Principles of Concurrency

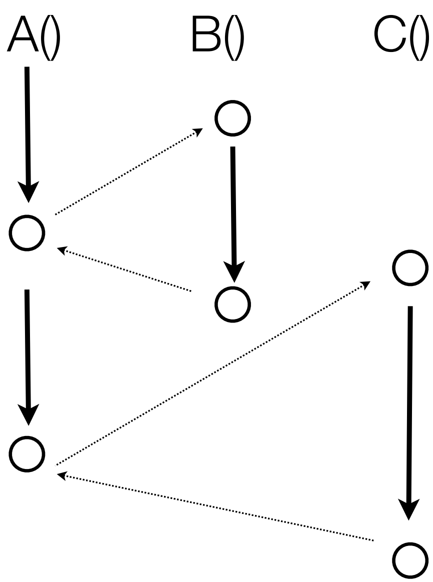
Lecture 2

Coroutines, Threads, and Processes

Coroutines

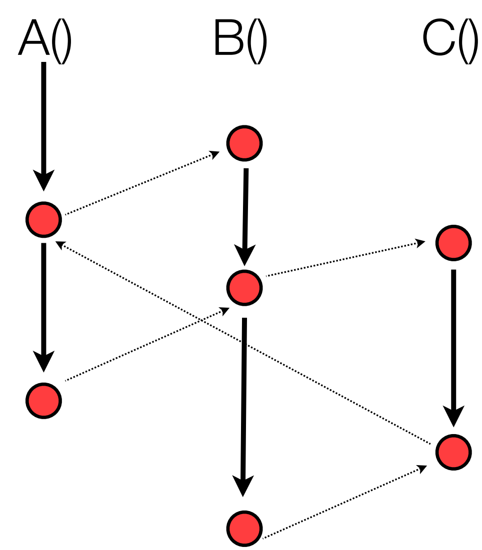
- Units of work that cooperate with one another to make progress.
- A generalization of iterators that remembers its state

Procedure calls



hat happens when A() calls B() again?

Coroutines



When a coroutine returns, it remembers its program state. Why is this useful?

nciples of Concurrency

Coroutines and Concurrency

- How would you implement coroutines?
  - Typically, implementations of procedures and procedure calls involving pushing and popping "activation frames" on the stack
  - These frames hold the arguments and local variables for the call.
  - The frame is popped when the procedure is returned.
  - How do we preserve the state that will be used when we make the next call?

    Keep multiple stacks, one for each coroutine

    Essential feature of threads

Continuations

A reified representation of a program's control stack.

Example:

```
proc f(x) = { ...
              g(y);
              ... ;   A
            }

proc h(y) = { ...
              f(...);
              ... ;   B
            }
```

When g is called, the program stack retains enough information to "remember" that **A** must be executed and then **B**.

The stack captures the "rest of the computation" - it is the *continuation* of the call to g().

If the computation were preempted immediately after the call to g() returns, its resumption would entail execution of the continuation

Continuations

- Can we reify this notion into a source language?
  - result is a continuation, a reified representation

    (in the form of an abstraction) of a program control-stack.
  - Define a primitive operation called call/cc:
    - call-with-current-continuation
    - callcc (fn k => e)
      - captures the current continuation, binds to k, and evaluate e
      - the notation fn k => e defines an anonymous function that takes k as an argument
    - (k x)
      - apply continuation k with argument x

