# פרק 6
# סנכרון תהליכים

***מרצה: אליאב מנשה***

# Background

- A ***cooperating process*** is one that can **affect** or **be affected** by other processes executing in the system.
  - Cooperating → **concurrent access to shared data**.
    - The data section of threads.
    - Shared memory among processes.
  - **May result in data inconsistency**.
- In this chapter, we discuss various mechanisms to ensure the orderly execution of cooperating processes.
  - Data consistency is maintained.

# Background

- In Ch. 3.4.1 (shared-memory systems), we present the concept of cooperating process by using the classical ***producer-consumer problem***.
  - ***Producer process*** produces items on a shared, bounded buffer.
  - ***Consumer process*** consumes the produced information from the buffer.

```
while (TRUE)
{ /* produce an item and put
  in nextProduced */
  while (count == BUFFER_SIZE)
    ; // do nothing
  buffer [in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
  count++;
}
```

***producer***

```
while (TRUE)
{
  while (count == 0)
    ; // do nothing
  nextConsumed =  buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  count--;
  /*  consume the item in
  nextConsumed */
}
```

***consumer***

Although both the producer and consumer routines are correct separately, they may not function correctly when executed concurrently.

- Suppose that the value of the variable `counter` is 5.
- The producer and consumer processes execute the statements "`counter++`" and "`counter--`" concurrently.
- Then the value of the variable of counter may be 4, 5, or 6!!

Why?

- The statement "`counter++` (or `counter--`)" may be implemented i machine language as:

```
register1 = counter
register1 = register1 + 1
counter = register1
```

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- The **concurrent execution** of "`counter++`" and "`counter--`" is equivalent to a sequential execution where the lower-level statements presented previously are **interleaved in some arbitrary order**.

But the order within each high-level statement is preserved.

- For instance:

| | | | |
|---|---|---|---|
| $T_0$: producer | execute | $register_1 = count$ | $\{register_1 = 5\}$ |
| $T_1$: producer | execute | $register_1 = register_1 + 1$ | $\{register_1 = 6\}$ |
| $T_2$: consumer | execute | $register_2 = count$ | $\{register_2 = 5\}$ |
| $T_3$: consumer | execute | $register_2 = register_2 - 1$ | $\{register_2 = 4\}$ |
| $T_4$: producer | execute | $count = register_1$ | $\{count = 6\}$ |
| $T_5$: consumer | execute | $count = register_2$ | $\mathbf{\{count = 4\}}$ |

- We arrive at this incorrect state because we allowed both processes to manipulate the variable `counter` concurrently.

- ***Race condition***:
  - Several processes access and manipulate the same data concurrently.
  - The outcome of the execution depends on the particular order in which the access takes place.

