



---

# Sistemas Operativos

Pedro Corcuera

Dpto. Matemática Aplicada y  
Ciencias de la Computación

**Universidad de Cantabria**

`corcuerp@unican.es`

# Índice General

---

- Introducción
- Ecuaciones diferenciales
- Método Euler
- Método Runge Kutta
- Ejemplo de uso de librería NR
- Medida de tiempo y sleep
- Simulación de control PID
- Simulación de control de nivel
- Ajuste de controladores PID
- Simulación de circuitos lógicos combinacionales

# What is an Operating System?

---

- A program that acts as an intermediary between a user of a computer and the computer hardware
- Operating system goals:
  - Execute user programs and make solving user problems easier
  - Make the computer system convenient to use
  - Use the computer hardware in an efficient manner

# Operating System Definition

---

- OS is a *resource allocator*
  - Manages all resources
  - Decides between conflicting requests for efficient and fair resource use
- OS is a *control program*
  - Controls execution of programs to prevent errors and improper use of the computer
- “The one program running at all times on the computer” is the *kernel*. Everything else is either a system program (ships with the operating system) or an application program.

# Computer-System Operation

---

- I/O devices and the CPU can execute concurrently
- Each device controller is in charge of a particular device type
- Each device controller has a local buffer
- CPU moves data from/to main memory to/from local buffers
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by causing an **interrupt**

# Common Functions of Interrupts

---

- Interrupt transfers control to the interrupt service routine generally, through the **interrupt vector**, which contains the addresses of all the service routines
  - Interrupt architecture must save the address of the interrupted instruction
  - Incoming interrupts are *disabled* while another interrupt is being processed to prevent a *lost interrupt*
  - A *trap* is a software-generated interrupt caused either by an error or a user request
  - An operating system is **interrupt driven**
-

# Interrupt Handling

---

- The operating system preserves the state of the CPU by storing registers and the program counter
- Determines which type of interrupt has occurred:
  - **polling**
  - **vectored** interrupt system
- Separate segments of code determine what action should be taken for each type of interrupt

# Direct Memory Access Structure

---

- Used for high-speed I/O devices able to transmit information at close to memory speeds
- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention
- Only one interrupt is generated per block, rather than the one interrupt per byte

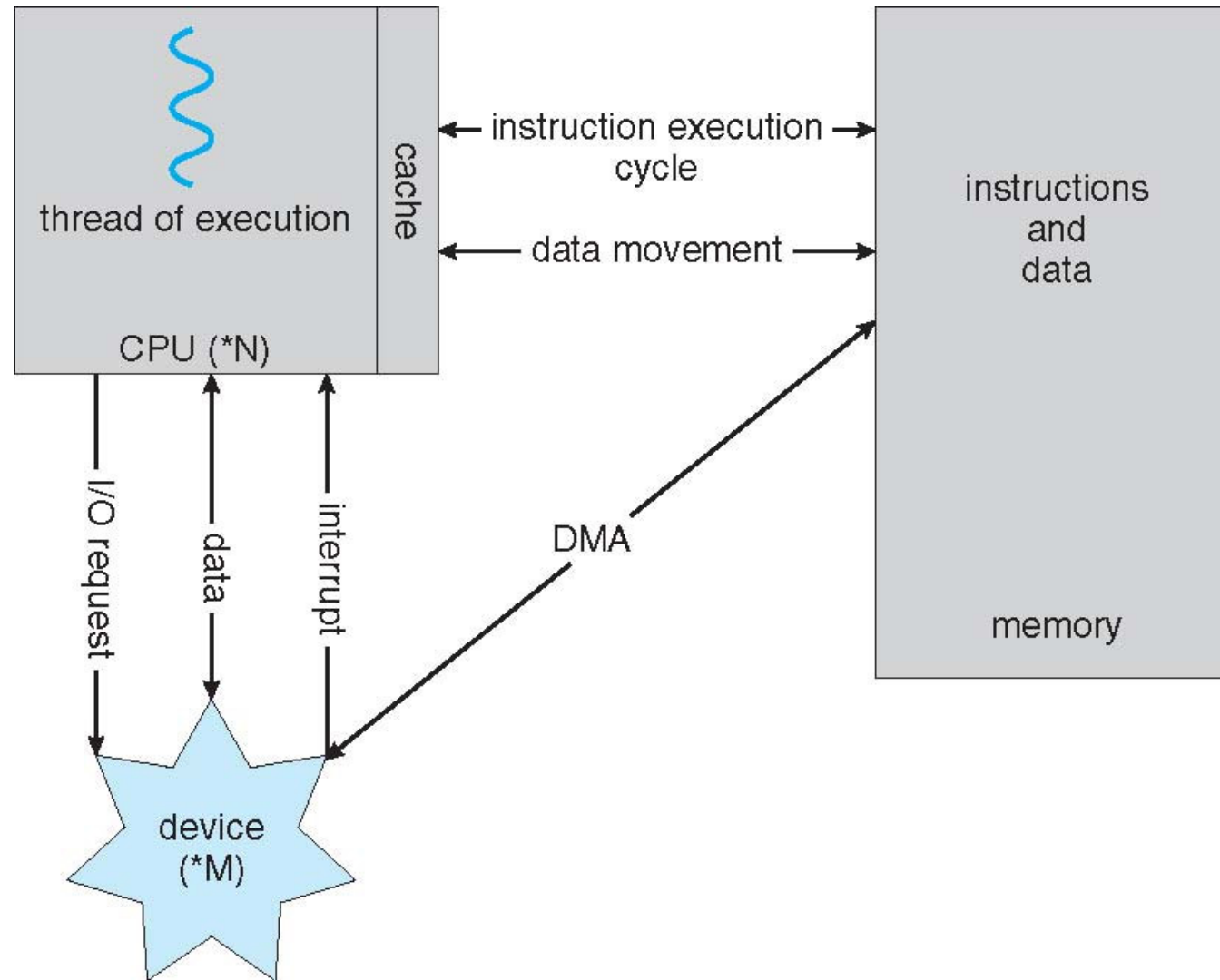


# Computer-System Architecture

---

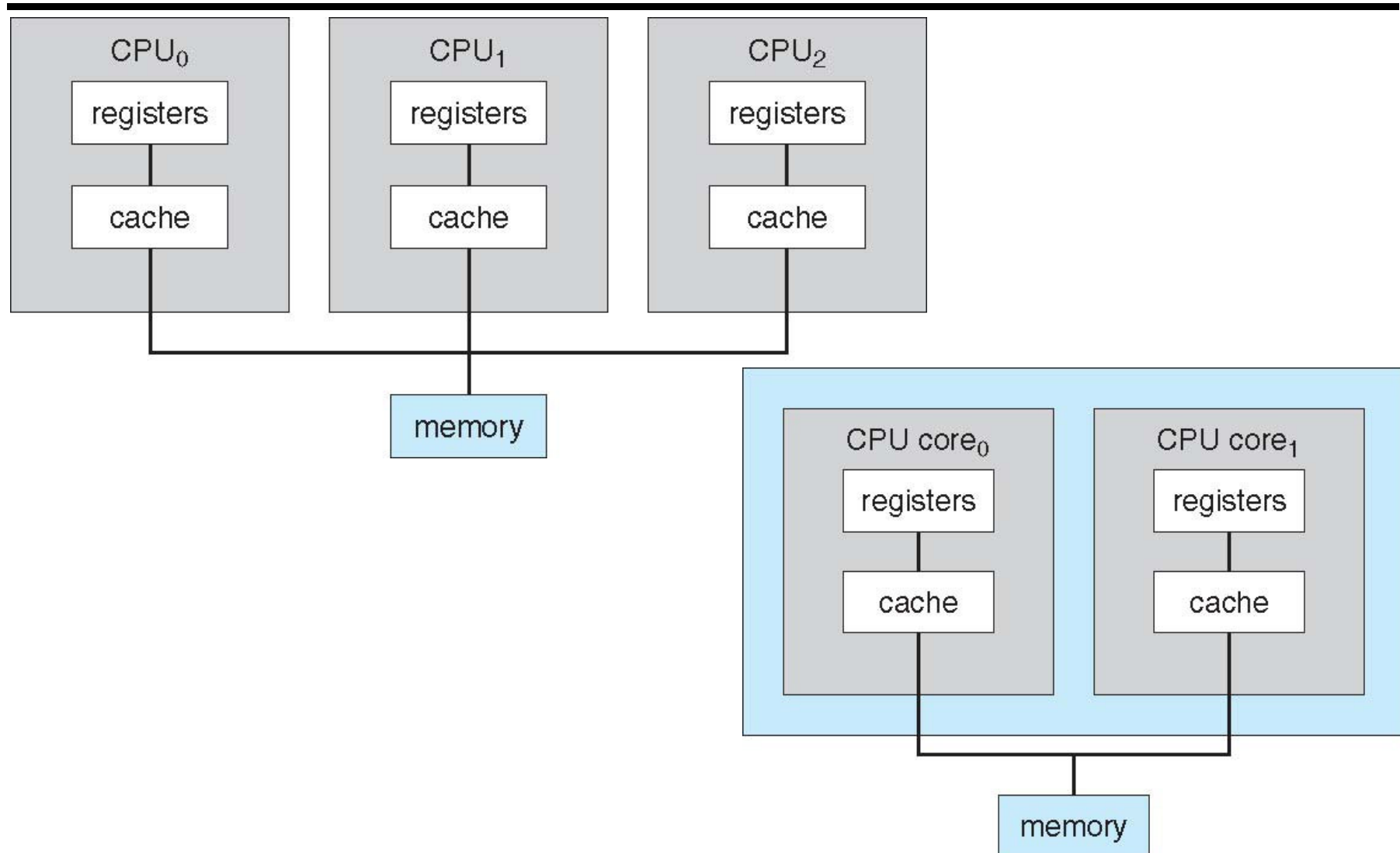
- Most systems use a single general-purpose processor
  - **Multiprocessors** systems growing in use and importance
    - Also known as **parallel systems**, **tightly-coupled systems**
    - Advantages include:
      1. **Increased throughput**
      2. **Economy of scale**
      3. **Increased reliability – graceful degradation or fault tolerance**
    - Two types:
      1. **Asymmetric Multiprocessing**
      2. **Symmetric Multiprocessing**
-

