Interprocess Communication

Yücel Saygın

These slides are based on your text book and on the slides prepared by Andrew S. Tanenbaum
Inter-process Communication

- **Race Conditions**: two or more processes are reading and writing on shared data and the final result depends on who runs precisely when
- **Mutual exclusion**: making sure that if one process is accessing a shared memory, the other will be excluded from doing the same thing
- **Critical region**: the part of the program where shared variables are accessed
Interprocess Communication
Race Conditions

Two processes want to access shared memory at same time
Critical Regions (1)

Four conditions to provide mutual exclusion

1. No two processes simultaneously in critical region
2. No assumptions made about speeds or numbers of CPUs
3. No process running outside its critical region may block another process
4. No process must wait forever to enter its critical region
Critical Regions (2)

Mutual exclusion using critical regions
Mutual Exclusion via Disabling Interrupts

- Process disables all interrupts before entering its critical region
- Enables all interrupts just before leaving its critical region
- CPU is switched from one process to another only via clock or other interrupts
- So disabling interrupts guarantees that there will be no process switch
- Disadvantage:
  - Should give the power to control interrupts to user (what if a user turns off the interrupts and never turns them on again?)
  - Does not work in case of multiple CPUs. Only the CPU that executes the disable instruction is effected.
- Not suitable approach for the general case but can be used by the Kernel when needed
Mutual Exclusion with Busy Waiting:
(1) Lock Variables

- Testing a variable until some value appears is called **busy waiting**.
- A lock that uses busy waiting is called a **spin lock**.
- A single shared lock variable (lock) initially 0.
- **When a process wants to enter its critical region**
  - Check if the lock is 0
  - If lock = 0 then set it to 1 and enter the critical region, set it back to 0 before leaving the critical region.
  - If lock = 1 then wait until it is set to 0.
- Does the above scheme work?
Mutual Exclusion with Busy Waiting:
(2) Strict Alternation

- A variable (turn) is used to run two processes in alternation (i.e., process 1 runs, then process 2 runs, then process 1 runs again, and so on).
Mutual Exclusion with Busy Waiting: Strict Alternation

Initially turn = 0

while (TRUE) {
    while (turn != 0)       /* loop */;
    critical_region();
    turn = 1;
    noncritical_region();
}

(a) Process 0.

while (TRUE) {
    while (turn != 1)       /* loop */;
    critical_region();
    turn = 0;
    noncritical_region();
}

(b) Process 1.

- It wastes CPU time, so we should avoid busy waiting as much as we can
- Can be used only when the waiting period is expected to be short
- However there is a problem in the above approach!
while (TRUE) {
    while (turn != 0) /* loop */ ;
    critical_region();
    turn = 1;
    noncritical_region();
}

while (TRUE) {
    while (turn != 1) /* loop */ ;
    critical_region();
    turn = 0;
    noncritical_region();
}
Mutual Exclusion with Busy Waiting: Strict Alternation

while (TRUE) {
    while (turn != 0) /* loop */;
critical_region();
    turn = 1;
    noncritical_region();
}

(a) Process 0.

while (TRUE) {
    while (turn != 1) /* loop */;
critical_region();
    turn = 0;
    noncritical_region();
}

(b) Process 1.

- A problem exists in the above approach
- Scenario:
  1. Process 0 leaves its critical region and sets turn to 1, enters its non-critical region
  2. Process 1 enters its critical region, sets turn to 0 and leaves its critical region
  3. Process 1 enters its non-critical region, quickly finishes its job and goes back to the while loop

Since turn is 0, process 1 has to wait for process 0 to finish its non-critical region so that it can enter its critical region.
This violates the third condition of providing mutual exclusion.
How About this solution?

While (TRUE) {
    turn = 0;
    while (turn ==0);
    critical_region();
    noncritical_region();
}

While (TRUE) {
    turn = 1;
    while (turn ==1);
    critical_region();
    noncritical_region();
}
How About this solution then?

