ELEC3730
Embedded Systems

Real-time kernels
Shared Resources

Adrian Wills, EA-204, Ext. 16028

Previous Lecture

- Task states
  - dormant, ready, running, waiting
- Scheduler
  - When: preemtive & non-preemtive kernels
  - Which: assigning priorities (RMS)
  - How: context switching
- Shared resources
  - critical sections
  - scheduler locking
  - semaphores

Topics Covered

- Shared Resources
  - Review:
    - Critical sections
    - Schedule locking
    - Semaphores
  - Priority inversion
  - Mutexes (mutual exclusion semaphores)
  - Deadlock
  - Reentrant functions

Shared Resources

- Tasks often need exclusive access to shared resources
- Methods for obtaining exclusive access include
  - ***Critical sections:*** disable interrupts
  - ***Disabling scheduler:*** do not schedule any other tasks (ISR's can still run)
  - ***Semaphores:*** must obtain special key to access the resource

Critical Sections

```
#define  OS_ENTER_CRITICAL()  asm {PUSHF; CLI}  /* Disable interrupts */

#define  OS_EXIT_CRITICAL()   asm  POPF         /* Enable interrupts */
```

```
int Temp

void swap(int *x, int *y)
{
  OS_ENTER_CRITICAL();

  Temp = *x;
  *x = *y;
  *y = Temp;

  OS_EXIT_CRITICAL();
}
```

Interrupts disabled

Guaranteed execution order

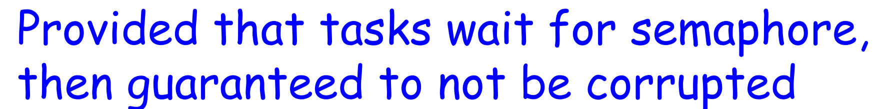
Interrupts enabled

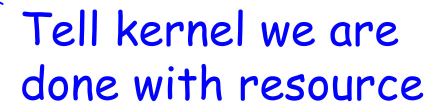
Disabling Scheduler

```
int Temp;

void swap(int *x, int *y)
{
  OSSchedLock();

  Temp = *x;
  *x = *y;
  *y = Temp;

  OSShedUnlock();
}
```

