OPERATING SYSTEMS

#7

After A.S.Tanenbaum, Modern Operating Systems
3rd edition
Uses content with permission from Assoc. Prof. Florin Fortis, PhD

IPC

Inter-Process Communication

IPC

- There are three requirements for inter-process communication:

1. Establish a way for passing information between processes.
2. Emphasize the means by which one can guarantee the activities of processes
3. Assure means for a correct order of execution of the process.

IPC. RACE CONDITIONS

- A situation where the result of the execution for two or more processes that share some resources depends on the execution order of these processes is called a *race condition*.
- Example:
  - In order to print data in a UNIX system, a special folder is used to keep print orders – the spooler directory. For this directory specific mechanisms are implemented, to guarantee that all requests are properly solved.
  - A *race condition* can occur when two processes request simultaneously print orders, and these orders are registered at the same time in the spooler directory. By interleaving the operations of obtaining a print ticket, saving the print job and updating the printing ticket value, it is possible that the request of one of these processes to be never solved.

IPC. RACE CONDITIONS INTERLEAVING

- 1. S=1
- 2. S=S+2

- A. S=2
- B. S=S+2

- Can you offer a final value after the execution of the two processes?

- The two processes are executed simultaneously. S variable is shared.

IPC. CRITICAL REGIONS

- The part of a process where competitive services are required (eventually from the OS, usually because of some shared resources) is called ***critical region***.
- The protection of critical regions can be realized by using specific mechanisms, able to assure the mutual exclusion of processes.
  - The mechanisms of mutual exclusion are also good candidates for race condition avoidance.
- A race condition can be avoided if it is possible to offer a mechanism that guarantee that two processes are not in simultaneously in their own critical regions.

IPC. CRITICAL REGIONS

- There are several conditions for a correct solution for the critical section problem:

1. Two processes cannot be simultaneously inside their own critical regions.

2. In a solution for the critical region, there shouldn't be made any assumption about the speed or the number of processors.

