Processes

System Calls

Gives privileged access to the hardware. This is to restrict users from not messing with the hardware directly. It is called "Kernel Mode" in *NIX systems. Allows users to use kernel-level operations. Most users are not aware that they are using kernel operations. System calls tell the OS that the user wants to do something privileged. Some system calls:

1. `fork()`
2. `wait()`
3. `exit()`

Unix Processes

> *A program that has started is manifested in the context of a process*

It is represented by:

- Identification
- State info
- Control info
- User stack
- Private user addr, program, and data space
- Shared addr space

All of this is maintained by the OS in a *Process Control Block*

UNIX Process states

1. *User running*: Process executes in user mode
2. *Kerner running*: execs in kernel mode
3. *Ready*: waiting to be scheduled
4. *Asleep*: waiting for an even (I/O)
5. *Preempted*: the process was yanked for another process
6. *Created*: newly created not ready to run
7. *Zombie*: process has ended, but kept in mem for a parent to collect

Creating process

New processes in UNIX are created by `fork()`. On calling fork, the OS does:

1. makes new PCB in table
2. assigns new ID to the child
3. makes a copy of parent into the new PCB (except for shared memory)
4. returns 0 to the child and the pid of the child to the parent

> *This process is simple because we `fork` processes' a lot.*

This creates a tree of processes. UNIX has a root proc, from which all subsequent proc are forked. Parents can block or not block the child.

Process coordination

> *Nice point of synchronization between processes.*

`exit(i)` is used to terminate a process, `i` is returned as the status. `wait()` is used to temporarily suspend the parent process until ay one child proc terminates.

The signature of `wait()` is `pid_t wait(int *status)` where the exit code is stored in status. It also returns the pid of the child. returns -1 when no child exits.

If the parent is alive, and the child exits, the child enters a *zombie/defunct* state. But if the parent is not alive, the child dies.

In non-blocking programs, the functionality of `wait` is handled by signals/events. UNIX Kernel notifies the parent that the child has terminated by using the `SIGCHLD` signal. Termination of a child is an asynchronous event. This is because blocking is mostly a wait of resources. The function that asynchronously runs on receipt of a signal, is a signal handler. The default action of `SIGCHLD` is to do nothing.

A process that calls `wait` or `waitpid` can block if all of its children are still running. It returns immediately if:

1. A child has been terminated and is waiting for its termination status to be fetched. Returns with the pid and termination status of the child
2. No child processes. Returns with errors

If wait is called because it received a SIGCHLD signal, we expect wait to return immediately. But if called at any random point of time, wait may block

```c
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Other system calls

`getpid()` and `getppid()` get the pid of the proc and its parent resp. Zombie procs can be identified by the `ps` command.

`kill` and `killall`

It sends signals to processes of a given pid.

```
kill [options] pid
```

`-l` lists all signal numbers. `-<number>` sends the corresponding signal. To terminate a proc, send it SIGINT, which has the number 9.

```
kill -9 24607 # Kills pid 24607
```

`killall` sends a signal to all user-owned processes.
