DISTRIBUTED COMPUTING

UNIT-3.

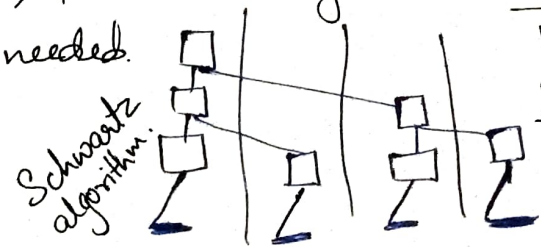
BLOCKS OF INDEPENDENT COMPUTATION: [Pg: 126]

* Parallel programs are more scalable when they emphasize blocks of computation: typically the larger the blocks the better, that minimize the inter-thread dependences.
* The use of large blocks is important because if we assume a fixed problem size, then each process will operate on a smaller block as the number of processes increases.
* There is generally a lower limit to the size of an effective block, but until the point is reached, continued scaling is possible. This point of limit should be determined during tuning phase in the programming process / when the prog. is ported to new h/w.

SCHWARTZ' ALGORITHM: [Pg: 127]

* proposes tree operations: + - reduce.
* main observation → tree structure should be used to connect the processes rather than to connect all the items.
* Approaches to compute +-reduce for P process & n values (P<n):
  (1) Introduce logical threads that implement a combining tree that literally encodes solution with n/2 logical concurrency.
  (2) have each process add n/P of the items locally, & then combine the P intermediate sums with P-leaf tree that connects the processes.
  ↳ faster to add nos. in a tight loop than to multiplex threads.

→ provides greater logical concurrency, but there is never more than P-fold actual parallelism, & the extra logical concurrency leads to extra work that is not needed.



The Reduce & Scan Abstractions: [Pg: 129]

* Why are reduce & scan important abstractions?

(1) Reduce → combines a set of values to produce a single value
↳ always needed as in parallel computation it is always necessary to compare/combine results produced by diff. threads, to summarize the computation/to control its execution.

(2) Scan → embodies the logic that performs a sequential operation in parts & carries along the intermediate results.
↳ Loop iterations often appear to be sequential but they can be solved using a scan, which admits more parallelism.

* Reduce & Scan are high level,
↳ Their use conveys info about program logic.

* Use of abstraction allows the implementation to be customized to the target machine.

* Examples of Generalized Reduces & Scans [Pg: 130]

1. Second smallest array element:
Solution is to keep two variables, smallest & next-smallest, each initialized to +infinity & to compare their values against each element of the array. If the array's value is smaller than the smallest, both variables are updated accordingly, otherwise if the array's value is smaller than the next-smallest, the value of next-smallest is updated

