Synchronization

Access to the same variable

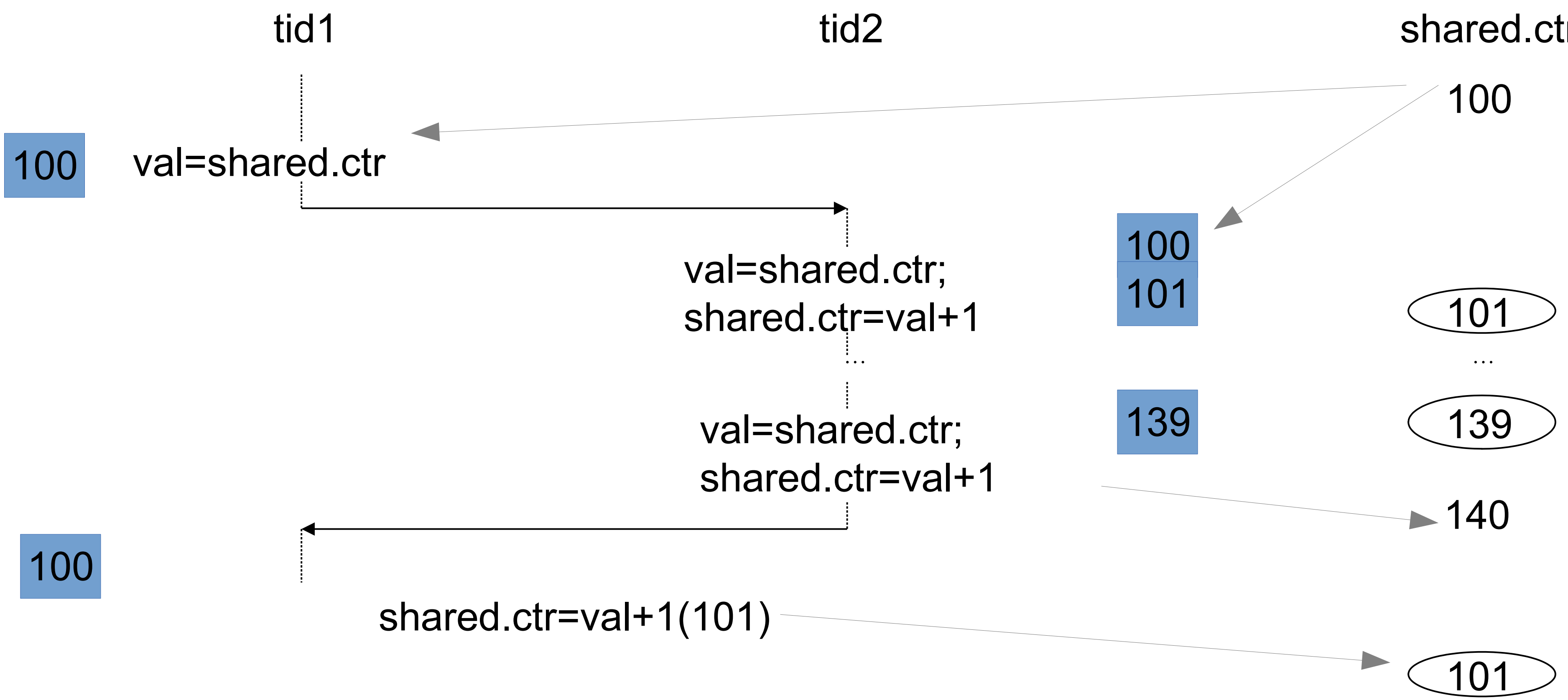
- Multiprogramming in OS allows different processes or threads to access common data
  - Read
  - write
- The order by which tasks are schedule can produce multiple results
  - Multiple accesses to the shared variable
  - Multiple accesses to the shared data structure
  - Multiple accesses to a device
    - Solved by the kernel :)
  - Multiple accesses to the same file
    - Depends on the granularity

Access to the same variable

```
int shared;
                void *count(void *arg) {
                  int i,val;
                  long this= (long)pthread_self();
                  for (i=0; i<NITERS; i++) {
                     val=shared;
                     printf("%lu: %d\n“, this, val);
                     shared=val+1;
                   }
                  return NULL;
                }
```

Race condition

- The xecution of a thread can be interrupted
  - between
    - val=shared.ctr; and shared.ctr=val+1;



Access to the same variable

```
int shared;
                void *count(void *arg) {
                  int i,val;
                  long this= (long)pthread_self();
                  for (i=0; i<NITERS; i++) {

                     printf("%lu: %d\n“, this,
                         shared ++);

                   }
                  return NULL;
                }
```

Access to the same variable

```
int shared;
                void *count(void *arg) {
                    int i,val;
                    long this= (long)pthread_self();
                    for (i=0; i<NITERS; i++) {
                        Shared++;


                     }
                     printf("%lu: %d\n“, this,shared);
                    return NULL;
                }
```

Access to the same variable

- `int shared;`
-
  - `void *count(void *arg) {`
  - ` int i,val;`
  - ` long this= (long)pthread_self();`
  - ` for (i=0; i<NITERS; i++) {`
  - `   printf("%lu: %d\n“, this, val);`
  - `   shared ++;`
  - `  }`
  - `  return NULL;`
  - `}`

