INGENIERÍA DE SISTEMAS Y AUTOMÁTICA
INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

Universidad de Oviedo

Control por Computador

Implantación en equipos con Sistema Operativo

Ignacio Alvarez García

Septiembre - 2019

Indice

- Introducción al control de procesos por computador
- Implantación del control en el computador
- Programación bajo Sistema Operativo
- Programación orientada a objetos
- Programación orientada a eventos con Qt-C++

Indice

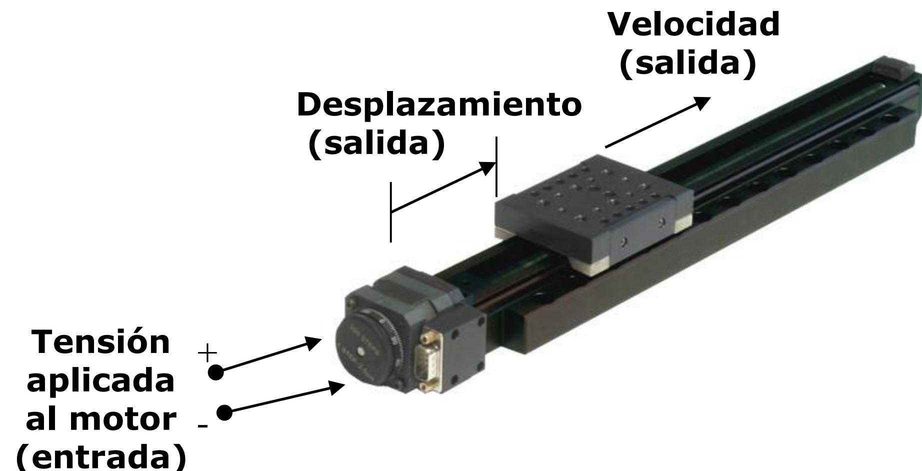
- **Introducción al control de procesos por computador**
- Implantación del control en el computador
- Programación bajo Sistema Operativo
- Programación orientada a objetos
- Programación orientada a eventos con Qt-C++

El Control de Procesos por Computador

- Proceso o sistema
  - Conjunto de elementos físicos que cumplen un cometido común
  - **Salidas del sistema**: valores medibles cuya variación en el tiempo se desea controlar
  - **Entradas al sistema**: acciones que se pueden realizar para modificar los valores de las salidas

Ejemplo 1: sistema térmico (calentador de fluido por gas)

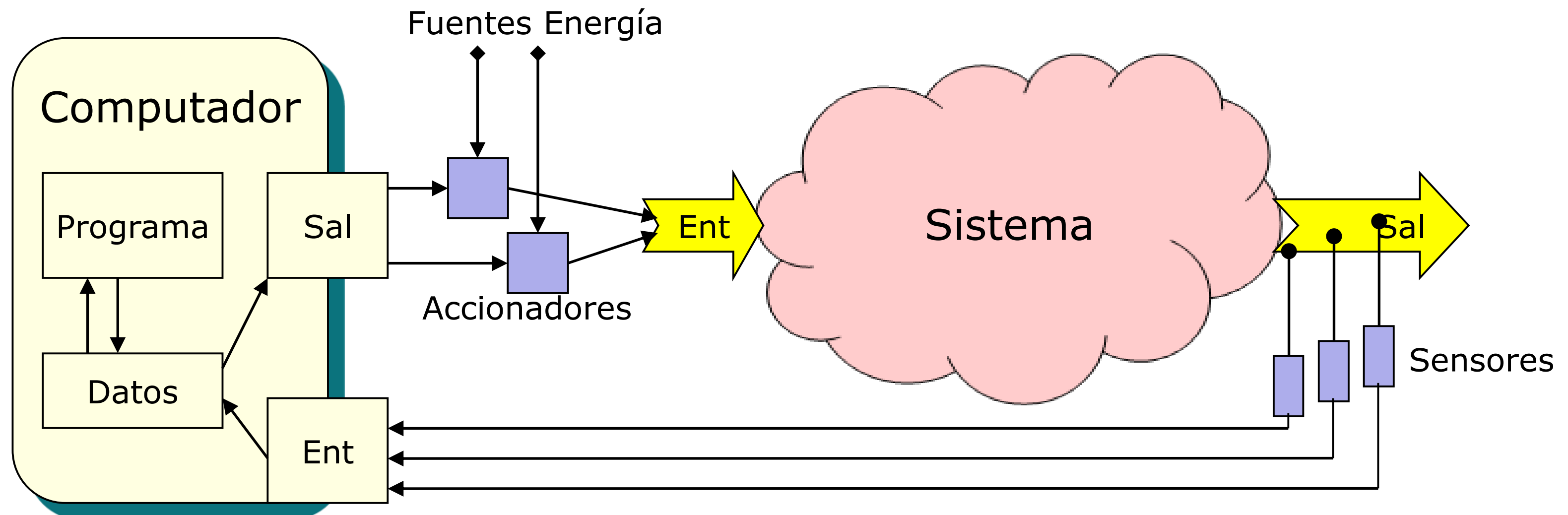
Ejemplo 2: sistema mecánico (motor CC + guía lineal)



El Control de Procesos por Computador

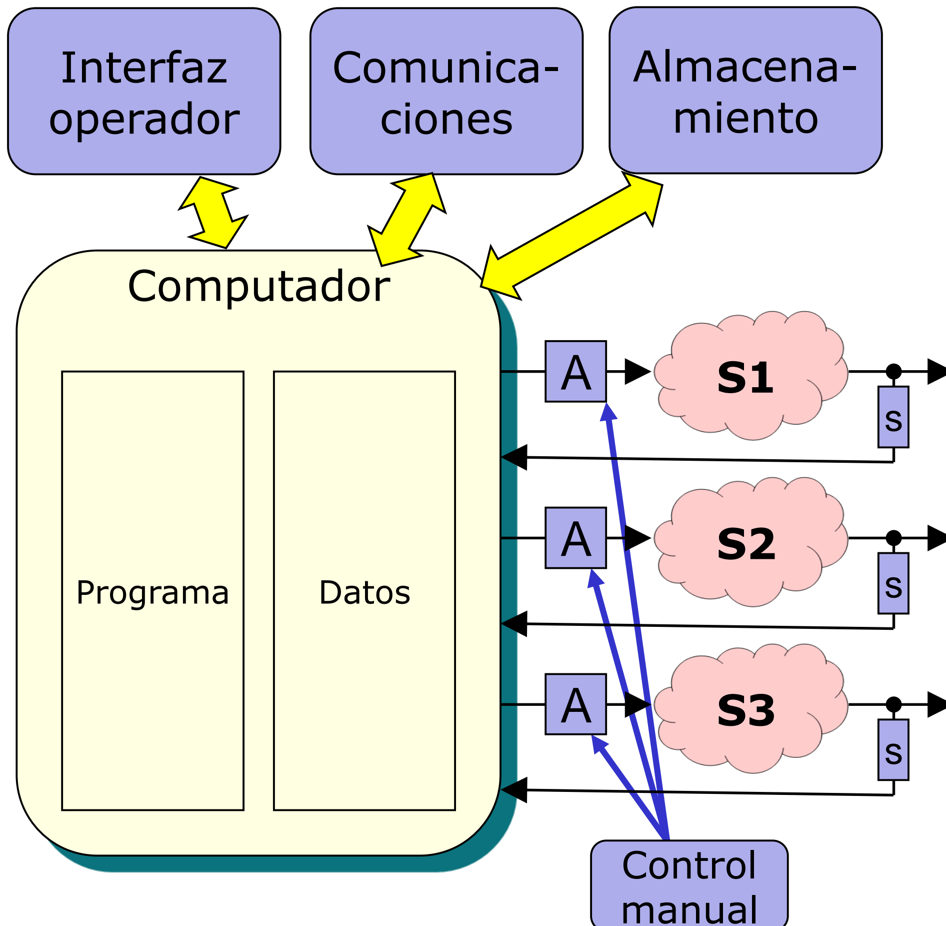
- Control de un proceso o sistema
  - Variables principales involucradas:
    - **Referencia o consigna**: valor deseado para la salida en cada momento
    - **Error**: Valor referencia – Valor salida
    - **Acción de control**: valor a aplicar en la entrada para conseguir (idealmente) error cero
  - Todas las variables evolucionan en el tiempo

