Real-Time Operating Systems (0_KRI)
Concurrent Programming

Ivan Cibrario Bertolotti

IEIIT-CNR / Politecnico di Torino

Academic Year 2006-2007

Outline

Definition of Process

The most central concept in any operating system is the **process**: an abstraction of a running program.

- Most operating systems can do several things at the same time, for example run a user program while reading from a disk.
- **Multiprogrammed** systems also switch the CPU from one program to another, thus giving the users the illusion of their parallel execution.
- Keeping track of multiple, parallel activities is hard to do. Hence, operating system designers have introduced a conceptual model, based on **sequential processes**, to better describe and deal with parallelism.

The Process Model

- All the runnable software on the computer, often including the operating system itself, is organized into a number of sequential **processes**.
- A process is an **executing program**, and includes all the state information that represents the execution.
- Informally, each process has “its own” virtual CPU, even if in reality the real CPU switches back and forth from process to process.

What Is There in a Process?

- The program being executed, or **text**
- The **CPU state**: program counter, registers, . . .
- The memory **address space** and its contents: variable values, . . .
- The state of other **resources**: files, I/O devices, . . .

Observation

The process state exactly represents the information that the operating system must be aware of, and must save/restore while switching from one process to another.

Processes Versus Programs

The distinction between a **process** and a **program** is subtle, but crucial.

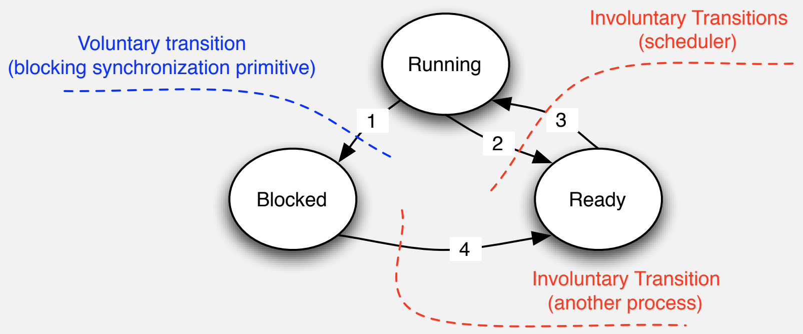
- A **program** is a **static** entity, and represents an algorithm expressed in some suitable programming language.
- A **process** is an **activity**: the activity consisting of executing the program.
- A process cannot be fully characterized by its corresponding program, because it also has **input**, **output**, and a **state**.
- Several processes can share the same program but nevertheless be distinct from each other, because their **states** are different.
- There is no correspondence between **processes** and **processors**: for example, the same processor may be shared among multiple processes through multiprogramming.

Process State Diagram

- A process may be ready for execution, and have a CPU allocated to it, so it is **running**.
- It is also possible for a process that is **ready** for execution and able to run to be stopped instead, because all the CPUs are currently allocated to other processes.
- When a process interacts with another process, or with the external environment through a device, it may need to **block**.
- For example, a process logically cannot continue if an input it requires is not yet available.

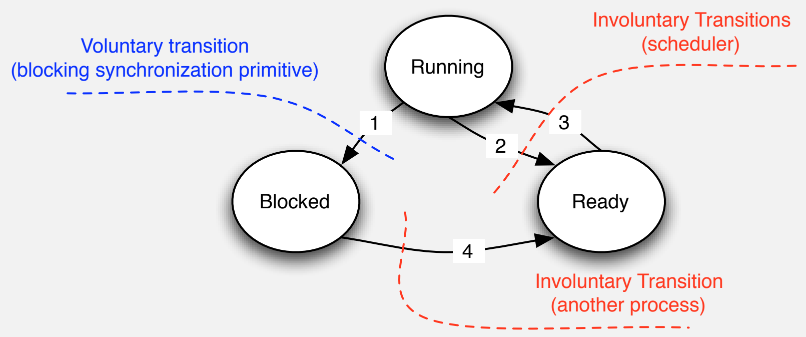
All these situations, and the corresponding transitions, can be modeled by means of a **process state diagram** (PSD).

