Thread Library to solve process synchronization problem

Dr. Nada A.Z. Abdullah

University of Baghdad

Abstract:

Most programs needed to be written in a way that are aware of the existence of each other and which can co-operate with each other towards some common goal. The standard solution to this problem is interprocess communication mechanisms (IPC). Any IPC mechanism is chosen, there is still the context switch overhead to contend with whenever normal processes need to work together. The way to overcome this is to use the concept of threads. The idea behind threads is that each process can have its processor time slices shared between several concurrent threads, each of which also shares the memory and data structures of the process to which it belongs.

In this paper a new and simple user level threads library is presented. This library can be easily used by the programmers to gain the benefits of threads. As an application a solution to the unbounded buffer problem is implemented using the developed thread library. C language under Linux Operating system is used for programming.

1. Introduction

In Linux the main unit of concurrency is the process. In order to switch between executing one process and the next quite a lot of work is involved. This context switch overhead is necessary because, in general, Linux needs to make sure that any pair of processes are protected from interfering with each other, especially when the processes may not even belong to the same user.

Processes can co-operate with each other towards some common goal. The standard solution to this problem is interprocess communication mechanisms (IPC). Using any IPC mechanism there is a context switch overhead to contend with whenever normal processes need to work together.

The way to overcome this is to use the concept of threads. *Threads*, like processes, are a mechanism to allow a program to do more than one thing at a time. As with processes, threads appear to run concurrently; the Linux kernel schedules them asynchronously, interrupting each thread from time to time to give others a chance to execute. Conceptually, a thread exists within a process. Threads are a finer-grained unit of execution than processes. When you invoke a program, Linux creates a new process and in that process creates a single thread, which runs the program sequentially. That thread can create additional threads; all these threads run the same program in the same process, but each thread may be executing a different part of the program at any given time.[1]

The amount of the effort involved in context switching between two co-operating threads within a single process is much smaller than the effort of switching between two processes. This is mainly because the threads within a process are all sharing the same text, data and system memory areas and structures so that no effort is expended in rearranging these things. Since the threads are specifically designed to be co-operative and not antagonistic, nothing needs to be done by the kernel to protect the threads within a single process from each other. Threads may, therefore, be considered as lightweight processes; light in the amount of the effort required to context switch between them when compared to the effort involved with context switching standard (heavyweight) process.

Threads can generally be implemented in one of two different ways:[2]

- Within the kernel, as a kernel task;
- Within the process itself, as a user space task.

When threads are implemented within the kernel, the scheduling of the threads can be preemptive, in a similar way to the scheduling of process themselves. This means that, whatever a thread is doing, when its time is up the kernel will interrupt it and pass control to the next thread to run. In the case of threads provided in user space, the implementation is very much simpler, though now the thread scheduling becomes non-preemptive. This means that a context switch between threads can only take place when the current thread voluntarily releases control of the processor itself.

Non-preemptive scheduling is usually avoided in multi-user environments because it allows one user to hog the processor by refusing to give up control to allow other users to take a turn. In the case of threads, however, non-preemptive scheduling does not present this problem as the threads are supposed to be co-operating on some task, not fighting over the processor. The only real problem that does occur with user level threads, which can be avoided by kernel level threads, is with blocking I/O. with a non-preemptive scheduler, if one of the threads was to block on I/O (waiting for keyboard input, for instance), then that thread would be unable to release its control of the processor to allow another thread to run. This would effectively bring all the threads within that particular process to a halt until the blocked I/O was complete and control could be transferred. This just means that care must be taking when using user level threads to ensure that blocking threads on I/O does not take place.

In this paper an implementation to a new, and simple, user level threads library that can be used in any program to solve process synchronization.

2. Library Calls

The threads library really is simple, especially in terms of the user interface. Threads are created dynamically at run time, as they are required. This is done with a call to the new thread () function:

int new-thread(int (*start-addr)(avoid) , int stack-size) where start-addr is a pointer to a function which will act, for the newly created thread, like the main () function acts for a process. The stack-size parameter specifies how much space, in bytes, should be set aside for this thread to use as its stack.

