/* This Note is compiled from various sources and can be taken as reference only. */

Chapter 3 Process Communication and Synchronization

What is Concurrency?

Concurrency is the concept where multiple tasks or processes are executed **in overlapping time periods** — not necessarily at the same instant.

This is possible in:

- **Single-core systems**: via **context switching**
- **Multi-core systems**: via **true parallelism**

Key Goals of Concurrency:

- Improve **performance**
- Increase **CPU utilization**
- Enable **asynchronous operations** (e.g., I/O)
- Maintain **responsiveness** (especially in UI systems)

Examples:

- A web browser loading pages while downloading files
- An OS handling keyboard input while playing music

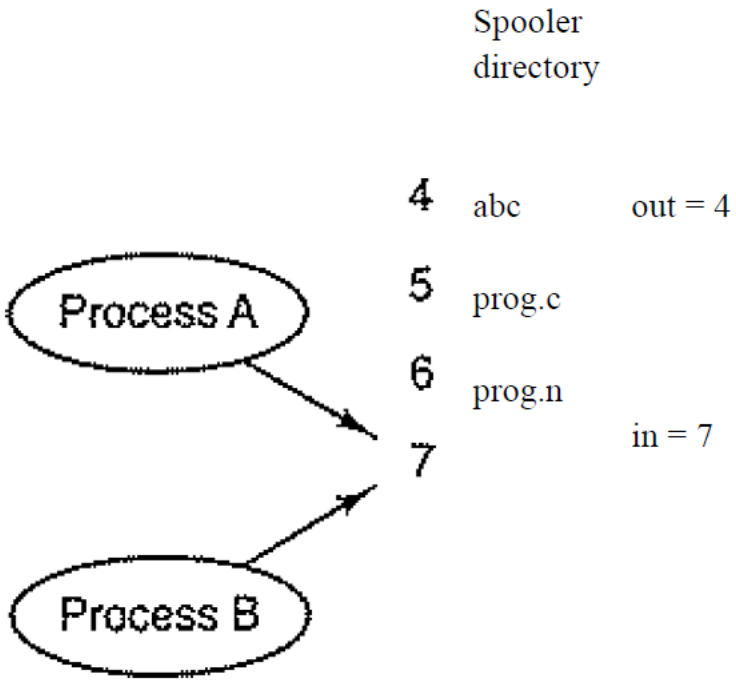
---

2. Race Condition

When two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called race conditions.

Example: *a print spooler*. When a process wants to print a file, it enters the file name in a special spooler directory.
Another process, the printer daemon, periodically checks to see if there are any files to be printed, and if there are, it prints them and then removes their names from the directory.

Consider our spooler directory has a very large number of slots, numbered 0, 1, 2 each one capable of holding a file name. Also imagine that there are two shared variables, ***out***, which points to the next file to be printed, and ***in***, which points to the next free slot in the directory. These two variables might well be kept on a two-word file available to all processes. At a certain instant, slots 0 to 3 are empty and slots 4 to 6 are full. More or less simultaneously, processes A and B decide they want to queue a file for printing.



Process A reads in and stores the value, 7, in a local variable called next-free slot. Just then a clock interrupt occurs and the CPU decides that process A has run long enough, so it switches to process B. Process B also reads in, and also gets a 7. It too stores it in its local variable next-free slot. At this instant both processes think that the next available slot is 7. Process B now continues to run. It stores the name of its file in slot 7 and updates in to be an 8. Then it goes off and does other things. Eventually, process A runs again, starting from the place it left off. It looks at *next-free* slot, finds a 7 there, and writes its file name in slot 7, erasing the name that process B just put there.
Then it computes *next-free* slot + 1, which is 8, and sets in to 8. The spooler directory is now internally consistent, so the printer daemon will not notice anything wrong, but process B will never receive any output.

Critical Region
Sometimes a process has to access shared memory or files, or do other critical things that can lead to races. That part of the program where the shared memory is accessed is called the critical region or critical section. If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid races. Although this requirement avoids race conditions, it is not sufficient for having parallel processes cooperate correctly and efficiently using shared data.
We need four conditions to hold to have a good solution:
1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block other processes.
4. No process should have to wait forever to enter its critical region.

Here process *A* enters its critical region at time Tj. A little later, at time T2 process *B* attempts to enter its critical region but fails because another process is already in its critical region and we allow only one at a time. Consequently, *B* is temporarily suspended until time 73 when A leaves its critical region, allowing *B* to enter immediately. Eventually *B* leaves (at T4) and we are back to the original situation with no processes in their critical regions.

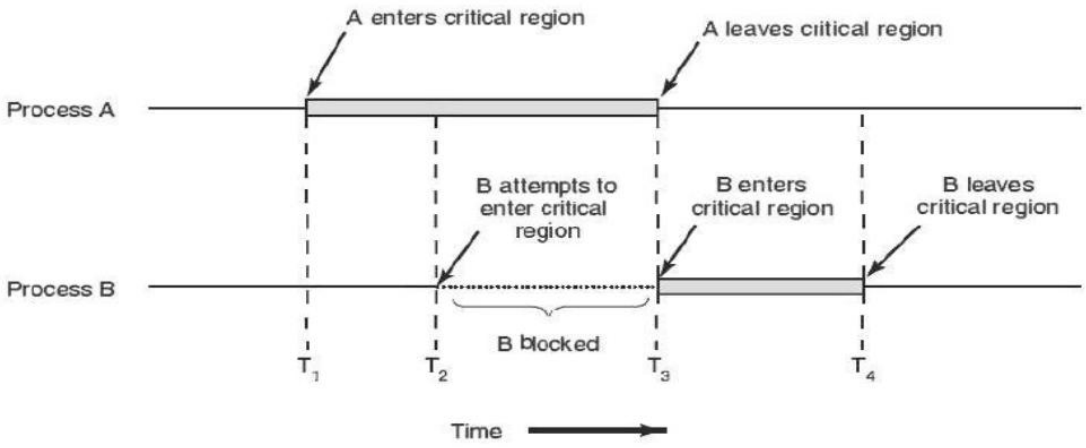


Figure 1. Mutual exclusion using critical regions.