- Control de proceso por computador
  - Un **Computador** lee las Salidas del sistema, realiza cálculos de control, y modifica las Entradas del sistema de acuerdo con estos cálculos
  - Operación en **Tiempo Real**: es importante que la acción de control sea correcta y realizada en tiempo



El Control de Procesos por Computador

- Elementos de un equipo de control



- **Computador:**
  - El "cerebro" del control
  - Programa y datos alojados en memoria
- **Sensores:**
  - Dan al computador información del estado de los sistemas, midiendo sus salidas.
- **Accionadores:**
  - Permiten al computador generar cambios sobre los sistemas, modulando la energía entregada por fuentes externas
- **Interfaz operador:**
  - Panel(es) que permite(n) al humano intervenir en el control, generando órdenes y recibiendo informaciones
- **Comunicaciones:**
  - Interconexión con otros computadores para intercambio de informaciones y comandos
- **Almacenamiento:**
  - De configuraciones y resultados
- **Control manual:**
  - Reemplaza las salidas del computador en caso de error o alarma

El Control de Procesos por Computador

- Resumen de las bases y programación del control de procesos:
  - Enlace a "Programación de control.pdf"
  - Enlace a "Programación en lenguaje C.pdf"

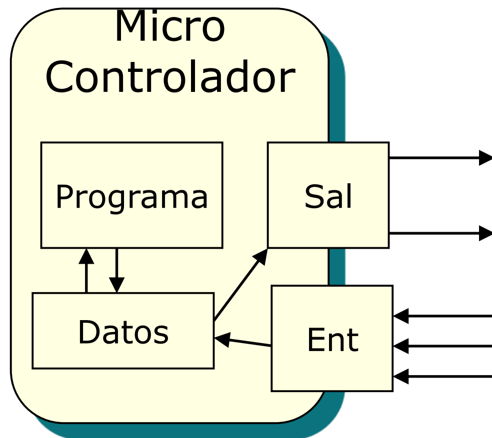
Sistemas complejos

- Características de los sistemas complejos:
  - Muchas entradas y salidas
  - Grandes cantidades de datos (imagen, sonido, etc.)
  - Aplicaciones divisibles en módulos “casi” independientes
  - Uso de dispositivos de E/S con procesamiento propio
  - Interfaz de operador con estilo gráfico
  - Comunicaciones avanzadas
  - Almacenamiento en bases de datos, archivos de configuración, log, etc.

- Computadores con S.O.
- Programación orientada a objetos
- Programación orientada a eventos

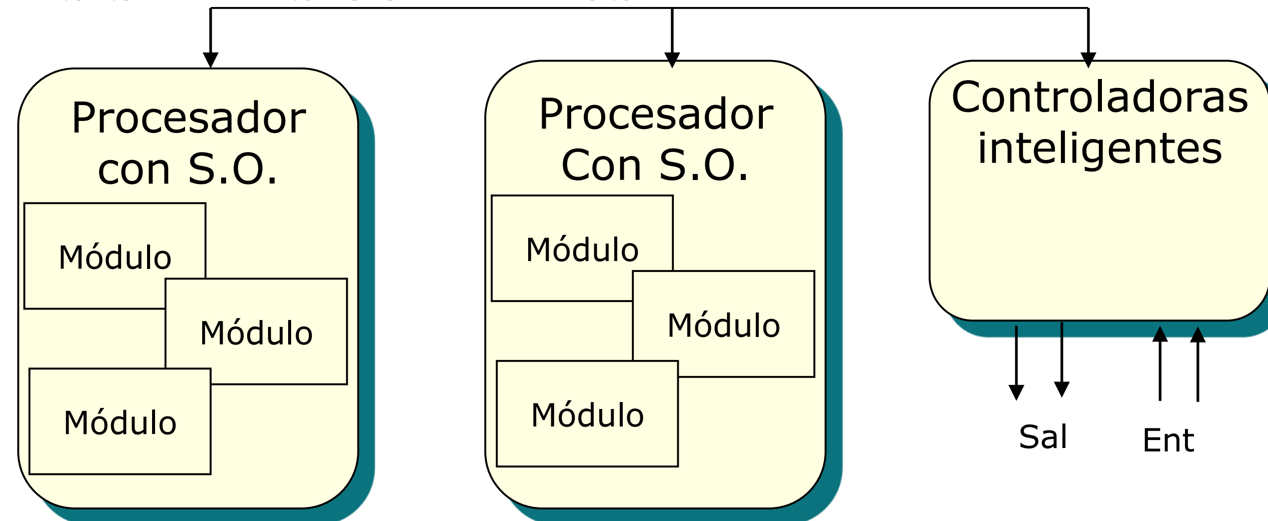
Sistemas complejos

DE SISTEMAS SENCILLOS:



- 1 solo programa gestiona todo el control
- Toda la "inteligencia" en un solo equipo
- Interfaz usuario sencillo modo texto
- Comunicaciones básicas
- Sin opciones para bb.dd. , interfaz gráfico, comunicaciones red, etc.

