CHAPTER 6 - SYNCHRONIZATION

OBJECTIVES

- Introduce the critical-section problem → solutions used to ensure consistency of shared data.
- software and hardware solutions of the critical-section problem.
- examine classical process-synchronization problems.
- explore several tools that are used to solve process synchronization problems.

BACKGROUND

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

THE PROBLEM

Want to provide a solution to the consumer-producer problem that fills all the buffers.

Have integer counter that keeps track of the number of full buffers.

- Initially, counter is set to 0.
- incremented by the producer after it produces a new buffer
- 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
```

INITIALLY COUNT=5

T_0	register1 = counter	register1 = 5
T_1	register1 = register1 + 1	register1 = 6
T_2	register2 = counter	register2 = 5
T_3	register2 = register2 – 1	register2 = 4
T_4	counter = register1	counter = 6
T_5	counter = register2	counter = 4

THE CRITICAL-SECTION PROBLEM

CRITICAL SECTION PROBLEM

Consider system of n processes {p_0, p_1, … p_{n-1} }

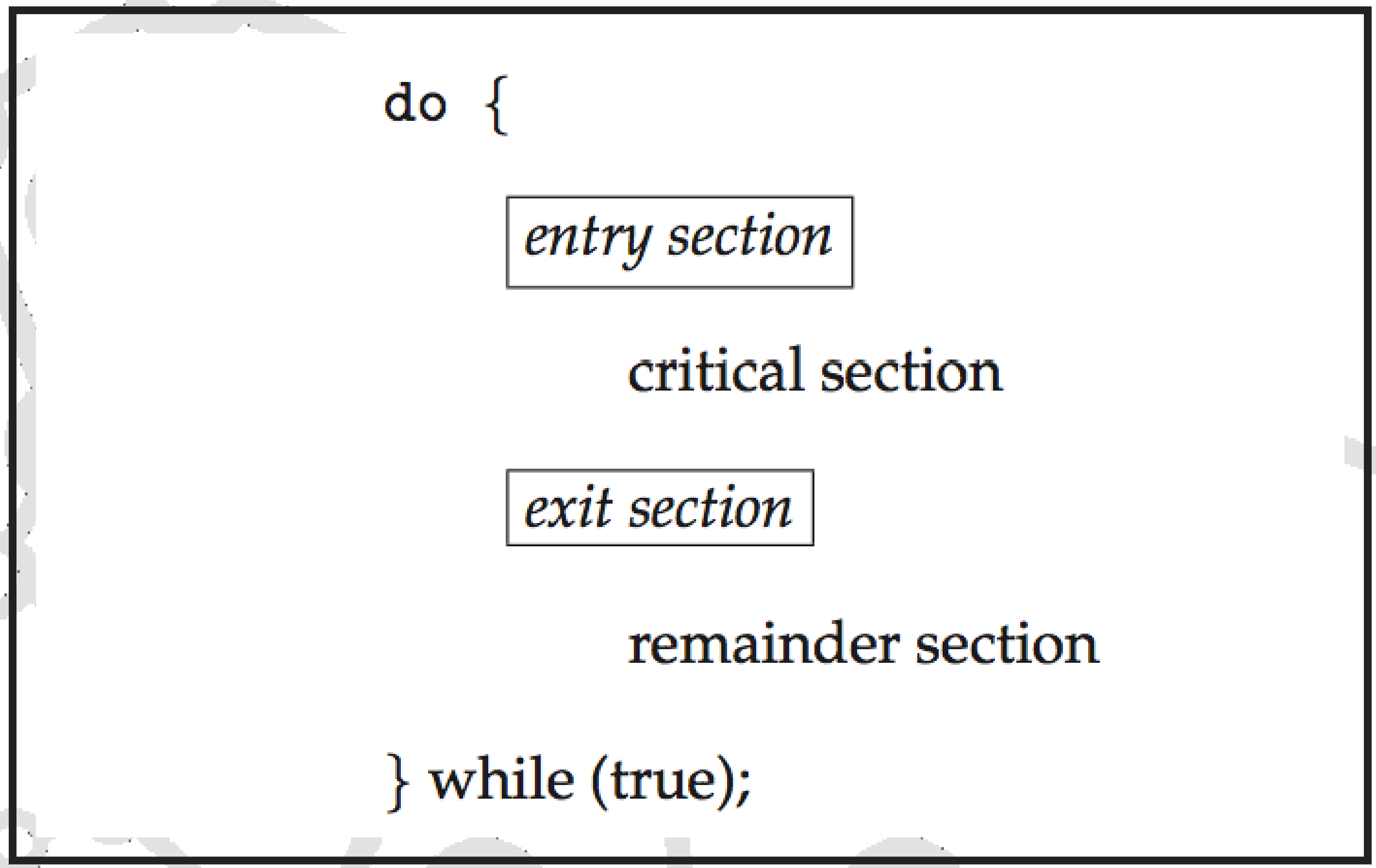
Each process has critical section segment of code

- 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

CRITICAL SECTION

General structure of process p_i is



