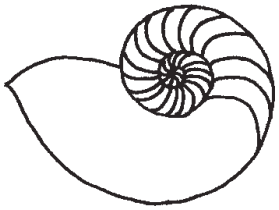
Volume 4, Number 8 June 1992 $2.50



MUUG Lines

Manitoba UNIX®User Group

Newsletter of the Manitoba UNIX® User Group

MUUG Goes Online!

By Roland Schneider

For any of our members who don't know about the *MUUG Online* project yet, a little background is in order. The University of Manitoba Computer Centre has given MUUG access to a Sun386i workstation which they no longer require. The machine is located at the university, and is connected to the university's LAN and to the Internet. Dialup access to the machine, named "mona" (for "**M**UUG **O**nline **N**etwork **A**ccess") is through the modem pool on the university's UMnet (Develnet). MUUG is responsible for the operation and maintenance of the machine.

The purpose of the *MUUG Online* project is to give our members access to Internet services like e-mail, ftp, and Usenet news. Any MUUG member can apply for an account on MONA to get interactive access to the Internet. We will also be providing dialup UUCP links to members who want to receive e-mail and news on their own systems instead having to manually call MONA.

Services

E-mail

Electronic mail is a very effective means of communication. It allows anyone on a networked computer, from PCs to mainframes, to send, receive, archive, forward, and distribute messages, or computer data of any kind. Many organizations have been using e-mail internally for some years. With *MUUG Online*, you will be able to either log in to MONA interactively to send mail anywhere in the world, or establish a UUCP link and integrate your present office system with the world-wide access offered by the Internet.

Everyone with an account on MONA has an e-mail address starting with their userid, like `jsmith@muug.mb.ca`, as well as one containing their full name, like `john.smith@muug.mb.ca`. Those with a UUCP connection will also have an address containing the name of their own computer, like `john@JJConsult.muug.mb.ca`. J&J Consulting may have users other than John, so there may also be an address like `joe@JJConsult.muug.mb.ca`, even though Joe doesn't have an interactive MONA account.

Usenet News

Usenet news is like a huge, world-wide, non-interactive bulletin board. Anyone with access can read and post items, and read everything everyone else has posted. The news is divided into newsgroups, each with a different topic. Topics range from politics, to humour, to computers (lots of computers...), to specialized scientific research areas.

MUUG Online intends to carry most of the newsgroups our members are interested in, taking into account disk space and communications limitations. We have also set up some local newsgroups to allow the same sort of interchange among MUUG members. News will be available interactively on MONA and via UUCP.

(Continued on page 13)

THIS MONTH'S MEETING

Meeting Location:

Our June meeting is scheduled for Tuesday, June 2, at 6:00 PM (a week and 1.5 hour earlier than usual). This meeting will be the traditional TUUG June BBQ. This year, Roland Schneider is hosting it at his home in Selkirk. A map is included in this month's newsletter.

Meeting Agenda:

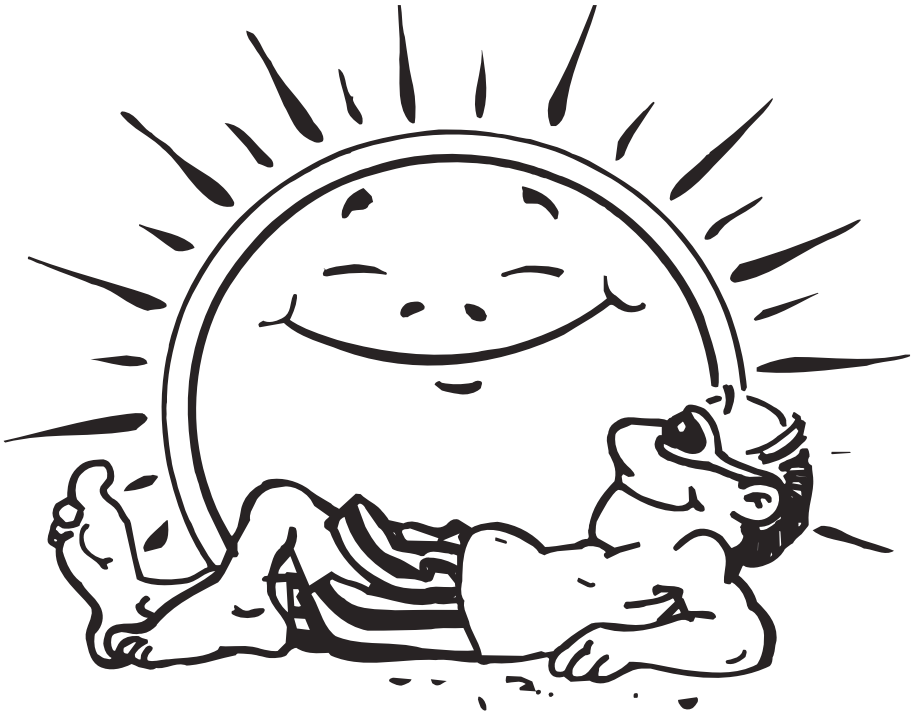
Eat, drink, and be merry – but no computer talk!

INSIDE THIS ISSUE

President's Corner
Interim Financial Statements
Hands-on: Using the XView Open Look Toolkit; Shared Memory for Inter-Process Communication, pt. 2
The Fortune File
May 12th Meeting Minutes
Map to June 2nd BBQ

UNIX is a registered trademark of UNIX System Laboratories, Inc.

Have a great Summer!



The 1991-1992 Executive

President:	Susan Zuk	(W) 788-7312
Past President:	Eric Carsted	1-883-2570
Vice-President:	Richard Kwiatkowski	589-4857
Treasurer:	Rick Horocholyn	(W) 474-4533
Secretary:	Roland Schneider	1-482-5173
Membership Sec.:	Allan Moulding	269-8054
Mailing List:	Gilles Detillieux	489-7016
Meeting Coordinator:	Kathy Norman	(W) 474-8311
Newsletter editor:	Gilbert Detillieux	489-7016
Information:	Susan Zuk	(W) 788-7312
		(FAX) 788-7450
(or)	Gilbert Detillieux	(H) 489-7016
		(FAX) 269-9178

Copyright Policy and Disclaimer

This newsletter is ©opyrighted by the Manitoba UNIX User Group. Articles may be reprinted without permission, for non-profit use, as long as the article is reprinted in its entirety and both the original author and the Manitoba UNIX User Group are given credit.

