1993

# Design and Implementation of a Threads Library

Janche Sang

Felipe Knop

Vernon Rego
*Purdue University*, rego@cs.purdue.edu

Report Number:
93-043

# Design and Implementation of a Threads Library

Janche Sang and Felipe Knop
Vernon Rego

CSD-TR-93-043
July 1993

# Design and Implementation of a Threads Library

Janche Sang
Felipe Knop
Vernon Rego*
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

July 12, 1993

**Abstract**

The purpose of the Xthreads library is to provide a cheap concurrent programming environment. The design of Xthreads library is patterned after Xinu, a small and elegant operating system developed at Purdue University. The processes in the Xinu operating system share a single address space and hence enjoy reduced overheads in process creation, interprocess communication, etc. Our approach is to map the Xinu process structure into the Xthreads thread structure in a Unix-like process. This report describes the design and implementation issues of the Xthreads library on the nCUBE2, iPSC860 and RS6000 machines and some early performance measurements of the system on these machines.

* Research supported in part by NSF award CCR-9102331, NATO award 900108 and the Mathematical Sciences Section of Oak Ridge National Laboratory under contract contract DE-AC05-84OR21400 with Martin Marietta Energy Systems, Inc.

# 1 Introduction

Lightweight processes are threads of control existing within a single host process, and consequently sharing a single address space. In fundamental structure, a lightweight process is no different from a process; each has its own stack, local variables, and program counter. However, as compared to a process, a lightweight process is lighter in terms of the overhead associated with creation, context-switching, interprocess communication, and other routine functions. This is because these primitives can be executed within a single domain. In general, the purpose of a threads system is to provide a cheap concurrent programming environment within a process.

In this report, we describe issues of design and implementation concerning the library *Xthreads* on the nCUBE2 machine and the modifications to port Xthreads on the iPSC860 and RS6000 computers. The design of the Xthreads library is patterned after Xinu [2], a small and elegant operating system developed at Purdue University over several years. The processes in the Xinu operating system share a single address space and hence enjoy reduced overheads in process creation, interprocess communication, etc. Our approach is to map the Xinu process structure into the Xthread thread structure in a Unix-like process. We provide a high-level concurrent programming interface that is simple and straightforward, but sophisticated enough to meet the needs of a wide range of applications.

# 2 Design Issues

The design of our thread mechanism was strongly influenced by the need to parallelize a process-interaction simulation system called $\mathcal{S}i$ [8]. The original $\mathcal{S}i$ system encapsulates the SUN LWP library and enhances the capabilities of the C programming language in the form of library primitives with sets of predefined data structures. However, the disadvantage of using a vendor-supplied package is its lack of portability. This makes running simulations over a network of heterogeneous machines impossible. Moreover, the $\mathcal{S}i$ system had to build its scheduling discipline on top of LWP scheduling routines, which increased the overheads in $\mathcal{S}i$. Since simulations are usually time-consuming, the need for significantly reducing the execution time of simulation programs is a critical one. Therefore, constructing a threads library with efficiency and modularity was a major concern.

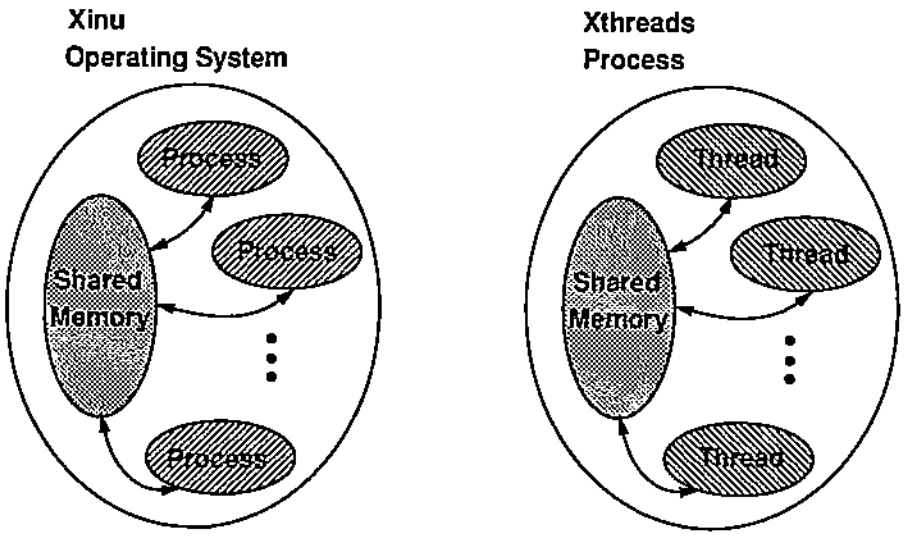


Figure 1: Xinu vs. Xthreads

Our approach to building a threads library was to follow the Xinu system structure. The Xinu operating system supports shared memory between processes. If we view the Xinu system as a process in UNIX, then a process in Xinu becomes a thread in a Unix process (see Figure 1). However, it is unnecessary to translate all layers of Xinu into a process since the organization of an OS is more complicated than what is required. We remove some upper layers such as the device driver layer, file system layer, etc. and focus only on the process manager, coordination, and interprocess communication layers in Xinu. There are two advantages of designing a threads library by patterning it after an existing operating system. First, a well-defined OS provides skeletons which ease the design work. Second, the layering structure in an OS provides modularity that supports easy modifications and extensions.

