**Process Management Tool**



**Fall 2024**

**System Programming Lab**

Submitted by:

Name : **Hassan Zaib Jadoon, Ahsan Raza, Mutahhar Fayyaz**

Reg no**. : 22PWCSE2144, 22PWCSE2099, 22PWCSE2176**

ClassSection **: A**

Submitted to:

**Engr. Abdullah Hamid**

**Department of Computer Systems Engineering**

**University of Engineering and Technology, Peshawar**

**Project Report: Process Management Tool**

**1. Introduction**

* The **Process Management Tool** is a command-line utility designed to monitor and manage system processes on a Linux-based operating system. It provides a user-friendly interface for listing active processes, terminating processes, analyzing system load, tracking specific processes, and starting new processes. The program is written in C and leverages system calls and Linux's /proc filesystem to gather and display process-related information.

**2. Features**

The program offers the following features:

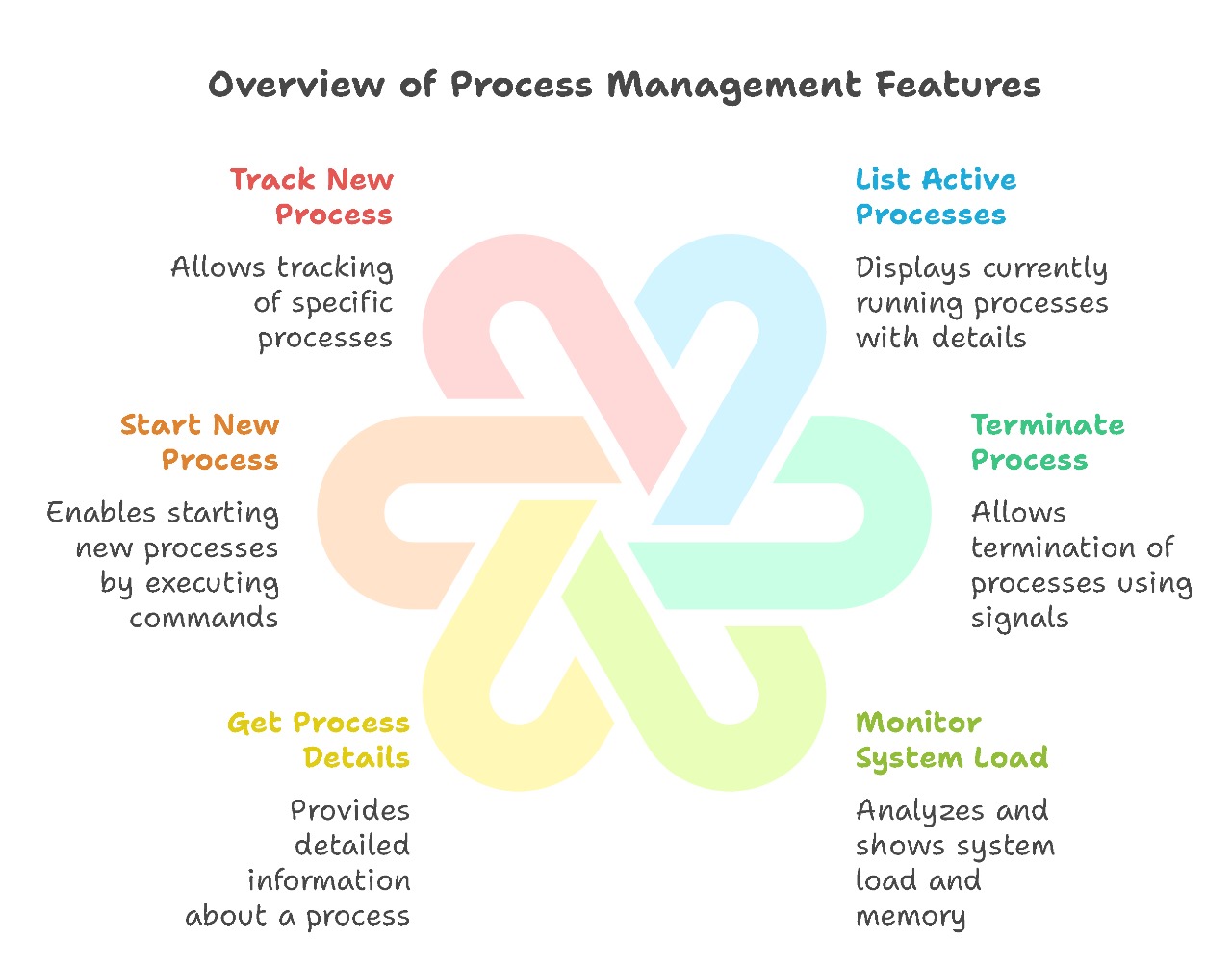
1. **List Active Processes**: Displays a list of all active processes with details such as PID, user, state, and command.
2. **Terminate Process**: Allows the user to terminate a process using either SIGTERM or SIGKILL.
3. **Monitor System Load**: Provides an analysis of system load averages and memory usage.
4. **Get Process Details**: Displays detailed information about a specific process, including its state, memory usage, and command.
5. **Start New Process**: Executes a user-specified command as a new process.
6. **Track Process**: Tracks specific processes and displays their runtime and state.
7. **Show Tracked Processes**: Lists all currently tracked processes and their status.

