Advanced Python Programming

# LESSON 7: Practical Concurrency Patterns

**Why This Matters:** Imagine you're running a restaurant. If you only have one waiter handling one customer at a time, service will be painfully slow. But if you have multiple waiters working simultaneously on different tasks, you can serve many customers efficiently. That's concurrency - and understanding when to use different types of "workers" (threads vs processes) makes the difference between a fast, responsive application and a slow, frustrating one.

**The Problem We're Solving:** Your applications will face two types of bottlenecks:

- **Waiting time**: Your program sits idle waiting for files to download, databases to respond, or APIs to return data
- **Computation time**: Your program works hard crunching numbers, processing images, or analyzing data

Using the wrong concurrency approach is like having marathon runners deliver pizza (slow) or pizza delivery drivers run marathons (inefficient). We'll learn when to use which type of "worker."

Learning Objectives:

By the end of this lesson, participants will be able to:

- Understand the mental model of Python's GIL and when it matters
- Choose between threads and processes for different workload types
- Use thread pools effectively for I/O-bound tasks
- Use process pools effectively for CPU-bound tasks
- Apply simple queue patterns for coordinated work

Lesson Outline:

**I. The Mental Model: GIL and Two Types of Work (10 min)**

**Think of Python's GIL (Global Interpreter Lock) like a single bathroom key in an office:**

- **The key rule**: Only one person (thread) can use the bathroom (Python interpreter) at a time
- **The exception**: When someone's waiting (doing I/O), they give up the key for others to use
- **The workaround**: Having separate offices (processes) means each has its own bathroom

**Two Types of Work - Different Solutions:**

1. **I/O-Bound Work ("Waiting Work")**
    - Like waiting for elevators, phone calls, or package deliveries
    - Examples: downloading files, database queries, API calls
    - **Solution**: Use threads (multiple waiters sharing the Python interpreter)
    - **Why threads work**: While one thread waits for I/O, Python gives the "key" to another thread
2. **CPU-Bound Work ("Thinking Work")**

- Like solving math problems, processing images, or analyzing data
- Examples: calculations, data transformation, image processing
- **Solution**: Use processes (separate Python interpreters)
- **Why processes work**: Each process has its own "key" (interpreter)

Let's see this in action:

```
import threading
import time
import multiprocessing
import requests

def demonstrate_io_vs_cpu_work():
    """Show why threads help I/O but not CPU work."""

    def io_task(task_id, duration):
        """Simulate I/O work - waiting for something"""
        print(f"I/O Task {task_id} starting (will wait {duration}s)")
        time.sleep(duration)  # This releases the GIL!
        print(f"I/O Task {task_id} completed")
        return f"IO-{task_id}"

    def cpu_task(task_id, work_amount):
        """Simulate CPU work - actual computation"""
        print(f"CPU Task {task_id} starting (will compute {work_amount}
items)")
        # This DOESN'T release the GIL - pure Python computation
        result = sum(i ** 2 for i in range(work_amount))
        print(f"CPU Task {task_id} completed")
        return result

    print("=== I/O Work: Sequential vs Threaded ===")

    # Sequential I/O (slow)
    start = time.time()
    io_task(1, 1)
    io_task(2, 1)
    sequential_io_time = time.time() - start
    print(f"Sequential I/O time: {sequential_io_time:.2f}s\n")

    # Threaded I/O (fast!)
    start = time.time()
    threads = [
        threading.Thread(target=io_task, args=(1, 1)),
        threading.Thread(target=io_task, args=(2, 1))
    ]
    for t in threads:
        t.start()
    for t in threads:
        t.join()
    threaded_io_time = time.time() - start
    print(f"Threaded I/O time: {threaded_io_time:.2f}s")
```

```
    print(f"I/O speedup: {sequential_io_time / threaded_io_time:.1f}x\n")

    print("=== CPU Work: Sequential vs Threaded ===")

    # Sequential CPU
    start = time.time()
    cpu_task(1, 1_000_000)
    cpu_task(2, 1_000_000)
    sequential_cpu_time = time.time() - start
    print(f"Sequential CPU time: {sequential_cpu_time:.2f}s\n")

    # Threaded CPU (won't help much due to GIL)
    start = time.time()
    threads = [
        threading.Thread(target=cpu_task, args=(1, 1_000_000)),
        threading.Thread(target=cpu_task, args=(2, 1_000_000))
    ]
    for t in threads:
        t.start()
    for t in threads:
        t.join()
    threaded_cpu_time = time.time() - start
    print(f"Threaded CPU time: {threaded_cpu_time:.2f}s")
    print(f"CPU 'speedup': {sequential_cpu_time / threaded_cpu_time:.1f}x
(should be ~1.0)")

# Run the demonstration
demonstrate_io_vs_cpu_work()
```

