## Table of Contents

# $ bg

Intermediate

Resume a suspended job in the background

The bg command resumes a suspended job in the background. When you press Ctrl+Z, the foreground process is suspended (stopped). bg then restarts it as a background job, freeing your terminal for other commands.

bg is a shell built-in used for job control. It works with job numbers from the jobs ...

## Options & Flags

| | |
|---|---|
| %N | Resume job number N in background |
| %% | Resume current (most recent) job |
| %- | Resume previous job |

## Practical Examples

**Example: Resume last suspended job**

```
$ bg
[1]+ make -j4 &
```

Resumes the most recently suspended (Ctrl+Z) job in the background.

**Example: Resume specific job**

```
$ bg %2
```

Resumes job number 2 in the background.

**Example: Typical workflow**

```
$ make -j4 # Press Ctrl+Z bg # Now make runs in background
```

Suspend a long-running build and continue it in the background.

**Example: Multiple background jobs**

```
$ bg %1 bg %2
```

Resumes multiple suspended jobs in the background.

## Tips & Best Practices

**Pro Tip:** Ctrl+Z then bg workflow: Started a long command? Press Ctrl+Z to suspend, then bg to run in background. Your terminal is free again.

**Note:** bg vs &: command & starts a new job in background. bg resumes a suspended job in background. Both result in background execution.

**Warning:** Output goes to terminal: Background jobs still print to your terminal. Redirect output before backgrounding: command > log 2>&1 then Ctrl+Z, bg.

# $ chrt

Advanced

Set or get real-time scheduling attributes

The chrt command sets and retrieves the real-time scheduling policy and priority of processes. Linux supports several CPU scheduling policies: SCHED_OTHER (default, time-sharing), SCHED_FIFO (real-time, first-in-first-out), SCHED_RR (real-time, round-robin), SCHED_BATCH (batch processing), SCHED_...

## Options & Flags

| | |
|---|---|
| `-p PID` | Get or set scheduling of running process |
| `-f` | Set SCHED_FIFO policy (real-time) |
| `-r` | Set SCHED_RR policy (real-time, round-robin) |
| `-o` | Set SCHED_OTHER policy (normal/default) |
| `-b` | Set SCHED_BATCH policy (batch processing) |
| `-i` | Set SCHED_IDLE policy (lowest priority) |
| `-d` | Set SCHED_DEADLINE policy |
| `-m` | Show minimum and maximum priority for each policy |
| `-v` | Verbose output |

## Practical Examples

**Example: Check process scheduling**

```
$ chrt -p $(pgrep -f postgres)
pid 1234's current scheduling policy: SCHED_OTHER
pid 1234's current scheduling priority: 0
```

Show the current scheduling policy and priority of a running process.

**Example: Run with real-time priority**

```
$ sudo chrt -f 50 ./audio-processor
```

Start process with FIFO real-time priority 50 (range 1-99). Higher priority = runs first.

**Example: Set running process to real-time**

```
$ sudo chrt -r -p 30 $(pgrep -f myapp)
```

Change a running process to SCHED_RR with priority 30. No restart needed.

**Example: Run batch job at lowest priority**

```
$ chrt -b 0 make -j$(nproc)
```

Run compilation as batch job. Gets CPU time only when no interactive processes need it.

**Example: Background task with SCHED_IDLE**

```
$ chrt -i 0 ./backup-script.sh
```

Run at absolutely lowest priority. Only gets CPU time when the system is completely idle.

## Tips & Best Practices

**Warning:** Real-time can hang system: A SCHED_FIFO process with high priority that never yields CPU can make the system unresponsive. Always test with lower priorities first. Use SCHED_RR for time-sliced real-time.

**Pro Tip:** Priority 99 is reserved for kernel: Avoid priority 99 - it is used by critical kernel threads (watchdog, migration). Use 1-98 for user processes. Priority 50 is a good starting point.

**Note:** Requires root for real-time: Setting SCHED_FIFO or SCHED_RR requires root (or CAP_SYS_NICE capability). SCHED_BATCH and SCHED_IDLE can be set by regular users for their own processes.

**Pro Tip:** Use with taskset for CPU pinning: Combine chrt (scheduling) with taskset (CPU affinity) for maximum control: sudo chrt -f 50 taskset -c 0-3 ./myapp

# $ fg

Intermediate

Bring a background job to the foreground

The fg command brings a background or suspended job to the foreground, making it the active process in your terminal. It is the complement to bg - fg brings jobs forward, bg sends them back.

fg is a shell built-in used for job control. Without arguments, it brings the most recent background/susp...

