# *Introduction to OS*
# Processes in Unix, Linux, and Windows
## MOS 2.1

Mahmoud El-Gayyar

elgayyar@ci.suez.edu.eg

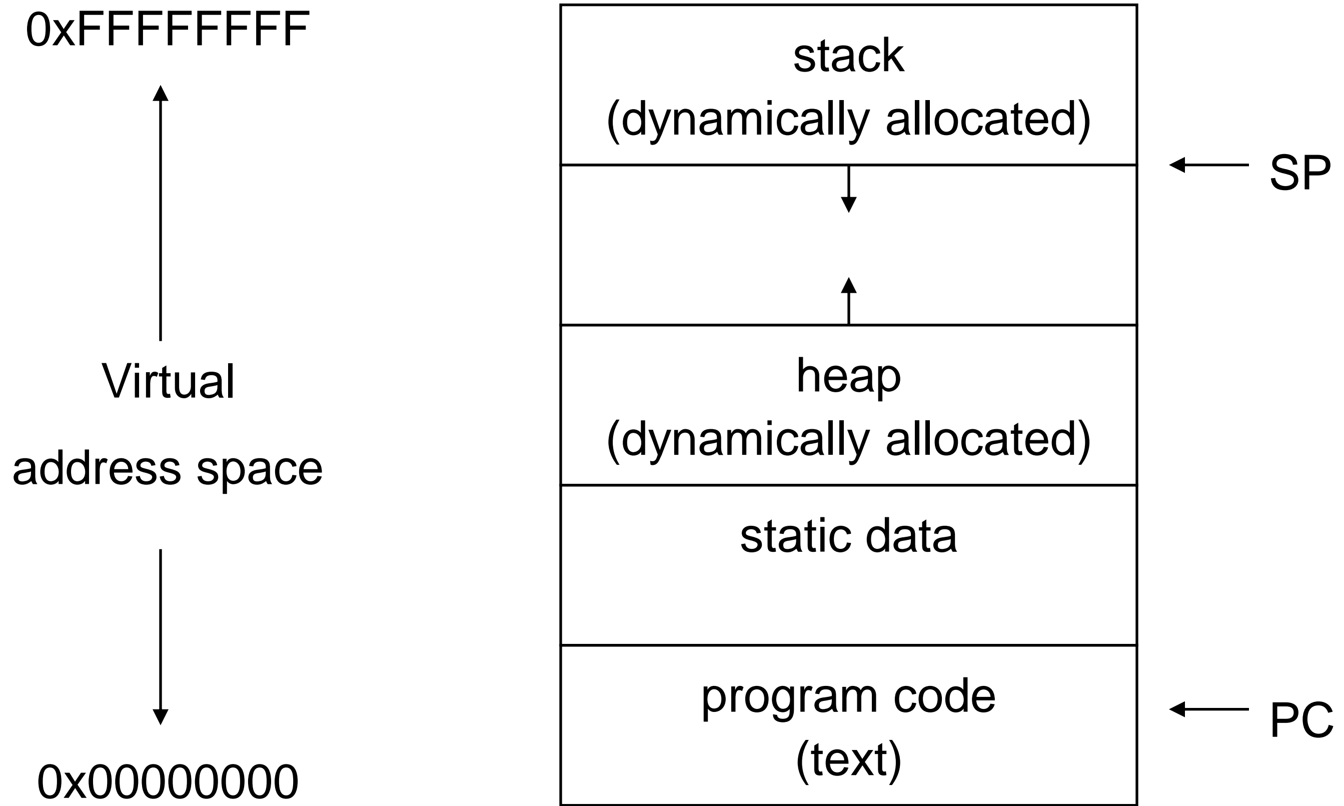
# Processes in Unix, Linux, and Windows

- Unix pre-empted generic term "*process*" to mean something very specific
- Linux and Windows adopted Unix definition