3. A process that is functioning outside its own critical region cannot block the activity of any other process.

4. A process cannot wait forever for its entrance in his own critical region.

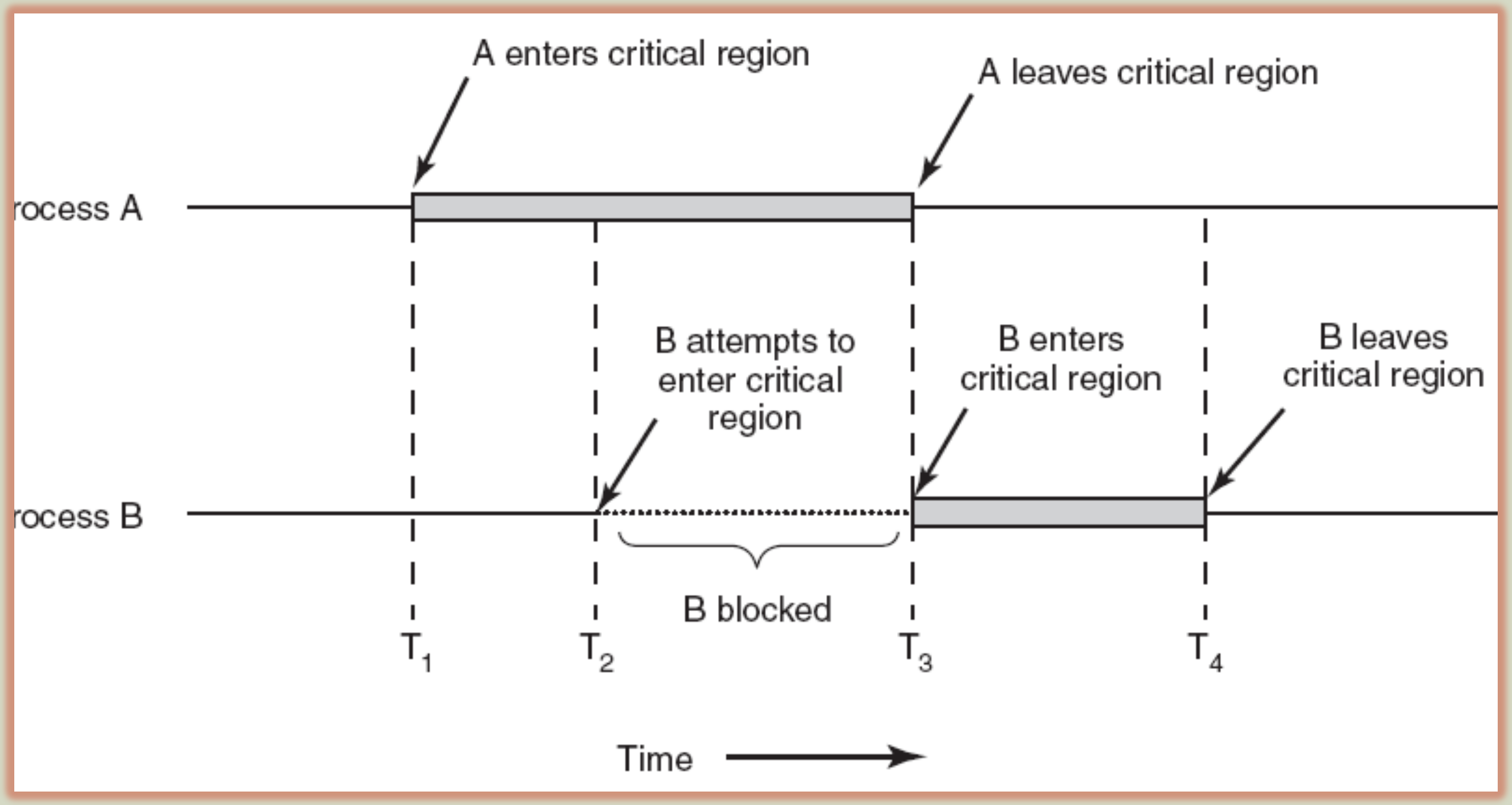
IPC. CRITICAL REGIONS

- However, there is a second version for these conditions (due to W.Stallings) :

1. Only one process can be in its own critical region at one moment.
2. In a solution for the critical region, there shouldn't be made any assumption about the speed or the number of processors.
3. A process that is blocked outside its critical region should not alter the functioning of any other process.
4. No process that is waiting for its entrance in its critical region should be postponed indefinitely.
5. When there is no process inside a critical region, the entrance in a critical region should be granted for the first process that want to access its own critical region.
6. A process can be in its critical region only for a limited period of time.

Exercise: SHOW THAT THESE TWO SETS OF RULES ARE EQUIVALENT.

IPC. ACHIEVING MUTUAL EXCLUSION BY CRITICAL REGIONS



IPC.

Mechanisms for mutual exclusion

IPC. MECHANISMS FOR MUTUAL EXCLUSION

Proposals for achieving mutual exclusion:

1. Disabling interrupts
2. LOCK variables
3. Strict alternation
4. Peterson’s solution
5. The TSL instruction

IPC. MECHANISMS FOR MUTUAL EXCLUSION

Interrupt blocking

- This is the simplest solution that one can imagine:
  - every process that enter its critical region should block (or deactivate) interrupts.
- No interrupts are possible for other processes
  - Notice that this will include … timer interrupts!
- Moreover, the processor cannot be allocated to other processes (because of the timer interrupts);
  - the current process is able to perform now its job without any interference from another processes.

Exercise: **which of the four (or six) rules are not satisfied? Why?**

- However, several problems can occur in this solution.

For example, in a multi-processor system, the interrupts are deactivated only for a single processor. Any other process, running on other processor, could enter its own critical region.

Can you identify other problems?

IPC. MECHANISMS FOR MUTUAL EXCLUSION

LOCK variables

This is a simple software solution, based on the following idea:

- Any process that is willing to enter its critical region should test first the value of the LOCK variable;
- If its value is 0, it sets it on 1, and enters the critical region;
- Otherwise, the process should wait until the value f LOCK is 0 again.

- Theoretically, this is a “*good*” solution. However, because the operation on the LOCK variable are not made atomically, it is possible that two processes to be allowed simultaneously in their own critical regions (for example, setting LOCK value on 1 occurs only after the other process checked the value of the LOCK variable – 0)
- This kind of situation usually occur because of the phenomenon of operation *interleaving*.

IPC. MECHANISMS FOR MUTUAL EXCLUSION

Strict alternation

- This is another version for the previous solution, using LOCK variables.
  - In this situation, two processes share a variable, turn, used only in order to control the entrance in the critical region.
  - Every process should modify this value when leaving its critical region, in order to offer this region to the other process. The other process is blocked in a cycle of permanent checking, until the required value for entering the critical region is reached.
- This permanent verification is called ***busy waiting***.
- A situation where a process is blocked due to an *busy wait* is called… ***spin lock***.
- Busy waiting should be avoided, as much as possible, in the activity of a process. However, it could be used only when the estimated “busy waiting” time is short enough.

IPC. MECHANISMS FOR MUTUAL EXCLUSION STRICT ALTERNATION

```
while (1) {
   while (turn != 0) ; // busy wait
   critical_section () ;
   turn = 1 ; // pass to process 2
   noncritical_section () ;
 }
```

```
while (1) {
   while (turn != 1) ; // busy wait
   critical_section () ;
   turn = 0 ; // pass to process 1
   noncritical_section () ;
 }
```

- In this solution a situation when a faster process is obliged to wait after a slower process when the faster one wants to enter its own critical region could occur.
  - Thus, the first (and the fastest) process could execute twice its critical section and arrive again to the entry point for another critical section before the second one is able to finish its first execution of its non-critical section!

Exercise: are there any rules that are not satisfied in this solution? Which ones?

