Operating Systems
CS F372

Lect 37: Synchronization

BIJU K RAVEENDRAN

General Structure of a Process Pi

Only 2 processes say P0 and P1

General structure of Process Pi will be

```
do{
        ENTRY SECTION
                CRITICAL SECTION
        EXIT SECTION
                REMAINDER SECTION
} while(1);
```

Processes may share some common variable to synchronize there action.

Solution to Critical Section Problem

1. Mutual Exclusion - If process 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

Algorithm 1

- Shared variables:
  - **int turn**;
    initially *turn* = 0
  - *turn* - *i* $\Rightarrow$ P_i can enter its critical section
- Process P_i

```
do {
        while (turn !=i);
        critical section
      turn = j;
        reminder section
}while(1);
```

Algorithm 1

Shared variable int turn (initial value 0)

Process 0	Process 1
do { **while** (*turn* !=0); // do nothing <critical section> *turn = 1*; <reminder section > }while(1);	do { **while** (*turn* !=1); // do nothing <critical section> *turn = 0*; <reminder section > }while(1);

- Satisfies mutual exclusion, but not progress

Algorithm 1

- This solution guarantees mutual exclusion
- Drawback 1: processes must strictly alternate
  - Pace of execution of one process is determined by pace of execution of other processes
- Drawback 2: if one processes fails other process is permanently blocked
- This problem arises due to fact that it stores name of the process that may enter critical section rather than the process state

Algorithm 2

- Shared variables
  - *Boolean flag[2]*;
    initially *flag* [0] = *flag* [1] = *false*.
  - *flag* [*i*] = *true* $\Rightarrow$ P_i ready to enter its critical section
- Process P_i

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

Algorithm 2

Shared variable Boolean flag[2] (flag[0] & flag[1] initial value false)

Process 0	Process 1
do { flag[0]=true; **while** (flag[1]); // do nothing <critical section> flag[0]=false; <reminder section > }while(1);	do { flag[1]=true; **while** (flag[0]); // do nothing <critical section> flag[1]=false; <reminder section > }while(1);

- Satisfies mutual exclusion, but not progress requirement.
- If flag[0]=flag[1]=true → infinite loop

Algorithm 2

- This approach Satisfy mutual exclusion
- This approach may lead to dead lock

What is wrong with this implementation?

- A process sets its state without knowing the state of other. Dead lock occurs because each process can insist on its right to enter critical section
- There is no opportunity to back off from this situation

Peterson's Solution – Algorithm 3

- Not guaranteed to work on modern architectures! (But 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 3

- Combined shared variables of algorithms 1 and 2.
- Process P_i

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

Algorithm 3

Shared variable Boolean flag[2] (flag[0] & flag[1] initial value false) and turn (initial value 0)

Process 0	Process 1
do { flag[0]=true; turn = 1; **while** (flag[1] && turn ==1*)*; // do nothing <critical section> flag[0]=false; <reminder section > }while(1);	do { flag[1]=true; turn = 0; **while** (flag[0] && turn ==0*)*; // do nothing <critical section> flag[1]=false; <reminder section > }while(1);

- Meets all three requirements
- Solves the critical-section problem for two processes.

Peterson's Solution – Algorithm 3

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

     $\mathbf{P_i}$ enters CS only if:

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

Peterson's Solution – Algorithm 3

- Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern architectures.
- Understanding why it will not work is also useful for better understanding race conditions.
- To improve performance, processors and/or compilers may reorder operations that have no dependencies.
- For single-threaded this is ok as the result will always be the same.
- For multithreaded the reordering may produce inconsistent or unexpected results!

Peterson's Solution – Algorithm 3

- Two threads share the data:

```
boolean flag = false;
int x = 0;
```

- Thread 1 performs

```
while (!flag);
print x
```

- Thread 2 performs

```
x = 100;
flag = true
```

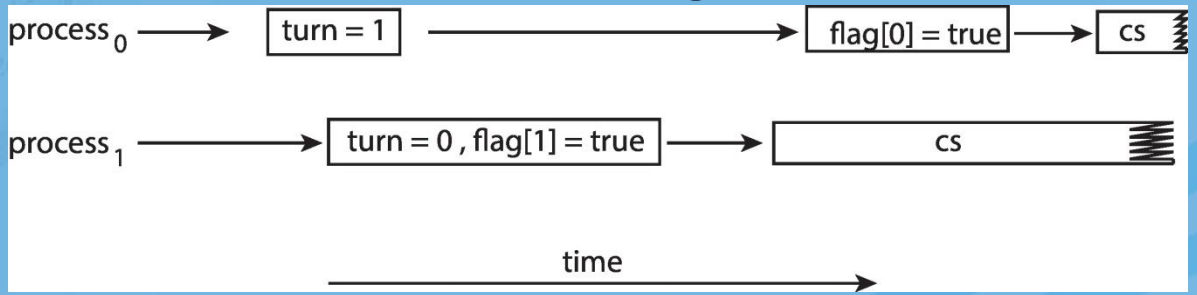
- What is the expected output?

Peterson's Solution – Algorithm 3

- 100 is the expected output.
- However, the operations for Thread 2 may be reordered:

```
flag = true;
x = 100;
```

- If this occurs, the output may be 0!
- The effects of instruction reordering in Peterson's Solution



- This allows both processes to be in their critical section at the same time!

Operating Systems
CS F372

Lect 38: Synchronization

BIJU K RAVEENDRAN

Bakery Algorithm

- Critical section for ‘n’ processes
  - Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
  - If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.
  - The numbering scheme always generates numbers in non-decreasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...

Bakery Algorithm

- Notation
  - lexicographical order(ticket #, process id #)
  - (a,b) < (c,d) if a < c or if a = c and b < d
  - max (a_0,…, a_{n-1}) is a number, k, such that k ≥ a_i for i = 0,…., n – 1
- Shared data

  int *number[n]; //* initialized to 0

Bakery Algorithm

```
do {
    number[i] = max(number[0], number[1], …,
                                    number [n – 1])+1;

    for ( j = 0 ; j < n ; j++ )  {
        while ( number[j] != 0) &&
                ( number [ j , j ] ) < ( number [ i , i ] ) ;
    }

        CRITICAL SECTION

    number[i] = 0 ;

        REMAINDER SECTION

} while(1) ;
```

Synchronization Hardware

- Hardware is faster than the software & can have better efficiency
- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption until it invokes an operating system service or until it is interrupted
  - Disabling interrupts guarantees mutual exclusion
  - Processor is limited in its ability to interleave programs

- Generally too inefficient on multiprocessor systems
- Disabling interrupt in a multiprocessor machine is time consuming, as the message must pass to all the processors. This message passing delays entry into each critical section & system efficiency decreases.
- The system clock functionality may be disturbed, if the clock is kept updated by the interrupt.
- Modern machines provide special atomic (non-interruptable) hardware instructions
  - Performed in a single instruction cycle
  - Access to the memory location is blocked for any other instructions
  - Either test memory word and set value
  - Or swap contents of two memory words

