8. Concurrency and Mutual Exclusion

- Mutual Exclusion in Shared Memory
- Semaphores
- Mutual Exclusion and Deadlocks
- File Locks
- System V IPC

Concurrent Resource Management

Concurrency Issues

- Multiple *non-modifying accesses* to *shared resources* may occur in parallel without conflict
- Problems arise when *accessing a shared resource* to *modify its state*
  - Concurrent file update
  - Concurrent shared memory update
- General problem: enforcing *mutual exclusion*

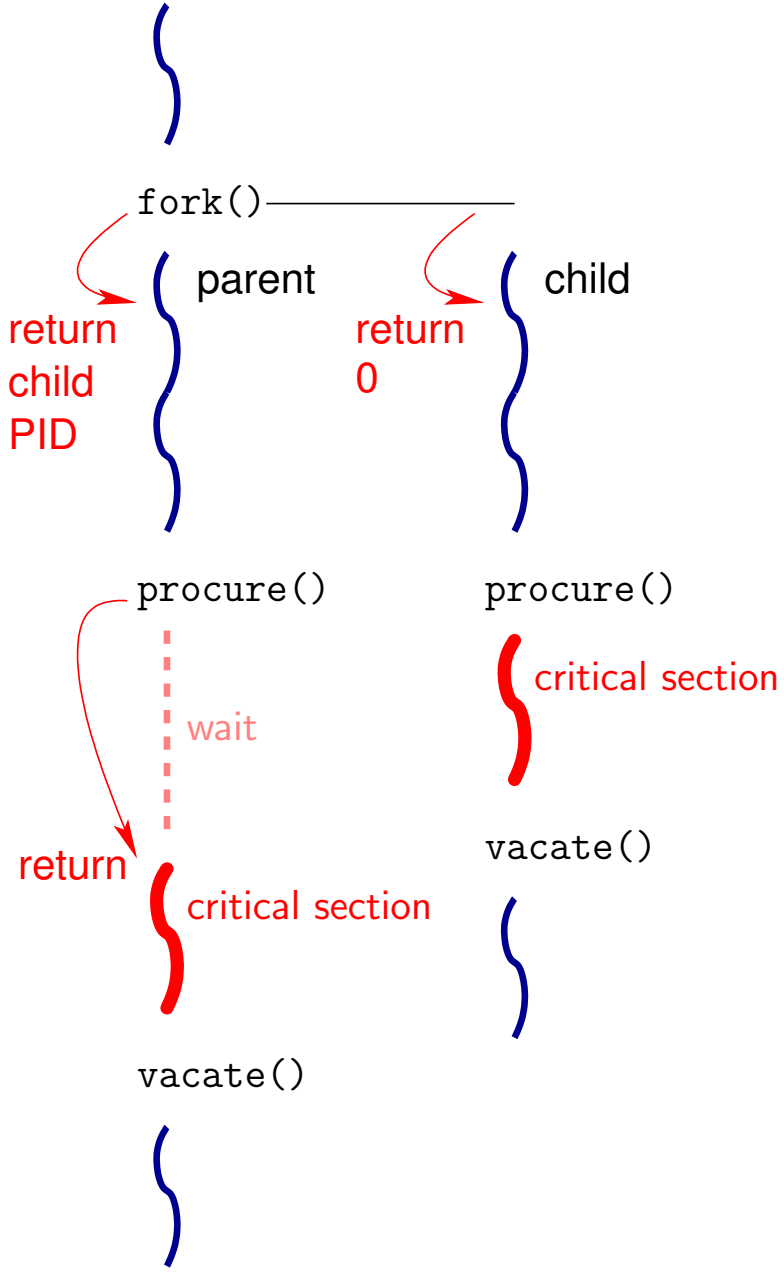
Principles of Concurrent Resource Management

Critical Section

- Program section accessing shared resource(s)
- *Only one process can be in this section at a time*

