Quick Guide to Using sunfire for CSEP 524 Students

Matthew Kehrt

1 Introduction

As a student in CSEP 524, you have access to a Sunfire T2000 for writing programming assignments on. This is entirely optional, but there’s no reason not to take advantage of it. The Sunfire T2000 is a SPARC machine running Solaris 10 with 8 cores.

2 Connecting to sunfire

The machine is named sunfire. You should have received an email specifying your username and password for it. This was sent to your u.washington.edu address; if for some reason you did not receive this email, please contact me at mkehrt@cs.washington.edu and I will send you this information again.

You will need to connect to sunfire over ssh. If you have a Mac or a Linux machine, you should be able to use ssh from a terminal. If you have a Windows machine, you will need to download an ssh client. I recommend using PuTTY, which is available at http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html.

The fully qualified hostname for sunfire is sunfire.cs.washington.edu. If you are connecting to the machine from outside the UW CSE network, you will need to use this name.

To connect to sunfire over ssh from a command line (Mac or Linux), type

```
ssh <username>@sunfire.cs.washington.edu
```

You will be prompted for your password.

For users of PuTTY (Windows), you should be able to enter the hostname you wish to connect to, your username and your password from a dialog box and save these values for later use.

In both cases, the first time you connect to any machine, it will probably give a warning that the host key is unknown, and ask if you want to add it to your local database. Say yes to this.

3 Using sunfire

sunfire is a Unix machine. If you have never used one before, explaining simple Unix usage is beyond the scope of this document. However, the Internet is full of tutorials. One that looks fairly
good is available at http://www.ee.surrey.ac.uk/Teaching/Unix/.

For those of you who are used to Linux, Solaris may offer some unexpected surprises. These will mainly come in the form of command line switches not doing what you expect. Moreover, many programs expect all command line switches to come before other arguments. When in doubt, read the man pages!

The first thing that you should do on logging into sunfire is to set your password. Do this by running passwd. It will prompt you for your existing password and then ask you to enter and then to confirm a new password.

Currently, C is the only language installed and working on sunfire. I hope to have C++ and Java working soon. make should also be working.

4 pthreads Introduction

To write threaded code in C, you will need to use the pthreads library. This can be done by including pthreads.h.

The most basic pthreads function is pthread_create, which is used to create a new thread. It has the following signature:

```c
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, 
void *(*start_routine)(void*), void *arg);
```

pthread_create takes four arguments. The first is a pointer to a pthread_t, into which pthread_create will put a descriptor used to manipulate this thread in the future. The memory location this pointer points to will be overwritten and in many cases will be a new pthread_t declared on the stack.

The second argument is a pointer to a pthread_attr_t, which is a struct describing optional attributes for the thread. This should in general be NULL, which will use default attributes.

The next argument is a pointer to the function the thread will actually run. It takes a void* and returns a void*. The return value of this function will be made available by calling pthread_join on the pthread_t of the thread running it (see below for more about this). While the syntax of function pointers in C are notoriously difficult, it suffices to use the name of a function defined elsewhere in your code here.

The final argument is a void* which is the argument passed to the thread when it begins to run.

Finally, pthread_create returns an error code. If no errors occurred, it returns 0.

An example usage of pthread_create follows. Suppose we have defined the following function.

```c
void* go(void* str)
{
    printf("In thread: %s\n", str);
}
```
We then can write the following lines in another C function.

```c
pthread_t t;
char* str = "Hello, world!";

pthread_create(&t, NULL, go, (void*)str);
```

Which will run `go` and print "In thread: Hello world!". However, simply running this code may not actually run the thread. The main thread may exit, exiting the program and not running the thread created by `pthread_create`. Moreover, we as yet have no way of getting the return value of `go`. We can solve both of these problems with `pthread_join`.

`pthread_join` waits for a thread to return and gets its return value, which is defined as being either the return value of the function that the thread was started with, or a value the thread passed to `pthread_exit` (discussed below). The signature of `pthread_join` follows.

```c
int pthread_join(pthread_t thread, void **value_ptr);
```

`pthread_join` takes two arguments. The first is the thread descriptor of the thread to wait on. Recall that this was set by `pthread_create` above. The second is a pointer to a `void*` to write the return value of the thread into. Finally, `pthread_join` returns an error code, 0 on success.

For example, the following code defines a function which, when run in a thread, returns the value returned by `printf`.

```c
#define GO (void*) go(void* str)
{
  int ret;
  ret = printf("In thread: %s\n", str);
  return ret;
}
```

This can be used with the following code to spawn a thread and then wait for the return value of `go`.

```c
pthread_t t;
char* str = "Hello, world!";
int ret;

pthread_create(&t, NULL, go, (void*)str);
pthread_join(t, (void**)&ret);
```

Finally, two other functions must be mentioned. The first is `pthread_exit`. This is used to prematurely exit a thread with a given return value. The signature of `pthread_exit` is

```c
void pthread_exit(void *value_ptr);
```
This takes a `void*` as a return value and exits the thread it was called from. It does not return.

Finally, to allow other threads to join on an exited thread, some information about the exited thread must be kept around after the thread has exited. If you do not join on the thread, this information is kept around indefinitely. To prevent this from happening, you must use `pthread_detach` to free this information. This will make joining on the thread impossible, but will prevent this space leak. `pthread_detach` has the following signature.

```c
int pthread_detach(pthread_t thread);
```

It is called on a thread descriptor. It returns an error code which is 0 on success.

More information about pthreads is available online. A simple tutorial is at http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html