Access to the same record

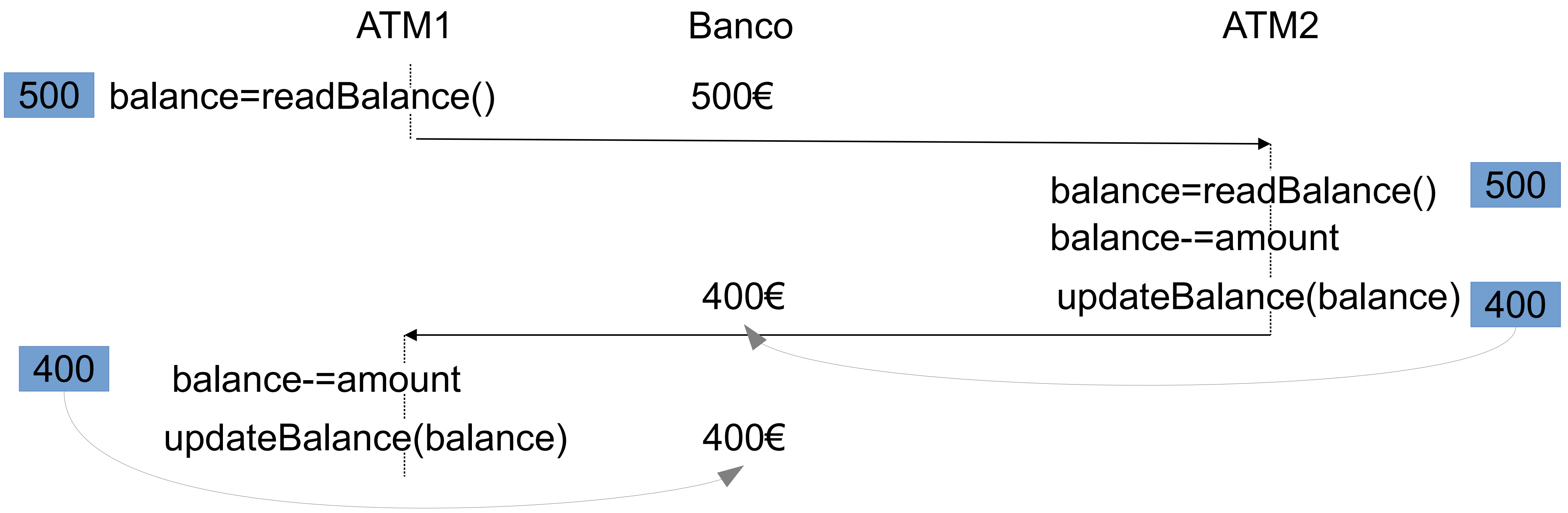
- Processe communicate using shared files
  - Two account management functions.

```
int deposit(int account, int amount) {
   balance = readBalance(account);
   balance += amount;
   updateBalance(account, balance);
   return balance;
}
```

```
int withdraw(int account, int amount) {
   balance = readBalance(account);
   balance -= amount;
   updateBalance(account,  balance);
   return balance;
}
```

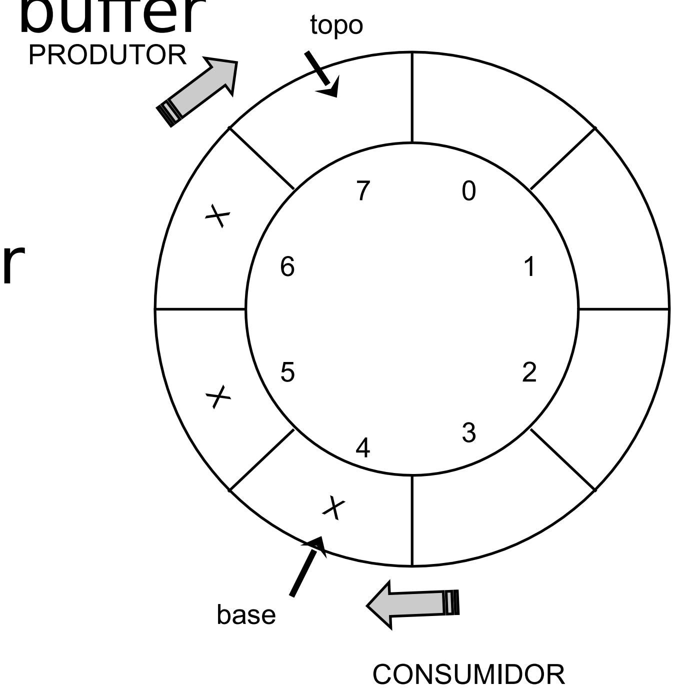
Access to the same variable

- Tow concurrent withdrawals
  - Initial value 500€,
  - Withdrawal of 100€



Access to the shared data-structure

- Producer-consumer also present race conditions
  - Two tasks share a common buffer.
    - Producer and consumer
    - The producer write data into the buffer
      - If not full
      - Blocks when buffer is full
    - Consumer reads data from buffer
      - If not empty
      - Blocks if no data is available



Access to the shared data-structure

Producer

```
while(1) {
 x = create_data()
 while(counter==SIZE){ }
   /*buffer cheio*/
 buffer[topo]=X;
 counter++;
 topo=(topo+1)%SIZE;
}
```

Consumer

```
While(1) {
  while(counter==0){ }
   /*buffer vazio*/

   Y=buffer[base];
   counter--;
   base=(base+1)%SIZE;
   consume_data(y)
}
```

1 producer + 1 consumer

Producer

```
while(1) {
  x = create_data()
  while(counter==SIZE){ }
    /*buffer cheio*/
  buffer[topo]=X;
  counter++;
  topo=(topo+1)%SIZE;
}
```

MOV %EAX,counter

INC %EAX

MOV counter,%EAX

Consumer

```
While(1) {
  while(counter==0){ }
    /*buffer vazio*/

    Y=buffer[base];
    counter--;
    base=(base+1)%SIZE;
    consume_data(y)
}
```

MOV %EAX,counter

DEC %EAX

MOV counter,%EAX\

1 producer + 1 consumer

- Example of race conditions
  - with counter==4

Entidade	INstrução	Registo	Counter
produtor	MOV %EAX,counter	EAX ← 4	4
produtor	INC %EAX	EAX ← 5	4
consumidor	MOV %EAX,counter	EAX ← 4	4
consumidor	DEC %EAX	EAX ← 3	4
produtor	MOV counter, %EAX	EAX ← 5	5
consumidor	MOV counter, %EAX	EAX ← 3	3

1 producer + 1 consumer

- Producer
  - `while(1) {`
  - `/* gera dado X */`
  - `while(counter==SIZE)`
  - `/*buffer cheio*/ ;`
  - `buffer[topo]=X;`
  - **`counter++;`**
  - `topo=(topo+1)%SIZE;`
  - `}`
- Consumer
  - `While(1) {`
  - `/* consome dados */`
  - `while(counter==0)`
  - `/*buffer vazio*/ ;`
  - `Y=buffer[base];`
  -
  - **`counter--;`**
  - `base=(base+1)%SIZE;`
  - `}`

1 producer + 1 consumer

- Solution to race condition
  - with counter==4
- Similar solution solves multiple consumers

Entidade	INstrução	Registo	Counter
produtor	MOV %EAX,counter	EAX ← 4	4
produtor	INC %EAX	EAX ← 5	4
produtor	MOV counter, %EAX	EAX ← 5	5
consumidor	MOV %EAX,counter	EAX ← 4	5
consumidor	DEC %EAX	EAX ← 3	5
consumidor	MOV counter, %EAX	EAX ← 3	4

1 producer + 1 producer

- Producer
  - while(1) {
  - /* gera dado X */
  - while(counter==SIZE)
  -    /*buffer cheio*/ ;
  - buffer[topo]=X;
  - counter++;
  - topo=(topo+1)%SIZE;
  - }

- Producer
  - while(1) {
  - /* gera dado X */
  - while(counter==SIZE)
  -    /*buffer cheio*/ ;
  -
  -
  -
  - buffer[topo]=X;
  - counter++;
  - topo=(topo+1)%SIZE;
  - }

