4D Record Locking and ORDA Entity Locking

By Erick Lui, Technical Services Engineer, 4D Inc.

Technical Note 18-22

Table of Contents

Abstract

Record locking is an essential feature when handling a database with multiple users. Users can simultaneously add, modify, or remove records independently, and there must be a safety mechanism to prevent record conflicts. To prevent corrupted data, there are two types of locking mechanisms: optimistic and pessimistic locks. Both of these types will have advantages and disadvantages, but with the option to adhere to standard 4D locks and the newly added entity locks with ORDA, 4D developers will now have further control and customization on how to handle multi-user record access.

Introduction

Traditional 4D commands adhere to an automatic **pessimistic** type locking meaning only one user or process can modify the currently accessed record. Once the record is loaded, the record is locked and cannot be modified by another process until it is unloaded. In general, pessimistic locking may be preferable under the assumption that a record will have a high rate of access and chance of conflicts. The main issue is that the first user who accesses the record will be the only one with read/write access while everyone else must wait in the read-only state until the record is unlocked to modify data.

On the other hand, **optimistic** locking allows multiple users to modify the same record simultaneously and only compares the record version upon saving. Typically, this mechanism can be implemented via record date, timestamp, checksum, or hash. In ORDA's case, the entity's **stamp** is compared which will be explained in more detail in later sections. Optimistic locking is often used when records have a low chance of conflict with only a few users. Both locking types will have their own tradeoffs and it will be up to the developers to consider which locking type will be best for their app. However, this tech note will primarily emphasize on the comparison between classic 4D locks and the implementation of ORDA entity locks in 4Dv17.

Standard 4D Record Locking

Table Read-Only and Read/Write States

In 4D, each table per process can only be in one of two states: **read-only** or **read/write state**. The read-only state means that the record can be loaded but existing records cannot be modified while the read/write state allows the record to be loaded and modified. However, the read-only state still allows adding or creation of records. It is possible to change the status of a table between the two states. Switching between states does not affect the current loaded record as this change will only affect records loaded afterwards. Otherwise if not specified, 4D will default tables will be in read/write mode. Listed below are some examples of 4D commands that will alter the state of a table.

Read-Only and Read/Write Commands

Sets Table to Read-Only	Sets Table to Read-Write
READ ONLY	READ WRITE
DISPLAY SELECTION	
DISTINCT VALUES	
EXPORT DIF	
EXPORT SYLK	
EXPORT TEXT	
PRINT SELECTION	
PRINT LABEL	
QR REPORT	
SELECTION TO ARRAY	
SELECTION RANGE TO ARRAY	

Loading and Unloading Records

When a record is loaded (E.G. QUERY, NEXT RECORD, PREVIOUS RECORD, etc.), there will be several different outcomes depending on the table and record state. A useful way to check the record's state is using the **Locked** command which returns true if the record is locked and false the record is unlocked. The table below shows the possible scenarios.

Table State	Record Locked?	Load Record Result	Locked function output
Read-Only	Yes	Record loaded as read-only	True
Read-Only	No	Record loaded as read-only	True
Read/Write	Yes	Record loaded as read-only	True
Read/Write	No	Record can be modified	False

In short, the only time a record can be modified in standard 4D is when the table is in read/write state and the current record is not locked. Otherwise, the record will be loaded as read-only. Here's a simple example of the loading and unloading process would look like in code for the [People] table.

```
// Given that a current selection already exists
READ WRITE([People])
MODIFY RECORD([People])
UNLOAD RECORD([People])
READ ONLY([People])
```

First the table is set to read/write mode to allow a user to modify any subsequent loaded records and locks the record for other users. The **MODIFY RECORD** command then attempts to load the current record which displays the default input form if record is not locked or

displays a dialog warning if the record is already locked. Once the user is done modifying the record, the **UNLOAD RECORD** is called to unlock the record and allow read/write access for other users. Lastly, the table is then set back to read-only as a safety measure to make sure the table is not accidentally left in read/write mode. In addition, read-only mode can be beneficial for efficient memory management and faster table access speed. The following images further demonstrates this scenario using a listbox with current selection.