Whenever new-threads () is called, a new thread structure is created and initialized and added to a circular doubly linked list of threads structures. This list is used by the thread context switcher (scheduler) to determine which is the next thread to execute when the current thread gives up its control of the processor. The first time new-thread () is called, usually from within the main () function of a process, the execution of main () itself is suspended and the thread scheduler started up. Only when all the threads created within a particular process have terminated is control finally returned to the main () function, to the point just after the initial new-thread() call. On all calls to new-thread(), other than the first, a return is made immediately so that the calling thread can continue its execution. The return value from new- thread() is 0 on error (malloc() failure) or 1 on success. The only other function needed to use the threads library is the call which allows thread to give up its control of the processor back to the thread scheduler:

int release (void);

Calling this function will cause the thread scheduler to pass execution control to the next thread in the circular list of thread structures created by new-thread().

To be on the safe side, we should add a release () to our code anywhere where it might loop, so that even an infinite loop will not stop the other threads from being scheduled. Using this criterion, the places that need release () statements are: inside the body of each while, for and do loop, after each program label so that a goto can't cause a loop without a release().

In the situation where two or more threads need to communicate with each other this can be achieved by the use of variables which are global to the whole process (and therefore to each of its threads). Communication via global variables is common with threads, but suffers some- what from being

asynchronous. This means that without setting up some kind of special arrangement, a data producing thread which writes to a global variable has no way to know that the data has been read by a data consuming thread when it needs to replace the current contents of the variable with the next data value.

In order to make it easier to write threads whose communication is synchronized, the tiny threads library provides a simple communication mechanism between threads, based on the rendezvous concept. The basic idea behind a rendezvous is that it doesn't matter which of the two communicating partners (producer or consumer) arrives at the rendezvous point in its code first; its execution will be suspended until the other partner arrives at the same point. At this time, the data is passed between the two threads, the suspended thread's execution is resumed, and both threads then continue their execution with the communication guaranteed to have taken place[3]. The threads library provides three function calls related to the rendezvous mechanism: get-channel(), send() and receive(). The get channel() calls has the prototype:

```
int get-channel (int number):
```

where number is the communication channel number is used, and the return value is a channel descriptor value will be passed forward into the other rendezvous related calls (or zero on error).

This is similar in concept to opening a file except that we use a channel number instead of a file name, and get a channel descriptor. The way that channels implemented allows multiple producer and consumers all to share access to the same channel. In this case, the library will ensure that the producer's messages are properly queued and that each message can only be read by one consumer and will then be removed from the channel, in FIFO order. Having obtained a channel descriptor, messages can be communicated over the channel with send() and receive().

```
Int send( int  cd,   char  *buf ,  int  size);

Int receive( int  cd,  char  *buf ,  int  size);
```

where *cd* is a channel descriptor returned from a get_channel () call, *buf* is a pointer to a buffer from which data will be sent or to which data will be received, and *size* is the size of the data to be transferred, in bytes.

If matching send() and receive() calls specify different data size then the smaller of the two will be used. The return values from both the send() and receive() calls will be the actual number of bytes transferred.

3. Calling C Functions

Every C program starts by executing the function main(). In fact, when you compile a C program and link it for execution, a system specific piece of code is bolted onto the front of your program which deals with any command line parameters and performs a call to the program's main() function. Functions are called in the same way as subroutines are called in machine code, that is, a return address is pushed onto the machine stack and then control is transferred to the start of the called function. When this function terminates, the return address on the top of the stack is used to get back to the calling function at the instruction following the subroutine call[4].

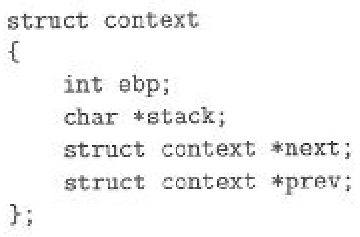
In fact, not only is the stack used to store function return addresses, but it is also used for passing function parameter values and for holding the values of automatic local variables declared within functions. This means that some care must be taken not to confuse any values stored on the stack, as any form of stack corruption can lead to some pretty sticky bugs.

