VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

OPERATING SYSTEMS

DIGITAL ASSIGNMENT

STUDENT	AATMIKA DIXIT 22BPS1183
STUDENT	ASHISH 22BAI1062
STUDENT	MILIND RANJAN 22BPS1167
STUDENT	VEERAJ KUTE 22BPS1129
COURSE	BCSE303P
CLASS NO	CH2023240100703
FACULTY	VALLIDEVI K

Processing large amounts of data in an operating system involves several techniques and algorithms. Here are some key approaches and well-known algorithms:

- **Memory Management:** The OS manages memory efficiently to ensure that data can be loaded and processed. This includes techniques like paging and segmentation.
- **File Systems:** The OS uses file systems to organise and store large volumes of data. Popular file systems include NTFS, ext4, and HFS+.
- **I/O Scheduling:** Algorithms like the C-SCAN, FCFS, and SSTF are used to schedule disk I/O requests, optimising data access.
- **Buffering:** Buffering involves using a buffer to temporarily hold data, reducing the frequency of I/O operations.Techniques like double buffering and circular buffering are used.
- **Caching:** Caching algorithms like LRU (Least Recently Used) and LFU (Least Frequently Used) help improve data access times by keeping frequently used data in memory.
- **Data Compression:** OS can use compression algorithms like Lempel-Ziv-Welch (LZW) or Deflate to reduce the size of data for storage and transmission.
- **Parallel Processing:** Multithreading and multiprocessing techniques allow the OS to distribute data processing tasks across multiple cores or processors.
- **Virtual Memory:** Virtual memory techniques, including demand paging and page replacement algorithms like LRU and FIFO, are used to efficiently manage memory for large datasets.
- **Data Indexing:** For searching and retrieval of data, indexing algorithms like B-trees, Hashing, and various search algorithms are employed.
- **Distributed File Systems:** In a networked environment, distributed file systems like Hadoop HDFS and Google File System (GFS) are used to process and store large datasets across multiple machines.
- **Data Partitioning:** Large datasets can be divided into smaller partitions, and MapReduce-style algorithms are used for parallel processing, with frameworks like Hadoop.
- **Database Management Systems:** When dealing with large datasets in a database, DBMS systems like MySQL, PostgreSQL, and NoSQL databases are employed for efficient data retrieval and management.

- **Streaming Algorithms:** For processing data streams, algorithms like Count-Min Sketch and HyperLogLog are used to estimate statistics from large data streams in real-time.

Operating systems use scheduling algorithms to manage the execution of multiple processes or threads efficiently. The scheduling algorithm's primary goal is to allocate CPU time to different tasks or processes while maximising system throughput, fairness, and responsiveness. Here are some common scheduling algorithms used by operating systems:

- **First-Come, First-Served (FCFS):**
- - It schedules processes based on their arrival time. The process that arrives first gets executed first.
- - It is simple but can lead to poor utilisation of CPU time, especially if long processes arrive early (known as the "convoy effect").
- **Shortest Job First (SJF):**
- - Prioritises processes with the smallest CPU burst or execution time.
- - It minimises average waiting time but requires knowledge of each process's execution time, which may not always be available.
- **Round Robin (RR):**
- - Allocates fixed time slices (time quantum) to each process in a circular manner.
- - Provides fairness and prevents a single long-running process from hogging the CPU.
- - Shorter time slices lead to more frequent context switches.
- **Priority Scheduling:**
- - Assigns a priority to each process, and the CPU is allocated to the highest priority process.
- - Can be either preemptive (higher priority process can interrupt a lower one) or non-preemptive.
- **Multilevel Queue Scheduling:**
- - Divides the ready queue into multiple priority queues, each with its own scheduling algorithm.
- - Higher-priority queues receive more CPU time, and lower-priority queues are serviced only when the higher-priority queues are empty.

