CIS520 - Threads and Concurrency

All you really need to know for the moment is that the universe is a lot more complicated than you might think, even if you start from a position of thinking it's pretty damn complicated in the first place.

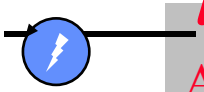
Douglas Adams, "Mostly Harmless"

A First Look at Some Key Concepts

kernel

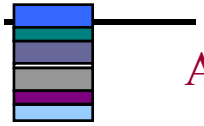
The software component that controls the hardware directly, and implements the core privileged OS functions.

Modern hardware has features that allow the OS kernel to protect itself from untrusted user code.



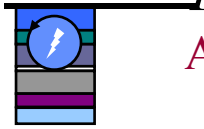
thread

An executing stream of instructions and its CPU register context.



virtual address space

An execution context for thread(s) that provides an independent name space for addressing some or all of physical memory.



process

An execution of a program, consisting of a virtual address space, one or more threads, and some OS kernel state.

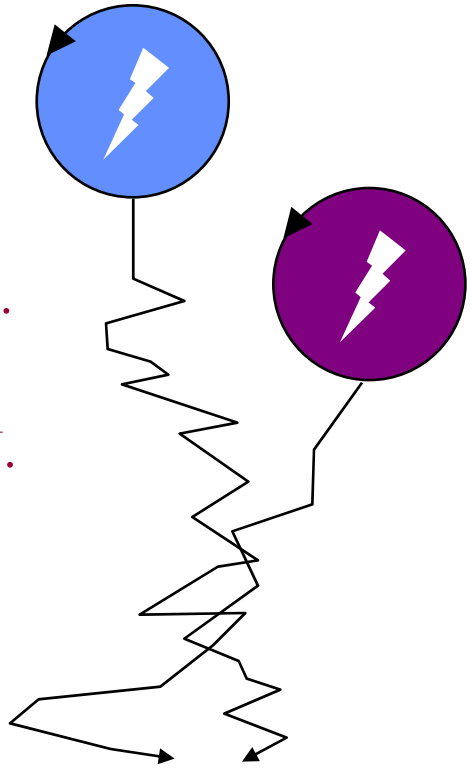
Threads

A *thread* is a schedulable stream of control.

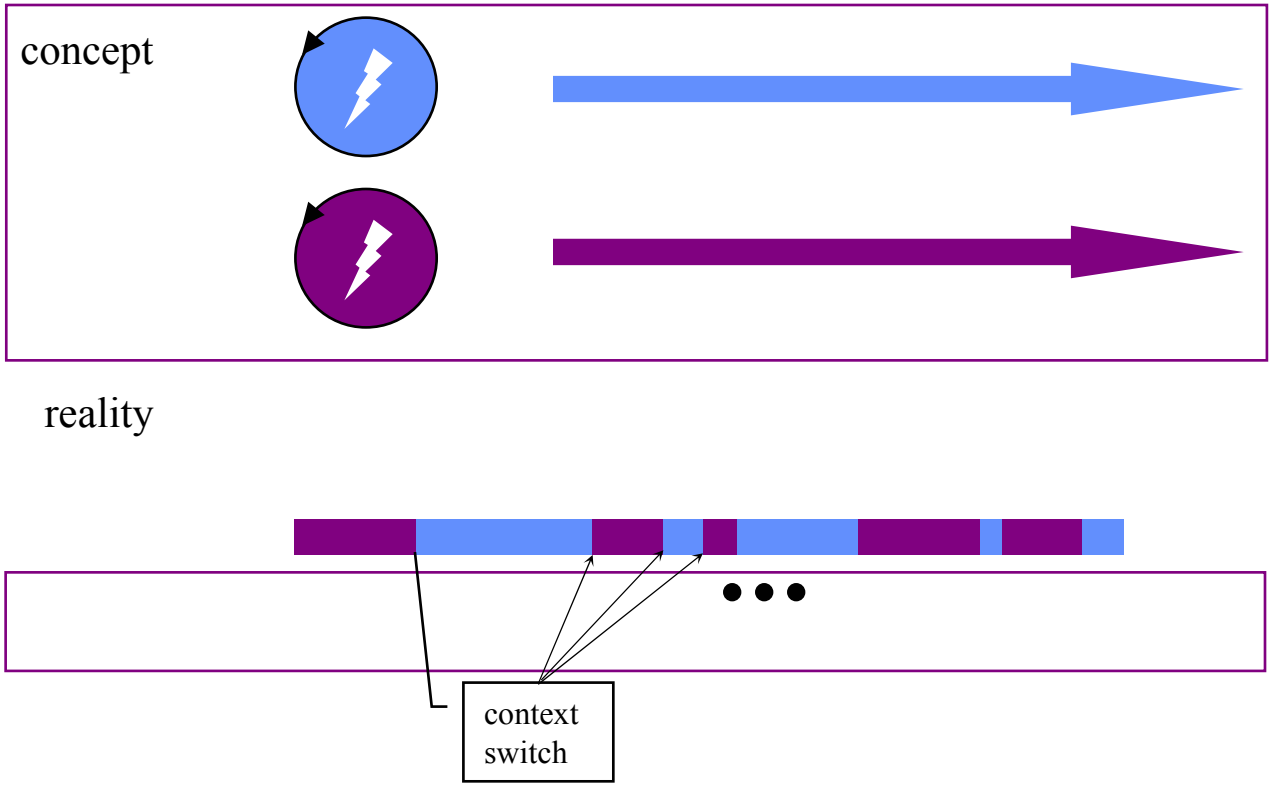
- defined by CPU register values (PC, SP)
- *suspend*: save register values in memory
- *resume*: restore registers from memory

