PROCESS: A **process** is an instance of a program in execution. It is an active entity that contains the program code, its current activity, and resources allocated by the OS.
The memory layout of a process is typically divided into multiple sections.
These sections include:
• Text section— the executable code
• Data section—global variables

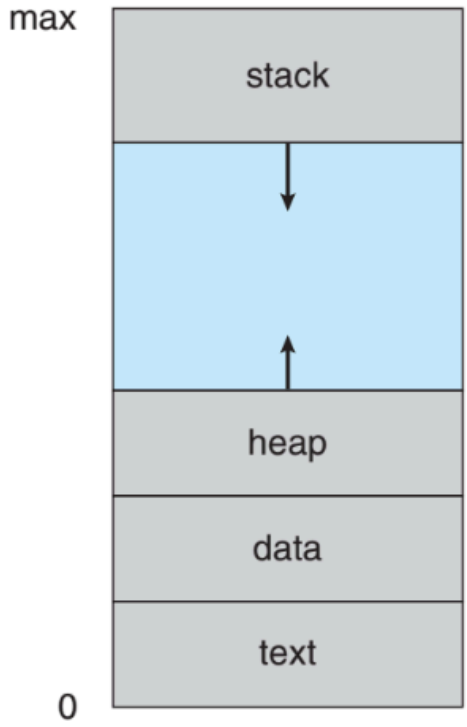


Figure 3.1 Layout of a process in memory.

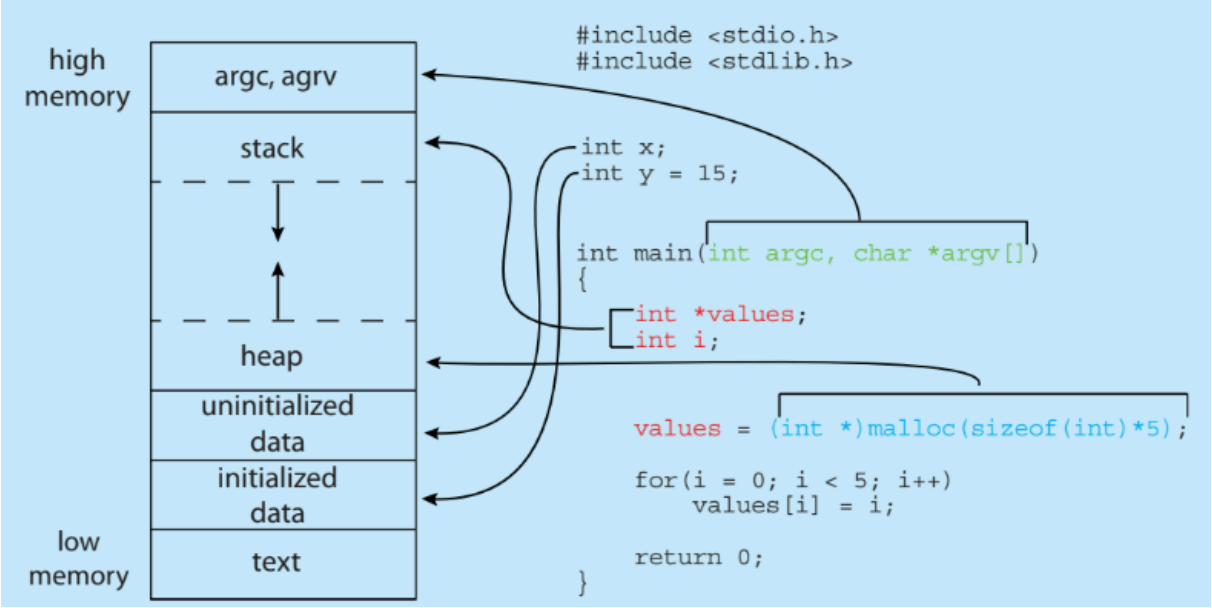
An **activation record** (also called a **stack frame**) is a data structure that stores information about a function call during program execution. It is pushed onto the **stack** when a function is called and popped when the function returns. The activation record helps manage function execution by storing important details needed to resume execution after the function call completes.

Contents of an Activation Record: An activation record typically includes:

1. **Return Address** – The memory location where the program should return after the function execution.
2. **Function Parameters** – The arguments passed to the function.
3. **Local Variables** – Variables that are declared inside the function.
4. **Saved Registers** – CPU registers that need to be restored after the function returns.
5. **Dynamic Link (Optional)** – A pointer to the previous activation record, used in nested function calls.

PROCESS STATE: A **process state** represents the current condition of a process in the operating system. A process changes states as it executes, depending on factors like CPU availability, I/O operations, and scheduling decisions.

MEMORY LAYOUT OF A C PROGRAM:



STATES:

• New. The process is being created.

• Running. Instructions are being executed.

• Waiting. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).

• Ready. The process is waiting to be assigned to a processor.

• Terminated. The process has finished execution.

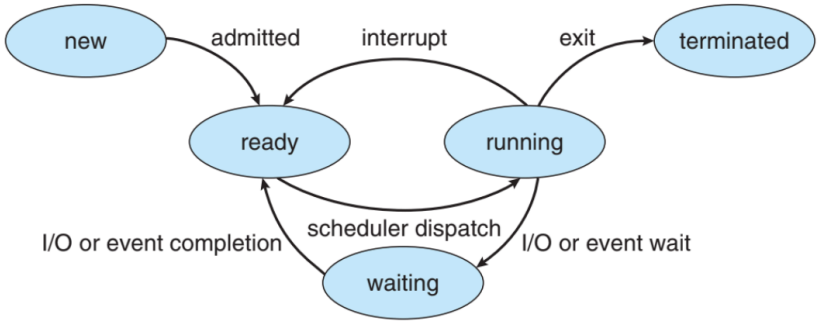


Figure 3.2 Diagram of process state.

PROCESS CONTROL BLOCK:

Each process is represented in the operating system by a process control block (PCB)—also called a task control block.

Process State: The current state of the process (e.g., new, ready, running, waiting, terminated).

Process Number: Unique identification number for the process.

Program Counter: The address of the next instruction to be executed for the process.

CPU Registers: Includes accumulators, index registers, stack pointers, and general-purpose registers.

Memory Limits: Information about the process's allocated memory.

List of Open Files: Files that the process has opened.

CPU-Scheduling Information: Includes process priority, pointers to scheduling queues, etc.

Memory-Management Information: Information such as base and limit registers, page tables, or segment tables.

Accounting Information: Details like the amount of CPU and real-time used, time limits, account numbers, etc.

I/O Status Information: List of I/O devices allocated to the process, a list of open files, etc.

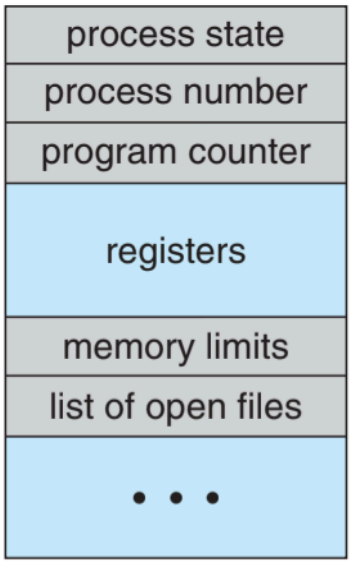


Figure 3.3 Process control block (PCB).

THREAD: A **thread** is the smallest unit of a process that can be scheduled for execution. Essentially, it's a sequence of programmed instructions that the CPU can execute. Threads within the same process share resources such as memory, while each thread maintains its own stack, program counter, and local variables. This allows for parallel execution of tasks within a single process, enhancing efficiency, especially on multicore systems.

PROCESS SCHEDULING AND REPRESENTATION IN LINUX:

1. Objectives of Process Scheduling:

- **Multiprogramming:** Ensures that some process is always running to maximize CPU utilization.
- **Time-Sharing:** Switches the CPU among processes frequently to allow user interaction with running programs.

2. Process Control in Linux:

- Linux represents each process using a **Process Control Block (PCB)**, defined in the task_struct structure in <linux/sched.h>.
- Important fields in task_struct:
    - long state → Stores the process state.
    - struct sched_entity se → Scheduling information.
    - struct task_struct *parent → Pointer to parent process.
    - struct list_head children → List of child processes.
    - struct files_struct *files → Open file list.
    - struct mm_struct *mm → Address space details.
- **Process Management:**
    - Active processes are stored in a **doubly linked list** of task_struct.
    - The kernel maintains a pointer current, pointing to the currently executing process.
    - Example: Changing the state of the running process → current->state = new_state;

3. Scheduling Queues:

- **Ready Queue:** Holds processes waiting for CPU execution.
- **Wait Queue:** Stores processes waiting for an event (e.g., I/O completion).

4. Multiprocessing and Process Types:

- **Single-core systems:** Only one process runs at a time.
- **Multicore systems:** Multiple processes run simultaneously, but if there are more processes than cores, some must wait.

- **Degree of multiprogramming:** The number of processes in memory at a time.
- **Process Behavior:**
    - **I/O-bound processes** → Spend more time on I/O operations.
    - **CPU-bound processes** → Spend more time on computations.

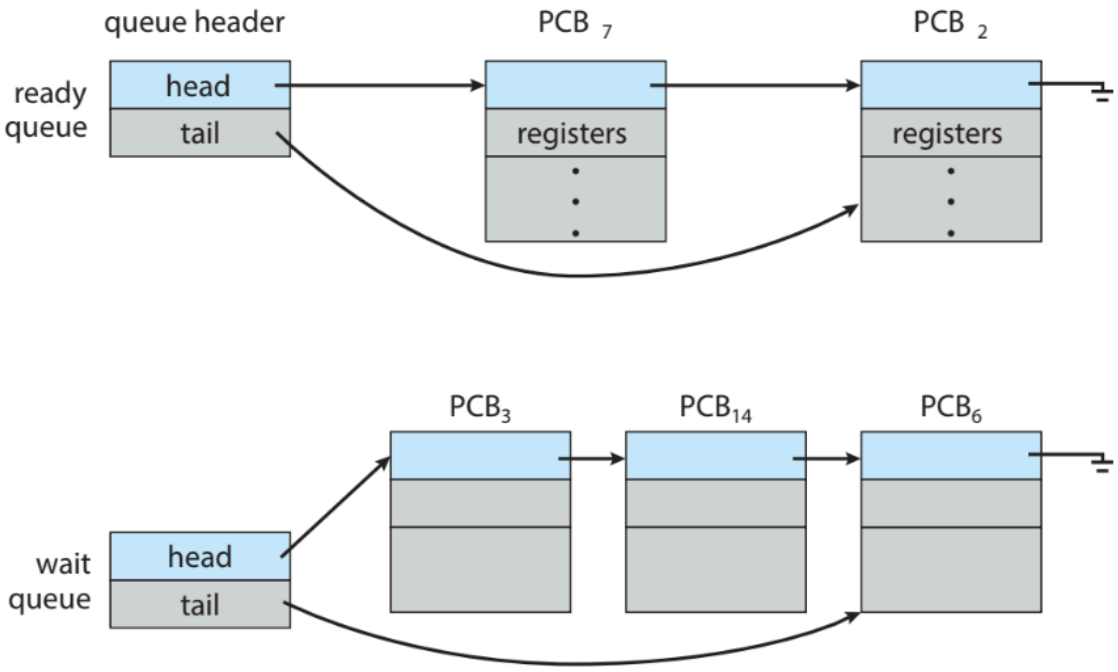


Figure 3.4 The ready queue and wait queues.

