Subject: Kqueue/epoll based interface for TcpSocket and WebSocket
Posted by shutalker on Fri, 06 Apr 2018 13:39:53 GMT
View Forum Message <> Reply to Message

Hello all!
I'm working on the implementation of a cross-platform interface that encapsulates event multiplexing interfaces (kqueue/epoll/select). This interface is planned to be used with TcpSockets and WebSockets instead of SocketWaitEvent. The reason why I need this interface is that the select interface does not scale well on large number of sockets, therefore it's useless in developing of high-load server that must keep a lot of tcp connections.
It would be great if that kind of interface become native for UPP. It must be compatible with existing code base, so there is no necessity in modification of existing users' projects. At the same time it can be included and used in user's code like SocketWaitEvent.
At the moment I'm implementing the following interface:

```
template <class T_SOCKET>
class SocketEventQueue: NoCopy
{
public:
 SocketEventQueue(): errorCode(NOERR) { InitEventQueue(); }
 ~SocketEventQueue();

 bool ClearEventQueue();
 QueueHandler GetQueueHandler() const;

 bool IsError() const { return errorCode != NOERR; }
 ErrorCode GetErrorCode();

 bool SubscribeSocketRead(const T_SOCKET &sock);
 bool SubscribeSocketWrite(const T_SOCKET &sock);
 bool SubscribeSocketReadWrite(const T_SOCKET &sock);

 bool DisableSocketRead(const T_SOCKET &sock);
 bool DisableSocketWrite(const T_SOCKET &sock);
 bool DisableSocketReadWrite(const T_SOCKET &sock);

 bool RemoveSocket(const T_SOCKET &sock);

 bool IsSocketSubscribedRead(const T_SOCKET &sock) const { return IsSockeSubscribed(sock,
WAIT_READ); }
 bool IsSocketSubscribedWrite(const T_SOCKET &sock) const { return IsSockeSubscribed(sock,
WAIT_WRITE); }

 bool IsSocketDisabledRead(const T_SOCKET &sock) const { return IsSockeDisabled(sock,
WAIT_READ); }
 bool IsSocketDisabledWrite(const T_SOCKET &sock) const { return IsSockeDisabled(sock,
WAIT_WRITE); }
```

```
 Vector<SocketEvent<T_SOCKET>> Wait(int timeout);
};
```

SocketEvent is a helper class:

```
template <typename T_SOCKET>
class SocketEvent: Moveable<SocketEvent<T_SOCKET>>
{
public:
 SocketEvent(T_SOCKET *sock=nullptr, EventFlag events=0)
 : socket(sock)
 , triggeredEvents(events)
 {}

 T_SOCKET *GetSocket() const { return socket; }

 bool IsTriggeredRead() const { return triggeredEvents & WAIT_READ; }
 bool IsTriggeredWrite() const { return triggeredEvents & WAIT_WRITE; }
 bool IsTriggeredException() const { return triggeredEvents & WAIT_IS_EXCEPTION; }

 void SetTriggeredRead(bool triggerState=true) { SetTrigger(triggerState, WAIT_READ); }
 void SetTriggeredWrite(bool triggerState=true) { SetTrigger(triggerState, WAIT_WRITE); }
 void SetTriggeredException(bool triggerState=true) { SetTrigger(triggerState,
WAIT_IS_EXCEPTION); }

private:
 T_SOCKET *socket;
 EventFlag triggeredEvents;

 void SetTrigger(bool triggerState, EventFlag event)
 {
  if(!triggerState)
  {
   triggeredEvents &= ~event;
   return;
  }

  triggeredEvents |= event;
 }
};
```

How do you like the idea?

There are two problems.