Mutual Exclusion

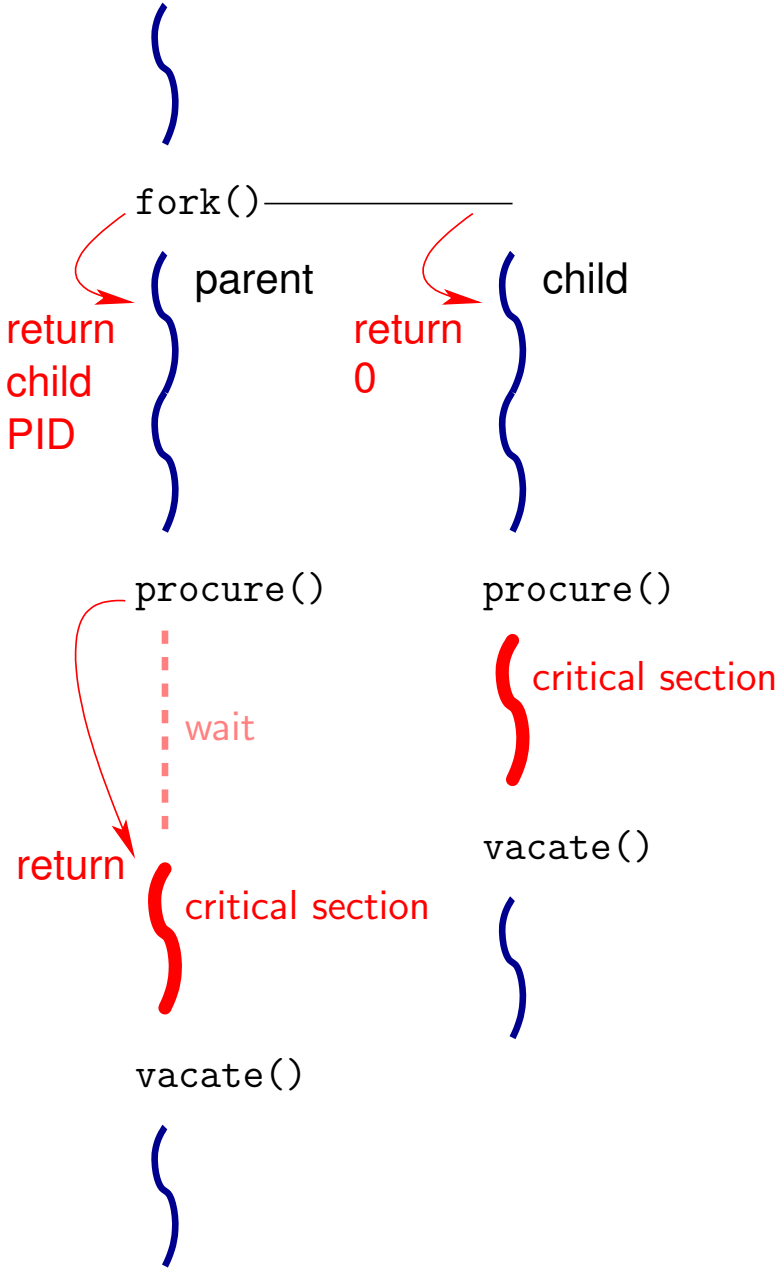
- Make sure at most one process may enter a critical section
- Typical cases
  - Implementing file locks
  - Concurrent accesses to shared memory



Principles of Concurrent Resource Management

Source of Major Headaches

- *Correctness*: prove process alone in critical section
- *Absence of deadlock*, or detection and lock-breaking
- *Guaranteed progress*: a process enters critical section if it is the only one to attempt to do it
- *Bounded waiting*: a process waiting to enter a critical section will eventually (better sooner than later) be authorized to do so
- *Performance*: reduce overhead and allow parallelism to scale



8. Concurrency and Mutual Exclusion

Mutual Exclusion in Shared Memory

Dekker's Algorithm

```
int try0 = 0, try1 = 0;
int turn = 0; // Or 1

// Fork processes sharing variables try0, try1, turn
// Process 0                                  // Process 1
try0 = 1;                                     try1 = 1;
while (try1 != 0)                              while (try0 != 0)
  if (turn != 0) {                              if (turn != 1) {
    try0 = 0;                                     try1 = 0;
    while (try1 != 0) { }                         while (try0 != 0) { }
    try0 = 1;                                     try1 = 1;
  }                                             }
turn = 0;                                     turn = 1;
// Critical section                           // Critical section
try0 = 0;                                     try1 = 0;
// Non-critical section                       // Non-critical section
```

Mutual Exclusion in Shared Memory

Peterson's Algorithm

```
int try0 = 0, try1 = 0;
int turn = 0; // Or 1

// Fork processes sharing variables try0, try1, turn
// Process 0                          // Process 1
try0 = 1;                             try1 = 1;
turn = 0;                             turn = 1;
while (try1 && !turn) { }             while (try0 && turn) { }
// Critical section                   // Critical section
try0 = 0;                             try1 = 0;
// Non-critical section               // Non-critical section
```

- Unlike Dekker's algorithm, enforces fair turn alternation
- Simpler and easily extensible to more than two processes

