Operating Systems

Complutense University of Madrid

2020-2021

Unit 3.3: Communication and Synchronization between Threads/Processes

Juan Carlos Sáez

OS

Contents

1 Introduction

- Classical Problems of Concurrency

2 Synchronization and communication methods

- Mutexes
- Semaphores
- Condition variables
- Shared memory between processes

3 Implementation of synchronization primitives

OS

Contents

1 Introduction

- Classical Problems of Concurrency

2 Synchronization and communication methods

- Mutexes
- Semaphores
- Condition variables
- Shared memory between processes

3 Implementation of synchronization primitives

Concurrent processes

- **Concurrency**: simultaneous or interleaved execution of multiple instruction streams from different processes or threads
- Models
  - Multiprogramming in a single processor
  - Multiprocessors
  - Multi-computer environment (distributed computing)
- Goals
  - Share physical resources (e.g. CPUs, memory, I/O devices)
  - Share logical resources (e.g. files, data structures)
  - Accelerate calculations
  - Modularity

Classical Problems of Concurrency

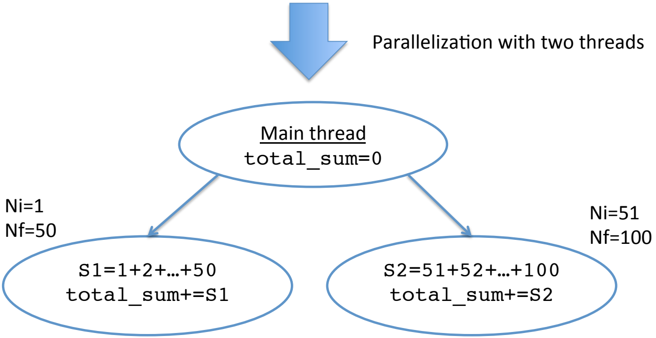
- The critical section problem
- The producer-consumer problem
- The reader-writer problem
- The dining philosophers problem

The critical section problem

- Consider a concurrent program consisting of *n* processes/threads
- Each process/thread includes a code fragment that accesses/-modifies a shared resource
  - Critical section
- Our goal is to ensure that only one process/thread at a time can execute its critical section

The critical section problem: Example 1

```
int i;
int total_sum= 0;
for (i = 1; i <= 100; i++)
        total_sum+=i;
```



The critical section problem: Example 1

- Calculate $\sum_{i=1}^{N} i$ using multiple threads

```
int total_sum = 0; // Shared variable

void calculate_partial_sum(int ni, int nf) {
    int j = 0;
    int partial_sum = 0; // Private variable
    for (j = ni; j <= nf; j++)
        partial_sum = partial_sum + j;
    total_sum = total_sum + partial_sum;
    pthread_exit(0);
}
```

- In the event that several threads run this code concurrently, the final result may be wrong

The critical section problem: Example 1

- Potential implementation in assembly code for the critical section

```
total_sum = total_sum + partial_sum;

LDR R1,total_sum     #R1=0 (first time)
LDR R2,partial_sum   #R2=1275
ADD R1,R1,R2         #R1=1275
STR R1,total_sum     #total_sum=1275
```

The critical section problem: Example 1

- Potentially conflicting situation:

```
    LDR R1,total_sum      #R1=0
    LDR R2,partial_sum    #R2=1275

############# Context Switch #############
    LDR R1,total_sum      #R1=0
    LDR R2,partial_sum    #R2=3775
    ADD R1,R1,R2          #R1=3775
    STR R1,total_sum      #total_sum=3775

############# Context Switch #############
    ADD R1,R1,R2          #R1=1275
    STR R1,total_sum      #total_sum=1275
```

The critical section problem: Example 1

- Solution:
  - Request permission to enter the critical section
  - Notify when exiting the critical section

```
int total_sum = 0; // Shared variable

void calculate_partial_sum(int ni, int nf) {
    int j = 0;
    int partial_sum = 0; // Private variable
    for (j = ni; j <= nf; j++)
        partial_sum = partial_sum + j;

    <Enter critical section>
    total_sum = total_sum + partial_sum;
    <Exit critical section>
    pthread_exit(0);
}
```

The critical section problem: Example 2

- A bank stores the balance of clients' accounts in a per-account file
  - For every money deposit in an account, the balance must be updated in the associated file

```
void perform_deposit(char *account, int amount) {
    int balance, fd;
    fd = open(account, O_RDWR);
    read(fd, &balance, sizeof(int));
    balance = balance + amount;
    lseek(fd, 0, SEEK_SET);
    write(fd, &balance, sizeof(int));
    close(fd);
    return;
}
```

- If two processes run this code concurrently, some deposits may be performed incorrectly

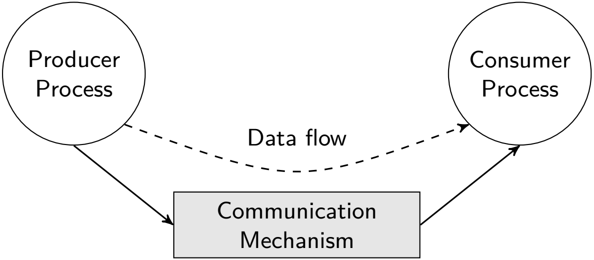
The critical section problem: Example 2

```
void perform_deposit(char *account, int amount) {
    int balance, fd;
    fd = open(account, O_RDWR);
    <Enter critical section>
    read(fd, &balance, sizeof(int));
    balance = balance + amount;
    lseek(fd, 0, SEEK_SET);
    write(fd, &balance, sizeof(int));
    <Exit critical section>
    close(fd);
    return;
}
```

Solution to the critical section problem

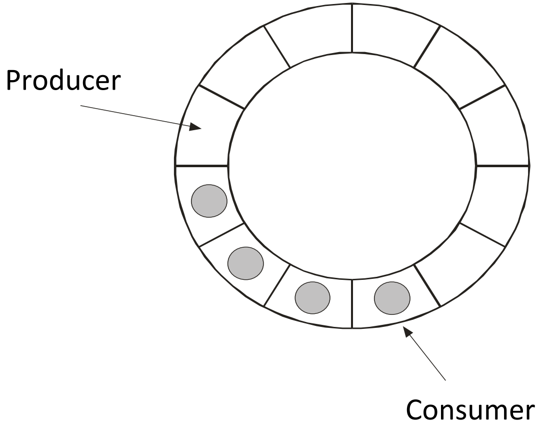
- Requirements of any solution to the critical section (CS) problem:
  - **Mutual exclusion**: only one process in the CS at a time
  - **Efficiency / Progress**: If no process is executing in the critical section, the decision on what process enters the CS depends on the processes that wish to enter the CS
    - A process should enter the CS as quickly as possible when no other process is inside the CS
  - **Avoid starvation / guarantee bounded waiting**: No process must wait forever to enter its CS
- We must also keep in mind:
  1. No assumptions should be made about the relative speeds of the processes or on the number of competing processes
  2. A process remains inside its critical section a finite amount of time