The Xthreads library is currently supported at the user level. This means that all thread operations, such as creation, synchronization, context switching, etc. require no intervention from the operating system kernel. Therefore, a user level threads library can be more efficient than a kernel supported threads system in terms of operation costs[6]. Furthermore, a user-level threads library is easily portable to other UNIX systems. Several existing thread packages are based on the user-level approach[3, 7]. However, user-level threads suffer performance losses, in that when a thread invokes a system call or encounters a page fault, the whole process has to be blocked. This effect can be attenuated (but not eliminated) by using non-blocking system calls

```
/* creation and destruction */
XTCALL xthread_create(xptr,attrp,func,narg,arg1,arg2,...);
XTCALL xthread_destroy(xptr);

/* destroy itself */
XTCALL xthread_exit();

/* yield control to a thread pointed to by xptr */
XTCALL xthread_yield(xptr);

/* find out who I am */
XTCALL xthread_self(*xptr);

/* find out if a thread alive or not */
XTCALL xthread_ping(xptr);

/* event */
XTCALL xthread_event();
XTCALL xthread_wait(e);
XTCALL xthread_set(e);

/* message passing */
XTCALL xthread_send(xptr,msg);
XTCALL xthread_receive();
```

Table 1: The Primitive Functions in Xthreads

and signals. For example, a thread calling asynchronous I/O routines is blocked in a waiting queue and will be unblocked by the SIGIO signal when the I/O operation is completed.

# 3 Implementation Issues

The primitive functions of Xthreads are shown in Table 1. The type XTCALL is a predefined integer type. The Xthreads routines will return XTERR which is a predefined constant $-1$ if unsuccessful completion of the operation occurs. For example, trying to use `xthread_destroy()` to kill a non-existent thread will return XTERR to the caller.

### Stacks Allocation

For a normal program which is not using the Xthreads library, the program structure on system memory usually consists of four parts: user code area, global data area, dynamic heap

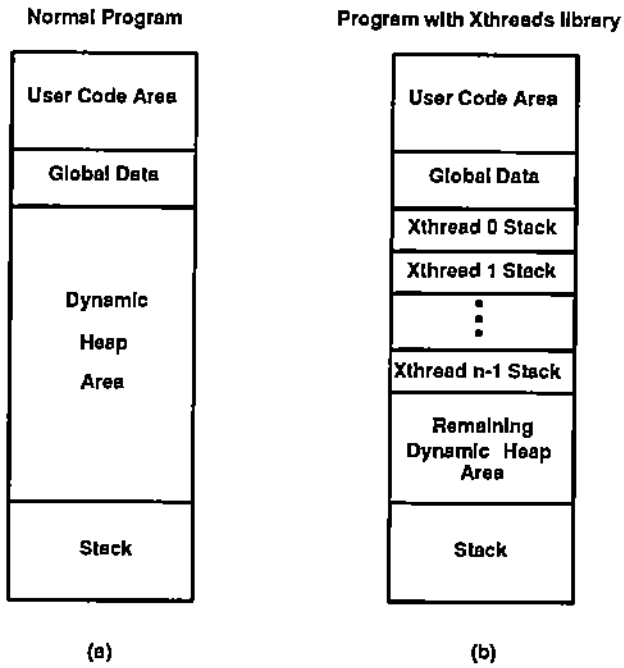


Figure 2: The Memory Layout

area, and runtime stack, as shown in Figure 2(a). Note that these four areas may not be contiguous in memory. The stack is used, when a function is invoked, for storing the actual parameters, local variables, return address to the caller, etc. Since a thread has its independent flow of control, a separate stack is required for each thread. Our implementation strategy is to allocate these default stacks in dynamic heap area via `malloc()` system calls during system initialization stage. A key advantage of stack pre-allocation is the reduction of thread creation overhead. A similar experience has been reported in [1, 9, 7]. The re-organized memory layout can be found in Figure 2 (b).

Though we pre-allocated a default stack for each thread, users can still provide a different stack area through the `attr` attribute structure pointer which is the second parameter of the function `xthread_create()` (to be described later).

An alternative approach to allocating the threads stacks is to break the original stack into several pieces via `alloca()`.

### The Thread Table

The Xthread table, `xtab`, is an array with the size of `NPROC` entries. For each thread, there is one entry in this thread table (see Figure 3) (also called Thread Control Block).

The Xthread library stores all information about threads, such as thread identifier, thread priority, thread state, registers, etc. in the thread table. he array structure yields a simple and efficient implementation because it can reduce the number of times the function *malloc()* is called. Though the table pre-allocation limits the number of threads in a program, we may use *realloc()* or *malloc()* to obtain another threads table pool at run time, to increase the array size when all entries in the array have been used. The structure of the Xthread table is declared in the file `xthreads.h` and initialized in the file `main.c`.

In the thread entry,tThe field `xid` represents the identifier of a thread. The boundaries of each thread's stack are stored in `xlimit` and `xbase`. The array `xregs` is used as a buffer to save register values when thread context switch occurs.

**The Calling Sequence on nCUBE2**

To build the Xthreads library, the machine dependent portion merely exists where a thread is being created and where a thread context switch occurs. Therefore, it is necessary to know the calling conventions of the system on which Xthread was built. This is not only hardware dependent but also compiler dependent. Our implementation on nCUBE2 is based on the Parallel Software Environment (PSE) version 3.1.

The PSE's calling convention passes the first four parameters in registers (though these parameters also have some space on stack), while the rest parameters (they are called *overflow parameters*) are push onto the stack backward. The term 'backward' means that the direction is reverse to the direction which the stack grows. In Figure 5, we show a simple function `foo()` with six parameters and its corresponding assembly code. The assembly code is obtained by using the -S option when compiled.

Inside the assembly code from Line 2 to Line 9, the offsets of the parameters and local variables are defined. These offsets are related to the adjusted stack pointer, which is updated in Line 13 when just entering the function `foo()` and is reset back in Line 24 before returns. From these two lines, we know that the stack grows from high address memory area to low address memory area.

