```solidity
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v4.7.0) (token/ERC20/extensions/ERC20Snapshot.sol)

pragma solidity ^0.8.0;

import "../ERC20.sol";
import "../../../utils/Arrays.sol";
import "../../../utils/Counters.sol";

/**
 * @dev This contract extends an ERC20 token with a snapshot mechanism. When a snapshot is created, the balances and
 * total supply at the time are recorded for later access.
 *
 * This can be used to safely create mechanisms based on token balances such as trustless dividends or weighted voting.
 * In naive implementations it's possible to perform a "double spend" attack by reusing the same balance from different
 * accounts. By using snapshots to calculate dividends or voting power, those attacks no longer apply. It can also be
 * used to create an efficient ERC20 forking mechanism.
 *
 * Snapshots are created by the internal {_snapshot} function, which will emit the {Snapshot} event and return a
 * snapshot id. To get the total supply at the time of a snapshot, call the function {totalSupplyAt} with the snapshot
 * id. To get the balance of an account at the time of a snapshot, call the {balanceOfAt} function with the snapshot id
 * and the account address.
 *
 * NOTE: Snapshot policy can be customized by overriding the {_getCurrentSnapshotId} method. For example, having it
 * return `block.number` will trigger the creation of snapshot at the beginning of each new block. When overriding this
 * function, be careful about the monotonicity of its result. Non-monotonic snapshot ids will break the contract.
 *
 * Implementing snapshots for every block using this method will incur significant gas costs. For a gas-efficient
 * alternative consider {ERC20Votes}.
 *
 * ==== Gas Costs
 *
 * Snapshots are efficient. Snapshot creation is _O(1)_. Retrieval of balances or total supply from a snapshot is _O(log
 * n)_ in the number of snapshots that have been created, although _n_ for a specific account will generally be much
 * smaller since identical balances in subsequent snapshots are stored as a single entry.
 *
 * There is a constant overhead for normal ERC20 transfers due to the additional snapshot bookkeeping. This overhead is
 * only significant for the first transfer that immediately follows a snapshot for a particular account. Subsequent
 * transfers will have normal cost until the next snapshot, and so on.
 */

abstract contract ERC20Snapshot is ERC20 {
    // Inspired by Jordi Baylina's MiniMeToken to record historical balances:
    // https://github.com/Giveth/minime/blob/ea04d950eea153a04c51fa510b068b9dded390cb/contracts/MiniMeToken.sol

    using Arrays for uint256[];
    using Counters for Counters.Counter;

    // Snapshotted values have arrays of ids and the value corresponding to that id. These could be an array of a
    // Snapshot struct, but that would impede usage of functions that work on an array.
    struct Snapshots {
        uint256[] ids;
```

```
53              uint256[] values;
54          }
55
56          mapping(address => Snapshots) private _accountBalanceSnapshots;
57          Snapshots private _totalSupplySnapshots;
58
59          // Snapshot ids increase monotonically, with the first value being 1. An id of 0
            is invalid.
60          Counters.Counter private _currentSnapshotId;
61
62          /**
63           * @dev Emitted by {_snapshot} when a snapshot identified by `id` is created.
64           */
65          event Snapshot(uint256 id);
66
67          /**
68           * @dev Creates a new snapshot and returns its snapshot id.
69           *
70           * Emits a {Snapshot} event that contains the same id.
71           *
72           * {_snapshot} is `internal` and you have to decide how to expose it externally.
            Its usage may be restricted to a
73           * set of accounts, for example using {AccessControl}, or it may be open to the
            public.
74           *
75           * [WARNING]
76           * ====
77           * While an open way of calling {_snapshot} is required for certain trust
            minimization mechanisms such as forking,
78           * you must consider that it can potentially be used by attackers in two ways.
79           *
80           * First, it can be used to increase the cost of retrieval of values from
            snapshots, although it will grow
81           * logarithmically thus rendering this attack ineffective in the long term.
            Second, it can be used to target
82           * specific accounts and increase the cost of ERC20 transfers for them, in the
            ways specified in the Gas Costs
83           * section above.
84           *
85           * We haven't measured the actual numbers; if this is something you're interested
            in please reach out to us.