## Options & Flags

| | |
|---|---|
| `%N` | Bring job number N to foreground |
| `%%` | Bring current (most recent) job to foreground |
| `%-` | Bring previous job to foreground |
| `%string` | Bring job whose command starts with string |

## Practical Examples

**Example: Bring last job to foreground**

```
$ fg
```

Resumes the most recent background/suspended job in the foreground.

**Example: Bring specific job**

```
$ fg %2
```

Brings job number 2 to the foreground.

**Example: Bring job by name**

```
$ fg %vim
```

Brings the job whose command starts with "vim" to the foreground.

**Example: Typical workflow**

```
$ vim file.txt # Press Ctrl+Z to suspend ls -la # Check files fg # Back to vim
```

Suspend editor, run a quick command, return to editor.

## Tips & Best Practices

**Pro Tip:** Quick task switching: Use Ctrl+Z and fg to quickly switch between editing and shell commands without opening another terminal.

**Note:** fg vs bg: fg makes the job active in your terminal (you interact with it). bg runs it in background (no interaction, terminal is free).

**Warning:** One foreground process: Only one process can be in the foreground at a time. The previous foreground process will be suspended.

# $ htop

Beginner

Interactive process viewer with color and mouse support

htop is an interactive process viewer with an improved, color-coded interface compared to top. It provides real-time monitoring of CPU, memory, and swap usage with visual bar graphs and supports mouse interaction.

htop shows individual CPU cores, memory and swap bars, load average, uptime, and a...

## Options & Flags

| | |
|---|---|
| `-d` | Set update delay in tenths of seconds |
| `-u` | Show only processes of a user |
| `-p` | Monitor specific PIDs |
| `-t` | Start in tree view mode |
| `-s` | Sort by column |
| `-C` | Monochrome mode (no colors) |
| `--no-mouse` | Disable mouse support |

## Practical Examples

**Example: Standard monitoring**

```
$ htop
```

Opens the interactive process viewer with color-coded bars and process list.

**Example: Tree view**

```
$ htop -t
```

Shows processes in a tree hierarchy showing parent-child relationships.

**Example: Monitor specific user**

```
$ htop -u nginx
```

Filters to show only processes owned by the nginx user.

**Example: Sort by memory**

```
$ htop -s PERCENT_MEM
```

Starts sorted by memory usage percentage.

**Example: Fast refresh**

```
$ htop -d 5
```

Updates every 0.5 seconds (5 tenths) for high-resolution monitoring.

## Tips & Best Practices

**Pro Tip:** Key shortcuts: F2=Setup, F3=Search, F4=Filter, F5=Tree, F6=Sort, F9=Kill, F10=Quit. Also: / to search, \ to filter, Space to tag.

**Note:** CPU bar colors: Blue=low priority, Green=normal, Red=kernel, Cyan=virtualization. Understanding colors helps identify what is consuming CPU.

**Warning:** Not always pre-installed: htop is not installed by default on many systems. Install with: apt install htop (Debian/Ubuntu) or yum install htop (RHEL).

# $ jobs

Intermediate

Display status of background jobs in current shell

The jobs command displays the status of background and suspended jobs in the current shell session. Each job has a job number (different from PID) that can be used with fg, bg, kill, and other job control commands.

jobs shows whether each background task is running, stopped, or completed. It is ...

## Options & Flags

| | |
|---|---|
| `-l` | List jobs with PIDs |
| `-r` | Show only running jobs |
| `-s` | Show only stopped jobs |
| `-p` | Show only PIDs |

## Practical Examples

**Example: List all jobs**

```
$ jobs
[1]+  Running   ./backup.sh &\n[2]-  Stopped   vim config.yml
```

Shows all background and suspended jobs in the current shell.

**Example: List with PIDs**

```
$ jobs -l
[1]+ 12345 Running   ./backup.sh &
```

Shows job numbers along with process IDs.

**Example: Background a task**

```
$ sleep 300 &\njobs
[1]+ Running   sleep 300 &
```

Starts a background job and lists it.

**Example: Resume stopped job**

```
$ fg %1
```

Brings job 1 back to foreground.

**Example: Show only running jobs**

```
$ jobs -r
```

Lists only jobs that are currently running.

## Tips & Best Practices

**Pro Tip:** Job control shortcuts: Ctrl+Z suspends foreground process. bg resumes it in background. fg brings it to foreground. & at end of command starts in background.

**Note:** Job numbers vs PIDs: %1 refers to job 1, $! refers to the last background PID. Use jobs -l to see both job numbers and PIDs.

**Warning:** Jobs are per-shell: jobs only shows jobs from the current shell session. Background jobs started in other terminals are not visible.

