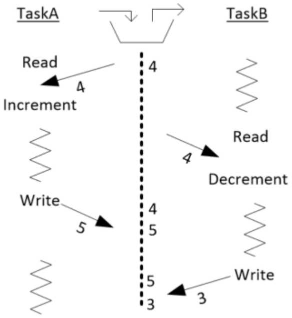
ECEN 3753: Real-Time Operating Systems

Inter-Task Communication

When Tasks Collide

- A single piece of HW can be accessed by both Task1 and Task2
  - The HW is a shared resource
- Task1 wants to write to the HW
  - . . . but halfway through making its updates, it gets swapped out for Task2
- Task2 also wants to write to the same HW
  - . . . and finishes its operation
- What state is the HW in?
- Is this behavior deterministic?

Tasks Collide: Illustrative Example



Types of Shared Resources

- Memory
  - Instances of data structures in RAM or ROM
    - Heap data
    - Global data
    - Static data within a function (non-reentrant)
  - Memory mapped peripherals (ADC, GPIO, communication ports, etc.)
    - Could be local to single CPU or shared across CPUs
- Files
- On-Board Peripherals
  - Shared across CPUs in multi-processing environment
  - DMA engine
- External Hardware Devices
  - Shared across a communication link or network (printer, motor, valve, switch, ...)
- Others? Counter-examples, by design?

It Is Not Just About Resources

Sometimes, tasks that are responsible for different parts of a algorithm must synchronize or coordinate their activities.

- a Rubic's cube solver with different motors for each axis cannot perform pre-computed operations
  - at the same time
  - in the wrong order

and expect to get the results you want.

Assembly robots, too!

Two Approaches For Resolving Conflicts

IPC: Allow entities to communicate with each other to minimize direct conflict (Inter-Process (or Task) Communication)

- Task1 may access a resource. (could be **SW Resource Mgr**)
- Task2 does not access the same resource directly.
  - If Task2 wants to access the resource, it signals Task1 to operate the resource appropriately. E.g. UI tasks typically don't touch HW outside the UI–they signal some other task to do what the user requested.

Mutual Exclusion: Prevent multiple tasks from accessing a shared resource simultaneously Scheduling is essentially a MutEx for the CPU.

- 1st task takes exclusive ownership of the shared resource
  - No other tasks are allowed to access the resource until this task indicates that it is 'done' with its updates.
- 2nd task may try to access the resource but must wait (or block) until the first task finishes.

Inter-Task (Process) Communication

- Overview
- Condition Flag
- Semaphore
- Event
- Signal
- Pool
- Message Queue
- Mailbox

Coordination vs Synchronization

- Coordination
  - Tasks execute in correct order
  - Meet specific condition
  - Ignore timing exactness
  - Tasks don't wait for each other
- Synchronization
  - Require timing exactness
  - In correct order
  - Meet specific condition
  - Task waits for other task

Overview

- Communication w/o Data Transfer
  - Coordination
    - Condition Flag
  - Synchronization
    - Semaphore
    - Event
    - Signal
- Data Transfer Only
  - Pool
  - Message Queue
- Synchronization w/ Data Transfer
  - Mailbox

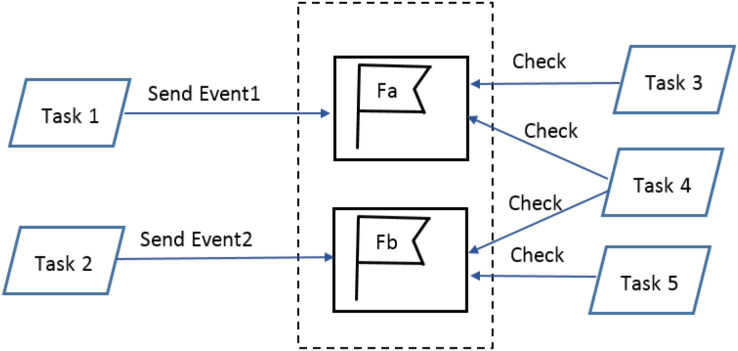
Condition Flag

- One task sets the flag
- Another task reads it, and resets it (after it is read)
- Neither task waits

