5.3 Peterson's Solution

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

Algorithm for Process P_i

```
do {
    req[i] = true;
    turn = j;
    while (req[j] && turn == j);
    critical section
    req[i] = false;
    remainder section
 } while (true);
```

Algorithm for Process P_i

```
do {
     req[0] = true;
     turn = 1;
     while (req[1] && turn == 1);
     critical section
     req[0] = false;
     remainder section
 } while (true);
```

```
do {
     req[1] = true;
     turn = 0;
     while (req[0] && turn == 0);
     critical section
     req[1] = false;
     remainder section
 } while (true);
```

Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

     P_i enters CS only if:

     either **`reg[j]==false`** or **`turn==i`**
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

- **<u>Note:</u>** The two threads are setting the turn variable. This is ***<u>not</u>*** a read-modify-write and does not result in a critical race condition.

5.4 Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All H/W solutions described in this section are 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 since it requires sending a disable interrupts message to all cores.
    - Operating systems using this approach are 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

Process A

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

Process B

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

test_and_set Instruction

Definition:

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

1. Executed atomically (**it is a single machine instruction**)

it is a single machine instruction

1. Returns the original value of the lock variable (`*target`)
2. Set the new value of lock variable (*target) to "TRUE".

Using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- A possible solution to critical section problem?

```
do {
   /* Wait till lock is false i.e. not locked, then acquire it */
   while (test_and_set(&lock));

   /* critical section */
   . . .
   /* release the lock at the end (i.e. make it false) */
   lock = false;

   /* remainder section */
   . . .

} while (true);
```

fetch_and_add Instruction

Definition:

```
int fetch_and_add (int *target, int inc)
{
     int rv = *target;
     *target = *target + inc;
     return rv:
}
```

1. Executed atomically (**it is a single machine instruction**)
2. Returns the original value of the lock variable (*`target`)
3. Set the new value of (*target) to (*target) + inc.

compare_and_swap Instruction

Definition:

```
int compare_and_swap(int *value, int expected, int new_value) {
    int rv = *value;

    if (*value == expected)
       *value = new_value;
    return rv;
}
```

1. Executed atomically
2. Returns the original value of the lock variable `(*value`)
3. Set the variable "value" the value of the passed parameter "new_value" but only if "*value" =="expected". That is, the swap takes place only under this condition.

Using compare_and_swap

- Shared integer "lock" initialized to 0;
- A possible solution to critical section problem?

```
do {

   /* Wait for value to be zero (i.e. lock is released), then acquire lock */
   while (compare_and_swap(&lock, 0, 1) != 0);


   /* critical section */

   . . .

   /* release the lock when done with CS */

   lock = 0;


   /* remainder section */

   . . .

} while (true);
```

Bounded-waiting Mutual Exclusion with test_and_set

- **Previous H/W algorithms didn't satisfy the bounded wait requirement.**
- This algorithm uses common data structures:
  `bool waiting[n];`
  `bool lock;`
- The variable `Key` is not shared
- Proof of mutual exclusion:
  - P_i can enter its critical section only if either `waiting[i]== false` OR `key==false`.
  - The value of `key` can become false only if `test_and_set()` is executed. The first process to execute it will find `key == false`; all others must wait.
  - The variable waiting[i] can become false only if another process leaves its critical section; only one waiting[i] is set to false, maintaining the mutual-exclusion requirement.

```
do {
  waiting[i] = true;
  key = true;
  while (waiting[i] && key)
     key = test_and_set(&lock);
  waiting[i] = false;

  /* critical section */
  . . .
  /* Select next process to run
  j = (i + 1) % n;
  while ((j != i) && !waiting[j])
     j = (j + 1) % n;

  if (j == i)
     lock = false;
  else
     waiting[j] = false;
  /* remainder section starts below*/
  . . .
} while (true);
```

Bounded-waiting Mutual Exclusion with test_and_set

- Proof of progress:
  - Since a process exiting the critical section either sets `lock` to `false` or sets `waiting[j]` to `false`. Both allow a process that is waiting to enter its critical section to proceed.
- Proof of bounded wait:
  - When a process leaves its critical section, it scans the array waiting in the cyclic ordering ($i + 1$, $i + 2$, ..., $n - 1$, 0, ..., $i - 1$). It designates the first process in this ordering that is in the entry section (`waiting[j] ==true`) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within $n - 1$ turns.

