Process management

By,
Nabaraj Bahadur Negi

Topics Covered

- Threads
- Inter process communication(IPC)
- Implementing mutual exclusion
- Classical IPC problems

Thread

- A **thread (or lightweight process)**
  - It consists of : program counter, register set and stack space.
  - A **system registers** which hold its current working variables, a **stack** which contains the execution history and **counter** that keeps track of which instruction to execute next**.**
  - A thread shares the following with peer threads:
    - Code section, data section and OS resources (open files, signals)
    - No protection between threads
  - Collectively called a task.
  - May have own **PBC**
- Traditional (heavyweight) processes have a single thread of control - there is one program counter, and one sequence of instructions that can be carried out at any given time.

- **Multithreading:** a single program made up of a number of different concurrent activities .
- An application typically is implemented as a separate process with several threads of control.
- A Web browser might have one thread display images or text while another thread retrieves data from the network.
- For Example, **A word processor** may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.
- In certain situations, a single application may be required to perform several similar tasks. For example, a **Web server** accepts client requests for Web pages, images, sound, and so forth. A busy Web server may have several clients concurrently accessing it.

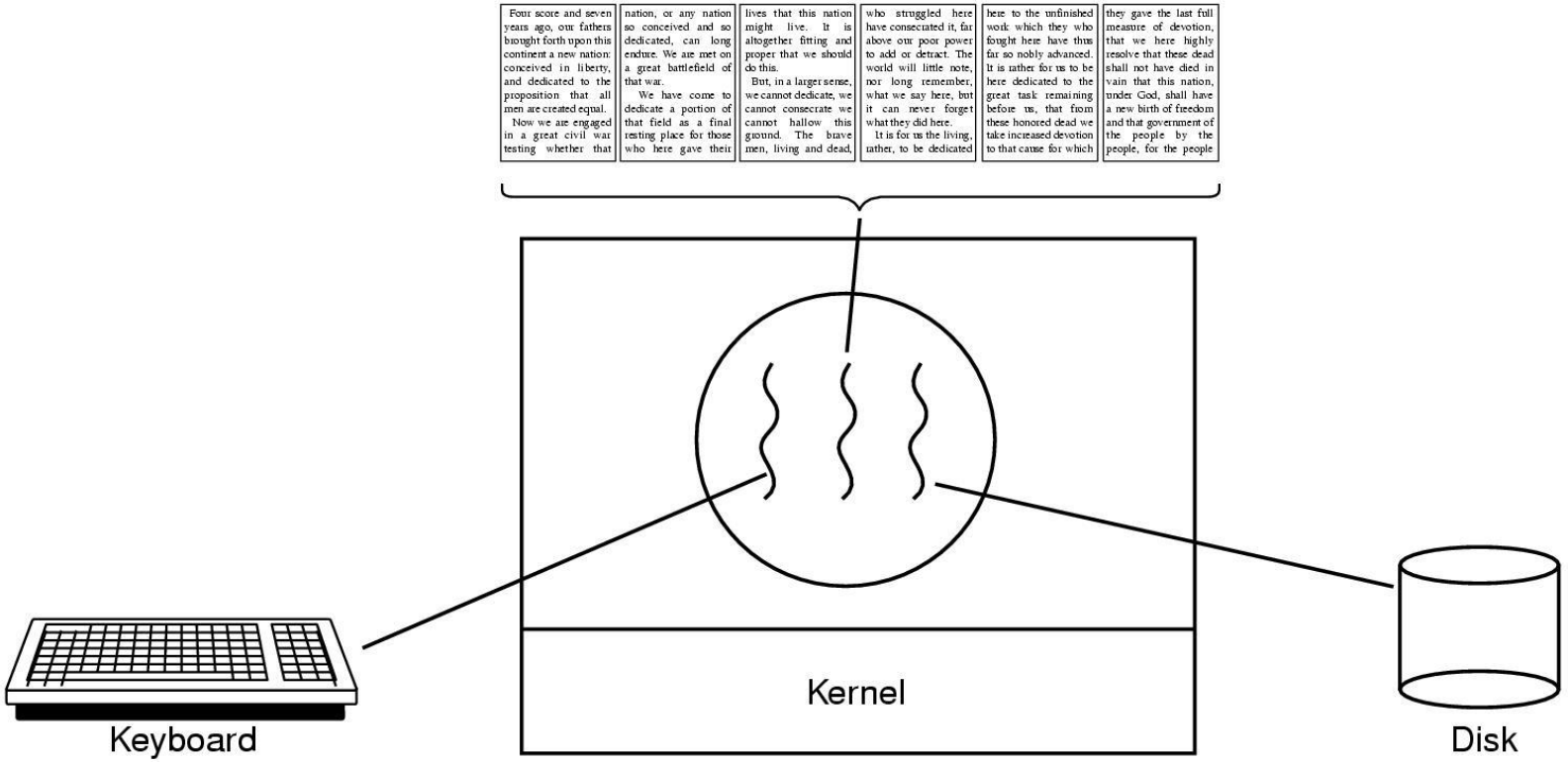


Figure. A word processor with three threads

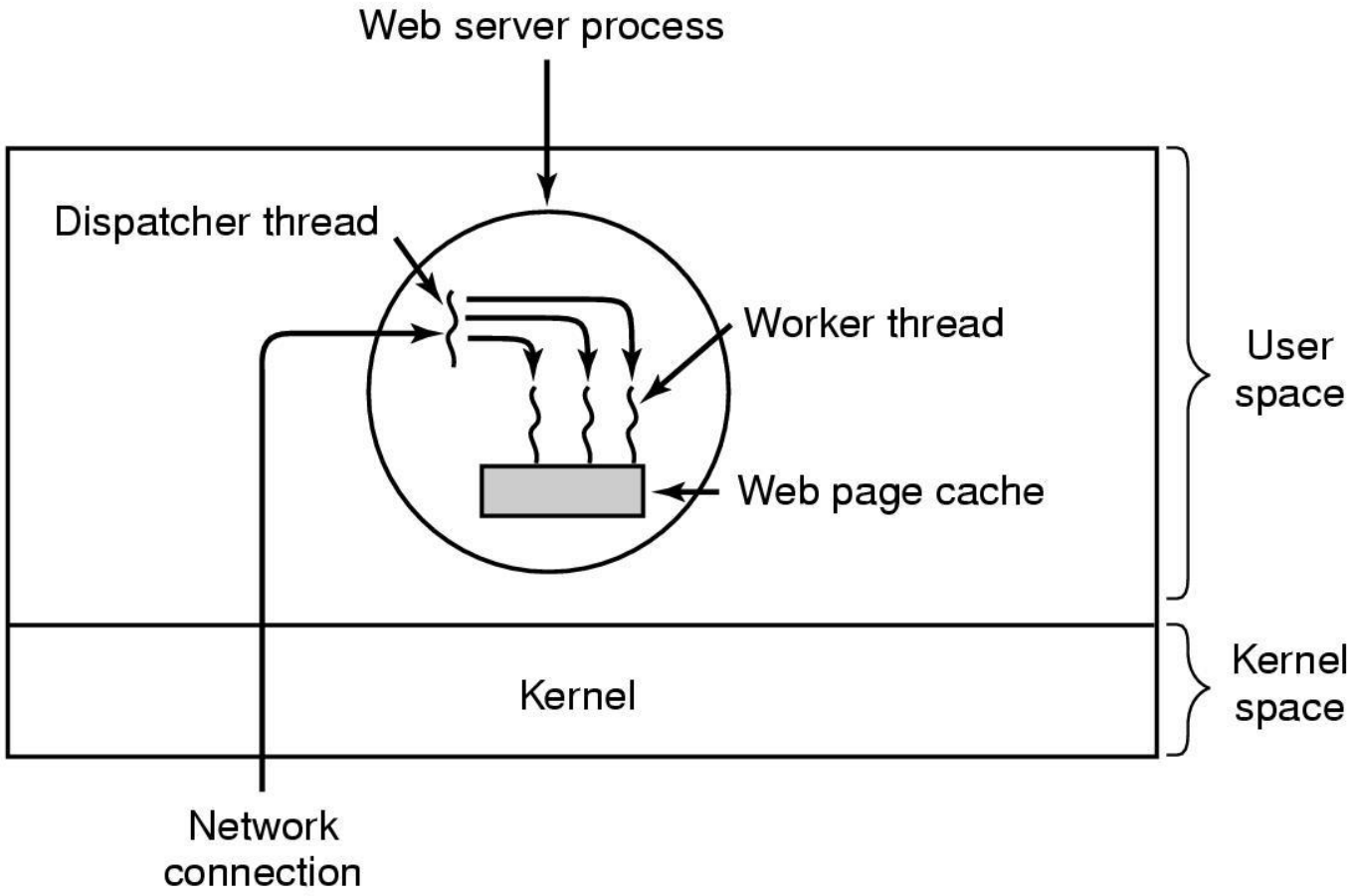


Figure. A multithreaded Web server

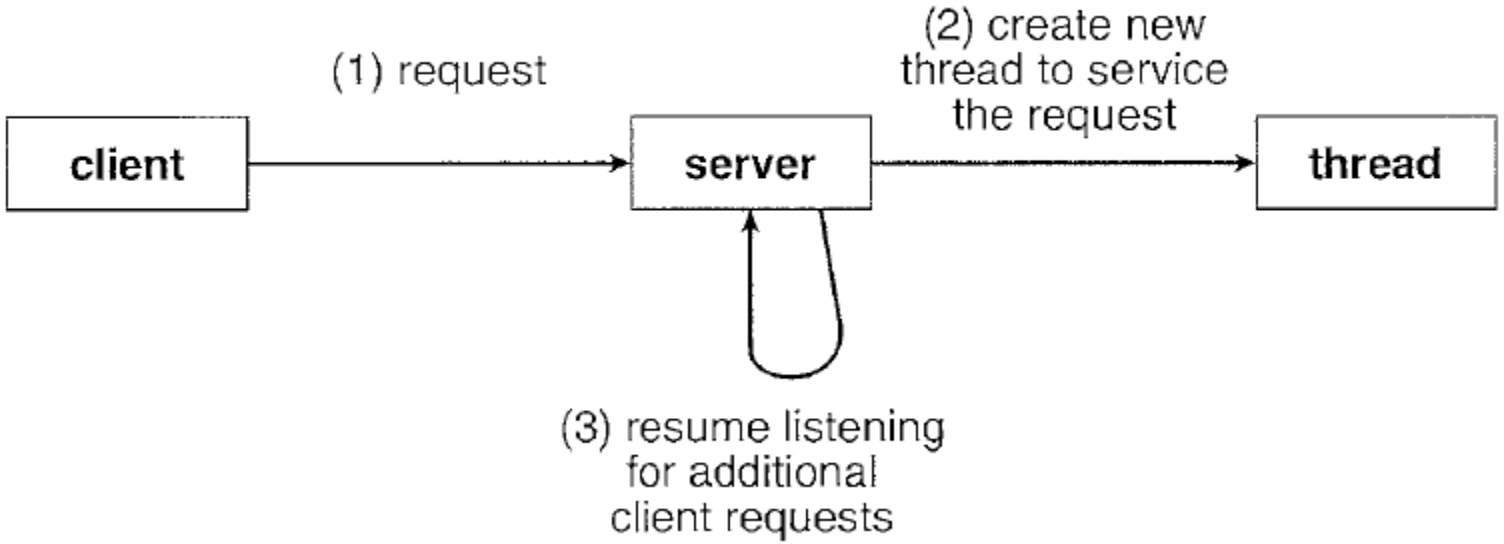


Figure .Multithreaded server architecture.

- Multi-threaded applications have multiple threads within a single process, each having their own program counter, stack and set of registers, but sharing common code, data, and certain structures such as open files.

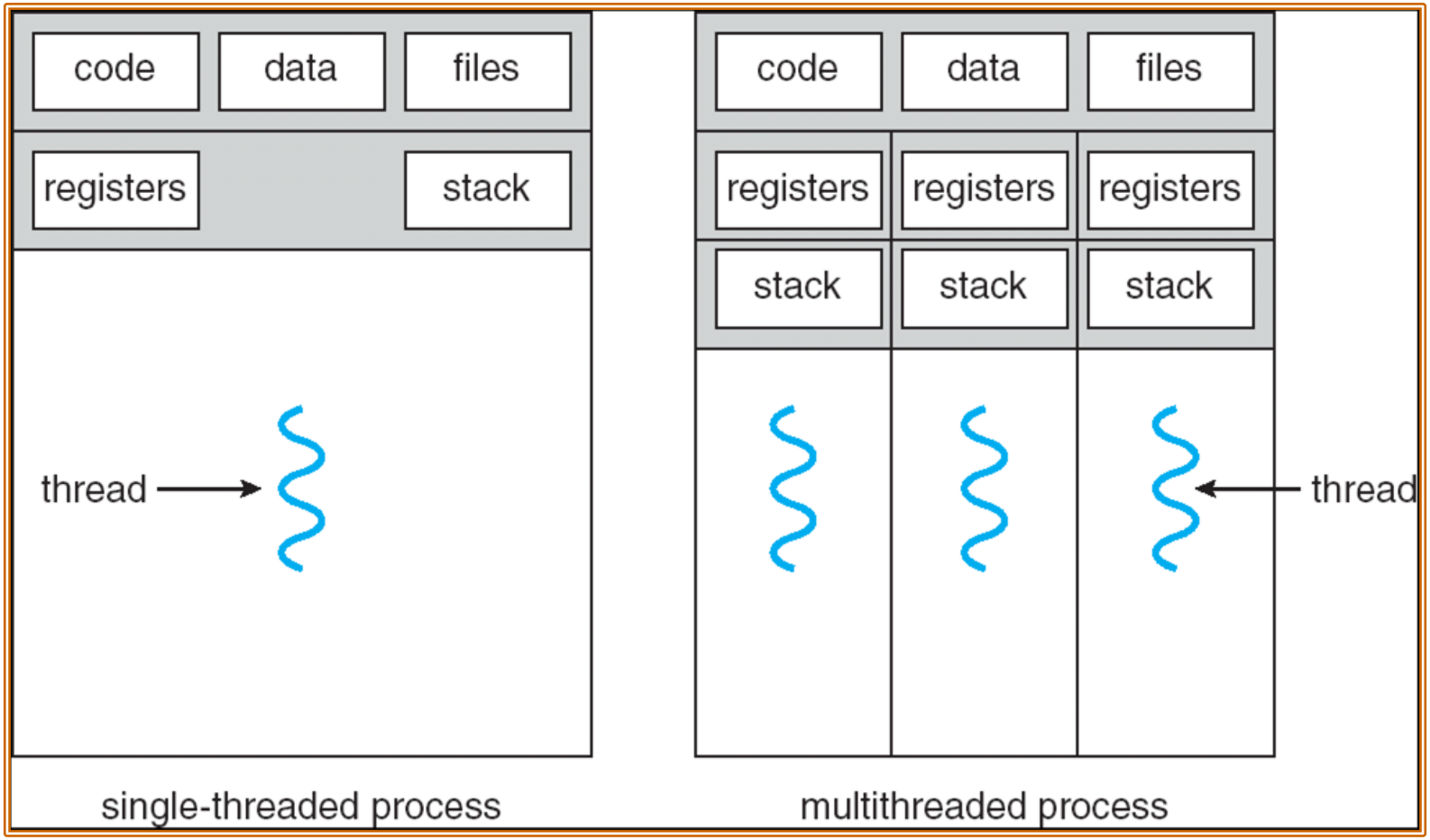


Figure .Single-threaded and multithreaded processes.

Multicore Programming

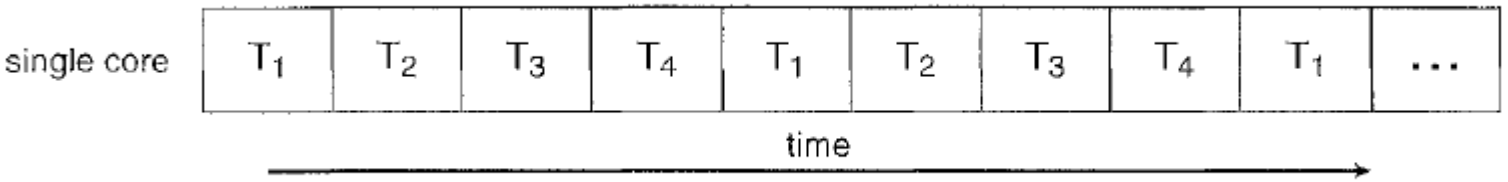


Figure .Concurrent execution on a single-core system.

