Concurrent Programming III:
message passing, deadlock, livelock

Anna Lina Ruscelli – Scuola Superiore Sant’ Anna
Outline

- Model of concurrency: message passing
- Deadlock
- Livelock
Model of Concurrency: message passing
Interaction model of Concurrency

- Activities can interact according to two fundamental models:
  - **Shared memory**
    - All activities access the same memory space;
    - Semaphores, Mutexes.
  - **Message passing**
    - All activities communicate each other by sending messages through OS primitives (*send*, *receive*).
Message passing

- Message passing systems are based on the basic concept of message.

- Two basic operations
  - `send(destination, message);`
    - send can be synchronous or asynchronous.
  - `receive(source, &message);`
    - receive can be symmetric or asymmetric.
Competition

Cooperative and competitive activities need different models of execution and synchronization.

- **Competing activities** need to be “protected” from each other
  - separate memory spaces.
- The resource allocation and the synchronization must be centralized
  - competitive activities request for services to a central manager (the OS or some dedicated process) which allocates the resources in a fair way.

- **Client/Server model**
  - communication is usually done through messages.
- The **process model** of execution is the best one.
In a client/server system
- a server manages the resource exclusively
  - for example, the printer
- if a process needs to access the resource, it sends a request to the server
  - for example, printing a file, or asking for the status
- the server can send back the responses,
- the server can also be on a remote system.

Two basic primitives
- send and receive.
Producer/Consumer with MP

- The **producer** executes `send(consumer, data);`
- The **consumer** executes `receive(producer, &data);`
- No need for a special communication structure (already contained in the send/receive semantic).
Resources and message passing

- There are **no shared resources** in the message passing model
  - all the resources are **allocated statically**, accessed in a dedicated way;
  - separated address spaces of the processes;
- Each resource is handled by a **manager process** that is the only one that has the right to access to a resource.
- The consistency of a data structure is guaranteed by the manager process
  - there is no more competition, only cooperation!!!
Resource allocation

- Allocation of resource can be
  - **static**: once the resource is granted, it is never revoked,
  - **dynamic**: resource can be granted and revoked dynamically
    - manager

- Access to a resource can be
  - **dedicated**: one activity at time only is granted access to the resource,
  - **shared**: many activities can access the resource at the same time
    - mutual exclusion.

<table>
<thead>
<tr>
<th></th>
<th>Dedicated</th>
<th>Shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>Compile Time</td>
<td>Manager</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Manager</td>
<td>Manager</td>
</tr>
</tbody>
</table>
Example: Word processor application

- Problem with **multiple processes**
  - All processes need to access the same data structure, the document.
  - Which process holds the data structure?
- Solution with **message passing**
  - A dedicated process holds the data, all the others communicate with it to read/update the data.
  - Very inefficient!
The message passing systems classification is based on:

1. **Synchronization type** (synchronous or asynchronous)
2. **Specification of source and sink** (symmetric or asymmetric).
1.1 Synchronous communication

- **Synchronous send/receive**
- no buffers!

```c
producer: s_send(consumer, d);
```

```c
consumer: s_receive(producer, &d);
```
1.2 Async send/ sync receive

- Asynchronous send/synchronous receive
  - there is probably a send buffer somewhere.

```c
producer:  
a_send(consumer, d);

consumer:  
s_receive(producer, &d);
```
2.1 Symmetric and Asymmetric receive

- **Symmetric receive**
  - `receive(source, &data);`
  - the consumer wants a message from a **given** producer.

- **Asymmetric receive**
  - `source = receive(&data);`
  - often, we do not know who is the sender/producer
    - imagine a web server;
    - the programmer cannot know in advance the address of the browser that will request the service;
    - many browser can ask for the same service.
Send: Remote Procedure Call

- In a **client server system**, a client wants to request an action to a server
  - that is typically done using a Remote Procedure Call (RPC).
Message passing systems

In message passing

- each resource needs one threads manager,
- the threads manager is responsible for giving access to the resource.

