Sistemas de Operação / Fundamentos de Sistemas Operativos

Interprocess communication

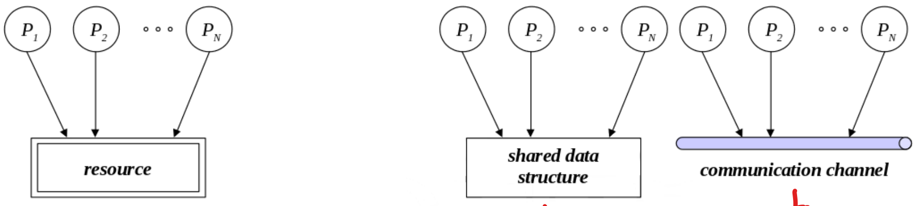
Artur Pereira <artur@ua.pt>

DETI / Universidade de Aveiro

Outline

Concepts
Independent and collaborative processes

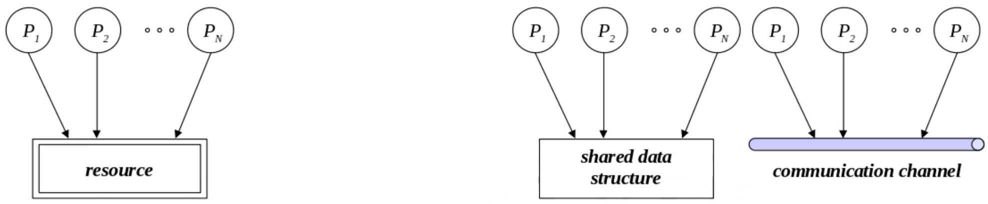
- In a multiprogrammed environment, two or more processes can be:
  - independent – if they, from their creation to their termination, never explicitly interact
    - actually. there is an implicit interaction, as they compete for system resources
    - ex: jobs in a batch system; processes from different 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



Data

Messages } Only applies to Processes, not Threads

Concepts
Independent and collaborative processes (2)

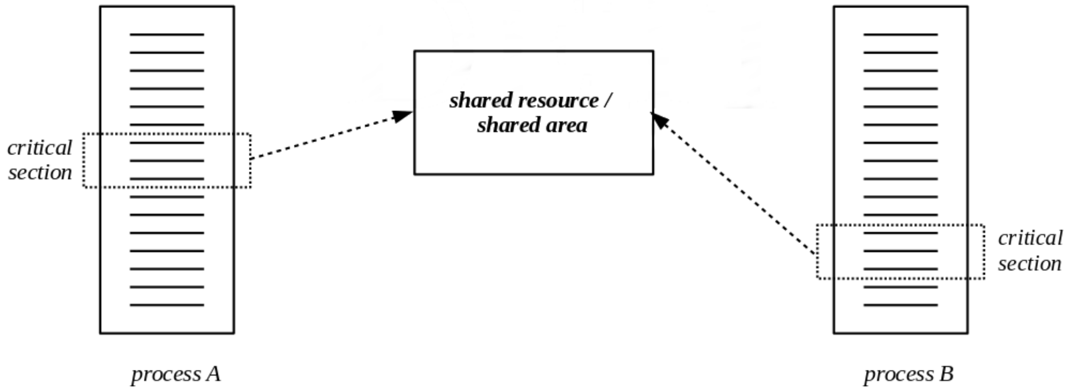


- Independent processs competing for a resource
- It is the responsibility of the OS to ensure the assignment of resources to processes is done in a controlled way, such that no information lost occurs
- In general, this imposes that only one process can use the resource at a time – mutual exclusive access

- Cooperative processes sharing information or communicating
- It is the responsibility of the processes to ensure 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 access the shared area at a time – mutual exclusive access
- The communication channel is typically a system resource, so processes compete for it

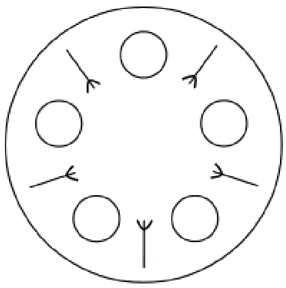
Concepts
Critical section

- Having access to a resource or to a shared area actually means executing the code that does the access
- This section of code, if not properly protected, can result in race conditions
  - which can result in lost of information
  - It is called critical section
- Critical sections should execute in mutual exclusion



