Operating Systems / Sistema de Operação

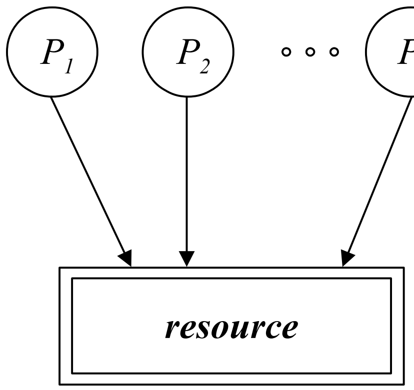
Interprocess communication

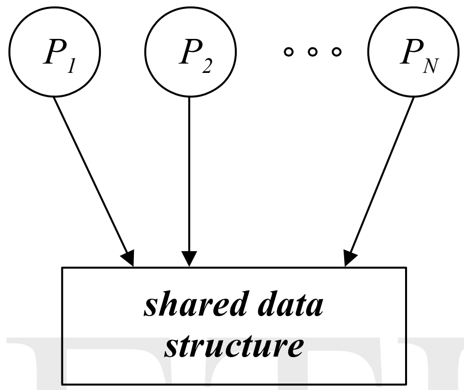
António Rui Borges / Artur Pereira

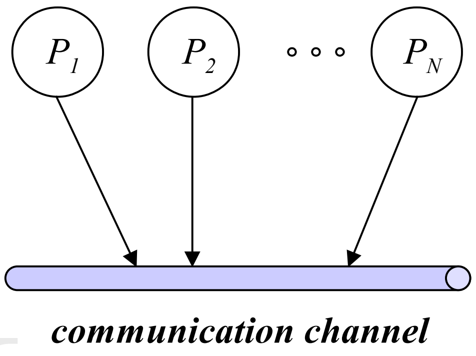
Concepts

- In a multiprogrammed environment, two or more processes can be:
  - *independent* – if they, from their creation to their termination, never explicitly interact
    - actually. there is an implicit interaction, as they compete for system resources
    - ex: jobs in a batch system; processes from different users
  - *cooperative* – if they share information or explicitly communicate
    - the *sharing* requires a common address space
    - *communication* can be done through a common address space or a communication channel connecting them

Concepts





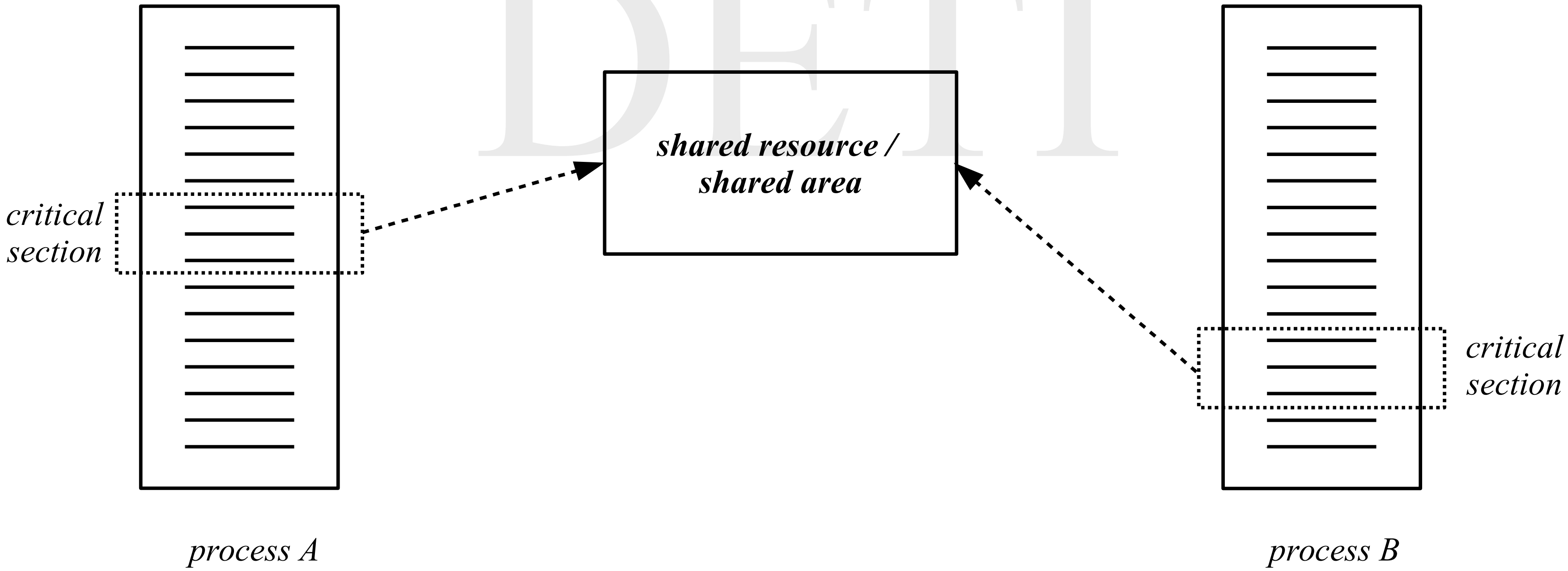


- *independent processs* competing for a resource
- is responsability of the OS to guarantee that the assignment of resources to processes is done in a controlled way, such that no information lost occurs
- in general, this imposes that only one process can use the resource at a time -- mutual exclusive access

- *cooperative processes* sharing information or communicating
- is responsability of the processes to guarantee that access to the shared area is done in a controlled way, such that no information lost occurs
- in general, this imposes that only one process can use the resource at a time -- mutual exclusive access
- the communication channel is typically a system resource; so processes compete for it

Concepts

- Having access to a resource or to a shared area, actually means executing the code that do the access
- This code, because needs to avoid **race conditions** (that result in lost of information), is called **critical section**



Concepts

- Mutual exclusion in the access to a resource or shared area can result in:
  - ***deadlock*** – when two or more processes are waiting forever for access to their respective critical section, waiting for events that can be demonstrated will never happen
    - operations are blocked
  - ***starvation*** – when one or more processes compete for access to a critical section and, due to a conjunction of circumstances in which new processes that exceed them continually arise, access is successively deferred
    - operations are continuously postponed

Access to a resource

```
/* processes competing for a resource - p = 0, 1, ..., N-1 */
void main (unsigned int p)
{
  forever
  {
    do_something();
    access_resource(p);
    do_something_else();
  }
}
```

```
enter_critical_section(p);
use_resource();
leave_critical_section(p);
```

critical section

Access to a shared area

```
/* shared data structure */
shared DATA d;
/* processes sharing data - p = 0, 1, ..., N-1 */
void main (unsigned int p)
{
  forever
  {
    do_something();
    access_shared_area(p);
    do_something_else();
  }
}
```

```
enter_critical_section(p);
manipulate_shared_area();
leave_critical_section(p);
```

critical section

Producer / consumer relationship

```
/* communicating data structure: FIFO of fixed size */

shared FIFO fifo;

/* producer processes - p = 0, 1, ..., N-1 */
void main (unsigned int p)
{
    DATA val;
    bool done;

    forever
    {
        produce_data(&val);
        done = false;
        do
        {
            enter_critical_section(p);
            if (fifo.notFull())
            {
                fifo.insert(val);
                done = true;
            }
            leave_critical_section(p);
        } while (!done);
        do_something_else();
    }
}
```

critical section

Producer / consumer relationship

```
/* communicating data structure: FIFO of fixed size */

shared FIFO fifo;

/* consumer processes - p = 0, 1, ..., M-1 */
void main (unsigned int p)
{
    DATA val;
    bool done;

    forever
    {
        done = false;
        do
        {
            enter_critical_section(p);
            if (fifo.notEmpty())
            {
                fifo.retrieve(&val);
                done = true;
            }
            leave_critical_section(p);
        } while (!done);

        consume_data(val);
        do_something_else();
    }
}
```

critical section

Access to a critical section

- Requirements that should be observed in accessing a critical section:
  - effective mutual exclusion – access to the critical sections associated with the same resource, or shared area, can only be allowed to one process at a time, among all processes that compete for access
  - independence on the number of intervening processes or on their relative speed of execution
  - a process outside the critical section can not prevent another from entering there
  - a process requiring access to the critical section should not have to wait indefinitely
  - length of stay inside a critical section should be necessarily finite

