

Processes

Instructor: William Zheng

Email:

william.zheng@digipen.edu

PHONE EXT: 1745

Outline

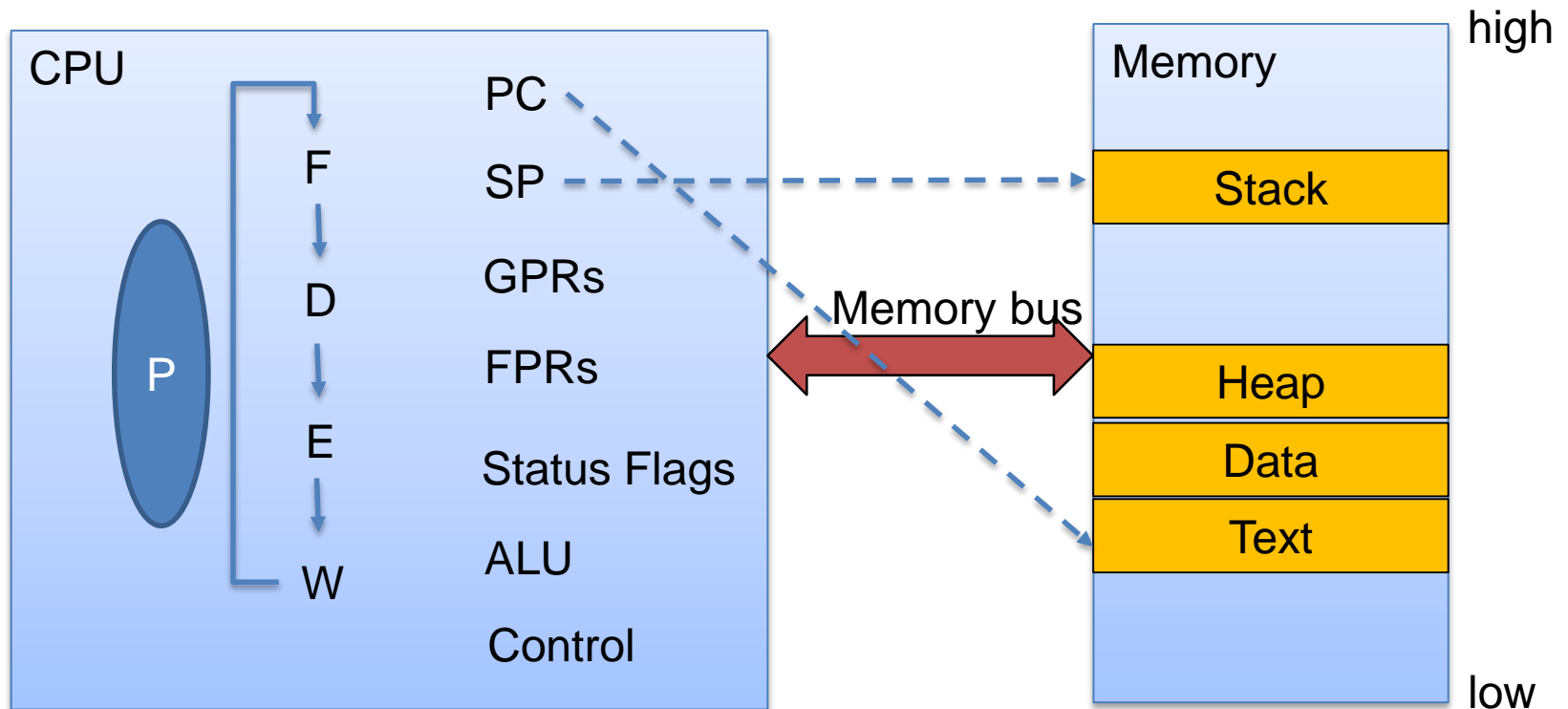
- “running programs” (Why inverted commas?)
- **What is in a Process?**
 - A snapshot view
 - Context saving and restoring
- Process States and PCB
- Operations on process
 - Process Scheduling
 - Process Creation
 - Process Termination

Snapshot view - I

- Assume a process using the CPU at a point in time.
 - What are the “resources” that the running program would be using?

Snapshot view - II

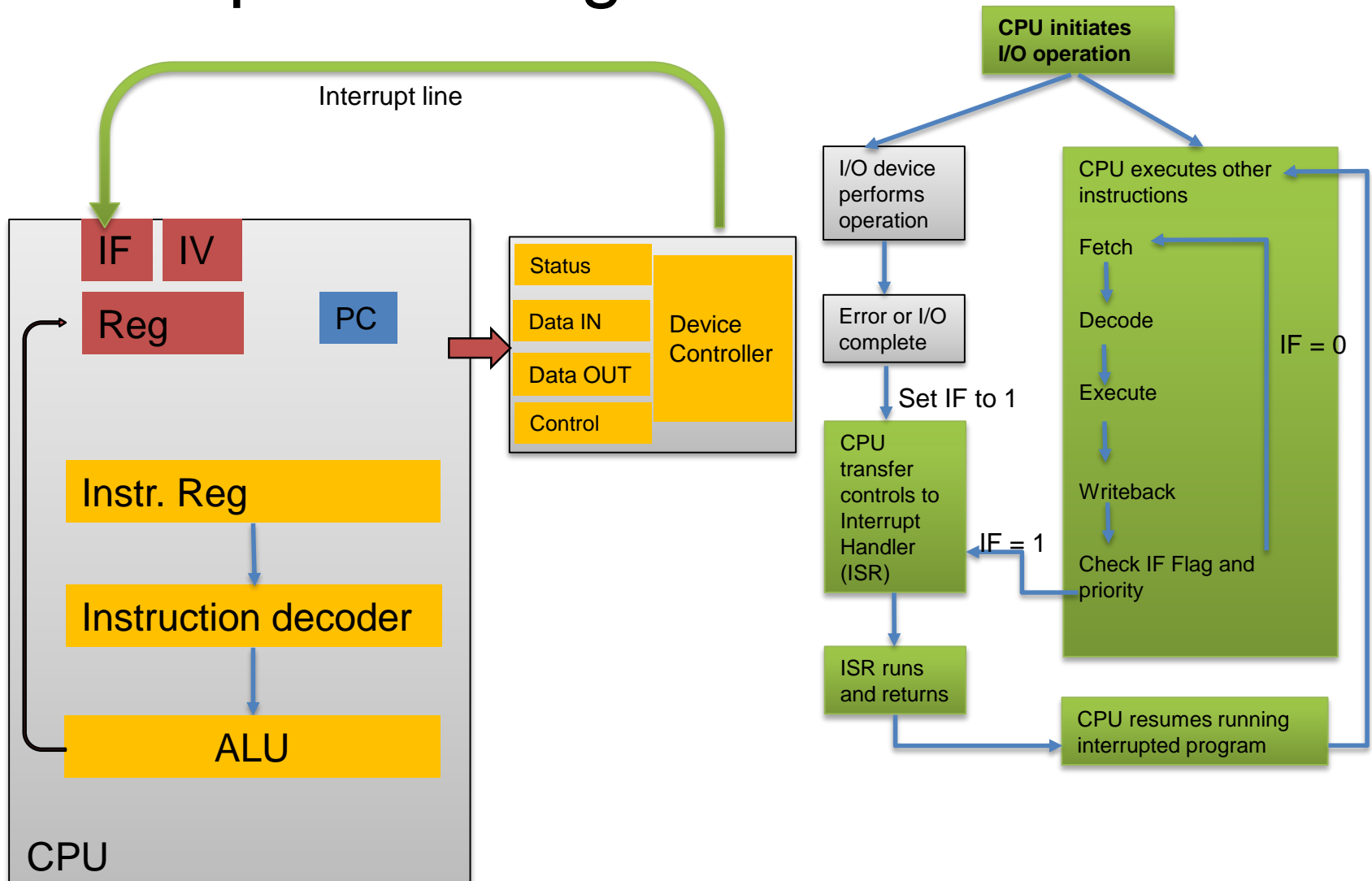
- Process is running. So at least using Memory and CPU.
- What is it using in ?



Context of a process

- User registers (including FPs, PC, SP).
- Status flags and other special registers (more later)
- Why registers only?
 - How about ALU, Control Unit and all the stuff in Memory?
 - Treat ALU and Control Unit like a black box
 - Registers are the input into ALU and Control Unit
 - Save the registers values and restore them later.
 - Same register values ensure the same behaviour from ALU and Control Unit.
 - Memory taken care of by OS memory management.