The producer-consumer problem

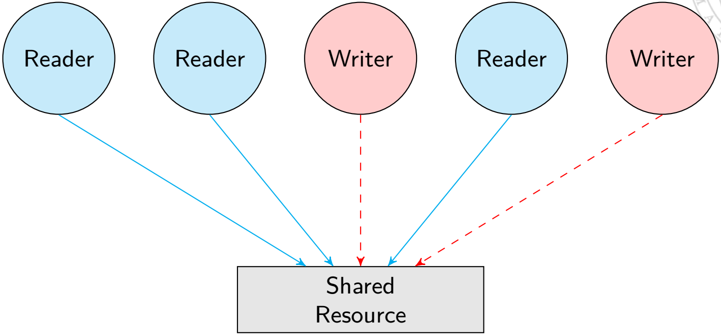


- The consumer will remain blocked until data is received
- The producer will block when attempting to send data over the communication mechanism when it is full

The producer-consumer problem: ring buffer



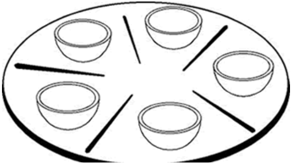
The readers-writers problem



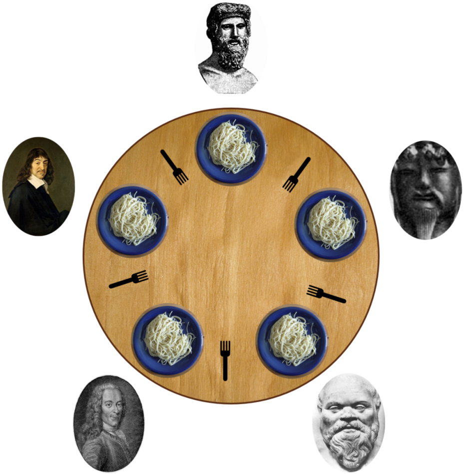
- Readers can execute their critical section (CS) simultaneously
- If a writer is inside the CS, no other process (neither reader nor writer) can be inside the CS

The dining philosophers (Dijkstra'65)

- Five philosophers, sitting around a table, just think and eat rice:
  - They think and eat for a finite amount of time
  - The philosophers need two chopsticks to eat, which must be grabbed one after another
  - No philosopher should starve to death (avoid deadlock and starvation)
- Philosopher's code (infinite loop):
  1. Think...
  2. Grab a chopstick, grab another chopstick
  3. Eat
  4. Put the chopsticks back on the table (one after another) and go to 1.



The dining philosophers (Dijkstra'65)



The dining philosophers (Dijkstra'65)

Solutions:

- Round robin approach
  - Wastes resources
- A waiter controls access to chopsticks
  - Requires a supervisor
- Assign a number (0..4) to each chopstick and philosopher. The philosopher grabs the chopstick with a lowest number first and then the other one. After eating put them back in reverse order.
  - Bad for the last philosopher
- If a philosopher cannot grab the second chopstick, put back the first one on the table
  - What if the neighbor philosophers eat in an alternative way?

Contents

1 Introduction

- Classical Problems of Concurrency

2 Synchronization and communication methods

- Mutexes
- Semaphores
- Condition variables
- Shared memory between processes

3 Implementation of synchronization primitives

Synchronization and communication methods

- All classical problems have several things in common:
  - Processes/threads need to share information
    - They all need to access a shared variable or data structure
  - Processes/threads need to synchronize the execution
    - A process must wait for another at a given point
- We will analyze the most widely used mechanisms provided by modern OSes
  - We will study how each mechanism is used to aid in creating concurrent programs

Communication mechanisms

- Files
- Unnamed and named pipes (FIFOs)
  - Not covered by this course
- Shared memory
  1. Implicit: threads
     - Threads of the same process share data via global variables/data structures
  2. Explicit: different processes
     - Threads from different processes do not share memory
     - Processes communicate through shared memory regions
     - A specific API is needed to create shared memory regions

Synchronization mechanisms

- Operating system services:
  - Locks or mutexes
  - Semaphores
  - Condition variables
  - Signals: asyncronous, cannot be enqueued
  - Unnamed and named pipes (FIFOs)
    - Not covered by this course
- The operations supported by each mechanism must be **atomic**

Locks or mutexes

- Most suitable mechanism to solve the critical-section problem with threads (it enables to enforce MUTual EXclusion)
- A mutex has two possible states:
  - locked or unlocked
- It supports two atomic operations:
  - *Acquire the mutex (lock)*

    ```
    lock(m) {
      while(m.state!=unlocked)
        .. wait ..

      m.state=locked;
    }
    ```
  - *Release the mutex (unlock)*

    ```
    unlock(m) { m.state=unlocked; }
    ```
    - it must be invoked by the thread that invoked `lock()` earlier (*owner*)

Types of locks

```
lock(m) {
  while(m.state!=unlocked)
    .. wait ..

  m.state=locked;
}
```

3 Types of locks or mutexes

1. **Blocking**: The thread goes to *sleep* when waiting
   - State ⇒ `WAITING`
   - `unlock()` *wakes up* a thread ⇒ `READY` state
2. **Busy waiting**: Thread consumes CPU cycles while repeatedly checking the condition
   - Aka *spin lock*
   - Goal: reduce number of context switches
3. **Adaptive**: Thread busy waits for a while and then blocks

Locks or mutexes (*blocking*)

- A lock or mutex is a mechanism specifically tailored for synchronization between threads
  - Well-suited to the critical section problem, since it enforces MUTual EXclusion
- We can think of a mutex as an object with 3 attributes and 2 atomic methods

```
/* locked/unlocked */
state_t state;

/* queue of blocked
     threads */
queue_t q;

/* Owner thread */
thread_id owner;
```

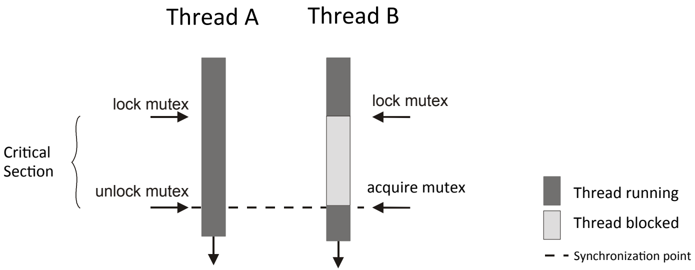
```
lock(m) {
 if (m->state==locked) {
   queue_add(m->q,curThread);
   blockCurThread();
   queue_del(m->q,curThread);
  }
  m->state=locked;
  m->owner=curThread;
}
```

```
unlock(m) {
if (m->owner==curThread) {
  m->state=unlocked;
  m->owner=NULL;
  if (m->q.notEmpty())
    wakeUpOneThread(m->q);
}
else
  error!!
}
```

Critical sections with mutexes

```
lock(m);   /* enter the critical section */
< critical section >
unlock(m); /* exit the critical section */
```

- The `unlock()` operation must be invoked always by the mutex's owner thread



POSIX services for mutexes

- `int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);`
  - Initialize a mutex
- `int pthread_mutex_destroy(pthread_mutex_t *mutex);`
  - Free up the resources associated with a mutex
- `int pthread_mutex_lock(pthread_mutex_t *mutex);`
  - Acquire a mutex. If the mutex is held by another thread, the caller thread blocks until it becomes the mutex owner
- `int pthread_mutex_unlock(pthread_mutex_t *mutex);`
  - Unlocks a mutex. The caller thread must be the mutex owner.