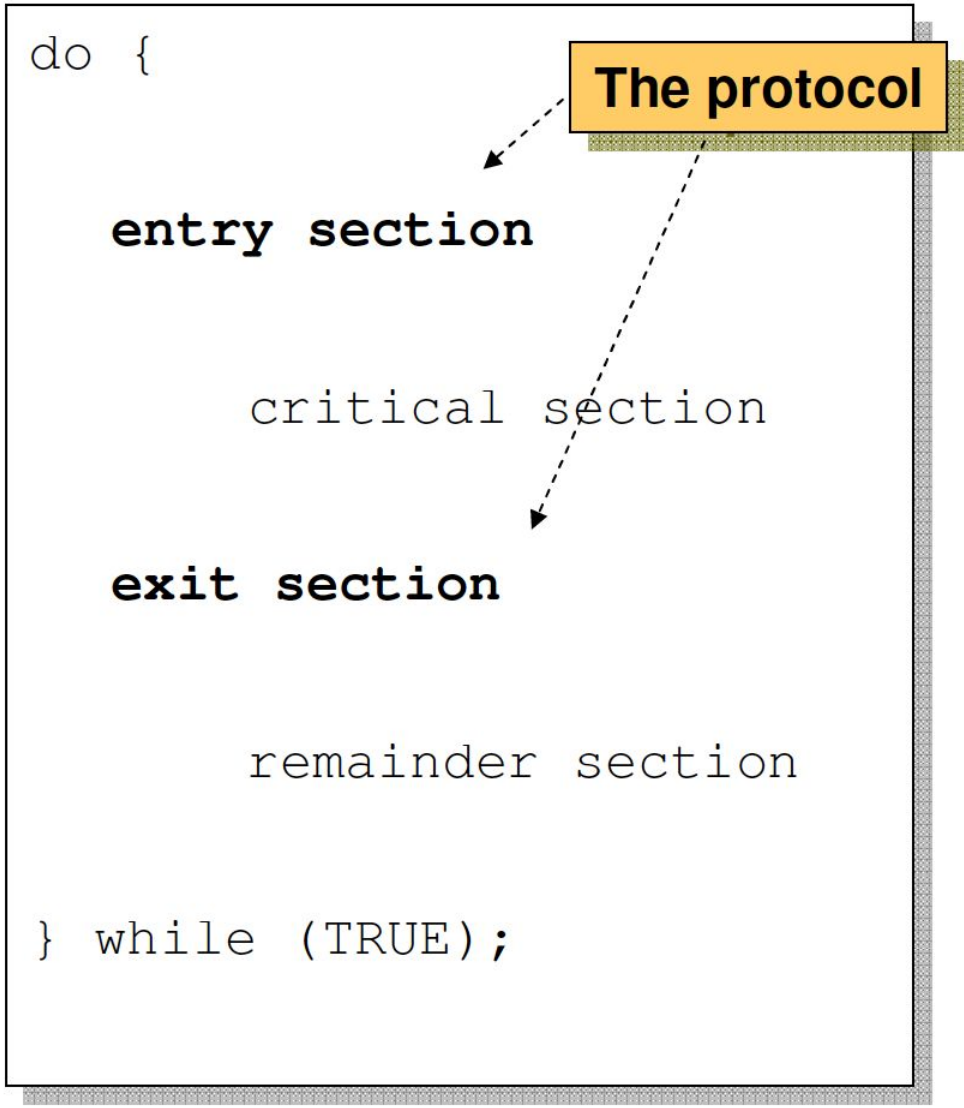
- To guard against the race condition mentioned previously, we need to ensure that **only one process at a time** can be manipulating the variable `counter`.

- Race conditions occur frequently in operating systems.
  - Clearly, we want the resulting changes not to interfere with one another.
  - In this chapter, we talk about *process synchronization* and *coordination*.

# Critical Section

- ***Critical section***:
  - A segment of code where a process may be changing common (shared) variables, updating a table, writing a file, and so on.
- Consider a system consisting of *n* processes:
  - Each process has a critical section.
  - Clearly, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section.
- The critical-section problem is to design a ***protocol*** that the processes can use to cooperate.
  - That is, no two processes are executing in their critical sections at the same time.

- For a critical-section problem solution:
  - Each process must request permission to enter its critical section.
    - The section of code implementing this request is the ***entry section***.
  - The critical section may be followed by an ***exit section***.
  - The remainder code is the **remainder section**.



- A solution to the critical-section problem must satisfy the following three requirements:

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 only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section.
   - The selection 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.

- The code implementing an operating system (kernel code) is subject to several possible race conditions.
  - For instance:
    - A kernel data structure that maintains a list of all open files in the system.
    - If two processes were to open files simultaneously, the updates to this list could result in a race condition.
  - Other kernel data structures: structures for maintaining memory allocation, for maintaining process lists …
- The kernel developers have to ensure that the operating system is free from such race conditions.

- Two approaches, used to handle critical sections in operating systems:
  - ***Nonpreemptive kernels***:
    - Does not allow a process running in kernel mode to be preempted.
    - Is thus free from race conditions on kernel data structures, as only one process is active in the kernel at a time.
  - ***Preemptive kernels***:
    - Allow a process to be preempted while it is running in kernel mode.
    - The kernel must be carefully designed to ensure that shared kernel data are free from race conditions.

- Why prefer preemptive kernel?
  - Less risk that a kernel-mode process will run for an arbitrarily long period.
  - Allow a real-time process to preempt a process currently running in the kernel.

- Windows XP and Windows 2000 are nonpreemptive kernels.
- Several commercial versions of UNIX (including Solaris and IRIX) and Linux 2.6 kernel (and the later version) are preemptive.

