



NAME: Adwait S Purao

UID: 2021300101

Batch: B2/B

Experiment: 3

Aim:

Demonstrate the usage of fork() system call and thus show Zombie and Orphan processes.

Theory:

A system call is a mechanism provided by the operating system that allows a program or process to request a service or resource from the kernel. In other words, it's an interface between a user program and the operating system, allowing the program to access the hardware and low-level services of the computer system.

When a program makes a system call, it transitions from user mode to kernel mode, where it can execute privileged instructions and access resources that are not available to user-mode programs. The system call interface provides a set of functions that can be used by a program to interact with the operating system, such as creating a new process, reading or writing files, allocating memory, and performing network operations.

Examples of system calls include **fork**, **wait**, **sleep**, **open**, **read**, **write**, and **socket**. These system calls allow programs to perform tasks that require privileged access to the system's resources, such as creating new processes, managing memory, and interacting with hardware devices.

Fork:

The fork system call is used to create a new process by duplicating the calling process. After the fork call, there are two processes running independently. The new process is called the child process, and the original process is called the parent process. The child process is an exact copy of the parent



process except for a few differences like its PID and the value returned by the fork system call.

Wait:

The wait system call is used by a parent process to wait for its child process to terminate. The parent process suspends its execution until the child process exits or terminates. When the child process terminates, the parent process resumes its execution and retrieves the exit status of the child process. The exit status contains information about the child process, such as the reason for its termination.

Sleep:

The sleep system call is used to suspend the execution of a process for a specified period of time. During the sleep period, the process is in a blocked state and does not consume CPU resources. Once the sleep period has elapsed, the process is unblocked and resumes its execution.



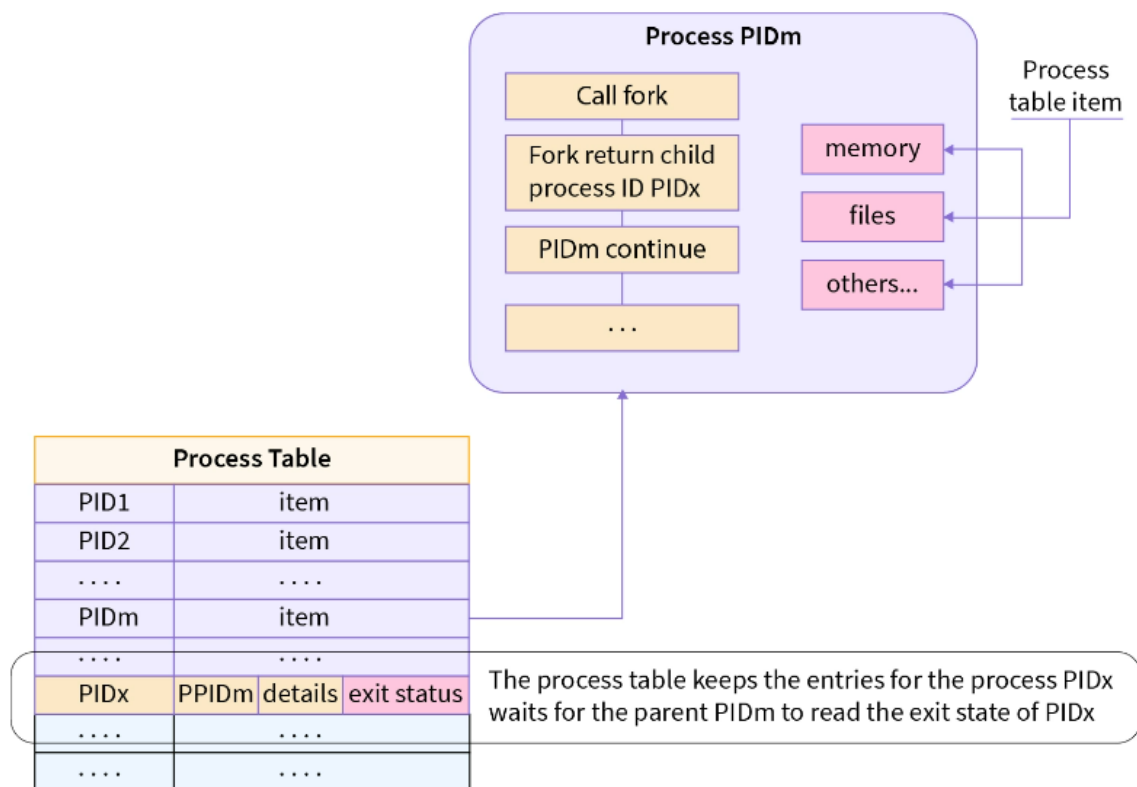
System Call	Purpose	Parameters	Returns	Process State
wait()	Waits for a child process to terminate	None	Exit status of the child process	Blocked
sleep()	Suspends process execution for a specified time	Time duration	0	Blocked

