Sistemas Encaixados

Tema 3: Sistemas Operativos

Oscar García Lorenzo

Escola Politécnica Superior de Enxeñaría

Sistemas Operativos

Índice

Índice

Índice

Índice

Sinais*

Para que se usan as sinais?

- Notificar aos procesos os eventos que ocorren no sistema
  - Identificación con enteiros e/ou constantes simbólicas
- Mecanismo de comunicación y sincronización entre procesos

Tratamento das sinais

- Finalizar o proceso, ignorar a sinal, suspender o proceso, reanudar o proceso, etc.
- Para algunas sinais podese definir unha acción diferente
  - Programando un manexador (*handler*) (sinal capturada)

Sinais*

Signal	Cause
SIGABRT	Sent to abort a process and force a core dump
SIGALRM	The alarm clock has gone off
SIGFPE	A floating-point error has occurred (e.g., division by 0)
SIGHUP	The phone line the process was using has been hung up
SIGILL	The user has hit the DEL key to interrupt the process
SIGQUIT	The user has hit the key requesting a core dump
SIGKILL	Sent to kill a process (cannot be caught or ignored)
SIGPIPE	The process has written to a pipe which has no readers
SIGSEGV	The process has referenced an invalid memory address
SIGTERM	Used to request that a process terminate gracefully
SIGUSR1	Available for application-defined purposes
SIGUSR2	Available for application-defined purposes

- SIGINT: Interrupción. Envíase cando se pulsa CRTL+C no tecrado. Por defecto interrumpese o programa
- SIGCHLD: Terminación de algún proceso fillo. Envíase ao proceso pai. Ignorada por defecto

Procesado de Sinais*

Xeración de Sinais

- Excepcións
- Outros procesos (`kill`)
- Interrupcións do terminal (CTRL+C)
- Control de tarefas. Terminación dun proceso fillo
- Xestión de cuotas
- Notificacións de E/S
- Alarmas

Recepción de Sinais

- A acción, incluíndo a terminación do proceso, a realiza o proceso receptor
- Necesita ser planificado para executar a acción correspondente

Sinais*

Sinais vs Interrupcións

- Interrupcións:
  - Veñen desde fora da CPU (periféricos) ou da propia CPU (excepcións, traps)
  - Enmascadadas por CPU
  - A CPU interrumpe o proceso, lanza o manexador (ISR, software)
  - O manexador (ISR) pode estar no SO
  - Esencialmente: As usa a CPU para comunicarse co SO
- Sinais:
  - As inicia o SO (núcleo/kernel) ou un proceso (`kill`)
  - Enmascadadas por Proceso
  - As manexa o SO, as envía ao proceso, o manexador está no proceso
  - Esencialmente: As usa o SO para comunicarse cos procesos
- Periferico/CPU -> *Interrupción* -> SO -> *Sinal* -> Proceso

Envío de sinais (POSIX)*

Chamada ao sistema `kill`

- En C: `resultado=kill(par,sinal)`

```
#include <signal.h>
main()
{    int a;
     if ((a=fork())==0) {
          while(1) {
             printf("pid do fillo = \%d\n",
                getpid());
             sleep(1); } }
     sleep(10);
     printf("Terminación do proceso con pid=
          \%d\n",a);
     kill(a,SIGTERM); }
```

- Desde consola: `kill -señal pid`

Captura de sinais (POSIX)*

Chamada ao sistema `signal` (manexador)

- En C: `resultado=signal(sinal, acción)`

```
#include <signal.h> #include <stdio.h>
#include <string.h>
void manexador(int sig);
main() {
   if (signal(SIGUSR1,manexador)==SIG_ERR)
         exit(1);
   for(;;);   }
void manexador(int sig){
     printf("\n\n%s recibida. \n",
          strsignal(sig));
     exit(2); }
```

- Desde consola: `kill -SIGUSR1 pid`

Captura de Sinais (POSIX)*

Definir o comportamento da sinal

- En C: `resultado=signal(sinal, acción)`
- `acción`
  - Dirección del manexador
  - SIG_DFL: acción por defecto
  - SIG_IGN: ignorar a sinal

Captura de señales en POSIX: Chamada a sistema `sigaction`

- Además do manexador, permite especificar outros parámetros e bandeiras de xestión da sinal
  - Máscaras de sinais bloqueadas
  - Desactivar o manexador unha vez recibida a sinal
  - Información sobre a sinal e o proceso que a enviou
  - etc.