Initially flag1 and flag2 are both 0

```
While (TRUE) {
    flag1 = 1;
    turn = 0;
    while ((turn ==0) && (flag2 == 1));
    critical_region();
    flag1 = 0;
    noncritical_region();
}
```

```
While (TRUE) {
    flag2 = 1;
    turn = 1;
    while ((turn ==1) && (flag1 == 1));
    critical_region();
    flag2 = 0;
    noncritical_region();
}
```

How can you solve this problem for more than two processes?
Mutual Exclusion with Busy Waiting: (3) Peterson’s Solution

• The previous solution solves the problem of one process blocking another process while its outside its critical section (not a good mutual exclusion)

• Peterson’s Solution is a neat solution with busy waiting, that defines the procedures for entering and leaving the critical region.
Mutual Exclusion with Busy Waiting (2)

```c
#define FALSE 0
#define TRUE 1
#define N 2          /* number of processes */

int turn;          /* whose turn is it? */
int interested[N]; /* all values initially 0 (FALSE) */

void enter_region(int process); /* process is 0 or 1 */
{
    int other; /* number of the other process */

    other = 1 - process; /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process; /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process) /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}
```

- Wait until the other process sets the turn: while (turn == process)
- Wait while the another process is interested: while (interested[other] == TRUE)
- Go out of the while loop after the other process has left its critical region
Mutual Exclusion with Busy Waiting

(3) TSL Instruction

**enter**\_**region**:

```
TSL REGISTER, LOCK         | copy lock to register and set lock to 1
CMP REGISTER, #0           | was lock zero?
JNE enter_region           | if it was non zero, lock was set, so loop
RET                         | return to caller; critical region entered
```

**leave**\_**region**:

```
MOVE LOCK, #0               | store a 0 in lock
RET                         | return to caller
```

- TSL instruction is provided by the hardware
- **TSL REGISTER, LOCK** reads the contents of the lock into REGISTER and sets LOCK to 1 in an atomic fashion
- No other processor can reach LOCK during TSL
  - In order to achieve that, CPU executing TSL instruction locks the bus to prevent other processors accessing LOCK
Mutual Exclusion with Busy Waiting

- Causes a problem called “PRIORITY INVERSION”
- Two processes: H (high priority) and L (Low Priority)
- Scheduler must make sure that H is executed whenever it is in ready state
Mutual Exclusion with Busy Waiting

**Scenario:**

– Process H blocks for I/O
– Process L now can execute and enters its critical section
– I/O operation H waits for is now complete
– H is scheduled now but since L is in the critical section, H does busy waiting
– But L is never scheduled while H is running
– And the system is blocked forever.

```c
HIGH
while (TRUE) {
    while (turn != 0) /* loop */;
    critical_region();
    turn = 1;
    noncritical_region();
}

(a)
```

```c
LOW
while (TRUE) {
    while (turn != 1) /* loop */;
    critical_region();
    turn = 0;
    noncritical_region();
}

(b)
```
Producer Consumer (bounded buffer) Problem

- Formalizes the programs that use a buffer (queue)
- Two processes: producer and consumer that share a fixed size buffer
- Producer puts an item to the buffer
- Consumer takes out an item from the buffer

Max size = 10
Producer Consumer (bounded buffer) Problem

- Formalizes the programs that use a buffer (queue)
- Two processes: producer and consumer that share a fixed size buffer
- Producer puts an item to the buffer
- Consumer takes out an item from the buffer
- What happens when the producer wants to put an item to the buffer while the buffer is already full?

Buffer

Max size = 10

producer

consumer
Producer Consumer (bounded buffer) Problem

- Formalizes the programs that use a buffer (queue)
- Two processes: producer and consumer that share a fixed size buffer
- Producer puts an item to the buffer
- Consumer takes out an item from the buffer
- What happens when the producer wants to put an item to the buffer while the buffer is already full?
- OR when the consumer wants to consume an item from the buffer when the buffer is empty?

Buffer

Max size = 10

producer

consumer
Mutual Exclusion with Sleep and wakeup

- Solution is to use sleep and wakeup
- Sleep: a system call that causes the caller process to block
- Wakeup: a system call that wakes up a process (given as parameter)
- When the producer wants to put an item to the buffer and the buffer is full then it sleeps
- When the consumer wants to remove an item from the buffer and the buffer is empty, then it sleeps.
Sleep and Wakeup

#include <unistd.h>

#define N 100

int count = 0; /* number of slots in the buffer */
/* number of items in the buffer */

void producer(void) {
    int item;

    while (TRUE) {
        /* repeat forever */
        item = produce_item(); /* generate next item */
        if (count == N) sleep(); /* if buffer is full, go to sleep */
        insert_item(item);
        count = count + 1; /* put item in buffer */
        if (count == 1) wakeup(consumer); /* increment count of items in buffer */
        /* was buffer empty? */
    }
}

void consumer(void) {
    int item;

    while (TRUE) {
        /* repeat forever */
        if (count == 0) sleep(); /* if buffer is empty, go to sleep */
        item = remove_item(); /* take item out of buffer */
        count = count - 1; /* decrement count of items in buffer */
        if (count == N - 1) wakeup(producer); /* was buffer full? */
        consume_item(item); /* print item */
    }
}

What problems exist in this solution?