Type of solutions

- In general, a memory location is used to control access to the critical region
- ***software solutions*** – solutions that are based on the typical instructions to the access memory location
  - read and write are done by different instructions
- ***hardware solutions*** – solutions that are based on special instructions to access the memory location
  - these instructions allow to read and then write a memory location in an atomic way

Strict alternation

```
/* control data structure */
#define  R     ...     /* process id = 0, 1, …, R-1 */

shared unsigned int access_turn = 0;

void enter_critical_section(unsigned int own_pid)
{
  while (own_pid != access_turn);
}

void leave_critical_section(unsigned int own_pid)
{
  if (own_pid == access_turn)
     access_turn = (access_turn + 1) % R;
}
```

Strict alternation

- Not a valid solution
  - Dependence on the relative speed of execution of the intervening processes
    - The process with less accesses imposes its rhythm to the others
  - A process outside the critical section can prevent another from entering there
    - If it is not its turn, a process has to wait, even if no one else wants to enter

Constructing a solution

```
/* control data structure */
#define  R    2          /* process id = 0, 1 */

shared bool is_in[R] = {false, false};

void enter_critical_section(unsigned int own_pid)
{
    unsigned int other_pid_ = 1 - own_pid;

    while (is_in[other_pid]);
    is_in[own_pid] = true;
}

void leave_critical_section(unsigned int own_pid)
{
  is_in[own_pid] = false;
}
```

Constructing a solution

- Not a valid solution
  - Mutual exclusion is not guaranteed
  - Assume that:
    - P_0 enters `enter_critical_section` and tests `is_in[1]`, which is *false*
    - P_1 enters `enter_critical_section` and tests `is_in[0]`, which is *false*
    - P_1 changes `is_in[0]` to *true* and enters its critical section
    - P_0 changes `is_in[1]` to *true* and enters its critical section
    - Thus, both processes enter the critical sections

  - It seems that the failure is a result of testing first the other's control variable and then change its own variable

Constructing a solution

```
/* control data structure */
#define  R    2             /* process pid = 0, 1 */

shared bool want_enter[R] = {false, false};

void enter_critical_section (unsigned int own_pid)
{
  unsigned int other_pid = 1 - own_pid;

  want_enter[own_pid] = true;
  while (want_enter[other_pid]);
}

void leave_critical_section (unsigned int own_pid)
{
  want_enter[own_pid] = false;
}
```

Constructing a solution

- Not a valid solution
  - Mutual exclusion is guaranteed, but deadlock can occur
  - Assume that:
    - P_0 enters `enter_critical_section` and sets `want_enter[0]` to *true*
    - P_1 enters `enter_critical_section` and sets `want_enter[1]` to *true*
    - P_1 tests `want_enter[0]` and, because it is *true*, keeps waiting to enter its critical section
    - P_0 tests `want_enter[1]` and, because it is *true*, keeps waiting to enter its critical section
    - Thus, both processes enter in deadlock

  - To solve the deadlock at least one of the processes have to go back

Constructing a solution

```
/* control data structure */
#define  R    2          /* process id = 0, 1 */

shared bool want_enter[R] = {false, false};

void enter_critical_section(unsigned int own_pid)
{
  unsigned int other_pid = 1 - own_pid;

  want_enter[own_pid] = true;
  while (want_enter[other_pid])
  {
    want_enter[own_pid] = false;
    random_delay();
    want_enter[own_pid] = true;
  }
}

void leave_critical_section(unsigned int own_pid)
{
  want_enter[own_pid] = false;
}
```

Constructing a solution

- An almost valid solution
  - The Ethernet protocol uses a similar approach to control access to the communication medium
- But, still not completely valid
  - Even if unlikely, deadlock and starvation can still be present
- The solution needs to be deterministic, not random

Dekker algorithm (1965)

```
#define  R    2    /* process id = 0, 1 */

shared bool want_enter[R] = {false, false};
shared uint p_w_priority = 0;
```

```
void enter_critical_section(uint own_pid)
{
   uint other_pid = 1 - own_pid;

   want_enter[own_pid] = true;
   while (want_enter[other_pid])
   {
      if (own_pid != p_w_priority)
      {
         want_enter[own_pid] = false;
         while (own_pid != p_w_priority);
         want_enter[own_pid] = true;
      }
   }
}
```

```
void leave_critical_section(uint own_pid)
{
   uint other_pid = 1 - own_pid;
   p_w_priority = other_pid;
   want_enter[own_pid] = false;
}
```

***Dekker algorithm* (*1965*)**

- The algorithm uses an alternation mechanism to solve the conflict
- Mutual exclusion in the access to the critical section is guaranteed
- Deadlock and starvation are not present
- No assumptions are done in the relative speed of the intervening processes

- However, it can not be generalized to more than 2 processes, satisfying all the requirements

Dijkstra algorithm (1966)

```
#define  R    ...   /* process id = 0, 1, ..., R-1 */

shared uint want_enter[R] = {NO, NO, ... , NO};
shared uint p_w_priority = 0;

void enter_critical_section(uint own_pid)
{
   uint n;
   do
   {
      want_enter[own_pid] = WANT;
      while (own_pid != p_w_priority)
         if (want_enter[p_w_priority] == NO)
            p_w_priority = own_pid;
      want_enter[own_pid] = DECIDED;
      for (n = 0; n < R; n++)
         if (n != own_pid && want_enter[n] == DECIDED)
            break;
   } while (n < R);
}
```

```
void leave_critical_section(uint own_pid)
{
  p_w_priority = (own_pid + 1) % R;
  want_enter[own_pid] = NO;
}
```

- Can suffer from starvation

Peterson algorithm (1981)

```
#define  R    2     /* process id = 0, 1 */

shared bool want_enter[R] = {false, false};
shared uint last;


void enter_critical_section(uint own_pid)
{
    uint other_pid = 1 - own_pid;

    want_enter[own_pid] = true;
    last = own_pid;
    while ((want_enter[other_pid]) && (last == own_pid));
}

void leave_critical_section(uint own_pid)
{
    want_enter[own_pid] = false;
}
```

Peterson algorithm (1981)

- The Peterson algorithm uses the order of arrival to solve conflicts
  - Each process has to write its ID in a shared variable (last)
  - The subsequent reading allows to determine which was the last one
- It is a valid solution
  - Guarantees mutual exclusion
  - Avoids deadlock and startvation
  - Make no assumption about the relative speed of intervening processes
- Can be generalized to more than processes
  - The general solution is similar to a waiting queue

Generalized Peterson algorithm (1981)

```
#define  R    ...     /* process id = 0, 1, ..., R-1 */

shared int want_enter[R] = {-1, -1, ... , -1};
shared int last[R-1];

void enter_critical_section(uint own_pid)
{
   for (uint i = 0; i < R-1; i++)
   {
      want_enter[own_pid] = i;
      last[i] = own_pid;
      do
      {
         test = false;
         for (uint j = 0; j < R; j++)
            if (j != own_pid)
               test = test || (want_enter[j] >= i);
      } while (test && (last[i] == own_pid));
   }
}
```

```
void leave_critical_section(int own_pid)
{
   want_enter[own_pid] = -1;
}
```

Hardware solutions - disabling interrupts

<u>Uniprocessor computational system</u>

- The switching of processes, in a multiprogrammed environment, is always caused by an external device:
  - *real time clock* (*RTC*) – causing the time-out transition in preemptive systems
  - *device controller* – can cause the preemp transitions in case of wake up of a higher priority process
  - In any case, interruptions of the processor
- Thus, access in mutual exclusion can be implemented disabling interrupts
- Only valid in kernel
  - Malicious or buggy code can completely block the system

<u>Multiprocessor computational system</u>

- Disabling interrupts in one processor has no effect

Hardware solutions - special instructions

```
shared bool flag = false;

bool test_and_set(bool * flag)
{
   bool prev = *flag;
   *flag = true;
   return prev;
}


void lock(bool * flag)
{
   while (test_and_set(flag);
}


void unlock(bool * flag)
{
   *flag = false;
}
```

- The **test_and_set** function, if implemented atomically (without interruptions), can be used to construct the lock (enter critical section) primitive
- In the instruction set of some of the current processors, there is an atomic instruction implementing that behavior
- Surprisingly, it is often called TAS (test and set)

