Architectural features useful for OS services

OS Service	Hardware Support
Protection	Kernel/User mode
	Protected Instructions
	Base and Limit Registers
Interrupts	Interrupt Vectors
System calls	Trap instructions and trap vectors
I/O	Interrupts or Memory-Mapping
Scheduling, error recovery, billing	Timer
Synchronization	Atomic instructions
Virtual memory	Translation look-aside buffers

Today we'll introduce the organization and components in an OS. The rest of the course will fill in the details about each component and how they fit together.

1. **OS Components**
   - **Processes**
   - Synchronization
   - Memory & Secondary Storage Management
   - File Systems
   - I/O Systems
   - Distributed Systems
2. **Three example OS organizations**
   - Monolithic kernel
   - Layered architecture
   - Microkernel

From the Architecture to the OS to the User

Architectural resources, OS management, and User Abstractions

Hardware	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, Transparent file sharing

Processes

- The OS manages a variety of activities:
  - User programs
  - Batch jobs and command scripts
  - System programs: printers, spoolers, name servers, file servers, network listeners, etc.
- Each of these activities is encapsulated in a **process**.
- A process includes the execution context (PC, registers, VM, resources, etc.) and all the other information the activity needs to run.

$\Longrightarrow$ A process is not a program. A process is one instance of a program in execution. Many processes can be running the same program. Processes are independent entities.

- The OS creates, deletes, suspends, and resumes processes.
- The OS schedules and manages processes.
- The OS manages inter-process communication and **synchronization**.
- The OS allocates resources to processes.

Synchronization Example:

Banking transactions

- Cooperating processes on a single account: ATM machine transaction, balance computation, Monthly interest computation and addition.

$\longrightarrow$ All of the processes are trying to access the same account simultaneously. What can happen?

Memory & Secondary Storage Management

Main memory

- is the direct access storage for the CPU.
- Processes must be stored in main memory to execute.
- The OS must
  - allocate memory space for processes,
  - deallocate memory space,
  - maintain the mappings from virtual to physical memory (page tables),
  - decide how much memory to allocate to each process, and when a process should be removed from memory (policies).

File System

Secondary storage devices (disks) are too crude to use directly for long term storage.

- The file system provides logical objects and operations on these objects (files).
- A file is the long-term storage entity: a named collection of persistent information that can be read or written.
- File systems support directories which contain the names of files and other directories along with additional information about the files and directories (e.g., when they were created and last modified).

The File System provides *file management*, a standard interface to

- create and delete files and directories
- manipulate (read, write, extend, rename, copy, protect) files and directories
- map files onto secondary storage

The File System also provides general services such as backups, maintaining mapping information, accounting, and quotas.

Secondary Storage (disk)

Secondary Storage is

- the persistent memory, i.e., it endures system failures (we hope).
- Low-level OS routines are typically responsible for low-level disk functions, such as scheduling of disk operations, head movement, and error handling.
- These routines may also be responsible for managing the disk space (for example, keeping track of the free space).
- The line between managing the disk space and the file system is very fuzzy, these routines are sometimes in the file system.

Example: A program executable is stored in a file on disk. To execute a program, the OS must load the program from disk into memory.

I/O Systems

The I/O system supports communication with external devices: terminal, keyboard, printer, mouse, ...

The I/O system

- Supports buffering and spooling of I/O
- Provides a general device driver interface, hiding the differences among devices, often mimicking the file system interface
- Provides device driver implementations specific to individual devices.