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;

    /* critical section */
    . . .
    /* Select next process to run
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;

    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    /* remainder section starts below*/
    . . .
} while (true);
```

entry section

exit section

5.5 Mutex Locks

- The OS provides abstraction for the hardware tools previously described, particularly since they require some shared lock variables.
- Simplest is **mutex**.
- Usage: Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Lock = Boolean variable indicating if lock is available or not

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

An implementation using atomic acquire() and release()

- May be implemented via hardware atomic instructions such as
  - `test_and_set` or `compare_and_swap`
- This lock sometimes referred to as a **spinlock** because it requires **busy waiting**, thus
  - **NOT EFFICIENT**.
  - When used, the critical section must be very short
- A mutex may also be implemented without a spinlock by using **wait queues**. The method is explained in the next section for semaphores.

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

5.6 Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- **Theoretically,** it can only be accessed via **indivisible (atomic) operations (shown in blue rectangles)** **`wait()`** and **`signal()`** (Originally called **`P()`** and **`V()`**)

These are NOT UNIX wait() and signal() API calls

```
wait(S){
    while(true){   // busy wait till S>0
        if(S > 0){
          S--;
          break;
        }
    }
}
```

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

S>0 (i.e. 1 and above) indicates that the semaphore is not locked

BLUE RECTANGLES indicate atomic operations

Semaphore Usage

- **Counting semaphore usage** – integer value can range over an unrestricted domain
  - May be used to organize usage of a resource that only allows access to N processes at a time -> semaphore needs to be initialized to N.
- **Binary semaphore usage**– integer value can range only between 0 and 1
  - Same as a **mutex lock**, except that it has a different polarity **(initialized to 1)**
