UNIX Process

- Program vs. Process (Process ≠ Program)
  - A program is a set of machine code instructions and data stored in an executable image on disk, not in memory yet
  - A process is a program in action / execution, i.e. a running program, the program is loaded in memory
- UNIX is a multiple processes (multi-users) Operating System
  - Many processes are kept in memory at the same time
  - Each individual process runs in its own virtual address space
  - Operating system (the kernel) manages all the processes, scheduling the processes to share the resources, etc.
    - A scheduler uses a number of scheduling strategies to ensure fairness, such as deciding which process to run next
- Each process has a unique ID, called Process ID (PID)
- Each process also has one parent process, PPID

`ps` - Get a Snapshot of all the Current Active Processes

- Without any options, it gives the processes running in the shell where the command (ps) is being executed

```
  PID TTY          TIME CMD
10289 pts/4    00:00:00 bash
10520 pts/4    00:00:00 ps
```

- with -f: **`ps –f`**

```
UID        PID  PPID  C STIME TTY          TIME CMD
hlin     10289 10288  0 14:06 pts/4    00:00:00 -bash
hlin     10526 10289  0 14:22 pts/4    00:00:00 ps -f
```

- See EVERY process on the system
  - using Unix (System V)-style (with short dash)
    - **`ps –e; ps –ef; ps –A;`**
  - using BSD-style (without the short dash)
    - **`ps aux; ps ax`**

Process Tree with "pstree"

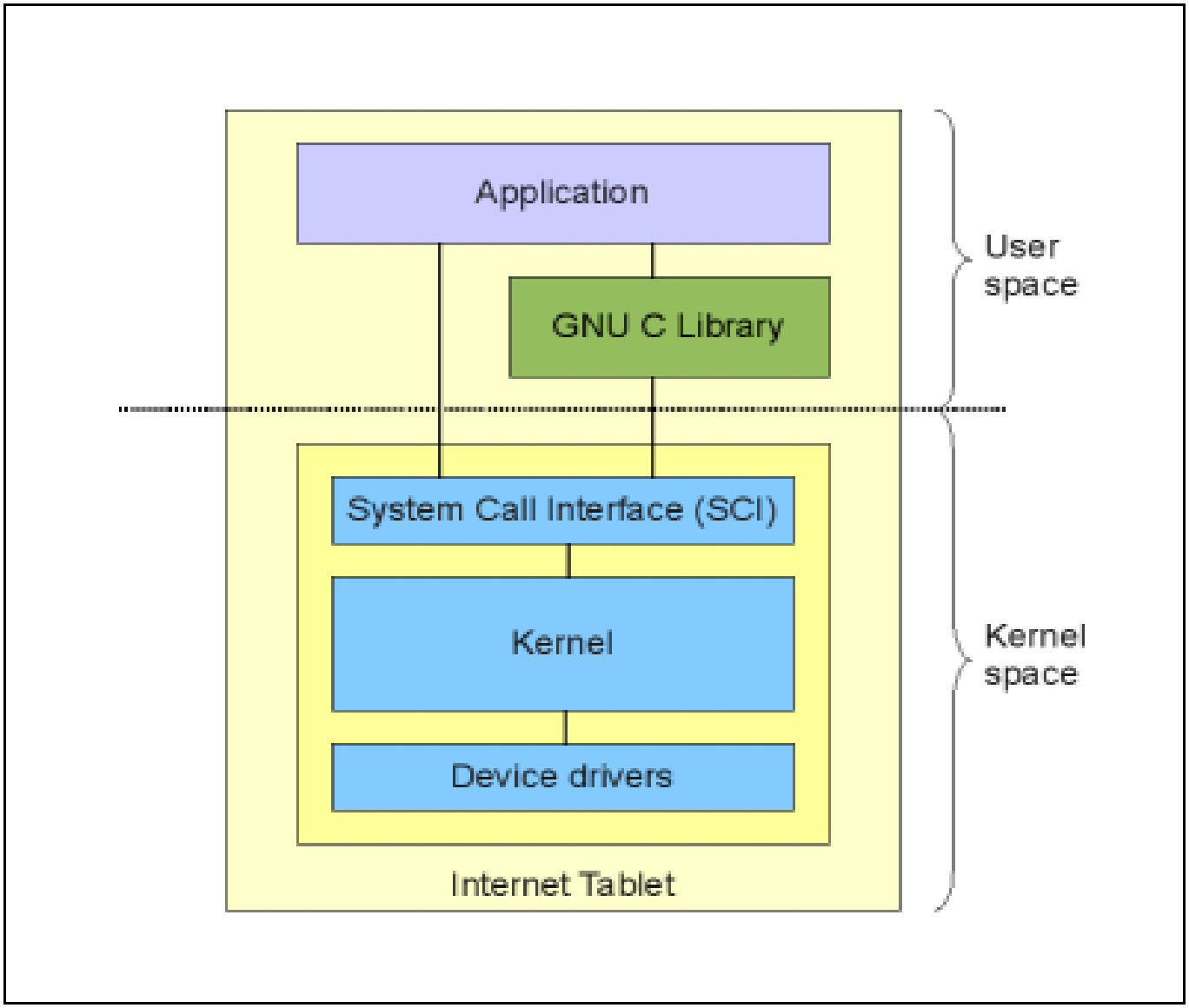
- Shows running processes as a tree with the parent-child relationship

```
init─┬─accounts-daemon───2*[{accounts-daemon}]
     ├─acpid
     ├─at-spi-bus-laun─┬─dbus-daemon
     │                 └─3*[{at-spi-bus-laun}]
     ├─avahi-daemon───avahi-daemon
     ├─colord───2*[{colord}]
     ├─console-kit-dae───64*[{console-kit-dae}]
     ├─cron
     ├─5*[getty]
     ├─gvfsd───{gvfsd}
     ├─irqbalance
     ├─libvirtd───10*[{libvirtd}]
     ├─lightdm─┬─Xorg
     │         ├─lightdm─┬─lightdm-greeter───lightdm-gtk-gre───2*[{lightdm-gtk-gre}]
     │         │         └─{lightdm}
     │         ├─lightdm
     │         └─2*[{lightdm}]
     ├─login───bash
     ├─lpd
```

