

Name: Arsalaan Hasan Shaikh, Falak Nishant Chalasany	Roll No.: 9, 16
Class: TY-AIDS	Batch: A
Academic Year: 2026-27	

Practical No. 10: Microproject Activity

Develop any ONE of the following microprojects in a group of 2–3 students:

1. Page Replacement Algorithms: Prepare a comparative statement to calculate and analyze the number of page faults for a given page reference string using different page replacement algorithms.
2. Linux Shell Scripting: Develop a help guide using shell scripts to demonstrate and explain commonly used Linux commands.
3. CPU Scheduling Algorithms: Prepare a comparative chart to calculate and analyze the average waiting time and turnaround time of processes using different CPU scheduling algorithms.

Report 1

A Comparative Study of the FIFO, LRU and Optimal Page Replacement Algorithms

Abstract

We implemented FIFO, LRU and Optimal (OPT) page replacement as a Python program and a web page, and counted page faults on two reference strings with one to five frames. On a standard 18-reference string with three frames, FIFO produced 13 faults, LRU 12 and OPT 9. OPT was never beaten. LRU was not always better than FIFO: it lost at two frames on the first string and at three frames on the second. On a 12-reference string FIFO produced 9 faults with three frames and 10 with four, which is Belady's anomaly. LRU and OPT did not show it.

1. Introduction

A process may use more pages than the machine has frames. When it asks for a page that is not in memory, a page fault occurs and the operating system must load the page from disk. If every frame is occupied, one resident page must be evicted first. A disk read costs far more than a memory access, so a replacement policy that evicts the wrong page slows every program that follows.

This report compares three policies on identical input. The aim is to measure how many faults each one causes, to see how that changes as frames are added, and to build a tool that lets a student watch the frames change step by step.

2. Background

FIFO evicts the page that has been resident longest. It needs only a queue, and a hit changes nothing. LRU evicts the page whose last use is oldest, on the assumption that recent use predicts near-future use. It must update its ordering on every reference, which in hardware needs counters or a stack. OPT evicts the page whose next use lies furthest ahead, or that is never used again. It needs the future of the reference string, so no real system can run it. It serves as a lower bound against which practical policies are judged (Belady, 1966).

A policy is called a stack algorithm if the set of pages held with n frames is always a subset of the set held with $n+1$ frames. LRU and OPT have this property, so adding frames cannot increase their faults (Mattson et al., 1970). FIFO does not, and for some strings more frames cause more faults (Belady, Nelson and Shedler, 1969).

3. Method and Implementation

The program `pr.py` simulates memory as a fixed list of frame slots. A page keeps its slot until it is evicted, and the new page takes that same slot, which matches how the tables are drawn by hand. A queue holds the eviction order. For FIFO the queue is ordered by arrival and is not

touched on a hit. For LRU a hit moves the page to the back of the queue. For OPT the victim is the resident page with the furthest next use, and a page never used again is chosen first. Ties go to the page that entered earliest.

The output is a horizontal grid per algorithm. Pages run across the top, frames run upward from Frame 1 at the bottom, and a row below the grid marks each step f for a fault or h for a hit. A closing table gives the fault and hit totals. The web version uses the same logic in JavaScript and draws the same grids.

Two strings were used. String L is the 18-reference string from Experiments 8 and 9: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7. String B is the 12-reference string 1 2 3 4 1 2 5 1 2 3 4 5, which is known to expose Belady's anomaly. The Python program and the web page gave identical counts, and the FIFO and LRU totals for String L at three frames (13 and 12) match the earlier lab outputs.

4. Results

Table 1. String L, three frames

Algorithm	Faults (f)	Hits (h)	Hit ratio
FIFO	13	5	27.8%
LRU	12	6	33.3%
OPT	9	9	50.0%

Table 2. String L, faults by number of frames

Frames	FIFO	LRU	OPT
1	18	18	18
2	14	15	12
3	13	12	9
4	10	8	8
5	7	7	7

Table 3. String B, faults by number of frames

Frames	FIFO	LRU	OPT
1	12	12	12
2	12	12	9
3	9	10	7
4	10	8	6
5	5	5	5

With one frame every algorithm faults on every change of page, so the three agree. At the other end, String L has six distinct pages, so no policy can fault fewer than six times. At five frames all three need seven, one more than that floor. Between those extremes OPT is lowest, or tied for

lowest, in every row. LRU matches OPT at four frames on String L (8 and 8) but is three faults behind at three frames.