Distributed Systems

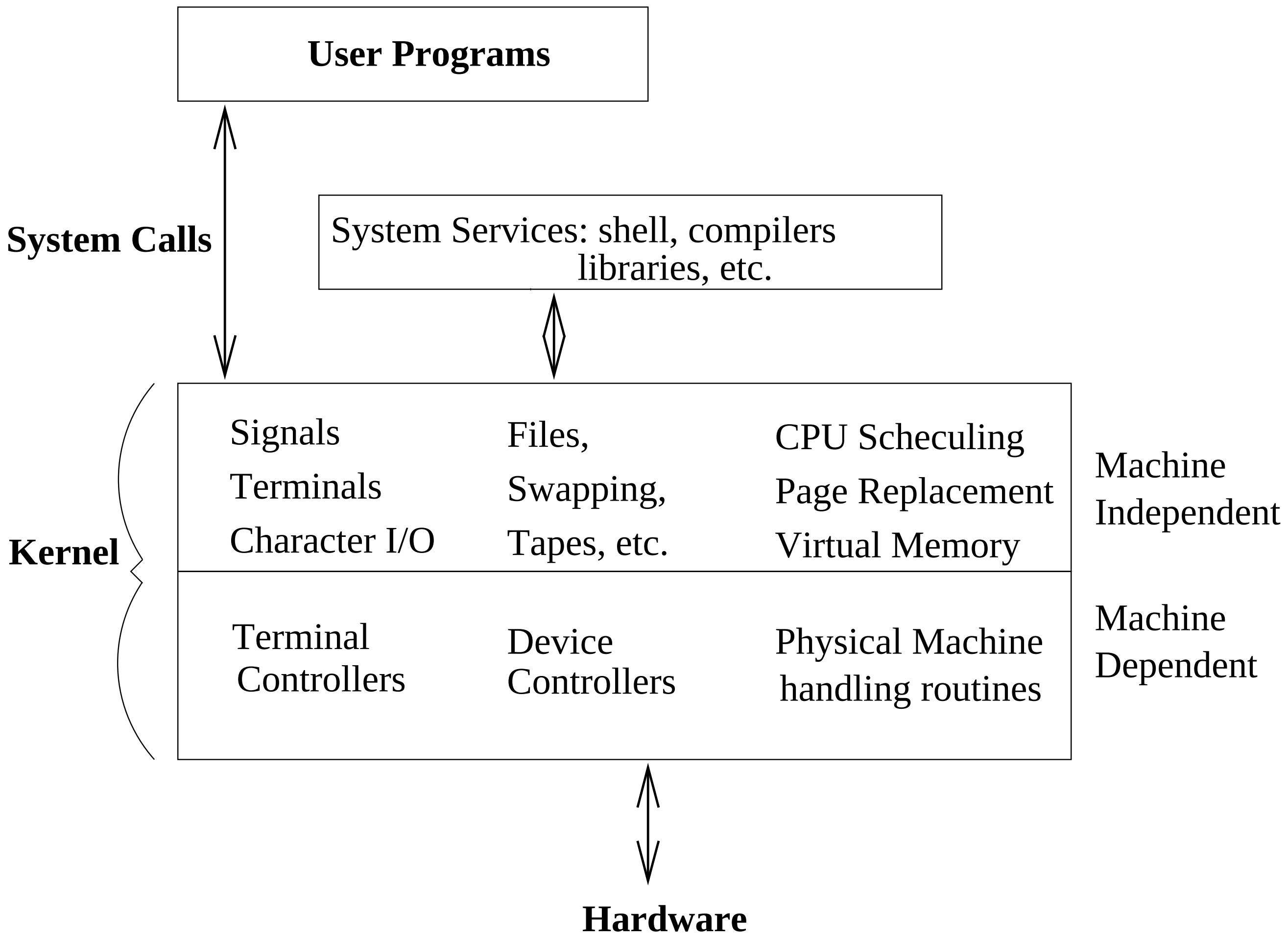
A distributed system is a collection of processors that do not share memory or a clock.

- To use non-local resources in a distributed system, processes must communicate over a network,
- The OS must provide additional mechanisms for dealing with failures and deadlock that are not encountered in a centralized system.

The OS can support a distributed file system on a distributed system.

- Users, servers, and storage devices are all dispersed among the various sites.
- The OS must carry out its file services across the network and manage multiple, independent storage devices.