PSD States



- The **Running** processes are actually using the CPU at that instant.
- The **Ready** processes are runnable, but cannot run because they lack the CPU.
- The **Blocked** processes are unable to run at that instant.

PSD Transitions



- Transition **1** occurs when a process discovers that it cannot continue, for example when it executes a blocking $P(s)$ on a semaphore s.
- Transition **4** occurs when the event a process was waiting for happens. For example, when another process executes a $V(s)$ on a semaphore s, one of the processes waiting on it, if any, is awakened.
- Transitions **2** and **3** are caused by the operating system scheduler.

Threads

- Traditional processes have their own address space and a **single** thread of control.
- This thread of control is characterized by:
  - a **program counter**, to keep track of which instruction to execute next,
  - a set of **registers**, containing its current working data,
  - a **stack**, which represents its execution history by holding a frame for each pending procedure call, along with its local variables.
- What threads add to the process model is to allow multiple executions, i.e. threads of control, to take place in the same process environment. The executions are largely independent of one another.

Threads are useful to represent in a natural way multiple activities going on at once in the same application.

Thread Advantages

1. All the threads of the same process implicitly share the same address space, but they are independent for what concerns execution. This is ability is essential for certain applications.
2. Threads are easier to create, destroy and manage than processes, because their context is more lightweight.
3. It is more efficient to switch from thread to thread (within the same process), that from process to process.

Many real-time operating systems provide a **single process** with **multiple threads**, due to performance considerations and/or hardware constraints.

Shared Variable-Based Communication

The major difficulties associated with concurrent programming arise from process **interaction**, that is nonetheless necessary.

- The correct behavior of a concurrent application is critically dependent on **synchronization** and **communication** among its processes:
  - **Synchronization** is the satisfaction of constraints on the interleaving of the actions performed by different processes.
  - **Communication** is the passing of information from one process to another.
- Interprocess communication may be based upon either **shared variables** or **message passing**.
- Shared variables appear to be a straight forward way of passing information between processes (or threads), but their use is prone to **race conditions**.

Race Conditions, an Example

Race conditions occur even in very simple cases

Consider two processes updating a shared variable `i` with the C assignment `i=i+1`.

- The assignment above is often considered to be an indivisible, **atomic** operation, but this is **not true**.
- On most hardware, it will be implemented as a **sequence** of three distinct instructions, for example:
  1. load the value of `i` from memory into a register,
  2. increment the value in the register by 1
  3. store the contents of the register back to `i`.
- As the three operations are not indivisible, the two process could follow an interleaving that does not produce the expected results.

An Unfortunate Interleaving

If the initial value of `i` is 8, and two processes *A* and *B* increment it by 1, the final result **should** be 10, but. . .

- Process *A* loads `i` from memory, gets the value 8, puts it into a register and increments it. The value of *A*'s register is 9.
- Before *A* stores the result into memory, process *B* loads `i` from memory, still gets the value 8, puts it into a register and increments it. The value of *B*'s register is 9.
- Both *A* and *B* store the result into memory. Regardless of the order in which the stores occur, the final value of `i` is **9 instead of 10**.

Critical Regions

Definitions

- A sequence of statements where the shared memory is accessed, and that must appear to be executed indivisibly to avoid race conditions is called a **critical region**, or **critical section**.
- The synchronization required to protect a critical region is known as **mutual exclusion**.

- The mutual exclusion problem was first described by Dijkstra (1965), and is of great practical, as well as theoretical, interest.
- In practice, atomicity is assumed to be present at the memory load and store level, at least for some simple object types, for example a memory word.
- Instead, it is not possible to assume that atomicity implicitly holds when updating a structured object in memory.

Other Kinds of Synchronization

Mutual exclusion is not the only kind of synchronization of importance.

- **Condition** synchronization is another significant requirement. It is needed when a process can perform an operation in a meaningful way only after another process has itself taken another action, or is in some specific state.
- For example, when a set of producer and consumer process share a circular memory buffer, a consumer must not attempt to take data from the buffer if the buffer is empty.
- Symmetrically, a producer must not attempt to store data into the buffer if the buffer is full.
- Moreover, mutual exclusion is **still needed** so that, for example, multiple producers operating concurrently do not corrupt the “next free slot” pointer of the buffer.

