Processes

COMP 3361: Operating Systems I

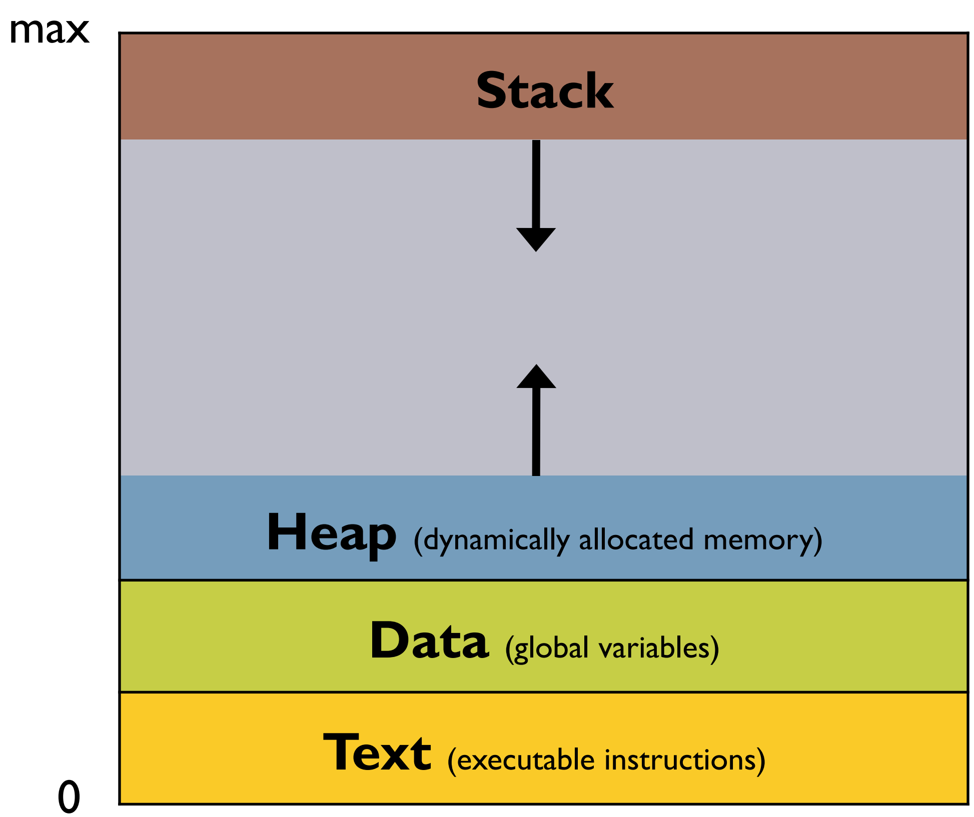
Winter 2015

http://www.cs.du.edu/3361

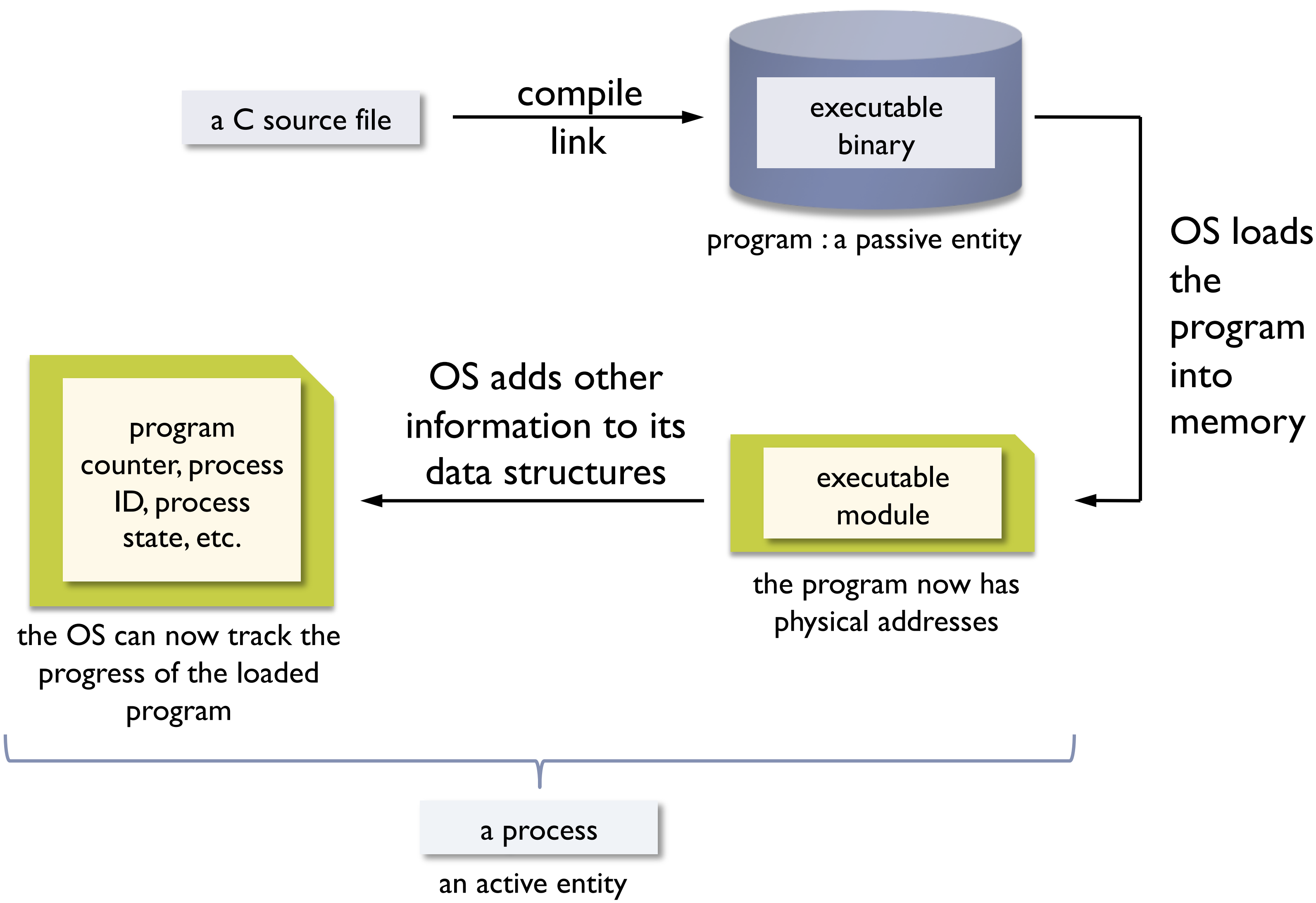
What is a Process?

- A **process** is a program in execution
- A program by itself is ***not*** a process
- A process also includes
  - a program counter
  - a stack
  - a data section
  - often a heap
  - a **process identifier (PID)**
  - …

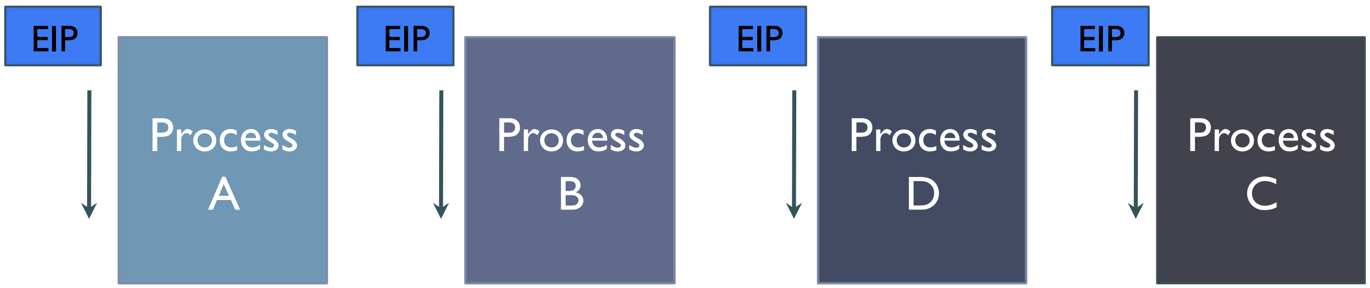
Process Address Space



Program to Process



4 Parallel Execution of Processes



Four independent processes, each with its own program counter

Multiprogramming

- But, a single-core single CPU has only one program counter (EIP register)

- Creating the illusion of parallel execution
  - each process has its logical program counter (stored in memory)
  - the value is loaded on the physical program counter before the process runs
  - when the CPU decides to run another process, the physical value is written to the logical program counter
  - overtime, all processes will make progress

- Only one process is running at any point in time!