The Manitoba UNIX User Group, the editor, and contributors of this newsletter do not assume any liability for any damages that may occur as a result of information published in this newsletter.

Our Address

Manitoba UNIX User Group
P.O. Box 130
Saint-Boniface, Manitoba
R2H 3B4

Internet E-mail:
editor@muug.mb.ca

Group Information

The Manitoba UNIX User Group meets at 7:30 PM the second Tuesday of every month, except July and August. The newsletter is mailed to all paid up members one week prior to the meeting. Membership dues are $20 annually and are due at the October meeting. Membership dues are accepted by mail and dues for new members will be pro-rated accordingly.

PRESIDENT'S CORNER

As the Summer Approaches

By Susan Zuk, President

Much has happened in the last month in terms of group activity. The name Manitoba UNIX User Group is officially ours, we have officially become an affiliate of UniForum Canada, MUUG was visited by the President of UniForum Canada, Tom Vassos, we participated in the Muddy Waters Computer Society Computer Show, and the process of setting up internet access for our members is moving ahead quickly. So you can see this is a very exciting and busy time for the group.

The exposure we received at the Computer Show, on April 26th at the Convention Centre was excellent. Just under 4,000 people passed through the doors and we had over 400 stop and talk to us about our group and our activities. This was MUUG's first publicity event and was very worthwhile. Thanks to Gilbert Detillieux, Gilles Detillieux, Allan Moulding, Roland Schneider, and Rick Horocholyn for spending one of their precious Sundays helping to promote the group. Your dedication is wonderful.

The affiliate form was completed and sent back to the UniForum office with Tom Vassos this previous week. The first event where our group is represented is at UniForum's Open Systems Show held May 27-29. UniForum is holding one of its semi-annual meetings on Friday. This is a full day session called a National Council Meeting. The National Council is comprised of the President's of the local affiliates. I will report on the day's events in the next newsletter.

Our meeting with Tom Vassos, the President of UniForum Canada, was very enlightening. Tom spoke to the executive about various programs being pursued by the National Committee. UniForum Canada has changed its by-line to UniForum Canada, the Canadian Association of Open Systems Professionals. The feeling is that this will allow the organization to offer a fuller range of information to its members since UNIX is not the only thing which makes an environment open. Such items as standards, other operating systems and hardware can also be addressed.

Tom informed us that a Western Road Show is being planned for Edmonton, Calgary and Vancouver in the fall timeframe. This is something we could consider becoming involved with in the upcoming years. UniForum Canada is also working on developing a package price for UNIX publications for its members. They are busy talking to the publishers of the various publications. Another program is to setup a Canadian User Alliance, so that Canadian companies and users may be able to have a voice and some involvement in what is going on in the Opens Systems environment. An alliance has already been organized in the U.S. As well, industry sectors such as the Petroleum companies have organized their own groups to lobby industry to comply to their specifications. There were more items which were discussed but I will wait to receive more information and report to you in the next newsletter.

Just a reminder about our Annual Barbecue on June 9th. Come down and join us. You will find more details later in the newsletter. If you are unable to attend the Barbecue, I would like to wish you a wonderful summer and look forward to seeing you again in the fall.

FINANCE

Manitoba UNIX User Group

Interim Financial Statements

By Rick Horocholyn, Treasurer

Balance Sheet
As of 1992/05/31

Assets	
Cash	80.00
Chequing	1454.28
Investments	8000.00
Accounts Receivable	30.00
Equipment	168.68
Other Assets	54.72
Total Assets	**9787.68**
Liabilities	
Accounts Payable	662.34
Total Liabilities	662.34
Equity	
Net income to date	8614.01
Retained Earnings (previous year)	511.33
Total Equity	9125.34
Total Liabilities and Equity	**9787.68**

Profit and Loss Statement
For the 7 months ending 1992/05/31

Revenues	
Membership Fees	1526.00
Other Revenue (Fall Symposium)	8000.00
Total Revenues	9526.00
Expenses	
Newsletter (paper & postage)	492.15
Advertising	63.00
Entertainment (Symposium wind-up)	59.55
Speaker Fees & Expenses	82.08
Bank Charges	4.83
Postal Box Rental	80.25
Miscellaneous Expenses	130.13
Total Expenses	911.99
Net Income	**8614.01**

Rick Horocholyn *works for Manitoba Hydro. He's been a member of the group for several years, and has been the group's treasurer since October, 1991.*

Using the XView OPEN LOOK Toolkit

An easy way to build a modern Graphical User Interface

By Roland Schneider

Most modern applications demand easy-to-use and easy-to-learn graphical user interfaces. (GUI's) Easy-to-use does not imply easy-to-program, of course. Toolkits, like Sun's XView, make the task more manageable, and encourage adherence to GUI standards, which in turn makes the application easier to learn because many aspects of the interface behavior will already be familiar to the user.

It is important to get a few terms straight before diving into the technical details. The underlying graphics standard is "X Windows", or "X11" or just "X". X, through the functions in the library Xlib, lets you create windows, draw lines and polygons, and monitor keyboard and mouse activity on local or networked X display servers. (display devices like workstations or X terminals) A GUI standard, like OPEN LOOK or Motif, specifies how the user interacts with a program. The functions of the mouse buttons, the appearance and operation of scrollbars, control buttons, menus, and pop-up windows are all part of the standard. A toolkit, like XView, is a set of functions which help in implementing a GUI conforming to a standard, in this case OPEN LOOK. The term "application program interface" or "API", refers to this set of functions and the arguments you pass to them.

The Details

There are very few functions to call in XView, but most take a `NULL`-terminated, variable-length list of arguments. The functions are easy to remember, the arguments are not. The nice thing is that most parameters have default values, so you can usually get away with only a few arguments. Functions which create something return a *handle* to the created object, which can later be used to control and query the object.

The first step is to initialize XView and create a base window into which everything else will go.

```
xv_init(NULL);
base_frame = (Frame) xv_create(NULL,
  FRAME,
  FRAME_LABEL, "My Window",
  NULL);
```

The arguments to `xv_create()` specify that the parent is the root window (the background), that we want to create a frame, and that its label should be "My Window." We could also have specified an icon to use when the window is closed, an initial size, whether or not the frame can be resized with the mouse, footers in addition to the header label, etc. We didn't, so XView assigns default values.