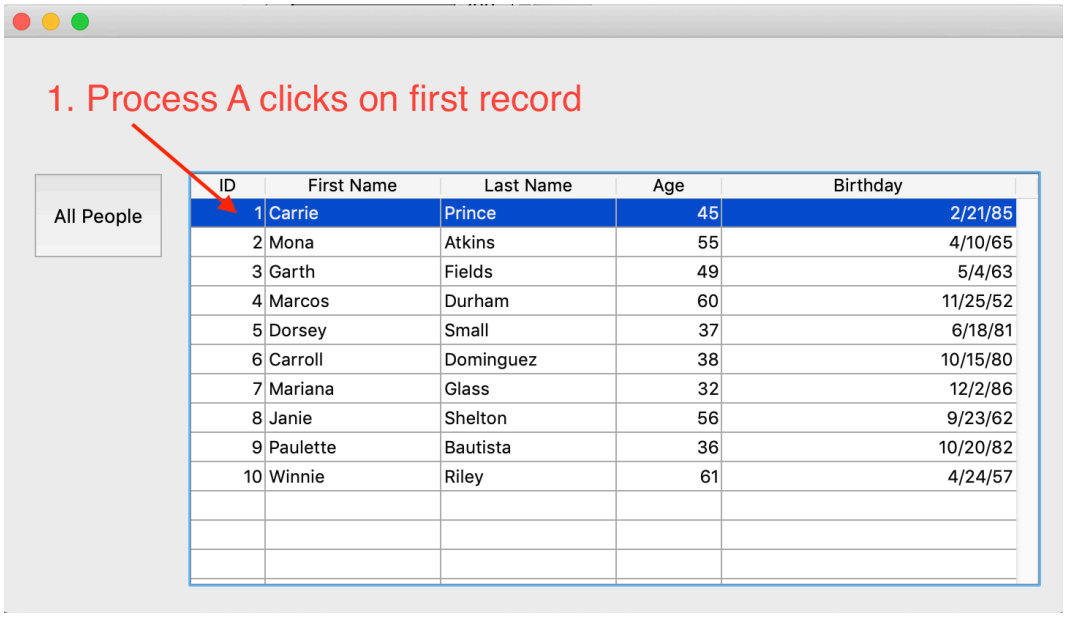

Image 1: First record is accessed by Process A

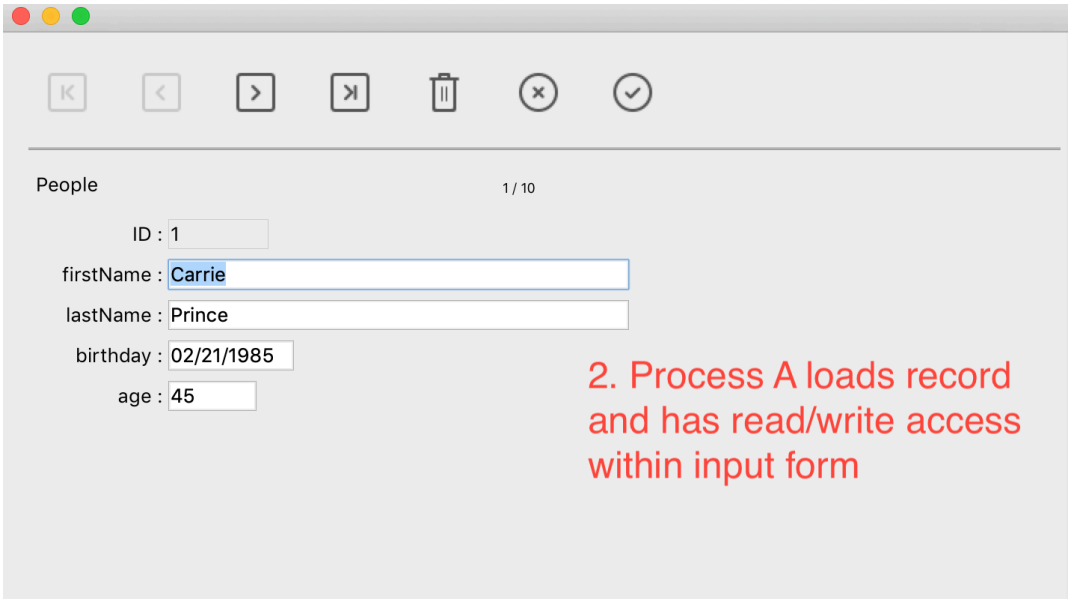

Image 2: Process A has read/write access

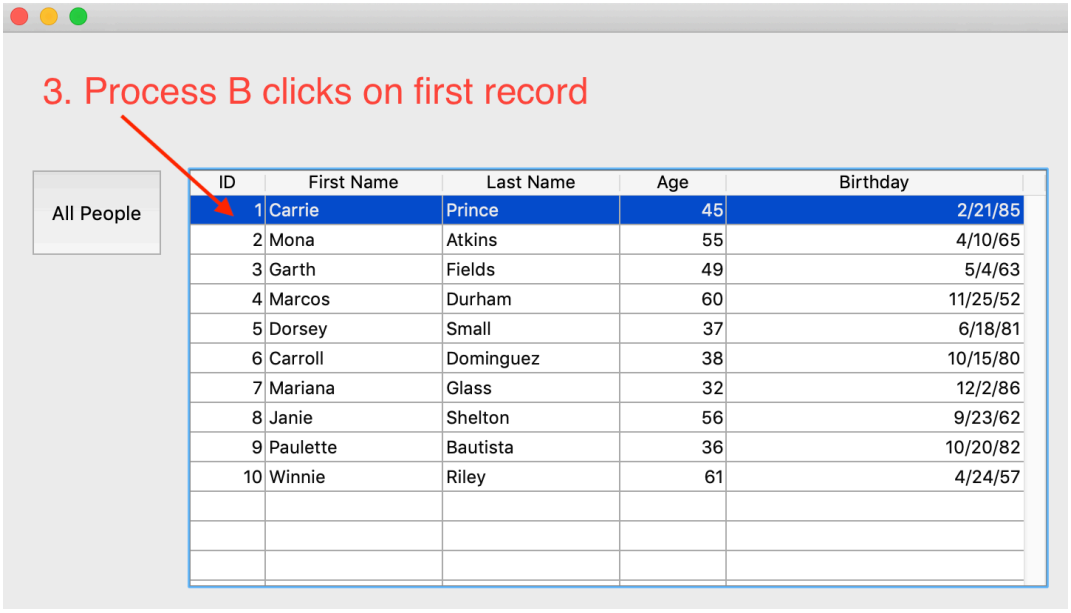

Image 3: Process B attempts to access same record

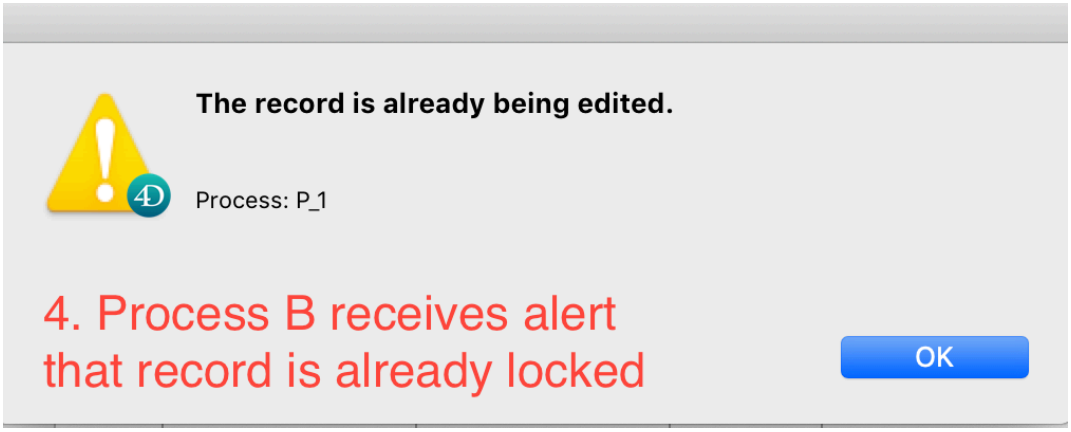

Image 4: Pessimistic lock prevents read/write access for Process B

Checking for Locked Records and Locked Info

Sometimes receiving an alert with just the process id is not enough information. If more information is needed regarding who is using the record, the commands listed below can provide more context about the process that locked the record.

Commands	Description
Locked	Checks whether current record is locked
LOCKED BY	Returns info about user and process that locked the record

Get locked records info	Returns info regarding all locked records in a table

The **Locked** command is useful for checking the locked state of a record and is ideally used in conditional statements. If the record is locked, a dialog message can be displayed to notify a user that the record is in use. Otherwise, the else block can be used for record modifications and record unloading.

```
If (Locked([Table_1]))
	// display info regarding who locked the record
Else
	// do record modifications here
End if
```

For the **LOCKED BY** command, it will take the table name as input and output the process number, 4D user name, session user, and process name as shown below.

```
C_LONGINT($processNum_l)
C_TEXT($userName_t;$sessionName_t;$processName_t)

// Returns locked info into 4 variables above regarding current record from
Table_1
LOCKED BY([Table_1];$processNum_l;$userName_t;$sessionName_t;$processName_t)
```

If more information regarding all records for a table is needed, **Get locked info** would be preferred as it will return an object containing an array of all locked records and the processes that are currently accessing these records.

```
C_OBJECT($lockedInfo_ob)

$lockedInfo_ob:=Get locked records info([Table_1])
```

```
// $lockedInfo_ob result with 1 locked record

{
	"records": [
		{
