240-229: SDA (Operating Systems session)

Lecture 3: Concurrency

Associate Professor Dr.Sangsuree Vasupongayya
sangsuree.v@psu.ac.th
Department of Computer Engineering
Prince of Songkla University

Outline

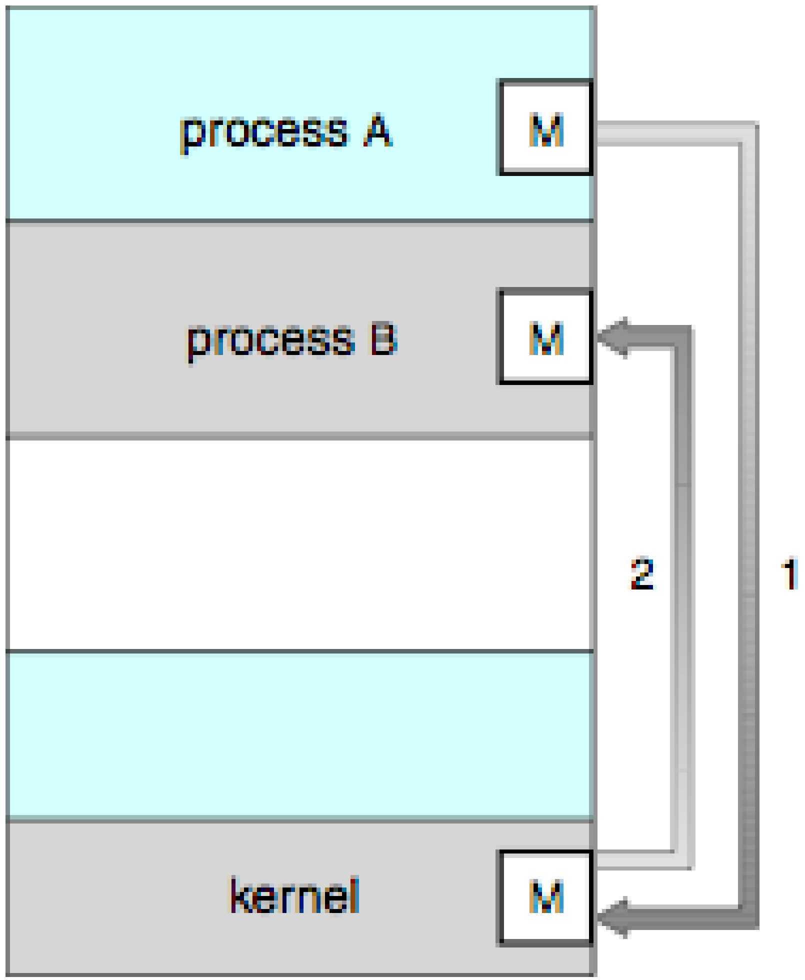
- Lecture:
  - Interprocess communication
  - Concurrency concept
  - The Critical-Section Problem
  - Solutions to Critical-section
  - Deadlock
  - Classic Problems of Synchronization
- Lab:
  - Pipe
  - Shared memory
  - Mutex lock

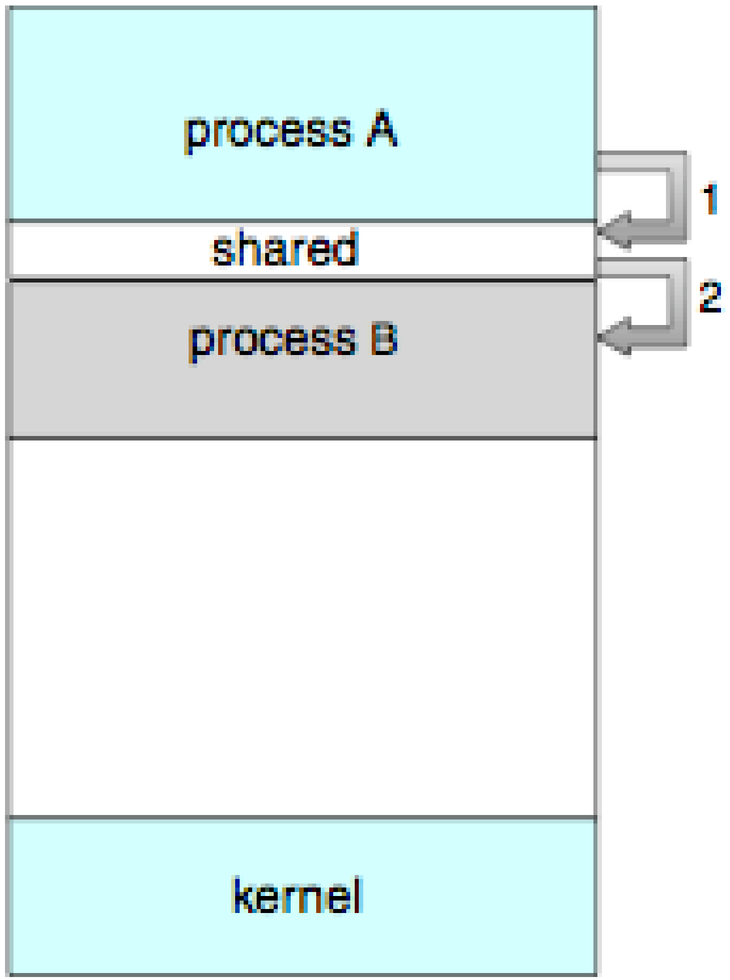
Interprocess Communication (IPC)

