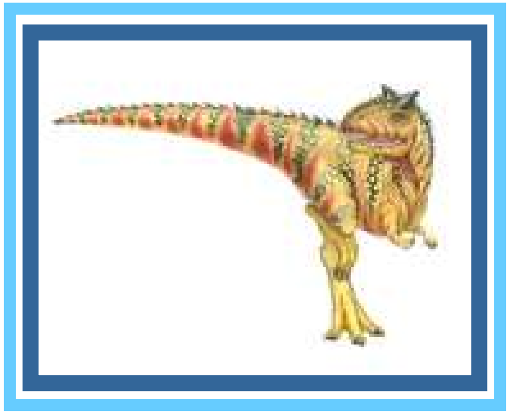
Chapter 5: Process Synchronization



Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Producer Consumer problem:

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.

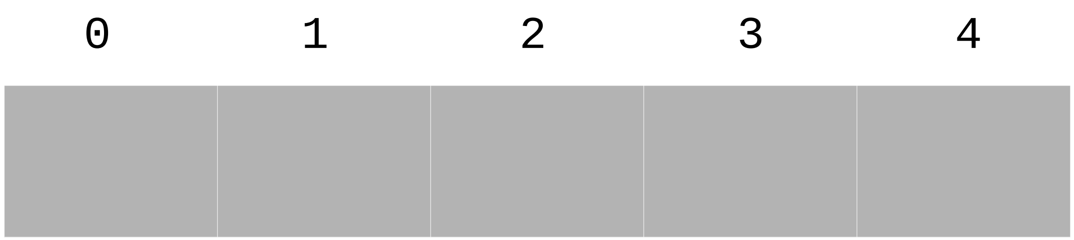
number of produced items

counter=0 --> consumer have to wait
counter=buffer_size --> producer have to wait

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++;
}
```



Buffer

Consumer

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

0	1	2	3	4

Buffer

Race Condition

- **`counter++`** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **`counter--`** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "counter = 5" initially:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 – 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

Critical Section Problem

Several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition.

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design a protocol to solve the problem of race condition.
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process P_i

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

Algorithm for Process P_i

```
do {

   while (turn == j);

        critical section

   turn = j;

        remainder section

 } while (true);
```

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the ***n*** processes

Critical-Section Handling in OS

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

- **Preemptive** – allows preemption of process when running in kernel mode race condition
  - Shared kernel data structures are free from race conditions; difficult to design for SMP architectures
- **Non-preemptive** – runs until it exits kernel mode, blocks, or voluntarily yields control of the CPU No race condition
  - Essentially free of race conditions in kernel mode

Which one to choose? Why?

Consider Responsiveness and thus more suitable for RT programming

Peterson's Solution

- Good algorithmic description of solving the problem
- **Two process** solution
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- Load and store refers to loading between memory & register
- Load/store approach requires both the operands in the register
- The two processes share two variables:
  - **`int turn;`**
  - **`Boolean flag[2]`**
- The variable **`turn`** indicates whose turn it is to enter the critical section
- The **`flag`** array is used to indicate if a process is ready to enter the critical section. **`flag[i] = true`** implies that process P_i is ready!

Algorithm for Process P_i

```
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
        critical section
   flag[i] = false;
        remainder section
 } while (true);
```

easy

Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

     P_i enters CS only if:

     either **`flag[j] = false`** or **`turn = i`**
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

Peterson's Solution (Cont.)

- Peterson's solution is not likely to work on modern computers due to the nature of load/store instructions

bcz due to the atomic condition CPU have to wait bcz memory is slow swap in swap out took time

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

```
do {
   acquire lock
       critical section
   release lock
       remainder section
} while (TRUE);
```

test_and_set Instruction

Definition:

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

1. Executed atomically
2. Returns the original value of passed parameter

returns whatever value you passed

3. Set the new value of passed parameter to "TRUE".

Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

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

compare_and_swap Instruction

Definition:

```
int compare _and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
       *value = new_value;
 return temp;
}
```

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

Solution using compare_and_swap

- Shared integer "lock" initialized to 0;
- Solution:

```
do {
     while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */
   /* critical section */
 lock = 0;
   /* remainder section */
} while (true);
```

test_and_swap & compare_and_swap

- Both the algorithms compare_and_swap and test_and_set provides for mutual exclusion.
- They do not satisfy the bounded-waiting requirements.
- Next slide solves this bounded-waiting issue.

Bounded-waiting Mutual Exclusion with test_and_set

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;
   /* critical section */
   /* end of CS */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = false;
   else
      waiting[j] = false;
   /* remainder section */
} while (true);
```

