

LINUX

GNU Screen & Linux Job Control

A practical course on keeping long-running work alive on remote Linux machines — screen sessions, windows and split regions, shell job control, job IDs versus PIDs, nohup and disown, rescuing running jobs into screen with reptyr, and automating screen — closing with a full worked SSH scenario.

10 Chapters

Philip Osztromok

Generated with Claude

CHAPTER 1: WHY SCREEN? TERMINAL MULTIPLEXERS & PERSISTENT SESSIONS

GNU Screen & Linux Job Control

Chapter 1 · Why Screen? Terminal Multiplexers & Persistent Sessions

You SSH into a server, start a backup, a large file copy, or a database import that will take an hour — and then your Wi-Fi drops, your laptop sleeps, or you simply need to go home. When you reconnect, the job is gone. This course is about stopping that from happening, and about understanding exactly how Linux manages the jobs you start.

Why Jobs Die When the Connection Drops

Every program you start from an SSH session is attached to a **terminal** — strictly, a *pseudo-terminal* that `sshd` creates for your login. Your shell, and every job it starts, reads from and writes to that terminal. When the connection breaks, the terminal "hangs up." The system then sends the **SIGHUP** ("hang-up") signal to the shell, and the shell passes it on to its jobs. The default reaction to SIGHUP is to exit, so your hour-long import stops wherever it was.

```
# Without screen: the job belongs to the SSH session's terminal
laptop  →ssh→  sshd  →  bash  →  long-running job
                                   (connection drops → SIGHUP → bash and the job exit)

# With screen: the job belongs to screen's own terminal
laptop  →ssh→  sshd  →  bash  →  screen client
                                   | (connection drops → only the client dies)
                                   screen server → bash → long-running job
                                   (keeps running, waiting for you to reattach)
```

What a Terminal Multiplexer Does

GNU Screen is a *terminal multiplexer*. It runs a background server process that owns its own pseudo-terminals, and your shells and jobs run inside those instead of inside the SSH session's terminal. What you see is just a client connected to that server. Disconnect, and only the client disappears; the server, and everything running in it, carries on. This gives you three things:

- **Persistence** — *detach* from a session and *reattach* later, even from a different computer (Chapter 2).
- **Multiplexing** — many shells, called *windows*, inside one session, and split views of several at once (Chapter 4).
- **Sharing and logging** — two people can attach to the same session, and a window's output can be logged to a file (Chapter 9).

THE PROGRAMS DON'T KNOW

As the GNU Screen documentation puts it, applications running inside screen don't even know the terminal has detached. No special support is needed: any command-line program works.

A Short History, and Screen vs. tmux

Screen was designed by Oliver Laumann and Carsten Bormann at the Technical University of Berlin and first published in 1987. It is now part of the GNU Project and still actively maintained, with a 5.x release line. Its best-known alternative, **tmux**, first appeared in 2007 and has a similar feature set.

	GNU SCREEN	TMUX
Age	1987	2007
Prefix key	Ctrl-a	Ctrl-b
Licence	GPL	ISC
Strengths	Very widely available, including on old systems; serial-console support	More modern design, easier scripting and configuration

The ideas are the same in both — sessions, detaching, windows, panes — so what you learn here carries over. This course uses screen because it is the one you are most likely to find already installed on an unfamiliar server.

Installing and Checking Screen

```
# Is it already installed?
screen -v

# Debian / Ubuntu
sudo apt install screen

# Fedora (and RHEL-family systems, via EPEL where needed)
sudo dnf install screen

# Arch Linux
sudo pacman -S screen
```

NOT EVERYWHERE BY DEFAULT

Many minimal server images don't include screen, and Red Hat Enterprise Linux 8 dropped it from its main repositories in favour of tmux. On those systems it can usually be installed from EPEL, or you can use tmux instead — the concepts in this course still apply.

What About Job Control?

Screen is one answer to the "my job died" problem, but not the only one. The shell itself has **job control**: running commands in the background with `&`, pausing them with `Ctrl-Z`, listing them with `jobs`, and protecting them from SIGHUP with `nohup` and `disown`. Chapters 5–7 cover these, and Chapter 8 combines both — including the common situation of wishing you had started a job inside screen *after* it is already running.

Hands-On Exercises

EXERCISE 1

In your own words, explain why a long-running command started directly in an SSH session stops when the connection drops, and why the same command started inside screen keeps running.

 [View solution](#)

EXERCISE 2

On a Linux machine (a virtual machine or WSL is fine), check whether screen is installed, install it if not, and report its version. Then run `ps -o pid, tty, comm` both outside and inside a plain `screen` session and note what changes in the TTY column.

 [View solution](#)

EXERCISE 3

For each situation, decide whether screen is a good fit and explain why: (a) a two-hour database import over SSH; (b) a web server that should start automatically at boot; (c) watching a log file in one window while editing a config file in another.

 [View solution](#)

CHAPTER 1 QUICK REFERENCE

- Jobs started in an SSH session belong to its pseudo-terminal; a dropped connection sends **SIGHUP**, which stops them by default
- **GNU Screen** is a terminal multiplexer: a background server owns the terminals your jobs run in, so they survive disconnection
- It provides persistence (detach/reattach), multiple windows and split regions, and session sharing and logging
- Designed by Oliver Laumann and Carsten Bormann, first published in 1987; tmux (2007) is the main alternative
- `screen -v` checks the version; install with `apt`, `dnf` (EPEL on RHEL-family), or `pacman`
- Shell job control (`&`, `Ctrl-Z`, `jobs`, `nohup`, `disown`) is the other half of this course

CHAPTER 2: STARTING, DETACHING & REATTACHING SESSIONS

GNU Screen & Linux Job Control

Chapter 2 · Starting, Detaching & Reattaching Sessions

Chapter 1 explained why screen keeps jobs alive. This chapter is the everyday workflow: start a session, leave it running, and come back to it — including what to do when the session is still attached somewhere else.

Starting a Session

```
# Start an unnamed session
screen

# Start a session with a meaningful name (recommended)
screen -S backup

# Start a session that runs one command directly
screen -S import mysql mydb < dump.sql
```

A new session opens a fresh shell that looks exactly like a normal terminal. Nothing on screen tells you you're inside it, which is easy to forget. The `-S` option names the session, and the name shows up later when you list or reattach sessions (Chapter 3). If you give screen a command, the session ends when that command finishes.

AM I INSIDE SCREEN?

Run `echo $STY`. Inside a screen session it prints the session's ID, such as `12345.backup`; outside, it prints nothing.

The Command Key: Ctrl-a

Screen listens for one special key combination, the **command character**, which is `Ctrl-a` by default. You press `Ctrl-a`, let go, and then press a second key to give screen a command. Everything else you type goes straight to the program in the window.

KEYS	WHAT IT DOES
<code>Ctrl-a d</code>	Detach: leave the session running and return to your normal shell
<code>Ctrl-a ?</code>	Show the key bindings
<code>Ctrl-a k</code>	Kill the current window (asks for confirmation)
<code>Ctrl-a \</code>	Quit: kill all windows and end the session
<code>Ctrl-a a</code>	Send a real <code>Ctrl-a</code> to the program in the window

CTRL-A NORMALLY MEANS SOMETHING ELSE

In the bash shell, `Ctrl-a` moves the cursor to the start of the line. Inside screen that keystroke is taken by screen, so use `Ctrl-a a` to get the old behaviour.