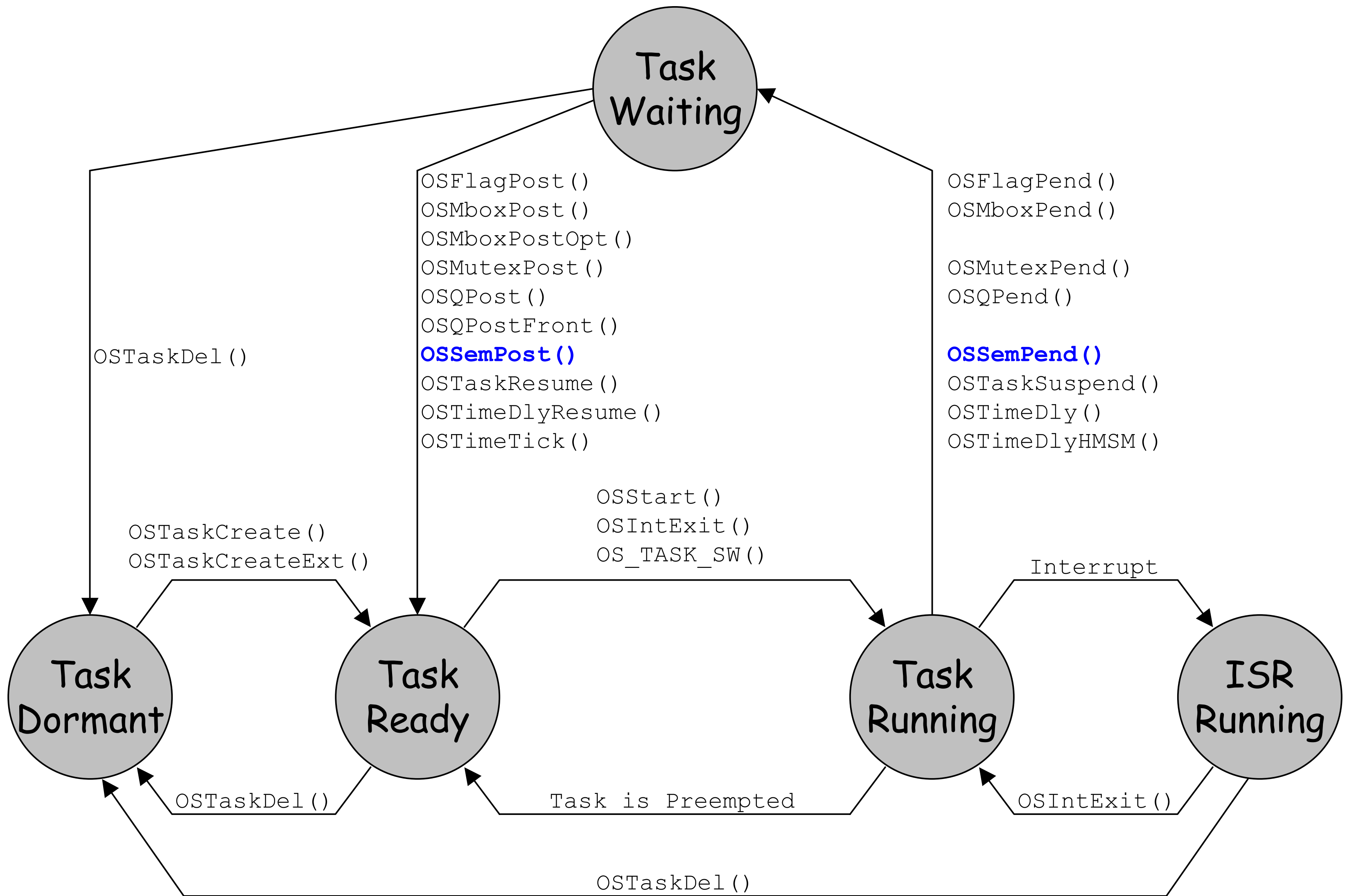
Semaphore

```
int Temp;

void swap(int *x, int *y)
{
  OSSemPend(SharedIntSem, 0, &err);

  Temp = *x;
  *x = *y;
  *y = Temp;

  OSSemPost(SharedIntSem);
}
```