SCHEDULING QUEUES:

Ready Queue:

- When processes enter the system, they are placed in the ready queue where they await execution on the CPU's core.
- This queue is typically implemented as a linked list with each process represented by a Process Control Block (PCB).

Wait Queues:

- Besides the ready queue, there are various wait queues for processes waiting for certain events, such as I/O completion.
- Processes move between the ready queue and wait queues based on their state.

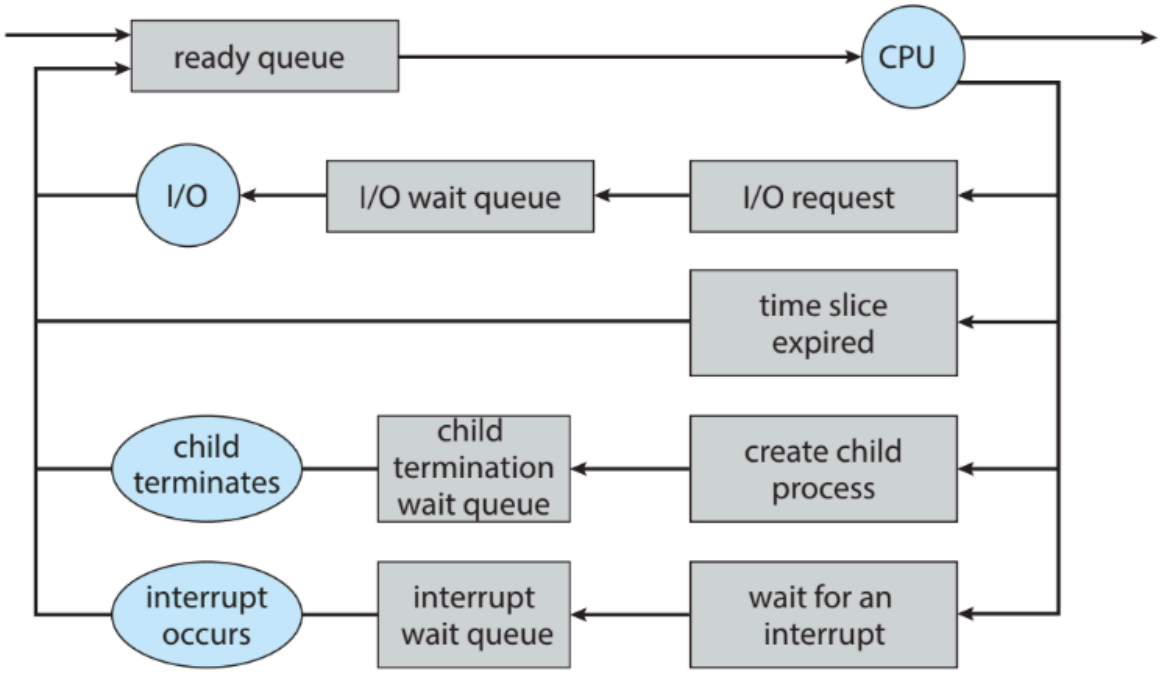


Figure 3.5 Queueing-diagram representation of process scheduling.

• The process could issue an I/O request and then be placed in an I/O wait queue.

• The process could create a new child process and then be placed in a wait queue while it awaits the child's termination.

• The process could be removed forcibly from the core, as a result of an interrupt or having its time slice expire, and be put back in the ready queue.

In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

CPU SCHEDULING:

Process Migration:

- Processes move among the ready queue and various wait queues during their lifetime.

Role of CPU Scheduler:

- The CPU scheduler selects processes from the ready queue and allocates a CPU core to them.
- It must frequently select new processes to ensure efficient CPU utilization.

Process Execution:

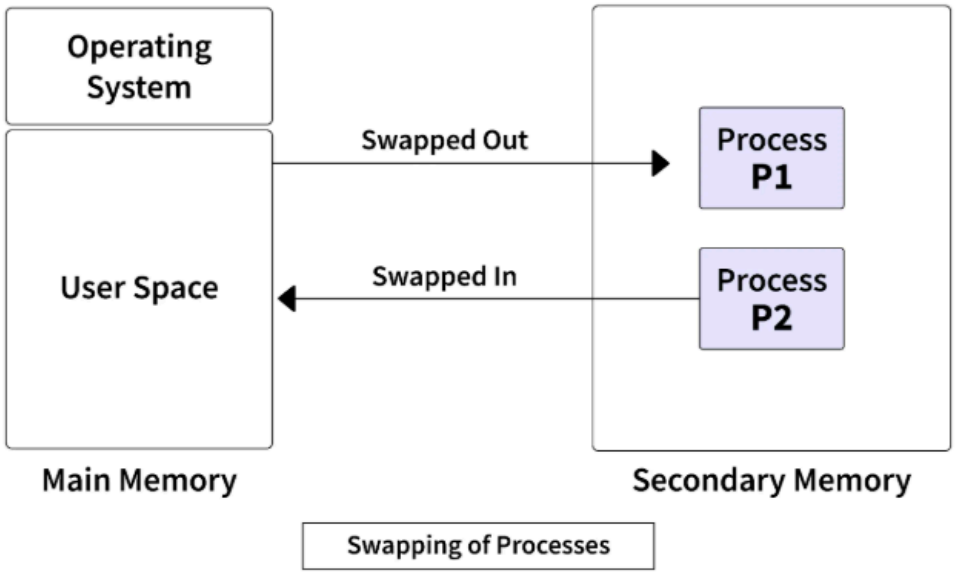
- **I/O-bound Processes:** Execute for a few milliseconds before waiting for an I/O request.
- **CPU-bound Processes:** Require longer CPU time but are not allowed to monopolize the CPU.

Scheduler Frequency:

- The CPU scheduler operates at least once every 100 milliseconds, often more frequently.

Swapping:

- Swapping is an intermediate form of scheduling where processes are temporarily removed from memory to reduce active contention for the CPU.
- Swapped processes are saved to disk and can be reloaded into memory to resume execution.
- Swapping is useful when memory is overcommitted.



CONTEXT SWITCHING:

Interrupts cause the operating system to switch a CPU core from its current task to run a kernel routine. When an interrupt occurs, the system saves the current context of the running process so that it can restore that context later. This essentially suspends the process and then resumes it. The context is represented in the process's Process Control Block (PCB) and includes CPU registers, process state, and memory-management information.

1. **State Save and Restore:**
    - When switching the CPU core to another process, the system performs a state save of the current process and a state restore of the new process. This is known as a **context switch**.
2. **Context Switch Process:**
    - The kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.
    - Context-switch time is pure overhead, as no useful work is done during switching.
3. **Variation in Switching Speed:**
    - The speed of context switching varies from one machine to another.

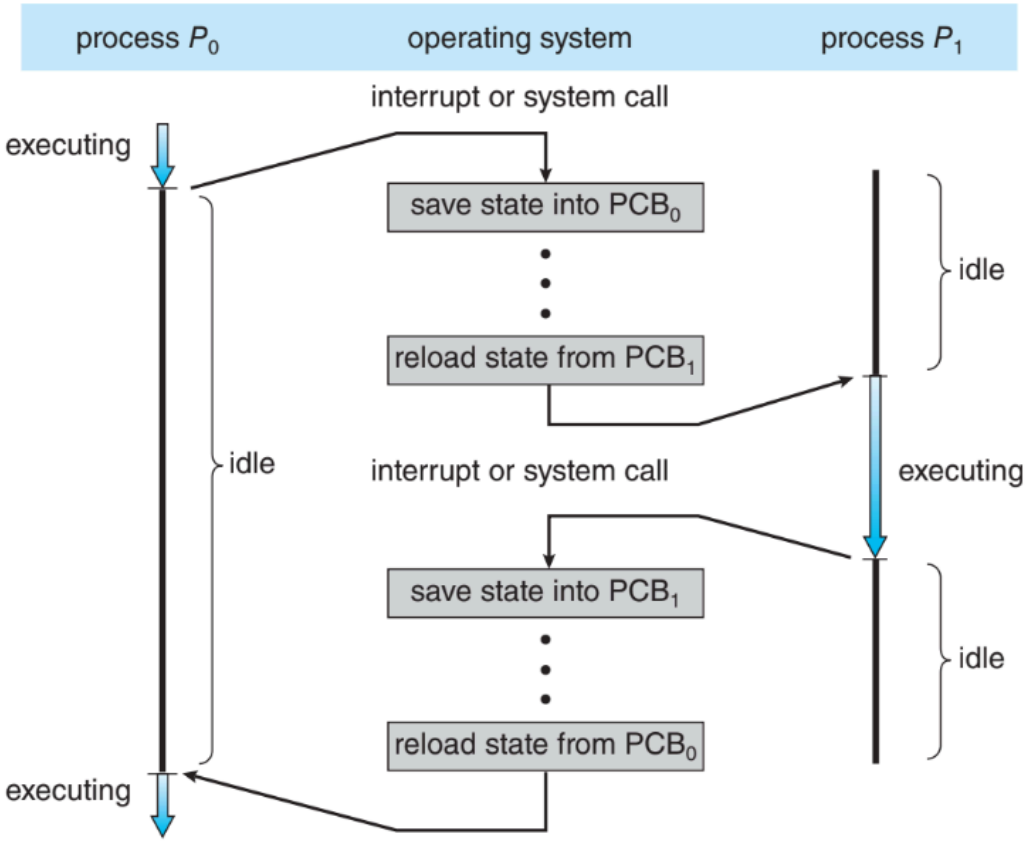


Figure 3.6 Diagram showing context switch from process to process.

PROCESS CREATION:

Parent and Child Processes:

- A process may create several new processes during its execution.
- The creating process is called a parent process, and the new processes are called children.
- Each new process can create other processes, forming a tree structure.

Process Identifiers (pid):

- Operating systems like UNIX, Linux, and Windows assign a unique process identifier (pid) to each process.
- The pid is an integer value used to identify and access various attributes of a process within the kernel.

Process Tree Example:

- In Linux, the systemd process (pid 1) is the root parent process for all user processes and is the first user process created during system boot.
- The systemd process creates other processes that provide various services, like web servers or ssh servers.
- Example: systemd has children like logind and sshd. The logind process manages clients that log onto the system, while sshd manages clients connecting via ssh.

Listing Processes:

- The ps -el command lists complete information for all processes currently active in the system.
- The pstree command displays a tree of all processes in the system.