DS in common are initialized to false.
Boolean waiting[n] and
Boolean lock

1) Pi can enter only if waiting[i] is false or key = false. Value of key can become false if test_and_set is executed ! Once executed only this process enters CS, rest waits. waiting[i] can become false only if another process leaves CS

2) As the process exiting CS either sets lock to false or sets waiting[j] to false; another process waiting for CS will have chance to enter CS

3) When process leaves the CS, it scans the array in the cyclic order i+1, ...n,, 0, 1, ….i-1) and examines waiting[j] = true? So if any process is waiting, it will get chance to enter at least in n-1 rounds.

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock.
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean variable available indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**
  - **No context switch is required ! Useful when locks are held for a short period of time**

acquire() and release()

- ```
  acquire() {
      while (!available)
          ; /* busy wait */
      available = false;;
  }
  ```
- ```
  release() {
      available = true;
  }
  ```
- ```
  do {
      acquire lock
          critical section
      release lock
          remainder section
  } while (true);
  ```
  ```

Sample Programs

Refer programs

sample_mutex_prog1.c

sample_mutex_prog2.c

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable. Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Note: Originally called **`P()`** and **`V()`** **`….wait…..signal.`**
- Definition of the **`wait() operation`**

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

- Definition of the **`signal() operation`**