Shared Memory Consistency Models

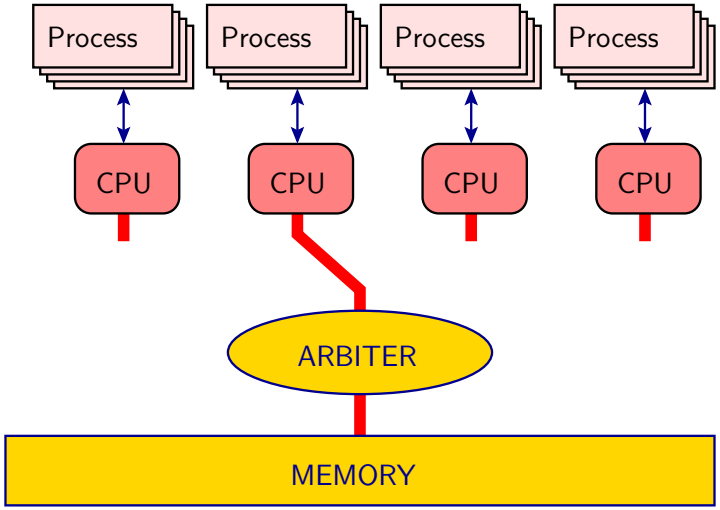
Memory Consistency

- "When the outcome of a memory access is visible from another process"
- Dekker and Petersen algorithms require the strongest memory model: *sequential consistency*
- Definition of sequential consistency by Leslie Lamport:
  "*The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*"

Shared Memory Consistency Models

Memory Consistency

- "When the outcome of a memory access is visible from another process"
- Dekker and Petersen algorithms require the strongest memory model: *sequential consistency*
- Definition of sequential consistency by Leslie Lamport:
  "*The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*"



Shared Memory Consistency Models

Memory Consistency

- "When the outcome of a memory access is visible from another process"
- Dekker and Petersen algorithms require the strongest memory model: *sequential consistency*
- Definition of sequential consistency by Leslie Lamport:
  "*The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*"

Weak Consistency Models

- Hardware, run-time libraries and compilers prefer *weaker consistency models*
  - Compiler optimizations: loop-invariant code motion, instruction reordering
  - Hardware/run-time optimizations: out-of-order superscalar execution (local), out-of-order cache coherence (multi-processor)
- Impossibility result for mutual exclusion: Attiya et al. POPL 2011

Memory Consistency Examples

Paradoxical Example

```
int f = 1;
int x = 0;

// Fork processes sharing variables f, x
// Process 0                                  // Process 1
while (f) { }                                 x = 1;
printf("x = %d\n", x);                        f = 0;
```

Analysis

- What is the value of `x` printed by `Process 0`?
  (assuming no other process may access the shared variables)

Memory Consistency Examples

Paradoxical Example

```
int f = 1;
int x = 0;

// Fork processes sharing variables f, x
// Process 0                                  // Process 1
while (f) { }                                 x = 1;
printf("x = %d\n", x);                        f = 0;
```

Analysis

- What is the value of `x` printed by `Process 0`?
  (assuming no other process may access the shared variables)
  - **1** with sequential consistency

Memory Consistency Examples

Paradoxical Example

```
int f = 1;
int x = 0;

// Fork processes sharing variables f, x
// Process 0                                  // Process 1
while (f) { }                                 x = 1;
printf("x = %d\n", x);                        f = 0;
```

Analysis

- What is the value of `x` printed by `Process 0`?
  (assuming no other process may access the shared variables)
  - **1** with sequential consistency
  - May be **0** with weaker models

Solution: Hardware Support

Serializing Memory Accesses

- Memory *fences* (for the hardware and compiler)
  - Multiprocessor
    → In general, commit all pending memory and cache coherence transactions
  - Uniprocessor (cheaper and weaker)
    → Commit all local memory accesses
  - Can be limited to read or write accesses
  - Depending on the memory architecture, cheaper implementations are possible
  - Forbids cross-fence code motion by the compiler
- ISO C `volatile` attribute (for the compiler)
  - `volatile int x`
    Informs the compiler that asynchronous modifications of `x` may occur
  - No compile-time reordering of accesses to volatile variables
  - Never consider accesses to volatile variables as dead code
- Combining fences and volatile variables fixes the problems of Dekker's and Peterson's algorithms
- Modern programming languages tend to merge both forms into more abstract constructs (e.g., Java 5)

Solution: Hardware Support

Atomic Operations

- Fine grain *atomic operations* permit higher performance than synchronization algorithms with fences
  - *Atomic Exchange*: exchange value of a register and a memory location, atomically
  - *Test-and-Set*: set a memory location to **1** and return whether the old value was null or not, atomically
    - Can be implemented with atomic exchange

```
int test_and_set(int *lock_pointer) {
  int old_value = 0;
  if (*lock_pointer)
    old_value = atomic_exchange(lock_pointer, 1);
  return old_value != 0;
}
```

