Process Synchronization

Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages. Concurrent access to shared data may result in data inconsistency. We discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

1. Race condition

Concurrent access to shared data may result in data inconsistency. Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes. Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

- The code for the producer process:

```
     while (true)
     {
/* produce an item in next Produced */
     while (counter == BUFFER_SIZE)
      ; /* do nothing */
     buffer [in] = nextProduced;
       in = (in + 1) % BUFFER_SIZE;
        counter++;
     }
```

- The code for the consumer process:

```
     while (true)
     {
     while (counter == 0)
     ; /* do nothing */
     nextConsumed = buffer [out] ;
     out = (out + 1) % BUFFER_SIZE;
     counter--;
     /* consume the item in nextConsumed */
     }
```

- count++ could be implemented as

  $register_1$ = count
  $register_1 = register_1 + 1$
  count = register1

- count-- could be implemented as

  $register_2$ = count
  $register_2 = register_2 - 1$
  count = $register_2$

- Consider this execution interleaving with “count = 5” initially:

S0: producer execute $register_1$ = count {$register_1 = 5$}
S1: producer execute $register_1 = register_1 + 1$ {$register_1 = 6$}
S2: consumer execute $register_2$ = count {$register_2 = 5$}
S3: consumer execute $register_2 = register_2 - 1$ {register2 = 4}

S4: producer execute count = $register_1$ {count = 6 }
S5: consumer execute count = $register_2$ {count = 4}

The value of count variable depends upon the particular order in which the access take place. This kind of situation where several processes access and manipulate the same data concurrently and outcome depends upon the particular order in which the access takes place, is called **race condition**.

- Although both the producer and consumer routines are correct separately, they may not function correctly when executed concurrently.
- We would arrive at incorrect state because we allowed both processes to manipulate the variable counter concurrently.
- A **race condition** is a situation where two or more processes access shared data concurrently and final value of shared data depends on timing (race to access and modify data)
- To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter (process synchronization).

The section of code that is under the race condition is called the **critical section** of any process.

2. Solution to Critical-Section Problem

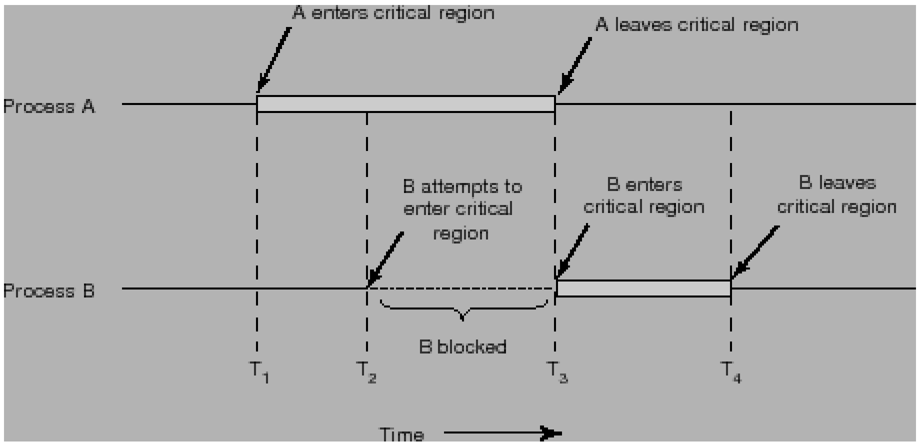
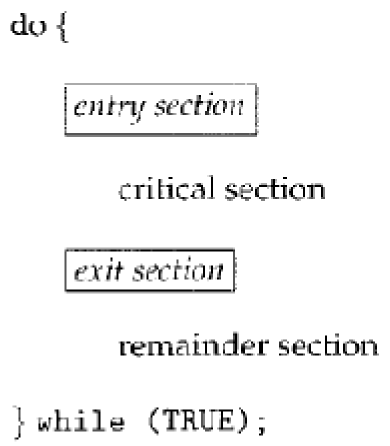


Figure: Mutual exclusion using critical regions.