```
signal(S) {
    S++;
}
```

Semaphore Usage

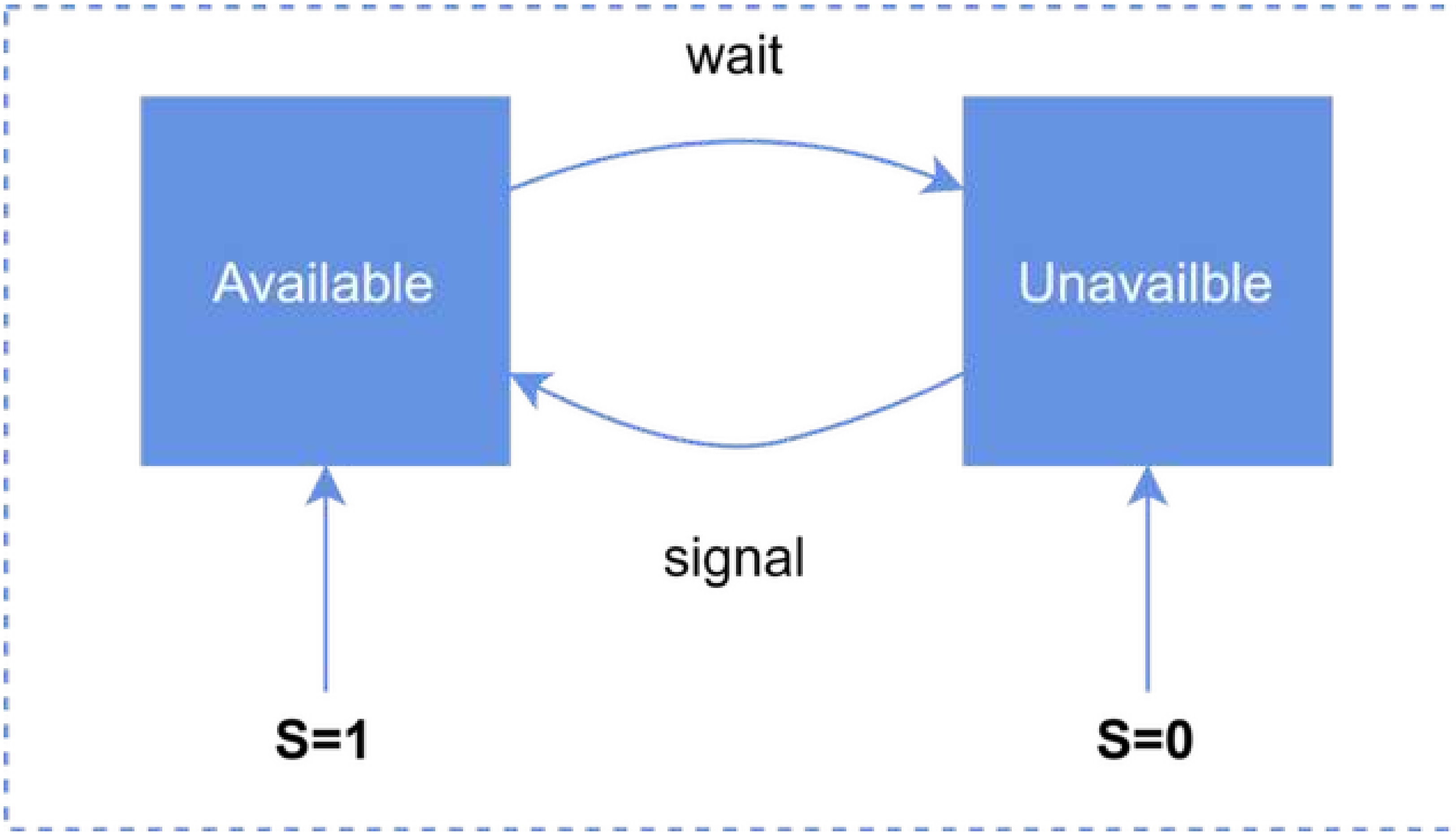
- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider P_1 and P_2 concurrent processes that require S_1 to happen before S_2

  Create a shared semaphore "**synch**" initialized to **0**

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

- Can implement a counting semaphore S as a binary semaphore on systems that do not provide mutex lock