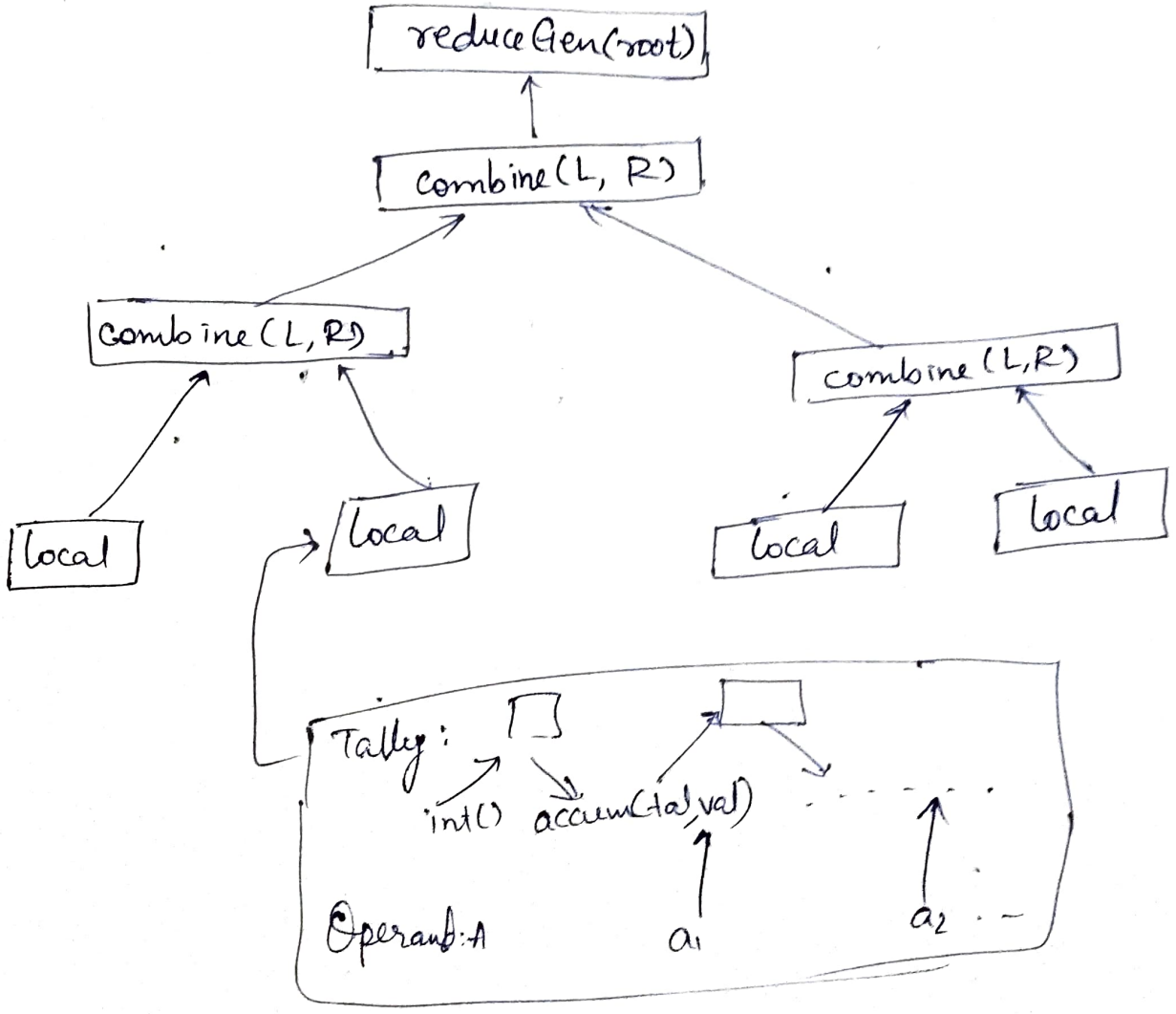
2. Histogram (k-way):
The solution assumes that min & max reduces have been used to compute the smallest & largest value in the range. Once ~~these~~

3. Length of Longest run of 1s
4. Index of first occurrence of x.
5. Team Standings.
6. Index of last occurrence of.
7. Keep the longest sequence of 1s.

asic Structure of Reduce & Scan: [Pg: 132]

1. init() → initializes "tally" in preparation for local computation
2. accum() → performs local accumulation of operand element into 'tally'.
3. combine() → composes intermediate 'tally' results from its two subtrees and passes the result to its parent.
4. r-gen() → takes global result & generates the final answer.
   ↳ it is separate for reduce & scan.

