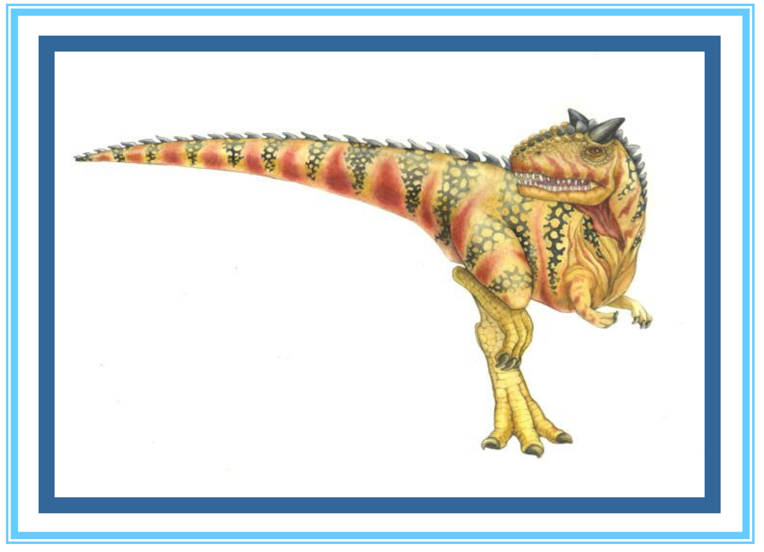
Chapter 6: Synchronization



Chapter 6: Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Alternative Approaches

Objectives

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:
  Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer **`counter`** that keeps track of the number of full bufrs. Initially, **`counter`** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {
    /* produce an item in next produced */

    while (counter == BUFFER SIZE) ;
        /* do nothing */
    buffer[in] = next produced;
    in = (in + 1) % BUFFER SIZE;
    counter++;
}
```

Consumer

```
while (true) {
    while (counter == 0)
        ; /* do nothing */
    next consumed = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    counter--;
    /* consume the item in next consumed
*/
}
```

Race Condition

- `counter++` could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- `counter--` could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 – 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

Critical Section Problem

- Consider system of ***n*** processes {$p_0, p_1, \ldots p_{n-1}$}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

A General Framework for Synchronization

the Critical-Section Problem

```
do {
permission request  →   entry section;
                        critical section;
exit notification   →   exit section;
                        remainder section;
} while (1);
```

Assumptons:

- Atomic execution of each statement line
- Interleaving execution among processes

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process $\boldsymbol{P_i}$ is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the ***n*** processes

Solution to Critical-Section Problem

1. Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode

Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the **`load`** and **`store`** instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - **`int turn;`**
  - **`Boolean flag[2]`**
- The variable **`turn`** indicates whose turn it is to enter the critical section
- The **`flag`** array is used to indicate if a process is ready to enter the critical section. **`flag[i] = true`** implies that process P_i is ready!

Algorithm for Process P_i

```
do {
    flag[i] = true;
    turn = 1-i;
    while (flag[1-i] && turn == 1-i);
        critical section
    flag[i] = false;
        remainder section
} while (true);
```

- Provable that

1. Mutual exclusion is preserved
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

Peterson's Solution

Proof for the mutual exclusion (1/3)

Lemma 1: When a Pi is is in either the entry or the critical sections, flag[i] = true.

Proof. Straightforward. ■

```
do {
1.    flag[i] = TRUE;
2.    turn = 1-i;
3.    while (flag[1-i] && turn == 1-i);
4.              critical section
5.    flag[i] = FALSE;
6.              remainder section
} while (TRUE);
```

Peterson's Solution

Proof for the mutual exclusion (2/3)

Lemma 2: Mutual exclusion is maintained by Peterson's algorithm.

Proof: For convenience, a state is denoted as [t,h,k,f0,f1]

- t the value of turn,
- h is the statement index of P0,
- k the statement index of P1,
- f0 the value of flag[0], and
- f1 the value of flag[1].

According to lemma 1, we assume that [0,4,4,1,1] happens.

This implies that P0 enters the critical section last from [0,3,4,1,1].

Proof for the mutual exclusion (3/3)

There are two possibilities of the predecessor to [0,3,4,1,1].

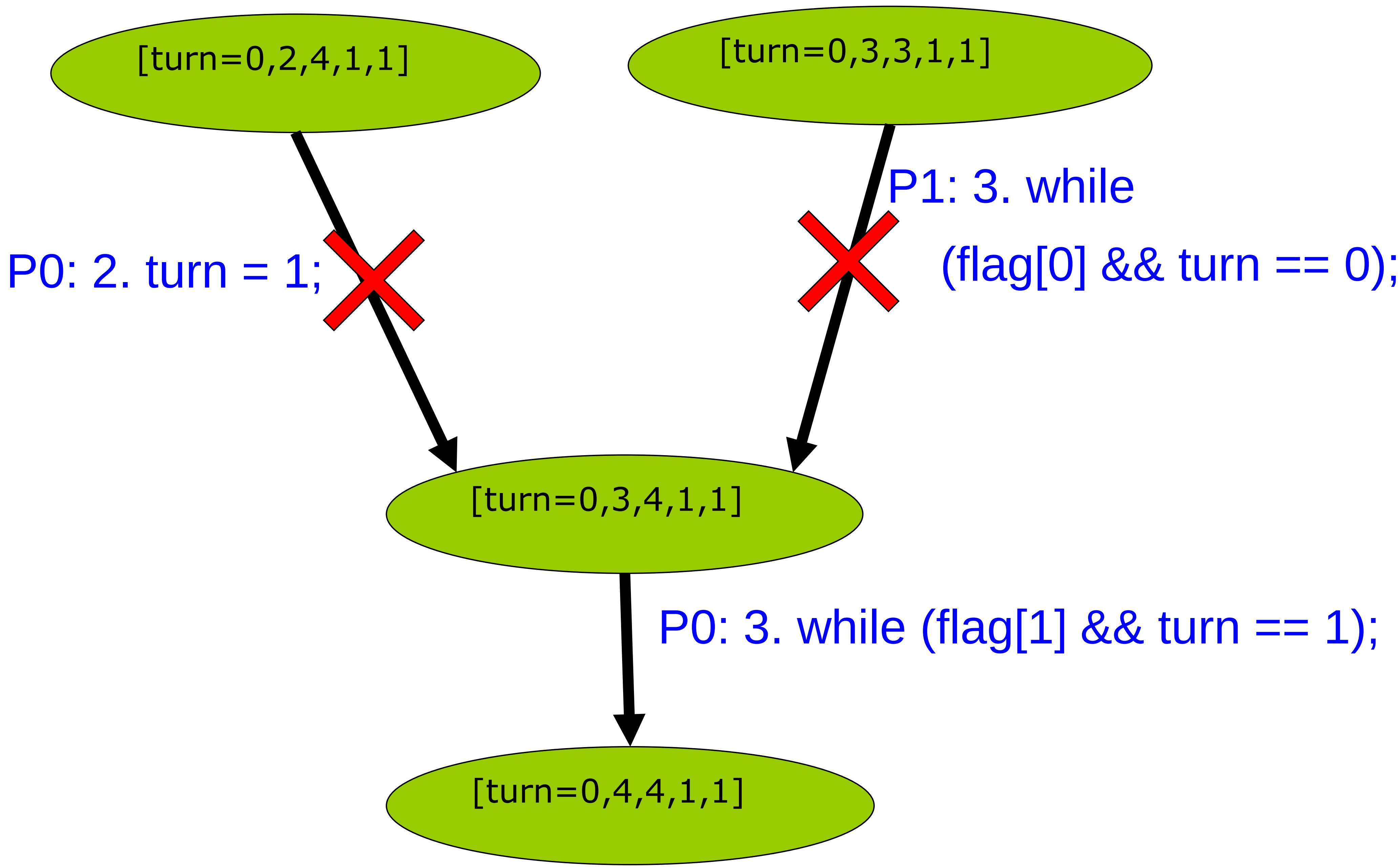
- One possible predecessor of [0,3,4,1,1] is [0,3,3,1,1] which is impossible.
  - From [0,3,3,1,1], the while loop condition for P1 is false.
- The other possible predecessor of [0,3,4,1,1] is [?,2,4,1,1] which is also impossible.
  - From [?,2,4,1,1], statement 2 for P0 changes turn to 1 instead of 0.

Since both possibilities are contradictions, the assumption of violation of mutual exclusion is a contradiction.

Thus the lemma is proven. +

Peterson’s algorithm

Backward refutation tree



2014/11/25 stopped here

Peterson's Solution Properties

- Mutual Exclusion
  - The eventual value of *turn* determines which process enters the critical section.
- Progress
  - A process can only be stuck in the while loop, and the process which can keep it waiting must be in its critical sections.
- Bounded Waiting
  - Each process wait at most one entry by the other process.

Peterson's Solution Properties

- How to argue for the Bounded Waiting property ?

[1,3,3,1,1]

↓ P1: while (flag[0] && turn == 0);

[1,3,4,1,1]

↓ P1: critical section

[1,3,5,1,1]

↓ P1: flag[1] = false;

[1,3,6,1,0]

↓ P0: while (flag[1] && turn == 1);

[1,4,6,1,0] Wrong argument!

The critical-section problem

A solution for n processes

Bakery Algorithm

- Originally designed for distributed systems
- Token-based
  - Processes which are ready to enter their critical section must take a number and wait till the number becomes the lowest.
- Two arrays of local variables
  - int number[i]:
    - Pi's token number if it is nonzero.
  - boolean choosing[i]:
    - Pi is taking a number.

The critical-section problem

A solution for n processes