**The Decision Tree - When to Use What:**

```
def concurrency_decision_helper(task_description):
    """Help decide between threads and processes."""

    io_keywords = ['download', 'upload', 'request', 'database', 'file',
'network', 'api', 'web']
    cpu_keywords = ['calculate', 'compute', 'process', 'analyze',
'transform', 'generate']

    description_lower = task_description.lower()

    print(f"Task: {task_description}")

    # Check for I/O indicators
    io_score = sum(1 for keyword in io_keywords if keyword in
description_lower)
    cpu_score = sum(1 for keyword in cpu_keywords if keyword in
description_lower)

    if io_score > cpu_score:
        print(" This looks I/O-bound -> Use THREADS")
        print("  Reason: Lots of waiting, GIL will be released during
```

```
I/O")
        print("  Tool: ThreadPoolExecutor")
    elif cpu_score > io_score:
        print(" This looks CPU-bound -> Use PROCESSES")
        print("  Reason: Heavy computation, need separate interpreters")
        print("  Tool: ProcessPoolExecutor")
    else:
        print("? Mixed or unclear -> Start with THREADS, profile, then
decide")
        print("  Reason: Threads are simpler; switch to processes if
needed")

    print()

# Test the decision helper
example_tasks = [
    "Download 100 web pages and save to files",
    "Resize 1000 images to create thumbnails",
    "Query database for user data and send emails",
    "Calculate prime numbers up to 1 million",
    "Fetch stock prices from 50 different APIs",
    "Apply machine learning model to analyze text data"
]

for task in example_tasks:
    concurrency_decision_helper(task)
```

**Key Mental Models:**

- **GIL = Single bathroom key**: Only one thread can use Python interpreter at a time
- **I/O releases the key**: Waiting operations let other threads work
- **Processes = Separate bathrooms**: Each has its own Python interpreter
- **Match the tool to the bottleneck**: Threads for waiting, processes for computing

---

**II. Thread Pools for I/O-Bound Tasks - "The Waiting Game" (15 min)**

**Mental Model: The Multi-Window Bank**

Imagine a bank with one teller (single-threaded) vs multiple tellers (thread pool). When customers need to wait for approvals or paperwork, the single teller sits idle. But with multiple tellers, while one waits, others can help new customers. That's exactly how thread pools help with I/O-bound tasks.

**Pattern 1: Basic Thread Pool for Web Requests**

```
import concurrent.futures
import time
import requests

def fetch_url_info(url):
    """Fetch URL and return timing info."""
```

```
        try:
            start_time = time.time()
            response = requests.get(url, timeout=5)
            end_time = time.time()

            return {
                'url': url,
                'status': response.status_code,
                'time': end_time - start_time,
                'size': len(response.content),
                'success': True
            }
        except Exception as e:
            return {
                'url': url,
                'error': str(e),
                'success': False
            }

    def download_sequential_vs_threaded():
        """Compare sequential vs threaded downloads."""

        # Test URLs (using httpbin for reliable testing)
        urls = [
            "https://httpbin.org/delay/1",    # 1 second delay
            "https://httpbin.org/delay/1",    # 1 second delay
            "https://httpbin.org/delay/1",    # 1 second delay
            "https://httpbin.org/json",       # Quick response
            "https://httpbin.org/uuid"        # Quick response
        ]

        print("=== Sequential Downloads (One at a Time) ===")
        start_time = time.time()
        sequential_results = []

        for url in urls:
            result = fetch_url_info(url)
            sequential_results.append(result)
            status = "" if result['success'] else ""
            print(f"{status} {url}")

        sequential_time = time.time() - start_time
        print(f"Sequential total: {sequential_time:.2f}s\n")

        print("=== Threaded Downloads (Multiple Workers) ===")
        start_time = time.time()
        threaded_results = []

        # The magic: ThreadPoolExecutor manages the threads for us
        with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
            # Submit all tasks at once
            future_to_url = {executor.submit(fetch_url_info, url): url for url
    in urls}
```

```
        # Collect results as they complete
        for future in concurrent.futures.as_completed(future_to_url):
            result = future.result()
            threaded_results.append(result)
            status = "" if result['success'] else ""
            print(f"{status} {result['url']}")

    threaded_time = time.time() - start_time
    print(f"Threaded total: {threaded_time:.2f}s")

    speedup = sequential_time / threaded_time if threaded_time > 0 else 0
    print(f"Speedup: {speedup:.1f}x faster!")

    return sequential_results, threaded_results

# Test it out (requires internet connection)
try:
    download_sequential_vs_threaded()
except Exception as e:
    print(f"Network test failed: {e}")
    print("That's okay - the concept is what matters!")
```