When function is called, any parameter expressions, whose values you will pass into the function, are first evaluated and their values pushed onto the stack. Next, a machine code call is made to the function which also results in a return address being stacked. At the top of the called function, the names of any parameter variables specified are used to label the appropriate positions on the stack where the parameter values have already been pushed. In this way, any parameter values are automatically assigned to their respective parameter variables.

Once inside the function proper, the value of one of the processor registers (ebpin fact, which is the extended base pointer register) is also pushed onto the stack because that register is going to be used as a place holder to be able to find the return address on the stack when the function terminates. The value of ebp on entering the function, is pushed right next to the return address and then the current stack pointer value (pointing to the ebp value just pushed) is copied into the ebp register, overwriting its previous value, which is now safely stored on the stack.

4. Thread scheduling

All the threads within a single process share the program and all the global environment of the process. They only private parts of a thread are the contents of any significant processor registers and the stack, which it uses during its execution. This suggests that in order to switch contexts between two threads within a single process, all that is required is to swap the values in a few processor registers specifically, the program counter (or instruction register), ebp and the current stack pointer. Each thread has associated with it a data structure, which is used to save the per thread context when the thread is not currently running. The data structure has the following layout[3]:



```
struct context
{
    int ebp;
    char *stack;
    struct context *next;
    struct context *prev;
};
```

The next and prev fields of the structure are used to form the forward and backward linkes that tie this structure into the doubly linked, circular list of structures which form the scheduling loop. The stack field is a pointer to an array of bytes which is used as the private stack space for this thread, and the ebp field is used as a temporary store for the contents of the processor ebp register when the CPU is processing some other thread.

5.Context switching

The first piece of threads library code to look at is the release() function, which is called by a thread when it wants to pass its control of the CPU on to the next thread in the scheduling loop. When we examine this code, the variable current is a pointer to the struct context associated with the currently running thread:

```
release(void)
{
    /* 1 */
    if (thread_count<=1)
        return 0;

    /* 2 */
    current = current->next;
    switch_context(current->prev, current);
    return 1;
}
```

```
static switch_context(struct context *from, struct context *to)
{
    /* 3 */
    __asm__
    (
        "movl 8(%ebp),%eax\n\t"
        "movl %ebp,(%eax)\n\t"
        "movl 12(%ebp),%eax\n\t"
        "movl (%eax),%ebp\n\t"
    );
}
```

The relase () function calls another function (switch-context()) which is declared to be static so that it cannot be called by any code outside the threads library file. The switch-context() function consists of a few lines of embedded assembly code, which are used to allow direct access to specific processor registers – a level of control not

directly available in standard C. The numbered comments are:

1. **If there is only one thread in the scheduling loop then there is no point in doing a context switch at all. In this case, the release() function just returns straight away and hence the current thread resumes execution until the next release() call is encountered.**

2. **Move current to point to the next thread in the scheduling loop. Then call switch-context(), passing the old value of current and its new value as parameters.**
3. **Inside the switch-context() function there are four instructions in 386 assembly code. All these instructions do is to take the CPU ebp register contents and store them in the ebp field of the from structure, and then load the CPU ebp register from the ebp field of the to structure instead. The switch-context() function then returns.**

Thread's code resumes execution by an ordinary return from the release() function. In fact, as far as any of the threads is concerned, they just call release() at intervals, which then returns, allowing them to continue. The threads themselves are unaware of anything unusual happening during the release() call.

6. Starting New Threads

Starting a new threads just mean creating a struct for the thread and allocating a block of memory for its stack. These data structures can then be initialized and added to the scheduling loop to get the opportunity to run[5]. Thread's stack initialized, as several values need to be hand crafted into the empty stack before it can be used:

The new-thread() function takes two parameters, the first is the address of the function where execution of this thread will begin and the second is the size of the memory block (in bytes) to allocate for stack to the new thread.

```
new_thread(int(*start_addr) (void),int stack_size)
{       struct context *ptr;
        int esp;
        /* 1 */
        if (!(ptr=(struct context *)malloc(sizeof(struct context))))
                return 0;
         /* 2 */
        if (!(ptr->stack =( char *)malloc(stack_size)))
```

```
            return 0;
     /* 3 */
    esp = (int)(ptr->stack+(stack_size-4));
    *(int *)esp = (int)exit_thread;
    *(int *)(esp-4) =(int)start_addr;
    *(int *)(esp-8) = esp-4;
    ptr->ebp = esp-8;

    /* 4 */
    if( thread_count++)
    {
            /* 5 */
            ptr->next=current->next;
            ptr->prev=current;
            current->next->prev=ptr;
            current->next=ptr;
    }       else
    {
            /* 6 */
            current=ptr->next=ptr->prev=ptr;
            switch_context(&main_thread,current);
    }       return 1;
}
```