# How a Modern Computer Works



# Symmetric Multiprocessing Architecture

## A Dual-Core Design



# Operating System Structure

---

- **Multiprogramming** needed for efficiency
  - Single user cannot keep CPU and I/O devices busy at all times
  - Multiprogramming organizes jobs (code and data) so CPU always has one to execute
  - A subset of total jobs in system is kept in memory
  - One job selected and run via **job scheduling**
  - When it has to wait (for I/O for example), OS switches to another job

# Operating System Structure

---

- **Timesharing (multitasking)** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing
  - **Response time** should be  $< 1$  second
  - Each user has at least one program executing in memory  $\Rightarrow$  **process**
  - If several jobs ready to run at the same time  $\Rightarrow$  **CPU scheduling**
  - If processes don't fit in memory, **swapping** moves them in and out to run
  - **Virtual memory** allows execution of processes not completely in memory

# Process Management

---

- A process is a program in execution. It is a unit of work within the system. Program is a *passive entity*, process is an *active entity*.
- Process needs resources to accomplish its task
  - CPU, memory, I/O, files
  - Initialization data
- Process termination requires reclaim of any reusable resources

# Process Management

---

- Single-threaded process has one **program counter** specifying location of next instruction to execute
  - Process executes instructions sequentially, one at a time, until completion
- Multi-threaded process has one program counter per thread
- Typically system has many processes, some user, some operating system running concurrently on one or more CPUs
  - Concurrency by multiplexing the CPUs among the processes / threads

# Distributed Computing

---

- Collection of separate, possibly heterogeneous, systems networked together
  - Network is a communications path
    - Local Area Network (**LAN**)
    - Wide Area Network (**WAN**)
    - Metropolitan Area Network (**MAN**)
- Network Operating System provides features between systems across network
  - Communication scheme allows systems to exchange messages
  - Illusion of a single system



# Special-Purpose Systems

---

- Real-time embedded systems most prevalent form of computers
  - Vary considerable, special purpose, limited purpose OS, **real-time OS**
- Multimedia systems
  - Streams of data must be delivered according to time restrictions
- Handheld systems
  - PDAs, smart phones, limited CPU, memory, power
  - Reduced feature set OS, limited I/O

# Web-Based Computing

---

- Web has become ubiquitous
- PCs most prevalent devices
- More devices becoming networked to allow web access
- New category of devices to manage web traffic among similar servers: **load balancers**
- Use of operating systems like Windows 95, client-side, have evolved into Linux and Windows XP, which can be clients and servers

# System Calls

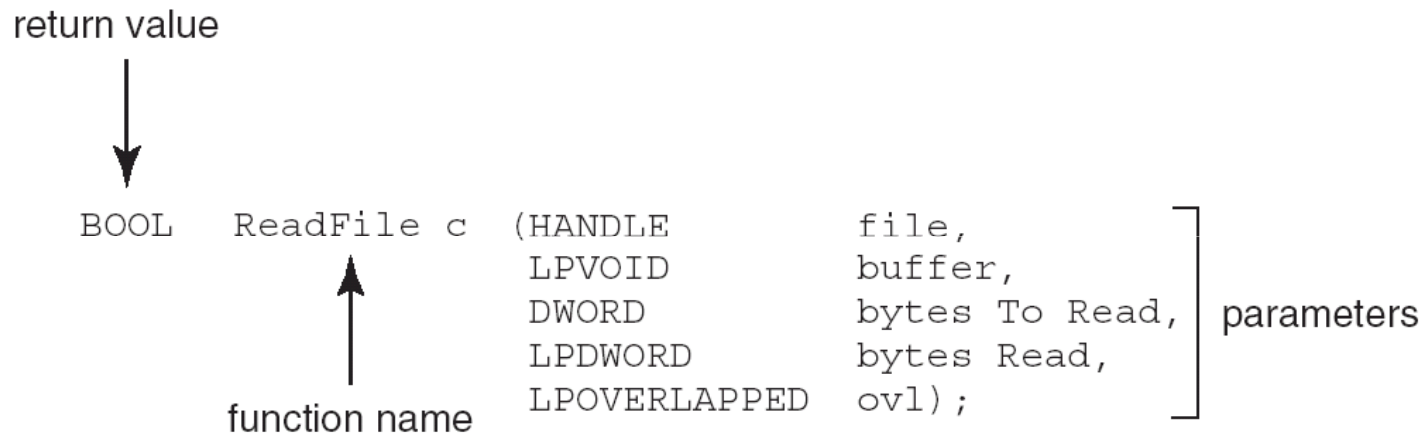
---

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Program Interface (API)** rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

# Example of Standard API

---

- Consider the ReadFile() function in the
- Win32 API—a function for reading from a file



- A description of the parameters passed to ReadFile()
  - HANDLE file—the file to be read
  - LPVOID buffer—a buffer where the data will be read into and written from
  - DWORD bytesToRead—the number of bytes to be read into the buffer
  - LPDWORD bytesRead—the number of bytes read during the last read
  - LPOVERLAPPED ovl—indicates if overlapped I/O is being used