The growth of stack is depicted detailly in Figure 6. Assume the `main()` function calls the function `foo()`. Before transfering the control to `foo()` (the left part in Figure 6), the return address and parameters have to be saved on the stack according to their corresponding positions. Note the return address is saved on the stack where the current stack pointer points

```
 1  /*    xthreads.h    */

 2  typedef int WORD;

 3  /* nCUBE2 machine dependent */
 4  #define PNREGS 22              /* size of saved register area      */

 5  #define SP     0               /* index in the xregs               */
 6  #define PC     1
 7  #define R0     2

 8  /* state */
 9  #define XFREE    0             /* thread slot is free              */
10  #define XBORN    1             /* baby thread, also on ready queue */
11  #define XREADY   2             /* thread  is on ready queue        */
12  #define XCURR    3             /* thread is currently running      */

13  #define NPROC   64             /* set the number of threads        */

14  #define STKSIZE 4096           /* default stack size               */

15  #define DEFAULT_PRIO 20        /* default priority                 */

16  struct xentry  {

17          int  xid;              /* thread id                        */
18          WORD xbase;            /* base of stack (lower bound)      */
19          WORD xlimit;           /* stack upper boundary             */
20          int  xstate;           /* thread state: XCURR, etc.        */
21          int  xprio;            /* thread priority                  */
22          WORD xregs[PNREGS];    /* save SP, ...,                    */
23  };

24  typedef struct xentry * xthread_t;

25  typedef int XTCALL;
26   .....
```

Figure 3: The header file xthreads.h

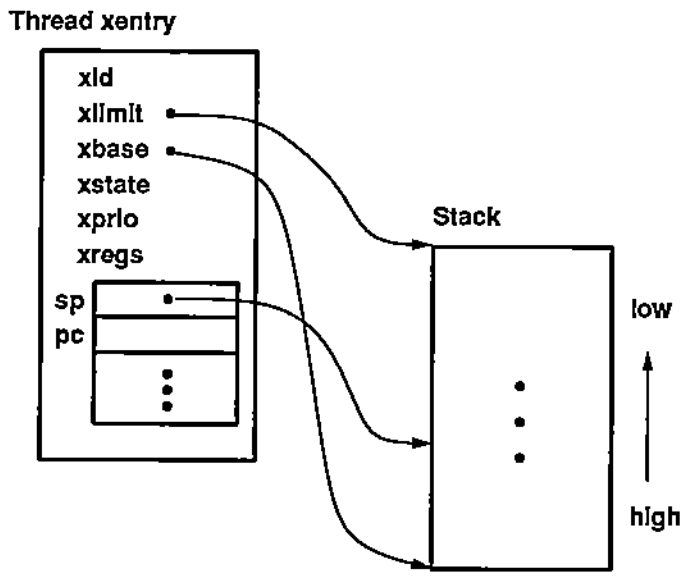


Figure 4: The thread structure

to. Upon leaving the function `foo()`, this return address will be popped up into program counter, while stack pointer is adjusted back to the place before calling `foo()`.

### Thread Creation

A thread can be created by calling the function `xthread_create(xtptr, aptr, procaddr, nargs, arg1, arg2, ...)`, as shown in Figure 7. At beginning, this function calls an internal function `new_xthread()` to find an unused (free) slot in the `xtab` table and then fills necessary information to the entry. Note that we don't need to allocate a stack for each thread creation, since each thread entry is associated with a default stack space. Therefore, due to the saving of the system call to `malloc()`, the time to create a thread can be greatly reduced.

Patterned after the Xinu design for process creation, we make a *pseudo call* by pushing parameters and return address on the stack to simulate a procedure call which starts a thread execution. Therefore, when the thread starts, it behaves exactly as if it had been called from another procedure. The only difference is that, when the thread terminates execution, it will not be returned to the place where it was created. Instead, it returns to a designated function `userret()` since the address of function `userret()` was pushed on the stack as the return address when we made the pseudo call (see Line 54) . In `userret()` (see Figure 8), it releases the dying thread table entry by setting the `xstate` field to be XFREE and calls the scheduling function `resched()` (will be discussed later) to select the next thread to run. We stored the

```
1       int foo(a,b,c,d,e,f)
2       int a,b,c,d,e,f;
3       {
4       int x, y;

5       x = a + b + c ;
6       y = d * e * f ;
7       return(x+y);

8       }
```

```
1     .file     "foo.c"

2   e.4     .equ     32
3   f.5     .equ     36
4   y.7     .equ     4
5   x.6     .equ     8
6   d.3     .equ     12
7   c.2     .equ     16
8   b.1     .equ     20
9   a.0     .equ     24

10    .code


11  _foo:
12       .pub     _foo
13       adsp     #-28
14       movw     r0, a.0(sp)
15       movw     r1, b.1(sp)
16       movw     r2, c.2(sp)
17       movw     r3, d.3(sp)
18       addw3     b.1(sp), a.0(sp), r0
19       addw3     c.2(sp), r0, x.6(sp)
20       mulw3     e.4(sp), d.3(sp), r0
21       mulw3     f.5(sp), r0, y.7(sp)
22       addw3     y.7(sp), x.6(sp), r0

23  $b3.1:
24       adsp     #28
25       ret
26       .elftype     _foo, ?function
27       .size     _foo, .-_foo
```