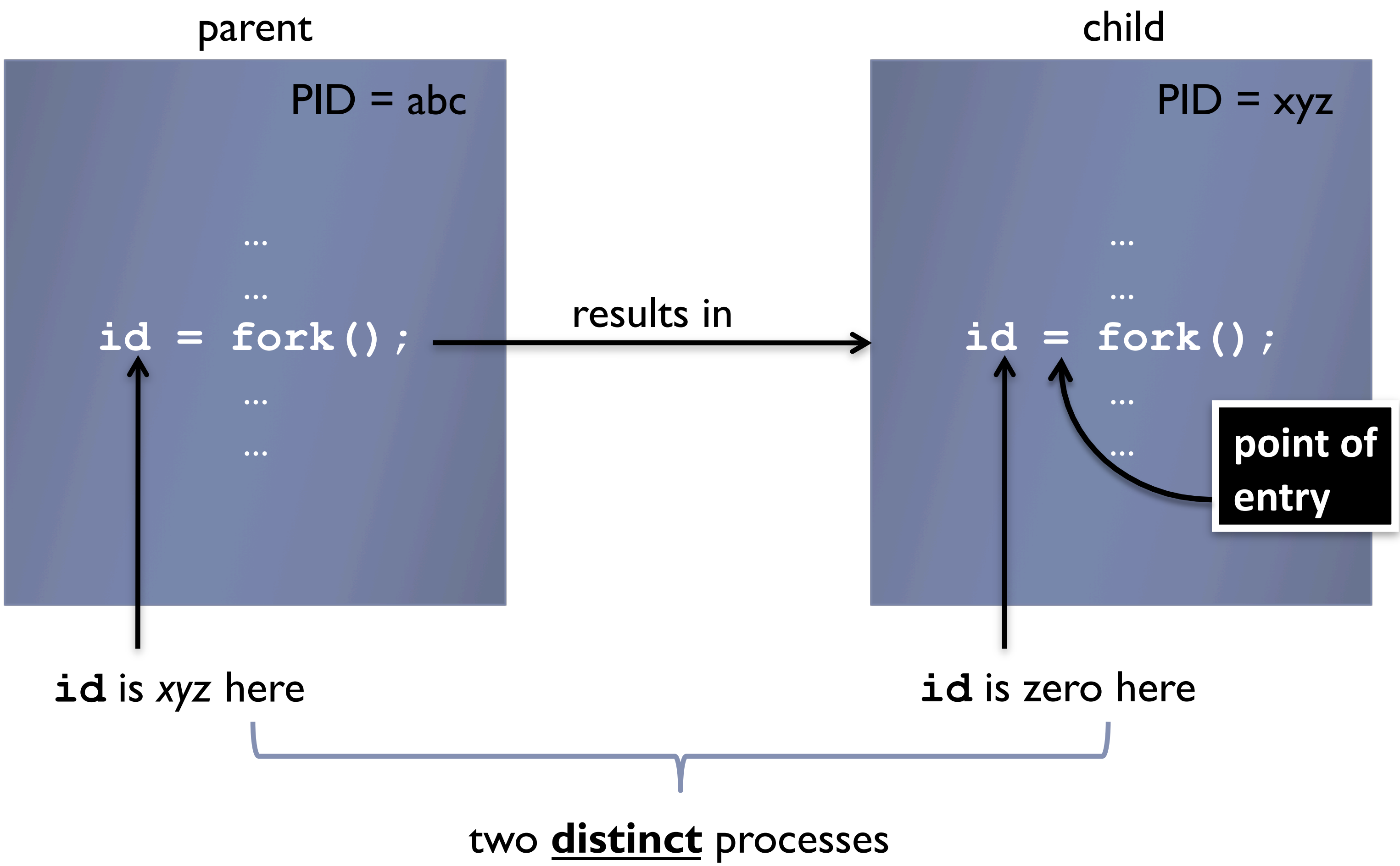
6 When are Processes Created

- During system initialization
  - usually to handle one or more system level task
- One process issues a system call to create another process
  - division of work
- User actions trigger the creation of a new process
  - command line or GUI based action to run a program
- Initiation of a batch job
  - execution of some queued task