Hardware solutions - special instructions

```
shared int value = 0;

int compare_and_swap(int * value, int expected, int new_value)
{
   int v = *value;
   if (*value == expected)
      *value = new_value;
   return v;
}

void lock(int * flag)
{
   while (compare_and_swap(&flag, 0, 1) != 0);
}


void unlock(bool * flag)
{
   *flag = 0;
}
```

- The **compare_and_swap** function, if implemented atomically (without interruptions), can be used to construct the lock (enter critical section) primitive
- In the instruction set of some of the current processors, there is an atomic instruction implementing that behavior
- In some instruction sets, there is a compare_and_set variant that returns a bool

Busy waiting

- The previous solutions suffer from ***busy waiting*** – the lock primitive is in the active state (using the CPU) while waiting
  - They are often referred as **spinlocks**, as the process spins around the variable while waiting for access

- In uniprocessor systems, busy waiting is unwanted, as there is
  - **loss of efficiency** – the time quantum of a process can be used for nothing
  - **risk of deadlock** – if a higher priority process calls lock while a lower priority process is inside its critical section, none of them can proceed

- In multiprocessor system with shared memory, busy waiting can be less critical
  - switching processes cost time, that can be higher than the time spent by the other process inside its cririical section

block and wake_up

- In general, at least in uniprocessor systems, there is the requirement of blocking a process while it is waiting for entering its critical section

```
#define  R    ...         /* process id = 0, 1, ..., R-1 */

shared unsigned int access = 1;


void enter_critical_section(unsigned int own_pid)
{
  if  (access == 0) block(own_pid);
      else access -= 1;
}
```

atomic operation
(can not be interrupted)

```
void leave_critical_section(unsigned int own_pid)
{
  if  (there_are_blocked_processes)wake_up_one();
      else access += 1;
}
```

atomic operation
(can not be interrupted)

- Atomic operations are still required
- Note that `access` is an integer, not a boolean

Semaphores

- A *semaphore* is a synchronization mechanism, defined by a data type plus two atomic operations, *down* and *up*
  - The operations are also referred to as *wait* and *signal/post*, respectively
- Data type:

```
typedef struct
{
   unsigned int val;        /* can not be negative */
   PROCESS *queue;          /* queue of waiting blocked processes */
} SEMAPHORE;
```

- Operations:
  - ***down*** – block process if `val` is zero; decrement val otherwise
  - ***up*** – if `queue` is not empty, wake up one process (accordingly to a given policy); increment `val` otherwise
- Note that `val` can only be manipulated through these operations

Semaphores

```
/* array of semaphores defined in kernel */
#define  R   ... /* semid = 0, 1, ..., R-1 */

static SEMAPHORE sem[R];

void sem_down(unsigned int semid)
{
    disable_interruptions;
    if (sem[semid].val == 0)
        block_on_sem(getpid(), semid);
    else
        sem[semid].val -= 1;
    enable_interruptions;
}

void sem_up(unsigned int semid)
{
    disable_interruptions;
    if (sem[sem_id].queue != NULL)
        wake_up_one_on_sem(semid);
    else
        sem[semid].val += 1;
    enable_interruptions;
}
```

- This implementation is typical of uniprocessor systems. *Why?*

- Semaphores can be binary or not binary
- How to implement *mutual exclusion* using semaphores?

Bounded-buffer problem

```
shared FIFO fifo;     /* fixed-size FIFO memory */
```

```
/* producers - p = 0, 1, ..., N-1 */

void producer(unsigned int p)
{
    DATA data;
    forever
    {
        produce_data(&data);
        bool done = false;
        do
        {
            lock(p);
            if (fifo.notFull())
            {
                fifo.insert(data);
                done = true;
            }
            unlock(p);
        } while (!done);
        do_something_else();
    }
}
```

```
/* consumers - c = 0, 1, ..., M-1 */

void consumer(unsigned int c)
{
    DATA data;
    forever
    {
        bool done = false;
        do
        {
            lock(c);
            if (fifo.notEmpty())
            {
                fifo.retrieve(&data);
                done = true;
            }
            unlock(c);
        } while (!done);
        consume_data(data);
        do_something_else();
    }
}
```

- How to implement using semaphores?
  - Guaranteeing mutual exclusion and absence of busy waiting

Solving the bounded-buffer problem using semaphores

```
shared FIFO fifo;        /* fixed-size FIFO memory */
shared sem access;       /* semaphore to control mutual exclusion */
shared sem nslots;       /* semaphore to control number of available slots*/
shared sem nitems;       /* semaphore to control number of available items */
```

```
/* producers - p = 0, 1, ..., N-1 */
void producer(unsigned int p)
{
    DATA val;

    forever
    {
        produce_data(&val);
        sem_down(nslots);
        sem_down(access);
        fifo.insert(val);
        sem_up(access);
        sem_up(nitems);
        do_something_else();
    }
}
```

```
/* consumers - c = 0, 1, ..., M-1 */
void consumer(unsigned int c)
{
    DATA val;

    forever
    {
        sem_down(nitems);
        sem_down(access);
        fifo.retrieve(&val);
        sem_up(access);
        sem_up(nslots);
        consume_data(val);
        do_something_else();
    }
}
```

- fifo.empty() and fifo.full() are not necessary. *Why?*
- What are the initial values of the semaphores?

Wrong solution of the bounded-buffer problem

```
shared FIFO fifo;       /* fixed-size FIFO memory */
shared sem access;      /* semaphore to control mutual exclusion */
shared sem nslots;      /* semaphore to control number of available slots*/
shared sem nitems;      /* semaphore to control number of available items */
```

```
/* producers - p = 0, 1, ..., N-1 */
void producer(unsigned int p)
{
    DATA data;
    forever
    {
        produce_data(&data);
        sem_down(access);
        sem_down(nslots);
        fifo.insert(data);
        sem_up(access);
        sem_up(nitems);
        do_something_else();
    }
}
```

```
/* consumers - c = 0, 1, ..., M-1 */
void consumer(unsigned int c)
{
    DATA data;
    forever
    {
        sem_down(nitems);
        sem_down(access);
        fifo.retrieve(&data);
        sem_up(access);
        sem_up(nslots);
        consume_data(data);
        do_something_else();
    }
}
```

- What is wrong with this solution?

Analysis of semaphores

- Concorrent solutions based on semaphores have advantages and disadvantages
- *Advantages:*
  - *support at the operating system level* – operations on semaphores are implemented by the kernel and made available to programmers as *system calls*
  - *general* – they are low level contructions and so they are versatile, being able to be used in any type of solution
- *Disadvantages*
  - *specialized knowledge* – the programmer must be aware of concorrent programming principles, as race conditions or deadlock can be easily introduced
    - See the following example, as an illustration of this

Monitors

- A problem with semaphores is that they are used both to implement mutual exclusion and to synchronize processes
- Being low level primitives, they are applied in a bottom-up perpective
  - if required conditions are not satisfied, processes are blocked before they enter their critical sections
  - this approach is prone to errors, mainly in complex situations, as synchronization points can be scattered throughout the program
- A higher level approach should followed a top-down perpective
  - processes must first enter their critical regions and then block if pursuance conditions are not satisfied
- A solution is to introduce a (concurrent) construction at the programming language level that separately deals with mutual exclusion and synchronization

- A *monitor* is a synchronization mechanism, independently proposed by Hoare and Brinch Hansen, supported by a (concurrent) programming language
  - It is composed of an internal data structure, inicialization code and a number of accessing primitives

