COS40003 Concurrent Programming

Lecture 2: Process: Concept and API

CRICOS 00111D
TOID 3059

Outline

- What is a process?
- Why we need processes?
- How a process is working?
- Process APIs

What is a process?

Let us start with program VS process

CRICOS 00111D
TOID 3059

What is a program? (narrow sense)

- Program
  - Binary machine code, a sequence of machine-language instructions stored in a file
- Run a program
  - Load a list of machine-language instructions into memory, and read the instructions into CPU, and have the processor (CPU) execute the instructions one by one

(Note that, this is a simple abstraction. (a) cache ignored; (b) multi-instruction optimization ignored)

What is a program? (broad sense)

- General computer code
  – What are you doing? I'm writing C/Java programs.

- Process
  – Your system is slow, because there are too many programs.

(Actually, you want to say: there are too many processes, i.e. you are running too many programs.)

What is a process?

- A Process
  - A program in action

To sum up:

- Program
  - Code (a set of instructions), which is static
- Process
  - Current program and its activity

Outline

- What is a process?
- Why we need processes?
- How a process is working?
- Process APIs

Why we need processes?

i.e. Why people came up with the concept “processes”?

CRICOS 00111D
TOID 3059

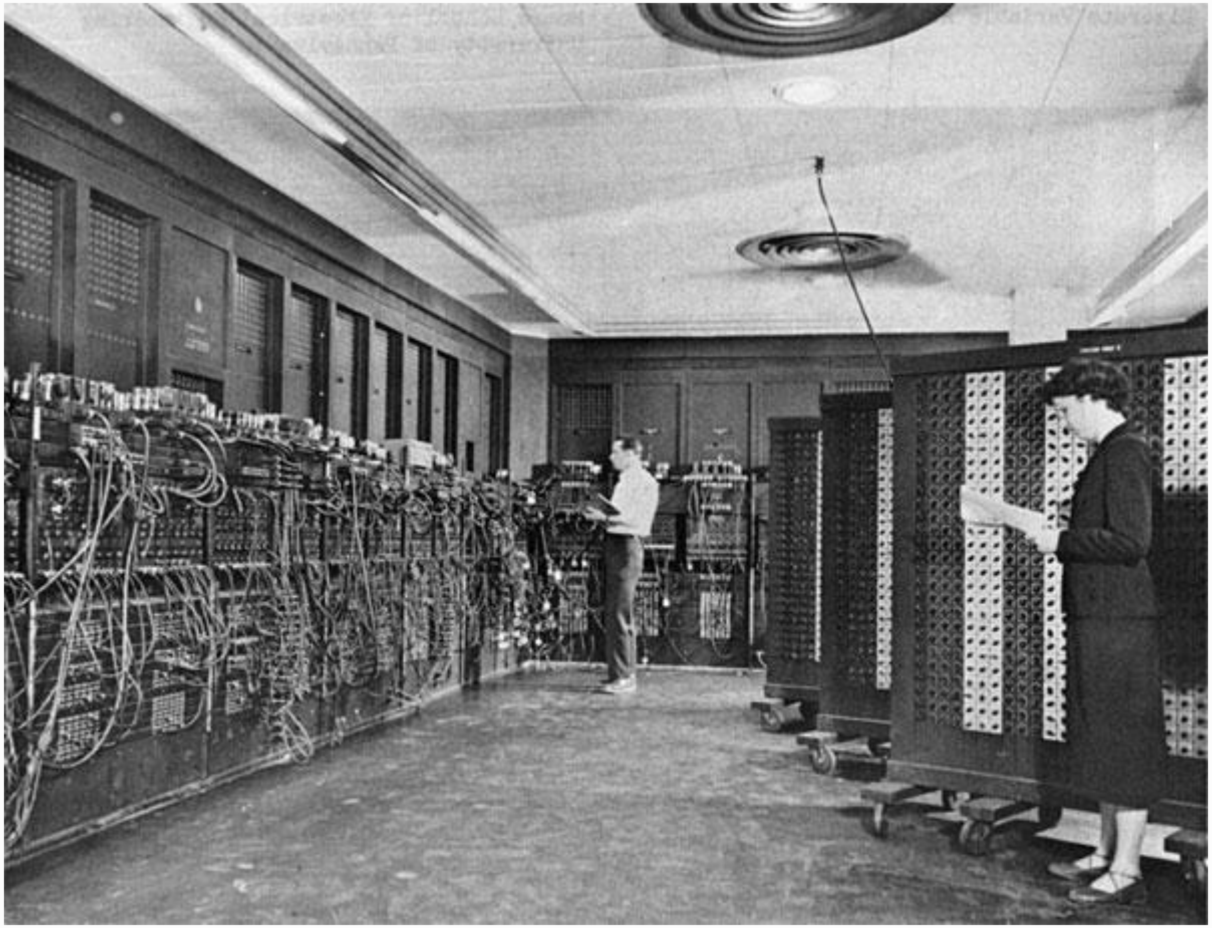
Why people came up with processes?

Review Computer History

- Period 1:   single command
  - Computer is very simple.
  - A user typed a command, and then a computer did an operation. The user stopped, the computer stopped

→ low efficiency ☹

ENIAC: first digital computer



Ref: https://www.computerhope.com/issues/ch000984.htm

Why people came up with processes?

- Period 2: - batch processing
  - Write commands as a list (a program), let computers run → ☺

One old Apple computer

https://apple2history.org/history/ah02/

Question: use one word to describe the computer

Cute！

Why people came up with processes?