Now we have to create a panel in the frame onto which we'll put our buttons and so on.

```
control_panel = (Panel) xv_create(base_frame,
  PANEL,
  PANEL_LAYOUT, PANEL_VERTICAL,
  NULL);
```

The parent of the panel is the base frame and the layout of the items on the panel will be vertical. Now let's put a simple button on the panel.

```
xv_create(control_panel,
  PANEL_BUTTON,
  PANEL_LABEL_STRING, "Quit",
  PANEL_NOTIFY_PROC, proc_do_quit,
  NULL);
```

The parent is the control panel we created before, the thing being created is a button, the label is "Quit", and the function to call when the button is pushed is specified by the function pointer `proc_do_quit`.

Inverted Program Structure

"Wait a minute, that isn't how I handle user input. I read a command, parse it, and then act on it — what's this stuff about XView calling my function?" Well, that's how XView wants you to do it. It actually works quite well, since it encourages you to design your program so that it reacts to user input, or is "event driven." Unfortunately, converting an existing program to this structure can be very painful, depending on how it's currently written. There are ways to use XView with a traditional command-reading loop, but it's difficult, and many things don't happen as automatically as they do when you use the default structure.

Let's finish our program. We'll add a slider, and a non-exclusive choice item.

```
bcontrol = (Panel_item)xv_create(control_panel,
  PANEL_SLIDER,
  PANEL_LABEL_STRING, "Bright:",
  PANEL_MIN_VALUE, 0,
  PANEL_MAX_VALUE, 100,
  PANEL_SLIDER_WIDTH, 50,
  PANEL_TICKS, 5, PANEL_SHOW_VALUE, FALSE,
  PANEL_VALUE, 70,
  PANEL_NOTIFY_PROC, proc_set_brightness,
  NULL);


xv_create(control_panel, PANEL_TOGGLE,
  PANEL_LABEL_STRING, "What:",
  PANEL_CHOICE_STRINGS,
        "Inside",
        "Outside",
        NULL,
  PANEL_VALUE, 1,
  PANEL_NOTIFY_PROC, proc_set_what,
  NULL);
```

Notice that the label is always set by `PANEL_LABEL_STRING`, and the notification function by `PANEL_NOTIFY_PROC`, no matter what kind of panel item is being created. The slider has lots of parameters because the

defaults didn't suit our purpose. PANEL_CHOICE_STRINGS is followed by its own NULL-terminated list of labels for the choices. You can specify pictures instead of text for labels by using PANEL_LABEL_IMAGE instead of PANEL_LABEL_STRING. Similarly, PANEL_CHOICE_STRINGS can be replaced by PANEL_CHOICE_IMAGES. The PANEL_VALUE parameter specifies an initial value for the choice item.

The sizes of the frame and the panel have never been specified. We really just want them big enough to contain whatever is inside, so we call

```
window_fit(control_panel);/* buttons, etc. */
window_fit(base_frame);   /* contains panel */
```

Drawing Graphics

Graphics are drawn into a *canvas,* so we create one and get the corresponding X window for drawing into using Xlib calls, which won't be discussed here.

```
my_canvas = (Canvas) xv_create(base_frame,
  CANVAS,
  XV_WIDTH, 200,
  XV_HEIGHT, 100,
  NULL);

my_xwin=xv_get(canvas_paint_window(my_canvas),
        XV_XID);
```

It's also possible to set an input handling function for the canvas which will be called with the mouse coordinates, mouse buttons, keyboard keys, etc. whenever an event happens while the cursor is over the canvas.

Finishing It Off

We added the canvas to the frame, so the frame has to be resized using

```
window_fit(base_frame);   /* panel & canvas */
```

This will also stretch the panel so that it fills the entire width of the frame, which is now wider because of the canvas.

Now, finally, we give control to XView so that it can handle the input events from the mouse and keyboard, provide feedback like inverting the image of a panel button when the left mouse button is pressed, and call our notify procedure when the left mouse button is released.

```
window_main_loop(base_frame);
```

It's possible to avoid most of this programming by using software which lets you use the mouse to set panel item properties and then place them. Sun has such a product, but I've never used it, so I can't say much about it. Although this user-interface code tends to be long-winded, using it amounts to more cutting and pasting than anything else, and the compiled code isn't very large.

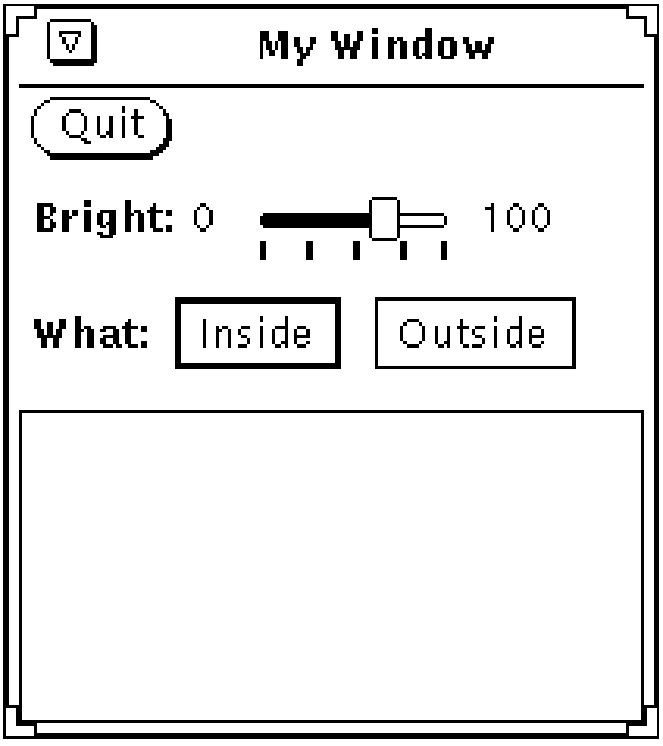
Notify Procedures

The processing in the program is performed by the notify procedures. Their functions can range from very simple, like the one for the quit button, to complicated and time-consuming. The parameters they are called with depend on the type of panel item being processed. Here are procedures for the panel items we created:

```
int do_inside = 1;
int do_outside = 0;
void proc_do_quit(button, event)
      Panel_item button;
      Event *event;
{
      printf("Quit button was pressed\n");
      exit(0);
}

void proc_set_brightness(item, value, event)
      Panel_item item;
      int value;
      Event *event;
{
      printf("Slider set to %d\n", value);
      /* do whatever control is supposed to do */
}

void proc_set_what(item, value, event)
      Panel_item item;
      int value;
      Event *event;
{     /* each bit reps one of the choices */
      if (value & 1) do_inside = 1;
            else do_inside = 0;
      if (value & 2) do_outside = 1;
            else do_outside = 0;
      if (value == 0)   /* nothing set */
            xv_set(bcontrol,
                  PANEL_INACTIVE, TRUE,
                  NULL);
      else xv_set(bcontrol,
                  PANEL_INACTIVE, FALSE,
                  NULL);
}
```



Notify procedures are always called with a handle to the panel item which generated the event. In this example, the handle was never needed because each notify procedure handles only a single panel item, something which is impractical in a large program. Notice that `proc_set_what()` uses `xv_set()` to deactivate and gray-out the brightness control if nothing is selected. In fact, `xv_set()` can change most of the parameters which can be specified to `xv_create()`, including the item labels, choice strings, values, etc.

