3.1 What will the output be at LINE A in the program?

Answer: At LINE A, the output will be `PARENT: value = 5`. This occurs because the `fork()` system call creates a child process with its own separate copy of the `value` variable. The child process modifies `value` (adding 15), but this change does not affect the parent's copy, which remains 5.

3.2 How many processes are created by the program shown in Figure 3.31?

Answer: The program creates a total of **eight processes**. The first `fork()` creates one additional process (2 total), the second `fork()` creates two more (4 total), and the third `fork()` doubles the number of processes again (8 total). Thus, including the original parent, there are eight processes in total.

3.3 What are three major complications that concurrent processing adds to an operating system?

Answer:

1. **Synchronization Issues**: Ensuring that processes access shared resources in a controlled manner to prevent race conditions and data inconsistencies requires robust synchronization mechanisms, like mutexes and semaphores.
2. **Deadlock**: Concurrent processes may end up in a deadlock situation, where each process waits indefinitely for resources held by others. This necessitates the implementation of deadlock detection and prevention strategies.
3. **Resource Management**: Efficiently allocating CPU time, memory, and I/O resources among multiple concurrent processes becomes more complex, requiring sophisticated scheduling algorithms to balance fairness and performance.

3.4 What happens during a context switch if the new context is already loaded in a system with multiple register sets?

Answer: If the new context is already loaded in one of the register sets during a context switch, the operating system can switch directly to that register set without saving the current context or restoring the new one. This leads to a faster context switch. However, if the new context is in memory and all register sets are in use, the system must first save the current context to memory before loading the new context, which incurs additional overhead and delays.

3.5 Which states are shared between the parent process and the child process after a `fork()` operation?

Answer: The state that is shared between the parent process and the child process after a `fork()` operation is **c. Shared memory segments**. Both processes have their own copies of the stack and heap, but shared memory segments are accessible to both processes.

3.6 Does the algorithm for implementing the "exactly once" semantic execute correctly even if the ACK message sent back to the client is lost?

Answer: If the ACK message is lost, the algorithm may not preserve the "exactly once" semantic. The client, upon not receiving the ACK, might resend the request, leading to potential duplicate executions on the server. For "exactly once" to be maintained, mechanisms must be in place to differentiate between original and duplicate requests based on unique identifiers, ensuring that each request is processed only once.

3.7 What mechanisms are required to guarantee the "exactly once" semantic for RPCs in a distributed system susceptible to server failure?

Answer: To guarantee the "exactly once" semantic for RPCs in a system with potential server failures, the following mechanisms are essential:

1. **Idempotent Operations**: Ensure that the operations can be safely retried without adverse effects.
2. **Unique Request Identifiers**: Assign a unique identifier to each RPC request, allowing the server to recognize and ignore duplicate requests.
3. **Logging and Recovery**: Maintain logs of processed requests to allow the server to recover its state after a failure and reprocess requests as needed.
4. **Acknowledgments**: Implement a reliable acknowledgment mechanism to confirm receipt of requests and prevent unnecessary retransmissions.

3.8 Describe the actions taken by a kernel to context-switch between processes.

Answer: When a kernel performs a context switch, it typically follows these steps:

1. **Save the Current Context**: The kernel saves the state of the currently running process, including CPU registers, program counter, and stack pointer, into its Process Control Block (PCB).
2. **Update Process States**: The kernel updates the state of the current process to reflect that it is no longer running (e.g., set it to "waiting" or "ready").
3. **Select a New Process**: The kernel chooses the next process to run based on scheduling algorithms (e.g., round-robin, priority-based).
4. **Restore the New Context**: The kernel loads the saved state of the selected process from its PCB, restoring CPU registers and program counter.
5. **Update Process States Again**: The kernel updates the new process's state to "running" and proceeds to execute it.

3.9 Construct a process tree similar to Figure 3.7.

Answer: To create a process tree, you would use the command `ps -ael` on a UNIX or Linux system to display the current processes along with their parent process IDs. Here is a simple example:

```
swift
Copy code
PID  PPID CMD
1    0    init
2    1    bash
3    1    ps
4    2    grep
```

This shows a simple hierarchy where `init` is the parent of `bash`, and `bash` is the parent of `ps` and `grep`.

3.10 Explain the role of the init (or systemd) process on UNIX and Linux systems in regard to process termination.

Answer: The `init` (or `systemd`) process is the first process started by the kernel during booting and has a PID of 1. It is responsible for managing system startup and shutdown, and it handles orphaned processes. When a child process terminates, `init` reaps the process (by executing `wait`) to collect its exit status and free up system resources. This prevents zombie processes, ensuring proper cleanup of terminated processes.