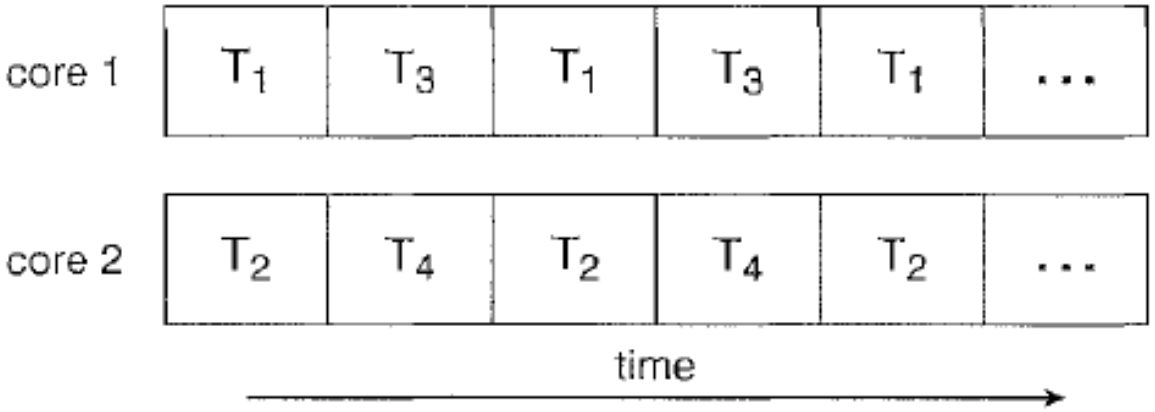


Figure . Parallel execution on a multicore system.

- A recent trend in computer architecture is to produce chips with multiple **cores**, or CPUs on a single chip.
- A multi-threaded application running on a traditional single-core chip would have to interleave the threads,
- On a multi-core chip, the threads could be spread across the available cores, allowing true parallel processing.

Types of Threads

- User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system.

User-level threads

- User-level threads are implemented by users and the kernel is not aware of the existence of these threads.
- User level thread is also called many-to-one mapping thread because the operating system maps all threads in a multithreaded process to a single execution context. The operating system considers each multithreaded processes as a single execution unit.
- User level thread uses space for thread scheduling. The threads are transparent to the operating system

- User threads are implemented by users and managed entirely by the run-time system.
- If one user level thread performs blocking operation then entire process will be blocked.
- Implementation is by a thread library at the user level
- User level threads are created by runtime libraries that cannot execute privileged instructions.
- User-level threads requires non-blocking systems call i.e., a multithreaded kernel. Otherwise, entire process will blocked in the kernel, even if there are runnable threads left in the processes. For example, if one thread causes a page fault, the process blocks.
- Example : **Java thread, POSIX threads**.

Kernel-supported threads

- Supported by the kernel.
- Threads are constructed and controlled by system calls. The system knows the state of each thread
- Kernel level thread support one-to-one thread mapping. Mapping requires each user thread with kernel thread.
- Kernel level threads are implemented by Operating system. In the kernel level thread, thread management is done by kernel.
- Implementation of kernel thread is complicated.
- Since, kernel managing threads, kernel can schedule another thread if a given thread blocks rather than blocking the entire processes.
- Kernel level threads are slower than user level threads
- Examples: Windows XP/2000,Solaris 2,Linux,Mac OS X

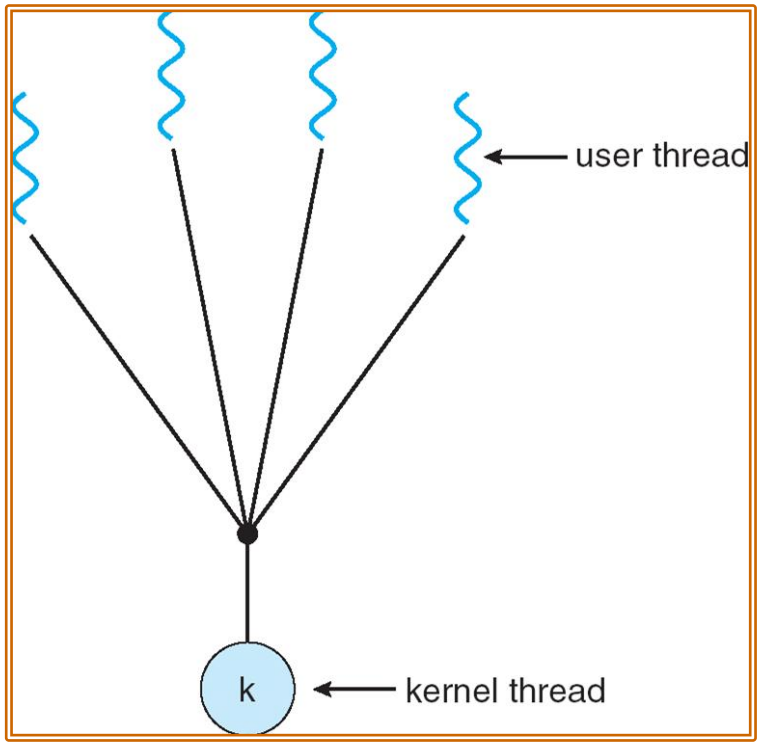
Thread libraries: Thread libraries provide programmers with an API for creating and managing threads.

- Thread libraries may be implemented either in user space or in kernel space. Library entirely in user space, with no kernel support. The latter involves system calls, and requires a kernel with thread library support.
- Three primary thread libraries:
  - **POSIX Ptheads:** may be provided as either a user or kernel library, as an extension to the POSIX standard.
  - **Win32 threads:** provided as a kernel-level library on Windows systems.
  - **JAVA threads:** Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pthreads or Win32 threads depending on the system.

Multithreading Models

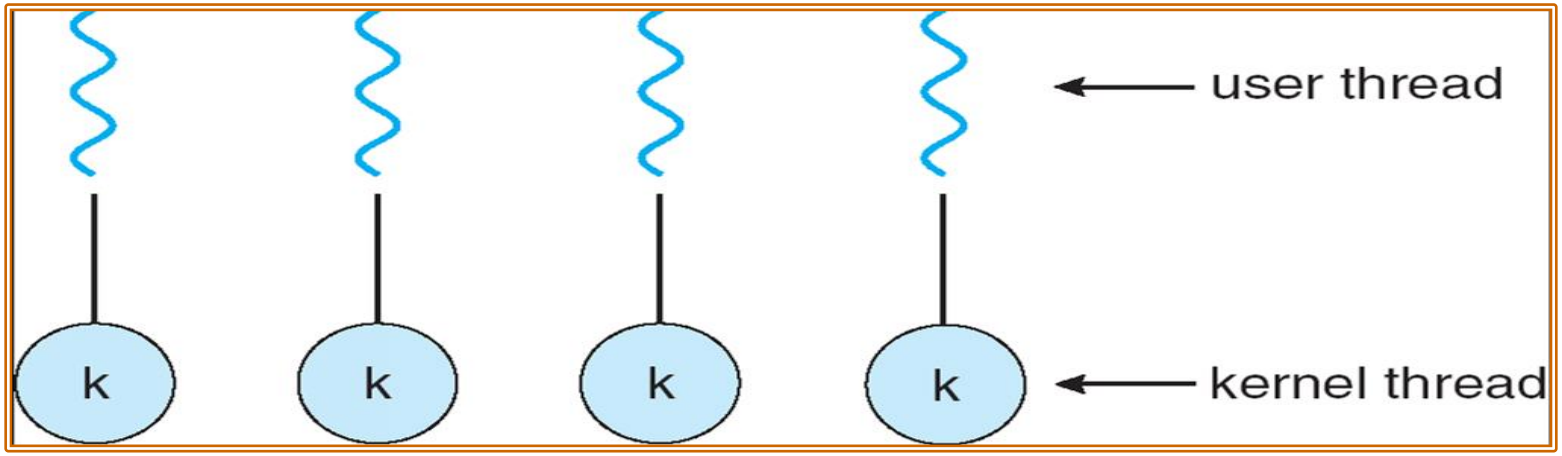
Many-to-One

- Many user-level threads mapped to single kernel thread
- Thread management is done by the thread library in user space
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads
- If one thread initiates a blocking system call, then no other thread in the same process can execute
- No parallelism.



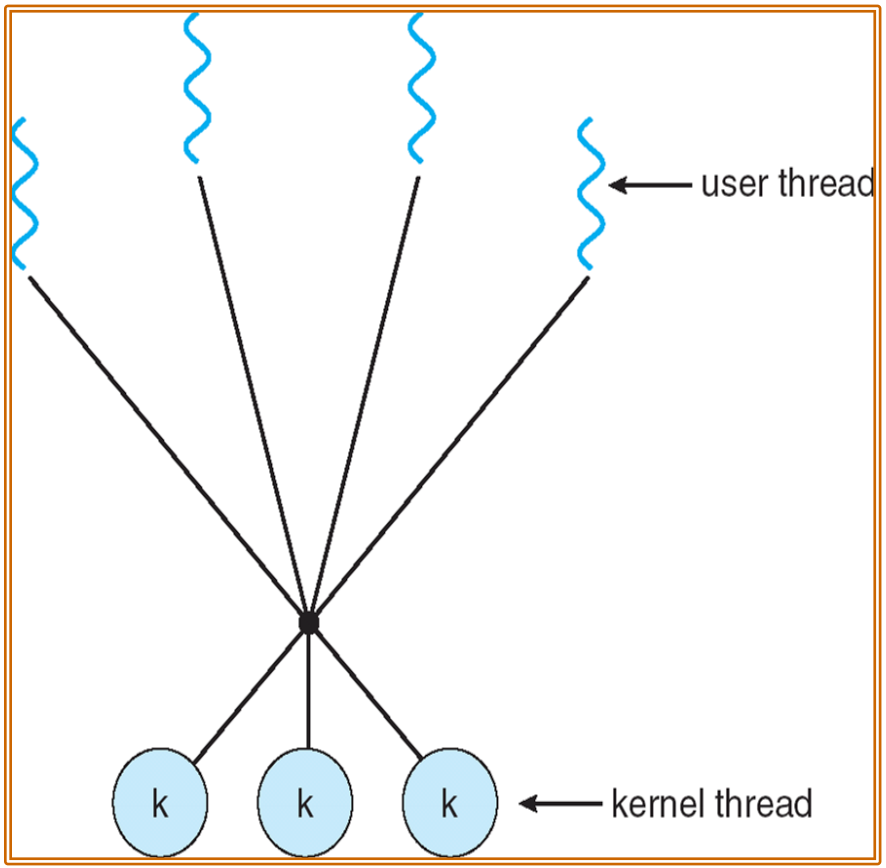
One-to-One

- Each user-level thread maps to kernel thread
- It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.
- Allows multiple threads to run in parallel on multiprocessors.
- Examples: Windows NT/XP/2000; Linux; Solaris 9 and later
- Drawback: Creating a user thread requires creating the corresponding kernel thread



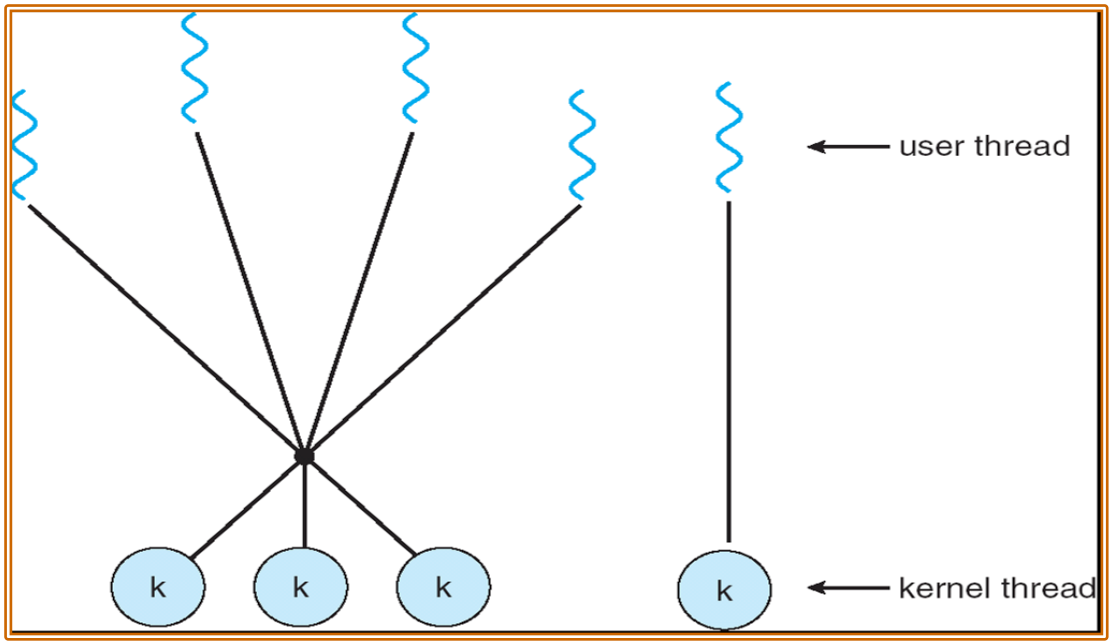
Many-to-Many Model

- Allows many user level threads to a smaller or equal number of kernel threads.
- Allows the operating system to create a sufficient number of kernel threads
- Users have no restrictions on the number of threads created.
- Blocking kernel system calls do not block the entire process.
- Processes can be split across multiple processors.
- Solaris prior to version 9,Windows NT/2000 with the *ThreadFiber* package



Two-level Model

- One popular variation on the many-to-many model , which allows either many-to-many or one-to-one operation
- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples: IRIX, HP-UX, Tru64 UNIX, Solaris 8 and earlier



Threads: benefits

User responsiveness: When one thread blocks, another may handle user I/O

- But: depends on threading implementation

Resource sharing: Memory is shared (i.e., address space shared)

- Open files, sockets shared

Economy: Creating a thread is fast

- Context switching among threads may be faster

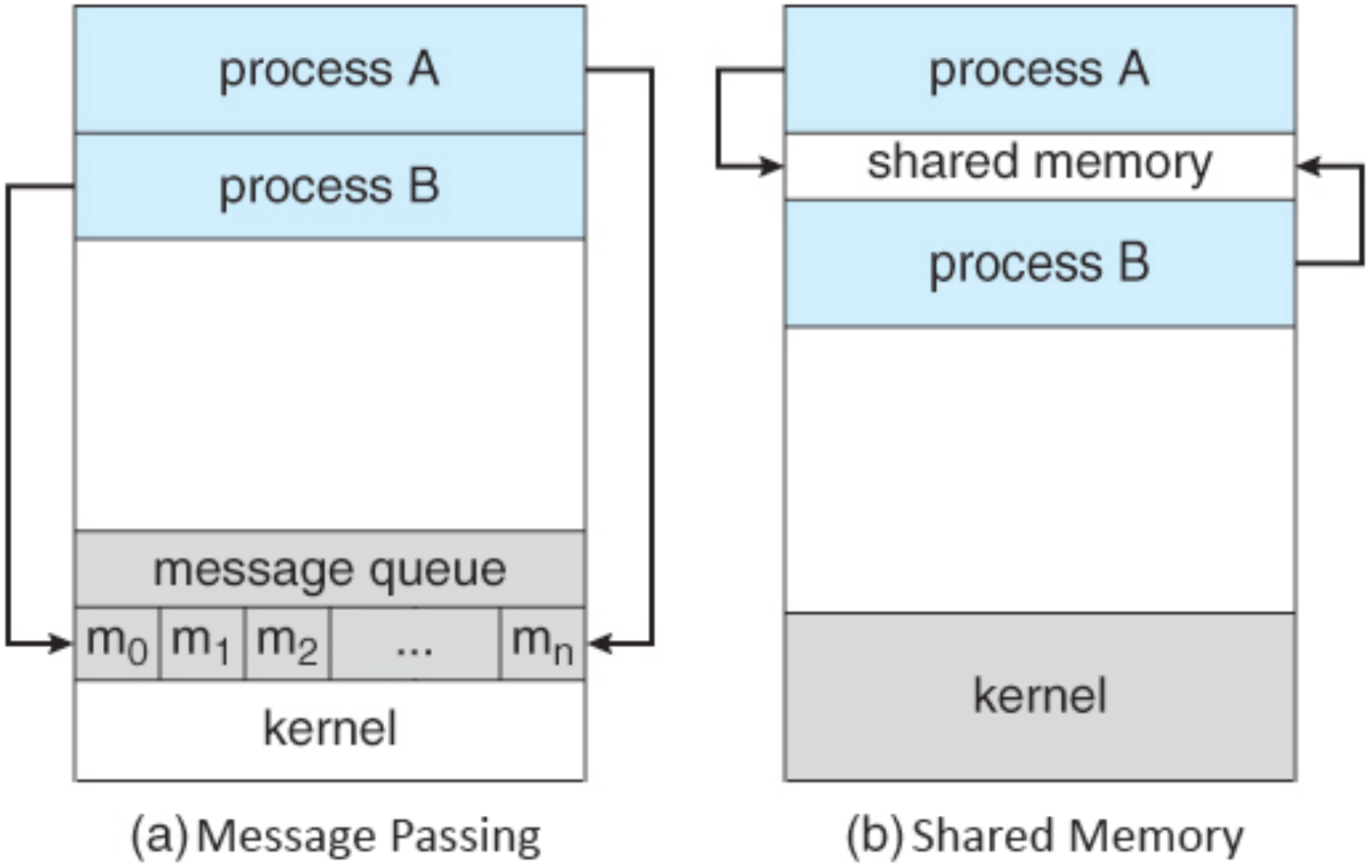
Speed: E.g., Solaris: thread creation about 30x faster than heavyweight process creation; context switch about 5x faster with thread