Figure 5: A function in C and its corresponding Assembly code

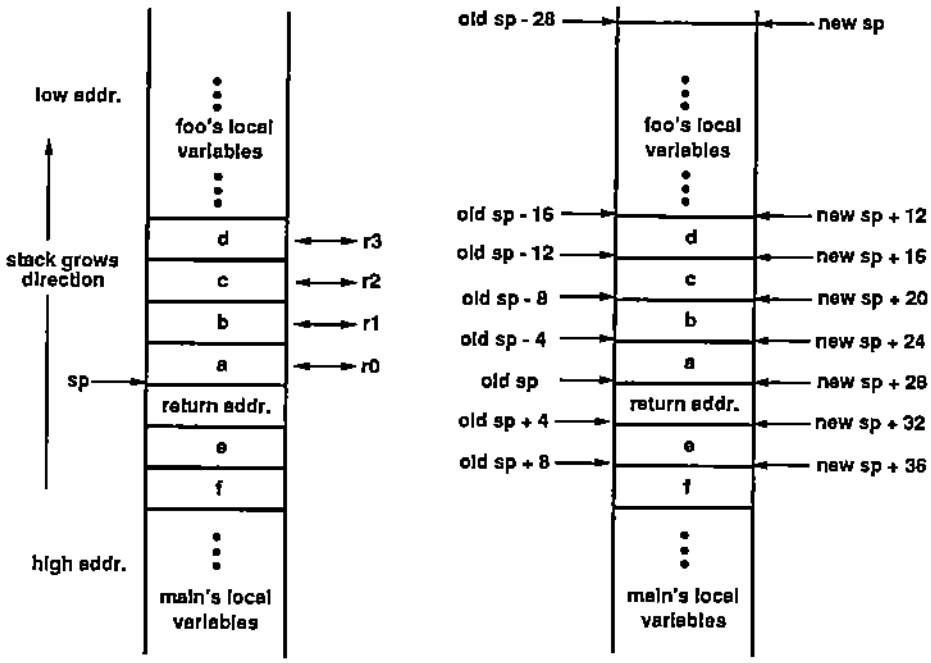


Figure 6: The PSE's Calling Convention on nCUBE2

starting address and stack pointer in the thread registers buffer `xregs` (Line 52 and 53). These two values will be loaded into the program counter register and the stack pointer register when the thread starts execution.

We used C library `varargs` to access variable arguments on stack and pushed these arguments onto newly created thread's stack. In the function `xthread_create()`, Line 45 to Line 49 are to store the first four arguments in the registers buffer `xregs`, while Line 55 and 56 are to store the overflow parameters. We show two examples in Figure 9. The stack in the left is for the new thread running function `foo()` with six parameters. The stack in the right is set up for the new thread executing function `bar()` with only three parameters.

### Scheduling in Xthreads

The current Xthreads library uses a simple scheduling policy based on priority. The scheduling principle is that the highest priority thread with state ether XREADY, or XBORN, or XCURR has the right to run. If two threads have the same highest priority, First-Come, First-Serve rule is applied. The current running thread can run utill it is terminated or suspended.

The function `resched()`, as shown in Figure 10, does the thread scheduling task. Recall that the function `userret()` will mark the current running thread state to be XFREE and

```
1
2  #include <stdio.h>
3  #include <xthreads.h>
4  #include <varargs.h>

5  xthread_t new_xthread()
6  {
7      xthread_t xptr;
8      int i, xid;
9
10     for(i=0; i<NPROC; i++) { /* check all slots */
11         xid = nextproc;
12         if((++nextproc) >= NPROC)
13             nextproc = 0;
14         xptr = &xtab[xid];
15         if(xptr->xstate == XFREE)
16             return(xptr);
17     }
18     printf("XThreads Error: run out of process table ! \n");
19     exit(0);
20 }

21 /*------------------------------------------------------------------
22  *  xthread_create  -  create a process to start running a procedure
23  *------------------------------------------------------------------
24  */
25 xthread_create(xtptr,aptr,procaddr,nargs)
26 xthread_t *xtptr;      /* pointer to newly created thread struct */
27 xthread_attr_t *aptr; /* pointer to newly created thread attr */
28 int *procaddr;        /* procedure address            */
29 int narge;            /* number of args that follow   */
30 va_dcl                /* arguments (treated like an   */
31                       /* array in the code)           */
32 {
33     WORD *saddr;              /* stack address */
34     xthread_t cptr,xptr;
35     int i;
36     va_list args;
37
38     xptr = new_xthread(); /* get a new xentry in the xtab */
39     xptr->xstate = XBORN; /* it's a baby; it doesn't know how to walk */
40     xptr->xprio = DEFAULT_PRIO; /* default priority */
41     insert(xptr->xid,rdyhead,xptr->xprio);
42
43     *xtptr = xptr;

44     /* move args to the stack */
45     va_start(args);
46     saddr = xptr->xbase - ( (narge > 4)?(nargs-4+1):0 ) * sizeof(WORD);
47     for(i=0; i<=3 && nargs > 0; i++, nargs--)
48         /* store paprmeters in registers */
49         xptr->xregs[i+R0] = *saddr -- = va_arg(args,WORD);

50     saddr = xptr->xbase - nargs * sizeof(WORD);
51
52     xptr->xregs[PC] = procaddr; /* starting address */
53     xptr->xregs[SP] = saddr ;   /* SP value which points to userret */
54     *saddr ++ = userret;

55     for(i=0; i<nargs; i++) /* store overflow parameters on the stack */
56         *saddr ++ = va_arg(args,WORD);
57 }
```

Figure 7: The function xthread_create()

```
 1 #include <proc.h>

 2 /*-----------------------------------------------------------------
 3  * userret  --  entered when a thread exits by normal return
 4  *-----------------------------------------------------------------
 5  */
 6 void userret()
 7 {
 8     _currxt->xstate = XFREE;
 9     resched();
10 }
```

