uC/OS-II Part 6: Resource Synchronization

Real-Time Computing
Prof. Li-Pin Chang
ESSLab@NCTU

- Continued from Part 2: Kernel structure

Critical Sections

- A piece of code that must be atomically executed
  - Must be protected against race conditions
  - Could be implemented using
    - Interrupt enable/disable
    - Scheduler lock/unlock
    - IPC mechanisms such as 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 safely invoked simultaneously by many tasks
  - Use local variables (on stacks)
  - Use synchronization mechanisms (such as semaphores)

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

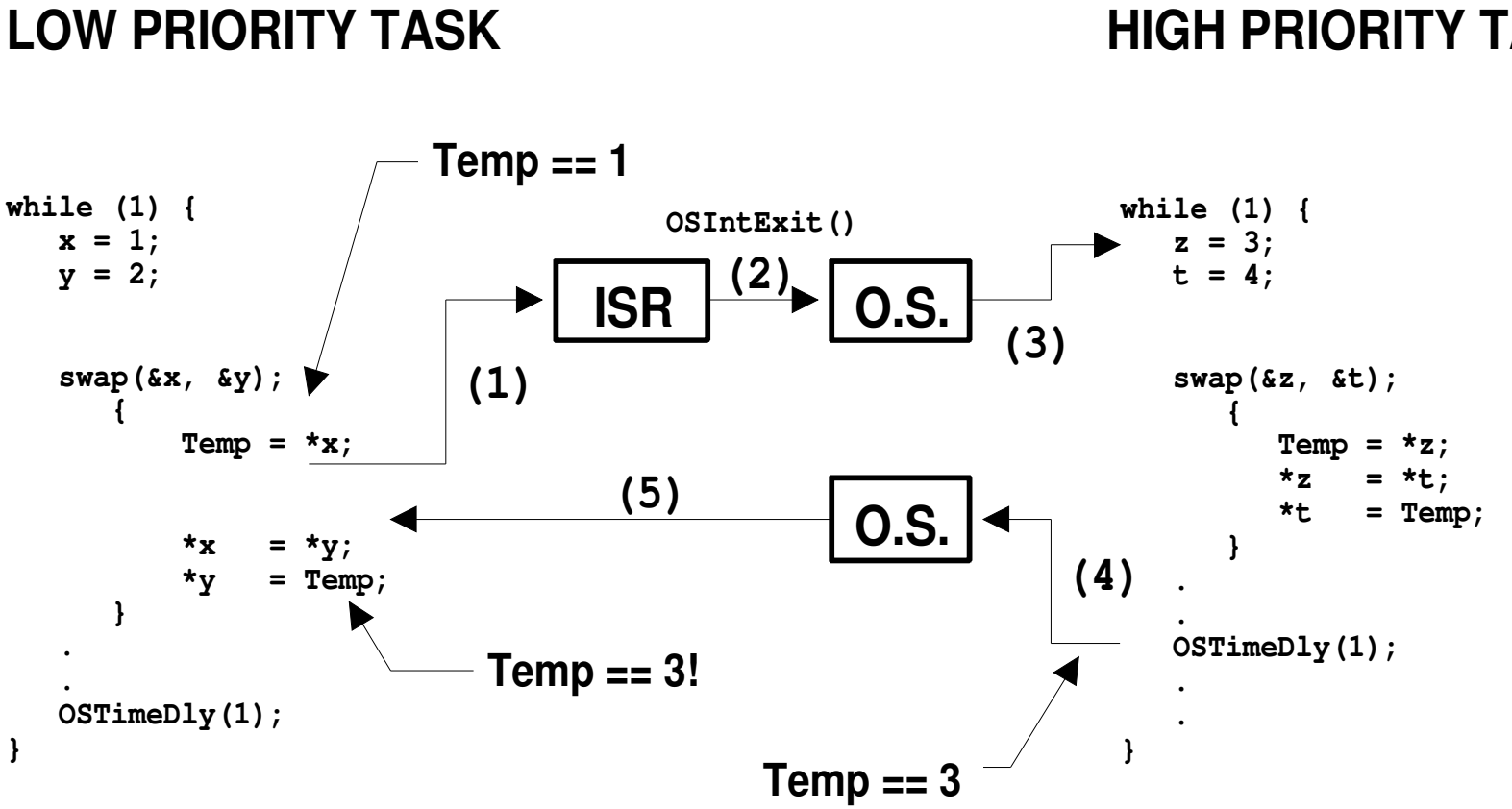
Non-Reentrant Functions

- In non-reentrant functions, shared data might be corrupted because of 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 functions reentrant:
  - Declare TEMP as a local variable
  - Disable interrupts and then enable interrupts
  - Use a semaphore

