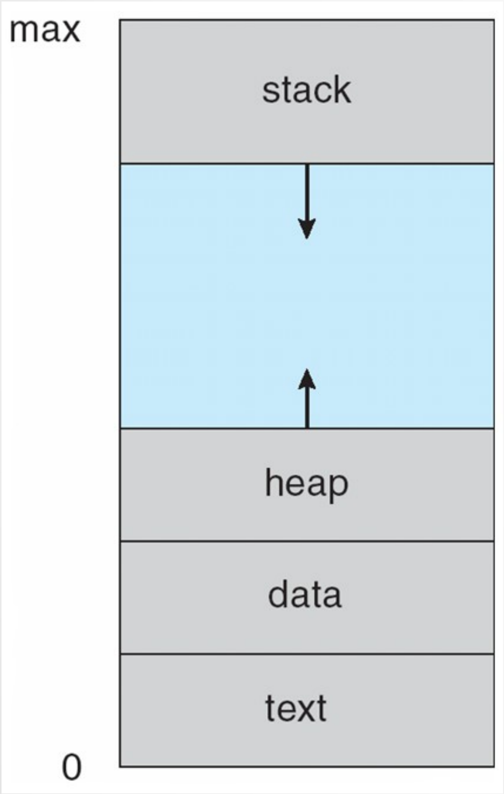
CST 347 – Real Time OS

Lecture 06 – Interprocess Communication

Troy Scevers

Introduction

- Recall what defines a Process

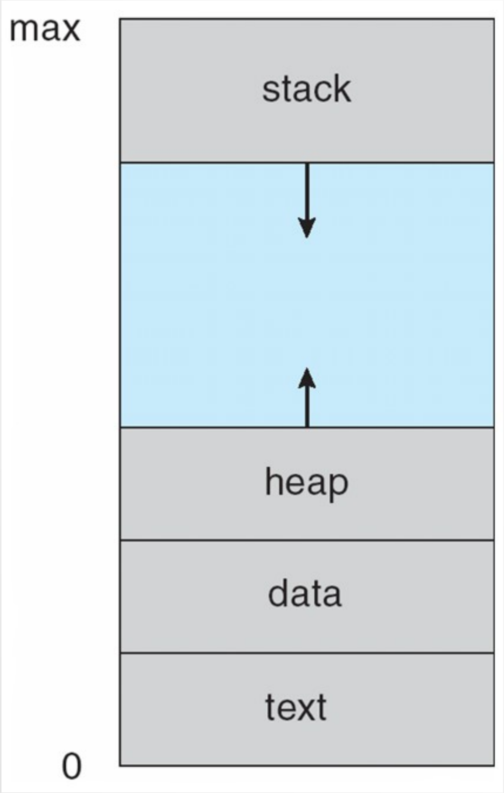


Physical

Memory

Introduction

- Recall what defines a Process

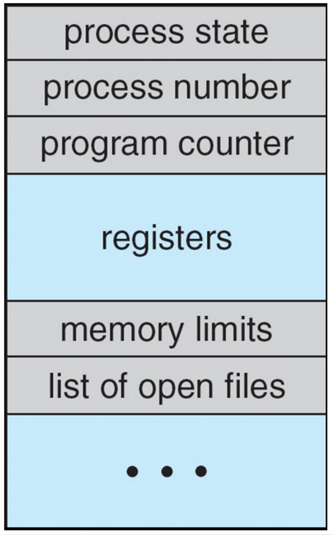


Physical

Memory

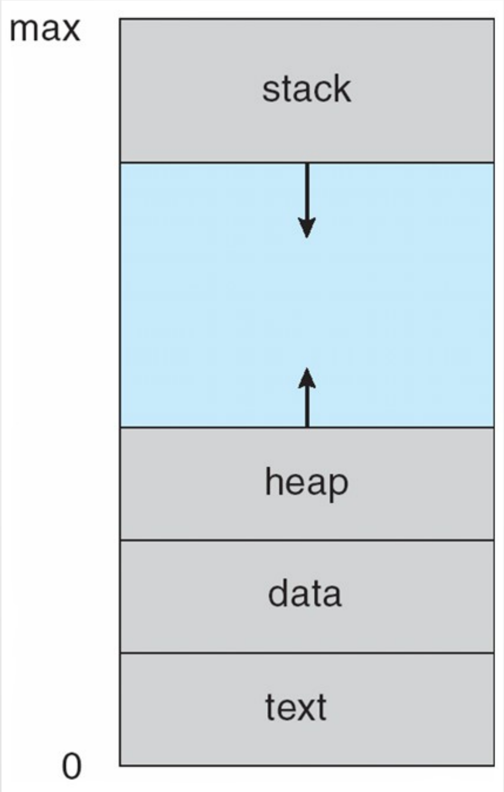
Logical

Definition



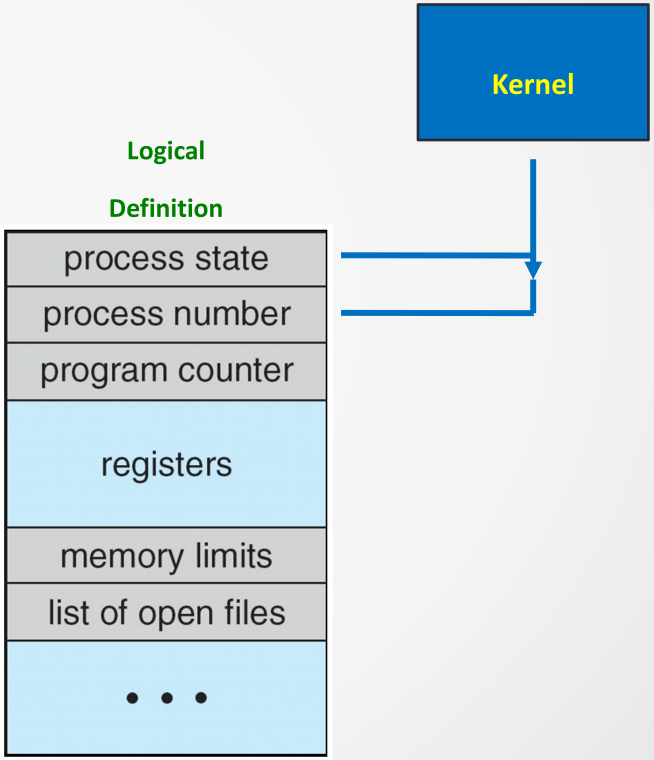
Introduction

- Recall what defines a Process



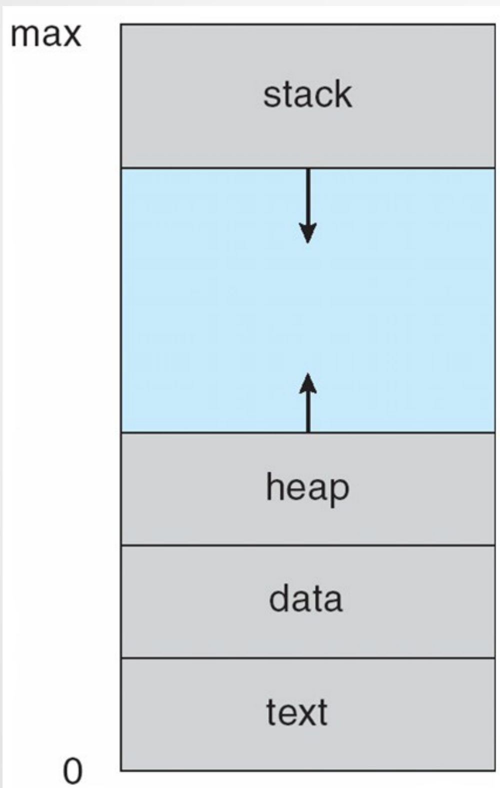
Physical

Memory



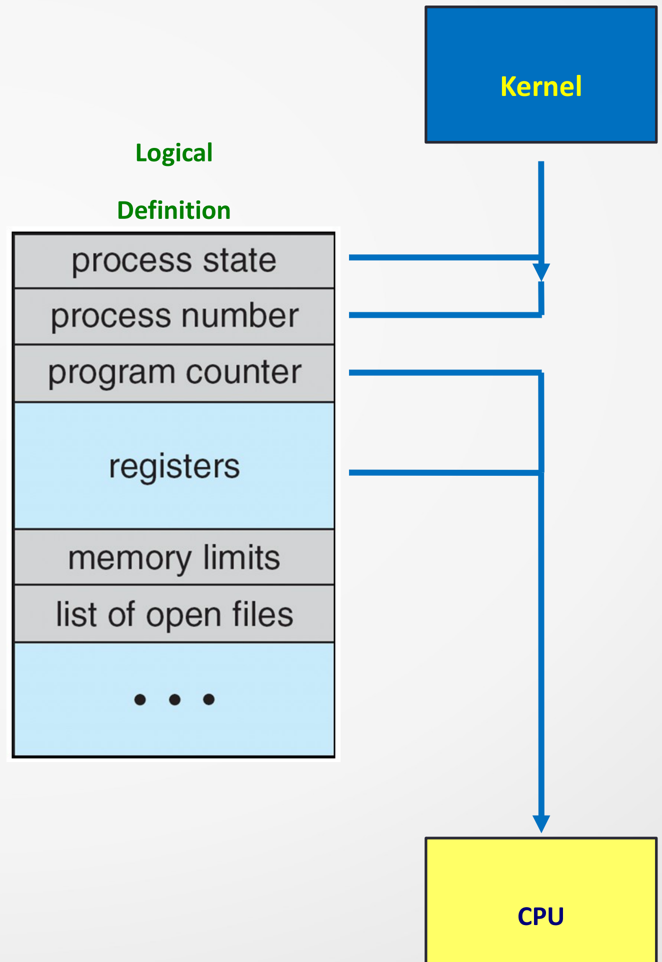
Introduction

- Recall what defines a Process



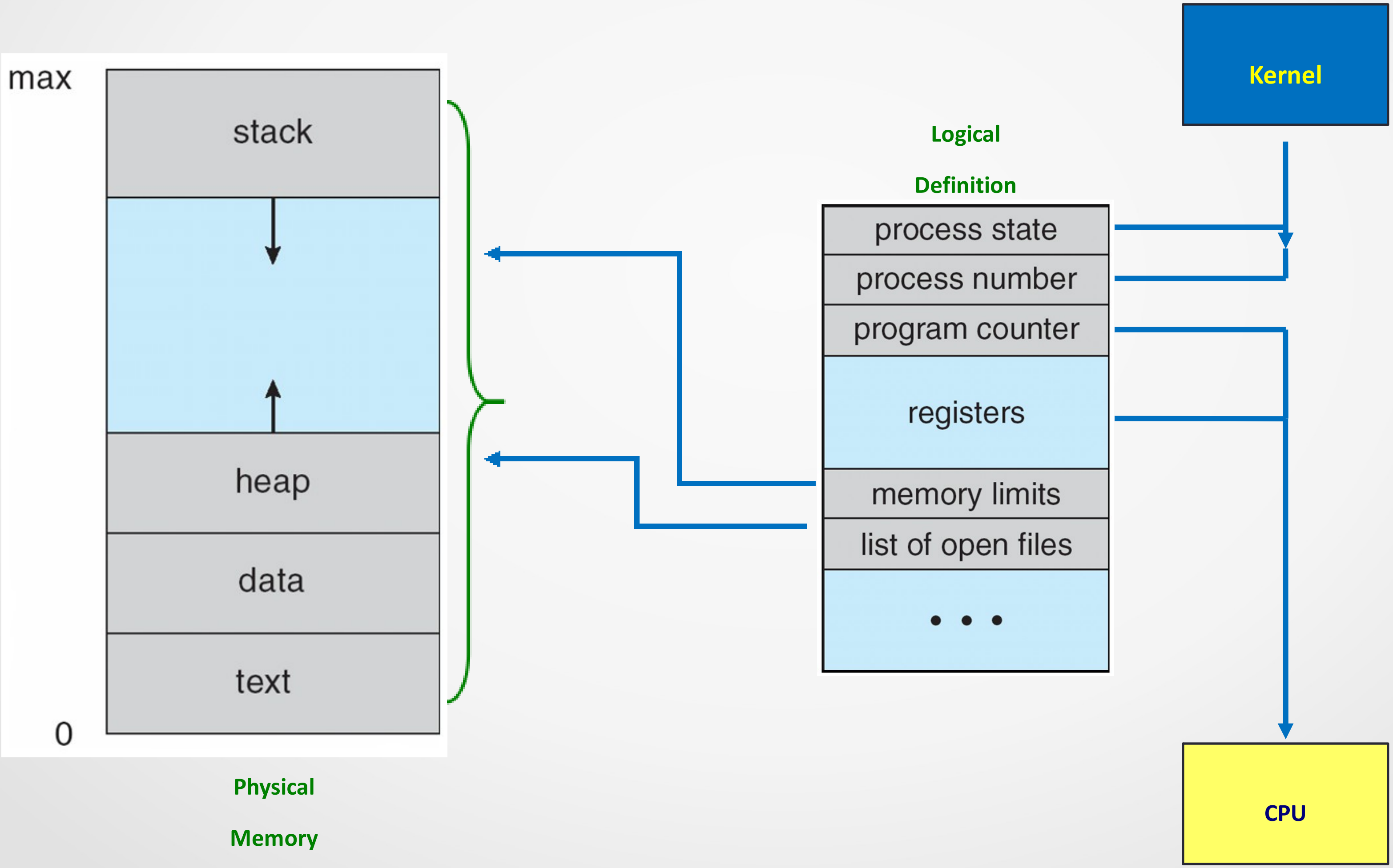
Physical

Memory



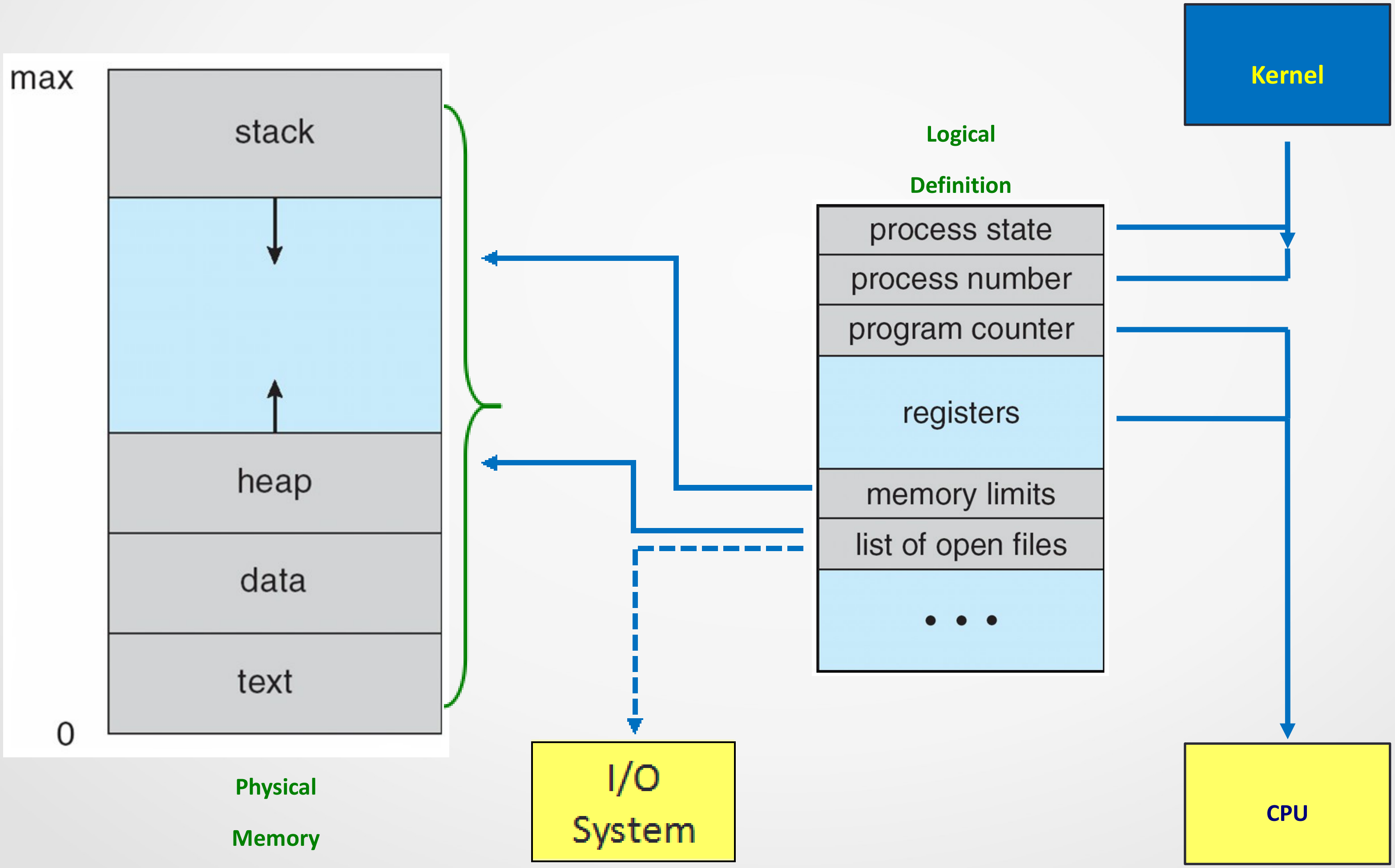
Introduction

- Recall what defines a Process



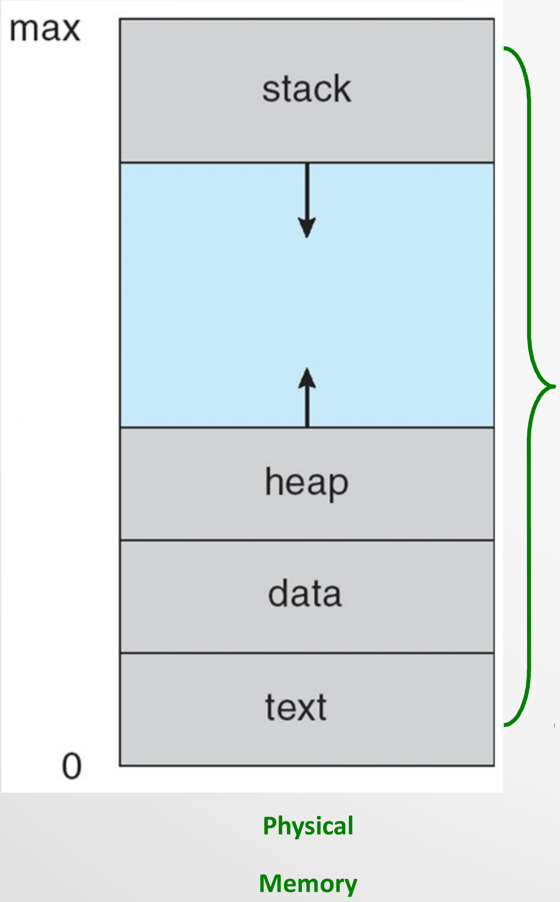
Introduction

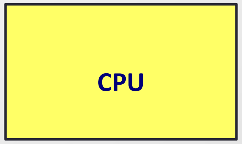
- Recall what defines a Process



Introduction

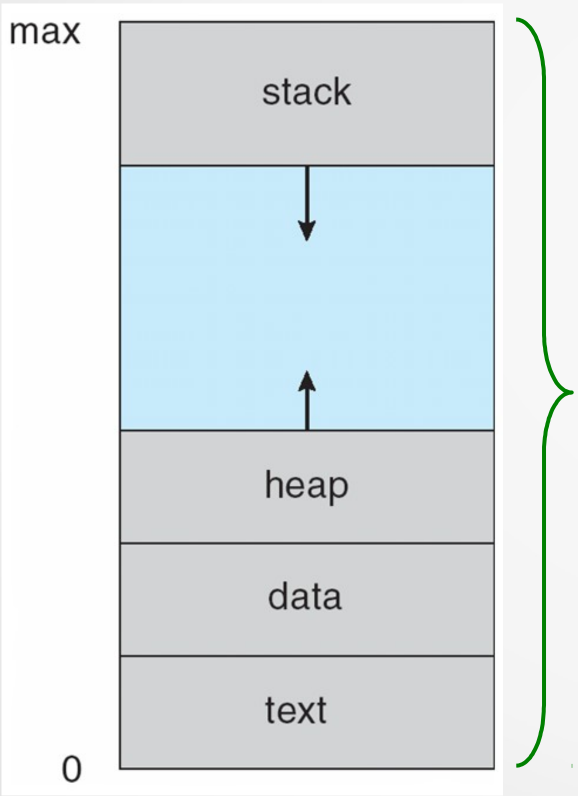