Captura de Sinais (POSIX)*

Exemplo de uso de `sigaction`

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
main(void)
{
      int codigo_error=0;
      struct sigaction gestion;
      gestion.sa_handler = gestor;
      gestion.sa_mask = 0;
      gestion.sa_flags = SA_ONESHOT;
      codigo_error = sigaction  ( SIGINT, gestion, 0);
      if( codigo_error == SIG_ERR )
      {
            perror("Error al definir el gestor de SIGINT");
            exit(-1);
      }
      /** Código del programa ***/
      while(1);
}

void gestor( int señal )
{
     printf("Señal SIGINT recibida");
}
```

Chamadas ao sistema relacionadas con procesos (POSIX)*

System call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, opts)	Wait for a child to terminate
s = execve(name, argv, envp)	Replace a process' core image
exit(status)	Terminate process execution and return status
s = sigaction(sig, &act, &oldact)	Define action to take on signals
s = sigreturn(&context)	Return from a signal
s = sigprocmask(how, &set, &old)	Examine or change the signal mask
s = sigpending(set)	Get the set of blocked signals
s = sigsuspend(sigmask)	Replace the signal mask and suspend the process
s = kill(pid, sig)	Send a signal to a process
residual = alarm(seconds)	Set the alarm clock
s = pause( )	Suspend the caller until the next signal

Comunicación entre Procesos

Comunicación entre Procesos (IPC)

- Os procesos teñen memoria independente, como se comunican?
- Exemplos: pipes (canalización no shell), colas, mensaxes, notificacións, sinais ...
- O máis directo é crear zonas de memoria compartida
- Cuestións:
  - Pasar información
  - Evitar interferencias
  - Garantir orde correcto (dependencias)
- Mesmas cuestións para procesos e fíos
  - Fíos é máis fácil pasar información (memoria xa compartida), pode ter máis problemas no resto
- Multiprogramación

Comunicación entre Procesos

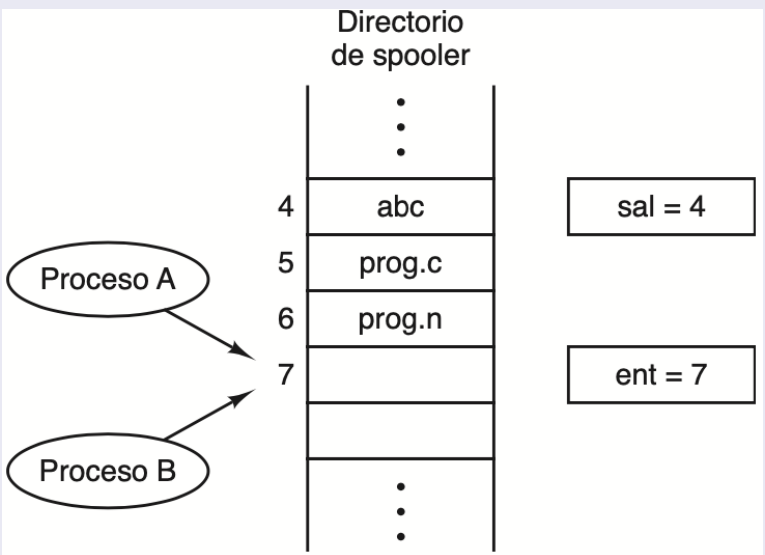
Recursos compartidos