- Period 2: - batch processing
  - Write commands as a list (a program), let computers run → ☺
  - People wrote different programs and let a computer run in turn
  - When program A is running, and program A needs to read/write a lot of data (I/O operations), CPU is waiting for I/O to be finished. We are wasting CPU time. → ☹
  - Can we let program B use CPU, while program A is doing I/O ?

Why people came up with processes?

- Period 2: - batch processing
  - Before, the computer ran one program at a time, i.e. the memory held one program only. Now we want the memory to hold multiple programs!
  - The questions come:
    - How to identify different programs, code/data segments, etc?
    - How to restore running a program after its suspension is over.

Why people came up with processes?

- Period 3: - Process is invented!
  - A program is encapsulated in a process.
  - Every process is allocated a chunk of memory, and can only use its own memory. The state of the process and the resources the process is using can be saved.
  - When switched back, easily restore the previous state and continue
  - Different processes will not affect each other.

→ ☺

Outline

- What is a process?
- Why we need processes?
- How a process is working?
- Process APIs

How a process is working?

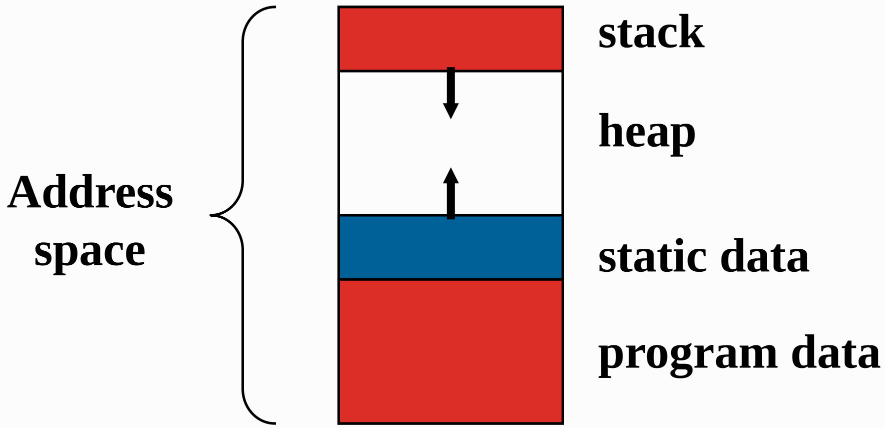
CRICOS 00111D
TOID 3059

How a process is working?

- Process consists of:
  - An image of a program
  - memory (program instructions, static data, heap and stack)
  - CPU state (registers, program counter(PC), stack pointer(SP), etc)
  - operating system state (opened files, accounting statistics, etc)

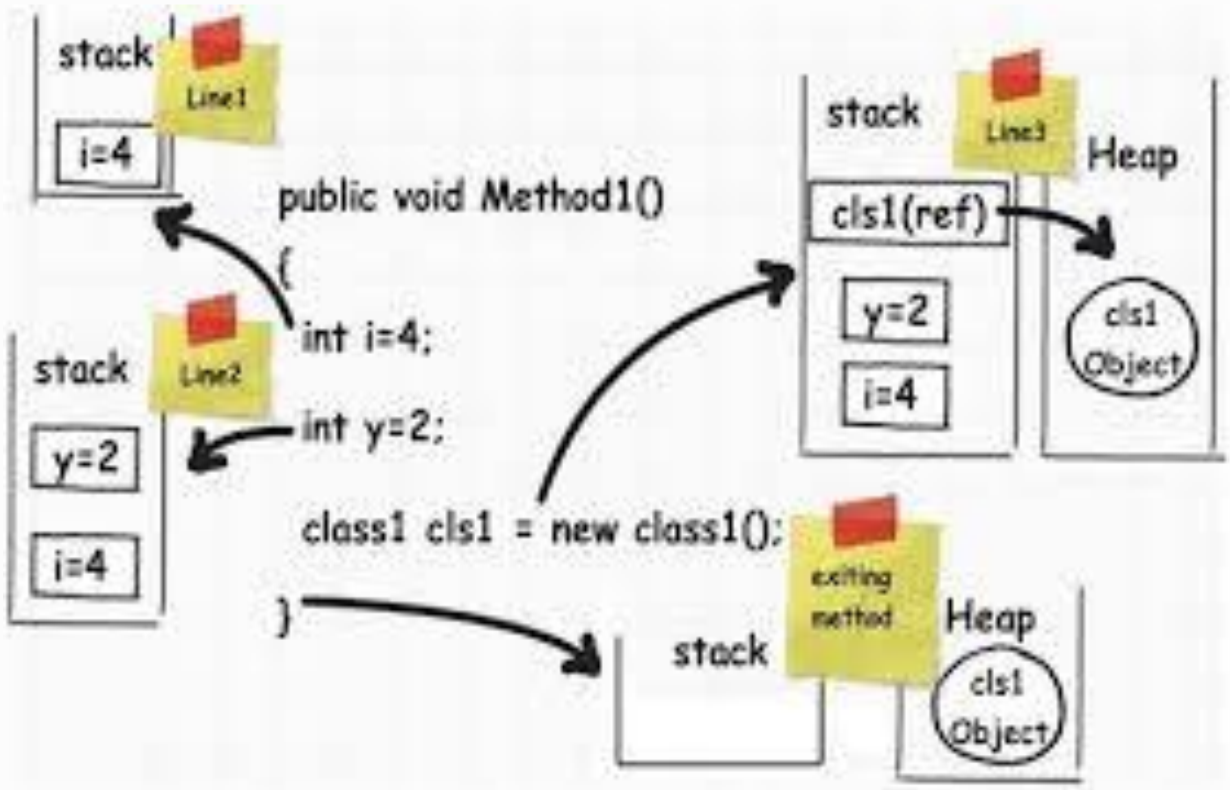
Process Address Space (memory)