- "**pstree**" is for Linux systems : **pstree -pu**
- **"top":** display Linux processes

UNIX Process Execution Modes

- Process execution modes => The state of a CPU
  - Two Modes: User Mode & System mode (kernel mode)
- **User mode**
  - When the CPU is executing the code for a user program which accesses its own (user) data space
- **System mode**, also called kernel mode
  - The state of a CPU where the kernel needs to ensure that it has privileged access to data and physical devices.
  - Runs on behalf of a user process and is a part of the user process
- Switch from user mode to system mode by making system calls
  - Code running in user mode must delegate to system APIs to access hardware or memory
- **System call**
  - Is a fundamental interface b/w an application and OS (kernel)
  - Is a request by user program for kernel services
  - Use man page: **man syscalls** to learn more and a list of system calls on that Linux system



http://www.linfo.org/kernel_mode.html

Process Creation Mechanism

- On UNIX, process creation, execution and termination are done by a set of four system calls
  - **`fork`**, **`wait`**, **`exec`**, and **`exit`**
- The **`fork`** system call
  - A process is created in UNIX with the **`fork()`** system call
    - **`fork`** creates a duplicate process of the calling process with a new PID
    - The calling process is called the parent process
    - The duplicate process is the child process of the calling one
  - A parent process can have many child processes, but a child process can only have one parent process
  - The child process starts right after the call of fork. It is responsible to do the work, such as the command called through command line
- The **`wait`** system call
  - Suspend the parent process after creating the child process with the **`fork`** system call

