Resource Synchronization

Embedded OS Implementation

Prof. Ya-Shu Chen
National Taiwan University
of Science and Technology

Critical Sections

- A portion of code must be indivisible
  - To protect shared resources from being corrupted due to race conditions
  - Could be implemented by using interrupt enable/disable or IPC mechanisms
    - Semaphores, events, mailboxes, etc

Resources

- An entity used by a task
  - Memory objects
    - Such as tables, global variables …
  - I/O devices.
    - Such as disks, communication transceivers.
- A task must gain exclusive access to a piece of shared resource to prevent data (or I/O status) from being corrupted.
  - Mutual exclusion

Reentrant Functions

Reentrant functions can be invoked simultaneously without corrupting any data.

- Reentrant functions use either local variables (on stacks) or synchronization mechanisms (such as semaphores).

```
void strcpy(char *dest, char *src)
  {
      while (*dest++ = *src++) {
          ;
      }
      *dest = NUL;
  }
```

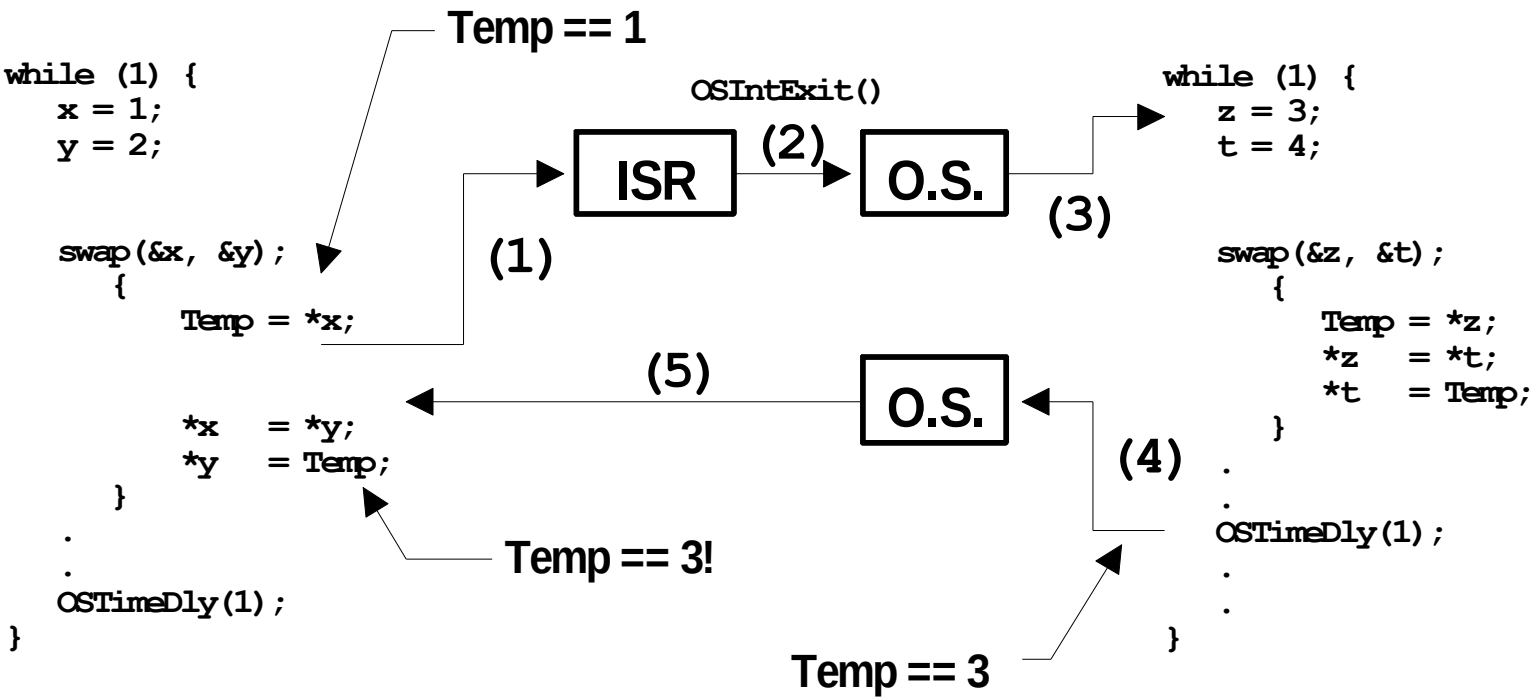
Non-Reentrant Functions

Non-Reentrant functions might corrupt shared resources under race conditions.

```
int Temp;

  void swap(int *x, int *y)
  {
       Temp = *x;
       *x   = *y;
       *y   = Temp;
  }
```

Temp: a global variable



(1) When swap() is interrupted, TEMP contains 1.

(2)

(3) The ISR makes the higher priority task ready to run, so at the completion of the ISR, the kernel is invoked to switch to this task. The high priority task sets TEMP to 3 and swaps the contents of its variables correctly. (i.e., z=4 and t=3).

