Discussion about additional progress-rules in MPI-4.1

Rolf Rabenseifner
rabenseifner@hlrs.de

University of Stuttgart
High-Performance Computing-Center Stuttgart (HLRS)
www.hlrs.de

Progress rule for MPI_Request_get_status (1/3)

- MPI-4.0 page 83

3.7.6 Non-Destructive Test of status

This call is useful for accessing the information associated with a request, without freeing the request (in case the user is expected to access it later). It allows one to layer libraries more conveniently, since multiple layers of software may access the same completed request and extract from it the status information.

MPI_REQUEST_GET_STATUS(request, flag, status)

IN	request	request (handle)
OUT	flag	boolean flag, same as from MPI_TEST (logical)
OUT	status	status object if ag is true (status)

C binding …

Sets flag = true if the operation is complete, and, if so, returns in status the request status. However, unlike test or wait, it does not deallocate or inactivate the request; a subsequent call to test, wait or free should be executed with that request. It sets flag = false if the operation is not complete.

One is allowed to call MPI_REQUEST_GET_STATUS with a null or inactive request argument. In such a case the procedure returns with flag = true and empty status.

Dan and Tony want to priovide a new Issue + PR to clarify those parts

- Proposed text (added at the end of existing definition) for MPI-4.1 – Issue 468 (K) – PR 564

The progress rule for MPI_TEST, as described in Section 3.7.4, also applies to MPI_REQUEST_GET_STATUS.

Progress rule for MPI_Request_get_status (2/3)

- Proposed text (added at the end of existing definition) for MPI-4.1 – Issue 468 (K) – PR 564

  The progress rule for MPI_TEST, as described in Section 3.7.4, also applies to MPI_REQUEST_GET_STATUS.

- And change-log entry:

  Section 3.7.6 on page 83.
  The progress rule of MPI_REQUEST_GET_STATUS was clarified.

- MPI-4.0 Section 3.7.4, page 75

  3.7.4 Semantics of Nonblocking Communications

  Progress A call to MPI_WAIT … …

  If an MPI_TEST that completes a receive is repeatedly called with the same arguments, and a matching send has been started, then the call will eventually return flag = true, unless the send is satisfied by another receive. If an MPI_TEST that completes a send is repeatedly called with the same arguments, and a matching receive has been started, then the call will eventually return flag = true, unless the receive is satisfied by another send.

June 2, 2021: Straw vote:
- 6xS: This short sentence as above
- 6xL: Or long version as with MPI_TEST and TEST substituted by REQUEST_GET_STATUS
- 5x Abstain

→ no clear WG straw vote
→ decision by straw vote during June 2021 meeting by whole forum

Proposed voting alternative:
- Regular reading and 2 votes → A future global progress solution may later change all voted text 4.0 and 4.1 progress texts.
- Only vote on the rule itself, not on the text change itself (→ current rules require voting on text)
- Only reading and straw vote (→ 1st+2nd votes only if there is no better global solution)
- Only reading + 1st vote (→ 2nd vote only if there is no better global solution)

Progress rule for MPI_Request_get_status (3/3)

Preparation of a straw vote – never done because we decided to hold the vote for one meeting, because Dan and Tony want to provide a text update, see slide 2

- Short version:

  The progress rule for MPI_TEST, as described in Section 3.7.4, also applies to MPI_REQUEST_GET_STATUS.

- Long version:

  If an MPI_REQUEST_GET_STATUS that is called with a request handle of a receive operation is repeatedly called with the same arguments, and a matching send has been started, then the call will eventually return flag = true, unless the send is satisfied by another receive. If an MPI_REQUEST_GET_STATUS is called with a request handle of a send operation is repeatedly called with the same arguments, and a matching receive has been started, then the call will eventually return flag = true, unless the receive is satisfied by another send.

- No text with the risk that readers (users and implementors) do not understand whether MPI_REQUEST_GET_STATUS is required to do progress and the forum has not decided about.

