Mutual Exclusion

Intermediate Summary

• Disabling Interrupts	Not recommended for multi-user systems.
• Lock Variables	Problem remains the same.
• Strict Alternation	Violation of condition 3. Busy waiting.
• Peterson Algorithm	Busy waiting.
• TSL instruction	Solves the problem through atomic operation. Should be used without busy waiting.

In essence, what the last three solutions do is this: A process checks whether the entry to its critical region is allowed. If it is not, the process just sits in a tight loop waiting until it is.

Unexpected side effects, such as *priority inversion problem*.

Priority Inversion Problem

Consider a computer with two processes

Process ***H*** with high priority
Process ***L*** with low priority

The scheduling rules are such that ***H*** is run whenever it is in ready state.

At a certain moment, with ***L*** in its critical region, ***H*** becomes ready and is scheduled. ***H*** now begins busy waiting, but since ***L*** is never scheduled while ***H*** is running, ***L*** never has the chance to leave its critical region. ***H*** loops forever. This is sometimes referred to as the *priority inversion* problem.

Solution: *blocking* a process instead of wasting CPU time.

Sleep and wake up

`sleep()`

A system call that causes the caller to block, that is, the process voluntarily goes from the *running* state into the *waiting* state. The scheduler switches over to another process.

`wakeup(`*process*`)`

A system call that causes the process ***process*** to awake from its **`sleep()`** and to continue execution. If the process ***process*** is not a**`sleep`** at that moment, the wakeup signal is lost.

Note: these two calls are fictitious representatives of real system calls whose names and parameters depend on the particular operating system.

Producer – Consumer Problem

- Shared buffer with limited size

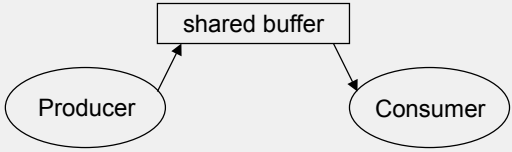
  The buffer allows for a maximum of *N* entries (it is bounded). The problem is also known as *bounded buffer* problem.

- Producer puts information into buffer

  When the buffer is full, the producer must wait until at least one item has been consumed.

- Consumer removes information from buffer

  When the buffer is empty the consumer must wait until at least one new item has been entered.



Producer – Consumer Implementation Example
This implementation suffers from race conditions

```
const int N = 100;
int count = 0;

void producer() {
  while (TRUE) {                          constantly producing
    int item = produce_item();            produce item
    if (count == N) sleep();              sleep when buffer is full
    insert_item(item);                    enter item to buffer
    count++;                              adjust item counter
    if (count == 1) wakeup(consumer);     when the buffer was empty beforehand
  }                                       (and thus now has 1 item), wakeup any
}                                         consumer(s) that may be waiting

                    (A)

void consumer() {
  while(TRUE) {                           constantly consuming
    if (count == 0) sleep();              sleep when buffer is empty
    item = remove_item();                 remove one item
    count--;                              adjust item counter
    if (count == N-1) wakeup(producer);   when the buffer was full beforehand
    consume_item(item);                   (and thus now has N-1 items), wakeup
  }                                       producer(s) that may be waiting.
}
```

Producer – Consumer Problem

A race condition may occur in this case:

The buffer is empty and the consumer has just read **`count`** to see if it is 0. At that instant (see (A) in listing) the scheduler decides to switch over to the producer.

The producer inserts an item in the buffer, increments **`count`** and notices that **`count`** is now 1. Reasoning that **`count`** was just 0 and thus the consumer must be sleeping, the producer calls **`wakeup()`** to wake the consumer up.

However, the consumer was not yet asleep, it was just taken away the CPU shortly before it could enter **`sleep()`**. The wakeup signal is lost.

When the consumer is rescheduled and resumes at (A), it will go to sleep. Sooner or later the producer has filled up the buffer and goes asleep as well.

Both processes will sleep forever.

