Pay the Piper

Authored by: **John Fehr**
Updated by: **Thomas Fletcher**

Implementing an efficient buffered piping mechanism between two threads can be quite a challenge. Obviously we need some kind of locking mechanism on the pipe's buffer so that we don't overwrite or overread data. There are a number of methods we could use, and mutexes seem to be a good choice, since they're quite lightweight. But is it the best solution? Let's find out.

A slow tune.

Lets start our implementation with a structure that we'll use to hold the buffered data and pointers:

```
#include <pthread.h>
// buffer size
#define BSIZE 1024
typedef struct
{
 char buf[BSIZE];         // our buffered data
 int rpos;                // current read position
 int wpos;                // current write position
 int full;                // if rpos==wpos, is the buffer full or
empty?
  pthread_mutex_t mutex;   // lock for our buffer
} piper_t;
```

You gotta keep em separated!

That seems pretty straight forward. Our pipe read and write functions will be almost identical, except we'll be setting the full flag in the writer (when the buffer gets totally full), and we'll be resetting it in the reader, since we'll always be reading at least one byte.

Another difference of course, will be that we're moving the **wpos** pointer in the writer, and the **rpos** pointer in the reader.

Finally, in both cases we check the full flag if the read and write positions are identical. In the writer, if the full flag is set and the read and write positions are identical, this means that there is no room in the buffer to write more bytes, so we release our lock, and try again after a very short delay. In the reader, if the full flag is not set and the read and write positions are identical, this means that the buffer is totally empty, so we release our lock, and try again after a very short delay.

Sound straight forward? Here's the code:

```
// Write any number of bytes to the pipe, even if there are more
// than the buffersize of the pipe.
int pipe_write(piper_t *p,char *data,int len)
{
 // keep looping until we've written all the data we want
 while (len>0)
```

```
 {
  int towrite;
  // lock our pipe
  pthread_mutex_lock(&p->mutex);
  // if we have no room to write, we unlock the mutex,
  // delay a bit, and try again.
  if (p->wpos==p->rpos && p->full)
  {
   pthread_mutex_unlock(&p->mutex);
   usleep(100);
   continue;
  }
  // figure out how many bytes we can write this loop
  if (p->rpos<=p->wpos)
   towrite=BSIZE-p->wpos;
  else
   towrite=(p->rpos-p->wpos);
  if (towrite>len) towrite=len;
  // copy our buffer and move the write pointer
  memcpy(&p->buf[p->wpos],data,towrite);
  p->wpos=(p->wpos+towrite)%BSIZE;
  // move our buffer pointer and decrement our len left
  data+=towrite;
  len-=towrite;
  // check if we're now 'full'
  if (p->wpos==p->rpos) p->full=1;
  // unlock our pipe so the reader can access it if neccessary.
  pthread_mutex_unlock(&p->mutex);
 }
 return 0;
}
// Read any number of bytes from the pipe, even if there are more
// than the buffersize of the pipe.
int pipe_read(piper_t *p,char *data,int len)
{
 // keep looping until we've gotten all the data we requested
 while (len>0)
 {
  int toread;
  // lock our pipe
  pthread_mutex_lock(&p->mutex);
  // if we have no room to read, we unlock the mutex,
  // delay a bit, and try again.
  if (p->rpos==p->wpos && !p->full)
  {
   pthread_mutex_unlock(&p->mutex);
   usleep(100);
   continue;
  }
  // figure out how much we can read this loop
  if (p->rpos<p->wpos)
   toread=(p->wpos-p->rpos);
  else
   toread=BSIZE-p->rpos;
  if (toread>len) toread=len;
```

```
   // copy our buffer and move the read pointer
   memcpy(data,&p->buf[p->rpos],toread);
   p->rpos=(p->rpos+toread)%BSIZE;
   // move our buffer pointer and decrement our len left
   data+=toread;
   len-=toread;
   // indicate that the buffer is no longer completely full
   p->full=0;
   // unlock our pipe so the writer can access it if neccessary.
   pthread_mutex_unlock(&p->mutex);
  }
  return 0;
 }
```

I need two lines, and a picture ID.

Now we'll need two threads, one reading from the pipe, and the other writing to the pipe. To make sure that it works over a wide variety of buffer sizes, we'll write from **1** to **BSIZE*2-1** bytes in the writer thread, and read from **BSIZE*2-1** to **1** byte in the reader thread. This should be exhaustive enough to give our pipe a good workout. Here are the two functions for our two threads:

```
 void *writer(void *ptr)
 {
  char data[BSIZE*2];
  int i;
  piper_t *p=(piper_t*)ptr;
  char n=0;
  for (i=1;i<BSIZE*2;i++)
  {
   int j;
   for (j=0;j<i;j++)
    data[j]=n++;
   pipe_write(p,data,i);
  }
  printf("Completed writer task.n");
 }
 void *reader(void *ptr)
 {
  char data[BSIZE*2];
  int i;
  piper_t *p=(piper_t*)ptr;
  char n=0;
  for (i=BSIZE*2-1;i>0;i--)
  {
   int j;
   pipe_read(p,data,i);
   // verify our data
   for (j=0;j<i;j++)
   {
    if (data[j]!=n) printf("data is %d, should be %dn",data[j],n);
    n++;
   }
  }
  printf("Completed reader task.n");
 }
```

Let 'er rip!