- Recall what defines a Process





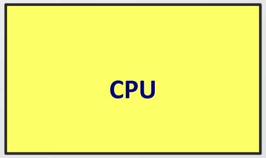
Multiple Processes

- Two or more Processes



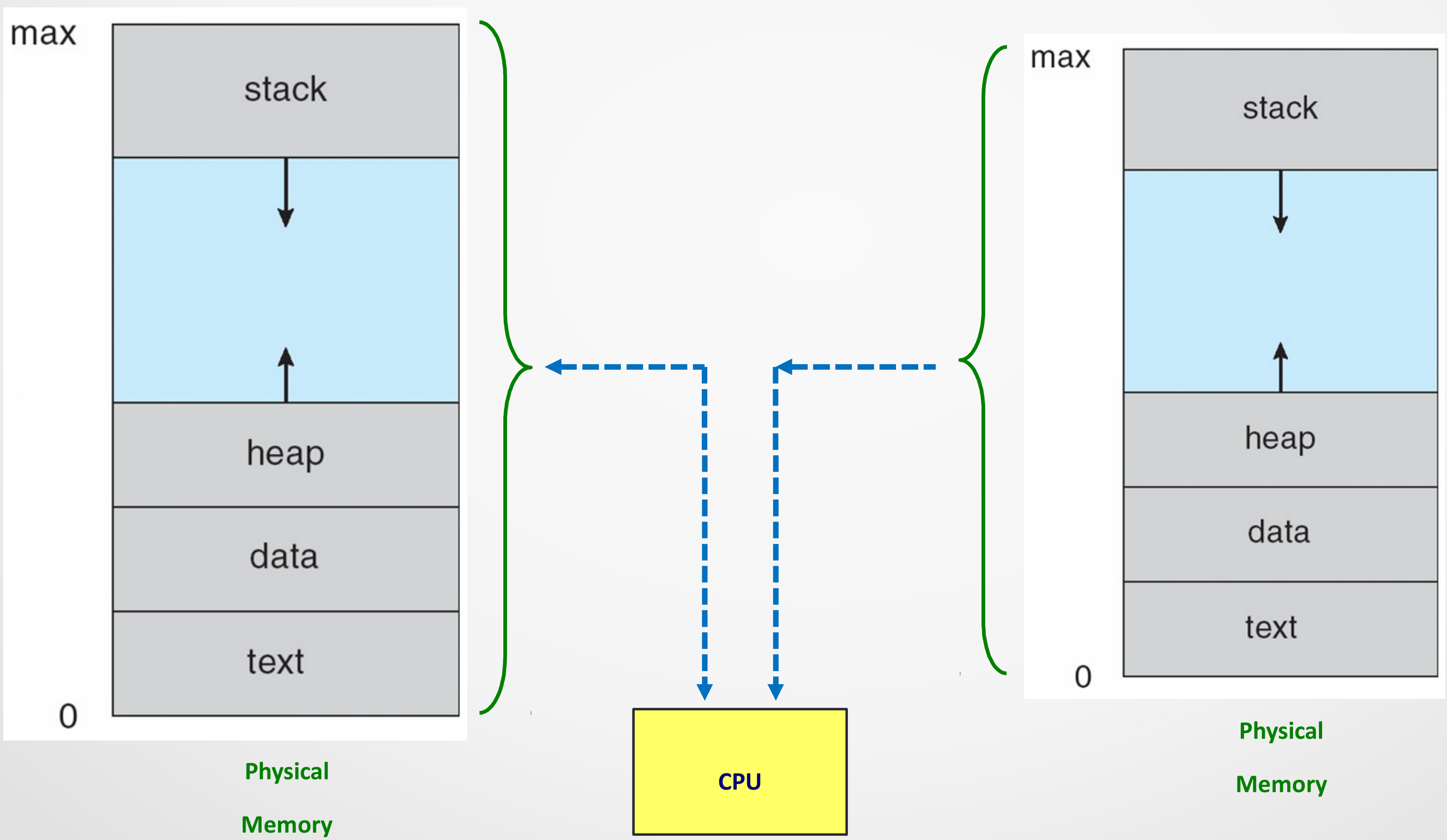
Physical

Memory



Multiple Processes

- Two or more Processes → *Time sharing*



Multiple Processes

- What if processes need to interact?

Multiple Processes

- What if processes need to interact?
- Why would processes need to interact?
  - Independent – Not necessary
  - Cooperating

Multiple Processes

- Reasons for cooperation
  - Information sharing (e.g. Producer/Consumer)
  - Computation speedup (multiprocessor)
  - Modularity (function to task/thread)
  - Convenience (e.g. User productivity)

Multiple Processes

- Embedded car *intelligence*
  - Information sharing (Suggestions???)
  - Computation speedup (Suggestions???)
  - Modularity (Suggestions???)
  - Convenience (Suggestions???)

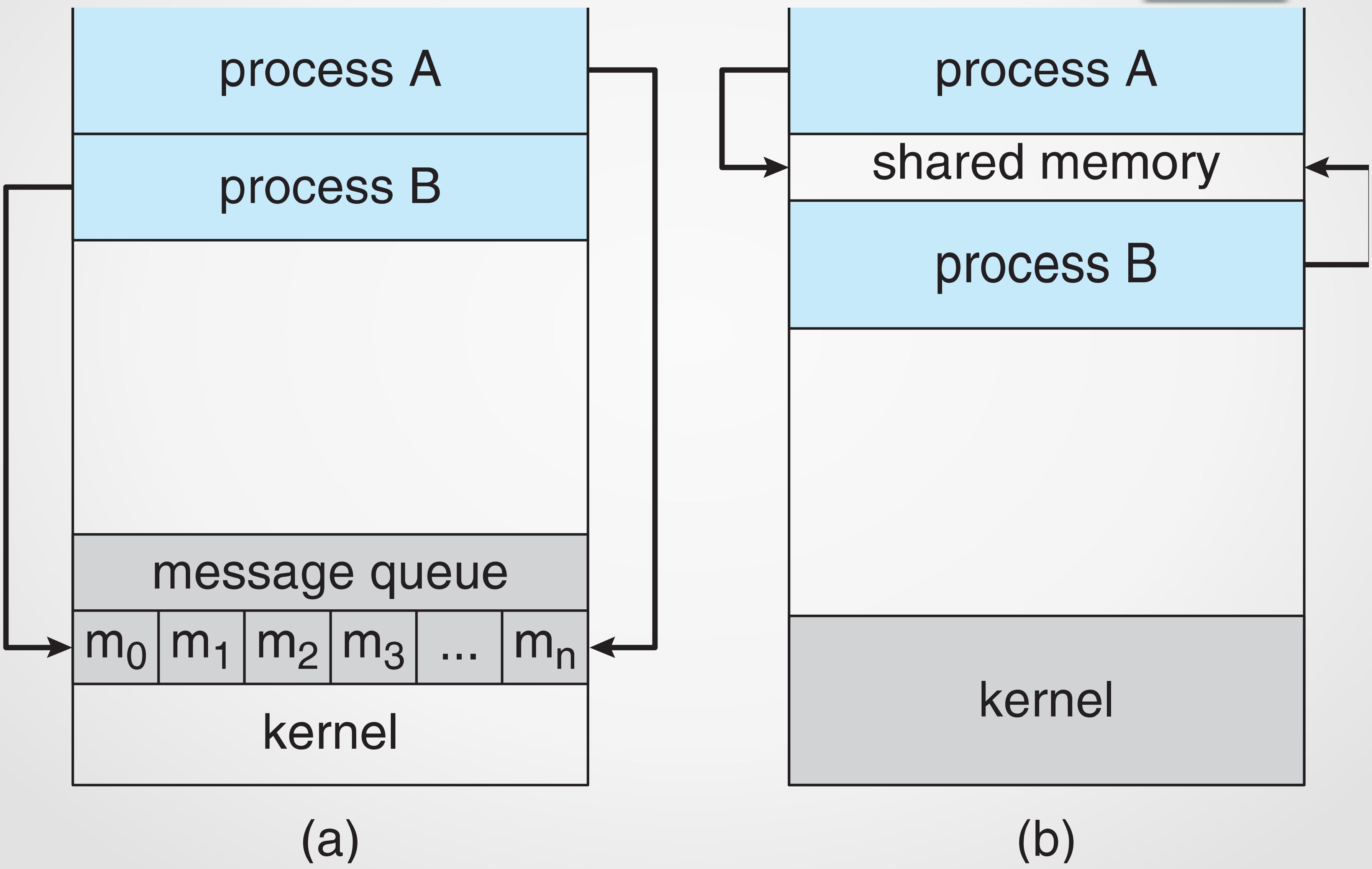
Multiple Processes

- Embedded car *intelligence*
  - Information sharing (Cruise/Engine Sync)
