Solving Difficult HTM Problems Without Difficult Hardware

Owen Hofmann, Donald Porter, Hany Ramadan,
Christopher Rossbach, and Emmett Witchel

University of Texas at Austin

Intro

- Processors now scaling via cores, not clock rate
  - 8 cores now, 64 tomorrow, 4096 next week?
- Parallel programming increasingly important
  - Software must keep up with hardware advances
  - But parallel programming is *really* hard
    - Deadlock, priority inversion, data races, etc.
- STM is here
- We would like HTM

Difficult HTM Problems

- Enforcing atomicity and isolation requires *conflict detection* and *rollback*
- TM Hardware only applies to memory and processor state
  - I/O, System calls may have effects that are not isolated, cannot be rolled back

```
mov $norad, %eax
mov $canada, %ebx
launchmissiles %eax, %ebx
```

Outline

- Handling kernel I/O with minimal hardware
- User-level system call rollback
- Conclusions

Outline

- Handling kernel I/O with minimal hardware
- User-level system call rollback
- Conclusions

cxspinlocks

- *Cooperative transactional spinlocks* allow kernel to take advantage of limited TM hardware
  - Optimistically execute with transactions
  - Fall back on locking when hardware TM is not enough
    - I/O, page table operations
    - overflow?
- Correctness provided by isolation of lock variable
  - Transactional threads read lock
  - Non-transactional threads write lock

cxspinlock guarantees

- Multiple transactional threads in critical region
- Non-transactional thread excludes all others

cxspinlocks in action

lockA:

unlocked

Thread 1

```
cx_optimistic(lockA);
modify_data();
if(condition) {
   perform_IO();
}
cx_end(lock);
```

read	write

Thread 2

```
cx_optimistic(lockA);
modify_data();
cx_end(lock);
```

read	write

cxspinlocks in action

- Transactional threads read unlocked lock variable

lockA:

unlocked

Thread 1: TX

```
cx_optimistic(lockA);
modify_data();
if(condition) {
   perform_IO();
}
cx_end(lock);
```

read	write
lockA	

Thread 2: TX

```
cx_optimistic(lockA);
modify_data();
cx_end(lock);
```

read	write
lockA	

cxspinlocks in action

- Transactional threads read unlocked lock variable

lockA:

unlocked

Thread 1: TX

```
cx_optimistic(lockA);
modify_data();
if(condition) {
   perform_IO();
}
cx_end(lock);
```

read	write
lockA	

Thread 2: TX

```
cx_optimistic(lockA);
modify_data();
cx_end(lock);
```

read	write
lockA	

cxspinlocks in action

- Transactional threads read unlocked lock variable

lockA:

unlocked

Thread 1: TX

```
cx_optimistic(lockA);
modify_data();
if(condition) {
   perform_IO();
}
cx_end(lock);
```

read	write
lockA	

Thread 2:

```
cx_optimistic(lockA);
modify_data();
cx_end(lock);
```

read	write
lockA	

cxspinlocks in action

lockA:

unlocked

Thread 1:

```
cx_optimistic(lockA);
modify_data();
if(condition) {
   perform_IO();
}
cx_end(lock);
```

read	write

Thread 2:

```
cx_optimistic(lockA);
modify_data();
cx_end(lock);
```

read	write

cxspinlocks in action

lockA:

unlocked

Thread 1:

```
cx_optimistic(lockA);
modify_data();
if(condition) {
   perform_IO();
}
cx_end(lock);
```

read	write

Thread 2:

```
cx_optimistic(lockA);
modify_data();
cx_end(lock);
```

read	write

cxspinlocks in action

lockA:

unlocked

Thread 1: TX

```
cx_optimistic(lockA);
modify_data();
if(condition) {
   perform_IO();
}
cx_end(lock);
```

read	write
lockA	

Thread 2: TX

```
cx_optimistic(lockA);
modify_data();
cx_end(lock);
```

read	write
lockA	

cxspinlocks in action

lockA:

unlocked

Thread 1: TX

```
cx_optimistic(lockA);
modify_data();
if(condition) {
   perform_IO();
}
cx_end(lock);
```

read	write
lockA	

Thread 2: TX

```
cx_optimistic(lockA);
modify_data();
cx_end(lock);
```

read	write
lockA	

cxspinlocks in action

lockA:

unlocked

Thread 1: TX

```
cx_optimistic(lockA);
modify_data();
if(condition) {
   perform_IO();
}
cx_end(lock);
```

read	write
lockA	

Thread 2: TX

```
cx_optimistic(lockA);
modify_data();
cx_end(lock);
```

read	write
lockA	

cxspinlocks in action

- Hardware restarts transactions for I/O

lockA:

unlocked

Thread 1: TX

```
cx_optimistic(lockA);
modify_data();
if(condition) {
   perform_IO();
}
cx_end(lock);
```

read	write
lockA	

Thread 2: TX

```
cx_optimistic(lockA);
modify_data();
cx_end(lock);
```

read	write
lockA	

cxspinlocks in action

lockA:

unlocked

Thread 1:

```
cx_optimistic(lockA);
modify_data();
if(condition) {
   perform_IO();
}
cx_end(lock);
```

read	write

Thread 2: TX

```
cx_optimistic(lockA);
modify_data();
cx_end(lock);
```

read	write
lockA	

cxspinlocks in action

- contention managed CAS

lockA:

unlocked

Thread 1:

```
cx_optimistic(lockA);
modify_data();
if(condition) {
   perform_IO();
}
cx_end(lock);
```

read	write

Thread 2: TX

```
cx_optimistic(lockA);
modify_data();
cx_end(lock);
```

read	write
lockA	

cxspinlocks in action

- contention managed CAS

lockA:

unlocked

Thread 1:

```
cx_optimistic(lockA);
modify_data();
if(condition) {
   perform_IO();
}
cx_end(lock);
```

read	write

Thread 2: TX

```
cx_optimistic(lockA);
modify_data();
cx_end(lock);
```

read	write
lockA	

cxspinlocks in action

- contention managed CAS

lockA:

unlocked

Thread 1:

```
cx_optimistic(lockA);
modify_data();
if(condition) {
   perform_IO();
}
cx_end(lock);
```

read	write

Thread 2:

```
cx_optimistic(lockA);
modify_data();
cx_end(lock);
```

read	write

cxspinlocks in action

lockA:

locked

Thread 1: Non-TX

```
cx_optimistic(lockA);
modify_data();
if(condition) {
   perform_IO();
}
cx_end(lock);
```

read	write

Thread 2:

```
cx_optimistic(lockA);
modify_data();
cx_end(lock);
```

read	write

cxspinlocks in action

lockA:

locked

Thread 1: Non-TX

```
cx_optimistic(lockA);
modify_data();
if(condition) {
   perform_IO();
}
cx_end(lock);
```

read	write

Thread 2:

```
cx_optimistic(lockA);
modify_data();
cx_end(lock);
```

read	write

cxspinlocks in action

lockA:

locked

Thread 1: Non-TX

```
cx_optimistic(lockA);
modify_data();
if(condition) {
   perform_IO();
}
cx_end(lock);
```

read	write

Thread 2:

```
cx_optimistic(lockA);
modify_data();
cx_end(lock);
```

read	write

cxspinlocks in action

- conditionally add variable to read set

lockA:

locked

Thread 1: Non-TX

```
cx_optimistic(lockA);
modify_data();
if(condition) {
   perform_IO();
}
cx_end(lock);
```

read	write

Thread 2: TX

```
cx_optimistic(lockA);
modify_data();
cx_end(lock);
```

read	write

cxspinlocks in action

- conditionally add variable to read set

lockA:

locked

Thread 1: Non-TX

```
cx_optimistic(lockA);
modify_data();
if(condition) {
   perform_IO();
}
cx_end(lock);
```

read	write

Thread 2: TX

```
cx_optimistic(lockA);
modify_data();
cx_end(lock);
```

read	write

cxspinlocks in action

- conditionally add variable to read set

lockA:

unlocked

Thread 1:

```
cx_optimistic(lockA);
modify_data();
if(condition) {
   perform_IO();
}
cx_end(lock);
```

read	write

Thread 2: TX

```
cx_optimistic(lockA);
modify_data();
cx_end(lock);
```

read	write

cxspinlocks in action

lockA:

unlocked

Thread 1:

```
cx_optimistic(lockA);
modify_data();
if(condition) {
   perform_IO();
}
cx_end(lock);
```

read	write

Thread 2: TX

```
cx_optimistic(lockA);
modify_data();
cx_end(lock);
```

read	write
lockA	

cxspinlocks in action

lockA:

unlocked

Thread 1:

```
cx_optimistic(lockA);
modify_data();
if(condition) {
   perform_IO();
}
cx_end(lock);
```

read	write

Thread 2: TX

```
cx_optimistic(lockA);
modify_data();
cx_end(lock);
```

read	write
lockA	

cxspinlocks in action

lockA:

unlocked

Thread 1:

```
cx_optimistic(lockA);
modify_data();
if(condition) {
   perform_IO();
}
cx_end(lock);
```

read	write

Thread 2:

```
cx_optimistic(lockA);
modify_data();
cx_end(lock);
```

read	write

Implementing cxspinlocks

- Return codes: Correctness
  - Hardware returns status code from xbegin to indicate when hardware has failed (I/O)
- xtest: Performance
  - Conditionally add a memory cell (e.g. lock variable) to the read set based on its value
- xcas: Fairness
  - Contention managed CAS
  - Non-transactional threads can wait for transactional threads
- Simple hardware primitives *support* complicated behaviors without *implementing* them

Outline

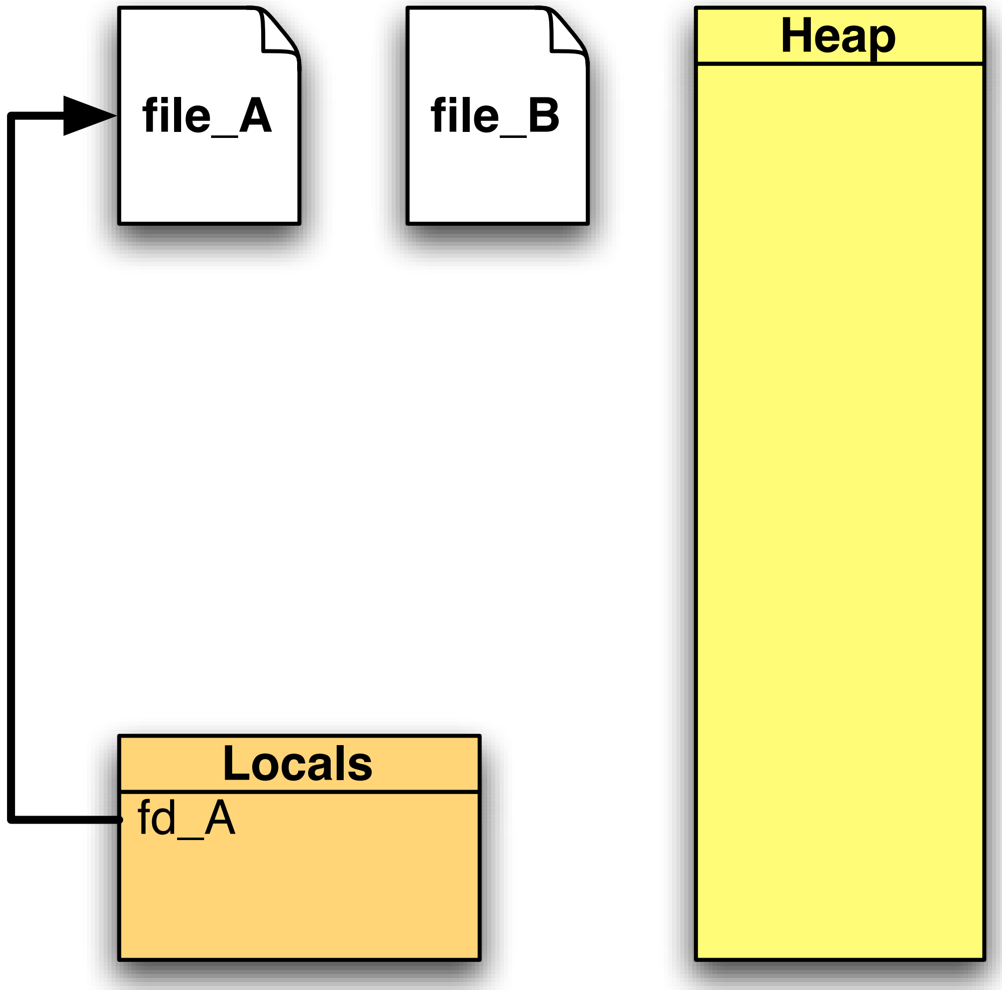
- Handling kernel I/O with minimal hardware
- User-level system call rollback
- Conclusions

User-level system call rollback

- Open nesting requires user-level syscall rollback
- Many calls have clear inverses
  - mmap, munmap
- Even simple calls have many side effects
  - e.g. file write
- Even simple calls might be irreversible

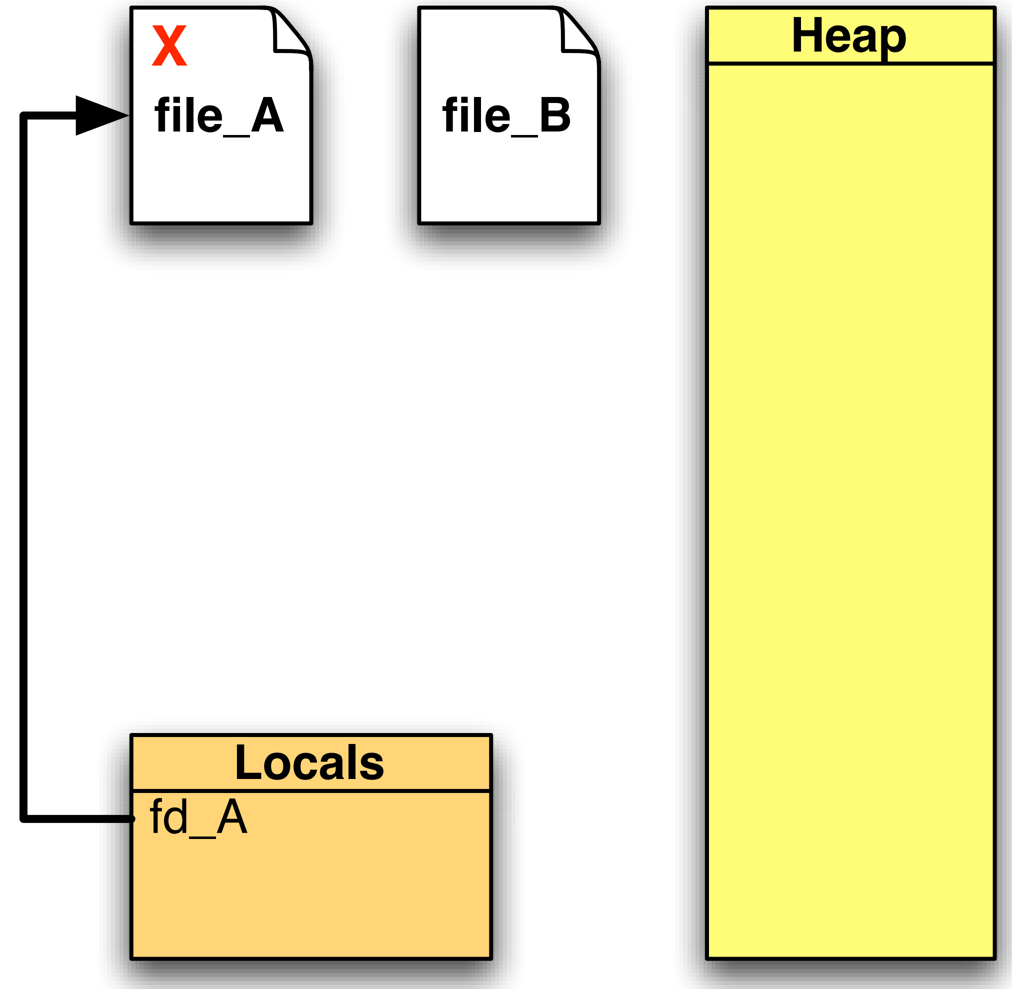
Users can’t always roll back

```
fd_A = open(“file_A”);
unlink(“file_A”);
void *map_A =
    mmap(fd=fd_A,
        size=4096);
close(fd_A);

xbegin;
modify_data();
fd_B = open(“file_B”);
xbegin_open;
void *map_B =
    mmap(fd=fd_B,
        start=map_A,
        size=4096);
xend_open(
    abort_action=munmap);
xrestart;
```