# Peterson's Solution

- A **software-based** solution to the critical-section problem.
- Is restricted to **two processes** that alternate execution between their critical section and remainder sections.
  - The processes are numbered $P_0$ and $P_1$, or $P_i$ and $P_j$.
- The method requires two data items to be shared between the two processes:

```
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!

- The Peterson's solution for process $P_i$. ( j = 1 – i ) ($P_0$ and $P_1$)

```
do {
    flag[i] = TRUE;
    turn = j;
    while ( flag[j] && turn == j);

    CRITICAL SECTION

    flag[i] = FALSE;

    REMAINDER SECTION

} while (TRUE);
```

If other process ($P_j$) wishes to enter the critical section, it can do so.

- If both processes try to enter at the same time …
  - `turn` will be set to both *i* and *j* at roughly the same time.
  - The eventual value of turn decides which can go.

- To prove the method is a solution for the critical-section problem, we need to show:

1. **Mutual exclusion is preserved**.
   - 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 only those processes that are not executing in their remainder section can participate in the decision on which will enter its critical section next and this selection 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

- To prove the method is a solution for the critical-section problem, we need to show:

1. **Mutual exclusion is preserved**.
   - $P_i$ enters its critical section only if either `flag[j]== false` or `turn==i`.
   - If both processes want to enter their critical sections at the same time, then `flag[i] == flag[j] == true`.
   - However, the value of `turn` can be either 0 or 1 **but cannot be both**.
   - Hence, one of the processes must have successfully executed the while statement (to enter its critical section), and the other process has to wait, till the process leaves its critical section. → *mutual exclusion is preserved*.

2. **The progress requirement is satisfied**.

- Case 1:
  - $P_i$ is ready to enter its critical section.
  - If $P_j$ is not ready to enter the critical section (it is in the remainder section).
  - Then `flag[j] == false`, and $P_i$ can enter its critical section.

- Case 2:
  - $P_i$ and $P_j$ are both ready to enter its critical section.
  - `flag[i] == flag[j] == true`.
  - Either `turn == i` or `turn == j`.
  - If `turn == i`, then $P_i$ will enter the critical section.
  - If `turn == j`, then $P_j$ will enter the critical section.

3. **The bounded-waiting requirement is met**.

- Once $P_j$ exits its critical section, it will reset `flag[j]` to `false`, allowing $P_i$ to enter its critical section.

- Even if $P_j$ **immediately** resets `flag[j]` to `true`, it must also set `turn` to `i`.

- Then, $P_i$ will enter the critical section after at **most one entry** by $P_j$.

# Synchronization Hardware

- We can state that any solution to the critical-section problem requires a simple tool – a ***lock***.
  - A process must acquire a lock before entering a critical section.
  - It releases the lock when it exits the critical section.

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

- Here, we follow the above concept and explore solutions to the critical-section problem *using hardware techniques*.

- Many systems provide hardware support for critical section code.
- Uniprocessors – could **disable interrupts**.
  - Currently running code would execute **without preemption**.
- Modern machines provide special **atomic hardware instructions**.
  - **Atomic → uninterruptible or indivisibly.**
  - ***Test and set*** the content of a word, atomically.
  - ***Swap*** the contents of two words, atomically.
  - Can be used to design solutions of the critical-section problem in a relatively simple manner.

# פעולות אטומיות

**פעולה אטומית** היא רצף של פקודות מכונה או פקודת מכונה יחידה שיבוצעו על ידי מעבד, מבלי שיכולה להיעשות החלפת הקשר בזמן ביצוען ומבלי שרכיב אחר במחשב יוכל להבחין או לשנות משהו בכל מצב ביניים אלא רק במצב ההתחלתי או הסופי של הפעולה

https://he.wikipedia.org/wiki/%D7%A4%D7%A2%D7%95%D7%9C%D7%94_%D7%90%D7%98%D7%95%D7%9E%D7%99%D7%AA

מעבדים שונים במחשבים שונים מאפשרים פעולות אטומיות שונות.
הפעולות הנפוצות הן:

- קידום - הוספת 1 או החסרת מיחידת זיכרון (++ או --)
- השמה - החלפת ערך ביחידת זיכרון לערך חדש.
- השווה והחלף - החלפת ערך ביחידת זיכרון לערך חדש רק במידה ויש ערך מסוים ביחידת הזיכרון לפני ההחלפה. פעולה זו יכולה להיכשל.
- פעולות אריתמטיות - הפעלת פעולה אריתמטית על יחידת זיכרון (כגון הוספה, החסרה, הכפלה וכדומה).
- פעולות ביטים - הפעלת פעולת ביטים על יחידת זיכרון (כגון and, or, xor וכדומה).

רוב הפעולות האטומיות מחזירות את הערך הקודם שהיה ביחידת הזיכרון לפני הפעולה

- The **atomic *TestAndSet()*** instruction:

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

- We can implement "*mutual exclusion*" by declaring a Boolean variable `lock`, initialized to `false`.

- Then the structure of process $P_i$ is:

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

- The **atomic *`Swap()`*** instruction.

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

- We can implement "*mutual exclusion*" by declaring a Boolean variable `lock`, initialized to `false`.

- Then the structure of process $P_i$ is:

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

- Do these algorithms (using `TestAndSet()` and `Swap()`) satisfy the bounded-waiting requirement? NO!!
  - But they do satisfy the mutual-exclusion requirement.
- An algorithm using the `TestAndSet()` instruction that satisfies all the critical-section requirements.
  - The method requires two data items to be shared between *n* processes:

```
Boolean waiting[n];
Boolean lock;
```

`waiting[i]` is true is $P_i$ is waiting.

  - All initialized to false.

```
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);
```

$P_i$ can enter its critical section only if either `waiting[i] == false` or `key == false`.

The first process to execute the `TestAndSet()` will find `key == false`.
All others must wait.

Find the next waiting process if any.

If no waiting process, release the lock.

Hold the lock, and $P_j$ is granted to enter its critical section.

- The mutual-exclusion requirement is met:
  - When a process leaves its critical section, only one `waiting[j]` is set to false.
- The progress requirement is met:
  - A process exiting the critical section either sets `lock` to `false` or sets `waiting[j]` to `false`.
  - Both allow a process that is waiting to enter its critical section to proceed.
- The bounded-waiting requirement is met:
  - When a process leaves its critical section, it scans the array `waiting` in the cyclic ordering.
  - Any waiting process will thus do so **within *n-1* turns**.

# Semaphores

- The hardware-based solutions are complicated for application programmers to use.

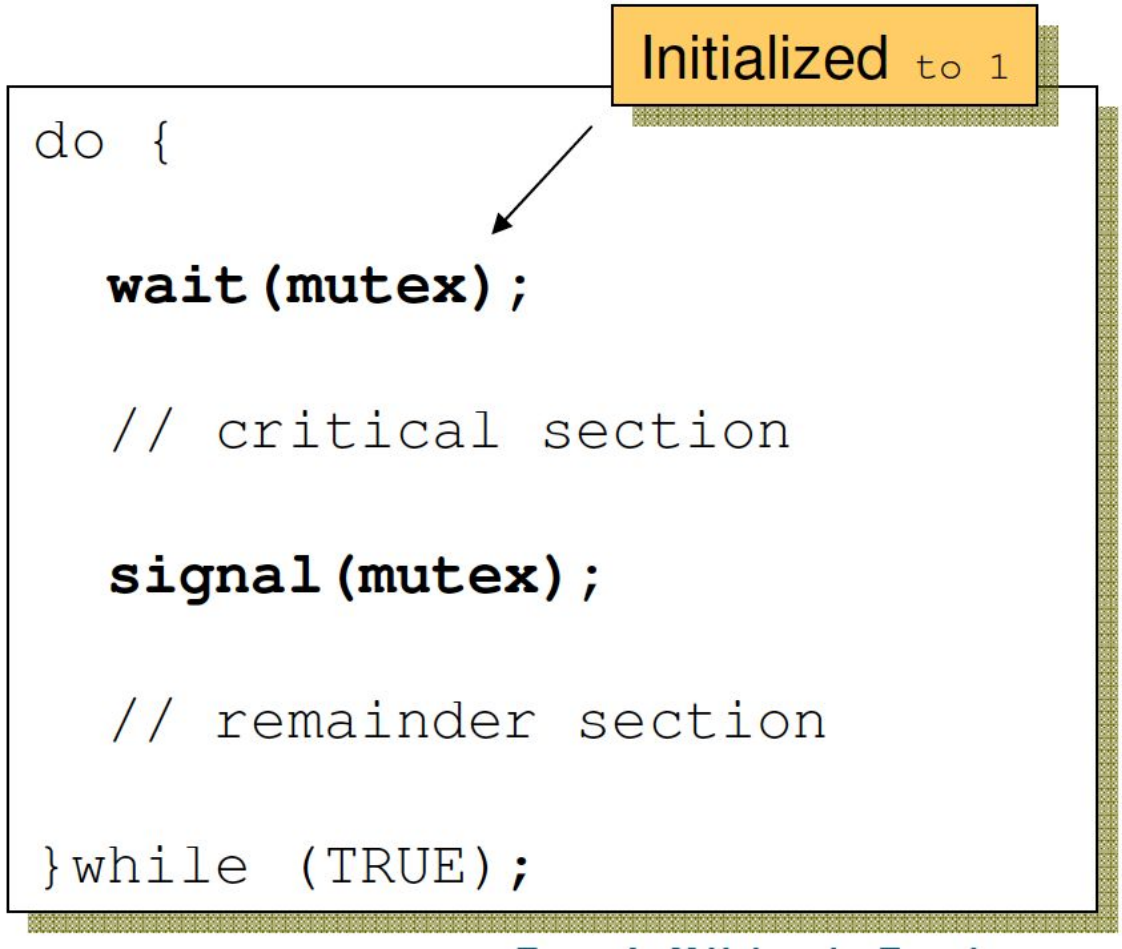
- ***Semaphore*** – a simple solution tool.
  - Is an integer.
  - Apart from initialization, is accessed only through two standard **atomic** operations: `wait()` and `signal()`.
  - When one process modifies the semaphore value, no other process can simultaneously modify the same semaphore value.

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

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

All the modifications (and testing) to the integer value of the semaphore must be executed **indivisibly**.

- ***Binary semaphore***:
  - Also known as ***mutex*** locks (***mut***ual ***ex***clusion).
  - Range only between 0 and 1.
  - Used to deal with the critical-section problem for multiple processes.



# Semaphores – Usage

- ***Counting semaphores***.
  - Range over an unrestricted domain.
  - Used to control access to a given resource consisting of a finite number of instances.
    - The semaphore is initialized to the number of resources available.
    - Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count).
    - When a process releases a resource, it performs a signal() operation (incrementing the count).
    - When the count for the semaphore goes to 0, all resources are being used; processes that wish to use a resource will block until the count becomes greater than 0.

- Semaphores can be used to solve synchronization problems.
  - 2 concurrently running processes: $P_1$ with a statement $S_1$ and $P_2$ with a statement $S_2$.
  - We require that $S_2$ be executed only after $S_1$ has completed.
  - A common semaphore `synch`, initialized to 0.