(4) The high priority task eventually relinquishes control to the low priority task be calling a kernel service to delay itself for one clock tick.

(5) The lower priority task is thus resumed. Note that at this point, TEMP is still set to 3! When the low priority task resumes execution, the task sets y to 3 instead of 1.

Non-Reentrant Functions

There are several ways to make the code reentrant:

- Declare TEMP as a local variable
- Disable interrupts and then enable interrupts
- Use a semaphore

Mutual Exclusion

- Mutual exclusion must be adopted to protect shared resources.
  - Global variables, linked lists, pointers, buffers, and ring buffers.
  - I/O devices.
- When a task is using a resource, the other tasks which are also interested in the resource must not be scheduled to run
- Common techniques
  - disable/enable interrupts
  - a test-and-set instruction
  - disabling scheduling
  - IPC

Mutual Exclusion

Disabling/enabling interrupts:

- OS_ENTER_CRITICAL() and OS_EXIT_CRITICAL()
- All events are masked since interrupts are disabled
- Tasks which do not affect the resources-to-protect are also postponed
- Must not disable interrupt before calling system services

```
void Function (void)
{
    OS_ENTER_CRITICAL();
    .
    .     /* You can access shared data in here */
    .
    OS_EXIT_CRITICAL();
}
```

Mutual Exclusion

Disabling/Enabling Scheduling:

- No preemptions could happen while the scheduler is disabled
- However, interrupts still happen
  - ISR's could still corrupt shared data
  - Once an ISR is done, the interrupted task is always resumed even there are high priority tasks ready
- Rescheduling might happen right after the scheduler is re-enabled
- Higher overheads and weaker effects than enabling/disabling interrupts

```
void Function (void)
{
    OSSchedLock();
    .      /* You can access shared data
    .         in here (interrupts are recognized) */
    OSSchedUnlock();
}
```

Mutual Exclusion

- Semaphores:
  - Provided by the kernel
  - Semaphores are used to:
    - Control access to a shared resource
    - Signal the occurrence of an event
    - Allow tasks to synchronize their activities
  - Higher priority tasks which does not interested in the protected resources can still be scheduled to run
    - Higher concurrency

```
OS_EVENT *SharedDataSem;
void Function (void)
{
    INT8U err;
    OSSemPend(SharedDataSem, 0, &err);
    .     /* You can access shared data
    .        in here (interrupts are recognized) */
    OSSemPost(SharedDataSem);
}
```

Mutual Exclusion

- Semaphores:
  - OSSemPend() / OSSemPost()
  - A semaphore consists of a wait list and an integer counter.
  - OSSemPend:
    - Counter--;
    - If the value of the semaphore <0, the task is blocked and moved to the wait list immediately.
    - A time-out value can be specified .
  - OSSemPost:
    - Counter++;
    - If the value of the semaphore >= 0, a task in the wait list is removed from the wait list.
      - Reschedule if needed.

Mutual Exclusion

- Semaphores:
  - **Three** kinds of semaphores:
    - Rendezvous semaphore (init = 0)
    - Binary semaphore (init = 1)
    - Counting semaphore (init >1)
  - On event posting, a waiting task is released from the waiting queue
    - The highest-priority task is released
    - FIFO (not supported by uC/OS-2)
  - Interrupts and scheduling are still enabled under the use of semaphores.

Example

```
S=0;
boy()
{
        wait(S);
        // eat
}

girl()
{
        // eat
        signal(S);
}
```

Example

- A DMA controller offers 4 DMA channels for simultaneous data transfers.

```
S=4;T=1;c[4];

proc()
{
        wait(S);
        wait(T);

        // pick one unused channel among c[0],c[1],c[2],c[3]
        // setup DMA transfer

        signal(T);
        signel(S);
}
```

