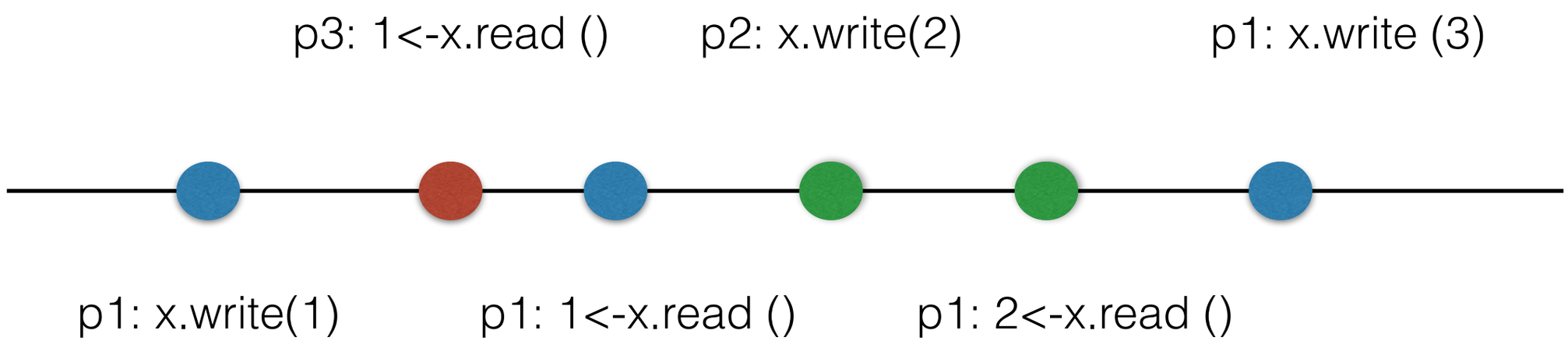
Mutual exclusion

- Sequential processes (asynchronous)
- shared objects
- E.g.:
  - read and write on a cell
  - atomic register



1 x

x.write (1) —> x:=1

x.read() —> x

- part of a code must be executed sequentially :

critical section (CS)

- code of the processes :

No failure

loop forever

non critical section

entry code

critical section

exit code

Assuming no process remains forever in CS

- part of a code must be executed sequentially :

critical section (CS)

- code of the processes :

loop forever

non critical section

entry code

critical section

exit code

requesting process

Mutual exclusion

- Safety : No two processes are in their critical sections (CS) at the same time
- Liveness :
    - **Deadlock-freedom**: at least one requesting process eventually enters its CS
    - **Starvation-freedom**: every requesting process eventually enters its CS

- Software solution: (atomic) read/write on register
  - Peterson’s lock, Lamport’s bakery algorithm
- Hardware solution :
  - test-and-set (TAS), compare-and-swap (CAS)

Peterson’s lock

- Two processes : P0 and P1
- Shared registers

first solution

P0:

```
//Entry code:
flag[0]:= true

while (flag[1] )
{//busy wait
}

//Exit code
flag[0]:=false
```

P1:

```
//Entry code:
flag[1]:= true

while (flag[0])
{//busy wait
}

//Exit code
flag[1]:=false
```

second solution

P0:

```
//Entry code:

victim:=0;
while (victim ==0)
{//busy wait
}

//Exit code
```

P1:

```
//Entry code:

victim:=1;
while (victim ==1)
{//busy wait
}

//Exit code
```

P0:

```
//Entry code:
flag[0]:= true
victim:=0;
while (flag[1] and victim ==0)
{//busy wait
}

//Exit code
flag[0]:=false
```

P1:

```
//Entry code:
flag[1]:= true
victim=1;
while (flag[0] and victim==1)
{//busy wait
}

//Exit code
flag[1]:=false
```

Peterson’s lock

- $n \geq 2$ processors : $P_0, \ldots P_{(n-1)}$
- There are n-1 « waiting room » called levels

- At each level
    - At least one thread passes
    - At least one blocked if many try
    - At most n-L threads pass level L
- Only one thread makes it through —-> CS

code de Pi

```
//Entry code:
for L:=1 to n-1 do
    level[i]:=L
    victim[L]:=i
    while (  ∃k ≠ i(level[k]≥L and victim[L] ==i))
      {//busy wait
       }

//Exit code
level[i]:=0
```

No more than n-L processes at level L

Initially a process is at level 0; a process P_a is at level j (j>0) when it completes the waiting loop with level[a]>= j

Lemme: For j (0<=j<=n-1) there are at most n-j threads at level j

Induction on j

- (base case) no more than n at level 0 —trivial
- (induction step) assume no more than n -L +1 processes at level L -1
    - at level L one process gets stuck (the last to write victim[L])

Safety

- safety :
  - no more than that n -L at level L
  - at level n-1 at most one process

Liveness

- **Starvation freedom**:
  - By reverse induction on level
  - Basic case: level n-1—trivial
  - General case: hyp: Every thread that reaches level l+1 or higher eventually enters in SC (and leaves SC)

Fairness

- Threads can be overtaken by others:
  - victim[1]=1; victim[2]=2; victim[3]= 3; 4 in SC
  - 4 out of Sc; 3 in SC; 3 out of SC; 3 and 4 require SC
  - victim[1]=3; victim[2]=1; victim[3]= 2; 4 then 2 in SC. 2 and 4 require SC
  - victim[1]=2; victim[2]=3; victim[3]= 1; 4 then 1 in SC. 1 and 4 require SC
  - victim[1]=1; victim[2]=2; victim[3]= 3; 4 then 3 in SC….

BakeryAlgorithm

- Mutual exclusion algorithm
- Fairness: first come first serve
- Lamport 1974

Init:
flag[0..n-1] init false
label[0..n-1] init 0

Code P_i

//Entry code
flag[i]:=true
label[i]=max(label[0],…,label[n-1])+1
while ($\exists k \neq i$ (flag[k] and (label[i],i)>>label[k],k)) do
{//buzy waiting}

//Exit code
flag[i]:=false

Bakery algorithm

- Mutual exclusion
- Fairness

Other synchronization problem

- Readers-writers problem
  - writer updates a file
  - reader read some part of file
  - read and write are non atomic

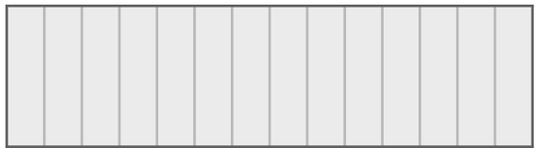
Readers-writers problem

```
Writer
write("abcdefghijkl")

write("123456789012")
```

```
Reader
read("abcdef ...")

read("... 789012")
```



abcdef789012

123456ghijkl

without synchronization inconsistent values might be read

Producer-consumer

- producers put items in the buffer (of bounded size)
- consumers get items from the buffer
- every item is consumed, no item is consumed twice
- without synchronization: items can get lost, consumed several time, bad item may be consume

- how a simple solution with a counter does not work?

Producer

```
//produce item
while (counter==MAX) {}
buffer[in]:=item
in:=(in+1) mod MAX
counter:=counter +1
```

Consumer

```
//to consume item
while (counter==0) {}
 item:=buffer[out]
out:=(out+1) mod MAX
counter:= counter -1
```

Synchronization tools

- **Test-and-set**
  - TAS(x) = if (x==1) then return 1 else x:=1 return 0
  - atomic
  - hardware implementation

init boolean lock:=0

- mutex:

```
Pi
//entry code
while (TAS(lock)){
//busy wait}

//Exit code
lock:=0
```

Drawback

- if a lot of part of the code are in mutual exclusion

  —> degrade the performance

- what happen in case of asynchrony/failures?