Process Synchronization

Bounded-Buffer – Producer

```
item next_produced;
while (true) {
        /* produce an item in next produced */
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

Bounded Buffer – Consumer

```
item next_consumed;
while (true) {
        while (in == out)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        /* consume the item in next consumed */
}
```

Producer

```
while (true) {
      /* produce an item in next
produced */

      while (counter == BUFFER_SIZE)
;
            /* do nothing */
      buffer[in] = next_produced;
      in = (in + 1) % BUFFER_SIZE;
      counter++;
}
```

Consumer

```
while (true) {
      while (counter == 0)
            ; /* do nothing */
      next_consumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;

          counter--;
      /* consume the item in next
consumed */
}
```

Race Condition

- `counter++` could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- `counter--` could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

  S1: producer execute `register1 = counter` {register1 = 5}
  S1: producer execute `register1 = register1 + 1` {register1 = 6}
  S2: consumer execute `register2 = counter` {register2 = 5}
  S3: consumer execute `register2 = register2 – 1` {register2 = 4}
  S4: producer execute `counter = register1` {counter = 6 }
  S5: consumer execute `counter = register2` {counter = 4}

Critical Section Problem

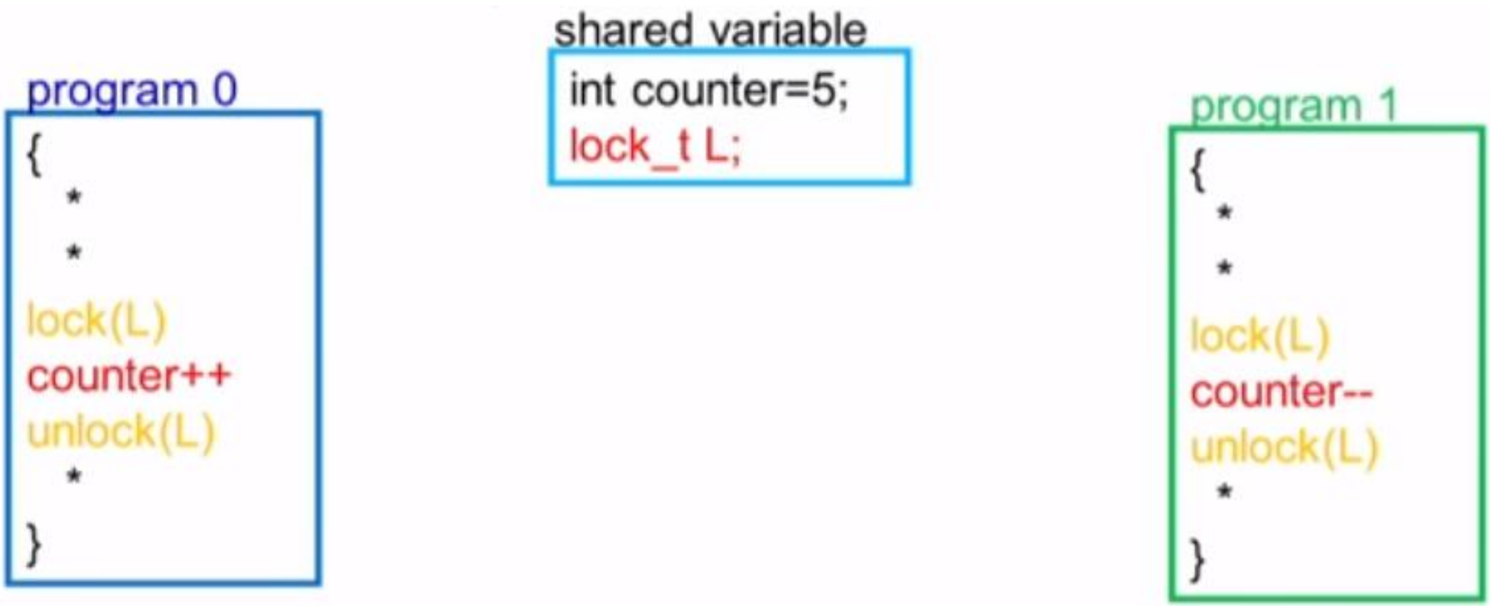
- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process ***P**$_i$*

```
do {
        entry section
              critical section
        exit section
              remainder section
} while (true);
```

Lock & Unlock



lock(L) : acquire lock L exclusively

- Only the process with L can access the critical section

unlock(L) : release exclusive access to lock L

- Permitting other processes to access the critical section

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections

2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the n processes

Solution to Critical Section Problem

Using Interrupt Disabling

```
lock   --> while(1){
              disable interrupts ()
              critical section
unlock -->    enable interrupts ()
              other code
           }
```

```
while(1){
   disable interrupts ()
   critical section
   enable interrupts ()
   other code
}
```

- Simple
  - When interrupts are disabled, context switches won't happen
- Requires privileges
  - User processes generally cannot disable interrupts
- Not suited for multicore systems

Software Solution

First Attempt

Shared

```
int turn=1;
```

Process 1

```
while(1){
    while(turn == 2); // lock
    critical section
    turn = 2;  // unlock
     other code
}
```

Process 2

```
while(1){
    while(turn == 1); // lock
    critical section
    turn = 1; // unlock
    other code
}
```

First Attempt

Shared

