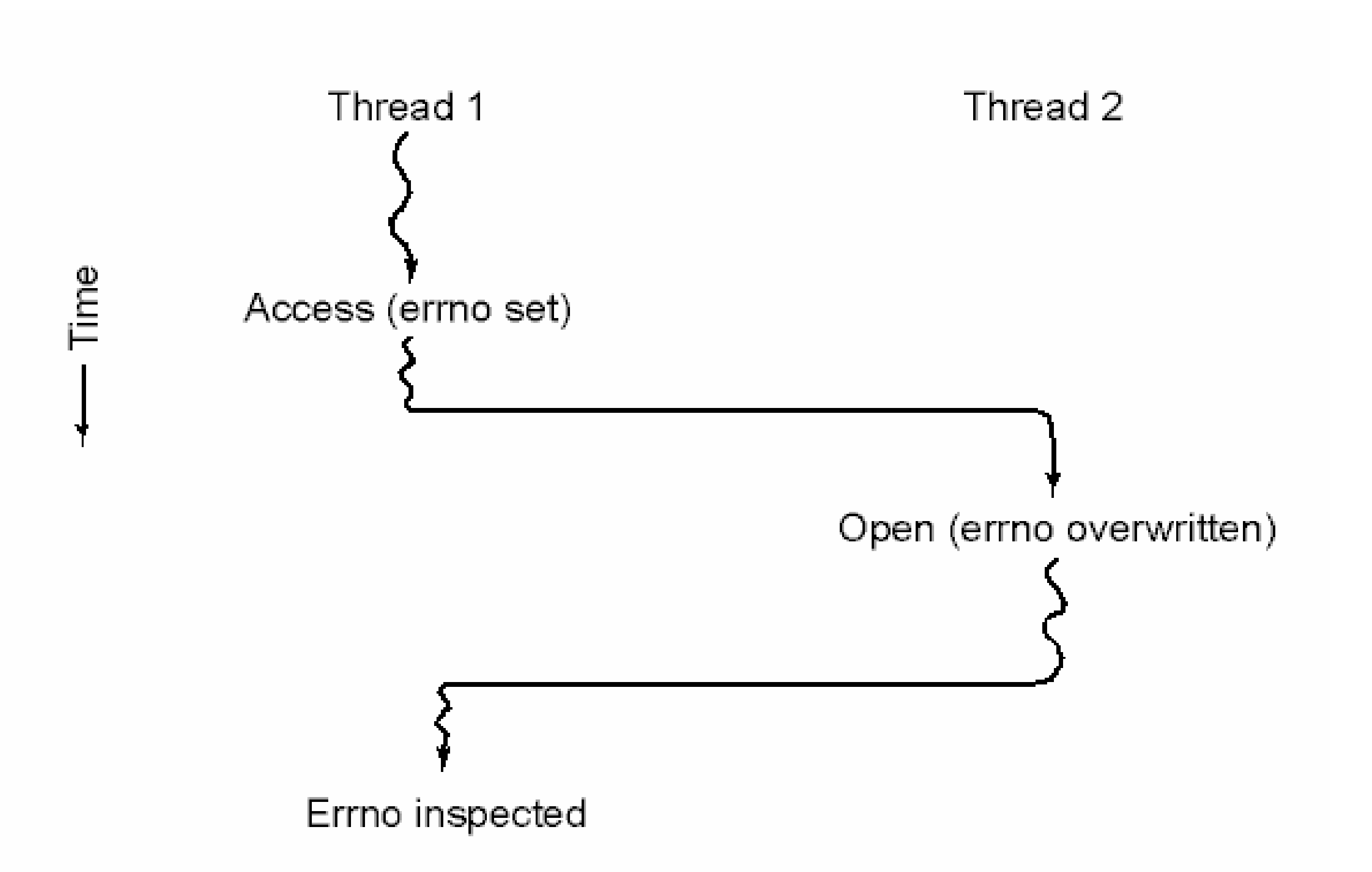
# Concurrency and Synchronisation

# Textbook

- Sections 2.3 & 2.4

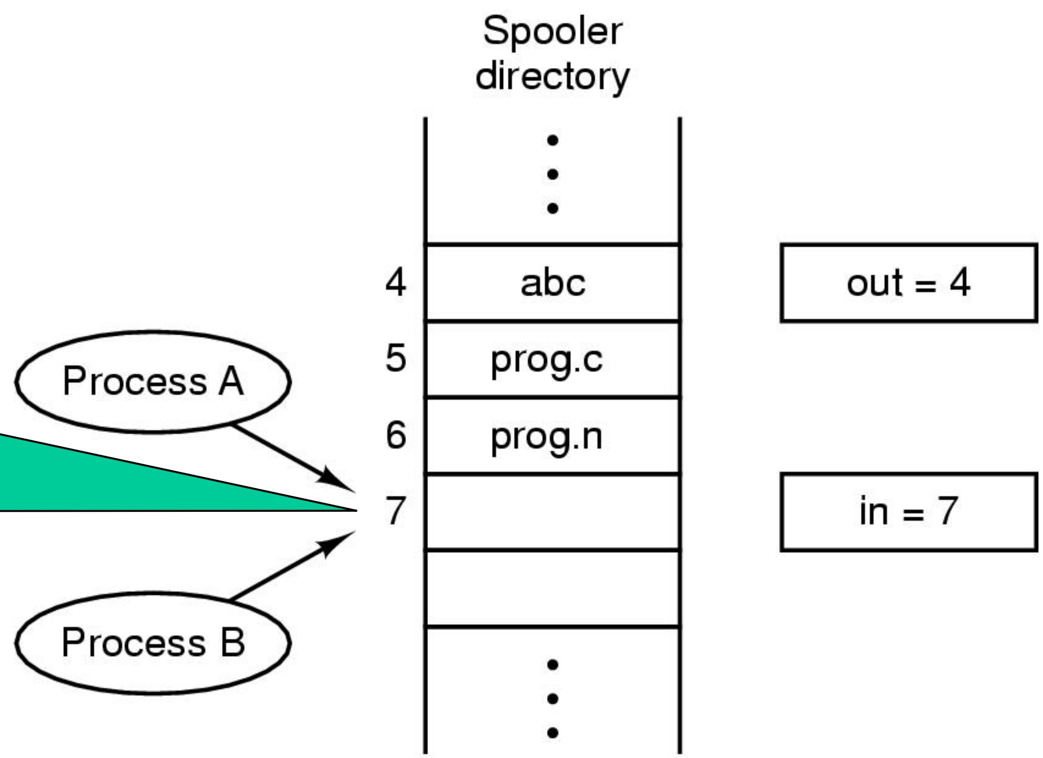
# Making Single-Threaded Code Multithreaded




Conflicts between threads over the use of a global variable

# Inter- Thread and Process Communication

We have a *race condition*




