Chapter 3 Processes - Notes

3.3 Learning Objectives:

- Describe a process and the resources associated w ith it.
- Describe the role of the **init** process.
- Distinguish betw een processes, programs, and threads.
- Understand process attributes, permissions, and states, and know how to control limits.
- Explain the difference betw een running in user and kernel modes.
- Describe the **daemon** processes.
- Understand how new processes are forked (created).
- Use **nice** and **renice** to set and modify priorities.
- Understand how shared and static libraries are used.

3.4 Processes, Programs, and Threads

Process: executing program and associated resources, including environment, open files, signal handlers, etc. Same program may be executing more than once simultaneously -> responsible for multiple processes.

Tw o or more tasks (threads of execution) can share various resources (eg. entire memory spaces or particular memory areas, open files, etc.). When **everything shared** circumstance -> **multi-threaded** process.

Other OS -> distinction betw een full **heavy weight** and **light weight** processes. Heavy w eight process may include number of light w eight processes, or just one.

Linux -> different situation. Each thread of execution considered individually, difference betw een heavy and light -> sharing of resources, somew hat faster context sw itching betw een threads of execution.

Linux -> fast job of creating, destroying, sw itching betw een processes (unlike other OS) -> model adopted for multi-threaded application resembles multiple processes: each thread scheduled individually and normally (like stand-alone process). Done instead of involving more levels of complication (eg. having separate method of scheduling along threads of process, having scheduling method betw een different processes).

Linux -> also respects POSIX and other standards for multi-threaded processes, eg. each thread returns same process ID (called thread group ID internally) w hile returning distinct thread ID (called process ID internally). May lead to confusion for developers, invisible to administrators.

3.5 The init Process

First user process on system -> **init**, has `process ID` **= 1**. Started as soon as kernel initialized + root filesystem mounted.

Runs until system shut dow n -> last user process to be terminated. Serves as ancestral parent of all other user processes, both directly/indirectly.

3.6 Processes

Process -> instance of program in execution. Number of different states (eg. running, sleeping). Every process has **pid** (Process ID), **ppid** (Parent Process ID), **pgid** (Process Group ID); also has program code, data, variables, file descriptors, environment.

init: first user process run on system, becomes ancestor of all subsequent processes running on system (except for those initiated directly from kernel, show up with **[]** around name in **ps** listing).

If parent process dies before child -> **ppid** of child set to 1 (ie. process id adopted by **init**). (Note: in recent Linux systems using systemd, **ppid** set to 2, corresponding to internal kernel thread **kthreadd** that has taken over role of adopter of orphaned children from **init**).

zombie (or defunct) process: child process which terminated (either normally/abnormally) before parent that has not waited for it and examined its exit code. Releases almost all resources and remains only to convey exit status. Checking on adopted children + let terminated process die gracefully -> one function of **init** process -> sometimes known as **zombie killer**, or **child reaper** (yeesh).

Processes controlled by **scheduling** (completely preemptive). Only kernel has right to preempt process, processes cannot do it to each other.

Largest `PID` limited to 16-bit number, 32768, for historical reasons. Possible to alter this value by changing `/proc/sys/kernel/pid_max`, since may be inadequate for larger servers. As processes created, will eventually reach `pid_max`, and then will start again at `PID = 300`.

3.7 Process Attributes

Attributes for all processes:

- Program being executed
- Context (state)
- Permissions
- Associated resources

Every process -> executing some program.

Context of process: snapshot of itself by trapping state of its CPU registers, where it is executing in program, what is in process' memory, and other information.

Processes scheduled in and out when sharing CPU time with other (or put to sleep while waiting for some condition to be fulfilled, such as user request or data arrival) -> ability to store entire context when swapping out process, and restore upon execution resumption *critical* to kernel's ability to do **context switching**.

3.8 Controlling Processes with ulimit

ulimit: built-in bash command, displays or sets a number of resource limits associated with processes running under shell.

Screenshot: running **ulimit** with `-a` argument.
ulimit

If run as root, result in `Command not found`, since limits shell-specific.