```
int turn=1;
```

Process 1

```
while(1){
    while(turn == 2); // lock
    critical section
    turn = 2;  // unlock
     other code
}
```

Process 2

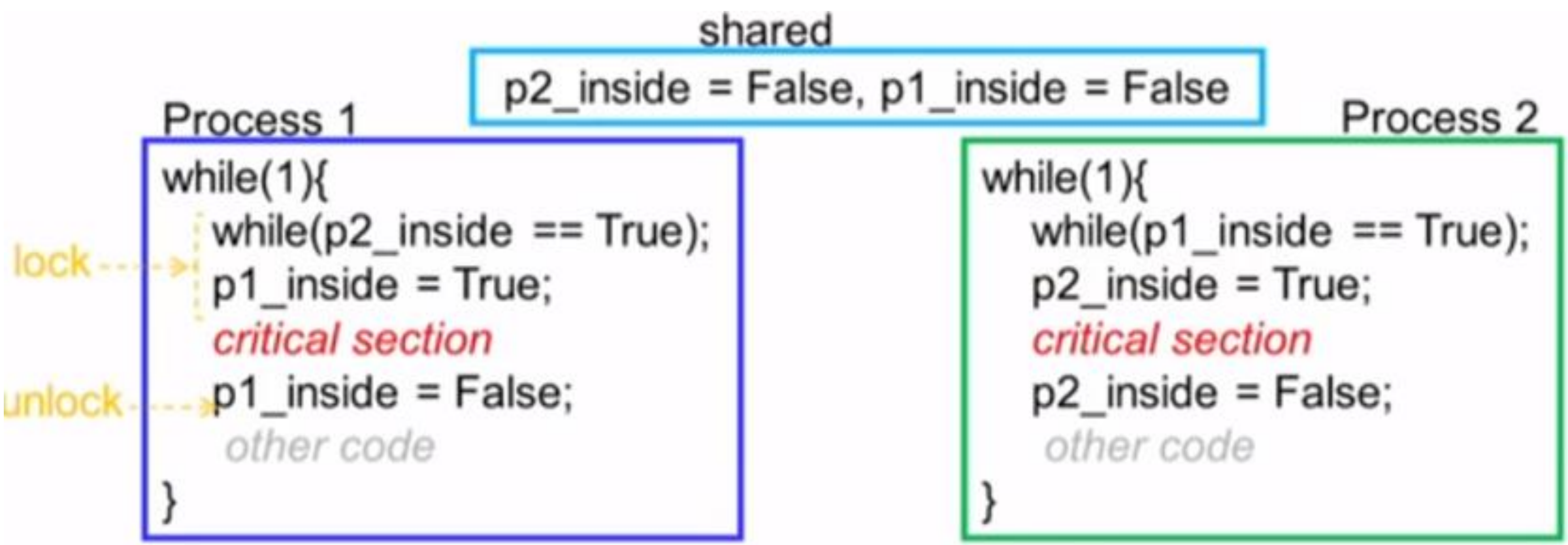
```
while(1){
    while(turn == 1); // lock
    critical section
    turn = 1; // unlock
    other code
}
```

- Achieves mutual exclusion
- Busy waiting – waste of power and time
- Needs to alternate execution in critical section

  process1 →process2 →process1 →process2

- Ensures Mutual exclusion
- Ensures Bounded Waiting
- Violates Progress

Second Attempt



Second Attempt

shared

```
p2_inside = False, p1_inside = False
```

Process 1

```
while(1){
    while(p2_inside == True);
    p1_inside = True;
    critical section
    p1_inside = False;
    other code
}
```

lock

unlock

Process 2

```
while(1){
    while(p1_inside == True);
    p2_inside = True;
    critical section
    p2_inside = False;
    other code
}
```

- Need not alternate execution in critical section
- Does not guarantee mutual exclusion

CPU	p1_inside	p2_inside
while(p2_inside == True);	False	False
context switch		
while(p1_inside == True);	False	False
p2_inside = True;	False	True
context switch		
p1_inside = True;	True	True

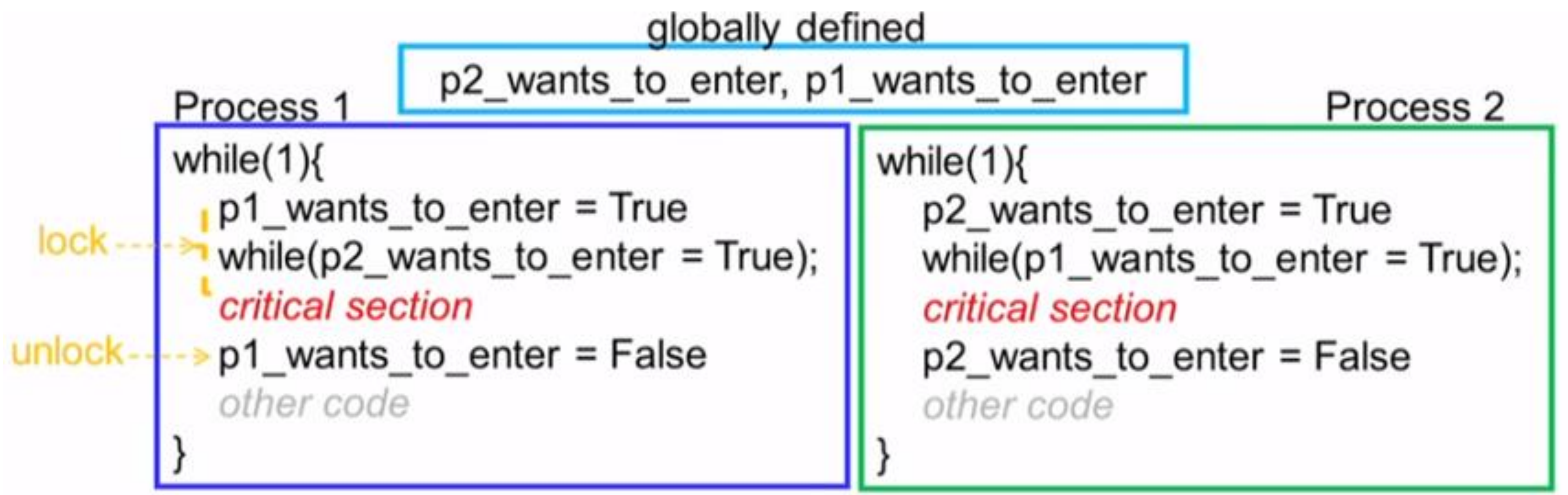
time

Both p1 and p2 can enter into the critical section at the same time