**Pattern 2: File Processing with Thread Pools**

```
import os
import time

def process_file_simulation(filename):
    """Simulate processing a file (reading, analyzing, etc.)."""
    print(f"Processing {filename}...")

    # Simulate I/O operations
    time.sleep(0.5)  # Simulate reading file
    time.sleep(0.3)  # Simulate processing
    time.sleep(0.2)  # Simulate writing results

    # Return processing results
    return {
        'filename': filename,
        'lines_processed': len(filename) * 10,  # Fake metric
        'processing_time': 1.0,
        'status': 'completed'
    }

def batch_file_processor(filenames, use_threads=False, max_workers=3):
    """Process multiple files sequentially or with threads."""

    start_time = time.time()
    results = []

    if use_threads:
        print(f"Processing {len(filenames)} files with {max_workers}
```

```
    threads...")

            with
    concurrent.futures.ThreadPoolExecutor(max_workers=max_workers) as
    executor:
                # Submit all file processing tasks
                future_to_file = {
                    executor.submit(process_file_simulation, filename):
    filename
                    for filename in filenames
                }

                # Collect results as they complete
                for future in concurrent.futures.as_completed(future_to_file):
                    result = future.result()
                    results.append(result)
                    print(f"   Completed {result['filename']}")

        else:
            print(f"Processing {len(filenames)} files sequentially...")

            for filename in filenames:
                result = process_file_simulation(filename)
                results.append(result)
                print(f"   Completed {result['filename']}")

        total_time = time.time() - start_time
        method = "threaded" if use_threads else "sequential"
        print(f"{method.title()} processing completed in {total_time:.2f}s\n")

        return results, total_time

    # Test with sample files
    sample_files = [
        "data_report_2024.txt",
        "user_logs_january.txt",
        "sales_summary.txt",
        "inventory_update.txt",
        "customer_feedback.txt"
    ]

    print("Comparing file processing approaches:")
    sequential_results, seq_time = batch_file_processor(sample_files,
    use_threads=False)
    threaded_results, thread_time = batch_file_processor(sample_files,
    use_threads=True, max_workers=3)

    speedup = seq_time / thread_time if thread_time > 0 else 0
    print(f"File processing speedup: {speedup:.1f}x")
```

**Pattern 3: The "Submit and Forget" Pattern**

```
def monitor_services():
    """Monitor multiple services concurrently."""

    def check_service_health(service_name, check_duration):
        """Simulate checking if a service is healthy."""
        print(f"Checking {service_name}...")
        time.sleep(check_duration)  # Simulate network request

        # Simulate random health status
        import random
        is_healthy = random.choice([True, True, True, False])  # Mostly
healthy

        return {
            'service': service_name,
            'healthy': is_healthy,
            'response_time': check_duration,
            'timestamp': time.time()
        }

    services = [
        ('Database Server', 0.8),
        ('Web API', 0.5),
        ('Cache Service', 0.3),
        ('File Storage', 1.0),
        ('Email Service', 0.7)
    ]

    print("Monitoring all services concurrently...")
    start_time = time.time()

    with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
        # Submit all health checks
        futures = [
            executor.submit(check_service_health, service, duration)
            for service, duration in services
        ]

        # Wait for all to complete and collect results
        results = []
        for future in concurrent.futures.as_completed(futures):
            result = future.result()
            results.append(result)

            status = " HEALTHY" if result['healthy'] else " UNHEALTHY"
            print(f"  {result['service']}: {status}
({result['response_time']:.1f}s)")

    total_time = time.time() - start_time
    print(f"\nAll service checks completed in {total_time:.2f}s")

    # Summary
    healthy_count = sum(1 for r in results if r['healthy'])
```

```
    print(f"Health Summary: {healthy_count}/{len(results)} services
healthy")

    return results

# Run service monitoring
service_status = monitor_services()
```

**Key Patterns for Thread Pools:**

- **Submit all at once**: Get all tasks started immediately
- **Collect as completed**: Process results as they finish
- **Use context managers**: `with ThreadPoolExecutor()` handles cleanup
- **Right-size workers**: Usually 2-5x your CPU cores for I/O tasks
- **Handle exceptions**: Always wrap worker functions in try/except

---

### III. Process Pools for CPU-Bound Tasks - "The Think Tank" (10 min)

#### Mental Model: The Research Team

Imagine you need to solve 4 complex math problems. You could:

- **One researcher (sequential)**: Solve them one by one (slow)
- **One researcher with helpers (threads)**: Still limited by one brain thinking at a time
- **Four researchers (processes)**: Each with their own brain, working simultaneously (fast!)