Interrupt Handling - revisited



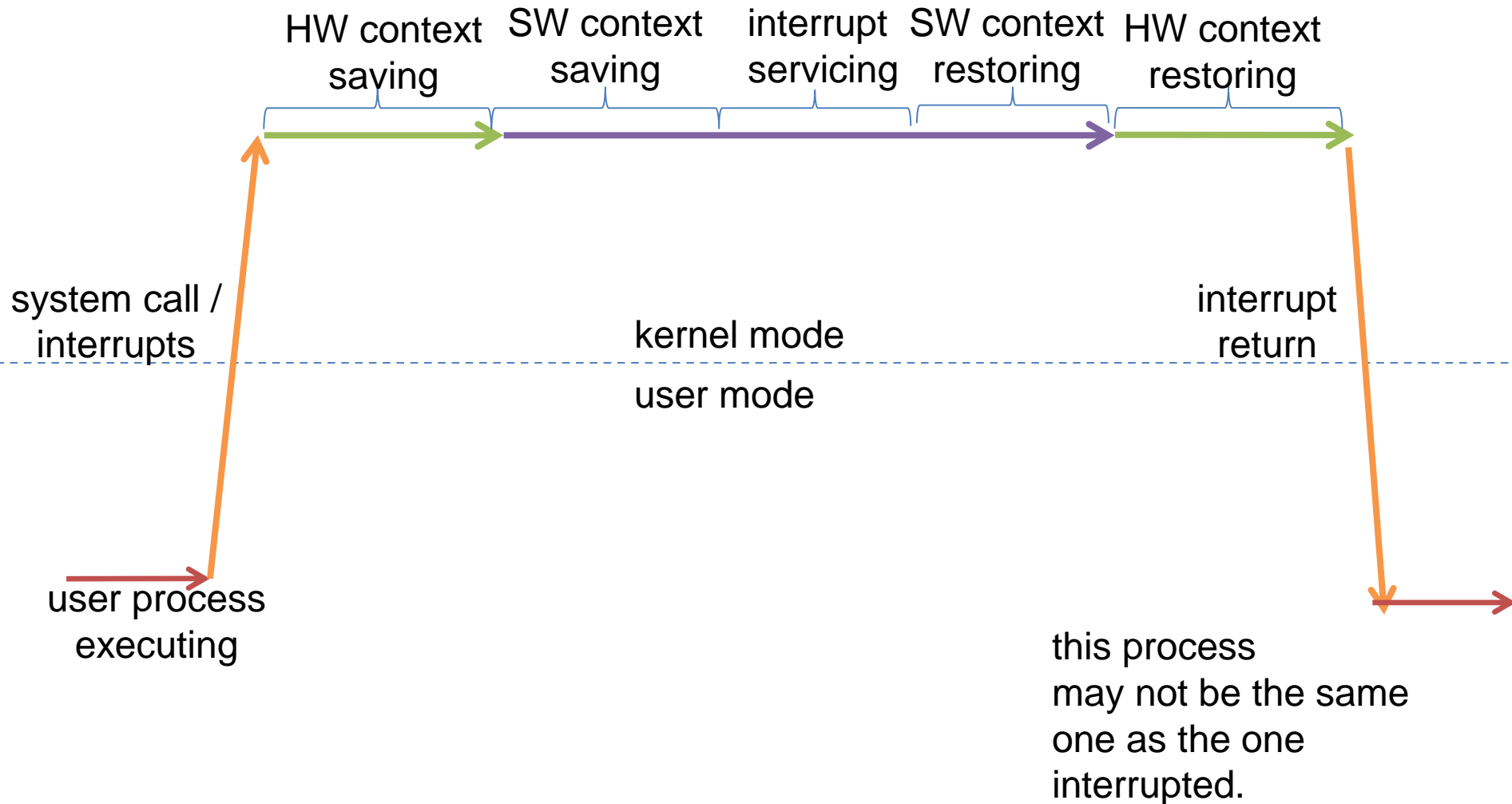
HW context saving / restoring

- Automatic (no instructions involved)
- What if the CPU don't support HW saving?
 - HW MUST support some kind of saving.
 - At least HW must save the PC and SP.
 - Too late for ISR to save PC and SP by the time ISR is running.
- Where does it save to?
 - HW reads the SP
 - Save the registers to the stack.
- Restoring is the reverse of saving.

SW context saving/restoring

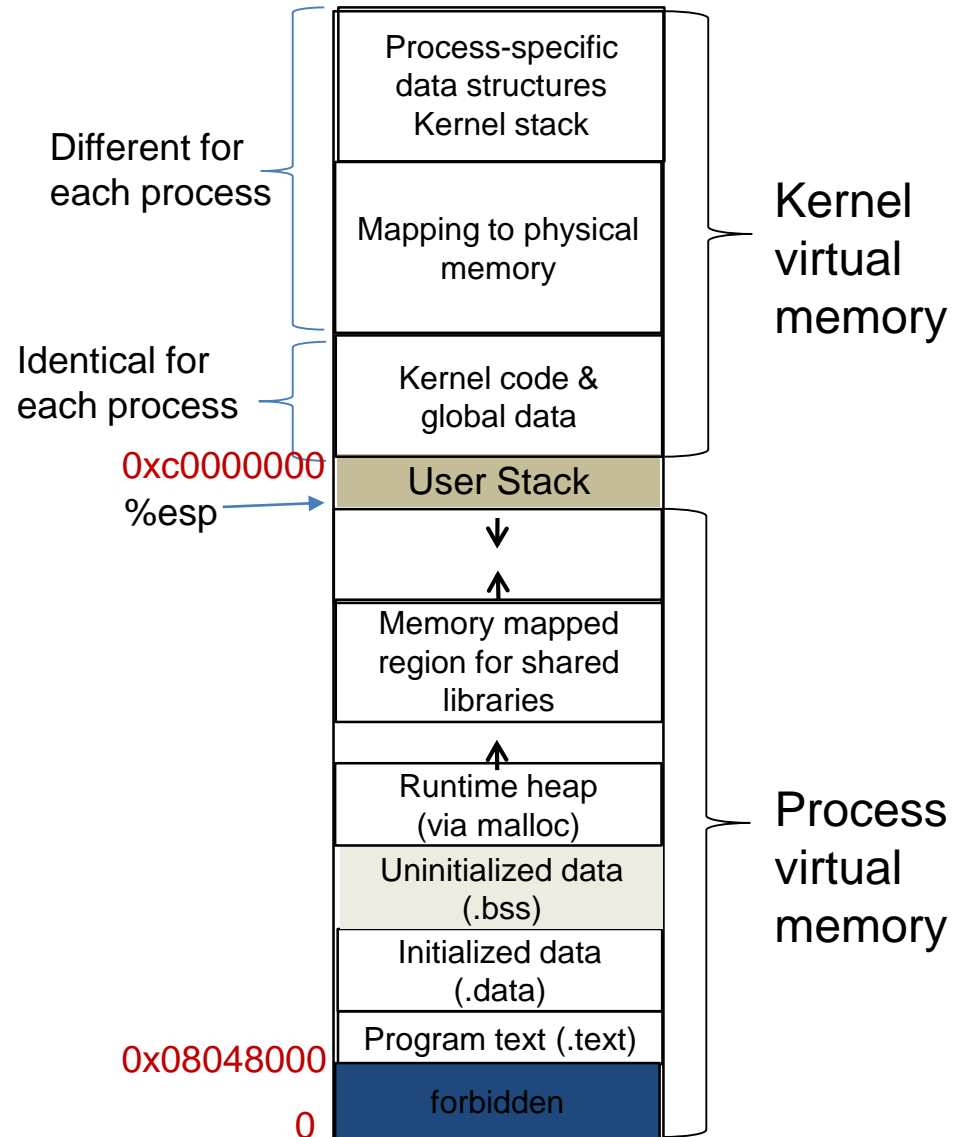
- Usually the 1st thing done in ISR
- Sometimes HW context saving doesn't save all the registers!
 - e.g., x86 only saves the GPR registers but not the FPRs.
 - FPRs not saved unless used
- SW context saving : selective vs HW saves the registers regardless whether required or not!

System call/Interrupts Revisited



Example of Virtual Memory Layout

- Process context
 - Syscall
 - Save state on its kernel stack
- Interrupt context
 - User mode → kernel mode
 - Kernel mode
 - Save on kernel mode stack for intrpt (intrpt stack)



x86 Interrupt

before ISR code execution

- Save current stack pointer
- Save current program counter
- Save current processor status word (condition codes)
- Switch to kernel stack; put SP, PC, PSW on stack
- Switch to kernel mode

