Context Switching and Scheduling in C--

Shawn Hsiao

Olushola Olateju

May 11, 2000

Abstract

We present a framework for preemptive threads in C--. We describe the front- and back-end runtime system support requirements of our model. Our model extends the current C-- framework with new services. In particular, we suggest two extensions to the back-end runtime interface: a mechanism for saving the state of a suspended execution and a revised mechanism for resuming the state of a suspended execution. We demonstrate the plausibility of our model for the simplest case and conjecture on its extensibility to the general case.

1.0 Introduction

Most high level programming languages available today offer some notions of concurrency. In order to support concurrency in C--, the level of front- and back-end support must be determined carefully. The guiding principle is to have C-- provide the mechanisms and let the higher level programming languages use these mechanisms in their own runtime system to implement different policies. We are interested in exploring the exact mechanisms that should be provided in C-- for high level programming languages to implement concurrency, especially threads. In a language like C, threads are created by function calls and giving entry points – which are other functions. In a language like Newsqueak [7][8], threads are created by statements; it similarly uses functions as an entry point. These higher level policies should not be limited by C--'s mechanism. For example, C-- should not limit the thread creation interface to functions only. A thread system could be either preemptive or non-preemptive. A thread in a preemptive thread system may be interrupted at periodic intervals and designated points, at which point control is transferred to another thread by the scheduler. In a non-preemptive system, a change in control occurs only when the currently executing thread relinquishes control to the scheduler or to another named thread.

This paper presents a runtime framework for preemptive threads in C--. In particular, we will explore the capacity of the current C-- back-end to support our preemptive thread model and provide modifications to the current back-end runtime system or suggest new services where necessary. The thread framework presented in this paper is largely based on the POSIX thread model. The current C-- back-end specification suggests a partial framework for non-preemptive threads. We demonstrate the plausibility of our framework with a simple model. We conjecture that proving the simplest case – two C-- threads multiplexed within an OS thread – implies the plausibility of extending the simple model into a more general one.

C-- [3][4][5][6] is a portable assembly language that provides an abstraction of hardware. The primary motivation behind C-- is to enable a front-end runtime system to implement high level services while using C-- as a code generator. The C-- framework separates policy from mechanism. The front-end runtime system writer determines the policy, while the back-end runtime system provides the mechanism. C-- employs a runtime interface that allows the inspection or modification of the state of a suspended execution. In addition, C-- is desirable because:

- It does not exhibit some of the deficiencies of C as an assembly language; in particular the difficulty of implementing tail-call optimizations and the unknown stack frame layout (since a C compiler can layout its stack frames as it pleases)
- It can perform tasks that can not be done in C e.g. garbage collection with registers
- It allows the front-end to determine the cost model
- It reduces and could, perhaps, eliminate the need for virtual machines

Some of the more interesting C-- features include:

- It supports only the minimum machine data types. Since all hardware may not support all data types, it is important for the front-end to know beforehand the data types supported by the underlying hardware
- Variables are abstractions of registers – when a variable of some type is declared, it could potentially fit in a register
- C-- attempts to put all variables in registers; if there are no registers available, the variable is stored in memory
- C-- supports procedures with multiple return values and general tail calls

The rest of this paper is structured as follows; we discuss our primary underlying assumptions in section 2 and present the front- and back-end services in sections 3 and 4. We give a demonstration of the basic system in section 5, discuss some outstanding issues in section 6, and conclude with a quick summary in section 7.

2.0 Assumptions

Before getting into the details of our model, it is necessary to clarify our assumptions about the services provided by our front-end framework and the back-end C-- runtime interface.

