Race Condition

Traditionally, communicate (voluntarily or necessarily) thru **shared memory**

In the presence of preemptive scheduling, unfettered shared memory leads to **race conditions**

Example, I

Assembly instructions are (usually) atomic, but language statements rarely are

Consider:

```
foo = foo + 1;
```

1 "high level" statement probably compiles into 3 assembly instructions:

```
A. read foo into register
B. increment register
C. write register back to foo
```

Each instruction may be atomic, but the whole is certainly not: threads may switch between any two instructions

Example, II

Suppose 2 threads each execute
`foo = foo + 1` simultaneously

And suppose `foo` is initially 4

This could happen:

```
1-A. thread #1 reads 4
     ... thread #1 is de-scheduled ...
2-A. thread #2 reads 4
2-B. thread #2 increments register to 5
2-C. thread #2 writes 5
     ... thread #2 runs to end of its time slice ...
1-B. thread #1 increments register to 5
1-C. thread #1 writes 5
```

Two increments occurred, but following
both operations `foo` has increased by only 1!

How to Prevent Race Conditions

1. **Non-preemptive scheduling**: threads switch among themselves only at safe times; the OS does not preempt thread execution

2. Preemptive scheduling: OS schedules threads at unpredictable times, BUT ... threads protect their **critical sections** with some **mutual exclusion** mechanism

1. Non-preemptive Scheduling, I

Avoid race conditions by switching between threads only at "safe" times

Two realizations of non-preemptive scheduling:

1. OS provides scheduler "yield" operation & TRUSTS threads to yield to each other correctly (outdated)
2. Threads within a single process trust each other to yield correctly

1. Non-preemptive Scheduling, II

The difference: in #2, same code (hence, same author(s) or at least authors that trust each other) are executing in all threads—a trust relationship can exist; also, separation among processes guarantees that a problem in one process affects only that process

Trust relationship is reason why non-preemptive scheduling cannot be used by everybody in general multi-user setting

Coroutines, I

Coroutine is *language* primitive for non-preemptive scheduling

Coroutine is like function, except control resumes in place where it left off last time—replace both `call` & `return` with `resume`