Example: let’s try to implement mutual exclusion with message passing primitives

- one thread will ensure mutual exclusion
- every thread that wants to access the resource must:
  - send a message to the manager thread to access the resource,
  - access the critical section,
  - send a message to signal the leaving of the critical section.
Mutual exclusion with sync send / sync receive

```c
void * manager(void *)
{
    thread_t source;
    int d;
    while (true) {
        source = s_receive(&d);
        s_receive_from(source, &d);
    }
}

void * thread(void *)
{
    int d;
    while (true) {
        s_send(manager, d);
        <critical section>
        s_send(manager, d);
    }
}
```
Mutual exclusion with async send and sync receive

```c
void * manager(void *)
{
    thread_t source;
    int d;
    while (true) {
        source = s_receive(&d);
        a_send(source, d);
        s_receive_from(source, &d);
    }
}
```

```c
void * thread(void *)
{
    int d;
    while (true) {
        a_send(manager, d);
        s_receive_from(manager, &d);
        <critical section>
        a_send(manager, d);
    }
}
```
Problem

- Implement readers/writers with message passing.

Hints:
- define a thread manager,
- the service type (read/write) can be passed as data,
- use asynchronous send and synchronous receive,
- use symmetric and asymmetric receive.
Deadlock and Livelock
Deadlock and livelock

- **Deadlock** is the situation where a group of threads is permanently blocked waiting for some resource.
- Deadlock can happen in many subtle cases.
  - Example: dining philosophers.
- Here we will study the ways of avoiding deadlock situations.

- **Livelock** is the situation where a group of threads tries to get some resource, but these threads never succeed.
  - The idea is that they have a non-blocking wait.
  - Example: dining philosophers with non-blocking wait.

- Deadlocks and Livelocks can be total or partial.
Producer/Consumer problem

```c
struct CircularArray_t {
    int array[10];
    int head, tail;
    sem_t full, empty;
    sem_t mutex;
}

void init_CA(struct CircularArray_t*c) {
    c->head=0; c->tail=0;
    sem_init(&c->empty, 0);
    sem_init(&c->full, 10);
    sem_init(&c->mutex, 1);
}

void insert_CA(struct CircularArray_t *c, int elem){
    sem_wait(&c->mutex);
    sem_wait(&c->full);
    c->array[c->head]=elem;
    c->head = (c->head+1)%10;
    sem_post(&c->empty);
    sem_post(&c->mutex);
}

void extract_CA(struct CircularArray_t *c, int *elem){
    sem_wait(&c->mutex);
    sem_wait(&c->empty);
    *elem = c->array[c->tail];
    c->tail = (c->tail+1)%10;
    sem_post(&c->full);
    sem_post(&c->mutex);
}
```
Producers/Consumers: deadlock situation

- **Deadlock situation**
  - a thread executes `sem_wait(&c->mutex)` and then blocks on a synchronisation semaphore;
  - to be unblocked another thread must enter a critical section guarded by the same mutex semaphore!
  - So, the first thread cannot be unblocked and free the mutex!
  - The situation cannot be solved, and the two threads will never proceed.
- As a rule, **never insert a blocking synchronization inside a critical section!!!**
Producers/Consumers: correct solution

struct CircularArray_t {
    int array[10];
    int head, tail;
    Semaphore full, empty;
    Semaphore mutex;
}

void init_CA(struct CircularArray_t*c) {
    c->head=0; c->tail=0;
    sem_init(&c->empty, 0);
    sem_init(&c->full, 10);
    sem_init(&c->mutex, 1);
}

void insert_CA(struct CircularArray_t *c, int elem){
    sem_wait(&c->full);
    sem_wait(&c->mutex);
    c->array[c->head]=elem;
    c->head = (c->head+1)%10;
    sem_post(&c->mutex);
    sem_post(&c->empty);
}

void extract_CA(struct CircularArray_t *c, int *elem){
    sem_wait(&c->empty);
    sem_wait(&c->mutex);
    elem = c->array[c->tail];
    c->tail = (c->tail+1)%10;
    sem_post(&c->mutex);
    sem_post(&c->full);
}
Example of deadlock