# $ kill

Beginner

Send signals to processes (terminate, stop, continue)

The kill command sends signals to processes, most commonly used to terminate them. Despite its name, kill can send any signal, not just termination signals. Signals are the primary inter-process communication mechanism in Unix/Linux.

The most important signals are SIGTERM (15, graceful shutdown)...

## Options & Flags

| | |
|---|---|
| `-15 / -TERM` | Send SIGTERM - graceful termination (default) |
| `-9 / -KILL` | Send SIGKILL - force kill (cannot be caught) |
| `-1 / -HUP` | Send SIGHUP - reload configuration |
| `-STOP` | Pause a process |
| `-CONT` | Resume a paused process |
| `-l` | List all signal names |
| `-0` | Check if process exists (no signal sent) |

## Practical Examples

**Example: Graceful termination**

```
$ kill 1234
```

Sends SIGTERM - the process can catch this and clean up before exiting.

**Example: Force kill**

```
$ kill -9 1234
```

Sends SIGKILL - immediately terminates the process. Use only when SIGTERM fails.

**Example: Reload configuration**

```
$ kill -HUP $(cat /var/run/nginx.pid)
```

Sends SIGHUP to nginx - causes it to reload its configuration without restarting.

**Example: Kill multiple processes**

```
$ kill 1234 5678 9012
```

Sends SIGTERM to multiple processes at once.

**Example: Kill all user processes**

```
$ kill $(ps -u baduser -o pid=)
```

Terminates all processes owned by a specific user.

## Tips & Best Practices

**Warning:** Try SIGTERM before SIGKILL: Always try kill PID (SIGTERM) first. Only use kill -9 if the process does not respond. SIGKILL prevents cleanup and can leave corrupted data.

**Pro Tip:** Kill by name: Use killall processname or pkill processname instead of finding the PID first. More convenient for most cases.

**Note:** Signal 0 for checking: kill -0 PID does not send a signal - it just checks if the process exists and you have permission to signal it. Useful in scripts.

# $ killall

Beginner

Kill processes by name

killall sends a signal to all processes with a given name. Unlike kill which requires a PID, killall targets processes by their command name, making it convenient for stopping all instances of a program.

killall matches the exact process name (the basename of the executable). It supports case-in...

## Options & Flags

| | |
|---|---|
| `-9` | Send SIGKILL (force kill) |
| `-i` | Ask for confirmation before each kill |
| `-I` | Case-insensitive name matching |
| `-u` | Kill only processes owned by user |
| `-v` | Verbose - report if signal was sent |
| `-w` | Wait for all killed processes to die |
| `-r` | Use regex for process name matching |
| `-o` | Kill only processes older than specified time |

## Practical Examples

**Example: Kill all instances by name**

```
$ killall nginx
```

Sends SIGTERM to all nginx processes.

**Example: Force kill all instances**

```
$ killall -9 firefox
```

Force kills all firefox processes immediately.

**Example: Kill with confirmation**

```
$ killall -i python3
```

Asks yes/no before killing each matching process.

**Example: Kill user processes**

```
$ killall -u john
```

Terminates all processes owned by user john.

**Example: Kill and wait**

```
$ killall -w myservice
```

Sends SIGTERM and waits until all matching processes have terminated.

## Tips & Best Practices

**Warning:** Exact name matching: killall matches the exact process name, not command arguments. killall python3 will not match /usr/bin/python3 script.py on some systems. Use pkill for broader matching.

**Pro Tip:** Use -w for scripts: In scripts, use killall -w to block until processes are dead before starting new ones. Prevents race conditions in restart scripts.

**Note:** Linux vs Solaris: On Linux, killall kills matching processes. On Solaris, killall kills ALL processes! Always verify behavior on non-Linux systems.

# $ nice

Intermediate

Run a command with modified scheduling priority

nice runs a command with a modified scheduling priority. The niceness value ranges from -20 (highest priority) to 19 (lowest priority). The default niceness is 0.

Higher niceness means the process is "nicer" to other processes - it yields CPU time. Lower (negative) niceness means the process tak...

## Options & Flags

| | |
|---|---|
| `-n` | Set niceness adjustment value |
| `default` | Without -n, uses niceness 10 |
| `negative` | Higher priority (root only) |
| `range` | Values from -20 to 19 |

## Practical Examples

**Example: Run with low priority**

```
$ nice -n 15 make -j$(nproc)
```

Compiles code using all cores but at low priority so the system stays responsive.

**Example: Default nice**

```
$ nice tar czf backup.tar.gz /home/
```

Runs with niceness 10 (default nice increment).

**Example: Lowest priority**

```
$ nice -n 19 find / -name "*.log" -mtime +30 -delete
```

Runs cleanup at the lowest possible priority.

**Example: High priority (root)**

```
$ sudo nice -n -10 critical-service
```

Starts a critical service with higher priority than normal processes.

**Example: Background job with nice**

```
$ nice -n 19 rsync -av /data/ /backup/ &
```

Runs a backup sync in the background at lowest priority.

## Tips & Best Practices

**Pro Tip:** Always nice background jobs: Use nice -n 19 for backups, builds, and batch jobs. This prevents them from making the system sluggish for interactive users.

**Note:** renice for running processes: nice sets priority at launch. To change priority of an already running process, use renice: renice -n 10 -p PID.

**Warning:** Negative values need root: Only root can set negative niceness (higher priority). Regular users can only increase niceness (lower priority).

# $ nohup

Intermediate

Run a command immune to hangups (keeps running after logout)

nohup (no hangup) runs a command immune to hangup signals, allowing it to continue running after you disconnect from a terminal session. When you close a terminal or SSH session, SIGHUP is sent to all processes - nohup prevents this from killing your process.

nohup redirects stdout to nohup.out ...

## Options & Flags

| | |
|---|---|
| `basic` | Run command immune to hangup |
| `redirect` | Redirect output to specific file |
| `disown` | Remove from shell job table |

## Practical Examples

**Example: Run in background after logout**

```
$ nohup ./backup.sh &
nohup: appending output to nohup.out
```

Runs backup script that survives terminal/SSH disconnection. Output goes to nohup.out.

**Example: Custom output file**

```
$ nohup python3 train_model.py > training.log 2>&1 &
```

Runs a long ML training job with output redirected to a specific log file.

**Example: Run and disown**

```
$ nohup rsync -av /data/ /backup/ > rsync.log 2>&1 & disown
```

Runs rsync completely detached from the terminal.

**Example: Check nohup output**

```
$ tail -f nohup.out
```

Monitors the output of a nohup process in real-time.

**Example: Multiple commands**

```
$ nohup bash -c "cd /app && make build && make deploy" > deploy.log 2>&1 &
```

Runs multiple commands as a single background task.

## Tips & Best Practices

**Pro Tip:** Always redirect stderr: Use 2>&1 to capture both stdout and stderr. Without it, error messages may be lost: nohup cmd > logfile 2>&1 &

**Note:** screen/tmux is better: For interactive sessions you want to reconnect to, use screen or tmux. nohup is for fire-and-forget tasks.

**Warning:** nohup.out grows forever: Without explicit redirection, nohup appends to nohup.out in the current directory. This file can grow very large. Always redirect to a specific log file.

# $ pgrep

Intermediate

Look up processes based on name and attributes

pgrep searches for processes by name or other criteria and outputs their PIDs. It is the lookup companion to pkill - pgrep finds processes, pkill kills them, using identical matching syntax.

pgrep replaces the common ps aux | grep pattern | grep -v grep idiom with a cleaner, more reliable comman...

## Options & Flags

| | |
|---|---|
| `-a` | Show PID and full command line |
| `-f` | Match against full command line |
| `-u` | Match by effective user |
| `-c` | Count matching processes |
| `-l` | Show PID and process name |
| `-x` | Exact match only |
| `-n` | Show only newest match |
| `-o` | Show only oldest match |
| `-d` | Set output delimiter |

## Practical Examples

**Example: Find process PIDs**

```
$ pgrep nginx
1234
1235
1236
```

Lists PIDs of all nginx processes.

**Example: Show PIDs with command lines**

```
$ pgrep -a python
1234 python3 manage.py runserver
5678 python3 celery worker
```

Shows PIDs and full command lines for all Python processes.

**Example: Count instances**

```
$ pgrep -c php-fpm
8
```

Returns the count of running PHP-FPM processes.

**Example: Check if running (scripting)**

```
$ pgrep -x nginx > /dev/null && echo "Running" || echo "Stopped"
Running
```

Checks if nginx is running for monitoring scripts.

**Example: Match full command line**

```
$ pgrep -af 'node.*server.js'
```

Finds processes with "node" and "server.js" anywhere in the command line.

## Tips & Best Practices

**Pro Tip:** Better than ps | grep: pgrep -a nginx is cleaner and more reliable than ps aux | grep nginx | grep -v grep. It doesn't match itself and supports exact matching.

**Note:** Exit codes: pgrep returns 0 if at least one match, 1 if no matches. Use in scripts: if pgrep -x nginx > /dev/null; then ...

**Warning:** Default matches process name only: Without -f, pgrep matches only the process name (like ps -C). Add -f to match full command line arguments.