Vector through interrupt table

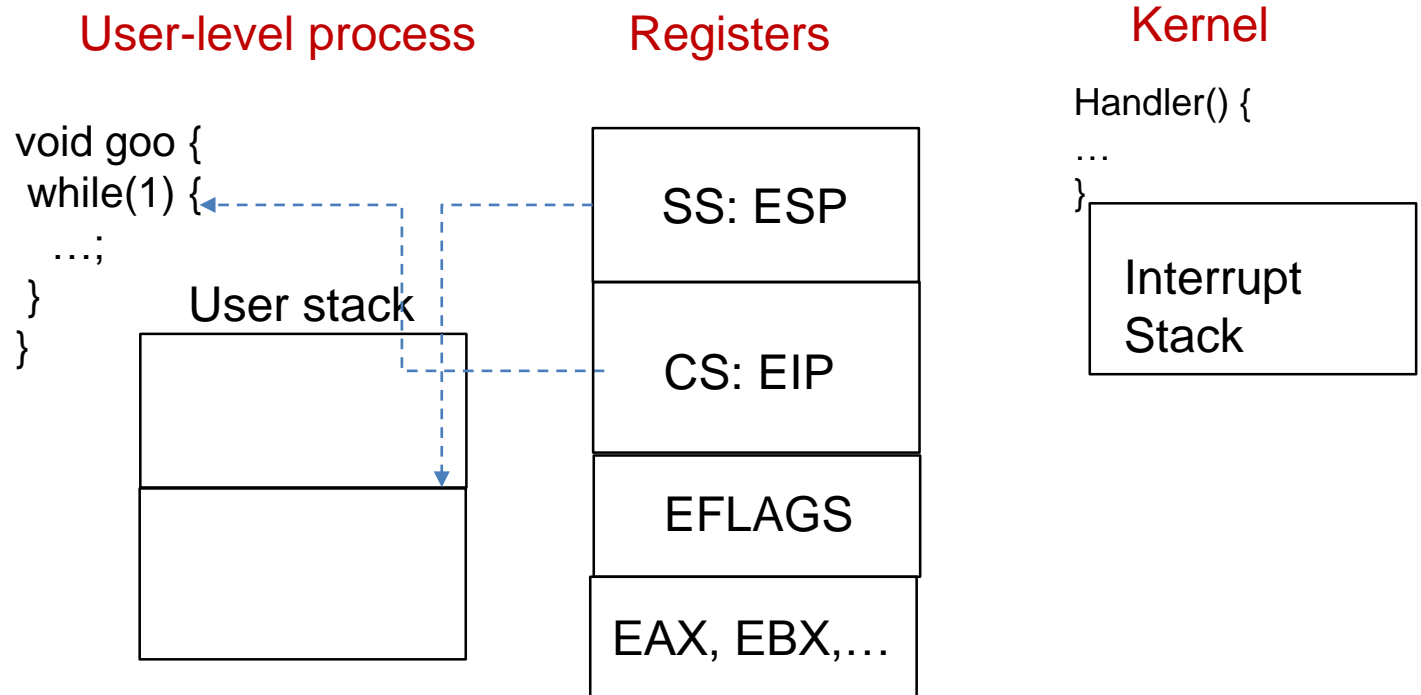
ISR saves registers it might clobber

Execution of handler code

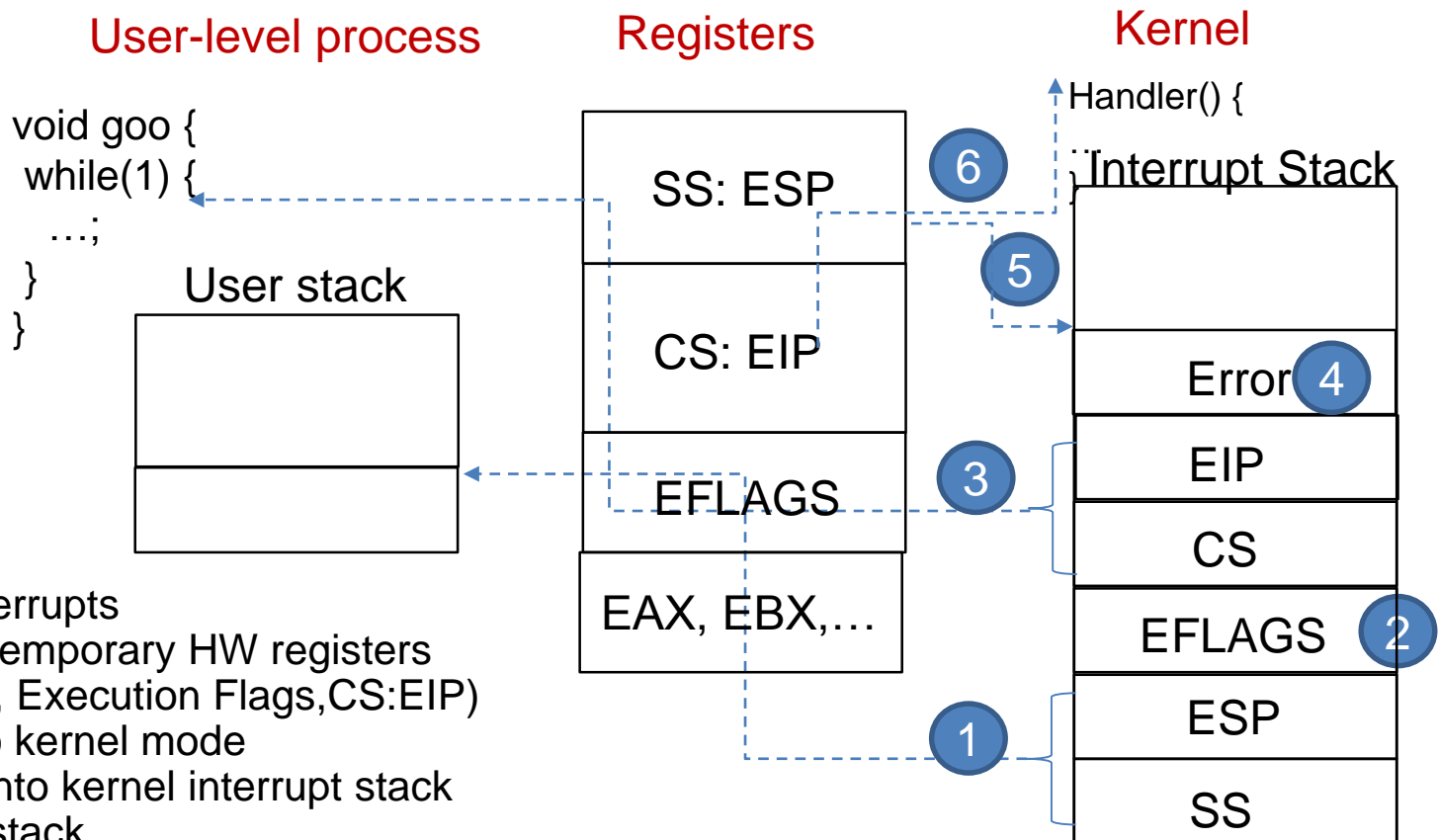
Return / Restoring of process state (IRET instruction)

Before Interrupt

- pointers: segment base+offset
- Current instructions: CS+EIP
- Current privilege: low-order bit of the ESP