Detaching

Press `Ctrl-a d`. You're returned to the shell you started screen from, with a message showing which session you left:

```
[detached from 12345.backup]
```

Everything inside the session keeps running. You can now log out, or simply lose your connection — the session carries on either way. Detaching is deliberate; losing the connection is accidental; to screen they have the same effect.

Reattaching

```
# Reattach when there is only one detached session
screen -r

# Reattach a particular session by name or ID
screen -r backup
screen -r 12345
```

If there are several detached sessions, a plain `screen -r` doesn't guess: it lists them and asks you to choose one. This is where naming sessions with `-S` pays off.

When the Session Is Still Attached Somewhere Else

A common situation: your connection dropped, but the server hasn't noticed yet, so it still thinks you're attached. `screen -r` refuses, because the session isn't detached. You have three choices.

COMMAND	BEHAVIOUR	USE IT WHEN
<code>screen -d -r name</code>	Reattach, detaching it from the other terminal first if necessary	The old connection is dead or you've moved to a new computer
<code>screen -D -r name</code>	As above, but also logs out the other terminal	You want to be sure the old login is closed
<code>screen -x name</code>	Attach <i>as well</i> , without detaching the other terminal	You want to share the session or watch it from two places (Chapter 9)

Reattach-or-Create

Screen can also create the session if it doesn't exist yet, which makes a convenient "take me to my work session" command:

```
# Reattach "work", detaching it elsewhere if needed, or create it
screen -d -R -S work

# "Attach here and now": reattach, logging out elsewhere if needed, or create
screen -D -R
```

A HANDY ALIAS

Adding `alias work='screen -d -R -S work'` to your `~/.bashrc` gives you a single command that always drops you back into the same session.

Detaching From Outside, and Renaming

```
# Detach a session from another terminal, without attaching to it
screen -d backup

# Inside a session: rename it (press Ctrl-a, then type : to get a command prompt)
Ctrl-a :sessionname nightly-backup
```

Ending a Session

A session ends when its last window closes. The usual way is simply to type `exit` in the shell (or let the command you started finish). `Ctrl-a \` closes every window at once, after asking you to confirm.

DETACH, DON'T EXIT

A very common mistake is typing `exit` when you meant to detach. That ends the shell and stops anything running in it. If you want to come back later, use `Ctrl-a d`.

Hands-On Exercises

EXERCISE 1

Start a session named `counter` and run `while true; do date; sleep 5; done` in it. Detach, wait a minute, reattach, and confirm the loop kept running while you were away. Then end the session cleanly.

 [View solution](#)

EXERCISE 2

Open two terminals. Attach to one session from the first. In the second, try `screen -r`, then `screen -x`, then `screen -d -r`. Describe what happens in each terminal after each command.

 [View solution](#)

EXERCISE 3

Write a `~/.bashrc` alias or small function that takes you into a session called `work`, whether or not it exists and whether or not it's attached elsewhere. Explain which options you chose and why.

 View solution

CHAPTER 2 QUICK REFERENCE

- `screen` starts a session; `screen -S name` names it; `echo $STY` shows whether you're inside one
- `Ctrl-a` is the command key: `d` detach, `?` help, `k` kill window, `\` quit, `a` send a literal Ctrl-a
- `screen -r` reattaches (by name or ID if there are several)
- `-d -r` detaches elsewhere then reattaches; `-D -r` also logs out the other terminal; `-x` shares the session
- `screen -d -R -S name` reattaches or creates; `screen -D -R` means "attach here and now"
- `screen -d name` detaches from outside; `Ctrl-a :sessionname new` renames
- A session ends when its last window closes — detach with `Ctrl-a d` instead of typing `exit`

CHAPTER 3: LISTING & IDENTIFYING SESSIONS

GNU Screen & Linux Job Control

Chapter 3 · Listing & Identifying Sessions

Once you use screen regularly, you'll have several sessions running — some you remember, some you don't. This chapter shows how to list them, read their IDs, tell which state each one is in, and clean up the ones you no longer need.

Listing Sessions

```
screen -ls          # or the longer form: screen -list
```

Typical output looks like this:

```
There are screens on:
    24817.backup      (27/09/26 10:15:02)      (Detached)
    23904.pts-0.myserver (27/09/26 09:02:44)      (Attached)
2 Sockets in /run/screen/S-philip.
```

`screen -ls` doesn't start a session; it just prints one line per session, with its ID, often its start time, and its state. The exact date format and the socket directory vary between systems and versions.

Reading a Session ID

PART	EXAMPLE	MEANING
PID	24817	The process ID of that session's screen server process
Name	backup	The name you gave with <code>-S</code>
tty.host	pts-0.myserver	Used as the name when you didn't give one: the terminal it was started from and the machine's hostname

So a session started with plain `screen` gets an ID like `23904.pts-0.myserver`, while one started with `screen -S backup` gets `24817.backup`. The PID really is a process: you can see it with `ps`.

```
ps -p 24817 -o pid,user,cmd
#  PID USER      CMD
# 24817 philip    SCREEN -S backup
```

SCREEN IN CAPITALS

The server process shows up in `ps` as `SCREEN` in capitals; the lower-case `screen` process is a client attached to it. That's a quick way to tell them apart when you look through a process list (Chapter 6 goes further).

Referring to a Session

Anywhere `screen` wants a session — `-r`, `-x`, `-d`, `-S` with `-X` — you can use the full ID, the PID alone, the name alone, or the start of the name, as long as only one session matches.

```
screen -r 24817.backup    # full ID
screen -r 24817           # PID only
screen -r backup          # name only
```

NAMES DON'T HAVE TO BE UNIQUE

Screen lets you start two sessions both called `backup`. If you then use the name alone, `screen` lists the matches and asks you to be more specific. Using the PID, or the full `PID.name`, always identifies exactly one session.

Session States

STATE	MEANING	WHAT TO DO
Detached	Running, with nobody connected	<code>screen -r</code> to reattach

STATE	MEANING	WHAT TO DO
Attached	Running, and connected to a terminal — possibly one that's no longer really there	<code>screen -d -r</code> or <code>screen -x</code> (Chapter 2)
Multi	Shared in multi-user mode (Chapter 9)	Attach as allowed by its access settings
Dead ???	The server process is gone (for example after a crash or reboot) but its socket file is left behind	<code>screen -wipe</code> to remove it

Where Sessions Live: Sockets

Each session has a **socket** — a special file clients use to talk to the server — in a per-user directory, shown on the last line of `screen -ls`. On many modern systems that's `/run/screen/S-yourname`; older systems use `/var/run/screen` or `/tmp/screens`, and the `SCREENDIR` environment variable can change it. The socket's file name is the session ID.

SESSIONS DON'T SURVIVE A REBOOT

A screen session is a set of running processes. When the machine restarts, they're gone, and all that may be left is a dead socket. Screen protects jobs from lost connections, not from the machine going down.

Ending Sessions You Don't Need

```
# From inside the session: quit all windows
Ctrl-a \

# From outside: send the "quit" command to a session
screen -S backup -X quit

# Remove the leftovers of dead sessions
screen -wipe
```

`-X` sends one of screen's own commands to a running session without attaching to it; Chapter 9 uses it much more. If a session is truly stuck, `kill` on the server PID also works, but it's a last

resort — it gives the programs inside no chance to shut down in an orderly way.

Hands-On Exercises

EXERCISE 1

Start one unnamed session and two named ones (`web` and `db`), detaching from each. Run `screen -ls` and, for every line, identify the PID, the name, and the state.

 [View solution](#)

EXERCISE 2

Start two sessions both named `test`. Try `screen -r test`, then work out two different ways to reattach to one specific session.