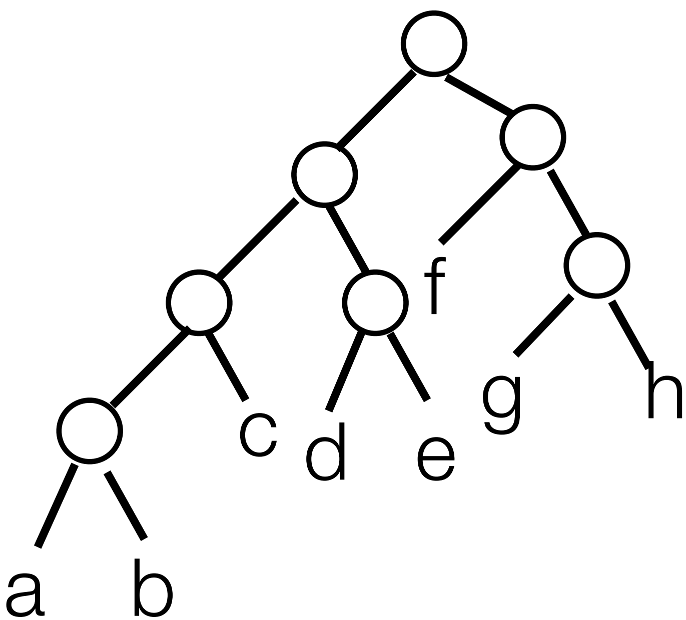
Examples

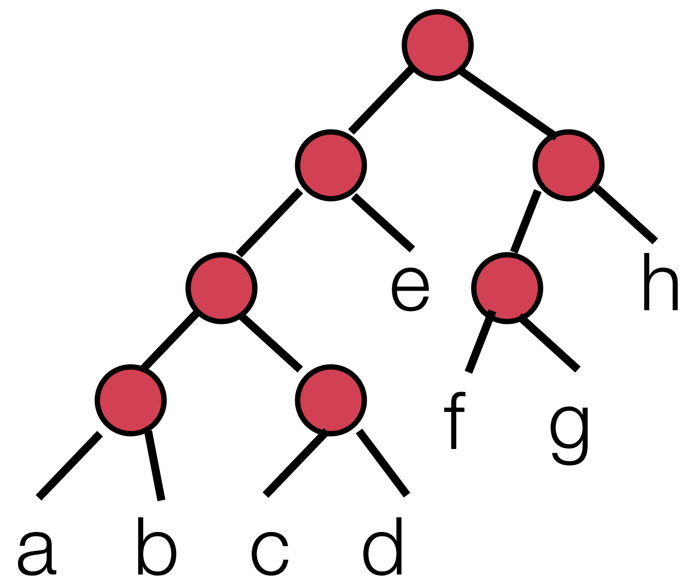

```
(+ (call/cc (lambda (k) (k 3) + 2)) 1)

(let ((f (call/cc (lambda (k)
                         (lambda (x)
                            (k (lambda (y) (+ x y)))))))
   (f 6))
```

Example: Samefringe

- Two binary trees have the same fringe if they have exactly the same leaves reading left to right





Samefringe

First approach:

- Collect leaves of both trees into two lists, and compare elements

```
(define (collect-leaves tree)
  (cond ((empty-tree tree) '())
        ('t (let ((left-leaves (collect-leaves (left tree)))
                  (right-leaves (collect-leaves (right tree))))
              (append left-leaves right-leaves)))))


(define (samefringe t1 t2)
  (letrec ((t1-leaves (collect-leaves t1))
           (t2-leaves (collect-leaves t2))
           (compare (lambda (l1 l2)
                      (cond ((eq? l1 '())  (eq? l2 '()))
                            ((eq? l2 '()) #f)
                            ('t (cond ((equal? (car l1) (car l2))
                                       (compare (cdr l1) (cdr l2)))
                                      ('else #f)))))))
    (compare t1-leaves t2-leaves)))
```

- What’s wrong with this approach?

Samefringe Using Coroutines

- Rather than collecting all leaves or transforming tree eagerly, generate leaf values for two trees lazily
- Create generators for the two trees that yield the next leaf when invoked, and return control back to the caller, remembering where they are

```
(define samefringe-lazy
  (lambda (tree1 tree2)
    (let ((gen1 (make-generator tree1))
        (gen2 (make-generator tree2)))
      (driver gen1 gen2))))

(define driver
  (lambda (gen1 gen2)
    (let ((leaf1 (gen1))
        (leaf2 (gen2)))
      (if (= leaf1 leaf2)
       (if (zero? leaf1)
           #t
           (driver gen1 gen2))
       #f))))
```

```
(define make-generator
  (lambda (tree)
    (letrec
          ((caller '*)
          (generate-leaves
           (lambda ()
             (letrec ((loop (lambda (tree)
                               (if (leaf? tree)
                                   (call/cc
                                     (lambda (genrest)
                                        (set! generate-leaves
                                          (lambda ()
                                            (genrest '*)))
                                        (caller tree)))
                                   (begin (loop (car tree))
                                           (loop (cadr tree)))))))
               (loop tree)))))
      (lambda ()
        (call/cc (lambda (k)
                   (set! caller k)
                   (generate-leaves)
                   (caller 0)))))))
```