86           * ====
87           */
88          function _snapshot() internal virtual returns (uint256) {
89              _currentSnapshotId.increment();
90
91              uint256 currentId = _getCurrentSnapshotId();
92              emit Snapshot(currentId);
93              return currentId;
94          }
95
96          /**
97           * @dev Get the current snapshotId
98           */
99          function _getCurrentSnapshotId() internal view virtual returns (uint256) {
100             return _currentSnapshotId.current();
101         }
102
103         /**
104          * @dev Retrieves the balance of `account` at the time `snapshotId` was created.
105          */
106         function balanceOfAt(address account, uint256 snapshotId) public view virtual
            returns (uint256) {
107             (bool snapshotted, uint256 value) = _valueAt(snapshotId,
                _accountBalanceSnapshots[account]);
108
109             return snapshotted ? value : balanceOf(account);
110         }
111
112         /**
113          * @dev Retrieves the total supply at the time `snapshotId` was created.
114          */
115         function totalSupplyAt(uint256 snapshotId) public view virtual returns (uint256) {
```

```
116             (bool snapshotted, uint256 value) = _valueAt(snapshotId,
                _totalSupplySnapshots);
117
118             return snapshotted ? value : totalSupply();
119         }
120
121         // Update balance and/or total supply snapshots before the values are modified.
                This is implemented
122         // in the _beforeTokenTransfer hook, which is executed for _mint, _burn, and
                _transfer operations.
123         function _beforeTokenTransfer(address from, address to, uint256 amount) internal
                virtual override {
124             super._beforeTokenTransfer(from, to, amount);
125
126             if (from == address(0)) {
127                 // mint
128                 _updateAccountSnapshot(to);
129                 _updateTotalSupplySnapshot();
130             } else if (to == address(0)) {
131                 // burn
132                 _updateAccountSnapshot(from);
133                 _updateTotalSupplySnapshot();
134             } else {
135                 // transfer
136                 _updateAccountSnapshot(from);
137                 _updateAccountSnapshot(to);
138             }
139         }
140
141         function _valueAt(uint256 snapshotId, Snapshots storage snapshots) private view
                returns (bool, uint256) {
142             require(snapshotId > 0, "ERC20Snapshot: id is 0");
143             require(snapshotId <= _getCurrentSnapshotId(), "ERC20Snapshot: nonexistent id"
                );
144
145             // When a valid snapshot is queried, there are three possibilities:
146             //  a) The queried value was not modified after the snapshot was taken.
                Therefore, a snapshot entry was never
147             //  created for this id, and all stored snapshot ids are smaller than the
                requested one. The value that corresponds
148             //  to this id is the current one.
149             //  b) The queried value was modified after the snapshot was taken.
                Therefore, there will be an entry with the
150             //  requested id, and its value is the one to return.
151             //  c) More snapshots were created after the requested one, and the queried
                value was later modified. There will be
152             //  no entry for the requested id: the value that corresponds to it is that
                of the smallest snapshot id that is
153             //  larger than the requested one.
154             //
155             // In summary, we need to find an element in an array, returning the index of
                the smallest value that is larger if
156             // it is not found, unless said value doesn't exist (e.g. when all values are
                smaller). Arrays.findUpperBound does
157             // exactly this.
158
159             uint256 index = snapshots.ids.findUpperBound(snapshotId);
160
161             if (index == snapshots.ids.length) {
162                 return (false, 0);
163             } else {
164                 return (true, snapshots.values[index]);
165             }
166         }
167
168         function _updateAccountSnapshot(address account) private {
169             _updateSnapshot(_accountBalanceSnapshots[account], balanceOf(account));
170         }
171
172         function _updateTotalSupplySnapshot() private {
173             _updateSnapshot(_totalSupplySnapshots, totalSupply());
174         }
175
```

```
176         function _updateSnapshot(Snapshots storage snapshots, uint256 currentValue)
            private {
177             uint256 currentId = _getCurrentSnapshotId();
178             if (_lastSnapshotId(snapshots.ids) < currentId) {
179                 snapshots.ids.push(currentId);
180                 snapshots.values.push(currentValue);
181             }
182         }
183
184         function _lastSnapshotId(uint256[] storage ids) private view returns (uint256) {
185             if (ids.length == 0) {
186                 return 0;
187             } else {
188                 return ids[ids.length - 1];
189             }
190         }
191     }
192
```