Mutual Exclusion with Busy Waiting

In this section we will examine various proposals for achieving mutual exclusion, so that while one process is busy updating shared memory in its critical region, no other process will enter *its* critical region and cause trouble.

1. Disabling Interrupts

On a single-processor system, the simplest solution is to have each process disable all interrupts just after entering its *critical region* and re-enable them just before leaving it. With interrupts disabled, no clock interrupts can occur. The CPU is only switched from process to process as a result of clock or other interrupts, after all, and with interrupts turned off the CPU will not be switched to another process. Thus, once a process has disabled interrupts, it can examine and update the shared memory without fear that any other process will intervene.

This approach is generally unattractive because it is unwise to give user processes the power to turn off interrupts. Suppose that one of them did it, and never turned them on again? That could be the end of the system. Furthermore, if the system is a multiprocessor (with two or possibly more CPUs) disabling interrupts affects only the CPU that executed the disable instruction. The other ones will continue running and can access the shared memory.

2. Lock Variables

As a second attempt, let us look for a software solution. Consider having a single, shared (lock) variable, initially 0. When a process wants to enter its critical region, it first tests the lock. If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process

just waits until becomes 0. Thus, a 0 means that no process is in its critical region, and a 1 means that some process is in its critical region.
Unfortunately, this idea contains exactly the same fatal flaw that we saw in the spooler directory. Suppose that one process reads the lock and sees that it is 0. Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1. When the first process runs again, it will also set the lock to 1, and two processes will be in their critical regions at the same time.

3. Strict Alternation

Two Process Solution

```
#define FALSE 0
#define TRUE 1
while (TRUE)
{
while (turn != 0) ;
critical _region();
turn = 1;
noncritical_region();
}
```

```
#define FALSE 0
#define TRUE 1
while (TRUE)
{
while (turn != 1);
critical _region();
turn = 0;
noncritical _region();
}
```

Semaphores
Semaphore is a simply a variable. This variable is used to solve critical section problem and to achieve process synchronization in the multiprocessing environment. The two most common kinds of semaphores are counting semaphores and binary semaphores. Counting semaphore can take non-negative integer values and Binary semaphore can take the value 0 & 1 only.
A semaphore can only be accessed using the following operations:
wait () :- called when a process wants access to a resource
signal ():- called when a process is done using a resource

```
int s = 1;
wait (Semaphore s)
{
while (s==0); /* wait until s>0 */
s=s-1;
}
signal (Semaphore s)
{
s=s+1;
}
```

Here is an implementation of mutual-exclusion using binary semaphores:

```
do
{
wait(s);
// critical section
```

signal(s);
// remainder section
} **while**(1);

Now, let us see how it implements mutual exclusion. Let there be two processes P1 and P2 and a semaphore s is initialized as 1. Now if suppose P1 enters in its critical section then the value of semaphore s becomes 0. Now if P2 wants to enter its critical section then it will wait until s>0, this can only happen when P1 finishes its critical section and calls signal () operation on semaphore s. This way mutual exclusion is achieved. Look at the below image for details.

How Race Conditions Occur

Conditions for a Race Condition:

1. **Shared Resource** (e.g., variable, file)
2. **Concurrency** (threads or processes)
3. **No Synchronization** (lack of locks or control)

Typical Scenarios:

- Incrementing counters
- Writing to shared memory
- Updating UI from multiple threads
- File I/O in multithreaded programs

Preventing Race Conditions

To avoid race conditions, **synchronization mechanisms** are used:

Mechanism	Description
Mutex	Locks a section of code to one thread at a time
Semaphore	Controls access based on counters
Monitors	High-level abstraction for synchronization
Atomic Operations	Hardware-supported operations that are indivisible

Example with Mutex:

```
pthread_mutex_lock(&lock);
counter = counter + 1;
pthread_mutex_unlock(&lock);
```

Now, only one thread can update counter at a time — **no race condition**.

Mutual Exclusion (Mutex for short)

What is Mutual Exclusion?

Mutual Exclusion (MutEx) is a principle that ensures that **only one thread or process accesses a shared resource at a time**.

Implementation:

The concept is implemented using **locking mechanisms**, such as:

- **Mutexes**
- **Semaphores**
- **Monitors**

1. **Mutex (Mutual Exclusion Lock)**

A **mutex** is a programming construct used to **enforce mutual exclusion**. It is a **binary lock** — either **locked** or **unlocked**.

How it works:

1. A thread **locks the mutex** before entering a critical section.
2. Other threads that try to lock it are **blocked**.
3. When the thread **unlocks**, another can proceed.

Example with Pseudocode

```
pthread_mutex_lock(&lock);
// critical section
counter++;
pthread_mutex_unlock(&lock);
```

Only one thread can enter the **critical section** at a time.

2. Semaphore

A **semaphore** is a more general **synchronization tool**. It's a **counter** that controls access to shared resources.

There are two types of semaphores:

Type	Description
Binary Semaphore	Acts like a mutex (0 or 1)
Counting Semaphore	Allows **N threads** to access simultaneously

---

- **How it works:**

  - It has a counter.
  - **Wait (P or down):** Decrease the count. If count < 0, block the process.
  - **Signal (V or up):** Increase the count. Wake up waiting processes if needed.