```
do {
    choosing[i]=true;
    number[i]=max(number[0], …number[n-1])+1;
    choosing[i]=false;
    for (j=0; j < n; j++) {
        while choosing[j] ;
        while (number[j] != 0 && (number[j],j)<(number[i],i)) ;
    }
    critical section
    number[i]=0;
    remainder section
} while (1);
```

An observation: If

- Pi is in its critical section, and
- Pk (k != i) has already chosen its number[k],

then (number[i],i) < (number[k],k).

Synchronization Hardware

- Many systems provide hardware support for critical section code
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

```
do {
   acquire lock
      critical section
   release lock
      remainder section
} while (TRUE);
```

test_and_set Instruction

- Definition:

```
boolean test_and_set (boolean *target)
 {
       boolean rv = *target;
       *target = TRUE;
       return rv:
 }
```

Solution using test_and_set()

- Shared boolean variable lock, initialized to FALSE
- Solution:

```
do {
     while (test_and_set(&lock))
      ; /* do nothing */
   /* critical section */
   lock = false;
   /* remainder section */
} while (true);
```

compare_and_swap Instruction

- Definition:

```
int compare and swap(int *value, int
  expected, int new value) {
   int temp = *value;
   if (*value == expected)
      *value = new value;
   return temp;
}
```

Solution using compare_and_swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key
- Solution:

```
do {
        while (compare and swap(&lock, 0, 1) != 0)
          ; /* do nothing */
          /* critical section */
     lock = 0;
          /* remainder section */
} while (true);
```

Bounded-waiting Mutual Exclusion with test_and_set

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = false;
   else
      waiting[j] = false;
   /* remainder section */
} while (true);
```

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Product critical regions with it by first **`acquire()`** a lock then **`release()`** it
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

acquire() and release()

```
acquire() {
   while (!available) ; /* busy wait */
   available = false;;
}
release() {
   available = true;
}
do {
   acquire lock
      critical section
   release lock
      remainder section
} while (true);
```

OS Solutions (馬江帆) Semaphores

- Synchronization tool that does not require busy waiting
- Semaphore ***S*** – integer variable
- Two standard operations modify ***S***: **`wait()`** and **`signal()`**
  - Originally called **`P()`** and **`V()`**
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

```
wait (S) {
    while (S <= 0) ; // busy wait
    S--;
}
signal (S) { S++; }
```

Semaphore Usage（張文博）

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Then a **mutex lock**
- Can implement a counting semaphore ***S*** as a binary semaphore
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

```
P1:      S1;
         signal(synch);


P2:      wait(synch);
         S2;
```

Semaphore Implementation（羅毅明）

- Must guarantee that no two processes can execute **`wait ()`** and **`signal ()`** on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

Semaphore Implementation with no Busy waiting (Cont.)

```
typedef struct{
   int value;
   struct process *list;
} semaphore;

wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}
```

```
signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let *S* and *Q* be two semaphores initialized to 1

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`.`	`.`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Deadlock and Starvation

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {
        ...
      /* produce an item in next_produced */
        ...
   wait(empty);
   wait(mutex);
        ...
      /* add next produced to the buffer */
        ...
   signal(mutex);
   signal(full);
} while (true);
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
do {
    wait(full);
    wait(mutex);
        ...
      /* remove an item from buffer to next_consumed
   */
        ...
    signal(mutex);
    signal(empty);
        ...
      /* consume the item in next consumed */
        ...
   } while (true);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are treated – all involve priorities

Readers-Writers Problem

- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
        wait(rw mutex);

          ...
        /* writing is performed */

          ...
     signal(rw mutex);
} while (true);
```

OS solutions

Readers-Writers Problem (Cont.)

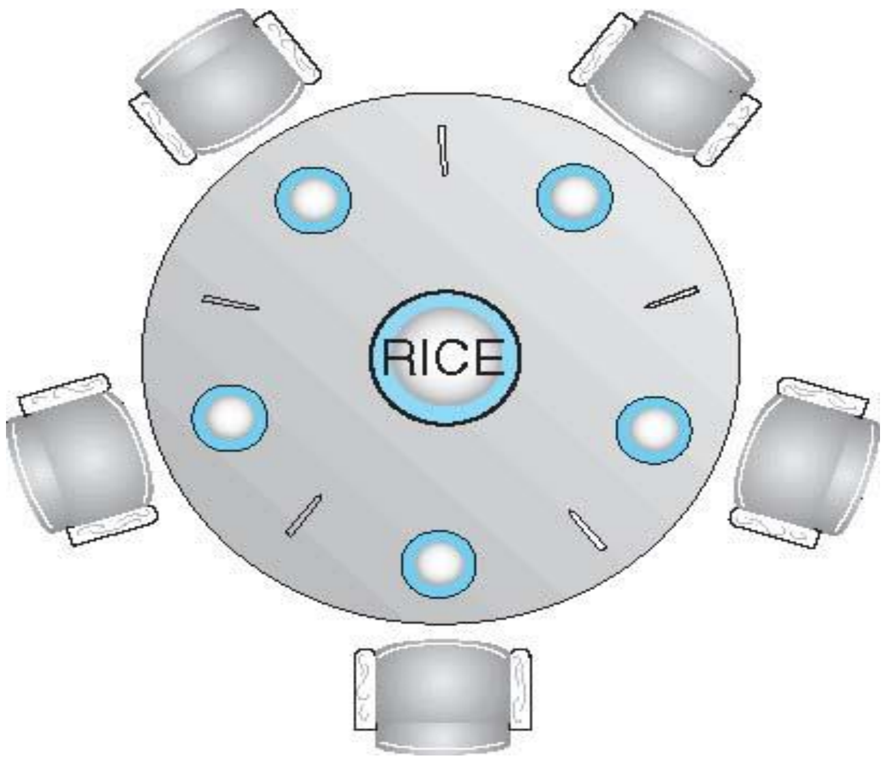
- The structure of a reader process

```
do {          // at any moment,
              // at most one reader in entry or exit section.
              wait (mutex) ;  // begin of entry section
              readcount ++ ;
              if (readcount == 1)
                     wait (wrt) ;
              signal (mutex) // end of entry section
           // critical section, reading is performed
               wait (mutex) ; // begin of exit section
               readcount  - - ;
               if (readcount  == 0)
                     signal (wrt) ;
               signal (mutex) ; // end of exit section
                 // remainder section.
    } while (TRUE);
```

Readers-Writers Problem Variations

- *First* variation – no reader kept waiting unless writer has permission to use shared object
- *Second* variation – once writer is ready, it performs write asap
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

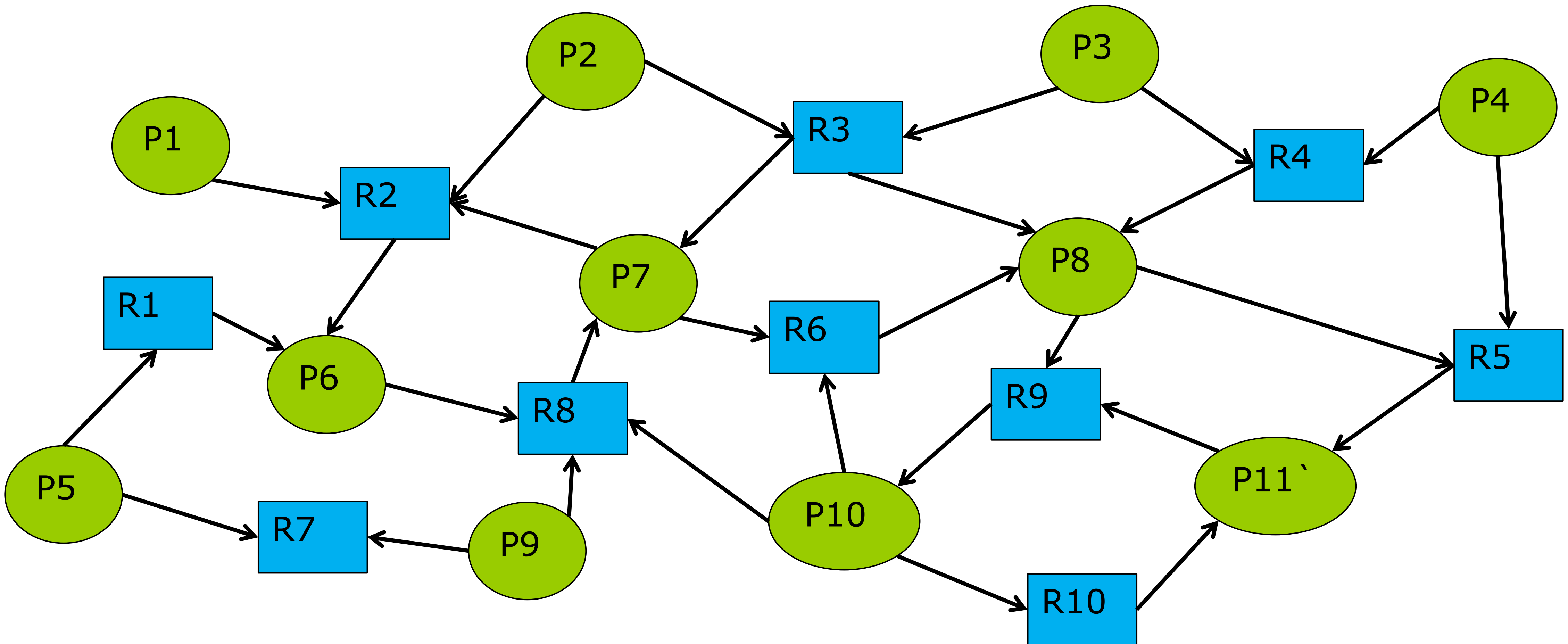
Dining-Philosophers Problem



- Philosophers spend their lives thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

