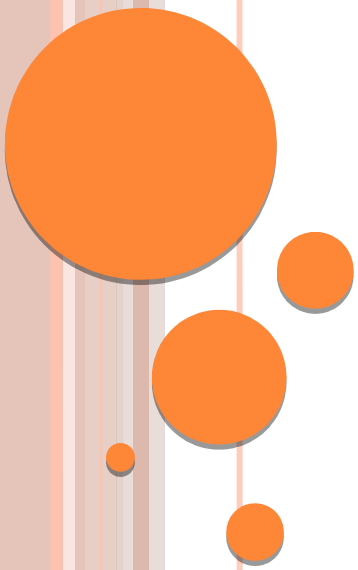


IŞLETİM SİSTEMLERİ

DERS 6 PROSESLER ARASI İLETİŞİM

PROSESLER ARASI İLETİŞİM (INTERPROCESS COMMUNICATION (IPC))

- prosesler, sıklıkla birbirleri ile iletişim kurarlar. Bir prosesin çıktısı başka bir prosesin girdi verisi olabilir. prosesler arası iletişimde üç konu önemlidir:

1. Bir proses diğerine nasıl veri gönderir?
2. Bir ya da daha fazla prosesin ortak alanları kullanırken dikkatli olmaları ve birbirlerinin iletişim yollarına girmemeleri gerekir. Örneğin bellekte kalan son 1MB kim kullanacak…
3. İletişimdeki uygun sıra nasıl olmalıdır? Bir proses veri gönderiyor diğeri bu veriyi yazdırıyorsa, ilk proses veri göndermediğinde ikincinin beklemesi ya da ikinci yazdırırken birincinin beklemesi gereklidir. İletişimde kullanılacak sıra önemlidir.

- prosesler arası iletişimde kullanılan yöntemler, threadler arasında da kullanılır çünkü threadlerde proseslerdeki gibi aynı verileri ortak kullanmaktadırlar.

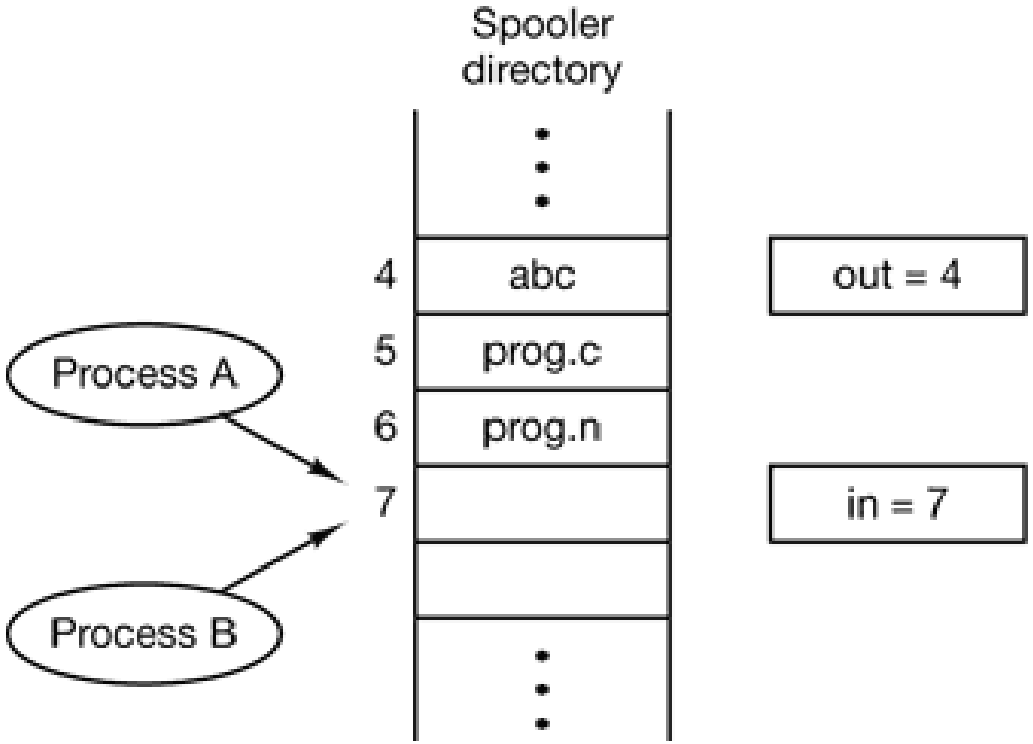
İletişim Niçin Yapılır ?