• Consumer is running
• It checks count when count == 0
• Scheduler decides to run Producer just before consumer sleeps
• Producer inserts an item and increments the count
• Producer notices that count is 1, and issues a wakeup call.
• Since consumer is not sleeping yet, the wakeup signal is lost
• Scheduler decides to run the consumer
• Consumer sleeps
• Producer is scheduled, which runs N times, and after filling up the buffer it sleeps
• Both processes sleep forever (or until the prince OS comes and sends a kiss signal to kill both)
Deadlock Illustrated
Solution is to use Semaphores

- Proposed by Dijkstra (11 May 1930 -- 6 August 2002)
- Born in Rotterdam, The Netherlands
- 1972 recipient of the ACM Turing Award (Nobel Prize for computing)
- responsible for
  - the idea of building operating systems as explicitly synchronized sequential processes,
  - the formal development of computer programs
- He is known for
  - His efficient shortest path algorithm and
  - for having designed and coded the first Algol 60 compiler.
- He famously campaigned for the abolition of the GOTO statement from programming!!!
Solution is to use Semaphores

- A semaphore can have
  - value 0 when no wakeups are present
  - Or value > 0 when there are pending wakeups

- Two operations on a semaphore
  - Down: checks the value of the semaphore
    - if value is > 0, then it decrements it (by using one wakeup signal) and continues
    - If value = 0, then the process is put to sleep without completing its down operation
  - Up: increments the value of the semaphore
    - If there are processes sleeping on the semaphore, then one of them is chosen, and it is allowed to complete its down operation

- Checking the value of a semaphore and updating them is done in an atomic fashion (Disabling interrupts when one CPU exists or TSL instruction could be used when multiple CPUs exist)
Semaphores (contd)

• Binary Semaphores are called mutex variables
• They can assume values 0 and 1 only.
• We will see the examples of the usage of mutex variables with threads during the recitations
• In the following slide we will see how mutexes and semaphores are used for mutual exclusion and process synchronization
The producer-consumer problem using semaphores

```c
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```
Mutexes

- Mutexes are simplified versions of semaphores
- they are used when the semaphores ability to count is not needed
- Mutex is a variable that can be in one of two states:
  - Locked (1)
  - Unlocked (0)
- When a thread (or process) needs access to a critical region, it calls `mutex_lock()` on a mutex
- While leaving the critical region it calls `mutex_unlock()`
- If a mutex is already in locked state, then the thread calling the `mutex_lock()` on that mutex blocks
- When a thread calls `mutex_unlock`, then one of the threads blocked on the mutex is unblocked and allowed to acquire the lock.
Mutexes

mutex_lock:
  TSL REGISTER,MUTEX | copy mutex to register and set mutex to 1
  CMP REGISTER,#0 | was mutex zero?
  JZE ok | if it was zero, mutex was unlocked, so return
  CALL thread_yield | mutex is busy; schedule another thread
  JMP mutex_lock | try again later
ok: RET | return to caller; critical region entered

mutex_unlock:
  MOVE MUTEX,#0 | store a 0 in mutex
  RET | return to caller

Implementation of \textit{mutex\_lock} and \textit{mutex\_unlock} using TSL instruction

- A thread calls \texttt{thread\_yield()} to voluntarily give up the CPU to another thread
- Thread yield is a call to the thread scheduler in user space therefore it is very fast
- What if we do not use \texttt{thread\_yield()} in case we are have a “user level” thread library?
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

/* number of slots in the buffer */
/* semaphores are a special kind of int */
/* controls access to critical region */
/* counts empty buffer slots */
/* counts full buffer slots */

void producer(void)
{
    int item;
    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;
    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}

What happens when we do a down on mutex first in the producer function?
Monitors (1)

**Example of a monitor**

