a simple process i active application i,
with a single, main-thread, in a
typical, Linux-System :

In the above set-up, following points are true:

→ There is a single, main-thread, only

→ there is an u/s-stack, for the main-thread

→ There is a s/s stack, for the main-thread

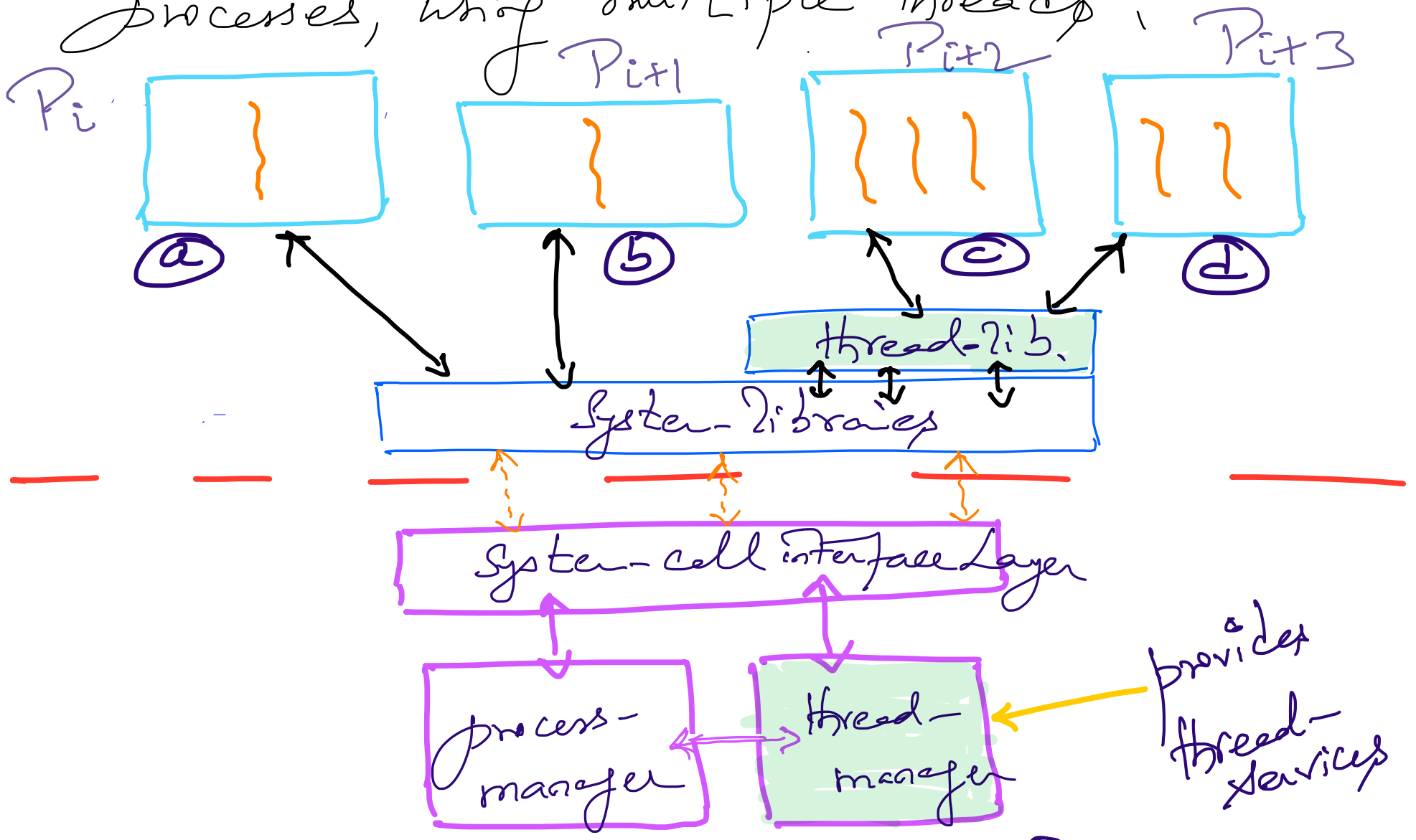
→ here, pdi does the job of process-descriptor and manages resources, using resource-descriptors

→ also, pdi does the job of thread-descriptor / tdi of main-thread

→ pdi is used, as a thread-descriptor, for blocking / unblocking / scheduling / dispatching

↳ this is thread-perspective

→ Many diagrams illustrates applications/ processes, using multiple threads :



→ in the above set-up, @ & ⓑ are single-threaded, applications/processes

→ ⓒ & ⓓ are multi-threaded, applications/ processes

→ Here is a thread-library, that provides APIs, for accessing thread services of OS — this is another, System-library

→ thread_manager provides core-services to user-space applications, which

sequoia thread- — it provides a frame-
services work

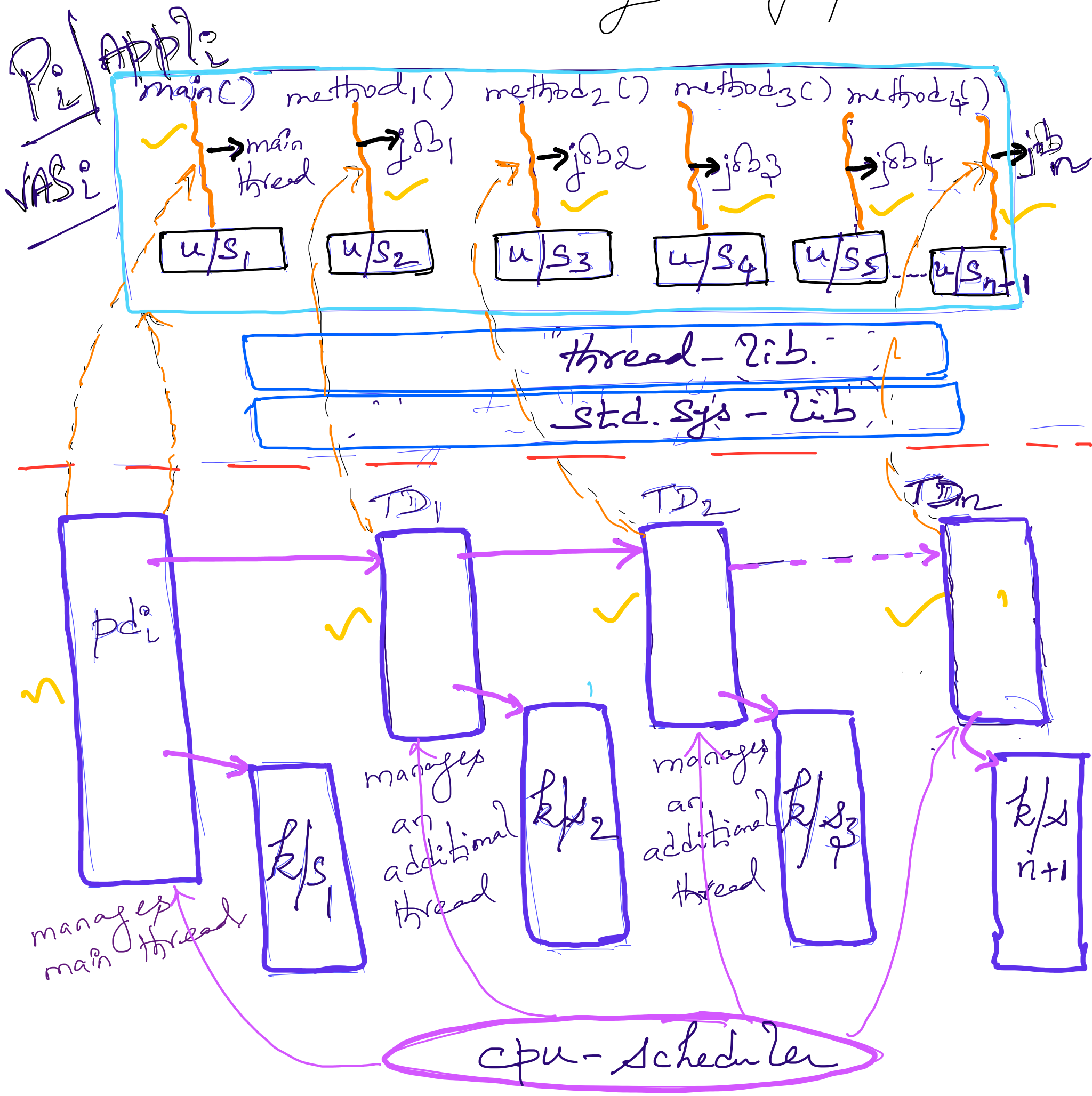
→ thread-library uses these core
thread-services of thread-manager
and also, provides its own
services, as well