1. Kaynak paylaşımı (dosya, I/O aygıtı,...)
2. Karşılıklı haberleşme (iki proses birbirine haber gönderir)
3. Senkronizayon (Bir prosesin çalışması başka bir prosesin belirli işlemleri tamamlamış olmasına bağlı olabilir)

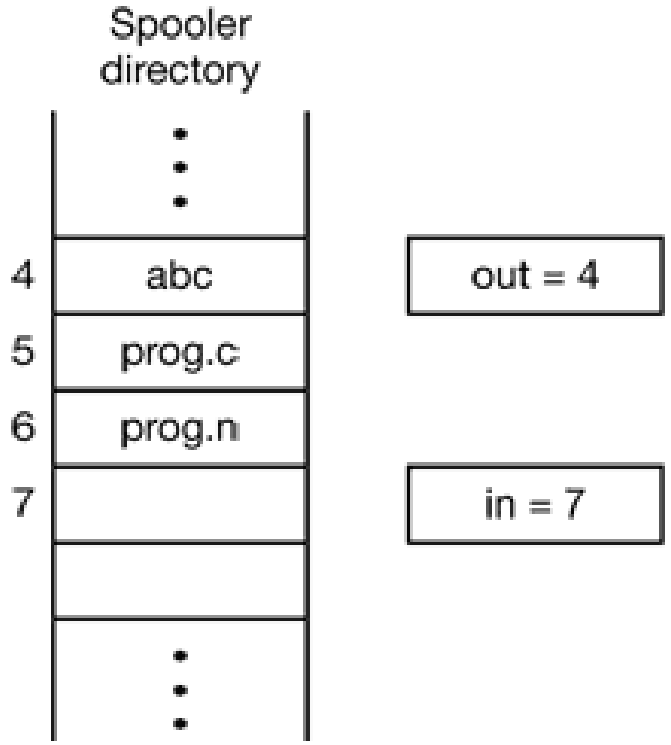
YARİŞ DURUMLARİ

- İki ya da daha fazla proses iletişim kurarken ortak bellek bölgelerini(shared memory), ortak dosyaları, tüm proseslerin erişebileceği kuyruk yapılarını, özel iletişim dosyalarını ve buna benzer sistemler kullanabilir.
- İletişime bir örnek verelim: prosesler istedikleri dosyaları yazıcıdan çıktı almak istesinler. Bir proses bir dosyayı yazdırmak istediğinde, bekletici(spooler) ismi verilen bir dosyaya yazdırmak istediği dosya bilgisini ekler.
- Başka bir proses belirli sürelerde periyodik olarak klasörü kontrol eder, eğer bir dosya görürse bu dosyayı yazdırır. Yazdırma işleminden sonra yazdırdığı dosyayı bekletici(spooler) klasöründen siler.

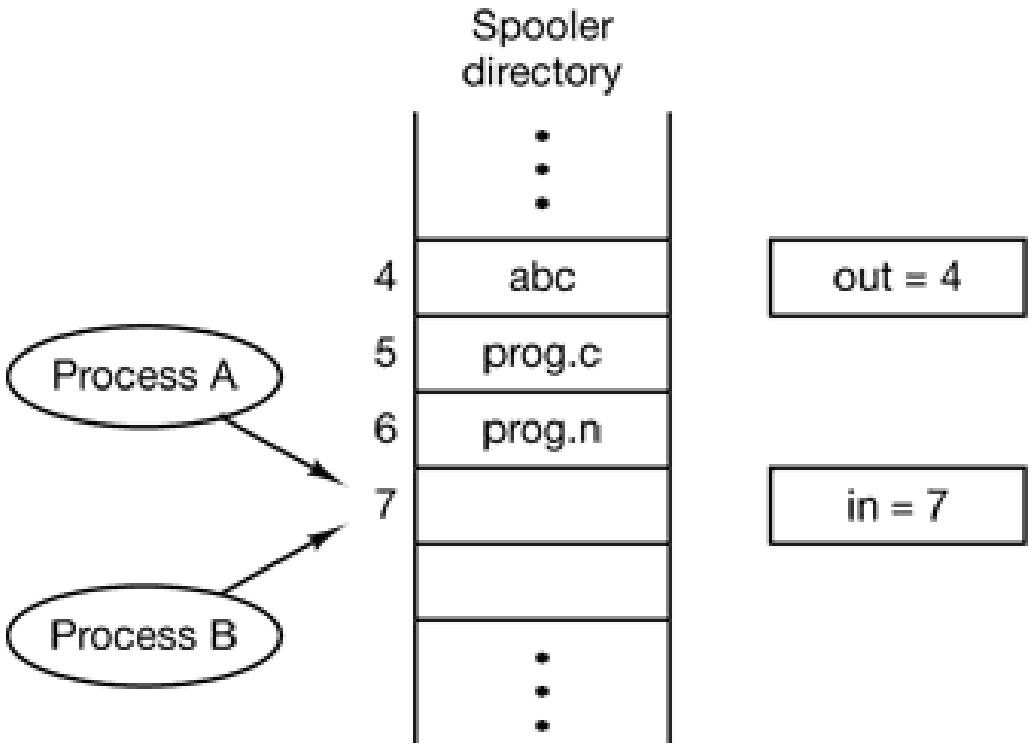
- Spooler dizininin 0,1,2, … diye numaralandırılmış birçok slotu olsun. Herbir slot, prosesler tarafından yazdırılmak için gönderilen bir dosyanın adını saklasın. ***out*** bir sonraki yazıdırılacak olan dosyanın adresini , ***in*** de spoolerdaki bir sonraki boş hücrenin adr