That's multiprocessing - separate Python interpreters, each with full thinking power.

#### Pattern 1: CPU-Intensive Work with Process Pools

```
import concurrent.futures
import math
import time

def heavy_computation(workload_size):
    """Simulate CPU-intensive work like data analysis or image
processing."""
    print(f"Starting computation with {workload_size:,} items...")

    # Simulate heavy mathematical computation
    total = 0
    for i in range(workload_size):
        # Complex calculations (pure Python, no I/O)
        value = math.sin(i) * math.cos(i) + math.sqrt(i + 1)
        total += value * math.log(i + 1)

    print(f"Completed computation: {workload_size:,} items")
    return {
        'workload_size': workload_size,
        'result': total,
```

```
        'computed_by': 'CPU worker'
    }

def compare_sequential_vs_multiprocess():
    """Show the power of multiprocessing for CPU work."""

    # Each task is a chunk of work
    work_chunks = [500_000, 500_000, 500_000, 500_000]  # 4 chunks

    print("=== Sequential Processing (One Brain) ===")
    start_time = time.time()
    sequential_results = []

    for i, chunk_size in enumerate(work_chunks, 1):
        result = heavy_computation(chunk_size)
        sequential_results.append(result)
        print(f"  Chunk {i} completed")

    sequential_time = time.time() - start_time
    print(f"Sequential total: {sequential_time:.2f}s\n")

    print("=== Multiprocess Processing (Multiple Brains) ===")
    start_time = time.time()
    multiprocess_results = []

    with concurrent.futures.ProcessPoolExecutor() as executor:
        # Submit all chunks simultaneously
        futures = [executor.submit(heavy_computation, chunk_size)
                   for chunk_size in work_chunks]

        # Collect results as they complete
        for i, future in
enumerate(concurrent.futures.as_completed(futures), 1):
            result = future.result()
            multiprocess_results.append(result)
            print(f"  Chunk {i} completed")

    multiprocess_time = time.time() - start_time
    print(f"Multiprocess total: {multiprocess_time:.2f}s")

    speedup = sequential_time / multiprocess_time if multiprocess_time > 0
else 0
    print(f"Speedup: {speedup:.1f}x faster with multiple processes!")

    return sequential_results, multiprocess_results

# Run the comparison
sequential_results, process_results = compare_sequential_vs_multiprocess()
```

**Pattern 2: Data Processing Pipeline**

```
def process_data_chunk(data_info):
    """Process a chunk of data (like analyzing a CSV section)."""
    chunk_id, start_value, size = data_info

    print(f"Processing chunk {chunk_id} ({size:,} items)...")

    # Simulate data processing: filtering, transforming, aggregating
    processed_items = []
    total_sum = 0

    for i in range(size):
        value = start_value + i
        # Simulate complex data transformations
        processed_value = value ** 2 + math.sin(value) * 100
        processed_items.append(processed_value)
        total_sum += processed_value

    result = {
        'chunk_id': chunk_id,
        'items_processed': len(processed_items),
        'sum': total_sum,
        'average': total_sum / len(processed_items),
        'min_value': min(processed_items),
        'max_value': max(processed_items)
    }

    print(f"Chunk {chunk_id} complete: {result['items_processed']:,} items
processed")
    return result

def parallel_data_processing():
    """Process large dataset in parallel chunks."""

    # Simulate dividing a large dataset into chunks
    data_chunks = [
        (1, 0, 100_000),      # chunk_id, start_value, size
        (2, 100_000, 100_000),
        (3, 200_000, 100_000),
        (4, 300_000, 100_000),
        (5, 400_000, 100_000)
    ]

    print(f"Processing {len(data_chunks)} data chunks in parallel...")
    start_time = time.time()

    with concurrent.futures.ProcessPoolExecutor(max_workers=4) as
executor:
        # Submit all chunks for processing
        future_to_chunk = {
            executor.submit(process_data_chunk, chunk_info): chunk_info[0]
            for chunk_info in data_chunks
        }
```

```
        # Collect and combine results
        all_results = []
        for future in concurrent.futures.as_completed(future_to_chunk):
            chunk_id = future_to_chunk[future]
            try:
                result = future.result()
                all_results.append(result)
            except Exception as e:
                print(f"Chunk {chunk_id} failed: {e}")

    processing_time = time.time() - start_time

    # Combine results from all chunks
    total_items = sum(r['items_processed'] for r in all_results)
    total_sum = sum(r['sum'] for r in all_results)
    overall_average = total_sum / total_items if total_items > 0 else 0

    print(f"\n=== Processing Complete ===")
    print(f"Total time: {processing_time:.2f}s")
    print(f"Total items processed: {total_items:,}")
    print(f"Overall average: {overall_average:.2f}")
    print(f"Processing rate: {total_items/processing_time:,.0f}
items/second")

    return all_results

# Run parallel data processing
processing_results = parallel_data_processing()
```

