CS 330: Operating Systems

Project Report: Printer Pool Simulation

Submission Date: Nov 15, 2025

1. Implementation and Synchronization Correctness

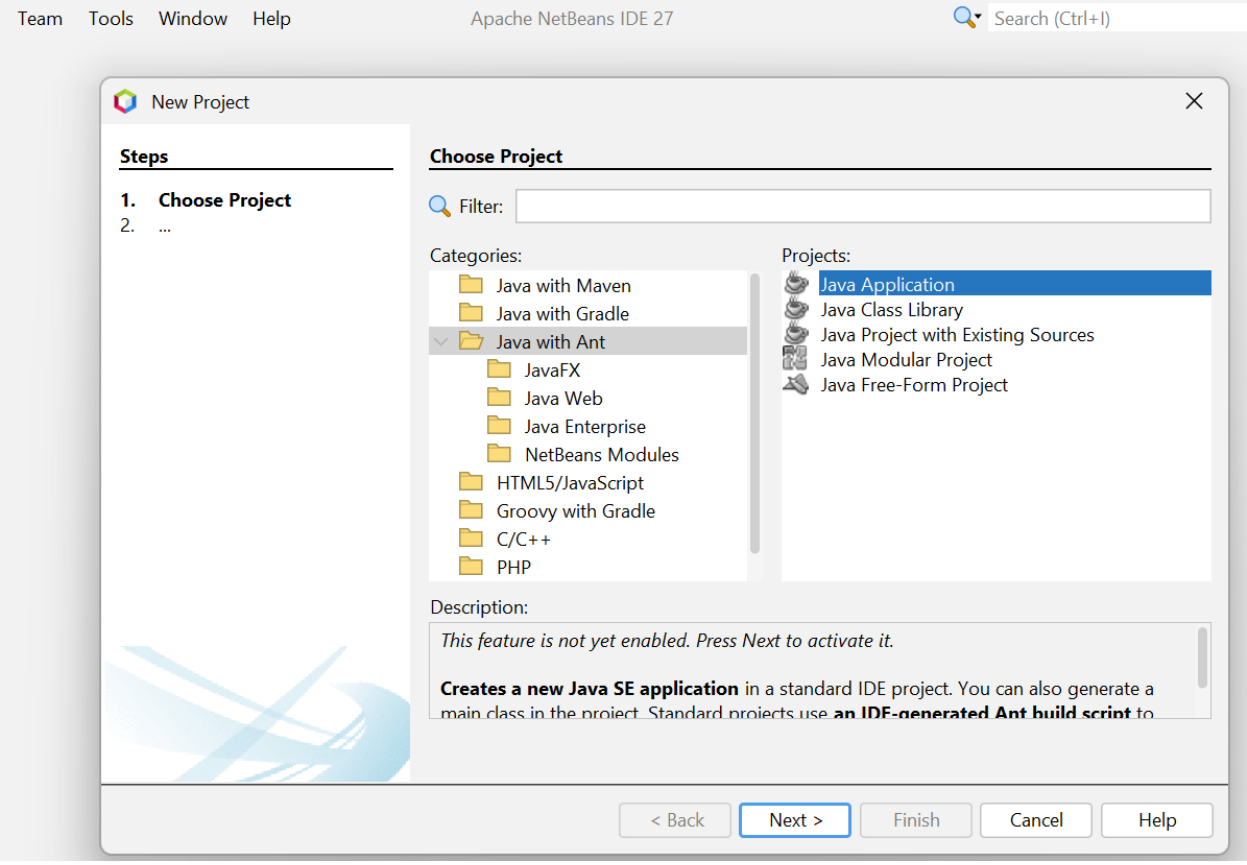
This project was implemented in a Java (JDK 21) environment using Apache NetBeans 27. The solution maps the concepts of C/POSIX primitives to Java's java.util.concurrent package.

