Inter-process communication

Interprocess communication

- **Interprocess communication (IPC):** OS provides mechanisms so that processes can pass data.

- **Two types of semantics:**
  - blocking: sending process waits for response;
  - non-blocking: sending process continues.
Interprocess communication (IPC) mechanisms

- Semaphores
  - Binary
  - counting
- Signals
- Mail boxes
- Queues
- Pipes
CMSIS-RTOS inter-thread communication

- **Mutexes** – mutual exclusion locks
  - “lock” a resource to use it, and unlock it when done
  - Kernel suspends threads that need the resource until unlocked

- **Sempahores** – control access to common resource
  - Semaphore object contains tokens (“counting” semaphore)
  - Thread can request a token (put to sleep if none available)

- **Signal flags** – for thread synchronization
  - Each thread can have up to 31 SFs.
  - A thread can wait for its SFs to be set by threads/interrupts.

- **Message Queues and Mail Queues**
  - Queue is a first-in/first-out (FIFO) structure
  - “Message” is an integer or a pointer to a message frame
  - “Mail” is a memory block to put on queue/get from queue
  - Suspend thread if “put” to full queue or “get” from empty queue
Interprocess communication styles

- **Shared memory:**
  - processes have some memory in common;
  - cooperate to avoid destroying/missing messages.

- **Message passing:**
  - processes send messages along a communication channel
    ---no common address space.
  - comm. channel may be physical or virtual
Shared memory

- Shared memory on a bus:
Race condition in shared memory

- Assume a “flag” used to synchronizes access to shared memory
  - Flag = 1 when shared item is being used
  - Flag = 0 when shared item is not being used
  - To access the item: CPU must see flag = 0 and write flag = 1

- Problem when two CPUs try to write the same location:
  - CPU 1 reads flag and sees 0.
  - CPU 2 reads flag and sees 0.
  - CPU 1 sets flag to one and writes location.
  - CPU 2 sets flag to one and overwrites location.
Atomic test-and-set

- Problem can be solved with an atomic test-and-set:
  - single bus operation reads memory location, tests it, writes it.
- ARM test-and-set provided by SWP (swap):

  ```assembly
  ADR    r0,SEMAPHORE
  LDR    r1,#1
  GETFLAG SWP  r1,r1,[r0]
  BNZ    GETFLAG
  ```
Critical regions

- **Critical region**: section of code that cannot be interrupted by another process.

- Examples:
  - writing shared memory;
  - accessing I/O device.
Mutual Exclusion Example

System variables: Lock = 0;

while (1) {
    While (!Test-And-Set(&Lock));
    Critical Region
    Lock = 0;
    Remainder
} /* end while (1) */
Task and Device Queues

Processes queued for shared device access

- Queue header
- Ready queue
- Mag tape unit 0
- Mag tape unit 1
- Disk unit 0
- Terminal unit 0
- PCB
- Registers
Semaphores

- **Semaphore**: OS primitive for controlling access to critical regions.

- **Protocol**:
  1. Get access to semaphore with \texttt{P()} function.  
     \textit{(Dutch “Proberen” – to test)}
  2. Perform critical region operations.
  3. Release semaphore with \texttt{V()} function.  
     \textit{(Dutch “Verhogen” – to increment)}
Binary semaphore

- **Semaphore S values**
  - S=1 : resource in use
  - S=0 : resource not in use

- **Semaphore S actions**
  - `wait(&S)` : test & set (read S, set S=1)
    - use resource if S was read as 0
    - wait if S was read as 1
  - `signal(&S)` : write S=0 to free up resource
Example

- **Access critical region**
  
  ```
  wait(&S); //continue if read S=1, o/w wait
  //execute “critical region”
  signal(&S); //free the resource
  ```

- **Task synchronization**
  
  Task1          Task2
  signal(&S1)    signal(&S2)
  wait (&S2)     wait(&S1)

  tasks synchronize at this point
Counting semaphore

- Semaphore S values
  - S=1 : resource free
  - S=0 : resource in use, no others waiting
  - S<0 : resource in use, others waiting

- Semaphore S actions
  - wait(&S) : S--, use resource if S=0, o/w wait
  - signal(&S) : S++, wake up other task if S<1

Also use for access to N copies of a resource – semaphore indicates number of copies free
Potential deadlock

- Tasks 1 and 2 each require two resources, R1 and R2, with access controlled by S1 and S2, respectively.

