

Class Notes Aug 30 2006.

If we want to send messages from one process to another, here is one way to do this. Create a shared array with "Size" number of elements.

```
shared int Buffer[Size];
```

We will also have two shared variables to manage the array, one that tells us where the next message should go, and one that tells us where the next message should be read from. These variables are both initialized to zero:

```
shared int SendPos = 0;
shared int RecPos = 0;
```

Then whenever we send a message we will do this:

```
Buffer[SendPos] = SMsg;
SendPos++;
```

Whenever we want to receive a message we will do this:

```
RMsg = Buffer[RecPos];
RecPos++;
```

There are several problems with these two algorithms. First, what happens when the buffer is full and I want to send another message? Second, what happens if the buffer is empty when I want to receive a message? Third, what happens when SendPos and RecPos become greater than or equal to Size?

We can fix problem 3 like this:

```
Buffer[SendPos] = SMsg;
SendPos++;
if (SendPos >= Size)
{
    SendPos = 0;
}
```

```
-----:
```

```

RMsg = Buffer[RecPos];
RecPos++;
if (RecPos >= Size)
{
    RecPos = 0;
}

```

Although it is less efficient to do so, we can write the increment in one line like this:

```

SendPos = (SendPos+1)%Size;
RecPos = (RecPos+1)%Size;

```

Detecting an empty buffer is easy too. Notice that in the beginning the buffer is empty and $\text{SendPos} = \text{RecPos} = 0$. If I send one message and receive one message, the buffer will be empty, and $\text{SendPos} = \text{RecPos} = 1$. So if SendPos and RecPos are equal, it is an indication that the buffer is empty. We rewrite the receiver code like this.

```

while (SendPos == RecPos);
RMsg = Buffer[RecPos];
RecPos = (RecPos+1)%Size;

```

Note the semicolon at the end of the while statement. This causes the while to execute over and over again until the condition becomes false. This is known as a “busy wait.” Busy waits are bad, but they work.

What about the buffer being full? If we add “Size” messages to the buffer without anything being received, we will increment SendPos until it is equal to Size . But the test at the end will then set SendPos back to zero. So $\text{SendPos} = \text{RecPos} = 0$. Unfortunately, this condition is identical to the buffer empty condition. We can get around this, by treating the buffer as full when there is one empty slot. To test this condition we can use the following test $(\text{SendPos}+1)\% \text{Size} == \text{RecPos}$. This says, “Will the buffer become full if I add one more message?” If the answer is “yes” then the buffer has one empty slot. I modify the sender code like this.

```

while ((SendPos+1)%Size == RecPos);
Buffer[SendPos] = SMsg;
SendPos = (SendPos+1)%Size;

```

Again, note the semicolon at the end of the while statement.

This code works pretty well, but this annoyance of having one empty slot in the buffer all the time should be fixable. Let’s try another approach. Let’s use a shared variable that tells us how many slots are available in the buffer.

```

shared int Slots = Size;

```

Now we modify the sender and receiver thus:

```
while (Slots < 1);
Buffer[SendPos] = SMsg;
Slots--;
SendPos = (SendPos+1)%Size;
-----
while (Slots == Size);
RMsg = Buffer[RecPos];
Slots++;
RecPos = (RecPos+1)%Size;
```

This gets rid of the empty slot, but unfortunately, it doesn't work. Why not?

If the Slots++ and Slots-- happen to interleave in the right way, the total could be incorrect. One or the other could appear not to have been executed.

Why didn't this problem affect the first solution? Here's a quick test.

For each process create two sets of variables, the Read set and the Write set. The Read set is all shared variables that are used by the process, but not changed. The Write set is all shared variables that are changed by the process.

So for the receiver the sets are:

Read: Buffer, SendPos

Write: RecPos

For the sender the sets are:

Read: RecPos

Write: Buffer, SendPos

If the write sets of two processes do not intersect, then the problems we encountered with updating shared variables cannot occur. For the modified algorithms, we would have to add Slots to the read set and the write set of both processes. This would indicate a potential for interference.

If a read set intersects with another process's write set, there is a potential for inconsistency if the write set contains structures or objects that have internal items that must be consistent with one another. (Linked list, plus counter, for example.)

In the bounded buffer case, the order of the operations prevents inconsistency from occurring. But, what happens if we change the order of these two statements?

```
Buffer[SendPos] = SMsg;
SendPos = (SendPos+1)%Size;
```