- Os procesos, fíos, tarefas tamén poden compartir recursos:
  - Periféricos/Hardware: impresoras, botóns, tecrado, pantalla ...
  - Software: memoria compartida, bases de datos, comunicacións ...
- Os Sistemas Operativos axudan moito, pero non resolven todos os problemas
- As solucións son as mesmas que as usadas para a comunicación entre procesos

Comunicación entre Procesos

Condicións de carreira (condicións de competencia)

- *Race Condition*
- Exemplo: Spooler de impresión
  - Directorio de spooler (Cola de impresión)
  - Demonio de impresión
- Ley de Murphy
- Depurar programas con condicións de carreira é complexo

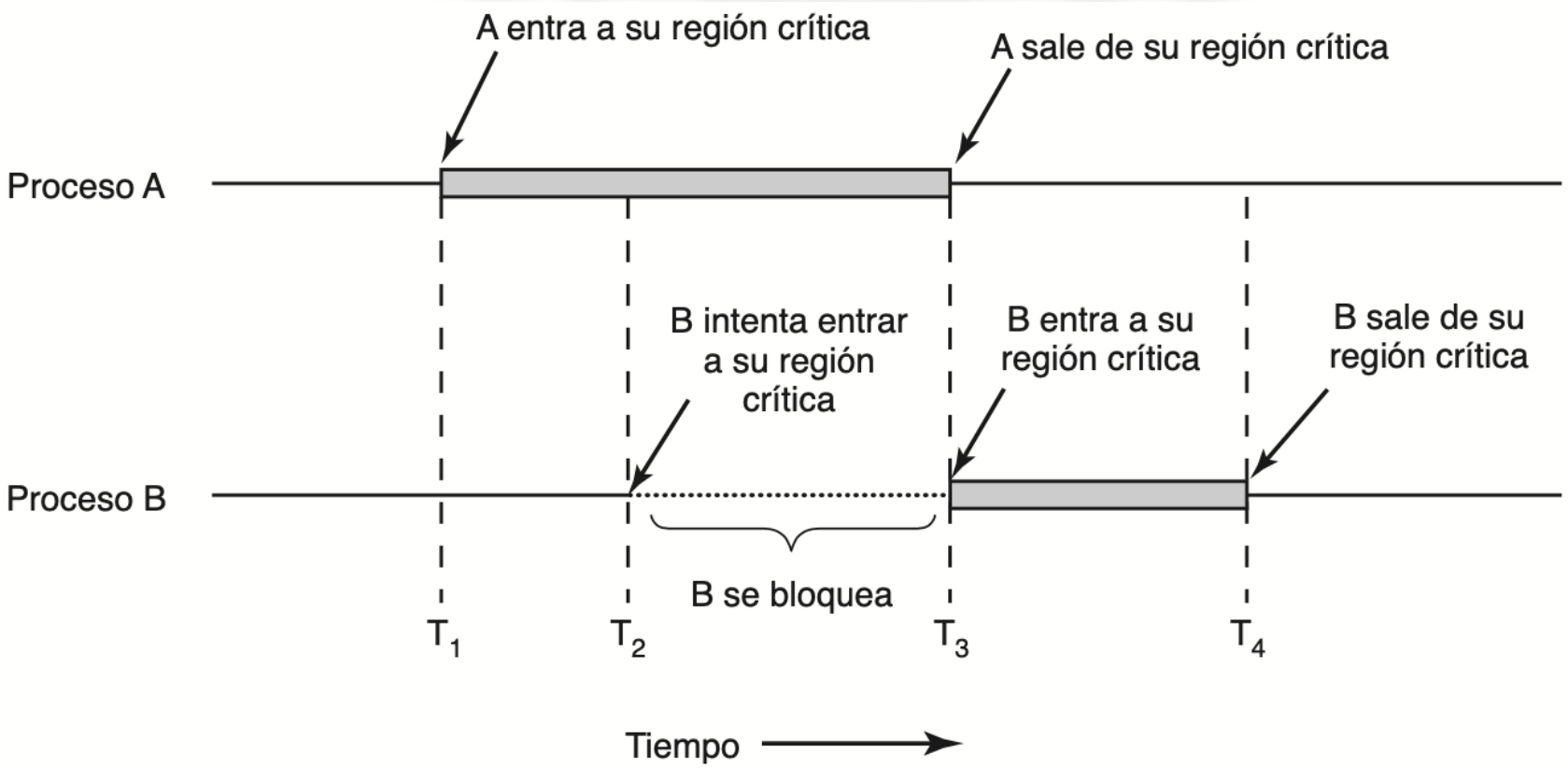


Comunicación entre Procesos

Rexión crítica

- Parte do programa que accede á memoria compartida/recurso compartido
- Necesítase **exclusión mutua**
- Condicións para unha boa solución:
  - Non pode haber dous procesos simultáneos na rexión crítica
  - Non se poden facer suposicións sobre velocidades ou número de CPUs
  - Un proceso non pode bloquear a outro sen estar na rexión crítica
  - Ningún proceso debe esperar indefinidamente ao acceso á rexión crítica
- Veremos varias solucións propostas históricamente, hoxe en día úsanse semáforos ou mutexes

Exclusión Mutua



Exclusión mutua con espera ocupada*

Deshabilitando interrupcións

- Deshabilitando interrupcións
  - Non se pode expropiar a un proceso (non hai cambios de contexto)
  - O kernel o fai con frecuencia, pero é perigoso en procesos do usuario
  - Máis común en Sistemas Encaixados, xa que non hai SO, e as interrupcións contrólanse directamente
  - É inutil en sistemas multprocesador ou multicore

Exclusión mutua con espera ocupada*

Variables de Candado (*Lock*)

- Intento de solución software
  - Candado=0 (ningún proceso está na súa rexión crítica)
  - Candado=1 (algún proceso está na súa rexión crítica)
- **Non funciona!**

Exclusión mutua con espera ocupada*

Alternancia Estricta

- Unha variable establece o turno (número de proceso)