- Consider a system consisting of n processesP_0, P_1, …….. P_{n-1}. Each process has a segment of code, called a **critical section (CS)**, in which the process may be changing common variables, updating a table, writing a file, and so on.
- The important feature of the system is that, when one process is executing in its CS, no other process is to be allowed to execute in its CS (Mutual exclusion).
- That is, no two processes are executing in their CSs at the same time.
- Each process must request permission to enter its CS. The section of code implementing this request is the **entry section**.
- The CS may be followed by an **exit section**.
- The remaining code is the **remainder section**.



```
do {
    entry section
        critical section
    exit section
        remainder section
} while (TRUE);
```

Figure: General structure of a typical process P_i.

2.1 A solution to the CS problem must satisfy the following requirements:

- **Mutual exclusion**. If process P_i is executing in its CS, then no other processes can be executing in their CSs.
- **Progress**. If no process is executing in its CS and some processes wish to enter their CSs, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its CS next, and this selection cannot be postponed indefinitely. (No process should have to wait forever to enter its CS.)
- **Bounded waiting**. There exists a bound, or limit, on the number of times that other processes are allowed to enter their CSs after a process has made a request to enter its CS and before that request is granted.

  No assumptions may be made about speeds or the number of CPUs.

2.2 **Atomic operation**.

Atomic means either an operation happens in its entirely or NOT at all (it cannot be interrupted in the middle). Atomic operations are used to ensure that cooperating processes execute correctly.

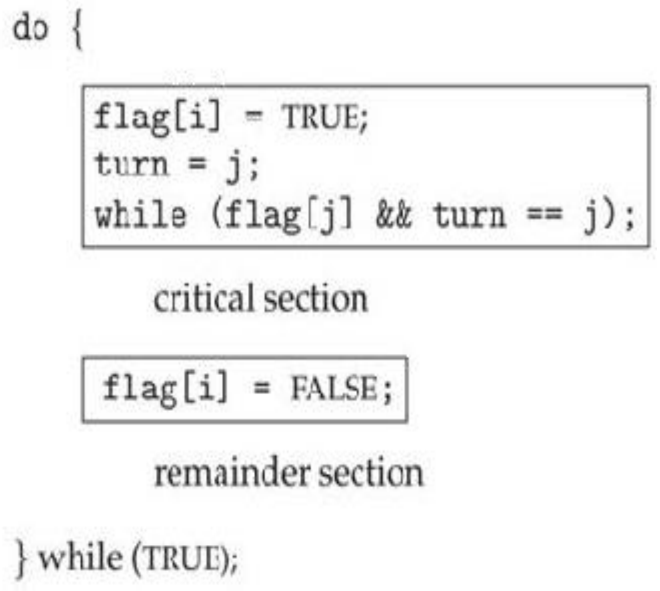
Machine instructions are atomic, high level instructions are not (count++; this is actually 3 machine level instructions, an interrupt can occur in the middle of instructions).

Various proposals for achieving mutual exclusion, so that while one process is busy updating shared memory in its CS, no other process will enter its CS and cause trouble.

- Disabling Interrupts
- Lock Variables
- Strict Alternation
- Peterson's Solution
- The TSL instructions (Hardware approach)

