Problem 1. Given that the splitter will be called concurrently by a number of \( N \) threads, we can think about this as selecting 1 thread to return \( \text{stop} \). All the threads arriving during this election but not chosen to return \( \text{stop} \) can return \( \text{left} \), and the ones arriving after the election can return \( \text{right} \). It is acceptable to not have any threads selected to get \( \text{stop} \) (e.g., in case more than 1 thread executes splitter), but it must never be possible to have more than 1 thread return \( \text{stop} \) during a concurrent execution.

We use two registers:

- \( P \) (multi-valued), and
- \( S \) (binary, initialized to false)

\( P \) holds the id of the thread that should get \( \text{stop} \). \( S \) marks whether a \( \text{stop} \) thread has been selected. When a thread calls splitter, it needs to check whether \( S \) is false, and if so, set it to true and return \( \text{stop} \). The difficulty is that we cannot use atomic getAndSet-type primitives, so multiple threads first reading the value of \( S \) and then updating it could mistakenly think they each got \( \text{stop} \). In order to decide which thread should get \( \text{stop} \), each thread volunteers itself by setting the value of \( P \) to their own id. The last thread to update \( P \) wins.

After volunteering, a thread checks the \( S \) flag, and if it is true, then the thread knows it arrived after the election, and so it gets \( \text{right} \). If \( S \) still false, then the thread (one of potentially many) arrived during the election, so it sets \( S \) to true, and checks if it won (i.e., if the value of \( P \) is equal to its own id). If the thread won, it simply gets \( \text{stop} \). Otherwise, it means some other thread managed to change \( P \) after it, hence the current thread lost and gets \( \text{left} \).

It is possible that a thread updates \( P \) and becomes the winner just as another thread sets \( S \) to true, but before checking to see if it won. In this case, 0 threads get \( \text{stop} \), as the winner then reads \( S \), finds it true, concludes it arrived after the election, and gets \( \text{right} \).

However, it is impossible for more than 1 thread to get \( \text{stop} \). Assume by way of contradiction that 2 threads with identifiers \( i \) and \( j \) both return \( \text{stop} \). Furthermore, assume without loss of generality that thread \( i \) first performed the read of \( P \) and then thread \( j \) read \( P \). Therefore, the order of events will be \( \text{read}_i(P = i) \rightarrow \text{read}_j(P = j) \) (i.e., since both threads return \( \text{stop} \) they read their own identifier when reading from \( P \)). We furthermore know that both threads write register \( P \) at the beginning of their execution and since both threads return \( \text{stop} \) they read \( S \) to be false. So we have the following ordering of events:

- \( \text{write}_i(P \leftarrow i) \rightarrow \text{read}_i(S = \text{false}) \rightarrow \text{write}_i(S \leftarrow \text{true}) \rightarrow \text{read}_i(P = i) \).
- \( \text{write}_j(P \leftarrow j) \rightarrow \text{read}_j(S = \text{false}) \rightarrow \text{write}_j(S \leftarrow \text{true}) \rightarrow \text{read}_j(P = j) \).

Since thread \( i \) read \( P = i \) (and thread \( j \) read \( P = j \)) it means that \( \text{write}_j(P \leftarrow j) \) takes place after \( \text{read}_i(P = i) \). So we have:

- \( \text{write}_i(P \leftarrow i) \rightarrow \text{read}_i(S = \text{false}) \rightarrow \text{write}_i(S \leftarrow \text{true}) \rightarrow \text{read}_i(P = i) \rightarrow \text{write}_j(P = j) \rightarrow \text{read}_j(S = \text{false}) \).

This is a contradiction, since thread \( i \) wrote \( \text{true} \) to \( S \) and then \( j \) read \( \text{false} \) from \( S \).
upon $\text{splitter}_i$

\begin{itemize}
\item $P \leftarrow i$;
\item if $S$ then return "right";
\item $S \leftarrow \text{true}$;
\item if $P = i$ then return "stop";
\item return "left";
\end{itemize}

\textbf{Algorithm 1:} Sample implementation of the $\text{splitter}$ object.
Problem 2.
Algorithm 2 presents the pseudocode of an atomic wait-free snapshot as described in class. For a program running \( N \) threads, in order to run a scan or a collect operation, all the registers of the \( N \) threads need to be read. Writes are done only on a thread’s register \( R[i] \). Since we know beforehand that many of the \( N \) threads will not use the snapshot, a better solution is to assign registers to threads on demand.