Looks like we're ready to go except for the **main**(). All we do there is launch a reader thread, run the writer in the main thread, and wait for the reader thread to finish:

```
main()
 {
  piper_t p;
  pthread_t rthread;
  void *retval;
  // make sure the buffer is clear before we start
  memset(&p,0,sizeof(p));
  // initialize our pipe's mutex
  pthread_mutex_init(&p.mutex,NULL);
  // start our reader thread
  pthread_create(&rthread,NULL,&reader,&p);
  // run our writer
  writer(&p);
  // wait for the reader thread to finish
  pthread_join(rthread,&retval);
  // destroy our pipe's mutex
  pthread_mutex_destroy(&p.mutex);
 }
```

Now we can compile and run this code:

```
$ gcc piper.c -o piper -O2
$ time ./piper
```

Your mileage may vary but on my machine it takes about 4 seconds to finish. We are sending about 1024*1024=1 MB of data through our pipe, so this doesn't seem like an ideal solution. There's got to be something faster!

Our first idea might be to take out the delays. If we try it, we'll notice that our CPU usage is now pinned, and after 20 seconds or so, we give up. The problem is that because QNX RTOS is realtime; a thread doesn't get rescheduled unless you explicitly do it, which is what the very small delay effectively did for us. If we replace the delay with **sched_yield**(), which explicitly reschedules the thread, we get a remarkable improvement. We still have the extra poling loops though, which is quite wasteful.

It'd be nice if we could just somehow block until someone signals us that there's room in the buffer to read or write data.

You guessed it, there is!

Conditionally yours.

Conditional variables do exactly what we want, and in fact work very closely with mutexes. We can signal that a conditional variable is set with **pthread_cond_signal**(). If we've locked our mutex, and we need to wait for our condition variable to be set, and that conditional variable can be set only in another thread that needs access to the same data (i.e. locks our mutex), we can simply call **pthread_cond_wait**(). It unlocks the mutex we specify, while at the same time blocking until the conditional variable we specify has been set. When the conditional variable we

specify has been set, it once again locks the mutex, and continues. We can set the conditional variable as often as we want; it will be ignored (and not queued) unless there is another thread waiting on it. This means we don't have to add logic to figure out if another thread is actually waiting for a signal before signaling. And as we all know, the less complex the code, the less chance of bugs wandering in.

Using conditional variables also removes the polling from our code, since the conditional variable is set only when there is data to be read/written, so we don't have to loop around again to check. Removing polling is a very good thing.

Lets put the conditional variable code into our application. There's 4 portions we need to modify. First, add a **pthread_cond_t** variable to our **piper_t** structure:

```
typedef struct
 {
  char buf[BSIZE];         // our buffered data
  int rpos;                // current read position
  int wpos;                // current write position
  int full;                // if rpos==wpos, is the buffer full or
empty?
  pthread_mutex_t mutex;   // lock for our buffer
  pthread_cond_t cond;     // our conditional variable
 } piper_t;
```

Now, we replace the lines:

```
    pthread_mutex_unlock(&p->mutex);
    usleep(100);
    continue;
```

with:

```
    pthread_cond_wait(&p->mutex, &p->mutex);
```

in the **pipe_write** and **pipe_read** functions. In this example we know that we only have two threads communicating so we don’t need to re-check the condition predicate. If we were woken up, it was because there is something for us to do (the mutex ensures the mutual exclusion in this case to prevent us from missing a signal from the “other thread”). If we had multiple threads reading and writing the buffer, then we would have to re-validate our predicate before we continued ... and if it wasn’t satisfied then wait again.

The mutex is automatically unlocked while we're waiting for the conditional variable to be set, and then re-locked before returning from the **pthread_cond_wait**() function call.)

We'll also need to add **pthread_cond_signal**() calls to both the **pipe_write** and **pipe_read** functions, just before the mutexes are unlocked:

```
   pthread_cond_signal(&p->cond);
   pthread_mutex_unlock(&p->mutex);
```

Finally, we'll need to initialize and destroy the conditional variable in our **main**():

```
main()
{
  piper_t p;
  pthread_t rthread;
  void *retval;
  // make sure the buffer is clear before we start
  memset(&p,0,sizeof(p));
  // initialize our pipe's mutex
  pthread_mutex_init(&p.mutex,NULL);
  // initialize our conditional variable
  pthread_cond_init(&p.cond,NULL);
  // start our reader thread
  pthread_create(&rthread,NULL,&reader,&p);
  // run our writer
  writer(&p);
  // wait for the reader thread to finish
  pthread_join(rthread,&retval);
  // destroy our pipe's mutex
  pthread_mutex_destroy(&p.mutex);
  // destroy our pipe's conditional variable
  pthread_cond_destroy(&p.cond);
}
```

If we recompile and run it again as before, we see (to put it mildly) a remarkable improvement. On my machine, it takes under 1/20th of a second! Conditional variables are very handy, wouldn't you say?

That's the end?

I hope I've illustrated at least one use for conditional variables. The best use for them is in places where we want to signal a blocked thread on the off chance that it might be waiting for our signal, but doesn't have to be. It can make life much easier in many multithreaded applications.