Chris Mennie
97024327
Chris Evans
97070019
Section 1
Group 18

CS452 RTOS Assignment #4

Contents

Introduction

The path to our code is "~cl2evans/cs452/RTOS-A4-submit/".

To build your own apps, you'll want to take a look at "user/syscall.h" and "structures/intQueue.h", "structures/message.h", and "structures/store.h". These are the most important header files. As well, "app/UserGlobals.h" is handy. To compile an app, you need to use the crt0.S that's found in the app directory, and compile in "app/taskInit.c". Task startup is very different here.

Why This Kernel Is a Real Time Kernel

First, we know that we never miss any interrupts. The amount of time spent with interrupts off is always smaller than every interrupt's fastest cycle. Thus we always catch each interrupt. We also keep track of the time in the kernel. Thus we know we never loose ticks. So we're always able to service everything, not lose ticks, and stay real timeish.

intQueue and Store HOWTO

The intQueue and Store structures are vital. The appropriate header files can be found in the structures directory. Essentially, they're both wrappers around arrays of integers. If we were using C++, they'd be classes, or templates.

The Store is a simple array of ints, with handy functions to add and remove entries. Complexity is linear. The intQueue on the other hand is a ring buffer. Thus it's easy to push and pop numbers on and off.

To use them, you must declare (for example) and intQueue, and then a buffer to go along with it. Then you need to call queueInit to associate them together. Like:

intQueue pollInputQueue;
int pollInputQueueBuffer[12];
queueInit(&pollInputQueue, &pollInputQueueBuffer[0], 12, "PollInputQueue", IntQueue, gl_MyPid);

Unfortunately this is all necessary. The "IntQueue" magic number is for sanity checking through the kernel. If you are creating a message queue, you need to set this to "MessageQueue" (or use MessageQueueInit).

The intended use of these structures are for message queues, and lists of message queues. They're very handy.

IPC HOWTO

IPC in this kernel is quite the beast. Instead of a single message queue per task, each task can create any number of message queues. On top of that, it's possible to not only receive on a single queue at a time, but on a number of queues at a time. Thus IPC is handled differently.

The basic idea, is that everything that's being passed around is a Message (found in structures/message.h). This structure keeps track of the actual buffer being sent, and the reply buffer to respond to, etc. So a Send essentially copies the Message struct to the destination queue, and a Receive retrieves it. Once a Message is received, a call to ReceiveData is needed to get the actual message contents. It may seem a bit much, but it allows a great deal of flexibility.

Say you wanted to wait for a message to arrive on any one of a number of queues. If you add the queues you want to use to a Store, then you can receive on that Store. When a message comes in on one of those queues, you'll be woken up and told which queue it was received on. For example:

Store_Init(&commandRequestStore, IntStore, &commandRequestStoreBuffer[0], 7);
AddMessageQueueToStore(&commandRequestStore, &stuffPollingQueue);
AddMessageQueueToStore(&commandRequestStore, &trainMoveRequestQueue);

activeQueue = ReceiveQueue(&commandRequestStore, &inMessage);

So in this example, a receive is done on the store, but the task is woken up when a message is sent to either queue.

To allow other tasks to send to your queue, you need to register it with the nameserver. All that's needed to do this, is a call to RegisterWithNameserver(). Later, a task that wants access to this server can find it with GetEntryFromNameserver() or WaitForRegistry(). In fact, WaitForRegistry() is a great way to ensure you have a communication path to the queue you want.

Good examples of this are the servers and notifiers in the daemons directory.

It is ok to send NULL messages, and reply to a message with a NULL. However, to reply with NULL, the sender must have sent a NULL message.

One thing to note, is all the addresses used in the Message are absolute values. Thus we can use absRead/absWrite on them.

User API

Builtin Task Data. For free, the following data are set in each task. The only cavet is that the name server and clock server queues aren't necessarily going to be set correctly for each notifier or deamon.

Variable Name	**Variable Description**
baseDS	Absolute address of the start of the .data segment
dataTop	Absolute address of the end of the stack
gl_currentTime	The current time in ticks since the OS started
gl_MyPid	The task's pid
gl_MyParentPid	The task's parent's pid
gl_MyPriority	The task's priority
gl_NameserverQueryQueue	Absolute address of the name server's query queue
gl_NameserverRegistryQueue	Absolute address of the name server's registry queue
gl_ClockserverRequestQueue	Absolute address of the clock server's request queue

General.