Two processes want to access shared memory at same time

# Critical Region

- We can control access to the shared resource by controlling access to the code that accesses the resource.

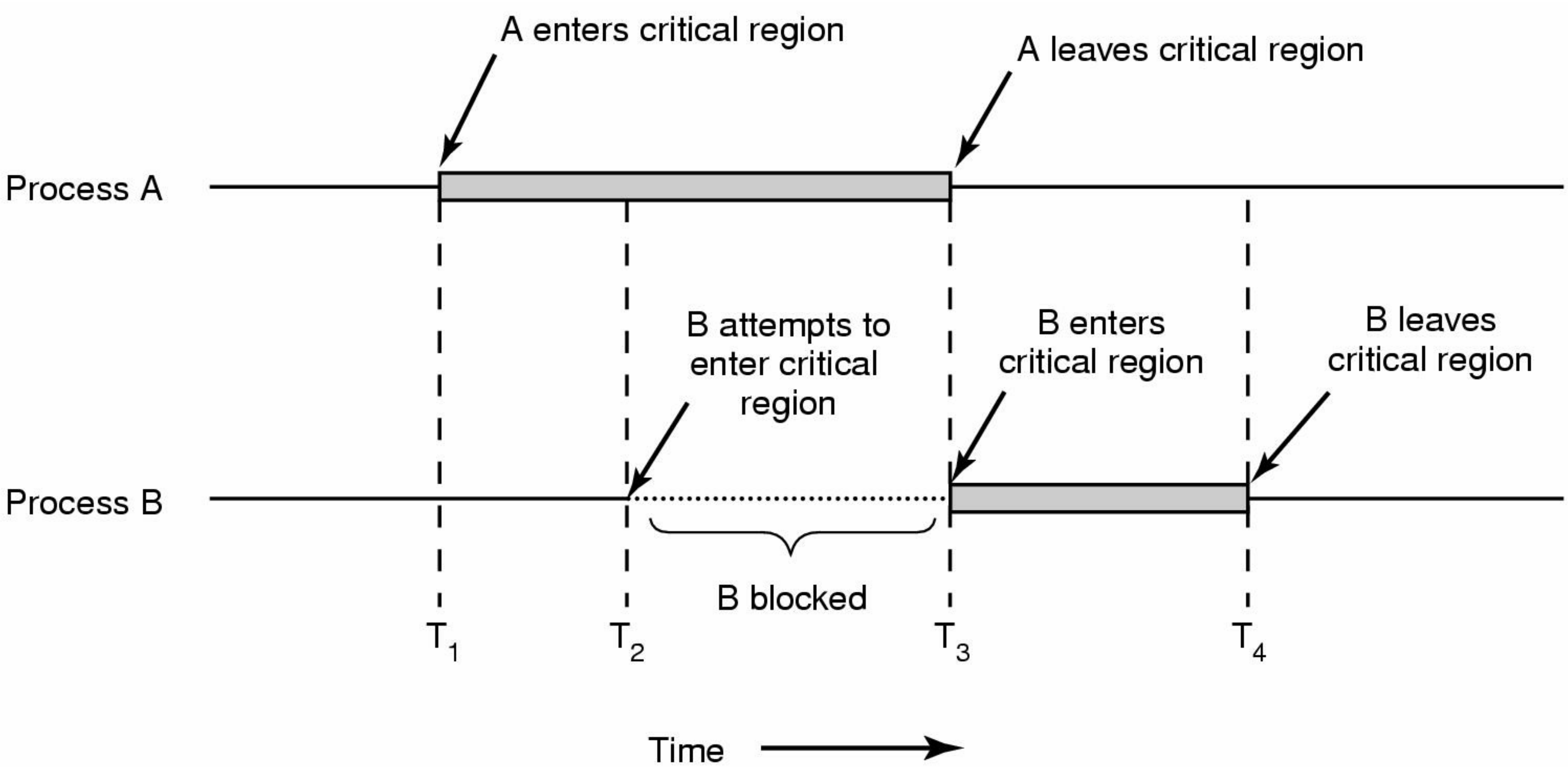
$\Rightarrow$ A *critical region* is a region of code where shared resources are accessed.

