**Universidad del Cauca**
**Departamento de Telemática**

Sistemas de Tiempo Real

# Sincronización y Comunicación

Dr. Ing. Álvaro Rendón Gallón
Popayán, abril de 2011



Departamento de Telemática

## Temario

- Introducción
- Señales
- Acceso a recursos compartidos
  - Exclusión mutua
  - Semáforos
  - Monitores y Variables condicionales
  - Mecanismos POSIX
- Paso de mensajes
  - Colas de mensajes

# Introducción

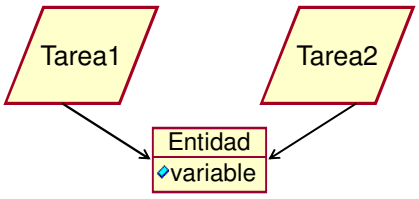
- Los sistemas de tiempo real son inherentemente concurrentes
  - "Programación concurrente es el nombre dado a las técnicas y notaciones de programación usadas para expresar el paralelismo potencial y resolver los problemas de sincronización y comunicación resultantes" (Ben-Ari, 1982)
- Su correcto funcionamiento depende en forma crítica de la sincronización y la comunicación entre las tareas
  - Sincronización: Cumplimiento de restricciones en la intercalación de las operaciones de los procesos.
  - Comunicación: Paso de información entre procesos.

  Los dos conceptos están ligados

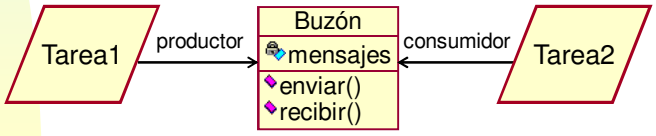


# Introducción

- Dos esquemas básicos de comunicación (que requieren sincronización):
  - Variables compartidas:




  - Paso de mensajes:



Departamento de Telemática

# Introducción

- Mecanismos de sincronización/comunicación
  - Señales
- Mecanismos de sincronización para compartir variables
  - Semáforos
  - Exclusión mutua *(mutex)*
  - Variables condicionales
- Mecanismos de paso de mensajes
  - Colas de mensajes



Departamento de Telemática

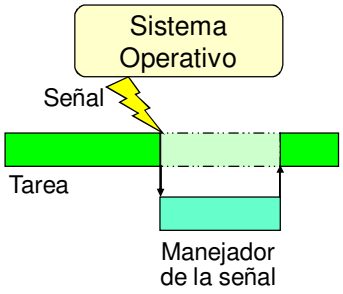
# Temario

- Introducción
- **Señales**
- Acceso a recursos compartidos
  - Exclusión mutua
  - Semáforos
  - Monitores y Variables condicionales
  - Mecanismos POSIX
- Paso de mensajes
  - Colas de mensajes

Departamento de Telemática

# Señales

- Mecanismo de comunicación asíncrono
- Equivalentes a las interrupciones físicas
- Producidas por el Sistema Operativo






Departamento de Telemática

# Señales

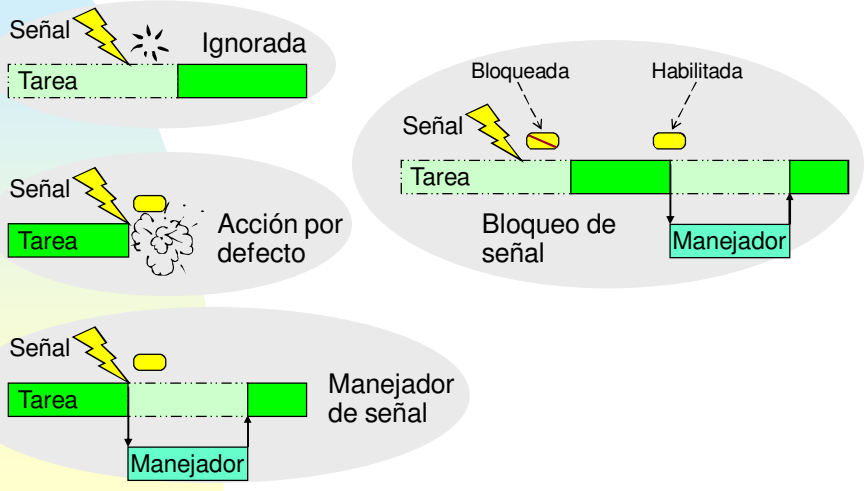
- Sirven para muchos propósitos
  - Manejo de excepciones (división por cero, violación de memoria, etc.)
  - Notificación de eventos asíncronos (E/S completa, temporización expirada, etc.)
  - Terminación de una tarea en circunstancias anormales
  - Emulación de multitarea
  - Comunicación explícita entre procesos: kill()
- Toda aplicación multitarea debe manejarlas adecuadamente

# Recepción de señales

- Una señal no siempre se atiende justo después de su envío
  - La tarea receptora puede estar suspendida
    - Atiende la señal al activarse
  - Las señales asociadas a interrupciones físicas sí son atendidas de inmediato
- Una tarea puede definir *a priori* cómo manejar cada señal que recibe:
  - Ignorarla
  - Ejecutar la acción por defecto asociada a la señal: Destruir, suspender o activar la tarea
  - Manejar la señal: Define una función para atender la señal
  - Bloquear la señal: Difiere su atención hasta cuando la desbloquea



# Recepción de señales



# Identificación de señales

Departamento de Telemática

- Cada tipo de señal tiene asignado un identificador
- A cada identificador le corresponde un número entero

| Identificador | Acción por defecto | Usada para |
|---|---|---|
| SIGABRT | Destrucción | Terminación anormal, aborto |
| SIGALRM | Destrucción | Expiración de la alarma del reloj |
| SIGFPE | Destrucción | Excepción de coma flotante |
| SIGILL | Destrucción | Excepción de instrucción ilegal |
| SIGINT | Destrucción | Destrucción interactiva (Ctrl-C) |
| SIGKILL | Destrucción | Destrucción no deshabilitable |
| ... | | |
| SIGUSR1 | Destrucción | Aplicaciones de usuario |
| SIGUSR2 | Destrucción | Aplicaciones de usuario |
| SIGCHLD | Ignorada | Destrucción o detención de hijo |
| SIGSTOP | Suspensión | Detención del proceso |
| SIGCONT | Activación | Continuación del proceso |
| ... | | |