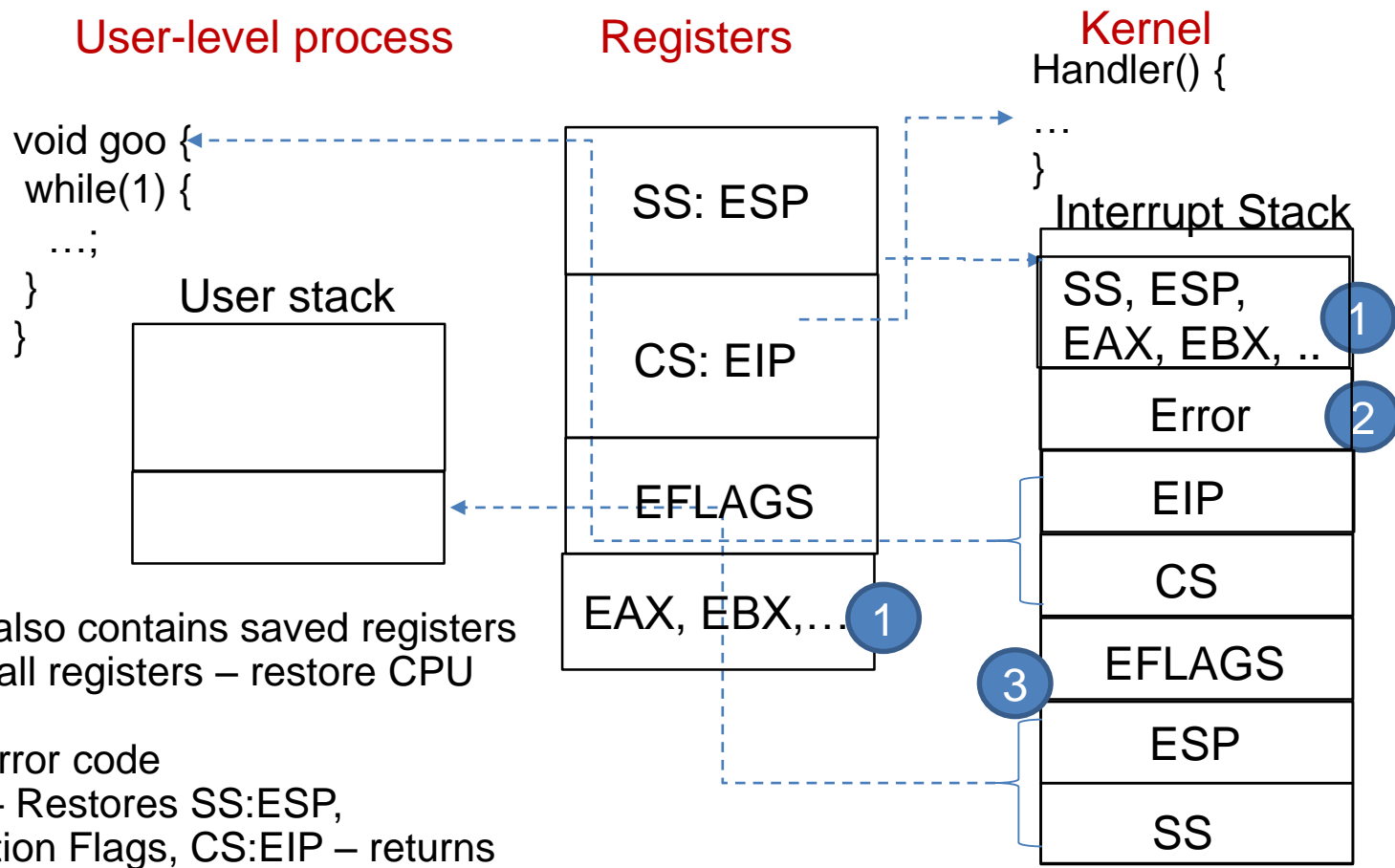


During Interrupt

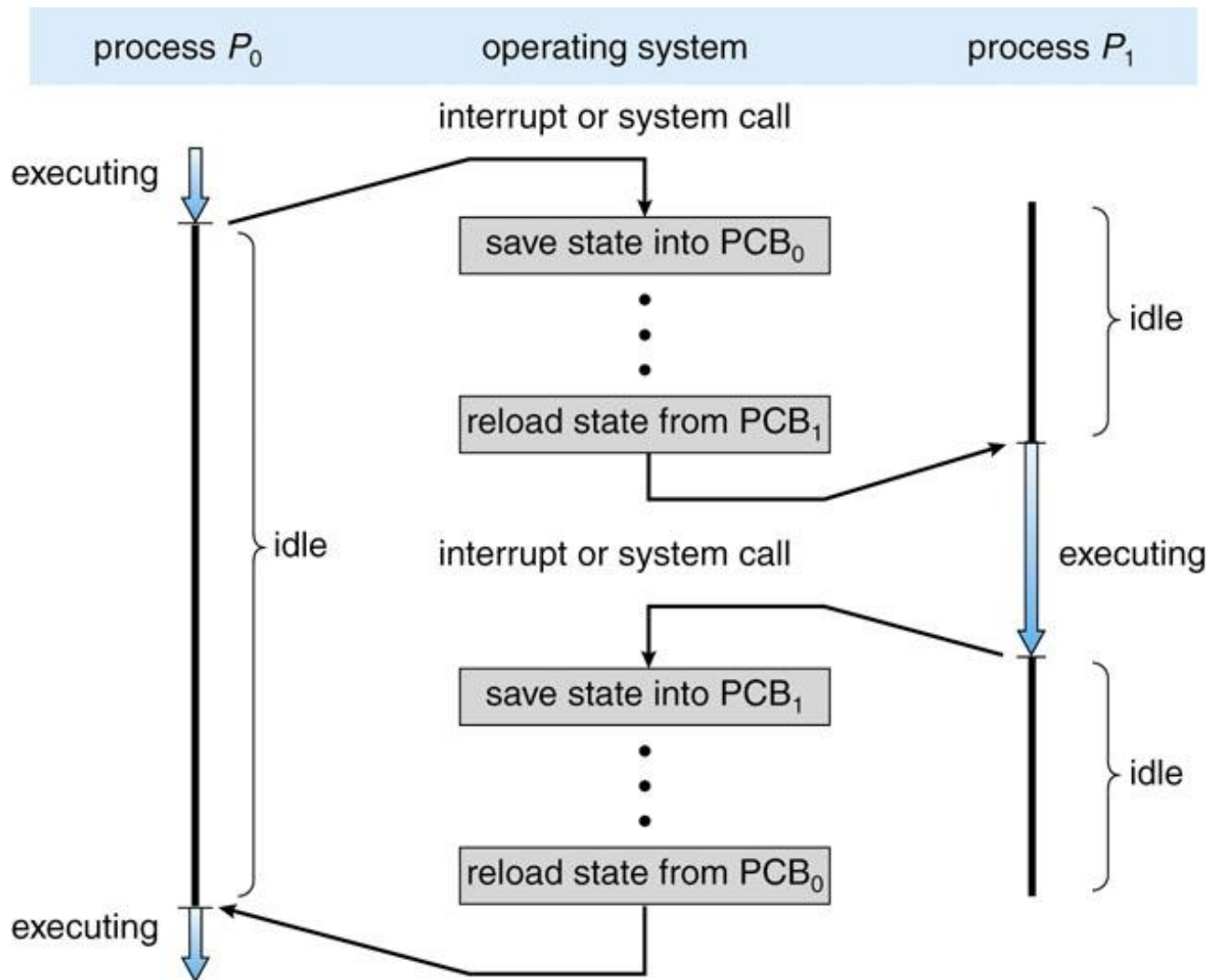


- Mask interrupts
- Save to temporary HW registers (SS:ESP, Execution Flags, CS:EIP)
- Switch to kernel mode
- Switch onto kernel interrupt stack
- Push to stack
- Save error code
- Invoke interrupt handler

After Interrupt



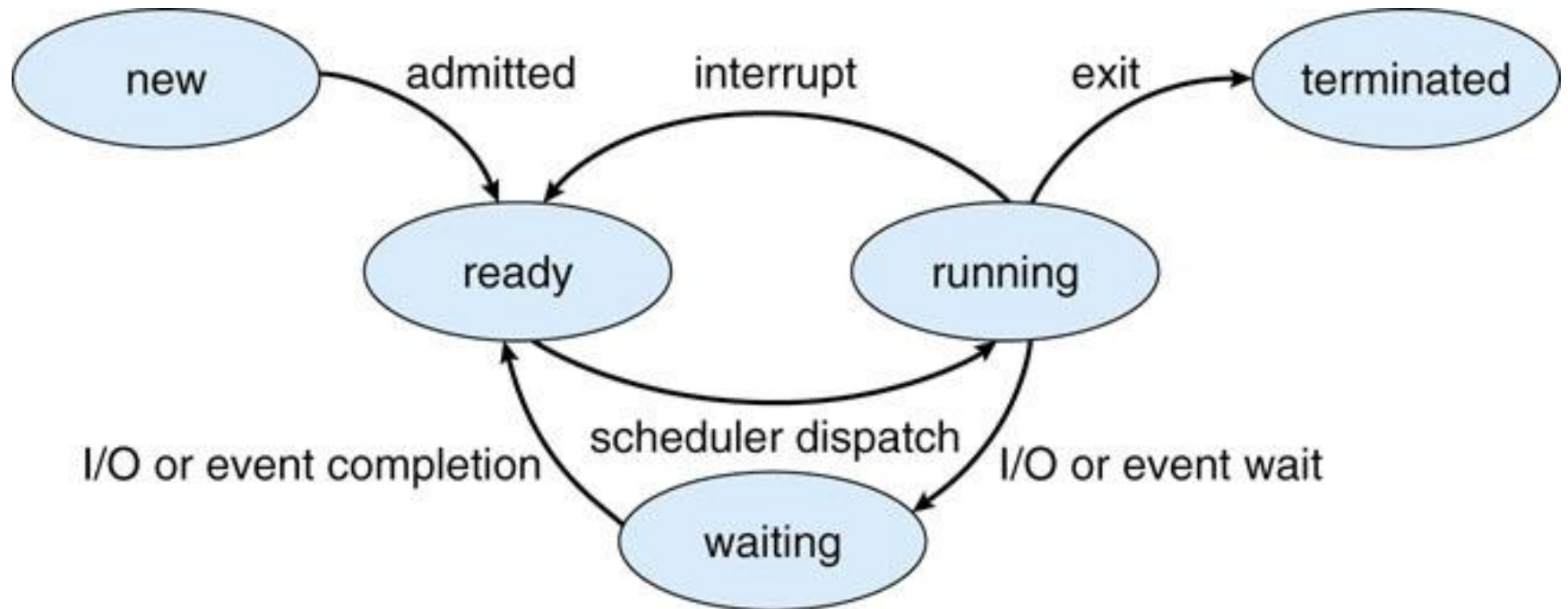
Context Switch



Outline

- “running programs” (Why inverted commas?)
- What is in a Process?
 - A snapshot view
 - Context saving and restoring
- **Process States and PCB**
- Operations on process
 - Process Scheduling
 - Process Creation
 - Process Termination

Process state transitions



Process state

- New – Process being prepared for loading and execution.
- Running – Process using the CPU.
- Waiting – Process waiting for some I/O event or signal.
- Ready – Process ready to run.
- Terminated – Process has finished execution.