```
While (TRUE) {
    while (turno !=0) ;
    region_critica();
    turno=1;
    region_no_critica();
}
```

```
While (TRUE) {
    while (turno !=1) ;
    region_critica();
    turno=0;
    region_no_critica();
}
```

- Espera ocupada (espera activa)
- Non se cumple a condición 3 (procesos bloqueados)
- Os procesos traballan ao ritmo do máis lento

Exclusión mutua con espera ocupada*

Solución de Peterson

```
#define FALSE 0
#define TRUE 1
#define N   2                                   /* número de procesos */

int turno;                                      /* ¿de quién es el turno? */
int interesado[N];                              /* al principio todos los valores son 0 (FALSE) */

void entrar_region(int proceso);                /* el proceso es 0 o 1 */
{
    int otro;                                   /* número del otro proceso */

    otro = 1 – proceso;                         /* el opuesto del proceso */
    interesado[proceso] = TRUE;                 /* muestra que está interesado */
    turno = proceso;                            /* establece la bandera */
    while (turno == proceso && interesado[otro] == TRUE)  /* instrucción nula */;
}

void salir_region(int proceso)                  /* proceso: quién está saliendo */
{
    interesado[proceso] = FALSE;                /* indica que salió de la región crítica */
}
```

Exclusión mutua con espera ocupada*

Instrucción TLS

- Axuda do hardware: é unha instrucción *atómica*: bloquea o bus de memoria
- Le e escribe ”á vez”
- Funciona para multicores
- Cortex-M non a ten (x86 e Cortex-A si)

```
entrar_region:
    TSL REGISTRO,CANDADO        |copia candado al registro y fija candado a 1
    CMP REGISTRO,#0             |¿era candado cero?
    JNE entrar_region           |si era distinto de cero, el candado está cerrado, y se repite
    RET                         |regresa al llamador; entra a región crítica

salir_region:
    MOVE CANDADO,#0             |almacena 0 en candado
    RET                         |regresa al llamador
```

Exclusión mutua con Durmir e Espertar*

Evitar a espera activa

- Usar chamadas ao sistema de comunicación
  - sleep
  - wakeup
- Evitar esperas activas que poden dar lugar ao problema de inversión de prioridades
- Usar unha posición en memoria para asociarse

Exclusión mutua con Durmir e Espertar*

Productor-Consumidor