POSIX 1003.1

Requeridas

Opcionales

- Sólo dos disponibles para las aplicaciones de usuario



# Señales en POSIX 1003.1

Departamento de Telemática

- Envío de señales
  - Enviar una señal a un proceso o grupo de procesos

  int **kill**(pid_t pid, int señal)

  pid >0: Proceso con ID de proceso pid
  0: Procesos con igual ID de grupo y permisos
  -1: Procesos con permisos
  <-1: Procesos con ID de grupo de procesos |pid| y permisos

  señal: señal. Si señal=0 (null), prueba (no se envía)

  retorna: 0: éxito, -1: error

  Los ID de usuario del proceso emisor debe coincidir con el del proceso receptor

# Señales en POSIX 1003.1

Departamento de Telemática

– Cambiar el manejador de una señal

int **sigaction**(int señal, const struct sigaction *accion, struct sigaction *accionant)

señal: Señal

accion: Acción para la señal

- sa_handler= SA_IGN: Ignora la señal
- sa_handler= SA_DFL: Acción por defecto

Si accion= NULL, sólo se obtiene la máscara actual

accionant: Acción anterior

Si accionant= NULL, no se guarda la máscara anterior



# Señales en POSIX 1003.1

Departamento de Telemática

- Manipulación de conjuntos de señales

int **sigemptyset**(sigset_t *conj)

int **sigfillset**(sigset_t *conj)

int **sigaddset**(sigset_t *conj, int señal)

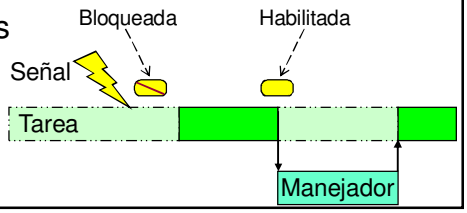
int **sigdelset**(sigset_t * conj, int señal)

int **sigismember**(const sigset_t * conj, int señal)

conj: Conjunto de señales

señal: Señal

retorna: 0: éxito, -1: error



# Señales en POSIX 1003.1

- Bloqueo de señales

  **Máscara de señales**: Estado de bloqueo/desbloqueo
  Se fijan (agregan) las señales que se desea bloquear

  – Fijar la máscara de señales de un proceso

  int **sigprocmask**(int como, const sigset_t *conj, sigset_t *conjant)

  como: SIG_BLOCK: Se agregan las señales en conj
  SIG_UNBLOCK: Se borran las señales en conj
  SIG_SETMASK: La máscara es conj

  conj: Conjunto de señales
  conjant: Máscara anterior

  La máscara inicial es heredada del padre



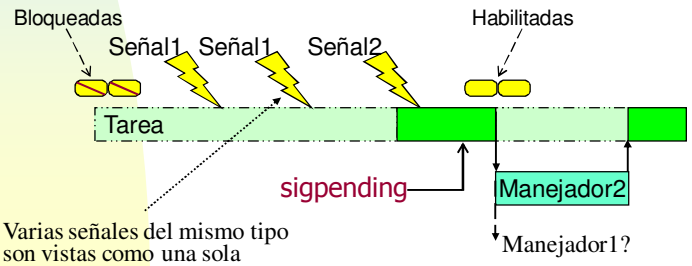
# Señales en POSIX 1003.1

- Examinar las señales pendientes

  int **sigpending**(sigset_t *conj)

  conj: Conjunto de señales (bloqueadas) que han sido recibidas y están pendientes

  retorna : 0: éxito, -1: error



# Señales en POSIX 1003.1

- Sincronización con señales
  - Esperar una señal

    int **sigsuspend**(const sigset_t *nmascara)

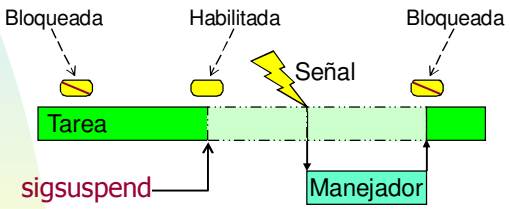
    nmascara: Nueva máscara de señales

  - Reemplaza la máscara actual por la nueva máscara
  - Suspende la tarea hasta que llega una señal no bloqueada, no ignorada (en la nueva máscara).
  - Cuando se reactiva, restablece la máscara original
  - La señal es atendida por el manejador previamente definido



# Señales en POSIX 1003.1

- Sincronización con señales (cont.)



Departamento de Telemática

# Problemas con las señales POSIX 1003.1

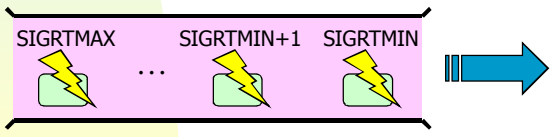
- Pocas señales para las aplicaciones de usuario: SIGUSR1, SIGUSR2
- No hay colas de señales: Varias señales del mismo tipo (e.g. SIGUSR1) enviadas antes de ser atendida la primera, son vistas como una sola
- No se establece en qué orden se atienden las señales bloqueadas
- Poca capacidad de transporte de información: Sólo se sabe que ha ocurrido un evento
- Velocidad: Cuando se recibe la señal hay que ejecutar siempre un manejador



Departamento de Telemática

# Señales en POSIX 1003.1b

- Identificadores: SIGRTMIN...SIGRTMAX
  - Como mínimo 8
- Se pueden enviar a través de una cola
- Se atienden en orden de prioridad
  - Primero las de identificador menor
- Transportan un dato



# Señales en POSIX 1003.1b

Departamento de Telemática

- Envío de señales
  - Enviar una señal a la cola de un proceso

  int **sigqueue**(pid_t pid, int señal,
  const union sigval valor)

  pid: ID del proceso de destino
  señal: Señal a enviar
  valor: Valor transportado por la señal

```
union sigval {
    ...
    int    sigval_int;
    int   *sigval_ptr;
    ...
}
```



# Señales en POSIX 1003.1b

Departamento de Telemática

- Establecimiento de la acción para una señal

  Estructura sigaction: Define la acción para una señal

```
struct sigaction {
   void (*)()                              sa_handler
   sigset_t                                sa_mask
   int                                     sa_flags
   void (*)(int, siginfo_t *, void *)      sa_sigaction
}
```

  sa_flags: SA_SIGINFO: Usar sa_sigaction

  - Cambiar el manejador de una señal

  int **sigaction**(int señal, const struct sigaction *accion,
  struct sigaction *accionant)

## Sincronización (rápida) con señales de tiempo real

- El servicio sigsuspend() tarda mucho en reactivar la tarea:
  - El S.O. arregla la ejecución del manejador de señal
  - Se ejecuta el manejador de señal
  - El S.O. arregla la continuación de la tarea suspendida
- La propia tarea ya está esperando la señal, y podría decidir qué hacer cuando le llegue
- Los siguientes servicios permiten esperar a una señal sin llamar al manejador de señales:
  sigwait(), sigwaitinfo(), sigtimedwait()