→ applications use services of
thread library | APIs | frame-works

→ thread manager is tightly-coupled,
with process-manager and many
other core, kernel-components

→ conventional, multi-threading is
well-standardized, in modern, GPOS-
systems — similarly, multi-threading
APIs are very well standarized

→ Following set-up illustrates user-space & kernel-space parts of a conventional, multi-threading design/model



Follow are the key points, based on the above set-up:

→ every thread has an u/s-stack

→ every thread has a s/s-stack

→ every thread has its own, thread-descriptor (main-thread is different)

→ each td manages fields, like scheduling-parameters and thread-state

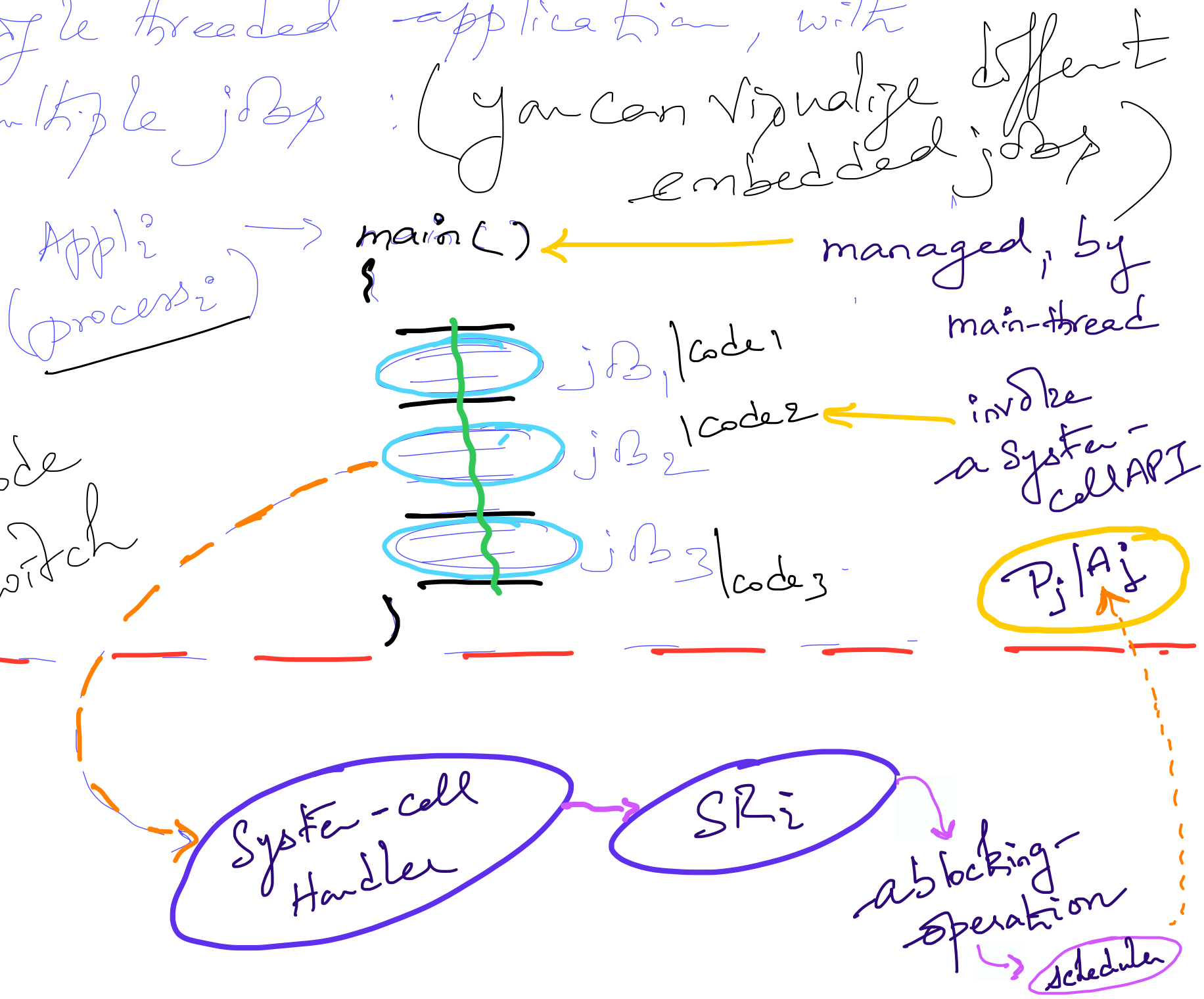
→ a pdi manages tds of threads of processi - pdi maintains a list of tds of processi

→ in the above model, pdi is used, as a process-descriptor, as well as td of

main - thread

→ we will see other,
multi-threading models,
in texts of multi-threading,
as well as ROS

Following is an illustration of a single threaded application, with multiple jobs : (you can visualize different embedded jobs)

