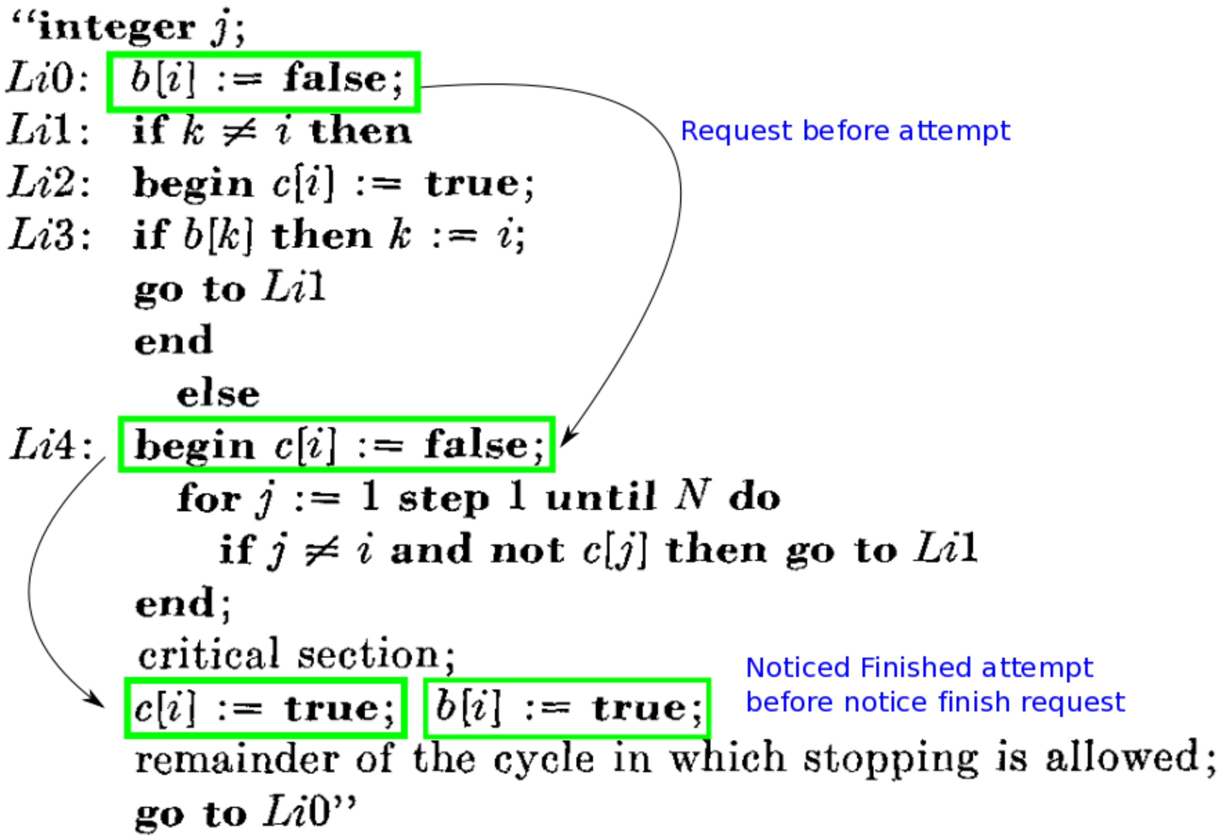
1a. the variable k is to label the intended computer, suggesting which computer should process critical section first. We could rename it as intended_index.

1b.



```
“integer j;
Li0:  b[i] := false;
Li1:  if k ≠ i then
Li2:  begin c[i] := true;
Li3:  if b[k] then k := i;
      go to Li1
      end
        else
Li4:  begin c[i] := false;
        for j := 1 step 1 until N do
          if j ≠ i and not c[j] then go to Li1
      end;
      critical section;
      c[i] := true; b[i] := true;
      remainder of the cycle in which stopping is allowed;
      go to Li0”
```

Request before attempt

Noticed Finished attempt
before notice finish request

The reason that we don’t want to b[i] = true and c[i]=false is that we don’t want a computer not requesting to enter critical section (which is b[i] =true) but attempting to enter critical section (which is c[i]=false). Based on the graph above, we see that the program follow the order:

a.	b[i] = false	request
b.	c[i] = false	attempt
c.	c[i] = true	finish attempt
d.	b[i] = true	finish request
e.	back to a	

