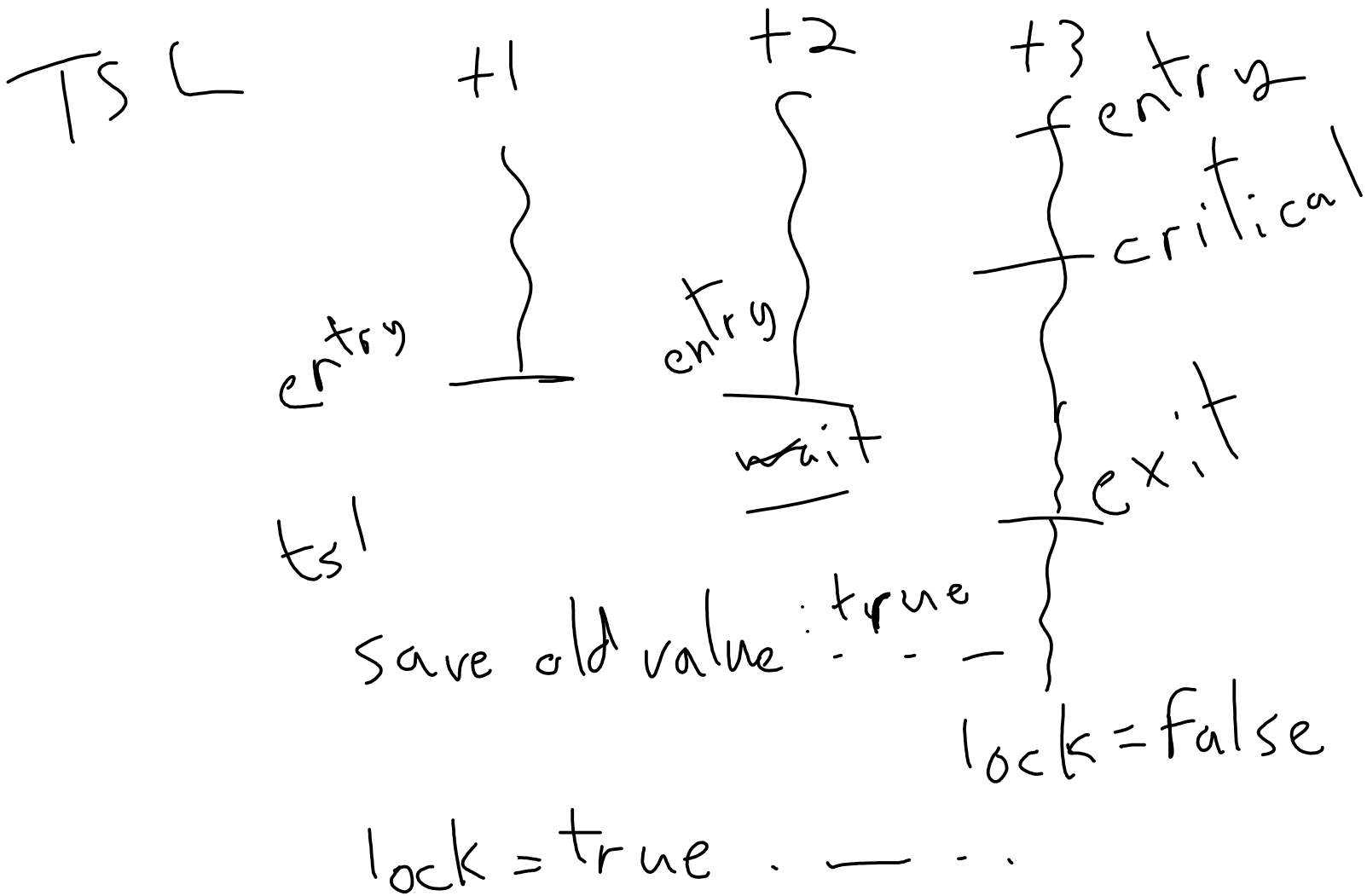
# Mutual Exclusion with Busy Waiting: Peterson's algorithm

```
#define N 2

int turn;
int interested[N];

void enter_region(int process){
 int other = 1 - process;
 interested[process] = TRUE;
 turn = process;
 while(turn==process && interested[other]==TRUE);
}

void leave_region(int process){
 interested[process] = FALSE;
}
```

# Instructions

test and set

atomic


TSL
t1
t2
t3
entry
entry
critical
wait
exit
ts1
save old value : true
lock=false
lock=true