Task states: Semaphore Pend/Post



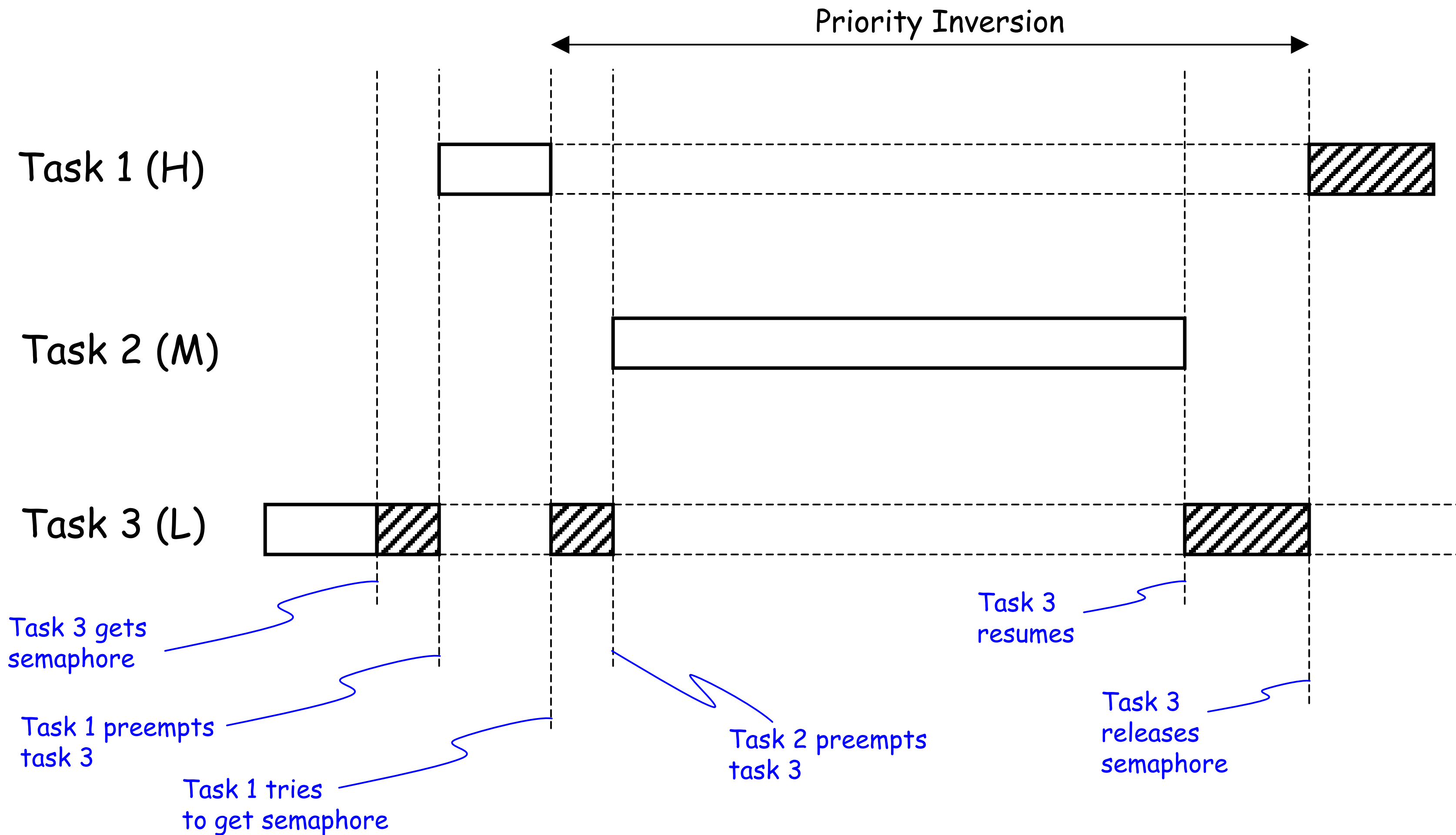
Semaphores

- Can have a ***count*** value if resource has multiple parts
- Can be used to synchronise tasks at some point in the code
- ISR's should never wait on a semaphore
  - ***if blocked, all other ISR's may be missed***

Semaphore Related Problems

- Priority inversion
  - high priority task waiting for resource owned by low priority task
  - low priority task is preempted by other medium priority tasks
  - high priority task is *effectively* reduced to low priority
- Deadlock
  - group of tasks waiting on resources owned by other tasks in group
  - how to avoid and/or detect?

Priority Inversion



Priority Inversion

- Bounded Priority Inversion
  - priority inversion duration equal to low-priority critical section
- Unbounded Priority Inversion
  - medium-priority task preempts the low-priority task during inversion
  - indefinite amount of time for medium level tasks

Mars Pathfinder

- Suffered Unbounded Priority Inversion
- Low-priority meteorological thread
  - Acquired the (shared) bus
- Medium-priority, long-running, communications thread
  - Woke up and preempted the meteorological thread
- High-priority bus management thread
  - Woke up and was blocked because it couldn't acquire the bus; when it couldn't meet its deadline it reinitialized the processor via a hardware reset

