1 Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Let's consider an example to illustrate these conditions:

Suppose we have two processes, P1 and P2, and two resources, R1 and R2. Both processes require both resources to complete their tasks. The following scenario demonstrates a potential deadlock:

1. Process P1 starts and acquires resource R1.
2. Process P2 starts and acquires resource R2.
3. P1 requests resource R2, but it is held by P2, so P1 enters a waiting state.
4. P2 requests resource R1, but it is held by P1, so P2 enters a waiting state.

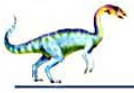
At this point, both processes are waiting for resources held by each other, forming a circular wait. Since none of the processes can proceed without acquiring the requested resource, a deadlock occurs.

In this example, the conditions for deadlock are satisfied: mutual exclusion (only one process can hold a resource at a time), hold and wait (processes hold allocated resources while waiting for others), no preemption (resources cannot be forcibly taken away), and circular wait (a circular chain of processes waiting for resources).

2 Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore S – integer variable
- Two standard operations modify S: `wait()` and `signal()`
  - Originally called `P()` and `V()`
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

```
wait (S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
signal (S) {
    S++;
}
```



Semaphore Usage

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1
  - Also called mutex locks.
- Can implement a counting semaphore S as a binary semaphore
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

Mutual exclusion implementation with semaphores

- Initially mutex=1 for p0 process

- **do{**
- **Wait(mutex);**
- **//critical section**
- **Signal(mutex);**
- **//remainder section**
- **}while(TRUE);**

- But this solution requires busy waiting and wastes CPU cycles
  - This type of semaphore called a spinlock because the process spins while waiting for lock.
  - If locks applied for short time spin lock is useful.

Semaphore Implementation with no Busy waiting

- With each semaphore 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 the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

```
typedef struct{
   int value;
   struct process *list;
} semaphore;

wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   } }
signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

- Illustration of the problem:

3 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 `counter` that keeps track of the number of full buffers. Initially, `counter` 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.

Producer

```
while (true) {
   /* produce an item in next produced */

   while (counter == BUFFER_SIZE) ;
      /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
   counter++;
}
```

Consumer

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

4 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
  - Writers – can both read and write
- 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 treated – all involve priorities
- Shared Data
  - Data set
  - Semaphore **rw_mutex** initialized to 1
  - Semaphore **mutex** initialized to 1
  - Integer **read_count** initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process

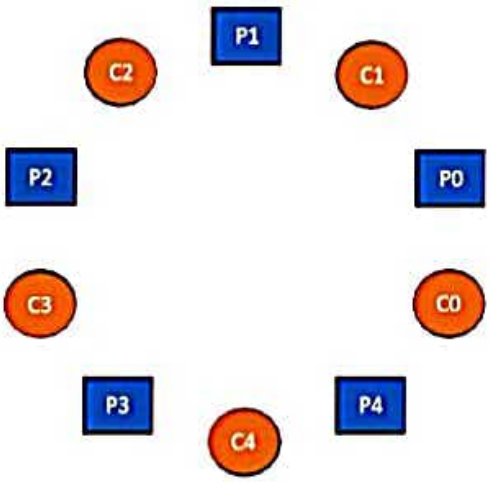
```
do {
      wait(rw_mutex);
     ...
      /* writing is performed */
     ...
   signal(rw_mutex);
} while (true);
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

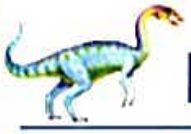
```
do {
     wait(mutex);
     read count++;
     if (read_count == 1)
   wait(rw_mutex);
 signal(mutex);
 ...
     /* reading is performed */
 ...
 wait(mutex);
     read count--;
     if (read_count == 0)
   signal(rw_mutex);
 signal(mutex);
} while (true);
```

5 Dining-Philosophers Problem



P = Philosopher
C = Chopstick

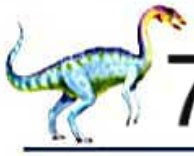
- There are three states of the philosopher: THINKING, HUNGRY, and EATING. Here there are two semaphores: Mutex and a semaphore array for the philosophers.
- Mutex is used such that no two philosophers may access the pickup or putdown at the same time. The array is used to control the behavior of each philosopher
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1



