Concurrency and Synchronisation

Leonid Ryzhyk

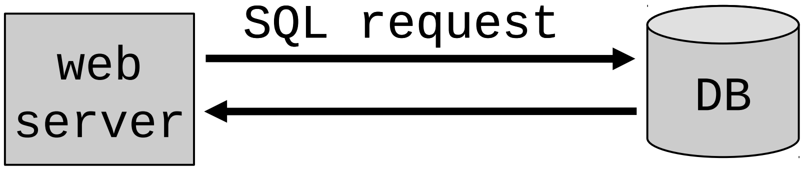
THE UNIVERSITY OF
NEW SOUTH WALES

Textbook

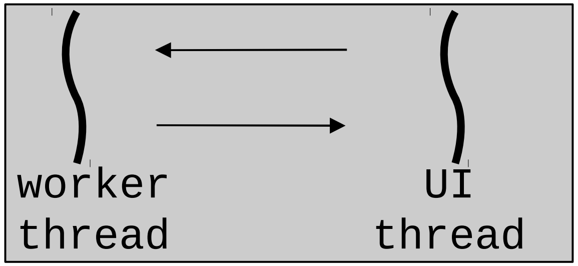
- Sections 2.3 & 2.5

Concurrency in operating systems

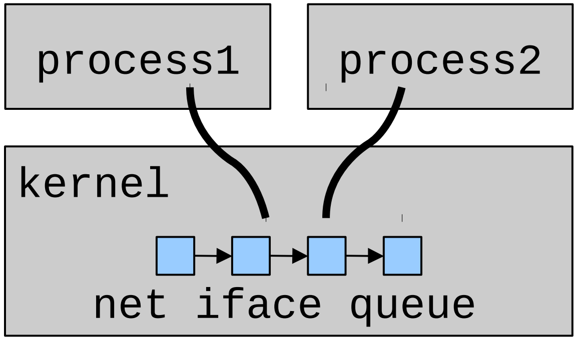
- Inter-process communication



- Intra-process communication

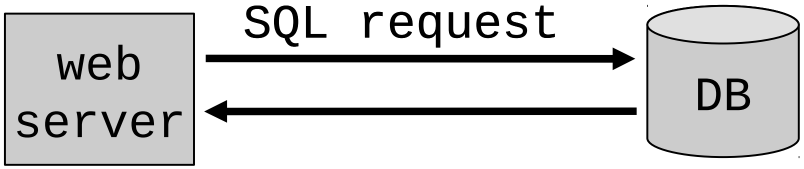


- Concurrency in the kernel

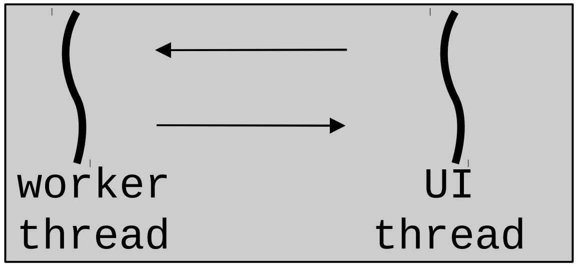


Concurrency in operating systems

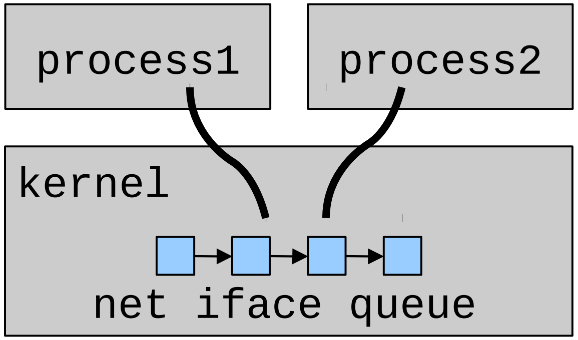
- Inter-process communication

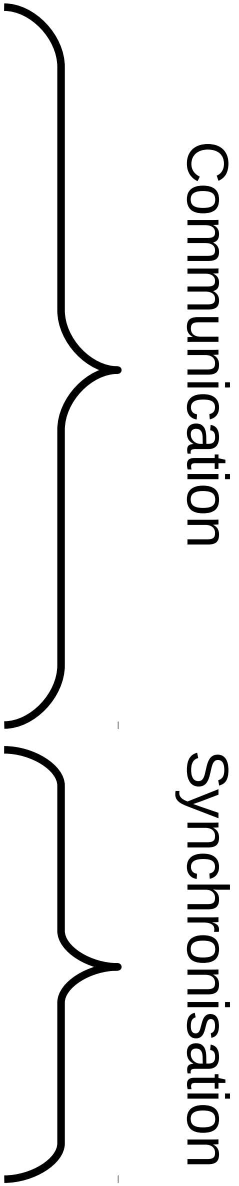


- Intra-process communication



- Concurrency in the kernel

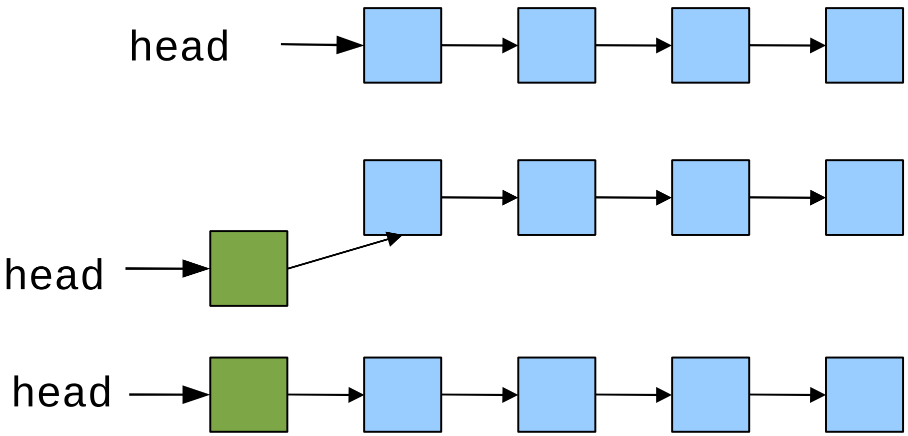




Concurrent vs sequential

- Sequential: program state depends on its previous state and the last instruction

```
void insert(struct node *item)
{
  item->next = head;
  head = item;
}
```