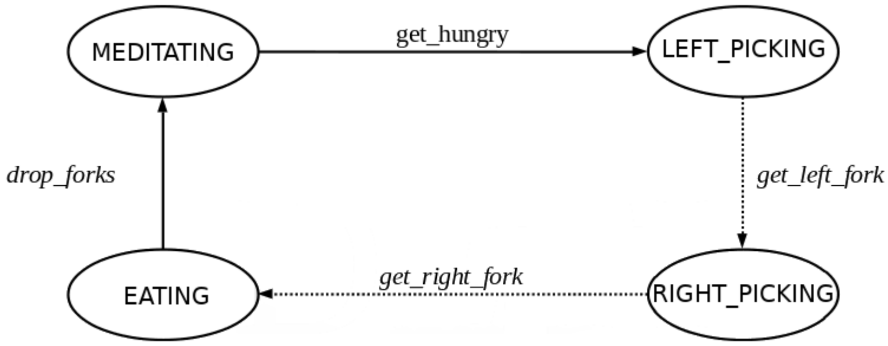
→ Until here, it's already approached subjects

Philosopher dinner
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/his left and right sides
  - Every philosopher alternates periods in which she/he medidates with periods in which she/he eats
- Modeling every philosopher as a different process or thread and the forks as resources, design a solution for the problem

Philosopher dinner
A solution – state diagram



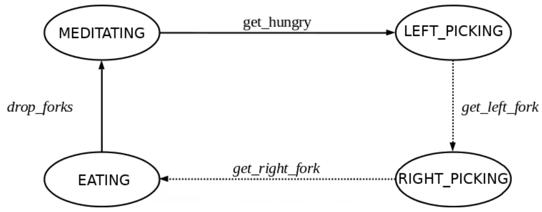
- This is a possible solution for the dining-philosopher problem
  - when a philosopher gets hungry, he/she first gets the left fork and then holds it while waits for the right one
- Let's look at an implementations of this solution!

Philosopher dinner
A solution – code

```
enum PHILO_STATE { MEDITATING, LEFT_PICKING, RIGHT_PICKING, EATING };

typedef struct TablePlace
{
        int state;
} TablePlace;

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

int set_table(unsigned int n);
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 execute the code

Philosopher dinner
A solution – a race condition



- This solution may work some times, but in general suffers from race conditions → Happens, because someone on the right is already using the fork ...
- Let's look at a code snippet:
  - get_right_fork:

```
while (table->place[right(f)].state == EATING or
       table->place[right(f)].state == RIGHT_PICKING);
```


Concepts
Deadlock and starvation

- Mutual exclusion in the access to a resource or shared area can result in
  - deadlock – when two or more processes are waiting forever to 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 primitives

Access to a resource or to a shared area

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

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

critical section

```
/* shared data structure */
shared DATA d;
/* processes sharing data - p = 0, 1, ..., N-1 */
void mainLoop (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

Access primitives

Producer-consumer example - producer

```
/* communicating data structure: FIFO of fixed size */
shared FIFO fifo;
/* producer processes - p = 0, 1, ..., N-1 */
void producer(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

Access primitives
Producer-consumer example - consumer

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

shared FIFO fifo;

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

    forever
    {
        done = false;
        do
        {
            enter_critical_section(p);
            if (fifo.notEmpty())
            {
                fifo.retrieve(&val);        <-- critical section
                done = true;
            }
            leave_critical_section(p);
        } while (!done);
        consume_data(val);
        do_something_else();
    }
}
```


Access primitives
Requirements

- 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 its critical section cannot prevent another process from entering its own critical section
  - No starvation – a process requiring access to its critical section should not have to wait indefinitely
  - Length of stay inside a critical section should be necessarily finite

Access primitives
Types of solutions

- In general, a memory location is used to control access to the critical section
  - it works as a binary flag
- Two types of solutions: software solutions and hardware solutions

- software solutions – solutions that are based on the typical instructions used to access memory location
  - read and write are done by different instructions
  - interruption can occur between read and write
- 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 (uninterruptible) way