Solution: Hardware Support

Atomic Operations

- Fine grain *atomic operations* permit higher performance than synchronization algorithms with fences
  - More powerful: *Compare-and-Swap* (cannot be implemented with the former)

```
bool compare_and_swap(int *accum, int *dest, int newval) {
  if (*accum == *dest) {
      *dest = newval;
      return true;
  } else {
      *accum = *dest;
      return false;
  }
```

  - Many others, implementable with atomic exchange or compare-and-swap, with or without additional control flow

8. Concurrency and Mutual Exclusion

From Simple Locks to Semaphores

```
void lock(volatile int *lock_pointer) {
  while (test_and_set(lock_pointer) == 1);
}
void unlock(volatile int *lock_pointer) {
  *lock_pointer = 0 // Release lock
}

int lock_variable = 1;
void lock_example() {
  lock(&lock_variable);
  // Critical section
  unlock(&lock_variable);
}
```

Generalization to Countable Resources: Semaphores

- Atomic increment/decrement primitives
  - **P()** or `procure()` from "*proberen*" (Dutch for "to wait")
  - **V()** or `vacate()` from "*verhogen*" (Dutch for "to increment")
- May use *simple lock* to implement atomicity

Semaphore

Unified Structure and Primitives for Mutual Exclusion

- Initialize the semaphore with `v` instances of the resource to manage

```
void init(semaphore s, int v) {
  s.value = v;
}
```

- Acquire a resource (entering a critical section)

```
void procure(semaphore s) {
  wait until (s.value > 0);
  s.value--;     // Must be atomic with the previous test
}
```

  Also called `down()` or `wait()`

- Release a resource (leaving a critical section)

```
void vacate(semaphore s) {
  s.value++;     // Must be atomic
}
```

  Also called `up()`, `post()` or `signal()`

Heterogeneous Read-Write Mutual Exclusion

Read-Write Semaphores

- Allowing *multiple readers* and a *single writer*

```
void init(rw_semaphore l) {
  l.value = 0;   // Number of readers (resp. writers)
                 //   if positive (resp. negative)
}
void procure_read(rw_semaphore l) {
  wait until (l.value >= 0);
  l.value++;     // Must be atomic with the previous test
}
void vacate_read(rw_semaphore l) {
  l.value--;     // Must be atomic
}
void procure_write(rw_semaphore l) {
  wait until (l.value == 0);
  l.value = -1;  // Must be atomic with the previous test
}
void vacate_write(rw_semaphore l) {
  l.value = 0;
}
```

IPC: Semaphores

POSIX Semaphores

- Primitives: `sem_wait()` (`procure()`) and `sem_post()` (`vacate()`)
  - `sem_wait()` blocks until the value of the semaphore is greater than **0**, then decrements it and returns
  - `sem_post()` increments the value of the semaphore and returns
- They can be named (associated to a file) or not
- `$ man 7 sem_overview`

Implementation in Linux

- Semaphore files are single inodes located in a specific *pseudo-file-system*, mounted under `/dev/shm`
- Must link the program with `-lrt` (real-time library)

System Call: sem_open()

Open and Possibly Create a POSIX Semaphore

```
#include <semaphore.h>

sem_t *sem_open(const char *name, int flags);
sem_t *sem_open(const char *name, int flags,
                mode_t mode, unsigned int value);
```

Description

- Arguments `flags` and `mode` allow for a subset of their values for `open()`
  - `flags`: only `O_CREAT`, `O_EXCL`; and `FD_CLOEXEC` flag is set automatically
  - `mode`: `S_IRUSR`, `S_IWUSR`, `S_IXUSR`, etc.
- `value` is used to initialize the semaphore, defaults to **1** if not specified
- Return the address of the semaphore on success
- Return `SEM_FAILED` on error (i.e., `(sem_t*)0`)

System Call: `sem_wait()`

Lock a POSIX Semaphore

```
#include <semaphore.h>

int sem_wait(sem_t *sem);
```

Description

- Block until the value of the semaphore is greater than **0**, then decrements it and returns
- Return **0** on success, **−1** on error

System Call: sem_post()

Unlock a POSIX Semaphore

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

Description

- Increment the value of the semaphore pointed to by `sem`
- Return **0** on success, **−1** on error

System Call: sem_close()

Close a POSIX Semaphore Structure

```
#include <semaphore.h>

int sem_close(sem_t *sem);
```

Description

- Similar to `close()` for semaphore pointers
- Undefined behavior when closing a semaphore other processes are currently blocked on

System Call: sem_unlink()

