uC/OS-II Part 6: Resource Synchronization

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

Critical Sections

- A piece of code that must be executed exclusively to be against races
- A critical section can be implemented by
  - Interrupt enabling/disabling
  - Scheduler locking/unlocking
  - IPC mechanisms, e.g., semaphores, mutex, etc

Resources

- An object shared among tasks
  - Memory objects
    - Global tables, global variables …
  - I/O devices.
    - GPIO port, radio transceivers.

- A task must gain mutually exclusive access to a piece of resource, preventing the resource from being corrupted

Reentrant Functions

- Reentrant functions can be safely invoked multiple tasks at the same time

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

- This function is re-entrant bcz all variables are local

Non-Reentrant Functions

- In non-reentrant functions, shared data might be corrupted by races

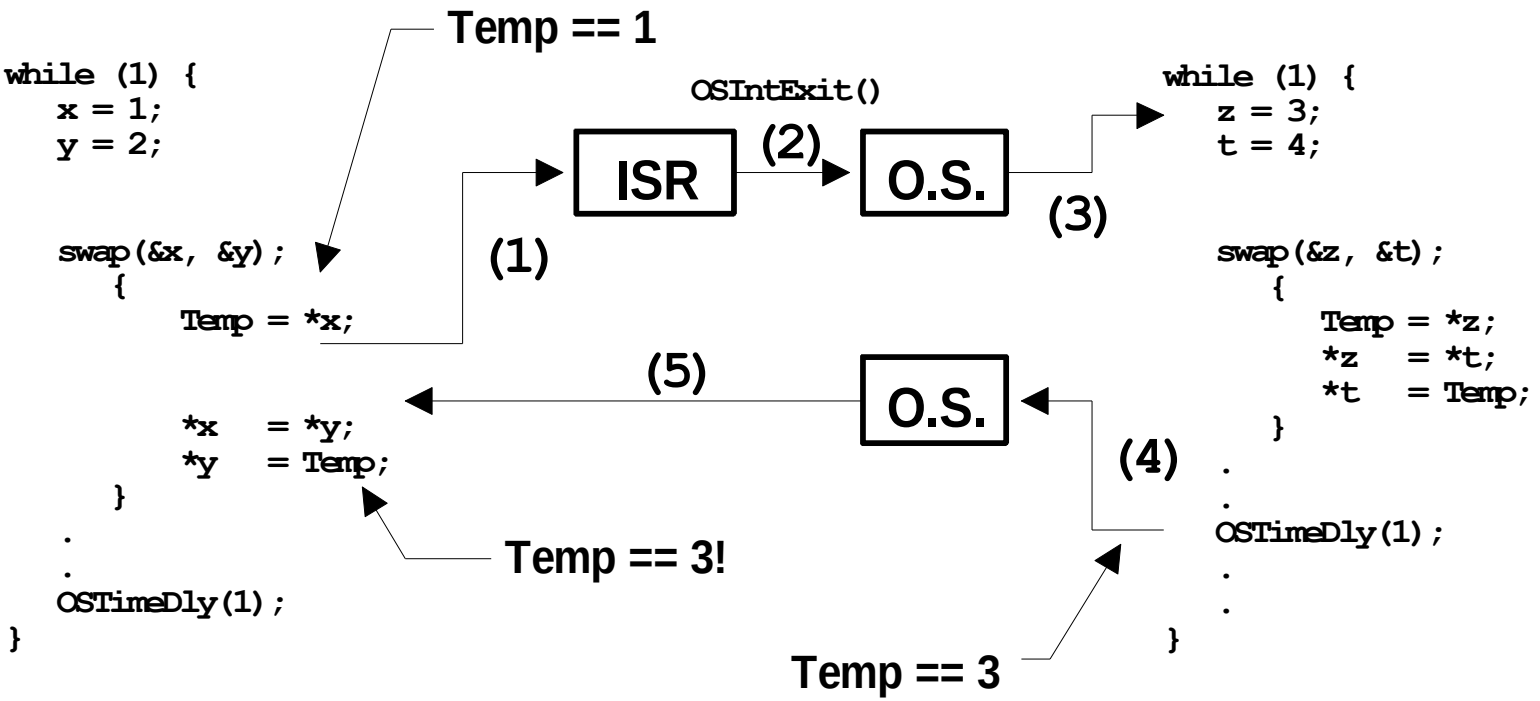
```
int Temp;

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

Temp: a global variable

LOW PRIORITY TASK

HIGH PRIORITY TASK



(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

- Solutions
  - Declare TEMP as a local variable is a quick solution
  - Otherwise, we can make swap() mutually exclusive

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()
  - High-priority tasks suffer

