COP4634: Systems & Networks I

Synchronization

Background on Process Synch.

- Concurrent access to shared data may result in data becoming inconsistency.

- Maintaining data consistency while supporting concurrent processing requires additional OS mechanisms.

Example: Concurrent Data Access

```
int g;
void *tFunc(void *param){
  char *str = (char *)param;
  int loc = 0;
  g += 10;
  loc = g + 5;
  printf("%s: g %d: loc %d\n", str, g, loc);
  pthread_exit(0);
}
int main(int argc, char **argv){
  pthread_t tid1, tid2;
  g = 10;
  pthread_create(&tid1, NULL, tFunc, (void *)"ONE");
  pthread_create(&tid2, NULL, tFunc, (void *)"TWO");
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
  return 0;
}
```

```
int g;
void *tFunc(void *param){
  char *str = (char *)param;
  int loc = 0;
  g += 10;
    LOAD R3, 10
    LOAD R4, g
    ADD R5,R4,R3
    STOR R5, g
  loc = g + 5;
    LOAD R3, 5
    LOAD R4, g
    ADD R5,R4,R3
    STOR R5, loc
  printf("%s: g %d: loc %d\n', str, g, loc);
    LOAD R3, str
    LOAD R4, g
    LOAD R5, loc
  …
  pthread_exit(0);
}
```

g

	ONE	TWO
R3		
R4		
R5		
loc		

Synchronization Concepts

Race Condition:

the final outcome of a result is determined by the order of process execution

Critical Section: a code section in a process that accesses shared data

```
void *tFunc(void *param){
  char *str = (char *)param;
  int loc = 0;
  g += 10;
  loc = g + 5;
  printf("%s: g %d: loc %d\n", str, g, loc);
  pthread_exit(0);
}
```

Solution to Critical Section Problem

- *Mutual Exclusion*:
  - only one thread at a time can be in their critical section
- *Progress*:
  - if no thread is in their critical section, other processes should be allowed to enter
- *Bounded waiting*:
  - there must be a bound on the time a thread waits to enter its critical section

Critical Sections and Threads

- Threads must proceed through an entry and exit sections.
- Threads must not die or quit inside a critical section.
- Threads may be context switched inside a critical section.
  - a context switch is different from an exit because

Bad Solution 1

```
int turn;

while(1) {
   …
   while (turn != i);
   CS();
   turn = j;
   …
}
```

- CS – Critical Section
- Mutal exclusion (mutex)
- `i` – pid of currently executing process
- `j` – pid of “other” process

Bad Solution 1

```
int turn;

while(1) {
   …
   while (turn != i);
   CS();
   turn = j;
   …
}
```

turn	P0	P1

Bad Solution 2

```
boolean wantCS[2] = {F, F};

while(1) {
   …
   wantCS[i] = T;
   while (wantCS[j]);
   CS();
   wantCS[i] = F;
   ….
}
```

wantCS	0	1

General Format

```
while (1) {
   …
   ENTRY_CODE();
   CS();
   EXIT_CODE();
   …
}
```

Controls access to CS

Insures MUTEX

Allows next P to enter CS

Assume no crashes for now

Atomic Actions

- Once started, must complete
- Can’t be interrupted in the middle
- Completes or doesn’t start
- Statements (generally) NOT atomic
- Instructions are atomic

Synchronization Solutions

- Peterson's and Bakery Algorithm
  - work but are difficult to generalize to all sorts of computing problems requiring synchronization

- Synchronization Hardware
  - atomic instructions that test and set a boolean value at the same time

Problems

- User-level implementation:
  - bugs can be easily introduced
  - synchronization code can be (un)intentionally skipped
  - requires busy wait loop

- What we want:
  - kernel implementation
  - user can use
  - user cannot alter
  - efficient implementation

LOCK

- Kernel implementation of MUTEX
- Has two functions
  - Acquire() requests lock, only returns when lock is given to process
  - Release() gives lock back to kernel to reallocate to another process
- Presented as “Class/Object” for clarity
- Really implemented in C

```
class LOCK {
  int status = FREE;
}
LOCK::Acquire() {
  Disable_Interrupts();
  while (status == BUSY) {
    Enable_Interrupts();
    Disable_Interrupts();
  }
  status = BUSY;
  Enable_Interrupts();
}
```

```
class LOCK {
  int status = FREE;
}
LOCK::Acquire() {
  …
}
LOCK::Release() {
  Disable_Interrupts();
  status = FREE;
  Enable_Interrupts();
}
```

Why not Perfect Solution?

- Busy wait loop still present
  - Eats CPU time while waiting
- High priority thread waiting?
  - Holder (low priority) never runs
- Waiting thread could be “infinitely unlucky”
  - How could this be?

```
class LOCK {
  int status = FREE;
}
LOCK::Acquire() {

  Disable_Interrupts();

  while (status == BUSY) {
    Enable_Interrupts();

    Disable_Interrupts();
  }
  status = BUSY;
  Enable_Interrupts();
}
```

P0 interrupted here

P1 Release(), Acquire(), `status` is `FREE`

Corrections

- Add “Waiting List for Lock” (queue)
- Need `enqueue()` and `dequeue()` for list
- Must be able to put “self” on list

Idea:

If lock is BUSY,

put self on waiting list

go to sleep

When releasing lock,

wake-up a sleeping thread

```
class LOCK{
  int status = FREE;
  queue waiting;
}
LOCK::Acquire() {
  Disable_Interrupts();
  while (status != FREE) {
    waiting.enqueue(self);
    Enable_Interrupts();
    block();
    Disable_Interrupts();
  }
  status = BUSY;
  Enable_Interrupts();
}
```

```
class LOCK{
  int status = FREE;
  queue waiting;
}
LOCK::Release() {
  Disable_Interrupts();
  if (! waiting.empty()) {
    proc = waiting.dequeue();
    readyList.insert(proc);
  }
  status = FREE;
  Enable_Interrupts();
}
```

Notes

- `block()` is internal kernel call (kernel mode)
- Interrupts must be enabled before calling `block()`
- Similar to context switch
  - Calling process taken off CPU
  - Calling process stored in PCB
  - Calling process **NOT** put on ready list
- Better be on some other list before calling!