```plaintext
monitor example
  integer i;
  condition c;

  procedure producer();
  .
  .
  .
  end;

  procedure consumer();
  .
  .
  .
  end;
end monitor;
```

- A High level synchronization primitive
- Makes our life easier when writing concurrent programs
- It is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module
- Processes may call the procedures in a monitor
- Processes can NOT directly access internal data structures of a monitor from procedures declared outside of the monitor
Monitors (1)

• Only one process can be active in a monitor at any instant
• Monitors are programming language constructs, so compiler handles calls to monitor procedures.
• When a process calls a monitor procedure:
  • the first couple of instructions of the procedure will check to see if any other process is currently using the monitor,
  • if so the calling process will will suspended until the other process leaves the monitor.
  • If no process is using the monitor, the calling process may enter.
Monitors

- For ensuring mutual exclusion, programmer turns all the critical regions into monitor procedures, this way no two process can enter their critical region at the same time
- **Condition variables** are used to block processes
- When a monitor procedure discovers that it can not continue, then it blocks on a condition variable
- A process can wake up another process waiting on a condition variable by issuing a signal operation on that condition variable.
- When a process does a *wait* on a *condition variable*, then it blocks, and another process can now enter its critical region (i.e., execute a procedure inside a monitor)
- When a process *signals* another process waiting on a condition variable, then it must block or exit the monitor immediately! (think why?)
  
  - Signal should be the last statement of a monitor procedure (we will assume this approach is used)
  - Or the process that will wakeup will wait for the process which issued the signal to exit the monitor
Monitors

- Condition variables are not counters (unlike semaphores)
- The signal is lost forever if nobody is waiting for it
- So wait operation must precede the signal!!
Monitors (2)

```plaintext
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count - 1;
        if count = N - 1 then signal(full)
    end;
    count := 0;
end monitor;
```

```plaintext
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

- **Outline of producer-consumer problem with monitors**
  - only one monitor procedure active at one time
  - buffer has $N$ slots
Monitors in Java

- Java is an object oriented programming language that supports user-level threads
- By adding the keyword `synchronized` to a method declaration guarantees that only one thread may execute the synchronized methods inside a class.
  - Ex: `class MyClass { public synchronized foo() { // ... } }
- “wait” and “notify” are the equivalents of “wait” and “signal” for monitors
public class ProducerConsumer {
    static final int N = 100; // constant giving the buffer size
    static producer p = new producer(); // instantiate a new producer thread
    static consumer c = new consumer(); // instantiate a new consumer thread
    static our_monitor mon = new our_monitor(); // instantiate a new monitor
    public static void main(String args[]) {
        p.start(); // start the producer thread
        c.start(); // start the consumer thread
    }
    static class producer extends Thread {
        public void run() { // run method contains the thread code
            int item;
            while (true) { // producer loop
                item = produce_item();
                mon.insert(item);
            }
        }
        private int produce_item() { ... } // actually produce
    }
    static class consumer extends Thread {
        public void run() { // run method contains the thread code
            int item;
            while (true) { // consumer loop
                item = mon.remove();
                consume_item(item);
            }
        }
        private void consume_item(int item) { ... } // actually consume
    }
}

Solution to producer-consumer problem in Java (part 1)
Monitors (4)

```java
static class our_monitor {
    // this is a monitor
    private int buffer[] = new int[N];
    private int count = 0, lo = 0, hi = 0; // counters and indices
    public synchronized void insert(int val) {
        if (count == N) go_to_sleep(); // if the buffer is full, go to sleep
        buffer[hi] = val; // insert an item into the buffer
        hi = (hi + 1) % N; // slot to place next item in
        count = count + 1; // one more item in the buffer now
        if (count == 1) notify(); // if consumer was sleeping, wake it up
    }
    public synchronized int remove() {
        int val;
        if (count == 0) go_to_sleep(); // if the buffer is empty, go to sleep
        val = buffer[lo]; // fetch an item from the buffer
        lo = (lo + 1) % N; // slot to fetch next item from
        count = count - 1; // one few items in the buffer
        if (count == N - 1) notify(); // if producer was sleeping, wake it up
        return val;
    }
    private void go_to_sleep() { try{wait();} catch(InterruptedException exc) {} }
}
```

Solution to producer-consumer problem in Java (part 2)

`go_to_sleep` is like `Wait` but it is uninterruptable

It performs waiting plus exception handling when wait is interrupted
Monitors

- Need to be supported by the programming language such as java
- Monitors (like semaphores) solve the mutual exclusion problem for multiple CPUs with one shared memory.
- In distributed systems where there are multiple CPUs with their own memory then message passing is needed for synchronizing processes
#define N 100
/* number of slots in the buffer */

void producer(void)
{
    int item;
    message m; /* message buffer */

    while (TRUE) {
        item = produce_item(); /* generate something to put in buffer */
        receive(consumer, &m); /* wait for an empty to arrive */
        build_message(&m, item); /* construct a message to send */
        send(consumer, &m); /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m); /* get message containing item */
        item = extract_item(&m); /* extract item from message */
        send(producer, &m); /* send back empty reply */
        consume_item(item); /* do something with the item */
    }
}