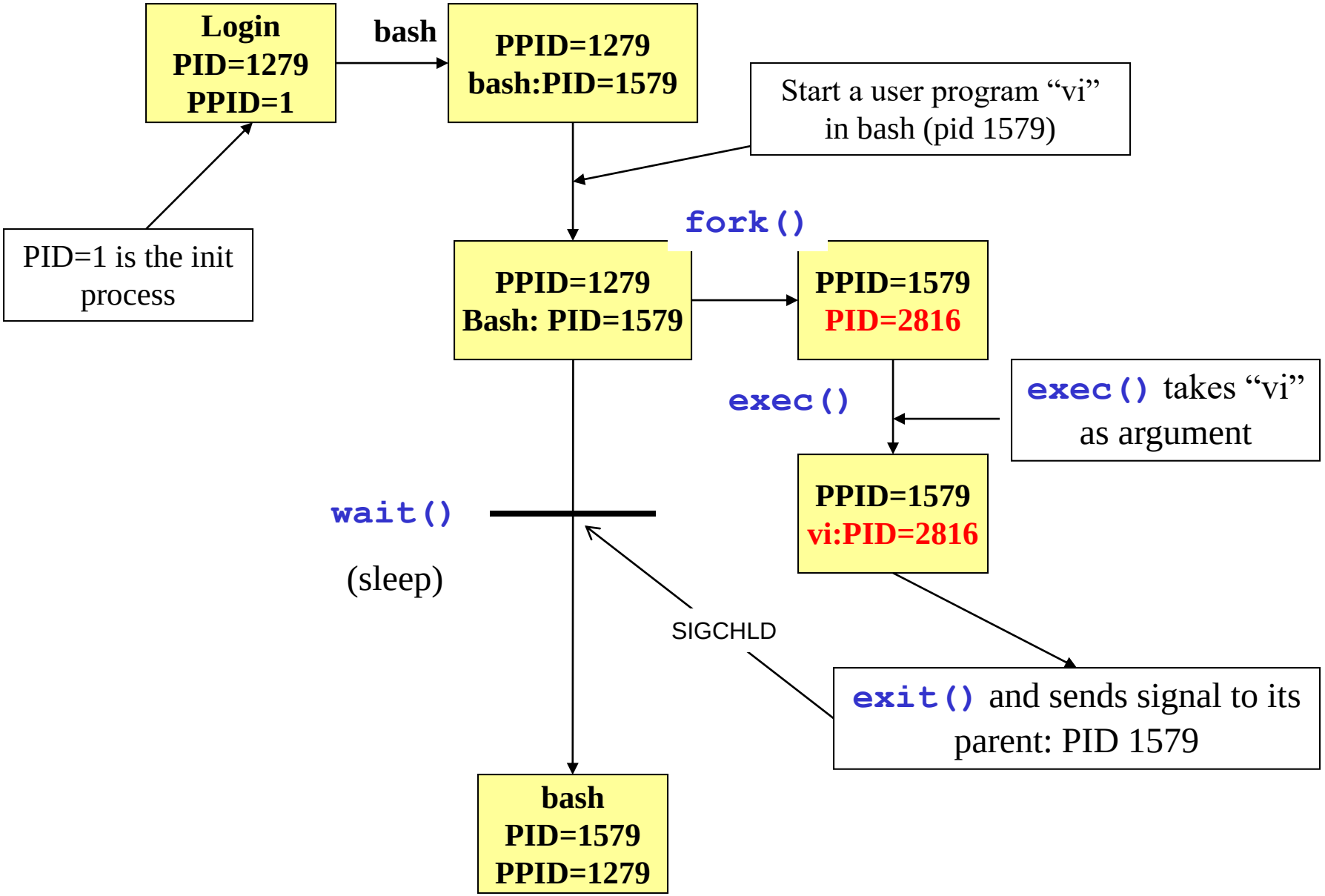
- The **exec** system call

  Real execution is done by the child process through the **exec** system call

  - The (forked) child process calls **exec** with the name of the command as its argument
  - The kernel loads this new program into memory in place of the process (the child) that calls it and becomes the child process (hold its PID) and starts executing.