- Each process runs in its own virtual memory *address space* that consists of:
  - *Program* – the program code (usually read only)
  - *Stack space* – used for function and system calls
  - *Data space* – variables (both static and dynamic allocation (heap))



- Invoking the same program multiple times results in the creation of multiple distinct address spaces

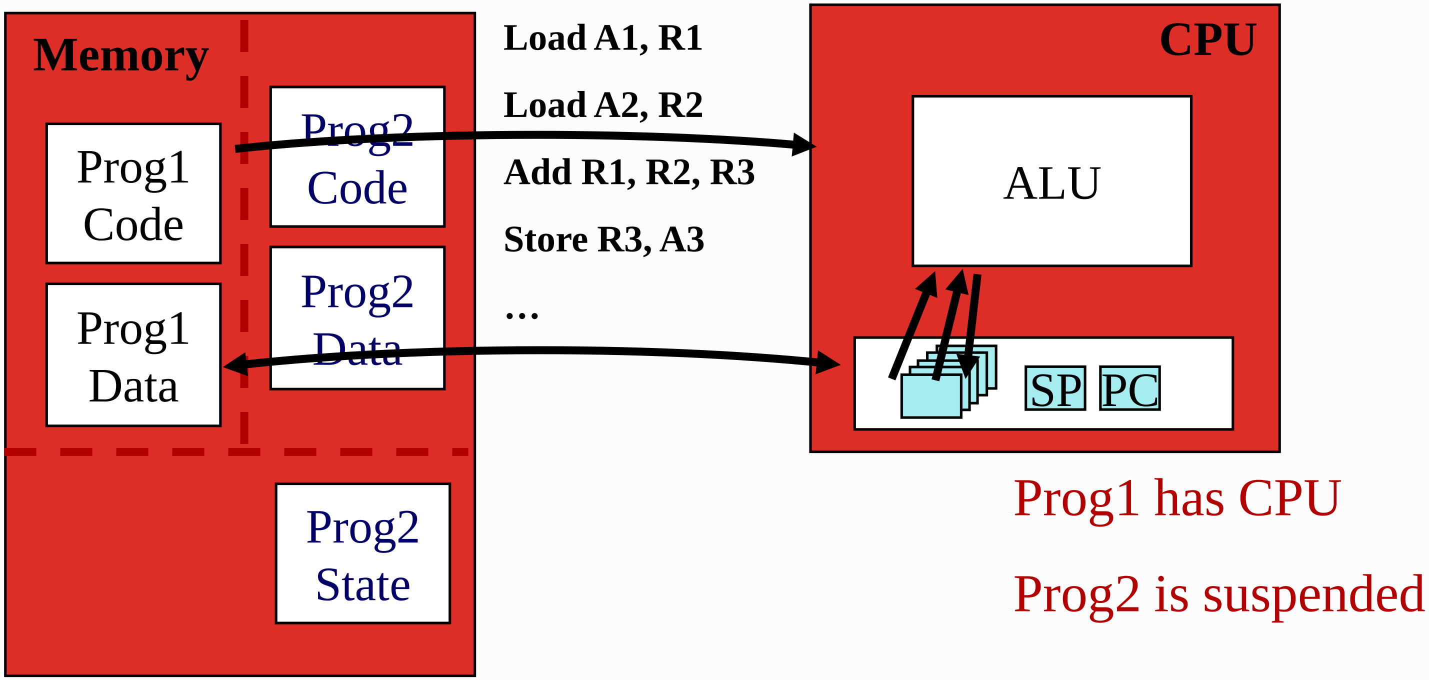
Stack memory and heap memory



https://www.codeproject.com/Articles/76153/Six-important-NET-concepts-Stack-heap-value-types#Stack%20and%20Heap

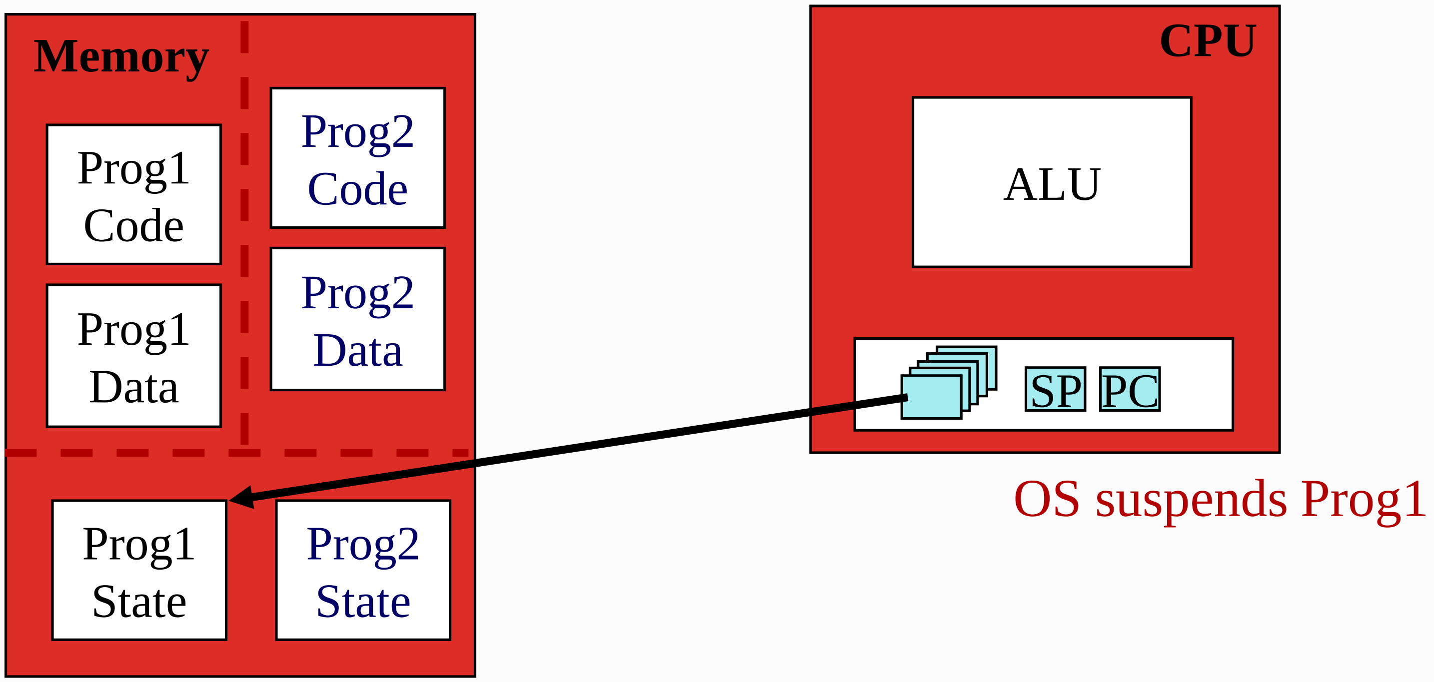
Context switch (phase 1)