Producer – Consumer Problem

Reasons for race condition

- The variable `count` is unconstrained
  Any process has access any time.
- Evaluating count and going asleep is a non-atomic operation
  The prerequisite(s) that lead to sleep() may have changed when sleep() is reached.

Workaround:

- Add a wakeup waiting bit
  When the bit is set, sleep() will reset that bit and the process stays awake.
- Each process must have a wakeup bit assigned
  Although this is possible, the principal problem is not solved.

What is needed is something that does *testing a variable and going to sleep* – dependent on that variable – in a single non-interruptible manner.

Semaphores

- Introduced by Dijkstra (1965)
- Counting the number of wakeups
  An integer variable counts the number of wakeups for future use.
- Two operations: `down` and `up`

down is a generalization of **sleep**. **up** is a generalization of **wakeup**. Both operations are carried out in a single, indivisible operation (usually in kernel). Once a semaphore operation is started, no other process can access the semaphore.

```
down(int* sem) {
 if (*sem < 1) sleep();
 *sem--;
}
```

principle of **down**-operation

```
up(int* sem) {
 *sem++;
 if (*sem == 1) wakeup a process
}
```

principle of **up**-operation

Semaphores

- `Up` and `down` are system calls

  in order to make sure that the operating system briefly disables all interrupts while carrying out the few machine instructions implementing **`up`** and **`down`**.

- Semaphores should be lock-protected

  This is recommended at least in multi-processor systems to prevent another CPU from simultaneously accessing a semaphore. TSL instruction helps out.

Producer – Consumer problem using semaphores (next page)

Definition of variables:

```
const int N = 10;
```

a semaphore is an integer — `typedef int semaphore;`

counting empty slots — `semaphore empty = N;`

counting full slots — `semaphore full = 0;`

mutual exclusion on buffer access — `semaphore mutex = 1;`

Producer – Consumer Implementation Example

This implementation does not suffer from race conditions

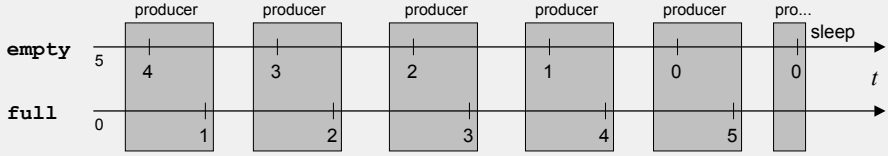
```
void producer() {
  while (TRUE) {
    int item = produce_item();
    down(&empty);          // possibly sleep, decrement empty counter
    down(&mutex);          // possibly sleep, claim mutex (set it to 0) thereafter
    insert_item(item);
    up(&mutex);            // release mutex, wake up other process
    up(&full);             // increment full counter, possibly wake up other ...
  }
}

void consumer() {
  while(TRUE) {
    down(&full);           // possibly sleep, decrement full counter
    down(&mutex);          // possibly sleep, claim mutex (set it to 0) thereafter
    item = remove_item();
    up(&mutex);            // release mutex, wake up other process
    up(&empty);            // increment empty counter, possibly wake up other ...
    consume_item(item);
  }
}
```

Semaphores

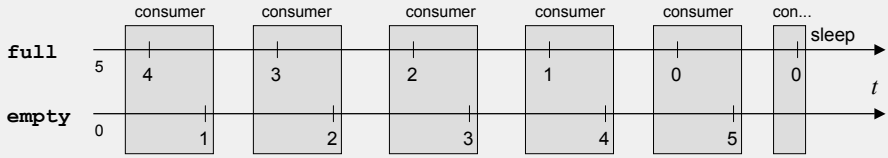
Assume **N** = 5. Initial condition: **empty** = 5, **full** = 0.

Mutual Exclusion



Scenario: producer is working, no consumer present

Initial condition: **empty** = 0, **full** = 5.

