Weekly Report

CS3500: Operating Systems

Visualisation Tool for Process Scheduling

Computer Science and Engineering

Indian Institute of Technology Madras

Jul - Nov 2024

Under the supervision

of

Prof. Janakiram D

submitted by

Team 8

- Anjali Samudrala (CS22B046)
- Chaitanya Sai Teja G (CS22B036)
- Jwala Likitha Reddy M (CS22B078)
- Karthikeya P (CS22B026)
- Naveen Koushik Reddy E (CS22B006)
- Navya Sree B (CS22B045)
- Rushi Babu G (CS22B040)
- Sravya Rangu (CS22B044)
- Yashwanth Sai P (CS22B002)
- Yaswanth Sai V (CS22B043)

1 Frontend Team

1.1 Tasks

- The tasks for this week is to identify the components that are required for the frontend. This includes the following.
  - Coming up with a frontend design and layout.
  - Ideation on what all different plots can be displayed.
  - Informing the backend team about the requirements of various informations.

1.2 Progress

1.2.1 Frontend Design and Basic implementation

-
  ```
  from flask import Flask
  from flask_socketio import SocketIO
  import psutil
  import threading
  import time
  app = Flask(__name__)
  socketio = SocketIO(app, cors_allowed_origins="*")
  def fetch_process_data():
      # Fetches PIDs running on Core 0 and emits them to the frontend.
      while True:
          # Initialize a list to store the PIDs running on Core 0
          pids_on_core_0 = []
          # Iterate over all processes to get their core and PID info
          for proc in psutil.process_iter(['pid']):
              try:
                  # Check if the process is running on core 0
                  if proc.cpu_num() == 0:  # Use proc.cpu_num() here
                      pids_on_core_0.append(proc.pid)
              except (psutil.NoSuchProcess, psutil.AccessDenied):
                  continue
          # Emit data to the frontend using Socket.IO
          socketio.emit('process_data', {'pids': pids_on_core_0})
          time.sleep(2)  # Adjust interval as needed
  @app.route('/')
  def index():
      return "Backend is running!"
  # Start the background thread to fetch data and emit to the frontend
  threading.Thread(target=fetch_process_data).start()
  if __name__ == '__main__':
      socketio.run(app, debug=True)
  ```

- This Flask app uses Flask-SocketIO to emit the PIDs of processes running on CPU core 0 to the frontend every 2 seconds.

- It continuously monitors processes in a background thread and sends updates in real time via WebSockets.

```
import React, { useEffect, useState } from 'react';
import io from 'socket.io-client';
import './App.css';

const socket = io('http://localhost:5000'); // Adjust the port if needed

function App() {
  const [pids, setPids] = useState([]);

  useEffect(() => {
    // Connect to Socket.IO and listen for 'process_data' events
    socket.on('process_data', (data) => {
      setPids(data.pids);
    });

    // Clean up on component unmount
    return () => {
      socket.off('process_data');
    };
  }, []);
   return (
    <div className="App">
      <h1>Running PIDs on Core 0</h1>
      <ul>
        {pids.map(pid => (
          <li key={pid}>PID: {pid}</li>
        ))}
      </ul>
    </div>
  );
  }
```

- This React component connects to a Flask-SocketIO server, listening for 'processdata' events that provide a list of PIDs of processes running on CPU core 0.
- It displays these PIDs in real time within an HTML list, updating whenever new data is received from the backend.

1.2.2 Ideas for Plots and information from backend

- **Processes Running on the System:**
  - We will display all the process that are in the CPU from the current time t to $t - \delta$ where δ is the time interval.
  - This plot will be updated in real-time to show the current processes running on the system.
  - The plot will be flowing backwards and at anypoint if you draw a line perpendicular to the time-axis, you will get the processes running at that time.

  - For this the start time of the processes and approximate end time of the processes will be required.

- **Gantt Chart of various Processes**
  - The Gantt chart can be used to visualize the execution of processes over time. Each process can be represented as a bar, with the length of the bar indicating the duration of the process.
  - The Gantt chart can show the start and end times of each process, as well as the CPU on which the process is running.
  - The bar for each process also has a color code to indicate the state of the process (e.g., running, waiting, etc.).
  - The chart can be updated in real time to reflect changes in process execution.
  - Initially since there can be many processes, we wish to create a Gantt chart for a few processes that take have a maximum core usage and the user can request the bar graph of the process that he requires.

- **Status wise distribution of processes**
  - We will display what all processes are running, sleeping, waiting, etc.
  - Each process can be indicated as a Bar which can be used to represent the CPU usage of the process.
  - Then this plot can be periodically updated to show the current status of the processes and movement of the processes from one state to another.
  - This will help us to analyse how the states of the processes are changing over time and what is the impact of the CPU usage on the state changes.
  - For this we need the CPU usage and the state of the processes.

- **CPU Utilization**
  - The CPU utilization is a sub-plot that can be used to see the dependency of the system on each core.
  - The plot can be updated in real-time to show the current CPU utilization.
  - The plot can show the CPU utilization of each core, as well as the overall CPU utilization.
  - The plot can also display the average CPU utilization over a specified time period.
  - The plot can be used to identify cores that are underutilized or overloaded.