A SISTEMAS COMPLEJOS:



- Programación en múltiples módulos (clases, hilos, procesos)
- Inteligencia distribuida
- Interfaz usuario de tipo gráfico
- Comunicaciones avanzadas

Indice

- Introducción al control de procesos por computador
- **Implantación del control en el computador**
- Programación bajo Sistema Operativo
- Programación orientada a objetos
- Programación orientada a eventos con Qt-C++

Computadores para control

- Microcontrolador

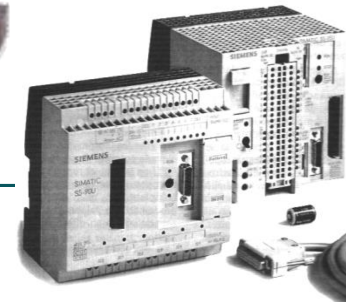
- Procesador Digital de Señal (DSP)

- Dispositivos Electrónicos Programables (FPGA, PLD)

- Computador embebido

- Ordenador Industrial

- Autómata Programable (PLC)



Universidad de Oviedo

Implantación del control

- Sistemas 'pequeños':
  - Sin soporte de Sistema Operativo
  - El programador accede directamente al hardware
    - Programación de puertos de E/S
    - Funciones para servicio de interrupciones
  - El programador debe realizar la planificación de la ejecución temporal de las diferentes tareas:
    - Round-robin: chequeo ordenado de las tareas a realizar en cada momento, sin uso de interrupciones.
    - Round-robin con interrupción: las tareas más prioritarias o puntuales son lanzadas por interrupciones hardware y servidas en Rutinas de Servicio de Interrupción (ISR).
    - Function-queue-scheduling: las ISR encolan las tareas a realizar con sus prioridades. Un planificador round-robin comprueba la cola y va sirviendo por orden de prioridad

Implantación del control

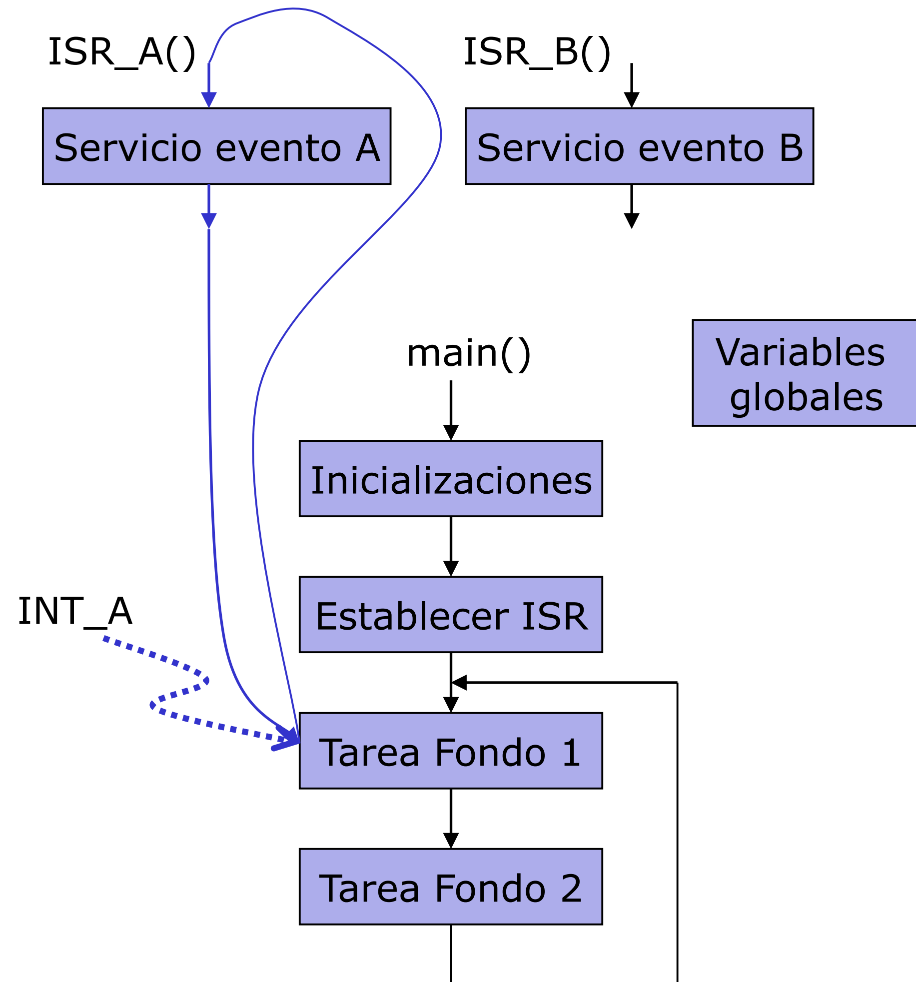
- Sistemas 'grandes':
  - Una tarea especial Sistema Operativo (S.O.) se encarga de dar servicio a:
    - Acceso al hardware (modo síncrono y asíncrono):
      - Nivel de S.O.
      - Nivel de driver
    - Planificación de tareas:
      - Cada vez que sucede un 'evento', el S.O. toma el control, actualiza estado de tareas, y pasa a ejecutar la más prioritaria.
    - Gestión de memoria
  - El programador debe realizar llamadas a funciones del S.O. para todos los servicios
  - El S.O. puede llamar a funciones del programador (callback) para rutinas de servicio asíncrono

Implantación del control

- Sistemas "pequeños" vs sistemas "grandes":
  - La tarea S.O. requiere recursos (memoria y tiempo de ejecución), que pueden no estar disponibles en sistemas pequeños.
  - La tarea S.O. requiere ejecutarse en modo privilegiado: la CPU debe disponer de 2 modos de funcionamiento (usuario y privilegiado).
  - La tarea S.O. facilita operaciones que pueden ser requeridas en sistemas grandes:
    - La programación de E/S (no hay que conocer todos los puertos e interrupciones)
    - La gestión de tareas (prioridades, sincronización, memoria)
    - Protecciones de seguridad basadas en privilegios y claves.
    - Gestión de comunicaciones, interfaz gráfico de usuario, almacenamiento, …

Sistemas “pequeños”

