perplexity

Project Overview: Resource-Aware, Priority-Based Parallel File Processor in Java

This implementation is aimed at efficiently migrating legacy file processing ETL jobs—originally in Talend—to a highly maintainable, scalable, and **resource-aware Java application**. The core goals are:

- **Process multiple large files in parallel**, honoring task priority.
- **Dynamically adapt to underlying machine limits** (CPU, RAM) by tuning thread and chunk sizes in real time.
- **Split each file into chunks** for parallel processing, with safe error handling and robust retry logic.
- **Enable prioritized intake of file tasks**, maximizing throughput without overcommitting system resources.

What This Implementation Does

- **Accepts file processing requests** with user-defined priority.
- **Queues file tasks** in a thread-safe, priority-based manner (higher priority processed first).
- **Splits each file into manageable chunks** for concurrent, parallel processing.
- **Dynamically adapts** chunk size and thread pool size based on real-time JVM heap and CPU usage.
- **Retries failed chunk operations**, ensuring robustness and resilience for unreliable disk/network operations.
- **Prints live diagnostic status** (heap, CPU, concurrency) for transparent monitoring.
- **Fully documented codebase**—ultra-detailed comments enable fast onboarding for developers of any level.

Approaches and Best Practices Used

1. Priority-Based Scheduling

- Uses Java's `PriorityBlockingQueue` to enforce task priorities for input files.
- Each file is assigned a priority; higher priority jobs are picked first for chunking and processing.

2. Adaptive Resource Management

- JVM heap memory and system CPU load are measured every few seconds.
- **Thread pool size** is increased or decreased according to available CPU, within user-specified bounds.
- **Chunk size** is increased if memory is ample, decreased to avoid `OutOfMemoryError` if JVM heap is tight.

3. Chunked Parallel Processing

- Each file is split into logical data "chunks."
- Each chunk is processed independently using a thread pool.
- Chunks are sized to not overload RAM or saturate I/O.

4. Retry and Fault Tolerance

- If a chunk fails (e.g., I/O exception), it is retried up to a configured limit, with exponential back-off.
- Permanent chunk failures are logged for operational troubleshooting.

5. Separation of Concerns and Extensibility

- Codebase is modular: the logic for adapting resources, handling files, handling chunks, and main orchestration are all cleanly separable.
- All methods and fields are **meticulously documented**, allowing rapid handover to new team members.

Workflow Diagram

Below is a detailed workflow diagram for this implementation:

(The diagram illustrates: task ingestion, priority queue, main processing worker, dynamic resource monitor, thread pool, chunk task generation, adaptive controls, retry logic, and status output.)

Example Use Case Documentation for Onboarding

What Problem Does This Solve?

- You need to replace Talend or similar ETL systems with Java-based infrastructure for file input/output.
- Your files may be too large or too numerous to process sequentially, but you don't want to overwhelm the machine's RAM or CPU.
- Some files are more important/urgent than others, requiring prioritized processing.

- You need a maintainable, single-node implementation, but want to get the most performance from available resources.

How Do We Solve It?

1. **Job requests** are submitted with priority.
2. **Resource monitor** adapts to live system constraints:
    - Scales up/down processing threads to use available CPU.
    - Scales up/down chunk size to use available RAM.
3. **Main processing worker**:
    - Grabs highest-priority file job.
    - Splits into chunks (adjusted for memory, CPU).
    - Dispatches chunk tasks to thread pool; waits for completion.
4. **Chunk tasks**:
    - Read and process their segment.
    - Retry on failure up to the configured limit.
    - Report stats/log errors on completion or failure.
5. **Status reports** print heap usage, CPU utilization, your current adaptive settings, and file-chunk completions.

Why This Approach?

- Approximates Talend's "automatic parallelization" on a single node.
- Prevents bottlenecks and crashes from memory overuse.
- Always matches system resource usage to what's actually available instead of fixed, hard-coded allocations.
- Simple Java codebase, heavily commented, ready for quick modification, debugging, and extension.

How to Get Started?

1. **Clone the code**; save as `AdaptivePriorityFileProcessor.java`.
2. **Edit the main method** to register your real file paths and custom priorities.
3. **Configure** your minimum/maximum threads and chunk sizes (or use defaults).
4. **Adjust business logic** in `processChunk()` to handle transformation, parsing, etc.
5. **Run as a standalone app**—the included console outputs will show you live system status, performance, and error handling.

Additional Notes

- Works well as a single-node alternative to Talend or Apache Spark when simple parallelization/priority is enough.
- Easily portable to an API or enterprise ETL pipeline with minimal rework.
- New developers can learn from the **ultra-detailed comments** on every single line and quickly adapt to changing requirements.

Use this documentation and diagram in your onboarding wiki, code reviews, or technical sessions to help new team members or stakeholders understand, extend, and debug the file processing module right away.