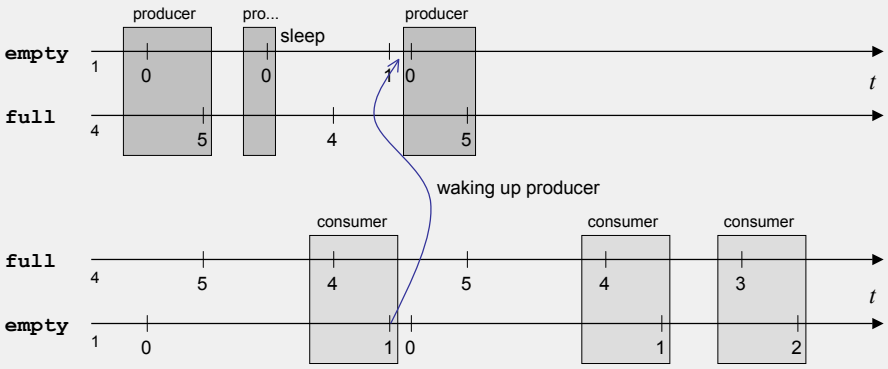


Scenario: consumer is working, no producer present

Semaphores

Assume **N** = 5. Initial condition: **empty** = 1, **full** = 4.

Mutual Exclusion

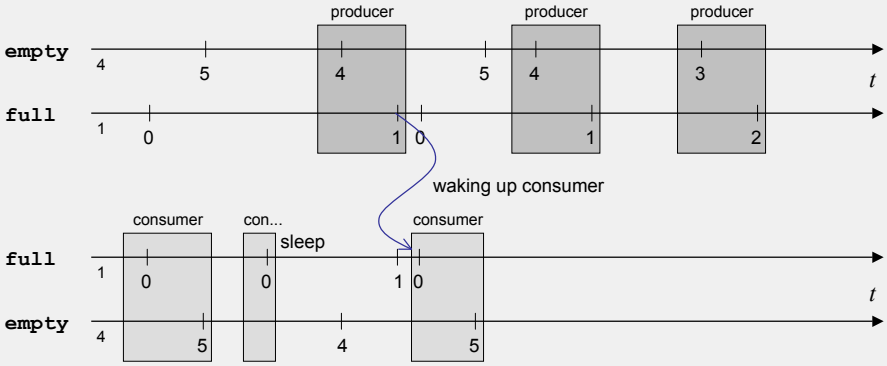


Scenario: Consumer waking up producer

Semaphores

Assume **N** = 5. Initial condition: **empty** = 4, **full** = 1.

Mutual Exclusion

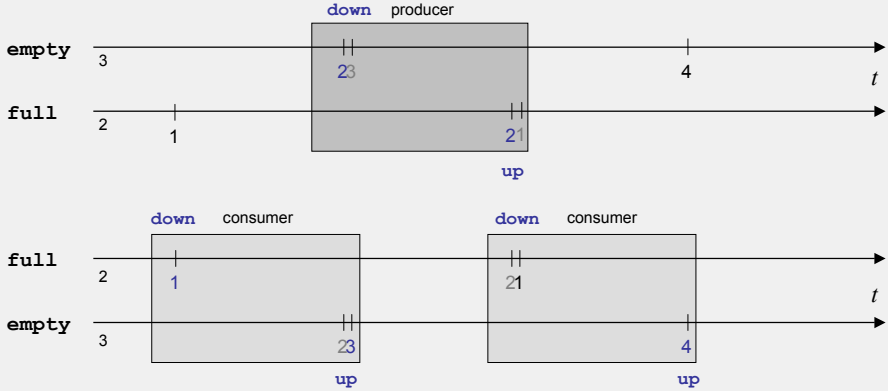


Scenario: Producer waking up consumer

Semaphores

Assume **N** = 5. Initial condition: **empty** = 3, **full** = 2.

Mutual Exclusion



If processes overlap, then temporary it may be that **empty + full ≠ N**

Note that consumer and producer may almost concurrently change the same semaphore legally.