System administrator may need to change some above values in either direction to:

- **Restrict** capabilities so user/process cannot exhaust system resources (eg. memory, cpu time, max number of processes on system)
- **Expand** capabilities so process does not run into resource limits, eg. server handling many clients -> default of 1024 open files, impossible to perform work

Two kinds of limits:

- **Hard**: max value that user can raise resource limit to. Set only by root user
- **Soft**: current limiting value. Can be modified by user, but cannot exceed hard limit

Can set any particular limit by:

```
$ ulimit [options] [limit]
```

as in

```
$ ulimit -n 1600
```

w hich increases max number of file descriptors to 1600.

Note: changes only affect current shell. TO make changes effective for all logged-in users, need to modify `/etc/security/limits.conf` (nicely self-documented file), and reboot.

3.9 Process Permissions and setuid

Every process -> permissions based on w hich user invoked + on w ho ow ns program file.

Programs w ith **s** execute bit -> have different **effective user id** than their **real user id** (explained later in local security section). Referred to as **setuid** programs. Run w ith user id of user w ho **owns** program. Non-**setuid** pgroams run w ith permissions of user w ho **runs** them.

setuid programs ow ned by root -> w ell-know n security problem.

passwd program -> example of setuid program. Any user can run. When user executes program, process must run w ith root permission to update w rite-restricted `/etc/passwd` and `/etc/shadow` files w here user passw ords maintained.

3.10 More on Process States

Processes can be in on of several possible states. Main ones:

- **Running**: process currently executing on CPU/CPU core, or sitting in **run queue** w aiting new time slice. Will resume w hen scheduler decides deserving to occupy CPU, or w hen another CPU idle and scheduler migrates process to idle CPU.
- **Sleeping** (ie, **Waiting**): w aiting on request (usually I/O) that it has made + cannot proceed until request completed. When request completed, kernel w ill w ake up process, put back on run queue, given time slice on CPU w hen scheduler decides to do so.
- **Stopped**: suspended process. Commonly experienced w hen programmer w ants to examine executing program's memory, CPU registers, flags, other attributes. One examination done, process may be resumed. Generally done w hen process run under debugger or user hits `Ctrl-Z`.
- **Zombie**: states w hen process terminates, and no other process (usually parent) inquires about its exit state (ie, reaped it). ALso called **defunct** process. Has released all its resources, except exit state and entry in process table. If parent of any process dies, process **adopted** by **init** (`PID = 1`) or **kthreadd** (`PID = 2`).

3.11 Execution Modes

Process (or any particular thread of multi-threaded process) -> may be executing in **user mode** or **system mode** (usually called **kernel mode** by kernel developers) at any given time.

Instructions that can be executed -> depends on mode + enforced at hardw are level (not softw are).

Mode: state of processor (eg. in multi-core/multi-CPU system, each unit can be in ow n individual state). *Not* state of system.

Intel parlance:

- User mode: **Ring 3**
- System mode: **Ring 0**

3.12 User Mode

Processes execute in **user mode** (lesser privileges than in kernel mode), *except* w hen executing **system call** (described in next section).

When process started -> isolated in ow n user space to protect from other processes -> promotes security + greater stability. Sometimes called **process resource isolation**.

Each user mode process -> ow n memory space, w here parts may be shared w ith other processes. Except shared memory segments, user process not able to read/w rite into/from memory space of any other process.

Even process run by root user or as **setuid** program -> runs in user mode, *except* w hen jumping into system call. Has only limited ability to access hardw are.

syscall

3.13 Kernel Mode

Kernel (system) mode -> CPU full access to all hardw are on system including peripherals, memory, disks etc. If application needs access to these resources, must issue **system call** -> causes **context switch** from user mode to kernel mode. Must follow this procedure w hen reading/w riting files, creating new process etc.

Application code -> never runs in kernel code, only system call itself (w hich is kernel code). When system call complete, return value produced -> process returns to user mode w ith inverse context sw itch.

Other times w hen system is in kernel mode that have nothing to do w ith processes -> handling hardw are interrupts, running scheduling routines, other management tasks for system.

