Chapter 8. Process Handling

BY HANSIKA BHAMBHANEY

OBJECTIVES:

1.Understand how to manage and control processes in Bash.

2.Learn how to use job control commands to handle foreground and background jobs.

3.Explore UNIX signals and how they interact with processes.

4.Use `trap` to catch and handle signals within scripts.

5.Understand subshells and how processes interact.

6.Implement coroutines and process substitution for parallelism.

8.1: Process IDs and Job Numbers

1. Every Command Runs as a Process

- When a command is executed in the shell, it starts a new process.
- Each process is assigned a unique **Process ID (PID)** by the system.

2. The Shell Assigns Job Numbers for Background Jobs

- When a command is executed in the background using &, the shell assigns it a **Job Number**.
- Job numbers are unique within the current terminal session and are used for job control.

3. Job Number and PID are Displayed Together

On running a background command, the shell shows:
EXAMPLE:

[1] 93

- [1] is the job number.
- 93 is the PID (process ID).

4. Background Jobs Are Managed by the Shell

Multiple background jobs can be run.

Example:

```
$ alice &
[1] 93

$ duchess &
[2] 102

$ hatter &
[3] 104
```

5. Job Completion or Failure is Shown by the Shell

On successful job completion:

[1]+ Done alice

On error:

[1]+ Exit 1 alice

6. Use `jobs` to View Background Jobs

- `jobs`: Lists background jobs.

```
$ jobs

[1]   Running   alice &

[2]-  Running   duchess &

[3]+  Running   hatter &
```

7. `jobs -l` Shows PIDs Alongside Job Numbers

Command:

```
$ jobs -l

[1]  93  Running   alice &

[2]- 102  Running   duchess &

[3]+ 104  Running   hatter &
```

8. `jobs -p` Lists Only PIDs

```
$ jobs -p

93

102

104
```

9. Job Control Options with `jobs`

- `-n`: Lists jobs whose status has changed.
- `-r`: Lists only running jobs.
- `-s`: Lists only stopped jobs.
- `-x`: Executes a command using a job's PID.

10. Referencing Jobs Using Special Syntax

REFERENCE	MEANING
%1	Job number 1
%duchess	Job whose command begins with 'duchess'
%?hat	Job whose command contains the string 'hat'
%% or %+	Most recent job
%-	Previous job before the most recent one

11. Jobs Can Be Controlled with `fg`, `bg`, `kill`

- `fg %1`: Brings job 1 to foreground.
- `bg %2`: Resumes job 2 in background.
- `kill %3`: Terminates job 3.

12. Job Numbers Are Simpler for Users than PIDs

- Shell users prefer job numbers for control over background jobs.
- PIDs are still required for system-level commands and scripting.

8.2 Job Control – Foreground and Background Jobs, Suspending and Resuming

1. Foreground and Background Jobs

- Normally, when you run a command, the shell runs it in the foreground — it takes control of your terminal until completion.
- If you want to run a long-running task and continue using the terminal, run the command in the background by appending an ampersand `&`.

EXAMPLE:

$ uncompress gcc.tar &

[1] 175

Here, `[1]` is the job number, and `175` is the PID of the background job

2. `jobs` Command to List Jobs

- The `jobs` command shows all background jobs and their status.

EXAMPLE:

$ jobs

[1]+ Running uncompress gcc.tar &

1. `jobs -l` lists jobs with PIDs.
2. `jobs -p` shows only PIDs.
3. `jobs -n`, `-r`, and `-s` filter by status (new, running, stopped)

3. Bringing Jobs to the Foreground (`fg`)

- Use `fg` to bring a background job to the foreground.

EXAMPLE:

$ fg

This brings the **most recently** backgrounded job to the foreground.