Concurrent vs sequential

- Concurrent: must take thread interleavings into account

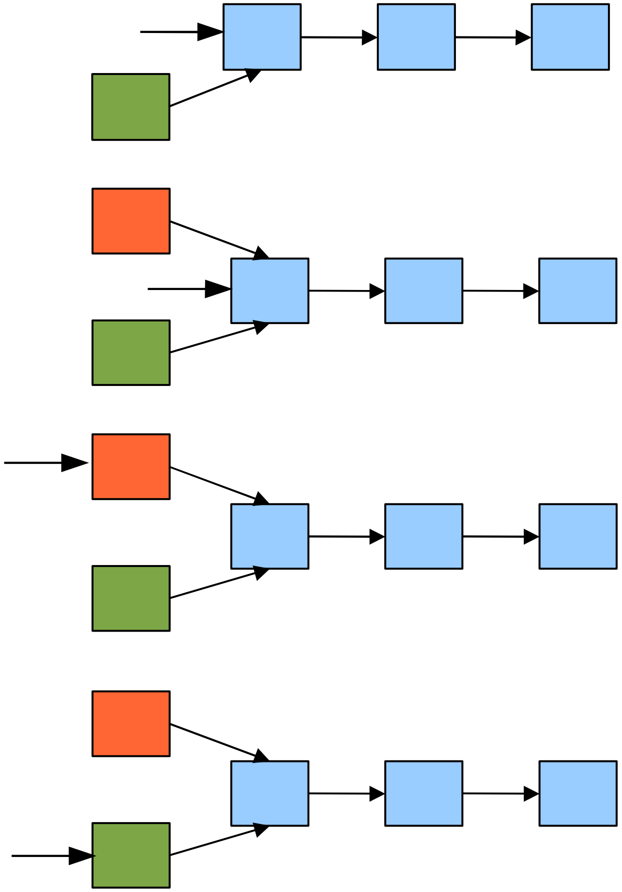
```
void insert(item)
{
  item->next = head;
  head = item;
}
```

```
void insert(item)
{
  item->next = head;
  head = item;
}
```

Concurrent vs sequential

- Concurrent: must take thread interleavings into account

```
void insert(item)
{
  item->next = head;
```

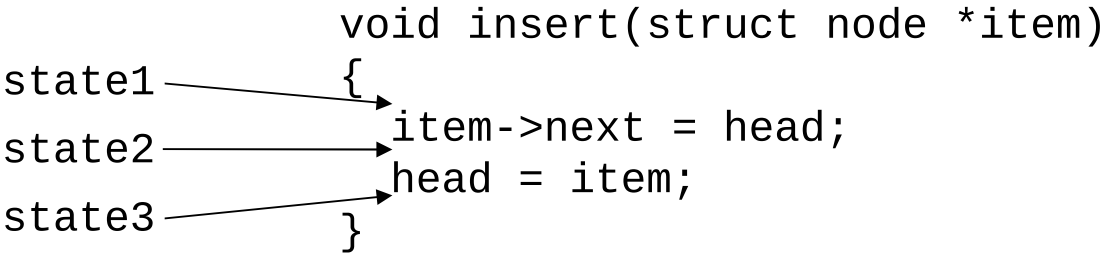


```
void insert(item)
{

  item->next = head;

  head = item;
}
```

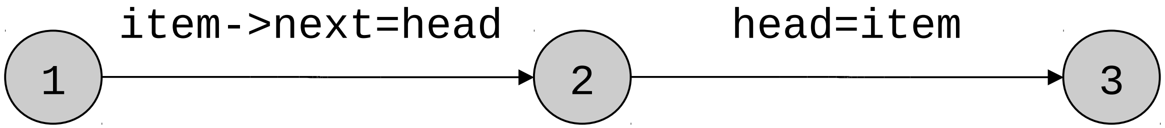
```
  head = item;
}
```

Race conditions

- Race condition: the result of the computation depends on the relative speed of two or more processes
  - Occur non-deterministically
  - Hard to debug



```
void insert(struct node *item)
{
    item->next = head;
    head = item;
}
```



3 states, 2 transition, 1 execution trace

Race conditions

- Race condition: the result of the computation depends on the relative speed of two or more processes
  - Occur non-deterministically
  - Hard to debug

```
void insert(struct node *item)
{
  item->next = head;
  head = item;
}
```

```
void insert(struct node *item)
{
  item->next = head;
  head = item;
}
```

- Question: How many states?

Question 1

Question 1

Race conditions

```
void insert(struct node *item)
{
  item->next = head;
  head = item;
}
```

...

```
void insert(struct node *item)
{
  item->next = head;
  head = item;
}
```

N processes

- Question: How many states?

Question 2

Observation

- Unfortunately, it is usually easier to show something does not work, than it is to prove that it does work.
  - Ideally, we'd like to prove, or at least informally demonstrate, that our solutions work.

Dealing with race conditions

- Approach 1: Mutual exclusion
  - Identify shared variables
  - Identify code sections that access these variables (*critical sections* or *critical regions*)
  - Ensure that at most one process can enter a critical section

Dealing with race conditions

- Approach 2: Lock-free data structures
  - Allow concurrent access to shared variables, but make sure that they end up in a consistent state
  - Hard for non-trivial data structures
  - Performance overhead in the non-contended case