- Abstain

Progress rule for MPI_Win_test

- MPI-4.0 page 600-601

12.5.2 General Active Target Synchronization

…

MPI_WIN_TEST(win, flag)

IN	win	window object (handle)
OUT	flag	success flag (logical)

C binding …

This is the nonblocking version of MPI_WIN_WAIT. It returns flag = true if all accesses to the local window by the group to which it was exposed by the corresponding MPI_WIN_POST call have been completed as signalled by matching MPI_WIN_COMPLETE calls, and flag = false otherwise. In the former case MPI_WIN_WAIT would have returned immediately. The effect of return of MPI_WIN_TEST with flag = true is the same as the effect of a return of MPI_WIN_WAIT. If flag = false is returned, then the call has no visible effect.

MPI_WIN_TEST should be invoked only where MPI_WIN_WAIT can be invoked. Once the call has returned flag = true, it must not be invoked anew, until the window is posted anew.

Assume that window …

- Two problems:
  1. "It returns flag = true if" does not mention any delay. Better: "Eventually" … in a "loop"
  2. The progress rule for MPI_Test only mentions point-to-point communication. However such progress must be always provided for all chapters of MPI.

Progress rule for MPI_Win_test – Problem 1

- Problem 1 (detected by Martin Schulz)

commA and commB should have same ranks for processes rank 0 and 1
winA is a window handle based on commA

```
if(myrank==0) { // target
      MPI_Win_post(group_of_commA_rank_1_only, /assert/ 0, winA);
      MPI_Recv(/*buf/..., /source/ 1, ..., commB, ...);
                                  // "all accesses … by matching MPI_WIN_COMPLETE" is now guaranteed
      MPI_Win_test(winA, &flag); // must return true according to current text "It returns flag = true if"
      if (!flag) {
            printf("Broken MPI library according to definition of MPI_Win_test in MPI-3.0 and 4.0\n");
            MPI_Win_wait(winA);
      }
} else if (myrank==1) { // origin
      MPI_Win_start(group_of_commA_rank_0_only, /assert/ 0, winA);
      MPI_Win_complete(winA); // has somehow to sent a flag to rank 0 in winA=commA
      MPI_Send(/token/..., /dest/ 0, ..., commB);
}
```

- No MPI library can guarantee flag = true in such case!
  - commA and commB may have significantly different latencies.
  - If commA == commB then still there is no guarantee for different communications (here **MPI_Win_complete** and **MPI_Send message** sent to rank 0) being delivered in timely order.

Progress rule for MPI_Win_test – Problem 1 – Solution

- Solution to Problem 1, see – Issue 499 (O) – PR 591 for MPI-4.1

Substitute

It returns flag = true if all accesses to the local window by the group to which it was exposed by the corresponding MPI_WIN_POST call have been completed as signalled by matching MPI_WIN_COMPLETE calls, and flag = false otherwise.
In the former case MPI_WIN_WAIT would have returned immediately.

by

Repeated calls to MPI_WIN_TEST with the same win argument will eventually return flag = true once all accesses to the local window by the group to which it was exposed by the corresponding MPI_WIN_POST call have been completed as indicated by matching MPI_WIN_COMPLETE calls,
i.e., once a call to MPI_WIN_WAIT instead of MPI_WIN_TEST would behave as if it were local.
Otherwise it returns flag=false.

June 2, 2021: Proposal was seen as needed and not controversial.
To be additionally mentioned in

- Change-log „clarifying“ – proposal as for MPI_Request_get_status:
  Sections 12.5.2 and 18.2.1 on pages 598 and 789.
  The progress rule of MPI_WIN_TEST was clarified.
- 18.2.1 Warnings Starting in MPI-4.1
  The definition of MPI_WIN_TEST in MPI-4.0 (or earlier) could be miss-interpreted and was therefore clarified.

Progress rule for MPI_Win_test – Problem 2

- Problem 2 (detected by Dan Holmes)

```
MPI_INIT
if (0==myrank) {
  MPI_WIN_CREATE(myAckSize)
  MPI_WIN_POST(win)
  MPI_ATTACH(myHugeBuffer)
  MPI_BSEND(myHugeMessage)
  sleep(10)
  flag=0; cnt=0; while(!flag) {cnt++; MPI_WIN_TEST(&flag, win)}
  print cnt
  MPI_DETACH(myHugeBuffer)
} else if (1==myrank) {
  MPI_WIN_CREATE(zeroSize)
  MPI_RECV(myHugeMessage)
  MPI_WIN_START(win)
  MPI_PUT(myAck)
  MPI_WIN_COMPLETE(win)
}
MPI_FINALIZE
```

- Which sentence in the MPI standard requires that a real MPI implementation must do progress for any ongoing operation while in a loop of MPI_Win_test?