Hardware Instructions

- Special hardware instructions that allow us to either *test-and-modify* the content of a word, or to *swap* the contents of two words atomically (uninterruptibly.)
- Test-and-Set instruction
- Compare-and-Swap instruction

TestAndSet Instruction

- Definition:

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

Solution: Shared boolean variable lock initialized to false.

```
do {
   while (TestAndSet (&lock )) ;// do nothing
   //    critical section
      lock = FALSE;
   //     remainder section
   } while (TRUE);
```

Swap Instruction

- Definition:

```
void Swap (boolean *a, boolean *b)
{   boolean temp = *a;
    *a = *b;
    *b = temp:
}
```

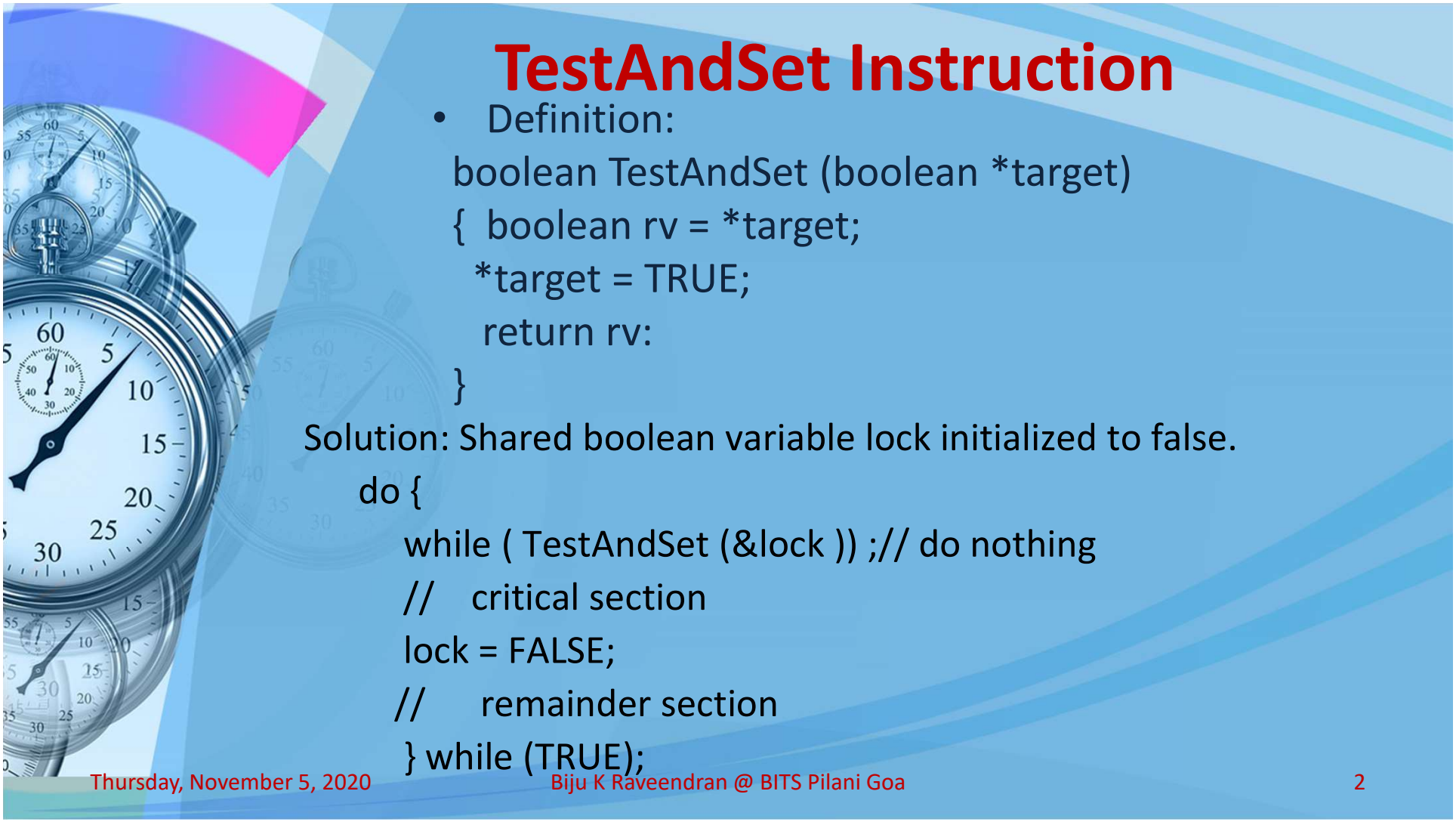
- Shared Boolean variable lock initialized to FALSE;
- Each process has a local Boolean variable key

```
do {   key = TRUE;
       while ( key == TRUE)
              Swap (&lock, &key );
       //    critical section
       lock = FALSE;
       //     remainder section
} while (TRUE);
```

Operating Systems
CS F372

Lect 39: Synchronization

BIJU K RAVEENDRAN



TestAndSet Instruction

- Definition:

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

Solution: Shared boolean variable lock initialized to false.

```
do {
    while ( TestAndSet (&lock )) ;// do nothing
    //    critical section
    lock = FALSE;
    //      remainder section
    } while (TRUE);
```

Swap Instruction

- Definition:

```
void Swap (boolean *a, boolean *b)
{    boolean temp = *a;
      *a = *b;
      *b = temp:
}
```

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

```
do {   key = TRUE;
       while ( key == TRUE)
              Swap (&lock, &key );
       //   critical section
        lock = FALSE;
       //     remainder section
} while (TRUE);
```

Mutual Exclusion Machine Instructions

- Advantages
  - Applicable to any number of processes on either a single processor or multiple processors sharing main memory
  - It is simple and therefore easy to verify
  - It can be used to support multiple critical sections

Mutual Exclusion Machine Instructions

- Disadvantages
  - Busy-waiting consumes processor time
  - Starvation is possible when a process leaves a critical section and more than one process is waiting.
  - Deadlock
    - If a low priority process has the critical region and a higher priority (real-time) process needs it
      - The higher priority process will execute busy waiting in processor which leads to deadlock

Bounded waiting Mutual Exclusion with TestAndSet

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)   key = TestAndSet(&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);
```

Semaphores

- The hardware solution is not easy to generalize for more complex problems.
- Special variable called a semaphore is used for signaling
- Semaphore is a variable that has an integer value
  - Apart from initialization, the integer value is accessed only through 2 standard atomic operations
  - May be initialized to a nonnegative number
  - **Wait** operation decrements the semaphore value
  - **Signal** operation increments semaphore value
  - Originally called P() (Dutch proberen – "to test") and V() (verhogen – "to increment")

Semaphores

```
wait (S) {
        while (S <= 0)
                ; // no-op
          S - - ;
 }


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

```
Semaphore mutex;//initialized to 1
do {
        wait (mutex);
     // Critical Section
   signal (mutex);
        // remainder section
} while (TRUE);
```

Semaphores

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

Implementation

- Disadvantage of the existing mechanism
  - Busy waiting
    - Process is waiting in the CPU without doing any work. This types of semaphore is also called Spin lock (process spins while waiting for the lock)
    - Spin locks may degrade the performance of Uniprocessor – multi programming machines
    - Spin lock may be advantageous in multiprocessor machines (Context switching time can be high than busy waiting time)
    - If locks are expected to be held for a short time then spin locks are useful.