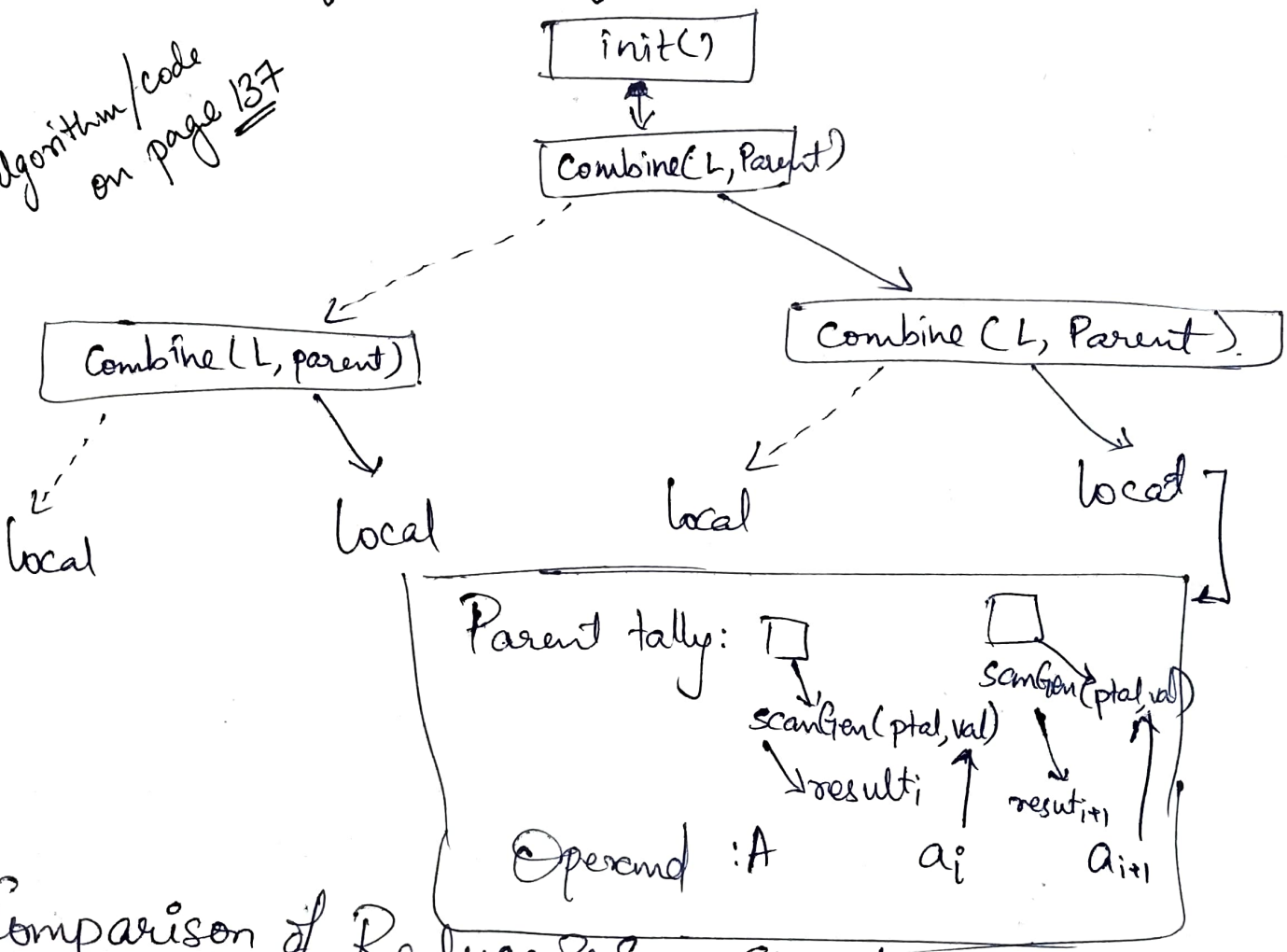
* Structure for Generalized Reduce [+/ A]: [Pg: 133, 134]



* Components of Generalized Scan: (Pg: 136)

→ The value at each process receives from its parent is the tally for the values that are left of the parent leftmost leaf.

algorithm/code on page 137



* Comparison of Reduce & Scan Structures:

→ 1. Global Storage (tally) is allocated to save the left child in scan during the up sweep.

2. Scan saves the left child values.

3. init func^n is called to provide the parent value to the root in scan.

Pg: 136.

Sir,
As attached.

Thanks & Regards
Tri

SIGNING WORK TO PROCESSES STATICALLY: (Pg:139)

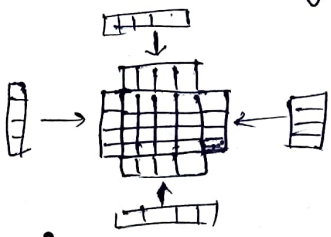
1. Block Allocation: (Pg:140)

* exploits locality → contiguous portions of a data structure should be allocated together on the same process.
  - ↳ 1-D arrays are assigned to processes in blocks of consequitive indices
  - ↳ 2-D arrays – allocating by 2-D blocks leads to efficient sol^{n}s.
    - ↳ makes more sense as block allocations can reduce comm^n.
* Stencil computations – computations that rely on neighboring values.

```
B[i,j] = A[i-1,j] +A[i,j+1] + A[i+1,j] + A[i,j-1]/4;
```

* A square like block of array values has the property that elements must be referenced by other processes for the stencil computation are on the edge, & as the size of block increases, the no. of edge elements grow much slower, reducing Communication cost.
* blocks have a surface area to volume advantage.