Example in Pseudocode:

```
semaphore = 3;

wait(semaphore);  // down: count--
  // critical section
signal(semaphore);  // up: count++
```

With a **counting semaphore**, up to 3 threads could access the section concurrently.

Difference Between Mutex and Semaphore

Feature	Mutex	Semaphore
Type	Binary (only 0 or 1)	Binary or Counting
Ownership	Has **owner** (thread-specific)	No ownership (can be released by any thread)
Use Case	Mutually exclusive access	Signaling or resource counting
Blocking	Only one thread at a time	Multiple (if count > 1)
Common Use	Protect a single resource	Manage limited resources (e.g., DB pool)

Message Passing and Monitor

1. **Message Passing**

Message Passing is a way for processes or threads to **communicate and synchronize** by **sending and receiving messages**, instead of sharing memory.

🧠 **Think of it like:**

Two people passing notes — one sends, one receives. They don't need to write on the same paper (shared memory); instead, they pass messages.

Key Features of message passing

- **No shared memory**: Processes don't need to access the same address space.
- Used in **distributed systems**, **microkernels**, and **modular applications**.
- Can be **synchronous** (blocking) or **asynchronous** (non-blocking)

Types of Message Passing

Type	Description
Synchronous	Sender waits until receiver gets the message
Asynchronous	Sender continues without waiting

Example in Pseudocode (Synchronous):

```
send(P1, "Hello")  → blocks until P2 receives
receive(P2)        → gets "Hello"
```

Example in Operating Systems:

- **Linux**: msgsnd(), msgrcv() system calls
- **Windows**: Named pipes or Windows Messages
- **Languages**: Python's multiprocessing.Queue, Java's Socket

---

Advantages:

- Good for **distributed systems**
- Safer (no direct memory sharing)
- Easier to **modularize** components

Disadvantages:

- **Slower** than shared memory
- More **overhead** for small or frequent messages

Monitors

A **monitor** is a high-level **synchronization construct** that:

- Combines **shared data**, **procedures**, and **synchronization** in a single unit
- Ensures **mutual exclusion** automatically: only one thread can execute inside a monitor at a time

🧠 **Think of it like:**

A building with one entrance and a manager. Only one person can be inside the building (critical section) at a time, and the manager decides who gets to go next.

🔹 **Components of a Monitor:**

1. **Shared variables**
2. **Procedures** to operate on those variables
3. **Synchronization mechanisms** (like condition variables)

Monitors are a higher-level synchronization construct that simplifies process synchronization by providing a high-level abstraction for data access and synchronization. Monitors are implemented as programming language constructs, typically in object-oriented languages, and provide mutual exclusion, condition variables, and data encapsulation in a single construct.

1. A monitor is essentially a module that encapsulates a shared resource and provides access to that resource through a set of procedures. The procedures provided by a monitor ensure that only one process can access the shared resource at any given time, and that processes waiting for the resource are suspended until it becomes available.
2. Monitors are used to simplify the implementation of concurrent programs by providing a higher-level abstraction that hides the details of synchronization. Monitors provide a structured way of sharing data and synchronization information, and eliminate the need for complex synchronization primitives such as semaphores and locks.
3. The key advantage of using monitors for process synchronization is that they provide a simple, high-level abstraction that can be used to implement complex concurrent systems. Monitors also ensure that synchronization is encapsulated within the module, making it easier to reason about the correctness of the system.

However, monitors have some limitations. For example, they can be less efficient than lower-level synchronization primitives such as semaphores and locks, as they may involve additional overhead due to their higher-level abstraction. Additionally, monitors may not be suitable for all types of synchronization problems, and in some cases, lower-level primitives may be required for optimal performance.

The monitor is one of the ways to achieve Process synchronization. The monitor is supported by programming languages to achieve mutual exclusion between processes. For example Java Synchronized methods. Java provides wait() and notify() constructs.

Bounded buffer program with Moniotr

```
monitor BoundedBuffer {
   buffer[10];
   int count = 0;
   condition notFull, notEmpty;

   procedure insert(item) {
      if (count == 10)
         wait(notFull);       // wait if full
      buffer[count] = item;
      count++;
      signal(notEmpty);       // signal consumer
   }

   procedure remove() {
      if (count == 0)
         wait(notEmpty);      // wait if empty
      item = buffer[count - 1];
      count--;
      signal(notFull);        // signal producer
      return item;
   }
}
```

Classical Problems of Process Synchronization

Dining Philosopher's problem

The **Dining Philosopher Problem** is a classic synchronization and concurrency problem that deals with resource sharing, deadlock, and starvation in systems where multiple processes require limited resources.

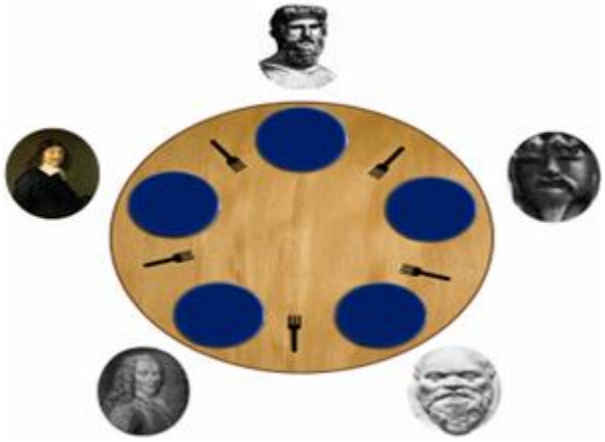
Problem Statement

The **Dining Philosopher Problem** involves 'n' philosophers sitting around a circular table. Each philosopher alternates between two states: **thinking** and **eating**. To eat, a philosopher needs two chopsticks, one on their left and one on their right. However, the number of chopsticks is equal to the number of philosophers, and each chopstick is shared between two neighboring philosophers.

