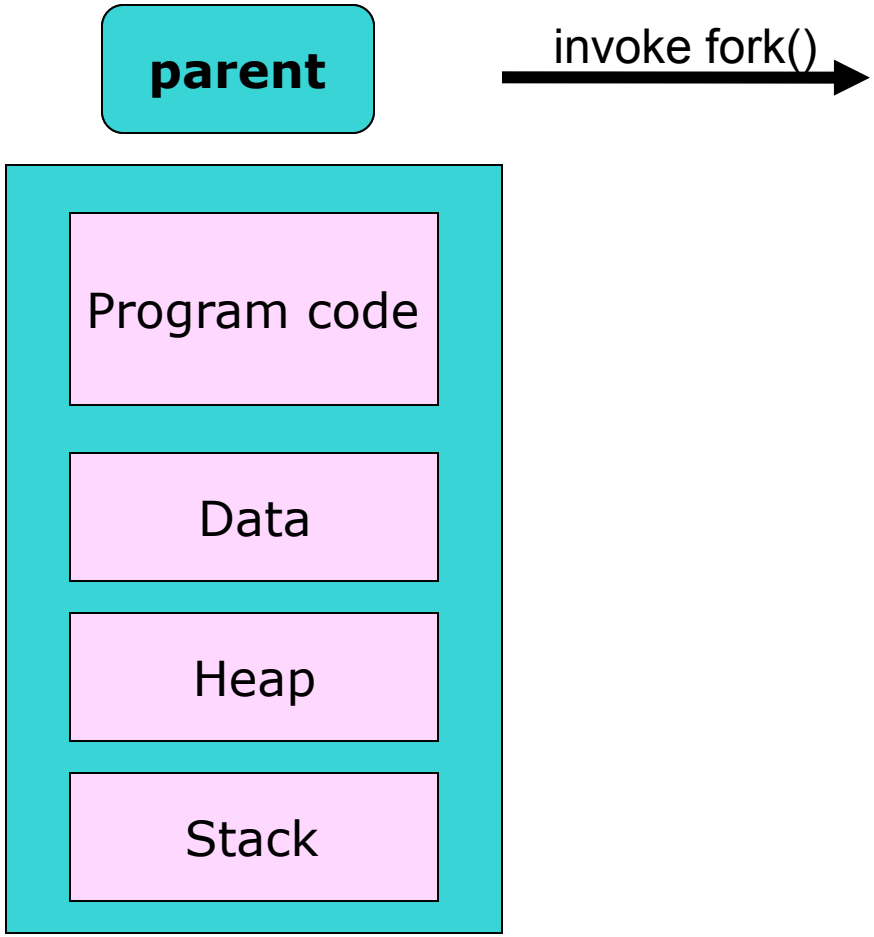
fork(): create a new process

- Synopsis:
  - #include <sys/types.h>
    #include <unistd.h>
  - pid_t fork(void);