- CPU is running Prog1



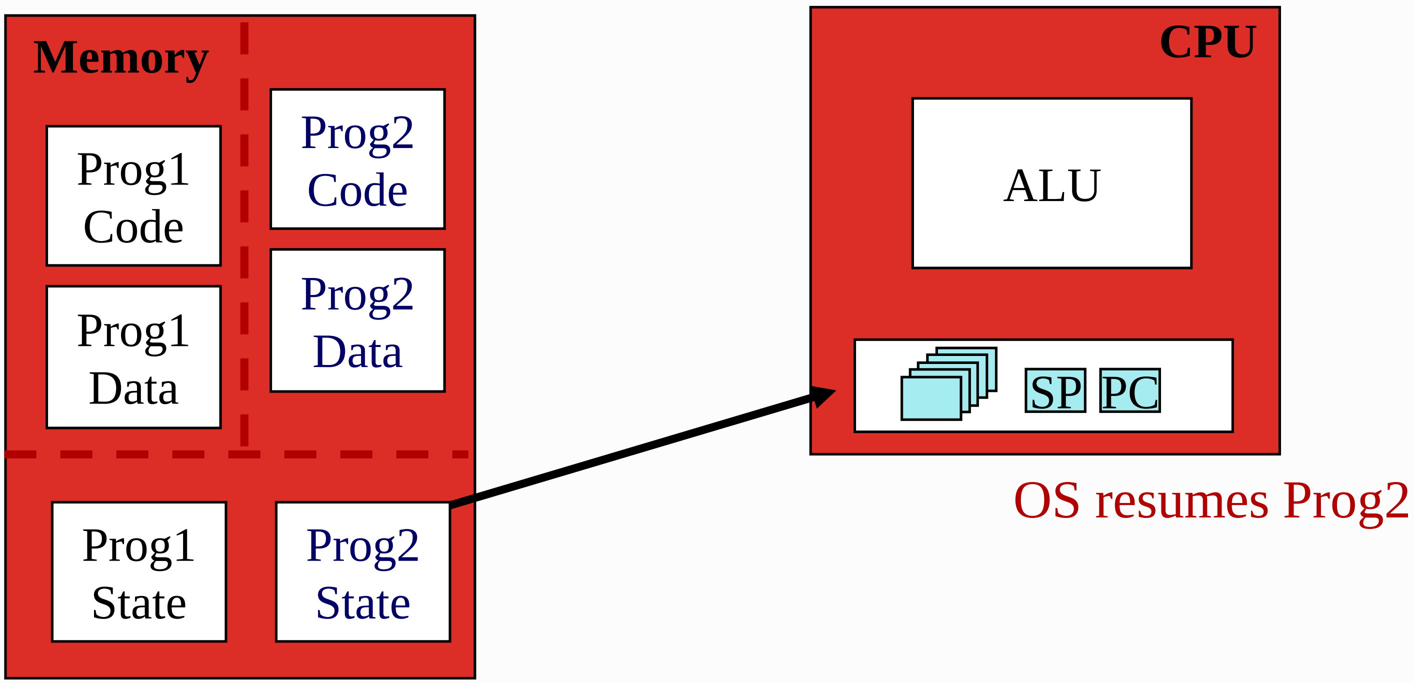
Context switch (phase 2)

