Assignment-2

Shivansh Garg (2024MCS2450) Rohit Patidar (2024JCS2042)

April 2025

Introduction

This document presents an in-depth explanation of the control flow mechanisms and scheduling strategy implemented in a custom shell environment. It focuses on how different key combinations (`Ctrl+C`, `Ctrl+B`, `Ctrl+F`, and `Ctrl+G`) trigger specific signals and the way these signals are handled within the shell and kernel space.

The first part provides a detailed step-by-step breakdown of the signal handling mechanism, illustrating how each key combination affects process states and scheduling behavior. This includes the termination signal (`SIGINT`), background suspension (`SIGBG`), foreground resumption (`SIGFG`), and custom user-defined signal (`SIGCUSTOM`).

The second part of the document explains the dynamic priority scheduling algorithm. The scheduling policy uses a weighted function of CPU usage and wait time, influenced by two tunable parameters: α and β. These parameters control how a process's priority evolves over time, enabling fine-grained control over system responsiveness and fairness.

Together, these mechanisms form a responsive and balanced process management system, suitable for managing multiple types of workloads in a multi-process environment.

Individual Control Flow Explanation

This section explains the control flow for each signal triggered by the key combinations Ctrl+C, Ctrl+B, Ctrl+F, and Ctrl+G.

1. Ctrl+C → SIGINT (Terminate)

1. **User Input:** User presses Ctrl+C.
2. **Shell Input Handling:**
   - The shell detects Ctrl+C and interprets it as a command to terminate processes.
   - It calls `send_signal_to_all(SIGINT);` to broadcast the termination signal.
3. **Inside send_signal_to_all():**
   - Iterates through all processes.
   - For each process (except shell and init):
     - Sets `p->killed = 1`
     - If state is `SLEEPING`, `SUSPENDED`, etc., sets `p->state = RUNNABLE` so that the process wakes up and gets killed in the next scheduler cycle.
4. **Result:** Processes are terminated.

```
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ test
Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
Hello 5
Ctrl-C is detected by xv6
PID: 3
TAT: 569
WT: 0
RT: 0
#CS: 0
$
```

Figure 1: Ctrl + C

2. Ctrl+B → SIGBG (Send to Background)

1. **User Input:** User presses Ctrl+B.
2. **Shell Input Handling:**
   - Shell calls `send_signal_to_all(SIGBG);` to suspend all other processes.
3. **Inside send_signal_to_all():**
   - Iterates over processes (except shell and init).
   - For eligible processes in RUNNING, RUNNABLE, etc.:
     - Sets `p->state = SUSPENDED`
     - Sets `p->suspended = 1`
     - Reassigns their parent to `initproc` (if pid > 2)
4. **Result:** All active processes are suspended and sent to background.

Figure 2: Ctrl + B

3. Ctrl+F → SIGFG (Bring to Foreground)

1. **User Input:** User presses Ctrl+F.
2. **Shell Input Handling:**
   - Shell calls `send_signal_to_all(SIGFG);` to resume suspended processes.
3. **Inside send_signal_to_all():**
   - Iterates over all processes.
   - For each process where `p->state == SUSPENDED`:
     - Sets `p->state = RUNNABLE`
     - Sets `p->suspended = 0`
4. **Result:** Suspended processes resume execution.