<table>
<thead>
<tr>
<th>Task1</th>
<th>Task2</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait(&amp;S1)</td>
<td>wait(&amp;S2)</td>
</tr>
<tr>
<td>//have R1</td>
<td>//have R2</td>
</tr>
<tr>
<td>wait (&amp;S2)</td>
<td>wait(&amp;S1)</td>
</tr>
<tr>
<td>//wait for R2</td>
<td>//wait for R1</td>
</tr>
</tbody>
</table>

DEADLOCK!!
CMSIS-RTOS Mutual Exclusion (MUTEX)

“Binary” Semaphore: mutually-exclusive access to a resource

- `osMutexDef(m1);` //Macro: MUTEX object definition
- `osMutexId m_id;` //MUTEX ID
- `m_id = osMutexCreate(osMutex(m1));` //create MUTEX obj
- `status = osMutexDelete(m_id);` //delete MUTEX obj
- `status = osMutexWait(m_id, timeout);`
  - Wait until MUTEX available or until time = “timeout”
    - `timeout = 0` to return immediately
    - `timeout = osWaitForever` for infinite wait
  - “status” = `osOK` if MUTEX acquired
    - `osErrorTimeoutResource` if not acquired within timeout
    - `osErrorResource` if not acquired when timeout=0 specified
- `status = osMutexRelease(m_id);` //release the MUTEX

Timeout arguments for other objects have same options
CMSIS-RTOS Semaphores

“Counting” Semaphore: allow up to \( t \) threads to access a resource

- \texttt{osSemaphoreDef(s1)} //Macro: define SEMAPHORE object \( s1 \)
- \texttt{osSemaphoreId s\_id;} //SEMAPHORE ID
- \( s\_id = \text{osSemaphoreCreate(osSemaphore(s1, t))}; \)
  - Create \( s1 \) and set initial \#tokens = \( t \)
- \( \text{ntok = osSemaphoreWait(s\_id,timeout);} \)
  - Wait until token available, or until timeout
  - Return \#tokens that were available
  - If \( \text{ntok >0, token is obtained and \#tokens is decremented} \)
  - If \#tokens=0, no token was available at timeout
- \texttt{osSemaphoreRelease(s\_id);} //increment \#tokens in \( s1 \)

- \texttt{osSemaphoreDelete(s\_id);} //delete the semaphore
CMSIS-RTOS semaphore example

```c
osSemaphoreId sid_Thread_Semaphore; // semaphore id
osSemaphoreDef (SampleSemaphore); // semaphore object

// Main thread: Create the semaphore
sid_Thread_Semaphore = osSemaphoreCreate(osSemaphore(SampleSemaphore), 1);
if (!sid_Thread_Semaphore) { // Semaphore object not created, handle failure }

// Application thread: Acquire semaphore; perform task; release semaphore
val = osSemaphoreWait (sid_Thread_Semaphore, 10); // wait up to 10 mSec
switch (val) {
  case osOK: //Semaphore acquired
    // Use protected code here...
    osSemaphoreRelease (sid_Thread_Semaphore); // Return token back to a semaphore
    break;
  case osErrorTimeoutResource: break; // Not acquired within timeout
  default: break;
}
```
POSIX semaphores

- POSIX supports counting semaphores with `_POSIX_SEMAPHORES` option.
  - Semaphore with N resources will not block until N processes hold the semaphore.
- Semaphores are given name:
  - Example: `/sem1`
- P() is `sem_wait()`
- V() is `sem_post()`
Semaphore example (1)

```c
int i, oflags;
sem_t *my_semaphore; //descriptor for sem.

//create the semaphore
my_semaphore = sem_open("/sem1", oflags);
    /* do useful work here */

//destroy the semaphore if no longer needed
i = sem_close(my_semaphore);
```
Semaphore example (2)

```c
int i;

i = sem_wait(my_semaphore);  // P()
// wait for semaphore, block if not free
// now do useful work
i = sem_post(my_semaphore);  // V()

// test without blocking
i = sem_trywait(my_semaphore);
```
Signals

- Originally, a Unix mechanism for simple communication between processes.
- Analogous to an interrupt---forces execution of a process at a given location.
  - But a signal is generated by one process with a function call.
- No data---can only pass type of signal.
CMSIS-RTOS Signal Flags

