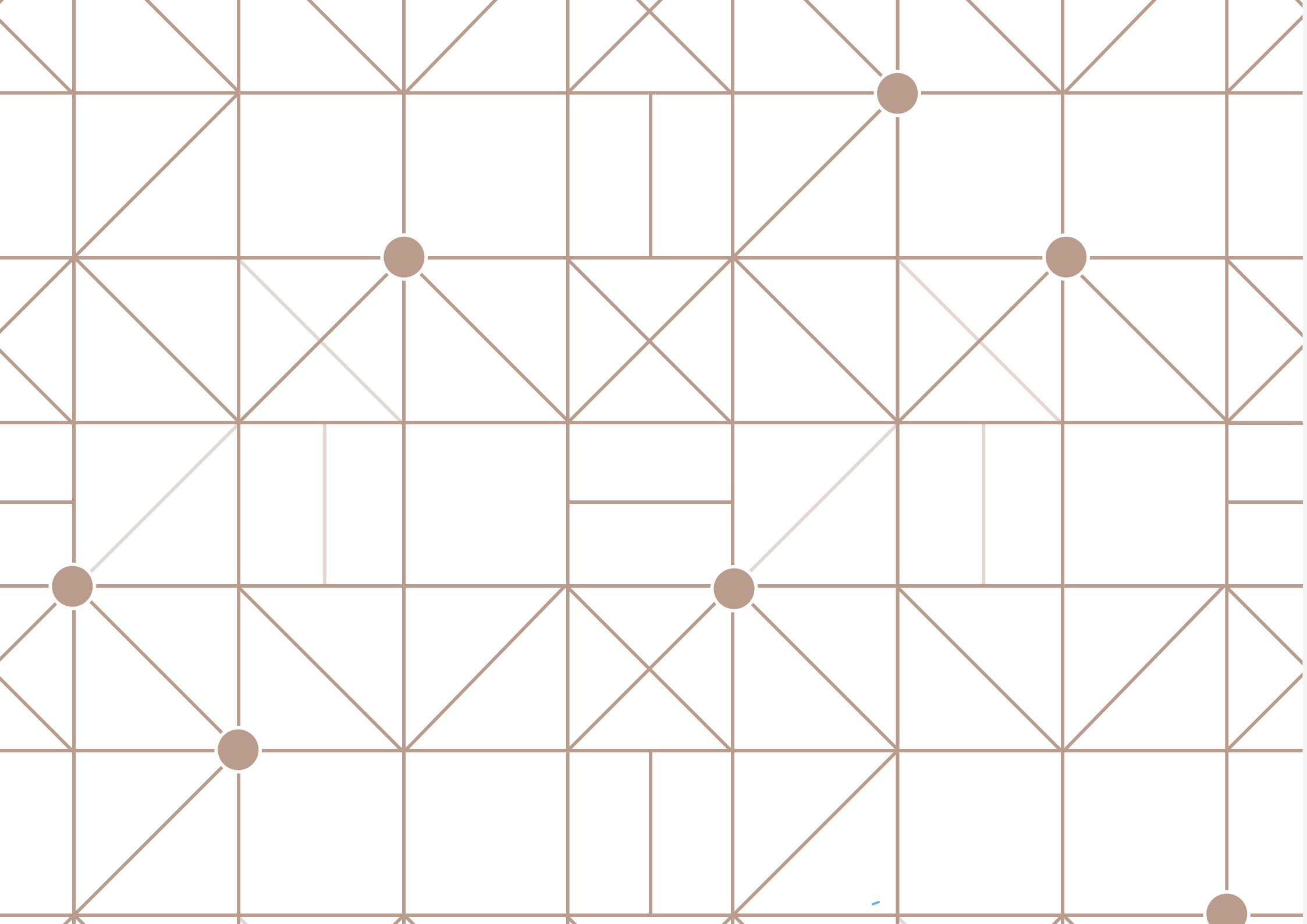
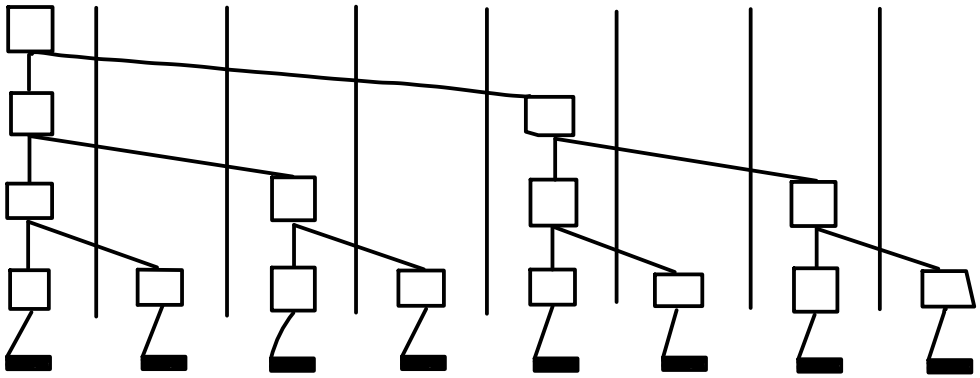


