Chapter 5: Process Synchronization
Chapter 5: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Alternative Approaches
Objectives

- Cooperating process affect [or, is affected by] other processes
  - Sharing data or code
  - How to maintain and ensure data consistency…?

- To present the concept of process synchronization.
- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems
Background

- Processes can execute concurrently
  - In Chap-3: Process scheduler switches among processes; *concurrency*
  - In Chap-4: Multiprogramming distributes tasks among cores; *parallelism*

- May be interrupted at any time, partially completing execution
  - How to preserve the *integrity* of data shared by several processes?

- Concurrent access to shared data may result in data inconsistency

- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

- **Illustration of the problem:** Suppose that we wanted to provide a solution to the consumer-producer problem that fills *all* the buffer; the original solution in Chap-3 allowed at most *BUFFER_SIZE* - 1 items in the buffer at the same time. We can do so by having an integer *counter* that keeps track of the number of items in the buffers. Initially, *counter* is set to 0. It is incremented by the producer after it produces a new item and is decremented by the consumer after it consumes an item.
while (true)
{
    /* produce an item in next produced */

    while (counter == BUFFER_SIZE) ;
        /* do nothing when buffer full */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;

    counter++;
}

Consumer

while (true)
{
    while (counter == 0) ;
    ; /* do nothing when buffer empty */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    counter--;  
    /* consume the item in next consumed */
}

```
Race Condition

- Producer and consumer may function incorrectly when executed concurrently

- `counter++` could be implemented as
  ```
  register1 = counter
  register1 = register1 + 1
  counter = register1
  ```

- `counter--` could be implemented as
  ```
  register2 = counter
  register2 = register2 - 1
  counter = register2
  ```

- Consider this execution interleaving with “count = 5” initially:
  ```
  S0: producer execute register1 = counter {register1 = 5}
  S1: producer execute register1 = register1 + 1 {register1 = 6}
  S2: consumer execute register2 = counter {register2 = 5}
  S3: consumer execute register2 = register2 - 1 {register2 = 4}
  S4: producer execute counter = register1 {counter = 6 }
  S5: consumer execute counter = register2 {counter = 4}
  ```

- **Race condition** = concurrent access to variable + result depends on order

  - **Solution**: synchronization; only one process at a time accesses data
Critical Section Problem

- Consider a system of \( n \) processes \( \{P_0, P_1, \ldots P_{n-1}\} \)

- Each process has a segment of code called a critical section, in which
  - It may be changing common variables, updating table, writing file, … etc
  - When it is in its critical section, no other should be in its critical section
    - No two processes are executing in their critical sections at the same time

- Critical section problem is to design a protocol to solve this
  - That is: protocol that processes can use to cooperate

- Each process must ask permission to enter its critical section via the entry section, leave the critical section through the exit section, then resumes its operations in the remainder section
Critical Section

- General structure of process $P_i$

```c
    do {
        entry section
        critical section
        exit section
        remainder section
    } while (true);
```
Solution to Critical-Section Problem

A solution to the critical section problem must satisfy the following 3 requirements:

1. **Mutual Exclusion** - If process $P_i$ is executing in its critical section, then no other processes can be executing in their critical sections.

2. **Progress** - If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. **Bounded Waiting** - A limit must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
   - We assume that each process executes at a nonzero speed.
   - We make no assumption concerning the relative speed of the $n$ processes.
   - Imagine case where many active kernel processes accessing a list of all open files in the system. This data structure is prone to possible race conditions.
Critical-Section Handling in OS

- Two approaches depending on if kernel is preemptive or non-preemptive

  - **Preemptive** – allows preemption of process when running in kernel mode
    - *Preemption* = The ability of the OS to interrupt a currently scheduled task in favor of a higher priority task. It is normally carried out by a *privileged task* or part of the system known as a *preemptive process scheduler*, which has the power to *preempt*, or interrupt, and later resume, other tasks in the system. This only applies to processes running in kernel mode.
    - More responsive to users, but
      - Shared kernel data *may not* be free from race conditions
      - Preemptive kernel must be carefully designed

  - **Non-preemptive** – process runs until it exits kernel mode, blocks, or voluntarily yields CPU
    - Essentially free of race conditions in kernel mode *as only one process is active in the kernel at a time*
Synchronization Hardware

Many systems provide hardware support for implementing the critical section code.

- Software-based solutions not guaranteed to be correct on modern comp arch
  - Example: *Peterson’s solution* to the CS problem (see Page-207)

All solutions in following slides are based on the idea of locking

- Protecting critical regions via locks

Uniprocessors – could disable interrupts

- Solve CS problem by *preventing interrupts* while modifying shared data
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable

Modern machines provide special atomic hardware instructions

- Atomic = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words
Solution to Critical-Section Problem Using Locks on Single-CPU Systems

\[
\begin{align*}
\text{do} & \\
\{ & \\
\text{acquire lock} & \\
\text{critical section} & \\
\text{release lock} & \\
\text{remainder section} & \\
\} & \\
\text{while (TRUE);}
\end{align*}
\]
**Test_and_Set Instruction**

- **Definition:**