Sleep	Wait
Used to pause execution for a specified time.	Used to pause execution until a certain condition is met.
Pauses execution for a specified amount of time.	Pauses execution until a specified signal or event is received.
Program does not consume resources during sleep.	Program consumes resources during wait.
Program does not respond to input during sleep.	Program can respond to input during wait.
Implemented as a blocking call.	Implemented as a non-blocking call.
Usually used for simulating delays or limiting resources.	Usually used for waiting on a condition or event.
Sleep periods are usually imprecise.	Wait conditions are usually precise.
Sleep periods are specified in milliseconds or seconds.	Wait conditions can be specified in various ways.
Actual sleep time may be longer or shorter than specified.	Program continues to execute while waiting for condition to occur.

What is the Zombie process?

Zombie process is also known as **"dead"** process. Ideally when a process completes its execution, its entry from the process table should be removed but this does not happen in case of zombie process.

Analogy: Zombie, mythological, is a dead person revived physically. Similarly, a zombie process in os is a dead process (completed its execution) but is still revived (its entry is present in memory).

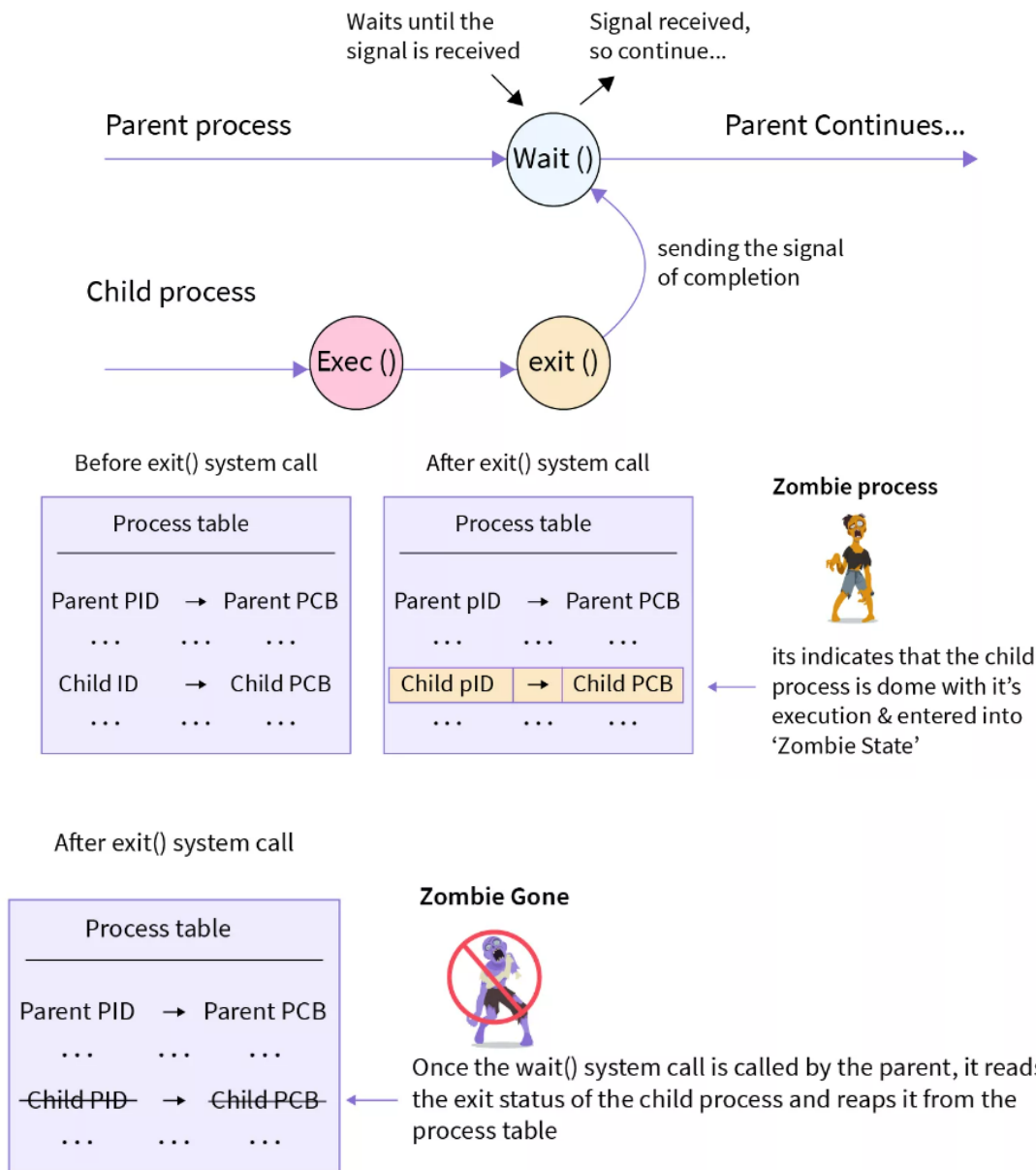


Note: Process table is a data structure in RAM to store information about a process.

What happens with the zombie processes?

- wait() system call is used for removal of zombie processes.