UNIT 3

SHWARTZ' ALGO

- An algo for tree operations — +- reduce
- His observation - tree structure should be used to connect the processes rather than the items.
- For a fixed no. of processes P & no of values n, P<n there are 2 approaches for computing +- reduce:
  (1) Introduce logical threads that implement a combining tree that literally encodes the soln. to summing in pairs w/ n/2 logical concurrency.
  (2) Have the processes add n/P of the items locally, & then combine the P intermediate sums w/ a P-leaf tree that connects the processes

- This algo illustrates the point — Unlimited Parallelism approach is inferior to the Scalable Parallelism approach.



All processes operate locally before using a tree to combine their results.
During the tree combination phase, the parallelism is uneven, as seen in the diagram

BASIC STRUCTURE OF SCAN & REDUCE

- As we need a scalable soln., we assume that the DS upon which the scan/reduce operates has been allocated in pieces to an unknown no. of processes.
- We use a Schwartz-like algo that computes locally & then uses a tree to complete the computation.
- Also assume the existence of a local variable in each process, tally, which is used to store intermediate results of the reduce & scan operations.
- with these assumptions, all reduce & scan ops can be implemented by defining variations of the following functions:

init() — Initializes the tally in prep. for the local computation.

accum() — Performs local accum. of the operand element into the tally.

combine() — composes interm. tally results from its 2 subtrees & passes the result to the parent.

x-gen() — Takes the global result & generates the final answer.

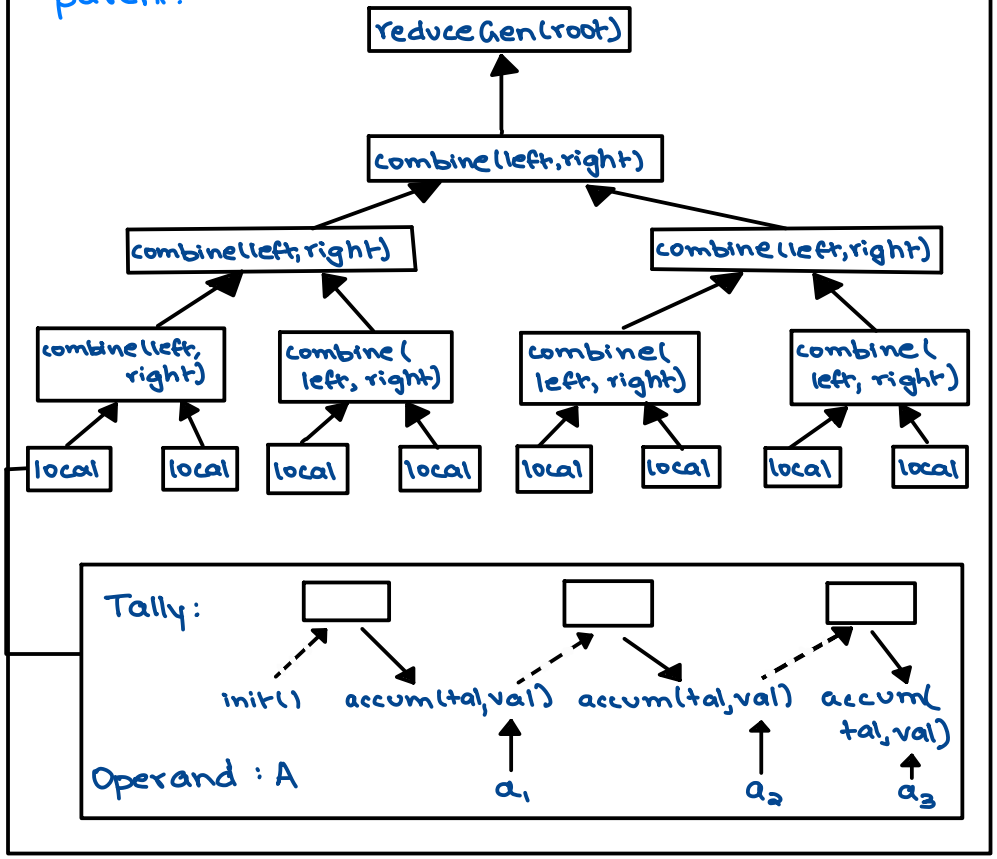
STRUCTURE FOR GENERALIZED REDUCE

init( )
sets up an integer (temp.) tally & initializes it to 0

accum(tally, val)
adds val, which is A[i] for some i to tally, & stores the result in tally.

combine(left, right)
adds left & right tally values & passes it to the parent.

