

FTPMAN Memory Leak

Problem analysis

Tue, Oct 15, 2002

From analysis of allocated memory blocks on Page K, it seems that at least two MiniBooNE front ends, node0640 and node0641, exhibit small blocks of memory that are not being freed. The contents of these memory blocks point the finger at FTPMAN, as they are used in support of 1 KHz digitizer snapshots.

From experience, one possible reason for such a leak can be the failure of a call to free memory because of the current task does not “own” the memory block, a concept that pSOS supports. A quick look at this problem earlier found nothing, but perhaps it was not looked at carefully enough.

There are a few tricky cases that should be explored. What task is the current task when an FTPMAN snapshot request is canceled? There are several possibilities. The first is that the ACReq task may be active, because a new request, or a cancel message, arrives that specifies the same message-id from the same source node as a currently active request. A second case is the arrival of a continuous or snapshot request when the same requesting task name, as found in the message itself, is already in use with another request; if so, the existing active request is canceled.

When any FTPMAN request arrives, it is processed first by ANet task, which passes it to a destination message queue according to a matching entry in the NETCT (Net Connect Table), which for FTPMAN is the same message queue on which the ACReq task awaits. When ACReq notices that the message is neither a RETDAT nor SETDAT message, it calls CHKPROTO, which invokes the LA using a match in the PROTO table.

The third case occurs when an ICMP “port unreachable” message is received because the client task “goes away” while a request is active. The SNAP task is active, because it supports interpretation of such messages. It calls CANCHAIN, which in turn calls CANACNET, which fakes a cancel message by preparing a block to hold it in the same way that a completed datagram block is used. It then passes a reference to this datagram to the message queue according to the relevant entry in the NETCT. This should wake up ACReq, since it awaits this same message queue, so that ACReq will be the running task when the internal cancel message is processed.

The fourth case is one that occurs because of the REQM local application, when it determines that the client computer is no longer responsive to dummy request messages targeting the null task (task name = 0x00000000), for which the expected response is a “no such task” Acnet error message. In order to cause a cancel to take place, REQM also calls CANCHAIN to get the job done.

For all cases, the code in FTPMAN that releases memory must take care that the current task owns the memory block. To that end, there are calls to Grab in order to “grab” ownership from the task currently owning the block. When the original allocation is made, ACReq is always the owning task, by means of the logic mentioned above, in which “net” calls to FTPMAN are made from ACReq, because the message queue found in NETCT is the one upon which ACReq awaits.

Included in the FTPMAN local application is a routine called update (not to be confused with

the update task) that is called to update active requests. Look for a possibility that the cancel flag may be set by any of this code, or by any other code. When the cancel flag is set in a field of the FTPMAN request block, the 15 Hz cycle code, which is called by the update task via Data Access Table processing, will notice it and perform the cancel to free the memory resources. The reason for using a cancel flag in this way is that the update routine code does not have the proper context—it cannot access FTPMAN static variables—to cancel the request. The update routine code has only a single argument that is a pointer to an active FTPMAN request block, so it has no way to refer to the FTPMAN static variables. A pointer to the update routine is registered in the PROTO table to be invoked by update task code to fulfill ordinary requests, or by server task code to fulfill server requests. It cannot release request-ids, for example, because they reside in the FTPMAN static variables structure. Only what it can find in the allocated request block is available to do its work. (Of course, if a pointer to the FTPMAN static variables were included in the request block, it would be possible to gain such access.)

Examination of the situation

Snapshot requests were made using the snapshot test client page application PAGESNAP in node0562. Node0571 was the target node, and results were monitored by node0509. When a snapshot request was made, four memory blocks were allocated in the target node: the request block, the reply message block, one snapshot data block, and the kHz data context block. Upon exit from the test page, all four blocks were freed. This means that the code does know how to do it correctly. Something unusual may be happening that causes the kHz context block not to be freed. The kHz context block has the following structure:

```
KHzCtxType= { kHz digitizer context block }
RECORD
  kHzCBHdr: MBHdrType; { memory block header }

  lStat: WStates; { last state/status }
  cycleFlag: Boolean; { marks 15Hz cycle case }
  residuum: Integer; { #slots already reached while not yet reaching kHzPer }
  sStep: Integer; { current slot step size }
  slot: Integer; { current slot in circular memory buffer --range 0-511 }

  regBaseP: RegArrayPtr; { ptr to registers of analog IP board }
  memBaseA: Longint; { base address of 64KB memory on analog IP board }

  slotTimeP: SlotTimePtr; { ptr to array of slot times }
  evtTimeP: EvtTimePtr; { ptr to array of clock event time records }

  evtNum: Integer; { trigger clock event# }
  delay: Integer; { delay time in slots }
  kHzStep: Integer; { step size in slots }
  nPts: Integer; { number of data points to collect }

  evtTim: Longint; { time of triggering event, if any }
  kHzPer: Integer; { kHz digitizer period in us }
  nPtsCnt: Integer; { #data points collected so far }

  delayCnt: Integer; { delay count in cycles }
END;
```

From an examination of the program logic, the Release routine is called to release associated

memory blocks when canceling a request. Its single argument is a pointer to the active request block that is being canceled. First, the reply block is freed, or it is marked to be freed by the `QMonitor` task in the case it is currently “queued to the network” and therefore busy. In terms of ownership, it first attempts to grab ownership from `QMon`. If unsuccessful, it tries to grab ownership from `ACRq`. Assuming that one or the other was correct, it frees the block. Second, all snapshot data blocks are freed. Grab ownership from `ACRq` if necessary. Third, in case of a kHz request, free the kHz context block, grabbing ownership from `ACRq` if necessary. Finally, free the request block, grabbing ownership from `ACRq` if necessary.

The owners of the various blocks are not a mystery. When they are created, the `ACRq` task is the owner, since it creates them all during request initialization. A network message block, when queued to the network successfully, has its ownership passed to the `QMonitor` task. This means that when it comes time to cancel a request in the case analyzed here, one can expect that the reply block may be owned by `QMon`, unless it is still owned by `ACRq`. The request block, the snapshot data blocks, and the kHz context block all are owned by `ACRq`. Accordingly, it seems that block ownership is handled properly by the `Release` routine.

Eureka

While poring over the FTPMAN source code logic, a “eureka moment” ensued.

Solution

For a kHz request, there is to be allocated one snapshot data block for each of up to 4 devices. But there is to be only a single kHz context block. Within the request block, there is one pointer for each of the snapshot data blocks required, but there is only a single pointer to a context block. Unfortunately, the code in the `SnapshotKHz` module, which is called once for each device, allocates a kHz context block each time it is called; thus, there can be more than one context block allocated, even though only a single pointer is available to remember it. When a snapshot request is canceled, each of the snapshot data blocks is freed, and the single known context block is freed. Any other allocated context blocks are “lost in the shuffle.”

For the test client, this problem did not show up, since it can only specify a single device.

To fix this problem for now, only allocate a context block when one has not already been allocated; i.e., only allocate if the context block pointer is `NIL`. This will ensure that only one such block is allocated when initializing a kHz snapshot request.

Final solution

The fix described above is the easiest path toward getting rid of the memory leak. But it is not quite correct. What we really need is a separate context block for each device, since it is possible that more than one block of 64 channels could exist within one front end. When the new HRMs are supported, there could be more than one attached to the same front end. Even in the IRM, it is possible to have more than one 64-channel analog interface, although this has so far happened only twice. So, the solution is not the final one, but it should eliminate the present memory leak. Once we design new snapshot support for the HRMs, it will be necessary to revisit this logic and thereby implement a better fix.