  ```
  boolean test_and_set(boolean *target) {
  boolean rv = *target;
  *target = TRUE;
  return rv;
  }
  ```

1. Executed atomically; a *non-interruptible unit* of execution
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

- Thus, we can implement mutual exclusion on multi-CPU systems
  - See next slide
Solution using Test_and_Set() Instruction on Multi-CPU Systems

- Shared Boolean variable `lock`, declared and initialized to FALSE

- Solution, for a process $P_i$:

  ```
  do
  {
    while (test_and_set(&lock))
      ; /* do nothing */
    /* critical section */
    lock = false;
    /* remainder section */
  }
  while (true);
  ```
Compare_and_Swap Instruction

- Definition:

```c
int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;
    if (*value == expected) {
        *value = new_value;
    }
    return temp;
}
```

1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” to “new_value” but only if “value” == “expected”. That is, the swap takes place only under this condition.

- Thus, we can implement mutual exclusion on multi-CPU systems
  - See next slide
Solution using `Compare_and_Swap()` Instruction on Multi-CPU Systems

- Shared integer variable `lock`, declared and initialized to 0;

- Solution; for a process $P_i$:
  
  ```
  do
  {
    while (compare_and_swap(&lock, 0, 1) != 0)
    ; /* do nothing */
    /* critical section */
    lock = 0;
    /* remainder section */
  }
  while (true);
  ```
Do /* Shared Boolean waiting[n] and lock initialized to false */
{
    waiting[i] = true;
    key = true;
    while (waiting[i] && key) { key = test_and_set(&lock); }
    waiting[i] = false;

    /* critical section; enters only if waiting[i] or key == false */

    j = (i + 1) % n;
    while ((j != i) && !waiting[j]) { j = (j + 1) % n; }

    if (j == i) { lock = false; }
    else { waiting[j] = false; }

    /* remainder section */
}

while (true);
Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers

- OS designers build software tools to solve critical section problem

- Simplest is mutex lock; short for mutual exclusion

- Protect a critical section by first `acquire()` a lock then `release()` the lock
  - Boolean variable indicating if lock is available or not

- Calls to `acquire()` and `release()` must be atomic
  - Usually implemented via hardware atomic instructions

- But this solution requires busy waiting
  - This lock therefore is called a spinlock
Acquire() and Release()

- acquire() {
  while (!available)
    ; /* busy wait */
    available = false;;
}

- release() {
  available = true;
}

- do {
  acquire lock
  critical section
  release lock
  remainder section
} while (true);
Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.

- Semaphore $S$ – integer variable

- Can only be accessed via two indivisible (atomic) operations
  - `wait()` and `signal()`
  - Originally called $P()$ and $V()$

- Definition of the `wait()` operation
  ```
  wait(S) {
    while (S <= 0) ; // busy wait
    S--;
  }
  ```

- Definition of the `signal()` operation
  ```
  signal(S) {
    S++;
  }
  ```
Semaphore Usage

- **Counting semaphore** – integer value $S$ can range over an unrestricted domain
  - Can be used to control access to resources ($S$ initialized to number of resources)
    - Process that wishes to use resource will execute `wait(S)`
    - Process that releases a resource will execute `signal(S)`
    - If $S = 0$ then all resources are being used, and, Processes block until $S > 0$;

- **Binary semaphore** – integer value $S$ can range only between 0 and 1
  - Behaves similarly to mutex lock

Can solve various synchronization problems

Consider concurrent $P_1$ and $P_2$ that require statement $S_1$ to happen before $S_2$

Create a common semaphore "synch" shared by $P_1$ and $P_2$ and initialized to 0

$P_1$:

```
S1;
signal(synch);
```

$P_2$:

```
wait(synch);
S2;  since synch is initialized to 0, P2 executes S2 only after P1 invokes signal(synch)
```

- Can implement a counting semaphore $S$ as a binary semaphore
Semaphore Implementation with no Busy Waiting

- With each semaphore $S$ there is an associated waiting queue

- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list

- Two operations:
  - `block` – place the process invoking it on the appropriate $S$ waiting queue
  - `wakeup` – transfer one of process from the $S$ waiting queue to ready queue

```c
typedef struct{
    int value;
    struct process *list;  // list of processes waiting on the semaphore $S$
} semaphore;
```
Semaphore Implementation with no Busy Waiting

wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
} wait() operation adds a process to the S queue and suspends it

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
} signal() operation removes one process from S queue and awakens it
Semaphore Implementation

- Must guarantee that no two processes can execute the `wait()` and `signal()` operations on the same semaphore S at the same time
  - Hence, if P executes `wait()` or `signal()` then no other processes should
    - This is a CS-problem
  - Also, recall that `wait()` and `signal()` operations must be atomic

- Thus, in semaphore implementation the `wait()` and `signal()` codes are placed in the critical section of a process P
  - We have moved `wait()` from entry section and `signal()` from exit section of P
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical sections of `wait()` and `signal()` rarely occupied

- Note that applications may spend lots of time in critical sections and therefore this is not a good solution;
Deadlock and Starvation

- **Deadlock** – a process is waiting indefinitely for an event that can be caused only by another waiting process

- Let $S$ and $Q$ be two semaphores initialized to 1

  $P_0$
  ```
  wait(S);
  wait(Q);
  ...
  signal(S);
  signal(Q);
  ```

  $P_1$
  ```
  wait(Q);
  wait(S);
  ...
  signal(Q);
  signal(S);
  ```

  P and Q are suspended here

- **Starvation** – indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended

- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via priority-inheritance protocol
Classic Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes

  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem
Bounded-Buffer Problem

- Producer and consumer processes share the following data

  - $n$ buffers, each can hold one item

  - Semaphore `mutex` initialized to the value 1; for permission access to buffer pool

  - Semaphore `full` initialized to the value 0; number of full buffers

  - Semaphore `empty` initialized to the value $n$; number of empty buffers
Bounded Buffer Problem

- The structure of the producer process; solution using semaphores

```c

while (true)
{
    /* produce an item in next_produced */
    ...
    wait(empty);
    wait(mutex);

    /* add next produced to the buffer */
    ...
    signal(mutex);
    signal(full);
}
```

```c
```
Bounded Buffer Problem

- The structure of the consumer process; solution using semaphores

Do

{  
  wait(full);
  wait(mutex);
  ...
  /* remove an item from buffer to next_consumed */
  ...
  signal(mutex);
  signal(empty);
  ...
  /* consume the item in next consumed */
  ...
}
while (true);
Readers-Writers Problem

A data set is shared among a number of concurrent processes

- Readers – only read the data set; they do not perform any updates
  - Never a problem if only readers access the data; no waiting necessary
- Writers – can both read and write;
  - Problem if some writer accesses data; thus, exclusive access only for writing

Problem – allow multiple readers to read at the same time

- Only one single writer can access the shared data at the same time

Several variations of how readers and writers are considered – all involve some form of priorities

One solution with semaphores: the reader processes share the following data

- Data set
- Semaphore `rw_mutex` initialized to 1; common to both readers and writers
- Semaphore `mutex` initialized to 1; mutual exclusion if `read_count` updated
- Integer `read_count` initialized to 0; # of processes currently reading
Readers-Writers Problem

The structure of a writer process

```c
  do
    { 
      wait(rw_mutex);  // mutual exclusion for writers
      ...  
      /* writing is performed */
      ...  
      signal(rw_mutex);
    }
  while (true);
```
Readers-Writers Problem

- The structure of a reader process