1 producer + 2 consumer

- Producer
  - while(1) {
  -
  - while(counter==SIZE)
  - /*buffer cheio*/ ;
  - buffer[topo]=X;
  - counter++;
  - topo=(topo+1)%SIZE;
  - }
- Consumer
  - While(1) {
  -
  - while(counter==0)
  - /*buffer vazio*/ ;
  - Y=buffer[base];
  - counter--;
  - base=(base+1)%SIZE;
  - }

Producer / consumer

- Active wait uses CPU cicles
  - OS should interrupt task
  -
-
- The consumer should be awaken when
  - Producer inserts value in memory
- The producer should be awaken when
  - Consumer remove a value from memory

Producer-consumidor

- Producer
  - `while(1) {`
  - `    /* gera dado X */`
  - `    if (counter==SIZE)`
  - `       pause() ;`
  - `    buffer[topo]=X;`
  - `    counter++;`
  - `    if (counter==1)`
  - `       signal(consumer);`
  - `    topo=(topo+1)%SIZE;`
  - `}`
- Consumer
  - `While(1) {`
  -
  - `    if (counter==0)`
  - `       pause() ;`
  - `    Y=buffer[base];`
  -
  - `    counter--;`
  - `    if (counter==N-1)`
  - `       signal(producer);`
  - `    base=(base+1)%SIZE;`
  - `}`

Pause signal considerations

- The pause and signal actions also show race conditions
- Example:
  - The consumer reads the last value in memory
  - Since counter == 0
    - The consumer is paused by the OS (before the pause())
    - Producer starts working
  - Producer inserts an element
  - Since counter==1,
    - executes **signal(consumer)**
    - That has no effect since the consumer has not executed the **pause().**
  - Consumer resumes executions
    - Gets blocked in the **pause()**
  - Producer continues producing
    - Filling the buffer
    - Not notifying the consumer

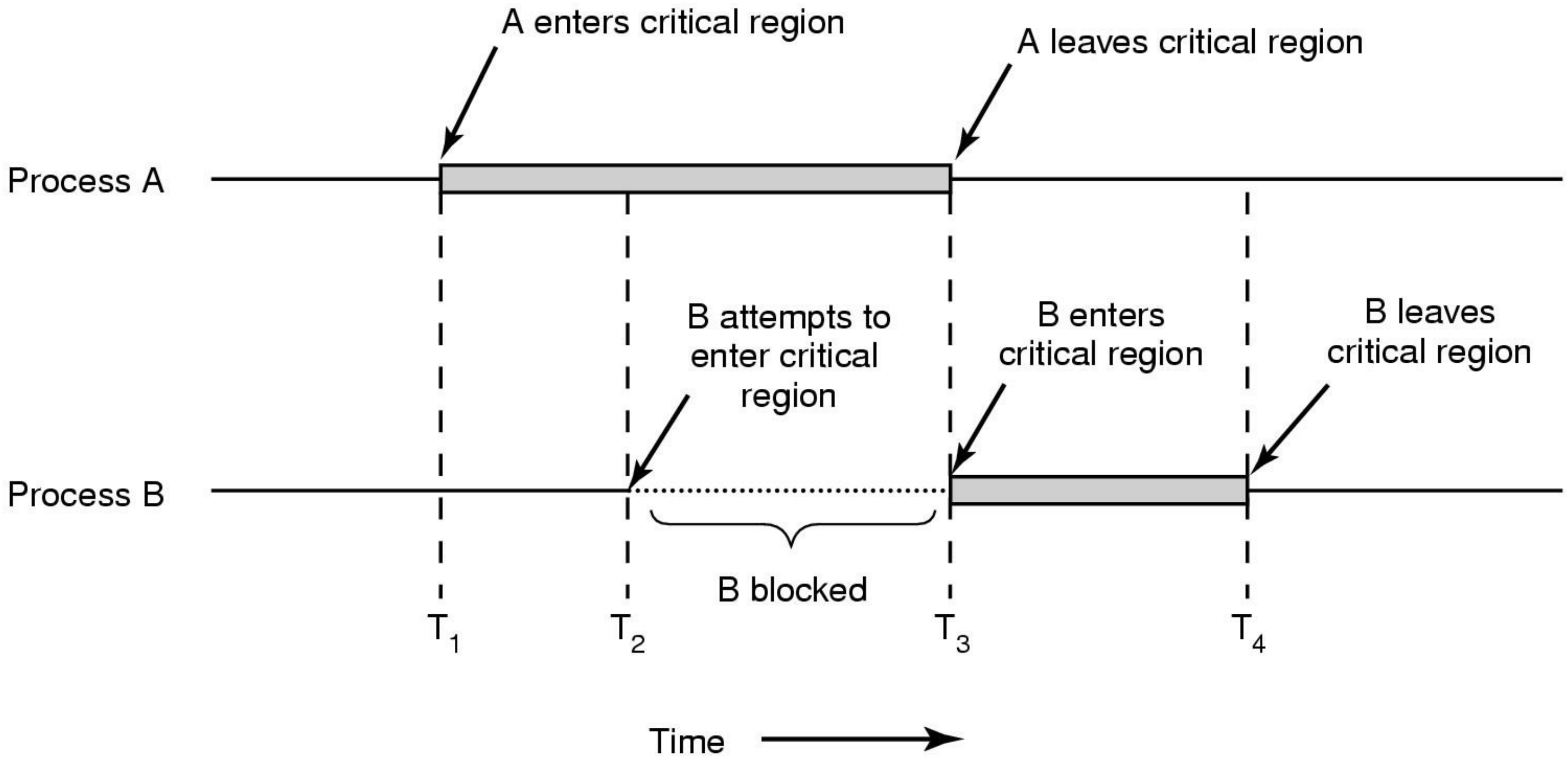
Race condition

- “race condition”
  - Occurs when the result of an execution on shared data
    - depends on the order of the interleaving of instructions.
- A race occurs when two threads can access (read or write) a data variable simultaneously and at least one of the two accesses is a write.
  - (Henzinger 04)
- Results from the fact that the result depends on the last task to conclude.
- To avoid race conditions,
  - Tasks should be synchronized
- Synchronization forces the order of events
  - Special functions
  - communication

Critical region

- Piece of code where resources are shared
  - That can only be executed by one tasks at a time
- Is delimited by read/write instructions to the shared resource
-
- Concurrent reads do not produce inconsistent/incoherent results
- If a task is inside a critical region
  - Other task trying to enter should be blocked

