CSE 3320
Operating Systems
Synchronization

Jia Rao

Department of Computer Science and Engineering

http://ranger.uta.edu/~jrao

Recap of the Last Class

- Multiprocessor scheduling
  - Two implementations of the ready queue
  - Load balancing
  - Parallel program scheduling
    - Synchronizations on shared data and execution phases
    - Causality among threads

Inter-process or thread communications

Inter-Process Communication (IPC)

- Three fundamental issues:
  - How one process can pass information to another
  - How to make sure two or more processes do not get into each other’s way when engaging in critical activities
  - How to maintain proper sequencing when dependencies present

How about inter-thread communication ?

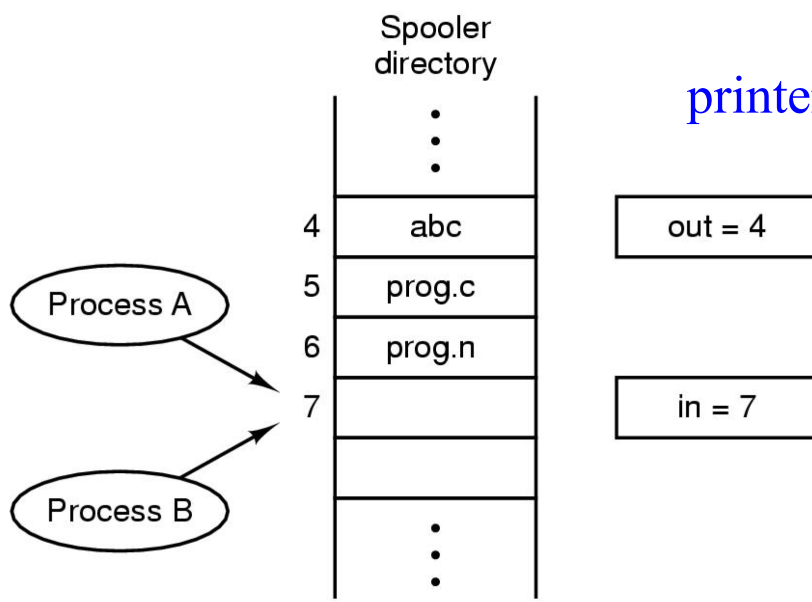
Race Conditions

- Race conditions: when two or more processes/threads are reading or writing some *shared data* and the final results depend on who runs precisely when
    - Interrupts, interleaved operations/execution

Dir[in] = X;
in ++;

Dir[in] = Y;
in ++;