  - Can synchronization two processes: Consider two processes P_1 and P_2 that require a statement S_1 to happen before S_2

    Create a semaphore named "**synch**" and **initialize it to 0**

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

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

- **Note that** generally you cannot initialize the semaphore's value to less than zero. See the man page for `sem_init()`.

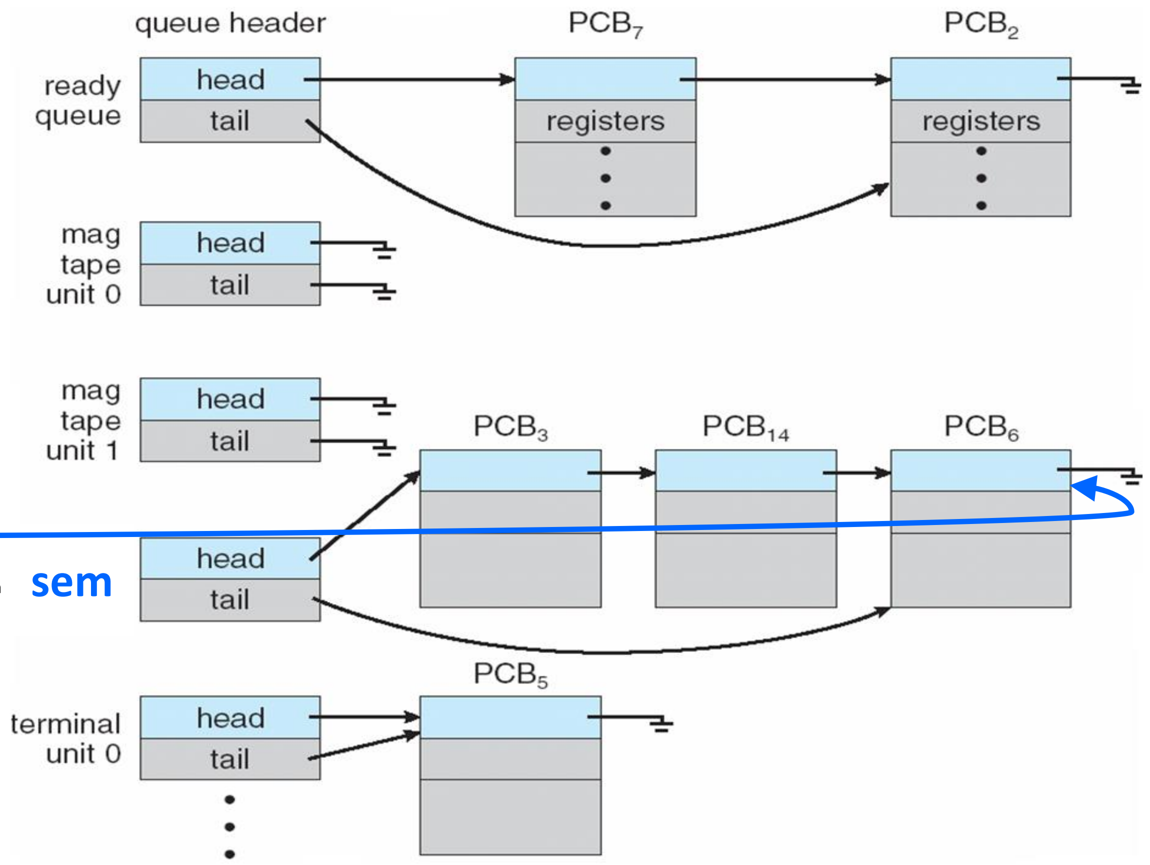
Semaphore Implementation

- Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore **concurrently** (i.e. blue rectangles must be guaranteed to execute **atomically**)
- In similarity to a mutex, processes and threads can be **busy waiting** for the semaphore to become available (i.e. >0)
- A process may thus spend a lot of time in entry sections waiting for the semaphore and not doing any useful work.
- Therefore it may be efficient from the system's point of view to block the process and move it into a waiting queue, and schedule another ready process to run instead.

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each semaphore has two data items:
  - value (of type integer)
  - pointer to first process in the linked-list queue.
- We define 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

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



Implementation with no Busy waiting (Cont.)

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

Reverse order compared to that used in busy-waiting

BLUE RECTANGLES indicate atomic operations

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

Implementation with no Busy waiting (Cont.)

In this implementation, semaphore values are:

- <0: indicates one or more processes are blocked waiting on the semaphore
  - This is different from previous implementation where the value cannot be <0.
- ==0: indicates the semaphore is not available but no process is blocked waiting on it.
- >0: (i.e. 1 and above) indicates the semaphore is available and thus no process is blocked waiting on it.

Unix/Pthreads Synchronization

- Named semaphores use names that start with '/' and are less than 251 characters.

```
sem_open
sem_post
sem_wait
sem_close
sem_unlink
```

- Unnamed (anonymous) semaphores use shared variables (for processes or threads) of type `sem_t`.