  – Variables, memory, files, etc…

- Uncoordinated entry to the critical region results in a race condition

  $\Rightarrow$ Incorrect behaviour, deadlock, lost work,…

# Critical Regions




Mutual exclusion using critical regions

# Critical Regions

Also called *critical sections*

Conditions required of any solution to the critical region problem

- Mutual Exclusion:
  - No two processes simultaneously in critical region
- No assumptions made about speeds or numbers of CPUs
- Progress
  - No process running outside its critical region may block another process
- Bounded
  - No process must wait forever to enter its critical region

# A non-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 non-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();
}
```

# A problematic execution sequence

```
while(TRUE) {

  while(lock == 1);
  lock = 1;

  critical();
  lock = 0
  non_critical();
}
```

```
while(TRUE) {
  while(lock == 1);


  lock = 1;
  critical();


  lock = 0
  non_critical();
}
```

# 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.

# Mutual Exclusion by Taking Turns

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Proposed solution to critical region problem
(a) Process 0. (b) Process 1.

# Mutual Exclusion by Taking Turns

- Works due to *strict alternation*
  - Each process takes turns
- Cons
  - Busy waiting
  - Process must wait its turn even while the other process is doing something else.
    - With many processes, must wait for everyone to have a turn
      - 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

- See the textbook

# 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

# Hardware Support for mutual exclusion

- Test and set instruction
  - Can be used to implement lock variables correctly
    - It loads the value of the lock
    - If lock == 0,
      - set the lock to 1
      - return the result 0 – we acquire the lock
    - If lock == 1
      - return 1 – another thread/process has the lock
  - Hardware guarantees that the instruction executes atomically.
    - Atomically: As an indivisible unit.

# Mutual Exclusion with Test-and-Set

```
enter_region:
      TSL REGISTER,LOCK             | copy lock to register and set lock to 1
      CMP REGISTER,#0               | was lock zero?
      JNE enter_region              | if it was non zero, lock was set, so loop
      RET | return to caller; critical region entered