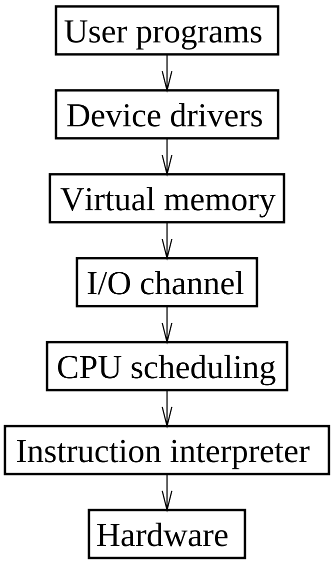
One Basic OS Structure



The *kernel* is the protected part of the OS that runs in kernel mode, protecting the critical OS data structures and device registers from user programs.

$\Longrightarrow$ There is a lot of debate about what functionality should go into the kernel – this is just one possible organization, used by original Unix OS.

Layered OS design



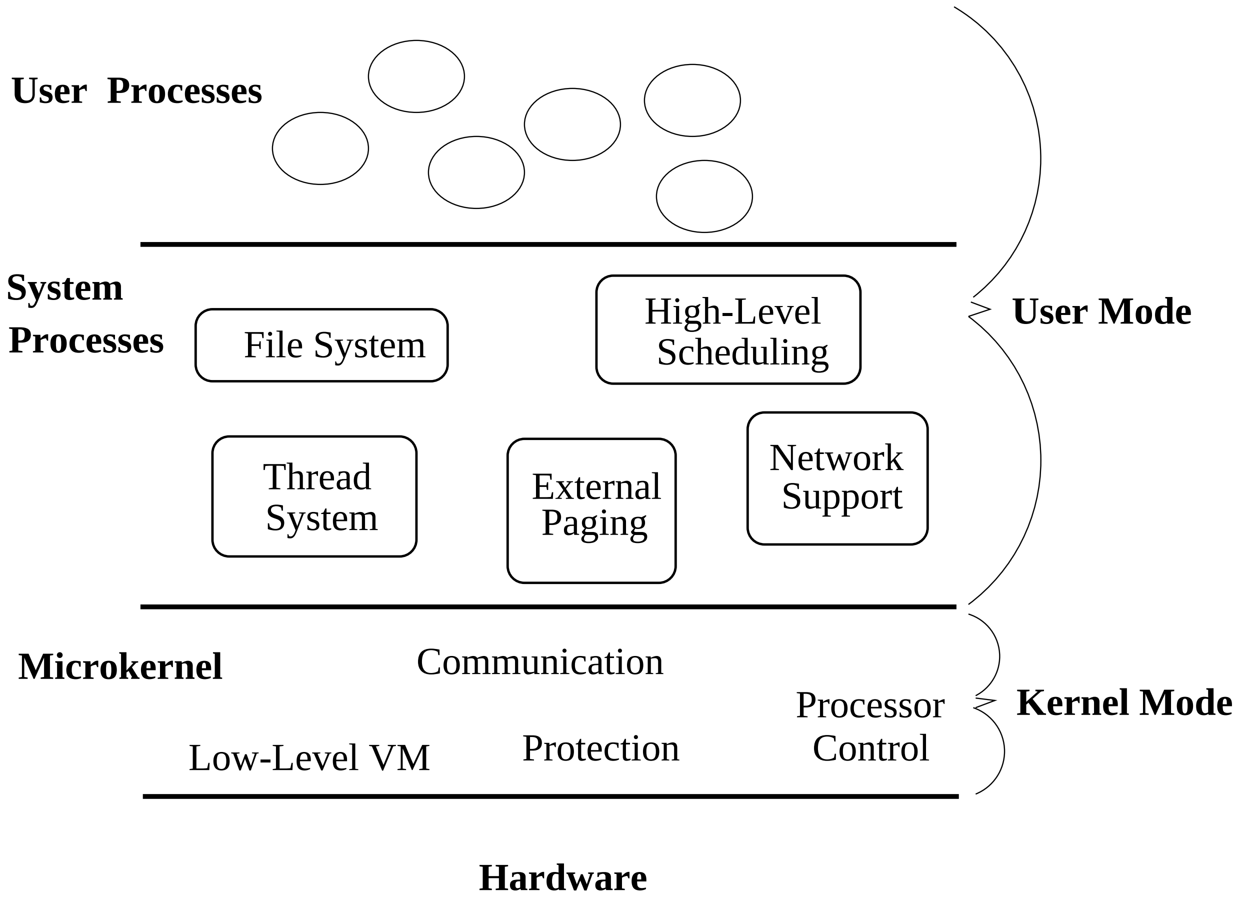
Advantages of a layered design

- Modularity
- Simplicity
- These make the OS easier to design, debug, and port to new machines.