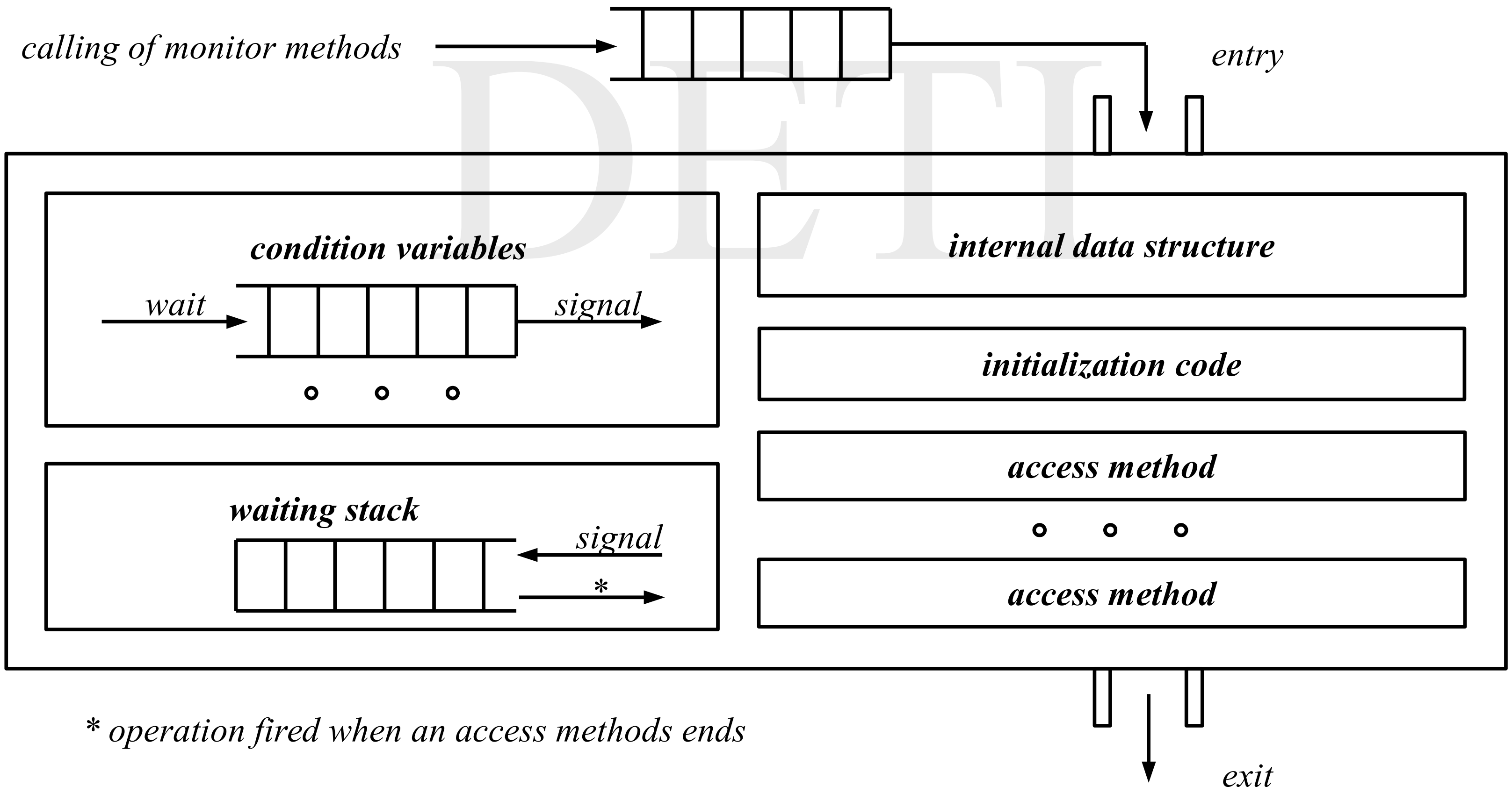
Monitors

```
monitor example
{
   /* internal shared data structure */
   DATA data;

   condition c; /* condition variable */

   /* access methods */
   method_1 (...)
   {
      ...
   }

   method_2 (...)
   {
      ...
   }

   ...

   /* initialization code */
   ...
}
```

- An application is seen as a set of threads that compete to access the shared data structure
- This shared data can only be accessed through the access methods
- Every method is executed in mutual exclusion
- If a thread calls an access method while another thread is inside another access method, its execution is blocked until the other leaves
- Synchronization between threads is possible through *condition variables*
- Two operation on them are possible:
  - *wait* – the thread is blocked and put outside the monitor
  - *signal* – if there are threads blocked, one is waked up. *Which one*?

Hoare monitor

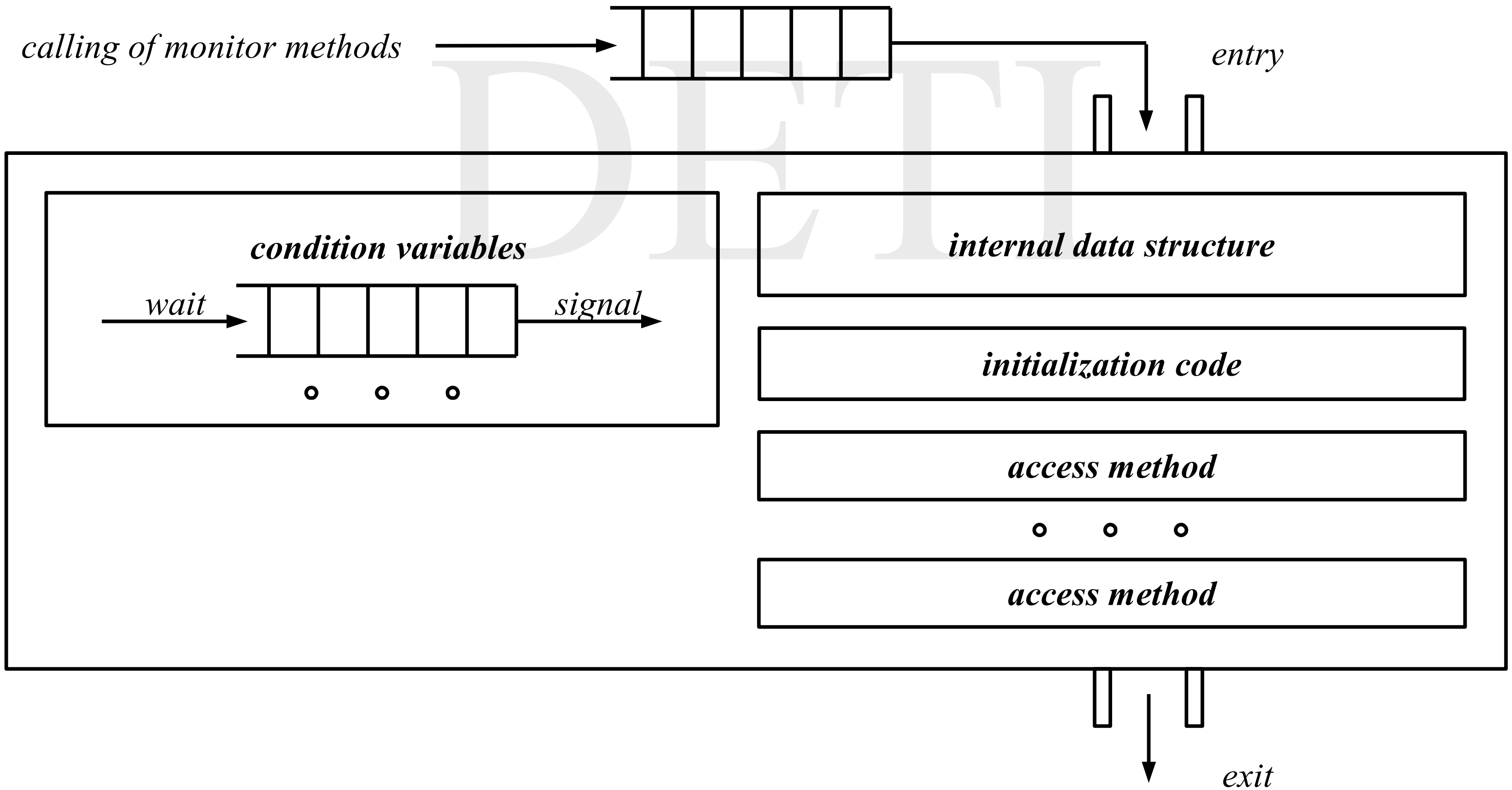
- What to do when *signal* occurs?
- *Hoare monitor* – the thread calling *signal* is put out of the monitor, so the just waked up thread can proceed
  - quite general, but its implementation requires a stack where the blocked thread is put