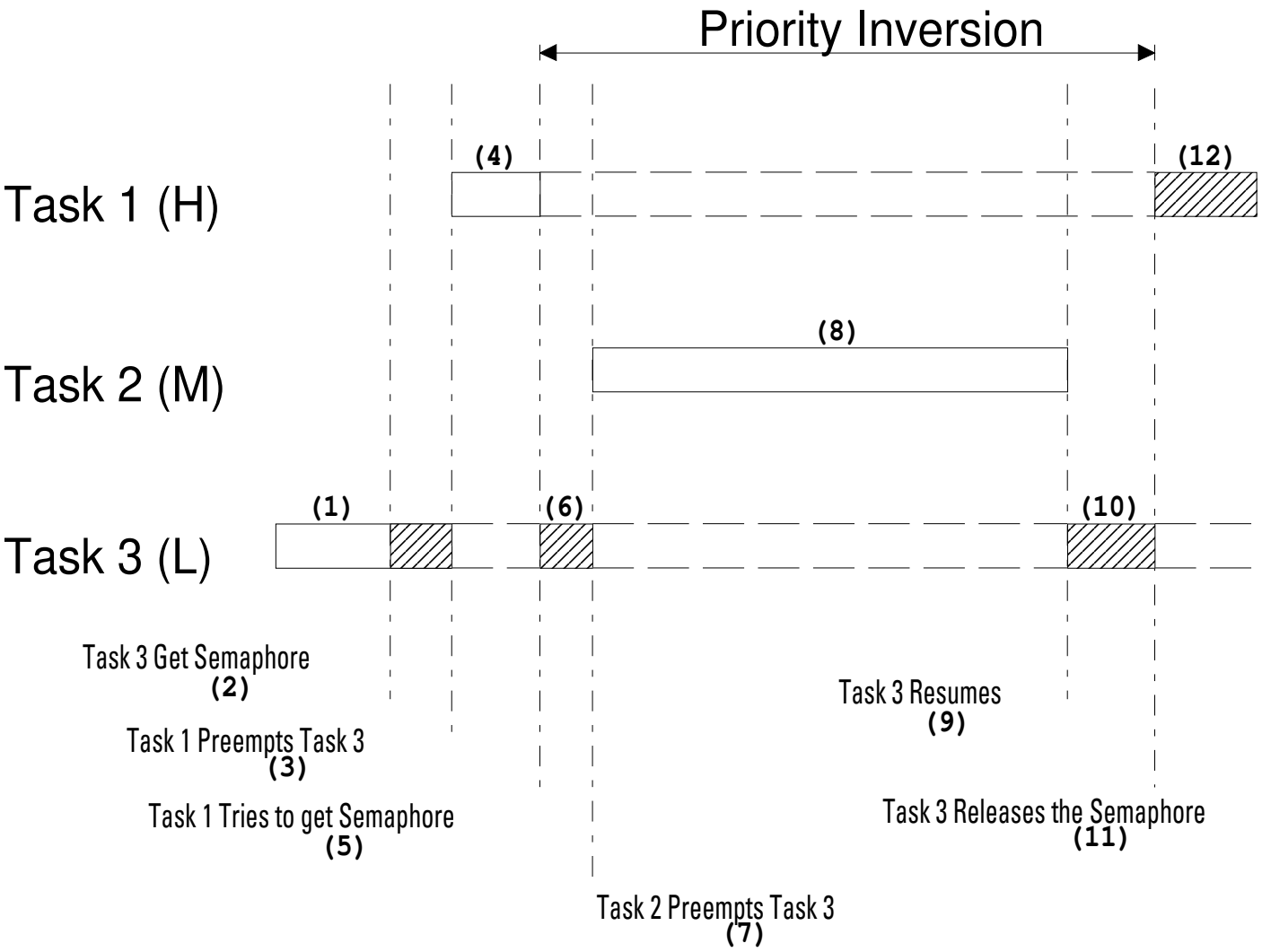
Priority Inversions (1/2)