3. **Peterson's Solution (Software approach)**

- A classic software-based solution to the critical-section problem known as Peterson's solution.
- Does not require strict alternation.
- Peterson's solution is restricted to two processes that alternate execution between their CSs and remainder sections. The processes are numbered P_0 and P_1.
- Peterson's solution requires two data items to be shared between the two processes:
  - int turn;
  - boolean flag[2];
    - The variable **turn** indicates whose turn it is to enter its CS. That is, if turn == i, then process P_i is allowed to execute in its CS.
    - The **flag array** is used to indicate if a process is ready to enter its CS. For example, if flag [i] is true, this value indicates that P_i is ready to enter its CS.
  - The algorithm for Peterson's solution is seen in Figure.



```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

Figure: The structure of process P_i in Peterson's solution.

- To enter the CS, process P_i first sets flag [j] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the CS, it can do so.
- If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately.
- The eventual value of turn decides which of the two processes is allowed to enter its CS first.

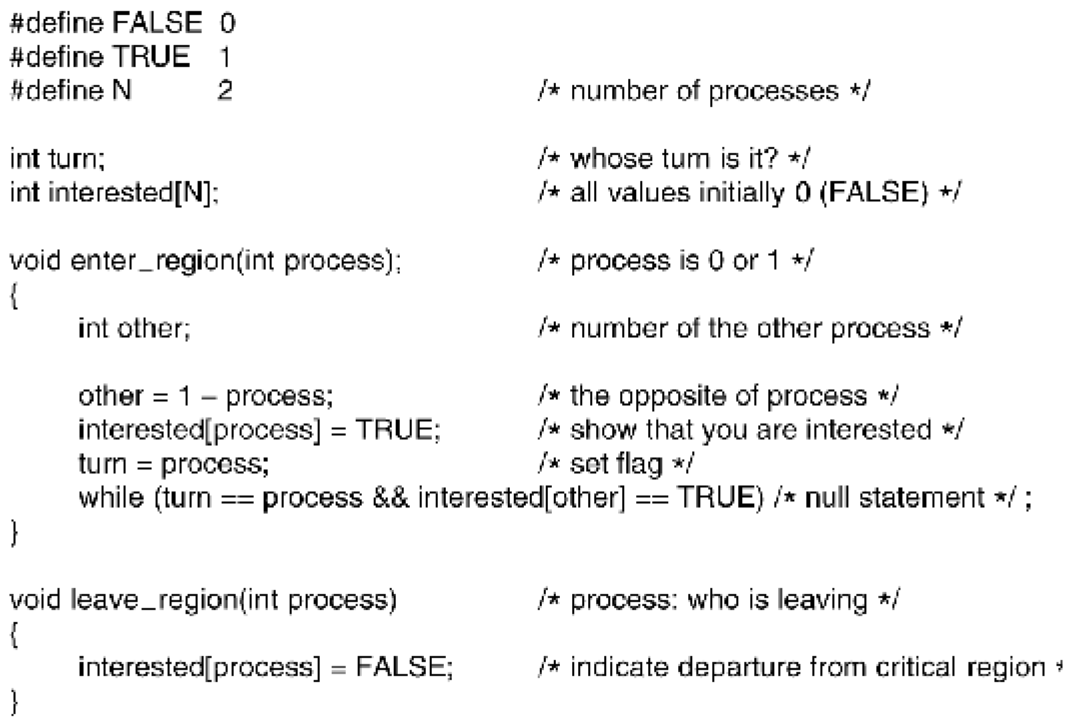
- **Mutual exclusion is preserved**.
- Each P_i enters its CS only if either flag[j] == false or turn == i.
- Also note that, if both processes can be executing in their CSs at the same time, then flag [0] == flag [1] == true. These two observations imply that P_0 and P_1could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both.
- Hence, one of the processes say P_j must have successfully executed the while statement, whereas P_i had to execute at least one additional statement ("turn == j").

- However, since, at that time, flag[j]== true, and turn==j , and this condition will persist as long as P_j is in its CS, the result follows: Mutual exclusion is preserved.

* **The progress requirement is satisfied & the bounded-waiting requirement is met**.
  - A process P_i can be prevented from entering the CS only if it is stuck in the while loop with the condition flag[j] ==true and turn == j; this loop is the only one possible.
  - If P_j is not ready to enter the CS, then flag[j]= false , and P_i can enter its CS.
  - If P_j has set flag [j] to true and is also executing in its while statement, then either turn==i or turn ==j.
  - If turn==i, then P_i will enter the CS. If turn=j, then P_j will enter the CS.
  - However, once P_i exits its CS, it will reset flag[j] to false, allowing P_i to enter its CS.
  - If P_j resets flag[j] to true, it must also set *turn* to i.
  - Thus, since P_i does not change the value of the variable turn while executing the while statement, P_i will enter the CS (progress) after at most one entry by P_j (bounded waiting).

3.1 Drawbacks of Peterson's solutions and corresponding solution

* Burns CPU cycles (requires busy waiting, can be extended to work for n processes, but overhead, cannot be extended to work for an unknown number of processes, unexpected effects).