The standard problem considers the value of 'n' as 5 i.e. we deal with 5 Philosophers sitting around a circular table.

Constraints and Conditions for the Problem

- Every Philosopher needs two chopsticks to eat.
- Every Philosopher may pick up the chopsticks on the left or right but only one chopstick at once.
- Philosophers only eat when they have two chopsticks. We have to design such a protocol i.e. pre and post protocol which ensures that a philosopher only eats if he or she has two chopsticks.
- Each chopstick is either clean or dirty.



Solution

The correctness properties it needs to satisfy are:

- **Mutual Exclusion Principle:** No two Philosophers can have the two chopsticks simultaneously.
- **Free from Deadlock:** Each philosopher can get the chance to eat in a certain finite time.
- **Free from Starvation:** When few Philosophers are waiting then one gets a chance to eat in a while.
- **No strict Alternation**
- **Proper utilization of time**

Algorithm

```
loop forever
    p1: think
    p2: preprotocol
    p3: eat
    p4: postprotocol
```

First Attempt

We assume that each philosopher is initialized with its index I and that addition is implicitly modulo 5. Each chopstick is modeled as a semaphore where wait corresponds to taking a chopstick and signal corresponds to putting down a chopstick.

Algorithm

```
semaphore array[0..4] chopstick ← [1, 1, 1, 1, 1]
loop forever
        p1 : think
        p2 : wait(chopstick[i])
        p3 : wait(chopstick[i + 1]%n)
        p4 : eat
        p5 : signal(chopstick[i])
        p6 : signal(chopstick[i + 1])
```

Problem with this solution:

This solution may lead to a deadlock under an interleaving that has all the philosophers pick up their left chopsticks before any of them tries to pick up a right chopstick. In this case, all the Philosophers are waiting for the right chopstick but no one will execute a single instruction.

Second Attempt

One way to tackle the above situation is to limit the number of philosophers entering the room to four. By doing this one of the philosophers will eventually get both the chopstick and execute all the instruction leading to no deadlock.

Algorithm

```
semaphore array[0..4] chopstick ← [1, 1, 1, 1, 1]

  semaphore room ← 4
  loop forever
      p1 : think
      p2 : wait(room)
      p3 : wait(chopstick[i])
      p4 : wait(chopstick[i + 1])
      p5 : eat
      p6 : signal(chopstick[i])
      p7 : signal(chopstick[i + 1])
      p8 : signal(room)
```

In this solution, we somehow interfere with the given problem as we allow only four philosophers.

Homework : Chandy/Misra Solution for Dining Philosopher's Problem. <सबैले गर्ने सोध्ने Roll No<=10>

The Dining Philosophers Problem — Simplified

🧠 **The Idea**

Imagine **5 philosophers** sitting around a circular table.
They do two things:

1. **Think**
2. **Eat**

To **eat**, they need **two chopsticks** — one on the left and one on the right.
Each chopstick is shared between two philosophers.

Each philosopher:

- Picks up the **left chopstick**
- Picks up the **right chopstick**
- Eats
- Puts both chopsticks down
- Thinks again

⚠️ **The Problem**

If **all philosophers pick up their left chopstick at the same time**, they will all **wait forever** for the right chopstick — which leads to a **deadlock** (no one can proceed).

☑ **Solutions (Made Simple)**

There are a few smart tricks to solve this.

☑ **1. Numbering and Order Rule**

Only allow **4 philosophers** to sit at the table at a time, or:

Odd philosopher picks up **left chopstick first**,
Even philosopher picks up **right chopstick first**.

This breaks the symmetry and avoids circular waiting.

☑ 2. Use a Waiter (Arbiter)

Introduce a **"waiter"** who controls who gets to eat.

Before picking up chopsticks, a philosopher **asks permission**.
The waiter only allows a maximum of 4 philosophers to pick up chopsticks at once — so deadlock can't happen.

☑ 3. Use Semaphores or Locks (Coding solution)

In programming, each chopstick is a **lock**.

We can:

- Use **mutexes** for each chopstick.
- Use a global **semaphore** with a value of 4 to limit simultaneous access.

This way, at most 4 philosophers can try to eat at the same time, again preventing deadlock.

The readers-writer problem

The readers-writer problem in operating systems is about managing access to shared data. It allows multiple readers to read data at the same time without issues but ensures that only one writer can write at a time, and no one can read while writing is happening. This helps prevent data corruption and ensures smooth operation in multi-user systems. Probably the most fundamental problem in concurrent programming is to provide safe access to shared resources. A classic problem used to illustrate this issue is Readers-Writers. It is a significant problem for showing how data structures might be synchronized such that consistency is guaranteed and efficiency is ensured. The Readers-Writers problem refers specifically to situations where a number of processes or threads may possibly have access to some common resource, like a database or a file. It also gives rise to the need for good synchronization mechanisms so as not to bias the requirement of readers against that of writers in access to the resource, considering the integrity of data.

What is The Readers-Writers Problem?

The Readers-Writers Problem is a classic synchronization issue in operating systems that involves managing access to shared data by multiple threads or processes. The problem addresses the scenario where:

- **Readers**: Multiple readers can access the shared data simultaneously without causing any issues because they are only reading and not modifying the data.
- **Writers**: Only one writer can access the shared data at a time to ensure data integrity, as writers modify the data, and concurrent modifications could lead to data corruption or inconsistencies.