3.14 Daemons

Daemon process: background process, sole purpose to provide specific service to users of system:

- Can be quite efficient because only operate w hen needed
- Many started at boot time
- Names often (but not alw ays) end w ith **d**
- Examples: **httpd**, **systemd-udevd**
- May respond to external events (**systemd-udevd**) or elapsed time (**crond**)
- Generally have no controlling terminal and no standard input/output devices
- Sometimes provide better security control

When using SysVinit, scripts in `/etc/init.d` directory start various system daemons. These scripts invoke commands as arguments to shell function named **`daemon`**, defined in `/etc/init.d/functions` file.

3.15 Creating Processes in a Command Shell

When user executes command in command shell interpreter such as **bash**:

- New process created (forked from user's login shell)
- Wait system call puts parent shell process to sleep
- Command loaded onto child process' space via **exec** system call (other w ords, code for command replaces **bash** program in child process' memory space)
- Command completes executing, child process dies via exit system call
- Parent shell re-aw akened by death of child process, proceeds to issue new shell prompt. Parent shell then w aits for next command request from user, at w hich time cycle repeated

If command issued for **background** processing (by adding ampersand - `&` - at end of command line), parent shell skips w ait request and free to issue new shell prompt immediately, allow ing background process to execute in parallel. Otherw ise, for **foreground** requests, shell w aits until child process completes or stopped via signal.

Some shell commands (eg. echo, kill) built into shell itself, do not involve loading of program files. For these commands, no **fork** or **exec** issued for execution.

3.16 Kernel-Created Processes

Not all processes created, or **forked** from user parents. Linux kernel directly creates tw o kinds of processes on ow n initiative:

- **Internal kernel processes**: take care of maintenance w ork (eg. making sure buffers get flushed out to disk, load on different CPUs balanced evenly, device drivers handle w ork queued up for them to do, etc.). Often run as long as system running, sleeping except w hen they have something to do.
- **External user processes**: run in user space like normal applications, but started by kernel. Very few, usually short lived.

Easy to see w hich processes are of this nature; w hen you run command such as

```
$ ps -elf
```

to list all processes on system w hile show ing parent process IDs, all w ill have `PPID = 2` referring to **kthreadd**, internal kernel thread w hose job is to create such processes, names w ill be encapsulated in square brackets such as `[ksoftirqd/0]`.

3.17 Process Creating and Forking

Average Linux system -> alw ays creating new processes. Often called **forking**; original parent process continues running w hile new child process starts.

Back w hen most computers had single processor: usually configured so parent initially pause w hile child started to run; UNIX expression: "Children come first." How ever w ith modern multi-CPU systems, both tend to run simultaneously on different CPUs.

Often, rather than just fork, one follow w ith **exex**, w here parent process terminates and child process inherits process ID of parent. Term **fork and exec** used so often, sometimes though of as one w ord.

spaw n: program often used by older UNIX systems, similar in many w ays to **fork and exec**, differing in details. Not part of POSIX standard, not normal part of Linux.

To see how new processes may start:

Example 1: Web server that handles many clients

- May launch new process every time new connection is made w ith client
- May also simply start only new **thread** as part of same process

Linux -> not much difference on technical level betw een creating full process or just new thread, each mechanism takes about same time and roughly same amount of resources

Example 2: **sshd** daemon

- started w hen **init** process executes **sshd** init script
- Script responsible for launching **sshd daemon**
- Daemon process listens for **ssh** requests from remote users
- When request received, **sshd** *creates a new copy of itself* to service request
- Each remote user get ow n copy of **sshd** daemon running to service their remote login
- **sshd** process w ill start login program to validate remote user
- if authentication successful, login process w ill fork off shell (eg. **bash**) to interpret user commands, and so on

3.18 Using nice to Set Priorities

Process priority -> can be controlled through **nice** and **renice** commands. Since early days of UNIX, idea: *nice* process low ers priority to yield to others. Thus, higher **niceness**, low er priority.