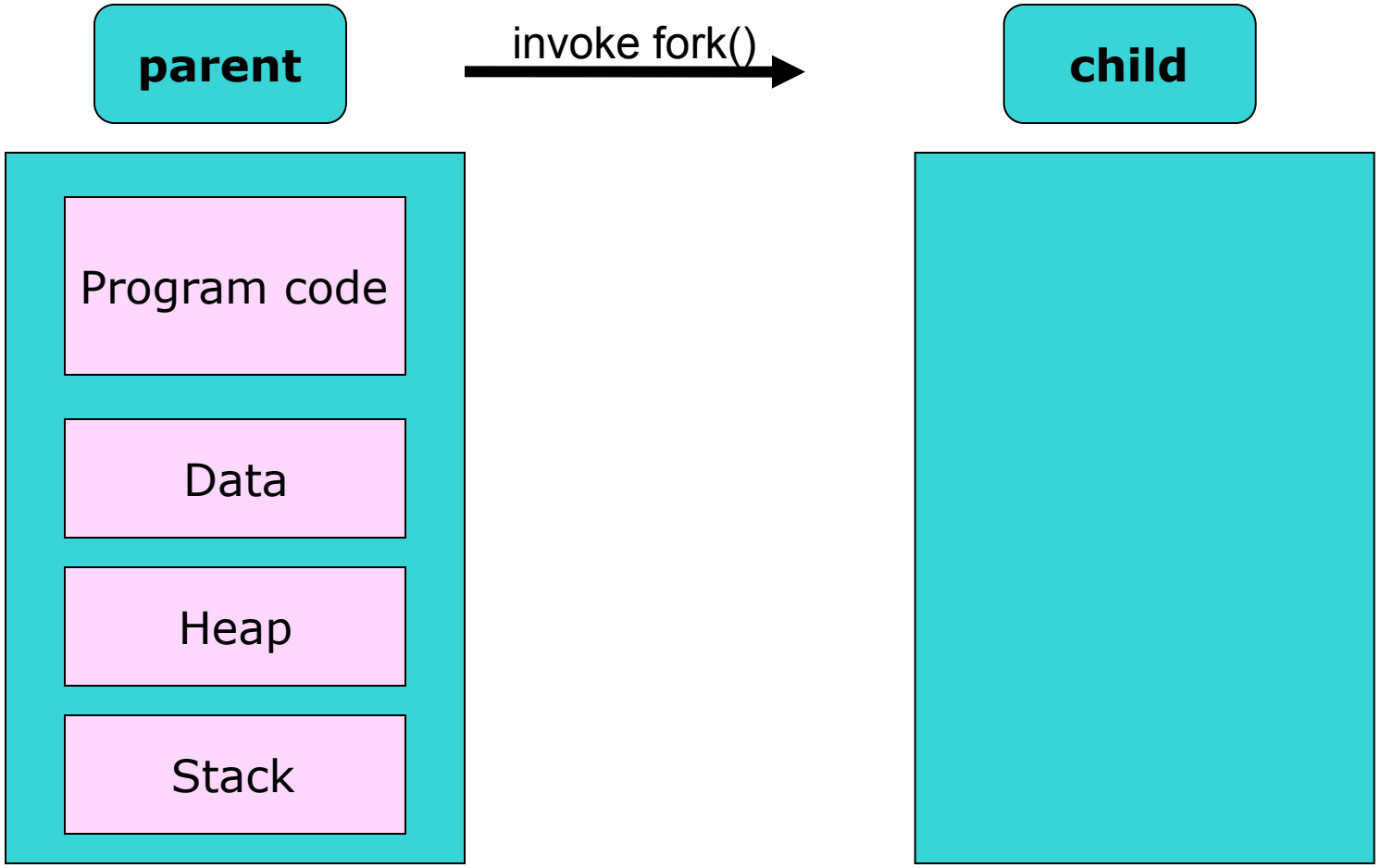
fork(): create a new process



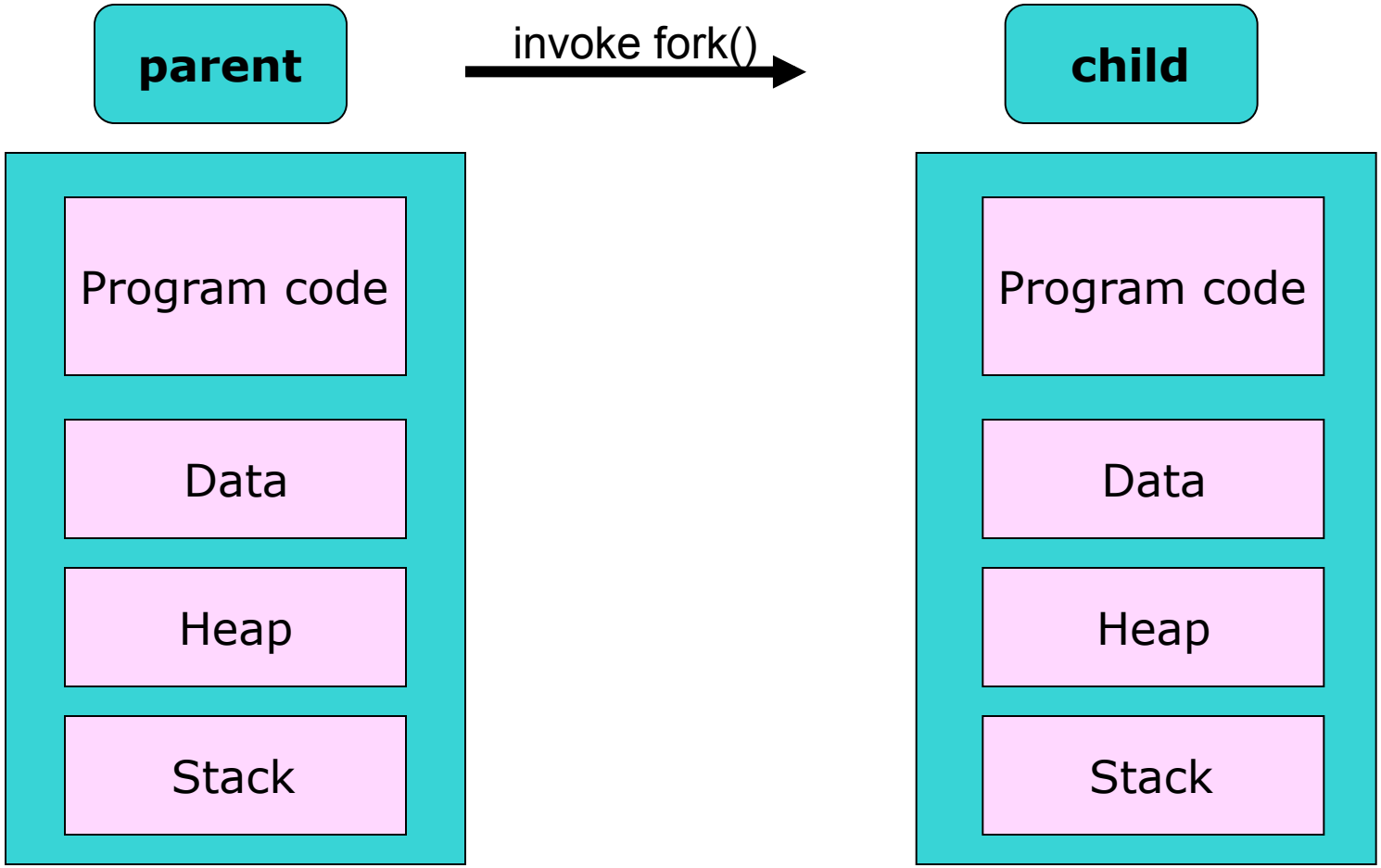
fork(): create a new process



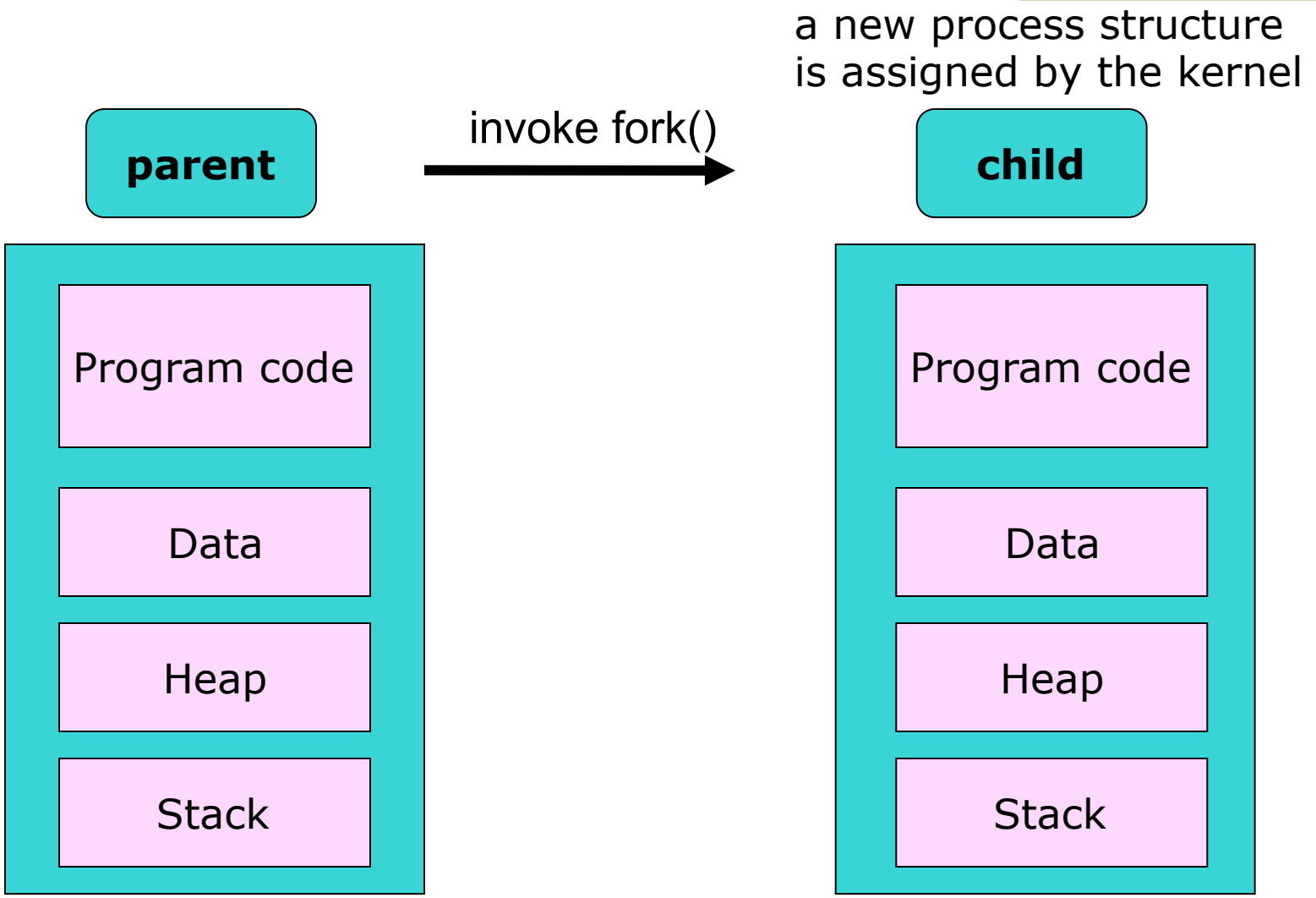
fork(): create a new process