Dining-Philosophers Problem Algorithm

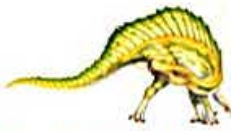
- The structure of Philosopher *i*:

```
do {
    wait (chopstick[i] );
     wait (chopStick[ (i + 1) % 5] );

        // eat

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

        // think

} while (TRUE);
```



7 Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  1. Priority of the process
  2. How long process has computed, and how much longer to completion
  3. Resources the process has used
  4. Resources process needs to complete
  5. How many processes will need to be terminated
  6. Is process interactive or batch?



Recovery from Deadlock: Resource Preemption

- **Selecting a victim** – minimize cost
- **Rollback** – return to some safe state, restart process for that state
- **Starvation** – same process may always be picked as victim, include number of rollback in cost factor

Resource Allocation Graph: 8

A resource allocation graph is a graphical representation used to illustrate the allocation and utilization of resources in a system. It visually depicts the relationships between processes and resources, showing which processes are currently holding or requesting which resources. It helps in analyzing resource allocation and detecting potential deadlocks.

In a resource allocation graph, the following symbols are used:

- Circle (O): Represents a process.
- Rectangle (R): Represents a resource.

Arrows are used to indicate the relationships between processes and resources. There are two types of arrows:

- Request Edge (R): Represents a process requesting a resource.
- Assignment Edge (A): Represents a resource being held by a process.

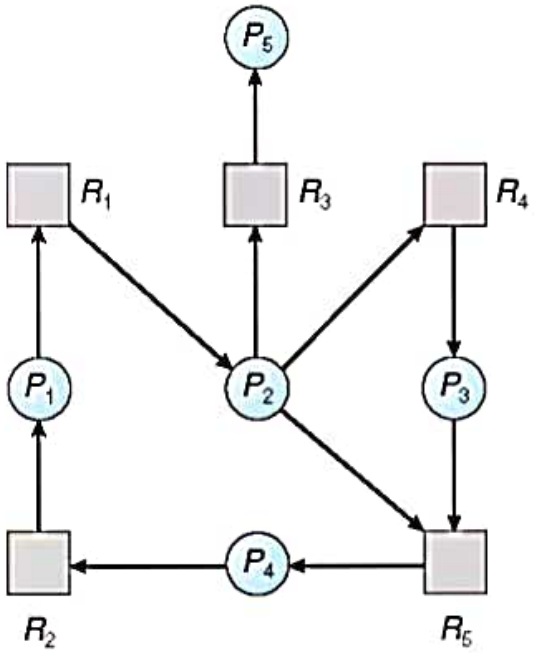
Wait-for Graph:

A wait-for graph (or wait-for diagram) is another graphical representation used to analyze the relationships among processes and their respective wait-for dependencies. It focuses on the wait-for relationships that arise when processes request resources and have to wait for other processes to release them. Wait-for graphs are useful in detecting deadlocks and understanding the sequence of process interactions.

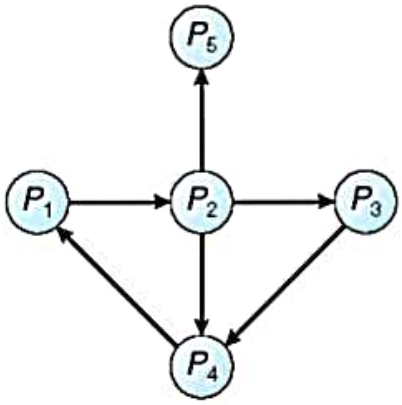
In a wait-for graph, the following symbols are used:

- Circle (O): Represents a process.
- Arrow (→): Represents a dependency, indicating that one process is waiting for another process.

Resource-Allocation Graph and Wait-for Graph



(a)



(b)

Resource-Allocation Graph

Corresponding wait-for graph