Figure 1: Overview of Features of Process Management Tools

**3. Technical Details**

**3.1 Data Structures**

ProcessInfo **Structure:**

1. typedef struct {

2. pid\_t pid;

3. char name[256];

4. char user[256];

5. long memory;

6. char state;

7. time\_t start\_time;

8. } ProcessInfo;

This structure stores information about tracked processes, including PID, name, user, memory usage, state, and start time.

**3.2 Key Functions**

list\_processes()

* Lists all active processes by reading the /proc directory.
* Displays PID, user, state, and command for each process.

terminate\_process(pid\_t pid, const char\* signal\_type)

* Terminates a process using either SIGTERM or SIGKILL.

analyze\_system\_load()

* Reads /proc/loadavg and /proc/meminfo to display system load averages and memory usage.

get\_process\_details(pid\_t pid)

* Retrieves detailed information about a specific process from /proc/[pid]/status and /proc/[pid]/cmdline.

start\_process(const char\* command)

* Executes a user-specified command using fork() and execvp().

track\_process(pid\_t pid)

* Tracks a process by storing its details in the tracked\_processes array.

display\_tracked\_processes()

* Displays the status of all tracked processes, including their runtime and current state.

**4. Implementation Details**

**4.1 Process Listing**

* The program reads the **/proc** directory, which contains information about all running processes.
* For each process, it reads the /proc/[pid]/status file to extract details such as PID, state, and memory usage.

**4.2 Process Termination**

* The **kill()** system call is used to send either SIGTERM or SIGKILL to the specified process.

**4.3 System Load Analysis**

* The /proc/loadavg file provides system load averages for the past 1, 5, and 15 minutes.
* The /proc/meminfo file provides detailed memory usage information.

**4.4 Process Tracking**

* The program maintains an array of ProcessInfo structures to store details of tracked processes.
* It periodically checks the status of tracked processes by verifying the existence of the /proc/[pid] directory.

**4.5 User Interface**

* The program uses ANSI color codes to enhance the user interface.
* A menu-driven interface allows users to interact with the program.

**5. Code Structure**

**Header Files**

Standard C libraries and system headers are included for functionality such as file I/O, process management, and signal handling.

**Global Variables**

* tracked\_processes[MAX\_PROCESS\_COUNT]: Array to store tracked processes.
* tracked\_count: Counter for the number of tracked processes.

**Main Function**

* Displays a banner and menu.
* Handles user input and invokes the appropriate functions.

**6. Sample Output**

**List Active Processes:**

ACTIVE PROCESSES:  
PID USER STATE COMMAND  
----------------------------------------  
1 root S init  
2 root S kthreadd

**Terminate Process:**

Enter PID **to** terminate: 1234  
Enter **signal** (SIGTERM/SIGKILL): SIGKILL  
**Process** 1234 terminated successfully **with** SIGKILL

**Monitor System Load:**

SYSTEM LOAD ANALYSIS:  
Load Averages: 0.25 (1m), 0.30 (5m), 0.35 (15m)  
  
Memory Information:  
MemTotal: 16384000 kB  
MemFree: 12000000 kB  
MemAvailable: 14000000 kB

**Track Process:**

Enter PID **to** track: 5678  
Now tracking process 5678 (bash)