Disadvantage - Performance

- Overhead in communicating between layers.
- Extra copying, parameter passing, bookkeeping, etc.

Microkernel



Goal is to minimize what goes in the kernel (mechanism, no policy), implementing as much of the OS in User-Level processes as possible. This results in

- better reliability, easier extension and customization
- mediocre performance (unfortunately)

First Microkernel was Hydra (CMU '70). Later systems include Chorus (France) and Mach (CMU).

Summary

Big Design Issue: How do we make the OS efficient, reliable, and extensible?

General OS Philosophy: The design and implementation of an OS involves a constant tradeoff between *simplicity* and *performance*. As a general rule, strive for simplicity except when you have a strong reason to believe that you need to make a particular component complicated to achieve acceptable performance (strong reason = simulation or evaluation study).

- 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?

What’s in a Process?

A **process** is the dynamic execution context of an executing program.

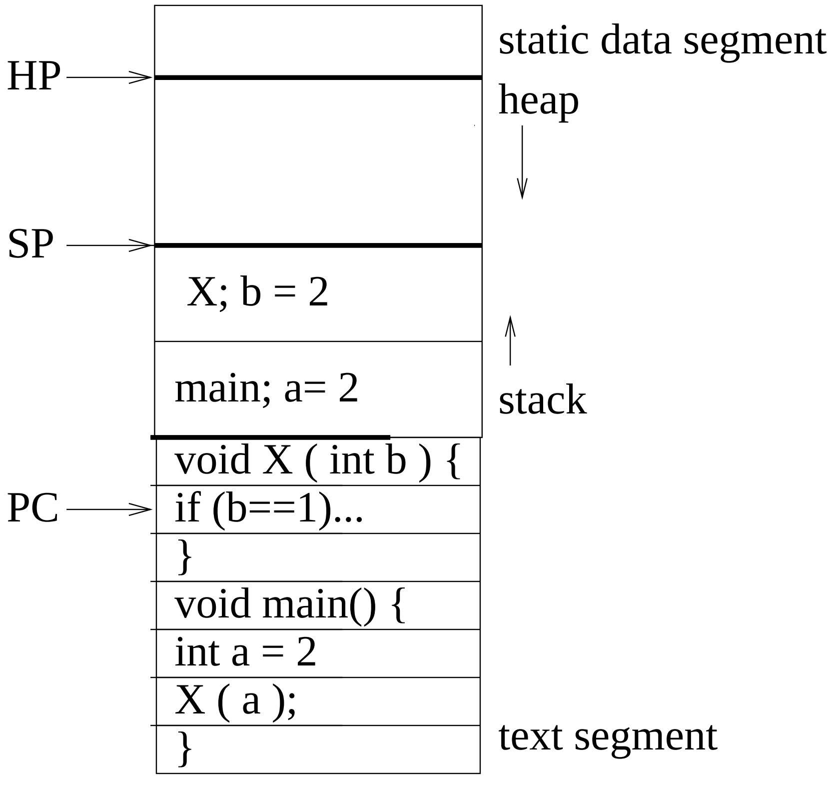
- Several processes may run the same program, but each is a distinct process with its own state (e.g., emacs).
- A sequential process is a program in execution that yields a sequential, one instruction at a time, ordering of the instructions.
- Process state consists of at least:
  - the code for the 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, connections to other programs, etc),
  - 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:



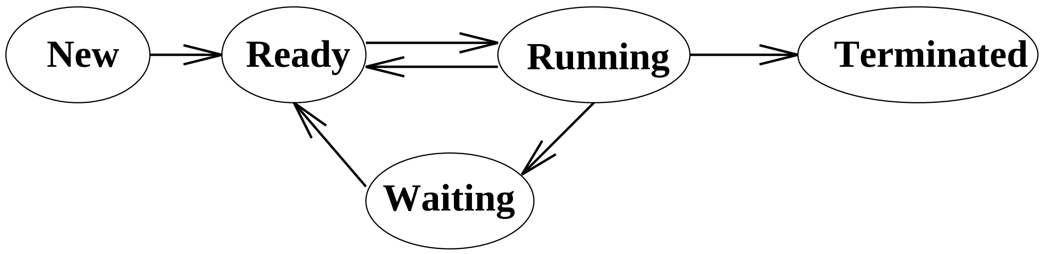
Process State

Process Execution State

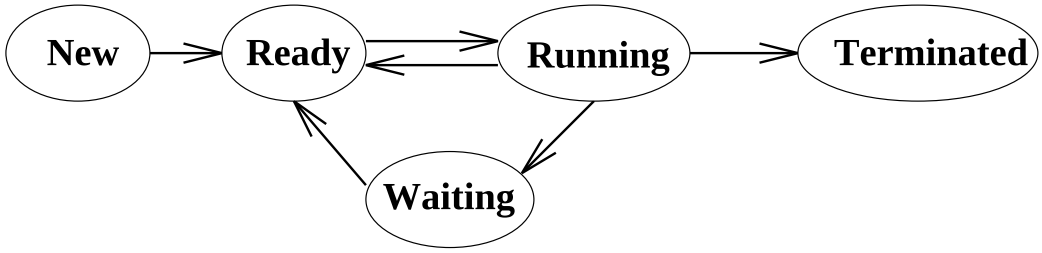
Each process has an *execution state* which indicates what it is currently doing.

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

Since multiple processes are active at any one time, the OS manages them by placing all the processes in the same state in a *state queue* (e.g., the *ready queue*).

Process Data Structures

The OS uses a *Process Control Block* (PCB), its own dynamic data structure, to keep track of all the processes (this data structure is in addition to the process state in memory). The PCB represents the execution state and location of each process when it is not executing (e.g., when it is *waiting*).

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 (e.g., the *waiting queue*)
- Scheduling information (e.g., priority)
- I/O status
- . . .

When a process is created, the OS allocates and initializes a new PCB, and then places the PCB on a state queue.