```
while(1){
   while(p2_inside == True);
   p1_inside = True;
   critical section
   p1_inside = False;
    other code
}
```

```
while(1){
   while(p1_inside == True);
   p2_inside = True;
   critical section
   p2_inside = False;
    other code
}
```

Third Attempt



Third Attempt

globally defined

```
p2_wants_to_enter, p1_wants_to_enter
```

Process 1

```
while(1){
    p1_wants_to_enter = True
    while(p2_wants_to_enter = True);
    critical section
    p1_wants_to_enter = False
    other code
}
```

lock

unlock

Process 2

```
while(1){
    p2_wants_to_enter = True
    while(p1_wants_to_enter = True);
    critical section
    p2_wants_to_enter = False
    other code
}
```

- Achieves mutual exclusion
- Does not achieve progress (could deadlock)

Peterson's Solution

- Two process solution
- The two processes share two variables:
  - **`int turn;`**
  - **`Boolean pi_wants_to_enter`**
- The variable **`turn`** indicates whose turn it is to enter the critical section
- The **`pi_wants_to_enter`** is used to indicate if a process **P_i** is ready to enter the critical section. **`pi_wants_to_enter =`** ***`true`*** implies that process **P_i** is ready!

Algorithm

```
do {
    P1_wants_to_enter= true;
    turn = 2;
    while (p2_wants_to_enter && turn = = 2);
          critical section
P1_wants_to_enter= false;
          remainder section
 } while (true);
```

Pricess P1

```
do {
    P2_wants_to_enter= true;
    turn = 1;
    while (p1_wants_to_enter && turn = = 1);
          critical section
P2_wants_to_enter= false;
          remainder section
 } while (true);
```

Pricess P2

- Mutual exclusion ensured

- Progress Ensured

- Bounded Waiting ensured

Bakery algorithm for many processes

- It utilizes the concept of tokens in bakery process
- Introduced by Leslie Lamport

Simplified Bakery Algorithm

```
lock(i){
    num[i] = MAX(num[0], num[1], …., num[N-1]) + 1
    for(p = 0; p < N; ++p){
        while (num[p] != 0 and num[p] < num[i]);
    }
}
```

critical section

```
unlock(i){
    num[i] = 0;
}
```

Original Bakery Algorithm

Without atomic operation assumptions

Introduce an array of N Booleans: *choosing*, initially all values False.

```
lock(i){
    choosing[i] = True
    num[i] = MAX(num[0], num[1], ...., num[N-1]) + 1
    choosing[i] = False
    for(p = 0; p < N; ++p){
        while (choosing[p]);
        while (num[p] != 0 and (num[p],p)<(num[i],i));
    }
}
```

doorway

critical section

```
unlock(i){
    num[i] = 0;
}
```

Choosing ensures that a process
Is not at the doorway
i.e., the process is not 'choosing'
a value for num

(a, b) < (c, d) which is equivalent to: (a < c) or ((a == c) and (b < d))

Hardware Solution

Synchronization Hardware

Does this scheme provide mutual exclusion?

lock=0

Process 1

```
while(1){
    while(lock != 0);
    lock= 1; // lock
    critical section
    lock = 0;  // unlock
    other code
}
```

Process 2

```
while(1){
    while(lock != 0);
    lock = 1; // lock
    critical section
    lock = 0; // unlock
    other code
}
```

No

```
lock = 0
-----------------------------
P1: while(lock != 0);
-----------------------------
P2: while(lock != 0);
P2: lock = 1;
-----------------------------
P1: lock = 1;
…. Both processes in critical section
```

context switch

What is the Problem?

Process 1