The numbered comments in the code are as follows:

1. Use malloc() to create a struct context for this thread. Check that the memory allocation was successful and return a zero value on error.

2. Use malloc() again to allocate a block of memory of the specified size for the thread's stack. The stack pointer in the new struct context is set to point to the stack memory. Again, a memory allocation error will cause a zero to be returned by new-thread().

3. Initialize the stack to the state shown in Figure 1 the pointer to the stack memory in the struct context points to the lowest numbered address of the block, wheras the action on the stack is taking place at the high address end of the block.

4. The next part depends on whether or not this is the first new thread within the current process. If it is, execution of the main program is suspended and a context switch to the new thread is performed; otherwise the new thread is just added to the scheduling loop to await its turn to run.

5. Adding a new thread to an existing scheduler loop is made a little more complex by the fact that the loop is doubly linked so that a forward and backward set of pointers need to be set up. A new thread is inserted into the scheduling loop in such a way that it will automatically be the next thread to run when the current thread performed a release().

6. No threads are currently running so the scheduling loop needs to be created and started by a first call to switch-context(). In order to make sure that the main program is suspended, the switch_context called main_thread. This structure is not part of the scheduling loop and so the main program will not be executed again, except by special arrangement, when there are no more threads left to run.

Figure 1 shows that the ebp element of the struct context associated with the new thread is set to point into the thread's stack. When a context switch is made to the new thread, this value is loaded into the CPU ebp register. The return from the switch_context() function then causes the stack pointer to point to the same stack location. Performing a stack pop into ebp loads this register appropriately and also makes the stack pointer point to the thread start function. An extra twist is required when a thread's start function terminates in order to remove the thread from the scheduling loop and free its malloc()ed memory back to the system. These actions are performed by the exit_thread() function.

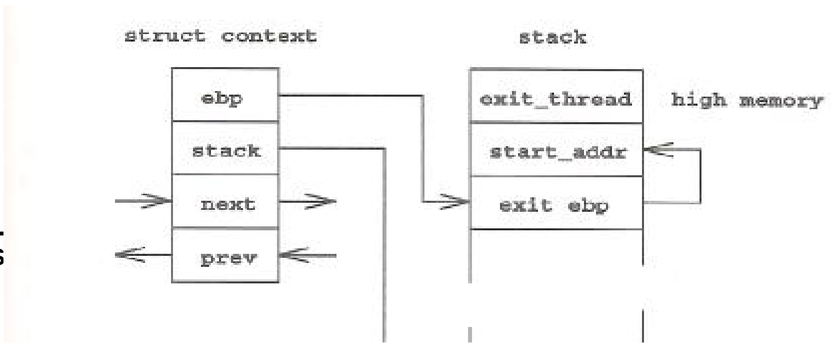
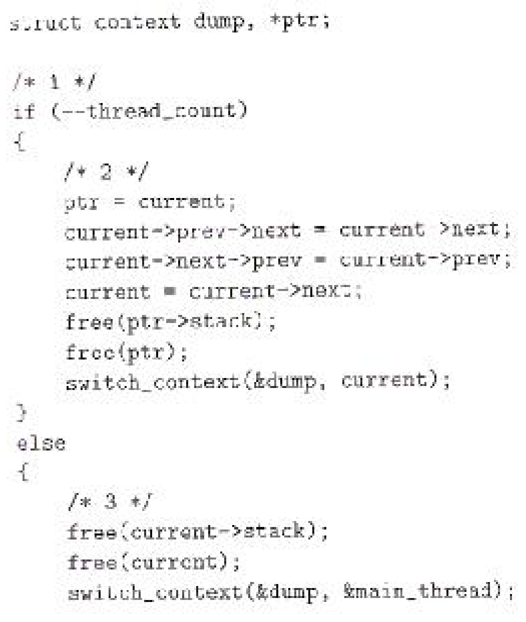
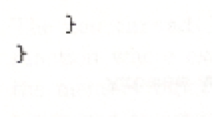