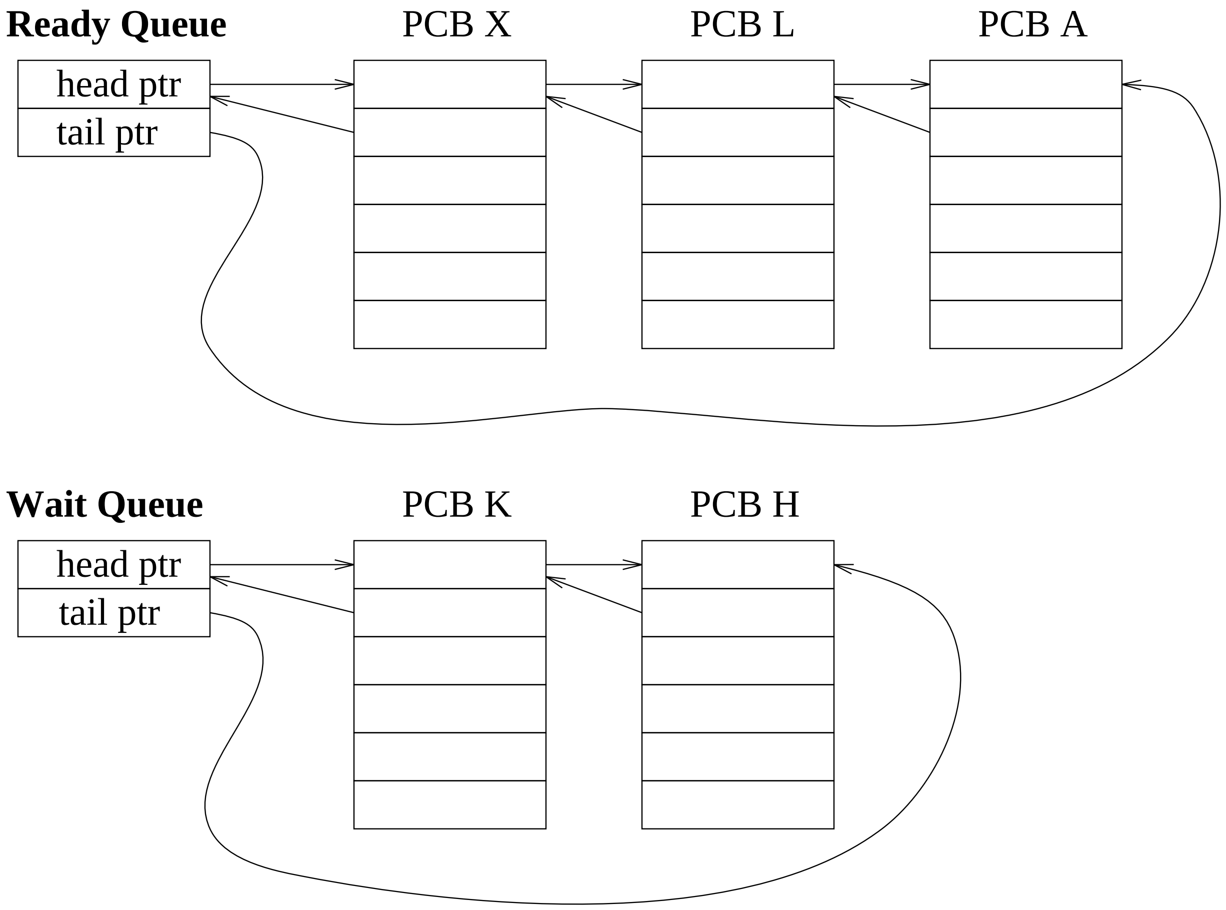
When a process terminates, the OS deallocates the PCB.

Process State Queues

The OS maintains the PCBs of all the processes in the system by using *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.

Process State Queues



PCBs and Hardware State

Starting and stopping processes is called a **context switch**, and is a relatively expensive operation.

- When the OS is ready to start executing a waiting process, it loads the hardware registers (PC, SP, and other registers) from the values stored in that process' 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 the PCB for that process.
- This action 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 to do the context switch and the time between process switches are closely related.)

Creating a Process

- One process can create other processes to do work. These are *child* processes, and the creator is the *parent* process.
- In some systems, the parent defines (or donates) resources and privileges to its children.
- When a parent creates a child, it can either wait for the child to complete, or continue in parallel.
- In Unix, subprocesses are created by a call to the system routine *fork*
  - the variable values and program counter are copied from the parent to the new child process
  - the return value of *fork* is different in the parent and child process. In the parent process, the return value is the child process id. In the child process, the return value is 0. *This is the only difference between the parent and child processes.*
  - The parent can wait for the child to terminate by executing the *wait* system call to wait for the process identified by fork to terminate, or
  - The parent can continue to execute.
  - The child often starts a new and different program within itself, via a call to *exec*, another Unix 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() forks a new child process that is a copy of the parent.

