Common Concurrency Problems

- concurrency is a tricky thing to get right
  - bugs may occur only in very specific circumstances and be difficult to reproduce
    - must reason carefully about the code rather than relying on test suites
- common types of bugs
  - non-deadlock bugs
    - atomicity violation
    - ordering violation
  - deadlock bugs
- understanding common error patterns and how to fix them helps prevent bugs in the first place

Atomicity-Violation Bugs

```
Thread 1::
if (thd->proc_info) {
  fputs(thd->proc_info, ...);
}

Thread 2::
thd->proc_info = NULL;
```

if thread 2 sets `thd→proc_info` to `NULL` after thread 1 has checked that it is not `NULL`, `fputs` crashes attempting to dereference a `NULL` pointer

- *atomicity violation*

> "The desired serializability among multiple memory accesses is violated (i.e. a code region is intended to be atomic, but the atomicity is not enforced during execution)."

  - a value gets changed between checking a condition and acting on it, rendering the condition no longer true when the action occurs
- fix
  - add a lock around any access to ensure mutual exclusion

Atomicity-Violation Bugs

```
Thread 1::
if (thd->proc_info) {
  fputs(thd->proc_info, ...);
}

Thread 2::
thd->proc_info = NULL;
```

```
pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;

Thread 1::
pthread_mutex_lock(&proc_info_lock);
if (thd->proc_info) {
  fputs(thd->proc_info, ...);
}
pthread_mutex_unlock(&proc_info_lock);

Thread 2::
pthread_mutex_lock(&proc_info_lock);
thd->proc_info = NULL;
pthread_mutex_unlock(&proc_info_lock);
```


Order-Violation Bugs

```
Thread 1::
void init() {
   mThread = PR_CreateThread(mMain, ...);
}

Thread 2::
void mMain(...) {
    mState = mThread->State;
}
```

thread 2 assumes that `mThread` has already been initialized by thread 1
  – if not, the good outcome is a crash due to dereferencing a `NULL` pointer
  – the bad outcome is that there's a value at `mThread→State`...

- *order violation*

> "The desired order between two (groups of) memory accesses is flipped (i.e., A should always be executed before B, but the order is not enforced during execution)"

  - things happen out of order
- fix
  - use condition variables to impose proper ordering

Order-Violation Bugs

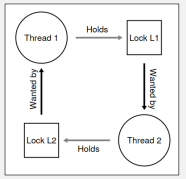
```
Thread 1::
void init() {
   mThread = PR_CreateThread
}

Thread 2::
void mMain(...) {
    mState = mThread->State;
}
```

```
pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  mtCond = PTHREAD_COND_INITIALIZER;
int mtInit             = 0;

Thread 1::
void init() {
   ...
   mThread = PR_CreateThread(mMain, ...);

   // signal that the thread has been created...
   pthread_mutex_lock(&mtLock);
   mtInit = 1;
   pthread_cond_signal(&mtCond);
   pthread_mutex_unlock(&mtLock);
   ...
}

Thread 2::
void mMain(...) {
    ...
    // wait for the thread to be initialized...
    pthread_mutex_lock(&mtLock);
    while (mtInit == 0)
        pthread_cond_wait(&mtCond, &mtLock);
    pthread_mutex_unlock(&mtLock);

    mState = mThread->State;
    ...
}
```


Deadlock Bugs

- *deadlock bugs* arise when there is a cycle of dependency
  - threads block while holding locks they won't release until they acquire a lock held by another

```
Thread 1:                    Thread 2:
pthread_mutex_lock(L1);      pthread_mutex_lock(L2);
pthread_mutex_lock(L2);      pthread_mutex_lock(L1);
```



- deadlock bugs are difficult to detect in large code bases
  - complex dependencies between components each needing various locks
  - encapsulation hides implementation details
    - specifics of locks are implementation details

Conditions for Deadlock

- deadlock requires four conditions

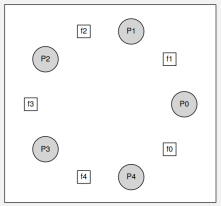
> - **Mutual exclusion:** Threads claim exclusive control of resources that they require (e.g., a thread grabs a lock).
> - **Hold-and-wait:** Threads hold resources allocated to them (e.g., locks that they have already acquired) while waiting for additional resources (e.g., locks that they wish to acquire).
> - **No preemption:** Resources (e.g., locks) cannot be forcibly removed from threads that are holding them.
> - **Circular wait:** There exists a circular chain of threads such that each thread holds one or more resources (e.g., locks) that are being requested by the next thread in the chain.

- to prevent deadlock, ensure that at least one of the four conditions isn't possible

Deadlock Prevention

> - **Circular wait:** There exists a circular chain of threads such that each thread holds one or more resources (e.g., locks) that are being requested by the next thread in the chain.