Figure 8: The function userret()

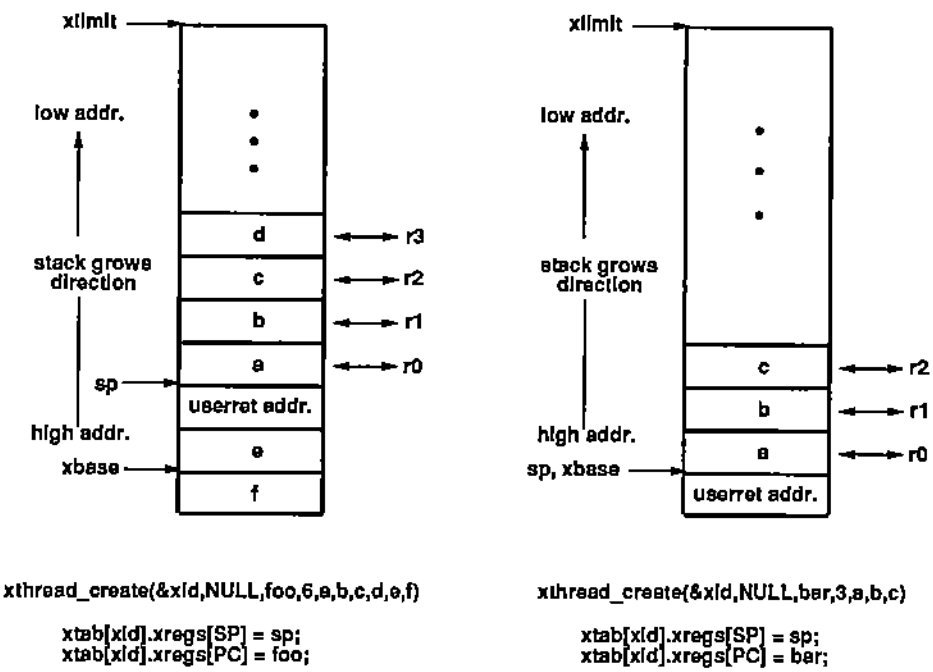


Figure 9: The Stack of Pseudo Call on nCUBE2

invoke `resched()`. Therefore, testing the thread state in Line 16 and 18 will prevent a dead (or even suspended) thread running next. After the thread with the highest priority is selected in Line 22, we will check whether this thread has started or not. If its state is XBORN which means the thread is newly created, the function `xtstart()` will be invoked to load the arguments and the thread starting function address into registers. Otherwise, `ctxsw()` is invoked to do simple context switch.

Though we only provide a priority-based scheduling policy in Xthreads, it is possible to build other scheduling policies on top of the default policy. For example, a round-robin time-sliced scheduler can be implemented by using the `signal()` and `setitimer/alarm()` UNIX functions. When the quantum which is given to the executing thread expired, a SIGALARM signal handling routine will be invoked. This routine will call the function `resched()` to suspend the current running thread, resume the execution of the other thread, and issue a SIGALARM request for the next quantum. Since the execution of a thread can be preempted unwillingly, mutual exclusion is required in some critical regions. For example, the SIGALARM signal has to be disabled when entering the function `xthread_create()` and restored when returning.

**Context Switching**

In the Xthreads library, the only code written in assembly language is in the context switching function. This is because context switching involves registers saving and restoring, which is hard to manipulate in high-level languages. The function `xtstart()`, shown in Figure 11, is used to transfer the control to a new thread which has not be executed yet. There are two input parameters of this function: the first one is the address of the current thread registers buffer and the second one is the address of the new thread registers buffer. Recall that in the nCUBE2 PSE environment, these two parameters will be passed via the register `r0` and `r1` respectively. Besides, the return address has already been pushed on the current running thread's stack before executing the function `xtstart()`. The processor register *sp* points to this location. Therefore, we only need to save the stack pointer into the register buffer (Line 3). After saving all the non-volatile registers (from r6 to r15), the new thread's stack pointer is loaded to register sp. The new thread starting function address is loaded (in Line 7) to a temporary register r4 and then is transfered to the program counter by using the jmp instruction in Line 12. The lower part in Figure 11 shows the stacks and register buffers of two threads;

```
 1  /* resched.c  -  resched */
 2  #include <stdio.h>
 3  #include <xthreads.h>
 4  #include <q.h>

 5  /*------------------------------------------------------------------------
 6   * resched  -- find the highest priority thread to run
 7   *
 8   *------------------------------------------------------------------------
 9   */
10  int    resched()
11  {
12          register struct xentry *cptr; /* pointer to old thread entry */
13          register struct xentry *xptr; /* pointer to new thread entry */
14          int temp;

15          cptr = _currxt;
16          if((cptr->xstate == XCURR) && (lastkey(rdytail) < cptr->xprio))
17                return;

18          if(cptr->xstate == XCURR) {
19              cptr->xstate = XREADY;
20              insert(cptr->xid,rdyhead,cptr->xprio);
21          }

22          if((temp = getlast(rdytail)) != EMPTY) {
23              _currxt = xptr = &xtab[ temp ];
24              if(xptr->xstate == XBORN) {
25                  xptr->xstate = XCURR;
26                  xtstart(cptr->xregs,xptr->xregs);
27              }
28              else {
29                  xptr->xstate = XCURR;
30                  ctxsw(cptr->xregs,xptr->xregs);
31              }
32              /* The OLD thread returns here when resumed. */
33          }
34          else
35              exit(1);
36  }
```

Figure 10: The function resched()

thread A is yielding control to a new thread B.