Critical region



Mutual exclusion

- Mechanism that assures that only one taks (at most) is inside a critical region
  - No more that one taks is execution the critical region code
  - All other tasks are
    - Runnig non critical code
    - blocked
- Consequences of mutual exclusion
  - starvation (live-lock)
    - A task is able to be executed, but is never scheduled
  - deadlock
    - Due to coding problme, several tasks are waiting for being unblocked by other tasks that are in the same state

Mutual exclusion

- Existing solutions
  - Active wait algorithms (Peterson, Lamport)
  - Hardware
    - Processor special instructions
  - Synchronization objects
    - semaphores, mutexes
  - Using OS services
    - monitors, message passing

Critical regions

- Requirements to be satisfied
  - Mutual exclusion
    - Only one task can be inside the Critical region
  - Progress
    - A task inside the Critical region can not block other task from enetring.
  - Limited wait
    - A task should wait a limited amount of time before entering
- A task should remain inside the Critical Region for a limited time
- The speed and number of processor is undefined
  - Should work for any combination

Critical region programming

```
do {
/* secure zone */
    EnterRegion()
    Critical region
Process(shared_data)
    LeaveRegion()
    Remaining code
/* secure zone */
} while(X);\
```

- Entry Region
  - Taks request permission to enter
  - Task get blocked
- Out region
  - Task signal exit of region
  - To allow other tasks to enter
- Remaining code
  - Code that does not access shared data

Mutual exclusion?

- `void *count(void *arg) {`
- ` int i,val;`
- ` long this= (long)pthread_self();`
- ` for (i=0; i<NITERS; i++) {`
- `   printf("%lu: %d\n“, this, val);`
- `   shared ++;`
- `  }`
- `  return NULL;`
- `}`

Mutual exclusion?

- Processe communicate using shared files
  - Two account management functions.

```
int deposit(int account, int amount) {
   balance = readBalance(account);
   balance += amount;
   updateBalance(account, balance);
   return balance;
}
```

```
int withdraw(int account, int amount) {
   balance = readBalance(account);
   balance -= amount;
   updateBalance(account,  balance);
   return balance;
}
```

Mutual exclusion?

- Producer
  - while(1) {
  - /* gera dado X */
  - while(counter==SIZE)
  - /*buffer cheio*/ ;
  - buffer[topo]=X;
  - counter++;
  - topo=(topo+1)%SIZE;
  - }
- Consumer
  - While(1) {
  - /* consome dados */
  - while(counter==0)
  - /*buffer vazio*/ ;
  - Y=buffer[base];
  -
  - counter--;
  - base=(base+1)%SIZE;
  - }

POSIX Synchronization primitives

- Synchronization variables
  - Mutexes ✓
  - Spinlocks
  - Read/write locks
  - Semaphores ✓
  - Conditional variables ✓
- Synchronization mechanics
  - Critical region implementations ✓
    - Using synchronization variable
    - Using messages
  - Rendez-vous ✓
  - Barriers ✓
  - Monitors
    - e.g. Java

Mutexes definition

- variable that can only have two values and and list of waiting tasks
  - locked and unlocked
- In linux
  - Introduced in Kernel 2.6.16, corresponding to binary semaphores (0-locked, 1-unlocked).
- Lighter than semaphores
  - semaphore takes 28 bytes vs 16bytes for mutex
  - faster
- A locked mutex is owned by a single task
  - only the task that locked it can unlock
- tasks that try to lock a mutex are stored in the list
    - order of unlock depended on the scheduling policies
- mutex ::= MUTual EXclusion

Mutex definition

- Mutex usage
  - Creation and initialization
  - several tasks execute Entry region
    - try to lock the mutex
  - only on continues
    - becomes the owner of the mutex
    - the owner executes the critical region
    - the owner enters the exit region and unlocks the mutex
  - Other task waiting in the entry region locks the mutex and becomes owner
  - ...
- Mutex is destroyed

POSIX Mutexes

- POSIX mutexes are associtaed to pthreads:
  - include file #include <pthread.h>
  - data type: pthread_mutex_t mux;
- A mutex should be initialized before used
- int pthread_mutex_init(pthread_mutex_t *restrict mutex,
- const pthread_mutexattr_t *restrict attr);
  - mutex - pointer to variable
  - attr - mutex attributes
    - PTHREAD_MUTEX_NORMAL PTHREAD_MUTEX_ERRORCHECK PTHREAD_MUTEX_RECURSIVE PTHREAD_MUTEX_DEFAULT
- mux=PTHREAD_MUTEX_INITIALIZER;
  - static default creation parameters (attr = NULL)
  - more efficient

POSIX Mutexes

- A mutex is destroyed by:
- **`int pthread_mutex_destroy( pthread_mutex_t *);`**

  - `int error;`
  - `pthread_mutex_t mux;`
  -
  - `if (error=`**`pthread_mutex_init`**`(&mux,NULL))`
    - `perror(“mutex_init: ”)`
  - ` /* … */`
  - `if (error=pthread_mutex_destroy(&mux))`
  - `   perror(mutex_destroy: “) ;`

POSIX Mutexes

- mutex locking
- **`int pthread_mutex_lock(pthread_mutex_t *mutex);`**
  - block thread until resource is available.
  - returns when task enters critical region
- **`int pthread_mutex_trylock(pthread_mutex_t *mutex);`**
  - retorna imediatamente.
- both functions return 0 in success
- pthread_mutex_trylock
  - if thread can not lock
    - error variable equals EBUSY
- Mutexes should be kept locked for the minimum amount of time

POSIX Mutexes

- Mutext unlock
- **`int pthread_mutex_unlock(pthread_mutex_t *mutex);`**
- Critical region can be guaranteed by
  - `pthread_mutex_t mux=PTHREAD_MUTEX_INITIALZER;`
  - `do{`
  - `   pthread_mutex_lock( &mux );   /* RE */`
  - `      /* RC */`
  - `   pthread_mutex_unlock( &mux ); /* RS */`
  - `   /* RR */`
  - `while(TRUE);`

POSIX Mutexes

- Mutexes attributes
  - pthread_mutexattr_ *
- protocol
  - PTHREAD_PRIO_INHERIT
    - thrd1 priority and scheduling are affected when higher-priority threads block on one or more mutexes owned by thrd1
  - PTHREAD_PRIO_NONE
    - A thread's priority and scheduling are not affected by the mutex ownership.
  - PTHREAD_PRIO_PROTECT
    - thrd2 priority and scheduling is affected when the thread owns one or more mutexes
- Shared
  - boolean - allow mutex to be shared among threads from more than one process

POSIX Mutexes

- Mutexes robusteness
- PTHREAD_MUTEX_STALLED
  - No special actions are taken if the owner of the mutex is terminated while holding the mutex lock.
  - This is the default value.
- PTHREAD_MUTEX_ROBUST
  - If the process containing the owning thread of a robust mutex terminates while holding the mutex lock,
    - the next thread that acquires the mutex shall be notified about the termination by the return value

POSIX Mutexes

- Mutexes type
- PTHREAD_MUTEX_NORMAL
  - This type of mutex does not detect deadlock.
  - A thread attempting to relock this mutex without first unlocking it will deadlock.
- PTHREAD_MUTEX_ERRORCHECK
  - This type of mutex provides error checking.
  - A thread attempting to relock this mutex without first unlocking it will return with an error.
- PTHREAD_MUTEX_RECURSIVE
  - A thread attempting to relock this mutex without first unlocking it will succeed in locking the mutex.
  - Multiple locks of this mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex.
- PTHREAD_MUTEX_DEFAULT
  - Attempting to recursively lock a mutex of this type results in undefined behaviour.