```
while(1){
    while(lock != 0);
    lock= 1; // lock
    critical section
    lock = 0;  // unlock
    other code
}
```

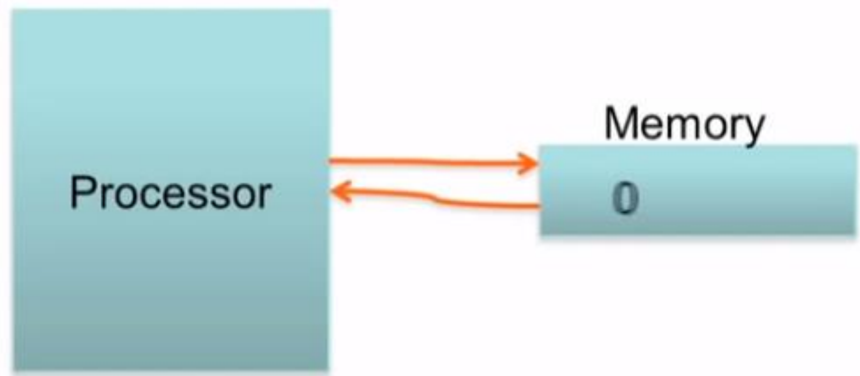
Make atomic

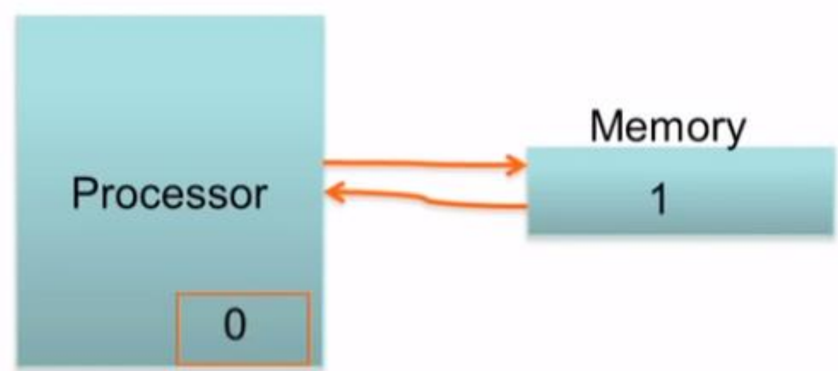
Test & Set

- Write to a memory location, return its old value

atomic

```
int test_and_set(int  *L){
  int prev = *L;
  *L = 1;
  return prev;
}
```





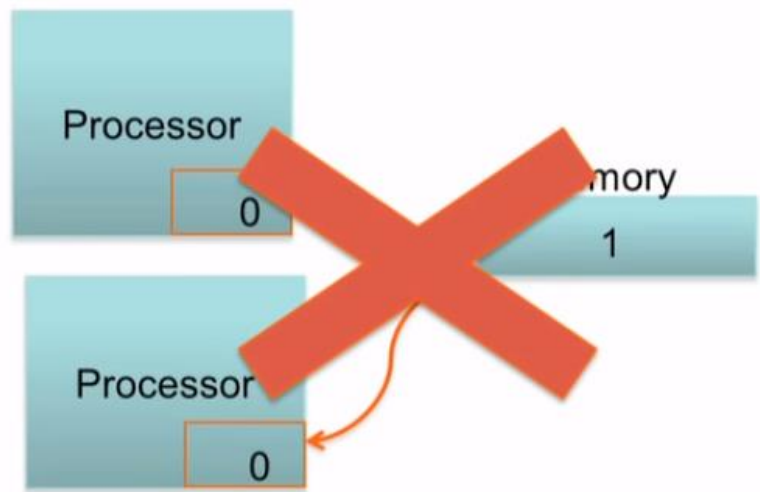
Test & Set in Multi-Processor Environment

- Write to a memory location, return its old value

atomic

```
int test_and_set(int  *L){
   int prev = *L;
   *L = 1;
   return prev;
}
```

equivalent software representation (the entire function is executed atomically)



Why does this work? If two CPUs execute test_and_set at the same time, the hardware ensures that one test_and_set does both its steps before the other one starts.

Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
boolean lock = False
do {
        while (test_and_set(&lock)) ; /*
         do nothing */
        /* critical section */
         lock = false;
         /* remainder section */
      } while (true);
```

swap Instruction

Definition:

```
void swap(boolean *a, boolean *b) {
    boolean temp = *a;

    *a=*b;
    *b=temp
}
```

1. Executed atomically
2. Exchanges value of a and b

Solution using swap

- Shared integer “lock” initialized to FALSE;
- Solution:

```
        do {
            key = TRUE;
            while(key) swap(&lock,&key);
  /* do nothing */
           /* critical section */
       lock = FALSE;
           /* remainder section */
      } while (true);
```

Bounded-waiting Mutual Exclusion with test_and_set

```
boolean waiting[n] and lock are initialized to
False
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = false;
   else
      waiting[j] = false;
   /* remainder section */
} while (true);
```

High Level Constructs

Spinlock

Process 1

```
acquire(&locked)
critical section
release(&locked)
```

Process 2

```
acquire(&locked)
critical section
release(&locked)
```

- One process will acquire the lock
- The other will wait in a loop repeatedly checking if the lock is available
- The lock becomes available when the former process releases it

- acquire(*locked){

```
key = true;
do {
    while(test_and_set(locked));
    break;
} while (true);
```

```
release(*locked){
    locked=0;
}
```

Characteristic : busy waiting

- Useful for short critical sections, where much CPU time is not wasted waiting
  - eg. To increment a counter, access an array element, etc.

- Not useful, when the period of wait is unpredictable or will take a long time
  - eg. Not good to read page from disk.
  - Use mutex instead (…mutex)

Mutex

Can we do better than busy waiting?