2. Overlap Regions: (Pg:142)



* type of s/w cache
* Advantages:
  1. once the overlap region is filled, all references in the computation are local, yielding large blocks of independent computation.
  2. computation uses the same index calculations for all references to the array, so it can be performed in a single loop nest with no special edge exceptions.
  3. by batching comm^n, effectively merge multiple cross-process dependences into b dependences, where the computation references data from b neighboring processes.
  4. reduce comm^n costs.

3. Cyclic & Block Cyclic Allocations : (Pg: 143)

* block allocations suffer from poor load balance when the amount of work is not proportional to the amount of data.
* Cyclic distribution allocates individual elements to diff. processes in round-robin fashion.
* ↳ they balance the load because they tend to distribute the hot spots across multiple processes.
  ↳ increases communication costs.

* Block cyclic allocation provides balance b/w locality & load balance
  ↳ blocks of elements are distributed cyclically.
  ↳ blocks dimensions doesnot need to divide the matrix's dimension.
  ↳ each process recieves blocks from throughout the matrix.
  ↳ Eg: Julia sets (based on Mandelbrot sets)

4. Irregular Allocations : (Pg: 146)

* Prior to executing large segment of code, data reference for its irregular data structures are inspected to identify those that are non-local & the process to which they are assigned. This technique is called 'inspector/executor'.
  ↳ all references to non-local process are batched and then fetched.
  ↳ Example of dynamic work allocation.

SIGNING WORK TO PROCESSES DYNAMICALLY (Pg:148)

* dynamic work allocation can balance the load when the amount of computational effort is not proportional to the amount of data.

1. WORK QUEUES: (Pg:148) ☆

↳ used when work is dynamically generated during the course of computation and the generalization to multiple processes is a natural one.

↳ simplest one → FIFO list of task descriptors.

↳ producer/consumer paradigm uses FIFO list.

* Collatz conjecture (3n+1 conjecture): trivial task expressed in work queue

$$a_i = \begin{cases} 3a_{i-1} + 1 & a_{i-1} \text{ odd} \\ a_{i-1}/2 & a_{i-1} \text{ even.} \end{cases}$$

↓ Pg: 148. Code on Page 149.

The conjecture is known to be true for all integers less than $3 * 2^{53}$

↳ The sol" postulates a work queue containing the next integers to test. For P processes, queue is initialized:

```
int i;
for (i=0; i<P; i++)
{ g[i] = i+1; }.
```

↳ The rationale for adding P is that P threads will be cheking integers at once, so advancing by P will skip those values that will be processed by other threads.

Variations in Work Queues: (Pg:151)

→ possible variations: FIFO, LIFO, randomized, priority

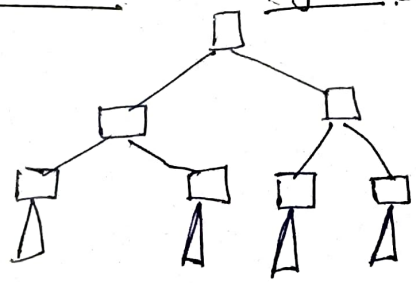
→ specific uses of work queues vary based on granularity of work

→ Small grain-size increases queue manipulation overhead increases the likelihood of contention.
→ large grain size increases chance of load imbalance.
→ Variable grain size, proportional to no. of elements in queue attempts to balance overhead and load ~~imbalance~~
→ Use multiple work queues, with each thread/process assigned to different queue / set of queues.
  ↳ reduces contention
  ↳ increase latency
  ↳ causes load imbalance → sol^n: work stealing

TREES: (Pg: 153) ☆

* represent hierarchical organizations
* kd-trees → used in graphics rendering & in gravitational simulations to partition space into hierarchical cells.
* Challenges of using trees in parallel programming:
1. trees are usually constructed using pointers & do not provide shared memory [pointers are local to only 1 process]
2. often implies extensive performance limiting communication.
3. irregular structure of some trees make it difficult to reason about communication & load balance