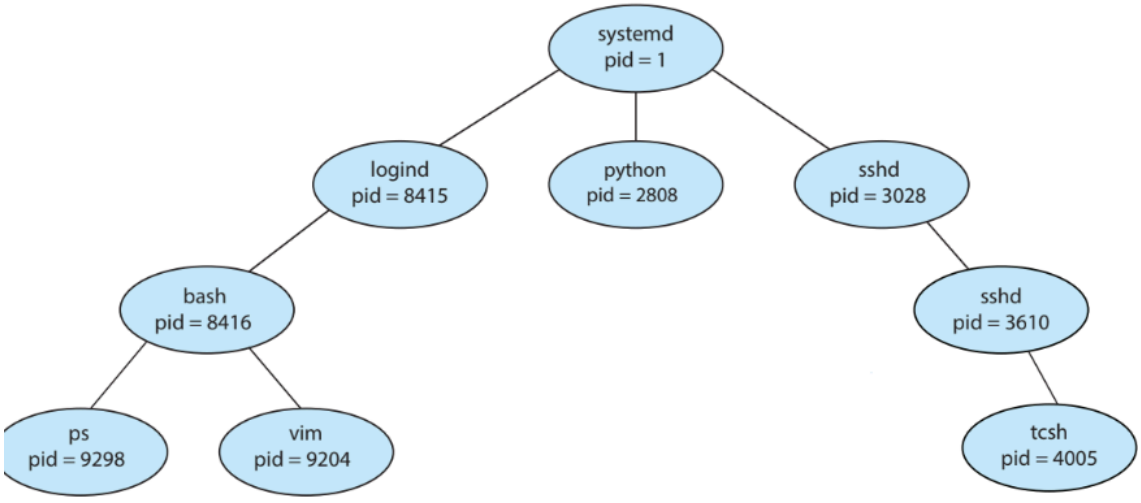


Figure 3.7 A tree of processes on a typical Linux system.

When a process creates a new process, two possibilities for execution exist:

1. The parent continues to execute concurrently with its children.

2. The parent waits until some or all of its children have terminated.

There are also two address-space possibilities for the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).

2. The child process has a new program loaded into it.

A new process is created by the fork() system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the fork(), with one difference: the return code for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

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

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

The C program shown in Figure 3.8 illustrates the UNIX system calls previously described. We now have two different processes running copies of the same program. The only difference is that the value of the variable pid for the child process is zero, while that for the parent is an integer value greater than zero (in fact, it is the actual pid of the child process). The child process inherits privileges and scheduling attributes from the parent, as well certain resources, such as open files. The child process then overlays its address space with the UNIX command /bin/ls (used to get a directory listing) using the execlp() system call (execlp() is a version of the exec() system call). The parent waits for the child process to complete with the wait() system call. When the child process completes (by either implicitly or explicitly invoking exit()), the parent process resumes from the call to wait(), where it completes using the exit() system call.

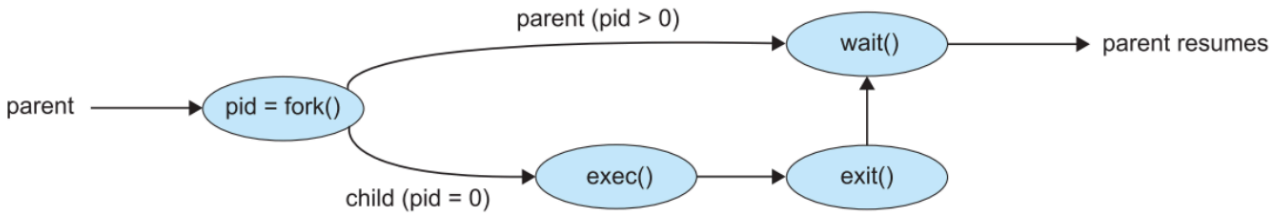


Figure 3.9 Process creation using the `fork()` system call.

PROCESS TERMINATION:

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call.

A parent may terminate the execution of one of its children for a variety of reasons, such as these:

• The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)

• The task assigned to the child is no longer required.

• The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Some systems do not allow a child to exist if its parent has terminated. In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as **cascading termination,** is normally initiated by the operating system.

Zombie Process

- **Definition:** A zombie process is a process that has completed execution but still has an entry in the process table.
- **State:** The process's resources are deallocated, but its process identifier (PID) and exit status remain because the parent process has not yet called the wait() system call.
- **Lifecycle:** All terminated processes briefly become zombies until the parent collects their exit status.
- **Purpose:** The exit status of the zombie process is retained in the process table to allow the parent to read and handle it.

Orphan Process

- **Definition:** An orphan process is a child process whose parent process has terminated.
- **Reassignment:** In traditional UNIX systems, orphan processes are adopted by the init process (the root of the process hierarchy). init takes over the orphaned processes and invokes wait() to collect their exit status.
- **Lifecycle:** Orphan processes exist until they are either terminated or adopted by init (or its modern equivalent, systemd).

Feature	Zombie Process	Orphan Process
Definition	Terminated process with an entry in the process table	Running process whose parent has terminated
System Resource	Does not use memory/CPU, only a PID slot	Uses normal system resources
Parent Process	Parent is alive but did not call **wait()**	Parent is terminated
Adoption	Not adopted by **init**, remains in process table	Adopted by **init** process
Handling Method	Call **wait()** in parent or kill the parent process	Automatically managed by **init**
Risk	Excessive zombies can exhaust PID pool	Generally not problematic

INTERPROCESS COMMUNICATION:

Processes within a system may be independent or cooperating.

Cooperating processes can affect or be affected by other processes, including sharing data.

Reasons for cooperating processes:

1. Information sharing
2. Computation speedup
3. Modularity
4. Convenience

Cooperating processes need interprocess communication (IPC).

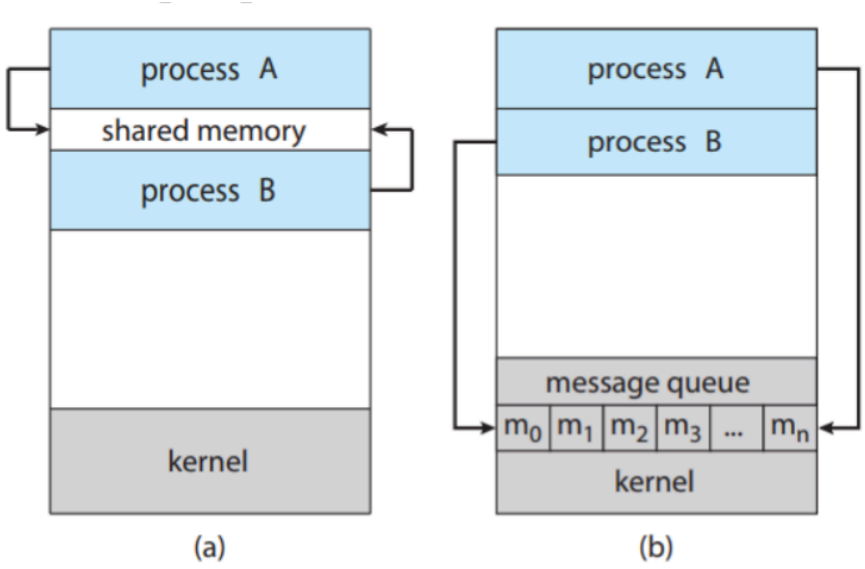


Figure 3.11 Communications models. (a) Shared memory. (b) Message passing.

Shared Memory IPC

• Definition: A form of inter-process communication (IPC) where multiple processes access a common region of memory. Processes can read and write directly to this shared memory area.

• Characteristics:

o Fast: No kernel intervention required for data access.

o Synchronization Required: Processes must use mechanisms like semaphores or mutexes to avoid race conditions.

o Direct Access: Processes share the same memory space, enabling high-speed data exchange.

Message-Passing IPC

• **Definition**: A form of IPC where processes communicate by sending and

receiving messages through a communication channel managed by the

operating system.

• Characteristics:

o Kernel-Mediated: Messages are passed via system calls, ensuring synchronization and data integrity.

o Slower: Involves kernel overhead for message handling.

o Indirect Access: Processes do not share memory; communication is explicit and structured.

Key Difference

• Shared Memory IPC: Processes share a common memory region for direct, high-speed communication, requiring explicit synchronization.

• Message-Passing IPC: Processes communicate through kernel-managed messages, ensuring synchronization but with higher overhead.

Paradigm for cooperating processes:

- producer process produces information that is consumed by a consumer process

Two variations:

unbounded-buffer places no practical limit on the size of the buffer

1. Producer never waits.
2. Consumer waits if there is no buffer to consume.

bounded-buffer assumes that there is a fixed buffer size

1. Producer must wait if all buffers are full.
2. Consumer waits if there is no buffer to consume.

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;

item next_produced;

while (true) {
     /* produce an item in next_produced */

     while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
}
```

Figure 3.12 The producer process using shared memory.

The shared buffer is implemented as a circular array with two logical pointers: in and out. The variable in points to the next free position in the buffer; out points to the first full position in the buffer. The buffer is empty when in == out; the buffer is full when ((in + 1) % BUFFER SIZE) == out.

One issue this illustration does not address concerns the situation in which both the producer process and the consumer process attempt to access the shared buffer concurrently.

```
item next_consumed;

while (true) {
     while (in == out)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;

     /* consume the item in next_consumed */
}
```

Figure 3.13 The consumer process using shared memory.

Message Passing Mechanism: Allows processes to communicate and synchronize without sharing the same address space, particularly useful in distributed environments.

Operations: The basic operations are send(message) and receive(message). Messages can be of fixed or variable size, impacting system complexity and programming ease.

Communication Link Implementation: Can be implemented in various ways, focusing on logical implementation:

- **Direct Communication**: Processes explicitly name the recipient or sender.
- **Indirect Communication**: Messages are sent to and received from mailboxes or ports.

Naming: Processes must refer to each other directly (naming the recipient) or indirectly (using mailboxes).

Synchronization:

- **Blocking** (synchronous) and **Nonblocking** (asynchronous) sends and receives.
- A combination of these can lead to a rendezvous between sender and receiver.

Buffering:

- **Zero Capacity**: No buffering; sender blocks until the recipient receives the message.
- **Bounded Capacity**: Limited queue size; sender blocks if the queue is full.

- **Unbounded Capacity**: Infinite queue length; sender never blocks.

Process Coordination Issue

- A **producer-consumer problem** arises when multiple processes access a shared resource. Blocking send/receive simplifies synchronization, ensuring that the producer waits for the consumer to be ready.

IPC POSIX PRODUCER:

```
gcc test.c -o te
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <unistd.h>

int main() {
    const int SIZE = 4096;          // the size (in bytes) of shared memory object
    const char *name = "OS";        // name of the shared memory object
    const char *message_0 = "Hello"; // strings written to shared memory
    const char *message_1 = "World!";
    int fd;    // shared memory file descriptor
    char *ptr; // pointer to shared memory obect

    fd = shm_open(name, O_CREAT | O_RDWR, 0666); // create the shared memory object
    ftruncate(fd, SIZE); // configure the size of the shared memory object
    // memory map the shared memory object
    ptr = (char *)mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    // write to the shared memory object
    sprintf(ptr, "%s", message_0);
    ptr += strlen(message_0);
    sprintf(ptr, "%s", message_1);
    ptr += strlen(message_1);

    return 0;
}
```

IPC POSIX CONSUMER:

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
int main() {
    const int SIZE = 4096;
    const char *name = "OS";
    int fd;
    char *ptr;
    fd = shm_open(name, O_RDONLY, 0666);
    ptr = (char *)mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    printf("%s", (char *)ptr); // read from the shared memory object
    shm_unlink(name);          // remove the shared memory object
    return 0;
}
```

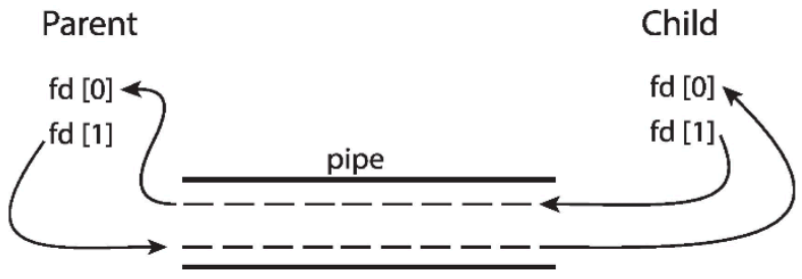
PIPES: Acts as a conduit allowing two processes to communicate

Ordinary pipes – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.