The Back-end runtime system maintains the thread local state*.* The thread local state (hereon referred to as the thread control block, TCB) holds information pertinent to a thread. The front-end and back-end have different views of a thread. The front-end is primarily concerned with how to context switch and schedule a thread. However, the front end is not concerned with information regarding the layout of the thread stacks and registers. The front-end TCB holds a pointer to the back-end TCB and some other data structures that could affect the thread policy like the thread priority, age, name, etc. The thread creation process typically involves the front-end allocating a chunk of memory – some of which it keeps for itself – and the rest is passed to the back-end runtime system. The back-end TCB structure is responsible for maintaining the layout of the memory passed to it. The layout would typically include the thread stacks, registers, and activation records. Since, we are not concerned with the back-end's view of a thread we assume that one exists. The following code represents a simple front-end TCB structure:

```
/* thread control block for front-end runtime */
struct front_end_tcb {
      struct tcb back_end_tcb;  /* C-- runtime tcb */
      int priority;
      int age;
         :
         :
};
```

All C-- threads are polite. Most system calls, whether on Unix or other platforms, block (or suspend) the calling threads until they complete, and continue its execution immediately following the call. Some systems also provide asynchronous (or *non-blocking*) forms of these calls; the kernel notifies the caller through some kind of out-of-band method when such a system call has completed. Asynchronous system calls are generally much harder for the programmer to deal with than blocking calls. Our primary objective does not include the provision of a robust thread system that is resilient against malicious threads. We therefore assume that threads will not block the whole system by blocking signals or executing system calls that will block indefinitely.

3.0 Front-end Services

Having stated the assumption we made in our thread framework, we now focus our attention on the services provided by the front-end runtime system.

3.1 Operating System Support

In order to perform a context switch, we need the operating system to provide support that periodically interrupts the currently executing thread. These periodic interrupts give the scheduler a chance to schedule other threads. Unix defines a set of signals for software and hardware conditions that may arise during the normal execution of a program. The signal facilities found in SVR4 and 4.4BSD were designed around a virtual-machine model, in which system calls are considered to be the parallel of machine's hardware instruction set. Signals are the software equivalents of traps or interrupts, and signal-handling routines perform the equivalent function of interrupt or trap service routines. By default, signals are delivered on the runtime stack of the process. However, 4.4BSD and SVR4 provide a mechanism for the delivery of signals on an alternative stack – through the `sigaltstack` call. Our framework adopts the 4.4BSD model – setting the delivery of signals on a separate stack. When a C-- program is launched, it needs to call the front-end runtime system to setup the thread system. These setup requires the initialization of several components; period timer, signal mask, and the data structure that will hold each thread's status:

```
int Thread_init() {
      struct itimerval period;
      struct sigaction sa;
      /* set up interrupts interval and period */
      period.it_value.tv_sec  = 0;
      period.it_value.tv_usec = 100000;
      period.it_internal.tv_sec     = 0;
      period.it_internal.tv_usec    = 100000;

      /* setup alternative stack */
      sa.sa_flags = SA_ONSTACK;
      /* interrupt handler */
      sa.sa_handler = interrupt_handler;
      if (sigaction(SIGVTALRM, &sa, NULL) < 0) {
            /* error checking code */
            return 0;
      }
      /* Setup signal delivery on separate stack */
      sigaltstack(&old, &new);
      setitimer(ITIMER_VIRTUAL, &period, NULL);
            :
}
```

The `it_value` field in an `itimerval` structure specifies the amount of time in seconds (`tv_sec`) and microseconds (`tv_usec`) to the next timer interrupt. The values in the `it_interval` field are used to reset the `it_value` field when the timer expires. `Thread_init` arranges for the timer interrupt to occur every 100ms.

Our model uses a VIRTUAL timer as compared to a REAL timer. The VIRTUAL timer decrements only when the process is running. We are not sure if this is a better solution because if a thread goes sleeps indefinitely, the whole process will be blocked (see section 6.1 for a discussion on dealing I/O and system calls that may block). On the other hand, the REAL timer decrements in real-time, even when the process is not running. This might be a problem when the system is overloaded and the process gets a small fraction of the CPU. In POSIX, system calls interrupted by a signal cause the call to be terminated prematurely and an "interrupted system call" error to be returned. The model requires that the thread programmer carefully examine the code returned from a system call to ensure it was completed and restart it if necessary. 4.4BSD provides a better mechanism through the `sigaction` system call, which can be passed a flag that requests that system calls interrupted by a signal be restarted automatically wherever possible and reasonable. We felt the responsibility of ensuring the completion of interrupted system calls may be rather cumbersome for the thread programmer.