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- wait() and signal() implementation will be a critical section problem
  - 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.

Solution for Busy waiting

- Can modify the definition of wait and signal semaphore operations
- When a process executes wait and finds the semaphore value is not positive, the process can block itself
- The block operation places the process into waiting queue associated with the semaphore, and the state of the process is switch to waiting state
- A process that is blocked, waiting on a semaphore S, would be restarted when some other process executes a signal operation
  - The state of the process changes from waiting to ready

Operating Systems
CS F372

Lect 40: Synchronization

BIJU K RAVEENDRAN

Solution for Busy waiting

- Can modify the definition of wait and signal semaphore operations
- When a process executes wait and finds the semaphore value is not positive, the process can block itself
- The block operation places the process into waiting queue associated with the semaphore, and the state of the process is switch to waiting state
- A process that is blocked, waiting on a semaphore S, would be restarted when some other process executes a signal operation
  - The state of the process changes from waiting to ready

Semaphore with NO busy waiting

- With each semaphore there is an associated waiting queue. Each entry in a semaphore 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 with NO busy waiting

- Each semaphore has an integer value & a list of processes
- When a process must wait on a semaphore, it is added to the list of processes
- A signal operation removes one process from the list and awakens that process

Semaphore with NO busy waiting

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

Implementation of wait

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

Implementation of signal

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

Semaphore with NO Busy waiting

- This implementation may have negative semaphore values.
  - If the semaphore value is negative its magnitude is the number of processes waiting on the semaphore.
- Use FIFO queue to satisfy bounded waiting
  - Semaphore contains both head & tail pointers to the queue.
- The semaphores are executed atomically by
  - disabling interrupts for uniprocessor systems
  - using appropriate hardware for multiprocessor systems

Semaphore implementation using TestAndSet

```
wait ( s ) {
  while (TestAndSet(s.flag));
  s.count - - ;
  if ( s.count < 0) {
    place this process in
          s.queue;
    s.flag=0;
    block this process;
     }
  else
    s.flag = 0;
}
```

```
signal ( s ) {
while (TestAndSet(s.flag));
  s.count + + ;
  if ( s.count <= 0) {
     remove a process P from
           s.queue;
     place process P on ready list;
  }
  s.flag = 0;
}
```

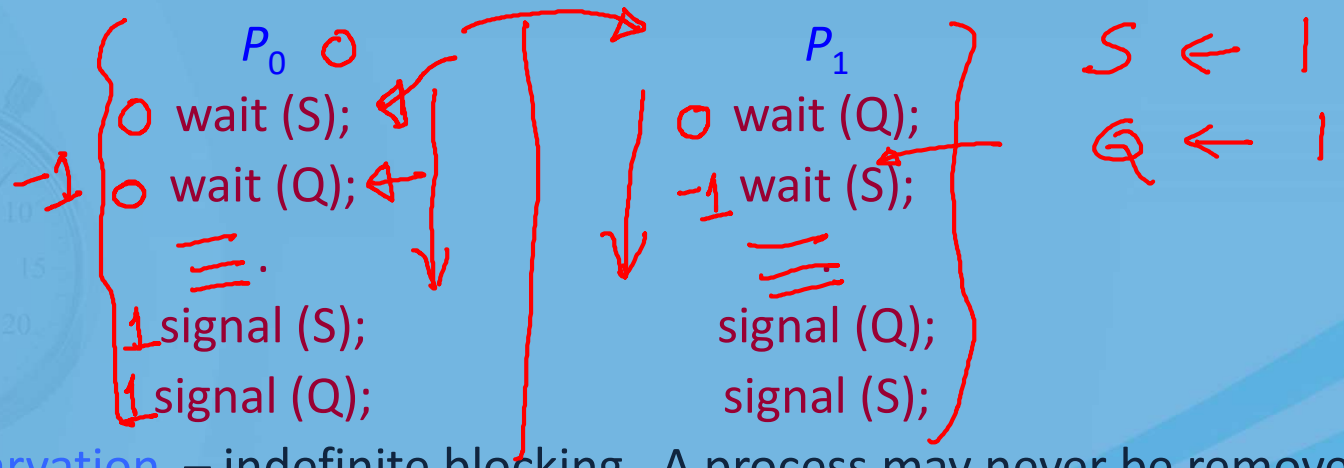
Semaphore implementation using Interrupts

```
wait ( s )  {
  inhibit interrupts;
  s.count - - ;
  if ( s.count < 0) {
     place this process in
                s.queue;
      block this process and
     allow interrupts;
       }
  else
     allow interrupts;
}
```

```
signal ( s )  {
inhibit interrupts;
  s.count + + ;
  if ( s.count <= 0) {
     remove a process P from
                s.queue;
     place process P on ready list;
  }
  allow interrupts;
}
```

Deadlock & 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



- 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

Implementation of a Counting Semaphore by using Two Binary Semaphores and a Count variable

- Binary semaphore will have an integer value 0 or 1.
- It is simpler to implement, depending on underlying hardware.
- Data structure needed for counting semaphore implementation
  - binary semaphores S1 , S2 ;
  - int C ;
  - Initialize S1 = 1; S2 = 0 ; C = initial value of counting semaphore S.

Implementation of a Counting Semaphore

Wait operation

```
wait ( S1 ) ;
C - - ;
if ( C < 0 ){
    signal ( S1 ) ;
    wait ( S2 );
}
else
signal ( S1 ) ;
```

Signal Operation

```
wait ( S1 ) ;
C + + ;
if ( C < = 0)
    signal ( S2 ) ;
    signal ( S1 ) ;
else
    signal ( S1 ) ;
```

Operating Systems
CS F372

Lect 41: Synchronization

BIJU K RAVEENDRAN

SEMAPHORE OPERATIONS IN C

semaphore.h

- int sem_init (sem_t *sem, int pshared, unsigned int value);
  - Initializes the unnamed semaphore at the address pointed to by sem
  - value specifies the initial value of the semaphore (Negative values are not allowed)
  - pshared = 0 (semaphore is shared between the threads of a process)
  - pshared = non-zero (semaphore is shared between the processes)
  - Returns 0 on success; -1 on error
- int sem_destroy(sem_t *sem)
  - Destroys the semaphore (initialized by sem_init) object
  - No threads should be waiting on the semaphore

- int sem_wait(sem_t *sem)
  - Decrements the semaphore count by 1.
  - If semaphore value is greater than 0 then decrement proceeds and the function returns immediately
  - If the semaphore currently has the value zero, then the call blocks until either
    - It becomes possible to perform the decrement (semaphore value raises above 0) or
    - A signal handler interrupts the call
- int sem_trywait(sem_t *sem)
  - Same as sem_wait except
    - If the decrement can not be immediately performed then call returns an error (EAGAIN) instead of blocking