 [View solution](#)

EXERCISE 3

Without attaching, end the `web` session from Exercise 1. Then deliberately `kill -9` the server PID of `db`, and use `screen -ls` and `screen -wipe` to see and clean up what's left.

 [View solution](#)

CHAPTER 3 QUICK REFERENCE

- `screen -ls` (`-list`) lists sessions without starting one
- Session IDs are `PID.name`, or `PID.tty.host` for unnamed sessions; the PID is the screen server process, shown as `SCREEN` in `ps`
- Refer to a session by full ID, PID, or unambiguous name; use the PID if names clash
- States: Detached, Attached, Multi, and Dead (a leftover socket)
- Sockets live in a per-user directory such as `/run/screen/S-user`; sessions don't survive a reboot

- End a session with `Ctrl-a \` or `screen -S name -X quit`; tidy dead ones with `screen -wipe`

CHAPTER 4: WINDOWS & SPLIT REGIONS INSIDE A SESSION

GNU Screen & Linux Job Control

Chapter 4 · Windows & Split Regions Inside a Session

So far every session has held a single shell. A session can hold many, each in its own **window**, and you can split your terminal to see several windows at once. This is screen's "multiplexing" side (Chapter 1): one SSH connection, many things going on.

SCREEN'S WORDS

A **session** is the whole thing you attach to. A **window** is one shell or program inside it. A **region** is a part of your terminal display that shows one window. Splitting the display makes regions; it doesn't create windows.

Creating and Moving Between Windows

KEYS	WHAT IT DOES
Ctrl-a c	Create a new window with a shell and switch to it
Ctrl-a n / Ctrl-a p	Next / previous window
Ctrl-a 0 - 9	Jump straight to window 0–9
Ctrl-a Ctrl-a	Toggle back to the window you were in before
Ctrl-a '	Prompt for a window number or name to switch to
Ctrl-a "	Show a list of all windows to pick from
Ctrl-a w	Briefly show the window list on the message line

Windows are numbered from 0. When you close one, its number becomes free and a new window may reuse it, so the numbers can end up with gaps.

Naming and Closing Windows

Window numbers are hard to remember. `Ctrl-a A` lets you type a title for the current window, which then appears in the `Ctrl-a "` list and can be used with `Ctrl-a '`. To close a window, exit its shell, or press `Ctrl-a k` and confirm. When the last window closes, the session ends (Chapter 2).

Opening a Window From the Command Line

Typing `screen` *inside* a session doesn't start a second, nested session: `screen` notices it's already running and opens a new window in the current session instead. That gives a quick way to open a window that runs a particular command, with a title:

```
# Inside a session: new window titled "logs" running tail
screen -t logs tail -f /var/log/syslog

# Inside a session: new window titled "top"
screen -t top top
```

A window started this way closes when its command ends. Chapter 8 shows how to do the same thing from *outside* the session.

Splitting the Display Into Regions

KEYS	WHAT IT DOES
<code>Ctrl-a S</code>	Split the current region horizontally (one above the other)
<code>Ctrl-a </code>	Split the current region vertically (side by side)
<code>Ctrl-a Tab</code>	Move the input focus to the next region
<code>Ctrl-a X</code>	Remove the current region
<code>Ctrl-a Q</code>	Remove every region except the current one

```
# After Ctrl-a S, then Ctrl-a | in the lower region
| window 0: vim config.yaml |
```

window 1: tail -f	(empty region)
-------------------	----------------

A NEW REGION STARTS EMPTY

A fresh split shows nothing until you put a window in it. Move to the new region with `Ctrl-a Tab`, then either create a window there with `Ctrl-a c` or show an existing one with `Ctrl-a n`, `Ctrl-a "`, or a number.

SPLITS ARE USUALLY LOST WHEN YOU DETACH

Regions belong to your *display*, not the session, so after you detach and reattach you normally get a single region again. The windows themselves, and everything running in them, are untouched. Screen's `layout` commands can save and restore an arrangement, if you want your splits back.

Watching Windows for Activity

When you have several windows, you don't want to keep flicking between them to see whether something happened. Screen can watch them for you:

- `Ctrl-a M` — monitor the current window for **activity**: you'll get a message when new output appears there while you're elsewhere.
- `Ctrl-a _` — monitor for **silence**: you'll get a message when the window stops producing output for a while, useful to know a long job has finished or hung.

Hands-On Exercises

EXERCISE 1

In one session, create three windows titled `edit`, `logs` and `shell`. Practise switching between them with `Ctrl-a n`, a number, `Ctrl-a "` and `Ctrl-a '`. Then close `logs` and open a new window: what number does it get, and why?

[View solution](#)

EXERCISE 2

Build the layout in this chapter's diagram: an editor across the top, `tail -f` on a log bottom-left, and `top` bottom-right. Then detach, reattach, and describe what you see.

[View solution](#)

EXERCISE 3

In one window, run `sleep 30; echo done` and turn on silence monitoring. Switch to another window and wait. What happens, and why might this be more useful than activity monitoring for a long-running job?

[View solution](#)

CHAPTER 4 QUICK REFERENCE

- A session holds **windows**; the display is divided into **regions**, each showing one window
- `Ctrl-a c` new window; `n`/`p` next/previous; `0-9` jump; `Ctrl-a` toggle; `'` prompt; `"` list; `w` quick list
- `Ctrl-a A` names a window; `Ctrl-a k` kills it
- Inside a session, `screen -t title command` opens a new window instead of a nested session
- `Ctrl-a S` horizontal split, `|` vertical split, `Tab` next region, `x` remove region, `Q` keep only current
- New regions start empty; splits are usually lost on reattach (use `layout` to keep them)
- `Ctrl-a M` monitors for activity, `Ctrl-a _` for silence

CHAPTER 5: SHELL JOB CONTROL BASICS

GNU Screen & Linux Job Control

Chapter 5 · Shell Job Control Basics

Screen gives you many terminals. **Job control** lets one terminal run several things at once: starting commands in the background, pausing them, and moving them between the background and the foreground. It's built into the shell. This chapter uses bash, but zsh and most other modern shells behave the same way. It's also the foundation for Chapter 8, where you move an already-running job into screen.

Foreground and Background

A command normally runs in the **foreground**: it owns the terminal, receives what you type, and you get your prompt back only when it finishes. Add `&` to the end of a command and it runs in the **background** instead, and the prompt comes back straight away.

```
$ sleep 300 &  
[1] 48213  
$ # prompt is back immediately
```

The shell prints two numbers: `[1]` is the **job number**, and `48213` is the **process ID** (PID). They are different things, and Chapter 6 is all about the difference. When the job finishes, the shell tells you the next time it shows a prompt:

```
[1]+  Done                  sleep 300
```

Pausing a Job: Ctrl-Z

If a command is already running in the foreground, `Ctrl-Z` **suspends** it. The shell sends the job the `SIGTSTP` signal: the job stops where it is — not ended, just frozen — and you get your prompt back.

```
$ tar czf backup.tar.gz /home/philip
^Z
[1]+  Stopped                  tar czf backup.tar.gz /home/philip
```

CTRL-Z IS NOT CTRL-C

`Ctrl-C` sends `SIGINT`, which normally ends the program. `Ctrl-Z` only pauses it. A stopped job does no work at all until you resume it — a paused backup isn't making progress in the background.

Listing Jobs

```
$ jobs
[1]-  Stopped                  tar czf backup.tar.gz /home/philip
[2]+  Running                  sleep 300 &
```

Each line shows the job number, a marker, the state (Running, Stopped, Done), and the command. The `+` marks the **current job**, the one commands act on by default; the `-` marks the **previous job**. `jobs -r` lists only running jobs, and `jobs -s` only stopped ones.