- The **exit** system call

  - When the child process terminates, it calls **exit** system call, and sends a signal (SIGCHLD) to its parent,
  - The parent wakes up from the **wait** when it receives the SIGCHILD signal, and you get your shell back in case of command line…


Login
PID=1279
PPID=1
bash
PPID=1279
bash:PID=1579
Start a user program "vi"
in bash (pid 1579)
PID=1 is the init
process
fork()
PPID=1279
Bash: PID=1579
PPID=1579
PID=2816
exec()
exec() takes "vi"
as argument
PPID=1579
vi:PID=2816
wait()
(sleep)
SIGCHLD
exit() and sends signal to its
parent: PID 1579
bash
PID=1579
PPID=1279

Process Termination

- Control-Key
  - `Ctrl-C`: send SIGINT (interrupt) signal
  - `Ctrl-\`: send SIGQUIT signal
    - stronger than ctrl-C, used when ctrl-C does not work
- Shell cmd `kill` with the following arguments and signals
  - PID=12345: `kill 12345` (how to get the pid?)
  - Process name: `killall foo`
  - Options `(send a signal to the process)`
    - `-TERM`: termination signal, by default (15)
    - `-QUIT`: quit signal (3)
    - `-KILL`: `kill -9 12345`
    - `-s signal(number of name) pid`
    - the most strongest signal (SIGKILL), the OS should terminate the process immediately and unconditionally. (SIGKILL is a deadly force!)
- **`kill -l`**
  - List all the available signals on the system

A fork/exec Program

```
gcc -o test fork.c
./test
PID=2367, var=88
Child process, pid=2368, ppid=2367
Thu Sep 11 14:50:13 CDT 2008
Child process is done, var = 88,
status=0
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void){
    int      var = 88;
    pid_t    pid;
    int status;
    printf("PID=%d, var=%d\n", getpid(),var);
    fflush(stdout);
    pid = fork();
    if (pid < 0) { printf("fork error\n");
    } else if (pid == 0) {          /* child */
        printf("Child process, pid=%d, ppid=%d\n", getpid(), getppid());
        var++;          /* modify variables */
        sleep(2);
        execl("/bin/date", "date", NULL); // try with and without this line
    }
    wait(&status);
    printf("Child process is done, var = %d, status=%d\n", var, status);
    exit(0);
}
```

Fore- & Back-ground Processes

- UNIX as a multitask OS, lets you run many jobs in the background while you can do something else in the foreground
  - Foreground processes, the shell running the process has to wait for the termination of the running process.
  - Background jobs, the shell has not to wait for the end of the process. The shell can run as many background processes as the system allows
- Running a background process with &
  - **&**: PUT an ampersand (&) at the end of the command line

    ```
    mining > output.txt 2>&1 &
    ```

  - **nohup**:

    ```
    nohup mining &   # The output will be saved in nohup.out
    ```

Process ID & Job Numbers

```
hlin@dakota:~/test> sleep 10 &
[1] 30796
```

- Job is a group of processes:
  - ls |wc two processes in one job
- [1]-Job number refers to the background processes that are running under the current shell
- 30796 is the PID, a unique number system wide
- $!: the most recently job (PID) put in the background
  - `echo $!` ➔ `30796`
- Job number can be used to kill a background process
  - `kill %1`
- Of course, it can be terminated using
  - `kill 30796`

More about bg & fg processes

- A running background job can be brought to front
  - `fg 12345`
- A running foreground job can be sent to background in two steps
  - 1st, suspend the foreground job first with ctrl-z,
    - you regain the control of the terminal
  - 2nd , run "`bg`" will send the suspended job to run in background

Process Priority

- OS schedules the processes based on their priorities
  - Processes with higher priority will run before those with a lower priority
  - Processes with the same priority are scheduled round robin
  - By default, the priority number is set to be your shell priority
- Modifying process priority
  - Run program with a different process priority
    - `nice program`
      - By default, nice add 10 to your current shell priority
    - `nice –n 20 program`
      - Add 20 to our current shell priority
  - Changing the process priority of a running process
    - `renice -10 –u hlin`
      - Increase by 10 for all process belonging to user hlin
    - `renice -10 12345`
      - Increate priority by 10 for process 12345
  - The niceness value ranges [-20,19] (lowest having highest priority