2 Backend Team

2.1 Extracting Process Information

2.1.1 Command: `ps -eo pid,lstart,comm`

Purpose: This command displays the details of all currently running processes with the following information:

- `pid`: The Process ID, which uniquely identifies each process.

- `lstart`: The start time of the process.
- `comm`: The command or program name that initiated the process.

This command is useful for tracking when a process was started and understanding its origin.

2.1.2 Command: `ps -eo pid,etime,comm`

Purpose: This command displays the following details for all running processes:

- `etime`: The elapsed time since the process started, which tells us how long the process has been running.

This information is useful for understanding how long a process has been active since it started.

2.1.3 Command: `cat /proc/[pid]/stat`

Purpose: This command is useful for retrieving detailed information about a specific process. It provides the following data:

- **State**: The current state of the process (e.g., running, sleeping, etc.).
- **Utime**: The total time the process has been running in user mode. This value represents the amount of CPU time consumed by the process in user mode.
- **Stime**: The total time the process has been running in kernel mode. This value represents the CPU time spent by the process in kernel space.

This command is valuable for getting low-level process statistics and understanding the process's CPU usage in both user and kernel modes.

2.1.4 Command: `ps aux`

Purpose: This command provides more detailed information about all running processes, including:

- **State**: The current state of the process.
- **Start time**: The start time of the process.
- **Total CPU time**: The total CPU time the process has consumed.

This command is useful for monitoring various aspects of processes and understanding their current state and resource usage.

2.1.5 Command: `pidstat`

Purpose: The `pidstat` command is used to gather detailed statistics for individual processes. It can provide more granular information about resource utilization, including CPU usage, memory usage, and more, for each process.

This tool is helpful for detailed performance analysis and understanding how specific processes affect system resources.

2.1.6 Command: `ps -p pid`

Purpose: To know whether the process ended or not.

- If it shows some data, it is still running.
- If no data is shown, the process ended.

Consider a specific time duration. During this time, we generate a list of all processes currently running. After this duration has passed, we check the list of running processes again. If we find that any process from the initial list is no longer running, it indicates that this process has completed or ended.

2.2 Linking Backend and Frontend

The objective is to create a backend server that fetches real-time system process statistics using the `pidstat` command and streams the data to a frontend via WebSockets. The application uses Flask as the web framework and Flask-SocketIO to establish real-time communication between the backend and frontend.

2.2.1 Tools and Libraries Used

- **Flask**: A lightweight web framework for Python that simplifies the development of web applications.
- **Flask-SocketIO**: An extension for Flask that enables real-time communication between the server and the client using WebSockets.
- **subprocess**: A Python module used to run external commands. In this case, it is used to execute the `pidstat` command, which collects CPU statistics for processes running on the system.
- **re (Regular Expressions)**: A Python module for matching patterns in strings. It is used to parse the output of the `pidstat` command.
- **threading**: A Python module used to create background threads. In this project, it allows the data-fetching process to run concurrently with the main server.

2.2.2 Architecture

- **Backend (Flask Server with SocketIO) :**
  - The backend is responsible for fetching real-time process statistics using the `pidstat` command, parsing the output, and sending the data to the frontend using WebSockets.
- **Real-Time Data Fetching :**
  - The `pidstat` command is used to gather CPU statistics for processes every second. The output contains several fields, including process ID (PID), user and system CPU usage, and the process command.
  - A background thread is created to run the `pidstat` command continuously, fetching data at specified intervals (e.g., every 2 seconds).

    - The data is parsed using a regular expression, and relevant statistics are extracted and formatted into a dictionary.

- **Real-Time Communication :**
    - `Flask-SocketIO` is used to emit the parsed data to the frontend in real time. This allows the frontend to display the latest statistics as they are gathered by the backend.
    - The data is emitted as a WebSocket event (`'pidstat_data'`), making it available for frontend visualization.

2.3 Process Migration

2.3.1 Enabling tracking to log the process migration

- Navigate to the tracing directory.

  ```
  cd /sys/kernel/debug/tracing
  ```

- Enable "sched_migrate_task" that allows to track and log the migration of tasks.

  ```
  echo 1 | sudo tee events/sched/sched_migrate_task/enable
  ```

- Start tracing the events.

  ```
  echo 1 | sudo tee tracing_on
  ```

- Wait for some time and let tracer log some migrations.
- Stop the tracing.

  ```
  echo 0 | sudo tee tracing_on
  ```

- Check the trace file.

  ```
  cat trace
  ```

2.3.2 Logged Data

- An example of the logged data is shown below.
- *Chrome_ChildIOT-33594 [003] d..2. 27300.882725: sched_migrate_task : comm=Compositor pid=32759 prio=120 orig_cpu=0 dest_cpu=3*
- In the above log entry, Chrome with PID 33594 initiated the migration of the Compositor process with PID 32759 from CPU 0 to CPU 3.
- $prio = 120$ is the priority of the process. Lower prio means higher priority.
- 27300.882725 is the timestamp of the event. (migration).

2.4 Completely Fair Scheduler(CFS) Understanding

- As we are implementing this project on Linux OS, an understanding of its scheduler is beneficial.
- It assigns priority to a process based on its niceness value which in turn is based on the vruntime of the process.
- The process with the least vruntime is given the highest priority. vruntime (virtual runtime) tracks the time a process has spent on the CPU, adjusted for its priority.