* operation fired when an access methods ends

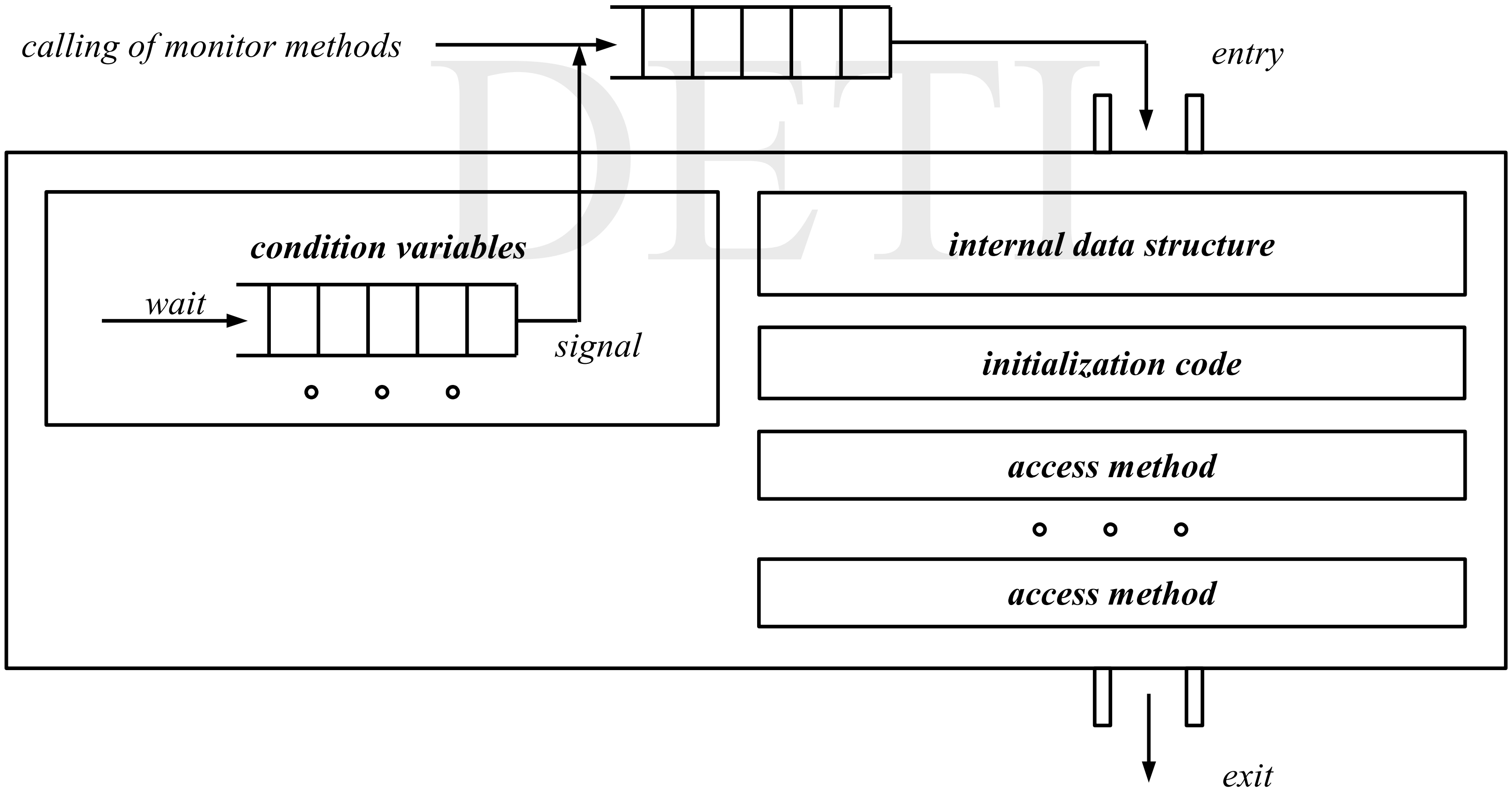
Brinch Hansen monitor

- *Brinch Hansen monitor* – the thread calling *signal* immediately leaves the monitor (*signal* is the last instruction of the monitor method)
  - easy to implement, but quite restrictive (only one signal allowed in a method)



Lampson / Redell monitors

- *Lampson / Redell monitor* – the thread calling signal continues its execution and the just waked up thread is kept outside the monitor, competing for access
  - easy to implement, but can cause starvation



Solving the bounded-buffer problem using monitors

```
shared FIFO fifo;      /* fixed-size FIFO memory */
shared mutex access;   /* mutex to control mutual exclusion */
shared cond nslots;    /* condition variable to control availability of slots*/
shared cond nitems;    /* condition variable to control availability of items */
```

```
/* producers - p = 0, 1, ..., N-1 */
void producer(unsigned int p)
{
   DATA data;
   forever
   {
      produce_data(&data);
      lock(access);
      if/while (fifo.isFull())
      {
         wait(nslots, access);
      }
      fifo.insert(data);
      unlock(access);
      signal(nitems);
      do_something_else();
   }
}
```

```
/* consumers - c = 0, 1, ..., M-1 */
void consumer(unsigned int c)
{
   DATA data;
   forever
   {
      lock(access);
      if/while (fifo.isEmpty())
      {
         wait(nitems, access);
      }
      fifo.retrieve(&data);
      unlock(access);
      signal(nslots);
      consume_data(data);
      do_something_else();
   }
}
```

- fifo.empty() and fifo.full() are now necessary. *Why?*
- What is the initial value of the mutex?

Message-passing

- Processes can communicate exchanging messages
  - A general communication mechanism, not requiring shared memory
  - Valid for uniprocessor and multiprocessor systems
- Two operation are required:
  - **send** and **receive**
- A communication link is required:
  - There are different logical ways of implementing it
    - Direct or indirect (through mailboxes or ports) addressing
    - Synchronous or asynchronous communication
    - Automatic or explicit buffering

Message-passing - direct vs. indirect

- Symmetric direct communication
  - A process that wants to communicate must explicitly name the recipient or sender
    - **send**`(P, message)` - send message to process P
    - **receive**`(P, message)` - receive message from process P
  - A communication link in this scheme has the following properties:
    - it is established automatically between a pair of communicating processes
    - it is associated with exactly two processes
    - between a pair of processes there exist exactly one link
- Asymetric direct communication
  - Only the sender must explicitly name the recipient
    - **send**`(P, message)` - send message to process P
    - **receive**`(id, message)` - receive message from any process

Message-passing - direct vs. indirect

- Indirect communication
  - The messages are sent and received from mailboxes, or ports
    - **`send`**`(M, message)` - send message to mailbox M
    - **`receive`**`(M, message)` - receive message from mailbox M
  - A communication link in this scheme has the following properties:
    - it is only established if the pair of communicating processes has a shared mailbox
    - it may be associated with mote than two processes
    - between a pair of processes there may exist more than one link (a mailbox per each)
  - The problem of two or more processes trying to receive a message from the same mailbox?
    - Is it allowed?
    - If allowed, which one will succeed?

Message-passing - synchronization

- There are different design options for implementing `send` and `receive`
  - **Blocking send** - the sending process blocks until the message is received by the receiving process or by the mailbox
  - **Nonblocking send** - the sending process sends the message and resumes operation.
  - **Blocking receive** - the receiver blocks until a message is available
  - **Nonblocking receive** - the receiver retrieves either a valid message or the indication that no one exits
- Different combinations of `send` and `receive` are possible

Message-passing - buffering

- There are different design options for implementing the link supporting the communication
  - **Zero capacity** - there is no queue
    - the sender must block until the recipient receives the message
  - **Bounded capacity** - the queue has finite length
    - if the queue is full, the sender must block until space is available
  - **Unbounded capacity** - the queue has (potentially) infinite length

Solving the bounded-buffer problem using messages

```
shared FIFO fifo;       /* fixed-size FIFO memory */
shared mutex access;    /* mutex to control mutual exclusion */
shared cond nslots;     /* condition variable to control availability of slots*/
shared cond nitems;     /* condition variable to control availability of items */
```

```
/* producers - p = 0, 1, ..., N-1 */
void producer(unsigned int p)
{
   DATA data;
   MESSAGE msg;

   forever
   {
      produce_data(&val);
      make_message(msg, data);
      send(msg);
      do_something_else();
   }
}
```

```
/* consumers - c = 0, 1, ..., M-1 */
void consumer(unsigned int c)
{
   DATA data;
   MESSAGE msg;

   forever
   {
      receive(msg);
      extract_data(data, msg);
      consume_data(data);
      do_something_else();
   }
}
```