- int sem_timedwait(sem_t *sem, __const struct timespec *time)
  - Same as sem_wait except
    - Argument “time” specifies a limit on the amount of time that the call should block if the decrement can not be immediately performed.
    - “time” argument points to a structure that specifies an absolute time out in seconds and nanoseconds since (00:00:00, 1st January 1970)

```
struct timespec {
    time_t tv_sec;      /* Seconds */
    long   tv_nsec;     /* Nanoseconds [0 .. 999999999] */
};
```

    - If the time out has already expired at the time of the call and semaphore could not be locked immediately then this call fails with a timeout error (ETIMEDOUT)

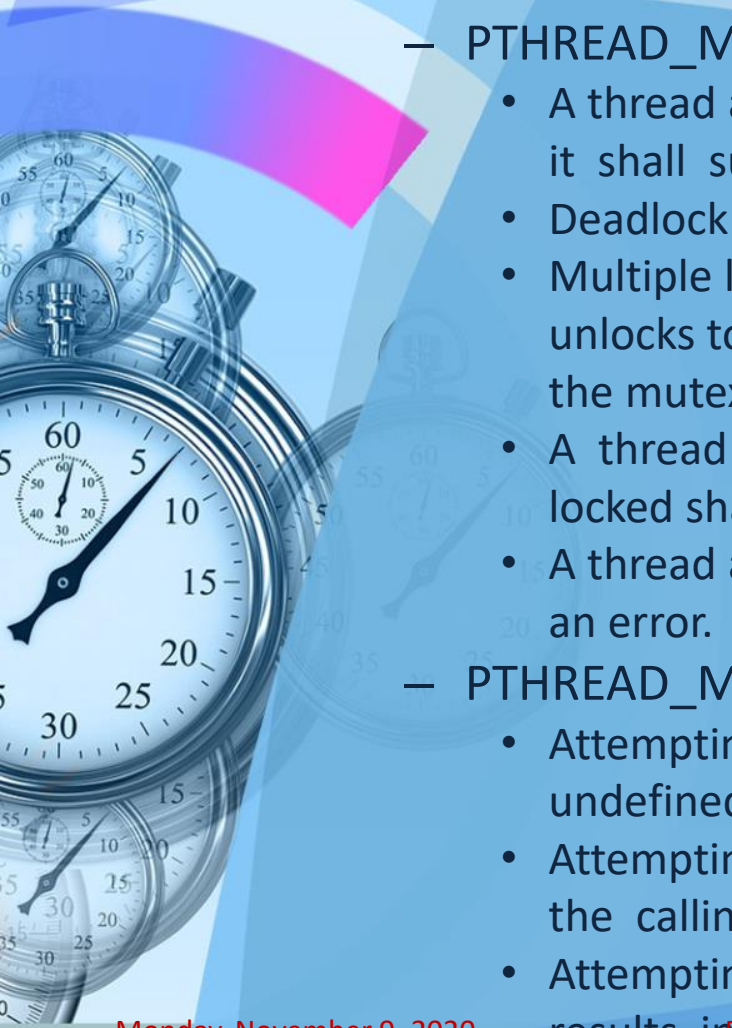
- sem_post (sem_t *sem)
  - Automatically increases the count by 1
  - This function never blocks
- sem_getvalue(sem_t *sem, int *sval)
  - Stores the count of the semaphore to sval
  - If one or more processes / threads are blocked waiting to lock the semaphore with sem_wait POSIX provides two possibilities for the value returned in sval
    - 0 is returned (Linux follows this)
    - A negative number whose absolute value is the count of the number of processes / threads currently blocked
  - Returns 0 on success; -1 otherwise

- Initialize a spin lock
  - int pthread_spin_init (pthread_spinlock_t *lock, int pshared)
- Destroy a spin lock
  - int pthread_spin_destroy (pthread_spinlock_t *lock)
- Try locking a spin lock
  - int pthread_spin_trylock (pthread_spinlock_t *lock)
- Lock a spin lock
  - int pthread_spin_lock (pthread_spinlock_t *lock)
- Unlock a spin lock
  - int pthread_spin_unlock (pthread_spinlock_t *lock)

- Initialize a mutex
  - int pthread_mutex_init (pthread_mutex_t *mutex, pthread_mutexattr_t *mutexattr)
- Destroy a mutex
  - int pthread_mutex_destroy (pthread_mutex_t *mutex)
- Try locking a mutex
  - int pthread_mutex_trylock (pthread_mutex_t *mutex)
- Lock a mutex
  - int pthread_mutex_lock (pthread_mutex_t *mutex)
- Timed Lock
  - int pthread_mutex_timedlock (pthread_mutex_t *mutex, struct timespec *time)

- int pthread_mutex_unlock (pthread_mutex_t *mutex)
- int pthread_mutexattr_init (pthread_mutexattr_t *attr)
- int pthread_mutexattr_destroy (pthread_mutexattr_t *attr)
- int pthread_mutexattr_getpshared (pthread_mutexattr_t *attr, int *pshared)
- int pthread_mutexattr_setpshared (pthread_mutexattr_t *attr, int pshared)
  - PTHREAD_PROCESS_SHARED
  - PTHREAD_PROCESS_PRIVATE (Default)

- int pthread_mutexattr_gettype (pthread_mutexattr_t *attr, int *kind)
- int pthread_mutexattr_settype (pthread_mutexattr_t *attr, int kind)
- “kind” value can be of 4 types
  - PTHREAD_MUTEX_NORMAL
    - Does not detect deadlock
    - A thread attempting to relock this mutex without unlocking it shall deadlock
    - Attempting to unlock a mutex locked by a different thread results in undefined behavior
    - Attempting to unlock an unlocked mutex results in undefined behavior.



- PTHREAD_MUTEX_RECURSIVE
  - A thread attempting to relock this mutex without first unlocking it shall succeed in locking the mutex.
  - Deadlock because of relocking cannot occur
  - Multiple locks of this mutex shall require the same number of unlocks to release the mutex before another thread can acquire the mutex
  - A thread attempting to unlock a mutex which another thread has locked shall return with an error
  - A thread attempting to unlock an unlocked mutex shall return with an error.
- PTHREAD_MUTEX_DEFAULT
  - Attempting to recursively lock a mutex of this type results in undefined behavior.
  - Attempting to unlock a mutex of this type which was not locked by the calling thread results in undefined behavior
  - Attempting to unlock a mutex of this type which is not locked results in undefined behavior

- PTHREAD_MUTEX_ERRORCHECK
  - Provides error checking
  - A thread attempting to relock this mutex without first unlocking it shall return with an error
  - A thread attempting to unlock a mutex which another thread has locked shall return with an error
  - A thread attempting to unlock an unlocked mutex shall return with an error.
- int pthread_mutexattr_getprotocol (pthread_mutexattr_t *attr, int *protocol)
- int pthread_mutexattr_setprotocol (pthread_mutexattr_t *attr, int protocol)
  - PTHREAD_PRIO_NONE, PTHREAD_PRIO_INHERIT, PTHREAD_PRIO_PROTECT

- PTHREAD_PRIO_NONE
  - Thread's priority and scheduling shall not be affected by its mutex ownership
- PTHREAD_PRIO_INHERIT
  - Inherit the priority of the highest priority thread waiting on any of the mutexes owned by this thread and initialized with this protocol
- PTHREAD_PRIO_PROTECT
  - The thread shall execute at the higher of its priority or the highest of the priority ceilings of all the mutexes owned by this thread and initialized with this attribute (regardless of whether other threads are blocked on any of these mutexes or not)