Condition Flag Group

- Group a set of flags into a single unit (i.e. a word)
- Each flag is a bit within the word
- Flags can be set/cleared with a single instruction
- Groups of bits can be changed using bit masking
- Task may read on a set of flags (OR, AND, etc)
- Task may broadcast flags to many tasks

Condition Flag Group example



Semaphore

- Semaphores are not ‘owned’ by any particular task.
- Semaphores are initialized with an initial counter value.
- Acquiring/Receiving (“**Wait/Pend**-ing”) a semaphore when the counter is:
  - Zero–causes our task to block.
  - Non-zero–decrements the counter, and our task continues to run.
- Releasing/Sending (“**Signal/Post**-ing”) a semaphore may increment the counter, if there are no other tasks waiting for the semaphore.
  - If a task(s) are waiting for the semaphore, the counter remains 0 and at least one waiting task is unblocked.
  - Semaphore options may be set to unblock either ‘one’ or ‘all’ waiting tasks when a semaphore is released.

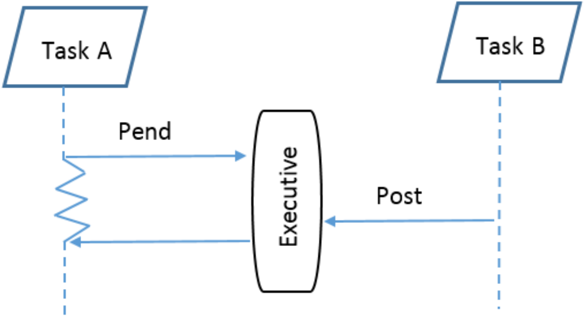
Semaphore

- Used for inter-task communication
- One task reaches certain point, and waits for another task to finish before continue
- Unilateral: Only one task waits
- Can also be used to protect Shared Resource (next lecture), even by use (in "binary" mode) to implement mutual exclusion on shared resources, but are not explicitly associated with data.

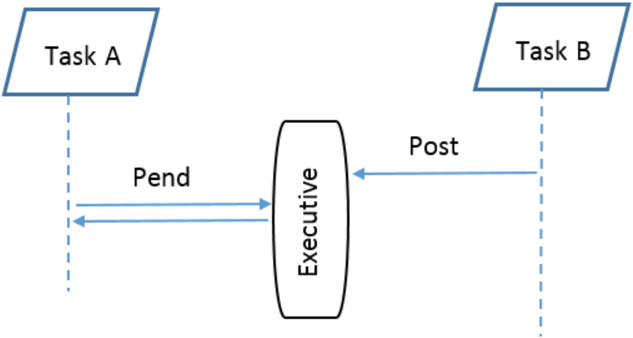
Semaphore flow

- Flag is initialized to clear state
- Receiver task calls Semaphore Pend
  - If flag is set, task continues
  - If flag is cleared, task waits
- Sender task calls Semaphore Post
  - If flag is in set state, just continue
  - If flag is in clear state, set state
    - If Receiver task is waiting, wake it up

Semaphore – Task A (receiver) reaches Pend first



Semaphore – Task B (sender) reaches Post first



Semaphores in Micrium

- Creating a semaphore
  - void OSSemCreate(OS_SEM* p_sem, CPU_CHAR* p_name, OS_SEM_CTR cnt, RTOS_ERR* p_err);
- Acquiring (and possibly waiting for) a semaphore. Calling task may block.
  - OS_SEM_CTR OSSemPend(OS_SEM* p_sem, OS_TICK timeout, OS_OPT opt, CPU_TS* p_ts, RTOS_ERR* p_err);
- Releasing (signaling) a semaphore
  - OS_SEM_CTR OSSemPost(OS_SEM* p_sem, OS_OPT opt, RTOS_ERR* p_err);
    - [opt] can be specified such that on release, either one or all waiting tasks become ready

Semaphore example (uC/OS-II API)

```
OS_EVENT* Isr1Semaphore = null;

/* Create semaphore */
void CreateIsr1Semaphore() {
   Isr1Semaphore = OSSemCreate(1); /* count  = 1 */
}

/* ISR to handle interrupt */
void ISR1() {
    ..... /* do stuff pre-SemPost */
     OSSemPost(Isr1Semaphore); /* semaphore to post */
    ..... /* if more to do */
}
```