- A prosesi **in** değerini 7 olarak kendi lokal bir değişkeni olan local_in degerine 7 değerini atasın. Hemen sonra işletim sistemi anahtarlama yapıp, B prosesine geçsin. B prosesi de **in** değerini 7 olarak okuyacak ve kendi local_in değişkenine 7 degerini atayacaktır. B prosesi **in=7** adresine kendi yazdırmak istediği dosyanın bilgisini eklesin ve **in** değerini 8 yapsın. B prosesinin çalışması kesilsin ve tekrar A prosesine geri dönülsün.
- A prosesi bir sonraki boş hücre değerini yani **in** değişkeninin değerini 7 olarak görmektedir. A prosesi de yazdırmak istediği dosya bilgisini 7. hücreye ekler ve **in=8** yapar.



- B prosesinin yazdırmak istediği dosya bilgisi silinir.
- Bu tip durumlara YARIŞ DURUMLARI (RACE CONDITIONS) denilir.



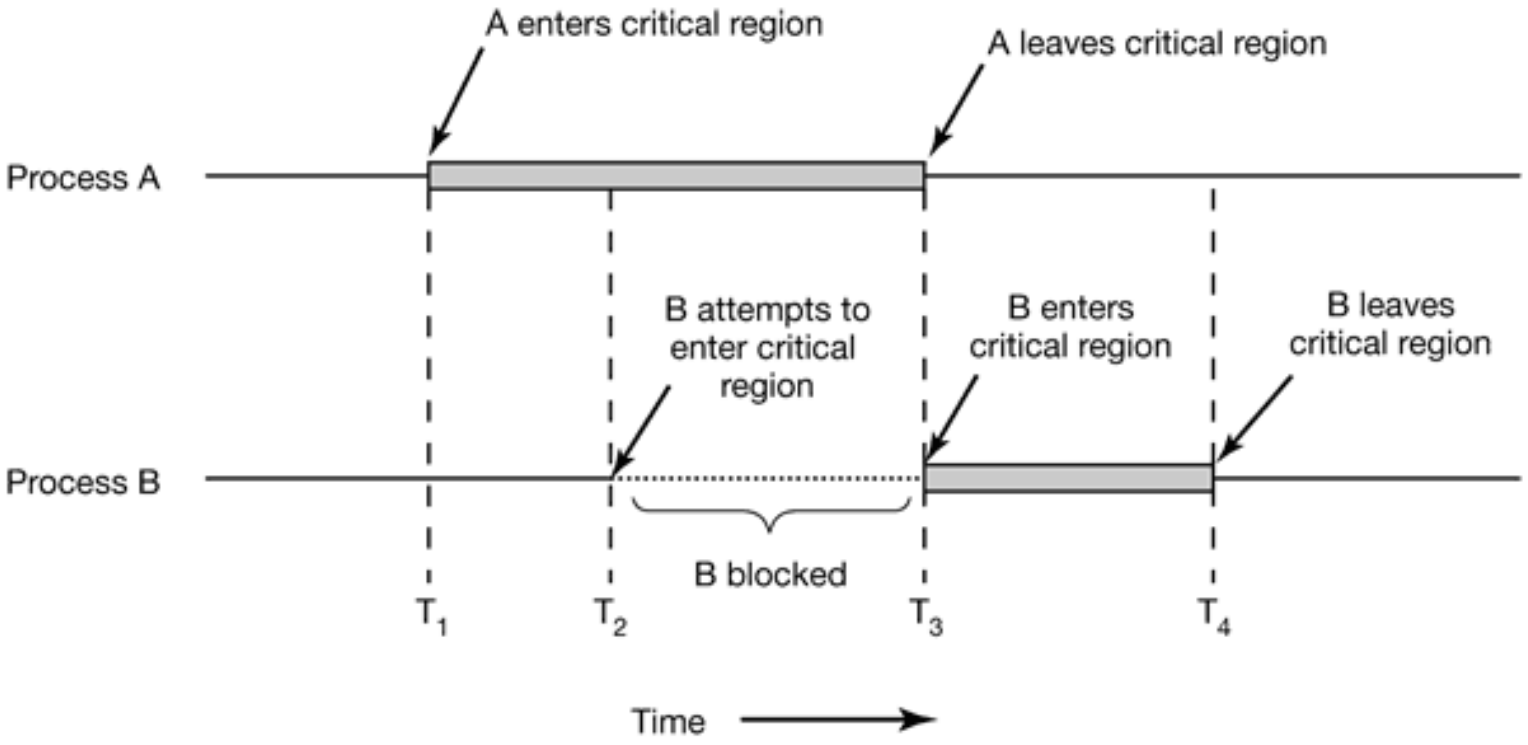
KRİTİK BOLGELER

- Bazen bir proses ortak kullanılan bir bellek bölgesine yada bir dosyaya ulaşmak durumunda kalabilir. Bu ortak alanda yarış durumlarının oluşmasına sebep olabilir. İşte bir programın ortak alanlara ulaşmaya çalıştığı kod kısımlarına **kritik bölge** adı verilir.
- Yarış durumlarından sakınmak için kullanılan yöntemlere genel olarak **Karşılıklı Dışlama** (mutual exclusion) yöntemleri denir. Karşılıklı Dışalama ile bir proses ortak değişken üzerinde işlem yapıyorsa başka bir proses bu değişken üzerinde işlem yapması engellenmiş olur.

- **Karşılıklı Dışlama** işletim sisteminin temel görevlerinden bir tanesidir.
- We need four conditions to hold to have a good solution for race conditions:

1. İki süreç aynı anda kritik bölge içerisine girmemelidir.No assumptions may be made about speeds or the number of CPUs.
2. Sistemdeki işlemci sayısı ve hızı ile ilgili kabuller yapılmamalıdır, bu değerlerden bağımsız olmalıdır
3. Kritik bölgede çalışmayan bir proses diğer prosesleri bloklalamalıdır.
4. Bir süreç kritik bölge içinde sonsuza kadar beklememelidir.

- T_1 anında A prosesi kritik bölgesine girer.
- T_2 anında B prosesi kritik bölgesine girmeye çalışır fakat başarısız olur ve T_3 anına kadar bloklanır.
- T_3 anında, B kritik bölgesine girer.
- T_4 anında, B kritik bölgeyi terk eder.



YOĞUN BEKLEMELİ KARŞİLİKLİ DİŞLAMA YÖNTEMLERİ

- Bu bölümde yoğun beklemeli olarak karşılıklı dışlamayı gerçekleştiren çeşitli yaklaşımlar incelenecektir. Bir proses kritik bölgesine girdiğinde, diğer proseslerine kritik bölgeye girmesi engellenecek, böylece ortak alanların kullanımı güvence altına alınacaktır.