Niceness values range from -20 (highest priority) to +19 (low est priority). Normal w ay to run **nice**:

```
$ nice -n 5 command [ARGS]
```

w hich w ould increase niceness by 5. Equivalent to doing:

```
$ nice -5 command [ARGS]
```

If no **nice** value given, default to increase niceness by 10. If no arguments given at all, reports current niceness:

```
$ nice
0

$ nice cat &
[1] 24908

$ ps -l
F S UID   PID  PPID C PRI NI ADDR SZ WCHAN  TTY       TIME CMD
0 S 500  4670  4603 0 80   0 - 16618 wait   pts/0 00:00:00 bash
0 S 500 24855  4670 0 80   0 - 16560 wait   pts/0 00:00:00 bash
0 T 500 24908 24855 0 90  10 - 14738 signal pts/0 00:00:00 cat
0 R 500 24909 24855 0 80   0 - 15887 -      pts/0 00:00:00 ps
```

Note: increasing niceness of process does not mean it w on't run. May even get all CPU time if nothing else w ith w hich to compete.

If large increment/decrement value outside -20 to 19 range supplied, increment value truncated.

3.19 Modifying the Nice Value

Default: only superuser can decrease niceness; i.e., increase priority. How ever, possible to give normal users ability to decrease niceness w ithin predetermined range by editing `/etc/security/limits.conf`.

To change niceness of already running process, easy to use **renice** command:

```
$ renice +3 13848
```

w hich w ill increase niceness by 3 of the process w ith `pid = 13848`. More than one process can be done at same time and also some other options, so see **man renice**.

3.21 Static and Shared Libraries

Programs built using **libraries** of code, developed for multiple purposes and used/reused in many contexts.

Tw o types of libraries:

- **Static**: code for library functions inserted in program at **compile time**. Does not change thereafter even if library updated
- **Shared**: code for library funcions loaded into program at **run time**. If library changed after, running program runs w ith new library modifications

Using shared libraries -> more efficient, because then can be used by many applications at once. Memory usage, executable sizes, application load time reduced.

Shared libraries also called **DLL** s (**D**ynamic **L**ink **L**ibraries).

3.22 Shared Library Versions

Shared libraries must be carefully versioned. If significant change to library and program not equipped to handle it, serious problems expected. Sometimes know n as **DLL Hell**.

Thus, programs can request specific **major** library version rather than latest one on system. Usually program w ill alw ays use latest **minor** version available.

Some application providers w ill use static libraries bundled into program to avoid problems. Dow n side: if improvements or bugs/security hole fixes in libraries, may not make it into applications in timely fashion.

Shared libraries file extension: **.so**. Typical full name: `libc.so.N`, w here `N` is major version number.

Under Linux, shared libraries carefully versioned:

```
c7:/usr/lib64>ls -lF libgdbm.so*
lrwxrwxrwx 1 root root 16 Apr 9 2015 libgdbm.so -> libgdbm.so.4.0.0*
lrwxrwxrwx 1 root root 16 Apr 9 2015 libgdbm.so.4 -> libgdbm.so.4.0.0*
-rwxr-xr-x 1 root root 36720 Jan 24 2014 libgdbm.so.4.0.0*
c7:/usr/lib64>
```

so program that just asks for `libgdbm` gets `libgdbm.so` and the others for specific major and minor versions.

3.23 Finding Shared Libraries

Program w hich uses shared libraries has to be able to find them at runtime.

ldd can be used to ascertain w hich libraries required by executable. Show s **soname** of library and w hat files it points to.

lddvi

ldconfig generally run at boot time (but can be run anytime). Uses file `/etc/ld.so.conf` w hich lists directories searched for shared libraries. Must run **ldconfig** as root and shared libraries should be stored in system directories only w hen stable and useful.

Besides searching data base built up by **ldconfig**, linker w ill first search any directories specified in environment variable

`LD_LIBRARY_PATH` (colon separated list of directories, as in `PATH` variable):

```
$ LD_LIBRARY_PATH=$HOME/foo/lib
$ foo [args]
```

or

```
$ LD_LIBRARY_PATH=$HOME/foo/lib foo [args]
```

##

Back to top

Previous Chapter - Table of Contents - Next Chapter