TY(IT)

Unix Operating System

Probable Assignment List for External Practical Exam

1. Write a program to transfer the contents of a file from the parent process to the child process using an **unnamed pipe**.
2. Implement a program using **named pipe (FIFO)** to allow one process to send input text and another process to receive and display it.
3. Write a program using a pipe filter that converts text from uppercase to lowercase before printing.
4. Write a program using **System V message queues** to send and receive messages between two processes.
5. Implement a program for a **chat application** between two processes using message queues.
6. Write a program where one process sends a sequence of numbers using message queues, and another process computes and prints their sum.
7. Write a program using **shared memory** (`shmget`, `shmat`, `shmdt`) where the first process writes a string and the second process reads it.
8. Implement a shared memory program in which:
    a. Process 1 takes numbers as input from the user,
    b. Process 2 sorts the numbers,
    c. Process 3 displays the sorted list.
9. Write a program where one process writes characters A–Z into shared memory, and another process writes the same data into a file.
10. Write a program using **semaphores** to synchronize two processes such that one process prints even numbers and the other prints odd numbers in order.
11. Implement the **Producer–Consumer problem** using shared memory and semaphores.
12. Implement the **Reader–Writer problem** using semaphores to control access.
13. Write a **TCP client–server program** in C where the client sends a message and the server replies with the reversed string.
14. Implement a **UDP client–server program** where the client sends numbers and the server responds with their factorial.
15. Write a program to implement an **Echo Server** using TCP sockets (both iterative and concurrent versions).
16. **Job scheduling system**: Write a program where a client submits jobs (e.g., factorial, Fibonacci, prime check) via message queues, and a server process executes and returns results.
17. **Priority-based messaging**: Implement a system where multiple clients send messages with different priorities, and the server displays them in priority order.
18. **Shared memory calculator**: One process writes two operands and an operator into shared memory. Another process reads the request, performs the calculation, and writes back the result.

19. **Multi-process shared memory sort**:
    a. Process 1 writes an array of numbers to shared memory.
    b. Process 2 sorts the numbers.
    c. Process 3 computes the median and mean.
    d. Process 4 displays the final results. Synchronize using semaphores.
20. **Shared memory file transfer**: Implement a program where one process reads data from a file and writes into shared memory in chunks, and another process reconstructs the file.
21. **Dining philosophers problem**: Implement the classical dining philosophers synchronization problem using semaphores.
22. **Producer–Consumer with bounded buffer**: Use shared memory and semaphores to implement multiple producers and multiple consumers.
23. **Reader–Writer with priority**: Implement the reader–writer problem using semaphores such that writers are given higher priority than readers.
24. **Concurrent TCP echo server**: Implement an echo server using TCP sockets that handles multiple clients concurrently using `fork()` or threads.
25. **UDP file transfer**: Implement a client-server application where the client requests a file and the server sends it via UDP in chunks, with acknowledgments.
26. **Distributed computation server**: Create a TCP server that accepts computation requests (factorial, matrix multiplication, string reversal) from multiple clients and returns results.
27. **Multi-user chat server**: Implement a group chat system using TCP sockets where multiple clients can join, send messages, and receive broadcasts from the server.
28. Implement a program that uses the `fork()` system call to create five child processes and assigns a distinct operation to each child.
29. Implement a program using `vfork()` where the child reads a login name and the parent reads the password.
30. Write a program that launches an application using the `fork()` system call.
31. Write a program that launches an application using the `vfork()` system call.
32. Demonstrate the use of `wait()` with `fork()` by writing a program that shows parent-child synchronization.
33. Write a program to illustrate different variants of the `exec()` family of system calls.
34. Create a program that demonstrates `exit()` combined with `wait()` and `fork()` (showing how children terminate and how parents collect status).
35. Write a program that uses `kill()` to send signals between two unrelated processes.
36. Write a program that uses `kill()` to send signals between related processes (created with `fork()`).
37. Implement a program that uses `alarm()` and signal handling to require user input within a specified time limit.
38. Create an alarm clock program using `alarm()` and signal handlers.
39. Write a program that reports file statistics using `stat()` (include important fields such as file access permissions, file type, etc.).
40. Write a program that reports file statistics using `fstat()` (include important fields such as file access permissions, file type, etc.).
41. Develop a multithreaded chat application in Java or C.
42. Create a program that spawns three threads: one prints even numbers, another prints odd numbers, and the third prints prime numbers.
43. Write a multithreaded program on Linux that uses the `pthread` library.
44. Implement the producer–consumer problem using multithreading in Java.