- Save registers, program counter, stack pointer of Prog1



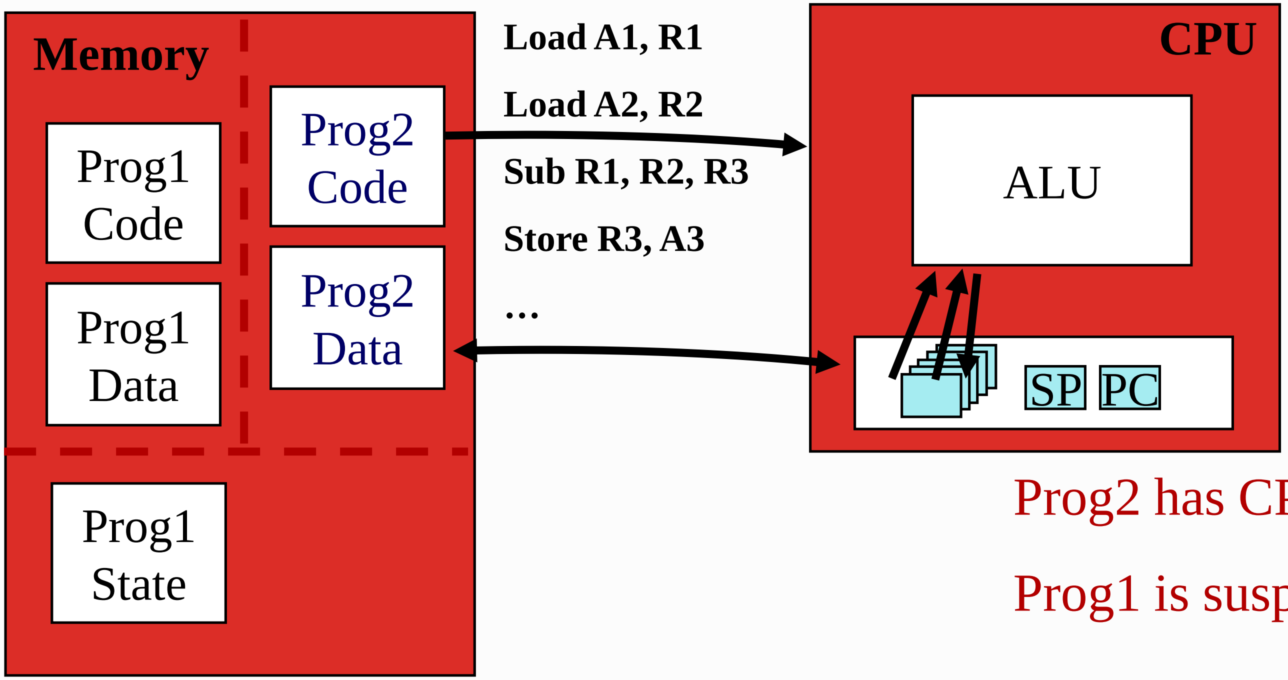
Context switch (phase 3)

- Restore registers, program counter, stack pointer of Prog2



Context switch (phase 4)

- Prog2 starts to run



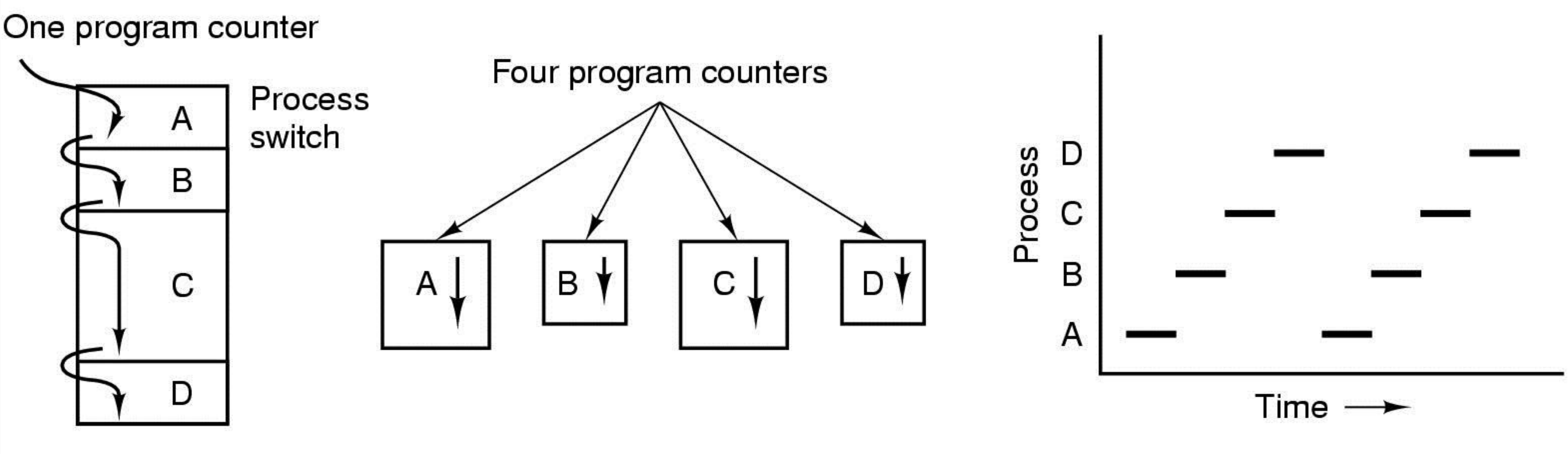
Prog2 has CPU

Prog1 is suspended

The Process Abstraction

A, B, C ,D are four processes

- (a) physical view
- (b) logical view
- (c) time-sharing



(a) (b) (c)