**When NOT to Use Multiprocessing:**

```
def small_task_demonstration():
    """Show when multiprocessing overhead isn't worth it."""

    def tiny_computation(n):
        """Very small computation - overhead will dominate."""
        return sum(range(n))

    small_tasks = [1000] * 10  # 10 tiny tasks

    print("=== Small Tasks: Sequential vs Multiprocess ===")

    # Sequential
    start = time.time()
    seq_results = [tiny_computation(n) for n in small_tasks]
    seq_time = time.time() - start
    print(f"Sequential: {seq_time:.4f}s")

    # Multiprocess (will be slower due to overhead!)
    start = time.time()
    with concurrent.futures.ProcessPoolExecutor() as executor:
        proc_results = list(executor.map(tiny_computation, small_tasks))
```

```
    proc_time = time.time() - start
    print(f"Multiprocess: {proc_time:.4f}s")

    print(f"Overhead cost: {proc_time/seq_time:.1f}x slower!")
    print("Lesson: Use multiprocessing for substantial work, not tiny
tasks")

small_task_demonstration()
```

**Key Principles for Process Pools:**

- **Substantial work only**: Process creation has overhead
- **CPU-bound tasks**: Where computation is the bottleneck
- **Independent chunks**: Tasks that don't need to share much data
- **Right-size workers**: Usually equal to your CPU core count
- **Clean data**: Pass simple data types, not complex objects

---

**IV. Simple Queue Coordination - "The Assembly Line" (15 min)**

**Mental Model: The Restaurant Kitchen**

In a busy restaurant kitchen:

- **Orders come in** (producer) **Order queue**
- **Cooks prepare food** (workers) **Ready queue**
- **Servers deliver** (consumer) **Happy customers**

Queues coordinate work between different stages, ensuring smooth flow even when stages work at different speeds.

**Pattern 1: Basic Producer-Consumer with Queues**

```
import queue
import threading
import time
import random

def simple_producer_consumer():
    """Demonstrate basic producer-consumer pattern."""

    # The "conveyor belt" between producer and consumer
    work_queue = queue.Queue(maxsize=5)  # Limit queue size to prevent
overload

    def producer(name, items_to_produce):
        """Produces work items (like taking orders)."""
        print(f"Producer {name} starting...")

        for i in range(items_to_produce):
            # Create work item
```

```
            work_item = f"{name}-item-{i+1}"

            # Put it on the queue (will block if queue is full)
            work_queue.put(work_item)
            print(f" Produced: {work_item}")

            # Simulate time between productions
            time.sleep(random.uniform(0.1, 0.3))

        print(f"Producer {name} finished")

    def consumer(name):
        """Consumes work items (like fulfilling orders)."""
        print(f"Consumer {name} starting...")
        processed_count = 0

        while True:
            try:
                # Get work from queue (wait up to 2 seconds)
                work_item = work_queue.get(timeout=2)

                # Process the work
                print(f" {name} processing: {work_item}")
                time.sleep(random.uniform(0.2, 0.5))  # Simulate work time

                # Mark task as done
                work_queue.task_done()
                processed_count += 1

            except queue.Empty:
                # No more work available
                print(f"Consumer {name} finished ({processed_count} items
processed)")
                break

    # Start producer and consumer threads
    producer_thread = threading.Thread(target=producer, args=
("OrderTaker", 8))
    consumer_thread = threading.Thread(target=consumer, args=("Worker"))

    producer_thread.start()
    consumer_thread.start()

    # Wait for both to complete
    producer_thread.join()
    consumer_thread.join()

    print("All work completed!")

print("=== Basic Producer-Consumer Example ===")
simple_producer_consumer()
```