IPC. MECHANISMS FOR MUTUAL EXCLUSION

Peterson's solution

- There is a solution offered by *Dekker* and *Peterson*, in order to improve the previous solution.
  - Now, processes can pass the control for the critical region by using the new variable ***turn***.
  - However, if a process holds the turn variable, and it is not really interested in the critical section, the other process could be allowed to enter (by an explicit expression of its interest) its critical section.
  - Always the interest for critical section should be announced before the critical section, and leaved after leaving the critical section.

IPC. MECHANISMS FOR MUTUAL EXCLUSION PETERSON'S SOLUTION

Leaving critical region

```
int turn ;
int interest[2] ;
void leave_region (int
  process) {
 interest[process] =
  FALSE ;
}
```

Entering critical region

```
void enter_region (int
  process) {
 int other ;
 other = 1 – process ;
 interest [process] =
  TRUE ;
 turn = process ;
 while ((turn == process)
  && (interest[other] ==
  TRUE)) ; // busy wait
}
```

IPC. MECHANISMS FOR MUTUAL EXCLUSION

TSL Instruction (Test, Set, Lock)

- This time we have a hardware solution, based on an instruction offered by several processors. The **TSL** instruction has the following syntax

  TSL Reg, LOCK

- The functioning of this instruction is quite simple:
  - it deposits first the value from LOCK in the register Reg;
  - then it stores a non-negative value in LOCK.
  - All these operations are guaranteed to be atomic (indivisible), so other process or processor is not able to access these memory addresses simultaneously.

IPC. MECHANISMS FOR MUTUAL EXCLUSION TSL INSTRUCTION

- A solution based on the TSL instruction can be easily modeled. One should use a shared variable, LOCK, initialized by 0. The value of LOCK is reset when leaving the critical section.

```
enter_region:
  TSL RX,LOCK
  CMP RX, #0
  JNE enter_region
  RET
```

```
leave_region:
  MOV LOCK, #0
  RET
```

IPC. MECHANISMS FOR MUTUAL EXCLUSION
TSL INSTRUCTION

- A second solution based on the TSL instruction can be identified. This solution is based on the "exchange" instruction.

```
enter_region:
  MOV RX,#1
  XCHG RX, LOCK
  CMP RX,#0
  JNE enter_region
  RET
```

```
leave_region:
  MOV LOCK, #0
  RET
```

IPC. BUSY WAITING

- All the solution presented until now has the same weakness: ***busy waiting***.
- Also, most of the solutions considered make the presumption of a strict protocol for accessing the critical region.
  - Busy waiting can generate a severe problem in an OS, namely the problem of *inverted priorities*. This is a situation when a process of higher priority depends on the activity of a process of lower priority (that has – accidentally – entered the critical section).
  - Because of the very different priorities, the process with a lower priority could have the "**bad luck**" of never exiting its critical region, because the process with a high priority is always preferred (by the planning algorithm). This situation cannot evolve, since the process of higher activity has the monopoly over the processor (because of its busy waiting cycle).

IPC. MECHANISMS FOR MUTUAL EXCLUSION

Sleep() and **wakeup**() primitives

- The ***sleep()*** primitive is used in order to block the activity of a process that is waiting on its entrance to the critical section, instead of *busy waiting*.
- The ***wakeup()*** primitive could be used in order to "wakeup" processes previously blocked.
- However, these two primitives require some addressing mechanisms (for example, by using a shared location of memory) in order to address the other process.

IPC. MECHANISMS FOR MUTUAL EXCLUSION SLEEP-WAKEUP: PRODUCER-CONSUMER

Producer process

```
void producer () {
  int item ;
  while (1) {
    item = create () ;
    if (count == N) sleep () ;
    add_item (item) ;
    count += 1 ;
    if (count == 1)
      wakeup (consumer) ;
  }
}
```

Consumer process

```
void consumer () {
  int item ;
  while (1) {
    if (count == 0) sleep () ;
    item = get_item () ;
    count -= 1 ;
    if (count == N-1)
      wakeup (producer) ;
    use_item (item) ;
  }
}
```

IPC. MECHANISMS FOR MUTUAL EXCLUSION SLEEP–WAKEUP: PRODUCER-CONSUMER

- In this solution, the producer is blocked when the buffer is full, and the consumer is blocked when the buffer is empty. Each process could be waked up when the block condition is no longer effective.
  - *A race condition still can occur*, because there is, for the moment, no constraint over the ***count*** variable.
- By using a scenario of interleaved operations (maybe based on some "collaboration" with the scheduler) it could be possible that a wakeup call to be wasted.
  - A consumer arriving at a sleep call could remain permanently blocked because it never knows that the buffer is no longer empty.
- After this moment, the producer could eventually fill the buffer of items. Finally, both processes ends by being blocked for a wakeup signal that never occurs.