Challenges of the Readers-Writer Problem

The challenge now becomes how to create a synchronization scheme such that the following is supported:

- **Multiple Readers**: A number of readers may access simultaneously if no writer is presently writing.
- **Exclusion for Writers**: If one writer is writing, no other reader or writer may access the common resource.

Solution of the Readers-Writer Problem

There are two fundamental solutions to the Readers-Writers problem:

- **Readers Preference:** In this solution, readers are given preference over writers. That means that till readers are reading, writers will have to wait. The Writers can access the resource only when no reader is accessing it.
- **Writer's Preference:** Preference is given to the writers. It simply means that, after arrival, the writers can go ahead with their operations; though perhaps there are readers currently accessing the resource.

Focus on the solution Readers Preference in this paper. The purpose of the Readers Preference solution is to give a higher priority to the readers to decrease the waiting time of the readers and to make the access of resource more effective for readers.

Problem Parameters

- One set of data is shared among a number of processes
- Once a writer is ready, it performs its write. Only one writer may write at a time
- If a process is writing, no other process can read it
- If at least one reader is reading, no other process can write
- Readers may not write and only read

Solution When Reader Has the Priority Over Writer

Here priority means, no reader should wait if the share is currently open for reading. There are four types of cases that could happen here.

Case	Process 1	Process 2	Allowed/Not Allowed
Case 1	Writing	Writing	Not Allowed
Case 2	Writing	Reading	Not Allowed
Case 3	Reading	Writing	Not Allowed
Case 4	Reading	Reading	Allowed

Three variables are used: **mutex, wrt, readcnt** to implement a solution.

1. **semaphore** mutex, wrt; // semaphore **mutex** is used to ensure mutual exclusion when **readcnt** is updated i.e. when any reader enters or exits from the critical section, and semaphore **wrt** is used by both readers and writers
2. **int** readcnt; //**readcnt** tells the number of processes performing read in the critical section, initially 0

Functions for Semaphore

Semaphores are synchronization tools used in operating systems to manage access to shared resources by multiple threads or processes. They use simple integer values and two main operations to control access:

- **wait() :** decrements the semaphore value.
- **signal() :** increments the semaphore value.

Writer Process

- Writer requests the entry to critical section.
- If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
- It exits the critical section.

```
do {
   // writer requests for critical section
```

```
    wait(wrt);

    // performs the write

    // leaves the critical section
    signal(wrt);

} while(true);
```

Reader Process

- Reader requests the entry to critical section.
- If allowed:
    - it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the **wrt** semaphore to restrict the entry of writers if any reader is inside.
    - It then, signals mutex as any other reader is allowed to enter while others are already reading.
    - After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore "wrt" as now, writer can enter the critical section.
- If not allowed, it keeps on waiting.

```
do {

    // Reader wants to enter the critical section
    wait(mutex);

    // The number of readers has now increased by 1
    readcnt++;

    // If this is the first reader, lock wrt to block writers
    // this ensure no writer can enter if there is even one reader
    // thus we give preference to readers here
    if (readcnt==1)
        wait(wrt);

    //Allow other readers to enter by unlocking the mutex
    // other readers can enter while this current reader is inside the critical section
    signal(mutex);

    // current reader performs reading here
    // a reader wants to leave
    //Lock the mutex to update readcnt
    wait(mutex);

    //The number of readers has now decreased by 1
    readcnt--;
```

```
    // If, no reader is left in the critical section, unlock wrt to allow writers
    if (readcnt == 0)
        signal(wrt);        // writers can enter

    //Allow other readers or writers to proceed by unlocking the mutex
    signal(mutex);

} while(true);
```

- Thus, the semaphore '**wrt**' is queued on both readers and writers in a manner such that preference is given to readers if writers are also there. Thus, no reader is waiting simply because a writer has requested to enter the critical section.

The Readers ad Writers Problem

```
semaphore mutex = 1; // Controls access to the reader count
semaphore db = 1; // Controls access to the database
int reader_count; // The number of reading processes accessing the data

Reader()
{
while (TRUE) { // loop forever
down(&mutex); // gain access to reader_count
reader_count = reader_count + 1; // increment the reader_count
if (reader_count == 1)
down(&db); // if this is the first process to read the database,
// a down on db is executed to prevent access to the
// database by a writing process
up(&mutex); // allow other processes to access reader_count
read_db(); // read the database
down(&mutex); // gain access to reader_count
reader_count = reader_count - 1; // decrement reader_count
if (reader_count == 0)
up(&db); // if there are no more processes reading from the
// database, allow writing process to access the data
up(&mutex); // allow other processes to access reader_countuse_data();
// use the data read from the database (non-critical)
}

Writer()
{
while (TRUE) { // loop forever
create_data(); // create data to enter into database (non-critical)
```

```
down(&db); // gain access to the database
write_db(); // write information to the database
up(&db); // release exclusive access to the database
}
```

A data object is to be shared among several concurrent processes. Some of these processes may want to only to read the content of the shared object, whereas others may want to update the shared object. We distinguish between these two types of processes by referring to those processes that are interested in only reading as **readers**, and to the rest as **writers**. Obviously, if two readers access the shared data object simultaneously, no adverse effects will result.
The readers-writers problem has several variations, all involving priorities. The simplest one, referred to as the **first** readers-writers problem, requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. The **second** readers-writers problem requires that, once a writer is ready, that writer performs its write as soon as possible. Here, the first readers-writers problem has been explained. At a given time, there is only one writer and any number of readers. When a writer is writing, the readers cannot enter into the database. The readers need to wait until the writer finishes writing on the database. Once a reader succeeds in reading the database, subsequent readers can enter into the critical section (in this case, say, and database) without waiting for the precedent reader finish to read. On the other hand, a writer who arrives later than the reader who is reading currently is required to wait the last reader finish to read. Only when the last reader finishes reading, the writer can enter into the critical section and is able to write on the database.