**Show Tracked Processes:**

TRACKED PROCESSES:  
PID NAME STATE **RUNTIME**(s)  
------------------------------------------------  
5678 bash S 120

**7. Limitations**

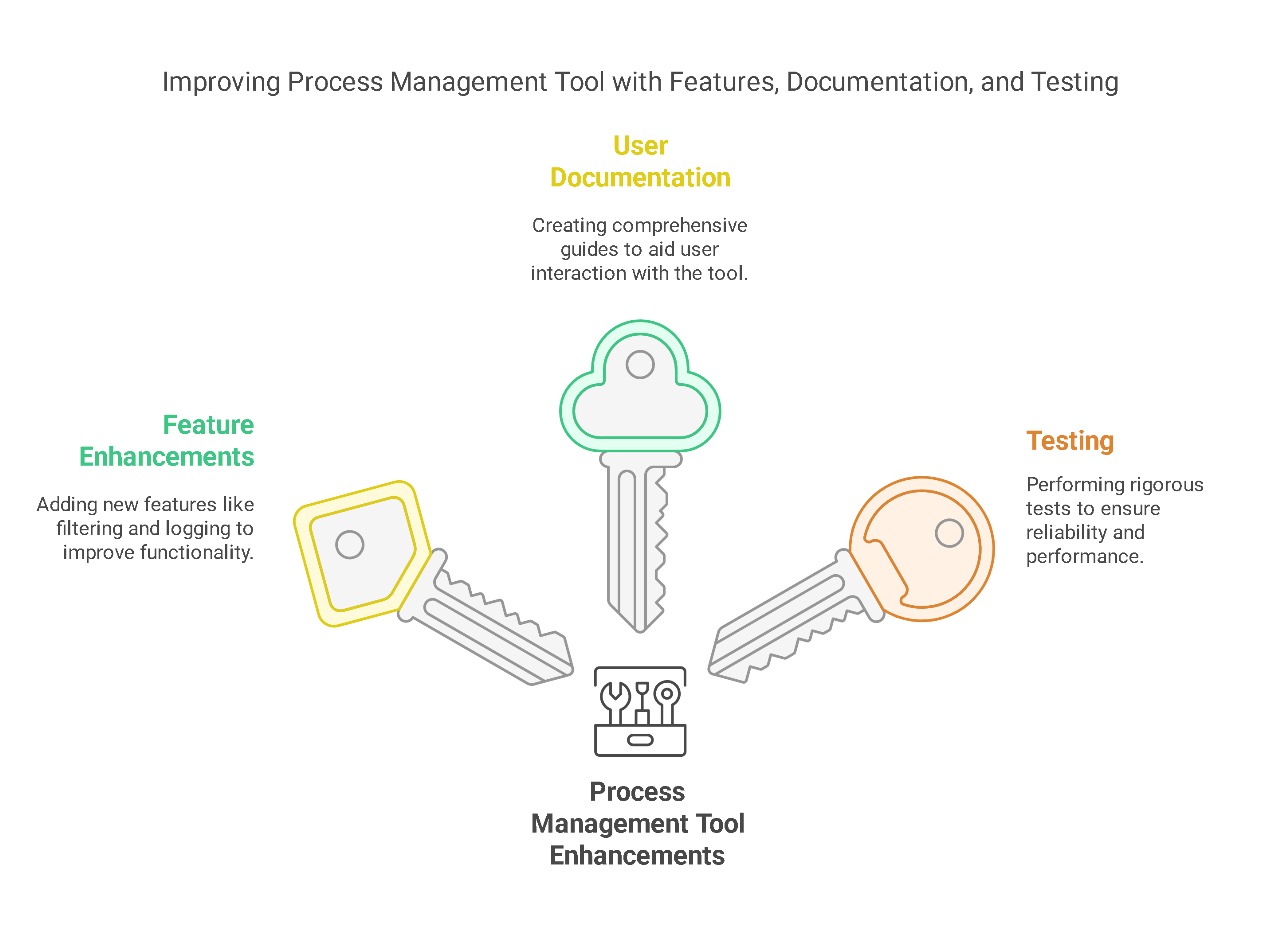
1. The program is designed for Linux systems and relies on the /proc

filesystem.

1. It can track a maximum of 1024 processes simultaneously.
2. Error handling is minimal, and some edge cases may not be handled gracefully.

**8. Future Enhancements**

1. **Enhanced Error Handling**: Improve error handling for invalid inputs and edge cases.
2. **Real-Time Monitoring**: Implement real-time monitoring of system load and processes.
3. **Graphical Interface**: Develop a graphical user interface (GUI) for better usability.
4. **Logging**: Add logging functionality to record process activities and system events.



**9. Conclusion**

The **Process Management Tool** is a powerful tool for system administrators and developers to monitor and manage processes on a Linux system. Its modular design and user-friendly interface make it easy to extend and customize for specific use cases. With further enhancements, it can become an indispensable tool for system monitoring and process management.