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Producer – Consumer Problem

- One or more producers are producing some items (data) and a single consumer is consuming these items one by one.
- Consumer can not consume until producer has produced
- While producer is producing, consumer can not consume and vice versa
- We assume producer can produce as many items it wants (infinite buffer)

Producer – Consumer Problem

Buffer[ ] (infinite size), in ← 0, out ← 0;

PRODUCER	CONSUMER
while (1) { // Produce item; Buffer[in] = item; in = in + 1; }	while (1) { while (in == out); item = Buffer[out]; out = out +1; }

Bounded Buffer Problem

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

Producer – Consumer Problem

semaphore mutex ← 1, full ← 0; in ← 0, out ← 0;

PRODUCER

```
while (1) {
    // Produce an item;
     wait (mutex);
            Buffer[in] = item;
            in = in + 1;
     signal (mutex);
    signal(full) ;
}
```

CONSUMER

```
while (1) {
     wait (full) ;
     wait (mutex);
            item=Buffer[out];
            out = out +1;
     signal (mutex);
}
```

Finite Buffer Problem

- If the buffer size is BSIZE then ?
- Conditions to satisfy
- Producer
  - Produce only if Buffer space is available
- Consumer
  - Consume only if item is available in Buffer

Producer – Consumer Problem

in ← 0, out ← 0; count ← 0;

PRODUCER

```
while (1) {
    // produce an item
    while (count = = BSIZE);
    buffer[in] = item;
    in = (in + 1) % BSIZE;
    count = count + 1;
}
```

CONSUMER

```
while (1) {
    while (count = = 0);
     item = buffer[out];
    out = (out + 1) % BSIZE;
    count = count – 1;
}
```

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.

Producer – Consumer Problem

semaphore mutex ← 1, full ← 0, empty ← BSIZE;
in ← 0, out ← 0;

PRODUCER

```
while (1) {
    // Produce an item;
    wait(empty);
    wait (mutex);
          Buffer[in] = item;
           in = (in + 1) % BSIZE;
    signal (mutex);
   signal(full) ;
}
```

CONSUMER

```
while (1) {
     wait (full) ;
     wait (mutex);
           item = Buffer[out];
            out = (out + 1) % BSIZE;
     signal (mutex);
     signal (empty);
}
```

Operating Systems
CS F372

Lect 42: Synchronization

BIJU K RAVEENDRAN

Classical Problems of Synchronization

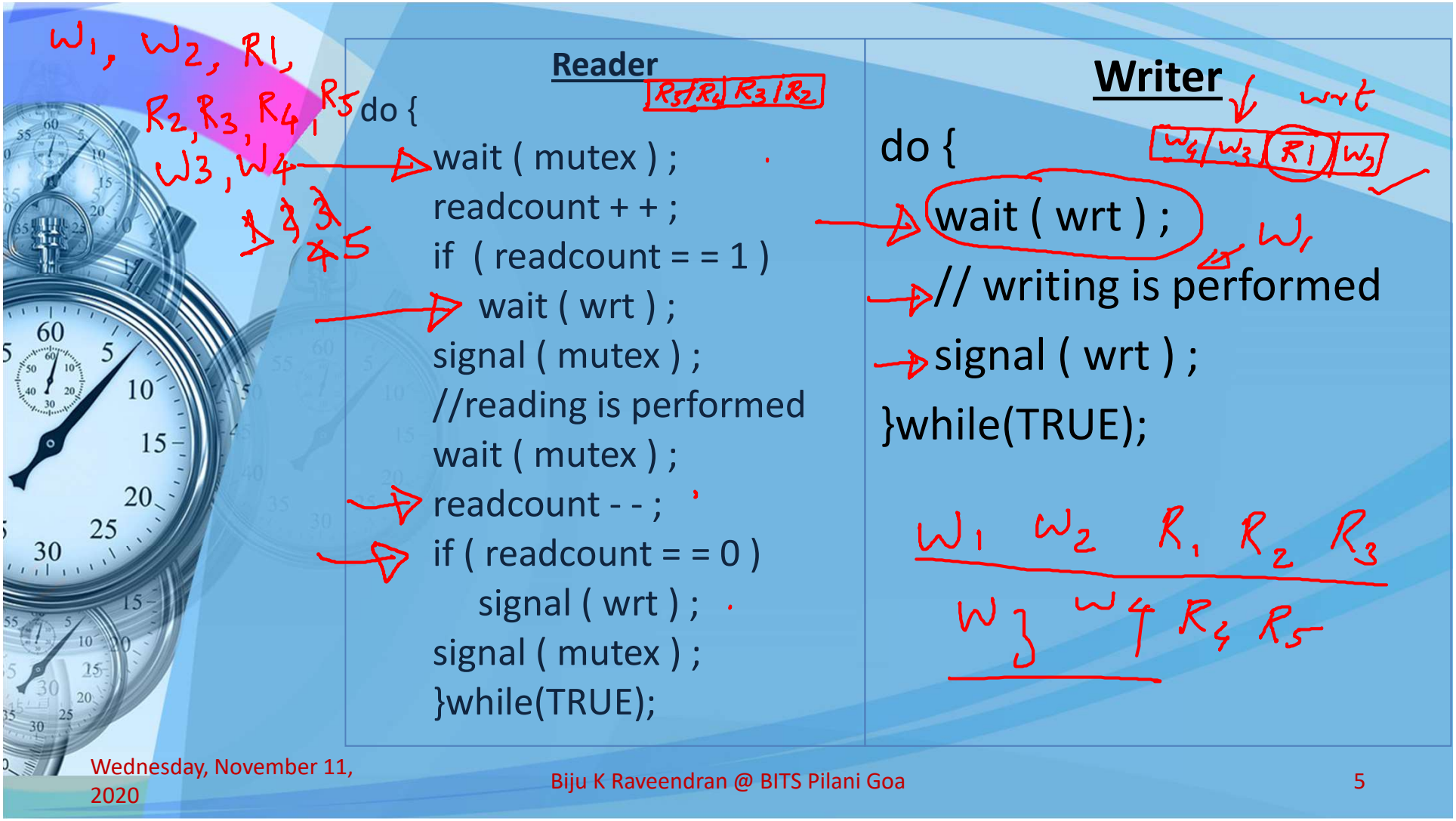
- Bounded-Buffer Problem
- **Readers and Writers Problem**
- **Dining-Philosophers Problem**

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
    - Reader and writer cannot access simultaneously.
    - No other processes are allowed to enter in the critical section when a writer is executing the critical section

Readers – Writers Problem

- Data structure support needed
  - semaphore mutex, wrt ;
  - int readcount ;
- Data structure Initialization
  - mutex = 1
  - wrt = 1
  - readcount = 0 ;