POSIX Mutexes

Mutex Type	Robustness	Relock	Unlock When Not Owner
NORMAL	non-robust	deadlock	undefined behavior
NORMAL	robust	deadlock	error returned
ERRORCHECK	either	error returned	error returned
RECURSIVE	either	recursive	error returned
DEFAULT	non-robust	undefined behavior	undefined behavior
DEFAULT	robust	undefined behavior	error returned

Spin Locks

Spin Locks

- Spin locks are a low-level synchronization mechanism
  - suitable for shared memory multiprocessors.
- When the calling thread requests a spin lock that is already held by another thread
  - the second thread spins in a loop to test if the lock has become available.
- When the lock is obtained,
  - it should be held only for a short time,
  - as the spinning wastes processor cycles.
- Callers should unlock spin locks before calling sleep operations to enable other threads to obtain the lock.

Spin Locks

- initialization
  - int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
- locking
  - int pthread_spin_lock(pthread_spinlock_t *lock);
  - int pthread_spin_trylock(pthread_spinlock_t *lock);
- unlocking
  - int pthread_spin_unlock(pthread_spinlock_t *lock);
- Destroying
  - int pthread_spin_destroy(pthread_spinlock_t *lock);

Read-Write Locks

Read-Write Lock

- Read-write locks permit concurrent reads and exclusive writes to a protected shared resource.
- The read-write lock is a single entity that can be locked in read or write mode.
- To modify a resource, a thread must first acquire the exclusive write lock.
  - An exclusive write lock is not permitted until all read locks have been released.
-
- Read-write locks support concurrent reads of data structures
  - read operation does not change the record's information.
- When the data is to be updated
  - the write operation must acquire an exclusive write lock.

Not fair

- Th1 ReadLock (enter)
- Th2 ReadLock (enter)
- Th3 WriteLock (blocked)
- Th2 unlock (leave)
- Th4 ReadLock (enter)
- Th1 unlock (leave)
- Th2 ReadLock (enter)
-
- Th3 is in starvation

- Th1 ReadLock (enter)
- Th2 ReadLock (enter)
- Th3 WriteLock (blocked)
- Th2 unlock (leave)
- Th4 ReadLock (blocked)
- Th1 unlock (leave)
  - Th3 enters
- Th2 ReadLock (blocked)

Read-Write Lock

- Initialization
  - int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
  - const pthread_rwlockattr_t *restrict attr);
    - rwlock will contain the reference to the lock
    - attr can be NULL
- Destruction
  - int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
- locking
  - int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
  - int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
- unlocking
  - int pthread_rwlock_unlock (pthread_rwlock_t *rwlock);

Semaphores

- Abstraction for a counter and a task list, used for synchronization purposes.
  - Proposed by Dijsktra in 1965.
  - Do not require active wait.
  - All modern OS include a version of semaphores (Unix, Windows, …)
- typedef struct{
- unsigned counter;
- processList *queue;
- }sem_t;

Semaphores

- S.counter
  - Defines how many tasks can pass the semaphore withpout blocking
- If s.coutner = 0
  - The next task to try to enter will get blocked
- If s.coutner is always <= 1
  - Mutual exclusion is guaranteed
  - The length of the queue depends on the number of tasks waiting to enter
-

Semaphores

- down(S)
  - Used by a tasks when trying to access a resource
    - Access is given by another task issuing up(S).
  - Task can get blocked if capacity is full (counter == 0)
- up(S)
  - Used by a task to signal the resource availability
  - up(S) is not blocking

Semaphores

- Critical region
  - Initialize a semaphore with counter = 1
  - Do a down when entering the critical region
  - Do a Up when leaving the critical region

- Rendezvous
  - Initialize a semaphore with counter = 0
  - The tasks that does down(S) will get blocked
    - Until other task does the UP(S)
  - Two tasks reandez-vous and continued together

Semaphores

- Wait(S), down(S), or P(S)
  - if (S.counter>0)
    - S.counter--;
  - else
    - Block(S); /* insert S into the queue */
- Signal(S), up(S), or V(S)
  - if (S.queue!=NULL)
    - WakeUp(S); /* removes a processfrom queue */
  - else
    - S.counter++;
- Wakeup and block depend on the OS
- P(S) e V(S) come from dutch words prolaag (decrement) and verhoog (incrementar)

POSIX Semaphores

- A semaphore is an integer whose value is never allowed to fall below zero.

- POSIX includes a set of functions
  - Man sem_overview
  - #include <semaphore.h>
  - Link program with -lpthread

POSIX Semaphores

- Two operations can be performed on semaphores:
  - increment the semaphore value by one (sem_post(3));
  - and decrement the semaphore value by one (sem_wait(3)).
- If the value of a semaphore is currently zero, then a sem_wait(3) operation will block until the value becomes greater than zero.

POSIX Semaphores

- POSIX offer two forms of semaphores with respect to creation
  - Named semaphores
    - Identified by a global name (null terminated string started with /
    - Multiple processes can operate on the same named semaphore by passing the same name to sem_open
  - Unamed (memory-based semaphores)
    - Created in memory shared by multiple threads or processes

POSIX Semaphores

- Posix offers two sharing mechanism
  - Not shared
    - Just threads of the process
  - Shared
    - Unnamed - Shared by threads in related processes parent children or using shared memory
    - Named - Shared by threads in multiple processes
  - Semaphores are also classified by the maximum number of tasks (N) that can access the resources
  - If N equals 1 => binary semaphore

POSIX Semaphores

Named		unnamed
sem_open()		sem_init()
	sem_wait() sem_trywait() sem_post() sem_getvalue()	
sem_close() sem_unlink()		sem_destroy()

Service	**POSIX**
up	sem_post
down	sem_wait

POSIX unnamed Semaphores

- An unnamed semaphore is a variable of type **sem_t**
  - #include <semaphore.h>
  - sem_t sem;
- Should be initialized before being used by
  - Child processes,
  - threads.
- **int sem_init(sem_t *sem, int pshared, unsigned int value**
  - sem_init() initializes the unnamed semaphore at the address pointed to by **sem**.
  - The **pshared** argument indicates whether this semaphore is to be shared between the threads of a process, or between processes.
  - The **value** argument specifies the initial value for the semaphore.