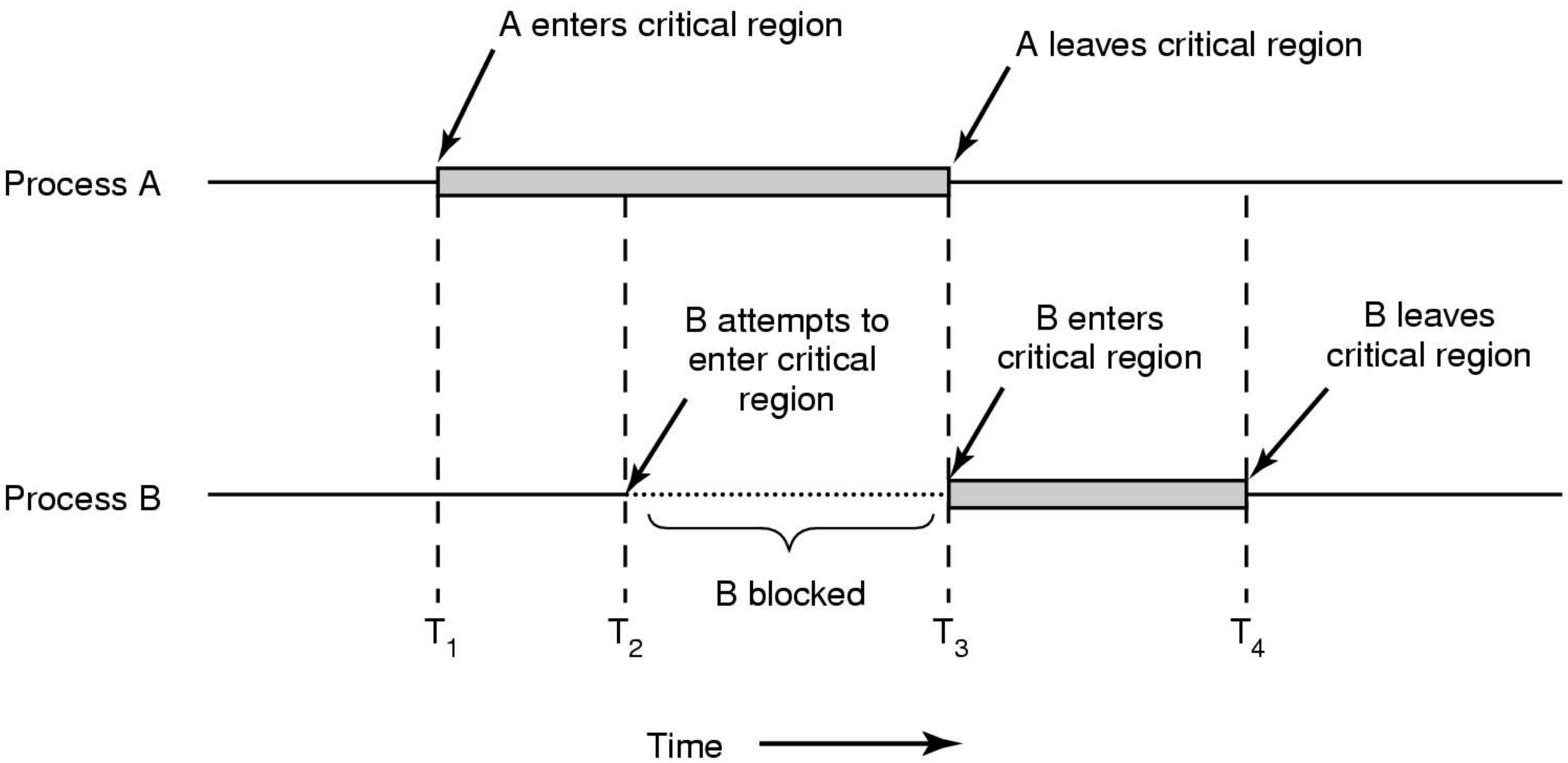
printer daemon



Mutual Exclusion and Critical Regions

- Mutual exclusion: makes sure if one process is using a shared variable or file, the other processes will be excluded from doing the same thing
  - Main challenge/issue to OS: to design appropriate primitive operations for achieving mutual exclusion
- Critical regions: the part of the program where the shared memory is accessed
- Four conditions to provide mutual exclusion
  - No two processes simultaneously in critical region
  - No assumptions made about speeds or numbers of CPUs
  - No process running outside its critical region may block another process
  - No process must wait forever to enter its critical region

Mutual Exclusion Using Critical Regions



Mutual exclusion using critical regions

Mutual Exclusion with Busy Waiting

- Disabling interrupts:
  - OS technique, not users’
  - multi-CPU?
- Lock variables:
  - test-set is a two-step process, not atomic
- Busy waiting:
  - continuously testing a variable until some value appears (*spin lock*)

Busy Waiting: Strict Alternation

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

(a)

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

(b)

Proposed *strict alternation* solution to critical region problem

(a) Process 0. (b) Process 1.

What if P1's noncritical_region() has lots more work than P0's?

Busy Waiting: Peterson's

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

int turn;                           /* whose turn is it? */
int interested[N];       sharing    /* 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 */
}     Different from strict alternation
```

Peterson's solution for achieving mutual exclusion

Busy Waiting: TSL

- TSL (Test and Set Lock)
  - Indivisible (atomic) operation, how? Hardware (multi-processor)
  - How to use TSL to prevent two processes from simultaneously entering their critical regions?