**Pattern 2: Multiple Workers Sharing Work**

```
def multiple_workers_example():
    """Multiple workers processing from shared queue."""

    task_queue = queue.Queue()
    results_queue = queue.Queue()

    def work_generator(num_tasks):
        """Generate work tasks."""
        print(f"Generating {num_tasks} tasks...")

        for i in range(num_tasks):
            task = {
                'id': i+1,
                'data': f"task_data_{i+1}",
                'complexity': random.randint(1, 5)  # 1=easy, 5=hard
            }
            task_queue.put(task)
            print(f" Generated task {task['id']}")

        print("All tasks generated")

    def worker(worker_id):
        """Worker that processes tasks from the shared queue."""
        print(f"Worker {worker_id} ready for work")
        tasks_completed = 0

        while True:
            try:
                # Get next available task
                task = task_queue.get(timeout=1)

                # Process the task
                print(f" Worker {worker_id} working on task {task['id']}")
                processing_time = task['complexity'] * 0.2  # Harder tasks
take longer
                time.sleep(processing_time)

                # Create result
                result = {
                    'task_id': task['id'],
                    'worker_id': worker_id,
                    'result': f"processed_{task['data']}",
                    'processing_time': processing_time
                }

                results_queue.put(result)
                task_queue.task_done()
                tasks_completed += 1

                print(f" Worker {worker_id} completed task {task['id']}")

            except queue.Empty:
```

```
                # No more tasks available
                print(f"Worker {worker_id} finished ({tasks_completed}
tasks)")
                break

    def results_collector():
        """Collect and summarize results."""
        print("Results collector starting...")
        all_results = []

        while True:
            try:
                result = results_queue.get(timeout=2)
                all_results.append(result)
                print(f" Collected result for task {result['task_id']}")
            except queue.Empty:
                break

        # Summary
        print(f"\n=== Work Summary ===")
        print(f"Total tasks completed: {len(all_results)}")

        by_worker = {}
        total_time = 0
        for result in all_results:
            worker = result['worker_id']
            by_worker[worker] = by_worker.get(worker, 0) + 1
            total_time += result['processing_time']

        for worker, count in by_worker.items():
            print(f"Worker {worker}: {count} tasks")

        print(f"Total processing time: {total_time:.2f}s")
        return all_results

    # Start the work system
    num_tasks = 12
    num_workers = 3

    # Start results collector
    collector_thread = threading.Thread(target=results_collector)
    collector_thread.start()

    # Start workers
    worker_threads = []
    for i in range(num_workers):
        worker_thread = threading.Thread(target=worker, args=(f"W{i+1}",))
        worker_thread.start()
        worker_threads.append(worker_thread)

    # Generate work
    generator_thread = threading.Thread(target=work_generator, args=
(num_tasks,))
    generator_thread.start()
```

```
    # Wait for everything to complete
    generator_thread.join()

    # Wait for all tasks to be processed
    task_queue.join()

    # Wait for workers and collector
    for worker_thread in worker_threads:
        worker_thread.join()

    collector_thread.join()

    print("Multiple workers example completed!")

print("\n=== Multiple Workers Example ===")
multiple_workers_example()
```

**Pattern 3: Pipeline Processing (Assembly Line)**

```
def processing_pipeline():
    """Multi-stage processing pipeline."""

    # Three queues for three stages
    raw_queue = queue.Queue(maxsize=3)      # Raw materials
    processed_queue = queue.Queue(maxsize=3) # Partially processed
    finished_queue = queue.Queue()           # Final products

    def stage1_prepare(worker_id):
        """Stage 1: Prepare raw materials."""
        print(f"Stage1 Worker {worker_id} ready")

        while True:
            try:
                raw_item = raw_queue.get(timeout=1)
                if raw_item is None:  # Shutdown signal
                    processed_queue.put(None)  # Pass shutdown signal
                    break

                print(f" Stage1-{worker_id}: Preparing {raw_item}")
                time.sleep(0.3)  # Preparation time

                prepared_item = {
                    'original': raw_item,
                    'stage1_done': True,
                    'prepared_by': f"stage1_{worker_id}"
                }

                processed_queue.put(prepared_item)
                raw_queue.task_done()

            except queue.Empty:
```

```
                break

        print(f"Stage1 Worker {worker_id} finished")

    def stage2_assemble(worker_id):
        """Stage 2: Assemble prepared items."""
        print(f"Stage2 Worker {worker_id} ready")

        while True:
            try:
                prepared_item = processed_queue.get(timeout=1)
                if prepared_item is None:  # Shutdown signal
                    finished_queue.put(None)  # Pass shutdown signal
                    break

                print(f" Stage2-{worker_id}: Assembling
{prepared_item['original']}")
                time.sleep(0.4)  # Assembly time

                assembled_item = {
                    **prepared_item,
                    'stage2_done': True,
                    'assembled_by': f"stage2_{worker_id}"
                }

                finished_queue.put(assembled_item)
                processed_queue.task_done()

            except queue.Empty:
                break

        print(f"Stage2 Worker {worker_id} finished")

    def quality_control():
        """Final stage: Quality control and packaging."""
        print("Quality Control ready")
        final_products = []

        while True:
            try:
                finished_item = finished_queue.get(timeout=1)
                if finished_item is None:  # Shutdown signal
                    break

                print(f" QC: Inspecting {finished_item['original']}")
                time.sleep(0.2)  # Inspection time

                final_product = {
                    **finished_item,
                    'quality_approved': True,
                    'completed_at': time.time()
                }

                final_products.append(final_product)
```

```
                print(f" QC: Approved {finished_item['original']}")

            except queue.Empty:
                break

        print(f"Quality Control finished: {len(final_products)} products
approved")
        return final_products

    # Start the pipeline
    raw_materials = [f"Item-{i:02d}" for i in range(1, 9)]  # 8 items

    # Start workers for each stage
    stage1_worker = threading.Thread(target=stage1_prepare, args=(1,))
    stage2_worker = threading.Thread(target=stage2_assemble, args=(1,))
    qc_worker = threading.Thread(target=quality_control)

    stage1_worker.start()
    stage2_worker.start()
    qc_worker.start()

    # Feed raw materials into the pipeline
    print(" Starting production pipeline...")
    for item in raw_materials:
        raw_queue.put(item)
        print(f" Added {item} to production")
        time.sleep(0.1)

    # Signal end of input
    raw_queue.put(None)

    # Wait for pipeline to complete
    stage1_worker.join()
    stage2_worker.join()
    qc_worker.join()

    print(" Pipeline completed!")

print("\n=== Processing Pipeline Example ===")
processing_pipeline()
```