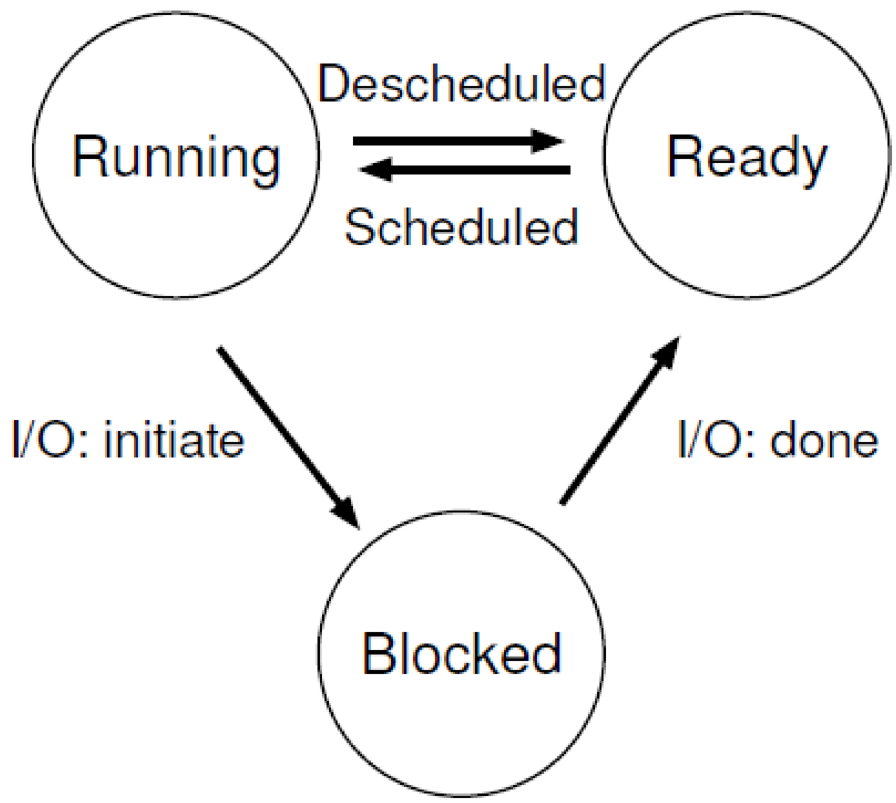
Process States

- **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. (eg., another process is running now)

- **Blocked**: 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. (eg., when a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.)

Process: State Transitions



Process: State Transitions

- A process can be moved between the ready and running states at the discretion of the OS.
- Being moved from ready to running means the process has been scheduled;
- Being moved from running to ready means the process has been descheduled.
- Once a process has become blocked (e.g., by initiating an I/O operation), the OS will keep it as such until some event occurs (e.g., I/O completion); at that point, the process moves to the ready state again

Example 1: Tracing Process State: CPU Only

- Process0 runs first and Process1 is ready, after Process0 finishes, Process1 starts to run.

Time	Process_0	Process_1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process_0 now done
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	Process_1 now done

Example 2: Tracing Process State: CPU and I/O

- Process0 issues an I/O after running for some time. At that point, Process0 is blocked, giving Process1 a chance to run.

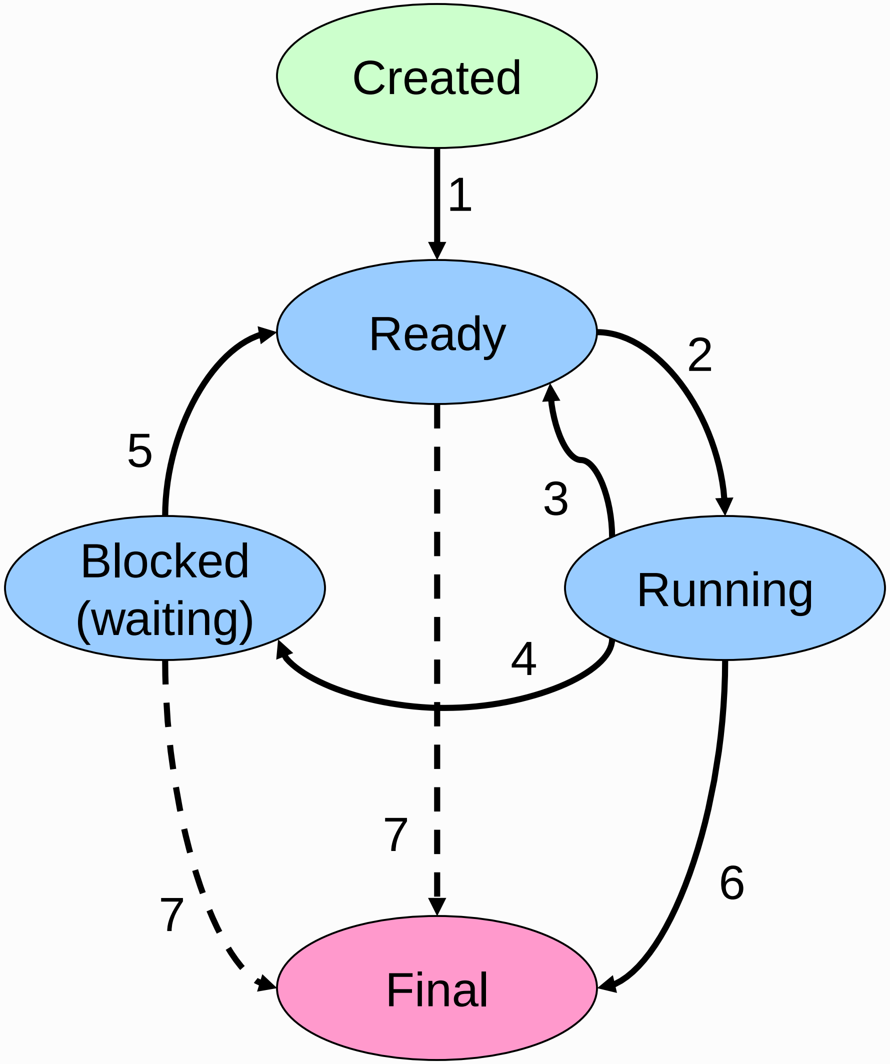
Time	Process_0	Process_1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process_0 initiates I/O
4	Blocked	Running	Process_0 is blocked,
5	Blocked	Running	so Process_1 runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process_1 now done
9	Running	–	
10	Running	–	Process_0 now done

Other process states:

- Initial/Created state
  - When a process is being created.
- Final/Terminated state
  - where it has exited but has not yet been cleaned up (in UNIX-based systems, this is called the zombie state).

