Concurrency and Synchronisation

Part II

Learning Outcomes

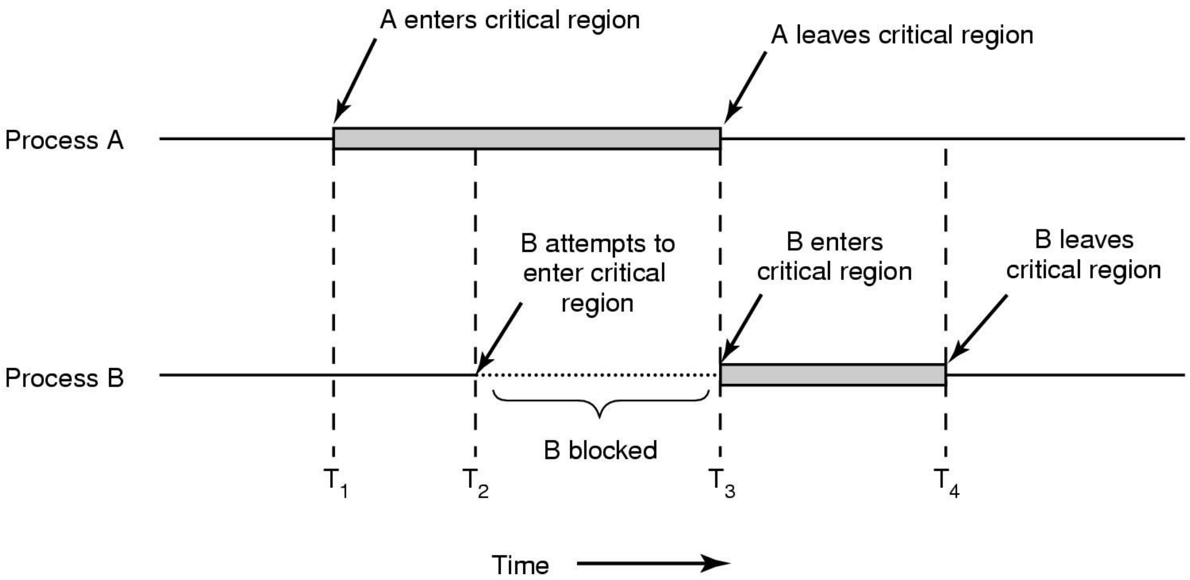
- Understand concurrency is an issue in operating systems and multithreaded applications
- Know the concept of a *critical region*.
- Understand how mutual exclusion of critical regions can be used to solve concurrency issues
  - Including how mutual exclusion can be implemented correctly and efficiently.
- Be able to identify and solve a *producer consumer bounded buffer* problem.
- Understand and apply standard synchronisation primitives to solve synchronisation problems.

Textbook

- Sections 2.3 - 2.3.7 & 2.5

Accessing Critical Regions

⇒ A *critical region* is a region of code where shared resources are accessed.



Mutual exclusion using critical regions

Test-and-Set

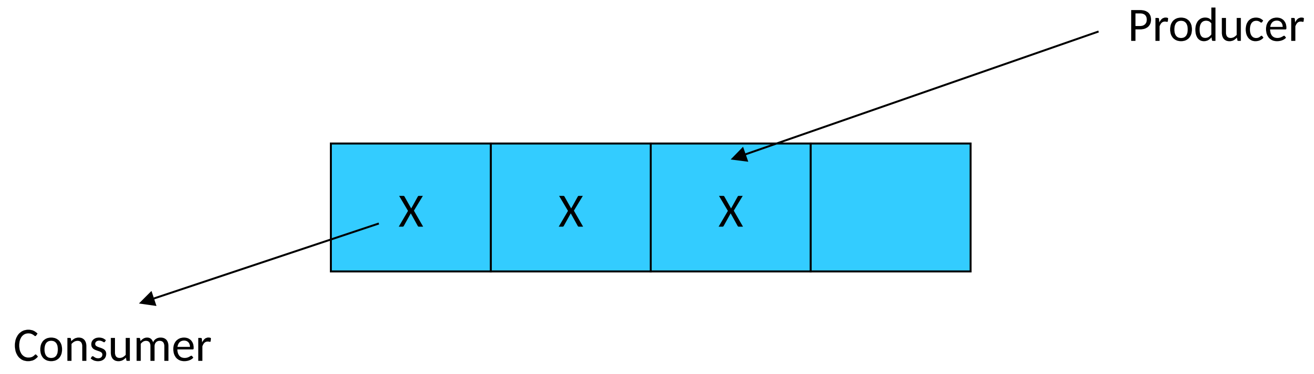
- We can use test-and-set to implement lock() and unlock() primitives
- Pros
  - Simple (easy to show it's correct)
  - Available at user-level
    - To any number of processors
    - To implement any number of lock variables
- Cons
  - Busy waits (also termed a *spin lock)*
    - Consumes CPU
    - Starvation is possible when a process leaves its critical section and more than one process is waiting.