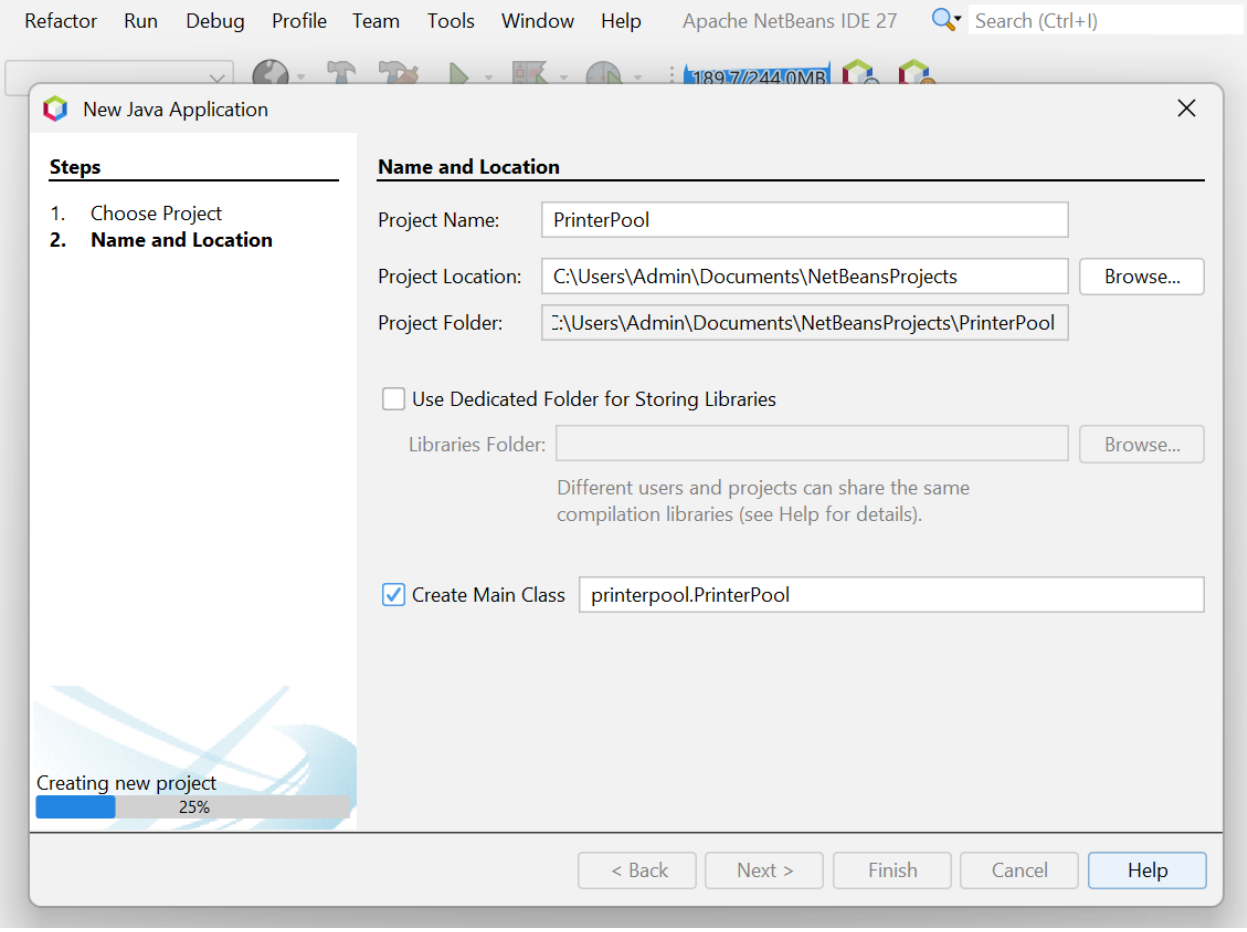
- **Student Threads (Producers):** A Runnable class that creates jobs.
- **Printer Threads (Consumers):** A Runnable class that processes jobs.
- **Queue (Shared Resource):** A custom bounded circular buffer.

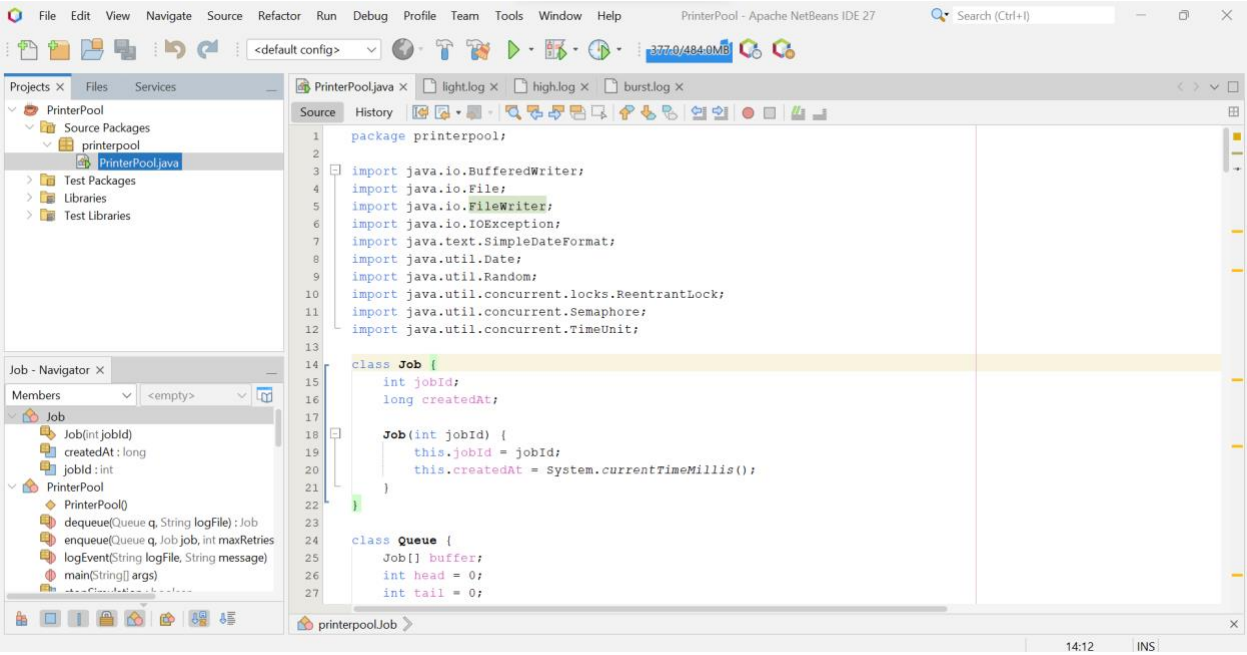
The following primitives were used to achieve synchronization:

1. **ReentrantLock (Mutex):** A ReentrantLock (equivalent to pthread_mutex_t) is used to protect the critical sections of the Queue (the enqueue and dequeue functions). This lock ensures that only one thread can modify the queue's internal array (in/out pointers) at any given time, preventing race conditions.
2. **Semaphore filledSlots:** This is a counting semaphore that tracks the number of jobs currently in the queue (number of filled slots). Printers (Consumers) call acquire() on this semaphore before taking a job. If the count is 0 (queue is empty), the Printer thread blocks (sleeps).
3. **Semaphore emptySlots:** This semaphore tracks the number of available empty slots in the queue. Students (Producers) call tryAcquire() on it before submitting a job. The use of tryAcquire() fulfills the project's "retry later" (non-blocking) requirement. If it returns false (meaning the queue is full), the Student thread enters its "retry" logic.

This implementation is a classic solution to the Producer-Consumer problem, and the logs (analyzed below) prove this synchronization strategy is effective.


Team
Tools
Window
Help
Apache NetBeans IDE 27
Search (Ctrl+I)
New Project
Steps
1. Choose Project
2. ...
Choose Project
Filter:
Categories:
Java with Maven
Java with Gradle
Java with Ant
JavaFX
Java Web
Java Enterprise
NetBeans Modules
HTML5/JavaScript
Groovy with Gradle
C/C++
PHP
Projects:
Java Application
Java Class Library
Java Project with Existing Sources
Java Modular Project
Java Free-Form Project
Description:
This feature is not yet enabled. Press Next to activate it.
Creates a new Java SE application in a standard IDE project. You can also generate a
< Back
Next >
Finish
Cancel
Help