fork(): create a new process



fork(): create a new process



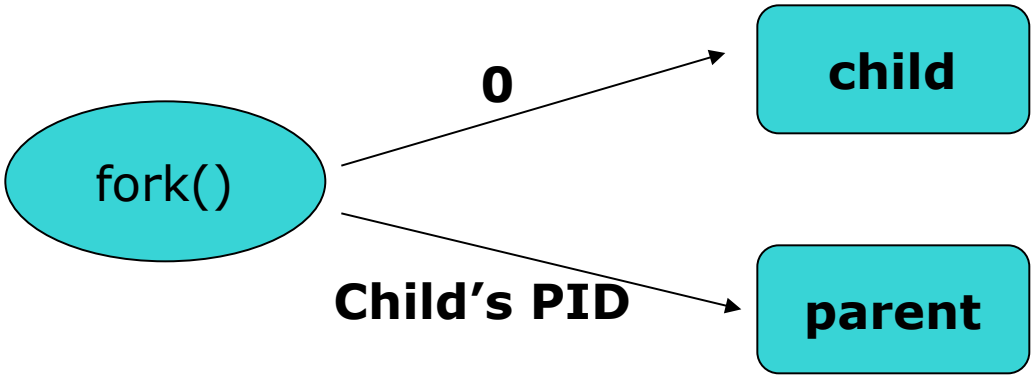
fork(): create a new process

Parent & Child:

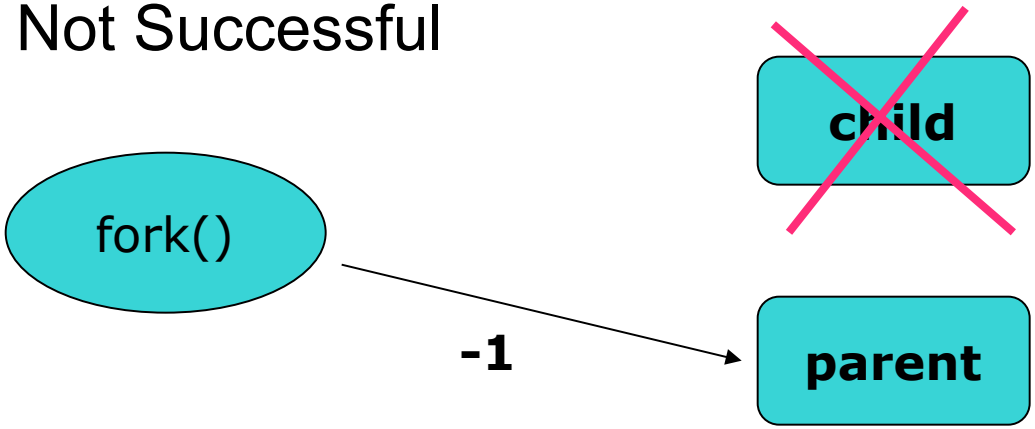
- Duplicated
  - Address space
  - Global & local variables
  - Current working directory
  - Root directory
  - Process resources
  - Resource limits
  - Etc…
- Different
  - PID
  - Running time
  - Running state
  - Return values from fork()