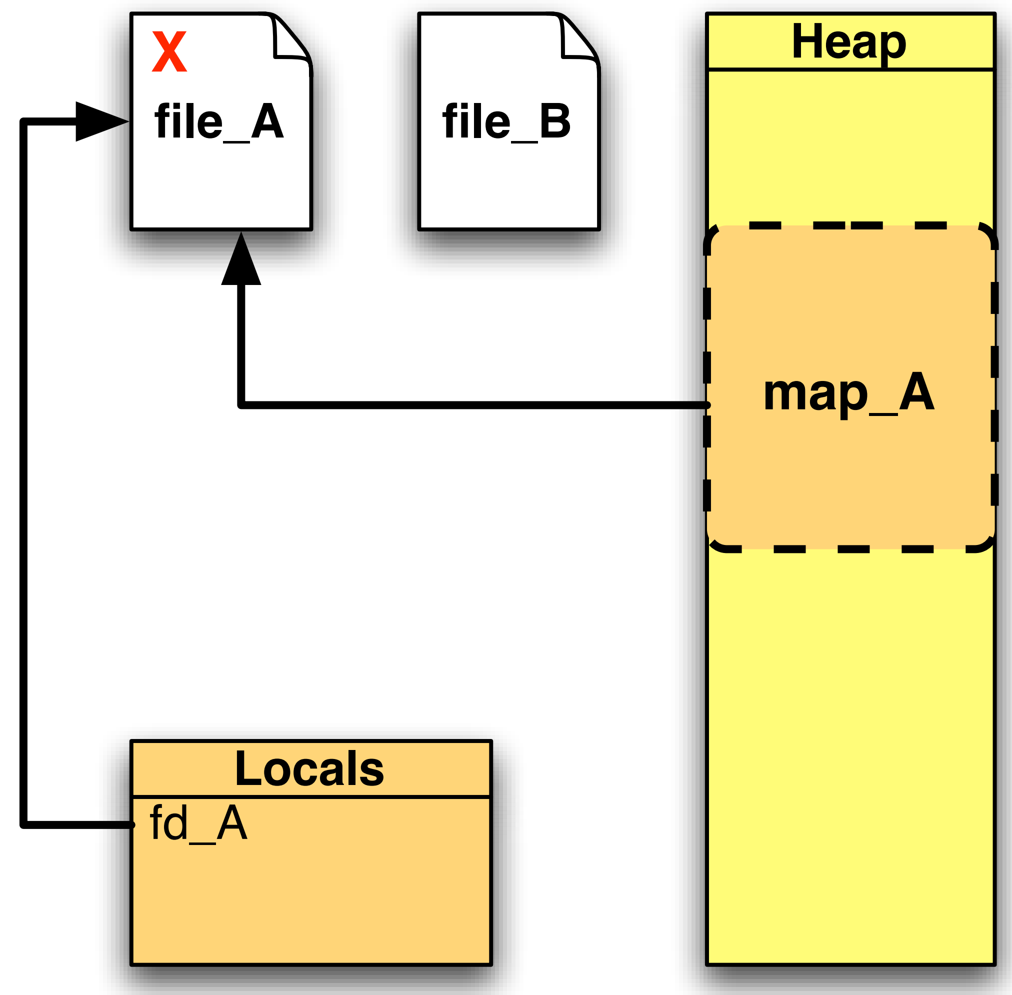
Users can't always roll back

```
fd_A = open("file_A");
unlink("file_A");
void *map_A =
    mmap(fd=fd_A,
        size=4096);
close(fd_A);

xbegin;
modify_data();
fd_B = open("file_B");
xbegin_open;
void *map_B =
    mmap(fd=fd_B,
        start=map_A,
        size=4096);
xend_open(
    abort_action=munmap);
xrestart;
```



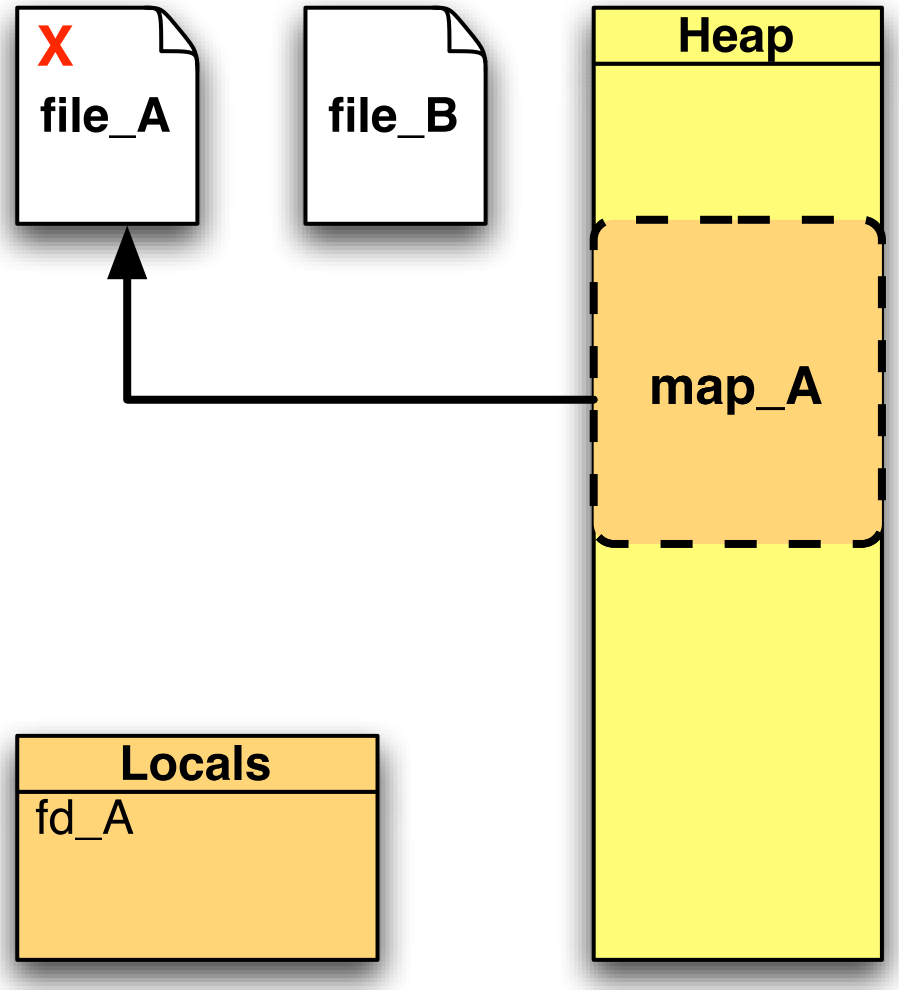
Users can’t always roll back

```
fd_A = open(“file_A”);
unlink(“file_A”);
void *map_A =
    mmap(fd=fd_A,
        size=4096);
close(fd_A);

xbegin;
modify_data();
fd_B = open(“file_B”);
xbegin_open;
void *map_B =
    mmap(fd=fd_B,
        start=map_A,
        size=4096);
xend_open(
    abort_action=munmap);
xrestart;
```