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

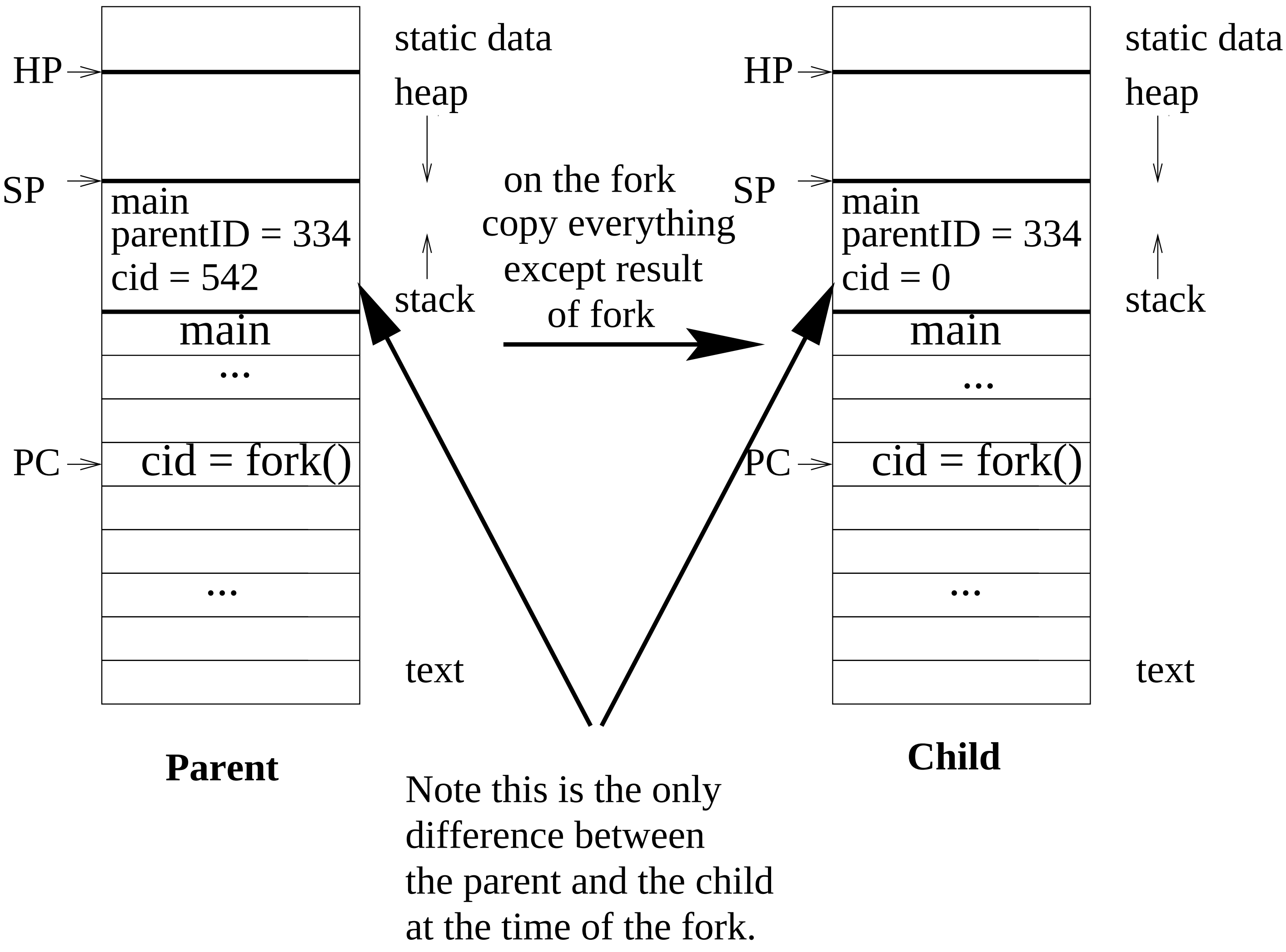
sleep() suspends execution for at least the specified time.

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

gets() reads a line from a file.

```
#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 program running
       in the child process 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);
} }
```

What is happening on the Fork



Process Termination

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

```
#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  */
    /* I'll exit myself after 5 seconds. */
    sleep (5);
    printf ( "Quitting child\n" );
    exit (0);
    /* We should never get here. */
    printf ( "Ack!  After exit call.!");
  } else {  /*  I'm the parent process  */
    /* The user can kill the child by typing any character. */
    printf ( "Type any character to kill the child.\n" );
    char answer[10];
    gets (answer);
    if ( !kill(cid, SIGKILL) ) {
      printf("Killed the child.\n");
} } }
```

- A process is the unit of execution.
- Processes are represented as Process Control Blocks in the OS that encapsulate program state, memory management information, scheduling information, I/O status, etc.
- At any time, 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