Return values of fork()

- Successful



- Not Successful



errno is set to indicate error

Return values of fork()

- The return value of the function is which discriminates the two processes of execution.
- Upon successful completion, fork() return 0 to the child process and return the process ID of the child process to the parent process.
- Otherwise, (pid_t)-1 is returned to the parent process, no child process is created, and errno is set to indicate the error.

A simple C program on fork()

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int value = 5;
int main()
{
    pid_t pid;
    pid = fork();
    if (pid == 0) { /* child process */
            value += 15;
            printf("CHILD: value = %d\n", value); /* Line A */
    }
    else if (pid > 0) { /* parent process */
            wait (NULL);
            value -=10;
            printf("PARENT: value = %d\n", value); /* Line B */
            exit(0);
    }
}
```

A simple C program on fork()

- Code can be downloaded from:
  - http://course.cs.ust.hk/comp3511/lab/lab03/testfork.c
- Compile
  - gcc -o testfork testfork.c
- Run and understand the result
  - ./testfork

A simple C program on fork()

- Line A: CHILD: value = 20
- Line B: PARENT: value = -5

  - Upon fork() system call, the variable “value” is made a copy in the child process, so it prints out 5+15=20;

  - after wait() system call, the child terminates and its copy of “value” is destroyed.

  - The parent has “value=5”, so it prints out 5-10= -5.

More example on fork()

- Consider the program segment with fork() instruction below, and suppose each process can run to completion, i.e., no interrupt in the middle of a process execution.
- Please answer the following two questions.
  - Which part runs as the parent process and which as the child process?
  - Without making any assumption on the order of executions, please show all possible outputs (suppose each process runs to completion, i.e. no interruption)

```
main()
{
    int x;
    x=0;
    if (fork())
        { x=x+1; /* A part */
          printf("A produces %2d\n", x);
        }
    else
        { x=x+1; /* B part */
          printf("B produces %2d\n", x);
        }
}
```

More example on fork()

- Which part runs as the parent process and which as the child process?
- A part: Parent B part: Child

- All possible outputs

  A produces 1

  B produces 1

  Or

  B produces 1

  A produces 1

More example on fork()

- How many processes are created, if the following program finishes successfully?

```
int main() {
int i=0;
for (i=1; i<=100; i++) { fork(); }
}
```

- **Answer: 2^100 or 2^100-1.**

  This can be deduced by 1 fork() generates 2 processes,2 fork() generates 4 process and so on

More example on fork()

- How many processes are created after the following program executes?

```
int main() {
if (fork()>0) fork();
}
```

- **Answer: 3 (if you do not count the original processes, the answer is 2)**

exec family: execute a program

- fork() can only duplicate a process
- How to execute other programs like “ls”?

