

CSE3241: Operating Systems

Lab Test

Name: Md Sajjad Hossain

ID: 2011176125

Session: 2019-20

May 25, 2024

Question 1

Solution Code of the problem:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
    pid_t child1, child2, child3;
    int a = 10, b = 5;

    printf("Parent Process (PID: %d) started.\n", getpid());

    child1 = fork();
    if (child1 == 0) {
        printf("Child-1 (PID: %d) performing addition: %d + %d = %d\n",
            getpid(), a, b, a + b);
        exit(0);
    } else {
        child2 = fork();
        if (child2 == 0) {
            printf("Child-2 (PID: %d) performing subtraction: %d - %d = %d\n",
                getpid(), a, b, a - b);
            exit(0);
        } else {
            child3 = fork();
            if (child3 == 0) {
                printf("Child-3 (PID: %d) performing multiplication: %d * %d = %d\n",
                    getpid(), a, b, a * b);
                exit(0);
            } else {
                for (size_t i = 0; i < 1000000000; i++){

                }
                waitpid(child1, NULL, 0);
                waitpid(child2, NULL, 0);
                waitpid(child3, NULL, 0);

                printf("Parent Process (PID: %d) completed.\n", getpid());
            }
        }
    }
}
```

```

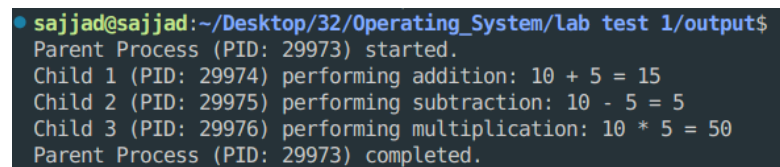
    }
}

}
return 0;
}

```

Explanation:

In this code, we are creating 3 child process without any grandchildren process. This is done by calling `fork()` every time from the parent of its child process avoiding any grandchildren process. Each child process is doing their specific task mentioned on the question like addition, multiplication, and subtraction. parent collecting the child termination report by `waitpid()`. Then main parent is terminated by printing a message. This is how this code works.



```

sajjad@sajjad:~/Desktop/32/Operating_System/lab test 1/output$ .
Parent Process (PID: 29973) started.
Child 1 (PID: 29974) performing addition: 10 + 5 = 15
Child 2 (PID: 29975) performing subtraction: 10 - 5 = 5
Child 3 (PID: 29976) performing multiplication: 10 * 5 = 50
Parent Process (PID: 29973) completed.

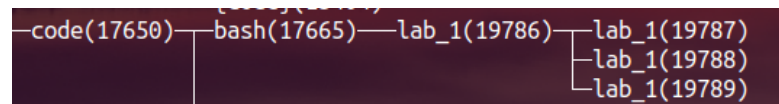
```

Figure 1: output of the code

The child processes are doing addition, subtraction and multiplication and the result are shown in the output.

By giving the command `ps tree -p` we can see the process tree and trace the parent and child processes on the process tree.

Trace of the parent and child processes in the process tree:



```

code(17650)---bash(17665)---lab_1(19786)---lab_1(19787)
                                           |
                                           |---lab_1(19788)
                                           |
                                           |---lab_1(19789)

```

Figure 2: The parent and child processes in the process tree

Question 2

Code to create a orphan process is given bellow:

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

int main() {
    pid_t pid = fork();
    if (pid > 0) {
        printf("Exiting - Parent - Process.\n");
        //sleep(5);
    }
}

```

```

        exit(0);
    } else if (pid == 0) {
        sleep(10);
        printf("Child process became an orphan (PPID: %d).\n", getppid());
    }
    return 0;
}

```

orphan process: An orphan process is a process whose parent has terminated or exited, leaving it without calling wait() system call. In Unix-like operating systems, when a process becomes an orphan, the init/systemd process (with PID 1) or a similar system process takes over the orphaned process. This ensures that the operating system can properly manage the orphaned process, including reaping its exit status when it terminates, thus preventing it from becoming a zombie process. The adoption by the init process maintains the integrity of the process hierarchy within the operating system.

Now let's trace the process at the process tree:

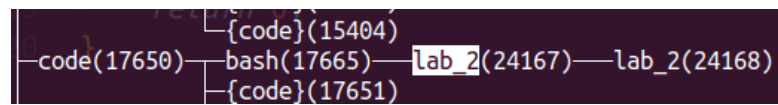


Figure 3: Initial Parent and child

Here, I used a sleep method after creating the process to prevent the parent process from exiting immediately. This allowed me to verify whether both the parent and child processes were created successfully and to keep track of the child process which is shown in the figure 3. Later, I commented out the sleep function. As a result, the exit function caused the parent process to terminate immediately, which made the child process an orphan process shown in the figure 4.

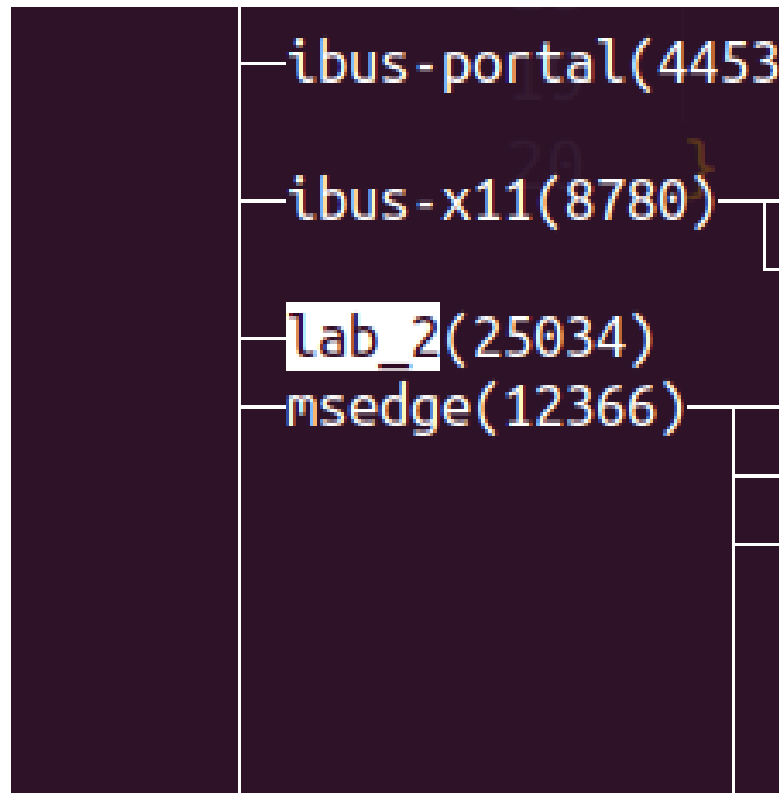


Figure 4: child became an orphan here

When the child process become and orphan it is taken over by systemd which is carried by the init process whose pid is 1. This is shown in the figure [5](#)

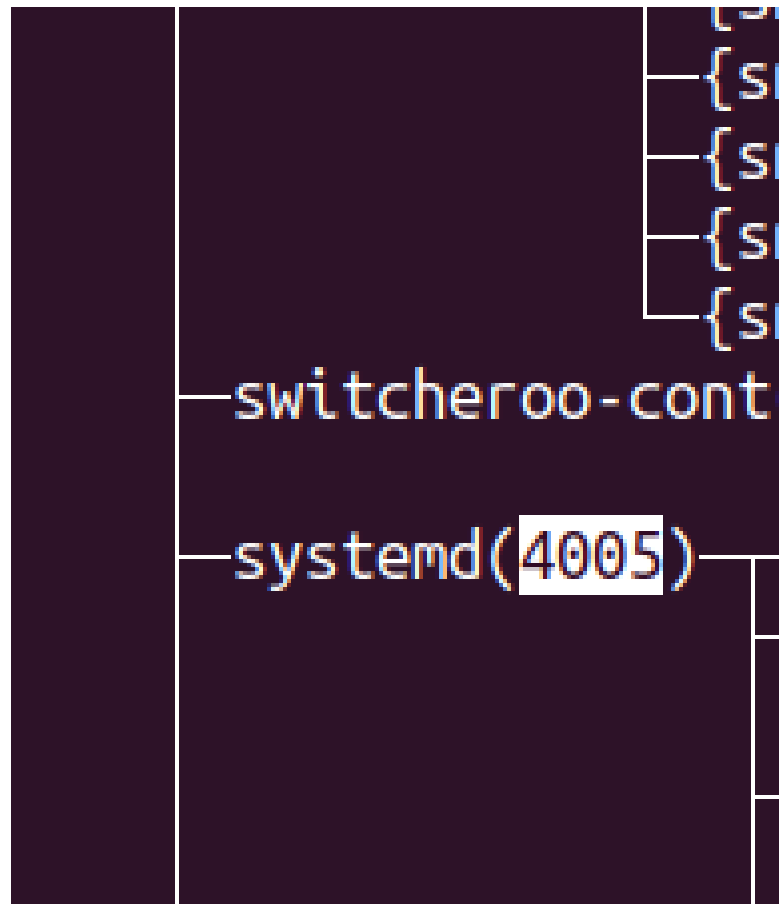


Figure 5: Taken over by init process systemd

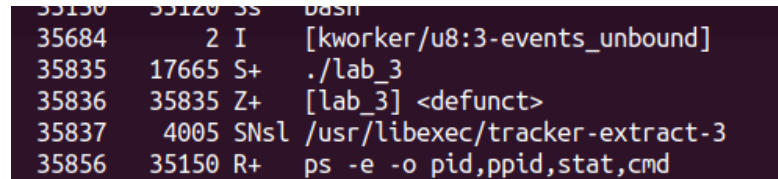
Question 3

Code to create a zombie process:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

int main() {
    pid_t pid = fork();
    if (pid > 0) {
        sleep(10);
    } else if (pid == 0) {
        printf("%d\n", getpid());
        printf("Child process is terminating.\n");
        exit(0);
    }
    return 0;
}
```

Zombie process: A zombie process is a child process that has finished execution but still has an entry in the process table. This occurs because the parent process hasn't yet called `wait()` or `waitpid()` to clean up the child's resources. The process entry exists to hold information about the process's termination status, which may be needed by its parent process for proper cleanup. However, the zombie process no longer performs any activity and consumes no system resources other than the process table entry.



35130	35120	SS	bash
35684	2	I	[kworker/u8:3-events_unbound]
35835	17665	S+	./lab_3
35836	35835	Z+	[lab_3] <defunct>
35837	4005	SNsl	/usr/libexec/tracker-extract-3
35856	35150	R+	ps -e -o pid,ppid,stat,cmd

Figure 6: Zombie Process

As we see in the figure [6](#) In a process tree, a zombie process is typically having its parent process as its immediate ancestor. It is appearing as a child process of its parent, just like any other process. However, its status is indicated as "Z" or "defunct" to signify that it is a zombie process. command to see the status:

```
ps -e -o pid,ps aux | grep 'Z'
```

Question 4

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

int main() {
    pid_t pid1, pid2, pid3;

    pid1 = fork();
    if (pid1 == 0) {
        printf("child_1-(PID: %d)\n", getpid());
        for (size_t i = 0; i < 100000000000; i++){
        }

        exit(0);
    } else {
        pid2 = fork();
        if (pid2 == 0) {
            printf("child_2-(PID: %d)\n", getpid());
            for (size_t i = 0; i < 100000000000; i++){
            }

            exit(0);
        } else {
            pid3 = fork();
            if (pid3 == 0) {
                printf("child_3-(PID: %d)\n", getpid());
                for (size_t i = 0; i < 100000000000; i++){
                }
            }
        }
    }
}
```

```

    }
    exit(0);
} else {
    printf("parent_process-(PID: %d)\n", getpid());
    for (size_t i = 0; i < 1000000000000; i++){
    }
    waitpid(pid1, NULL, 0);
    waitpid(pid2, NULL, 0);
    waitpid(pid3, NULL, 0);
}
}
}
return 0;
}

```

The trace of the processes is shown in the figure below:

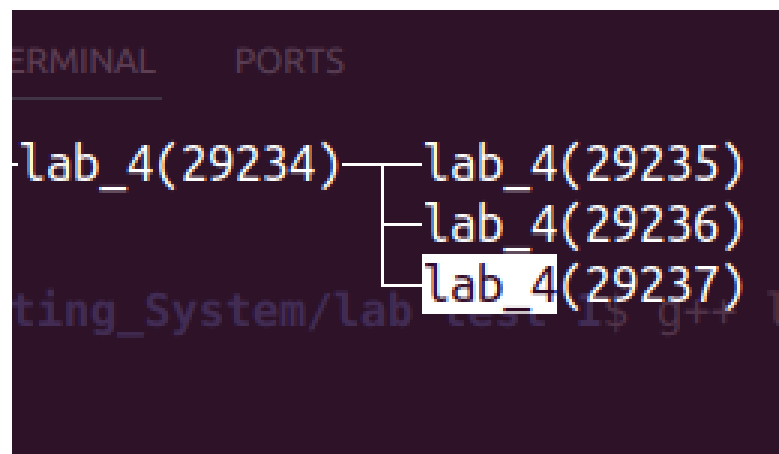


Figure 7: Parent having Three child

Output:

```

^C
sajjad@sajjad:~/Desktop/32/0
parent_process (PID: 29234)
child_2 (PID: 29236)
child_1 (PID: 29235)
child_3 (PID: 29237)

```

Figure 8: Output shows the pid of parent and 3 children

The program creates a parent process that forks three child processes sequentially. Each child process prints its PID and then enters a loop, simulating some computational work. The parent process waits for each child to terminate using `waitpid()`. However, the loop in the parent process may cause it to wait longer than necessary. Additionally, the loop counters in the child processes are set to very large values, potentially leading to long execution times. This is basically to trace the process tree; otherwise, the loop has no work here.

Question 5

Solution:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/prctl.h>

int main(int argc, char *argv[]) {
    if (argc < 3) {
        fprintf(stderr, "Usage: %s -n child_1 -child_2 - ... -child_n\n", argv
            [0]);
        return 1;
    }

    int n = atoi(argv[1]);
    pid_t pids[n];

    for (int i = 0; i < n; i++) {
        pids[i] = fork();
        if (pids[i] == 0) {
            prctl(PR_SET_NAME, argv[i + 2], NULL, NULL, NULL);
            sleep(10);
            printf("%s - (PID: %d)\n", argv[i + 2], getpid());
            exit(0);
        }
    }

    printf("parent process - (PID: %d)\n", getpid());

    for (int i = 0; i < n; i++) {
        wait(NULL);
    }

    return 0;
}
```

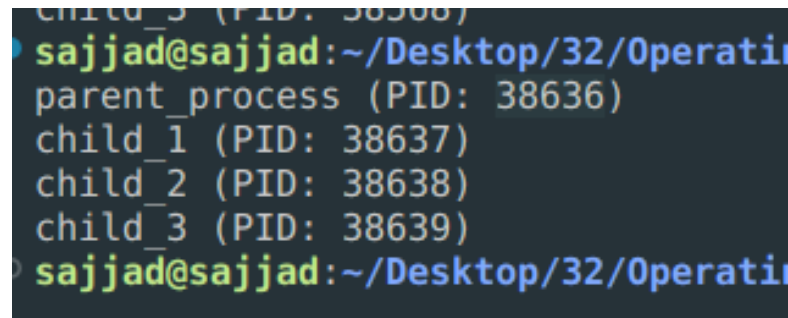
This program creates a specified number of child processes based on the command-line arguments. Each child process sleeps for 10 seconds and then prints its name (provided as a command-line argument) along with its PID. The parent process waits for all child processes to terminate before exiting. The number of child processes and their names are provided as command-line arguments. If fewer than three arguments are provided, the program prints a usage message and exits with an error code. To run the code first we need to compile the code using command in the terminal. Example:

```
g++ lab_5.c -o lab_5
```


This will compile the program. after that we will have to run the code and pass the argument in the commandline. If we want 3 child process having name child_1, child_2, child_3 then the typical command will be:

```
./lab_5 3 child_1 child_2 child_3
```

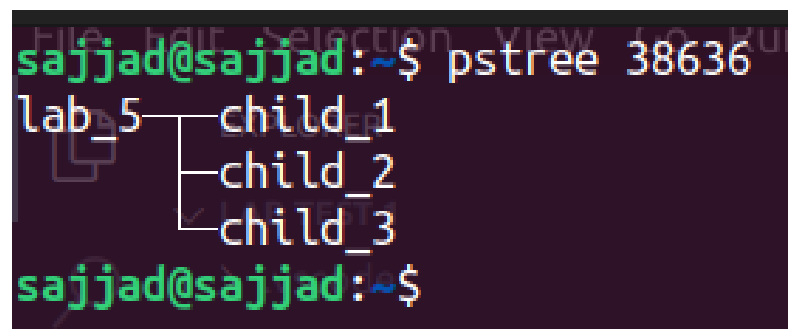
The program output is shown in the figure [9](#)



```
child_3 (PID: 38639)
sajjad@sajjad:~/Desktop/32/Operati
parent_process (PID: 38636)
child_1 (PID: 38637)
child_2 (PID: 38638)
child_3 (PID: 38639)
sajjad@sajjad:~/Desktop/32/Operati
```

Figure 9: Output of creating child giving argument in the commandline

Trace of the process created are given in the next figure [10](#)



```
File Edit Selection View Go Run
sajjad@sajjad:~$ pstree 38636
lab_5--child_1
      |--child_2
      |--child_3
sajjad@sajjad:~$
```

Figure 10: process tree when n=3

```
sajjad@sajjad:~$ pstree 38899
lab_5--child_1
      --child_2
      --child_3
      --child_4
      --child_5
      --child_6
sajjad@sajjad:~$
```

Figure 11: process tree when $n=6$

CSE3241: Operating Systems

Lab Test

Name: Md Sajjad Hossain

ID: 2011176125

Session: 2019-20

May 26, 2024

Question 1:

code:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <pthread.h>

void * forkthread(void *arg){
    printf("Thread-%d-started\n", (long)arg);
    pid_t thread1 = fork();
    if(thread1==0){
        printf("thread-1-created-a-child-process-of-pid-%d\n", getpid());
        exit(0);
    }else{
        printf("Parent-waiting-for-child-to-finish-its-work\n");
        wait(NULL);
    }
    return NULL;
}

void * execfunc(void *arg){
    printf("Thread-%d-started\n", (long)arg);
    pid_t thread2 = fork();
    if(thread2==0){
        char *args[] = {"./test", NULL};
        execv(args[0], args);
        exit(0);
    }else{
        wait(NULL);
        printf("Parent-got-control-back-from-exec-child-call\n");
    }
}

void * exec_new(void *arg){
    printf("Thread-%d-started\n", (long)arg);
    fork();
    char *args[] = {"./test2", NULL};
    execv(args[0], args);
}
```

```

        printf("Parent2-got-control-back\n");
    }

    int main() {
        pthread_t tid1, tid2, tid3;
        pthread_create(&tid1, NULL, forkthread, (void *)1);
        pthread_join(tid1, NULL);
        pthread_create(&tid2, NULL, execfunc, (void *)2);
        pthread_join(tid2, NULL);
        pthread_create(&tid3, NULL, exec_new, (void *)3);
        pthread_join(tid3, NULL);
    }

```

Another C program of process that will be called by `execv`:

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <pthread.h>

int main() {
    printf("test-process\n");
    exit(0);
}

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <pthread.h>

int main() {
    printf("test2-process\n");
    exit(0);
}

```

- A.** If one thread in the process calls `fork()`, does the new process duplicate all threads, or is the new process single-threaded?
- Some UNIX systems have chosen to have two versions of `fork()`, one that duplicates all threads and another that duplicates only the thread that invoked the `fork()` system call.
- B.** If a thread invokes the `exec()` system call, does it replace the entire code of the process?
- Yes, when a thread invokes `exec()`, it replaces the entire code of the process. The new program loaded by `exec()` runs as a single-threaded process.
- C.** If `exec()` is called immediately after forking, will all threads be duplicated?
- No, if `exec()` is called immediately after `fork()`, only the single thread that exists in the new process created by `fork()` will be replaced by the new program. Other threads from the original process are not duplicated.

Output:

```

Thread 1 started
Parent waiting for child to finish its work
thread 1 craeted a child process of pid 211051
Thread 2 started
test process
Parent got control back from exec child call
Thread 3 started
test2 process
test2 process
sajjad@sajjad:~/Desktop/32/Operating_System/practice_lab$

```

Figure 1: output-1

Question 2:

code:

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <pthread.h>
#include <signal.h>

int signal_received = 0;
void signal_handler(int sig){
    signal_received = sig;
}

void * func_1(void *arg){
    pthread_t thread2 = *(pthread_t *)arg;
    sleep(1);
    printf("Thread-1:-sending-signal-to-thread-2\n");
    pthread_kill(thread2, SIGUSR1);

    return NULL;
}

void * func_2(void *arg){
    printf("Thread-2:-waiting-for-Signal\n");
    signal(SIGUSR1, signal_handler);
    pause();
    printf("Thread-2:-Signal-received:%d\n", signal_received);
    printf("Thread-2:-Signal-Handling-complete\n");

    return NULL;
}

int main(){
    pthread_t thread1, thread2;
    pthread_create(&thread2, NULL, func_2, NULL);
    pthread_create(&thread1, NULL, func_1, &thread2);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
}

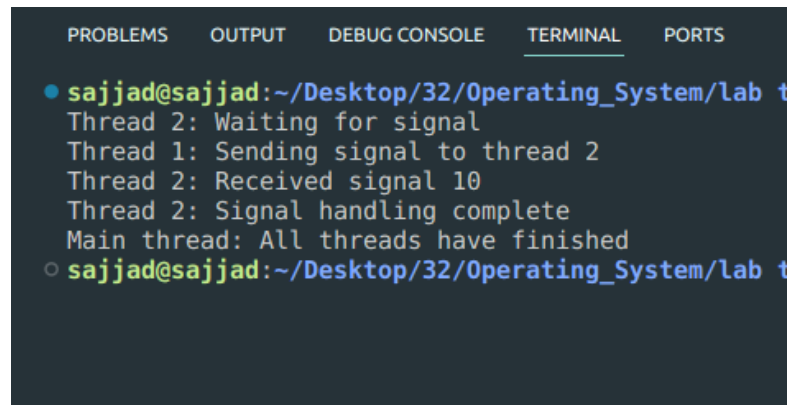
```

Explanation: Communication Between Threads Using Signals

This C program demonstrates how two threads can communicate with each other using signals.

1. **Thread 1 (thread_func1):**
 - Sends a SIGUSR1 signal to Thread 2 (thread_func2) using `pthread_kill`.
2. **Thread 2 (thread_func2):**
 - Sets up a signal handler using the `signal` function to catch the SIGUSR1 signal.
 - Waits until it receives the SIGUSR1 signal.
 - Upon receiving the signal, it handles the signal by setting a flag (`signal_received`) to indicate that the signal has been received.
3. **Signal Handler (signal_handler):**
 - Defined to handle the SIGUSR1 signal.
 - Updates the `signal_received` flag to indicate that the signal has been received.
4. **Main Function:**
 - Creates both threads: `thread1` and `thread2`.
 - Waits for both threads to finish execution using `pthread_join`.
 - Prints a message indicating that all threads have finished.

Output:



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
• sajjad@sajjad:~/Desktop/32/Operating_System/lab t
  Thread 2: Waiting for signal
  Thread 1: Sending signal to thread 2
  Thread 2: Received signal 10
  Thread 2: Signal handling complete
  Main thread: All threads have finished
○ sajjad@sajjad:~/Desktop/32/Operating_System/lab t
```

Figure 2: output-2

In summary, Thread 1 sends a signal to Thread 2, and Thread 2 responds to the signal by executing the signal handler function. This allows communication between the two threads using signals.

Question 3:

code:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int n = 0;

void *increment(void *arg) {
    for (int i = 0; i < 100000; i++) {
        n++;
        printf("Incrementing -n: %d\n", n);
    }
    return NULL;
}

void *decrement(void *arg) {
    for (int i = 0; i < 100000; i++) {
        n--;
        printf("Decrementing -n: %d\n", n);
    }
    return NULL;
}

int main() {
    pthread_t tid_inc, tid_dec;

    pthread_create(&tid_inc, NULL, increment, NULL);
    pthread_create(&tid_dec, NULL, decrement, NULL);

    pthread_join(tid_inc, NULL);
    pthread_join(tid_dec, NULL);

    printf("Final - value - of - shared - variable: %d\n", n);

    return 0;
}
```

Output:

```
Decrementing n: -68
Decrementing n: -69
Decrementing n: -70
Decrementing n: -71
Decrementing n: -72
Decrementing n: -73
Decrementing n: -74
Decrementing n: -75
Decrementing n: -76
Decrementing n: -77
Decrementing n: -78
Decrementing n: -79
Decrementing n: -80
Decrementing n: -81
Decrementing n: -82
Decrementing n: -83
Decrementing n: -84
Decrementing n: -85
Decrementing n: -86
Decrementing n: -87
Decrementing n: -88
Final value of shared variable: -88
```

Figure 3: output-3

Explanation: Data Inconsistency in Multi-threaded Process

This C program demonstrates how data inconsistency can arise in a multi-threaded process due to the lack of proper synchronization.

1. Thread Creation:

Two threads are created: one for incrementing 'n' and one for decrementing 'n'.

2. Thread Operations:

- Each thread performs its operation (increment or decrement) 100,000 times.
- Since both threads access 'n' without proper synchronization, data inconsistency can occur.
- The final value of 'n' depends on the interleaving of instructions executed by the threads, and it may not be the expected result of 0.

3. Final Value:

- After both threads finish execution, the final value of 'n' is printed.

- After running this program, we observed that the final value of 'n' is not always 0, indicating data inconsistency.
- This inconsistency arises due to the lack of synchronization between the threads accessing the shared data. which can be handled by mutex lock.

CSE3241: Operating Systems

Lab Test

Name: Md Sajjad Hossain

ID: 2011176125

Session: 2019-20

June 29, 2024

Question 1:

Solution code:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

double average = 0.0;
int maximum = 0;
int minimum = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void* calculate_average(void* arg) {
    int* numbers = (int*)arg;
    int count = numbers[0];
    int total = 0;

    for (int i = 1; i <= count; i++) {
        total += numbers[i];
    }

    pthread_mutex_lock(&mutex);
    average = (double)total / count;
    pthread_mutex_unlock(&mutex);

    return NULL;
}

void* find_maximum(void* arg) {
    int* numbers = (int*)arg;
    int count = numbers[0];
    int max_val = numbers[1];

    for (int i = 2; i <= count; i++) {
        if (numbers[i] > max_val) {
            max_val = numbers[i];
        }
    }
}
```

```

    }

    pthread_mutex_lock(&mutex);
    maximum = max_val;
    pthread_mutex_unlock(&mutex);

    return NULL;
}

void* find_minimum(void* arg) {
    int* numbers = (int*)arg;
    int count = numbers[0];
    int min_val = numbers[1];

    for (int i = 2; i <= count; i++) {
        if (numbers[i] < min_val) {
            min_val = numbers[i];
        }
    }

    pthread_mutex_lock(&mutex);
    minimum = min_val;
    pthread_mutex_unlock(&mutex);

    return NULL;
}

int main(int argc, char* argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s num1 num2 num3 ... \n", argv[0]);
        return 1;
    }

    int count = argc - 1;
    int* numbers = (int*)malloc((count + 1) * sizeof(int));

    numbers[0] = count;

    for (int i = 1; i <= count; i++) {
        numbers[i] = atoi(argv[i]);
    }

    pthread_t thread1, thread2, thread3;

    pthread_create(&thread1, NULL, calculate_average, (void*)numbers);
    pthread_create(&thread2, NULL, find_maximum, (void*)numbers);
    pthread_create(&thread3, NULL, find_minimum, (void*)numbers);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_join(thread3, NULL);

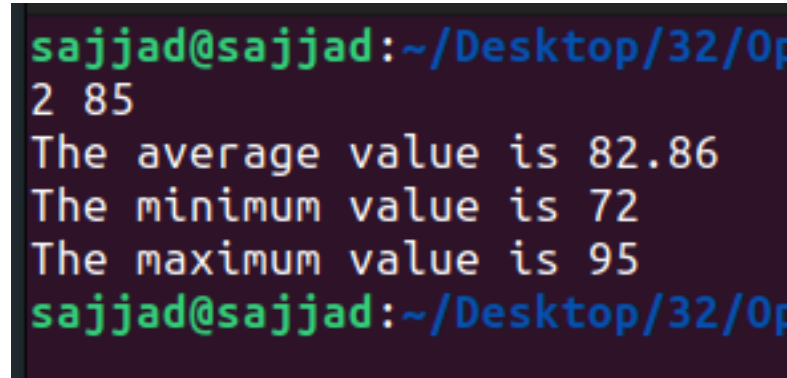
    printf("The average value is %.2f\n", average);
    printf("The minimum value is %d\n", minimum);
    printf("The maximum value is %d\n", maximum);

    free(numbers);
    pthread_mutex_destroy(&mutex);
}

```

```
        return 0;
    }
```

Output:

A terminal window with a dark background and light-colored text. The prompt is 'sajjad@sajjad:~/Desktop/32/Op'. The input is '2 85'. The output consists of three lines: 'The average value is 82.86', 'The minimum value is 72', and 'The maximum value is 95'. The prompt is repeated at the bottom.

```
sajjad@sajjad:~/Desktop/32/Op
2 85
The average value is 82.86
The minimum value is 72
The maximum value is 95
sajjad@sajjad:~/Desktop/32/Op
```

Figure 1: output-1

Explanation:

In this program, the `calculate_average()`, `find_maximum()`, and `find_minimum()` functions are defined to calculate the average, maximum, and minimum values, respectively. These functions store the results in global variables.

The `main()` function creates three worker threads using the threading module, each running one of the above functions. The main thread waits for the worker threads to finish using the `join()` method and finally outputs the results.

We can pass the list of numbers to the program as command-line arguments, and the program will output the average, minimum, and maximum values for those numbers.

Question 2:

Solution Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <pthread.h>

int * fib;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void * gen_fibo(void *arg){
    int cnt = *(int*)arg;
    pthread_mutex_lock(&mutex);
    fib[0] = 0;
```

```

        if(cnt>1){
            fib[1] = 1;
            for(int i=2; i<cnt; i++){
                fib[i] = fib[i-1] + fib[i-2];
            }
        }
        pthread_mutex_unlock(&mutex);
        return NULL;
    }

    int main(int argc, char *argv[]) {
        int cnt = atoi(argv[1]);
        fib = (int*) malloc(cnt*sizeof(int));

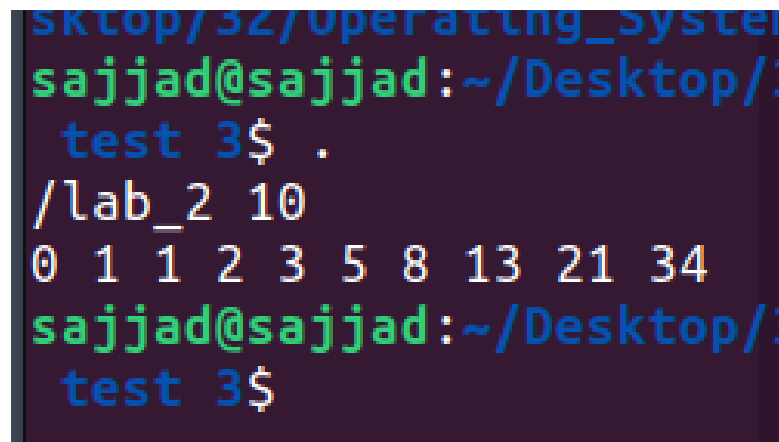
        pthread_t thread;
        pthread_create(&thread, NULL, gen_fibo, &cnt);

        pthread_join(thread, NULL);

        for(int i=0; i<cnt; i++){
            printf("%d ", fib[i]);
        }
        printf("\n");
        free(fib);
        pthread_mutex_destroy(&mutex);
    }

```

Output:



```

sajjad@sajjad:~/Desktop/3
test 3$ .
/lab_2 10
0 1 1 2 3 5 8 13 21 34
sajjad@sajjad:~/Desktop/3
test 3$

```

Figure 2: output-2

Explanation:

Main Function:

- The main function initializes the necessary variables and attributes.
- It creates a thread to generate the Fibonacci sequence.
- It waits (`pthread_join`) for the child thread to complete.

- After the child thread completes, the main thread prints the Fibonacci sequence.

Thread Function:

- The thread function generates the Fibonacci sequence and stores it in a shared array.

Question 3:

Solution:

Single threaded server

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/wait.h>

#define PORT 8080
#define BUFFER_SIZE 1024

void handle_client(int client_socket) {
    char buffer[BUFFER_SIZE] = {0};

    read(client_socket, buffer, BUFFER_SIZE);
    printf("Received: %s\n", buffer);

    send(client_socket, buffer, strlen(buffer), 0);
    printf("Echoed: %s\n", buffer);

    close(client_socket);
}

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);

    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
        perror("bind failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    if (listen(server_fd, 3) < 0) {
```

```

        perror("listen failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    while (1) {
        printf("Waiting for a connection...\n");

        if ((new_socket = accept(server_fd, (struct sockaddr *)&address, (
            socklen_t *)&addrlen)) < 0) {
            perror("accept failed");
            close(server_fd);
            exit(EXIT_FAILURE);
        }

        printf("Connection accepted\n");

        pid_t pid = fork();
        if (pid < 0) {
            perror("fork failed");
            close(new_socket);
        } else if (pid == 0) {
            close(server_fd);
            handle_client(new_socket);
            exit(0);
        } else {
            close(new_socket);
            waitpid(-1, NULL, WNOHANG);
        }
    }

    close(server_fd);
    return 0;
}

```

Multi-Threaded Server:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <arpa/inet.h>

#define PORT 8080
#define BUFFER_SIZE 1024

void *handle_client(void *client_socket) {
    int socket = *(int *)client_socket;
    char buffer[BUFFER_SIZE] = {0};

    read(socket, buffer, BUFFER_SIZE);
    printf("Received: %s\n", buffer);

    send(socket, buffer, strlen(buffer), 0);
    printf("Echoed: %s\n", buffer);

    close(socket);
    free(client_socket);
    pthread_exit(NULL);
}

int main() {

```

```

int server_fd , new_socket;
struct sockaddr_in address;
int addrlen = sizeof(address);

if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
    perror("socket - failed");
    exit(EXIT_FAILURE);
}

address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;
address.sin_port = htons(PORT);

if (bind(server_fd , (struct sockaddr *)&address , sizeof(address)) < 0) {
    perror("bind - failed");
    close(server_fd);
    exit(EXIT_FAILURE);
}

if (listen(server_fd , 3) < 0) {
    perror("listen - failed");
    close(server_fd);
    exit(EXIT_FAILURE);
}

while (1) {
    printf("Waiting for a connection...\n");
    if ((new_socket = accept(server_fd , (struct sockaddr *)&address , (
        socklen_t *)&addrlen)) < 0) {
        perror("accept - failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    printf("Connection - accepted\n");

    pthread_t thread_id;
    int *client_socket = malloc(sizeof(int));
    *client_socket = new_socket;
    if (pthread_create(&thread_id , NULL, handle_client , (void *)
        client_socket) != 0) {
        perror("pthread_create - failed");
        close(new_socket);
        free(client_socket);
    }
}

close(server_fd);
return 0;
}

```

Single-Threaded Server with Multiple Child Processes:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/wait.h>

```



```

#define PORT 8080
#define BUFFER_SIZE 1024

void handle_client(int client_socket) {
    char buffer[BUFFER_SIZE] = {0};

    read(client_socket, buffer, BUFFER_SIZE);
    printf("Received: %s\n", buffer);

    send(client_socket, buffer, strlen(buffer), 0);
    printf("Echoed: %s\n", buffer);

    close(client_socket);
}

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);

    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket - failed");
        exit(EXIT_FAILURE);
    }

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
        perror("bind - failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    if (listen(server_fd, 3) < 0) {
        perror("listen - failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    while (1) {
        printf("Waiting for a connection...\n");

        if ((new_socket = accept(server_fd, (struct sockaddr *)&address, (
            socklen_t *)&addrlen)) < 0) {
            perror("accept - failed");
            close(server_fd);
            exit(EXIT_FAILURE);
        }

        printf("Connection accepted\n");

        pid_t pid = fork();
        if (pid < 0) {
            perror("fork - failed");
            close(new_socket);
        } else if (pid == 0) {
            close(server_fd);
            handle_client(new_socket);
            exit(0);
        } else {

```

```

        close(new_socket);
        waitpid(-1, NULL, WNOHANG);
    }
}

close(server_fd);
return 0;
}

```

client code:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 8080
#define BUFFER_SIZE 1024

int main() {
    int sock = 0;
    struct sockaddr_in serv_addr;
    char buffer[BUFFER_SIZE] = {0};
    char *message = "Hello~from~client";
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Socket~creation~error");
        return -1;
    }

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);
    if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
        perror("Invalid~address/-Address~not~supported");
        return -1;
    }

    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
    {
        perror("Connection~failed");
        return -1;
    }

    send(sock, message, strlen(message), 0);
    printf("Message~sent:~%s\n", message);

    int valread = read(sock, buffer, BUFFER_SIZE);
    printf("Server~response:~%s\n", buffer);

    close(sock);

    return 0;
}

```

Output: server:

```
sajjad@sajjad:~/Desktop/32/Operating_System
Waiting for a connection...
Connection accepted
Waiting for a connection...
Received: Hello from client
Echoed: Hello from client
```

Figure 3: output-3.1

Client:

```
Server response: Hello from client
sajjad@sajjad:~/Desktop/32/Operating_Sy
Message sent: Hello from client
Server response: Hello from client
sajjad@sajjad:~/Desktop/32/Operating_Sy
```

Figure 4: output-3.2

Single-Threaded Server

When we run the single-threaded server code, it will:

- **Start Listening:** The server will start and listen for incoming client connections on port 8080.
- **Accept One Connection at a Time:** When a client connects, the server will accept the connection.
- **Process Request:** The server will read the data sent by the client, echo it back to the client, and then close the connection.
- **Wait for Next Connection:** The server will then wait for the next client to connect and repeat the process.

Client Experience: Only one client can be served at a time. If multiple clients try to connect simultaneously, subsequent clients will have to wait until the current client is served and disconnected.

Multi-Threaded Server

When we run the multi-threaded server code, it will:

- **Start Listening:** The server will start and listen for incoming client connections on port 8080.
- **Accept Connections Concurrently:** When clients connect, the server will accept each connection and create a new thread to handle the client's request.

