The Plan 9 Front Concurrent C Extensions

Benjamin Purcell (spew)
benjapurcell@gmail.com

ABSTRACT

The Plan 9 Front compilers extend the C programming language to provide built-in CSP style concurrency operations. This paper describes the usage and implementation of the extension.

1. Introduction and Motivation

CSP-style concurrency operations are an essential part of many programs in the Plan 9 operating system. Concurrent programs were originally written in Alef which had built-in concurrency operations. When Alef was retired, a library was written to allow access to CSP operations from programs written in C (see *thread*(2)). However, there are a number of deficiencies with *thread*(2); thread creation is inflexible, receiving or sending of multiple channels requires an awkward definition of an array of structures, and the send/receive operations are not type-safe. The extension aims to address those concerns to make threaded programs easier and safer to write without the need to maintain a separate compililer infrastructure such as Alef. This document assumes familiarity with *thread*(2).

2. The Extensions

The compiler extension provides for launching new threads and processes, declaring and allocating storage for typed channels, and type-safe sending and receiving from channels. It also provides a new control structure for type-safe sending or receiving of multiple channels.

2.1. Thread and Process Creation

Threads and processes are created using the keywords `coproc` and `cothread` which has the syntax of a function that takes two arguments. The first argument a function application, and the second is an unsigned int that specifies the stack size for the process or thread. The calls `coproc` and `cothread` return the resultant thread id.

```
int tid, pid
void fn(int arg1, double arg2, char *arg3);
...
tid = cothread(fn(a, b, c), 8192);
pid = coproc(fn(a, b, c), 8192);
```

The function passed to `coproc` and `cothread` can have any signature, though its return value will not be used. Instead of applying the function to its arguments, the calls of `cothread` and `coproc` tell the compiler to check the arguments to the function and then compile a call into *thread*(2) to start the function in a new thread or process with a memory allocated stack (see *malloc*(2)). Due to the type-checking, if `a`, `b`, and `c`, are of an incompatible type to `int`, `double`, and `char*` respectively, then the example above will not compile.

2.2. Channel Declarations

The extension reserves the character @ for declarations of typed Channels. A typed channel has a type associated with it; only values of that type may be sent or received from the channel. The @ symbol has the same precedence as the pointer dereference * and functions similarly. Thus

```
int @c;
```

declares c to be a channel for sending/receiving an int;

```
char *@c;
```

declares c to be a channel for sending/receiving a pointer to a char; and

```
int *(*@c[3])(int);
```

declares c to be an array of three channels for sending/receiving pointers to functions that take an int and return a pointer to an int.

The @ symbol can be viewed as a special kind of dereference. It represents the value in the channel.

2.3. Channel Allocation

Once a channel is declared, it must be configured for use by applying the compiler extension `chanset` to the channel. The usage is

```
int @c;
chanset(c, nelem);
```

Where `nelem` is an int that sets the number of values the channel can hold and whether the channel is buffered or unbuffered. See `chancreate` in *thread*(2).

2.4. Channel Operations

The compiler extensions allows for sending into and receiving typed values from channels. The syntax for receiving a channel mimics that of channel declarations. That is, for a channel for sending ints and an int as follows:

```
int @c, i;
```

the statement

```
i = @c;
```

receives an int from the channel `c` and assigns the value to `i`. This can be thought of as a kind of dereference that first calls into the *thread*(2) library in order to have the value available in the channel.

A new binary operator @= is used to send into a channel. The left-hand side of the expression must be a channel and the right-hand side's type must match the type of the values the channel is allowed to send/receive. Thus given

```
char *@c;
```

the statement

```
c @= "hello, world";
```

completes a send into the channel. The statement always evaluates to an int: 1 on succes and -1 if the send was interrupted.

2.5. Sending/Receiving of Multiple Channels

Channel sending/receiving may be multiplexed on a single statement using a new control flow statement called the alt-switch. It is similar to a switch with the expression value replaced by an @ character and the `case` keywords replaced by a new extension keyword `alt`. Instead of constant expressions, each `alt` is labeled by potential channel sends or receives. An optional default label handles the case where the underlying `doalt` operation (see *thread*(2)) is interrupted.

```
int @ichan, @req, i;
char *@schan, *s;

s = "hello";
switch @{
alt i = @ichan:
     print("%d\n", i);
     ...
     break;
alt @req:
     ...
     break;
alt schan @= s:
     print("Sent hello\n");
     ...
     break;
default:
     print("Interrupted!\n");
}
```

In the example above three potential channel operations are "multiplexed" on one alt-switch statement. Either an int is received from `@ichan` and assigned to `i`, an int is received from `@req` and its value thrown away, or the string `"hello"` is sent into `schan`. The operations are multiplexed in the sense that if at least one of those channel operations can proceed, one is chosen at random to be executed and control flow proceeds after the corresponding `alt` label. Otherwise the alt-switch statement blocks until one of the operations can proceed.

A non-blocking alt-switch statement is specified by using two @ symbols:

```
switch @@{
     ...
default:
     print("No channel operations can proceed.\n");
}
```

In this case, the statement does not block if no channel operations can proceed, but immediately continues execution at the default label. If a non-blocking alt-switch is interrupted while in the middle of executing a valid channel operation, then the alt-switch will continue execution at a `case` labeled by `-1`.

The channel send operation in an `alt` label is more restricted than an ordinary channel send in the sense that the right hand side of the @= binary operator must be addressable. Thus

```
alt ichan @= 5:
```

will not compile.

3. Summary of the Extension

In total the extension reserves the following new keywords or symbols

```
@ alt  chanset  cothread  coproc
```

and defines the following new expressions or statements:

Usage Summary	
Channel Operations	
`chanset(chan, nelem)`	Allocates and readies a channel
`chan @= val`	Channel Send
`@chan`	Channel dereference (receive)
Alt-Switch	
`switch @{...}`	Blocking alt-switch
`switch @@{...}`	Non-blocking alt-switch
`alt val = @chan:`	Alt label (receive)
`alt @chan:`	Alt label (receive, value thrown away)
`alt chan @= val:`	Alt label (send)
Thread Creation	
`coproc(fn(...), stksize)`	Starts a process in its own stack
`cothread(fn(...), stksize)`	Starts a thread in its own stack

Figure 1. Summary of compiler extensions and usage. `chan` denotes a typed channel and `val` is of the channel's sending type. `nelem` is an int, `fn` is a function of any signature, and `stksize` is an unsigned int.

4. Implementation

See /sys/src/cmd/cc/thread.c (I will actually write something here soon I promise).