Moving Jobs Around: fg and bg

COMMAND	WHAT IT DOES
<code>fg</code>	Bring the current job to the foreground (resuming it if stopped)
<code>fg %2</code>	Bring job 2 to the foreground
<code>bg</code>	Resume the current stopped job in the background
<code>bg %1</code>	Resume job 1 in the background
<code>kill %1</code>	Send job 1 the <code>SIGTERM</code> signal, asking it to end
<code>wait</code>	Wait until all background jobs have finished

THE MOST USEFUL PATTERN

You start a long command, then realise you want your prompt back. Press `Ctrl-Z`, then type `bg`. The job carries on in the background, and you can keep working. This is also the first step in Chapter 8's rescue technique.

Referring to Jobs: Job Specs

SPEC	REFERS TO
<code>%1</code>	Job number 1
<code>%%</code> or <code>%+</code>	The current job (marked <code>+</code>)
<code>%-</code>	The previous job (marked <code>-</code>)
<code>%tar</code>	The job whose command starts with <code>tar</code>
<code>%?backup</code>	The job whose command contains <code>backup</code>

%1 AND 1 ARE NOT THE SAME

`kill %1` means "job number 1 in this shell." `kill 1` means "process ID 1" — on most systems that's the init process, and an ordinary user isn't allowed to signal it. Forgetting the `%` is a common and potentially confusing mistake.

Background Jobs and the Terminal

- **Output still appears.** A background job's output is printed to your terminal, mixed in with whatever you're doing. Redirect it to a file to avoid this: `long-command > out.log 2>&1 &`.
- **Input stops the job.** If a background job tries to read from the terminal, it's sent the `SIGTTIN` signal and stops, waiting for you to bring it to the foreground.
- **Leaving with stopped jobs.** If you try to exit the shell while jobs are stopped, bash warns you. Exiting again anyway usually ends them.

JOB CONTROL BELONGS TO ONE SHELL

Job numbers only exist inside the shell that started the jobs. Another terminal, or a new login, has its own separate list — and when that shell goes away, so does its job list. That's exactly why Chapter 7 is needed.

Hands-On Exercises

EXERCISE 1

Start `sleep 600` in the foreground, suspend it, resume it in the background, start a second `sleep 900 &`, then use `jobs` to identify the current and previous jobs. Finally, end both using job specs.

 [View solution](#)

EXERCISE 2

Run `read -p "Name? " name &`. What does `jobs` show, and why? How do you finish the command?

 [View solution](#)

EXERCISE 3

Start `ping localhost` in the foreground. Without stopping it permanently, get your prompt back so that it keeps running but stops cluttering your screen. What problem remains, and how would you have started it differently?

 [View solution](#)

CHAPTER 5 QUICK REFERENCE

- `command &` runs in the background and prints `[job] PID`
- `Ctrl-Z` suspends the foreground job (`SIGTSTP`); `Ctrl-C` ends it (`SIGINT`)
- `jobs` lists jobs (`-r` running, `-s` stopped); `+` is the current job, `-` the previous
- `fg` / `bg` move jobs to the foreground/background; `kill %n` ends one; `wait` waits for all
- Job specs: `%n`, `%%` / `%+`, `%-`, `%string`, `!?string` — never forget the `%`

- Background jobs still print to the terminal, and stop (`SIGTTIN`) if they try to read from it
- Job numbers belong to one shell and disappear with it

CHAPTER 6: JOB IDS, PIDS & SESSION IDS: TELLING THEM APART

GNU Screen & Linux Job Control

Chapter 6 · Job IDs, PIDs & Session IDs: Telling Them Apart

By now you've met several kinds of number: `[1]` when a job starts, `48213` next to it, the `24817` at the front of a screen session ID, and window numbers inside a session. They're easy to mix up, and using the wrong one with `kill` can end the wrong thing. This chapter sorts them out and shows how to find any process, including one hidden inside a screen session.

Four Numbers, Four Scopes

NUMBER	EXAMPLE	WHO ASSIGNS IT	WHERE IT'S VALID
Job number	<code>%1</code>	Your shell	Only in the shell that started the job
Process ID (PID)	<code>48213</code>	The Linux kernel	System-wide, for as long as the process exists
Screen session ID	<code>24817.backup</code>	Screen (the PID of its server, plus a name)	For your user, through screen's commands
Window number	<code>2</code>	Screen	Only inside that one session

Only the PID means something to the kernel. The other three are labels kept by a particular program: the shell, or screen. That's why `kill 48213` works from any terminal, but `kill %1` only works in the shell that owns job 1.

One Job, Several Processes

A job isn't always one process. A pipeline such as `cat big.log | grep ERROR | sort` is a single job made of three processes, each with its own PID. The shell groups them into a **process group**, and the job's commands act on the whole group: `Ctrl-Z` or `kill %1` reaches all three.

```
$ cat big.log | grep ERROR | sort > errors.txt &
[1] 50122                # bash shows the PID of the last process in the pipeline
$ jobs -l
[1]+ 50120 Running      cat big.log
      50121              | grep ERROR
      50122              | sort > errors.txt &
```

Getting PIDs From the Shell

COMMAND

GIVES YOU

`jobs -l`

The job list with the PID of every process in each job

`jobs -p`

Just the PIDs (one per job, for a pipeline the group leader)

`$!`

The PID of the most recent background job

`$$`

The PID of the current shell itself

```
# Save the PID of a background job for later
./import.sh > import.log 2>&1 &
IMPORT_PID=$!
echo "Import running as PID $IMPORT_PID"
```

Finding Processes With ps and pgrep

Outside the shell that started them, you find processes with system tools instead of `jobs`.

```
# All your processes, with parent PID, group, terminal and state
ps -u $USER -o pid,ppid,pgid,ttty,stat,cmd

# Find processes by name, showing their full command line
pgrep -a sleep

# Match against the whole command line, not just the program name
pgrep -af "import.sh"
```

STAT	LETTER	MEANING
	R	Running (or ready to run)
	S	Sleeping, waiting for something
	T	Stopped — for example after <code>Ctrl-Z</code>
	Z	Zombie: finished, but its parent hasn't collected it yet
	+	In the foreground process group of its terminal

Finding a Process Inside a Screen Session

Every process in a session is a descendant of that session's `SCREEN` server process, whose PID is the number at the front of the session ID (Chapter 3). So `pstree` can show you exactly what a session is running:

```
$ screen -ls
      24817.backup      (Detached)
$ pstree -p -a 24817
SCREEN,24817 -S backup
├─bash,24818
│   └─rsync,24901 -a /data /mnt/backup
└─bash,25003
    └─top,25010
```

This session has two windows: one running `rsync` and one running `top`. The `-p` option shows PIDs and `-a` shows command-line arguments.

The Reverse: Which Session Is This Process In?

Sometimes you find a PID with `pgrep` and want to know which session it lives in. Walk up its parents with `pstree -s`:

```
$ pstree -s -p 24901
systemd(1)──SCREEN(24817)──bash(24818)──rsync(24901)
```

The `SCREEN(24817)` ancestor tells you it's in session `24817`. Two environment variables that screen sets for every window give the same information from the inside:

```
# Inside a window
echo $STY      # session ID, e.g. 24817.backup
echo $WINDOW   # window number, e.g. 0

# From outside, for one of your own processes
tr '\0' '\n' < /proc/24901/environ | grep -E '^(STY|WINDOW)='
```

ENVIRONMENT VARIABLES ARE A SNAPSHOT

`/proc/PID/environ` shows the environment the process *started* with. If a process was started elsewhere and later moved into screen (Chapter 8), it won't have `STY` set, so trust the `pstree -s` ancestry instead.