To bring a specific job:

$ fg %2

$ fg %duchess

1. `%+` or `%%`: most recent job

2. `%-`: previous job

4. Suspending a Foreground Job

- To **suspend** a job running in the foreground, press `CTRL+Z`.
- The shell will respond:

[1]+ Stopped command

Now the shell regains control of your terminal

5. Resuming in Background with `bg`

- After suspending a job with `CTRL+Z`, you can resume it in the background using:

EXAMPLE:
$ bg

This allows the job to continue running while freeing your terminal

6. Referencing Jobs

SYNTAX	DESCRIPTION
%1	Job number 1
%duchess	Job whose command starts with "duchess"
%?tea	Job whose command contains "tea"
%% or %+	Most recent background job
%-	Second most recent job

7.Example Session

```
$ alice &

[1] 93

$ duchess &

[2] 102

$ hatter &

[3] 104

$ jobs

[1]   Running alice &

[2]-  Running duchess &

[3]+  Running hatter &
```

To bring duchess to foreground:

$ fg %2

To suspend a running command:

CTRL+Z

To resume it in background:

$ bg

To resume in foreground:
$ fg

9. I/O Consideration in Background Jobs

- Background jobs shouldn't require terminal input.
- If they do, they may hang until brought to the foreground.
- To avoid mixed outputs, redirect background job output to a file:

Reference	Background job
%N	Job number *N*
%string	Job whose command begins with *string*
%?string	Job whose command contains *string*

7.3 SIGNALS

1.A signal is a short message sent to a process to notify it of an event.

2.Signals are used for interprocess communication (IPC).

3.They can be sent by the kernel, by another process, or via keyboard shortcuts.

4.Shell users mostly encounter signals when they stop or interrupt a job.

Common Control-Key Signals

CONTROL KEY	SIGNAL NAME	DESCRIPTION
CTRL+C	INT	Interrupts the current process
CTRL+Z	TSTP	Suspends the current process
CTRL+\	QUIT	Aborts and creates core dump

These are *keyboard signals* sent to foreground jobs by the shell

The `kill` Command

- Sends a signal to any process using its PID, job number, or command name.

- By default, `kill` sends the `TERM` (terminate) signal, similar to `CTRL+C`.

```bash
$ kill %1            # Sends TERM to job 1
$ kill -QUIT %1      # Sends QUIT to job 1
$ kill -KILL %1      # Forcefully kills job 1 (non-catchable)
```

Output examples:

- TERM: `[1]+ Terminated alice`
- QUIT: `[1]+ Exit 131 alice`
- KILL: `[1]+ Killed alice`

Signal Precedence in `kill` Usage

1. First try TERM — lets the process exit gracefully.

2. Then try QUIT — stronger than TERM.

3. As a last resort, use KILL — uncatchable, forces process termination.

The ps Command

- Displays current processes, used to get PID for `kill`.

```bash
$ ps
PID TTY     TIME COMD
146 pts/1   0:03 bash
2349 pts/1  0:03 alice
2390 pts/1  0:00 ps
```

1. `ps -a`: Shows all jobs on terminals.
2. `ps -e` or `ps -ax`: Shows all processes including daemons and zombies

Advanced: `killalljobs` Script Example

```bash
kill "$@" $(jobs -p)
```

Kills all background jobs using their process IDs

Signal Numbers vs. Names

- UNIX signals have both **numbers** and **names**.
- Use `kill -l` to list all signal names and their numbers.
- Use **names** in scripts for portability.

`stty` Command for Custom Control Keys

- Allows customization of control key bindings for signals:

```bash
$ stty intr ^X       # Sets interrupt signal to CTRL+X
```

Not recommended as it can confuse others working on your system

Summary of Signals Usage

- Signals are essential for managing, pausing, resuming, or killing processes.
- Use `CTRL+C`, `CTRL+Z`, `kill`, and `ps` as your main tools.
- Always prefer soft signals (`TERM`, `QUIT`) before using `KILL`.

8.4 trap Command — Signal Handling in Bash

```bash
trap 'commands' SIGNAL
```

- `commands`: The code to execute when the signal is received.
- `SIGNAL`: The signal name or number (e.g., `INT`, `EXIT`, `HUP`, `TERM`).

Example: Handling Interrupt Signal

```bash
trap 'echo "Interrupted!"; exit' INT
```

trap with `EXIT`

```bash
trap 'echo "Script finished."' EXIT
```

Executes the command when the script finishes or exits.