			"contextID": "6B668E5AB33A47889AC72A0E5C86AAFC",
			"contextAttributes": {
				"task_id": 5,
				"user_name": "Erick Lui",
				"user4d_id": 1,
				"host_name": "Erick's MacBook Pro",
				"task_name": "P_17",
				"client_version": -268364032
			},
			"recordNumber": 0
		}
	]
}
```

ORDA Entity Locking

Now introduced in v17 is ORDA entity locking which allows both pessimistic and optimistic lock types. Pessimistic type will lock the record upon access which prevents other users from modifying the same record at the same time. At most, standard 4D commands allowed multiple users to read and add records simultaneously. However, ORDA defaults to optimistic locking which will actually allow modifications by multiple users at the same time and will check whether an entity's update was valid upon saving the entity. Both of these locking types will have tradeoffs as optimistic locking sacrifices guaranteed write operations for multiple read/write access while pessimistic locking guarantees successful write operations but prevents multiple simultaneous updates on the same record. This decision will ultimately be up the developer to carefully consider the number of users and how often a record will be accessed.

Stamps

Stamps are a simple counter attached to each entity to keep track of how many times the entity has been saved. Upon a successfully saved, the stamp will increment by one. This stamp is also how ORDA determines whether another process has modified the currently accessed record before another attempted save. For example, Process A and B both load the entity at the same time with the entity's stamp default to 1. Process A makes its changes and saves, which updates the stamp counter to 2. Process B then makes its changes and attempts to save but receives an error since the stamp is now 2 when upon loading the stamp was originally 1. The stamp does not match its initial value upon loading, hence Process B must reload the entity before making more changes. The diagrams below further illustrate this sequence. Note that the entity is never locked upon access as ORDA solely relies on the stamp.

