# The Big Picture So Far

**From the Architecture to the OS to the User**: Architectural resources, OS management, and User Abstractions.

| Hardware Abstraction | Example OS Services | User Abstraction |
|---|---|---|
| Processor | Process management, Scheduling, Traps Protection, Billing, Synchronization | Process |
| Memory | Management, Protection, Virtual memory | Address space |
| I/O devices | Concurrency with CPU, Interrupt handling | Terminal, Mouse, Printer, System Calls |
| File system | Management, Persistence | Files |
| Distributed systems | Network security, Distributed file system | RPC system calls, file sharing |





# Today: Process Management

- A process as the unit of execution.
- How are processes represented in the OS?
- What are possible execution states and how does the system move from one state to another?
- How do processes communicate? Is this efficient?

# What's in a Process?

- **Process:** dynamic execution context of an executing program
- Several processes may run the same program, but each is a distinct process with its own state (e.g., MS Word).
- A process executes sequentially, one instruction at a time
- Process state consists of at least:
  - the code for the running program,
  - the static data for the running program,
  - space for dynamic data (the heap), the heap pointer (HP),
  - the Program Counter (PC), indicating the next instruction,
  - an execution stack with the program's call chain (the stack), the stack pointer (SP)
  - values of CPU registers
  - a set of OS resources in use (e.g., open files)
  - process execution state (ready, running, etc.).





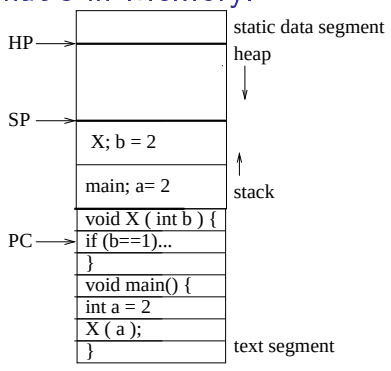
# Example Process State in Memory

What you wrote:

```
        void X ( int b ) {
PC→         if (b == 1) ...
        }

        main() {
            int a = 2;
            X ( a );
        }
```

What's in Memory:



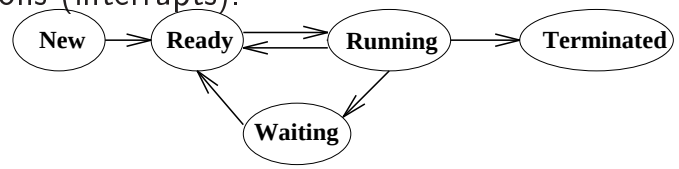

**Process State**

## Process Execution State

- Execution state of a process indicates what it is doing

| | |
|---|---|
| *new*: | the OS is setting up the process state |
| *running*: | executing instructions on the CPU |
| *ready*: | ready to run, but waiting for the CPU |
| *waiting*: | waiting for an event to complete (e.g., I/O) |
| *terminated*: | the OS is destroying this process |

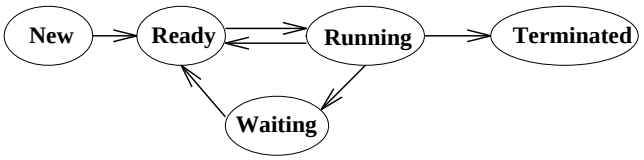
- As the program executes, it moves from state to state, as a result of the program actions (e.g., system calls), OS actions (scheduling), and external actions (interrupts).








## Process Execution State




**Example:**

```
  void main() {
    printf('Hello World\n');
}
```

**state sequence**
new
ready
running
waiting for I/O
ready
running
terminated

- The OS manages multiple active process using *state queues*
(More on this in a minute . . .)

## Process Data Structures

- Process Control Block (PCB): OS data structure to keep track of all processes
  - The PCB tracks the execution state and location of each process
  - The OS allocates a new PCB on the creation of each process and places it on a state queue
  - The OS deallocates the PCB when the process terminates
- The PCB contains:

- Process state (running, waiting, etc.)
- Process number
- Program Counter
- Stack Pointer
- General Purpose Registers
- Memory Management Information
- Username of owner
- List of open files
- Queue pointers for state queues
- Scheduling information (e.g., priority)
- I/O status
- ...