Mutex

- Simplified semaphore
  when counting is not needed.
- Two states
  Locked or *unlocked*. Used for managing mutual exclusion (hence the name).

```
mutex_lock:    TSL R, mutex        get and set mutex
               CMP R, #0           was it unlocked?
               JZ ok               if yes: jump to ok
               CALL thread_yield   if no: sleep
               JMP mutex_lock      try again acquiring mutex
ok:            RET

mutex_unlock:  MOV mutex, #0       unlock mutex
               RET
```

Pseudo assembler listing implementing mutex_lock() and mutex_unlock().

Monitors

- High level synchronization primitive
  at programming language level. Direct support by some programming languages.
- A collection of procedures, variables and data structures grouped together in a module
  - A monitor has multiple entry points
  - Only one process can be in the monitor at a time
  - Enforces mutual exclusion – less chances for programming errors
- Monitor implementation
  - Compiler handles implementation
  - Library functions using semaphores

Monitors


```
monitor example;
      integer i;
      condition c;

      procedure producer()
       …
       …
      end;

      procedure consumer()
       …
       …
      end;

end monitor;
```

Variables not accessible from outside the monitor's own methods (capsulation).

Functions (methods) publicly accessible to all processes, however only one process at a time may call a monitor function.

If the buffer is full, the producer must wait.
If the buffer is empty the consumer must wait.

A monitor in Pidgin Pascal, from [Ta01 p.115]

Monitors

- How can a process wait inside a monitor?

  Cannot put to sleep because no other process can enter the monitor meanwhile.

- Use a condition variable!

  A condition variable supports two operations.

  - **wait()**: suspend this process until it is signaled. The suspended process is not considered *inside* the monitor any more. Another process is allowed to enter the monitor.
  - **signal()**: wake up one process waiting on the condition variable. No effect if nobody is waiting. The signaling process automatically leaves the monitor (*Hoare* monitor).
  - Condition variables usable only inside a monitor.