Process Control Block (PCB)

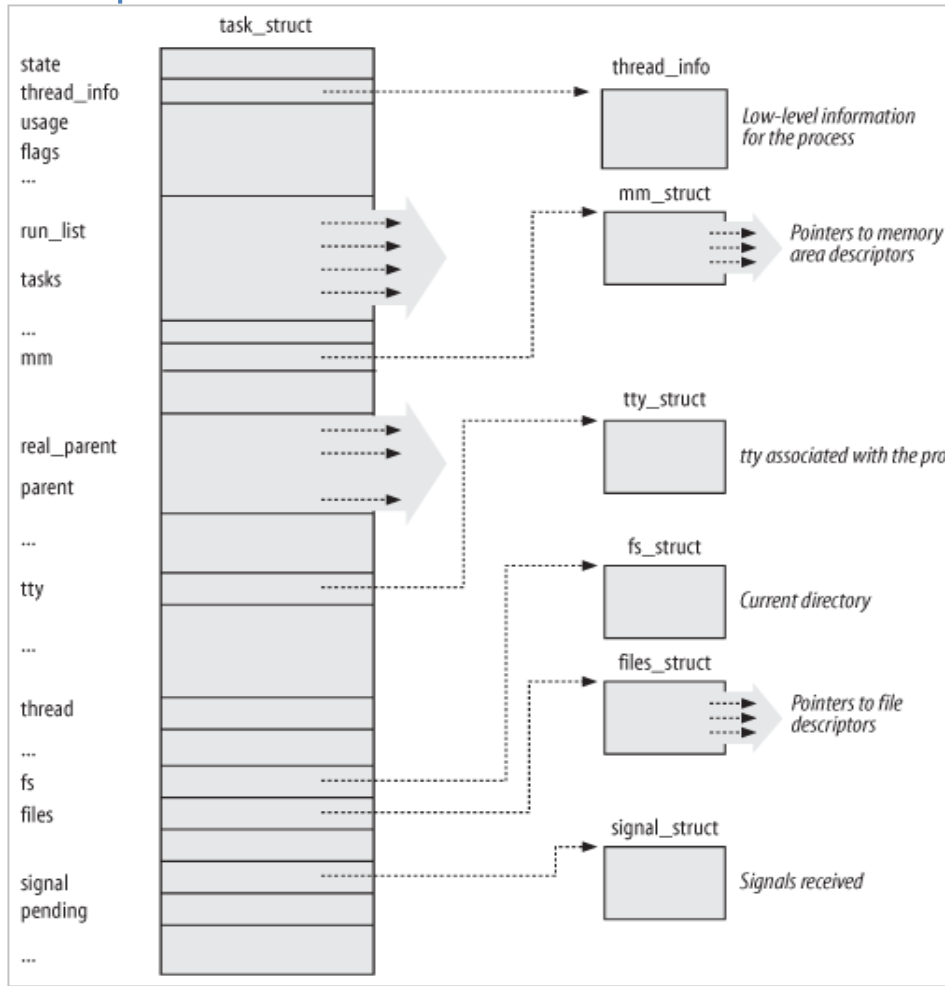
- Data struct **per** process.
- Maintained by OS.
- Not directly accessible.
- Usually a linked list.

What is stored in PCB?

- Process ID (identifier uniquely identifying the process)
- Process state (New, Running , Waiting etc)
- Context (or pointer to context)
 - Saved here during SW context saving.
 - PC, SP, GPRs etc.
- CPU scheduling info
 - Scheduling Policy / Priority
- Memory management Info (learn later)
- Accounting Info
- I/O Status
 - Files opened to the process, I/O devices used in the process etc

Struct task_struct

```
struct task_struct {
    volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
    struct thread_info *thread_info;
    atomic_t usage;
    unsigned long flags; /* per process flags, defined below */
    unsigned long ptrace;
    int lock_depth; /* Lock depth */
    int prio, static_prio;
    struct list_head run_list;
    prio_array_t *array;
    unsigned long sleep_avg;
    unsigned long long timestamp, last_ran;
    int activated;
    unsigned long policy;
    cpumask_t cpus_allowed;
    unsigned int time_slice, first_time_slice;
    ...
    unsigned long rt_priority;
    struct list_head tasks;
    ...
    struct mm_struct *mm, *active_mm;
    long exit_state;
    int exit_code, exit_signal;
    ...
    pid_t pid;
    pid_t tgid;
    ...
    uid_t uid, euid, suid, fsuid;
    gid_t gid, egid, sgid, fsgid;
    struct task_struct *real_parent; /* real parent process (w. debug) */
    struct task_struct *parent; /* parent process */
    struct list_head children; /* list of my children */
    struct list_head sibling; /* linkage in my parent's children list */
    struct task_struct *group_leader; /* threadgroup leader */
}
```



Process State Transitions

- New to Ready
 - PCB Initialized
 - Memory Allocated
- Ready to Running
 - Scheduler picked this process to run from the ready processes
 - You have another linked list of ready processes (ready queue)
- Running to Ready
 - Interrupt happens. (HW Interrupt Or Fault/Exception)
- Running to Waiting
 - During **some** system calls. (Especially I/O and process synchronization)
- Waiting to Ready
 - I/O event or Signal Process is waiting for has arrived.
- Running to Terminated
 - When program has finished, maintain info about this process for a while. (Why?)

Lecture Outline

- “running programs” (Why inverted commas?)
- What is in a Process?
 - A snapshot view
 - Context saving and restoring
- Process States and PCB
- Operations on process
 - Process Scheduling
 - Process Creation
 - Process Termination

Process Scheduling

- Selecting n process from ready queue to run in n CPUs.
- Which processes to choose?
 - Depends on scheduling algorithm and process attributes...

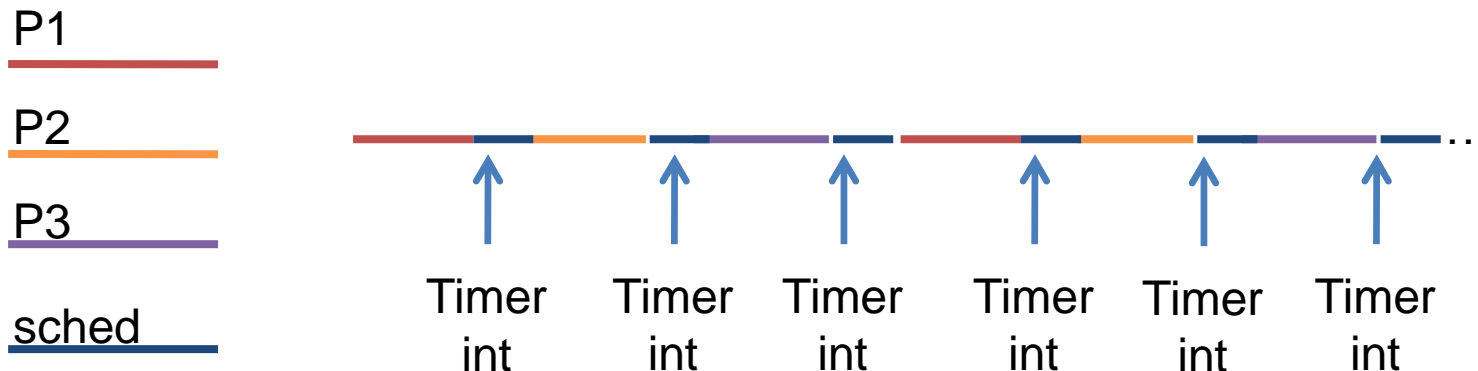
When is an opportunity for scheduling?

- A process goes from
 - running to waiting
 - running to ready
 - waiting to ready
 - running to terminated

Process scheduling

- Non-preemptive
 - Sequential execution
- Preemptive (time-shared)
 - Time slices/Timer Interrupt

Round-Robin scheduling



Process creation

- 2 things
 - Create a process
 - Loading and running a program
- Linux
 - 2 separate APIs
 - `fork()` – create process
 - `exec()` – run a new program
- Windows
 - 1 API
 - `CreateProcess` (both create process and run a new program)