Table 3 shows the anomaly. FIFO rises from 9 faults at three frames to 10 at four. LRU falls from 10 to 8 and OPT from 7 to 6. The table also shows that LRU is not always ahead of FIFO: at three frames on String B it has 10 faults to FIFO's 9. On String L at two frames LRU has 15 to FIFO's 14.

5. Discussion

The ordering OPT, then LRU, then FIFO holds on String L at three and four frames, which is the pattern expected for programs with locality. String B cycles through pages in a way that defeats LRU at small sizes: just before a page is needed again, it has become the least recently used and is evicted. FIFO happens to avoid some of those evictions. This is a property of the string and not a general advantage of FIFO, but it shows that the ranking depends on the access pattern.

The cost side matters as much. FIFO needs a queue. Exact LRU needs an update on every memory reference, which is why operating systems use cheaper approximations such as the clock algorithm. The gap between LRU and OPT on String L at three frames, 12 against 9, is the most that better prediction could recover for that input.

6. Limitations

Two short strings are enough to show the effects above but not to rank the policies in general. The strings are not traces from real programs. The simulation gives every fault the same cost and ignores dirty pages, which cost a write-back. Frame counts are fixed per run, and the clock and second-chance policies are not covered.

7. Conclusion

OPT gave the fewest faults in every test and LRU usually came second, but LRU lost to FIFO in two cases. FIFO showed Belady's anomaly on String B. The program and the web page let a reader reproduce every number in this report by entering the strings and frame counts given above.

References

- Belady, L. A. (1966). A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2), 78-101.
- Belady, L. A., Nelson, R. A., & Shedler, G. S. (1969). An anomaly in space-time characteristics of certain programs running in a paging machine. *Communications of the ACM*, 12(6), 349-353.
- Mattson, R. L., Gecsei, J., Slutz, D. R., & Traiger, I. L. (1970). Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2), 78-117.
- Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley.
- Tanenbaum, A. S., & Bos, H. (2014). *Modern Operating Systems* (4th ed.). Pearson.

Report 2

The Command Thesaurus: A Shell Script and Web Guide to Common Linux Commands

Abstract

Manual pages assume the reader already knows the name of the command. We built a help guide that works the other way: the reader says what they want to do and the guide names the command. It exists as a bash script, `guide.sh`, that runs examples in a disposable folder, and as a web book with a searchable index and synonym matching. It covers 39 entries in 12 groups, drawn from the commands practised in an online Linux course. All 26 examples in the script ran without error under bash on Linux.

1. Introduction

A beginner at the Linux shell usually knows the task and not the tool. Someone who wants to delete a file has to know that the command is `rm` before a manual page can help. This project builds a guide indexed by intent, so that searching for delete, copy or stop leads to `rm`, `cp` or `kill`.

The project had two requirements: shell scripts that demonstrate and explain common commands, and a design a first-time user can pick up without instruction.

2. Background

The commands were selected from the practical work of a 107-lecture beginner course on terminal use and shell scripting (Schürmann, n.d.), which covers navigation, file handling, redirection, permissions, processes, networking, regular expressions and Bash scripting. Command behaviour was checked against the GNU Bash Reference Manual (Ramey & Fox, n.d.) and Shotts (2019).

A thesaurus relates words by meaning. The guide applies that idea to commands: each entry carries a list of plain words that describe it, and related entries are linked, so that arithmetic leads to the command that formats the answer, and appending to a file leads to the command that prints text.

3. Design and Implementation

Every entry has the same parts: a one-line meaning, the exact text to type, the output to expect, other words for it, and related commands. The index is grouped by task and not by alphabet. A search box filters the index by command name, description and synonyms together, and a query of several words must match all of them.

The web book presents this as a closed book that opens to an index page on the left. Choosing a command turns the right-hand page. The styling is deliberately plain: a serif face, three colours

and no decoration beyond the page turn. A small hidden extra lives in the text for readers who click on a certain mythical animal.

The script `guide.sh` holds 26 examples in an array. It creates a temporary folder with a sample file, lists matching entries for a word given on the command line or typed at the prompt, and runs the example for a typed number. A trap removes the folder on exit, so `rm`, `mv`, `chmod` and `kill` examples cannot affect the user's own files. The examples tell a small story: a file is created with `>`, a line about unicorns is added with `>>`, and `sed` changes Scotland to Narnia.