Figure 1:Data structure set up in new thread

This means that the function is automatically executed when the thread's start function terminates, using a similar trick to the one which executed the start function in the first place



```
struct context dump, *ptr;

/* 1 */
if (--thread_count)
{
    /* 2 */
    ptr = current;
    current->prev->next = current->next;
    current->next->prev = current->prev;
    current = current->next;
    free(ptr->stack);
    free(ptr);
    switch_context(&dump, current);
}
else
{
    /* 3 */
    free(current->stack);
    free(current);
    switch_context(&dump, &main_thread);
```



```
    }
}
```

The exit_thread() function terminates the thread pointed to by current. The fact that the exit_thread() function is declared to be static prevent it from being called from outside the library. The numbered comments are:

1. If current is not the last thread in the scheduling loop, then unlink it from the scheduler and switch context to the next thread; otherwise, as there are no more threads to execute, resume execution of the main thread.
2. Take a copy of the pointer to the current thread. Unlink the current thread from the scheduling loop, then reassign current to point to the next thread to execute. Now free() the malloc()ed memory associated with the terminating thread and finally switch_context() from this thread to the new current thread.
3. It is only necessary to free() the malloc()ed memory associated with the final thread in the scheduling loop and then switch_context() back to the main program, to resume its execution.

7. Rendezvous

Two new data structures are used in the implementation of the rendezvous mechanism. The first (struct message) is used on a *per blocked message channel* basis. The second (struct message) is used on a *per blocked message* basis.

These two structures, along with the declaration of struct context and some global variables, etc., are all supplied in a separate header file, called def.h, which should be #included at the top of the program[3].

As each channel is created a struct channel ia allocated and added to a linked list of all the channels within the current process. Whenever an unmatched send() or receive() is performed on a channel, a struct message is allocated and entered on to the end of a message queue, hanging off the channel structure. Whenever a thread is blocked in a rendezvous awaiting a communication partner, the thread structure is unlinked from the scheduling loop and linked instead to its message structure. Figure 2 illustrates how the various structures are interconnected when send() and receive() operations are pending[6].

The diagram shows the existence of three communication channels in this example, whose data structures are linked together, with channel list as the pointer to the head of the list. A send() operation has been performed on channel 1 and a message structure has been attached to the channel structure. The context structure for the thread which performed the send() has been removed from the scheduling loop and attached to the message structure. This ensures that the thread cannot be run again until a matching receive()

operation is performed on the same channel at which time the context structure will be easy to locate and re-connect to the scheduling loop.

The diagram also shows that a second send() operation has also been performed by another thread on the same channel and that it, too, is queued on the channel awaiting the second receive() operation. Communication channel 2 has been created but there are no pending messages queued. Channel 3 shows that a receive() has been performed and that a send() on the channel is awaited.