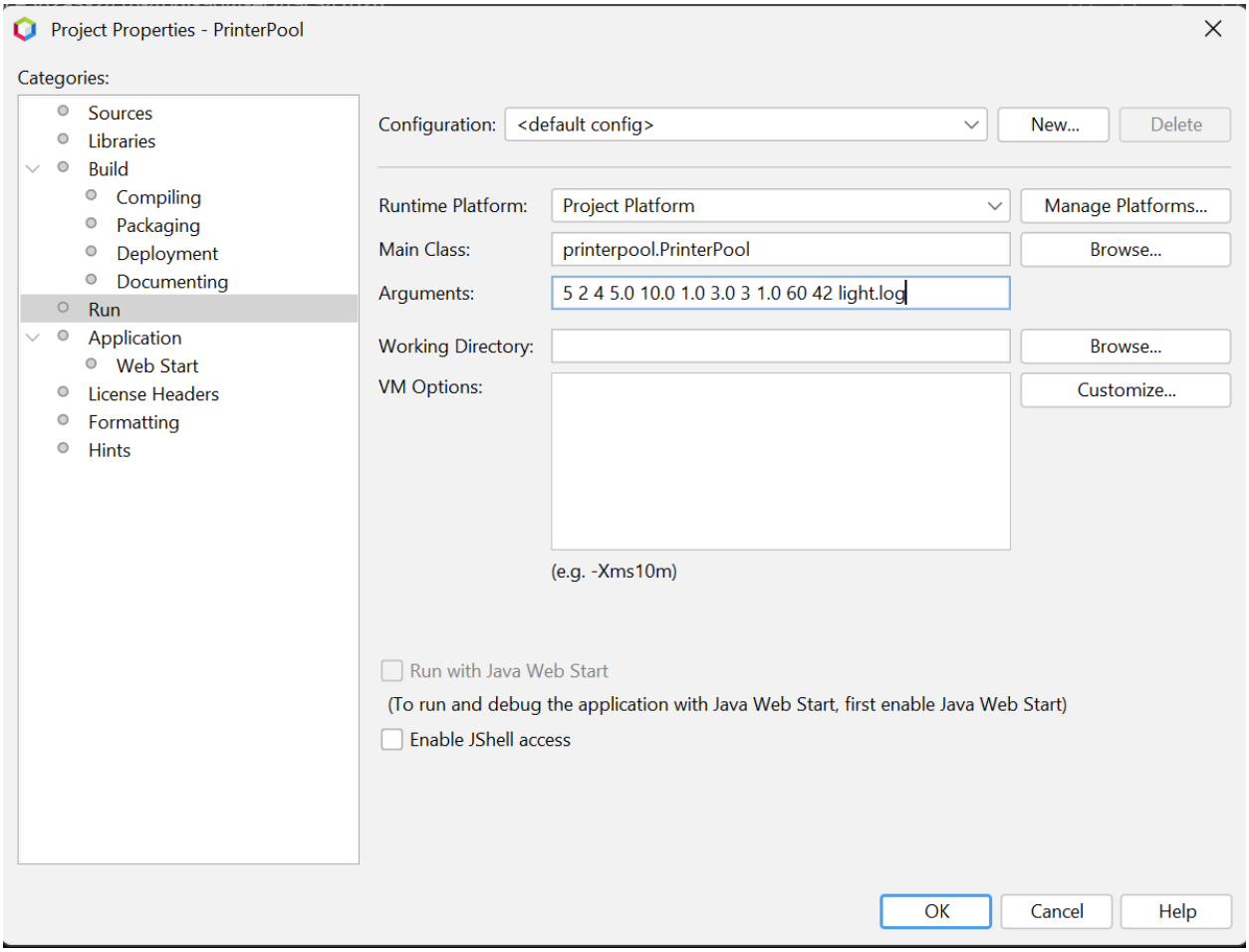

Refactor Run Debug Profile Team Tools Window Help
Apache NetBeans IDE 27
Search (Ctrl+I)
New Java Application
Steps
1. Choose Project
2. Name and Location
Name and Location
Project Name: PrinterPool
Project Location: C:\Users\Admin\Documents\NetBeansProjects
Browse...
Project Folder: C:\Users\Admin\Documents\NetBeansProjects\PrinterPool
Use Dedicated Folder for Storing Libraries
Libraries Folder:
Browse...
Different users and projects can share the same compilation libraries (see Help for details).
Create Main Class printerpool.PrinterPool
Creating new project
25%
< Back
Next >
Finish
Cancel
Help