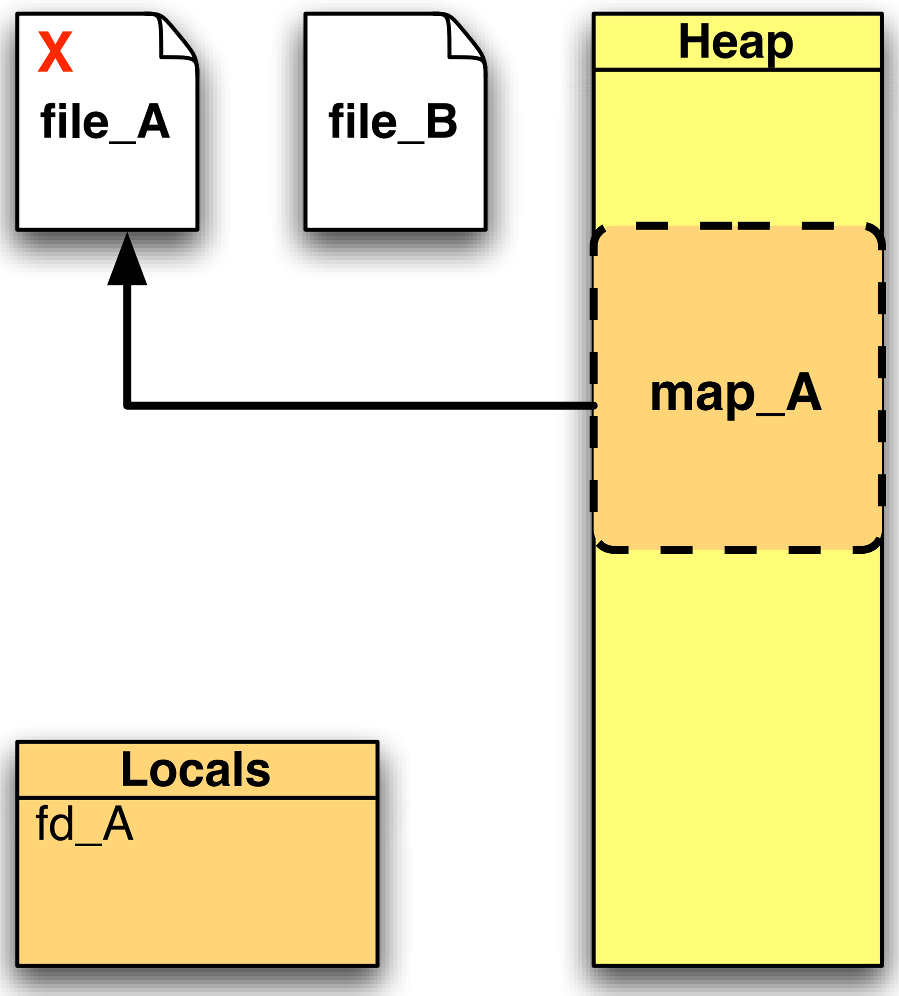
Users can’t always roll back

```
fd_A = open(“file_A”);
unlink(“file_A”);
void *map_A =
    mmap(fd=fd_A,
        size=4096);
close(fd_A);

xbegin;
modify_data();
fd_B = open(“file_B”);
xbegin_open;
void *map_B =
    mmap(fd=fd_B,
        start=map_A,
        size=4096);
xend_open(
    abort_action=munmap);
xrestart;
```



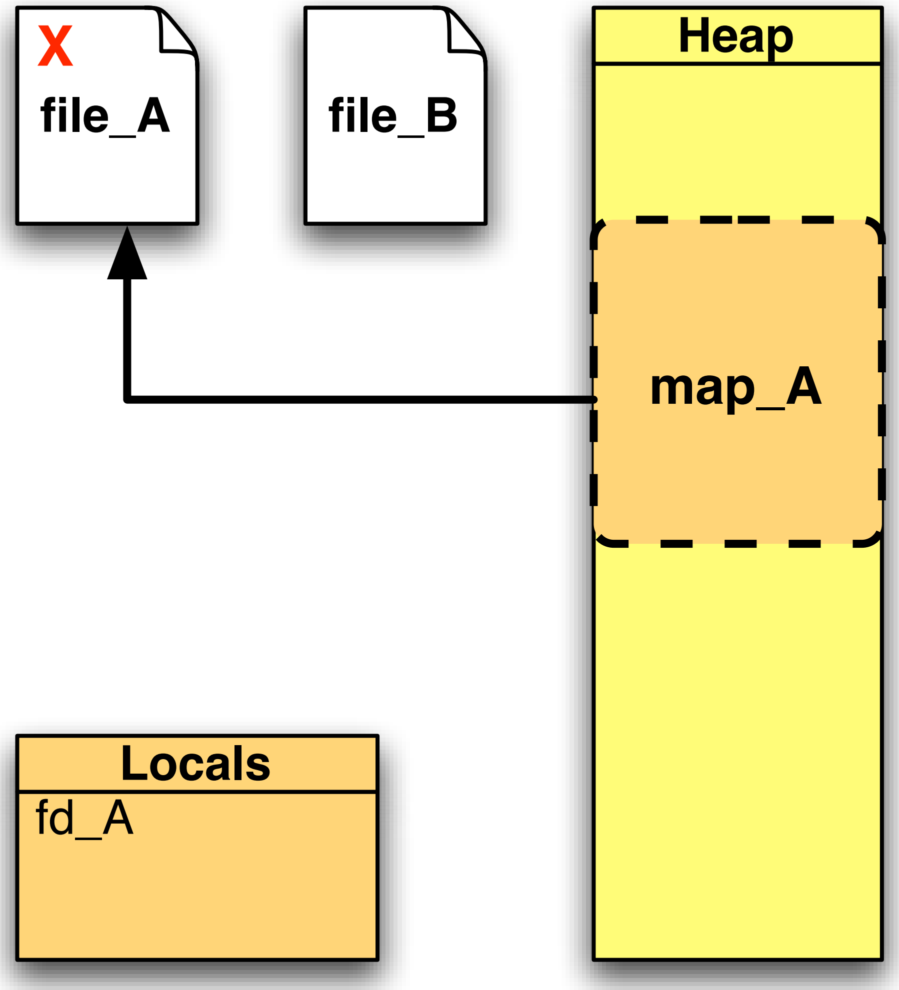
Users can’t always roll back

```
fd_A = open(“file_A”);
unlink(“file_A”);
void *map_A =
    mmap(fd=fd_A,
        size=4096);
close(fd_A);

xbegin;
modify_data();
fd_B = open(“file_B”);
xbegin_open;
void *map_B =
    mmap(fd=fd_B,
        start=map_A,
        size=4096);
xend_open(
    abort_action=munmap);
xrestart;
```