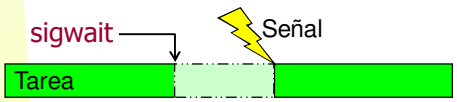
## Señales en POSIX 1003.1b

- Sincronización (rápida) con señales de tiempo real
  - Esperar una señal en la cola

  int **sigwait**(const sigset_t *conj, int *señal)

  conj: Máscara de señales esperadas
  señal: Señal recibida
  retorna: 0: éxito, -1: error

  Las señales a esperar conj deberían estar bloqueadas antes de llamar a sigwait()



# Señales en POSIX 1003.1b

Departamento de Telemática

– Esperar una señal en la cola (Cont.)

int **sigwaitinfo**(const sigset_t *conj, siginfo_t *info)

int **sigtimedwait**(const sigset_t *conj, siginfo_t *info, const struct timespec *tiempo)

conj: Máscara de señales esperadas
info: Información de la señal recibida (No., causa, valor)
tiempo: Tiempo máximo de espera
retorna: No. de la señal recibida
-1: error

Además del identificador de la señal, retornan la información que transporta



# Señales en POSIX 1003.1c

Departamento de Telemática

- Envío de señales
  – Enviar una señal a un hilo

  int **pthread_kill**(pthread_t hilo, int señal)

  hilo: Hilo de destino
  señal: Señal a enviar
  retorna: 0: éxito, -1: error

  Si señal=0 (null), prueba (no se envía)

# Señales en POSIX 1003.1c

- Bloqueo de señales
  - Fijar la máscara de señales de un hilo

  int **pthread_sigmask**(int como, const sigset_t *conj, sigset_t *conjant)

  como: SIG_BLOCK: Se agregan las señales en conj
  SIG_UNBLOCK: Se borran las señales en conj
  SIG_SETMASK: La máscara es conj

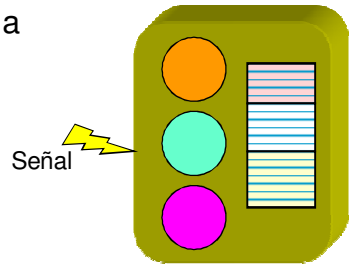
  conj: Conjunto de señales
  conjant: Máscara anterior

  La máscara inicial es heredada del padre



# Señales, procesos e hilos

- Las señales (síncronas) de error (e.g. SIGFPE, SIGILL) son enviadas al hilo que causa el error
- Las señales generadas por eventos asíncronos (E/S, mensajes, temporizadores) son enviadas al proceso
- Una señal enviada a un proceso es recibida por cualquiera de los hilos que no la tengan bloqueada
- Una tarea puede enviar señales a
  - un proceso: kill(), sigqueue()
  - un hilo: pthread_kill()



Departamento de Telemática

# Señales en RTJava

- RTJava soporta todas las señales del POSIX siempre y cuando el núcleo Linux las contenga, es decir, RTJava provee los métodos para administrar las señales POSIX pero no las implementa sobre el S.O.
- Las señales POSIX son representadas por la clase **AsyncEvent**.

  La ocurrencia de un evento invoca el método **AsyncEvent.fire()**
- Los manejadores de señales en RTJava heredan de la clase para atención de eventos asíncronos **AsyncEventHandler**.
- RTJava no implementa ninguna de las funciones POSIX para control de señales.



Departamento de Telemática

# Señales en RTJava

- Para asociar los manejadores a las señales POSIX se utiliza la clase **POSIXSignalHandler**.
- Los métodos disponibles para gestionar los manejadores de señales POSIX son:
  - **addHandler** (int SIGNAL, AsyncEventHandler handler)

    El manejador se registra como habilitado para atender la señal dada.
  - **removeHandler** (int SIGNAL, AsyncEventHandler handler)

    El manejador se elimina de la lista de manejadores para atender la señal dada.
  - **setHandler** (int SIGNAL, AsyncEventHandler handler)

    El manejador se registra como el único habilitado para atender la señal dada.

Departamento de Telemática

# Señales en RTJava

- Listado de señales

| Identificador | Acción por defecto | Usada para |
|---|---|---|
| SIGABRT | Destrucción | Terminación anormal, aborto |
| SIGALRM | Destrucción | Expiración de la alarma del reloj |
| SIGFPE | Destrucción | Excepción de coma flotante |
| SIGILL | Destrucción | Excepción de instrucción ilegal |
| SIGINT | Destrucción | Destrucción interactiva (Ctrl-C) |
| SIGKILL | Destrucción | Destrucción no deshabilitable |
| ... | | |
| SIGUSR1 | Destrucción | Aplicaciones de usuario |
| SIGUSR2 | Destrucción | Aplicaciones de usuario |
| SIGTRMIN | | Aplicaciones de usuario |
| ... | | Aplicaciones de usuario |
| SIGTRMAX | | Aplicaciones de usuario |
| SIGCHLD | Ignorada | Destrucción o detención de hijo |
| SIGSTOP | Suspensión | Detención del proceso |
| SIGCONT | Activación | Continuación del proceso |
| ... | | |



Departamento de Telemática

# Temario

- Introducción
- Señales
- **Acceso a recursos compartidos**
  - Exclusión mutua
  - Semáforos
  - Monitores y Variables condicionales
  - Mecanismos POSIX
- Paso de mensajes
  - Colas de mensajes

# Acceso a recursos compartidos

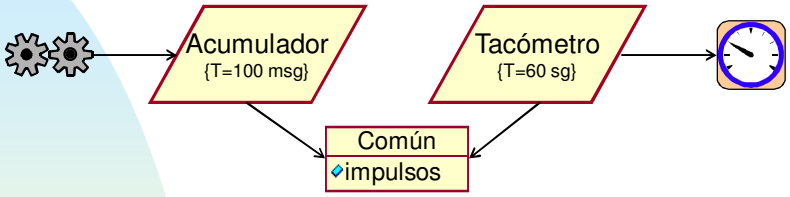
Departamento de Telemática

- En un sistema monoprocesador, la forma más directa de comunicación entre tareas es mediante datos compartidos
- También es común la competencia por el acceso a periféricos y otros recursos compartidos
- El acceso a recursos compartidos (datos, periféricos, etc.) no es fiable cuando no es controlado



# Acceso a recursos compartidos

Departamento de Telemática




```
/*-- Acumulador --*/
void *acumulador() {
  ...
  while (TRUE) {
    pthread_wait_np();
    if (pulso())
      impulsos++;
  }
}
```

```
/*-- Tacómetro --*/
void *tacómetro() {
  ...
  while (TRUE) {
    pthread_wait_np();
    desplegar(impulsos);
    impulsos= 0;
  }
}
```

# Acceso a recursos compartidos

Departamento de Telemática

```
/*-- Acumulador --*/
if (pulso()
  impulsos++
```

```
/*-- Tacómetro --*/
desplegar(impulsos)


impulsos= 0
```

Se pierde un pulso

```
/*-- Acumulador --*/
if (pulso()
  //impulsos++
  LOAD R1,impulsos


  INC R1
  LOAD impulsos,R1
```

```
/*-- Tacómetro --*/



desplegar(impulsos)
impulsos= 0
```

No se reinicia el contador!!