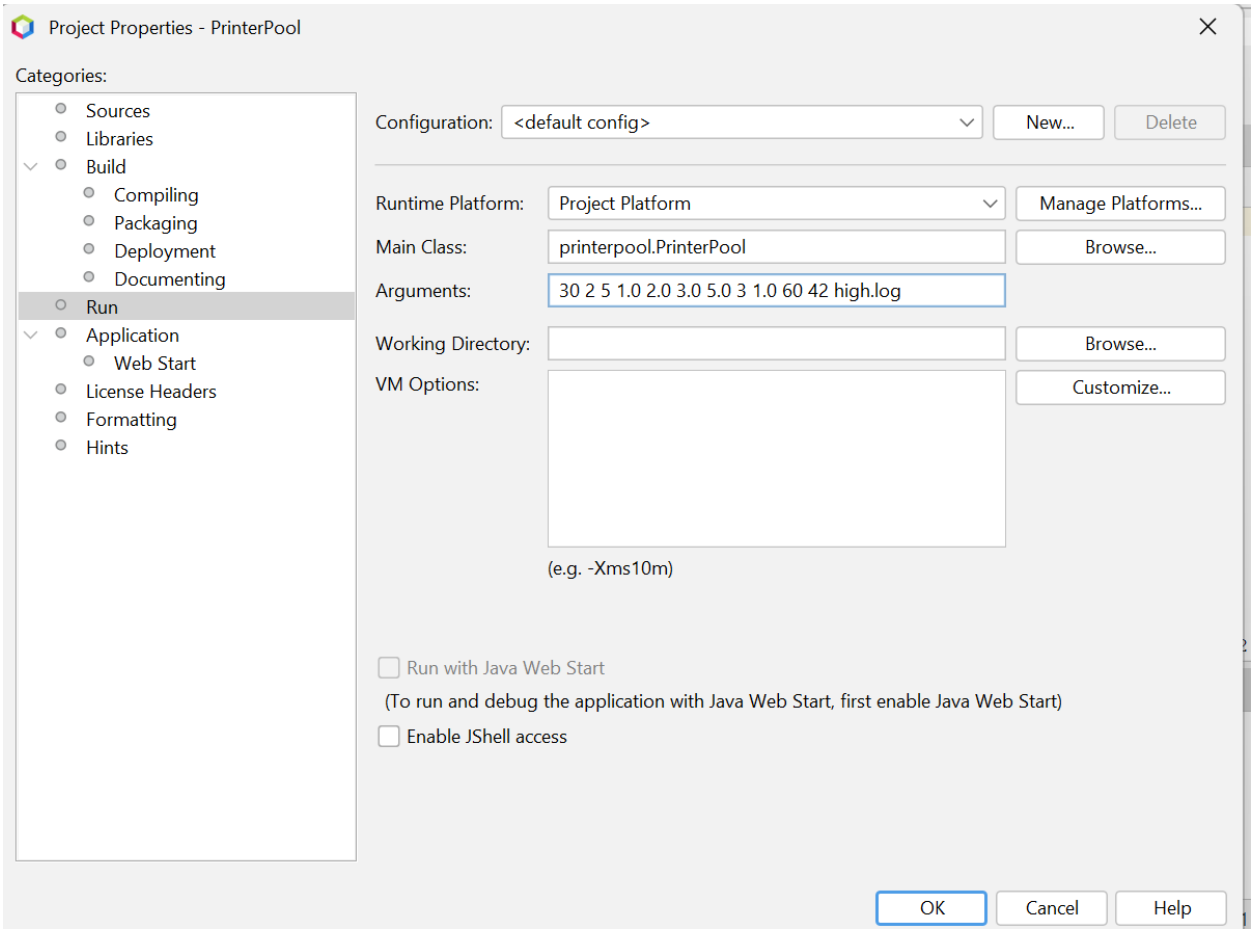

File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
PrinterPool - Apache NetBeans IDE 27
Search (Ctrl+I)
Projects
Files
Services
PrinterPool
Source Packages
printerpool
PrinterPool.java
Test Packages
Libraries
Test Libraries
PrinterPool.java
light.log
high.log
burst.log
Source
History
package printerpool;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Random;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.Semaphore;
import java.util.concurrent.TimeUnit;
class Job {
int jobId;
long createdAt;
Job(int jobId) {
this.jobId = jobId;
this.createdAt = System.currentTimeMillis();
}
}
class Queue {
Job[] buffer;
int head = 0;
int tail = 0;
Job - Navigator
Members
Job
Job(int jobId)
createdAt : long
jobId : int
PrinterPool
PrinterPool()
dequeue(Queue q, String logFile) : Job
enqueue(Queue q, Job job, int maxRetries)
logEvent(String logFile, String message)
main(String[] args)
printerpool.Job
14:12
INS

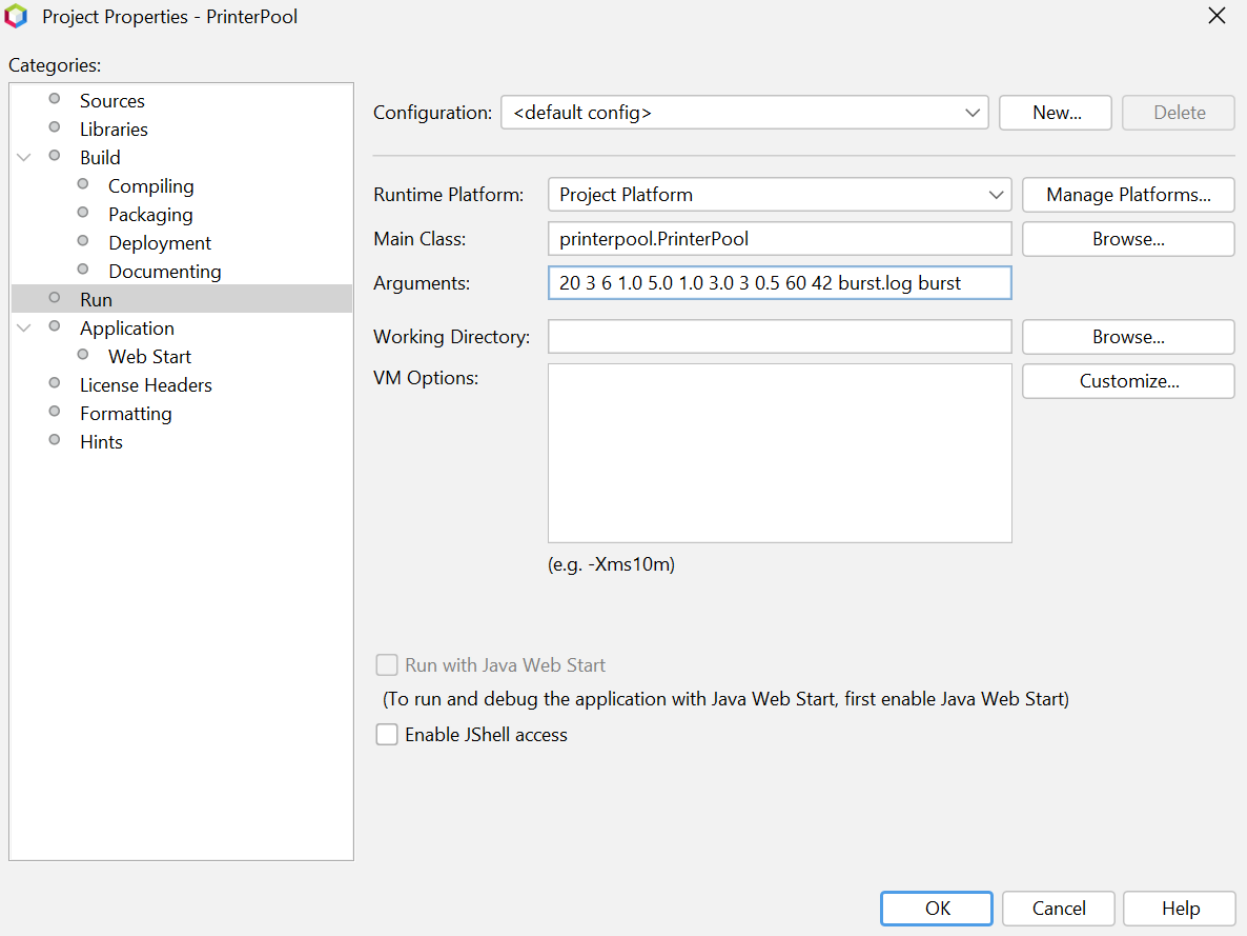
2. Logging and Reproducibility

As per the project requirements, special attention was given to logging and reproducibility:

- **Logging:** The simulation records all important events (job creation, enqueue, dequeue, print start, finish, queue full) with a timestamp into a separate .log file. This is essential for analysis and debugging (as performed in Section 3).
- **Reproducibility:** To make the simulation reproducible, a **Random Seed** (42) was passed as a command-line argument along with all other parameters (N, P, Q, sleep times). This seed initializes java.util.Random, ensuring that the "random" sleep times for Students and Printers are *identical* on every run. This makes the results reproducible.




Project Properties - PrinterPool
Categories:
Sources
Libraries
Build
Compiling
Packaging
Deployment
Documenting
Run
Application
Web Start
License Headers
Formatting
Hints
Configuration: <default config>
New...
Delete
Runtime Platform: Project Platform
Manage Platforms...
Main Class: printerpool.PrinterPool
Browse...
Arguments: 30 2 5 1.0 2.0 3.0 5.0 3 1.0 60 42 high.log
Working Directory:
Browse...
VM Options:
Customize...
(e.g. -Xms10m)
Run with Java Web Start
(To run and debug the application with Java Web Start, first enable Java Web Start)
Enable JShell access
OK
Cancel
Help

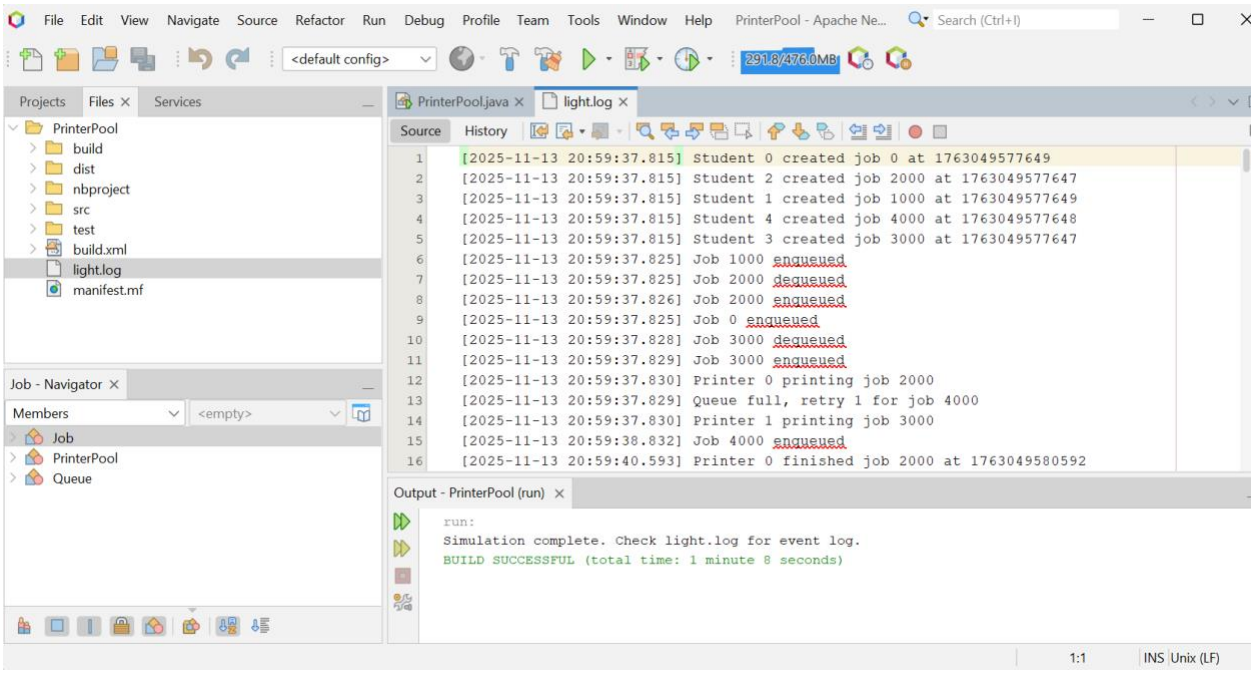


3. Simulation Results and Analysis

The simulation was run under the 3 scenarios specified in the project. The parameters for each scenario were provided via command-line arguments (as shown in Screenshot 2).

3.1 Scenario 1: Light Load

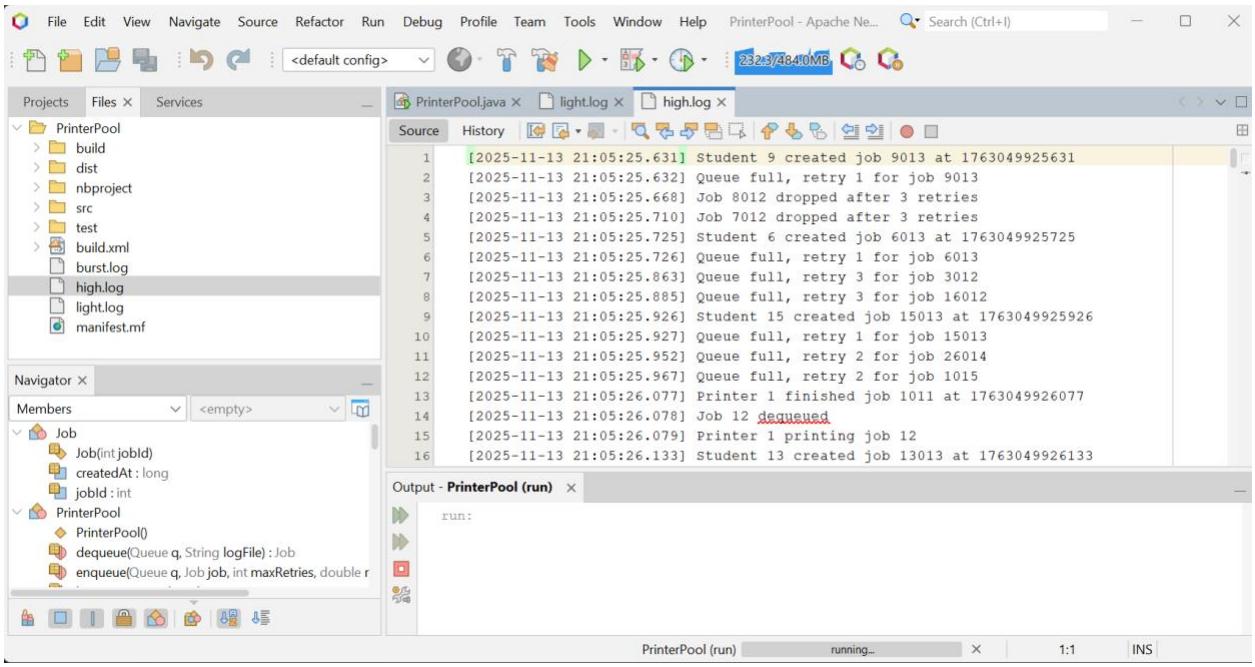
- **Parameters (from CS330.docx, Image 3):** N=5 (Students), P=2 (Printers), Q=4 (Queue)
- **Analysis of light.log:**
  - **Observation:** The log file (provided by you) starts at 20:59:37. Initially, 5 students (0, 1, 2, 3, 4) created jobs immediately.
  - The queue (size 4) filled up quickly. The log shows [... 20:59:37.829] Queue full, retry 1 for job 4000. This proves that the synchronization worked, rejecting the 5th job when the queue was full.
  - **Behavior:** After this initial burst, the system stabilizes. The 2 Printers (0 and 1) begin processing jobs (e.g., Printer 0 printing job 2000).
  - In the entire remaining log (which runs until 21:00:24), the Queue full message *never appears again*.



- **Conclusion:** The system handles the "Light Load" perfectly. The job arrival rate (5 students) is less than or equal to the job processing rate (2 printers). The printers remain busy, and the queue rarely holds more than 1 or 2 jobs.