Image 5: Process A and B load the same entity

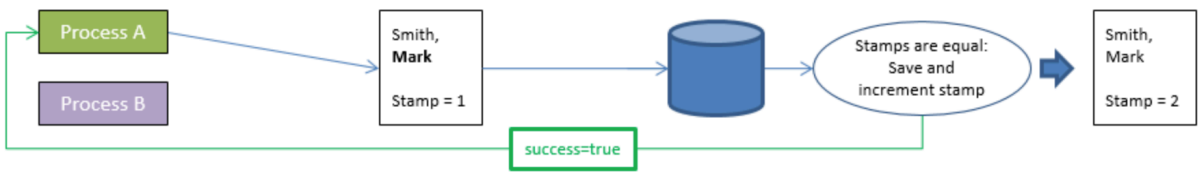

Image 6: Process A saves the entity first and stamp is automatically incremented

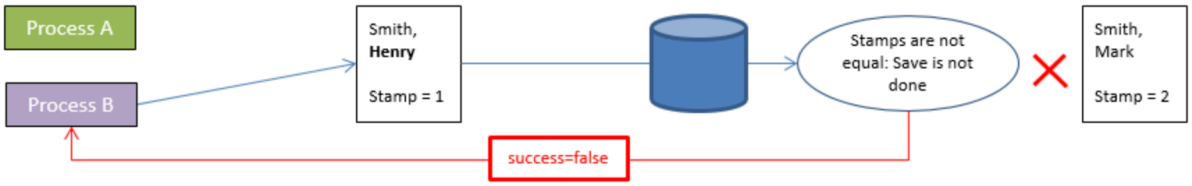

Image 7: Process B attempts to save the entity but fails due to stamp change

Scenario 1: Pessimistic locking using entity.lock() and entity.unlock()

Similar to standard 4D locking, ORDA provides the option to manually lock entities using the commands entity.lock() and entity.unlock(). Typically, these commands are called during the form events On Load and On Unload, respectively. When attempting to lock or unlock an entity, these commands will return an object containing the **success** property boolean and other properties depending on whether the entity was already locked. An example is provided below where the entity **Form.ent** is passed into a form and locked upon the form's On Load event.

```
Case of
	: (Form event=On Load)
		C_OBJECT($status)
		$status:=Form.ent.lock()

		If($status.success)
			// display success status
		Else
			// display error status
		End if

End case
```

Expression	Value
$status	{"success":true}

Image 8: Returned object from successful entity.lock()

Expression	Value
$status	{"status":3,"sta...success":false}
lockInfo	{"task_id":7,"us...":-268364032}
client_version	-268364032
host_name	"Erick's MacBook Pro"
task_id	7
task_name	"Process A"
user_name	"Erick Lui"
user4d_id	1
lockKind	1
lockKindText	"Locked By Record"
status	3
statusText	"Already Locked"
success	False

Image 9: Returned object from failed entity.lock()

The returned object from these commands can be extremely useful for checking whether the entity was successfully locked or displaying possible error messages. For example, ORDA by default does not prevent multiple users from accessing the same entity even when locked as only the first user that locked the entity will be allowed to save. This scenario can be misleading for the subsequent users as there needs to be some indication on the form to show that the entity is currently locked. The image below shows this scenario.

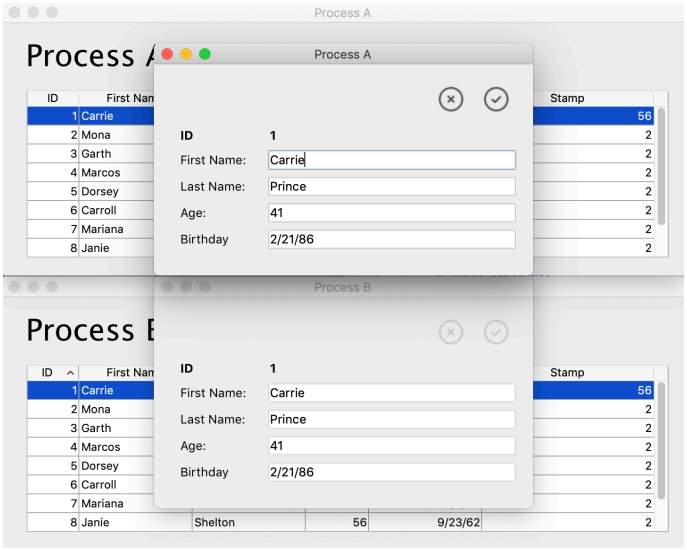

Image 10: No form indication on which user accessed the entity first

Using the properties **success** and **statusText** from the returned object**,** the form can be further stylized to clearly indicate the entity's lock status and disable enterable fields when the entity was locked by another user. In this case, Process A was the first to lock the entity and Process B attempted to lock the same entity afterwards.