Semaphores (Dijkstra'65)

- Synchronization mechanism
- For processes/threads in a shared-memory computer
- Object with 2 attributes
  - Queue of blocked processes/threads
  - Counter
    - initialized with a value ≥ 0
- 2 atomic operations
  - `wait()`
  - `signal()`

Operations on semaphores

```
wait(s){
  s.c = s.c - 1;
  if(s.c < 0){
     <block process>
  }
}

signal(s){
  s.c = s.c + 1;
  if (any_blocked_process){
     <Wake up a process blocked
          in wait()>
  }
}
```

Meaning of c

- $c \geq 0 \rightarrow c$ is the number of times wait(s) can be invoked in the concurrent program without blocking any caller process
- $c \leq 0 \rightarrow |c|$ is the number of processes blocked in the semaphore's queue

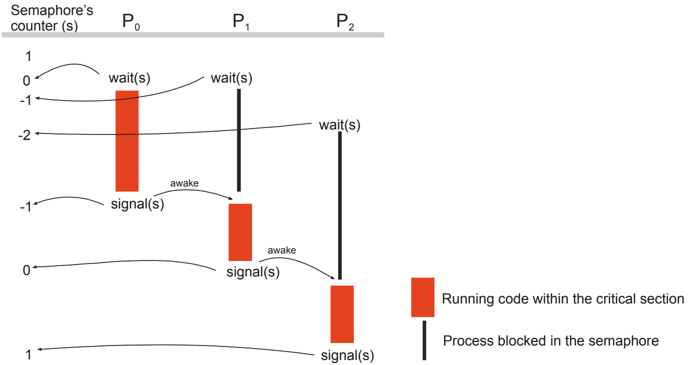
Using semaphores

- Typical use scenarios
  1. **Enforce mutual exclusion**
     - Example: solution to the critical section problem
     - Create global semaphore with c=1 initially
     - Enclose critical sections between `wait()` and `signal()`
  2. **Impose synchronization restrictions associated with conditions based on integer numbers**
     - Example: solution to the producer/consumer problem
  3. **Semaphore as a general-purpose wait queue**
     - Examples: exercise solutions of this unit
     - Create semaphore S with c=0 and define integer variable N to keep track of the number of processes blocked in the semaphore's queue
     - N must be protected by a mutex or by another semaphore, created with c=1

Critical section with semaphores

- Initialize global semaphore with c=1

```
wait(s);   /* enter the critical section */
< critical section >
signal(s); /* exit the critical section*/
```



POSIX semaphores: classification

- 2 variants of POSIX semaphores
  1. **Unnamed semaphores**
     - Make it possible to synchronize threads or processes (usually related)
     - Creation/Destruction via `sem_init()`/`sem_destroy()`
  2. **Named semaphores**
     - Typically used for synchronization between non-related processes
  - Each named semaphore has a global ID associated with (string)
     - Creation/Destruction via `sem_open()`/`sem_unlink()`
- `sem_wait()` and `sem_post()` implement the `wait` and `signal` operations, respectively
  - Can be used for either type of POSIX semaphore
- We will focus on unnamed semaphores

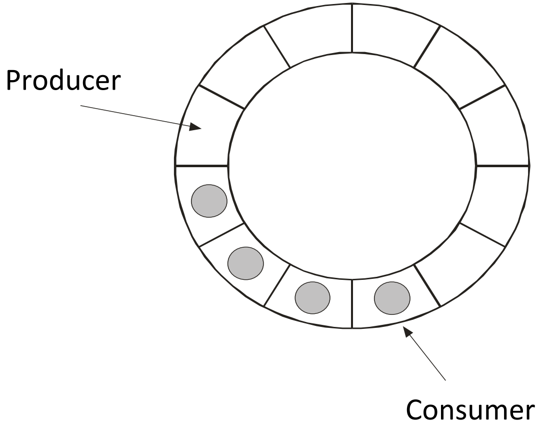
POSIX Semaphores: API

- `int sem_init(sem_t *sem, int shared, unsigned int val);`
  - Initialize an unnamed semaphore
    - `shared`: 0 → threads; 1 → processes
    - if `shared==1`, the semaphore descriptor (`sem_t`) must be stored in a memory region shared between processes
- `int sem_destroy(sem_t *sem);`
  - Free up the resources associated with an unnamed semaphore
- `int sem_wait(sem_t *sem);`
  - `wait` operation on the semaphore
- `int sem_post(sem_t *sem);`
  - `signal` operation on the semaphore

POSIX Semaphores: API

- `sem_t *sem_open(char* name,int flags,mode_t mode, unsigned int val);`
  - Opens and/or creates a named semaphore
- `int sem_close(sem_t *sem);`
  - Closes a named semaphore
- `int sem_unlink(char *name);`
  - Removes a named semaphore
- `int sem_wait(sem_t *sem);`
  - `wait` operation on the semaphore
- `int sem_post(sem_t *sem);`
  - `signal` operation on the semaphore

Producer-consumer: ring buffer



Producer-consumer with semaphores (I)

```
#define MAX_BUF         1024   /* buffer size */
#define PROD          100000   /* number of items to produce */
sem_t items;           /* number of items in the buffer */
sem_t gaps;                /* number of free gaps in the buffer */
int buffer[MAX_BUF];     /* shared buffer */

void main(void){
  pthread_t th1, th2; /* thread descriptors */
  /* semaphore initialization */
  sem_init(&items, 0, 0); sem_init(&gaps, 0, MAX_BUF);

  /* thread creation */
  pthread_create(&th1, NULL, Producer, NULL);
  pthread_create(&th2, NULL, Consumer, NULL);

  /* wait for thread completion */
  pthread_join(th1, NULL); pthread_join(th2, NULL);

  sem_destroy(&gaps); sem_destroy(&items);
  exit(0);
}
```

Producer-consumer with semaphores (II)

```c
void Producer(void){
 int widx = 0; /* write index */
 int data; /* data to produce */
 int i;

 for(i=0; i < PROD; i++ ){
    /* produce data */
    data = generate_data();
    /* one gap less */
    sem_wait(&gaps);
    buffer[widx] = data;
    widx = (widx + 1) % MAX_BUF;
    /* added one item */
    sem_post(&items);
 }

 pthread_exit(0);
}
```

```c
void Consumer(void){
 int ridx= 0; /* read index */
 int data; /* data to be consumed */
 int i;

 for(i=0; i<PROD; i++ ){
    /* an item will be removed */
    sem_wait(&items);
    data = buffer[ridx];
    ridx= (ridx+ 1) % MAX_BUF;
    /* one free gap more */
    sem_post(&gaps);
    do_something(data);
  }

 pthread_exit(0);
}
```