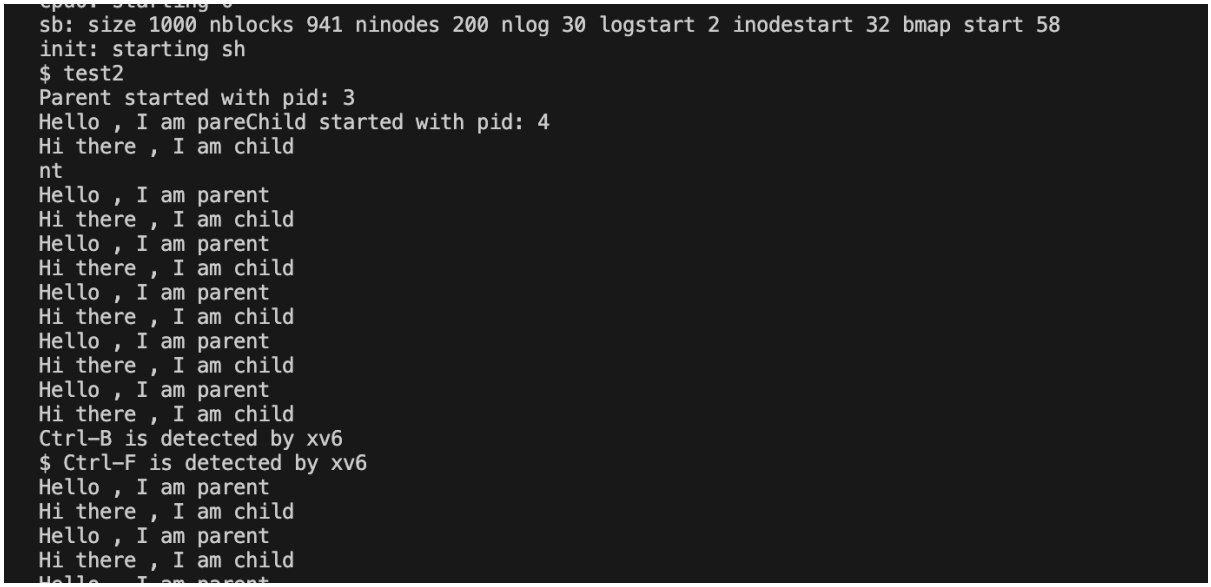

Figure 3: Ctrl+B and Ctrl + F

4. Ctrl+G → SIGCUSTOM (Custom Signal)

1. **User Input:** User presses Ctrl+G.
2. **Shell Input Handling:**
   - Shell calls `send_signal_to_all(SIGCUSTOM);` to notify all processes.
3. **Inside send_signal_to_all():**
   - Iterates over processes.
   - If a process has a custom handler (`p->signal_handler != 0`):
     - Sets `p->pending_signal = SIGCUSTOM`
     - If `p->state == SLEEPING`, wakes it up by setting `RUNNABLE`
4. **Result:** When scheduled, process checks for `pending_signal` and invokes its handler.

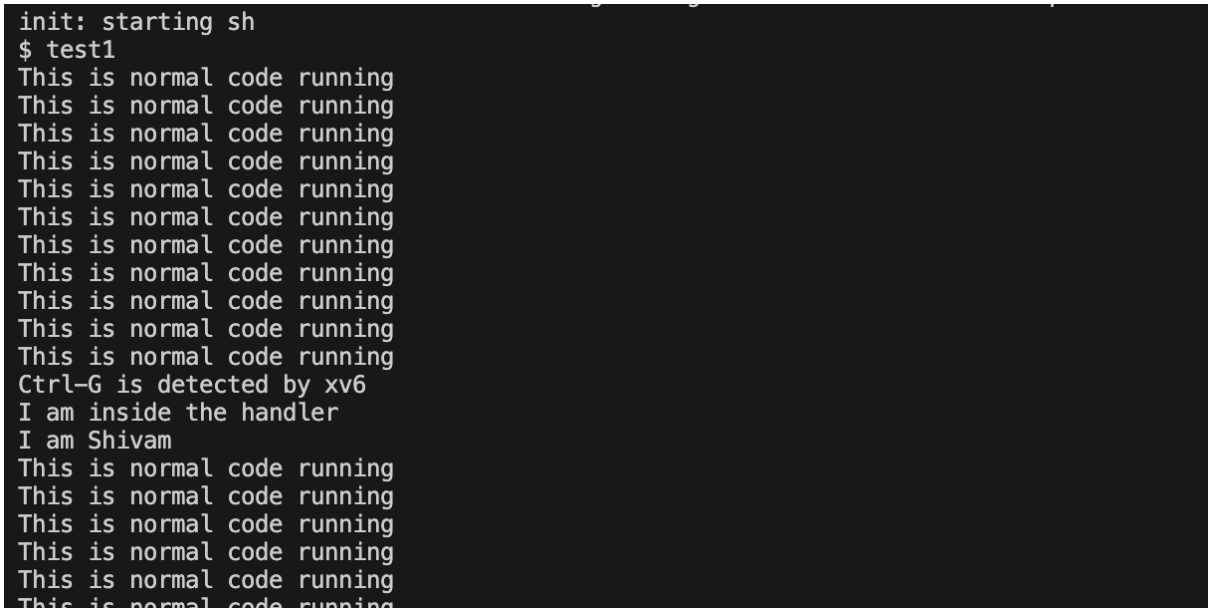

Figure 4: Ctrl+G

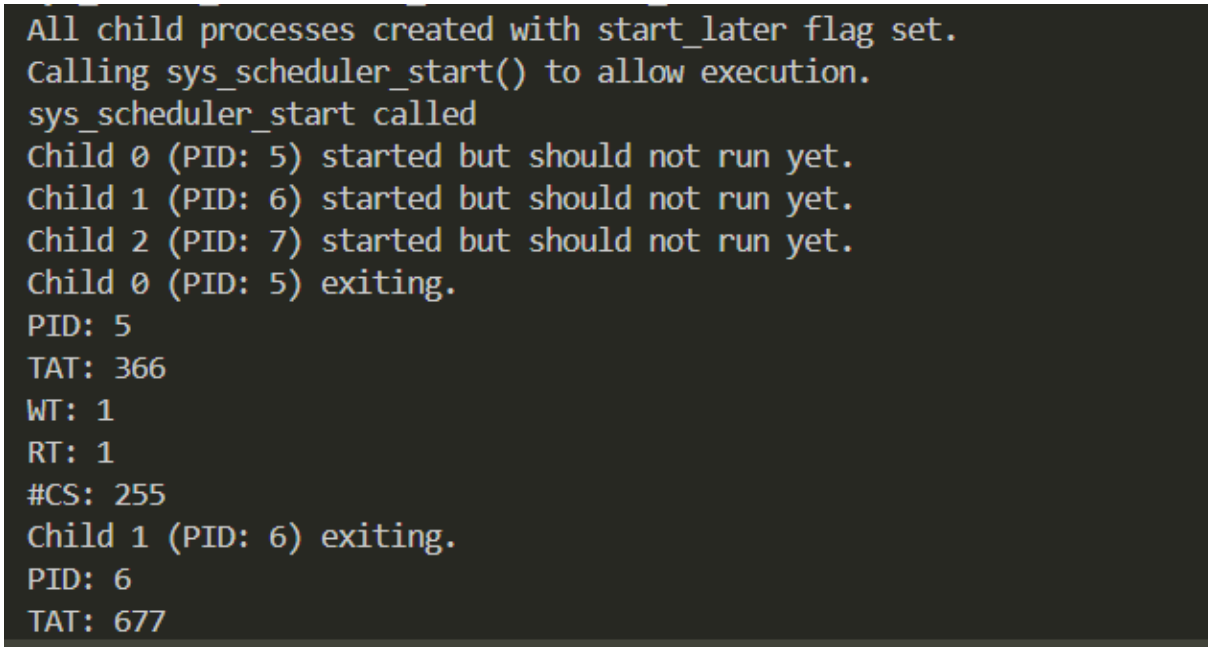

Figure 5: Scheduler

Dynamic Priority Scheduling: Effect of α and β Parameters

The scheduling algorithm implemented uses a dynamic priority function to determine which process should be scheduled next. The priority $\pi_i(t)$ of process P_i at time t is defined as:

$$\pi_i(t) = \pi_i(0) - \alpha \cdot C_i(t) + \beta \cdot W_i(t)$$

- $\pi_i(0)$: Initial priority of process P_i (defined in `Makefile`)
- $C_i(t)$: Total CPU time consumed by P_i up to time t
- $W_i(t)$: Waiting time of P_i since its creation
- α, β: Tunable parameters to control priority decay and boost (defined in `Makefile`)

Interpretation of Parameters

- **Parameter** α: Controls how quickly a process loses priority based on its CPU usage.
  - A higher α causes CPU-bound processes to lose priority faster.
  - Helps prevent long-running CPU-intensive tasks from hogging the CPU.
- **Parameter** β: Controls how much waiting time contributes to the priority.
  - A higher β rewards processes that have waited longer.
  - Helps I/O-bound and short processes gain CPU access sooner.

Effect on Different Process Types

Process Type	**High α, Low β**	**Low α, High β**
CPU-bound	Quickly penalized due to high CPU usage. Low chances of frequent scheduling.	Deprioritized slowly. May hog the CPU.
I/O-bound	May starve if β is too low.	Frequently scheduled due to accumulated wait time.
Short jobs	May not get enough priority once CPU usage increases.	Scheduled quickly, finish early.

Table 1: Effect of α and β on process behavior

Experimental Observation

In the profiling experiments conducted:

- With higher α (e.g., 3) and moderate β (e.g., 2):
  - CPU-bound jobs were penalized after initial execution.
  - Short/I/O-bound jobs were scheduled more frequently and completed quickly.
  - Fairness was observed with no starvation.
- With lower α (e.g., 1) and higher β (e.g., 4):
  - Even minor waiting resulted in a large priority boost.
  - I/O-bound processes preempted CPU-bound ones very frequently.
  - Potential starvation of long CPU-intensive jobs was observed.
- With $\alpha = 0$ and $\beta = 0$:
  - Priority remained static as $\pi_i(t) = \pi_i(0)$.
  - No dynamic adjustment based on CPU usage or waiting time.
  - Scheduling became unfair over time as processes with initially higher priority continued to dominate CPU access.
  - Both CPU-bound and I/O-bound processes risked starvation if they started with lower priority.

Conclusion

The α and β parameters significantly impact the system's responsiveness and fairness:

- To favor interactive/I/O-bound jobs and prevent starvation, use a higher β.
- To prevent long-running CPU-bound tasks from dominating, use a higher α.
- With $\alpha = 0$ and $\beta = 0$, the scheduler becomes static, resulting in potential long-term unfairness.
- An optimal balance ensures all processes get CPU access proportionately based on their needs.
- To favor **interactive/I/O-bound jobs** and prevent starvation, use a **higher** β.
- To prevent **long-running CPU-bound tasks from dominating**, use a **higher** α.
- An optimal balance ensures all processes get CPU access proportionately based on their needs.