The first problem is related to the current implementation of TcpSocket::RawWait(...). It uses select(...) system call for determining possibility of reading/writing data or exceptional state of socket. As far as I can understand, it means that server with large nubmer of sockets will work slowly anyway. I patched TcpSocket::RawWait(...) for BSD platform on my local machine (see below) before I started to implement the interface. Now I think that it's possible to use SocketEventQueue in purpose of determining socket state instead of raw kqueue/epoll/select. What do you think about this?
But there is another problem related to kqueue/epoll reaction on socket closing for both my patch and SocketEventQueue. So here's the patch:

```
#ifdef PLATFORM_BSD
 timespec *tvalp = NULL;
 timespec tval;
 if(end_time != INT_MAX || WhenWait) {
  to = max(to, 0);
  tval.tv_sec = to / 1000;
  tval.tv_nsec = 1000000 * (to % 1000);
  tvalp = &tval;
  if (to)
   LLOG("RawWait timeout: " << to);
 }

 struct kevent eventrx, eventw;
 struct kevent triggeredEvents[2];
 int kq;
 int eventFlags = EV_ADD | EV_ONESHOT;

 if( ( kq = kqueue() ) == -1 ) // queue fd should be created once at the moment of socket opening
 {         // and closed at the moment of socket closing
          // the same is for SocketEventQueue object

  LLOG("kq = kqueue() returned -1");
  SetSockError("wait");
  return false;
 }

 if(flags & WAIT_READ)
 {
  EV_SET( &eventrx, socket, EVFILT_READ, eventFlags, 0, 0, NULL );
  if( kevent( kq, &eventrx, 1, NULL, 0, NULL ) == -1 )
  {
   LLOG("kevent( kq, &eventrx, 1, NULL, 0, NULL ) returned -1");
   SetSockError("wait");
   close(kq);
   return false;
  }
 }
```

```
 if(flags & WAIT_WRITE)
 {
  EV_SET( &eventw, socket, EVFILT_WRITE, eventFlags, 0, 0, NULL );
  if( kevent( kq, &eventw, 1, NULL, 0, NULL ) == -1 )
  {
   LLOG("kevent( kq, &eventw, 1, NULL, 0, NULL ) returned -1");
   SetSockError("wait");
   close(kq);
   return false;
  }
 }

 int avail = kevent( kq, nullptr, 0, triggeredEvents, 2, tvalp ); // here is the problem if
socket
                                                                  // works in blocking mode
                                                                  // or if timeout is too long
 close( kq );
#else
    // default select implementation
```

Now let's imagine the situation:

```
TcpSocket server; // passes through TcpSocket::Listen()

...

void Server() // runs in several threads
{
 static StaticMutex serverMutex;

 while(!Thread::IsShutdownThreads())
 {
  TcpSocket     client;
  bool acceptStatus;

  {
   Mutex::Lock __(serverMutex);
   //acception is in blocking mode
   acceptStatus = client.Accept(server); // calls TcpSocket::RawWait(...)
  }

  ... // connection handling
 }
}
```

...

```
void SignalHandler(int sig)
{
 server.Close(); // Doesn't interrupt kevent system call in TcpSocket::RawWait(...)
   // close(socket) just makes kqueue to delete all events
   // associated with socket descriptor from it's kernel queue

 Thread::ShutdownThreads();
}
```

So I can't normally terminate the server if I work with sockets in blocking mode.
Do you have any ideas how to interrupt kevent waiting loop?
I've tried to call shutdown(socket, SD_BOTH) for sockets that hadn't been passed through TcpSocket::Listen(), and it works for me.
But I still can't deal with listening socket. Solution I've found is to use pipe-trick: read-end descriptor attaches to kqueue/epoll, and write-end descriptor attaches to socket. When socket closes, it writes some data in pipe with write-end descriptor.
But it means that socket must hold all queue write-end descriptors it was attached.
Could you help me with my problem?

Subject: Re: Kqueue/epoll based interface for TcpSocket and WebSocket
Posted by mirek on Mon, 30 Apr 2018 09:20:50 GMT
View Forum Message <> Reply to Message

Sorry for the delay...

Quote:

```
template <class T_SOCKET>
class SocketEventQueue: NoCopy
{
public:
 SocketEventQueue(): errorCode(NOERR) { InitEventQueue(); }
 ~SocketEventQueue();

 bool ClearEventQueue();
 QueueHandler GetQueueHandler() const;

 bool IsError() const { return errorCode != NOERR; }
 ErrorCode GetErrorCode();

 bool SubscribeSocketRead(const T_SOCKET &sock);
 bool SubscribeSocketWrite(const T_SOCKET &sock);
 bool SubscribeSocketReadWrite(const T_SOCKET &sock);
```

```
 bool DisableSocketRead(const T_SOCKET &sock);
 bool DisableSocketWrite(const T_SOCKET &sock);
 bool DisableSocketReadWrite(const T_SOCKET &sock);

 bool RemoveSocket(const T_SOCKET &sock);

 bool IsSocketSubscribedRead(const T_SOCKET &sock) const { return IsSockeSubscribed(sock,
WAIT_READ); }
 bool IsSocketSubscribedWrite(const T_SOCKET &sock) const { return IsSockeSubscribed(sock,
WAIT_WRITE); }

 bool IsSocketDisabledRead(const T_SOCKET &sock) const { return IsSockeDisabled(sock,
WAIT_READ); }
 bool IsSocketDisabledWrite(const T_SOCKET &sock) const { return IsSockeDisabled(sock,
WAIT_WRITE); }

 Vector<SocketEvent<T_SOCKET>> Wait(int timeout);
};
```

Why is T_SOCKET a template parameter? Because of websocket?