```
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 (and cache coherent) CPU instruction; variants include CAS (compare-and-swap) & XCHG
  - Wasting of CPU cycles on uni-processor systems

```
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 preemption if the scheduler has been locked
  - Interrupts are still accepted, so ISR-task race remains
  - Still impacting on high-priority tasks

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

Mutual Exclusion

- Semaphores (initialized to 1):
  - Supported by uC/OS2
  - 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
  - Supported by uC/OS-II
  - It implements priority inheritance to mange priority inversion

```
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

Priority Inversions (1/2)

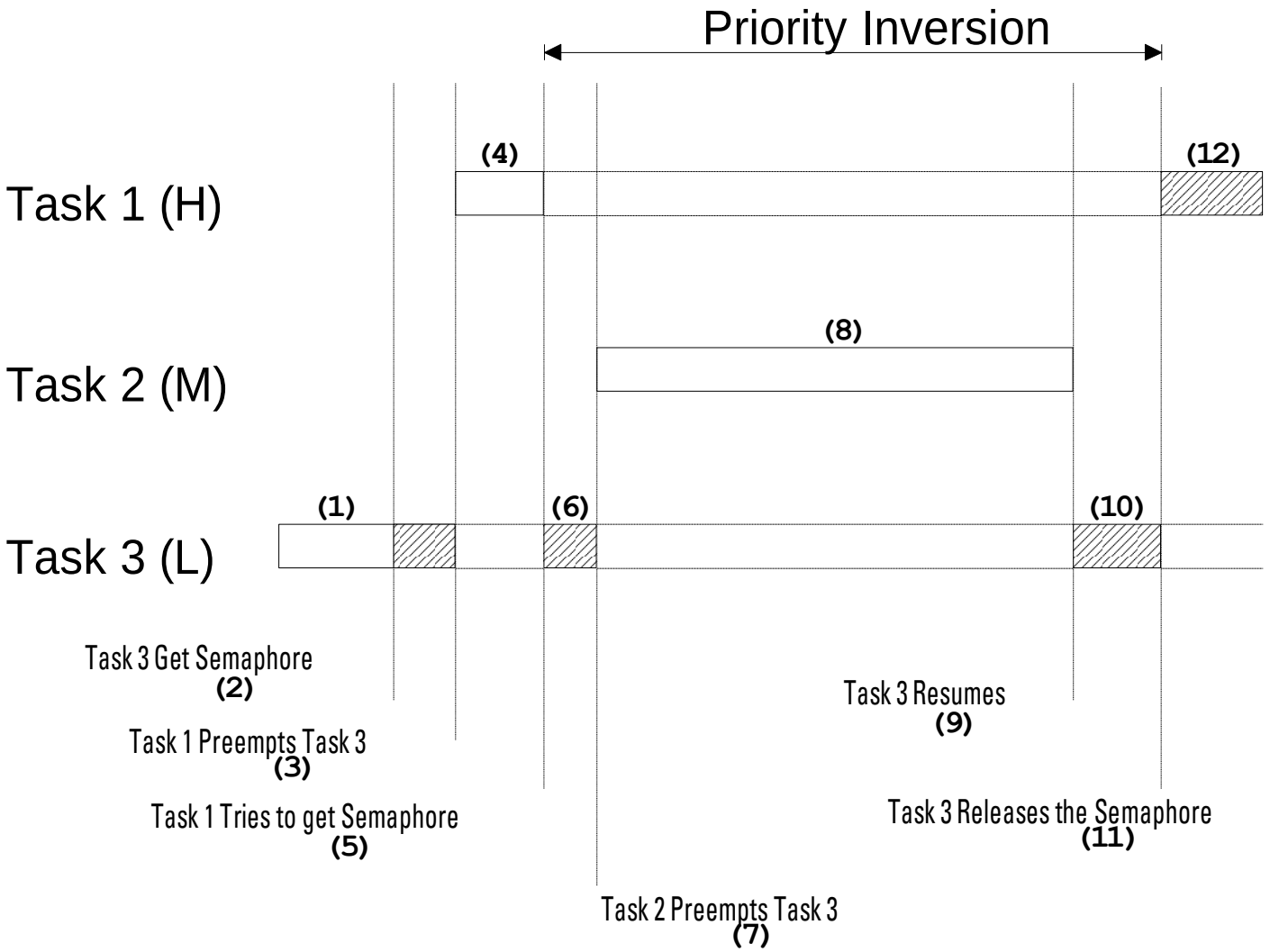
A high-priority ready task can not run because of resource contention from a low-priority task