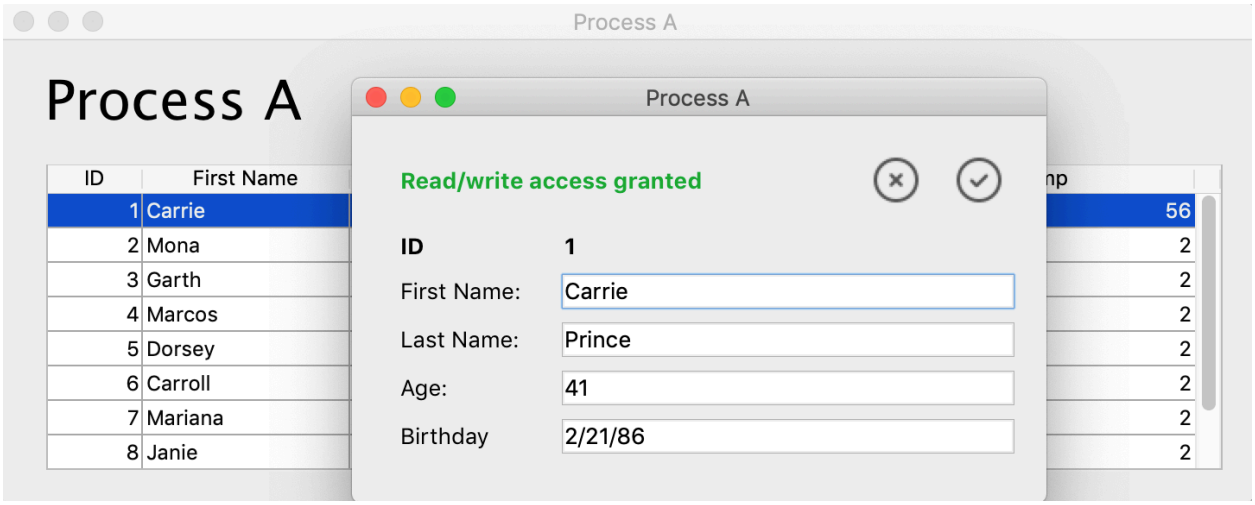

Image 11: Status text displayed for first user that locked the entity

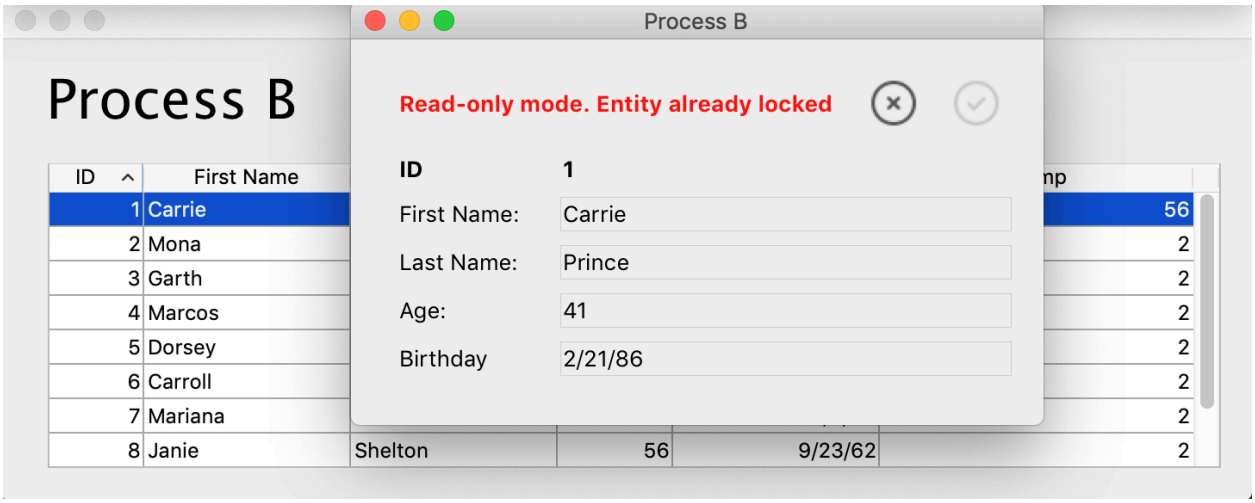

Image 12: Status text displayed, confirm button disabled, and fields set to non-enterable

Scenario 2: Optimistic locking using entity.save()

For this scenario, the commands entity.lock() and entity.unlock() are no longer needed. Instead, entity.save() is the only command that will be used which simply saves changes made to the entity. It will also return an object containing the success boolean and other error properties similar to the entity locking commands shown in images 8 and 9. When two

or more processes load the same entity, the only successful save will be from the first process. Otherwise, all subsequent processes will be unable to save changes to the entity as the stamp has changed since its loading. To resolve this issue, the entity has to be reloaded using entity.reload() or the form containing the entity has to be reopened. The images below provide more context in recreating this scenario.