N Producers - 1 consumer with semaphores

```
#define MAX_BUF          1024   /* buffer size */
#define PROD           100000   /* number of items to produce */
#define N 2   /* Number of producers */
sem_t items;           /* number of items in the buffer */
sem_t gaps;                /* number of free gaps in the buffer */
sem_t producers; /* To enforce mutual exclusion among producers*/
int widx=0;       /* Shared write index */
int buffer[MAX_BUF];    /* shared buffer */

void main(void){
   int i;
   pthread_t thp[N], thc; /* thread descriptors */
   /* semaphore initialization */
   sem_init(&items, 0, 0); sem_init(&gaps, 0, MAX_BUF);
   sem_init(&producers, 0, 1);

   /* thread creation */
   for (i=0;i<N;i++) pthread_create(&thp[i], NULL, Producer, NULL);
   pthread_create(&th2, NULL, Consumer, NULL);

   /* wait for thread completion */
   for (i=0;i<N;i++) pthread_join(&thp[i], NULL);
   pthread_join(th2, NULL);

   sem_destroy(&gaps); sem_destroy(&items); sem_destroy(&producers);
   exit(0);
}
```

N Producers - 1 consumer with semaphores

```
void Producer(void){
 int data; /* data to produce */
 int i;

 for(i=0; i < PROD; i++ ){
   /* produce data */
   data = generate_data();
   /* one gap less */
   sem_wait(&gaps);
   sem_wait(&producers);
   /* Critical section */
   buffer[widx] = data;
   widx = (widx + 1) % MAX_BUF;
   sem_post(&producers);
   /* added one item */
   sem_post(&items);
 }

 pthread_exit(0);
}
```

```
void Consumer(void){
 int ridx= 0; /* read index */
 int data; /* data to be consumed */
 int i;

 for(i=0; i<PROD; i++ ){
   /* an item will be removed */
   sem_wait(&items);
   data = buffer[ridx];
   ridx= (ridx+ 1) % MAX_BUF;
   /* one free gap more */
   sem_post(&gaps);
   do_something(data);
  }

 pthread_exit(0);
}
```

Readers-writers with semaphores (I)

```
int data = 5;     /* shared resource */
int nr_readers = 0; /* number of readers */
sem_t sem_nreaders; /* control access to nr_readers */
sem_t sem_read_write; /* mutual exclusion between reader-writer and
     writer-writer */

void main(void){
   pthread_t th1, th2, th3, th4;
   sem_init(&sem_read_write, 0, 1); sem_init(&sem_nreaders, 0, 1);

   pthread_create(&th1, NULL, Reader, NULL);
   pthread_create(&th2, NULL, Writer, NULL);
   pthread_create(&th3, NULL, Reader, NULL);
   pthread_create(&th4, NULL, Writer, NULL);

   pthread_join(th1, NULL); pthread_join(th2, NULL);
   pthread_join(th3, NULL); pthread_join(th4, NULL);

   /* destroy semaphores */
   sem_destroy(&sem_read_write); sem_destroy(&sem_nreaders);
   exit(0);
}
```

Readers-writers with semaphores (II)

```c
void Reader(void) {
  while(1){
    sem_wait(&sem_nreaders);
    nr_readers = nr_readers + 1;
    if (nr_readers == 1)
      sem_wait(&sem_read_write);
    sem_post(&sem_nreaders);

    /* read data */
    printf("%d\n", data);

    sem_wait(&sem_nreaders);
    nr_readers = nr_readers - 1;
     if (nr_readers == 0)
      sem_post(&sem_read_write);
    sem_post(&sem_nreaders);
  }
}
```

```c
void Writer(void) {
  while(1){
    sem_wait(&sem_read_write);

    /* modify the resource */
    data = data + 2;

    sem_post(&sem_read_write);

  }
}
```

Condition variables

- Synchronization mechanism for threads
- Each condition variable has a wait queue and a mutex associated with it
  - The mutex is typically shared between multiple condition variables in the same concurrent program
- Condition variables support three operations:
  1. `cond_wait()`: the caller thread blocks in the wait queue
  2. `cond_signal()`: wake up a thread waiting in the condition variable's wait queue (if there are blocked threads)
  3. `cond_broadcast()`: wake up all threads waiting in the condition variable's wait queue (if there are blocked threads)
- These operations must be invoked in a code snippet between `lock(mutex)` and `unlock(mutex)`

Operations on condition variables (I)

```
void cond_wait(lock_t m, vc_t varC ) {
    queue_add(varC->queue, curThread);
    unlock(m);
    park(); // the thread goes to sleep
    lock(m);
}
```

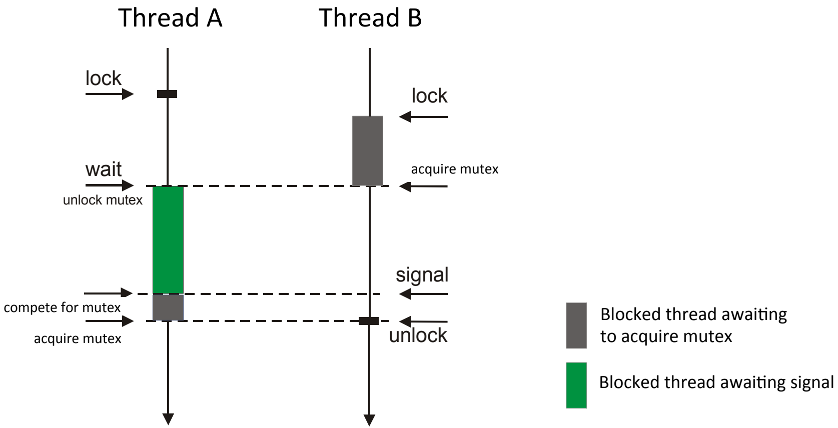
- The caller thread must be the owner of the mutex
- `cond_wait()` **always** blocks the caller thread
  - Before going to sleep, the thread unlocks the mutex so that another thread can acquire it
- When the thread is awoken, it attempts to reacquire (lock) the mutex again (the thread may block)
- When the `cond_wait()` operation returns, the caller thread is the owner of the mutex again

Operations on condition variables (II)