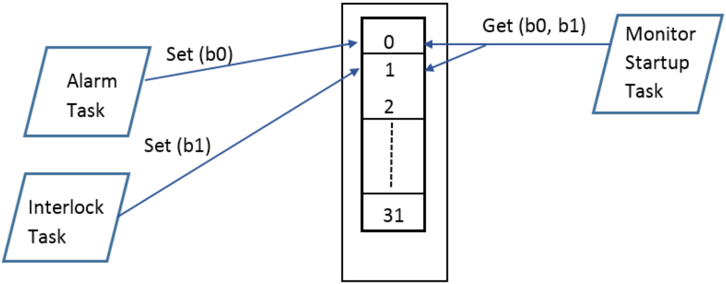
Semaphore example (continued)

```
/* Task: the main flow */
void MyTask() {
    while(1) {
        .....
        /* wait for semaphore to continue */
        OSSemPend(Isr1Semaphore,  /* semaphore to wait */
                  100,            /* timeout */
                  err);           /* error */
        ..... /* do stuff post-ISR critical handling */
    }
}
```

Events

- A series of bits in a word to hold current state of the events in the group
- Similar mechanism as Semaphore on Pend and Post
- Multiple tasks may post different flags
- Waiting task may wait for multiple flags (can be OR-ed, AND-ed)

Event example (from RTOS book)



Event example (from RTOS book, using uC/OS-II API)

```
/* Declaration, global variable on shared memory */
OS_FLAG_GRP* MotorStartEventGroup;
const OS_FLAGS AlarmEvent = 0x1;      /* bit 0 */
const OS_FLAGS InterLockEvent = 0x2;  /* bit 1 */

void main() {
    uint8_t error;
    ......
    MotorStartEventGroup= OSFlagCreate(
                  0,        /* initial value */
                  &error);  /* store error */
    ......
}
```

Event example (Pending)

```
void MotorTask() {
    ......
    OsFlagPend(MotorStartEventGroup, /* Wait on this */
        AlarmEvent + InterLockEvent, /* Events to check */
        OS_FLAG_WAIT_SET_ALL +       /* till all set */
        OS_FLAG_CONSUME,             /* then consume */
        0xFFFF);                     /* time out */
    ......
}
```

Event example (Posting)

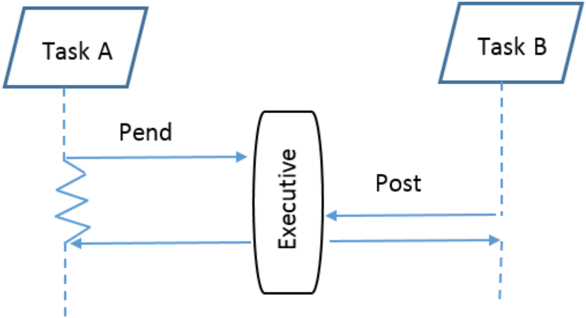
```
void AlarmTask() {
    uint8_t error;
    OSFlagPost(MotorStartEventGroup,  /* post in this */
               AlarmEvent,            /* Event to set */
               OS_FLAG_SET,           /* set bit */
               &error);               /* error */
}

void InterLockTask() {
    uint8_t error;
    OSFlagPost(MotorStartEventGroup,  /* post in this */
               InterLockEvent,        /* Event to set */
               OS_FLAG_SET,           /* set bit */
               &error);               /* error */
}
```

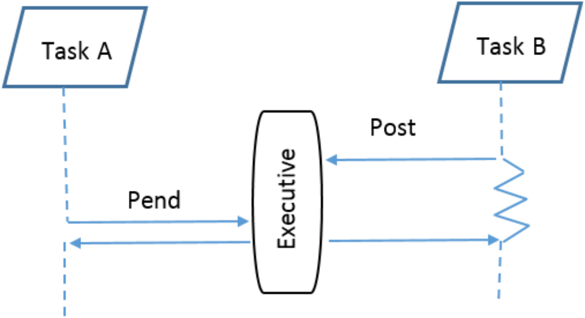
Signal

- Bilateral synchronization
- Both tasks wait for each other
- Both tasks are senders and receivers
- Few RTOS provide signal
    - Can be implemented by two Semaphores