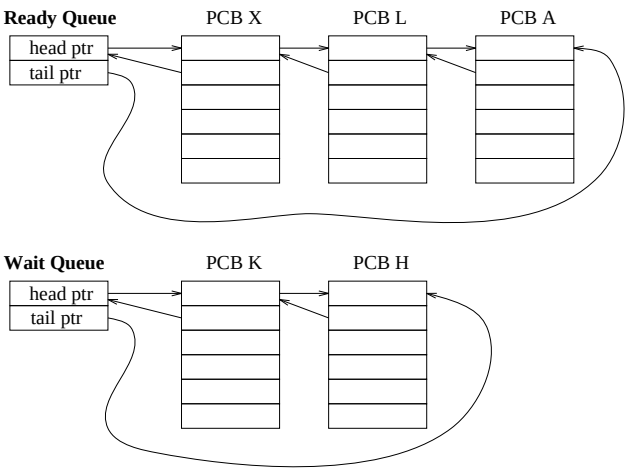




## Process State Queues

- The OS maintains the PCBs of all the processes in *state queues*.
- The OS places the PCBs of all the processes in the same *execution state* in the same queue.
- When the OS changes the state of a process, the PCB is unlinked from its current queue and moved to its new state queue.
- The OS can use different policies to manage each queue.
- Each I/O device has its own wait queue.

## State Queues: Example








## PCBs and Hardware State

- Starting and stopping processes is called a **context switch**, and is a relatively expensive operation.
- The OS starts executing a ready process by loading hardware registers (PC, SP, etc) from its PCB
- While a process is running, the CPU modifies the Program Counter (PC), Stack Pointer (SP), registers, etc.
- When the OS stops a process, it saves the current values of the registers, (PC, SP, etc.) into its PCB
- This process of switching the CPU from one process to another (stopping one and starting the next) is the context switch.
  - Time sharing systems may do 100 to 1000 context switches a second.
  - The cost of a context switch and the time between switches are closely related

## Creating a Process

- One process can create other processes to do work.
  - The creator is called the *parent* and the new process is the *child*
  - The parent defines (or donates) resources and privileges to its children
  - A parent can either wait for the child to complete, or continue in parallel
- In Unix, the *fork* system call called is used to create child processes
  - Fork copies variables and registers from the parent to the child
  - The *only difference* between the child and the parent is the value returned by fork
    - In the parent process, fork returns the process id of the child
    - In the child process, the return value is 0
  - The parent can wait for the child to terminate by executing the *wait* system call or continue execution
  - The child often starts a new and different program within itself, via a call to *exec* system call.





## Creating a Process: Example

- When you log in to a machine running Unix, you create a shell process.
- Every command you type into the shell is a child of your shell process and is an implicit *fork* and *exec* pair.
- For example, you type emacs, the OS *"forks"* a new process and then *"exec"* (executes) emacs.
- If you type an & after the command, Unix will run the process in parallel with your shell, otherwise, your next shell command must wait until the first one completes.

## Example Unix Program: Fork

```
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
main() {
  int parentID = getpid();    /* ID of this process */
  char  prgname[1024];
  gets(prgname); /* read the name of program we want to start */
  int cid = fork();
  if(cid == 0) {  /* I'm the child process */
    execlp( prgname, prgname, 0);  /* Load the program */
    /* If the program named prgname can be started, we never get
   to this line,  because the child program is replaced by prgname */
    printf("I didn't find program %s\n", prgname);
  } else { /* I'm the parent process */
    sleep (1);  /*  Give my child time to start. */
    waitpid(cid, 0, 0);  /*  Wait for my child to terminate. */
    printf("Program %s finished\n", prgname);
} }
```





## Example Unix Program: Explanation

**fork()** forks a new child process that is a copy of the parent.

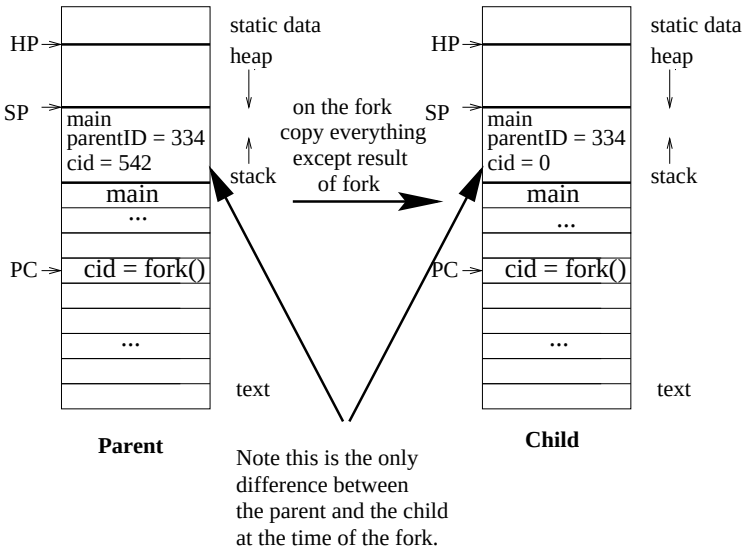
**execlp()** replaces the program of the current process with the named program.