Named pipes – can be accessed without a parent-child relationship.

ORDINARY PIPES:

1. Ordinary Pipes allow communication in standard producer-consumer style.
2. Producer writes to one end (the write-end of the pipe).
3. Consumer reads from the other end (the read-end of the pipe).
4. Ordinary pipes are therefore unidirectional.
5. Require parent-child relationship between communicating processes.
6. Windows calls these anonymous pipes



```c
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1
int main(void) {
    char write_msg[BUFFER_SIZE] = "Greetings";
    char read_msg[BUFFER_SIZE];
    int fd[2];
    pid_t pid;

    if (pipe(fd) == -1) { // create a pipe
        fprintf(stderr, "Pipe failed");
        return 1;
    }
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    if (pid > 0) { /* parent process */
        close(fd[READ_END]);
        write(fd[WRITE_END], write_msg, strlen(write_msg) + 1);
        close(fd[WRITE_END]);
    }
    else { /* child process */
        close(fd[WRITE_END]); // close the unused end of the pipe
        read(fd[READ_END], read_msg, BUFFER_SIZE); // read from the pipe
        printf("read %s", read_msg);
        close(fd[READ_END]); // close the read end of the pipe
    }
    return 0;
}
```

NAMED PIPES:

1. Named Pipes are more powerful than ordinary pipes.
2. Communication is bidirectional.
3. No parent-child relationship is necessary between the communicating processes.
4. Several processes can use the named pipe for communication.

5. Provided on both UNIX and Windows systems.

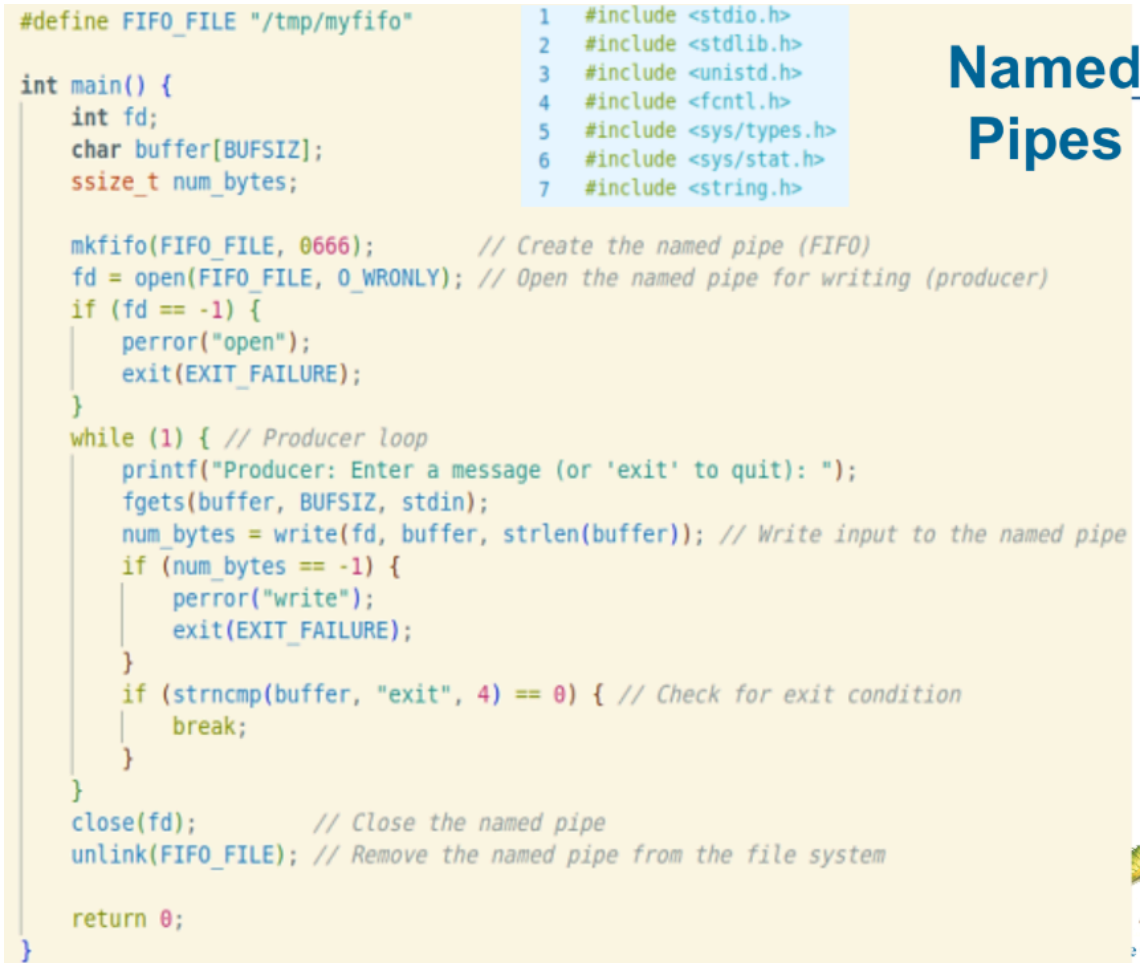
Named pipes FIFOs:

Persistence: They are stored as files in the file system and can be accessed by any process as long as they have the necessary permissions. Unrelated processes can communicate using them.

Bidirectional Communication: Multiple processes can write to and read from the same named pipe. Pipes are unidirectional.

Synchronization: Named pipes provide a natural synchronization mechanism. Processes can block when reading from an empty pipe or writing to a full pipe, allowing them to synchronize their activities effectively.

Multiple Readers/Writers: Named pipes allow multiple processes to read from or write to the pipe simultaneously. This feature enables more flexible communication patterns and can be useful in scenarios where multiple processes need to access the same data concurrently.




```c
#define FIFO_FILE "/tmp/myfifo"

int main() {
    int fd;
    char buffer[BUFSIZ];
    ssize_t num_bytes;

    mkfifo(FIFO_FILE, 0666);        // Create the named pipe (FIFO)
    fd = open(FIFO_FILE, O_WRONLY); // Open the named pipe for writing (producer)
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    while (1) { // Producer loop
        printf("Producer: Enter a message (or 'exit' to quit): ");
        fgets(buffer, BUFSIZ, stdin);
        num_bytes = write(fd, buffer, strlen(buffer)); // Write input to the named pipe
        if (num_bytes == -1) {
            perror("write");
            exit(EXIT_FAILURE);
        }
        if (strncmp(buffer, "exit", 4) == 0) { // Check for exit condition
            break;
        }
    }
    close(fd);          // Close the named pipe
    unlink(FIFO_FILE); // Remove the named pipe from the file system

    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
```

3.8.2 REMOTE PROCEDURE CALL (RPC):

1. **Message-based Communication**: RPC uses well-structured messages to communicate between client and server, unlike the more loosely structured IPC messages.
2. **Ports**: Each message is addressed to an RPC daemon listening to a specific port on the remote system.
3. **Stub Functions**: These are created on both the client and server sides to handle the communication and execution of the remote procedure.
4. **Parameter Marshaling**: This process converts machine-dependent data into a machine-independent format to ensure proper communication between different systems.
5. **Call Semantics**: RPCs can fail or be duplicated due to network errors. Handling these failures requires mechanisms like timestamps and acknowledgment messages.
6. **Binding**: The client must know the port number on the server. Binding can be predetermined or done dynamically through a rendezvous mechanism.
7. **Implementation**: RPCs are useful in implementing distributed file systems and other remote services by sending messages to the appropriate daemon and executing file-related system calls.

Q. Including the initial parent process, how many processes are created by the program shown in Figure 3.32?

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

int main()
{
    int i;

    for (i = 0; i < 4; i++)
       fork();

    return 0;
}
```

The fork() function is called within a loop that runs 4 times. Each call to fork() creates a new process. The number of processes doubles with each call to fork() since every existing process (both parent and child) creates a new process.

- Initially, there's 1 process.
- After the 1st fork() call: 2 processes (1 parent + 1 child).
- After the 2nd fork() call: 4 processes (each of the 2 processes forks, creating 2 new processes each, so 2 * 2 = 4).
- After the 3rd fork() call: 8 processes (each of the 4 processes forks, creating 2 new processes each, so 4 * 2 = 8).
- After the 4th fork() call: 16 processes (each of the 8 processes forks, creating 2 new processes each, so 8 * 2 = 16).

In total, there are 16 processes created.

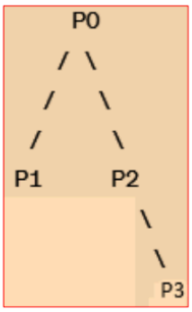
Suppose a process executes **fork () system call 2 times**. Draw a labelled diagram with meaningful hints to reflect your understanding of all the resultant processes along with their linkage. Explain which processes should issue wait () system for correct behavior. No grading without appropriate hints to show your understanding. [1.5 + 1]

Process hierarchy:

- 1st fork () creates P2 and controls return to 2nd fork () in P0, execution of which creates P1.
- The control returns in P2 to 2nd fork (), execution of which creates P3.

Wait():

- Process P0 should issue the wait () system call to wait for the termination of each of its child processes (P1 and P2).
- Process P2 should issue the wait () system call to wait for the termination of its child process P3.



b) Given the following C program. When and how will "LINE J" printed? Give technical explanation.

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

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
       printf("LINE J");
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

```
k200281kainat@k200281kainat-VirtualBox:~$ gedit code.c
k200281kainat@k200281kainat-VirtualBox:~$ gcc code.c -o out
k200281kainat@k200281kainat-VirtualBox:~$ ./out
a.out      code.c     Downloads         Music  palindrome.cpp  Templates
calc.cpp   Desktop    examples.desktop  OSlab  Pictures        Videos
check.c    Documents  Makefile.save     out    Public
Child completek200281kainat@k200281kainat-VirtualBox:~$
```

In the given C program, the child process executes the execlp() function:

```
execlp("/bin/ls", "ls", NULL);

printf("LINE J");
```

Understanding execlp() Behavior:

- The execlp() function replaces the current process image with a new process (the ls command).
- This means **all code after execlp() will not execute if execlp() succeeds**.
- "LINE J" will only print if execlp() **fails** (e.g., if /bin/ls does not exist or cannot be executed).

How Many Times Will "LINE J" Be Printed?

- Under normal circumstances (if execlp() succeeds), "LINE J" will **never** be printed.
- If execlp() fails, "LINE J" **will be printed once** (since only the child process executes this line).

Q) What will be the output at Line X and Y?

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

#define SIZE 5

int nums[SIZE] = {0,1,2,3,4};

int main()
{
int i;
pid_t pid;

   pid = fork();

   if (pid == 0) {
      for (i = 0; i < SIZE; i++) {
         nums[i] *= -i;
         printf("CHILD: %d ",nums[i]); /* LINE X */
      }
   }
   else if (pid > 0) {
      wait(NULL);
      for (i = 0; i < SIZE; i++)
         printf("PARENT: %d ",nums[i]); /* LINE Y */
   }

   return 0;
}
```

Figure 3.4 What output will be at Line X and Line Y?