```
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
```

Entering and leaving a critical region using the TSL instruction

Sleep and Wakeup

- Issue I with Peterson’s & TS: how to avoid CPU-costly busy waiting?
- Issue II: priority inversion problem
  - Consider two processes, H with (strict) high priority and L with (strict) low priority, L is in its critical region and H becomes ready; *does L have chance to leave its critical region*?
- Some IPC primitives that block instead of wasting CPU time when they are not allowed to enter their critical regions
  - Sleep and wakeup

Sleep and Wakeup – Producer-Consumer Problem

```
#define N 100                              /* number of slots in the buffer */
int count = 0;                             /* 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);               /* put item in buffer */
          count = count + 1;               /* increment count of items in buffer */
          if (count == 1) wakeup(consumer); /* was buffer empty? */
     }
}
```

Q1: What if the wakeup signal sent to a non-sleep (ready) process?

```
void consumer(void)
{
     int item;

     while (TRUE) {                        /* repeat forever */
          if (count == 0) sleep( );        /* if buffer is empty, got 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 */
     }
}
```

Q2: what is a wakeup waiting bit? Is one enough?

Semaphores and P&V Operations

- Semaphores: a variable to indicate the # of pending wakeups
- *Down* operation (P; request): lock
  - Checks if a semaphore is > 0,
    - if so, it decrements the value and just continue
    - Otherwise, the process is put to sleep without completing the down for the moment
- *Up* operation (V; release): unlock
  - Increments the value of the semaphore
    - if one or more processes are sleeping on the semaphore, one of them is chosen by the system (randomly) and allowed to complete its down (semaphore will still be 0)
- P & V operations are atomic, how to implement?
  - Single CPU: system calls, disabling interrupts temporally
  - Multiple CPUs: TSL help

The Producer-consumer Problem w/ Semaphores

```
#define N 100                          /* number of slots in the buffer */
typedef int semaphore;                 /* semaphores are a special kind of int */
semaphore mutex = 1;                   /* controls access to critical region */
semaphore empty = N;                   /* counts empty buffer slots */
semaphore full = 0;                    /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                     /* TRUE is the constant 1 */
        item = produce_item( );        /* generate something to put in buffer */
        down(&empty);                  /* decrement empty count */
        down(&mutex);                  /* enter critical region */
        insert_item(item);             /* put new item in buffer */
        up(&mutex);                    /* leave critical region */
        up(&full);                     /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                     /* infinite loop */
        down(&full);                   /* decrement full count */
        down(&mutex);                  /* enter critical region */
        item = remove_item( );         /* take item from buffer */
        up(&mutex);                    /* leave critical region */
        up(&empty);                    /* increment count of empty slots */
        consume_item(item);            /* do something with the item */
    }
}
```

For mutual exclusion and synchronization

Binary semaphores: if each process does a down before entering its critical region and an up just leaving it, mutual exclusion is achieved

Mutexes

- Mutex:
  - a variable that can be in one of two states: unlocked or locked
  - A simplified version of the semaphores [0, 1]

```
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
```

Give other chance to run so as to save self;
What is mutex_trylock()?

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

Mutexes – User-space Multi-threading

- What is a key difference between *mutex_lock* and *enter_region* in multi-threading and multi-processing?
  - For user-space multi-threading, a thread has to allow other threads to run and release the lock so as to enter its critical region, which is impossible with busy waiting *enter_region*

```
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
```

Two *processes* entering and leaving a critical region using the TSL instruction

Monitors

- Monitor: a higher-level synchronization primitive
  - Only one process can be active in a monitor at any instant, *with compiler's help*; thus, how about to put all the critical regions into monitor procedures for *mutual exclusion*?

monitor *example*
    integer *i*;
    condition *c*;

    procedure *producer*( );
    .
    .
    .
    end;

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

But, how processes block when they cannot proceed?

Condition variables, and two operations: *wait()* and *signal()*

Monitors (2)

Wakeup and sleep signals can lost, but not Wait and signal signals, why ?