Illustration:
Function Invocation

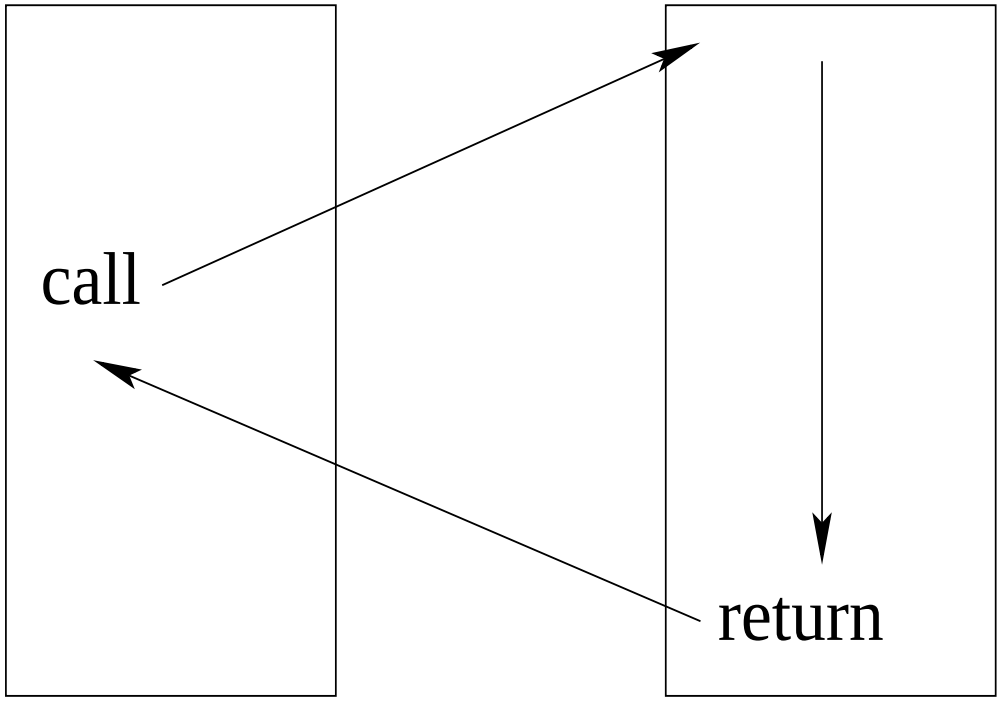
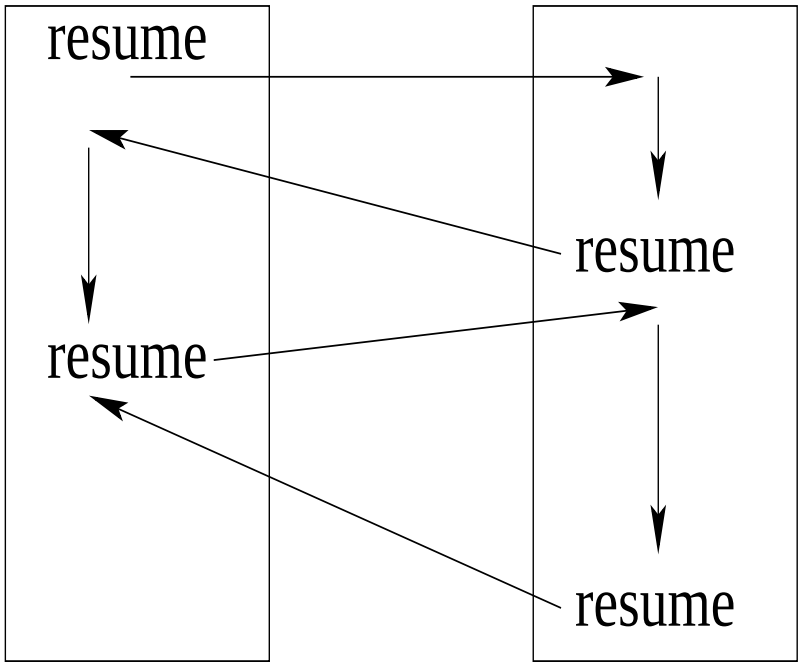


Illustration:
Coroutine Invocation



Coroutines, II

Activation record (AR) of coroutine has extra **resume point** field that describes *where within itself* to start execution

Invoking coroutine now requires jumping not to constant address, but to contents of constant offset into AR of called unit

Summary of Non-Preemptive Scheduling

Not useful in a general multi-user, timeshared setting

But ... might be useful to achieve predictable/coordinated scheduling in a setting where threads "trust" each other — e.g., real-time system

2. Preemptive Scheduling: Critical Section

Critical section is a code section that performs a *consistent transformation* or requires a *consistent view* of some data structure

Code section is "critical" because ...

if two threads are simultaneously in their critical sections, unfavorable preemptive scheduling might cause errors;
e.g., `foo = foo + 1` race condition mentioned above

Critical Section Examples

Funds transfer logic must execute **atomically**:

```
1. add N dollars to account X
2. subtract N dollars from account Y
```

To unlink element from doubly linked list must update TWO pointers:

1. Previous element's "next" pointer
2. Next element's "prev" pointer

If another thread interrupts either operation and affects the same variables, errors are likely

More on Critical Section

Critical section is defined wrt data structure(s) accessed by the code

If several sections of code access the same variable(s) then EACH of these sections is a critical section

Correct Execution of Critical Section

Q: How can critical sections be executed without danger?