What is DisableSocketRead supposed to do? Opposite of Subscribe?

If I am right about T_SOCKET, I have to disagree with the interface a bit.

Particulary, I think

Vector<SocketEvent<T_SOCKET>> Wait(int timeout);

is clumsy - this will IMO cause problems with mapping T_SOCKET back to its "processes".

When I was thinking about how to proceed with this, I was considering to simply expand SocketWaitEvent. I believe that the best would probably be to treat it as full array of sockets, using indices to identify the 'process'. Something like

```
class SocketWaitEvent {
......
public:
 void  Clear()                                  { socket.Clear(); }
 void  Add(SOCKET s, dword events)                    { socket.Add(MakeTuple((int)s, events)); }
 void  Add(TcpSocket& s, dword events)                 { Add(s.GetSOCKET(), events); }

 int   Wait(int timeout);
 dword Get(int i) const;
 dword operator[](int i) const                       { return Get(i); }
```

```
// new:
        void  Set(int ii, TcpSocket& s, dword events);
        void  Insert(int ii, TcpSocket& s, dword events);
        void  Remove(int ii, TcpSocket& s, dowrd events);

        Vector<int> WaitEvent(int timeout);
   // or perhaps
        Vector<Tuple<int, dword>> WaitEvent(int timeout); // dword part contains Get bitmask

// maybe:
        void  Clear(int ii); // makes index empty
        int   FindEmpty() const; // finds the first index that is empty


 SocketWaitEvent();
};
```

Is there a reason to make things more complicated that this?

Quote:
The first problem is related to the current implementation of TcpSocket::RawWait(...). It uses select(...) system call for determining possibility of reading/writing data or exceptional state of socket. As far as I can understand, it means that server with large nubmer of sockets will work slowly anyway. I patched TcpSocket::RawWait(...) for BSD platform on my local machine (see below) before I started to implement the interface. Now I think that it's possible to use SocketEventQueue in purpose of determining socket state instead of raw kqueue/epoll/select. What do you think about this?

I agree.

Quote:

But there is another problem related to kqueue/epoll reaction on socket closing for both my patch and SocketEventQueue. So here's the patch:

```
#ifdef PLATFORM_BSD
 timespec *tvalp = NULL;
 timespec tval;
 if(end_time != INT_MAX || WhenWait) {
  to = max(to, 0);
  tval.tv_sec = to / 1000;
  tval.tv_nsec = 1000000 * (to % 1000);
  tvalp = &tval;
  if (to)
```

```
  LLOG("RawWait timeout: " << to);
 }

 struct kevent eventrx, eventw;
 struct kevent triggeredEvents[2];
 int kq;
 int eventFlags = EV_ADD | EV_ONESHOT;

 if( ( kq = kqueue() ) == -1 ) // queue fd should be created once at the moment of socket opening
 {         // and closed at the moment of socket closing
          // the same is for SocketEventQueue object

  LLOG("kq = kqueue() returned -1");
  SetSockError("wait");
  return false;
 }

 if(flags & WAIT_READ)
 {
  EV_SET( &eventrx, socket, EVFILT_READ, eventFlags, 0, 0, NULL );
  if( kevent( kq, &eventrx, 1, NULL, 0, NULL ) == -1 )
  {
   LLOG("kevent( kq, &eventrx, 1, NULL, 0, NULL ) returned -1");
   SetSockError("wait");
   close(kq);
   return false;
  }
 }

 if(flags & WAIT_WRITE)
 {
  EV_SET( &eventw, socket, EVFILT_WRITE, eventFlags, 0, 0, NULL );
  if( kevent( kq, &eventw, 1, NULL, 0, NULL ) == -1 )
  {
   LLOG("kevent( kq, &eventw, 1, NULL, 0, NULL ) returned -1");
   SetSockError("wait");
   close(kq);
   return false;
  }
 }

  int avail = kevent( kq, nullptr, 0, triggeredEvents, 2, tvalp ); // here is the problem if
socket
                                                                  // works in blocking mode
                                                                  // or if timeout is too long
  close( kq );
#else
    // default select implementation
```