Use of `trap` in Functions

- Traps can be set inside functions to handle signals locally.

```bash
cleanup() {
  echo "Cleaning up..."
  rm -f /tmp/tempfile
}
trap cleanup EXIT
```

This ensures temporary files are removed when the script exits.

Ignoring Signals

```bash
trap '' INT
```

- This ignores the interrupt signal (`CTRL+C`). The script will not stop when the user tries to interrupt.

Resetting a Signal to Default

```bash
trap - INT
```

- Resets the trap for `INT` back to its default behavior (e.g., terminate on `CTRL+C`).

Process ID Variables and Temporary Files

Often combined with `trap` to manage temporary files using the current process ID (`$$`):

```bash
tmpfile="/tmp/mytemp_$$"
trap 'rm -f $tmpfile' EXIT
```

Ensures that `tmpfile` is deleted even if the script is interrupted or exits unexpectedly.

`disown` Command

- Used to remove a background job from the shell's job table.
- Prevents the shell from sending `HUP` (hangup) to it when you log out.

```bash
some_long_task &
disown
```

1. What Are Coroutines?

- In bash scripting, **coroutines** refer to processes that run in parallel but interact or synchronize with one another.

- Bash doesn't support true coroutines like some programming languages, but **background jobs combined with `wait`** and temporary files or pipes simulate coroutine-like behavior.
- STEPS:
  1. Create two subprocesses, which we'll call P1 and P2 (the fork system call).
- 2. Set up I/O between the processes so that P1's standard output feeds into P2's standard input (pipe).
- 3. Start /bin/ls in process P1 (exec).
- 4. Start /bin/more in process P2 (exec).
- 5. Wait for both processes to finish (wait).

🔹 **2. Using `wait` to Synchronize Jobs**

- The `wait` command pauses the execution of the script until **one or all** background jobs have completed.

- Useful when multiple parallel tasks must complete before the script continues.

Example:

```bash
sleep 5 &
sleep 3 &
wait
echo "Both background jobs completed."
```

- The script waits until both `sleep` commands finish, then prints the message.

3. `wait` with Specific PID

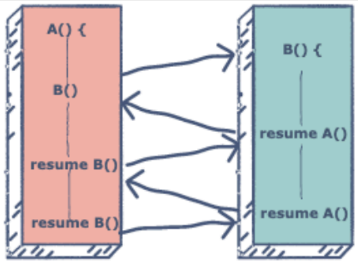
- You can `wait` on a specific process:

```
cmd1 &

pid1=$!

cmd2 &

pid2=$!


wait $pid1

echo "cmd1 finished"

wait $pid2

echo "cmd2 finished”
```

4. Coroutine Simulation Example

```bash
( echo "Task 1" ; sleep 2 ; echo "Task 1 done" ) > out1 &
( echo "Task 2" ; sleep 1 ; echo "Task 2 done" ) > out2 &

wait
cat out1 out2
```

- Both tasks run in parallel, and the output is collected after both finish.



When **A()** calls **B()**, the control is transferred to **B()** until it gives the control back to **A** using resume **A()** statement.
This way, both functions can coordinate with one another.

5. **Advantages of Coroutines in Bash**

- **Parallel Execution**: Run tasks simultaneously to save time.
- **Efficiency**: Ideal for I/O-bound operations or waiting tasks.
- **Control**: `wait` gives precise synchronization capability.

6. **Disadvantages of Coroutines in Bash**

- **Limited Communication**: Bash lacks native shared memory or message-passing between jobs.
- **Complexity**: Managing background jobs and output redirection can make scripts harder to maintain.
- **No True Coroutine Support**: Bash doesn't support yielding or resuming state like in coroutine-enabled languages.