Solving the Race Condition Problem

Any solution to the race condition problem must satisfy four conditions to be considered a good solution:

1. No two processes may be **simultaneously inside** their critical region.
2. No assumption may be made about the relative **speed** of the processes involved, or the **number of CPUs**.
3. No process, while running outside its critical region, may prevent another process from entering its own critical region.
4. No process should have to wait forever to enter its critical region.

Busy Waiting

Definition

Several methods for achieving mutual exclusion are based on **busy waiting**: processes that are waiting to enter their critical region actively consume CPU cycles, for example by continuously testing a variable in a loop, until it assumes a particular value.

Among these methods, we recall:

- Use of **spin locks** with hardware assistance, for example by means of the “test & set” instruction.
- Methods based on some form of **alternation**.
- The **Dekker and Peterson** algorithm for mutual exclusion.

Semaphores

Semaphores, suggested by Dijkstra in 1965, are a simple mechanism for programming both **mutual exclusion** and **condition synchronization**. They have the following main benefits:

1. They simplify the protocols for synchronization, because a single, **abstract** object is used to perform it.
2. They do not require **busy waiting**, because the wait is implemented by putting a process in the **blocked** state, so that no CPU time is wasted.

Semaphore Primitives

Definition

A **semaphore** is an object which has a non-negative integer **value** and may have zero or more processes **waiting** for, and **queued** on, it.

- The **value** of a semaphore is set during its initialization, and is **no longer** accessible thereafter.
- Apart from initialization, semaphores can only be acted upon by two procedures, that perform their job as a single, indivisible **atomic** action.
- The atomicity of the procedures is **essential**, and must be achieved through another, lower-level, mechanism. For example, temporarily disabling interrupts may be adequate for a single processor system.

Semantics of P() and V()

The $P(s)$ operation (also called **wait** or **down**) atomically checks whether the value of the semaphore s is greater than zero.
If so, it **decrements** its value by one and just continues, else it **blocks** the calling process.

The $V(s)$ operation (also called **signal**, **up**, or **post**) atomically checks whether there is at least one process that was blocked on the semaphore s.
In this case, **one** of them is chosen by the system (for example, in FIFO order) and returned to the **ready** state. Otherwise, the value of the semaphore s is **incremented** by one.
No process ever gets blocked by executing a $V()$.

Solving the Producer/Consumer Problem

```
#define N 100
sem_t mutex = (sem_t)1;
sem_t empty = (sem_t)N;
sem_t full = (sem_t)0;
int buf[N];

void produce(int v) {
  P(empty);
  P(mutex);
  buf[in] = v;
  in = (in + 1) % N;
  V(mutex);
  V(full);
}
```

```
int consume(void) {
  int v;

  P(full);
  P(mutex);
  v = buf[out];
  out = (out + 1) % N;
  V(mutex);
  V(empty);
  return v;
}
```

We assume that `sem_t` is the abstract data type that represents a semaphore, and allow a semaphore to be initialized with an integer that represent its initial value.

Meaning of the Semaphores

This solution uses three semaphores:

`mutex`	guarantees that producers and consumers do not access the shared buffer and its indexes at the same time.
`empty`	counts the number of empty slots in the buffer. The producers sleep on it when the buffer is completely full.
`full`	counts the number of full slots in the buffer. The consumers sleep on it when the buffer is completely empty.

Semaphore Usage Paradigms

In the example, semaphores have been used in **two** different ways, that should not be confused:

1. The **`mutex`** semaphore is used for **mutual exclusion**
   - The initial value of those semaphores is always 1.
   - *P*() and *V*() are placed, like “brackets”, around the critical regions.
2. The **`empty`** and **`full`** semaphores are needed to ensure that certain sequences of events do or do not occur, hence they are used for **condition synchronization**.

Criticisms of Semaphores

- Although semaphores are an elegant, and correct, solution to any synchronization problem, they are a **low-level** mechanism.
- Building a large real-time application only upon them is error-prone.
- For example, if an occurrence of a semaphore primitive is either missed or misplaced, the entire application will no longer run correctly.
- Problems may appear only in rare circumstances, possibly depending on the timing of events external to the system, hence they may be difficult to identify and fix.