Generators and Coroutines

- Procedures:

  single operation: call

  single stack, stack frame popped upon return

- Generators:

  two operations: suspend and resume

  - assymetric: generator suspends, caller resumes it

  single stack, generator is an "object" that maintains local state variables

  single entry point

- Coroutines:

  one operation: transfer

  - fully symmetric

  When A transfers to B it acts like a:

  - generator suspend wrt A
  - generator resume wrt B

  transfer names who gets control next

  - non stack-like

Can use continuations to model coroutines

Main characteristics:

- *cooperative vs preemptive*
- *scheduling of coroutines determined by application logic, not runtime*
- *can express concurrency but not parallelism*

Threads and processes

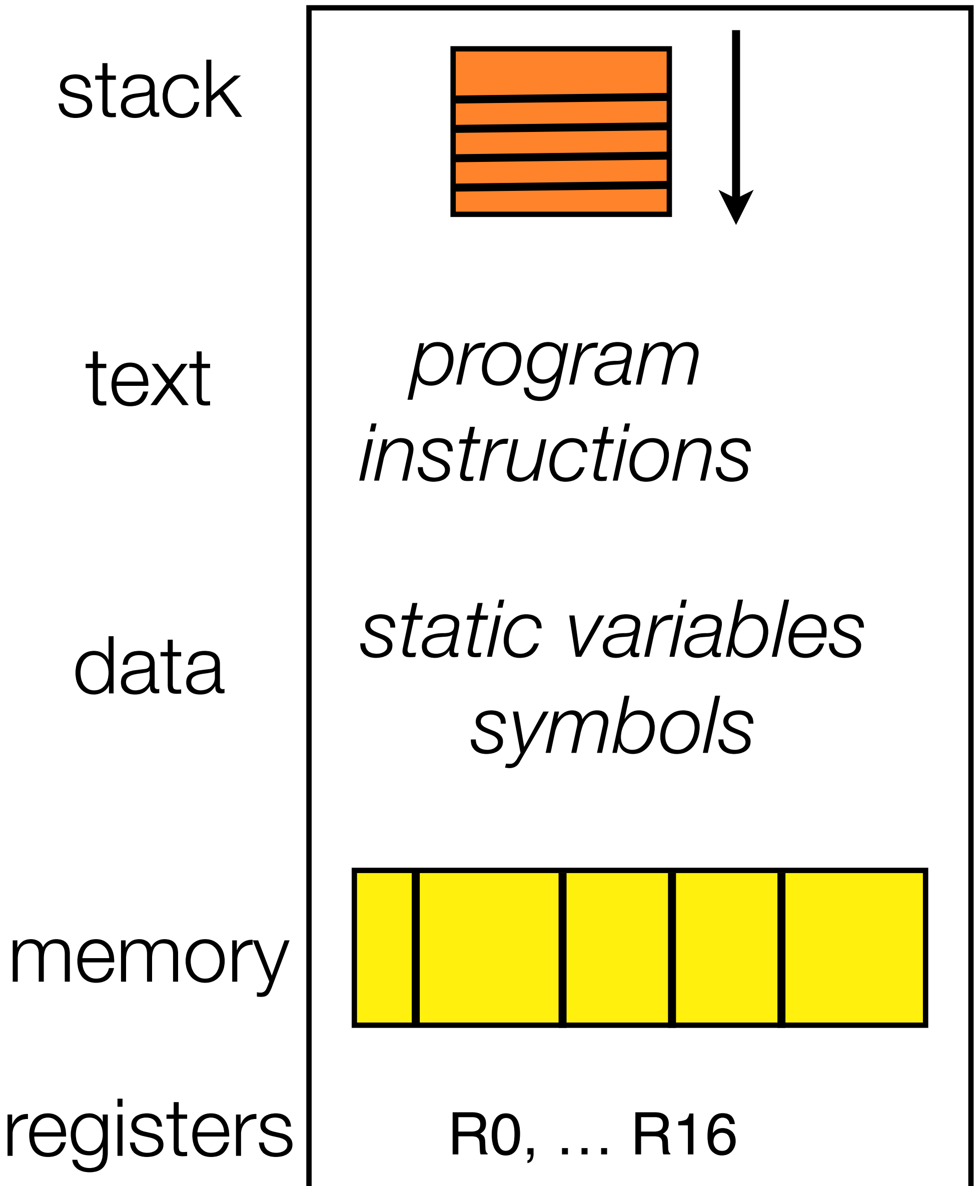
Thread: an independent unit of execution that shares resources with other threads