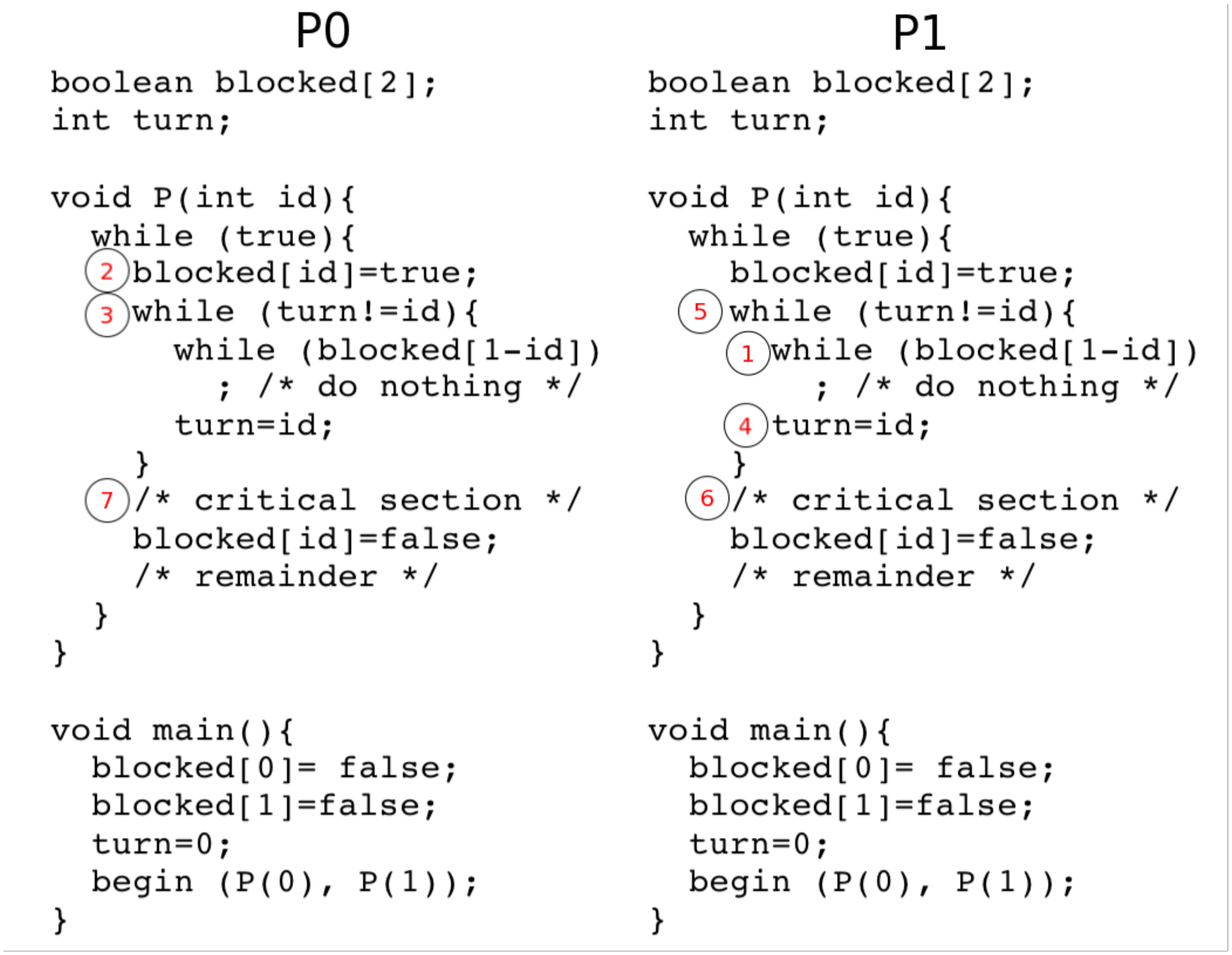
this guarantee that we will not have an attempt before have a request or we not finishing attempt but we release request.

1c. the purpose of the loop is to try to assign the the intended CS owner to current computer when the previous owner is not(or finished) requesting CS.

```
while(intended_index != i)
{                                   //not intended to give CS on this computer
   c[i] = true;                     //so this computer is not attempting to enter critical section(CS)
   if(b[intended_index]) {          //that intended computer is not requesting to enter CS
      intended_index=i;             //intend to give CS to this computer
   }
}
```

1d. Take a 32 bit integer, and for each computer i, use 1/0 in ith bit to indicate whether the ith computer is entering CS or not. Compare the integer with the number representing ith computer. This setup guarantee the checking is single memory access and atomic.

2.



P0

```
boolean blocked[2];
int turn;

void P(int id){
  while (true){
 ② blocked[id]=true;
 ③ while (turn!=id){
      while (blocked[1-id])
        ; /* do nothing */
      turn=id;
    }
 ⑦ /* critical section */
    blocked[id]=false;
    /* remainder */
  }
}

void main(){
  blocked[0]= false;
  blocked[1]=false;
  turn=0;
  begin (P(0), P(1));
}
```

P1

```
boolean blocked[2];
int turn;

void P(int id){
  while (true){
    blocked[id]=true;
 ⑤ while (turn!=id){
   ① while (blocked[1-id])
        ; /* do nothing */
   ④ turn=id;
    }
 ⑥ /* critical section */
    blocked[id]=false;
    /* remainder */
  }
}

void main(){
  blocked[0]= false;
  blocked[1]=false;
  turn=0;
  begin (P(0), P(1));
}
```

From the graph above we can see that with the order 1-7, this program will fail on mutual exclusion. Both P0 and P1 will enter critical section.