UNIX Process Creation

- A process can create a new process using **`fork()`**
- Calling process becomes the **parent**, and the created process is the **child**
- What happens on a **`fork()`**?
  - child receives a copy of the parent's memory image
  - return value is
    - zero in the child process
    - the child's process identifier (PID) in the parent process
    - negative value indicates error
  - both processes ***independently*** resume execution at the instruction after the fork

fork()

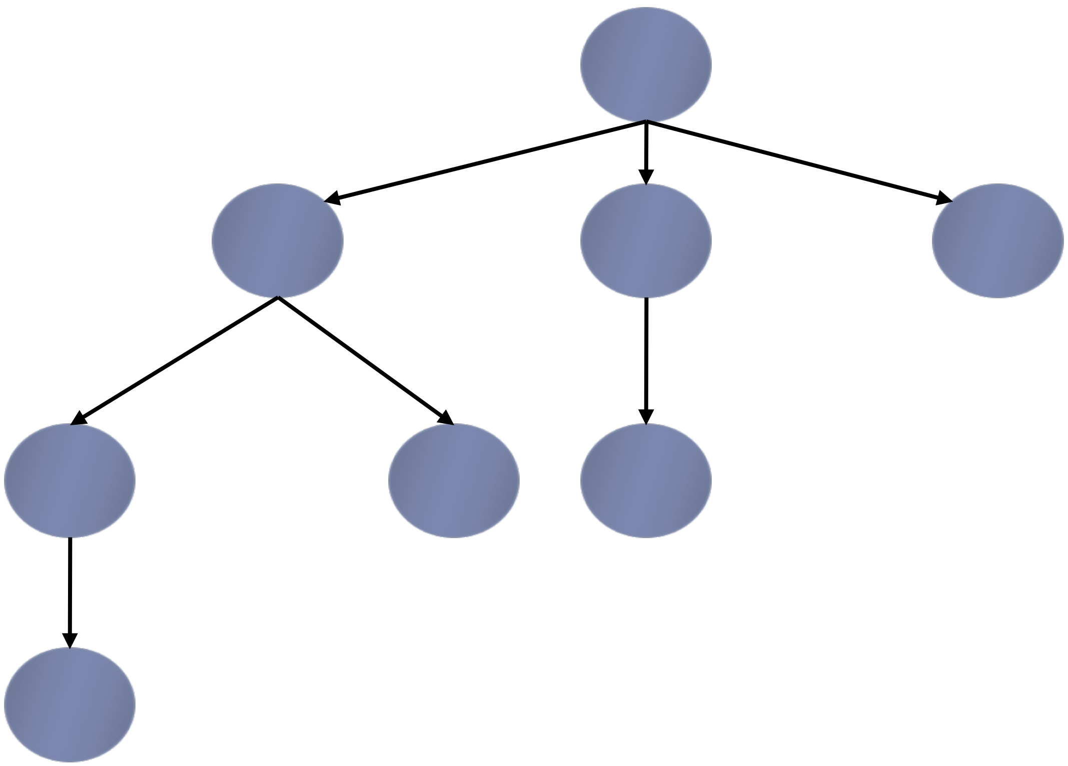

parent
PID = abc
…
…
id = fork();
…
…
results in
child
PID = xyz
…
…
id = fork();
…
…
point of entry
id is *xyz* here
id is zero here
two **distinct** processes

C fork Example


```
#include <unistd.h>

int main()
{
   pid_t  pid;
   /* fork another process */
   pid = fork();

   if (pid < 0) { /* error occurred */
        …
   }
   else if (pid == 0) { /* child process */
        …
   }
   else { /* parent process */
        …
   }
}
```

Process Tree

```
for (i=1; i<4; i++) {
   childpid = fork();
   if (childpid == -1) break;
}
```