Allocation by SubTree: [Pg: 154]



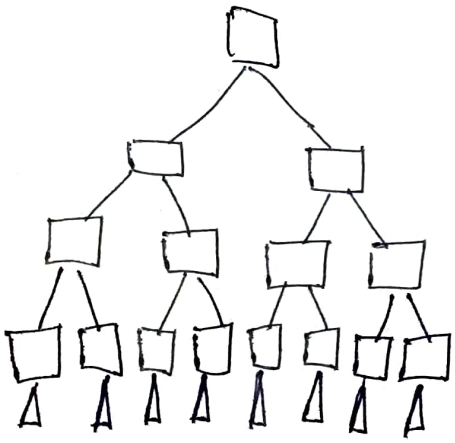
P = 4

→ each process is allocated one of the leaf sub trees along with a copy of the cap.
→ Such allocations work well when full tree is enumerated/when the nodes can be generated in level-order

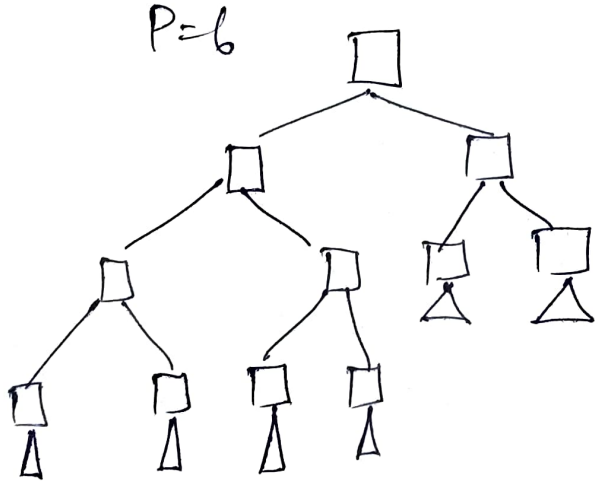
YNAMIC ALLOCATIONS : [Pg: 154 & 155] ★

→ some trees are allocated dynamically in an unpredictable manner.
→ Eg: search with alph-beta pruning.
→ may have uneven depths.
↳ in such cases, work queue can be used with queue entries representing unassigned nodes of tree.
↳ the results can either be communicated using shared memory or using some type of combining tree.

P=8



P=6



* Concurrent Memory Allocation: 152 Pg. ★

UNIT - 3

* Collatz Conjecture

```
float ef = 1.0;
int bestA = 1;
int head = 0;
forall (index in (0..P-1))
{ int a = 1
  float myEF = 1.0;
  int big, atest = 1;
  while (a < runSize)
  { exclusive
    { a = g[head];
      g[head] = a+P;
      head = [head+1] % P;
    }
    atest = a;
    big = a;
    while (atest != 1)
    { if (even(atest))
        atest = atest/2;
      else
      { atest = 3 * atest + 1;
        big = max(big, atest);
      }
    }
    myEF = big/a;
    exclusive
    { if (myEF > ef)
      { ef = myEF;
        bestA = a;
      }
    }
  }
}
```

* SCHWATZ CODE

```
int nodeVal[P];
forall (index in (0..P-1))
{ int tally;
  int stride = 1;
  while (stride < P)
  { if (index % (2*stride) == 0)
    { tally = tally + nodeVal[index+
                              stride
      stride = stride * 2;
    }
    else
    { nodeVal[index] = tally;
      break;
    }
  }
}.
```

* Concurrent Memory Allocation

Issue → implementing a memory allocator for a symmetric multiprocessor.

→ * A good allocator scales the no. of processors, avoids false sharing and uses virtual memory efficiently.

→ Solutions -

1. Use Single Heap that satisfies all requests. However, with multiple processors accessing this heap, contention for heap and its mutex becomes a bottleneck
2. use private heap per processor; there is no locking since there is no sharing. However suffers from unbounded memory allocation.
3. use private heap with ownership, each piece of memory is owned by the allocating processor. However, there is no sharing of free memory
4. Induce false sharing by allocating memory from the same cache line to diff. processors. Allocators donot perform padding.