What is required is a **more structured** synchronization primitive.

Monitors

The **monitor**, proposed by Hoare (1974) and Brinch Hansen (1975), is an higher-level synchronization primitive, and thus easier to use than the semaphore.

- A monitor is a collection of **procedures** (also called **methods**), **data structures** and **condition variables**.
- Processes are allowed to call the monitor's procedure, but cannot access the monitor's data structures from procedures defined outside the monitor itself.
- All procedure calls into the monitor are implicitly guaranteed to execute with **mutual exclusion**.

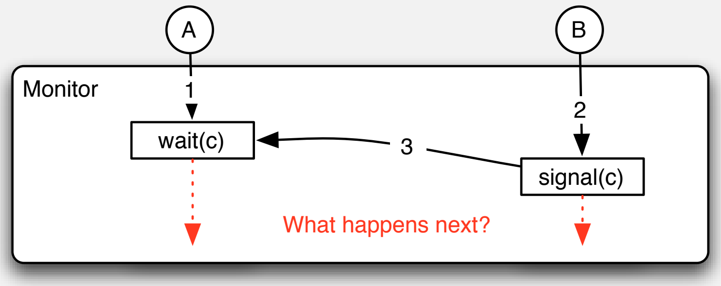
Condition Variables

In the monitor, mutual exclusion is implicit, but there is still a need for **condition synchronization** within the monitor; **condition variables** are used for this.

A condition variable can be accessed only within the monitor it belongs to, by means of two operations:

- **`wait(c)`** atomically blocks the calling process on the condition variable `c` and releases the mutual exclusive hold on the monitor. Unlike *P*(), `wait` **always** blocks the caller.
- **`signal(c)`** unblocks one of the processes blocked on the condition variable `c`, if any. Otherwise, unlike *V*(), it has **no effect**.

What Happens after Signal()?



1. Process *A* enters the monitor, executes `wait(c)`, and blocks.
2. Process *B* enters the monitor and executes `signal(c)`.
3. As a consequence, it wakes *A* up. Then, *A* and *B* are **both** executing inside the monitor.

The semantics of `wait()` and `signal()` just given are not complete. To **avoid race conditions**, it is necessary to carefully define what happens after a `signal()`.

Hoare's Approach

Hoare proposed that a `signal` operation which unblocks another process has the effect of **blocking** its caller until the monitor is free again.

- A `signal` may appear **anywhere** in a monitor procedure.
- A process executing a `signal` must be prepared to be **blocked**.
- The process that are blocked because of a `signal` action are placed on an “urgent” queue, and are chosen in preference to processes blocked on monitor entry, when the monitor is free again.

Brinch Hansen's Approach

Brinch Hansen proposed to circumvent the problem by requiring that a process executing a `signal` **must exit** the monitor immediately.

- A `signal` may appear at most **once** in a monitor procedure.
- It must be the **last** statement in that procedure.
- Conceptually simpler and easier to implement than Hoare's approach.

Solving the P/C Problem Again

```
#define N 100
monitor ProduceConsume
{
  condition full, empty;
  int in = 0, out = 0;
  int count = 0;
  int buf[N];

  void produce(int v)
  {
    if(count == N)
        wait(full);
    buf[in] = v;
    in = (in + 1) % N;
    count = count + 1;
    if(count == 1)
        signal(empty);
  }
```

```
  int consume(void)
  {
    int v;
    if(count == 0)
        wait(empty);
    v = buf[out];
    out = (out + 1) % N;
    count = count - 1;
    if(count == N-1)
        signal(full);
    return v;
  }
}
```

Final Notes on Monitors

- Monitors are a **programming language** construct: the compiler must recognize them and arrange for mutual exclusion.
- Unfortunately, they are **not** supported by the C language. Hence, the previous example has been written as “pseudo C” code, by introducing several fake keywords.
- The C language does not have semaphores either, but supporting semaphores is easy because $P()$ and $V()$ can be compiled as they were normal library calls.
- The IEEE Std 1003.1 (POSIX) has C-language monitors, but an appropriate **programming discipline** is needed to use them correctly, and **no encapsulation** is provided.