**Key Queue Principles:**

- **Decouple stages**: Producers and consumers work independently
- **Buffer variations**: Queues smooth out speed differences
- **Backpressure**: Limited queue sizes prevent memory issues
- **Clean shutdown**: Use sentinel values (like `None`) to stop workers
- **Error handling**: Always use timeouts and handle exceptions

---

**V. Recap & Practical Guidelines (10 min)**

**The Big Picture: Choosing Your Concurrency Strategy**

Think of concurrency as choosing the right vehicle for your journey:

- **Walking (sequential)**: Simple, reliable, but slow for long distances
- **Bicycle (threads)**: Great when you need to stop frequently (I/O waits)
- **Car (processes)**: Powerful for long stretches of highway (CPU work)
- **Bus (queues)**: Coordinate multiple vehicles working together

**Decision Framework - Start Here:**

```
def choose_concurrency_approach(task_description, expected_duration,
data_size):
    """Help choose the right concurrency approach."""

    print(f"Task: {task_description}")
    print(f"Expected duration: {expected_duration}s")
    print(f"Data size: {data_size}")
    print()

    # Quick filters
    if expected_duration < 0.1:
        print(" RECOMMENDATION: Stay sequential")
        print("   Reason: Task too small, concurrency overhead not worth
it")
        return "sequential"

    # Check for I/O indicators
    io_indicators = ['download', 'upload', 'request', 'database', 'file',
'api', 'network']
    cpu_indicators = ['calculate', 'compute', 'process', 'analyze',
'transform']

    task_lower = task_description.lower()
    is_io_bound = any(word in task_lower for word in io_indicators)
    is_cpu_bound = any(word in task_lower for word in cpu_indicators)

    if is_io_bound and not is_cpu_bound:
        print(" RECOMMENDATION: Use ThreadPoolExecutor")
        print("   Reason: I/O-bound work benefits from threads")
        print("   Workers: 2-5x CPU cores")
        return "threads"

    elif is_cpu_bound and not is_io_bound:
        if data_size == "large":
            print("  RECOMMENDATION: Use ProcessPoolExecutor with
caution")
            print("   Reason: CPU-bound but large data = expensive
copying")
            print("   Workers: Equal to CPU cores")
        else:
            print(" RECOMMENDATION: Use ProcessPoolExecutor")
```

```
            print("   Reason: CPU-bound work needs separate interpreters")
            print("   Workers: Equal to CPU cores")
        return "processes"

    else:
        print(" RECOMMENDATION: Start with threads, profile, then decide")
        print("   Reason: Mixed or unclear workload")
        print("   Strategy: Measure first, optimize second")
        return "threads_first"

# Test the decision framework
test_scenarios = [
    ("Download 50 web pages", 2.0, "small"),
    ("Resize 1000 images", 30.0, "medium"),
    ("Process 10GB CSV file", 60.0, "large"),
    ("Calculate prime numbers", 45.0, "small"),
    ("Quick database lookup", 0.05, "tiny")
]

for scenario in test_scenarios:
    choose_concurrency_approach(*scenario)
    print("-" * 50)
```