Hands-On Exercises

EXERCISE 1

Start `sleep 1000 | cat &`. Find its job number, the PID of each process in it, and the PID of your shell. Then open a second terminal and end the job from there. Why can't you use `%1` in the second terminal?

 [View solution](#)

EXERCISE 2

Create a detached session named `jobs` with two windows, one running `sleep 2000` and one running `top`. Without attaching, use `screen -ls` and `pstree` to show what's running inside it.

 [View solution](#)

EXERCISE 3

Using `pgrep`, find the `sleep 2000` process from Exercise 2, then prove which screen session and window it belongs to in two different ways.

 [View solution](#)

CHAPTER 6 QUICK REFERENCE

- Job numbers belong to one shell; PIDs are system-wide; session IDs and window numbers belong to screen
- A job can contain several processes (a pipeline), grouped into one process group
- `jobs -l` shows PIDs, `jobs -p` only PIDs; `$!` is the last background PID, `$$` the shell's own
- `ps -o pid,ppid,pgid,tty,stat,cmd` and `pgrep -a / -af` find processes; `STAT` T means stopped
- `pstree -p -a SCREEN_PID` shows everything in a session; `pstree -s -p PID` shows a process's ancestors
- Inside screen, `$STY` is the session ID and `$WINDOW` the window number

CHAPTER 7: KEEPING JOBS ALIVE AFTER LOGOUT

GNU Screen & Linux Job Control

Chapter 7 · Keeping Jobs Alive After Logout

Chapter 1 explained, briefly, why a job dies when your SSH connection drops: the hang-up signal, `SIGHUP`. Chapter 5 showed that job numbers belong to one shell and vanish with it. This chapter looks at that problem in detail, and at the tools other than screen that solve it — `nohup`, `disown` and `setsid` — plus one systemd setting that can undo all of them.

How SIGHUP Reaches Your Jobs

```

connection drops
  |
  ▼
the SSH session's terminal hangs up
  |
  ▼
the kernel sends SIGHUP to your shell (the terminal's controlling process)
  |
  ▼
bash sends SIGHUP to every job in its job table
(stopped jobs are also sent SIGCONT, so they wake up and receive it)
  |
  ▼
most programs exit when they receive SIGHUP

```

Every protection in this chapter breaks one link in that chain: either the job ignores the signal, or the shell never sends it, or the job isn't attached to the terminal at all.

"IT SURVIVED LAST TIME" ISN'T PROOF

Whether an ordinary background job outlives your session depends on how the session ends — typing `exit`, closing a window, or a dropped connection — and on shell settings such as bash's `huponexit` option. A job that happened to survive once may not the next time. If it matters, protect it deliberately.

nohup: Start a Job That Ignores SIGHUP

```
$ nohup ./import.sh &
[1] 70311
nohup: ignoring input and appending output to 'nohup.out'

# Better: choose where the output goes yourself
$ nohup ./import.sh > import.log 2>&1 &
```

`nohup` runs a command with `SIGHUP` set to be ignored. Because the terminal may not exist later, it also stops the command reading from the terminal, and if its output would go to the terminal, sends it to a file called `nohup.out` instead. Note the `&`: `nohup` doesn't put a job in the background on its own.

NOHUP MUST BE USED AT THE START

`nohup` is a program that starts another program. It can't be applied to a job that's already running — for that, you need `disown`.

disown: Protect a Job That's Already Running

`disown` is a bash builtin that works on jobs you've already started. It's the tool to reach for when you forgot `nohup`.

COMMAND	EFFECT
<code>disown %1</code>	Remove job 1 from the shell's job table, so the shell will never send it <code>SIGHUP</code> . It no longer appears in <code>jobs</code> .
<code>disown -h %1</code>	Keep job 1 in the job table, but mark it so the shell won't send it <code>SIGHUP</code> . You can still use <code>fg</code> on it until you log out.
<code>disown -a</code>	Apply to all jobs
<code>disown -r</code>	Apply only to running jobs

```
# You started something long in the foreground and need to leave
$ ./import.sh
```

```
^Z
[1]+  Stopped                  ./import.sh
$ bg                          # resume it in the background first
$ disown %1                   # then detach it from the shell
$ exit
```

ALWAYS BG BEFORE DISOWN

If you disown a *stopped* job, it stays stopped, with no shell left to resume it. Run `bg` first. You can still wake a stopped process later with `kill -CONT PID`, if you know its PID (Chapter 6).

WHAT DISOWN DOESN'T FIX

`disown` only stops the shell sending `SIGHUP`. The job is still attached to the old terminal: if it later tries to write to it after the terminal has gone, the write fails and many programs stop. Redirect output to a file when you start anything you might need to disown — or use `screen`.

setsid: Start Completely Detached

`setsid` runs a command in a brand-new session with no controlling terminal at all, so no terminal hang-up can ever reach it.

```
$ setsid ./import.sh > import.log 2>&1 < /dev/null &
```

Comparing the Options

TOOL	USE ON A RUNNING JOB?	CAN YOU INTERACT WITH IT LATER?	BEST FOR
<code>nohup</code>	No	No — read its log	Fire-and-forget commands you plan ahead
<code>disown</code>	Yes	No, once you log out	Rescuing a job you forgot to protect
<code>setsid</code>	No	No	Fully detached background programs

TOOL	USE ON A RUNNING JOB?	CAN YOU INTERACT WITH IT LATER?	BEST FOR
<code>screen</code>	Not directly (Chapter 8)	Yes — reattach and use it normally	Anything you want to watch, or type into, later

Screen's advantage is the last column. With the other tools, the job survives but you can never get its terminal back; with screen, the job's terminal survives too, so an interactive program — an installer asking questions, a database shell, `top` — carries on exactly as you left it.

The Catch: systemd's KillUserProcesses

On systems managed by systemd, the login manager has a setting, `KillUserProcesses=` in `/etc/systemd/logind.conf`. When it's `yes`, everything that belongs to your login session is killed when you log out — including `nohup` and `disown`ed jobs, and screen sessions.

CHECK YOUR SYSTEM

systemd's own default is `yes`, but many distributions set it to `no`. If your screen sessions vanish when you log out, this is the likely reason. Ways around it include asking an administrator to enable "lingering" for your user (`loginctl enable-linger`), or starting the session outside your login's scope, for example with `systemd-run --user --scope screen -S name`.

AND FOR REAL SERVICES

None of these tools is a substitute for a proper service. Anything that should start at boot and restart if it crashes belongs in a systemd unit (see the systemd in Depth course).

Hands-On Exercises

EXERCISE 1

In an SSH session, start `nohup sleep 600 &` and `sleep 700 &`. Close the terminal window (don't type `exit`). Log in again and use `pgrep -a sleep` to see which survived. Explain the result.

 [View solution](#)

EXERCISE 2

Start `sleep 800` in the foreground and protect it so it survives logout, without restarting it. Then explain the difference between using `disown` and `disown -h` here.

 [View solution](#)

EXERCISE 3

Find out whether your system kills user processes at logout. Where would you look, and what would you check first if screen sessions kept disappearing?