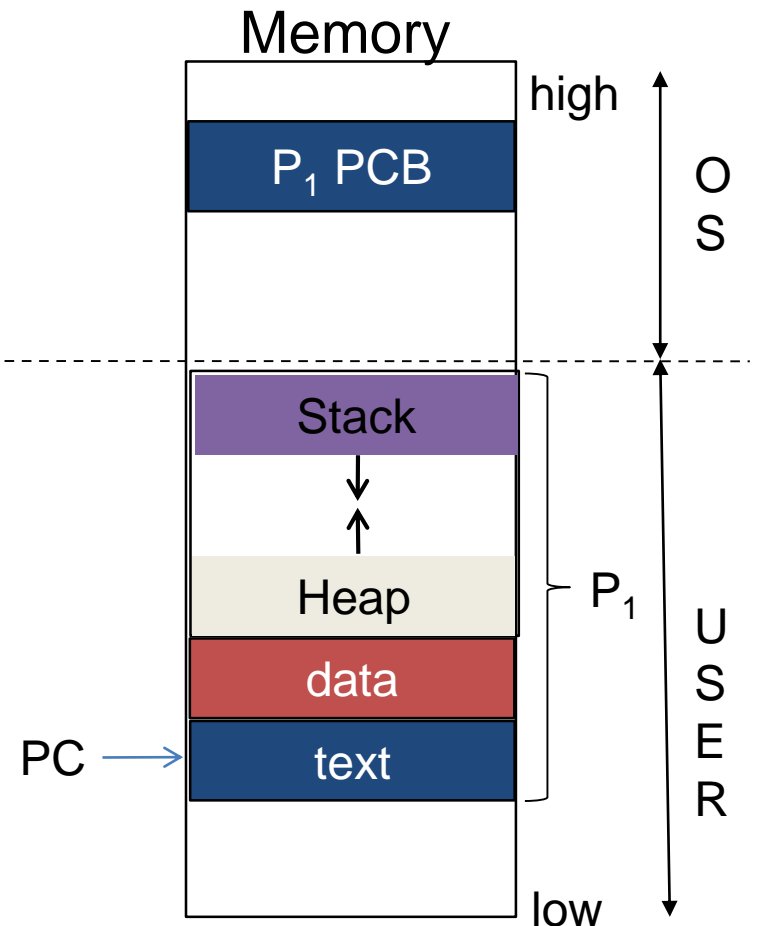
Linux Process Creation:

`fork()` - I

- parent **fork** child
- exact replica of parent (same memory contents, files, etc)
 - Different pid
 - Different return value for **fork()**
 - **Other diff see**
<https://linux.die.net/man/2/fork>
- **wait()** call and **exit()** call

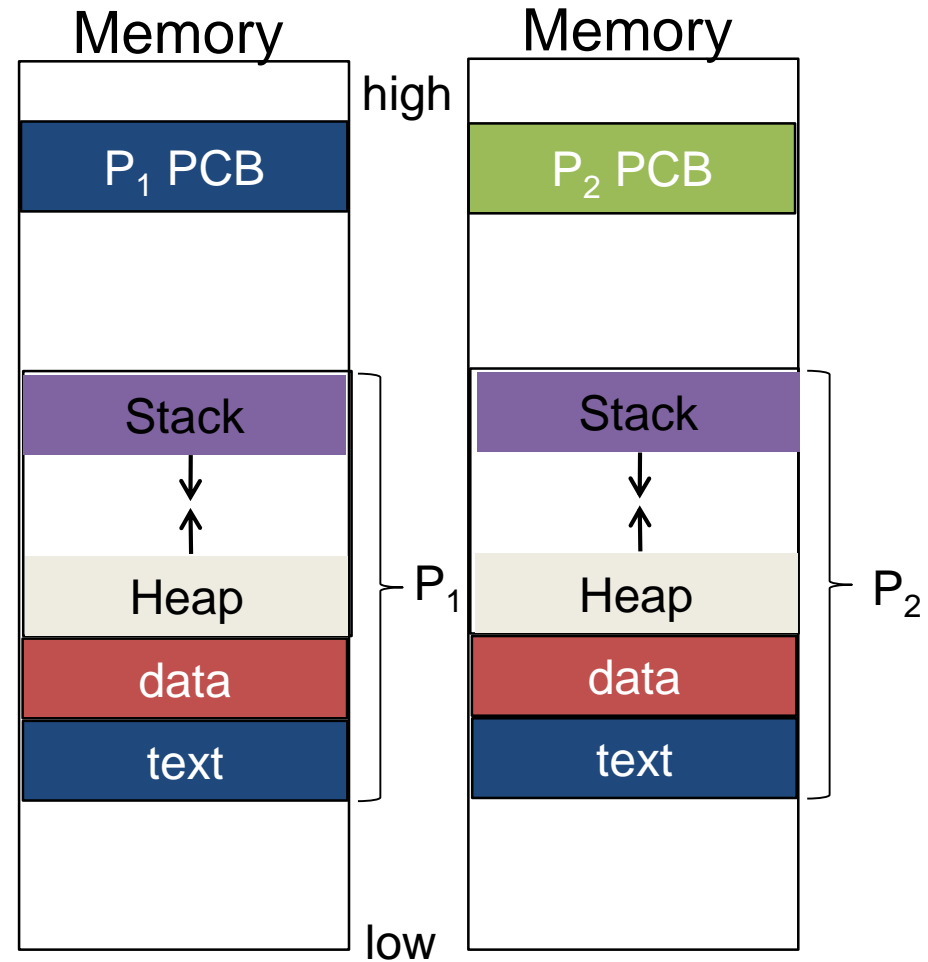
How fork works (1)

```
int main(void)
{
    ...
    ➔ int pid = fork();
    ...
}
```



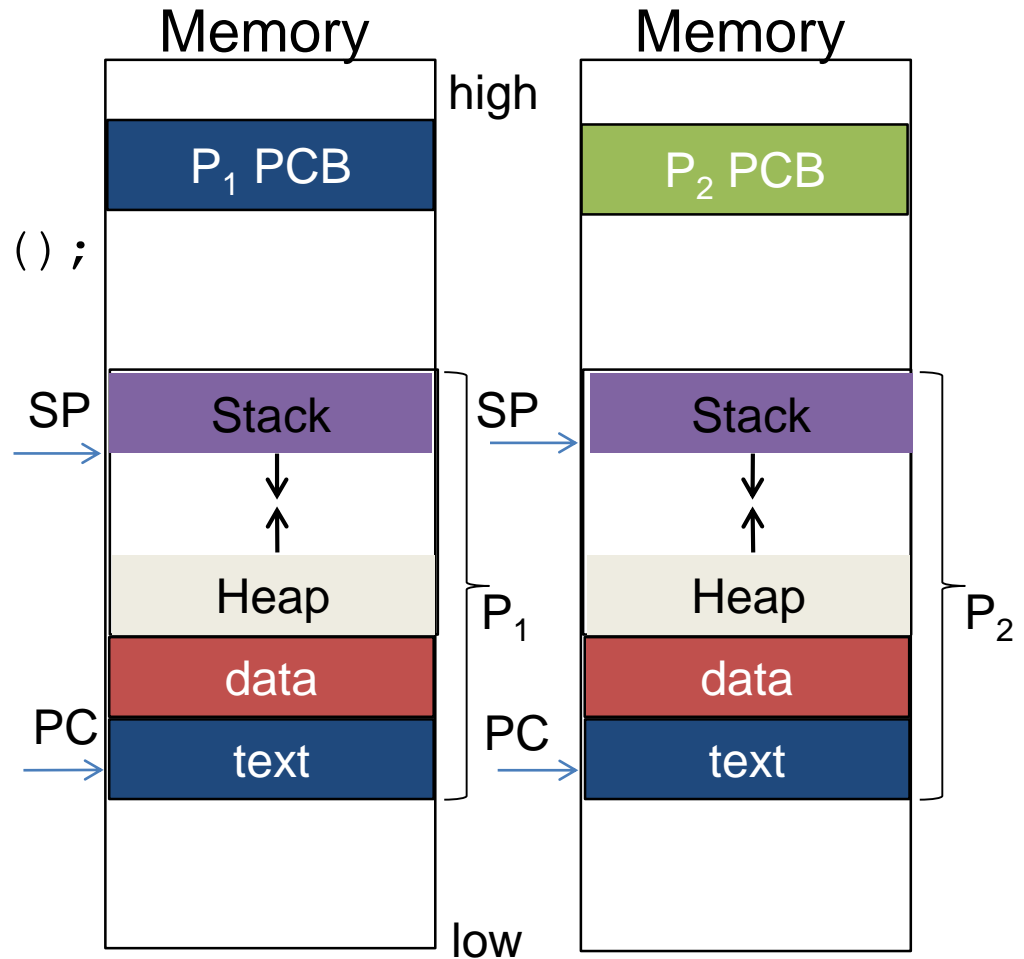
How fork works (2)

```
int main(void)
{
    ...
    int pid = fork();
    ...
}
```