  - Computation speedup (Brakes/Fuel)
  - Modularity (Locks/Lights)
  - Convenience (Navigation/Audio)

Multiple Cooperating Processes

- Mechanism/s to exchange data/information
  - Interprocess Communication (IPC)
- Two IPC models
  - Shared-memory systems
  - Message-Passing systems
- Remember, only Process should have access to its memory
  - Process willing gives up independence
  - Need synchronization tools to handle simultaneous access

Communications Models



Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - unbounded-buffer places no practical limit on the size of the buffer
  - bounded-buffer assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct ITEM {
. . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements

Bounded-Buffer – Producer

```
item next_produced;
while (true) {
   /* produce an item in next_produced */
   while (((in + 1) % BUFFER_SIZE) == out);
      /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER SIZE;
}
```

Bounded Buffer – Consumer

```
item next_consumed;
while (true) {
      while (in == out)  ; /* do nothing */
      next_consumed = buffer[out];
      out = (out + 1) % BUFFER SIZE;

      /* consume the item in next consumed */
}
```

Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a **communication link** between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., direct or indirect, synchronous or asynchronous, automatic or explicit buffering)

Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

Direct Communication

- Processes must name each other explicitly:
  - **`send`** (*P, message*) – send a message to process P
  - **`receive`**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Indirect Communication

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:

  `send`(*A, message*) – send a message to mailbox A

  `receive`(*A, message*) – receive a message from mailbox A

Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null

Synchronization (Cont.)