→ Best Solution. : Hoard memory allocator

- applied on 2 principles

① Limit local memory usage - if a private heap becomes too large, it moves blocks to global heap. When a heap is empty, it first attempts to satisfy an allocation request by using memory from global heap.

② Manage Memory in large blocks - use of large blocks reduces false sharing and reduces contention for global heap.

- used in Linux, Windows, Solaris
- has bounded memory blowup & low synchronization costs.

* Static Work Allocation | Dynamic Work Allocation

Static Work Allocation	Dynamic Work Allocation
* processors are allocated work before execution.	processors are allocated work during execution
* some cases can have load imbalance	balances load well,
* some processes may remain idle	idle processes can be assigned additional work.
* Block, Overlap, Cyclic, Irregular	Work Queue eg: LIFO, FIFO
* Eg: 2D array processing	* Eg: Server processing client requests.

UNIT-4

Q. POSIX THREADS THEORY

Goals of Posix →

(1) the first is to provide sufficient details about the POSIX threads standard that readers can begin writing Pthread programs

(2) Second is to explain that while the facilities of the model are each individually fairly simple, the overall programming model presents a no. of important performance and correctness issues, some of which are quite subtle.

Q. Thread CREATION & DESTRUCTION

```
#include <pthread.h>
int err;
void main()
{ pthread_t tid[MAX];
  for(i=0; i<t; i++)
     err = pthread_create (&tid[i], NULL, count3s_thread, i);
}
```

In the for loop, thread is created and initiated..

pthread_create is invoked with 4 parameters :

1. A pointer to a thread ID, which points to a valid thread ID when thread returns successfully
2. The thread's attributes [NULL → default attributes]
3. A pointer to start func^n, which thread will execute on creation
4. Argument to pass to the start routine, [integer b/w 0 & t-1 that is associated with each thread].

pthread_create() :

```
int pthread_create(
    pthread_t *tid,
    const pthread_attr_t *attr,
    void *(*start_routine)(void *),
    void *arg
);
```

* Thread destruction can be done in 3 ways:
(1) A thread can return from the start routine
(2) A thread can call pthread_exit()
(3) A thread can be cancelled by another thread.

```
void pthread_exit(
  void * status;          // completion status.
);
```

Q. MUTUAL EXCLUSION

* when threads have shared memory, it is better to use mutex locks to ensure mutual exclusion.
* PTHREAD_MUTEX_INITIALIZER : mutex is assigned default attributes

Problem

```
for(i=start; i<start+len; i++)
    if (arr[i] == 3)
        count++;
```

Solution.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
...
if (arr[i] == 3)
{   pthread_mutex_lock(&lock);
    count++;
    pthread_mutex_unlock(&lock);
}
```

* one thread is executed at a time by locking of a mutex upon entry and unlocking on exit.
* when a mutex is unlocked, the thread blocked previously can acquire the mutex.
* Mutex Creation & Destruction.

```
int pthread_mutex_init (                    // initialize mutex
    pthread_mutex_t *mutex,
    pthread_mutexattr_t *attr);

int pthread_mutexattr_init (                // initialize mutex attribute
    pthread_mutexattr_t *attr );

int pthread_mutex_destroy (                 // destroy a mutex
    pthread_mutex_t *mutex);

int pthread_mutexattr_destroy (             // destroy mutex attribute
    pthread_mutexattr_t *attr);
```

* Serializability: only one thread can access the mutex at a time
* Correctness issues: it is an error to unlock a mutex that has not been locked and it is an error to lock a muter which is held by thread. The latter leads to deadlock.

SYNCHRONIZATION

* mutexes are sufficient to provide atomicity for critical sections, but in many situations, it is better to synchronize a threads behavior with that of some other threads.

* condition variable makes a thread to wait until some condition becomes true.

* Eg: producer - consumer problem.

* Pthread_wait()

```
int pthread_cond_wait(
    pthread_cond_t * cond,
    pthread_mutex_t * mutex);
```

```
int pthread_cond_timedwait(
    pthread_cond_t *cond,
    pthread_mutex_t *mutex,
    const struct timespec *abstime);
```

abstime: time out value

* Pthread_signal()