Now let's imagine the situation:

```
TcpSocket server; // passes through TcpSocket::Listen()

...

void Server() // runs in several threads
{
 static StaticMutex serverMutex;

 while(!Thread::IsShutdownThreads())
 {
  TcpSocket     client;
  bool acceptStatus;

  {
   Mutex::Lock __(serverMutex);
   //acception is in blocking mode
   acceptStatus = client.Accept(server); // calls TcpSocket::RawWait(...)
  }

  ... // connection handling
 }
}

...

void SignalHandler(int sig)
{
 server.Close(); // Doesn't interrupt kevent system call in TcpSocket::RawWait(...)
   // close(socket) just makes kqueue to delete all events
   // associated with socket descriptor from it's kernel queue

 Thread::ShutdownThreads();
}
```

So I can't normally terminate the server if I work with sockets in blocking mode.
Do you have any ideas how to interrupt kevent waiting loop?
I've tried to call shutdown(socket, SD_BOTH) for sockets that hadn't been passed through TcpSocket::Listen(), and it works for me.
But I still can't deal with listening socket. Solution I've found is to use pipe-trick: read-end descriptor attaches to kqueue/epoll, and write-end descriptor attaches to socket. When socket closes, it writes some data in pipe with write-end descriptor.

But it means that socket must hold all queue write-end descriptors it was attached.
Could you help me with my problem?[/quote]

Well, I was fighting with this one too, years ago. Thats nasty little problem there.

I the end, I believe that the best solution is to make Accept return until there are any active threads by doing localhost connect.

```
TcpSocket s;
s.Connect("127.0.0.1", port);
```

You can check Skylart/App.cpp.

Another option, not always applicable, is not to bother and let the signal kill the application just as it is supposed to.... :)

Subject: Re: Kqueue/epoll based interface for TcpSocket and WebSocket
Posted by mirek on Sat, 05 May 2018 15:50:29 GMT
View Forum Message <> Reply to Message

Quote:
Hi, Mirek! Sorry for delay

Well, my delay was worse than yours... :)

Quote:
Thank you for the answer to this post:
https://www.ultimatepp.org/forums/index.php?t=msg&th=103 17&start=0&
>Why is T_SOCKET a template parameter? Because of websocket?
Yes. The interface is intended to be usable with both tcpsockets and websockets

But not at the same time, right? That would be a big problem IMO...

Quote:
>Vector<SocketEvent<T_SOCKET>> Wait(int timeout); is clumsy - this will IMO cause problems with mapping T_SOCKET back to its "processes".
Could you explain, what kind of problems with mapping will cause the interface?
Let me show, how`Wait(...)` can be used:

```
    if(event.IsTriggeredRead())
    {
```

```
if(event.GetSocket()->GetSOCKET() == server.GetSOCKET())
{
		// new connection
}
else
{
   CLOG("SERVER incoming data: " << event.GetSocket()->GetLine());
```

Now what. Where will these data go?

IMO, there will be a class instance that will provide the communication with single client and you will want to let it read the data. You will probably have some container of these instances. In the end, you need to map incomming event to a class instance somehow. If all ID you have is GetSocket, you will need map socket->instance.

Quote:
Main idea was to serve only 'triggered' sockets without necessity of searching them in array.

Well, I think what I see is quite opposite :) But I might be wrong.

Quote:
Why `Set(int ii, TcpSocket& s, dword events)`, `Insert(int ii, TcpSocket& s, dword events)` and `Remove(int ii, TcpSocket& s, dword events)` use `TcpSocket& s` but not `SOCKET s` as a parameter?

Sorry, that was my mistake. Actually, the interface between this and actual sockets is something to be carefully considered. With websocket, I have attempted some initial solution with GetWaitEvents and probably AddTo or something like that.

Quote:
Are `Clear(int ii)` and `FindEmpty()` intended to keep user's `Array<***Socket>` and SocketWaitEvent containers synchronized? Does it mean that user can replace one socket in array with another?

Yes.

Quote:
I think prototype for `Remove(...)` should be like this
```
// completely remove events attached to socket from event queue (usable for select/kqueue)
// for epoll it will work like 'Disable(int ii, dword events)' (see below) until there are still events that socket is subscribed for
void  Remove(int ii, dword events);
```

Another mistake in my proposal, it should have simply been Remove(int ii).

