Advanced Unix Concepts

Satyajit Rai

March 17, 2003

Contents

- √ Advanced Unix Concepts 1
- √ Contents 2
- ⇒ Process Creation 4
- `fork()` 5
- What you get from your parent? 6
- What is your own? 7
- `exit()` 8
- Login Procedure 9

- Process Group 10
- Session 11
- Controlling Terminal 12
- Contents 14
- Job Control 16
- Job Control Primitives 17
- Shell Support for Job Control 19
- Orphaned Process group 20
- References 21

Process Creation

- Every process is created using `fork()` system call.
- Exceptions are:
  - `init:` system specific initialization – parent of all orphans.
  - `swapper:` scheduling.
  - `pagedeamon:` Virtual memory management.
- These processes are specifically created during bootstrapping.

fork()

- Called once, returns twice.
- Returns 0 in child, `pid` in parent.
- A processes can have only one parent, but many children.
- *data, heap, and stack* segments are copied for child.
- All file descriptors are duplicated.
- Copy on write (COW) – Linux.
- `vfork()` – when child called `exec()` or `_exit()`, parent resumes.

What you get from your parent?

- process credentials (real/effective/saved UIDs and GIDs)
- environment, stack, memory
- open file descriptors (note that the underlying file positions are shared between the parent and child, which can be confusing)
- close-on-exec flags, signal handling settings
- nice value, scheduler class
- process group ID, session ID
- current working directory, root directory, file mode creation mask (umask)
- resource limits, controlling terminal

What is your own?

- process ID, parent process ID
- copy of file descriptors and directory streams.
- process, text, data and other memory locks are NOT inherited.
- process times, in the tms struct
- resource utilizations are set to 0
- pending signals initialized to the empty set
- timers created by timer_create not inherited
- asynchronous input or output operations not inherited

exit()

- `return` from `main()`.
- calling `exit()` form a function.
- calling `_exit()` – does not clos and flush I/O buffers.
- Abnormal termination – `abort()`, or receipt of some *signal*.

Login Procedure

- `init` spawn one getty process per terminal.

- Normal Terminal

  `init` → `getty` → `login` → `shell` ↔ `terminal driver` ↔ `user`

- Network Logins

  `init` → `...` → `inted` → `telnetd` → `login` → `shell` ↔ `psuedo terminal driver` ↔ `...` ↔ `user`

Process Group

- `pid_t getgrp(void)` - get the group ID.
- `int setpgid(pid_t pid, pid_t pgid)` – set the Process group id. (also see `setsid()`)
- if `pid == pigid`, the process becomes the process group leader.
- A process can set the process group id of only itself, and its children.
- If a system does not support job control, returns error with `errno` set to ENOSYS.

Session

- Session is a collection of one or more process groups.
- `pid_t setsid(void)` – return process group ID if OK, -1 on error.
- The process becomes the session leader of a newly created session.
- The process becomes the process group leader of a new process group. The new process group id is the process id of the calling process.
- The association with the controlling terminal is broken (if any).

Controlling Terminal

- A session can have a single controlling terminal.
- Session leader that establishes connection to the controlling terminal is the controlling process.
- One of the process groups is *foreground*, while others are *background*.
- Terminal keys (Ctrl-C, Ctrl-Z, Ctrl-\) are sent to foreground process group.

- Modem disconnect – `SIGHUP` is sent to session leader
- default action of `SIGHUP` is to *kill* the processes.
- Setting and getting foreground process group

```
pid_t tcgetpgrp(int filedes)
int tcsetpgrp(int filedes, pid_t pgrid)
```

Contents

- √ Advanced Unix Concepts 1
- √ Contents 2
- √ Process Creation 4
- √ `fork()` 5
- √ What you get from your parent? 6
- √ What is your own? 7
- √ `exit()` 8
- √ Login Procedure 9

√ ● Process Group 10

√ ● Session 11

√ ● Controlling Terminal 12

√ ● Contents 14

⇒ ● Job Control 16

● Job Control Primitives 17

● Shell Support for Job Control 19

● Orphaned Process group 20

● References 21

Job Control

Job control is a feature that allows us to

- control multiple jobs from single terminal.
- control which jobs can access terminal, and which to run in background.

Job control requires support form

- Shell
- Terminal driver in Kernel
- Job control signals (`SIGCHLD, SIGCONT, SIGSTOP, SIGTSTP, SIGTTIN, and SIGTTOU`)

Job Control Primitives

- Ctrl-C (`SIGINT`) – Terminates the foreground process group.
- Ctrl-\ (`SIGQUIT`) – Terminates foreground process group with core.
- Ctrl-Z (`SIGTSTP`) – Suspend the foreground process group.
- To start a background process, use & at the end of command line. e.g.

  ```
  $ make all > make.out &        # make is executed in background
  ```

- `bg` – send process group to background.
- `fg` *n* – bring process group (job id *n* – assigned by shell) to foreground.
- Background jobs can not access the stdin – instead stopped.
- We can also disable access to stdout using `stty tostop` (see `man stty`).

Shell Support for Job Control

C-Shell offers job control, whereas Bourne shell doesn't offer it. Job control with Korn Shell is dependent on system.

For a shell without job control

- A process group will have same `PGID`, `SID`, and `TPGID`.

For a shell with job control

- `PGID` of the background process group is different than that of `TPGID`.
- `TPGID` of the foreground process group is same as that of `TPGID`.

Orphaned Process group

If a child lives longer than its parent,

- Child receives `SIGHUP` – default action is *kill*.
- Child becomes orphaned and gets shelter under `init`.
- `PPID` of the child process becomes 1.
- Child becomes member of orphaned process group.

References

- Advance Programming in Unix Environment – Richard Stevens, *Addision-Wesley*.
- Unix FAQs (http://www.faqs.org/faqs/unix-faq/programmer/faq/)

Coming Up...

Signals in Unix

The End