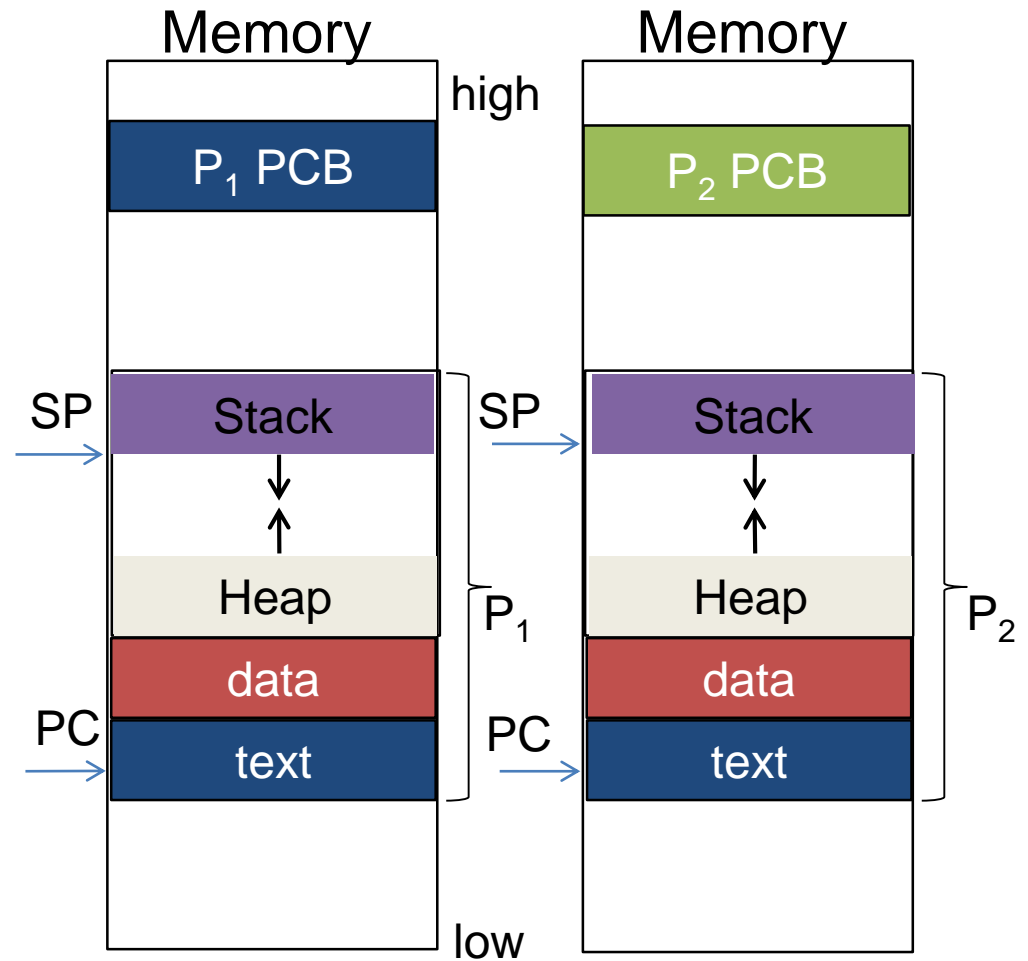
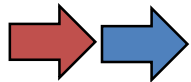
How fork works (3)

```
int main(void)
{
    ...
    int pid = fork();
    ...
}
```



How fork works (4)

```
int main(void)
{
    ...
    int pid = fork();
    ...
}
```



Linux Process Creation:

fork () - II

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>    // standard POSIX header file
#include <sys/wait.h>  // POSIX header file for 'wait' function

int main(void) {
    int i = -1;
    int pid;
    pid = getpid();
    fprintf(stdout, "parent pid = %d\n", pid);

    pid = fork();

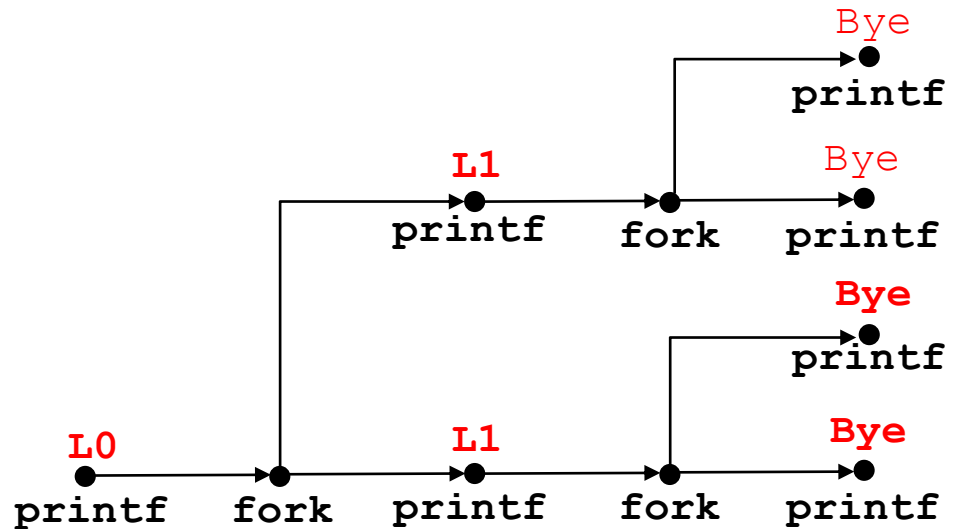
    if (pid == 0) {
        for (i=0; i < 10; ++i) {
            fprintf(stdout, "child process: %d\n", i);
            sleep(1);
        }
        exit(0);
    } else {
        fprintf(stdout, "child pid = %d\n", pid);
        fprintf(stdout, "waiting for child\n");
        wait(NULL);
        fprintf(stdout, "child terminated\n");
    }

    fprintf(stdout, "parent terminating\n");
    return 0;
}
```

fork Example: Two consecutive forks

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

forks.c



Feasible output:

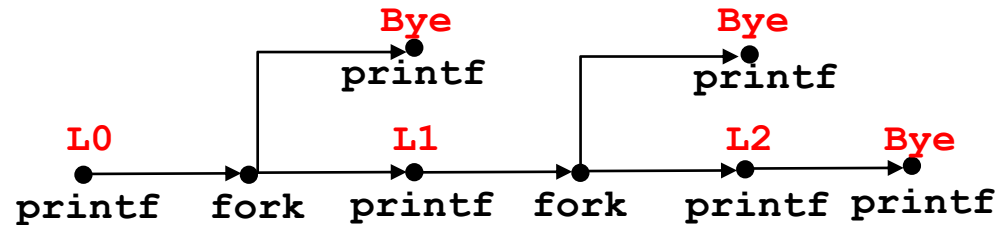
L0
L1
Bye
Bye
L1
Bye
Bye

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 output:

L0
L1
Bye
Bye
L2
Bye

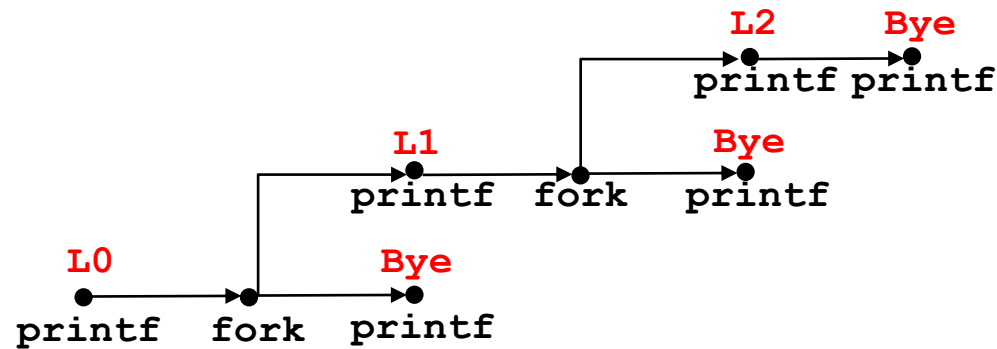
Infeasible output:

L0
Bye
L1
Bye
Bye
L2

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 output:

L0
Bye
L1
L2
Bye
Bye

Infeasible output:

L0
Bye
L1
Bye
Bye
L2

Homework – What does the following code print?

```
int main()
{
    ...
    fork();
    fork();
    printf("Hello World\n");
    ...
}
```

- How many processes are created by this?

Linux Process Creation: `exec()`

- `exec()` family of functions
- A function that executes a new program in the *same* process.
 - Or replace the image of a process with a new one.
- Basic arguments into `exec()`
 - File that contains the new program you want to execute.
 - Arguments for the new program.
 - Optional: Environment variables for the new program.

Different ways to pass the arguments

- Filename of new program
 - Either using PATH environment variable
 - Or Full Path Name
- Arguments for the new program
 - Either in a comma list. (NULL terminated)
 - Or a string array (NULL terminated)
- Environment Variables
 - Passed in a string array.

Linux Process Creation: `exec()`

- `fork` creates a process
- How to run a new program?
 - `exec()` family of functions
- Different front ends for `execve()`

```
int execve(const char *filename, char *const argv[],
           char *const envp[]);

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execlp(const char *path, const char *arg,
           ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```


Suffixes l, v, p and e

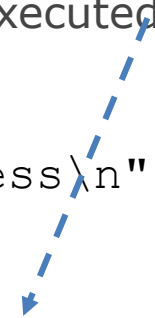
l	specifies that the argument pointers (arg0, arg1, ..., argn) are passed as separate arguments. Typically, the l suffix is used when you know in advance the number of arguments to be passed.
v	specifies that the argument pointers (argv[0] ..., arg[n]) are passed as an array of pointers. Typically, the v suffix is used when a variable number of arguments is to be passed.
p	specifies that the function searches for the file in those directories specified by the PATH environment variable (without the p suffix, the function searches only the current working directory). If the path parameter does not contain an explicit directory, the function searches first the current directory, then the directories set with the PATH environment variable.
e	specifies that the argument env can be passed to the child process, letting you alter the environment for the child process. Without the e suffix, child processes inherit the environment of the parent process.