Main progress rule – 1st draft
Progress rule for MPI_Win_test – Problem 2

- A 1st draft for a main progress rule in the terms chapter:

  For main progress rule (and only for this purpose) the term
  “xxxxx" is defined as follow:

  With an “xxxxx",
  - all MPI procedure calls that block are returning after some arbitrary delay after the semantically reason for blocking is resolved (for example an MPI_RECV is expected to return when the corresponding MPI_SEND, MPI_BSEND or MPI_SSEND is called), and
  - that the test routines MPI_TEST, MPI_REQUEST_GET_STATUS, MPI_IPROBE, MPI_IMPROBE, MPI_PARRIVED, and MPI_WIN_TEST called in a loop will eventually return flag=true after the semantically reason for returning flag=false is resolved.

  The main progress rule for (real) MPI implementations is:

  If an application is deadlock free when using an “xxxxx" then it is required that a real implementation of MPI must also execute this application deadlock-free.

  Advice to users. A real MPI implementation need not to be an “xxxxx".
  See for example the rationale in the progress section on one-sided communication (MPI-3.1 Sect.11.7.3).
  End of advice to users.

June 2, 2021: If this wording is used then xxxxx should be as neutral as possible
- xxxxx = always progressing MPI implementation
- E.g., in the AtoU: An xxxxx may have significant performance drawbacks compared to real MPI implementations.

How to tell the story to the users?
- As above? Or with other wording?

Is this story correct? → Problem with UNIFIED memory → see Dan‘s 2nd example on next slide
Do we really want this type of rule?
- Or should we better define something like minimal progress requirements
  as we already have distributed to many locations in the MPI standard text (binding, rationales, advices to implementors)
  → see 2nd draft

Progress rule for MPI_Win_test – Problem 3
(short version from Dan Holmes, extended to shared memory)

Counter-example to proposed "1st draft of a main progress rule"
- **Deadlock-free with an always progressing MPI library**
- **Deadlock with MPI libraries progressing MPI_Bsend only in MPI calls**

Or is it just wrong?

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv) {
  volatile int myAck=222;
  int myWorldRank, smSize, myRank=-1, tag=101, ranks[1], flag=0, size=1000*1000*1000, count=size/sizeof(double);
  void *pTemp;     MPI_Comm comm_sm;     MPI_Win win;     MPI_Group grp_sm, grpOther;
  double *myHugeMessage = malloc(size);    char *myHugeBuffer = malloc(size+MPI_BSEND_OVERHEAD);

  MPI_Init(&argc, &argv);
  MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL, &comm_sm);
  MPI_Comm_rank(MPI_COMM_WORLD, &myWorldRank);   MPI_Comm_rank(comm_sm, &myRank);
  MPI_Comm_size(comm_sm, &smSize);                MPI_Comm_group(comm_sm, &grp_sm);
  if (smSize < 2) {printf(("[rank:%d] smSize=%d < 2 → Abort\n", myWorldRank, smSize); MPI_Abort(comm_sm, MPI_ERR_SIZE);}
  MPI_Win_create(&myAck, sizeof(int), sizeof(int), MPI_INFO_NULL, comm_sm, &win); // -> memory_model is MPI_WIN_UNIFIED

  if (0==myRank) {
    ranks[0] = 1;
    MPI_Group_incl(grp_sm, 1, ranks, &grpOther);
    MPI_Buffer_attach(&(myHugeBuffer[0]), size+MPI_BSEND_OVERHEAD);
    MPI_Bsend(&myHugeMessage[0], count, MPI_DOUBLE, 1, tag,
              comm_sm);
    sleep(10); // to guarantee that the while starts after rank 1 is blocked in Recv
    while(myAck!=tag);
    MPI_Buffer_detach(&pTemp, &size);
  } else if (1==myRank) {
    ranks[0] = 0;
    MPI_Group_incl(grp_sm, 1, ranks, &grpOther);
    sleep(5); // to guarantee that Bsend is done before Recv is called
    MPI_Recv(&myHugeMessage[0], count, MPI_DOUBLE, 0, tag,
             comm_sm, MPI_STATUS_IGNORE);
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 0, 0, win);
    MPI_Put(&tag, 1, MPI_INT, 0, 0, 1, MPI_INT, win);
    MPI_Win_unlock(0, win);
  }