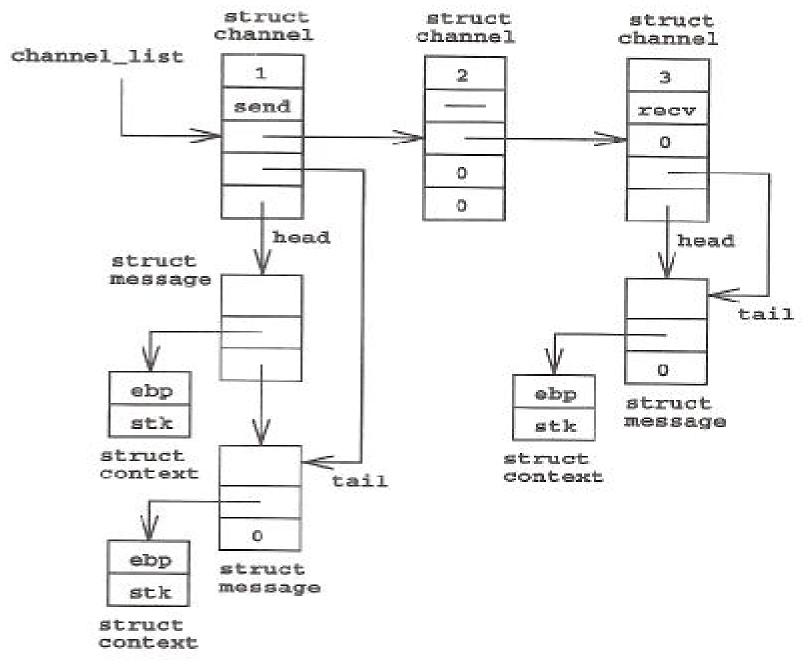
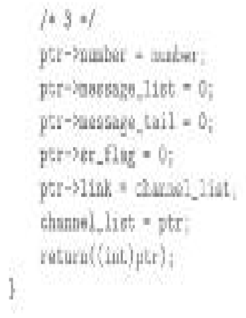


Figure 2:structure links created during rendezvous

8.Creating Channel

New communication channels are created with the get_channel() function as follows:

```
get_channel(int number)
{
    struct channel *ptr;

    /* 1 */
    for (ptr = channel_list; ptr; ptr = ptr->link)
        if (ptr->number==number)
            return((int)ptr);

    /* 2 */
```



```
    /* 3 */
    ptr->number = number;
    ptr->message_list = 0;
    ptr->message_tail = 0;
    ptr->er_flag = 0;
    ptr->link = channel_list;
    channel_list = ptr;
    return((int)ptr);
}
```

The get_channel() function returns a channel descriptor given a channel number as a parameter. If the channel does not already exist, then it will be created. The numbered comments are:

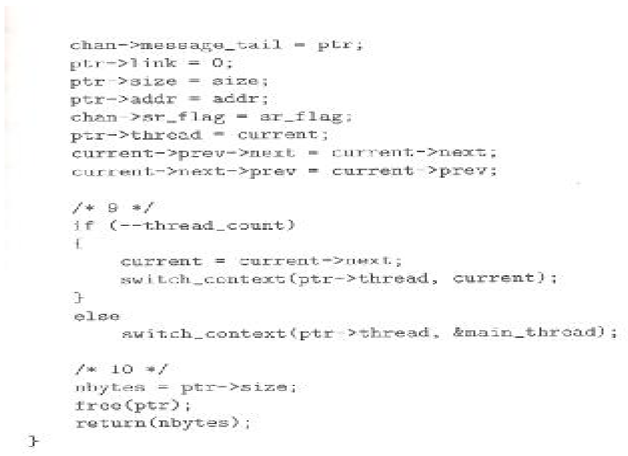
1. Starting at channel_list, search down the linked list of existing channels looking to see if the specified channel number already exists. If the specified number is found in the list then the address of the associated struct channel is cast to an int value and returned as the channel descriptor.
2. If the channel number does not already exist, then a new struct channel is allocated, or a zero is returned on error.
3. After the new struct channel has been successfully created its fields are initialized and the structure is added to the head of the channel list. Finally, the address of the new structure is cast to int and returned as the channel descriptor.

9.Send and receive

Once a channel descriptor has been obtained all that remains is to use send() and receive() to pass messages between threads.