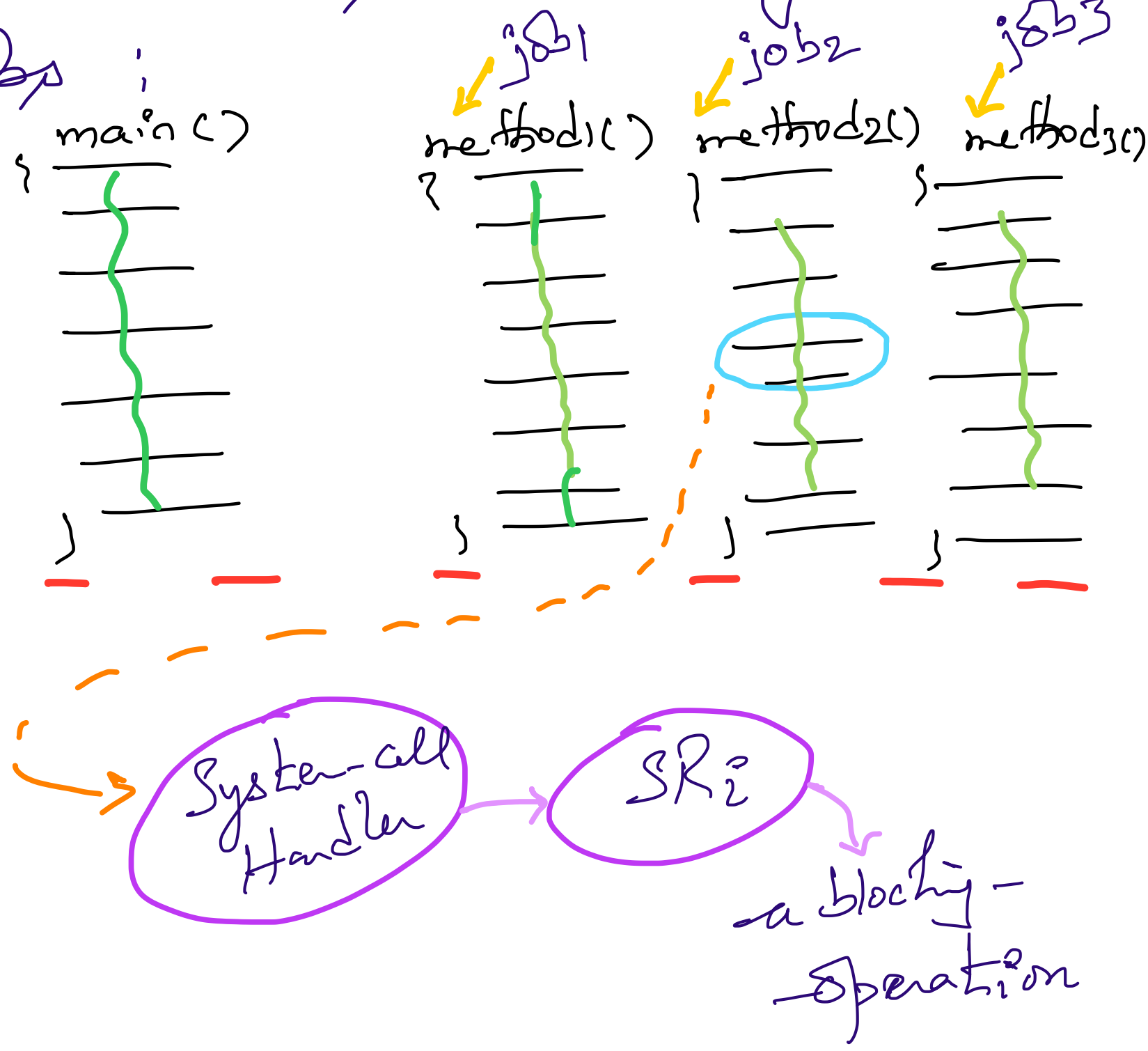


—> job2 invokes a syste-call API, which blocks main-thread of Pi

—> since this process is single-threaded, entire process is blocked — there are no more threads, in process i

→ So, scheduler will be invoked and another process will be selected & dispatched - actually, main-thread of process will be selected & dispatched

→ Following diagram illustrates a multi-threaded appli, in a process, where 3 threads are assigned 3 jobs :



→ in the above set-up, following are true:

→ main-thread is used manage other threads - it is a set-up

→ thread1 → method1 → job1

→ thread2 → method2 → job2

→ thread3 → method3 → job3

→ thread1, thread2, and thread3 are executed, concurrently - so, methods and jobs will be executed, concurrently

→ if method2 of thread2 invokes a blocking system-call API, td2 of thread2 will be blocked

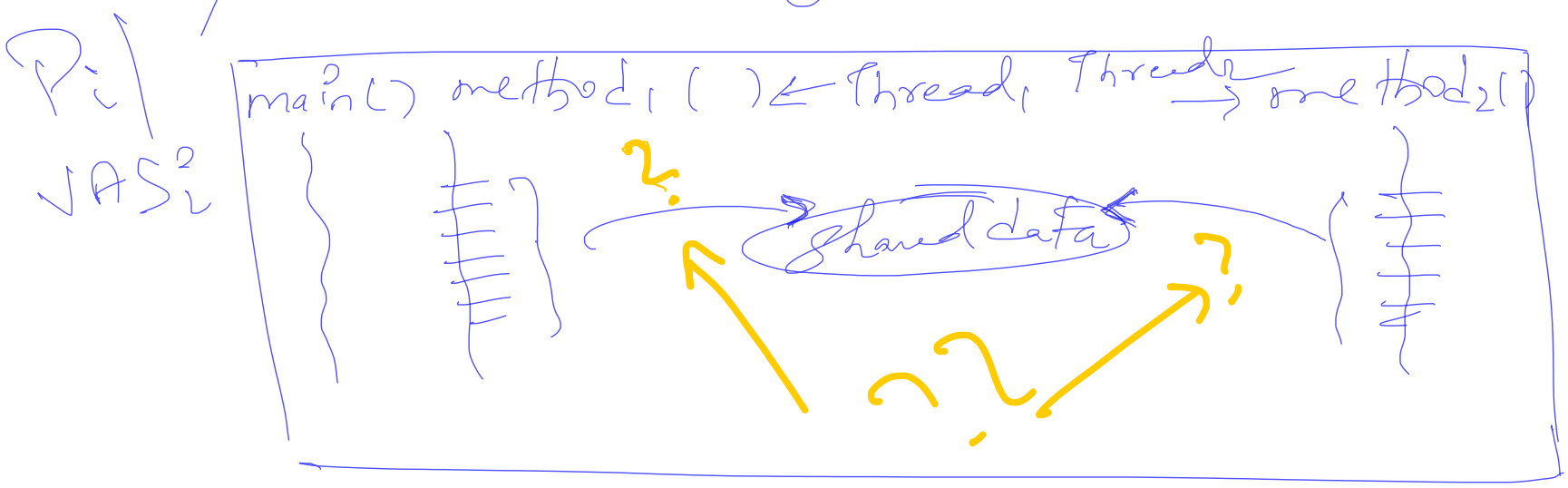
→ scheduler will be invoked and it will select a thread of this process, or a thread of another process

→ so, if thread 2 of process i is blocked, other threads of process i can still be scheduled & dispatched

→ multiple, concurrent threads of the above process, active application can efficiently handle multiple, concurrent jobs of the application

so, better responsiveness & possibly, better performance, in PiAi

Following diagram illustrates multi thread IPCs / data-exchange :