- Signal flags not “created” – each thread automatically includes 32 signal flags (SFs) – in one 32-bit word.

  | SF<sub>31</sub> | .... | SF<sub>2</sub> SF<sub>1</sub> SF<sub>0</sub> |

- Signals are sent to a thread (using its thread ID)
- `osSignalSet(tid, flags)` – set SFs of thread tid
- `osSignalClear(tid, flags)` – clear SFs of thread tid
  - `flags = int32_t;` each “1” bit in “flags” sets/clears the corresponding SF
  - Example: `flags=0x8002` => set/clear SF #15 and SF #0
  - Return `int32_t`, containing previous flags of tid

- `osSignalWait(flags, timeout)`
  - Wait for SFs corresponding to “1” bits in “flags” to be set, or until timeout
    - `timeout = 0` if no wait time desired
  - Return `osEventSignal` if designated SFs are set
    - `osEventTimeout` if no signal before timeout
    - `osOK` if `timeout=0` and no signal
// Thread 1
void ledOn (void constant *argument) {
    for (;;) {
        LED_On(0);
        osSignalSet(tid_ledOff, 0x0001);  // signal ledOff thread
        osDelay(2000);
    }
}

// Thread 2
void ledOff (void constant *argument) {
    for (;;) {
        osSignalWait(0x0001, osWaitForever);  // wait for signal from ledOn thread
        osDelay(500);
        LED_Off(0);
    }
}
POSIX signal types (partial list)

- SIGABRT: abort process
- SIGTERM: terminate process
- SIGFPE: floating point exception
- SIGILL: illegal instruction
- SIGKILL: unavoidable process termination
- SIGALRM: real-time clock expired
- SIGUSR1, SIGUSR2: user defined
POSIX signals

- Must declare a signal handler for the process using `sigaction()`.
  - what to do when signal received
  - handler is called when signal is received
    ```c
    retval=sigaction(SIGUSR1,&act,&oldact);
    ```

- Send signal with `sigqueue()`:
  ```c
  sigqueue(destpid,SIGRTMAX-1,sval);
  ```
Message passing

- Message passing on a network:
Message passing via mailboxes

- Mailbox = message buffer between two processes (FIFO)

Use semaphore to lock buffer during read/write
A message is one 32-bit value or pointer

- **osMessageQId** `q_id`;  
  // ID of queue object

- **osMessageQDef (name, queue_size, type);**  
  //Macro: define message queue object

  - `queue_size` = max #messages in the queue
  - `type` = 32-bit data type of message (32-bit integer or pointer)

- `q_id = osMessageCreate( osMessageQ(name), NULL);`
  Create and initialize a message queue, return queue ID

- `status = osMessagePut(q_id, msg, timeout );`
  Add “msg” to queue; wait for “timeout” if queue full
  Status = **osOK** : msg was put into the queue
  
  - **osErrorResource** : no queue memory available
  - **osErrorTimeoutResource** : no memory available at timeout

- `evt = osMessageGet(q_id, timeout);`  
  (evt is type osEvent - contains status & value)

  - Get message from queue; wait for “timeout” if no message
  - `evt.status = osOK` : no msg available and timeout=0
    - **osEventTimeout** : no message available before timeout
    - **osEventMessage** : msg received  
      (“evt” is “union” structure – value can be ptr or value)

  - `evt.value.p` = 32-bit “value” is a pointer to information
  - `evt.value.v` = 32-bit “value” is the data
CMSIS-RTOS message queue example

// “Message” will be a 32-bit data value
osMessageQId qid_MyData;        // message queue id

// Message queue macro definition for MyData;
osMessageQDef (MyData, MSGQUEUE_OBJECTS, uint32_t);

// Thread 1 code
uint32_t n;
… n = something;
osMessagePut (qid_MyData, n, osWaitForever);  //send n as msg
…

// Thread 2 code
osEvent msg; //pointer to an “event” structure
msg = osMessageGet(qid_MyData, 0)    //return immediately if no message
if (msg.status == osEventMessage)        //was there a message
    m = msg.value.v; //process its data

// Main creates threads, message queues, etc.
int main (void )
{
    qid_MyData = osMessageCreate(osThread(MyData),NULL);
    …
CMSIS-RTOS message queue example