Unlink a POSIX Semaphore File

```
#include <semaphore.h>

int sem_unlink(const char *name);
```

Description

- Semaphores files have *kernel* persistence
- Similar to `unlink()`

Other System Calls

- `sem_init()` and `sem_destroy()`: create unnamed semaphores and destroy them (equivalent to combined `sem_close()` and `sem_unlink()`)
- `sem_getvalue()`: get the current value of a semaphore
- `sem_trywait()` and `sem_timedwait()`: non-blocking and timed versions of `sem_wait()`

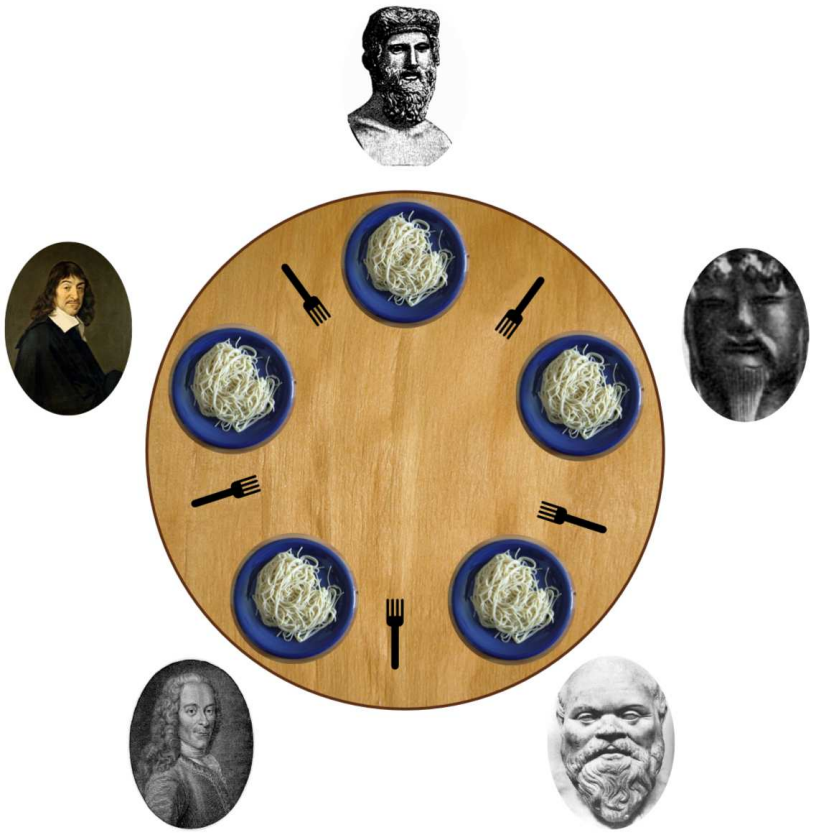
8. Concurrency and Mutual Exclusion

Mutual Exclusion and Deadlocks

Dining Philosophers Problem

- Due to Edsger Dijkstra and Tony Hoare
  - Eating requires two chopsticks (more realistic than forks...)
  - A philosopher may only use the closest left and right chopsticks

  Multiple processes acquire multiple resources
- Deadlock: all philosophers pick their left chopstick, *then* attempt to pick their right one
- Hard to debug non-reproducible deadlocks



Mutual Exclusion and Deadlocks

Preventing Deadlocks

- Eliminate symmetric or cyclic acquire/release patterns
  - Not always possible/desirable

Avoiding Deadlocks

- Use higher-level mutual exclusion mechanisms
  - Monitors
  - Atomic transactions
- Dynamic deadlock avoidance
  - Build a graph of resource usage
  - Detect and avoid cycles
  - Banker's algorithm for counted resources

Mutual Exclusion and Deadlocks

Breaking Deadlocks

1. Timeout
2. Analyze the situation
3. Attempt to reacquire different resources or in a different order

Beyond Deadlocks

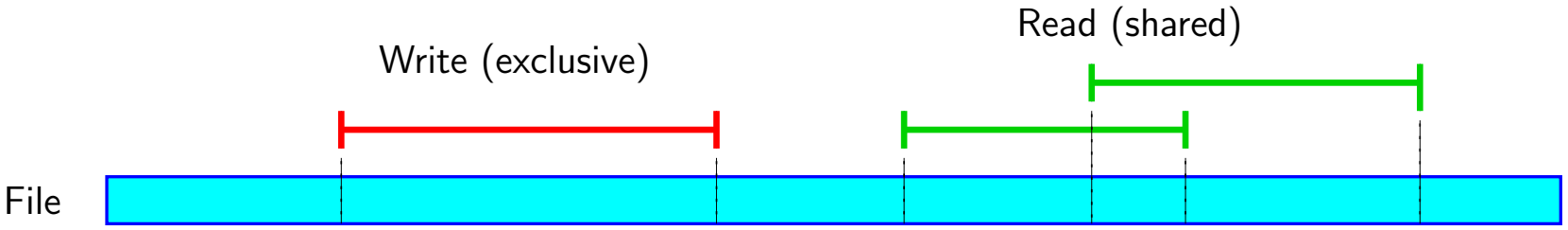
- Livelocks (often occurs when attempting to break a deadlock)
- Aim for fair scheduling: bounded waiting time
- Stronger form of fairness: avoid priority inversion in process scheduling