 [View solution](#)

CHAPTER 7 QUICK REFERENCE

- A lost terminal sends `SIGHUP` to the shell, which passes it to its jobs; most programs then exit
- `nohup cmd &` starts a job that ignores `SIGHUP`, with output in `nohup.out` unless redirected
- `disown %n` removes a running job from the shell's table; `disown -h` keeps it but stops the shell sending `SIGHUP`; always `bg` first
- `setsid` starts a command with no controlling terminal at all
- Only screen keeps the job's terminal, so you can interact with it again later
- systemd's `KillUserProcesses=yes` kills everything at logout, screen included; check `logind.conf`, and use lingering or `systemd-run --user --scope`

CHAPTER 8: MOVING AN ALREADY-RUNNING JOB INTO SCREEN

GNU Screen & Linux Job Control

Chapter 8 · Moving an Already-Running Job Into Screen

It happens to everyone: you start a long job straight in your SSH session, and only then realise you should have started it inside screen. This chapter covers the two different things "adding a job to a screen" can mean — rescuing a job that's *already running outside* screen, and *starting a new job* inside a session that's already running — and is honest about what each method can and can't do.

The Honest Answer First

Screen itself has **no command to adopt a running process**. A process's controlling terminal is set when it starts, and ordinary tools can't change it afterwards. That leaves three real options:

OPTION	JOB SURVIVES LOGOUT?	CAN YOU INTERACT WITH IT AGAIN?	NEEDS
<code>Ctrl-Z</code> , <code>bg</code> , <code>disown</code>	Yes (Chapter 7)	No	Nothing extra
<code>reptyr</code>	Yes	Yes — it becomes a normal screen window	The <code>reptyr</code> tool and ptrace permission
Stop it and restart it in screen	Yes	Yes	A job that's safe to restart

Method 1: Protect It With disown

If you only need the job to keep running, Chapter 7's sequence is enough, and it needs no extra software:

```
^Z          # suspend the foreground job
$ bg        # resume it in the background
$ disown %1  # stop the shell sending it SIGHUP
```

The job survives, but it's still writing to your old terminal, and you can never get it back to type into. For anything you want to *watch or control* later, you need reptyr.

Method 2: reptyr

reptyr, written by Nelson Elhage, attaches a running program to a new terminal. It uses the kernel's debugging interface, `ptrace`, to make the process switch its controlling terminal to the one you run reptyr in. Run it inside a screen window, and the job becomes part of that window: its output appears there, and `Ctrl-C` and `Ctrl-Z` reach it normally.

```
# Install it (Debian/Ubuntu; also packaged for Fedora and Arch)
sudo apt install reptyr

# 1. In the original terminal: suspend, background, find the PID, disown
^Z
$ bg
$ jobs -l
[1]+ 82417 Running      ./import.sh &
$ disown %1

# 2. Start (or attach to) a screen session
$ screen -S import

# 3. Inside the screen window: grab the process
$ reptyr 82417
```

The window now shows the job's output. You can detach with `Ctrl-a d`, log out, and reattach later exactly as if you'd started the job inside screen in the first place.

WHY DISOWN BEFORE REPTYR?

The original shell still considers the job one of its own. Disowning it means that when you close the original terminal, the shell won't send it `SIGHUP` or complain about running jobs. And if you've lost the original terminal already, find the PID with `pgrep -af` (Chapter 6).

The ptrace permission problem

Because `ptrace` is also what debuggers use to inspect other programs, many distributions restrict it for security, through the Yama setting `/proc/sys/kernel/yama/ptrace_scope`:

VALUE	MEANING	EFFECT ON REPTYR
0	Classic: you can trace any of your own processes	Works
1	Restricted: you can only trace your own descendants	Fails for normal users, since the job isn't reptyr's child
2	Only administrators can trace	Only works with <code>sudo</code>
3	No tracing at all	Doesn't work

```
# Check the current setting
cat /proc/sys/kernel/yama/ptrace_scope

# Allow it temporarily (until reboot), as root
echo 0 | sudo tee /proc/sys/kernel/yama/ptrace_scope

# Or run reptyr itself with sudo, when scope is 1 or 2
sudo reptyr 82417
```

LOOSENING PTRACE IS A SECURITY TRADE-OFF

The restriction exists to stop a compromised program from reading the memory of your other programs, such as passwords in an SSH agent or browser. Change it only temporarily, only if you're allowed to, and on shared or work systems ask an administrator first. If the file doesn't exist, your kernel doesn't use Yama and this restriction doesn't apply.

reptyr's limits

- It works most reliably on a **single process**. A pipeline, or a program that has started its own child processes on the same terminal, may not move cleanly; newer versions have a `-T` ("steal") mode designed for those cases.
- It can only take **your own** processes (or anyone's, with `sudo`).
- Output the job wrote to the old terminal *before* the move isn't copied over — you only see what it writes afterwards.
- Programs that draw full-screen interfaces may need a redraw (often `Ctrl-L`) after the move.

Method 3: Restart It Properly

If the job can be safely stopped and started again — it hasn't done much yet, or it can resume from where it left off, as `rsync` can — the simplest and most reliable fix is to stop it and start it again inside `screen`. Don't reach for `reptyr` just to avoid a restart that costs nothing.

Adding a New Job to a Session That's Already Running

The other meaning of "add a job to a screen" is starting a new command inside an existing session, often without attaching. `Screen`'s `-X` option sends a command to a session, and `screen` is itself one of `screen`'s commands: it opens a new window.

```
# Open a new window titled "logs" in the session "work", running tail
screen -S work -X screen -t logs tail -f /var/log/syslog

# Start a whole new session in the background, running one job
screen -dmS import ./import.sh

# Check it worked
screen -ls
```

From inside the session you can do the same thing with `screen -t title command` (Chapter 4). Chapter 9 covers `-dm` and `-X` in depth, including typing commands into an existing window.

THE BEST RESCUE IS NOT NEEDING ONE

The habit that avoids this whole chapter: before starting anything that might take more than a few minutes on a remote machine, run `screen -S name` first.

Hands-On Exercises

EXERCISE 1

Start `top` directly in a terminal (not in `screen`). Move it into a new `screen` session using `reptyr`, then detach, close the original terminal, and reattach to prove it survived. Record every command you used and any permission problem you hit.

[View solution](#)

EXERCISE 2

A colleague has a two-hour `pg_dump` running in their SSH session and must leave now. The server has no `reptyr` and they have no `sudo`. What would you advise, and what would they lose?

[View solution](#)

EXERCISE 3

With a detached session called `work` already running, add two new windows to it from outside — one running `htop` (or `top`) and one running `watch df -h` — without attaching. Then attach and confirm they're there.

[View solution](#)

CHAPTER 8 QUICK REFERENCE

- Screen can't adopt a running process by itself
- To keep a job alive only: `Ctrl-Z`, `bg`, `disown`
- To move it into screen: `Ctrl-Z`, `bg`, `jobs -l`, `disown`, start screen, then `reptyr PID` inside it
- `reptyr` needs ptrace permission: check `/proc/sys/kernel/yama/ptrace_scope` (0 works; 1 or 2 needs `sudo`)
- `reptyr` works best on single processes; `-T` steal mode is for harder cases; earlier output isn't copied
- If a restart is cheap, just restart the job inside screen
- Add a window to a running session: `screen -S name -X screen -t title cmd`; start a detached session: `screen -dmS name cmd`

CHAPTER 9: AUTOMATING SCREEN

GNU Screen & Linux Job Control

Chapter 9 · Automating Screen

Everything so far has been done by hand, one keystroke at a time. Screen can also be driven from scripts and from its configuration file: starting sessions in the background, typing commands into windows, logging what they print, and setting up the same layout every time you start. This chapter also covers scrollback and copying text, which make long-running output much easier to work with.