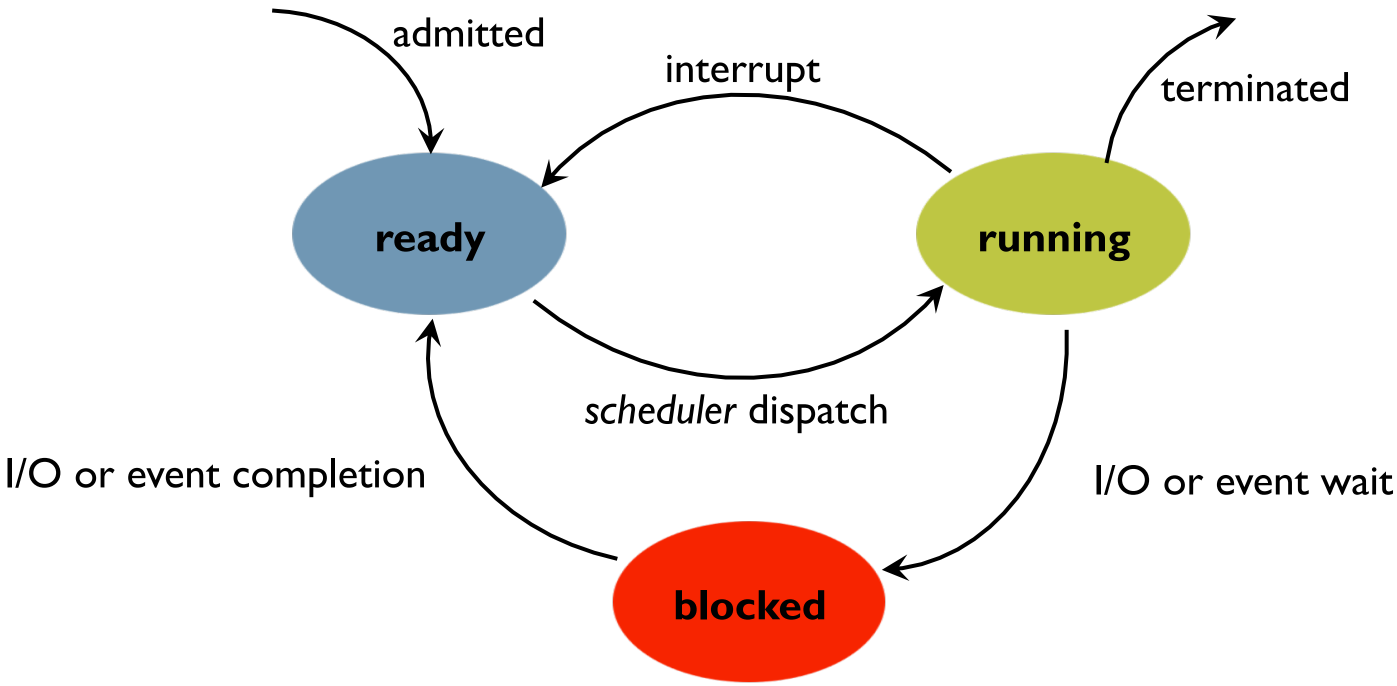


Process State

- A process changes state as it executes
  - **running**: instructions are being executed
  - **blocked**: the process is waiting for some event to occur
  - **ready**: the process is waiting to be assigned to a processor
- One running per CPU; many ready and waiting

12

Transitioning Between States



Process Control Block (PCB)

- The operating systems maintains all information related to a process in a data structure called the **process control block (PCB)**
- Information associated with each process includes
  - process ID and state
  - program counter
  - CPU registers
  - CPU-scheduling information
  - memory management information
  - accounting information
  - I/O status information
  - ...