Utilizing hardware parallelism: Like heavy weight processes, can also make use of multiprocessor architectures

Inter-process Communication(IPC)

- **Inter-process communication** is used for exchanging data between multiple threads in one or more processes or program and synchronize the data.
- There are numerous reasons for providing an environment or situation which allows process co-operation:
- **Information sharing:** exchange the information
- **Computation speedup:** If you want a particular work to run fast, you must break it into sub-tasks where each of them will get executed in parallel with the other tasks.
- **Modularity:** Build the system in a modular way by dividing the system functions into split processes or threads.
- **Convenience:** Even a single user may work on many tasks at a time. For example, a user may be editing, formatting, printing, and compiling in parallel.

- There are two primary models of Interprocess Communication:



- In the **shared memory system**, the cooperating process which wants to initiate the communication establishes a region of shared memory in its address space. The other cooperative process which requires the shared data has to attach itself to the address space of the initiating process.
- The process A has some data, to share with process B. Process A has to take the initiative and establish a shared memory region in its own address space and store the data or information to be shared in it’s shared memory region.
- Process B requires the information stored in the shared segment of the A. So, process B has to attach itself to the shared address space of A. Now, B can read out the data from there.
- Process A and B can exchange information or data by reading and writing data in the shared segment of the process.
- The **message-passing system** shares the data by passing the messages. If a process needs to share the data, it writes the data in the message and passes it on to the kernel. The process that requires the shared data read it out from the kernel.

Race Conditions

- **Process Synchronization** is a mechanism that deals with the synchronization of processes. It controls the execution of processes running concurrently to ensure that consistent results are produced.
- The situation where two or more processes reading or writing some share data and the final results depends on who runs precisely when are called **Race conditions**.
- A **race condition** occurs when two processes can interact and the outcome depends on the order in which the processes execute.
- When a process wants to print a file, it enters the file name in a special **spooler directory**.
- Imagine that our spooler directory has a very large number of slots, numbered 0, 1, 2, ..., each one capable of holding a file name.
- At a certain instant, slots 0 to 3 are empty (the files have already been printed) and slots 4 to 6 are full (with the names of files queued for printing).

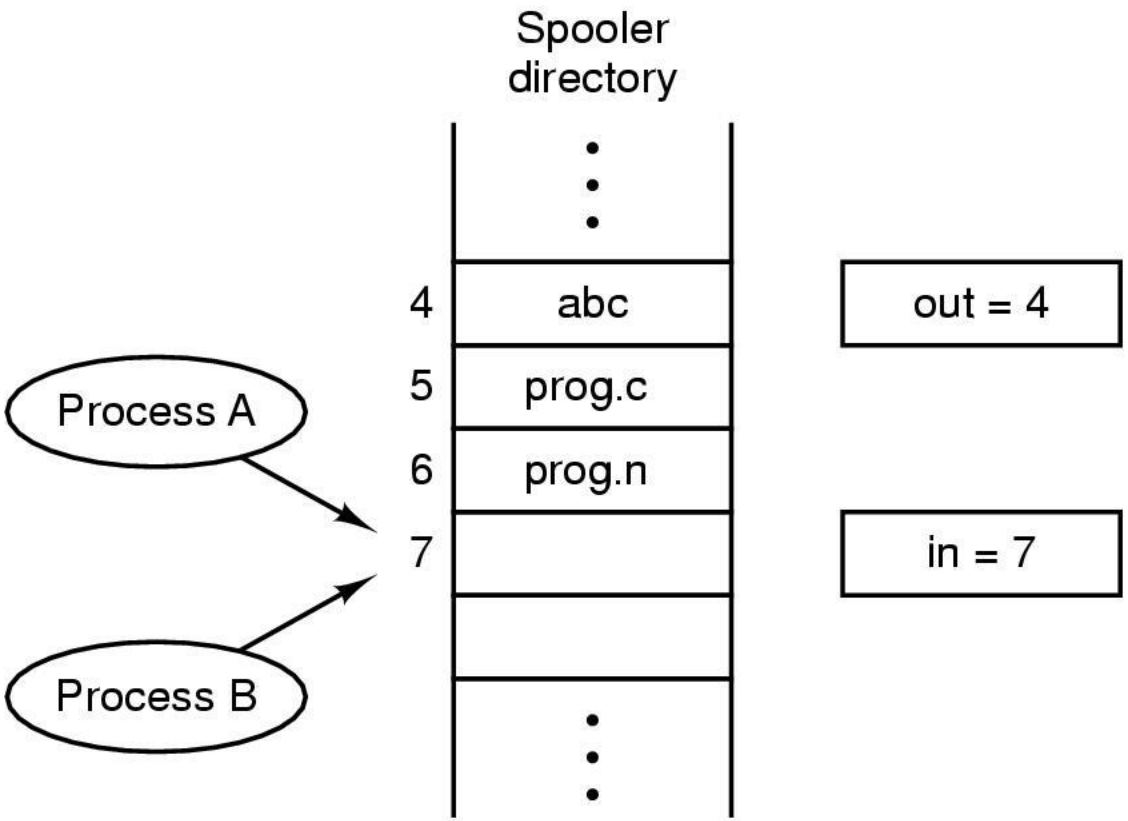


Figure .Two processes want to access shared memory at same time

- An operating system is providing printing services, “printer system process ” picks a print job from spooler. Each job gets a job number, and this is done by the spooler using two global variables “in” and “out”.
- Process A reads in and stores the value, 7, in a local variable called next free slot. Just then a clock interrupt occurs and the CPU decides that process A has run long enough, so it switches to process B.
- Process B also reads in, and also gets a 7, so it stores the name of its in slot 7 and update into be an 8. Then it goes off and does other things.
- Eventually, process A runs again, starting from the place it left off. It looks at next free slot, finds a 7 there, and writes its file name in slot 7, erasing the name that process B just put there. Then it computes next free slot + 1, which is 8, and sets in to 8.
- The spooler directory is now internally consistent, so the printer daemon will not notice anything wrong, but process B will never receive any output.

Critical Regions

- That part of the program where the shared memory is accessed is called the **critical region** or **critical section**.
- Atomic action is required in a critical section i.e. only one process can execute in its critical section at a time. All the other processes have to wait to execute in their critical sections.
- To avoid race condition we need mutual exclusion.
- **Mutual exclusion**: Mechanism which makes sure that two or more processes do not access a common resource at the same time.
- The difficulty in printer spooler occurs because process B started using one of the shared variables before process A was finished with it. If we could arrange matter such that no two processes were ever in there critical regions at the same time, we could avoid race conditions.

Solution to Critical Section Problem:

- No two processes may be simultaneously inside their critical regions.
- No assumptions may be made about speeds or the number of CPUs.
- No process running outside its critical region may block any process.
- No process should have to wait forever to enter its critical region.

Requirements of Synchronization mechanisms

Mutual Exclusion

Our solution must provide mutual exclusion. By Mutual Exclusion, we mean that if one process is executing inside critical section then the other process must not enter in the critical section.

Progress

Progress means that if one process doesn't need to execute into critical section then it should not stop other processes to get into the critical section.