45. Write a shell script that implements a simple calculator.
46. Implement a digital clock using a shell script.
47. Write a shell script that checks whether the system is connected to a network by using the `ping` command.
48. Write a shell script to sort ten given numbers in ascending order.
49. Create a program (or script) that prints “Hello World” with bold, blinking, and colored (red, blue, etc.) text effects.
50. Write a shell script that checks whether a specified file exists in a given folder or drive.
51. Create a shell script that displays disk partitions, their sizes, and disk usage/free space.
52. Write a shell script to locate a given file on the system using `find` or `locate`.
53. Write a shell script that downloads a webpage from a given URL using `wget`.
54. (Duplicate) Write a shell script to download a webpage from a given URL using `wget`.
55. Write a shell script that displays system users (using `finger` or `who`).
56. Implement a recursive Python function that generates prime numbers up to a given limit (limit passed as a parameter).
57. Write a program that prints a pyramid with the number of lines supplied by the user; center the pyramid as closely as possible. (Hint: nested loops)
58. Given a text file, write code to count word frequencies in the file (use file handling).
59. Generate a frequency list of all shell commands you’ve used and display the top five commands with their counts (hint: use `history`/`hist`).
60. Write a shell script that takes a filename and checks whether it is executable; then modify it to remove execute permission if the file is executable.
61. Produce a word-frequency list for `wonderland.txt` (hint: use `grep`, `tr`, `sort`, `uniq`, or similar tools).
62. Create a bash script that accepts two or more arguments where each argument is a filename; if fewer than two arguments are given print an error; if any file does not exist print an error; otherwise concatenate the files.
63. Write a shell script to download a file from a remote machine using `lftp` with `get`/`mget`.
64. Implement the producer–consumer problem using semaphores (`semaphore.h`) in C or Java.
65. Implement the reader–writer problem using semaphores.
66. Write a program to demonstrate IPC via message queues (`msgget`, `msgsnd`, `msgrcv`) for chatting between two or three users.
67. Implement IPC using shared memory (`shmget`, `shmat`, `shmdt`) where one process sends A–Z or 1–100 (from user input) and another process receives it.
68. Implement IPC via shared memory where one process reads A–Z or 1–100 from a file and writes it into shared memory, and another process reads from shared memory and writes to a different file (same directory, different filename).
69. Write a shared-memory IPC program where: Process 1 takes numbers from the user and writes to shared memory, Process 2 sorts the numbers and writes them back, and Process 3 displays the sorted data.
70. Create programs where different processes perform different shared-memory operations: create, delete, attach, detach (`shmget`, `shmat`, `shmdt`).
71. Implement programs that simulate common Linux commands such as `cat`, `ls`, `cp`, `mv`, `head`, etc.
72. Write a program using semaphores to ensure function `f1()` executes before `f2()` (example: prompt for username before prompting for password).

73. Write two programs that exchange messages to form the dialog:
74. Process 1 sends `"Hi?"`
75. Process 2 receives and replies `"Hello"`
76. Process 1 receives the reply and then sends `"I am fine"` Use System V message queues (`msgget`, `msgsnd`, `msgrcv`).
77. Implement a TCP program demonstrating socket system calls in C or Python.
78. Implement a UDP program demonstrating socket system calls in C or Python.
79. Implement an echo server over TCP using iterative and/or concurrent logic.
80. Implement an echo server over UDP using iterative and/or concurrent logic.
81. Write a program using an unnamed pipe to send data from parent to child.
82. Write a program using an unnamed pipe to send a file from parent to child.
83. Write a program using a pipe that acts as a filter to convert uppercase text to lowercase (reading from a command or a file).
84. Illustrate semaphore usage with `fork()` so two processes run simultaneously and coordinate via semaphores; include a short note on differences between `sem.h` and `semaphore.h`.
85. Create three separate programs: (1) initialize a semaphore and display its ID, (2) perform the P (wait) operation and print a message, (3) perform the V (signal) operation and print a message — all operating on the same semaphore.
86. Write a program to demonstrate file locking using `lockf()`.
87. Write a program to demonstrate file locking using `flock()`.