Process: an independent unit of execution isolated from all other processes and shares no resources with them

Resources:

- Registers
- Stack
- Heap
- Locks
- File descriptors
- Shared libraries
- Program instructions

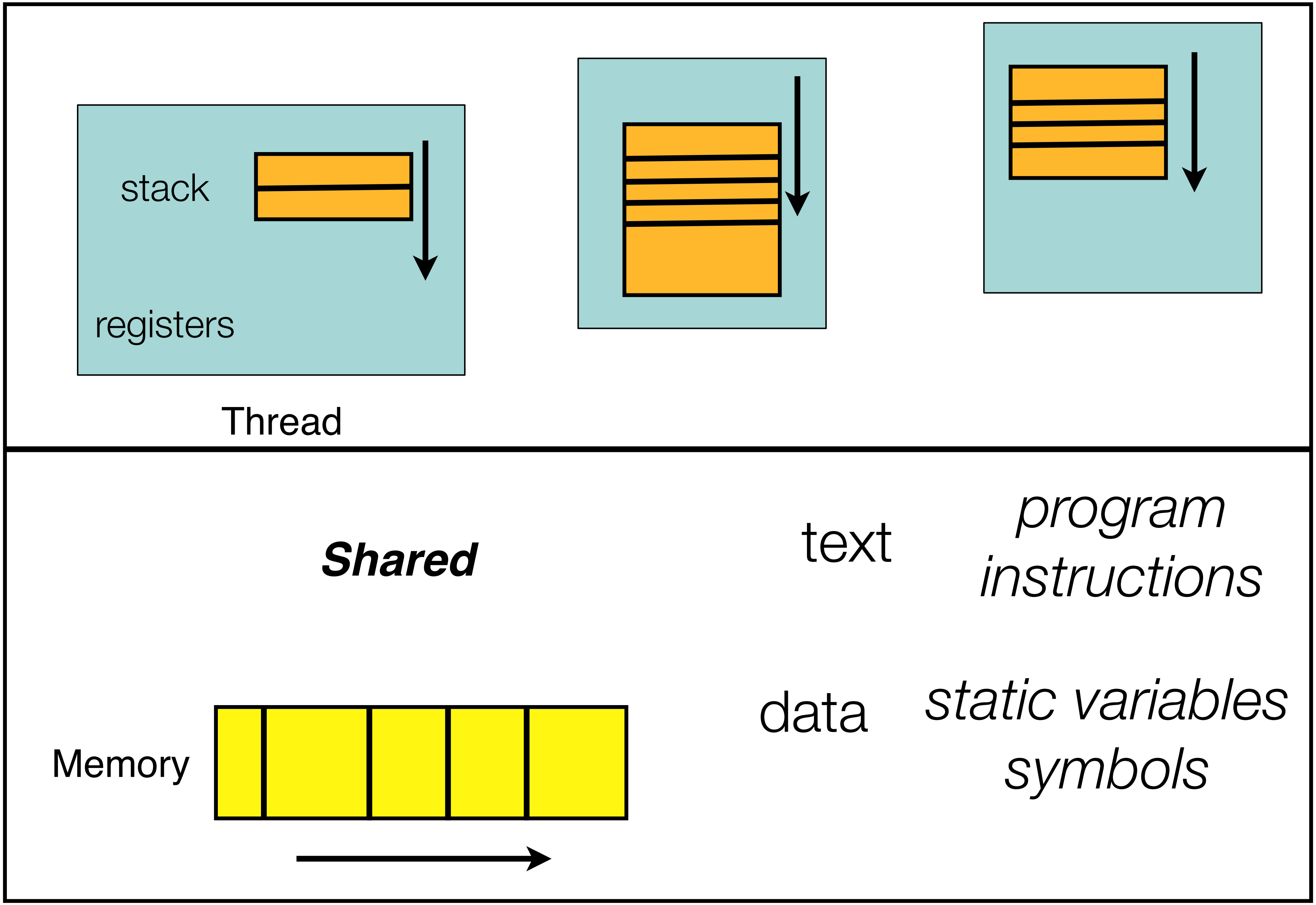
A Process



Shared Objects

Files
Locks
Sockets
Signals
Handlers

Threads Within a Process



Fork System Call

Most operating systems (e.g., Linux) provide a fork() system call

- Spawns a new child process (in a separate address space), identical to the parent except for a different process id.
- Communication typically through file descriptors and system calls

```
#include <unistd.h>
#include <sys/wait.h>
#include <signal.h>
#include <iostream>
using namespace std;


int main(){
   pid_t pid;
   int status, died, val;
   switch(pid=fork()){
   case -1: cout << "can't fork\n";
            exit(-1);
   case 0 : cout << "   I'm the child of PID " << getppid() << ".\n";
            cout << "   My PID is " <<  getpid() << endl;
            cout << "   What is the exit value you wish to pass to the parent?\n ";
            cin >> val;
            sleep(2);
            exit(val);
   default: cout << "I'm the parent.\n";
            cout << "My PID is " <<  getpid() << endl;
             died= wait(&status);
            cout << "The child, pid=" << pid << ", has returned " << WEXITSTATUS(status) << endl;
   }
}
```

Exec System Call

Can have child process execute a different image than parent using exec.

```
#include <unistd.h>
#include <sys/wait.h>
#include <iostream>
using namespace std;

int main(){
   pid_t pid;
   int status, died;
   switch(pid=fork()){
   case -1: cout << "can't fork\n";
            exit(-1);
   case 0 : execl("/bin/date","date",0); // this is the code the child runs
   default: died= wait(&status); // this is the code the parent runs
   }
}
```

Processes

Advantages