# Types of System Calls

---

- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes

# System Calls

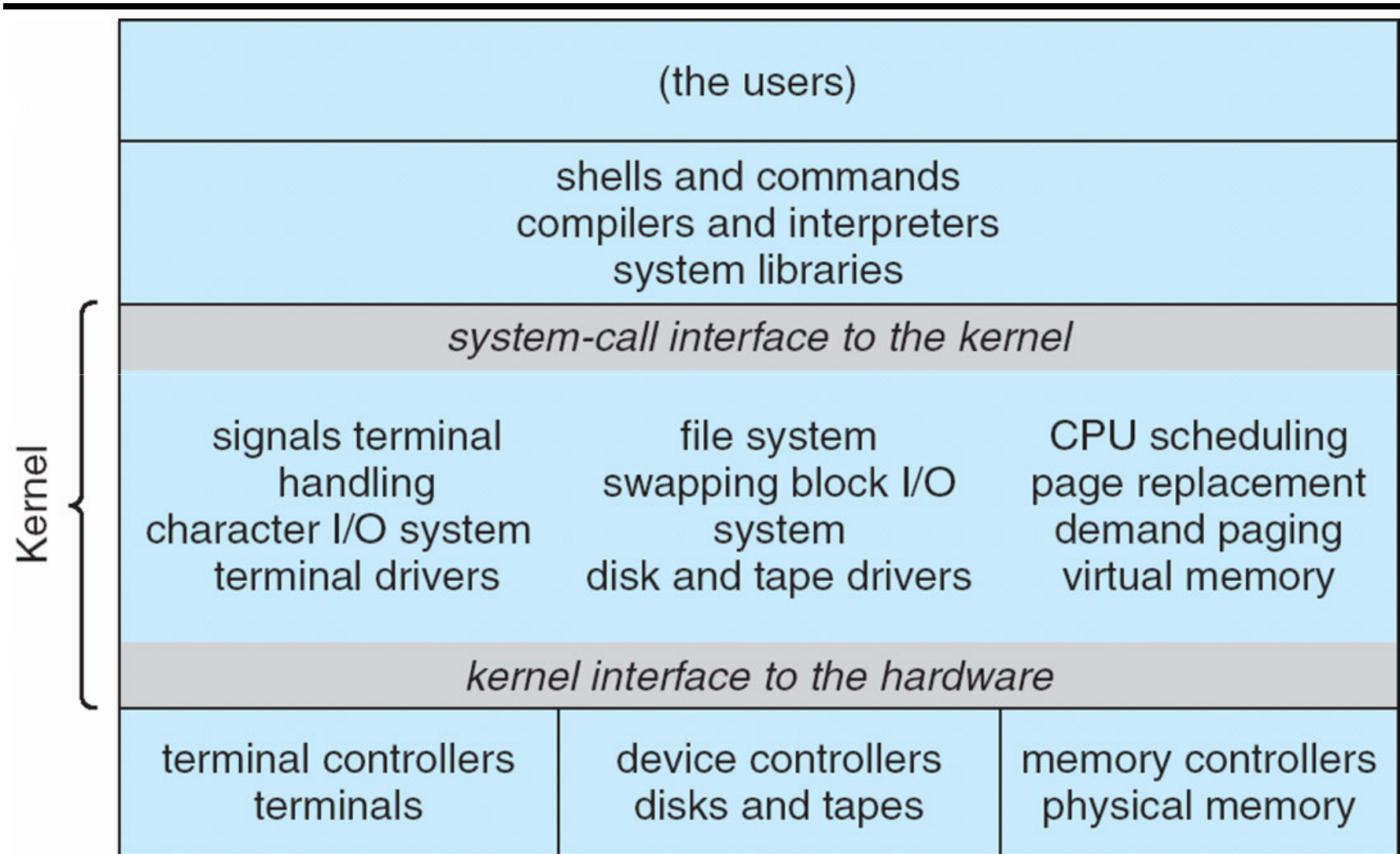
---

- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach and detach remote devices

# Examples of Windows and Unix System Calls

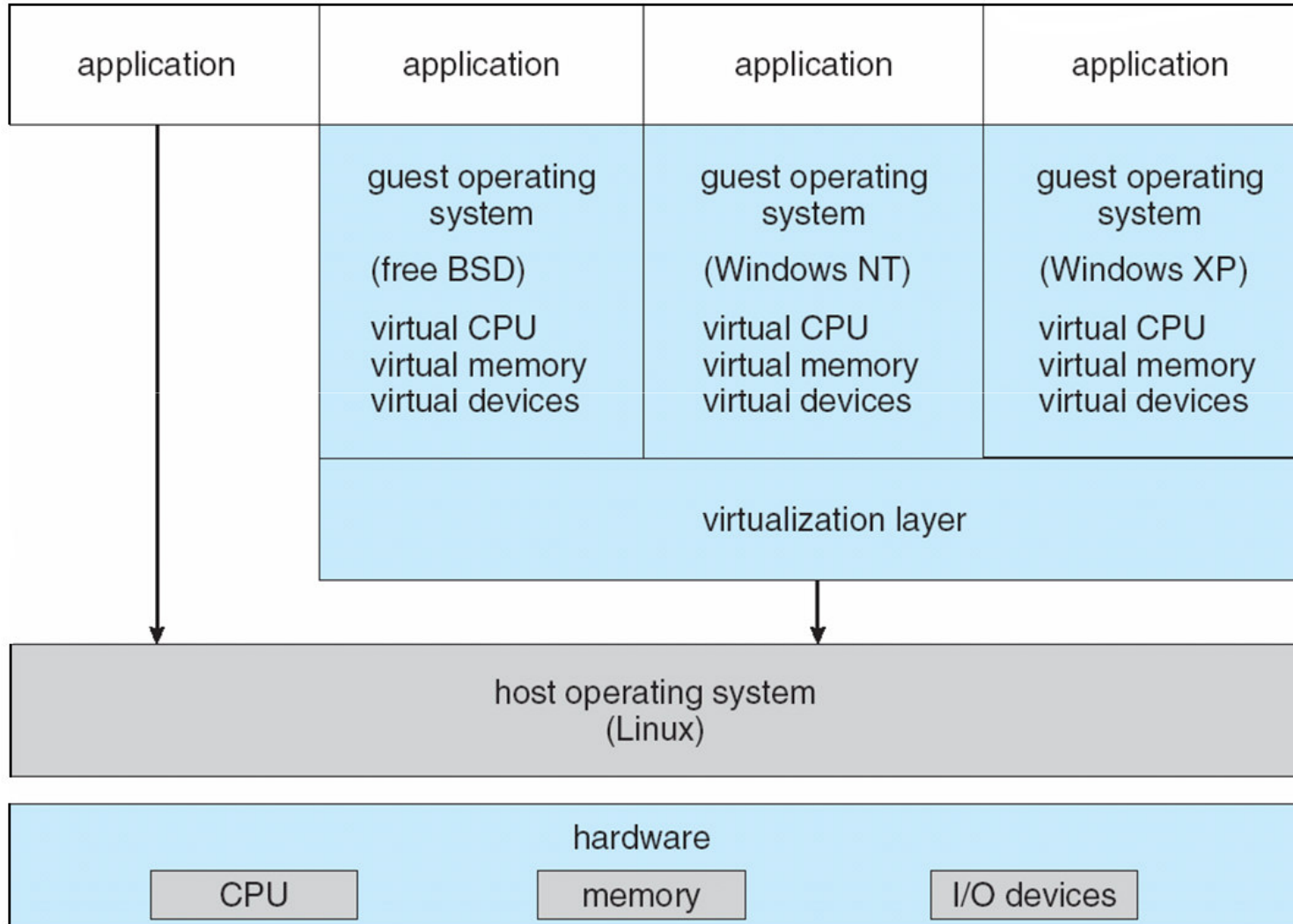
	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

# Traditional UNIX System Structure





# VMware Architecture



# The Process

---

- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time
- Program is passive entity, process is active
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program

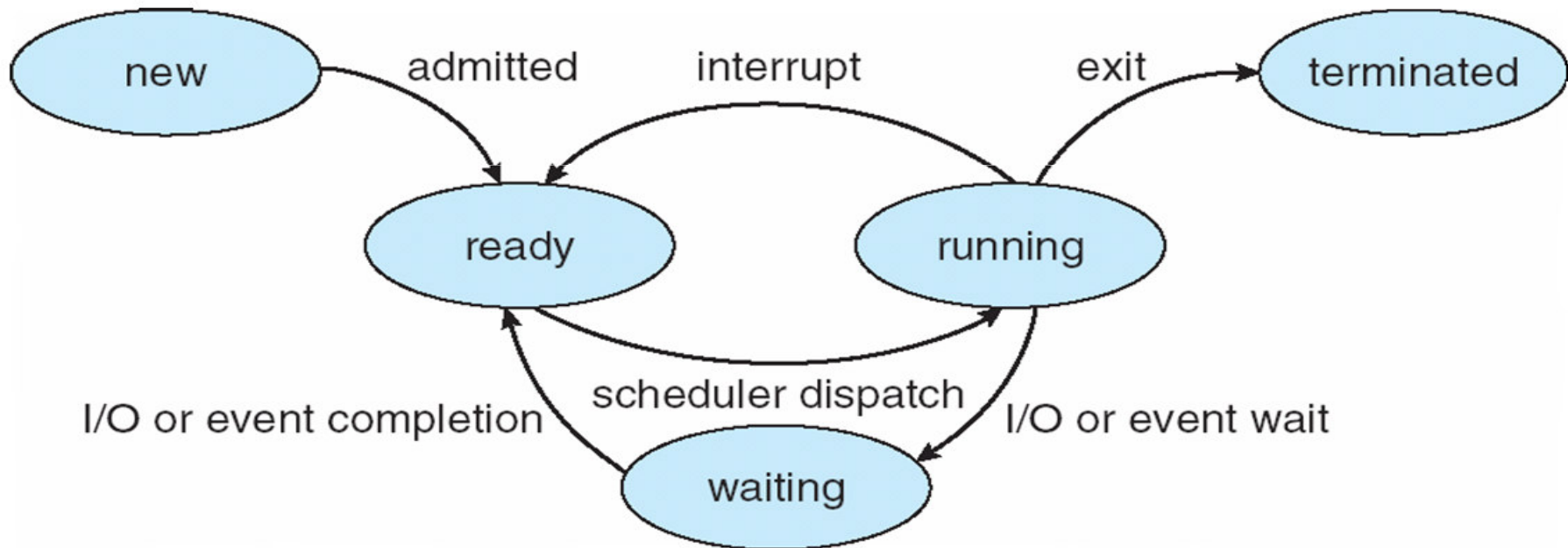
# Process State

---

- As a process executes, it changes *state*
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution

# Diagram of Process State

---



# Process Scheduling

---

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues

# Schedulers

---

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system

# Schedulers

---

- Short-term scheduler is invoked very frequently (milliseconds)  $\Rightarrow$  (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes)  $\Rightarrow$  (may be slow)
  - The long-term scheduler controls the *degree of multiprogramming*
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts

# Process Creation

---

- **Parent** process create **children** processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a **process identifier (pid)**
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources



# Process Creation

---

- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program

# C Program Forking Separate Process

---

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main() {
    pid_t pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed"); return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process parent will wait for the child */
        wait (NULL); printf ("Child Complete");
    }
    return 0;
}
```

# Interprocess Communication

---

- Processes within a system may be **independent** or **cooperating**. Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication (IPC)**. Two models of IPC
  - Shared memory
  - Message passing

# 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
- Examples: <http://www.cs.cf.ac.uk/Dave/C/>

# 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
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets

# Socket Communication

---

host X  
(146.86.5.20)



web server  
(161.25.19.8)



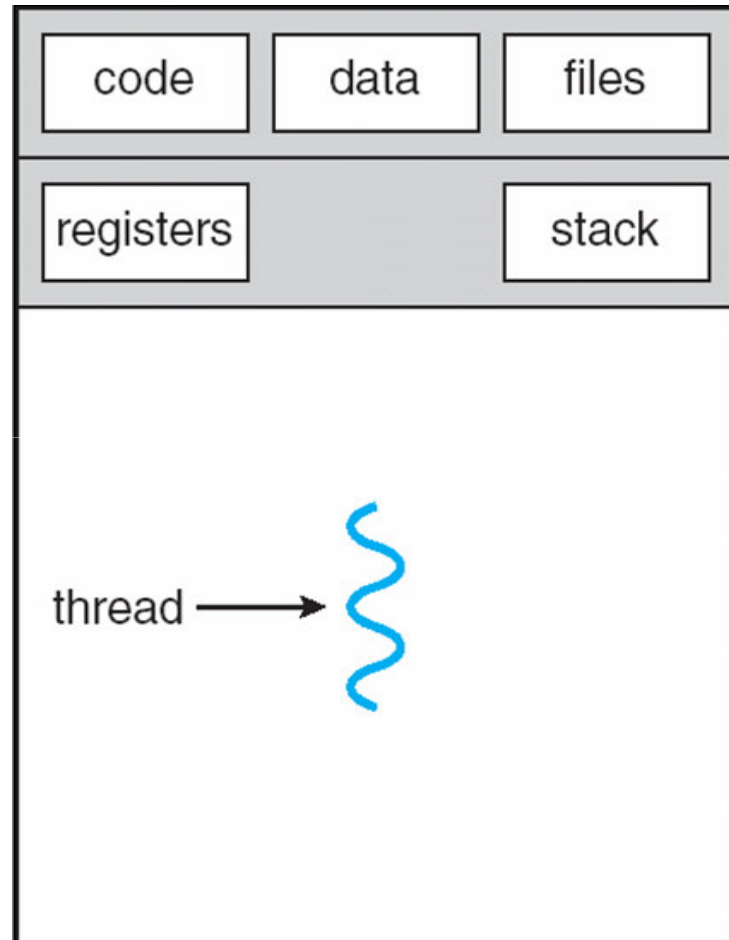
# Remote Procedure Calls

---

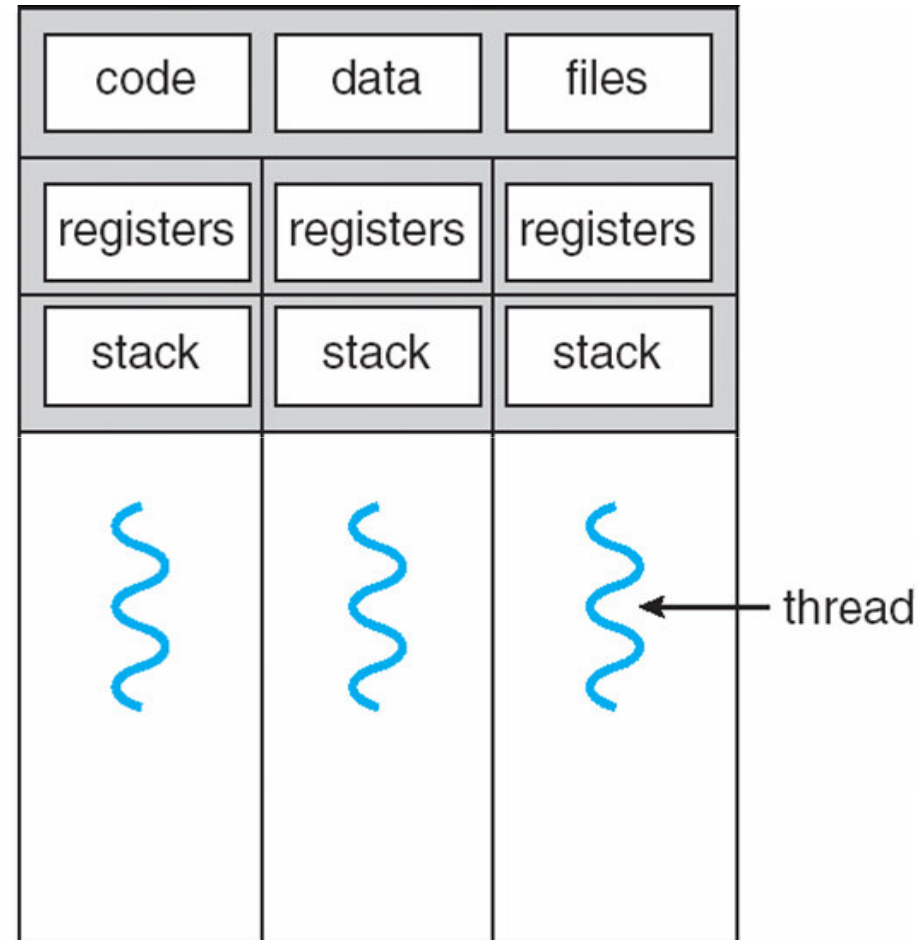
- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and *marshalls* the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server



# Single and Multithreaded Processes



single-threaded process



multithreaded process

# User Threads

---

- Thread management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Win32 threads
  - Java threads
- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

# Pthreads

---

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

# Pthreads Example

---

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    if (argc != 2) {
        fprintf(stderr, "usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr, "%d must be >= 0\n", atoi(argv[1]));
        return -1;
    }
}
```