```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose tum is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region *
}
```

Figure : Peterson's solution for achieving mutual exclusion.

* **Sleep and wakeup**. Peterson's solution has not only the defect of requiring **busy waiting** but it can also have unexpected effects;
  - Consider a computer with two processes, H, with high priority and L, with low priority. The scheduling rules are such that H is run whenever it is in ready state. At a certain moment, with L in its critical region, H becomes ready to run (e.g., an I/O

operation completes). H now begins busy waiting, but since L is never scheduled while H is running, L never gets the chance to leave its critical region, so H loops forever. This situation is sometimes referred to as the priority inversion problem.

- IPC primitive that blocks instead of wasting CPU time (while loop) when they are not allowed to enter their CRs. One of the simplest is the pair **sleep** and **wakeup**. Sleep is a system call that causes the caller to block, that is, be suspended until another process wakes it up. The wakeup call has one parameter, the process to be awakened.

3.2 Synchronization Hardware

In computer science, the **test-and-set** instruction is an instruction used to write to a memory location and return its old value as a single atomic (i.e., non-interruptible) operation. If multiple processes may access the same memory and if a process is currently performing a test-and-set, no other process may begin another test-and-set until the first process is done.

Software implementation of test-and-set

Many processors have an atomic test-and-set machine language instruction. Those that do not can still implement an atomic test-and-set using an atomic swap (as shown below) or some other atomic read-modify-write instruction.
The test and set instruction when used with boolean values behaves like the following function. Crucially the entire function is executed atomically: no process can interrupt the function mid-execution and hence see a state that only exists during the execution of the function. This code only serves to help explain the behaviour of test-and-set; atomicity requires explicit hardware support and hence can't be implemented as a simple function. NOTE: In this example, 'lock' is assumed to be passed by reference (or by name) but the assignment to 'initial' creates a new value (not just copying a reference).

```
function TestAndSet(boolean lock) {
    boolean initial = lock
    lock = true
    return initial
}
```

The above code segment is not atomic in the sense of the test-and-set instruction. It also differs from the descriptions of DPRAM hardware test-and-set above in that here the "set" value and the test are fixed and invariant, and the "set" part of the operation is done regardless of the outcome of the test, whereas in the description of DPRAM test-and-set, the memory is set only upon passage of the test, and the value to set and the test condition are specified by the CPU. Here, the value to set can only be 1, but if 0 and 1 are considered the only valid values for the memory location, and "value is nonzero" is the only allowed test, then this equates to the case described for DPRAM hardware (or, more specifically, the DPRAM case reduces to this under these constraints). From that viewpoint, this can correctly be called "test-and-set" in the full conventional sense of the term. The essential point to note, which this software function does embody, is the general intent and principle of test-and-set: that a value both is tested and is set in one atomic operation such that no other program thread might cause the target memory location to change after it is tested but before it is set, which would violate the logic requiring that the location will only be set when it has a certain value. (That is, critically, as opposed to merely when it very recently *had* that value.)
In the C programming language, the implementation would be like:

```
 #define LOCKED 1
 int TestAndSet(int* lockPtr) {
     int oldValue;
     oldValue = SwapAtomic(lockPtr, LOCKED);
     return oldValue == LOCKED;
 }
```
Where *SwapAtomic* atomically first reads the current value pointed to by *lockPtr* and then writes 1 to the location. Being atomic, *SwapAtomic* never uses cached values and always commits to the shared memory store (RAM).

The code also shows that TestAndSet is really two operations: an atomical swap and a test. Only the swap needs to be atomic. (This is true because delaying the value comparison itself by any amount of time will not change the result of the test, once the value to test has been obtained. Once the swap acquires the initial value, the result of the test has been determined, even if it has not been computed yet—e.g. in the C language example, by the == operator.)

Implementing mutual exclusion with test-and-set
```
boolean lock = false
function Critical(){
    while TestAndSet(lock)
        skip //spin until lock is acquired
    critical section //only one process can be in this section at a time
    lock = false //release lock when finished with the critical section
}
```
This function can be called by multiple processes, but it is guaranteed that only one process will be in the critical section at a time.

Both the above technique and "Test and Test-and-set" are examples of a spinlock: the while-loop spins waiting to acquire the lock.

4. Semaphores

- A synchronization tool called **semaphore**. Semaphores are variables that are used to signal the status of shared resources to processes.
- Dijkstra (1965) suggested using an integer variable to count the number of wakeups saved for future use. In his proposal, a new variable type, called a semaphore, was introduced.
- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait ( )(sleep) and signal( )(wakeup).
- A semaphore could have the value 0, indicating that no wakeups were saved, or some positive value if one or more wakeups were pending.
- Two operations, **down** and **up** (generalizations of sleep and wakeup, respectively);
- The definition of wait( )is as follows:
```
wait (S) {
 while S <= 0
 ;// no-op
S--;
   }
```
- The definition of Signal( )is as follows:
```
signal (S) {
    S++;
    }
```
- All the modifications to the integer value of the semaphore in the wait ( ) and signal ( ) operations must be executed **indivisibly**.

- That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
- In addition, in the case of wait ( ), the testing of the integer value of S(S<=0), and its possible modification (S--), must also be executed without interruption.

4.1 Usage

Counting and binary semaphores. The value of a counting semaphore can range over an unrestricted domain. The value of a binary semaphore can range only between 0 and 1. On some systems, binary semaphores are known as **mutex locks**, as they are locks that provide mutual exclusion.

- We can use binary semaphores to deal with the critical-section problem for multiple processes. The n processes share a semaphore, **mutex**, initialized to 1
- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available.
- Each process that wishes to use a resource performs a wait ( ) operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal ( ) operation (incrementing the count).
- When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

4.2 Implementation

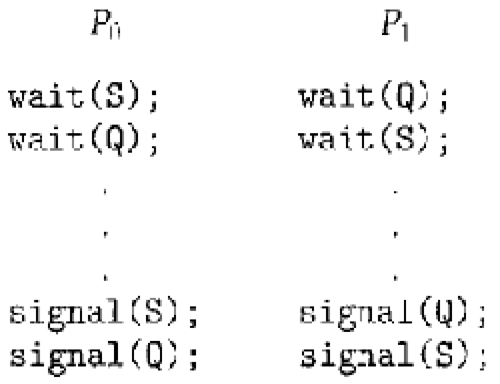
- The main disadvantage of the semaphore definition given here is that it requires busy waiting.
    - While a process is in its CS, any other process that tries to enter its CS must loop continuously in the entry code.
    - Busy waiting wastes CPU cycles that some other process might be able to use productively.
- This type of semaphore is also called a **spinlock** because the process ``spins" while waiting for the lock. (Spinlocks do have an advantage in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus, when locks are expected to be held for short times, spinlocks are useful.)
- A **spinlock** is a lock where the thread simply waits in a loop ("spins") repeatedly checking until the lock becomes available. Since the thread remains active but isn't performing a useful task, the use of such a lock is a kind of busy waiting. Once acquired, spinlocks will usually be held until they are explicitly released, although in some implementations they may be automatically released if the thread being waited on (that which holds the lock) blocks, or "goes to sleep".
- Spinlocks are efficient if threads are only likely to be blocked for a short period of time, as they avoid overhead from operating system process re-scheduling or context switching. For this reason, spinlocks are often used inside operating system kernels. However, spinlocks become wasteful if held for longer durations, preventing other threads from running and requiring re-scheduling. The longer a lock is held by a thread, the greater the risk that it will be interrupted by the O/S scheduler while holding the lock. If this happens, other threads will be left "spinning" (repeatedly trying to acquire the lock), while the thread holding the lock is not making progress towards releasing it. The result is an indefinite postponement until the thread holding the lock can finish and release it. This is especially true on a single-processor system,

where each waiting thread of the same priority is likely to waste its quantum (allocated time where a thread can run) spinning until the thread that holds the lock is finally finished.

- To overcome the need for busy waiting, we can modify the definition of the wait( ) and signal ( )semaphore operations.
    - When a process executes the wait ( ) operation and finds that the semaphore value is not positive, it must wait.
- A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal ( ) operation.
- The process is restarted by a wakeup ( ) operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)
- The list of waiting processes can be easily implemented by a link field in each process control block (PCB). Each semaphore contains an integer value and a pointer to a list of PCBs.
- The critical aspect of semaphores is that they be executed **atomically**. We must guarantee that no two processes can execute wait ( ) and signal ( ) operations on the same semaphore at the same time.

4.3 Deadlocks and Starvation

- The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.
- The event in question is the execution of a signal ( ) operation. When such a state is reached, these processes are said to be **deadlocked**.
- To illustrate this, we consider a system consisting of two processes, P_0 and P_1 , each accessing two semaphores, S and Q, set to the value 1:



```
     P0                P1

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

    - Suppose that P_0 executes wait(S) and then P_1 executes wait (Q).
    - When P_0 executes wait (Q), it must wait until P_1 executes signal (Q).
    - Similarly, when P_1 executes wait(S), it must wait until P_0 executes signal(S).

    - Since these Signal ( ) operations cannot be executed, P_0 and P_1 are deadlocked.
- Another problem related to deadlocks is **indefinite blocking**, or **starvation**, a situation in which processes wait indefinitely within the semaphore.
- Indefinite blocking may occur if we add and remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

5. Classic Problems of Synchronization

We present a number of synchronization problems as examples of a large class of concurrency-control problems. These problems are used for testing nearly every newly proposed synchronization scheme.

5.1 Bounded buffer or producer consumer problem

In computer science, **producer-consumer problem** (also known as the **bounded-buffer problem**) is a classical example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time the consumer is consuming the data (i.e., removing it from the buffer) one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

The solution for the producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer. The solution can be reached by means of inter-process communication, typically using semaphores. An inadequate solution could result in a deadlock where both processes are waiting to be awakened. The problem can also be generalized to have multiple producers and consumers.

Inadequate implementation

This solution has a race condition. To solve the problem, a careless programmer might come up with a solution shown below. In the solution two library routines are used, `sleep` and `wakeup`. When sleep is called, the caller is blocked until another process wakes it up by using the wakeup routine. `itemCount` is the number of items in the buffer.

```
int itemCount;

procedure producer() {
    while (true) {
        item = produceItem();

        if (itemCount == BUFFER_SIZE) {
            sleep();
        }

        putItemIntoBuffer(item);
        itemCount = itemCount + 1;
        if (itemCount == 1) {
            wakeup(consumer);
        }
    }
}

procedure consumer() {
    while (true) {

        if (itemCount == 0) {
            sleep();
        }

        item = removeItemFromBuffer();
        itemCount = itemCount - 1;

        if (itemCount == BUFFER_SIZE - 1) {
            wakeup(producer);
        }

        consumeItem(item);
    }
}
```

The problem with this solution is that it contains a race condition that can lead into a deadlock. Consider the following scenario:

1. The consumer has just read the variable `itemCount`, noticed it's zero and is just about to move inside the if-block.
2. Just before calling sleep, the consumer is interrupted and the producer is resumed.
3. The producer creates an item, puts it into the buffer, and increases `itemCount`.
4. Because the buffer was empty prior to the last addition, the producer tries to wake up the consumer.
5. Unfortunately the consumer wasn't yet sleeping, and the wakeup call is lost. When the consumer resumes, it goes to sleep and will never be awakened again. This is because the consumer is only awakened by the producer when `itemCount` is equal to 1.
6. The producer will loop until the buffer is full, after which it will also go to sleep.

Since both processes will sleep forever, we have run into a deadlock. This solution therefore is unsatisfactory.

Using semaphores

Semaphores solve the problem of lost wakeup calls. In the solution below we use two semaphores, `fillCount` and `emptyCount`, to solve the problem. `fillCount` is the number of items to be read in the buffer, and `emptyCount` is the number of available spaces in the buffer where items could be written. `fillCount` is incremented and `emptyCount` decremented when a new item has been put into the buffer. If the producer tries to decrement `emptyCount` while its value is zero, the producer is put to sleep. The next time an item is consumed, `emptyCount` is incremented and the producer wakes up. The consumer works analogously.

```
semaphore fillCount = 0; // items produced
semaphore emptyCount = BUFFER_SIZE; // remaining space

procedure producer() {
    while (true) {
        item = produceItem();
        down(emptyCount);
            putItemIntoBuffer(item);
        up(fillCount);
    }
}

procedure consumer() {
    while (true) {
        down(fillCount);
            item = removeItemFromBuffer();
        up(emptyCount);
        consumeItem(item);
    }
}
```

The solution above works fine when there is only one producer and consumer. Unfortunately, with multiple producers or consumers this solution contains a serious race condition that could result in two or more processes reading or writing into the same slot at the same time. To understand how this is possible, imagine how the procedure `putItemIntoBuffer()` can be implemented. It could contain two actions, one determining the next available slot and the other writing into it. If the procedure can be executed concurrently by multiple producers, then the following scenario is possible:

1. Two producers decrement `emptyCount`
2. One of the producers determines the next empty slot in the buffer
3. Second producer determines the next empty slot and gets the same result as the first producer
4. Both producers write into the same slot

To overcome this problem, we need a way to make sure that only one producer is executing `putItemIntoBuffer()` at a time. In other words we need a way to execute a critical section with mutual exclusion. To accomplish this we use a binary semaphore called mutex. Since the value of a binary semaphore can be only either one or zero, only one process can be executing between `down(mutex)` and `up(mutex)`. The solution for multiple producers and consumers is shown below.

```
semaphore mutex = 1;
semaphore fillCount = 0;
semaphore emptyCount = BUFFER_SIZE;

procedure producer() {
    while (true) {
        item = produceItem();
        down(emptyCount);
            down(mutex);
                putItemIntoBuffer(item);
            up(mutex);
        up(fillCount);
    }
    up(fillCount); //the consumer may not finish before the producer.
}

procedure consumer() {
    while (true) {
        down(fillCount);
            down(mutex);
                item = removeItemFromBuffer();
            up(mutex);
```

```
          up(emptyCount);
          consumeItem(item);
     }
}
```

Notice that the order in which different semaphores are incremented or decremented is essential: changing the order might result in a deadlock.

5.2 Dining philosopher's problem

In computer science, the **dining philosophers problem** is an example problem often used in concurrent algorithm design to illustrate synchronization issues and techniques for resolving them.

Illustration of the dining philosopher's problem

Problem statement

Five silent philosophers sit at a table around a bowl of spaghetti. A fork is placed between each pair of adjacent philosophers.
Each philosopher must alternately think and eat. Eating is not limited by the amount of spaghetti left: assume an infinite supply. However, a philosopher can only eat while holding both the fork to the left and the fork to the right (an alternative problem formulation uses rice and chopsticks instead of spaghetti and forks).
Each philosopher can pick up an adjacent fork, when available, and put it down, when holding it. These are separate actions: forks must be picked up and put down one by one.
The problem is how to design a discipline of behavior (a concurrent algorithm) such that each philosopher won't starve, i.e. can forever continue to alternate between eating and thinking.

Issues

The problem was designed to illustrate the problem of avoiding deadlock, a system state in which no progress is possible.
One idea is to instruct each philosopher to behave as follows:

- think until the left fork is available; when it is, pick it up;
- think until the right fork is available; when it is, pick it up
- eat
- put the left fork down
- put the right fork down
- repeat from the start

This solution is incorrect: it allows the system to reach deadlock, namely, the state in which each philosopher has picked up the fork to the left, waiting for the fork to the right to be put down.

Resource starvation might also occur independently of deadlock if a particular philosopher is unable to acquire both forks because of a timing problem. For example there might be a rule that the philosophers put down a fork after waiting five minutes for the other fork to become available and wait a further five minutes before making their next attempt. This scheme eliminates the possibility of deadlock (the system can always advance to a different state) but still suffers from the problem of live lock. If all five philosophers appear in the dining room at exactly the same time and each picks up the left fork at the same time the philosophers will wait five minutes until they all put their forks down and then wait a further five minutes before they all pick them up again.
Mutual exclusion is the core idea of the problem, and the dining philosophers create a generic and abstract scenario useful for explaining issues of this type. The failures these philosophers may experience are analogous to the difficulties that arise in real computer programming when multiple programs need exclusive access to shared resources. These issues are studied in the branch of Concurrent Programming. The original problems of Dijkstra were related to external devices like tape drives. However, the difficulties studied in the Dining Philosophers problem arise far more often when multiple processes access sets of data that are being updated. Systems that must deal with a large number of parallel processes, such as operating system kernels, use thousands of locks and synchronizations that require strict adherence to methods and protocols if such problems as deadlock, starvation, or data corruption are to be avoided.

A first solution using semaphores

```
/* program dining philosophers */
semaphore fork[5] = {1};
void philosopher(int i){
  while(true){
    think( );
    wait(fork[i]);            -- take the left fork
    wait(fork[(i+1) mod 5]);  -- then take the right fork
    eat( );
    signal(fork[(i+1) mod 5]);
    signal(fork[i]);
  }
}
void main( ){
  parbegin( philosopher(0), philosopher(1),
                    philosopher(2),philosopher(3), philosopher(4));
}
```

This solution may lead to deadlock

A second solution using semaphores

```
/* program dining philosophers */
semaphore fork[5] = {1};
void philosopher(int i){
  while(true){
    think( );
    <take both forks at once when available>;
    eat( );
    <put down both forks at once>;
  }
}
void main( ){
  parbegin( philosopher(0), philosopher(1),
                  philosopher(2),philosopher(3), philosopher(4) );
}
```

This solution may lead to starvation