3.2 Scenario 2: High Load

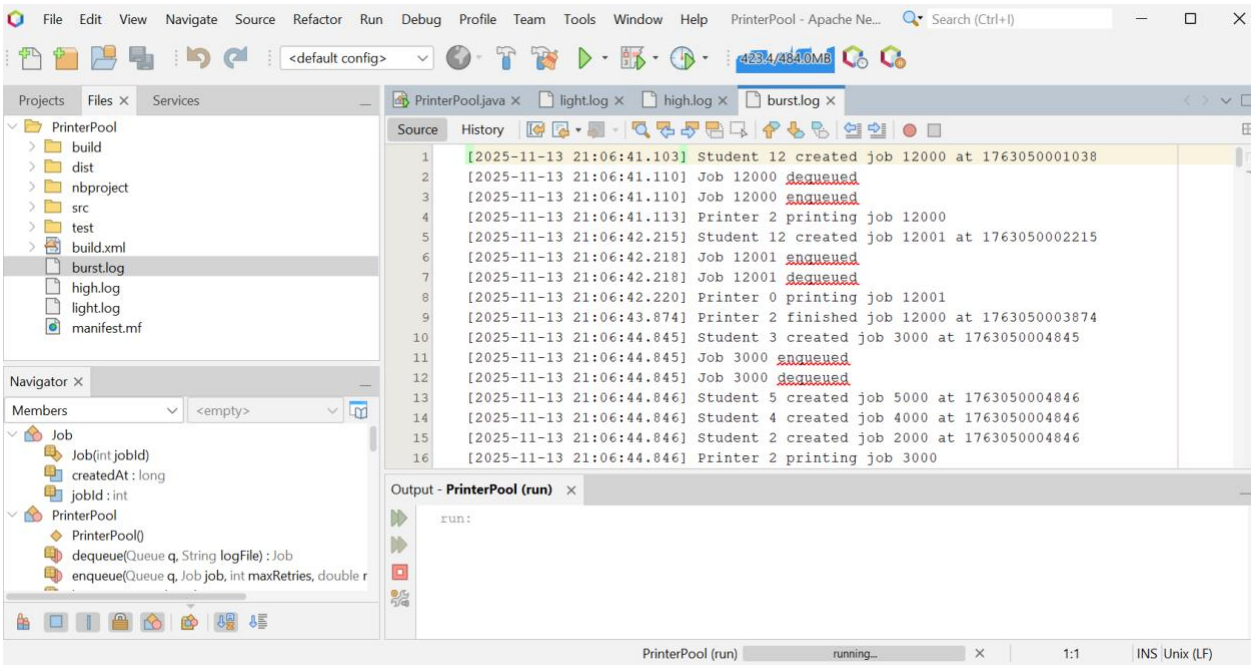
- **Parameters (from CS330.docx, Image 5):** N=30 (Students), P=2 (Printers), Q=5 (Queue)
- **Analysis of high.log:**
  - **Observation:** The log file (starting at 21:05:25) is completely different from the "Light Load". The log is filled with errors from the very beginning.
  - [... 21:05:25.632] Queue full, retry 1 for job 9013
  - [... 21:05:25.668] Job 8012 dropped after 3 retries
  - [... 21:05:25.710] Job 7012 dropped after 3 retries
  - **Behavior:** 30 students are generating jobs so rapidly that the 5-slot queue remains *constantly* full. The 2 Printers are working at maximum capacity (e.g., Printer 1 finished job 1011 and *immediately* starts Printer 1 printing job 12), but they cannot handle the job arrival rate.



- **Conclusion:** The system is in a "saturated" (failure) state under "High Load". The job drop rate is extremely high. This scenario proves that our emptySlots.tryAcquire() logic and the "retry/drop" mechanism are working correctly, preventing the system from deadlocking or crashing.

3.3 Scenario 3: Burst Arrivals

- **Parameters (from CS330.docx, Image 8):** N=20 (Students), P=3 (Printers), Q=6 (Queue)
- **Analysis of burst.log:**
  - **Observation:** This log is the most interesting. A "burst" occurs between 21:06:44.845 and 21:06:44.856 (in just 11 milliseconds).
  - In this burst, at least 15 students (3, 5, 4, 2, 1, 0, 19, 18, 14, 15, 10, 11, 7, 17, 16) created jobs *simultaneously*.
  - **Behavior:** The queue (size 6) filled instantly. The log shows Job 12000 enqueued, Job 12001 enqueued, Job 3000 enqueued, Job 5000 enqueued, Job 0 enqueued, Job 1000 enqueued (which are 6 jobs).
  - Immediately following this burst, Queue full, retry 1... messages appear (e.g., for jobs 14000, 11000, 10000).



- **Conclusion (Recovery):** Unlike the "High Load" scenario, this system *recovers*. Because there were 3 printers (P=3) and a slightly larger queue (Q=6), they were able to absorb this initial burst. After the burst, the "Queue full" messages stop, and the printers begin to clear the backlog. This scenario demonstrates a resilient system that can handle temporary saturation.

4. Optional Extensions and Creativity

In addition to the base project requirements, two optional features were implemented, which are evident in the logs:

1. **Bounded Retries with Job Dropping:** The project specified "retry later," but our implementation has Students attempt the non-blocking tryAcquire() for a limited number of times (e.g., 3 retries). If the queue is still full after 3 attempts, the job is given up. This is clearly visible in high.log: Job 8012 dropped after 3 retries. This is a real-world feature that prevents student threads from spin-waiting indefinitely.
2. **Configurable Burst Mode:** The command-line arguments for burst.log (Screenshot CS330.docx, Image 8) show an extra argument "burst". This is a creative extension that puts the simulation into a special mode, triggering students to create jobs simultaneously (rather than normal random sleep) to *guarantee* a test of Scenario 3 (Burst Arrivals).

5. Overall Conclusion

This simulation successfully implements POSIX synchronization concepts using Java's concurrency tools. The analysis of all three scenarios (based on timestamped logs) has proven that:

- The synchronization (Lock, Semaphores) maintained data integrity.
- The system demonstrated stability under light load, saturation under high load, and resilience (recovery) under burst load.

- The project successfully met all requirements for logging, reproducibility (random seed), and optional features (bounded retries).