static int64\_t nTimeCheck = 0;

static int64\_t nTimeForks = 0;

static int64\_t nTimeVerify = 0;

static int64\_t nTimeConnect = 0;

static int64\_t nTimeIndex = 0;

static int64\_t nTimeCallbacks = 0;

static int64\_t nTimeTotal = 0;

static int64\_t nBlocksTotal = 0;

/\*\* Apply the effects of this block (with given index) on the UTXO set represented by coins.

 \*  Validity checks that depend on the UTXO set are also done; ConnectBlock()

 \*  can fail if those validity checks fail (among other reasons). \*/

bool CChainState::ConnectBlock(const CBlock& block, CValidationState& state, CBlockIndex\* pindex,

                               CCoinsViewCache& view, const CChainParams& chainparams, bool fJustCheck)

{

    AssertLockHeld(cs\_main);

    assert(pindex);

    assert(\*pindex->phashBlock == block.GetHash());

    int64\_t nTimeStart = GetTimeMicros();

    if (!CheckBlock(block, state, chainparams.GetConsensus(), !fJustCheck, !fJustCheck)) {

        if (state.CorruptionPossible()) {

            return AbortNode(state, "Corrupt block found indicating potential hardware failure; shutting down");

        }

        return error("%s: Consensus::CheckBlock: %s", \_\_func\_\_, FormatStateMessage(state));

    }

    uint256 hashPrevBlock = pindex->pprev == nullptr ? uint256() : pindex->pprev->GetBlockHash();

    assert(hashPrevBlock == view.GetBestBlock());

    if (block.GetHash() == chainparams.GetConsensus().hashGenesisBlock) {

        if (!fJustCheck)

            view.SetBestBlock(pindex->GetBlockHash());

        return true;

    }

    nBlocksTotal++;

    bool fScriptChecks = true;

    if (!hashAssumeValid.IsNull()) {

        BlockMap::const\_iterator  it = mapBlockIndex.find(hashAssumeValid);

        if (it != mapBlockIndex.end()) {

            if (it->second->GetAncestor(pindex->nHeight) == pindex &&

                    pindexBestHeader->GetAncestor(pindex->nHeight) == pindex &&

                    pindexBestHeader->nChainWork >= nMinimumChainWork) {

                fScriptChecks = (GetBlockProofEquivalentTime(\*pindexBestHeader, \*pindex, \*pindexBestHeader, chainparams.GetConsensus()) <= 60 \* 60 \* 24 \* 7 \* 2);

            }

        }

    }

    int64\_t nTime1 = GetTimeMicros(); nTimeCheck += nTime1 - nTimeStart;

    LogPrint(BCLog::BENCH, "    - Sanity checks: %.2fms [%.2fs (%.2fms/blk)]\n", MILLI \* (nTime1 - nTimeStart), nTimeCheck \* MICRO, nTimeCheck \* MILLI / nBlocksTotal);

    bool fEnforceBIP30 = !pindex->phashBlock;

    static constexpr int BIP34\_IMPLIES\_BIP30\_LIMIT = 1983702;

    assert(pindex->pprev);

    CBlockIndex \*pindexBIP34height = pindex->pprev->GetAncestor(chainparams.GetConsensus().BIP34Height);

    fEnforceBIP30 = fEnforceBIP30 && (!pindexBIP34height || !(pindexBIP34height->GetBlockHash() == chainparams.GetConsensus().BIP34Hash));

    if (fEnforceBIP30 || pindex->nHeight >= BIP34\_IMPLIES\_BIP30\_LIMIT) {

        for (const auto& tx : block.vtx) {

            for (size\_t o = 0; o < tx->vout.size(); o++) {

                if (view.HaveCoin(COutPoint(tx->GetHash(), o))) {

                    return state.DoS(100, error("ConnectBlock(): tried to overwrite transaction"),

                                     REJECT\_INVALID, "bad-txns-BIP30");

                }

            }

        }

    }

    int nLockTimeFlags = 0;

    if (VersionBitsState(pindex->pprev, chainparams.GetConsensus(), Consensus::DEPLOYMENT\_CSV, versionbitscache) == ThresholdState::ACTIVE) {

        nLockTimeFlags |= LOCKTIME\_VERIFY\_SEQUENCE;

    }

    unsigned int flags = GetBlockScriptFlags(pindex, chainparams.GetConsensus());

    int64\_t nTime2 = GetTimeMicros(); nTimeForks += nTime2 - nTime1;

    LogPrint(BCLog::BENCH, "    - Fork checks: %.2fms [%.2fs (%.2fms/blk)]\n", MILLI \* (nTime2 - nTime1), nTimeForks \* MICRO, nTimeForks \* MILLI / nBlocksTotal);

    CBlockUndo blockundo;

    CCheckQueueControl<CScriptCheck> control(fScriptChecks && nScriptCheckThreads ? &scriptcheckqueue : nullptr);

    std::vector<int> prevheights;

    CAmount nFees = 0;

    int nInputs = 0;

    int64\_t nSigOpsCost = 0;

    CDiskTxPos pos(pindex->GetBlockPos(), GetSizeOfCompactSize(block.vtx.size()));

    std::vector<std::pair<uint256, CDiskTxPos>> vPos;

    vPos.reserve(block.vtx.size());

    blockundo.vtxundo.reserve(block.vtx.size() - 1);

    std::vector<PrecomputedTransactionData> txdata;

    txdata.reserve(block.vtx.size());

    CAmount nValueOut = 0;

    CAmount nValueIn = 0;

    for (unsigned int i = 0; i < block.vtx.size(); i++)

    {

        const CTransaction &tx = \*(block.vtx[i]);

        nInputs += tx.vin.size();

        if (!tx.IsCoinBase())

        {

            CAmount txfee = 0;

            if (!Consensus::CheckTxInputs(tx, state, view, pindex->nHeight, txfee)) {

                return error("%s: Consensus::CheckTxInputs: %s, %s", \_\_func\_\_, tx.GetHash().ToString(), FormatStateMessage(state));

            }

            nFees += txfee;

            if (!MoneyRange(nFees)) {

                return state.DoS(100, error("%s: accumulated fee in the block out of range.", \_\_func\_\_),

                                 REJECT\_INVALID, "bad-txns-accumulated-fee-outofrange");

            }

            prevheights.resize(tx.vin.size());

            for (size\_t j = 0; j < tx.vin.size(); j++) {

                prevheights[j] = view.AccessCoin(tx.vin[j].prevout).nHeight;

            }

            if (!SequenceLocks(tx, nLockTimeFlags, &prevheights, \*pindex)) {

                return state.DoS(100, error("%s: contains a non-BIP68-final transaction", \_\_func\_\_),

                                 REJECT\_INVALID, "bad-txns-nonfinal");

            }

        }

        nSigOpsCost += GetTransactionSigOpCost(tx, view, flags);

        if (nSigOpsCost > MAX\_BLOCK\_SIGOPS\_COST)

            return state.DoS(100, error("ConnectBlock(): too many sigops"),

                             REJECT\_INVALID, "bad-blk-sigops");

        txdata.emplace\_back(tx);

        if (!tx.IsCoinBase())

        {

            if (!tx.IsCoinStake())

                nFees += view.GetValueIn(tx) - tx.GetValueOut();

            nValueIn += view.GetValueIn(tx);

            std::vector<CScriptCheck> vChecks;

            bool fCacheResults = fJustCheck; /\* Don't cache results if we're actually connecting blocks (still consult the cache, though) \*/

            if (!CheckInputs(tx, state, view, fScriptChecks, flags, fCacheResults, fCacheResults, txdata[i], nScriptCheckThreads ? &vChecks : nullptr))

                return error("ConnectBlock(): CheckInputs on %s failed with %s",

                             tx.GetHash().ToString(), FormatStateMessage(state));

            control.Add(vChecks);

        }

        nValueOut += tx.GetValueOut();

        CTxUndo undoDummy;

        if (i > 0) {

            blockundo.vtxundo.push\_back(CTxUndo());

        }

        UpdateCoins(tx, view, i == 0 ? undoDummy : blockundo.vtxundo.back(), pindex->nHeight);

        vPos.push\_back(std::make\_pair(tx.GetHash(), pos));

        pos.nTxOffset += ::GetSerializeSize(tx, SER\_DISK, CLIENT\_VERSION);

    }

    if (block.IsProofOfStake() && pindex->nHeight > 1500)

    {

       const auto& tx = block.vtx[1];

       const auto& txin = tx->vin[0].prevout.hash;

       CAmount totalCreated = nValueOut - nValueIn;

       int64\_t nCoinAge = GetStakeInputAge(txin, block.nTime);

       CAmount nCoinStakeReward = GetProofOfStakeReward(nCoinAge);

       int stakeOutputs = tx->vout.size();

       if (stakeOutputs < 5 || stakeOutputs > 6) {

           LogPrintf("Invalid stake block (incorrect amount of outputs)\n");

           return false;