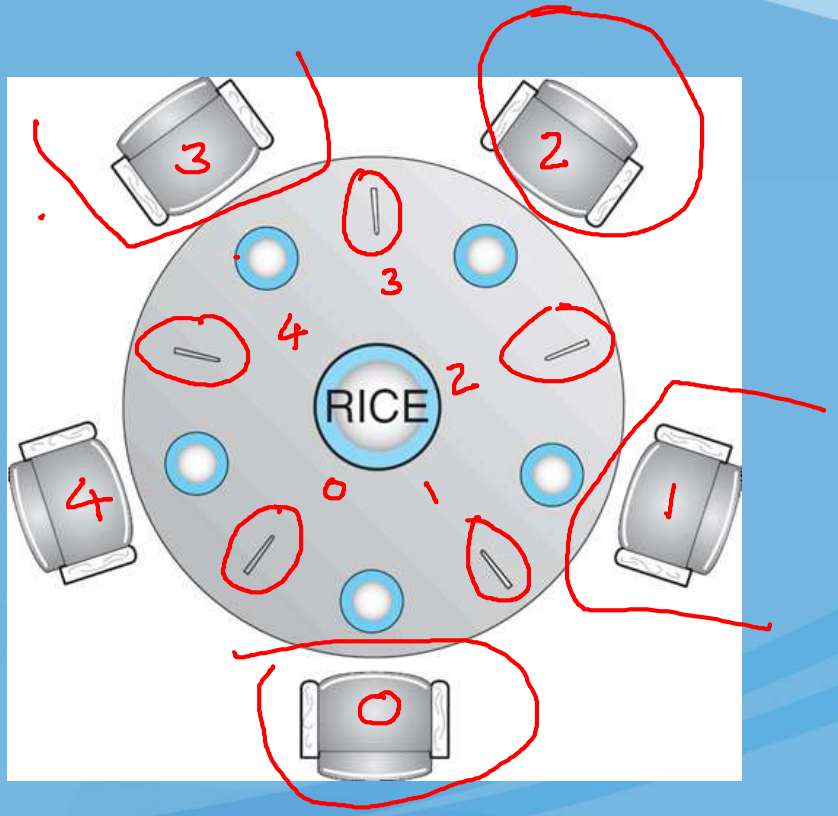
Reader

```
do {
    wait ( mutex ) ;
    readcount + + ;
    if  ( readcount = = 1 )
        wait ( wrt ) ;
    signal ( mutex ) ;
    //reading is performed
    wait ( mutex ) ;
    readcount - - ;
    if ( readcount = = 0 )
        signal ( wrt ) ;
    signal ( mutex ) ;
    }while(TRUE);
```

Writer

```
do {
    wait ( wrt ) ;
    // writing is performed
    signal ( wrt ) ;
}while(TRUE);
```

Dining Philosophers Problem



Dining Philosophers Problem

- Data structure support needed
  - semaphore chopstick [N] ;
- Data structure Initialization
  - for(int i=0; i< N; i + +)

    chopstick [i] = 1;