Message Passing

- Two primitives are used: send() and receive() which are system calls
  - Send(destination, &message)
  - Receive(source, &message)

The producer-consumer problem with N messages
Message Passing

- **Challenges:**
  - Lost messages (handled through ack messages, but what happens when the ack message is lost?)
  - Naming processes for sending and receiving messages
  - Authentication

- **Buffering:** mailboxes are used to hold messages that have been sent to the destination but that have not been accepted
- **Rendezvous:** no buffering at all, blocking send and receive calls, i.e., sender is blocked until the receiver receives the message.
- Example message passing system: MPI (Message Passing Interface)
- POSIX msgsnd, and msgrcv are used for sending and receiving messages among processes.
Barriers

• Some applications are divided into phases and no process may go to next phase until all processes are ready for it.
• This can be achieved by
  – placing a barrier at the end of each phase
  – And when a process reaches a barrier, it is blocked until all others reach the barrier
Barriers

- **Use of a barrier**
  - processes approaching a barrier
  - all processes but one blocked at barrier
  - last process arrives, all are let through
• Implement a barrier for 2 processes to wait for each other.
Classical IPC Problems

• **Dining philosophers problem (Dijkstra)**
  – Models processes competing for exclusive access to a limited number of resources such as I/O devices

• **Readers and writers problem (Courtois et al.)**
  – Models access to a database (both read and write)

• **Sleeping barber problem**
  – Models queuing situations such as a multi-person helpdesk with a computerized call waiting system for holding a limited number of incoming calls
Dining Philosophers (1)

- 5 Philosophers around a table and 5 forks
- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time (first the right fork then the left one and eat)

What problems may occur in this case?

```c
#define N 5

void philosopher(int i) {
    while (TRUE) {
        think();
        take_fork(i);
        take_fork((i+1) % N);
        eat();
        put_fork(i);
        put_fork((i+1) % N);
    }
}
/* number of philosophers */
/* i: philosopher number, from 0 to 4 */
/* philosopher is thinking */
/* take left fork */
/* take right fork; % is modulo operator */
/* yum-yum, spaghetti */
/* put left fork back on the table */
/* put right fork back on the table */
```
Dining Philosophers (1)

- All philosophers may take their right fork at the same time and block when the left forks are not available.
- Solution: (like collusion detection in Ethernet protocol)
  - pick left fork,
  - if the right fork is not available then release left fork and wait for sometime

There is still a problem!!!!
Dining Philosophers (1)

- What happens when all philosophers do the same thing at the same time?
- This situation is called starvation: a type of deadlock where everybody is doing something but no progress is made.
- Solution is to use a mutex_lock before taking the forks and release the lock after putting the forks back to the table.

Is this a good solution?

No, because only one philosopher can eat at a time but there are enough forks for two!!!


Dining Philosophers (3)

```c
#define N 5
#define LEFT (i+N-1)%N
#define RIGHT (i+1)%N
#define THINKING 0
#define HUNGRY 1
#define EATING 2
typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(int i)
{
    while (TRUE) {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}
```

/* number of philosophers */
/* number of i's left neighbor */
/* number of i's right neighbor */
/* philosopher is thinking */
/* philosopher is trying to get forks */
/* philosopher is eating */
/* semaphores are a special kind of int */
/* array to keep track of everyone's state */
/* mutual exclusion for critical regions */
/* one semaphore per philosopher */

/* i: philosopher number, from 0 to N-1 */
/* repeat forever */
/* philosopher is thinking */
/* acquire two forks or block */
/* yum-yum, spaghetti */
/* put both forks back on table */