LOCKs in Pthreads

```
#include <pthread.h>
pthread_mutex_t lock;
pthread_mutexattr_t mattr;
pthread_mutexattr_init( &mattr );
pthread_mutex_init( &lock, &mattr );
pthread_mutex_lock( &lock );
pthread_mutex_unlock( &lock );
pthread_mutex_destroy( &lock );
```

LOCKs Sample Code

```
pthread_mutex_t lockCounter;

int main ( void ) {
        …
        pthread_mutex_init( &lockCounter, NULL );

        // launch threads
        …
        // join threads
        …
        pthread_mutex_destroy( & lockCounter);
        …
        return 0;
}
```

LOCKs Sample Code (cont.)

```
void* thread_func ( void *param ) {
    …
    pthread_mutex_lock(&lockCounter);
    // execute CS()
    …
    pthread_mutex_unlock(&lockCounter);
    …
    pthread_exit(NULL);
}
```

LOCKs in C++

```
#include <mutex>

std::mutex lock;

// lock and unlock a section of code
lock.lock();
CS();
lock.unlock();
```

LOCKs Sample Code

```
std::mutex lockCounter;

int main ( void ) {
     …
     // launch threads
     …
     // join threads
     …
     return 0;
}
```

LOCKs Sample Code (cont.)

```
void thread_func ( ) {
      …
      lockCounter.lock();
      // execute CS()
      lockCounter.unlock();
      …
}
```

Hardware Help

- Read/Modify/Write instruction
  - test-n-set instruction (atomic instruction)

- Functional Definition (really single instruction)

```
int testNset(int &var) {
  int tmp;
  tmp = var;
  var = 1;
  return tmp;
}
```

Example:

1) inputVal = 0;
2) returnVal = testNset(inputVal);

Q1: What is the value of *inputVal* and *returnVal* after the execution of *testNset()*?
A1: inputVal = 1 and returnVal = 0.

Q2: What changes if *testNset()* is called again with the same value for *inputVal*?

Test-N-Set

- Every modern CPU has equivalent
  - exchange
  - compare-and-swap

```
LOCK::Acquire() {
  // spins a process as long as
  // status is true
  while (testNset(status) == 1);
}

LOCK::Release() {
  status = 0;
}
```

“Spinlock”

Limitations

- Locks are good for MUTEX only
- General concept is needed to solve synchronization problems:

→ Semaphores

- Synchronization concept first proposed by Edsger Dijkstra.
- Semaphores are objects maintained by the OS.
- Each semaphore has a value and two atomic operations:
  - wait
  - signal

Other Names & Original Definition

“test”

```
Proberen(){     P()  Wait()  sem_wait()
   while (count == 0);
   count--;
}
```

“increment”

```
Verhogen(){     V()  Signal()  sem_post()
   count++;
}
```

Extended Definition

```
int sem_wait( sem ) {
  Disable_Interrupts();
  sem.count--;
  if (sem.count < 0) {
    waitlist.enqueue(self);
     Enable_Interrupts();
    block();
     Disable_Interrupts();
  }
  Enable_Interrupts();
  return 0;
}
```

Extended Definition

```
int sem_post( sem ) {
  Disable_Interrupts();
  sem.count++;
  if (sem.count <= 0) {
    proc = waitlist.dequeue();
    readyList.insert(proc);
  }
  Enable_Interrupts();
  return 0;
}
```

Monitors for Synchronization

- Monitors are a high-level synchronization constructs.
- Monitors allow safe sharing of abstract data types among concurrent processes.
- Monitors provide synchronized methods that can only be entered by one process or thread at a time.

Monitors for Synchron. (Java)

```
/**
 * Models a bank account with a balance.
 */
public class BankAccount   {

  private double balance;

   /**
    * Constructs a bank account object with 0 balance.
    */
  public BankAccount()      {    balance = 0.0;      }

  /**
   * Deposits a specified amount into the account.
   * @param amount - the amount of money to be deposited
   */
  public synchronized void deposit(double amount)  {
              balance = balance + amount;
   }
}
```

Monitors for Synchronization (cont.)

- Monitors provide condition variables with two operations
  - wait: puts process or thread asleep
  - signal: triggers a waiting process or thread to resume operation <u>exactly at the position </u>where process or thread called wait

Condition Variables Implementation (pseudo code)