```
#define N 100                                   /* número de ranuras en el búfer */
int cuenta = 0;                                 /* número de elementos en el búfer */

void productor(void)
{
        int elemento;

        while (TRUE) {                          /* se repite en forma indefinida */
                elemento = producir_elemento();         /* genera el siguiente elemento */
                if (cuenta == N) sleep();               /* si el búfer está lleno, pasa a inactivo */
                insertar_elemento(elemento);            /* coloca elemento en búfer */
                cuenta = cuenta + 1;                    /* incrementa cuenta de elementos en búfer */
                if (cuenta == 1) wakeup(consumidor);    /* ¿estaba vacío el búfer? */
        }
}



void consumidor(void)
{
        int elemento;

        while (TRUE) {                          /* se repite en forma indefinida */
                if (cuenta == 0) sleep();               /* si búfer está vacío, pasa a inactivo */
                elemento = quitar_elemento();           /* saca el elemento del búfer */
                cuenta = cuenta – 1;                    /* disminuye cuenta de elementos en búfer */
                if (cuenta==N–1) wakeup(productor);     /* ¿estaba lleno el búfer? */
                consumir_elemento(elemento);            /* imprime el elemento */
        }
}
```

Exclusión mutua con Durmir e Espertar*

Productor-Consumidor

- Problema cando o bufer está cheo ou baleiro
  - O *productor* durme para esperar a que se baleire
  - O *consumidor* durme esperando a que haxa algo
- Carreira:
  - O *consumidor* ve un 0 en *cuenta*
  - Inmediatamente o planificador durme a *consumidor* porque termina o seu *quantum*
  - O *productor* executase, mete algo no buffer, esperta ao *consumidor*
  - O *consumidor* non sabe que pasou a *listo*, xa leu *cuenta* e durmese
  - A sinal de espertar pérdese
- Hai diversas maneiras de arranxalo, pero son cada vez máis complicadas según aumenten o número de procesos implicados

Semáforos

Semáforo

- É unha variable que toma valores enteiros
  - E. W. Dijkstra (1965), contar número de sinais de espertar
- Dúas operacións **atómicas**:
  - ***down***: se o semáforo é maior que 0 o decrementa sen máis, se é 0 bloquease o proceso
  - ***up***: incrementa o semáforo, se algún proceso estaba bloqueado polo semáforo sespertase para executar o seu *down*
- O SO ten que garantir a atomicidade
  - Inhabilitar interrupcións
  - Non facer cambios de contexto
  - Como son poucas instruccións non hai moito problema
  - Con varias CPU hai que garantir que so unha á vez o usa

Semáforos

Productor-Consumidor con Semáforos

```
#define N 100                                   /* número de ranuras en el búfer */
typedef int semaforo;                           /* los semáforos son un tipo especial de int */
semaforo mutex = 1;                             /* controla el acceso a la región crítica */
semaforo vacias = N;                            /* cuenta las ranuras vacías del búfer */
semaforo llenas = 0;                            /* cuenta las ranuras llenas del búfer */

void productor(void)
{
        int elemento;

        while(TRUE){                            /* TRUE es la constante 1 */
                elemento = producir_elemento(); /* genera algo para colocar en el búfer */
                down(&vacias);                  /* disminuye la cuenta de ranuras vacías */
                down(&mutex);                   /* entra a la región crítica */
                insertar_elemento(elemento);    /* coloca el nuevo elemento en el búfer */
                up(&mutex);                     /* sale de la región crítica */
                up(&llenas);                    /* incrementa la cuenta de ranuras llenas */
        }
}


void consumidor(void)
{
        int elemento;

        while(TRUE){                            /* ciclo infinito */
                down(&llenas);                  /* disminuye la cuenta de ranuras llenas */
                down(&mutex);                   /* entra a la región crítica */
                elemento = quitar_elemento();   /* saca el elemento del búfer */
                up(&mutex);                     /* sale de la región crítica */
                up(&vacias);                    /* incrementa la cuenta de ranuras vacías */
                consumir_elemento(elemento);    /* hace algo con el elemento */
        }
}
```

Semáforos

Productor-Consumidor con Semáforos

- 2 semáforos para **Sinconizar**: *llenas* y *vacias*
  - *llenas*: Número de ranuras cheas: 0 ao principio
  - *vacias*: Número de ranuras baleiras: Tamaño do buffer ao principio
  - **Sinconizar**: Garante o orde
    - *productor* deixa de executarse cando o buffer está cheo
    - *consumidor* deixa de executarse cando o buffer está baleiro
- 1 semáforo para *exclusión mutua*: *mutex*
  - *llenas*: A 1 ao principio
  - So vale 1 ou 0: **Semáforo Binario**
  - *Exclusión mutua*: So 1 proceso pode entrar á vez

Mutexes

Mutex

- Versión simplificada de semáforo, so vale 1 (pechado) ou 0 (aberto)
- Dúas operacións **atómicas**:
  - ***lock***: pechar
  - ***unlock***: abrir
- Garantido polo SO
- Versión modo usuario (non código thumb!):