```
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;
```

```
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;
```

Conditions are not counters; *wait()* before *signal()*

Outline of producer-consumer problem with monitors

- only one monitor procedure active at one time (a process doing *signal* must exit the monitor immediately); buffer has *N* slots

Monitor (3)

- Pros
  - Make mutual exclusion automatic
  - Make parallel programming less error-prone
- Cons
  - Compiler support

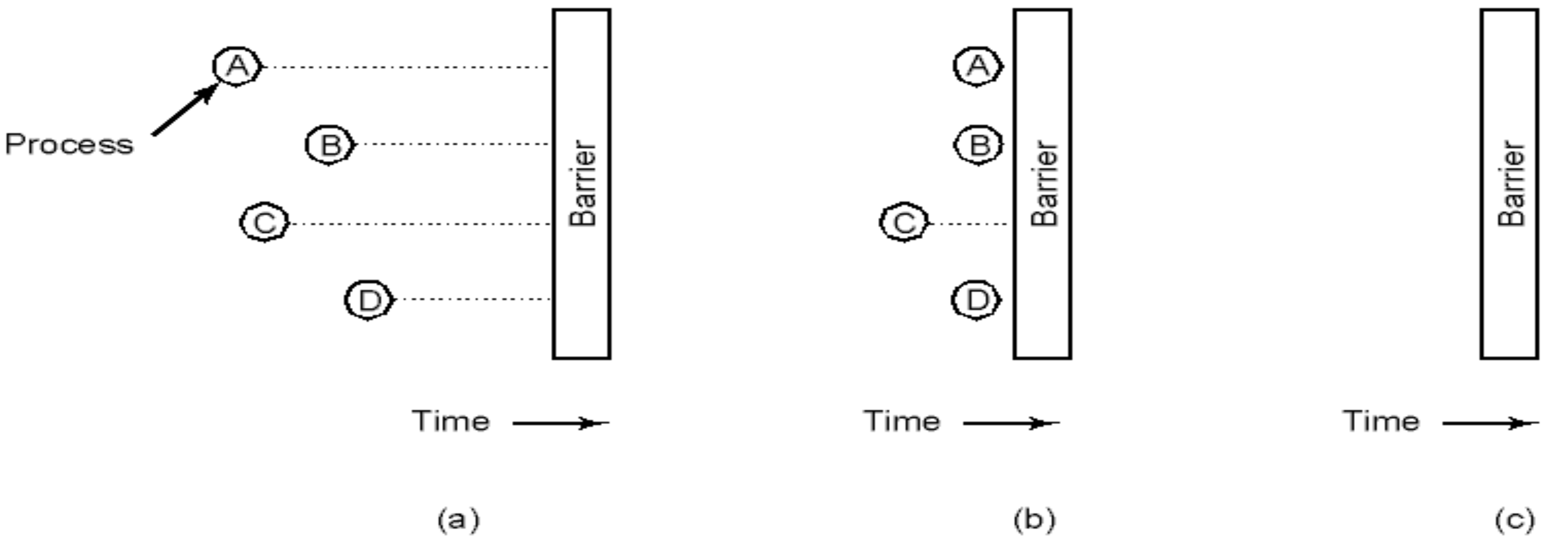
Message Passing