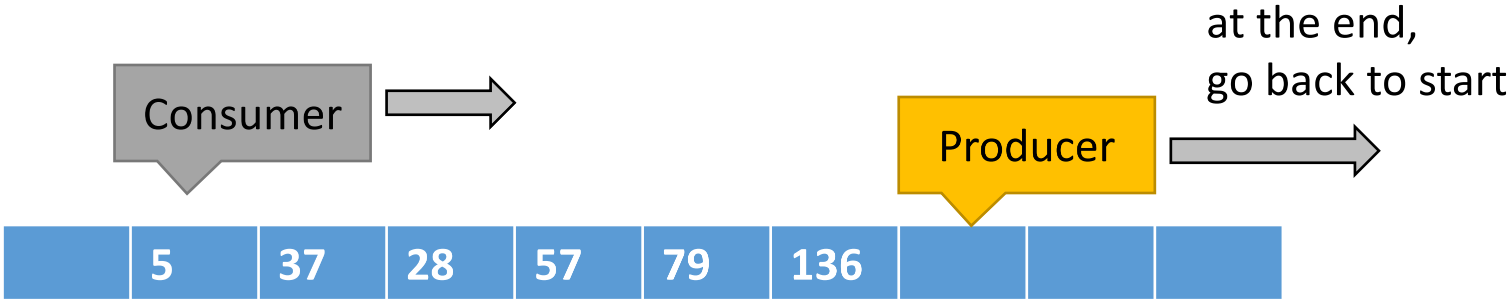
```
Wait( lock ) {
  // put one thread asleep
  lock.Acquire();
}

Signal( lock ) {
  // wake one sleeping thread
  lock.release();
}

Broadcast( lock ) {
  // wake-up all n threads repeat n times
    lock.release();
}
```

Producer/Consumer Synchr. Problem

- Producer makes widgets, Consumer eats widgets
- Storage facility (limited size) for widgets
  - Storage is a circular data structure
- Producer must stop generating data when buffer is full.
- Consumer must stop reading data when no data are available.
- Examples: Networks, Disk I/O, etc.



Producer Code

```
sem_t fullB;
sem_t emptyB;
pthread_mutex_t mutex;

void * producer(void *id) {
  int myID = (int)id;
  while(1) {
    sem_wait( &emptyB );
    pthread_mutex_lock( &mutex );
    addWidgetToBuffer();
    pthread_mutex_unlock( &mutex );
    sem_post( &fullB );
  }
}
```

Q: What is the initial value of *emptyB* and *fullB*?

Consumer Code

```
sem_t fullB;
sem_t emptyB;
pthread_mutex_t mutex;
void * consumer(void *id) {
  int myID = (int)id;
  while(1) {
    sem_wait( &fullB );
    pthread_mutex_lock( &mutex );
    eatWidgetFromBuffer();
    pthread_mutex_unlock( &mutex );
    sem_post( &emptyB );
  }
}
```

A: Initially, the value *emptyB* must be the size of the buffer and *fullB* must be 0.

File Print

`sem_wait(&printer);`
`sem_wait(&file);`
`printFile();`
`sem_post(&file);`
`sem_post(&printer);`
	`sem_wait(&file);`
	`sem_wait(&printer);`
	`printFile();`
	`sem_post(&printer);`
	`sem_post(&file);`

- Deadlock is possible
- Be careful with order

Sleeping Barber

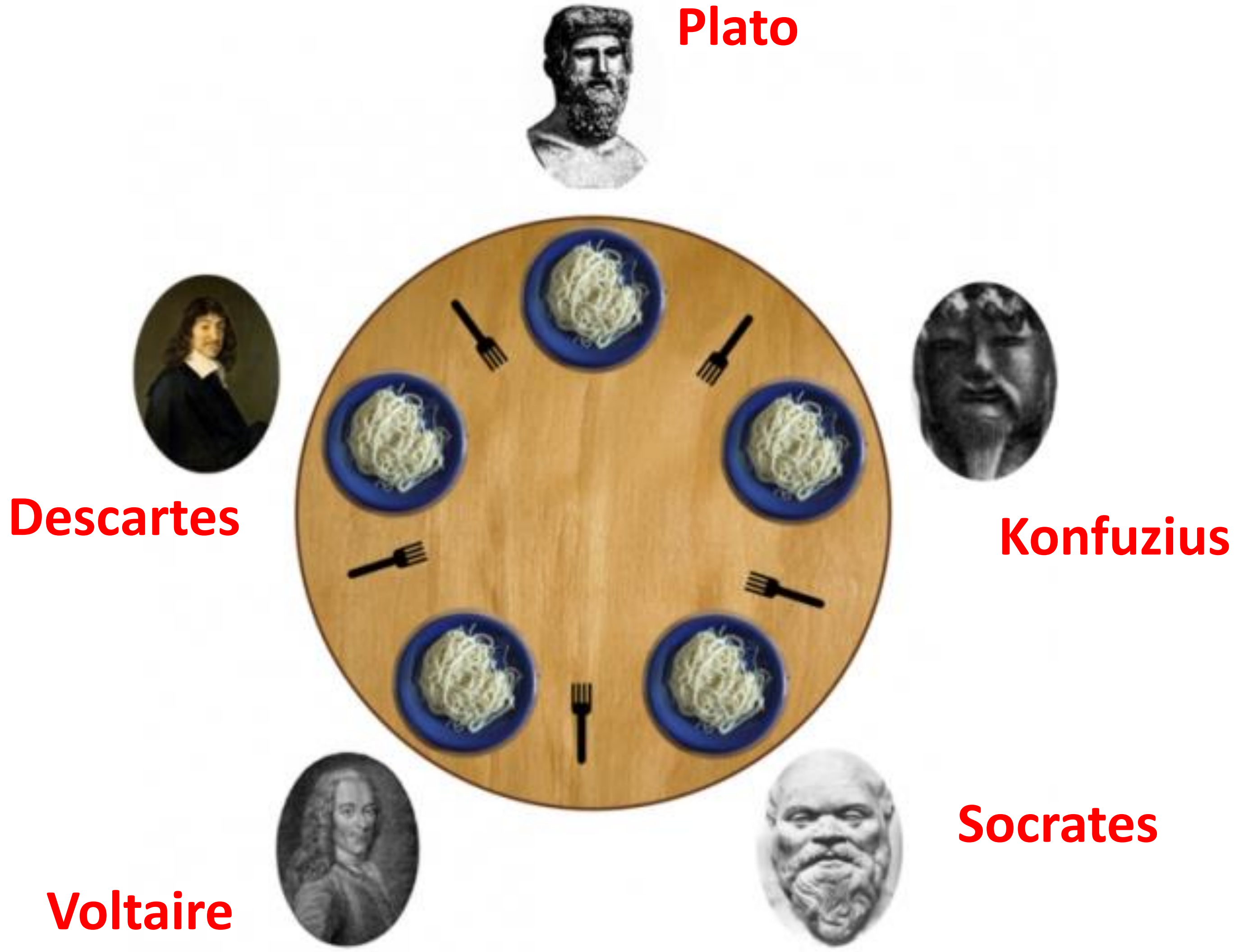
- Small town
- One barber shop w/ one barber
- Barber always at work
- Sleeps when possible
- 1st customer in shop wakes barber
- Other customers wait
- All customers done, barber goes back to sleep