# Acceso a recursos compartidos

Departamento de Telemática

- **Condición de carrera** (*race condition*): Comportamiento anómalo producido por una dependencia crítica ante la temporización relativa de los eventos
- **Sección crítica**: Secuencia de instrucciones que deben ejecutarse de manera atómica (indivisible)

```
/*-- Acumulador --*/
      impulsos++;
```

```
/*-- Tacómetro --*/
    desplegar(impulsos);
    impulsos= 0;
```

- **Exclusión mutua**: Mecanismo de sincronización requerido para proteger una Sección Crítica

Para efectos de análisis, se supondrá que el acceso a memoria es atómico: Sólo una tarea a la vez puede acceder a una posición de memoria.

# Acceso a la Sección Crítica

- Solución general:

```
/*-- Tarea --*/
void *tarea(void *arg) {
  ...
  while (true) {
    /* protocolo de entrada */
    /** sección crítica **/
    /* protocolo de salida */
    ...
  }
}
```



# Tipos de solución

- Fuerza bruta: Bloquear interrupciones
  - Puede hacerse por intervalos cortos (RT-Linux)
- Espera ocupada o activa
  - Las tareas “dan vueltas” (*spinning*) en torno a variables esperando luz verde para entrar en la sección crítica:
    - Uso de instrucciones de máquina (test-and-set)
    - Uso de variables de control
- Servicios del Sistema Operativo
  - Suspenden las tareas evitando la espera ocupada

Departamento de Telemática

# Semáforos

- Mecanismo propuesto por el holandés Edsger Dijkstra, que permite controlar la exclusión mutua para un número arbitrario de tareas
- Consta de una variable y dos operaciones:
  Variable: s (semáforo), entera, no negativa
  Operaciones: P (Probar) (del holandés *Proberen*)
  V (Incrementar) (del holandés *Verhogen*)
- La variable sólo es accedida por las operaciones
- Las operaciones también se llaman Wait (P) y Signal (V)
- Son atómicas (indivisibles), normalmente provistas por el sistema operativo
- Controlan exclusión mutua y sincronización condicional



Departamento de Telemática

# Operaciones de los semáforos

- Usando "espera ocupada"

```
P(s):  while (s <= 0)     /* P: Probar */
         /* esperar */;
       s= s-1;

V(s):  s= s+1;            /* V: Incrementar */
```

- Con cola de procesos suspendidos

```
P(s):  s= s-1;            /* Wait */
       if (s < 0)
         /* bloquear tarea en cola de S */;

V(s):  s= s+1;            /* Signal/Post */
       if (s <= 0)
         /* desbloquear una tarea de la cola */;
```

Departamento de Telemática

# Uso de los semáforos

- Exclusión mutua

```
/*-- Acumuluador --*/
void *acumulador() {
  ...
  while (TRUE) {
    pthread_wait_np();
    if (pulso()) {
      P(semaforo);
      impulsos++;
      V(semaforo)
    }
  }
}
```

```
/*-- Tacómetro --*/
void *tacometro() {
 ...
  while (TRUE) {
    pthread_wait_np();
    P(semaforo);
    desplegar(impulsos);
    impulsos= 0;
    V(semaforo)
  }
}
```



Departamento de Telemática

# Uso de los semáforos

- Exclusión mutua

```
semaforo = 1;
```

```
/*-- Tarea1 --*/
void *tarea1(void *arg) {
  ...
  while (TRUE) {
    P(semaforo);
    /** sección crítica
    V(semaforo)
  }
}
```

```
/*-- Tarea2 --*/
void *tarea2(void *arg) {
 ...
  while (TRUE) {
    P(semaforo);
    /** sección crítica **/
    V(semaforo)
  }
}
```

# Uso de los semáforos

Departamento de Telemática

- Exclusión mutua

```
semaforo = 1;
```

```
/*-- Tarea1 --*/
P(semaforo);
  s= 1-1 = 0
/** sección crítica **/



V(Semaforo)
  s= -1+1 = 0
  desbloquea Tarea2
```

```
/*-- Tarea2 --*/



P(Semaforo)
  s= 0-1 = -1
  bloquea Tarea2



/** sección crítica **/
```

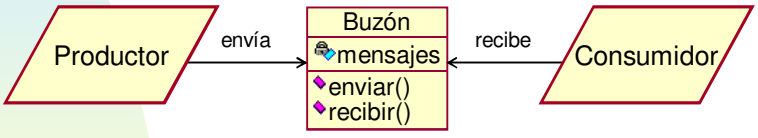
El valor inicial de **`semaforo`** determina cuántos procesos pueden entrar simultáneamente a la Sección Crítica -> Semáforos de conteo



# Sincronización condicional

Departamento de Telemática

- Se requiere cuando una tarea sólo puede realizar una operación si otra tarea ha tomado cierta acción o se encuentra en un determinado estado




- El Productor no puede enviar cuando el Buzón está lleno.
  Debe esperar a que el Consumidor reciba un mensaje
- El Consumidor no puede recibir cuando el Buzón está vacío
  Debe esperar a que el Productor envíe un mensaje

# Uso de los semáforos

Departamento de Telemática

- Sincronización condicional

```
semaforo = 0;
```

```
/*-- Tarea1 --*/
void *tarea1(void *arg) {
  ...
  while (TRUE) {
    ...
    P(semaforo);
    /** continuacion **/
    ...
  }
}
```

```
/*-- Tarea2 --*/
void *tarea2(void *arg) {
  ...
  while (TRUE) {
    /** acción **/
    V(semaforo);
    ...
  }
}
```

pthread_wait_np();

pthread_kill(Tarea1, RTL_SIGNAL_WAKEUP);

La **Tarea1** espera que la **Tarea2** la active



# Uso de los semáforos

Departamento de Telemática

- Sincronización condicional

```
semaforo = 0;
```

```
/*-- Tarea1 --*/
P(semaforo);
  s= 0-1 = -1
  bloquea Tarea1



/** continuacion **/
```

```
/*-- Tarea2 --*/



/** acción **/
V(Semaforo)
  s= -1+1 = 0
  desbloquea Tarea1
```