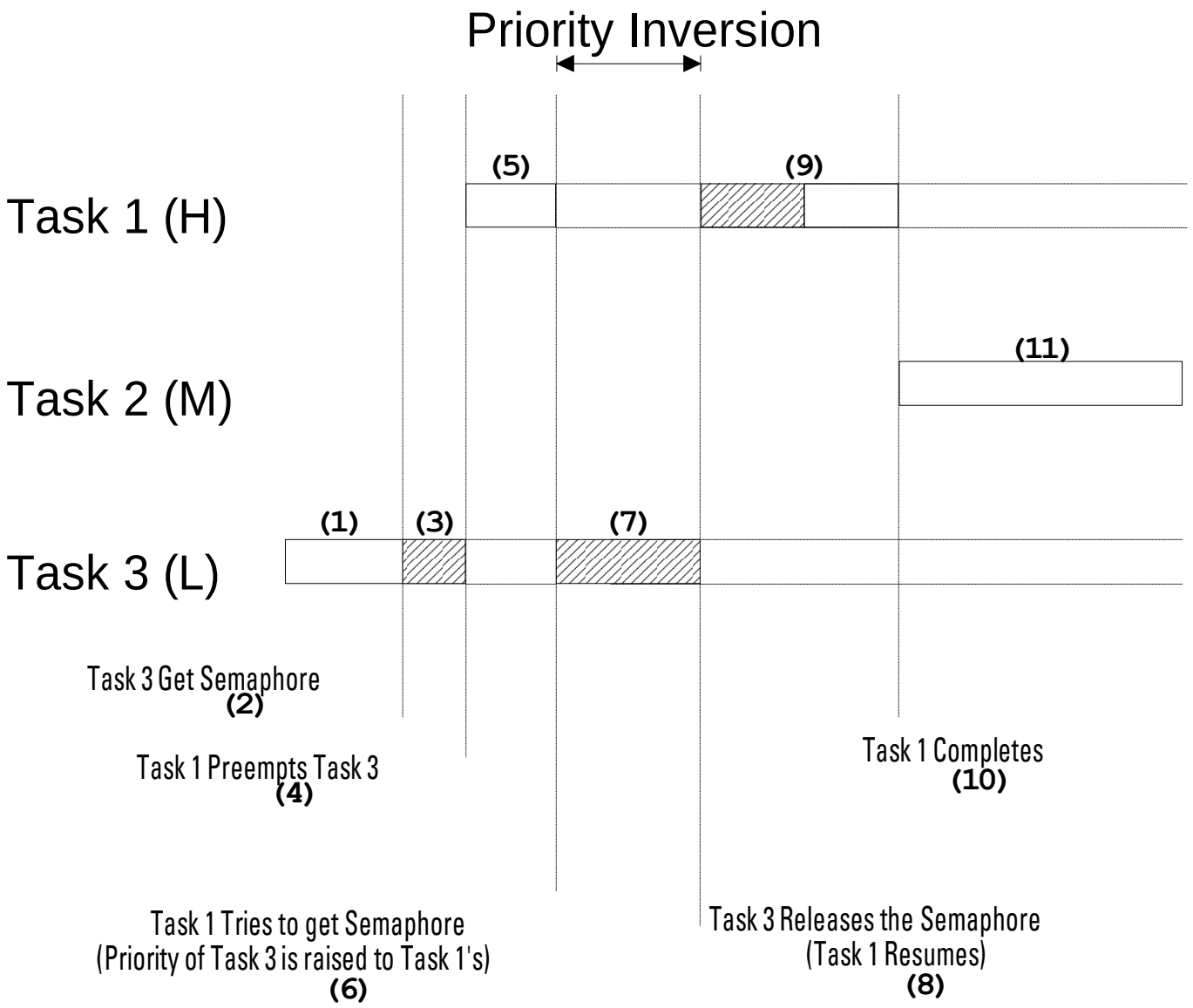
- Low-priority tasks won't be "blocked" by high-priority tasks; they are "preempted"
- Essential to manage the blocking time length, as blocking time is an extra cost

$$\forall_{i=1\ldots n} \left\{ \frac{b_i}{p_i} + \sum_{j=1}^{i} \frac{c_j}{p_j} \leq U(i) \right\}$$

Priority Inversions (2/2)

- Three undesirable effects stem from PI
  - Unbounded priority inversions
  - Timing anomalies
  - Deadlocks
- Priority inheritance protocol (PIP) avoids unbounded priority inversions
- Priority ceiling protocol (PCP) is a superset of PIP, and it has only one blocking and avoids deadlocks


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 difficult to implement
  - RTLinux
- PIP is much easier to implement
  - uC/OS-II, RTAI, FreeRTOS
- NPCS is the easiest
  - Equivalent to scheduler locking upon resource acquisition

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