```
sem_init
sem_post
sem_wait
sem_destroy
```

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);`

Priority inversion

- **Priority inversion** is a Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Consider having 3 processes L,M and H with low, medium and high priorities respectively, and consider also a resource R that is shared amongst them.
  - If process L acquired R, and then process H requested R, then H will be blocked.
  - If another process M (priority higher than L and is not requesting R) is ready to run, then it may preempt process L (due to the timer tick for example).
  - This indirectly causes priority inversion and it is sometimes problematic.
- Solved via **priority-inheritance protocol**
  - Priority of process L changes to high when H requests the shared resource, and thus won't be preempted by process M.
- This problem occurred on the Mars Pathfinder's robot in 1997 (running a VxWorks real-time OS).

5.7 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

- ***BUF_SZ*** elements inside the shared buffer
- Semaphore **`num_full_el`** initialized to the value 0 – keeps track of the number of elements that are full.
- Semaphore **`num_empty_el`** initialized to the value n – keeps track of number of elements that are empty.

- Why use semaphores, when the previous approach seemed to work fine (i.e. using the in and out indices without using any synchronization primitives)?

Bounded-Buffer Problem

- ***BUF_SZ*** elements inside the shared buffer
- Semaphore **`num_full_el`** initialized to the value 0 – keeps track of the number of elements that are full.
- Semaphore **`num_empty_el`** initialized to the value n – keeps track of number of elements that are empty.

- Why use semaphores, when the previous approach seemed to work fine (i.e. using the in and out indices without using any synchronization primitives)?
  - Because of the busy-waiting problem in which a process or thread may be spending valuable CPU time doing nothing but waiting in a loop!
  - We may still use the in and out variables to index a particular buffer in the pool.
- The full semaphore is used to indicate how many buffers are full, whereas the empty semaphore indicates how many are empty.

Bounded Buffer Problem (Cont.)

- The structure of the **producer** process

```
do {
   /* produce an item in next_produced
  */
   ...
   /* dec empty sem. */
   wait(num_empty_el);
   /* write/produce to an entry*/
   buffer[in] = next_produced;
   in = (in + 1) % BUF_SZ;
   /* inc full sem. */
   signal(num_full_el);
} while (true);
```

- The structure of the **consumer** process

```
do {
   /* dec full sem. */
   wait(num_full_el);
   /* read/consume an entry */
   next_consumed = buffer[out];
   out = (out + 1) % BUF_SZ;
   /*inc empty sem.*/
   signal(num_empty_el);
         ...
   /* consume the item in next consumed */
   ...
} while (true);
```

Readers-Writers Problem

- A data set (e.g. a database) is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can read and write
- Problem – allow multiple readers to read at the same time
  - Only one writer can access the shared data at the same time (i.e. no other writers or other readers are allowed)
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1 (to protect access to `read_count`)
  - 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);
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

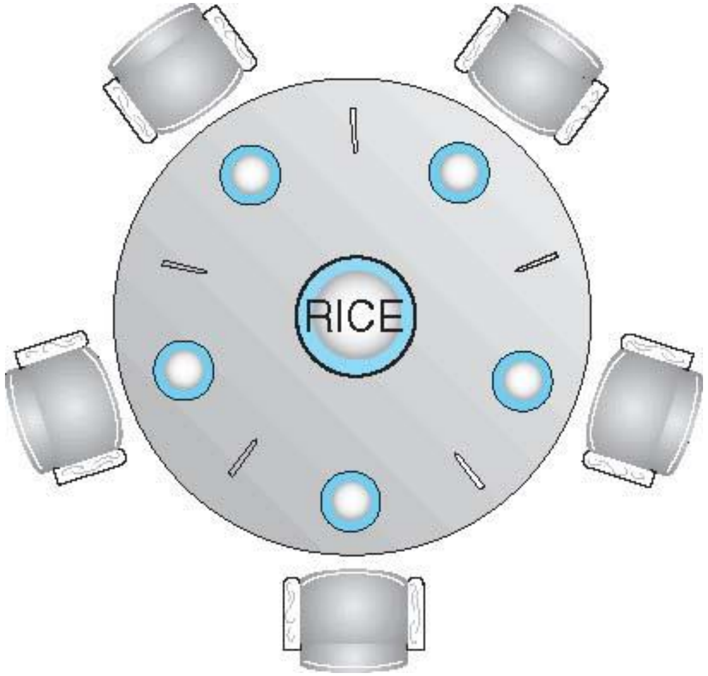
```
do {
     wait(mutex);
     read_count++;
     if (read_count == 1) /* only first reader locks rw_mutex */
           wait(rw_mutex);
     signal(mutex);
           ...
     /* reading dataset is performed, protected by rw_mutex */
     /* either one writer, or multiple readers at a time
           ...
     wait(mutex);
     read count--;
     if (read_count == 0) /* only last reader unlocks rw_mutex */
           signal(rw_mutex);
     signal(mutex);
} 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 the write ASAP (i.e. no readers are allowed to read till after the writer gets and is done with his access)
- Both may have **starvation** leading to even more variations
- In some systems, the kernel provides a reader-writer locks