Monitors

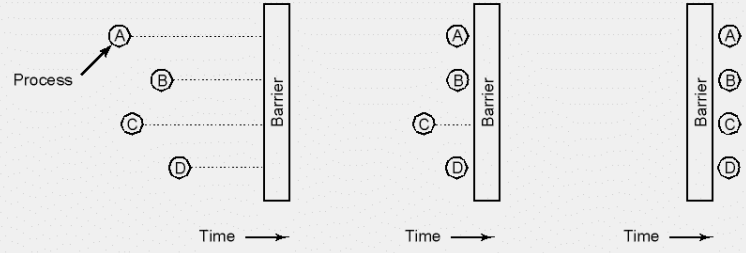

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        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;
```

Producer-Consumer problem with monitors, from [Ta01 p.117]

Barriers

- Group synchronization
  Intended for groups of processes rather than for two processes.
- Processes wait at a *barrier* for the others
  according to the *all-or-none* principle
- After all have arrived, all can proceed

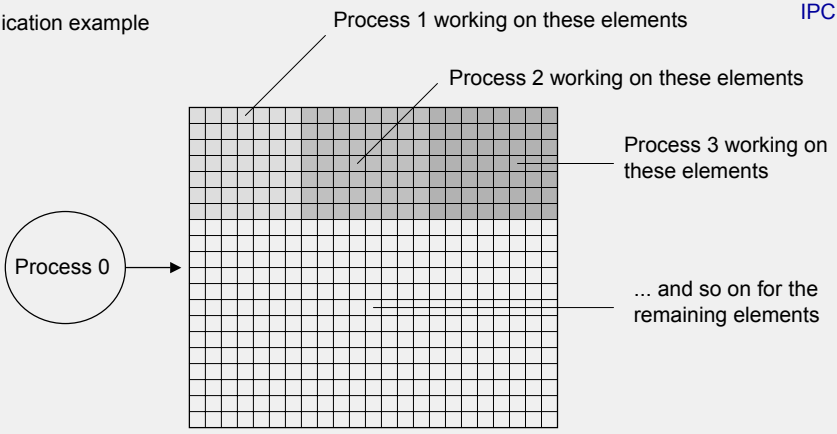


Processes approaching barrier | Waiting for C to arrive | All processes continuing

Figure from [Ta01 p.124]

Barriers

Application example



An array (e.g. an image) is updated frequently by some process 0 (producer). Many processes are working in parallel on certain array elements (consumers). All consumers must wait until the array has been updated and can then start working again on the updated input.

IPC

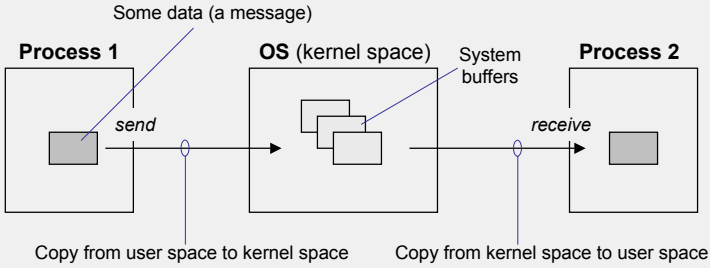
Intermediate Summary (II)

- Semaphores
  Counting variable, used in non-interruptible manner. **`Down`** may put the caller to sleep, **`up`** may wake up another process.
- Mutexes
  Simplified semaphore with two states. Used for mutual exclusion.
- Monitors
  High level construct for achieving mutual exclusion at programming language level.
- Barriers
  Used for synchronizing a group of processes.

These mechanisms all serve for process synchronization.
For data exchange among processes something else is needed: Messages.

Messages

- Kernel supported mechanism for data exchange
  Eliminates the need for ‚self-made' (user-programmed) communication via shared resources such as shared files or shared memory.
- Two basic operations:
  provided by the kernel (system calls)
  - **send()**: send data
  - **receive()**: receive data



Direct Communication

- Both processes must exist
  As the name *direct* implies, you cannot send a message to a future process.
- Processes must name each other explicitly
  - send(P, message): send data to process P
  - receive(Q, message): receive data from process Q

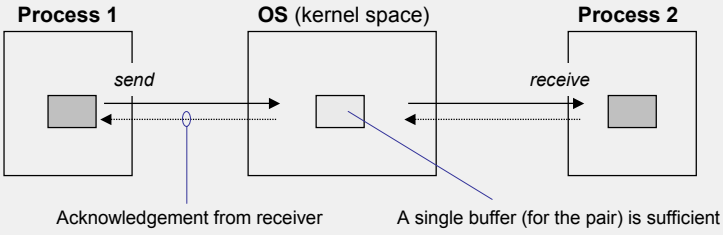
  Symmetry in addressing. Both processes need to know each other by some identifier. This is no problem if both were **fork()**ed off the same parent beforehand, but is a problem when they are ‚strangers' to each other.
- Communication link properties
  - One process pair has exactly one link
  - The link may be unidirectional or bidirectional

Indirect Communication

- Messages are send / received from mailboxes
  The mailbox must exist, not necessarily the receiving process yet.
  - Each mailbox has a unique identifier
  - Processes communicate when they access the same mailbox
- Primitives
  - send(A, message): send message to mailbox A
  - receive(A, message): receive message from mailbox A
- Communication link properties
  - Link is established when processes share a mailbox
  - A link may be associated with many processes (broadcast)
  - Unidirectional or bidirectional communication

Synchronous Communication

- Also called *blocking* send / receive
- Sender waits for receiver to receive the data
  The send() system call blocks until receiver has received the message.

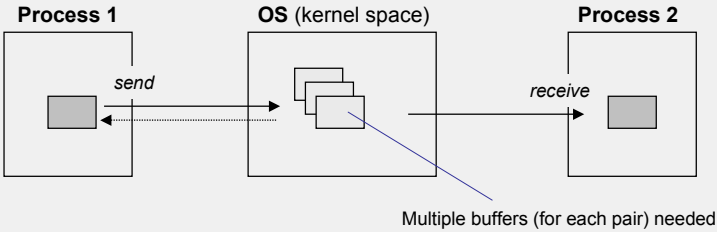


- Receiver waits for sender to send data
  The receive() system call blocks until a message is arriving.

Asynchronous Communication

- Also called *non-blocking* send / receive
- Sender drops message and passes on

  The send() system call returns to the caller when the kernel has the message.



- Receiver peeks for messages

  The receive() system does not block, but rather returns some error code telling whether there is a message or not. Receiver must do *polling* to check for messages.

Messages

- Send by copy

  The message is copied to kernel buffer at send time. At receive time the message is copied to the receiver. Copying takes time.
- Send by reference

  A reference (a memory address or a handle) is copied to the receiver which uses the reference to access the data. The data usually resides in a kernel buffer (is copied there beforehand). Fast read access.
- Fixed sized messages

  The kernel buffers are of fixed size – as are the messages. Straightforward system level implementation. Big messages must be constructed from many small messages which makes user level programming somewhat more difficult.
- Variable sized messages

  Sender and receiver must communicate about the message size. Best use of kernel buffer space, however, buffers must not grow indefinitely.

UNIX IPC Mechanisms

- **Pipes**

  Simple(st) communcation link between two processes. Applies *first-in first-out* principle. Works like an invisible file, but is no file. Operations: `read()`, `write()`.

- **FIFOs**

  Also called *named* pipe. Works like a file. May exist in modern Unices just in the kernel (and not in the file system). There can be more than one writer or reader on a FIFO. Operations: `open()`, `close()`, `read()`, `write()`.

- **Messages**

  Allow for message transfer. Messages can have types. A process may read all messages or only those of a particular type. Message communication works according to the *first-in first-out* principle.
  Operations: `msgget()`, `msgsnd()`, `msgrcv()`, `msgctl()`.

UNIX IPC Mechanisms

- **Shared memory**

  A selectable part of the address space of process P_1 is mapped into the address space of another process P_2 (or others). The processes have simultaneous access.
  Operations: `shmget()`, `shmat()`, `shmdt()`, `smhctl()`.

- **Semaphores**

  Creation and manipulation of sets of semaphores.
  Operations: `semget()`, `semop()`, `semctl()`.

For an introduction into the UNIX IPC mechanisms (with examples) see

Stefan Freinatis: *Interprozeßkommunikation unter Unix - eine Einführung*, Technischer Bericht, Fachgebiet Datenverarbeitung, Universität Duisburg, 1994.
http://www.fb9dv.uni-duisburg.de/vs/members/fr/ipc.pdf

Simple **pipe** example. Parent is writing, child is reading.