4. Results

Table 1. Coverage of the web book

Group	Commands	Entries
Look around	<code>pwd</code> , <code>ls</code> , <code>cd</code>	3
Make and move	<code>mkdir</code> , <code>touch</code> , <code>cp</code> , <code>mv</code>	4
Remove	<code>rm</code>	1
Edit files	<code>echo</code> , <code>></code> , <code>>></code> , <code>nano</code> , <code>heredoc</code> , <code>tee</code> , <code>sed</code>	7
Read and search	<code>cat</code> , <code>grep</code> , <code>find</code> , <code>head</code> , <code>tail</code> , <code>wc</code>	6
Numbers and types	<code>\$(())</code> , <code>bc</code> , <code>printf</code>	3
Permissions	<code>chmod</code>	1
Processes	<code>ps</code> , <code>kill</code>	2
Network	<code>ping</code>	1
Chain and shortcut	<code> </code> , <code>alias</code> , <code>ln</code>	3
Scripting	<code>vars</code> , <code>if</code> , <code>for</code> , <code>while</code> , <code>case</code> , <code>fn</code>	6
Projects	<code>mover</code> , <code>pomodoro</code>	2
Total		39

All 26 examples in `guide.sh` were run in sequence on Linux under `bash`, and none printed an error. The arithmetic example printed 42, the decimal example 3.33, and the formatted example 42 unicorn 42.0. Searching the script for unicorn returned the three entries that mention it, and searching for delete returned `rm`.

5. Separate text guide

Look Around

- `pwd`: Prints the folder you are currently in.
 - Example: `pwd`
- `ls`: Lists the files and directories inside a folder.
 - Example: `ls -lah`
- `cd`: Moves you into another folder.
 - Example: `cd ~/laboratorio_linux`

Make, Move, and Remove

- mkdir: Creates a new folder.
 - Example: `mkdir -p dir1/sub_dir`
- touch: Creates a new empty file.
 - Example: `touch archivo1.txt notas.md`
- cp: Copies a file.
 - Example: `cp archivo1.txt archivo_copia.txt`
- mv: Moves a file or renames it.
 - Example: `mv archivo2.txt archivo_movido.txt`
- rm: Deletes a file completely.
 - Example: `rm archivo_copia.txt`

Edit Files

- echo: Prints a line of text on the screen.
 - Example: `echo "Unicorns are the national animal of Scotland"`
- >: Writes text into a file, overwriting existing content.
 - Example: `echo "Welcome to the zoo" > zoo.txt`
- >>: Adds a line of text to the end of a file while keeping the rest.
 - Example: `echo "Unicorns are the national animal of Scotland" >> zoo.txt`
- nano: Opens a text editor directly in the terminal.
 - Example: `nano zoo.txt`
- heredoc: Writes multiple lines into a file in a single command block.
 - Example: `cat << 'EOF' > zoo.txt`
- tee: Displays text on the screen and saves it to a file simultaneously.
 - Example: `echo "Hello" | tee zoo.txt`
- sed: Finds a specific word in a file and swaps it for a different word.
 - Example: `sed -i 's/Scotland/Narnia/' zoo.txt`

Read and Search

- cat: Prints the full contents of a file on the screen.
 - Example: `cat editor.txt`
- grep: Finds specific lines in a file that contain a given word.
 - Example: `grep "palabra" editor.txt`
- find: Locates files based on their name or type.
 - Example: `find . -type f -name "*.txt"`
- head: Displays only the first lines of a file.
 - Example: `head -n 2 editor.txt`
- tail: Displays only the final lines of a file.
 - Example: `tail -n 1 editor.txt`
- wc: Counts the total lines, words, or characters provided.
 - Example: `cat editor.txt | grep "Linea" | wc -l`

Numbers and Types

- `$(( ))` (math): Performs arithmetic on whole numbers.
 - Example: `echo $((15 + 27))`
- `bc`: Performs arithmetic operations using decimal numbers.
 - Example: `echo "scale=2; 10 / 3" | bc`
- `printf`: Prints values formatted into a chosen output type and shape.
 - Example: `printf "%d %s %.1f\n" 42 unicorn 42`