       }

       CAmount expectedReward = 2 \* COIN;

       CAmount baseReward = GetBlockSubsidy(pindex->nHeight, Params().GetConsensus(), false);

       for (unsigned int i = 0; i < 3; i++)

            expectedReward += GetMasternodePayment(i, baseReward);

       expectedReward += nCoinStakeReward;

       LogPrintf("Height %d - %llu was created, %llu was expected\n", pindex->nHeight, totalCreated / COIN, expectedReward / COIN);

       if ((totalCreated/COIN) > (expectedReward/COIN)) {

          LogPrintf("Invalid stake block (reward for block is too high)\n");

          return false;

       }

       int voutOffset = stakeOutputs - 3;

       for (unsigned int i = 0; i < 3; i++) {

           CAmount mnTierReward = tx->vout[voutOffset+i].nValue;

           CAmount mnExpectedReward = GetMasternodePayment(i, baseReward);

           LogPrintf("Tier %d; expected %llu got %llu\n", i, mnTierReward / COIN, mnExpectedReward / COIN);

           if (mnTierReward != mnExpectedReward) {

               LogPrintf("Invalid stake block (tier payout tampering)\n");

               return false;

           }

       }

    }

    CAmount nMoneySupplyPrev = pindex->pprev ? pindex->pprev->nMoneySupply : 0;

    pindex->nMoneySupply = nMoneySupplyPrev + nValueOut - nValueIn;

    pindex->nMint = pindex->nMoneySupply - nMoneySupplyPrev;

    int64\_t nTime3 = GetTimeMicros(); nTimeConnect += nTime3 - nTime2;

    LogPrint(BCLog::BENCH, "      - Connect %u transactions: %.2fms (%.3fms/tx, %.3fms/txin) [%.2fs (%.2fms/blk)]\n", (unsigned)block.vtx.size(), MILLI \* (nTime3 - nTime2), MILLI \* (nTime3 - nTime2) / block.vtx.size(), nInputs <= 1 ? 0 : MILLI \* (nTime3 - nTime2) / (nInputs-1), nTimeConnect \* MICRO, nTimeConnect \* MILLI / nBlocksTotal);

    CAmount expectedReward = GetBlockSubsidy(pindex->pprev->nHeight,

                                             chainparams.GetConsensus());

    std::string strError = "";

    bool isProofOfStake = !block.IsProofOfWork();

    const auto& coinbaseTransaction = block.vtx[isProofOfStake];

    if (!IsBlockPayeeValid(coinbaseTransaction, pindex->nHeight, expectedReward, pindex->nMint)) {

        return state.DoS(0, error("ConnectBlock(XPChain): couldn't find masternode or superblock payments"),

                         REJECT\_INVALID, "bad-cb-payee");

    }

    if (!control.Wait())

        return state.DoS(100, error("%s: CheckQueue failed", \_\_func\_\_), REJECT\_INVALID, "block-validation-failed");

    int64\_t nTime4 = GetTimeMicros(); nTimeVerify += nTime4 - nTime2;

    LogPrint(BCLog::BENCH, "    - Verify %u txins: %.2fms (%.3fms/txin) [%.2fs (%.2fms/blk)]\n", nInputs - 1, MILLI \* (nTime4 - nTime2), nInputs <= 1 ? 0 : MILLI \* (nTime4 - nTime2) / (nInputs-1), nTimeVerify \* MICRO, nTimeVerify \* MILLI / nBlocksTotal);

    if (fJustCheck)

        return true;

    if (!WriteUndoDataForBlock(blockundo, state, pindex, chainparams))

        return false;

    if (!pindex->IsValid(BLOCK\_VALID\_SCRIPTS)) {

        pindex->RaiseValidity(BLOCK\_VALID\_SCRIPTS);

        setDirtyBlockIndex.insert(pindex);

    }

    if(g\_txindex) {

        if(!g\_txindex->WriteIndex(vPos)) {

            return AbortNode(state, "Failed to write transaction index");

        }

    }

    for (const CTransactionRef ptx: block.vtx) {

        const CTransaction& tx = \*ptx;

        if (tx.IsCoinBase())

            continue;

        for (const CTxIn in: tx.vin) {

            mapStakeSpent.insert(std::make\_pair(in.prevout, pindex->nHeight));

        }

    }

    for (auto it = mapStakeSpent.begin(); it != mapStakeSpent.end();) {

        if (it->second < pindex->nHeight - Params().MaxReorganizationDepth()) {

            it = mapStakeSpent.erase(it);

        }else {

            it++;

        }

    }

    assert(pindex->phashBlock);

    view.SetBestBlock(pindex->GetBlockHash());

    int64\_t nTime5 = GetTimeMicros(); nTimeIndex += nTime5 - nTime4;

    LogPrint(BCLog::BENCH, "    - Index writing: %.2fms [%.2fs (%.2fms/blk)]\n", MILLI \* (nTime5 - nTime4), nTimeIndex \* MICRO, nTimeIndex \* MILLI / nBlocksTotal);

    int64\_t nTime6 = GetTimeMicros(); nTimeCallbacks += nTime6 - nTime5;

    LogPrint(BCLog::BENCH, "    - Callbacks: %.2fms [%.2fs (%.2fms/blk)]\n", MILLI \* (nTime6 - nTime5), nTimeCallbacks \* MICRO, nTimeCallbacks \* MILLI / nBlocksTotal);

    return true;

}

bool static FlushStateToDisk(const CChainParams& chainparams, CValidationState &state, FlushStateMode mode, int nManualPruneHeight) {

    int64\_t nMempoolUsage = mempool.DynamicMemoryUsage();

    LOCK(cs\_main);

    static int64\_t nLastWrite = 0;

    static int64\_t nLastFlush = 0;

    std::set<int> setFilesToPrune;

    bool full\_flush\_completed = false;

    try {

        {

            bool fFlushForPrune = false;

            bool fDoFullFlush = false;

            LOCK(cs\_LastBlockFile);

            if (fPruneMode && (fCheckForPruning || nManualPruneHeight > 0) && !fReindex) {

                if (nManualPruneHeight > 0) {

                    FindFilesToPruneManual(setFilesToPrune, nManualPruneHeight);

                } else {

                    FindFilesToPrune(setFilesToPrune, chainparams.PruneAfterHeight());

                    fCheckForPruning = false;

                }

                if (!setFilesToPrune.empty()) {

                    fFlushForPrune = true;

                    if (!fHavePruned) {

                        pblocktree->WriteFlag("prunedblockfiles", true);

                        fHavePruned = true;

                    }

                }

            }

            int64\_t nNow = GetTimeMicros();

            if (nLastWrite == 0) {

                nLastWrite = nNow;

            }

            if (nLastFlush == 0) {

                nLastFlush = nNow;

            }

            int64\_t nMempoolSizeMax = gArgs.GetArg("-maxmempool", DEFAULT\_MAX\_MEMPOOL\_SIZE) \* 1000000;

            int64\_t cacheSize = pcoinsTip->DynamicMemoryUsage();

            int64\_t nTotalSpace = nCoinCacheUsage + std::max<int64\_t>(nMempoolSizeMax - nMempoolUsage, 0);

            bool fCacheLarge = mode == FlushStateMode::PERIODIC && cacheSize > std::max((9 \* nTotalSpace) / 10, nTotalSpace - MAX\_BLOCK\_COINSDB\_USAGE \* 1024 \* 1024);

            bool fCacheCritical = mode == FlushStateMode::IF\_NEEDED && cacheSize > nTotalSpace;

            bool fPeriodicWrite = mode == FlushStateMode::PERIODIC && nNow > nLastWrite + (int64\_t)DATABASE\_WRITE\_INTERVAL \* 1000000;

            bool fPeriodicFlush = mode == FlushStateMode::PERIODIC && nNow > nLastFlush + (int64\_t)DATABASE\_FLUSH\_INTERVAL \* 1000000;

            fDoFullFlush = (mode == FlushStateMode::ALWAYS) || fCacheLarge || fCacheCritical || fPeriodicFlush || fFlushForPrune;

            if (fDoFullFlush || fPeriodicWrite) {

                if (!CheckDiskSpace(0, true))

                    return state.Error("out of disk space");

                FlushBlockFile();

                {

                    std::vector<std::pair<int, const CBlockFileInfo\*> > vFiles;

                    vFiles.reserve(setDirtyFileInfo.size());

                    for (std::set<int>::iterator it = setDirtyFileInfo.begin(); it != setDirtyFileInfo.end(); ) {

                        vFiles.push\_back(std::make\_pair(\*it, &vinfoBlockFile[\*it]));

                        setDirtyFileInfo.erase(it++);