Dining-Philosophers Problem

- Philosophers spend their lives alternating between **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 analogy:
    - Each philosopher is a thread
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1



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);
```

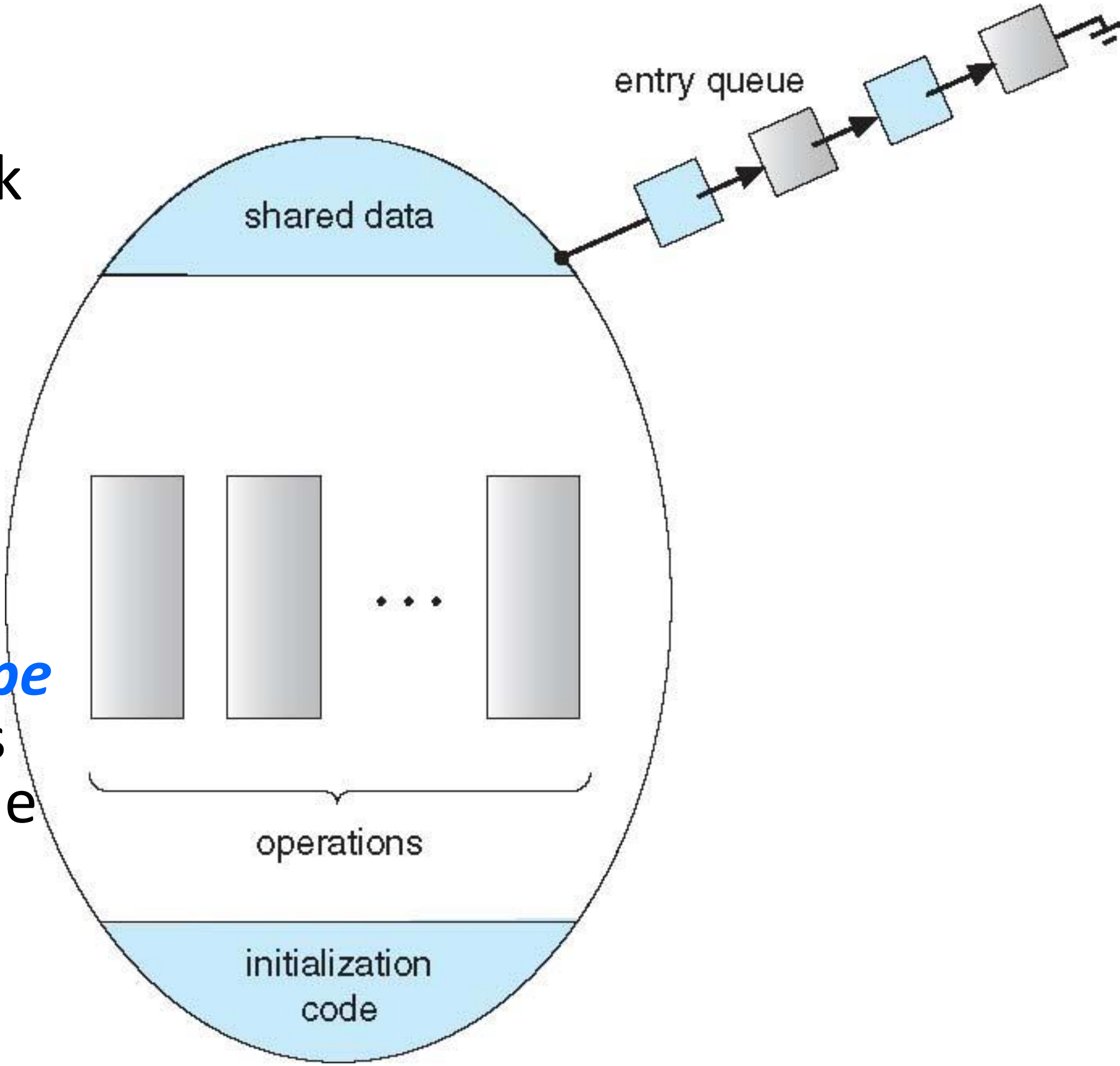
- What is the problem with this algorithm?

Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table of 5 chopsticks.