```
const int FIXSIZE=80
void main() {
   int fd[2];                          // file descriptors for pipe
   pipe(fd);                           // create pipe
   int result = fork();                // duplicate process
   if (result == 0) {                  // start child's code
     printf("This is the child, my pid is: %d\n", getpid());
     close(fd[1]);                     // we do not need writing
     char buf[256];                    // a buffer
     read(fd[0], buf, FIXSIZE)         // wait for message from parent
     printf("Child: received message was: %s\n", buf);
     exit(0);                          // good bye
   }                                   // end child, start parent
   close(fd[0]);                       // we do not need reading
   printf("This is the parent, my pid is: %d\n", getpid());
   write(fd[1], "Hallo!", FIXSIZE); // write message to child
}
```

Classical IPC Problems

The *dining philosophers*

An artificial synchronization problem posed and solved by Edsger Dijkstra 1965.

- Five philosophers sitting at a table
  The problem can be generalized to more than five philosophers, of course.
- Each either eats or thinks
- Five forks available
- Eating needs 2 forks
  Slippery spaghetti, one needs two forks!
- Pick one fork at a time
  Either first the right fork and then the left one, or vice versa.

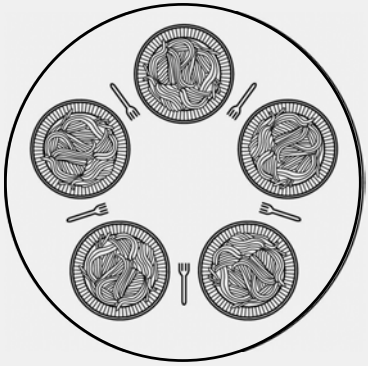


Figure from [Ta01 p.125]

Dining philosophers

The life of these philosophers consists of alternate periods of eating and thinking. When a philosopher becomes hungry, she tries to acquire her left and right fork, 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.

Text from [Ta01 p.125]

Can you write a program that

- makes the philosophers eating and thinking (thus creation of 5 threads or processes, one for each philosopher),
- allows maximum utilization (parallelism), that is, two philosophers may eat at a time (no simple solution with just one philosopher eating at a time),
- is not centrally controlled by somebody instructing the philosophers,
- and that never gets stuck?

Dining philosophers