# Pthreads Example

---

```
    /* get the default attributes */
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid, &attr, runner, argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid, NULL);

    printf("sum = %d\n", sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```

# Classical Problems of Synchronization

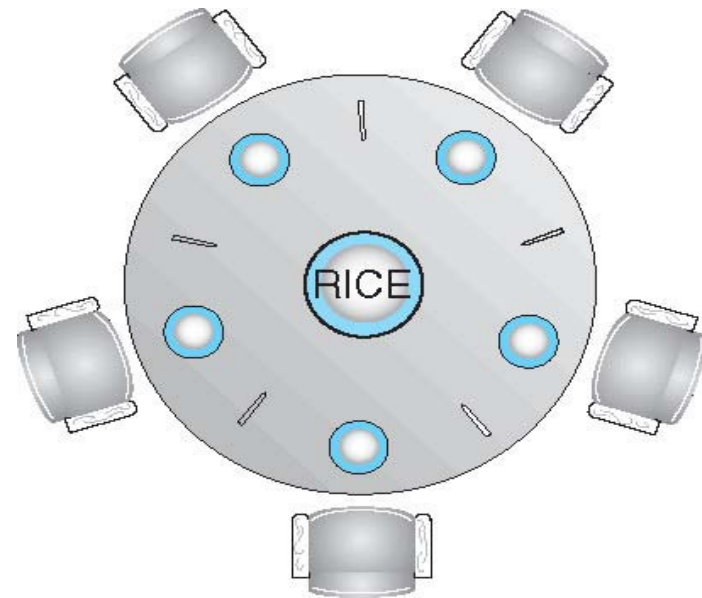
---

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

# Dining-Philosophers Problem

---

- Philosophers spend their lives thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore **chopstick [5]** initialized to 1



# Overview of Real-Time Systems

---

- A **real-time system** requires that results be produced within a specified deadline period.
- An **embedded system** is a computing device that is part of a larger system (i.e., automobile, airliner).
- A **safety-critical system** is a real-time system with catastrophic results in case of failure.
- A **hard real-time system** guarantees that real-time tasks be completed within their required deadlines.
- A **soft real-time system** provides priority of real-time tasks over non real-time tasks.



# Features of Real-Time Kernels

---

- Most real-time systems do not provide the features found in a standard desktop system
- Reasons include
  - Real-time systems are typically single-purpose
  - Real-time systems often do not require interfacing with a user
  - Features found in a desktop PC require more substantial hardware than what is typically available in a real-time system

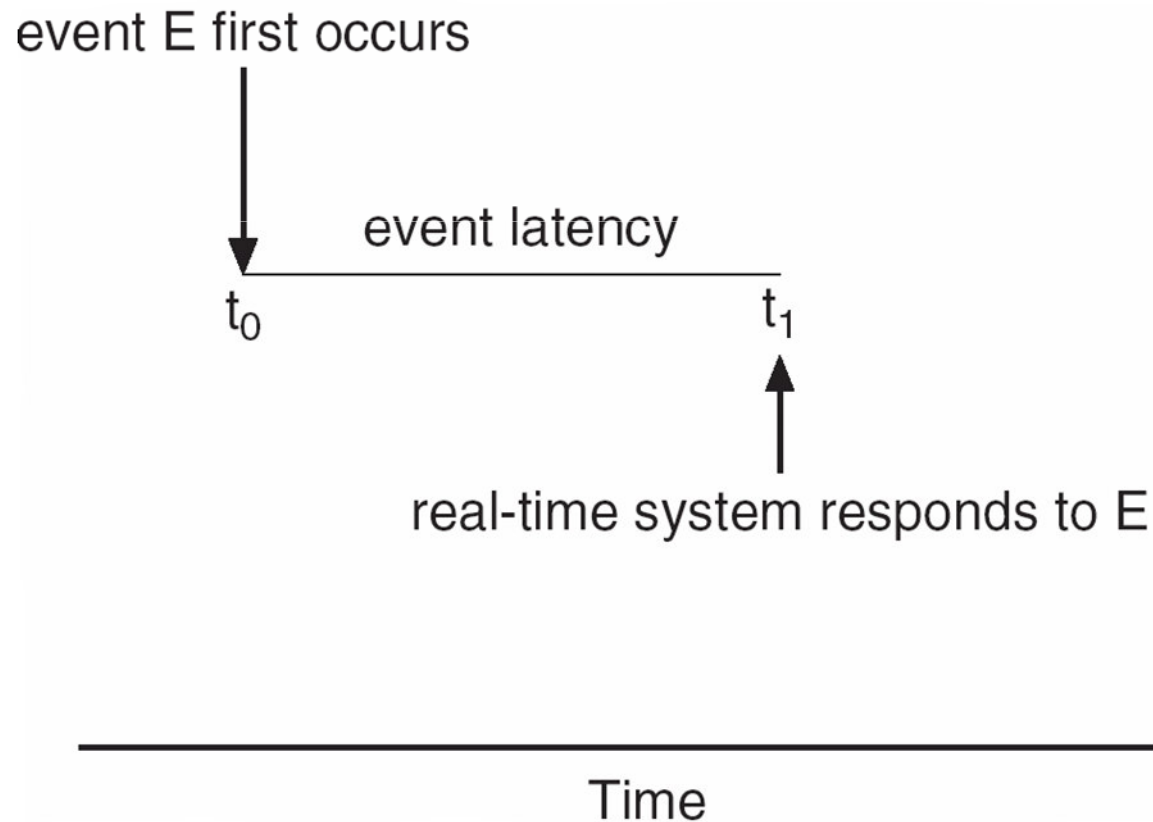
# Implementing Real-Time Systems

---

- In general, real-time operating systems must provide:
  1. Preemptive, priority-based scheduling
  2. Preemptive kernels
  3. Latency must be minimized

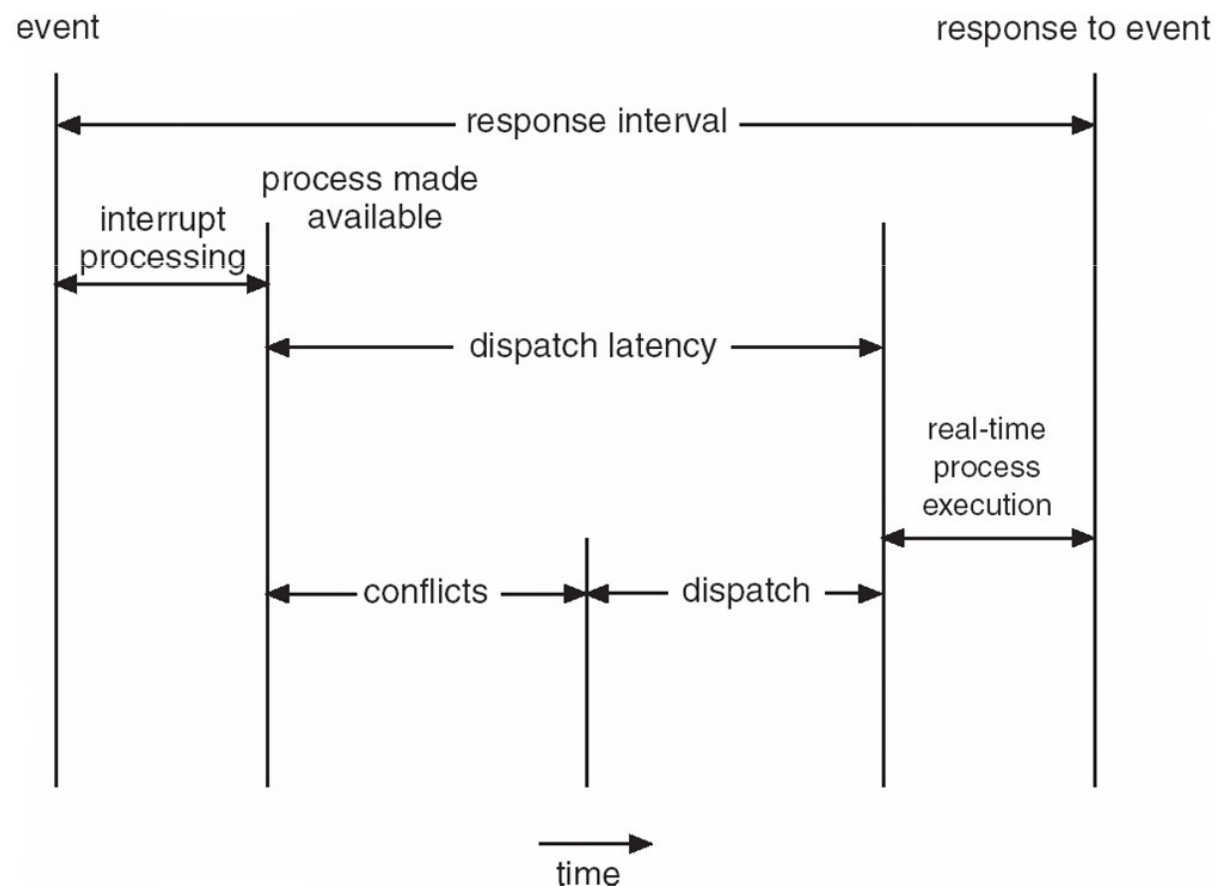
# Minimizing Latency

- **Event latency** is the amount of time from when an event occurs to when it is serviced.



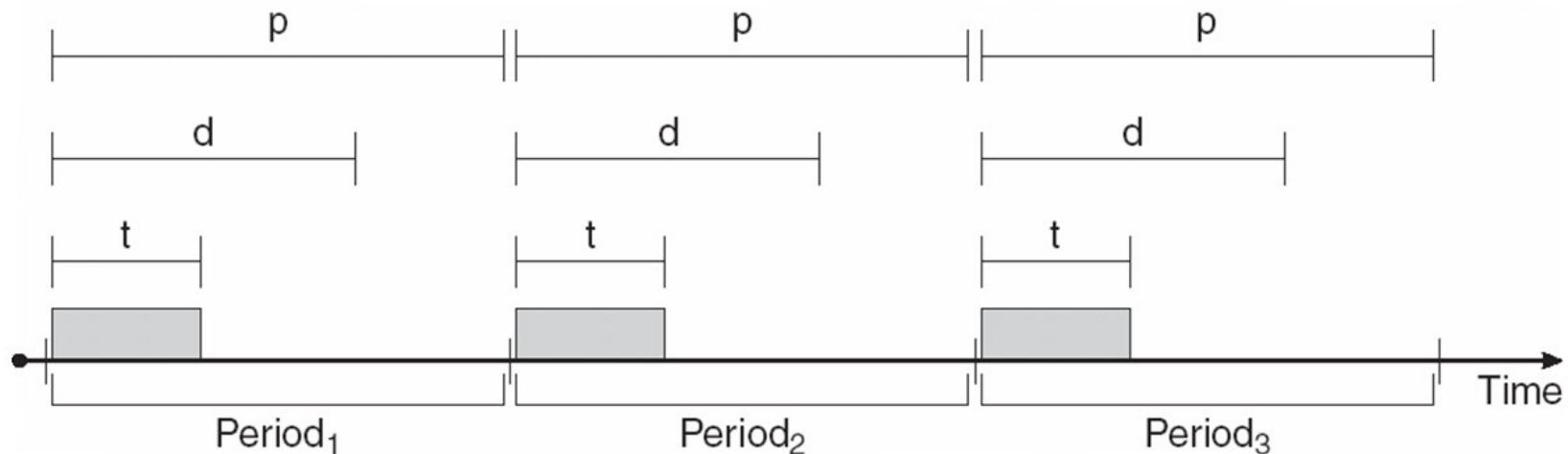
# Dispatch Latency

- **Dispatch latency** is the amount of time required for the scheduler to stop one process and start another



# Real-Time CPU Scheduling

- Periodic processes require the CPU at specified intervals (periods)
- $p$  is the duration of the period
- $d$  is the deadline by when the process must be serviced
- $t$  is the processing time



# Pthread Scheduling

---

- The Pthread API provides functions for managing real-time threads
- Pthreads defines two scheduling classes for real-time threads:
  1. `SCHED_FIFO` - threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority
  2. `SCHED_RR` - similar to `SCHED_FIFO` except time-slicing occurs for threads of equal priority

# RTOS example: FreeRTOS

---

- FreeRTOS (<http://www.freertos.org/>) is an free and opensource RealTime Operating system developed by Real Time Engineers Ltd.
- Its design has been developed to:
  - fit on very small embedded systems