POSIX unnamed Semaphores

- Unused semaphred should be destroyed
  - int sem_destroy(sem_t *);
-
- `sem_t semaforo;`
- `if (sem_init(&semaforo,0,1)==-1)`
- `   perror("Falha na inicializacao");`
-
- `if (sem_destroy(&semaforo)==-1)`
- `   perror("Falha na eliminacao");`

POSIX named Semaphores

- Allows the synchronization of processes without shared memory
  - Use a global name
    - string following the form **/name**
- Named semaphores are installed/located in /dev/shm, with the name **sem.name**
- A semaphore should be opened before used

POSIX named Semaphores

- sem_t *sem_open(const char *name, int oflag)
- sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value)
  - sem_open() creates a new POSIX semaphore or opens an existing semaphore.
  - The semaphore is identified by name.
  - The oflag argument specifies flags that control the operation of the call.
    - If O_CREAT is specified in oflag, then the semaphore is created if it does not already exist.
- If O_CREAT is specified in **oflag**, two additional arguments must be supplied
  - **mode_t**, premitions (owner, group, user)
  - The value argument specifies the initial value for the new sem aphore..
- If **oflag** contains O_CREAT and O_EXCL
    - Error if semaphore already exists
- If **oflag** is O_CREAT and semaphore exists
    - 3rd and 4th parameters are ignored

POSIX named Semaphores

- When the semaphore is of no use should be closed
  - `int sem_close(sem_t *);`
- The last process should
  - Close the sempahore (sem_close)
  - Remove the correponding file
    - `int sem_unlink(const char *);`
- If a process maintains the semaphore opened
  - sem_unlink is blocked until the semaphore is closed
- If the semaphore is not closed/unlinked
  - New uses of the semaphore are undefined....

POSIX Semaphores

- down(S) is implemented by the function
  - `int sem_wait(sem_t *);`
  - If counter is zero
    - The thread executing the function is blocked.
      - Remaining thread in the process continue executing
- up(S) is implemenetd by
  - `int sem_post(sem_t *);`

Mutual exclusion

- sem_init(&sem, 0, 1)
  - Or
- Sem = sem_open(... , O_CREAT, ... , 1);

- `do {`
- `    sem_wait( &sem ); /* RE */`
- `      /* RC */`
- `    sem_post( &sem ); /* RS */`
- `      /* RR */`
- `} while (TRUE);`

Semaphores

- A thread can pool the value of a semaphore
  - `int sem_getvalue(sem_t *sem, int *sval);`
  - sem_getvalue() places the current value of the semaphore pointed to sem into the integer pointed to by sval.
- If one or more processes or threads are blocked waiting to lock the semaphore with sem_wait(3),
  - POSIX.1-2001 permits two possibilities for the value returned in sval:
    - either 0 is returned;
    - negative number whose absolute value is the count of the number of processes and threads currently blocked in sem_wait(3).
  - Linux adopts the former behavior (return 0)

POSIX Semaphores

- int sem_trywait(sem_t *sem);
  - sem_trywait() is the same as sem_wait(), except that if the decrement cannot be immediately performed, then call returns an error (errno set to EAGAIN) instead of blocking.
- int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
  - same as sem_wait(), except that **abs_timeout** specifies a limit on the amount of time that the call should block
  - If the timeout has already expired by the time of the call, then sem_timedwait() fails with a timeout error (errno set to ETIMEDOUT).

Condition Variables

- We need additional mechanisms to wait inside locked regions
- However, holding the lock while waiting prevents other threads from entering the locked region
- Condition variables make it possible to sleep inside a critical section
  - Atomically release the lock & go to sleep

Condition Variables

- Each condition variable
  - Consists of a queue of threads
  - Provides three operations
    - Wait();
      - Atomically release the lock and go to sleep
      - Reacquire lock on return
    - Signal();
      - Wake up one waiting thread, if any
    - Broadcast();
      - Wake up all waiting threads
- The three operations can only be used inside locked regions

An Example of Using Condition Variables

```
AddToQueue() {
   lock.Acquire();
   // put 1 item to the queue
   condition.Signal(&lock);
   lock.Release();
}

RemoveFromQueue() {
   lock.Acquire();
   while nothing on queue
       condition.Wait(&lock);
   lock.Release();
   return item;
}
```

Condition variables in pthread

- Waiting and signaling on condition variables
  - Data type: pthread_cond_t data_cond = PTHREAD_COND_INITIALIZER;
- Routines
  - pthread_cond_wait(condition, mutex)
    - Blocks the thread until the specific condition is signalled.
    - Should be called with mutex locked
    - Automatically release the mutex lock while it waits
    - When return (condition is signaled), mutex is locked again
  - pthread_cond_signal(condition)
    - Wake up a thread waiting on the condition variable.
    - Called after mutex is locked, and must unlock mutex after
  - pthread_cond_broadcast(condition)
    - Used when multiple threads blocked in the condition

Condition Variables

- While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.
- Without condition variables, The programmer would need to have threads continually polling (usually in a critical section), to check if the condition is met.
- A condition variable is a way to achieve the same goal without polling (a.k.a. “busy wait”)

Condition Variables

- Useful when a thread needs to wait for a certain condition to be true.
- In pthreads, there are four relevant procedures involving condition variables:
  - pthread_cond_init(pthread_cond_t *cv, NULL);
  - pthread_cond_destroy(pthread_cond_t *cv);
  - pthread_cond_wait(pthread_cond_t *cv, pthread_mutex_t *lock);
  - pthread_cond_signal(pthread_cond_t *cv);

Creating and Destroying Conditional Variables

- Condition variables must be declared with type pthread_cond_t, and must be initialized before they can be used.
  - Statically, when it is declared. For example:
    - pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;
  - Dynamically
    - pthread_cond_init(cond, attr);
- pthread_cond_destroy(cond)
  - used to free a condition variable that is no longer needed.

pthread_cond_init

- A condition variable is a pthread_cont_t

```
#include <pthread.h>
```

```
pthread_cond_t cond;
```

- Should be initialized before being used

```
int pthread_cond_init(pthread_cond_t *, const
pthread_condattr_t *);
```