- wait() call ensures that the parent doesn't execute or sits idle till the child process is completed.

- When the child process completes executing, the parent process removes entries of the child process from the process table. This is called "reaping of child".



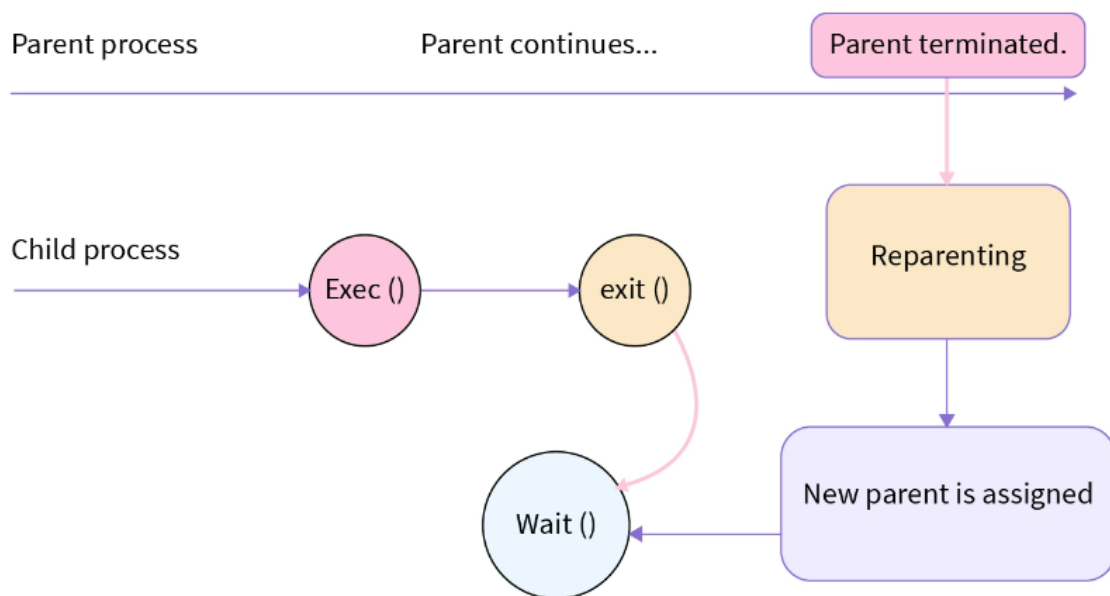
What is the Orphan Process?

We'll again use real life analogy to understand the orphan process.

- In the real world orphans are those children whose parents are dead.
- Similarly, a process which is executing (is alive) but its parent process has terminated (dead) is called an orphan process.

What will happen with the orphan processes?

- In the real world orphans are adopted by guardians who look after them.
- Similarly, the orphan process in linux is adopted by a new process, which is mostly init process (pid=1). This is called **re-parenting**.
- Reparenting is done by the kernel, when the kernel detects an orphan process in os, and assigns a new parent process.
- New parent process asks the kernel for cleaning of the PCB of the orphan process and the new parent waits till the child completes its execution.



Question 1.

Implement the C program in which main program accepts the integers to be sorted. Main program uses the fork system call to create a new process called a child process. Parent process sorts the integers using merge sort and waits for child process using wait system call to sort the integers using quick sort. Also demonstrate zombie and orphan states.