**Code:**

1. #include <stdio.h>

2. #include <stdlib.h>

3. #include <unistd.h>

4. #include <string.h>

5. #include <signal.h>

6. #include <dirent.h>

7. #include <sys/types.h>

8. #include <sys/wait.h>

9. #include <sys/resource.h>

10. #include <ctype.h>

11. #include <time.h>

12. #include <pwd.h>

13.

14. // ANSI Color Codes

15. #define COLOR\_RED "\x1b[31m"

16. #define COLOR\_GREEN "\x1b[32m"

17. #define COLOR\_YELLOW "\x1b[33m"

18. #define COLOR\_BLUE "\x1b[34m"

19. #define COLOR\_MAGENTA "\x1b[35m"

20. #define COLOR\_CYAN "\x1b[36m"

21. #define COLOR\_RESET "\x1b[0m"

22.

23. #define MAX\_COMMAND\_LENGTH 256

24. #define MAX\_PROCESS\_COUNT 1024

25. #define PATH\_MAX 4096

26.

27. typedef struct {

28. pid\_t pid;

29. char name[256];

30. char user[256];

31. long memory;

32. char state;

33. time\_t start\_time;

34. } ProcessInfo;

35.

36. ProcessInfo tracked\_processes[MAX\_PROCESS\_COUNT];

37. int tracked\_count = 0;

38.

39. // Added the missing functions

40. int terminate\_process(pid\_t pid, const char\* signal\_type) {

41. int sig;

42. if (strcmp(signal\_type, "SIGKILL") == 0) {

43. sig = SIGKILL;

44. } else {

45. sig = SIGTERM;

46. }

47.

48. if (kill(pid, sig) == 0) {

49. printf(COLOR\_GREEN "Process %d terminated successfully with %s\n" COLOR\_RESET, pid, signal\_type);

50. return 0;

51. } else {

52. printf(COLOR\_RED "Error terminating process %d\n" COLOR\_RESET, pid);

53. return -1;

54. }

55. }

56.

57. void get\_process\_details(pid\_t pid) {

58. char path[PATH\_MAX], line[256];

59. FILE \*fp;

60.

61. // Status file

62. snprintf(path, sizeof(path), "/proc/%d/status", pid);

63. fp = fopen(path, "r");

64. if (!fp) {

65. printf(COLOR\_RED "Unable to get details for process %d\n" COLOR\_RESET, pid);

66. return;

67. }

68.

69. printf(COLOR\_YELLOW "\nProcess Details for PID %d:\n" COLOR\_RESET, pid);

70. printf("----------------------------------------\n");

71.

72. while (fgets(line, sizeof(line), fp)) {

73. // Print important process information

74. if (strncmp(line, "Name:", 5) == 0 ||

75. strncmp(line, "State:", 6) == 0 ||

76. strncmp(line, "Pid:", 4) == 0 ||

77. strncmp(line, "PPid:", 5) == 0 ||

78. strncmp(line, "VmSize:", 7) == 0 ||

79. strncmp(line, "VmRSS:", 6) == 0 ||

80. strncmp(line, "Threads:", 8) == 0) {

81. printf("%s", line);

82. }

83. }

84. fclose(fp);

85.

86. // Cmdline

87. snprintf(path, sizeof(path), "/proc/%d/cmdline", pid);

88. fp = fopen(path, "r");

89. if (fp) {

90. if (fgets(line, sizeof(line), fp)) {

91. printf("Command: %s\n", line);

92. }

93. fclose(fp);

94. }

95. }

96.

97. int start\_process(const char\* command) {

98. pid\_t pid = fork();

99.

100. if (pid < 0) {

101. printf(COLOR\_RED "Error: Fork failed\n" COLOR\_RESET);

102. return -1;

103. } else if (pid == 0) {

104. // Child process

105. char\* args[] = {"/bin/sh", "-c", (char\*)command, NULL};

106. execvp("/bin/sh", args);

107. printf(COLOR\_RED "Error: Command execution failed\n" COLOR\_RESET);

108. exit(1);

109. } else {

110. // Parent process

111. int status;

112. waitpid(pid, &status, 0);

113. if (WIFEXITED(status)) {

114. printf(COLOR\_GREEN "Process completed with status %d\n" COLOR\_RESET, WEXITSTATUS(status));

115. return 0;

116. } else {

117. printf(COLOR\_RED "Process terminated abnormally\n" COLOR\_RESET);

118. return -1;

119. }

120. }

121. }

122.