- Operating system responsible for scheduling and resource management
  - simplifies application responsibility
- Each process executes within its own address space
  - additional protection and security
    - error or vulnerability in a process does not immediately compromise integrity of other processes
- Different processes can run different applications

Disadvantages

- More heavyweight:
  - operating system involvement in creation and destruction
  - inter-process communication more expensive
    - less useful when there is lots of communication among tasks
    - costs vary among different operating systems
    - reliant on provided OS services
- less control
  - scheduling and management controlled by operating system

Threads

Exists within a process

- But, independent control flow
- share common process resources (like heap and file descriptors)
  - changes made by one thread visible to others
  - pointers have meaning across threads
  - two threads can concurrently read and write to the same memory location

Maintain their own stack pointer

Local register file

Pending and blocked signals

Scheduling still managed by the operating system

Threads and Processes

Critical distinction between using processes and threads:

- References (i.e., locations) have meaning between threads
- They are interpreted independently between processes

  Sharing state among processes requires special care

  - memory-mapped regions, devices, etc.

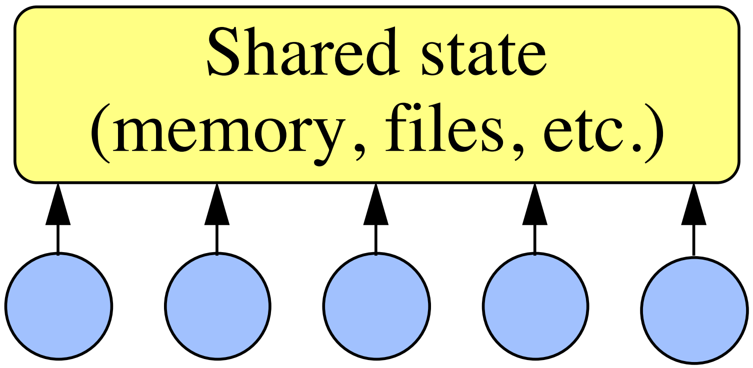
The state (resources) needed to execute a thread is managed directly by a process

Alternative: User-level or "Green" threads

- managed by an underlying runtime or virtual machine

Threads

An initial model



Mediation among threads through explicit synchronization (locks)

Scheduling is asynchronous

- Very flexible
- But, care is needed to deal with
  deadlock, livelock, ensure fairness, etc.