- **Multilevel Feedback Queue Scheduling:**
- - A variation of multilevel queue scheduling that allows processes to move between queues based on their behaviour (e.g., aging, I/O-bound vs. CPU-bound).
- - Provides adaptability and dynamic priority adjustments.
- **Lottery Scheduling:**
- - Allocates "lottery tickets" to processes, and a random drawing determines the process to execute.
- - Provides a way to implement probabilistic scheduling and fairness.
- **Completely Fair Scheduler (CFS):**
- - Used in Linux, it aims to divide CPU time fairly among processes, using a red-black tree to maintain the execution order.
- - Achieves fairness and responsiveness in dynamic workloads.
- **Earliest Deadline First (EDF):**
- - Used in real-time systems, processes are scheduled based on their deadlines.
- - Ensures that processes with the earliest deadlines get CPU time, critical for real-time applications.

Ubuntu, like many other Linux distributions, uses the Completely Fair Scheduler (CFS) as its default process scheduler. The CFS is part of the Linux kernel's scheduler and aims to provide a fair and efficient allocation of CPU time to processes. It uses a red-black tree to maintain the execution order and strives to divide CPU time fairly among processes, ensuring that no process hogs the CPU.

Simulation for CFS:

Code:

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <pthread.h>
4.  #include <unistd.h>
5.  #include <time.h>
6.  #include <sched.h>
7.
8.  #define NUM_THREADS 40
9.  #define DATA_FILE "data.txt"
10. #define RESULT_FILE "results.txt"
11.
12. void *thread_function(void *arg) {
13.     int thread_id = *(int *)arg;
```

```
    int sum = 0;

    // Open the data file for reading
    FILE *data_file = fopen(DATA_FILE, "r");
    if (data_file == NULL) {
        perror("Failed to open data file");
        pthread_exit(NULL);
    }

    // Read integers from the data file and calculate the sum
    int num;
    while (fscanf(data_file, "%d", &num) == 1) {
        sum += num;
    }

    fclose(data_file);

    // Get the core number where the thread is running
    int core_num = sched_getcpu();

    // Open the results file for writing
    FILE *results_file = fopen(RESULT_FILE, "a");
    if (results_file == NULL) {
        perror("Failed to open results file");
        pthread_exit(NULL);
    }

    // Write the result, thread ID, and core number to the results file
    fprintf(results_file, "Thread %d (Core %d): Sum = %d\n", thread_id, core_num, sum);
    fclose(results_file);

    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];
    clock_t start, end;
    double cpu_time_used;

    start = clock();

    // Open the results file and clear its contents
    FILE *results_file = fopen(RESULT_FILE, "w");
    if (results_file == NULL) {
        perror("Failed to open results file");
        return 1;
    }
    fclose(results_file);

```

```
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        int result = pthread_create(&threads[i], NULL,
thread_function, &thread_ids[i]);
        if (result) {
            perror("Thread creation failed");
            return 1;
        }
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    end = clock();
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;

    printf("All threads have finished. Total execution time: %f
seconds.\n", cpu_time_used);

    return 0;
}
```

Terminal Window After Execution:

```
ashish_22bai1062@ubuntu:~$ cd OS_DA
ashish_22bai1062@ubuntu:~/OS_DA$ touch cfs.c
ashish_22bai1062@ubuntu:~/OS_DA$ touch data.c
ashish_22bai1062@ubuntu:~/OS_DA$ touch data.txt
ashish_22bai1062@ubuntu:~/OS_DA$ touch results.txt
ashish_22bai1062@ubuntu:~/OS_DA$ gcc cfs.c -o cfs -lpthread -lrt
cfs.c: In function ‘thread_function’:
cfs.c:32:20: warning: implicit declaration of function ‘sched_getcpu’; did you mean
 ‘sched_getparam’? [-Wimplicit-function-declaration]
   32 |     int core_num = sched_getcpu();
      |                    ^~~~~~~~~~~~
      |                    sched_getparam
ashish_22bai1062@ubuntu:~/OS_DA$ ./cfs
All threads have finished. Total execution time: 0.011944 seconds.
ashish_22bai1062@ubuntu:~/OS_DA$
```

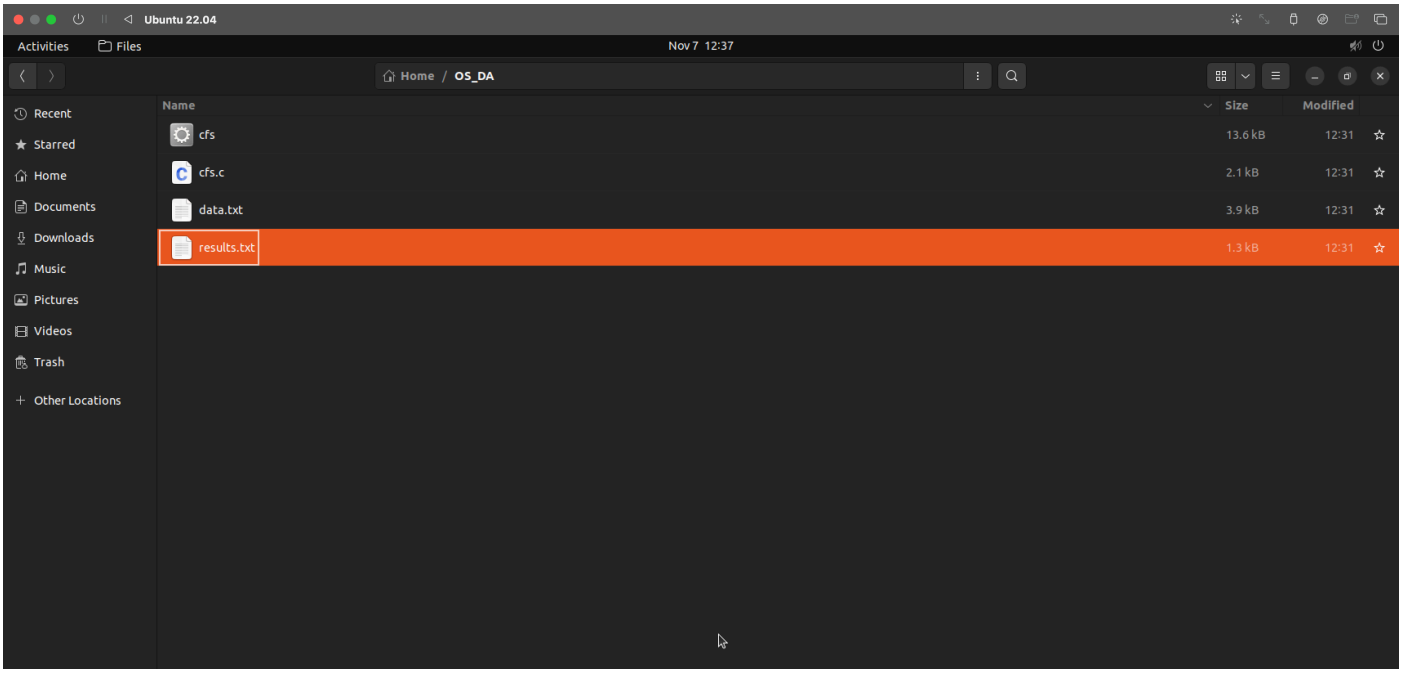
Data File:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51
52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135
136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171
172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207
208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243
244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279
280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315
316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351
352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387
388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423
424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459
460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495
496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531
532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567
568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603
604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639
640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675
676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711
712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747
748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783
784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819
820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855
856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891
892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927
928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963
964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999
```