Multiple threads can execute independently:

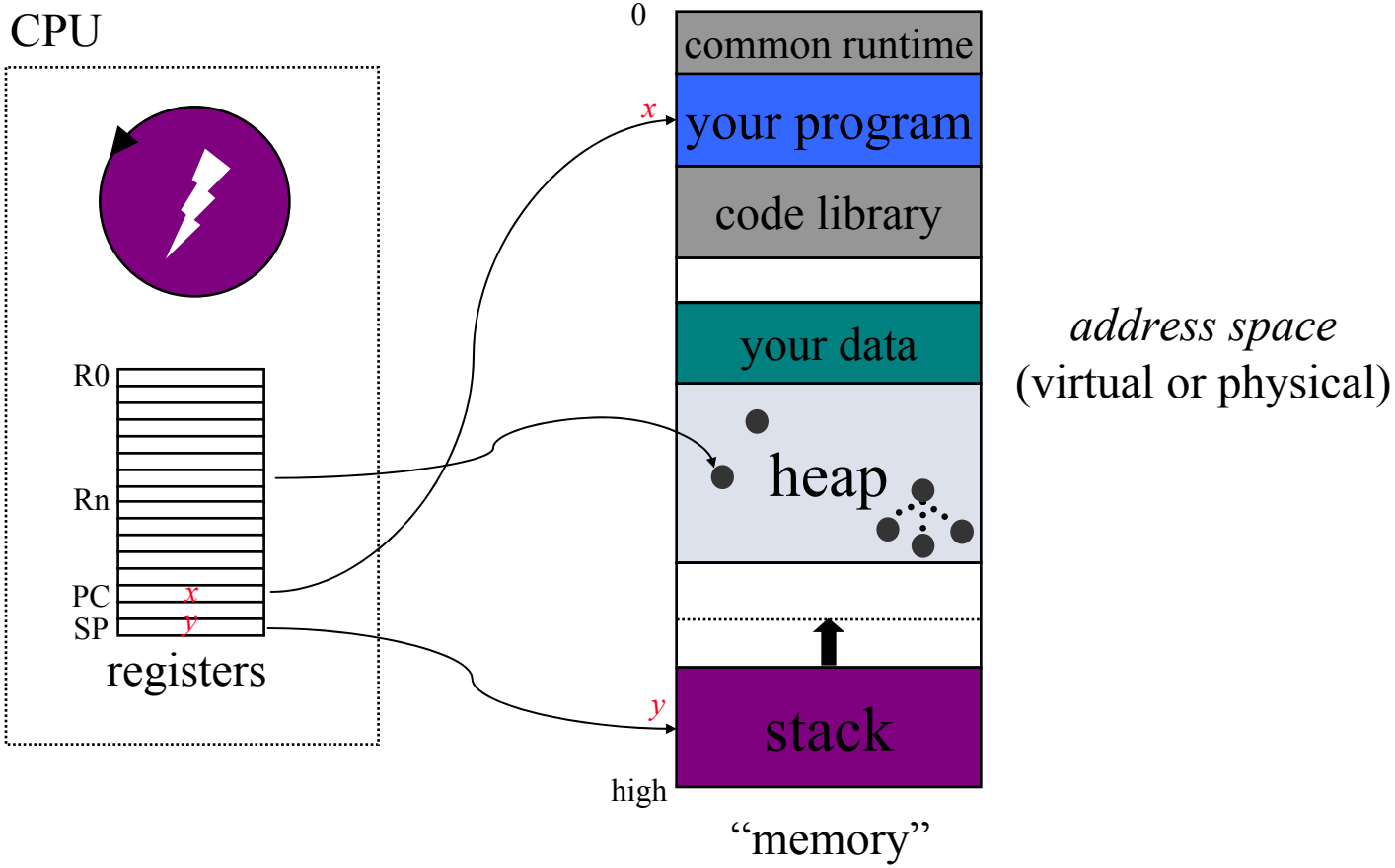
- They can run in parallel on multiple CPUs...
  - ***physical concurrency***
- …or arbitrarily interleaved on a single CPU.
  - ***logical concurrency***
- Each thread must have its own stack.



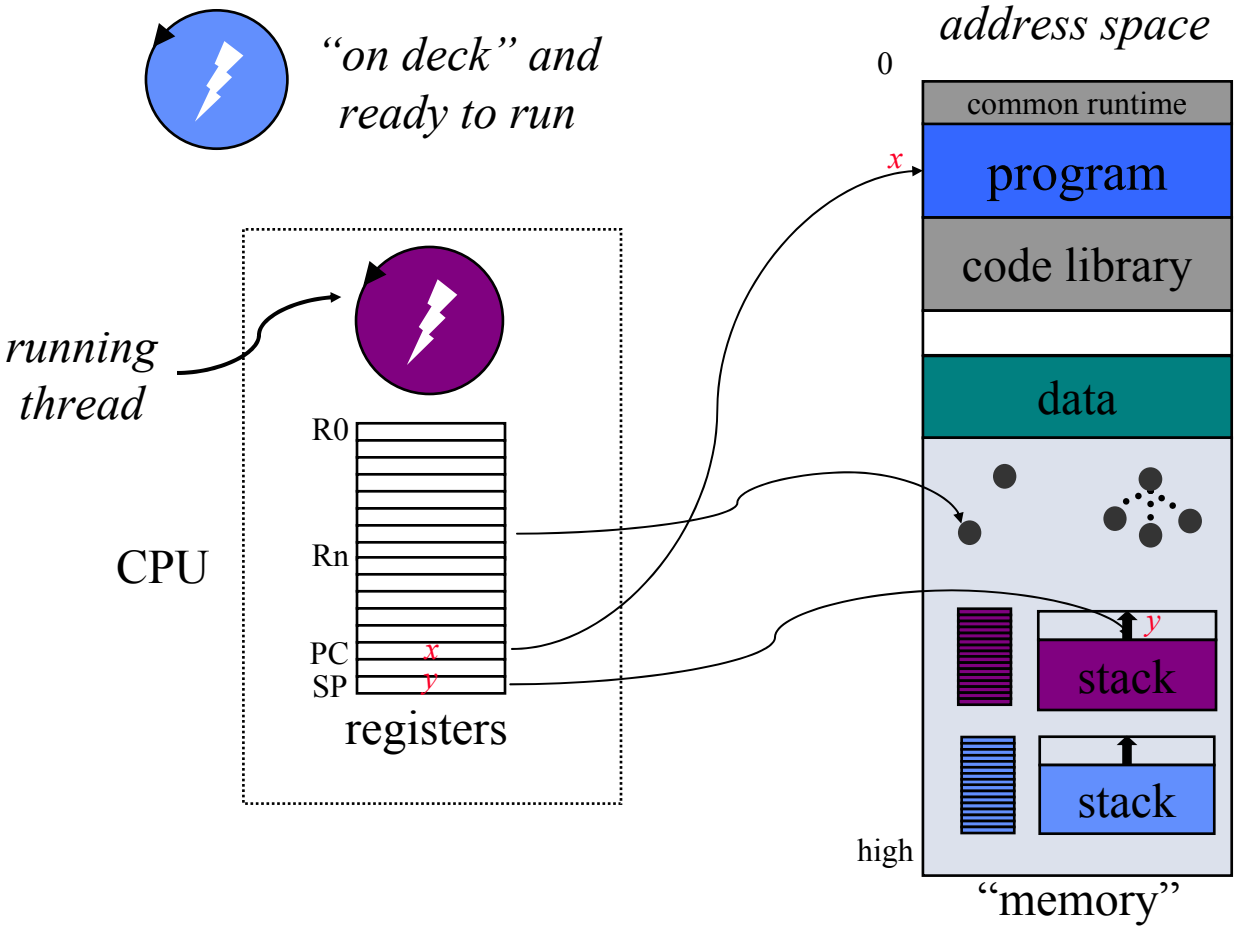
Two Threads Sharing a CPU



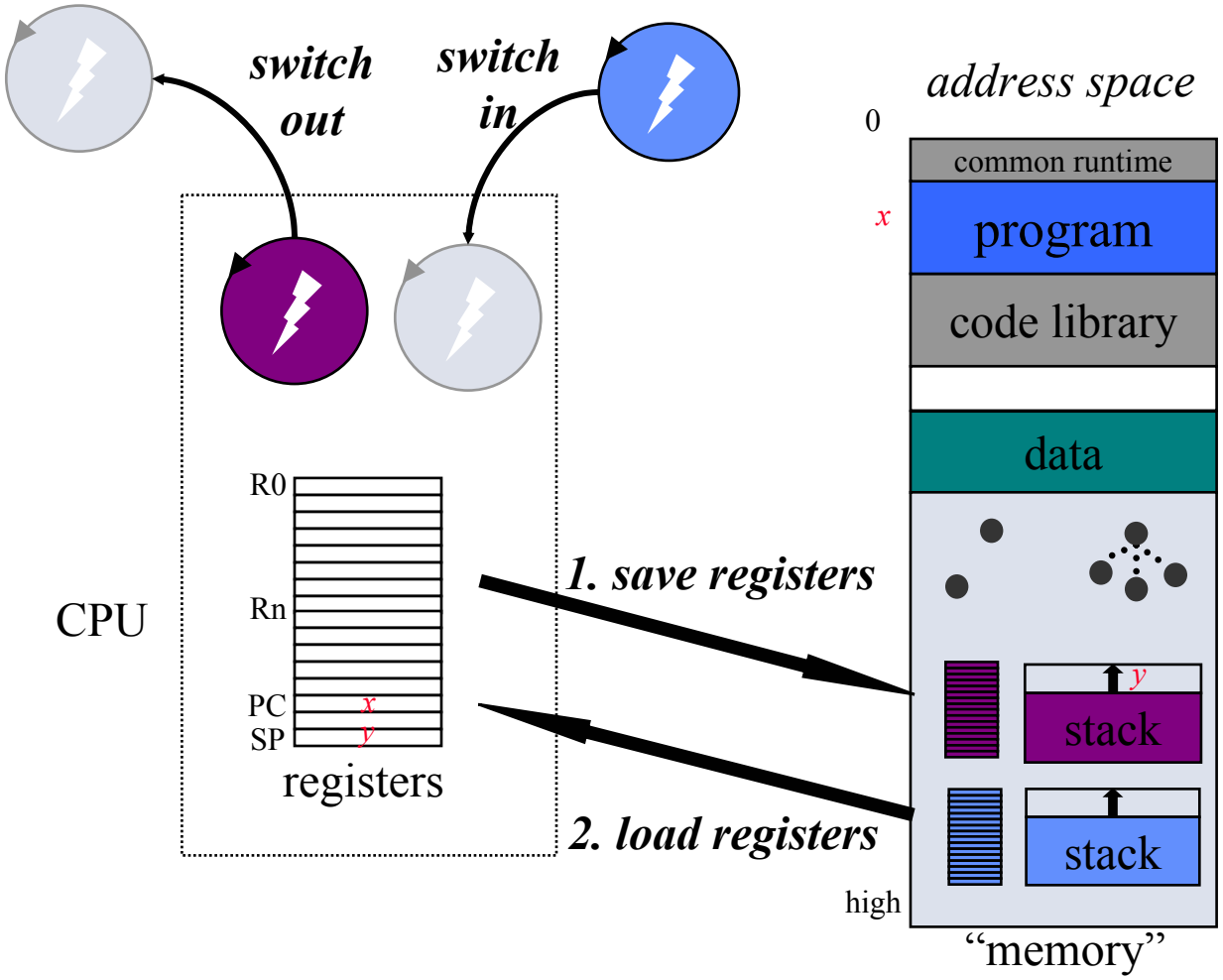
A Peek Inside a Running Program



A Program With Two Threads



Thread Context Switch



Context Switches: Voluntary and Involuntary

On a **uniprocessor**, the set of possible execution schedules depends on *when context switches can occur*.

- *Voluntary*: one thread explicitly yields the CPU to another.

  E.g., a Nachos thread can suspend itself with ***Thread::Yield***.

  It may also *block* to wait for some event with ***Thread::Sleep***.

- *Involuntary:* the system *scheduler* suspends an active thread, and switches control to a different thread.

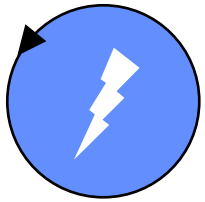
  Thread scheduler tries to share CPU fairly by *timeslicing*.

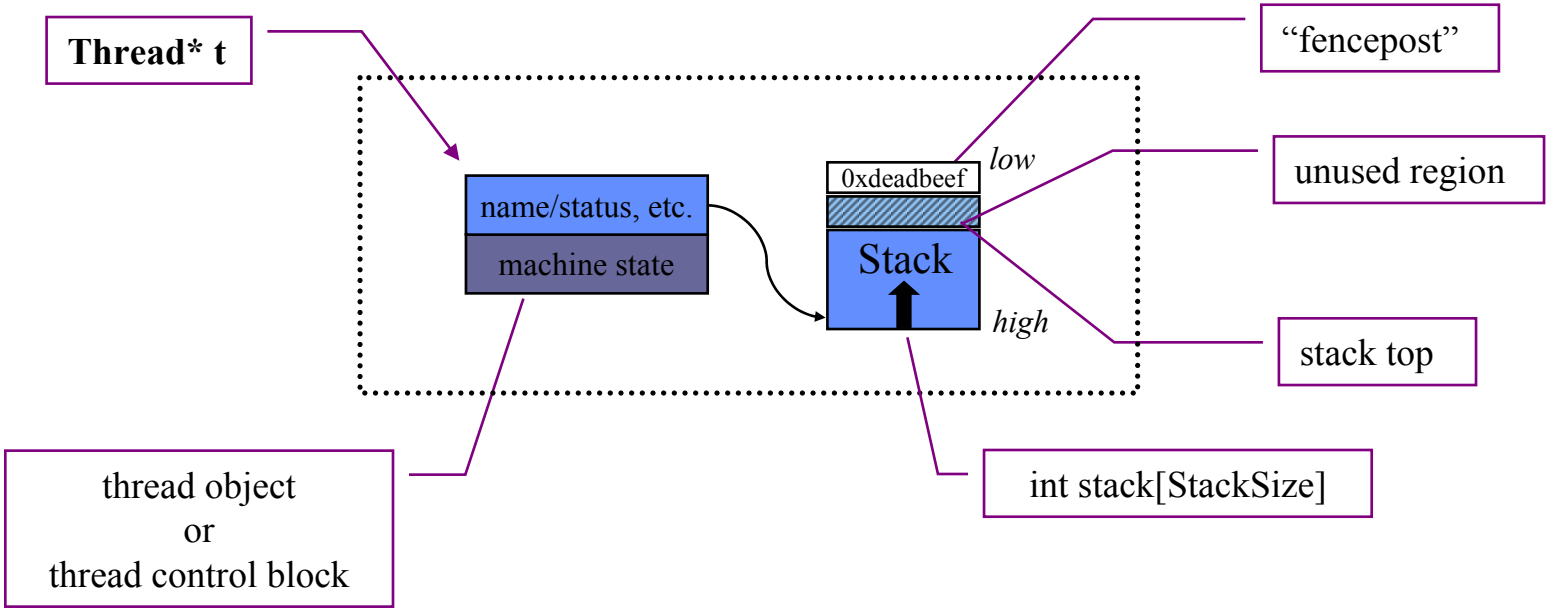
  Suspend/resume at periodic intervals (e.g., **nachos -rs**)

  Involuntary context switches can happen "any time".

A Nachos Thread

```
t = new Thread(name);
t->Fork(MyFunc, arg);
currentThread->Sleep();
currentThread->Yield();
```





Why Threads Are Important

1. There are lots of good reasons to use threads.

   "easy" coding of multiple activities in an application

   e.g., servers with multiple independent clients

   parallel programming to reduce execution time

2. Threads are great for experimenting with concurrency.

   context switches and interleaved executions

   race conditions and synchronization

   can be supported in a library (Nachos) without help from OS

3. We will use threads to implement processes in Nachos.

   (Think of a thread as a process running *within the kernel*.)

Concurrency

Working with multiple threads (or processes) introduces *concurrency*: several things are happening "at once".

How can I know the order in which operations will occur?

- ***physical concurrency***

  On a **multiprocessor**, thread executions may be arbitrarily interleaved at the granularity of individual instructions.

- ***logical concurrency***

  On a **uniprocessor**, thread executions may be interleaved as the system switches from one thread to another.

  context switch (suspend/resume)

Warning: concurrency can cause your programs to behave unpredictably, e.g., crash and burn.

The Dark Side of Concurrency

With interleaved executions, the order in which processes execute at runtime is *nondeterministic*.

- depends on the exact order and timing of process arrivals
- depends on exact timing of asynchronous devices (disk, clock)
- depends on scheduling policies

Some schedule interleavings may lead to incorrect behavior.

- Open the bay doors *before* you release the bomb.
- Two people can't wash dishes in the same sink at the same time.

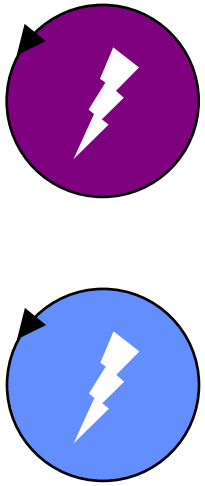
The system must provide a way to coordinate concurrent activities to avoid incorrect interleavings.

Example: Incrementing a variable

```
int a = 0;

void sum(int p) {
    a++;
   printf("%d : a = %d\n", p, a);
}

void main() {
   Thread *t = new Thread("child");
   t->Fork(sum, 1);
   sum(0);
}
```



In greater detail.

Sum()

1. first reads the value of a into a register.
2. then increments the register,
3. then stores the contents of the register back into a.
4. then reads the values of of the control string, p and a into the registers that it uses to pass arguments to the printf routine.
5. then calls printf, which prints out the data.

The best way to understand the instruction sequence is to look at the generated assembly language (cleaned up just a bit).

```
la      a, %r0
ld      [%r0],%r1
add     %r1,1,%r1
st      %r1,[%r0]
ld      [%r0], %o3 ! parms are
                   !passed in %o0
mov     %o0, %o1
la      .L17, %o0  !call printf
```

*You can have the compiler generate assembly code instead of object code by giving it the -S flag. It will put the generated assembly in the same file name as the .c or .cc file, but with a .s suffix.

Possible outcomes:

- So when execute concurrently, the result depends on how the instructions interleave.
- *What are possible results?*

0 : 1	0 : 1	1 : 2	1 : 1
1 : 2	1 : 1	0 : 1	0 : 1
1 : 1	0 : 2	0 : 2	1 : 2
0 : 2	1 : 2	1 : 1	0 : 2

- **So the results are nondeterministic** - you may get different results when you run the program more than once. So, it can be very difficult to reproduce bugs. Nondeterministic execution is one of the things that makes writing parallel programs much more difficult than writing serial programs.

Examples

1:2 & 1:1

```
Sum() {
  !increment a
la      a, %r0
ld      [%r0],%r1
add     %r1,1,%r1
st      %r1,[%r0]

  !call printf
ld      [%r0], %o3
mov     %o0, %o1
la      .L17, %o0
}
```

Atomic Operations

- To get consistent results, sum() must be **atomic**.
  - must prevent the interleaving of the instructions in a way that would interfere with the additions.
- **An atomic operation is one that executes without any interference from other operations** - in other words, it executes as one unit.
  - Typically build complex atomic operations up out of sequences of primitive operations.
  - In our case the primitive operations are the individual machine instructions.
- More formally, if several atomic operations execute, the final result is guaranteed to be the same as if the operations executed in some *serial order*.
- Otherwise we have a *race condition*.

Race Conditions Defined

1. Every data structure defines *invariant* conditions.
   - defines the space of possible *legal* states of the structure
   - defines what it means for the structure to be "well-formed"
2. Operations depend on and preserve the invariants.
   - The invariant must hold when the operation begins.
   - The operation may temporarily violate the invariant.
   - The operation restores the invariant before it completes.
3. Arbitrarily interleaved operations violate invariants.
   - Rudely interrupted operations leave a mess behind for others.
4. Therefore we must constrain the set of possible schedules.