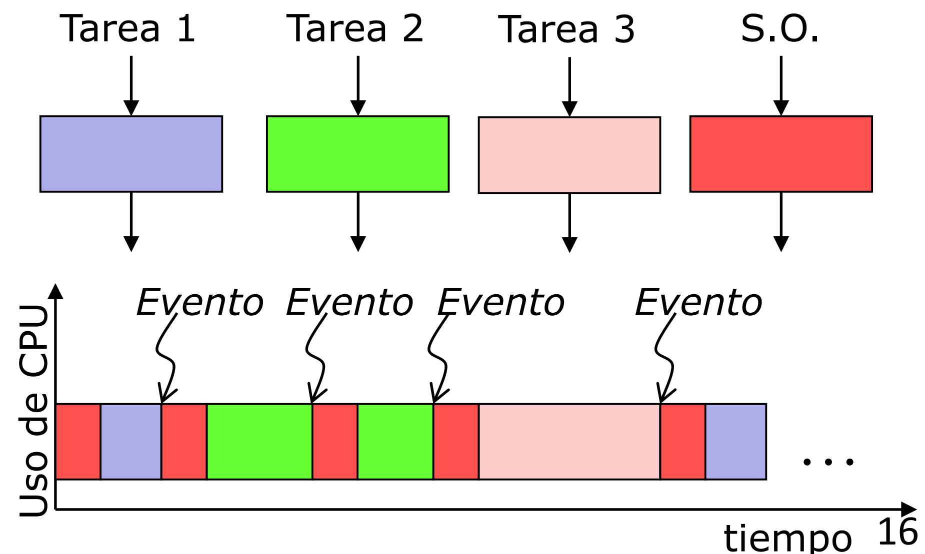
- Organización del programa:
  - Un bucle principal con la(s) tarea(s) de fondo.
  - Funciones de Servicio de Interrupción (ISR) para el servicio de eventos.
  - Las ISR detienen a la tarea de fondo o a otras ISR: ejecución rápida y sin esperas.
  - Gestión de prioridades de las ISR.
  - Las variables compartidas por el programa y las ISR deben ser globales.



Sistemas “grandes”

- El programador organiza su código en tareas, que se ejecutan “en paralelo” o, más exactamente, “en concurrencia”.
- Las tareas no siempre quieren ejecutar su código, están en muchas ocasiones en estado de espera por evento (pulsación teclado, vencimiento temporización, …)
- Una tarea especial, el Sistema Operativo, se encarga de:
  - Gestión de E/S mediante pequeños programas (drivers)
  - Gestión de eventos
  - Planificación de tareas
  - Gestión de memoria
  - …

Ante cualquier evento:
- El S.O.toma el control (la CPU pasa a ejecutar su código).
- El evento puede provocar el cambio de estado de alguna tarea.
- El S.O. revisa las tareas pendientes y cede el control a la más prioritaria que necesite ejecutarse, hasta el siguiente evento.



Sistemas “grandes”

- Variedad de Sistemas Operativos:
  - Windows, Unix, Linux, Mac OS, Android, VX-Works, …
- Tipos de Sistemas Operativos:
  - De tiempo compartido:
    - El retardo en ejecutar una tarea no es importante.
    - El S.O. distribuye tiempos de ejecución entre todas las tareas.
  - De tiempo real:
    - El retardo en ejecutar una tarea puede ser muy importante.
    - El S.O. ejecuta siempre la tarea más prioritaria.
- POSIX: Portable Operating System Interface:
  - Basado en Unix
  - Mismo interfaz con las tareas (system calls) para diferentes Sistemas Operativos.

Indice

- Introducción al control de procesos por computador
- Implantación del control en el computador
- **Programación bajo Sistema Operativo**
- Programación orientada a objetos
- Programación orientada a eventos con Qt-C++

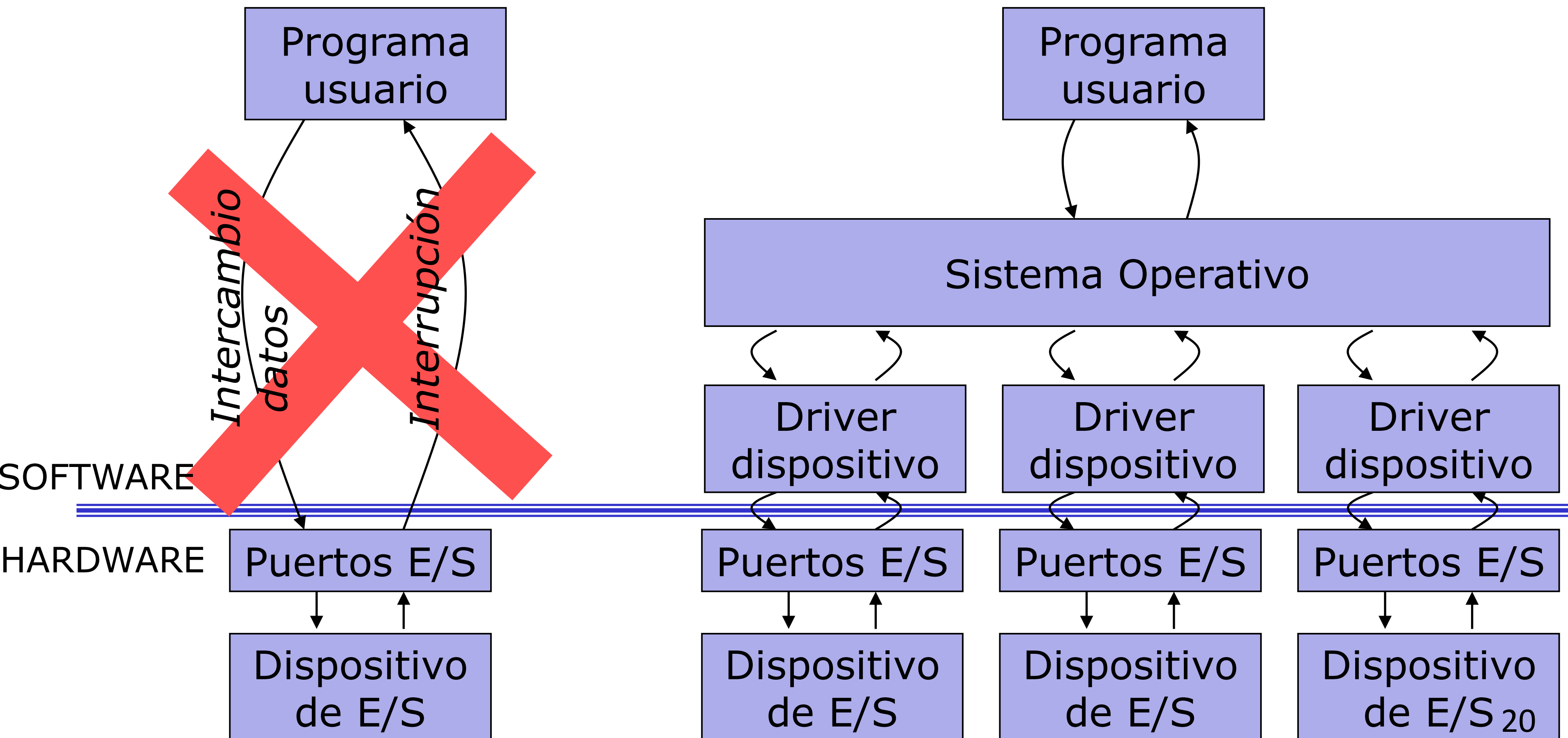
Programación bajo Sistema Operativo

- El programador organiza su código en tareas, que se ejecutan "en paralelo" o, más exactamente, "en concurrencia".
- Una tarea especial, el Sistema Operativo, se encarga de:
  - Gestión de dispositivos de E/S mediante pequeños programas (drivers)
  - Gestión de eventos y temporizaciones
  - Planificación de tareas
  - Gestión de memoria
  - Gestión de sistemas de archivos
  - Seguridad
  - Comunicaciones
  - Interfaz gráfico de usuario (GUI)
- Los programas de usuario solicitan servicios al Sistema Operativo llamando a funciones de éste

Universidad de Oviedo

Gestión de E/S con S.O.