  - Why final: this final state can be useful as it allows parent process to examine the return code of the child process and see whether the just-finished child process executed successfully

Process states



- Process in one of 5 states
  - Initial/Created
  - Ready
  - Running
  - Blocked
  - Final/Terminated
- Transitions between states
  - 1 - Process enters ready queue
  - 2 - Scheduler picks this process
  - 3 - Scheduler picks a different process
  - 4 - Process waits for event (such as I/O)
  - 5 - Event occurs
  - 6 - Process exits
  - 7 - Process ended by another process

To realize time-sharing

- Context switch
- Scheduling (to be discussed in Week 3)
  - A **scheduling policy** in the OS will make this decision, likely using
    - historical information (e.g., which program has run more over the last minute?)
    - workload knowledge (e.g., what types of programs are running)
    - performance metrics (e.g., is the system optimized for interactive performance, or throughput?)

Outline

- What is a process?
- Why we need processes?
- How a process is working?
- Process APIs

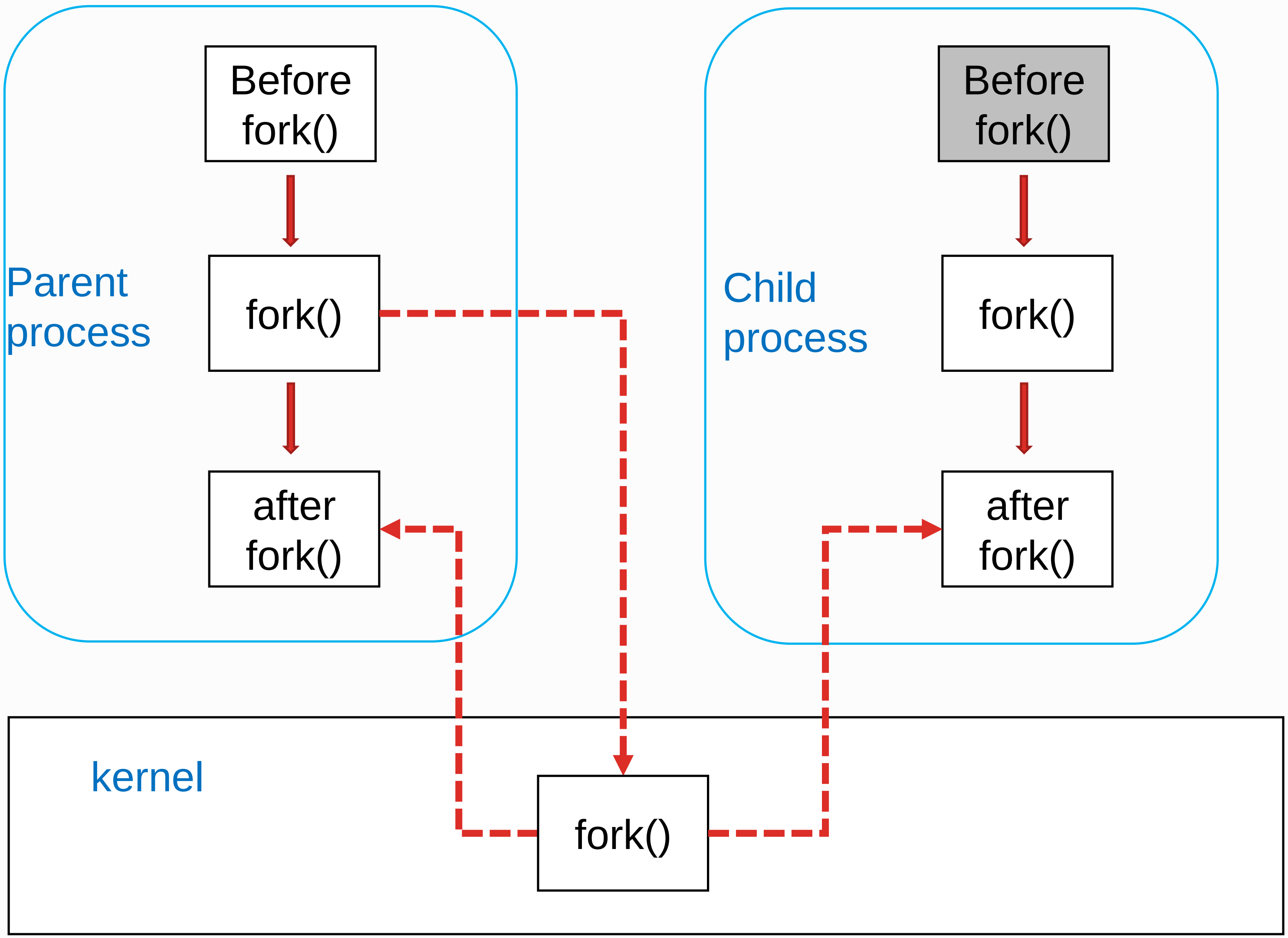
Process APIs

- fork()
- wait()
- exec()

How to create a new process?

- fork() – create a new process by duplication (clone)
- How?
  - 1. allocates a new chunk of memory and kernel data structures
  - 2. copies the original process into the new process
  - 3. adds the new process to the set of running processes
  - 4. returns control back to *both* processes

Fork() flow


Before
fork()
Parent
process
fork()
after
fork()
Before
fork()
Child
process
fork()
after
fork()
kernel
fork()

fork()

- Distinguishing Parent from Child

  - int pid;
  - pid = fork();

  - if (pid == 0)

            We are in the child process;

        else

            We are in the parent process;

```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; 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)
        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>
```

How does the parent wait for the child to exit?