```
int pthread_cond_signal(
    pthread_cond_t *cond);
```

cond

```
int pthread_cond_broadcast(
    pthread_cond_t *cond);
```

* wakes up all waiting threads.
* Only one awakened thread will hold the protecting mutex.

SAFETY ISSUES:

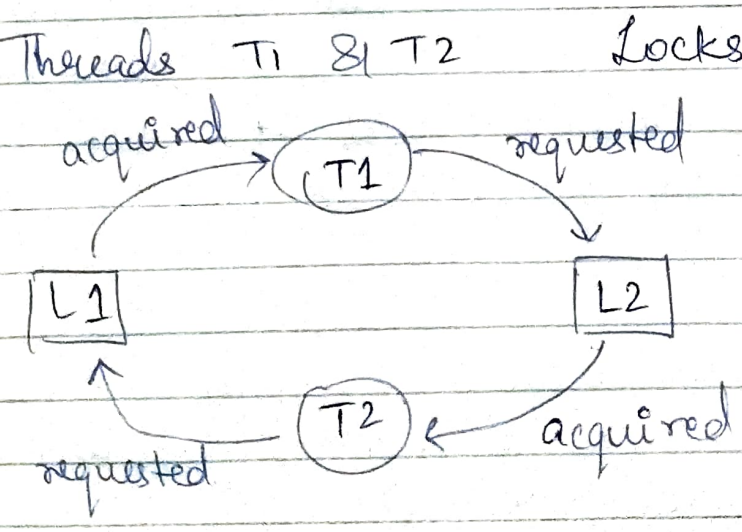
* Double Locking → occurs when a thread attempts to acquire a lock that it already holds.

* problems also occur if a thread accesses some shared variable without locking it or it acquires a lock and doesn't unlock it.

* Another issue is avoiding Deadlock.

DEADLOCK [Condition, Prevention & Avoidance].

* Conditions for deadlock:
  (i) mutual exclusion.
  (ii) Hold & wait
  (iii) No preemption
  (iv) Circular wait.



* Deal with Deadlocks → (i) prevent deadlocks.
  (ii) allow deadlocks to occur & find soln to break,

* Prevention of Deadlock:

(1) Lock Hierarchies:

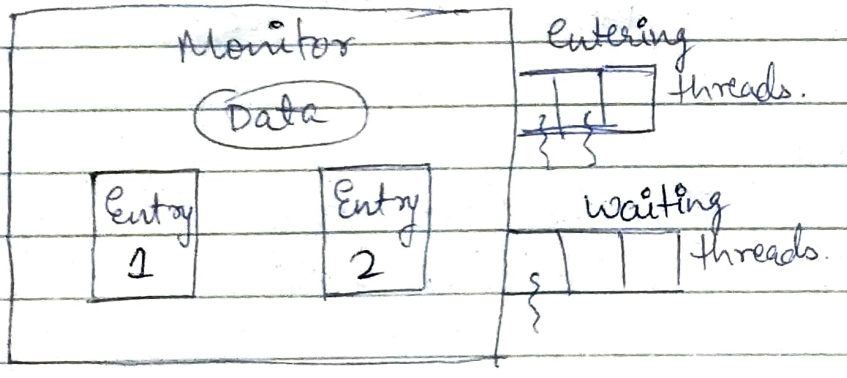
→ prevent cycles in resource allocation group.

→ mark the locks.

→ if a thread holds L1, L3 & L7, and requires L2, then it must release L3 & L7 first to acquire L2 which will eventually avoid deadlock.

→ it imposes an order on locks & by requiring that, all threads acquire their locks in the same order.

(2) Monitors:



* monitors encapsulates code & data and ensures mutual exclusion.
* monitor has well defined entry points, its data can only be accessed by the code that resides in it.

* only one thread can execute monitor's code at a time
* monitors can be implemented in C++.

(3) Re-Entrant Monitors:

→ monitors don't solve concurrency problem.

→ If a procedure tries to reenter a monitor, deadlock happens
↳ it has to unlock & then enter.

→ long procedures are put on wait after some time to maintain parallelism.