OS solutions

Dining-Philosophers Problem



- Shared resources
- Processes

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do  {
     wait ( chopstick[i] );
    wait ( chopStick[ (i + 1) % 5] );
         //  eat
    signal ( chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );
         //  think
} while (TRUE);
```

- What is the problem with this algorithm?

Problems with Semaphores

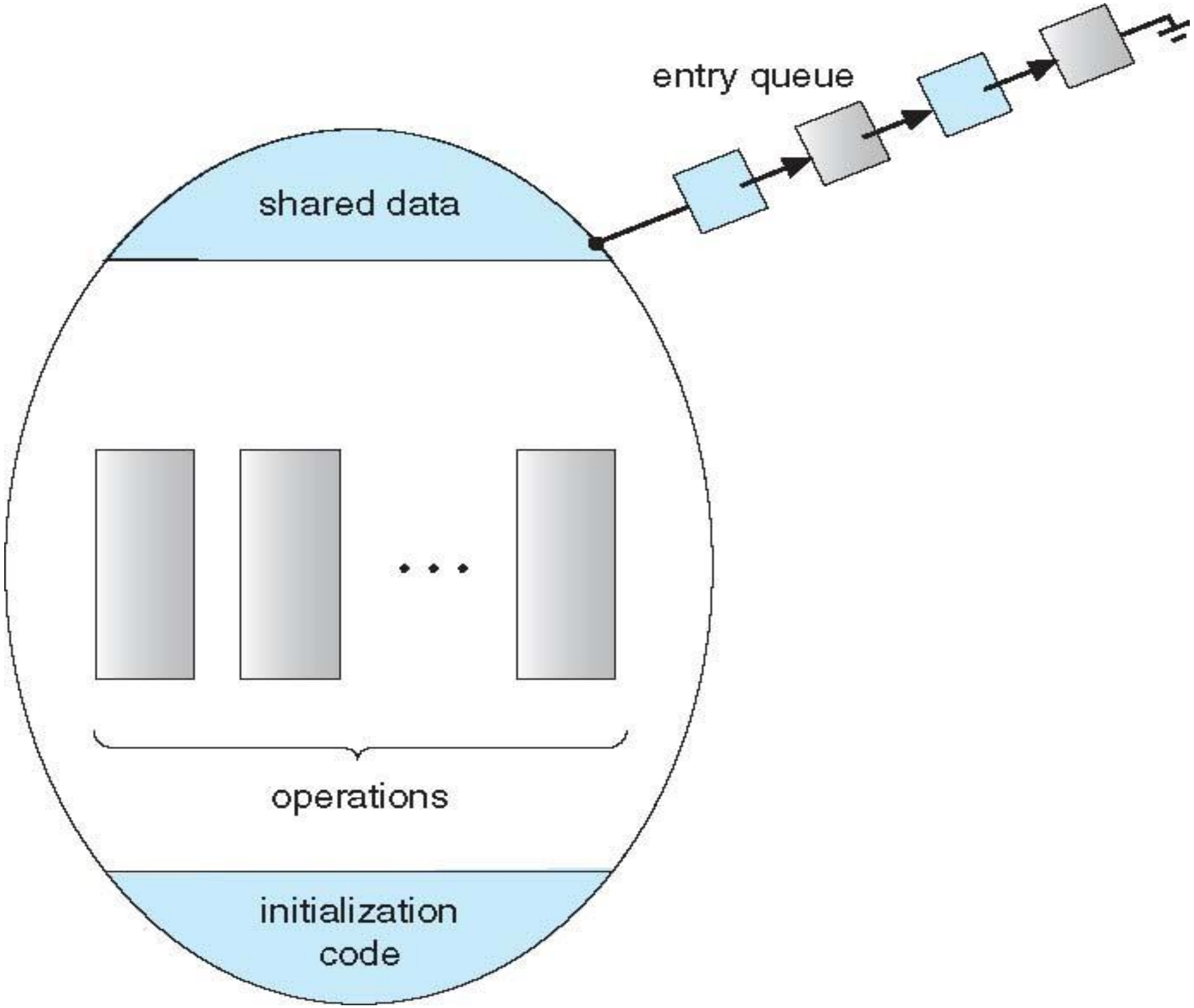
- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation

Programming Language (OO) Solutions Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

```
monitor monitor-name {
    // shared variable declarations
    procedure P1 (…) { …. }
    procedure Pn (…) {……}
     Initialization code (…) { … }
    }
}
```

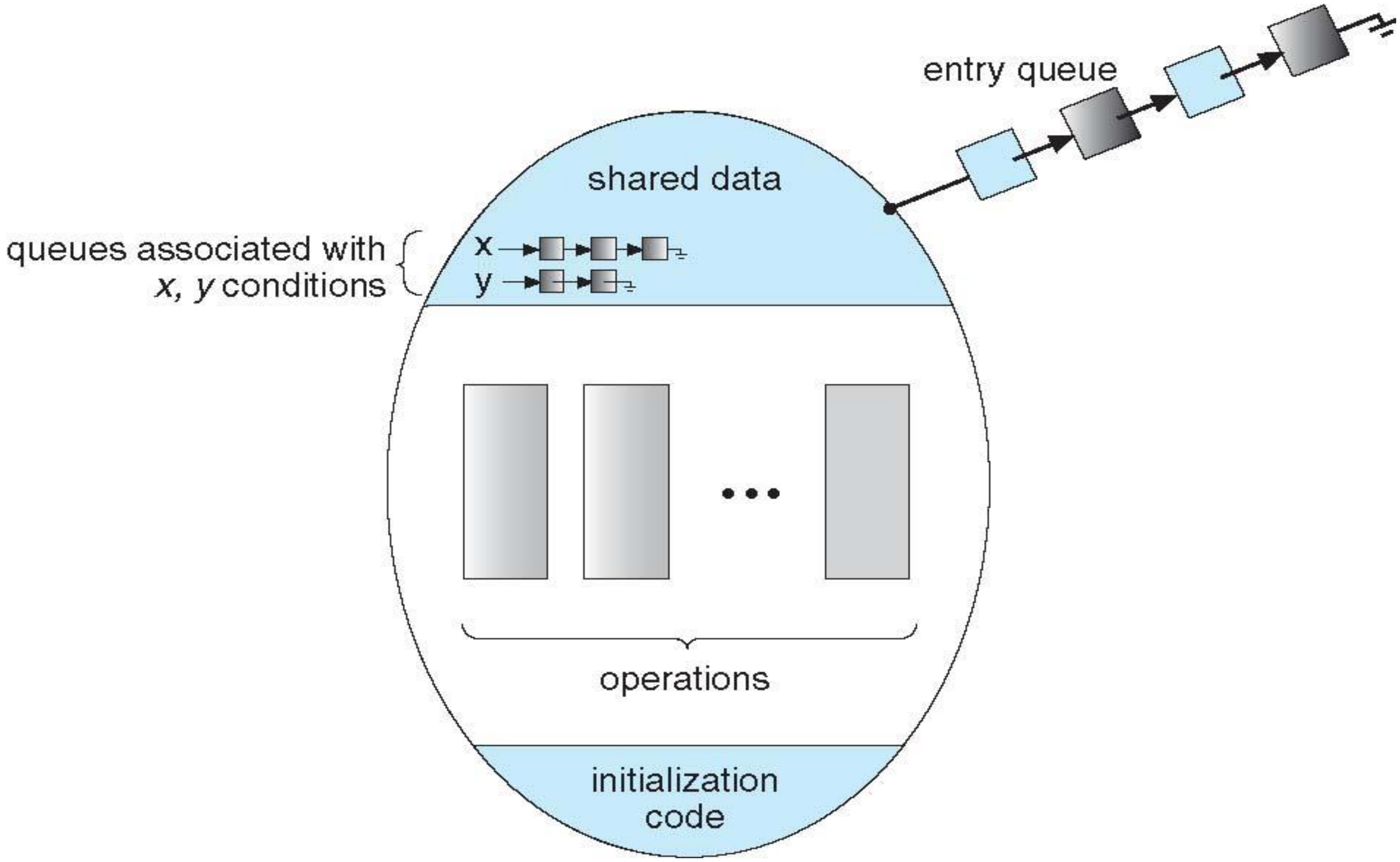
Schematic view of a Monitor



Condition Variables

- condition x, y;
- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended until x.signal ()
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()
    - If no x.wait () on the variable, then it has no effect on the variable

Monitor with Condition Variables



Condition Variables Choices

- If process P invokes x.signal (), with Q in x.wait () state, what should happen next?
  - If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q leaves monitor or waits for another condition
  - **Signal and continue** – Q waits until P leaves the monitor or waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java

Programming Language (OO) Solutions

Monitors

Guarantee of no simultaneous execution within a monitor

- Some implementation issues
  - Signal on conditional variables
    - signal and wait
      - P invokes signal and either
        - wait until Q leaves or
        - P wait for another condition
    - signal and continue
    - Q waits until P leaves or P waits for another condition.

Programming Language (OO) Solutions

Monitors

Guarantee of no simultaneous execution within a monitor

- Some implementation issues (continued)
  - Resumption order ?
    - FCFS
    - Given priority at suspension time
      - x.wait(c), c is a priority

Programming Language (OO) Solutions

Monitors

```
monitor DiningPhilosophers
  {
    enum { THINKING; HUNGRY, EATING) state [5] ;
    condition self [5];

    void pickup (int i) {
           state[i] = HUNGRY;
           test(i);
           if (state[i] != EATING) self [i].wait;
    }

     void putdown (int i) {
           state[i] = THINKING;
                 // test left and right neighbors
            test((i + 4) % 5);
            test((i + 1) % 5);
     }
```

Programming Language (OO) Solutions

Monitors (Cont.)