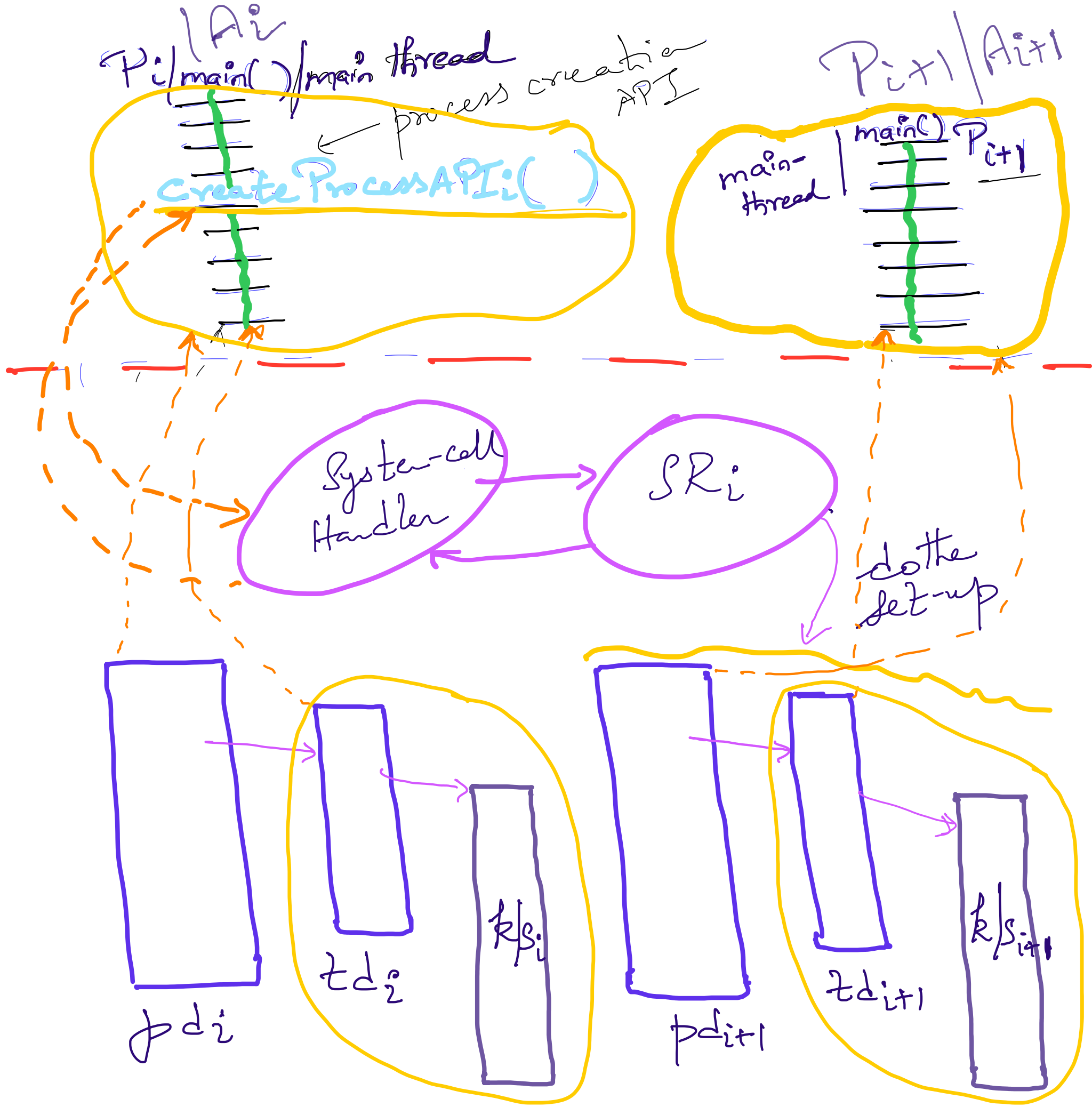


→ in the above set-up, two threads of P_i are sharing a piece of shared-data present, in a data-segment of VAS_i.

→ all the standard shared-data problems will be present, in this set-up, as well ??

→ in the above set-up, system call APIs are avoided, since all the data-exchange is, in user-space only

Following diagram illustrate process set-up and thread set-up, in chap6 | Charles-Crowley



→ in the above set-up, a new pd_{i+1} | pd_{i+1} | zd_{i+1} are set-up and zd_{i+1} will be added to an RQ

→ for low-level details, refer to chapter 6 slides of charles crawley

→ in the above set-up, an existing P_i creates a new P_{i+1}

→ in P_i, pd_i manages P_i

→ in P_i, td_i manages main thread/ main() of P_i / Appli

→ in P_{i+1}, pd_{i+1} is used to manage P_{i+1}

→ in P_{i+1}, td_{i+1} is used to manage main() / main-thread

main()

main-thread

$P_{i+1}|A_{i+1}|$

L

CreateThread( )

$P_{i+1}|A_{i+1}|$

main()

main-thread

R

method$_1$()

Thread$_1$

System-call Handler

SRi

do Set-up

pd_{i+1}

zd_{i+1}

$k|s_{i+1}$

zd_{i+2}

$k|s_{i+2}$

→ refer to chapter 6 slides of Cranley, for low-level details, hw-context set-up, and dispatching

→ we are creating a new-thread, an additional thread and this thread | tdif 2 will now manage and concurrently, execute method, main-thread | tdif 1 @ alongside main () method