In fact, as the following code shows, send() and receive() are fully symmetrical operations so that they both make an internal call to the function rendezvous() with just an extra flag parameter to specify in which direction the data transfer should be made:

```
send(int chan, char *addr, int size)
{
    /* 1 */
    return(rendezvous((struct channel *)chan, addr, size, 1));
}


receive(int chan, char *addr, int size)
{
    /* 2 */



    return(rendezvous((struct channel *)chan, addr, size, 2));
}


static int rendezvous(struct channel *chan, char *addr,
int size, int sr_flag)
{
    struct message *ptr;
    int nbytes;

    /* 3 */
    if (sr_flag==3-chan->sr_flag)
    {
        /* 4 */
        ptr = chan->message_list;
        chan->message_list = ptr->link;
        ptr->thread->next = current->next;
        ptr->thread->prev = current;
        current->next->prev = ptr->thread;
        current->next = ptr->thread;
        ++thread_count;
        /* 5 */
        nbytes = (size<ptr->size)?size:ptr->size;
        ptr->size = nbytes;

        /* 6 */
        if (sr_flag==1)
            memcpy(ptr->addr, addr, nbytes);
        else
            memcpy(addr, ptr->addr, nbytes);

        /* 7 */
        if (!chan->message_list)
            chan->sr_flag = 0;

        return(nbytes);
    }
    else
    {
        /* 8 */
        ptr = (struct message *)malloc(sizeof(struct message));

        if (!chan->message_list)
            chan->message_list = ptr;
        else
            chan->message_tail->link = ptr;
```



```
    chan->message_tail = ptr;
    ptr->link = 0;
    ptr->size = size;
    ptr->addr = addr;
    chan->sr_flag = sr_flag;
    ptr->thread = current;
    current->prev->next = current->next;
    current->next->prev = current->prev;

    /* 9 */
    if (--thread_count)
    {
        current = current->next;
        switch_context(ptr->thread, current);
    }
    else
        switch_context(ptr->thread, &main_thread);

    /* 10 */
    nbytes = ptr->size;
    free(ptr);
    return(nbytes);
}
```

The send() and receive() function both take three parameters. The first is a channel descriptor, the second is a memory buffer from which, or to which, the data transfer will take place, and the third parameter specifies the size of the data transfer, in bytes. The return value from both functions in the number of bytes actually transferred. This will be the smaller of the two sizes specified as the third parameters to the two functions. The numbered comments are:

1. The send() parameters values are passed directly to rendezvous() with an added fourth parameters value to specify the data transfer direction.
2. The receive() parameter values are passed directly to rendezvous() with an added fourth parameter value to specify the data transfer direction.
3. Inside rendezvous(), the first test checks to see if there is a communication partner of the correct type already queued. If there is, then the partner's struct context needs restoring to the scheduling loop and the data needs to be copied between the threads before the threads can continue their execution. If no partner is queued, then the current thread is removed from the scheduling loop and the communications request added to the queue of the requests associated with this channel.
4. This code section re-links the threads struct context to the scheduling loop and increments the count of the number of the threads ready to run.
5. The calculation is performed to determine the number of bytes that will be transferred. This value is also saved where the communications partner can get at it on its next turn to run.
6. The data transfer now takes place, making sure to copy the data in the right direction.
7. If this was the last message queued in the channel, then the message type flag is reset and the number of bytes transferred is returned to the caller.

8. In this code a new message is to be queued as no matching partners are ready yet. A new struct message is allocated and added to the channel message list. The threads needs to block while waiting for a communication partner so this code also removes the thread's struct context from the scheduling loop and stores it under the message structure.

9. The number of runnable threads is decremented and a context switch is performed to the next thread in the loop. As the newly blocked thread needs to find a communication partner before it can resume execution, there should always be other threads to which control can be passed. If for any reason this proves not to be the case then the context switch arranges to resume execution of the main thread.

10. The next time this thread gets scheduled will be after the rendezvous has taken place, when the number of bytes transferred can be recovered and, after free() ing the used struct message, can be returned.

11. The application (solve Unbounded Buffer problem)

A cooperating process is one that can affect or be affected by other processes executing in the system. Cooperating processes may either directly share a logical address space, or be allowed to share data only through files.

The former case is achieved through the use of threads. Concurrent access to shared data may result in data inconsistency, this requires mechanisms to allow processes to communicate with each other and to synchronize their actions.

Produced-consumer problem is a common paradigm for cooperating processes. A producer process produces information that is consumed by a consumer process, to allow producer and consumer processes to run concurrently, we must have available buffer of items that can be filled by the produced and emptied by the consumer. The producer and consumer must be synchronized so that the consumer does not try to consume an item that has not been produced.