```
    void test (int i) {
        if ( (state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
            state[i] = EATING ;
         self[i].signal () ;
         }
     }

     initialization_code() {
        for (int i = 0; i < 5; i++)
        state[i] = THINKING;
    }
}
```

Programming Language (OO) Solutions

Monitors (Cont.)

- Each philosopher *i* invokes the operations pickup() and putdown() in the following sequence:

```
DiningPhilosophers.pickup (i);

    EAT

DiningPhilosophers.putdown (i);
```

- No deadlock, but starvation is possible

Programming Language (OO) Solutions

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0;
```

- Each procedure ***F*** will be replaced by

```
wait(mutex);
    …
    // body of F;
    …
if (next_count > 0)
  signal(next)
else
  signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Programming Language (OO) Solutions

Monitor Implementation – Condition Variables

- For each condition variable ***x***, we have:

```
semaphore x_sem; // (initially  = 0)
int x_count = 0;
```

- The operation x.wait can be implemented as:

```
x-count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x-count--;
```

Programming Language (OO) Solutions

Monitor Implementation (Cont.)

- The operation x.signal can be implemented as:

```
if (x-count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

Programming Language (OO) Solutions

Resuming Processes within a Monitor

- If several processes queued on condition x, and x.signal() executed, which should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form x.wait(c)
  - Where c is **priority number**
  - Process with lowest number (highest priority) is scheduled next

Programming Language (OO) Solutions

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time) {
            if (busy)
                x.wait(time);
            busy = TRUE;
    }
```

