Introduction

The problem is to implement a subclass of Java class `Printer`, which should handle new requests for printing – jobs. At one time only one job can be processed by the printer (mutual exclusion) and the producers of jobs are implemented as threads (synchronization problem).

Solution should satisfy these points:

1. print each submitted document without unnecessary delay.
2. print at most one document at a time (i.e. having at most one invocation of `print(Job job)` at any time).
3. return from `handleNewJob` as quickly as possible, i.e. as soon as the specified job has been queued for printing.
4. be thread-safe, so that (e.g.) jobs are not lost in the face of multiple threads calling the `submit`, `getStatus` or `waitFor` concurrently.
5. be as efficient as possible, for example with no busy waiting.
6. terminate any internal threads in a timely manner when `stop()` is called.

Terminology

I implemented one thread, called `PrintWorker` thread, as dedicated thread for handling new elements in the job queue, I will call it *consumer*. Test class `TestPrinter` creates and runs five instances of `TestThread`. They simultaneously submit new job for printer. I will call them *producers*. It entails, there will be five consumers, one producer and one main thread running in parallel.

Implementation

My solution consists of two Java classes. First is the subclass of the abstract class `Printer`, called `Gutenberg` and the second is a wrapper class around queue (potentially infinite FIFO buffer) with thread safe operation (assuming multi-producer, one consumer environment), called `JobQueue`.

Class JobQueue

This class has only two private attributes:

- `Queue<Job> queue` – Queue implemented as `LinkedList<Job>`.
- `volatile int full` – Counter representing the number of elements in the queue (initialized to 0). Is set as volatile, because Java keeps cached values of variables for each thread and this variable is shared across many threads and its value is necessary for synchronizing the producers and consumer.

..and these methods:

- `synchronized void addJob(Job job)`

Used only by producers. Keyword `synchronized` forces mutual exclusion amongst the producers, because adding to queue is not an atomic operation neither incrementation of the field full. Last statement of this method is invocation of `notifyAll()`, so in the case the consumer is faster than producers and waits in *P-operation*, he will be woken up. In this case `notify()` will work as well.

- `Job removeJob()`

Used only by `consumer.` Returns the head of the queue. If there is no element, it will wait in *P-operation*. Since there is only one consumer, there is no need to make this method `synchronized.` It internally uses method `p()`.

- `boolean isJobQueued(Job job)`

Returns `true`, if the job is present in the queue. Used by method `getStatus` of class `Gutenberg.`

- `void p()`

Classical *P-operation* from semaphore theory.

Class Gutenberg

This class represents a printer. Besides the methods, whose signatures are prescribed in the abstract class `Printer`, it contains the inner class `PrintWorker` which extends class `Thread.` Its attributes are `Job actual` – it holds reference to the job currently being printed, `JobQueue queue` – the reference to the job queue and `PrintWorker worker` – the reference to the consumer thread (instance of the inner class).

Verification

1. Consumer is started in method initialize and then in infinite loop is trying to remove a job from the queue. If there is no item in the queue, this thread is suspended and waits for notification from any of producers – no busy waiting in while loop. As soon as the job is added to job queue, this thread is woken up and prints the document without unnecessary delay.
2. There is only one consumer using the resource (print method), so this point can't be violated. In other words, single threaded application can't run one method simultaneously.
3. Handling new job contains only these operations:

```
queue.add(job);

full++;

notifyAll();
```

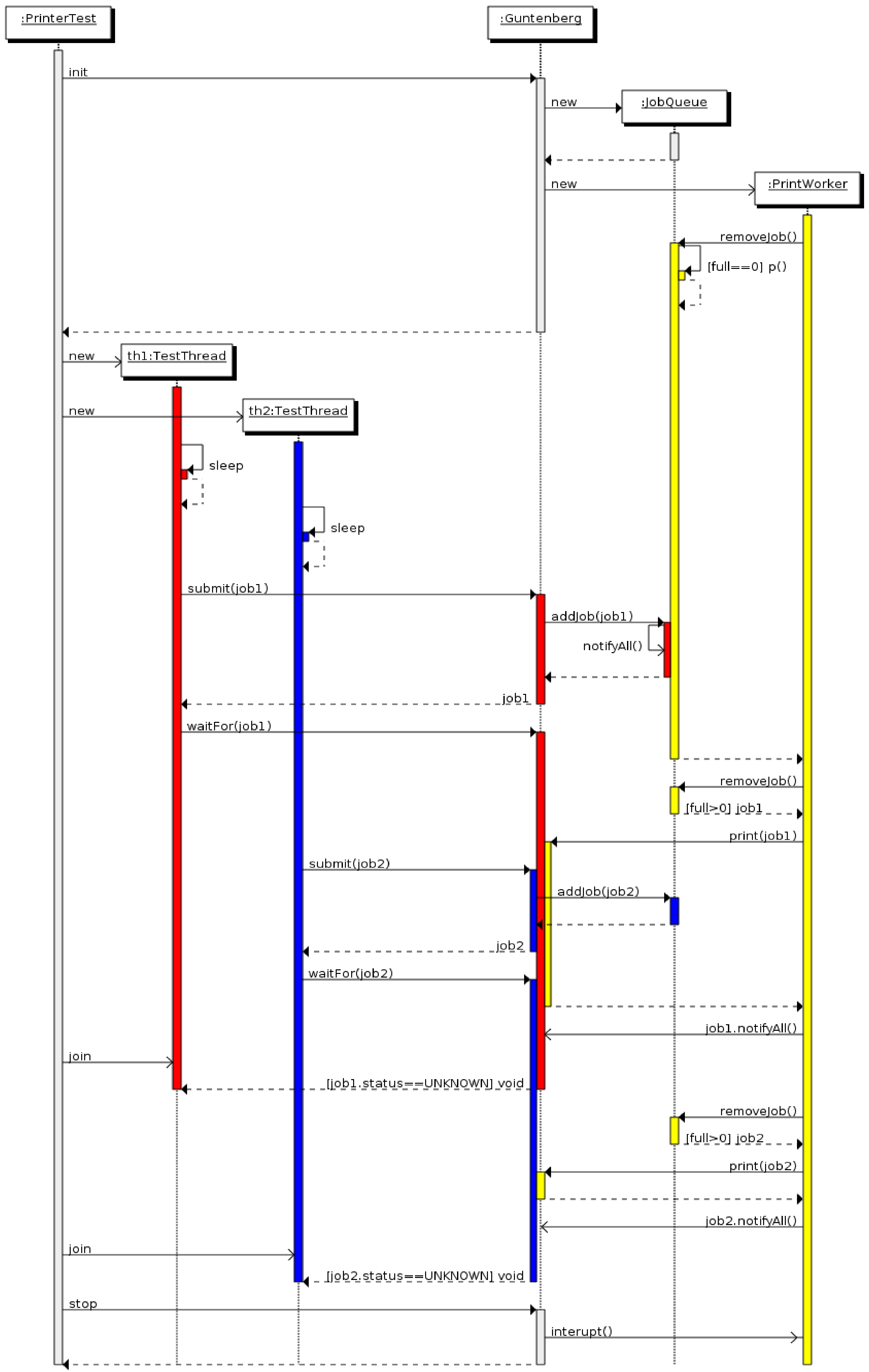
Therefore no unnecessary delay is done. Worker's priority is set to maximum.