- Process can be independent or cooperating
  - Independent – is not affecting or is not affected by the other processes executing in the system
  - Cooperating – can affect or be affected by the other processes executing in the system
- Why IPC?
  - Cooperating process use IPC to exchange data and information
- 2 Types
  - Message passing (e.g., pipe)
  - Shared memory

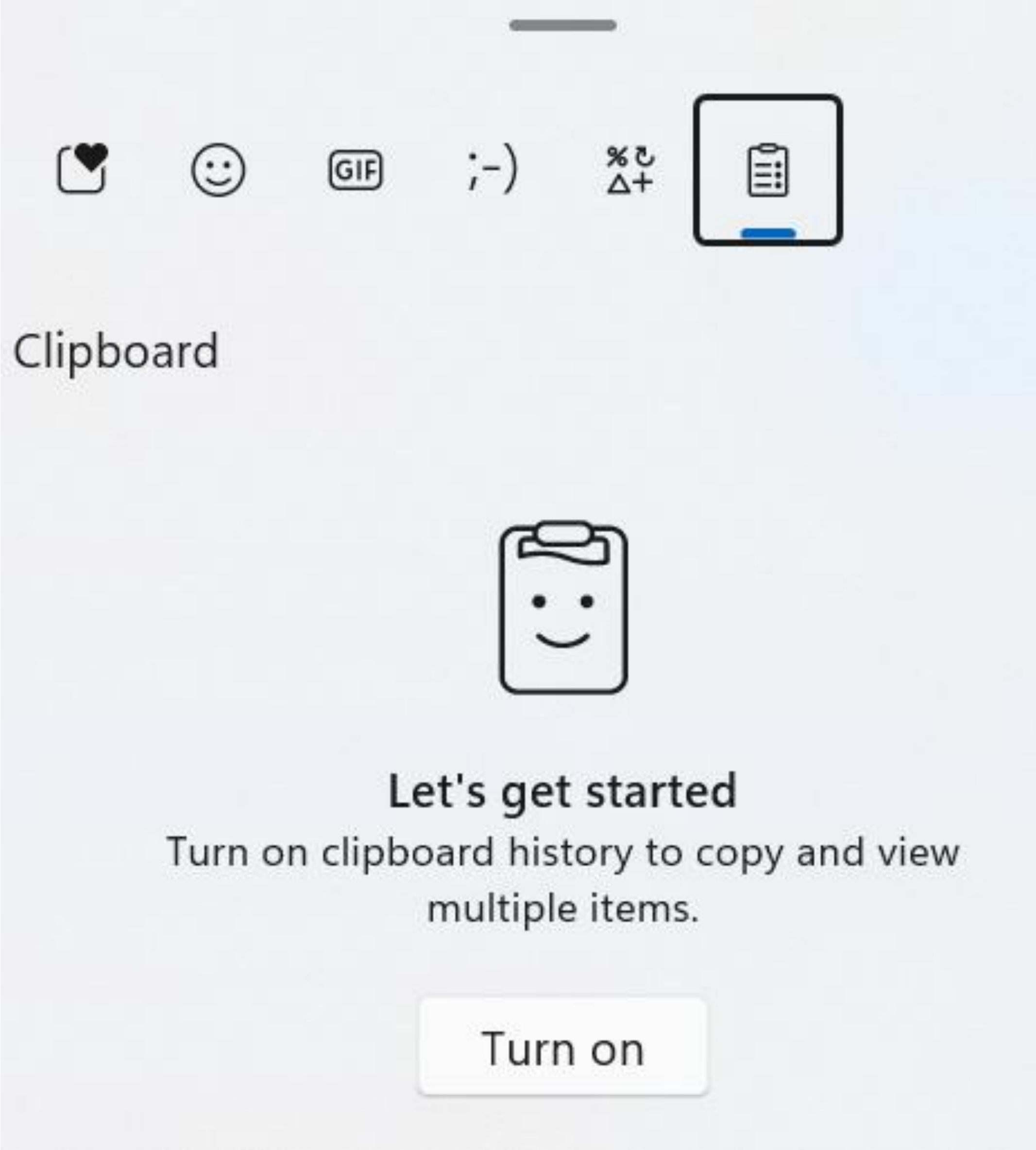
Interprocess Communication

Message Passing **Shared Memory**



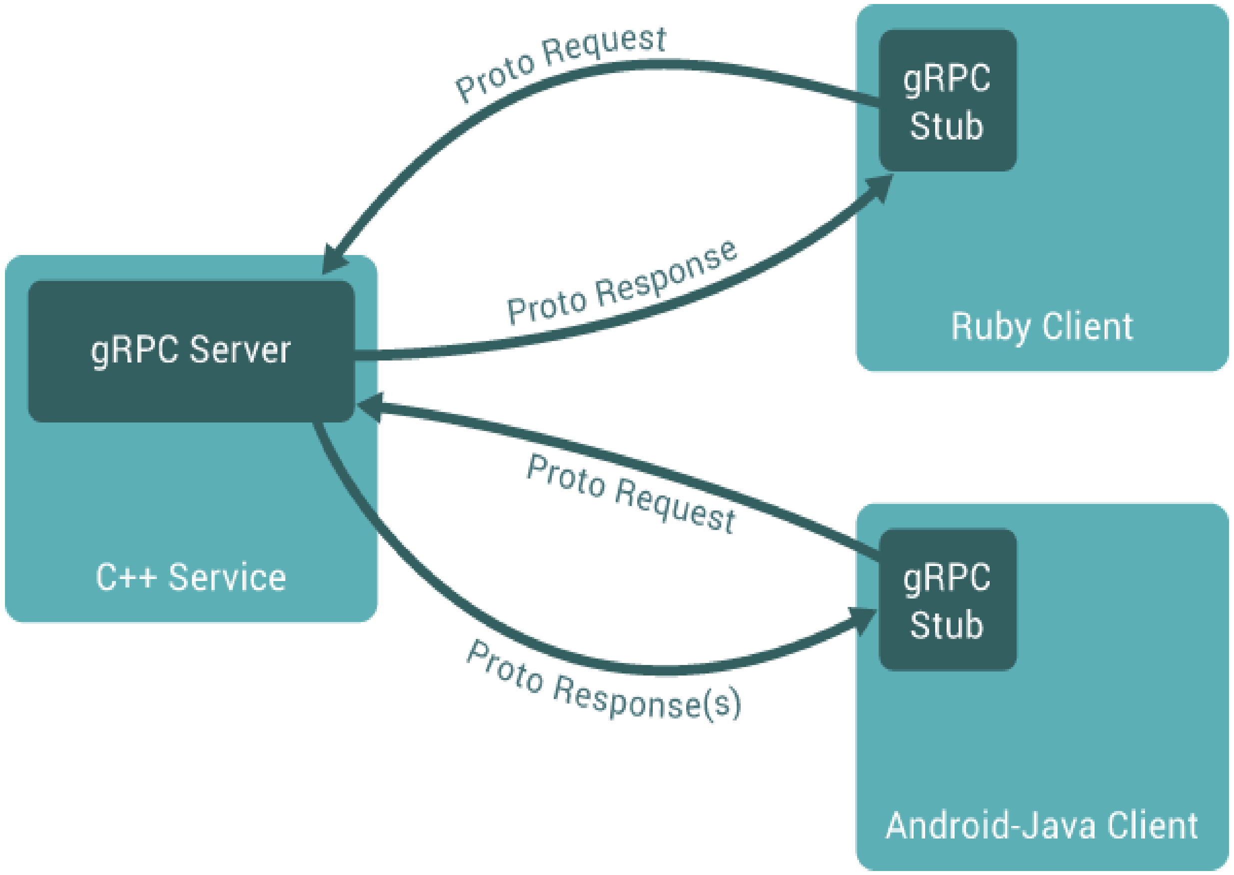


IPC supported by Windows



- Clipboard: very loosely coupled exchange medium, on agreed data format

gRPC (RPC = Remote Procedure Call)



- http://www.grpc.io/
- a high performance, open-source universal RPC framework
- Stubby used by Google as a single general-purpose RPC infrastructure to connect the large number of microservices running within and across multiple data centers.
- based around the idea of defining a service, specifying the methods that can be called remotely with their parameters and return types
- Supports various popular programming languages such as C
- uses protocol buffers as the Interface Definition Language (IDL)

Concurrency

- 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
  - having an integer count that keeps track of the number of full buffers

consumer-producer problem

- One of the solutions to a consumer-producer problem that fills all the buffers
  - having an integer count that keeps track of the number of full buffers
  - Initially, count is set to 0
  - count is incremented by the producer after it produces a new buffer
  - Count is decremented by the consumer after it consumes a buffer

Producer

- Produce an item
- Add the item to the buffer
- If the buffer is full, the producer does not add any item