**sleep()** suspends execution for at least the specified time.

**waitpid()** waits for the named process to finish execution.

**gets()** reads a line from a file.

# What is happening on the Fork








# Process Termination

- On process termination, the OS reclaims all resources assigned to the process.
- In Unix
  - a process can terminate itself using the *exit* system call.
  - a process can terminate a child using the *kill* system call.

## Example Unix Program: Process Termination

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
main() {
  int parentID = getpid();    /* ID of this process */
  int cid = fork();
  if(cid == 0) {  /*  I'm the child process  */
    sleep (5);     /* I'll exit myself after 5 seconds. */
    printf ( "Quitting child\n" );
    exit (0);
    printf ( "Error!  After exit call.!"); /* should never get here */
  } else {  /*  I'm the parent process  */
    printf ( "Type any character to kill the child.\n" );
    char answer[10];
    gets (answer);
    if ( !kill(cid, SIGKILL) ) {
      printf("Killed the child.\n");
} } }
```





## Cooperating Processes

- Any two process are either independent or cooperating
- Cooperating processes work with each other to accomplish a single task.
- Cooperating processes can
  - improve performance by overlapping activities or performing work in parallel,
  - enable an application to achieve a better program structure as a set of cooperating processes, where each is smaller than a single monolithic program, and
  - easily share information between tasks.

$\Rightarrow$ Distributed and parallel processing is the wave of the future. To program these machines, we must cooperate and coordinate between separate processes.

## Cooperating Processes: Producers and Consumers

```
n = 100;    // max outstanding items
in = 0;
out = 0;
producer
        repeat  forever {
            . . .
            nextp = produced item
            . . .
            // Make sure buffer not full
            while in+1 mod n = out do
                no-op
            buffer[in] = nextp
            in = in+1 mod n
        }
```

```
consumer
        repeat  forever {
            // Make sure buffer not empty
            while in = out do no-op
            nextc = buffer[out]
            out = out+1 mod n
            . . .
            consume nextc
            . . .
        }
```

- Producers and consumers can communicate using *message passing* or *shared memory*





## Communication using Message Passing

```
main()
    . . .
    if (fork() != 0) producerSR();
    else consumerSR();
end
```

```
producerSR
        repeat
            . . .
            produce item nextp
            . . .
            send (nextp, consumer)
```

```
consumerSR
        repeat
            receive (nextc, producer)
            . . .
            consume item nextc
            . . .
```

## Message Passing

- Distributed systems typically communicate using message passing
- Each process needs to be able to name the other process.
- The consumer is assumed to have an infinite buffer size.
- A bounded buffer would require the tests in the previous slide, and communication of the `in` and `out` variables (`in` from producer to consumer, `out` from consumer to producer).
- OS keeps track of messages (copies them, notifies receiving process, etc.).

$\Rightarrow$ How would you use message passing to implement a single producer and multiple consumers?





## Communication using Shared Memory

- Establish a mapping between the process's address space to a named memory object that may be shared across processes
- The mmap(...) systems call performs this function.
- Fork processes that need to share the data structure.

## Shared Memory Example

```
main()
    ...
    mmap(..., in, out, PROT_WRITE, MAP_SHARED, ...);
    in = 0;
    out = 0;
    if (fork() != 0) producer();
    else consumer();
end
```

```
producer
repeat
        ...
        produce item nextp
        ...
        while in+1 mod n = out do no-op
        buffer[in] = nextp
        in = in+1 mod n
```

```
consumer
        repeat
        while in = out do no-op
        nextc = buffer[out]
        out = out+1 mod n
        ...
        consume item nextc
        ...
```





## Process Management: Summary

- A process is the unit of execution.
- Processes are represented as Process Control Blocks in the OS
  - PCBs contain process state, scheduling and memory management information, etc
- A process is either New, Ready, Waiting, Running, or Terminated.
- On a uniprocessor, there is at most one running process at a time.
- The program currently executing on the CPU is changed by performing a context switch
- Processes communicate either with message passing or shared memory