Results File:

```
Thread 2 (Core 0): Sum = 499500
Thread 0 (Core 2): Sum = 499500
Thread 1 (Core 0): Sum = 499500
Thread 3 (Core 3): Sum = 499500
Thread 5 (Core 3): Sum = 499500
Thread 4 (Core 0): Sum = 499500
Thread 7 (Core 1): Sum = 499500
Thread 8 (Core 2): Sum = 499500
Thread 6 (Core 3): Sum = 499500
Thread 9 (Core 2): Sum = 499500
Thread 12 (Core 3): Sum = 499500
Thread 11 (Core 2): Sum = 499500
Thread 16 (Core 3): Sum = 499500
Thread 13 (Core 3): Sum = 499500
Thread 15 (Core 2): Sum = 499500
Thread 19 (Core 3): Sum = 499500
Thread 18 (Core 2): Sum = 499500
Thread 14 (Core 0): Sum = 499500
Thread 20 (Core 2): Sum = 499500
Thread 10 (Core 0): Sum = 499500
Thread 17 (Core 0): Sum = 499500
Thread 22 (Core 2): Sum = 499500
Thread 21 (Core 0): Sum = 499500
Thread 23 (Core 3): Sum = 499500
Thread 24 (Core 0): Sum = 499500
Thread 26 (Core 1): Sum = 499500
Thread 29 (Core 0): Sum = 499500
Thread 27 (Core 2): Sum = 499500
Thread 25 (Core 1): Sum = 499500
Thread 28 (Core 0): Sum = 499500
Thread 30 (Core 0): Sum = 499500
Thread 32 (Core 1): Sum = 499500
Thread 33 (Core 0): Sum = 499500
Thread 38 (Core 0): Sum = 499500
Thread 37 (Core 1): Sum = 499500
Thread 31 (Core 3): Sum = 499500
Thread 35 (Core 3): Sum = 499500
Thread 39 (Core 2): Sum = 499500
Thread 34 (Core 3): Sum = 499500
Thread 36 (Core 2): Sum = 499500
```

File System View:



A Real-Time Operating System (RTOS) is a specialized operating system designed for systems and applications that require precise and deterministic timing and rapid response to external events. Unlike general-purpose operating systems (such as Windows, Linux, or macOS), which are optimized for multitasking and may prioritize tasks based on various factors, RTOS is focused on meeting stringent timing requirements and ensuring that tasks are executed within specific time constraints. Here are some key characteristics and features of real-time operating systems:

1. **Deterministic Behavior:** RTOS systems are known for their deterministic behavior. They guarantee that tasks will be executed within predetermined time frames, making them suitable for applications where timing is critical. This predictability is crucial in fields like aerospace, automotive, industrial automation, and medical devices.

2. **Prioritization:** Real-time tasks are assigned priorities, and the scheduler ensures that higher-priority tasks are executed before lower-priority tasks. This priority-based scheduling helps meet timing deadlines.

3. **Hard Real-Time vs. Soft Real-Time:** RTOS can be categorized into hard real-time and soft real-time systems. In a hard real-time system, missing a deadline can have catastrophic consequences, while in a soft real-time system, occasional missed deadlines might be acceptable. Hard real-time systems demand more stringent timing guarantees.

4. **Task Synchronization and Communication:** RTOS provides mechanisms for tasks to synchronize and communicate with each other. Common synchronization tools include semaphores, mutexes, and message queues, which ensure that tasks do not interfere with each other.

5. **Interrupt Handling:** RTOS is equipped to efficiently handle hardware interrupts and manage real-time tasks in response to these events. This is crucial for systems where immediate response to external events is vital.

6. **Minimal Overhead:** RTOS is designed to have minimal overhead. This means that the operating system itself is lightweight and doesn't introduce significant delays or uncertainty into task execution.

7. **Resource Management:** Real-time operating systems manage resources efficiently. This includes memory allocation, CPU utilization, and other system resources to ensure that they are allocated optimally.

8. **Embedded Systems:** RTOS is commonly used in embedded systems, such as microcontrollers, where it is crucial to meet strict timing requirements in applications like automotive control systems, medical devices, robotics, and communication equipment.

9. **Task Constraints:** RTOS allows you to define timing constraints for tasks, such as maximum execution time, periodicity, and deadlines, to ensure that the system operates within the required parameters.