- Different combinations possible
  - If both send and receive are blocking, we have a rendezvous
- Producer-consumer becomes trivial

```
message next_produced;
while (true) {
     /* produce an item in next_produced */
    send(next_produced);
}
message next_consumed;
while (true) {
    receive(next_consumed);

    /* consume the item in next_consumed */
}
```

Buffering

- Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity – 0 messages
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of n messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

Examples of IPC Systems - POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment
  - shm_fd = shm_open(name, O CREAT | O RDRW, 0666);
  - Also used to open an existing segment to share it
  - Set the size of the object

```
ftruncate(shm fd, 4096);
```

  - Now the process could write to the shared memory

```
sprintf(shared memory, "Writing to shared memory");
```

IPC POSIX Producer

```
#include <stdio.h>
#include <stlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* create the shared memory object */
   shm_fd = shm_open(name, O_CREAT | O_RDRW, 0666);

   /* configure the size of the shared memory object */
   ftruncate(shm_fd, SIZE);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

   /* write to the shared memory object */
   sprintf(ptr,"%s",message_0);
   ptr += strlen(message_0);
   sprintf(ptr,"%s",message_1);
   ptr += strlen(message_1);

   return 0;
}
```

IPC POSIX Consumer

```
#include <stdio.h>
#include <stlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* open the shared memory object */
   shm_fd = shm_open(name, O_RDONLY, 0666);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

   /* read from the shared memory object */
   printf("%s",(char *)ptr);

   /* remove the shared memory object */
   shm_unlink(name);

   return 0;
}
```

Examples of IPC Systems - Mach

- Mach communication is message based
  - Even system calls are messages
  - Each task gets two mailboxes at creation- Kernel and Notify
  - Only three system calls needed for message transfer

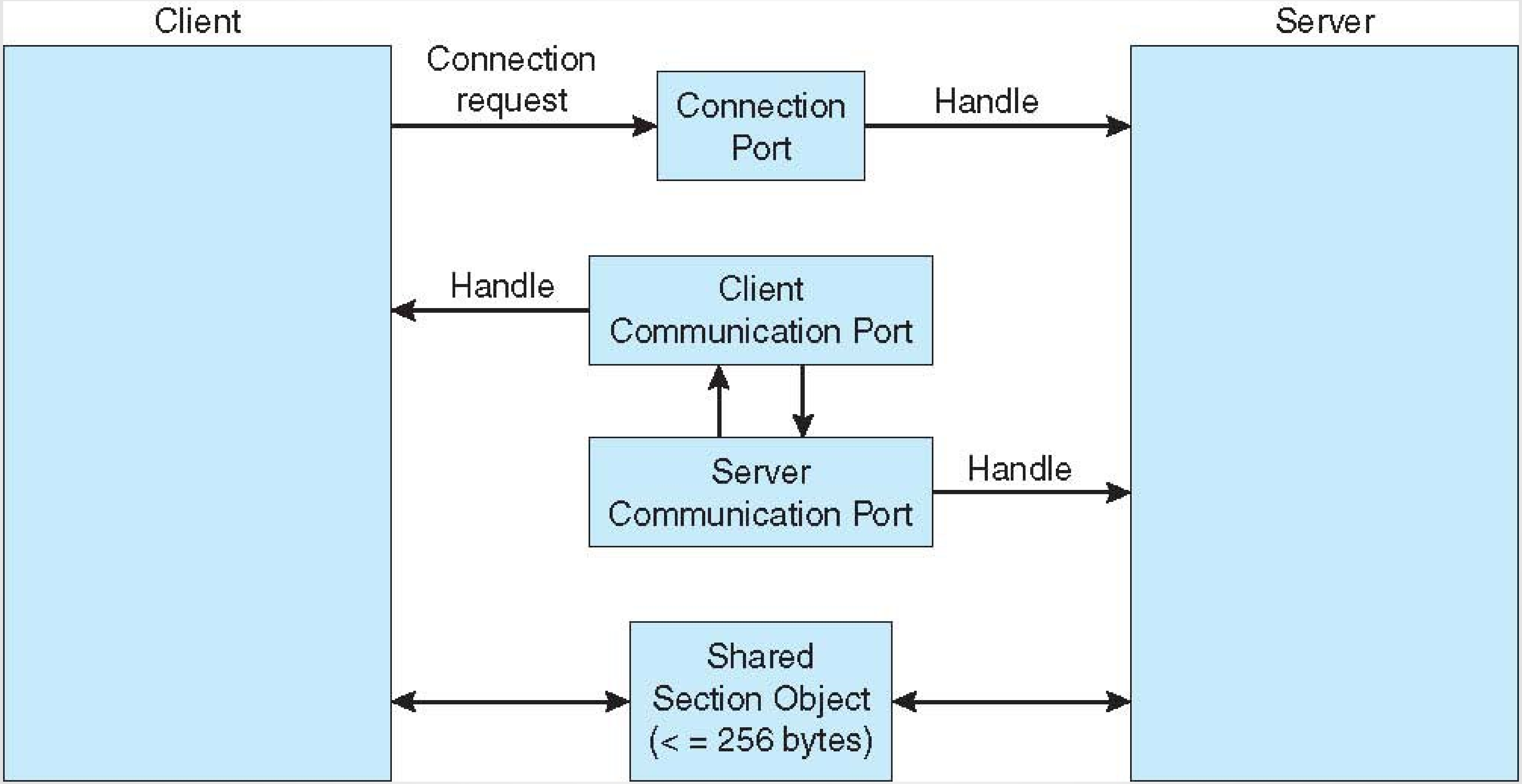
    `msg_send()`, `msg_receive()`, `msg_rpc()`
  - Mailboxes needed for communication, created via

    `port_allocate()`
  - Send and receive are flexible, for example four options if mailbox full:
    - Wait indefinitely
    - Wait at most n milliseconds
    - Return immediately
    - Temporarily cache a message

Examples of IPC Systems – Windows

- Message-passing centric via **advanced local procedure call** (**LPC**) facility
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels
  - Communication works as follows:
    - The client opens a handle to the subsystem's **connection port** object.
    - The client sends a connection request.
    - The server creates two private **communication ports** and returns the handle to one of them to the client.
    - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