  - Use an asymmetric solution
    - An odd-numbered philosopher picks up first the left chopstick and then the right chopstick.
    - An even-numbered philosopher picks up first the right chopstick and then the left chopstick.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section) → we may use **monitors** to implement this method.

5.8 Monitors

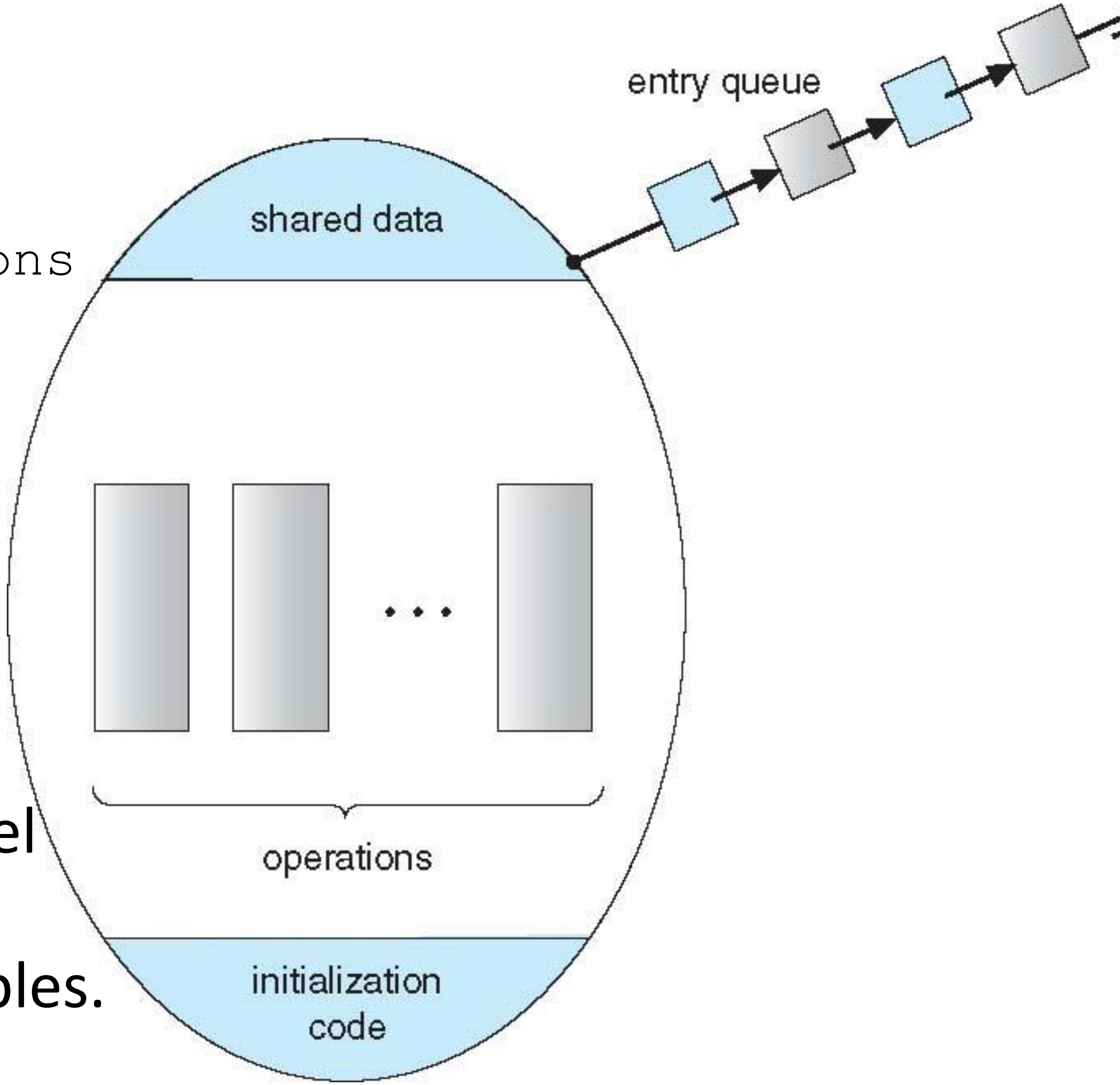
- The dining philosophers deadlock may be solved using monitors.
- A monitor is a high-level abstraction that provides a convenient and effective mechanism for process synchronization
- A monitor is an ***abstract data type*** *(i.e. an **object**)*, internal variables only accessible by code within the procedure
- Only one process may be **active** within the monitor at a time.



5.8 Monitors

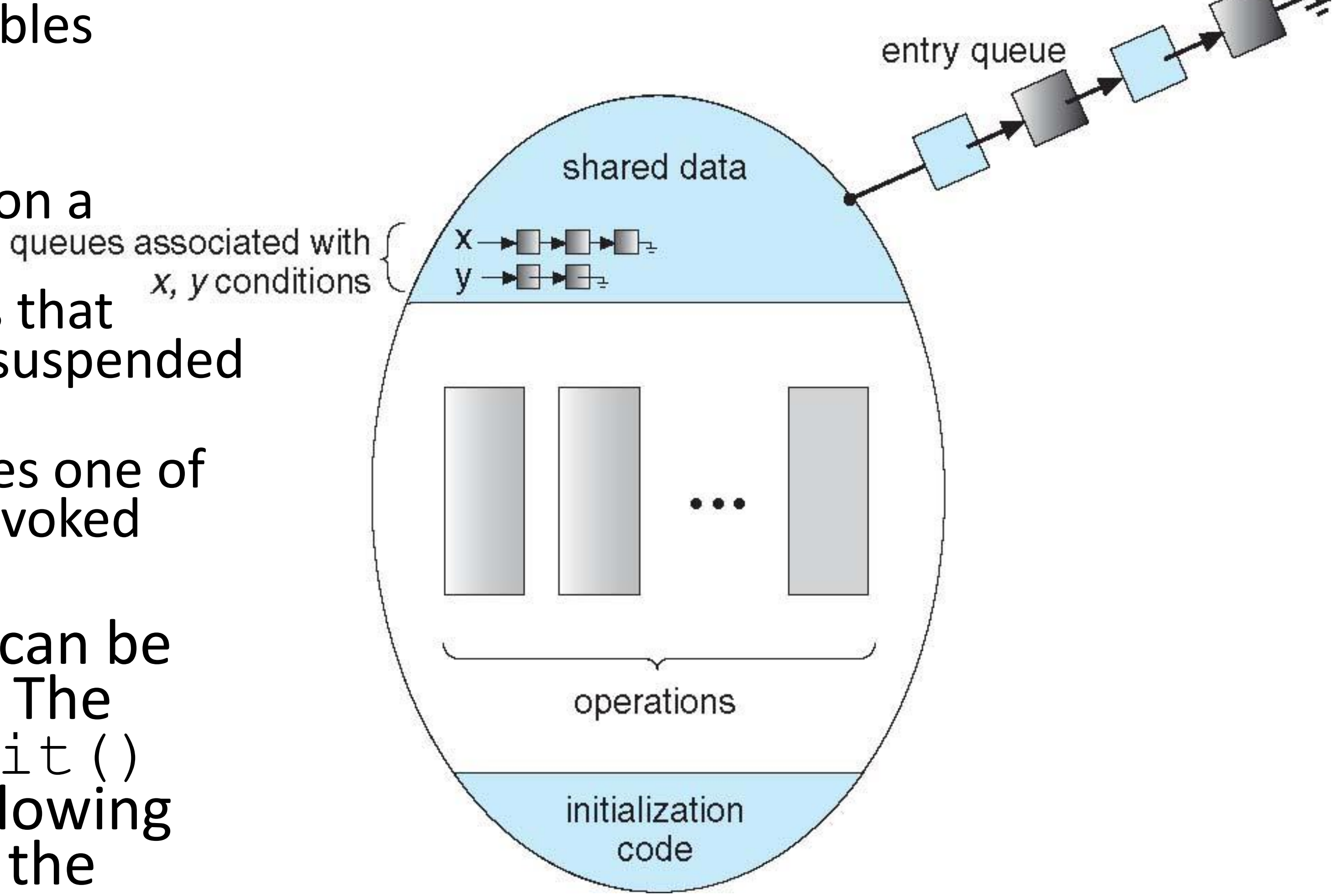
```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure Pn (…) {……}

 Initialization code (…) { … }

}
```

- But not powerful enough to model some synchronization schemes
- Thus we may use condition variables.



Condition Variables

- Condition variables are variables declared inside a monitor:
  `condition x, y;`
- Two operations are allowed 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()`
- Only one thread/process can be active inside the monitor. The process that issues `x.wait()` becomes inactive, thus allowing others to be active inside the monitor.



Condition Variables – cont.

- Contrast this operation with the `signal()` operation associated with semaphores, which always affects the state of the semaphore:
    - Unlike semaphores, if there are no threads/processes that are waiting on the condition variable `x`, then calling `x.signal()` does not affect the condition variable.
        - If you recall calling `signal(my_semaphore)` increments the semaphore whether there is someone waiting on it or not.

Condition Variables Choices

- If process P invokes `x.signal()`, and process Q was suspended in `x.wait()`, what should happen next?
  - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or Q blocks waiting on another condition (i.e. immediate effect).
  - **Signal and continue** – P continues till it leaves the monitor or blocks waiting on another condition, and then Q may become active (i.e. deferred effect).
  - Both have pros and cons – language implementer can decide

Monitor solution to dining philosophers

```
monitor DiningPhilosophers
{
  enum {THINKING, HUNGRY, EATING} state[5] ;
  condition cond[5];

  void pickup (int i) {
         state[i] = HUNGRY;
         test_and_signal(i); /* if successful: my state becomes EATING + signal
                                myself which is wasted cause I am not waiting*/
         if (state[i] != EATING) cond[i].wait(); /* test(i) did not succeed */
  }

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

Monitor solution to Dining Philosophers (Cont.)

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

  initialization_code() {
    for (int i = 0; i < 5; i++)
      state[i] = THINKING;
  }
} /* end of monitor */
```

Monitor solution to dining philosophers (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

Resuming Processes within a Monitor

- If several processes queued on condition x, and x.signal() executed, which should be resumed?
  - First-come-first-serve (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

Synchronization primitives used by the Linux kernel

- Prior to kernel Version 2.6, disables interrupts to implement short critical sections
- Version 2.6 and later, fully preemptive and many synchronization objects may be used within the kernel:
  - Atomic integers
  - Semaphores
  - Spinlocks
  - (with reader-writer version of semaphores and spinlocks).
- On **single-CPU** system, spinlocks replaced by **enabling and disabling kernel preemption**
  - Should be only used for short durations
- **For SMP, spinlocks are the primary tool**
  - Also should be only used for short durations.
  - A kernel thread holding a lock is not preemptable. The kernel uses a variable `preempt_count` to keep track of the number of locks a kernel thread holds.

Pthreads Synchronization

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

Alternative approaches: OpenMP

- OpenMP is a set of compiler directives and API that support parallel progamming.

```
void update(int value)
 {
      #pragma omp critical
      {
            count += value
      }
 }
```

- The code contained within the `#pragma omp critical` directive is treated as a critical section and performed atomically.