CS49K Programming Languages

Chapter 13: Concurrency

LECTURE 15: SYNCHRONIZATION

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Motivation for Concurrent Programming
- Types of Concurrent Programming:
  - Single-Threading
  - Multi-Threading
  - Multi-Processing
  - Multi-Programming
- Shared Locks
- Synchronization Schemes

Motivation

SECTION 1

Parallelism

- Vector Parallelism (Op-Level, Loop-Level)
- Instruction Level Parallelism (ILP)
- Loop-Level Parallelism (Loop-Unrolling)
- **Task-Level Parallelism (Multi-Threading/Multi-Processing)**
- Program-Level Parallelism (Distributed Computing/Message Passing)

Topics in Concurrent Programming

- Hardware and Memory Model (Computer Architecture)
- Thread, Process, and Program Management (OS and Language)
- Data Synchronization
- Performance Optimization (Utilization of Parallelism)
- Race, Deadlock Issues

Background and Motivation

- A PROCESS or THREAD is a potentially-active execution context
- Classic von Neumann (stored program) model of computing has single thread of control
- Parallel programs have more than one
- A process can be thought of as an abstraction of a physical PROCESSOR

Background and Motivation

Multithreading, Multiprogramming, and Distributed Computing

- Processes/Threads can come from
  - multiple CPUs
  - kernel-level multiplexing of single physical machine
  - language or library level multiplexing of kernel-level abstraction
- They can run
  - in true parallel
  - unpredictably interleaved
  - run-until-block
- Most work focuses on the first two cases, which are equally difficult to deal with

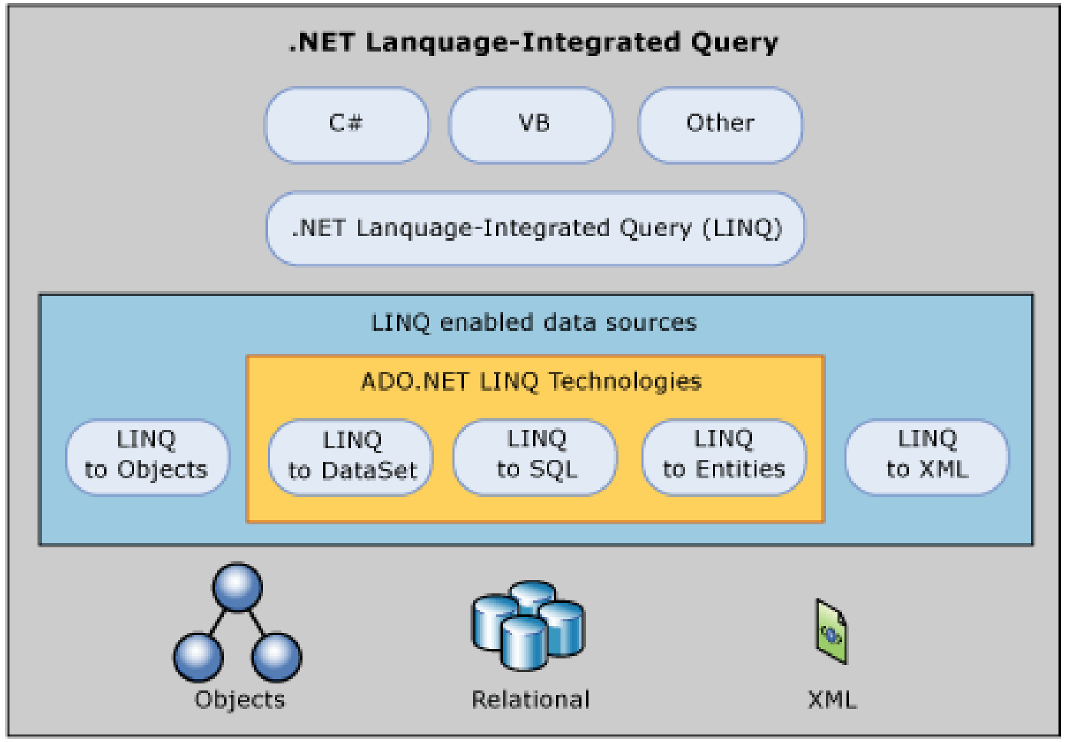
Background and Motivation

- Two main classes of programming notation
  - synchronized access to shared memory
  - message passing between processes that don't share memory
- Both approaches can be implemented on hardware designed for the other, though shared memory on message-passing hardware tends to be slow

Levels of Abstraction

Instruction, Method, Task, Program and System

```
Parallel.For(0, 100, i => { A[i] = foo(A[i]); } );
```

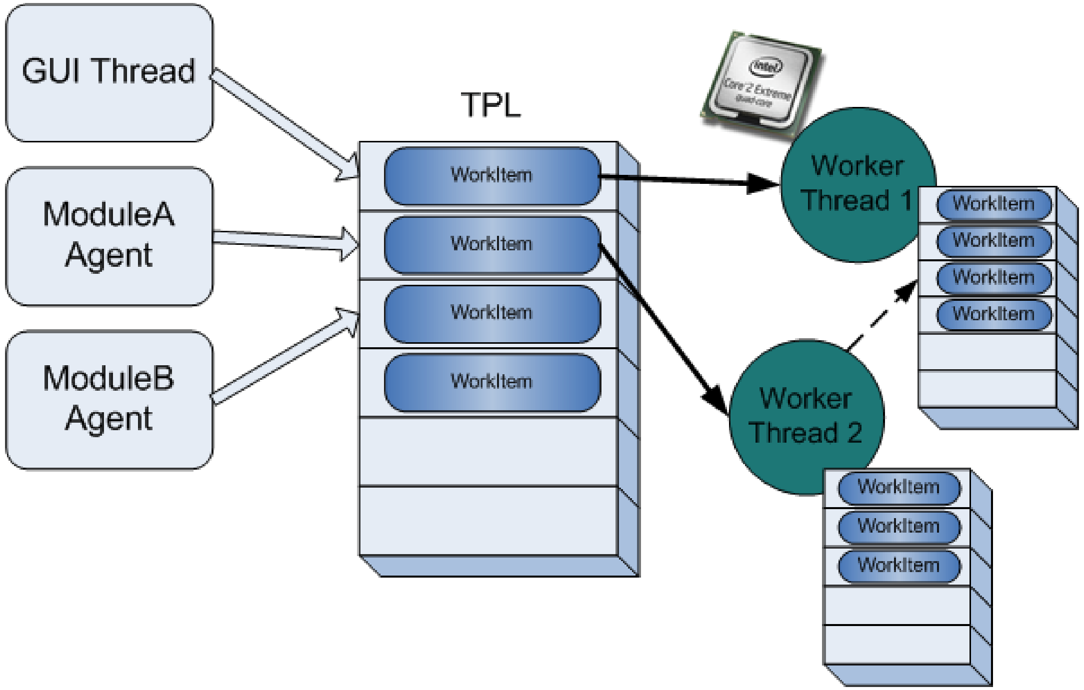


Black Box Parallel Libraries (LINQ)

Levels of Abstraction

Instruction, Method, Task, Program and System

```
Parallel.For(0, 100, i => { A[i] = foo(A[i]); } );
```



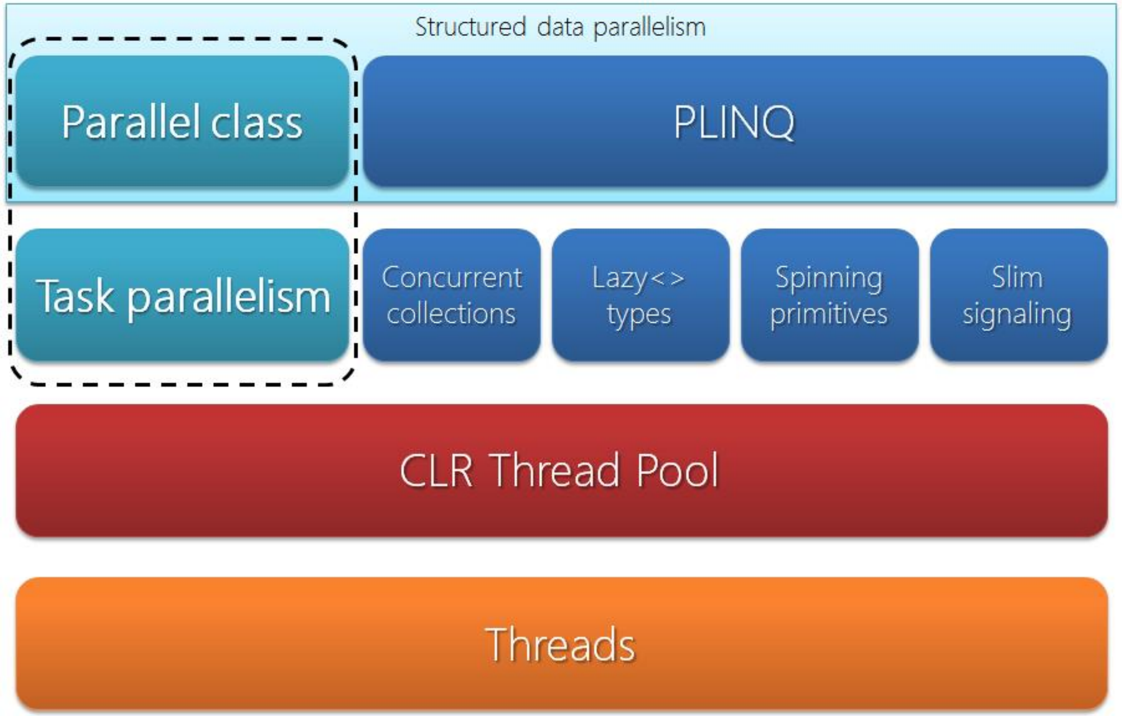
Task Parallel Library (TPL)

Levels of Abstraction

Instruction, Method, Task, Program and System

```
Parallel.For(0, 100, i => { A[i] = foo(A[i]); } );
```

Parallel FX



C# Current Environment

Background and Motivation

Race conditions

- A race condition occurs when actions in two processes are not synchronized and program behavior depends on the order in which the actions happen
- Race conditions are not all bad; sometimes any of the possible program outcomes are ok (e.g. workers taking things off a task queue)

Background and Motivation

Race conditions

- If the instructions interleave roughly as shown, both threads may load the same value of zero count, both may increment it by 1, and both may store the (only 1 greater) value back into zero count. **The result may be 1 less than what we expect.**

```
int zero_count;
public static int foo(int n) {
    int rtn = n - 1;
    if (rtn == 0) zero_count++;
    return rtn;
}
```

Thread 1	Thread 2
...	
r1 := zero_count	...
r1 := r1 + 1	r1 := zero_count
zero_count := r1	r1 := r1 + 1
...	zero_count := r1
	...

Background and Motivation

Race conditions

- Race conditions (we want to avoid race conditions):
  - Suppose processors A and B share memory, and both try to increment variable X at more or less the same time
  - Very few processors support arithmetic operations on memory, so each processor executes
    - **LOAD X**
    - **INC**
    - **STORE X**
  - Suppose **X** is initialized to 0. If both processors execute these instructions simultaneously, what are the possible outcomes?
    - could go up by one or by two

Background and Motivation

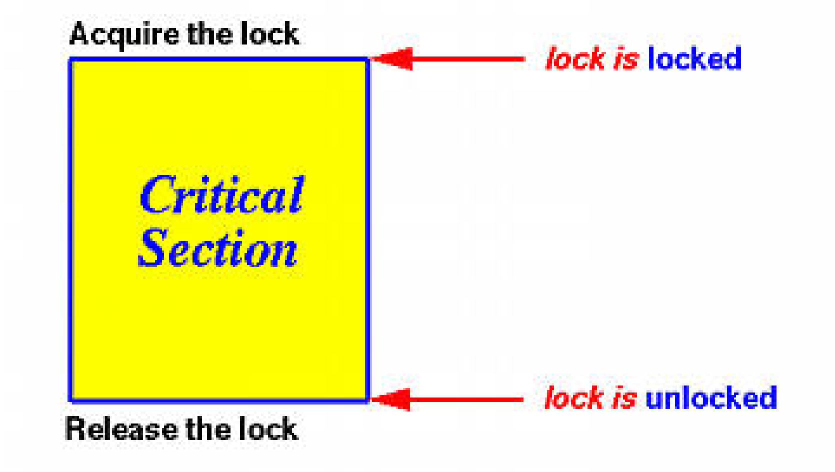
Synchronization

- SYNCHRONIZATION is the act of ensuring that events in different processes happen in a desired order
- Synchronization can be used to eliminate race conditions
- In our example we need to synchronize the increment operations to enforce MUTUAL EXCLUSION on access to X
- Most synchronization can be regarded as either
  - Mutual exclusion (making sure that only one process is executing a CRITICAL SECTION [touching a variable, for example] at a time), or as
  - CONDITION SYNCHRONIZATION, which means making sure that a given process does not proceed until some condition holds (e.g. that a variable contains a given value)