Solution to dining philosophers problem (part 1)
Dining Philosophers (4)

```c
void take_forks(int i) {
    down(&mutex); /* enter critical region */
    state[i] = HUNGRY; /* record fact that philosopher i is hungry */
    test(i); /* try to acquire 2 forks */
    up(&mutex); /* exit critical region */
    down(&s[i]); /* block if forks were not acquired */
}

void put_forks(i) {
    down(&mutex); /* enter critical region */
    state[i] = THINKING; /* philosopher has finished eating */
    test(LEFT); /* see if left neighbor can now eat */
    test(RIGHT); /* see if right neighbor can now eat */
    up(&mutex); /* exit critical region */
}

void test(i) {
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Solution to dining philosophers problem (part 2)
Process Geyikleri

- **redrum**: nerde bi process görsem basarım sopayı (kill())
- **k tuncay tekle**: yemek yiyen iki tane filozof görürsem çatalları bi yerlerine ........ kaçarım
- **redrum**: peki beş tanesine ne yaparsın?
- **k tuncay tekle**: ellerimle makarna yaparım
- **redrum**: bence makarnada tuz yerine fare zehiri kullan, hem sen kurtul hem ben kurtulayım hem de tüm cs* camiası kurtulsun.
- **lucifer**: abi o zaman da dying philosophers problemini çözerdi dijkstra...
- **k tuncay tekle**: rahmetli de çok şerefszizmiş doğrusu :)


Readers and Writers Problem

- Assume that there is a database, and processes compete for reading from and writing to the database
- Multiple processes may read the database without any problem
- A process can write to the database only if there are no other processes reading or writing the database
- Here are the basic steps or r/w problem assuming that rc is the reader count (processes currently reading the database)
  - A reader who gains access to the database increments rc (when rc=1, it will lock the database against writers)
  - A reader that finishes reading will decrement rc (when the rc=0 it will unlock the database so that a writer can proceed)
  - A writer can have access to the database when rc = 0 and it will lock the database for other readers or writers
  - Readers will access the database only when there are no writers (but there may be other readers)
The Readers and Writers Problem

typedef int semaphore;    /* use your imagination */
semaphore mutex = 1;      /* controls access to 'rc' */
semaphore db = 1;         /* controls access to the database */
int rc = 0;               /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {        /* repeat forever */
        down(&mutex);   /* get exclusive access to 'rc' */
        rc = rc + 1;    /* one reader more now */
        if (rc == 1) down(&db);  /* if this is the first reader ... */
        up(&mutex);     /* release exclusive access to 'rc' */
        read_data_base();  /* access the data */
        down(&mutex);    /* get exclusive access to 'rc' */
        rc = rc - 1;    /* one reader fewer now */
        if (rc == 0) up(&db);  /* if this is the last reader ... */
        up(&mutex);     /* release exclusive access to 'rc' */
        use_data_read(); /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {        /* repeat forever */
        think_up_data();   /* noncritical region */
        down(&db);         /* get exclusive access */
        write_data_base(); /* update the data */
        up(&db);           /* release exclusive access */
    }
}

A solution to the readers and writers problem

What is the problem with this solution?

The writer will starve when there is constant supply of readers !!!!!

Solution is to queue new readers behind the current writer at the expense of reduced concurrency
The Sleeping Barber Problem (1)

- There is one barber, and n chain of waiting customers
- If there are no customers, then the barber sits in his chair and sleeps (as illustrated in the picture)
- When a new customer arrives and the barber is sleeping, then he will wakeup the barber
- When a new customer arrives, and the barber is busy, then he will sit on the chairs if there is any available, otherwise (when all the chairs are full) he will leave.
The Sleeping Barber Problem (2)

Solution to sleeping barber problem.

```c
#define CHAIRS 5          /* # chairs for waiting customers */
typedef int semaphore;  /* use your imagination */
semaphore customers = 0; /* # of customers waiting for service */
semaphore barbers = 0;  /* # of barbers waiting for customers */
semaphore mutex = 1;    /* for mutual exclusion */
int waiting = 0;        /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers); /* go to sleep if # of customers is 0 */
        down(&mutex);    /* acquire access to `waiting` */
        waiting = waiting - 1; /* decrement count of waiting customers */
        up(&barbers);    /* one barber is now ready to cut hair */
        up(&mutex);      /* release `waiting` */
        cut_hair();      /* cut hair (outside critical region) */
    }
}

void customer(void)
{
    down(&mutex);        /* enter critical region */
    if (waiting < CHAIRS) {
        waiting = waiting + 1; /* if there are no free chairs, leave */
        up(&customers);      /* increment count of waiting customers */
        up(&mutex);          /* wake up barber if necessary */
        down(&barbers);      /* release access to `waiting` */
        get_haircut();       /* go to sleep if # of free barbers is 0 */
        up(&mutex);          /* be seated and be serviced */
    } else {
        up(&mutex);          /* shop is full; do not wait */
    }
}
```