```
    void release() {
            busy = FALSE;
            x.signal();
    }
    initialization code() {
     busy = FALSE;
    }
}
```

Programming Language (OO) Solutions Monitors

- Drawbacks - Access order violations
  - access without gaining permission
  - never release after permission
  - releases without gaining permission
  - double requests

Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses **adaptive mutexes** for efficiency when protecting data from short code segments
  - Starts as a standard semaphore spin-lock
  - If lock held, and by a thread running on another CPU, spins
  - If lock held by non-run-state thread, block and sleep waiting for signal of lock being released

Solaris Synchronization

- Uses **condition variables**
- Uses **readers-writers** locks when longer sections of code need access to data
- Uses **turnstiles** to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
  - Turnstiles are per-lock-holding-thread, not per-object
- Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile

Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted
- Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers
  - **Events**
    - An event acts much like a condition variable
  - Timers notify one or more thread when time expired
  - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)

Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - semaphores
  - spinlocks
  - reader-writer versions of both
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spinlocks

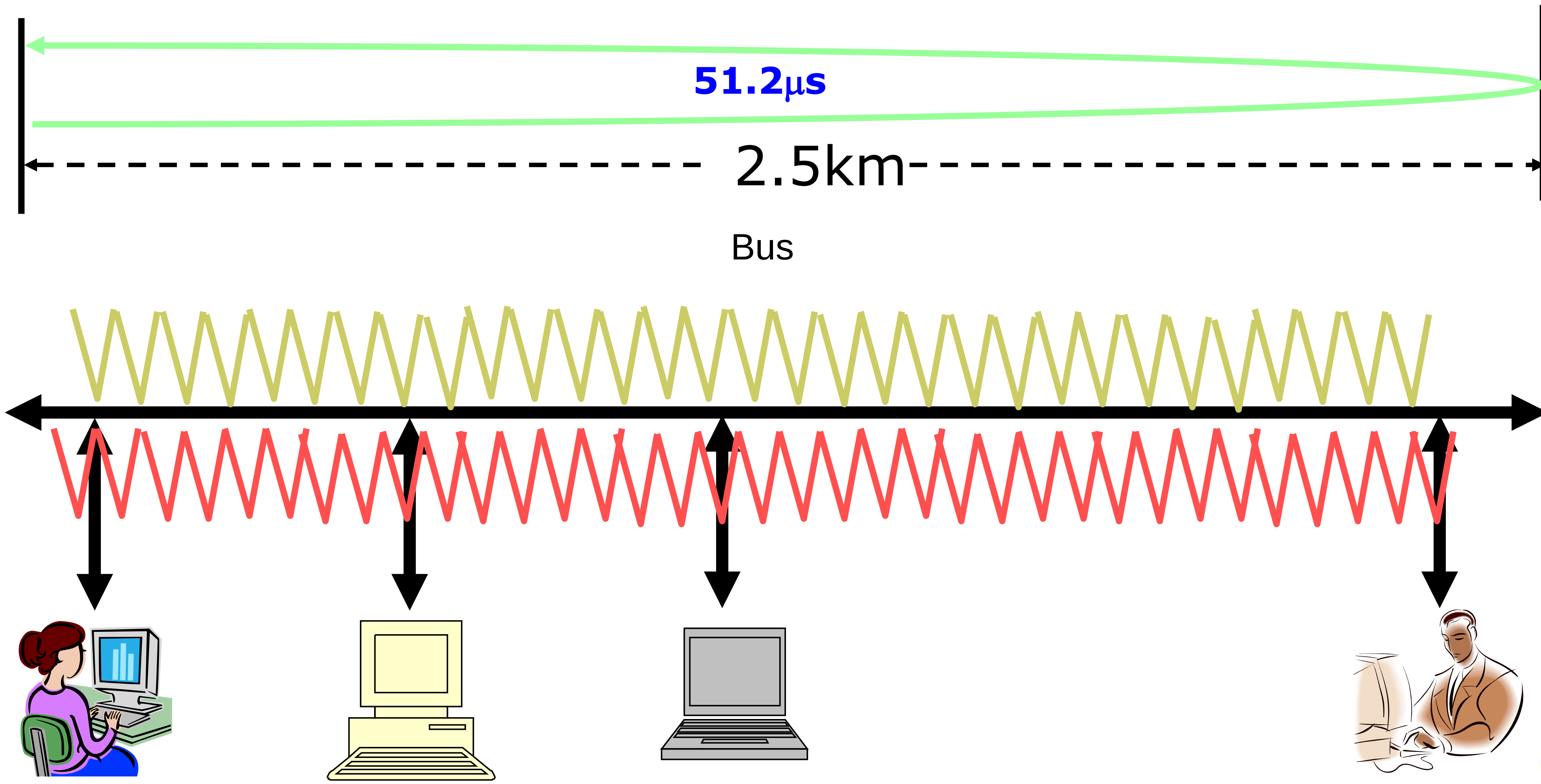
Protocols

- CSMA/CD (Carrier Sense, Multiple Access with Collision Detection)
  - For wired communication.
  - Used in Ethernet
  - Silent bus provides right to introduce new message
  - Retry after collision detection.
- CSMA/CA (Carrier Sense, Multiple Access with Collision Avoidance)

Network solutions

Ethernet bus arbitration algorithm (IEEE 802.3)

- Optimistic – why pessimistic ?
  - Use it and withdraw if bad things happen.
- Collision detection → bad things

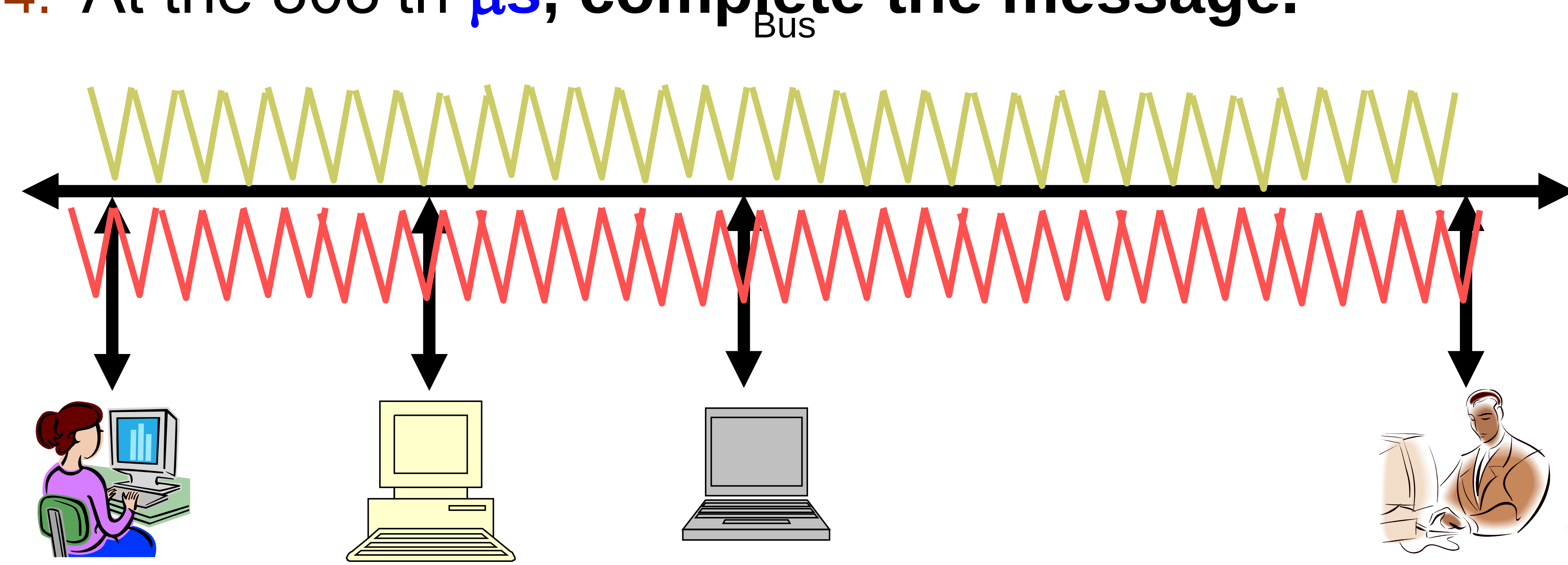


Network solutions

Ethernet bus arbitration algorithm IEEE 802.3

Ethernet bus arbitration algorithm

1. If there is some signals in the bus, then stop and try later.
2. Start sending the message and monitoring the bus.
3. If in 52μs **the message is corrupted, then stop and try later.**
4. At the 808'th μs**, complete the message.**



Database Solutions

Atomic Transactions

- System Model
- Log-based Recovery
- Checkpoints
- Concurrent Atomic Transactions

Database Solutions

System Model

- Assures that operations happen as a single logical unit of work, in its entirety, or not at all
- Related to field of database systems
- Challenge is assuring atomicity despite computer system failures

Database Solutions

System Model

- Transaction - collection of instructions or operations that performs single logical function
  - Here we are concerned with changes to stable storage – disk
  - Transaction is series of read and write operations
  - Terminated by commit (transaction successful) or abort (transaction failed) operation
  - Aborted transaction must be rolled back to undo any changes it performed

Database Solutions

Types of Storage Media

- Volatile storage – information stored here does not survive system crashes
  - Example: main memory, cache
- Nonvolatile storage – Information usually survives crashes
  - Example: disk and tape
- Stable storage – Information never lost
  - Not actually possible, so approximated via replication or RAID to devices with independent failure modes

Goal is to assure transaction atomicity where failures cause loss of information on volatile storage

Log-Based Recovery

- Record to stable storage information about all modifications by a transaction
- Most common is write-ahead logging
  - Log on stable storage, each log record describes single transaction write operation, including
    - Transaction name
    - Data item name
    - Old value
    - New value
  - <T_i starts> written to log when transaction T_i starts
  - <T_i commits> written when T_i commits
- Log entry must reach stable storage before operation on data occurs

Log-Based Recovery Algorithm

- Using the log, system can handle any volatile memory errors
  - Undo(T_i) restores value of all data updated by T_i
  - Redo(T_i) sets values of all data in transaction T_i to new values
- Undo(T_i) and redo(T_i) must be idempotent
  - Multiple executions must have the same result as one execution
- If system fails, restore state of all updated data via log
  - If log contains <T_i starts> without <T_i commits>, undo(T_i)
  - If log contains <T_i starts> and <T_i commits>, redo(T_i)

Checkpoints

- Log could become long, and recovery could take long
- Checkpoints shorten log and recovery time.
- Checkpoint scheme:
  1. Output all log records currently in volatile storage to stable storage
  2. Output all modified data from volatile to stable storage
  3. Output a log record <checkpoint> to the log on stable storage
- Now recovery only includes Ti, such that Ti started executing before the most recent checkpoint, and all transactions after Ti All other transactions already on stable storage

Failure Recovery

A Way to Achieve Atomicity

- Failures of Volatile and Nonvolatile Storages!
  - Volatile Storage: Memory and Cache
  - Nonvolatile Storage: Disks, Magnetic Tape, etc.
  - Stable Storage: Storage which never fail.
- Log-Based Recovery
  - Write-Ahead Logging
    - Log Records

< Ti starts >

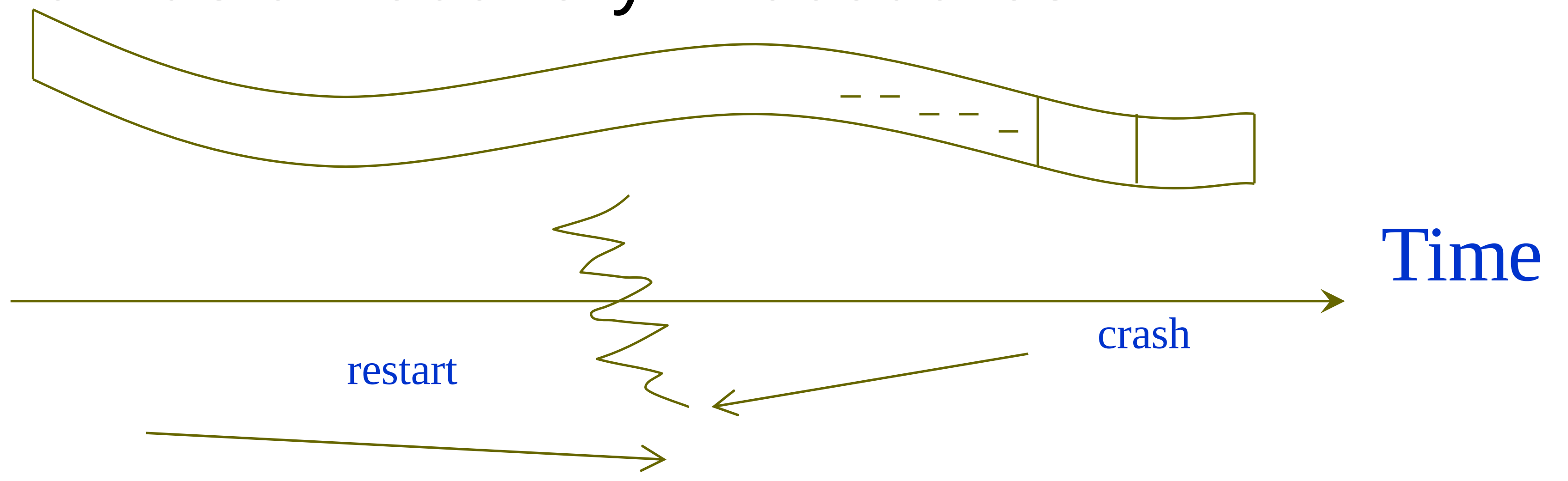
< Ti commits >

< Ti aborts >

< Ti, Data-Item-Name, Old-Value, New-Value>

Failure Recovery

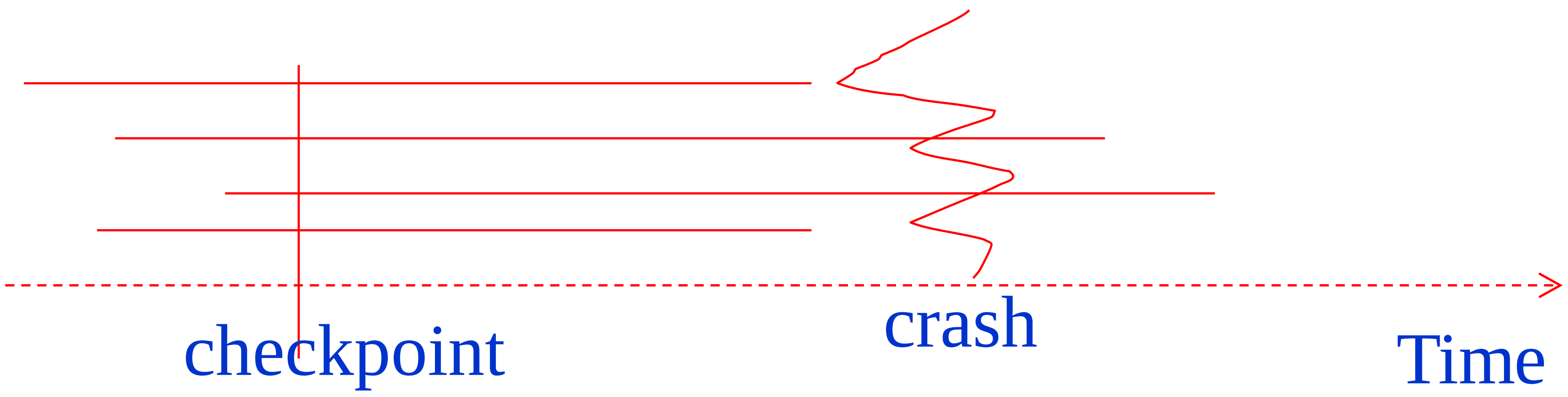
- Two Basic Recovery Procedures:



  - undo(Ti): restore data updated by Ti
  - redo(Ti): reset data updated by Ti
- Operations must be idempotent!
- Recover the system when a failure occurs:
  - “Redo” committed transactions, and “undo” aborted transactions.

Failure Recovery

- Why Checkpointing?
  - The needs to scan and rerun all log entries to redo committed transactions.
- CheckPoint
  - Output all log records, Output DB, and Write ⟨check point⟩ to stable storage!
  - Commit: A Force Write Procedure



Concurrent Transactions

- Must be equivalent to serial execution – serializability
- Could perform all transactions in critical section
  - Inefficient, too restrictive
- Concurrency-control algorithms provide serializability

Serializability

- Consider two data items A and B
- Consider Transactions T_0 and T_1
- Execute T_0, T_1 atomically
- Execution sequence called schedule
- Atomically executed transaction order called serial schedule
- For N transactions, there are N! valid serial schedules

Schedule 1: T_0 then T_1

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Nonserial Schedule

- Nonserial schedule allows overlapped execute
  - Resulting execution not necessarily incorrect
- Consider schedule S, operations O_i, O_j
  - Conflict if access same data item, with at least one write
- If O_i, O_j consecutive and operations of different transactions & O_i and O_j don't conflict
  - Then S' with swapped order O_j O_i equivalent to S
- If S can become S' via swapping nonconflicting operations
  - S is conflict serializable

Schedule 2: Concurrent Serializable Schedule

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule 2: Concurrent Serializable Schedule

- Conflict Serializable:
  - *S* is conflict serializable if *S* can be transformed into a serial schedule by swapping nonconflicting operations.

T0	T1
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

→

T0	T1
R(A)	
W(A)	
	R(A)
R(B)	
	W(A)
W(B)	
	R(B)
	W(B)

→

T0	T1
R(A)	
W(A)	
R(B)	
	R(A)
	W(A)
W(B)	
	R(B)
	W(B)

→

T0	T1
R(A)	
W(A)	
R(B)	
	R(A)
W(B)	
	W(A)
	R(B)
	W(B)

Schedule 3: Non-Serializable Schedule

3. Not serializable

T0	T1
Read(A)	
Write(A)	
	Read(B)
	Write(B)
Read(B)	
Write(B)	
	Read(A)
	Write(A)

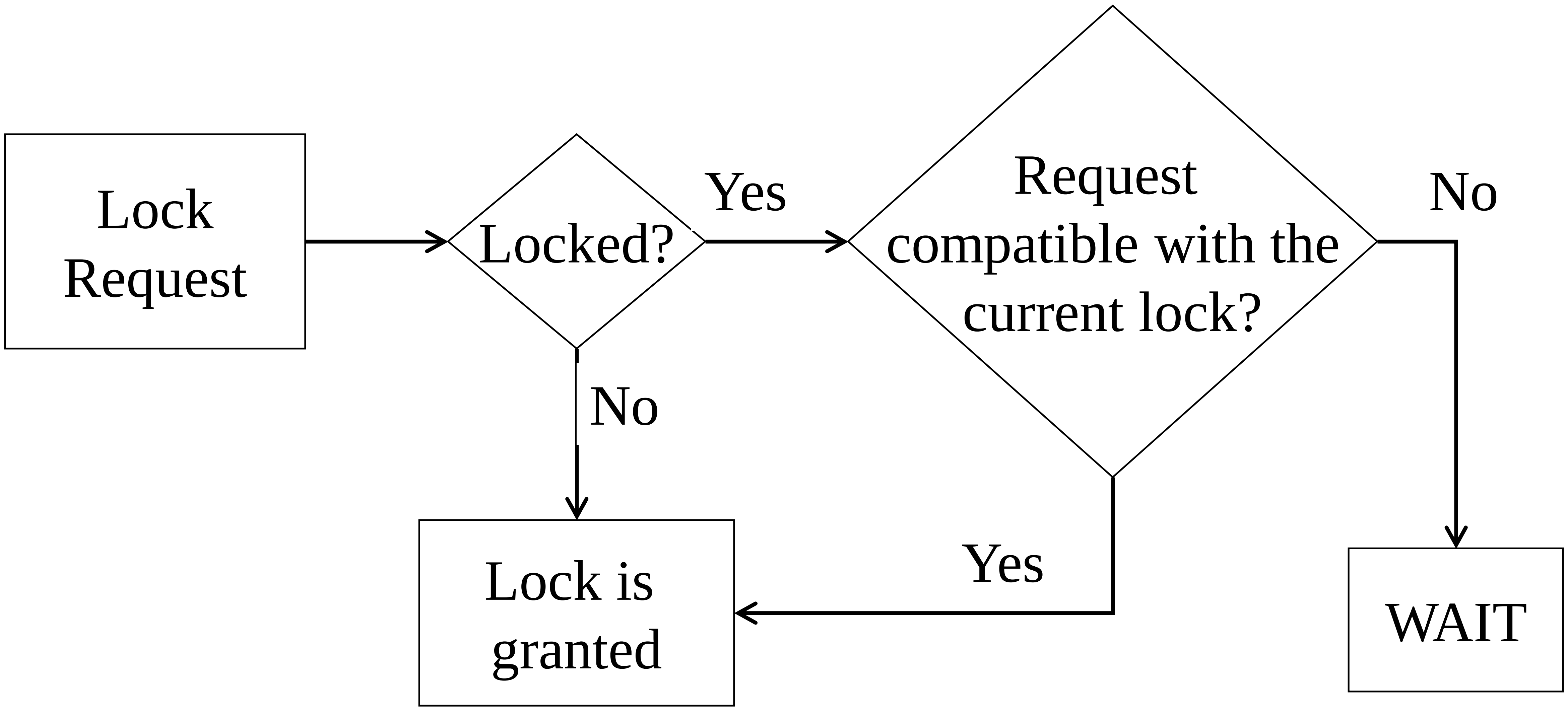
Two operations O_i & O_j conflict if

1. Access the same object
2. One of them is write

Locking Protocol

- Ensure serializability by associating lock with each data item
  - Follow locking protocol for access control
- Locks
  - Shared – T_i has shared-mode lock (S) on item Q, T_i can read Q but not write Q
  - Exclusive – Ti has exclusive-mode lock (X) on Q, T_i can read and write Q
- Require every transaction on item Q acquire appropriate lock
- If lock already held, new request may have to wait
  - Similar to readers-writers algorithm

Locking Protocol



Two-phase Locking Protocol

- Generally ensures conflict serializability
- Each transaction issues lock and unlock requests in two phases
  - Growing – obtaining locks
  - Shrinking – releasing locks
- Does not prevent deadlock

Timestamp-based Protocols

- Select order among transactions in advance – timestamp-ordering
- Transaction T_i associated with timestamp $TS(T_i)$ before T_i starts
  - $TS(T_i) < TS(T_j)$ if Ti entered system before T_j
  - TS can be generated from system clock or as logical counter incremented at each entry of transaction
- Timestamps determine serializability order
  - If $TS(T_i) < TS(T_j)$, system must ensure produced schedule equivalent to serial schedule where T_i appears before T_j

Timestamp-based Protocol Implementation

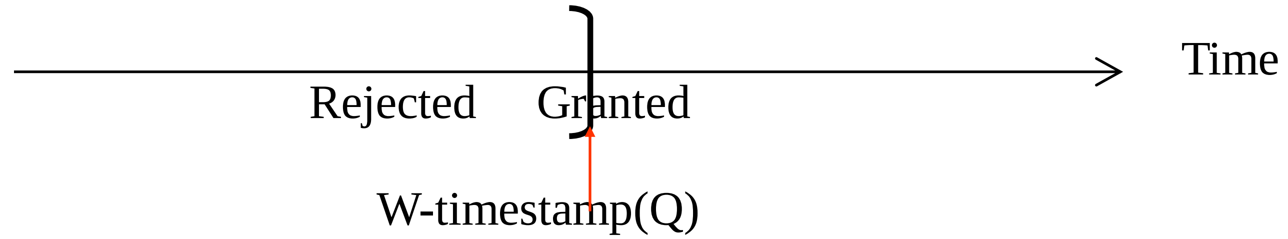
- Data item Q gets two timestamps
  - W-timestamp(Q) – largest timestamp of any transaction that executed write(Q) successfully
  - R-timestamp(Q) – largest timestamp of successful read(Q)
  - Updated whenever read(Q) or write(Q) executed
- Timestamp-ordering protocol assures any conflicting read and write executed in timestamp order
- Suppose Ti executes read(Q)