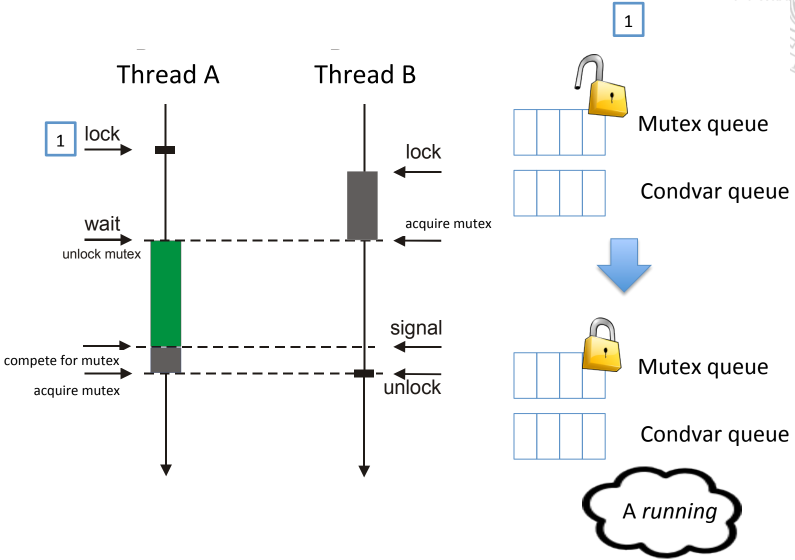
```
/* Wakes up one thread from the wait queue */
void cond_signal (vc_t varC ) {
   if (!isEmpty(varC->queue))
      unpark(queue_remove(varC->queue))
}

/* Wakes up all threads in the wait queue */
void cond_broadcast (vc_t varC ) {
   while (!isEmpty(varC->queue))
         unpark(queue_remove(varC-queue))
}
```

- It is strongly advisable that these operations are invoked in a code snippet between `lock(mutex)` and `unlock(mutex)`

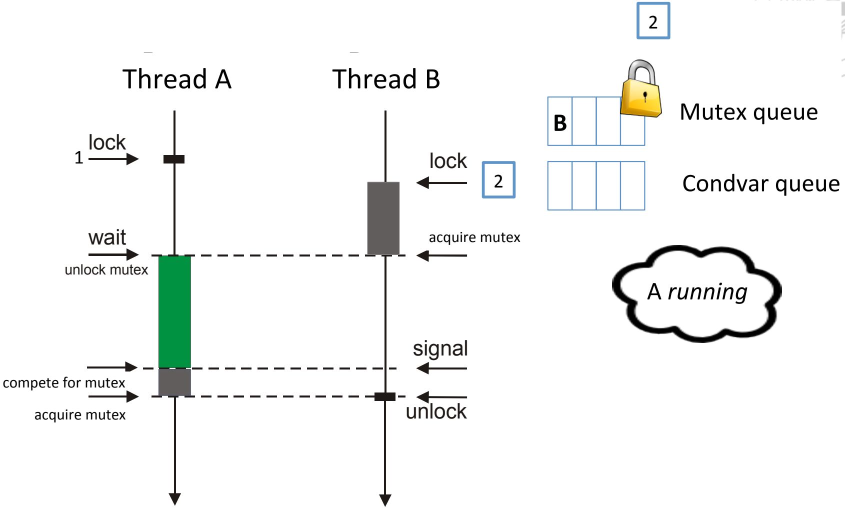
Operations on condition variables (III)