If the next running thread is not newly created, i.e, its state is not XBORN, the function `ctxsw()` will be invoked (see Figure 12). Similar to `xtstart()`, `ctxsw()` first saves the stack pointer of the current executing thread (i.e., the thread A in Figure 12) and then loads the stack pointer of the thread to be resumed (i.e., the thread B). The tricky part is the `ret` instruction in Line 7. Since the thread B's return address is saved on the stack, executing the command `ret` causes this address being loaded into the program counter register. Therefore, each thread will be resumed at either Line 32 in `resched()` or Line 29 in `xthread_yield()`, the place immediately after calling `xtstart()` or `ctxsw()`.

**Thread Control Transfer**

Xthreads offers a primitive function `xthread_yield()` (see Figure 13) which can transfer the control to a designated thread, while itself is suspended and waiting for resumption. However, the transfer cannot conflict with the scheduling policy. For example, if the priority based scheduling is used, a thread is not allowed to yield control to another thread with lower priority. This makes some of the code in `xthread_yield()` (from Line 11 to Line 13 and Line 16 to Line 18) dependent on scheduling disciplines. The rest part `xtstart()` or `ctxsw()` is the same as the code in `resched()`.

**Xthreads Initialization**

The Xthreads library has already provided the `main()` function (Figure 14) which does the initialization work. It first initializes system data structures such as the thread table, priority list, etc. and then creates the first thread which will execute the `xmain()` function supplied by users. The `xmain()` function of the application code can be treated like the main() routine in the C language.

**Porting the Thread Library on iPSC860 and RS6000**

In this subsection, we describe the necessary changes to port the Xthread library on iPSC860 and RS6000 machines. Since the library is designed modularly, the modifications is only limited in machine dependent portion, i.e., the thread creation and context switching functions. Table 3 lists the calling conventions of these two machines. Details can be found in [4, 5].

It is worthy to mention that the stack alignment is an important issue. For example, the iPSC860 requires the stack to be aligned on 16-byte boundaries to keep data arrays aligned.

```
 0  ! xtstart(cptr->xregs,xptr->xregs): r0 <-- cptr->xregs, r1 <-- xptr->xregs
 1 _xtstart:
 2    .pub _xtstart
 3    stpr      #16,(r0) ! save current thread  SP
 4    ... saving non-volatile registers to current thread's buffer ...
 5    ldsp      (r1)     ! load new SP with the addr containing the ret addr
 6    movw       r1, r0  ! use r0 as index
 7    movw    4(r0), r4  ! r4 contains the procaddr
 8    movw   20(r0), r3  ! put the fourth parameter (if undefined, garbage)
 9    movw   16(r0), r2  !    :    third     :
10    movw   12(r0), r1  !    :    second    :
11    movw    8(r0), r0  !    :    first     :
12    jmp     (r4)       ! jump to procaddr
13    .elftype    _xtstart, ?function
14    .size       _xtstart, .-_xtstart
15    .data
```

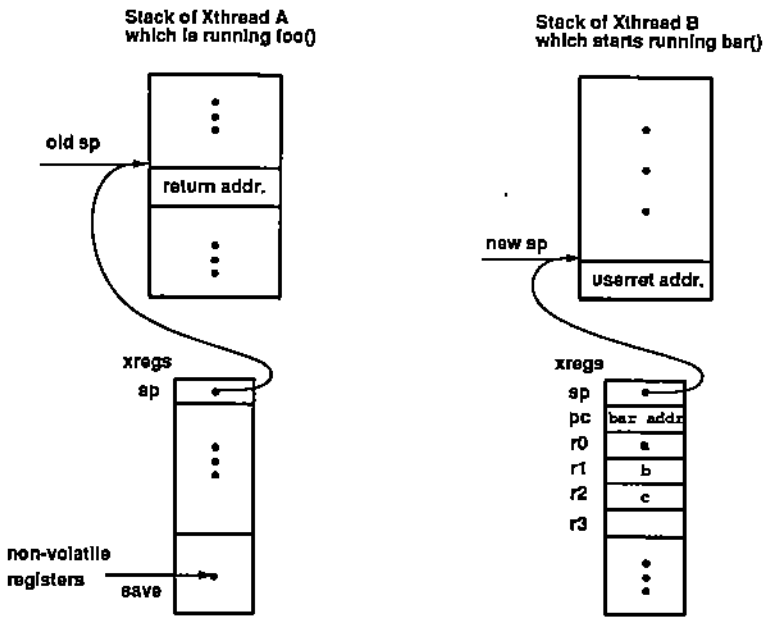


Figure 11: The function xtstart()

```
0   ! ctxsw(cptr->xregs,xptr->xregs): r0 <-- cptr->xregs, r1 <--  xptr->xregs
1   _ctxsw:
2      .pub _ctxsw
3      stpr    #16,(r0)      ! save current sp in current thread buffer
4      ... saving non-vloatile registers to current thread's buffer ...
5      ... restoring non-vloatile registers to next thread's buffer ...
6      ldsp    (r1)          ! load sp from next executing thread buffer
7      ret                   !
8      .elftype    _ctxsw, ?function
9      .size       _ctxsw, .-_ctxsw
```

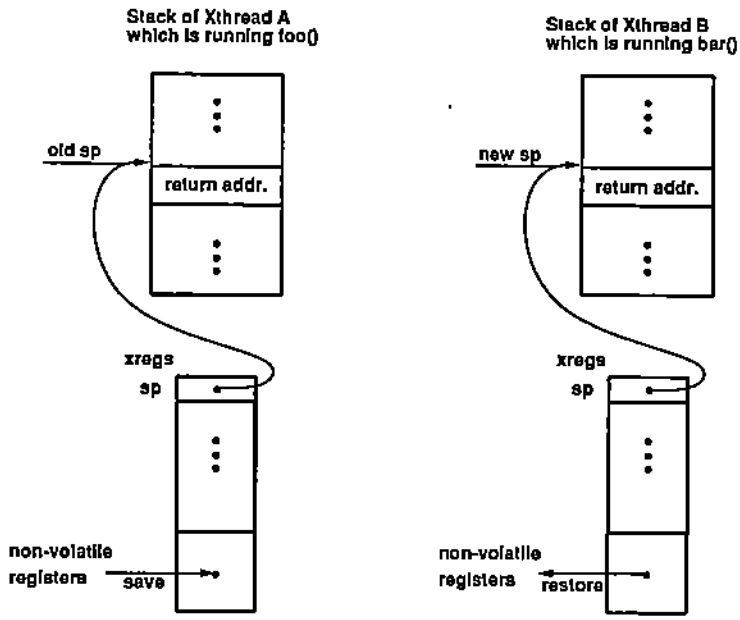


Figure 12: The function ctxsw()

| | iPSC860 | RS6000 |
|---|---|---|
| stack register | register r2 | register r1 |
| return address | stored in register r1 | stored in link register |
| input parameters | the first 12 arguments are passed in register r16 to r27. | the first 8 arguments are passed in register r3 to r10. |

Table 2: Calling conventions for iPSC860 and RS6000 machines

```
1  #include <xthreads.h>

2  /*-----------------------------------------------
3   *  xthread_yield  -  yield control to a thread
4   *-----------------------------------------------
5   */
6  void xthread_yield(xptr)
7  xthread_t xptr;
8  {
9      register xthread_t cptr;

10     cptr = _currxt;
11     /* priority baed scheduling dependent */
12     if(xptr->xprio < cptr->xprio) /* fails if yield to a low prio thread */
13             return;

14     _currxt = xptr; /* update global variable : current xt pointer*/
15
16     /* the following is for priority basd scheduling */
17     dequeue(xptr->xid);
18     insert(cptr->xid,rdyhead,cptr->xprio);
19     cptr->xstate = XREADY;
20
21     if(xptr->xstate == XBORN) {
22         xptr->xstate = XCURR;
23         xtstart(cptr->xregs,xptr->xregs);
24     }
25     else {
26         xptr->xstate = XCURR;
27         ctxsw(cptr->xregs,xptr->xregs);
28     }
29 }
```

Figure 13: The function xthread_yield()

```
 1  #include <stdio.h>
 2  #include <proc.h>
 3  #include <q.h>

 4  extern int xmain();

 5  struct xentry xtab[NPROC];

 6  xthread_t _currxt;

 7  int nextproc = 0;

 8  struct qent q[NPROC+2];
 9  int rdyhead, rdytail;

10  main(argc, argv)
11  int argc;
12  char *argv[];
13  {
14      register struct xentry *xptr;
15      struct xentry mainentry;
16      xthread_t xmainp;
17      int i;

18      for(i=0 ; i < NPROC; i++){
19         xptr = &xtab[i];
20         xptr->xid = i;
21         xptr->xlimit = malloc(STKSIZE);
22         xptr->xbase = xptr->xlimit + STKSIZE - sizeof(WORD);
23         xptr->xstate = XFREE;
24      }

25      rdytail = 1 + (rdyhead = newqueue());
26      xthread_create(&xmainp,NULL,xmain,2,argc, argv);

27      _currxt = &mainentry;  /* main is pretended as a thread */
28      _currxt->xstate = XFREE;
29      resched();
30      /* never returns */

31  }
```

Figure 14: The function main()

Failure to keep this alignment would raise exceptional conditions which increase the execution time tremendously.

Because of some unknown reason, we cannot use the assembly command `brl` to transfer control to a newly created thread in the RS6000 machine, while assigning the thread's starting function address to the link register. Our current solution is to jump to a intermediate routine in which the thread's starting function will be invoked.

### Event

The Xthreads library provides two distinct coordination mechanisms to support synchronization between processes. One mechanism is through *events*, effected by calling *xthread_wait()* and *xthread_set()* primitives. A thread is suspended and put in the waiting queue if it invokes function *xthread_wait(e)* while event e has not yet occurred. Event e is said to occur when some other thread invokes function *xthread_set(e)*. At this point, all processes waiting for event *e* are reactivated simultaneously and put back on the ready list. In Xthreads, an event e is declared to be of type Xevent and initialized by the xthread_event() function.

### Message Passing

The other mechanism for thread synchronization is through message-passing. Messages can be sent and received by using the functions *xthread_send()* and *xthread_receive()*, respectively. The function *xthread_send(xptr,msg)* deposits the message *msg* to the thread *xptr*. If the thread *xptr* is awaiting the arrival of this message, *xthread_send()* enables it to access the message and consequently be reactivated. The reverse function *xthread_receive()* returns the message that has been sent to the thread. If no message is available, the invoking thread has to be suspended until a message arrives. For simplicity, the size of a message is limited to one word (i.e., the size of an integer or pointer). However, since threads can access shared memory, it is also possible to pass pointers to buffers as messages.