Binary Semaphore



Semaphore Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that for applications that spend lot of time in critical sections this is not a good solution

Semaphore Implementation with no Busy waiting

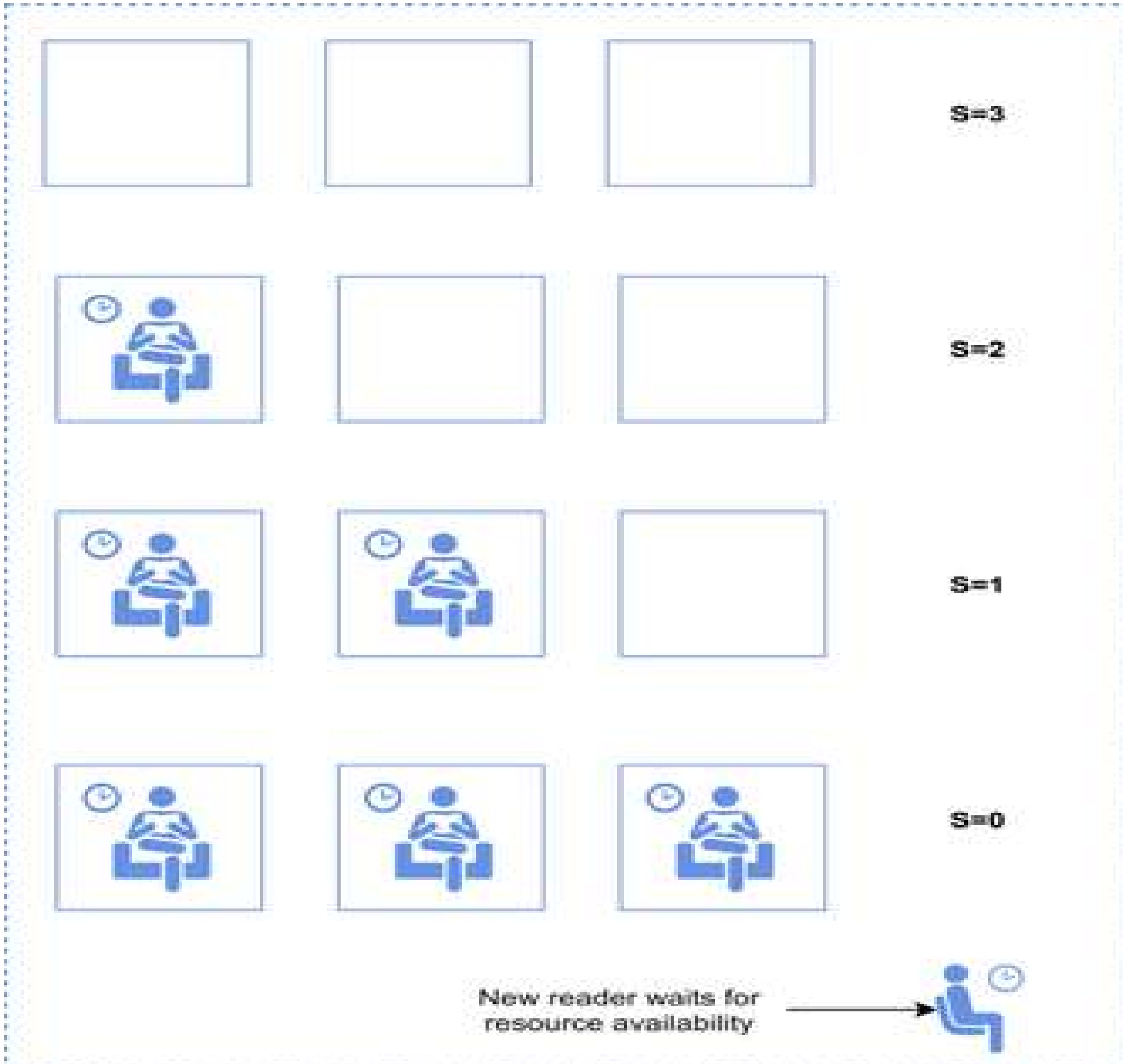
- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue
-
```
typedef struct{
 int value;
 struct process *list;
 } semaphore;
```

Implementation with no Busy waiting (Cont.)