Motivation of Message Passing

Message passing is an alternative to shared variable synchronization and communication.

- A **single** construct is used for **both** synchronization and communication.
- It can be used even without sharing memory. This is important, for example, when we go to a distributed system.
- It does not require support at the programming language level.

Theme and Variations

Message Passing Primitives

Most message passing systems foresee two primitives:

`send`	to send a message to a given destination, and
`receive`	to receive a message from a given source, or from any.

Within this broad scheme, several semantics variations exist, mainly concerned with:

- The model of synchronization.
- The method of process naming and message addressing.
- The message structure.

Model of Synchronization

- With all message passing systems, there is an implicit synchronization in `receive`: the receiving process cannot get a message before that message has been sent.
- Of course, **non-blocking** variants of `receive` may exist, but they do not perform synchronization at all.
- More variations are possible on the semantics of `send`, that may be:
  - **Asynchronous** or **no-wait**: the sender proceeds immediately, regardless of whether the message has been received or not.
  - **Synchronous** or **rendezvous**: the sender blocks until the message has been received.
  - **Remote invocation** or **extended rendezvous**: the sender blocks until a reply has been returned from the receiver.

Relationships between the Variants of Send

There is a relationship between the three forms of `send` we just discussed.

1. Two asynchronous interactions can constitute a synchronous relationship if an acknowledge message is always sent by the receiver, and always waited for by the sender.
2. Two synchronous communications, in opposite directions, can be used to construct a remote invocation.

Since the asynchronous model can be used to construct the other two, it is often the model that languages and operating systems adopt, even if some Authors argue that it gives “too much freedom” to the programmer and makes applications more difficult to **understand** and to **prove correct**.

Message Buffers

Asynchronous or synchronous?

The choice between the asynchronous and the synchronous variations of `send` also influences the amount of **buffering** needed to implement the mechanism.

- For the synchronous `send`, **no buffers** are needed, because the message can be directly transferred from one process to the other.
- For the asynchronous `send`, we need potentially **infinite buffers** to store messages that have not been received yet.
- Since the latter situation is clearly undesirable, another variation on the semantics of `send` is to set an **upper limit** on the number of messages that may be queued for reception (or on their size), and make `send` **fail** when the limit is exceeded.

Direct vs. Indirect Naming Schemes

- In a **direct** naming scheme, the sender of a message directly and explicitly names the receiver in the invocation of `send`.
- With an **indirect** naming scheme, the sender names some intermediate entity, often known as a **mailbox**.
- Direct naming has the advantage of being simpler, indirect naming aids the decomposition of the software, because the mailbox can be seen as the interface between two program modules.
- Even with mailboxes, message passing can still be synchronous, even if mailboxes are often associated with message buffers.

Symmetric vs. Asymmetric Naming Schemes

- A naming scheme is **symmetric** if both sender and receiver name each other, either directly or through a mailbox.
- Instead, it is **asymmetric** if the receiver names no specific source but accepts messages from any process or mailbox.
- Asymmetric naming fits better the client-server paradigm, in which a server waits for requests from any of a number of clients. In this case, the implementation must support a queue of clients waiting for the server.
- When combined with indirect naming, asymmetric naming becomes more complex, because the intermediary mailbox could have either a one-to-one, a one-to-many, a many-to-one, or a many-to-many structure.

Message Structure

In an ideal world. . .

A language should allow **any** data object of **any** type, even a user-defined type, to be transmitted in a message, but. . .

- Data objects may have different **representations** at the sender and receiver (for example, big endian vs. little endian).
- Arbitrary objects may include **pointers**, that are no longer meaningful if they are naively transported from one process to another.

- Some languages restrict message contents to unstructured, fixed-size objects, whose type is well-known to the system.
- Other languages remove these restrictions, but still require data to be converted into a stream of bytes before transmission and make the programmer responsible of preserving the meaning of the message across the transmission.