Sleeping Barber

```
Barber() {
 while (1) {
  barber.Wait();
  mutex.Acquire();
  numWaiting--;
  mutex.Release();
  done.Signal();
 }
}
```

```
Customer() {
 mutex.Acquire();
 if (numWaiting < CHAIRS){
  numWaiting++;
  mutex.Release();
  barber.Signal();
  done.Wait();
 } else
   mutex.Release();
 leave();
}
```

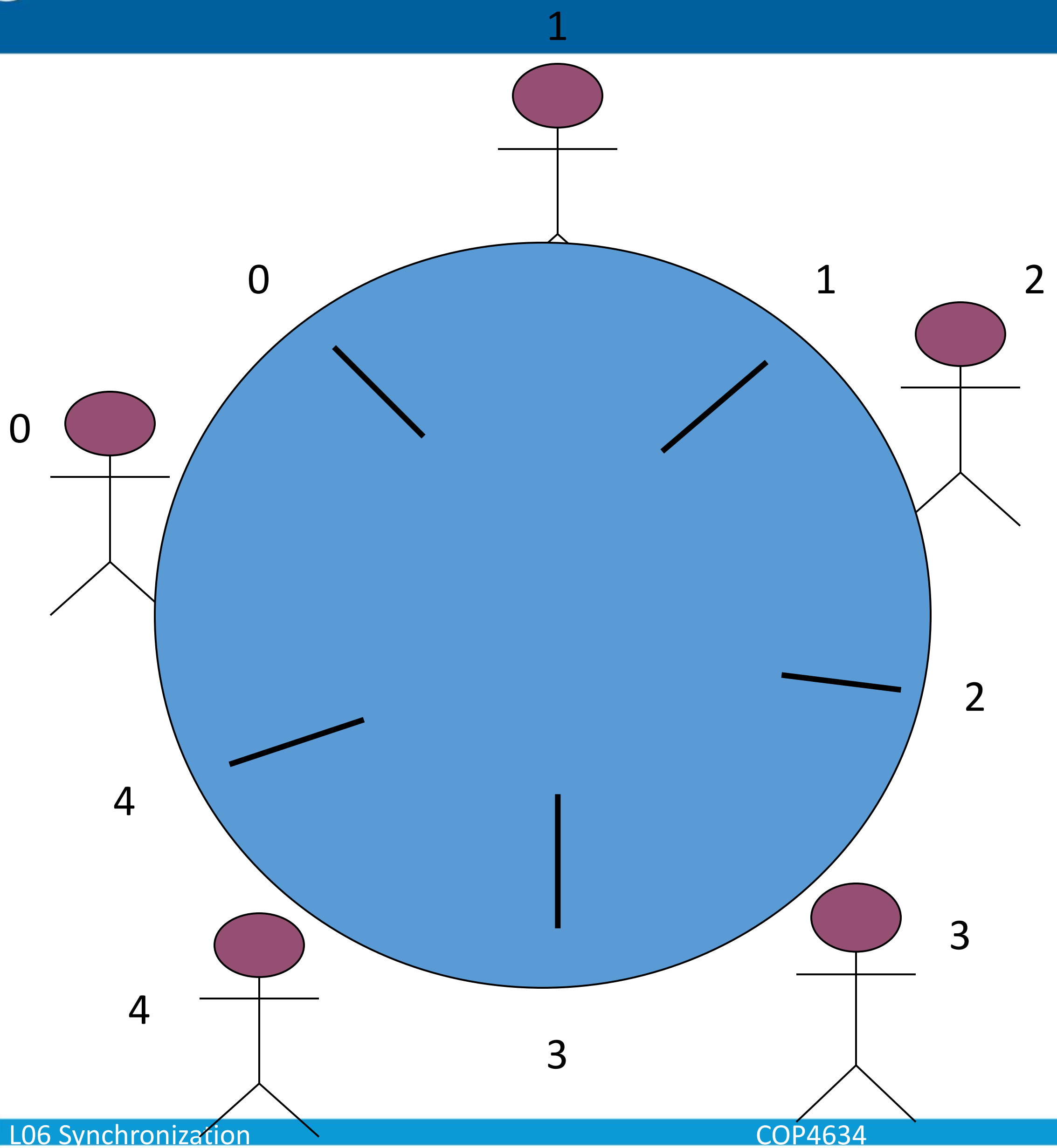
Dining Philosophers (DP)



DP Problem Statement

- N philosophers at round table
- Bowl of rice in middle of table
- Fork between each pair of philosophers
- Need 2 forks to eat
- think, get hungry, grab forks, eat, …
- Avoid deadlock
- Avoid starvation
- Let as many eat as possible