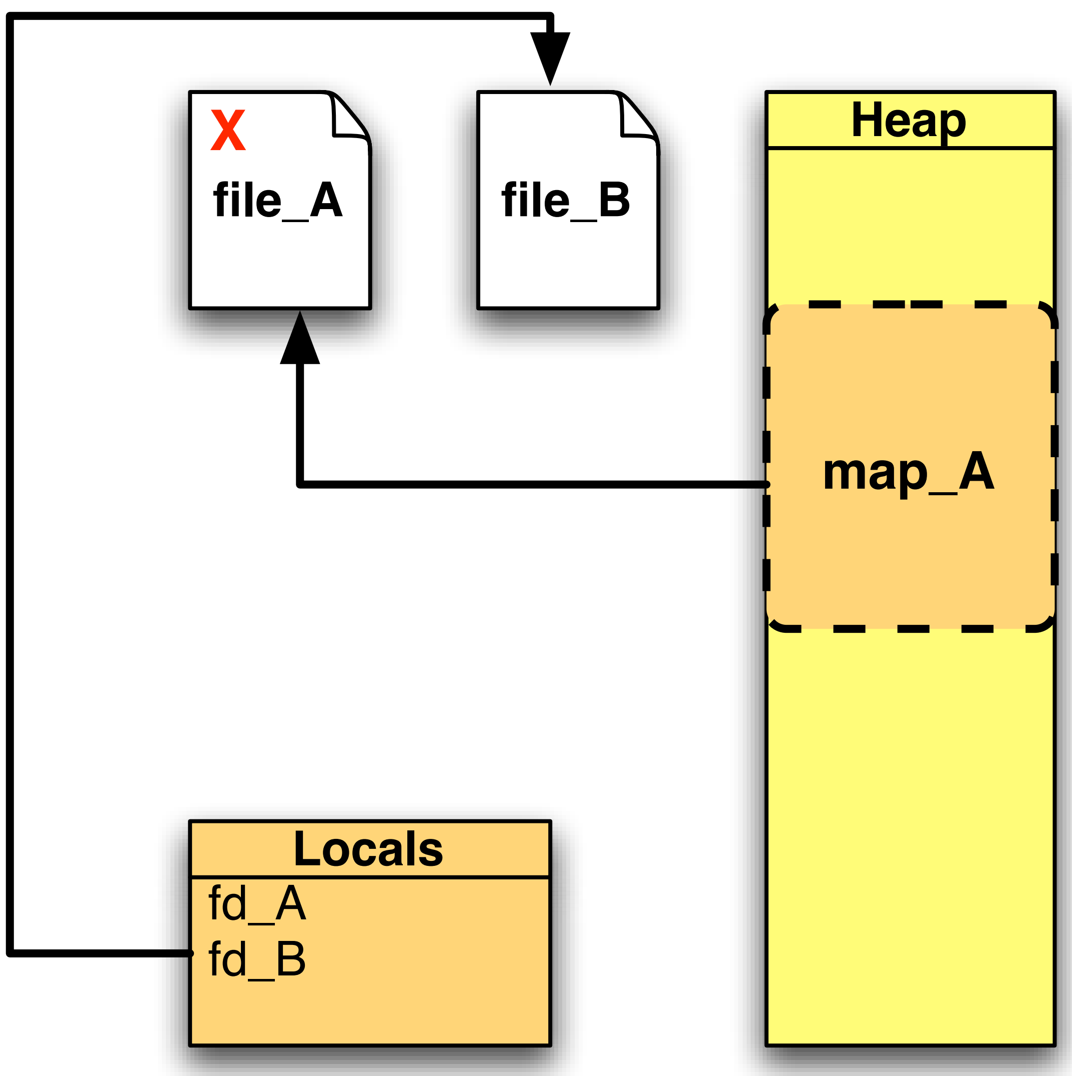
Users can’t always roll back

```
fd_A = open(“file_A”);
unlink(“file_A”);
void *map_A =
    mmap(fd=fd_A,
        size=4096);
close(fd_A);

xbegin;
modify_data();
fd_B = open(“file_B”);
xbegin_open;
void *map_B =
    mmap(fd=fd_B,
        start=map_A,
        size=4096);
xend_open(
    abort_action=munmap);
xrestart;
```



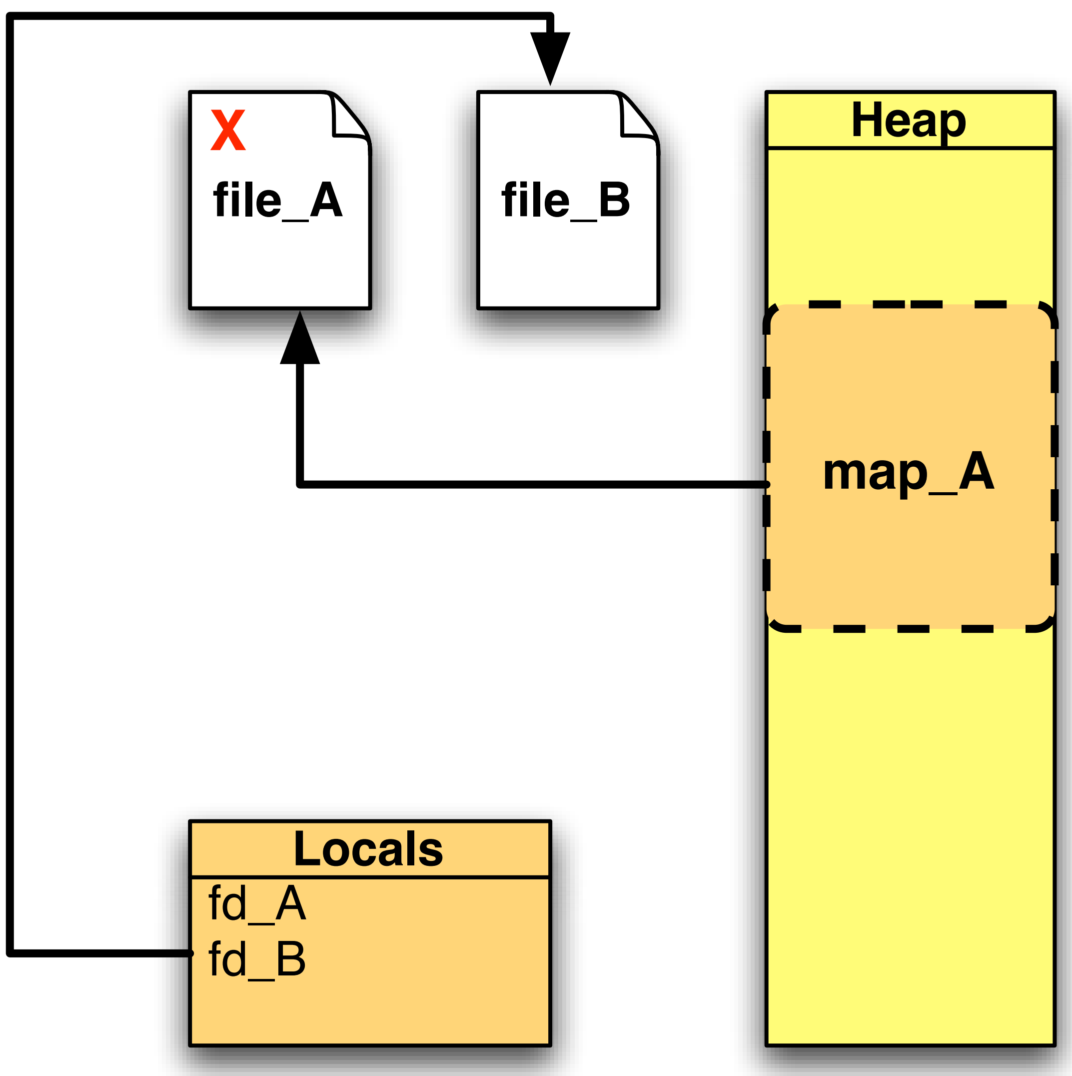
Users can’t always roll back

```
fd_A = open(“file_A”);
unlink(“file_A”);
void *map_A =
    mmap(fd=fd_A,
        size=4096);
close(fd_A);

xbegin;
modify_data();
fd_B = open(“file_B”);
xbegin_open;
void *map_B =
    mmap(fd=fd_B,
        start=map_A,
        size=4096);
xend_open(
    abort_action=munmap);
xrestart;
```