Producer consumer problem:

- It is also known as bounded buffer problem. It is a multi-process synchronization problem.
- The problem describes two processes, the producer and consumer, who share common fixed size buffer.
- Producer: the producer's job is to generate a piece of data, put into the buffer and start again.
- Consumer: the consumer is consuming the data (i.e. removing it from the buffer) one piece at a time

Problem:

- The problem is to make sure that the producer won't try to add data into the buffer if it is full and that the customer won't try to remove data from an empty buffer.

Solution:

- Producer either go to sleep or discard data if the buffer is full
- The next time the Consumer removes the item from the buffer and it notifies the producer who starts to fill the buffer again.
- Consumer can go to sleep if it finds the buffer to be empty.
- The next time the producer puts data into the buffer , it weeks up the sleeping customer

<Self Study -Sleeping Barber Problem>

Deadlock: Introduction:

In a multiprogramming environment, several processes may compete for finite number of resources. A process request resources; if the resources are not available at that time, the process enters in a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called **deadlock** i.e. a deadlock is a situation in which two or more transactions are waiting for one another to give up locks. A set of processes is in a deadlock state if every process in the set is waiting for an event (release) that can only be caused by some other process in the same set.

Example:

Process-1 requests the printer, gets it
Process-2 requests the tape unit, gets it
Process-1 requests the tape unit, waits
Process-2 requests the printer, waits

} Process-1 and Process-2 are deadlocked!

Necessary Conditions:

A deadlock situation can arise if the following four conditions hold simultaneously in a system.

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

1. Mutual exclusion

At least one resource must be held in a non-sharable mode i.e. only one process at a time can use the resource. If any other process requests this resource, then that process must wait for the resource to be released.

2. Hold and wait

A process must be holding at least one resource and waiting for at least one resource that is currently being held by some other process.

3. No preemption

Resources cannot be preempted i.e. once a process is holding a resource (i.e. once its request has been granted), then that resource cannot be taken away from that process until the process voluntarily releases it.

4. **Circular wait**

A set {P0, P1, P2, ..., Pn} of waiting processes must exist such that:

- P0 is waiting for a resource held by P1,
- P1 is waiting for a resource held by P2, …
- Pn-1 is holding for a resource held by Pn, and
- Pn is waiting for resource held by P0

Resource Allocation Graph

Deadlock can also be described in terms of a directed graph called resource allocation graph. This graph consists of a set of vertices V and set of edges E. The set of vertices V is portioned into two different types of nodes:

- The set of all actives process P = {P1, P2, P3, … , Pn} in the system and
- The set consisting of all resource types in the system.

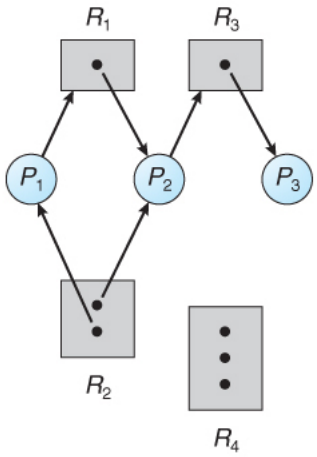


Fig: Resource allocation Graph

Request Edges - A set of directed arcs from Pi to Rj, indicating that process Pi has requested Rj, and is currently waiting for that resource to become available.
Assignment Edges - A set of directed arcs from Rj to Pi indicating that resource Rj has been allocated to process Pi, and that Pi is currently holding resource Rj.
In above diagram, there are:
The sets P, R, and E:
- P = {P1, P2, P3}
- R = {R1, R2, R3, R4}
- E = { P1 → R1, P2 → R3, R1 → P2, R3 → P3, R2 → P1, R2 → P2}

Resources Instances:
- One instance of resource R1 and R3
- Two instance of resource R2
- Three instances of resource R4

Process states:
- Process P1 is holding an instance of resources type R2 and is waiting for an instance of resource type R1.
- Process P2 is holding an instance of resources type R1 and an instance of R2 is waiting for an instance of resource type R3.
- Process P3 is holding an instance of resources type R3

If a resource-allocation graph contains no cycles, then the system is not deadlocked. If the graph contains a cycle, then a deadlock may exist. (In case of single instance of each resources-> deadlock Occurs)
If each process type has exactly one instance, then a cycle implies that a deadlock has occurred.
If the cycle involves only a set of resources types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.
In following diagram, there are two minimal cycles

P1 → R1 → P2 → R3 → P3 → R2 → P1
P2 → R3 → P3 → R2 → P2
Here, Process P1, P2 and P3 are deadlocked. Process P2 is waiting for the resources R3, which is held by process P3. Process P3 is waiting for either process P1 or process P2 to release resource R2. In addition, process P1 is waiting for process P2 to release resource R1.

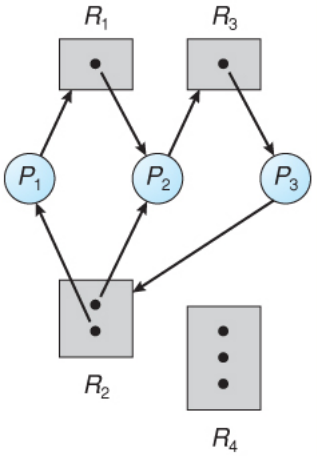


Fig: Resource allocation graph with deadlock
If each resource type has several instances, then a cycle does not necessarily implies that a deadlock has occurred. In this case, a cycle graph is necessary but not a sufficient condition existence of deadlock.
Let us consider following graph. It has: P1 → R1 → P3 → R2 → P1
There is no deadlock. The process P4 may release its instances of resource type R2 and that resource can be allocated to P3, breaking the cycle.