Solution Outline

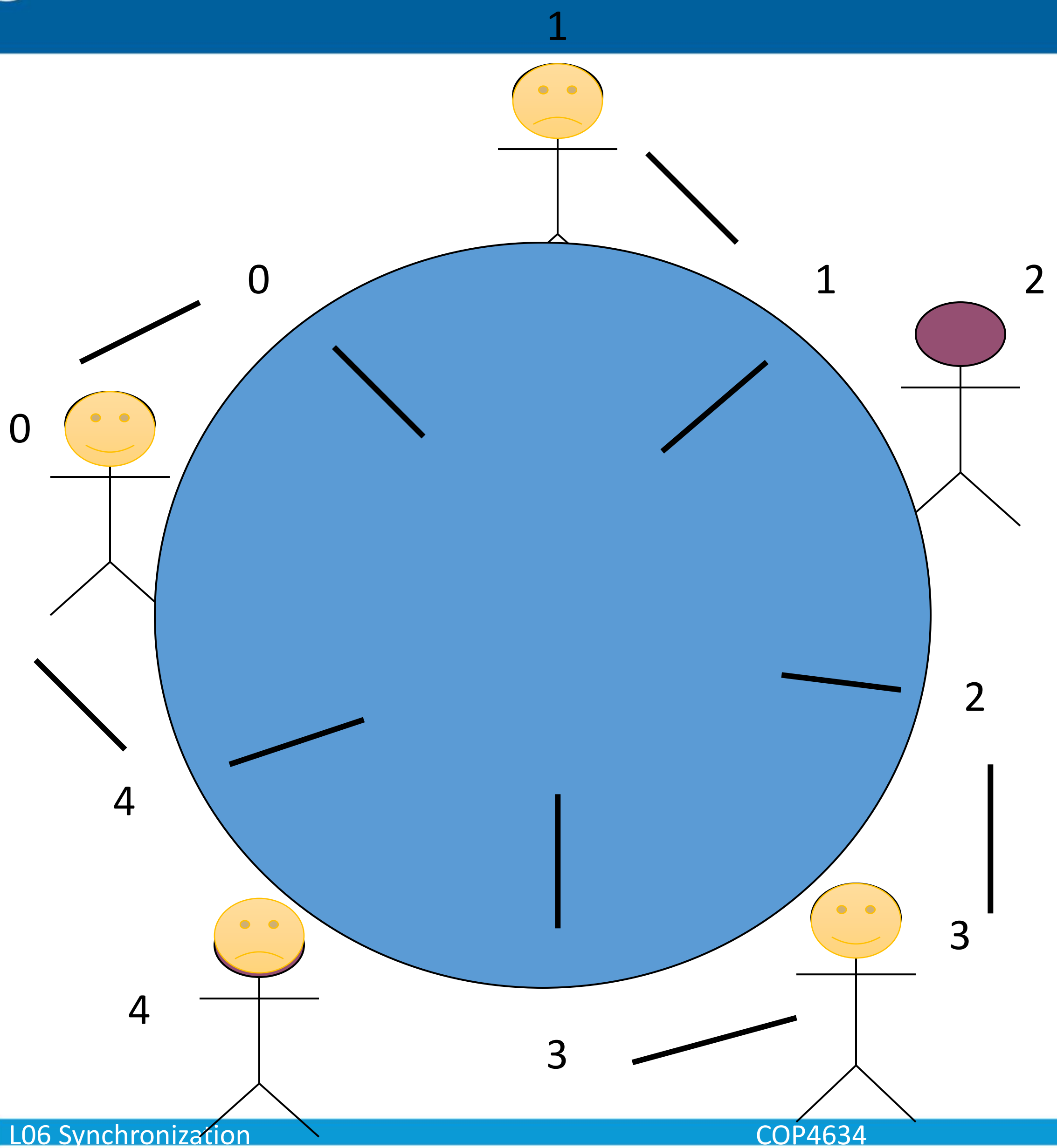


- Philosophers are identified by number
- Forks are identified by number
- Pi can grab F_i and F_{i-1} (mod N)
- Forks are non-preemptive resources.

How about this?

```
void *philosopherThread(void *id){
  int myID = (int)id;
  while(1) {
    think();
    grab(myID - 1);
    grab(myID);
    eat();
    replace(myID - 1);
    replace(myID);
  }
}
```