Departamento de Telemática

# Problemas de los semáforos

- Es una elegante primitiva de bajo nivel...
  - puede ser usada en la construcción de servicios de más alto nivel
- ... pero su uso puede llevar a cometer errores
  - El olvido o mala colocación de una operación P o V es fatal
- Puede haber mecanismos más estructurados y fiables
- “Tienen importancia histórica mas no son adecuados para tiempo Real” (Burns)



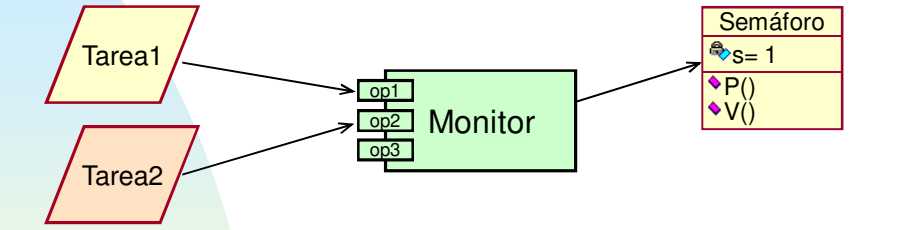
Departamento de Telemática

# Monitores

- Concepto propuesto por Dijkstra, Brinch-Hansen y Hoare, para estructurar el uso de regiones críticas
- Un **Monitor** es un módulo de programación cuyos procedimientos (operaciones) encapsulan regiones críticas
- Las operaciones del monitor garantizan la exclusión mutua. No requieren un semáforo externo para ser usadas

Departamento de Telemática

# Monitores




```
/*-- Tarea1 --*/
void *tarea1(void *arg) {
  ...
  while (TRUE) {
    ...
    monitor.op1(100);
    ...
  }
}
```

```
void op1(int arg) {
  P(semaforo);
  /** sección crítica **/
  ...
  V(semaforo)
}
```

```
/*-- Tarea2 --*/
void *tarea2(void *arg) {
  ...
  while (TRUE) {
    ...
    monitor.op2('c');
}
```

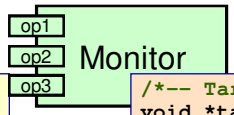
```
void op2(char arg) {
  P(semaforo);
  /** sección crítica **/
  ...
  V(semaforo)
}
```



Departamento de Telemática

# Variables Condicionales

- ¿Qué hacer cuando una tarea que ha accedido al Monitor debe esperar que se cumpla una condición?
- Hoare propuso las **variables condicionales**




```
/*-- Tarea1 --*/
void *tarea1(void *arg) {
  ...
  while (TRUE) {
    ...
    monitor.op1(100);
    ...
  }
}
```

```
void op1(int arg) {
  P(semaforo);
  ...
  P(Cond);
  ...
  V(semaforo)
}
```

```
/*-- Tarea3 --*/
...
  while (TRUE) {
    /** accion **/
    V(Cond);
    ...
```

```
/*-- Tarea2 --*/
void *tarea2(void *arg) {
  ...
  while (TRUE) {
    ...
    monitor.op2('c');
    ...
  }
```

```
void op2(char arg) {
  P(semaforo);
  /** sección crítica **/
  ...
  V(semaforo)
}
```

Departamento de Telemática

# Variables Condicionales

- Primitivas similares a un semáforo con valor inicial cero, para sincronización condicional
- Están asociadas a un monitor
- Ofrecen dos operaciones atómicas: wait y signal (similares a P y V)

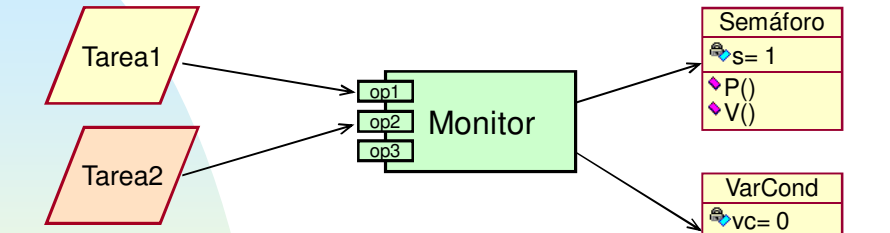
```
wait(vc):
      /* bloquear tarea en cola de vc */;
      /* desbloquear el acceso al monitor */;

signal(vc):
      if (/* hay tareas en la cola de vc */) {
         /* desbloquear una tarea de la cola */;
         /* bloquear el acceso al monitor */;
      }
```



Departamento de Telemática

# Variables Condicionales




```
/*-- Tarea1 --*/
void *tarea1(void *arg) {
  ...
  while (TRUE) {
    ...
    monitor.op1(100);
    ...
  }
}
```

```
void op1(int arg) {
  P(semaforo);
  ...
  wait(varCond);
  ...
  V(semaforo)
}
```

```
/*-- Tarea3 --*/
...
  while (TRUE) {
    /** accion **/
    signal(varCond);
    ...
```

Departamento de Telemática

# Variables Condicionales

- Características
  - La operación de bloqueo por la condición y el desbloqueo del acceso al monitor es atómica
  - La operación de desbloqueo por la condición y bloqueo del acceso al monitor es atómica
- Problema

  Cuando se llama a signal pueden resultar dos tareas dentro del monitor: llamante y desbloqueada
- Soluciones
  - Permitir signal sólo al final del procedimiento
  - signal fuerza la salida del monitor
  - signal bloquea la tarea llamante si desbloquea otra



Departamento de Telemática

# Sincronización en POSIX 1003.1b

- Semáforos de conteo con nombre
  - Utilizan una interfaz de servicios y nombrado similar a la de los archivos
  - El Sistema Operativo suministra los recursos, compartidos entre procesos
  - El manejo de nombres puede ser engorroso
- Semáforos de conteo en memoria (sin nombre)
  - Más flexibles
  - El usuario suministra una dirección de memoria para el semáforo
  - Deben colocarse en memoria compartida para ser usados entre diversos procesos

Departamento de Telemática

## Semáforos de conteo con nombre

- Creación/apertura

  sem_t ***sem_open**(const char *nombre, int bands, mode_t modo, unsigned int valor_inicial)

  nombre: Nombre del semáforo (empezar con "/")
  bands: O_CREAT, O_EXCL
  modo: Permisos (S_IRWXU, S_IRWXG, S_IRWXO)
  valor_inicial >= 0
  retorna: Descriptor del semáforo (un apuntador)



Departamento de Telemática

## Semáforos de conteo con nombre

- Cierre y destrucción
  - Cerrar el acceso del proceso a un semáforo

  int **sem_close**(sem_t *semaforo)
  semaforo: Descriptor
  retorna: 0: éxito, -1: error