```
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);
   }
}
```

Semaphore Analogy - Library example



Implementation with no Busy waiting (Cont.)

To declare a semaphore, the data type is **sem_t**.

A semaphore is initialized by using **sem_init**(for processes or threads) or sem_open (for IPC).

sem_init(sem_t *sem, int pshared, unsigned int value);

sem : Specifies the semaphore to be initialized.

pshared : This argument specifies whether or not the newly initialized semaphore is shared between processes or between threads. A non-zero value means the semaphore is shared between processes and a value of zero means it is shared between threads.

value : Specifies the value to assign to the newly initialized semaphore.

Implementation with no Busy waiting (Cont.)

To destroy a semaphore, we can use sem_destroy.

sem_destroy(sem_t *sem);

To lock a semaphore or wait we can use the sem_wait function:

int sem_wait(sem_t *sem);

To release or signal a semaphore, we use the sem_post function:

int sem_post(sem_t *sem);

Please refer to sample_semaphore_prog1.c

Semaphores vs Mutex

Semaphores	Mutex
Signalling mechanism	Busy locking mechanism
Data type integer	Object
Two types: Couting, Binary	No such types
Multiple threads can access single semaphore simultaneously	Mutex prohibits simultaneous access
Only wait() and signal() operations to modify value	Modifiable by a process accessing the criitical resource

Deadlock – Necessary Conditions

- **Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes**

- **Necessary Conditions of Deadlock**

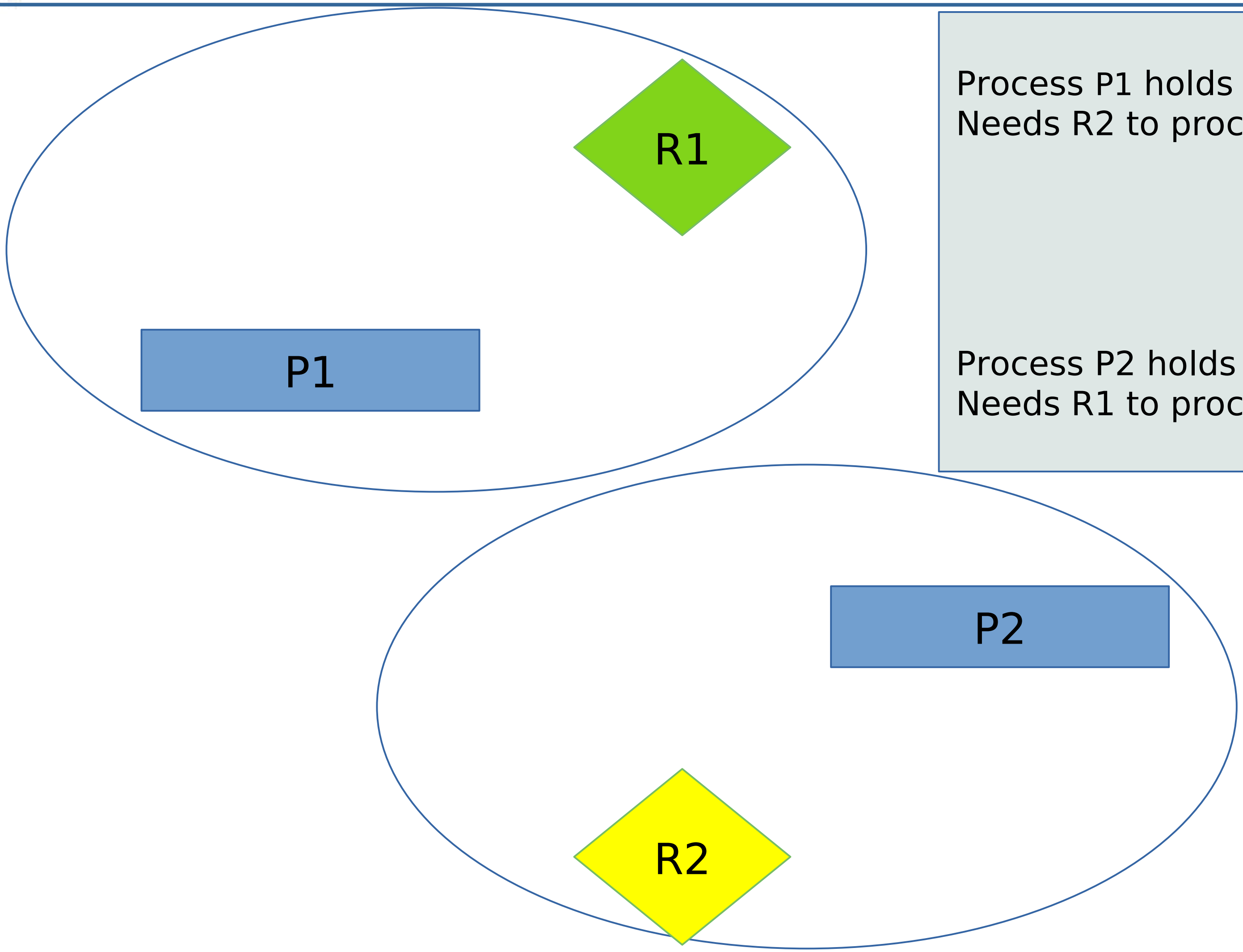
- **Mutual Exclusion:** Only one process can use a resource at any given time i.e. the resources are non-sharable.

- **Hold and wait:** A process is holding at least one resource at a time and is waiting to acquire other resources held by some other process.

- **No preemption:** The resource can be released by a process voluntarily i.e. after execution of the process.

- **Circular Wait:** A set of processes are waiting for each other in a circular fashion.

Deadlock



Process P1 holds resource R1
Needs R2 to proceed further.

Process P2 holds resource R2
Needs R1 to proceed further.

Deadlock – Prevention & Avoidance

- **Deadlock Prevention – Violate at least one of the necessary condition for deadlock.**

- **Deadlock Avoidance (Banker's Algorithm)**

- When a process requests for a resource, before granting the resource, compute the following
- Total Number of Resources (Resource Type, Quantity) [Available]
- Total Number of Resources allocated [Allocation]
- Total Number of Resources requests [Max Demand]
- Total Number of Resources needed by each process [Need = Max - Allocation]
- Compute the state of the system in terms of being Safe or Unsafe

- **Deadlock Detection and Recovery**

Deadlock Vs Starvation

Deadlock	Starvation
More than one process is blocked due to non availability of the resource caused by one of the blocked process(es) holding it.	Process(es) are never allocated resources which they require.
Resources are blocked by a set of processes in a circular fashion.	Resources are continuously used by other possibly high-priority resources.
Can be prevented by avoiding anyone of the necessary condition required for a deadlock or can be recovered using a recovery algorithm.	It can be prevented by aging.
In a deadlock situation, none of the processes get executed.	In starvation, Some (possibly higher priority) processes execute while some processes are postponed indefinitely.
	Starvation is also called LiveLock.

Deadlock and Starvation

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

```
P0                      P1
    wait(S);                wait(Q);
    wait(Q);                wait(S);
    ...                     ...
    signal(S);              signal(Q);
    signal(Q);              signal(S);
```

- **Starvation** – **indefinite blocking in Semaphore Implementation**
  - A process may never be removed from the semaphore queue in which it is suspended (e.g., signal() implements LIFO)
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

End of Chapter 5