- El programador interacciona con los dispositivos de E/S a través del S.O. y drivers de dispositivo.



Universidad de Oviedo

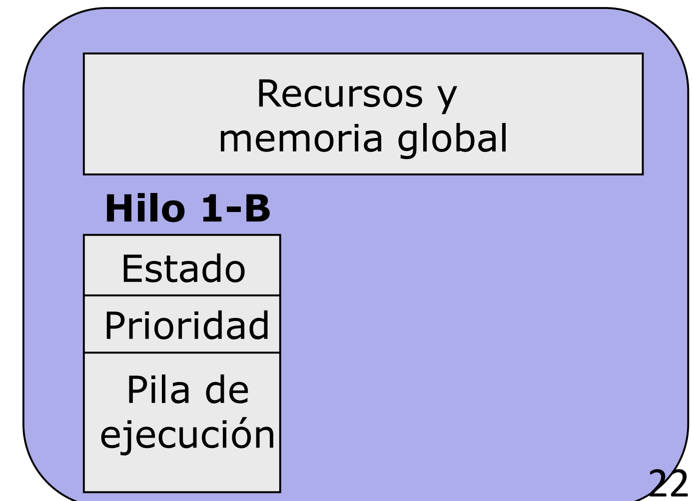
INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

Gestión de eventos y tareas

- Los drivers sirven las interrupciones de los dispositivos de E/S, y generan eventos al S.O.
- Las tareas se ejecutan concurrentemente, esto es, en competencia por usar un recurso único: la CPU.
- Las tareas indican al S.O. si necesitan ejecutar código o están pendientes de algún evento.
- Las tareas indican al S.O. su prioridad de ejecución frente a otras tareas.
- Las tareas necesitan sincronizarse entre sí, para generar evento o utilizar recursos compartidos.
- Ante cualquier evento del driver o solicitud de una tarea:
  - El S.O. actualiza el estado de la(s) tarea(s).
  - El S.O. revisa la(s) tarea(s) en estado de necesidad de ejecución.
  - El S.O. cede la ejecución a una de esas tareas según sus criterios internos.

Gestión de tareas

- Las tareas se clasifican en **procesos** e **hilos** (threads).
- Procesos:
  - El S.O. ejecuta los procesos en espacios de memoria separados (memoria virtual), y protege sus recursos de otros procesos.
  - Los procesos sólo pueden comunicarse a través del S.O.
  - Cada proceso tiene al menos un hilo.
- Hilos:
  - Los hilos son subprocesos dentro de un proceso.
  - Cada hilo tiene su propia pila de ejecucion, prioridad y estado.
  - Los hilos de un proceso comparten todos los recursos, excepto su pila.
  - Los hilos de un proceso pueden comunicarse mediante las variables globales.

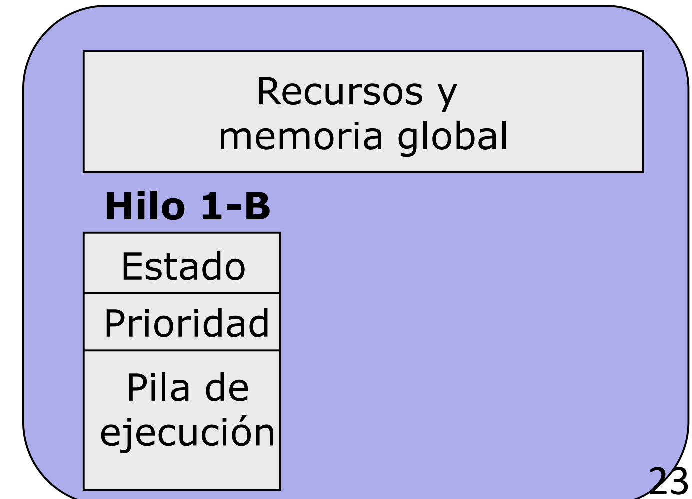


Gestión de tareas

- El Sistema Operativo se encarga de:
  - Determinar qué proceso e hilo se debe ejecutar en cada momento.
  - Guardar el contexto de cada proceso e hilo cuando no se ejecuta.
  - Recuperar el contexto del proceso e hilo que pasa a ejecutarse.
- Contexto de un hilo:
  - Valor de los registros en el momento de su suspensión (PC, SP, etc.).
- Contexto de un proceso:
  - Identificadores de streams abiertos.
  - Asignaciones de memoria virtual.
  - Derechos de ejecución y acceso.

Proceso A

Proceso B



Programacion orientada a eventos y tareas

- Múltiples opciones para elegir: lenguajes, entornos, sistemas operativos, etc.
- Nuestra elección:
  - Entorno de programación: Qt-Creator (enlace instalación y uso)
  - Lenguaje de programación: C++ (enlace introducción)
  - Librería de programación: Qt-SDK
    - Facilita la programación orientada a eventos (enlace documentación)
    - Facilita el desarrollo de interfaz de usuario gráfico (GUI) (enlace documentación)

Indice

- Introducción al control de procesos por computador
- Implantación del control en el computador
- Programación bajo Sistema Operativo
- **Programación orientada a objetos**
- Programación orientada a eventos con Qt-C++

Programacion orientada a objetos

- Terminología:
  - **Clase**: definición de características y funcionalidades de los miembros que pertenecen a ella
  - **Objeto**: instancia (variable específica) de una clase
  - **Herencia**: una clase derivada contiene todos los atributos de una clase base y le añade/modifica algunos
  - Ejemplo:
    - **Clase**: mamífero
      - Características: peso, tipo de alimentación, …
      - Funcionalidades: comer, moverse, ...
    - **Objetos** tipo mamífero: mi_perro, mi_gato, yo … (instancias de esa clase)
    - **Clase**: humano , deriva de mamífero y además añade:
      - Características: idioma, …
      - Funcionalidades: hablar, …
    - **Objetos** tipo humano: tú, yo (instancias de esa clase)

Programacion orientada a objetos

- La Programación Orientada a Objetos (OOP) facilita la gestión de sistemas de complejidad media/alta:
  - Las clases encapsulan la programación del comportamiento completo de ciertas entidades (motor, RZ, señal temporal, gráfico, …) aislándolo del resto del programa.
  - Las clases permiten una interacción limitada desde otros objetos, reduciendo la posibilidad de errores.
  - Las clases pueden ser desarrolladas y probadas más fácilmente por separado, favoreciendo el trabajo en equipo.
  - Las clases pueden realizarse de forma incremental, utilizando la herencia.
  - Existen numerosas librerías de clases ya realizadas que facilitan el desarrollo del código.
- Más info: capítulos 1 a 4 de:

  http://ee402.eeng.dcu.ie/introduction/chapter-1---introduction-to-object-oriented-programming

Programacion orientada a objetos

- Programación orientada a funciones (procedural)
  1) El programador declara las variables necesarias
  2) El programador determina las operaciones (algoritmos) requeridas sobre las variables para producer el resultado desado.
  3) Si el programador detecta partes de código “repetidas” (se aplican los mismos cálculos sobre variables diferentes) las encapsula en funciones. Las funciones facilitan la reutilización del código y la reducción de errores, independizando algunos algoritmos del programa para el que son usados.
  4) Si el programador detecta conjuntos de variables “repetidas” las encapsula en estructuras.
  5) Las estructuras y las funciones se gestionan aparte unas de otras.