Other Panel Items

This is only a small sample of the panel items available. There are may kinds of choice items, including abbreviated choices which automatically generate a pull-down menu. There are also text items for typing into, buttons with menus attached, message and gauge items for showing display-only information, and scrolling lists.

Other Objects

There are other kinds of XView objects too. Menus can be created and attached to panel buttons so that they will be shown when the right mouse button is pressed over the button. Alternatively, you can display a menu whenever you want, for example from a canvas input handler in response to a right mouse button press over the canvas.

One of the unique features of an OPEN LOOK menu is that it can be *pinned* by the user, making it into a small window which doesn't go away when its functions are executed. This has implications for the design of the user interface, because it allows obscure functions which may be heavily used from time to time to be buried deep in the menu structure. The user only has to "walk" through the menus once, and pin the one containing the required commands.

There are also text subwindows, which allow simple text editing, tty subwindows for running programs which want terminal I/O, scrollbars which allow the user to pan and split canvases and notices for displaying warnings and error messages. XView also provides access to selection services for cutting and pasting between applications as well as control of fonts, colormaps, cursors, and most other aspects of a modern interface.

Special Features

A serious drawback of the inverted XView programming style is that you can end up with a large number of global variables, like the handle `bcontrol`, and the status variables `do_inside` and `do_outside`, and a lot of very simple notify procedures. XView contains two useful, although somewhat subtle, features which help avoid this.

The first is a pair of functions which can be used to "walk through" all the items on a panel, avoiding the need to use a global variable for each panel item if the requirement is simply to set or read the current values. No notify procedure is needed in this case, because the values can be read with the `xv_get()` function.

The second feature is the ability to attach arbitrary data, usually in the form or pointers or integers, to any XView object, including panel and menu items. The call

```
xv_set(item,
  XV_KEY_DATA, 123, "This is my item",
  NULL);
```

attaches the key 123 and a pointer to the string "This is my item" to the item. Later, the call

```
txt = (char *)xv_get(item, XV_KEY_DATA, 123);
```

will retrieve the text. Any number of keyed data items can be attached to each XView object. This can be a useful way to communicate information from one part of a program to another and allows you to write "generic" notify procedures which can handle events from a variety of related panel items, because the items can be labeled with character string identifiers or handles to the objects they are supposed to control.

Compatibility

Theoretically, XView programs will work with any X server, but special fonts, which are now included in most systems, are required for the buttons, menus, etc. to look right. Also, some features may not be available if you don't use an OPEN LOOK compliant window manager. The window manager is the program which allows you to open, close, resize, and move windows around on the screen, and also provides the "pinning" function described earlier.

How to Get It

If you have a Sun workstation running OpenWindows, you already have XView. If you don't, source is available in both the X11R4 and X11R5 distributions from MIT. I believe the R4 version of XView had only been ported to Sun and DEC 3100 workstations; I don't know about the R5 version. X11R5 is available from MIT for the cost of distribution, and will be available for free to MUUG members when we put together our public domain software tape.

To Read Further

The standard books on X windows are published by O'Reilly and Associates, Sebastopol CA. Of particular interest are:

Volumes 1 and 2:
Xlib Programming Manual
Xlib Reference Manual
ISBN 0-937175-13-7 (set of both books)

Volume 7:
XView Programming Manual
ISBN 0-937175-52-8

Prices are about $37 per book. All three are included in Sun's on-line AnswerBook documentation.

Roland Schneider *is a Ph.D. student in Electrical Engineering at the University of Manitoba. He is also the MUUG secretary since October, 1991.*

Using Shared Memory for Inter-Process Communication

Part 2 of 3

By Peter Graham

Last time we talked about the potential uses for shared memory both in uniprocessor and multiprocessor Unix machines. (Admittedly with a leaning towards multiprocessors. :-) This article discusses the "overhead" associated with shared memory applications; namely "synchronization."

Whenever multiple concurrent (or pseudo-concurrent) processes share data there is the possibility of contention. This simply means that more than one process may be attempting to access the data at the same time. If both of these processes read the data, then everything is alright, but if one or more attempt to write it then problems occur. Synchronization is the solution to these contention problems. By coding *semaphore* calls (or perhaps using other synchronization primitives) the programmer "synchronizes" the access of the processes to the shared data. Effectively enforcing an access order on them.

Why do we really need synchronization? This is most easily answered with a simple example of what can happen if you do not provide *synchronization*. Consider a system of automatic teller machines (ATMs). Suppose that a certain Computer Science PhD student and his wife both have ATM cards which allow access to their account. If these malicious individuals both attempt to withdraw $500 from their account at the same time (presumably using two side-by-side machines) then the bank's computer might receive the following two transactions simultaneously.

Transaction 1	Transaction 2
read balance from acct	read balance from acct
if balance < $500 then	if balance < $500 then
reject transaction	reject transaction
else	else
balance=balance-500	balance=balance-500
endif	endif
write balance to acct	write balance to acct
dispense $500	dispense $500

If we further suppose that these two identical transactions are being performed on a multiprocessor and are executed concurrently then what will happen? If the original balance was $501 (graduate students are poor :^>) then both transactions running on different processors will read the balance and find it to be $501. Since this is greater than $500, both will dispense $500 after subtracting 500 from the balance and writing the new ($1) balance back. The ending balance is correct, but $1000 has been dispensed. This is because the transactions (i.e. processes) were not synchronized. Clearly, a bank manager would not be pleased with this occurrence.

Now, before all you married folk out there go rushing out to the ATMs at your local mall, I can pretty much guarantee that the banks have closed this little loophole. This is the most common form of synchronization problem there is and it is well understood by pretty much anyone who has taken an Operating Systems or Database class.

