Data Structures, Buffers, and Interprocess Communication

We've looked at several examples of interprocess communication involving the transfer of data from one process to another process. We know of three mechanisms that can be used for this transfer:

- Files
- Shared Memory
- Message Passing

The act of transferring data involves one process writing or sending a *buffer*, and another reading or receiving a *buffer*.

Most of you seem to be getting the basic idea of sending and receiving data for IPC… it's a lot like reading and writing to a file or stdin and stdout. What seems to be a little confusing though is HOW that data gets copied to a buffer for transmission, and HOW data gets copied out of a buffer after transmission.

First… let's look at a piece of data.

```
typedef struct {
  char   ticker[TICKER_SIZE];
  double price;
} item;

.
.
.
item next;
.
.
.
```

The data we want to focus on is "next". "next" is an object of type "item". "next" occupies memory in the process. What we'd like to do is send "next" from processA to processB via some kind of IPC.

IPC Using File Streams

If we were going to use good old C++ filestreams as the IPC mechanism, our code would look something like this to write the file:

```
// processA is the sender…
ofstream out;
out.open("myipcfile");

item next;
strcpy(next.ticker,"ABC");
next.price = 55;

out << next.ticker << " " << next.price << endl;
out.close();
```

Notice that we didn't do this:

```
out << next << endl;
```

Why? Because the "<<" operator doesn't know what to do with an object of type "item". The object is too complex. However, the "<<" operator does know what to do with simple data types like "char *" and "double". So to get "next" into our file, we break it down into simpler components before we write it to the file. (You now from earlier courses that you can write some code to make "<<" work with objects of type "item" but that's a different course.)

Now we need to have our other process get data out of the file. So:

```
// processB is the receiver…
ifstream in;
in.open("myipcfile");

item next;

in >> next.ticker >> next.price >> endl;

in.close();
```

Nothing too fancy here, but note again that we had to break next down into its basic components to read it in from the file. Also note the ORDER that we read the data out. We got the ticker information followed by the price information.

So what's the lesson here? Unless we do something fancy with the "<<" and ">>" operators, the only way we can read/write objects of type "item" is to break them down into more basic components that the operators do understand. And the order we read things out of the file is the same order we wrote things into the file.

IPC Using Message Passing

The basic message passing mechanisms that we are studying in this course use a "buffer" as the communication vehicle. We have to come up with a mechanism for copying data from objects into a buffer for sending a transmission, and copying data from a buffer to the objects for receiving a transmission. It's the same idea as the file I/O we saw in the previous section, only this time we're reading and writing from a buffer instead of a filestream.

Let's look at a specific function to send data across a pipe:

```
write(int fd, void * buffer, size_t size);
```

`write` takes 3 arguments:

- fd – the file descriptor, which in our case is going to be the file descriptor for the writing end of the pipe we're using to communicate between processA and processB
- buffer – a pointer to a block of memory that contains the data we want to send
- size – the number of bytes of data that we want to send

In order to get our "next" data item sent from processA to processB using pipes we have to get "next" into a buffer, and pass that buffer into the function "write".

Ok… so what IS this buffer? The buffer is simply ANY *continuous* chunk of memory owned by processA. You can create this *continuous* chunk of memory lots of different ways:

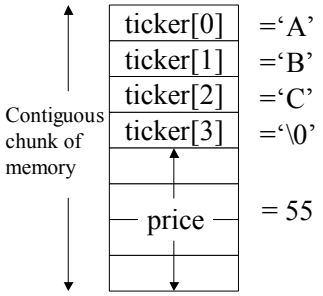
```
char buffer1;
int buffer2;
char buffer3[1000];
char *buffer4 = new char[1000];
```

What's important about any buffer you pass to write() is that you know the starting address of the buffer, and the size of the buffer. Here we have four different buffers. buffer1 takes up a single byte of memory and starts at address &buffer1. buffer2 takes up 4 bytes of memory (on your Linux box) and starts at address &buffer2. buffer3 takes up 1000 bytes of memory and starts at address buffer3… etc. Any one of these buffers would be a valid parameter to write() for example:

```
write(thePipe[1], &buffer1, 1);
```

would write the character located in buffer1 across the pipe from processA to processB.

OK… so what about our object "next"? Let's take a closer look at "next" in memory. The memory occupied by "next" consists of two components: an array of TICKER_SIZE characters and a double. These components are adjacent to each other in memory, which means that all of the data that comprise "next" is in one continuous chunk of memory. Do you see where this is leading?



Memory diagram of original "next" object.

Since "next" is one continuous chunk of memory consisting of an array of characters and a double, "next" can actually BE the buffer that you pass to write()! All we need to tell write() is the address of the buffer (&next) and the size of the buffer. Computing the size of the buffer is pretty easy. It's TICKER_SIZE characters or bytes, plus the size of the double (4 bytes on your Linux box) = TICKER_SIZE+4.

