BY:

NATHAN DEKEYREL
AND WADE FORTSON

ANALYSIS OF THE COMPLETELY FAIR SCHEDULER

GOALS OF PROJECT

- We wanted to provide an analysis of an actual scheduling algorithm used in modern operating systems
- While researching scheduling algorithms, we found the Completely Fair Scheduler used in Linux
- We wanted implement CFS to compare our implementation and Linux's

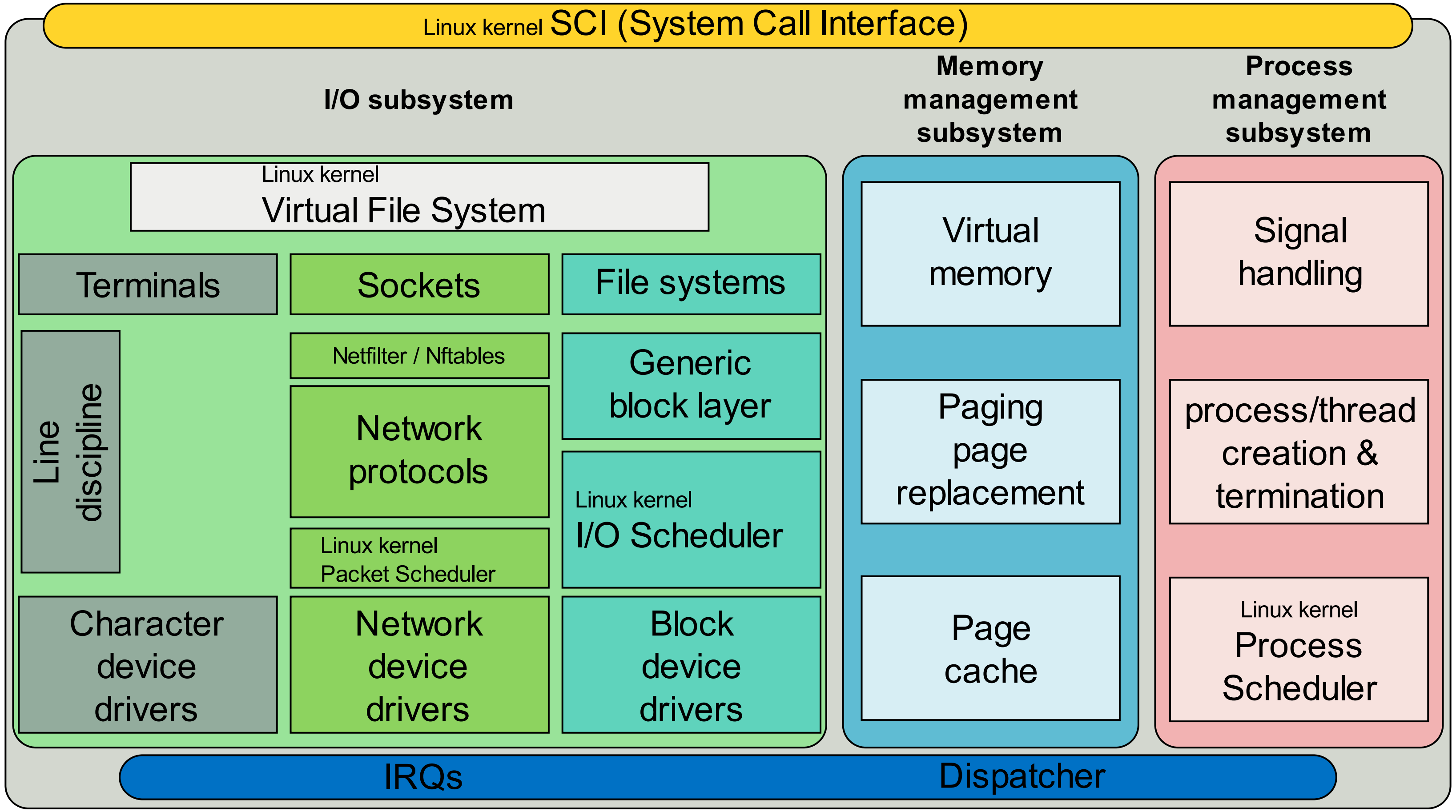
BACKGROUND

In class we saw 6 different scheduling algorithms

- FCFS, RR, SPN, SRT, HRRN, and FB

With a good understanding of these, we wanted to see what's used in industry

- Seeing how our own implementation compares



SIMPLIFIED LINUX DIAGRAM

CFS IN LINUX

- The main scheduler used by the Linux kernel since 2007
- Uses Red-black tree for sorting – O(log n) insertion and deletion
- Inspired by 'Ideal Fair Queueing' in network packet scheduling
- Replaced the O(1) scheduler previously used

O(1) SCHEDULER

Did not contain any algorithms that ran worse than O(1) time

Every part of scheduler guaranteed to complete in an upper-bounded, definite time regardless of the size of input

Algorithm

Uses 2 arrays – Active and Expired

Each process given fixed time quantum

- After exhausted, preempted and moved to Expired

Once active emptied, swap arrays (using pointers) and repeat

ISSUES WITH O(1) SCHEDULER

Identifies interactivity based on average sleep time

Processes that wait for user input could be assumed inactive

- Gives priority to interactive tasks
- Penalizing non-interactive by lowering priority

All calculations to determine interactivity are prone to miscalculation

High complexity

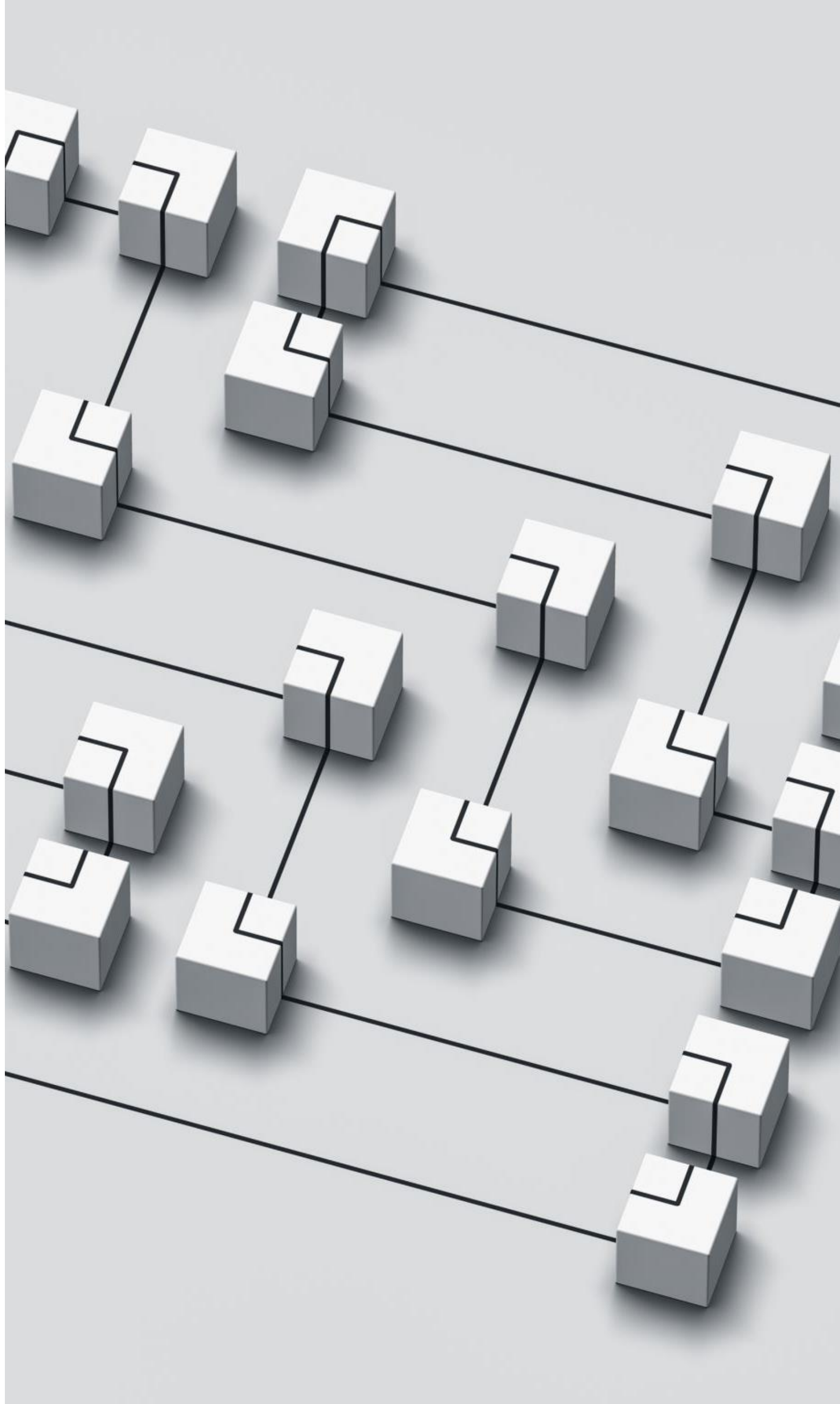
Extremely complicated algorithms with many edge cases

Unpredictable

Prioritizing response to user input can cause random shift in ordering

DESIGN OF CFS

- Goals
  - Reduce complexity – No special cases
  - Fair CPU distribution across processes
  - Better responsiveness – no misclassification of tasks
  - Predictable



CFS GROUNDWORK

Task – minimal entity that can be scheduled

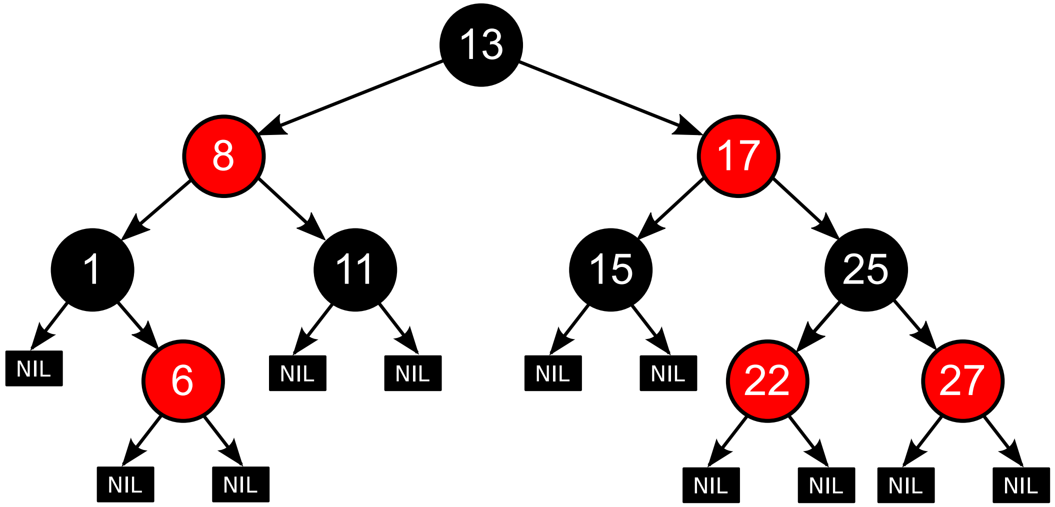
- Can also be a group of threads, leading to schedulable entities

Sorts schedulable entities in tree based on time

- Minimum vruntime stored in leftmost node
- Maximum execution time based on 'ideal processor'
  - Calculated as time_waiting/total_num_of_processes

RBTREE RECAP

- Every node is either Red or Black
  - Root is black
  - Leaves (which are NIL) are black
  - Red nodes can't have red children
  - All paths have the same number of black nodes
- Guarantees O(log n) for Insert, Delete, and finding minimum node
  - Also, self balancing



RBTREE FUNCTIONS

Helper functions:

- Rotate left
- Rotate right
- Handle Node Removal Issues

Tree Operations

- Insert
- Delete

Tree Traversal

- First
- Last
- Next
- Prev

IMPLEMENTATION DECISIONS

Linux	Us
• Handles concurrency • Advanced memory management • Multiple tree traversal strategies • Production level code	• Not concurrent • Simple memory management • Only uses one traversal strategy • Only suitable for simulation

CFS ALGORITHM

When scheduler is invoked to run a new task

- Leftmost node of tree is chosen and sent for execution
- If process completes, remove from system and tree
- If it reaches maximum execution time or is stopped, reinsert into tree
  - Reinsertion based on newly spent execution time
- New leftmost node selected, repeating process

This ensures that process that spend a lot of time sleeping don’t get assigned a low priority – also is given a boost when it needs it

CFS V O(1)

CFS	O(1)

• CFS
  • Pros
    • Predictable
    • Simple
    • Good for interactivity
  • Cons
    • Higher time complexity
    • Can lead to latency spikes
    • Expensive if lots of tasks
    • Memory overhead due to vruntime
	• O(1)
	  • Pros
	    • Defined upper-bound runtime regardless of workload size
	  • Cons
	    • Unpredictable
	    • Subject to miscalculations
	    • Low responsiveness

"CFS Basically models an 'ideal, precise multitasking CPU' on real hardware" – Ingo Molnar

PROGRAM DESIGN

SYSTEM ARCHITECTURE

1

Initialize workloads

1. Using python, we initialize workloads with semi-random values
2. These are generated as CSVs

2

Parse csv

- Parse the CSVs into a queue of process structures
- Containing the necessary info for our scheduler

3

Design the CFS algorithm

1. Using a red-black tree for scheduling operations

PROCESS DESIGN

We created a general process struct holding information relevant to each scheduler

We categorized attributes into

- Basic
- Timing
- CFS required
- State
- Queue Management

PROCESS DESIGN

Base attributes needed

- Process id
- priority

Timing – contains necessary attributes for scheduling and simulation

- Arrival time
- First run time
- Completion time
- Burst time
- Total time

PROCESS DESIGN

CFS specific attributes

vruntime

rb_node

State

We decided to use an enum for

- NEW, READY, RUNNING, WAITING, and COMPLETED processes

Queue Management

Keep track of next and previous process in queue

METRICS

Response Time	first runtime – arrival time
Turnaround Time	completion time – start time
Virtual Runtime Variance	delta_exec_weighed = delta_exec * (NICE_0_LOAD / curr->load.weight)

SHOW CODE

Generally, the drawbacks of CFS aren't noticeable

- For most cases, the default CFS algorithm works well enough
  - In modern OS, there usually aren't enough tasks to cause performance decreases
  - For specialized cases, tuning options are available within the Linux kernel
- For real-time tasks, the Linux kernel makes 2 separate schedulers available
  - SCHED_RR and SCHED_FIFO

If you have a highly specialized system, you can make a scheduler to fit your needs as the Linux kernel is modular

FINAL THOUGHTS