Software solutions
Constructing a solution - 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;
}
```

- 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, until its predecessor enters and give it access on leaving

Software solutions
Constructing a solution - 1st step

```
/* 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;
}
```

- Not a valid solution
  - Mutual exclusion is not guaranteed

Software solutions
Constructing a solution - 1st step

```
/* 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;
}
```

- Assume the following sequence of execution:
  - P_0 enters `enter_critical_section` and tests `is_in[1]` as being `false`
  - P_1 enters `enter_critical_section` and tests `is_in[0]` as being `false`
  - P_1 changes `is_in[1]` to `true` and enters its critical section
  - P_0 changes `is_in[0]` to `true` and enters its critical section
- Thus, both processes enter their critical sections
- It seems that the failure is a result of testing first the other's control variable and then change its own variable

Software solutions
Constructing a solution - 2nd step

```
/* 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;
}
```

- Not a valid solution
  - Mutual exclusion is guaranteed, but deadlock can occur

→ IF we have more than one PID, they could be all waiting on each other

Software solutions
Constructing a solution - 2nd step

```
/* 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;
}
```

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

Software solutions

Constructing a solution - 3rd step

```
/* 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;
}
```

→ Using a random element allows our programs to be more unpredictable and non-deterministic, but it also introduces "wasted" processing

- An almost valid solution
  - The Ethernet protocol uses a similar approach to control access to the communication medium

What is a delay? It's just a counter. This approach will increase throughput

Software solutions

Constructing a solution - 3rd step

```
/* 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;
}
```

- 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

Software solutions
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;
}
```


Software solutions
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;
}
```

- The algorithm uses an alternation mechanism (on the priority) 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

Software solutions
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;
}
```

- Works, but can suffer from starvation

Software solutions
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 = other_pid;
    while ((want_enter[other_pid]) && (last == other_pid));
}
void leave_critical_section(uint own_pid)
{
    want_enter[own_pid] = false;
}
```

- The Peterson algorithm uses the order of arrival to solve conflicts
  - Each process has to write the other's ID in a shared variable (last)
  - The subsequent reading allows to determine which was the last one

Software solutions
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 = other_pid;
    while ((want_enter[other_pid]) && (last == other_pid));
}
void leave_critical_section(uint own_pid)
{
    want_enter[own_pid] = false;
}
```

- The Peterson algorithm uses the order of arrival to solve conflicts
  - Each process has to write the other's 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 starvation
  - Makes no assumption about the relative speed of intervening processes

It's a sequential Algorythm

Software solutions
Generalized Peterson algorithm (1981)

```
#define  R   ...    /* process id = 0, 1, ..., R-1 */
shared int level[R] = {-1, -1, ... , -1};
shared int last[R-1];
void enter_critical_section(uint own_pid)
{
   for (uint i = 0; i < R-1; i++)
   {
      level[own_pid] = i;
      last[i] = own_pid;
      do
      {
         test = false;
         for (uint j = 0; j < R; j++)
            if (j != own_pid)
               test = test || (level[j] >= i);
      } while (test && (last[i] == own_pid));
   }
}
void leave_critical_section(int own_pid)
{
   level[own_pid] = -1;
}
```

- Can be generalized to more than two processes
  - The general solution is similar to a waiting queue → Makes sense because it uses the order of arraival

Hardware solutions
disabling interrupts

- *Uniprocessor computational system*
  - The switching of processes, in a multiprogrammed environment, is always caused by an external device: → Access to I/O, because it doesn't have other processors to switch to
    - real time clock (RTC) – cause the time-out transition in preemptive systems
    - device controller – can cause the preempt transitions in case of waking 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
- *Multiprocessor computational system*
  - Disabling interrupts in one processor has no effect

Hardware solutions
special instructions – TAS

```
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 this behavior
- Surprisingly, it is often called TAS (test and set)

Hardware solutions
special instructions – CAS

```
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;
}
```

→ keep wasting resources waiting for the flag to change

- 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 this returns a bool

Hardware solutions
Busy waiting

- The previous solutions suffer from *busy waiting*
  - The lock primitive is in the active state (using the CPU) while waiting
  - It is often referred to as a spinlock, 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 is 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 systems 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 critical section

Hardware solutions
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;
}
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 operation* (can not be interrupted)

or signal all if we have a Broadcast

- Atomic operations are still required

Semaphores
Definition

- A semaphore is a synchronization mechanism, defined by a data type plus two atomic operations, down and up
- 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 waiting process (accordingly to a given policy)
    - increment `val` otherwise
- Note that `val` can only be manipulated through these operations
  - It is not possible to check the value of `val`

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

- Internally, the block_on_sem function must enable interruptions
- This implementation is typical of uniprocessor systems. Why?

- Semaphores can be binary or not binary
- How to implement mutual exclusion using semaphores?
  - Using a binary semaphore

Semaphores
Bounded-buffer problem – solving using semaphores

```
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
        {
            if (fifo.notFull())
            {
                fifo.insert(data);
                done = true;
            }
        } while (!done);
        do_something_else();
    }
}
```

```
/* consumers - c = 0, 1, ..., M-1 */
void consumer(unsigned int c)
{
    DATA data;
    forever
    {
        bool done = false;
        do
        {
            if (fifo.notEmpty())
            {
                fifo.retrieve(&data);
                done = true;
            }
        } while (!done);
        consume_data(data);
        do_something_else();
    }
}
```

- This solution can suffer race conditions

Semaphores
Bounded-buffer problem – solving using semaphores

```
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
        {
            if (fifo.notFull())
            {
                lock(p);
                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
        {
            if (fifo.notEmpty())
            {
                lock(c);
                fifo.retrieve(&data);
                done = true;
                unlock(c);
            }
        } while (!done);
        consume_data(data);
        do_something_else();
    }
}
```

- Mutual exclusion is guaranteed, but suffers from busy waiting

Semaphores
Bounded-buffer problem – solving using semaphores

```
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
        {
            if (fifo.notFull())
            {
                fifo.insert(data);
                done = true;
            }
        } while (!done);
        do_something_else();
    }
}
```

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

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

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

Semaphores
Bounded-buffer problem – solving 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);   → 1
        sem_down(access);   → 1
        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);   → 0
        sem_down(access);
        fifo.retrieve(&val);
        sem_up(access);
        sem_up(nslots);
        consume_data(val);
        do_something_else();
    }
}
```

- `fifo.notEmpty()` and `fifo.notFull()` are no longer necessary. Why?
- What are the initial values of the semaphores?

Semaphores
Bounded-buffer problem – wrong solution

```
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);
don't   { sem_down(access);   → 1
switch  { sem_down(nslots);   → 0
        fifo.insert(val);
        sem_up(access);          gains access
        sem_up(nitems);          but can't
        do_something_else();     produce
    }
}
```

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

    forever
    {
        sem_down(nitems);   → can consume
        sem_down(access);   → can't access
        fifo.retrieve(&val);
        sem_up(access);
        sem_up(nslots);
        consume_data(val);
        do_something_else();
    }
}
```

- One can easily make a mistake
- What is wrong with this solution? It can cause deadlock

Semaphores
Analysis of semaphores

- Concurrent 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 concurrent programming principles, as race conditions or deadlock can be easily introduced
    - See the previous example, as an illustration of this

Monitors
Introduction

- A problem with semaphores is that they are used both to implement mutual exclusion and for synchronization between 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 sections and then block if continuation conditions are not satisfied
- A solution is to introduce a (concurrent) construction at the programming language level that deals with mutual exclusion and synchronization separately

- 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

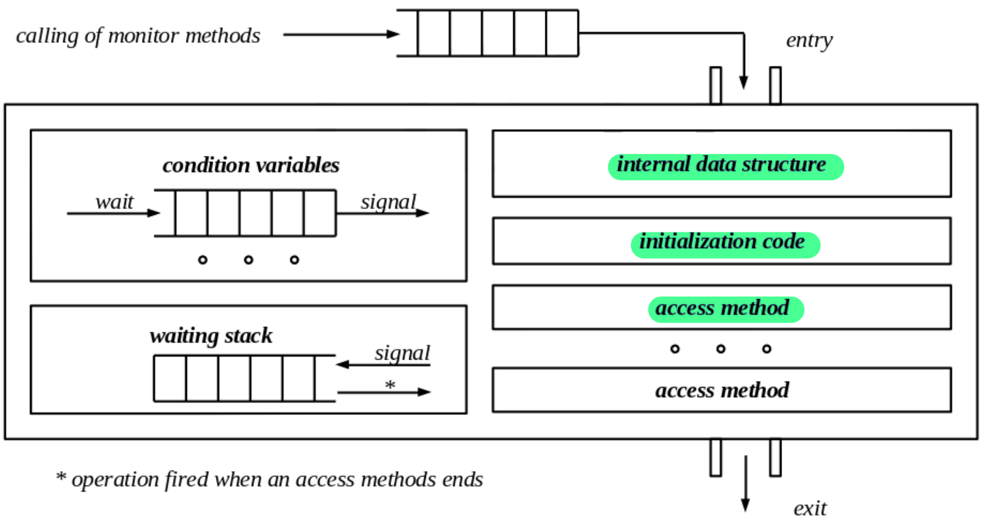
Monitors
Definition

