***Chapter 8. Process Handling***

***Introduction:-***

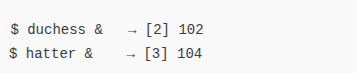
1. UNIX Philosophy:
   * Built on simple but powerful concepts (e.g., pipes, standard I/O, text tools, file system).
2. Multitasking Innovation:
   * UNIX was one of the first OSes to allow users to control multiple processes at once.
3. User-Controlled Multitasking:
   * Users can run background processes (e.g., using &) and manage them actively.
4. Subshells:
   * Shell scripts and some command groups run in subshells, which are separate process environments.
5. Chapter Focus:
   * Covers most of Bash’s multitasking and process-handling features.
   * Skips deep system programming details unless necessary.
6. Practical Orientation:
   * Focus on usable, high-level features; minimal low-level technicalities.
   * Recommends external resources for deeper UNIX internals.
7. Hands-On Emphasis:
   * Encourages trying out examples—process behavior can be hard to grasp theoretically.

Process IDs and Job Numbers:-

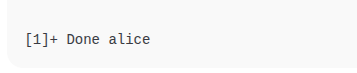
1. **Process ID (PID)**:
   * A **unique number** assigned by the operating system to each running process.
   * Applies to **all processes** on the system, across all users.
2. **Job Number**:
   * Assigned by the **shell**, not the OS.
   * Refers to **background jobs** started from *your shell session only*.
   * Easier to remember than PIDs (e.g., [1], [2], etc.).
3. **Example of a Background Job**:



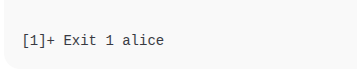
**4. Multiple Background Jobs**: Shell assigns job numbers incrementally:



**5.Job Completion Messages**: When a background job finishes:



If it exits with an error (non-zero status):



**6.+ Sign Meaning:**

Indicates the most recently stopped or finished job (explained later in chapter).

**7. Output Timing:**

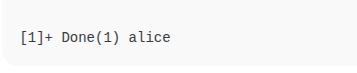
Completion messages appear before the next prompt, to avoid interrupting your command output.

**8. Optional Behavior (set -b):**

Use set -b to immediately show job status messages as soon as jobs end.

**9. POSIX Mode Difference:**

In POSIX mode, messages may look like:



### **Job Control in Bash:-**

Why Job Numbers Matter:

* Process IDs are system-wide and not often needed by regular users.
* Job numbers are more relevant in shell sessions for controlling background jobs.

Job Control Commands:

* &: Starts a job in the background.
* fg: Brings a background job to the foreground (so it can receive input).
* jobs: Lists all current background jobs.

Using fg:

* Without arguments: brings the most recent background job to the foreground.
* With job reference:
  + By job number: fg %2
  + By command name: fg %duchess
  + By process ID: fg 104

Job Reference Symbols:

* %N: Job number (e.g., %1)
* %string: Job whose command starts with string
* %?string: Job whose command contains string
* %+ or %%: Most recently backgrounded job
* %-: Second most recent

jobs Command Options:

* jobs -l: Lists job numbers with process IDs
* jobs -p: Lists only PIDs
* jobs -n: Lists jobs whose status has changed since last shown
* jobs -r: Lists only running jobs
* jobs -s: Lists only stopped jobs
* jobs -x cmd %job: Runs cmd with job’s PID substituted

**Example**:

$ hatter mad &[1] 189

$ hatter teatime &[2] 190

1. **Command Name Collisions**:
   * If multiple jobs use the same command, use:
     + Job number (%1)
     + Partial match (%?arg), if arguments differ
2. **Plus/Minus Signs (+, -) in jobs Output**:
   * +: Most recent job (default target for fg)
   * -: Second most recent job

Suspending Jobs and signals:-

1. **CTRL-Z** suspends a foreground job and returns control to the shell.
   * The job is **stopped**, not terminated.
   * You’ll see: [1]+ Stopped command
2. **fg** resumes the suspended job in the **foreground**.
   * If you have multiple jobs, specify the job: fg %1 or fg %command.
3. **bg** resumes a suspended job in the **background**.
   * Use after CTRL-Z to continue a task without blocking your terminal.
4. **Use Case Example**:
   * Suspend vi with CTRL-Z, run another command (e.g., troff), then bring vi back with fg.
5. **CTRL-Y** also suspends, but only stops the process when it tries to read terminal input (less common than CTRL-Z).
6. **Not all commands behave well** when suspended and resumed, especially **network-based** or **interactive** ones.

🔹 Signals

1. A **signal** is a message sent to a process to trigger a specific behavior (e.g., stop, kill, continue).
   * Part of **interprocess communication (IPC)**.
2. **Common Signals via Keyboard**:
   * CTRL-C → **SIGINT**: Interrupt (terminate)
   * CTRL-Z → **SIGTSTP**: Suspend (stop)
   * CTRL-\ → **SIGQUIT**: Quit with core dump (stronger than INT)
3. Use kill -l to list signal names and numbers available on your system.
4. Signal names are more **portable** than numbers between different UNIX systems.

### **🔹 Changing Control Keys**

1. You can customize control keys using stty. Example:

stty intr ^X

* Changes the interrupt key to CTRL-X.
* But: **not recommended**—can confuse users/admins.

### **🔹 Other Notes**

1. Older UNIX systems (e.g., System III, early System V, Xenix) may **not support** job control features.
2. More signals exist for low-level errors (e.g., segmentation fault, floating-point error) used by the OS and systems programmers.

### **Basics of kill**

1. **kill** is a built-in shell command used to **send signals to processes**.
2. You can target:
   * A **process ID** (PID)
   * A **job number** (with %)
   * A **command name** (less common and ambiguous)
3. **Default signal**: TERM (terminate), similar to CTRL-C.
4. Syntax examples:  
   * kill %1 – kills job number 1
   * kill 150 – kills process with PID 150
   * kill -SIGNAL %1 – sends a specific signal (e.g., QUIT, KILL)
5. If TERM doesn't stop the process:
   * Try kill -QUIT %1
   * If that fails, use the **last resort**: kill -KILL %1

### **🔹 Signal Behavior**

1. Most signals **terminate** the process.
2. But some programs handle signals differently—for example:  
   * A text editor may **save your work** on TERM or INT.
3. **KILL signal**:
   * **Cannot be caught, ignored, or blocked**
   * Terminates immediately and unconditionally
   * Use **only when necessary**, since it prevents cleanup

### **🔹 Script Example: killalljobs**

1. Kill all background jobs with:



Purpose of ps

1. ps displays **information about running processes**, including their **process IDs (PIDs)**.
2. It’s **essential** when you need to find and **control a process**, especially for commands like kill.

🔹 Basic Usage

1. Running ps **without arguments** shows:
   * The **current shell**
   * Any **background jobs** from that shell
   * The ps command itself
2. Common output columns:
   * **PID** – Process ID
   * **TTY / TT** – Terminal associated with the process
   * **TIME** – CPU time used (not real time)
   * **COMMAND / COMD** – The command that started the process
   * (On BSD systems, also shows **arguments**)

🔹 System Differences

1. ps syntax and output **vary between UNIX versions**:
   * **System V** vs. **BSD-based** (e.g., macOS, SunOS)
   * Always check your system's manual (man ps) for details

🔹 Using ps -a

1. ps -a shows **all processes associated with any terminal**, not just your current shell.
2. On **System V** systems:
   * It omits **parent shells (group leaders)**
   * Useful when you have **multiple terminal windows open**
   * Helps identify **processes running in other windows**

🔹 Example Scenario

1. With three terminal windows and background jobs in each:
   * ps in window 1 shows only jobs in that window
   * ps -a shows **all background jobs** across windows
   * Use this info to **locate and kill a runaway process**

🔹 Limitations

1. ps does **not correlate PIDs with shell job numbers** (%1, %2, etc.).
2. It **cannot see jobs** from another shell using job notation—use **PIDs instead**.

***ps on BSD Systems:-***

1. ps -a on BSD systems:
   * Lists all jobs started on any terminal, including group leaders (parent shells).
   * Acts like a combined output of ps for every user session.
2. Limitation:
   * Zombie or orphan processes (runaway processes with lost terminal/session association) do not appear in ps -a.

🔹 Detecting Hidden Processes

1. Use ps -ax (BSD) or ps -e (System V) to list all processes, including:
   * Daemon processes (background services)
   * Processes not associated with terminals
   * Runaway/zombie/orphan processes
2. Look for processes with ? in the TTY/TT column—these likely include:
   * System daemons (normal)
   * Processes in trouble (potential runaways)
3. Examine the COMMAND column to identify suspicious or runaway jobs.

🔹 ***Signals and Core Dumps:-***

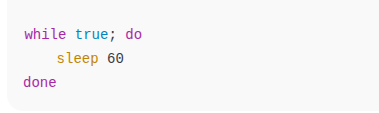
1. Some signals (like CTRL-\ or QUIT) cause a core dump file (core) in the current directory.
   * This file is used for debugging.
   * It can be safely deleted unless needed.
2. When a shell script receives a signal, it exits with code 128 + signal number (e.g., QUIT is 3 → exit code 131).

🔹 ***trap Command:-***

1. trap allows shell scripts to handle signals gracefully.
   * Syntax: trap "command" SIGNAL
   * When SIGNAL is received, the command runs instead of immediate termination.
2. Useful for:
   * Handling CTRL-C (INT), kill (TERM), etc.
   * Cleaning up or notifying the user before exiting
3. Example:-

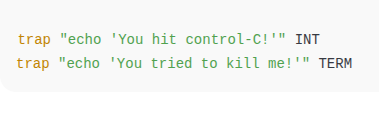


**Using trap with Scripts:-**



11.Adding a trap lets the script **respond** to signals (e.g., print a message instead of stopping).

12. Different traps for different signals

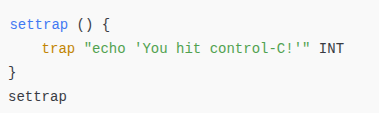


### **Traps and Functions**

1. **Traps are inherited by functions**:

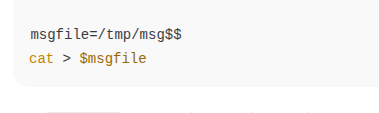
* A function can **override a trap** set in the parent shell.
* The **last trap defined** takes effect globally after the function runs.

1. Example:



16. If a function redefines a trap, it **overrides** the earlier one—even **after** it finishes.

17. Compose a message using cat and a temporary file:



* $msgfile is uniquely named using the script’s process ID ($$).
* User types the message and ends it with CTRL-D (end-of-text).

### **Process ID Variables & Temporary Files**

* **$$**: Represents the **process ID** (PID) of the current shell. Useful for generating **unique temp filenames**.
* **$!**: Holds the PID of the **most recently run background job**.
* **/tmp** and **/var/tmp**: Standard directories for temporary files.
* **Temp files** should be **cleaned up** to avoid disk clutter.

### **🔹 Signal Handling with trap**

* Use the trap command to specify actions on receiving **signals** like INT (CTRL-C) and TERM.

**Example:**  
trap 'mv $msgfile dead.letter; exit' INT TERM

* Saves unfinished mail to dead.letter if script is interrupted.

### **🔹 Using Functions for Cleanup**

Better practice: use a **function** to handle cleanup, especially when checking for file existence.  
function cleanup {

if [ -e $msgfile ]; then

mv $msgfile dead.letter

fi

exit

}

trap cleanup INT TERM

### **🔹 Ignoring Signals**

* trap "" SIGNAL tells the shell to **ignore** a signal (e.g., HUP).
* Useful to **prevent hangups** from killing background jobs.

### **🔹 nohup Command**

* nohup prevents a process from being terminated by a hangup (HUP) signal.

Redirects output by default to nohup.out if not specified.  
nohup ./script.sh > logfile 2>&1 &

### **🔹 disown Built-in**

* disown: Removes a job from the shell’s job table.
* disown -h: Prevents HUP signal from being sent to the process.
* disown -a: Applies to all jobs; -r: applies only to running jobs.

### **🔹 Resetting Traps**

Use trap - SIGNAL to **reset signal handling** to default.  
 trap - INT TERM

### **🔹 Best Practices**

* Use signal traps **only when needed**, especially in larger scripts.
* Avoid overengineering small scripts with full signal-handling logic.
* Focus on handling only cases where signal interruption would cause **serious issues** (e.g., data loss, resource leaks).

### **Coroutines**

* **Definition**: Coroutines are multiple processes programmed to run simultaneously and optionally communicate.
* **Pipeline as Coroutine**: A pipeline (e.g., ls | more) is a built-in coroutine structure using system calls:
  + fork → create subprocesses
  + pipe → connect outputs/inputs
  + exec → replace process with new command
  + wait → wait for process completion

### **Running Background Jobs (Without Communication):-**

Syntax:  
alice &

hatter

wait

* The wait command ensures the script waits for all background jobs to finish before exiting.
* You can also wait [PID] for a specific background job.

### **Advantages of Coroutines:-**

* **Improved performance** even on single CPU if jobs use different resources:
  + CPU-intensive vs I/O-intensive processes.
* **Parallel execution** minimizes total run time.

### **Disadvantages of Coroutines:-**

* **Thrashing** occurs if multiple processes contend for the same resource (e.g., disk).
* **Less efficiency** if jobs are similar in resource usage.
* **Concurrency control issues** when multiple processes access the same resource.

### **Parallelization:-**

* **Useful in multi-CPU systems** — true simultaneous execution.
* **Speedup =** runtime of longest process + minor overhead.

Script Example (parallel copy):  
file=$1

shift

for dest in "$@"; do

cp $file $dest &

done

wait

* **Problem**: Duplicate destinations can cause file corruption → needs duplication checks.

### **Concurrency Control:-**

* Even trivial parallelization (like file copy) introduces risks (e.g., race conditions).
* Writing safeguards (like duplicate detection) can offset performance gains.

### **Subshells:-**

* A **subshell** is a separate shell process spawned by the parent shell.
* **Inherits**:
  + Current directory
  + Environment variables
  + Standard input/output/error
  + Ignored signals
* **Does NOT inherit**:
  + Shell variables (unless exported)
  + Custom signal handlers

#### **Nested Subshells:**

Created using parentheses ( ), e.g.:  
( var=value; command )

* **Variables/traps inside subshell** are not accessible outside.
* Compare with **command blocks** { }, which **share scope**.

### **🧪 Example: Subshell vs Command Block**

#### **Command block:**

{

hatter=mad

trap "echo 'CTRL-C!'" INT

}

# hatter and trap are accessible here

#### **Subshell:-**

(

hatter=mad

trap "echo 'CTRL-C!'" INT

)

# hatter and trap are NOT accessible here

### **🔄 Process Substitution**

* Allows using process output as if it were a file.
* Two forms:
  + <(command) → command output as file input
  + >(command) → file input goes to command

#### **Example:**

#### cmp <(prog1) <(prog2)

* Both prog1 and prog2 run concurrently; outputs compared using cmp.