1.KESMELERİ DEVRE DİŞİ BİRAKMAK

- Süreçler kritik bölgeye girdiklerinde tüm kesmeleri devre dışı bırakabilirler. Kritik bölge dışına çıktıklarında kesmeleri tekrar aktif ederler.
- Bu durumda eğer bir süreç bu işlemi uygulasa ve çıktığında eski durumuna getirmez ise sistem çalışmaz duruma gelir. Bu kontrolü kullanıcı programalarına vermek oldukça sakıncalı bir durumdur.
- Sistemde birden fazla işlemci varsa bu işlem sadece tek bir işlemciyi etkiler. Kullanıcı süreçlerine bu hakkı vermek oldukça tehlikelidir. Çekirdek bazen bu yöntemi kullanır.

- This approach is generally unattractive because it is not a good idea to give user processes the power to turn off interrupts.
- Suppose that one of them did it and never turned it on again? That could be the end of the system.
- Furthermore if the system is a multiprocessor, with two or more CPUs, disabling interrupts affects only the CPU that executed the disable instruction. The other ones will continue running and can access the shared memory.
- It is frequently proper for the kernel itself to disable interrupts for a few instructions while it is updating variables or lists.

2. LOCK VARİABLES

- This is a software solution. Consider having a single, shared (lock) variable, initially 0. When a process wants to enter its critical region, it first tests the lock. If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0. Thus, a 0 means that no process is in its critical region, and a 1 means that some process is in its critical region.
- This method also is not a good solution.Suppose that one process reads the lock and sees that it is 0. Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1. When the first process runs again, it will also set the lock to 1, and two processes will be in their critical regions at the same time.

3. STRİCT ALTERNATİON

- An integer variable *turn*, initially 0, keeps track of whose turn it is to enter the critical region and examine or update the shared memory.

Process 1

```
while (TRUE) {
    while (turn != 0)    /* loop */ ;
    critical_region();
    turn = 1;
    noncritical_region();
}
```

Process 2

```
while (TRUE) {
    while (turn != 1);   /* loop */ ;
    critical_region();
    turn = 0;
    noncritical_region();
}
```

- Continuously testing a variable until some value appears is called **busy waiting**. It should usually be avoided, since it wastes CPU time.