Bounded Waiting

We should be able to predict the waiting time for every process to get into the critical section. The process must not be endlessly waiting for getting into the critical section.

Architectural Neutrality

Our mechanism must be architectural natural. It means that if our solution is working fine on one architecture then it should also run on the other ones as well.

Example :

- Process *A* enters its critical region at time *T*1. A little later, at time *T*2 process *B* attempts to enter its critical region but fails because another process is already in its critical region and we allow only one at a time.
- Process *B* is temporarily suspended until time *T*3 when *A* leaves its critical region, allowing *B* to enter immediately. Eventually *B* leaves (at *T*4) and we are back to the original situation with no processes in their critical regions.

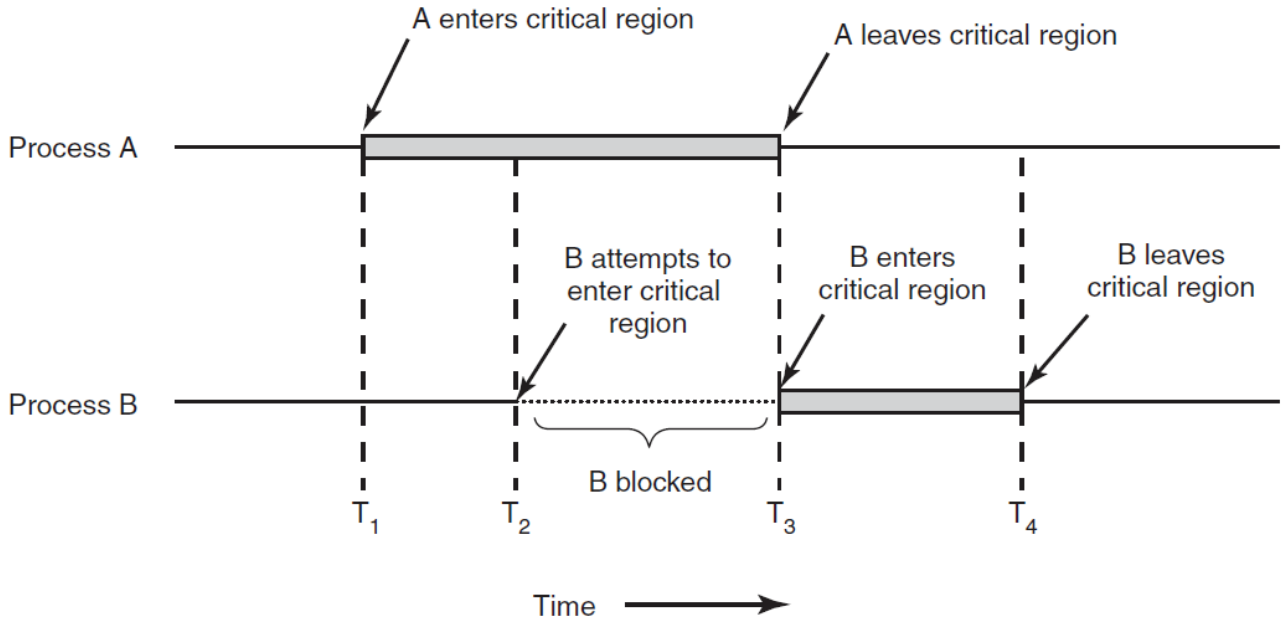


Figure . Mutual exclusion using critical regions.

Mutual Exclusion with Busy Waiting

Proposals for achieving mutual exclusion:

- Disabling interrupts
- Lock variables
- Strict alternation
- Peterson's solution
- The TSL instruction

Disabling interrupts

- Simplest solution, After entering critical region, disable all interrupts
- The CPU is only switched from process to process as a result of clock or other interrupts, after all, and with interrupts turned off the CPU will not be switched to another process.
- May be used inside operating system kernel when some system Structures are to be updated, but is not recommended for Implementation of mutual exclusion in user space.
- Advantage: process inside critical region may update shared resources Without any risk of races
- Disadvantage: if after leaving the region interrupts are not reenabled There will be a crash of the system. Moreover: useless in multiprocessor Architectures.

Lock variables

- Software solution at user level, multiprogram med solution.
- A software solution. Considering having a single, shared lock variable, initially 0.
- When a process wants to enter critical region, it first tests the lock.
- If the lock = 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.
- The pseudo code of the mechanism looks like following.

```
Entry Section →
While (lock! = 0);
Lock = 1;
//Critical Section
Exit Section →
Lock =0;
```

Strict Alternation/Turn Variable

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

(a)

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

(b)

Figure. A proposed solution to the critical region problem. (a) Process P0. (b) Process P1. In both cases, be sure to note the semicolons terminating the while statements.

- Turn Variable or Strict Alternation Approach is the software mechanism implemented at user mode.
- It keeps track of whose turn it is to enter the critical region and examine or update the shared memory.
- Initially, process P0 inspects turn, finds it to be 0, and enters its critical region. Process P1 also finds it to be 0 and therefore sits in a tight loop continually testing turn to see when it becomes 1.
- Continuously testing a variable until some value appears is called **busy waiting**. It should usually be avoided, since it wastes CPU time. A lock that uses busy waiting is called a **spin lock**.
- When process P0 leaves the critical region, it sets turn to 1, to allow process P1 to enter its critical region. This way no two process can enters critical region simultaneously.

Peterson's Solution

```
#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 */
}
```

Figure . Peterson's solution for achieving mutual exclusion.

- Peterson's solution is used for mutual exclusion and allowed to processes to share a single use resource without conflict. It uses only shared memory for communication.
- Peterson's solution originally worked only with two processes, but has been generalized for more than two.
- 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 makes a call to enter region, it will hang there until interested[0] goes to FALSE, an event that happens only when process 0 calls leave region to exit the critical region.
- Now consider the case that both process call **"Enter Region"** almost simultaneously. Then both will store the process number in 'turn'. Which ever store is done last is the one that counts 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 a critical region until process 0 exit its critical region.

TSL 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 not 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
```

Figure . Entering and leaving a critical region using the TSL instruction.

- Software solution has many problem like, high processing overhead, logical errors etc.
- Hardware solution, multiprocess solution(n>=2), some computer architectures offer an instruction **TEST AND SET LOCK** (**TSL**).
- 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.

- An alternative instruction to TSL is XCHG, which exchanges the contents of two locations atomically, for example, a register and a memory word.
- All Intel x86 CPUs use XCHG instruction for low-level synchronization.

```
enter_region:
        MOVE REGISTER,#1          | put a 1 in the register
        XCHG REGISTER,LOCK        | swap the contents of the register and lock variable
        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
```

Figure . Entering and leaving a critical region using the XCHG instruction.

Sleep and Wakeup

- When a process is not permitted to access its critical section, it uses a system call known as **sleep**, which causes that process to block. The process will not be scheduled to run again, until another process uses the **wakeup** system call.
- In most cases, **wakeup** is called by a process when it leaves its critical section if any other processes have blocked.
- There is a popular example called **producer consumer problem** which is the most popular problem simulating **sleep and wake** mechanism.
- The problem describe two processes, **User and Consumer**, who share a common **fixed size buffer.** Producer consumer problem also known as the **"Bounded Buffer Problem"** is a multi-process synchronization problem.
- Disadvantage: wakeup signal may be lost, which leads to deadlock.

Producer: The producer's job is to generate a bit of data, put it into the buffer and start again.

Consumer: The consumer is consuming the data (i.e remaining it from the buffer) one piece at a time.

- If the buffer is empty, then a consumer should not try to access the data item from it. Similarly, a producer should not produce any data item if the buffer is full.
- Counter: It counts the data items in the buffer. or to track whether the buffer is empty or full. Counter is shared between two processes and updated by both.

How it works?

• Counter value is checked by consumer before consuming it.

• If counter is 1 or greater than 1 then start executing the process and updates the counters.

• Similarly producer check the buffer for the value of Counter for adding data.

• If the counter is less than its maximum values, it means that there is some space in Buffer.