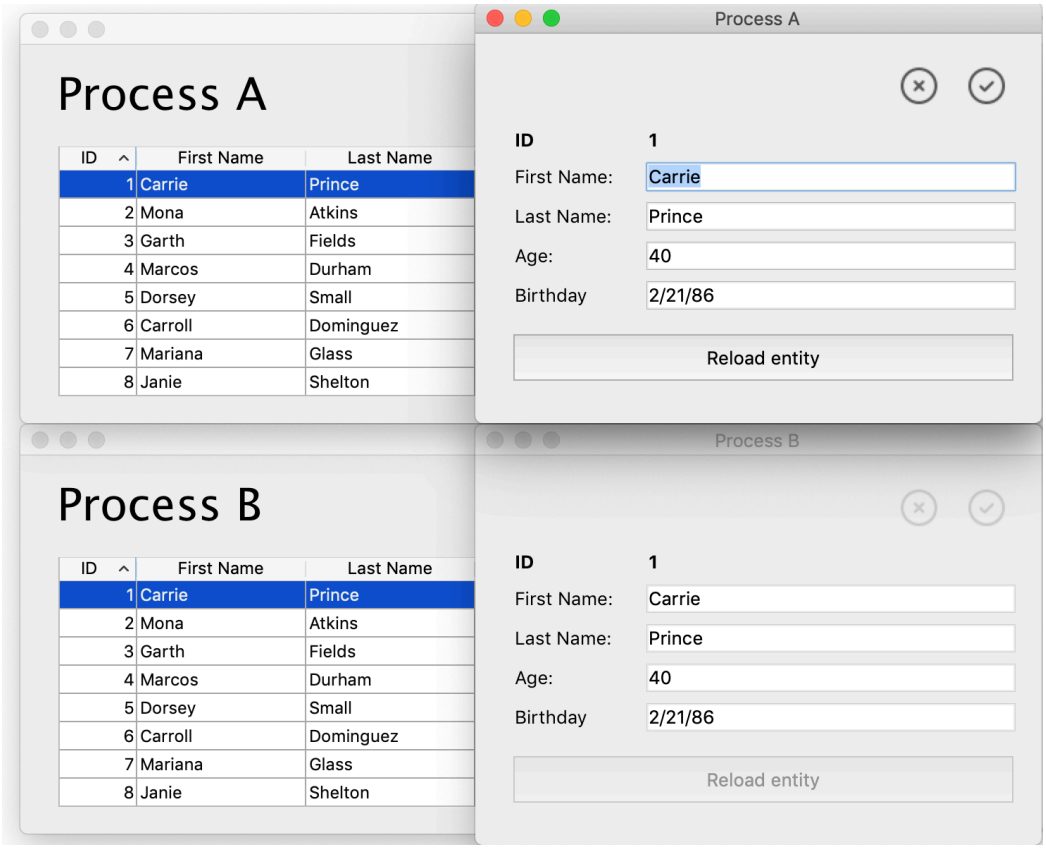

Image 13: Two processes load the same entity

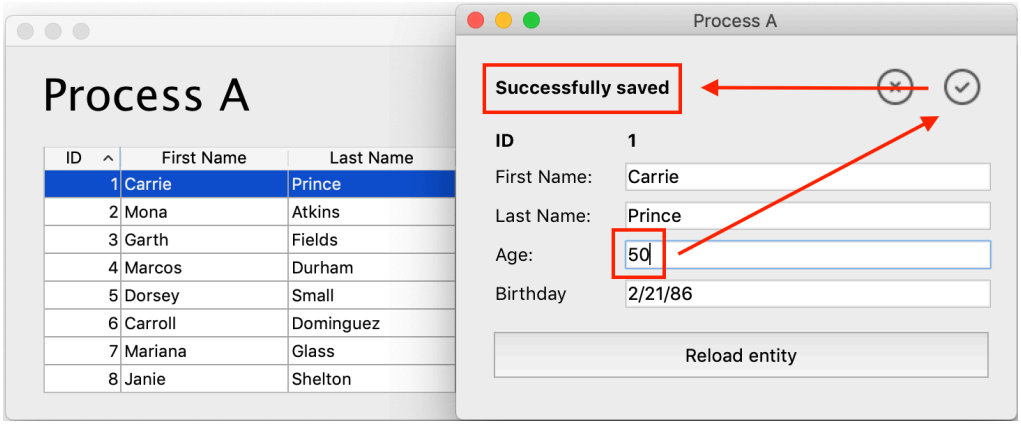

Image 14: Process A edits age field and saves entity successfully

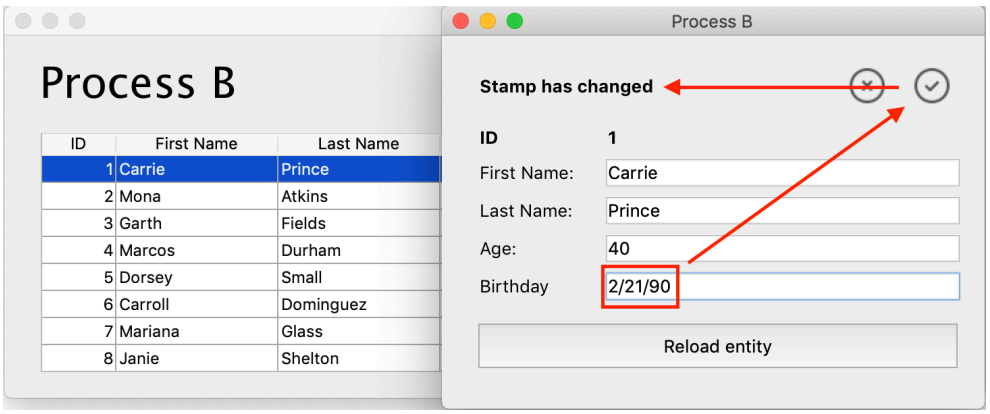

Image 15: Process B edits birthday field but save returns error message

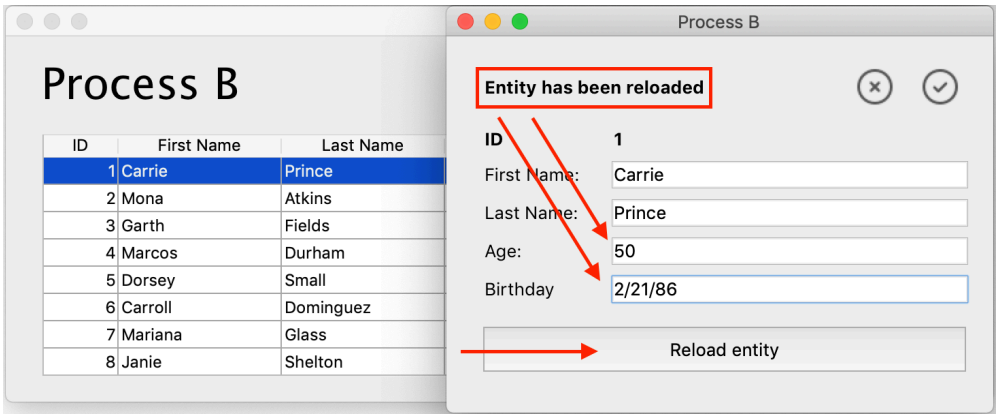

Image 16: Process B reloads entity and loads data from most recent data from Process A

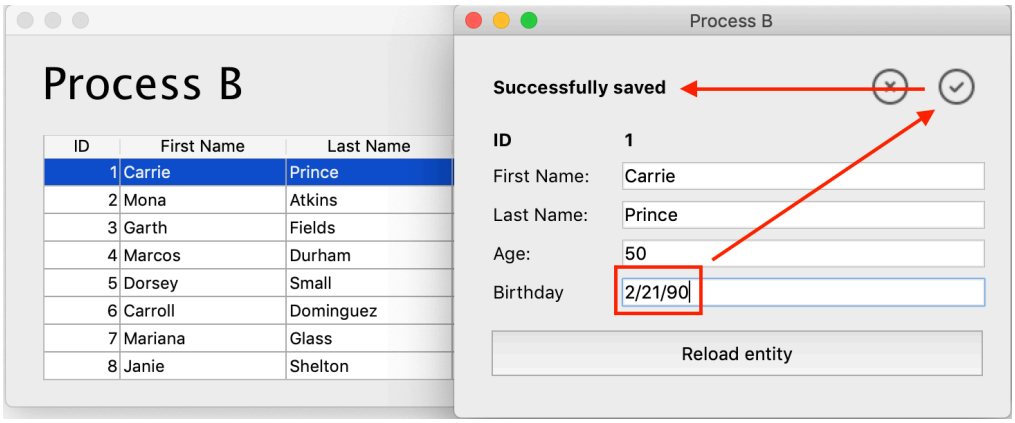

Image 17: Process B now edits birthday field again and saves successfully

Scenario 3: Optimistic locking using entity.save(dk auto merge)

The entity save command includes an additional mode called dk auto merge which automatically merges any changes made to an entity when loaded by multiple processes.

This mode alleviates the problem shown in scenario 2 where Process B had to reload the entity since Process A saved the entity first. Now using dk auto merge, both processes will save successfully.