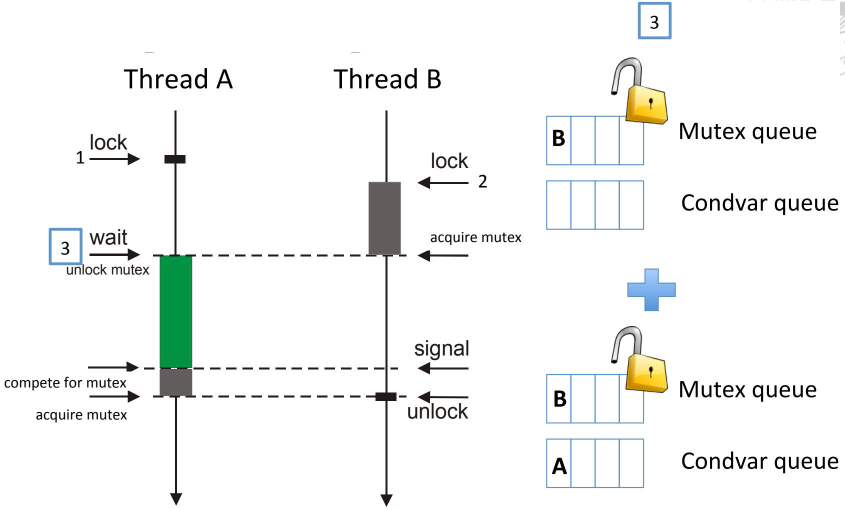
Using condition variables



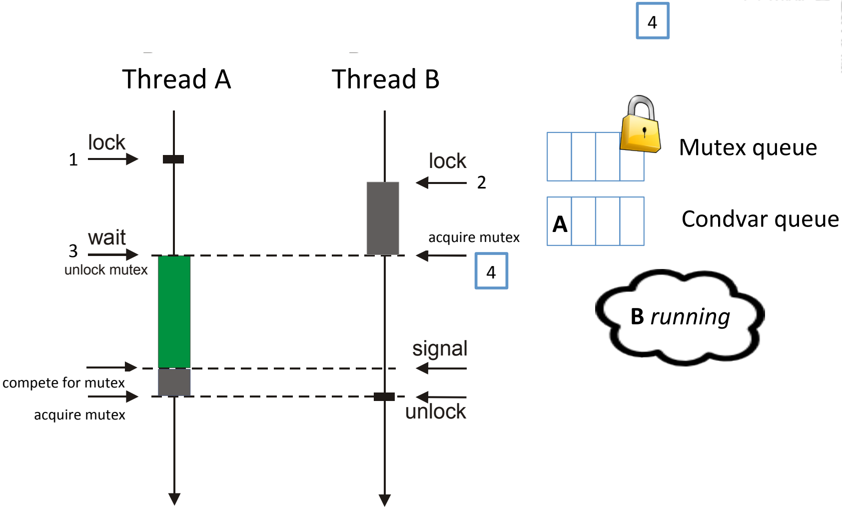
Using condition variables



Using condition variables

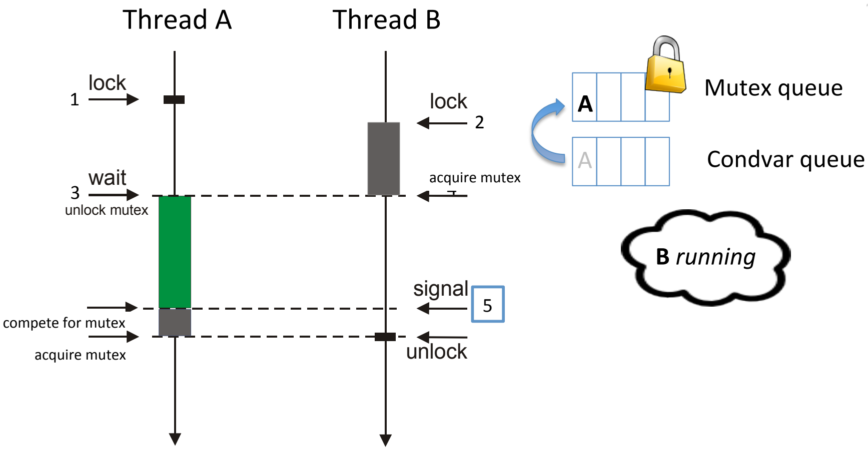


Using condition variables

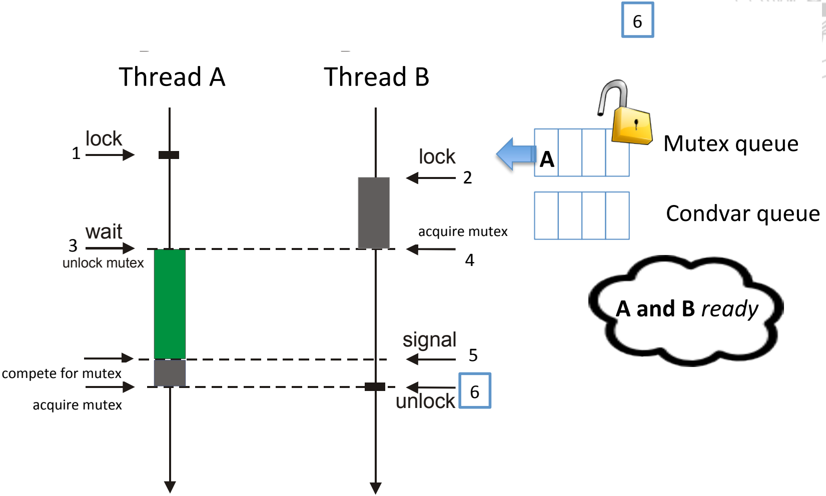


Using condition variables

5



Using condition variables



Using condition variables with a mutex

- Thread A

```
lock(mutex); /* enter the CS */
<operations on the shared resource (mutual exclusion)>
while (condition related to the resource == false)
   cond_wait(condition, mutex); /* thread blocks */
<desired actions once the condition is met>
unlock(mutex); /* exit the CS */
```

- Thread B

```
lock(mutex); /* enter the CS */
<operations on the shared resource (mutual exclusion)>
/* Since we may have affected the condition, wake up other thread */
cond_signal(condition, mutex);
<more operations on the shared resource>
unlock(mutex); /* exit the CS */
```

We must use `while` to re-evaluate the condition

Producer-Consumer with cond. variables (I)

- For simplicity, assume that the ring buffer is already implemented as an abstract data type (`cbuffer_t`)
  - Operations on `cbuffer_t`:
    - `boolean is_empty(cbuffer_t cb);`
    - `boolean is_full(cbuffer_t cb);`
    - `void add(cbuffer_t cb, item_t data);`
    - `item_t remove(cbuffer_t cb);`
- Operation restrictions
  1. Thread-unsafe implementation
     - The operations cannot be invoked simultaneously from multiple threads
     - A lock is needed to serialize accesses to the data structure
  2. `add()` cannot be invoked if buffer is full
  3. `remove()` cannot be invoked if buffer is empty

Producer-Consumer with cond. variables (II)

Global variables

```
cbuffer_t b; /* shared ring buffer (max capacity: N items) */
mutex m; /* Mutual exclusion when accessing buffer */
condvar prod,cons; /* To block producer/consumer, respectively */
```

```
void producer() {
  item_t data;

  while (true) {
    data=produce();
    lock(m);




    add(b,data);




    unlock(m);
    delay(...);
  }
}
```

```
void consumer() {
 item_t data;

 while (true) {
    lock(m);




    data=remove(b);




    unlock(m);
    do_something(data);
   }
}
```

Producer-Consumer with cond. variables (II)

Global variables

```
cbuffer_t b; /* shared ring buffer (max capacity: N items) */
mutex m; /* Mutual exclusion when accessing buffer */
condvar prod,cons; /* To block producer/consumer, respectively */
```

```
void producer() {
  item_t data;

  while (true) {
    data=produce();
    lock(m);

    while (is_full(b))
       cond_wait(prod,m);

    add(b,data);



    unlock(m);
    delay(...);
   }
}
```

```
void consumer() {
 item_t data;

 while (true) {
    lock(m);

    while (is_empty(b))
        cond_wait(cons,m);

    data=remove(b);



    unlock(m);
    do_something(data);
   }
}
```

Producer-Consumer with cond. variables (II)

Global variables

```
cbuffer_t b; /* shared ring buffer (max capacity: N items) */
mutex m; /* Mutual exclusion when accessing buffer */
condvar prod,cons; /* To block producer/consumer, respectively */
```

```
void producer() {
  item_t data;

  while (true) {
    data=produce();
    lock(m);

    while (is_full(b))
       cond_wait(prod,m);

    add(b,data);

    cond_signal(cons);

    unlock(m);
    delay(...);
   }
}
```

```
void consumer() {
 item_t data;

 while (true) {
    lock(m);

    while (is_empty(b))
        cond_wait(cons,m);

    data=remove(b);

    cond_signal(prod);

    unlock(m);
    do_something(data);
   }
}
```

POSIX Services (II)

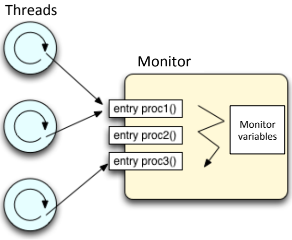
- `int pthread_cond_init(pthread_cond_t* cond, pthread_condattr_t* attr);`
  - Initialize a condition variable
- `int pthread_cond_destroy(pthread_cond_t *cond);`
  - Free up the resources associated with a condition variable
- `int pthread_cond_signal(pthread_cond_t *cond);`
  - Unblock one thread waiting in the queue associated with a condition variable
  - It has no effect if the wait queue is empty (different behavior to that of the signal operation for semaphores).
- `int pthread_cond_broadcast(pthread_cond_t *cond);`
  - Unblock all threads waiting in the queue associated with a condition variable
- `int pthread_cond_wait(pthread_cond_t* cond, pthread_mutex_t* mutex);`
  - Blocks the caller thread until it is eventually awoken by `pthread_cond_signal()` or `pthread_cond_broadcast()`

Monitors

- Semaphores enable us to:
  1. Enforce mutual exclusion
  2. Impose synchronization restrictions in the program
- However, implementations with semaphores are usually hard to follow and difficult to debug
- Monitors make it possible to separate both aspects by
  - using mutexes to ensure mutual exclusion
  - and condition variables to impose more elaborate synchronization restrictions based on conditions depending on the state of shared variables among threads
- More suitable for object oriented programming (e.g. Java)
- A monitor is not a synchronization mechanism provided by the OS but instead we must build it specifically for our concurrent program