Solution? Outline

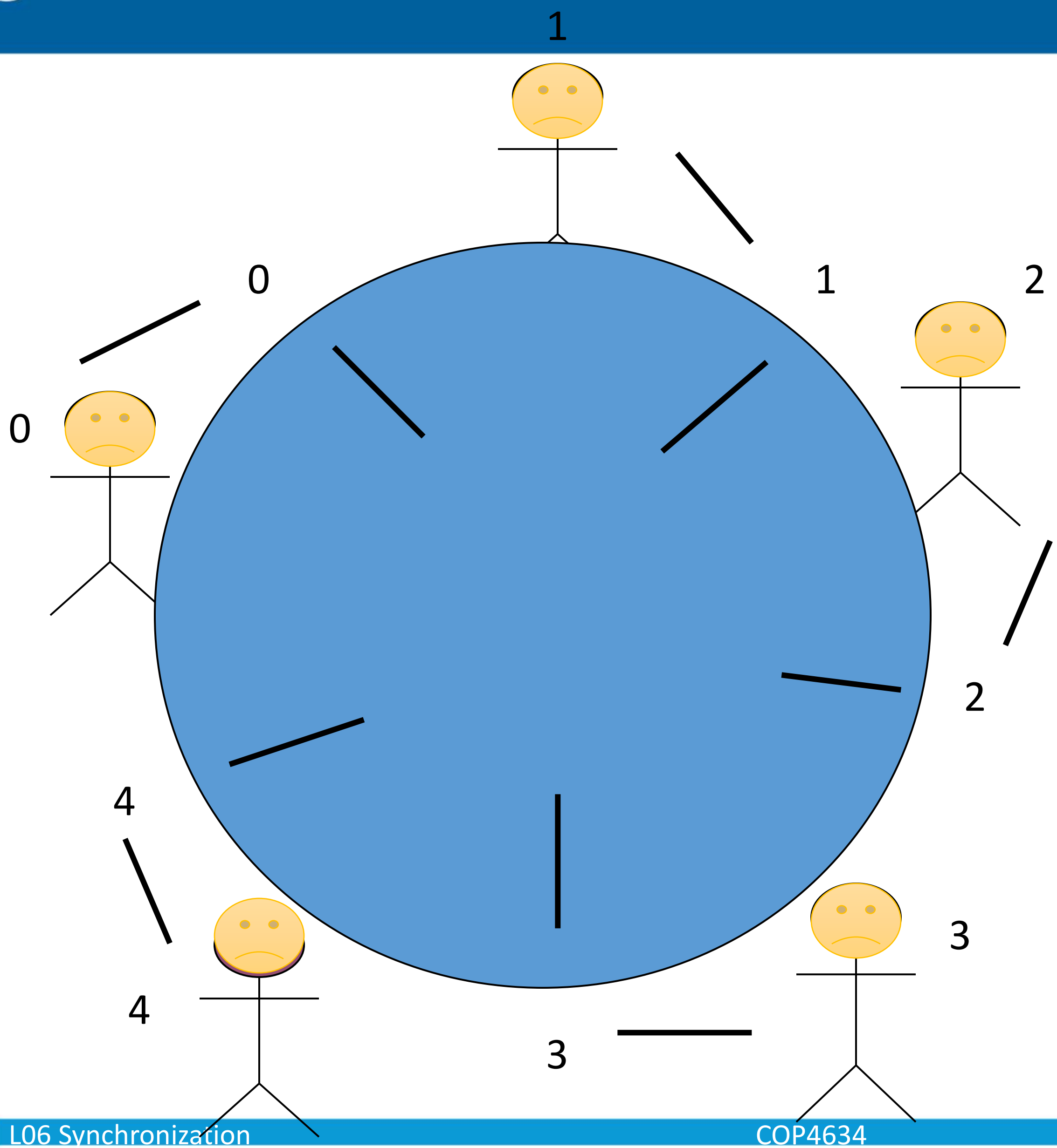


- P_0: $G(F_0)$, $G(F_4)$
- P_1: $G(F_1)$, $G(F_0)$
- P_3: $G(F_3)$, $G(F_2)$
- P_4: $G(F_4)$

What if…

- One philosopher’s stomach growls
- All philosophers realize they are hungry
- All philosophers grab left fork
- All philosophers sleep waiting for right fork
- Deadlock!
- How can we fix the problem?