- If critical section is locked then yield CPU
  - Go to a SLEEP state
- While unlocking, wake up sleeping process

- lock(*locked){

```
key = true;
 if (!test_and_set(locked))
      break;
else
           sleep();
 }

unlock(*locked){
      locked=0;
      wakeup();
  }
```

Thundering Herd Problem & Solution

A large number of processes wake up (almost simultaneously) when the event occurs.

- All waiting processes wake up
- Leading to several context switches
- All processes go back to sleep except for one, which gets the critical section
  - Large number of context switches
  - Could lead to starvation

- When entering critical section, push into a queue before blocking
- When exiting critical section, wake up only the first process in the queue

Priority and Lock

What happens when a high priority task requests a lock, while a low priority task is in the critical section

- Priority Inversion
- Possible solution
    - Priority Inheritance

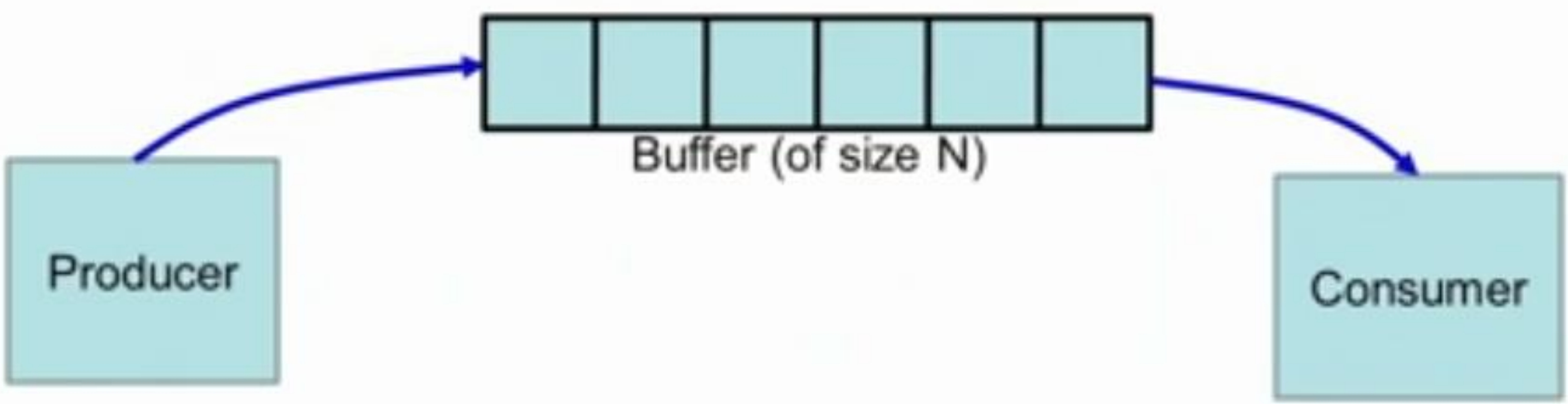
Producer Consumer using Mutex

Also known as *Bounded buffer Problem*

Producer produces and stores in buffer, Consumer consumes from buffer

Trouble when

- Producer produces, but buffer is full
- Consumer consumes, but buffer is empty



Buffer of size N
int count=0;
Mutex mutex, empty, full;

```
void producer(){
    while(TRUE){
        item = produce_item();
        if (count == N) sleep(empty);
        lock(mutex);
        insert_item(item); // into buffer
        count++;
        unlock(mutex);
        if (count == 1) wakeup(full);
    }
}
```

```
void consumer(){
    while(TRUE){
        if (count == 0) sleep(full);
        lock(mutex);
        item = remove_item(); // from buffer
        count--;
        unlock(mutex);
        if (count == N-1) wakeup(empty);
        consume_item(item);
    }
}
```

Lost Wakeup

Consider the following context of instructions

Assume buffer is initially empty

```
3 read count value // count ← 0
-------------------------------------------
3 item = produce_item();
5 lock(mutex);
6 insert_item(item); // into buffer
7 count++;  // count = 1
8 unlock(mutex)
9 test (count == 1)  // yes
9 signal(full);
-------------------------------------------  context switch
3 test (count == 0) // yes
3 wait();
```

Note, the wakeup is lost.
Consumer waits even though buffer is not empty.
Eventually producer and consumer will wait infinitely

consumer
still uses the old value of count (ie 0)

Semaphore

- Proposed by Dijkstra in 1965
- Functions down and up must be atomic
- down also called **P** (Proberen Dutch for try)
- up also called **V** (Verhogen, Dutch form make higher)
- Can have different variants
  - Such as blocking, non-blocking
- If S is initially set to 1,
  - Blocking semaphore similar to a Mutex
  - Non-blocking semaphore similar to a spinlock

```
void down(int *S){
 while( *S <= 0);
 *S--;
}

void up(int *S){
  *S++;
}
```

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "`synch`" initialized to 0

```
P1:
    S1;
    up(synch);
P2:
    down(synch);
    S2;
```

- Can implement a counting semaphore S as a binary semaphore

Semaphore Implementation

- Must guarantee that no two processes can execute the `down()` and `up()` on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the `down` and `up` code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