A Very Simple PCB

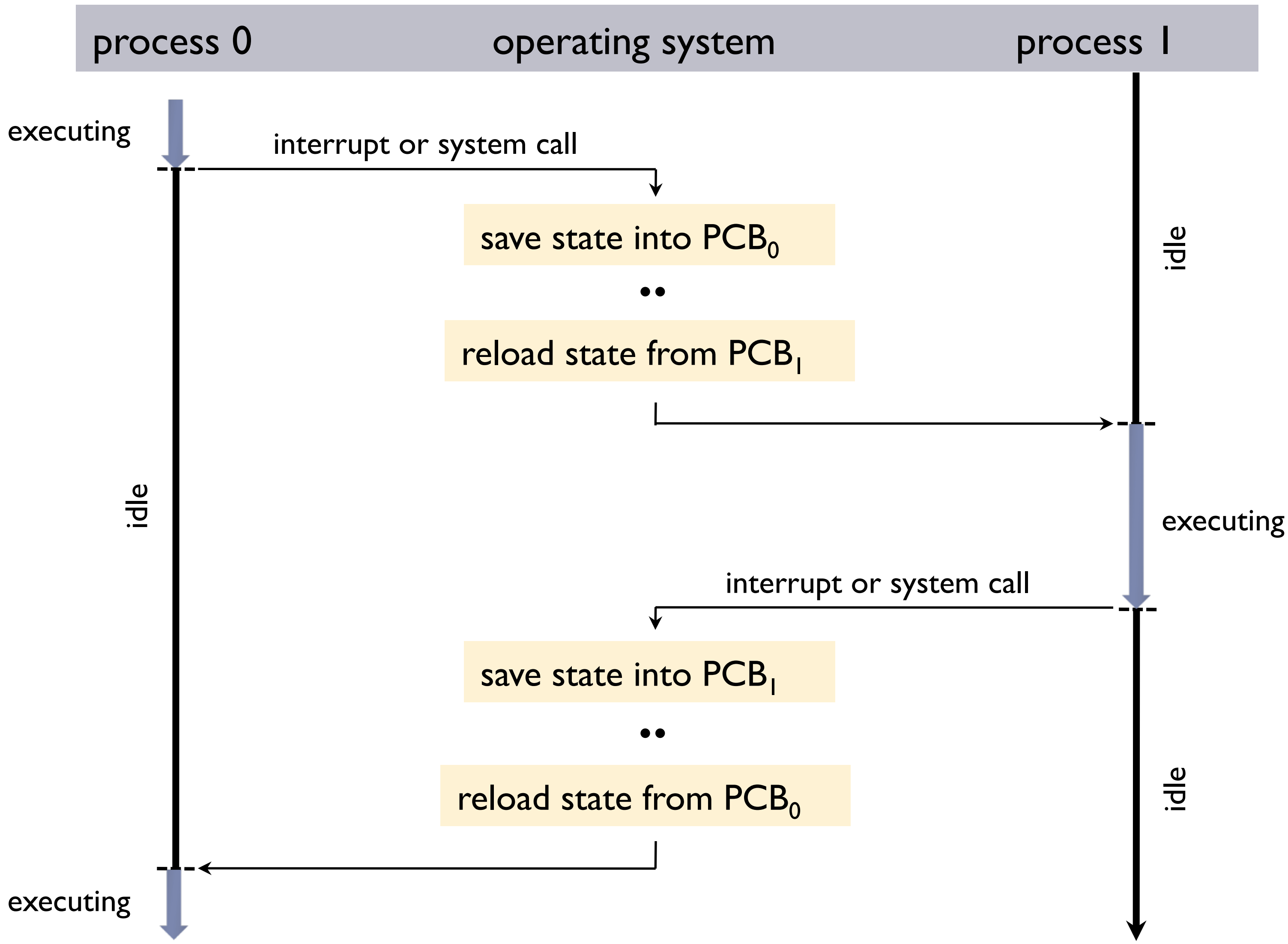
```
/*** Process Control Block (everything about a process) ***/
typedef struct process_control_block {
      struct {
            uint32_t ss;
            uint32_t cs;
            uint32_t esp;
            uint32_t ebp;
            uint32_t eip;
            uint32_t eflags;
            uint32_t eax;
            uint32_t ebx;
            uint32_t ecx;
            uint32_t edx;
            uint32_t esi;
            uint32_t edi;
      } cpu;

      uint32_t pid;
      enum {NEW, READY, RUNNING, WAITING, TERMINATED} state;
      uint32_t sleep_end;

      struct process_control_block *prev_PCB, *next_PCB;

      struct {
            uint32_t start_code;
            uint32_t end_code;
            uint32_t start_brk;
            uint32_t brk;
            uint32_t start_stack;
            PDE *page_directory;
      } mem;

      struct {
            uint32_t LBA;
            uint32_t n_sectors;
      } disk;
} __attribute__ ((packed)) PCB;
```

```
PCB process_table[100];
```