```
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?* → We don't know, one that is listening

Monitors
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

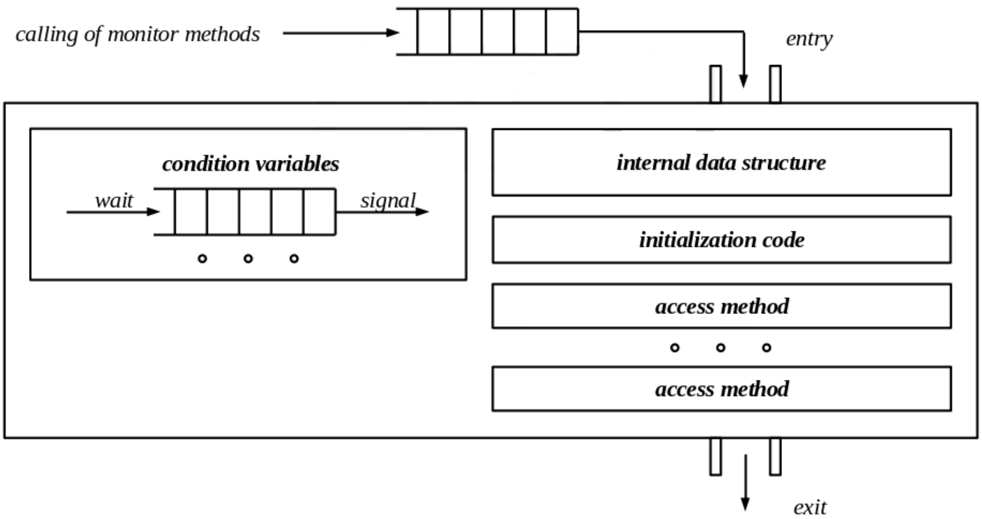


* operation fired when an access methods ends

Monitors

Brinch Hansen monitor

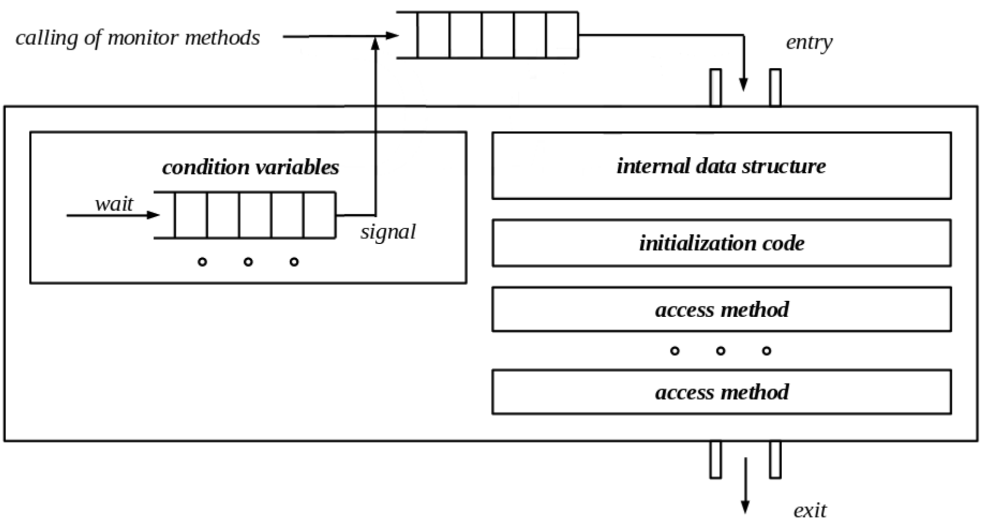
- What to do when signal occurs?
- Brinch Hansen monitor – the thread calling signal immediately leaves the monitor (signal is the last instruction of the monitor method)
  - easy to implement, but quite restrictive (only one signal allowed in a method)



Monitors

Lampson / Redell monitor

- What to do when signal occurs?
- 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



Monitors
Bounded-buffer problem – solving 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); = 1
        if/while (fifo.isFull())
        {
            wait(nslots, access);
        }
        fifo.insert(data);
        signal(nitems);
        unlock(access);
        do_something_else();
    }
}
```

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

- What is the initial state of the mutex? → The Producer should be launched first

Message-passing
Introduction

- Processes can communicate exchanging messages
  - A general communication mechanism, not requiring explicit shared memory, that includes both communication and synchronization
  - Valid for uniprocessor and multiprocessor systems
- Two operations are required:
  - send and receive
- A communication link is required
  - That can be categorized in different ways:
    - Direct or indirect communication
    - Synchronous or asynchronous communication
    - Type of buffering

Message-passing
Direct and indirect communication

- Symmetric direct communication
  - A process that wants to communicate must explicitly name the receiver or sender
    - `send(P, msg)` – send message `msg` to process `P`
    - `receive(P, msg)` – receive message `msg` 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 communicating processes there exist exactly one link

- Asymetric direct communication
  - Only the sender must explicitly name the receiver
    - `send(P, msg)` – send message `msg` to process `P`
    - `receive(id, msg)` – receive message `msg` from any process

Message-passing
Direct and indirect communication

- Indirect communication
  - The messages are sent to and received from mailboxes, or ports
    - `send(M, msg)` – send message `msg` to mailbox `M`
    - `receive(M, msg)` – receive message `msg` 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 more 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

- From a synchronization point of view, 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

Message-passing
Bounded-buffer problem – solving using messages

```
shared MailBox mbox;
```

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

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

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

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

- There is no need to deal with mutual exclusion and synchronization explicitly
  - the send and receive primitives take care of it

Unix IPC primitives
POSIX support for monitor implementation

- 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

Unix IPC primitives
Semaphores

- System V semaphores
  - creation: `semget`
  - down and up: `semop`
  - other operations: `semctl`
- 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

Unix IPC primitives
Message-passing

- 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 → Buffering with Bounded Capacity. Finite length buffe
  - The receive operation has an argument to specify the type of message to receive: a given type, any type or a range of types
    - 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`, · · ·

Unix IPC primitives
Shared memory

- Address spaces of processes 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 → So Processes have to compete for it
- System V shared memory
  - creation – `shmget`
  - mapping and unmapping – `shmat`, `shmdt`
  - other operations – `shmctl`
- POSIX shared memory
  - creation - `shm_open`, `ftruncate`
  - mapping and unmapping - `mmap`, `munmap`
  - other operations - `close`, `shm_unlink`, `fchmod`, $\cdots$

Bibliography

- Operating Systems: Internals and Design Principles, W. Stallings, Prentice-Hall International Editions, 7th Ed, 2012
  - Chapter 5: Concurrency: mutual exclusion and synchronization (sections 5.1 to 5.5)
- Operating Systems Concepts, A. Silberschatz, P. Galvin and G. Gagne, John Wiley & Sons, 9th Ed, 2013
  - Chapter 3: Processes (section 3.4)
  - Chapter 4: Process synchronization (sections 5.1 to 5.8)
- Modern Operating Systems, A. Tanenbaum and H. Bos, Pearson Education Limited, 4th Ed, 2015
  - Chapter 2: Processes and Threads (section 2.3)