Tackling the Busy-Wait Problem

- Sleep / Wakeup
  - The idea
    - When process is waiting for an event, it calls sleep to block, instead of busy waiting.
    - The event happens, the event generator (another process) calls wakeup to unblock the sleeping process.
    - Waking a ready/running process has no effect.

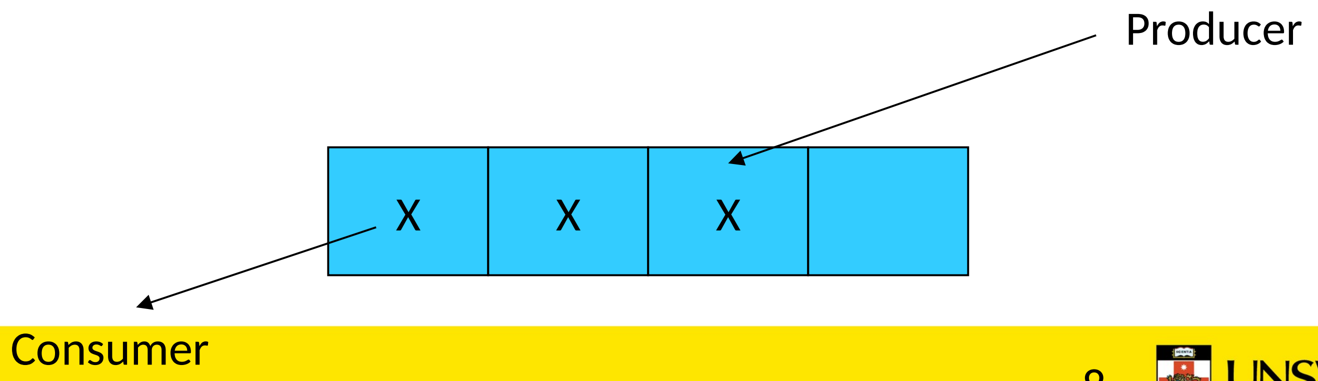
The Producer-Consumer Problem

- 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
    - should 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
    - should 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(prod);
      insert_item(item);
      count++;
      if (count == 1)
           wakeup(con);
 }
}
```

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

Problems

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

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

Concurrent uncontrolled access to the buffer

Problems

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

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

Concurrent uncontrolled access to the counter

Proposed Solution

- Lets use a locking primitive based on test-and-set to protect the concurrent access

Proposed solution?

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

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

Problematic execution sequence

```
con() {
  while(TRUE) {
  if (count == 0)
```

```
prod() {
  while(TRUE) {
        item = produce()
        if (count == N)
                sleep(prod);
        acquire_lock(buf_lock)
        insert_item(item);
        count++;
        release_lock(buf_lock)
        if (count == 1)
                wakeup(con);
```

wakeup without a matching sleep is lost

```
  sleep(con);
  acquire_lock(buf_lock)
  remove_item();
  count--;
  release_lock(buf_lock);
  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:

```
acquire_lock(buf_lock)
...
if (count == N)
            sleep(prod);
...
release_lock(buf_lock)
```

The lock is held while asleep ⇒ count will never change

```
acquire_lock(buf_lock)
...
if (count == N - 1)
            wakeup(prod);
...
release_lock(buf_lock)
```

Semaphores

- Dijkstra (1965) introduced two primitives that are more powerful than simple sleep and wakeup alone.
  - P(): *proberen*, from Dutch *to test*.
  - V(): *verhogen*, from Dutch *to increment*.
  - Also called *wait* & *signal*, *down* & *up*.