Starting Sessions Detached: -dm

```
# New detached session called "build", running a shell
screen -dmS build

# New detached session running one command (ends when the command ends)
screen -dmS backup rsync -a /data /mnt/backup

# Keep the window open after the command, by running it inside a shell
screen -dmS backup bash -c 'rsync -a /data /mnt/backup; exec bash'
```

`-d -m` creates a session without attaching to it, so it works from scripts, cron jobs and remote commands such as `ssh server 'screen -dmS job ./run.sh'`. The third form is useful when you want to reattach later and read the final output: without it, the window, and the session, vanish as soon as the command finishes.

Sending Commands to a Session: -X and -p

`-X` sends one of screen's own commands to a running session (Chapters 3 and 8 used `quit` and `screen`). `-p` picks which window it applies to, by number or title.

COMMAND

WHAT IT DOES

```
screen -S work -X screen -t logs tail -f app.log
```

Open a new window

COMMAND	WHAT IT DOES
<code>screen -S work -p 0 -X title editor</code>	Rename window 0
<code>screen -S work -p logs -X kill</code>	Close the window titled "logs"
<code>screen -S work -X quit</code>	End the whole session

Typing Into a Window: stuff

The `stuff` command puts text into a window's input, exactly as if someone had typed it. Add a newline to press Enter:

```
# Run "uptime" in window 0 of the session "work"
screen -S work -p 0 -X stuff $'uptime\n'

# Answer a prompt from a program that's waiting in window "install"
screen -S work -p install -X stuff $'y\n'
```

The `$'...\n'` is bash syntax for a string containing a real newline. This is the one way to add a job to an *existing* window, rather than a new one: type the command into its shell.

STUFF TYPES BLINDLY

Screen doesn't check what the window is doing. If the shell is busy, or a different program is in the foreground, your text goes to that program instead. Only use `stuff` on a window you know is sitting at a prompt or waiting for exactly that input.

Logging and Snapshots

METHOD	WHAT IT DOES
<code>screen -L</code>	Start with logging turned on; output goes to <code>screenlog.n</code> (<code>n</code> is the window number) in the starting directory
<code>screen -L -Logfile build.log</code>	Choose the log file name (newer versions)

METHOD	WHAT IT DOES
<code>Ctrl-a H</code>	Turn logging on or off for the current window
<code>Ctrl-a h</code>	Save a snapshot of what's on screen now to <code>hardcopy.n</code>
<code>-X hardcopy file</code>	Take the same snapshot from outside the session

CHECK ON A JOB WITHOUT ATTACHING

`screen -S backup -p 0 -X hardcopy /tmp/backup.txt` followed by `cat /tmp/backup.txt` shows you exactly what that window currently displays — handy in a monitoring script.

Scrollback and Copy Mode

Inside screen your terminal's own scroll bar usually doesn't show earlier output, because screen redraws the display itself. Screen keeps its own history, called **scrollback**, reached through **copy mode**.

KEYS	WHAT IT DOES
<code>Ctrl-a [</code>	Enter copy mode (also <code>Ctrl-a Esc</code>)
Arrow keys, <code>PgUp</code> / <code>PgDn</code>	Move around the history (vi-style keys like <code>k</code> , <code>j</code> and <code>/</code> to search also work)
<code>Space</code>	Mark the start of a selection, then again to mark the end and copy it
<code>Esc</code>	Leave copy mode without copying
<code>Ctrl-a]</code>	Paste the copied text into the current window

The default history is short. Set a bigger one in your configuration (below) with `defscrollback`, or for the current window with `Ctrl-a :scrollback 10000`.

The .screenrc File

Screen reads commands from `~/.screenrc` when it starts (or from another file given with `-c`). A system-wide file, usually `/etc/screenrc`, is read first. Any command you can type after `Ctrl-a :` can go in it.

```
# ~/.screenrc

# Skip the copyright page on startup
startup_message off

# Keep 10,000 lines of history per window
defscrollback 10000

# Always show a status line at the bottom: host, then the window list,
# with the current window in brackets
hardstatus alwayslastline "%H | %-w[%n %t]%-w"

# Open three named windows every time a new session starts
screen -t shell 0
screen -t logs 1 tail -f /var/log/syslog
screen -t top 2 top
select 0
```

THE STATUS LINE SOLVES "WHICH WINDOW AM I IN?"

A permanent status line showing the window list is the single most useful setting for everyday use: you always know which window you're in, and that you're inside screen at all.

TEST CHANGES SAFELY

Try a new configuration with `screen -c ~/test.screenrc -S test` before replacing your real `~/.screenrc`. A mistake in the file can leave every new session starting in an odd state.

Sharing a Session

As Chapter 2 showed, `screen -x` lets you attach to your own session from two terminals at once, which is useful for showing someone your work over a shared login. Letting a *different* user attach needs screen's `multiuser on` and `acladd` commands, and on most systems also needs screen to be installed with special permissions. That's a security-sensitive setup best left to an administrator.

Hands-On Exercises

EXERCISE 1

Write a short shell script, `start-backup.sh`, that starts a detached, logged screen session called `backup` running a (harmless) command such as `for i in 1 2 3; do date; sleep 20; done`, and keeps the window open afterwards. Then check on it without attaching.

 [View solution](#)

EXERCISE 2

From outside a detached session called `work`, run `ls -l ~` in its window 0, then capture what the window shows into a file and display it — without ever attaching.

 [View solution](#)

EXERCISE 3

Write a `~/screenrc` that turns off the startup message, keeps 5,000 lines of scrollback, shows a status line, and opens two titled windows. Test it with `-c` first, then use copy mode to copy a line from one window and paste it into the other.

 [View solution](#)

CHAPTER 9 QUICK REFERENCE

- `screen -dmS name [cmd]` starts a detached session; wrap the command in `bash -c '...; exec bash'` to keep the window open
- `-X` sends a screen command to a session; `-p` selects the window by number or title
- `-X stuff '$'cmd\n'` types a command into a window — only into one waiting at a prompt
- `-L` (and `-logfile`) or `Ctrl-a H` log output; `Ctrl-a h` or `-X hardcopy` takes a snapshot
- Copy mode: `Ctrl-a [`, move, `Space` to mark start and end, then `Ctrl-a ]` to paste; set `defscrollback`
- `~/screenrc` (or `-c file`) holds settings: `startup_message off`, `defscrollback`, `hardstatus`, startup windows

- `-x` shares your own session; sharing with other users needs `multiuser` / `acladd` and admin setup

CHAPTER 10: CAPSTONE — RUNNING AND RESCUING LONG JOBS OVER SSH

GNU Screen & Linux Job Control

Chapter 10 · Capstone: Running and Rescuing Long Jobs Over SSH

This chapter puts the whole course together in one realistic day of work on a remote server. You'll start a long job safely, lose your connection, find your way back, rescue a job that was started the wrong way, watch everything from outside, and clean up. Each step names the chapter it comes from.

THE SCENARIO

You're migrating a database on a server called `dbhost`. You need to run a long import, keep an eye on disk space, and — as it turns out — rescue a compression job a colleague started straight in their SSH session before they had to leave.