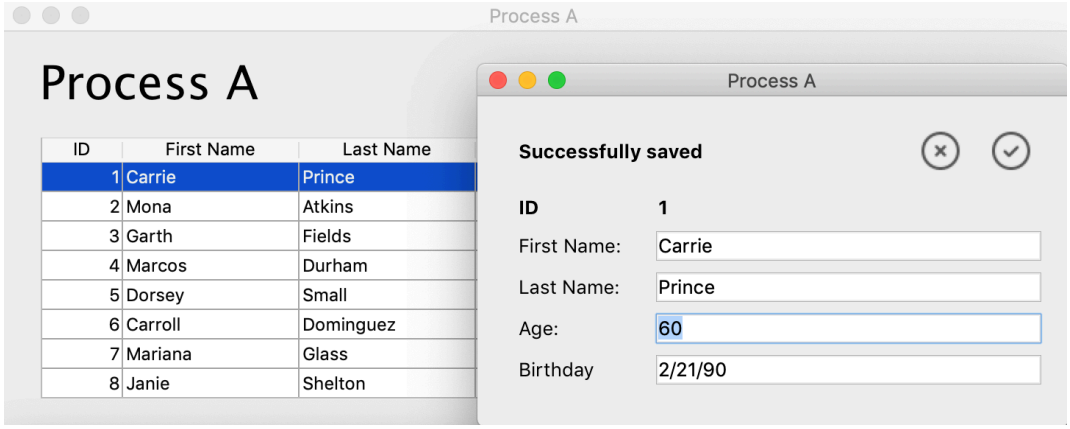

Image 18: Process A successfully saves changes to age field

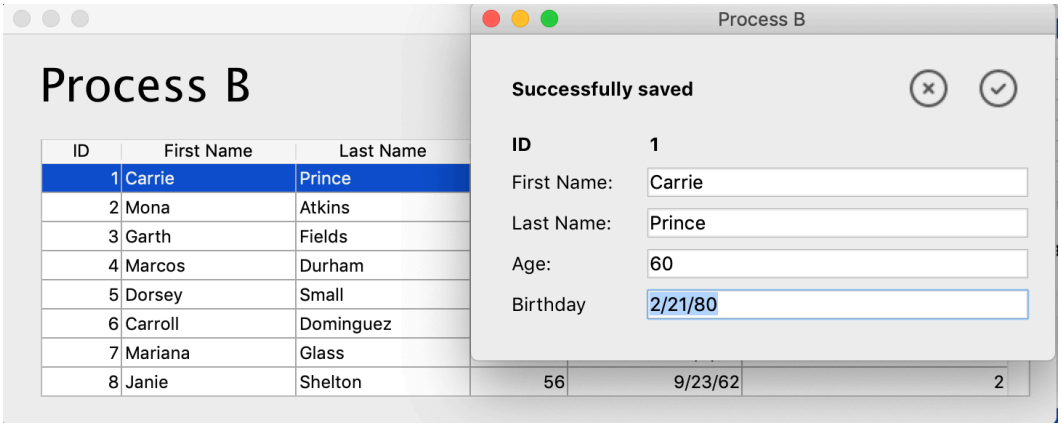

Image 19: Process B successfully saves changes to birthday field

However, the main caveat with dk auto merge mode is that the merge will fail for the subsequent process that attempted to save changes that were made on the same field. An example is provided below where both processes try save changes made to the age field.

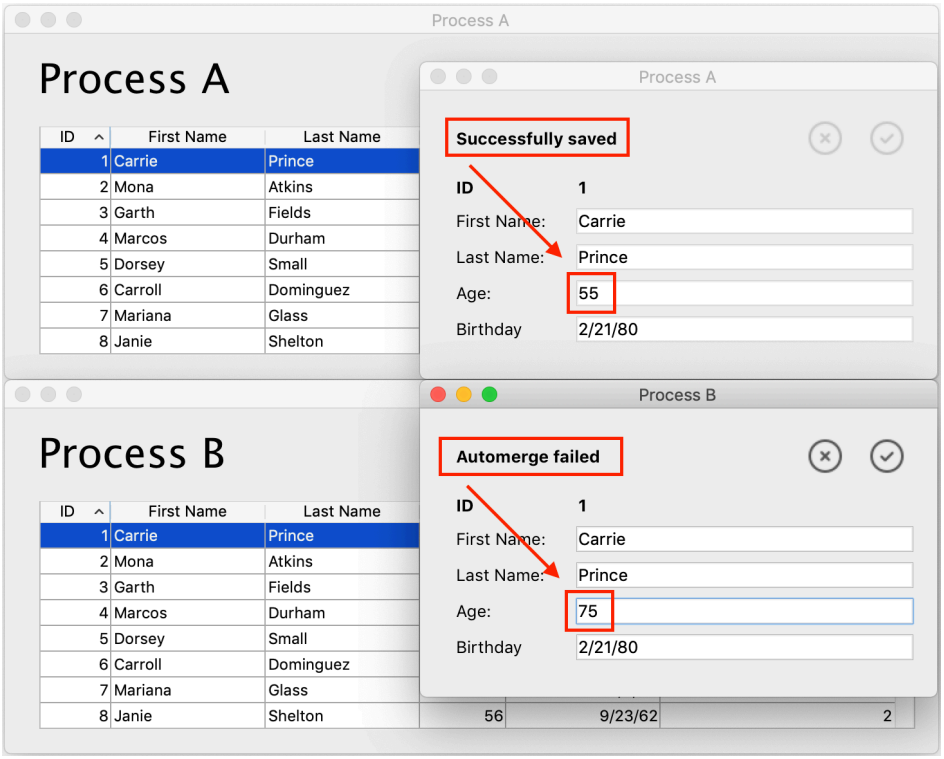

Image 20: Process B fails automerge due to modifying same field as Process A

To resolve any issue related to failed automerge or stamp change errors, Process B simply needs to reopen the form containing the entity or reload the entity to get the most updated data for that specific entity.

Conclusion

This technical note provided the general overview and differences between the two locking systems provided by classic 4D and ORDA entity locks. Classic 4D uses pessimistic locks where only one process is granted read/write access while all other processes attempting to load the same record is granted read-only access. ORDA by default provides optimistic locks where multiple processes are granted read/write access, but also allows pessimistic locks using commands like entity.lock() and entity.unlock(). Neither locking type is explicitly better than its counterpart, and it will be ultimately up to the developers to decide which system is better fit for their database.