1. **Initialization:**
   - A global array nums[SIZE] = {0,1,2,3,4} is defined.
   - A fork is created, meaning we now have **two processes (parent and child)**.
2. **Child Process Execution (pid == 0):**

The child process modifies the nums array, multiplying each element by -i:
nums[i] *= -i;

   - Since nums is a global array, each modification in the child process affects only its **own copy** of the array (due to process memory separation after fork()).
   - It prints the modified values at **Line X**.
3. **Parent Process Execution (pid > 0):**
   - The parent process **waits** for the child process to finish (wait(NULL)).
   - It then prints the nums array at **Line Y**.

Child Process Execution (Line X):

Each iteration modifies nums[i] as follows:

i	nums[i] (before)	nums[i] *= -i (modified)	Printed Value
0	0	0 * -0 = 0	CHILD: 0
1	1	1 * -1 = -1	CHILD: -1
2	2	2 * -2 = -4	CHILD: -4
3	3	3 * -3 = -9	CHILD: -9
4	4	4 * -4 = -16	CHILD: -16

Output at Line X (child process):

CHILD: 0 CHILD: -1 CHILD: -4 CHILD: -9 CHILD: -16

Parent Process Execution (Line Y):

- The parent **does not** modify the nums array.
- Since the child process modifies its **own copy**, the parent's nums remains unchanged.
- The parent prints the original nums array.

Output at Line Y (parent process):

PARENT: 0 PARENT: 1 PARENT: 2 PARENT: 3 PARENT: 4

Final Output of the Program:

CHILD: 0 CHILD: -1 CHILD: -4 CHILD: -9 CHILD: -16

PARENT: 0 PARENT: 1 PARENT: 2 PARENT: 3 PARENT: 4

Key Observations:

1. **Child and Parent have separate memory spaces** due to fork(), so modifications in one do not affect the other.
2. **Parent waits (wait(NULL))**, ensuring the child process runs first.
3. **Global variables are duplicated** for each process after fork().

Q) Using the program in Figure 3.34, identify the values of pid at lines A, B, C, and D. (Assume that the actual pids of the parent and child are 2600 and 2603, respectively.)

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

int main()
{
pid_t pid, pid1;

   /* fork a child process */
   pid = fork();

   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
   }
   else if (pid == 0) { /* child process */
      pid1 = getpid();
      printf("child: pid = %d",pid); /* A */
      printf("child: pid1 = %d",pid1); /* B */
   }
   else { /* parent process */
      pid1 = getpid();
      printf("parent: pid = %d",pid); /* C */
      printf("parent: pid1 = %d",pid1); /* D */
      wait(NULL);
   }

   return 0;
}
```

Figure 3.23 What are the pid values?

Child Process:

When pid == 0, it indicates the child process:

- **At Line A:** printf("child: pid = %d", pid);
  - Since pid is 0 for the child, the output will be:
    child: pid = 0
- **At Line B:** printf("child: pid1 = %d", pid1);
  - pid1 = getpid(); fetches the child process's own PID, which is **2603**.
    The output will be:
    child: pid1 = 2603

Parent Process:

When pid > 0, it indicates the parent process:

- **At Line C:** printf("parent: pid = %d", pid);
  - pid here holds the child's PID, which is **2603**.
    The output will be:
    parent: pid = 2603
- **At Line D:** printf("parent: pid1 = %d", pid1);
  - pid1 = getpid(); fetches the parent's own PID, which is **2600**.
    The output will be:
    parent: pid1 = 2600

Final Output:

child: pid = 0
child: pid1 = 2603
parent: pid = 2603
parent: pid1 = 2600

Q) How many processes are created?

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

int main()
{
    /* fork a child process */
    fork();

    /* fork another child process *
    fork();

    /* and fork another */
    fork();

    return 0;
}
```

Figure 3.31 How many processes are created?

1. **Initial Process:** Before any fork() calls, there is **1** process (the parent).
2. **First fork():**
   - The parent creates 1 child process.
   - Total processes = **2** (1 parent + 1 child).
3. **Second fork():**
   - Each of the 2 existing processes (parent and child) executes fork(), creating 2 more processes.

- Total processes = **4** (2 existing × 2).

4. **Third fork():**
   - Each of the 4 existing processes executes fork(), creating 4 more processes.
   - Total processes = **8** (4 existing × 2).

The total number of processes created is **8**, including the original parent process.

Q) What will be the output at Line A?

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

int value = 5;

int main()
{
pid_t pid;

   pid = fork();

   if (pid == 0) { /* child process */
      value += 15;
      return 0;
   }
   else if (pid > 0) { /* parent process */
      wait(NULL);
      printf("PARENT: value = %d",value); /* LINE A */
      return 0;
   }
}
```

Figure 3.30 What output will be at Line A?

Child Process (pid == 0):

- value += 15; updates value to 20 (5 + 15).
- The child process then terminates with return 0;.

Parent Process (pid > 0):

- wait(NULL); causes the parent to wait until the child process finishes.
- Since value is a global variable but not shared between parent and child (as fork() creates a separate copy of the address space), any changes made by the child process do not affect the parent's value.
- Thus, value remains 5 in the parent process.

Output at Line A:
PARENT: value = 5

Q) When a process creates a new process using the fork() operation, which of the following states is shared between the parent process and the child process?
a. Stack b. Heap c. Shared memory segments

When a process creates a new process using the fork() operation, the following happens regarding the sharing of states between the **parent process** and the **child process**:

- **Stack: Not shared.** Each process gets its own copy of the stack. Changes in the stack of one process do not affect the other. The stack contains function calls, local variables, and execution history, which remain independent for both processes.
- **Heap: Not shared.** The child process receives a **copy** of the parent's heap, but it is a **separate copy**. Any modifications to the heap memory by one process do not affect the other. However, the **Copy-on-Write (COW)** technique may be used to optimize memory usage until a modification occurs.
- **Shared Memory Segments: Shared.** If the parent process has **shared memory segments** created using mechanisms like **shmget()** or **mmap**, these segments are **shared** between the parent and the child process. Both processes can **read and write** to the same **shared memory**, enabling **inter-process communication (IPC)**.

Option (c) Shared memory segments is the correct choice.

Q) How many processes will be created?

```
v1 = fork();
v2 = fork();
v3 = fork();

if (v3 == 0) {
    fork();
}
```

Total: 12 processes

Q) Write a code with the process hierarchy Po being the child of P1 and P2 being the child of P3 with each process printing 5 numbers from the array arr = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15].

```c
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
#include<sys/wait.h>
int main()
{
int arr[] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
pid_t pid1;
pid1 = fork();
if(pid1==0)
{
for(int i=0;i<5;i++)
{
printf("%d",arr[i]);
}
pid1 = fork();
if(pid1==0)
{
for(int i=5;i<10;i++)
{
printf("%d",arr[i]);
}
}
else
{
wait(NULL);
}
}
else
{
wait(NULL);
wait(NULL);
for(int i=10;i<15;i++)
{
printf("%d",arr[i]);
}
}
return 0;
}
```

```
k200281kainat@k200281kainat-VirtualBox:~$ gedit midq.c
k200281kainat@k200281kainat-VirtualBox:~$ gcc -o out midq.c
k200281kainat@k200281kainat-VirtualBox:~$ ./out
k200281kainat@k200281kainat-VirtualBox:~$ ./out
123456789101234511121314 15k200281kainat@k200281kainat-VirtualBox:~$
```

```c
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
#include<sys/wait.h>
int main()
{
int arr[] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
pid_t pid1,pid2;
pid1 = fork();
if(pid1==0)
{
for(int i=5;i<10;i++)
{
printf("%d",arr[i]);
}
pid2 = fork();
if(pid2==0)
{
for(int i=10;i<15;i++)
{
printf("%d",arr[i]);
}
return 0;
}
else
{
return 0;

}
}
else
{
for(int i=0;i<5;i++)
{
printf("%d",arr[i]);
}
}
return 0;
}
```

```
k200281kainat@k200281kainat-VirtualBox:~$ gcc midq.c -o out
k200281kainat@k200281kainat-VirtualBox:~$ ./out
123456789101112131415k200281kainat@k200281kainat-VirtualBox:
```

```c
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
#include<sys/wait.h>
int main()
{
int arr[] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
pid_t pid1,pid2;
pid1 = fork();
if(pid1==0)
{
for(int i=0;i<5;i++)
{
printf("%d",arr[i]);
}
pid2 = fork();
if(pid2==0)
{
for(int i=5;i<10;i++)
{
printf("%d",arr[i]);
}
return 0;
}
else
{
return 0;
}
}
else
{
wait(NULL);
wait(NULL);
for(int i=10;i<15;i++)
{
printf("%d",arr[i]);
}
}
return 0;
}
```

```
k200281kainat@k200281kainat-VirtualBox:~$ gedit midq.c
k200281kainat@k200281kainat-VirtualBox:~$ gcc -o out midq.c
k200281kainat@k200281kainat-VirtualBox:~$ ./out
123451234567891011121314 15k200281kainat@k200281kainat-VirtualBox:~$
```

OSLAB4 SYSTEM CALLS

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
int main()
{
pid_t p;
printf("before fork\n");
p=fork();
if(p==0)
{
printf("I am child having id %d\n",getpid());
printf("My parent's id is %d\n",getppid());
}
else{
printf("My child's id is %d\n",p);
printf("I am parent having id %d\n",getpid());
}
printf("Common\n");
}
```

Output:

```
baljit@baljit:~/Programs$ gcc fork.c
baljit@baljit:~/Programs$ ./a.out
before fork
My child's id is 28
I am child having id 28
I am parent having id 27
My parent's id is 27
common
common
```

Working of fork() system call:

After compiling the program with gcc it creates an output file "a.out". The moment you run a.out using the command, ./a.out, a new process is created (parent). This process gets the process id (PID) 27. The PID will differ from system to system and each time you run the program. The process starts to run and it prints before fork. Next it executes the fork() system call. If it gets executed a child process is created having a different PID. Now there are two process in the system both having the same code to run. But since the code has been run till this line the

execution will continue from the next line in both the process. fork() on success returns either 0 or a non-zero value. Since, the same code is both the processes the variable ‘p’ will have some value in both the process. In the parent process it gets a non-zero positive value (which actually is the PID of the child). In the child process ‘p’ gets the value ‘0’.
Hence the next lines of code has been written to check the value of ‘p’. When the if-else code runs from within the parent the condition becomes false and the else part is run. So, the lines

My child’s id is 28
I am parent having id 27
Are printed by the parent process.

When the if-else case runs from within the child the if condition becomes true and hence the lines

I am child having id 28
My parent’s id is 27
Are printed by the child process.

Since the printf(“common\n”) line was out of the if-else it has been printed twice; once by the parent process and once by the child process.

By analysing the PID printed by each process the parent-child relationship can also be verified between them. The process with PID 27 says its child has a PID 28. Similarly, the process with PID 28 says its parent has a PID 27.

Note: The function getpid() is used to print the PID of a process while the function getppid() is used to print the PID of the parent process.