- wait()
  - 1. pauses the calling program until a child finishes;
  - 2. retrieves the value the child process has passed to exit;

  - pid = wait(&status);

  - pid: the id terminated process;

    -1, if there is an error;

  - status:
    - Success
    - Failure:
      - eg., running out of memory
    - Death:
      - killed by a Unix signal

```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 wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
                rc, wc, (int) getpid());
    }
    return 0;
}
```

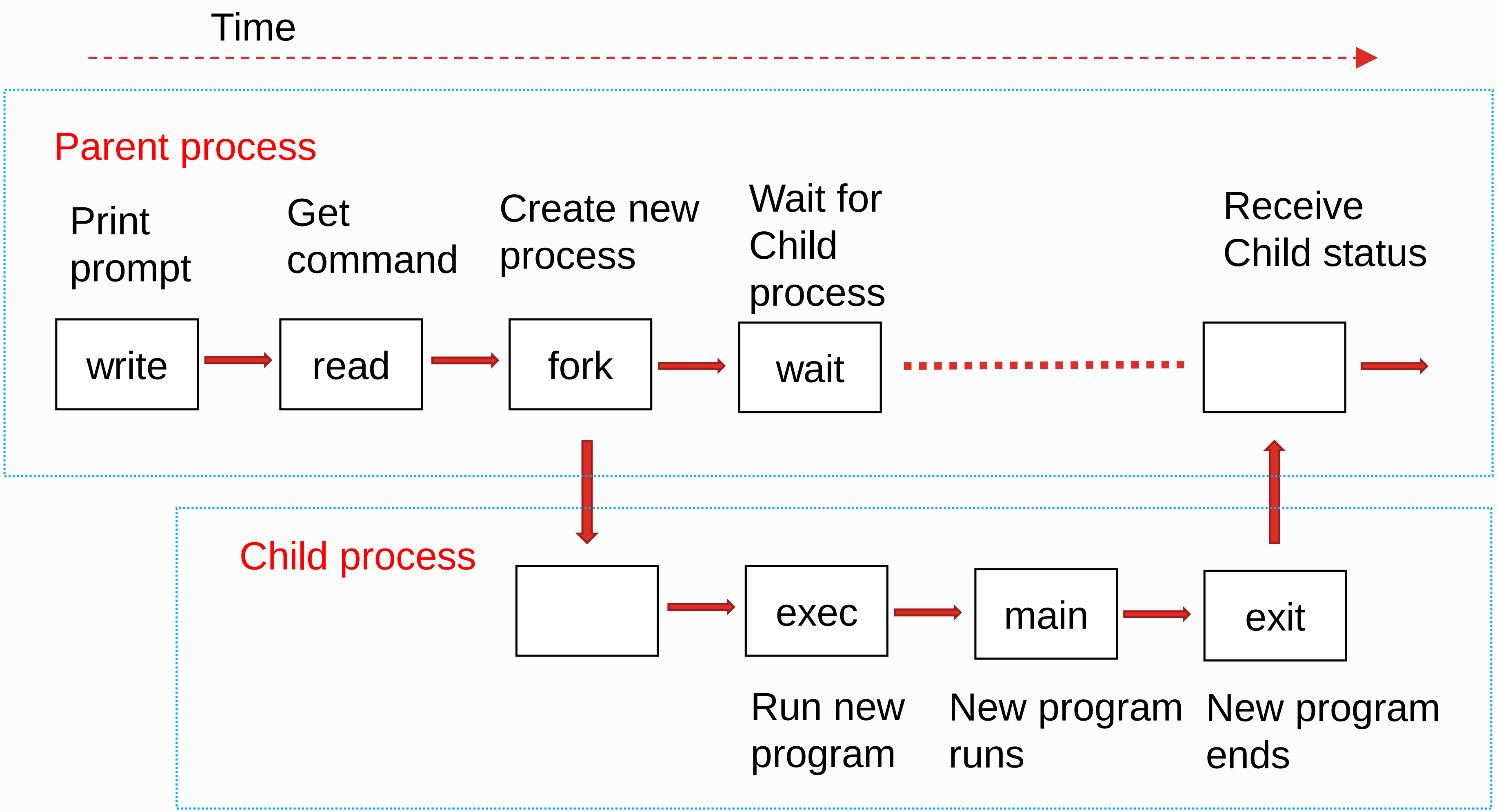
```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (wc:29267) (pid:29266)
prompt>
```

How does a process run a program?

- exec() (execl, execle, execlp, execv, execvp, execvP)

- The OS loads the new program into the current process, replacing the code and data of that process.
  - The exec system call clears out the machine-language code of the current program from the current process.
  - Put the code of the program named in the exec call and run that new program.
  - Exec changes the memory allocation of the process to fit the space requirements of the new program
  - Process is the same, the contents are new.
  - Analogy - Brain Transplant

How the shell runs a program



```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 wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
                rc, wc, (int) getpid());
    }
    return 0;
}
```

Exercise

- How many lines of output this program will produce: (ignore syntax error)

```
main(){
        printf("my pid is %d\n", getpid() );
        fork();
        fork();
        fork();
        printf("my pid is %d\n", getpid() );
}
```

Acknowledgement

- Chapter 4-5
  - Operating Systems: Three Easy Pieces

- 2.ppt
  - Intro to Operating System at Portland State University
  - by Jonathan Walpole

Questions?

CRICOS 00111D
TOID 3059