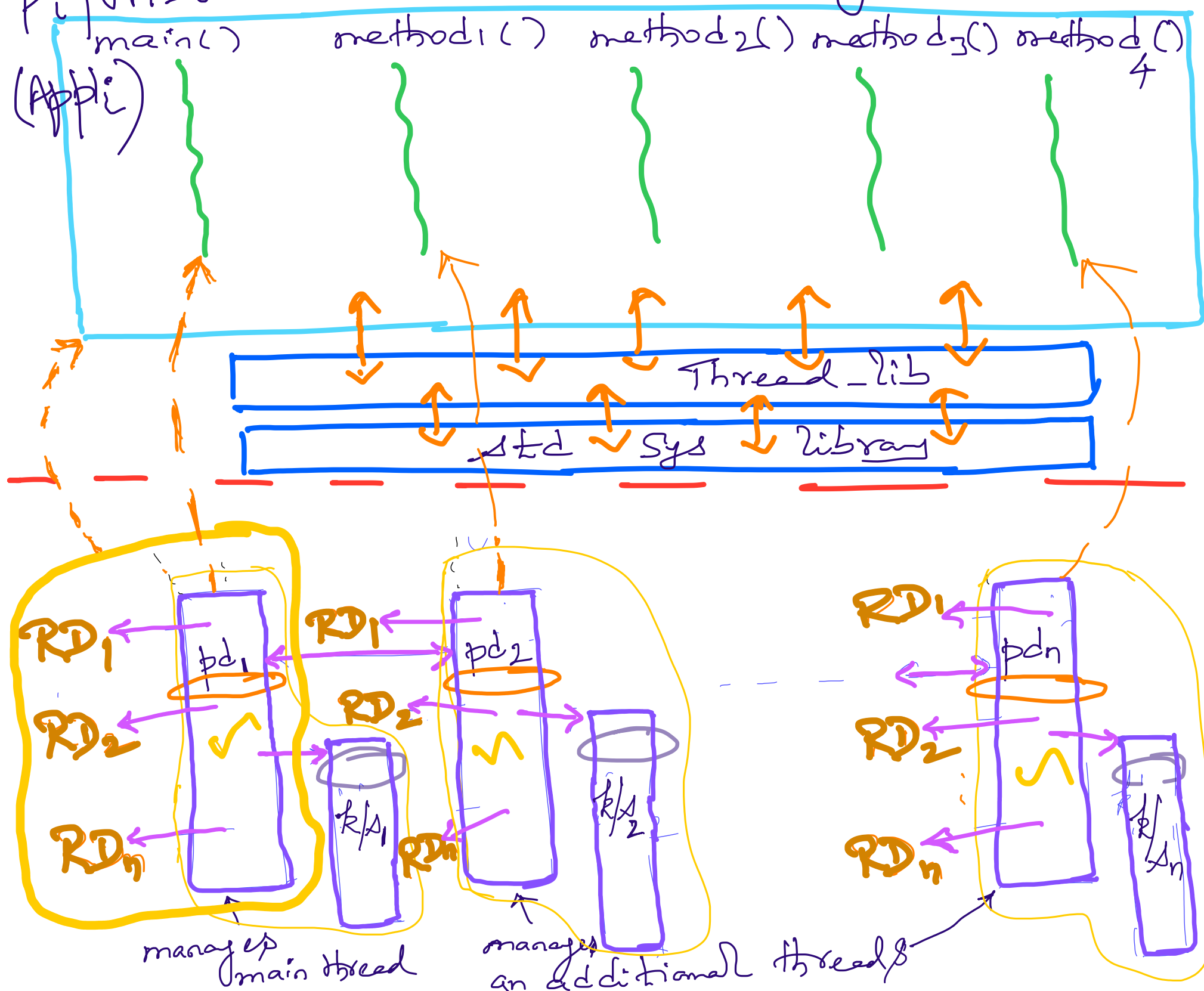
→ in the above set-up, we have achieved multi-threading, using an additional thread @ its thread-method

→ we can set-up many additional-threads, as described above, in

$P_i[t+1] \mid A_i[t+1]$

→ a typical, multi-threaded process in a Linux-System

$P_i[VAS_i]$ (App_i)



→ first pd of process$_i$ is used, as td$_1$ of main-thread of process$_i$ - it is the set-up

→ first pd also, manages process$_i$, using RD1, RD2, --, RDn, which are resource-descriptors

→ pd$_2$, pd$_3$, --, pdn will be used to manage additional, threads

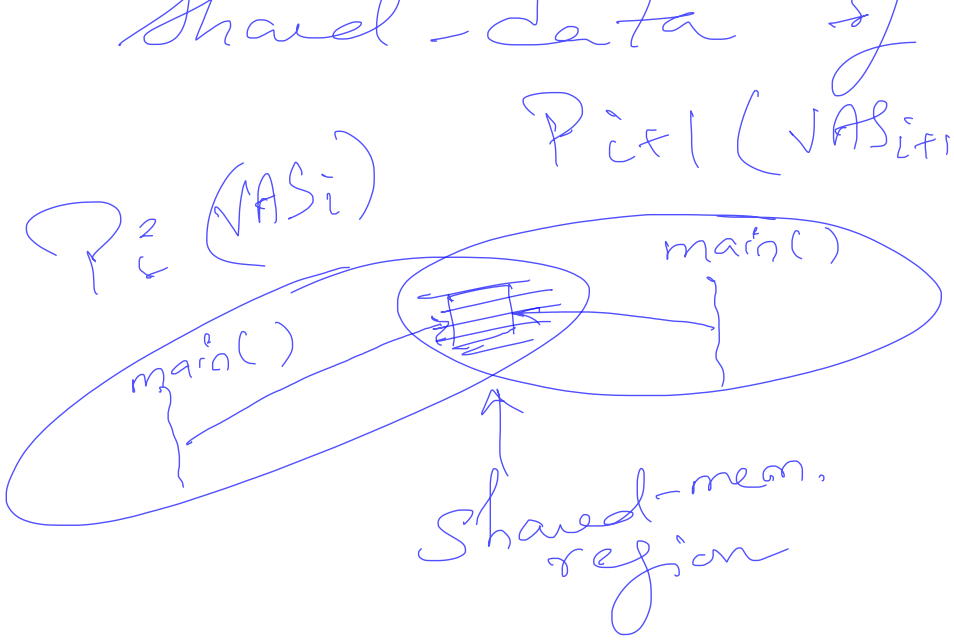
→ however, pd$_1$, pd$_2$, pd$_3$, --, pdn will be sharing the same VAS$_i$ and Resources of P$_i$, using the same set of resource-descriptors

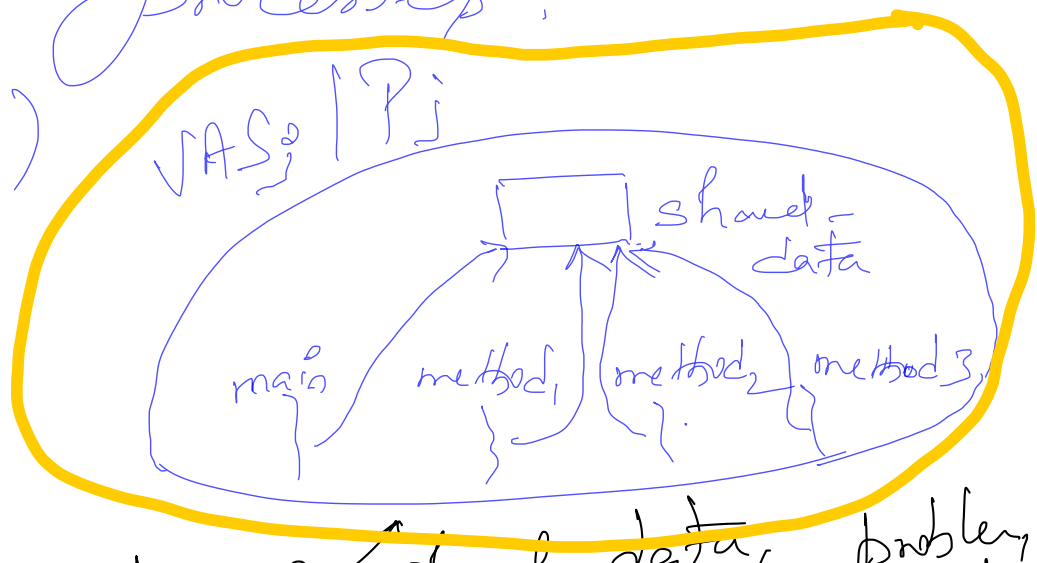
→ so, additional, pds are setting-up light-weight processes

→ so, effectively, pds are doing the jobs of tds

→ pds are also, used as tds

in the following illustration of multi-threading and shared-data, understand the scenario, which is similar to shared-data of processes!





typical shared-data problem in multi-threading

→ in the above set-up of P_i & P_{i+1}, there is a conventional shared-data problem, which involves 2 concurrent processes

→ in the above set-up of P_j (and its threads), there is a shared-data problem, which involves a set of concurrent threads
↳ due to shared data-segment (s)
↳ due to shared heap-segment

→ do, do we have similar problems, in the second set-up ?? YES

→ how do we solve, such shared-data problems ??

