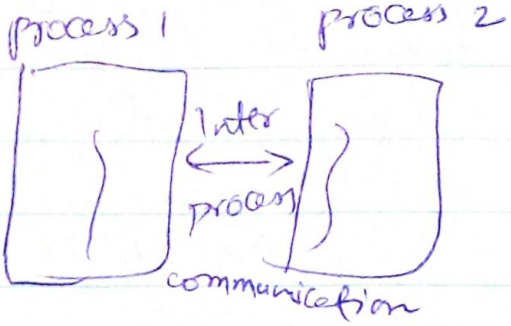
Concurrent Programming

There are two ways for concurrent pgming

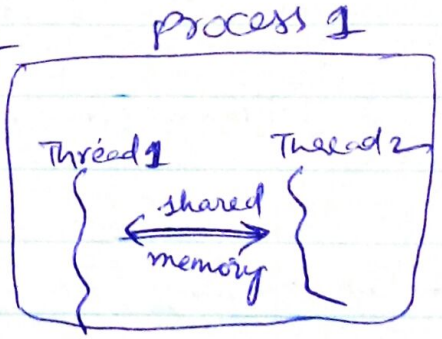
1) Multiprocessing:

- Processes communicate using files, pipes, messages queues



2) Multi-threading:

- Thread is a light-weight process.



Pros:-

- Fast to start
- Low overhead

Cons:-

- Difficult to implement
- Can't run on distributed systems.

→ * If threads instead of using shared memory uses some form of communication similar to

processes. (like IPC) It'll be easy to convert the code into multiprocessing program and run on a Distributed system.

Sample Program:

```
#include <iostream>
#include <thread>
using namespace std;

void function_1() {
    std::cout<<"Beauty is only skin-deep" << std::endl;
}
int main() {
    std::thread t1(function_1); // t1 starts running
    t1.join(); // main thread waits for t1 to finish
    // t1.detach(); // t1 will freely run on its own
                    // t1 will become a daemon process
    return 0;
}
```

** You can use either join (or) detach else it'll crash the pgm

* if we use detach() ⇒ "main" thread will run faster than "function_1" and prints nothing on the screen. as the 'main' thread which owns 'std::cout' is completed (terminated).

*

```
t1.detach()
t1.join()
```

✗ we cannot do this & it will crash the program.

- once a thread is detached, it cannot be joined again.
- but we can do, "t1.joinable()" method to check whether thread is joinable (or) not

⇒

```
t1.detach()
if (t1.joinable())
{
    t1.join()
}
```

✓ It'll not crash the program.

(17)

Thread Management:

- We should either "join()" or "detach()" before the object goes out of scope. (or) it'll terminate the program.

- wrap up the parent-thread using try { } catch { } block, so that incase any exception occurs, it'll not affect the joining of the child thread (or) crash it.

Pass by Reference in threads:

- By default variables will not be passed by pass by reference in threads. You have to pass in the below manner.

→ Functor -

```
class Fnctor {
    public:
        void operator () (string &msg) {
        cout << "t1 says: " << msg << "\n";
        }
};
int main() {
    string s = "where is this";
    std::thread t1(Fnctor(), s);
    try {
        cout << "from main: " << s;
    } catch (...) {
        t1.join();
        throw;
    }
    t1.join();
}
```

If we want to pass by reference we have to write std::ref(s)

* Threads cannot be copied from one thread to another, then can only be moved:

```
std:: thread t1(Fnctor());
std::thread t2 = t1 ✗
✓ std:: thread t2 = std:: move(t1);
```

* Print "ID" of the thread:-

```
cout << std:: this_thread :: get_id();
```

for thread, "t1"

```
cout << std:: t1.get_id();
```

* Over Subscription:-

. If there are more number of threads running than available no of CPU Cores, is called oversubscription