🔹 **7. Use in Parallelization**

- Common in:
  - Parallel file downloads
  - Multiple system monitoring tasks
  - Concurrent logging or data processing

Example: Parallel File Checking

```
for file in *; do

  grep "ERROR" "$file" > "${file}.log" &

done



wait
```

```
echo "Error logs created for all files."
```

OUTPUT:

```
ansika_34@DESKTOP-TQ8I4PM:~$ bash CH8P2.sh
grep: 4: Is a directory
grep: CODES: Is a directory
grep: CHAPTER: Is a directory
grep: Homo_sapiens: Is a directory
Error logs created for all files.
```

8.SUMMARY TABLE:

FEATURE	DESCRIPTION
&	Starts a command in background
wait	Waits for background jobs
wait <PID>	Waits for specific job
Parallelization	Done using background jobs + wait
Advantage	Faster execution of independent tasks
Disadvantage	No built-in communication, increased complexity

8.6 Subshells in Bash

1. What is a Subshell?

- A subshell is a child process launched by the shell to execute a command or group of commands in isolation.
- Created using:
  - Parentheses: `( ... )`
  - Command substitution: `$( ... )`
  - Piping: `cmd1 | cmd2`

2. What is Inherited by Subshells

- Subshells inherit the environment from the parent shell:
    - Environment variables
    - Current directory
    - File descriptors
- They do not share variable changes back to the parent shell.

Example:

```bash
x=5
( x=10; echo "Inside subshell: $x" )
echo "Outside subshell: $x"
```

OUTPUT:

```
Inside subshell: 10
Outside subshell: 5
```

The variable x was modified in the subshell but not in the parent shell.

3. Subshells vs. Functions

FEATURE	SUBSHELL	FUNCTION
Scope of variables	Isolated	Shared with parent
Process created	New process	Same shell process
Performance	Slightly slower	Faster

4. Nested Subshells

- You can nest subshells using more parentheses
- Each nested subshell gets a **copy** of the environment from the one above it.

5. Practical Uses of Subshells

- Temporary variable changes.
- Isolated command execution.
- Avoid polluting the current shell environment.

Example: Changing directories temporarily

```bash
( cd /tmp && ls )
pwd  # Still shows original directory
```

6. Subshells and Piping

- Each command in a pipeline is run in a separate subshell:

```bash
x=10
echo "Hello" | read y
echo $y  # Will print nothing because read occurred in subshell
```

8.7 Process Substitution in Bash

1. What is Process Substitution?

- Process substitution allows the output or input of a command to be treated as if it were a file.
- It enables two commands that normally require files as input/output to **interact through streams**.

🔹 2. Syntax of Process Substitution

- Input-like substitution (output of command used as file input):

- <(command)
- Output-like substitution (input is redirected to command):
- >(command)

3. Example: Comparing Command Outputs

diff <(ls dir1) <(ls dir2)

1.`ls dir1` and `ls dir2` are run in parallel.

2.Their outputs are fed to `diff` as if they were two separate files.

4. How It Works Internally

- Bash creates **named pipes** or **temporary files** that link the file descriptor of a command's output/input.

- The command being passed via `<( )` or `>( )` runs in a **subshell**.

5. Use Case: Feeding Output to Multiple Commands

tee >(grep ERROR > error.log) >(grep WARN > warn.log) < logfile.txt

- `tee` reads `logfile.txt` and splits it:
  - One stream to grep errors
  - Another to grep warnings
- Both outputs are redirected and saved in separate files.

6. Important Considerations

- Not available in POSIX sh — it's **Bash-specific**.

- Requires commands to be able to read from **named pipes or FIFOs**.

- May not work on platforms like macOS without `/dev/fd`.

7. Summary Table

Syntax	Purpose
`<(command)`	Use command's output as a file
`>(command)`	Send data to command's input as a file
`diff <(a) <(b)`	Compare outputs of two commands
`tee >(a) >(b)`	Split input and redirect to multiple cmds

SUMMARY

Chapter 8 explains process handling in Bash. Every command runs as a process with a unique PID, and background jobs are managed using job numbers. Commands like `jobs`, `fg`, `bg`, and `kill` help control these jobs. Signals such as `CTRL+C` (INT) and `CTRL+Z` (TSTP) can interrupt or pause processes, and the `trap` command allows scripts to handle signals gracefully. `disown` detaches jobs from the terminal. Coroutines simulate parallel execution using background jobs and `wait`. Subshells provide isolated environments that don't affect parent variables. Process substitution using `<( )` and `>( )` enables efficient data flow between commands.