8. Concurrency and Mutual Exclusion

Alternative: I/O Synchronization With Locks

Purpose

- Serialize processes accessing the same region(s) in a file
- When at least *one process is writing*
- Two kinds of locks: *read* (a.k.a. *shared* and *write* (a.k.a. *exclusive*)
- Two independent APIs supported by Linux
  - POSIX with `fcntl()`
  - BSD with `flock()`



I/O System Call: `fcntl()`

Manipulate a File Descriptor

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, struct flock *lock);
```

Main Commands

`F_DUPFD`: implements `dup()`

`F_GETLK`/`F_SETLK`/`F_SETLKW`: acquire, test or release *file region* (a.k.a. *record*) lock, as described by third argument `lock`

I/O System Call: fcntl()

Manipulate a File Descriptor

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, struct flock *lock);
```

Return Value

- On success, fcntl() returns a (non-negative) value which depends on the command, e.g.,

  F_DUPFD: the new file descriptor
  F_GETLK/F_SETLK/F_SETLKW: **0**

- Return **−1** on error

I/O System Call: `fcntl()`

Manipulate a File Descriptor

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, struct flock *lock);
```

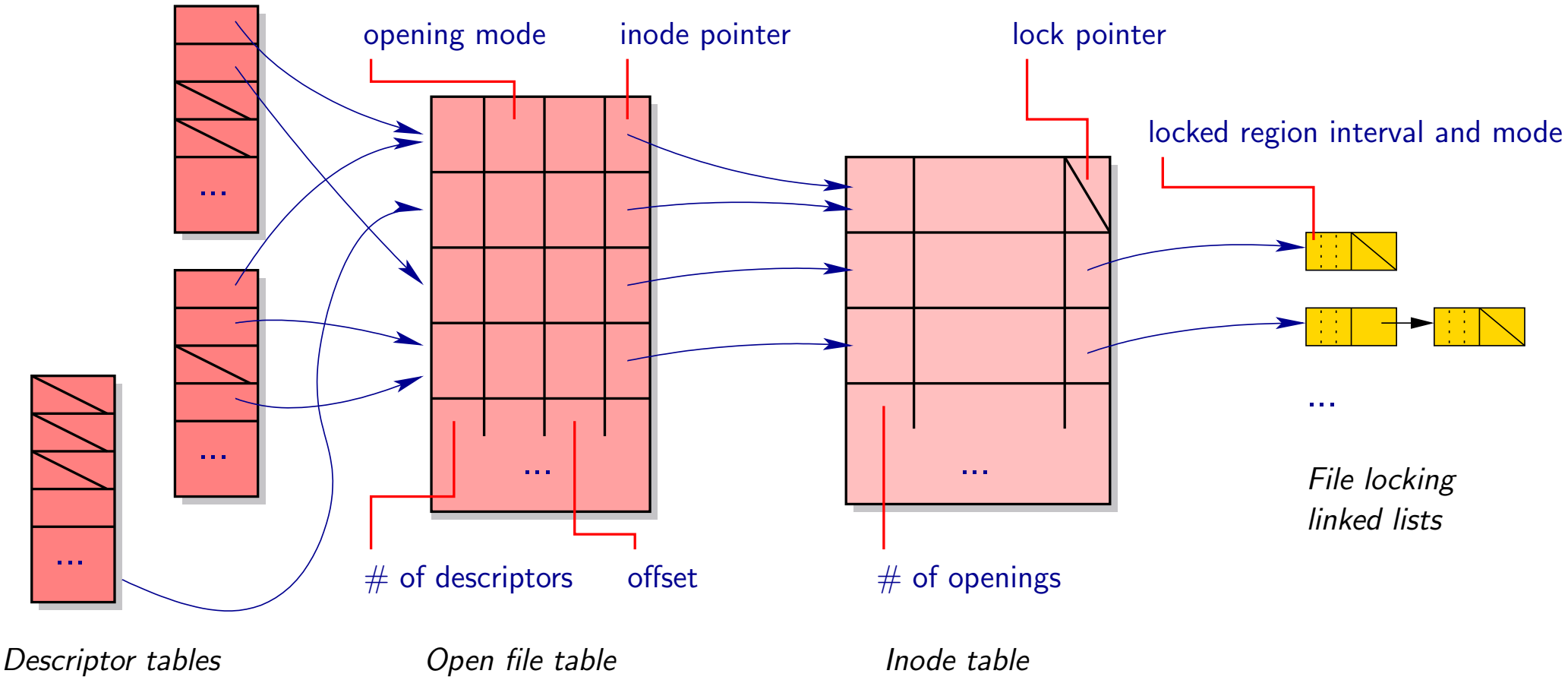
About File Locks

- `fcntl()`-style locks are POSIX locks; *not* inherited upon `fork`
- BSD locks, managed with the `flock()` system call, inherited upon `fork()`
- Both kinds are *advisory*, preserved across `execve()`, fragile to `close()` (releases locks), removed upon termination, and supported by Linux
- `$ man 2 fcntl` and `$ man 2 flock`
- Linux supports SVr3 mandatory `fcntl()`-style locks (mount with `-o mand`)
  - Disabled by default: very deadlock-prone (especially on NFS)
  - Linux prefers *leases* (adds signaling and timeout)

More About File Locks

Implementation Issues

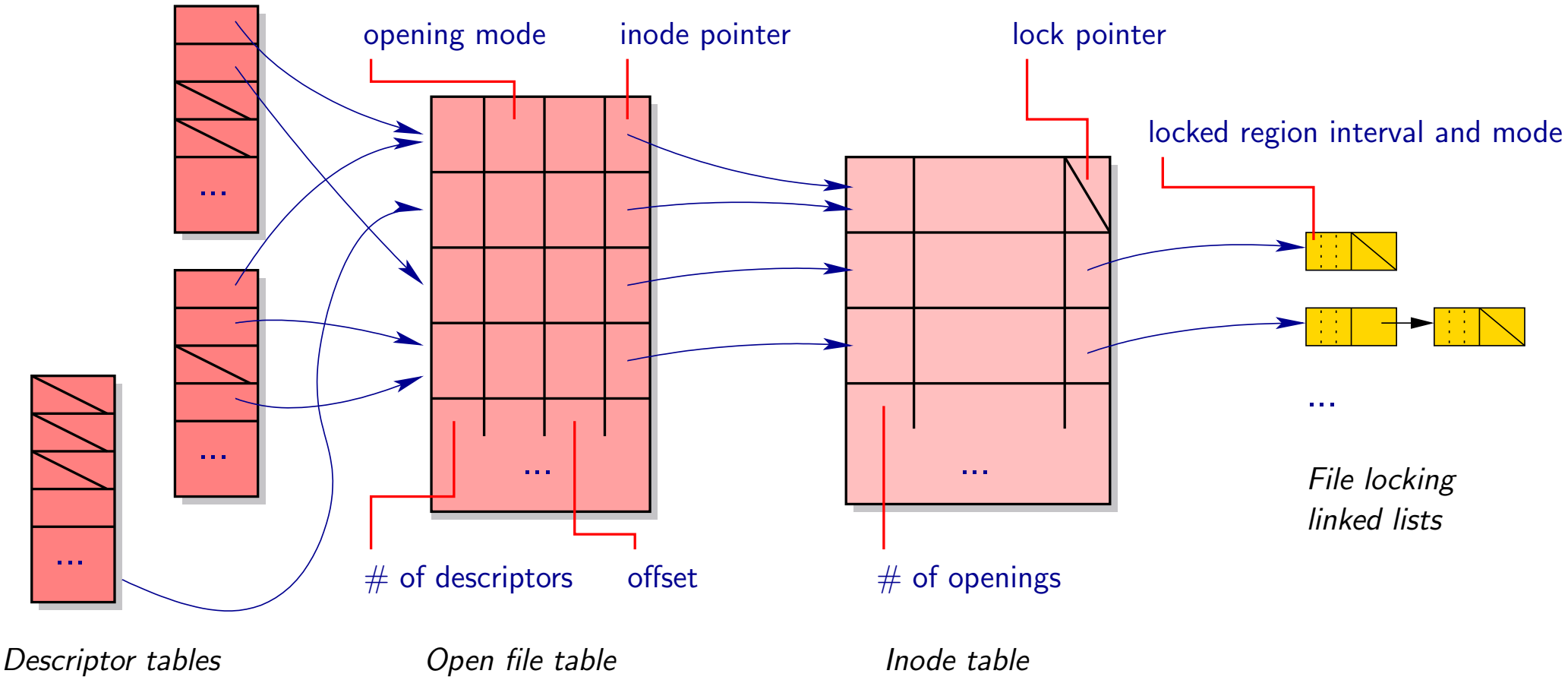