# Process in Unix-Linux-Windows includes

- an ***address space*** – usually protected and virtual – mapped into memory
- the ***code*** for the running program
- the ***data*** for the running program
- an ***execution stack*** and ***stack pointer*** (SP); also ***heap***
- the ***program counter*** (PC)
- a set of processor ***registers*** – general purpose and status
- a set of system ***resources***
  - files, network connections, pipes, …
  - privileges, (human) user association, …

# Process Address Space (traditional Unix)



# Processes in the OS – Representation

- To users (and other processes) a process is identified by its ***Process ID*** (PID)
- In the OS, processes are represented by entries in a ***Process Table*** (PT)
  - PID is index to (or pointer to) a PT entry
  - PT entry = *Process Control Block* (PCB)
- PCB is a large data structure that contains or points to all info about the process
  - Linux – defined in **`task_struct`** (over 70 fields)
    - see **`include/linux/sched.h`**
  - Windows XP – defined in *EPROCESS* – about 60 fields

# Processes in the OS – PCB

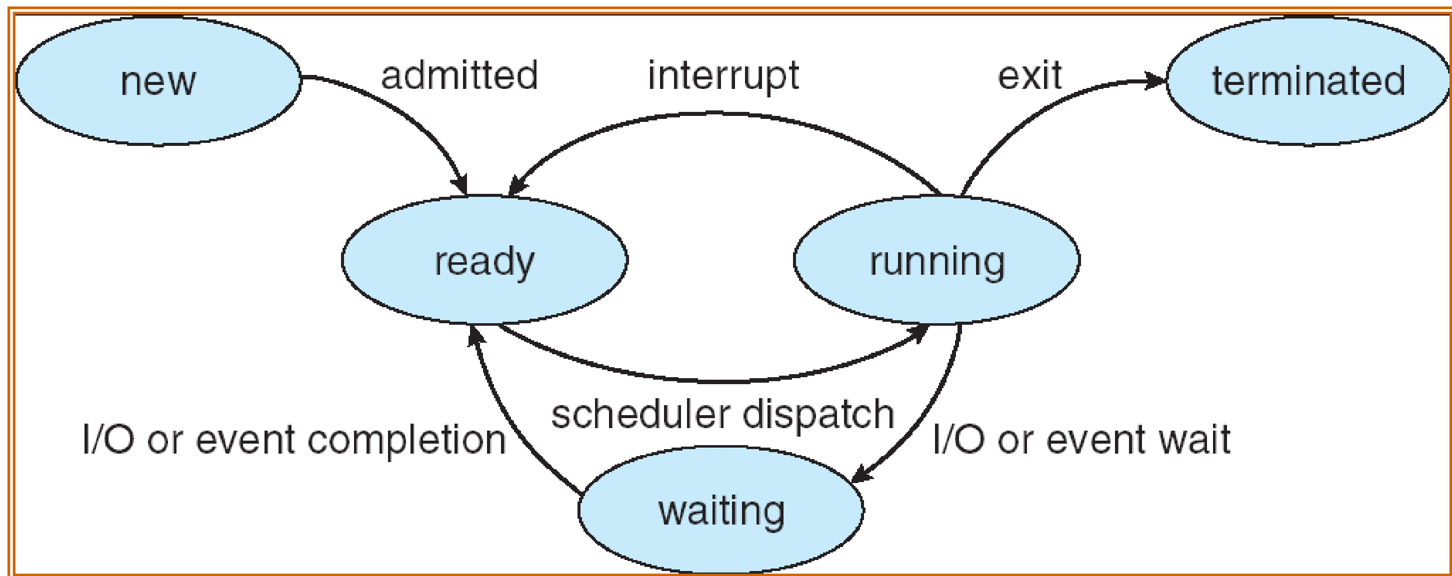
- Typical PCB contains:
  - execution state
  - PC, SP & processor registers – stored when process is not in *running* state
  - memory management info
  - privileges and owner info
  - scheduling priority
  - resource info
  - accounting info