```c
    do
    {
        wait(mutex);
        read_count++; /* critical section when read_count being modified */
        if (read_count == 1)
            wait(rw_mutex); /* in case a writer is writing */
        signal(mutex);
        ...
        /* reading is performed */
        ...
        wait(mutex);
        read_count--; /* critical section when read_count being modified */
        if (read_count == 0)
            signal(rw_mutex); /* release lock */
            signal(mutex); /* release lock */
    }
    while (true);
```
Readers-Writers Problem Variations

- **First** variation – no reader kept waiting unless writer has permission to use shared object

- **Second** variation – once writer is ready, it performs the write ASAP

- Both may have starvation leading to even more variations

- Problem is solved on some systems by kernel providing *reader-writer* locks
Dining-Philosophers Problem

- Philosophers spend their lives alternating thinking and eating.

- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (but can pick only one at a time) to eat from bowl.
  - Need both to eat, then release both when done.

- In the case of 5 philosophers (solution with semaphores):
  - Shared data:
    - Bowl of rice (data set)
    - Semaphore chopstick[5] initialized to 1’s; each chopstick is a semaphore.
Dining-Philosophers Problem Algorithm

- The structure of Philosopher $i$: (no two neighbors are eating simultaneously)
  
  ```
  do
  {
    wait(chopstick[i]);
    wait(chopstick[(i + 1) % 5]);

    // eat
    signal(chopstick[i]);
    signal(chopstick[(i + 1) % 5]);

    // think
  }
  while (TRUE);
  ```

- What is the problem with this algorithm?
  - Deadlock if all 5 philosophers are hungry at the same time and pick a stick
Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  
  - Allow a philosopher to pick up both chopsticks only if both are available (picking must be done in a critical section).
  
  - Use an asymmetric solution -- An odd-numbered philosopher picks up first her left chopstick and then her right chopstick. Even-numbered philosopher picks up first her right chopstick and then her left chopstick.
Problems with Semaphores

- Incorrect use of semaphore operations:
  - `signal (mutex) .... wait (mutex)`
  - `wait (mutex) ... wait (mutex)`
  - Omitting of `wait (mutex)` or `signal (mutex)` (or both)

- Deadlock and starvation are possible.
Synchronization Examples

- Solaris
- Windows
- Linux
- Pthreads
Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing

- Uses **adaptive mutexes** for efficiency to protect only data accessed by short code segments
  - Starts as a standard semaphore implemented as a spin-lock; a normal mutex
  - If lock held by a thread currently running on another CPU, then the thread spins
  - If lock held by a non-running-state thread, then block and sleep waiting for signal of lock being released

- Uses **condition variables** (skipped; see Monitors in Section-5.8)

- Uses **readers-writers** locks when longer sections of code need access to data
  - To protect *frequently accessed read-only* data

- Uses **turnstile**s to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock; queues containing threads blocked on a lock
  - Turnstiles are per-lock-holding-thread, not per-object

- Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile
Windows Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems

- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted

- Provides **dispatcher objects** in user-mode which may act as mutexes, semaphores, events, and timers
  - **Events**
    - An event acts much like a condition variable (see Monitors in Section-5.8)
      - Notify a waiting thread when a desired condition occurs
  - Timers notify one or more thread when time expired
    - Notify threads that a specified amount of time has expired
  - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)
Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive

- Linux provides:
  - Semaphores
  - atomic integer operations
  - spinlocks
  - reader-writer versions of both

- On single-CPU system, spinlocks replaced by enabling and disabling kernel preemption
Pthreads Synchronization

- Pthreads API is OS-independent

- It provides:
  - mutex locks
  - condition variable

- Non-portable extensions include:
  - read-write locks
  - spinlocks
End of Chapter 5
Algorithm for Process $P_i$

do
{
    while (turn == j);

    critical section

    turn = j;

    remainder section

} while (true);
Peterson’s Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]

- The variable `turn` indicates whose turn it is to enter the critical section
- The `flag` array is used to indicate if a process is ready to enter the critical section. `flag[i] = true` implies that process `P_i` is ready!
Algorithm for Process $P_i$

do  
  
  flag[i] = true;
  turn = j;
  while (flag[j] && turn == j);

  critical section
  flag[i] = false;

  remainder section

} while (true);
Provable that the three CS requirement are met:

1. Mutual exclusion is preserved
   \( P_i \) enters CS only if:
   - either \( flag[j] = false \) or \( turn = i \)
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met
Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

```plaintext
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { .... }

    procedure Pn (...) {......}

    Initialization code (...) { ... }
}
```
Schematic view of a Monitor

- Entry queue
- Shared data
- Operations
- Initialization code
Condition Variables

- `condition x, y;`
- Two operations are allowed on a condition variable:
  - `x.wait()` – a process that invokes the operation is suspended until `x.signal()`
  - `x.signal()` – resumes one of processes (if any) that invoked `x.wait()`
    - If no `x.wait()` on the variable, then it has no effect on the variable
Monitor with Condition Variables

- Shared data
- Queues associated with \( x, y \) conditions
- Operations
- Initialization code
- Entry queue
Condition Variables Choices

- If process P invokes `x.signal()`, and process Q is suspended in `x.wait()`, what should happen next?
  - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait

- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing `signal` immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java
Monitor Solution to Dining Philosophers

```c
monitor DiningPhilosophers
{
    enum { THINKING, HUNGRY, EATING } state [5];
    condition self [5];

    void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) self[i].wait;
    }

    void putdown (int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }
}
```
void test (int i) {
    if ((state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
        state[i] = EATING ;
        self[i].signal () ;
    }
}

initialization_code() {
    for (int i = 0; i < 5; i++)
        state[i] = THINKING;
}
Each philosopher $i$ invokes the operations `pickup()` and `putdown()` in the following sequence:

```c
DiningPhilosophers.pickup(i);
EAT
DiningPhilosophers.putdown(i);
```

No deadlock, but starvation is possible.
Monitor Implementation Using Semaphores

- Variables

```c
semaphore mutex;  // (initially = 1)
semaphore next;   // (initially = 0)
int next_count = 0;
```

- Each procedure $F$ will be replaced by

```c
wait(mutex);
...
  body of $F$;
  ...
  if (next_count > 0)
    signal(next)
  else
    signal(mutex);
```

- Mutual exclusion within a monitor is ensured
Monitor Implementation – Condition Variables

- For each condition variable \( x \), we have:

\[
\text{semaphore } x\_\text{sem}; \quad \text{// (initially } = 0) \\
\text{int } x\_\text{count} = 0;
\]

- The operation \( x\.\text{wait} \) can be implemented as:

\[
\begin{align*}
x\_\text{count} &++; \\
\text{if } (\text{next}_\text{count} > 0) & \quad \text{signal}(\text{next}); \\
\text{else } & \quad \text{signal}(\text{mutex}); \\
\text{wait}(x\_\text{sem}); \\
x\_\text{count} &--;
\end{align*}
\]
The operation `x.signal` can be implemented as:

```c
if (x_count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```
Resuming Processes within a Monitor

- If several processes queued on condition x, and x.signal() executed, which should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form x.wait(c)
  - Where c is **priority number**
  - Process with lowest number (highest priority) is scheduled next
Allocate a single resource among competing processes using priority numbers that specify the maximum time a process plans to use the resource.

R.acquire(t);
...
access the resource;
...
R.release;

Where R is an instance of type ResourceAllocator.
A Monitor to Allocate Single Resource

monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = TRUE;
    }
    void release() {
        busy = FALSE;
        x.signal();
    }
    initialization code() {
        busy = FALSE;
    }
}
Alternative Approaches

- Transactional Memory
- OpenMP
- Functional Programming Languages
A memory transaction is a sequence of read-write operations to memory that are performed atomically.

```c
void update()
{
    /* read/write memory */
}
```
OpenMP

- OpenMP is a set of compiler directives and API that support parallel programming.

```c
void update(int value)
{
    #pragma omp critical
    {
        count += value
    }
}
```

The code contained within the `#pragma omp critical` directive is treated as a critical section and performed atomically.
Functional programming languages offer a different paradigm than procedural languages in that they do not maintain state.

Variables are treated as immutable and cannot change state once they have been assigned a value.

There is increasing interest in functional languages such as Erlang and Scala for their approach in handling data races.