```
#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 */
     }
}
```

Figure. The producer-consumer problem with a fatal race condition.

Types of mutual exclusion

- Semaphores
- Monitors
- Locks (mutexes)
- Message passing
- Bounded Buffer(Producer consumer)

Semaphores

- In 1965, when E. W. Dijkstra (1965) suggested using an integer variable to count the number of wakeups saved for future use.
- Proposed variable or abstract datatype called **semaphore ,**A semaphore could have the value 0, indicating that no wakeups were saved, or some positive value if one or more wakeups were pending.
- Dijkstra proposed having two operations on semaphores, now usually called down and up (generalizations of sleep and wakeup). he used the names P and V instead of down/wait and up/signal.
- The down operation on a semaphore checks to see if the value is greater than 0. If so, it decrements the value (i.e., uses up one stored wakeup) and just continues.
- If the value is 0, the process is put to sleep without completing the down for the moment. Checking the value, changing it, and possibly going to sleep, are all done as a single, indivisible **atomic action**.

Two operations which can be used to access and change the value of semaphore variable.

```
P(semaphore s){
While(s==0);       /* wait until s=0*/
S=s-1;
}
V(semaphore s){
S=s+1;
}
```

```
#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 = 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 */
     }
}
```

Figure. The producer-consumer problem using semaphores.

- 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.
- Semaphores that are initialized to 1 and used by two or more processes to ensure that only one of them can enter its critical region at the same time are called **binary semaphores**, only takes the values 0 and 1 during execution of a program. Hence it is often called a **mutex**
- **Counting semaphore** can be used when we need to have more than one process in the critical section at the same time.
- Counting = $-\infty$ to $+\infty$, Binary=0,1
- It is a very popular tool used for **process synchronization**.

Monitors

- To make it easier to write correct programs, brinch hansen (1973) and hoare (1974) proposed a higher-level synchronization primitive called a **monitor**.
- A monitor is a collection of procedures, Variables, and data structures that are all Grouped together in a special kind of module Or package.
- Monitor is same as a class type: like object of class are created, the variable of monitor type are defined.

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .       .       .
        end;
end monitor;
```

Figure. A monitor.

```
monitor ProducerConsumer
        condition full, empty;
        integer count;

        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;

        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count - 1;
                if count = N - 1 then signal(full)
        end;

        count := 0;
end monitor;

procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;

procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

Figure . An outline of the producer-consumer problem with monitors

Mutexes

```
mutex_lock:
        TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
        CMP REGISTER,#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
ok:     RET                     | return to caller; critical region entered

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

Figure. Implementation of *mutex lock* and *mutex unlock*.

- A **mutex** is a shared 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.
- Tw o procedures are used with mutexes. When a thread (or process) needs access to a critical region, it calls mutex lock. If the mutex is currently unlocked (meaning that the critical region is available), the call succeeds and the calling thread is free to enter the critical region.
- On the other hand, 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.

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Figure . Some of the Pthreads calls relating to mutexes.

Thread call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them

Figure . Some of the Pthreads calls relating to condition variables.

Message Passing

- This method of interprocess communication uses two primitives, **send** and **receive,** which, like semaphores and unlike monitors, are system calls rather than language constructs.

```
send(destination, &message);
and
receive(source, &message);
```

two methods of message addressing:

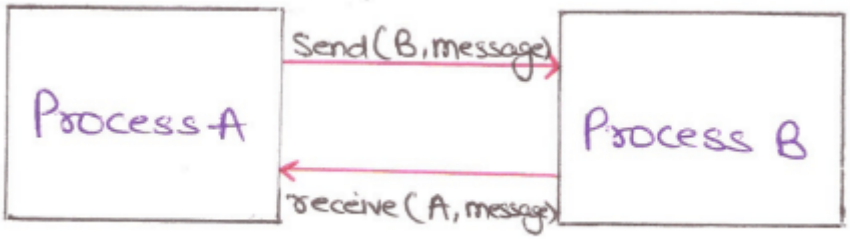
- **Direct addressing**, each process contains unique address. The following rendezvous mechanism may be used:
  - If **send** called before receive, the sending process suspended till the moment of the very message sending after receive call,
  - If **receive** called before send, the receiving process suspended till the moment of the very message sending after send call.

Example:

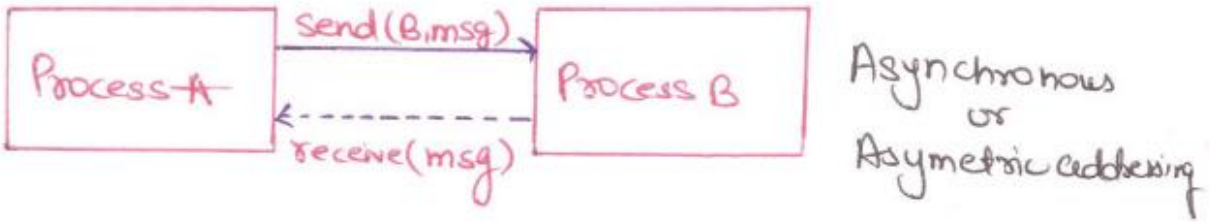
If process A sends a message to process B, then

send(B, message);
Receive(A, message);

- By message passing a link is established between A and B. Here the receiver knows the Identity of sender message destination. This type of arrangement in direct communication is known as **Symmetric Addressing.**



- Another type of addressing known as asymmetric addressing where receiver does not know the ID of the sending process in advance.



- **Indirect addressing**, via some mailbox playing the role of the Intermediate buffer. Send and receive has as an argument mailbox Address, not the address of any particular process.
- The sender and receiver processes should share a mailbox to communicate.

```
#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(item);                   /* do something with the item */
     }
}
```

Figure. The producer-consumer problem with *N* messages

Classical IPC Problems

The Dining philosopher problem

- In 1965, Dijkstra posed and then solved a **synchronization** problem he called the **dining philosophers problem**.
- Five philosophers are seated around a circular table. Each philosopher has a plate of spaghetti. The spaghetti is so slippery that a philosopher needs two forks to eat it. Between each pair of plates is one fork.
- A philosopher either eat or think
- When a philosopher gets sufficiently hungry, she tries to acquire her left and right forks, one at a time, in either order. If successful in acquiring two forks, she eats for a while, then puts down the forks, and continues to think.

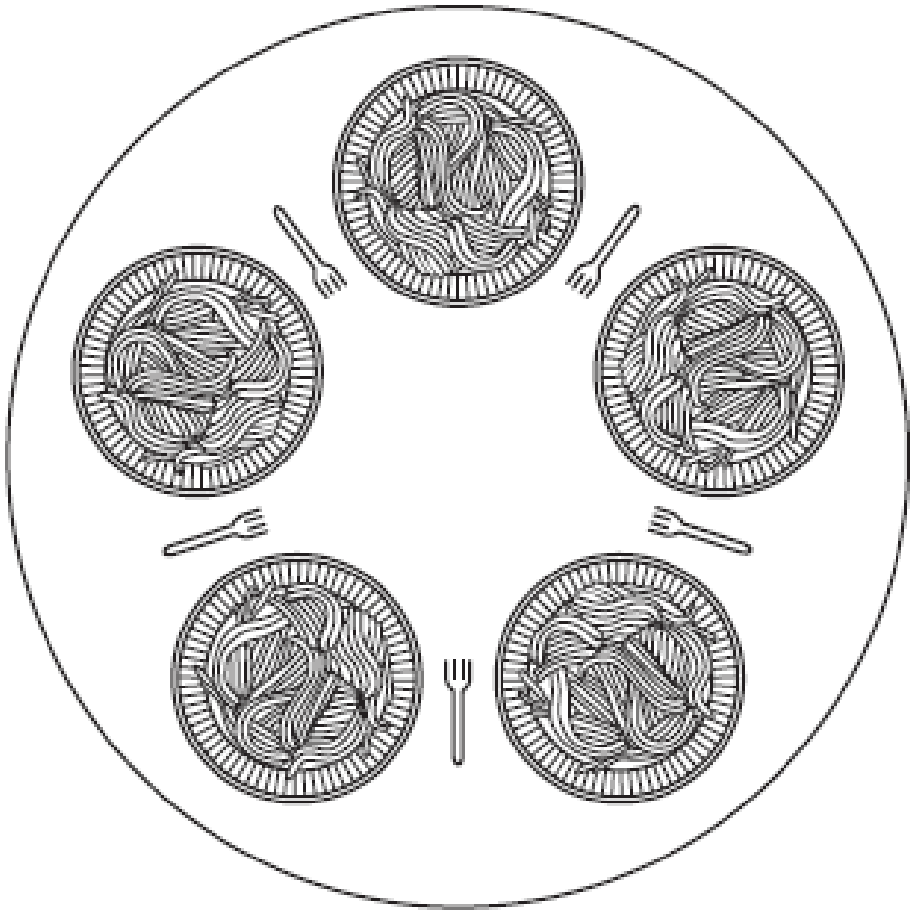


Figure . Lunch time in the Philosophy Department.

```
#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 for k */
take fork((i+1) % N);           /* take right for k; % is modulo operator */
eat( );                         /* yum-yum, spaghetti */
put fork(i);                    /* put left for k back on the table */
put fork((i+1) % N);            /* put right for k back on the table */
}
}
```

Figure. A nonsolution to the dining philosophers problem.