Permissions, Processes & Network

- `chmod`: Modifies who is permitted to read, write, or run a file.
 - Example: `chmod 644 editor.txt`
- `ps`: Lists the current running programs on the system.
 - Example: `ps aux | head -n 5`
- `kill`: Stops a currently running process.
 - Example: `kill -9 $!`
- `ping`: Tests if another machine on the network answers.
 - Example: `ping -c 2 127.0.0.1`

Chains, Shortcuts & Scripting

- `|` (pipe): Takes the output of one command and sends it directly into the next.
 - Example: `cat editor.txt | grep "Linea" | wc -l`
- `alias`: Assigns a short, custom name to a longer command sequence.
 - Example: `alias ll='ls -la'`
- `ln`: Creates a link to a file, using `-s` for a symbolic link.
 - Example: `ln -s editor.txt enlace_simbolico.txt`
- `vars`: Stores a specific value assigned to a name.
 - Example: `NOMBRE="Estudiante"`
- `if`: Evaluates a condition to decide if a command should run.
 - Example: `if [ $N -gt 10 ]; then echo mayor; else echo menor; fi`
- `for`: Loops a command to repeat for every item in a list.
 - Example: `for i in {1..3}; do echo "Iteracion $i"; done`
- `while`: Repeats a command continuously as long as its test condition passes.
 - Example: `while [ $c -le 2 ]; do echo "Contador: $c"; ((c++)); done`
- `case`: Matches a value to select one specific action out of several options.
 - Example: `case $O in inicio) echo iniciado;; parar) echo detenido;; *) echo no valida;; esac`
- `fn`: Assigns a name to a block of commands so they can be reused easily.
 - Example: `saludar() { echo "Saludos para $1"; }`

Integrated Script Projects

- `mover`: A script designed to organize files by automatically sorting them into folders based on their file extensions.
- `pomodoro`: A script that functions as a timer, running a short focus block followed by a rest block.

6. Discussion

Searching by intent removes the first obstacle for a beginner, which is not knowing a name to search for. The synonym lists carry most of that value, and they are also the weakest part: they were written by hand and will miss words a reader might choose. Linking related commands helps readers move from one task to the next, as from arithmetic to output formatting.

Running examples in a throwaway folder lets a learner try destructive commands safely, at the cost of showing only small demonstrations. The script needs bash 4 or newer, so it does not work with the default shell on macOS, and the web book is the fallback there.

7. Limitations

The guide covers 39 entries, not the whole shell. Synonyms are English only, although the source course is in Spanish. Commands needing a terminal editor, administrator rights or a network (nano, sudo, ssh) are described but not run by the script. The ping entry in the web book shows typical output and is not run by guide.sh. No usability testing with new users was carried out.

8. Conclusion

The guide shows that a command reference indexed by intent is practical to build with plain shell and web tools. The next steps are more entries, synonyms gathered from real learners' searches, and a short trial with first-year students.

References

Ramey, C., & Fox, B. (n.d.). GNU Bash Reference Manual. Free Software Foundation.

Schürmann, N. (n.d.). Ultimate Linux: de uso de la terminal hasta shell scripting [Online course]. Udemy.

Shotts, W. (2019). The Linux Command Line (5th ed.). No Starch Press.

Report 3

A Comparative Study of CPU Scheduling Algorithms by Turnaround and Waiting Time

Abstract

We implemented six scheduling policies (FCFS, non-preemptive and preemptive SJF, non-preemptive and preemptive priority, and Round Robin) in a Python program and a web page, and compared average turnaround time (TAT) and average waiting time (WT) on two process sets. SRTF gave the lowest averages on both sets: WT 6.50 against 8.75 for FCFS on the first set, and 4.50 against 18.75 on the second, where one long process arrives first. Round Robin was the worst on the first set and second best on the second, so its result depends on the quantum as well as the workload.

1. Introduction

When several processes are ready, the CPU scheduler decides which runs next. The choice changes how long each process waits and how long it takes to finish. This report measures that effect for six policies on identical workloads, using two standard metrics, and describes the tool built to run the comparison.

2. Background

For a process with arrival time AT and burst time BT that completes at time CT, turnaround time is $TAT = CT - AT$ and waiting time is $WT = TAT - BT$. FCFS runs processes in arrival order. SJF runs the ready process with the shortest burst. Its preemptive form, SRTF, switches whenever a newly arrived process has less work remaining than the one running. Priority scheduling runs the ready process with the best priority, and here a lower number means a higher priority. It also has a preemptive form. Round Robin gives each ready process a fixed quantum in turn and returns an unfinished process to the back of the queue.