                    }

                    std::vector<const CBlockIndex\*> vBlocks;

                    vBlocks.reserve(setDirtyBlockIndex.size());

                    for (std::set<CBlockIndex\*>::iterator it = setDirtyBlockIndex.begin(); it != setDirtyBlockIndex.end(); ) {

                        vBlocks.push\_back(\*it);

                        setDirtyBlockIndex.erase(it++);

                    }

                    if (!pblocktree->WriteBatchSync(vFiles, nLastBlockFile, vBlocks)) {

                        return AbortNode(state, "Failed to write to block index database");

                    }

                }

                if (fFlushForPrune)

                    UnlinkPrunedFiles(setFilesToPrune);

                nLastWrite = nNow;

            }

            if (fDoFullFlush && !pcoinsTip->GetBestBlock().IsNull()) {

                if (!CheckDiskSpace(48 \* 2 \* 2 \* pcoinsTip->GetCacheSize()))

                    return state.Error("out of disk space");

                if (!pcoinsTip->Flush())

                    return AbortNode(state, "Failed to write to coin database");

                nLastFlush = nNow;

                full\_flush\_completed = true;

            }

        }

        if (full\_flush\_completed) {

            GetMainSignals().ChainStateFlushed(chainActive.GetLocator());

        }

    } catch (const std::runtime\_error& e) {

        return AbortNode(state, std::string("System error while flushing: ") + e.what());

    }

    return true;

}

void FlushStateToDisk() {

    CValidationState state;

    const CChainParams& chainparams = Params();

    if (!FlushStateToDisk(chainparams, state, FlushStateMode::ALWAYS)) {

        LogPrintf("%s: failed to flush state (%s)\n", \_\_func\_\_, FormatStateMessage(state));

    }

}

void PruneAndFlush() {

    CValidationState state;

    fCheckForPruning = true;

    const CChainParams& chainparams = Params();

    if (!FlushStateToDisk(chainparams, state, FlushStateMode::NONE)) {

        LogPrintf("%s: failed to flush state (%s)\n", \_\_func\_\_, FormatStateMessage(state));

    }

}

static void DoWarning(const std::string& strWarning)

{

    static bool fWarned = false;

    SetMiscWarning(strWarning);

    if (!fWarned) {

        AlertNotify(strWarning);

        fWarned = true;

    }

}

/\*\* Check warning conditions and do some notifications on new chain tip set. \*/

void static UpdateTip(const CBlockIndex \*pindexNew, const CChainParams& chainParams) {

    mempool.AddTransactionsUpdated(1);

    {

        WaitableLock lock(g\_best\_block\_mutex);

        g\_best\_block = pindexNew->GetBlockHash();

        g\_best\_block\_cv.notify\_all();

    }

    std::vector<std::string> warningMessages;

    if (!IsInitialBlockDownload())

    {

        int nUpgraded = 0;

        const CBlockIndex\* pindex = pindexNew;

        for (int bit = 0; bit < VERSIONBITS\_NUM\_BITS; bit++) {

            WarningBitsConditionChecker checker(bit);

            ThresholdState state = checker.GetStateFor(pindex, chainParams.GetConsensus(), warningcache[bit]);

            if (state == ThresholdState::ACTIVE || state == ThresholdState::LOCKED\_IN) {

                const std::string strWarning = strprintf(\_("Warning: unknown new rules activated (versionbit %i)"), bit);

                if (state == ThresholdState::ACTIVE) {

                    DoWarning(strWarning);

                } else {

                    warningMessages.push\_back(strWarning);

                }

            }

        }

        for (int i = 0; i < 100 && pindex != nullptr; i++)

        {

            int32\_t nExpectedVersion = ComputeBlockVersion(pindex->pprev, chainParams.GetConsensus());

            if (pindex->nVersion > VERSIONBITS\_LAST\_OLD\_BLOCK\_VERSION && (pindex->nVersion & ~nExpectedVersion) != 0)

                ++nUpgraded;

            pindex = pindex->pprev;

        }

        if (nUpgraded > 0)

            warningMessages.push\_back(strprintf(\_("%d of last 100 blocks have unexpected version"), nUpgraded));

        if (nUpgraded > 100/2)

        {

            std::string strWarning = \_("Warning: Unknown block versions being mined! It's possible unknown rules are in effect");

            DoWarning(strWarning);

        }

    }

    LogPrintf("%s: new best=%s height=%d version=0x%08x log2\_work=%.8g tx=%lu date='%s' progress=%f cache=%.1fMiB(%utxo)", \_\_func\_\_, /\* Continued \*/

              pindexNew->GetBlockHash().ToString(), pindexNew->nHeight, pindexNew->nVersion,

              log(pindexNew->nChainWork.getdouble())/log(2.0), (unsigned long)pindexNew->nChainTx,

              FormatISO8601DateTime(pindexNew->GetBlockTime()),

              GuessVerificationProgress(chainParams.TxData(), pindexNew), pcoinsTip->DynamicMemoryUsage() \* (1.0 / (1<<20)), pcoinsTip->GetCacheSize());

    if (!warningMessages.empty())

        LogPrintf(" warning='%s'", boost::algorithm::join(warningMessages, ", ")); /\* Continued \*/

    LogPrintf("\n");

}

/\*\* Disconnect chainActive's tip.

  \* After calling, the mempool will be in an inconsistent state, with

  \* transactions from disconnected blocks being added to disconnectpool.  You

  \* should make the mempool consistent again by calling UpdateMempoolForReorg.

  \* with cs\_main held.

  \*

  \* If disconnectpool is nullptr, then no disconnected transactions are added to

  \* disconnectpool (note that the caller is responsible for mempool consistency

  \* in any case).

  \*/

bool CChainState::DisconnectTip(CValidationState& state, const CChainParams& chainparams, DisconnectedBlockTransactions \*disconnectpool)

{

    CBlockIndex \*pindexDelete = chainActive.Tip();

    assert(pindexDelete);

    std::shared\_ptr<CBlock> pblock = std::make\_shared<CBlock>();

    CBlock& block = \*pblock;

    if (!ReadBlockFromDisk(block, pindexDelete, chainparams.GetConsensus()))

        return AbortNode(state, "Failed to read block");

    int64\_t nStart = GetTimeMicros();

    {

        CCoinsViewCache view(pcoinsTip.get());

        assert(view.GetBestBlock() == pindexDelete->GetBlockHash());

        if (DisconnectBlock(block, pindexDelete, view) != DISCONNECT\_OK)

            return error("DisconnectTip(): DisconnectBlock %s failed", pindexDelete->GetBlockHash().ToString());

        bool flushed = view.Flush();

        assert(flushed);

    }

    LogPrint(BCLog::BENCH, "- Disconnect block: %.2fms\n", (GetTimeMicros() - nStart) \* MILLI);

    if (!FlushStateToDisk(chainparams, state, FlushStateMode::IF\_NEEDED))

        return false;

    if (disconnectpool) {

        for (auto it = block.vtx.rbegin(); it != block.vtx.rend(); ++it) {

            disconnectpool->addTransaction(\*it);

        }

        while (disconnectpool->DynamicMemoryUsage() > MAX\_DISCONNECTED\_TX\_POOL\_SIZE \* 1000) {

            auto it = disconnectpool->queuedTx.get<insertion\_order>().begin();

            mempool.removeRecursive(\*\*it, MemPoolRemovalReason::REORG);

            disconnectpool->removeEntry(it);

        }

    }

    chainActive.SetTip(pindexDelete->pprev);

    UpdateTip(pindexDelete->pprev, chainparams);

    GetMainSignals().BlockDisconnected(pblock);

    return true;

}

bool CChainState::DisconnectBlocks(int blocks)

{

    LOCK(cs\_main);

    CValidationState state;

    const CChainParams& chainparams = Params();

    LogPrintf("DisconnectBlocks -- Got command to replay %d blocks\n", blocks);

    for(int i = 0; i < blocks; i++) {

        if(!DisconnectTip(state, chainparams, nullptr) || !state.IsValid()) {

            return false;

        }

    }

    return true;

}

static int64\_t nTimeReadFromDisk = 0;

static int64\_t nTimeConnectTotal = 0;

static int64\_t nTimeFlush = 0;

static int64\_t nTimeChainState = 0;

static int64\_t nTimePostConnect = 0;

struct PerBlockConnectTrace {

    CBlockIndex\* pindex = nullptr;

    std::shared\_ptr<const CBlock> pblock;

    std::shared\_ptr<std::vector<CTransactionRef>> conflictedTxs;

    PerBlockConnectTrace() : conflictedTxs(std::make\_shared<std::vector<CTransactionRef>>()) {}

};

class ConnectTrace {

private:

    std::vector<PerBlockConnectTrace> blocksConnected;

    CTxMemPool &pool;

public:

    explicit ConnectTrace(CTxMemPool &\_pool) : blocksConnected(1), pool(\_pool) {

        pool.NotifyEntryRemoved.connect(boost::bind(&ConnectTrace::NotifyEntryRemoved, this, \_1, \_2));