IPC. MECHANISMS FOR MUTUAL EXCLUSION

Semaphores

- The semaphores were introduced by *Dijkstra* in order to "avoid" the occurrence of wasted wakeup signals. The semaphores are used in order to count the wakeup calls that has been realized.
- A semaphore is just a non-negative integer value, together with two basic operations: **down** () and **up** () (**P** and **V**, in the original – Dutch – notation).
  - The *down* operation verifies the value of the semaphore, and it is different from 0, it is decremented (and the process can pass). If the semaphore has a 0 value, the process is blocked until it can continue the *down* operation.
  - The *up* operation is used only to increment the value of a semaphore. Both operations are guaranteed to be indivisible, and they can be carried out safely, without any interference from another process.

IPC. MECHANISMS FOR MUTUAL EXCLUSION SEMAPHORES

- One can model a solution for the producer-consumer problem by using two general semaphores:
  - ***full***, used to count filled positions from the buffer, and
  - ***empty***, used to count free positions from the buffer.
- Initially, *full* is 0, and *empty* is N (this being the dimension of the buffer).
- By simply using these two semaphores we cannot obtain the necessary mechanisms to avoid race conditions. For this purpose, one should also use a supplemental (binary) semaphore, ***mutex***, used to control the access to the critical region. This supplemental semaphore offer the guarantee that the producer and the consumer are not able to access simultaneously the buffer.
- This solution also offers the guarantee that both the producer and the consumer are blocked in extreme situations.

IPC. MECHANISMS FOR MUTUAL EXCLUSION SEMAPHORES. PRODUCER-CONSUMER PROBLEM

Producer process

```
void producer () {
  int item ;
  while (1) {
    item = create () ;
    down (&empty) ;
    down (&mutex) ;
    add_item (item) ;
    up (&mutex) ;
    up (&full) ;
  }
}
```

Consumer process

```
void consumer () {
  int item ;
  while (1) {
    down (&full) ;
    down (&mutex) ;
    item = get_item () ;
    up (&mutex) ;
    up (&empty) ;
  }
}
```

What happens when these two lines are swapped?

IPC. MECHANISMS FOR MUTUAL EXCLUSION

MUTEX

- The idea for this mechanism is based on binary semaphores (these semaphores are also used under the name of… *mutexes*).
- ***Mutex*** variables are constructed with only two states; ***blocked*** and ***free*** (or unblocked). The basic implementation is similar with the TSL based implementation.
- However, unlike **TSL**, in the implementation for mutex variables the busy waiting cycles can be avoided.
- Two basic procedures are provided, ***mutex_lock*** and ***mutex_unlock***, similar by construction with those from TSL, ***enter_region*** and ***exit_region***.
- Mutex variables and solutions based on these variables are very good for modeling critical section control for thread-based applications.
- Because the basic functioning of mutex variables does not require any switch to the kernel mode of functioning, thread switch operation can be solved quite quickly.

IPC. MECHANISMS FOR MUTUAL EXCLUSION MUTEX IMPLEMENTATION

- A simple implementation could be based on the TSL solution.

```
mutex_lock:
  TSL RX,MUTEX
  CMP RX,#0
  JZE ok
  CALL thread_yield
  JMP mutex_lock
ok: RET
```

```
mutex_unlock:
  MOV MUTEX, #0
  RET
```

IPC. MECHANISMS FOR MUTUAL EXCLUSION

1. Exercise: check if the solution based on semaphores and sleep-wakeup primitives verify the 4 (or 6) conditions for a correct solution.
2. Exercise: is the solution of Peterson correct?
3. Exercise: are there any problems that can occur in the solution based on semaphores when the down (&mutex) operation occurs before the down (&empty) operation?

IPC. MECHANISMS FOR MUTUAL EXCLUSION

- Solutions based on semaphores are very sensible in their construction. A simple error in application logic is enough to generate a deadlock situation: a situation when two or more processes wait indefinitely for some events to occur, and these events cannot occur.
- C.A.R Hoare and B. Hansen proposed a powerful synchronization primitive of higher level (implemented in several concurrent programming languages), called ***monitors***.

IPC. MECHANISMS FOR MUTUAL EXCLUSION MONITORS

- A monitor is a collection of procedures, variables and data structures, grouped together in modules. Processes can call procedures from these modules (monitor), without any direct access to its internal structures.
- Mutual exclusion is guaranteed as follows: the monitor guarantees that there is only one active process inside its modules; any process that is willing to call a monitor-procedure should be suspended if there is another process that is active inside the monitor, until the uniqueness condition is satisfied.
- However, only this simple implementation for mutual exclusion does not offer any guarantees that the 4 (6) conditions are satisfied: a consumer that is blocked inside a monitor (for example, a consumer trying to access an empty buffer) indefinitely blocks all the other processes that are trying to access the monitor simultaneously.

IPC. MECHANISMS FOR MUTUAL EXCLUSION CONDITION VARIABLES

Condition variables

- These situations can be protected by a novel mechanism: condition variables. These special structures are implemented together with two simple operations: ***wait*** and ***signal***.
- When current process identifies a situation where its activity can be blocked inside a monitor, it can use a wait operation on a condition variable. Now, the current process could be suspended and the monitor released for other processes. The blocked process could be waked-up when the condition variable is signaled and it should continue its activity only when the uniqueness condition is satisfied.
- The mechanisms introduced by monitors are used in different programming languages (most known being, of course, Java). Also, similar mechanisms are used together with threads, as synchronization mechanisms.