// “Message” will be a pointer to a data structure
typedef struct {
    // object data type
    uint8_t Buf[32];
    uint8_t Idx;
} osMessageQId mid_MsgQueue;    // message queue id
osMessageQDef (MsgQueue, MSGQUEUE_OBJECTS, MSGQUEUE_OBJ_t);

// Allocate memory for the message structure
mpid_MemPool2 = osPoolCreate (osPool (MemPool2));      // create Mem Pool
if(!mpid_MemPool2) {
    ; // MemPool object not created, handle failure
}

mid_MsgQueue = osMessageCreate(osMessageQ(MsgQueue), NULL);  // create msg queue
if(!mid_MsgQueue) {
    ; // Message Queue object not created, handle failure
}

Continued on next slide
void Thread_MsgQueue1(void const *argument) {
    MEM_BLOCK_t *pMsg = 0;
    while(1) {
        // Insert thread code here...
        pMsg = (MEM_BLOCK_t *)osPoolCAlloc (mpid_MemPool2); // get Mem Block
        if(pMsg) { // Mem Block was available
            pMsg->Buf[0]  = 0x55; pMsg->Idx = 0; // Fill in the data
            osMessagePut(mid_MsgQueue, (uint32_t)pMsg, osWaitForever); // Send Message
        }
        osThreadYield(); // suspend thread
    }
}

void Thread_MsgQueue2(void const *argument) {
    osEvent evt;
    MEM_BLOCK_t *pMsg = 0;
    while(1) {
        // Insert thread code here...
        evt = osMessageGet(mid_MsgQueue, osWaitForever); // wait for message
        if (evt.status == osEventMessage) {
            pMsg = evt.value.p;
            if(pMsg) {
                // process data
                osPoolFree(mpid_MemPool2, pMsg); // free memory allocated for message
            }
        }
    }
}
CMSIS-RTOS Mail queues

- \texttt{osMailQId \ q\_id;} \quad // \text{ID of \textit{mail queue} object}
- \texttt{osMailQDef (name, queue\_size, type);} \quad // \text{Macro: mail queue name, \# entries, mail type}
- \texttt{\ q\_id = osMailCreate( osMailQ(name), NULL);}  
  \hspace{1cm} // \text{Create and initialize a message queue, return queue ID}
- \texttt{mptr = osMailAlloc(q\_id, timeout);} \quad (\texttt{osMailCAloc()} – \text{allocate and clear memory})
  \hspace{1cm} // \text{Allocate a memory block in the queue that can be filled with mail info}
  \hspace{1cm} // \text{“mptr” = pointer to the memory block (NULL if no memory can be obtained)}
  \hspace{1cm} // \text{Wait, with timeout, if necessary for a mail slot to become available}
- \texttt{status = osMailFree(q\_id, mptr);} \quad // \text{free allocated memory}

- \texttt{status = osMailPut(q\_id, mptr);}  
  \hspace{1cm} // \text{Add mail (pointed to by mptr) to queue; wait for “timeout” if queue full}
  \hspace{1cm} // \text{Status = osOK : mail was put into the queue}
  \hspace{1cm} \hspace{1cm} = osErrorValue : mail was not allocated as a memory slot
- \texttt{status = osMailGet(q\_id, timeout);}  
  \hspace{1cm} // \text{Get mail from queue; wait for “timeout” if no mail available}
  \hspace{1cm} // \text{Status = osOK : no mail available and timeout=0}
  \hspace{1cm} \hspace{1cm} = osEventTimeout : no mail available before timeout
  \hspace{1cm} \hspace{1cm} = osEventMail : mail received, pointer = value.p
MicroC/OS-II Mailboxes

- **OSMboxCreate(msg)**
  - create mail box & insert initial msg
- **OSMboxPost(box, msg)**
  - add msg to box
- **OSMboxAccept(box)**
  - get msg if there, o/w continue
- **OSMboxPend(box, timeout)**
  - get msg if there, o/w wait up to timeout
- **OSMboxQuery(box,&data)**
  - return information about the mailbox
Example: Mentor Graphics
“Nucleus” POSIX Kernel
Nucleus POSIX demo program
(Mentor Graphics EDGE tools for SoC/ARM)