Dining Philosophers

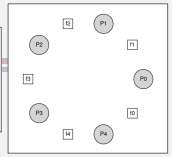



```
while (1) {
    think();
    get_forks(p);
    eat();
    put_forks(p);
}
```

- philosophers sit around a circular table, eating and thinking
  - when they think, they don't need any forks
  - when they eat, they need two forks
- want to maximize concurrency – philosophers should be eating if they want to and forks are available – while avoiding deadlock and starvation

Dining Philosophers

```
while (1) {
    think();
    get_forks(p);
    eat();
    put_forks(p);
}
```



have a mutex lock for each fork – only one philosopher can use the fork at a time

```
void get_forks(int p) {
    sem_wait(&forks[left(p)]);
    sem_wait(&forks[right(p)]);
}

void put_forks(int p) {
    sem_post(&forks[left(p)]);
    sem_post(&forks[right(p)]);
}
```

when a philosopher wants to eat, grab the fork on the left and then the fork on the right

when a philosopher is done eating, put down the fork on the left and then the fork on the right

does this work?

if everyone grabs their left fork before someone reaches for their right fork, everyone is stuck → deadlock

the problem is that threads block holding locks that can only be released by another thread after acquiring the held lock

circular wait: everyone grabs their left fork (fork *i* for philosopher *i*), then is waiting on the right fork (*i*+1)%5

0 → 1 → 2 → 3 → 4 → 0 → …

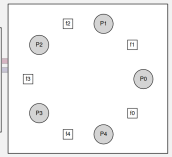
Deadlock Prevention

> • **Circular wait:** There exists a circular chain of threads such that each thread holds one or more resources (e.g., locks) that are being requested by the next thread in the chain.

- fix
  - impose a *total ordering* on lock acquisition
    - i.e. there's a sequence of locks 1..*n* such that lock *k* is always acquired before lock *k*+1 for $1 \le k \le n$
  - when a total order is difficult or impossible, a *partial order* on lock acquisition can be sufficient
  - why does this work? with a total order –
    - a thread t_1 is waiting for lock *j*
    - if the thread t_2 holding lock *j* waits before it releases lock *j*, it can only be for a lock > *j* – which thread t_1 isn't holding because it only holds locks < *j*
    - …
    - the thread t_3 holding lock *n* doesn't need to wait on any locks before releasing lock *n*

Dining Philosophers

```
while (1) {
    think();
    get_forks(p);
    eat();
    put_forks(p);
}
```



have a mutex lock for each fork – only one philosopher can use the fork at a time

```
void get_forks(int p) {
    if (p == 4) {
        sem_wait(&forks[right(p)]);
        sem_wait(&forks[left(p)]);
    } else {
        sem_wait(&forks[left(p)]);
        sem_wait(&forks[right(p)]);
    }
}
```

```
void put_forks(int p) {
    sem_post(&forks[left(p)]);
    sem_post(&forks[right(p)]);
}
```

solution – vary the pattern

for most philosophers, when they want to eat, grab the fork on the left and then the fork on the right

for at least one philosopher, grab the fork on the right and then the fork on the left

when a philosopher is done eating, put down the fork on the left and then the fork on the right

this breaks the circular wait: everyone grabs their first fork (philosophers 0-3 grab left forks 0-3, 4 tries to grab right fork 0, philosophers 0-2 try to grab right forks 1-3, philosopher 3 grabs right fork 4)

4 → 0 → 1 → 2 → 3 but 3 is not waiting on anything and can eat, eventually putting down their forks

Enforcing Lock Orders

- ensuring lock acquisition orders can be tricky

a function taking two or more locks as parameters can't control the order the caller passes them in

use the memory address of the lock to determine the order – guarantees at least a partial order without relying on the caller to get things right

TIP: ENFORCE LOCK ORDERING BY LOCK ADDRESS

In some cases, a function must grab two (or more) locks; thus, we know we must be careful or deadlock could arise. Imagine a function that is called as follows: `do_something(mutex_t *m1, mutex_t *m2)`. If the code always grabs `m1` before `m2` (or always `m2` before `m1`), it could deadlock, because one thread could call `do_something(L1, L2)` while another thread could call `do_something(L2, L1)`.

To avoid this particular issue, the clever programmer can use the *address* of each lock as a way of ordering lock acquisition. By acquiring locks in either high-to-low or low-to-high address order, `do_something()` can guarantee that it always acquires locks in the same order, regardless of which order they are passed in. The code would look something like this:

```
if (m1 > m2) { // grab in high-to-low address order
  pthread_mutex_lock(m1);
  pthread_mutex_lock(m2);
} else {
  pthread_mutex_lock(m2);
  pthread_mutex_lock(m1);
}
// Code assumes that m1 != m2 (not the same lock)
```

By using this simple technique, a programmer can ensure a simple and efficient deadlock-free implementation of multi-lock acquisition.

Deadlock Prevention

- **Hold-and-wait:** Threads hold resources allocated to them (e.g., locks that they have already acquired) while waiting for additional resources (e.g., locks that they wish to acquire).

- solution – acquire all needed locks at once
  - use a mutex lock to ensure lock-acquisition steps can't be interrupted

```
pthread_mutex_lock(prevention);   // begin acquisition
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
...
pthread_mutex_unlock(prevention); // end
```

- problems
  - at odds with encapsulation
    - need to know what locks are required
    - can't bundle lock acquisition with the operation that needs it
  - reduces concurrency
    - acquiring all locks up front means some may be held longer than necessary

Deadlock Prevention

- **No preemption:** Resources (e.g., locks) cannot be forcibly removed from threads that are holding them.

- solution – release held locks and try again instead of blocking
  - `trylock` acquires the lock if possible, but returns an error code instead of blocking otherwise

```
top:
  pthread_mutex_lock(L1);
  if (pthread_mutex_trylock(L2) != 0) {
    pthread_mutex_unlock(L1);
    goto top;
  }
```

- challenges
  - can result in *livelock* – two threads competing with each other keep trying the sequence without success
    - fix: add random delay before retrying
  - encapsulation is still problematic
    - harder to jump back to the beginning of the process
  - becomes more complex for longer sequences of locks
    - have to release all locks held