IPC. MECHANISMS FOR MUTUAL EXCLUSION MESSAGE PASSING

Message passing

- This mechanism is based on two simple procedures: **send**() and **receive**(). There are different methods of implementation, using different methods for the identification of communicating parts.
- For example, the **send**() primitive could be used in order to send messages to a fixed destination, or the **receive**() primitive could be used in order to wait from messages from (very) different sources.

IPC. MECHANISMS FOR MUTUAL EXCLUSION MESSAGE PASSING

Requirements

- A system based on message passing should satisfy several requirements for a correct implementation:
  - ***Establish a protocol*** for confirmation of message reception. This is necessary since there is a permanent threat that messages can be lost in a distributed environment. If there is no confirmation, the sender should issue again the message.
  - ***Multiple messages avoidance***. When using the protocol from previous point, it is possible to have several occurrences of the same message.
  - ***Process naming***. By using this mechanism one should avoid any ambiguity in the identity of communicating parts. This problem/requirement is highly related with process *authentication*.

IPC. MECHANISMS FOR MUTUAL EXCLUSION MESSAGE PASSING

Producer process

```
void producer () {
 int item ; message msg ;
 while (1) {
   item = create_item () ;
   receive (consumer,
  &msg);
   generate (&msg, item) ;
   send (consumer, &msg);
 }
}
```

Consumer process

```
void consumer () {
 int item ; message msg ;
 for (i=0; i<N; i++)
   send (producer, &msg) ; //
  empty
 while (1) {
   receive (producer, &msg) ;
   item = get_item () ;
   send (producer, &item) ;
   use_item (item) ;
 }
}
```

IPC. MECHANISMS FOR MUTUAL EXCLUSION MESSAGE PASSING

- This solution makes the presumption that there is a fixed number of messages in the system. Initially empty messages (containers) transmitted by the consumer, they should be filled-in by the producer and empty again by the consumer. The number of the messages in the system defines the buffer size.
- A different approach is based on mail-boxes, used in order to store messages at destination. Send and Receive should access now mail-boxes instead of processes. An write action in a full buffer should suspend the process until the mail-box is able to support the operation.
- Mail-box utilization emphasize several buffering mechanisms: the mail-box is the buffer of un-processed messages.
- If there is no buffering mechanism, the solution thus obtained is the *rendez-vous*.

THE PRODUCER-CONSUMER PROBLEM (1)

A Java solution for producer-consumer problem

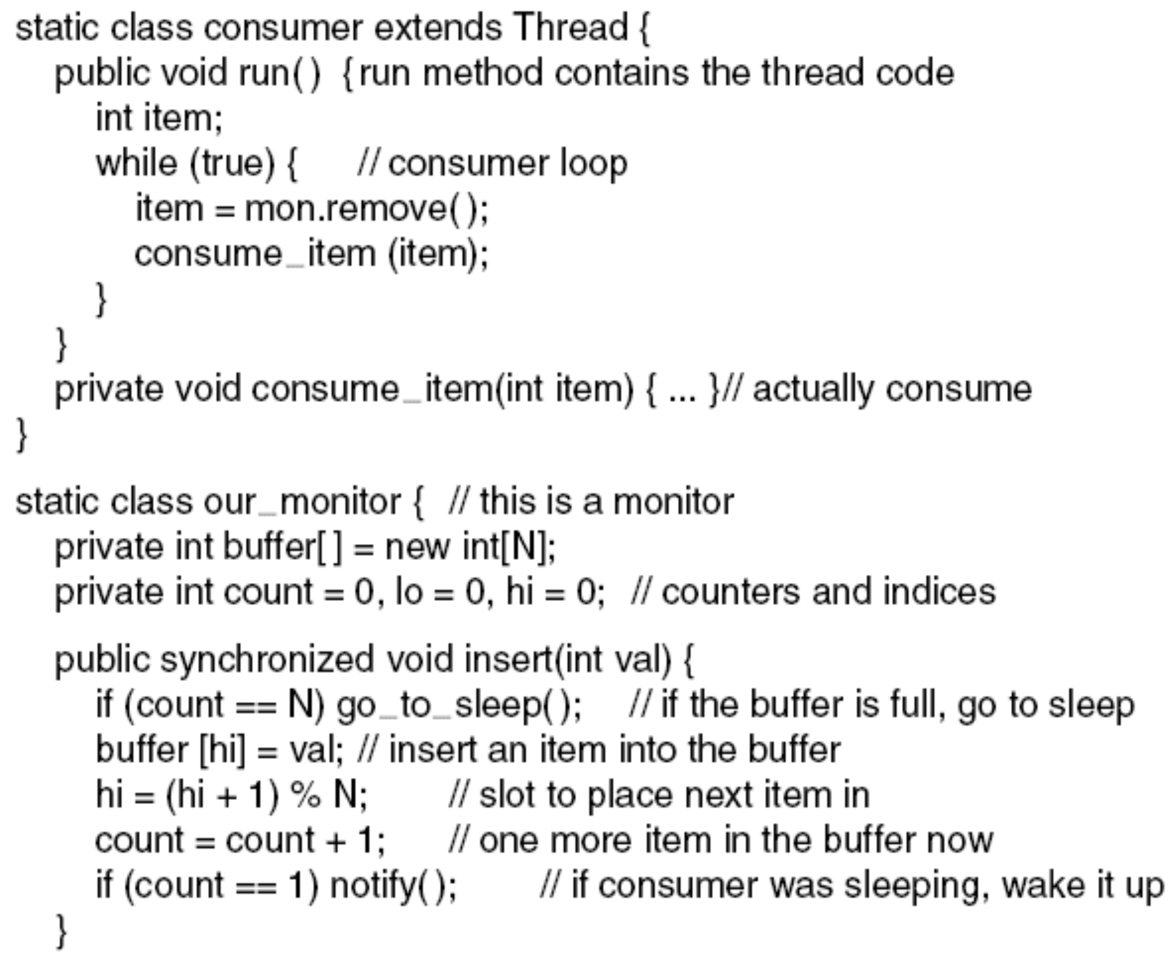
```
public class ProducerConsumer {
      static final int N = 100;      // constant giving the buffer size
      static producer p = new producer( );    // instantiate a new producer thread
      static consumer c = new consumer( ); // instantiate a new consumer thread
      static our_monitor mon = new our_monitor( );      // instantiate a new monitor

      public static void main(String args[ ]) {
        p.start( );      // start the producer thread
        c.start( );      // start the consumer thread
      }

      static class producer extends Thread {
        public void run( )  {// run method contains the thread code
          int item;
          while (true) {      // producer loop
            item = produce_item( );
            mon.insert(item);
          }
        }
        private int produce_item( ) { ... }      // actually produce
      }
```