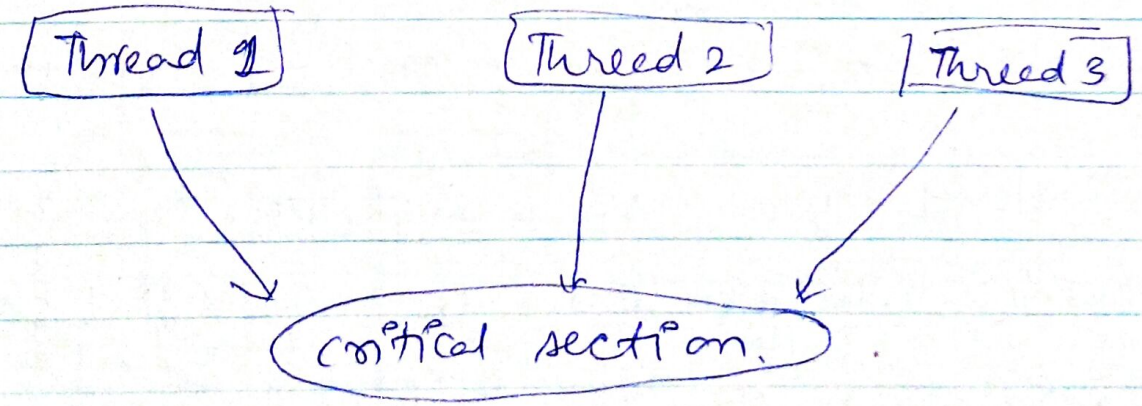
. It creates lot of context switching and generates a lot of overhead

(4) Threads

- Hardware Concurrency :
  - It gives the number which states how many threads can be run truly concurrently.
  - following function gives the number;
  - std::thread::hardware_concurrency()

RACE CONDITION AND MUTEX

- Concurrent accesses to shared resources can lead to race condition.



- A race condn occurs when two or more threads can access shared data and they try to change it at the same time. As a result, the values of variables may be unpredictable and vary depending on the timings of context switches of the processes.

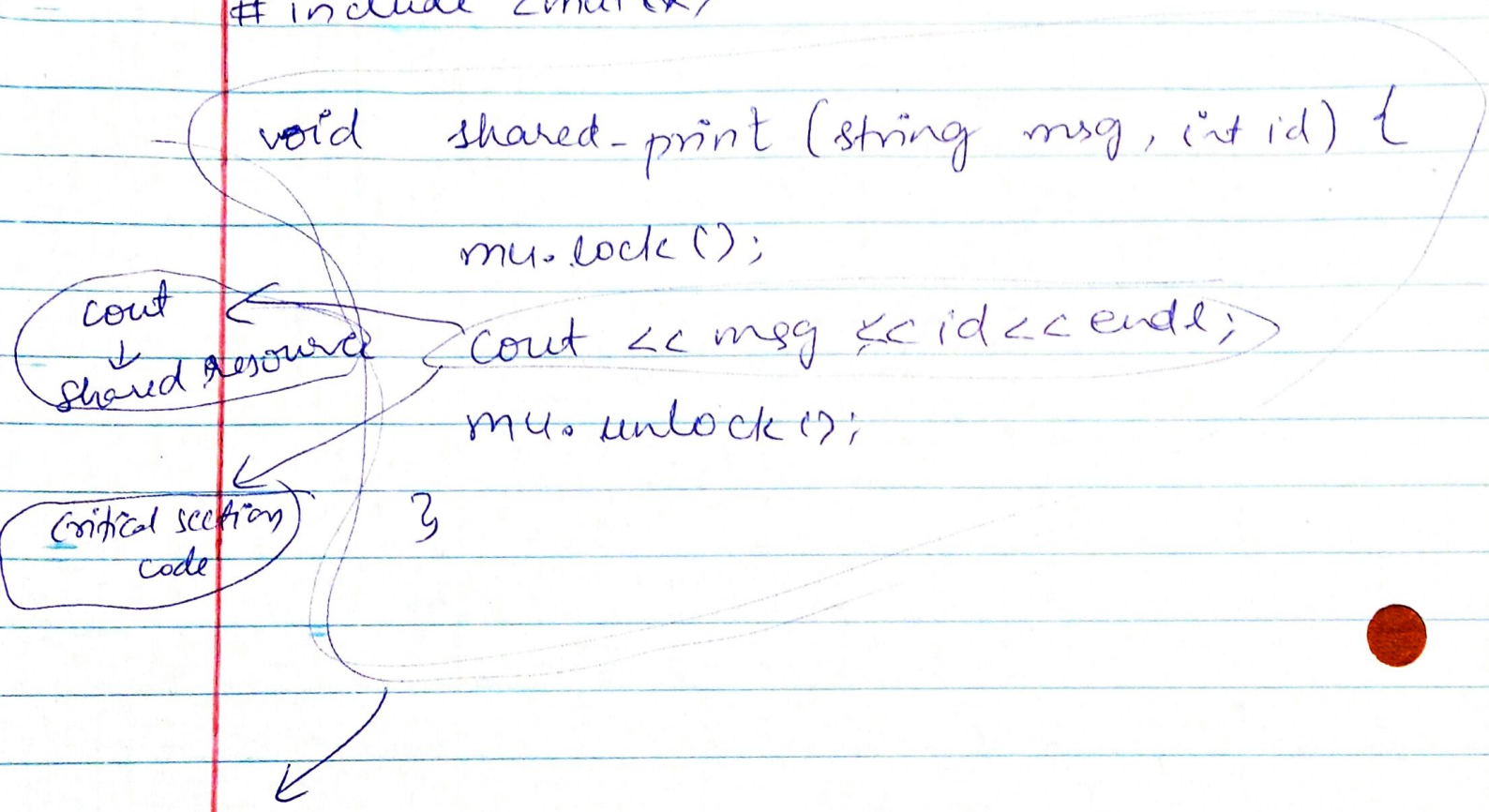
- Mutex is used to avoid race condition and synchronize the threads
- Protect, critical section code using mutex.

```
#include <mutex>

void shared_print(string msg, int id) {
    mu.lock();
    cout << msg << id << endl;
    mu.unlock();
}
```