or

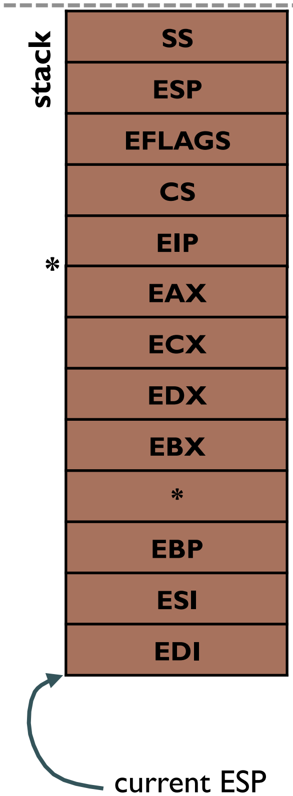
```
PCB *process_table_head;
```

15 Switch Between Processes



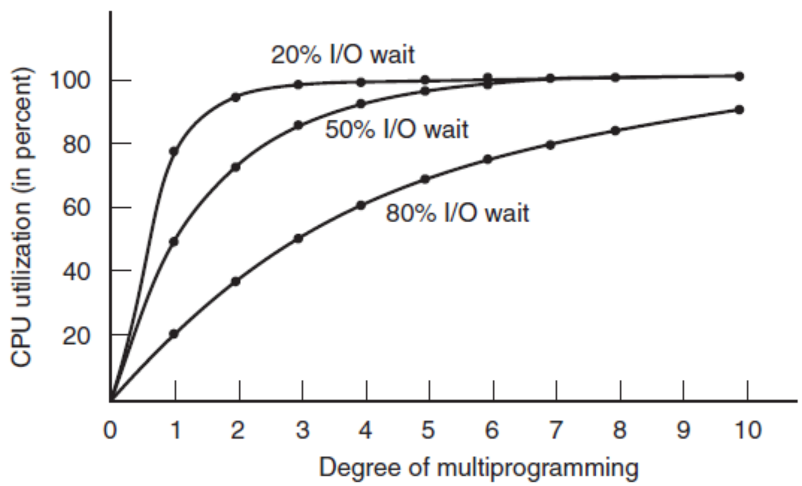
A System Call Handler

```
asm("handler_syscall_0X94_entry: \n" // no interruption until done
      // CPU would have already pushed these in order:
      // SS, ESP, EFLAGS, CS and EIP of calling process
      // Push EAX, EBX, ECX, EDX (system call arguments)
      "pushal\n"
      "movl %esp, %ecx\n"
      "call handler_syscall_0X94\n"
);
__attribute__((fastcall)) void handler_syscall_0X94(void) {
      // reload stack pointer (discards C function prologue)
      asm volatile ("movl %ecx, %esp\n");
-----------------------------------------------------------------------------
      // must reset the segment selectors before
      // accessing any kernel data
      asm volatile ("movl $0x10, %eax\n"
                    "movl %eax, %ds\n"
                    "movl %eax, %es\n"
                    "movl %eax, %fs\n"
                    "movl %eax, %gs\n");

      // save CPU state in process PCB
      asm volatile ("movl %%esp, %0\n": "=r"(current_process->cpu.edi));
      asm volatile ("movl 4(%%esp), %0\n": "=r"(current_process->cpu.esi));
      asm volatile ("movl 8(%%esp), %0\n": "=r"(current_process->cpu.ebp));
      asm volatile ("movl 16(%%esp), %0\n": "=r"(current_process->cpu.ebx));
      asm volatile ("movl 20(%%esp), %0\n": "=r"(current_process->cpu.edx));
      asm volatile ("movl 24(%%esp), %0\n": "=r"(current_process->cpu.ecx));
      asm volatile ("movl 28(%%esp), %0\n": "=r"(current_process->cpu.eax));
      asm volatile ("movl 32(%%esp), %0\n": "=r"(current_process->cpu.eip));
      asm volatile ("movl 36(%%esp), %0\n": "=r"(current_process->cpu.cs));
      asm volatile ("movl 40(%%esp), %0\n": "=r"(current_process->cpu.eflags));
      asm volatile ("movl 44(%%esp), %0\n": "=r"(current_process->cpu.esp));
      asm volatile ("movl 48(%%esp), %0\n": "=r"(current_process->cpu.ss));

      execute_0x94(); // handle system call

      schedule_something();   // call scheduler to pick a process
}
```



Multiprogramming Model

- n processes, each spending a fraction p of its time waiting for I/O
- Probability that all processes are waiting: p^n
- CPU utilization: $(1-p^n)$



Process Termination

- Process executes last statement and asks the operating system to delete it
  - via a system call automatically inserted by the compiler
  - process' resources are de-allocated by operating system
- A process may also be terminated
  - due to an error
  - another process issued a system call to terminate it
  - cascading termination

References

- Chapter 2.1, Modern Operating Systems, A. Tanenbaum and H. Bos, 4th Edition.