```
const int N=5;
void philosopher(int i) {          // N philosophers in parallel
   while(TRUE){                    // for the whole life
     think();
     take_fork(i);                 // take left fork
     take_fork((i+1)%N);           // take right fork
     eat();
     put_fork(i);                  // put left fork
     put_fork((i+1)%N);            // put right fork
   }
}
```

A *non*solution to the dining philosophers problem

If all philosophers take their left fork simultaneously, none will be able to take the right fork. All philosophers get stuck. Deadlock situation.

Classical IPC Problems

The *Readers and Writers* Problem

An artificial shared database access problem by Courtois et. al, 1971

- Database system
  such as an airline reservation system.
- Many competing processes wish to read and write
  Many reading processes is not the problem, but if one process wants to write, no other process may have access – not even readers.

How to program the readers and writers?

- Writer waits until all readers are gone
  Not good. Usually there are always readers present. Indefinite wait.
- Writer blocks new readers
  A solution. Writer waits until old readers are gone and meanwhile blocks new readers

Classical IPC Problems

The *sleeping barber* problem

An artificial queuing situation problem

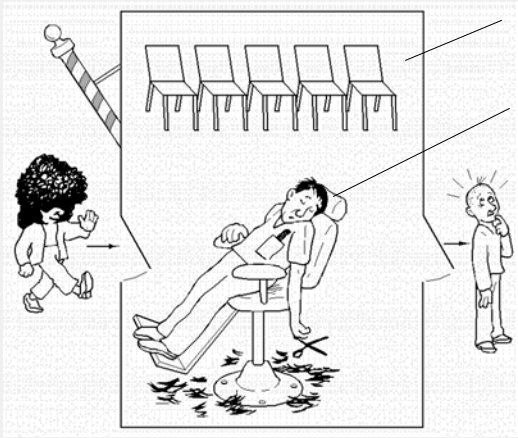


Figure from [Ta01 p.130]

Sleeping Barber

The barber shop has one barber, one barber chair, and *n* chairs for customers, if any, to sit on. If there are no customers present, the barber sits down in the barber chair and falls asleep. 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).

Text from [Ta01 p.129]

How to program the barber and the customers without getting into race conditions?

```
const int CHAIRS=5;          // number of chairs
typdef int semaphore;
semaphore customers = 0;     // number of customers waiting
semaphore barbers = 0;       // number of barbers waiting
semaphore mutex = 1;         // for mutual exclusion
int waiting = 0;
```



```
void barber() {                  // barber process
   while(TRUE){                  // for the whole life
     down(&customers);           // sleep if no customers
     down(&mutex);               // acquire access to 'waiting'
     waiting--;
     up(&barbers);               // one barber ready to cut
     up(&mutex);                 // release 'waiting'
     cut_hair();                 // cut hair (non critical)
   }
}
```

A solution to the sleeping barber problem [Ta01 p.131]

```
void customer() {                // customer process
  down(&mutex);                  // enter critical region
  if (waiting < CHAIRS){         // when seats available
    waiting++;                   // one more waiting
    up(&customers);              // tell barber if first customer
    up(&mutex);                  // release 'waiting'
    down(&barbers);              // sleep if no barber available
    get_haircut();               // get serviced
  } else up(&mutex);             // shop is full, leave
}
```