  MPI_Win_free(&win);
  MPI_Finalize();
}
```

1. Using ≥2 processes on a shared memory system, the window handle should provide memory_model = MPI_WIN_UNIFIED, see MPI-4.0 page 560 lines 21-27

4. Therefore, only with **always** progressing MPI implementation and MPI_WIN_UNIFIED, the data will be available after some delay ("eventually")

2. With **always** progressing MPI implementation, it will return

2. With **weakly** progressing MPI implementation, it will **block** because there is no MPI call in Rank 0.
→ lock/put/unlock will never be called
→ no further MPI call in Rank 0 → deadlock

3. Only with **always** progressing MPI implementation, this MPI_Put is executed

Progress rule for MPI_Win_test – Problem 3

(modified version with a direct assignment instead of MPI_Put)

Counter-example to proposed "1st draft of a main progress rule"

- **Deadlock-free with an always progressing MPI library**
- **Deadlock with MPI libraries progressing MPI_Bsend only in MPI calls**

Or is it just wrong?

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv) {
  volatile int myAck=222;
  int myWorldRank, smSize, myRank=-1, tag=101, ranks[1], flag=0, size=1000*1000*1000, count=size/sizeof(double);
  void *pTemp;     MPI_Comm comm_sm;     MPI_Win win;     MPI_Group grp_sm, grpOther;
  double *myHugeMessage = malloc(size);    char *myHugeBuffer = malloc(size+MPI_BSEND_OVERHEAD);

  MPI_Init(&argc, &argv);
  MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL, &comm_sm);
  MPI_Comm_rank(MPI_COMM_WORLD, &myWorldRank);   MPI_Comm_rank(comm_sm, &myRank);
  MPI_Comm_size(comm_sm, &smSize);                MPI_Comm_group(comm_sm, &grp_sm);
  if (smSize < 2) {printf("[rank:%d] smSize=%d < 2 → Abort\n", myWorldRank, smSize); MPI_Abort(comm_sm, MPI_ERR_SIZE);}
  MPI_Win_create(&myAck, sizeof(int), sizeof(int), MPI_INFO_NULL, comm_sm, &win); // -> memory_model is MPI_WIN_UNIFIED

  if (0==myRank) {
    ranks[0] = 1;
    MPI_Group_incl(grp_sm, 1, ranks, &grpOther);
    MPI_Buffer_attach(&(myHugeBuffer[0]), size+MPI_BSEND_OVERHEAD);
    MPI_Bsend(&myHugeMessage[0], count, MPI_DOUBLE, 1, tag,
              comm_sm);
    sleep(10); // to guarantee that the while starts after rank 1 is blocked in Recv
    while(myAck!=tag);
    MPI_Buffer_detach(&pTemp, &size);
  } else if (1==myRank) {
    ranks[0] = 0;
    MPI_Group_incl(grp_sm, 1, ranks, &grpOther);
    sleep(5); // to guarantee that Bsend is done before Recv is called
    MPI_Recv(&myHugeMessage[0], count, MPI_DOUBLE, 0, tag,
             comm_sm, MPI_STATUS_IGNORE);
    int  *Ack_inRank0;    MPI_Aint  Ack_bytesize;   int  Ack_dispunit;
    MPI_Win_shared_query(win, 0, &Ack_bytesize, &Ack_dispunit, &Ack_inRank0);
    *Ack_inRank0 = tag;   // instead of MPI_Put(&tag, 1, MPI_INT, 0, 0, 1, MPI_INT, win);
  }

  MPI_Win_free(&win);
  MPI_Finalize();
}
```

Problematic: If blue and purple is in two different libraries!

Is this "signaling" within the **unified** memory model just not allowed and the MPI standard forgot to document this restriction?

The communication in a shared memory can be done with a normal assignment. This assignment is eventually visible in process rank 0 without additional RMA calls, see MPI-4.0 12.4, page 592, lines 42-48.

Or are these examples just wrong?