Local Procedure Calls in Windows XP



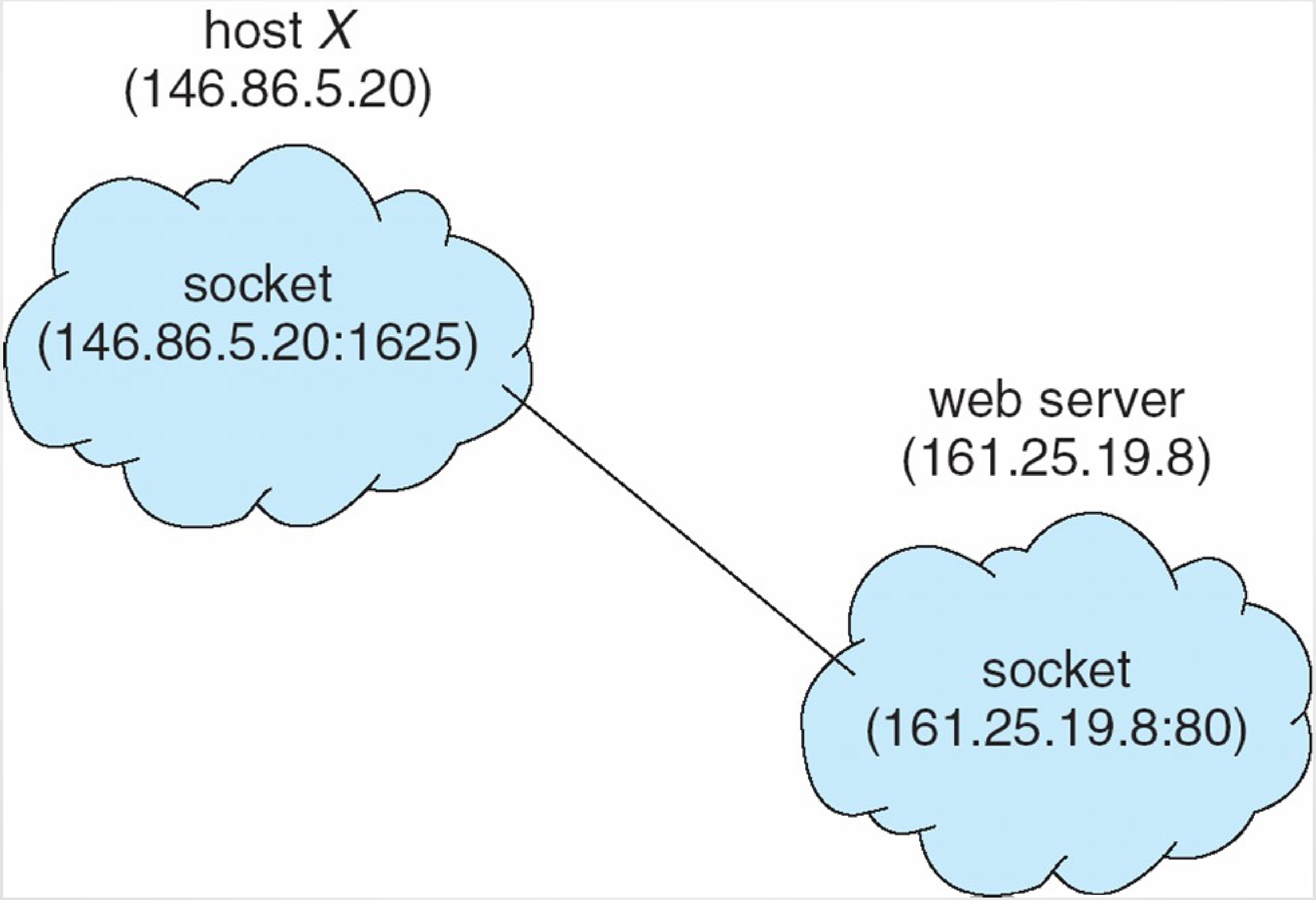
Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Pipes
- Remote Method Invocation (Java)

Sockets

- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All ports below 1024 are ***well known***, used for standard services
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running

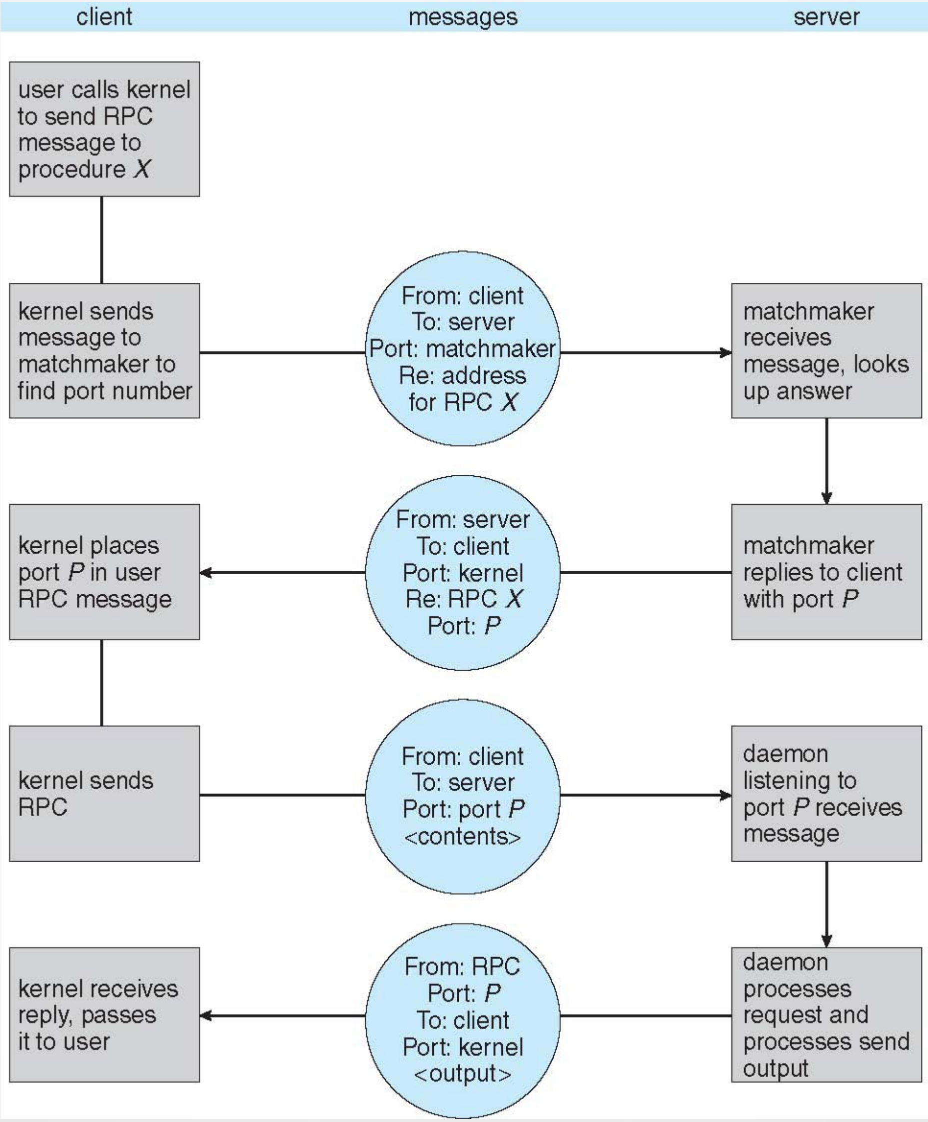
Socket Communication



Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and **marshalls** the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- On Windows, stub code compile from specification written in **Microsoft Interface Definition Language** (**MIDL**)
- Data representation handled via **External Data Representation** (**XDL**) format to account for different architectures
  - **Big-endian** and **little-endian**
- Remote communication has more failure scenarios than local
  - Messages can be delivered ***exactly once*** rather than ***at most once***
- OS typically provides a rendezvous (or **matchmaker**) service to connect client and server