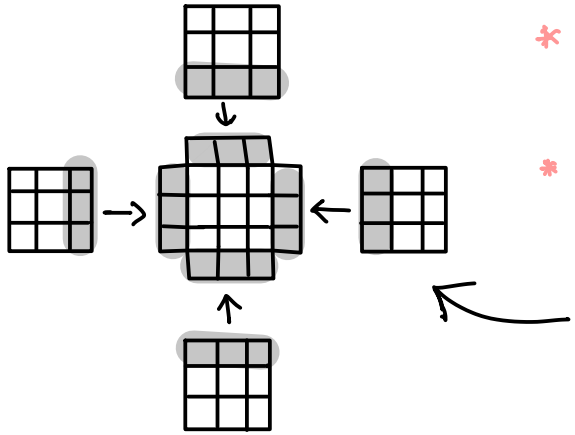
reduce-gen(root)
No-op for this simple operation.



ASSIGNING WORK TO PROCESSES STATICALLY

BLOCK ALLOCATIONS

- To exploit locality, for most computations contiguous portions of a data structure should be allocated on the same process
- 1D arrays are assigned to processes in blocks of consecutive indices.
- For 2D arrays — 2D blocks - consecutive indices in both dimensions.
- Blocks make more sense than allocating whole rows as they reduce communication.



OVERLAP REGIONS

- Type of software cache.
- Example → In many array-based computations, values are computed by referencing a fixed set of neighboring values — producing stencil computations — as their memory reference pattern is a stencil applied at each point.

  They're referenced as follows:

  * When allocating storage for the local portion, allocate extra space to hold the non-local values to be referenced. This extra space — overlap region.
  * Obtain these non-local values & save them in the overlap region.
  * Perform the computation on what are now all local data values.

Contd.

ADVANTAGES OF OVERLAP REGIONS

- All references in the computation are local, yielding large blocks of independent computation.
- Reduces commun. costs
- Merges multiple cross-process dependences into b dependences. (b → no. of neighboring processes)

CYCLIC & BLOCK ALLOCATIONS

- **Cyclic allocations** address the issue of load imabalance (amount of work is not proportional to the amount of data) by allocating elements to different processes in a round-robin fashion.
- However, cyclic distribution can increase communication costs by increasing the no. of cross-process dependences
- **Block cyclic allocations:**
  - → A block's dimensions doesn't have to divide the matrix's; the block is truncated where necessary.
  - → Each process receives blocks from throughout the matrix, so as the computation proceeds, there will be complete as well as incomplete portions on the processes.

IRREGULAR ALLOCATIONS

- Many algos use data structures other than arrays, like — unstructured grids & irregularly shaped grids typically made up of triangles.
- Assuming that interactions occur at grid boundaries, comm. b/w processes can be minimized by identifying large portions of the grid that have a large surface area to volume ratio. Techniques for such partitioning can be
  - based on geometric partitioning
  - " " graph theoretic "
- As the data references are irregular & often not known until runtime, it is inefficient to fetch the non-local values.
- To avoid this- **inspector/executor technique**

  ↓

  * Before executing an iteration or segment of code, the data references for its irregular data structures are "inspected", i.e, analyzed to identify those that are non-local & the process to which they are assigned.
  * All non-local references are batched & fetched.
  * Then, w/ all the data local, the executor performs the computation.

ASSIGNING WORK TO PROCESSES DYNAMICALLY

- In many cases, its impossible to adopt a fixed work assignment as new work is created during the computation.
- In other cases, static allocation leads to poor load balance

WORK QUEUES

→ It is a data structure for dynamically assigning work to processes.
→ They are particularly used when the work is dynamically generated during the computation.
→ The simplest work queue - FIFO list of task descriptors - newly created tasks are added to 1 end of the queue & tasks to be processed are removed from the other end.
→ Example - producer/consumer paradigm.