Mars Pathfinder Article

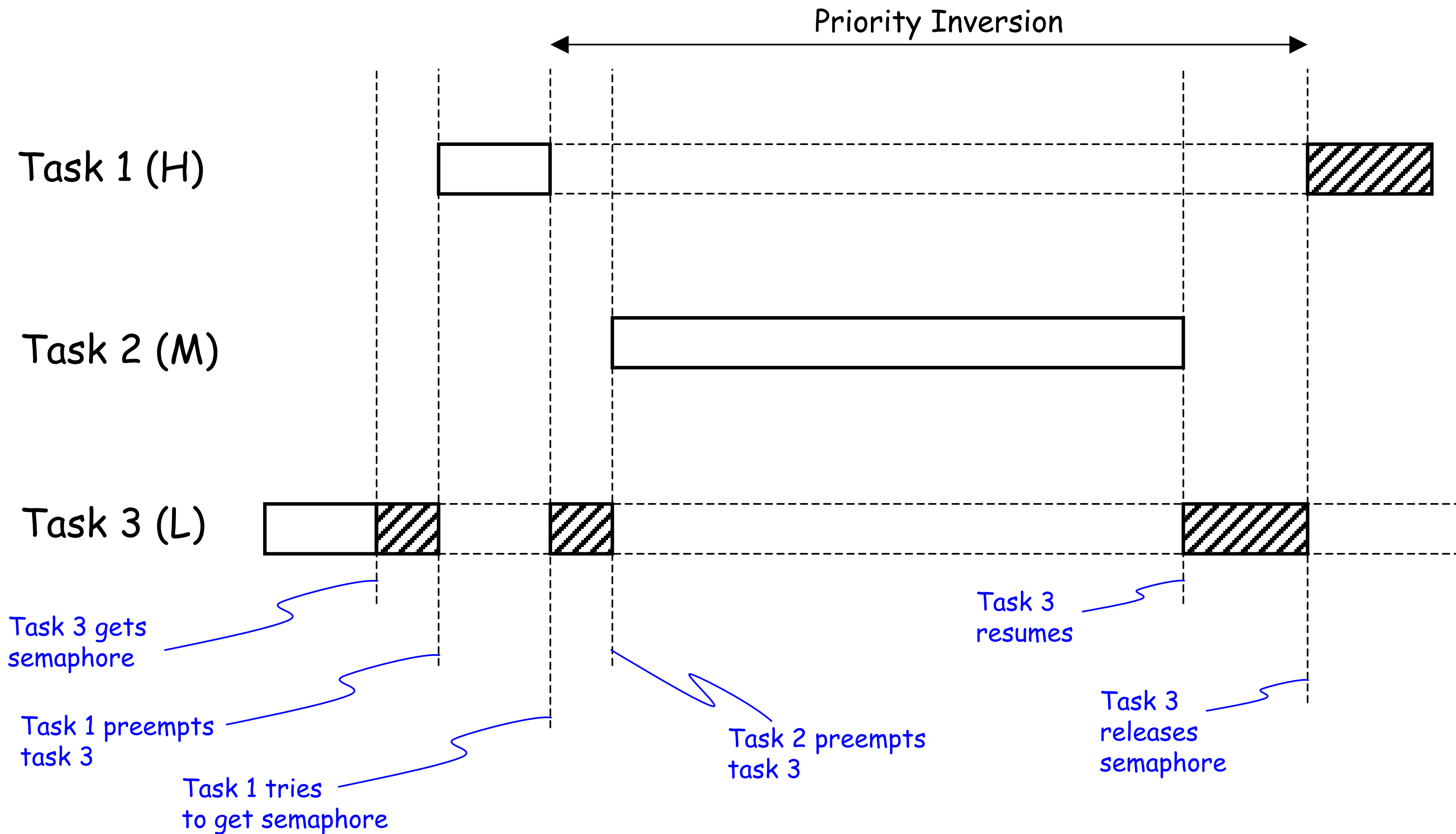
Limiting the Duration of an Unbounded Priority Inversion

- Objective:
  - prevent low-priority task from being preempted by medium-priority task during priority inversion
- Strategies:
  - Priority Inheritance Protocol (Micro-C/OS-II)
  - Priority Ceiling Protocol
- Technique:
  - Manipulate task priorities at run-time
  - Micro-C/OS-II has special semaphores called mutexes that offer this priority inheritance