```
#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 */
    }
}
```

Communication without sharing memory

The producer-consumer problem with N messages

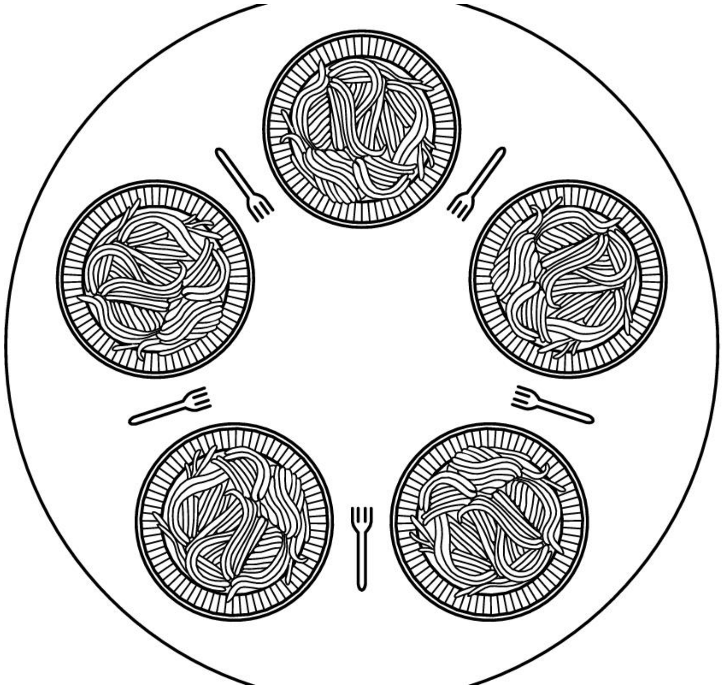
Barriers



- Use of a barrier (for programs operate in *phases,* neither enters the next phase until all are finished with the current phase) for groups of processes to do synchronization
  - (a) processes approaching a barrier
  - (b) all processes but one blocked at barrier
  - (c) last process arrives, all are let through

Class IPC Problems: Dining Philosophers

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock & starvation
  - *Deadlock*: both are blocked on some resource
  - *Starvation*: both are running, but no progress made



The problem is useful for modeling processes that are competing for exclusive access to a limited number of resources, such as I/O devices

Dining Philosophers (2)

```
#define N 5                          /* number of philosophers */

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

A non-solution to the dining philosophers problem

What happens if all philosophers pick up their left forks simultaneously?

Or, all wait for the same amount of time, then check if the right available?

What if random waiting, then check if the right fork available?

What performance if *down* and *up* on *mutex* before acquiring/replacing a fork?

Dining Philosophers (3): Solution part1

```
#define N            5           /* number of philosophers */
#define LEFT         (i+N−1)%N   /* number of i's left neighbor */
#define RIGHT        (i+1)%N     /* number of i's right neighbor */
#define THINKING     0           /* philosopher is thinking */
#define HUNGRY       1           /* philosopher is trying to get forks */
#define EATING       2           /* philosopher is eating */
typedef int semaphore;           /* semaphores are a special kind of int */
int state[N];                    /* array to keep track of everyone's state */
semaphore mutex = 1;             /* mutual exclusion for critical regions */
semaphore s[N];                  /* one semaphore per philosopher */

void philosopher(int i)          /* i: philosopher number, from 0 to N−1 */
{
     while (TRUE) {              /* repeat forever */
          think( );              /* philosopher is thinking */
          take_forks(i);         /* acquire two forks or block */
          eat( );                /* yum-yum, spaghetti */
          put_forks(i);          /* put both forks back on table */
     }
}
```

Dining Philosophers (4): Solution part2

```
void take_forks(int i)                  /* i: philosopher number, from 0 to N–1 */
{
    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)                       /* i: philosopher number, from 0 to N–1 */
{
    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)                            /* i: philosopher number, from 0 to N–1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

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 */
     }
}
```

Summary

- Race conditions
- Mutual exclusion and critical regions
- Two simple approaches
  - Disabling interrupt and Lock variables
- Busy waiting
  - Strict alternation, Peterson's and TSL
- Sleep and Wakeup
- Semaphores
- Mutexes
- Classical IPC problems
- Additional practice
  - Read Linux documentation: LINUX_SRC/Documentation/spinlocks.txt
  - Find the implementation of `down` and `up` in LINUX_SRC/kernel/semaphore.c
  - Spinlock v.s. Mutex: http://stackoverflow.com/questions/5869825/when-should-one-use-a-spinlock-instead-of-mutex