- When process 1 leaves the critical region, it sets *turn* to 1, to allow process 2 to enter its critical region. Suppose that process 2 finishes its critical region, so both processes are in their noncritical regions, with *turn* set to 0.
- Suddenly, process 2 finishes its noncritical region before process 1 finishes its noncritical region and goes back to the top of its loop. Unfortunately, it is not permitted to enter its critical region now, because *turn* is 0 and process 1 is busy with its noncritical region. It hangs in its while loop until process 1 sets *turn* to 1.
- This situation violates condition 3 set out above: process 1 is being blocked by a process not in its critical region.

```
while (TRUE) {
    while (turn != 0)    /* loop */ ;
    critical_region();
    turn = 1;
    noncritical_region();
}
```

```
while (TRUE) {
    while (turn != 1);  /* loop */ ;
    critical_region();
    turn = 0;
    noncritical_region();
}
```

4.PETERSON'S SOLUTİON

```
#define FALSE 0
#define TRUE  1
#define N     2       /* number of processes */

int turn;             /* whose turn is it? */
int interested[N];    /* all values initially 0 (FALSE) */

void enter_region(int process)      /* process is 0 or 1 */
{
    int other;                      /* number of the other process */

    other = 1 - process;            /* the opposite of process */
    interested[process] = TRUE;     /* show that you are interested */
    turn = process;                 /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */;
}

void leave_region (int process)     /* process, who is leaving */
{
    interested[process] = FALSE;    /* indicate departure from critical region
}
```

- Before using the shared variables each process calls *enter_region* with its own process number, 0 or 1, as parameter. This call may cause it to wait until it is safe to enter. After it has finished with the shared variables, the process calls *leave_region* to indicate that it is done and to allow the other process to enter.
- Let us see how this solution works. Initially neither process is in its critical region. Now process 0 calls *enter_region.* It indicates its interest by setting its array element and sets *turn* to 0. Since process 1 is not interested, *enter_region* returns immediately. If process 1 now calls *enter_region,* it will hang there until *interested*[0] goes to *FALSE*, an event that only happens when process 0 calls *leave_region* to exit the critical region.

- Now consider the case that both processes call *enter_region* almost simultaneously. Both will store their process number in *turn.* Whichever is stored last falls in loop; the first one is overwritten and lost. Suppose that process 1 stores last, so *turn* is 1. When both processes come to the while statement, process 0 executes it zero times and enters its critical region. Process 1 loops and does not enter its critical region until process 0 exits its critical region.

5. THE TSL INSTRUCTİON

- Many computers, especially those designed with multiple processors in mind, have an instruction:

  TSL RX,LOCK

- (Test and Set Lock) that works as follows. It reads the contents of the memory word *lock* into register RX and then stores a nonzero value at the memory address *lock.* The operations of reading the word and storing into it are guaranteed to be indivisible—no other processor can access the memory word until the instruction is finished. The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

- When *lock* is 0, any process may set it to 1 using the TSL instruction and then read or write the shared memory. When it is done, the process sets *lock* back to 0 using an ordinary move instruction.

```
enter_region:
    TSL REGISTER,LOCK   | copy lock to register and set lock to 1
    CMP REGISTER,#0     | was lock zero?
    JNE enter_region    | if it was non zero, lock was set, so loop
    RET | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0        | store a 0 in lock
    RET | return to caller
```