Semaphores in Unix/Linux

- POSIX semaphores
  - down and up
    - `sem_wait`, `sem_trywait`, `sem_timedwait, sem_post`
  - Two types: named and unnamed semaphores
  - Named semaphores
    - `sem_open`, `sem_close`, `sem_unlink`
    - created in a virtual filesystem (e.g., /dev/sem)
  - unnamed semaphores - memory based
    - `sem_init`, `sem_destroy`
  - execute `man sem_overview` for an overview
- System V semaphores
  - creation: `semget`
  - down and up: `semop`
  - other operations: `semctl`

POSIX support for monitors

- Standard *POSIX*, *IEEE 1003.1c*, defines a programming interface (API) for the creation and synchronization of *threads*.
  - In unix, this interface is implemented by the *pthread* library
- It allows for the implementation of monitors in C/C++
  - Using mutexes and condition variables
  - Note that they are of the Lampson / Redell type
- Some of the available functions:
  - *`pthread_create`* – creates a new thread; similar to `fork`
  - *`pthread_exit`* – equivalent to `exit`
  - *`pthread_join`* – equivalent a `waitpid`
  - *`pthread_self`* – equivalent a `getpid()`
  - *`pthread_mutex_*`* – manipulation of mutexes
  - *`pthread_cond_*`* – manipulation of condition variables
  - *`pthread_once`* – inicialization

Message-passing in Unix/Linux

- System V implementation
  - Defines a message queue where messages of diferent types (a positive integer) can be stored
  - The send operation blocks if space is not available
  - The receive operation has an argument to specify the type of message to receive: a given type, any type or a range of type
    - The oldest message of given type(s) is retrieved
    - Can be blocking or nonblocking
  - see system calls: `msgget`, `msgsnd`, `msgrcv`, and `msgctl`
- POSIX message queue
  - Defines a priority queue
  - The send operation blocks if space is not available
  - The receive operation removes the oldest message with the highest priority
    - Can be blocking or nonblocking
  - see functions: mq_open, mq_send, mq_receive, ...

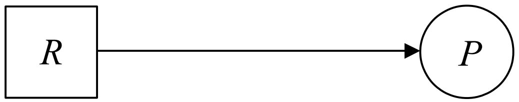
Shared memory in Unix/Linux

- Address spaces are independent
- But address spaces are virtual
- The same physical region can be mapped into two or more virtual regions
- This is managed as a resource by the operating system
- POSIX shared memory
  - creation - shm_open, ftruncate
  - mapping - mmap, munmap
  - other operations - close, shm_unlink, fchmod, ...
- System V shared memory
  - creation - shmget
  - mapping - shmat, shmdt
  - other operations - shmctl

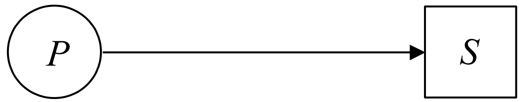
Deadlock

- Generically, a *resource* is something a process needs in order to proceed with its execution
  - *physical components of the computational system* (processor, memory, I/O devices, etc)
  - *common data structures* defined at the operating system level (PCT, communication channels, etc,) or among processes of a given application
- Resources can be:
  - *preemptable* – if they can be withdraw from the processes that hold them
    - ex: processor, memory regions used by a process address space
  - *non-preemptable* – if they can only be released by the processes that hold them
    - ex: a printer, a shared memory region that requires exclusive access for its manipulation
- Em termos o deadlock, onde non-preemptable resources are relevant

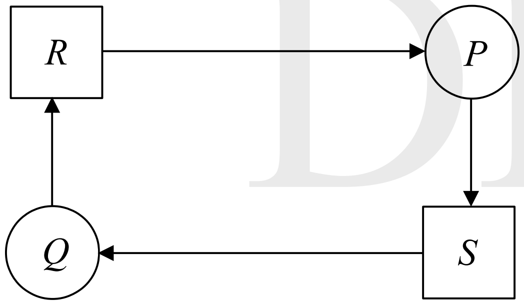
Deadlock



process P holds resource R in its possession



process P requests resource S



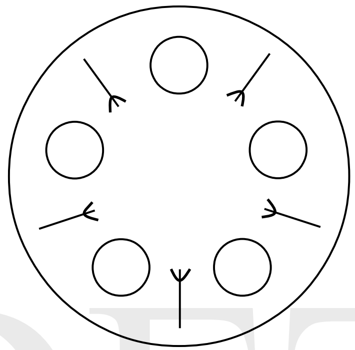
typical deadlock situation
(*the simplest one*)

- What are the conditions for the occurrence of deadlock

Necessary conditions for deadlock

- It can be proved that when *deadlock* occurs 4 conditions are necessarely observed:
  - *mutual exclusion* – at least one resource must be held in a nonsharable mode
    - if another process requests that, it must wait until it is released
  - *hold and wait* – a process must be holding at least one resource, while waiting for another that is being held by another process
  - *no preemption* - resources are non-preemptable
    - only the process holding a resource can release it
  - *circular wait* - a set of waiting processes must exist such that each one is waiting for resources held by other processes in the set

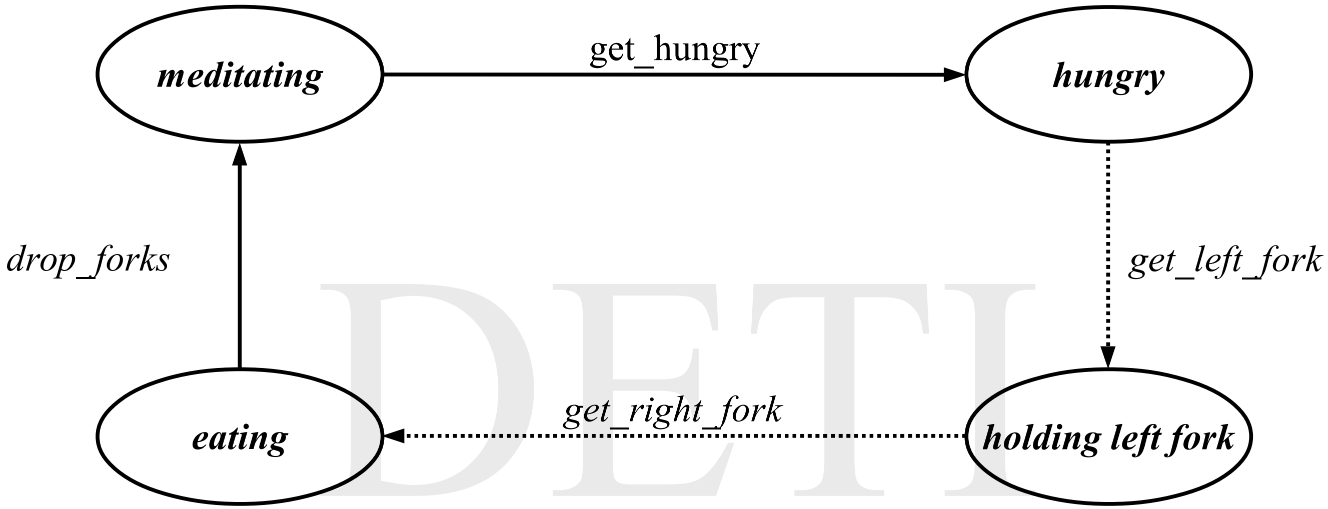
Necessary conditions for deadlock



Problem statement

- 5 philosophers are seated around a table, with food in from of them
  - To eat, every philosopher needs two forks, the ones at her left and right sides
  - Every philosopher alternates periods in which she medidates with periods in which she eats
- Modelling every philosopher as a different process or thread and the forks as resources, design a solution for the problem

Necessary conditions for deadlock



- This is a possible solution for the dining-philosopher problem
  - when a philosopher gets hungry, she first gets the left fork and then holds it while waits for the right one
- This solution can suffer from deadlock
  - Try to identify the four necessary conditions

Necessary conditions for deadlock

```
enum {MEDITATING, HUNGRY, HOLDING, EATING};

typedef struct TablePlace
{
        int state;
} TablePlace;

typedef struct Table
{
        Int semid;
        int nplaces;
        TablePlace place[0];
} Table;

int set_table(unsigned int n, FILE *logp);
int get_hungry(unsigned int f);
int get_left_fork(unsigned int f);
int get_right_fork(unsigned int f);
int drop_forks(unsigned int f);
```

- Let's look at the code!