Monitors

- A monitor is an object consisting of 3 components:
  1. Monitor procedures (*methods*)
  2. Variables and data structures shared between threads (*attributes*)
  3. A mutex + N condition variables
     - The mutex is implicit in some monitor implementations
- Monitor procedures (methods) can be invoked concurrently by multiple threads
  - Concurrency issues are addressed inside each monitor procedure



Monitors: Implementation (I)

- A thread "enters the monitor" when it invokes a monitor procedure; the thread "exits the monitor" when it completes the execution of the procedure
- Implementation restrictions:
  1. Only one active (non-blocked) thread can be inside a monitor procedure at a time
     - The thread running inside the monitor accesses the monitor's variables in mutual exclusion with respect to other threads
  2. One or several threads can be blocked inside the monitor
     - Threads may remain blocked in the condition variables associated with the monitor

Monitors: Implementation (II)

- In the C/POSIX-threads monitor implementation, the aforementioned restrictions can be imposed as follows:
  - A monitor procedure is a C function with a single return point
  - The body of the monitor procedure is enclosed between `lock()` and `unlock()` of the monitor mutex

```
monitor_procedure() {
    pthread_mutex_lock(&mutex); /* enter the monitor */
    <procedure body>
    pthread_mutex_unlock(&mutex); /* exit the monitor */
}
```

- Every thread blocked in a monitor's condition variable, releases the mutex implicitly when invoking `cond_wait()`

Using monitors

- Threads in a concurrent program invoke the monitor procedures for different reasons:
  1. Modify/access the monitor variables safely
  2. Enter/exit a critical section

```
entry_procedure();
<< critical section >>
exit_procedure();
```

Producer-consumer with monitor

```c
#define MAX_BUFFER 1024 /* buffer size */
#define DATA_TO_PRODUCE 10000
pthread_mutex_t mutex;  /* monitor's mutex */
pthread_cond_t c_full;  /* to block the producer */
pthread_cond_t c_empty; /* to block the consumer */
int nr_items;           /* # of items in the buffer */
int buffer[MAX_BUFFER]; /* shared buffer */
int ridx,widx;        /* R/W positions in the buffer */

void init_monitor(void); /* Initialize monitor */
void destroy_monitor(void); /* Free up monitor resources */
/* Monitor procedures */
void produce(int item); /* Insert item into the buffer */
int consume(void);   /* Extract an item from the buffer */
```

Producer-consumer with monitor

```
void Producer(void) {
 int i,item;
 for (i=0;i<DATA_TO_PRODUCE;i++){
   item=... generate an item ...
   produce(item);
  }
 pthread_exit(0);
}
```

```
void Consumer(void) {
 int i,item;
 for (i=0;i<DATA_TO_PRODUCE;i++){
  item=consume();
    ... Do something with item ...
  }
  pthread_exit(0);
}
```

```
int main(int argc, char *argv[]){
    pthread_t th1, th2;
    init_monitor();
    pthread_create(&th1, NULL, Producer, NULL);
    pthread_create(&th2, NULL, Consumer, NULL);
    pthread_join(th1, NULL); pthread_join(th2, NULL);
    destroy_monitor();
    return 0;
}
```

Producer-consumer with monitor

```
void init_monitor(void)
{
   pthread_cond_init(&c_full, NULL);
   pthread_cond_init(&c_empty, NULL);
   pthread_mutex_init(&mutex, NULL);
   ridx=widx=nr_items=0;
}

void destroy_monitor(void)
{
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&c_full);
    pthread_cond_destroy(&c_empty);
}
```

Producer-consumer with monitor

```
void produce(int item) {
 /* "enter the monitor" */
 pthread_mutex_lock(&mutex);

 /* block while buffer full */
while (nr_items == MAX_BUFFER)
 pthread_cond_wait(&c_full,&mutex);

 buffer[widx] = item;
 widx= (widx+ 1) % MAX_BUFFER;
 nr_items++;

 /* buffer is not empty */
 pthread_cond_signal(&c_empty);

 /* "exit the monitor" */
 pthread_mutex_unlock(&mutex);
}
```

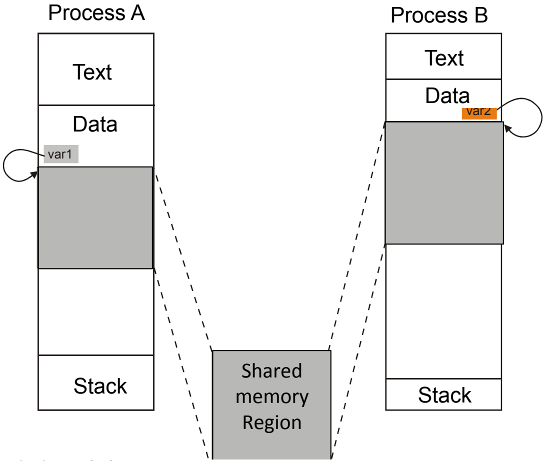
```
int consume(void) {
 int item;
 /* "enter the monitor" */
  pthread_mutex_lock(&mutex);

/* block while buffer empty */
 while (nr_items == 0)
  pthread_cond_wait(&c_empty,&mutex);

 item = buffer[ridx];
 ridx= (ridx + 1) % MAX_BUFFER;
 nr_items--;

 /* buffer is not full */
 pthread_cond_signal(&c_full);

 /* "exit the monitor" */
 pthread_mutex_unlock(&mutex);
 return item;
}
```

Communication mechanisms

- Files
- Unnamed and named pipes (FIFOs)
  - Not covered by this course
- Shared memory
  1. Implicit: threads
     - Threads of the same process share data via global variables/data structures
  2. Explicit: different processes
     - Threads from different processes do not share memory
     - Processes communicate through shared memory regions
     - A specific API is needed to create shared memory regions

Shared memory between processes

- Independent declaration of variables inside each process that refers to the same physical memory location



POSIX shared memory

- **`void *mmap(void *addr, size_t length, int prot, int flags,int fd, off_t offset);`**
  - maps part of a file (described by `fd`) into memory and returns a pointer to that memory region (`addr`)
  - The memory region can be either shared or private:
    - `flags`: `MAP_SHARED` or `MAP_PRIVATE`
  - The memory region can be declared without an associated file:
    - `flags`: `MAP_ANONYMOUS` (shared between parent and child)
    - By using `shm_open` to obtain a file descriptor
- **`int munmap(void *addr, size_t length);`**
  - delete the mappings for the specified address range, causing further references to addresses within the range to generate invalid memory references
- **`int msync(void *addr, size_t len, int flags);`**
  - writes modified whole pages back to the filesystem and updates the file modification time.