Useful Defines.

Define name	Description
MAXPRIORITIES	32
MAXUSERPRIORITY	16
DAEMON_PRIORITY	23
NOTIFIER_PRIORITY	23
INVALID_PID	0xffffffff
SYSCALL_MAGICNUM	0xfe783a51
TICKS_PER_SEC	20
SystemCall(x)	Call kernel system call x, placing return value into gl_retval
PushVal(x)	Push value x onto stack
PopVal(x)	Pop top value off stack and store it in x
cabsGrant()	Change DS and SS to point to the entire address space
cabsRevoke()	Restore DS and SS back to the task's original DS and SS
DEBUG	For debugging, this needs to be set to a debug level
DEBUGprintf(level, fmt, args...)	If DEBUG >= level, then do a printf with the given fmt and args

MAXPROCESS	129
pid_type	unsigned int
Bool	char
TRUE	1
FALSE	0
NULL	0
MAXREGISTRYNAME	30

EventTypes.

The event types are an enumerated type containing: PIT_Event, RTC_Event, Serial0_Error_Event, Serial0_Send_Event, Serial0_Receive_Event, Serial1_Send_Event, Serial1_Receive_Eve nt, Serial1_Error_Event, and NumEvents

int AwaitEvent(EventTypes event).

This function waits for a given event. The task blocks until the external event occurs, at which point the kernel places the task back on the ready queue. When a task blocks on an event, it is placed on a queue. Therefore, if more than one task blocks on the same event, the second (and so on) task will have to wait for the second (and so on) event before being woken up. It is expected however, that this function only be used by notifiers, and only one task per event will make use of it.

A return value of 0 means there were no problems. A return value of -1 indicates an error occurred, which in this case means an invalid event was given.

void Pass(void).

Causes task to be rescheduled.

int Create(char programName, int priority).*

Creates a new task of given priority from the entry in the bindfile with the given name. The pid of the new task is returned.

void Exit(void).

Causes the task to never be rescheduled again.

Information Functions.

int MyPid(void).

Return task's pid.

int ValidPid(int pid).

If the given pid is a created task, that isn't dead or zombied, this returns 1. Otherwise 0 is returned.

int MyParentPid(void).

Returns the pid of the task's parent. 0 means the task's parent is the kernel.

int MyPriority(void).

Returns the creation priority of the current task.

Name Registry.

void SetNameserverRegistryQueue(void pointer).*

Register with the kernel the given pointer as the name server registry message queue. Only the name server should do this. Once the first registry has been done, any subsequent ones will be rejected (nothing in the kernel will change). The given pointer is relative to the task's baseDS (in other words, you just pass a pointer as normal), and it will be converted to an absolute address in the kernel.

void GetNameserverRegistryQueue(void).*

Return the absolute address of the name server's registry message queue. For user tasks this function is depreciated since the returned value is placed in the task's gl_NameserverRegistryQueue global variable. For daemons and notifiers however, this function is of use since there is no guarantee when they are created, that the gl_NameserverRegistryQueue variable is correct.

Upon return, the value of the gl_NameserverRegistryQueue global variable is updated to the given pointer value. If the result is NULL, then the name server's registry queue hasn't been registered with the kernel.

void SetNameserverQueryQueue(void pointer).*

Register with the kernel the given pointer as the name server query message queue. Only the name server should do this. Once the first registry has been done, any subsequent ones will be rejected (nothing in the kernel will change). The given pointer is relative to the task's baseDS (in other words, you just pass a pointer as normal), and it will be converted to an absolute address in the kernel.

void GetNameserverQueryQueue(void);*

Return the absolute address of the name server's query message queue. For user tasks this function is depreciated since the returned value is placed in the task's gl_NameserverQueryQueue global variable. For daemons and notifiers however, this function is of use since there is no guarantee when they are created, that the gl_NameserverQueryQueue variable is correct.

Upon return, the value of the gl_NameserverQueryQueue global variable is updated to the given pointer value. If the result is NULL, then the name server's query queue hasn't been registered with the kernel.