  - Destruir un semáforo

  int **sem_unlink**(char *nombre)
  nombre: Nombre del semáforo
  retorna: 0: éxito, -1: error

## Semáforos de conteo en memoria

Departamento de Telemática

- Creación

  int **sem_init**(sem_t *sem_dir, int interproc, unsigned int valor_inicial)

  sem_dir: Dirección de memoria para el semáforo

  interproc: Compartición entre procesos

  1: Compartido con otros procesos

  0: No compartido con otros procesos (sólo para hilos)

  valor_inicial >= 0

  retorna: 0: éxito, -1: error



## Semáforos de conteo en memoria

Departamento de Telemática

- Destrucción

  int **sem_destroy**(sem_t * sem_dir)

  sem_dir: Dirección de memoria del semáforo

  retorna: 0: éxito, -1: error

  **Atención**: No mezclar las operaciones de creación y destrucción de los semáforos con nombre y los semáforos en memoria!

# Semáforos de conteo (ambos)

Departamento de Telemática

- Operaciones P (*wait*) y V (*post*)
  - Operación P

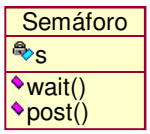
    int **sem_wait**(sem_t *semaforo)
  - Operación P no bloqueante

    int **sem_trywait**(sem_t *semaforo)
  - Operación V

    int **sem_post**(sem_t *semaforo)

    semaforo: Descriptor

    retorna: 0: éxito, -1: error






# Semáforos de conteo (ambos)

Departamento de Telemática

- Depuración
  - Obtener el valor de un semáforo

    int **sem_getvalue**(sem_t semaforo, int *valor)

    semaforo: Descriptor

    valor: Valor del semáforo al invocar la función

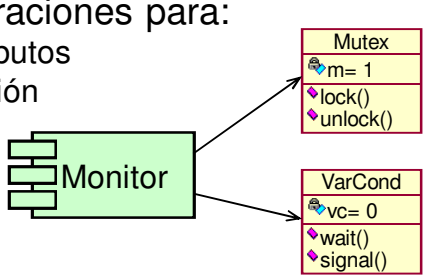
    retorna: 0: éxito, -1: error

    Es un valor orientativo

Departamento de Telemática

# Sincronización en POSIX 1003.1c

- Dos mecanismos que permiten construir monitores:
  - **Exclusión mutua (*mutex*)**: Semáforos binarios
  - **Variables condicionales**: Están asociadas a los *mutex*, para la implementación de monitores
- Ambos ofrecen operaciones para:
  - Inicialización de atributos
  - Creación y destrucción
  - Uso

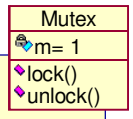





Departamento de Telemática

# Exclusión mutua (*mutex*)

- Los *mutex* son semáforos binarios
- Usados para controlar la exclusión mutua: sólo una tarea puede acceder a la región crítica




```
P(m):   if (m == 1)
          m = 0;
        else
          /* bloquear hilo en cola de mutex */;

V(m):   if (/* hay hilos en la cola */)
          /* desbloquear un hilo de la cola */;
        else
          m = 1;
```

Departamento de Telemática

# Exclusión mutua (*mutex*)

- Inicialización de atributos
  - Crear un objeto atributo para *mutex* e inicializarlo con los valores por defecto

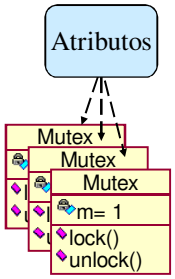
    int **pthread_mutexattr_init**(
    const pthread_mutexattr_t* atributo)

    atributo: Objeto atributo a crear

    retorna: 0: éxito, -1: error
  - Destruir un objeto atributo para *mutex*

    int **pthread_mutexattr_destroy**(
    pthread_mutexattr_t* atributo)

    atributo: Objeto atributo a destruir






Departamento de Telemática

# Exclusión mutua (*mutex*)

- Obtener el atributo process-shared(*) de un objeto atributo para *mutex*

  int **pthread_mutexattr_getpshared**(
  const pthread_mutexattr_t* atributo, int* pshared)

  atributo: Objeto atributo

  pshared: Dirección para el atributo process-shared
- Fijar el atributo process-shared

  int **pthread_mutexattr_setpshared**(
  pthread_mutexattr_t* atributo, int pshared)

  atributo: Objeto atributo

  pshared: Valor del atributo process-shared

(*)PTHREAD_PROCESS_SHARED: *mutex* compartido entre procesos

Departamento de Telemática

# Exclusión mutua (*mutex*)

- Creación y destrucción
  - Crear un *mutex*

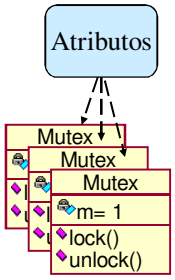
    int **pthread_mutex_init**(pthread_mutex_t* mutex,
    const pthread_mutexattr_t* atributo)

    mutex: *mutex* a crear

    atributo: Objeto atributo para *mutex*
  - Destruir un *mutex*

    int **pthread_mutex_destroy**(
    pthread_mutex_t* mutex)

    mutex: *mutex* a destruir






Departamento de Telemática

# Exclusión mutua (*mutex*)

- Operaciones de cierre (P) y apertura (V)
  - Cierre de un *mutex*

    int **pthread_mutex_lock**(pthread_mutex_t* mutex)

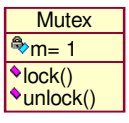
    mutex: *mutex* a cerrar
  - Intento de cierre de un *mutex* (no bloqueante)

    int **pthread_mutex_trylock**(
    pthread_mutex_t* mutex)

    mutex: *mutex* a cerrar
  - Apertura de un *mutex*

    int **pthread_mutex_unlock**(
    pthread_mutex_t* mutex)

    mutex: *mutex* a abrir. Deber pertenecer al hilo llamante



Departamento de Telemática

# Variables condicionales

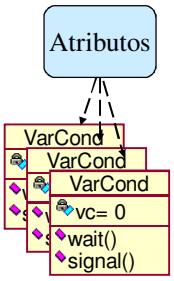
- Inicialización de atributos
  - Crear un objeto atributo para variable condicional e inicializarlo con los valores por defecto

    int **pthread_ condattr_init**(
    pthread_condattr_t* atributo)

    atributo: Objeto atributo a crear
  - Destruir un objeto atributo para variable condicional

    int **pthread_ condattr_destroy**(
    pthread_condattr_t* atributo)

    atributo: Objeto atributo a destruir






Departamento de Telemática

# Variables condicionales

- Obtener el atributo process-shared(*) de un objeto atributo para variable condicional

