```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define MAXPAROLA 30
#define MAXRIGA 80

int main(int argc, char *argv[])
{
    int freq[MAXPAROLA] ; /* vettore di contatori
    delle frequenze delle lunghezze delle parole */
    char riga[MAXRIGA] ;
    int i, inizio, lunghezza ;
    FILE * f ;

    for(i=0; i<MAXPAROLA; i++)
        freq[i]=0 ;

    if(argc != 2)
    {
        fprintf(stderr, "ERRORE, serve un parametro con il nome del file\n");
        exit(1);
    }
    f = fopen(argv[1], "rt") ;
    if(f==NULL)
    {
        fprintf(stderr, "ERRORE, impossibile aprire il file %s\n", argv[1]);
        exit(1);
    }

    while( fgets( riga, MAXRIGA, f ) != NULL )
```

# Synchronization

# Critical sections

Stefano Quer, Pietro Laface, and Stefano Scanzio
Dipartimento di Automatica e Informatica
Politecnico di Torino

skenz.it/os       stefano.scanzio@polito.it

# Concurrency and Synchronization

❖ Development environment

> ➢ **concurrent** programming (using P or T)

> ➢ cooperating processes or threads

Processes or threads

❖ Issues

> ➢ Need to manipulate shared data

> ➢ **Race conditions** may arise

> ➢ There may be sections of **not-reentrant** code

Results dependent on the order of execution

❖ Solution strategy

Uninterruptible code

> ➢ appropriately **synchronize** P or T, to make their results not dependent on their relative speed

# "Too much milk problem"

| Time | Person A | Person B |
| --- | --- | --- |
| 10.00 | Look in fridge; out of milk | |
| 10.05 | Leave for store | |
| 10.10 | Get to the store | Look in fridge; out of milk |
| 10.15 | Buy milk | Leave for store |
| 10.20 | Back home | Get to the store |
| 10.25 | Put milk in fridge | Buy milk |
| 10.30 | | Back home |
| 10.35 | | Put milk in fridge<br>Hops!!! |

# LIFO - Stack

$P_i / T_i$                                                                      $P_j / T_j$

```
void push (int val) {
   if(top>=SIZE)
      return;
   stack[top] = val;
   top++;
   return;
}
```
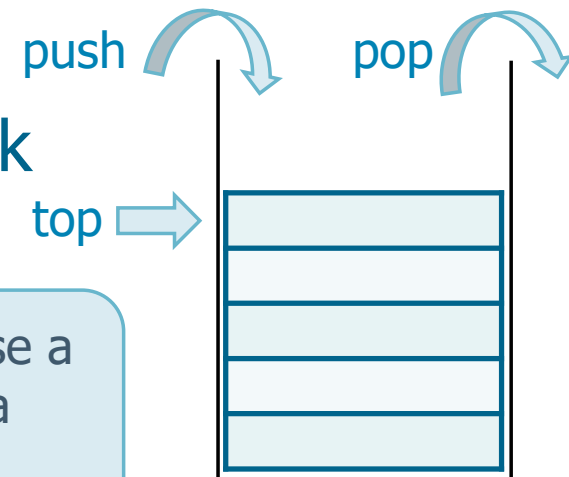
```
int pop (int *val) {
   if(top<=0)
      return;
   top--;
   *val=stack[top];
   return;
}
```

❖ **push** and **pop**

  ➢ Operate on the same end of the stack

  ➢ Variable **top** is shared

push        pop

top

top++ then top-- or viceversa
Problems?!

Can overwrite a value (lose a push), make a pop of a nonexistent value, etc.

# FIFO – Queue – Circular Buffer

$P_i / T_i$

$P_j / T_j$

```
void enqueue (int val) {
   if (n>SIZE) return;
   queue[tail] = val;
   tail=(tail+1)%SIZE;
   n++;
   return;
}
```

register = n
register = register + 1
n = register
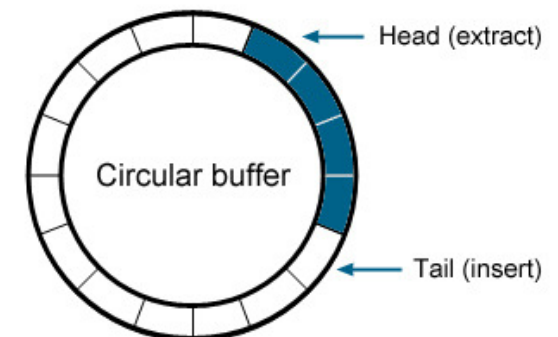
```
int dequeue (int *val) {
   if (n<=0) return;
   *val=queue[head];
   head=(head+1)%SIZE;
   n--;
   return;
}
```

register = n
register = register - 1
n = register

❖ **enqueue** and **dequeue**

➢ Operate on the different ends of the queue, using two variables **tail** and **head**

➢ Variable **n** is still shared

Increments and decrements can be lost

Head (extract)

Circular buffer

Tail (insert)

# Critical sections

❖ Critical Section (**CS**) or Critical Region (**CR**)

➢ A section of code, common to multiple processes (or threads), in which they can access (read and **write**) shared objects

❖ i.e, a **CS** or **CR** is

➢ A section of code in which multiple processes (or threads) are competing for the use (read and **write**) of shared resources (e.g., data or devices)

# Critical sections

❖ The race conditions could be prevented if

  ➢ No P (or T) executes in the same CS simultaneously

  ➢ No other P (or T) can execute, when a P (or T) executes in the CS

  ➢ The code in the CS is executed by a single P (or T) at a time

  ➢ The code in the CS is executed **in mutual exclusion**

In other words, Bernstein's conditions must fulfill

# Solution

- ❖ Solution
  - ➢ Establish an **access protocol** that enforces **mutual exclusion** for each CS
- ❖ i.e.
  - ➢ Entering a CS, a thread executes a "**reservation**" code
    - ▪ The reservation code must block (lock out) the P (or T) if another P (or T) is using its CS
  - ➢ Leaving its CS, a P (or T) executes a code to **release** the CS
    - ▪ The release possibly unlocks another P (or T) which was waiting in the "reservation" code of its CS

# Access protocol

$P_i / T_i$                                    $P_j / T_j$

```
while (TRUE) {
   ...
   reservation code
   Critical Section
   release code
   ...
   non critical section
}
```

```
while (TRUE) {
   ...
   reservation code
   Critical Section
   release code
   ...
   non critical section
}
```

❖ Every CS is protected by an
  ➢ enter code (reservation, or prologue)
  ➢ exit code   (release, or epilogue)
❖ **Non-critical sections** should **not** be **protected**

# Conditions

❖ Each solution to the CS problem **must** match the following requirements

➢ Mutual exclusion

▪ Only one P (or T) at a time must gain access to the CS

➢ Progress

▪ If no P (or T) is in the CS, and a P (or T) wants to enter, it must be able to do it in a defined time

● Only the P (or T) in the reservation phase can participate to the selection

● No P (or T) outside the CS can block other P (or T)

▪ That is, **deadlock** between P (or T) must be avoided

# Conditions

➢ Defined wait

- There must be a maximum number of times in which other P (or T) can access the CS, before a specific P (or T) can access
- That is, we must avoid **starvation** of P (or T)

➢ Each solution should be symmetrical

- The selection of the P (or T) that must access the CS should not depend on
  - Relative priority between P (or T)
  - Relative speed between P (or T)

# Solutions

❖ **Software functions**

  ➢ Solutions without special CPU instructions, which depend on the logic of an **algorithm**

❖ **Hardware**

  ➢ Solutions based on special **hardware characteristics**, or special (atomic) CPU instructions

❖ **System calls**

  ➢ The **kernel** provides the data structures, and the related system calls, that the programmer can properly use for solving the mutual exclusion problem

**Semaphore**: introduced by Dijkstra [1965]