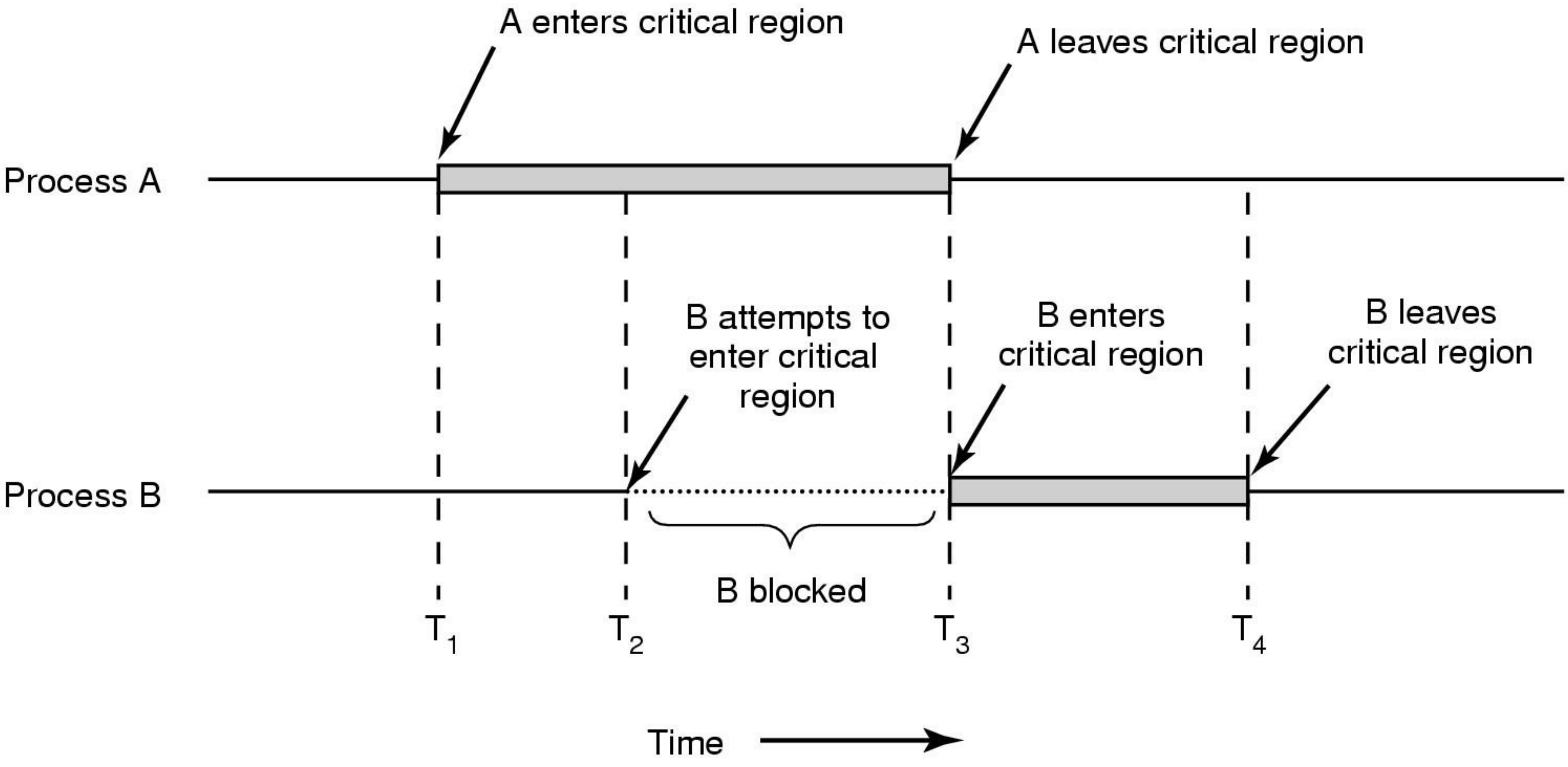
Dealing with race conditions

- Approach 3: Message-based communication
  - Eliminate shared variables
  - Processes communicate and synchronise using message passing

Mutual exclusion

- We can control access to the shared resource by controlling access to the code that accesses the resource
- Programming primitives:
  - `enter_region()` - called at the entrance to the critical region
  - `leave_region()` - called at the exit from the critical region

Mutual exclusion



Mutual exclusion using critical regions

Example critical sections

```
void insert(struct node *item)
{
   enter_region(lock);
   item->next = head;
   head = item;
   leave_region(lock);
}
```

```
struct node *remove(void)
{
   struct node *t;
   enter_region(lock);
   t = head;
   if (t != NULL) {
      head = head->next;
   }
   leave_region(lock);
   return t;
}
```

Implementing `enter_region` and `leave_region`

Requirements

- Mutual exclusion
  - No two processes simultaneously in the critical section
- No assumptions made about speeds of numbers of CPUs
- Liveness
  - No process must wait forever to enter the critical section

A solution?

- A lock variable
  - If lock == 1,
    - somebody is in the critical section and we must wait
  - If lock == 0,
    - nobody is in the critical section and we are free to enter

A solution?

```
while(TRUE) {
   while(lock == 1);
   lock = 1;
   critical();
   lock = 0
   non_critical();
}
```

```
while(TRUE) {
    while(lock == 1);
    lock = 1;
    critical();
    lock = 0
    non_critical();
}
```

- Question: Any issues with this solution?

Question 3

Mutual exclusion by taking turns

```
while(TRUE) {
   while (turn!=0);
   critical();
   turn = 1;
   non_critical();
}
```

```
while(TRUE) {
   while (turn!=0);
   critical();
   turn = 1;
   non_critical();
}
```

- Works due to strict alternation
- Process must wait its turn even while the other process is doing something else.
  - Does not guarantee progress if a process no longer needs a turn.
  - Poor solution when processes require the critical section at differing rates

Peterson’s solution

```
int turn;
int interested[2];

void enter_region(int process) {
   int other
   other = 1 – process;
   intereseted[process] = true;
   turn = process;
   while (turn == process && interested[other == TRUE]);
}
void leave_region(int process) {
   interested[process] = FALSE;
}
```

- Can be generalised to arbitrary number of processes
  - Run time is proportional to the maximal number of processes

Hardware support for mutual exclusion

```
while(TRUE) {
    while(lock == 1);
    lock = 1;
    critical();
    lock = 0
    non_critical();
}
```

If we could complete these 2 operations atomically, there would be no race

- Test and set instruction
  - Writes **1** to a memory location and returns its old value as a single atomic operation
    - Atomic: As an indivisible unit (even on a multiprocessor).

Mutual exclusion with test-and-set

```
void enter_region(bool* lock)
{
    while(test_and_set(lock) == 1);
}

void leave_region(bool* lock)
{
   *lock = 0;
}
```

Other atomic instructions

- Compare-and-swap
  - atomically compares the contents of a memory location to a given value and, if they are the same, modifies the contents of that memory location to a given new value.
- x86 supports atomic versions of most arithmetic instructions (using the lock prefix)

Mutual exclusion for uniprocessors

- A uniprocessor system runs one thread at a time
- Concurrency arises from preemptive scheduling

- Question (recap of week 2): how does a thread switch occur?

Question 4

Mutual exclusion by disabling interrupts

- Before entering a critical region, disable interrupts
- After leaving the critical region, enable interrupts
- Pros
  - Simple
- Cons
  - Only available in the kernel
  - Blocks everybody else, even with no contention
    - Slows interrupt response time
  - Does not work on a multiprocessor

Tackling the busy-wait problem

- Most implementations of mutual exclusion discussed so far rely on busy waiting
  - A process sits in a tight loop waiting for the critical section to become available

```
while(test_and_set(lock) == 1);
```

  - Waste of CPU cycles and energy

- Sleep / Wakeup
  - Call `sleep` to block, instead of busy waiting
  - Another process calls `wakeup` to unblock the sleeping process

Tackling the busy-wait problem

```
void enter_region(bool* lock)
{
   if (test_and_set(lock) == 1)
      sleep();
}
```

```
void leave_region(bool* lock)
{
   *lock = 0;
   wakeup();
}
```

- Question: What's wrong with this implementation?

Question 5

Tackling the busy-wait problem

- Correct solution:

```
typedef struct {
   bool locked;
   queue_t q;    // queue of processes waiting for the mutex
   bool guard;   // busy lock that protects access to the queue
} mutex;
```

Hmm, you're using a lock to implement another lock. Isn't this a chicken and egg problem?

No, because we already know how to implement a busy lock.

Tackling the busy-wait problem

- Correct solution:

```
void mutex_lock(mutex* lock) {
   enter_region(&lock->guard);
   add current process to lock->q
   mark current process as sleeping; //but keep it on the run queue
   leave_region(&lock->guard);
   schedule();                //move the process to the inactive queue
}                             //(if marked as sleeping)
```

```
void mutex_unlock(mutex* lock) {
   enter_region(&lock->guard);
   wake the first process in lock->q
   leave_region(&lock->guard);
}
```

Mutual exclusion for user-level code

- Busy locks can be implemented at the user level, but are seldom used outside the kernel

- Blocking locks can only be implemented in the kernel and can be accessed from user-level processes via system calls.