SJF and SRTF minimise average waiting time when burst lengths are known (Silberschatz et al., 2018), but real systems must estimate them. Priority scheduling can starve a low-priority process indefinitely, and aging is the usual remedy.

3. Method and Implementation

The program `cp.py` reads each process as an id, an arrival time, a burst time and a priority, all whole numbers. The user picks one algorithm from a menu, or picks All. A single choice prints a one-line timeline, a table with CT, TAT and WT per process, and the averages. All prints only the averages. Round Robin asks for a quantum. The web version offers the same menu and the same output.

Preemptive modes advance one time unit at a time and re-pick the process after each unit. Non-preemptive modes run the chosen process to completion. Ties are broken by earlier arrival.

In Round Robin, a process that arrives while another is running joins the queue before the preempted process does. Context-switch time is taken as zero. The averages for the first set under SJF were checked by hand (waiting times 0, 7, 9 and 15).

Table 1. Process sets

Set	Process	Arrival	Burst	Priority
A	a	0	8	2
A	b	1	4	1
A	c	2	9	4
A	d	3	5	3
B	p1	0	24	3
B	p2	1	3	1
B	p3	2	3	2
B	p4	3	6	4

4. Results

Table 2. Average TAT and WT by algorithm

Algorithm	A: TAT	A: WT	B: TAT	B: WT
FCFS	15.25	8.75	27.75	18.75
SJF	14.25	7.75	27.75	18.75
SRTF	13.00	6.50	13.50	4.50
Priority	14.25	7.75	27.75	18.75
Priority (preemptive)	13.50	7.00	17.75	8.75
Round Robin (q=3)	20.00	13.50	15.75	6.75

Table 3. Round Robin by quantum

Quantum	A: TAT	A: WT	B: TAT	B: WT
1	19.00	12.50	17.50	8.50
2	19.25	12.75	18.25	9.25
3	20.00	13.50	15.75	6.75
4	18.25	11.75	16.75	7.75
5	17.50	11.00	17.75	8.75
8	16.25	9.75	18.75	9.75

On Set A, SRTF is best, with TAT 13.00 and WT 6.50. Preemptive priority is next. SJF and non-preemptive priority give the same figures because both happen to run b, d, c after a. FCFS follows, and Round Robin with quantum 3 is last at 20.00 and 13.50.

Set B is built to show the convoy effect. Process p1 needs 24 units and arrives first. Under FCFS, SJF and non-preemptive priority it runs alone while three short processes wait behind it, giving WT 18.75 for all three. SRTF reduces this to 4.5, because the short processes preempt p1 as soon as they arrive. Round Robin, which also cuts p1 short, reaches 6.75 at quantum 3.

Table 3 shows that Round Robin does not improve steadily with quantum size. On Set A it improves from 3 upward, reaching WT 9.75 at quantum 8. On Set B the best value is at quantum 3, and larger quanta move back toward FCFS.

5. Discussion

Preemption was the most useful property on both sets, and the benefit was largest when a long process arrived first. Knowing burst times in advance is what makes SRTF work, and the program is given them directly. A real scheduler would have to predict them, so these figures are a best case. Round Robin trades average time for responsiveness: every process gets the CPU within one round, which the averages here do not reward.

Priority scheduling was only as good as the priorities supplied. On Set B the process with the best priority had a short burst, so preemptive priority did well. If the priorities had favoured p1, it would not have. Nothing in the program ages priorities, so a steady stream of high-priority work could hold back a low-priority process forever.

6. Limitations

Two small process sets and whole-number times limit what the numbers can support. Context switches cost nothing in the model, which flatters preemptive policies and small quanta. Processes have only CPU bursts, with no input or output waits. Multilevel queues and multiprocessor scheduling are not covered.

7. Conclusion

SRTF gave the lowest average waiting and turnaround time on both sets, FCFS suffered when a long process came first, and Round Robin depended on the quantum. The program and web page reproduce every figure above from the process sets in Table 1.

References

- Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating System Concepts (10th ed.). Wiley.
- Tanenbaum, A. S., & Bos, H. (2014). Modern Operating Systems (4th ed.). Pearson.