WAIT SYSTEM CALL:

wait() system call takes only one parameter which stores the status information of the process. Pass NULL as the value if you do not want to know the exit status of the child process and are simply concerned with making the parent wait for the child. On success, wait returns the PID of the terminated child process while on failure it returns -1.

Important: wait() can be used to make the parent wait for the child to terminate(finish) but not the other way around.

```
#include<unistd.h>
#include<sys/types.h>
#include<stdio.h>
#include<sys/wait.h>
int main()
{
pid_t p;
printf("before fork\n");
p=fork();
if(p==0)//child
{
printf("I am child having id %d\n",getpid());
printf("My parent's id is %d\n",getppid());
}
else//parent
{
wait(NULL);
printf("My child's id is %d\n",p);
printf("I am parent having id %d\n",getpid());
}
printf("Common\n");
}
```

Output

```
baljit@baljit:~/cse325/Process$ ./a.out
before fork
I am child having id 458
My parent's id is 457
Common
My child's id is 458
I am parent having id 457
Common
```

child

Parent

How it works?

The execution begins by printing "before fork". Then fork() system call creates a child process. wait() system call is added to the parent section of the code. Hence, the moment processor starts processing the parent, the parent process is suspended because the very first statement is wait(NULL). Thus, first, the child process runs, and the output lines are all corresponding to the child process. Once the child process finishes, parent resumes and prints all its printf()

statements. The NULL inside the wait() means that we are not interested to know the status of change of state of child process.

Program to replace process image using execl( )

There is a family of function which can be used for replacing the current process with a new process. They differ in the number of arguments and the way they start a new process. The various functions are *execl, execlp, execle, execv* and *execvp.* In this post we will write a Program to replace process image using execl( )

We will discuss the use of *execl* and the others can be used on similar lines to replace process image. The syntax for execl is

*int execl(const char *path, const char *arg0, ..., (char *)0);*

The 1st argument is the PATH for the program *while the last argument will be NULL.*
Program: Program to replace process image using execl( ). Process *ps* will replace the image of current process

```
#include<stdio.h>
#include<unistd.h>
int main()
{
     printf("Before execl\n");
     execl("/bin/ps","ps","-a",NULL);//
     printf("After execlp\n");
}
```

Output

```
Before execl
  PID TTY          TIME CMD
  122 tty1     00:00:00 ps
```

How it Works?

The program prints the first message and then calls execl. execl searches for the program ps in the PATH variable and executes it replacing the original program. Hence the second message doesn't get printed. Also, It can be seen in the output that the name of the process running is ps and not the user process exec.

Normally the exec() of function is to replace the image of a child process. By default the child process duplicates the parent which means that the child will have exactly the same code as the parent. Now, if you want a different code in child then use execl() function.

write()/read() system call :

read() and write() system calls are used to read and write data respectively to a file descriptor. To understand the concept of write()/read() system calls let us first start with write() system call. **write()** system call is used to write to a file descriptor. In other words write() can be used to write to any file (all hardware are also referred as file in Linux) in the system but rather than specifying the file name, you need to specify its file descriptor.

Syntax:

```
#include<unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

The first parameter (fd) is the file descriptor where you want to write. The data that is to be written is specified in the second parameter. Finally, the third parameter is the total bytes that are to be written.

To understand better lets look at the first program below:

Program1: To write some data on the standard output device (by default – monitor)

//Name the program file as "write.c"

```
#include<unistd.h>
int main()
{
write(1,"hello\n",6); //1 is the file descriptor, "hello\n" is the
data, 6 is the count of characters in data
}
```

How it works?

The write() system call takes three parameters: "1" which is the file descriptor of the file where we want to write. Since we want to write on standard output device which is the screen, hence the file descriptor, in this case, is '1', which is fixed (0 is the file descriptor for standard input device (e.g. keyboard) and 2 is for standard error device)).

Next thing is what we want to write on the screen. In this case its "hello\n" i.e. hello and newline(\n), so a total of 6 characters, which becomes the third parameter. The third parameter is how much you want to write, which may be less than the data specified in the second parameter. You can play around and see the change in output.

Output:

Once you compile and run this, the output on the screen will be the word "hello", as shown below

```
baljit@baljit:~$ gcc write.c
baljit@baljit:~$ ./a.out
hello
```

On success, the write() system call returns the 'number of bytes written' i.e. the count of how many bytes it could write. This you can save in an integer variable and checked. The write() system call on failure returns -1.
Note: students get confused by thinking that write() return the data that is written. Remember, it returns the count of characters written. Refer to the program below.

Program 2

```
#include<stdio.h>
#include<unistd.h>
int main()
{
int count;
count=write(1,"hello\n",6);
printf("Total bytes written: %d\n",count);
}
```

How it Works?

The program is similar to the previous one except that this time it also explicitly prints the count of bytes that write() system call was able to write on file descriptor 1.

Output:

```
baljit@baljit:~$ gcc write.c
baljit@baljit:~$ ./a.out
hello
Total bytes written: 6
```

Variations

Try making the changes as shown in the codes below and observe the output to understand the working of the write() system call in detail.

Program 3:

```
#include<unistd.h>
int main()
{
write(1,"hello\n",60); //the bytes to be printed (third parameter)
are more than the data specified in 2nd parameter
}
```

Output: hello (it was printing special chars garbage values)

Program4:

```
#include<unistd.h>
int main()
{
write(1,"hello\n",3);//the bytes to be printed (third parameter) are
less than the data specified in 2nd parameter
}
```

Output: hel

Program5:

```
#include<unistd.h>
#include<stdio.h>
int main()
{
int count;
count=write(3,"hello\n",6); //the file descriptor is not one of the
pre-specified ones i.e., 0, 1 or 2
printf("Total bytes written: %d\n",count);
}
```

Will this call to write() be successful? If not then what will it return?

read()

The use of *read()* system call is to read from a file descriptor. The working is same as write(), the only difference is read() will read the data from file pointed to by file descriptor.

Syntax:

```
#include<unistd.h>
ssize_t read(int fd, const void *buf, size_t count);
```

The first parameter is the file descriptor. The second parameter is the buffer where the read data will be saved. Lastly, the third parameter is the number of bytes that you want to read.
Think of the buffer as a temporary storing area. As you are reading from this page and before typing the program on your system you temporarily store it in your brain. So your brain is the buffer. Although this page contains a lot of data, you might want to read only 20 characters. Hence, the third parameter (count) tells, how much you want to read?

Program 6: To read data from the standard input device and write it on the screen

```
//read.c
#include<unistd.h>
int main()
{
char buff[20];
```

```
read(0,buff,10);//read 10 bytes from standard input device(keyboard),
store in buffer (buff)
write(1,buff,10);//print 10 bytes from the buffer on the screen
}
```

How it works?

The read() system call reads the input typed by the user via the keyboard (file descriptor 0) and stores it in the buffer (buff) which is nothing but a character array. It will read a maximum of 10 bytes (because of the third parameter). This can be less than or equal to the buffer size. No matter how much the user types only first 10 characters will be read.
Finally, the data is printed on the screen using the write() system call. It prints the same 10 bytes from the buffer (buff) on the screen (file descriptor 1).

Output:

```
baljit@baljit:~$ ./a.out
1234567890
1234567890baljit@baljit:~$
```

Variations: you can try different values of the third parameter in read and write to understand the working better. The read() system call returns -1 on failure and "the count of bytes read" on success.

Important:

A common question that comes to our mind is that as a programmer you can not guarantee how much the user will type as an input. Hence, you can not specify the correct bytes in write() systemc all's 3rd parameter. Hence, the output may vary from what you expect.
Now, remember what read() returns on success! the number of bytes read, and that's the key as demonstrated below.

Program 7: To read data from the standard input device and write it on the screen

//read.c

```
#include<unistd.h>
int main()
{
int nread;
char buff[20];
nread=read(0,buff,10);//read 10 bytes from standard input
device(keyboard), store in buffer (buff)
write(1,buff,nread);//print 10 bytes from the buffer on the screen
}
```

How it works? This time we store the count of bytes read by read() in read variable and then use variable in the write() to print exactly the same number of bytes on the screen.

Program on open() system call In the previous section on read/write system call we learned how to read from the standard input device and how to write to a standard output device. But, normally we would either read from a user-created file or write to a user-created file. Hence, the question comes that how do know the file descriptor of these files because to read or to write the first parameter in the read()/write() system calls is the file descriptor. We can use the *open()* system call to get the file descriptor of any file.

Syntax

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

The open() system call has two syntax. We discuss the first one here:

```
int open(const char *pathname, int flags);
```

The first parameter is the name of the file that you want to open for reading/writing. The second parameter is the mode in which to open the file i.e., for reading or for writing. For reading from a file, the flag used is O_RDONLY, for writing O_WRONLY and for both reading and writing O_RDWR.

Other commonly used flags are O_CREAT, O_APPEND, O_TRUNC, O_EXCL

On success, the open() system call return the file descriptor of the file. This descriptor now can be used in read()/write() system call for further processing. The value of the file descriptor will always be a positive number greater than 2. Whereas on failure it returns -1.

Program 1: Write a program using open() system call to read the first 10 characters of an existing file “test.txt” and print them on screen. `//open.c`

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
int main()
{
int n,fd;
char buff[50];
fd=open("test.txt",O_RDONLY); //opens test.txt in read mode and the
file descriptor is saved in integer fd.
printf("The file descriptor of the file is: %d\n,fd); // the value of
the file descriptor is printed.
n=read(fd,buff,10);//read 10 characters from the file pointed to by
file descriptor fd and save them in buffer (buff)
```

```
write(1,buff,n); //write on the screen from the buffer
}
```

How it works?

First, create a file “test.txt” and write some content into it(more than 10 characters). The open() system call opens the file test.txt in read-only mode and returns the file descriptor. This file descriptor is saved in variable ‘fd’. You can print it to check the value of file dexcriptor of the file. next, use read() to read 10 characters from the file into the buffer. Finally, the buffer values are printed on the screen. The file descriptor used here is ‘0’ which is the file descriptor for standard output device.

Output

Step1: create the file test.txt and write “1234567890abcdefghij54321” into it
$nano test.txt
Step2: compile the program
$gcc open.c
Step3: run
$./a.out

Syntax 2:

The second syntax is used when the file involved does not already exist in the system and you want to create it on the go.

```
int open(const char *pathname, int flags, mode_t mode);
```

The first parameter is the file name. The second parameter is the mode (read/write). In this case apart from writing O_RDONLY, O_WRONLY and O_RDWR you also need to write O_CREAT, to create the file. The third parameter secifies the permissions on the created file (read/write/execute).

Note: O_RDONLY and O_WRONLY flags are different from read and write permission in the manner that even if a file has write permision on it it will not open in write mode until you specify the O_WRONLY mode.

Now, we extend Program 1 to read from file text.txt and write the contents into a npon existing file “towrite.txt”

Program2: To read 10 characters from file “test.txt” and write them into non-existing file “towrite.txt”

```
//open2.c
#include<unistd.h>
#include<sys/types.h>
```

```
#include<sys/stat.h>
#include<fcntl.h>
int main()
{
int n,fd,fd1;
char buff[50];
fd=open("test.txt",O_RDONLY);
n=read(fd,buff,10);
fd1=open("towrite.txt",O_WRONLY|O_CREAT,0642);//use the pipe symbol
(|) to separate O_WRONLY and O_CREAT
write(fd1,buff,n);
}
```

How it works?