Step 1: Prepare Your Environment (Chapter 9)

A one-off setup that makes every later step easier: a status line, so you always know you're inside screen and which window you're in, and plenty of scrollback.

```
# ~/.screenrc on dbhost
startup_message off
defscrollback 10000
hardstatus alwayslastline "%H | %-w[%n %t]%+w"
```

Step 2: Start the Long Job the Right Way (Chapters 2, 4, 9)

```
laptop$ ssh dbhost
dbhost$ cd ~/migration
dbhost$ screen -S migrate -L          # named session, with logging

# Inside the session, window 0:
Ctrl-a A  import                      # name the window
$ mysql newdb < dump.sql
```

```
# A second window to keep an eye on disk space:
Ctrl-a c
Ctrl-a A    disk
$ watch df -h

Ctrl-a 0                                # back to the import
Ctrl-a _    # tell me when it goes quiet (finished or hung up)
```

Step 3: Lose the Connection (Chapters 1, 7)

Your laptop goes to sleep and the SSH connection drops. Without screen, the terminal hang-up would send `SIGHUP` to the import and it would stop. Here, only the screen *client* is lost: the import runs inside a terminal owned by the screen server, which nothing has hung up.

Step 4: Find Your Way Back (Chapters 2, 3)

```
laptop$ ssh dbhost
dbhost$ screen -ls
There is a screen on:
      41200.migrate  (Attached)          # the server hasn't noticed you left
1 Socket in /run/screen/S-philip.

dbhost$ screen -r migrate
# refused: the session still counts as attached

dbhost$ screen -d -r migrate          # detach the dead client, reattach here
```

You're back in window 0. Scroll up with `Ctrl-a [` to see anything the import printed while you were away, or read `screenlog.0` (Chapter 9).

Step 5: Investigate the Colleague's Job (Chapters 5, 6)

Your colleague messages: they started a large compression job directly in their SSH session, then had to leave. You're both working under the same shared deployment account, so you can see and act on the process. First, find it and check where it lives.

```
$ pgrep -af xz
41877 xz -9 archive.tar

$ pstree -s -p 41877
systemd(1)──sshd(41820)──sshd(41855)──bash(41856)──xz(41877)

$ ps -o pid,stat,TTY,cmd -p 41877
  PID STAT TTY      CMD
41877 R+    pts/3    xz -9 archive.tar
```

The ancestry shows no **SCREEN** process: **xz** is a direct child of an SSH login's shell, on that login's terminal **pts/3**, and in its foreground (**+**). If that SSH connection drops, it dies. Its job number, **%1** in your colleague's shell, means nothing in yours — only the PID does.

Step 6: Rescue It Into Screen (Chapters 7, 8)

You can't press **Ctrl-Z** in someone else's terminal, but you can pull the process into a new window of your own session with **reptyr**.

```
# Check ptrace permission first
$ cat /proc/sys/kernel/yama/ptrace_scope
1                                # restricted: needs sudo

# In a new window of the migrate session
Ctrl-a c
Ctrl-a A    compress
$ sudo reptyr 41877
```

xz now runs in the **compress** window. When your colleague's connection closes, nothing reaches it. A quick **pstree -s -p 41877** would now show your session's **SCREEN** in its ancestry.

IF REPTYR WEREN'T AVAILABLE

The fallback would be to have your colleague, before leaving, press **Ctrl-Z**, run **bg** and **disown %1** (Chapter 7). The job would survive, but nobody could watch it again, only check that the process still exists and that its output file keeps growing.

Step 7: Watch Everything From Outside (Chapters 8, 9)

You detach and go to a meeting, but want to check in from your phone's SSH app without disturbing the layout.

```
# Add one more window without attaching: watch the import's table count
$ screen -S migrate -X screen -t tables watch "mysql -e 'SHOW TABLES' newdb | wc -l"

# Snapshot what the import window is showing right now
$ screen -S migrate -p import -X hardcopy /tmp/import.txt
$ tail /tmp/import.txt

# See everything running in the session
$ pstree -p -a 41200
```

Step 8: Finish and Clean Up (Chapters 2, 3, 4)

```
# The silence monitor from Step 2 reports that window 0 has gone quiet
$ screen -r migrate
# Check the import finished cleanly, then close windows you no longer need
Ctrl-a k                                # in each finished window

# When everything's done: end the session and tidy leftovers
$ screen -S migrate -X quit
$ screen -wipe
$ screen -ls
No Sockets found in /run/screen/S-philip.
```

Where Each Step Came From

CHAPTER	USED IN THIS CAPSTONE
1 — Why Screen?	Why the import survived the dropped connection
2 — Detach & Reattach	Named session; <code>-d -r</code> for the stale "Attached" session
3 — Listing Sessions	<code>screen -ls</code> , the session ID, <code>-X quit</code> and <code>-wipe</code>

CHAPTER	USED IN THIS CAPSTONE
4 — Windows & Regions	Named windows and silence monitoring
5 — Job Control Basics	Why a job number couldn't be used across shells
6 — IDs Compared	<code>pgrep</code> , <code>pstree -s</code> and <code>ps</code> to locate the colleague's job
7 — Surviving Logout	SIGHUP, and the <code>disown</code> fallback
8 — Rescuing Running Jobs	<code>reptyr</code> , <code>ptrace_scope</code> , and adding a window with <code>-X screen</code>
9 — Automating Screen	<code>.screenrc</code> , <code>-L</code> logging, <code>hardcopy</code> , scrollbar

What This Course Doesn't Cover

- **tmux** in depth — the same ideas with different keys and a more modern design (Chapter 1).
- **Multi-user sessions** between different accounts, which need administrator setup (Chapter 9).
- **Permanent services** — anything that should start at boot and restart when it fails belongs in systemd (see the systemd in Depth course).
- **Job schedulers** such as `cron`, `at` and systemd timers, for jobs that should run at a set time without you.

Hands-On Exercises

EXERCISE 1

Recreate Steps 2–4 on your own machine, using a harmless stand-in for the import (such as a loop that prints the date every 10 seconds). Simulate the dropped connection by closing the terminal window, and recover the session. Note the exact message `screen -ls` gives you.

 [View solution](#)

EXERCISE 2

Recreate Steps 5–6: in one terminal start `yes > /dev/null` outside screen; from a second terminal, find it, prove it's not in screen, and move it into a screen window. Then prove it *is* in screen.

 [View solution](#)

EXERCISE 3

Write a short checklist, in your own words, for starting any job on a remote server that might take more than ten minutes. It should cover starting, checking progress, surviving a disconnection, and cleaning up.

 [View solution](#)

COURSE QUICK REFERENCE

- **Start:** `screen -S name` (add `-L` to log); `screen -dmS name cmd` to start detached
- **Leave and return:** `Ctrl-a d`; `screen -r name`; `screen -d -r name` if it's still marked Attached
- **List and identify:** `screen -ls`; IDs are `PID.name`; `$STY` and `$WINDOW` inside
- **Windows:** `Ctrl-a c`, `n/p`, `"`, `A`, `k`; splits `S`, `|`, `Tab`; monitoring `M` and `_`
- **Job control:** `&`, `Ctrl-Z`, `jobs -l`, `fg`, `bg`, `kill %n`, `$!`
- **Find processes:** `pgrep -af`, `pstree -s -p PID`, `pstree -p -a SCREEN_PID`
- **Survive logout:** `nohup`, `bg` then `disown`, `setsid`; check systemd's `KillUserProcesses`
- **Rescue into screen:** `reptyr PID` inside a window (check `ptrace_scope`)
- **Automate:** `-X screen`, `-p`, `-X stuff '$cmd\n'`, `-X hardcopy`, `~/.screenrc`
- **Clean up:** `screen -S name -X quit`, `screen -wipe`

Course Complete

This completes **GNU Screen & Linux Job Control**, 10/10 chapters.