The important thing to understand is why the problem occurs. The code reads, modifies, and then writes the account balance. The problem arises because once one transaction has read the value, the other should not be able to read it until the first has finished writing the new balance. If the second transaction can read the same balance, they will both come to the conclusion that there is $500 available in the account and will both dispense $500. Such a piece of code is called a *critical section* and access to critical sections (CSs) must be synchronized. Stated in more general terms, operations on shared data must be done *indivisably* (one at a time to completion). By synchronizing the transactions we force one to go before the other and say that they are executed *mutually exclusively*.

Now that we have that out of the way, and assuming you still want to use shared memory due to its high-efficiency, we are left with the task of learning to do synchronization using Unix. Unix provides a very flexible set of semaphore primitives for mutual exclusion (mutex). While semaphores are considered to be a low-level mutex facility, they are very general and are programming language independent. This was likely why they were chosen for use in Unix.

A semaphore is a special type of variable which is used for synchronization. Two operations are normally defined on semaphores; "wait" and "signal." The semaphore maintains an integer value which is typically either zero or one. The "wait" operation stops a given process from executing until the value of the semaphore is non-zero. It then decrements the value and returns. If a semaphore's value is zero, the "signal" operation restarts one of the processes waiting on that semaphore. Otherwise, "signal" just increments the value of the semaphore. The key here is that these are operations done in the kernel which makes them indivisible. That is, the kernel ensures that only one process can access a semaphore's value at a time. If this were not the case then we would have critical sections for accessing and updating the semaphore's value.

We can use these simple ("binary") semaphores to solve our previous synchronization problem. If we refer to the entire transaction (our CS) as 'Ti' then we simply code the following:

Transaction 1	Transaction 2
wait(mutex_sema)	wait(mutex_sema)
Ti	Ti
signal(mutex_sema)	signal(mutex_sema)

Assuming that the semaphore ('mutex_sema') is initialized to one then this will force the two transactions to be synchronized. Even if both transactions execute their 'wait' operations simultaneously, the kernel will ensure that only one is performed at a time. Thus, one transaction will get to enter its critical section while the other will be stopped. When the

transaction which first enters the CS issues its 'signal' the other process will be awakened and only then will it be allowed to enter its CS.

It is also possible to have "counting" semaphores which allow the semaphore to take on other positive values besides one and zero. The descriptions of wait and signal given above still hold in this case. We will postpone discussing their use for the time being but merely acknowledge their existence and assume their usefullness.

Unix extends the notion of the basic binary semaphore described previously by allowing counting semaphores and by also allowing semaphores to be grouped into "semaphore sets." Unix's semaphore operations operate on semaphore sets not individual semaphores (of course a set may consist of a single semaphore). The advantage of having semaphore sets is that by grouping logically related semaphores, we can perform operations on *all* members of the set at once while still being ensured of mutual exclusion.

The Unix semaphore calls consist of 'semget' to create a semaphore set, 'semop' to perform operations on a semaphore set, and 'semctl' to perform a variety of functions including destroying unused semaphores. We will discuss each in turn briefly as we did in part one of this article leaving out the details of the more esoteric features each provides.

If we require a single semaphore, the first thing we must do is create it. This can be done with the call:

```
semid=semget(key, 1, IPC_CREAT);
```

A call to 'semget', like a call to 'shmget' requires a unique key (formed using 'ftok') to identify the semaphore set. The second argument specifies the number of semaphores required (in the set) and the final argument specifies that we want to create this semaphore set if it does not already exist. As you might expect, 'semget' returns a semaphore identifier for use with the 'semop' operations or -1 if an error occurs.

A 'semop' call has the following form:

```
semop(int semid, struct sembuf **opsptr,
        unsigned int numops);
```

The first argument identifies the semaphore set being operated on and the second argument specifies a pointer to an array of semaphore operations. The third argument specifies the size of that array (i.e. the number of operations to be done.) The type 'struct sembuf' is defined as:

```
struct sembuf {
  ushort sem_num;   /* sem. # in the set */
  short sem_op;     /* op. to be performed */
  short sem_flg;    /* operation flags */
};
```

The 'sem_op' member specifies the operation to be done on the given semaphore in the set. It is used as follows:

1. If sem_op is positive, its value is added to the semaphore.
2. If sem_op is zero, the caller is made to wait until the semaphore's value becomes zero.
3. If sem_op is negative, the caller is made to wait until the semaphore's value becomes greater than or equal to the absolute value of sem_op.

There are a number of different possibilities for the 'sem-flg' member. The most useful of these permit non-blocking waits, and a roll-back facility which will correct for any outstanding semaphore operations when a process terminates abnormally. The 'semop' call returns either 0 or -1 indicating success or failure.

The 'semctl' call can be used for many purposes including explicit setting or reading of a semaphore's value. We will only consider its use in removing a semaphore which is no longer required. This can be done using the following call assuming our single semaphore example:

```
semctl(semid,0,IPC_RMID,0);
```

See the online man pages for more details about these primitive semaphore operations.

Now that we have the necessary primitives, let's look at some code which uses both shared memory and semaphores. We will implement a much simplified version of a bank's ATM system. We create a banker (server) process which initializes a set of accounts in shared memory which are then available to a collection of ATM processes. Admittedly, this is not the way an ATM system would actually be implemented but it serves its purpose for illustrating simple semaphore usage. The code in 'bank.c' creates the segment and initializes all ten accounts so they have $501.00 each. The code in 'forker.c' creates ten processes running the code in 'atm.c'. All that each ATM process does is perform the transaction described at the beginning of this article. Even numbered ATM processes operate on the account numbered with their "process number" while odd numbered processes operate on the same account as the preceding even numbered process. Thus, odd numbered accounts will not be touched while even ones will have to $500 debit transactions performed against them.