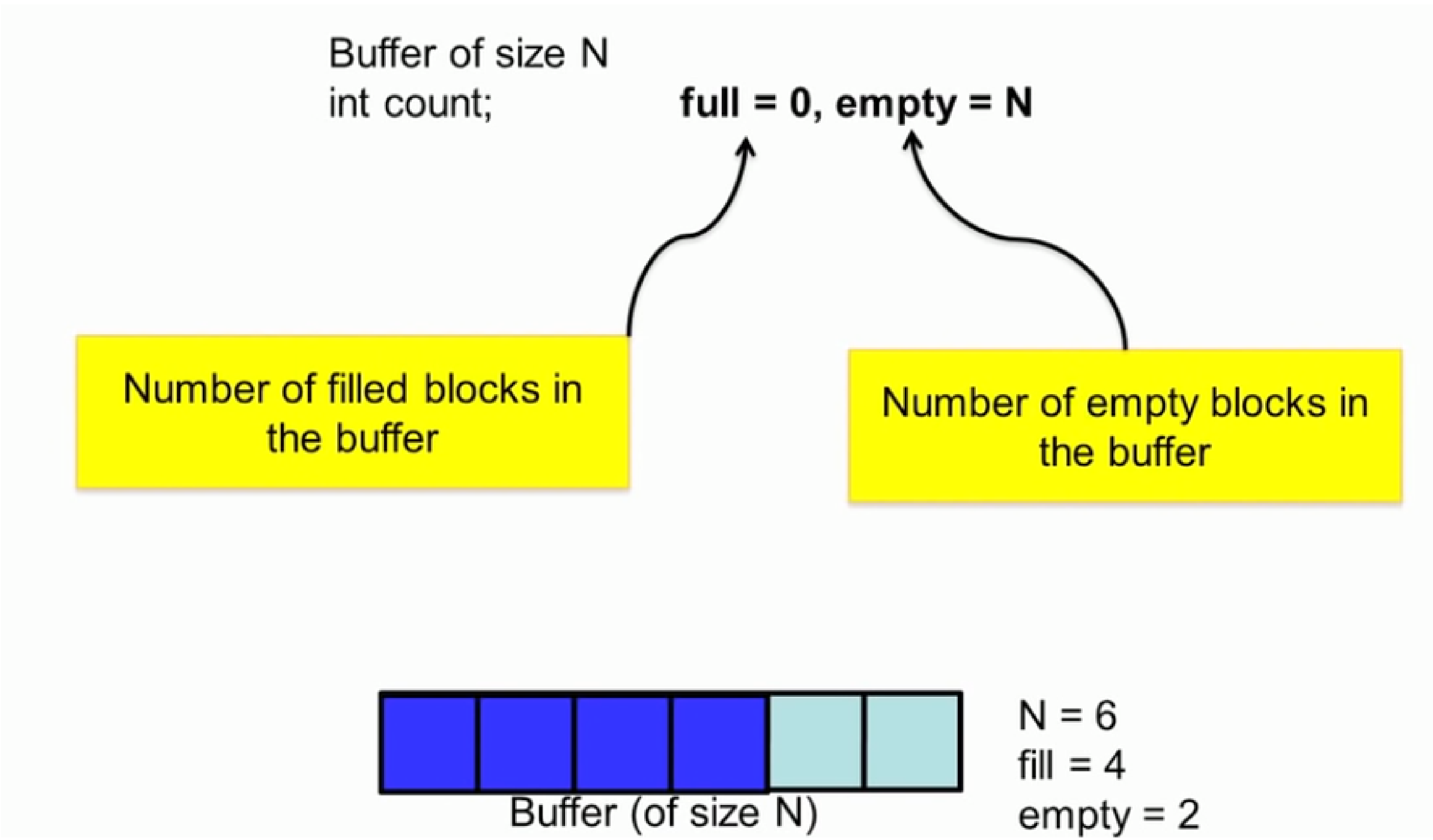
- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue
-
```
typedef struct{
 int value;
 struct process *list;
 } semaphore;
```

Implementation with no Busy waiting (Cont.)

```
down(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}

up(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

Producer Consumer with Semaphore



Producer Consumer With Semaphore

```
void producer(){
   while(TRUE){
      item = produce_item();
      down(empty);

     insert_item(item); // into buffer

      up(full);
   }
}
```

```
void consumer(){
   while(TRUE){
      down(full);

      item = remove_item(); // from buffer

      up(empty);
      consume_item(item);
   }
}
```

Serializing Access to the Buffer

```
void producer(){
    while(TRUE){
        item = produce_item();
        down(empty);
        lock(mutex)
        insert_item(item); // into buffer
        unlock(mutex)
        up(full);
    }
}
```

```
void consumer(){
    while(TRUE){
        down(full);
        lock(mutex)
        item = remove_item(); // from buffer
        unlock(mutex)
        up(empty);
        consume_item(item);
    }
}
```

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore `rw_mutex` initialized to 1
  - Semaphore `mutex` initialized to 1
  - Integer `read_count` initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
     wait(rw_mutex);
        ...
     /* writing is performed */
        ...
   signal(rw_mutex);
} while (true);
```

Readers-Writers Problem (Cont.)