  - 1º parâmetro: address of condition variable
  - 2º parâmetro: attributes (can be NULL)
- static initilaization
  - `cont=PTHREAD_COND_INITIALZER;`

pthread_cond_destroy

- termination

```
int pthread_cond_destroy( pthread_cond_t *);
```

- Condition variable usage:
  - Created and initialized
  - If condition is not satisfied block/ otherwise continue.
  - Other thread changes the condition and
    - signals other threads about the new condition values
    - one by one or broadcast
  - Signaled thread continues execution.
  -
  - Condition variable is terminated.

pthread_cond_wait

- pthread_cond_wait(cv, lock)
  - is called by a thread when it wants to block and wait for a condition to be true.
- It is assumed that the thread has locked the mutex indicated by the second parameter.
- The thread releases the mutex, and blocks until awakened by a pthread_cond_signal() call from another thread.
- When it is awakened,
  - it waits until it can acquire the mutex,
  - once acquired, it returns from the pthread_cond_wait() call.

pthread_cond_wait

- Waiting on a condition variable:

```
int pthread_cond_wait(pthread_cont_t *,
pthread_mutex_t *);
```

  - wait until other thread signals/broadcasts
  -

```
int pthread_cond_timedwait(pthread_cont_t *,
  pthread_mutex_t *,const struct timespec*);
```

  - Timed wait.

Condition variables

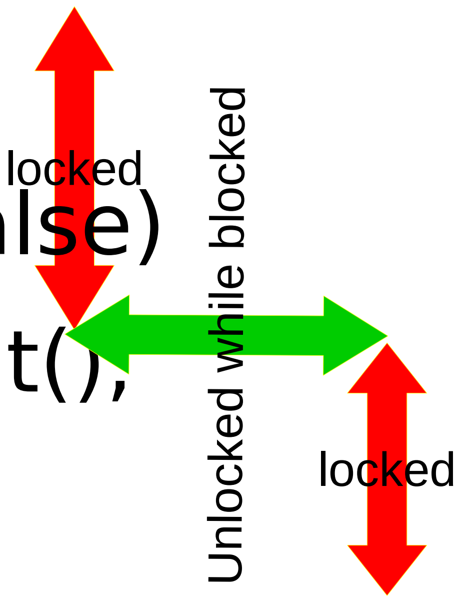
- **pthread_cond_wait**
- **pthread_cond_timedwait**
  - 1st parameter: condition variable
  - 2nd parameter: mutex that guards the critical region.
  - pthread_cond_timedwait
    - 3º parâmetro: função de temporização da espera
- pthread_cond_wait() e pthread_cond_timedwait()
  - should be called after a thread_mutex_lock().

pthread_cond_signal

- pthread_cond_signal()
  - checks to see if there are any threads waiting on the specified condition variable.
  - If not, then it simply returns.
- If there are threads waiting,
  - then one is awakened.
- There can be no assumption about the order in which threads
  - It is natural to assume that they will be awakened in the order in which they waited, but that may not be the case…
- Use loop or pthread_cond_broadcast() to awake all waiting threads.

Condition variables

- **`pthread_timedwait`**
- **`pthread_cond_timedwait`**
- pthread_mutex_lock();
-
- while(condition_is_false)
- pthread_cond_wait();
-
- pthread_mutex_unlock();



pthread_cond_signal pthread_cond_broadcast

- the thread can change the condition variable

```
int pthread_cond_signal(  pthread_cont_t *);
```

  - unlocks at least one thread

```
int pthread_cond_broadcast(pthread_cont_t *);
```

  - allocks all threads

- Other Thread only resumes after this trhread releases mutual exclusion

Condition variables

- void *consumer(void *) {
- `while(1) {`
- `pthread_mutex_lock(&mux);`
- `if (var==threshlold) break;`
- `pthread_mutex_unlock(&mux);`
- `}`
- pthread_mutex_lock(&data_mutex);
- <Extract data from queue;>
- if (queue is empty)
- data_avail = 0;
- pthread_mutex_unlock(&data_mutex);
- <Consume Data>
- }

- void *consumer(void *) {
- pthread_mutex_lock(&data_mutex);
- **while(!data_avail);**
- **/* do nothing */**
- <Extract data from queue;>
- if (queue is empty)
- data_avail = 0;
- pthread_mutex_unlock(&data_mutex)
- <Consume Data>
- }

Condition variables

```
void *producer(void *) {
  <Produce data>
  pthread_mutex_lock(&data_mutex);
  <Insert data into queue;>
  data_avail = 1;

  pthread_cond_signal(&data_cond);
  pthread_mutex_unlock(&data_mutex);
}
```

```
void *consumer(void *) {
  pthread_mutex_lock(&data_mutex);
  while( !data_avail ) {
  /* sleep on condition variable*/
    pthread_cond_wait(&data_cond,
                      &data_mutex);
  }
  /* woken up */
 <Extract data from queue;>
  if (queue is empty)
    data_avail = 0;
  pthread_mutex_unlock(&data_mutex);
  <Consume Data>
}
```

- Task Consumer 1
  - void *consumer(void *) {
  - **pthread_mutex_lock(&data_mutex);**
  - while(!data_avail) {
  - **pthread_cond_wait(&data_cond,**
  - **&data_mutex);**
  - }
  - **pthread_mutex_unlock(&data_mutex);**
  - <Consume Data>
  - }

- Task Consumer 2
  - void *consumer(void *) {
  - **pthread_mutex_lock(&data_mutex);**
  - while(!data_avail) {
  - /* sleep on condition variable*/
  - **pthread_cond_wait(&data_cond,**
  - **&data_mutex);**

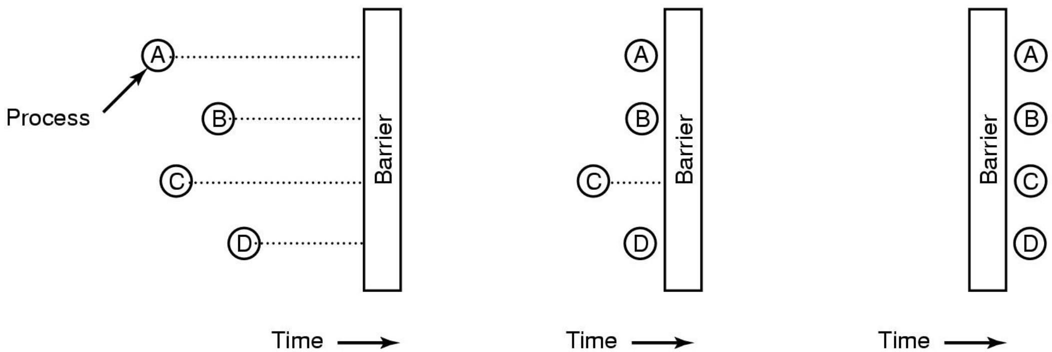
Condition variables

- task Producer
  - void *producer(void *) {
  - <Produce data>
  - **pthread_mutex_lock(&data_mutex);**
  -
  -
  -
  - <Insert data into queue;>
  - data_avail = 1;
  - **pthread_cond_signal(&data_cond);**
  - **pthread_mutex_unlock(&data_mutex);**
  - }