The unbounded-buffer produced consumer problem places no practical limit on the sze of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.[1]

As an application to the use of the threads library, the following code presents a solution to the unbounded buffer problem; the program is developed as a C program Under Linux Operating system:

```
int buffer(void);
int start(void);

int ch_desc1, ch_desc2;

main(void)
{
    ch_desc1 = get_channel(1);
    ch_desc2 = get_channel(2);
    new_thread(start, 1024);
}

start(void)
{
    int i, n;
```

```
    new_thread(buffer, 1024);

    for (i = 0; i<10, ++i)
    {
        send(ch_desc1, &i, sizeof(int));
        release();
    }

    for (i = 0; i<10, ++i)
    {
        receive(ch_desc2, &n, sizeof(int));
        printf("i=%d n=%d\n", i, n);
        release();
    }
}

buffer(void)
{
    int i;

    receive(ch_desc1, &i, sizeof(int));
    new_thread(buffer, 1024);
    send(ch_desc2, &i, sizeof(int));
}
```

In the above program, two channels in use with multiple senders and receivers. Also the program uses one instance of the buffer() function. This solution dynamically creates more buffer space as it is required and it dynamically tidies up after itself when the buffer elements are finished with.

12. Conclusion

In this paper an attempt is made to develop a thread library which can be included and used in any program deals with thread to make the use of thread easy and simple.

While developing the thread library we realized the following aspects:

1. Creating a new thread is cheaper than creating a process

2. Switching to a different thread within the same process is cheaper than switching between threads belonging to different processes.

3. Threads within a process are not protected from one another.

4..All threads in a program must run the same executable. A child process, on the other hand, may run a different executable by calling an exec function.

5.. An errant thread can harm other threads in the same process because threads share the same virtual memory space and other resources. For instance, a wild memory write through an uninitialized pointer in one thread can corrupt memory visible to another thread. An errant process, on the other hand, cannot do so because each process has a copy of the program's memory space.

6. Copying memory for a new process adds an additional performance overhead relative to creating a new thread. However, the copy is performed only when the memory is changed, so the penalty is minimal if the child process only reads memory.

7. Threads should be used for programs that need fine-grained parallelism. For example, if a problem can be broken into multiple, nearly identical tasks, threads may be a good choice. Processes should be used for programs that need coarser parallelism.

8. Sharing data among threads is trivial because threads share the same memory. (However, great care must be taken to avoid race conditions, as described previously.) Sharing data among processes requires the use of IPC mechanisms.

References

[1] P.B.Galvin "Operating System concepts" Fifth edition,Addison-Wesley,1998.

[2] G.Coulouris "Distributed systems" third edition, Addison-Wesley,2001.

[3]Phil Cornes, "The Linux A-Z" Prentice Hall Europe,2000.

[4]D.P.Bovet and M.Cesati "Understanding the Linux kernel",O'Reilly &Associates,Inc.2001.

[5]Linux tutorial "http//www.yolinux.com/tutorial/"

[6] P.B.Galvin "Applied OS concepts", Addison-Wesley ,1999.

انشاء مكتبة روابط لحل مشكلة تزامن البرامج

د.ندا عبدالزهرة عبدالله

جامعة بغداد

المستخلص :

أكثر البرامج يراد أن تكتب بطريقة لتهتم بوجود بعضها البعض و تتعاون مع بعضها لانجاز هدف معين. الحل الذي يضمن تعاون البرامج مع بعضها هو ميكانيكية الأتصال المتداخل للبرامج. بغض النظر عن الميكانيكية المستخدمة، هناك كلفة إضافية متأتية من تبديل سياق تنفيذ البرامج طالما إن هناك برامج تعمل معاً.

للتغلب على هذه المشكلة يفضل استخدام مفهوم الروابط. فكرة الروابط هو إن كل برنامج يكون له جزء من وقت المعالج مشترك بين عدد من الروابط تعمل في نفس الوقت ,وكل رايط يشترك مع بقية الروابط بنفس هياكل البياناتو الذاكرة للبرنامج التابعين له.

في هذا البحث تم إنشاء مكتبة روابط جديدة وسهلة الأستخدام في مستوى المستخدم. وكتطبيق على استخدام هذه المكتبة تم بناء برنامج لحل مشكلة الخزن غير المحدد في المخزن المؤقت .أستخدمت لغة سي للبرمجة تحت نظام التشغيل لينكس.