This is the reorganized output from a run without synchronization. (That is, I removed all the semaphore code.) Notice that $1000 dollars was incorrectly dispensed from all even numbered accounts.

```
account#0:initial balance is 501.000000.
account#1:initial balance is 501.000000.
account#2:initial balance is 501.000000.
account#3:initial balance is 501.000000.
account#4:initial balance is 501.000000.
account#5:initial balance is 501.000000.
account#6:initial balance is 501.000000.
account#7:initial balance is 501.000000.
account#8:initial balance is 501.000000.
account#9:initial balance is 501.000000.
acct #0:$500.00 dispensed. Please bank here again!
acct #0:$500.00 dispensed. Please bank here again!
acct #2:$500.00 dispensed. Please bank here again!
acct #2:$500.00 dispensed. Please bank here again!
acct #4:$500.00 dispensed. Please bank here again!
acct #4:$500.00 dispensed. Please bank here again!
acct #6:$500.00 dispensed. Please bank here again!
acct #6:$500.00 dispensed. Please bank here again!
acct #8:$500.00 dispensed. Please bank here again!
acct #8:$500.00 dispensed. Please bank here again!
account#0:final balance is 1.000000.
account#1:final balance is 501.000000.
```

```
account#2:final balance is 1.000000.
account#3:final balance is 501.000000.
account#4:final balance is 1.000000.
account#5:final balance is 501.000000.
account#6:final balance is 1.000000.
account#7:final balance is 501.000000.
account#8:final balance is 1.000000.
account#9:final balance is 501.000000.
```

This is the reorganized output from a run with synchronization. Notice that the system now performs correctly since all transactions are synchronized.

```
account#0:initial balance is 501.000000.
account#1:initial balance is 501.000000.
account#2:initial balance is 501.000000.
account#3:initial balance is 501.000000.
account#4:initial balance is 501.000000.
account#5:initial balance is 501.000000.
account#6:initial balance is 501.000000.
account#7:initial balance is 501.000000.
account#8:initial balance is 501.000000.
account#9:initial balance is 501.000000.
acct #0:$500.00 dispensed. Please bank here again!
acct #0: Sorry not enough money.
acct #2:$500.00 dispensed. Please bank here again!
acct #2: Sorry not enough money.
acct #4:$500.00 dispensed. Please bank here again!
acct #4: Sorry not enough money.
acct #6:$500.00 dispensed. Please bank here again!
acct #6: Sorry not enough money.
acct #8:$500.00 dispensed. Please bank here again!
acct #8: Sorry not enough money.
account#0:final balance is 1.000000.
account#1:final balance is 501.000000.
account#2:final balance is 1.000000.
account#3:final balance is 501.000000.
account#4:final balance is 1.000000.
account#5:final balance is 501.000000.
account#6:final balance is 1.000000.
account#7:final balance is 501.000000.
account#8:final balance is 1.000000.
account#9:final balance is 501.000000.
```

Notice that the code does not remove the semaphore at any point. This is because we cannot guarantee an ordering on the execution of the ATM processes so none of them can destroy it for fear of doing so before all the other processes were finished. The semaphore could have been created by the banker process and explicitly removed after the 60 seconds. In the interest of brevity I simply omitted this code.