```
mutex_lock:
        TSL REGISTRO,MUTEX      |copia el mutex al registro y establece mutex a 1
        CMP REGISTRO,#0         |¿el mutex era 0?
        JZE ok                  |si era cero, el mutex estaba abierto, entonces regresa
        CALL thread_yield       |el mutex está ocupado; planifica otro hilo
        JMP mutex_lock          |intenta de nuevo
ok:     RET                     |regresa al procedimiento llamador; entra a la región crítica


mutex_unlock:
        MOVE MUTEX,#0           |almacena un 0 en el mutex
        RET                     |regresa al procedimiento llamador
```

Mutexes

Variables de Condición (POSIX Threads)

- Úsanse en conxunto cos mutexes en pthreads
- Serven para bloquear fíos ata que se cumpra unha determinada condición
- Están asociadas a mutexes

Semáforos e Mutexes

En FreeRTOS

- Hai semáforos e mutexes
- Hai semáforos binarios, sen herdanza de prioridade, para sincronización
  - Os mutexes mellor para exclusión mutua
- Hai mutexes recursivos
  - Pódese tomar, *lock* varias veces polo seu dono (xSemaphoreTakeRecursive())
  - Desbloqueanse chamando *unlock* tantas veces coma *lock* (xSemaphoreGiveRecursive())

Semáforos e Mutexes

En FreeRTOS

- Non o arranxan todo
- Pode haber **interbloqueos** por erros de programación fáciles de cometer e difíciles de depurar
- Poden crearse estruturas software chamadas *monitores* para organizar mellor o seu uso
- Agrupan procedementos, variables e estructuras de datos nun tipo especial de módulo ou paquete (ou clase)
- Os procesos poden chamar aos procedementos/acceder aos datos do monitor, pero so de un en un, con exclusión mutua
  - Son programación bastante avanzada, non a veremos
  - Excesiva para SE? En internet vese xente que os usa con FreeRTOS e Arduino

Interbloqueos

Interbloqueo

- Situación na que dous ou máis procesos quedan bloqueados indefinidamente ao entrar en conflicto as súas diferentes necesidades
- Ocorren porque os procesos compiten polo uso de recursos e porque se comunican e sincronizan entre eles
- Poden ocorrer en moitas situacións

Interbloqueos

Recursos

- Son obxetos otorgados polo sistema operativo aos procesos. O problema pode xurdir cando se otorgan de maneira exclusiva
  - Dispositivos hardware
  - Unha peza de información

Recursos

Apropiativos e Non Apropiativos

- Apropiativo: Pódeselle quitar
- Non Apropiativo: Non se lle Póde quitar
- Os problemáticos en términos de interbloqueos son os non apropiativos
- Secuencia de accións:
  1. Solicitar recurso
  2. Usar recurso
  3. Liberar recurso

Adquisición de recursos

Adquisición de recursos

```
typedef int semaforo;
semaforo recurso_1;


void proceso_A(void) {
  down(&recurso_1);
  usar_recurso_1();
  up(&recurso_1);
}
```

```
typedef int semaforo;
semaforo recurso_1;
semaforo recurso_2;


void proceso_A(void) {
  down(&recurso_1);
  down(&recurso_2);
  usar_ambos_recursos();
  up(&recurso_2);
  up(&recurso_1);
}
```

Adquisición de recursos

Adquisición de recursos

Sen Interbloqueo