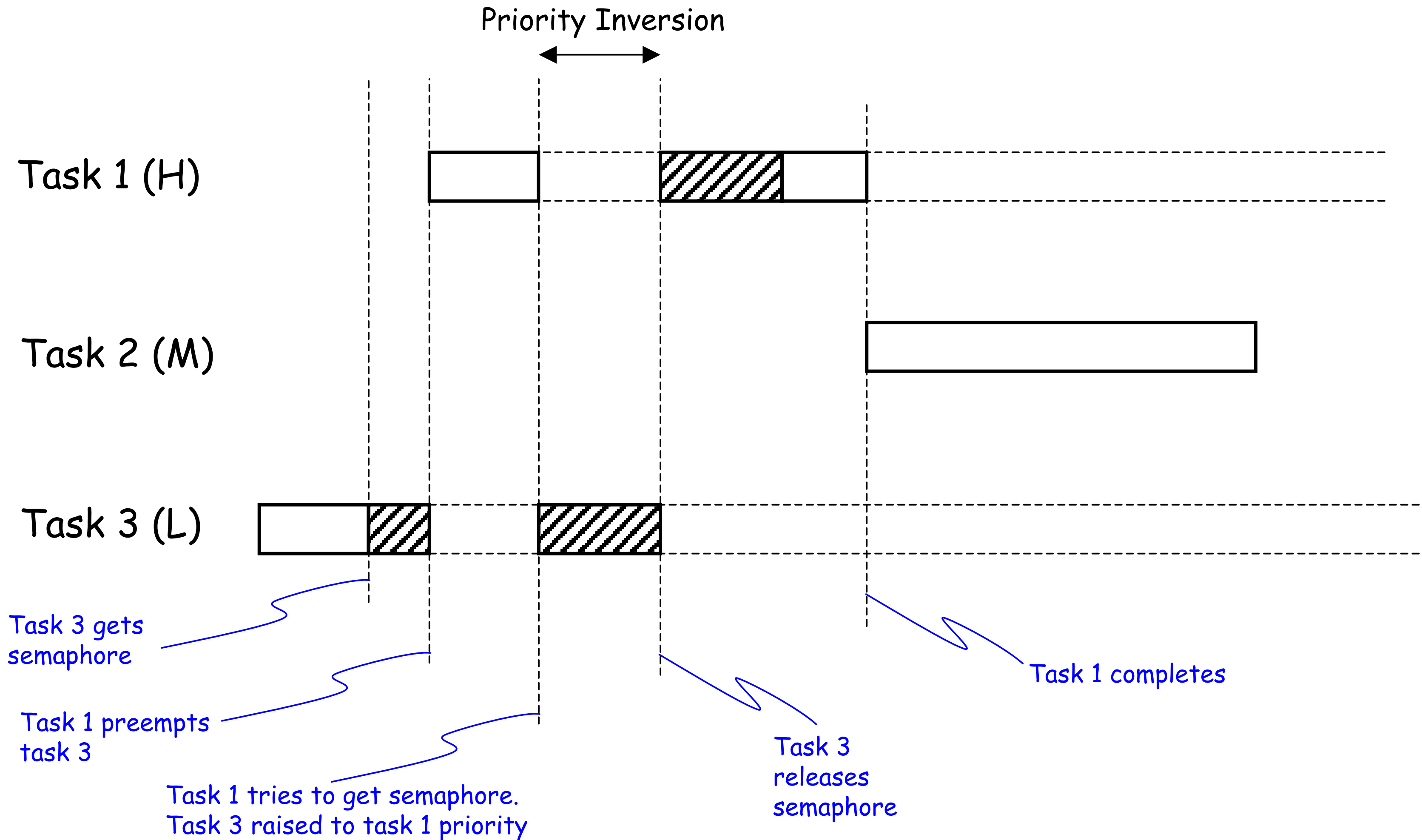
Priority Inheritance Protocol

- High-priority task attempts to get semaphore
  - raise priority of low-priority task to match that of blocked task
  - wait until low-priority task unlocks semaphore
- Advantage
  - It is transparent to the application.
- Disadvantage
  - Adds complexity to the kernel.