Necessary conditions for deadlock

- This solution works most of the times
  - But, it can suffer from deadlock
- The four necessary conditions for the occurrence of deadlock are satisfied
  - *mutual exclusion* – the forks are sharable
  - *wait and hold* – each philosopher while waiting to acquire the right fork holds the left one
  - *no preemption* – each philosopher keeps the forks until she finishes eating
  - *circular wait* – if every philosopher can acquire the left fork, there is a chain in which every philosopher waits for a fork in possession of another philosopher

Deadlock prevention

deadlock ⇒ *mutual exclusion* **and**
hold and wait **and**
no preemption **and**
circular wait

- that is equivalent to

not *mutual exclusion* **or**
not *hold and wait* **or**
not *no preemtion* **or**
not *circular wait* ⇒ **not** *deadlock*

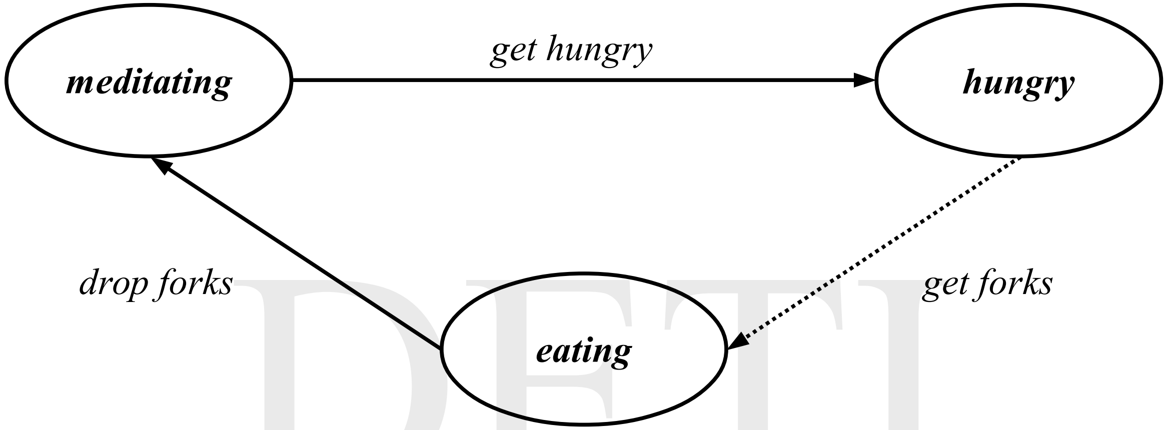
- So, if at least one of the necessary condition does not hold, there is no deadlock
- This is called *deadlock prevention*

Deadlock prevention

- Denying the *mutual exclusion* condition is only possible if the resources are shareable
  - Otherwise *race conditions* can occur
  - In the dining-philosopher problem, the forks are not shareable
- Thus, in general, only the other conditions are used to implement deadlock prevention

- Denying the *hold-and-wait* condition can be done if a process requests all required resources at once
  - In the dining-philosopher problem, the two forks must be acquired at once
  - In this solution, *starvation* can occur
    - *Aging* mechanisms are often used to solve starvation

Deadlock prevention

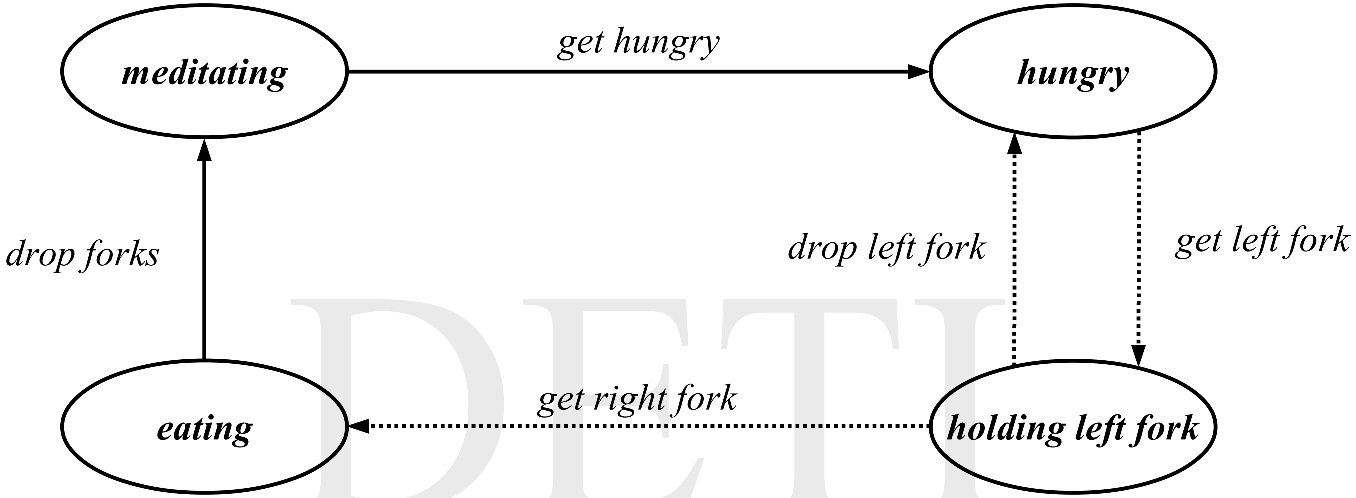


- This solution is equivalent to the one proposed by Dijkstra
- Every philosopher, when want to eat acquire the two forks at the same time
- If they are not available, she waits in the hungry state
- Starvation is not avoided

Deadlock prevention

- Denying the *no preemption* condition can be done if a process releases the already acquired resources if it fails acquiring the next one
  - Later on she can try the acquition again
  - In the dining-philosopher problem, a philosopher must release the left fork if she fails acquiring the right one
  - In this solution, *starvation* and *busy waiting* can occur
    - *Aging* mechanisms are often used to solve starvation
    - To avoid busy waiting, the process should block and be waked up when the resource is released

Deadlock prevention

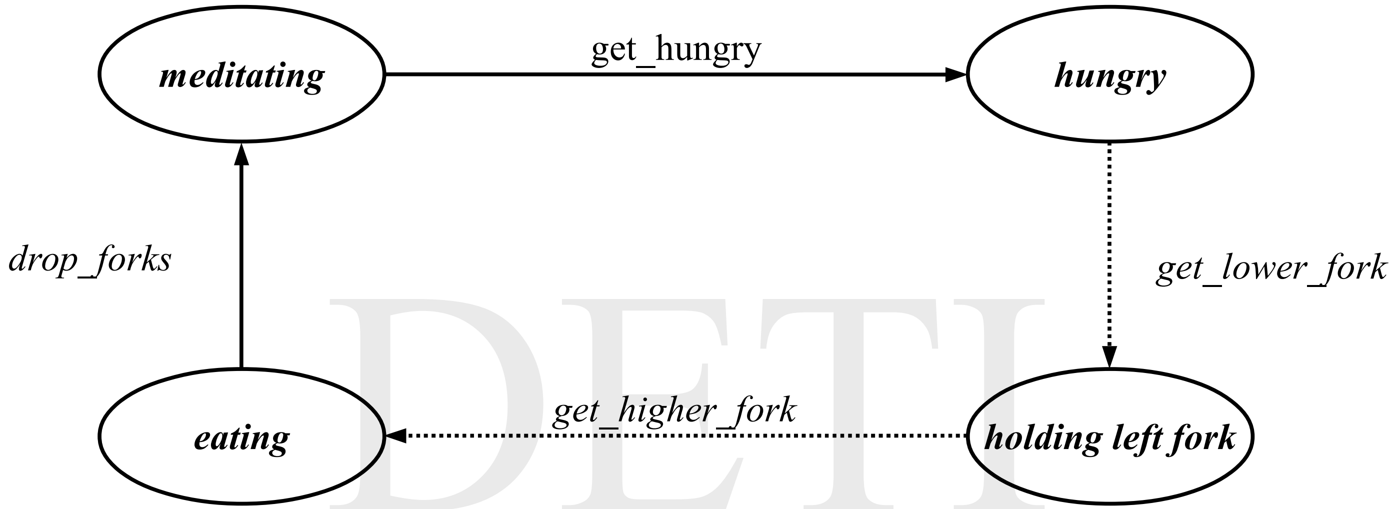


- When a philosopher gets hungry, she first acquire the left fork
- Then she tries to acquired the right one, releasing the left if she fails and returning to the hungry state
- *busy waiting* and starvation were not avoided in this solution

Deadlock prevention

- Denying the *circular wait* condition can be done asigning a different numeric id to every resource and imposing that the acquisition of resources have to be done either in ascending or descending order
  - This way the circular chain is always avoided
  - Starvation is not avoided
  - In the dining-philosopher problem, one of the philosophers acquire first the right fork and then the left one

Deadlock prevention



- Philosophers are numbered from 0 to N
- Every fork is assigned an id, equal to the id of the philisipher at its right. for instance
- Every philosopher, acquires first the fork with the lower id
- This way, philosophers 0 to N-2 acquire first the left fork, while philosopher N-1 acquires first the right one

Deadlock prevention

- Deadlock prevention policies are in general quite restrictive, not efficient and hard to apply in many situations
  - *denying mutual exclusion* – can only be applied to shareable resources
  - *denying hold and wait* – requires the a priori knowledge of all the necessary resources and always consider the worst case (all resources simultaneously)
  - *denying no preemption* – imposing the release and re-acquisition of resources, can introduce long delays in the processing the task
  - *denying circular wait* – can introduce a bad use of resources

Deadlock avoidance

- *Deadlock avoidanve* is less restrictive than deadlock prevention
  - None of the necessary conditions is denied
  - Instead, the system is monitored continuosly and a resource request is only granted if the system does not enter an unsafe state in consequence
- A state is said to be safe if there is a sequence of assignments of resources such that all intervening processes do terminate (no deadlock)
  - Otherwise it is assumed to be unsafe
- drawbacks:
  - the list of all resources must be known
  - the intervening processes have to declare at start their needs in terms of resources
- Note that usafe does not mean deadlock
  - worst conditions are conidered

Deadlock avoidance

- Let

  NTR_i – be the total no. of resources of type i (with $i = 0,1,\ldots,N\text{-}1$)

  $R_{i,j}$ – be the no. of resources of type i requered by process j

  (with i = 0,1,...,N-1, j = 0,1,...,M-1)

- The system can prevent a new process M to start if its termination can not be guaranteed
  - It is only launched if

$$NTR_i \geqslant R_{i,M} + \sum_{j=0}^{M-1} R_{i,j}$$

Deadlock avoidance

- Let

  NTR_i – be the total no. of resources of type i (with $i = 0,1,...,N\text{-}1$)

  $R_{i,j}$ – be the no. of resources of type i requered by process j

  (with i = 0,1,...,N-1, j = 0,1,...,M-1)

  $A_{i,j}$ – be the no. of resources of type i assigned to process j

  (with i = 0,1,...,N-1, j = 0,1,...,M-1)

- A new resource of type i is only assigned to a process if and only if there is a sequence $j' = f\,(i,j)$ such that

$$R_{i,j'} - A_{i,j'} < NTR_i - \sum_{k \geqslant j'}^{M-1} A_{i,k}$$

- This approach is called the ***banker's algotithm***

Banker's algorithm

		A	B	C	D
	total	6	5	7	6
	free	3	0	1	2
maximum	p1	3	3	2	2
	p2	1	2	3	4
	p3	1	3	5	0
granted	p1	1	2	2	1
	p2	1	0	3	3
	p3	1	2	1	0
needed	p1	2	1	0	1
	p2	0	2	0	1
	p3	0	0	4	0
new Grant	p1	0	0	0	0
	p2	0	0	0	0
	p3	0	1	0	0

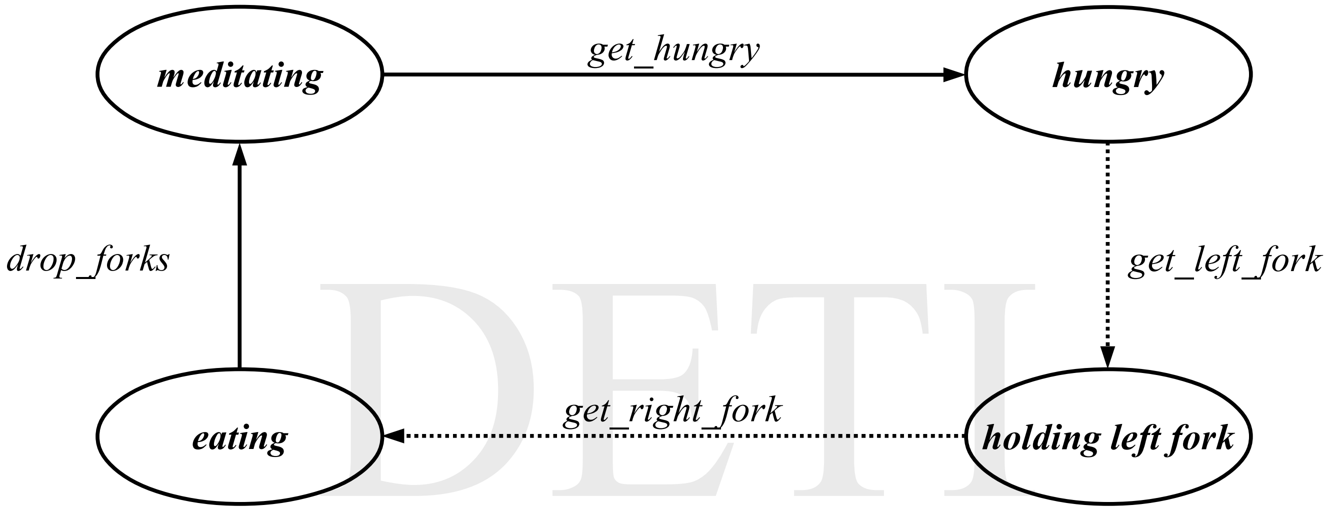
- If p3 requests 2 resources of type C, the request is denied
  - Because only 1 is available
- If p3 requests 1 resource of type B, the request is also denied
  - Why?

Banker's algorithm

		A	B	C	D
	total	6	5	7	6
	free	3	1	1	2
maximum	p1	3	3	2	2
	p2	1	2	3	4
	p3	1	3	5	0
granted	p1	1	2	2	1
	p2	1	0	3	3
	p3	1	2	1	0
needed	p1	2	1	0	1
	p2	0	2	0	1
	p3	0	1	4	0
new Grant	p1	0	0	0	0
	p2	0	0	0	0
	p3	0	0	0	0

- If p3 requests 2 resources of type C, the request is denied
  - Because only 1 is available
- If p3 requests 1 resource of type B, the request is also denied
  - Why?

Banker's algorithm



- Every philosopher first gets the left fork and then gets the right one
- However, in a specific situation the request of the left fork can be denied
    - What situation? Why?

Deadlock detection

- No deadlock-prevention or deadlock-avoidance is used
  - So, deadlock situations may occur
- In these cases
  - The state of the system should be examined to determine whether a deadlock has occurred
  - In such a case, a recover procedure from deadlock should exist and be applied

- In a more naive approach
  - The problem can simply be ignored

Deadlock detection

- If deadlock has occurred, the circular chain of processes and resources need to be broken
- This can be done:
  - *release resources from a process* – if it is possible
    - The process is suspended until the resource can be returned back
    - Efficient but requires the possibility of saving the process state
  - *rollback* – if the states of execution of the different processes is periodically saved
    - A resource is released from a process, whose state of execution is rolled back to the time the resource was assigned to it
  - *kill processes* –
    - Radical but easy to implement method

Bibliography

Operating Systems Concepts, Silberschatz, Galvin, Gagne, John Wiley & Sons, 9th Ed

- Chapter 3: *Processes* (section 3.4)
- Chapter 5: *Process synchronization* (sections 5.1 to 5.8)
- Chapter 7: *Deadlocks* (sections 7.1 to 7.6)

Modern Operating Systems, Tanenbaum, Prentice-Hall International Editions, 3rd Ed

- Chapter 2: *Processes and Threads* (sections 2.3 and 2.5)
- Chapter 6: *Deadlocks* (sections 6.1 to 6.6)

Operating Systems, Stallings, Prentice-Hall International Editions, 7th Ed

- Chapter 5: *Concurrency: mutual exclusion and synchronization* (sections 5.1 to 5.5)
- Chapter 6: *Concurrency: deadlock and starvation* (sections 6.1 to 6.7)