We assume that there exists an \( \text{obtain}() \) operation that each thread can call to get a register that is assigned only to itself. Algorithm 3 presents the implementation of \( \text{update}(i) \) and \( \text{scan}(i) \) using the aforementioned operation. Importantly, the number of registers that need to be parsed now in \( \text{scan}(i) \) is dependent on the number of threads that have written to the object (and thus have been assigned a register).

\[
\text{upon scan}_i \quad t_1 \leftarrow \text{collect}(), t_2 \leftarrow t_1; \\
\text{while true do} \\
\quad t_3 \leftarrow \text{collect}(); \\
\quad \text{if } t_3 = t_2 \text{ then return } \langle t_3[1].\text{val}, \ldots, t_3[N].\text{val} \rangle; \\
\quad \text{for } k \leftarrow 1 \text{ to } N \text{ do} \\
\quad\quad \text{if } t_3[k].ts \geq t_1[k].ts + 2 \text{ then return } t_3[k].\text{snapshot}; \\
\quad t_2 \leftarrow t_3;
\]

\[
\text{procedure } \text{collect}() \\
\quad \text{for } k \leftarrow 1 \text{ to } N \text{ do} \\
\quad\quad x[k] \leftarrow R[k]; \\
\quad \text{return } x;
\]

\[
\text{procedure } \text{update}(i, v) \\
\quad ts \leftarrow ts + 1; \\
\quad \text{snapshot} \leftarrow \text{scan}(); \\
\quad R[i] \leftarrow \langle ts, v, \text{snapshot} \rangle;
\]

Algorithm 2: Sample implementation of a non-adaptive snapshot. Each thread has its own register.

\[
\text{procedure } \text{update}(v) \\
\quad \text{if } myreg = \bot \text{ then} \\
\quad\quad myreg \leftarrow \text{obtain}(); \\
\quad ts \leftarrow ts + 1; \\
\quad \text{snapshot} \leftarrow \text{scan}(); \\
\quad R[myreg] \leftarrow \langle ts, v, \text{snapshot} \rangle;
\]

Algorithm 3: Sample implementation of \( \text{update}(v) \) and \( \text{scan}(i) \) in an adaptive snapshot. Each thread that affects the snapshot calls \( \text{obtain}(\) to get assigned a register.
Implementing **obtain()**

Recall the splitter object implemented in the previous exercise: it allows selecting at most 1 thread out of multiple accessing the object concurrently, while partitioning the remaining threads into 2 separate pools (left, right). Keeping this in mind, we create a matrix of registers and splitters, as presented in figure 1. A thread calling obtain() starts from the top-left corner and calls the splitter in that cell. If it gets stop, then it obtains that register. Otherwise, it moves 1 column to the right, or 1 row downwards for left, and repeats the process.

\[
\text{procedure obtain()}
\]
\[
x \leftarrow 1, y \leftarrow 1;
\]
\[
\text{while true do}
\]
\[
s \leftarrow S[x, y].\text{splitter();}
\]
\[
\text{if } s = \text{“stop” then return } R[x, y];
\]
\[
\text{else if } s = \text{“left” then } y \leftarrow y + 1;
\]
\[
\text{else } x \leftarrow x + 1;
\]

Algorithm 4: Implementation of obtain() using a matrix of registers and splitter objects.

Implementing **collect()**

Finally, we need to adapt the collect() call to the matrix of registers now being used. The insight here is that all the registers that have been assigned from each matrix diagonal that has had at least 1 splitter used need to be taken into account.

\[
\text{procedure collect()}
\]
\[
C \leftarrow \langle \rangle;
\]
\[
d \leftarrow 1;
\]
\[
\text{while diagonal } d \text{ has a splitter that has been traversed do}
\]
\[
C \leftarrow C \cdot \langle \text{values of all non-} \bot \text{ registers on diagonal } d \rangle;
\]
\[
d \leftarrow d + 1;
\]
\[
\text{return } C;
\]

Algorithm 5: Implementation of collect() using a matrix of registers and splitter objects.

Figure 1: Matrix of registers and splitter objects.

Figure 2: Matrix of registers and splitter objects.