Sistemas de Operação

Semaphores and shared memory

Artur Pereira <artur@ua.pt>

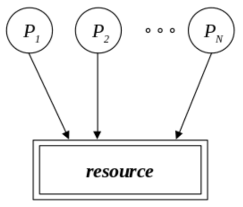
DETI / Universidade de Aveiro

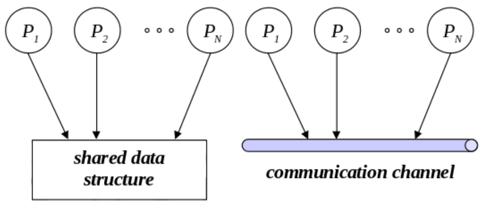
Outline

1. Key concepts
2. Shared memory
3. Unix IPC primitives for shared memory
4. Bounded buffer problem, using shared memory
5. Semaphores
6. Unix IPC primitives for semaphores
7. Bounded buffer problem, making the implementation safe
8. Client-server application, using shared memory and semaphores

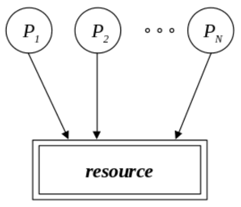
Key concepts
Independent and interacting processes

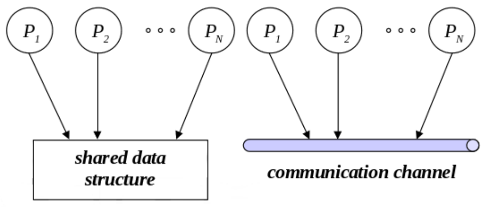
- In a multiprogrammed environment, two or more processes can be:
  - independent – if they, from their creation to their termination, never explicitly interact
    - actually. there is an implicit interaction, as they compete for system resources
    - ex: jobs in a batch system; processes from different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





Key concepts
Independent and interacting processes (2)



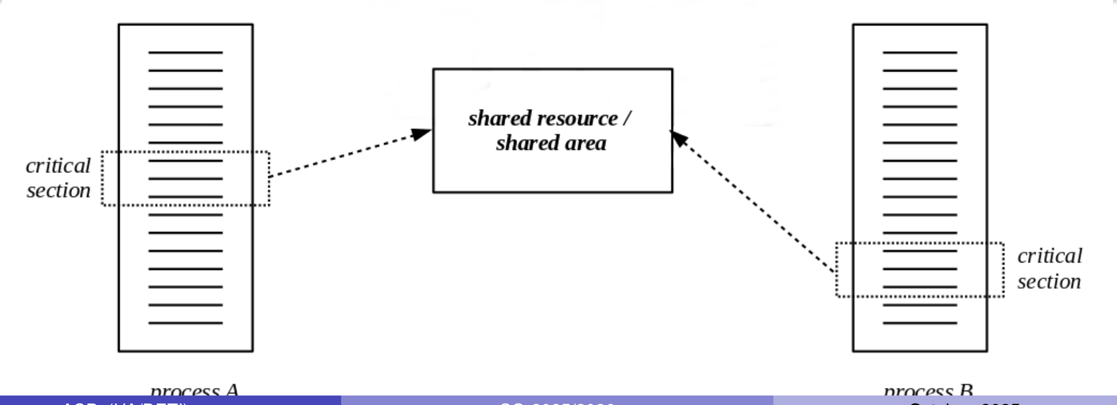


- Independent processes competing for a resource
- It is the responsibility of the OS to ensure the assignment of resources to processes is done in a controlled way, such that no information lost occurs
- In general, this imposes that only one process can use the resource at a time – mutual exclusive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

- Interacting 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

Key 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, and can lead to undesirable behaviour
- Critical sections should execute in mutual exclusion

