OS lab 2

Maryam soleimani 401222075

Producer/consumer:

For the producer consumer the overall approach is a s follows for both using pipe and queue:

The producer has a pro_count. This is a counter that counts the number of items in the buffer. When producer produces an item it puts that item in the buffer and through the output_pipe notifies the consumer that it has produced an item. Then it adds 1 to the pro_count because one item is put in the buffer. There is another pipe here which is the notify_pipe and the consumer notifies the producer through this pipe that it has consumed an item and when an item is consumed it means that one item is popped out of buffer so 1 will be subtracted from pro_count. I the pro_count is equal or higher than the buffer_size it means that the buffer is full and it must wait until the consumer consumes some item from the buffer and then it has the capacity and then the producer can put an item in the buffer.

On the consumer side, The consumer has a con_count. This is a counter that counts the number of items in the buffer. When producer produces an item it puts that item in the buffer and through the output_pipe notifies the consumer that it has produced an item. Then it adds 1 to the con_count because one item is put in the buffer. There is another pipe here which is the notify_pipe and the consumer notifies the producer through this pipe that it has consumed an item and when an item is consumed it means that one item is popped out of buffer so 1 will be subtracted from con_count. If the con_count becomes equal to zero, it means that the buffer is empty and it must wait until the producer produces some item to put in the buffer and then it can consume it. when the producer is waiting because of full buffer and the consumer is waiting because of empty buffer, it must not only be shown in the terminal but also they must actually wait in the system.

Pipe code:

Key Components

1. **Pipes:**
    - Pipe(True): Creates bidirectional pipes for communication between processes.
    - Pipes used:
        - pipe_1: To send production status ("produced" or None) from the producer to the consumer.
        - notify_pipe: To send acknowledgments ("consumed") from the consumer to the producer.
        - data_pipe: To transfer the actual data (items) produced by the producer to the consumer.
2. **Producer (create_items):**
    - Produces a fixed number (30) of random items.

- Sends production status ("produced") and the produced item to the consumer using output_pipe and dataa_pipe.
- Tracks the number of items in the buffer using pro_counter.
- Waits when the buffer is full (pro_counter >= buffer_size) and processes acknowledgment messages ("consumed") to decrement pro_counter.

3. **Consumer (consume_items):**
    - Receives production status ("produced") and increments its buffer counter (con_counter).
    - Retrieves items from data_pipe for processing.
    - Sends acknowledgment ("consumed") to notify the producer.
    - Waits if the buffer is empty (con_counter == 0).
    - Terminates gracefully upon receiving None from the producer, indicating production is complete.
4. **Main Process:**
    - Initializes pipes for communication.
    - Creates and starts producer and consumer processes.
    - Ensures unused pipe ends are closed in the main process to avoid resource leaks.
    - Waits for both processes to complete.

Workflow

1. **Initialization:**
    - pipe_1, notify_pipe, and data_pipe are created for communication between the producer and consumer.
    - Two processes (producer and consumer) are started.
2. **Producer Process (create_items):**
    - Produces 30 random items.
    - Sends each item via data_pipe and signals the consumer using pipe_1 ("produced").
    - Increments pro_counter to track items in the buffer.
    - Waits when pro_counter reaches the buffer_size limit.
    - Processes acknowledgment messages ("consumed") from the consumer to decrement pro_counter.

- Sends None to pipe_1 to indicate production is complete.

3. **Consumer Process (consume_items):**
    - Receives production signals ("produced") from pipe_1.
    - Increments con_counter and retrieves items from data_pipe.
    - Sends acknowledgment ("consumed") to the producer.
    - Decrements con_counter after processing each item.
    - Waits if the buffer is empty (con_counter == 0).
    - Terminates gracefully upon receiving None from the producer.
4. **Main Process:**
    - Waits for the producer and consumer processes to complete using join().

Key Concepts and Behavior

1. **Buffer Management:**
    - pro_counter: Tracks the number of unacknowledged items in the buffer (items produced but not yet consumed).
    - con_counter: Tracks the number of unprocessed items in the consumer's local buffer.
2. **Synchronization:**
    - The producer waits when the buffer is full (pro_counter >= buffer_size) and resumes after acknowledgments.
    - The consumer waits when the buffer is empty (con_counter == 0).
3. **Termination:**
    - The producer signals completion by sending None to the consumer.
    - The consumer exits upon receiving None.
4. **Pipes for Communication:**
    - pipe_1: Coordinates production status between the producer and consumer.
    - notify_pipe: Sends acknowledgments from the consumer to the producer.
    - data_pipe: Transfers produced items to the consumer.