SLEEP AND WAKEUP

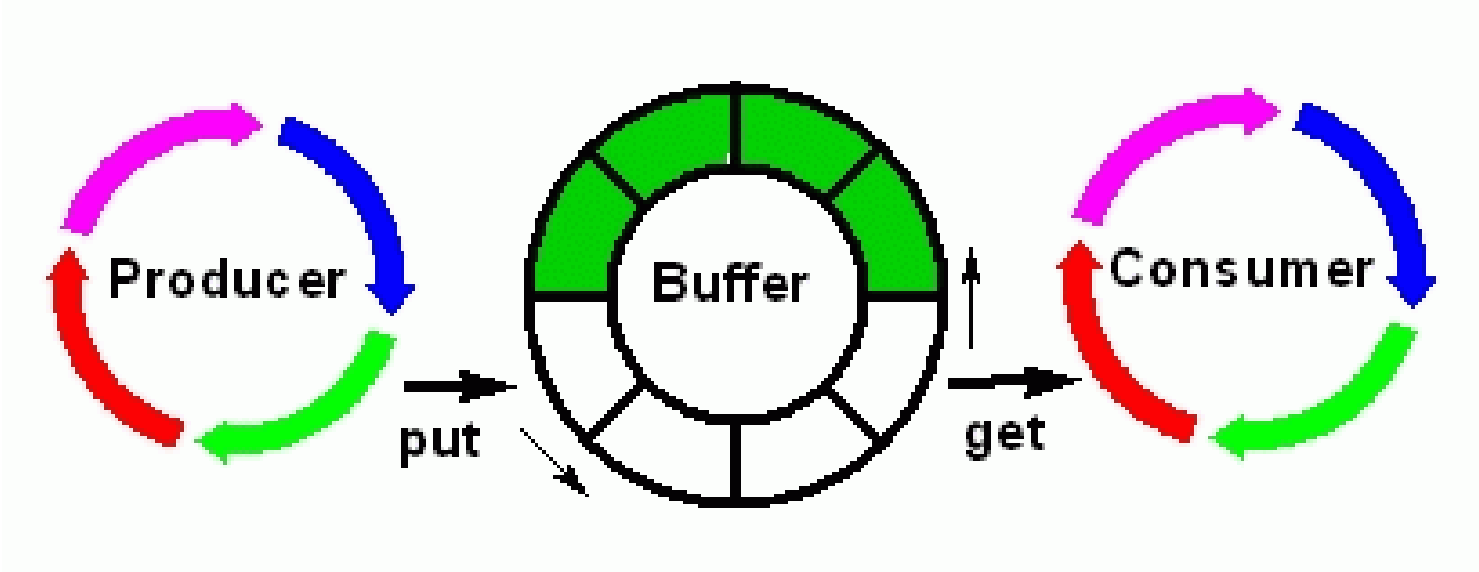
- Both Peterson's solution and the solution using TSL are correct, but both have the defect of requiring busy waiting. When a process wants to enter its critical region, it checks to see if the entry is allowed. If it is not allowed, the process just sits in a tight loop waiting until it is allowed.
- Beside of wasting CPU time, this approach can also have unexpected effects. Consider a computer with two processes, *H*, with high priority and *L*, with low priority. The scheduling rules are such that *H* runs whenever it is in ready state. At a certain moment, with *L* is in its critical region, *H* becomes ready to run. *H* now begins busy waiting. Before *H* is completed, *L* can not be scheduled. So *L* never gets the chance to leave its critical region, so *H* loops forever.

- Now let me look at some interprocess communication primitives that block processes when they are not allowed to enter their critical regions, instead of wasting CPU time in an empty loop.

- One of the simplest primatives is the pair **sleep** and **wakeup**. Sleep is a system call that causes the caller to block, the caller is suspended until another process wakes it up. The wakeup call has one parameter, the process to be awakened.

THE PRODUCER-CONSUMER PROBLEM

- As an example of how these primitives can be used, let us consider the **producer-consumer** problem. Two processes share a common, fixed-size buffer. One of them, the producer, puts information into the buffer, and the other one, the consumer, takes it out.



- Trouble arises when the producer wants to put a new item in the buffer, but it is already full. The solution is for the producer to go to sleep, and sleeping producer will be awakened when the consumer has removed one or more items. Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up.

```
#define N 100          /* number of slots in the buffer */
int count = 0;         /* number of items in the buffer */

void producer (void)
{
    int item;

    while (TRUE) {                  /* repeat forever */
        item = produce_item();      /* generate next item */
        if (count == N) sleep();    /* if buffer is full, go to sleep */
        insert_item(item);          /* put item in buffer */
        count = count + 1;          /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);  /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                  /* repeat forever */
        if (count == 0) sleep();    /* if buffer is empty, got to sleep */
        item = remove_item();       /* take item out of buffer */
        count = count - 1;          /* decrement count of items in buffer */
        if (count == N - 1) wakeup(producer); /* was buffer full? */
        consume_item(item);         /* print item */
    }
}
```

- This approach sounds simple enough, but it leads to the same kinds of race conditions we saw earlier with the spooler directory. Race conditon can occur because access to *count* is unconstrained.

- The following situation could possibly occur. The buffer is empty and the consumer has just read *count* to see if it is 0. At that instant, the scheduler decides to stop running the consumer temporarily and switches to the producer. The producer inserts an item in the buffer, increments *count*, and notices that it is now 1. Because *count* was just 0, the consumer must be sleeping and the producer calls *wakeup* to wake the consumer up.

- But the consumer is not yet logically asleep, so the wakeup signal is lost. When the consumer next runs, it will test the value of *count* and find it to be 0, and go to sleep. Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever.
- A **wakeup waiting bit** can be added to fix this problem. When a wakeup is sent to a process that is still awake, this bit is on. Later, when the process tries to go to sleep, if the wakeup waiting bit is on, the process will not sleep and stay awake. Then **wakeup waiting bit** will be turned off. But if there are 8,16 or 32 consumers and producers, it is needed to add **wakeup waiting bit** for every consumers and producers.

SEMAPHORES

- Semaphore is an integer variable to count the number of wakeup processes saved for future use. A semaphore could have the value 0, indicating that no wakeup processes were saved, or some positive value if one or more wakeups were pending.
- There are two operations, down (wait) and up(signal) (generalizations of sleep and wakeup, respectively). The down operation on a semaphore checks if the value is greater than 0. If so, it decrements the value and just continues. If the value is 0, the process is put to sleep without completing the down for the moment.