In this we open the "towrite.txt" in write mode and also use O_CREAT to create it with read and write permission for user, read for the group and write for others. The data is read from test.txt using file descriptor fd and stored in buff. It is then written from buff array into file towrite.txt using file descriptor fd1.

Output

Process Management

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

int main() {
    pid_t child_pid;
    int status;

    // Fork a child process
    child_pid = fork();

    if (child_pid < 0) {
        perror("fork failed");
        exit(EXIT_FAILURE);
    } else if (child_pid == 0) {
        // This is the child process
        printf("Child process: PID = %d\n", getpid());
        printf("Child process: PPID = %d\n", getppid());
        printf("Child process: Executing ls command...\n");
        execlp("ls", "ls", "-l", NULL);
        // execlp() will only return if there's an error
        perror("execlp failed");
        exit(EXIT_FAILURE);
    } else {
        // This is the parent process
        printf("Parent process: PID = %d\n", getpid());
        printf("Parent process: Waiting for child process to terminate...\n");
        wait(&status);
        if (WIFEXITED(status)) {
            printf("Parent process: Child process exited with status %d\n",
WEXITSTATUS(status));
        } else {
            printf("Parent process: Child process exited abnormally\n");
        }
    }

    return 0;
}
```

Information Maintenance

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

int main() {
    // Get and print the process ID (PID) of the current process
    pid_t pid = getpid();
    printf("Process ID (PID): %d\n", pid);

    // Get and print the parent process ID (PPID) of the current process
    pid_t ppid = getppid();
    printf("Parent Process ID (PPID): %d\n", ppid);

    // Get and print the user ID (UID) of the current user executing the program
    uid_t uid = getuid();
    printf("User ID (UID): %d\n", uid);

    return 0;
}
```

File Management

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

int main() {
    int fd1, fd2;
    ssize_t bytes_read, bytes_written;
    char buffer[1024];

    // Open input file "input.txt" for reading
    fd1 = open("input.txt", O_RDONLY);
    if (fd1 == -1) {
        perror("Failed to open input file");
        exit(EXIT_FAILURE);
    }

    // Open output file "output.txt" for writing (create if not exists, truncate
if exists)
    fd2 = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
    if (fd2 == -1) {
        perror("Failed to open output file");
        exit(EXIT_FAILURE);
    }
```

```
    // Read from input file and write to output file
    while ((bytes_read = read(fd1, buffer, sizeof(buffer))) > 0) {
        bytes_written = write(fd2, buffer, bytes_read);
        if (bytes_written != bytes_read) {
            perror("Write error");
            exit(EXIT_FAILURE);
        }
    }

    if (bytes_read == -1) {
        perror("Read error");
        exit(EXIT_FAILURE);
    }

    // Close input and output files
    if (close(fd1) == -1) {
        perror("Failed to close input file");
        exit(EXIT_FAILURE);
    }
    if (close(fd2) == -1) {
        perror("Failed to close output file");
        exit(EXIT_FAILURE);
    }

    printf("File copied successfully.\n");

    return 0;
}
```

Process Creation

1. **fork()**
    - Used by parent to create a child process.
    - Returns child process ID to parent, 0 to the child.
    - Creates a duplicate process with separate address space.
2. **wait()**
    - Used by parent to wait for child process to finish.
    - Suspends parent execution until the child exits.
    - Returns child's exit status.
3. **exec()**
    - Replaces current process image with a new one.
    - Has multiple variants (execl(), execv(), etc.).
    - Does not return if successful.
4. **exit()**
    - Terminates the current process.

- Returns status to the parent.
- Performs cleanup tasks before termination.

Information Maintenance

1. **sleep()**
   - Suspends process execution for a specified time.
   - Resumes execution after the time interval or upon receiving a signal.
2. **getpid()**
   - Returns the process ID of the current process.
3. **getppid()**
   - Returns the parent process ID.
4. **alarm()**
   - Sets an alarm for a specified number of seconds.
   - Sends SIGALRM signal to the process when time expires.

File Management

1. **open()**
   - Opens a file and returns a file descriptor.
2. **read()**
   - Reads data from an open file.
3. **write()**
   - Writes data to an open file.
4. **close()**
   - Closes an open file descriptor and releases resources.

PIPE SYSTEM CALL

Syntax:

```
#include<unistd.h>
int pipe(int pipefd[2]);
```

pipe() function creates a unidirectional pipe for IPC. On success it return two file descriptors pipefd[0] and pipefd[1]. pipefd[0] is the reading end of the pipe. So, the process which will receive the data should use this file descriptor. pipefd[1] is the writing end of the pipe. So, the process that wants to send the data should use this file descriptor.
The program below creates a child process. The parent process will establish a pipe and will send the data to the child using writing end of the pipe and the child will receive that data and print on the screen using the reading end of the pipe.

Q. Program to send a message from parent process to child process using pipe()

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

```
#include<sys/wait.h>
int main()
{
int fd[2],n;
char buffer[100];
pid_t p;
pipe(fd); //creates a unidirectional pipe with two end fd[0] and
fd[1]
p=fork();
if(p>0) //parent
{
printf("Parent Passing value to child\n");
write(fd[1],"hello\n",6); //fd[1] is the write end of the pipe
wait();
}
else // child
{
printf("Child printing received value\n");
n=read(fd[0],buffer,100); //fd[0] is the read end of the pipe
write(1,buffer,n);
}
}
```

How it works?

The parent process create a pipe using pipe(fd) call and then creates a child process using fork(). Then the parent sends the data by writing to the writing end of the pipe by using the fd[1] file descriptor. The child then reads this using the fd[0] file descriptor and stores it in buffer. Then the child prints the received data from the buffer onto the screen.

Output

```
baljit@baljit:~/cse325$ ./a.out
Passing value to child
Child printing the received value
hello
```

Figure1: pipe() output

Q1: Which kind of data channel is created by pipe() system call: Unidirectional or bidirectional?

The pipe() system call creates a **unidirectional** data channel. This means data flows in one direction only—from one end of the pipe to the other.

Q2: What does the pipe() system call return on success?

On success, pipe() returns **0**. It also populates the file descriptors in the array provided as an argument:

- fd[0] is for **reading**.
- fd[1] is for **writing**.

Q3: What does the pipe() system call return on failure?

If the pipe() system call fails, it returns **-1**. Additionally, it sets the errno variable to indicate the specific error.

Q4: Why is fork() used in the above program?

The fork() system call is used to create a **child process**. This allows **Inter-Process Communication (IPC)** between the parent and child processes through the pipe.

Q5: Which process (parent or child) in the above code is using the writing end of the pipe?

Typically, the **child process** writes to the pipe (using fd[1]), while the **parent process** reads from the pipe (using fd[0]). However, this can vary based on the specific implementation of the program.

Program for IPC using named pipes (mkfifo())

The third method for IPC is using mkfifo() function. mkfifo() creates a named pipe which can be used exactly like a file. So, if you know how to read/write in a file this is a convenient method for IPC

Syntax:

```
#include<sys/types.h>
#include<sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

mkfifo() makes a FIFO special file with the name specified by *pathname* and the permissions are specified by *mode*. On success mkfifo() returns 0 while on error it returns -1.

The advantage is that this FIFO special file can be used by any process for reading or writing just like a normal file. This means to sender process can use the write() system call to write data into the pipe and the receiver process can use the read() system call to read data from the pipe, hence, completing the communication.
It is same as a pipe except that it is accessed as part of the filesystem. Multiple process can access it for writing and reading. When the FIFO special files is used for exchange of data by process, the entire data is passed *internally* without writing it on the filesystem. Hence, if you open this special file there will be no content written in it.

Note: The FIFO pipe works in blocked mode(by default) i.e., the writing process must be present on one end while the reading process must be present on the other side at the same time else the communication will not happen. Operating the FIFO special file in non-blocking mode is also possible.

The entire IPC process will consist of three programs:
Program1: to create a named pipe
Program2: process that will write into the pipe (sender process)
Program3: process that will receive data from pipe (receiver process)

//Program1: Creating fifo/named pipe (1.c)

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
int main()
        {
                int res;
                res = mkfifo("fifo1",0777); //creates a named pipe
with the name fifo1
                printf("named pipe created\n");
        }
```

//Now compile and run this program.

How it works?

This will simply create a named pipe (fifo1) with read, write and execute permission for all users. You can change this to whatever you prefer
Step 2 is to create a process which will use this pipe to send data. The below program will do that.

//**Program2:** Writing to a fifo/named pipe (2.c)

```
       #include<unistd.h>
        #include<stdio.h>
        #include<fcntl.h>
        int main()
        {
                    int res,n;
                    res=open("fifo1",O_WRONLY);
                    write(res,"Message",7);
                    printf("Sender Process %d sent the
data\n",getpid());
        }
```

Compile this program as
$gcc -o 2 2.c
//Note: If you run this you will not see any output

How it works?

The above code opens the pipe created previously in writing mode (because it wants to send data). Then it uses “write” system call to write some data into it. Finally, it prints a message

using printf. But when you compile and run it, it won't run because by default the sender runs in BLOCKING mode which means that until the receiver is not there the sender process gets blocked. Hence, you need a receiver process also.

The **third step** is to create the receiver process. The below program does so.

//**Program 3**: Reading from the named pipe (3.c)

```
#include<unistd.h>
 #include<stdio.h>
 #include<fcntl.h>
 int main()
 {
                int res,n;
                char buffer[100];
                res=open("fifo1",O_RDONLY);
                n=read(res,buffer,100);
                printf("Reader process %d started\n",getpid());
                printf("Data received by receiver %d is:
%s\n",getpid(), buffer);
 }
```

Compile the program as

$ gcc -o 3 3.c

How it works?

This program connects to the pipe in reading mode and reads the data into buffer and prints it. But again this program will not run. Because the receiver is BLOCKED until the sender is there.

Therefore, run both the object files simultaneously as

$./2 & ./3

and you will see the output as