There are three scenarios:

1. consumer and producer consume and produce at the same time (they are synch):

   there are 4 sleep times in the code set them as follows from up to down as follows:

   0.01 0.01 0.01 0.01
2. Empty buffer (consumer works faster so its sleep time is less) :
   there are 4 sleep times in the code set them as follows from up to down as follows:
   1 1 0.01 0.01

3. Full buffer (producer works faster so its sleep time is less):
   0.01 0.01 0.01 1

Queue code:

Key Components

1. **Queues:**
   - **queue**: Signals the status of production to the consumer, sending messages like "produced" or None.
   - **notify_queue**: Sends acknowledgment messages ("consumed") from the consumer to the producer.
   - **data_queue**: Holds the actual data (produced items) for the consumer to retrieve.
2. **Producer Function (create_items):**
   - Produces 30 random numbers between 0 and 1000.
   - Tracks the number of unacknowledged items using pro_counter.
   - Sends a "produced" message to the consumer via queue and the actual item to data_queue.
   - Waits when the buffer is full (i.e., pro_counter >= buffer_size) until it receives enough acknowledgments from the consumer to continue.
3. **Consumer Function (consume_items):**
   - Retrieves production signals ("produced") from queue and increments con_counter.
   - Processes items by retrieving them from data_queue and sending "consumed" acknowledgments to the producer via notify_queue.
   - Exits gracefully when it receives a None signal from the producer, indicating that production is complete.

- Waits when the buffer is empty (con_counter == 0 and data_queue.empty()).

4. **Main Process (__main__):**
    - Initializes three queues for communication.
    - Starts two separate processes: one for the producer and one for the consumer.
    - Waits for both processes to finish using join().

Workflow

1. **Initialization:**
    - Three queues (queue, notify_queue, and data_queue) are created.
    - The producer and consumer processes are started.
2. **Producer Process:**
    - Produces 30 random items and puts each into data_queue.
    - Sends a "produced" signal to queue to notify the consumer.
    - Waits when the buffer is full by monitoring pro_counter and decrementing it upon receiving "consumed" acknowledgments.
    - Sends None to queue after finishing production, signaling the consumer to terminate.
3. **Consumer Process:**
    - Waits for "produced" signals from queue to start consuming.
    - Retrieves items from data_queue, processes them, and sends "consumed" acknowledgments via notify_queue.
    - Exits when it receives None from queue.
4. **Synchronization:**
    - The producer monitors pro_counter to avoid overflowing the buffer (data_queue).
    - The consumer waits when con_counter reaches 0 or data_queue is empty, ensuring it doesn’t consume non-existent items.

Important Variables

1. **pro_counter:**
    - Tracks the number of unacknowledged items in the buffer.
    - Incremented every time an item is produced.

- Decremented upon receiving "consumed" acknowledgment from the consumer.

2. **con_counter:**
    - Tracks the number of items ready to be consumed.
    - Incremented every time a "produced" signal is received.
    - Decremented after consuming an item.

Graceful Termination

- The producer signals completion by sending None to queue after producing all items.
- The consumer detects None and exits its loop, ensuring the process terminates gracefully.

There are three scenarios:

4. consumer and producer consume and produce at the same time (they are synch):

   there are 4 sleep times in the code set them as follows from up to down as follows:

   0.01 0.01 0.01 0.01
5. Empty buffer (consumer works faster so its sleep time is less) :
   there are 4 sleep times in the code set them as follows from up to down as follows:
   1 1 0.01 0.01

6. Full buffer (producer works faster so its sleep time is less):

0.01 0.01 0.01 1

Reader writer:

Reader priority:

Key Components

1. **Shared Resources:**
    - **shared_resource:** Represents the resource being read or written. It is an integer shared across all processes and is updated by writers.
    - **reader_count:** Tracks the number of active readers. It's an integer wrapped in a multiprocessing.Value to ensure atomic updates.
2. **Locks:**
    - **write_lock:** Ensures that only one writer can access the shared resource at a time.

- **reader_count_lock:** Protects reader_count to prevent race conditions when readers increment or decrement it.