- The up operation increments the value of the semaphore addressed. If one or more processes were sleeping on that semaphore, unable to complete an earlier down operation, one of them is chosen by the system (e.g., at random) and is allowed to complete its down.

- Checking the value, changing it and possibly going to sleep is as a single indivisible **atomic action**. It is guaranteed that when a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked.

SOLVİNG THE PRODUCER-CONSUMER PROBLEM USİNG SEMAPHORES

- Semaphores solve the lost-wakeup problem. It is essential that they are implemented in an indivisible way. The normal way is to implement up and down as system calls, with the operating system briefly disabling all interrupts while it is testing the semaphore, updating it, and putting the process to sleep.
- If multiple CPUs are being used, each semaphore should be protected by a lock variable, with the TSL instruction used to make sure that only one CPU at a time examines the semaphore.

- This solution uses three semaphores: one called *full* for counting the number of slots that are full, one called *empty* for counting the number of slots that are empty, and one called *mutex* to make sure the producer and consumer do not access the buffer at the same time. *Full* is initially 0, *empty* is initially equal to the number of slots in the buffer (N), and *mutex* is initially 1.
- The *mutex* semaphore is used for mutual exclusion. It is designed to guarantee that only one process at a time will be reading or writing the buffer and the associated variables. The other use of semaphores is for **synchronization**. The *full* and *empty* semaphores are used to guarantee synchronization. In this case, they ensure that the producer stops running when the buffer is full, and the consumer stops running when it is empty.

```
#define N 100                   /* number of slots in the buffer */

typedef int semaphore;          /* semaphores are a special kind of int */
semaphore mutex = 1;            /* controls access to critical region */
semaphore empty = N;            /* counts empty buffer slots */
semaphore full = 0;             /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {              /* TRUE is the constant 1 */
        item = produce_item(); /* generate something to put in buffer */
        down(&empty);           /* decrement empty count */
        down(&mutex);           /* enter critical region */
        insert_item(item);      /* put new item in buffer */
        up(&mutex);             /* leave critical region */
        up(&full);              /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {              /* infinite loop */
        down(&full);            /* decrement full count */
        down(&mutex);           /* enter critical region */
        item a= remove_item(); /* take item from buffer */
        up(&mutex);             /* leave critical region */
        up(&empty);             /* increment count of empty slots */
        consume_item(item);     /* do something with the item */
```

MUTEXES

- Mutex is simplified version of the semaphore. When the semaphore's ability to count is not needed, mutex can be used. Mutexes are good for managing mutual exclusion to some shared resource or piece of code. They are easy and efficient to implement.
- A **mutex** is a variable that can be in one of two states: unlocked or locked. Consequently, only 1 bit is required to represent it, but in practice an integer often is used, with 0 meaning unlocked and all other values meaning locked.

- Two procedures are used with mutexes. When a thread (or process) needs access to a critical region, it calls *mutex_lock*. If the mutex is current unlocked (meaning that the critical region is available), the call succeeds and the calling thread is free to enter the critical region.

- If the mutex is already locked, the calling thread is blocked until the thread in the critical region is finished and calls *mutex_unlock*. If multiple threads are blocked on the mutex, one of them is chosen at random and allowed to acquire the lock.

- Because mutexes are so simple, they can easily be implemented in user space if a TSL instruction is available

```
mutex_lock:
    TSL REGISTER,MUTEX | copy mutex to register and set mutex to 1
    CMP REGISTERS,#0   | was mutex zero?
    JZE ok             | if it was zero, mutex was unlocked, so return
    CALL thread_yield  | mutex is busy; schedule another thread
    JMP mutex_lock     | try again later
ok:   RET | return to caller; critical region entered

mutex_unlock:
    MOVE MUTEX,#0      | store a 0 in mutex
    RET | return to caller
```

- The code of *mutex_lock* is similar to the code of *enter_region* of TSL Instruction but with a crucial difference.
- The difference is valid for threads. When *enter_region* fails to enter the critical region, it keeps testing the lock repeatedly (busy waiting). In *mutex_lock, w*hen the later fails to acquire a lock, it calls *thread_yield* to give up the CPU to another thread. Consequently there is no busy waiting.

```
mutex_lock:
    TSL REGISTER,MUTEX | copy mutex to register and set mutex to 1
    CMP REGISTERS,#0   | was mutex zero?
    JZE ok             | if it was zero, mutex was unlocked, so return
    CALL thread_yield  | mutex is busy; schedule another thread
    JMP mutex_lock     | try again later
ok:   RET | return to caller; critical region entered
```

MESSAGE PASSİNG

- Interprocess communication uses two primitives, send and receive. They can easily be put into library procedures, such as

  send(destination, &message);

  receive(source, &message);

- The former call sends a message to a given destination and the latter one receives a message from a given source. If no message is available, the receiver can block until one arrives. Alternatively, it can return immediately with an error code.