Design Issues for Message Passing Systems

Message passing systems have many challenging design issue which do not arise with semaphores or monitors, **especially** if processes are communicating across a network.

- Message can be **lost** by the network, hence some form of acknowledgment, retransmission and message numbering must be provided.
- **Authentication** is an issue, too: how can a process tell that it is really communicating with its intended partner, and not with an impostor?
- Last, **performance** issues are important, even when the sender and the receiver are on the same machine.

Message Passing, an Example

In this example, we assume that:

- The send operation is asynchronous, with a buffer that is able to hold at least `N` messages.
- The naming scheme is direct (by means of the process identifiers `producer` and `consumer`) and symmetric.
- All messages are the same size, their type is `message_t`, and can contain an `int`.
- `receive` blocks the caller if there are no messages waiting to be retrieved.
- We only consider **one** producer and **one** consumer.

Solving the P/C Problem with Message Passing

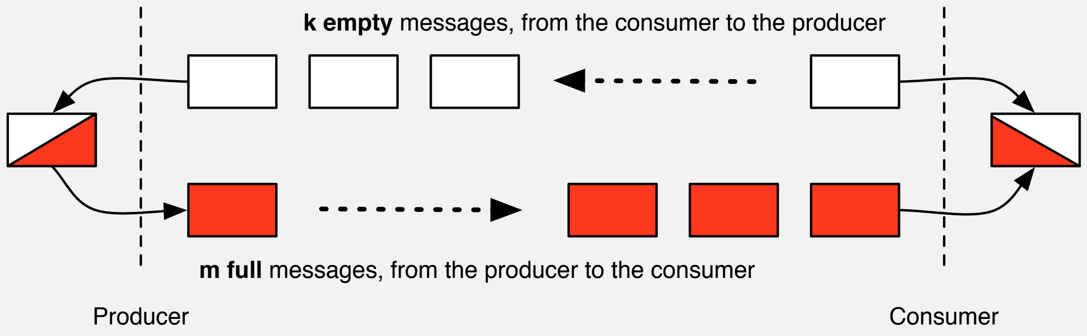
```
#define N 100
void producer(void)
{
  int item;
  message_t m;

  while(1) {
    prod_item(&item);
    receive(consumer, &m);
    build_msg(&m, item);
    send(consumer, &m);
  }
}
```

```
void consumer(void)
{
  int item, i;
  message_t m;

  for(i=0; i<N; i++) send(producer, &m);
  while(1) {
    receive(producer, &m);
    extract_item(&m, &item);
    send(producer, &m);
    cons_item(item);
  }
}
```

Notes to the Solution

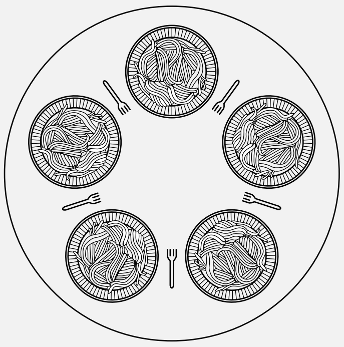
- The functions `build_msg()` and `extract_item()` construct a message to send, and extract the information item from a received message, respectively.
- The consumer starts out by sending `N` empty messages to the producer.
- Whenever the producer has a message to send to the consumer, it first waits for an empty message and then sends back a full one.
- Symmetrically, the consumer sends an empty message back to the producer after receiving a full one and consuming its contents.
- As a consequence, the total number of messages in the system remains (almost) constant in time. This guarantees that:
  - The amount of **memory** used to store these messages remains **constant** as well.
  - A **flow control** mechanism is established between the producer and the consumer.

Flow Control


k empty messages, from the consumer to the producer
m full messages, from the producer to the consumer
Producer
Consumer

- The producer blocks if $k = 0$
- The consumer blocks if $m = 0$
- $N - 1 \leq k + m \leq N$

The Dining Philosophers (Dijkstra)



- Five philosophers are seated around a circular table. Each of them has a plate of spaghetti, and there is a fork between each pair of plates.
- The life of a philosopher consists of alternate periods of thinking and eating.
- The spaghetti is slippery, so each philosopher needs two forks to eat it.