    }

    ~ConnectTrace() {

        pool.NotifyEntryRemoved.disconnect(boost::bind(&ConnectTrace::NotifyEntryRemoved, this, \_1, \_2));

    }

    void BlockConnected(CBlockIndex\* pindex, std::shared\_ptr<const CBlock> pblock) {

        assert(!blocksConnected.back().pindex);

        assert(pindex);

        assert(pblock);

        blocksConnected.back().pindex = pindex;

        blocksConnected.back().pblock = std::move(pblock);

        blocksConnected.emplace\_back();

    }

    std::vector<PerBlockConnectTrace>& GetBlocksConnected() {

        assert(!blocksConnected.back().pindex);

        assert(blocksConnected.back().conflictedTxs->empty());

        blocksConnected.pop\_back();

        return blocksConnected;

    }

    void NotifyEntryRemoved(CTransactionRef txRemoved, MemPoolRemovalReason reason) {

        assert(!blocksConnected.back().pindex);

        if (reason == MemPoolRemovalReason::CONFLICT) {

            blocksConnected.back().conflictedTxs->emplace\_back(std::move(txRemoved));

        }

    }

};

bool CChainState::ConnectTip(CValidationState& state, const CChainParams& chainparams, CBlockIndex\* pindexNew, const std::shared\_ptr<const CBlock>& pblock, ConnectTrace& connectTrace, DisconnectedBlockTransactions &disconnectpool)

{

    assert(pindexNew->pprev == chainActive.Tip());

    int64\_t nTime1 = GetTimeMicros();

    std::shared\_ptr<const CBlock> pthisBlock;

    if (!pblock) {

        std::shared\_ptr<CBlock> pblockNew = std::make\_shared<CBlock>();

        if (!ReadBlockFromDisk(\*pblockNew, pindexNew, chainparams.GetConsensus()))

            return AbortNode(state, "Failed to read block");

        pthisBlock = pblockNew;

    } else {

        pthisBlock = pblock;

    }

    const CBlock& blockConnecting = \*pthisBlock;

    int64\_t nTime2 = GetTimeMicros(); nTimeReadFromDisk += nTime2 - nTime1;

    int64\_t nTime3;

    LogPrint(BCLog::BENCH, "  - Load block from disk: %.2fms [%.2fs]\n", (nTime2 - nTime1) \* MILLI, nTimeReadFromDisk \* MICRO);

    {

        CCoinsViewCache view(pcoinsTip.get());

        bool rv = ConnectBlock(blockConnecting, state, pindexNew, view, chainparams);

        GetMainSignals().BlockChecked(blockConnecting, state);

        if (!rv) {

            if (state.IsInvalid())

                InvalidBlockFound(pindexNew, state);

            return error("ConnectTip(): ConnectBlock %s failed", pindexNew->GetBlockHash().ToString());

        }

        nTime3 = GetTimeMicros(); nTimeConnectTotal += nTime3 - nTime2;

        LogPrint(BCLog::BENCH, "  - Connect total: %.2fms [%.2fs (%.2fms/blk)]\n", (nTime3 - nTime2) \* MILLI, nTimeConnectTotal \* MICRO, nTimeConnectTotal \* MILLI / nBlocksTotal);

        bool flushed = view.Flush();

        assert(flushed);

    }

    int64\_t nTime4 = GetTimeMicros(); nTimeFlush += nTime4 - nTime3;

    LogPrint(BCLog::BENCH, "  - Flush: %.2fms [%.2fs (%.2fms/blk)]\n", (nTime4 - nTime3) \* MILLI, nTimeFlush \* MICRO, nTimeFlush \* MILLI / nBlocksTotal);

    if (!FlushStateToDisk(chainparams, state, FlushStateMode::IF\_NEEDED))

        return false;

    int64\_t nTime5 = GetTimeMicros(); nTimeChainState += nTime5 - nTime4;

    LogPrint(BCLog::BENCH, "  - Writing chainstate: %.2fms [%.2fs (%.2fms/blk)]\n", (nTime5 - nTime4) \* MILLI, nTimeChainState \* MICRO, nTimeChainState \* MILLI / nBlocksTotal);

    mempool.removeForBlock(blockConnecting.vtx, pindexNew->nHeight);

    disconnectpool.removeForBlock(blockConnecting.vtx);

    chainActive.SetTip(pindexNew);

    UpdateTip(pindexNew, chainparams);

    int64\_t nTime6 = GetTimeMicros(); nTimePostConnect += nTime6 - nTime5; nTimeTotal += nTime6 - nTime1;

    LogPrint(BCLog::BENCH, "  - Connect postprocess: %.2fms [%.2fs (%.2fms/blk)]\n", (nTime6 - nTime5) \* MILLI, nTimePostConnect \* MICRO, nTimePostConnect \* MILLI / nBlocksTotal);

    LogPrint(BCLog::BENCH, "- Connect block: %.2fms [%.2fs (%.2fms/blk)]\n", (nTime6 - nTime1) \* MILLI, nTimeTotal \* MICRO, nTimeTotal \* MILLI / nBlocksTotal);

    connectTrace.BlockConnected(pindexNew, std::move(pthisBlock));

    return true;

}

CBlockIndex\* CChainState::FindMostWorkChain() {

    do {

        CBlockIndex \*pindexNew = nullptr;

        {

            std::set<CBlockIndex\*, CBlockIndexWorkComparator>::reverse\_iterator it = setBlockIndexCandidates.rbegin();

            if (it == setBlockIndexCandidates.rend())

                return nullptr;

            pindexNew = \*it;

        }

        CBlockIndex \*pindexTest = pindexNew;

        bool fInvalidAncestor = false;

        while (pindexTest && !chainActive.Contains(pindexTest)) {

            assert(pindexTest->nChainTx || pindexTest->nHeight == 0);

            bool fFailedChain = pindexTest->nStatus & BLOCK\_FAILED\_MASK;

            bool fMissingData = !(pindexTest->nStatus & BLOCK\_HAVE\_DATA);

            if (fFailedChain || fMissingData) {

                if (fFailedChain && (pindexBestInvalid == nullptr || pindexNew->nChainWork > pindexBestInvalid->nChainWork))

                    pindexBestInvalid = pindexNew;

                CBlockIndex \*pindexFailed = pindexNew;

                while (pindexTest != pindexFailed) {

                    if (fFailedChain) {

                        pindexFailed->nStatus |= BLOCK\_FAILED\_CHILD;

                    } else if (fMissingData) {

                        mapBlocksUnlinked.insert(std::make\_pair(pindexFailed->pprev, pindexFailed));

                    }

                    setBlockIndexCandidates.erase(pindexFailed);

                    pindexFailed = pindexFailed->pprev;

                }

                setBlockIndexCandidates.erase(pindexTest);

                fInvalidAncestor = true;

                break;

            }

            pindexTest = pindexTest->pprev;

        }

        if (!fInvalidAncestor)

            return pindexNew;

    } while(true);

}

/\*\* Delete all entries in setBlockIndexCandidates that are worse than the current tip. \*/

void CChainState::PruneBlockIndexCandidates() {

    std::set<CBlockIndex\*, CBlockIndexWorkComparator>::iterator it = setBlockIndexCandidates.begin();

    while (it != setBlockIndexCandidates.end() && setBlockIndexCandidates.value\_comp()(\*it, chainActive.Tip())) {

        setBlockIndexCandidates.erase(it++);

    }

    assert(!setBlockIndexCandidates.empty());

}

bool CChainState::ActivateBestChainStep(CValidationState& state, const CChainParams& chainparams, CBlockIndex\* pindexMostWork, const std::shared\_ptr<const CBlock>& pblock, bool& fInvalidFound, ConnectTrace& connectTrace)

{

    AssertLockHeld(cs\_main);

    const CBlockIndex \*pindexOldTip = chainActive.Tip();

    const CBlockIndex \*pindexFork = chainActive.FindFork(pindexMostWork);

    bool fBlocksDisconnected = false;

    DisconnectedBlockTransactions disconnectpool;

    while (chainActive.Tip() && chainActive.Tip() != pindexFork) {

        if (!DisconnectTip(state, chainparams, &disconnectpool)) {

            UpdateMempoolForReorg(disconnectpool, false);

            return false;

        }

        fBlocksDisconnected = true;

    }

    std::vector<CBlockIndex\*> vpindexToConnect;

    bool fContinue = true;

    int nHeight = pindexFork ? pindexFork->nHeight : -1;

    while (fContinue && nHeight != pindexMostWork->nHeight) {

        int nTargetHeight = std::min(nHeight + 32, pindexMostWork->nHeight);

        vpindexToConnect.clear();

        vpindexToConnect.reserve(nTargetHeight - nHeight);

        CBlockIndex \*pindexIter = pindexMostWork->GetAncestor(nTargetHeight);

        while (pindexIter && pindexIter->nHeight != nHeight) {

            vpindexToConnect.push\_back(pindexIter);

            pindexIter = pindexIter->pprev;

        }

        nHeight = nTargetHeight;

        for (CBlockIndex \*pindexConnect : reverse\_iterate(vpindexToConnect)) {

            if (!ConnectTip(state, chainparams, pindexConnect, pindexConnect == pindexMostWork ? pblock : std::shared\_ptr<const CBlock>(), connectTrace, disconnectpool)) {

                if (state.IsInvalid()) {

                    if (!state.CorruptionPossible())

                        InvalidChainFound(vpindexToConnect.back());

                    state = CValidationState();

                    fInvalidFound = true;

                    fContinue = false;

                    break;

                } else {

                    UpdateMempoolForReorg(disconnectpool, false);

                    return false;

                }

            } else {

                PruneBlockIndexCandidates();

                if (!pindexOldTip || chainActive.Tip()->nChainWork > pindexOldTip->nChainWork) {

                    fContinue = false;

                    break;

                }

            }

        }

    }

    if (fBlocksDisconnected) {

        UpdateMempoolForReorg(disconnectpool, true);

    }

    mempool.check(pcoinsTip.get());

    if (fInvalidFound)

        CheckForkWarningConditionsOnNewFork(vpindexToConnect.back());

    else

        CheckForkWarningConditions();

    return true;

}

static void NotifyHeaderTip() {

    bool fNotify = false;

    bool fInitialBlockDownload = false;

    static CBlockIndex\* pindexHeaderOld = nullptr;

    CBlockIndex\* pindexHeader = nullptr;

    {

        LOCK(cs\_main);

        pindexHeader = pindexBestHeader;

        if (pindexHeader != pindexHeaderOld) {

            fNotify = true;

            fInitialBlockDownload = IsInitialBlockDownload();

            pindexHeaderOld = pindexHeader;

        }

    }

    if (fNotify) {

        uiInterface.NotifyHeaderTip(fInitialBlockDownload, pindexHeader);

        GetMainSignals().NotifyHeaderTip(pindexHeader, fInitialBlockDownload);

    }

}

bool CChainState::ActivateBestChain(CValidationState &state, const CChainParams& chainparams, std::shared\_ptr<const CBlock> pblock) {

    AssertLockNotHeld(cs\_main);

    CBlockIndex \*pindexMostWork = nullptr;

    CBlockIndex \*pindexNewTip = nullptr;

    int nStopAtHeight = gArgs.GetArg("-stopatheight", DEFAULT\_STOPATHEIGHT);

    do {

        boost::this\_thread::interruption\_point();

        if (GetMainSignals().CallbacksPending() > 10) {

            SyncWithValidationInterfaceQueue();

        }

        const CBlockIndex \*pindexFork;

        bool fInitialDownload;

        {

            LOCK(cs\_main);

            ConnectTrace connectTrace(mempool);

            CBlockIndex \*pindexOldTip = chainActive.Tip();

            if (pindexMostWork == nullptr) {

                pindexMostWork = FindMostWorkChain();

            }

            if (pindexMostWork == nullptr || pindexMostWork == chainActive.Tip())

                return true;

            bool fInvalidFound = false;

            std::shared\_ptr<const CBlock> nullBlockPtr;

            if (!ActivateBestChainStep(state, chainparams, pindexMostWork, pblock && pblock->GetHash() == pindexMostWork->GetBlockHash() ? pblock : nullBlockPtr, fInvalidFound, connectTrace))

                return false;

            if (fInvalidFound) {

                pindexMostWork = nullptr;

            }

            pindexNewTip = chainActive.Tip();

            pindexFork = chainActive.FindFork(pindexOldTip);

            fInitialDownload = IsInitialBlockDownload();

            for (const PerBlockConnectTrace& trace : connectTrace.GetBlocksConnected()) {

                assert(trace.pblock && trace.pindex);

                GetMainSignals().BlockConnected(trace.pblock, trace.pindex, trace.conflictedTxs);

            }

            GetMainSignals().UpdatedBlockTip(pindexNewTip, pindexFork, fInitialDownload);

            if (pindexFork != pindexNewTip) {

                uiInterface.NotifyBlockTip(fInitialDownload, pindexNewTip);

            }

        }

        if (nStopAtHeight && pindexNewTip && pindexNewTip->nHeight >= nStopAtHeight) StartShutdown();

        if (ShutdownRequested())

            break;

    } while (pindexNewTip != pindexMostWork);

    if (!IsInitialBlockDownload())

    CheckBlockIndex(chainparams.GetConsensus());

    if (!FlushStateToDisk(chainparams, state, FlushStateMode::PERIODIC)) {

        return false;

    }

    return true;

}

bool ActivateBestChain(CValidationState &state, const CChainParams& chainparams, std::shared\_ptr<const CBlock> pblock) {

    return g\_chainstate.ActivateBestChain(state, chainparams, std::move(pblock));

}

bool CChainState::PreciousBlock(CValidationState& state, const CChainParams& params, CBlockIndex \*pindex)

{

    {

        LOCK(cs\_main);

        if (pindex->nChainWork < chainActive.Tip()->nChainWork) {

            return true;

        }

        if (chainActive.Tip()->nChainWork > nLastPreciousChainwork) {

            nBlockReverseSequenceId = -1;

        }

        nLastPreciousChainwork = chainActive.Tip()->nChainWork;

        setBlockIndexCandidates.erase(pindex);

        pindex->nSequenceId = nBlockReverseSequenceId;

        if (nBlockReverseSequenceId > std::numeric\_limits<int32\_t>::min()) {

            nBlockReverseSequenceId--;

        }

        if (pindex->IsValid(BLOCK\_VALID\_TRANSACTIONS) && pindex->nChainTx) {

            setBlockIndexCandidates.insert(pindex);

            PruneBlockIndexCandidates();

        }

    }

    return ActivateBestChain(state, params, std::shared\_ptr<const CBlock>());

}

bool PreciousBlock(CValidationState& state, const CChainParams& params, CBlockIndex \*pindex) {

    return g\_chainstate.PreciousBlock(state, params, pindex);

}

bool CChainState::InvalidateBlock(CValidationState& state, const CChainParams& chainparams, CBlockIndex \*pindex)

{

    AssertLockHeld(cs\_main);

    bool pindex\_was\_in\_chain = false;

    CBlockIndex \*invalid\_walk\_tip = chainActive.Tip();

    DisconnectedBlockTransactions disconnectpool;

    while (chainActive.Contains(pindex)) {

        pindex\_was\_in\_chain = true;

        if (!DisconnectTip(state, chainparams, &disconnectpool)) {

            UpdateMempoolForReorg(disconnectpool, false);

            return false;

        }

    }

    while (pindex\_was\_in\_chain && invalid\_walk\_tip != pindex) {

        invalid\_walk\_tip->nStatus |= BLOCK\_FAILED\_CHILD;

        setDirtyBlockIndex.insert(invalid\_walk\_tip);

        setBlockIndexCandidates.erase(invalid\_walk\_tip);

        invalid\_walk\_tip = invalid\_walk\_tip->pprev;

    }

    pindex->nStatus |= BLOCK\_FAILED\_VALID;

    setDirtyBlockIndex.insert(pindex);

    setBlockIndexCandidates.erase(pindex);

    m\_failed\_blocks.insert(pindex);

    UpdateMempoolForReorg(disconnectpool, true);

    BlockMap::iterator it = mapBlockIndex.begin();

    while (it != mapBlockIndex.end()) {

        if (it->second->IsValid(BLOCK\_VALID\_TRANSACTIONS) && it->second->nChainTx && !setBlockIndexCandidates.value\_comp()(it->second, chainActive.Tip())) {

            setBlockIndexCandidates.insert(it->second);

        }

        it++;

    }

    InvalidChainFound(pindex);

    if (pindex\_was\_in\_chain) {

        uiInterface.NotifyBlockTip(IsInitialBlockDownload(), pindex->pprev);

    }

    return true;

}

bool InvalidateBlock(CValidationState& state, const CChainParams& chainparams, CBlockIndex \*pindex) {

    return g\_chainstate.InvalidateBlock(state, chainparams, pindex);

}

bool CChainState::ResetBlockFailureFlags(CBlockIndex \*pindex) {

    AssertLockHeld(cs\_main);

    int nHeight = pindex->nHeight;

    BlockMap::iterator it = mapBlockIndex.begin();

    while (it != mapBlockIndex.end()) {

        if (!it->second->IsValid() && it->second->GetAncestor(nHeight) == pindex) {

            it->second->nStatus &= ~BLOCK\_FAILED\_MASK;

            setDirtyBlockIndex.insert(it->second);

            if (it->second->IsValid(BLOCK\_VALID\_TRANSACTIONS) && it->second->nChainTx && setBlockIndexCandidates.value\_comp()(chainActive.Tip(), it->second)) {

                setBlockIndexCandidates.insert(it->second);

            }

            if (it->second == pindexBestInvalid) {

                pindexBestInvalid = nullptr;

            }

            m\_failed\_blocks.erase(it->second);

        }

        it++;

    }

    while (pindex != nullptr) {

        if (pindex->nStatus & BLOCK\_FAILED\_MASK) {

            pindex->nStatus &= ~BLOCK\_FAILED\_MASK;

            setDirtyBlockIndex.insert(pindex);

        }

        pindex = pindex->pprev;

    }

    return true;

}

bool ResetBlockFailureFlags(CBlockIndex \*pindex) {

    return g\_chainstate.ResetBlockFailureFlags(pindex);

}

static void AcceptProofOfStakeBlock(const CBlock &block, CBlockIndex \*pindexNew)

{

    if(!pindexNew)

        return;

    if (block.IsProofOfStake()) {

        pindexNew->SetProofOfStake();

        pindexNew->prevoutStake = block.vtx[1]->vin[0].prevout;

        pindexNew->nStakeTime = block.nTime;

    } else {

        pindexNew->prevoutStake.SetNull();

        pindexNew->nStakeTime = 0;

    }

    pindexNew->bnChainTrust = (pindexNew->pprev ? pindexNew->pprev->bnChainTrust : ArithToUint256(0 + pindexNew->GetBlockTrust()));

    if (!pindexNew->SetStakeEntropyBit(pindexNew->GetStakeEntropyBit()))

        LogPrintf("AcceptProofOfStakeBlock() : SetStakeEntropyBit() failed \n");

    uint256 hash = block.GetHash();

    if (pindexNew->IsProofOfStake()) {

        if (!mapProofOfStake.count(hash))

            LogPrintf("AcceptProofOfStakeBlock() : hashProofOfStake not found in map \n");

        pindexNew->hashProofOfStake = mapProofOfStake[hash];

    }

    uint64\_t nStakeModifier = 0;

    bool fGeneratedStakeModifier = false;

    if (!ComputeNextStakeModifier(pindexNew, nStakeModifier, fGeneratedStakeModifier))

        LogPrintf("AcceptProofOfStakeBlock() : ComputeNextStakeModifier() failed \n");

    pindexNew->SetStakeModifier(nStakeModifier, fGeneratedStakeModifier);

    pindexNew->nStakeModifierChecksum = GetStakeModifierChecksum(pindexNew);

    if (!CheckStakeModifierCheckpoints(pindexNew->nHeight, pindexNew->nStakeModifierChecksum)) {

        LogPrintf("AcceptProofOfStakeBlock() : Rejected by stake modifier checkpoint height=%d, modifier=%s \n", pindexNew->nHeight, std::to\_string(nStakeModifier));

        LogPrintf("pindexNew->nStakeModifierChecksum = %08x\n", pindexNew->nStakeModifierChecksum);

    }

    setDirtyBlockIndex.insert(pindexNew);

}

CBlockIndex\* CChainState::AddToBlockIndex(const CBlockHeader& block, bool fProofOfStake)

{

    AssertLockHeld(cs\_main);

    uint256 hash = block.GetHash();

    BlockMap::iterator it = mapBlockIndex.find(hash);

    if (it != mapBlockIndex.end())

        return it->second;

    CBlockIndex\* pindexNew = new CBlockIndex(block);

    pindexNew->nSequenceId = 0;

    BlockMap::iterator mi = mapBlockIndex.insert(std::make\_pair(hash, pindexNew)).first;

    pindexNew->phashBlock = &((\*mi).first);

    BlockMap::iterator miPrev = mapBlockIndex.find(block.hashPrevBlock);

    if (miPrev != mapBlockIndex.end())

    {

        pindexNew->pprev = (\*miPrev).second;

        pindexNew->nHeight = pindexNew->pprev->nHeight + 1;

        pindexNew->BuildSkip();

    }

    pindexNew->nTimeMax = (pindexNew->pprev ? std::max(pindexNew->pprev->nTimeMax, pindexNew->nTime) : pindexNew->nTime);

    if (fProofOfStake)

        pindexNew->SetProofOfStake();

    pindexNew->nChainWork = (pindexNew->pprev ? pindexNew->pprev->nChainWork : 0) + GetBlockProof(\*pindexNew);

    pindexNew->RaiseValidity(BLOCK\_VALID\_TREE);

    if (pindexBestHeader == nullptr || pindexBestHeader->nChainWork < pindexNew->nChainWork)

        pindexBestHeader = pindexNew;

    setDirtyBlockIndex.insert(pindexNew);

    return pindexNew;

}

/\*\* Mark a block as having its data received and checked (up to BLOCK\_VALID\_TRANSACTIONS). \*/

bool CChainState::ReceivedBlockTransactions(const CBlock &block, CValidationState& state, CBlockIndex \*pindexNew, const CDiskBlockPos& pos, const Consensus::Params& consensusParams)

{

    pindexNew->nTx = block.vtx.size();

    pindexNew->nChainTx = 0;

    pindexNew->nFile = pos.nFile;

    pindexNew->nDataPos = pos.nPos;

    pindexNew->nUndoPos = 0;

    pindexNew->nStatus |= BLOCK\_HAVE\_DATA;

    if (IsWitnessEnabled(pindexNew->pprev, consensusParams)) {

        pindexNew->nStatus |= BLOCK\_OPT\_WITNESS;

    }

    pindexNew->RaiseValidity(BLOCK\_VALID\_TRANSACTIONS);

    setDirtyBlockIndex.insert(pindexNew);

    if (pindexNew->pprev == nullptr || pindexNew->pprev->nChainTx) {

        std::deque<CBlockIndex\*> queue;

        queue.push\_back(pindexNew);

        while (!queue.empty()) {

            CBlockIndex \*pindex = queue.front();

            queue.pop\_front();

            pindex->nChainTx = (pindex->pprev ? pindex->pprev->nChainTx : 0) + pindex->nTx;

            {

                LOCK(cs\_nBlockSequenceId);

                pindex->nSequenceId = nBlockSequenceId++;

            }

            if (chainActive.Tip() == nullptr || !setBlockIndexCandidates.value\_comp()(pindex, chainActive.Tip())) {

                setBlockIndexCandidates.insert(pindex);

            }

            std::pair<std::multimap<CBlockIndex\*, CBlockIndex\*>::iterator, std::multimap<CBlockIndex\*, CBlockIndex\*>::iterator> range = mapBlocksUnlinked.equal\_range(pindex);

            while (range.first != range.second) {

                std::multimap<CBlockIndex\*, CBlockIndex\*>::iterator it = range.first;

                queue.push\_back(it->second);

                range.first++;

                mapBlocksUnlinked.erase(it);

            }

        }

    } else {

        if (pindexNew->pprev && pindexNew->pprev->IsValid(BLOCK\_VALID\_TREE)) {

            mapBlocksUnlinked.insert(std::make\_pair(pindexNew->pprev, pindexNew));

        }

    }

    return true;

}

static bool FindBlockPos(CDiskBlockPos &pos, unsigned int nAddSize, unsigned int nHeight, uint64\_t nTime, bool fKnown = false)

{

    LOCK(cs\_LastBlockFile);

    unsigned int nFile = fKnown ? pos.nFile : nLastBlockFile;

    if (vinfoBlockFile.size() <= nFile) {

        vinfoBlockFile.resize(nFile + 1);

    }

    if (!fKnown) {

        while (vinfoBlockFile[nFile].nSize + nAddSize >= MAX\_BLOCKFILE\_SIZE) {

            nFile++;

            if (vinfoBlockFile.size() <= nFile) {

                vinfoBlockFile.resize(nFile + 1);

            }

        }

        pos.nFile = nFile;

        pos.nPos = vinfoBlockFile[nFile].nSize;

    }

    if ((int)nFile != nLastBlockFile) {

        if (!fKnown) {

            LogPrintf("Leaving block file %i: %s\n", nLastBlockFile, vinfoBlockFile[nLastBlockFile].ToString());

        }

        FlushBlockFile(!fKnown);

        nLastBlockFile = nFile;

    }

    vinfoBlockFile[nFile].AddBlock(nHeight, nTime);

    if (fKnown)

        vinfoBlockFile[nFile].nSize = std::max(pos.nPos + nAddSize, vinfoBlockFile[nFile].nSize);

    else

        vinfoBlockFile[nFile].nSize += nAddSize;

    if (!fKnown) {

        unsigned int nOldChunks = (pos.nPos + BLOCKFILE\_CHUNK\_SIZE - 1) / BLOCKFILE\_CHUNK\_SIZE;

        unsigned int nNewChunks = (vinfoBlockFile[nFile].nSize + BLOCKFILE\_CHUNK\_SIZE - 1) / BLOCKFILE\_CHUNK\_SIZE;

        if (nNewChunks > nOldChunks) {

            if (fPruneMode)

                fCheckForPruning = true;

            if (CheckDiskSpace(nNewChunks \* BLOCKFILE\_CHUNK\_SIZE - pos.nPos, true)) {

                FILE \*file = OpenBlockFile(pos);

                if (file) {

                    LogPrintf("Pre-allocating up to position 0x%x in blk%05u.dat\n", nNewChunks \* BLOCKFILE\_CHUNK\_SIZE, pos.nFile);

                    AllocateFileRange(file, pos.nPos, nNewChunks \* BLOCKFILE\_CHUNK\_SIZE - pos.nPos);

                    fclose(file);

                }

            }

            else

                return error("out of disk space");

        }

    }

    setDirtyFileInfo.insert(nFile);

    return true;

}

static bool FindUndoPos(CValidationState &state, int nFile, CDiskBlockPos &pos, unsigned int nAddSize)

{

    pos.nFile = nFile;

    LOCK(cs\_LastBlockFile);

    unsigned int nNewSize;

    pos.nPos = vinfoBlockFile[nFile].nUndoSize;

    nNewSize = vinfoBlockFile[nFile].nUndoSize += nAddSize;

    setDirtyFileInfo.insert(nFile);

    unsigned int nOldChunks = (pos.nPos + UNDOFILE\_CHUNK\_SIZE - 1) / UNDOFILE\_CHUNK\_SIZE;

    unsigned int nNewChunks = (nNewSize + UNDOFILE\_CHUNK\_SIZE - 1) / UNDOFILE\_CHUNK\_SIZE;

    if (nNewChunks > nOldChunks) {

        if (fPruneMode)

            fCheckForPruning = true;

        if (CheckDiskSpace(nNewChunks \* UNDOFILE\_CHUNK\_SIZE - pos.nPos, true)) {

            FILE \*file = OpenUndoFile(pos);

            if (file) {

                LogPrintf("Pre-allocating up to position 0x%x in rev%05u.dat\n", nNewChunks \* UNDOFILE\_CHUNK\_SIZE, pos.nFile);

                AllocateFileRange(file, pos.nPos, nNewChunks \* UNDOFILE\_CHUNK\_SIZE - pos.nPos);

                fclose(file);

            }

        }

        else

            return state.Error("out of disk space");

    }

    return true;

}

static bool CheckBlockHeader(const CBlockHeader& block, CValidationState& state, const Consensus::Params& consensusParams, bool fCheckPOW = true)

{

    return true;

}

bool CheckBlock(const CBlock& block, CValidationState& state, const Consensus::Params& consensusParams, bool fCheckPOW, bool fCheckMerkleRoot, bool fCheckContractOutpoint)

{

    if (block.fChecked)

        return true;

    if (!CheckBlockHeader(block, state, consensusParams, fCheckPOW && block.IsProofOfWork()))

        return false;

    if (fCheckMerkleRoot) {

        bool mutated;

        uint256 hashMerkleRoot2 = BlockMerkleRoot(block, &mutated);

        if (block.hashMerkleRoot != hashMerkleRoot2)

            return state.DoS(100, false, REJECT\_INVALID, "bad-txnmrklroot", true, "hashMerkleRoot mismatch");

        if (mutated)

            return state.DoS(100, false, REJECT\_INVALID, "bad-txns-duplicate", true, "duplicate transaction");

    }

    if (block.vtx.empty() || block.vtx.size() \* WITNESS\_SCALE\_FACTOR > MAX\_BLOCK\_WEIGHT || ::GetSerializeSize(block, SER\_NETWORK, PROTOCOL\_VERSION | SERIALIZE\_TRANSACTION\_NO\_WITNESS) \* WITNESS\_SCALE\_FACTOR > MAX\_BLOCK\_WEIGHT)

        return state.DoS(100, false, REJECT\_INVALID, "bad-blk-length", false, "size limits failed");

    if (block.vtx.empty() || !block.vtx[0]->IsCoinBase())

        return state.DoS(100, false, REJECT\_INVALID, "bad-cb-missing", false, "first tx is not coinbase");

    for (unsigned int i = 1; i < block.vtx.size(); i++)

        if (block.vtx[i]->IsCoinBase())

            return state.DoS(100, false, REJECT\_INVALID, "bad-cb-multiple", false, "more than one coinbase");

    if (block.IsProofOfStake()) {

        if (block.vtx.empty() || !block.vtx[1]->IsCoinStake())

            return state.DoS(100, error("CheckBlock() : second tx is not coinstake"));

        for (unsigned int i = 2; i < block.vtx.size(); i++)

            if (block.vtx[i]->IsCoinStake())

                return state.DoS(100, error("CheckBlock() : more than one coinstake"));

    }

    for (const auto& tx : block.vtx) {

        if (!CheckTransaction(\*tx, state, true)) {

            doublespend++;

        LogPrintf("doublespend detected #%d - tx input %s \n", doublespend, tx->GetHash().ToString());

        }

    }

#if 0

            return state.Invalid(false, state.GetRejectCode(), state.GetRejectReason(),

                                 strprintf("Transaction check failed (tx hash %s) %s", tx->GetHash().ToString(), state.GetDebugMessage()));

#endif

    unsigned int nSigOps = 0;

    for (const auto& tx : block.vtx)

    {

        nSigOps += GetLegacySigOpCount(\*tx);

    }

    if (nSigOps \* WITNESS\_SCALE\_FACTOR > MAX\_BLOCK\_SIGOPS\_COST)

        return state.DoS(100, false, REJECT\_INVALID, "bad-blk-sigops", false, "out-of-bounds SigOpCount");

    if (fCheckPOW && fCheckMerkleRoot)

        block.fChecked = true;

    return true;

}

bool IsWitnessEnabled(const CBlockIndex\* pindexPrev, const Consensus::Params& params)

{

    LOCK(cs\_main);

    return (VersionBitsState(pindexPrev, params, Consensus::DEPLOYMENT\_SEGWIT, versionbitscache) == ThresholdState::ACTIVE);

}

bool IsNullDummyEnabled(const CBlockIndex\* pindexPrev, const Consensus::Params& params)

{

    LOCK(cs\_main);

    return (VersionBitsState(pindexPrev, params, Consensus::DEPLOYMENT\_SEGWIT, versionbitscache) == ThresholdState::ACTIVE);

}

static int GetWitnessCommitmentIndex(const CBlock& block)

{

    int commitpos = -1;

    if (!block.vtx.empty()) {

        const auto &commitmentTx = block.vtx[0];

        for (size\_t o = 0; o < commitmentTx->vout.size(); o++) {

            if (commitmentTx->vout[o].scriptPubKey.size() >= 38 && commitmentTx->vout[o].scriptPubKey[0] == OP\_RETURN && commitmentTx->vout[o].scriptPubKey[1] == 0x24 && commitmentTx->vout[o].scriptPubKey[2] == 0xaa && commitmentTx->vout[o].scriptPubKey[3] == 0x21 && commitmentTx->vout[o].scriptPubKey[4] == 0xa9 && commitmentTx->vout[o].scriptPubKey[5] == 0xed) {

                commitpos = o;

            }

        }

    }

    return commitpos;

}

void UpdateUncommittedBlockStructures(CBlock& block, const CBlockIndex\* pindexPrev, const Consensus::Params& consensusParams)

{

    int commitpos = GetWitnessCommitmentIndex(block);

    static const std::vector<unsigned char> nonce(32, 0x00);

    if (commitpos != -1 && IsWitnessEnabled(pindexPrev, consensusParams) && !block.vtx[0]->HasWitness()) {

        CMutableTransaction tx(\*block.vtx[0]);

        tx.vin[0].scriptWitness.stack.resize(1);

        tx.vin[0].scriptWitness.stack[0] = nonce;

        block.vtx[0] = MakeTransactionRef(std::move(tx));

    }

}

std::vector<unsigned char> GenerateCoinbaseCommitment(CBlock& block, const CBlockIndex\* pindexPrev, const Consensus::Params& consensusParams)

{

    std::vector<unsigned char> commitment;

    int commitpos = GetWitnessCommitmentIndex(block);

    std::vector<unsigned char> ret(32, 0x00);

    auto segwitState = VersionBitsState(pindexPrev, consensusParams, Consensus::DEPLOYMENT\_SEGWIT, versionbitscache);

    if (segwitState == ThresholdState::ACTIVE || segwitState == ThresholdState::LOCKED\_IN || segwitState == ThresholdState::STARTED) {

        if (commitpos == -1) {

            uint256 witnessroot = BlockWitnessMerkleRoot(block, nullptr);

            CHash256().Write(witnessroot.begin(), 32).Write(ret.data(), 32).Finalize(witnessroot.begin());

            CTxOut out;

            out.nValue = 0;

            out.scriptPubKey.resize(38);

            out.scriptPubKey[0] = OP\_RETURN;

            out.scriptPubKey[1] = 0x24;

            out.scriptPubKey[2] = 0xaa;

            out.scriptPubKey[3] = 0x21;

            out.scriptPubKey[4] = 0xa9;

            out.scriptPubKey[5] = 0xed;

            memcpy(&out.scriptPubKey[6], witnessroot.begin(), 32);

            commitment = std::vector<unsigned char>(out.scriptPubKey.begin(), out.scriptPubKey.end());

            CMutableTransaction tx(\*block.vtx[0]);

            tx.vout.push\_back(out);

            block.vtx[0] = MakeTransactionRef(std::move(tx));

        }

    }

    UpdateUncommittedBlockStructures(block, pindexPrev, consensusParams);

    return commitment;

}

/\*\* Context-dependent validity checks.

 \*  By "context", we mean only the previous block headers, but not the UTXO

 \*  set; UTXO-related validity checks are done in ConnectBlock().

 \*  NOTE: This function is not currently invoked by ConnectBlock(), so we

 \*  should consider upgrade issues if we change which consensus rules are

 \*  enforced in this function (eg by adding a new consensus rule). See comment

 \*  in ConnectBlock().

 \*  Note that -reindex-chainstate skips the validation that happens here!

 \*/

static bool ContextualCheckBlockHeader(const CBlockHeader& block, CValidationState& state, const CChainParams& params, const CBlockIndex\* pindexPrev, int64\_t nAdjustedTime)

{

    assert(pindexPrev != nullptr);

    const int nHeight = pindexPrev->nHeight + 1;

    const Consensus::Params& consensusParams = params.GetConsensus();

    if (fCheckpointsEnabled) {

        CBlockIndex\* pcheckpoint = Checkpoints::GetLastCheckpoint(params.Checkpoints());

        if (pcheckpoint && nHeight < pcheckpoint->nHeight)

            return state.DoS(100, error("%s: forked chain older than last checkpoint (height %d)", \_\_func\_\_, nHeight), REJECT\_CHECKPOINT, "bad-fork-prior-to-checkpoint");

    }

    if (!Checkpoints::CheckSync(nHeight))

        return state.DoS(1, error("%s: forked chain older than synchronized checkpoint (height %d)", \_\_func\_\_, nHeight), REJECT\_CHECKPOINT, "bad-fork-prior-to-synch-checkpoint");

    if (block.GetBlockTime() <= pindexPrev->GetMedianTimePast())

        return state.Invalid(false, REJECT\_INVALID, "time-too-old", "block's timestamp is too early");

    if (block.GetBlockTime() > nAdjustedTime + MAX\_FUTURE\_BLOCK\_TIME)

        return state.Invalid(false, REJECT\_INVALID, "time-too-new", "block timestamp too far in the future");

    if((block.nVersion < 2 && nHeight >= consensusParams.BIP34Height) ||

            (block.nVersion < 3 && nHeight >= consensusParams.BIP66Height) ||

            (block.nVersion < 4 && nHeight >= consensusParams.BIP65Height))

        return state.Invalid(false, REJECT\_OBSOLETE, strprintf("bad-version(0x%08x)", block.nVersion),

                             strprintf("rejected nVersion=0x%08x block", block.nVersion));

    return true;

}

/\*\* NOTE: We need this function in order to place the commitment into the coinstake as last CTxOut.

 \* The problem is that when we were building the witness commitment our coinstake was without extra output.

 \* We will hack here in order to get correct hash without commitment.

 \*/

static uint256 WitnessComittmentForPoSBlock(const CBlock &block, int commitpos, bool &malleated)

{

      return BlockWitnessMerkleRoot(block, &malleated);

}

/\*\* NOTE: This function is not currently invoked by ConnectBlock(), so we

 \*  should consider upgrade issues if we change which consensus rules are

 \*  enforced in this function (eg by adding a new consensus rule). See comment

 \*  in ConnectBlock().

 \*  Note that -reindex-chainstate skips the validation that happens here!

 \*/

static bool ContextualCheckBlock(const CBlock& block, CValidationState& state, const Consensus::Params& consensusParams, const CBlockIndex\* pindexPrev)

{

    const int nHeight = pindexPrev == nullptr ? 0 : pindexPrev->nHeight + 1;

    if (block.GetHash() == Params().GenesisBlock().GetHash())

        return true;

    if (!pindexPrev)

        return state.DoS(100, false, REJECT\_INVALID, "bad-pindex-prev", false, strprintf("current block is not genesis but has null previous"));

    uint256 hashProofOfStake = uint256();

    if (block.IsProofOfStake())

    {

        uint256 hash = block.GetHash();

        if(!CheckProofOfStake(block, hashProofOfStake))

           return state.DoS(100, error("CheckBlock(): check proof-of-stake failed for block %s\n", hash.ToString().c\_str()));

        if(hashProofOfStake == uint256())

           return state.DoS(100, error("CheckBlock(): check proof-of-stake failed for block %s\n", hash.ToString().c\_str()));

        if(!mapProofOfStake.count(hash))

            mapProofOfStake.insert(std::make\_pair(hash, hashProofOfStake));

    }

    int nLockTimeFlags = 0;

    if (VersionBitsState(pindexPrev, consensusParams, Consensus::DEPLOYMENT\_CSV, versionbitscache) == ThresholdState::ACTIVE) {

        nLockTimeFlags |= LOCKTIME\_MEDIAN\_TIME\_PAST;

    }

    int64\_t nLockTimeCutoff = (nLockTimeFlags & LOCKTIME\_MEDIAN\_TIME\_PAST)

            ? pindexPrev->GetMedianTimePast()

            : block.GetBlockTime();

    for (const auto& tx : block.vtx) {

        if (!IsFinalTx(\*tx, nHeight, nLockTimeCutoff)) {

            return state.DoS(10, false, REJECT\_INVALID, "bad-txns-nonfinal", false, "non-final transaction");

        }

    }

    if (nHeight >= consensusParams.BIP34Height)

    {

        CScript expect = CScript() << nHeight;

        if (block.vtx[0]->vin[0].scriptSig.size() < expect.size() ||

                !std::equal(expect.begin(), expect.end(), block.vtx[0]->vin[0].scriptSig.begin())) {

            return state.DoS(100, false, REJECT\_INVALID, "bad-cb-height", false, "block height mismatch in coinbase");

        }

    }

    bool fHaveWitness = false;

    if (VersionBitsState(pindexPrev, consensusParams, Consensus::DEPLOYMENT\_SEGWIT, versionbitscache) == ThresholdState::ACTIVE) {

        int commitpos = GetWitnessCommitmentIndex(block);

        if (commitpos != -1) {

            bool malleated = false;

            uint256 hashWitness = WitnessComittmentForPoSBlock(block, commitpos, malleated);

            if (block.vtx[0]->vin[0].scriptWitness.stack.size() != 1 || block.vtx[0]->vin[0].scriptWitness.stack[0].size() != 32) {

                return state.DoS(100, false, REJECT\_INVALID, "bad-witness-nonce-size", true, strprintf("%s : invalid witness reserved value size", \_\_func\_\_));

            }

            CHash256().Write(hashWitness.begin(), 32).Write(&block.vtx[0]->vin[0].scriptWitness.stack[0][0], 32).Finalize(hashWitness.begin());

            if (memcmp(hashWitness.begin(), &block.vtx[0]->vout[commitpos].scriptPubKey[6], 32)) {

                return state.DoS(100, false, REJECT\_INVALID, "bad-witness-merkle-match", true, strprintf("%s : witness merkle commitment mismatch", \_\_func\_\_));

            }

            fHaveWitness = true;

        }

    }

    if(block.IsProofOfStake())

    {

        bool isWitnessBlock = GetWitnessCommitmentIndex(block) != -1;

        if(!isWitnessBlock && block.vtx[0]->vout.size() != 1)

            return state.DoS(100, error("CheckBlock() : wrong number of outputs in coinbase for proof-of-stake block"));

        if (!block.vtx[0]->vout[0].IsEmpty())

            return state.DoS(100, error("CheckBlock() : coinbase output not empty for proof-of-stake block"));

    }

    if (!fHaveWitness) {

        for (const auto& tx : block.vtx) {

            if (tx->HasWitness()) {

                return state.DoS(100, false, REJECT\_INVALID, "unexpected-witness", true, strprintf("%s : unexpected witness data found", \_\_func\_\_));

            }

        }

    }

    if (GetBlockWeight(block) > MAX\_BLOCK\_WEIGHT) {

        return state.DoS(100, false, REJECT\_INVALID, "bad-blk-weight", false, strprintf("%s : weight limit failed", \_\_func\_\_));

    }

    return true;

}