How do they work

- If a resource is not available, the corresponding semaphore blocks any process waiting for the resource
- Blocked processes are put into a process queue maintained by the semaphore (avoids busy waiting!)
- When a process releases a resource, it signals this by means of the semaphore
- Signalling resumes a blocked process if there is any, or stores the signal to be read by the next waiting task
- Wait (P) and signal (V) operations cannot be interrupted
- Complex coordination can be implemented by multiple semaphores

Semaphore Implementation

- Define a semaphore as a record

```
typedef struct {
    int count;
    struct process *L;
} semaphore;
```

- Assume two simple operations:
  - **sleep** suspends the process that invokes it.
  - **wakeup(*P*)** resumes the execution of a blocked process **P**.

- Semaphore operations now defined as

wait(*S*):

while (S.count <= 0) {

add this process to **S.L;**
sleep;

}

S.count --;

signal(*S*):
S.count++;

if (S.count <= 1) {

remove a process **P** from **S.L;**
wakeup(P);

}

- Each primitive is atomic
  - E.g. interrupts are disabled for each code fragment

Semaphore Implementation of a Mutex

```
/* initialise mutex */
semaphore mutex;
mutex.count = 1;

/* enter the critcal region */
wait(mutex);

critical();

/* exit the critical region */
signal(mutex);
```

A semaphore can restrict a region to access by N threads.

If N=1, this implements mutual exclusion.

- A mutex object.
- Also called a lock.

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();
      wait(empty);
      wait(mutex);
      insert_item();
      signal(mutex);
      signal(full);
 }
}
```

```
con() {
 while(TRUE) {
      wait(full);
      wait(mutex);
      remove_item();
      signal(mutex);
      signal(empty);
 }
}
```

Summarising Semaphores

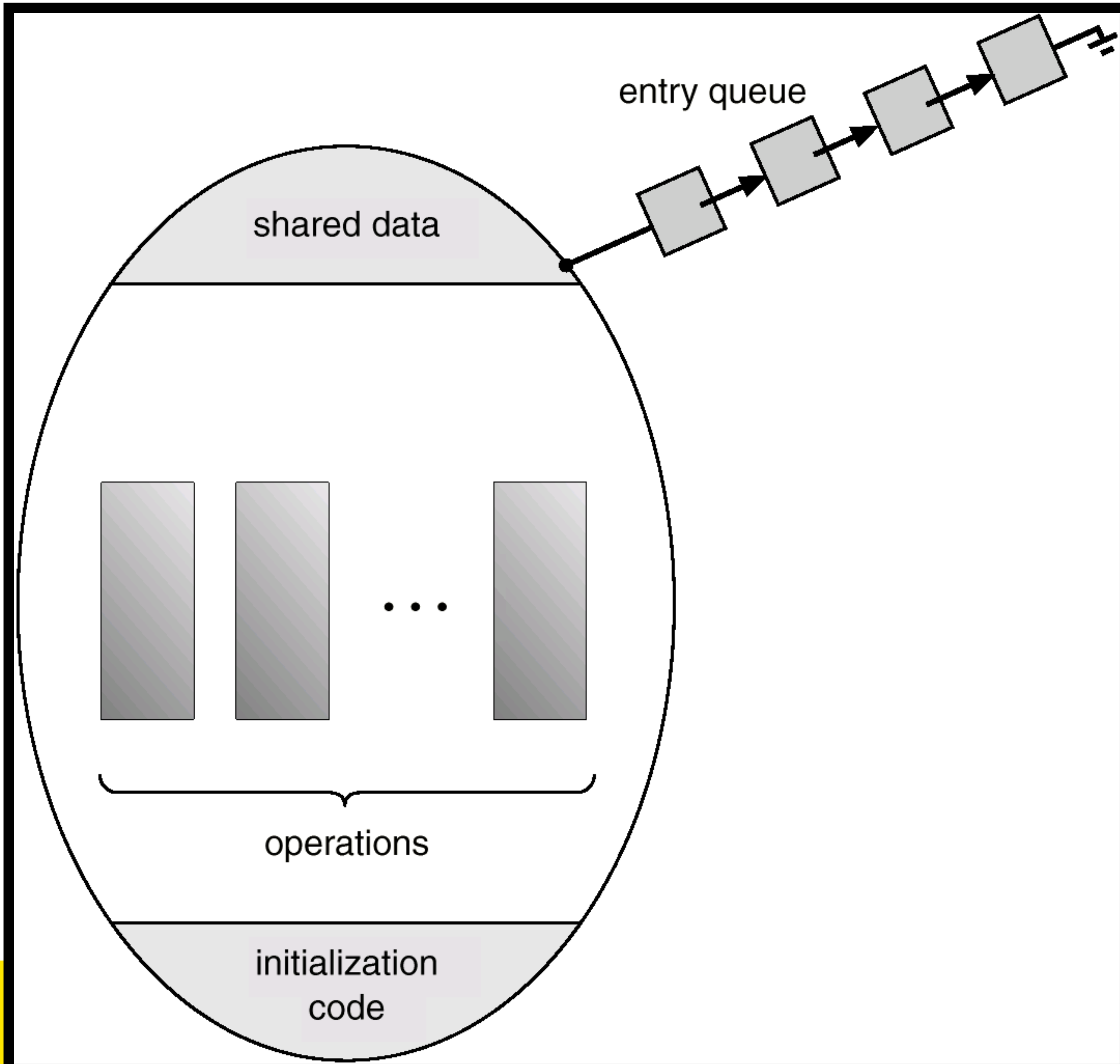
- Semaphores can be used to solve a variety of concurrency problems
- However, programming with them can be error-prone
  - E.g. must *signal* for every *wait* for mutexes
    - Too many, or too few signals or waits, or signals and waits in the wrong order, can have catastrophic results

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.



Monitors