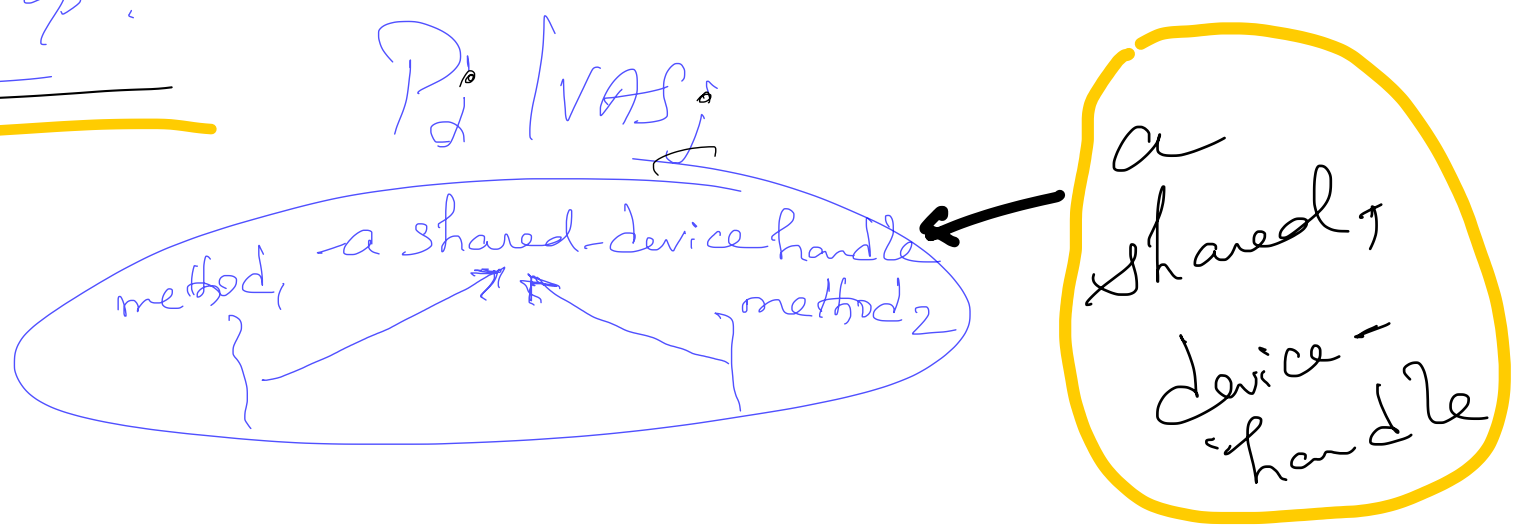
→ we can use a binary-semaphore to protect, such shared-data, critical sections

→ we can use a mutex-lock to protect, such shared-data, critical sections

→ there are certain merits and demerits ??

→ we will see more details, in threads @ ipcs, in 11_ipcs_2.txt

Following scenario is common, in embedded applications:



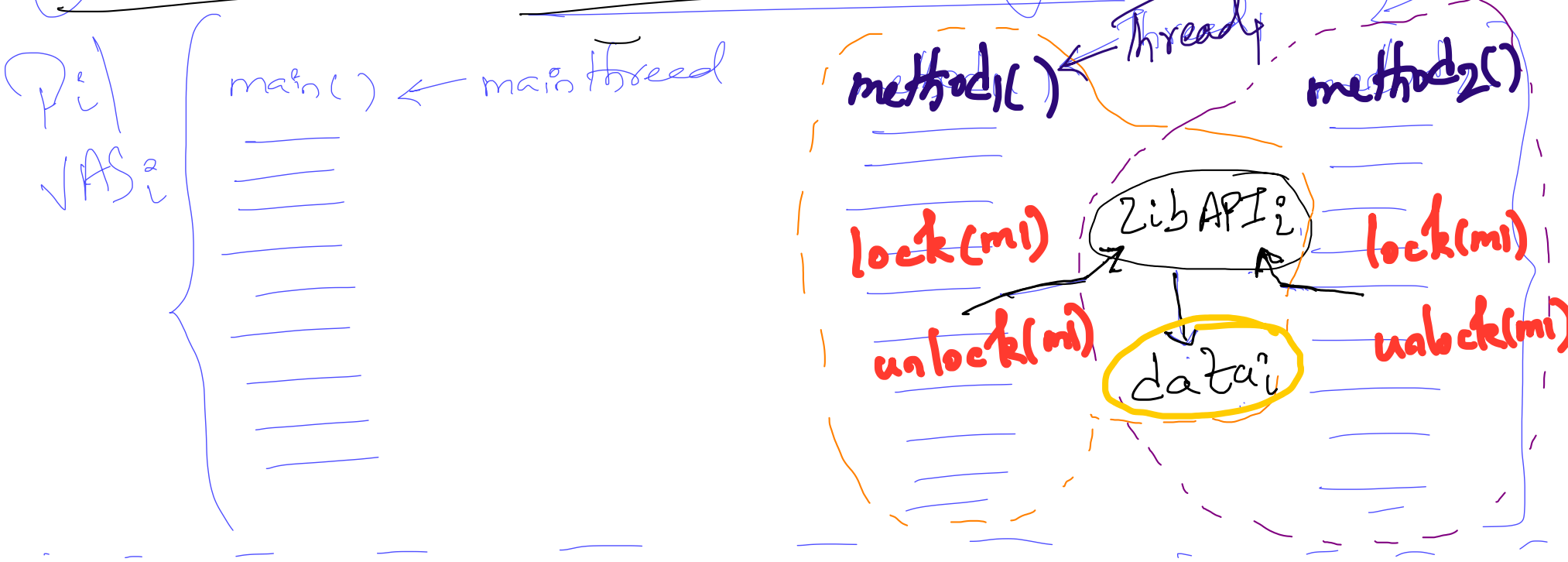
—> the above set-up is similar to shared-data problem, but here is a shared hw resource involved —

Otherwise, most of the issues of race-conditions still apply ??

—> we see specific scenarios, in embedded applications ??

by in vAS specific documents

Now is a different scenario of shared-data problem, in multi-threading ?



→ in the above set-up, thread1/method1 and thread2/method2 are accessing a single "library API1", concurrently — this library API1 is using data1 (shared data)

→ will be there be shared-data problems, in the above scenario ??

↳ YES

→ since libraies' data-segments are shared, the above scenario is a

form of shared-data, concurrent access problem — but not the same, as before (<the typical shared-data problem)

→ such library APIs are known as

non-reentrant lib APIs

→ this scenario is not an explicit shared-data, concurrent access — this is an implicit / hidden shared-data, concurrent access problem

→ such problems apply to any commonly used routines in library in multi-threading (including APIs)

→ What is the optimal solution to the above set of problems??