```
do {

    entry section

        critical section

    exit section

        remainder section

} while (true);
```

SOLUTION TO CRITICAL SECTION PROBLEM

A solution must satisfy 3 requirements

MUTUAL EXCLUSION

If process P_i is executing in its critical section, then no other processes can be executing in their critical sections

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

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

TWO APPROACHES

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

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

PETERSON’S SOLUTION

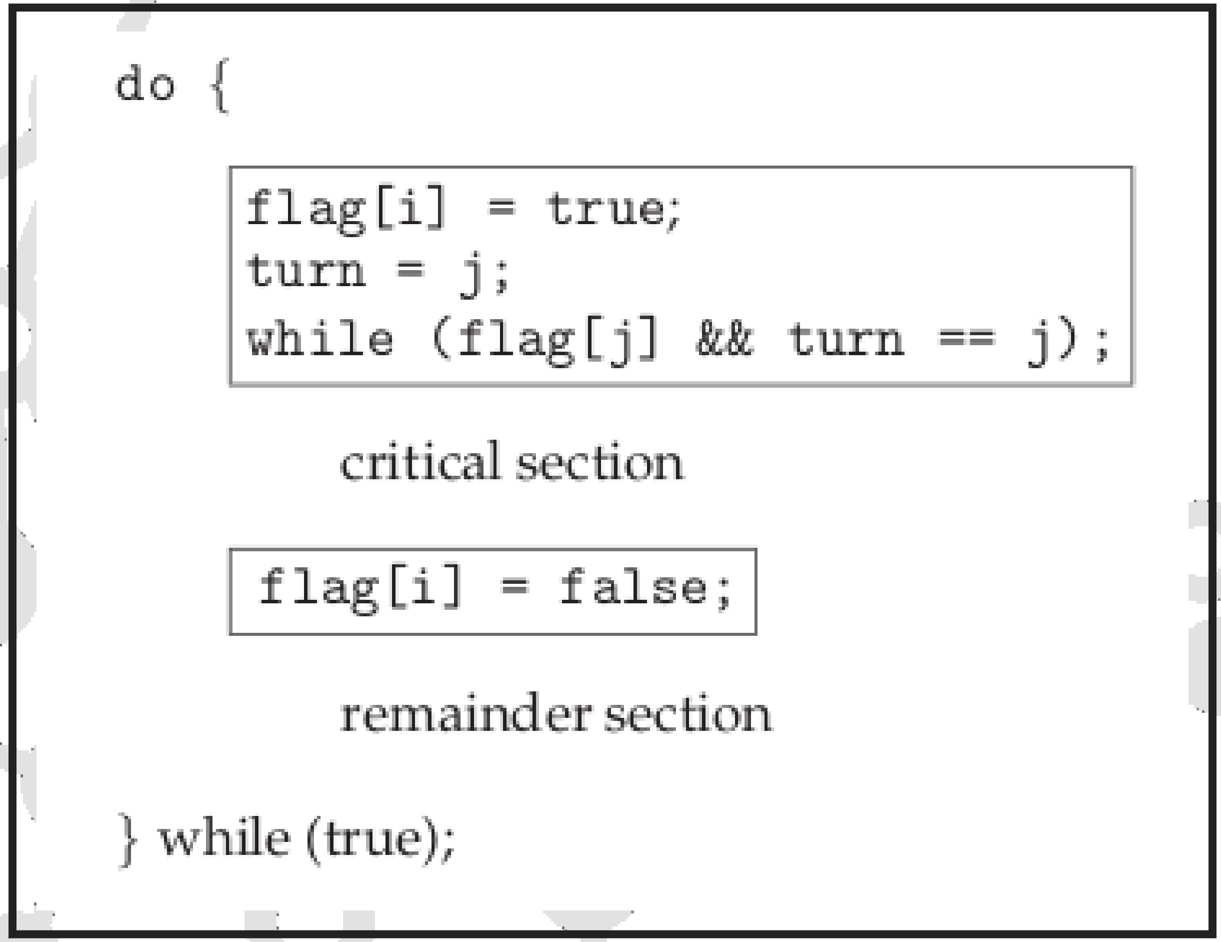
The two processes share two variables:

```
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 = j;
        while (flag[j] && turn == j);

            critical section

        flag[i] = false;

            remainder section

} while (true);
```

Provable that 1), 2) and 3) holds

SYNCHRONIZATION HARDWARE

SYNCHRONIZATION HARDWARE

Many systems provide hardware support for critical section code

All solutions below based on idea of locking

- Protecting critical regions via locks

SYNCHRONIZATION HARDWARE

Uniprocessors – could disable interrupts

- Currently running code would execute without preemption
- Generally too inefficient on multiprocessor systems
  - Operating systems using this not broadly scalable

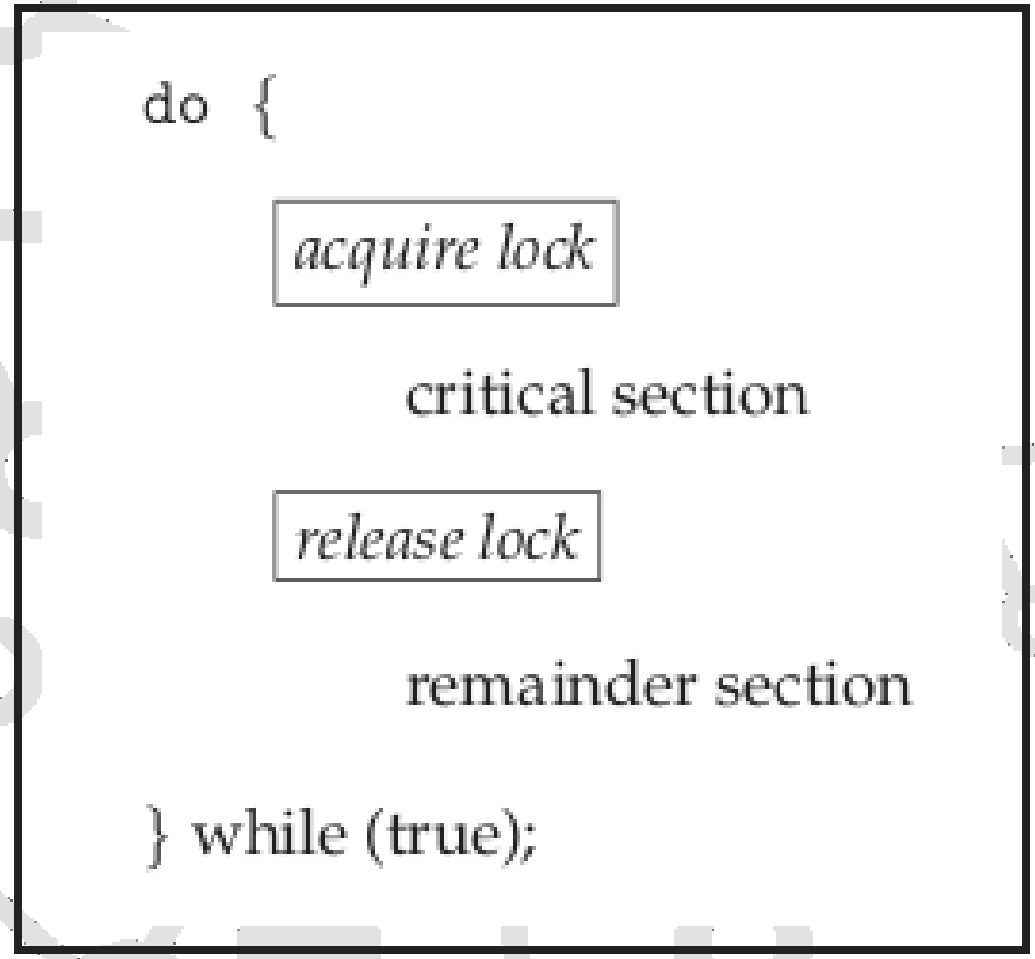
SYNCHRONIZATION HARDWARE

Modern machines provide special atomic hardware instructions

- Atomic = non-interruptible
- Either test memory word and set value
- Or swap contents of two memory words

USING LOCKS

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