123. // Rest of the existing functions remain the same

124. void display\_banner() {

125. printf(COLOR\_CYAN);

126. printf(

127. "╔═══════════════════════════════════════════╗\n"

128. "║ 🖥 Advanced Process Manager 2.0 🖥 ║\n"

129. "╠═══════════════════════════════════════════╣\n"

130. "║ System Monitoring & Control Center ║\n"

131. "╚═══════════════════════════════════════════╝\n"

132. COLOR\_RESET);

133. }

134.

135. void list\_processes() {

136. DIR \*dir;

137. struct dirent \*entry;

138. char path[PATH\_MAX], line[256], user[256];

139. FILE \*fp;

140. struct passwd \*pw;

141.

142. printf(COLOR\_GREEN "\nACTIVE PROCESSES:\n" COLOR\_RESET);

143. printf("%-8s %-15s %-12s %-8s\n", "PID", "USER", "STATE", "COMMAND");

144. printf("----------------------------------------\n");

145.

146. dir = opendir("/proc");

147. if (!dir) {

148. perror(COLOR\_RED "Failed to open /proc" COLOR\_RESET);

149. return;

150. }

151.

152. while ((entry = readdir(dir))) {

153. if (!isdigit(\*entry->d\_name))

154. continue;

155.

156. pid\_t pid = atoi(entry->d\_name);

157. snprintf(path, sizeof(path), "/proc/%d/status", pid);

158.

159. fp = fopen(path, "r");

160. if (!fp) continue;

161.

162. char state = '?';

163. uid\_t uid = 0;

164. char command[256] = "unknown";

165.

166. while (fgets(line, sizeof(line), fp)) {

167. if (strncmp(line, "State:", 6) == 0) {

168. sscanf(line, "State: %c", &state);

169. } else if (strncmp(line, "Uid:", 4) == 0) {

170. sscanf(line, "Uid: %d", &uid);

171. } else if (strncmp(line, "Name:", 5) == 0) {

172. sscanf(line, "Name: %255s", command);

173. }

174. }

175. fclose(fp);

176.

177. pw = getpwuid(uid);

178. strncpy(user, pw ? pw->pw\_name : "unknown", sizeof(user)-1);

179.

180. printf("%-8d %-15s %-12c %-8s\n",

181. pid, user, state, command);

182. }

183. closedir(dir);

184. }

185.

186. void analyze\_system\_load() {

187. FILE \*fp;

188. char line[256];

189. double loads[3];

190.

191. printf(COLOR\_MAGENTA "\nSYSTEM LOAD ANALYSIS:\n" COLOR\_RESET);

192.

193. // CPU Load

194. fp = fopen("/proc/loadavg", "r");

195. if (fp) {

196. if (fscanf(fp, "%lf %lf %lf", &loads[0], &loads[1], &loads[2]) == 3) {

197. printf("Load Averages: %.2f (1m), %.2f (5m), %.2f (15m)\n",

198. loads[0], loads[1], loads[2]);

199. }

200. fclose(fp);

201. }

202.

203. // Memory Info

204. fp = fopen("/proc/meminfo", "r");

205. if (fp) {

206. printf("\nMemory Information:\n");

207. int count = 0;

208. while (fgets(line, sizeof(line), fp) && count < 3) {

209. printf("%s", line);

210. count++;

211. }

212. fclose(fp);

213. }

214. }

215.

216. void track\_process(pid\_t pid) {

217. if (tracked\_count >= MAX\_PROCESS\_COUNT) {

218. printf(COLOR\_RED "Maximum tracking limit reached\n" COLOR\_RESET);

219. return;

220. }

221.

222. char path[PATH\_MAX], line[256];

223. snprintf(path, sizeof(path), "/proc/%d/status", pid);

224.

225. FILE \*fp = fopen(path, "r");

226. if (!fp) {

227. printf(COLOR\_RED "Process %d not found\n" COLOR\_RESET, pid);

228. return;

229. }

230.

231. ProcessInfo \*proc = &tracked\_processes[tracked\_count];

232. proc->pid = pid;

233. proc->start\_time = time(NULL);

234.

235. while (fgets(line, sizeof(line), fp)) {

236. if (strncmp(line, "Name:", 5) == 0) {

237. sscanf(line, "Name: %255s", proc->name);

238. } else if (strncmp(line, "State:", 6) == 0) {

239. sscanf(line, "State: %c", &proc->state);

240. }

241. }

242. fclose(fp);

243.

244. tracked\_count++;

245. printf(COLOR\_GREEN "Now tracking process %d (%s)\n" COLOR\_RESET,

246. pid, proc->name);

247. }

248.