Semaphores

- Semaphores, introduced by Dijkstra (1965), are a generalisation of mutual exclusion
  - A mutex allows at most one process to use a resource
  - A semaphore allows at most N processes

- Conceptually, a semaphore is a counter with two operations:
  - `down()` - **atomically** decrement the counter or block if the counter is 0
  - `up()` - **atomically** wake up one blocked process or increment the counter if there are no blocked processes

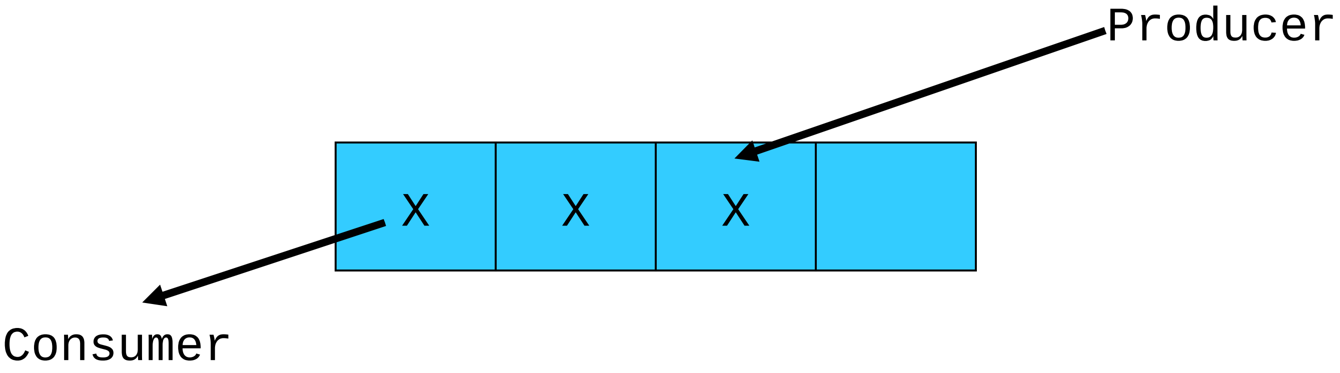
Semaphores

- A semaphore with the counter initialised to one can be used as a mutex

- Implementation of semaphores is similar to the blocking mutex implementation
  - It uses a queue of waiting processes, a counter, and a busy lock used to protect the queue and the counter
  - Sleeping is implemented via calls to the OS scheduler

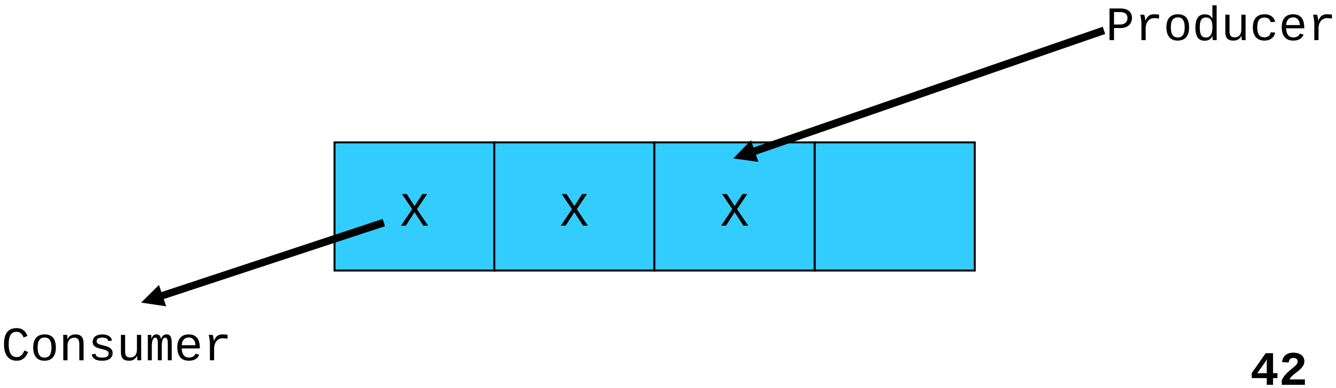
The producer-consumer problem

- Also called the bounded buffer problem
- A producer produces data items and stores the items in a buffer
- A consumer takes the items out of the buffer and consumes them.



Issues

- We must keep an accurate count of items in buffer
  - Producer
    - can sleep when the buffer is full,
    - and wakeup when there is empty space in the buffer
      - The consumer can call wakeup when it consumes the first entry of the full buffer
  - Consumer
    - Can sleep when the buffer is empty
    - And wake up when there are items available
      - Producer can call wakeup when it adds the first item to the buffer



Pseudo-code for producer and consumer

```
int count = 0;
#define N 4 /* buf size */
prod() {
   while(TRUE) {
    item = produce()
    if (count == N)
        sleep();
    insert_item();
    count++;
    if (count == 1)
        wakeup(con);
   }
}
```

```
con() {
    while(TRUE) {
        if (count == 0)
            sleep();
        remove_item();
        count--;
        if (count == N-1)
            wakeup(prod);
    }
}
```

- Question: Any issues with this pseudo-code?

Question 6

Question 6

Proposed solution

- Lets use a mutex to protect the concurrent access

Proposed solution

```
int count = 0;
#define N 4 /* buf size */
prod() {
   while(TRUE) {
    item = produce()
    if (count == N)
        sleep();
    enter_region()
    insert_item();
    count++;
    leave_region()
    if (count == 1)
        wakeup(con);
   }
}
```

```
con() {
    while(TRUE) {
        if (count == 0)
            sleep();
        enter_region()
        remove_item();
        count--;
        leave_region();
        if (count == N-1)
            wakeup(prod);
    }
}
```

Problematic execution sequence

```
prod() {
   while(TRUE) {
    item = produce()
    if (count == N)
        sleep();
    enter_region()
    insert_item();
    count++;
    leave_region()
    if (count == 1)
        wakeup(con);
   }
}
```

wakeup without a matching sleep is lost

```
con() {
    while(TRUE) {
        if (count == 0)
            sleep();
        enter_region()
        remove_item();
        count--;
        leave_region();
        if (count == N-1)
            wakeup(prod);
    }
```

Problem

- The test for *some condition* and actually going to sleep needs to be atomic

- The following does not work

```
enter_region()
if (count == N)
 sleep();
leave_region()
```

The lock is held while asleep ⇒ count will never change

Solving the producer-consumer problem with semaphores

```
#define N = 4

semaphore mutex = 1;

/* count empty slots */
semaphore empty = N;

/* count full slots */
semaphore full = 0;
```