Signal: Task A reaches Pend first



Signal: Task B reaches Post first



Signal example: implemented using Semaphore

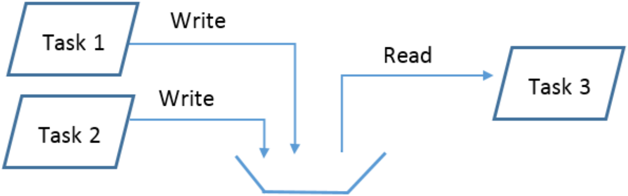
```
OS_EVENT* SemaSync1, SemaSync2;

void main(int argc char* argv[]) {
    .....
    SemaSync1 = OSSemCreate(0);  /* initial value */
    SemaSync2 = OSSemCreate(0);  /* initial value */
    ......
}

void* Task1(void* param) {
    ......
    uint8_t error;
    OSSemPost(SemaSync1);                  /* done w/ my par
    OSSemPend(SemaSync2, 0xFFFF, &error); /* wait for other
    ......
}
```

Pool

- Asynchronous info exchange
- Read-write random access (no requirement on read/write order)
- Non-destructive read
- Data may be protected (so that only one task access it)



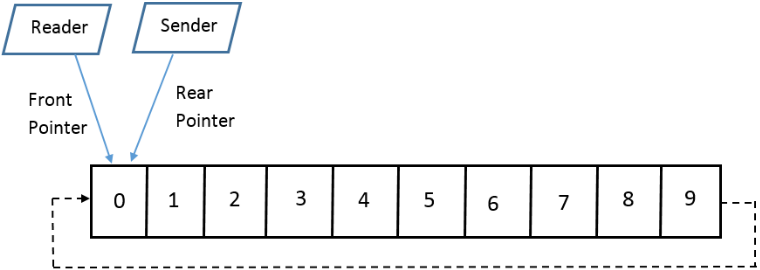
Message Queue

- Other name: channel, buffer, pipe
- Asynchronous access
- One Task writes
- One Task read (destructive read)
- First In First Out
- Can be data or pointer to data
- Can be implemented by linked list or circular buffer

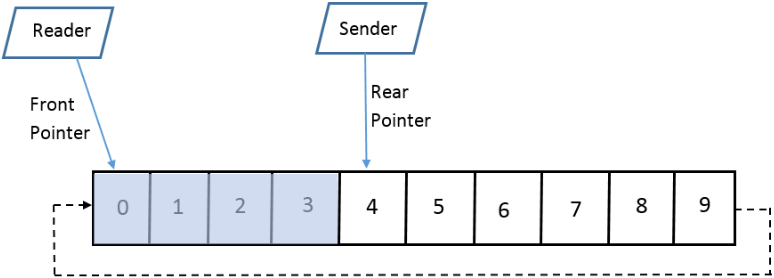
Message Queue: using circular buffer

- Fixed memory space
- Reuse memory space after data is read
- Sender task may pend when buffer is full
- Reader task may pend when buffer is empty

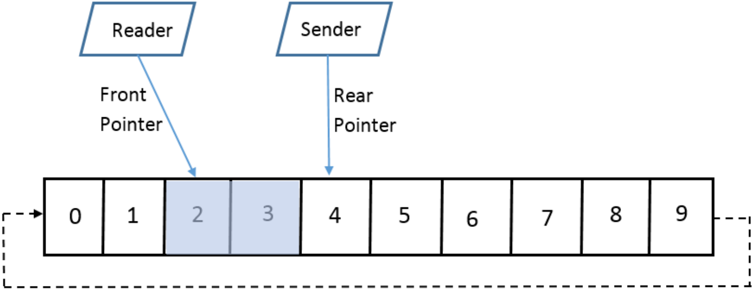
Circular Buffer Queue - Initial State



Circular Buffer Queue - Four Data Sent



Circular Buffer Queue - Two Data Read



Queue: Micrium example

```
/* Initialization Code */
OS_Q  App_QUSB;

void main() {
    ......
    OSQCreate((OS_Q     *)&App_QUSB,
              (CPU_CHAR *)"USB Queue",
              (OS_MSG_QTY)20,
              (RTOS_ERR *)&err);
    ......
}
```

What exactly does the OS need to do in OSQCreate?