leave_region:
      MOVE LOCK,#0                  | store a 0 in lock
      RET | return to caller
```

Entering and leaving a critical region using the TSL instruction

THE UNIVERSITY OF NEW SOUTH WALES

# Test-and-Set

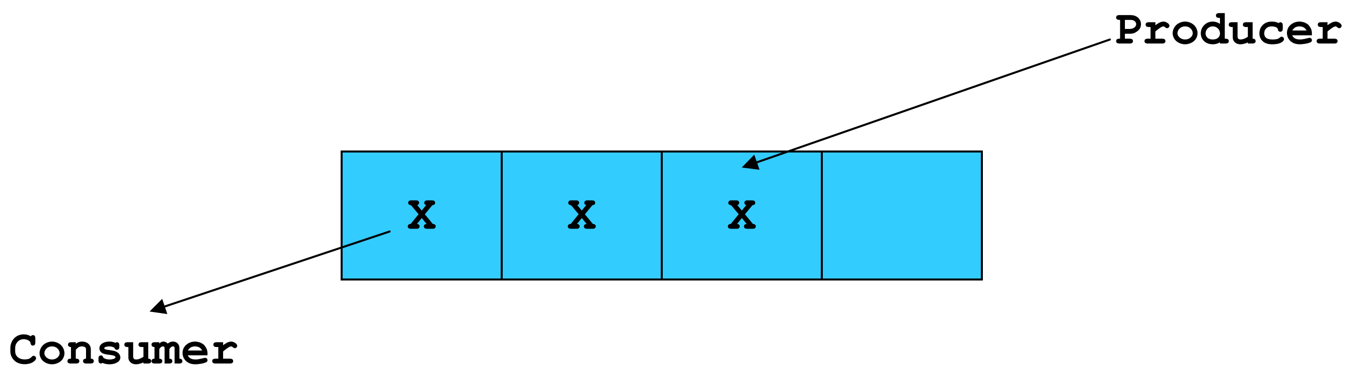
- Pros
  - Simple (easy to show it's correct)
  - Available at user-level
    - To any number of processors
    - To implement any number of lock variables
- Cons
  - Busy waits (also termed a *spin lock)*
    - Consumes CPU
    - Livelock in the presence of priorities
      - If a low priority process has the lock and a high priority process attempts to get it, the high priority process will busy-wait forever.
    - 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 the event happens, the event generator (another process) calls wakeup to unblock the sleeping process.

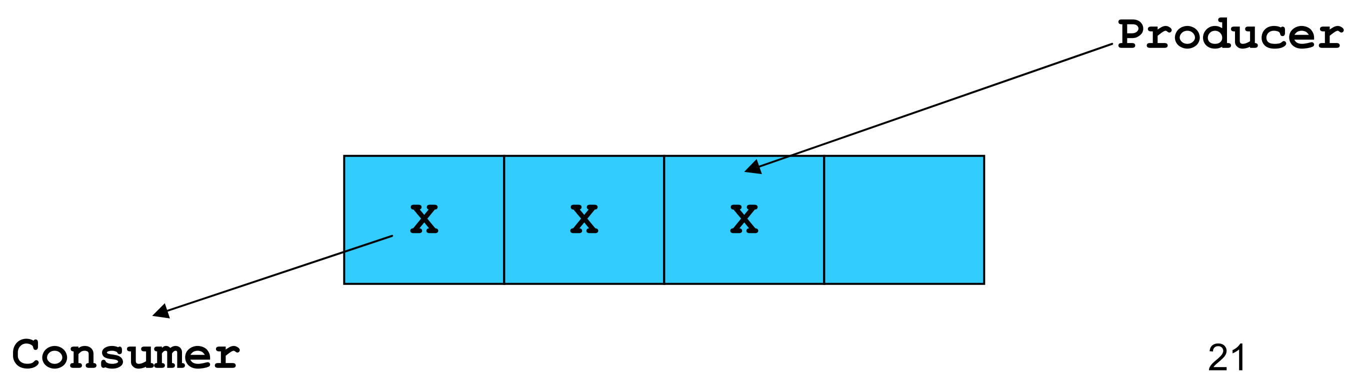
# The Producer-Consumer Problem

- 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);
   }
}
```

# Problems

```
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);
   }
}
```

Concurrent uncontrolled access to the buffer

# Problems

```
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);
   }
}
```

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

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

wakeup without a matching sleep is lost

THE UNIVERSITY OF NEW SOUTH WALES

# Problem

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

- The following does not work

```
acquire_lock()
if (count == N)
               sleep();
release_lock()
```

The lock is held while asleep ⇒ count will never change

# 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
- Wait and signal 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):
        S.count--;
        if (S.count < 0) {
                add this process to S.L;
                sleep;
        }

signal(S):
        S.count++;
        if (S.count <= 0) {
                remove a process P from S.L;
                wakeup(P);
        }
```

- Each primitive is atomic

# Semaphore as a General Synchronization Tool

- Execute *B* in $P_j$ only after *A* executed in $P_i$
- Use semaphore *count* initialized to 0
- Code:

| $P_i$ | $P_j$ |
|---|---|
| ⋮ | ⋮ |
| *A* | *wait*(*flag*) |
| *signal*(*flag*) | *B* |

# Semaphore Implementation of a Mutex

- Mutex is short for Mutual Exclusion
  - Can also be called a lock

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

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

Blahblah();

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

Notice that the initial count determines how many waits can progress before blocking and requiring a signal $\Rightarrow$ mutex.count initialised as 1

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