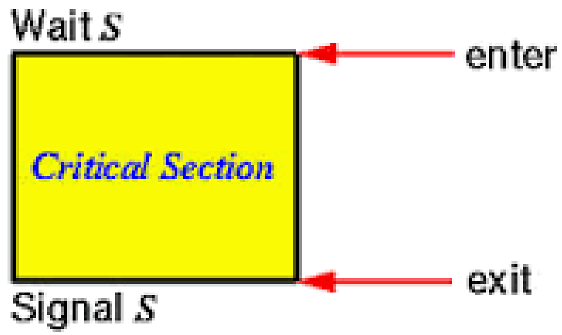
Background and Motivation

Critical Section (Critical Region): Atomic Code Block

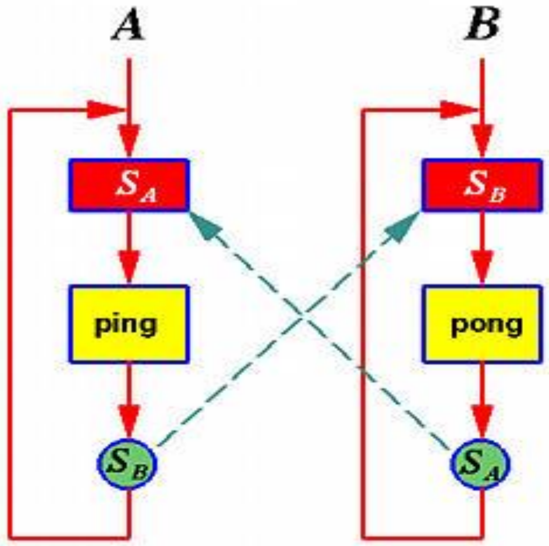
Lock Unlock



Wait-Signal



Semaphore



Condition Variable, Monitor, Critical Condition Region, STM

Background and Motivation

- One might be tempted to think of mutual exclusion as a form of condition synchronization (the condition being that nobody else is in the critical section), but it isn't
  - The distinction is basically *existential vs. universal* quantification
    - Mutual exclusion requires multi-process consensus
- We do NOT in general want to over-synchronize
  - That eliminates parallelism, which we generally want to encourage for performance
- Basically, we want to eliminate "bad" race conditions, i.e., the ones that cause the program to give incorrect results

Background and Motivation

Historical development of shared memory ideas

- To implement synchronization you have to have something that is ATOMIC
  - that means it happens all at once, as an indivisible action
  - In most machines, reads and writes of individual memory locations are atomic (note that this is not trivial; memory and/or busses must be designed to arbitrate and serialize concurrent accesses)
  - In early machines, reads and writes of individual memory locations were *all* that was atomic
- To simplify the implementation of mutual exclusion, hardware designers began in the late 60's to build so-called read-modify-write, or fetch-and-phi, instructions into their machines

Multi-threaded Programs

SECTION 2

Coroutine

- Coroutines are computer program components that generalize subroutines for **non-preemptive multitasking**, by allowing multiple entry points for suspending and resuming execution at certain locations. Coroutines are well-suited for implementing more familiar program components such as cooperative tasks, exceptions, event loop, iterators, infinite lists and pipes.

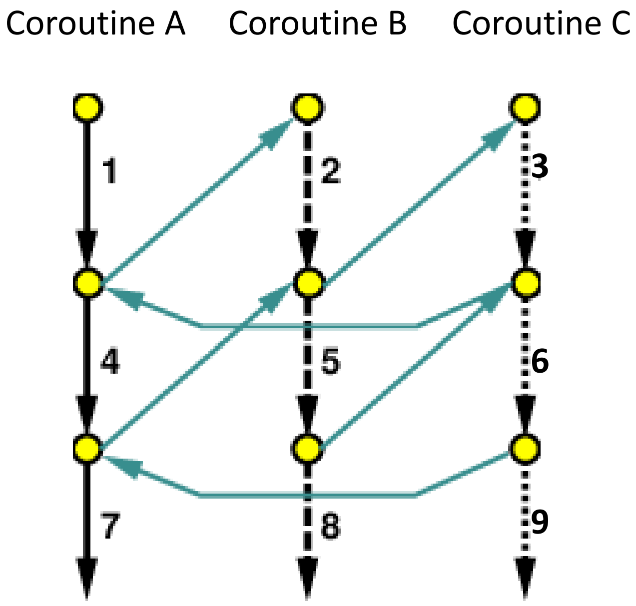
Producer/Consumer Model

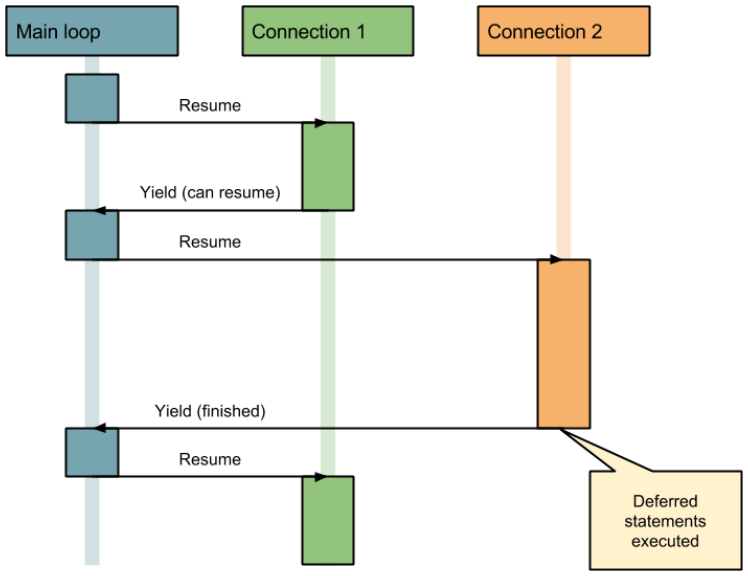
Coroutine (Discrete-Event Simulation for Concurrency)

```
var q := new queue
```

```
coroutine produce
    loop
        while q is not full
            create some new items
            add the items to q
        yield to consume
```

```
coroutine consume
    loop
        while q is not empty
            remove some items from q
            use the items
        yield to produce
```

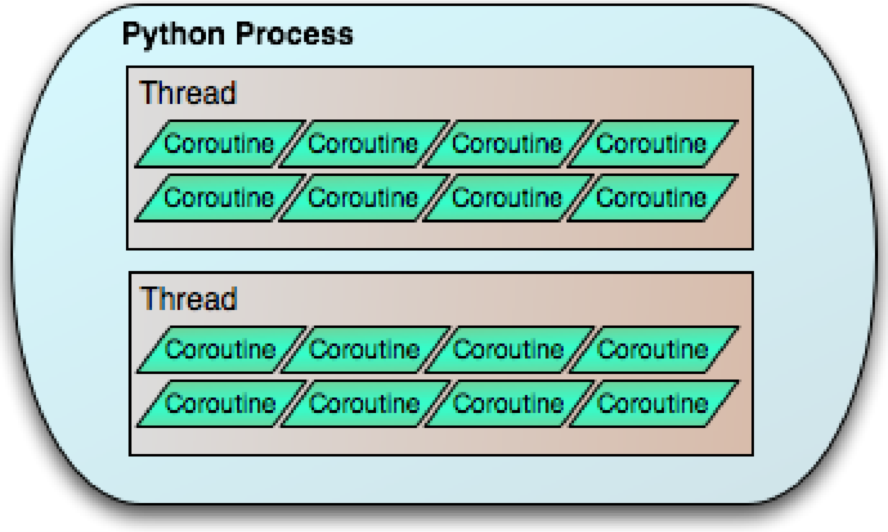


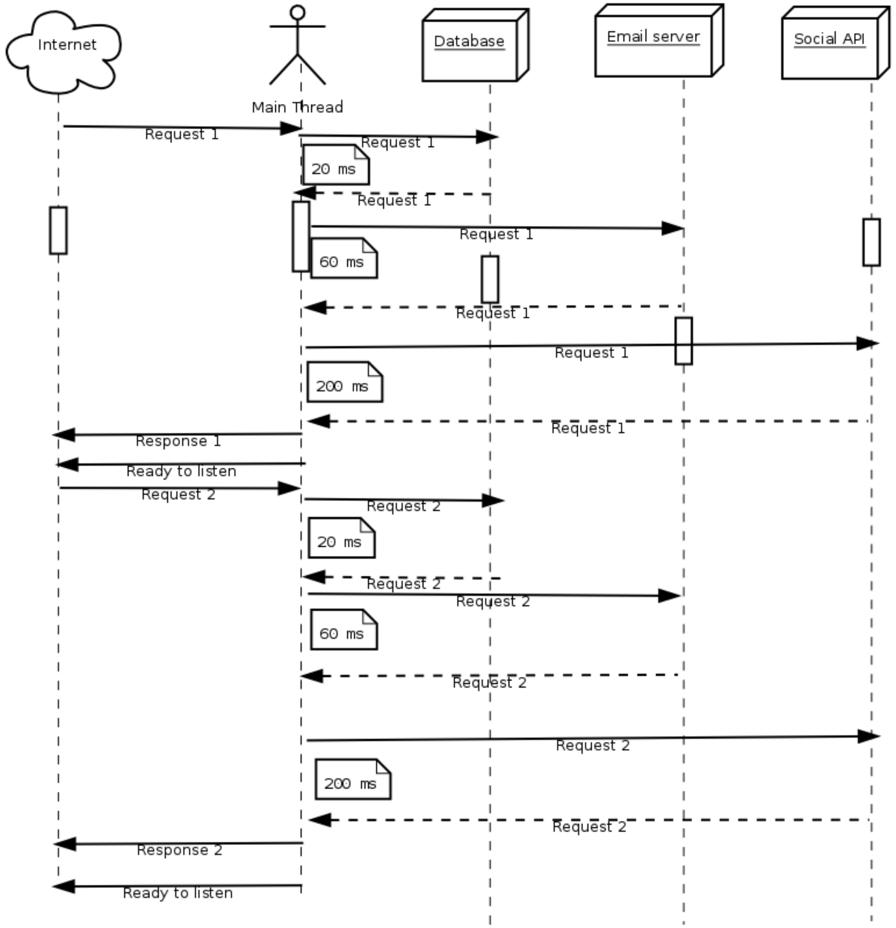


Achieve time-shared concurrency in centralized or distributed environment.

Coroutine for Asynchronous Control Flow

Web-Applications, Event-Loop, Exceptions, Cooperative Tasks, and etc.





Web-Brower Parser

- Thread-based code from a hypothetical Web browser.
- To first approximation, the parse page subroutine is the root of a recursive descent parser for HTML.
- In several cases, however, the actions associated with recognition of a construct (background, image, table, frameset) proceed concurrently with continued parsing of the page itself.
- In this example, concurrent threads are created with the fork operation. An additional thread would likely execute in response to keyboard and mouse events.

```
procedure parse_page(address : url)
    contact server, request page contents
    parse_html_header
    while current_token in {"<p>", "<h1>", "<ul>", . . . ,
            "<background", "<image", "<table", "<frameset", . . . }
        case current_token of
            "<p>"      :  break_paragraph
            "<h1>"     :  format_heading; match("</h1>")
            "<ul>"     :  format_list; match("</ul>")
            . . .
            "<background" :
                          a : attributes := parse_attributes
                          fork render_background(a)
            "<image" :    a : attributes := parse_attributes
                          fork render_image(a)
            "<table"   :  a : attributes := parse_attributes
                          scan forward for "</table>" token
                          token_stream s :=. . .     – – table contents
                          fork format_table(s, a)
            "<frameset" :
                          a : attributes := parse_attributes
                          parse_frame_list(a)
                          match("</frameset>")
        . . .
    . . .
procedure parse_frame_list(a1 : attributes)
    while current_token in {"<frame", "<frameset", "<noframes>"}
        case current_token of
            "<frame" :  a2 : attributes := parse_attributes
                        fork format_frame(a1, a2)
```

The Dispatch Loop Alternative

Dispatch-Loop Browser

- **Data structures** associated with the dispatch loop keep track of all the tasks the browser has yet to complete.
- It must also identify the various subtasks of the page (images, tables, frames, etc.) so that we can find them all and reclaim their state if the user clicks on a “stop” button.
- To guarantee good interactive response, we must make sure that no sub-action of continue task takes very long to execute. (Very Busy)
- The principal problem with a dispatch loop—beyond the complexity of sub-dividing tasks and saving state—is that it hides the algorithmic structure of the program.

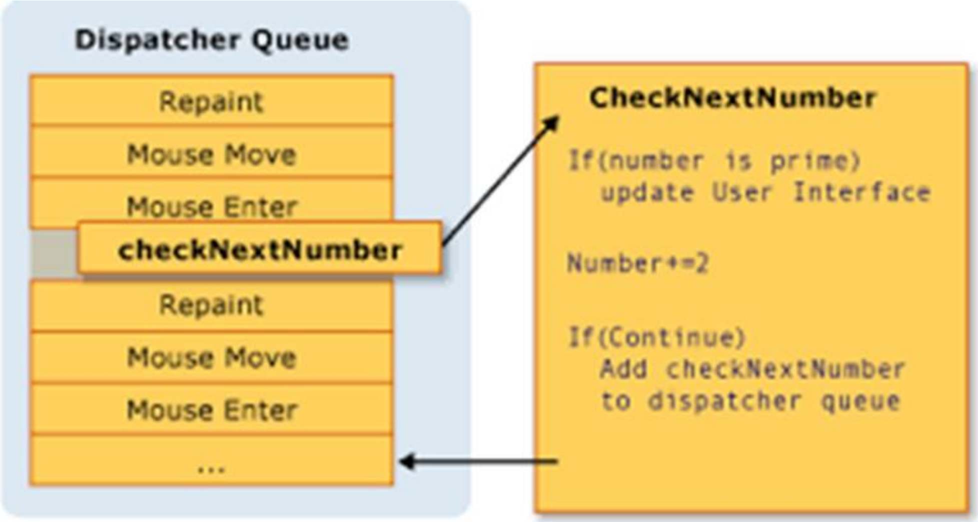
The Dispatch Loop Alternative

Dispatch-Loop Browser

- Every distinct task could be described elegantly with standard control-flow mechanisms, if not for the fact that we must return to the top of the dispatch loop at every delay-inducing operation. In effect, the dispatch loop turns the program “inside out,” making the management of tasks explicit and the control flow within tasks implicit.
- The resulting complexity is similar to what we encountered when trying to enumerate a recursive set with iterator objects in Section 6.5.3, only worse. Like true iterators, a thread package turns the program “right side out,” making the management of tasks (threads) implicit and the control flow within threads explicit.