Linux Process Creation: `exec()`

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
```

```
int main(void) {
    int pid;
    fprintf(stdout, "creating child process\n");
    pid = fork();
    if (pid == 0)
        execl("/usr/games/gnome-sudoku", "sudoku", NULL);
    else {
        fprintf(stdout, "waiting for child to terminate\n");
        wait(NULL);
        fprintf(stdout, "parent terminating\n");
    }
    return 0;
}
```

The first argument, by convention, should point to the filename associated with the file being executed

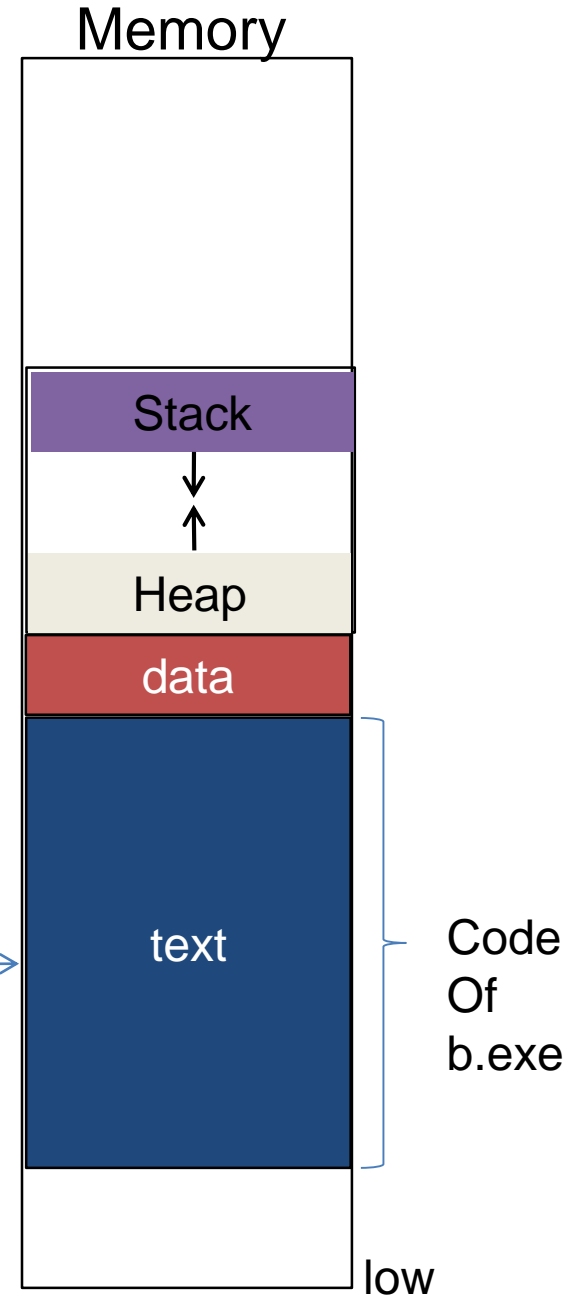


How exec works (1)

```
int main(void)
{
    ...
    → execl("/bin/ls", "ls", NULL);
    ...
}
```

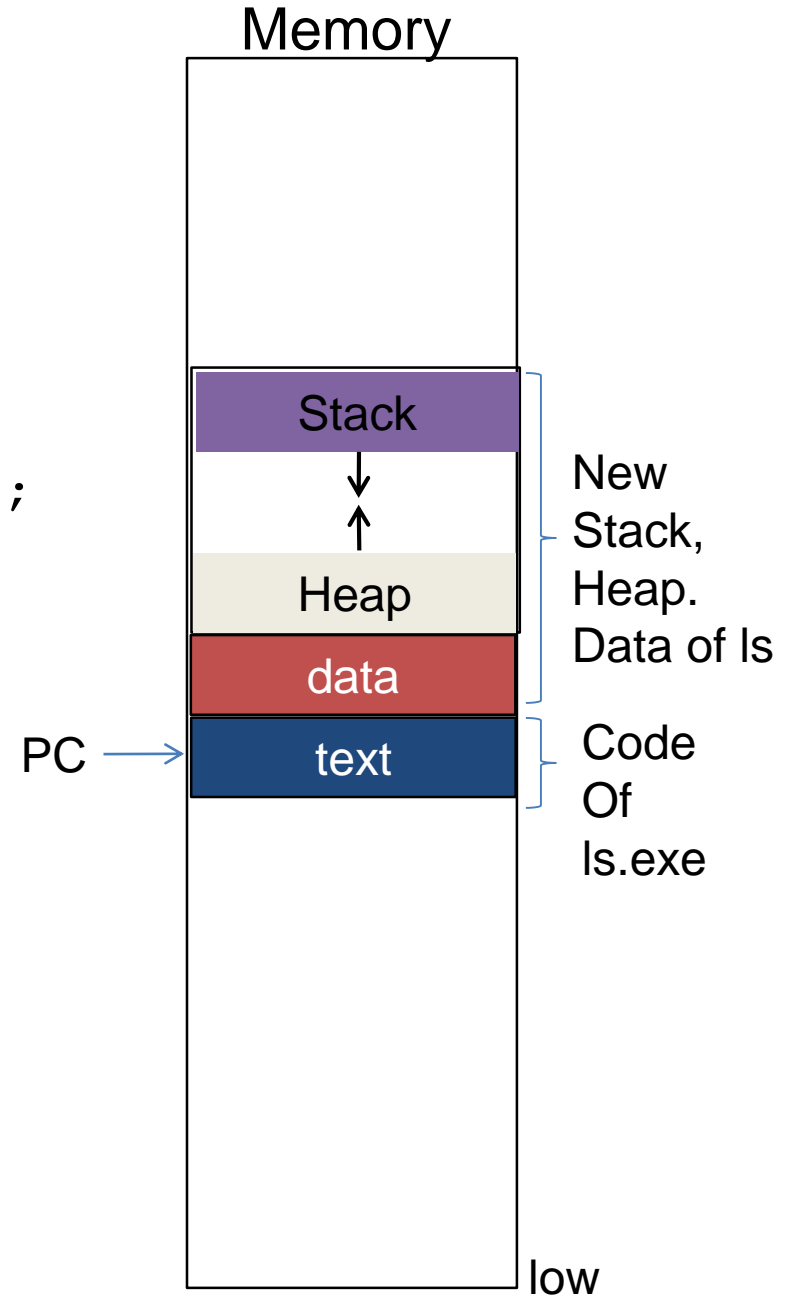
↓ compiles
b.exe

PC →
Pointing to
call to execl



How exec works (2)

```
int main(void)
{
    ...
    ➔ execl("/bin/ls", "ls", NULL);
    ...
}
```



Windows Process Creation

- `CreateProcess()`
- **Takes in 10 arguments!**
- `WaitForSingleObject()`
- `WaitForMultipleObjects()`

Example: CreateProcess

```
#include <iostream>
#include <windows.h>
using namespace std;
int main(void) {
    STARTUPINFO start_info;
    PROCESS_INFORMATION proc_info;
    ZeroMemory(&start_info, sizeof(STARTUPINFO));
    ZeroMemory(&proc_info, sizeof(PROCESS_INFORMATION));
    cout << "creating child process" << endl;
    CreateProcess("c:\\windows\\system32\\mspaint.exe", 0, 0, 0, FALSE,
        0, 0, 0, &start_info, &proc_info);
    cout << "waiting for child to terminate" << endl;
    WaitForSingleObject(proc_info.hProcess, INFINITE);
    cout << "parent terminating" << endl;
    CloseHandle(proc_info.hProcess);
    CloseHandle(proc_info.hThread);
    return 0;
}
```

Multiple child processes

```
#include <iostream>
#include <windows.h>
using namespace std;
int main(void) {
    const int COUNT = 2;
    HANDLE proc[COUNT], thread[COUNT];
    for (int i=0; i < COUNT; ++i) {
        STARTUPINFO start_info;
        PROCESS_INFORMATION proc_info;
        ZeroMemory(&start_info, sizeof(STARTUPINFO));
        ZeroMemory(&proc_info, sizeof(PROCESS_INFORMATION));

        CreateProcess("c:\\windows\\system32\\mspaint.exe", 0, 0, 0, FALSE, 0, 0, 0, &start_info, &proc_info);
        proc[i] = proc_info.hProcess;
        thread[i] = proc_info.hThread;
    }
    WaitForMultipleObjects(COUNT, proc, TRUE, INFINITE);
    for (int i=0; i < COUNT; ++i) {
        CloseHandle(proc[i]);
        CloseHandle(thread[i]);
    }
    return 0;
}
```

Zombie and Orphan Processes

- Zombie process: a process has completed execution (via the exit system call) in the "Terminated state" but still has an entry in the process table
- Orphan process: a process whose parent process has finished or terminated, though it remains runnable.