```
do {
    while( test_and_set(&lock) ) {
        ; /* DO NOTHING */
    }
    /* Chritical 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

```
do {
    while( compare_and_swap(&lock, 0, 1) != 0 ) {
        ; /* DO NOTHING */
    }

    /* Chritical 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

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

MUTEX LOCKS

Calls to `acquire()` and `release()` must be atomic

Boolean variable indicating if lock is available or not → Usually implemented via hardware atomic instructions

But this solution requires **busy waiting** → This lock therefore called a **spinlock**

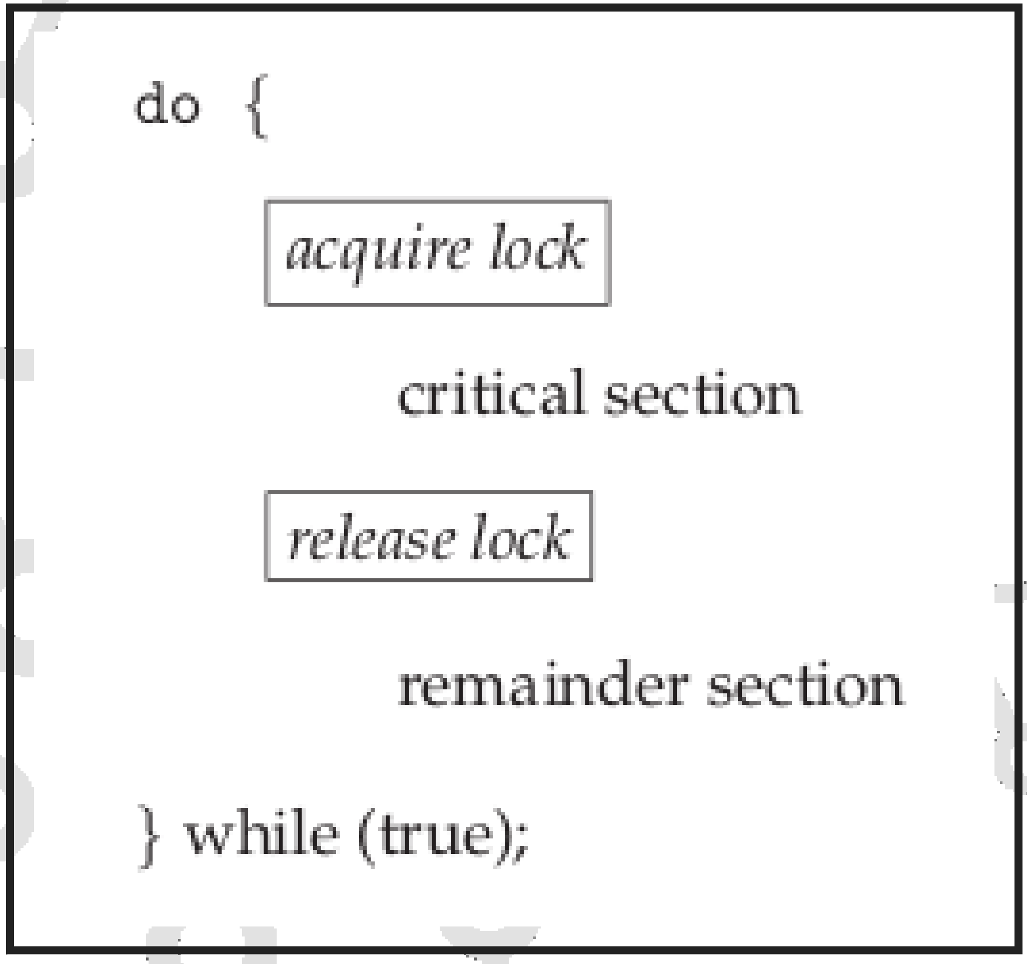
ACQUIRE()

```
acquire() {
    while (!available) {
        ; /* busy wait */
    }
    available = false;;
}
```

RELEASE()

```
release() {
    available = true;
}
```

SOLUTION USING MUTEX



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

SEMAPHORES

SEMAPHORE

Synchronization tool that does not require busy waiting

Semaphore S – integer variable

Two standard operations modify S

```
wait()
signal()
```

Originally called P ( ) and V ( )

SEMAPHORE

Can only be accessed via two indivisible (atomic) operations

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

signal (S) {
    S++;
}
```

SEMAPHORE USAGE

- **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

SEMAPHORE USAGE

Consider P1 and P2 that require S1 to happen before S2

```
P1:
  S1;
  signal(synch);
P2:
  wait(synch);
  S2;
```

SEMAPHORE IMPLEMENTATION

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

SEMAPHORE IMPLEMENTATION

- 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 - 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

SEMAPHORE IMPLEMENTATION - NO BUSY WAITING

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 - NO BUSY WAITING

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

SEMAPHORE IMPLEMENTATION - NO BUSY WAITING

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

SEMAPHORE IMPLEMENTATION - NO BUSY WAITING

```
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

Starvation – indefinite blocking A process may never be removed from the semaphore queue in which it is suspended

DEADLOCK AND STARVATION

Let S and Q be two semaphores initialized to 1