The problem is. . .

Write a program for each philosopher that allows all of them to survive and never gets stuck.

An example of Deadlock

```
#define N 5

void philosopher(int i)
{
   while(TRUE) {
      think();
      take_fork(i); take_fork((i+1) % N);
      eat();
      put_fork(i); put_fork((i+1) % N);
   }
}
```

- `take_fork(i)` waits until fork `i` is available, and then takes it.
- `put_fork(i)` puts fork `i` back on the table.

Deadlock
If all five philosophers take their left fork simultaneously, none of them will ever be able to take their right fork.

Readers and Writers (Courtois et al.)

- A set of processes accesses some shared data either to read, or to write them.
- It is acceptable to have multiple processes (the **readers**) reading the data at the same time.
- Instead, if one process (a **writer**) is writing the data, no other process may access to the data, not even readers.

Useful to model, for example, an airline reservation system.

An Example of Starvation (I)

```
sem_t mutex=1, db=1;
int rc=0;

void enter_reader(void) {
   P(mutex);
      rc = rc + 1;
      if(rc == 1) P(db);
   V(mutex);
}

void exit_reader(void) {
   P(mutex);
      rc = rc - 1;
      if(rc == 0) V(db);
   V(mutex);
}
```

```
void enter_writer(void) {
   P(db);
}

void exit_writer(void) {
   V(db);
}
```

Readers (writers) use `enter_reader`/`exit_reader` (`enter_writer`, `exit_writer`) to delineate the code that reads (writes) the shared data.

An Example of Starvation (I)

```
sem_t mutex=1, db=1;
int rc=0;

void enter_reader(void) {
   P(mutex);
      rc = rc + 1;
      if(rc == 1) P(db);
   V(mutex);
}

void exit_reader(void) {
   P(mutex);
      rc = rc - 1;
      if(rc == 0) V(db);
   V(mutex);
}
```

```
void enter_writer(void) {
   P(db);
}

void exit_writer(void) {
   V(db);
}
```

`mutex` guarantees mutual exclusion among the readers when they access the shared variable `rc` that counts how many active readers there are.

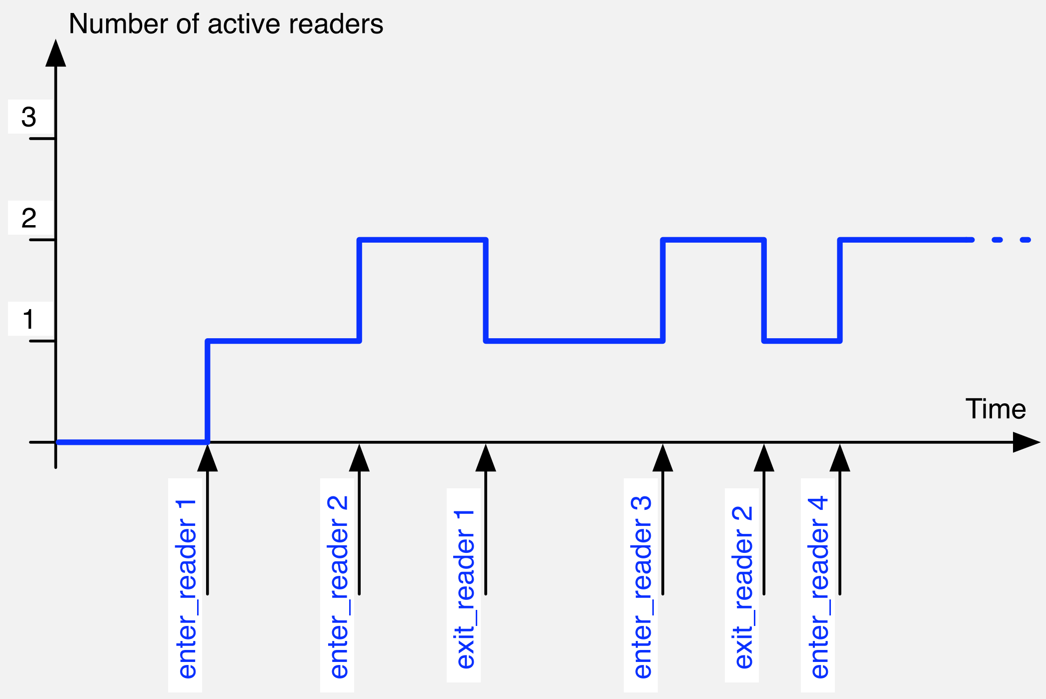
An Example of Starvation (I)

```
sem_t mutex=1, db=1;
int rc=0;

void enter_reader(void) {
   P(mutex);
      rc = rc + 1;
      if(rc == 1) P(db);
   V(mutex);
}

void exit_reader(void) {
   P(mutex);
      rc = rc - 1;
      if(rc == 0) V(db);
   V(mutex);
}
```