```
monitor example
    integer i;
    condition c;

    procedure producer( );
    .
    .
    .
    end;

    procedure consumer( );
    .
    .
    .
    end;
end monitor;
```

Example of a monitor

Simple example

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

Note: “paper” language

- 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`**.
- For instance, **synchronized** methods in Java.

How do we block waiting for an event?

- We can use locks to block waiting for an object, held by another task
- We can use semaphores to solve the producer/consumer problem directly
- We would like a mechanism to block waiting for a kind of event (and also respect mutual exclusion)
  - e.g. in the producer-consumer problem
  - A blocked consumer is not waiting on just one producer
- *Condition Variables*

Condition Variable

- 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

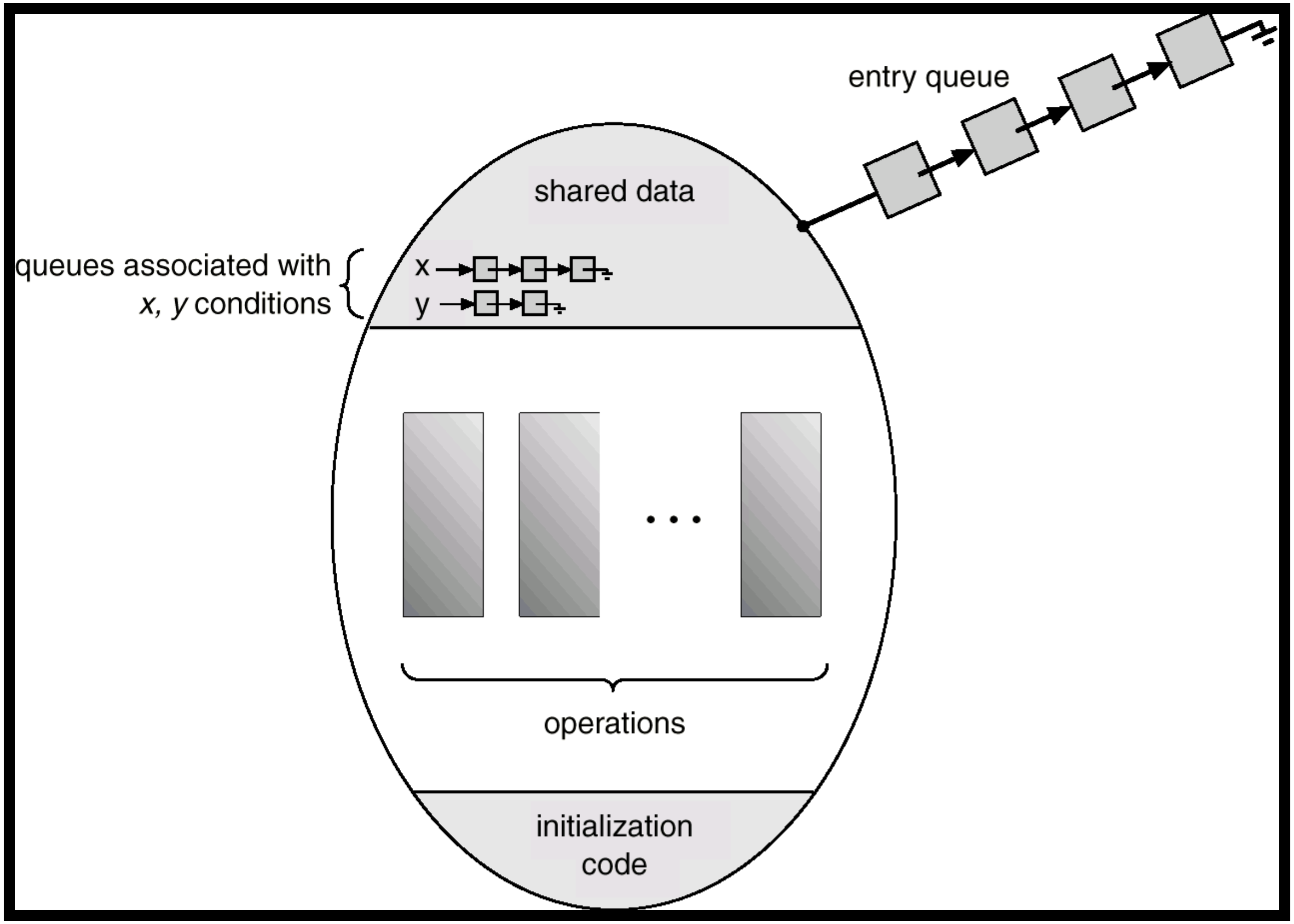
    x.wait();

    - means that the process invoking this operation is suspended until another process invokes
    - Another thread can enter the monitor while original is suspended

    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



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;
```

- Outline of producer-consumer problem with monitors
  - only one monitor procedure active at one time
  - buffer has *N* slots

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 functions must hold the lock passed in.

Condition Variables and Bounded Buffers

Non-solution

```
lock_acquire(c_lock)
if (count == 0)
        sleep();
remove_item();
count--;
lock_release(c_lock)
 ;
```

Solution

```
lock_acquire(c_lock)
while (count == 0)
 cv_wait(c_cv, c_lock);
remove_item();
count--;
lock_release(c_lock);
```

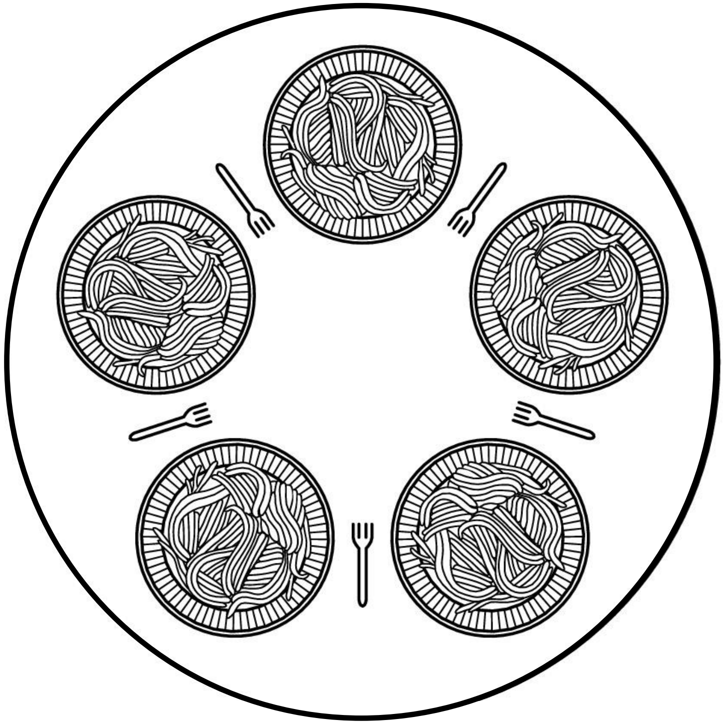
Alternative Producer-Consumer Solution Using OS/161 CVs

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

```
con() {
 while(TRUE) {
        lock_acquire(l)
        while (count == 0)
            cv_wait(empty,l);
        item = remove_item();
        count--;
        cv_signal(full,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