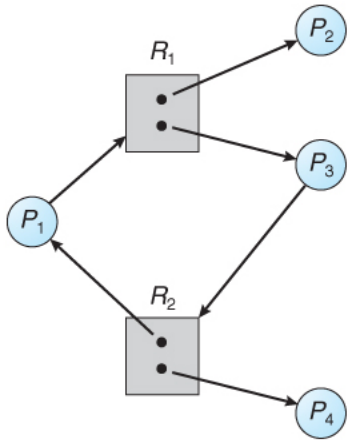


Fig: Resource allocation graph with a cycle but no deadlock
Note:
- If a Resource allocation graph does not have a cycle, the system is not in a deadlock state.
- If there is a cycle, the system may or not be in a deadlock state.

Methods for Handling Deadlocks

• Deadlock prevention
• Deadlock avoidance
• Deadlock detection and recovery
- Deadlock Ignorance

1. Deadlock prevention

Different protocols are used to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state. To prevent the system from deadlocks, one of the four conditions that may create a deadlock should be discarded. The methods for those conditions are as follows:

a) Mutual Exclusion:

The mutual exclusion condition must hold for non shareable Sources. For example, a printer cannot be simultaneously shared by several processes. Read only files are good example of shareable resources. If several process attempts to open a read only file at the same time, they can simultaneously access to the file. A process never needs to wait or a shareable resource. In general, we do not have systems with all resources being sharable. Some resources like printers, processing units are non-sharable. So it is not possible to prevent deadlocks by denying mutual exclusion.

b) Hold and Wait:

• One protocol to ensure that hold-and-wait condition never occurs says each process must request and get all of its resources before it begins execution.
• Another protocol is —Each process can request resources only when it does not occupy any resources.

Example:
The difference between two protocols:
 Let us consider a process that copies data from DVD drive to a file on disk, sorts the file and then prints the result to a printer. If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file and printer. It will hold the printer for its entire execution, even though it needs the printer at the end.
 The second method allows the process to request initially only the DVD drive and disk file. It copies from DVD drive to disk and then releases both the DVD drive and the disk file. The process must then again request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminate.

Both these protocols have two main disadvantages:
 Resource utilization may be low, since resources may be allocated but unused for a long period.
 Starvation is possible. A process that needs several popular resources my may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

c) No Preemption:

There should be no preemption of resources that have already been allocated. To ensure that this condition does not hold we can use following protocols:
 One protocol is —If a process that is holding some resources and requests another resource and that resource cannot be immediately allocated to it (i.e. the process must wait), then all resources the process is currently holding are preempted i.e. it must release all resources that are currently allocated to it. The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it regain its old resources, as well as the new ones that is requesting.

 Another protocol is ―When a process requests some resources, if they are available, allocate them. If a resource it requested is not available, then we check whether,. it is being used or it is allocated to some other process that is waiting for other additional resources. If that resource is not being used, then the OS preempts it from the waiting process and allocate it to the requesting process. If that resource is used, the requesting process must wait.‖

This protocol can be applied to resources whose states can easily be saved and restored such as CPU registers, memory space). It cannot be applied to resources like printers.

d) Circular Wait:

One protocol to ensure that the circular wait condition never holds is ―Impose a linear ordering of all resource types.‖ Then, each process can only request resources in an increasing order of priority.
For example, set priorities for r1 = 1, r2 = 2, r3 = 3, and r4 = 4. With these priorities, if process P wants to use r1 and r3, it should first request r1, then r3.
Another protocol is ―Whenever a process requests a resource rj, it must have released all resources rk with priority (rk) ≥ priority (rj).
For example:
Let us consider R = {R1, R2, R3, … , Rn} be the set of resource types. For example, if the set of resources types R includes: tape drives, disk drives, and printer, then the function F might me defined as follows:
F (tape drive) = 1
F (disk drive) = 5
F (printer) = 12
We can now consider the following protocol to prevent deadlocks. Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of resource type – Say R_i. After that, the process can request instances of resource type Rj if and only if $F(R_j) > F(R_i)$.

2. Deadlock avoidance

Deadlock can be avoided if certain information about processes is available to the operating system before allocation of resources, such as which resources a process will consume in its lifetime. For every resource request, the system sees whether granting the request will mean that the system will enter an *unsafe* state, meaning a state that could result in deadlock. The system then only grants requests that will lead to *safe* states. In order for the system to be able to determine whether the next state will be safe or unsafe, it must know in advance at any time:

- resources currently available
- resources currently allocated to each process
- resources that will be required and released by these processes in the *future*

Deadlock avoidance algorithm can be classified into following two methods:
a) Safe state
b) Resource allocation graph algorithm

Safe state:
A state is safe if the system can allocate resources to each process in some order and still avoid a deadlock.
Let us consider a system with 12 magnetic tape drives and 3 processes P_0, P_1, and P_2. The process P_0 requires 10 tape drives; P_1 may need 4 tape drives and process P_2 needs up to 9 tape drives. Suppose at time T_0, process P_0 is holding 5 tape drives, process P_1 is holding 2 tape drives and process P_2 is holding 2 tape drives. It is shown in following table

	Maximum Needs	Current needs
P0	10	5
P1	4	2
P2	9	2

At the time T_0, the system is in safe state. The sequence $< P_1, P_0, P_2 >$ satisfies the safety condition.
Process P_1 can immediately be allocated all its tape drives and then return them (the system will have 5 available tape drives), then process P_0 can get all its tape drives and return them (the system will have 10 available tape drives) and finally process P_2 can get all tape drives and return them (the system will then have all 12 tape drives available).
A system can go from a safe state to unsafe state. Suppose that, at time T_1, process P_2 requests and is allocated one more tape drive. The system is no longer in safe state. At this state only process P_1 can be allocated all its tape drives. When it returns them, the system will have only four available tape drives. Since process P_0 is allocated 5 tape drives but has a maximum of 10, it may request 5 more tape drives. If it does so, it will have to wait, because they are unavailable. Similarly, process P_2 may request 6 additional tape drives and have to wait, resulting in deadlock.
The mistake was in granting the request from process P_2 for 1 more tape drive. If the process P2 wait until either of the other processes had finished and released its resources, then we could have avoided the deadlock.