Priority Inversion



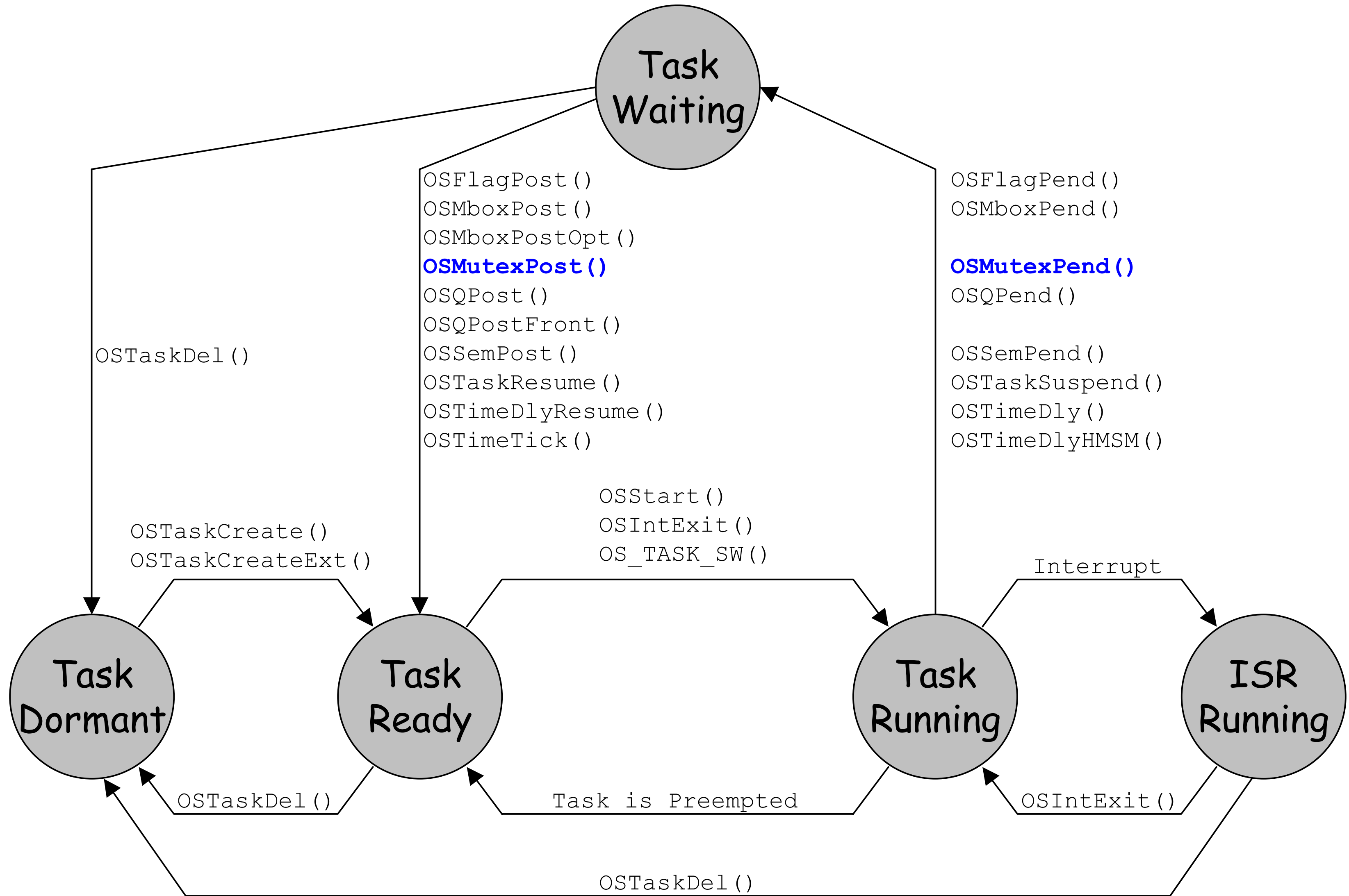
Priority Inversion

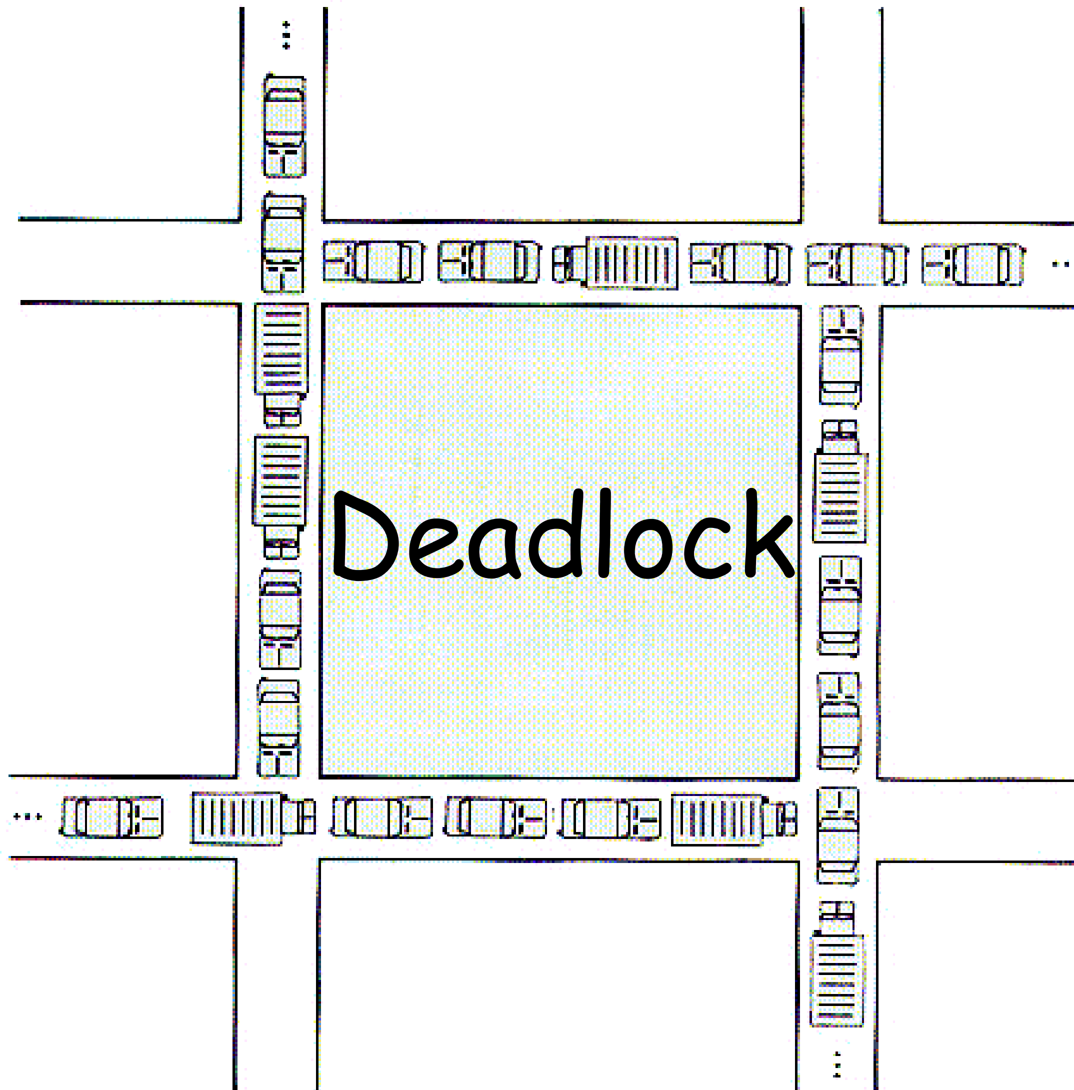


Mutex

- Mutual Exclusion Semaphores (mutexes)
  - binary semaphore
  - kernel provides priority inversion remedy for mutexes
- Micro-C/OS-II
  - cannot inherit same priority level
  - OSMutexCreate(INT8U prio, INT8U *err)
  - must specify priority level that task is raised to in event of priority inversion