Desired Structure

Programs can be decomposed into discrete independent tasks

The points where they overlap should be easily discerned and amenable for protection

Three basic structures

- master-worker
  - Master coordinates activities of workers and collects results
  - Workers perform (mostly independent) tasks concurrently
    - what happens when work is not independent
- result-oriented
  - Output of a computation in the form of a data structure
  - Each concurrent task fills in one part of the structure
- pipeline-oriented
  - assembly line model
    - each task specialized to one task, forwarding its output to the next specialized unit

Issues

Synchronization

- How should two threads manage communication?
  - Shared Memory
  - Use a lock
- What happens if we forget, or we use the wrong lock?
  - Race conditions
  - Aggressive synchronization can lead to deadlock

Architectural abstraction

Shared memory

- Every thread can observe actions of other threads on non-thread-local data (e.g., heap)
- Data visible to multiple threads must be protected (synchronized) to ensure the absence of data races
  - A data race consists of two concurrent accesses to the same shared data by two separate threads, at least one of which is a write

Thread safety

- Suppose a program creates n threads, each of which calls the same procedure found in some library
- Suppose the library modifies some global (shared) data structure
- Concurrent modifications to this structure may lead to data corruption

Alternative Model …

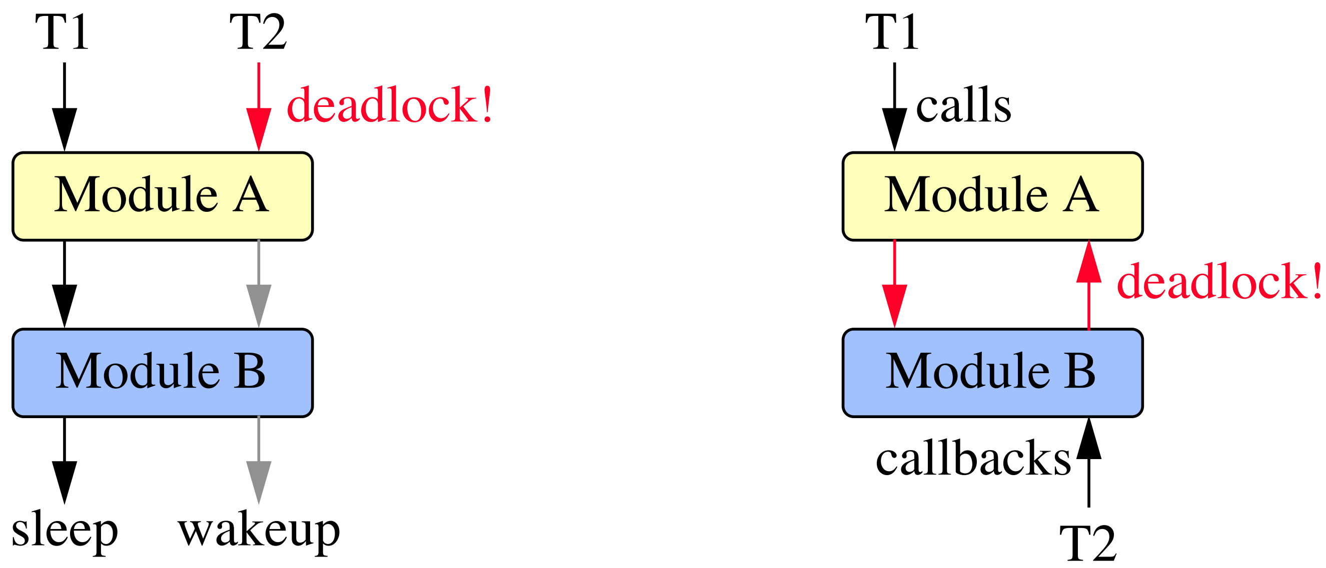
Message-passing

- Threads communicate via messages
- Data found on messages can either be
  - copies - typically in distributed memory environments or
  - references - typical for shared-memory systems
- Senders and receivers can coordinate message delivery either
  - synchronously: sender blocks until receiver available
  - asynchronously: sender buffers data and proceeds even if receiver not available
- Don’t have synchronization issues found in shared-memory concurrency, but
  - program structure more complex and different from sequential version
  - data consistency is still an issue (multiple copies of the same object)
  - replace data race concerns with deadlock concerns
    - receivers block if there is no message available to read

Composability

Threads that communicate using locks can easily break abstractions

- Lower layers in the software stack may need to know behavioral properties of higher layers, and vise versa



Locking

- Suppose that two threads increment a shared memory location:

```
x = 0
```

Thread 1	Thread 2
`tmp1 = *x;`	`tmp2 = *x;`
`*x = tmp1 + 1;`	`*x = tmp2 + 1;`

- If both threads read `0`, (even in an ideal world) `x == 1` is possible:

```
tmp1 = *x; tmp2 = *x; *x = tmp1 + 1; *x = tmp2 +1
```

Locking

