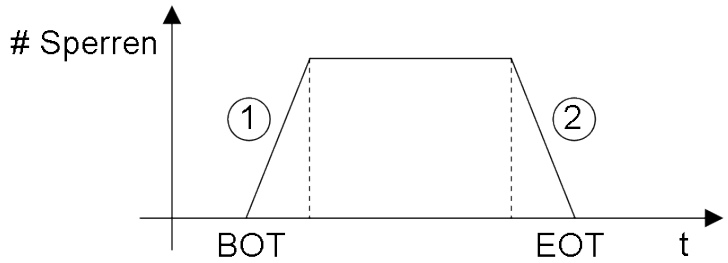
## *Wiederholung (2)*

**Legaler Schedule**

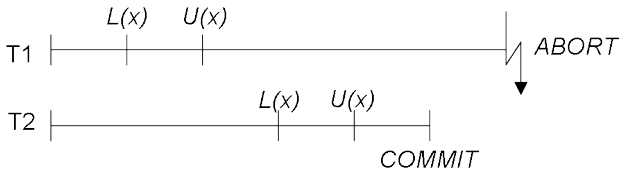
- LOCK (L) vor jedem Zugriff
- UNLOCK (U) spätestens bei TA-Ende
- keine TA fordert Sperre an, die sie bereits besitzt
- Sperren werden respektiert

**2-Phasen-Sperrprotokoll (2PL)**
Jede TA durchläuft 2 Phasen

1. Wachstumsphase (LOCK)
2. Schrumpfungsphase (UNLOCK)




Problem des 2PL: Kaskadierendes Rücksetzen:

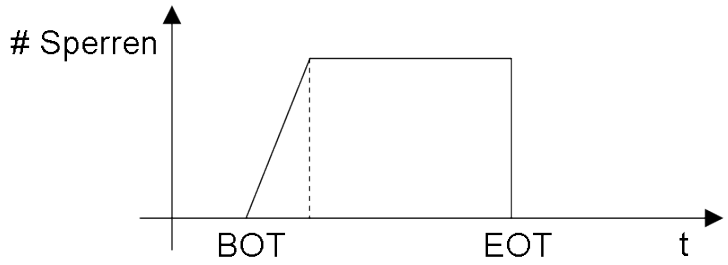



$T_1$ wird nach UNLOCK $U_1(x)$ zurückgesetzt $\Rightarrow$ alle (auch schon abgeschlossenen) TAs, die nach $U_1(x)$ auf $x$ zugegriffen haben, müssen auch zurückgesetzt werden: Widerspruch zur Durability!

Das Problem hier ist, dass durch das Rollback die Änderungen von $T_1$ auf $x$ wieder rückgangig gemacht werden müssen, dadurch geschieht am Ende nochmal ein $w_1(x)$, obwohl $T_1$ den Lock von $x$ gar nicht mehr hat. Dieses $w_1(x)$ beim Rollback von $T_1$ erzeugt ein Dirty Read/Write (je nachdem, ob $T_2$ liest/schreibt).
Abhilfe: **Striktes 2PL**

- alle Sperren werden bis zum COMMIT gehalten

- COMMIT wird atomar durchgeführt



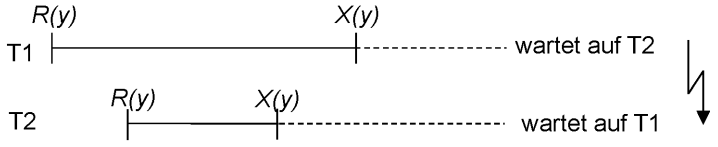

**Es gilt**

- 2PL $\Rightarrow$ serialisierbar
- ABER: serialisierbar $\neg \Rightarrow$ 2PL (Siehe $S_2$ in Aufgabe 2-1)
  2PL ist also hinreichend, aber nicht notwendig für die Serialisierbarkeit.

**RX-Protokoll**

| | *R (bestehend)* | *X (bestehend)* |
|---|---|---|
| *R (angefordert)* | + | - |
| *X (angefordert)* | - | - |

- **Vorteil**: höhere Parallelität für Leseanfragen
- **Problem**:




$T_1$ setzt $R$-Sperre auf Objekt $y$, $T_2$ setzt $R$-Sperre auf Objekt $y$. $T_2$ kann dann keine $X$-Sperre auf Objekt $y$ setzen, solange eine $R$-Sperre auf Objekt $y$ durch $T_1$ besteht und umgekehrt. $\Rightarrow$ *Verklemmung*

**RUX-Protokoll**

| | *R (bestehend)* | *U (bestehend)* | *X (bestehend)* |
|---|---|---|---|
| *R (angefordert)* | + | - | - |
| *U (angefordert)* | + | - | - |
| *X (angefordert)* | - | - | - |

- **U-Sperre**: *exklusive* Lesesperre für spätere $X$-Sperre; zur Änderung erfolgt Konversion $U \rightarrow X$, andernfalls $U \rightarrow R$
- **Vorteil**: kein Verhungern bei $U \rightarrow X$, da keine neuen Leser bei einer $U$-Sperre zugelassen werden (es muss nur auf alte Leser gewartet werden).
- **Nachteil**: wenig Parallelität, weil neue Leser blockiert sind, bis die Update($U$)-Transaktion zu $R$ konvertiert, oder bis Sie fertig ist und Ihren $X$-Lock zurückgibt.

**RAX-Protokoll**

| | *R (bestehend)* | *A (bestehend)* | *X (bestehend)* |
|---|---|---|---|
| *R (angefordert)* | + | + | - |
| *A (angefordert)* | + | - | - |
| *X (angefordert)* | - | - | - |

- **A-Sperre**: Lesesperre für spätere $X$-Sperre; zur Änderung erfolgt Konversion $A \rightarrow X$ (weitere $R$-Sperren sind aber erlaubt)
- **Vorteil**: höhere Parallelität als bei RUX
- **Problem**: bei bestehender $A$-Sperre und angeforderter $R$-Sperre kann die $A$-Sperre nur dann zu einer $X$-Sperre konvertiert werden, falls keine weiteren $R$-Sperren bestehen. $\Rightarrow$ *Verhungern*

**RIX-Protokoll**

| | *R (bestehend)* | *X (bestehend)* | *IR (bestehend)* | *IX (bestehend)* | *RIX (bestehend)* |
|---|---|---|---|---|---|
| *R (angefordert)* | + | - | + | - | - |
| *X (angefordert)* | - | - | - | - | - |
| *IR (angefordert)* | + | - | + | (+) | (+) |
| *IX (angefordert)* | - | - | (+) | (+) | - |
| *RIX (angefordert)* | - | - | (+) | - | - |

- **IR-Sperre**: tiefere Ebene hat $R$-Sperre
- **IX-Sperre**: tiefere Ebene hat $X$-Sperre
- **RIX-Sperre**: volle Lesesperre, tiefe Schreibsperre