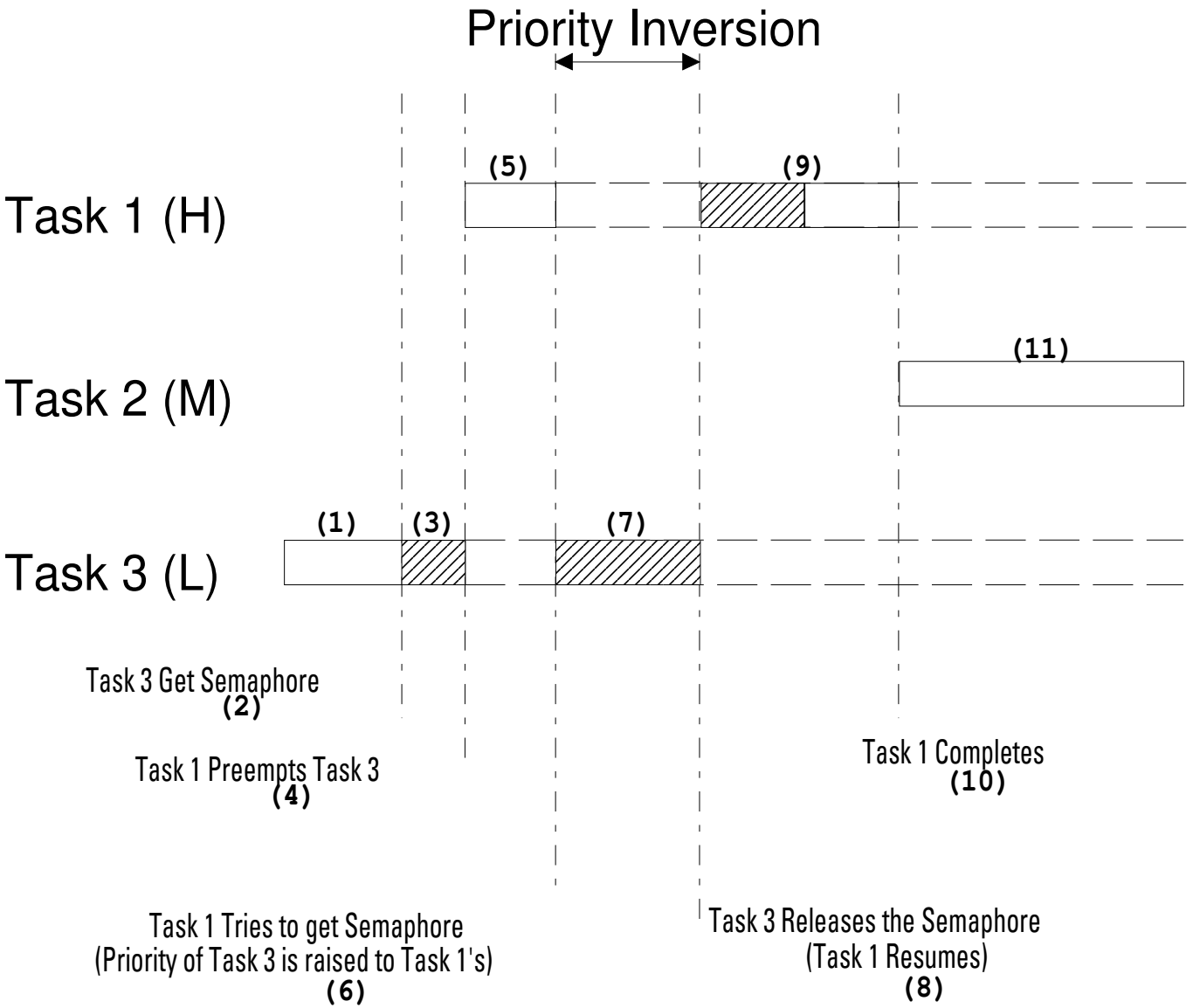
- Priority inversion refers to that a high-priority ready task can not run because of the interference from a low-priority task
- Low-priority tasks won't be "blocked" by high-priority tasks
- To properly control the times and the duration of priority inversion is important
  - Blocking time impose schedulability loss on tasks

$$\forall j, \sum_{i \neq j} \frac{c_i}{p_i} + \frac{c_j + b_j}{p_j} \leq U(n)$$

Priority Inversions (2/2)