The Dispatch Loop Alternative

Dispatch-Loop Browser



Busy loop

```
type task_descriptor = record
    -- fields in lieu of thread-local variables, plus control-flow information
    ...
ready_tasks : queue of task_descriptor
...
procedure dispatch
    loop
        -- try to do something input-driven
        if a new event E (message, keystroke, etc.) is available
            if an existing task T is waiting for E
                continue_task(T, E)
            else if E can be handled quickly, do so
            else
                allocate and initialize new task T
                continue_task(T, E)
        -- now do something compute bound
            if ready_tasks is nonempty
                continue_task(dequeue(ready_tasks), 'ok')

procedure continue_task(T : task, E : event)
    if T is rendering an image
        and E is a message containing the next block of data
            continue_image_render(T, E)
    else if T is formatting a page
        and E is a message containing the next block of data
            continue_page_parse(T, E)
    else if T is formatting a page
        and E is 'ok'        -- we're compute bound
            continue_page_parse(T, E)
    else if T is reading the bookmarks file
        and E is an I/O completion event
            continue_goto_page(T, E)
    else if T is formatting a frame
        and E is a push of the "stop" button
            deallocate T and all tasks dependent upon it
    else if E is the "edit preferences" menu item
        edit_preferences(T, E)
    else if T is already editing preferences
        and E is a newly typed keystroke
            edit_preferences(T, E)
    ...
```

Memory Coherence in Concurrent Environment

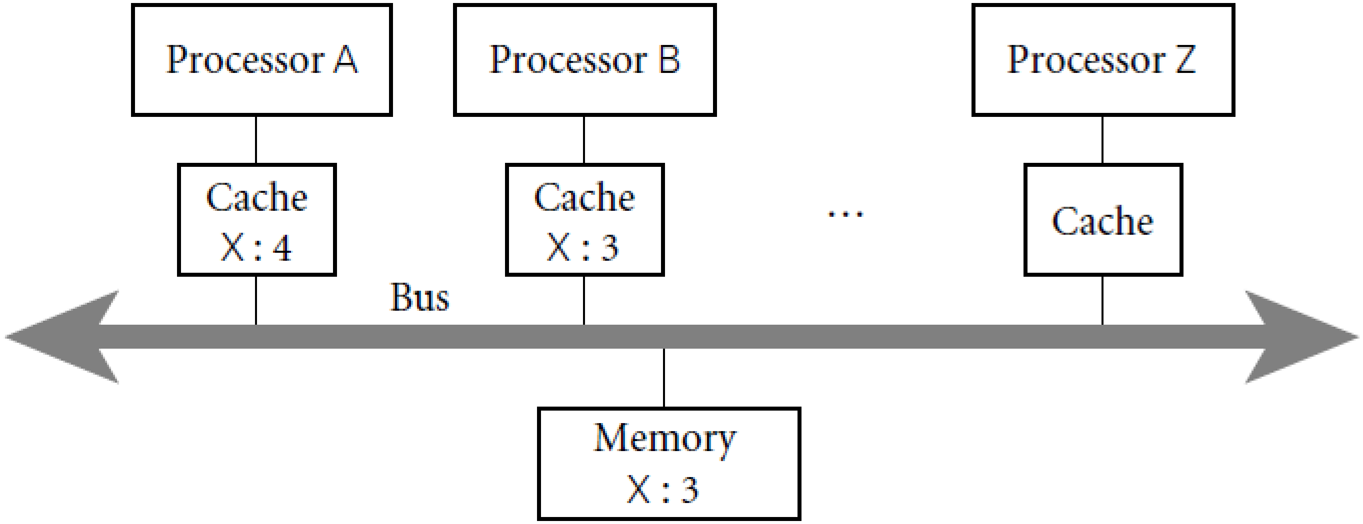
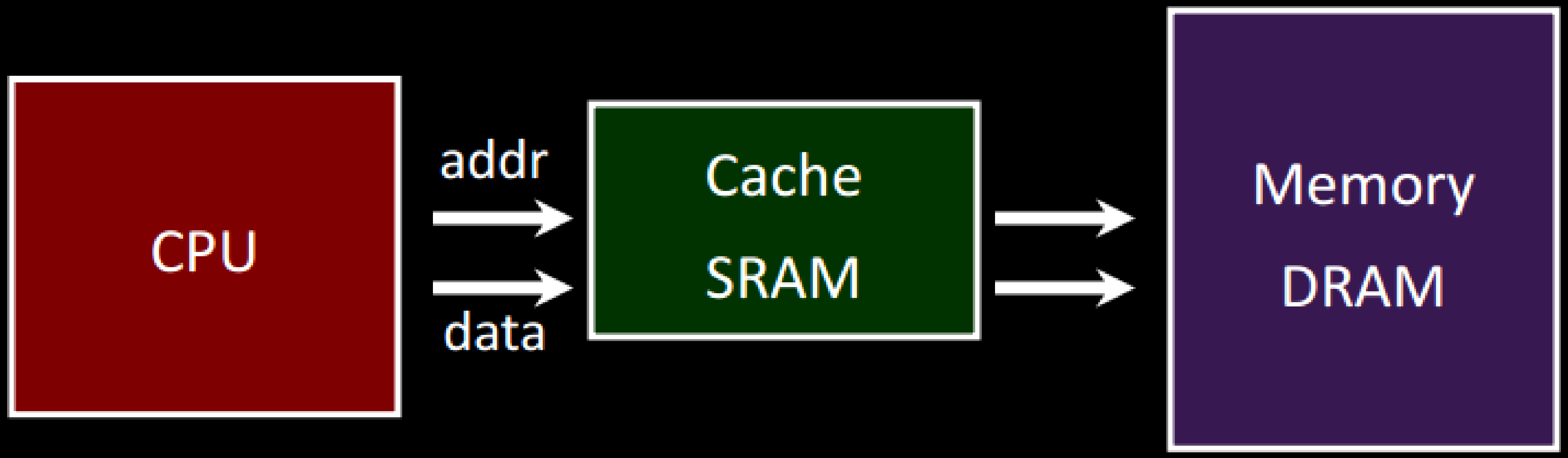


Figure 12.3 **The cache coherence problem for shared-memory multiprocessors.** Here processors A and B have both read variable X from memory. As a side effect, a copy of X has been created in the cache of each processor. If A now changes X to 4 and B reads X again, how do we ensure that the result is a 4 and not the still-cached 3? Similarly, if Z reads X into its cache, how do we ensure that it obtains the 4 from A's cache instead of the stale 3 from memory?



If data is already in the cache...

No-Write

- writes invalidate the cache and go directly to memory

Write-Through

- writes go to main memory and cache

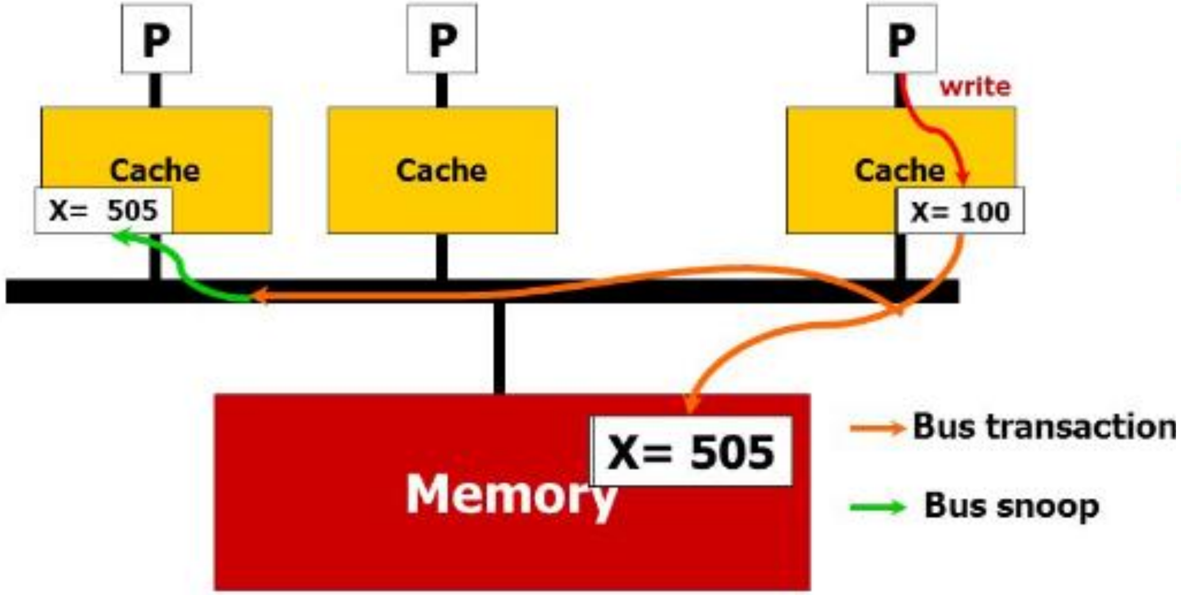
Write-Back

- CPU writes only to cache
- cache writes to main memory later (when block is evicted)

Wait until we kick the block out of cache (write-back policy)

Snoopy Cache-Coherence Protocol

Write-Update, Write-invalidate, and Write-back (Write-Update)



Write-Update (Snooping Bus)

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Write broadcast of X	1	1	1
CPU B reads X		1	1	1

Write-back Cache

Other Micro-processor issues have been discussed in other chapters.

Concurrent Programming Fundamentals I

Languages

SECTION 3

Languages and Libraries

Widely used parallel programming systems

	Shared memory	Message passing	Distributed computing
Language	Java, C#		
Extension	OpenMP		Remote procedure call
Library	`pthreads`, Win32 threads	MPI	Internet libraries

There is also a very large number of experimental, pedagogical, or niche proposals for each of the regions in the table (including regions where no system is currently widely used).

Languages

Shared Memory

Java

Scala

HASKELL

OpenMP

Qt

pthreads

Microsoft .NET

Message Passing

golang

MPI

Distributed Systems

ERLANG

SIMGRID

Thread Creation Syntax

- Almost every concurrent system allows threads to be created (and destroyed) dynamically.
- Syntactic and semantic details vary considerably from one language or library to another, but most conform to one of six principal options: **co-begin**, **parallel loops**, **launch-at-elaboration**, **fork (with optional join)**, **implicit receipt**, and **early reply**.
- The first two options delimit threads with special control-flow constructs. The others use syntax resembling (or identical to) subroutines.

Thread Creation Syntax

Co-Begin End (C/C++)

co-begin
    stmt_1
    stmt_2
    ...
    stmt_n
end

```
#pragma omp sections
{
#   pragma omp section
    { printf("thread 1 here\n"); }

#   pragma omp section
    { printf("thread 2 here\n"); }
}
```

In **C**, **OpenMP** directives all begin with #pragma omp. (The # sign must appear in column 1.) Most directives, like those shown here, must appear immediately before a loop construct or a compound statement delimited with curly braces.

All n statements run concurrently.

Thread Creation Syntax

Parallel Loops

Many concurrent systems, including OpenMP, several dialects of Fortran, and the recently announced Parallel FX Library for **.NET**, provide a loop whose iterations EXAMPLE 13.8 are to be executed concurrently. In OpenMP for C, we might say

```
#pragma omp parallel for
for (int i = 0; i < 3; i++) {
    printf("thread %d here\n", i);
}
```

C# with Parallel FX Library

```
Parallel.For(0, 3, i => {
    Console.WriteLine("Thread " + i + " here");
});
```

Fortran 95

```
forall (i=1:n-1)
    A(i) = B(i) + C(i)
    A(i+1) = A(i) + A(i+1)
end forall
```

Thread Creation Syntax

Parallel Loops (C with Reduction in OpenMP)

- Optional "clauses" on parallel directives can specify how many threads to create, and which iterations of the loop to perform in which thread.
- They can also **specify which program variables should be shared by all threads**, and which should be split into a separate copy for each thread.
- It is even possible to specify that a private variable should be reduced across all threads at the end of the loop, using a commutative operator. To sum the elements of a very large vector, for example, one might write

Thread Creation Syntax

Parallel Loops (C with Reduction in OpenMP)

```
double A[N];
...
double sum = 0;
#pragma omp parallel for schedule(static) \
    default(shared) reduction(+:sum)
for (int i = 0; i < N; i++) {
    sum += A[i];
}
printf("parallel sum: %f\n", sum);
```

schedule(static) clause indicates that the compiler should divide the iterations evenly among threads, in contiguous groups.

default(shared) clause indicates that all variables (other than i) should be shared by all threads

reduction(+:sum) clause makes sum an exception: every thread should have its own copyand the copies should be combined (with +) at the end.

Launch-at-Elaboration

Elaborated tasks in Ada

- In several languages, Ada among them, the code for a thread may be declared with syntax resembling that of a subroutine with no parameters.
- When the declaration is elaborated, a thread is created to execute the code. In Ada (which calls its threads tasks) we may write

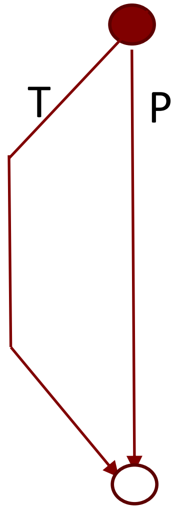
Launch-at-Elaboration

Elaborated tasks in Ada

main program

calls

```
procedure P is
task T is
...
end T;
begin -- P
...
end P;
```



- Task T begins to execute as soon as control enters procedure P. If P is recursive, there may be many instances of T at the same time.
- The main program behaves like an initial default task. When control reaches the end of procedure P, it will wait for the appropriate instance of T (the one that was created at the beginning of this instance of P) to complete before returning.

Thread Creation Syntax

Fork-Join

- In Java one obtains a thread by constructing an object of some class derived from a predefined class called Thread:

```
class ImageRenderer extends Thread {
    ...
    ImageRenderer( args ) {
        // constructor
    }
    public void run() {
        // code to be run by the thread
    }
}
...
ImageRenderer rend = new ImageRenderer( constructor_args );
```

Thread Creation Syntax

Fork-Join

- In Java, the new thread does not begin execution when first created. To start it, the parent (or some other thread) must call the method named start, which is defined in Thread:

```
rend.start();
```

- Start makes the thread runnable, arranges for it to execute its run method, and returns to the caller. The programmer must define an appropriate run method in every class derived from Thread. The run method is meant to be called only by start; programmers should not call it directly, nor should they redefine start.
- There is also a join method: `rend.join();`
- The constructor for a Java thread typically saves its arguments in fields that are later accessed by run. In effect, the class derived from Thread functions as an object closure.

Life Time of Threads

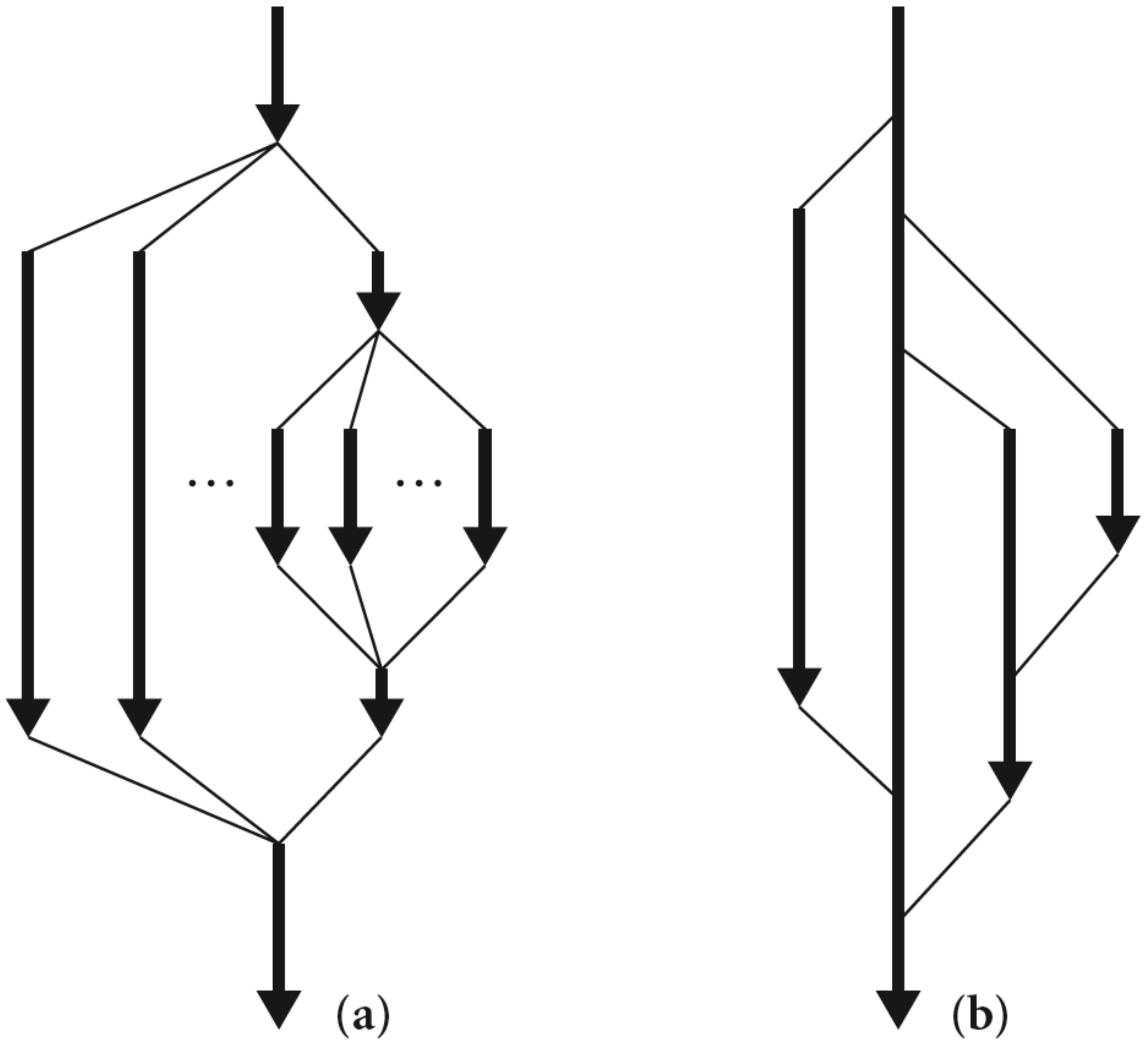
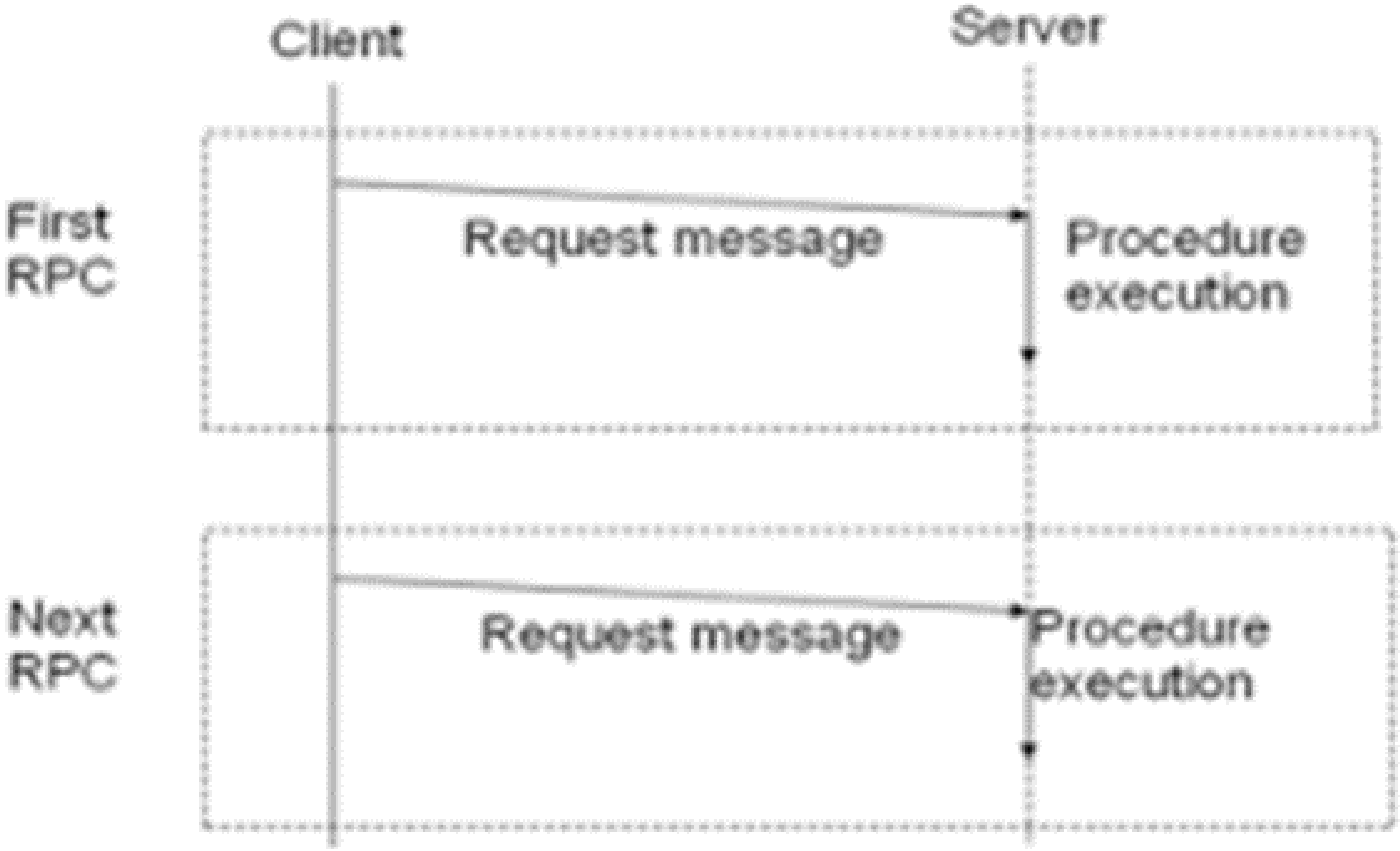


Figure 13.5 Lifetime of concurrent threads. With co-begin, parallel loops, or launch-at-elaboration (a), threads are always properly nested. With fork/join (b), more general patterns are possible.

Thread Creation Syntax

Implicit Receipt

- We have assumed in all our examples so far that newly created threads will run in the address space of the creator. In **RPC** systems it is often desirable to create a new thread automatically in response to an incoming request from some other address space. Rather than have an existing thread execute a receive operation, a server can bind a communication channel to a local thread body or subroutine.
- When a request comes in, a new thread springs into existence to handle it. In effect, the bind operation grants remote **clients** the ability to perform a fork within the **server**'s address space, though the process is often less than fully automatic.


Client
Server
First RPC
Request message
Procedure execution
Next RPC
Request message
Procedure execution

Early Reply

Modeling subroutines with fork/join

- We normally think of sequential subroutines in terms of a single thread, which saves its current context (its program counter and registers), executes the subroutine, and returns to what it was doing before. The effect is the same, however, if we have two threads—one that executes the caller and another that executes the callee. **In this case, the call is essentially a fork/join pair.**
- The caller waits for the callee to terminate before continuing execution. Nothing dictates, however, that the callee has to terminate in order to release the caller; all it really has to do is complete the portion of its work on which result parameters depend.

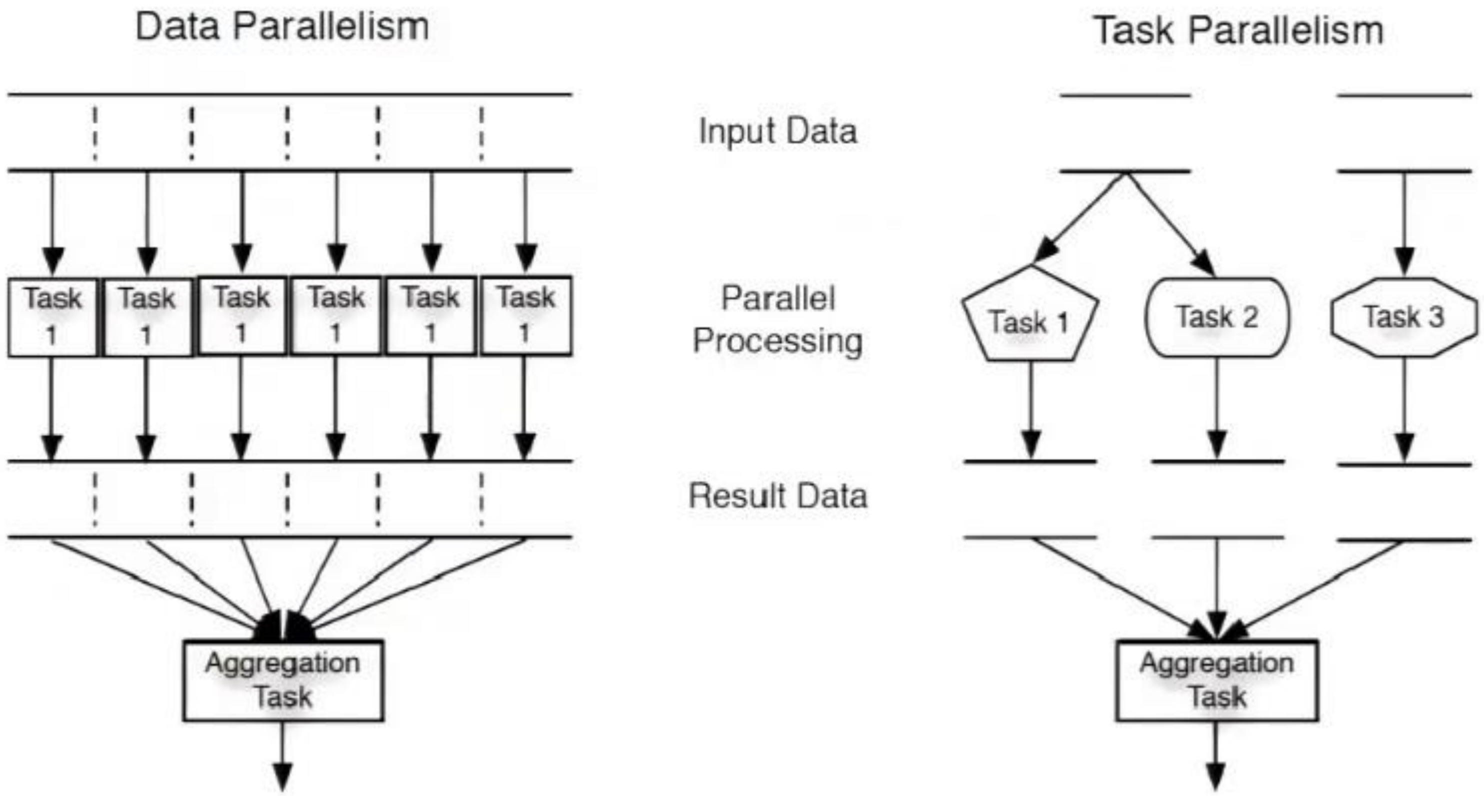
Concurrent Programming Fundamentals II

Implementation

SECTION 4

Multi-Threading

Task Level Parallelism



Vector Processing

Multi-Threading

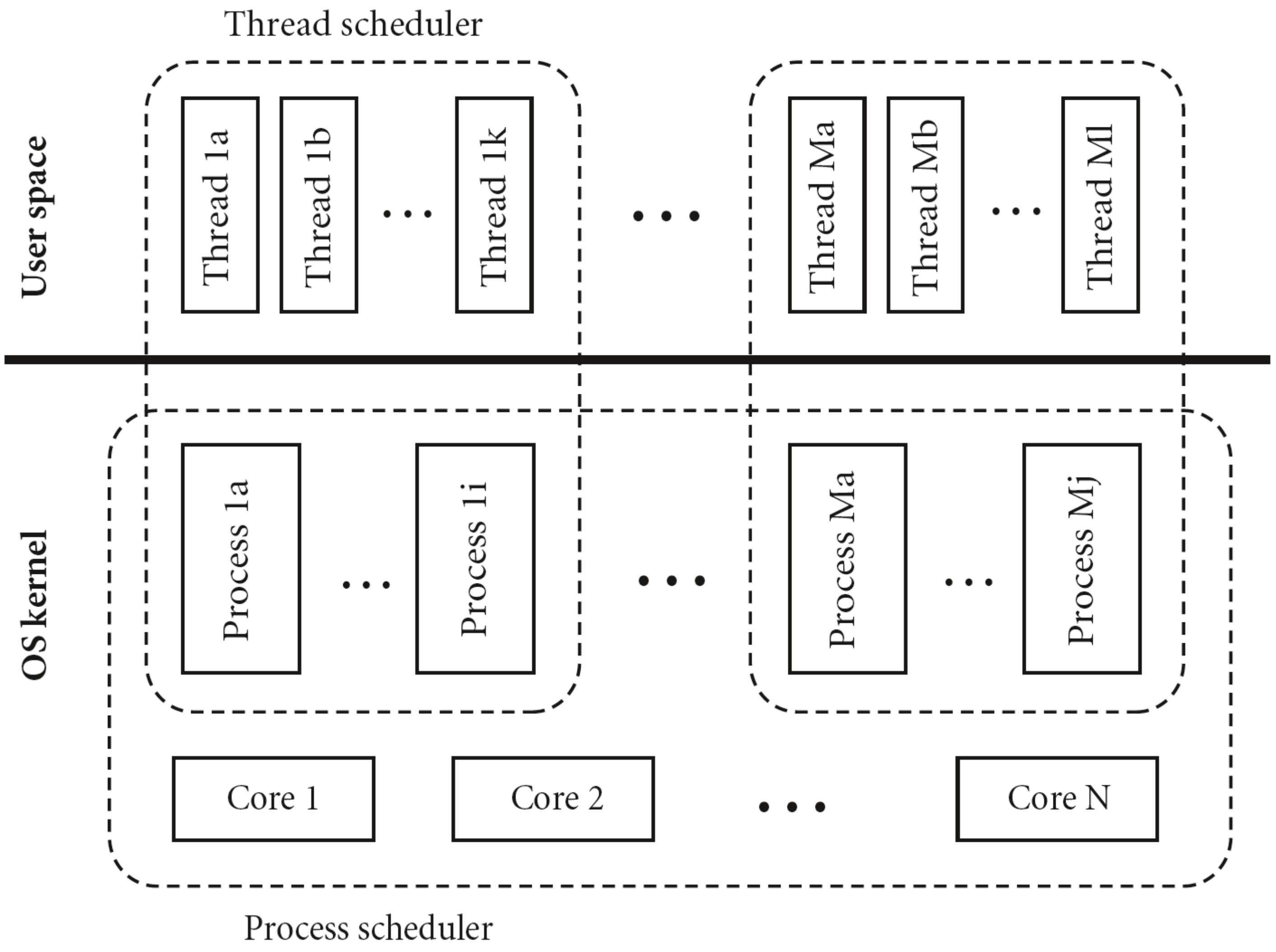


Figure 13.6 Two-level implementation of threads. A thread scheduler, implemented in a library or language run-time package, multiplexes threads on top of one or more kernel-level processes, just as the process scheduler, implemented in the operating system kernel, multiplexes processes on top of one or more physical cores.

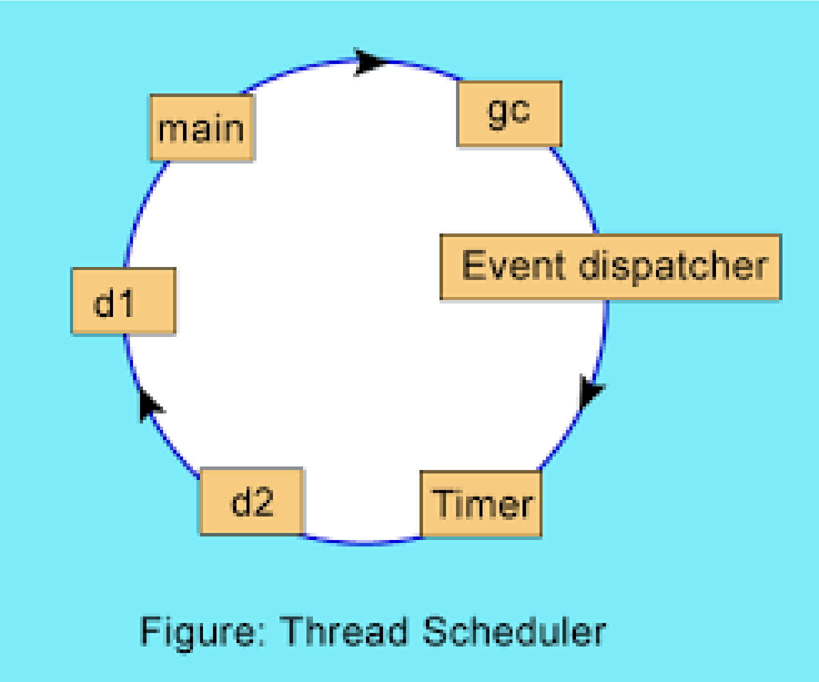
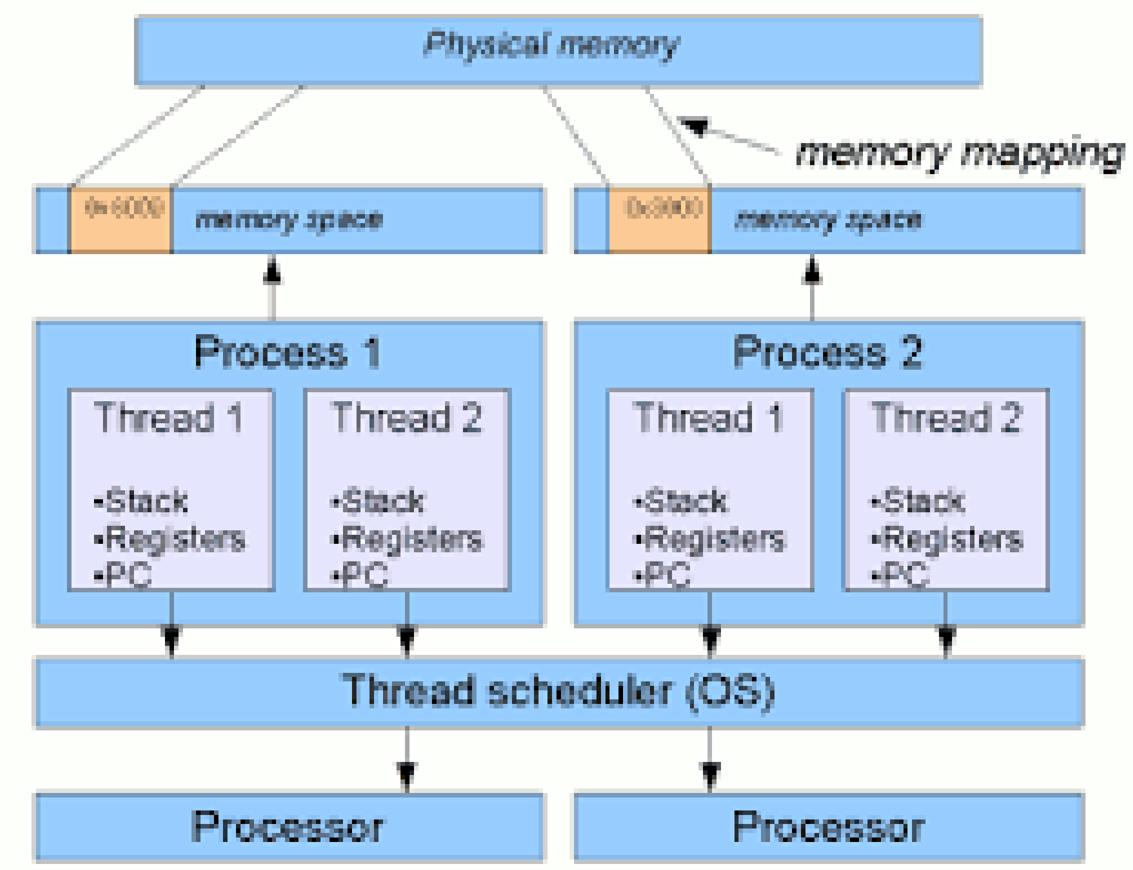


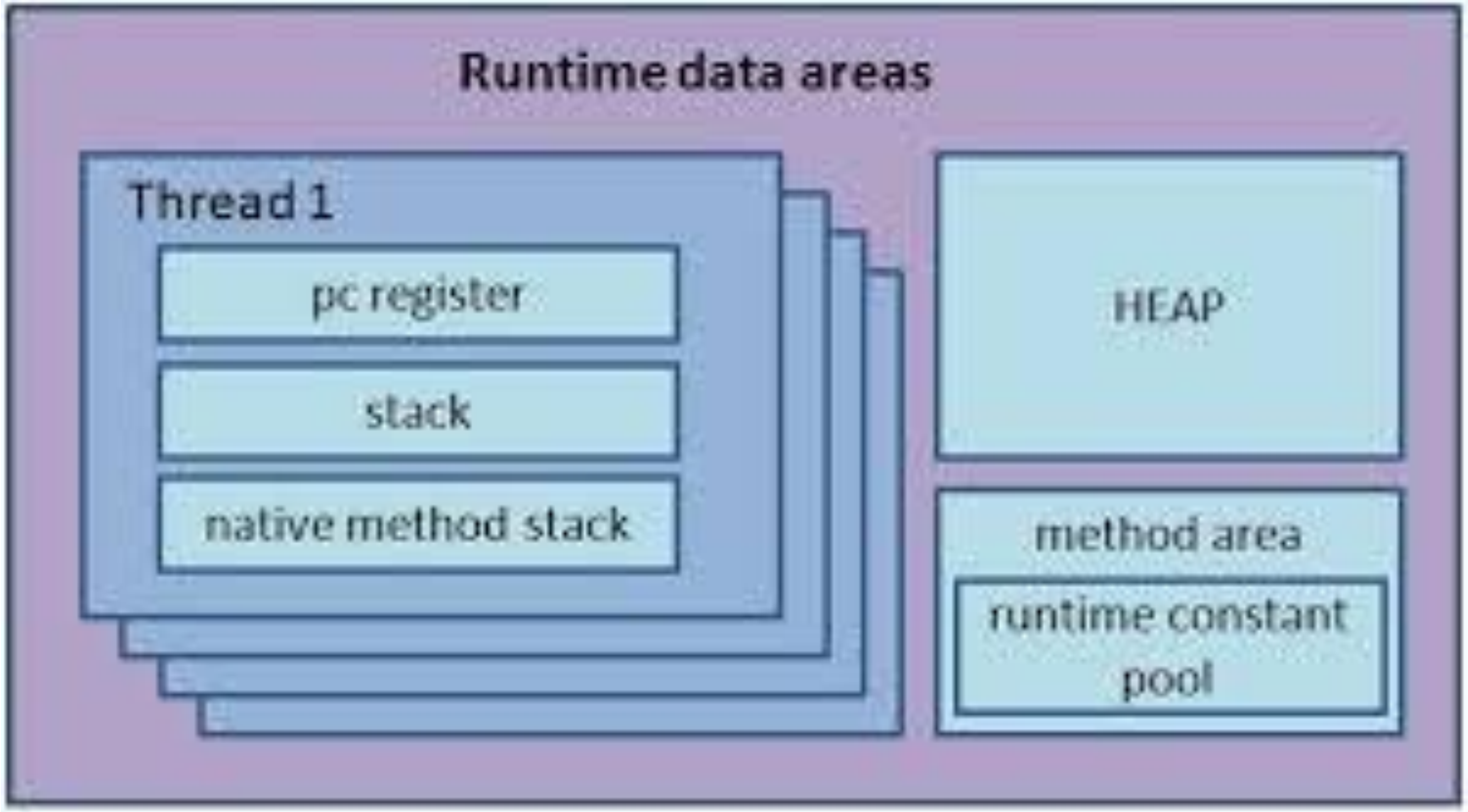
Figure: Thread Scheduler

Java JVM (Language Level - User)

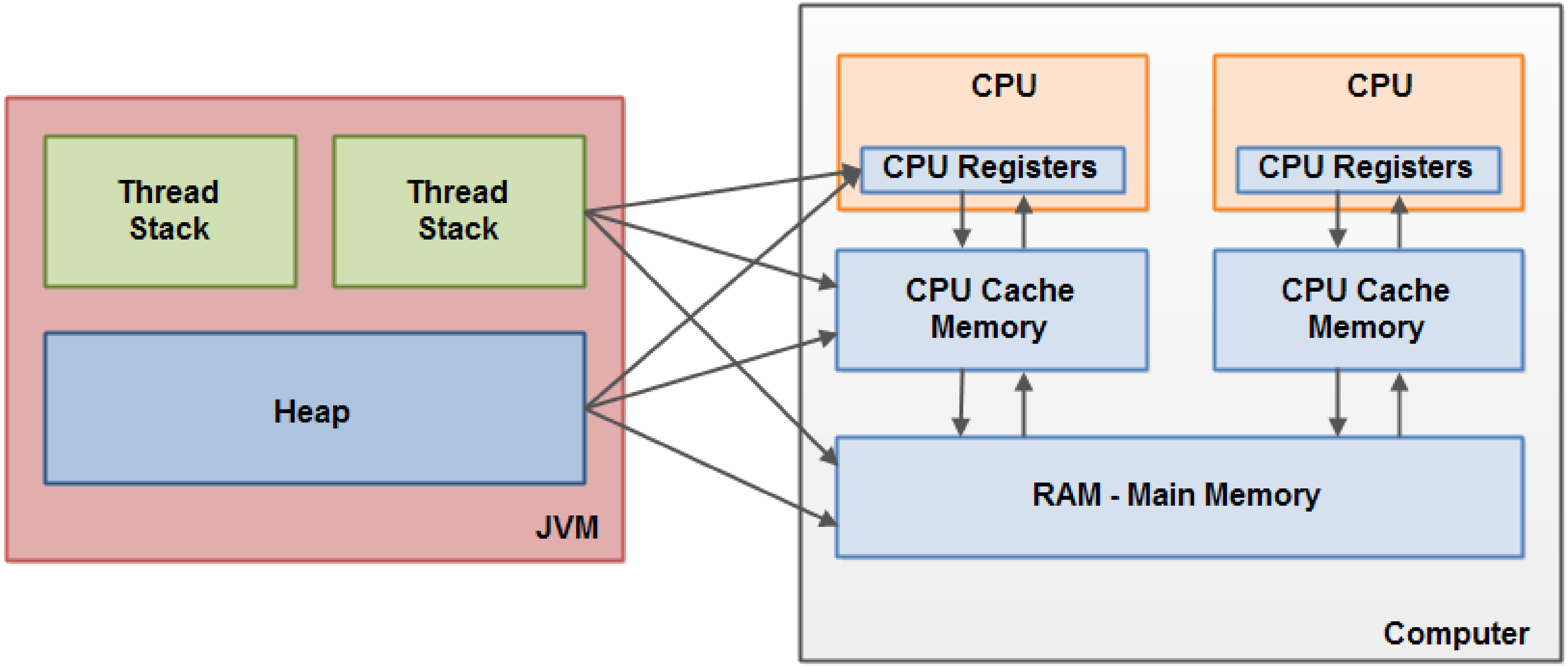


OS – Process Scheduler (Kernel)

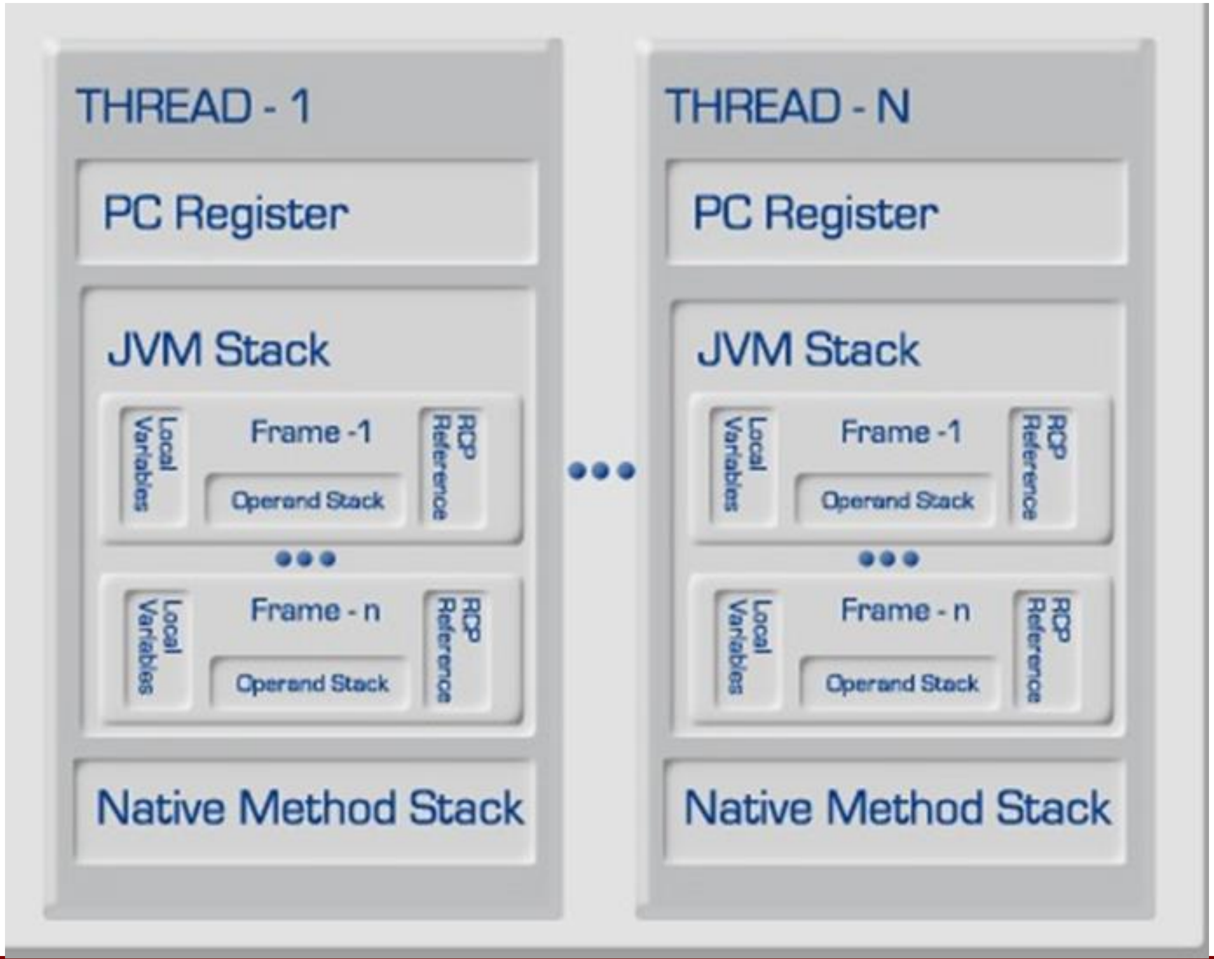
Java Memory Model



Java Memory Model



Per Thread

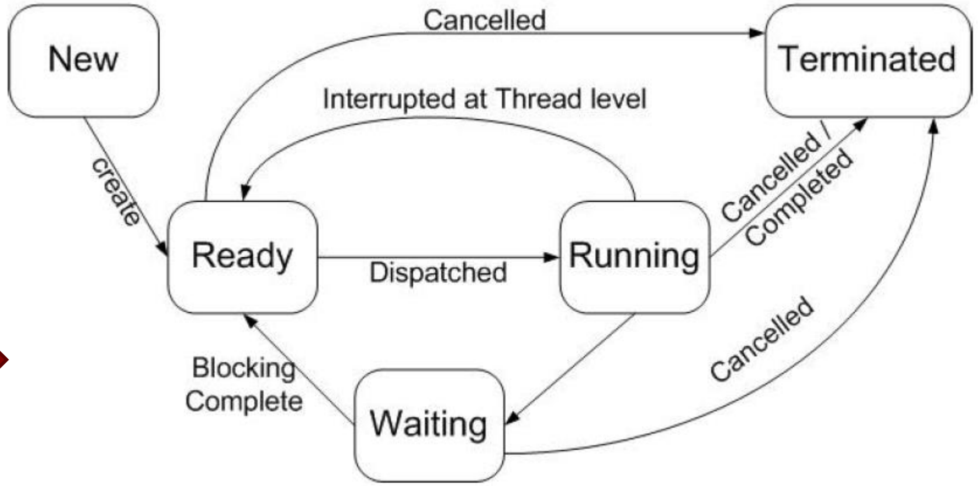


- Program Counter (PC)
- JVM Stack
- **Frame (Call Frame)**
- **Local Variable Array**
- **Operand Stack**
- Dynamic Linking
- Native Method Stack

Concurrent Programming Fundamentals

- SCHEDULERS give us the ability to "put a thread/process to sleep" and run something else on its process/processor
  - start with coroutines
  - make uniprocessor run-until-block threads
  - add preemption(Prioritized)
  - add multiple processors

OS Thread States



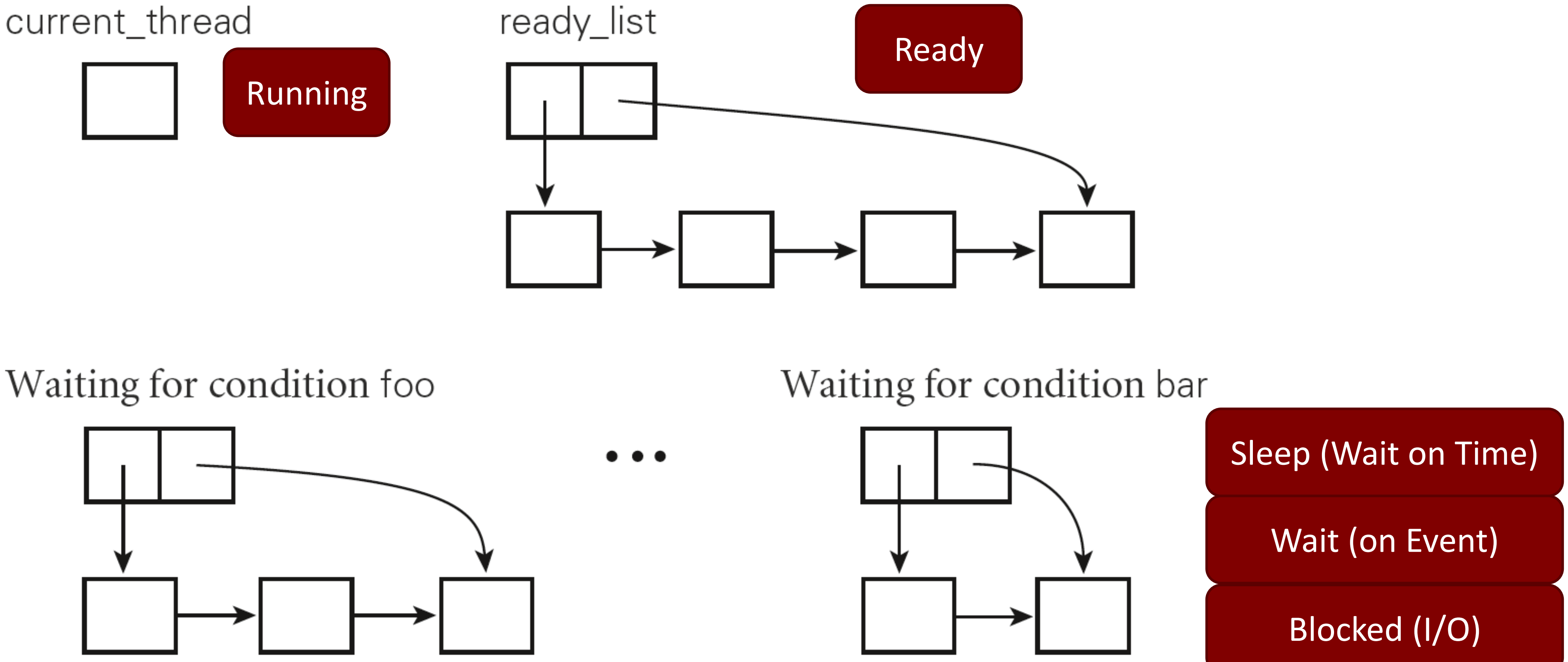


Figure 13.7 Data structures of a simple scheduler. A designated current_thread is running. Threads on the ready list are runnable. Other threads are blocked, waiting for various conditions to become true. If threads run on top of more than one OS-level process, each such process will have its own current_thread variable. If a thread makes a call into the operating system, its process may block in the kernel.

Uniprocessor Scheduling

- At any time, a thread is either Runnable(Ready) or Blocked(Wait).
- A runnable thread is running or in ready queue(list).
- Queue is prioritized. (Run-Robin if same priority.)
- To yield the processor to other thread, **run reschedule**.
- If time-expire (when sharing) or blocked, a thread can **yield** the processor to other thread and return to queue.
- To enter into wait states **run sleep_on** (on time-stamp, on event, or on exception handling)
- Cooperative multi-threading: long running thread must yield at certain point to be fair.

```
procedure reschedule
    t : thread := dequeue(ready_list)
    transfer(t)

procedure yield
    enqueue(ready_list, current_thread)
    reschedule

procedure sleep_on(ref Q : queue of thread)
    enqueue(Q, current_thread)
    reschedule
```

Concurrent Programming Fundamentals

Preemption

- Use timer interrupts (in OS) or signals (in library package) to trigger involuntary yields
- Requires that we protect the scheduler data structures:

```
procedure yield:
    disable_signals
    enqueue(ready_list, current)
    Reschedule
    re-enable_signals
```

- Note that reschedule takes us to a different thread, possibly in code other than yield Invariant: EVERY CALL to reschedule must be made with signals disabled, and must re-enable them upon its return

```
disable_signals
if not <desired condition>
    sleep_on <condition queue>
re-enable signals
```

Multi-Processor Scheduling

- We can extend our preemptive thread package to run on top of more than one OS-provided process by arranging by sharing the ready list and related data structures.
  - More than one thread will be able to run at once. (Multiprocessor)
  - In a single shared processor, the program will be able to make forward progress even when all but one of the processes are blocked in the operating system.
- Any thread that is runnable is placed in the ready list. When a process calls re-schedule, the queue-based ready list will give it the longest-waiting thread. (Round-Robin Algorithm)

Multi-Processor Scheduling

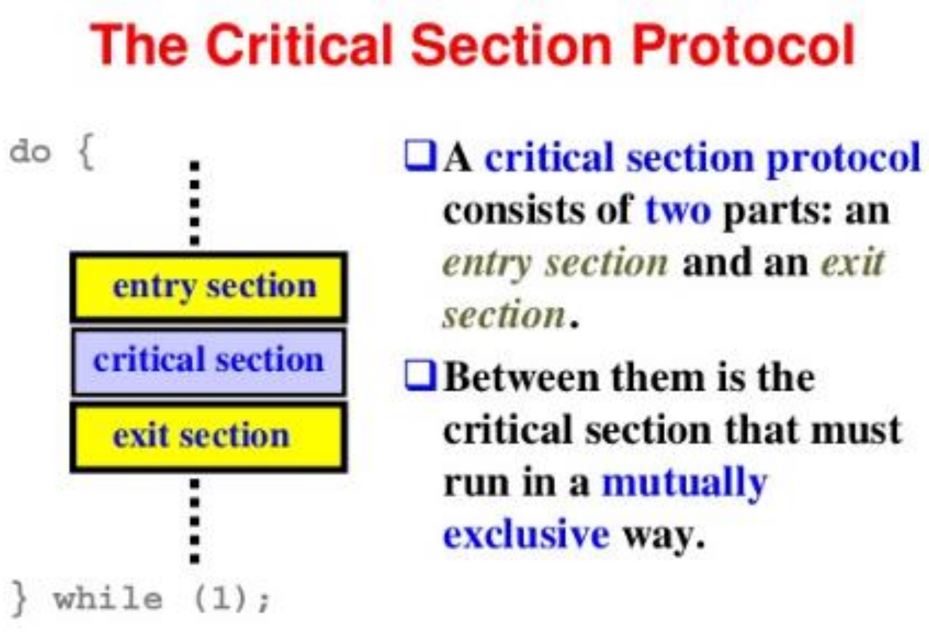
- The ready list of a scheduler might give priority to interactive or time-critical threads, or to threads that last ran on the current processor, and may therefore still have data in the cache.
- True or quasi-parallelism introduces races between calls in separate OS processes. To resolve the races, we must implement additional synchronization to make scheduler operations in separate processes atomic.

Synchronization I

Lock and Barriers

SECTION 5

Critical Section



Implementation of Synchronization

- Disable Interrupts
- Bakery Algorithms
- Spinlock
- Barriers
- Semaphores
- Conditional Critical Regions
- Condition Variables
- Monitor

http://people.eecs.berkeley.edu/~kubitron/courses/cs162-F06/hand-outs/synch.html

Implementing Synchronization

- Condition synchronization with atomic reads and writes is easy
  - You just cast each condition in the form of "location X contains value Y" and you keep reading X in a loop until you see what you want
- **Mutual exclusion is harder**
  - Much early research was devoted to figuring out how to build it from simple atomic reads and writes
  - Dekker is generally credited with finding the first correct solution for two processes in the early 1960s
  - Dijkstra published a version that works for N processes in 1965
  - Peterson published a much simpler two-process solution in 1981

Implementing Synchronization

- Repeatedly reading a shared location until it reaches a certain value is known as **SPINNING** or **BUSY-WAITING**
- A **busy-wait** mutual exclusion mechanism is known as a SPIN LOCK
  - The problem with spin locks is that they waste processor cycles
  - Synchronization mechanisms are needed that interact with a thread/process scheduler to put a process to sleep and run something else instead of spinning
  - Note, however, that spin locks are still valuable for certain things, and are widely used
    - In particular, it is better to spin than to sleep when the expected spin time is less than the rescheduling overhead

Spin Lock: Test and Set Lock

A simple test-and-test_and_set lock (Lock/Unlock)

BUSY

```
type lock = Boolean := false;

procedure acquire_lock(ref L : lock)
    while not test_and_set(L)
        while L
            -- nothing -- spin
procedure release_lock(ref L : lock)
    L := false
```

test-and-test_and_set lock

Spin Lock is request lock by busy waiting.

It sets a Boolean variable to true and returns an indication of whether the variable was previously false. Given **test_and_set**, acquiring a spin lock is almost trivial:

```
while not test_and_set(L)
    -- nothing -- spin
```

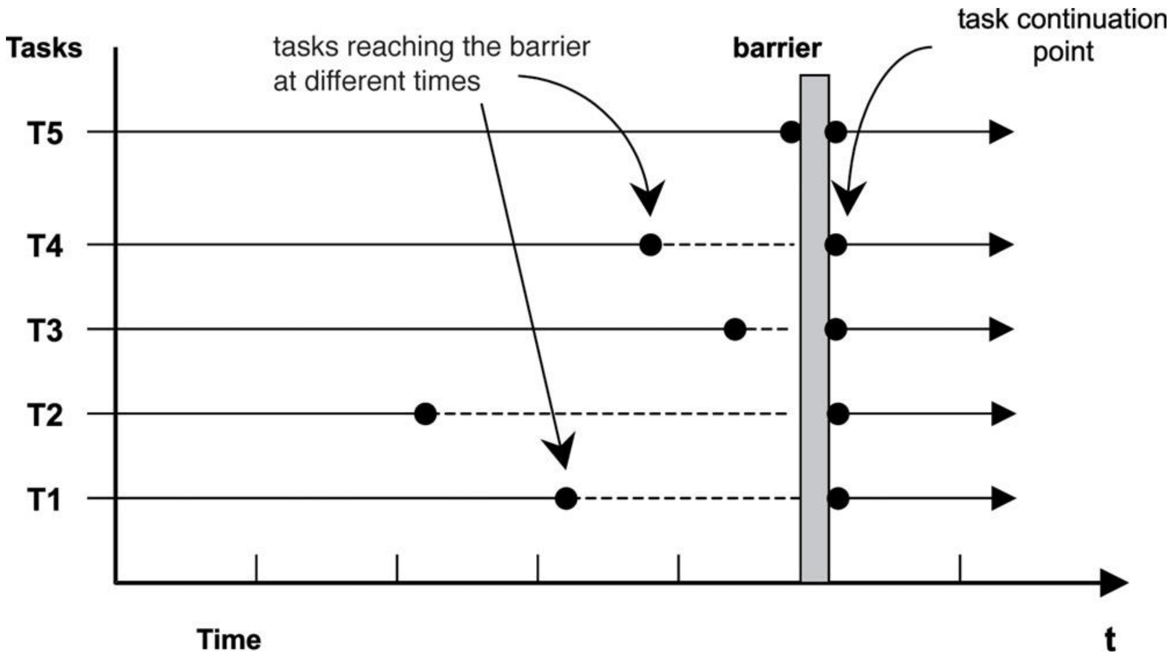
Embedding **test_and_set** in a loop tends to result in unacceptable amounts of communication on a multiprocessor, as the cache coherence mechanism attempts to reconcile writes by multiple processors attempting to acquire the lock. This overdemand for hardware resources is known as **contention.**

To reduce contention, the writers of synchronization libraries often employ Test-and-test and set a **test-and-test_and_set lock**, which spins with ordinary reads (satisfied by the cache) until it appears that the **lock is free**.

Barriers: Sense-reversing Barrier

Each thread has its own copy of local sense. Threads share a single copy of count and sense.

BUSY




```
shared count : integer := n
shared sense : Boolean := true
per-thread private local_sense : Boolean := true

procedure central_barrier
    local_sense := not local_sense
        –– each thread toggles its own sense
    if fetch_and_decrement(count) = 1
        –– last arriving thread
        count := n              –– reinitialize for next iteration
        sense := local_sense    –– allow other threads to proceed
    else
        repeat
            –– spin
        until sense = local_sense
```

fetch_and_decrement(count)

Nonblocking Algorithms

No-lock Synchronization

Suppose we wish to make an arbitrary update to a shared location:

```
x := foo(x);
```

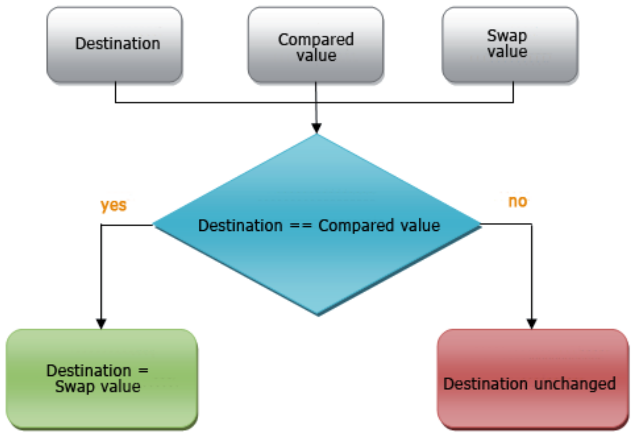
Note that this update involves at least two accesses to x: one to read the old value and one to write the new. We could protect the sequence with a lock:

```
acquire(L)
    r1 := x
    r2 := foo(r1)
    x := r2
release(L)
```

But we can also do this without a lock, using **compare_and_swap**:

```
start:
    r1 := x
    r2 := foo(r1)
    r2 := CAS(x, r1, r2)
    if !r2 goto start
```

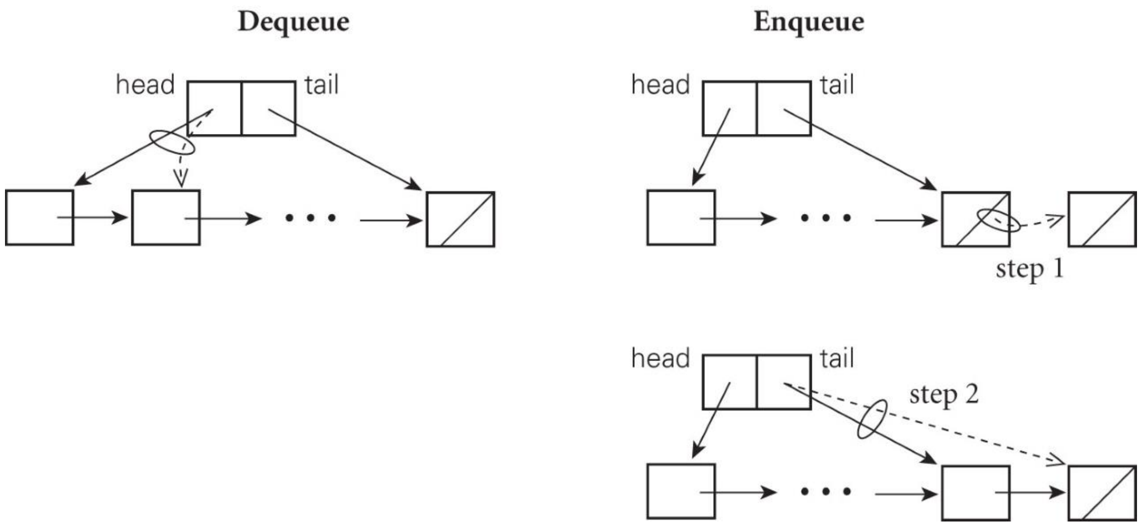
CAS is a universal primitive for single-location atomic update. A similar primitive, known as **load_linked/store_conditional**, is available on MIPS, Alpha, and PowerPC processors



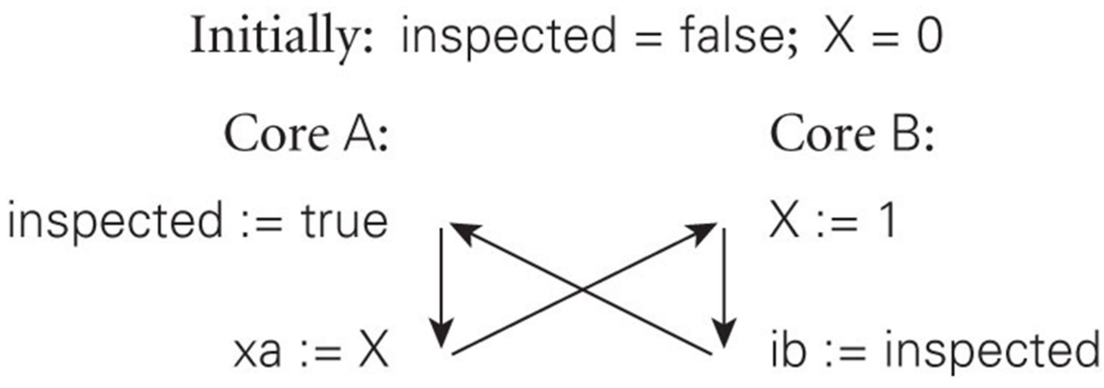

```
repeat
    prepare
    CAS
until success
clean up
```

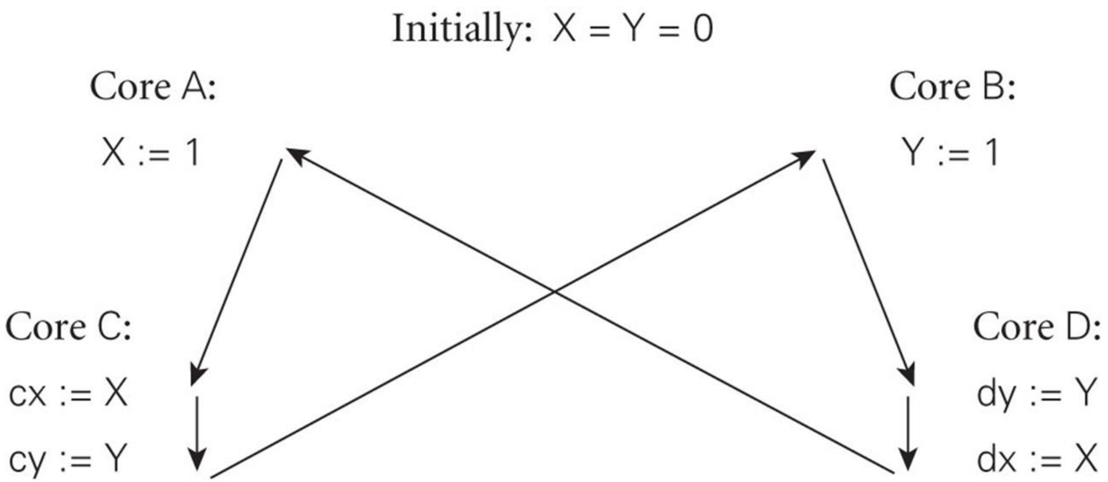
Operations on a Nonblocking Concurrent Queue

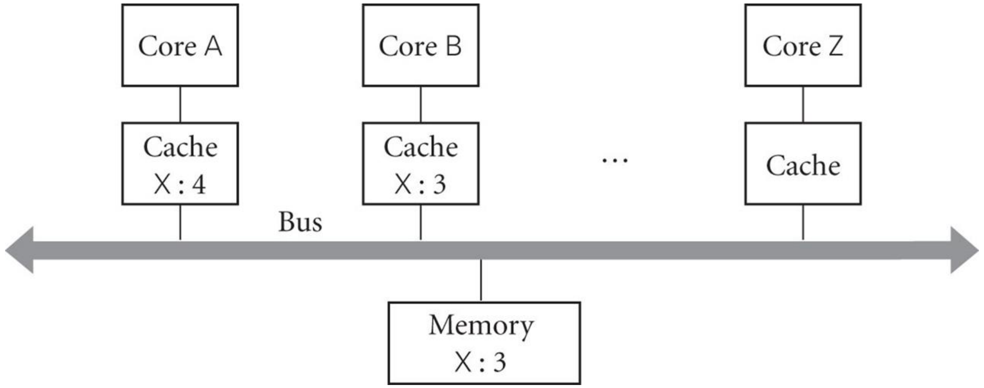
- In the dequeue operation (left), a single CAS swings the head pointer to the next node in the queue.
- In the enqueue operation (right), a first CAS changes the next pointer of the tail node to point at the new node, at which point the operation is said to have logically completed.
- A subsequent “cleanup” CAS, which can be performed by any thread, swings the tail pointer to point at the new node as well.



Memory Consistency Models







- **Write-Through**
- **Snoopy Bus**
- **Write-Back**
- **Write-invalidate**

bas been discussed.

Synchronization II

Scheduler Lock

SECTION 6

Scheduler Implementation of Synchronization

Disable Timer-Scheduler Lock (Get the exclusive access to scheduler)

- To implement user-level threads, OS-level processes must synchronize access to the ready list and condition queues, generally by means of spinning.
- Code for a simple reentrant thread scheduler appears in a later slide. As in the code, we **disable timer signals** before entering **scheduler code**, to protect the ready list and condition queues from concurrent access by a process and its own signal handler. Our code assumes a single “low-level” lock (**scheduler lock**) that protects the entire scheduler.
- Before saving its context block on a queue (e.g., in yield or sleep on), a thread must acquire the scheduler lock. It must then release the lock after returning from reschedule.

Pseudocode for part of a simple reentrant scheduler

- Every process has its own copy of current thread. There is a single shared scheduler lock and a single ready list. If processes have dedicated processors, then the low level lock can be an ordinary spin lock; otherwise it can be a **“spin-then-yield”** lock.

- The loop inside reschedule **busy-waits** until the ready list is nonempty. The code for sleep on cannot disable timer signals and acquire the scheduler lock itself, because the caller needs to test a condition and then block as a single atomic operation.

```
shared scheduler_lock : low_level_lock
shared ready_list : queue of thread
per-process private current_thread : thread

procedure reschedule()
    –– assume that scheduler_lock is already held and that timer signals are disabled
    t : thread
    loop
        t := dequeue(ready_list)
        if t ≠ null
            exit
        –– else wait for a thread to become runnable
        release_lock(scheduler_lock)
        –– window allows another thread to access ready_list (no point in reenabling
        –– signals; we're already trying to switch to a different thread)
        acquire_lock(scheduler_lock)
    transfer(t)
    –– caller must release scheduler_lock and reenable timer signals after we return

procedure yield()
    disable_signals()
    acquire_lock(scheduler_lock)
    enqueue(ready_list, current_thread)
    reschedule()
    release_lock(scheduler_lock)
    reenable_signals()

procedure sleep_on(ref Q : queue of thread)
    –– assume that caller has already disabled timer signals and acquired
    –– scheduler_lock, and will reverse these actions when we return
    enqueue(Q, current_thread)
    reschedule()
```

Scheduler Implementation of Synchronization

Disable Timer-Scheduler Lock

- The code for yield can implement synchronization itself, because its work is self-contained.
- The code for sleep on cannot, because a thread must generally check a condition and block if necessary as a single atomic operation:

```
disable_signals
acquire_lock(scheduler_lock)
if not desired_condition
    sleep_on(condition_queue)
release_lock(scheduler_lock)
reenable_signals
```

Code for Sleep on if it need Scheduler-Lock

Scheduler Implementation of Synchronization

Bounded-Buffer

- A **bounded buffer** is a concurrent queue of limited size into which producer threads insert data, and from which consumer threads remove data. The buffer serves to even out fluctuations in the relative rates of progress of the two classes of threads, increasing system throughput.
- A correct implementation of a bounded buffer requires both **atomicity** and **condition synchronization**: the former to ensure that no thread sees the buffer in an inconsistent state in the middle of some other thread's operation; the latter to force consumers to wait when the buffer is empty and producers to wait when the buffer is full.

Scheduler Implementation of Synchronization

Bounded-Buffer

- Start by imagining an unbounded (infinite) buffer
- Producer process writes data to buffer
  - Writes to In and moves rightwards
- Consumer process reads data from buffer
  - Reads from Out and moves rightwards
  - Should not try to consume if there is no data

Need an infinite buffer

Scheduler-Based Synchronization

- The problem with busy-wait synchronization is that it consumes processor cycles, cycles that are therefore unavailable for other computation. Busy-wait synchronization makes sense only if
  - (1) one has nothing better to do with the current processor, or
  - (2) the expected wait time is less than the time that would be required to switch contexts to some other thread and then switch back again.
- To ensure acceptable performance on a systems, most concurrent programming languages employ scheduler-based synchronization mechanisms, which switch to a different thread when the one that was running blocks.

Scheduler-Based Synchronization

- In the following subsection we consider semaphores, the most common form of scheduler-based synchronization. In Lecture about language mechanisms, we consider the higher level notions of monitors conditional critical regions, and transactional memory. In each case, scheduler-based synchronization mechanisms remove the waiting thread from the scheduler's ready list, returning it only when the awaited condition is true (or is likely to be true). By contrast, a spin-then-yield lock is still a busy-wait mechanism: the currently running process relinquishes the processor, but remains on the ready list. It will perform a **`test_and_set`** operation every time every time the lock appears to be free, until it finally succeeds.
- Scheduler-based synchronization is **"level-dependent"**—it is specific to **threads** when implemented in the language run-time system, or to processes when implemented in the operating system.

Synchronization III

Semaphores

SECTION 7

Implementing Synchronization

- SEMAPHORES were the first proposed **SCHEDULER-BASED** synchronization mechanism, and remain widely used
- **CONDITIONAL CRITICAL REGIONS** and **MONITORS** came later
- **Monitors** have the highest-level semantics, but a few sticky semantic problem - they are also widely used
- Synchronization in Java is sort of a hybrid of monitors and CCRs (Java 3 will have true monitors.)
- Shared-memory synch in Ada 95 is yet another hybrid

Implementing Synchronization

- A **semaphore** is a special counter (Synchronized Counter)
- It has an initial value and two operations, P and V, for changing that value
- A semaphore keeps track of the difference between the number of **P** and **V** operations that have occurred
- A P operation is delayed (the process is de-scheduled) until **#P-#V <= C**, the initial value of the semaphore

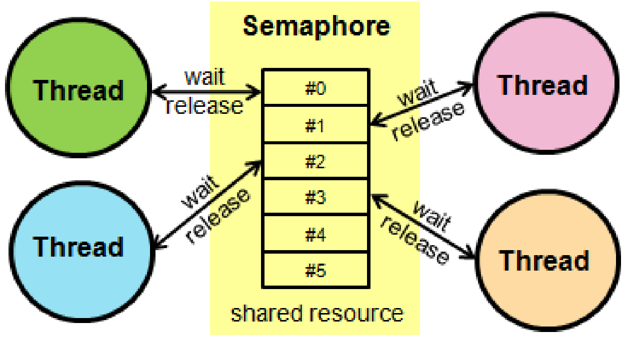
```
shared scheduler_lock : low_level_lock
shared ready_list : queue of thread
per-process private current_thread : thread

procedure reschedule()
    –– assume that scheduler_lock is already held and that timer signals are disabled
    t : thread
    loop
        t := dequeue(ready_list)
        if t ≠ null
            exit
        –– else wait for a thread to become runnable
        release_lock(scheduler_lock)
        –– window allows another thread to access ready_list (no point in reenabling
        –– signals; we're already trying to switch to a different thread)
        acquire_lock(scheduler_lock)
    transfer(t)
    –– caller must release scheduler_lock and reenable timer signals after we return

procedure yield()
    disable_signals()
    acquire_lock(scheduler_lock)
    enqueue(ready_list, current_thread)
    reschedule()
    release_lock(scheduler_lock)
    reenable_signals()

procedure sleep_on(ref Q : queue of thread)
    –– assume that caller has already disabled timer signals and acquired
    –– scheduler_lock, and will reverse these actions when we return
    enqueue(Q, current_thread)
    reschedule()
```

Figure 13.13 **Pseudocode for part of a simple reentrant (parallelism-safe) scheduler.** Every process has its own copy of current_thread. There is a single shared scheduler_lock and a single ready_list. If processes have dedicated cores, then the low_level_lock can be an ordinary spin lock; otherwise it can be a "spin-then-yield" lock (Figure 13.14). The loop inside reschedule busy-waits until the ready list is nonempty. The code for sleep_on cannot disable timer signals and acquire the scheduler lock itself, because the caller needs to test a condition and then block as a single atomic operation.

Note: a possible implementation is shown on the next slide

Semaphore

Semaphore operations, for use with the scheduler code



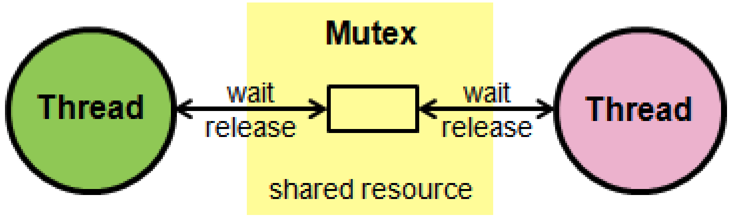
Semaphore

```
synchronized void P() {
    s = s – 1;
}

synchronized void V() {
    s = s + 1;
}
```

Use a **mutex** so that increment (V) and decrement (P) operations on the counter are **atomic**.

```
type semaphore = record
    N : integer –– always non-negative
    Q : queue of threads

procedure P(ref S : semaphore)
    disable_signals()
    acquire_lock(scheduler_lock)
    if S.N > 0
        S.N –:= 1
    else
        sleep_on(S.Q)
    release_lock(scheduler_lock)
    reenable_signals()

procedure V(ref S : semaphore)
    disable_signals()
    acquire_lock(scheduler_lock)
    if S.Q is nonempty
        enqueue(ready_list, dequeue(S.Q))
    else
        S.N +:= 1
    release_lock(scheduler_lock)
    reenable_signals()
```

Mutual exclusion in Java



- Mutexes are built in to every Java object.
  - no separate classes
- Every Java object is/has a **monitor**.
  - At most one thread may "own" a monitor at any given time.
- A thread becomes **owner** of an object's monitor by
  - executing an object method declared as **synchronized**
  - executing a block that is **synchronized** on the object

```
public synchronized void increment()
{
        x = x + 1;
}
```

```
public void increment() {
        synchronized(this) {
                x = x + 1;
        }
}
```

Semaphore

Semaphore-based code for a bounded buffer.

The mutex binary semaphore protects the data structure proper. The full slots and empty slots general semaphores ensure that no operation starts until it is safe to do so.

```
shared buf : array [1..SIZE] of bdata
shared next_full, next_empty : integer := 1, 1
shared mutex : semaphore := 1
shared empty_slots, full_slots : semaphore := SIZE, 0

procedure insert(d : bdata)
    P(empty_slots)
    P(mutex)
    buf[next_empty] := d
    next_empty := next_empty mod SIZE + 1
    V(mutex)
    V(full_slots)

function remove() : bdata
    P(full_slots)
    P(mutex)
    d : bdata := buf[next_full]
    next_full := next_full mod SIZE + 1
    V(mutex)
    V(empty_slots)
    return d
```

	Mutex	Semaphore
Speed	Somewhat slower than a semaphore	A semaphore is generally faster than a mutex and requires fewer system resources
Thread ownership	Only one thread can own a mutex	No concept of thread ownership for a semaphore – any thread can decrement a counting semaphore if its current count exceeds zero
Priority Inheritance	Available only with a mutex	Feature not available for semaphores
Mutual Exclusion	Primary purpose of a mutex – a mutex should be used only for mutual exclusion	Can be accomplished with the use of a binary semaphore, but there may be pitfalls
Inter-thread synchronization	Do not use a mutex for this purpose	Can be performed with a semaphore, but an event flags group should be considered also
Event Notification	Do not use a mutex for this purpose	Can be performed with a semaphore
Thread Suspension	Thread can suspend if another thread already owns the mutex (depends on value of wait option)	Thread can suspend if the value of a counting semaphore is zero (depends on value of wait option)

Implementing Synchronization

- It is generally assumed that semaphores are fair, in the sense that processes complete P operations in the same order they start them
- Problems with semaphores
  - They're pretty low-level.
    - When using them for mutual exclusion, for example (the most common usage), it's easy to forget a P or a V, especially when they don't occur in strictly matched pairs (because you do a V inside an if statement, for example, as in the use of the spin lock in the implementation of P)
  - Their use is scattered all over the place.
    - If you want to change how processes synchronize access to a data structure, you have to find all the places in the code where they touch that structure, which is difficult and error-prone