Dining Philosophers Problem

- 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);
```

Dining Philosophers Problem

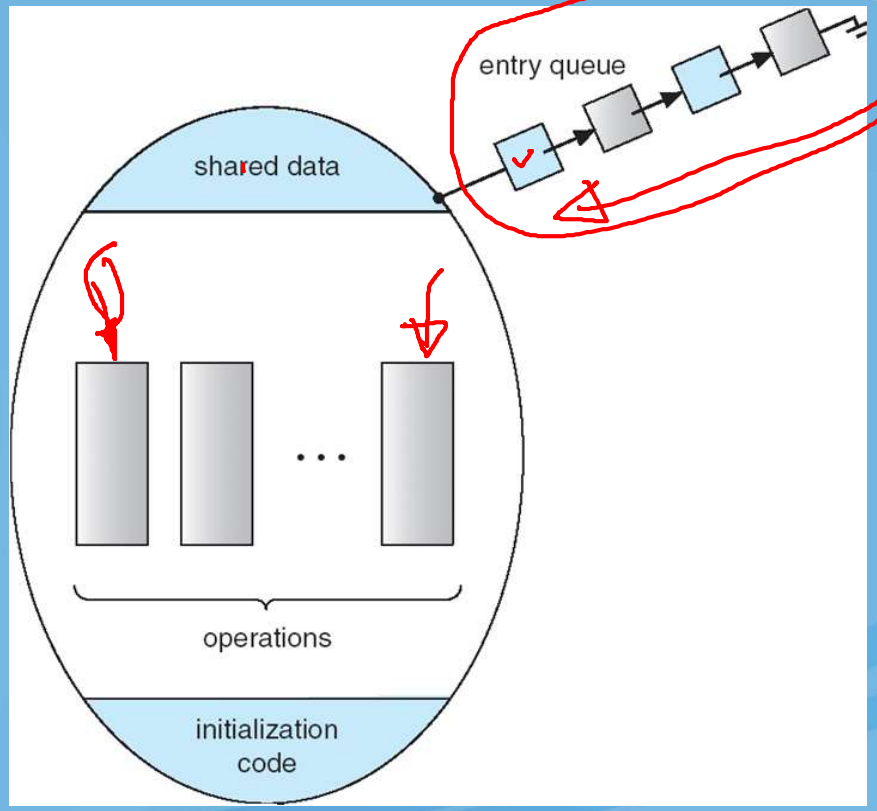
- Draw back
  - Dead lock is possible → if everyone picks up left/right chopstick
- Solution
  - The philosopher can pick up a chopstick if both the left & right one are free.
  - Let the odd philosopher pick the left chopstick and even philosopher pick the right chopstick.
  - Take only even number of philosophers

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time
- Process enters monitor by invoking one of its procedures
- Local data variables are accessible only by the monitor

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

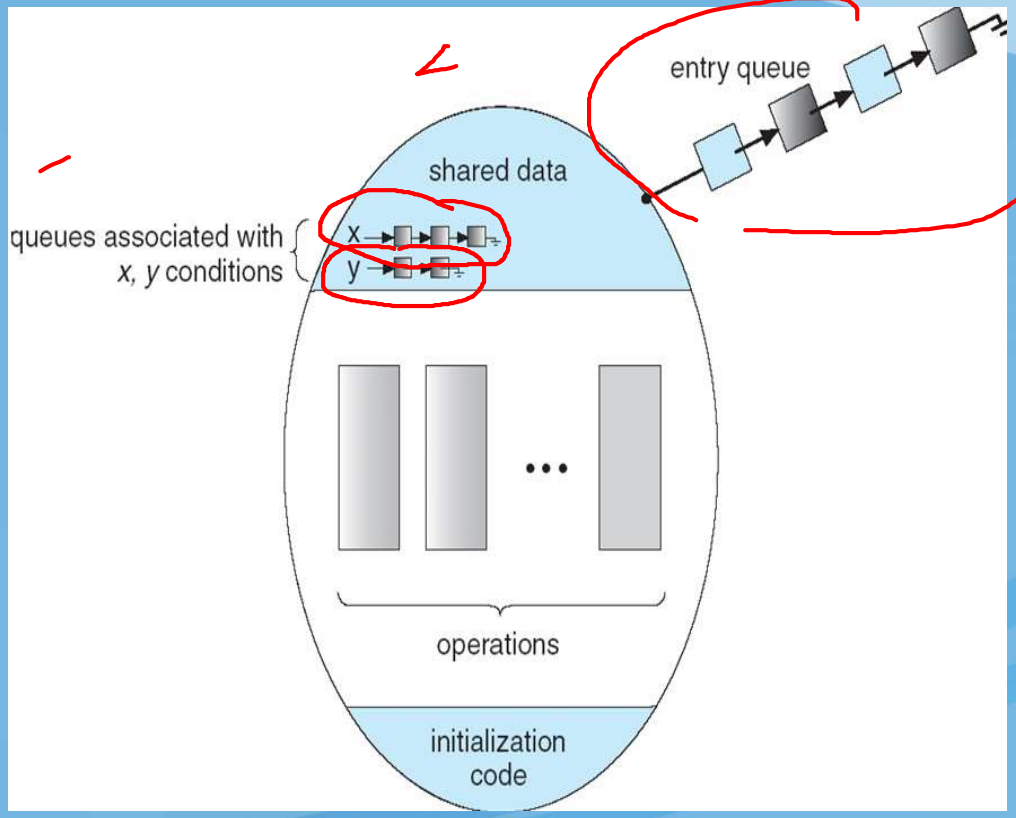
Schematic View of a Monitor

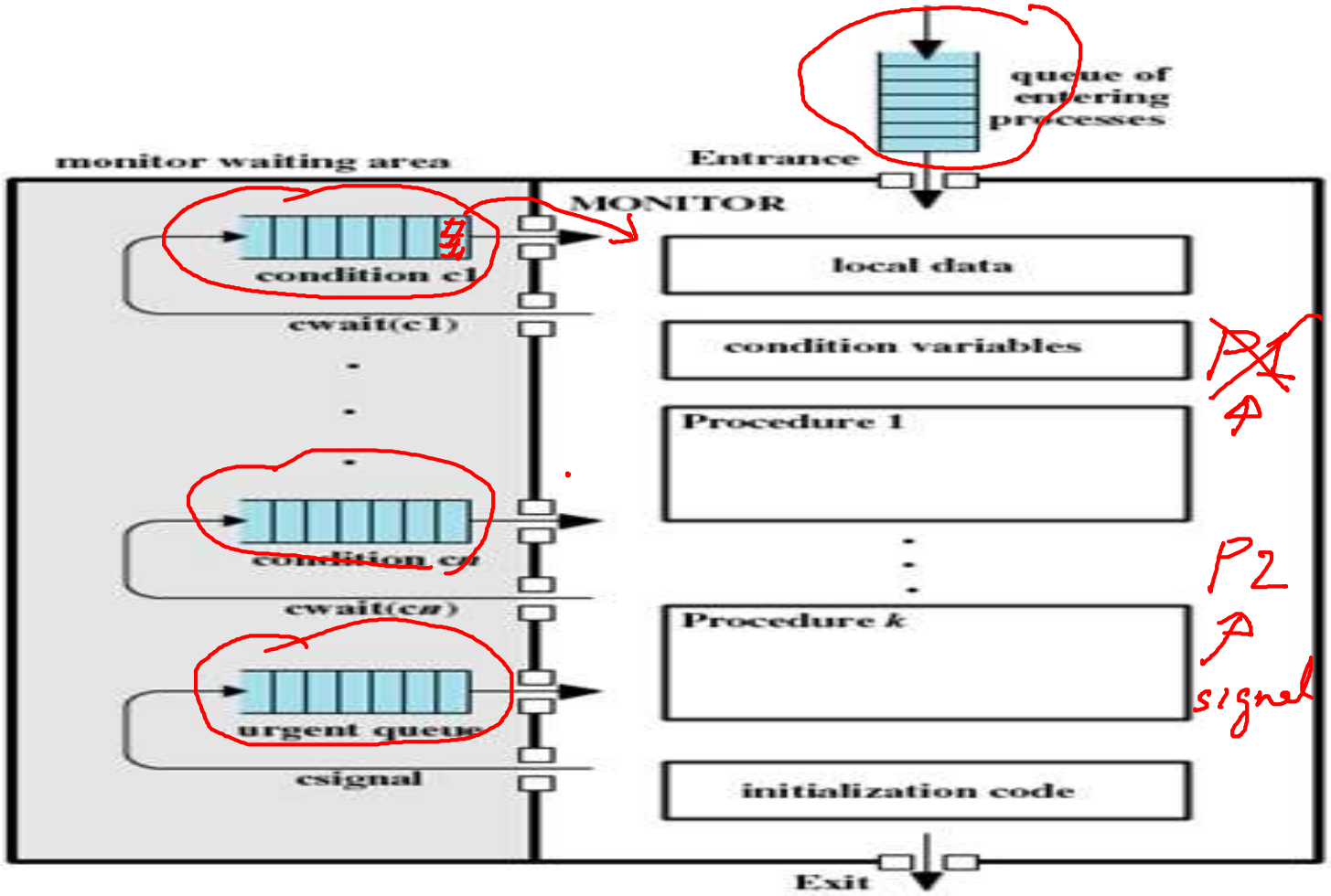


Condition variables

- To allow a process to wait within the monitor, a *condition* variable must be declared, as:
  - condition x, y;
- Two operations on a condition variable:
  - x.wait ()
    - a process that invokes the operation is suspended **until another process invokes x.signal**
  - x.signal ()
    - resumes exactly one suspended process**. If no process is suspended, then the signal operation has no effect.**

Monitor with Condition variables




queue of entering processes
monitor waiting area
Entrance
MONITOR
condition c1
cwait(c1)
condition cx
cwait(cx)
urgent queue
csignal
local data
condition variables
Procedure 1
Procedure k
initialization code
Exit

Condition Variables Choices

- If process P invokes `x.signal()`, and process Q is 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 it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it 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

Solution to Dining Philosopher Problem

```
monitor DP
 {
   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);
  }
```

```
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;
}

}
```

Operating Systems
CS F372

Lect 43: Synchronization Deadlock

BIJU K RAVEENDRAN

Solution to Dining Philosopher Problem

```
monitor DP
 {
   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);
   }
```

```
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 Philosopher Problem

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

DiningPhilosophters.pickup (i);

EAT

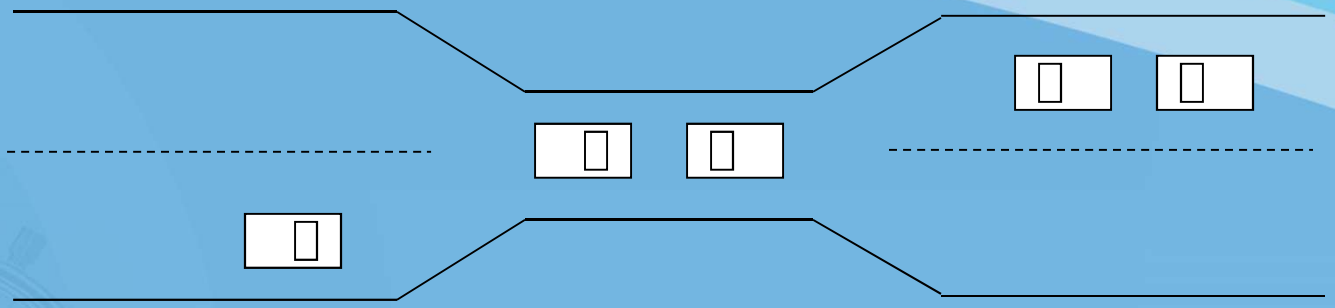
DiningPhilosophers.putdown (i);