```
typedef int semaforo;
  semaforo recurso_1;
  semaforo recurso_2;

  void proceso_A(void) {
    down(&recurso_1);
    down(&recurso_2);
    usar_ambos_recursos();
    up(&recurso_2);
    up(&recurso_1);
  }

  void proceso_B(void) {
    down(&recurso_1);
    down(&recurso_2);
    usar_ambos_recursos();
    up(&recurso_2);
    up(&recurso_1);
  }
```

Con potencial Interbloqueo

```
  semaforo recurso_1;
  semaforo recurso_2;

  void proceso_A(void) {
    down(&recurso_1);
    down(&recurso_2);
    usar_ambos_recursos();
    up(&recurso_2);
    up(&recurso_1);
  }

  void proceso_B(void) {
    down(&recurso_2);
    down(&recurso_1);
    usar_ambos_recursos();
    up(&recurso_1);
    up(&recurso_2);
  }
```

Intebloqueos

- A complexidade do problema aumenta co número de procesos e de recursos
- Existe moita investigación sobre interbloqueos
- Os pode haber por diversas causas
- Hai maneiras de solucionalos
  - Teoría de grafos
  - Traxectorias de recursos
  - ...
- Nos non as veremos, pero tede en conta de que estas cousas pasan

Comunicación entre Procesos*

Paso de mensaxes

- Funcións *send* e *receive*
  - Chamadas ao SO
- Hai moito investigado, teñen diversos problemas, pero solucionados
  - Mensaxes perdidas
    - Acuse de recibo (ACK, acknowledgement)
  - Autentificación
  - Lentitude na mesma máquina, copias
  - Buffers de envío e recepción
- Na materia ”Redes e Comunicacións” veranse protocolos de paso de mensaxes (para internet e redes, pero os conceptos son moi similares)

Paso de mensaxes*

Colas (POSIX)

- Úsanse para o paso de mensaxes entre procesos en Linux
- Implementanse coma ficheiros
  - Pódense ver con *ls /dev/mqueue*
- Teñen un nome
- Funcións *mq_send()* e *mq_receive()*
- Con *mq_send()* métese unha mensaxe cunha prioridade nunha cola en concreto
  - As mensaxes ordenanse na cola por prioridade
  - A igual prioridade a última mensaxe está arriba
- Con *mq_receive()* cóllese a mensaxe de máis arriba (maior prioridade)
- As funcións *mq_send()* e *mq_receive()* bloquean se a cola está chea ou non hai mensaxes
  - Uso de sinais para comprobar se hai datos

Paso de mensaxes*

Colas (POSIX)

- Complicadas de usar con máis de dous procesos
- Teñen funcións para enviar e recibir mensaxes cronometradas, para uso en tempo real
- Moita xente prefire usar *sockets* para isto (veranse en ”Redes e Comunicacións”)

Paso de mensaxes

Colas (FreeRTOS)

- Máis sinxelas, xeralmente FIFO (First in First Out)
- Os datos copianse á cola, a tarefa que envía os pode sobreescribir despois
  - FreeRTOS encárgase de reservar e liberar a memoria das colas
  - Se é necesario pódese definir para usar punteiros
- Unha sola cola pode recibir de varias tarefas e mensaxes diferentes
  - Faise unha estrucura con ”tipo de mensaxe” e ” mensaxe”
- As mensaxes non teñen prioridade
- Para uso con memoria protexida
- Hai unha API separada para o seu uso con interrupcións, máis rápida (ten que comprobar menos cousas)

Paso de mensaxes

Colas (FreeRTOS)

- Permiten definir un *tempo de bloqueo*
- Cando unha tarefa intenta ler pasa automáticamente a *bloqueada*
  - Pasa a *lista* cando a mensaxe está preparada
  - Pasa a *lista* cando acaba o *tempo de bloqueo*
- Cando unha tarefa intenta escribir pasa automáticamente a *bloqueada*
  - Pasa a *lista* cando hai espazo na cola
  - Pasa a *lista* cando acaba o *tempo de bloqueo*

Notificación Directa a Tarefas

Direct To Task Notifications (FreeRTOS)

- Alternativa máis rápida a semáforos, mutexes e colas
  - Mandan a notificación directamente á tarefa, evitan pasar polo semáforo, mutex, etc.
- Teñen dous valores:
  - Estado, *notification state*: Pode ser *pending* ou *non pending*