Solving the producer-consumer problem with semaphores

```
prod() {
   while(TRUE) {
       item = produce()
       down(empty);
       down(mutex)
       insert_item();
       up(mutex);
       up(full);
   }
}
```

```
con() {
   while(TRUE) {
       down(full);
       down(mutex);
       remove_item();
       up(mutex);
       up(empty);
   }
}
```

Summarising semaphores

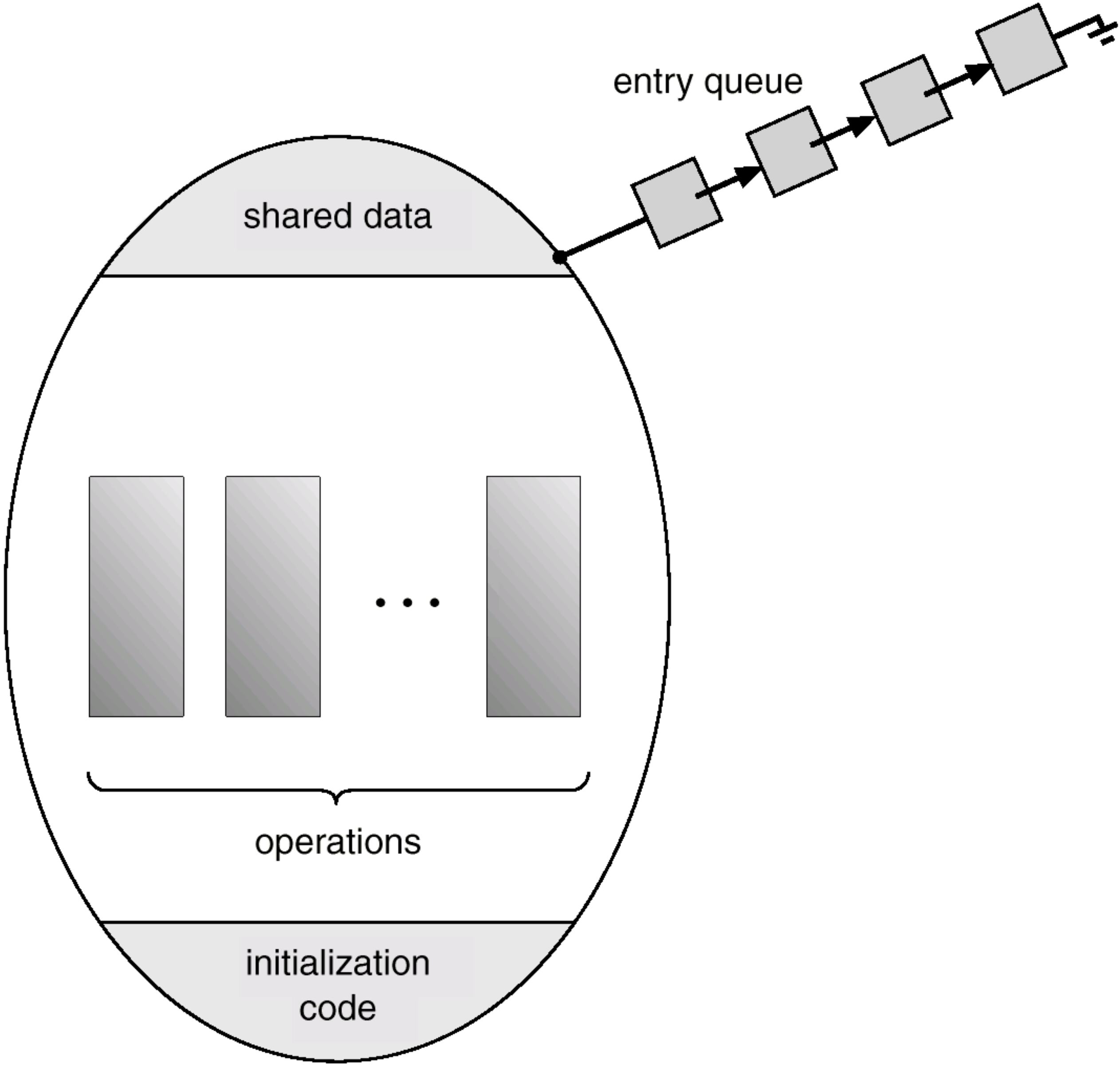
- Semaphores can be used to solve a variety of concurrency problems
- However, programming with then can be error-prone
  - Must up for every down for mutexes
    - Too many, or too few up's or down's, or up's and down's in the wrong order, can have catastrophic results
  - Must make sure that every use of a shared resource is protected by the semaphore

Monitors

- To ease concurrent programming, Hoare (1974) proposed monitors.
  - A higher level synchronisation primitive
  - Programming language construct
- Idea
  - A set of procedures, variables, data types are grouped in a special kind of module, a monitor.
    - Variables and data types only accessed from within the monitor
  - Only one process/thread can be in the monitor at any one time
    - Mutual exclusion is implemented by the compiler (which should be less error prone)

Monitor

- When a thread calls a monitor procedure that has a thread already inside, it is queued and it sleeps until the current thread exits the monitor.



Simple example

```
monitor counter
{
   int count;
   procedure inc() {
    count = count + 1;
   }
   procedure dec() {
    count = count –1;
   }
}
```

- Compiler guarantees only one thread can be active in the monitor at any one time
- Easy to see this provides mutual exclusion
  - No race condition on count.

Simple example

- Monitors provide more than just mutual exclusion
- Imagine that we want to implement a producer-consumer buffer as a monitor.

```
monitor ProducerConsumer
   integer count;
   procedure insert(item: integer);
   begin
      if count=N then
         sleep;
      ...
   end
```

```
    procedure remove: integer;
    begin
       ...
       wakeup;
       ...
     end
end monitor;
```

sleeping inside the monitor prevents other threads from entering the monitor...

...hence wakeup will never be called

How do we block waiting for an event?

- We need a mechanism to block waiting for an event inside a monitor
- *Condition Variables*

Condition variables

- To allow a process to wait within the monitor, a **condition** variable must be declared, as

  condition x, y;

- Condition variable can only be used with the operations **wait** and **signal**.
  - The operation

    x.wait();

  means that the process invoking this operation is suspended until another process invokes

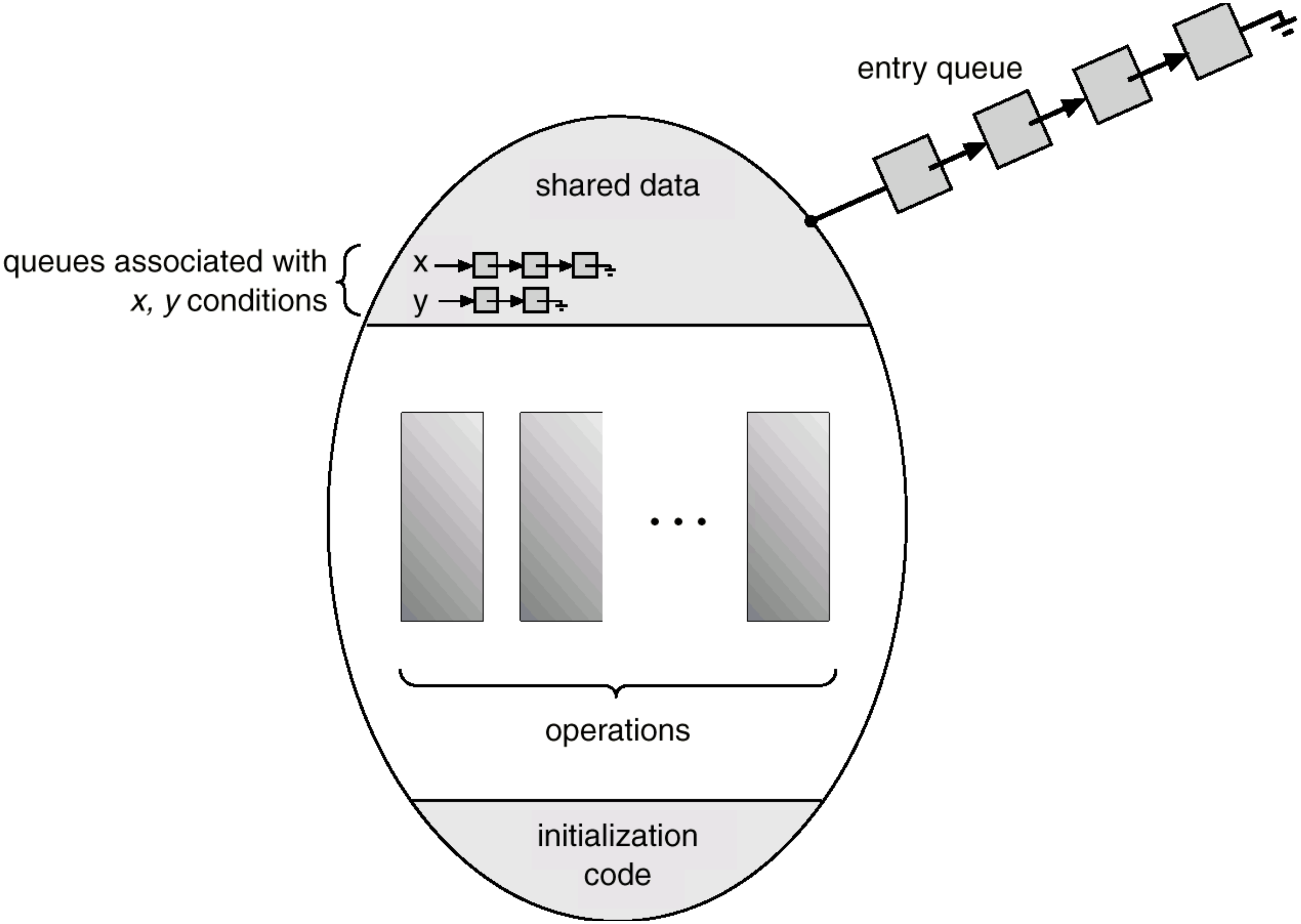
    x.signal();

  - The **x.signal** operation resumes exactly one suspended process. If no process is suspended, then the **signal** operation has no effect.

Condition variables

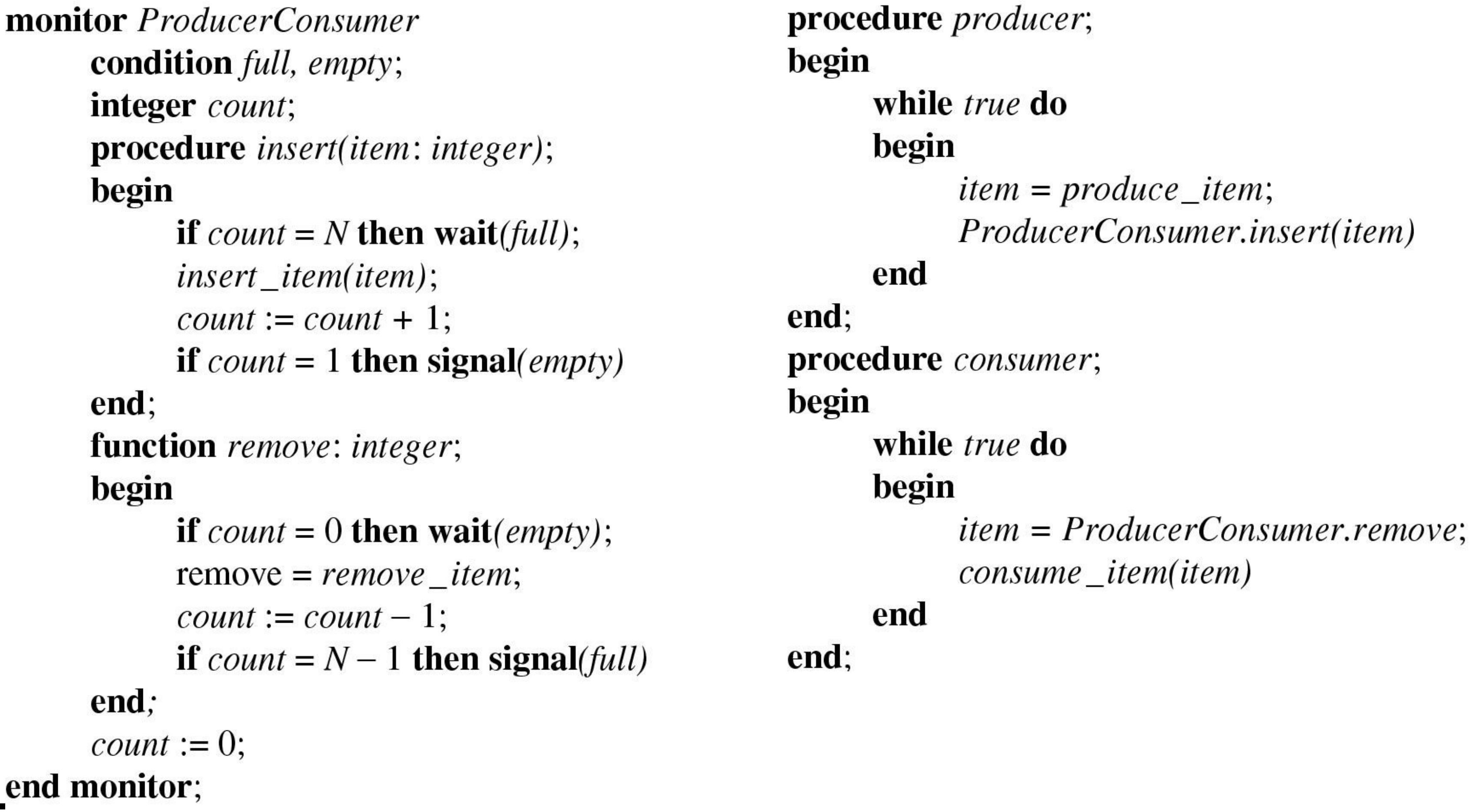
- wait() releases the monitor lock, so that other processes can enter the monitor
- The lock is re-acquired before wait() returns
- To avoid race conditions, releasing the lock and blocking the process happen as one atomic operation

Condition variables



Monitors

- Outline of producer-consumer problem with monitors



```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count − 1;
        if count = N − 1 then signal(full)
    end;
    count := 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

OS/161 provided synchronisation primitives

- Locks
- Semaphores
- Condition Variables

Locks

- Functions to create and destroy locks

```
struct lock *lock_create(const char *name);
void         lock_destroy(struct lock *);
```

- Functions to acquire and release them

```
void lock_acquire(struct lock *);
void lock_release(struct lock *);
```

Example use of locks

```
int count;
struct lock *count_lock

main() {
    count = 0;
    count_lock =
        lock_create(“count lock”);
    if (count_lock == NULL)
        panic(“I’m dead”);
    stuff();
}
```

```
procedure inc() {
    lock_acquire(count_lock);
    count = count + 1;
    lock_release(count_lock);
}
procedure dec() {
    lock_acquire(count_lock);
    count = count –1;
    lock_release(count_lock);
}
```

Semaphores

```
struct semaphore *sem_create(const char *name, int
       initial_count);
void              sem_destroy(struct semaphore *);

void              P(struct semaphore *);
void              V(struct semaphore *);
```

Example use of semaphores

```
int count;
struct semaphore *count_mutex;

main() {
    count = 0;
    count_mutex =
       sem_create(“count”, 1);
    if (count_mutex == NULL)
       panic(“I’m dead”);
    stuff();
}
```

```
procedure inc() {
    P(count_mutex);
    count = count + 1;
    V(count_mutex);
}
procedure dec() {
    P(count_mutex);
    count = count –1;
    V(count_mutex);
}
```

Condition variables

```
struct cv *cv_create(const char *name);
void       cv_destroy(struct cv *);

void       cv_wait(struct cv *cv, struct lock *lock);
```

- Releases the lock and blocks
- Upon resumption, it re-acquires the lock
  - Note: we must recheck the condition we slept on

```
void       cv_signal(struct cv *cv, struct lock *lock);
void       cv_broadcast(struct cv *cv, struct lock *lock);
```

- Wakes one/all, does not release the lock
- First "waiter" scheduled after signaller releases the lock will re-acquire the lock

Note: All three variants must hold the lock passed in.

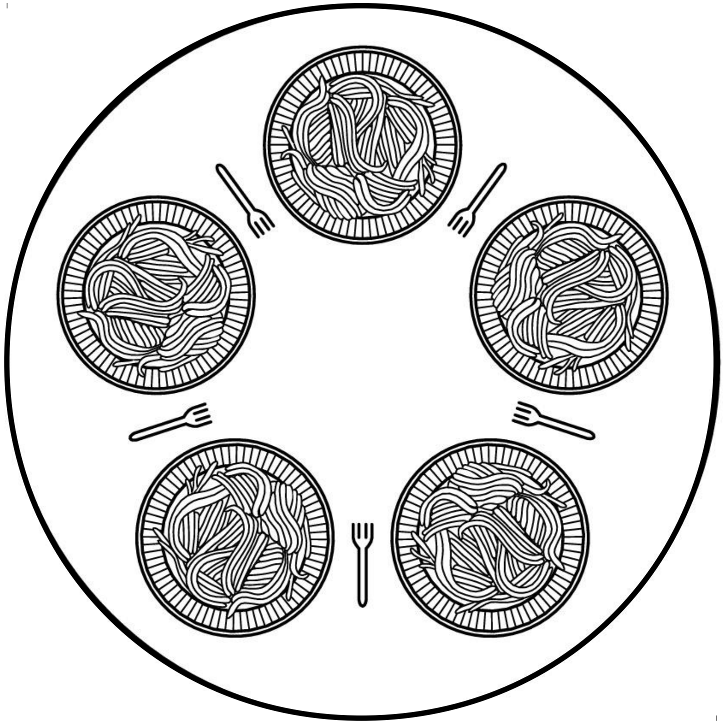
A producer-consumer solution using OS/161 CVs

```
int count = 0;
#define N 4 /* buf size */
prod() {
    while(TRUE) {
        item = produce()
        lock_aquire(l)
        while (count == N)
            cv_wait(f,l);
        insert_item(item);
        count++;
        if (count == 1)
    cv_signal(e,l);
        lock_release()
    }
}
```

```
con() {
    while(TRUE) {
        lock_acquire(l)
        while (count == 0)
            cv_wait(e,l);
        item = remove_item();
        count--;
        if (count == N-1)
            cv_signal(f,l);
        lock_release(l);
        consume(item);
    }
}
```

Dining philosophers

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock

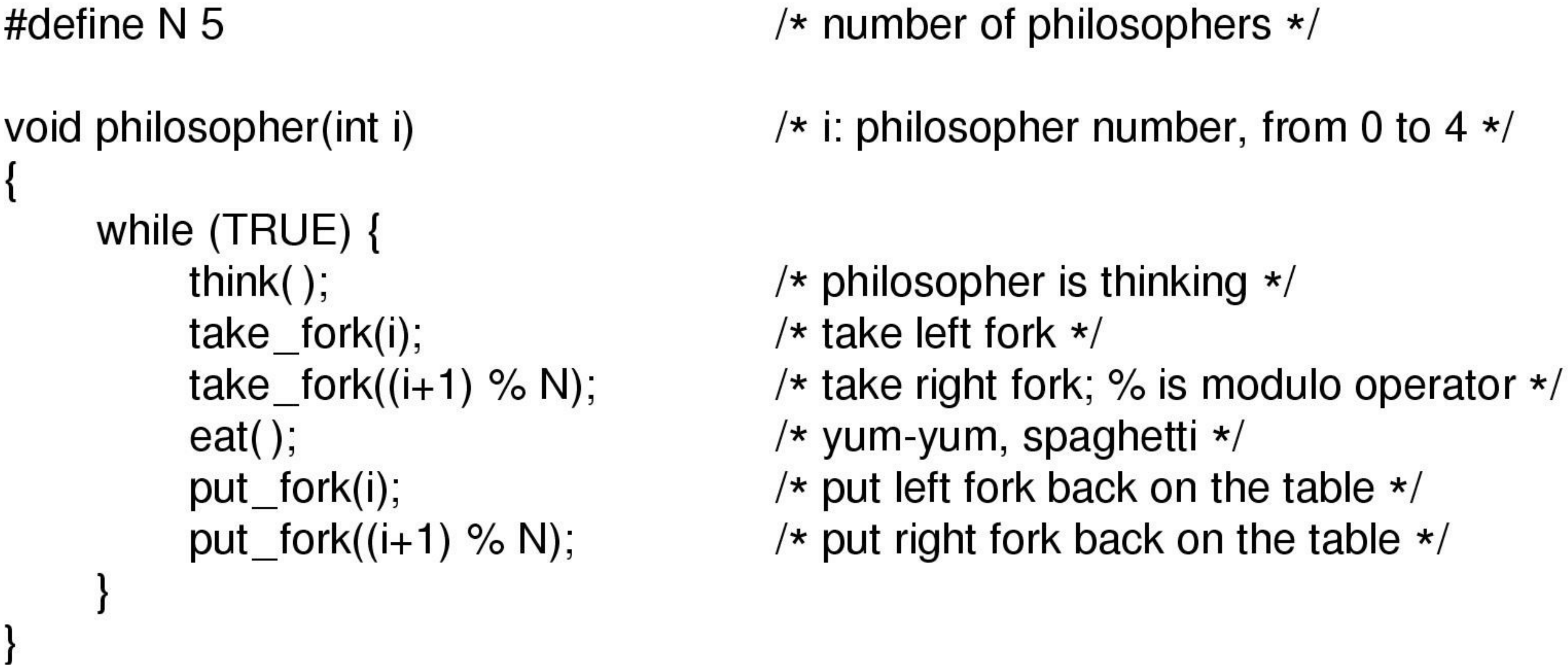


Dining philosophers

```
#define N           5            /* number of philosophers */
#define LEFT        (i+N−1)%N    /* number of i’s left neighbor */
#define RIGHT       (i+1)%N      /* number of i’s right neighbor */
#define THINKING    0            /* philosopher is thinking */
#define HUNGRY      1            /* philosopher is trying to get forks */
#define EATING      2            /* philosopher is eating */
typedef int semaphore;           /* semaphores are a special kind of int */
int state[N];                    /* array to keep track of everyone’s state */
semaphore mutex = 1;             /* mutual exclusion for critical regions */
semaphore s[N];                  /* one semaphore per philosopher */

void philosopher(int i)          /* i: philosopher number, from 0 to N−1 */
{
     while (TRUE) {              /* repeat forever */
          think( );              /* philosopher is thinking */
          take_forks(i);         /* acquire two forks or block */
          eat( );                /* yum-yum, spaghetti */
          put_forks(i);          /* put both forks back on table */
     }
}
```

Solution to dining philosophers problem (part 1)

Dining philosophers



```
#define N 5                          /* number of philosophers */

void philosopher(int i)              /* i: philosopher number, from 0 to 4 */
{
     while (TRUE) {
          think( );                  /* philosopher is thinking */
          take_fork(i);              /* take left fork */
          take_fork((i+1) % N);      /* take right fork; % is modulo operator */
          eat( );                    /* yum-yum, spaghetti */
          put_fork(i);               /* put left fork back on the table */
          put_fork((i+1) % N);       /* put right fork back on the table */
     }
}
```

A nonsolution to the dining philosophers problem

Dining philosophers

```
void take_forks(int i)                    /* i: philosopher number, from 0 to N−1 */
{
     down(&mutex);                        /* enter critical region */
     state[i] = HUNGRY;                   /* record fact that philosopher i is hungry */
     test(i);                             /* try to acquire 2 forks */
     up(&mutex);                          /* exit critical region */
     down(&s[i]);                         /* block if forks were not acquired */
}

void put_forks(i)                         /* i: philosopher number, from 0 to N−1 */
{
     down(&mutex);                        /* enter critical region */
     state[i] = THINKING;                 /* philosopher has finished eating */
     test(LEFT);                          /* see if left neighbor can now eat */
     test(RIGHT);                         /* see if right neighbor can now eat */
     up(&mutex);                          /* exit critical region */
}

void test(i)                              /* i: philosopher number, from 0 to N−1 */
{
     if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
          state[i] = EATING;
          up(&s[i]);
     }
}
```

Solution to dining philosophers problem (part 2)

The readers and writers problem

- Models access to a database
    - E.g. airline reservation system
  - Can have more than one concurrent reader
    - To check schedules and reservations
  - Writers must have exclusive access
    - To book a ticket or update a schedule

The readers and writers problem

```
typedef int semaphore;              /* use your imagination */
semaphore mutex = 1;                /* controls access to ’rc’ */
semaphore db = 1;                   /* controls access to the database */
int rc = 0;                         /* # of processes reading or wanting to */

void reader(void)
{
     while (TRUE) {                 /* repeat forever */
          down(&mutex);             /* get exclusive access to ’rc’ */
          rc = rc + 1;              /* one reader more now */
          if (rc == 1) down(&db);   /* if this is the first reader ... */
          up(&mutex);               /* release exclusive access to ’rc’ */
          read_data_base( );        /* access the data */
          down(&mutex);             /* get exclusive access to ’rc’ */
          rc = rc – 1;              /* one reader fewer now */
          if (rc == 0) up(&db);     /* if this is the last reader ... */
          up(&mutex);               /* release exclusive access to ’rc’ */
          use_data_read( );         /* noncritical region */
     }
}


void writer(void)
{
     while (TRUE) {                 /* repeat forever */
          think_up_data( );         /* noncritical region */
          down(&db);                /* get exclusive access */
          write_data_base( );       /* update the data */
          up(&db);                  /* release exclusive access */
     }
}
```

A solution to the readers and writers problem

The sleeping barber problem

The sleeping barber problem

```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
     while (TRUE) {
          down(&customers);         /* go to sleep if # of customers is 0 */
          down(&mutex);             /* acquire access to 'waiting' */
          waiting = waiting − 1;    /* decrement count of waiting customers */
          up(&barbers);             /* one barber is now ready to cut hair */
          up(&mutex);               /* release 'waiting' */
[redacted]
     }
}

void cu[redacted]
{
[redacted]
     if (waiting < CHAIRS) {        /* if there are no free chairs, leave */
          waiting = waiting + 1;    /* increment count of waiting customers */
          up(&customers);           /* wake up barber if necessary */
          up(&mutex);               /* release access to 'waiting' */
          down(&barbers);           /* go to sleep if # of free barbers is 0 */
          get_haircut( );           /* be seated and be serviced */
     } else {
          up(&mutex);               /* shop is full; do not wait */
     }
```

See the textbook

Solution to sleeping barber problem.

FYI

- Counting semaphores versus binary semaphores:
  - In a counting semaphore, *count* can take arbitrary integer values
  - In a binary semaphore, *count* can only be 0 or 1
    - Can be easier to implement
  - Counting semaphores can be implemented in terms of binary semaphores (how?)

- Strong semaphores versus weak semaphores:
  - In a strong semaphore, the *queue* adheres to the FIFO policy
  - In a weak semaphore, any process may be taken from the *queue*
  - Strong semaphores can be implemented in terms of weak semaphores (how?)