```
procedure echo;
var outp, inp: character;
begin
   input (inp, keyboard);
   outp := inp;
   output (outp,display);
end
```

Assume several processes use it, put it in memory so that it's shared
P_0 and P_1 access it
This can result in incorrect program flow

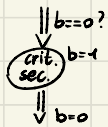
this is prevented via **mutual exclusion** within critical sections
then the problem of starvation and deadlocks arises

Software solutions

processes take care of it themselves

bit per critical section

∀ critical section keep variable to tell if there is a process there



issues:
- the test is not an atomic operation
- busy waiting

Dekker's algorithm

1st attempt: use global variable turn

PROCESS 0	PROCESS 1
..	..
while turn ≠ 0 **do** {nothing};	**while** turn ≠ 1 **do** {nothing};
<critical section>	<critical section>
turn := 1;	turn := 0;
..	..

faster process gets delayed
initialitation of turn favors one of the two

2nd attempt: keep variable flag[i] for each process i

PROCESS 0	PROCESS 1
..	..
while flag[1] **do** {nothing};	**while** flag[0] **do** {nothing};
flag[0]:=true;	flag[1]:=true;
<critical section>	<critical section>
flag[0]:=false;	flag[1]:=false;
..	..

if process gets interrupted after a while loop the other enters as well
no mutual exclusion

⇓ must switch loop and assignment

3rd attempt

PROCESS 0	PROCESS 1
..	..
flag[0]:=true;	flag[1]:=true;
while flag[1] **do** {nothing};	**while** flag[0] **do** {nothing};
<critical section>	<critical section>
flag[0]:=false;	flag[1]:=false;
..	..

if both manage to do flag[i]=true then **deadlock**

4th attempt: 3rd attempt + pause (if other wants to enter, we wait)

PROCESS 0	PROCESS 1
..	..
flag[0]:=true;	flag[1]:=true;
while flag[1] **do**	**while** flag[0] **do**
flag[0]:=false;	flag[1]:=false;
<delay a short time>	<delay a short time>
flag[0]:=true;	flag[1]:=true;
end;	**end;**
<critical section>	<critical section>
flag[0]:=false;	flag[1]:=false;
..	..

waiting time not bounded

Final solution

single variable turn to indicate which waits even if it wants to execute

```
PROCESS 0
..
flag[0]:=true;
while flag[1] do
 if turn = 1 then
    flag[0]:=false;
    while turn = 1 do {nothing};
    flag[0]:=true;
 end;
end;
<critical section>
turn:=1;
flag[0]:=false;
..
```

```
PROCESS 1
..
flag[1]:=true;
while flag[0] do
 if turn = 0 then
    flag[1]:=false;
    while turn = 0 do {nothing};
    flag[1]:=true;
 end;
end;
<critical section>
turn:=0;
flag[1]:=false;
..
```

- Busy waiting
- Priority to process that entered critical section least recently
- not trivial to generalize for n processes

Peterson's algorithm

```
PROCESS 0
..
flag[0]:=true;
turn = 1;
while flag[1] and turn = 1 do {nothing };
<critical section>
flag[0]:=false;
..
```

- announce desire to enter the CS
- set turn to the other process
- verify if P_i doesn't have intention or is in CS

can be easily generalized for n processes

```
PROCESS i
..
for j = 1 to n-1 do
   flag[i]:= j;
   turn[j] := i;
   wait until (∀ k ≠ i, flag[k] < j) or (turn[j] ≠ i);
end;
<critical section>
flag[i]:=0;
..
```

each process crosses n-1 phases before CS

Flag[i] phase of P_i

Turn[j] which process was last in phase j

Hardware based solutions

- prohibits interrupt during cs
- machine instruction support
  - test and set -> bit per cs
  - cons -> busy waiting (no guarantee against starvation

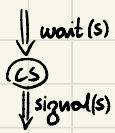
PROCESS i
..
while (testset(bolt)) **do** {nothing}; (testset(bolt): atomic)
<critical section>
bolt:=false;
..

Semaphores

Counting semaphore s is a record:
- count
- wait (s) -> count--, if count<0 block process that called wait(s) (FIFO)
- signal(s) -> count ++, if count≤0 deque from blocked processes

```
wait(s);
s.count := s.count - 1;
if s.count < 0
  then begin
    place this process in s.queue;
    block this process
  end;
```

```
signal(s);
s.count := s.count + 1;
if s.count ≤ 0
  then begin
    remove a process P from s.queue;
    place process P on the ready list
  end;
```



Binary semaphore

count only 1 or 0
- waitB(s) if counter=1 -> counter=0, else block
- signalB(s) if no process blocked, counter=1, else unlock one process

as useful as counting ones

wait and signal are atomic

Producer/consumer problem

can only take if something produced
item takeable once
must have space for new item

with binary semaphores

```
Producer:
repeat
   produce;
   waitB(s);
   append;
   n := n +1;
   if n = 1 then signalB(ne);
   signalB(s);
forever;
```
(n=num. produced - num. consumed)

```
Consumer:
waitB(ne);
repeat
   waitB(s);
   take;
   n := n - 1;
   signalB(s);
   if n = 0 then waitB(ne);
   consume;
forever;
```
(has to wait until first produced; no more to consume)

with counting semaphores

```
Producer:
repeat
   produce;
   wait(s);
   append;
   signal(s);
   signal(n);
forever;
```

```
Consumer:
repeat
   wait(n);
   wait(s);
   take;
   signal(s);
   consume;
forever;
```
(has to wait for first to be produced)

Readers/writers problem

readers want to read data but writers can modify by writers

readers can do simultaneous access (no writers allowed)

max 1 writer always active (no reader active)

r.count -> number of readers

semaphore x -> enforces mutual exclusion for r.count

wsem -> prevents writers as long as there are readers, if r.count==0 -> signal(wsem)

if r.count == 1 then wait(wsem)

Procedure: reader

```
wait(x);
   rcount := rcount+1;
   if rcount = 1 then wait(wsem);
signal(x);
READ
wait(x);
   rcount := rcount-1;
   if rcount = 0 then signal(wsem);
signal(x);
```

Procedure: writer

```
wait(wsem);
   WRITE;
signal(wsem);
```

with priority writers

procedure reader

```
wait(rsem);
 wait(x);
   rcount := rcount+1;
   if rcount = 1 then wait(wsem);
 signal(x);
signal(rsem);
READ
wait(x);
 rcount := rcount-1;
 if rcount = 0 then signal(wsem);
signal(x);
```

procedure writer:

```
wait(y);
   wcount := wcount+1;
   if wcount = 1 then wait(rsem);
signal(y);
wait(wsem);
   WRITE;
signal(wsem);
wait(y);
   wcount := wcount-1;
   if wcount = 0 then signal(rsem);
signal(y);
```