A: Each critical section must follow a convention for mutual exclusion

Note:

- "Convention" means writing correct code is programmer's responsibility; mutual exclusion will not be arranged by language, compiler, operating system, etc.
- Every critical section that accesses shared variable(s) must follow the convention

Example Convention: Locking

There are several conventions for mutual exclusion

Most common and easiest to understand is the **lock**

E.g., all critical sections that access shared variable(s) must be written like this:

```
1. get lock
2. ... access shared variable(s) ...
3. drop lock
```

Why Locking Works, I

Only one thread can hold lock at a time

If a 2nd thread tries to execute its critical section while lock is held, 2nd thread's attempt to get lock will **block**

To "block" means call to get lock does not return for a long time, until lock becomes available following 1st thread dropping it

Why Locking Works, II

Q: But ... isn't the lock a shared variable just like the shared variable(s) it is protecting? Accessing the lock variable re-creates the same problem!

Good point. The answer:

A: Hardware can access certain data (specifically: one word) *atomically*

A lock variable is a single machine word. The lock is always accessed atomically, usually by special hardware instructions designed for the purpose.

2. Preemptive Scheduling: Mutual Exclusion

Conditions for satisfactory mutual exclusion within critical section:

1. No two threads may be simultaneously inside their critical sections (**partial correctness**)
2. No thread should wait arbitrarily long to enter its critical section (**liveness**—freedom from **starvation**)
3. No thread stopped outside its critical section should block other threads
4. No assumptions about relative speeds of threads or number of CPUs
5. No knowledge (at coding time) about number/identity of other threads

Aside:

Correctness and Liveness

To prove an algorithm "correct," often prove two properties separately: partial correctness and liveness

Roughly ...

- Partial correctness: algorithm never does the wrong thing (although may do nothing)
- Liveness: algorithm always does something
- Partial correctness + liveness = algorithm always does the right thing

Aside:
Starvation and Livelock

Starvation means: one particular thread can never make progress

Example: priority scheduling—highest priority ready thread is given CPU; if there is always a high priority thread, then low priority thread will never run

Livelock (as opposed to deadlock) is when set of threads are starved

Livelock Example

Consider retransmission algorithm of original Ethernet:

```
1. Send packet on Ethernet
2. Listen to Ethernet: did packet collide?
3. if (collision occurred) {
      try again in N time units
   }
```

What happens if 2 stations transmit simultaneously?

That's why ACTUAL retransmission algorithm is:

```
1. Send packet on Ethernet
2. Listen to Ethernet: did packet collide?
3. if (collision occurred) {
      try again in RANDOM amount of time
               between 0 and N time units
   }
```


Mechanisms for Mutual Exclusion

1. Disable interrupts

2. *Without* hardware atomic instructions other than single-word read or single-word write

3. Hardware instructions for locking

4. Hardware instructions for atomic access/update of certain common data structures

5. Higher level solutions (built on lower level solutions)

Real usage: #3 and #5 at user level,
#1 and #3 within OS,
#4 used mostly only to implement #3,
#2 is a curiosity

1. Disabling Interrupts

How it works:

1. `disable interrupts`
2. `perform critical transformation/view`
3. `re-enable interrupts`

Works because pre-emptive scheduling is temporarily disabled

Satisfies all 5 conditions

1. Disabling Interrupts

Requires privilege

Wasteful—halts ALL threads, even if only SOME want mutex

Susceptible to programming bugs: if you forget to re-enable interrupts, whole CPU freezes

Doesn't work on multiprocessor

1. Disabling Interrupts

Despite all these drawbacks, once was THE mutex mechanism within uniprocessor OSes

Now outdated, as all OSes have been upgraded to run on multiprocessors

UNIX was once filled with `splnet`, `splbio`, `spltty`, and `splx` macros

2. Mutex Without Special Atomic Instructions

Q: Is it possible to arrange mutual exclusion through clever use of atomic load & store instructions?

A: Yes, although the solutions are complex & have serious disadvantages

Classic solutions:

- Strict Alternation
- Peterson's solution
- Dekker's solution