Universidad de Oviedo

Programacion orientada a objetos

- Ejemplo: gestión de datos de dos señales temporales
  - Aproximación inicial:
  - Con funciones:

```
int main()
{
   float* s1_k,*s2_k;
   int n1=4,n2=8;
   s1_k =(float*) malloc(n1*sizeof(float));
   for (int i=0;i<n1;i++)
       s1_k[i]=0;
   s2_k =(float*) malloc(n2*sizeof(float));
   for (int i=0;i<n2;i++)
       s2_k[i]=0;
   ...
   int keep_going=1;
   while (keep_going)
   {
       for (int i=n1-1;i>0;i--)
          s1_k[i]=s1_k[i-1];
       for (int i=n2-1;i>0;i--)
          s2_k[i]=s2_k[i-1];

       s1_k[0] =  ... New value ... ;
       s2_k[0] =  ... New value ... ;
       ... use s1_k and s2_k ...
   }
   free(s1_k);
   free(s2_k);
   return 0;
}
```

```
float* AsignaMemo(int n)
{
    float* pt;
    pt=(float*) malloc(n*sizeof(float));
    return pt;
}
void IniciaTabla(float* t,int n,float v_init)
{    ...  }
void DesplazaTabla(float* t,int n,float new_t0)
{   ...   }

int main()
{
   float* s1_k,*s2_k;
   int n1=4,n2=8;

   s1_k=AsignaMemo(n1);   IniciaTabla(s1_k,n1,0);
   s2_k=AsignaMemo(n2);   IniciaTabla(2_k,n2,0);
    ...
   int keep_going=1;
   while (keep_going)
   {
       DesplazaTabla(s1_k,n1,new_value_s1);
       DesplazaTabla(s2_k,n2,new_value_s2);
       ... use s1_k and s2_k ...
   }
   free(s1_k);
   free(s2_k);
   return 0;
}
```

Programacion orientada a objetos

- Ejemplo: gestión de datos de dos señales temporales
  - Con funciones:

```
float* AsignaMemo(int n)
{
    float* pt;
    pt=(float*) malloc(n*sizeof(float));
    return pt;
}
void IniciaTabla(float* t,int n,float v_init)
{    ...  }
void DesplazaTabla(float* t,int n,float new_t0)
{   ...    }

int main()
{
   float* s1_k,*s2_k;
   int n1=4,n2=8;

   s1_k =AsignaMemo(n1);   IniciaTabla(s1_k,n1,0);
   s2_k =AsignaMemo(n2);   IniciaTabla(s2_k,n2,0);
    ...
   int keep_going=1;
   while (keep_going)
   {
       DesplazaTabla(s1_k,n1,new_value_s1);
       DesplazaTabla(s2_k,n2,new_value_s2);
       ... use s1_k and s2_k ...
   }
   free(s1_k);
   free(s2_k);
   return 0;
}
```

  - Con funciones y estructuras:

```
struct vector {
  float* data;
  int n;
} ;

struct vector AsignaVector(int n)
{
    struct vector v;
    v.data=(float*) malloc(n*sizeof(float));
    v.n=n;
    return v;
}
void IniciaVector(struct vector* ptV,float v_init)
{    ...  }
void DesplazaVector(struct vector* ptV,float new_t0)
{   ...    }

int main()
{
   struct vector s1_k,s2_k;

   s1_k =AsignaVector(4);   IniciaVector(s1_k,0);
   s2_k =AsignaVector(8);   IniciaVector(s2_k,0);
    ...
   int keep_going=1;
   while (keep_going)
   {
       DesplazaVector(s1_k,new_value_s1);
       DesplazaTabla(s2_k,new_value_s2);
       ... use s1_k and s2_k ...
   }
   free(s1_k);
```

Programacion orientada a objetos

- Programación orientada a objetos (OOP)
  1) El programador piensa en clases, que agrupan las variables necesarias para un tipo de objetos y las funciones que las manipulan
  2) El programador "encapsula" los contenidos de la clase: parte pública (accesible a todos) y parte privada (accesible sólo para las funciones de la clase)
  3) El resto del programa sólo necesita conocer y puede usar variables y funciones públicas de la clase.
  4) Se declaran variables (objetos) con el tipo de la clase. Como en las estructuras, se accede a sus contenidos con el operador punto.
  5) Si el programador detecta que una clase puede "heredar" el contenido de otra, añadiendo o modificando algunos de sus elementos constituyentes, utiliza la derivación.

Programacion orientada a objetos

- Ejemplo: gestión de datos de una señal temporal
  - Sin clases:

```
struct vector { ... } ;

struct vector AsignaVector(int n)
{    ...   }
void IniciaVector(struct vector* ptV,float v_init)
{     ...   }
void DesplazaVector(struct vector* ptV,float new_t0)
{    ...    }

int main()
{
   struct vector s1_k,s2_k;
   s1_k =AsignaVector(4);   IniciaVector(s1_k,0);
   s2_k =AsignaVector(8);   IniciaVector(s2_k,0);

   int keep_going=1;
   while (keep_going)
   {
       DesplazaVector(s1_k,new_value_s1);
       DesplazaTabla(s2_k,new_value_s2);
       ... use s1_k and s2_k ...
   }
   free(s1_k);
   free(s2_k);
   return 0;
}
```

El programador de main() se hace cargo de todo:
- Más código en main()
- Más propenso a errores
- Más difícil de compartir para co-desarrollo

  - Con clases:

```
class Signal
{
private:
    float* data;
    int n;
public:
    Signal(int i_n,float v_init);
    ~Signal();
    void Desplaza(float t0);
};
Signal::Signal(int i_n,float v_init) { ... }
Signal::~Signal() { ... }
void Signal::Desplaza(float t0) { ... }

main()
{
   Signal s1_k(4,0.0),s2_k(8,0.0);

   int keep_going=1;
   while (keep_going)
   {
       s1_k.Desplaza(new_value_s1);
       s2_k.Desplaza(new_value_s2);
       ... use s1_k and s2_k ...
   }
return 0;
```