  - If $TS(T_i) < W\text{-timestamp}(Q)$, Ti needs to read value of Q that was already overwritten
    - read operation rejected and T_i rolled back
  - If $TS(T_i) \geq W\text{-timestamp}(Q)$
    - read executed, R-timestamp(Q) set to max(R-timestamp(Q), $TS(T_i)$)

Timestamp-ordering Protocol

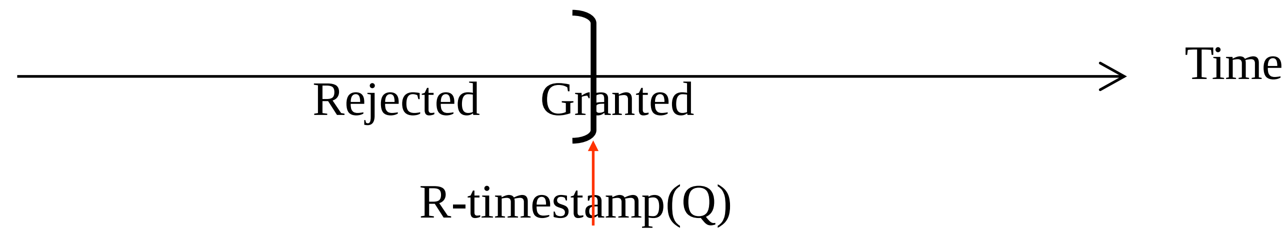
- Suppose Ti executes write(Q)
  - If $TS(T_i)$ < R-timestamp(Q), value Q produced by T_i was needed previously and T_i assumed it would never be produced
    - Write operation rejected, T_i rolled back
  - If $TS(T_i)$ < W-timestamp(Q), T_i attempting to write obsolete value of Q
    - Write operation rejected and T_i rolled back
  - Otherwise, write executed
- Any rolled back transaction T_i is assigned new timestamp and restarted
- Algorithm ensures conflict serializability and freedom from deadlock

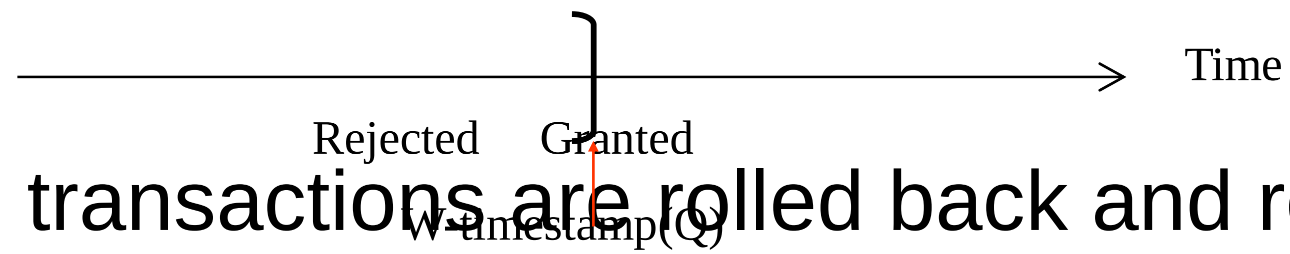
Timestamp-ordering Protocol

- R(Q) requested by T_i → check $TS(T_i)$!



- W(Q) requested by T_i → check $TS(T_i)$!





- Rejected transactions are rolled back and restarted with a new time stamp.

A game of time-stamped protocol

time →

0	1	2	3	4	5	6	7	8	9
						T1	RA		WA
	T2							WB	
		T3		WA	RA				

TS1	TS2	TS3
6	1	2

	Time-Stamp Write	Time-Stamp Read
A	6	6
B	1	
C		

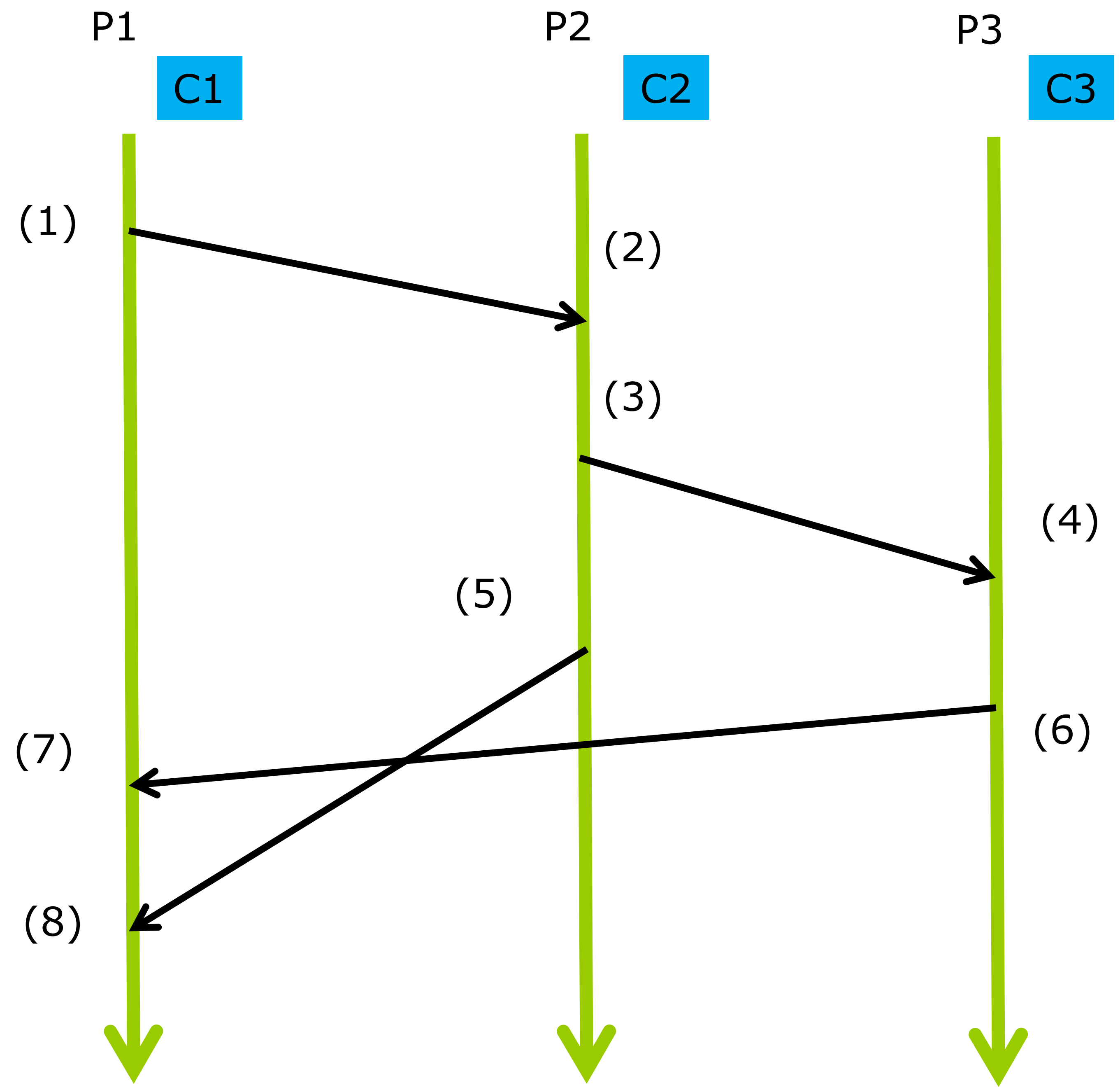
Schedule Possible Under Timestamp Protocol

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)

- Some conflict-serializable schedules are OK with 2-phase locking protocol but not with TT protocol.
- Some conflict-serializable schedules are OK with TT protocol but not with 2-phase locking protocol.

Leslie Lamport’s timestamp

A natural event ordering: (1)→(2)→(3)→(4)→(5)→(6)→(7)→(8)



Timestamps:

must observe the following ordering constraints.

(1)→(7)→(8)

(2)→(3)→5)

(4)→(6)

(1)→(2)

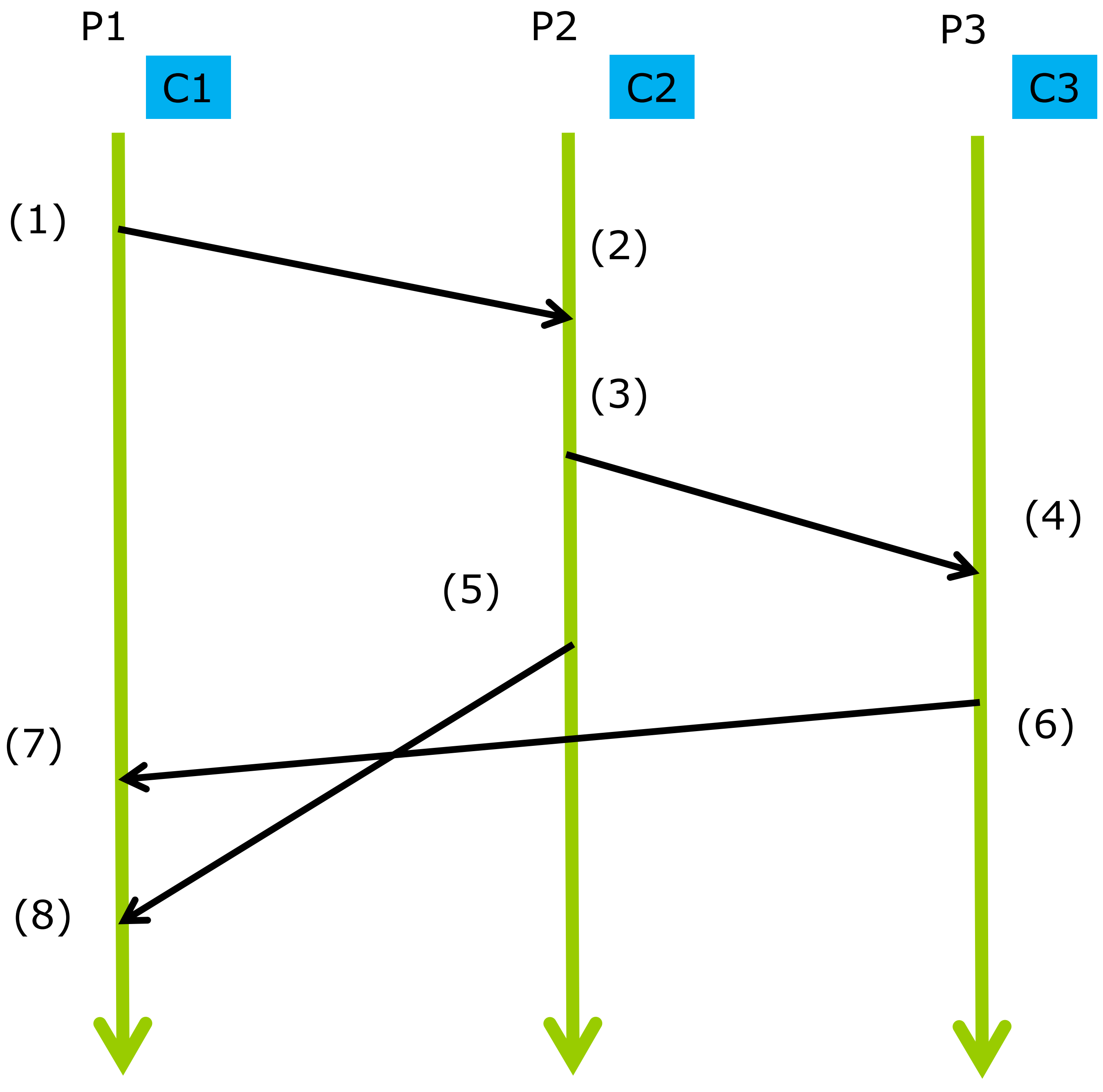
(3)→(4)

(6)→(7)

(5)→(8)

Leslie Lamport's timestamp

A natural event ordering: (1)→(2)→(3)→(4)→(5)→(6)→(7)→(8)



Distributed algorithm for maintaining local clocks:

1. local clock readings ci transmitted with all meesages m.
2. When pj receives (ci,m), let

   cj = max(cj+1,ci+1)

Exercise (1/4)

interested in Erlang and Scala, and in further details about functional languages in general, are encouraged to consult the bibliography at the end of this chapter for additional references.

6.11 Summary

Given a collection of cooperating sequential processes that share data, mutual exclusion must be provided to ensure that a critical section of code is used by only one process or thread at a time. Typically, computer hardware provides several operations that ensure mutual exclusion. However, such hardware-based solutions are too complicated for most developers to use. Mutex locks and semaphores overcome this obstacle. Both tools can be used to solve various synchronization problems and can be implemented efficiently, especially if hardware support for atomic operations is available.

Various synchronization problems (such as the bounded-buffer problem, the readers–writers problem, and the dining-philosophers problem) are important mainly because they are examples of a large class of concurrency-control problems. These problems are used to test nearly every newly proposed synchronization scheme.

The operating system must provide the means to guard against timing errors, and several language constructs have been proposed to deal with these problems. Monitors provide a synchronization mechanism for sharing abstract data types. A condition variable provides a method by which a monitor function can block its execution until it is signaled to continue.

Operating systems also provide support for synchronization. For example, Windows, Linux, and Solaris provide mechanisms such as semaphores, mutex locks, spinlocks, and condition variables to control access to shared data. The Pthreads API provides support for mutex locks and semaphores, as well as condition variables.

Several alternative approaches focus on synchronization for multicore systems. One approach uses transactional memory, which may address synchronization issues using either software or hardware techniques. Another approach uses the compiler extensions offered by OpenMP. Finally, functional programming languages address synchronization issues by disallowing mutability.

Exercises

6.1 Race conditions are possible in many computer systems. Consider a banking system that maintains an account balance with two functions: `deposit(amount)` and `withdraw(amount)`. These two functions are passed the amount that is to be deposited or withdrawn from the bank account balance. Assume that a husband and wife share a bank account. Concurrently, the husband calls the `withdraw()` function and the wife calls `deposit()`. Describe how a race condition is possible and what might be done to prevent the race condition from occurring.


```
do {
   flag[i] = true;

   while (flag[j]) {
      if (turn == j) {
         flag[i] = false;
         while (turn == j)
            ; /* do nothing */
         flag[i] = true;
      }
   }

      /* critical section */

   turn = j;
   flag[i] = false;

      /* remainder section */
} while (true);
```

Figure 6.21 The structure of process P_i in Dekker's algorithm.

✓6.2 The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P_0 and P_1, share the following variables:

```
boolean flag[2]; /* initially false */
int turn;
```

The structure of process P_i (`i == 0` or 1) is shown in Figure 6.21. The other process is P_j (`j == 1` or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.

6.3 The first known correct software solution to the critical-section problem for n processes with a lower bound on waiting of $n - 1$ turns was presented by Eisenberg and McGuire. The processes share the following variables:

```
enum pstate {idle, want_in, in_cs};
pstate flag[n];
int turn;
```

All the elements of `flag` are initially `idle`. The initial value of `turn` is immaterial (between 0 and n − 1). The structure of process P_i is shown in Figure 6.22. Prove that the algorithm satisfies all three requirements for the critical-section problem.

✓6.4 Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user-level programs.

Exercise (2/4)


```
do {
  while (true) {
    flag[i] = want_in;
    j = turn;

    while (j != i) {
      if (flag[j] != idle) {
        j = turn;
      else
        j = (j + 1) % n;
    }

    flag[i] = in_cs;
    j = 0;

    while ( (j < n) && (j == i || flag[j] != in_cs))
      j++;

    if ( (j >= n) && (turn == i || flag[turn] == idle))
      break;
  }

    /* critical section */

  j = (turn + 1) % n;

  while (flag[j] == idle)
    j = (j + 1) % n;

  turn = j;
  flag[i] = idle;

    /* remainder section */
} while (true);
```

Figure 6.22 The structure of process P_i in Eisenberg and McGuire's algorithm.

6.5 Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.

6.6 The Linux kernel has a policy that a process cannot hold a spinlock while attempting to acquire a semaphore. Explain why this policy is in place.

6.7 Describe two kernel data structures in which race conditions are possible. Be sure to include a description of how a race condition can occur.

6.8 Describe how the `compare_and_swap()` instruction can be used to provide mutual exclusion that satisfies the bounded-waiting requirement.

6.9 Consider how to implement a mutex lock using an atomic hardware instruction. Assume that the following structure defining the mutex lock is available:

```
typedef struct {
    int available;
} lock;
```

(`available == 0`) indicates that the lock is available, and a value of 1 indicates that the lock is unavailable. Using this `struct`, illustrate how the following functions can be implemented using the `test_and_set()` and `compare_and_swap()` instructions:

- `void acquire(lock *mutex)`
- `void release(lock *mutex)`

Be sure to include any initialization that may be necessary.

6.10 The implementation of mutex locks provided in Section 6.5 suffers from busy waiting. Describe what changes would be necessary so that a process waiting to acquire a mutex lock would be blocked and placed into a waiting queue until the lock became available.

6.11 Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism—a spinlock or a mutex lock where waiting processes sleep while waiting for the lock to become available:

- The lock is to be held for a short duration.
- The lock is to be held for a long duration.
- A thread may be put to sleep while holding the lock.

6.12 Assume that a context switch takes T time. Suggest an upper bound (in terms of T) for holding a spinlock. If the spinlock is held for any longer, a mutex lock (where waiting threads are put to sleep) is a better alternative.

6.13 A multithreaded web server wishes to keep track of the number of requests it services (known as *hits*). Consider the two following strategies to prevent a race condition on the variable `hits`. The first strategy is to use a basic mutex lock when updating `hits`:

```
int hits;
mutex_lock hit_lock;

hit_lock.acquire();
hits++;
hit_lock.release();
```

Exercise (3/4)

A second strategy is to use an atomic integer:

```
atomic_t hits;
atomic_inc(&hits);
```

Explain which of these two strategies is more efficient.

```
#define MAX_PROCESSES 255
int number_of_processes = 0;

/* the implementation of fork() calls this function */
int allocate_process() {
int new_pid;

   if (number_of_processes == MAX_PROCESSES)
      return -1;
   else {
      /* allocate necessary process resources */
      ++number_of_processes;

      return new_pid;
   }
}

/* the implementation of exit() calls this function */
void release_process() {
   /* release process resources */
   --number_of_processes;
}
```

Figure 6.23 Allocating and releasing processes.

6.14 Consider the code example for allocating and releasing processes shown in Figure 6.23.

a. Identify the race condition(s).

b. Assume you have a mutex lock named `mutex` with the operations `acquire()` and `release()`. Indicate where the locking needs to be placed to prevent the race condition(s).

c. Could we replace the integer variable

```
int number_of_processes = 0
```

with the atomic integer

```
atomic_t number_of_processes = 0
```

to prevent the race condition(s)?

6.15 Servers can be designed to limit the number of open connections. For example, a server may wish to have only *N* socket connections at any point in time. As soon as *N* connections are made, the server will not accept another incoming connection until an existing connection is released. Explain how semaphores can be used by a server to limit the number of concurrent connections.

6.16 Windows Vista provides a lightweight synchronization tool called **slim reader–writer** locks. Whereas most implementations of reader–writer locks favor either readers or writers, or perhaps order waiting threads using a FIFO policy, slim reader–writer locks favor neither readers nor writers, nor are waiting threads ordered in a FIFO queue. Explain the benefits of providing such a synchronization tool.

6.17 Show how to implement the `wait()` and `signal()` semaphore operations in multiprocessor environments using the `test_and_set()` instruction. The solution should exhibit minimal busy waiting.

6.18 Exercise 4.21 requires the parent thread to wait for the child thread to finish its execution before printing out the computed values. If we let the parent thread access the Fibonacci numbers as soon as they have been computed by the child thread—rather than waiting for the child thread to terminate—what changes would be necessary to the solution for this exercise? Implement your modified solution.

6.19 Demonstrate that monitors and semaphores are equivalent insofar as they can be used to implement solutions to the same types of synchronization problems.

6.20 Design an algorithm for a bounded-buffer monitor in which the buffers (portions) are embedded within the monitor itself.

6.21 The strict mutual exclusion within a monitor makes the bounded-buffer monitor of Exercise 6.20 mainly suitable for small portions.

a. Explain why this is true.

b. Design a new scheme that is suitable for larger portions.

6.22 Discuss the tradeoff between fairness and throughput of operations in the readers–writers problem. Propose a method for solving the readers–writers problem without causing starvation.

6.23 How does the `signal()` operation associated with monitors differ from the corresponding operation defined for semaphores?

6.24 Suppose the `signal()` statement can appear only as the last statement in a monitor function. Suggest how the implementation described in Section 6.8 can be simplified in this situation.

6.25 Consider a system consisting of processes $P_1, P_2, \ldots, P_n$, each of which has a unique priority number. Write a monitor that allocates three identical printers to these processes, using the priority numbers for deciding the order of allocation.

6.26 A file is to be shared among different processes, each of which has a unique number. The file can be accessed simultaneously by several processes, subject to the following constraint: the sum of all unique numbers associated with all the processes currently accessing the file must be less than *n*. Write a monitor to coordinate access to the file.

Exercise (4/4)

6.27 When a signal is performed on a condition inside a monitor, the signaling process can either continue its execution or transfer control to the process that is signaled. How would the solution to the preceding exercise differ with these two different ways in which signaling can be performed?

✓ **6.28** Suppose we replace the `wait()` and `signal()` operations of monitors with a single construct `await(B)`, where `B` is a general Boolean expression that causes the process executing it to wait until `B` becomes `true`.

a. Write a monitor using this scheme to implement the readers–writers problem.

b. Explain why, in general, this construct cannot be implemented efficiently.

c. What restrictions need to be put on the `await` statement so that it can be implemented efficiently? (Hint: Restrict the generality of B; see [Kessels (1977)].)

6.29 Design an algorithm for a monitor that implements an *alarm clock* that enables a calling program to delay itself for a specified number of time units (*ticks*). You may assume the existence of a real hardware clock that invokes a function `tick()` in your monitor at regular intervals.

Programming Problems

6.30 Programming Exercise 3.13 required you to design a PID manager that allocated a unique process identifier to each process. Exercise 4.15 required you to modify your solution to Exercise 3.13 by writing a program that created a number of threads that requested and released process identifiers. Now modify your solution to Exercise 4.15 by ensuring that the data structure used to represent the availability of process identifiers is safe from race conditions. Use Pthreads mutex locks, described in Section 6.9.4.

6.31 Assume that a finite number of resources of a single resource type must be managed. Processes may ask for a number of these resources and will return them once finished. As an example, many commercial software packages provide a given number of ***licenses***, indicating the number of applications that may run concurrently. When the application is started, the license count is decremented. When the application is terminated, the license count is incremented. If all licenses are in use, requests to start the application are denied. Such requests will only be granted when an existing license holder terminates the application and a license is returned.

The following program segment is used to manage a finite number of instances of an available resource. The maximum number of resources and the number of available resources are declared as follows:

```
#define MAX_RESOURCES 5
int available_resources = MAX_RESOURCES;
```