# Semaphores

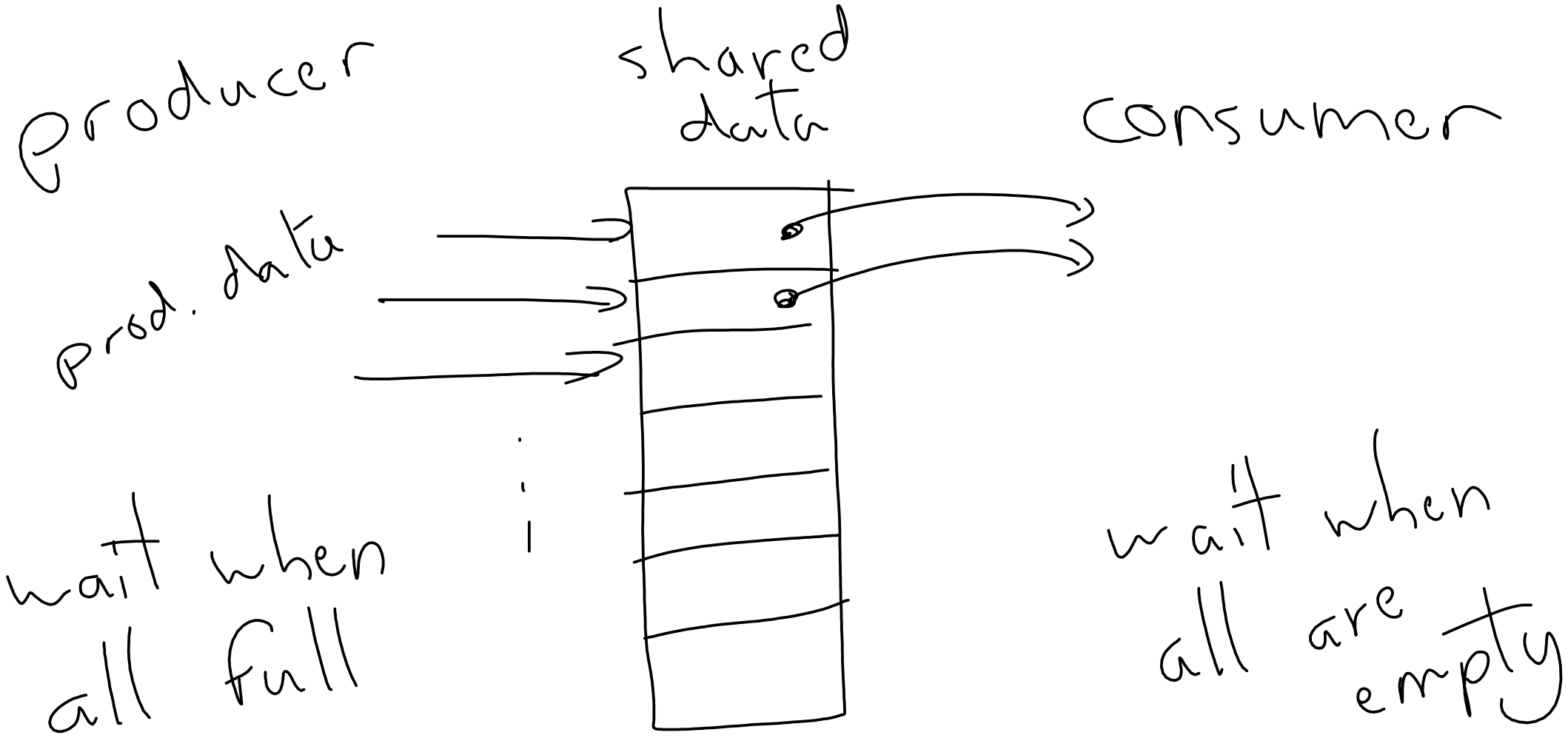
integer

# of permits

operations

| | | | |
|---|---|---|---|
| P | down | sleep | acquire |
| V | up | wake | release |

# Producer/Consumer (Bounded Buffer)



# Semaphores: Producer-Consumer (1 of 5)

```
#define N 100

typedef int sema;
sema mutex=1;
sema empty=N, full=0;

void producer(void){
 int item;
 while(TRUE){
  item = produce_item();
  down(&empty);
  down(&mutex);
  insert_item(item);
  up(&mutex);
  up(&full);
 }
}
```

```
void consumer(void){
                    int item;
                    while(TRUE){
   down(&full);
   down(&mutex);
   item = remove_item();
   up(&mutex);
   up(&empty);
   consume_item(item);
            }
}
```

# Readers/Writers Problem

# Readers/Writers (1 of 6)

- *N* processes access (i.e., read or write) some shared data
- At any given time: *R* readers **or** *1* writer allowed. Basic solution:

```
typedef int sema;
sema mutex = 1;
sema db = 1;
int rc = 0;
```

```
void reader(){
 while(TRUE){
  down(&mutex);
  rc++;
  if(rc==1) down(&db);
  up(&mutex);
  read_db();
  down(&mutex);
  rc--;
  if(rc==0) up(&db);
  up(&mutex);
  use_data_read();
 }
}
```

```
void writer(){
 while(TRUE){
  think_up_data();
  down(&db);
  write_db();
  up(&db);
 }
}
```