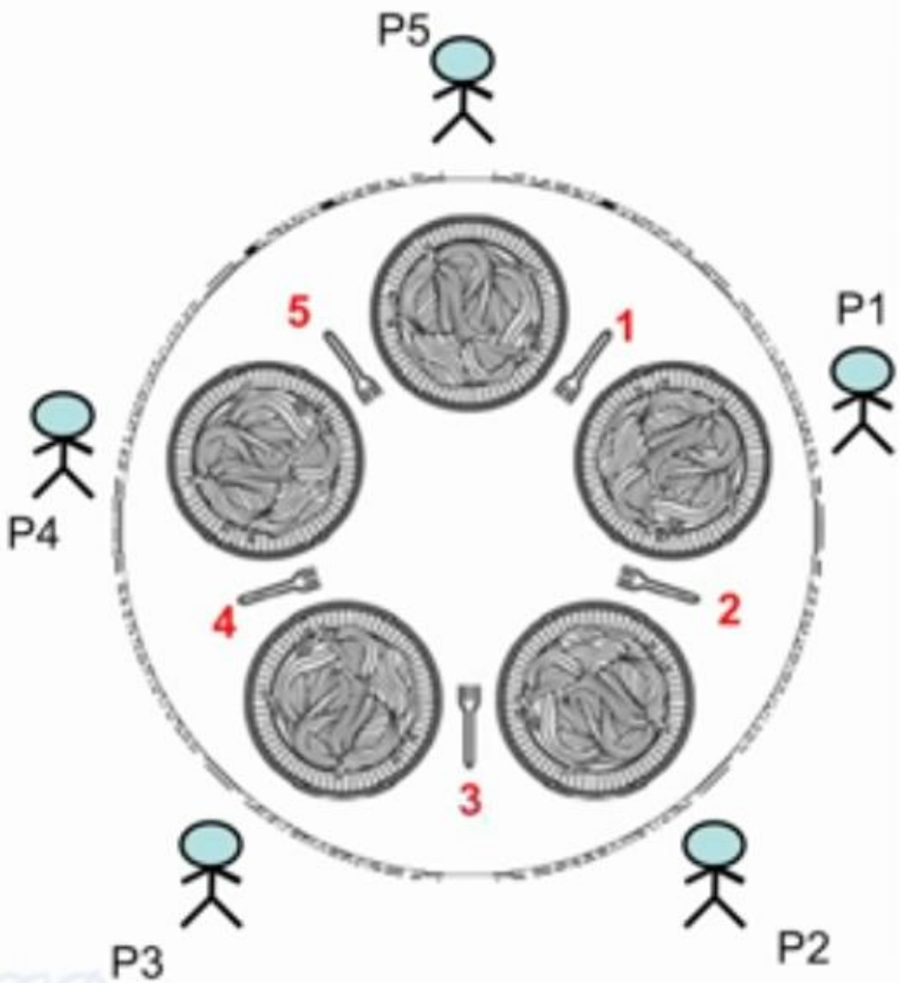
- The structure of a reader process

```
do {
        wait(mutex);
        read_count++;
        if (read_count == 1)
         wait(rw_mutex);
     signal(mutex);
          ...
        /* reading is performed */
          ...
     wait(mutex);
        read count--;
        if (read_count == 0)
     signal(rw_mutex);
     signal(mutex);
  } while (true);
```

Readers-Writers Problem Variations

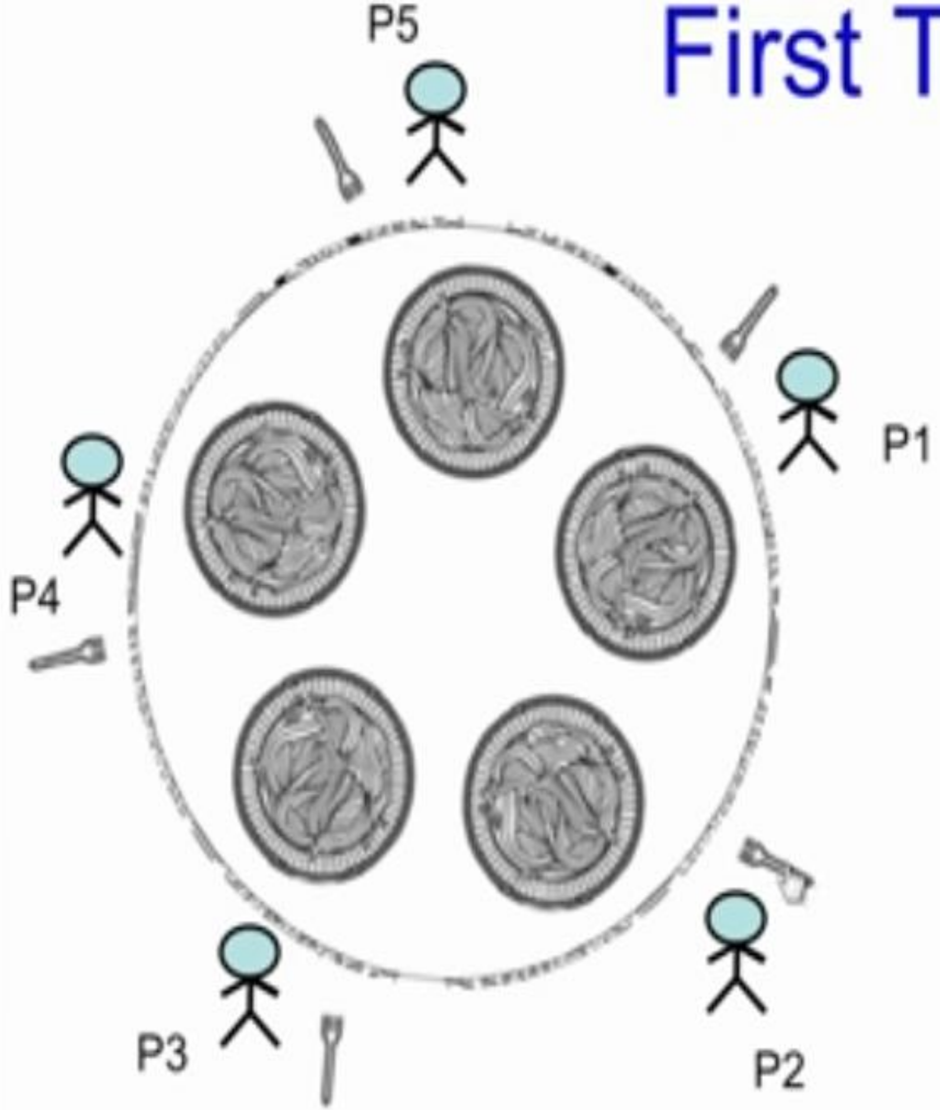
- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Dining Philosophers Problem



