Sistemas de Operação / Fundamentos de Sistemas Operativos

Semaphores and shared memory

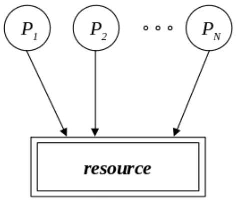
Artur Pereira <artur@ua.pt>

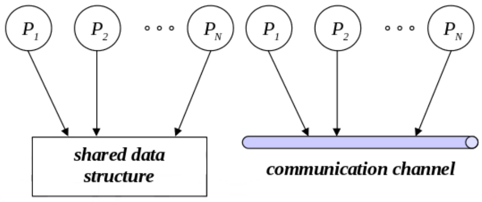
DETI / Universidade de Aveiro

Outline

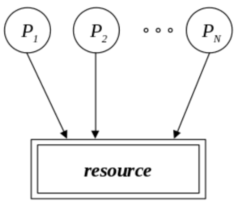
Concepts
Independent and collaborative processes

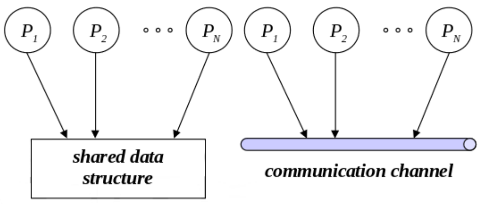
- In a multiprogrammed environment, two or more processes can be:
  - independent – if they, from their creation to their termination, never explicitly interact
    - actually. there is an implicit interaction, as they compete for system resources
    - ex: jobs in a batch system; processes from different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





Concepts
Independent and collaborative processes (2)



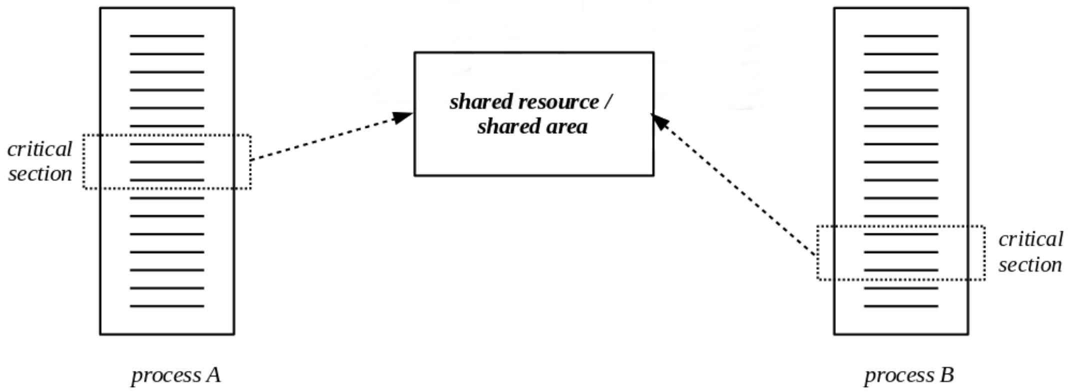


- Independent processs competing for a resource
- It is the responsibility of the OS to ensure the assignment of resources to processes is done in a controlled way, such that no information lost occurs
- In general, this imposes that only one process can use the resource at a time – mutual exclusive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

- 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, called critical section , if not properly protected, can result in race conditions
- A race condition is a condition where the behaviour (output, result) depends on the sequence or timing of other (uncontrollable) events
  - Can result in undesirable behaviour
- Critical sections should execute in mutual exclusion



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 → Usually occurs when Priority is envolved
    - operations are continuously postponed

Semaphores
Definition

- A semaphore is a synchronization mechanism, defined by a data type plus two atomic operations, down and up
  ↳ can't be altered or divided
- 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
    - increment `val`
    - if `queue` is not empty, wake up one waiting process (accordingly to a given policy)
- 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

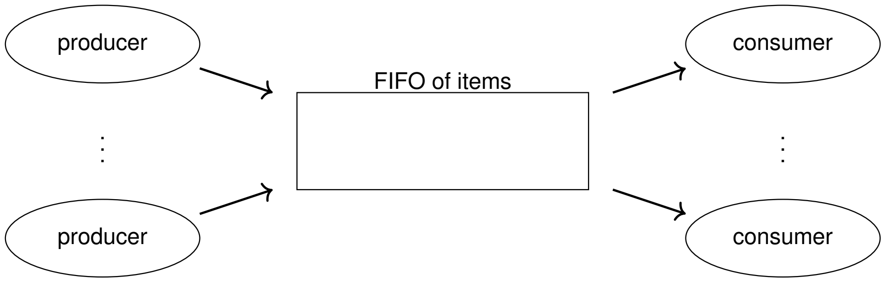
```
/* array of semaphores defined in kernel */
#define  R   ... /* semid = 0, 1, ..., R-1 */
static SEMAPHORE sem[R];
void sem_down(unsigned int semid)
{
    disable_interruptions;
    if (sem[semid].val == 0)
        block_on_sem(getpid(), semid);
    sem[semid].val -= 1;
    enable_interruptions;
}

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;
}
```

- 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 – problem statement



- In this problem, a number of entities (producers) produce information that is consumed by a number of different entities (consumers)
- Communication is carried out through a buffer with bounded capacity

- Assume that every producer and every consumer run in a different process
  - Hence the FIFO must be implemented in shared memory so the different processes can access it
- How to guarantee that race conditions doesn't arise?
  - Enforcing mutual exclusion in the access to the FIFO

Semaphores
Bounded-buffer problem – implementation

```
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();
    }
}
```

• When one Process is accessing the value of a variable, other one may be altering it

- This solution can suffer race conditions
  - How to avoid it?

Semaphores

Bounded-buffer problem – implementation

```
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();
    }
}
```

- Introducing mutual exclusion
  - Mutual exclusion is guaranteed, but suffers from busy waiting →

when The Process checks for the condition it's always False, only when He checks

Semaphores

Bounded-buffer problem – implementation

```
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 a safe solution 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);
        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);
        sem_down(access);
        sem_down(nslots);   -> We gain access but there are no more slots
        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);   -> can't consume, cause The Producer has access
        sem_down(access);
        fifo.retrieve(&val);
        sem_up(access);
        sem_up(nslots);
        consume_data(val);
        do_something_else();
    }
}
```

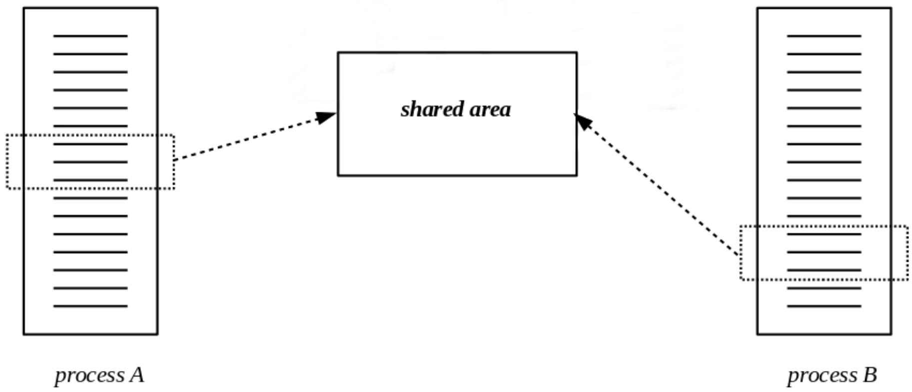
- One can easily make a mistake
  - What is wrong with this solution? It can cause deadlock

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

Shared memory
A resource

- Address spaces of processes are independent
- But address spaces are virtual
- The same physical region can be mapped into two or more virtual regions
- Shared memory is managed as a resource by the operating system
- Two actions are required:
  - Requesting a segment of shared memory to the OS
  - Maping that segment in the process' address space



Unix IPC primitives
Semaphores

- System V semaphores → Non - binary
  - creation: `semget`
  - down and up: `semop`
  - other operations: `semctl`
  - execute `man semget`, `man semop` or `man semctl` for a description
- POSIX semaphores → Binary
  - 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`
  - down and up
    - `sem_wait`, `sem_trywait`, `sem_timedwait`, `sem_post`
  - execute `man sem_overview` for an overview

Unix IPC primitives
Shared memory

- System V shared memory
  - creation – `shmget`
  - mapping and unmapping – `shmat`, `shmdt`
  - other operations – `shmctl`
  - execute `man shmget`, `man shmat` `man shmdt` or `man shmctl` for a description
- POSIX shared memory
  - creation - `shm_open`, `ftruncate`
  - mapping and unmapping - `mmap`, `munmap`
  - other operations - `close`, `shm_unlink`, `fchmod`, ···
  - execute `man shm_overview` for an overview