- Six tasks/five “functions”
  - a system timer
  - a task that queues messages
  - a task that retrieves queued messages
  - two instances of a task that uses a resource for 100 “ticks”
  - a task that waits for event signals
Demo program: Application structures

#include "nucleus.h" /* OS function def's */

/* Define Nucleus objects */

NU_TASK Task_0;
NU_TASK Task_1;
NU_TASK Task_2;
NU_TASK Task_3;
NU_TASK Task_4;
NU_TASK Task_5;
NU_QUEUE Queue_0;
NU_SEMAPHORE Sempahore_0;
NU_EVENT_GROUP Event_Group_0;
NU_MEMORY_POOL System_Memory;
Demo program: Global variables

/* Define demo global variables */
UNSIGNED Task_Time;
UNSIGNED Task_1_messages_sent;
UNSIGNED Task_2_messages_received;
UNSIGNED Task_2_invalid_messages;
NU_TASK *Who_has_the_resource;
UNSIGNED Event_Detections;
Nucleus POSIX process/task creation (done by startup procedure)

/* Create a system memory pool to allocate to tasks */
status = NU_Create_Memory_Pool(
    &System_Memory,          //pointer to sys memory
    "SYSMEM",                //name
    first_available_memory,  //address of 1st location
    SYSTEM_MEMORY_SIZE,      //size of allocated memory
    50,                      //minimum allocation
    NU_FIFO                  //if memory not available,
                          //resume tasks in FIFO order
);
Nucleus POSIX process/task creation (done by startup procedure)

/* Create task 0 */
//allocates memory for task stack from memory pool
NU_Allocate_Memory(&System_Memory, &pointer,
    TASK_STACK_SIZE, NU_NO_SUSPEND);

//create task activation record
status = NU_Create_Task(&Task_0, "TASK 0", task_0, 0,
    NU_NULL, pointer, TASK_STACK_SIZE, 1, 20,
    NU_PREEMPT, NU_START);
Nucleus POSIX process/task creation
(done by startup procedure)

/* Create task 1 - Queue sending task*/
NU_Allocate_Memory(&System_Memory, &pointer,
                     TASK_STACK_SIZE, NU_NO_SUSPEND);

status = NU_Create_Task(&Task_2, "TASK 2", task_2, 0,
                         NU_NULL, pointer, TASK_STACK_SIZE, 10, 5,
                         NU_PREEMPT, NU_START);

/* repeat for tasks 2-5 */
void task_0()
{
    Task_Time = 0;
    while (1) {
        NU_Sleep (100); /*suspend for 100 ticks */

        Task_Time++;    /* increment time */

        /* set event flag to lift suspension of Task 5 */
        status = NU_Set_Events(&Event_Group_0,1,NU_OR);
    }
}
Demo program: Queue-sending task

```c
void task_1( )
{
    Send_Message = 0;
    while (1) {
        /* queue a message */
        /* suspend if queue full or time slice */
        status = ND_Send_To_Queue(&Queue_0,
                                    &Send_Message, 1, NU_SUSPEND);
        Send_Message++;
    }
}
```
void task_2( )
{
    message_expected = 0;
    while (1) {
        /* retrieve a message */
        /* suspend if queue empty or time slice */
        status = ND_Receive_From_Queue(&Queue_0,
                                        &Receive_Message, 1, &received_size,
                                        NU_SUSPEND);
        message_expected++;
    }
}
Demo program: Use resource task
(two instances in the demo)

/* two tasks compete for use of a resource */
void tasks_3_and_4( )
{
    while (1) {
        /* set semaphore to lock resource */
        status = ND_Obtain_Semaphore(&Semaphore_0,
                                         NU_SUSPEND);

        if (status == NU_SUCCESS) {
            Who_has_resource = ND_Current_Task_Pointer();
            /* hold resource 100 ticks to suspend other task */
            NU_Sleep (100);
            NU_Release_Semaphore (&Semaphore_0);
        }
    }
}
Demo program: Wait for event to be set by Task 0

```c
void task_5( )
{
    event_detections = 0;
    while (1) {
        /* wait for event and consume it */
        status = ND_Retrieve_Events(&Event_Group_0, 1,
                                    NU_OR_CONSUME, &event_group, NU_SUSPEND);
        if (status == NU_SUCCESS) {
            Event_Detections++;
        }
    }
}
```