pragma solidity ^ 0.5.1;

import "./Auction.sol";

contract VickreyAuction is Auction {

enum Phase {

GracePeriod,

Commitment,

Withdrawal,

Opening,

Finished

}

Phase public phase;

uint startPhaseBlock;

uint reservePrice;

uint min\_deposit;

uint commitment\_len;

uint withdrawal\_len;

uint opening\_len;

address payable highestBidder;

uint highestBid;

uint secondHighestBid;

bool firstOpen = true;

struct Bid {

uint value;

bytes32 nonce;

bytes32 hash;

uint deposit;

}

mapping(address => Bid) bids;

event withdrawalStarted();

event openingStarted();

event withdrawalExecuted(address bidder, uint value, address seller, uint val);

constructor(

string memory \_itemName,

uint \_reservePrice,

uint \_min\_deposit,

uint \_commitment\_len,

uint \_withdrawal\_len,

uint \_opening\_len

) public {

require(\_reservePrice > 0, "Reserve price should be greater than zero.");

require(\_min\_deposit >= \_reservePrice, "The deposit should be greater than the reserve price");

require(\_commitment\_len > 0, "The lenght of commitment should be greater than zero.");

require(\_withdrawal\_len > 0, "The lenght of withdrawal should be greater than zero.");

require(\_opening\_len > 0, "The lenght of opening should be greater than zero.");

description.seller = msg.sender;

description.itemName = \_itemName;

phase = Phase.GracePeriod;

reservePrice = \_reservePrice;

min\_deposit = \_min\_deposit;

commitment\_len = \_commitment\_len;

withdrawal\_len = \_withdrawal\_len;

opening\_len = \_opening\_len;

startPhaseBlock = block.number;

}

modifier duringCommitment {

require(phase == Phase.Commitment, "Commitment phase not started yet");

require((block.number - startPhaseBlock) <= commitment\_len, "Commitment phase is ended");

\_;

}

modifier duringWithdrawal {

require(phase == Phase.Withdrawal, "Withdrawal phase not started yet");

require((block.number - startPhaseBlock) <= withdrawal\_len, "Withdrawal phase is ended");

\_;

}

modifier duringOpening {

require(phase == Phase.Opening, "Opening phase not started yet");

require((block.number - startPhaseBlock) <= opening\_len, "Opening phase is ended");

\_;

}

function activateAuction() public onlySeller {

require(phase == Phase.GracePeriod, "To activate the contract you must be in the Grace Period");

require(block.number - startPhaseBlock > 20, "Grace period is not finished yet");

phase = Phase.Commitment;

description.startBlock = block.number;

startPhaseBlock = block.number;

emit auctionStarted();

}

function bid(bytes32 \_bidHash) public duringCommitment payable {

require(msg.value >= min\_deposit, "The value sent is not sufficient");

require(bids[msg.sender].value == 0, "You have already submitted a bid");

Bid memory tmp\_bid;

tmp\_bid.hash = \_bidHash;

tmp\_bid.deposit = msg.value;

bids[msg.sender] = tmp\_bid;

}

function startWithdrawal() public onlySeller {

require(phase == Phase.Commitment, "You can't start withdrawal before commitment");

require((block.number - startPhaseBlock) > commitment\_len, "Commitment period is not finished yet");

phase = Phase.Withdrawal;

startPhaseBlock = block.number;

emit withdrawalStarted();

}

function withdrawal() public duringWithdrawal {

//1. Checks

require(bids[msg.sender].deposit > 0, "You don't have any deposit to withdraw");

uint bidderRefund = bids[msg.sender].deposit / 2;

uint sellerRefund = bids[msg.sender].deposit - bidderRefund;

//2. Effects

bids[msg.sender].deposit = 0;

emit withdrawalExecuted(msg.sender, bidderRefund, description.seller, sellerRefund);

//3. Interaction

description.seller.transfer(sellerRefund);

msg.sender.transfer(bidderRefund);

}

function startOpening() public onlySeller {

require(phase == Phase.Withdrawal, "You can't start opening before withdrawal");

require((block.number - startPhaseBlock) > withdrawal\_len, "Commitment period is not finished yet");

phase = Phase.Opening;

startPhaseBlock = block.number;

emit openingStarted();

}

function open(bytes32 \_nonce) public duringOpening payable {

//control the correctness of the bid

require(keccak256(abi.encodePacked(msg.value, \_nonce)) == bids[msg.sender].hash, "Wrong hash");

//refund the deposit

uint deposit = bids[msg.sender].deposit;

bids[msg.sender].deposit = 0;

msg.sender.transfer(deposit);

//update the bid status

bids[msg.sender].value = msg.value;

bids[msg.sender].nonce = \_nonce;

//if it is the first opening

if (firstOpen) {

highestBidder = msg.sender;

highestBid = msg.value;

//if there is only one bid, the winner have to pay at least the reservePrice

secondHighestBid = reservePrice;

firstOpen = false;

} else {

//if the msg.value is more than the highest bid

if (msg.value > highestBid) {

//the highest bid becomes the second highest bid

secondHighestBid = highestBid;

//now we need to refund the bidder of the (old) highest bid

refund(highestBidder, highestBid);

//set the new highest bidder and its own bid

highestBidder = msg.sender;

highestBid = msg.value;

} else {

//check whether the msg.value is higher than the second highest bid

if (msg.value > secondHighestBid) secondHighestBid = msg.value;

//since the current opening is not the highest we can refund the sender

refund(msg.sender, msg.value);

}

}

}

function refund(address payable \_dest, uint value) internal {

\_dest.transfer(value);

}

function finalize() public onlySeller {

require(phase == Phase.Opening, "You can't finalize the contract before opening");

require((block.number - startPhaseBlock) > opening\_len, "Opening period is not finished yet");

//if there is a winner (at least one bid)

if (highestBidder != address(0)) {

description.winnerAddress = highestBidder;

description.winnerBid = secondHighestBid;

//refund the winner

highestBidder.transfer(highestBid - secondHighestBid);

//send ehter to the seller of the item

description.seller.transfer(description.winnerBid);

}

phase = Phase.Finished;

emit auctionFinished(description.winnerAddress, description.winnerBid, address(this).balance);

}

}