```
P0                  P1
wait(S);            wait(Q);
wait(Q);            wait(S);
...                 ...
signal(S);          signal(Q);
signal(Q);          signal(S);
```

PRIORITY INVERSION

- Assume 3 processes with priority `L < M < H`
- Resource `R` is being held by `L`
- Proces `H` require resource `R`, waits for `L`
- Proces `M` becomes runnable, preempting `L`
- Result: `M` affects how long process `H` must wait for `L`

💡 Let processes borrow higher priority while waiting for resources

PRIORITY INVERSION

Priority Inversion – Scheduling problem when lower-priority process holds a lock needed by higher-priority process

- Solved via **priority-inheritance protocol**

CLASSIC PROBLEMS OF SYNCHRONIZATION

CLASSIC 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

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0;
```

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);
```

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

READERS-WRITERS PROBLEM

- **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

and

```
int read_count = 0;
semaphore rw_mutex = 1;
semaphore mutex = 1;
```

WRITERS PROCESS

```
do {
     wait(rw_mutex);
          . . .
     /* writing is performed */
          . . .
     signal(rw_mutex);
} while (true);
```

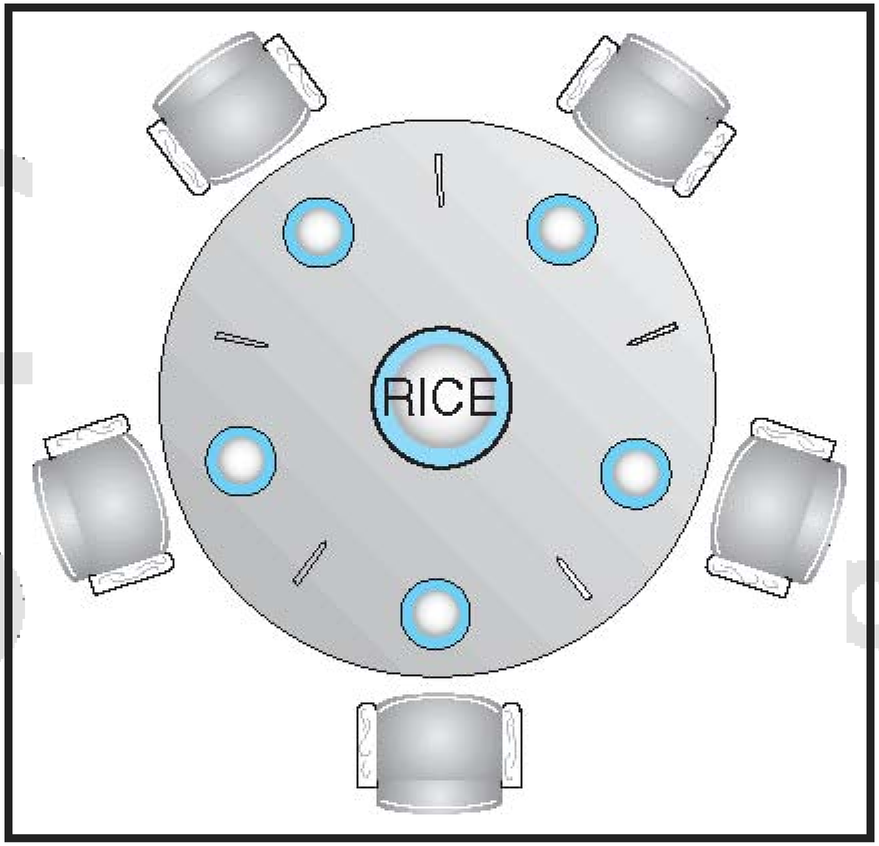
READER PROCESS

```
do {
    wait(mutex);
    read count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);
        . . .
    /* reading is performed */
        . . .
    wait(mutex);
    read count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);
} while (true);
```

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



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

DINING-PHILOSOPHERS PROBLEM

Shared data

- Bowl of rice (data set)
- Semaphore `chopstick[5]` initialized to 1

DINING-PHILOSOPHERS PROBLEM

```
do {
     wait(chopstick[i]);
     wait(chopstick[(i+1) % 5]);
          . . .
     /* eat for awhile */
          . . .
     signal(chopstick[i]);
     signal(chopstick[(i+1) % 5]);
          . . .
     /* think for awhile */
          . . .
} while (true);
```

MONITORS

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

MONITORS

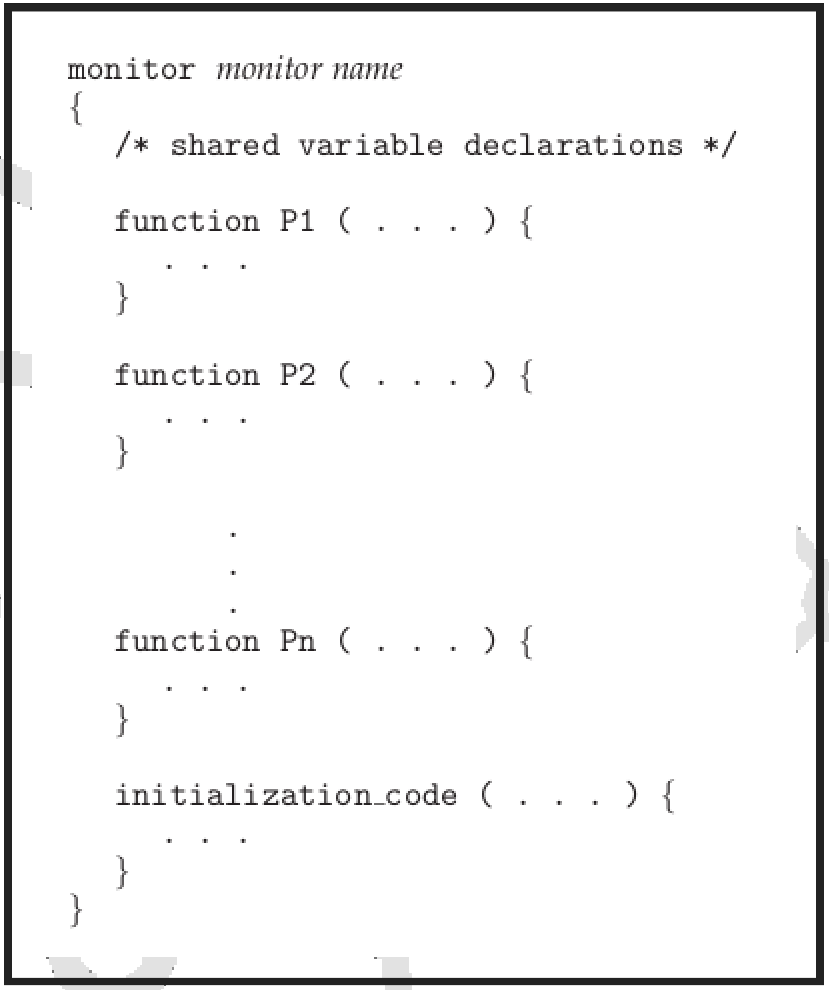
A high-level abstraction that provides a convenient and effective mechanism for process synchronization

Abstract data type, internal variables only accessible by code within the procedure

Only one process may be active within the monitor at a time

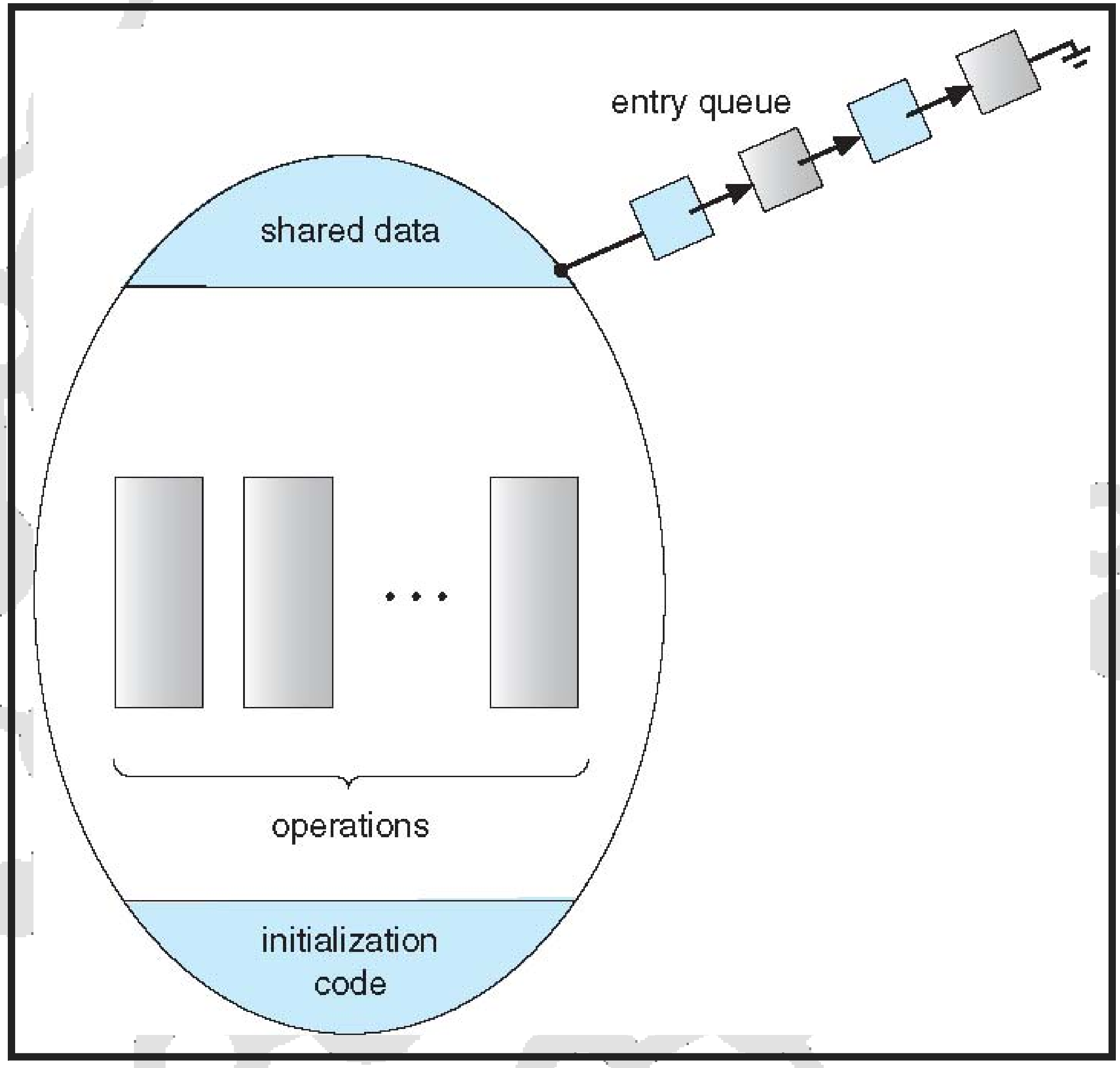
But not powerful enough to model some synchronization schemes

MONITORS