An alternative way to setup timer interrupts is using the `alarm()` system call. However, this is made obsolete by `setitimer` and only works with second granularity. On most Unix system, the default timer granularity is 10ms. A finer granularity can be achieved by modifying the kernel.

3.2 Interrupt Handler

The interrupt handler is called when a periodic timer expires. The semantics of signal delivery on 4.4BSD blocks future occurrence of the same signal. This reduces the complexity of the interrupt handler but also introduces the chance that a process may be blocked during the interrupt handler. The semantics may be different on other platforms. We can unblock the signal by calling `sigmask()` – this is done in the scheduler. The interrupt handler has two primary responsibilities: ensuring it safe to interrupt the current executing thread, and invoking the scheduler. It is necessary for the interrupt handler to be aware of atomic thread operations. For example, a thread should not be interrupted in the middle of a memory allocation operation. Our framework adopts the critical-sections approach described in Cormack's micro kernel [1]. When the current running thread enters a critical region, it sets a global static variable `__CRITICAL__`. The interrupt handler checks the value of `__CRITICAL__`, and if it is set, returns immediately and thus ignores the current timer interrupt. It is safe to multiplex the current thread if it is not in a critical region. The interrupt handler then invokes the scheduler. A primary drawback of using critical sections is the performance implications when the critical sections are too long or if a page fault occurs while a thread is in a critical section. We discuss some of these issues in section 6.0. The code for the interrupt handler should look like:

```
extern int __CRITICAL__;

void interrupt_handler(int sig, int code, struct sigcontext *scp) {
/* check if the thread is in a critical section */
if (__CRITICAL__) { return; }
scheduler(scp);
}
```

The `sig` argument carries the signal number, and `code` supplies additional data for some signals. The `scp` argument is a pointer to a `sigcontext` structure that captures the current state of the machine.

3.3 Scheduler

The scheduler determines the next thread to run. The scheduler runs on the interrupt handler's stack. For the simplest case, we assume a single run queue that contains runnable threads. The scheduler first takes a snapshot of the current thread using the back-end service `updateTCB`. It is necessary to take a snapshot of the current thread to ensure that the thread can be resumed at the exact point where it was interrupted. The scheduler then enqueues the interrupted thread in the run queue. The thread at the head of the run-queue is the selected as the next runnable thread and run with a call to `swtch`:

```
struct front_end_tcb *current_thread;

void scheduler(struct sigcontext *scp) {
      __CRITICAL__ = 1;
      updateTCB(current_thread->back_end_tcb, scp);
      runQueue->enqueue(current_thread);
      current_thread = runQueue->dequeue();
      sigsetmask(scp->sc_mask);           /* unblocks signal */
      __CRITICAL__ = 0;                   /* leaving scheduler */
      swtch(current_thread->back_end_tcb);
}
```

The semantics details of `runQueue->enqueue()` and `runQueue->dequeue()` are hidden in this example code. The scheduling policy may be implemented with different techniques. Different policies may require other parameters to describe a thread; these parameters should be stored in the front-end TCB. In our front-end TCB, we include thread priority and age parameters simply to demonstrate the ease of extending the scheduler for different policies.

4.0 Back-end Services

Our preemptive thread model depends on some services provided by the C-- back-end. In particular, we expect the C-- back-end to provide services for creating a thread, updating the state of a thread execution, and resuming the state of a suspended execution. The current C-- back-end specification provides `InitTCB(tcb *t, int size, program_counter pc)` and `Resume(tcb *t)` for thread creation and resumption respectively. However, the present back-end specification does not provide services for saving the state of an execution. We suggest the addition of such a service to the back-end interface and describe its framework. We further suggest modifications to the `Resume(tcb *t)` service, which resolves a potential stack overflow problem.