MPI-4.0 Section 2.7 Processes, page 26, lines 28-41:
This document species the behavior of a parallel program assuming that **only MPI calls** are used. The interaction of an MPI program **with other possible means of communication**, I/O, and process management **is not specified**. Unless otherwise stated in the specification of the standard, MPI places no requirements on the result of its interaction with external mechanisms that provide similar or equivalent functionality. This includes, but is not limited to, interactions with external mechanisms for process control, shared and remote memory access, file system access and control, interprocess communication, **process signaling**, and terminal I/O. High quality implementations should strive to make the results of such interactions intuitive to users, and attempt to document restrictions where deemed necessary. *Advice to implementors.* Implementations that support such additional mechanisms for functionality supported within MPI are expected to **document** how these interact with MPI. *(End of advice to implementors.)*

Main progress rule – 2nd draft

June 2, 2021: This proposal was not discussed, however during older discussions, there was also some significant rejection on defining such details as binding text.
With the 1st draft, these rules could be an *adv. to implem.*

Definition of "progress chunk": That a stage or parts of it can be finished, this often requires that another process or several other processes act.
Those activities can be within an operation stage but can also be enabled by an operation stage and being no longer part of this operation stage. We name such an activity "progress chunk".
An example is a call to MPI_BSEND while the corresponding MPI_RECV is not yet called. The MPI_BSEND can internally buffer the message and can generate such a progress chunk which will send the data for example when the corresponding process calls MPI_RECV.
Progress chunks may also generate new progress chunks in the own process or in any other connected process.

Rule 1+2: If a progress chunk is able to make progress within an MPI process, then MPI implementations must guarantee that the progress chunk makes progress, while blocked in

- (Rule 1:) an MPI call or
- (Rule 2:) repeatedly calling one of the test procedures MPI_TEST, MPI_REQUEST_GET_STATUS, MPI_IPROBE, MPI_IMPROBE, MPI_PARRIVED, and MPI_WIN_TEST.

Rule 1+2-b: This progress must be enabled independent of which session, communicator, window or file handle the progress chunk and the blocked MPI call or test procedure is related. Prevented from returning

Rule 3a+b: An MPI procedure call may be blocked because it is a non-local procedure waiting for

- (Rule 3a:) a semantically related procedure call or operation stage in another or the same process,
- (Rule 3b:) but it can also be blocked because waiting that a progress chunk is progressed in another or the same process, which is possible for local and non-local procedures.

For example MPI_RECV must behave the same as a local procedure if the corresponding sending process has already called a related send procedure. If the sending process has buffered the message and already finished the sending MPI procedure then the MPI_RECV may be blocked until the sending process makes progress on the related progress chunk.

Rule 4: During finalization of MPI, each process must guarantee that all progress chunks that are related to the finalization call are finished, independent on whether they are (Rule 4a:) already created or (Rule 4b:) whether they will be created by the same process or another process during the finalization.
This rule for example allows that an ongoing tree based protocol can still be finished in other processes while a buffering MPI_BCAST already returned and the finalization is already started in the root process of this call.
Rationale. Especially Rule 4b is the reason, why a finalization has to be allowed to synchronize, see MPI_FINALZE and MPI_SESSION_FINALIZE for details. *End of rationale.*

Rationale. All these rules specify minimal progress requirements. All these rules do not specify when which progress is really done. The rule 2b for tests also does not require that the progress is done during the test calls. *End of rationale.*

June 2, 2021: This proposal was not discussed, however it tried in Issue 493 (N) – PR 575 to summarize current knowledge of pieces on progress within the MPI standard

References on progress rules

\section{Progress}
\mpitermtitleindex{progress}

% 1. Summary on progress during blocking MPI procedures:
\MPI/ requires that every \MPI/ process makes progress on
all \mpitermni{enabled}\mpitermindex{operation!enabled} \MPI/ operations it participates in,
while blocked on an \MPI/ call, % sentence copied from one-sided-2.tex
as described in the paragraphs \mpicode{Progress} in Sections~\ref{sec:pt2pt-semantics} and~\ref{subsec:pt2pt-semantics},
and Sections~\ref{subsec:dynamic:threads:general} and~\ref{sec:1sided-progress}.