Banker's Algorithm:
The resource allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type. Banker's algorithm is applicable for such system but is less efficient than the resource-allocation graph. This algorithm requires prior information about the maximum number of each resource class that each process may request. Using this information we can define a deadlock avoidance algorithm.
When a new process enters the system, it must declare the maximum number of instances of each resources type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.
Several data structures must be maintained to implement the banker's algorithm. Some of them are, where ***n*** is the number of processes in the system and ***m*** is the number of resources types:
- **Available:** a vector of length **m** indicates the number of available resources of each type. If ***Available[j]*** equals ***k***, then ***k*** instances of resource type R_j are available.
- **Max:** an ***n x m*** matrix defines the maximum demand of each process. If ***Max[i][j]*** equals **k**, then process P_i may request at most ***k*** instances of resource type ***Rj.***
- **Allocation:** An ***n x m*** matrix defines the number of resources of each type currently allocated to each process. If ***Allocation[i][j]*** equals ***k***, then process P_i is currently allocated ***k*** instances of resource type ***Rj***
- **Need:** An ***n x m*** matrix indicates the remaining resource need of each process. If ***Need[i][j]*** equals ***k,*** then process Pi may need ***k*** more instances of resource type ***Rj*** to complete its task.

Note: Need [i][j] = matrix [i][j] – Allocation [i][j]

Safety Algorithm:

This algorithm can be described as follows:

1. Let Work and Finish be vectors of length m and n respectively. Intitizlize Work = Available and Finish[i] = false for i = 0, 1, 2, … , n – 1

2. Find an index i such that both

a. Finish[i] = = false

b. $Need_i \leq Work$

If no such i exists, go to step 4.

3. Work = Work + Allocation

Finish[i] = true

Go to step 2

4. If Finish[i] == true for all i, then the system is in safe state.

Example:

Let us consider a system with 5 processes P_0 to P_4 and three resource types A, B, and C. Resource type A has 10 instances, resource type B has 5 instances, and resource type C has 7 instances. Suppose that, at time T_0, the following snapshots of the system has been taken:

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

Then content of the matrix Need is defined to be Max – Allocation and is as follows:

	Allocation		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

The system is currently in safe state. The sequence $<P_1, P_3, P_4, P_2, P_0>$ satisfies the safety criteria.

Deadlock detection and recovery.
If a system does not employ a protocol to ensure that deadlocks will never occur, then a deadlock-detection and recovery scheme is employed.
Detection:
A deadlock detection algorithm must be invoked to determine whether a deadlock has occurred i.e.it determines the existence of the deadlock.
Single instance of each resource type:
If all resources have only a single instance, then we can define a deadlock-detection algorithm by a ***wait-for graph***. This graph can be obtained by resource allocation graph by removing the resource nodes and collapsing the appropriate edges.

Several instance of a resource type:
The wait-for graph is not applicable for a resource-allocation system with multiple instances of each resource type. For this banker's algorithm is applicable.

Recovery from Deadlock
Detection algorithm determines the existence of the deadlock. There are two options for breaking a deadlock: simply abort one or more processes to break the circular wait and to preempt some resources from one or more of the deadlocked processes. It is described in following section:
A. Process termination

Following methods are used to eliminate deadlocks by aborting a process.

Abort all deadlocked process: This method will break the deadlock cycle. . Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on a printer, the system must reset the printer to a correct state before printing the next job.

Abort one process at a time until the deadlock cycle is eliminated: This method incurs considerable overhead, since after each process is aborted a deadlock-detection algorithm must be invoked to

determine whether any processes are still deadlocked. Many factors may affect which process is chosen, such as:

- What is the priority of the process?
- How long the process has computed and how much longer the process will compute before completing its designated task.
- How many and what types of resources the process has used
- How many more resources the process needs in order to complete
- How many processes will need to be terminated?
- Whether the process is interactive or batch.

B. Resource preemption

In this method, some resources are preempted from other processes and give these resources to other processes until the deadlock cycle is broken. For preemption following issues are needed to be addressed:

- **Selecting a victim:** for process termination, we must determine the order of preemption to minimize cost. Cost factors includes different parameters such as the number of resources a deadlock process is holding, amount of time the process has consumed during its execution.

- **Roolback:** Rollback means to abort the process and the restart it. In general, it is difficult to determine what a safe state is; the simple solution is a total rollback. This method requires the system to keep more information about the state of all running processes.

- **Starvation:** Generally cost factor is a base for selecting victim process but it may happen that the same process is always picked as a victim and as a result, this process never completes its designated task that means starvation. We have to ensure that a process can be picked as a victim for only a finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

Deadlock Ignorance :

Stick your head in the sand and pretend there is no problem at all, this method of solving any problem is called Ostrich Algorithm. This Ostrich algorithm is the most widely used technique in order to ignore the deadlock and also it used for all the single end-users uses. If there is deadlock in the system, then the OS will reboot the system in order to function well. The method of solving any problem varies according to the people.