# Process – Starting and Ending

- Processes are created …
  - When the system boots
  - By the actions of another process (more later)
  - By the actions of a user
  - By the actions of a batch manager
- Processes terminate …
  - Normally – exit
  - Voluntarily on an error
  - Involuntarily on an error
  - Terminated (killed) by action of
    - a user or
    - another process

# Processes – States

- Process has an execution **state**
  - *ready*: waiting to be assigned to CPU
  - *running*: executing on the CPU
  - *waiting*: waiting for an event, e.g. I/O



# Processes – State Queues

- The OS maintains a collection of *process state* queues
  - typically one queue for each state – e.g., ready, waiting, …
  - each PCB is put onto a queue according to its current state
  - as a process changes state, its PCB is unlinked from one queue, and linked to another
- Process state and the queues change in response to events – interrupts, traps

# Processes – Privileges

- Users are given privileges by the system administrator
- Privileges determine user *rights*
  - Unix/Linux – (9 bits) Read|Write|eXecute by user, group and "other" (i.e., "world")
  - WinNT – *A*ccess *C*ontrol *L*ist
- Processes "inherit" privileges from user
  - or from creating process

# Process Creation – Unix & Linux

- Create a new (child) process – **`fork();`**
  - Allocates new PCB
  - Clones the calling process (almost exactly)
    - Copy of parent process address space
    - Copies resources in kernel (e.g. files)
  - Places new PCB on *Ready queue*
  - Return from **`fork()`** call
    - 0 for child
    - child PID for parent

# Example of *fork( )*

```
int main(int argc, char **argv)
{
  char *name = argv[0];
  int child_pid = fork();
  if (child_pid == 0) {
    printf("Child of %s sees PID of %d\n",
           name, child_pid);
    return 0;

  } else {
    printf("I am the parent %s. My child is %d\n",
                 name, child_pid);
    return 0;
  }
}
```

---

**% ./forktest**
**Child of forktest sees PID of 0**
**I am the parent forktest. My child is 486**

# Starting New Programs

- Unix & Linux:–
  - `int exec (char *prog, char **argv)`
  - Check privileges and file type
  - Loads program at path *prog* into address space
    - Replacing previous contents!
    - Execution starts at `main()`
  - Initializes context – e.g. passes arguments
    - `*argv`
  - Place PCB on *ready queue*
  - Preserves, pipes, open files, privileges, etc.

# Executing a New Program (Linux-Unix)

- `fork()` followed by `exec()`
- Creates a new process as clone of previous one
  - I.e., same program, but different execution of it
- First thing that clone does is to replace itself with new program

# *Fork* + *Exec* – shell-like

```
int main(int argc, char **argv)
{ char *argvNew[5];
  int pid;
  if ((pid = fork()) < 0) {
        printf( "Fork error\n“);
        exit(1);
  } else if (pid == 0) { /* child process */
        argvNew[0] = "/bin/ls";  /* i.e., the new program */
        argvNew[1] = "-l";
        argvNew[2] = NULL;
        if (execve(argvNew[0], argvNew, environ) < 0) {
           printf( "Execve error\n“);
           exit(1);  /* program should not reach this point */
        }
   } else { /* parent */
        wait(pid); /* wait for the child to finish */
   }
}
```

# Processes – Windows

- Windows NT/XP – combines **`fork`** & **`exec`**
  - **`CreateProcess(10 arguments)`**
  - Not a parent child relationship
  - *Note* – privileges required to create a new process

# Traditional Unix

- *Processes* are in *separate* address spaces
  - By default, no shared memory
- *Processes* are unit of scheduling
  - A process is *ready*, *waiting*, or *running*
- *Processes* are unit of resource allocation
  - Files, I/O, memory, privileges, …
- *Processes* are used for (almost) everything!

# Review

- What is the difference/s between processes in Linux and Windows?