```
//ProgramA.c

int main()
{
        :
  want to execute “ls”
        :
}
```

exec family: execute a program

- The exec system call family has the following members:
  - execl
  - execlp
  - execle
  - execv
  - execvp
  - execve
- The exec system call family changes the process image of the calling process.

exec(): execute a file

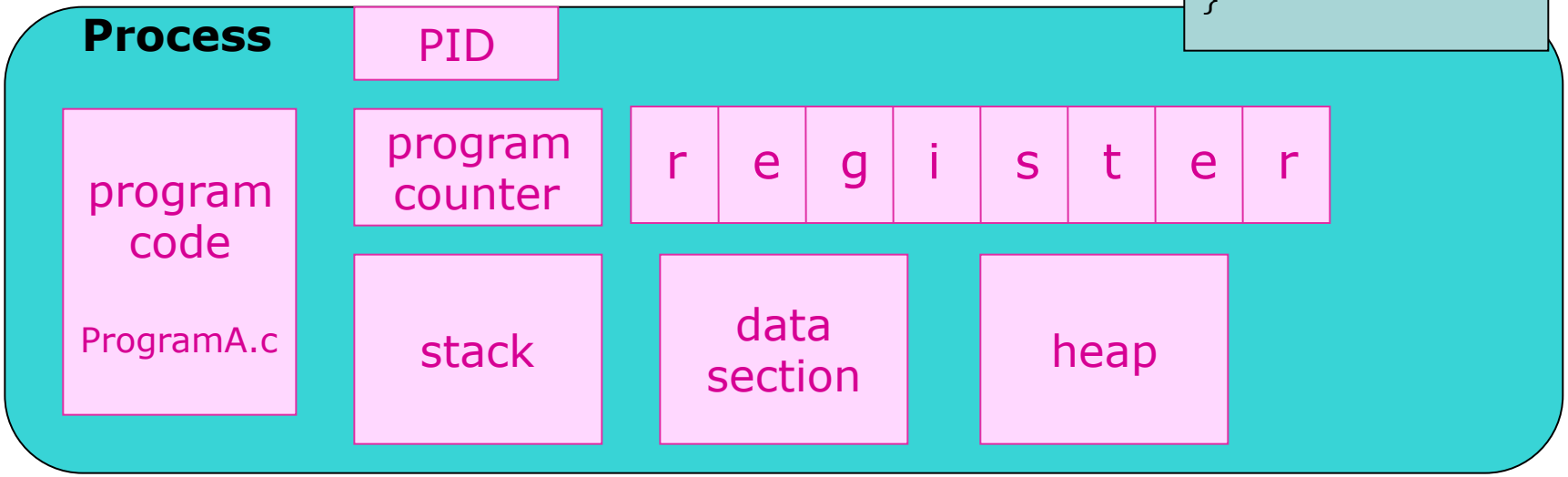
- Each of the functions in the exec family replaces the current process image with a new process image.
- The new image is constructed from a regular, executable file called the new process image file. This file is either an executable object file or a file of data for an interpreter.
- There is no return from a successful call to one of these functions because the calling process image is overlaid by the new process image.

exec family: execute a program

- Before calling exec

```
//ProgramA.c

int main()
{
                :
//want to execute "ls"
execlp("/bin/ls", "ls", NULL);
                :
}
```

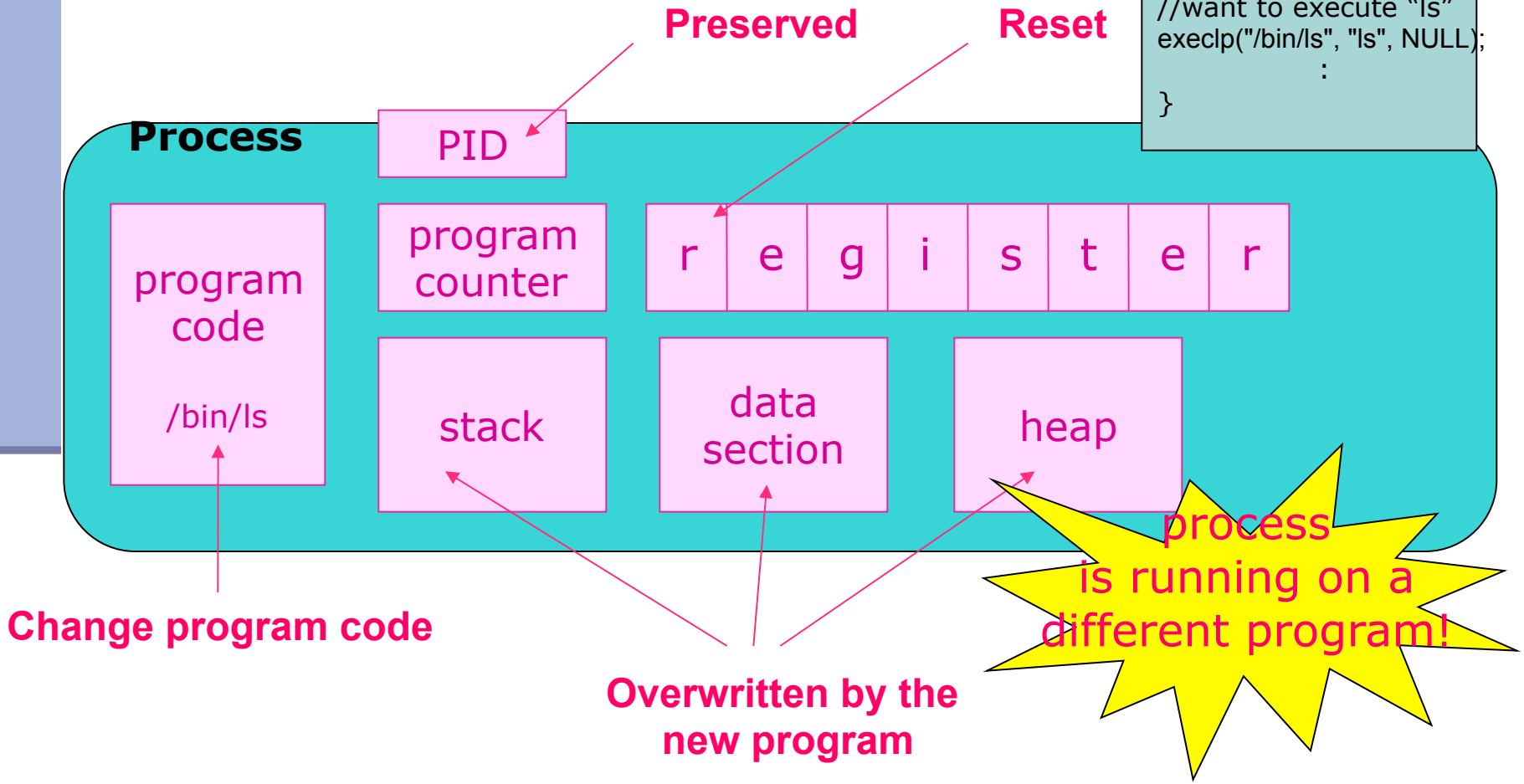


exec family: execute a program

- While calling exec