```
while (count == BUFFER_SIZE)
   ; // do nothing

// add an item to the buffer
++count;
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
```

Consumer

- Remove an item from the buffer
- If the buffer is empty, the consumer does not remove any item

```
while (count == 0)
   ; // do nothing

// remove an item from the buffer
--count;
item = buffer[out];
out = (out + 1) % BUFFER_SIZE;
```

Producer-Consumer

```
while (count == BUFFER_SIZE)
    ; // do nothing

// add an item to the buffer
++count;
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
```

Let count = 5
And BUFFER_SIZE = 5

Producer

Consumer

```
while (count == 0)
    ; // do nothing

// remove an item from the buffer
--count;
item = buffer[out];
out = (out + 1) % BUFFER_SIZE;
```

Producer-Consumer

```
while (count == BUFFER_SIZE)
   ; // do nothing

// add an item to the buffer
++count;
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
```

Producer

Consumer

```
while (count == 0)
   ; // do nothing

// remove an item from the buffer
--count;
item = buffer[out];
out = (out + 1) % BUFFER_SIZE;
```

Let count = 5
And BUFFER_SIZE = 5

Case 1:
- **producer**
  - **loop do nothing**
- **consumer**
  - **remove 1 item**
- **producer**
  - **add 1 item**

Producer-Consumer

```
while (count == BUFFER_SIZE)
   ; // do nothing

// add an item to the buffer
++count;
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
```

Producer

Consumer

```
while (count == 0)
   ; // do nothing

// remove an item from the buffer
--count;
item = buffer[out];
out = (out + 1) % BUFFER_SIZE;
```

Let count = 5
And BUFFER_SIZE = 5

Case 1:
- **producer**
  - **loop do nothing**
- **consumer**
  - **remove 1 item**
- **producer**
  - **add 1 item**