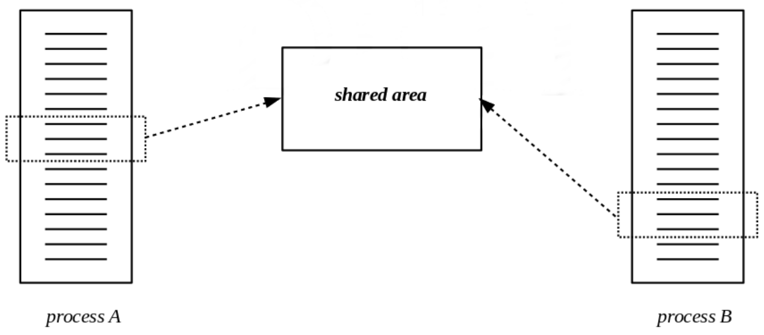


Key concepts
Deadlock and starvation

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

Shared memory
Shared memory as a resource

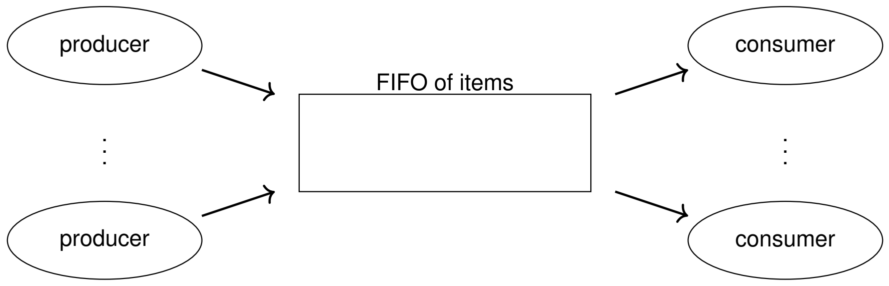
- Address spaces of processes are independent
- But address spaces are virtual
- Two or more virtual regions can be mapped to the same physical region (the physical region is shared)
- Shared memory is managed as a resource by the operating system
- Two actions are required:
  - Requesting a segment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



Unix IPC primitives
Shared memory

- System V shared memory
  - creation – `shmget`
  - mapping and unmapping – `shmat`, `shmdt`
  - other operations – `shmctl`
  - execute `man 7 sysvipc` for an overview description
  - execute `man shmget`, `man shmat` `man shmdt` or `man shmctl` for specific descriptions
- 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 description
  - execute `man shm_open`, `man mmap`, `man munmap`, `man shm_close`,... for specific descriptions

Illustration example
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, shared by all intervening entities

- Assume that every producer and every consumer run as a different process
  - Hence the FIFO must be implemented in shared memory so the different processes can access it
- Let look at an implementation...

Illustration example
Bounded-buffer problem – fifo definition

```
#define N 100

struct FIFO
{
    bool full;
    uint32_t size; // effective size of the FIFO
    uint32_t in, out;
    uint32_t dummyDelay; // added to enhance the probability of occurrence of race conditions
    struct item
    {
        uint32_t id; // id of the producer
        uint32_t v1; // a general purpose value
        uint32_t v2; // another general purpose value
    } data[N];
};

void fifoInit(FIFO *f, uint32_t sz, uint32_t delay);

bool fifoIsFull(FIFO *f);

bool fifoIsEmpty(FIFO *f);

void fifoInsert(FIFO *f, uint32_t id, uint32_t v1, uint32_t v2);

void fifoRetrieve(FIFO *f, uint32_t *idp, uint32_t *v1p, uint32_t *v2p);

void fifoDestroy(FIFO *f);
```

Illustration example
Bounded-buffer problem – unsafe implementation of the fifo

```
void fifoInsert(FIFO *f, uint32_t id, uint32_t v1, uint32_t v2)
{
    /* wait until fifo is not full */
    while (fifoIsFull(f))
    {
        bwDelay(10); // wait for a while
    }

    /* make insertion */
    f->data[f->in].id = id;
    f->data[f->in].v1 = v1;
    bwDelay(f->dummyDelay); // to enhance the probability of occurrence of race conditions
    f->data[f->in].v2 = v2;
    f->in = (f->in + 1) % f->size;
    f->full = (f->in == f->out);
}

void fifoRetrieve(FIFO *f, uint32_t *idp, uint32_t *v1p, uint32_t *v2p)
{
    /* wait until fifo is not empty */
    while (fifoIsEmpty(f))
    {
        bwDelay(10); // wait for a while
    }

    /* make retrieval */
    *idp = f->data[f->out].id;
    *v1p = f->data[f->out].v1;
    bwDelay(f->dummyDelay); // to enhance the probability of occurrence of race conditions
    *v2p = f->data[f->out].v2;
    f->out = (f->out + 1) % f->size;
    f->full = false;
}
```

- How to guarantee absence of race conditions?
  - enforcing mutual exclusion

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 and queue 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 in general to check the value of `val`

Semaphores
A possible 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

Unix IPC primitives
Semaphores

- System V semaphores
  - creation: `semget` (actually creates an array of semaphores)
  - down and up: `semop`
  - other operations: `semctl`
  - execute `man semget`, `man semop` or `man semctl` for a description
- POSIX semaphores
  - 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 description
  - execute `man sem_open`, `man sem_wait`, ..., for a specific description

Illustration example
Bounded-buffer problem – making the implementation safe

```
void fifoInsert(FIFO *f, uint32_t id, uint32_t v1, uint32_t v2)
{
    /* wait until fifo is not full */
    while (fifoIsFull(f))
    {
        bwDelay(10); // wait for a while
    }

    /* make insertion */
    f->data[f->in].id = id;
    f->data[f->in].v1 = v1;
    bwDelay(f->dummyDelay); // to enhance the probability of occurrence of race conditions
    f->data[f->in].v2 = v2;
    f->in = (f->in + 1) % f->size;
    f->full = (f->in == f->out);
}

void fifoRetrieve(FIFO *f, uint32_t *idp, uint32_t *v1p, uint32_t *v2p)
{
    /* wait until fifo is not empty */
    while (fifoIsEmpty(f))
    {
        bwDelay(10); // wait for a while
    }

    /* make retrieval */
    *idp = f->data[f->out].id;
    *v1p = f->data[f->out].v1;
    bwDelay(f->dummyDelay); // to enhance the probability of occurrence of race conditions
    *v2p = f->data[f->out].v2;
    f->out = (f->out + 1) % f->size;
    f->full = false;
}
```

Illustration example
Bounded-buffer problem – making the implementation safe

```
void fifoInsert(FIFO *f, uint32_t id, uint32_t v1, uint32_t v2)
{
    /* wait until fifo is not full */

    reserve a slot, waiting if necessary

    lock
    /* make insertion */
    f->data[f->in].id = id;
    f->data[f->in].v1 = v1;
    bwDelay(f->dummyDelay); // to enhance the probability of occurrence of race conditions
    f->data[f->in].v2 = v2;
    f->in = (f->in + 1) % f->size;
    f->full = (f->in == f->out);
}   unlock
    make an item available
void fifoRetrieve(FIFO *f, uint32_t *idp, uint32_t *v1p, uint32_t *v2p)
{
    /* wait until fifo is not empty */

    reserve an item, waiting if necessary

    lock
    /* make retrieval */
    *idp = f->data[f->out].id;
    *v1p = f->data[f->out].v1;
    bwDelay(f->dummyDelay); // to enhance the probability of occurrence of race conditions
    *v2p = f->data[f->out].v2;
    f->out = (f->out + 1) % f->size;
    f->full = false;
}   unlock
    make a slot available
```

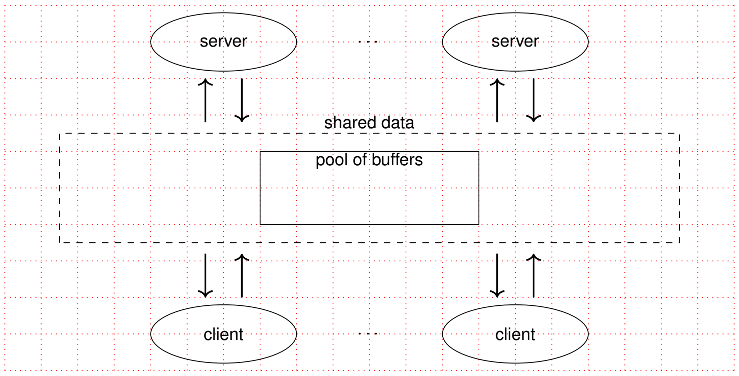
Illustration example
Bounded-buffer problem – fifo definition

```
#define N 100

#define ACCESS 0
#define SLOTS 1
#define ITEMS 2

struct FIFO
{
    bool full;
    uint32_t size; // effective size of the FIFO
    uint32_t in, out;
    uint32_t dummyDelay; // added to enhance the probability of occurrence of race conditions
    struct item
    {
        uint32_t id; // id of the producer
        uint32_t v1; // a general purpose value
        uint32_t v2; // another general purpose value
    } data[N];

    /* support for safeness, not used in the unsafe implementation */
    int sem;
};

void fifoInit(FIFO *f, uint32_t sz, uint32_t delay);

bool fifoIsFull(FIFO *f);

bool fifoIsEmpty(FIFO *f);

void fifoInsert(FIFO *f, uint32_t id, uint32_t v1, uint32_t v2);

void fifoRetrieve(FIFO *f, uint32_t *idp, uint32_t *v1p, uint32_t *v2p);

void fifoDestroy(FIFO *f);
```

Illustration example
Bounded-buffer problem – safe implementation using semaphores

```
void fifoInsert(FIFO *f, uint32_t id, uint32_t v1, uint32_t v2)
{
    /* reserve a slot, waiting if necessary */
    psem_down(f->sem, SLOTS);

    /* lock access to fifo */
    psem_down(f->sem, ACCESS);

    /* make insertion */
    f->data[f->in].id = id;
    f->data[f->in].v1 = v1;
    bwDelay(f->dummyDelay); // to enhance the probability of occurrence of race conditions
    f->data[f->in].v2 = v2;
    f->in = (f->in + 1) % f->size;
    f->full = (f->in == f->out);

    /* release access to fifo */
    psem_up(f->sem, ACCESS);

    /* notify there is one more item available */
    psem_up(f->sem, ITEMS);
}

void fifoRetrieve(FIFO *f, uint32_t *idp, uint32_t *v1p, uint32_t *v2p)
{
    /* reserve an item, waiting if necessary */
    psem_down(f->sem, ITEMS);

    /* lock access to fifo */
    psem_down(f->sem, ACCESS);

    /* make retrieval */
    *idp = f->data[f->out].id;
    *v1p = f->data[f->out].v1;
    bwDelay(f->dummyDelay); // to enhance the probability of occurrence of race conditions
    *v2p = f->data[f->out].v2;
    f->out = (f->out + 1) % f->size;
    f->full = false;

    /* release access to fifo */
    psem_up(f->sem, ACCESS);

    /* notify there is one more slot available */
    psem_up(f->sem, SLOTS);
}
```

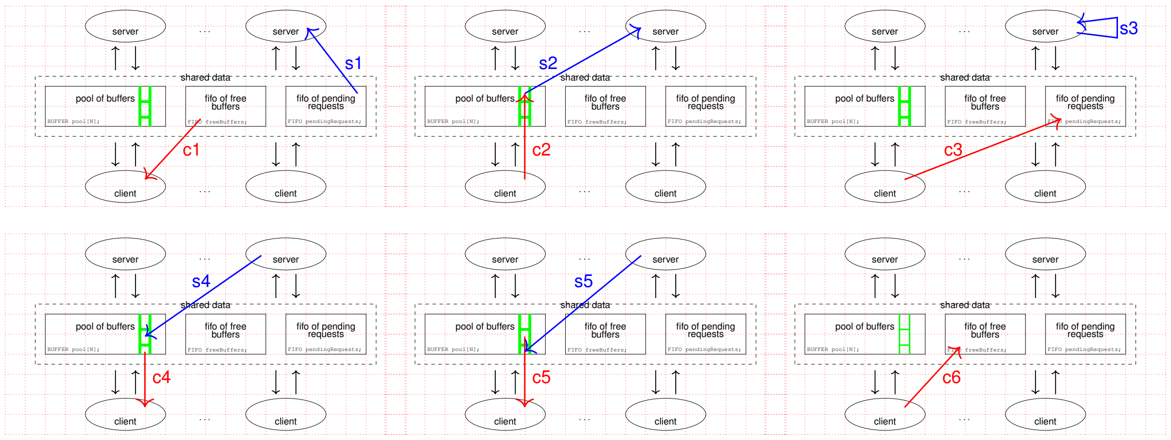
Illustration example
Client-server example – problem statement



- In this problem, a number of entities (clients) interact with a number of other entities (servers) to request a service
- Communication is carried out through a pool of buffers shared by all

- Every producer and consumer must run as a different process
  - Hence the data structure must be implemented in shared memory so the different processes can access it

Illustration example
Client-server example – interaction cycles



client interaction cycle	**server interaction cycle**
c1: tk = getFreeBuffer()	**s1:** tk = getPendingRequest()
c2: putRequestData(data, tk)	**s2:** req = getRequestData(tk)
c3: submitRequest(tk)	**s3:** resp = produceResponse(req)
c4: waitForResponse(tk)	**s4:** putResponse(resp, tk)
c5: resp = getResponseData(tk)	**s5:** notifyClient(tk)
c6: releaseBuffer(tk)	