Queue example (continued)

```
void* Task1() {
    ......
    p_buf = Get Buffer from pool;
    OSQPost((OS_Q       *)&App_QUSB,
            (void       *)p_buf,
            (OS_MSG_SIZE)buf_size,
            (OS_OPT      )OS_OPT_POST_FIFO,
            (RTOS_ERR   *)&err);
    ......
}
```

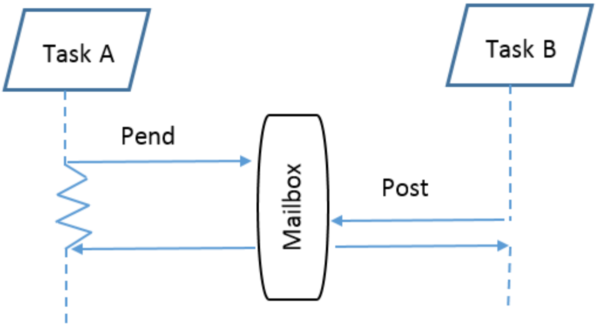
Queue Example (continued)

```
void* Task2() {
    ......
    while (1) {
        p_buf = OSQPend((OS_Q          *)&App_QUSB,
                        (OS_TICK        )0,
                        (OS_OPT)OS_OPT_PEND_BLOCKING,
                        (OS_MSG_SIZE *)&msg_size,
                        (CPU_TS        *)&ts,
                        (RTOS_ERR      *)&err);
        Process packet;
        Free buffer back to pool;
    }
    ......
}
```

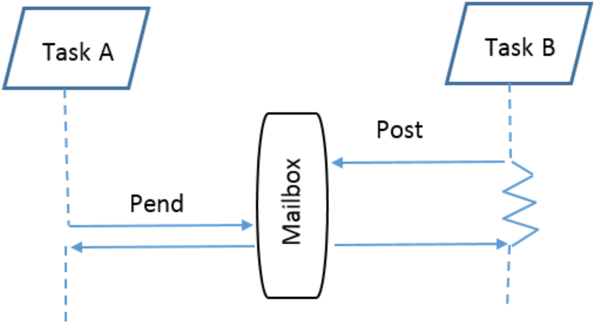
Mailbox

- Task synchronization with Data Transfer
- Signal + Data transfer
- May be many-to-one communication
  - The one can change over time!

Mailbox: Task A Reaches Post First



Mailbox: Task B Reaches Pend First



Mailbox example

```
/* Create a global mailbox*/
OS_EVENT* GlobalMailBoxA2B;

void main() {
    ......
    GlobalMailBoxA2B = OSMboxCreate((void*)0);
    ......
}

void* SenderTask() {
    ......
    uint8_t DataForMboxA2B[50];
    uint8_t err;
    err = OSMboxPost(GlobalMailBoxA2B,
       (void*)&DataForMboxA2B[0]);
    ......
```

Mailbox example (continued)

```
void* ReceiverTask() {
    ......
    void* DataFromMboxA2B;
    uint8_t Wait200 = 200;
    uint8_t err;
    DataFromMboxA2B = OSMboxPend(GlobalMailBoxA2B,
       Wait200, &err);
    ......
}
```

Summary

- Different inter-task communication approaches were covered
  - Coordination and Synchronization w/o data transfer
    - Condition Flag, Semaphore, Event, Signal
    - Semaphore is not covered in RTOS book, but it is widely used
  - Data transfer only: Pool, Message Queue
  - Synchronization w/ data transfer: Mailbox
- Only main APIs for each approach were covered
  - Many other APIs are available from OS, like OSSemDel, OSQFlush, OSQQuery, etc
- Implementation of most approaches were covered
  - If you are writing OS, you may need to implement more