- **Philosophers either think or eat**
- To eat, a philosopher needs to hold both forks (the one on his left and the one on his right)
- If the philosopher is not eating, he is thinking.
- Problem Statement : Develop an algorithm where no philosopher starves.

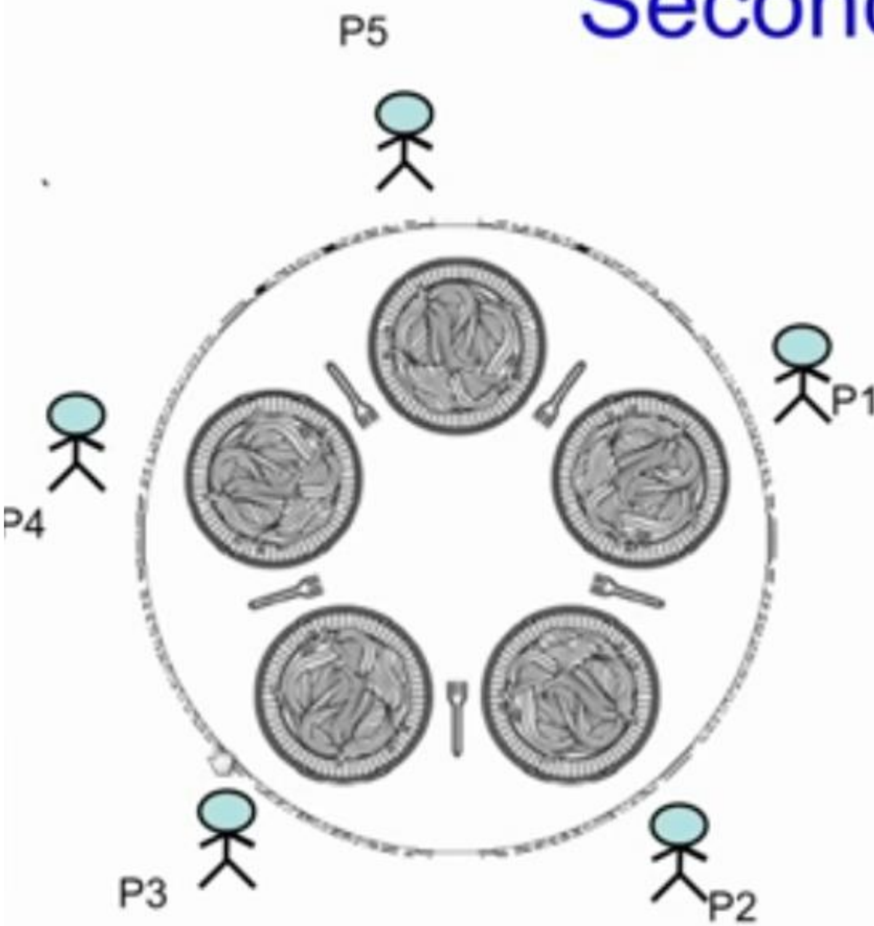
First Try




```
#define N 5

void philosopher(int i){
 while(TRUE){
     think(); // for some_time
     take_fork(R_i);
     take_fork(L_i);
     eat();
     put_fork(L_i);
     put_fork(R_i);
 }
}
```

Second try




```
#define N 5

void philosopher(int i){
  while(TRUE){
     think();
     take_fork(R_i);
     if (available(L_i){
       take_fork(L_i);
       eat();
       put_fork(R_i);
       put_fork(L_i);
     }else{
        put_fork(R_i);
        sleep(T)
     }
}
```

Solution with Mutex

Protect critical sections with a mutex

Prevents deadlock

But has performance issues

- Only one philosopher can eat at a time

```
#define N 5

void philosopher(int i){
  while(TRUE){
     think(); // for some_time
     lock(mutex);
     take_fork(R_i);
     take_fork(L_i);
     eat();
     put_fork(L_i);
     put_fork(R_i);
     unlock(mutex);
  }
}
```

Solution with Semaphore

Uses N semaphores (s[1], s[2], …., s[N]) all initialized to 0, and a mutex
Philosopher has 3 states: HUNGRY, EATING, THINKING
A philosopher can only move to EATING state if neither neighbor is eating

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

```
void test(int i){
    if (state[i] = HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING){
        state[i] = EATING;
        up(s[i]);
    }
}
```

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible.