**COURSE PROJECT**

SUBJECT : **OPERATING SYSTEM**

**TY ET D BATCH 1 GROUP 2**

**GROUP MEMBERS:**

SANDESH NARENDRA BAGMARE(14) (12110445)

PREM SHEJOLE(26) (1211)

PRATHAMESH SAWLE (19)(12110339)

**Project Title: Multithreaded Web Crawler**

**Aim:**

The aim of this project is to develop a multithreaded web crawler that can efficiently crawl and index web pages. This project highlights the principles of multithreading, concurrent programming, and synchronization in the context of operating systems.

**Requirements:**

1. Implement a web crawler capable of traversing web pages and collecting URLs.

2. Utilize multithreading to parallelize the crawling process for improved performance.

3. Ensure proper synchronization to avoid race conditions and maintain data integrity.

4. Implement a custom HTML parser for extracting links from web pages.

5. Create a userfriendly interface for specifying project details and monitoring progress.

**Theory:**

Multithreading: Explain the concept of multithreading in the context of operating systems. Discuss the benefits of multithreading, such as improved resource utilization and responsiveness.

Synchronization: Describe the importance of synchronization in multithreaded programs. Discuss common synchronization mechanisms like locks, semaphores, and queues.

Web Crawling: Provide an overview of web crawling techniques, including depthfirst and breadthfirst strategies.

HTML Parsing: Explain the role of HTML parsers in extracting data from web pages.

Queuebased Architecture: Describe how a queuebased architecture can be used for managing the crawling tasks in a multithreaded environment.

**Project Flow:**

1. User enters the project name and homepage URL.

2. The system initializes a queue for URL management.

3. Multiple threads (as specified by the user) are created to crawl web pages concurrently.

4. Each thread retrieves URLs from the queue and crawls the corresponding web page.

5. The HTML parser extracts links from the crawled page, which are added to the queue for further crawling.

6. Proper synchronization ensures that multiple threads can access the queue safely.

7. The process continues until the queue is empty.

8. Crawled and queued URLs are stored in separate text files.

**Algorithm:**

1. **Thread Management Algorithm**:

**Objective:** The thread management algorithm is responsible for creating and coordinating multiple threads to perform web crawling concurrently.

**Algorithm Description:**

* Initialize a fixed number of worker threads (specified by the user) to perform crawling tasks.
* Each worker thread runs a continuous loop until there are no more URLs to crawl.
* In each iteration, a worker thread retrieves a URL from the shared queue of URLs to crawl.
* The thread then invokes the web crawling function for the retrieved URL.
* Proper synchronization mechanisms, such as locks or semaphores, are used to ensure that multiple threads can access the queue safely without conflicts.
* Threads are set to run as daemon threads to allow the main program to exit when crawling is complete.

2. **URL Crawling Algorithm:**

**Objective**: The URL crawling algorithm is responsible for fetching web pages and extracting links from the crawled pages.

**Algorithm Description:**

* When a worker thread receives a URL to crawl, it sends an HTTP request to the server to fetch the web page content.
* The thread then uses an HTML parsing library (e.g., BeautifulSoup) or a custom HTML parser to parse the HTML content and extract links (URLs) from the page.
* Extracted links are added to a list or queue to be processed in subsequent crawling iterations.
* The crawled web page may also be stored or processed further, depending on the project requirements.

3**. Link Extraction** **Algorithm**:

**Objective**: The link extraction algorithm focuses on extracting URLs (links) from the HTML content of a web page.

**Algorithm Description:**

* Initialize an HTML parser object, which is responsible for parsing the HTML content.
* When parsing the HTML content, the parser searches for anchor (`<a>`) tags.
* For each anchor tag found, the algorithm extracts the value of the `href` attribute, which contains the URL.
* The extracted URLs are added to a set or a list to ensure uniqueness.
* Proper URL normalization and validation may be performed to ensure that the extracted URLs are valid and ready for further crawling.

These algorithms work together to enable the multithreaded web crawler to efficiently crawl web pages, fetch their content, and extract links for subsequent crawling. Proper synchronization and data sharing mechanisms are crucial to prevent race conditions and ensure that the crawling process is efficient and safe in a multithreaded environment.

**Pseudocode:**

**Thread Creation Pseudocode:**

function create\_workers(number\_of\_threads):

for i from 1 to number\_of\_threads:

thread = create\_thread(target=work)

set\_thread\_as\_daemon(thread) # Daemon threads allow the program to exit when crawling is done

start\_thread(thread)

function work():

while true:

url = queue.get() # Get a URL to crawl from the queue

crawl\_page(current\_thread\_name(), url) # Call the crawling function

queue.task\_done() # Mark the task as done in the queue

**URL Crawling Pseudocode:**

function crawl\_page(thread\_name, url):

try:

page\_content = fetch\_page\_content(url) # Fetch the web page content

links = extract\_links\_from\_html(page\_content) # Extract links from the HTML content

for link in links:

if link not in crawled\_set: # Check if the link has not been crawled already

queue.put(link) # Add the new link to the queue for further crawling

add\_link\_to\_crawled\_set(link) # Mark the link as crawled

save\_page\_content\_to\_file(url, page\_content) # Optionally, save the crawled page content to a file

except Exception as e:

log\_error(thread\_name, e) # Handle and log any exceptions that occur during crawling

**Synchronization Mechanisms Pseudocode (Using Locks):**

# Initialize locks for thread synchronization

queue\_lock = create\_lock()

crawled\_set\_lock = create\_lock()

function crawl():

while not queue.is\_empty():

url = queue.get()

crawl\_page(current\_thread\_name(), url)

queue.task\_done()

function crawl\_page(thread\_name, url):

try:

queue\_lock.acquire() # Acquire the queue lock to access the shared queue

page\_content = fetch\_page\_content(url)

links = extract\_links\_from\_html(page\_content)

for link in links:

crawled\_set\_lock.acquire() # Acquire the crawled set lock to access the shared set

if link not in crawled\_set:

queue.put(link)

add\_link\_to\_crawled\_set(link)

crawled\_set\_lock.release() # Release the crawled set lock

queue\_lock.release() # Release the queue lock

except Exception as e:

log\_error(thread\_name, e)

Testing:

**Conclusion:**

1. Successful implementation of a multithreaded web crawler capable of efficiently crawling and indexing web pages.
2. Utilized concurrent programming to enable parallel crawling, significantly improving the crawler's performance.
3. Implemented proper synchronization mechanisms to avoid race conditions and ensure data integrity during multithreaded execution.

**Lessons Learned:**

* The importance of synchronization mechanisms (locks, semaphores) in preventing data conflicts among threads.
* The need for robust error handling to address unexpected issues during crawling.
* Strategies for load balancing and managing the queue of URLs to optimize crawler performance.

**Relevance of Multithreading in Operating Systems:**

* Multithreading and concurrent programming are fundamental in operating systems to:
* Enhance system responsiveness by allowing multiple tasks to run concurrently.
* Efficiently utilize CPU resources, especially in modern multicore processors.
* Handle parallel and asynchronous operations, such as I/O operations and task scheduling.
* Improve system efficiency and throughput.

**Future Enhancements and Extensions:**

* Potential future directions for the project:
* Implementation of a distributed web crawler to handle largerscale crawling tasks.
* Integration of advanced crawling strategies, such as prioritizing important pages or respecting robots.txt files.
* Enhanced error handling and logging to provide more detailed insights into crawling issues.
* Development of a userfriendly web interface for configuring and monitoring the crawler.
* Integration with a database or indexing system to store and search crawled data efficiently.