→ one possibility is to use a mutex lock around, such

Lib APIs and routines

→ Often possibility is to use reentrant lib APIs and routines only

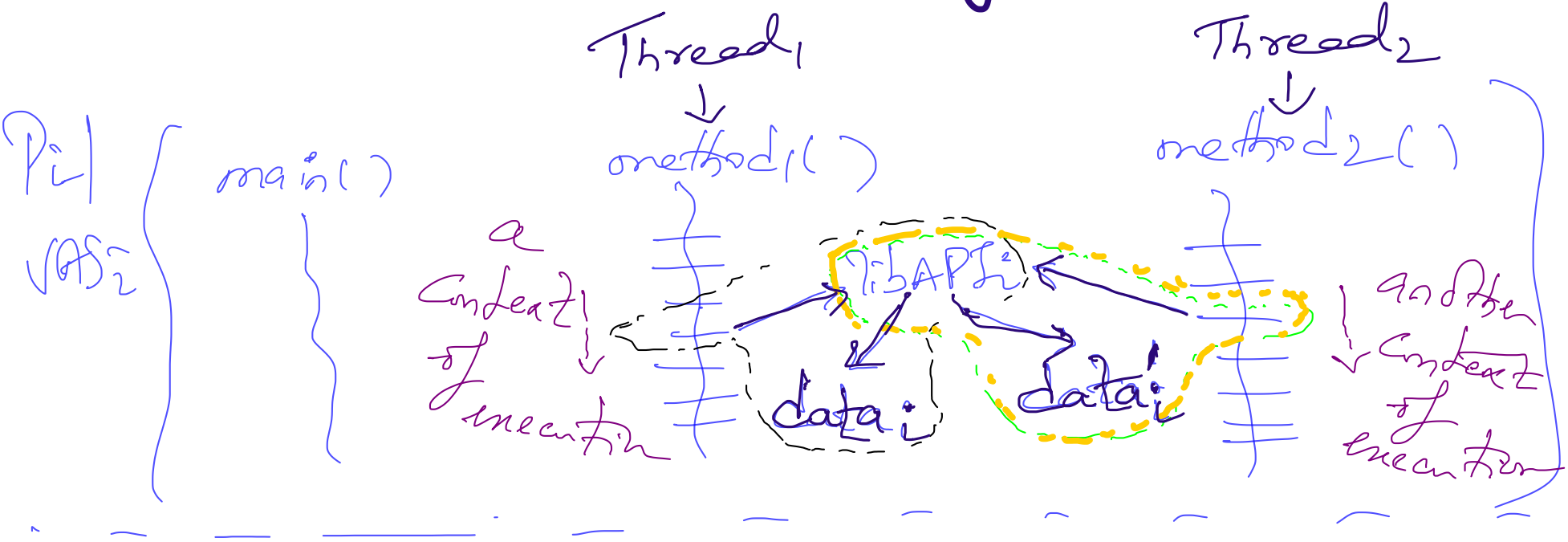
→ In a few remote cases, we may need to use non-reentrant methods/routines only (protected using locks)

→ refer to the following slides, for an introduction to reentrant lib APIs/routines ??

→ mutex locks are better forms of binary semaphore locks

→ refer to 11_ipcs_2.txt, threads and IPCs

Re-entrant, library-apis



→ in the above set-up, method1 / thread1 invokes libAPI1 and libAPI2 uses data1, in the context of execution

→ in the above set-up, method2 / thread2 invokes libAPI2 and latter uses data1, in this context of concurrent execution

→ do we expect shared-data, concurrent problems, in the above set-up ??

NO

→ Here is no shared-data, but independent, private data is used,

in each concurrent context of execution — now go back into txt file and complete ??

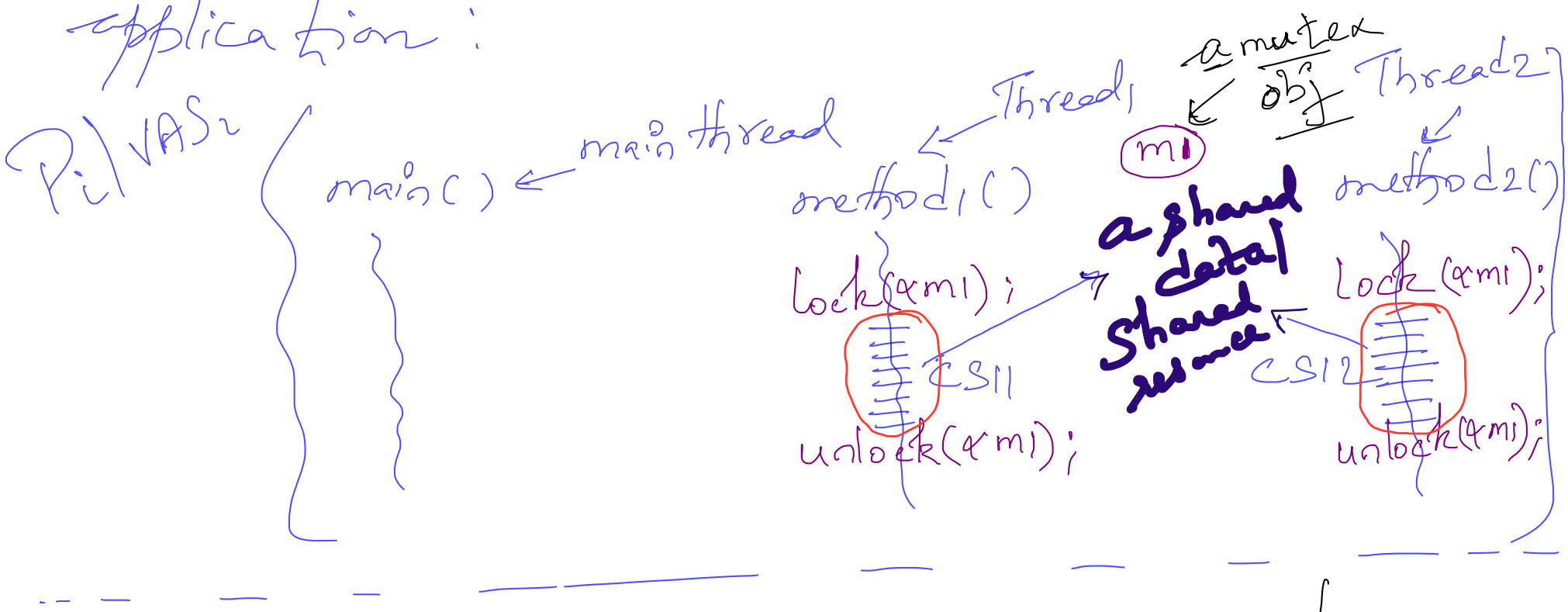
→ all the above points apply to any reentrant method / routine

system API	system call API	lib API

↳ Such a library-api is known as a re-entrant lib-api

→ following is an illustration of a re-entrant thread method used, in multiple threads :

Following is an illustration of usage of thread mutex, in a multi-threaded application:

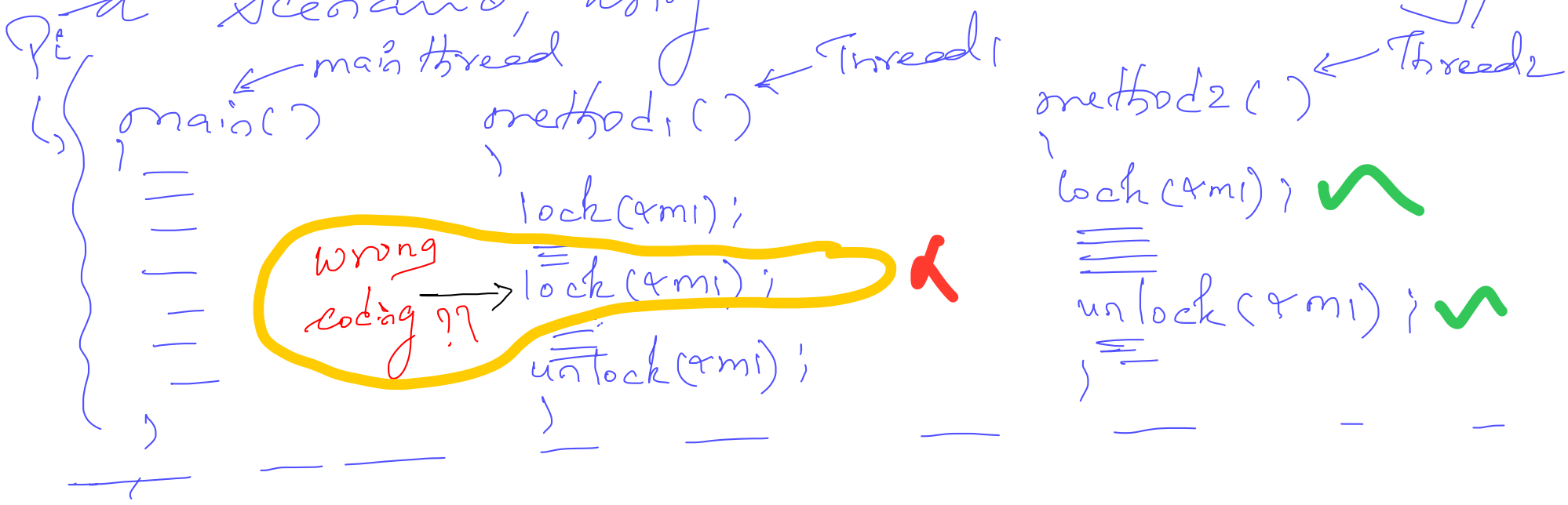


→ in the above set-up, m1 is a mutex object resident, in user-space, shared data-segment of threads

→ lock(&m1) & unlock(&m1) are operations, that will be mostly completed, in user-space - these operations are typically thread library APIs & refer to ll_pes-2.txt

atomic

Following is a typical illustration of a scenario, using a NORMAL mutex type



in the above set-up, due to programming errors, in thread1 / method1, after a lock(m1) operation, another lock(m1) operation is invoked — what will be the result of this erroneous operation ??

→ in the above set-up, since T_1 (method1) has already acquired mutex lock m1, the second lock(m1) will be forced to block T_1 (method1) and add td_1 to WQ of m1 — so, T_1 / td_1 is blocked

→. T_2 (method$_2$) will be blocked, when lock(am1) of method$_2$ is invoked — td_2 will be added to wq of m1

→ now, T_1 (method$_1$) cannot progress, as well as T_2 (method$_2$) cannot progress

→ What is this scenario, known as ??

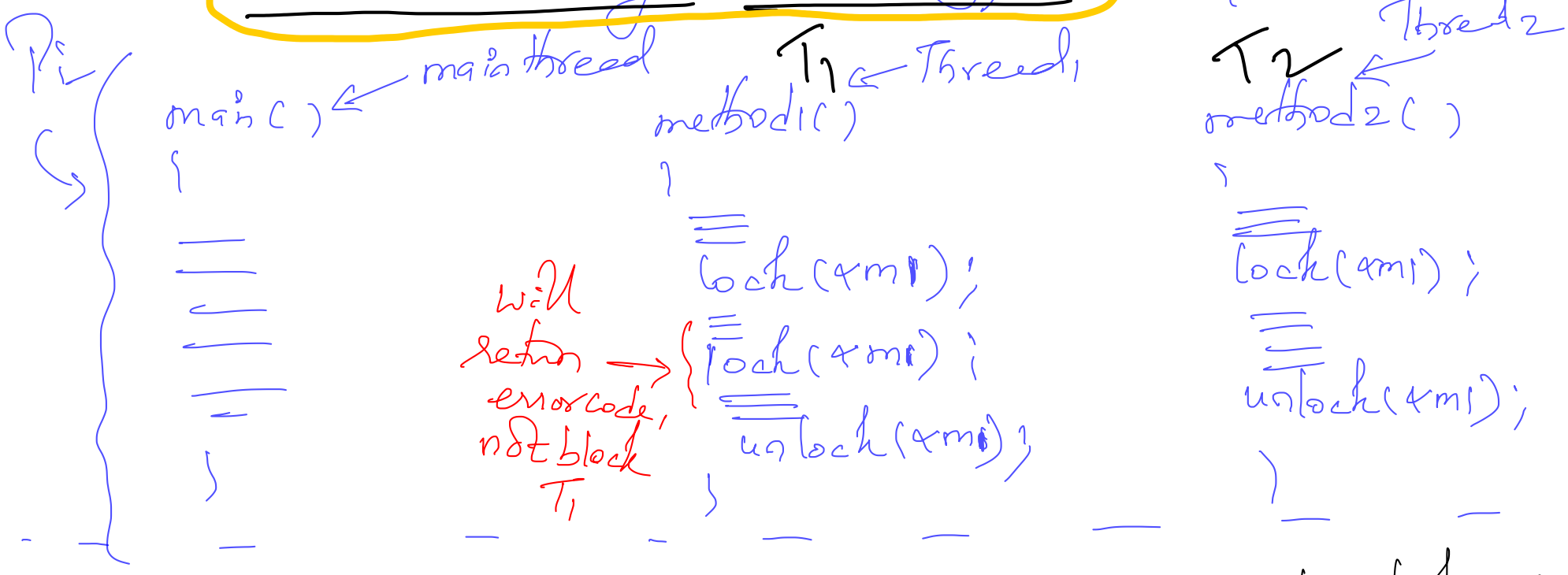
→ such a scenario is known as a dead-lock / deadly-embrace

→ two threads are dependent on a common set of resources/locks, but none can progress, so both cannot progress — Here is a dead-lock scenario

→ in effect, application is stalled ??

↳ This is a serious problem

in the above scenario, what will happen, if we replace a NORMAL mutex type, with an error checking mutex type ??



→ in the above scenario, when a second lock(&m1) is done on an error checking mutex, lock(&m1) operation returns an error code, and does not block T_1 (method1)

→ so, developer will be able catch the error & troubleshoot the problem

→ now, refer to 11_pcs_2.txt, for a detailed descriptions of diffent types of Mutex locks ??

Following diagram illustrates a typical multi-threaded application's VAS: