CS162, Spring 2004 **Discussion #3** **Amir Kamil**
UC Berkeley **2/5/04**

Topics: Review, Context Switching, Synchronization, Nachos

1 Review

Recall from last time and lecture the major issues associated with threads and concurrency:

1. *Efficiency*: what is the overhead associated with threads?
   This includes both runtime overhead (thread creation and context switching) and implementation overhead for the operating system designer.

2. *Synchronization*: how to allow threads to share data safely.
   Recall that threads corresponding to the same process share the same address space. Suppose two threads write to the same shared variable `x`, each executing the increment `x++`. This gets translated to the sequence:

   ```
   load(reg0, x)  // read value of x into register
   reg0 := reg0 + 1  // increment value
   store(reg0, x)  // store new value of x.
   ```

   Suppose now that `x` is initially 0, and that both threads execute this code simultaneously. It is possible for them to both read 0 for `x` and then to store 1 into `x`, resulting in a final value of 1 rather than the desired 2. This is known as a *race condition*. We would like to avoid this from happening by using synchronization constructs, which we will discuss over the next week or so.

3. *Scheduling*: how to pick the next thread to run.
   Our goals here are to be fair to all threads and minimize the time a thread takes to complete. We will talk about this in a few weeks.

We've already seen some of the efficiency impacts of threads, and we review context switching. Then we turn our attention to synchronization. Scheduling will wait for a few more weeks.

2 Context Switching

In order to understand exactly how a context switch works, it is useful to look at a simple implementation of the `switch` function in figure 1. As input, this function takes in the thread control blocks for both the source and target thread. (We assume that the source thread has already called the scheduler to determine which thread gets to run next.) The function saves all the state of the source thread in its TCB, including the address at which the source is executing (recall the MIPS convention that this is in the `$ra` register), then loads the state of the target thread and jumps to its point of execution. This sequence of events is shown in figure 2.

There are couple of things to point out about the `switch` function. First, note that it does not save temporary registers. This is because in the MIPS calling convention, the function that calls `switch` must save any temporary registers that it is using, so saving them in the TCB would be redundant. Second, note that a call to `switch` is in general slower than a normal procedure call, since it must save all callee-saved registers. A normal procedure call only has to save those that it clobbers, but in a context switch, there is no way of knowing which ones the new thread will use.

3 Synchronization

Consider the following piece of code, where `x` is a shared variable but `y` is non-shared:

```
/* SWITCH(oldThread, newThread)
 *      oldThread  - The current thread that was running, where the
 *              CPU register state is to be saved.
 *      newThread  - The new thread to be run, where the CPU register
 *              state is to be loaded from.
 */
SWITCH:
        sw      $sp, SP($a0)            # save new stack pointer
        sw      $s0, S0($a0)            # save all the callee-save registers
        sw      $s1, S1($a0)
        sw      $s2, S2($a0)
        sw      $s3, S3($a0)
        sw      $s4, S4($a0)
        sw      $s5, S5($a0)
        sw      $s6, S6($a0)
        sw      $s7, S7($a0)
        sw      $fp, FP($a0)            # save frame pointer
        sw      $ra, PC($a0)            # save return address

        lw      $sp, SP($a1)            # load the new stack pointer
        lw      $s0, S0($a1)            # load the callee-save registers
        lw      $s1, S1($a1)
        lw      $s2, S2($a1)
        lw      $s3, S3($a1)
        lw      $s4, S4($a1)
        lw      $s5, S5($a1)
        lw      $s6, S6($a1)
        lw      $s7, S7($a1)
        lw      $fp, FP($a1)
        lw      $ra, PC($a1)            # load the return address

        jr      $ra
```

Figure 1: The `switch` call implemented in MIPS assembly. Here, `SP`, `S0`, etc. refer to offsets in the TCB.

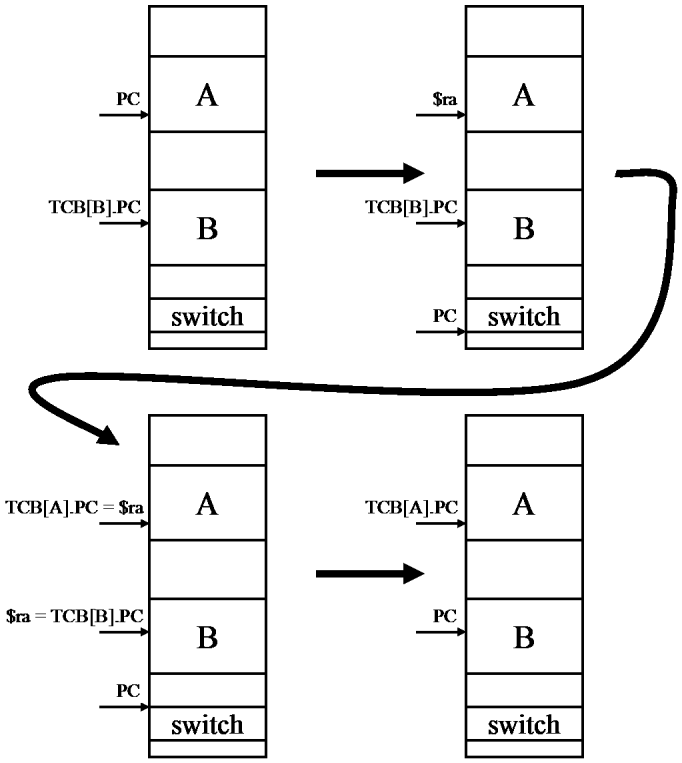

Figure 2: Execution of the `switch` call from thread A to thread B in MIPS.

```
x := y
print(x)
```

Suppose that threads A and B execute this code simultaneously, with `y = 1` in A and `y = 2` is B. The following is a possible sequence of statements executed:

```
A:  x := y   // x is now 1
--- Context Switch to B ---
B:  x := y   // x is now 2
... // some statements
--- Context Switch to A ---
A:  print(x)  // 2 is printed out
```

In this sequence, A prints out the unexpected value 2 to the screen.

Ideally, we would like the result of A's `print` statement to be deterministic, by preventing any other thread from executing the above code while A is. Such a section of code, that only one thread can run at a time, is called a *critical section*. The synchronization mechanism we use to restrict access to a critical section must obey three conditions:

1. *Mutual exclusion*: only one thread can execute the critical section at a time
2. *Progress*: one thread must always be allowed to enter the critical section
3. *Bounded waiting*: a thread should not have to wait indefinitely to enter the critical section

The simplest synchronization construct that satisfies these conditions is the *lock*. A lock has a single owner, and two functions, `acquire` which grants the calling thread ownership of the lock if the lock is unowned or forces it to wait if it is owned, and `release` which results in the calling thread losing ownership of the lock and notifying a thread waiting on the lock. Using a lock, we can rewrite the critical section as follows, where `l` is a shared lock:

```
class Lock {
  Thread owner;
  boolean locked;
  Queue waiters;
  void acquire() {
    if (locked) {
      waiters.enqueue(Thread.currentThread());
      sleep();  // ignore implementation of this method, assume it works
    } else {
      locked = true;
      owner = Thread.currentThread();
    }
  }
  void release() {
    if (Thread.currentThread() != owner) {
      throw new Exception("not lock owner");
    } else if (!waiters.isEmpty()) {{
      owner = (Thread) waiters.dequeue();
    } else {
      locked = false;
    }
  }
}
```

Figure 3: Sample Java implementation of a lock.

```
acquire(l)
x := y
print(x)
release(l)
```

Let's take a look at one possible lock implementation. A lock needs to keep track of its owner, the threads waiting on it, and whether or not it is in use. The `acquire()` code must check if the lock is in use, and if so, put the current thread to sleep, and if not, grant the current thread ownership of the lock. Figure 3 provides a possible Java implementation.

But does this lock work? Suppose both A and B call `acquire()` at the same time. Then it is possible for the following sequence of statements to execute:

```
A:  if (locked) // locked is false, jump to else
A:  <jump to else>
--- Context switch to B ---
B:  if (locked)  // locked is false, jump to else
B:  <jump to else>
B:  locked = true;  // B obtains lock
... B enters critical section
--- Context switch to A ---
A: locked = true;  // A obtains lock
... A enters critical section
```

Both A and B think they've acquired the lock! The `acquire()` method itself is now a critical section as well.

Synchronization cannot be implemented completely in software and must require some level of hardware support. Locks are usually implemented using atomic synchronization primitives provided by the hardware. We look at two, `TestAndSet` and `Swap`. For now, we will use these primitives directly to protect a critical section, though in practice we would use locks or some other higher level construct. The lecture notes contain a lock implementation using these primitives.

```
boolean TestAndSet(int x) {
  int tmp = x;
  x = 1;
  return tmp == 1;
}
```

Figure 4: The `TestAndSet` function.

```
void Swap(int *x, int *y) {  // need pointer access if pass by value
  int tmp = *x;
  *x = *y;
  *y = tmp;
}
```

Figure 5: The `Swap` function.

The first synchronization primitive is `TestAndSet`. Given a variable, `TestAndSet` sets it to 1 and returns `true` if its previous value was 0, and `false` if it was 1. Figure 4 shows the equivalent operations `TestAndSet` execute, though it does them atomically.

Using `TestAndSet`, we can rewrite our critical section as follows, with `k` a shared integer:

```
while (TestAndSet(k))
x := y
print(x)
k := 0  // release k
```

Suppose A executes this code first. Then `k` is 0, so `TestAndSet(k)` returns `false`, and A enters the critical section. Now suppose B starts executing the code. Now `k` is 1, so `TestAndSet` will keep returning true, and B will not continue. But once A exits the critical section and resets `k` to 0, B will continue. Thus this synchronization implementation works.

Alternatively, we can implement synchronization using `Swap`. Taking two integers as input, `Swap` obviously exchanges the values of the two variables. Figure 5 shows the equivalent operation of the `Swap` function.

Using `Swap`, we can rewrite the critical section as follows, with `k` a shared integer and `m` unshared:

```
m := 1
do Swap(k, m) while (m == 1)
x := y
print(x)
k := 0  // release k
```

Again, suppose A executes this code first. Then `k` is 0, so `Swap(k, m)` sets `k` to 1 and `m` to 0, and A enters the critical section. Now suppose B starts executing the code. Now `k` is 1, so `Swap` will keep setting `k` to 1 and `m` to 1, and B will not continue. But once A exits the critical section and resets `k` to 0, B will continue. Thus this synchronization implementation also works.

While the above are correct synchronization implementations, they exhibit poor performance as they require *busy waiting*. Thread B continually checks whether or not it is safe to enter the critical section, wasting CPU cycles. This is particularly bad if B is a high priority thread and A is low priority, as B will hog the CPU not doing any useful work. A better implementation would put B to sleep, and have A wake B up when it exits the critical section. This is one reason we use higher-level constructs like locks that don't use busy waiting.

4 Nachos

For phase 1, you need to understand the following in detail:

- `nachos.threads.KThread`
- `nachos.threads.ThreadedKernel`
- `nachos.threads.Lock`
- `nachos.machine.TCB` (the interface in detail, and basic understanding of the code)

All your work for this phase is going to be done in the `nachos.threads` package, so you don't need to understand the other packages in any detail. You will need to read and comprehend the above classes. For this phase, I suggest you start reading at `KThread.fork()` rather than at `Machine.main()`.

In the Nachos code, you'll notice that the kernel does privileged operation on occasion (via a call to `doPrivileged()`). There are certain operations that you are not allowed to do, such as creating a Java thread or use a Java `File`. Attempting to use them will result in the Nachos `SecurityManager` crashing your code. In addition, you cannot use the `synchronized` keyword, we will grep for it. All synchronization, thread, and file operation must be done through Nachos.

The interface to and descriptions of the methods you need to understand is in figure 6. The Nachos machine implements kernel threads on top of Java threads, but it ensures that only one Java thread is running at a time. The Nachos thread operations are detailed in figure 7.

I recommend you get started on the design for phase 1 immediately. Since we haven't yet discussed priority scheduling, I suggest that you delay working on it until later. We haven't done condition variables yet either, but much of the project uses them, so read over the lecture notes and the code to teach yourself how to use them. You should put in a lot of time and thought into the boat problem, as it is likely to be the hardest part and the most difficult to get correct. We will discuss condition variables, priority scheduling, and design documents next time.

`nachos.machine.*` – the internals of the implementation
Not really important to understand how this works, but need to know what the public methods are

`Machine.interrupt().disable()`
Disable interrupts, return flag of previous interrupt state
`Machine.interrupt().enable()`
Enable interrupts
`Machine.interrupt().restore()`
Restore to previously saved flag

`TCB.contextSwitch()`
Context switch to this `TCB`. Used internally by `KThread`, you shouldn't ever need to call this
`TCB.start()`
Used to bootstrap a new `TCB`; used internally by `KThread`

`nachos.threads.*` – what you will be modifying

`KThread(Runnable target)`
Create a new kernel thread and associate with it the code in `target.run()`
`KThread.setName(String name)`
Associate a new name, can be retrieved with `getName`
`KThread.fork()`
Fork the given thread - that is, start it running
`KThread.yield()`
Cause the current thread to yield the CPU
`KThread.sleep()`
Cause the current thread to block - will be woken up later
`KThread.ready()`
Move this thread to the ready queue, i.e. wake it up

`Lock.acquire()`
Sleep until this lock can be acquired
`Lock.release()`
Release the lock, and wake waiting thread if any

Figure 6: Interface and description of Nachos thread methods.

Fork (`KThread.fork()`, executed by parent thread)

1. create new `TCB` for target thread
2. call `TCB.start()`

Thread creation

- `TCB.start()`, executed by parent thread
  1. create new Java thread for target, that runs `threadroot()`
  2. start Java thread
  3. go to sleep
- `TCB.threadroot()`, executed by child thread
  1. wakes previous thread
  2. yields (i.e. goes to sleep)

Context switch (`TCB.contextSwitch()`, executed by previous thread)

1. set current `TCB`'s state to `ready` (not `running`)
2. interrupt next thread (wake it up)
3. go to sleep

Notes:

- Going to sleep implemented using Java `wait()` method, waking up using `notify()`.
- There is a global variable corresponding to the current `TCB`.
- Read over this code very carefully: thread A can be running code in `Thread` B (distinguish between thread and `Thread`), so keep track of which thread is running at all times.

Figure 7: Thread operations in Nachos.