Solution? Outline

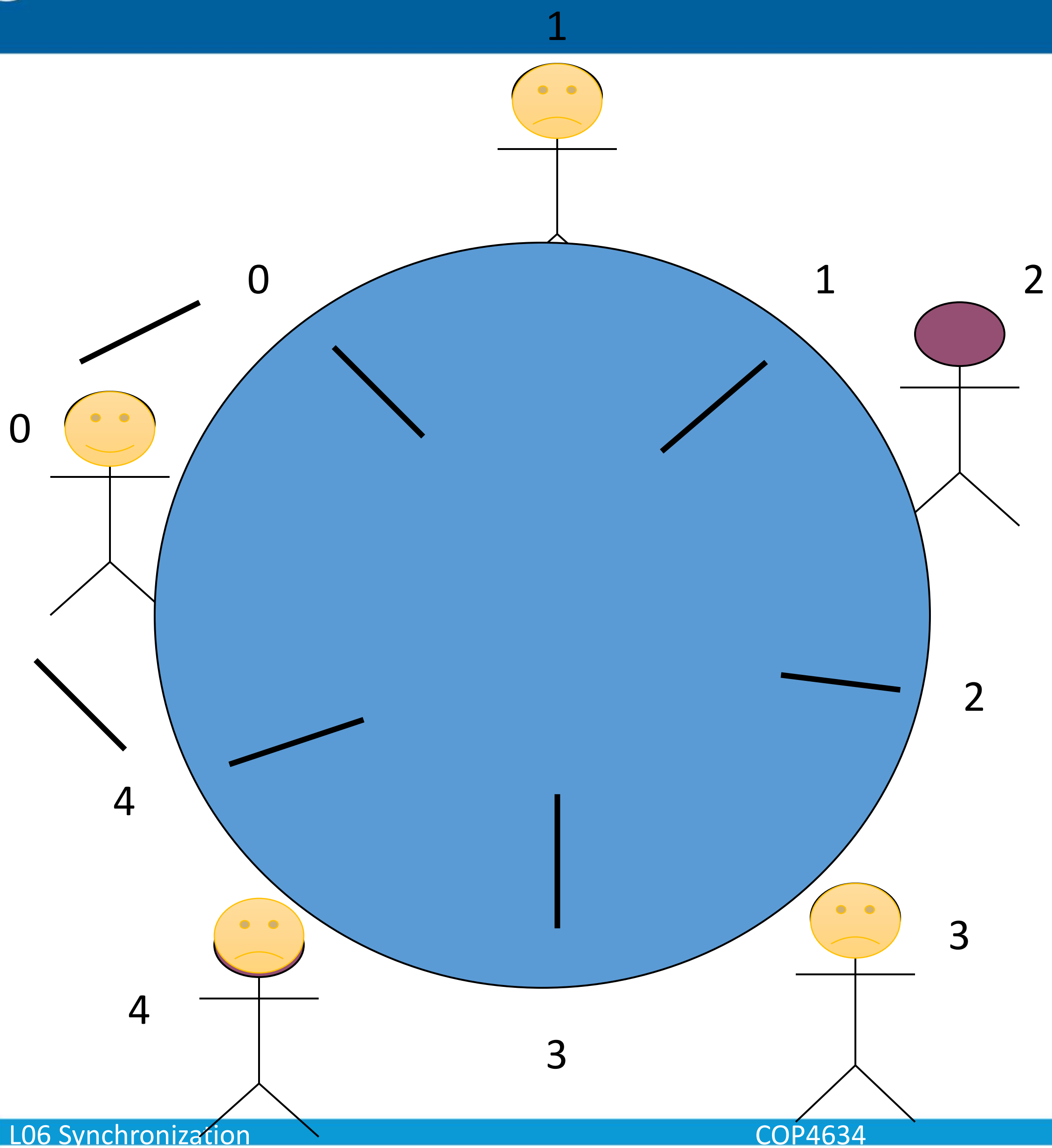


- P_0: $G(F_0)$, $G(F_4)$
- P_1: $G(F_1)$, $G(F_0)$
- P_3: $G(F_3)$, $G(F_2)$
- P_4: $G(F_4)$, $G(F_3)$
- P_2: $G(F_2)$, $G(F_1)$

Another Try

```
void *philosopherThread(void *id){
  int myID = (int)id;
  while(1) {
    think();
    pthread_mutex_lock( &mutex );
    grab(myID - 1);
    grab(myID);
    eat();
    replace(myID - 1);
    replace(myID);
    pthread_mutex_unlock( &mutex );
  }
}
```

Solution? Outline

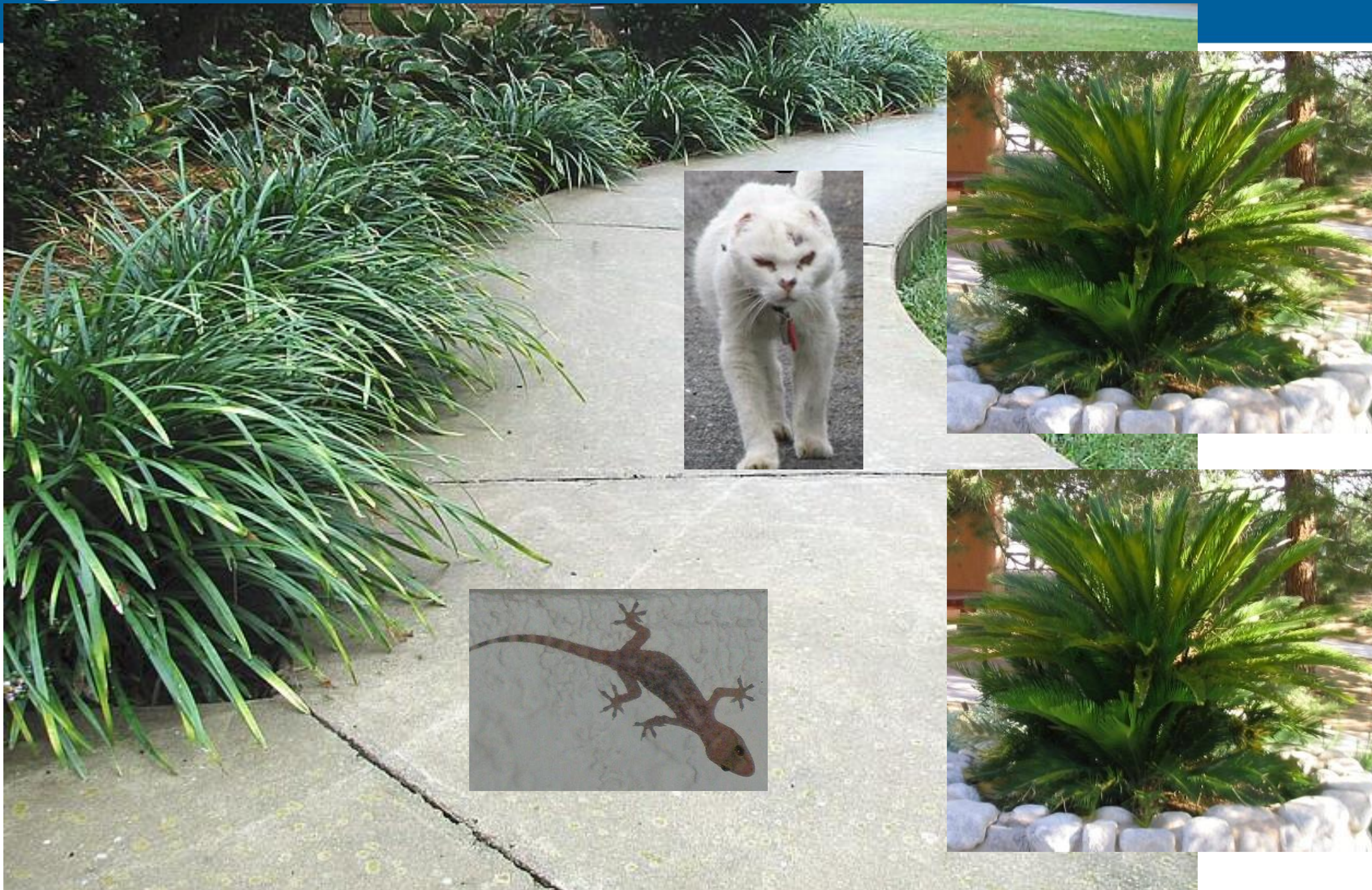


- P_0: $G(F_0)$, $G(F_4)$
- P_1: $G(F_1)$
- P_3: $G(F_3)$
- P_4: $G(F_4)$

Did it Work?