3.11 How many processes are created by the program shown in Figure 3.32?

Answer: The program creates a total of **8 processes**. Each call to `fork()` doubles the number of processes:

1. First `fork()` creates 2 (parent + child).
2. Second `fork()` doubles it to 4.
3. Third `fork()` doubles it again to 8.

3.12 Explain the circumstances under which the line of code marked `printf("LINE J")` will be reached.

Answer: The line `printf("LINE J")` will be reached if the `execlp()` function call fails. If the `execlp()` successfully replaces the child process's memory with the `ls` command, the line will not be executed. If it fails (e.g., due to an invalid command), the child process will proceed to execute the `printf` statement.

3.13 Identify the values of `pid` at lines A, B, C, and D in the program.

Answer:

- **Line A (Child Process)**: The `pid` value is `0` because this is executed by the child process.
- **Line B (Child Process)**: The `pid1` value is the actual PID of the child process (e.g., `2603`).
- **Line C (Parent Process)**: The `pid` value will be the actual PID of the child process (e.g., `2603`).

- **Line D (Parent Process)**: The `pid1` value is the actual PID of the parent process (e.g., `2600`).

3.14 When are ordinary pipes more suitable than named pipes and vice versa?

Answer:

- **Ordinary Pipes**: More suitable for communication between closely related processes (e.g., a parent and its child) that are running in a single application, as they are simpler and faster to set up.
- **Named Pipes**: More suitable for communication between unrelated processes or processes that are not in a parent-child relationship. They allow for more flexible inter-process communication as they can be accessed by any process that knows the pipe's name.

3.15 Describe the undesirable consequences of not enforcing "at most once" or "exactly once" semantics.

Answer: Not enforcing these semantics can lead to several issues:

- **Duplicate Executions**: If a request is retried without the "exactly once" guarantee, the operation may be executed multiple times, leading to inconsistent states (e.g., double debiting an account).
- **Lost Messages**: Without "at most once," messages may be processed multiple times, causing resource wastage or corruption of data.

Uses for a mechanism without these guarantees: A system might benefit from a non-reliable RPC mechanism in scenarios where occasional duplicates or message losses are acceptable and the overhead of implementing strict guarantees is not justified (e.g., logging events).

3.16 What will the output be at lines X and Y in the program?

Answer:

- **Line X (Child Output)**: The child will print modified values of `nums`, which will be `0, -1, -4, -9, -16` for indices 0 to 4, respectively.
- **Line Y (Parent Output)**: The parent will print the original values of `nums`, which will be `0, 1, 2, 3, 4`, since the parent and child have separate memory spaces.

3.17 What are the benefits and disadvantages of synchronous and asynchronous communication?

Answer:

- **Synchronous Communication**:

    - *Benefits*: Simpler design; processes can coordinate directly and easily share data.
    - *Disadvantages*: Can lead to blocking; a slow process can hold up others, reducing efficiency.
- **Asynchronous Communication**:
    - *Benefits*: Non-blocking; processes can continue executing while waiting for messages, improving throughput.
    - *Disadvantages*: More complex to implement; requires mechanisms for handling message ordering and delivery confirmation.

3.18 What are the benefits and disadvantages of automatic and explicit buffering?

Answer:

- **Automatic Buffering**:
    - *Benefits*: Simplifies programming; the system manages buffers without manual intervention.
    - *Disadvantages*: Less control for the programmer; may lead to inefficiencies if the buffer size is not optimal.
- **Explicit Buffering**:
    - *Benefits*: Greater control over how and when data is buffered; can optimize performance based on specific needs.
    - *Disadvantages*: Increased complexity; programmers must handle buffer management, which can lead to errors.

3.19 What are the benefits and disadvantages of send by copy and send by reference?

Answer:

- **Send by Copy**:
    - *Benefits*: Simpler to understand; each process gets its own copy, preventing unintended modifications.
    - *Disadvantages*: Higher overhead in terms of memory and performance, especially for large data structures.
- **Send by Reference**:
    - *Benefits*: More efficient, as it avoids copying large data structures, saving memory and time.
    - *Disadvantages*: Risk of unintended modifications and complexities related to memory management and pointer validity.

3.20 What are the benefits and disadvantages of fixed-sized and variable-sized messages?

Answer:

- **Fixed-Sized Messages**:
  - *Benefits*: Simplicity in design and processing; easier to allocate and manage buffers.
  - *Disadvantages*: Inefficiency when sending variable-sized data, leading to potential waste of space.
- **Variable-Sized Messages**:
  - *Benefits*: More efficient for varying data sizes, reducing waste of memory.
  - *Disadvantages*: Increased complexity in managing buffers and ensuring correct handling of message sizes.