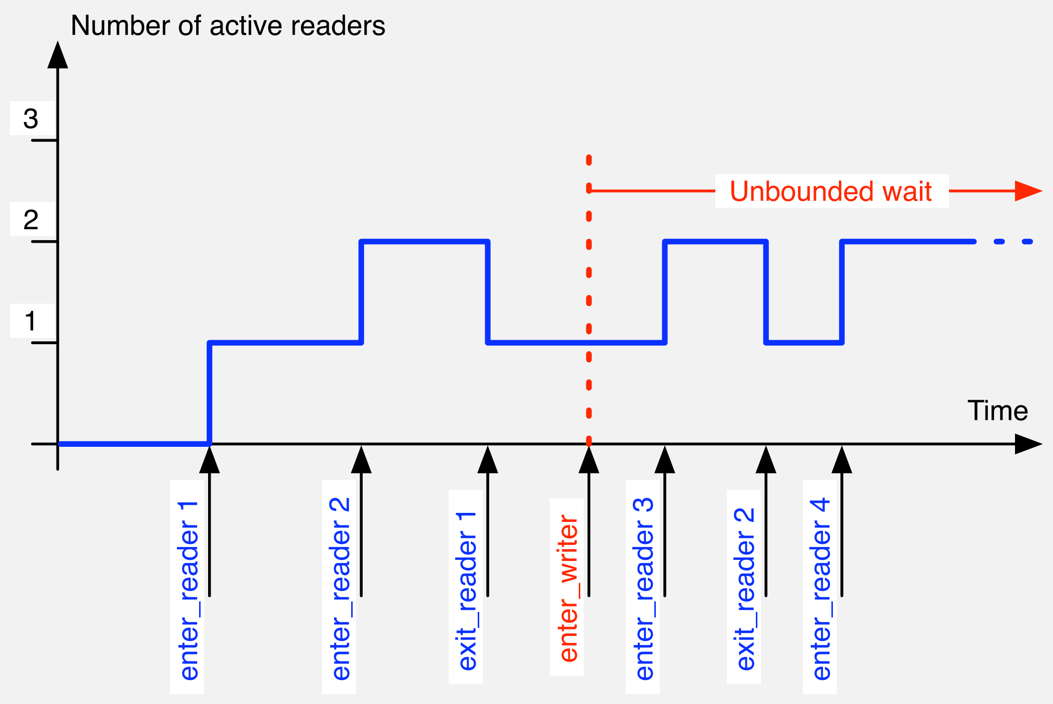
```
void enter_writer(void) {
   P(db);
}

void exit_writer(void) {
   V(db);
}
```

db ensures mutual exclusion among writers, as well as between a writer and a group of readers.

An Example of Starvation (II)



An Example of Starvation (II)



Definition of Starvation

Starvation

Starvation, also called **indefinite postponement** or **lockout**, is a condition whereby a process that wishes to gain access to a resource is never allowed to do so because there are always other processes gaining access before it.

- Starvation does not necessarily involve all the processes that compete for a resource, but just a subset of them.
- Starvation can be avoided by using a first-come, first-served resource allocation policy.
- With this approach, the process waiting the longest gets served next and, in due course of time, any given process will get the resource.
- Generally speaking, introducing **priorities** or **cost factors** into the resource assignment policy (like is often done in real-time systems) increases the likelihood of starvation.