We now understand the basic process lifecycle & process control blocks. Now, let us look into more concrete details about these aspects & more!

Process creation

- *Traditionally, OS transparently created all processes*
- Modern OSs provide *system call* for process creation
- First process (e.g., *init* in Unix) created at boot time
  - This process creates other processes at start up & later as needed
- Other processes can also create new processes as needed

Process creation steps

- Assign unique *identifier* to new process
- Allocate & set up memory space for process (*process memory image*)
  - Process control block (*PCB*)
  - Program & data → organized into regions
    - Code/text space
    - Global data space
    - Heap (dynamically allocated data) space
    - Stack (local function data) space

- Set up memory management structures for process
  - *We will look at details of these structures later…*
- Other structures OS may keep for performance monitoring, etc.

Process creation mechanisms

- 1st option → *cloning*
  - Process spawns process that is a copy of itself
    - Adopted by Unix based Oss → *fork()* system call

- 2nd option → *creation from scratch*
  - Process creates new process with appropriate parameters
    - Adopted by Windows → *CreateProcess()* system call

Unix process creation (*fork*)

- Process invokes *fork* to initiate creation of *new process*
- Creating process is *parent* & new process is *child*
- System call creates new process
- Data from parent process *copied* to memory of child process
  - Memory image, environment settings, I/O handles, etc.
  - Of course, *child* gets its own ID, scheduling info, etc.

If cloning is the only way to create processes, does that mean all processes can only execute identical code?

Not really!

- *fork* call returns a value
  - Upon success, fork returns child ID to parent process…
  - …and returns 0 to child process
- Parent/child processes independently continue execution from statement after *fork*
- *[If process spawn fails, fork returns -1 to parent & no child is created]*

```
pid_t id = fork();
if(id == -1) {
    std::cout << "Error creating process\n";
} else if (id == 0) {
    std::cout << "I'm a child process!\n";
} else {
    std::cout << "I just became a parent!\n";
}
```

In other words…

- …return value to enable conditional execution after *fork*
  - So, even though program is same, paths can be different!
  - Different behavior can thus be achieved in parent & child

Alternatively

- Child can *change* the program it executes
  - Invoke *exec* family of system calls to change its memory image
  - Stops executing code in parent program & starts new program

```
pid_t id = fork();
 if(id == -1) {
  std::cout << “Error creating process\n”;
 } else if (id == 0) {
  // child process functionality
  char* args[] = {"echo", "hello", NULL};
  execvp(args[0], args);
 } else {
  std::cout << “I just became a parent!\n”;
 }
```

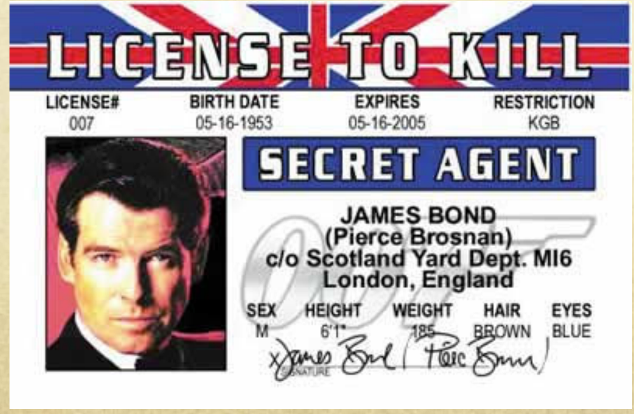
Must include <unistd.h>

Since cloning requires copying...

- ...it has high *overhead*
- Overhead somewhat reduced using *copy on write* concept
  - Start with both parent & child sharing same memory
  - Make copy if & when either process modifies data
    - So, if child changes memory image, copy need never be made!

Process termination

- As discussed earlier, process may terminate for many reasons
  - Voluntary exit upon task completion
  - Voluntary exit due to fatal error
  - Involuntary exit due to error/bug
  - Involuntary exit due to *kill* command by OS or other process
    - Of course, *killer* process must have appropriate *authorization*



On Unix based systems...

- OS maintains notion of process *hierarchy*
    - A process, its children, their children, etc. (i.e., all descendants) form process *group*
- Parent must be allowed to read child's *exit status*
- If *parent* terminates before *child*...
    - Child is now an *orphan* process
    - Orphan processes are *adopted* by *init* process

- If a *child* process terminates before *parent*
  - System will still need to keep child's *PCB*
  - Child process becomes a *zombie* process
    - "Dead", but not "reaped"
  - Parent process can reap children by waiting for them to terminate
    - OS provides system call for this → *wait*