Task states: Semaphore Pend/Post

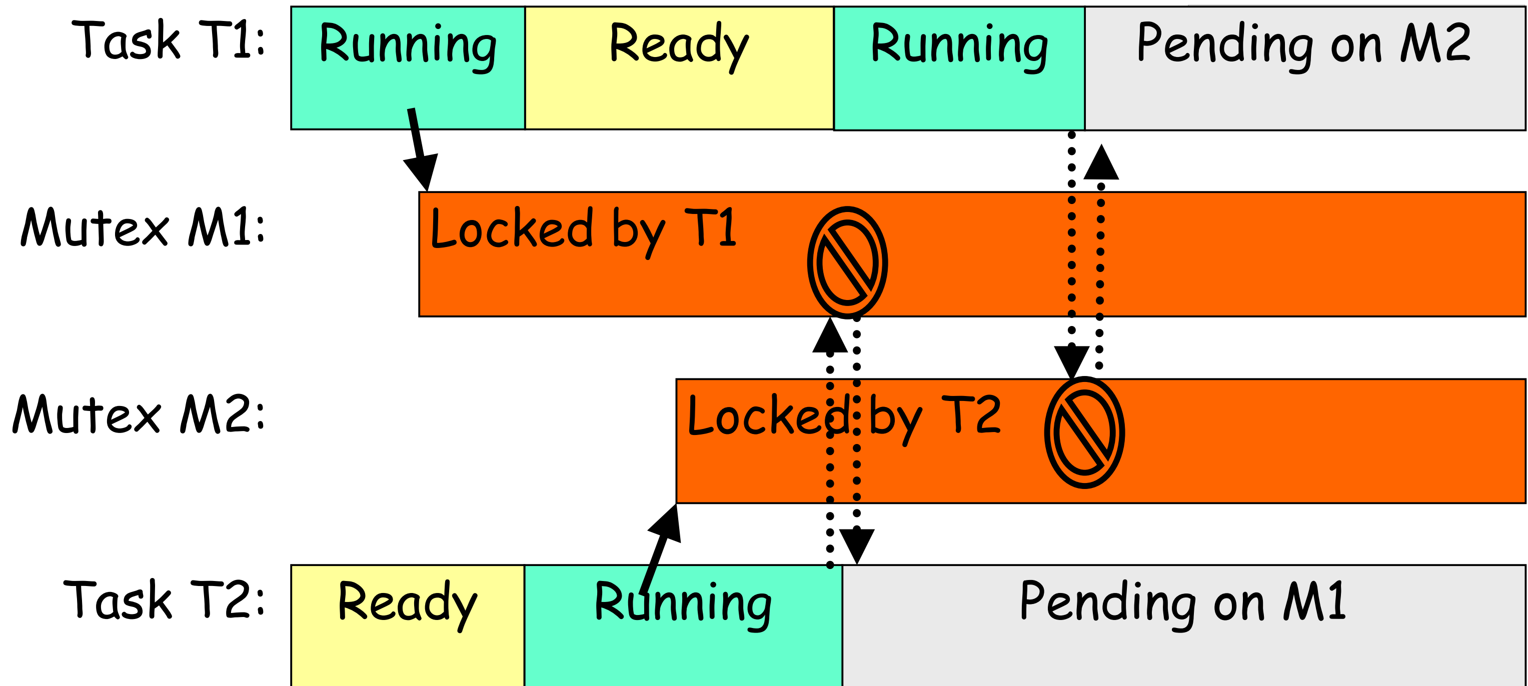



Deadlock

Deadlock

- Definition
  - Two or more tasks blocked waiting on each other's resources
- Necessary Condition
  - Threads require exclusive access to two or more shared resources simultaneously

Entering Deadlock



Preventing Deadlock

- Require all tasks to acquire resources in the same order
  - Often locks resources longer than necessary
  - Occasionally there is no order that is convenient for all tasks
- Tasks obtain exclusive access to *all* resources before proceeding
- Require threads to release *all* resources and start over if any one resource can't be acquired

Reentrant Functions

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

Non-Reentrant Functions

```
int Temp

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

Non-Reentrant Functions

Low-priority task

```
while (1)
{
  int x = 1;
  int y = 2;

  swap(&x, &y)
  {
    Temp = *x;


    *x = *y;
    *y = Temp;
  }
  .
  .
  OSTimeDly(100);
}
```

Temp == 1

Context Switch

ISR → OS → High-priority task

High-priority task

```
while (1)
{
  int z = 1;
  int t = 2;

  swap(&z, &t)
  {
    Temp = *z;
    *z = *t;
    *t = Temp;
  }
  .
  .
  OSTimeDly(100);
  .
  .
}
```

Temp == 3

OS → Low-priority task

y == 3