1. Fork Program (`1.fork.c`)

Concept:

- **fork()**: This system call is used to create a new process by duplicating the calling process. After a successful `fork()`, the parent and child processes continue executing from the point of the `fork()`, with different process IDs.

Code Walkthrough:

```
c
Copy code
#include <stdio.h>
#include <unistd.h>

int main()
{
    printf("Before FORK \\n");
    fork();  // Creates a child process.
    printf("After FORK \\n\\n");  // Both parent and child will execute this line.
    return 0;
}
```

- `fork()`: The `fork` system call creates a new process. After `fork()`, both the parent and child processes continue from the same place in the code.
- **Result**: The output will contain the "After FORK" message twice, once for the parent process and once for the child process.

Viva Questions:

1. **What does `fork()` do?** `fork()` creates a new process by duplicating the calling process.
2. **What is the return value of `fork()`?** It returns 0 in the child process and the child's PID in the parent process.

---

2. Exec Program (`2.exec.c`)

Concept:

- **execv()**: This function replaces the current process with a new program. The current program is entirely replaced and does not continue.

Code Walkthrough:

```
c
Copy code
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    char *args[] = {"/bin/ls", NULL};  // Arguments for execv, ending with NULL.
    execv("/bin/ls", args);  // Replaces current process with "ls".

    perror("execv failed");  // This line is executed only if execv fails.
    return 0;
}
```

- **execv()**: Replaces the current process with the `/bin/ls` command. If successful, the current process does not continue; if it fails, it prints an error message.

Viva Questions:

1. **What is the role of `execv()`?** It replaces the current process image with a new one, in this case, the `ls` command.
2. **Why does the `perror` function execute only when `execv` fails?** If `execv` is successful, it never returns, as the current process is completely replaced by the new one.

---

3. Getpid Program (`3.getpid.c`)

Concept:

- **getpid()**: This system call returns the process ID of the calling process.

Code Walkthrough:

```
c
Copy code
#include<stdio.h>
#include <unistd.h>
#include<sys/types.h>

int main()
{
    printf("\\n parent process id %d", getppid());  // Get parent process ID
    printf("\\n child process id %d\\n", getpid());  // Get current process ID
}
```

- **getpid()**: Returns the process ID of the calling process.
- **getppid()**: Returns the parent process ID.

Viva Questions:

1. **What does `getpid()` return?** It returns the process ID of the current process.
2. **What is the difference between `getpid()` and `getppid()`?** `getpid()` returns the current process ID, while `getppid()` returns the parent process ID.

---

4. Exit Program (`4.exit.c`)

Concept:

- **exit()**: This function terminates the calling process immediately.

Code Walkthrough:

```
c
Copy code
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
```

```
int main() {
    int p = fork();
    if (p == 0) {
        printf("\\nChild process created\\n");
        exit(0);  // Terminates the child process.
    } else if (p < 0) {
        printf("Failed to create child process\\n");
        exit(-1);  // If fork failed, terminate with an error code.
    }
    return 0;
}
```

- **fork()**: Creates a child process.
- **exit()**: Terminates the child process immediately.

Viva Questions:

1. **What does `exit(0)` mean?** It indicates that the process terminated successfully.
2. **What happens when `exit()` is called?** The process is immediately terminated.

5. Wait Program (`5.wait.c`)

Concept:

- **wait()**: This system call makes the parent process wait until all of its child processes have terminated. It also retrieves the exit status of the terminated child.

Code Walkthrough:

```
c
Copy code
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    int pid, i = 0;
    printf("Ready to fork\\n");
    pid = fork();
    if (pid == 0) {  // Child process
        printf("Child process starts\\n");
        for (i = 0; i < 10; i++) {
            printf("Child process running %d\\n", i);
            sleep(1);  // Sleep for 1 second.
        }
        printf("Child process ends\\n");
    } else {  // Parent process
        wait(NULL);  // Wait for the child process to finish.
        for (i = 0; i < 10; i++) {
            printf("Parent process running %d\\n", i);
            sleep(1);  // Sleep for 1 second.
        }
        printf("Parent process ends\\n");
    }
    return 0;
}
```

- **fork()**: Creates a child process.
- **wait()**: The parent waits for the child process to complete. After the child process finishes, the parent resumes execution.

Viva Questions:

1. **What is the role of `wait()` in a parent process?** It makes the parent wait for its child process to terminate and retrieves its exit status.
2. **Why do we use `sleep()` in this program?** It is used to introduce a delay between each iteration, simulating work done by both parent and child processes.

---

6. Open/Close File Program (`6.open,close.c`)

Concept:

- **open()**: Opens a file descriptor for a given file.
- **close()**: Closes an open file descriptor.

Code Walkthrough:

```
c
Copy code
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    int fd, i;
    fd = open("test.txt", O_CREAT | O_RDWR | O_APPEND, 0644);  // Open or create
"test.txt".
    if (fd < 0) {
        perror("Failed to open file");
        return 1;
    }
    for (i = 0; i < 5; i++)
        write(fd, "CVSR\\n", 5);  // Write "CVSR" 5 times to the file.
    close(fd);  // Close the file descriptor.
    return 0;
}
```

- **open()**: Opens the file "test.txt" with the flags `O_CREAT` (create if it doesn't exist), `O_RDWR` (read and write), and `O_APPEND` (append to the file).
- **close()**: Closes the file descriptor once writing is complete.

Viva Questions:

1. **What does `open()` return?** It returns a file descriptor, a non-negative integer used for subsequent file operations.
2. **Why is `close()` important after opening a file?** To free up system resources associated with the open file.

---

7. Stat Program (`7.stat.c`)

Concept:

- **stat()**: Retrieves file information such as size, permissions, and other metadata.

Code Walkthrough:

```
c
Copy code
#include <stdio.h>
#include <sys/stat.h>

int main(int argc, char *argv[]) {
    struct stat fileStat;
    if (argc != 2 || stat(argv[1], &fileStat) < 0) return 1;

    printf("File: %s\\nSize: %ld bytes\\nPermissions: %o\\n", argv[1],
fileStat.st_size, fileStat.st_mode & 0777);
    return 0;
}
```

- **stat()**: This system call retrieves file attributes such as size, permissions, and more. It stores this information in the `fileStat` structure.
- **argc/argv**: The program expects a filename as an argument.

Viva Questions:

1. **What does `stat()` return?** It returns 0 on success and -1 on failure, filling the `stat` structure with file information on success.
2. **What does `st_mode` represent?** It represents the file permissions and mode (e.g., read, write, execute permissions).

---

8. Open/Close Directory Program (`8.opendir-closedir.c`)

Concept:

- **opendir()**: Opens a directory stream for reading its contents.
- **readdir()**: Reads directory entries.
- **closedir()**: Closes the directory stream.

Code Walkthrough:

```
c
Copy code
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <dirent.h>

int main(int argc, char *argv[]) {
    DIR *dp;
    struct dirent *dirp;

    if (argc != 2) {
        printf("A single argument (the directory name) is required.\\n");
        exit(1);
    }
    if ((dp = opendir(argv[1])) == NULL) {  // Open the directory specified in argv[1].
        printf("Cannot open directory: %s\\n", argv[1]);
        exit(1);
    }
    while ((dirp = readdir(dp)) != NULL) {  // Read the directory entries one by one.
        printf("Name: %s | Inode: %ld\\n", dirp->d_name, dirp->d_ino);  // Print file
name and inode.
    }
    closedir(dp);  // Close the directory stream.
    return 0;
}
```

- **opendir()**: Opens a directory stream for reading its contents.
- **readdir()**: Reads the next directory entry in the stream.
- **closedir()**: Closes the directory stream.

Viva Questions:

1. **What is the purpose of `opendir()`?** It opens a directory stream that allows reading the directory entries using `readdir()`.
2. **How does `readdir()` work?** It reads each entry in the directory stream one by one.

---

1. FCFS Scheduling Program (`fcfs.c`)

Concept:

- **First-Come, First-Serve (FCFS)**: A scheduling algorithm where the process that arrives first is executed first. It is non-preemptive and simple to implement.

Code Walkthrough:

```
c
Copy code
#include<stdio.h>

int main() {
    int bt[20], wt[20], tat[20], i, n;
    float wtavg, tatavg;

    printf("\\nEnter the number of processes -- ");
    scanf("%d", &n);  // Read number of processes.

    for(i=0; i<n; i++) {
        printf("\\nEnter Burst Time for Process %d -- ", i);
        scanf("%d", &bt[i]);  // Input burst time for each process.
    }

    wt[0] = wtavg = 0;  // Waiting time for the first process is 0.
    tat[0] = tatavg = bt[0];  // Turnaround time for the first process is its burst 
time.

    for(i=1; i<n; i++) {
        wt[i] = wt[i-1] + bt[i-1];  // Waiting time is cumulative burst time of 
previous processes.
        tat[i] = tat[i-1] + bt[i];  // Turnaround time includes current process's burst 
time.
        wtavg = wtavg + wt[i];  // Sum up waiting times.
        tatavg = tatavg + tat[i];  // Sum up turnaround times.
    }

    printf("\\t PROCESS \\tBURST TIME \\t WAITING TIME\\t TURNAROUND TIME\\n");
    for(i = 0; i < n; i++)
        printf("\\n\\t P%d \\t\\t %d \\t\\t %d \\t\\t %d", i, bt[i], wt[i], tat[i]);

    printf("\\nAverage Waiting Time -- %f", wtavg/n);
    printf("\\nAverage Turnaround Time -- %f", tatavg/n);

    return 0;
}
```

- **Burst Time**: Time required by a process for execution.
- **Waiting Time**: Time a process spends waiting in the ready queue before its execution.
- **Turnaround Time**: Total time taken from submission to completion.

Viva Questions:

1. **What is FCFS scheduling?** FCFS executes processes in the order they arrive. It is non-preemptive.
2. **How do you calculate the waiting time in FCFS?** The waiting time is the sum of the burst times of all previous processes.

---

2. SJF Scheduling Program (`sjf.c`)

Concept:

- **Shortest Job First (SJF)**: A scheduling algorithm where the process with the smallest burst time is executed first. It can be either preemptive or non-preemptive. In this case, the non-preemptive version is implemented.

Code Walkthrough:

```
c
Copy code
#include<stdio.h>

int main() {
    int p[20], bt[20], wt[20], tat[20], i, k, n, temp;
    float wtavg, tatavg;

    printf("\\nEnter the number of processes -- ");
    scanf("%d", &n);  // Read number of processes.

    for(i=0; i<n; i++) {
        p[i] = i;  // Process index for identification.
        printf("Enter Burst Time for Process %d -- ", i);
        scanf("%d", &bt[i]);  // Input burst time for each process.
    }

    // Sort processes based on burst time
    for(i=0; i<n; i++) {
        for(k=i+1; k<n; k++) {
            if(bt[i] > bt[k]) {  // Sort by burst time (ascending)
                temp = bt[i];  // Swap burst times
                bt[i] = bt[k];
                bt[k] = temp;

                temp = p[i];  // Swap process numbers
                p[i] = p[k];
                p[k] = temp;
            }
        }
    }

    wt[0] = wtavg = 0;  // First process has no waiting time.
    tat[0] = tatavg = bt[0];  // Turnaround time for the first process is its burst
time.

    for(i=1; i<n; i++) {
        wt[i] = wt[i-1] + bt[i-1];  // Cumulative burst time for waiting time.
        tat[i] = tat[i-1] + bt[i];  // Turnaround time includes current burst time.
        wtavg = wtavg + wt[i];  // Sum up waiting times.
        tatavg = tatavg + tat[i];  // Sum up turnaround times.
    }

    printf("\\n\\t PROCESS \\tBURST TIME \\t WAITING TIME\\t TURNAROUND TIME\\n");
    for(i = 0; i < n; i++)
        printf("\\n\\t P%d \\t\\t %d \\t\\t %d \\t\\t %d", p[i], bt[i], wt[i], tat[i]);

    printf("\\nAverage Waiting Time -- %f", wtavg/n);
    printf("\\nAverage Turnaround Time -- %f", tatavg/n);

    return 0;
}
```

- **Sorting**: The processes are sorted based on their burst time.
- **SJF Optimality**: It gives minimum average waiting time for a given set of processes, but is difficult to implement in real-time systems because of the need to predict burst time.

Viva Questions:

1. **What is the SJF scheduling algorithm?** SJF selects the process with the smallest burst time to execute next.
2. **Why is SJF optimal?** It minimizes the average waiting time compared to other algorithms.

---

3. Priority Scheduling Program (`priority.c`)

Concept:

- **Priority Scheduling**: A scheduling algorithm where each process is assigned a priority, and the process with the highest priority is executed first. Lower priority numbers denote higher priority.

Code Walkthrough:

```
c
Copy code
#include <stdio.h>

int main() {
    int p[20], bt[20], pri[20], wt[20], tat[20], i, k, n, temp;
    float wtavg, tatavg;

    printf("Enter the number of processes --- ");
    scanf("%d", &n);  // Read number of processes.

    for (i = 0; i < n; i++) {
        p[i] = i;  // Process index.
        printf("Enter the Burst Time & Priority of Process %d --- ", i);
        scanf("%d %d", &bt[i], &pri[i]);  // Input burst time and priority for each
process.
    }

    // Sort processes based on priority (lower number means higher priority).
    for (i = 0; i < n; i++) {
        for (k = i + 1; k < n; k++) {
            if (pri[i] > pri[k]) {  // Sort by priority (ascending).
                temp = p[i];  // Swap process number.
                p[i] = p[k];
                p[k] = temp;

                temp = bt[i];  // Swap burst time.
                bt[i] = bt[k];
                bt[k] = temp;

                temp = pri[i];  // Swap priority.
                pri[i] = pri[k];
                pri[k] = temp;
            }
        }
    }

    wt[0] = wtavg = 0;  // First process has no waiting time.
    tat[0] = tatavg = bt[0];  // Turnaround time for the first process is its burst
time.

    for (i = 1; i < n; i++) {
        wt[i] = wt[i-1] + bt[i-1];  // Waiting time based on cumulative burst time.
        tat[i] = tat[i-1] + bt[i];  // Turnaround time includes current burst time.
        wtavg = wtavg + wt[i];  // Sum up waiting times.
        tatavg = tatavg + tat[i];  // Sum up turnaround times.
    }

    printf("\\nPROCESS\\tPRIORITY\\tBURST TIME\\tWAITING TIME\\tTURNAROUND TIME\\n");
    for (i = 0; i < n; i++)
        printf("%d\\t\\t%d\\t\\t%d\\t\\t%d\\t\\t%d\\n", p[i], pri[i], bt[i], wt[i],
tat[i]);

    printf("\\nAverage Waiting Time is --- %f", wtavg/n);
    printf("\\nAverage Turnaround Time is --- %f", tatavg/n);

    return 0;
}
```

- **Sorting by Priority**: The processes are sorted based on priority, with lower numbers indicating higher priority.
- **Priority Scheduling Drawback**: It can lead to starvation if low-priority processes are continually ignored.

Viva Questions:

1. **What is priority scheduling?** Processes are executed based on their priority; higher priority processes are executed first.
2. **What is starvation in priority scheduling?** Starvation occurs when low-priority processes wait indefinitely due to the continuous arrival of high-priority processes.

Producer-Consumer Problem (`producerconsumer.c`)

Concept:

- **Producer-Consumer Problem**: A classical synchronization problem where producers generate data and add it to a buffer, and consumers remove data from the buffer. The producer should wait if the buffer is full, and the consumer should wait if the buffer is empty.

Code Walkthrough:

```
c
Copy code
#include <stdio.h>
#include <stdlib.h>
#define BUFFERSIZE 10  // Size of the buffer

int mutex = 1;  // Mutex for mutual exclusion
int empty = BUFFERSIZE;  // Number of empty slots in the buffer
int full = 0;  // Number of full slots in the buffer
int buffer[BUFFERSIZE];  // Buffer array to store produced items
int in = 0, out = 0;  // `in` is the index where producer inserts, `out` is the index
where consumer removes
int n;  // Number of items to produce and consume

// Wait function (P operation)
void wait(int *s) {
    while (*s <= 0);  // Busy wait if semaphore value is <= 0
    (*s)--;  // Decrease the semaphore value
}

// Signal function (V operation)
void signal(int *s) {
    (*s)++;  // Increase the semaphore value
}

// Producer function
void producer() {
    int item;
    if (in < n) {  // Only produce if there are remaining items to produce
        wait(&empty);  // Wait if buffer is full
        wait(&mutex);  // Lock the buffer (mutex)
        printf("\\nEnter an item: ");
        scanf("%d", &item);  // Read an item from the producer (user input)
        buffer[in] = item;  // Insert the item into the buffer
        printf("Produced item: %d at position %d\\n", item, in);  // Output produced
item
        in = (in + 1) % BUFFERSIZE;  // Move to the next buffer position
        signal(&mutex);  // Release the buffer (mutex)
        signal(&full);  // Increase the full count (signal that a new item is in the
buffer)
    }
}

// Consumer function
void consumer() {
    if (out < in) {  // Only consume if there are items in the buffer
        wait(&full);  // Wait if buffer is empty
        wait(&mutex);  // Lock the buffer (mutex)
        int item1 = buffer[out];  // Remove the item from the buffer
        printf("Consumed item: %d from position %d\\n", item1, out);  // Output
consumed item
        out = (out + 1) % BUFFERSIZE;  // Move to the next buffer position
        signal(&mutex);  // Release the buffer (mutex)
        signal(&empty);  // Increase the empty count (signal that a slot is free)
    }
}

int main() {
    printf("Enter the number of items to produce and consume: ");
    scanf("%d", &n);  // Number of items to produce and consume

    // Produce items
    while (in < n) {
        producer();
    }

    // Consume items
    while (out < in) {
        consumer();
    }

    return 0;
}
```

Explanation:

- **Buffer Size**: The buffer has a fixed size defined by `BUFFERSIZE` (10 slots in this case).
- **Mutex**: Used to ensure mutual exclusion when the producer or consumer accesses the shared buffer.
- **empty/full**: Semaphores used to track how many empty and full slots are available in the buffer.
- **Producer**: Adds an item to the buffer if there are empty slots. It waits on the `empty` semaphore (if the buffer is full) and the `mutex` to ensure exclusive access to the buffer.
- **Consumer**: Removes an item from the buffer if there are full slots. It waits on the `full` semaphore (if the buffer is empty) and the `mutex` to ensure exclusive access to the buffer.

Viva Questions:

1. **What is the Producer-Consumer problem?** The Producer-Consumer problem is a synchronization problem where producers add items to a shared buffer, and consumers remove items. It ensures proper synchronization between producers and consumers.
2. **What is a semaphore?** A semaphore is a variable used to control access to a common resource by multiple processes in a concurrent system, and it helps in avoiding critical section problems.
3. **Why do we need mutual exclusion in the Producer-Consumer problem?** To prevent the producer and consumer from accessing the buffer at the same time, which could lead to race conditions.
4. **What are the conditions for the producer to add an item?** The producer adds an item if there is at least one empty slot in the buffer (`empty > 0`).
5. **What are the conditions for the consumer to remove an item?** The consumer removes an item if there is at least one full slot in the buffer (`full > 0`).
6. **How does `wait()` work in the context of semaphores?** `wait()` (also called P operation) decreases the semaphore if its value is greater than 0. If the semaphore is 0 or less, it blocks the process until the semaphore becomes positive.
7. **What is the role of `signal()`?** `signal()` (also called V operation) increases the semaphore's value, effectively signaling that a resource is now available.

---

1. Fork Program (fork.c)

- **Key Concept**: `fork()` creates a new process by duplicating the calling process. The process ID (PID) is different for parent and child.
- **Important Function**: `fork()`
- **Output**: Parent and child processes both run the next code statement.

Mnemonic:

"Fork creates a Fork in the road."

- Think of two paths (parent and child processes) that start together but run independently after the fork.

---

2. Exec Program (exec.c)

- **Key Concept**: `execv()` replaces the current process with a new program. Once replaced, the original process does not continue.
- **Important Function**: `execv()`
- **Output**: The current process is replaced by the new one (`ls` in this case).

Mnemonic:

"Execv = Execute and Vanish."

- The current process *vanishes* and is replaced by the new process (like a magician's trick).

---

3. Getpid Program (getpid.c)

- **Key Concept**: `getpid()` gets the process ID of the current process; `getppid()` gets the parent process ID.
- **Important Functions**: `getpid()`, `getppid()`

Mnemonic:

"Get PIDs to identify who you are and who your parent is."

- "PID" stands for Process ID, so `getpid()` identifies the process itself, and `getppid()` identifies its parent.

---

4. Exit Program (exit.c)

- **Key Concept**: `exit()` terminates the current process. The child process created by `fork()` can be terminated using `exit(0)`.
- **Important Function**: `exit()`
- **Output**: Child process exits after printing.

Mnemonic:

"Exit closes the door."

- Just as you exit a room and close the door, `exit()` closes the process.

---

5. Wait Program (wait.c)

- **Key Concept**: `wait()` makes the parent process wait until the child process completes execution.
- **Important Function**: `wait()`
- **Output**: Parent waits until child finishes execution before continuing.

Mnemonic:

"Wait for the Child to finish."

- Parents wait for their children, and in this case, the parent process waits for the child process.

---

6. Open/Close File Program (open,close.c)

- **Key Concept**: `open()` opens a file, `write()` writes to it, and `close()` closes the file.
- **Important Functions**: `open()`, `write()`, `close()`

Mnemonic:

"Open, Write, Close — like writing a letter."

- You open a file, write data, and then close the file, just like writing a letter and putting it in an envelope.

---

7. Stat Program (stat.c)

- **Key Concept**: `stat()` retrieves file information such as file size, permissions, and more.
- **Important Function**: `stat()`

Mnemonic:

"Stat collects file stats."

- Just like statistics gather information, `stat()` collects information about files.

---

8. Open/Close Directory Program (opendir-closedir.c)

- **Key Concept**: `opendir()` opens a directory, `readdir()` reads the directory entries, and `closedir()` closes it.
- **Important Functions**: `opendir()`, `readdir()`, `closedir()`

Mnemonic:

"Open, Read, Close — exploring a folder."

- Think of opening a folder, reading its contents, and then closing it, just like browsing files on your computer.

---

9. FCFS Scheduling (fcfs.c)

- **Key Concept**: First-Come, First-Serve (FCFS) scheduling executes processes in the order they arrive.
- **Key Formulas**:
  - **Waiting Time** = Sum of previous burst times.
  - **Turnaround Time** = Waiting Time + Burst Time.

Mnemonic:

"First Arrive, First Served"

- Just like standing in line, the first process to arrive is the first to be executed.

---

10. SJF Scheduling (sjf.c)

- **Key Concept**: Shortest Job First (SJF) scheduling executes processes with the shortest burst time first.
- **Key Formulas**:
  - **Waiting Time** = Sum of previous shorter burst times.
  - **Turnaround Time** = Waiting Time + Burst Time.

Mnemonic:

"Shortest Job gets served first"

- The smallest task gets priority, like choosing a quick errand before a long one.

---

11. Priority Scheduling (priority.c)

- **Key Concept**: Processes are executed based on priority (lower number means higher priority).
- **Key Formulas**:
  - **Waiting Time** = Sum of previous high-priority burst times.
  - **Turnaround Time** = Waiting Time + Burst Time.

Mnemonic:

"High Priority, High Speed"

- Just like in real life, high-priority tasks get completed first.

---

12. Producer-Consumer Problem (producerconsumer.c)

- **Key Concept**: A synchronization problem where producers add items to a buffer and consumers remove items. The producer must wait if the buffer is full, and the consumer must wait if it is empty.
- **Important Functions**: `wait()`, `signal()`

Mnemonic:

"Producer fills, Consumer eats"

- Think of a producer like a chef filling a tray with food, and the consumer eating it. If the tray is full, the chef waits, and if it's empty, the consumer waits.

General Mnemonic/Shortcut to Remember System Calls:

FORK-WAIT-EXEC-EXIT-GETPID

1. **F**ork - Creates new process.
2. **W**ait - Parent waits for the child process.
3. **E**xec - Replaces the process.
4. **E**xit - Terminates the process.
5. **G**etpid - Retrieves process ID.

Think of **"FWEEG"** as the basic sequence of a process's lifecycle.

Viva Quick Summary Points:

1. **fork()** creates a child process.
2. **exec()** replaces the current process.
3. **getpid()** retrieves process ID.
4. **exit()** terminates a process.
5. **wait()** makes the parent wait for the child process.
6. **open()** and **close()** deal with file I/O.
7. **stat()** retrieves file metadata.
8. **opendir()** and **readdir()** manage directories.
9. **FCFS** and **SJF** are scheduling algorithms, SJF being optimal for reducing waiting time.
10. **Producer-Consumer** solves synchronization with semaphores (`wait()` and `signal()`).