```
baljit@baljit:~/cse325$ ./2 & ./3
[1] 33
Sender Process 33 sent the data
Reader process 34 started
Data received by receiver 34 is: Message
[1]+  Done                    ./2
baljit@baljit:~/cse325$
```

Q1: Is a named pipe bidirectional or unidirectional?

A **named pipe (FIFO)** is **bidirectional**, meaning data can flow in both directions. However, in practice, one direction is typically used for writing, and the other for reading to avoid conflicts. Also, this behavior may depend on how the pipe is opened (e.g., `O_WRONLY` or `O_RDONLY`).

Q2: What is the default mode of a named pipe: blocking or non-blocking?

The **default mode** of a named pipe is **blocking**.

- **Blocking mode** means that if a process tries to read from an empty pipe or write to a full pipe, it will pause (block) until the pipe is ready for the operation.

Q3: What is meant by Blocking-Send?

Blocking-Send refers to a communication mechanism where:

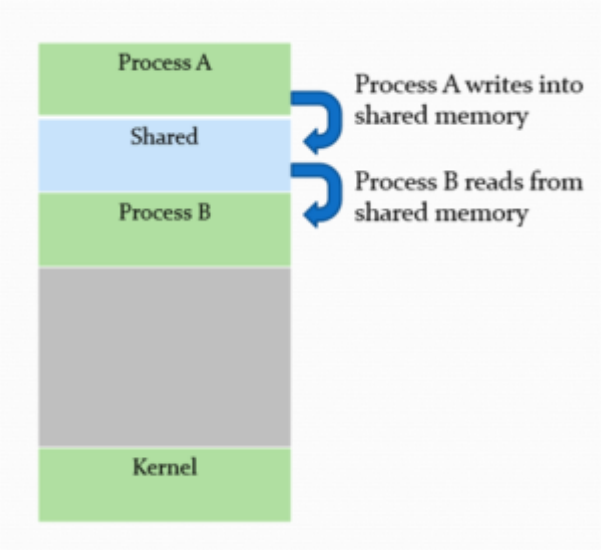
- The sending process is **blocked (paused)** until the message it wants to send is successfully delivered or queued by the recipient.
- This ensures reliability but can cause delays if the recipient is not ready.

Q4: Can a process create two named pipes for communication with two different processes?

Yes, a process can create **two named pipes (FIFOs)** to communicate with two different processes. Each named pipe will have its own unique name in the filesystem, and the process can use them simultaneously for communication.

Program for IPC using shared memory

Shared Memory is the fastest inter-process communication (IPC) method. The operating system maps a memory segment in the address space of several processes so that those processes can read and write in that memory segment.The overview is as shown below:
Two functions:shmget() and shmat() are used for IPC using shared memory. shmget() function is used to create the shared memory segment while shmat() function is used to attach the shared segment with the address space of the process.



Shared Memory

Syntax (shmget()):

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
```

The first parameter specifies the unique number (called key) identifying the shared segment. The second parameter is the size of the shared segment e.g. 1024 bytes or 2048 bytes. The third parameter specifies the permissions on the shared segment. On success the shmget() function returns a valid identifier while on failure it return -1.

Syntax (shmat()):

```
#include <sys/types.h>
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

shmat() is used to attach the created shared segment with the address space of the calling process. The first parameter here is the identifier which shmget() function returns on success. The second parameter is the address where to attach it to the calling process. A NULL value of second parameter means that the system will automatically choose a suitable address. The third parameter is '0' if the second parameter is NULL, otherwise, the value is specified by SHM_RND.
We will write two program for IPC using shared memory. *Program 1* will create the shared segment, attach to it and then write some content into it. Then *Program 2* will attach itself to the shared segment and read the value written by Program 1.

//Program 1: This program creates a shared memory segment, attaches itself to it and then writes some content into the shared memory segment.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
int i;
void *shared_memory;
char buff[100];
int shmid;
shmid=shmget((key_t)2345, 1024, 0666|IPC_CREAT); //creates shared
memory segment with key 2345, having size 1024 bytes. IPC_CREAT is
used to create the shared segment if it does not exist. 0666 are the
permisions on the shared segment
printf("Key of shared memory is %d\n",shmid);
shared_memory=shmat(shmid,NULL,0); //process attached to shared
memory segment
printf("Process attached at %p\n",shared_memory); //this prints the
address where the segment is attached with this process
printf("Enter some data to write to shared memory\n");
read(0,buff,100); //get some input from user
strcpy(shared_memory,buff); //data written to shared memory
printf("You wrote : %s\n",(char *)shared_memory);
}
```

How it works?

shmget() function creates a segment with key 2345, size 1024 bytes and read and write permissions for all users. It returns the identifier of the segment which gets store in shmid. This identifier is used in shmat() to attach the shared segment to the address space of the process. NULL in shmat() means that the OS will itself attach the shared segment at a suitable address of this process.

Then some data is read from the user using read() system call and it is finally written to the shared segment using strcpy() function.

Output

```
Key of shared memory is 0
Process attached at 0x7ffe040fb000
Enter some data to write to shared memory
Hello World
You wrote : Hello World
```

//Program 2: This program attaches itself to the shared memory segment created in Program 1. Finally, it reads the content of the shared memory

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
int i;
void *shared_memory;
char buff[100];
int shmid;
shmid=shmget((key_t)2345, 1024, 0666);
printf("Key of shared memory is %d\n",shmid);
shared_memory=shmat(shmid,NULL,0); //process attached to shared
memory segment
printf("Process attached at %p\n",shared_memory);
printf("Data read from shared memory is : %s\n",(char
*)shared_memory);

}
```

How it works?

shmget() here generates the identifier of the same segment as created in Program 1. Remember to give the same key value. The only change is, do not write IPC_CREAT as the shared memory segment is already created. Next, shmat() attaches the shared segment to the current process. After that, the data is printed from the shared segment. In the output, you will see that it is the same data that you have written while executing the Program 1.

Output

```
Key of shared memory is 0
Process attached at 0x7f76b4292000
Data read from shared memory is : Hello World
```

3.26 Design a program using ordinary pipes in which one process sends a string message to a second process, and the second process reverses the case of each character in the message and sends it back to the first process. For example, if the first process sends the message `Hi There`, the second process will return `hI tHERE`. This will require using two pipes, one for sending the original message from the first to the second process and the other for sending the modified message from the second to the first process. You can write this program using either UNIX or Windows pipes.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <ctype.h>
#define BUFFER_SIZE 100
void reverse_case(char *str) {
    for (int i = 0; str[i] != '\0'; i++) {
        if (islower(str[i]))
            str[i] = toupper(str[i]);
        else if (isupper(str[i]))
            str[i] = tolower(str[i]);
    }
}
int main() {
    int pipe1[2], pipe2[2]; // Two pipes
    pid_t pid;
    char message[BUFFER_SIZE] = "Hi There";
    char modified_message[BUFFER_SIZE];
```

```c
    if (pipe(pipe1) == -1 || pipe(pipe2) == -1) {
        perror("Pipe creation failed");
        exit(1);
    }
    pid = fork();
    if (pid < 0) {
        perror("Fork failed");
        exit(1);
    }
    if (pid > 0) { // Parent process
        close(pipe1[0]); // Close read end of pipe1
        close(pipe2[1]); // Close write end of pipe2

        write(pipe1[1], message, strlen(message) + 1);
        close(pipe1[1]); // Close write end after sending

        read(pipe2[0], modified_message, BUFFER_SIZE);
        close(pipe2[0]); // Close read end after receiving
        printf("Modified Message: %s\n", modified_message);
    } else { // Child process
        close(pipe1[1]); // Close write end of pipe1
        close(pipe2[0]); // Close read end of pipe2

        read(pipe1[0], modified_message, BUFFER_SIZE);
        close(pipe1[0]); // Close read end after receiving

        reverse_case(modified_message);

        write(pipe2[1], modified_message, strlen(modified_message) + 1);
        close(pipe2[1]); // Close write end after sending
    }
    return 0;
}
```

3.27 Design a file-copying program named `filecopy.c` using ordinary pipes. This program will be passed two parameters: the name of the file to be copied and the name of the destination file. The program will then create an ordinary pipe and write the contents of the file to be copied to the pipe. The child process will read this file from the pipe and write it to the destination file. For example, if we invoke the program as follows:

```
./filecopy input.txt copy.txt
```

the file `input.txt` will be written to the pipe. The child process will read the contents of this file and write it to the destination file `copy.txt`. You may write this program using either UNIX or Windows pipes.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#define BUFFER_SIZE 1024
int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <source_file> <destination_file>\n", argv[0]);
        exit(1);
    }
```

```c
int pipefd[2];
if (pipe(pipefd) == -1) {
    perror("Pipe creation failed");
    exit(1);
}
pid_t pid = fork();
if (pid < 0) {
    perror("Fork failed");
    exit(1);
}
```

```c
if (pid > 0) { // Parent process
    close(pipefd[0]); // Close read end

    int src_fd = open(argv[1], O_RDONLY);
    if (src_fd == -1) {
        perror("Error opening source file");
        exit(1);
    }
    char buffer[BUFFER_SIZE];
    ssize_t bytes_read;
    while ((bytes_read = read(src_fd, buffer, BUFFER_SIZE)) > 0) {
        write(pipefd[1], buffer, bytes_read);
    }

    close(src_fd);
    close(pipefd[1]); // Close write end
} else { // Child process
    close(pipefd[1]); // Close write end
    int dest_fd = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (dest_fd == -1) {
        perror("Error opening destination file");
        exit(1);
    }
    char buffer[BUFFER_SIZE];
    ssize_t bytes_read;
    while ((bytes_read = read(pipefd[0], buffer, BUFFER_SIZE)) > 0) {
        write(dest_fd, buffer, bytes_read);
    }
    close(dest_fd);
    close(pipefd[0]); // Close read end
}
return 0;
```

3.21 The Collatz conjecture concerns what happens when we take any positive integer n and apply the following algorithm:

$$n = \begin{cases} n/2, & \text{if n is even} \\ 3 \times n + 1, & \text{if n is odd} \end{cases}$$

The conjecture states that when this algorithm is continually applied, all positive integers will eventually reach 1. For example, if $n = 35$, the sequence is

$$35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1$$

Write a C program using the `fork()` system call that generates this sequence in the child process. The starting number will be provided from the command line. For example, if 8 is passed as a parameter on the command line, the child process will output $8, 4, 2, 1$. Because the parent and child processes have their own copies of the data, it will be necessary for the child to output the sequence. Have the parent invoke the `wait()` call to wait for the child process to complete before exiting the program. Perform necessary error checking to ensure that a positive integer is passed on the command line.

```c
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <ctype.h>
void collatz_recursive(int n) {
    if (n == 1) {
        printf("\n");
        return;
    }
    if (n % 2 == 0)
        {
        n=n/2;
        printf("%d ", n);
        collatz_recursive(n);
        }
    else
        {
        n=(3*n)+1;
        printf("%d ", n);
        collatz_recursive(n);
        }
}
int main(int argc, char *argv[]) {
 if (argc != 2) {
        fprintf(stderr, "Usage: %s <positive integer>\n", argv[0]);
        return 1;
    }
    int n = atoi(argv[1]);
    if (n <= 0) {
        fprintf(stderr, "Error: Please enter a positive integer.\n");
        return 1;
    }
    pid_t pid = fork();
    if (pid < 0) {
        fprintf(stderr, "Fork failed\n");
        return 1;
    }
    if (pid == 0) {
        printf("%d",n);
        collatz_recursive(n);
    } else {
        wait(NULL);
    }
    return 0;
}
```

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

```
void collatz_recursive(int n) {
        if (n == 1) {
        printf("\n");
        return;
        }
        if (n % 2 == 0)
   {
        n=n/2;
        printf("%d ", n);
        collatz_recursive(n);
   }
        else
   {
        n=(3*n)+1;
        printf("%d ", n);
        collatz_recursive(n);
   }
}
int main(int argc, char *argv[]) {
 if (argc != 2) {
        fprintf(stderr, "Usage: %s <positive integer>\n", argv[0]);
        return 1;
        }
        int n = atoi(argv[1]);
        if (n <= 0) {
        fprintf(stderr, "Error: Please enter a positive integer.\n");
        return 1;
        }
        pid_t pid = fork();
        if (pid < 0) {
        fprintf(stderr, "Fork failed\n");
        return 1;
        }
        if (pid == 0) {
        printf("%d",n);
        collatz_recursive(n);
        } else {
        wait(NULL);
        }
        return 0; }
```