cout → shared resource

Critical section code



* In the above C.S. code, if there is any exception, mutex remains unlocked.. So, instead we use "lock_guard" to lock the C.S code, which uses the RAII process.

```
void shared_print(string msg, int id) {
    std::lock_guard<std::mutex> guard(mutex);
    cout << msg << id << endl;
}
```

Mutex vs Semaphore:-

Mutex:

1) Mutex is a object owned by thread, so there is a ownership in mutex. i.e, before entering critical section, thread locks the mutex and owns it, and after existing it unlocks.

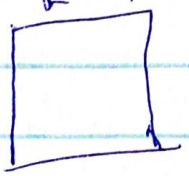
2) Mutex allow only one thread to access resource

→ Semaphore:

- It is a signaling mechanism.
- It allows a number of threads to access shared resources.
- for ex, if the semaphore = 5 at max, it means it can allow '5' threads into C.S, as every thread reduces the 'semaphore' by '1', when it enters into C.S.

MUTEX

ROOM



→ 1 key

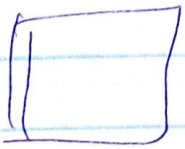
→ 2 persons

!!

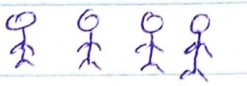
1 at a time by taking the key can enter the room

SEMAPHORE

ROOM



→ 3 keys



→ 4 persons

→ 3 people can enter the room by taking the key

In semaphore, signal() → S.value++

wait() → S.value--

Major differences between pthreads and C++11 threads

C++11 std::threads vs Posix threads

*) The 'std::thread' library is implemented on top of pthreads in an environment supporting pthreads

*) pthreads is a 'C' library, and was not designed with some issues critical to C++ in mind, most importantly object lifetimes and exceptions.

*) pthreads provides the function 'pthread_cancel' to cancel a thread. C++11 provides no equivalent to this

*) pthreads provides control over the size of the stack of created threads. C++11 does not address this issue.

4. C++11 provides the class 'thread' as an abstraction for a thread of execution

5. C++11 provides several classes and class templates for mutexes, condition variables, and locks, intending RAII to be used for their management.

6. C++11 provides a sophisticated set of function and class templates to create callable objects and anonymous functions (lambda expressions) which are integrated into the thread facilities

- RAII is at the center of the design of the C++11 thread library and all of its facilities.