Figuring out the size in bytes of a data structure is even easier, however, with the sizeof() utility. If you want to find out the size in bytes of any type, struct, or class use sizeof():

```
cout << "The size of an int is: " << sizeof(int) << endl;
cout << "The size of an item is: " << sizeof(item) << endl;
```

Alrighty then… "next" can be our buffer, and we know its address and size. So let's write the code to transmit it from processA to processB:

```
// processA is the sender…
.
```

```
.
.

item next;
strcpy(next.ticker,"ABC");
next.price = 55;

write(thePipe[1],&next,sizeof(item));
```

On the receiving end, we also need a buffer, and we can use a "next" object that exists in processB:

```
// processB is the receiver…
.
.
.

item next;

read(thePipe[0],&next,sizeof(item));
```

Buffers and Complex Objects

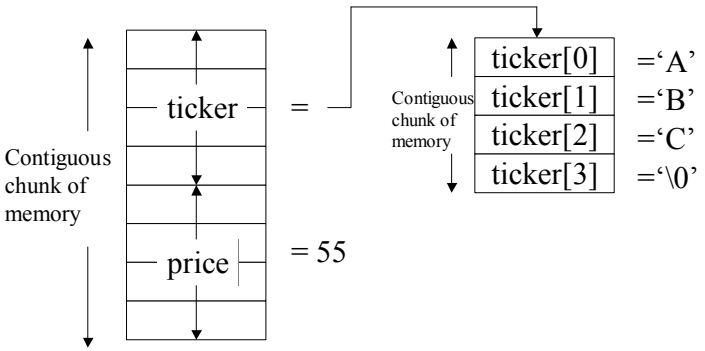
An object of type "item" occupies a contiguous chunk of memory so the object can be used as the buffer for message passing. What happens, though, when an object does not occupy a contiguous chunk of memory?

First lets take a look at an object with this property:

```
typedef struct {
  char   *ticker;
  double price;
} item;

.
.
.
item next;
next.ticker = new char[TICKER_SIZE];
strcpy(next.ticker,"ABC");
next.price = 55;
.
.
.
```

We made a minor change to the "item" struct by changing the ticker member to be a pointer to a character (or array of characters), instead of declaring ticker to be an array of TICKER_SIZE characters. This minor change means that any object declared of type "item" will occupy TWO chunks of memory. One chunk for ticker and price and one chunk for the array of characters that ticker points to:



Memory diagram of complex "next" object.

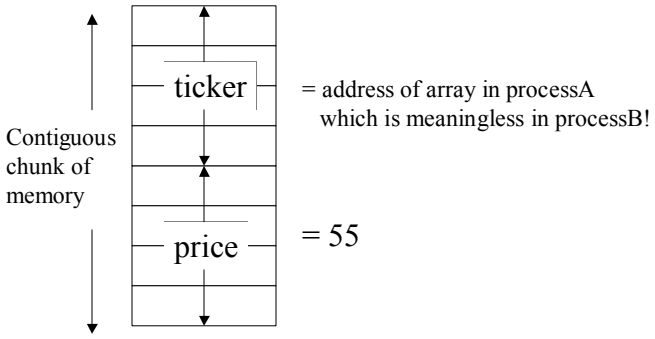
Now we have a problem. We can't use "next" as the buffer to transmit data because the data is contained in two separate chunks of memory. To illustrate this problem, let's look at what would happen if we tried to use the object "next" to transmit the object and all of it's data from processA to processB:

```
write(thePipe[1],&next,sizeof(item));
```

This call to write() will transmit the chunk of memory in the "next" object consisting of the pointer to the ticker array and the price. When it arrives at processB, it's read in:

```
read(thePipe[0],&next,sizeof(item);
cout << "The ticker is: " << next.ticker << " The price is: " << next.price << endl;
```

The cout statement will probably crash with a segmentation fault. Why? Because when cout tries to dereference the next.ticker pointer which points to the array of characters, the pointer will be to a memory location that is owned by processA… not processB! This pointer to a memory location in processA is meaningless to processB. Trying to access processA memory from processB violates the memory protection mechanism put in place by the hardware and operating system.



Memory diagram of complex "next" object after read() in processB.