El programador de Signal se hace cargo de gran parte:
- main() más sencillo
- Limitación de errores
- Más fácil para co-desarrollo de proyectos grandes

Programacion orientada a objetos

- Ejemplo: reducción de posibilidades de error
  - Sin clases:

```
float* AsignaMemo(int n);
void IniciaTabla(float* t,int n,float v_init);
void DesplazaTabla(float* t,int n,float new_t0);

main()
{
   float* my_signal_k;
   int n=4;
   my_signal_k=AsignaMemo(n);
   IniciaTabla(my_signal_k,n,0.0);

   // En cada Tm ...
   nuevo_valor = ...;
   DesplazaTabla(my_signal_k,n+2,nuevo_valor);

   // Usar tabla my_signal_k
   ...

   // Al terminar
   free(my_signal_k);
}
```

Ejemplo de error: usar n+2 en lugar de n provoca un acceso incorrecto a la tabla y una corrupción de la pila

  - Con clases:

```
class Signal
{
private:
    float* t;
    int n;
public:
    Signal(int i_n,float v_init);
    ~Signal();
    void Desplaza(float t0);
};

main()
{
   Signal my_signal_k(4,0.0);

   // En cada Tm ...
   nuevo_valor = ...;
   my_signal_k.Desplaza(nuevo_valor);
// Usar clase my_signal_k
   ...
}
```

Si la clase está bien realizada, el programador de main() no puede salirse de la tabla ya que no puede modificar n (está embebido en la clase y no es public)

Programacion orientada a objetos

- Ejemplo: facilidad de uso y reducción de código
  - Sin clases:

```
float* AsignaMemo(int n);
void IniciaTabla(float* t,int n,float v);
void DesplazaTabla(float* t,int n,float new_t0);

main()
{
   float *curpos_k,*targetpos_k,*curspeed_k;
   int n1=4,n2=7,n3=2;

   curpos_k=AsignaMemo(n1);
   IniciaTabla(curpos_k,n1,0.0);
   targetpos_k=AsignaMemo(n2);
   IniciaTabla(targetpos_k,n2,0.0);
   curspeed_k=AsignaMemo(n3);
   IniciaTabla(curspeed_k,n3,0.0);
   ...
   DesplazaTabla(curpos_k,n1,nuevo_valor1);
   DesplazaTabla(targetpos_k,n2,nuevo_valor2);
   DesplazaTabla(curspeed_k,n3,nuevo_valor3);
   ...
   free(curpos_k);
   ...
}
```

Para tres señales: se repite el código 3 veces, con las posibilidades de error aumentadas (copy/paste)

  - Con clases:

```
class Signal
{
   ...
};

class MotorSignalSet
{
private:
   Signal curpos_k,targetpos_k,curspeed_k;
public:
    SignalSet(int n_curpos,int n_targetpos,
              int n_curspeed);
    ~SignalSet();
    void Desplaza(float new_curpos,float
new_targetpos,float new_curspeed);
};

main()
{
   MotorSignalSet my_motor(4,7,2);
   ...
   my_motor.Desplaza(new_v1,new_v2,new_v3);
   ...
}
```

Para tres señales: se puede hacer una clase que use a la anterior para facilitar el trabajo de main() y asegurar que todo se realiza correctamente

Programacion orientada a objetos

- Ejemplo: uso de clases ya existentes
  - Sin clases:

```
float* AsignaMemo(int n)
{
   return (float*) malloc(n*sizeof(float));
}

void IniciaTabla(float* t,int n,float v)
{
   int i;
   for (i=0;i<n;i++)
      t[i]=v;
}

void DesplazaTabla(float* t,int n,float new_value)
{
   int i;
   for (i=n-1;i>0;i--)
      t[i]=t[i-1];
   t[0]=new_value;
}
```

El código de las funciones debe en muchos casos ser realizado por el programador

  - Con clases:

```
#include <QVector>

class Signal
{
private:
    QVector<float> data;
...
} ;

Signal::Signal(int i_n,float v) : data(i_n,v)
{
}

void Signal::Desplaza(float t0)
{
   data.removeLast();
   data.prepend(t0);
}
```

La clase QVector ya está realizada y contiene las funcionalidades habituales para quien use un vector

Programacion orientada a objetos

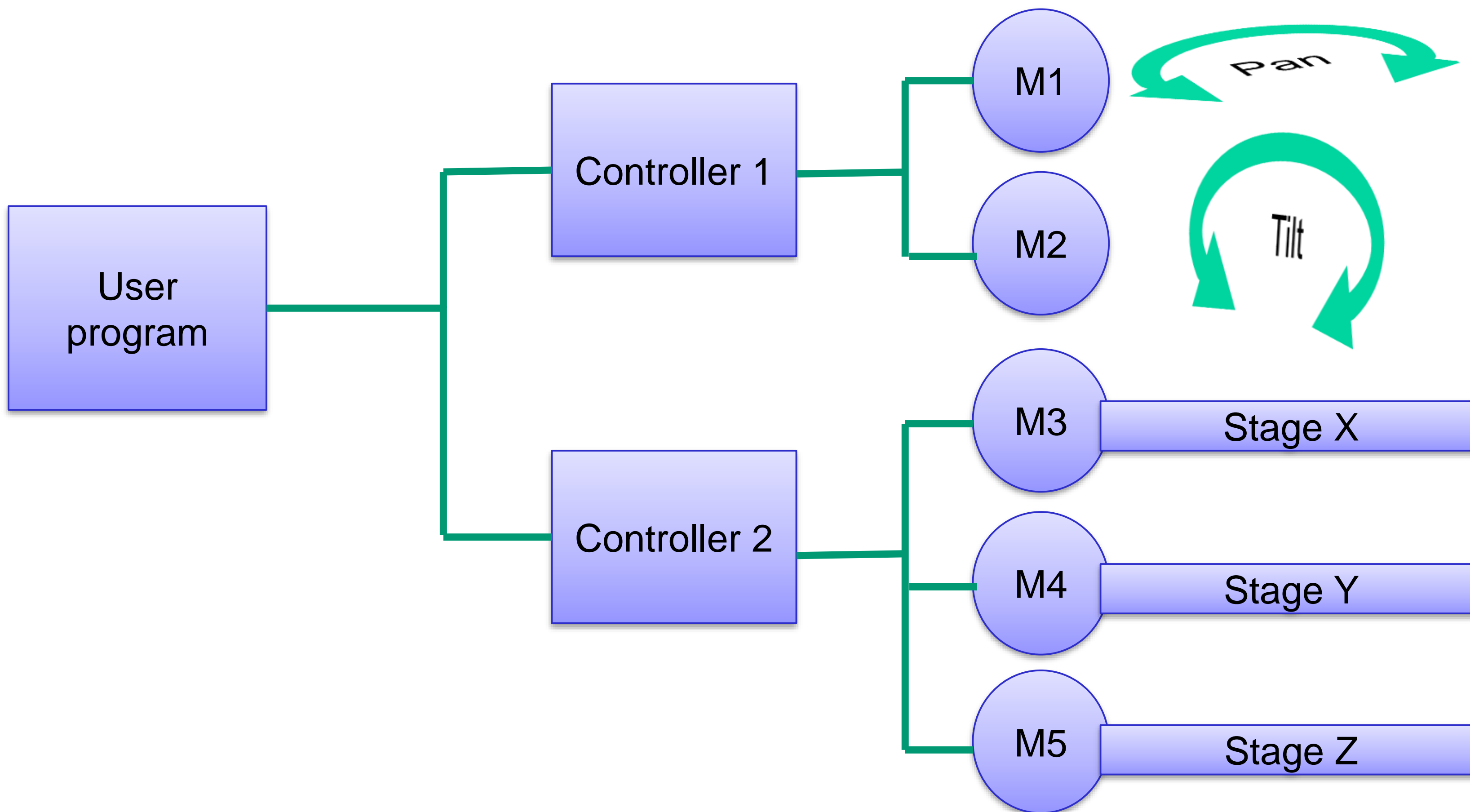
- Diseño de clases para un proyecto:
  - Ver http://ee402.eeng.dcu.ie/introduction/chapter-1---introduction-to-object-oriented-programming#TOC-Object-Oriented-Analysis-and-Design
  - Importante a tener en cuenta:
    - Buscar clases ya disponibles en librerías comerciales / opensource (librerías C++ standard, Qt-SDK, boost, opencv, …)
    - Pensar más allá de nuestro proyecto actual (organizar de clases para su reutilización)
    - Diseñar primero la parte pública (en proyectos grupales, acordarla entre el desarrollador de cada clase y sus usuarios)
    - No pretender desarrollar todo en una clase: utilizar inteligentemente la jerarquía

Programacion orientada a objetos

- Diseño de clases para un proyecto:
  - Ejemplo: clases para gestión de accionamientos → diversidad
    - Múltiples tipos de motores (DC, AC, …)
    - Una controladora comercial puede gestionar varios motores
    - Múltiples formas de conocer la posición (encoder, potenciómetro, …)
    - Acoplamientos variados (traslación, giro, reductora, …)
    - Relaciones diferentes entre rpm y movimiento deseado
    - Sensores auxiliaries (final de Carrera, …)
  - Pero lo que queremos hacer con un accionamiento es habitualmente lo mismo:
    - Configurar
    - Mover a posición deseada con perfil de velocidad predefinido
    - Mover a velocidad constante
    - Parar
    - Saber su posición y/o velocidad
    - Que nos avise cuando llegado a su destino

Programacion orientada a objetos

- Diseño de clases para un proyecto:
  - Ejemplo: clases para gestión de motores



Programacion orientada a objetos

- Diseño de clases para un proyecto:
  - Ejemplo: clases para gestión de motores

```
class Controller
{
private:
   QString name;
   QVector<Motor*> motors;
public:
   MotorController(const QString& xml_init);
   int AddMotor(const QString& name,Motor* m);
   bool SetMotorParams(int motorIndex,const QString& xml_init);
   bool MoveToPos(int motorIndex,float pos,bool is_abs_pos);
   float GetPos(int motorIndex);
   ...
} ;
```

```
class Motor
{
private:
   float target_pos,current_pos;
   int current_state;
   QString name;
public:
   Motor(const QString& name,int type);
   bool SetParams(const QString& xml_init);
   bool SetSpeedForMovement(float speed);
   bool HasLeftLimitSwitch();
   bool GotoLeftLimitSwitch();
...
} ;
```

```
class NanotecController : public Controller
{
private:
   QSerialPort comm;
public:
    NanotecController QString& xml_init);
    int AddMotor(const QString& name,Motor* m);
    ...
} ;
```

```
class NanotecMotor : public Motor
{
private:
   int rs485Id;
public:
   NanotecMotor (const QString& xml_init);
protected:
   virtual bool SetParamsOverloaded(const QString& xml_init);
} ;
```

Indice

- Introducción al control de procesos por computador
- Implantación del control en el computador
- Programación bajo Sistema Operativo
- Programación orientada a objetos
- **Programación orientada a eventos con Qt-C++**

Programacion orientada a eventos con Qt/C++

- Librería Qt-SDK:
  - Proporciona clases para el manejo de los datos más comunes en programación
  - Proporciona clases para la gestión de eventos
  - Proporciona clases para realizar GUI
  - …
  - Integrada con entorno de desarrollo Qt-Creator
- Y además es multi-plataforma:
  - El mismo código se puede compilar para Windows y Linux sobre PC, Android sobre tfno móvil o tablet, Linux sobre sistemas embebidos (Raspberry Pi, Beaglebone, etc.), Sistemas en Chip SOC (Xilinx Zynq, …)

Programacion orientada a eventos con Qt/C++

- Librería Qt-SDK:
  - Proporciona clases para el manejo de los datos más comunes en programación:
    - **QString, QByteArray, QStringList** : gestión de cadenas de caracteres
    - **QVector, QList, QMap**: gestion de vectores y listas
    - **QTime, QDate, QDateTime, QElapsedTimer** : gestión de fecha/hora, temporizaciones, etc.
    - **QColor** : gestión de colores
    - **QFile, QDir, QFileInfo**: gestión de archivos y directories
    - **QSize(F), QPoint(F), QRect(F)**: geometría en 2D

Programacion orientada a eventos con Qt/C++

- Librería Qt-SDK:
  - Proporciona clases para la organización de programas orientados a eventos mediante el mecanismo signal/slot:
    - **QCoreApplication** : gestion de aplicación
    - **QTimer**: temporizaciones por evento
    - **Clases de usuario**: cualquier clase derivada de **QObject** puede hacer uso del sistema de eventos

Programacion orientada a eventos con Qt/C++

- Librería Qt-SDK:
  - Proporciona clases para la creación de interfaz gráfico de usuario (GUI) mediante el mecanismo signal/slot:
    - **QWidget** : elemento o conjunto de elementos de interfaz
    - **QDialog**: diálogo de interacción con el usuario
    - **QMenu**: menu de selección
    - **QButton, QSlider, QSpinBox, QDoubleSpinBox, QComboBox** … : multiples elementos de interfaz
    - **Clases de usuario**: se pueden derivar de cualquiera de las anteriores
  - Integrada con Qt-Creator mediante el Form Editor

Programacion orientada a eventos con Qt/C++

- Librería Qt-SDK:
  - Proporciona clases para comunicaciones:
    - **QSerialPort** : comunicaciones por puerto serie
    - **QUdpSocket, QTcpSocket, QTcpServer**: comunicación por red
    - …
  - Proporciona clases para gestión de tareas (procesos e hilos):
    - **QThread, QProcess, QMutex, QSemaphore, …**