249. void display\_tracked\_processes() {

250. if (tracked\_count == 0) {

251. printf(COLOR\_YELLOW "No processes being tracked\n" COLOR\_RESET);

252. return;

253. }

254.

255. printf(COLOR\_GREEN "\nTRACKED PROCESSES:\n" COLOR\_RESET);

256. printf("%-8s %-15s %-10s %-15s\n", "PID", "NAME", "STATE", "RUNTIME(s)");

257. printf("------------------------------------------------\n");

258.

259. time\_t now = time(NULL);

260. for (int i = 0; i < tracked\_count; i++) {

261. ProcessInfo \*proc = &tracked\_processes[i];

262. long runtime = now - proc->start\_time;

263.

264. // Verify if process still exists

265. char path[PATH\_MAX];

266. snprintf(path, sizeof(path), "/proc/%d", proc->pid);

267. if (access(path, F\_OK) != -1) {

268. printf("%-8d %-15s %-10c %-15ld\n",

269. proc->pid, proc->name, proc->state, runtime);

270. } else {

271. printf("%-8d %-15s %-10s %-15s\n",

272. proc->pid, proc->name, "ENDED", "-");

273. }

274. }

275. }

276.

277. void display\_menu() {

278. printf("\n" COLOR\_BLUE);

279. printf("╔═══════════════════════════════════════════╗\n");

280. printf("║ MENU OPTIONS ║\n");

281. printf("╠═══════════════════════════════════════════╣\n");

282. printf("║ 1. " COLOR\_CYAN "List Active Processes " COLOR\_BLUE "║\n");

283. printf("║ 2. " COLOR\_CYAN "Terminate Process " COLOR\_BLUE "║\n");

284. printf("║ 3. " COLOR\_CYAN "Monitor System Load " COLOR\_BLUE "║\n");

285. printf("║ 4. " COLOR\_CYAN "Get Process Details " COLOR\_BLUE "║\n");

286. printf("║ 5. " COLOR\_CYAN "Start New Process " COLOR\_BLUE "║\n");

287. printf("║ 6. " COLOR\_CYAN "Track New Process " COLOR\_BLUE "║\n");

288. printf("║ 7. " COLOR\_CYAN "Show Tracked Processes " COLOR\_BLUE "║\n");

289. printf("║ 8. " COLOR\_RED "Exit " COLOR\_BLUE "║\n");

290. printf("╚═══════════════════════════════════════════╝\n");

291. printf(COLOR\_GREEN "Enter your choice: " COLOR\_RESET);

292. }

293.

294. int main() {

295. int choice;

296. char input[256];

297. pid\_t pid;

298.

299. display\_banner();

300.

301. while (1) {

302. display\_menu();

303. if (scanf("%d", &choice) != 1) {

304. while (getchar() != '\n'); // Clear input buffer

305. printf(COLOR\_RED "Invalid input. Please enter a number.\n" COLOR\_RESET);

306. continue;

307. }

308. while (getchar() != '\n'); // Clear input buffer

309.

310. switch(choice) {

311. case 1:

312. list\_processes();

313. break;

314. case 2:

315. printf("Enter PID to terminate: ");

316. if (scanf("%d", &pid) == 1) {

317. printf("Enter signal (SIGTERM/SIGKILL): ");

318. scanf("%s", input);

319. terminate\_process(pid, input);

320. }

321. while (getchar() != '\n'); // Clear input buffer

322. break;

323. case 3:

324. analyze\_system\_load();

325. break;

326. case 4:

327. printf("Enter PID for details: ");

328. if (scanf("%d", &pid) == 1) {

329. get\_process\_details(pid);

330. }

331. while (getchar() != '\n');

332. break;

333. case 5:

334. printf("Enter command to execute: ");

335. fgets(input, sizeof(input), stdin);

336. input[strcspn(input, "\n")] = 0;

337. start\_process(input);

338. break;

339. case 6:

340. printf("Enter PID to track: ");

341. if (scanf("%d", &pid) == 1) {

342. track\_process(pid);

343. }

344. while (getchar() != '\n');

345. break;

346. case 7:

347. display\_tracked\_processes();

348. break;

349. case 8:

350. printf(COLOR\_RED "Exiting Process Manager.\n" COLOR\_RESET);

351. return 0;

352. default:

353. printf(COLOR\_RED "Invalid choice. Please try again.\n" COLOR\_RESET);

354. }

355. }

356. return 0;

357. }

358.