Code :

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



```
#include<sys/wait.h>

void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];
    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void merge_sort(int arr[], int l, int r)
{
    if (l < r)
    {
```



```
        int m = 1 + (r - 1) / 2;
        merge_sort(arr, l, m);
        merge_sort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

void quick_sort(int arr[], int low, int high)
{
    if (low < high)
    {
        int pi = arr[high];
        int i = (low - 1);
        for (int j = low; j <= high - 1; j++)
        {
            if (arr[j] <= pi)
            {
                i++;
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;
        int pid = fork();
        if (pid == 0)
        {
            printf("Child process sorting...\n");
            quick_sort(arr, low, i);
            exit(0);
        }
        else
        {
            printf("Parent process sorting...\n");
            merge_sort(arr, i + 2, high);
            printf("Parent process waiting for child process...\n");
            wait(NULL);
        }
    }
}

int main()
{
    int n;
    printf("Enter the number of integers to be sorted: ");
    scanf("%d", &n);
```




```
int arr[n];
printf("Enter %d integers to be sorted: ", n);
for (int i = 0; i < n; i++)
{
    scanf("%d", &arr[i]);
}
printf("Before sorting:\n");
for (int i = 0; i < n; i++)
{
    printf("%d ", arr[i]);
}
quick_sort(arr, 0, n - 1);
printf("\nAfter sorting:\n");
for (int i = 0; i < n; i++)
{
    printf("%d ", arr[i]);
}
return 0;
}
```

Output:

```
adwait@adwait: ~/Documents/OS EXPERIMENTS
adwait@adwait:~/Documents/OS EXPERIMENTS$ gcc 0531.c
adwait@adwait:~/Documents/OS EXPERIMENTS$ ./a.out
Enter the number of integers to be sorted: 6
Enter 6 integers to be sorted: 3 2 1 6 4 8
Before sorting:
3 2 1 6 4 8 Parent process sorting...
Parent process waiting for child process...
3 2 1 6 4 8 Child process sorting...
Parent process sorting...
Parent process waiting for child process...
Child process sorting...
Parent process sorting...
Parent process waiting for child process...
Child process sorting...
After sorting:
3 2 1 6 4 8 adwait@adwait:~/Documents/OS EXPERIMENTS$ ps -a1
  PID TTY          STAT TIME  COMMAND
    1 ?           Ss   0:03 /sbin/init splash
 1666 tty2        Ssl+  0:00 /usr/libexec/gdm-wayland-session env GNOME_SHELL_SE
 1678 tty2        Sl+   0:00 /usr/libexec/gnome-session-binary --session=ubuntu
 5898 pts/0        Ss   0:00  bash
 7932 pts/0        R+   0:00  ps -a1
adwait@adwait:~/Documents/OS EXPERIMENTS$
```

Modifying the **quick_sort** function to create a zombie and an orphan process:

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



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

void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];
    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void merge_sort(int arr[], int l, int r)
{

```



```
    if (l < r)
    {
        int m = l + (r - l) / 2;
        merge_sort(arr, l, m);
        merge_sort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

void quick_sort(int arr[], int low, int high)
{
    if (low < high)
    {
        int pi = arr[high];
        int i = (low - 1);
        for (int j = low; j <= high - 1; j++)
        {
            if (arr[j] <= pi)
            {
                i++;
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;
        int pid = fork();
        if (pid == 0)
        {
            printf("\nChild process sorting...\n");
            quick_sort(arr, low, i);
            sleep(10); // Create a zombie process by sleeping for 10
seconds after child process completes its work
            printf("\nChild process exiting...\n");
            exit(0);
        }
        else
        {
            printf("\nParent process sorting...\n");
            merge_sort(arr, i + 2, high);
            printf("\nParent process waiting for child process...\n");
        } // Create an orphan process by calling wait outside the
parent process's else block
        wait(NULL);
        printf("\nParent process exiting...\n");
    }
}
```



```
    }  
}  
int main()  
{  
    int n;  
    printf("\nEnter the number of integers to be sorted: ");  
    scanf("%d", &n);  
    int arr[n];  
    printf("\nEnter %d integers to be sorted: ", n);  
    for (int i = 0; i < n; i++)  
    {  
        scanf("%d", &arr[i]);  
    }  
    printf("\nBefore sorting:\n");  
    for (int i = 0; i < n; i++)  
    {  
        printf("%d ", arr[i]);  
    }  
    quick_sort(arr, 0, n - 1);  
    printf("\nAfter sorting:\n");  
    for (int i = 0; i < n; i++)  
    {  
        printf("%d ", arr[i]);  
    }  
    return 0;  
}
```

Output

For Zombie Process:



```
adwait@adwait: ~/Documents/OS EXPERIMENTS
Child process exiting...
Parent process exiting...
Child process exiting...
Parent process exiting...

After sorting:
adwait@adwait:~/Documents/OS EXPERIMENTS$ ./a.out
Enter the number of integers to be sorted: 6
Enter 6 integers to be sorted: 5 3 2 1 53 4
Before sorting:
5 3 2 1 53 4
Parent process sorting...
Parent process waiting for child process...
5 3 2 1 53 4
Child process sorting...
Parent process sorting...
Parent process waiting for child process...
Child process sorting...
Child process exiting...
Parent process exiting...
Child process exiting...
Parent process exiting...

After sorting:
3 2 1 4 5 53 adwait@adwait:~/Documents/OS EXPERIMENTS$ ps aux | egrep "Z|defunct"
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
adwait    8411  0.0  0.0  17864 2468 pts/0    S+   20:11   0:00 grep -E --color=auto Z|defunct
adwait@adwait:~/Documents/OS EXPERIMENTS$
```

For Orphan Process:

```
adwait@adwait: ~/Documents/OS EXPERIMENTS
After sorting:
3 2 1 4 5 53 adwait@adwait:~/Documents/OS EXPERIMENTS$ ps aux | egrep "Z|defunct"
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
adwait    8411  0.0  0.0  17864 2468 pts/0    S+   20:11   0:00 grep -E --color=auto Z|defunct
adwait@adwait:~/Documents/OS EXPERIMENTS$ ps -eo pid,ppid,cmd | awk 'S2==1 && S1!=1 {print $0}'
266      1 /lib/systemd/systemd-journald
324      1 /lib/systemd/systemd-udev
534      1 /lib/systemd/systemd-oomd
535      1 /lib/systemd/systemd-resolved
664      1 /usr/libexec/accounts-daemon
665      1 /usr/sbin/acpid
667      1 avahi-daemon: running [adwait.local]
668      1 /usr/sbin/cron -f -P
669      1 @dbus-daemon --system --address=systemd: --nofork --nopidfile --systemd-activation --syslog-only
671      1 /usr/sbin/NetworkManager --no-daemon
678      1 /usr/sbin/irqbalance --foreground
680      1 /usr/bin/python3 /usr/bin/networkd-dispatcher --run-startup-triggers
682      1 /usr/libexec/polkitd --no-debug
683      1 /usr/libexec/power-profiles-daemon
684      1 /usr/sbin/rsyslogd -n -iNONE
688      1 /usr/libexec/switcheroo-control
693      1 /lib/systemd/systemd-logind
698      1 /usr/libexec/udisks2/udisksd
701      1 /sbin/wpa_supplicant -u -s -O /run/wpa_supplicant
761      1 /usr/sbin/ModemManager
763      1 /usr/sbin/cupsd -l
768      1 /usr/bin/python3 /usr/share/unattended-upgrades/unattended-upgrade-shutdown --wait-for-signal
831      1 /usr/sbin/cups-browsed
837      1 /usr/sbin/kerneloops --test
839      1 /usr/sbin/kerneloops
1087      1 /usr/bin/VBoxDRMClient
1089      1 /usr/sbin/VBoxService --pidfile /var/run/vboxadd-service.sh
1107      1 /usr/sbin/gdm3
1160      1 /usr/libexec/rtkit-daemon
1296      1 /usr/libexec/upowerd
1359      1 /usr/libexec/packagekitd
1507      1 /usr/libexec/colord
1601      1 /lib/systemd/systemd --user
1652      1 /usr/bin/gnome-keyring-daemon --daemonize --login
2423      1 /usr/libexec/fwupd/fwupd
7533      1 /usr/lib/snapd/snapd
adwait@adwait:~/Documents/OS EXPERIMENTS$
```

Question 2



Implement the C program in which main program accepts an integer array. Main program uses the fork system call to create a new process called a child process. Parent process sorts an integer array and passes the sorted array to child process through the command line arguments of execve system call. The child process uses execve system call to load new program that uses this sorted array for performing the binary search to search the particular item in the array.

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

void bubble_sort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n - 1; i++)
    {
        for (j = 0; j < n - i - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

int binary_search(int arr[], int l, int r, int x)
{
    if (r >= l)
    {
        int mid = l + (r - l) / 2;
        if (arr[mid] == x)
            return mid;
        if (arr[mid] > x)
            return binary_search(arr, l, mid - 1, x);
        return binary_search(arr, mid + 1, r, x);
    }
    return -1;
}

int main()
```



```
{  
    int n;  
    printf("Enter the size of the integer array: ");  
    scanf("%d", &n);  
    int arr[n];  
    printf("Enter %d integers: ", n);  
    for (int i = 0; i < n; i++)  
    {  
        scanf("%d", &arr[i]);  
    }  
    int pid = fork();  
    if (pid == 0)  
    {  
        char *args[n + 3];  
        args[0] = "./binary_search";  
        for (int i = 0; i < n; i++)  
        {  
            args[i + 1] = (char *)malloc(10 * sizeof(char));  
            sprintf(args[i + 1], "%d", arr[i]);  
        }  
        args[n + 1] = (char *)malloc(10 * sizeof(char));  
        sprintf(args[n + 1], "%d", n);  
        args[n + 2] = NULL;  
        execve(args[0], args, NULL);  
    }  
    else  
    {  
        wait(NULL);  
        printf("Sorting the integer array in parent process...\n");  
        bubble_sort(arr, n);  
        printf("Sorted integer array in parent process:\n");  
        for (int i = 0; i < n; i++)  
        {  
            printf("%d ", arr[i]);  
        }  
        printf("\nPassing the sorted integer array to child  
process...\n");  
        char *args[n + 3];  
        args[0] = "./binary_search";  
        for (int i = 0; i < n; i++)  
        {  
            args[i + 1] = (char *)malloc(10 * sizeof(char));  
            sprintf(args[i + 1], "%d", arr[i]);  
        }  
        args[n + 1] = (char *)malloc(10 * sizeof(char));  
        sprintf(args[n + 1], "%d", n);  
    }  
}
```



```
    args[n + 2] = NULL;  
    execve(args[0], args, NULL);  
}  
return 0;  
}
```

Explanation

In this program, the **bubble_sort** function sorts an array using bubble sort, while the **binary_search** function searches for a particular item in a sorted array using binary search.

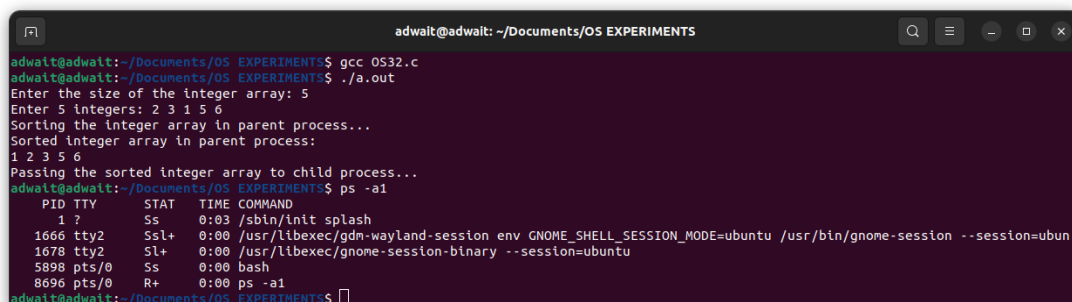
The parent process creates a child process using the **fork** system call, and passes the integer array to the child process through the command line arguments of the **execve** system call.

The child process loads a new program called **binary_search**, which uses the sorted array to perform a binary search to search for a particular item.

To demonstrate the use of the **execve** system call, the parent process calls **execve** twice: once to pass the unsorted integer array to the child process, and once to pass the sorted integer array to the child process. The child process uses the **argc** and **argv** arguments passed to its **main** function to access the sorted integer array.

We don't explicitly handle zombie or orphan states. However, the parent process waits for the child process to finish using the **wait** system call, which ensures that the child process doesn't become a zombie process. If the parent process were to terminate before the child process, the child process would become an orphan process. However, in this case, the child process would terminate shortly after the parent process, so it wouldn't remain an orphan for long.

Output:



```
adwait@adwait: ~/Documents/OS EXPERIMENTS  
adwait@adwait:~/Documents/OS EXPERIMENT$ gcc OS32.c  
adwait@adwait:~/Documents/OS EXPERIMENT$ ./a.out  
Enter the size of the integer array: 5  
Enter 5 integers: 2 3 1 5 6  
Sorting the integer array in parent process...  
Sorted integer array in parent process:  
1 2 3 5 6  
Passing the sorted integer array to child process...  
adwait@adwait:~/Documents/OS EXPERIMENT$ ps -a1  
PID TTY STAT TIME COMMAND  
1 ? Ss 0:03 /sbin/init splash  
1666 tty2 Ssl+ 0:00 /usr/libexec/gdm-wayland-session env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-session --session=ubun  
1678 tty2 Ssl+ 0:00 /usr/libexec/gdm-session-binary --session=ubuntu  
5898 pts/0 Ss 0:00 bash  
8696 pts/0 R+ 0:00 ps -a1  
adwait@adwait:~/Documents/OS EXPERIMENT$
```

Question 3



Write a program that creates exactly 16 copies of itself by calling fork () only twice within a loop. The program should also print a tree of the pids.

Code:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
int main()
{
    for (int i = 0; i < 2; i++)
    {
        fork();
        fork();
        wait(NULL);
    }
    printf("Child:%d , Parent:%d\n", getpid(),getppid());
}
```

Output:

```
adwait@adwait:~/Documents/OS EXPERIMENTS$ gcc OSExp3p3.c
adwait@adwait:~/Documents/OS EXPERIMENTS$ ./a.out
Child:8977 , Parent:8974
Child:8979 , Parent:8976
Child:8980 , Parent:8975
Child:8976 , Parent:8974
Child:8975 , Parent:8973
Child:8974 , Parent:8972
Child:8981 , Parent:8978
Child:8978 , Parent:1601
Child:8983 , Parent:8973
Child:8985 , Parent:8972
Child:8972 , Parent:5898
Child:8986 , Parent:8984
Child:8984 , Parent:1601
adwait@adwait:~/Documents/OS EXPERIMENTS$ Child:8973 , Parent:1601
Child:8987 , Parent:8982
Child:8982 , Parent:1601
```

Question 4

Write a program that calls fork(). Before calling fork(), have the main process access a variable (e.g., x) and set its value to something (e.g., 100). What value



is the variable in the child process? What happens to the variable when both the child and parent change the value of x?

Code:

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

int main()
{
    int x = 100;
    pid_t pid = fork();
    if (pid == 0)
    {
        // This is the child process
        printf("Child: x = %d\n", x); // Prints "Child: x = 100" x = 200; //
Changes the value of x in the child process
        printf("Child: x = %d\n", x); // Prints "Child: x = 200"
    }
    else if (pid > 0)
    {
        // This is the parent process
        printf("Parent: x = %d\n", x); // Prints "Parent: x = 100" x = 300; //
Changes the value of x in the parent process
        printf("Parent: x = %d\n", x); // Prints "Parent: x = 300"
    }
    else
    { // fork() failed
        printf("Error: fork() failed\n");
        return 1;
    }
    return 0;
}
```

Explanation:

When the program runs, the parent process and the child process each have their own separate copies of the **x** variable, which initially has the value 100.

In the child process, **x** is also 100 initially. When the child process changes the value of **x** to 200, it only changes its own copy of **x**. The parent process still has its own copy of **x**, which is unchanged and still has the value of 100.

In the parent process, x is also 100 initially. When the parent process changes the value of x to 300, it only changes its own copy of x. The child process still has its own copy of x, which is unchanged and still has the value of 100.

In general, changes made to a variable in one process do not affect the value of the same variable in another process.

Output:



```
adwait@adwait: ~/Documents/OS EXPERIMENTS
adwait@adwait:~/Documents/OS EXPERIMENTS$ gcc OS4.c
adwait@adwait:~/Documents/OS EXPERIMENTS$ ./a.out
Parent: x = 100
Parent: x = 100
Child: x = 100
Child: x = 100
adwait@adwait:~/Documents/OS EXPERIMENTS$ ps -a1
  PID TTY          STAT       TIME COMMAND
    1 ?           Ss          0:03 /sbin/init splash
  1666 tty2      Ssl+        0:00 /usr/libexec/gdm-wayland-session env GNOME_SHELL_SESSIO
  1678 tty2      Sl+         0:00 /usr/libexec/gnome-session-binary --session=ubuntu
  5898 pts/0     Ss          0:00 bash
  9180 pts/0     R+         0:00 ps -a1
adwait@adwait:~/Documents/OS EXPERIMENTS$ a
```

Question 5

Now write a program that uses `wait()` to wait for the child process to finish in the parent.