3. **Processes:**
    - **writer:** Writes to the shared resource. It waits until there are no active readers and then writes exclusively.
    - **reader:** Reads the shared resource. Multiple readers can access it simultaneously, but no reader can proceed if a writer has acquired the lock.
4. **Process Manager (process_manager):**
    - Spawns and manages multiple reader and writer processes based on user input.

How It Works

Writer Function

1. **Waiting for Readers to Finish:**
    - Continuously checks the reader_count using reader_count_lock.
    - If there are active readers (reader_count > 0), it waits and retries after a short delay.
2. **Exclusive Writing:**
    - Once there are no active readers, it acquires the write_lock to write to the shared resource.
    - Simulates writing by updating shared_resource and sleeping for 1 second.
    - Releases the lock after writing.

Reader Function

1. **Incrementing Reader Count:**
    - Before reading, increments the reader_count under the protection of reader_count_lock.
    - This signals to writers that there is at least one active reader.
2. **Reading the Resource:**
    - Reads the shared_resource and prints its value.
    - Simulates reading by sleeping for 1 second.
3. **Decrementing Reader Count:**
    - After reading, decrements the reader_count under the reader_count_lock.

Process Manager

1. Initializes the locks and shared variables.
2. Spawns a specified number of reader and writer processes.
3. Starts all processes and waits for them to complete using join().

Execution Flow

1. **Readers:**
    - Start reading as soon as they are spawned.
    - Multiple readers can read simultaneously because the write_lock does not restrict them.
2. **Writers:**
    - Wait until all active readers finish (reader_count == 0) before acquiring the write_lock.
    - Write exclusively, ensuring no other readers or writers interfere.
3. **Synchronization:**
    - reader_count_lock ensures atomic updates to reader_count, preventing inconsistencies in the count of active readers.
    - write_lock ensures only one writer can access the shared resource at a time.

Writer priority:

Key Components

1. **Shared Resource:**
    - **shared_resource:** A shared integer wrapped in multiprocessing.Value that allows safe access and updates across multiple processes.
2. **Lock:**
    - **write_lock:** A multiprocessing.Lock to ensure mutual exclusion when accessing the shared resource. Only one process (reader or writer) can access the resource at a time.
3. **Processes:**
    - **writer:** Writes to the shared resource exclusively, incrementing its value.
    - **reader:** Reads the shared resource but checks if the lock is available before proceeding. If the resource is locked by a writer, the reader waits.
4. **Process Manager (process_manager):**

- Creates and manages multiple reader and writer processes based on user-specified numbers.

How It Works

Writer Function

1. **Acquire Lock:**
    - A writer process acquires the write_lock before accessing the shared resource.
    - This ensures exclusive access to prevent conflicts with readers or other writers.
2. **Write to the Resource:**
    - The writer increments the value of shared_resource and simulates writing with a sleep of 1 second.
    - Prints messages indicating the start and completion of the writing process.
3. **Release Lock:**
    - After writing, the lock is released, allowing other readers or writers to access the resource.
4. **Repeat Writing:**
    - The writer sleeps for 1 second before attempting to write again, simulating intermittent write operations.

Reader Function

1. **Attempt to Acquire Lock Non-Blocking:**
    - The reader tries to acquire the write_lock using block=False.
    - If successful, it reads the value of shared_resource.
    - If the lock is held by a writer, the reader prints a message indicating it is waiting and retries later.
2. **Read the Resource:**
    - If the lock is acquired, the reader prints the current value of shared_resource and simulates a read operation with a sleep of 1 second.
3. **Release Lock:**
    - After reading, the lock is released, allowing other processes to proceed.
4. **Repeat Reading:**

- The reader sleeps for a random interval (1 to 3 seconds) before attempting to read again.

Process Manager

1. **Initialize Shared Resources:**
    - Creates write_lock and shared_resource.
2. **Spawn Writer Processes:**
    - Creates num_writers writer processes and starts them.
3. **Spawn Reader Processes:**
    - Creates num_readers reader processes and starts them.
4. **Wait for Completion:**
    - Calls join() on all processes to ensure they complete their execution.

Execution Flow

1. **Writers:**
    - Each writer process writes to the shared resource in mutual exclusion.
    - Writers cannot operate simultaneously with readers or other writers.
2. **Readers:**
    - Each reader process attempts to acquire the lock to read the resource.
    - If a writer is writing, readers wait.
3. **Synchronization:**
    - The write_lock ensures mutual exclusion for both readers and writers.
    - Readers can only proceed if no writer is active