CIS 415 Operating Systems

Project 5 Report Collection

Submitted to:

Prof. Allen Malony

Author:

Arturo Diaz

arturod

951759326

Report

Introduction

This project involved the development of an MCP (Master Control Program) that manages workload processes with scheduling and monitoring functionalities. The MCP aims to create a time-shared environment by scheduling processes with equal time slices, allowing only one workload process to execute at a time. Through four main parts, this project incrementally expanded the MCP's capabilities. Part 1 focused on creating an MCP that forks and manages processes, while Part 2 introduced control over process execution through signals. Part 3 added round-robin scheduling with adjustable time slices, and Part 4 implemented resource monitoring using the /proc filesystem to gather system resource data per process.

The purpose of this project is to understand process management, including the use of system calls (fork, execvp, kill), signals (SIGUSR1, SIGSTOP, SIGCONT), and scheduling strategies. Additionally, accessing and analyzing the /proc directory gives insight into real-time resource usage, which is essential for operating systems.

Background

The MCP project integrates multiple operating systems concepts, particularly process scheduling, signal handling, and resource monitoring. Round-robin scheduling was chosen for its simplicity and fairness, as it allocates equal CPU time to each process in a cyclic order. This approach is commonly used in time-shared systems where each process gets an equal opportunity to run, preventing any single process from monopolizing the CPU.

The project also leverages system calls to manage processes. Specifically, fork creates new processes, execvp allows each process to run its workload, and kill sends signals to control these processes. The use of signals like SIGUSR1, SIGSTOP, SIGCONT, and SIGALRM allows MCP to manage when each workload process runs or pauses. By accessing /proc/[pid]/stat for each child process, MCP gathers essential metrics like system CPU time, virtual memory usage, and resident set size (RSS), providing a detailed view of each process's resource usage.

Implementation

The MCP implementation is divided into four parts:

Process Creation (Part 1):

In this initial stage, MCP forks processes for each workload and uses execvp to execute workload programs. This approach allows the parent MCP process to control and monitor child processes.

```
//loop until the file is over
while (getline (&line_buf, &len, inFPtr) != -1){
        //tokenize line buffer, large token is seperated by ";"
        large_token_buffer = str_filler (line_buf, " ");

        // fork
        pid_t process = fork();
        if (process < 0){
                fprintf(stderr, "fork failed\n");
                exit(-1);
        } else if (process == 0){
                execvp(large_token_buffer.command_list[0], large_token_buffer.command_list);
                exit(0);
        }

        //free large token and reset variable
        free_command_line (&large_token_buffer);
        memset (&large_token_buffer, 0, 0);
}

while(wait(NULL) > 0);

// Close and free buffer
fclose(inFPtr);
free (line_buf);
```

Controlled Execution (Part 2):

Part 2 focused on pausing each child process before execvp execution, then resuming them with SIGUSR1. This mechanism gives MCP control over when each process starts.

```
//loop until the file is over
while (getline (&line_buf, &len, inFPtr) != -1){
        //tokenize line buffer, large token is seperated by ";"
        large_token_buffer = str_filler (line_buf, " ");

        // fork
        process[line_num] = fork();

        if (process[line_num] < 0){
                fprintf(stderr, "fork failed\n");
                exit(-1);
        } else if (process[line_num] == 0){
                // Child process: Set up to wait for SIGUSR1 signal
                sigset_t sigset;
                sigemptyset(&sigset);
                sigaddset(&sigset, SIGUSR1);
                sigprocmask(SIG_BLOCK, &sigset, NULL);
                int sig;
                sigwait(&sigset, &sig);  // Wait for SIGUSR1

                execvp(large_token_buffer.command_list[0], large_token_buffer.command_list);
                exit(0);
        }
        line_num = line_num + 1;
        //free large token and reset variable
        free_command_line (&large_token_buffer);
        memset (&large_token_buffer, 0, 0);
}

// Make sure children dont get sent before signals get sent
sleep(1);

// Loop through proccess ID's and send SIGUSER1 signal to resume
printf("Sending SIGUSR1 to all processes...\n");
for (int i = 0; i < line_num; i++) {
        kill(process[i], SIGUSR1);
}

// Send SIGSTOP to all child processes to suspend them
printf("Sending SIGSTOP to process...\n");
for (int i = 0; i < line_num; i++) {
        kill(process[i], SIGSTOP);
}

// Send SIGCONT to wake each process
printf("Sending SIGCONT to process...\n");
for (int i = 0; i < line_num; i++) {
        kill(process[i], SIGCONT);
}

// Wait for all child processes to complete
int status;
for (int i = 0; i < line_num; i++) {
        waitpid(process[i], &status, 0);
}
```

Round-Robin Scheduling (Part 3):