  - implements only a very minimalist set of functions (basic handle of tasks and memory management, just sufficient API concerning synchronization)

# FreeRTOS

---

- Among its features are the following characteristics:
  - preemptive tasks, a support for 23 microcontroller architectures<sup>1</sup> by its developers, a small footprint<sup>2</sup> (4.3Kbytes on an ARM7 after compilation<sup>3</sup>)
  - written in C and compiled with various C compiler (some ports are compiled with gcc, others with openwatcom or borland c++).
  - Allows an unlimited number of tasks to run at the same time and no limitation about their priorities as long as used hardware can afford it. Finally, it implements queues, binary and counting semaphores and mutexes.

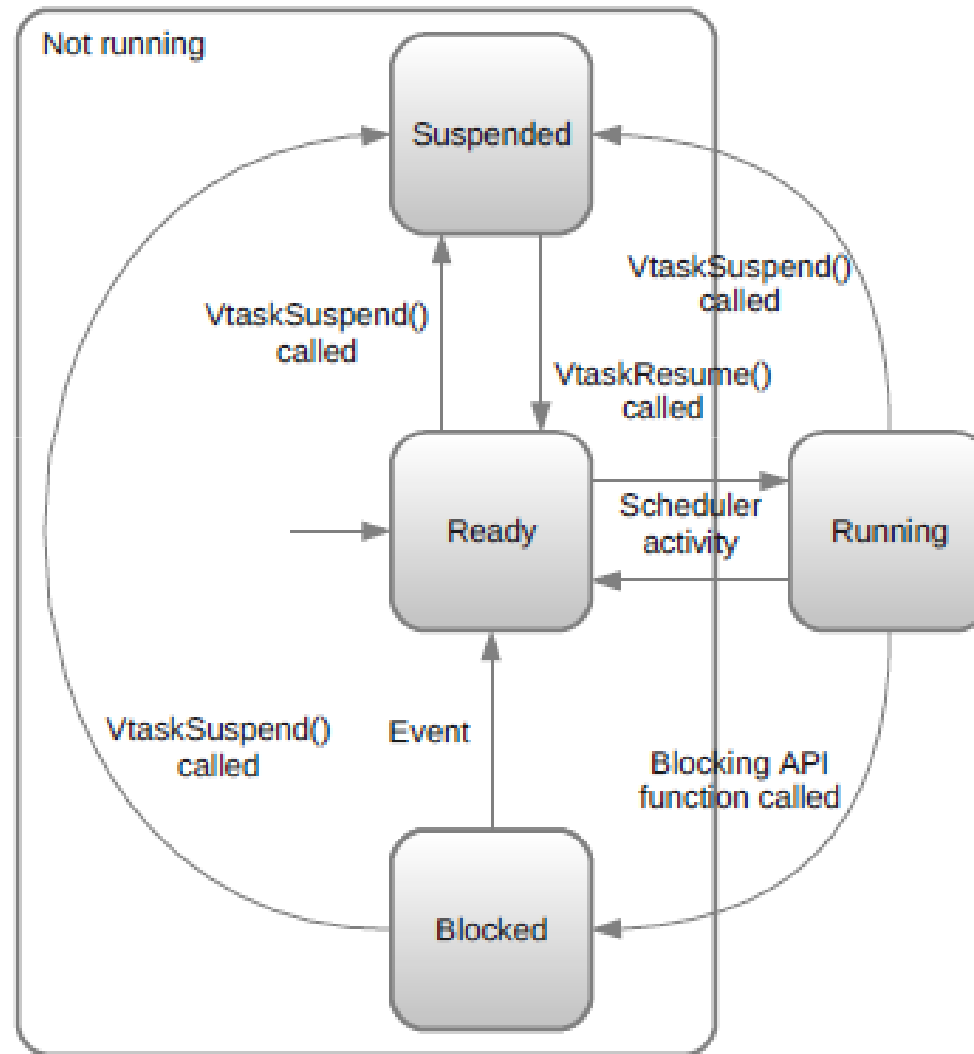


# Tasks in FreeRTOS

---

- FreeRTOS allows an unlimited number of tasks to be run as long as hardware and memory can handle it and is able to handle both cyclic and acyclic tasks.
- A task is defined by a *simple C function*, taking a void\* parameter and returning nothing (void).
- Several functions are available to manage tasks:
  - task creation (vTaskCreate()), destruction (vTaskDelete()), priority management (uxTaskPriorityGet(), vTaskPrioritySet()) or delay/resume ((vTaskDelay(), vTaskDelayUntil(), vTaskSuspend(), vTaskResume(), vTaskResumeFromISR()).

# Life cycle of a task



# Tasks in FreeRTOS

---

- FreeRTOS allows an unlimited number of tasks to be run as long as hardware and memory can handle it and is able to handle both cyclic and acyclic tasks.
- A task is defined by a *simple C function*, taking a void\* parameter and returning nothing (void).
- Several functions are available to manage tasks:
  - task creation (vTaskCreate()), destruction (vTaskDelete()), priority management (uxTaskPriorityGet(), vTaskPrioritySet()) or delay/resume ((vTaskDelay(), vTaskDelayUntil(), vTaskSuspend(), vTaskResume(), vTaskResumeFromISR()).

# Creating a task

---

- A task can be created using `vTaskCreate()`
  - **pvTaskCode**: a pointer to the function where the task is implemented.
  - **pcName**: given name to the task. This is intended to debugging purpose.
  - **usStackDepth**: length of the stack for this task in words. The actual size of the stack depends on the micro controller. If stack is 32 bits (4 bytes) and `usStackDepth` is 100, then 400 bytes (4 times 100) will be allocated.
  - **pvParameters**: a pointer to arguments given to the task. A good practice consists in creating a dedicated structure, instantiate and fill it then give its pointer to the task.
  - **uxPriority**: priority given to the task, a number between 0 and `MAX_PRIORITIES - 1`.
  - **pxCreatedTask**: a pointer to an identifier that allows to handle the task. If the task does not have to be handled in the future, this can be leaved NULL.

# Creating a task

---

```
portBASE_TYPE xTaskCreate( pdTASK_CODE pvTaskCode,
                           const signed portCHAR * const pcName,
                           unsigned portSHORT usStackDepth,
                           void *pvParameters,
                           unsigned portBASE_TYPE uxPriority,
                           xTaskHandle *pxCreatedTask
                           );

void ATaskFunction( void *pvParameters )
{
    /* Variables can be declared just as per a normal function. Each instance
       of a task created using this function will have its own copy of the
       iVaribleExample variable. If the variable is declared static only one
       copy of the variable would exist and would be shared by each task. */
    int iVaribleExample = 0;
    /* A task will normally be implemented as in infinite loop. */
    for( ;; )
    {
        /* The code to implement the task functionality will go here. */
    }
    /* Should the task implementation ever break out of the above loop
       then the task must be deleted before reaching the end of this function.
       The NULL parameter passed to the vTaskDelete() function indicates that
       the task to be deleted is the calling (this) task. */
    vTaskDelete( NULL );
}
```

---

## Deleting a task

---

- A task is destroyed using `xTaskDestroy()` routine. It takes as argument `pxCreatedTask` which is given when the task was created.

```
void vTaskDelete( xTaskHandle pxTask );
```

- When a task is deleted, it is responsibility of idle task to free all allocated memory to this task by kernel. Notice that all memory dynamically allocated must be manually freed.

# FreeRTOS example: installation & PC Demo

---

- Descargar FreeRTOS [[FreeRTOSV7.3.0.exe](#)] de <http://www.freertos.org> ,e instalar .
- Ir a la página de <http://www.openwatcom.org> y descargar open-watcom-c-win32-1.9.exe , instalar en la ruta sugerida [C:\WATCOM]
  - Cuando el instalador pregunte sobre el tipo de instalacion, seleccionar Full installation.
- Abrir proyecto dirFreeRTOS\Demo\PC\rtosdemo.wpj y en el archivo main.c comentar la linea 193, así:

```
//vStartComTestTasks( mainCOM_TEST_PRIORITY, serCOM1, ser115200 );
```

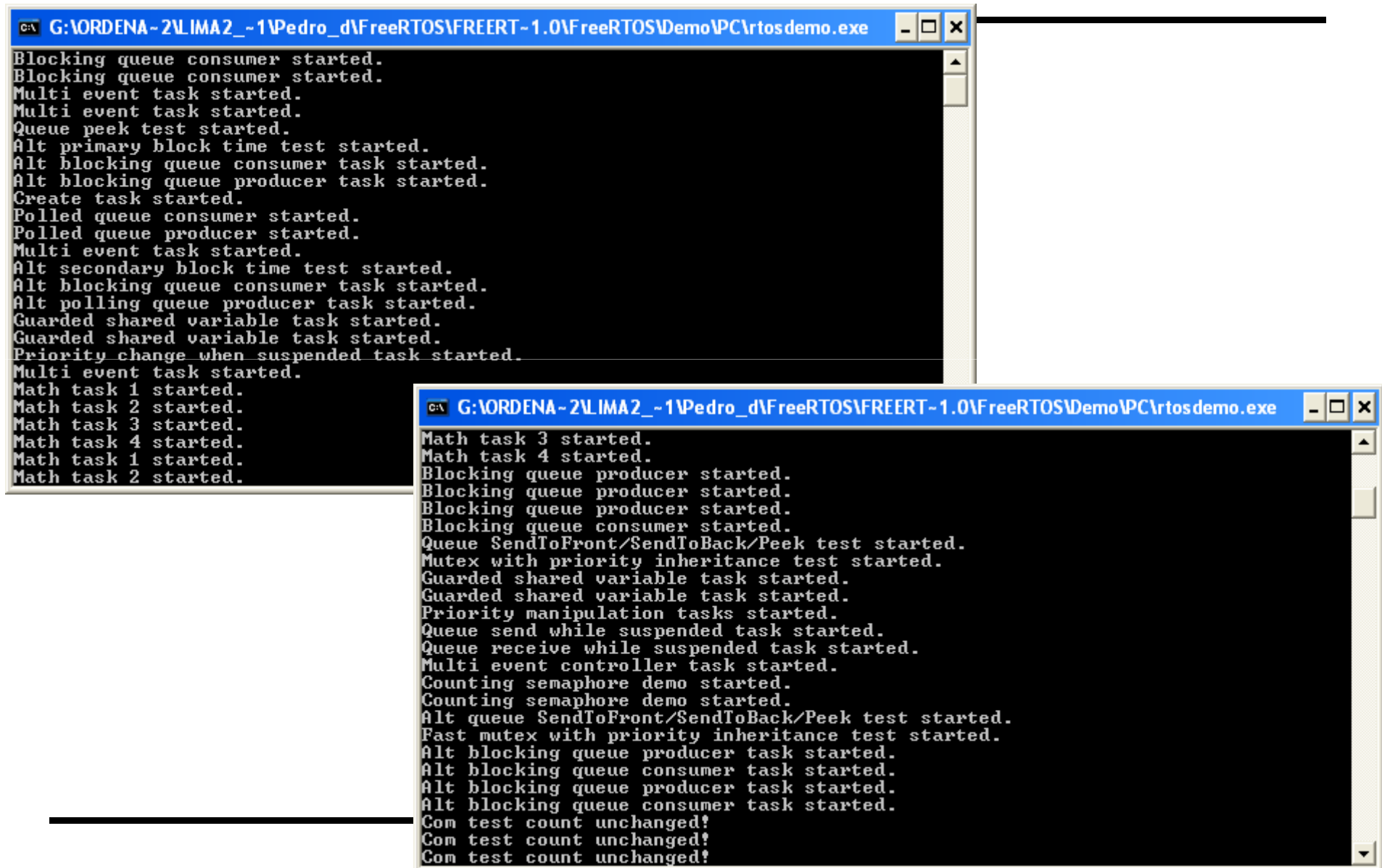
# FreeRTOS example: installation & PC Demo

---

- Abrir el IDE de Watcom, presionar F5 (make). Si todo va bien, se debe generar el archivo ejecutable rtosdemo.exe. Ejecutar
- Analizar resultado y código



# FreeRTOS example: installation & PC Demo



The image shows two overlapping Windows command prompt windows. The top window displays the output of the 'rtosdemo.exe' application, listing various tasks and tests that have started. The bottom window shows a continuation of the output, including math tasks, queue operations, mutex tests, and semaphore demonstrations.

```
G:\VORDENA-2\LIMA2_-1\Pedro_d\FreeRTOS\FREERT-1.0\FreeRTOS\Demo\PC\rtosdemo.exe
Blocking queue consumer started.
Blocking queue consumer started.
Multi event task started.
Multi event task started.
Queue peek test started.
Alt primary block time test started.
Alt blocking queue consumer task started.
Alt blocking queue producer task started.
Create task started.
Polled queue consumer started.
Polled queue producer started.
Multi event task started.
Alt secondary block time test started.
Alt blocking queue consumer task started.
Alt polling queue producer task started.
Guarded shared variable task started.
Guarded shared variable task started.
Priority change when suspended task started.
Multi event task started.
Math task 1 started.
Math task 2 started.
Math task 3 started.
Math task 4 started.
Math task 1 started.
Math task 2 started.