- Three undesirable effects
  - Unbounded priority inversions
  - Timing anomalies
  - Deadlocks
- Priority inheritance protocol (PIP) avoids unbounded priority inversions
- Priority ceiling protocol (PCP), which is a super-set of PIP, avoids deadlocks
  - The total blocking time of a task is deterministic


Priority Inversion
Task 1 (H)
Task 2 (M)
Task 3 (L)
(1)
(2)
Task 3 Get Semaphore
(3)
Task 1 Preempts Task 3
(4)
(5)
Task 1 Tries to get Semaphore
(6)
(7)
Task 2 Preempts Task 3
(8)
(9)
Task 3 Resumes
(10)
(11)
Task 3 Releases the Semaphore
(12)


Priority Inversion
Task 1 (H)
Task 2 (M)
Task 3 (L)
(1)
(2)
Task 3 Get Semaphore
(3)
(4)
Task 1 Preempts Task 3
(5)
(6)
Task 1 Tries to get Semaphore
(Priority of Task 3 is raised to Task 1's)
(7)
(8)
Task 3 Releases the Semaphore
(Task 1 Resumes)
(9)
(10)
Task 1 Completes
(11)

Implementation Issues (1/2)

- In practice, PCP is extremely hard to implement
  - Although RTLinux has it
- PIP is much easier to implement
  - WINCE, uC/OS-II, RTAI, FreeRTOS
- NPCS is the easiest
  - Equivalent to scheduler locking upon resource acquisition

Implementation Issues (2/2)

- NPCS
  - Kills response
- PIP
  - A task may suffer from multiple times of priority inversion
- PCP
  - One duration of priority inversion
  - No deadlocks

Mutual Exclusion

- Mutual exclusion is to protect single-instance resource
- Many ways to enforce mutual exclusion
  - Disabling interrupts
  - The test-and-set instruction
  - Lock the scheduler
  - A counting semaphore with init value=1
  - A mutex

Mutual Exclusion

- Disabling/enabling interrupts:
  - OS_ENTER_CRITICAL() and OS_EXIT_CRITICAL()
  - Affect tasks that do not use any resources

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

Mutual Exclusion

- The test-and-set instruction:
  - An atomic operation (a CPU instruction) that locks a guarding variable
  - It is equivalent to the SWAP instruction
  - Waste of CPU cycles on uni-processor systems
  - Starvation is theoretically possible

```
int lock=1;

swap(&flag,&lock);  /* corresponds to SWAP instruction */

if(flag == 1)
   Locking is failed.
   flag remains 1.
else
   Locking is success.
   flag is set as 1 after the swapping.
   ... critical section ...
```

Mutual Exclusion

- Disabling/Enabling the scheduler:
  - No preemptions could happen while the scheduler is disabled
  - Can still handle interrupts
    - But ISR-task race exists
    - Better than interrupt disabling
  - May affect tasks not using any resources

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

Mutual Exclusion

- Semaphores:
  - Provided by the kernel
  - Tasks that are not involved in resource contention won't be affected
    - Good response
    - But higher overhead
  - uCOS-2's semaphores do not manage priority inversion

```
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

- Mutex:
  - Semantically, a binary semaphore
  - Introduced in newer versions of uC/OS-II
  - Priority inversion is managed

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

Mutex vs. Semaphores

- Semaphore with init=1 is functionally identical to mutex lock (in non-real-time systems)
- A subtle difference between mutex and sem=1
  - Priority inheritance (mutex only)
  - Mutex can only be unlocked by its locker

Mutual Exclusion

- Summary:
  - Interrupt disabling is good enough for very short critical sections
  - In consideration of response, semaphore or mutex should be taken
  - In uc/OS-II, prioirty inversion is managed only by mutex locks