  - Valor, *notification value*: valor de 32 bits
- Hai un array de ata CONFIGTASK_NOTIFICATION_ARRAY_ENTRIES
  - A posición 0 é especial
  - Pódese mandar unha notificación a calquera posición

Notificación Directa a Tarefas

Direct To Task Notifications (FreeRTOS)

- Ao mandar unha notificación pódese cambiar o seu valor
  - Sobreescribilo aínda que non se lera
  - Sobreescribilo so si se leu antes
  - Cambiar un bit
  - Incrementar
- Limitacións
  - So pode haber unha tarefa receptora
    - O usual, p.e. cando unha interrupción esperta á tarefa que a vai procesar
  - Se se usa como cola a tarefa que envía non pode bloquearse ata que terminou o envío

Tempo real en Linux

- Existen versións de Linux para tempo real (RTLinux)
- Poden executar Linux coma un fío de menos prioridade que as tarefas en tempo real
  - As tarefas en tempo real e os ISR non son molestados polo kernel
  - É un parche sobre Linux
  - Ten un API próximo a POSIX Threads
    - Planificador expulsivo por prioridades fixas, sinais, sistema de arquivos POSIX (open, close, etc.) semáforos e variables condición
  - Acceso directo ao hardware (portos e interrupcións)

Memoria Virtual

Memoria Virtual (Linux)

- Fai que cada proceso pense que ten toda a memoria para el
- O SO traduce de direccións virtuais a físicas
- Se a memoria física non chega úsase o disco
- A memoria dividese en páxinas para administrar mellor
- Require unha MMU (Memory Management Unit) no procesador
  - Os microcontrodalores non a teñen
  - Cortex-M: Memoru Protection Unit (MPU), moito máis sinxela

Memoria Virtual

Memoria Protexida (FreeRTOS)

- Cortex-M: Memoru Protection Unit (MPU)
- Permite as tarefas correr en modo privilexiado ou non
  - O modo privilexiado está pensado para SO e baixo nivel, permite facer máis cousas co procesador
- Permite limitar o acceso á RAM, periféricos, código executable e memoria fora da pila (stack) da tarefa (modo non privilexiado)
- Como so está nos Cortex-M o seu soporte aínda requiere bastante traballo, non é tan automático
- Aumenta moito a seguridade
- Soe ser mala idea usar variables globais, así as evitas

Seguridade en Fíos

Thread Safe

- Unha función/procedemento/rutina é *thread safe* se se pode chamar desde diferentes fíos sen violar a seguridade
- Iso implica que ten que funcionar aínda que se chame á vez
- Ten que poder chamarse *durante outra chamada*: *reentrante*
  - Se usa variables globais non o será, esas variables poden cambiar
  - Se so usa variables da pila, depende so dos argumentos de entrada e non chama a funcións non reentrantes
- Variables locias ao fío
- Usar exclusión mutua
- Operación atómicas

Seguridade en Fíos

Interrupt Safe

- Non o mesmo que *thread safe*, pero similar
- A función/procedemento/rutina ten que poder ser interrumpida e non dar problemas
- Se a ISR intenta entrar ao mesmo recurso compartido pode haber problemas
  - A ISR ten máis prioridade, ten que executarse
  - A función está nunha sección crítica, ten o mutex
  - A ISR non pode obter o mutex, pero tampouco pode ceder o procesador
  - *Deadlock*
- Por iso é bo ter un SO polo medio
- En FreeRTOS hai función acabadas en *_FromISR* para usar en ISR

Varias cousas

- Coidado coas variables globais en multiprogramación
- FreeRTOS permite reservar memoria dinámicamente coas suas propias funcións (*heap*)
  - Malloc() e demás non son deterministas
  - Non son ”thread safe” (hai versións que si)
  - Non se soportan en microcontroladores (memoria virtual)
  - FreeRTOS ten unha versión das funcións para cada sistema que as pode soportar

Referencias

Andrew S. Tanenbaum "Sistemas operativos modernos"(3a edición). Editorial Prentice-Hall, 2009
Seccións 2.3, 6.1