While processes are alive…

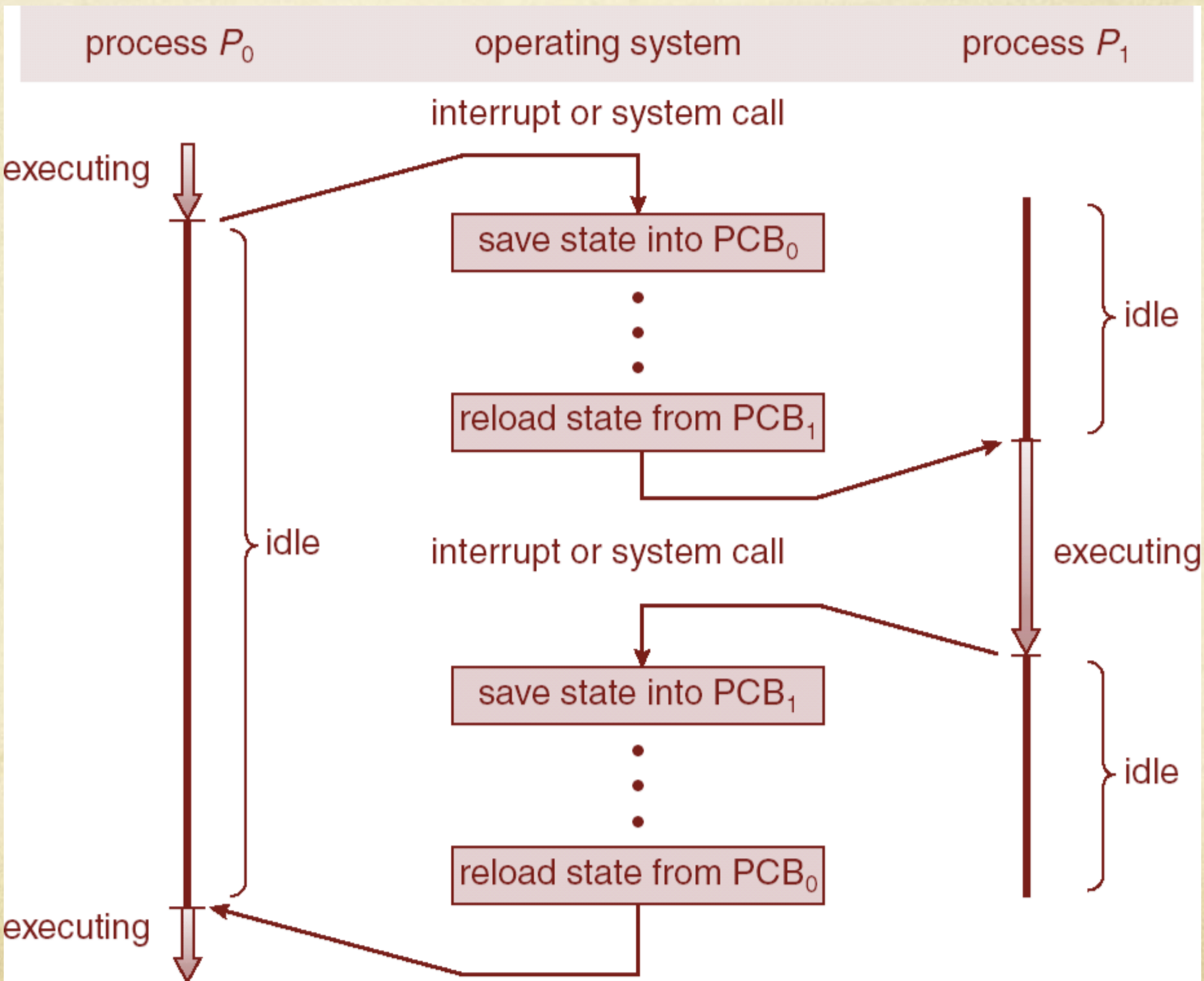
- …they must be appropriately managed by OS
- Multiple processes may be concurrent
  - Big part of management is *switching* between processes
- OS needs to have access to PCBs of processes
  - Maintains *process table* to hold PCBs
- Questions that arise
  - *When* does OS switch to different process?
  - *Which* process does OS switch to?

    Policies
  - *How* does OS perform process switch? → *Mechanism*

Process switching/context switching

- Example scenarios that may trigger process switch
  - Process termination (voluntary/involuntary)
  - New process activation
  - Executing process gets blocked (e.g., due to system call)
  - Event completion
  - Time slice expiration
- Process switching *mechanism* needs hardware & OS support
  - Requires transition to *kernel* mode → uses *interrupt*
  - Overhead of switch depends on hardware support (1-1000 μs)

Process switch execution flow



Typical steps

- [*hw*] Save program counters & some registers on stack
- [*hw*] Load program counter specified in *interrupt vector*
  - Interrupt vector → has address of *interrupt service routine*
- [*asm*] *Save* context info (registers, etc.) in *PCB* of curr process
- [*asm*] Set up new stack
- [C] Remaining work for specific interrupt type
- [*C-sched*] Choose new process to be scheduled
- [*asm*] *Restore* context info for new process & start it