Next time we will look at the development of a print spooler application which uses shared memory and semaphores, and the resulting code.

```
/**********/
/* acct.h */
/**********/
/* balances for 10 accounts - small bank */
typedef struct acct {
      float balances[10];
} ACCT, *ACCTPTR;

/************/
/* forker.c */
/************/

main()
{
      int   i,          /* simple counter       */
            pid;        /* forked process' pid  */
      char  argstr[16]; /* converted pid string */

      /* this code simply creates ten identical */
      /* processes to run the program 'atm.c'   */
      /* which simulates the actions of an      */
      /* Automated Teller Machine (ATM).        */
      for (i=0;i<10;i++) {
            pid=fork(); /* fork a new process   */
            if (pid==0) {
                  /* we are child process so... */
                  /* execute the atm program,   */
                  /* passing it a process num   */
                  sprintf(argstr,"%d",i);
                  execl("/home/cs/staff/pgraham/
misc/tuug/shm_article/part2/atm",
                        "atm",argstr,(char *)0);
            }
      }
} /* end main */
```

```
/**********/
/* bank.c */
/**********/

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "acct.h"

/* necessary because of problem  */
/* with SunOS include file shm.h */
#define SHM_W 0200      /* shm write permission */
#define SHM_R 0400      /* shm read permission  */

main()
{
      int         i;          /* simple counter */
      key_t       acct_shmkey; /* key to shm seg*/
      int         acct_segid; /* shm segment ID */
      ACCTPTR     acct_segaddr;     /* seg ptr  */
      /* need for call to 'shmctl()':           */
      struct shmid_ds   *acct_shmbfptr;

      /* this code creates the shared memory    */
      /* segment and initializes it.            */

      /* creating unique key/name for acct seg  */
      if ((acct_shmkey=ftok("/home/cs/staff/
pgraham/misc/tuug/shm_article/part2/forker.c",
                                    'M'))==-1) {
            printf(
           "Couldn't create shared memory key.\n");
            exit(-1);
      }
```

```
      /* call shmget to create it               */
      if ((acct_segid=shmget(acct_shmkey,
                  sizeof(ACCT),
                  IPC_CREAT|SHM_R|SHM_W))==-1) {
            printf(
      "Couldn't get the shared memory segment.\n");
            exit(-1);
      }

      /* and map it into our address space at   */
      /* address returned in 'acct_segaddr'.    */
      if ((acct_segaddr=(ACCTPTR)
                  shmat(acct_segid,(char *)0,0))
                        ==(ACCTPTR) (-1)) {
            printf(
       "Couldn't attach shared memory segment.\n");
            exit(-1);
      }

      /* Put some balances in the accounts      */
      for (i=0;i<10;i++) {
            acct_segaddr->balances[i]=501.00;
      }
      printf(
      "Bank is open for business for 1 minute.\n");
      /* allow time to run the ATM processes    */
      sleep(60);

      /* detach the shared memory segment       */
      if (shmdt((char *) acct_segaddr)==-1) {
            printf(
       "Couldn't detach shared memory segment.\n");
      }

      /* now remove shared segment altogether   */
      if (shmctl(acct_segid,IPC_RMID,
                        acct_shmbfptr)==-1) {
            printf(
       "Couldn't remove shared memory segment.\n");
      }
}

/*********/
/* atm.c */
/*********/

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include "acct.h"

/* necessary because of problem  */
/* with SunOS include file shm.h */
#define SHM_W 0200      /* shm write permission */
#define SHM_R 0400      /* shm read permission  */

/* Semaphore wait operation */
s_wait(sema)
int   sema;
{
      static struct sembuf op_wait[2] = {
            0, 0, 0,    /* wait for zero semaph.*/
            0, 1, 0     /* then increment by 1  */
      };

      semop(sema, &op_wait[0], 2);
} /* s_wait */

/* Semaphore signal operation */
s_signal(sema)
int   sema;
{
      static struct sembuf op_signal[1] = {
            0, -1, IPC_NOWAIT,
            /* decrement semaphore by 1         */
            /* NOWAIT ensures no waiting        */
      };

      semop(sema, &op_signal[0], 1);
} /* s_signal */


transaction(acct,acct_segaddr)
int         acct; /* account number to work on  */
ACCTPTR     acct_segaddr; /* mapped segment ptr */
{
      float balance;    /* current balance      */

      /* read the balance */
      balance=acct_segaddr->balances[acct];

      /* sleep so as to force a context switch  */
      /* to simulate concurrent execution of    */
      /* the processes.                         */
      sleep(1);

      /* check the balance */
      if (balance < 500.00) {
            printf(
       "acct #%d: Sorry not enough money.\n",acct);
      } else {
            /* debit the balance */
            balance-=500.00;

            /* write the balance back */
            acct_segaddr->balances[acct]=balance;

            /* dispense the money */
            printf("acct #%d:%s\n",
      "$500.00 dispensed. Please bank here again!",
                  acct);
      }
} /* transaction */


main(argc,argv)
int   argc;
char  *argv[];
{
      int   i,          /* simple counter      */
            process_num; /* this process' num   */
      key_t acct_shmkey, /* shm segment key     */
            acct_semkey; /* shm semaphore key   */
```

```
      int   acct_segid,  /* shm segment ID      */
            acct_semid;  /* shm semaphore ID    */
      /* pointer to mapped shared segment:      */
      ACCTPTR     acct_segaddr;

      /* this code runs some sample             */
      /* transactions against the accounts      */

      /* create unique key/name for shared seg  */
      if ((acct_shmkey=ftok("/home/cs/staff/
pgraham/misc/tuug/shm_article/part2/forker.c",
                                    'M'))==-1) {
            printf(
           "Couldn't create shared memory key.\n");
            exit(-1);
      }

      /* call shmget to "open" it */
      if ((acct_segid=shmget(acct_shmkey,
                  sizeof(ACCT),SHM_R|SHM_W))
                                        ==-1) {
            printf(
      "Couldn't get the shared memory segment.\n");
            exit(-1);
      }

      /* map it into our address space at an    */
      /* address returned in 'acct_segaddr'.    */
      if ((acct_segaddr=(ACCTPTR)
                  shmat(acct_segid,(char *)0,0))
                              ==(ACCTPTR) (-1)) {
            printf(
       "Couldn't attach shared memory segment.\n");
            exit(-1);
      }

      /* create unique key/name for semaphore   */
      if ((acct_semkey=ftok("/home/cs/staff/
pgraham/misc/tuug/shm_article/part2/forker.c",
                                    'S'))==-1) {
            printf("Couldn't create semaphore.\n");
            exit(-1);
      }

      /* call semget to get the semaphore       */
      if ((acct_semid=semget(acct_semkey,1,
                  IPC_CREAT|SHM_R|SHM_W))==-1) {
            printf(
                  "Couldn't get the semaphore.\n");
            exit(-1);
      }

      /* extract our process num from argv[1]   */
      sscanf(argv[1],"%d",&process_num);

      /* print initial balances */
      printf(
             "account#%d:initial balance is %f.\n",
             process_num,
             acct_segaddr->balances[process_num]);

      /* If we are an even process, we reference*/
      /* acct number specified by our process #.*/
      /* If odd, we reference the previous acct.*/
      /* This way we ensure the possibility of  */
      /* synchronization problems.              */
      if ((process_num%2)==0) { /* even numbered*/
            s_wait(acct_semid);
            transaction(process_num,acct_segaddr);
            s_signal(acct_semid);
      } else {          /* odd numbered process */
            s_wait(acct_semid);
            transaction(process_num-1,
                        acct_segaddr);
            s_signal(acct_semid);
      }

      /* print final balances */
      printf("account#%d:final balance is %f.\n",
            process_num,
            acct_segaddr->balances[process_num]);

      /* detach the shared memory segment       */
      if (shmdt((char *) acct_segaddr)==-1) {
            printf(
       "Couldn't detach shared memory segment.\n");
      }
}
```


Misc. Riddles

Submitted by Adam Thompson

"VMS is a text-only adventure game. If you win you can use unix."

Q: Is there a UNIX FORTRAN optimizer?
A: Yeah, "rm *.f"

Q: How many Unix Support staff does it take to screw in a light bulb?
A: Read the man page!

Annual MUUG Barbecue

June 2, 6:00 pm

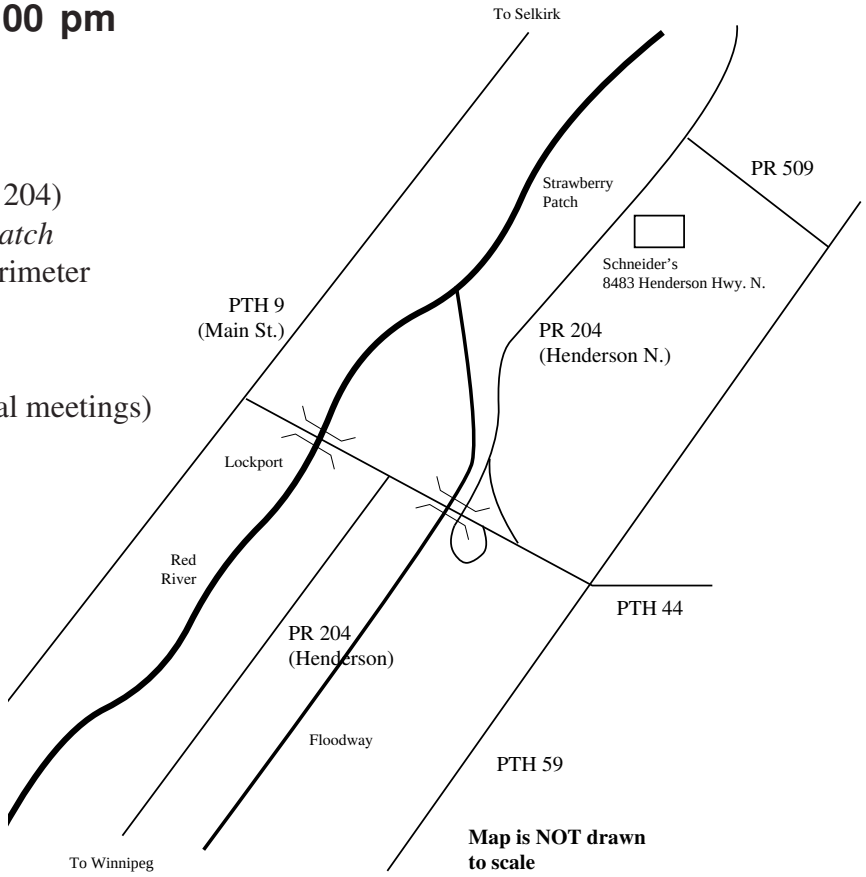
Host: Roland Schneider
phone: 1-482-5173