Case 2:
- **consumer**
  - **remove 1 item**
- **producer**
  - **add 1 item**

Both cases, count = 5

Producer-Consumer

```
while (count == BUFFER_SIZE)
   ; // do nothing

// add an item to the buffer
++count;
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
```

Producer

Consumer

```
while (count == 0)
   ; // do nothing

// remove an item from the buffer
--count;
item = buffer[out];
out = (out + 1) % BUFFER_SIZE;
```

Let count = 5
And BUFFER_SIZE = 5

Case 1:
- **producer**
  - **loop do nothing**
- **consumer**
  - **remove 1 item**
- **producer**
  - **add 1 item**

Case 2:
- **consumer**
  - **remove 1 item**
- **producer**
  - **add 1 item**

Both cases, count = 5

Race Condition

- count++ could be implemented as

  ```
  register1 = count
  register1 = register1 + 1
  count = register1
  ```

- count-- could be implemented as

  ```
  register2 = count
  register2 = register2 - 1
  count = register2
  ```

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

  S0: producer executes register1 = count
  S1: producer executes register1 = register1 + 1
  S2: consumer executes register2 = count
  S3: consumer executes register2 = register2 - 1
  S4: producer executes count = register1
  S5: consumer executes count = register2

Race Condition (cont.)

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

S0: producer executes register1 = count {reg1 = 5}
S1: producer executes register1 = register1 + 1 {reg1 = 6}
S2: consumer executes register2 = count {reg2 = 5}
S3: consumer executes register2 = register2 - 1 {reg2 = 4}
S4: producer executes count = register1 {count = 6 }
S5: consumer executes count = register2 {count = 4}

Critical Section

Section of code where shared data is accessed which is a segment of code that the process may be changing common variables, updating a table, writing a file, and so on

- Race Condition - When there is concurrent access to shared data and the final outcome depends upon order of execution
- Solution:
  - Only one process at a time can execute in its critical section

Solution to a Critical-Section Problem

```
while (true) {

        entry section

            critical section

        exit section

            remainder section

}
```

- Entry Section - Code that requests permission to enter its critical section.
- Critical Section
- Exit Section - Code that is run after exiting the critical section
- Remainder Section

Solution to a Critical-Section Problem

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

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

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 using Locks

```
while (true) {

    acquire lock

        critical section

    release lock

        remainder section

}
```

Semaphore

- a synchronization tool
- Semaphore *S* – integer variable
- Two standard operations modify
  - S: acquire()
  - R: release()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

```
acquire() {
    while value <= 0
        ; // no-op
    value--;
}

release() {
    value++;
}
```

Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks

```
Semaphore S = new Semaphore();

S.acquire();

  // critical section

S.release();

  // remainder section
```

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.

Implementation with no Busy waiting (Cont.)

- Implementation of **acquire()**:

```
acquire(){
    value--;
    if (value < 0) {
        add this process to list
        block;
    }
}
```

- Implementation of **release()**:

```
release(){
    value++;
    if (value <= 0) {
        remove a process P from list
        wakeup(P);
    }
}
```

Problems with Semaphores

- Incorrectly use can cause timing errors
  - release() …. acquire()
  - release() … release()
  - acquire() … acquire()
  - acquire()   or release() alone
- Errors are difficult to detect

  Error only happens if some particular execution sequences take place and these sequences do not always occur
- Correct use of semaphore operations:
  - acquire() …. release()

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spin locks

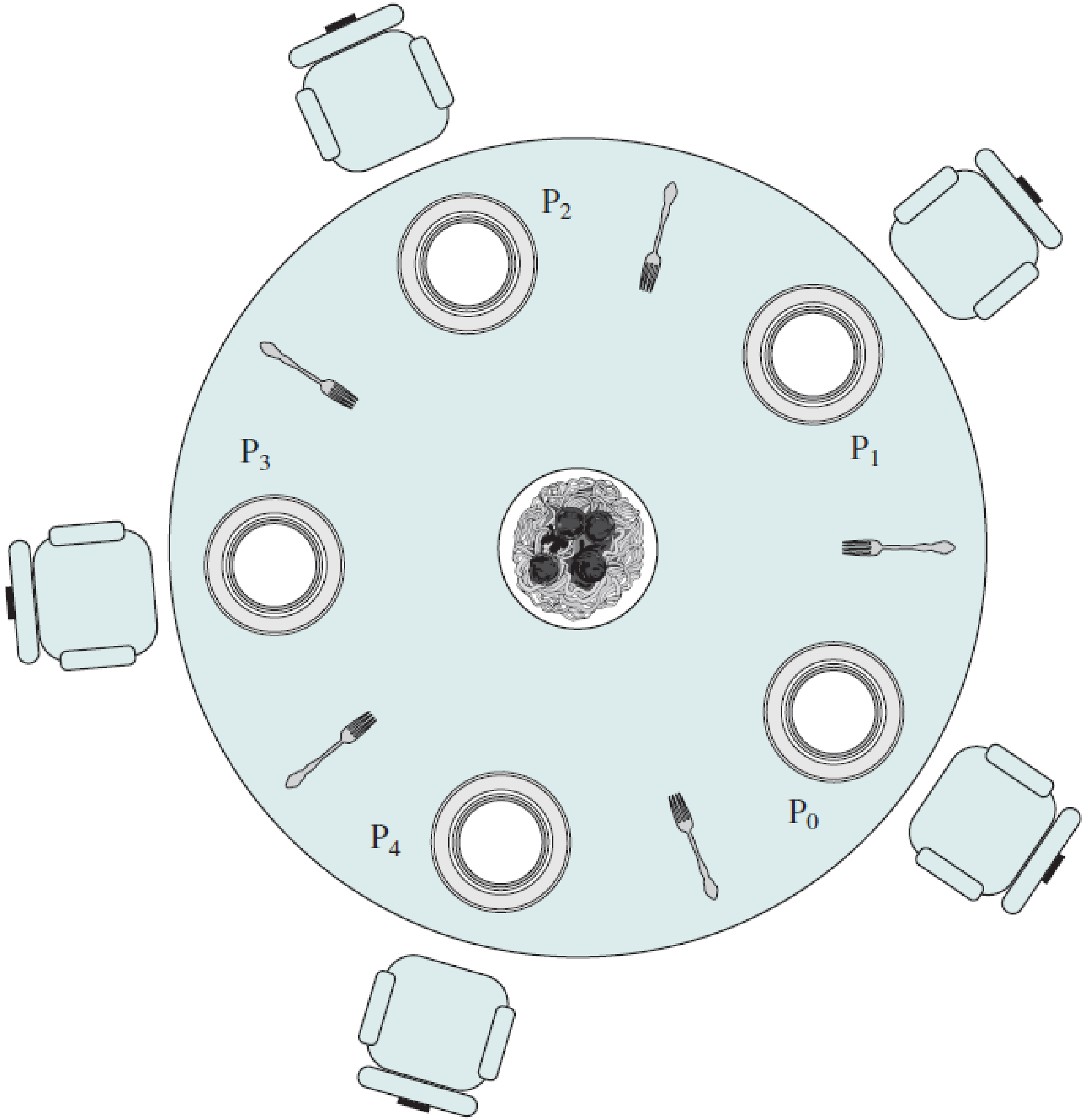
Atomic Transactions

- Assures that operations happen as a single logical unit of work, in its entirety, or not at all
- Related to field of database systems
- Challenge is assuring atomicity despite computer system failures
- Transaction - collection of instructions or operations that performs single logical function
  - Transaction is series of read and write operations
  - Terminated by commit (transaction successful) or abort (transaction failed) operation
  - Aborted transaction must be rolled back to undo any changes it performed

Deadlock and Starvation

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

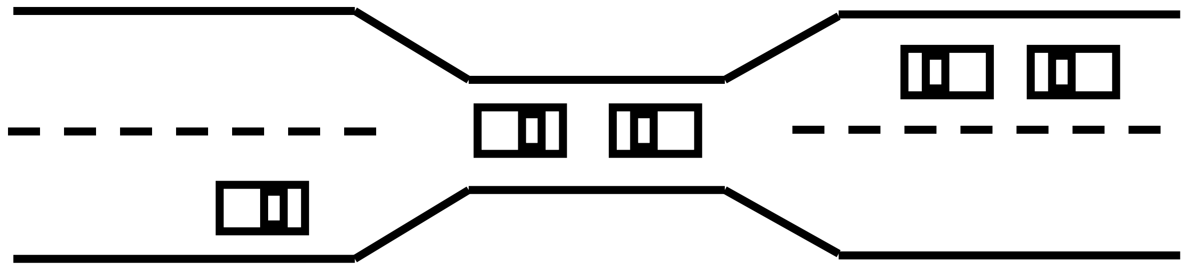
Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.



P_2
P_1
P_3
P_0
P_4

The Deadlock Problem

A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.



- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible

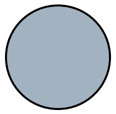
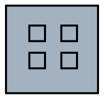
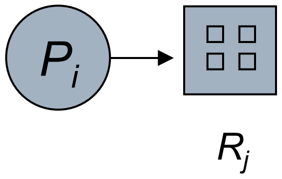
Deadlock Characterization

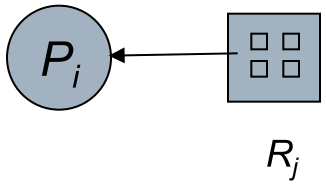
Deadlock can arise if four conditions hold simultaneously

- **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**: A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain (e.g., P0 → P1 → P2 → …. → Pn → P0)

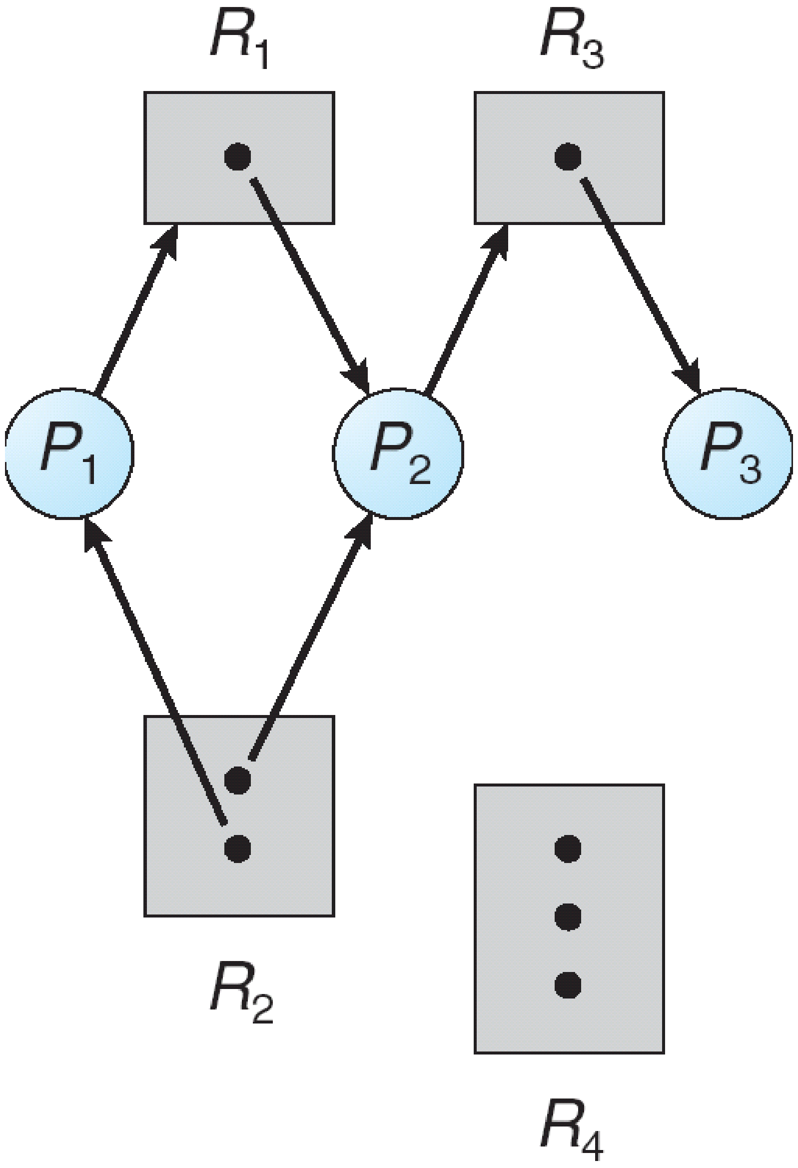
Resource-Allocation Graph

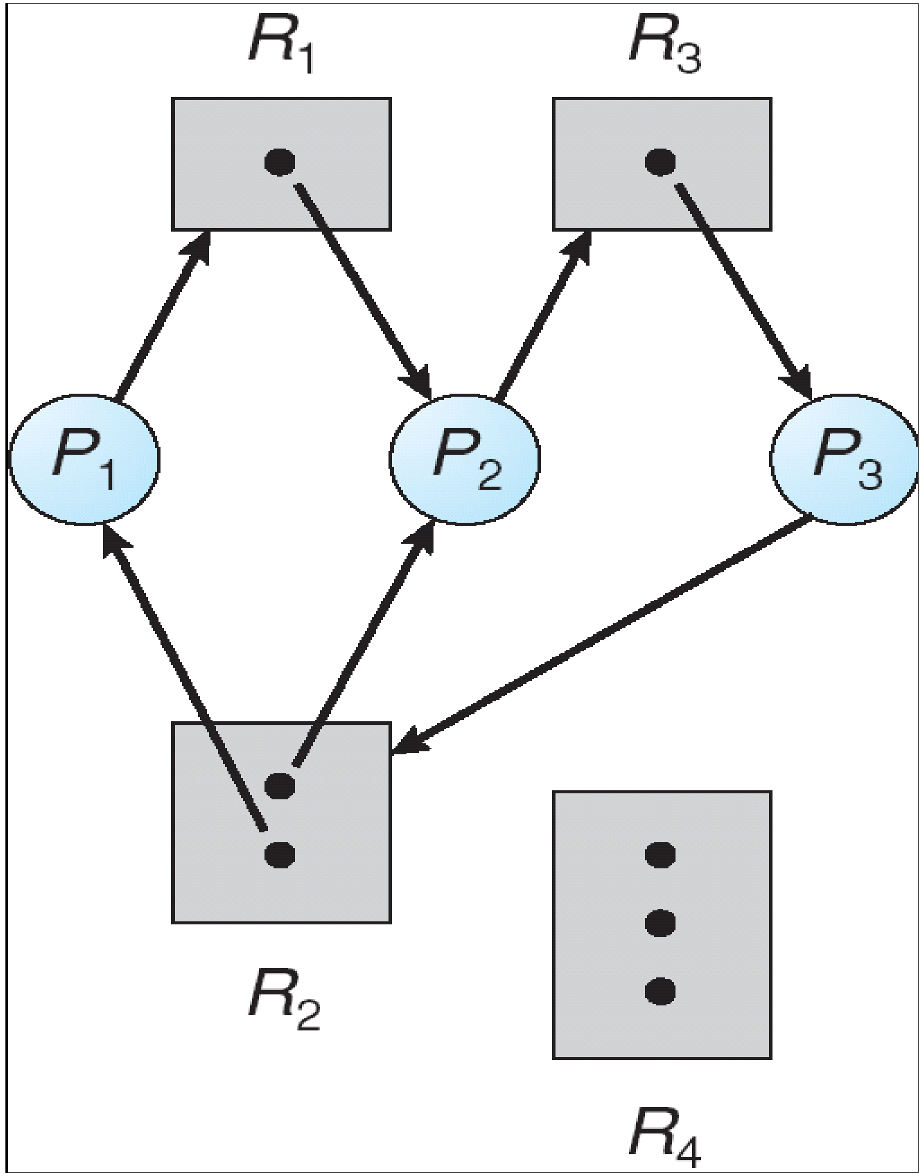
A set of vertices *V* and a set of edges *E*.

- Process 
- Resource Type with 4 instances 
- P_i requests instance of R_j 

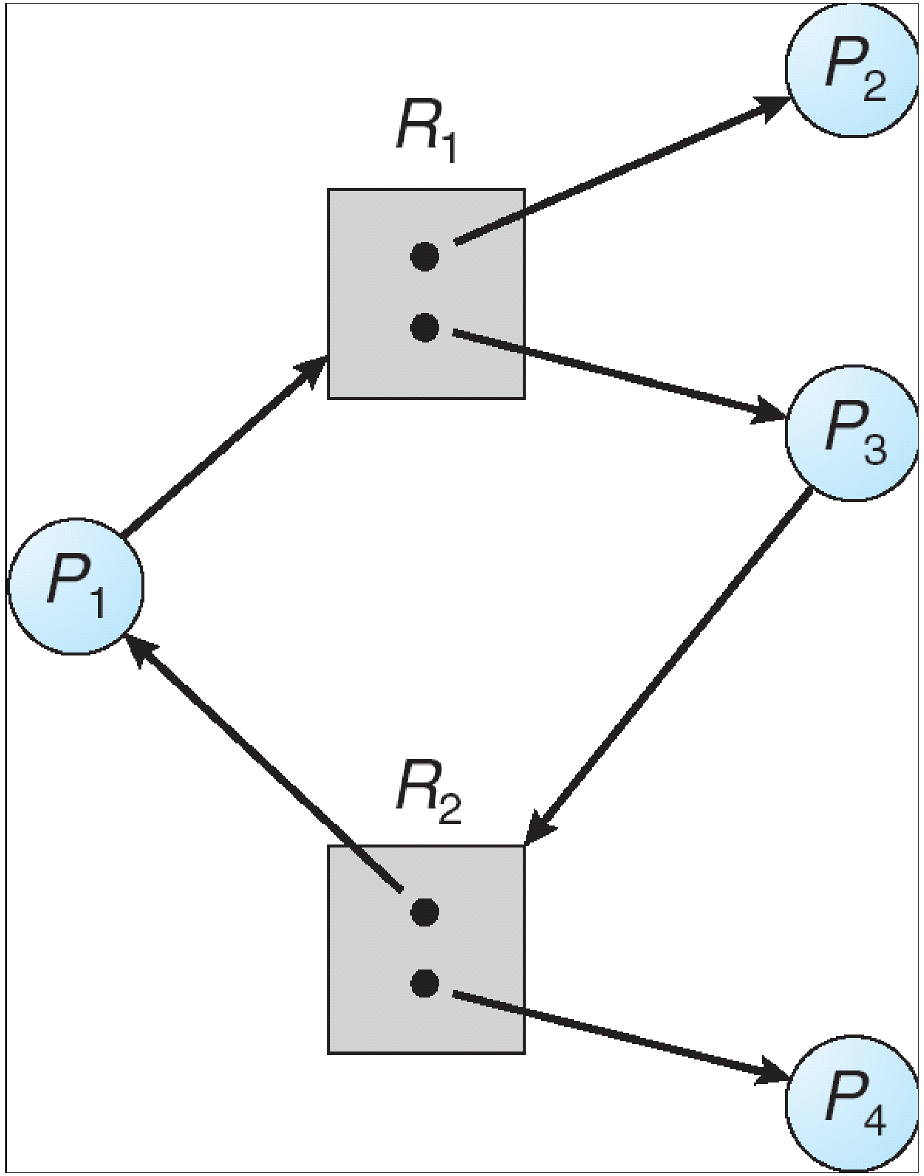
- P_i is holding an instance of R_j 

Resource Allocation Graph





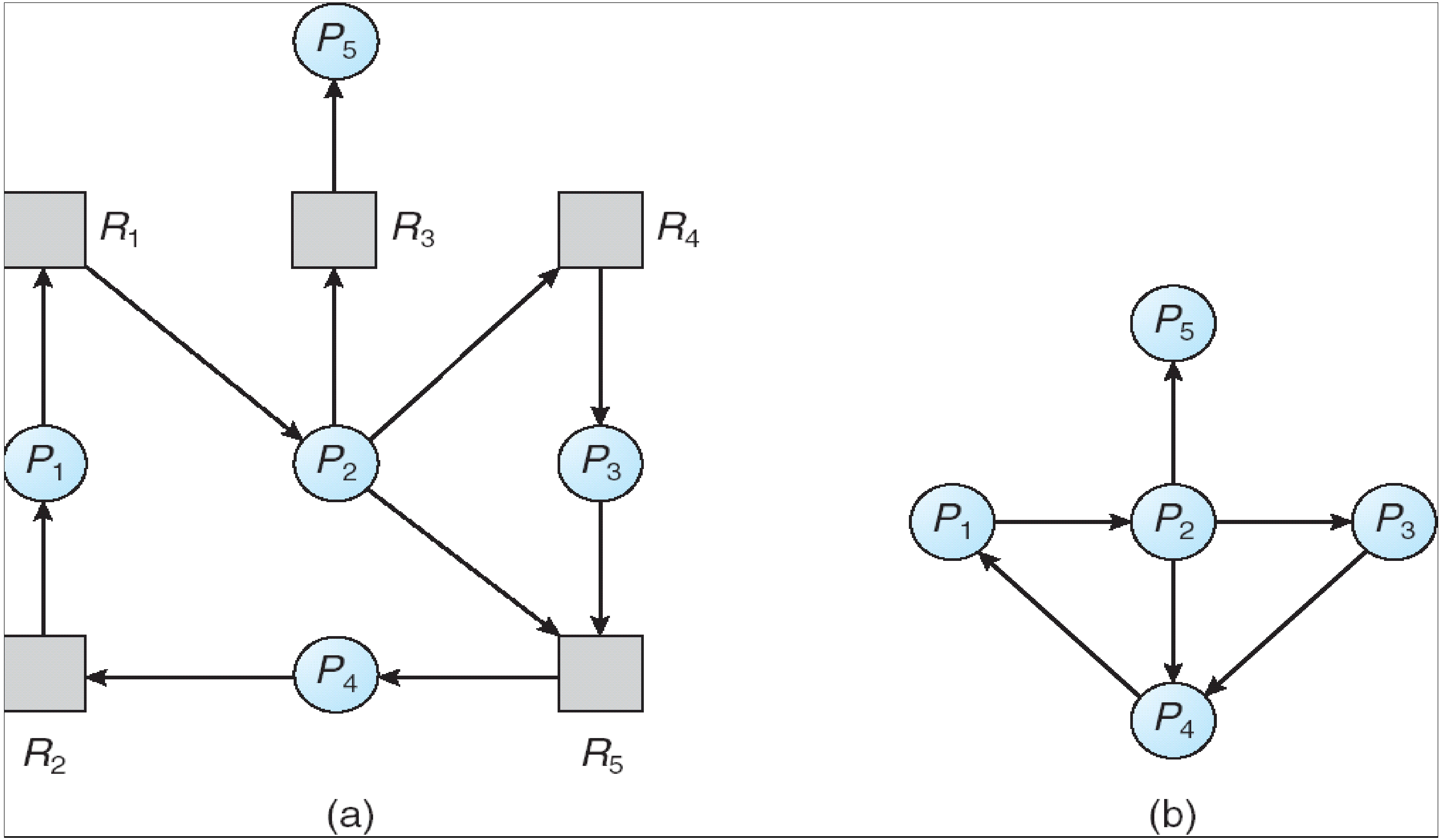
Resource Allocation Graph (cont.)



Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph   Corresponding wait-for graph

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?
  - Priority of the process.
  - How long process has computed, and how much longer to completion.
  - Resources the process has used.
  - Resources process needs to complete.
  - How many processes will need to be terminated.
  - 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.

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem

Bounded-Buffer Problem

- A.k.a consumer-producer problem
  - Two processes (the producer and the consumer) who share a common buffer.
    - The producer's job is to generate a piece of data, put it into the buffer and start again.
    - The consumer's job is to consume the data (i.e., remove it from the buffer) one piece at a time.
  - The problem
    - The producer won't try to add data into the buffer if it's full
    - The consumer won't try to remove from an empty buffer

Bounded-Buffer Solution

- Solution
  - *N* buffers, each can hold one item
  - Semaphore mutex initialized to the value 1
  - Semaphore full initialized to the value 0
  - Semaphore empty initialized to the value N.

Bounded-Buffer Solution

- Solution
  - *N* buffers, each can hold one item
  - Semaphore mutex initialized to the value 1
  - Semaphore full initialized to the value 0
  - Semaphore empty initialized to the value N.

```
/* producer */
while (true) {
  empty.acquire();
  mutex.acquire();

  // add an item

  mutex.release();
  full.release();
}
```

```
/* consumer */
while (true) {
  full.acquire();
  mutex.acquire();

  // remove an item

  mutex.release();
  empty.release();
}
```

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 update
  - 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.

Readers-Writers Solution

- Semaphore mutex initialized to 1.
- Semaphore db initialized to 1.
- Integer readerCount initialized to 0.

Readers-Writers Solution

- Semaphore mutex initialized to 1.
- Semaphore db initialized to 1.
- Integer readerCount initialized to 0.

```
/* reader  */
while (true) {

   db.acquire();

   // read from the database

   db.release();

}
```

Readers-Writers Solution

- Semaphore mutex initialized to 1.
- Semaphore db initialized to 1.
- Integer readerCount initialized to 0.

```
/* reader  */
while (true) {

   ++readerCount;
    db.acquire();

    // read from the database


    --readerCount;
    db.release();

}
```

Readers-Writers Solution

- Semaphore mutex initialized to 1.
- Semaphore db initialized to 1.
- Integer readerCount initialized to 0.

```
/* reader  */
while (true) {
  mutex.acquire();
  ++readerCount;
  if(readerCount == 1) { db.acquire(); }  // first reader
  mutex.release();

   // read from the database

   mutex.acquire();
  --readerCount;
  if(readerCount == 0) { db.release(); }  // last reader
  mutex.release();
}
```

Readers-Writers Solution

```
/* writer  */
while (true) {
  db.acquire();
  // update the database
  db.release();
}

/* reader  */
while (true) {
  mutex.acquire();
  ++readerCount;
  if(readerCount == 1) { db.acquire(); }  // first reader
  mutex.release();
   // read from the database
   mutex.acquire();
  --readerCount;
  if(readerCount == 0) { db.release(); }  // last reader
  mutex.release();
}
```

```
acquire() {
    while value <= 0
        ; // no-op
    value--;
}

release() {
    value++;
}
```