- Locks are associated with *open file entries*:
  → Lost when closing all descriptors related to a file



More About File Locks

Implementation Issues

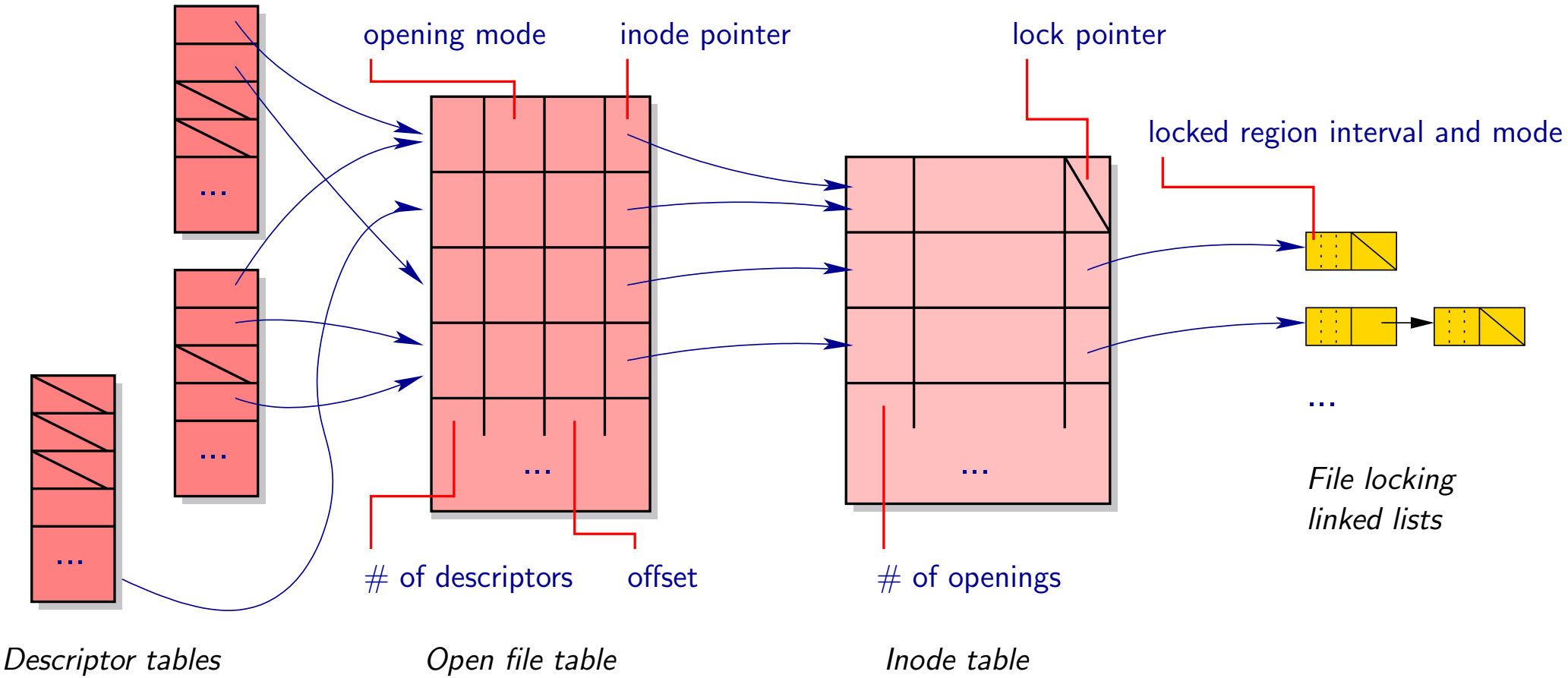
- Incompatible with C library I/O (advisory locking and buffering issues)



More About File Locks

Consequences for the System Programmer

- File locks may be of some use for *cooperating* processes only
- Then, why not use *semaphores*?



8. Concurrency and Mutual Exclusion

System V IPC

Old IPC Interface

- Shared motivation with POSIX IPC
  - Shared memory segments, message queues and semaphore sets
  - Well-defined semantics, widely used, but widely criticized API
  - `$ man 7 svipc`
- But poorly integrated into the file system
  - Uses (hash) *keys* computed from unrelated files
    `$ man 3 ftok`
  - Conflicting and non-standard naming
  - Ad-hoc access modes and ownership rules
- Eventually deprecated by POSIX IPC in 2001