Semaphore s1(1);
Semaphore s2(1);

void *threadA(void *)
{
    ...
    sem_wait(&s1);
    sem_wait(&s2);
    ...
    sem_post(&s2);
    sem_post(&s1);
    ...
}

void *threadB(void *)
{
    ...
    sem_wait(&s2);
    sem_wait(&s1);
    ...
    sem_post(&s2);
    sem_post(&s1);
    ...
}
Deadlock: graphical situation
Example with no deadlock (1)

Diagram showing a timeline with events such as get s1, release s1, get s2, release s2, and TB (Task Block). The diagram illustrates the scenario where deadlock is inevitable based on the sequence of operations.
Example with no deadlock (2)
Consumable and reusable resources

- **Reusable resources**
  - they can be safely used by **only one thread at time** and are **not depleted by the use**;
  - threads must request the resource and later release it, so it can be **reused** by other threads;
  - examples are processor, memory, semaphores, etc.

- **Consumable resources**
  - they are **created and destroyed dynamically**;
  - once the resource is acquired by a thread, it is immediately “destroyed” and cannot be reused;
  - examples are messages in a FIFO queue, interrupts, I/O data, etc.
Deadlock with reusable resources

- Bad situations can happen even when the resource is not “on-off”.
- Consider a memory allocator
  - suppose that the maximum memory allocable is 200 Kb.

```c
void * threadA(void *)
{
    request(80kb);
    ...
    request(60kb);
    ...
    release(140kb);
}
```

```c
void * threadB(void *)
{
    request(70kb);
    ...
    request(80kb);
    ...
    release(150kb);
}
```
Deadlock with consumable resources

void *threadA(void *)
{
    s_receive_from(threadB, msg1);
    ...
    s_send(threadB, msg2);
    ...
}

void *threadB(void *)
{
    s_receive_from(threadA, msg1);
    ...
    s_send(threadA, msg2);
    ...
}
Conditions for deadlock

- **Three conditions:**
  - **dynamic allocation of dedicated resources** (in mutual exclusion)
    - only one process at the time may use the resource
  - **hold and wait**
    - a process may hold allocated resources when it blocks
  - **no preemption**
    - the resource cannot be revoked
      (note: the CPU is a revokable resource)

- If the three above conditions hold **and**
  - **circular wait**
    - a **closed** chain of threads exists such that each thread holds at least one resource needed by the next thread in the chain
      then a deadlock can occur!

- These are **necessary and sufficient conditions for a deadlock.**
Deadlock: something that cannot be changed

- There is something that cannot be disallowed, because some behavior is forced by the interaction between the different concurrent activities
  - mutual exclusion
  - communication
- There is nothing we can do!
- But we can act on the mentioned conditions.
How to solve the problem of deadlock

- The basic idea is to avoid that one of the previous conditions holds.

- To prevent deadlock from happening we can distinguish two classes of techniques:
  - **static**: we impose strict rules in the way resources may be requested so that a deadlock cannot occur;
  - **dynamic**: dynamically, we avoid the system to enter in dangerous situations.

- Three strategies
  - 1) deadlock prevention (static),
  - 2) deadlock avoidance (dynamic),
  - 3) deadlock detection (dynamic).
1) Deadlock prevention: three methods

1) Take all the resources at the same time.

2) Preempt a thread and give the resource to someone else.

3) Resource allocation in a given order.
1) Deadlock prevention: conditions

1) Hold and wait

- we can impose the tasks to take all resources at the same time with a single operation;
- this is very restrictive! Even if we use the resource for a small interval of time, we must take it at the beginning!
- This reduces concurrency.
1) Deadlock prevention: conditions

- **2) No preemption**
  - this technique can be done only if we can actually suspend what we are doing on a resource and give that to another thread;
  - for the “processor” resource, this is what we do with a thread switch!
  - For other kinds of resources, we should “undo” what we were doing on the resource;
  - this may not be possible in many cases!
1) Deadlock prevention: conditions

3) Circular wait

- This condition can be prevented by defining a linear ordering of the resources.
- For example: we impose that each thread must access resources in a certain well-defined order.

```c
void *threadA(void *)
{
    ...
    sem_wait(&s1);
    sem_wait(&s2);
    ...
    sem_post(&s2);
    sem_post(&s1);
    ...
}

void *threadB(void *)
{
    ...
    sem_wait(&s2);
    sem_wait(&s1);
    ...
    sem_post(&s2);
    sem_post(&s1);
    ...
}
```
Example with deadlock prevention
Deadlock prevention: why this strategy works?