% 2. Local inquiry procedures MPI_TEST, MPI_REQUEST_GET_STATUS, MPI_IPROBE, MPI_IMPROBE, ~~and~~
MPI_PARRIVED and MPI_WIN_TEST must eventually return "true":
Furthermore, some inquiry procedures, if called repeatedly with the same input parameter values,
will eventually return \mpiarg{flag}\mpicode{ = true} % words copied from pt-to-pt.tex
once the matching operation % using "operation" instead of the two examples "send" and "receive"
has been \mpitermni{started}, % words copied from pt-to-pt.tex,
for examples, see \mpifunc{MPI_TEST} in \sectionref{subsec:pt2pt-semantics},
and the progress rule for
\mpifunc{MPI_REQUEST_GET_STATUS} in \sectionref{subsec:pt2pt-teststatus},
\mpifunc{MPI_IPROBE} in \sectionref{subsec:pt2pt-probe},
\mpifunc{MPI_IMPROBE} in \sectionref{sec:matching-probe},
\mpifunc{MPI_PARRIVED} in \sectionref{subsec:part-commend},
and \mpifunc{MPI_WIN_TEST} in \sectionref{sec:1sided-progress}.

→ Continued on next slide

Please explore the references, see page 28 (68 of 1139)
https://github.com/mpi-forum/mpi-issues/files/6591009/mpi40-report-rc-4-june-N-PR575-v03.pdf#page=68

References on progress rules – continued

% 3. MPI_FINALIZE and MPI_SESSION_FINALIZE must complete progress before returning:
In addition, the synchronization characteristics of
the finalizing procedures of \MPI/
% Copied from that advice to implementors
% and the corresponding text in the description of MPI_SESSION_FINALIZE:
allow for the completion of all \mpitermni{enabled}\mpitermindex{operation!enabled} \MPI/ operations it participates in and that may not yet be completed from the viewpoint of the underlying \MPI/ system.
See the advice to implementors related to Example~\ref{example:cancel-finalize} in \sectionref{sec:inquiry-finalize} and the description of \mpifunc{MPI_SESSION_FINALIZE} in \sectionref{subsec:sessions:init-finalize}.

% 4. Local procedures may (and some must) be implemented as weak-local, i.e.,
% for returning, they may require progress in the other process
Local \MPI/ procedures
may require the involvement % process 1: Wording from the rationale in Section~\ref{sec:1sided-progress}
of another \MPI/ process:
its return may depend on (remote) progress, i.e.,
an unspecific, not semantically-related \MPI/ procedure call % compare with the definition of non-local
in that other process, see \mpifunc{MPI_CANCEL} in Example~\ref{example:cancel-finalize},
and Figure~\ref{fig:1sided-deadlck2} and its
explanation together with the rationale in Section~\ref{sec:1sided-progress}.

% 5. Further hints, see general index:
Further progress rules exist, for example on \MPI/ collective communication, \MPI/ and threads, \MPI/ one-sided communication, and \MPI/ file I/O.
At the keyword \mpitermni{progress}
in the \hyperref[index:general]{general index}, %%ALLOWLATEX%
one can find some further references about the progress rules of \MPI/.

Please explore the references, see page 28 (68 of 1139)
https://github.com/mpi-forum/mpi-issues/files/6591009/mpi40-report-rc-4-june-N-PR575-v03.pdf#page=68

ANNEX

Progress rule for MPI_Win_test – Problem 3
(full & original version from Dan Holmes)

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv) {
  volatile int myAck=222;
  int myRank=-1, tag=101, ranks[1], flag=0, size=1000*1000*1000, count=size/sizeof(double);
  void *pTemp;
  MPI_Win win;
  MPI_Group grpWorld, grpOther;
  double *myHugeMessage = malloc(size);
  char *myHugeBuffer = malloc(size+MPI_BSEND_OVERHEAD);

