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 to `cothread` and `coproc` tell the compiler to check the arguments to the function and then compile a call to thread to start the function in a new thread or process with a memory allocated stack (see *malloc*(2)). Thus, if `a`, `b`, and `c`, are of an incompatible type to `int`, `double`, and `char*` respectively, then the 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 de-reference * 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 de-reference. 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 de-reference that first calls into the *thread*(2) library in order to retrieve have the value available in the channel.