10. **Safety-Critical Applications:** Real-time operating systems are often used in safety-critical applications where reliability and predictability are paramount. This includes systems in the aerospace industry, nuclear reactors, and medical devices.

Some well-known real-time operating systems include:

- **FreeRTOS:** An open-source RTOS with a large user base and a wide range of supported microcontrollers.

- **VxWorks:** A commercial real-time operating system used in various critical applications, including aerospace and automotive.

- **QNX:** Known for its reliability and used in applications like automotive infotainment systems and medical devices.

- **RTOS-32:** An RTOS designed for Windows systems, providing real-time capabilities on Windows platforms.

RTOS plays a crucial role in ensuring that critical systems operate with high reliability and predictability. When choosing an RTOS for a specific application, factors like timing requirements, hardware compatibility, and licensing costs should be carefully considered.

CODE:

```
#include <stdio.h>
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"
#include "esp_timer.h"

#define NUM_TASKS 200
#define NUM_CORES 4
#define DATA_FILE "data.txt"
#define RESULT_FILE "results.txt"

// Define a mutex to protect the file access
SemaphoreHandle_t fileMutex;

```

```
15. void thread_function(void *pvParameters) {
16.     int task_id = (int)pvParameters;
17.     int sum = 0;
18. 
19.     // Record the start time
20.     int64_t start_time = esp_timer_get_time();
21. 
22.     // Open the data file for reading
23.     FILE *data_file = fopen(DATA_FILE, "r");
24.     if (data_file == NULL) {
25.         perror("Failed to open data file");
26.         vTaskDelete(NULL);
27.     }
28. 
29.     // Read integers from the data file and calculate
    the sum
30.     int num;
31.     while (fscanf(data_file, "%d", &num) == 1) {
32.         sum += num;
33.     }
34. 
35.     fclose(data_file);
36. 
37.     // Record the end time
38.     int64_t end_time = esp_timer_get_time();
39. 
40.     // Get the core number where the task is running
41.     int core_num = xPortGetCoreID();
42. 
43.     // Take the mutex to protect file access
44.     if (xSemaphoreTake(fileMutex, portMAX_DELAY) ==
    pdTRUE) {
45.         // Open the results file for writing
46.         FILE *results_file = fopen(RESULT_FILE, "a");
47.         if (results_file == NULL) {
48.             perror("Failed to open results file");
49.             vTaskDelete(NULL);
50.         }
51. 
52.         // Write the result, task ID, core number, and
    execution time to the results file
53.         fprintf(results_file, "Task %d (Core %d): Sum =
    %d, Start Time = %lld, End Time = %lld\n", task_id,
    core_num, sum, start_time, end_time);
54.         fclose(results_file);
```

```
55.
56.         // Release the mutex
57.         xSemaphoreGive(fileMutex);
58.     }
59.
60.     vTaskDelete(NULL);
61. }
62.
63. int main() {
64.     // Initialize the mutex for file access
65.     fileMutex = xSemaphoreCreateMutex();
66.
67.     if (fileMutex == NULL) {
68.         perror("Failed to create mutex");
69.         return 1;
70.     }
71.
72.     // Create FreeRTOS tasks on multiple cores
73.     TaskHandle_t tasks[NUM_TASKS];
74.     for (int i = 0; i < NUM_TASKS; i++) {
75.         int task_id = i;
76.         // Create tasks on different cores (cyclic
    assignment)
77.         int core_num = task_id % NUM_CORES;
78.         xTaskCreatePinnedToCore(thread_function,
    "Task", configMINIMAL_STACK_SIZE, (void *)task_id,
    tskIDLE_PRIORITY + 1, &tasks[i], core_num);
79.     }
80.
81.     // Start the FreeRTOS scheduler
82.     vTaskStartScheduler();
83.
84.     return 0;  // Should never reach here
85. }
```

Thank You!!