Code:

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

int main()
{
    int x = 100;
    pid_t pid = fork();
    if (pid == 0)
    {
        // This is the child process
        printf("Child: x = %d\n", x); // Prints "Child: x = 100" x =
200; // Changes the value of x in the child process
        printf("Child: x = %d\n", x); // Prints "Child: x = 200"
    }
    else if (pid > 0)
    {
        // This is the parent process
```



```
    printf("Parent: x = %d\n", x); // Prints "Parent: x = 100" x =
300; // Changes the value of x in the parent process
    printf("Parent: x = %d\n", x); // Prints "Parent: x = 300"
    int status;
    wait(&status); // Wait for the child process to finish
    printf("Parent: Child process finished with status %d\n",
status);
}
else
{ // fork() failed
    printf("Error: fork() failed\n");
    return 1;
}
return 0;
}
```

Explanation:

When the parent process calls **wait(&status)**, it waits for the child process to finish and then collects its exit status. The **wait()** function returns the process ID of the child that terminated, or -1 if there was an error.

In the child process, if you call **wait()**, it will result in an error because the child process does not have any child processes to wait for.



- **wait()** only works for child processes that were created using **fork()**. If you want to wait for a specific process that was not created by **fork()**, you would need to use a different function, such as **waitpid()**.

Output:

```
adwait@adwait: ~/Documents/OS EXPERIMENTS
adwait@adwait:~/Documents/OS EXPERIMENTS$ gcc 055.c
adwait@adwait:~/Documents/OS EXPERIMENTS$ ./a.out
Child: x = 100
Child: x = 100
Parent: x = 100
Parent: x = 100
Parent: Child process finished with status 0
adwait@adwait:~/Documents/OS EXPERIMENTS$ ps -a1
  PID TTY          STAT       TIME COMMAND
    1 ?           Ss          0:03 /sbin/init splash
  1666 tty2        Ssl+        0:00 /usr/libexec/gdm-wayland-session env GNOME_SHELL_SESSION_MODE=ubuntu /usr/bin/gnome-sess
  1678 tty2        Sl+         0:00 /usr/libexec/gnome-session-binary --session=ubuntu
  5898 pts/0      Ss          0:00 bash
  9433 pts/0      R+          0:00 ps -a1
adwait@adwait:~/Documents/OS EXPERIMENTS$
```

Question 6

Write a program in C to demonstrate zombie and orphan processes in the same program.

Code :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main()
{
    pid_t zombie_pid, orphan_pid;
    zombie_pid = fork();
    if (zombie_pid == 0)
        printf("Child process [%d] running...\n", getpid());
    else if (zombie_pid > 0)
    {
        printf("Parent process [%d] running...\n", getpid());
        sleep(15);
        printf("Parent process exiting...\n");
    }
    else
        printf("System failure");
}
```



```
orphan_pid = fork();
if (orphan_pid == 0)
{
    printf("Child process [%d] running...\n", getpid());
    sleep(15);
    printf("Child process exiting...\n");
}
else if (orphan_pid > 0)
{
    printf("Parent process [%d] running...\n", getpid());
    sleep(10);
}
else
    printf("System failure");
return 0;
}
```

Output:

```
adwait@adwait: ~/Documents/OS EXPERIMENTS
adwait@adwait:~/Documents$ cd OS\ EXPERIMENTS/
adwait@adwait:~/Documents/OS EXPERIMENTS$ gcc ZombieOrphan.c
adwait@adwait:~/Documents/OS EXPERIMENTS$ ./a.out
Parent process [4406] running...
Child process [4407] running...
Parent process [4407] running...
Child process [4408] running...
Parent process exiting...
Child process exiting...
Parent process [4406] running...
Child process [4410] running...
adwait@adwait:~/Documents/OS EXPERIMENTS$ Child process exiting...
ps aux | egrep "Z|defunct"
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
adwait    4421  0.0  0.0  17864  2400 pts/0    S+   22:28   0:00 grep -E --col
or=auto Z|defunct
adwait@adwait:~/Documents/OS EXPERIMENTS$
```

Conclusion:

To summarize my understanding from the experiment, I learned that the `fork()` system call in an OS creates a new child process by duplicating the parent process, allowing it to run independently with its own memory space. Parent



processes can wait for child processes to complete by using `wait()`. I also learned about orphan processes that continue to execute after their parent process has terminated and are adopted by the initial process. Furthermore, I learned about zombie processes that have completed execution but still have an entry in the process table, consuming system resources until their parent process calls `wait()` to read their exit status. Proper management of orphan and zombie processes is crucial for efficient utilization of system resources in an OS.