### Software Interrupts Handling

Since Xthreads is implemented at user level, interrupts are represented by UNIX signals. Therefore, we have to provide an interrupt handling routine for each type of signal. For example, in order to overlap communication with communication, we used the iPSC860 supported function *hrecv(long typesel, char *buf, long len, void (*proc)())* to provide a receive handler for receiving message from the remote processors in the iPSC860 parallel machine. Users can use the Xthreads supported function *xrecv()* to get a message. Function *xrecv()*, first checks

| Operations | nCUBE2 | iPSC860 | RS6000 | Sparc IPC |
|---|---|---|---|---|
| Thread Creation | 87 | 13 | 15 | 660 |
| Thread Switch | 30 | 5 | 6 | 70 |
| Process Fork | 97750 | NA | 1143 | 4200 |
| Process Switch | 90 | NA | 37 | 133 |
| MIPS | 7 | 40 | 28.5 | 15.7 |

Table 3: Latency of Operations (in microseconds)

whether the message has arrived or not. If there is such message, it will return the message immediately. Otherwise, it suspends the currently running thread and calls *resched()* to switch to a runnable thread. When the message does arrive, the function *(*proc)()* will be invoked and will unblock the suspended thread. Note that this interrupt handling portion in Xthreads is system dependent since different operating systems may support different signals.

# 4 Performance

In this section we present some early performance measurements for the tasks of thread creation and thread switching time in Xthreads on the nCUBE2, iPSC860 and RS6000 machines. For measuring the creation plus deletion time, we created a thread which will terminates immediately after starting to execute (see Figure 15). For measuring the thread switching time, we created two threads which yield to one another for a large number of times and then calculated the average(see Figure 16). Note that the times presented here also include the overhead resulting from priority-based scheduling in Xthreads. Table 4 shows the early measurements of the operation cost in Xthreads. For comparison purpose, we have added the process forking time and process context switching time. The technique used in measuring the process operation costs is through the use of signals. For example, A process signals another suspended process and then suspends itself for a signal back from the recently awaken process. This technique has been used in the early study ??.

# 5 Conclusions and Experiences

Our approach to designing and implementing a simple threads library, following an operating system structure, has been proved successful. Because of its simplicity, the operations of

```
 1  #include <stdio.h>
 2  #include <time.h>

 3  #include <xthreads.h>

 4  #define CPU_PER_US 20

 5  clock_t clock(void);
 6  clock_t cputime;

 7  xthread_t xidfoo;

 8  foo()
 9  {
10  }

11  xmain()
12  {
13     int i,COUNT;

14     scanf("%d",&i);
15     COUNT = i;

16     clock();

17     for(; i>=0 ; i--) {
18        xthread_create(&xidfoo,NULL,foo,0);
19        xthread_yield(xidfoo);
20     }

21     cputime = clock();
22     printf("Creation Time = %f  us\n",
23             (float)cputime/(float)COUNT/(float)CPU_PER_US);
24  }
```

Figure 15: Measuring the creation and destruction cost in Xthreads

```
 1  #include <stdio.h>
 2  #include <time.h>

 3  #include <xthreads.h>

 4  #define CPU_PER_US 20

 5  clock_t clock(void);
 6  clock_t cputime;  /* the type of clock_t is long long */

 7  xthread_t xidfoo, xidmain;

 8  int i;

 9  foo()
10  {
11     while(--i>0)
12        xthread_yield(xidmain);
13  }

14  xmain()
15  {
16     int COUNT;

17     scanf("%d",&i);
18     COUNT = i;

19     xthread_self(&xidmain);
20     xthread_create(&xidfoo,NULL,foo,0);

21     clock();

22     while(--i > 0) {
23        xthread_yield(xidfoo);
24     }

25     cputime = clock();

26     printf("CTXSW Time = %f  us\n",
27             (float)cputime/(float)COUNT/(float)CPU_PER_US);
28  }
```

Figure 16: Measuring the context switching cost in Xthreads

thread creation, synchronization, and context switching are efficient in terms of performance. Because of its layered design philosophy, the Xthreads library has been easily ported to three different machines within a short period of time. Further, because of its modularity, which unambiguously defines interfaces to the library's functional components, new scheduling disciplines are readily implemented and then incorporated into lightweight-process simulation systems, requiring only a simple need to match interfaces.

To build a thread library on a different machine, we address the following two issues:

- **the calling conventions.** This involves the parameters passing method between functions, the parameters positions on the stack, and the non-volatile registers which are required to save.

- **the stack alignment.** For example, the iPSC860 requires the stack to be aligned on 16-byte boundaries to keep data arrays aligned. Failure to keep this alignment would not cause system crashed, but would raise exceptional conditions which increase the execution time tremendously.

The early measurements show that the thread context switching cost can be only about one-third of the process switching cost and the thread creation latency can be even less than 0.1% of the cost of process fork on nCUBE2 machine.

## References


[1] C. Binding. Cheap Concurrency in C. *SIGPLAN Notices*, 20:21–26, September 1985.

[2] D. Comer. *Operating System Design The XINU Approach.* Prentice-Hall, 1984.

[3] T. W. Doeppner Jr. Threads - a system for the support of concurrent programming. Technical Report CS-87-11, Computer Sciences Department, Brown University, 1987.

[4] IBM AIX version 3 RISC System/6000. *Assembler Language Reference*, March 1990.

[5] Intel Corporation. *The i860 Microprocessor Family Programmer's Reference Manual*, 1992.

[6] B. M. March, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class User-level Threads. In *Symposium on Operating System Principles*, 1991.

[7] F. Mueller. A Library Implementation of POSIX Threads under UNIX. In *Proceedings of the Winter USENIX Conference*, 1993.

[8] J. Sang, K. Chung, and V. Rego. Si: A simulation package based on lightweight processes. Technical report, Computer Sciences Department, Purdue University, Spetember 1992.

[9] H. D. Schwetman. CSIM: A C-based process-oriented simulation language. In *Proceedings of the 1986 Winter Simulation Conference*, pages 387–396, 1986.