Quote:
and there should be another member function in SocketWaitEvent interface:
```// do not remove event attached to socket from queue but disable notification delivery
void  Disable(int ii, dword events);```

Disable would be really hard to use. I am pretty sure that you need some sort of Set with new set of bit flags. Check GetWaitEvents (in HttpRequest and WebSocket).

Quote:
The problem i can see there is that we must to look through the whole `Vector<Tuple<int, dword>> socket`  to find indices of triggered sockets after select/epoll/kqueue 'wait for events' system call has returned. It may work slow on large number of sockets. (edited)

So exactly the same problem I can see with your proposal :)

Anyway, here the idea is that you will maintain that 'int' to be directly mapped to the index of instance of class that handles the communication.

I think here I need to be a little bit more clear about the Clear and FindEmpty - these are meant to make possible to have 'array with holes', in a sense that some indicies are not connected to active instances so that you do not need to do inserts/removes to the array. This might be overkill, as we can make inserts/removes really fast with InArray...

Mirek

---

Subject: Re: Kqueue/epoll based interface for TcpSocket and WebSocket
Posted by shutalker on Mon, 07 May 2018 07:54:53 GMT
View Forum Message <> Reply to Message

Quote:
But not at the same time, right? That would be a big problem IMO...

Yes.

Quote:
IMO, there will be a class instance that will provide the communication with single client and you will want to let it read the data. You will probably have some container of these instances. In the end, you need to map incomming event to a class instance somehow. If all ID you have is GetSocket, you will need map socket->instance.

There is no need in mapping socket->instance. Let me clear things up with how
```

SocketEventQueue::Wait(...) works. First of all, sockets that need to be tracked for events are attached to the kernel event queue via SocketEventQueue::SubscribeSocket****(...). Then Wait(...) is invoked to obtain pending events from kernel event queue:

```
   // kqueue-based 'wait for events' system call
   int avail = kevent(queueFD, nullptr, 0, rawEvents, TRIGGERED_SET_SIZE,  tvalp);

   if(avail < 0)
   {
      // error handling
   }

   VectorMap<SocketHandler, SocketEvent<T_SOCKET>> triggered;

   for(int iEvent = 0; iEvent < avail; ++iEvent)
   {
      SocketHandler  handler = rawEvents[iEvent].ident;  // kqueue based implementation to obtain socket descriptor

      if(handler == ancillaryReadFD)
      {
         // listening socket close handling (pipe-trick)

                  ...

         // sockhdlr is a listening socket descriptor that was closed
         // socket is a pointer to correnponding socket class instance
         SocketEvent<T_SOCKET>  &event  = triggered.GetAdd(sockhdlr, SocketEvent<T_SOCKET>(socket));
         event.SetTriggeredException();

         continue;
      }

      T_SOCKET *socket  = socketHandler.Get(handler, nullptr);  //socketHandler is VectorMap<SOCKET, *SOCKET_T>

      // if socket was removed after ancillaryReadFD data was delivered
      // pipe-trick implementation to track listening socket closing
      if(!socket)
         continue;

      SocketEvent<T_SOCKET>  &event  = triggered.GetAdd(handler, SocketEvent<T_SOCKET>(socket));

      if(rawEvents[iEvent].flags & (EV_EOF | EV_ERROR))
         event.SetTriggeredException();
```

```
        if(rawEvents[iEvent].filter == EVFILT_READ)
            event.SetTriggeredRead();

        if(rawEvents[iEvent].filter == EVFILT_WRITE)
            event.SetTriggeredWrite();
    }

    return pick(triggered.GetValues());
```

I use VectorMap containers instead of Vector as SocketEventQueue member, therefore searching of triggered sockets may be a little bit more effective than iterating over the Vector.
All events instances are generated every time Wait(...) was invoked, so the only necessity for user is to keep socket instance pointer valid (by placing sockets in Array, for example).

I really like your idea to expand existing SocketWaitEvent interface and work with indices, not helper class instances. And I think it's possible to use VectorMap<SOCKET, int> with VectorMap::Find(...) returning index instead of Vector<Tuple<SOCKET, dword>>. What do you think about this?

Subject: Re: Kqueue/epoll based interface for TcpSocket and WebSocket
Posted by mirek on Tue, 08 May 2018 10:02:47 GMT
View Forum Message <> Reply to Message

shutalker wrote on Mon, 07 May 2018 09:54Quote:
But not at the same time, right? That would be a big problem IMO...

Yes.

Quote:
IMO, there will be a class instance that will provide the communication with single client and you will want to let it read the data. You will probably have some container of these instances. In the end, you need to map incomming event to a class instance somehow. If all ID you have is GetSocket, you will need map socket->instance.

There is no need in mapping socket->instance. Let me clear things up with how SocketEventQueue::Wait(...) works. First of all, sockets that need to be tracked for events are attached to the kernel event queue via SocketEventQueue::SubscribeSocket****(...). Then Wait(...) is invoked to obtain pending events from kernel event queue:

I think you got me wrong here. I am speaking about interface (and its use in client code), not about implementation.

Mirek