4.1 Updating the thread state

It is necessary to save the state of a suspended execution so we can resume to it later. In most cases, the front-end runtime system has up to date information about the machine states while the back-end runtime system may not. However, only the back-end runtime system has the knowledge on how to deal with some thread state information like stacks and registers. The current C-- back-end interface does not provide a clear mechanism for saving the state of the current execution. For this reason, we propose extending the back-end interface with the addition of a function that can be used to save the state of a suspended thread. `updateTCB(tcb *t, struct sigcontext *scp)` updates the state of a suspended execution. The parameters to `updateTCB` are a pointer to the back-end TCB of the currently interrupted thread and a pointer to the `sigcontext` structure. The `sigcontext` structure is the kernel's view of the current machine state. This structure contains, amongst other things, the program counter, stack pointer, and frame pointer. From our viewpoint, the proposed `updateTCB` service provide a cheap and easy way for the front-end runtime to communicate up to date machine state information to the back-end runtime. On platforms that do not support `sigcontext`, some other notion of machine states may exist. However, it may be impossible to implement user-level threads on platforms that do not support any notion of machine states from the operating system's view. We feel that this does not represent a deficiency to our model because our primary task it to show the framework works for the simplest case and then generalize it.

4.2 Resuming a thread

The current C-- runtime interface provides the `Resume` service, which is used to restart execution of a stopped thread. `Resume(tcb *t)` returns an integer code that is the result of the yield code returned by the thread when it yields control. A problem with the current `Resume` service is the return value, which mandates stack preservation until `Resume` returns. This problem could quickly result in stack overflow whenever we perform a context switch – it does not matter if signals are delivered on a separate or on the process stack, stack preservation could make stack management a big issue and could potentially overflow the process stack. For this reason, we suggest the addition of a new back-end service `void swtch(tcb *t)`. `swtch` is very similar to `Resume` except its semantics guarantees that it will not return. `swtch` is simply an extension of `Resume`.

5.0 The Basic System

The following state transition diagrams (figure 1a - e) show the execution of our system for the simplest case; two C-- threads (Thread A and Thread B) multiplexed in one OS thread. There are five logical entities in the state transition diagrams, four of them; the interrupt handler, scheduler, Thread A, and Thread B are within the same protection domain; the process. When a software interrupt occurs, the OS takes control from Thread A. If Thread A is not in a critical section (`__CRITICAL__` is not set), the interrupt hander invokes the scheduler, which switches to Thread B. If Thread A is in a critical section, then the interrupt handler returns immediately allowing Thread A to complete its critical section. The OS invokes the interrupt handler.

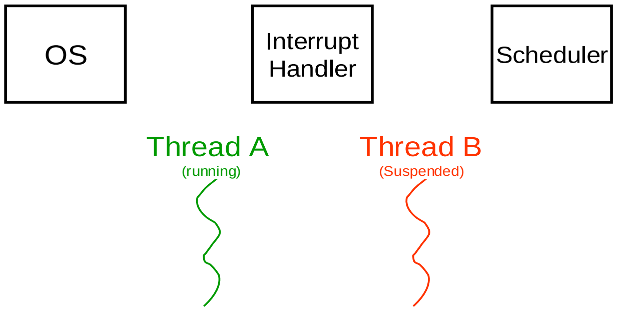

Figure 1a: The initial system state, we assume Thread A and Thread B were successfully created. Thread A is the currently running while Thread B is suspended. Thread A, Thread B, and the OS run on separate stacks. The interrupt handler and scheduler execute on the same stack, which is separate from the OS, and thread stacks.

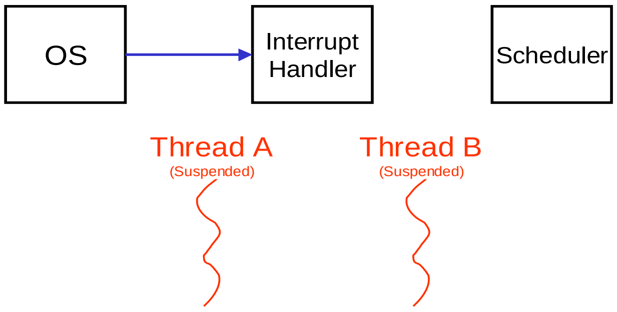

Figure 1b: The periodic timer expires and traps to the OS. Thread A is suspended while the OS calls the interrupt handler. The interrupt handler now determines if is safe to invoke the scheduler.

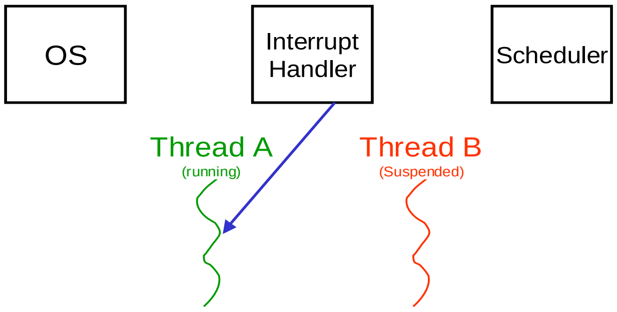

Figure 1c: If Thread A is in a critical then the interrupt handler returns immediately and Thread A resumes execution. The system returns to the state depicted on Figure 1a at the next software interrupt.

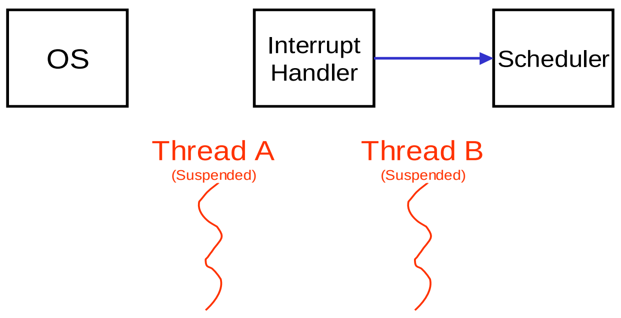

Figure 1d: The interrupt handler invokes the scheduler if it is safe to interrupt Thread A (i.e. `__CRITICAL__` is not set)

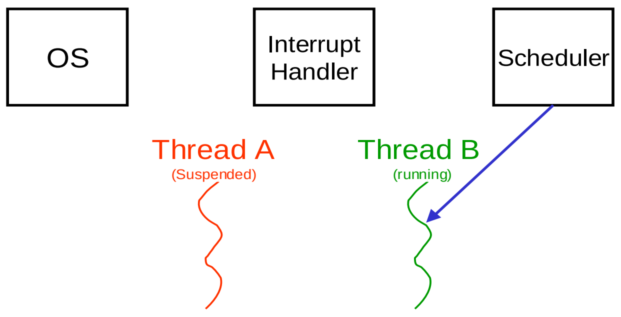

Figure 1e: The scheduler saves Thread A's state, enqueues it and resumes Thread B.

6.0 Discussion and Issues

In this section, we discuss some of the issues we encountered while developing this model and address some of the issues raised in our presentation.

6.1 I/O and Blocking system calls

Our model uses a VIRTUAL timer. It is possible for a thread to block the system by making system calls like `read()` and `write()`, or even `sleep()`. We propose a solution to extend the system and prevent a thread from blocking the system when it is executing system calls.

A monitor could be used to multiplex all system calls that may potentially block the system. Each time a thread wants to make system calls that may block the system, it registers with the monitor, so the monitor can then poll to determine if the operating system is ready to serve the system call, and thus allow the thread to execute the system call.

For example, when a thread wants to make a system call like `read()` to get some data from a file descriptor, it first marks the file descriptor as non-blocking and makes the system call. If the system call will block, it returns immediately. Then the thread registers the file descriptor with the monitor, and goes

to the wait queue. Each time the scheduler is invoked, the monitor will get a chance to check whether the data is ready for the thread, using `select()` or `poll()`. If the data is ready, the corresponding thread will be put back to on the run queue. The next time the thread is scheduled to run, it will be able to read data from the file descriptor.
This solution could resolve the problem that occurs when the system can be blocked by a single thread executing a system call. In addition, this approach increases the processor utilization because the system will be able to schedule other threads to run when one is being blocked.

`sleep()`, on the other hand, might block the system like I/O system calls. The front-end can provide another function call, say `thread_sleep()`, that has the same semantics as `sleep()`, but without entering the operating system kernel. `thread_sleep()` can be implemented as follows: when the function is called, the scheduler puts the thread into a wait queue, and marks when the thread will become ready again. This will not block the system and other threads can be scheduled when the potentially blocking thread is sleeping.

All these system calls can be wrapped within other function calls, so the different mechanisms can be transparent to users.

6.2 Other critical section techniques

We use the global variable `__CRITICAL__` to guard a thread from being interrupted when it is in some atomic operations, for example, allocating memory from the heap. In addition, it is employed to protect the scheduler from being reentrant – to guarantee the integrity of the scheduler.

Code labels is another technique that may be used to protect the current thread from being interrupted while it is doing some atomic operations. For example, we can arrange all functions that require atomic execution be within one continuous memory region, and use two global variables to describe the region, say, MONITOR and END_MONITOR. When a thread is interrupted, the interrupt handler can examine the program counter and see if it falls within the range of the region. If it does, the thread is in an atomic operation and should not be interrupted. All the functions to be put in the region can be implemented by the front-end writer as front-end services. This technique is mentioned in the Cormack paper [1], also the Hanson book [2]. Our interrupt handler can be easily modified to support this synchronization technique as follows :

```
void interrupt_handler(int sig, int code, struct sigcontext *scp) {
/* check if the thread is in a critical section */
      if (scp->sc_pc >= (unsigned long)_MONITOR &&
            scp->sc_pc <= (unsigned long)_ENDMONITOR) {
                return;
      }
```

```
        scheduler(scp);
}
```

If the atomic operation sequences are scattered, or in the form of RAS (Restartable Atomic Sequence), each of these sequences can be registered with the interrupt handler, describing the start and the end of each sequence, so the interrupt handler can store them in a table, and check if the program counter falls in the range of those sequences each time it is called. If the interrupted thread is in a RAS, the interrupt handler is not required to return control to the thread immediately. Instead, it can set the program counter to the start of the sequence, so next time the thread resumes execution, it can start from the correct position.

6.3 Stack Movement

We assume thread stack does not move in the basic system, but it is a rather draconian restriction. What changes are required to the basic system or what should the scheduler do when a thread's stacks are allowed to move? For example, a copying garbage collector may want to move stacks, stacks might overflow or underflow and trigger an overflow/underflow handler that relocates it. It is also possible that the stack is segmented into several small chunks.

The scheduler needs to maintain thread states – it allocates memory regions so the back-end can store thread local states and stacks. We are not sure whether the scheduler should handle the case when a stack overflow or underflow happens. However, the stack information kept by the front-end needs to be updated if any change to the stack takes place.

A simple solution is to have the scheduler export the data structure of the front-end thread states to, or provide a set of interfaces to the garbage collector and stack overflow/underflow handler. These services, garbage collector and stack overflow/underflow handler, will now be able to update front-end thread states when necessary. They will also need to notify the interrupt handler that they require atomic operations by using the techniques mentioned earlier.

On the other hand, the back-end runtime system might need to be notified when the stack is moved, so it can update its own thread local states as well. This could be done by extending back-end services to allow these changes.

6.4 Abort Critical Section

It might be a problem when a thread sets the global variable `__CRITICAL__` and blocks. We have no solution to prevent this from happening. This will stop scheduler from having a chance to interrupt the current running (blocking) thread and schedule other threads to run. Consider an example where a thread is in a critical section, and a page fault happens. The scheduler can not do anything until the page fault is recovered. On a uniprocessor machine, we can do nothing except deferring the page fault recovery. On a multiprocessor machine, we can schedule another processor to recover the page fault. In either case, we need operating system support to notify the scheduler so it can put the thread into the wait queue, make any other actions, and schedule others threads.

6.5 Ideas on Implementing the proposed back-end interfaces

We propose `updateTCB` and `swtch` to extend the back-end services for the context of preemptive threads. To show the feasibility of the basic system, we describe some ideas for implementing these interfaces.

`updateTCB(tcb *t, struct sigcontext *scp)` takes the thread's local states and machine register dumps as arguments. It then update the thread's local state according to the machine register dumps.

`swtch(tcb *t)` takes the thread's local states as argument, and resumes the execution of the thread. Since the interrupt handler and scheduler run on a separate signal stack, it is easier for `swtch` to set the machine state back to the suspended point. There is no need to manipulate the original stack to preserve it. `swtch` is very similar to `Resume`, they differ mainly in the semantics; `Resume` may return while `swtch` never returns.

The envisioned implementation of `Resume` and `swtch` are similar in that they must first identify the thread to resume or "machine state" to restore from their parameters. `swtch` then restores some of the registers such as general registers, registers for frame and stack pointer so it can jump to the program counter and continue execution as if the thread was never suspended. `Resume` takes a slightly different path. It too will restore some of the registers, but it also needs to setup the thread's activation records, so that when it calls `yield()`, control can be transferred back to the scheduler. We are not familiar with how this would be done.

6.6 Supporting Multiple OS threads, run queues, and policies

Our model describes the simplest case; one OS thread, two C-- threads and a single run queue. To make this framework useful, it is necessary to extend it to the more general case. In the situation where we have multiple OS thread (maybe because of a multi-processor), we suggest that each OS thread run its own scheduler and interrupt handler. Multiple OS thread can still share the same queue but that requires using a global locking mechanism to protect the integrity of the queue. However, this could be problematic because a thread or scheduler may hold the lock and block. For this reason, it may be better that every OS thread maintains a separate run queue and locking mechanism. The binding of a thread and processor may require other front- and back-end services -- which we do not discuss.

The scheduler can be easily extended to support different scheduling policies. The front-end may have to provide services that the thread programmer can use to specify the scheduling policy.

7.0 Summary

We have shown a framework for supporting preemptive threads in C--. Our model extends the current C-- framework with new services. In particular, we suggest and describe two simple extensions to the back-end runtime interface: a mechanism for saving the state, `updateTCB`, and a revised mechanism for resuming the state of a suspended execution, `swtch`. Our model describes the simplest case; two C-- threads multiplexed on an OS thread using single run queue. We conjecture that proving the simplest case demonstrates the plausibility of our system and its extensibility to the general case.

References

[1] Cormack, G. V. 1988. A micro-kernel for concurrency in C. *Software---Practice and Experience*, 18(5):485--92.

[2] Hanson, David R. 1996. C Interfaces and Implementations. *Benjamin/Cummings. Chapter 20: Threads*

[3] Norman Ramsey and Simon Peyton Jones. Machine-Independent Support for Garbage Collection, Debugging, Exception Handling, and Concurrency, [Sept 1998]

[4] Norman Ramsey and Simon Peyton Jones. The C-- runtime interface for concurrency. [March 2000]

[5] Simon Peyton Jones, Norman Ramsey, and Fermin Reig. C--: a portable assembly language that supports garbage collection*, Invited talk at PPDP'99* [July 1999]

[6] Norman Ramsey and Simon Peyton Jones. A Single Intermediate Language That Supports Multiple Implementations of Exceptions*, PLDI'00.* [March 2000]

[7] Pike, R. Newsqueak: A language for communicating with mice. *Computer Science Technical Report 143,* AT&T Bell Laboratories, Murray Hill, New Jersey, 1989

[8] Pike, R. The implementation of Newsqueak. *Software practice and experience,* AT&T Bell Laboratories, Murray Hill, New Jersey