Bool RegisterWithNameServer(char entryName, void* store).*

Add an entry to the name server with the given name and pointer. It is assumed that in general pointers will be registered with the name server (for example, pointers to message queues), and so baseDS is implicitly added to the value of "store" when this function is called. To register a value that is not offset by baseDS, it is necessary to explicitly subtract baseDS from "store" when making this call.

If the result is TRUE, then the entry was added to the server. Otherwise there are three reasons why this would fail. If the name server registry message queue hasn't been registered with the kernel, this will fail. If there is already an entry in the name server with the same name, then this will fail. Finally, if there was some sort of internal error in the name server, then this will fail (though this should never be the case).

void GetEntryFromNameServer(char* entryName).*

Return the pointer stored in the name server that is associated with the given name.

A result of NULL indicates one of three things. First, the name server's query queue may not have been registered with the kernel. Secondly, the entry may not have been found by the name server. Third, it is possible that a value of NULL was registered with the name server (to poison a value, for example). The only way to determine the second case from the third case is to attempt to add an entry to the name server with the given name. If the addition succeeds, then it was the second case, otherwise it was the third. Since there is no way currently to remove an entry from the name server, it is probably a bad idea to poison entries.

Any other result indicates the entry was found, and registered with the name server with the given result.

*void *WaitForRegistry(char *name).*

This function will continually poll the name server for an entry matching the given name. While the name server continues to return NULL, this function will not exit. This can be used to synchronize between tasks (for example, a one shot condition variable could be done with this).

The return value is the same as if GetEntryFromNameServer(name) was called (assuming name existed in the name server).

*Bool RegisterAs(char *entryName).*

Registers with nameserver the task's pid, under the given name. This is intended to conform to CS452 kernel specs. Returns true if registry was successful, FALSE otherwise. Failure reasons are the same as for the RegisterWithNameserver command.

*pid_type WhoIs(char *entryName).*

Returns the pid of the task registered under the given name. Function is equivalent to GetEntryFromNameserver, but with a different return type.

Time.

void SetClockserverNotifyQueue(void pointer).*

Register with the kernel the given pointer as the clock server notify message queue. Only the name server should do this. Once the first registry has been done, any subsequent ones will be rejected (nothing in the kernel will change). The given pointer is relative to the task's baseDS (in other words, you just pass a pointer as normal), and it will be converted to an absolute address in the kernel.

void GetClockserverNotifyQueue(void).*

Return the absolute address of the clock server's notify message queue. This function is intended to be used by the clock server's notifier only. If the result is NULL, then the clock server's notify queue hasn't been registered with the kernel.

void SetClockserverRequestQueue(void pointer).*

Register with the kernel the given pointer as the clock server request message queue. Only the name server should do this. Once the first registry has been done, any subsequent ones will be rejected (nothing in the kernel will change). The given pointer is relative to the task's baseDS (in other words, you just pass a pointer as normal), and it will be converted to an absolute address in the kernel.

void GetClockserverRequestQueue(void).*

Return the absolute address of the clock server's request message queue. For user tasks this function is depreciated since the returned value is placed in the task's gl_ClockserverRequestQueue global variable. For daemons and notifiers however, this function is of use since there is no guarantee when they are created, that the gl_ClockserverRequestQueue variable is correct.

Upon return, the value of the gl_ClockserverRequestQueue global variable is updated to the given pointer value. If the result is NULL, then the clock server's registry queue hasn't been registered with the kernel.

void Delay(int wait).

Delay for at least "wait" ticks.

void DelayUntil(int wakeUp).

Delay until the current time is greater than or equal to "wakeUp" (which is given in system ticks).

unsigned int GetTime(void).

Returns the current time in ticks since the start of the kernel.

Serial Functions.

*char Get(int port, Bool *errorOccured).*

Receives a byte from the given serial port, and returns it. The port can be either WYSE or TRAIN. The value of "errorOccured" indicates if there was some problem with receiving the byte. This value is TRUE, if there was an error, or FALSE otherwise. If no error condition is required, a NULL may be passed in for "errorOccured".

Bool Put(int port, char Byte).

Write a byte out to the given serial port. The port can be either WYSE or TRAIN. The return value is TRUE if the write was successful, and FALSE otherwise.

*int Write(int port, char *buffer, unsigned int length).*

Write a string out to the given serial port. The port can be either WYSE or TRAIN. This function is atomic, in the sense that the string will be written out in it's entirety before any other Write or Put request is serviced. A return value of TRUE means the Write was a success. A return of FALSE, means an error occured, and -1 means the port given was invalid.