                                printf("Got here 0\n");
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
                                printf("[rank:%d] Got here 1\n", myRank);
  MPI_Comm_group(MPI_COMM_WORLD, &grpWorld);
                                printf("[rank:%d] Got here 2\n", myRank);
  MPI_Win_create(&myAck, sizeof(int), sizeof(int), MPI_INFO_NULL, MPI_COMM_WORLD, &win);
                                printf("[rank:%d] Got here 3\n", myRank);
  if (0==myRank) {
    ranks[0] = 1;
    MPI_Group_incl(grpWorld, 1, ranks, &grpOther);
                                printf("[rank:%d] Got here 3.1\n", myRank);
                                printf("[rank:%d] Got here 3.2 pointer is %p\n", myRank, myHugeBuffer);
    MPI_Buffer_attach(&(myHugeBuffer[0]), size+MPI_BSEND_OVERHEAD);
                                printf("[rank:%d] Got here 3.3 pointer is %p\n", myRank, myHugeBuffer);
    MPI_Bsend(&myHugeMessage[0], count, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD);
                                printf("[rank:%d] Got here 3.4 myAck=%d\n", myRank, myAck);
  sleep(10);
//  MPI_Request request;
//  MPI_Issend(&myRank, 1, MPI_INT, 1, 1, MPI_COMM_WORLD, &request);
//  MPI_Status status;
//  for (int i;i<10000000;++i) {
//    MPI_Test(&request, &flag, &status);
//  }
//                                printf("[rank:%d] Got here 3.4 flag=%d\n", myRank, flag);
//  if (!flag) {
//    MPI_Cancel(&request);
//    MPI_Wait(&request, &status);
//  }
                                printf("[rank:%d] Got here 3.5 myAck=%d (looking for tag=%d)\n", myRank, myAck, tag);
    while(myAck!=tag);
                                printf("[rank:%d] Got here 3.6\n", myRank);
    MPI_Buffer_detach(&pTemp, &size);
  }
  else if (1==myRank) {
    ranks[0] = 0;
    MPI_Group_incl(grpWorld, 1, ranks, &grpOther);
                                printf("[rank:%d] Got here 3.1\n", myRank);
    MPI_Recv(&myHugeMessage[0], count, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
                                printf("[rank:%d] Got here 3.2\n", myRank);
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 0, 0, win);
                                printf("[rank:%d] Got here 3.3\n", myRank);
    MPI_Put(&tag, 1, MPI_INT, 0, 0, 1, MPI_INT, win);
                                printf("[rank:%d] Got here 3.4\n", myRank);
    MPI_Win_unlock(0, win);
  }
                                printf("[rank:%d] Got here 4\n", myRank);
  MPI_Win_free(&win);
                                printf("[rank:%d] Got here 5\n", myRank);
  MPI_Finalize();
                                printf("[rank:%d] Got here 6\n", myRank);
}
```

Progress rule for MPI_Win_test – Problem 3 (short version from Dan Holmes)

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv) {
  volatile int myAck=222;
  int myRank=-1, tag=101, ranks[1], flag=0, size=1000*1000*1000, count=size/sizeof(double);
  void *pTemp;    MPI_Win win;    MPI_Group grpWorld, grpOther;
  double *myHugeMessage = malloc(size);
  char *myHugeBuffer = malloc(size+MPI_BSEND_OVERHEAD);

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
  MPI_Comm_group(MPI_COMM_WORLD, &grpWorld);
  MPI_Win_create(&myAck, sizeof(int), sizeof(int), MPI_INFO_NULL, MPI_COMM_WORLD, &win);

  if (0==myRank) {
    ranks[0] = 1;
    MPI_Group_incl(grpWorld, 1, ranks, &grpOther);
    MPI_Buffer_attach(&(myHugeBuffer[0]), size+MPI_BSEND_OVERHEAD);
    MPI_Bsend(&myHugeMessage[0], count, MPI_DOUBLE, 1, tag,
                      MPI_COMM_WORLD);
    sleep(10); // to guarantee that the while starts after rank 1 is blocked in Recv
    while(myAck!=tag);
    MPI_Buffer_detach(&pTemp, &size);
  } else if (1==myRank) {
    ranks[0] = 0;
    MPI_Group_incl(grpWorld, 1, ranks, &grpOther);
    sleep(5); // to guarantee that Bsend is done before Recv is called
    MPI_Recv(&myHugeMessage[0], count, MPI_DOUBLE, 0, tag,
                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 0, 0, win);
    MPI_Put(&tag, 1, MPI_INT, 0, 0, 1, MPI_INT, win);
    MPI_Win_unlock(0, win);
  }

  MPI_Win_free(&win);
  MPI_Finalize();
}
```