DEADLOCK

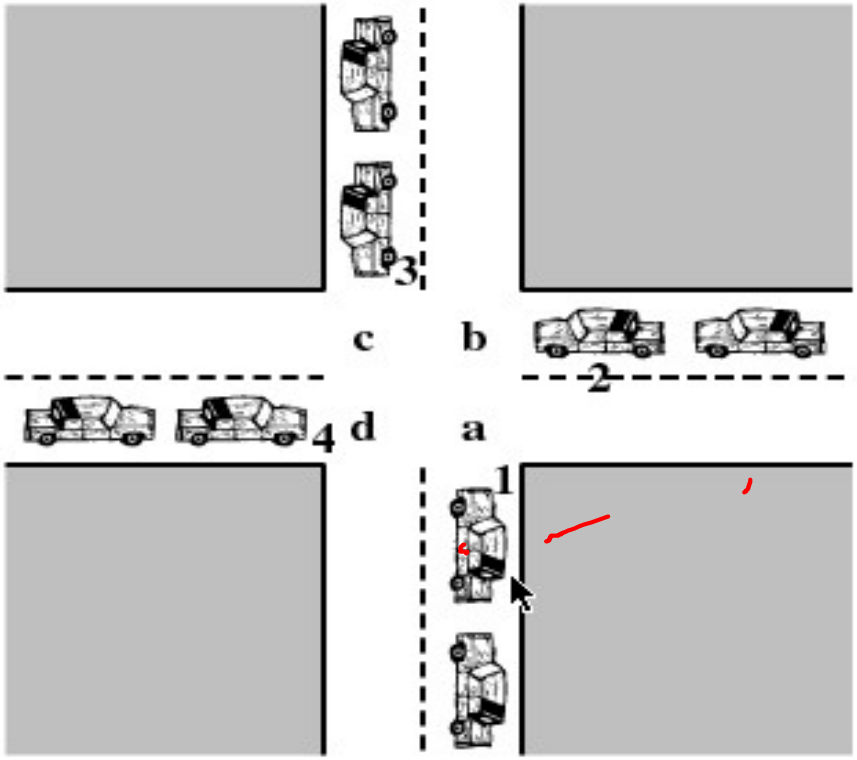
The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- Permanent blocking of a set of processes that either compete for system resources or communicate with each other
- No efficient solution

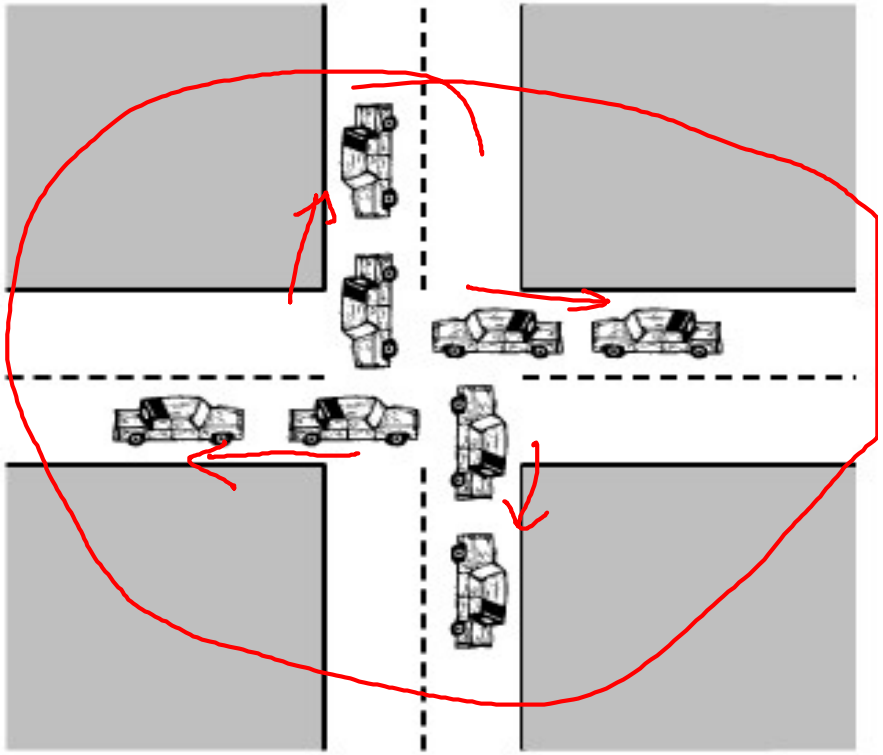
Bridge Crossing Example



- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible
- Note – Most OSes do not prevent or deal with deadlocks



(a) Deadlock possible



(b) Deadlock

Figure 6.1 Illustration of Deadlock

System Model

- Processes P1, P2, …, Pn
- Resource types R_1, R_2, . . ., R_m

  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.

Resource Allocation

- Request
  - If the request cannot be granted immediately then the requesting process must wait until it can acquire the resource.
- Use
  - Process operates on resource.
- Release
  - Process release the resource.
- Resources
  - Physical resource -- Printer,Memory space…..
  - Logical resource --- Semaphore,Files…..

Reusable Resources

- Used by only one process at a time and not depleted by that use
- Processes obtain resources that they later release for reuse by other processes
- Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores
- Deadlock occurs if each process holds one resource and requests the other

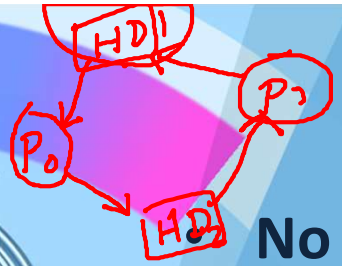
Consumable Resources

- Created (produced) and destroyed (consumed)
- Interrupts, signals, messages, and information in I/O buffers
- There is no limit on the number of consumable resources of a particular type.

Deadlock Characterization

Deadlock can arise if all four conditions hold simultaneously.

- **Mutual exclusion:**
  - only one process at a time can use a resource.
  - At least one resource must be in a non sharable mode; that is only one process at a time can use a resource.
  - If any other process request the resource, the requesting process must be delayed until the resource has been released.
- **Hold and wait:**
  - a process **holding** at least one resource is **waiting** to acquire additional resources held by other processes



Deadlock Characterization

- **No preemption:**
  - a resource can be released only voluntarily by the process holding it, after that process has completed its task .(The resource cannot be preempted).
- **Circular wait:**

  there exists a set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that
  - P_0 is waiting for a resource that is held by P_1,
  - P_1 is waiting for a resource that is held by P_2,
  - …,
  - P_{n-1} is waiting for a resource that is held by P_n, and
  - P_n is waiting for a resource that is held by P_0.