IPC Functions.

void MessageQueueInit(intQueue queueToInit, int* queueBuffer, uint msgQueueSize, char name[]).*

Initializes a message queue to use the given buffer. msgQueueSize is the size of the buffer (in elements, not bytes), and name is the name of the queue (used for debugging).

void AddMessageQueueToStore(Store store, intQueue* queue).*

Add a message queue to a queue store. Essentially, the absolute address of the message queue is added to the store (to allow other tasks to access it).

*int Send(pid_type Pid, char *message, unsigned int messageLen, char *replyBuf, unsigned int replyBufLen).*

As specified in the kernel specs, this sends a message to task "Pid". Implicitly this uses SendQueue, to a default message queue set up for the task.

*pid_type Receive(char *receiveBuf, unsigned int receiveBufLen, Bool *errorOccured).*

Receive a message from the task's default message queue. Works as specified in the kernel specs. Implicitly this consists of a ReceiveQueue and ReplyMessage.

*unsigned int Reply(pid_type sender, char *replyBuf, unsigned int replyBufLen).*

Reply to a sending task, as specified in the kernel specs. It should be noted that it is possible to wake up a task that isn't blocked on the task doing the waking up.

*int SendQueue(void *messageQueue, Message *message).*

Sends a message to the given queue. Returns 0 on success, and -1 on failure.

*intQueue *ReceiveQueue(void *MessageQueueOrQueueList, Message *message).*

Waits for a message on the given queue, or queue list. The received message is written to "message" and the queue it was received on, is returned. If receiving on a queue list (store), the order that each queue was added to the list matters. The later a queue was added, the greater it's priority is. Queues with greater priority will be serviced before those with lower priority. It is not possible for two queues to have the same priority.

*int SendQueueNoBlock(void *messageQueue, Message *message).*

Sends a message to the given queue. Returns 0 on success, and -1 on failure. This call doesn't block, unlike SendQueue.

*intQueue *ReceiveQueueNoBlock(void *MessageQueueOrQueueList, Message *message).*

Same as ReceiveQueue, except it doesn't block.

*unsigned int ReceiveData(Message *message, char *receiveBuf, unsigned int receiveBufLen).*

After receiving a message, this function will retrieve the actual message contents. The length of the data retrieved is returned, or 0 on error.

unsigned int ReplyMessage(Message message, char *replyBuf, unsigned int replyLen).*

Replies to a message with the given response. A reply of NULL is valid, only if the sender sent a NULL message.

Program Discussion

We've made a number of nice changes to the code. We've refactored the system call handler in the kernel. Instead of that nasty if/else construct, we now use an array of function pointers to polymorphicaly do the handling. Now the handler is very short and sweet and clean.

Init

Because of how we've chosen our priorities, the startup procedure begins with the idle task. This task then makes itself the maximum priority (with a system call), and creates all the daemons and notifiers, and the "menu" user task, before returning back to the lowest priority (also with a system call). Then all the daemons and notifiers have a chance to do their respective setup before the "menu" task runs. This way, by the time the first user program runs, we can assert that all the daemons have been created, and have set up all their external interfaces.

It was decided that each task should have a number of data on hand before starting. Instead of calling MyPid() or GetNameserverRegistryQueue(), it made sense to us to provide this information directly to the task. We added a number of data to the end of crt0.S, and now instead of jumping to main, we jump to taskInit(). This function makes a system call that tells the kernel where it can find a number of data for the task. Then the kernel updates these values with the relevant data. These values include the task's pid, parent pid, priority, name server queues, clock server request queue, and the current time. The location of the current time is stored in the task's PCB.

Once the kernel has filled in all this information and returned control to the task, the main function is called.

The rationale behind this is that we wanted the task to always have the current time. Since we added functionality for one variable, it was trivial to do it for any number. The data that was included is always set and static by the time the user task runs, so there didn't seem to be any need make the user task make kernel calls to look it up.

As a consequence, the only reason for keeping around functions like GetNameserverRegistryQueue() is for daemons, who may not at that point have had the correct value set.

Scheduling

We divided priorities up into lower, userland priorities, and upper, daemon/notifier priorities. We have also decided to provide user tasks with the current time every time they're scheduled. Since we have a hook to the tasks's gl_currentTime value, we just copy the current kernel time into there during the context switch.

Message Passing

A default message queue is created for a task, and it is used to implement the Send(), Receive(), Reply() primitives as they are listed in the given kernel specification. However, tasks may create additional message queues, and ReceiveQueue() on some subset of those. This subset is stored in a data structure called a Store. When looking through these message queue sets, we needed to assure that nobody else was messing with them. We also needed to ensure that the tasks were in the proper states at the proper time. For example, we were having problems earlier on with the receiver trying to reply block a task before it was send blocked. In the end we went with using the super priority code for larger sections of Send and Receive. This code changes the priority of a process so that it is guaranteed to be scheduled before all others. We ensure that we have mutual exclusion, while allowing the kernel to continue handling interrupts.

Since we treat userland tasks differently from daemons and notifiers, we needed to add a new kernel call. The SYSCALL_MAKESUPERSUPERPRIORITY call changes the task's priority to MAXPRIORITIES-1, and is used for the daemons and notifiers.

It is permissible to reply to a message with a NULL response, providing the reply buffer was also NULL. We decided that if a message is desired, then the receiver must reply with one. The receiver can only reply with a NULL message if the sender didn't want one.

Name Server

Since we implemented message passing differently (with explicit queues as opposed to implicit ones), our nameserver has some different uses. We've added a WaitForRegistry() call, which essentially continues to poll the name server until the given name is registered. This is useful for synchronization between master and servant tasks. We have also used it to solve bootstrapping problems. For example, say we want to have a bunch of user tasks enumerate themselves starting at some arbitrary value. They wait for a specific base value to be registered with the name server, and then use that to calculate their respective values (using their pids as well).

While the nameserver was originally intended to register pointers, it can also be used to register any given number with the RegisterAs() system call. The WhoIs() system call is the analogous call for a lookup.

Clock Server

The clock server is fairly simple. It Recieve()'s on a notify and request queue. When it gets woken up by a request, it expects the message buffer to contain four bytes, which is the absolute time at which the requester wants to be woken up at. By using a simple insertion sort, we add this task and its time to be woken up into a list, sorted by time. The clock server doesn't reply, so as to keep the process sleeping. It is also safe for it to discard all information past the sender pid and time to be woken up, since the eventual reply will be NULL.

When a message is received on the notify queue, the clock server goes through the sorted list, picking off the first element, if the time to wake up is less than or equal to the current time. The clock server replies to the corresponing task, to wake it up. Note that the clock server always has the current time in it's gl_currentTime data member (which the kernel updates every time the server is scheduled to run). This removes the need to make a system call whenever the current time is required.

Associated with the clock server is the clock notifier. It's a simple task that waits for a PIT event, and then sends to the clock server's notify queue to wake it up.

We've added the user commands Delay and DelayUntil, to delay a relative, and specific amount of time, respectively. Ultimately though, Delay makes a call to DelayUntil, since the user task knows what the current time is.

There are also functions to get and set the clock server queues, similar to the ones for the nameserver. They are only needed by the clock server and clock notifier.

Debugger

A small debugger has been added, however, it is mainly useful to see the current state of the processes in the system, and to display the task and register infomation for a task which GPFs. To enter the debugger, press any key on the keyboard, and then press the space bar on the WYSE terminal. More information about what commands are available by typing "help" at the debugger prompt. If you press 'q' instead of space on the WYSE terminal, the machine will perform a soft reset, which is very handy for Eos 0.

Test Harness

A menu system was added in order to run different test cases without having to repost. To make and post the menu and associated test suite, run the shell script ./runTestAll from the UNIX prompt.

From the main menu, you can run tests 1-L. Test K is a submenu for tests based on Send(), Receive(), Reply().

Test 1

Tests the MyPid(), MyParentPid(), MyPriority(), ValidPid(), Exit(), and Pass() syscalls.

Test 2

Tests creation of many processes with a small stack size. No more processes are created once there is no memory left.

Test 3

Similar to test 1, but with more processes.

Test 4

Tests creation of many processes with a large stack size. No more processes are created once there is no memory left.

Test 5

Tests the nameserver primitive RegisterWithNameServer() (not the same as RegisterAs()) and GetEntryFromNameServer() (not the same as WhoIs()). The processes coordinate through the nameserver in order to send messages.

Test 6

Tests the Store data structure. It adds, removes, and finds integers.

Test 7

Similar to 5, except that the test programs are started at the same priority, effectively testing ReceiveQueue() before SendQueue().

Test 8

Tests SendQueue() before ReceiveQueue().

Test 9

This test is the standard eater/ender test given in Part III. Each eater waits "i seconds between each erasure", and thus the eaters don't wait at the end. We expect the order of completion to be { 8, { 10, 9, 6 }, 7, 5, { 4, 3 }, 2 }, and that is the observed output. There is a #define in user/syscall.h, TICKS_PER_SEC, which is normally defined as 20. To speed up the tests, a value as small as 1 can be defined, and the order of completion is not affected.

Test A

This test ensures that a process can wait for zero ticks, a negative number of ticks, and for 30 seconds. The process immediately returns from the Delay(0) and Delay(-5) system calls, and then waits 30 seconds. The observed output matches this behaviour.

Test B

This test runs three processes which each wait depending on the order in which they were run. The first one waits 600 ticks, the next waits 400 ticks, and the final one waits 200 ticks. The test ensures that the sorting of process' wait times is correct, because the first process waits the longest. The observed output matches this behaviour.

Test C

In Part II, two test programs were given which would recursively calculate 6! with message passing. This test was implemented based on the design given. 720 is the final result, which is what we expect.

Test D

Similar to Test C, however, the priorites are set so that the first process sends after the second process has already made a Receive() syscall. We see the expected result of 720.

Test E

This test takes input on the WYSE terminal and echoes it back.

Test F

This test ensures non-blocking send works. Messages are sent and received asynchronously.

Test G

This test times the maximum and minimum times for Send/Receive/Reply with a 16 byte message.

Test H

This test times the maximum and minimum times for Send/Receive/Reply with a 0 byte message.

Test I

This test times the maximum and minimum times for a context switch.

Test J

This test times the maximum and minimum times for a Create() system call. However, this time also includes the setup of the task by crt0.

Test K

(a): tests send with an invalid PID. The return value is -1.

(b): tests reply with an invalid PID. The return value is -1.

(c): tests reply with a PID of a process which is not reply blocked. The return value of Reply is -1.

(d): tests Send() before Receive(). The message is sent correctly.

(e): tests Receive() before Send(). The message is sent correctly.

(f): tests two processes doing a send before receive. Both messages are received correctly.

Test L

Test L is a sample train application, similar to assignment 1. Unfortunately, the sensor updates are occurring on the wrong part of the screen, however, polling for bytes appears to be working.

Assignment Questions

Eos 4 and Eos 0 seem to have two distinct 'modes', a 'slow mode' and a 'fast mode'. The 'fast mode' of Eos 4 seems to be about twice as fast as its 'slow mode'. The 'fast mode' of Eos 0 is about 10 times faster than its 'slow mode'. This odd behaviour is most likely caused by a malfunction in the machines' turbo button. Eos 4 in its 'fast mode' beat all the other machines in all cases.

To test the times of Send, Receive, Reply, we read the PIT directly in a user task before and after a Send() system call. If the PIT wrapped around, then there could be many context switches in addition to the time for Send, Receive, Reply due to the clock event (the clock notifier is woken up, sends a message to the clock server, etc). Thus, we disregard such values.

For a 16 byte message over 30000 trials, we have the minimum time of approximately 0.3700 ms. For a 0 byte message over 30000 trials, we have the minimum time of approximately 0.3929 ms. For a 256 byte message over 30000 trials, we have the minimum time of approximately 0.357 ms.

Over 250 trials, an average create request on Eos 4 in its 'fast mode' takes approximately 9.121 ms, including the time spent in crt0 during creation. In order to allow a process to create a default message queue and register it with the kernel (in the PCB) for the Send/Receive/Reply semantics in the kernel specification (on the web page), a process must be allowed to complete its initialization before another process runs. It performs this initialization with interrupts on, with a temporarily high priority. Create is the longest path through the kernel, and thus the maximum bound on interrupts being off is the length of time spent in Create in the kernel. Thus, this time (9.121 ms) is also very close to the maximum time spent with interrupts off. Unfortunately, due to our implementation, finding the exact amount of time with interrupts off would require writing most of the timing in assembly.