Processes, Channels, and Semaphores

(Version 2)

Jeffrey Mock
Pixar

ABSTRACT

An interface to the concurrency features of the transputer is described. The interface is a library of C functions that allow the user to implement multiple processes, interprocess communication through message passing or shared data, and semaphores. This second version of the software implements several new functions and re-implements several existing functions to fix a few bugs and enhance performance. The new functions provide better control over internal memory usage and provide a means of establishing reliable communication between processors.

Introduction

The transputer has a large set of instructions for implementing concurrent systems. The set includes instructions to start, stop, and pass messages between processes. The hardware includes features to block and restart processes waiting on communication and to select a new process after a predetermined timeslice has elapsed, placing the timesliced process at the end of an active process queue.

The subroutine library described below allows full access to these concurrency features. The package is intended as the lowest level interface in a concurrent system. Higher level packages can be built using the described routines.

Concurrency

Before a new process can be executed, a stack frame must be allocated for it. The transputer contains a single 32 bit linear address space that all processes execute within. Stack space for a process is allocated out of this space using malloc(). After allocating the space, the stack frame is initialized to a valid state for the process to begin executing. A new process is allocated as follows:

```
#include <con.h>
Process *ProcAlloc(func, sp, nparam, p1, p2, ..., pn)
    int (*func)();
    int sp;
    int nparam;
```
ProcAlloc() takes a pointer a function that contains the code for the process. The parameter sp indicates the amount of stack space required for the process, a value of zero for this parameter causes the routine to allocate DEFAULTWSSIZE bytes (64K) of stack space. nparam specifies the number of words of space to allocate off the stack space initially for parameters to the function.

On successful completion, ProcAlloc() returns a pointer to the structure that constitutes the process. If ProcAlloc() fails for some reason or it is unable to allocate stack space, it returns a null pointer. If a small stack space is requested and it is not sufficient to produce the initial stack frame with the requested parameters, the stack size will be increased appropriately.

ProcAlloc() uses malloc() to allocate the space for the Process structure and the stack space for the process. If it is desirable to manage the allocation of this space in the application program, the lower level routine ProcInit() can be used:

```
#include <con.h>
ProcAlloc(p, func, ws, wssize, nparam, p1, p2, ..., pn)
Process *p;
int (*func)();
char *ws;
int wssize;
int nparam;
```

ProcInit() takes a pointer to a Process structure (p) and a pointer to the stack space to be utilized (ws) and initializes the process structure and workspace according to the passed parameters func, wssize, and the parameters for the process. ProcInit() is the lower level routine used by ProcAlloc(). ProcInit() does not have a return value, it initializes the passed process pointer and workspace.

Once a process is allocated or initialized, the parameters can be altered:

```
#include <con.h>
ProcParam(p, param1, param2, ..., paramN)
Process *p;
```

There are several routines for executing processes. They are listed below:

```
#include <con.h>
ProcRun(p)
Process *p;
ProcRunHigh(p)
Process *p;
ProcRunLow(p)
Process *p;
ProcPar(p1, p2, p3, ..., pn, 0)
Process *p1, *p2, ..., *pn;
```
ProcParList(plist)
    Process **plist;

ProcPriPar(phigh, plow)
    Process *phigh, *plow;

ProcRun(), ProcRunHigh(), and ProcRunLow() execute unsynchronized processes. The process begins execution and is out of the control of the initiating process. The initiating process has no means for determining or altering the state of the created process except through a communication means the user explicitly establishes. ProcRun() executes the process at the priority of the current process, ProcRunHigh() executes the process at high priority and ProcRunLow() executes the process at low priority. Refer to transputer technical documentation for a description of the differences between high and low priority processes.

ProcPar(), ProcParList(), and ProcPriPar() start a group of processes. Control is returned to the initiating process when all of the initiated processes terminate. ProcPar() takes an explicit null terminated list of processes, all of the processes are executed at the current priority. ProcParList() takes a null terminated array of pointers to processes, all of the processes are executed at the priority of the current process. ProcPriPar() takes two parameters. The first process is executed at high priority and the second is executed at low priority. As with the other par functions, ProcPriPar() returns when both processes complete.

Process Details

The first parameter to a newly executing process is a pointer to its own process structure, the second parameter is the first parameter assigned by the last call to ProcAlloc() or ProcParam(). An example of process creation and instantiation is shown below:

newprocess(p, apple, pear, banana)
    Process *p;
    { /* do stuff here */ }

main()
    { Process *x; int apple, pear, banana;
      /* Three parameters: apple, pear, and banana */
      x = ProcAlloc(newprocess, 0, 3, apple, pear, banana);
      ProcRun(x);
      /* Do other stuff while newprocess() is running */
    }
Processes share the same global data space. Any statics or externals used by a process are shared by other executing processes. The only private data space to a process are auto variables. Processes also share the same heap, calls to malloc() and free() allocate data from the same heap space.

Interprocess Communication

The transputer supports a message passing protocol for interprocess communication. A channel is a unidirectional message stream between two processes. When a process performs input or output to a channel, the process is blocked until the corresponding process performs its respective output or input. This way, channels can be used as a synchronization mechanism in addition to a communication mechanism. The only caveat in channel operation is that the two process must perform operations of the same data size. If one process attempts to output 50 bytes while the corresponding process attempts in input 49 bytes, an unpredictable operation will result. There are six routines for performing communication along channels:

#include <con.h>
ChanOut (c, cp, cnt)
    Channel *c;
    char    *cp;
    int     cnt;

ChanOutChar(c, ch)
    Channel *c;
    char    ch;

ChanOutInt(c, n)
    Channel *c;
    int     n;

ChanIn(c, cp, cnt)
    Channel *c;
    char    *cp;
    int     cnt;

int
ChanInInt(c)
    Channel *c;

char
CharInChar(c)
    Channel *c;

Channels require initialization before they can be used for communication. The following routines are provided for channel allocation and initialization. ChanReset() resets a channel, returning information that was already contained in the channel. ChanAlloc() returns a pointer to an initialized channel.
The channel concept extends beyond the bounds of a single transputer. The four serial links of a T414 transputer correspond to eight channel pointers (four input, four output) with specific hardware addresses. These addresses are contained in con.h:

```c
/* Addresses for physical links on T4's and T8's */
#define LINK0OUT ((Channel *) 0x80000000)
#define LINK1OUT ((Channel *) 0x80000004)
#define LINK2OUT ((Channel *) 0x80000008)
#define LINK3OUT ((Channel *) 0x8000000c)
#define LINK0IN ((Channel *) 0x80000010)
#define LINK1IN ((Channel *) 0x80000014)
#define LINK2IN ((Channel *) 0x80000018)
#define LINK3IN ((Channel *) 0x8000001c)
```

### Alternation

A series of calls are available to determine the status of channels and possibly wait until a channel is ready for input. There are six alternation routines:

```c
int ProcAlt(cl, c2, ..., cn, 0)
    Channel *cl;
    Channel *c2;
    Channel *cn;

int ProcAltList(clist)
    Channel **clist;

int ProcSkipAlt(cl, c2, ..., cn, 0)
    Channel *cl;
    Channel *c2;
    Channel *cn;

int ProcSkipAltList(clist)
    Channel **clist;
```
int ProcTimerAlt(time, c1, c2, ..., 0)
    Channel *c1;
    Channel *c2;
    Channel *cn;

int ProcTimerAltList(time, clist)
    Channel **clist;

ProcAlt() and ProcAltList() cause the current process to block until one of the channels in its argument list is ready for input. On completion, the routine returns an index into the parameter list for the channel ready for input.

ProcSkipAlt() and ProcSkipAltList() check specified channels. If one of the channels is ready for input, and index into the parameter list is returned, otherwise -1 is returned. These routines do not block waiting for one of the channels, they return immediately.

ProcTimerAlt() and ProcTimerAltList() block the current process until one of the channels is ready for input or the value of the clock is after the time parameter. If the routine times out, a -1 is returned, otherwise an index into the parameter list is returned.

ProcAlt(), ProcSkipAlt() and ProcTimerAlt() take an explicit null terminated list of pointers to channels as parameters. ProcAltList(), ProcSkipAltList(), and ProcTimerAltList() take a null terminated array of pointers to channels as a parameter.
Semaphores

The library provides a semaphore facility. It's use is discouraged, the routines are provided to make stdio functions behave reasonably in a concurrent environment. Transputers provide no explicit support for semaphores although they can be efficiently implemented using the concurrency instructions of the transputer.

An initialized semaphore can be created two ways:

```c
#include <con.h>
Semaphore sem = SEMAPHOREINIT;

/* or */
Semaphore *newsem;
newsem = SemAlloc();
```

A semaphore is acquired like this:

```c
#include <con.h>
SemP(sem)
    Semaphore sem;
```

Semaphores are released like this:

```c
#include <con.h>
SemV(sem)
    Semaphore sem;
```

Note that both SemP() and SemV() take the semaphore as a parameter rather than a pointer to the semaphore. This is different than both channels and processes. SemP() blocks the current process and places it on a queue if the semaphore is in use, otherwise it sets the semaphore to acquired and execution continues. The routine will not return until the process successfully acquires the semaphore. SemV() releases the semaphore and runs the first process on the queue if any processes are waiting.

Semaphores are used in libc to make it behave reasonably in a concurrent environment. For instance, malloc(), free(), and realloc() have been modified to use semaphores to prevent the heap from being corrupted when multiple processes perform mallocs simultaneously. Each file descriptor also possesses a semaphore.

Miscellaneous

The value of the clock can be obtained with the Time() function. Time() is an atomic function. The clock is different for low and high priority processes. The low priority clock is incremented every 64 uS, the high priority clock is incremented every 1 uS. Execution can be blocked until a specified time with the ProcAfter(). Execution can be suspended for a specified number of clock periods using ProcWait(). If it is necessary for a process to 'busy wait' on a resource, the process can be placed at the end of the active process queue by calling ProcReschedule(). A process can determine its
priority by calling ProcGetPriority(). This routine returns 1 for low priority processes and 0 for high priority processes. ProcGetPriority() is an atomic operation. A process can be descheduled using ProcStop(). Under normal circumstances, the process will never be rescheduled after ProcStop() is called. These miscellaneous functions are listed below:

```c
int
Time()

ProcAfter(time)
   int time;

ProcWait(time)
   int time;

ProcReschedule()

int
ProcGetPriority()

ProcStop()
```

### Reliable Communication

On occasion it may be necessary to attempt communication along a channel that may be inoperative. The standard routines ChanIn() and ChanOut() are inadequate for this task. The standard routines cause the initiating process to stall until the communication completes. If the communication never completes due to a faulty connection, the initiating processes will never continue.

Four routines are provided:

```c
int
ChanOutTimeFail(chan, cp, cnt, time)
   Channel *chan;
   char    *cp;
   int     cnt;
   int     time;

int
ChanOutChanFail(chan, cp, cnt, failchan)
   Channel *chan;
   char    *cp;
   int     cnt;
   Channel *failchan;

int
ChanInTimeFail(chan, cp, cnt, time)
   Channel *chan;
   char    *cp;
   int     cnt;
   int     time;
```
int ChanInChanFail(chan, cp, cnt, failchan)
  Channel *chan;
  char    *cp;
  int     cnt;
  Channel *failchan;

The first three parameters to each of these routines have the same semantics as the
routines ChanIn() and ChanOut(). These routines contain an additional parameter that
allows the process to reschedule based on a terminating condition. In the case of
ChanInTimeFail() and ChanOutTimeFail(), communication is attempted until the
clock reaches the value of the time parameter to the routine. If communication
completes before the timeout, the routines return a status of 0. If communication has not
completed when the timeout occurs, the offending channel is reset and a status of 1 is
returned from the function.

ChanInChanFail() and ChanOutChanFail() provide a similar facility, but
instead of aborting communication when a certain time value is reached, communication
is aborted when the parameter channel failchan is ready for input. This way, the
decision to abort communication can be made by another process. As with the other
routines, a return value of 0 indicates communication completed normally, a return value
of 1 indicates that the communication was aborted by a message sent from another
process. When communication is aborted, the channel is reset to prevent communication
from completing at a later time.

These routines offer somewhat more overhead than their ChanIn() and
ChanOut() counterparts. They should be used to establish the integrity of the link
between two unfamiliar processors and not as the standard method of communicating
between two processors. When a failure occurs, the problem of re-establishing reliable
communication is a complex one not addressed by these routines.

**Internal Memory and Stack Frames**

The single most important optimization to improve the performance of code
running on transputers is to place the stack frame of compute intensive code in the
internal memory of the transputer. To aid in this process, a routine is provided to call a
subroutine using an indicated area of memory as the stack space:

```c
int ProcCall(func, ws, wssize, p1, p2, p3, p4, p5)
  int (*func)();
  char    *ws;
  int     wssize;

ProcCall() first builds a stack frame in the array ws of size wssize bytes. The
stack frame is initialized with up to five parameters. After building the stack frame, the
routine is called and the value returned by ProcCall() is the value returned by func.
An example of its use is shown below:
#define SIZE 512
internal char stack[SIZE];
extern int compute();

main()
{
    ProcCall(compute, stack, SIZE);
}

In this case, the routine compute() is called with no parameters. The routine is executed with its stack stored in 512 bytes of internal memory. It's important that the routine compute() not exceed a maximum stack depth of 512 bytes. C programs show a typical 40% speed improvement by placing the stack in internal memory. The overhead of ProcCall() is about 5x the overhead of a standard procedure call.