```
//ProgramA.c

int main()
{
                :
//want to execute "ls"
execlp("/bin/ls", "ls", NULL);
                :
}
```



execlp()

- execlp() initiates a new program in the same environment in which it is operating.
- An executable and arguments are passed to the function.
  - int execlp(const char **file*, const char **arg0*, ..., const char **argn*, char * /*NULL*/);
  - e.g. execlp("/bin/ls", "ls", NULL);
- It will use environment variable **PATH** to determine which executable to process. Thus a fully qualified path name would not have to be used.

execlp()

- What is the **PATH** environment variable?
  - The locations for the exec call to search for the executables.
  - e.g.1 execlp("/bin/ls", "ls", NULL);
  - e.g.2 execlp(“ls”, “ls”, NULL)
    “PATH=/bin;/usr/bin”.
  - Then, the execlp call will only search the specified two locations, /bin and /usr/bin, for the executable named “ls”.
- PATH is stored inside .cshrc

exec family: execute a program

- There is no return from a successful call!
- Because the calling process image is overlaid by the new process image

```
//ProgramA.c

int main()
{
        :
   //execute "ls"
   execlp("/bin/ls", "ls", NULL);

   printf("successful!");
}
```

exec family: execute a program

- There is no return from a successful call!
- Because the calling process image is overlaid by the new process image

```
//ProgramA.c

int main()
{
        :
   //execute "ls"
   execlp("/bin/ls", "ls", NULL);

   printf("successful!");
}
```

This WILL NOT be printed if "ls" is successfully executed!

exec family: execute a program

- There is no return from a successful call!
- Because the calling process image is overlaid by the new process image

```
//ProgramA.c

int main()
{
        :
   //execute "ls"
   execlp("/bin/abc", "abc", NULL);

   printf("command not found!");
}
```

exec family: execute a program

- There is no return from a successful call!
- Because the calling process image is overlaid by the new process image

```
//ProgramA.c

int main()
{
        :
   //execute "ls"
   execlp("/bin/abc", "abc", NULL);

   printf("command not found!");
}
```

This WILL be printed if "abc" is NOT successfully executed!

Useful links

- For more about fork, exec, and process control: http://www.yolinux.com/TUTORIALS/ForkExecProcesses.html
- Use "**info**" to learn more details from UNIX Manual Pages for fork, exec, and execlp