Mutual Exclusion

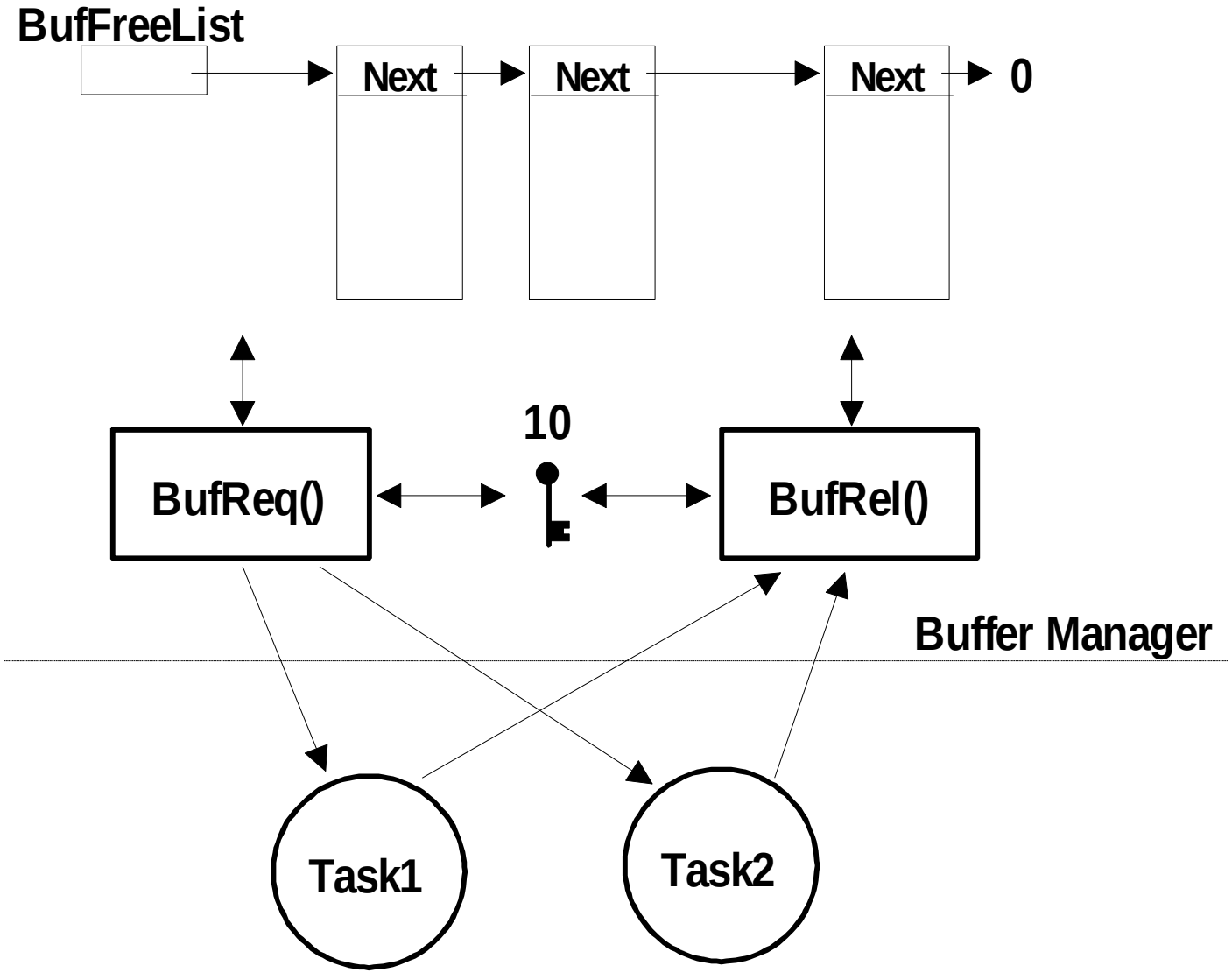
- Using a counting semaphore on a buffer
  - When the buffer is empty, the consumer is blocked
  - Interrput enable/disable is to protect pointers

```
BUF *BufReq(void)
{
   BUF *ptr;

   Acquire a semaphore;
   Disable interrupts;
   ptr        = BufFreeList;
   BufFreeList = ptr->BufNext;
   Enable interrupts;
   return (ptr);
}
```

```
void BufRel(BUF *ptr)
{
   Disable interrupts;
   ptr->BufNext = BufFreeList;
   BufFreeList  = ptr;
   Enable interrupts;
   Release semaphore;
}
```

**Red statements can be replaced by a binary semaphore. Here we disable/enable interrupts for the consideration of efficientcy.

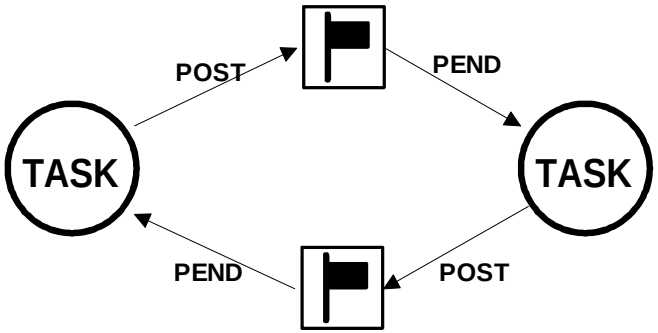

BufFreeList
Next
Next
Next
0
10
BufReq()
BufRel()
Buffer Manager
Task1
Task2

Synchronization

- Different from mutual exclusion, it is much like waiting for an event
- Two semaphores could be used to rendezvous two tasks.

Synchronization

```
Task1()
{
    for (;;) {
            Perform operation;
            Signal task #2;                 (1)
            Wait for signal from task #2;   (2)
            Continue operation;
    }
}
```



** Semaphores are both initialized to 0

```
Task2()
{
    for (;;) {
            Perform operation;
            Signal task #1;                 (3)
            Wait for signal from task #1;   (4)
            Continue operation;
    }
}
```

Summary

- Check list
  - Critical sections
  - Mutual exclusion
  - Synchronization