Here, MCP introduced round-robin scheduling using SIGALRM and signal_handler. By switching processes every 1–2 seconds based on system CPU time, MCP ensured equal time sharing. MCP also monitored the system CPU time for each process and adjusted the time slice dynamically based on a threshold, using /proc/[pid]/stat to track each process's system time.

```
void signal_handler(int sig){
        int i = 0;
        int j = 0;

        // Stop whats currenlty running
        kill(process[curr_process], SIGSTOP);

        // Loop through processes, starting at next
        for (i = curr_process + 1; i < line_num + curr_process + 1; i++) {

                // Get mod
                j = i%line_num;

                int ret = kill(process[j], 0); // Null signal

                if (ret == 0){
                        // Hasn't finished running
                        kill(process[j], SIGCONT);
                        break;
                }
        }

        alarm(1);

        // Update current signal
        curr_process = j;
}
```

Resource Monitoring (Part 4):

The final part extended MCP to retrieve and display system resource information (CPU time, memory, I/O) for each workload process. MCP reads data from /proc/[pid]/stat to extract system CPU time (stime), user CPU time (utime), virtual memory size, and RSS. This data is formatted into a table that updates each cycle, similar to Linux's top command. Based on the cumulative system CPU time, MCP adjusts the alarm frequency, setting it to 2 seconds if a process's system time exceeds a defined threshold, else to 1 second.

```c
void print_process_info(pid_t pid) {
    char proc_path[64];
    snprintf(proc_path, sizeof(proc_path), "/proc/%d/stat", pid);

    FILE *fp = fopen(proc_path, "r");
    if (fp == NULL) {
        printf("Process has finished running, no data to populate...\n");
        return;
    }

    int pid_num;
    char comm[256];
    char state;
    unsigned long utime, stime, vsize;
    long rss;

    // Read specific fields from /proc/[pid]/stat
    fscanf(fp, "%d %s %c", &pid_num, comm, &state);
    for (int i = 0; i < 10; i++) fscanf(fp, "%*s");  // Skip fields 4-13
    fscanf(fp, "%lu %lu", &utime, &stime);           // Fields 14 (utime) and 15 (stime)
    for (int i = 0; i < 6; i++) fscanf(fp, "%*s");   // Skip fields 16-21
    fscanf(fp, "%lu %ld", &vsize, &rss);             // Field 22 (vsize) and 23 (rss)

    fclose(fp);

    // Display process info in table row format
    printf("%-10d %-20lu %-20lu %-15lu %-10ld\n",
           pid_num, utime, stime, vsize, rss);
}
```

Adaptive Scheduling with Adjustable Time Slices (Part 5 - Extra Credit):

The final part implemented an adaptive scheduler that adjusts each process's time slice based on its I/O or CPU-bound behavior. This refinement helps optimize the workload further by identifying I/O-bound and CPU-bound processes and allocating them specific time slices.

> ***Behavior Detection:*** *MCP uses system CPU time as an indicator. A high system CPU time suggests a CPU-bound process, while a lower system time indicates I/O-bound behavior.*
>
> ***Dynamic Time Slicing:*** *MCP sets a time slice of 1 second for I/O-bound processes and 2 seconds for CPU-bound processes, adjusting the interval to maximize CPU utilization.*
>
> ***Enhanced Tabular Display:*** *The real-time table output now includes additional columns for Allocated Time Slice and Process Type, which show the assigned time interval and detected workload type (CPU-bound or I/O-bound) for each process.*

```c
if (system_cpu_time[curr_process] > threshold) {
        //printf("Scheduler: High system CPU time for process %d, setting alarm to 2 seconds\n", process[curr_process]);
        alarm(2);
} else {
        //printf("Scheduler: System CPU time below threshold for process %d, setting alarm to 1 second\n", process[curr_process]);
        alarm(1);
}
```

Performance Results and Discussion

The MCP demonstrated accurate process control, efficient resource management, and dynamic scheduling, resulting in a well-balanced workload distribution across processes. Testing confirmed that MCP allocates time slices based on each process's behavior, making it effective in reducing idle CPU time for I/O-bound processes and providing longer uninterrupted execution for CPU-bound processes. The real-time resource monitoring table provided updated feedback every cycle, showing process information such as PID, CPU time, memory usage, allocated time slice, and process type.

Part 5 Testing Observations:

Processes classified as CPU-bound due to higher system CPU times were consistently assigned a 2-second slice, allowing them to complete intensive tasks with fewer context switches.

I/O-bound processes that required frequent waiting for user input or data I/O were given 1-second slices, minimizing idle time and making more CPU time available for active processes.

By tailoring time slices to process types, MCP successfully demonstrated how dynamic scheduling can lead to a more efficient distribution of CPU resources, achieving both time-sharing and workload optimization.

Conclusion

This project reinforced OS fundamentals, covering key topics such as process control, scheduling algorithms, signal handling, and system monitoring. The progressive development of MCP highlighted the practical challenges in building a fully functional scheduler and adaptive time-sharing system. Each part built on the previous one, adding complexity in terms of process control, resource management, and adaptive scheduling.

The implementation of real-time monitoring from /proc provided valuable insights into each process's resource consumption, while the adaptive scheduler in Part 5 demonstrated an effective approach for handling mixed I/O and CPU-bound workloads. Overall, the project showcases the essential mechanisms required for efficient process management in an operating system, simulating a mini-scheduler that dynamically adjusts to optimize system resources.