Where: The Schneiders'
8483 Henderson Hwy N. (PR 204)
Across from *The Strawberry Patch*
About 20 minutes from the Perimeter
(See map)

When: Tuesday, June 2, 6:00 pm
(1 week earlier than our normal meetings)

Bring: Meat to cook
Beer
Lawn chairs

RSVP: Roland Schneider
1-482-5173
(days and evenings)
e-mail: rsch@muug.mb.ca
or
Susan Zuk
788-7312 (days)



We will supply chips and other nibble food, soft drinks, salads, and a cake. (and insect repellent if necessary) Spouses or significant others, and children, are also welcome. Bring a swimsuit if you want to take a dip in our pond.

TUUG Meeting Minutes

Tuesday, May 12, 1992, 7:30 PM

234B Engineering Bldg., University of Manitoba, Ft. Garry Campus

Chair: Susan Zuk
Attendance: 47

Business meeting:

a) President's Report
- There was significant interest in MUUG's booth at the MWCS computer fest.
- The MUUG Online project is making progress.
- MUUG hopes for future joint projects with CIPS.
- The process of affiliation with UniForum is proceeeding.

b) Membership Report
- TUUG currently has 84 members

c) Treasuer's Report
- MUUG has $1800 in chequing account and $8000 invested.

d) New Business
- Moved by Gilbert Detillieux, seconded by Peter Graham, that the MUUG executive be authorized to spend up to $2600 to aquire a large SCSI disk for the MUUG Online system.
  - other, preferably free, alternatives will be explored before a disk is purchased.
  - passed unanimously.

Presented topic:
Future trends at Intel – Jon Coxworth, Intel Corp.

MUUG Goes Online!

Continued from page 1

Ftp

Ftp stands for 'File Transfer Protocol', and is used to copy ASCII and binary information across the Internet. Ftp is the usual method of obtaining PD software from other sites. It is usually used interactively to search through a restricted area on a remote site's disk and retrieve the desired data. Because ftp is an interactive program, this service will not be available through UUCP, although we may be able to set up an ftp-via-UUCP or ftp-via-e-mail service in the future.

UUCP

UUCP (**U**NIX to **U**NIX **C**o**P**y) is a system of programs which allows files, e-mail, news, and other information to be transmitted via dialup modem connections. When properly set up, all communications happen in the background, and the link is essentially transparent. *MUUG Online* will be providing UUCP links as soon as possible. To get a UUCP link, you will first need an account on MONA. Further information will be posted on MONA when we are ready to begin offering UUCP service.

PD Software

There is a lot of public domain software available for UNIX systems. Instead of having everyone retrieve the same software from the Internet repeatedly, we want to establish a repository of current versions of general-interest software on MONA. This software will then also be made available to MUUG members via tape and/or floppy disk.

Costs

There are some real costs involved in running the *MUUG Online* project. Although we don't pay anything for the Sun386i or it's network connections, we do have to fix or replace it if it breaks, and we have to obtain a large SCSI disk (1Gb) to provide spool space for news and storage for downloaded PD software. We are working on getting the disk for as little money as possible, perhaps even for free. Access to *MUUG Online* will be free for MUUG members until October, after which there will likely be a small ($30-$40 per year) charge for using the system.

Instructions

To access MONA interactively, you will need to get an account application form, fill it in, and mail it to MUUG or give it to a member of the executive. Account applications were included in last month's newsletter.

Once you have an account, use your modem to dial 275-6100 (300/1200/2400 bps) or 275-6132 (9600 bps, V.42bis, MNP5). Press RETURN once the connection is established and answer "`muug`" to the classname prompt. You will then be connected to MONA and asked for your userid and password. You will then be asked for your terminal type. Many communications packages emulate a DEC VT100. If you don't know what type of terminal you are emulating, "`vt100`" or "`ansi`" are good choices, with "`dumb`" as a last resort. (After you have logged on, look in the file `/etc/termcap` for a list of all the available terminal types, or type "`help termtype`")

Once you have logged in, you can type the command "`news`" to find out about the status of the MUUG Online project and other information of interest to MUUG members. Note that this is not related to the Usenet news described above. Use the command "`help`" to obtain advice about various procedures. Use "`man`" to access the online UNIX manuals.

Thanks

The entire *MUUG Online* project would not be possible without the generosity of the University of Manitoba Computer Centre and the assistance of the people there. We would especially like to thank Bill Reid, Kathy Norman, and Gary Mills. MUUG members continue to put a lot of effort into the project. Thanks are due to Andrew Chan for setting MONA up and getting all sorts of stuff working, to Gilles Detillieux for his work on the e-mail configuration, and to Gilbert Detillieux for setting up and coordinating many other aspects of the project.

To Find Out More

To find out more about *MUUG Online*, or to get a *MUUG Online* application form, please contact Roland Schneider at 1-482-5173 (days and evenings) or send e-mail to "`info@muug.mb.ca`".

Roland Schneider *is a Ph.D. student in Electrical Engineering at the University of Manitoba. He is also the MUUG secretary since October, 1991.*

Coming Up

Meeting:
Our next meeting is scheduled for Tuesday, September 8, at 7:30 PM (since we don't meet in July and August). Meeting topic and location will be given in September's newsletter. Meanwhile, enjoy the summer!

Online:
Watch for changes this summer to MUUG Online – more disk space, more software. Also, we'll try to keep a round table forum going on the local news groups.

Newsletter:
We will likely continue with RPC Programming by Scott Balneaves. We will also have part 3 on shared memory by Peter Graham, and several "filler" articles by Roland Schneider. Thanks again to all those who submitted those great articles throughout the year. Please keep the articles coming this summer – we'll need lots of material for the fall. Also keep in mind that nominations for the elections come up in September.