- Message passing systems have many problems, especially if the communicating processes are on different machines connected by a network. For example, messages can be lost on the network. To solve this problem, as soon as a message has been received, the receiver will send back a special **acknowledgement** message. If the sender has not received the acknowledgement within a certain time interval, it retransmits the message.
- Now consider what happens if the message itself is received correctly, but the acknowledgement is lost. The sender will retransmit the message, so the receiver will get it twice. It is essential that the receiver is able to distinguish a new message from the retransmission of an old one.

- Usually, this problem is solved by putting consecutive sequence numbers in each original message. If the receiver gets a message bearing the same sequence number as the previous message, it knows that the message is a duplicate that can be ignored.

- There are also design issues that are important when the sender and receiver are on the same machine. One of these is performance. Copying messages from one process to another is always slower than doing a semaphore operation.

THE PRODUCER-CONSUMER PROBLEM WİTH MESSAGE PASSİNG

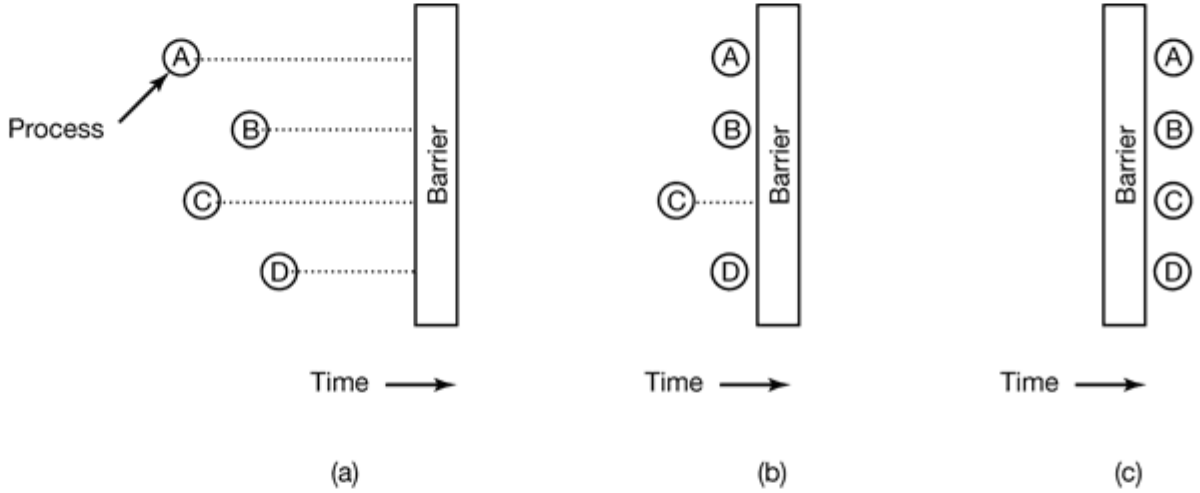
- We assume that all messages are the same size and that messages sent but not yet received are buffered automatically by the operating system. In this solution, a total of *N* messages are used, analogous to the *N* slots in a shared memory buffer. The consumer starts out by sending *N* empty messages to the producer. Whenever the producer has an item to give to the consumer, it takes an empty message and sends back a full one.
- If the producer works faster than the consumer, all the messages will be full, waiting for the consumer: the producer will be blocked, waiting for an empty to come back. If the consumer works faster, all the messages will be empty waiting for the producer to fill them up: the consumer will be blocked, waiting for a full message.

```
#define N 100     /* number of slots in the buffer */
void producer(void)
{
    int item;
    message m;    /* message buffer */

    while (TRUE) {
        item = produce_item( );        /* generate something to put in buffer */
        receive(consumer, &m);         /* wait for an empty to arrive */
        build_message (&m, item);      /* construct a message to send */
        send(consumer, &m);            /* send item to consumer */
    }
}

void consumer(void) {
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
    while (TRUE) {
        receive(producer, &m);          /* get message containing item */
        item = extract_item(&m);        /* extract item from message */
        send(producer, &m);             /* send back empty reply */
        consume_item(tem);              /* do something with the item */
    }
}
```

BARRİERS

- Our last synchronization mechanism is intended for groups of processes rather than two-process producer-consumer type situations. Some applications are divided into phases and have the rule that no process may proceed into the next phase until all processes are ready to proceed to the next phase. This behavior may be achieved by placing a **barrier** at the end of each phase. When a process reaches the barrier, it is blocked until all processes have reached the barrier.

- We see four processes approaching a barrier (Fig a). After a while, the first process finishes all the computing during the first phase. It then executes the barrier primitive, generally by calling a library procedure. The process is then suspended. A little later, a second and then a third process finish the first phase (Fig b). Finally, when the last process, *C*, hits the barrier, all the processes are released (Fig c)



ANY QUESTION?