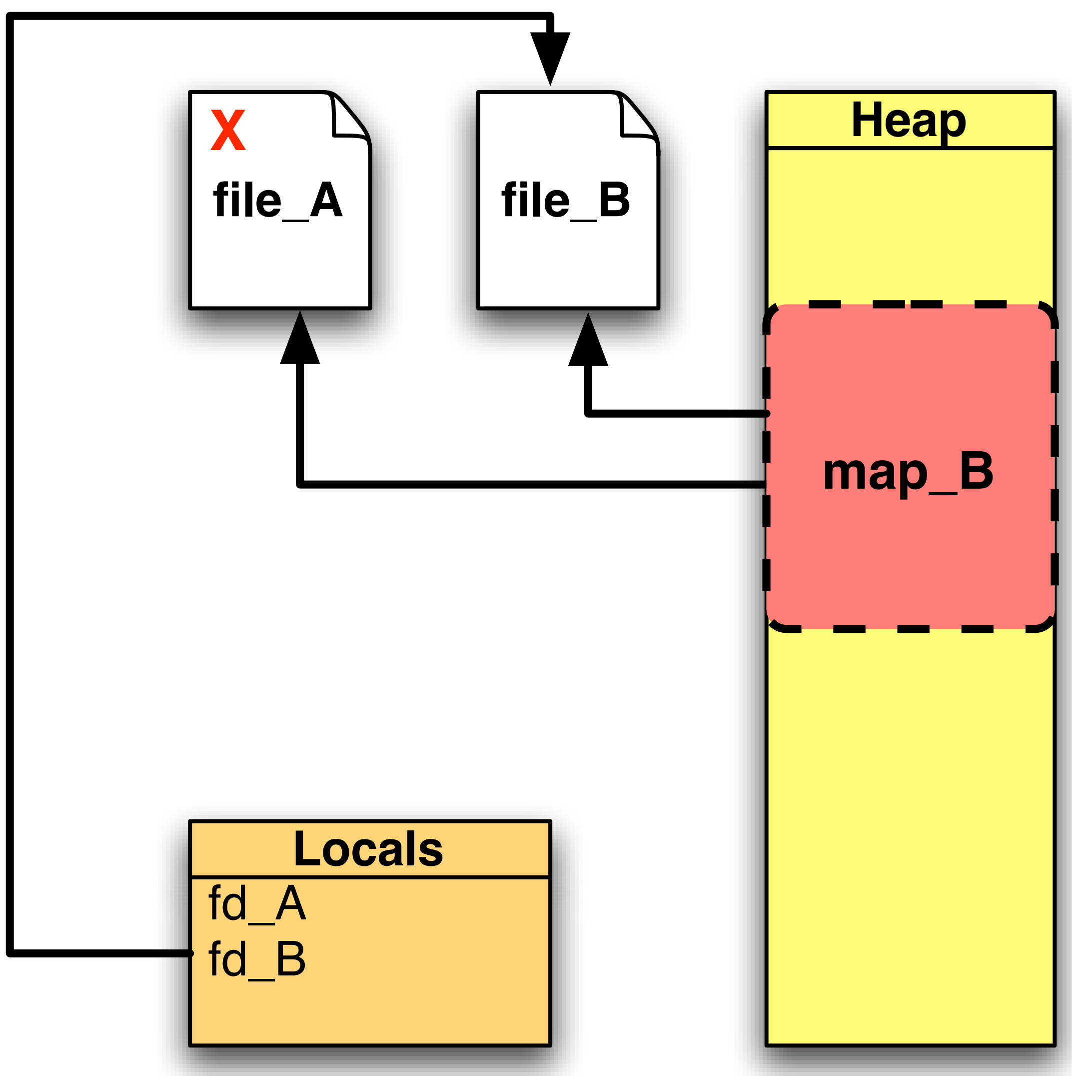
Users can't always roll back

```
fd_A = open("file_A");
unlink("file_A");
void *map_A =
    mmap(fd=fd_A,
        size=4096);
close(fd_A);

xbegin;
modify_data();
fd_B = open("file_B");
xbegin_open;
void *map_B =
    mmap(fd=fd_B,
        start=map_A,
        size=4096);
xend_open(
    abort_action=munmap);
xrestart;
```



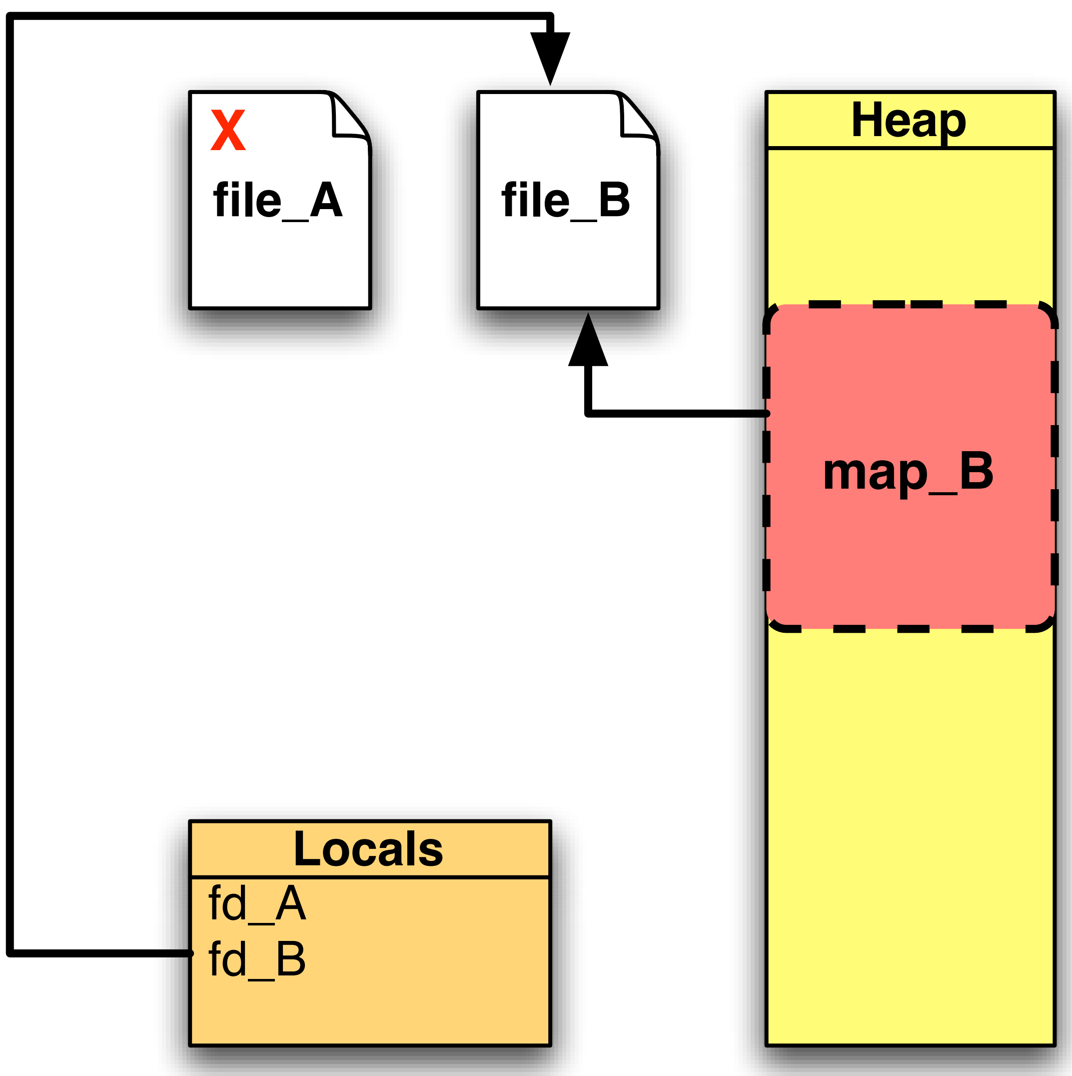
Users can't always roll back

```
fd_A = open("file_A");
unlink("file_A");
void *map_A =
    mmap(fd=fd_A,
        size=4096);
close(fd_A);

xbegin;
modify_data();
fd_B = open("file_B");
xbegin_open;
void *map_B =
    mmap(fd=fd_B,
        start=map_A,
        size=4096);
xend_open(
    abort_action=munmap);
xrestart;
```



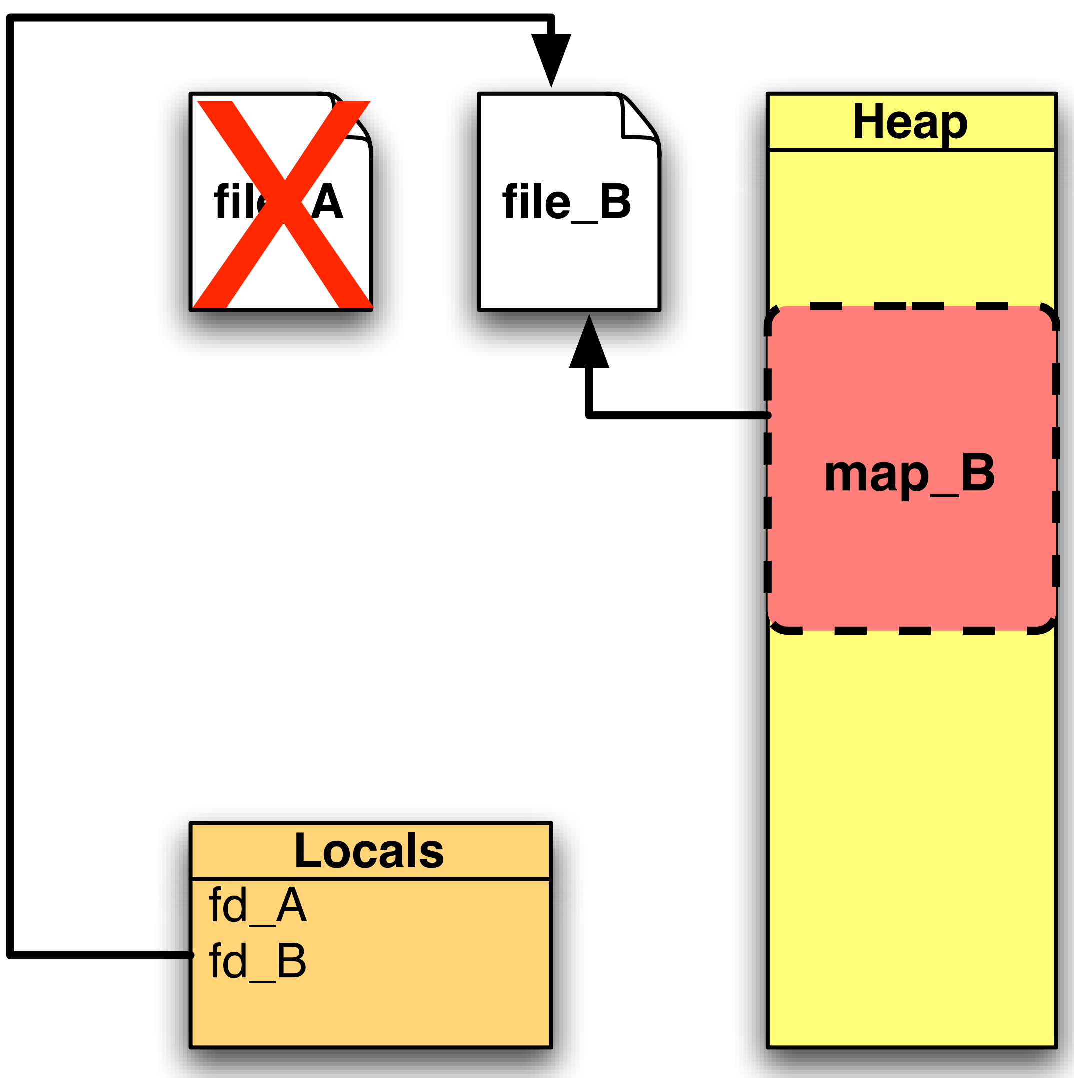
Users can’t always roll back

```
fd_A = open(“file_A”);
unlink(“file_A”);
void *map_A =
    mmap(fd=fd_A,
        size=4096);
close(fd_A);

xbegin;
modify_data();
fd_B = open(“file_B”);
xbegin_open;
void *map_B =
    mmap(fd=fd_B,
        start=map_A,
        size=4096);
xend_open(
    abort_action=munmap);
xrestart;
```