**Essential Patterns Summary:**

```
# Pattern 1: I/O-Bound Tasks (Use ThreadPoolExecutor)
import concurrent.futures

def io_bound_pattern(tasks):
    """Template for I/O-bound work."""
    with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
        # Submit all tasks
        futures = [executor.submit(task_function, task) for task in tasks]

        # Collect results as they complete
        results = []
        for future in concurrent.futures.as_completed(futures):
            try:
                result = future.result()
                results.append(result)
            except Exception as e:
                print(f"Task failed: {e}")

    return results

# Pattern 2: CPU-Bound Tasks (Use ProcessPoolExecutor)
def cpu_bound_pattern(work_chunks):
    """Template for CPU-bound work."""
    with concurrent.futures.ProcessPoolExecutor() as executor:
        # Submit all chunks
        futures = [executor.submit(cpu_function, chunk) for chunk in
work_chunks]
```

```
        # Collect results
        results = []
        for future in concurrent.futures.as_completed(futures):
            try:
                result = future.result()
                results.append(result)
            except Exception as e:
                print(f"Chunk failed: {e}")

    return results

# Pattern 3: Producer-Consumer (Use Queues)
import queue
import threading

def producer_consumer_pattern(items_to_process):
    """Template for coordinated work."""
    work_queue = queue.Queue()
    result_queue = queue.Queue()

    def worker():
        while True:
            try:
                item = work_queue.get(timeout=1)
                result = process_item(item)
                result_queue.put(result)
                work_queue.task_done()
            except queue.Empty:
                break

    # Start workers
    workers = [threading.Thread(target=worker) for _ in range(3)]
    for w in workers:
        w.start()

    # Add work
    for item in items_to_process:
        work_queue.put(item)

    # Wait and collect
    work_queue.join()
    results = []
    while not result_queue.empty():
        results.append(result_queue.get())

    return results
```

**Common Gotchas and Solutions:**

```
#  DON'T: Use threads for heavy CPU work
def bad_cpu_threading():
```

```
    with concurrent.futures.ThreadPoolExecutor() as executor:
        # This won't help due to GIL!
        futures = [executor.submit(heavy_math, data) for data in chunks]

#  DO: Use processes for heavy CPU work
def good_cpu_processing():
    with concurrent.futures.ProcessPoolExecutor() as executor:
        futures = [executor.submit(heavy_math, data) for data in chunks]

#  DON'T: Use processes for tiny tasks
def bad_tiny_multiprocessing():
    with concurrent.futures.ProcessPoolExecutor() as executor:
        # Overhead will dominate!
        futures = [executor.submit(lambda x: x*2, i) for i in range(100)]

#  DO: Batch tiny tasks together
def good_batched_processing():
    def process_batch(batch):
        return [x*2 for x in batch]

    batches = [list(range(i, i+25)) for i in range(0, 100, 25)]
    with concurrent.futures.ProcessPoolExecutor() as executor:
        futures = [executor.submit(process_batch, batch) for batch in
batches]
```

**Performance Rules of Thumb:**

- **Thread workers**: 2-5x your CPU cores for I/O-bound tasks
- **Process workers**: Equal to your CPU cores for CPU-bound tasks
- **Queue sizes**: 1-3x your worker count to prevent memory bloat
- **Task granularity**: Aim for 0.1-1 second per task for good balance
- **Error handling**: Always wrap worker functions in try/except

**Final Decision Tree:**

1. **Is it worth parallelizing?** (>0.1s total work) If no, stay sequential
2. **What's the bottleneck?** I/O waiting or CPU thinking?
3. **I/O-bound** ThreadPoolExecutor
4. **CPU-bound + small data** ProcessPoolExecutor
5. **CPU-bound + large data** Consider alternatives or chunking
6. **Complex coordination needed** Add queues
7. **Mixed workload** Start with threads, profile, adjust

**Remember: Simple is better than complex. Start with the simplest solution that works, then optimize only when you have real performance problems.**

**Final Practice Question:**

You need to process 1000 images (resize each one). Each image takes 0.5 seconds to process. What's the best approach?

A. Sequential processing - simple and reliable B. ThreadPoolExecutor - images are I/O operations
C. ProcessPoolExecutor - image processing is CPU-intensive D. Queues with multiple stages

(Answer: C. ProcessPoolExecutor - Image resizing is CPU-intensive work that benefits from multiple processes, even though it involves file I/O. The actual computation (resizing) is the bottleneck, not the file reading.)