- **Lock** and **unlock** are primitives that prevent the two threads from interleaving their actions.

`x = 0`

`lock();`
`tmp1 = *x;`
`*x = tmp1 + 1;`
`unlock();`
	`lock();`
	`tmp2 = *x;`
	`*x = tmp2 + 1;`
	`unlock();`

- In this case, the interleaving below is forbidden, and we are guaranteed that `x == 2` at the end of the execution.

FORBIDDEN

`tmp1 = *x;` `tmp2 = *x;` `*x = tmp1 + 1;` `*x = tmp2 +1`

Subtleties

Lazy initialization

Replace

```
  int x = computeInitValue();      // eager initialization
  …                                // clients refer to x
```

with:

```
int xValue() {
  static int x = computeInitValue(); // lazy initialization
  return x;
} ...                          // clients refer to xValue()
```

Lazy Initialization

A possible implementation of this behavior:

```
class Singleton {
public:
  static Singleton *instance (void) {
    if (instance_ == NULL)
      instance_ = new Singleton;
    return instance_;
  }
…                                  // other methods omitted
private:
  static Singleton *instance_;   // other fields omitted
};

…
Singleton::instance () -> method ();
```

But, this is incorrect in the presence of concurrently executing threads. Why?

Double-check locking

An alternative implementation:

```
class Singleton {
public:
  static Singleton *instance (void) {
   // First check
   if (instance_ == NULL) {
     // Ensure serialization
     Guard<Mutex> guard (lock_);
     // Double check
     if (instance_ == NULL)
       instance_ = new Singleton;
   }
   return instance_;
  }
private: [..]
};
```

grab a lock only if the instance is nil and re-check its status

Double-check locking

Problem:

The instruction

```
instance_ = new Singleton
```

does three things:

1) allocate memory

2) construct the object

3) assign to `instance_` the address of the memory

Not necessarily in this order! For example:

```
instance_ =                              // 3
  operator new(sizeof(Singleton)); // 1
new (instance_) Singleton                // 2
```

If this code is generated, the order is 1,3,2.

Double-check locking

Solution is still broken ...

```
if (instance_ == NULL) {                    // Line 1
   Guard<Mutex> guard (lock_);
   if (instance_ == NULL) {
     instance_ =
        operator new(sizeof(Singleton));   // Line 2
     new (instance_) Singleton; }}
```

Thread 1:
executes through Line 2 and is suspended; at this point, `instance_` is non-NULL, but no singleton has been constructed.

Thread 2:
executes Line 1, sees `instance_` as non-NULL, returns, and dereferences the pointer returned by `Singleton` (i.e., `instance_`).

Thread 2 attempts to reference an object that is not there yet!

Need to instruct the compiler to issue a different code sequence for this pattern - relevant only in the presence of concurrency