```
S1;
signal(synch);
```

$P_1$

```
wait(synch);
S2;
```

$P_2$

$P_2$ will execute $S_2$ only after $P_1$ has invoked `signal(sync)`.

# Semaphores – Implementation

- The main disadvantage of the semaphore definition is that it requires ***busy waiting***.
  - Also called a ***spinlock***; because the process spins while waiting for the lock.
  - A waiting process must loop continuously in the entry code.
  - Waste CPU cycles that some other process might be able to use productively.
  - However, spinlocks are useful on multiprocessor systems when locks are expected to be held for short times.
    - One thread can “spin” on one processor while another thread performs its critical section on another processor.

- To overcome the need for busy waiting.
  - In `wait()`:
    - When a process finds that the semaphore value is not positive, rather than engaging in busy waiting, **it blocks itself**.
    - The block operation places a process into a waiting queue (**waiting state**) associated with the semaphore.
    - The CPU scheduler then selects another process to execute.
  - In `signal()`:
    - When a process executes a `signal()` operation, a waiting process is restarted by a wakeup operation.
    - It changes the process from the **waiting state** to the **ready state** and place the process in the ready queue.
    - However, the CPU **may** or **may not** be switched to the newly ready process, depending on the CPU-scheduling algorithm.

- We define a semaphore as a "C" `struct`:

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

Each semaphore has an integer value and a list of processes list. When a process must wait on a semaphore, it is added to the list of processes.

Can be the link field in PCBs.

- The `wait()` semaphore operation:

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

The block() operation suspends the process that invokes it.

- The `signal()` semaphore operation:

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

The magnitude is the number of waiting process.

The wakeup() operation resumes the execution of a blocked process P.

- The list can use any queuing strategy, such as FIFO.

- We must guarantee that `wait()` and `signal()` are atomic.
  - No two processes can execute `wait()` and `signal()` operations on the same semaphore at the same time.

- In a single-processor environment, we can do it by **inhibiting interrupts** during the time of the operations are executing.



  - So that, instructions from different processes cannot be interleaved.
  - Until interrupts are reenabled and the scheduler can regain control.

# Deadlocks and Starvation

- ***Deadlock*** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
  - Let `S` and `Q` be two semaphores initialized to 1

| $P_0$ | $P_1$ |
|---|---|
| `wait (S);` | `wait (Q);` |
| `wait (Q);` | `wait (S);` |
| . | . |
| . | . |
| . | . |
| `signal (S);` | `signal (Q);` |
| `signal (Q);` | `signal (S);` |

Suppose that $P_0$ executes wait(S) and then $P_1$ executes wait(Q). When $P_0$ executes wait(Q), it must wait until $P_1$ executes signal(Q).

Similarly, when $P_1$ executes wait(S), it must wait until $P_0$ executes signal(S). Since these signal() operations cannot be executed, $P_0$ and $P_1$ are deadlocked.

# Starvation

- ***Starvation*** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.
  - May occur with a semaphore in LIFO (last-in, first-out) order.

# Classic Problems of Synchronization

# Classic Problems of Synchronization

- Next, a number of synchronization problems as examples of a large class of concurrency-control problems will be presented.
- These problems are used for testing nearly every newly proposed synchronization scheme.
- In our solutions to the problems, semaphores will be used for synchronization.

# The Bounded-Buffer Problem

- We assume that the pool consists of $n$ buffers, each capable of holding one item.
- The `mutex` semaphore provides mutual exclusion for accesses to the buffer pool.
  - Initialized to the value 1.
- The `empty` and `full` semaphores count the number of empty and full buffers.
  - The semaphore `empty` is initialized to the value $n$.
  - The semaphore `full` is initialized to the value 0.

- The structure of producer process:

```
do {
   ...
   // produce an item in nextp
   ...

   wait(empty);
   wait(mutex);

   // add nextp to buffer

   signal(mutex);
   signal(full);

} while (TRUE);
```

- The structure of the consumer process:

```
do {
   wait(full);
   wait(mutex);

   // remove an item from buffer
   // to nextc

   signal(mutex);
   signal(empty);

   // consume the item in nextc


} while (TRUE);
```

- Note the symmetry between the producer and consumer
- The producer is producing full buffers for the consumer and the consumer producing empty buffers for the producer.

# The Readers-Writers Problem

- A database is to be shared among several concurrent processes.
  - Some processes may want only to read the database – **readers**.
  - Others may want to update (to read and write) – **writers**.
- If two readers access the shared data simultaneously, no adverse affects will result.
- However, if a writer and some other thread (either a reader or a writer) access the database simultaneously, chaos may ensue.
- We require that the writers have **exclusive access** to the shared database.

- The readers-writers problem has several variations, all involving priorities.
  - The *first* readers-writers problem:
    - No reader will be kept waiting unless a writer has already obtained permission to use the shared object.
    - Or, no reader should wait for other readers to finish.
    - Or readers have higher priorities.
  - The *second* readers-writers problem:
    - Once a writer is ready, the writer performs its write as soon as possible.
- Here, we present a solution to the first readers-writers problem.

- The **reader processes** share the following data structures:

```
semaphore mutex, wrt;
int readcount;
```

  - `readcount` is initialized to 0.
    - Keep track of how many processes are currently reading the database.
  - `mutex` and `wrt` are initialized to 1.
    - `mutex` is used to ensure mutual exclusion when the variable `readcount` is updated.
    - `wrt`, **shared with writers**, functions as a mutual-exclusion semaphore for the writers.

- The structure of a writer process:

```
do {
   wait(wrt);

   ...
   // writing is performed
   ...

   signal(wrt);