  int **pthread_condattr_getpshared**(
  const pthread_condattr_t* atributo, int* pshared)

  atributo: Objeto atributo
  pshared: Dirección para el atributo process-shared
- Fijar el atributo process-shared

  int **pthread_condattr_setpshared**(
  pthread_condattr_t* atributo, int pshared)

  atributo: Objeto atributo
  pshared: Valor del atributo process-shared

(*)PTHREAD_PROCESS_SHARED: V.C. compartida entre procesos

# Variables condicionales

– Obtener el atributo clock de un objeto atributo para variable condicional

int **pthread_condattr_getclock**(
const pthread_condattr_t* atributo, clockid_t* clock)

atributo: Objeto atributo
clock: ID del reloj

– Fijar el atributo clock de un objeto atributo para variable condicional

int **pthread_condattr_setclock**(
pthread_condattr_t* atributo, clockid_t clock)

atributo: Objeto atributo
clock: ID del reloj



# Variables condicionales

- Creación y destrucción
  – Crear una variable condicional
  int **pthread_cond_init**(pthread_cond_t* cond, pthread_condattr_t* atributo)
  cond: Variable condicional a crear
  atributo: Objeto atributo para variable condicional

  – Destruir una variable condicional
  int **pthread_cond_destroy**(pthread_cond_t* cond)
  cond: Variable condicional a destruir

Departamento de Telemática

# Variables condicionales

- Operaciones *wait*
  - Esperar en una variable condicional

    int **pthread_cond_wait**(pthread_cond_t* cond,
    pthread_mutex_t* mutex)

    cond: Variable condicional

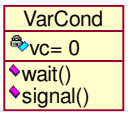
    mutex: *mutex* a desbloquear
  - Esperar con temporización en una variable cond.

    int **pthread_cond_timedwait**(pthread_cond_t* cond,
    pthread_mutex_t* mutex,
    const struct timespec* tiempo)

    cond: Variable condicional

    mutex: *mutex* a desbloquear

    tiempo: Tiempo absoluto límite de espera






Departamento de Telemática

# Variables condicionales

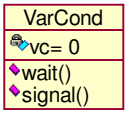
- Operaciones *signal*
  - Desbloquear un hilo que espera en una variable condicional (el de mayor prioridad)

    int **pthread_cond_signal**(
    pthread_cond_t* cond)

    cond: Variable condicional
  - Desbloquear todos los hilos que esperan en una variable condicional (en orden de prioridad)

    int **pthread_cond_broadcast**(
    pthread_cond_t* cond)

    cond: Variable condicional



Departamento de Telemática

## Sincronización en RTJava

- La sincronización en RTJava no implementa nuevos mecanismos al Java2, simplemente extiende los existentes, por tanto no existe nada POSIX en ellos.
  - Se utiliza la clase **MonitorControl** para definir las políticas de acceso a recursos compartidos de todo el sistema o ciertos hilos; implementa "herencia de prioridad" y "techo de prioridad"
  - Todos los eventos asíncronos para acceso a recursos o control de interrupciones se hace a través de la clase **AsyncEvent**.



Departamento de Telemática

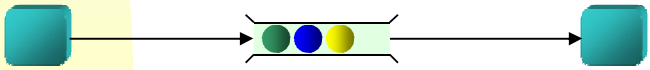
## Temario

- Introducción
- Señales
- Acceso a recursos compartidos
  - Exlusión mutua
  - Semáforos
  - Monitores y Variables condicionales
  - Mecanismos POSIX
- **Paso de mensajes**
  - Colas de mensajes

Departamento de Telemática

# Paso de mensajes

- Probablemente, es la forma de comunicación más general y abstracta
- Es un mecanismo que encaja con la sincronización y comunicación entre tareas en entornos centralizados y distribuidos
- La mayoría de las aplicaciones se formulan en términos de mensajes que se cruzan entre las tareas, así no se implementen con paso de mensajes
- En POSIX se dispone de varios mecanismos:
  Señales, tubos (*pipes*), FIFOs, y colas de mensajes

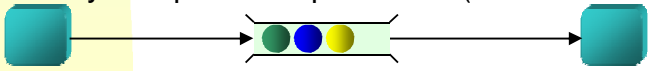




Departamento de Telemática

# Colas de mensajes en POSIX

- Características
  - Designación indirecta: uso de buzones
  - Copia del mensaje completo
  - Comunicación asíncrona
  - Longitud variable
- Operaciones (POSIX 1003.1b)
  - Creación y destrucción de colas de mensajes
  - Envío y recepción de mensajes
  - Notificación de mensajes
  - Inicialización de atributos de colas de mensajes
  - Envío y recepción temporizados (POSIX 1003.1d)



Departamento de Telemática

# Colas de mensajes

- Creación/apertura

  mqd_t **mq_open**(char *nombre, int bands, mode_t modo, struct mq_attr *atributos)

  nombre: Nombre de la cola de mensajes (empezar con "/")

  bands: O_RDONLY, O_WRONLY, O_RDWR, O_NONBLOCK, O_CREAT, O_EXCL

  modo: Permisos (e.g. S_IWUSR, S_IROTH)

  atributos: Atributos de la cola de mensajes
  - Máximo tamaño de los mensajes
  - Número máximo de mensajes en la cola

  retorna: Descriptor de la cola de mensajes

  modo y atributos sólo si O_CREAT



Departamento de Telemática

# Colas de mensajes

- Cierre del acceso

  int **mq_close**(mqd_t *cola)

  cola: Descriptor de la cola de mensajes

  retorna: 0: éxito, -1: error

- Destrucción

  int **mq_unlink**(const char* nombre)

  nombre: Nombre de la cola

  retorna: 0: éxito, -1: error

  Se hace efectiva cuando todos los procesos han cerrado su acceso a la cola

Departamento de Telemática

# Colas de mensajes

- Envío de mensajes

  int **mq_send**(mqd_t cola, const char *mensaje,
  size_t longitud, unsigned int prioridad)

  cola: Descriptor de la cola de mensajes

  mensaje: Dirección del mensaje

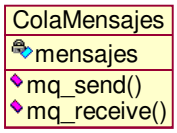
  longitud: Tamaño del mensaje

  prioridad: Prioridad del mensaje
  - A mayor valor, mayor prioridad
  - Los mensajes se almacenan en la cola en orden FIFO dentro de cada prioridad

  Si la cola está llena:

  Si O_NONBLOCK ==TRUE, retorna error

  Si O_NONBLOCK ==FALSE, la tarea se bloquea






Departamento de Telemática

# Colas de mensajes

- Recepción de mensajes

  int **mq_receive**(mqd_t cola, char *mensaje,
  size_t longitud, unsigned int *prioridad)

  cola: Descriptor de la cola de mensajes

  mensaje: Dirección para guardar el mensaje

  longitud: Tamaño del espacio para el mensaje

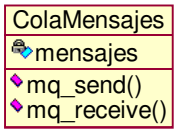
  prioridad: Prioridad del mensaje recibido

  Los mensajes de mayor prioridad se reciben antes que los de menor prioridad, y en orden FIFO dentro de cada prioridad

  Si la cola está vacía:

  Si O_NONBLOCK ==TRUE, retorna error

  Si O_NONBLOCK ==FALSE, la tarea se bloquea



Departamento de Telemática

# Colas de mensajes

- Notificación de mensajes

  int **mq_notify**(mqd_t cola,
  const struct sigevent *notificacion)

  cola: Descriptor de la cola de mensajes

  notificacion: Información de la señal a enviar:
  - Número de la señal
  - Valor a transportar por la señal

  Si la cola de mensajes está vacía, y no hay tareas bloqueadas esperando mensajes de la cola, y llega un mensaje, se envía la señal indicada a la tarea que se registra para notificación

  Sólo una tarea se puede registrar para notificación en la cola de mensajes



Departamento de Telemática

# Colas de mensajes

- Fijación (parcial) de atributos

  int **mq_setattr**(mqd_t cola,
  const struct mq_attr *nuevos_atr,
  struct mq_attr *viejos_atr)

  cola: Descriptor de la cola de mensajes
  nuevos_atr: Nuevos atributos
  viejos_atr: Atributos anteriores

  Sólo se afecta el atributo banderas: O_NONBLOCK

Departamento de Telemática

# Colas de mensajes

- Obtención de atributos

  int **mq_getattr**(mqd_t cola,
  struct mq_attr *atributos)

  cola: Descriptor de la cola de mensajes
  atributos: Dirección para los atributos de la cola de mensajes.
  Los atributos son:
  – Número máximo de mensajes en la cola
  – Máximo tamaño de los mensajes
  – Banderas: O_NONBLOCK
  – Número actual de mensajes en la cola



Departamento de Telemática

# Colas de mensajes

- Envío temporizado de mensajes

  int **mq_timedsend**(mqd_t cola,
  const char *mensaje, size_t longitud,
  unsigned int prioridad,
  const struct timespec* tiempo)

  cola: Descriptor de la cola de mensajes
  mensaje: Dirección del mensaje
  longitud: Tamaño del mensaje
  prioridad: Prioridad del mensaje
  tiempo: Tiempo absoluto límite de espera

Si llegado el tiempo no se ha logrado enviar el mensaje, la operación retorna con error = ETIMEDOUT

Departamento de Telemática

# Colas de mensajes

- Recepción temporizada de mensajes

  int **mq_timedreceive**(mqd_t cola, char *mensaje,
  size_t longitud, unsigned int prioridad,
  const struct timespec* tiempo)

  cola: Descriptor de la cola de mensajes
  mensaje: Dirección para guardar el mensaje
  longitud: Tamaño del espacio para el mensaje
  prioridad: Prioridad del mensaje recibido
  tiempo: Tiempo absoluto límite de espera
  retorna: Tamaño del mensaje recibido, ó -1 (error)

Si llegado el tiempo no se ha recibido el mensaje, la operación retorna con error = ETIMEDOUT



Departamento de Telemática

# Colas de mensajes en RTJava

- RTJava provee un robusto y completo conjunto de clases y métodos para el manejo de paso de mensajes.
- Las operaciones no son POSIX.
- Existen tres clases para implementación de colas de acuerdo a las necesidades del sistema, a saber:
  - **WaitFreeWriteQueue**
  - **WaitFreeReadQueue**
  - **WaitFreeDequeue**

# Colas de mensajes en RTJava

Departamento de Telemática

- **WaitFreeWriteQueue**
  - Escritura no sincronizada y no bloqueante (retorna falso si la cola está llena)
  - Lectura sincronizada y bloqueante (espera hasta que hayan datos para retornar).
- **WaitFreeReadQueue**
  - Lectura no sincronizada y no bloqueante (devuelve null si la cola está vacía)
  - Escritura sincronizada y bloqueante (espera hasta que haya espacio en la cola para escribir).
- **WaitFreeDequeue**
  - Escritura y lectura no sincronizadas y no bloqueantes.
  - Escritura y lectura sincronizadas y bloqueantes.

Sincronizada: La operación sólo puede ser ejecutada por un hilo al tiempo. Los demás hilos se quedan suspendidos



# Colas de mensajes en RTJava

Departamento de Telemática

- Operaciones comunes
  - Enviar un mensaje

    boolean **write** (java.lang.Object object)
  - Recibir un mensaje

    java.lang.Object **read** ()
  - Conocer estado de la cola

    boolean **isEmpty()**

    boolean **isFull()**

    int **size()**

Departamento de Telemática

# Referencias (1)


- A. Burns, A. Wellings. “Real-Time Systems and their Programming Languages”. Addison-Wesley. 1992.
- B.O. Gallmeister. “POSIX.4. Programming for the Real World”. O´Reilly, 1995.
- M. Milenlenkovic. “Sistemas Operativos. Conceptos y Diseño”. McGraw-Hill. 1988.
- The Open Group. “The Open Group Base Specifications Issue 6 (IEEE Std 1003.1-2001)”. The Single UNIX Specification C950. 2001. http://www.opengroup.org/publications/catalog/t950x.htm
- José Ismael Ripoll Ripoll. “Tutorial de Real Time Linux”. Universidad Politécnica de Valencia. 2001. http://bernia.disca.upv.es/~iripoll/rt-linux/rtlinux-tutorial/
- FSMLabs. “The definitive list of RTLinux functions”. 2002. http://www.fsmlabs.com/developers/man_pages/function_list.htm




Departamento de Telemática

# Referencias (2)


- The Real-Time for Java Expert Group. “The Real-Time specification for Java”. Addison-Wesley. 2000. http://www.rtj.org
- TimeSys Corporation. “The concise handbook of Real-Time systems”. 2002. http://www.ece.cmu.edu/~ece749/docs/RTSHandbook.pdf
- J.A. de la Puente. “Transparencias sobre sistemas de tiempo real”. Universidad Politécnica de Madrid. 2001.
- José Luis Villarroel Salcedo. “Transparencias del Curso Sistemas de Tiempo Real”. Universidad de Zaragoza. 2001.
- Sebastián Sánchez Prieto (Chan). “Transparencias del curso Arquitectura de Computadores”. Universidad de Alcalá. 2002.