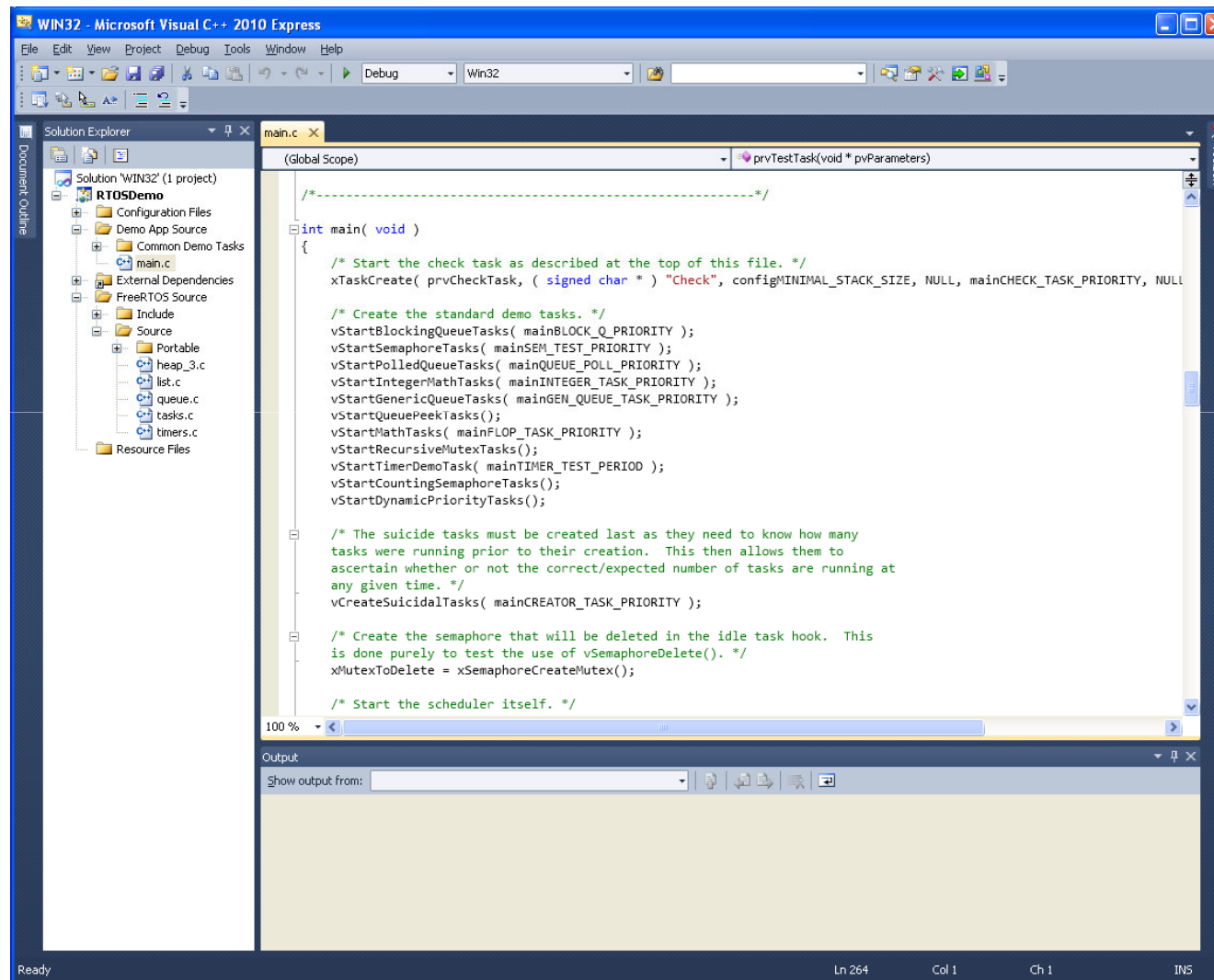
G:\VORDENA-2\LIMA2_-1\Pedro_d\FreeRTOS\FREERT-1.0\FreeRTOS\Demo\PC\rtosdemo.exe
Math task 3 started.
Math task 4 started.
Blocking queue producer started.
Blocking queue producer started.
Blocking queue producer started.
Blocking queue consumer started.
Queue SendToFront/SendToBack/Peek test started.
Mutex with priority inheritance test started.
Guarded shared variable task started.
Guarded shared variable task started.
Priority manipulation tasks started.
Queue send while suspended task started.
Queue receive while suspended task started.
Multi event controller task started.
Counting semaphore demo started.
Counting semaphore demo started.
Alt queue SendToFront/SendToBack/Peek test started.
Fast mutex with priority inheritance test started.
Alt blocking queue producer task started.
Alt blocking queue consumer task started.
Alt blocking queue producer task started.
Alt blocking queue consumer task started.
Com test count unchanged!
Com test count unchanged!
Com test count unchanged!
```

## FreeRTOS example: Simulator

---

- Ejecutar Visual C++ 2010 Express
- Abrir solución en Demo\Win32-MSVC\WIN32.sln
- Seleccionar Debug->Build Solution o tecla F7
- Seleccionar Debug->Start Debug
- Analizar resultado y código

# FreeRTOS example: simulator



The screenshot shows the Microsoft Visual C++ 2010 Express IDE. The Solution Explorer on the left displays a project named 'RTOSDemo' with a file tree including 'main.c'. The main editor window shows the source code for 'main.c'. The code includes comments and function calls for starting various tasks and the scheduler. The Output window at the bottom is currently empty.

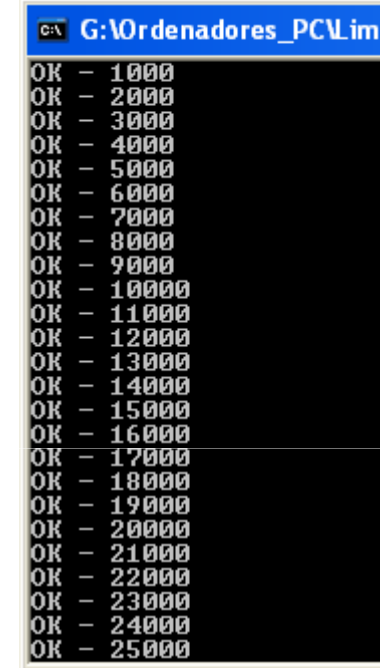
```
int main( void )
{
    /* Start the check task as described at the top of this file. */
    xTaskCreate( prvCheckTask, ( signed char * ) "Check", configMINIMAL_STACK_SIZE, NULL, mainCHECK_TASK_PRIORITY, NULL

    /* Create the standard demo tasks. */
    vStartBlockingQueueTasks( mainBLOCK_Q_PRIORITY );
    vStartSemaphoreTasks( mainSEM_TEST_PRIORITY );
    vStartPolledQueueTasks( mainQUEUE_POLL_PRIORITY );
    vStartIntegerMathTasks( mainINTEGER_TASK_PRIORITY );
    vStartGenericQueueTasks( mainGEN_QUEUE_TASK_PRIORITY );
    vStartQueuePeekTasks();
    vStartMathTasks( mainFLOP_TASK_PRIORITY );
    vStartRecursiveMutexTasks();
    vStartTimerDemoTask( mainTIMER_TEST_PERIOD );
    vStartCountingSemaphoreTasks();
    vStartDynamicPriorityTasks();

    /* The suicide tasks must be created last as they need to know how many
    tasks were running prior to their creation. This then allows them to
    ascertain whether or not the correct/expected number of tasks are running at
    any given time. */
    vCreateSuicidalTasks( mainCREATOR_TASK_PRIORITY );

    /* Create the semaphore that will be deleted in the idle task hook. This
    is done purely to test the use of vSemaphoreDelete(). */
    xMutexToDelete = xSemaphoreCreateMutex();

    /* Start the scheduler itself. */
```



The terminal window shows the output of the simulation, consisting of 25 lines of 'OK' messages followed by a task ID number, ranging from 1000 to 25000.

```
G:\Ordenadores_PCL\im
OK - 1000
OK - 2000
OK - 3000
OK - 4000
OK - 5000
OK - 6000
OK - 7000
OK - 8000
OK - 9000
OK - 10000
OK - 11000
OK - 12000
OK - 13000
OK - 14000
OK - 15000
OK - 16000
OK - 17000
OK - 18000
OK - 19000
OK - 20000
OK - 21000
OK - 22000
OK - 23000
OK - 24000
OK - 25000
```