} while(TRUE);
```

- The structure of a reader process:

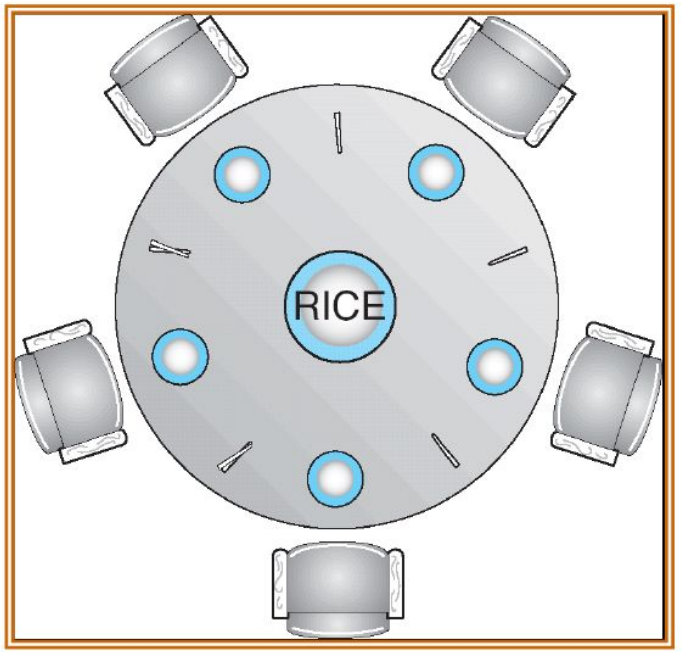
```
do {
   wait(mutex);
   readcount++;
   if (readcount == 1)
     wait(wrt);
   signal(mutex);

   ...
   // reading is performed
   ...

   wait(mutex);
   readcount--;
   if (readcount == 0)
     signal(wrt);
   signal(mutex);

} while(TRUE);
```

Note that if a writer is in the critical section and n readers are waiting, then one reader is queued on wrt, and n – 1 readers are queued on mutex.

Note that when a writer executes `signal(wrt)`, we may resume either the waiting readers or a **single** waiting writer.

# The Dining-Philosopher Problem

- Five philosophers spend their lives **thinking** and **eating**.
- The table is laid with five **single** chopsticks.
- When a philosopher gets hungry, she tries to pick up the two chopsticks that are closet to her.
  - Only one chopstick can be picked up at a time.
  - Obviously, she can not pick up a chopstick that is already in the hand of a neighbor.
- When a philosopher is finished eating, she puts down both of her chopsticks and starts thinking again.



- This problem is considered a classic synchronization problem.
  - It represents the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

- A plausible solution:
  - To represent each chopstick with a semaphore.

    ```
    semaphore chopstick[5];
    ```

    - All initialized to 1.
  - A philosopher tries to grab a chopstick by executing a `wait()` operation on that semaphore.
  - She releases her chopsticks by executing the `signal()`.

- The structure of philosopher *i*:

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);

   ...
   // eat
   ...

   signal(chopstick[i]);
   signal(chopstick[(i+1)%5]);

   ...
   // think
   ...

}while (TRUE);
```

- What is the problem with the last solution??
  - **DEADLOCK!!**

- Suppose that all five philosophers become hungry simultaneously and each grabs her left chopstick.
  - When each philosopher tries to grab her right chopstick, she will be delayed forever.

- Possible remedies:
  - At most four philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up her chopsticks only if both chopstick are available.
  - Asymmetric method – an odd philosopher picks up first her left chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

**But … the deadlock-free solution does not Necessarily eliminate the possibility of starvation.**

# שאלה

What is the meaning of the term busy waiting? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.

Answer: Busy waiting means that a process is waiting for a condition to be satisfied in a tight loop without relinquish the processor. Alternatively, a process could wait by relinquishing the processor, and block on a condition and wait to be awakened at some appropriate time in the future. Busy waiting can be avoided but incurs the overhead associated with putting a process to sleep and having to wake it up when the appropriate program state is reached.

# שאלה

Explain why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems?

Answer: Spinlocks are not appropriate for single-processor systems because the condition that would break a process out of the spinlock could be obtained only by executing a different process. If the process is not relinquishing the processor, other processes do not get the opportunity to set the program condition required for the first process to make progress. In a multiprocessor system, other processes execute on other processors and thereby modify the program state in order to release the first process from the spinlock.

# שאלה

Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems?

Answer: Interrupts are not sufficient in multiprocessor systems since disabling interrupts only prevents other processes from executing on the processor in which interrupts were disabled; there are no limitations on what processes could be executing on other processors and therefore the process disabling interrupts cannot guarantee mutually exclusive access to program state.