- The obvious solution is wrong. Suppose that all five philosophers take their left forks ,then waiting for right fork (None will be able to take their right forks), and there will be a **deadlock**.
- **Deadlock,** the ultimate form of starvation, occurs when two or more threads are waiting on a condition that cannot be satisfied. Deadlock most often occurs when two (or more) threads are each waiting for the other(s) to do something.
- **Starvation** occurs when one or more threads in your program are blocked from gaining access to a resource and, as a result, cannot make progress.
- A **solution** of the **Dining Philosophers Problem** is to use a semaphore to represent a forks. A forks can be picked up by executing a wait operation on the semaphore and released by executing a signal semaphore.

```
#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 for ks */
#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 for ks or block */
eat( );                                         /* yum-yum, spaghetti */
put forks(i);                                   /* put both for ks back on table */
}
}
```

```
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);                                /* tr y to acquire 2 for ks */
up(&mutex);                             /* exit critical region */
down(&s[i]);                            /* block if for ks were not acquired */
}
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]);
}
}
```

Figure . A solution to the dining philosophers problem.

The readers and writers problem

- Reader-writer problem in operating system which is used for process synchronization, which access the database.
- If a process is writing something on a file and another process also starts writing on the same file at the same time, then the system will go into the inconsistent state.
- Another problem is that if a process is reading the file and another process is writing on the same file at the same time, then this may lead to dirty-read.
- For example, an airline reservation system, with many competing processes wishing to read and write it. It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other processes may have access to the database, not even readers.

- Special type of mutual exclusion problem. A special solution can do better than a general solution.

Solution one:

- Readers have priority. Unless a writer is currently writing, readers can always read the data.

Solution two:

- Writers have priority. Guarantee no new readers are allowed when a writer wants to write.

Other possible solutions:

Weak reader's priority or weak writer's priority.

Weak reader's priority: An arriving reader still has priority over waiting writers. However, when a writer departs, both waiting readers and waiting writers have equal priority.

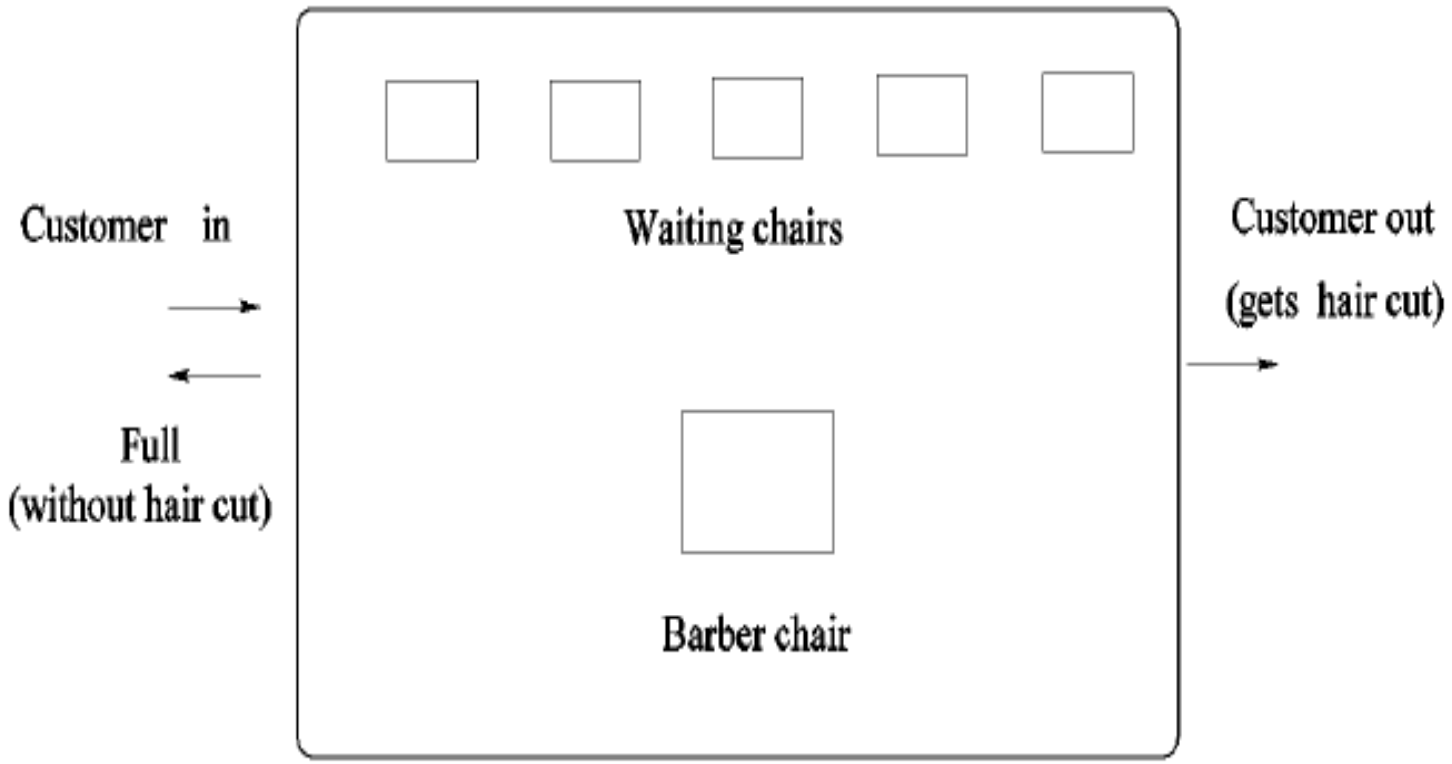
```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */
void 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 few er now */
if (rc == 0) up(&db);                   /* if this is the last reader ... */
up(&mutex);                             /* release exclusive access to rc */
use data read( );                       /* noncritical region */
}}
```

```
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 */
}
}
```

Figure. A solution to the readers and writers problem.

The Sleeping Barber Problem

Problem description:



- The barber shop has one barber, one barber chair, and n chairs for waiting customers, if any, to sit on.
- If there are no customers present, the barber sits down in the barber chair and falls asleep, as shown in the figure in the previous slide
- When a customer arrives, he has to wake up the sleeping barber. If additional customers arrive while the barber is cutting a customer's hair, they either sit down (if there are empty chairs) or leave the shop (if all chairs are full).

The solution uses three semaphores:

- **customers** , which counts waiting customers (excluding the customer in the barber chair, who is not waiting),
- **barbers** , no. of barbers (0 or 1) based on barber is idle/waiting for customers,
- **mutex** , which is used for mutual exclusion.

- We also need a variable, waiting , which also counts the waiting customers. It is essentially a copy of customers .

```
#define CHAIRS 5                    /* number of chairs for waiting customers */
typedef int semaphore;
semaphore customers = 0;            /* number of waiting customers */
semaphore barbers = 0;              /* number of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting not being haircut */
void Barber(void){
 while (TRUE){
 down(customers);                   /* go to sleep if number of customers is 0 */
 down(mutex);                       /* acquire access to 'waiting' */
 waiting = waiting – 1;             /* decrement count of waiting customers */
 up(barbers);                       /* one barber is now ready to cut hair */
 up(mutex);                         /* release 'waiting' */
 cut_hair();                        /* cut hair, non-CS */
 }}
```

```
void customer(void){
 down(mutex);                          /* enter CS */
 if (waiting < CHAIRS){
 waiting = waiting + 1;                /* increment count of waiting customers */
 up(customers);                        /* wake up barber if necessary */
 up(mutex);                            /* release access to 'waiting' */
 down(barbers);                        /* wait if no free barbers */
 get_haircut();                        /* non-CS */
 }else{
 up(mutex);                            /* shop is full, do not wait */
 }
}
```

Figure. A solution to the Sleeping Barber Problem