Lock Ordering

Prof. Naga Kandasamy
ECE Department, Drexel University

Consider two threads that use nested locks A, B, and C as follows:

Thread T_1:

```
lock (C);
/* Code */
lock (B);
/* Code */
lock (A);
/* Code */
unlock (B);
/* Code */
unlock (C);
/* Code */
unlock (A);
```

Thread T_2:

```
lock (B);
/* Code */
lock (A);
/* Code */
lock (C);
/* Code */
unlock (C);
/* Code */
unlock (A);
/* Code */
unlock (B);
```

- The above code has the potential to deadlock. Consider the following scenario. Thread T_1 has acquired lock `C` whereas T_2 has acquired locks `B` and `A`. At this point, T_1 is holding `C` and waiting for T_2 to release `B`; and T_2 is holding `B` waiting for T_1 to release `C`. This is a *circular wait* condition necessary for a deadlock to occur.
- Lock ordering is a *deadlock prevention* technique that works as follows. Let us denote the set of resource types as $R = \{R_1, R_2, \ldots, R_m\}$. We then assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether

one precedes another in our ordering. More formally, we can define a one-to-one function $F : R \rightarrow N$, where N is the set of natural numbers. For example, $F(R_1) = 1$, $F(R_2) = 2$, $F(R_3) = 3$, and so on. The lock ordering protocol proceeds as follows:

– Each thread can request resources only in an increasing order of enumeration. That is, a thread can initially request any number of instances of a resource type, say R_i. After that a thread can request instances of resource type R_j if and only if $F(R_j) > F(R_i)$.

– A thread requesting an instance of resource type R_j must have released any resource R_i such that $F(R_i) \geq F(R_j)$. If several instances of the same resource type are needed, a single request for all of them must be issued.

Let us order the locks as $F(A) = 1$, $F(B) = 2$, and $F(C) = 3$, and apply the lock ordering protocol to the code in thread T_1. In Line 3, T_1 is holding `C` and is requesting `B` which has a lower enumeration than `C`. Therefore, as per the protocol, T_1 must release `C` and try to acquire locks `B` and `C` in order. A similar situation applies to Line 5 where the process tries to acquire `A`.

The revised code for T_1 that follows the lock ordering protocol is as follows.

```
lock (C);
/* Code */
unlock (C);
lock (B);
lock (C);
/* Code */
unlock(B);
unlock (C);
lock (A);
lock (B);
lock (C);
/* Code */
unlock (B);
/* Code */
unlock (C);
/* Code */
unlock (A);
```

The revised code for T_2 is as follows.

```
lock (B);
/* Code */
unlock (B);
lock (A);
lock (B);
/* Code */
lock (C);
/* Code */
unlock (C);
/* Code */
unlock (A);
/* Code */
unlock (B);
```