- task Consumer 1
  - void *consumer(void *) {
  - **pthread_mutex_lock(&data_mutex);**
  - while(!data_avail) {
  - /* sleep on condition variable*/
  - **pthread_cond_wait(&data_cond,**
  - **&data_mutex);**
  - }
  -
  -
  - if (queue is empty)
  - data_avail = 0;
  - **pthread_mutex_unlock(&data_mutex);**

Barriers

- [Def] Barries:
  - Synchronization mechanism that blocks threads until a defined number of threads arrives at the barrier



(a) (b) (c)

- Are used when processing is done in steps, that require the completion of a number of threads
  - The next step is only started (by all threads at the same time) after aqll threads have finished the preceding step

Barriers

- Pthreads provides the type pthread_barrier_t

```
pthread_barrier_init(pthread_barrier_t *,
                     pthread_barrierattr_t *,
                     unsigned int)
```

  - 1st parameter - barrier object
  - 2nd parameter - barrier attributes
  - 3rd parameter - number of threads that must call pthread_barrier_wait() before any of them successfully return from the call.

```
pthread_barrier_destroy(pthread_barrier_t *)
```

Barriers

- Wait / Synchronization

```
int pthread_barrier_wait(
                    pthread_barrier_t *)
```

- All threads block
- When the count value (3rd argument from ini) is reached
  - Count thread unblock and start executing
    - Function returns
- Return value
  - one thread - PTHREAD_BARRIER_SERIAL_THREAD,
  - All others - 0

Semaphores using mutex

```
int semaphore_wait (semaphore_t *sem) {
  sem->count --;
  if (sem->count < 0)
     bloquear
  }
int semaphore_post (semaphore_t *sem) {
    sem->count ++;
    if (sem->count <= 0) {
    Desbloqueia tarefa
    }
  }
```

Semaphores using mutex

```
int semaphore_wait (semaphore_t *sem) {
  int res = pthread_mutex_lock(&(sem->mutex));
  if (res != 0) return res;
  sem->count --;
  if (sem->count < 0)
    pthread_cond_wait(&(sem->cond),&(sem->mutex));
  pthread_mutex_unlock(&(sem->mutex));
  return res;
}
```

Semaphores using mutex

```
int semaphore_wait (semaphore_t *sem) {
  sem->count --;
  if (sem->count < 0)
    bloquear
  }
int semaphore_post (semaphore_t *sem) {
  sem->count ;
  if (sem->count < 0)
    bloquear
  }
```

Semaphores using mutex

```
int semaphore_post (semaphore_t *sem) {
  int res = pthread_mutex_lock(&(sem->mutex));
  if (res != 0)   return res;
   sem->count ++;
   if (sem->count <= 0) {
       res = pthread_cond_signal(&(sem->cond));
   }
  pthread_mutex_unlock(&(sem->mutex));
   return res;
}
```

Bounded buffer

- Implement a queue that has two functions
  - enqueue() - adds one item into the queue. It blocks if queue if full
  - dequeue() - remove one item from the queue. It blocks if queue is empty
- The queue has fixed limit
- How to signal that writes can enqueue?
- How to signal that reads can dequeue?

Bounded buffer

```
enqueue(int val){
  sem_wait(&_fullSem);
    _queue[_tail]=val;   _tail = (_tail+1)%MaxSize;
  sem_post(_emptySem);
}
int dequeue(){
  sem_wait(&_emptySem);
  int val = _queue[_head];   _head = (_head+1)%MaxSize;
  sem_post(_fullSem);
  return val;
}
```

Bounded buffer

- Initialization:
  - sem_init(&_emptySem, 0, 0);
  - sem_init(&_fullSem, 0, MaxSize);

```
void enqueue(int val){
  sem_wait(&_fullSem);
  mutex_lock(_mutex);
  _queue[_tail]=val;
  _tail = (_tail+1)%MaxSize;
  mutex_unlock(_mutex);
  sem_post(_emptySem);
}
```

```
int dequeue(){
  sem_wait(&_emptySem);
  mutex_lock(_mutex);
  int val = _queue[_head];
  _head = (_head+1)%MaxSize;
  mutex_lock(_mutex);
  sem_post(_fullSem);
  return val;
}
```

RW locks

- Multiple readers may read the data structure simultaneously
- Only one writer may modify it and it needs to exclude the readers.
- Interface:
  - ReadLock() – Lock for reading. Wait if there are writers holding the lock
  - ReadUnlock() – Unlock for reading
  - WriteLock() - Lock for writing. Wait if there are readers or writers holding the lock
  - WriteUnlock() – Unlock for writing

RW locks

- How to gurantee just one writer?
  - void RWLock::writeLock(){
  -     sem_wait(&_semAccess);
  - }
  - void RWLock::writeUnlock(){
  -     sem_post(&_semAccess);
  - }

RW locks

- How to guarantee that read and writers are not at the same time?
  - first reader blocks writer
  - last read unblocks writer

```
void readLock(){
   _nreaders++;
   if( _nreaders == 1 ) {
   //This is the first reader
   //Get sem_Access
   sem_wait(&_semAccess);
      // read
   }
}
```

```
void readUnlock(){
   _nreaders--;
   if( _nreaders == 0 ) {
   //This is the last reader
   //Allow one writer to
      //proceed if any
   sem_post( &_semAccess );
   }
}
```

RW locks

```
void readLock(){
  mutex_Lock( &_mutex );
  _nreaders++;
  if( _nreaders == 1 )  {
    //This is the first reader
    //Get sem_Access
   sem_wait(&_semAccess);
  }
  mutex_unlock( &_mutex );
}
```

```
void readUnlock(){
  mutex_Lock( &_mutex );
  _nreaders--;
  if( _nreaders == 0 )  {
    //This is the last reader
    //Allow one writer to
    //proceed if any
    sem_post( &_semAccess );
  }
  mutex_unlock( &_mutex );
}
```

- Fairness in locking:
  - First-come-first serve
- Mutexes and semaphores are fair.
  - The thread that has been waiting the longest is the first one to wake up.
- Spin locks (active wait) do not guarantee fairness, the one waiting the longest may not be the one getting it
  - This should not be an issue in the situation when one wants to use spin locks, namely low contention, and short lock holding time