THE PRODUCER-CONSUMER PROBLEM (2)

A Java solution for producer-consumer problem



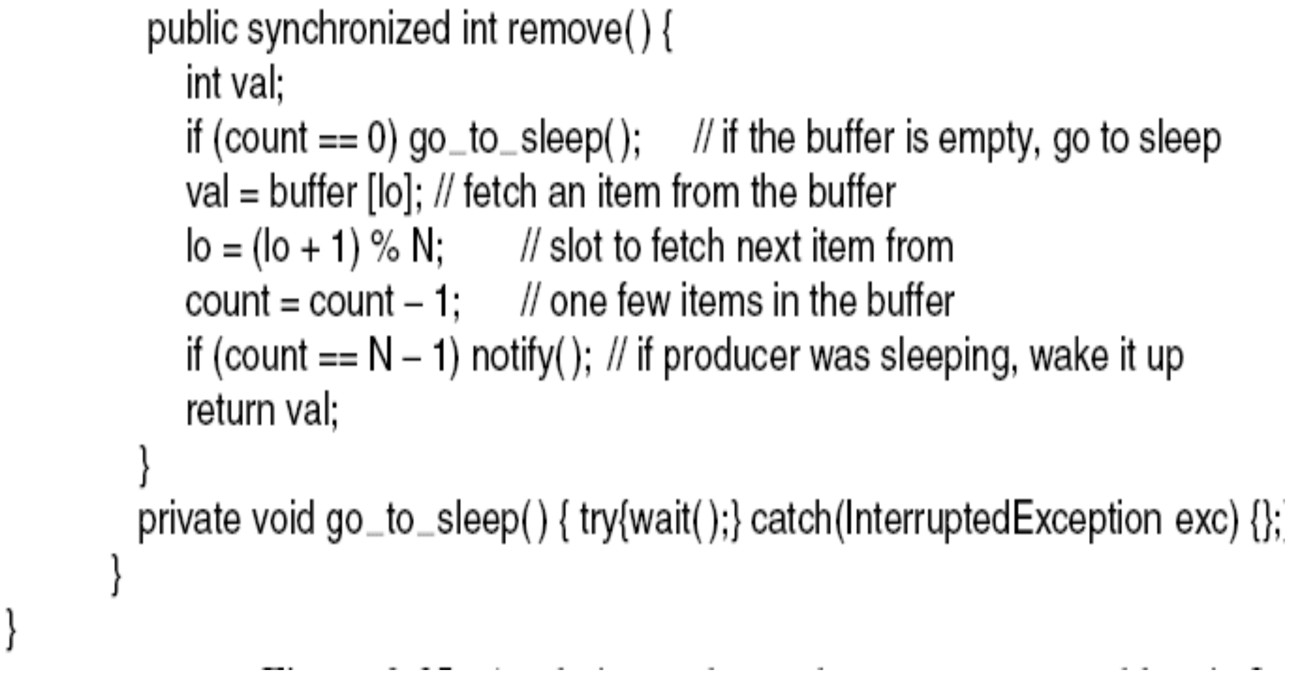
```
static class consumer extends Thread {
   public void run( )  {run method contains the thread code
      int item;
      while (true) {      // consumer loop
         item = mon.remove( );
         consume_item (item);
      }
   }
   private void consume_item(int item) { ... }// actually consume
}

static class our_monitor {   // this is a monitor
   private int buffer[ ] = new int[N];
   private int count = 0, lo = 0, hi = 0;   // counters and indices

   public synchronized void insert(int val) {
      if (count == N) go_to_sleep( );     // if the buffer is full, go to sleep
      buffer [hi] = val; // insert an item into the buffer
      hi = (hi + 1) % N;        // slot to place next item in
      count = count + 1;       // one more item in the buffer now
      if (count == 1) notify( );        // if consumer was sleeping, wake it up
   }
```