Execution of RPC

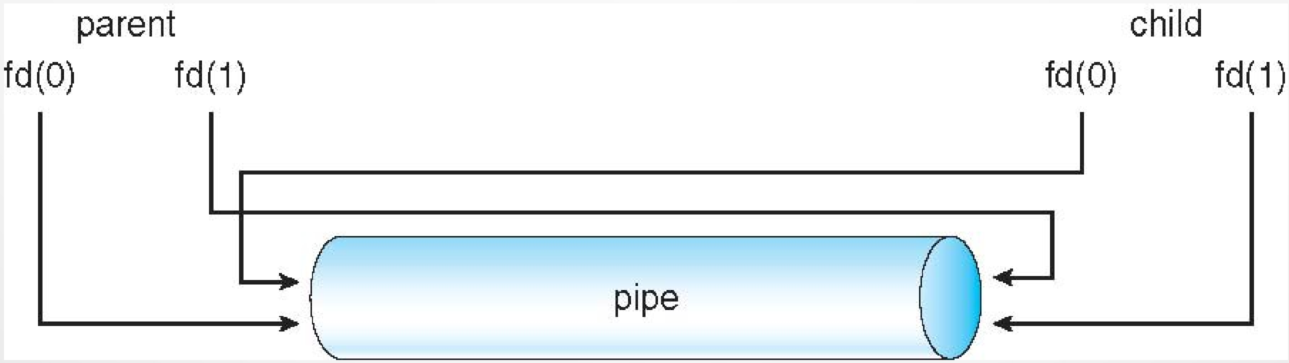


Pipes

- Acts as a conduit allowing two processes to communicate
- **Issues**
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e. ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?

Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes



- Windows calls these **anonymous pipes**
- See Unix and Windows code samples in textbook

Named Pipes

- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems

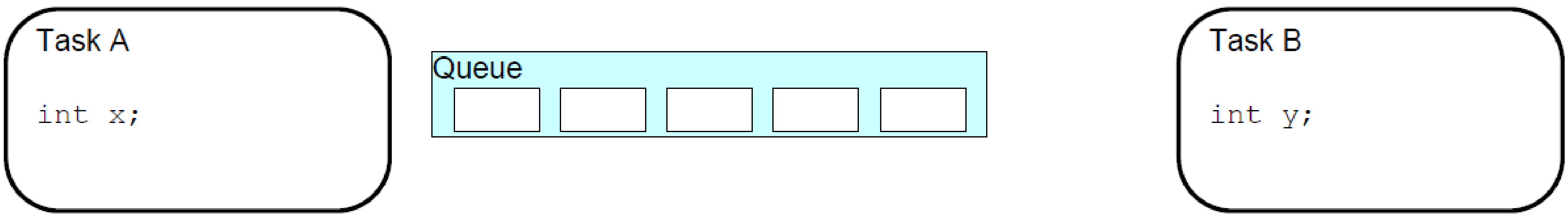
FreeRTOS IPC

- ***xQueueCreate***()
- **xQueueSendToBack**()
- **xQueueSendToFront**()
- ***xQueueReceive***()
- ***uxQueueMessagesWaiting***()

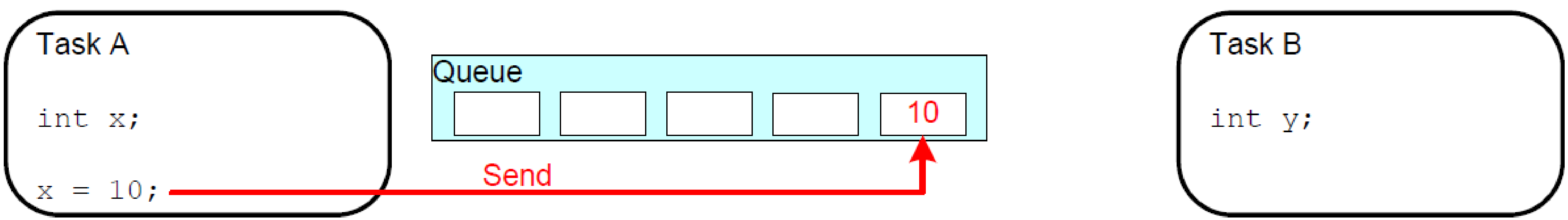
FreeRTOS IPC

- Characteristics of a Queue
  - Data Storage
    - Hold Finite amount of Data
    - Number of items called the queue length
  - Access by multiple tasks
    - Objects in their own right
    - Generally multiple writers, one reader
  - Queue reads and writes can block
    - Specify the block time

FreeRTOS IPC

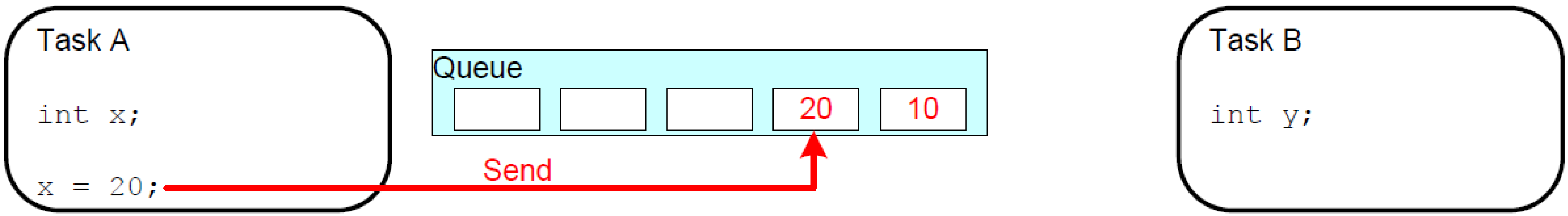


A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.

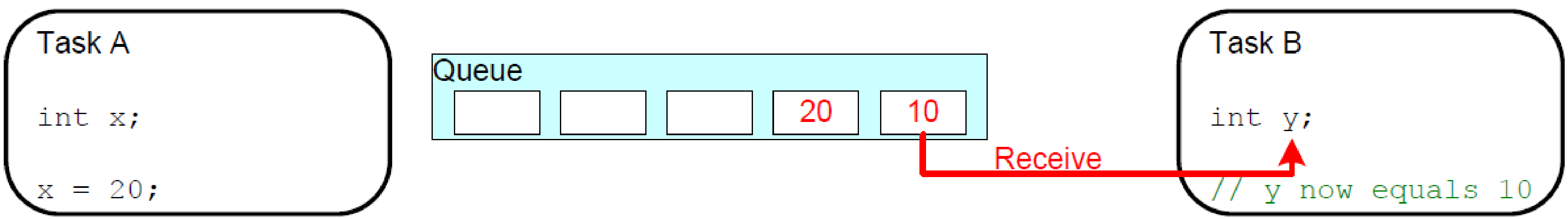


Task A writes (sends) the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.

FreeRTOS IPC

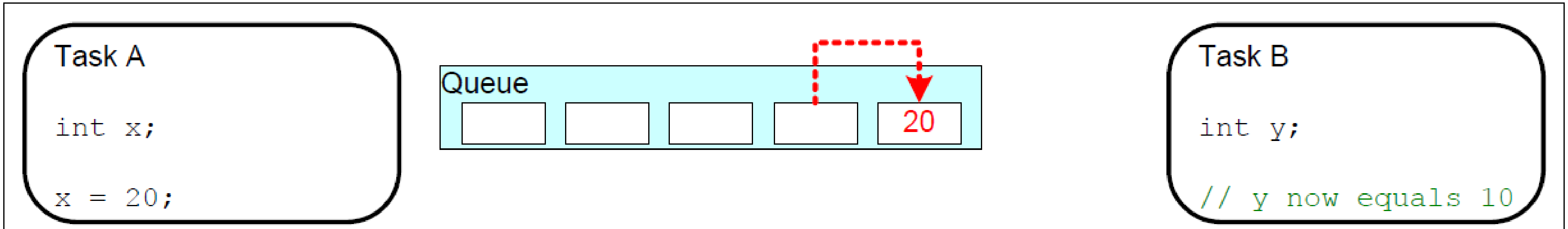


Task A changes the value of its local variable before writing it to the queue again. The queue now contains copies of both values written to the queue. The first value written remains at the front of the queue, the new value is inserted at the end of the queue. The queue has three empty spaces remaining.



Task B reads (receives) from the queue into a different variable. The value received by Task B is the value from the head of the queue, which is the first value Task A wrote to the queue (10 in this illustration).

FreeRTOS IPC



Task B has removed one item, leaving only the second value written by Task A remaining in the queue. This is the value Task B would receive next if it read from the queue again. The queue now has four empty spaces remaining.

FreeRTOS IPC

- Creates a new queue instance. This allocates the storage required by the new queue and returns a handle for the queue.

```
QueueHandle_t xQueueCreate(
   unsigned BaseType_t uxQueueLength,
   unsigned BaseType_t uxItemSize );
```

  - `uxQueueLength`
    - The maximum number of items that the queue can contain.
  - `uxItemSize`
    - The number of bytes each item in the queue will require. Items are queued by copy, not by reference, so this is the number of bytes that will be copied for each posted item. Each item on the queue must be the same size.
  - Returns
    - If the queue is successfully create then a handle to the newly created queue is returned. If the queue cannot be created then 0 is returned.

FreeRTOS IPC

```
struct AMessage {
  char ucMessageID;
  char ucData[ 20 ];
};

void vATask( void *pvParameters )
{
  QueueHandle_t xQueue1, xQueue2;
  // Create a queue capable of containing 10 unsigned long values.
  xQueue1 = xQueueCreate( 10, sizeof( unsigned long ) );
  if( xQueue1 == 0 )
  {
    // Queue was not created and must not be used.
  }
  // Create a queue capable of containing 10 pointers to AMessage structures.
  // These should be passed by pointer as they contain a lot of data.
  xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );
  if( xQueue2 == 0 )
  {
    // Queue was not created and must not be used.
  }
  // ... Rest of task code.
}
```

FreeRTOS IPC

- Delete a queue - freeing all the memory allocated for storing of items placed on the queue.

```
void vQueueDelete(
   QueueHandle_t xQueue );
```

  - `xQueue`
    - A handle to the queue to be deleted.

- Resets a queue to its original empty state.

```
BaseType_t xQueueReset(
   QueueHandle_t xQueue );
```

  - `xQueue`
    - The handle of the queue being reset

FreeRTOS IPC

- Post an item on a queue.

```
BaseType_t xQueueSendToBack(
                QueueHandle_t xQueue,
                const void * pvItemToQueue,
                TickType_t xTicksToWait );
```

  - **xQueue**
    - The handle to the queue on which the item is to be posted.
  - **pvItemToQueue**
    - A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from **pvItemToQueue** into the queue storage area.
  - **xTicksToWait**
    - The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0.
    - The time is defined in tick periods so the constant **portTICK_PERIOD_MS** should be used to convert to real time if this is required.
    - If **INCLUDE_vTaskSuspend** is set to '1' then specifying the block time as **portMAX_DELAY** will cause the task to block indefinitely (without a timeout).
  - Returns
    - **pdTRUE** if the item was successfully posted, otherwise **errQUEUE_FULL**.

FreeRTOS IPC

- Post an item to the front of a queue

```
BaseType_t xQueueSendToFront(
                QueueHandle_t xQueue,
                const void * pvItemToQueue,
                TickType_t xTicksToWait );
```

  - **xQueue**
    - The handle to the queue on which the item is to be posted.
  - **pvItemToQueue**
    - A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from **pvItemToQueue** into the queue storage area.
  - **xTicksToWait**
    - The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0.
    - The time is defined in tick periods so the constant **portTICK_PERIOD_MS** should be used to convert to real time if this is required.
    - If **INCLUDE_vTaskSuspend** is set to '1' then specifying the block time as **portMAX_DELAY** will cause the task to block indefinitely (without a timeout).
  - Returns
    - **pdTRUE** if the item was successfully posted, otherwise **errQUEUE_FULL**.

FreeRTOS IPC

- Return the number of messages stored in a queue

```
unsigned BaseType_t uxQueueMessagesWaiting(
  QueueHandle_t xQueue );
```

  - `xQueue`
    - A handle to the queue being queried.
  - Returns
    - The number of messages available in the queue.
- Return the number of free spaces in a queue

```
unsigned BaseType_t uxQueueSpacesAvailable(
  QueueHandle_t xQueue );
```

  - `xQueue`
    - A handle to the queue being queried.
  - Returns
    - The number of free spaces available in the queue.

FreeRTOS IPC

- Receive an item from a queue

```
BaseType_t xQueueReceive(
                QueueHandle_t xQueue,
                void *pvBuffer,
                TickType_t xTicksToWait );
```

  - `xQueue`
    - The handle to the queue on which the item is to be posted.
  - `*pvBuffer`
    - Pointer to the buffer into which the received item will be copied.
  - `xTicksToWait`
    - The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0.
    - The time is defined in tick periods so the constant `portTICK_PERIOD_MS` should be used to convert to real time if this is required.
    - If `INCLUDE_vTaskSuspend` is set to '1' then specifying the block time as `portMAX_DELAY` will cause the task to block indefinitely (without a timeout).
  - Returns
    - `pdTRUE` if an item was successfully received from the queue, otherwise `pdFALSE`

FreeRTOS IPC

- Receive an item from a queue without removing the item from the queue

```
BaseType_t xQueuePeek(
                QueueHandle_t xQueue,
                void *pvBuffer,
                TickType_t xTicksToWait );
```

  - **`xQueue`**
    - The handle to the queue on which the item is to be posted.
  - **`*pvBuffer`**
    - Pointer to the buffer into which the received item will be copied.
  - **`xTicksToWait`**
    - The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0.
    - The time is defined in tick periods so the constant **`portTICK_PERIOD_MS`** should be used to convert to real time if this is required.
    - If **`INCLUDE_vTaskSuspend`** is set to '1' then specifying the block time as **`portMAX_DELAY`** will cause the task to block indefinitely (without a timeout).
  - Returns
    - **`pdTRUE`** if an item was successfully received (peeked) from the queue, otherwise **`pdFALSE`**.

FreeRTOS IPC

- A version of **`xQueueSendToBack()`** that will write to the queue even if the queue is full, overwriting data that is already held in the queue.
- **`xQueueOverwrite()`** is intended for use with queues that have a length of one, meaning the queue is either empty or full.

```
BaseType_t xQueueOverwrite(
              QueueHandle_t xQueue,
              const void * pvItemToQueue );
```

  - **`xQueue`**
    - The handle of the queue to which the data is to be sent.
  - **`pvItemToQueue`**
    - A pointer to the item that is to be placed in the queue. The size of the items the queue will hold was defined when the queue was created, and that many bytes will be copied from pvItemToQueue into the queue storage area.
  - Returns
    - **`pdPASS`** is the only value that can be returned because **`xQueueOverwrite()`** will write to the queue even when the queue was already full.