- Let us define an **oriented graph**
  - a vertex can be:
    - a thread (round vertex),
    - a resource (square vertex);
  - an arrow from a thread to a resource denotes that the thread requires the resource,
  - an arrow from a resource to a thread denotes that the resource is granted to the thread.

- **Deadlock definition**
  - a deadlock happens if at some point in time there is a **cycle** in the graph.
Deadlock prevention: graph

void *threadA(void *)
{
    ...
    sem_wait(&s1);
    sem_wait(&s2);
    ...
    sem_post(&s2);
    sem_post(&s1);
    ...
}

void *threadB(void *)
{
    ...
    sem_wait(&s2);
    sem_wait(&s1);
    ...
    sem_post(&s2);
    sem_post(&s1);
    ...
}
Deadlock prevention: theorem

- **If** all threads access resources **in a given order**, **then** a deadlock cannot occur.

**Proof:**
- by contradiction.
- Suppose that a deadlock occurs. Then, there is a cycle.
- By hypothesis all threads access resources in a given order, therefore, **each thread is blocked on a resource that has an order number greater than the resources it holds.**
- Starting from a thread and following the cycle, the order number of the resources should always increase. However, since there is a cycle, we go back to the first thread. **Then there must be a thread T that holds a resource Ra and requests a resource Rb with Ra < Rb.**
- This is a contradiction!
2) Deadlock avoidance

- This technique consists in monitoring the system to avoid deadlock
- We check the behavior of the system
- If we see that we are going into a dangerous situation, we block the thread that is doing the request, even if the resource is free
- That algorithm is called the Banker's algorithm
  - we skip it.
3) Deadlock detection

- In this strategy, we monitor the system to check for deadlocks after they happen.
  - We look for cycles between threads and resources.
  - How often should we look?
    - It is a complex thing to do, so it takes precious processing time.
    - It can be done not so often.
    - A good point to do that is when we lock (but it is computationally expensive).
    - Once we discover deadlock, we must recover.

- The idea is to:
  - Kill some blocked threads,
  - Return an error in the wait statement if there is a cycle.
  - That is the POSIX approach.
Recovery strategies

1. **Abort all threads**
   - Used in almost all OSs. The simplest thing to do.

2. **Check point**
   - All threads define **safe** check points. When the OS discovers a deadlock, all involved threads are restarted to a previous check point.
   - Problem: they can go in the same deadlock again!

3. **Abort one thread at time**
   - Threads are aborted one after the other until deadlock disappears.

4. **Successively preempt resources**
   - Preempt resources one at time until the deadlock disappears.
Dining philosophers problem (Dijkstra, Hoare)

- N filosofi siedono intorno ad un tavolo circolare al cui centro c'è un piatto di spaghetti e su cui sono disposti N piatti ed N forchette (in modo che ogni filosofo abbia un piatto, una forchetta alla sua destra e una alla sua sinistra).
- Il ciclo di vita di ogni filosofo è composto dall’alternarsi di 2 fasi mutuamente esclusive:
  - mangiare,
  - pensare.
**Dining philosophers problem (Dijkstra, Hoare)**

- Per mangiare: ogni filosofo ha bisogno di entrambe le forchette che prende una alla volta (in alcune versioni si parla di riso e di bacchette).
- Ogni filosofo, quando ha finito di mangiare, rimette a posto le due forchette che ha usato e comincia a pensare.
Dining philosophers

- Classical example of
  - threads synchronization with partially shared resources,
  - deadlock,
  - starvation.

- A possible solution is to brake the symmetry in the synchronization rules (they take forks in an increasing order, first on the left then on the right).

- An alternative solution: a manager thread (a waiter) for the shared resources.
Summary

- Concurrent Programming:
  - Synchronization,
  - Mutual exclusion.

- Shared memory systems:
  - Semaphores,
  - Mutex + condition variables.

- Private memory:
  - Message passing.