4. It is difficult to prove, that my solution is thread-safe in the scope of two pages without making model and verifying this model against some formulas of modal/temporal logic by model checking. However, I think it is thread-safe and race condition can never happens.
5. My implementation never waits in infinite while loop exploiting resources. Whenever a thread should wait, it is suspended by `wait()` and then eventually woken up by `notifyAll()`. For storing jobs is used the linked list which is not as effective as an array, but it can provide infinite buffer.
6. All test threads can possibly wait forever in the method `waitFor(job)`, but hopefully there is running instance of thread `PrintWorker`, which notifies those test threads by running `notifyAll()` on the lock object `job`. Only thread to possibly never end is the consumer, so it has to be interrupted in method `stop()` (flag of this thread is set to interrupted and system scheduler then ends this thread).

Detail of possible (since interleaving of actions is nondeterministically scheduled by JVM) interaction among the threads can be seen on the sequence diagram in appendix. For simplicity, there are only two test threads.

Output

```
$ java PrinterTest Gutengerg
1269199576078 main: Creating printer Gutengerg
1269199576079 main: Initialise printer
1269199576080 main: Testing printer...
1269199576081 main: Waiting for threads to end
1269199576205 Thread-1: Print I am thinking of 6
1269199576207 Thread-0: Printing Printer$Job@1186fab...
1269199576209 Thread-1: Got job Printer$Job@1186fab - status PRINTING; waiting...
1269199576389 Thread-0: Done printing Printer$Job@1186fab: I am thinking of 6
1269199576390 Thread-1: Waited for Printer$Job@1186fab - status UNKNOWN
1269199576504 Thread-5: Print I am thinking of 2
1269199576504 Thread-5: Got job Printer$Job@1d5550d - status QUEUED; waiting...
1269199576504 Thread-0: Printing Printer$Job@1d5550d...
1269199576657 Thread-3: Print I am thinking of 9
1269199576658 Thread-3: Got job Printer$Job@c2ea3f - status QUEUED; waiting...
1269199576686 Thread-0: Done printing Printer$Job@1d5550d: I am thinking of 2
1269199576686 Thread-0: Printing Printer$Job@c2ea3f...
1269199576686 Thread-5: Waited for Printer$Job@1d5550d - status UNKNOWN
1269199576726 Thread-4: Print I am thinking of 2
1269199576727 Thread-4: Got job Printer$Job@a0dcd9 - status QUEUED; waiting...
1269199576747 Thread-2: Print I am thinking of 9
1269199576747 Thread-2: Got job Printer$Job@1034bb5 - status QUEUED; waiting...
1269199576869 Thread-0: Done printing Printer$Job@c2ea3f: I am thinking of 9
1269199576869 Thread-0: Printing Printer$Job@a0dcd9...
1269199576870 Thread-3: Waited for Printer$Job@c2ea3f - status UNKNOWN
1269199577052 Thread-0: Done printing Printer$Job@a0dcd9: I am thinking of 2
1269199577052 Thread-0: Printing Printer$Job@1034bb5...
1269199577052 Thread-4: Waited for Printer$Job@a0dcd9 - status UNKNOWN
1269199577237 Thread-0: Done printing Printer$Job@1034bb5: I am thinking of 9
1269199577238 Thread-2: Waited for Printer$Job@1034bb5 - status UNKNOWN
1269199577238 main: Stopping printer
1269199577238 main: Done - should exit now
```


:PrinterTest
:Guntenberg
init
new
:JobQueue
new
:PrintWorker
removeJob()
[full==0] p()
new
th1:TestThread
new
th2:TestThread
sleep
sleep
submit(job1)
addJob(job1)
notifyAll()
job1
waitFor(job1)
removeJob()
[full>0] job1
print(job1)
submit(job2)
addJob(job2)
job2
waitFor(job2)
job1.notifyAll()
join
[job1.status==UNKNOWN] void
removeJob()
[full>0] job2
print(job2)
job2.notifyAll()
join
[job2.status==UNKNOWN] void
stop
interupt()