Summary

- Threads:
  - Shared memory (global variables)
  - Mutexes, condition variables, semaphores, monitors.
- Related processes (`fork()`):
  - Unnamed Pipes, shared memory
  - Unnamed semaphores (memory-mapped)
- Non-related processes:
  - Named semaphores
  - Named Pipes, shared memory

P&C with shared memory and semaphores

- Producer:
  - Creates named semaphores (`sem_open`)
  - Creates a file (`open`)
  - Assigns space to it (`ftruncate`)
  - Maps the file into the process address space (`mmap`)
  - Uses the shared memory region
  - Unmaps the shared memory region (`munmap`)
  - Closes the file and deletes it
- Consumer:
  - Opens the named semaphores (`sem_open`)
  - The consumer must wait until the file is created to open it (`open`)
  - Maps the file into the process address space (`mmap`)
  - Uses the shared memory region
  - Closes the file

Producer's code

```
#define MAX_BUFFER      1024   /* buffer size */
#define DATA_TO_PRODUCE 100000 /* # elements to produce */
sem_t *elements; /* # of elements in the buffer */
sem_t *gaps;   /* # of free gaps in the buffer */

void main(int argc, char *argv[]){
   int shd;
   int *buffer;  /* shared buffer */

   /* the producer creates the file */
   shd = open("BUFFER", O_CREAT|O_WRONLY, 0700);
   ftruncate(shd, MAX_BUFFER * sizeof(int));

   /* Maps the file into the process address space */
   buffer = (int*) mmap(NULL, MAX_BUFFER * sizeof(int),
               PROT_WRITE, MAP_SHARED, shd, 0);
```

Producer's code (II)

```
/* The producer creates the semaphores */
   elements = sem_open("ELEMENTS", O_CREAT, 0700, 0);
   gaps =   sem_open("GAPS", O_CREAT, 0700, MAX_BUFFER);

   /* core producer's code */
   Producer(buffer);

   /* Unmap shared buffer */
   munmap(buffer, MAX_BUFFER * sizeof(int));
   close(shd);      /* close the shared memory region */
     unlink("BUFFER"); /* delete the shared memory region */

   sem_close(elements);
   sem_close(gaps);
   sem_unlink("ELEMENTS");
   sem_unlink("GAPS");
}
```

Consumer's code

```
#define MAX_BUFFER      1024   /* buffer size */
#define DATA_TO_PRODUCE 100000 /* # elements to produce */
sem_t *elements; /* # of elements in the buffer */
sem_t *gaps;   /* # of free gaps in the buffer */

void main(int argc, char *argv[]){
   int shd;
   int *buffer;  /* shared buffer */

   /* the consumer opens the file */
   shd = open("BUFFER", O_RDONLY);

  /* Maps the file into the process address space */
   buffer = (int *) mmap(NULL, MAX_BUFFER * sizeof(int),
                         PROT_READ, MAP_SHARED, shd, 0);
```

Consumer's code (II)

```
    /* Consumer opens semaphores */
    elementos = sem_open("ELEMENTS", 0);
    huecos   = sem_open("GAPS", 0);

    /* consumer's core processing */
    Consumer(buffer);

    /* unmap shared buffer */
    munmap(buffer, MAX_BUFFER * sizeof(int));
    close(shd); /* close shared memory object */

    /* close semaphores */
    sem_close(elements);
    sem_close(gaps);
}
```

Producer's core processing

```
void Producer(int *buffer)
{
   int pos = 0; /* write index */
   int item;    /* data to produce */
   int i;

   for(i=0; i < DATA_TO_PRODUCE; i++ ) {
      item = produce_item();
      sem_wait(gaps);
      buffer[pos] = item;
      pos = (pos + 1) % MAX_BUFFER;
      sem_post(elements);
   }
}
```

Consumer's core processing

```
void Consumer(char *buffer)
{
   int pos = 0; /* read index */
   int i, item;

   for(i=0; i < DATA_TO_PRODUCE; i++ ) {
      sem_wait(elements);
      dato = buffer[pos];
      pos = (pos + 1) % MAX_BUFFER;
      sem_post(gaps);
      printf("Received %d\n", item);
   }
}
```

Contents

1 Introduction

- Classical Problems of Concurrency

2 Synchronization and communication methods

- Mutexes
- Semaphores
- Condition variables
- Shared memory between processes

3 Implementation of synchronization primitives

Implementation of lock()/unlock()

- The OS ensures that lock() and unlock() are **atomic operations**
  - Note that the implementation entails solving the critical section problem (shared variables)

```
lock(m) {
if (m->state==locked) {
  queue_add(m->q,curThread);
  blockCurThread();
  queue_del(m->q,curThread);
 }
 m->state=locked;
 m->owner=curThread;
}
```

```
unlock(m) {
if (m->owner==curThread) {
  m->state=unlocked;
  m->owner=NULL;
  if (m->q.notEmpty())
    wakeUpOneThread(m->q);
}
else
  error!!
}
```

Solutions to the critical section problem

Types of solutions

- Based on busy waiting
  - Without HW support
    - Based on control variables (Peterson 1981)
  - With HW support
    - Test And Set (TAS), XCHG, LL/SC
- Without busy waiting (blocked waiting)
  - OS support needed
  - The OS blocks the process/thread

Machine instructions

- A machine instruction is used to update the contents of a memory address atomically
- Solution works for *any* number of processes/threads:
  - **RMW** memory cycle (*read/modify/write*)
- These special instructions are not negativelly affected by other instructions
- The mechanism can be applied to multiple critical sections
- Simple mechanism that can be easily verified

Machine instructions: example

General

- Test and set (T&S)
- Fetch and add (F&A)
- Swap/Exchange
- Compare and Swap (exchange)
- Load link/ Store conditional (LL/SC)

Intel (x86)

- Many atomic instructions supported
- F&A lock; xaddl eax, [mem_addr];
- XCHG xchg eax, [mem_addr]
- CMPXCHG -> lock cmpxchg [mem_addr], eax

ARM (and others)

- LL/SC LDREX y STREX

Semantics of Swap/Exchange

Exchange

```
xchg src, dst
-------------
   rtmp  <- Mem [src]
   Mem [src] <- Mem [dst]
   Mem [dst] <- rtmp
```

- It is a machine instruction rather than a function
  - It is atomic, uninterruptible
- Swaps two values (Both potentially in memory)
  - On Intel processors, just one of both (src or dst) can be on memory

Usage of Swap/Exchange

- Solution to the **critical section problem with XCHG**

Implementation

```
/* it may be stored in a register */
tmp = 1;
/* Busy wait */
while( tmp== 1)
   xchg(MemAddr, tmp);

Critical_section();

*MemAddr= 0;
```

Semantics of LL/SC

Load Link

```
ll src
-------
   rout <- Mem [src]
```

Store Conditional

```
sc src, value
--------------
  if nobody accessed src since the last LL
    Mem[src]= value
    rout <- 1
  sino
    rout <- 0
```

- LL/SC constitute two independent machine instructions
  - LL takes care of loading the data in a register
  - SC only stores the data if no writes were performed to that memory location prior to the execution of LL

Usage of LL/SC

- Solution to the **critical section problem with LL/SC**

Implementation

```
while (1) {
    while(ll(MemAddr)== 1);
    if (sc(MemAddr,1)==1) break;
    /* otherwise, execute Load-Link again */
}
Critical_section();
*MemAddr= 0;
```