- Do we avoid deadlock?
- How many can eat at a time?
- Do you think this is a good solution?
- How can we improve it?
  - Analyze the problem
  - What is *really* causing deadlock
  - How can we avoid it?
  - Can we make our solution a *little* better?

Next Programming Assignment



Shared Queue

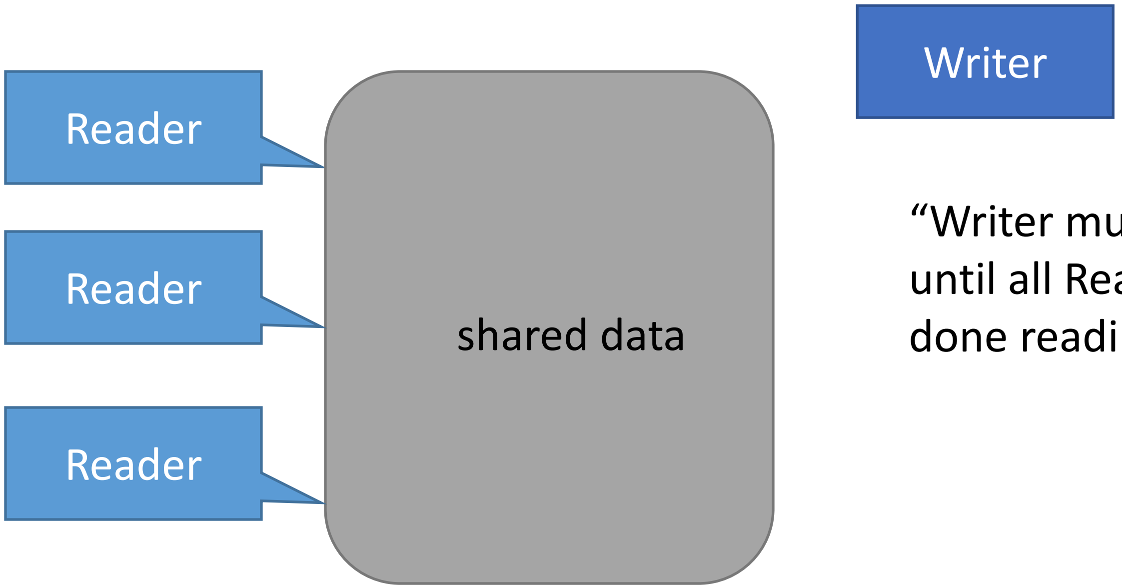
```
LOCK lock = FREE;
AddToQueue( item ) {
  lock.Acquire();
  list.append(item);
  lock.Release();
}
RemoveFromQueue() {
  lock.Acquire();
  if(! list.isEmpty() )
       item = list.remove();
  lock.Release();
  return item;
}
```

Shared Queue

```
RemoveFromQueue() {
  lock.Acquire();
  if( list.isEmpty() )
      lock.Release();
      Sleep();
      lock.Acquire();
  item = list.remove();
  lock.Release();
  return item;
}
```

Multiple Reader, Single Writer

- Multiple Reader processes may access simultaneously shared data, provided Writer process has no access to it.
- Writer process may not access shared data unless no Reader process accesses shared data.



“Writer must wait until all Readers are done reading data.”

Example – DB Access (pseudo code)

```
monitor DB {
  int activeReaders = 0;
  int activeWriters = 0;
  int waitingReaders = 0;
  int waitingWriters = 0;
  Condition OKtoRead;
  Condition OKtoWrite;
  Lock lock = FREE;
```

Example – DB Access (pseudo code)

```
ReadStart() {
    lock.Acquire();
    while((activeWriters + waitingWriters) > 0) {
      waitingReaders++;
      OKtoRead.Wait(lock);
      waitingReaders--;
    }
    activeReaders++;
    lock.Release();
}
```

Example – DB Access (pseudo code)

```
ReadStop() {
    lock.Acquire();
    activeReaders--;
    if ((activeReaders == 0) && (waitingWriters > 0))
      OKtoWrite.Signal(lock);
    lock.Release();
}
```

Example – DB Access (pseudo code)

```
WriteStart() {
    lock.Acquire();
    while((activeWriters + activeReaders) > 0) {
      waitingWriters++;
      OKtoWrite.Wait(lock);
      waitingWriters--;
    }
    activeWriters++;
    lock.Release();
}
```

Example – DB Access (pseudo code)

```
WriteStop() {
    lock.Acquire();
    activeWriters--;
    if ( waitingWriters > 0 )
      OKtoWrite.Signal(lock);
    else if ( waitingReaders > 0 )
      OKtoRead.Broadcast(lock);
    lock.Release();
}
```

Usage

```
Reader() {
  while(1) {
    ReadStart();
    ReadDatabase();
    ReadEnd();
  }
}
```

```
Writer() {
  while(1) {
    WriteStart();
    WriteDatabase();
    WriteEnd();
  }
}
```

Comparison

Semaphore

- Has a value
- Implies a history
- `wait()` may sleep or may proceed depending on value
- `signal()` will increment a value and wake a sleeper
- `signal()` before `wait()` will effect result of wait

Condition Variable

- Doesn't have a value
- Implies no memory
- `wait()` always sleeps
- `signal()` will wake a sleeper
- `signal()` before `wait()` has no effect

Summary

- Threads/processes need to be synchronized to
  - prevent race conditions
  - establish an order of execution
- Monitors combined with condition variables and semaphores can be used as synchronization instruments.
- Many problems require thread and process synchronization:
  - read/writer problem
  - bounded buffer
  - mutual exclusion