bitcoind source code 3

arch meeting 2014/06/24

bhangra

立ち上がりからメッセージ処理スレッド

```
int main(int argc, char* argv[]){          //bitcoind.cpp
    noui_connect();
    fRet = AppInit(argc, argv){
        if (fDaemon){
            pid_t pid = fork();
            if (pid > 0){ // Parent process, pid is child process id
                CreatePidFile(GetPidFile(), pid);
                return true;
            }
            pid_t sid = setsid();
            fRet = AppInit2(threadGroup){   //init.cpp
                StartNode(threadGroup){  //net.cpp
        // Process messages
  threadGroup.create_thread(boost::bind(&TraceThread<void (*)()>, "msghand", &ThreadMessageHandler));

                }
            }
        }

    }

}
```

net.cpp ThreadMessageHandler

```
void ThreadMessageHandler(){
    SetThreadPriority(THREAD_PRIORITY_BELOW_NORMAL);
    while (true){                    // Poll the connected nodes for messages
      CNode* pnodeTrickle = NULL;
      if (!vNodesCopy.empty())
      pnodeTrickle = vNodesCopy[GetRand(vNodesCopy.size())];
      bool fSleep = true;
      BOOST_FOREACH(CNode* pnode, vNodesCopy){
            if (pnode->fDisconnect)
              continue;
            // Receive messages
            {
              TRY_LOCK(pnode->cs_vRecvMsg, lockRecv);
              if (lockRecv){
                  if (!g_signals.ProcessMessages(pnode))                    //受信したメッセージの処理
                      pnode->CloseSocketDisconnect();
                  if (pnode->nSendSize < SendBufferSize()){
                      if (!pnode->vRecvGetData.empty() || (!pnode->vRecvMsg.empty() && pnode->vRecvMsg[0].complete())){
                          fSleep = false;
                      }
                  }
              }
            }
            boost::this_thread::interruption_point();
            // Send messages
            {
              TRY_LOCK(pnode->cs_vSend, lockSend);
              if (lockSend)
                  g_signals.SendMessages(pnode, pnode == pnodeTrickle); //static CNodeSignals g_signals;

            }
            boost::this_thread::interruption_point();
      }
      if (fSleep)
        MilliSleep(100);
    }                                // while (true)
}
```

ThreadMessageHandler
から呼び出された
ProcessMessagesの処理

main.cpp ProcessMessages

```
// requires LOCK(cs_vRecvMsg)
bool ProcessMessages(CNode* pfrom){
    std::deque<CNetMessage>::iterator  it = pfrom->vRecvMsg.begin();
     while (!pfrom->fDisconnect && it != pfrom->vRecvMsg.end()) {
            CNetMessage& msg = *it;                          // get next message
            it++;
                                                             // Scan for message start
            if (memcmp(msg.hdr.pchMessageStart, Params().MessageStart(), MESSAGE_START_SIZE) != 0) {
                    break;
            }
            CMessageHeader& hdr = msg.hdr;                   // Read header
             string strCommand = hdr.GetCommand();
            CDataStream& vRecv = msg.vRecv;                  // Checksum
            uint256 hash = Hash(vRecv.begin(), vRecv.begin() + nMessageSize);
            unsigned int nChecksum = 0;
            memcpy(&nChecksum, &hash, sizeof(nChecksum));
             bool fRet = false;
            try
            {
                    fRet = ProcessMessage(pfrom, strCommand, vRecv);
                    boost::this_thread::interruption_point();
            }
```

main.cpp ProcessMessage

```
bool static ProcessMessage(CNode* pfrom, string strCommand, CDataStream& vRecv){
    if (strCommand == "version")
    {
    }
    else if (strCommand == "inv")
    {
    }
    else if (strCommand == "getblocks")
    {
    }
    else if (strCommand == “tx”)
    {
    }
    else if (strCommand == "block" && !fImporting && !fReindex)
    {
    }
}
```

main.cpp strCommand==“inv”

```
else if (strCommand == "inv"){
    vector<CInv> vInv;
    vRecv >> vInv;
    unsigned int nLastBlock = (unsigned int)(-1);                    // find last block in inv vector
    for (unsigned int nInv = 0; nInv < vInv.size(); nInv++) {
      if (vInv[vInv.size() - 1 - nInv].type == MSG_BLOCK) {
        nLastBlock = vInv.size() - 1 - nInv;
        break;
      }
    }
    LOCK(cs_main);
    for (unsigned int nInv = 0; nInv < vInv.size(); nInv++)       {
      const CInv &inv = vInv[nInv];
      boost::this_thread::interruption_point();
      pfrom->AddInventoryKnown(inv);
      bool fAlreadyHave = AlreadyHave(inv);
      LogPrint("net", "  got inventory: %s  %s¥n", inv.ToString(), fAlreadyHave ? "have" : "new");
      if (!fAlreadyHave) {                                           //invにあるブロックを持ってないなら
        if (!fImporting && !fReindex)
          pfrom->AskFor(inv);
      } else if (inv.type == MSG_BLOCK && mapOrphanBlocks.count(inv.hash)) {
        PushGetBlocks(pfrom, chainActive.Tip(), GetOrphanRoot(inv.hash));
      } else if (nInv == nLastBlock) {
        // In case we are on a very long side-chain, it is possible that we already have
        // the last block in an inv bundle sent in response to getblocks. Try to detect
        // this situation and push another getblocks to continue.
        PushGetBlocks(pfrom, mapBlockIndex[inv.hash], uint256(0));
        if (fDebug)
          LogPrintf("force request: %s¥n", inv.ToString());
      }
      // Track requests for our stuff
      g_signals.Inventory(inv.hash);
    }
}
```

main.cpp AlreadyHave

```
bool static AlreadyHave(const CInv& inv)
{
    switch (inv.type)
    {
    case MSG_TX:
        {
            bool txInMap = false;
            txInMap = mempool.exists(inv.hash);
            return txInMap || mapOrphanTransactions.count(inv.hash) ||
                pcoinsTip->HaveCoins(inv.hash);
        }
    case MSG_BLOCK:
        return mapBlockIndex.count(inv.hash) ||
               mapOrphanBlocks.count(inv.hash);
    }
    // Don't know what it is, just say we already got one
    return true;
}
```

main.cpp strCommand==“block”

```
else if (strCommand == "block" && !fImporting && !fReindex) // Ignore blocks received while
    importing
{
    CBlock block;
    vRecv >> block;

    LogPrint("net", "received block %s¥n", block.GetHash().ToString());
    // block.print();

    CInv inv(MSG_BLOCK, block.GetHash());
    pfrom->AddInventoryKnown(inv);

    LOCK(cs_main);
    // Remember who we got this block from.
    mapBlockSource[inv.hash] = pfrom->GetId();

    CValidationState state;
    ProcessBlock(state, pfrom, &block);
}
```

main.cpp **ProcessBlock** 1

```
bool ProcessBlock(CValidationState &state, CNode* pfrom, CBlock* pblock, CDiskBlockPos *dbp){
    AssertLockHeld(cs_main);
    uint256 hash = pblock->GetHash();
    if (!CheckBlock(*pblock, state)) {                                    // Preliminary checks
        if (state.CorruptionPossible())
            mapAlreadyAskedFor.erase(CInv(MSG_BLOCK, hash));
        return error("ProcessBlock() : CheckBlock FAILED");

    }
 // If we don't already have its previous block, shunt it off to holding area until we get it
if (pblock->hashPrevBlock != 0 && !mapBlockIndex.count(pblock->hashPrevBlock)) {
        LogPrintf("ProcessBlock: ORPHAN BLOCK %lu, prev=%s¥n", (unsigned long)mapOrphanBlocks.size(), pblock->hashPrevBlock.ToString())
        if (pfrom) {                                                      // Accept orphans as long as there is a node to request its parents from
            PruneOrphanBlocks();
            COrphanBlock* pblock2 = new COrphanBlock();
            {
                CDataStream ss(SER_DISK, CLIENT_VERSION);
                ss << *pblock;
                pblock2->vchBlock = std::vector<unsigned char>(ss.begin(), ss.end());
            }
            pblock2->hashBlock = hash;
            pblock2->hashPrev = pblock->hashPrevBlock;
            mapOrphanBlocks.insert(make_pair(hash, pblock2));
            mapOrphanBlocksByPrev.insert(make_pair(pblock2->hashPrev, pblock2));
            PushGetBlocks(pfrom, chainActive.Tip(), GetOrphanRoot(hash));          // Ask this guy to fill in what we're missing
        }
        return true;
    }
```

main.cpp **ProcessBlock** 2

```
  if (!AcceptBlock(*pblock, state, dbp))                    // Store to disk
    return error("ProcessBlock() : AcceptBlock FAILED");
  vector<uint256> vWorkQueue;
  vWorkQueue.push_back(hash);                               // Recursively process any orphan blocks that depended on this one
  for (unsigned int i = 0; i < vWorkQueue.size(); i++)    {
    uint256 hashPrev = vWorkQueue[i];
    for (multimap<uint256, COrphanBlock*>::iterator mi = mapOrphanBlocksByPrev.lower_bound(hashPrev);
         mi != mapOrphanBlocksByPrev.upper_bound(hashPrev);
         ++mi)
    {
      CBlock block;
      {
        CDataStream ss(mi->second->vchBlock, SER_DISK, CLIENT_VERSION);
        ss >> block;
      }
      block.BuildMerkleTree();
      // Use a dummy CValidationState so someone can't setup nodes to counter-DoS based on orphan resolution (that is, feeding people an
     invalid block based on LegitBlockX in order to get anyone relaying LegitBlockX banned)
      CValidationState stateDummy;
      if (AcceptBlock(block, stateDummy))
        vWorkQueue.push_back(mi->second->hashBlock);
      mapOrphanBlocks.erase(mi->second->hashBlock);
      delete mi->second;
    }
    mapOrphanBlocksByPrev.erase(hashPrev);
  }
  LogPrintf("ProcessBlock: ACCEPTED¥n");
  return true;
}
```

main.cpp AcceptBlock 1

```
bool AcceptBlock(CBlock& block, CValidationState& state, CDiskBlockPos* dbp){      // Check for duplicate
    uint256 hash = block.GetHash();
    if (mapBlockIndex.count(hash))
        return state.Invalid(error("AcceptBlock() : block already in mapBlockIndex"), 0, "duplicate");
    CBlockIndex* pindexPrev = NULL;
    int nHeight = 0;
    if (hash != Params().HashGenesisBlock()) {                                      // Get prev block index
        map<uint256, CBlockIndex*>::iterator mi = mapBlockIndex.find(block.hashPrevBlock);
        if (mi == mapBlockIndex.end())
            return state.DoS(10, error("AcceptBlock() : prev block not found"), 0, "bad-prevblk");
        pindexPrev = (*mi).second;
        nHeight = pindexPrev->nHeight+1;
        if (block.nBits != GetNextWorkRequired(pindexPrev, &block))                 // Check proof of work
            return state.DoS(100, error("AcceptBlock() : incorrect proof of work"),
                             REJECT_INVALID, "bad-diffbits");
        if (block.GetBlockTime() <= pindexPrev->GetMedianTimePast())                // Check timestamp against prev
            return state.Invalid(error("AcceptBlock() : block's timestamp is too early"),
                                 REJECT_INVALID, "time-too-old");
        BOOST_FOREACH(const CTransaction& tx, block.vtx)                            // Check that all transactions are finalized
            if (!IsFinalTx(tx, nHeight, block.GetBlockTime()))
                return state.DoS(10, error("AcceptBlock() : contains a non-final transaction"),
                                 REJECT_INVALID, "bad-txns-nonfinal");
                                                                                    // Check that the block chain matches the known block chain up to a checkpoint
        if (!Checkpoints::CheckBlock(nHeight, hash))
            return state.DoS(100, error("AcceptBlock() : rejected by checkpoint lock-in at %d", nHeight),
                             REJECT_CHECKPOINT, "checkpoint mismatch");
```

main.cpp AcceptBlock 2

```
 // Don't accept any forks from the main chain prior to last checkpoint
CBlockIndex* pcheckpoint = Checkpoints::GetLastCheckpoint(mapBlockIndex);
if (pcheckpoint && nHeight < pcheckpoint->nHeight)
    return state.DoS(100, error("AcceptBlock() : forked chain older than last checkpoint (height %d)", nHeight));

// Reject block.nVersion=1 blocks when 95% (75% on testnet) of the network has upgraded:
if (block.nVersion < 2){
    if ((!TestNet() && CBlockIndex::IsSuperMajority(2, pindexPrev, 950, 1000)) || (TestNet() && CBlockIndex::IsSuperMajority(2, pindexPrev
 75, 100))){
        return state.Invalid(error("AcceptBlock() : rejected nVersion=1 block"),
         REJECT_OBSOLETE, "bad-version");
    }
}

// Enforce block.nVersion=2 rule that the coinbase starts with serialized block height
if (block.nVersion >= 2){
    // if 750 of the last 1,000 blocks are version 2 or greater (51/100 if testnet):
  A
    if ((!TestNet() && CBlockIndex::IsSuperMajority(2, pindexPrev, 750, 1000)) ||  (TestNet() && CBlockIndex::IsSuperMajority(2, pindexPre
 51, 100))) {
        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, error("AcceptBlock() : block height mismatch in coinbase"),
                             REJECT_INVALID, "bad-cb-height");
    }
}
}
```

main.cpp AcceptBlock 3

```
    try {                                                                    //Write block to history file
        unsigned int nBlockSize = ::GetSerializeSize(block, SER_DISK, CLIENT_VERSION);
        CDiskBlockPos blockPos;
        if (dbp != NULL)
                blockPos = *dbp;
        if (!FindBlockPos(state, blockPos, nBlockSize+8, nHeight, block.nTime, dbp != NULL))
                return error("AcceptBlock() : FindBlockPos failed");
        if (dbp == NULL)
            if (!WriteBlockToDisk(block, blockPos))
                return state.Abort(_("Failed to write block"));
        if (!AddToBlockIndex(block, state, blockPos))
                return error("AcceptBlock() : AddToBlockIndex failed");
    } catch(std::runtime_error &e) {
        return state.Abort(_("System error: ") + e.what());
    }
                                        // Relay inventory, but don't relay old inventory during initial block download
    int nBlockEstimate = Checkpoints::GetTotalBlocksEstimate();
    if (chainActive.Tip()->GetBlockHash() == hash){
        LOCK(cs_vNodes);
        BOOST_FOREACH(CNode* pnode, vNodes)
            if (chainActive.Height() > (pnode->nStartingHeight != -1 ? pnode->nStartingHeight - 2000 : nBlockEstimate))
                pnode->PushInventory(CInv(MSG_BLOCK, hash));
    }
    return true;
}
```