THE PRODUCER-CONSUMER PROBLEM (3)

A Java solution for producer-consumer problem



```
         public synchronized int remove( ) {
            int val;
            if (count == 0) go_to_sleep( );     // if the buffer is empty, go to sleep
            val = buffer [lo]; // fetch an item from the buffer
            lo = (lo + 1) % N;        // slot to fetch next item from
            count = count – 1;        // one few items in the buffer
            if (count == N – 1) notify( ); // if producer was sleeping, wake it up
            return val;
        }
        private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};
      }
}
```

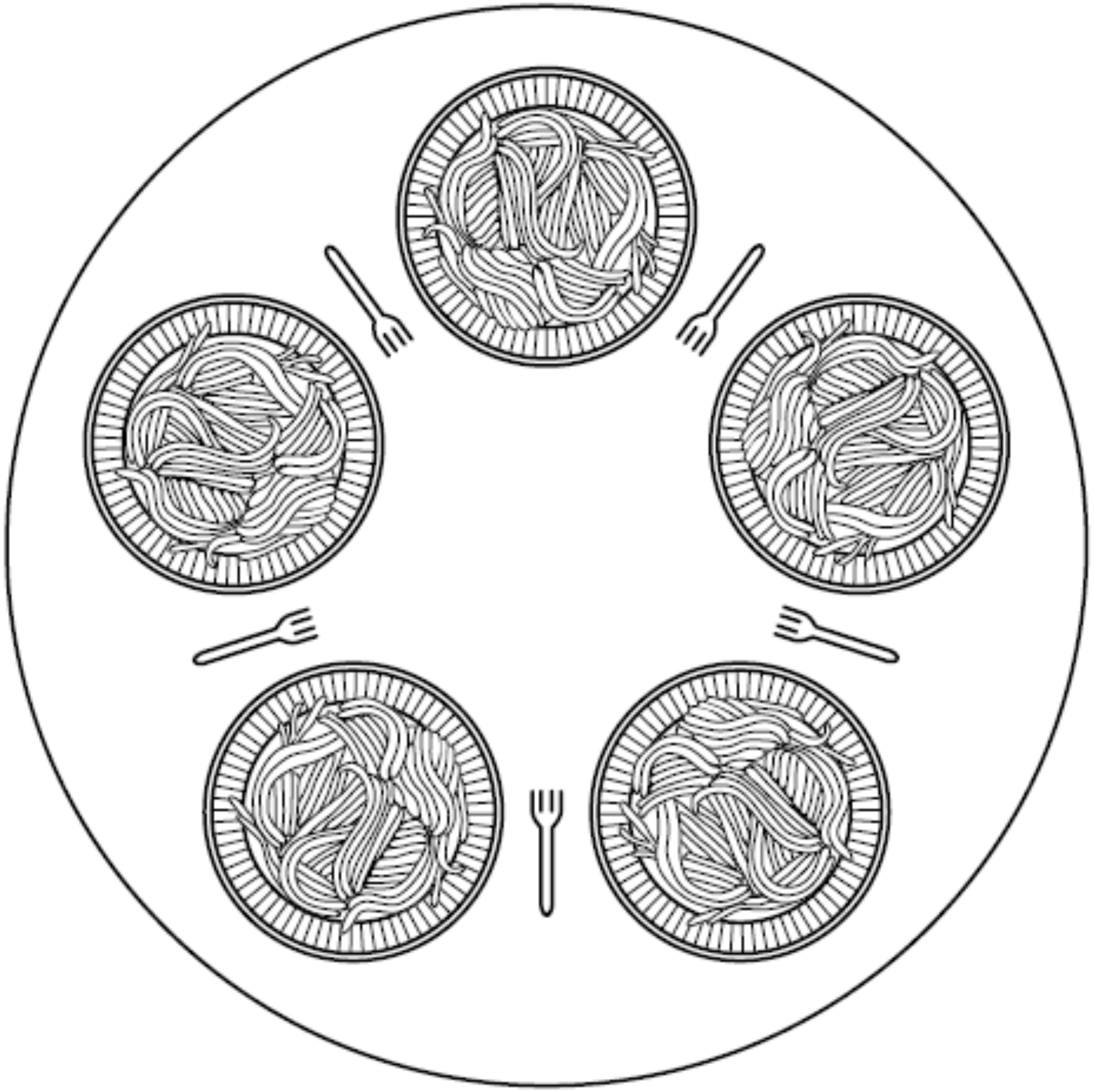
CLASSICAL COMMUNICATION PROBLEMS

These problems are for OWN STUDY.

Please read, understand, implement!

CLASSICAL COMMUNICATION PROBLEMS
DINING PHILOSOPHERS

- Here, five philosophers are sitting on a circular table. Each philosopher has a plate for food and two forks (or chopsticks), shared with the neighbors from left and right. There is a bowl (permanently) full of food. In order to feed, every philosopher needs the two forks (chopsticks).
- The life of a philosopher is made up from thinking periods and feeding periods. A hungry philosopher should take both forks (for example, first the left one and then the right one) and start feeding immediately he holds the two forks.
- Several situations can occur:
  - All the philosophers take the (left) fork simultaneously.
  - The philosophers does not pick a fork if the other is not available. Now, one can enforce a philosopher to wait indefinitely (starvation) by a bad “collaboration” of the others..
  - Provide exclusive access to the chopsticks. This is an acceptable solution. However, only one philosopher is able now to feed at a time.



DINING PHILOSOPHERS

Philosophers on table

DINING PHILOSOPHERS

A “bad” solution for dining philosophers

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

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

DINING PHILOSOPHERS (1)

A good solution for dining philosophers

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

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

DINING PHILOSOPHERS (2)

A good solution for dining philosophers

```
void take_forks(int i)                    /* i: philosopher number, from 0 to N–1 */
{
     down(&mutex);                        /* enter critical region */
     state[i] = HUNGRY;                   /* record fact that philosopher i is hungry */
     test(i);                             /* try to acquire 2 forks */
     up(&mutex);                          /* exit critical region */
     down(&s[i]);                         /* block if forks were not acquired */
}
```

DINING PHILOSOPHERS (3)

A good solution for dining philosophers

```
void put_forks(i)                         /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                       /* enter critical region */
      state[i] = THINKING;                /* philosopher has finished eating */
      test(LEFT);                         /* see if left neighbor can now eat */
      test(RIGHT);                        /* see if right neighbor can now eat */
      up(&mutex);                         /* exit critical region */
}

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

CLASSICAL COMMUNICATION PROBLEMS
READERS AND WRITERS

- In this problem there are several processes that are able to use resources (without consumption) and processes that are able to produce/update these resources.
- It should be possible that several processes use the resources simultaneously. It is not acceptable that several processes to produce/update items at the same time.
- Possible solutions:
  - Writers are active only after all the readers finish their activity. There is no protection against new readers, it is possible that the writers are obliged to wait indefinitely. In this situation, the readers are always preferred.
  - Writers are immediately active. Now, it is possible that the readers wait indefinitely, due to a high rate of writers.
  - New readers wait until a writer ends its activity. This solution has another weakness: the multiprogramming level is decreased since only one writer can be active, any other process should block.

READERS AND WRITERS (1)

A good solution for “readers and writers”

```
typedef int semaphore;              /* use your imagination */
semaphore mutex = 1;                /* controls access to 'rc' */
semaphore db = 1;                   /* controls access to the database */
int rc = 0;                         /* # of processes reading or wanting to */

void reader(void)
{
     while (TRUE) {                 /* repeat forever */
          down(&mutex);             /* get exclusive access to 'rc' */
          rc = rc + 1;              /* one reader more now */
          if (rc == 1) down(&db);   /* if this is the first reader ... */
          up(&mutex);               /* release exclusive access to 'rc' */
          read_data_base();         /* access the data */
          down(&mutex);             /* get exclusive access to 'rc' */
          rc = rc – 1;              /* one reader fewer now */
          if (rc == 0) up(&db);     /* if this is the last reader ... */
          up(&mutex);               /* release exclusive access to 'rc' */
          use_data_read();          /* noncritical region */
     }
}
```

READERS AND WRITERS (2)

A good solution for “readers and writers”

```
void writer(void)
{
     while (TRUE) {                      /* repeat forever */
          think_up_data( );              /* noncritical region */
          down(&db);                     /* get exclusive access */
          write_data_base( );            /* update the data */
          up(&db);                       /* release exclusive access */
     }
}
```

CLASSICAL COMMUNICATION PROBLEMS
BARBERSHOP

- In this problem there is a barber (namely, the processor), with a single sit (the active process) and several waiting chairs (ready processes). The barber is waiting if there are no clients, or it should service the customers in a well defined order. The clients should sit in empty chairs, could occupy the barber sit or leave the barbershop, if there are no empty chairs.
- This problem consists in planning the activity of the barber in order to avoid the occurrence of a race condition.