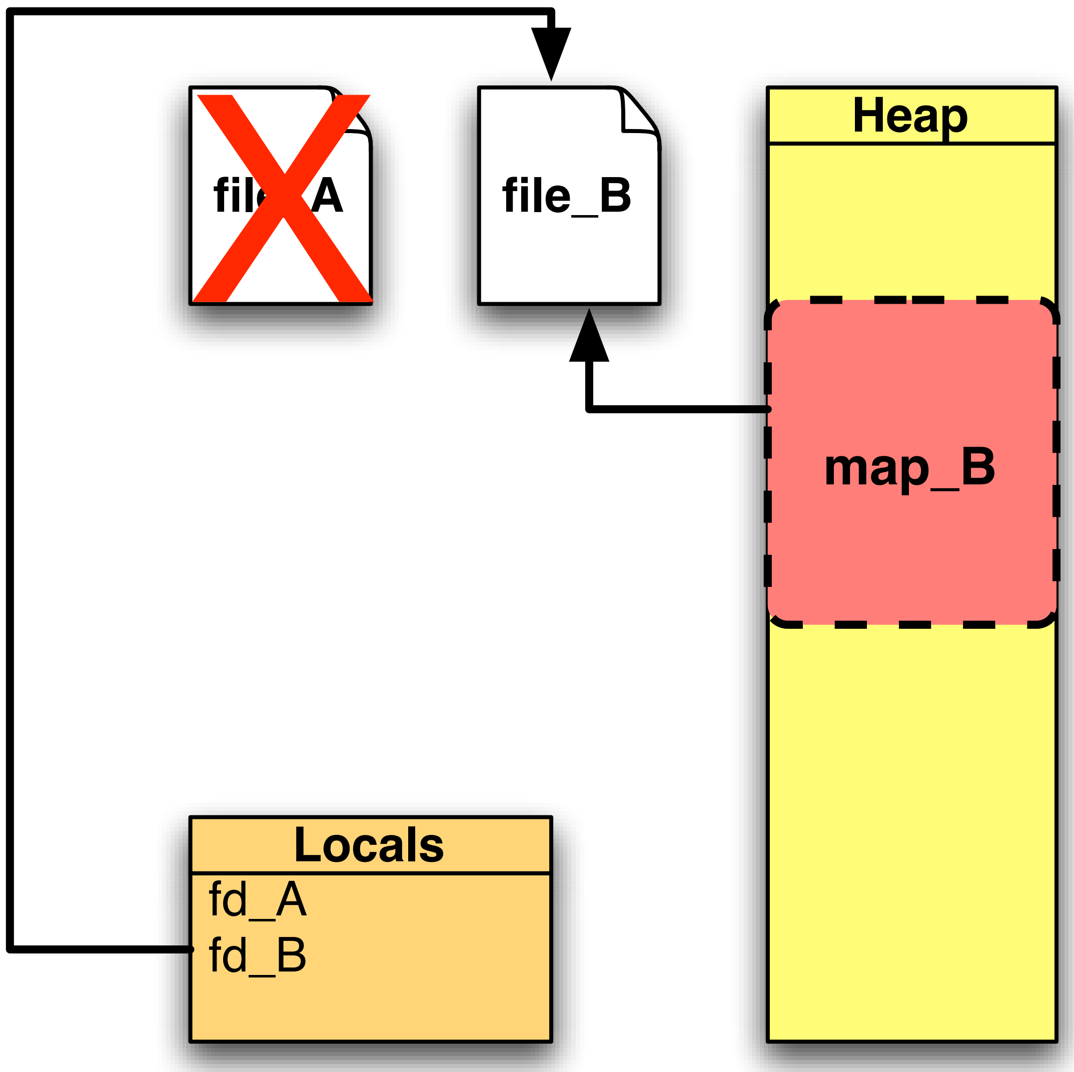
Users can't always roll back

```
fd_A = open("file_A");
unlink("file_A");
void *map_A =
    mmap(fd=fd_A,
        size=4096);
close(fd_A);

xbegin;
modify_data();
fd_B = open("file_B");
xbegin_open;
void *map_B =
    mmap(fd=fd_B,
        start=map_A,
        size=4096);
xend_open(
    abort_action=munmap);
xrestart;
```



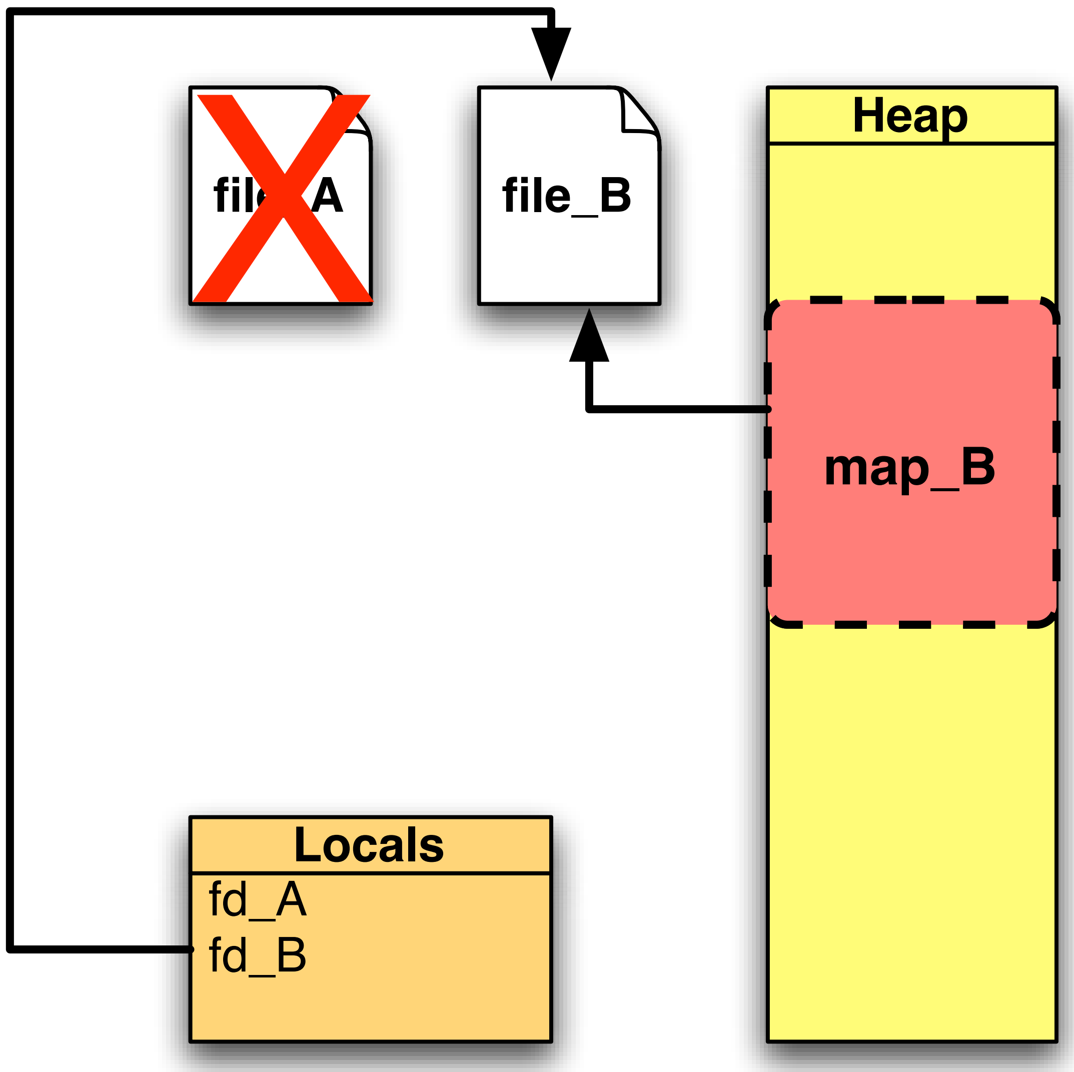
Users can't always roll back

```
fd_A = open("file_A");
unlink("file_A");
void *map_A =
    mmap(fd=fd_A,
        size=4096);
close(fd_A);

xbegin;
modify_data();
fd_B = open("file_B");
xbegin_open;
void *map_B =
    mmap(fd=fd_B,
        start=map_A,
        size=4096);
xend_open(
    abort_action=munmap);
xrestart;
```



Users can't always roll back

```
fd_A = open("file_A");
unlink("file_A");
void *map_A =
    mmap(fd=fd_A,
        size=4096);
close(fd_A);

xbegin;
modify_data();
fd_B = open("file_B");
xbegin_open;
void *map_B =
    mmap(fd=fd_B,
        start=map_A,
        size=4096);
xend_open(
    abort_action=munmap);
xrestart;
```



Kernel participation required

- Users can’t track all syscall side effects
  - Must also track all non-tx syscalls
- Kernel must manage transactional syscalls
- Kernel enhances user-level programming model
  - Seamless transactional system calls
  - Strong isolation for system calls?

Related Work

- Other I/O solutions
  - Hardware open nesting [Moravan 06]
  - Unrestricted transactions [Blundell 07]
- Transactional Locks
  - Speculative lock elision [Rajwar 01]
  - Transparently Reconciling Transactions & Locks [Welc 06]
- Simple hardware models
  - Hybrid TM [Damron 06, Shriraman 07]
  - Hardware-accelerated STM [Saha 2006]

Conclusions

- Kernel can use even minimal hardware TM designs
  - May not get full programming model
  - Simple primitives support complex behavior
- User-level programs can't roll back system calls
  - Kernel must participate