```
monitor monitor name
{
   /* shared variable declarations */

   function P1 ( . . . ) {
      . . .
   }

   function P2 ( . . . ) {
      . . .
   }

          .
          .
          .
   function Pn ( . . . ) {
      . . .
   }

   initialization_code ( . . . ) {
      . . .
   }
}
```

SCHEMATIC VIEW OF A MONITOR

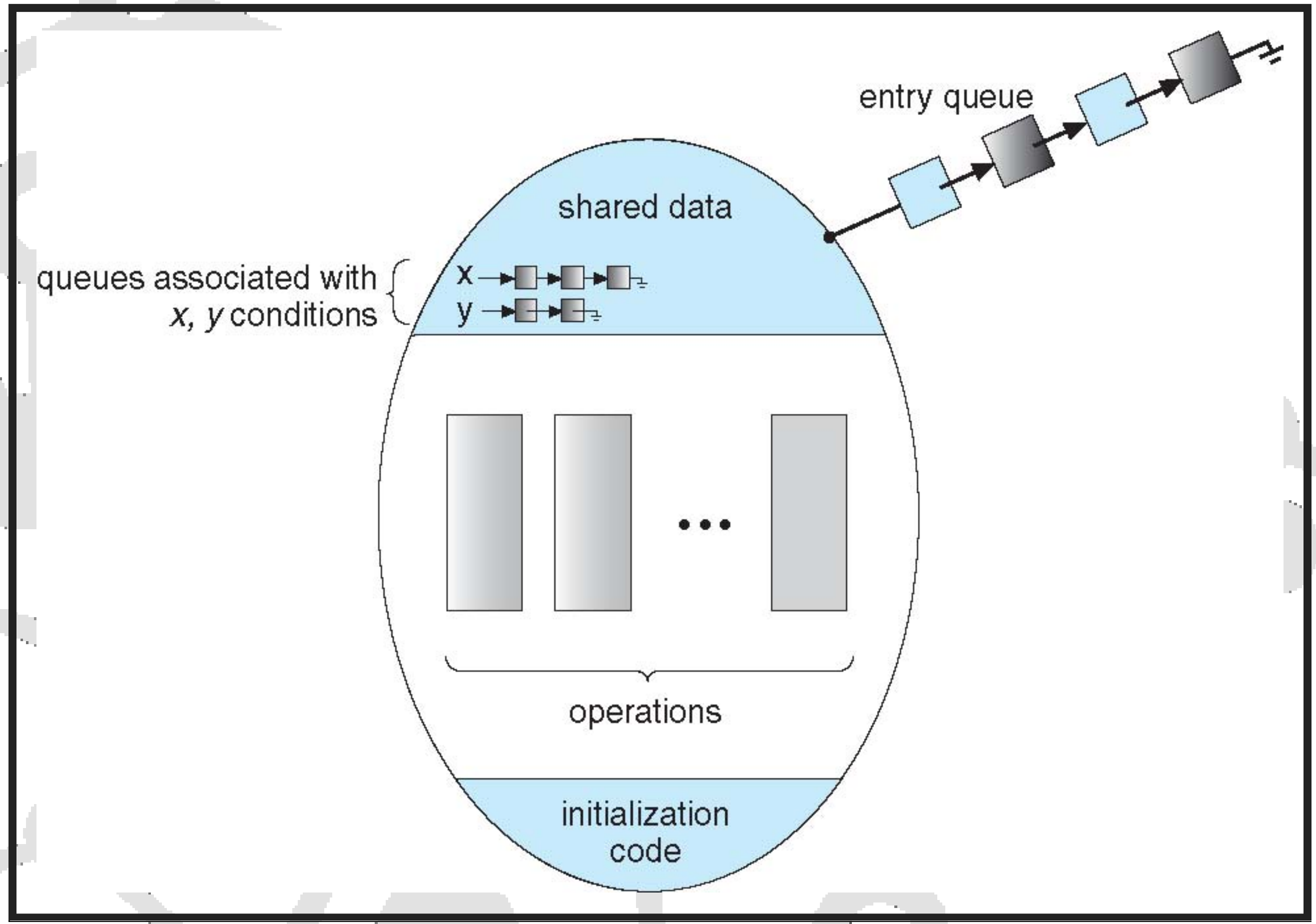


CONDITION VARIABLES

```
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

CONDITION VARIABLES CHOICES

- 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

SOLUTION TO DINING PHILOSOPHERS

```
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((i + 4) % 5);
        test((i + 1) % 5);
    }

    ....
```

SOLUTION TO DINING PHILOSOPHERS

```
        ...

        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;
        }
    }
```

SOLUTION TO DINING PHILOSOPHERS

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

MONITOR IMPLEMENTATION USING SEMAPHORES

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

MONITOR IMPLEMENTATION USING SEMAPHORES

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

MONITOR IMPLEMENTATION – CONDITION VARIABLES

For each condition variable x, we have:

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

MONITOR IMPLEMENTATION – CONDITION VARIABLES

The operation `x.wait` can be implemented as:

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

MONITOR IMPLEMENTATION – CONDITION VARIABLES

The operation `x.signal` can be implemented as:

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

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

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;
    }
}
```

SYNCHRONIZATION EXAMPLES

SYNCHRONIZATION EXAMPLES

- Solaris
- Windows XP
- Linux
- Pthreads

SOLARIS SYNCHRONIZATION

- Implements a variety of locks to support multitasking, multi-threading (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

WINDOWS XP SYNCHRONIZATION

- **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

- 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

ALTERNATIVE APPROACHES

TRANSACTIONAL MEMORY

originated in database theory

A memory transaction is a sequence of memory read –write operations that are atomic. If all operations in a transaction are completed, the memory transaction is committed. Otherwise, the operations must be aborted and rolled back.

EXAMPLE USING LOCKS

```
void update() {
    acquire();
    /* Modify Shared data */
    release();
}
```

TRANSACTIONAL MEMORY

```
void update() {
    atomic {
      /* Modify Shared data */
    }
}
```

TRANSACTIONAL MEMORY

- transactional memory system—not the developer is responsible for guaranteeing atomicity.
- no locks are involved, deadlock is not possible.
- can identify which statements in atomic blocks can be executed concurrently
  - concurrent read access to a shared variable.
- Software or hardware solution

OPENMP

```c
void update(int value) {

    #pragma omp critical
    {
        counter += value;
    }
}
```

FUNCTIONAL PROGRAMMING LANGUAGES

- C, C++, Java, and C# → imperative or procedural languages.
    - program state is mutable, as variables may be assigned different values over time.

FUNCTIONAL PROGRAMMING LANGUAGES

- Erlang, Haskell, Frege and Scala → functional programming languages
  - do not maintain state.
  - once a variable has been defined and assigned a value, its value is immutable
  - no issues like race conditions and deadlocks.

QUESTIONS

BONUS

- Exam question number 4: **Synchronization**