- **Process Requests Concurrently:** Each client request will be handled in its own thread. The server can serve multiple clients simultaneously.
- **Threads Terminate After Serving:** After handling a client's request, the corresponding thread will terminate.

Client Experience: Multiple clients can be served concurrently. Each client gets a dedicated thread, resulting in faster response times and improved concurrency.

Single-Threaded Server with Multiple Child Processes

When we run the multi-process server code, it will:

- **Start Listening:** The server will start and listen for incoming client connections on port 8080.
- **Accept Connections and Fork Processes:** When clients connect, the server will accept each connection and fork a new child process to handle the client's request.
- **Process Requests Concurrently:** Each client request will be handled in a separate child process. The server can serve multiple clients simultaneously through separate processes.
- **Processes Terminate After Serving:** After handling a client's request, the corresponding child process will terminate.

Client Experience: Multiple clients can be served concurrently. Each client gets a dedicated process, providing better isolation and stability but potentially higher overhead due to process creation. [\[1\]](#)

Question 4:

Write a multithreaded C program to perform some multithreaded tasks. The program should do the following:

One thread gets numbers from the user. Then, a second thread orders those numbers using a sorting algorithm. And lastly, a third thread should print the sorted numbers.

In order to get a full grade, you should explain your code using comments.

Solution

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int *numbers;
int size;

pthread_mutex_t lock;

void *get_numbers(void *param) {
    pthread_mutex_lock(&lock);

    printf("Enter the number of elements: ");
```

```

scanf("%d", &size);

numbers = (int *)malloc(size * sizeof(int));
if (numbers == NULL) {
    fprintf(stderr, "Memory allocation failed\n");
    exit(EXIT_FAILURE);
}

printf("Enter %d numbers: ", size);
for (int i = 0; i < size; i++) {
    scanf("%d", &numbers[i]);
}

pthread_mutex_unlock(&lock);
pthread_exit(0);
}

void *sort_numbers(void *param) {
    pthread_mutex_lock(&lock); // Lock the mutex

    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (numbers[j] > numbers[j + 1]) {
                int temp = numbers[j];
                numbers[j] = numbers[j + 1];
                numbers[j + 1] = temp;
            }
        }
    }

    pthread_mutex_unlock(&lock); // Unlock the mutex
    pthread_exit(0);
}

void *print_numbers(void *param) {
    pthread_mutex_lock(&lock);

    printf("Sorted numbers: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", numbers[i]);
    }
    printf("\n");

    pthread_mutex_unlock(&lock);
    pthread_exit(0);
}

int main() {
    pthread_t tid1, tid2, tid3;

    pthread_mutex_init(&lock, NULL);

    pthread_create(&tid1, NULL, get_numbers, NULL);
    pthread_join(tid1, NULL);

    pthread_create(&tid2, NULL, sort_numbers, NULL);
    pthread_join(tid2, NULL);
    pthread_create(&tid3, NULL, print_numbers, NULL);
    pthread_join(tid3, NULL);

    pthread_mutex_destroy(&lock);

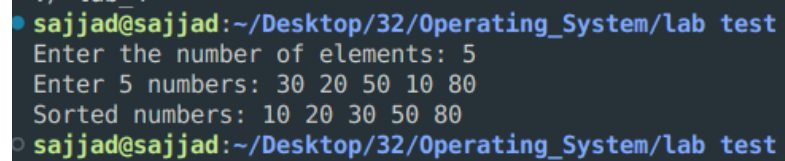
```

```

    free(numbers);
    return 0;
}

```

Output:



```

sajjad@sajjad:~/Desktop/32/Operating_System/lab test
Enter the number of elements: 5
Enter 5 numbers: 30 20 50 10 80
Sorted numbers: 10 20 30 50 80
sajjad@sajjad:~/Desktop/32/Operating_System/lab test

```

Figure 5: output-4

Explanation

Main Function

- **Initialize Mutex:** The mutex is initialized to synchronize threads.
- **Create Threads:** Three threads are created:
 - **tid1:** For getting numbers from the user.
 - **tid2:** For sorting the numbers.
 - **tid3:** For printing the sorted numbers.
- **Wait for Threads:** The main function waits for each thread to complete using `pthread_join`.
- **Destroy Mutex:** The mutex is destroyed after all threads are done.
- **Free Memory:** The dynamically allocated memory for the numbers is freed.

Thread Functions

- **get_numbers:** This function prompts the user to input the number of elements and the elements themselves. It stores these numbers in a globally accessible array.
- **sort_numbers:** This function sorts the array using the bubble sort algorithm. It ensures that the sorting operation is synchronized using a mutex.
- **print_numbers:** This function prints the sorted numbers to the console. It also ensures synchronization using a mutex.

References

- [1] OpenAI. *ChatGPT: A Large Language Model*. Accessed: 2024-06-30. 2024. URL: <https://www.openai.com/chatgpt>

CSE3241: Operating Systems

Class Test

Name: Md Sajjad Hossain

ID: 2011176125

Session: 2019-20

May 23, 2024

Question-1:

a. Can you kill processes having PID 0 and PID 1? If yes, write down the steps of killing these processes.

The process with PID 0 is responsible for paging, which is always referred to as the swapper or sched process. This process is a part of the kernel and is not a regular user-mode process. So we cant really access it using `top` or `ps` command. On the other hand process with pid 0 is known as systemd process or init process. On linux based or most of the operating system this process is the root of all user process.

Killing Process with PID 0 (Idle Process)

1. Using the kill command:

```
sudo kill -9 0
```

2. Simply terminate the terminal.

Killing Process with PID 1 (Init Process)

1. Using kill command:

```
kill -9 1
```

2. This command results in operation not permitted. So we cant kill a process with kill command.
3. With sudo permission, we may be able to kill systemd process that will result in system instability or may not work properly.

A. Process Creation

A.

```
pid_t pid;  
pid = fork();  
if (pid == 0) {  
    fork();  
    pthread_create(&tid, NULL, (void*)thread_handler, NULL);  
}  
fork();
```

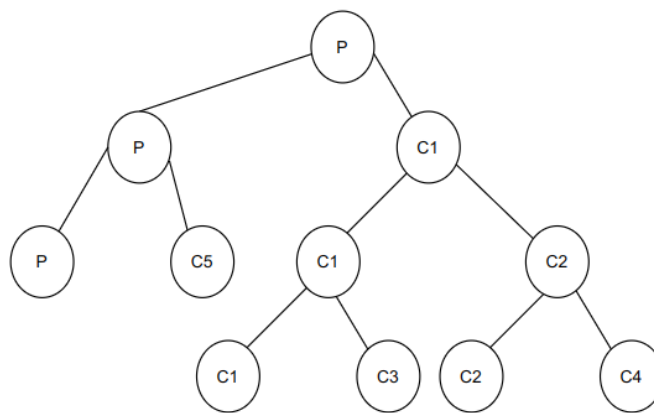


Figure 1: Process Tree

A. Total Process: 6 Unique process will be created

B. Total Thread: 2 Unique Thread will be created

b. What will be the output for the following code segment?

The given code segment invokes the 'fork()' system call three times in sequence.

```
fork();  
printf("Bangladesh");  
fork();  
printf("Bangladesh");  
fork();
```

This code prints Bangladesh 16 times in the terminal. It's not usual behavior. It is supposed to print Bangladesh 6 times. But when I add an escape sequence in the print statement the code gives the intended output.


```

output.in
1 BangladeshBangladeshBangladeshBangladeshB
  angladeshBangladeshBangladeshBangladeshBa
    ngladeshBangladeshBangladeshBangladeshBan
      gladeshBangladeshBangladeshBangladesh

```

Figure 2: Output of the code

c. How many child processes will be created if the following loop is in a program code? Give a proper explanation for your answer.

```

for (i = 0; i < n; i++)
    fork();

```

When $n = 4$, the loop will iterate 4 times. Hence, the total number of processes created will be $2^4 = 16$. And Total number of child process created will be $2^4 - 1 = 15$,

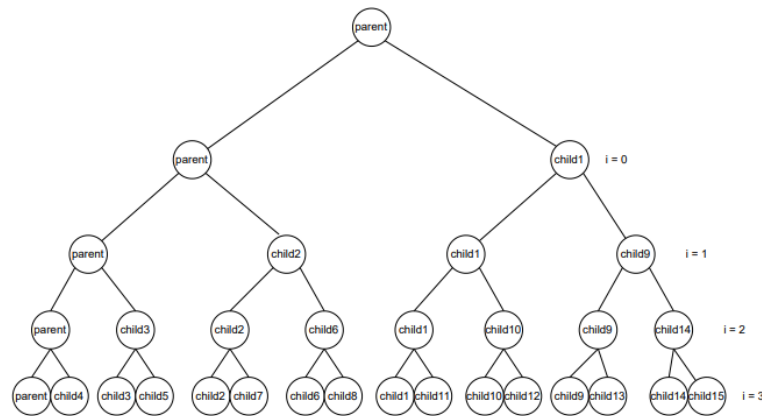


Figure 3: Total Processes

Question-2:

a. Give two reasons why caches are useful

- They increase the speed of data access for the CPU.
- They reduce the average time to access data from the main memory.

b. What problems do they solve?

Cache increases data retrieval performance by reducing the need to access the underlying slower storage layer. Caches are extremely fast storage systems that hold the copy of the information to be accessed temporarily, so that if that information is needed again, it can be fetched from the cache, allowing faster access and shit

c. What problems do they cause?

Some problems caused by the caches is the need to manage it, since they have limited size. They can also cause cache coherency issues in multiprocessor systems.

d. If a cache can be made as large as the device for which it is caching (for instance, a cache as large as a disk), why not make it that large and eliminate the device?

Manufacturing caches is a lot more expensive than manufacturing slow secondary storage devices. Due to cost and diminishing returns on performance beyond a certain size the large cache is not produced.

Question-3: What do you know about:

a. Orphan Process: If a parent process terminates before its child, it will not have a chance to call the `wait()` system call to collect the child's termination status, which would otherwise be stored in the child's process control block (PCB). This results in the child process becoming an orphan. When a parent terminates without invoking `wait()`, the child process becomes an orphan, and it is subsequently adopted by the `systemd` process. After adoption, `systemd` will eventually call `wait()` at a certain point to retrieve the exit information of the orphaned child.

Zombie Process:

A zombie process is a child process that has finished execution but still has an entry in the process table. This occurs because the parent process hasn't yet called `wait()` to clean up the child's resources. The process entry exists to hold information about the process's termination status, which may be needed by its parent process for proper cleanup. However, the zombie process no longer performs any activity and consumes no system resources other than the process table entry.

b. When can you call a process an orphan process or a zombie process? Orphan Process: An orphan process is a process whose parent process has terminated or finished execution before the child process. As a result, the orphan process is adopted by the `init` process (usually with process ID 1), ensuring it continues execution.

Zombie Process: A zombie process is a process that has completed execution but still has an entry in the process table because its parent process has not yet collected its termination status using the `wait()` system call. In this state, the process consumes no system resources other than the process table entry.

c. Can you create them? If yes, describe your steps with code.

1. Creating an Orphan Process: To create an orphan process, we fork a child process and then terminate the parent process before the child process completes its execution. Below is an example of creating an orphan process.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    pid_t child_pid = fork();
```

```

        if (child_pid == 0) {
            sleep(10);
        } else {
            exit(EXIT_SUCCESS);
        }

        return 0;
    }

```

2. Creating a Zombie Process: To create a zombie process, we fork a child process and then allow the parent process to continue execution without invoking `wait()` system call. This causes the child process to become a zombie. Below is an example of creating an orphan process:

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
    pid_t child_pid = fork();
    if (child_pid == 0) {
        exit(EXIT_SUCCESS);
    } else {
        sleep(5);
    }

    wait(NULL);

    return 0;
}

```

d. How can we trace an orphan and zombie process in a Linux based OS?

In a Linux-based operating system, we can trace orphan and zombie processes using various system monitoring and debugging tools. Here are some common methods:

1. **ps Command:** The `ps` command can be used to list running processes along with their status. To find orphan processes, you can look for processes whose parent process ID (PPID) is 1 (init process). To find zombie processes, you can look for processes with a status of "Z" (zombie).

```
ps aux | grep defunct
```

2. **top Command:** The `top` command provides an interactive real-time view of system processes, including their status. You can press "z" to toggle the display of zombie processes.

`top`

3. **htop Command:** Similar to `top`, `htop` is an interactive process viewer but with more features and a better user interface. Zombie processes are shown in a different color.

`htop`



e. Explain how a Linux based OS treats them.

In a Linux-based operating system, orphan and zombie processes are treated differently:

- **Orphan Processes:** When a parent process terminates before its child processes, these child processes become orphaned. In Linux, orphan processes are adopted by the `systemd` process (process ID 1), which acts as the ultimate parent of all processes. The `systemd` process reaps orphaned child processes, preventing them from becoming zombies, and ensures proper cleanup of system resources.
- **Zombie Processes:** Zombie processes are created when a child process terminates, but its parent process has not yet called the `wait()` system call to retrieve its exit status. In Linux, zombie processes consume system resources, such as a process ID and an entry in the process table, but they do not consume any CPU time. The kernel retains zombie processes until the parent process calls `wait()` or `waitpid()` to collect their exit status. Once collected, the kernel removes the zombie process entry from the process table and releases the associated resources.

e. What are the advantages and disadvantages of having orphan and zombie processes?

Advantages:

- **Orphan Processes:**
 - **Resource Cleanup:** Orphan processes ensure that system resources are properly released even if the parent process terminates unexpectedly. By re-parenting orphaned processes to the `init` process, Linux prevents resource leakage and maintains system stability.
- **Zombie Processes:**
 - **Parent Process Notification:** Zombie processes serve as a notification mechanism for parent processes. They indicate that a child process has terminated but its exit status has not yet been collected by the parent. This allows the parent process to take appropriate action, such as releasing additional resources or performing error handling.

Disadvantages:

- **Orphan Processes:**

- **Potential Resource Leakage:** If orphan processes are not re-parented and terminated properly by the init process, they may continue to consume system resources indefinitely. This can lead to resource exhaustion and degradation of system performance.

- **Zombie Processes:**

- **Resource Consumption:** Zombie processes consume system resources, such as process identifiers and entries in the process table, until their exit status is collected by the parent process. If the parent process fails to collect the exit status of zombie processes, it may lead to resource wastage and potential performance issues.

Question 4

Say there is an exe file named **Addition**. What happens when:

- A. A single user does double click on its file icon for a single time?
- B. A single user does double click on its file icon for 'n' times?
- C. Multiple users do separate double clicks on its file icon almost the same time?
- D. A single user presses ENTER after typing `./Addition` in a terminal.

Consider program-vs-process concept, memory layout, role of OS loader while answering these questions.

Answers

- A. When a single user double clicks on the **Addition** file icon for a single time, the operating system's loader will load the executable file into memory and create a new process. The process will have its own memory layout, including text, data, heap, and stack segments. The OS will allocate resources and start executing the process.
- B. When a single user double clicks on the **Addition** file icon for 'n' times, the OS will create 'n' separate processes, each with its own memory layout and resources. Each instance will be independently loaded into memory and executed concurrently, as separate processes.
- C. When multiple users do separate double clicks on the **Addition** file icon almost at the same time, the OS will create a separate process for each double click event from each user. Each process will have its own isolated memory space and resources. The OS scheduler will manage these processes, providing the illusion of concurrent execution.
- D. When a single user presses ENTER after typing `./Addition` in a terminal, the terminal sends a command to the shell to execute the **Addition** executable. The OS loader will load the executable into memory, create a new process with its own memory layout, and start execution. This is similar to double-clicking the file icon but initiated via a command-line interface.

Question 5: Do we need an operating system for all the cases? Justify your answer.

- A. **A single user wants to do a single task using a computer system having a finite set of computer hardware.**

An operating system may not be strictly necessary if the task is simple and can be handled by firmware or a single application running directly on the hardware.

- B. **A single user wants to do multiple tasks simultaneously using a computer system having a finite set of computer hardware.**

An operating system is necessary to manage multitasking, ensuring that the tasks are scheduled properly and the hardware resources are allocated efficiently. The OS will handle context switching and resource management.

- C. **A single user wants to do multiple tasks consecutively using a computer system having a finite set of computer hardware.**

An operating system is beneficial but not strictly necessary. If the tasks are performed one after another without requiring multitasking, the user might be able to run each task sequentially without an OS.

- D. **A single user wants to do a single task using a single hardware.**

An operating system is not necessary if the task is very simple and specific, as it can be programmed directly onto the hardware. Examples include simple embedded systems or dedicated devices.

- E. **Multiple users want to do multiple tasks simultaneously using a computer system having a finite set of computer hardware.**

An operating system is essential in this case to manage multiple users and tasks. It handles user authentication, resource allocation, process scheduling, and ensures that each user's tasks do not interfere with others.



Question 7

- a. **How many processes can be in a running state, waiting state, ready state at a time?**

- **Running state:** Only one process can be in the running state at a time per CPU core.
- **Waiting state:** Multiple processes can be in the waiting state simultaneously.
- **Ready state:** Multiple processes can be in the ready state simultaneously.

- b. **Who can have access to the Process Control Block (PCB) of a process? Justify your answer.**

The operating system kernel has access to the PCB of a process. This is because the PCB contains critical information about process management, scheduling, and resource allocation, which must be managed securely by the kernel.

- c. **Where are PCBs stored? Justify your answer.**

PCBs are stored in the operating system's memory space, typically within the kernel space. This ensures that they are protected from user-level access and can be managed efficiently by the OS.

- d. **Without PCBs, is it possible to handle multiple processes? Justify your answer.**
Without PCBs, it would be very difficult to handle multiple processes. PCBs are essential for storing process state information, scheduling, and resource management, which are necessary for multitasking and process control in modern operating systems.

Question 8:

- a. **Write down the difference between:**

i. **Unnamed pipe and named pipe**

- **Unnamed Pipe:** Used for communication between related processes (parent-child). It exists only during the lifetime of the processes. That means parent-child relationship is required. It is Unidirectional
- **Named Pipe (FIFO):** Used for communication between unrelated processes. It persists beyond the life of the processes and is identified by a name in the file system. No parent-child relation is required here. It is Bidirectional but with half-duplex communication system.

ii. **Named pipe and regular file**

- **Named Pipe:** Facilitates inter-process communication (IPC) and data is read in the order it is written.
- **Regular File:** Stores data permanently and allows random access.

iii. **Program and process**

- **Program:** A static set of instructions stored on disk.
- **Process:** A dynamic instance of a program in execution, including its current state, variables, and resources.

iv. **User mode and kernel mode**

- **User Mode:** Limited access to system resources. User applications run here.
- **Kernel Mode:** Full access to all system resources. Core OS functions run here.

- b. **What happens when a child process updates its local and global variables?**

When a child process updates its local and global variables, these changes do not affect the parent process. The child process has its own copy of the memory space, variables, both local and global, due to the process creation mechanism (e.g., ‘fork’ in Unix/Linux).

- c. **Explain how many ways two processes can work on shared data when two processes:**

i. **Have a child-parent relationship**

- Shared Memory: Use shared memory segments.
- Pipes: Use unnamed pipes for communication.
- Files: Use files to read and write shared data.
- Signals: Use signals for simple notifications.

ii. **Do not have a child-parent relationship**

- Named Pipes (FIFOs): Use named pipes for communication.
- Shared Memory: Use shared memory segments.
- Message Queues: Use message queues for communication.
- Sockets: Use network sockets for communication.
- Files: Use files to read and write shared data.

Question 9:

- a. **What happens when two processes want to do read and write operations through a pipe in opposite order.**

When two processes attempt to read and write through a pipe in opposite order, one process will write to the pipe while the other reads. If the reading process reads before the writing process writes, it will block and wait until data is available. Similarly, if the writing process tries to write to a full pipe, it will block until the pipe has space.

- b. **Explain with a figure what happens if there is an instruction `printf("Operating System");` in a C program.** The `printf` function from the user's code calls the standard C library, which in turn uses the `write()` system call to send data to the output. The `write()` call transitions from user mode to kernel mode, allowing the operating system to handle the actual output operation.

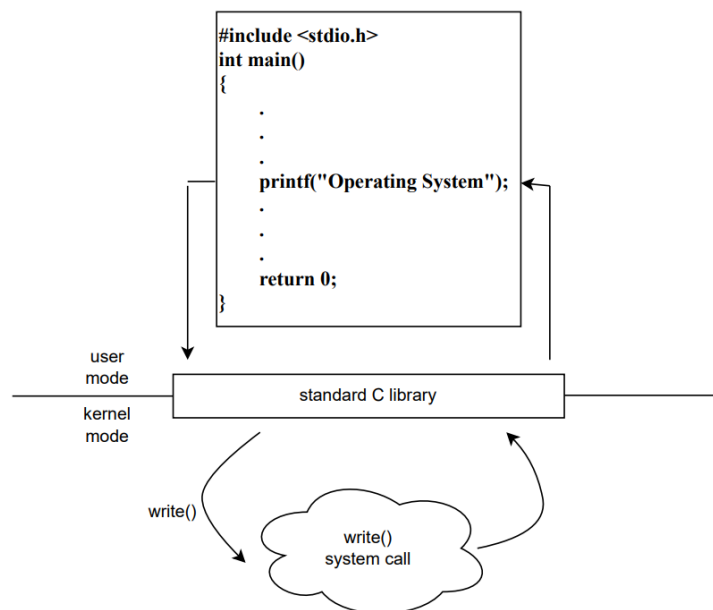


Figure 4: Printf execution

- c. **Does the CPU remain idle when context switching happens? Justify your answer.**

The CPU does not remain entirely idle during context switching. While the actual user process execution is paused, the CPU performs the context switching operations, which involve saving the state of the current process and loading the state of the next process to be executed from the PCB. These operations are managed by the operating system kernel.

- d. **What the operating system can do to keep the CPU busy all the time.**

The operating system can use several strategies to keep the CPU busy:

- **Process Scheduling:** Implement efficient scheduling algorithms (e.g., Round Robin, Priority Scheduling) to ensure there is always a process ready to run.

- **Multithreading:** Allow multiple threads within processes to utilize CPU cores efficiently.
- **Load Balancing:** Distribute tasks evenly across multiple CPU cores.
- **I/O Management:** Overlap I/O operations with CPU computations using asynchronous I/O and DMA (Direct Memory Access).
- **Idle Tasks:** Execute low-priority background tasks or maintenance tasks when no high-priority tasks are ready.

Question 10:

- a. **What is the main difficulty that a programmer must overcome in writing an operating system for a real-time environment?**

I don't have any idea about real-time operating systems. But from the internet I learned this: The main difficulty in writing an operating system for a real-time environment is ensuring a predictable and timely response to external events or stimuli. Real-time systems must meet strict timing constraints, requiring precise control over task scheduling, and resource allocation, and minimizing non-deterministic behavior.

- b. **Some computer systems do not provide a privileged mode of operation in hardware. Is it possible to construct a secure operating system for these computer systems? Give arguments both that it is and that it is not possible.**

Yes, it is possible: Secure operating systems can still be constructed by implementing security mechanisms at the software level, such as encryption, access control, and sandboxing. While lacking hardware-supported privilege modes adds complexity, it does not inherently prevent the implementation of security features.

No, it is not possible: Without hardware-supported privilege modes, it becomes more challenging to enforce memory protection, access control, and isolation between processes, making it easier for malicious code to interfere with system integrity and compromise security.

- c. **What is the purpose of interrupts? How does an interrupt differ from a trap? Can traps be generated intentionally by a user program? If so, for what purpose?**

Interrupts are signals sent to the CPU by external devices or internal conditions to request attention and interrupt the current execution flow. Interrupts allow the CPU to handle asynchronous events promptly, such as I/O operations or hardware errors.

An interrupt is initiated by external hardware or internal conditions, whereas a trap (also known as an exception) is generated by the CPU itself due to a condition detected during the execution of an instruction, such as divide-by-zero or invalid memory access.

Traps can be intentionally generated by a user program for error handling, debugging, or implementing system calls. For example, a program may intentionally divide by zero to trigger a divide-by-zero trap and handle the error gracefully.

References

- [1] OpenAI. *ChatGPT: A Large Language Model*. Accessed: 2024-05-23. 2024. URL: <https://www.openai.com/chatgpt>.

CSE3241: Operating Systems

Class Test

Name: Md Sajjad Hossain

ID: 2011176125

Session: 2019-20

May 26, 2024

Question 1: a.

A. Sending a signal to another process

To send a signal to another process, we use the `kill` system call:

```
#include <signal.h>
int kill(pid_t pid, int sig);
```

where:

- `pid` is the Process ID of the target process.
- `sig` is the signal number to be sent.

For example, to send the `SIGTERM` signal to a process with PID 1234:

```
kill(1234, SIGTERM);
```

B. Sending a signal to a specific thread of the same process

To send a signal to a specific thread within the same process, we use the `pthread_kill` function:

```
#include <pthread.h>
int pthread_kill(pthread_t thread, int sig);
```

where:

- `thread` is the thread ID of the target thread.
- `sig` is the signal number to be sent.

For example, to send the `SIGUSR1` signal to a specific thread identified by `thread_id`:

```
pthread_kill(thread_id, SIGUSR1);
```

C. Sending a signal to a specific thread of a different process

POSIX does not provide a direct way to send a signal to a specific thread of another process. However, on Linux, we can use the `tgkill` system call:

```
#include <sys/syscall.h>
#include <linux/signal.h>
#include <unistd.h>

int tgkill(pid_t tgid, pid_t tid, int sig);
```

where:

- `tgid` is the thread group ID (i.e., the PID of the main thread of the process).
- `tid` is the thread ID of the target thread.
- `sig` is the signal number to be sent.

For example, to send the `SIGUSR1` signal to a thread with TID 5678 in a process with PID 1234:

```
syscall(SYS_tgkill, 1234, 5678, SIGUSR1);
```

This method is specific to Linux.

Summary

- **Another process:** Use `kill(pid, sig)`.
- **Specific thread of the same process:** We use `pthread_kill(thread, sig)`.
- **Specific thread of a different process:** We use the Linux-specific `syscall(SYS_tgkill, tgid, tid, sig)` if available.

b. What happens if a parent process and its child process try to kill each other by sending a `SIGKILL` signal? Consider as many scenarios as possible.

When a parent process and its child process attempt to kill each other by sending a `SIGKILL` signal, the following scenarios can be considered:

Scenario 1: Child sends `SIGKILL` to Parent first

- The child process sends the `SIGKILL` signal to the parent process.
- The operating system immediately terminates the parent process.
- Once the parent process is terminated, it cannot send any signals, including `SIGKILL`, to the child process.
- The child process continues to run until it completes its execution or is terminated by another event.

Scenario 2: Parent sends SIGKILL to Child first

- The parent process sends the **SIGKILL** signal to the child process.
- The operating system immediately terminates the child process.
- Once the child process is terminated, it cannot send any signals, including **SIGKILL**, to the parent process.
- The parent process continues to run until it completes its execution or is terminated by another event.

Scenario 3: Parent and Child send SIGKILL to each other simultaneously

- Both the parent and the child process send the **SIGKILL** signal to each other at the same time.
- The operating system handles these signals based on its scheduling and signal handling mechanism.
- If the operating system processes the child's **SIGKILL** signal to the parent first, the parent process is terminated, and thus, it cannot complete sending the **SIGKILL** signal to the child.
- If the operating system processes the parent's **SIGKILL** signal to the child first, the child process is terminated, and thus, it cannot complete sending the **SIGKILL** signal to the parent.
- There is also a possibility, albeit less common, that both processes are terminated almost simultaneously if the operating system handles the signals in such a manner.

Summary

- If the child sends the **SIGKILL** signal first, the parent is terminated, and the child continues to run.
- If the parent sends the **SIGKILL** signal first, the child is terminated, and the parent continues to run.
- If both send the **SIGKILL** signal simultaneously, the operating system's scheduling and signal handling determine which process is terminated first, with the other process being unable to send the signal post-termination.

c. In Linux, how many signals can be caught and cannot be caught by a user-defined signal handler? Make lists of these two categories signals.

In Linux, signals can be categorized into those that can be caught by a user-defined signal handler and those that cannot be caught.

Signals That Can Be Caught

The following signals can be caught by a user-defined signal handler:

- **SIGHUP** (1) - Hangup detected on controlling terminal or death of controlling process.
- **SIGINT** (2) - Interrupt from keyboard.
- **SIGQUIT** (3) - Quit from keyboard.
- **SIGILL** (4) - Illegal Instruction.
- **SIGTRAP** (5) - Trace/breakpoint trap.
- **SIGABRT** (6) - Abort signal from `abort(3)`.
- **SIGBUS** (7) - Bus error (bad memory access).
- **SIGFPE** (8) - Floating-point exception.
- **SIGUSR1** (10) - User-defined signal 1.
- **SIGSEGV** (11) - Invalid memory reference.
- **SIGUSR2** (12) - User-defined signal 2.
- **SIGPIPE** (13) - Broken pipe: write to pipe with no readers.
- **SIGALRM** (14) - Timer signal from `alarm(2)`.
- **SIGTERM** (15) - Termination signal.
- **SIGSTKFLT** (16) - Stack fault on coprocessor (unused).
- **SIGCHLD** (17) - Child stopped or terminated.
- **SIGCONT** (18) - Continue if stopped.
- **SIGSTOP** (19) - Stop process.
- **SIGTSTP** (20) - Stop typed at terminal.
- **SIGTTIN** (21) - Terminal input for background process.
- **SIGTTOU** (22) - Terminal output for background process.
- **SIGURG** (23) - Urgent condition on socket (4.2BSD).
- **SIGXCPU** (24) - CPU time limit exceeded (4.2BSD).
- **SIGXFSZ** (25) - File size limit exceeded (4.2BSD).
- **SIGVTALRM** (26) - Virtual alarm clock (4.2BSD).
- **SIGPROF** (27) - Profiling timer expired.
- **SIGWINCH** (28) - Window resize signal (4.3BSD, Sun).
- **SIGIO** (29) - I/O now possible (4.2BSD).
- **SIGPWR** (30) - Power failure (System V).
- **SIGSYS** (31) - Bad system call (SVr4).

Signals That Cannot Be Caught

The following signals cannot be caught by a user-defined signal handler:

- **SIGKILL (9)** - Kill signal.
- **SIGSTOP (19)** - Stop process.

The **SIGKILL** and **SIGSTOP** signals are special in that they are used by the system to immediately terminate or stop a process, respectively, and cannot be intercepted, ignored, or handled by the process in any way.

Question 2: a.

Write down the differences between:

A. User Thread vs. Kernel Thread

- **Definition**
 - **User Thread:** Threads that are managed at the user level by a thread library.
 - **Kernel Thread:** Threads that are managed directly by the operating system kernel.
- **Management**
 - **User Thread:** Managed by user-level libraries, without kernel intervention.
 - **Kernel Thread:** Managed by the operating system kernel, requiring system calls for thread management.
- **Performance**
 - **User Thread:** Faster to create and manage as they do not require kernel mode privileges.
 - **Kernel Thread:** Slower to create and manage due to the overhead of kernel mode operations.
- **Scheduling**
 - **User Thread:** Scheduled by the thread library in user space; the kernel is unaware of user-level threads.
 - **Kernel Thread:** Scheduled by the operating system kernel, allowing preemptive multitasking.
- **Blocking Operations**
 - **User Thread:** If a user thread performs a blocking operation, the entire process can be blocked.
 - **Kernel Thread:** If a kernel thread performs a blocking operation, other threads within the same process can continue executing.
- **Portability**
 - **User Thread:** More portable as they are implemented in user space libraries, independent of the kernel.
 - **Kernel Thread:** Less portable as they depend on the kernel's implementation.

Differences between Child Process and Thread

B. Child Process vs. Thread

- **Definition**
 - **Child Process:** A separate instance of a program created by a parent process using the `fork()` system call.
 - **Thread:** A lightweight unit of execution within a process, sharing the same memory space.
- **Memory Space**
 - **Child Process:** Has its own separate memory space, although it inherits the memory space from the parent process.
 - **Thread:** Shares the same memory space with other threads within the same process.
- **Resource Sharing**
 - **Child Process:** Resources such as file descriptors can be shared between parent and child, but memory space is separate.
 - **Thread:** Shares resources including memory, file descriptors, and other process attributes such as stack, register, and program counter with other threads in the same process.
- **Creation Overhead**
 - **Child Process:** Higher overhead due to the need to duplicate the process memory space.
 - **Thread:** Lower overhead as threads are created within an existing process without duplicating the entire memory space.
- **Communication**
 - **Child Process:** Inter-process communication (IPC) mechanisms such as pipes, message queues, or shared memory are required for communication.
 - **Thread:** Communication is easier and faster as threads can directly access shared memory within the same process.
- **Failure Impact**
 - **Child Process:** Failure of a child process does not directly affect the parent process.
 - **Thread:** Failure of a thread (e.g., due to a segmentation fault) can potentially bring down the entire process.



b. Performance Implications of Many-to-Many Threading Model in a Multicore System

Consider a multicore system and a multithreaded program using the many-to-many threading model, where the number of user-level threads is greater than the number of processing cores. The performance implications of different scenarios based on the number of kernel threads allocated to the program are as follows:

A. Number of Kernel Threads Less Than Number of Processing Cores

In the many-to-many threading model, multiple user threads are mapped to a smaller or equal number of kernel threads. If the number of kernel threads assigned to the program is fewer than the available processing cores, only a portion of the cores can be utilized for executing the process. As a result, some cores remain idle, unable to contribute to processing tasks. This limited use of cores leads to suboptimal parallelism and efficiency, as fewer threads are able to run concurrently. Consequently, the process takes longer to complete because the workload is distributed across fewer kernel threads than there are cores available, reducing the potential for speedup and increasing the overall execution time.

B. Number of Kernel Threads Equal to Number of Processing Cores

When the number of kernel threads allocated to a program matches the number of processing cores, all cores can be utilized for executing the process. However, an issue arises if a thread makes a blocking system call, such as waiting for I/O. During this blocking period, the corresponding processing core remains idle for the process, reducing effective system utilization. This can lead to inefficient use of resources, as the idle core cannot contribute to other threads in the program until the blocking operation is resolved.

C. Number of Kernel Threads Greater Than Number of Processing Cores but Less Than Number of User-Level Threads

When the number of kernel threads allocated to a program is more than the number of processing cores but fewer than the number of user-level threads, it helps address the limitations of the previous scenarios. In this setup, all processing cores are fully utilized, allowing the process to run efficiently. If a thread encounters a blocking system call, it can be swapped out, allowing another thread to take its place on the core. This way, there are always threads available for execution, maximizing core usage and improving overall performance, even in the presence of blocking operations.

Question 3:

a. In Linux, if one thread in a program calls `fork()` , does the new process duplicate all threads, or is the new process single-threaded?

Behavior of `fork()` in a Multithreaded Program in Linux

In Linux, when a thread in a multithreaded program calls `fork()`, the behavior with respect to threads in the new process is as follows: In Linux, if a thread in a multithreaded program calls `fork()`, only the calling thread is duplicated in the new process. The newly created process is therefore single-threaded, containing only the forked thread from the original process. Other threads from the parent process are not duplicated, so they do not exist in the child process. This design ensures that the new process starts with a clean state in terms of threads.

b. Is the Linux kernel multithreaded? Justify your answer.

Is the Linux Kernel Multithreaded?

Yes, the Linux kernel is multithreaded. At system startup, multiple kernel threads are spawned to manage various core functions concurrently, improving system efficiency and responsiveness. Each kernel thread is dedicated to a specific task; for example, kswapd handles memory swapping when the system is low on memory, kblockd manages block device requests, and khungtaskd monitors for tasks that are hung or unresponsive. This multithreaded design allows the Linux kernel to handle multiple operations at once, ensuring that tasks like memory management, device handling, and system monitoring are managed simultaneously for smoother overall performance. Also, it supports SMP architecture.

c. Serving 1000 Clients on a Server: Evaluating Approaches

Imagine a server needs to serve 1000 clients. The following approaches can be considered:

- **A. Put service instructions in the main thread of the server process.**
- **B. Create a separate child process to serve each client.**
- **C. Create a separate thread to serve each client.**

A. Putting Service Instructions in the Main Thread

This implementation will lead to serving one client at a time where other client will have to wait for the serving client to finish its task. Which will lead to unexpected user experience that we don't want.

B. Creating a Separate Child Process for Each Client

- **Isolation:** Each client is handled by a separate process, providing strong isolation between clients. If one process crashes, it does not affect others.
- **Resource Usage:** Creating 1000 child processes can be resource-intensive in terms of memory and CPU overhead due to the need to duplicate the process address space.
- **Performance:** Context switching between processes is more costly than between threads, which can degrade performance with a large number of processes.
- **Complexity:** Managing 1000 processes can be complex and may require inter-process communication (IPC) mechanisms, which add further overhead.

C. Creating a Separate Thread for Each Client

- **Scalability:** Creating a separate thread for each client is generally more scalable than creating separate processes. Threads are lighter weight and share the same address space.
- **Resource Usage:** Threads use less memory and have lower overhead compared to processes. The creation and destruction of threads are less resource-intensive.

- **Performance:** Context switching between threads is faster than between processes, leading to better performance in a highly concurrent server environment.
- **Complexity:** While easier to manage than 1000 processes, threading introduces complexities such as the need for synchronization mechanisms to manage shared data and avoid race conditions.
- **Responsiveness:** The server can be highly responsive as each client is handled by a dedicated thread, allowing for concurrent processing of client requests.

D. Hybrid Structure: Child Processes with Multiple Threads and Child Processes

- Offers good scalability by combining the benefits of processes and threads.
- Optimal resource usage by balancing the number of processes and threads.
- Allows for efficient handling of clients with a combination of processes and threads.
- Moderately complex due to the need to manage both processes and threads, but provides good flexibility and control.
- Can maintain high responsiveness by dedicating leaf child processes and threads to serve individual clients.

Question 4:

Multi-threading and Multi-process Programming

a. Reasons for Separate Registers, Stack, and Program Counter

In a multi-threaded process, separate registers, stack, and program counter are used for the following reasons:

- **Registers:** Each thread may have its own set of registers to store its local variables and intermediate computation results. This allows threads to execute independently without interfering with each other's data.
- **Stack:** Each thread typically has its own stack to store function call parameters, return addresses, and local variables. Using separate stacks ensures that each thread's function calls and local variables are isolated from other threads, preventing data corruption and conflicts.
- **Program Counter:** Each thread has its own program counter, which keeps track of the instruction being executed by that thread. Separate program counters allow threads to execute different code paths concurrently, enabling parallelism and avoiding synchronization issues.

b. Problems Faced by Multiple Child Processes

When multiple child processes try to update local and global variables, they may face the following problems:

Updating Local Variables

- **Isolation:** Each child process has its own address space, including its stack and local variables. Therefore, updates to local variables in one child process do not affect other child processes.
- **No Sharing:** Local variables are not shared between processes. Each process has its own copy of local variables, ensuring data integrity and avoiding conflicts.

Updating Global Variables

- **Potential Race Conditions:** If multiple child processes attempt to update the same global variable concurrently, race conditions may occur. Race conditions can lead to unpredictable behavior and data corruption.
- **Need for Synchronization:** To avoid race conditions, synchronization mechanisms such as locks, semaphores, or atomic operations are required to ensure that only one process updates the global variable at a time.

c. Problems Faced by Multiple Threads

Similarly, when multiple threads try to update local and global variables, they may face the following problems:

Updating Local Variables

- **Isolation:** Each thread has its own stack and local variables, ensuring that updates to local variables in one thread do not affect other threads.
- **No Sharing:** Like processes, threads do not share local variables. Each thread maintains its own copy of local variables, providing thread isolation and avoiding data conflicts.

Updating Global Variables

- **Race Conditions:** Multiple threads accessing and updating the same global variable concurrently can lead to race conditions, resulting in data corruption and inconsistent behavior.
- **Need for Synchronization:** To prevent race conditions, synchronization mechanisms such as mutexes, condition variables, or atomic operations are required to coordinate access to shared global variables among threads.

2

Question 5:

Effects of Main Thread Termination

a. Termination of Main Thread Before Child Processes and Threads

When the main thread terminates before the termination of child processes and threads:

- **Child Processes:** The child processes continue execution independently of the main thread. They become orphaned processes and are inherited by the systemd process (typically process ID 1). The termination of the main thread does not affect the execution or termination of child processes.
- **Subordinate Threads:** If the main thread exits before subordinate threads, the entire process terminates, and all threads, including the subordinate ones, are terminated abruptly. Subordinate threads do not continue execution once the main thread exits.

b. Examples of When Multithreading is Beneficial

Multithreading is beneficial in various scenarios where parallelism and concurrency can improve performance, responsiveness, and resource utilization. Here are five examples:

1. **GUI Applications:** Multithreading allows GUI applications to remain responsive while performing background tasks such as I/O operations, data processing, or network communication. For example, a web browser can use separate threads for rendering the UI and fetching web content.
2. **Server Applications:** Multithreading enables server applications to handle multiple client connections concurrently. Each client connection can be served by a separate thread, allowing the server to handle a large number of clients simultaneously without blocking.
3. **Parallel Processing:** Multithreading facilitates parallel processing of computationally intensive tasks. For example, in scientific simulations or data analysis applications, different threads can work on separate chunks of data in parallel, speeding up overall processing time.
4. **Real-Time Systems:** Multithreading is crucial in real-time systems where tasks must meet strict timing requirements. Using multiple threads allows for the concurrent execution of tasks with different priorities, ensuring timely response to events.
5. **Multimedia Applications:** Multithreading is essential for multimedia applications such as video playback, audio processing, and image manipulation. Separate threads can handle decoding, rendering, and user interaction concurrently, providing smooth and responsive multimedia experiences.

References

- [1] OpenAI. *ChatGPT: A Large Language Model*. Accessed: 2024-05-23. 2024. URL: <https://www.openai.com/chatgpt>
- [2] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. 9th. Hoboken, NJ: Wiley, 2012. ISBN: 978-1118063330.

CSE3241: Operating Systems Class Test

November 10, 2024

1 Question 1:

Question 1(a): Figure out how often context switches occur in your system and describe it with a screen shot.

To determine the frequency of context switches in our system we can use

```
vmstat 1
```

This will show system stats every 1 second. Looking at the cs (context switches) column, which shows the number of context switches per second. The screenshot is provided below:

```
sajjad@sajjad: ~  
sajjad@sajjad:~$ vmstat 1  
procs -----memory----- --swap-- -----io---- --system-- -----cpu-----  
---  
 r  b  swpd   free   buff  cache   si   so    bi    bo    in    cs  us  sy  id  wa  st  
gu  
0  0    768 651384 147328 3430084    0    0   315   265 4284   12 20   7 73   0  
0  0  
0  0    768 644396 147336 3434232    0    0     0  140 2724 2164   5  3 92   0  
0  0  
0  0    768 644396 147336 3434232    0    0     0    0 2572 1843   4  3 93   0  
0  0  
0  0    768 648744 147336 3430052    0    0     0   20 2729 2925   6  4 90   0  
0  0
```

Figure 1: output-1

Question 1(b): Context Switch Frequency for a specific process:

To determine the frequency of context switches for a specific process in our system, I created a simple C program that spawns a child process and performs some CPU-bound work. By monitoring this child process, we can observe the context switches.

C Program

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

void busy_work() {
    for (int i = 0; i < 10000000000; i++) {
    }
}

int main() {
    pid_t pid = fork();

    if (pid < 0) {
        perror("fork failed");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        printf("Child process PID: %d\n", getpid());
        busy_work();
        printf("Child process completed\n");
        exit(EXIT_SUCCESS);
    } else {
        wait(NULL);
        printf("Parent process completed\n");
    }

    return 0;
}
```

Step-by-Step Procedure

1. **Compiled the Program:**
2. **Observed Context Switches:** I Used `pidstat` to monitor context switches for the specific PID of the child process. In this case, the child process PID was printed to the console.

```
pidstat -w -p <child_pid> 1
```

Example:

```
pidstat -w -p 21158 1
```

Observed Output

child pid : 21158.

```

sajjad@sajjad:~$ pidstat -w -p 21158 1
Linux 6.5.0-35-generic (sajjad) ... c 30/5/24 x _x86_64_ (4 CPU)

07:44:15 অম্মা হুণ +06 UID PID cswch/s nvcswh/s Command
07:44:16 অম্মা হুণ +06 1000 21158 0.00 #inc7.00e context o.h>
07:44:17 অম্মা হুণ +06 1000 21158 0.00 #inc6.00e context lib.h>
07:44:18 অম্মা হুণ +06 1000 21158 0.00 #inc11.00e context
07:44:19 অম্মা হুণ +06 1000 21158 0.00 #inc5.00e context td.h>
07:44:20 অম্মা হুণ +06 1000 21158 0.00 #inc10.00e context wait.h>
07:44:21 অম্মা হুণ +06 1000 21158 0.00 10.00 context
07:44:22 অম্মা হুণ +06 1000 21158 0.00 8.00 context
07:44:23 অম্মা হুণ +06 1000 21158 0.00 void test() {
07:44:24 অম্মা হুণ +06 1000 21158 0.00 for i = 0; i < 100
07:44:25 অম্মা হুণ +06 1000 21158 0.00 y work to gen

```

Figure 2: output-1

Explanation of the Screenshot

- The `cswch/s` column shows the number of voluntary context switches per second.
- The `nvcswh/s` column shows the number of non-voluntary context switches per second.
- The output demonstrates that the context switches occur at varying rates during the execution of the child process.

This observation shows how often context switches are occurring for the specific process, giving insights into the system's behavior while the process is running.

Question 1(c):

A. Processor Affinity

Processor affinity refers to the practice of keeping a thread running on the same processor to take advantage of the "warm cache." When a thread executes on a processor, the most recently accessed data is stored in that processor's cache, speeding up subsequent memory operations. If the thread migrates to another processor for load balancing, the cache of the original processor must be invalidated, and the cache of the new processor must be repopulated, which is an expensive operation. To avoid this overhead, operating systems with SMP (Symmetric Multiprocessing) aim to maintain thread locality, minimizing thread migration and optimizing performance by using processor affinity.

B. Dispatcher

The dispatcher is a key component of the operating system's process scheduler. It is responsible for giving control of the CPU to the process selected by the scheduler. Key features include:

Context switching involves the dispatcher saving the state of the current process and loading the next process to run. It ensures efficient CPU utilization by minimizing idle time through quick process switching. The dispatcher works with the scheduler to guarantee fair CPU time allocation and maintain system responsiveness. This helps in balancing process execution and system performance.

C. Starvation of a Process

Starvation, also known as indefinite blocking, occurs when a process is perpetually denied necessary resources to proceed with its execution results in indefinitely staying in waiting state. This may occur on priority scheduling algorithm where due to lower priority one process may not get the CPU access for a indefinite time. Key aspects of process starvation include:

- **Resource Contention:** Starvation typically occurs in systems where multiple processes compete for limited resources, and certain processes are continually overlooked.
- **Priority Inversion:** A situation where higher-priority processes are waiting for lower-priority ones to release resources, leading to potential starvation of the lower-priority processes.
- **Fairness Issues:** Starvation indicates a fairness issue in the scheduling algorithm, where some processes get disproportionately less CPU time or resources.
- **Solutions to Starvation:** Techniques such as aging (gradually increasing the priority of waiting processes) and fair scheduling algorithms can mitigate starvation.

1

Question 2a: Scheduling Algorithms

Consider the following set of processes, with the length of the CPU burst time given in milliseconds:

Process	Burst Time	Priority
P1	2	2
P2	1	1
P3	8	4
P4	4	2
P5	5	3

The processes are assumed to have arrived in the order P1, P2, P3, P4, P5, all at time 0.

A. Execution Order Using Different Scheduling Algorithms

1. First-Come, First-Served (FCFS)

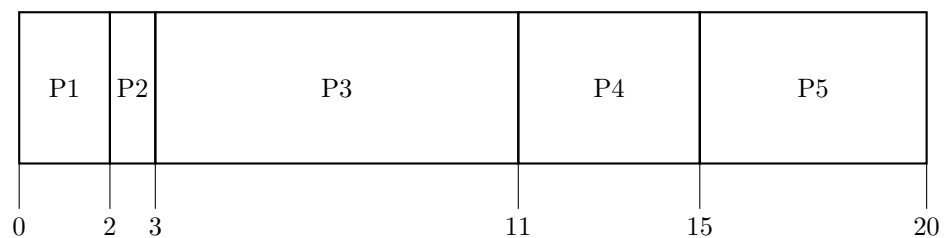


Figure 1: FCFS

2. Shortest Job First (SJF)

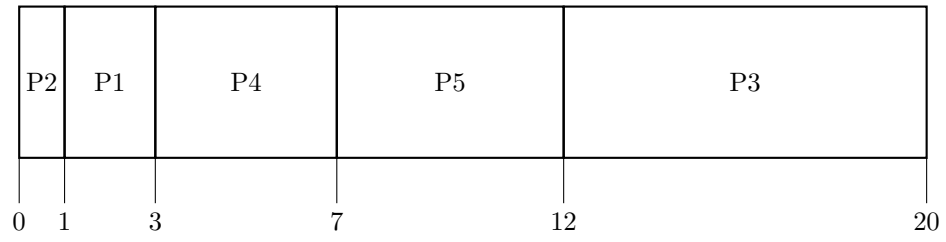


Figure 2: SJF Scheduling

3. Non-Preemptive Priority (Higher number = higher priority)

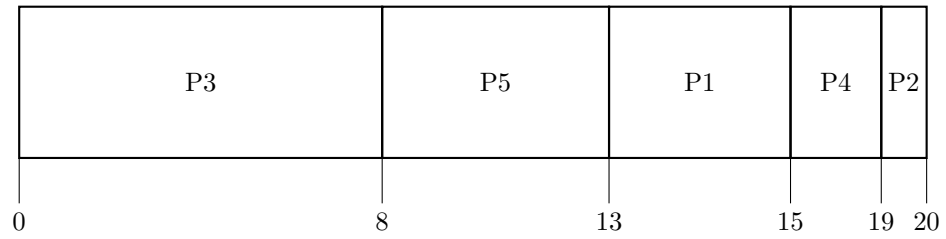


Figure 3: Non-Preemptive Priority (Higher number = higher priority)

4. Round Robin (RR) (Quantum = 2)

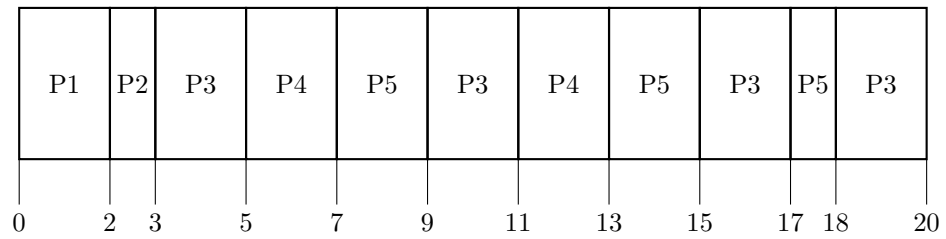


Figure 4: Round Robin (RR) Scheduling

B. Turnaround Time of Each Process

Process	FCFS	SJF	Non-Preemptive Priority	RR (Quantum = 2)
P1	2 ms	3 ms	15 ms	2 ms
P2	3 ms	1 ms	20 ms	3 ms
P3	11 ms	20 ms	8 ms	20 ms
P4	15 ms	7 ms	19 ms	13 ms
P5	20 ms	12 ms	13 ms	18 ms

C. Waiting Time of Each Process

Process	FCFS	SJF	Non-Preemptive Priority	RR (Quantum = 2)
P1	0 ms	1 ms	13 ms	0 ms
P2	2 ms	0 ms	19 ms	2 ms
P3	3 ms	12 ms	0 ms	12 ms
P4	11 ms	3 ms	15 ms	9 ms
P5	15 ms	7 ms	8 ms	13 ms

D. Minimum Average Waiting Time

Among the four scheduling algorithms, **Shortest Job First (SJF)** results in the minimum average waiting time. Here are the average waiting times for each algorithm:

- FCFS: $\frac{0+2+3+11+15}{5} = 6.2$ ms
- SJF: $\frac{1+0+12+3+7}{5} = 4.6$ ms
- Non-Preemptive Priority: $\frac{13+9+0+15+8}{5} = 11$ ms
- RR: $\frac{0+2+12+9+13}{5} = 7.2$ ms

Thus, the SJF scheduling algorithm results in the minimum average waiting time.

Question 3. a.

The traditional UNIX scheduler recalculates process priorities once per second using the following function:

$$\text{Priority} = \left(\frac{\text{recent CPU usage}}{2} \right) + \text{base}$$

where base = 60.

Given recent CPU usage for processes P_1 , P_2 , and P_3 as 40, 18, and 10 respectively, the new priorities are calculated as follows:

$$\text{Priority}_{P_1} = \left(\frac{40}{2} \right) + 60 = 80$$

$$\text{Priority}_{P_2} = \left(\frac{18}{2} \right) + 60 = 69$$

$$\text{Priority}_{P_3} = \left(\frac{10}{2} \right) + 60 = 65$$

Based on this information, the traditional UNIX scheduler lowers the relative priority of a CPU-bound process as the recent CPU usage increases.

b.

In the Linux scheduler, a red-black tree is used to maintain the set of processes with their priorities. This data structure allows efficient insertion, deletion, and searching of processes based on their priority values.

An alternative data structure that could be used is a heap (specifically a min-heap or max-heap, depending on the requirement). However, compared to a red-black tree, a heap is less efficient for operations like deletion and searching, which are crucial in a scheduler. Therefore, a red-black tree is preferred in the Linux scheduler for its balanced nature and efficient performance.

Question 4. a.

An ideal thread refers to a thread of execution within a process that can be independently scheduled by the operating system. It is ideal in the sense that it exhibits properties such as perfect parallelism, where multiple ideal threads can execute simultaneously on multiple processors without contention or synchronization overhead.

An ideal processor, on the other hand, is a theoretical concept representing a processor that can execute instructions infinitely fast, with unlimited resources, and with zero overhead. It is ideal because it can process any workload instantaneously without any constraints or limitations.

b.

Scheduling Order (Gantt Chart):

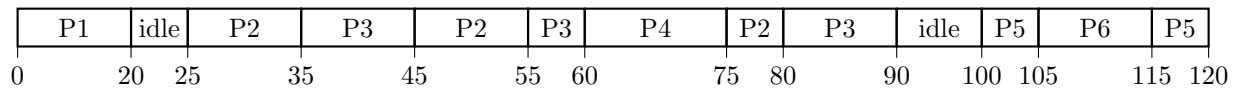


Figure: Scheduling Order (Gantt Chart)

Turnaround Time:

Process	Turnaround Time
P1	20 units
P2	55 units
P3	60 units
P4	15 units
P5	20 units
P6	10 units

c.

Waiting Time:

Process	Waiting Time
P1	0 units
P2	30 units
P3	35 units
P4	0 units
P5	10 units
P6	0 units

d.

CPU Utilization Rate:

The CPU utilization rate is calculated as the total time the CPU was busy divided by the total time. $\left(\frac{105}{120}\right) = 87.5\%$.

So, the CPU utilization rate is 87.5%.

References

- [1] OpenAI. *ChatGPT: A Large Language Model*. Accessed: 2024-05-30. 2024. URL: <https://www.openai.com/chatgpt>.

CSE3241: Operating Systems

Class Test

Name: Md Sajjad Hossain

ID: 2011176125

Session: 2019-20

June 04, 2024

Question-1: What advantages does Linux have of being a preemptive kernel?

A preemptive kernel allows a process to be preempted while it is running in kernel mode. This ability enhances the system's responsiveness and efficiency, allowing it to switch between tasks more fluidly and serve high-priority or time-sensitive tasks without delay.

The preemptive nature of the Linux kernel works by allowing the scheduler to take control at almost any point in kernel mode, making it possible to interrupt ongoing processes and allocate CPU time to tasks as needed. This design improves the system's multitasking capabilities and responsiveness, especially useful for real-time applications.

a. Advantages of Linux as a Preemptive Kernel

- **Responsiveness:** Linux as a preemptive kernel enhances system responsiveness by allowing higher priority tasks to preempt lower priority ones.
- **Fairness:** It ensures fair CPU allocation among tasks, preventing any single task from monopolizing the CPU.
- **Real-time Capabilities:** Preemptive kernels improve real-time application performance by ensuring timely task execution.

b. Issues with Parallel Execution of `i++` and `i--`

- When executing the instructions `i++` and `i--` in parallel on a shared variable `i`, a race condition can occur.
- Both `i++` and `i--` involve three steps: read, modify, and write.
- Without proper synchronization, the operations can overlap, leading to incorrect results.

For example:

Initial value of i : 5
Process 1 ($i++$ reads i) : 5
Process 2 ($i-$ reads i) : 5
Process 1 ($i++$ writes i) : 6
Process 2 ($i-$ writes i) : 4
Final value of i : 4 (should be 5)

- This leads to inconsistent and incorrect values due to the unsynchronized access. This happens because processor loads the value to specific register and then performs the increment and decrement operation and then it stores the data to actual location. If the loading is done simultaneously like the example it can result in a race condition.

Question-2. Illustrate three cases of race conditions.

a. Three Cases of Race Conditions

1. **Mutual exclusion:** If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress:** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. **Bounded waiting:** There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Question 3.

a. a. Show that, if the `wait()` and `signal()` semaphore operations are not executed atomically, then mutual exclusion may be violated.

The `wait()` and `signal()` semaphore operations need to be executed atomically to ensure mutual exclusion. If they are not executed atomically, it may lead to two processes ending up in the critical section at the same time.

Mutual exclusion is an important concept in operating systems that ensures that only one process can access a critical section at a time. Semaphores are used to implement mutual exclusion, where `wait()` and `signal()` operations are used to acquire and release the semaphore, respectively. If these operations are not executed atomically, a race condition may occur, leading to a violation of mutual exclusion.

Example: Let's consider two processes P_1 and P_2 , both of which are trying to enter a critical section protected by a semaphore. Suppose P_1 executes the `wait()` operation first and is blocked by the semaphore. At this point, the CPU switches to P_2 , which also executes the `wait()` operation and is blocked. Now, the CPU switches back to P_1 , which executes the `signal()` operation and releases the semaphore. However, before P_1 can enter the critical

section, the CPU switches back to P2, which executes the `signal()` operation and also releases the semaphore. Now, both P1 and P2 are unblocked, and there is nothing preventing them from entering the critical section simultaneously, violating mutual exclusion. To avoid this scenario, the `wait()` and `signal()` semaphore operations must be executed atomically. This can be achieved through hardware support or by disabling interrupts during the execution of these operations.

Code:

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t semaphore;
int shared_data = 0;

void* process_P1(void* arg) {
    printf("P1 attempting to enter critical section\n");
    sem_wait(&semaphore);
    printf("P1 entered critical section\n");
    sleep(1);
    printf("P1 leaving critical section\n");
    sem_post(&semaphore);
    return NULL;
}

void* process_P2(void* arg) {
    printf("P2 attempting to enter critical section\n");
    sem_wait(&semaphore);
    printf("P2 entered critical section\n");
    sleep(1);
    printf("P2 leaving critical section\n");
    sem_post(&semaphore);
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    sem_init(&semaphore, 0, 1);

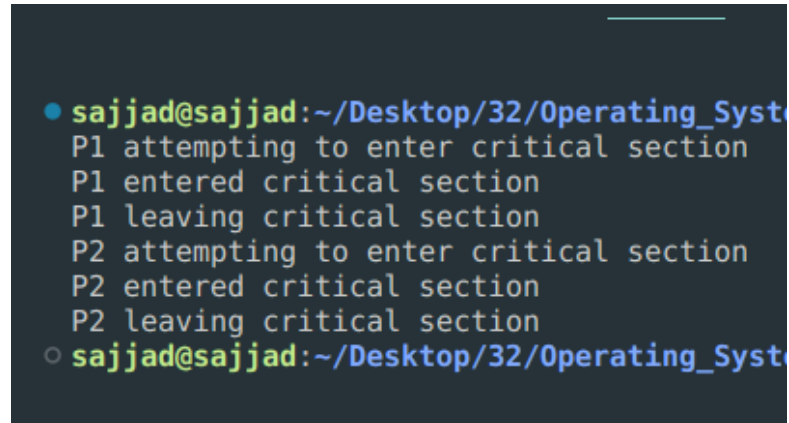
    pthread_create(&thread1, NULL, process_P1, NULL);
    sleep(1);
    pthread_create(&thread2, NULL, process_P2, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    sem_destroy(&semaphore);
}
```

```
    return 0;  
}
```

Output:



```
● sajjad@sajjad:~/Desktop/32/Operating_Systems  
P1 attempting to enter critical section  
P1 entered critical section  
P1 leaving critical section  
P2 attempting to enter critical section  
P2 entered critical section  
P2 leaving critical section  
○ sajjad@sajjad:~/Desktop/32/Operating_Systems
```

Figure 1: output-1

If the semaphore operations are not atomic, the output might incorrectly show both P1 and P2 entering the critical section simultaneously, which would demonstrate a race condition and violation of mutual exclusion. But this never happens because the semaphore operations are done atomically. As a result my code is working correctly without causing a race condition.

b.

In a multi-core system, the selection between a spinlock and a mutex lock depends on various factors, including the duration the lock needs to be held and the potential for threads to be put to sleep while holding the lock.

Scenario a) Short Duration: In situations where the lock is expected to be held for a short duration, a spinlock is often preferred. Spinlocks are lightweight and avoid the overhead of context switching associated with putting a process to sleep. Instead of relinquishing the CPU, a thread attempting to acquire a spinlock continuously polls the lock until it becomes available. This approach is efficient when the duration of the lock contention is brief and the overhead of context switching outweighs the benefits of putting the process to sleep. However, it's essential to ensure that spinlocks are used judiciously, as prolonged spinning can lead to excessive CPU consumption and potential performance degradation.

Scenario b) Long Duration: For locks that need to be held for a prolonged period, a mutex lock is typically more appropriate. Unlike spinlocks, mutex locks allow waiting processes to sleep while waiting for the lock to become available. This sleeping mechanism reduces CPU consumption and contention, making mutex locks preferable in scenarios where lock contention is expected to persist for an extended duration. By allowing waiting threads to be suspended, mutex locks facilitate efficient resource utilization and help prevent excessive CPU spinning, which can negatively impact overall system performance.

Scenario c) Threads Sleeping While Holding the Lock: In scenarios where threads may need to sleep while holding the lock, such as when waiting for I/O operations or synchronization with external events, a mutex lock is again the better choice. Unlike spinlocks, which cannot release the lock while a thread is holding it, mutex locks allow the lock to be released when a thread is put to sleep. This feature prevents potential deadlock situations where threads are indefinitely blocked due to their inability to release the lock. By allowing sleeping threads to relinquish the lock temporarily, mutex locks enhance system robustness and prevent resource contention issues that can arise from deadlock scenarios.

In summary, while spinlocks offer advantages in scenarios requiring short-duration locks, mutex locks are preferable for long-duration locks and situations involving potential thread sleeping while holding the lock. Careful consideration of these factors is essential when designing concurrent systems to optimize performance, resource utilization, and robustness.

Question-4.

a. Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user-level programs.

Issues with Interrupts for Synchronization Primitives

Single Processor Systems

In a single processor system, implementing synchronization primitives by disabling interrupts is not appropriate in user-level programs because it can lead to poor performance and potential deadlocks. Disabling interrupts blocks all interrupts, including those from the system kernel, which can prevent important system functions from executing. Additionally, disabling interrupts for an extended period of time can lead to missed interrupts, which can cause delays and other synchronization issues. Instead, user-level synchronization primitives should be implemented using more efficient and reliable methods, such as locking mechanisms or atomic operations.

b. Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.

Multiprocessor Systems

In a multiprocessor system, interrupts are not appropriate for implementing synchronization primitives because interrupts can be generated on any of the processors, which can lead to inconsistencies in shared data.

For example, if one processor is interrupted while it is updating a shared variable, and another processor tries to access that variable at the same time, the value of the variable may be inconsistent between the two processors. This can lead to race conditions and other synchronization issues.

Question 5.

Between using a mutex lock and an atomic integer to prevent race conditions on the shared variable 'hits', the atomic integer strategy is more efficient. Atomic operations run in constant

time, are thread-safe, and do not require locks.

Explanation: In computer programming, both mutex and atomic integer are used to ensure synchronization when multiple threads access shared data. However, the efficiency between the two varies. Using a mutex lock to update a shared variable is a traditional method and is safe but can add overhead as threads can possibly wait for the lock to be released.

On the other hand, the strategy of using an atomic integer is more efficient. Atomic operations run in constant time, they are thread-safe, do not require any locks and cannot be interrupted. Atomic instructions can be completed in a single operation ensuring that no other thread can change the state of the variable in the middle of the update. Thus, this strategy has less overhead of unnecessary waiting and is generally more efficient. [1]

References

- [1] *Brainly*. <https://brainly.com/question/36770166>. Accessed on June 4, 2024.

CSE3241: Operating Systems

Class Test

Name: Md Sajjad Hossain

ID: 2011176125

Session: 2019-20

June 09, 2024

Question 1:

a.

Answer: The atomic integer `val` starts with a value set to 10, then undergoes a series of operations resulting in a final value of 7.

Explanation: When dealing with an atomic integer in a Linux system, the operations described will sequentially modify the value of the integer in a thread-safe manner, preventing race conditions. Here's how the value changes after each operation:

- `atomic_set(&val, 10)`: Sets the value of `val` to 10.
- `atomic_sub(8, &val)`: Subtracts 8 from `val`, making the value now 2.
- `atomic_inc(&val)`: Increments `val` by 1, changing the value to 3.
- `atomic_inc(&val)`: Increments `val` by 1, changing the value to 4.
- `atomic_add(6, &val)`: Adds 6 to `val`, making the value 10.
- `atomic_sub(3, &val)`: Subtracts 3 from `val`, resulting in a final value of 7.

After all the operations, the value of `val` will be 7.

b.

Answer: The dining philosophers' problem is a classic example used to illustrate synchronization issues and the potential for deadlock in concurrent systems. Here's how deadlock can occur in this scenario:

1. **Setup:** Five philosophers sit around a circular table with a fork placed between each pair of adjacent philosophers. Each philosopher alternates between periods of thinking and eating.
2. **Rules:** To eat, a philosopher needs both the fork on their left and the fork on their right. They can only pick up one fork at a time and must put down both forks after eating.

3. **Deadlock Scenario:** If each philosopher simultaneously picks up the fork on their left, all forks become occupied. Since each philosopher now holds one fork, they all need the fork on their right to eat, but that fork is already held by their neighbor.
4. **Outcome:** As each philosopher waits for the fork on their right to become available, no one can proceed to eat, and no one will put down their left fork. This results in a cyclic dependency where every philosopher is waiting for a fork that another philosopher holds, causing the system to enter a deadlock state. In this state, no philosopher can eat, and they remain stuck indefinitely.

By carefully designing a solution, such as introducing rules to prevent all philosophers from picking up their left forks simultaneously, we can avoid deadlock. The dining philosophers' problem highlights the critical importance of considering synchronization and resource allocation in concurrent systems.

Some possible remedies to deadlock:

- Allow at most four philosopher to be sitting simultaneously at the table.
- Allow a philosopher to pick up chopstick only if both chopstick are available.
- Allow Odd position philosopher to pick left chopstick first and even position philosopher to pick right chopstick first.

c.

Answer:

Operating systems like Windows and Linux use different locking mechanisms such as spinlocks, mutex locks, semaphores, and condition variables. Each has unique characteristics that make them suitable for various synchronization scenarios.

Spinlocks:

Spinlocks are simple locks used when a thread is expected to release the lock quickly. They work by making the thread that needs the lock wait in a loop, continuously checking if the lock is available. This waiting consumes CPU cycles because the thread keeps running without giving up control. Spinlocks are useful when dealing with multiple CPUs or cores, as they reduce delays and avoid the overhead of switching between threads. However, if the lock isn't released quickly, spinlocks can waste CPU resources and degrade performance. They are best for situations with short-term contention where the lock will be held only briefly.

Mutex Locks:

Mutex locks block the thread that wants the lock if it isn't available, allowing the CPU to work on other tasks. This makes them ideal for situations where a thread might need the lock for a longer time, as they prevent unnecessary CPU usage. While mutex locks save CPU cycles, they can cause issues like priority inversion (where a lower-priority thread holds a lock needed by a higher-priority thread), deadlocks (where two or more threads wait indefinitely for each other), livelocks (where threads keep changing states without making progress), and starvation (where a thread never gets the lock). Mutex locks are suitable for long-term contention when it's acceptable to block the CPU.

Semaphores:

Semaphores control access to shared resources using a counter, allowing multiple threads to wait on the same semaphore. They are useful for managing limited resources like network sockets, printing queues, or database connections. Counting semaphores, a type of semaphore, can count up and down, acting as a simple counter to coordinate tasks requiring a specific count

or quota. Semaphores help prevent race conditions (where the outcome depends on the sequence of events) and improve system stability by managing concurrent tasks effectively.

Condition Variables:

Condition variables allow threads to wait for specific conditions to be met without consuming CPU cycles. They work with mutex locks or read-write locks, putting threads to sleep until a condition changes. Condition variables are particularly useful in scenarios like producer-consumer models, where threads need to wait for resources to become available or for data to be ready. By avoiding busy waiting, condition variables promote efficient resource use and improve overall system performance.

Operating systems use these different locking mechanisms because they offer distinct advantages depending on the situation. Spinlocks are good for short waits with low overhead, mutex locks handle longer waits while saving CPU resources, semaphores manage access to shared resources and prevent race conditions, and condition variables allow efficient waiting for specific conditions. Choosing the right mechanism helps maximize system performance and minimize delays.

Question 2:

a. The content of the matrix Need can be calculated by subtracting the Allocation matrix from the Max matrix:

$$\text{Need} = \text{Max} - \text{Allocation}$$

Here's the Need matrix:

	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>
T_0	0	0	0	0
T_1	0	7	5	0
T_2	1	0	0	2
T_3	0	0	2	0
T_4	0	6	4	2

b. To determine if the system is in a safe state, we will use the Banker's Algorithm:

1. Set the Available vector equal to the Available vector:

$$\text{Available} = [1 \quad 5 \quad 2 \quad 0]$$

2. Find a process T_i for which $\text{Need}_i \leq \text{Available}$ and $\text{Finish}_i = \text{false}$.
3. If such a process exists, set $\text{Finish}_i = \text{true}$ and $\text{Available} = \text{Available} + \text{Allocation}_i$. Repeat step 2.
4. If all processes have $\text{Finish}_i = \text{true}$, then the system is in a safe state.

Applying the algorithm, we get the following safe sequence:

$$T_0 \rightarrow T_3 \rightarrow T_4 \rightarrow T_2 \rightarrow T_1$$

c. To check if the request from P_1 for $(0, 4, 2, 0)$ can be granted immediately, we compare it to the Available vector:

$$\text{Request} = [0 \quad 4 \quad 2 \quad 0]$$

$$\text{Available} = [1 \quad 5 \quad 2 \quad 0]$$

Since the request is within the available resources, the request can be granted immediately.

Question 3:

a. Can a computer system detect when processes are starving?

Answer: Yes, a computer system can detect when some of its processes are starving. Process starvation occurs when a process is unable to obtain the resources it needs to complete its execution, leading to it being stuck or delayed indefinitely.

How to avoid process starvation in a computer system? To avoid process starvation, various scheduling algorithms can be implemented in the operating system:

- **Priority-based scheduling:** Processes with higher priorities are given access to resources before lower priority processes.
- **Time-slice or round-robin scheduling:** Ensures that each process gets a fair share of CPU time.

b. As an operating system developer, what do you prefer: deadlock avoidance, deadlock prevention, or recovery from deadlock? Justify your answer.

As an operating system developer, my preference would lean towards deadlock Ignorance also known as ostrich method. Most of operating system follow this strategy in case of deadlock. This is because deadlock is a case is less likely to occur on operating system. So there's no need to write any code to handle deadlock. Considering this OS developer handle deadlock as if there is no deadlock scenario would likely to occur on the system, like the ostrich bird avoid sand storm by putting their head inside the sand as if nothing happening outside. I will go with this method. But as ignorance method is not mentioned in the question I may prefer deadlock prevention in the second position. Deadlock prevention strategies aim to structure the system in such a way that deadlocks become impossible or at least highly unlikely to occur. This proactive approach helps maintain system stability and prevents the wastage of resources that can result from deadlocks. There are 4 condition for deadlock.

- Mutual Exclusion.
- No preemption.
- Hold and wait.
- Circular wait.

If we somehow can prevent this 4 condition then there will be no deadlock occurring in our system. So secondly I would prefer this method.

c. Answer: Differences between Spinlocks and Mutex Locks:

a) Spinlocks:

- Spinlocks are busy-waiting locks where a thread repeatedly checks to acquire the lock without yielding the CPU.

- They are more efficient in scenarios where the lock is expected to be held for a short duration.
- They may cause wastage of CPU cycles if the lock is held for an extended period.
- Suitable for scenarios with low lock contention and short critical sections.

b) Mutex Locks:

- Mutex locks provide mutual exclusion, allowing only one thread to access a resource at a time.
- They support blocking mechanisms where a thread waits until it can acquire the lock, thus avoiding busy waiting.
- More suitable for scenarios with high lock contention and longer critical sections.
- They incur context-switching overhead when a thread is blocked waiting for the lock.

Differences between Deadlock and Livelock:

a) Deadlock:

- Deadlock occurs when two or more threads are blocked indefinitely, each waiting for the other to release a resource.
- The threads involved in a deadlock remain blocked until external intervention occurs.
- It leads to a complete halt in program execution and requires intervention, such as manual termination of processes or preemptive resource allocation.

b) Livelock:

- Livelock is a situation where two or more threads continuously change their states in response to the actions of the other threads, but none make progress.
- Threads in a livelock are not blocked but are unable to complete their tasks due to constant interaction.
- It often occurs in distributed systems or with concurrent algorithms when multiple threads are attempting to resolve a resource conflict.

Question 4:

Given the resource-allocation policy:

Requests for and releases of resources are allowed at any time. If a request for resources cannot be satisfied because the resources are not available, then we check any threads that are blocked waiting for resources. If a blocked thread has the desired resources, then these resources are taken away from it and are given to the requesting thread. The vector of resources for which the blocked thread is waiting is increased to include the resources that were taken away.

For example, a system has three resource types, and the vector Available is initialized to (4, 2, 2). If thread T0 asks for (2, 2, 1), it gets them. If T1 asks for (1, 0, 1), it gets them. Then, if T0 asks for (0, 0, 1), it is blocked (resource not available). If T2 now asks for (2, 0, 0), it gets the available one (1, 0, 0), as well as one that was allocated to T0 (since T0 is blocked). T0's Allocation vector goes down to (1, 2, 1), and its Need vector goes up to (1, 0, 1).

(a) Can deadlock occur?

Answer:

No, deadlock cannot occur under this policy. Deadlock requires all four necessary conditions to be present:

- Mutual Exclusion.
- No preemption.
- Hold and wait.
- Circular wait.

In this policy, the condition of **No Preemption** is not satisfied because resources can be preempted from blocked threads and allocated to other requesting threads. Therefore, deadlock cannot occur as one of the necessary conditions is not met.

(b) Can indefinite blocking occur?

No, indefinite blocking (or starvation) will not occur under this policy. The policy allows resources to be taken from blocked threads and reallocated to other requesting threads. This means that even if a thread is blocked, the resources it holds can be reassigned to other threads, ensuring that all threads will eventually obtain the resources they need to proceed. Consequently, no thread will be indefinitely blocked.

Question 5:

Solution to Assignment: Banker's Algorithm

Given the following system snapshot:

Allocation Matrix

<i>T</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>
<i>T0</i>	3	0	1	4
<i>T1</i>	2	2	1	0
<i>T2</i>	3	1	2	1
<i>T3</i>	0	5	1	0
<i>T4</i>	4	2	1	2

Max Matrix

<i>T</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>
<i>T0</i>	5	1	1	7
<i>T1</i>	3	2	1	1
<i>T2</i>	3	3	2	1
<i>T3</i>	4	6	1	2
<i>T4</i>	6	3	2	5

Need Matrix (Max - Allocation)

T	A	B	C	D
$T0$	2	1	0	3
$T1$	1	0	0	1
$T2$	0	2	0	0
$T3$	4	1	0	2
$T4$	2	1	1	3

We need to determine if the given states are safe or unsafe using the Banker's algorithm.

(a) Available = (0, 3, 0, 1)

1. If $T0$ can be satisfied:

Need = (2, 1, 0, 3), Available = (0, 3, 0, 1) \implies $T0$ cannot be satisfied.

2. If $T1$ can be satisfied:

Need = (1, 0, 0, 1), Available = (0, 3, 0, 1) \implies $T1$ can not be satisfied.

3. If $T2$ can be satisfied:

Need = (0, 2, 0, 0), Available = (0, 3, 0, 1) \implies $T2$ can be satisfied.

Update Available:

New Available = (3, 4, 2, 2)

4. If $T1$ can be satisfied:

Need = (1, 0, 0, 1), Available = (3, 4, 2, 2) \implies $T3$ can be satisfied.

Update Available:

New Available = (5, 6, 3, 2)

5. If $T3$ can be satisfied:

Need = (4, 1, 0, 2), Available = (5, 6, 3, 2) \implies $T4$ can be satisfied.

Update Available:

New Available = (5, 11, 4, 2)

No other process can be satisfied. So there is no safe sequence in this scenario.

(b) Available = (1, 0, 0, 2)

1. If $T1$ can be satisfied:

Need = (1, 0, 0, 1), Available = (1, 0, 0, 2) \implies $T1$ can be satisfied.

Update Available:

New Available = (3, 2, 1, 2)

3. If $T2$ can be satisfied:

Need = (0, 2, 0, 0), Available = (3, 2, 1, 2) \implies $T2$ can be satisfied.

Update Available:

$$\text{New Available} = (6, 3, 3, 3)$$

4. If $T3$ can be satisfied:

$$\text{Need} = (4, 1, 0, 2), \quad \text{Available} = (6, 3, 3, 3) \implies T3 \text{ can be satisfied.}$$

Update Available:

$$\text{New Available} = (6, 8, 4, 3)$$

5. If $T4$ can be satisfied:

$$\text{Need} = (2, 1, 1, 3), \quad \text{Available} = (6, 8, 4, 3) \implies T4 \text{ can be satisfied.}$$

Update Available:

$$\text{New Available} = (10, 10, 5, 5)$$

6. If $T0$ can be satisfied:

$$\text{Need} = (2, 1, 0, 3), \quad \text{Available} = (10, 10, 5, 5) \implies T0 \text{ can be satisfied.}$$

Thus, the safe sequence is:

$$T1 \rightarrow T2 \rightarrow T3 \rightarrow T4 \rightarrow T0$$

Therefore, the state is **safe** with the safe sequence: $T1, T2, T3, T4, T0$.

Question 6:

(a) Arguments for Installing the Deadlock-Avoidance Algorithm

1. Prevention of Deadlocks:

- Implementing the deadlock-avoidance algorithm can minimize or eliminate the occurrence of deadlocks. This ensures that jobs can be completed without being terminated due to deadlocks, resulting in higher productivity and efficiency.

2. Cost Reduction:

- Currently, the operator has to terminate and rerun about ten jobs per deadlock, resulting in wasted CPU time and potential financial losses. By implementing the deadlock-avoidance algorithm, the number of deadlocks can be significantly reduced, thereby reducing the associated costs of termination and rerunning of jobs.

3. Increased Job Completion:

- The termination of jobs during deadlocks means that valuable CPU time is wasted. By avoiding deadlocks, more jobs can be completed, leading to increased job throughput and better resource utilization.

(b) Arguments against Installing the Deadlock-Avoidance Algorithm

1. Increased Execution Time:

- Installing the deadlock-avoidance algorithm would result in an average increase of 10 percent in the execution time per job. This could lead to longer turnaround times for individual jobs and potentially affect the overall system performance.

2. Increased Turnaround Time:

- Implementing the deadlock-avoidance algorithm would result in an average increase of 20 percent in the turnaround time. This could impact the responsiveness of the system and potentially result in delays in job completion.

3. Idle Time:

- Currently, the machine has 30 percent idle time, meaning there is still available capacity to run all 5,000 jobs per month. Implementing the deadlock-avoidance algorithm may decrease the idle time and potentially lead to resource underutilization.

Conclusion

The decision to install the deadlock-avoidance algorithm depends on weighing the benefits of preventing deadlocks and reducing costs against the potential drawbacks of increased execution and turnaround times. [1]

References

- [1] Brainly. <https://brainly.com/question/36770166>. Accessed on June 9, 2024.

CSE3241: Operating Systems

Class Test

Name: Md Sajjad Hossain

ID: 2011176125

Session: 2019-20

June 25, 2024

Question 2: Page Numbers and Offsets for Given Address References

Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):

Here the 1 KB (2^{10} B) page size indicates that the lower order 10 bits are used for page offset and the remaining higher order bits are for page number. Now to Determine the page number and page offset we consider the decimal converting in binary and then taking lower order 10 bits for page offset and the remaining for page number. The representations are given bellow:

Address (decimal)	Address (binary)	Page Number (binary/decimal)	Page Offset (binary/decimal)
3085	1100 0000 1101	11 / 3	00 0000 1101 / 13
42095	1010 0100 0110 1111	1010 01 / 41	00 0110 1111 / 111
215201	0011 0100 1000 1010 0001	0011 0100 10 / 210	00 1010 0001 / 161
650000	1001 1110 1011 0000 0000	1001 1110 10 / 634	11 0000 0000 / 784
2000001	1 1110 1000 0100 0000 0001	1 1110 1000 01 / 1953	00 1000 0001 / 129

Table 1: Address Decoding

Question 3(a). Calculating the Number of Page Table Entries

a) Conventional Single-Level Page Table

For a conventional single-level page table, the number of entries can be calculated by dividing the total addressable memory by the page size. Given that the operating system can address 2^{21} KB of memory and the page size is 2 KB (2^{11} bytes), the number of entries is:

$$\text{Number of entries} = \frac{2^{21}}{2^{11}} = 2^{10} = 1024 \text{ entries}$$

b) Inverted Page Table

In an inverted page table, the number of entries is determined by the number of frames in physical memory rather than the size of the virtual address space. Given a physical address space of 16 bits and a page size of 2 KB (2^{11} bytes), the number of entries is:

$$\text{Number of entries} = \frac{2^{16}}{2^{11}} = 2^5 = 32 \text{ entries}$$

Maximum Amount of Physical Memory

The maximum amount of physical memory can be determined by considering the maximum number of page table entries and the page size. With 1024 entries and each page being 2 KB, the maximum physical memory is:

$$\text{Maximum physical memory} = 2^{10} \times 2^6 = 64 \text{ KB}$$

b. Avoiding Internal and External Fragmentation

Internal Fragmentation

Internal fragmentation happens in fixed-length partitioning when allocated memory blocks are larger than the actual data they store. Since partitions are of a fixed size, any unused space within an allocated partition remains unusable, leading to wasted memory. For instance, if a process requires 12 KB and is allocated a 16 KB partition, 4 KB remains unused, causing internal fragmentation. This can accumulate across multiple partitions, significantly reducing efficient memory usage. While simple to implement, fixed-length partitioning often results in substantial memory waste due to this issue.

To avoid internal fragmentation, variable-length partitioning allocates memory based on the exact size required by each process, minimizing unused space within partitions. This approach dynamically adjusts partition sizes, reducing wasted memory. However, it may lead to external fragmentation, where free memory is scattered.

External Fragmentation

External fragmentation happens when we use variable length partitioning. Due to the contiguous allocation of memory small holes are created in the partition. These small holes lead to wastage of memory as when a process comes with a large size that does not fit in any holes we can't give access to memory for that process. Because the contiguous place should be there to give access to memory for that process. For this, it results in external fragmentation.

To avoid external fragmentation we can use the worst fit algorithm. Also, there are strategies like compaction to avoid external fragmentation. However, this strategy is so costly that it takes a lot of CPU to allocate and deallocate memory for the whole system. Another way to avoid external fragmentation is having noncontiguous memory allocation. One of the non-contiguous memory allocation strategies is dividing the process into pages known as paging.

Question 4(a). Comparison of Memory Allocation Strategies

In terms of efficient use of memory, Best-fit is the most efficient (leaving a free memory space of 777 KB and all processes completely assigned), followed by First-fit (which also leaves 777 KB but available in smaller partitions), and then Worst-fit (which leaves a free space of 1152 KB but cannot assign one of the processes). The details are as follows:

Explanation

We have six free memory partitions: 300 KB (F1), 600 KB (F2), 350 KB (F3), 200 KB (F4), 750 KB (F5), and 125 KB (F6) (in order).

Using First-fit

First-fit assigns the first available memory partition that can fit a process. Best-fit assigns the smallest available memory partition that can fit a process.

Worst-fit assigns the largest available memory partition to a process.

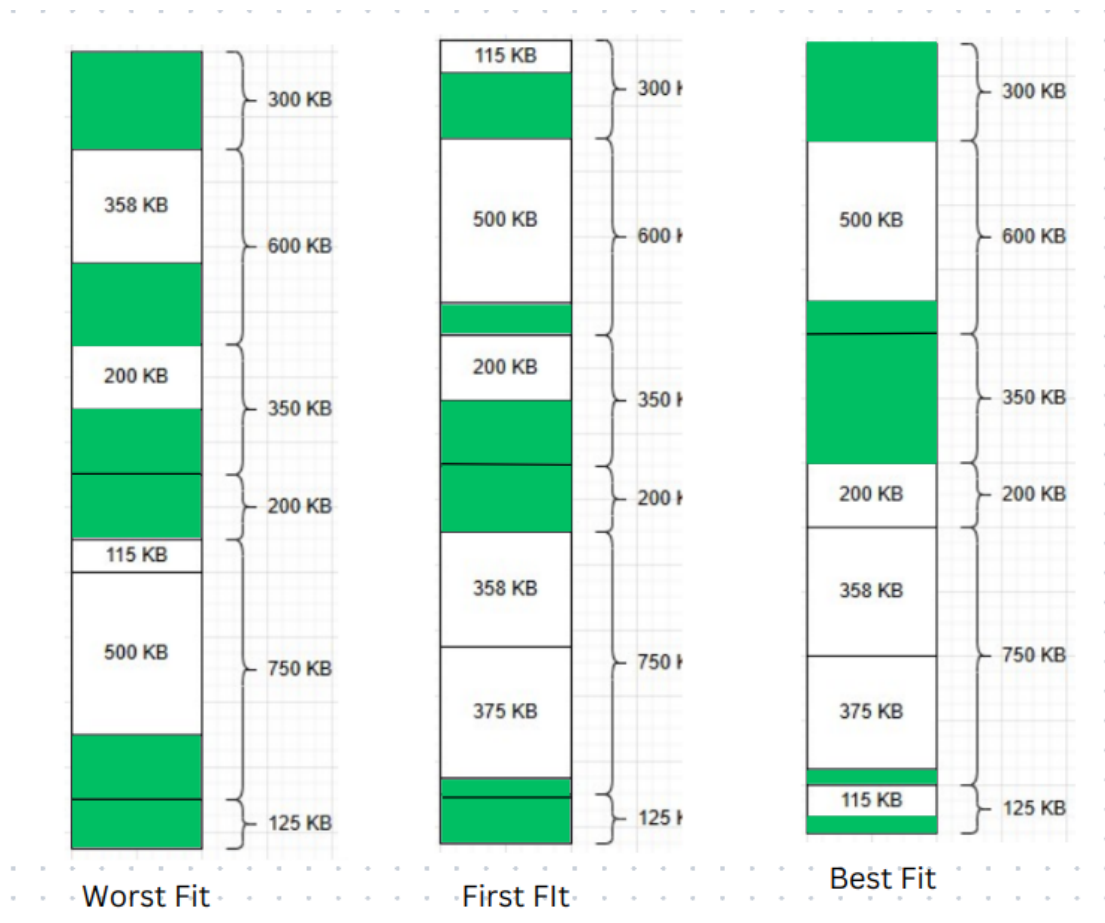


Figure 1: Worst Fit, First Fit and Best Fit

So in the case of the worst fit algorithm, we can't give memory to the 375kB process. It must be put into the waiting queue. The other two algorithms can easily satisfy our process needs.

b. Logical and Physical Address Bit Calculation

The logical address requires 16 bits to represent it, while the physical address requires 15 bits.

a) Logical Address

To determine the number of bits in the logical address, we need to consider the number of pages in the logical address space.

Here number of pages is 64 (2^6) and the page offset is 1024 (2^{10}).

Thus, the logical address consists of:

$$6 \text{ bits (page number)} + 10 \text{ bits (offset)} = 16 \text{ bits}$$

b) Physical Address

For the physical address, we need to calculate the number of bits required to represent the frame number. There are 32 (2^5) frames in physical memory.

Therefore, the physical address consists of:

$$5 \text{ bits (frame number)} + 10 \text{ bits (offset)} = 15 \text{ bits}$$

Question 5: a. Logical and Physical Address Bit Calculation

Logical Address

There are 256 (2^8) pages and the page size is 4 KB (2^{12}). Then logical address will be $8+12 = 20$ bits. Therefore, 20 bits are required in the logical address.

Physical Address

To determine the number of bits required in the physical address, we need to find the number of frames. Given that the physical memory has 64 (2^6) frames: Hence $6+12 = 18$ bits required for physical address.

b. Page Table Size Calculation

Given data:

- Logical address = 32-bit = 2^{32} bytes
- Page size = 4 KB = 2^{12} bytes
- Physical memory = 512 MB = 2^{29} bytes

Calculation

(a) Number of Pages

To calculate the number of pages:

$$\begin{aligned} \text{Number of pages} &= \frac{\text{Logical address space}}{\text{Page size}} \\ &= \frac{2^{32}}{2^{12}} \\ &= 2^{20} \text{ pages} \end{aligned}$$

(b) Number of Frames

To calculate the number of frames:

$$\begin{aligned} \text{Number of frames} &= \frac{\text{Physical memory}}{\text{Page size}} \\ &= \frac{2^{29}}{2^{12}} \\ &= 2^{17} \text{ frames} \end{aligned}$$

Conclusion

Therefore, in the context of memory management:

- Conventional single-level page table requires 2^{20} pages.
- Inverted page table requires 2^{17} frames. [\[1\]](#)

Question 6:

A. Standard swapping and Swapping with paging

Standard swapping: In standard swapping, entire processes are moved between main memory (RAM) and secondary storage (usually disk) as a whole unit. This is typically done when the system needs to free up memory space for other processes.

Swapping with paging: Swapping with paging involves moving data between main memory and secondary storage in smaller fixed-size units called pages. Pages are typically smaller than the entire process and are managed by the operating system to efficiently utilize memory and reduce overhead.

B. Logical address and physical address

Logical address: A logical address is an address generated by the CPU. It refers to the location in the virtual address space of a process. It consists of a page number and an offset within the page, and it does not directly correspond to physical memory locations.

Physical address: A physical address is the actual location in the physical memory (RAM) where data is stored. It is generated after the address translation process by mapping the logical address to its corresponding physical address. This is done by Memory Management Unit (MMU).

C. Internal fragmentation and external fragmentation

Internal fragmentation: Internal fragmentation happens in fixed-length partitioning when allocated memory blocks are larger than the actual data they store. Since partitions are of a fixed size, any unused space within an allocated partition remains unusable, leading to wasted memory. For instance, if a process requires 12 KB and is allocated a 16 KB partition, 4 KB remains unused, causing internal fragmentation. This can accumulate across multiple partitions, significantly reducing efficient memory usage. While simple to implement, fixed-length partitioning often results in substantial memory waste due to this issue.

External fragmentation: External fragmentation happens when we use variable length partitioning. Due to the contiguous allocation of memory small holes are created in the partition. These small holes lead to wastage of memory as when a process comes with a large size that does not fit in any holes we can't give access to memory for that process. Because the contiguous place should be there to give access to memory for that process. For this, it results in external fragmentation

Question 1(a). Advantages and Disadvantages of Separate Base-Limit Registers for Code and Data

The major advantage of this scheme is that it is an effective mechanism for code and data sharing. For example, only one copy of an editor or a compiler needs to be kept in memory, and this code can be shared by all processes needing access to the editor or compiler code. Another advantage is the protection of the code against erroneous modification. The disadvantage is that the code and data must be separated, which is usually adhered to in compiler-generated code.

b. Why Page Sizes Are Powers of 2

Page sizes in memory management are always powers of 2 primarily due to the following reasons: Page sizes are typically powers of 2 to simplify address translation. Every address generated by the CPU can be split into a page number and an offset, with the page number identifying the page and the offset locating the position within that page. By using a power of 2 for the page size, the offset can be directly represented by the lower-order bits of the address, while the higher-order bits represent the page number. This division allows efficient and fast computation, as it only requires bitwise operations rather than complex arithmetic. For example, a page size of 2^n uses the lower n bits for the offset.

In summary, using powers of 2 for page sizes aligns well with the binary nature of computer systems, facilitating efficient and straightforward memory management and hardware optimization.

c. Effect of Multiple Page Table Entries Pointing to the Same Page Frame

Shared Page Frame

Allowing two entries in a page table to point to the same page frame results in the sharing of that physical memory frame between different virtual addresses. This shared mapping has significant implications:

- **Efficient Memory Copying:** Instead of physically copying a large amount of memory from one place to another, both source and destination can point to the same memory frame. This technique, known as *copy-on-write*, reduces the time and resources needed for copying.
- **Impact of Updates:** If a byte in one of the shared pages is updated, the change is immediately reflected in the other page because both page table entries reference the same physical memory. This can lead to unintended side effects if the pages are expected to be independent after copying.

2

References

- [1] Brainly. <https://brainly.com> Accessed on June 25, 2024.

- [2] OpenAI. *ChatGPT: A Large Language Model*. Accessed: 2024-06-25, 2024. URL: <https://www.openai.com/chatgpt>.

CSE3241: Operating Systems

Class Test

Name: Md Sajjad Hossain

ID: 2011176125

Session: 2019-20

June 26, 2024

Question 1(a)

Memory Allocation using First-Fit Algorithm

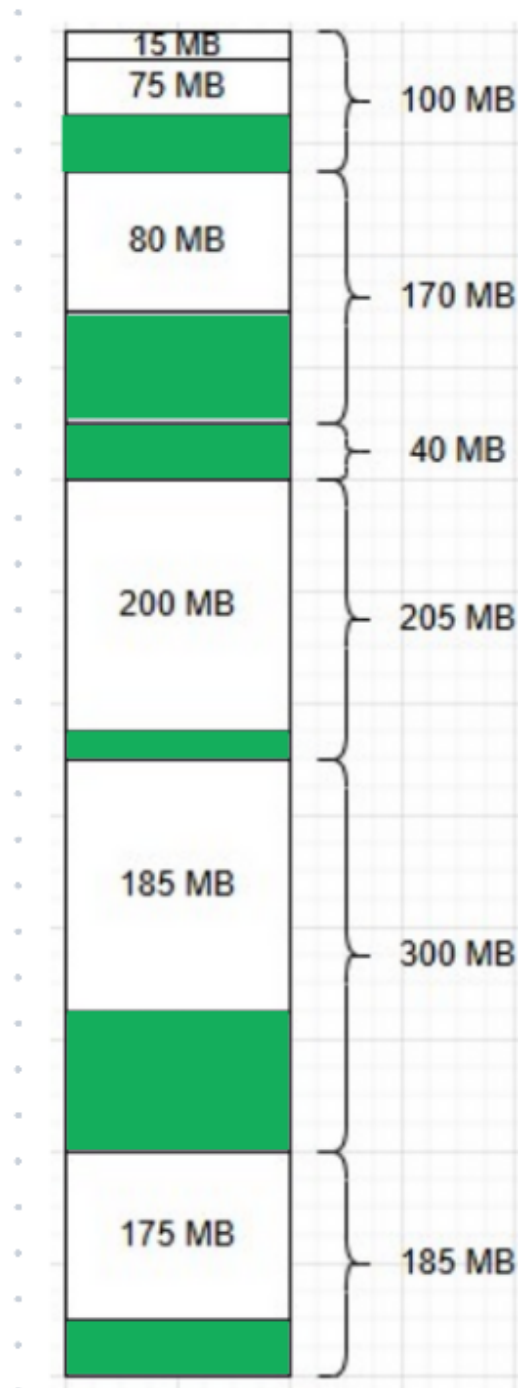


Figure 1: First Fit

Memory Allocation using Best-Fit Algorithm

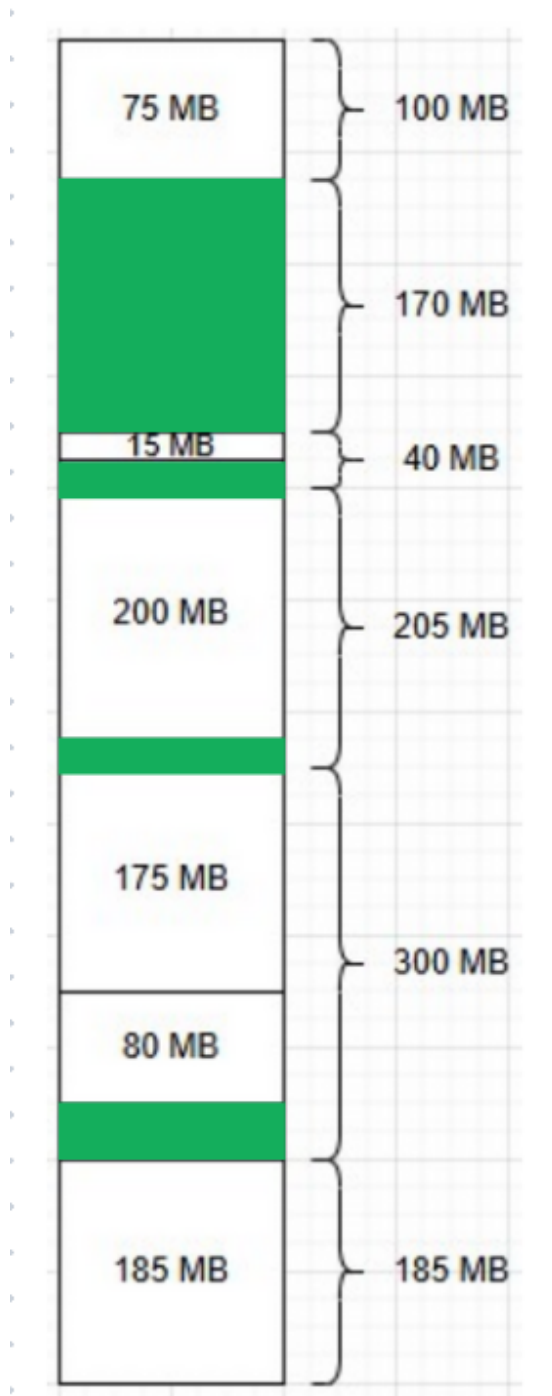


Figure 2: Best Fit

Memory Allocation using Worst-Fit Algorithm

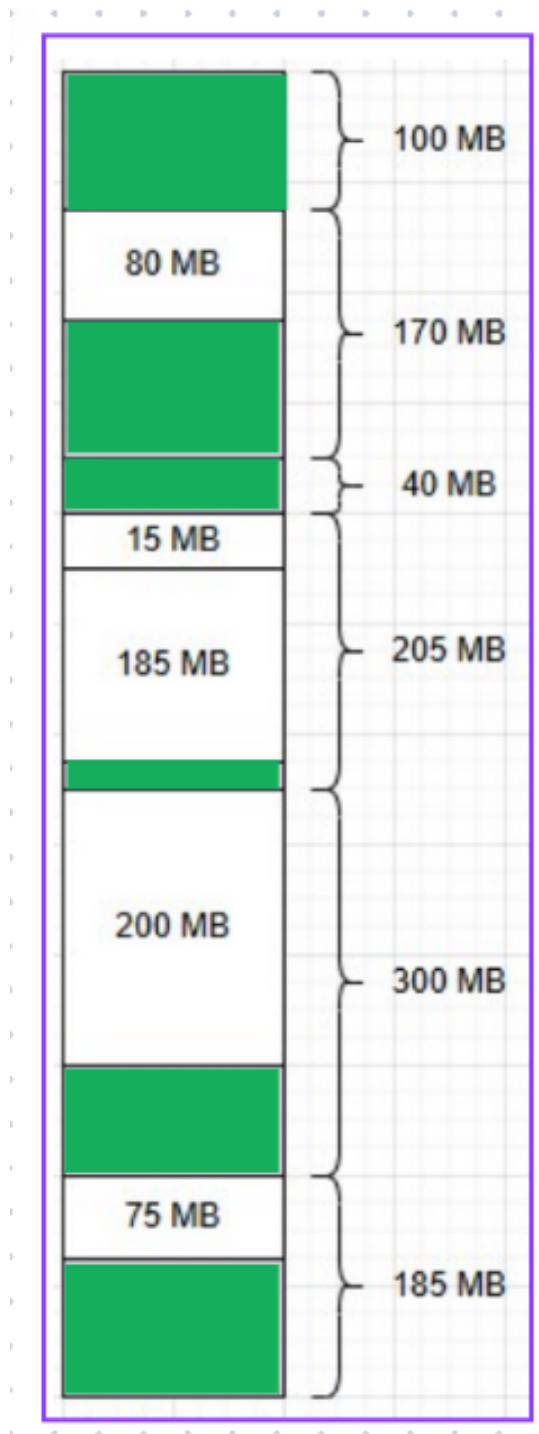


Figure 3: Worst Fit

In this case 175Mb process cant be allocated to any available memory space. So it must wait until any of the processes leave the memory by completing it's task.

Efficiency Comparison

The Best-Fit algorithm is the most efficient in terms of minimizing wasted space, while the Worst-Fit algorithm is the least efficient as not all memory is allocated properly.

1(b) Question

Fragmentation on a storage device can be eliminated through compaction. Typical disk devices do not have relocation or base registers (such as those used when memory is to be compacted), so how can we relocate files? Give three reasons why compacting and relocating files are often avoided.

Answer

Disk fragmentation is managed by the operating system using a file system to track file locations. During defragmentation or compaction, files are moved and rearranged on the disk to occupy contiguous areas. These changes are made in the background while the files appear unchanged in their directories.

Explanation

When it comes to disk fragmentation, the operating system (OS) manages file allocation on the hard drive. Unlike memory compaction, which may use relocation or base registers, disk defragmentation relies on the file system—a crucial component of the OS. The file system maintains a record of where each piece of a file is stored on the disk. Essentially, it is a data structure (information index) that tracks all files and their respective locations.

During disk defragmentation or compaction, the OS moves and rearranges fragmented files so they occupy contiguous areas on the disk. This process is handled by defragmentation software, either built into the OS or provided by third-party applications. Although the physical locations of the files on the disk change, the file system updates to reflect these new locations. From the user's perspective, the files appear unchanged in their original directories.

Reasons Why Compacting and Relocating Files Are Often Avoided

1. Performance Overhead:

The defragmentation process can be time-consuming and resource-intensive, especially for large disks with significant fragmentation. It requires moving large amounts of data, which can slow down system performance while the process is running.

2. Wear and Tear on Storage Devices:

For traditional hard disk drives (HDDs), the mechanical movement involved in relocating files can lead to increased wear and tear. For solid-state drives (SSDs), the frequent writing involved in defragmentation can reduce the lifespan of the drive due to the limited number of write cycles SSDs can endure.

3. Risk of Data Corruption:

There is always a small risk of data corruption during the defragmentation process, particularly if there is a power failure or system crash. This risk makes it crucial to ensure proper backups are in place before defragmenting.

By managing fragmentation, the operating system ensures more efficient file access and better overall performance. However, the trade-offs, such as performance overhead, potential hardware wear, and risk of data corruption, often lead to defragmentation being performed less frequently and with caution.

1(c)

By analyzing the data structures, the system can identify contiguous blocks of unallocated memory and construct a new free-space list. If the pointer to the free-space list is lost, the system will not be able to directly reconstruct the free-space list. The pointer acts as a reference to the location of the list, allowing the system to access and manipulate it. Without the pointer, the system loses the ability to locate the free-space list in memory.

However, there are indirect ways to reconstruct the free-space list. One approach is to scan the entire memory and identify blocks of free space by examining the memory allocation data structures. For example, the system may maintain a bitmap or a linked list to keep track of allocated and free memory blocks.

Question 2:

a. Will the thread change state if it incurs a page fault? If so, to what state will it change?

Yes, the thread will change state if it incurs a page fault. It will change to the **blocked** state because the thread needs to wait for the required page to be loaded from the disk into memory.

b. Will the thread change state if it generates a TLB miss that is resolved in the page table? If so, to what state will it change?

No, the thread will not change state if it generates a TLB miss that is resolved in the page table. The resolution of a TLB miss by checking the page table is typically handled within the processor, and the thread remains in the **running** state during this process.

c. Will the thread change state if an address reference is resolved in the page table? If so, to what state will it change?

No, the thread will not change state if an address reference is resolved in the page table. The resolution of an address reference by the page table is a normal part of memory access and does not require the thread to change state. The thread remains in the **running** state.

Question 3: (a)

How do caches help improve performance?

Caches help improve performance by providing faster access to frequently used data. The primary advantages include:

- **Reduced Latency:** Caches are typically much faster than main memory, allowing the processor to access data more quickly.
- **Increased Throughput:** By keeping frequently accessed data closer to the processor, caches reduce the time the processor spends waiting for data, thereby increasing overall system throughput.
- **Locality of Reference:** Caches exploit the principle of locality, where programs tend to access the same set of data or instructions repeatedly over short periods of time (temporal locality) or access data locations that are close to each other (spatial locality).

Why do systems not use more or larger caches if they are so useful?

Despite their usefulness, there are several reasons why systems do not use more or larger caches:

- **Cost:** Caches are made from faster and more expensive memory technologies, such as SRAM, which increases the overall cost of the system.
- **Diminishing Returns:** As the size of the cache increases, the incremental performance gains decrease. Larger caches may have longer access times, which can reduce the benefits of increased cache size.
- **Complexity:** Managing larger caches increases the complexity of the cache control logic, which can lead to higher power consumption and more difficult design and verification processes.
- **Physical Space:** Larger caches require more physical space on the processor die, which can limit the amount of space available for other critical components and features.
- **Thrashing:** Due to Thrashing effect the page fault increase hence reduce the performance.

3(b)

Answer:

64 terabytes

Explanation:

Given data:

- Size of the disk block = 8 KB
- Size required by the pointer to a disk block = 4 bytes

Now,

Size required by 12 direct disk blocks = 12×8 KB

Size required by single indirect disk block = 2048×8 KB

Size required by double indirect disk block = $2048 \times 2048 \times 8$ KB

Size required by triple indirect disk block = $2048 \times 2048 \times 2048 \times 8$ KB

Now, the maximum size of the file that can be stored in this file system is:

$$\begin{aligned} &12 \times 8 \text{ KB} + 2048 \times 8 \text{ KB} + 2048 \times 2048 \times 8 \text{ KB} + 2048 \times 2048 \times 2048 \times 8 \text{ KB} \\ &= 64 \text{ terabytes} \end{aligned}$$

3(c)

When one user thread in a process incurs a page fault while accessing its stack, the other user threads belonging to the same process would also be affected by the page fault, and they would have to wait for the faulting page to be brought into memory.

This is because in the many-to-one mapping of user threads to kernel threads, all user threads share the same address space. When one thread incurs a page fault, it means that the required page is not present in the physical memory, and it needs to be fetched from disk. Since all user threads within the same process share the same address space, they all use the same virtual memory, and thus, the page fault affects all threads.

Therefore, any page faults that occur in one user thread of a process will affect all other user threads within the same process, and they will have to wait for the required page to be loaded into memory before they can resume execution.

Question 4(a)

Consider a file consisting of 100 blocks. Assuming the file-control block (and index block, in the case of indexed allocation) is already in memory, we calculate the number of disk I/O operations required for contiguous, linked, and indexed (single-level) allocation strategies under various operations.

For contiguous allocation:

- Adding a block at the beginning requires reading all 100 blocks to shift them by one position, totaling 201 I/O operations (100 reads + 100 writes + 1 write for the new block).
- Adding a block in the middle requires reading and writing 101 blocks (1 read + 100 writes), totaling 101 I/O operations.
- Adding a block at the end requires only 1 write operation for the new block, totaling 1 I/O operation.
- Removing a block from the beginning similarly requires reading all 100 blocks to shift them, totaling 198 I/O operations (100 reads + 98 writes).
- Removing a block from the middle requires reading and writing 98 blocks (1 read + 97 writes), totaling 98 I/O operations.

- f. Removing a block from the end requires no additional I/O operations, totaling 0 I/O operations.

For linked allocation:

- a. Adding a block at any position requires 1 write operation to update the previous block's pointer, totaling 1 I/O operation.
- b. Adding and removing a block in the middle or end requires 1 read and 1 write operation (to update pointers), totaling 2 I/O operations per operation.

For indexed allocation:

- a. Adding a block at any position requires 1 write operation to update the index block, totaling 1 I/O operation.
- b. Removing a block from any position requires 1 read operation to update the index block, totaling 1 I/O operation.

In summary, contiguous allocation generally requires more I/O operations for insertions and deletions at the beginning and middle due to block movements, while linked and indexed allocation typically require fewer operations due to their direct pointer or index updates.

4(b)

Suppose that your replacement policy (in a paged system) is to examine each page regularly and discard it if it has not been used since the last examination. What would you gain and what would you lose by using this policy rather than LRU or second-chance replacement?

Such an algorithm could be implemented with the use of a reference bit. After every examination, the bit is set to zero; it is set back to one if the page is referenced. The algorithm would then select an arbitrary page for replacement from the set of unused pages since the last examination.

Advantages:

- **Simplicity:** This algorithm is straightforward and requires minimal overhead since only a reference bit needs to be maintained for each page.

Disadvantages:

- **Ignored Locality:** It ignores locality of reference by only considering a short time frame (since the last examination) to decide whether to evict a page. This could lead to premature eviction of pages that are part of a process's working set but haven't been referenced recently.

In contrast to this policy, LRU (Least Recently Used) and second-chance replacement strategies consider the actual usage patterns of pages over a longer period, thus better reflecting the temporal locality of reference. However, these strategies require more complex data structures (like timestamps or circular queues) to maintain and may incur higher overhead.

Question 5: (a)

Consider the following page-replacement algorithms. Rank these algorithms on a five-point scale from "bad" to "perfect" according to their page-fault rate. Separate those algorithms that suffer from Belady's anomaly from those that do not. Belady's anomaly refers to the situation where increasing the number of allocated frames can result in more page faults instead of fewer.

- **LRU replacement:** Low page-fault rate as it replaces the least recently used page. Does not suffer from Belady's anomaly. **Rating:** 4 (Good).
- **FIFO replacement:** High page-fault rate, especially when the oldest pages are still frequently used. Suffers from Belady's anomaly. **Rating:** 2 (Poor).
- **Optimal replacement:** Perfect page-fault rate by replacing the page that won't be used for the longest. Does not suffer from Belady's anomaly. **Rating:** 5 (Perfect).
- **Second-chance replacement:** Moderate page-fault rate, as it approximates LRU with a second chance mechanism. Does not suffer from Belady's anomaly. **Rating:** 3 (Average).

5(b)

Assuming demand paging with three frames, the number of page faults that would occur for the following replacement algorithms are:

ii FIFO:

	7	2	3	1	2	5	3	4	6	7	7	1	0	5	4	6	2	3	0	1
F ₁	7	7	7	1	1	1	1	1	6	6	6	6	0	0	0	6	6	6	0	0
F ₂		2	2	2	2	5	5	5	5	7	7	7	7	5	5	5	2	2	2	1
F ₃			3	3	3	3	3	4	4	4	4	1	1	1	4	4	4	3	3	3
	m	m	m	m	H	m	H	m	m	m	H	m	m	m	m	m	m	m	m	m

Total page fault = 17

9) = 11.500 29.07 1.01.07

ii optimal:

	7	2	3	1	2	5	3	4	6	7	7	1	0	5	4	6	2	3	0	1
F ₁	7	7	7	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
F ₂		2	2	2	2	5	5	5	5	5	5	5	5	5	4	6	2	3	3	3
F ₃			3	3	3	3	3	4	6	7	7	7	0	0	0	0	0	0	0	0
	m	m	m	m	H	m	H	m	m	m	H	H	m	H	m	m	m	m	H	H

Total page fault = 43

Figure 4: Best Fit

#LRU

	7	2	3	1	2	5	3	4	6	7	7	1	0	5	4	6	2	3	0	1
F ₁	7	7	7	1	1	1	3	3	3	7	7	7	7	5	5	5	2	2	2	1
F ₂		2	2	2	2	2	2	4	4	4	4	1	1	1	4	4	4	3	3	3
F ₃			3	3	3	5	5	5	6	6	6	6	0	0	0	6	6	6	0	0
	m	m	m	m	H	m	m	m	m	m	H	m	m	m	m	m	m	m	m	m

Total page fault = 18

Figure 5: Best Fit

- **LRU Replacement Algorithm:** 18 Page Faults.
- **FIFO Replacement Algorithm:** 17 Page Faults.
- **Optimal Replacement Algorithm:** 13 Page Faults.

5(c): Explanation of Minor and Major Page Faults

In demand paging systems, both minor and major page faults occur when a requested page is not found in memory and must be loaded from secondary storage (such as disk). The difference between minor and major page faults lies in how they are resolved and their impact on the system's performance.

- **Minor Page Faults:** Occur when the requested page is already in memory but is currently not marked as such in the page table. Resolving a minor page fault involves simply updating the page table to mark the page as present and restarting the faulting instruction. This process is relatively quick because no data transfer is needed from disk.
- **Major Page Faults:** Occur when the requested page is not in memory and must be brought in from disk. Resolving a major page fault involves a longer process of disk I/O operations,

including seeking the page on disk, transferring it to memory, and updating the page table. This process is slower and more time-consuming compared to minor page faults.

Why Minor Page Faults Take Less Time to Resolve:

- Minor page faults are resolved by simply updating the page table without any disk I/O operations. The page is already present in memory; it just needs to be marked as such. This operation is typically handled within the context of the operating system's memory management routines and can be resolved quickly.
- In contrast, major page faults require fetching the page from disk, which involves significant overhead due to disk access latency and data transfer time. The time to resolve a major page fault is directly dependent on the speed of the disk and the size of the page being transferred.
- Therefore, minor page faults are resolved much faster than major page faults because they do not involve the time-consuming process of disk I/O. This efficiency in handling minor faults contributes to the overall responsiveness and performance of demand paging systems.



References

- [1] OpenAI. *ChatGPT: A Large Language Model*. Accessed: 2024-06-26. 2024. URL: <https://www.openai.com/chatgpt>.