So how do we solve this problem? The solution is to:

- create a buffer that we can use to transmit data from processA to processB.
- copy data from the "next" memory chunks in processA into the buffer

- transmit the buffer to processB
- copy data from buffer to "next" memory chunks in processB

Step 1: Creating the buffer

We need to figure out how large the buffer is going to be. We need to transmit price which is a double (4 bytes in Linux) and we need to transmit the array of characters that makes up the ticker (TICKER_SIZE bytes). We DO NOT want to transmit the pointer to the array of characters that is part of the "next" object… as we've already seen it's meaningless when it arrives at processB. Let's use sizeof() to make our lives easier:

```
char *buffer=new char[sizeof(double)+sizeof(char)*TICKER_SIZE];
```

Step 2: Copying data to buffer

We need to copy the data from the ticker array and price into our newly created buffer. There are lots of ways for you to do this. I'm going to show you how to do it using the system call "memcpy":

```
void *memcpy(void *s1, const void *s2, size_t n);
```

memcpy takes 3 arguments:

- s1 – the address of the contiguous block of memory you want to copy data to
- s2 – the address of the contiguous block of memory you want to copy data from
- n – the number of bytes that you want to copy

Alright. Now we're ready to copy the data from our "next" object. First we have to figure out what order we're going use. Should we copy the ticker character array first or the price first? This doesn't matter… but what does matter is that the data be copied out from the buffer in processB in the same order that it was copied in for processA. Let's copy the ticker array first:

```
memcpy(buffer,next.ticker,sizeof(char)*TICKER_SIZE);
```

Now we need to copy the price. In order to copy the price, we need to make sure that the buffer address we pass to memcpy appears after the ticker array we've already copied. Also notice that we're passing the ADDRESS of price, not its value. This will copy the bytes directly out of the memory location occupied by price into our buffer:

```
memcpy(buffer+sizeof(char)*TICKER_SIZE,&ticker.price,sizeof(double));
```

Step 3: Transmitting buffer

Now it's time for processA to transmit the buffer to processB:

```
write(thePipe[1],buffer, sizeof(char)*TICKER_SIZE+sizeof(double));
```

Step 4: Receiving buffer

Now processB receives the buffer from processA:

```
read(thePipe[0], buffer, sizeof(char)*TICKER_SIZE+sizeof(double));
```

Step 5: Copy contents of buffer to “next” object in processB

We’re going to use memcpy again:

```
Item next;
next.ticker = new char[TICKER_SIZE];
memcpy(next.ticker,buffer,sizeof(char)*TICKER_SIZE);
memcpy(&next.price,buffer+sizeof(char)*TICKER_SIZE,sizeof(double));
cout << “The ticker is: “ << next.ticker << “ The price is: “ << next.price << endl;
```

Test your understanding…

Now let’s figure out if you got it. Here’s another complex data structure that we want to transmit from processA to processB:

```
typedef struct {
  char   **tickerList;
  int    sizeList;
} item;

.
.
.
item next;
cout << “Type in the number of tickers you want to store” << endl;
cin >> next.sizeList;
next.tickerList = new char* [sizeList];
for (int i=0; i<next.sizeList; i++) {
   next.tickerList[i] = new char[TICKER_SIZE];
   strcpy(next.tickerList[i],”ABC”);
}
.
.
.
```

Write the code that will copy the object “next” to a buffer and transmit the buffer from processA to processB. Write the code that will receive the buffer in processB and copy the data to a new “next” object.

Here’s a shell for a program that will do this:

```
#include <iostream>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <time.h>
#include <mm.h>
#include <main.h>

const char TICKER_SIZE = 4;
typedef struct {
  char   **tickerList;
  int    sizeList;
} item;

int main(void)
{
  int thePipe[2];

  // Create a single pipe.  The pipe has two ends:
  // - thePipe[0] receiving end - messages are taken out of this end
  // - thePipe[1] sending   end - messages are placed in this end
  // NOTE: This MUST BE DONE BEFORE YOU FORK!
  pipe(thePipe);

  // create a child
  int pid = fork();

  // Error forking
  if (pid < 0) {
    cerr << "main: Fork failed!" << endl;
    exit(-1);
  }

  ////////////////////////////////////////////////////////////////////////
  // Child process is the CONSUMER
  ////////////////////////////////////////////////////////////////////////
  if (pid ==0) {

    // Pipe manipulation for the RECEIVER
    // close thePipe[1] we're not using the sending end
    close(thePipe[1]);

    // Read item from pipe
    item nextConsumed;

      // Add your code to create a buffer and copy to nexConsumed

    read(thePipe[0],xxx,xxx);


  }

  ////////////////////////////////////////////////////////////////////////
  // Parent process is the PRODUCER
  ////////////////////////////////////////////////////////////////////////
  else {
    int count=0;

    // Pipe manipulation for the PRODUCER
    // close thePipe[0] we're not using the receiving end
    close(thePipe[0]);

    // Create item
    item nextProduced;
         cout << "Type in the number of tickers you want to store" << endl;
         cin >> nextProduced.sizeList;
         next.tickerList = new char* [sizeList];
         for (int i=0; i<nextProduced.sizeList; i++) {
                 nextProduced.tickerList[i] = new char[TICKER_SIZE];
                 strcpy(nextProduced.tickerList[i],"ABC");
         }

    // Send item

    // Add your code to create a buffer and copy nextProduced

         write(thePipe[1],xxx,xxx);
  }

  return 1;
}
```