VARIATIONS OF WORK QUEUES

- Elements of the queue can be processed in FIFO, LIFO, randomized, or priority order.
- Specific uses of these queues can vary based on granularity of work.
  - L> A small grain-size increases queue manipulation overhead & increases the likelihood of contention; a large grain-size increases the chances of load imbalance
  - L> Use of variable grain size that is proportional to the no. of elements in the queue finds a balance b/w overhead & load imbalance
- Using multiple queues is a scalable approach that reduces contention but can increase latency.
- Multiple queues causes load imbalance. Soln. → Work Stealing - ler threads retrieve work from queues to which it wasn't assigned.

TREES

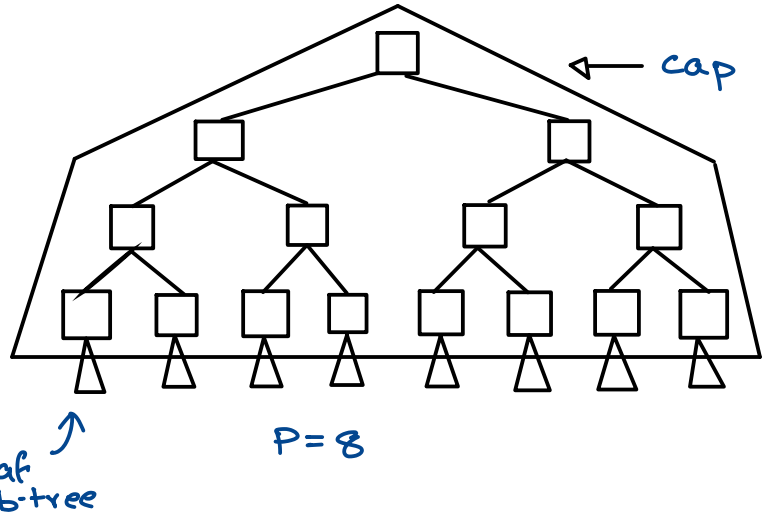
* Data structures that represent hierarchical organizations.
* They present challenges in parallel computing:
  - Trees are constructed using pointers & in languages that don't provide shared memory, pointers are local only to 1 process.
  - We use trees for their dynamic behavior, but this behavior often implies extensive performance-limiting communication.
  - The irregular structure of some trees can make it difficult to reason about comm. & load balance

ALLOCATION BY SUBTREE

* We could replicate the top of the tree, the "cap", & assign a copy of it & a sub-tree to each of the processes.
* Cap allocation is effective as it is small relative to the rest of the tree.

contd

contd.



- As the root & its immediate descendents are available on all processes, interactions that propagate through the root can use local data to identify the correct destination subtree.
- As computation proceeds, changes in the cap must be maintained coherently across processes.

DYNAMIC ALLOCATIONS

- Some trees are allocated dynamically in an unpredictable fashion, other trees are explored to uneven depths.
- Example → A search w/ alpha-beta pruning (in game searches) will prune away portions of the hierarchical search space based on previously seen results, leading to a search tree w/ leaves at different depths. In such cases a work queue can be used.

UNIT 4

POSIX THREAD CREATION

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

ARGUMENTS:

→ A thread ID of the successfully created thread

→ The thread's attributes; the NULL value is default

→ The function that the thread will execute once it is created.

→ An argument to the start_routine

Return value - 0 if successful, error code from <errno.h> otherwise.

Notes: use a structure to pass multiple arguments to the start routine.

MUTEX CREATION & DESTRUCTION

```
int pthread_mutex_init(
   pthread_mutex_t *mutex);

int pthread_mutex_destroy(
   pthread_mutex_t *mutex);
```

POSIX THREAD TERMINATION

```
pthread_exit(
   void *status
);
```

ARGUMENTS:

→ The completion status of the thread that has exited. This pointer value will be available to other threads.

Return value: none

Notes: When the thread exits by returning from the start_routine, the thread's completion status is set to the start_routine's return value.

SYNCHRONIZATION (mutex)

- A condition variable allows threads to wait until some condition becomes true, at which point one of the waiting threads is nondeterministically chosen to stop waiting.

BOUNDED BUFFER PROBLEM

→ w/ Producers & Consumers

→ when the buffer is empty, the consumers must wait. when " full, producers " ".

MUTUAL EXCLUSION

- To allow threads to interact constructively, we need methods for coordinating their interaction.
- In particular, when 2 threads share access to memory, its useful to employ a lock, mutex, to provide mutually exclusive access to the variable.
- To use the mutex, its address is passed to the lock & unlock routines
- Bracket all critical sections - i.e; that code must be executed atomically by only a thread at a time - by the locking of the mutex upon entrance & unlocking upon exit.
- When a mutex is unlocked, 1 of the threads that was blocked while attempting to acquire the lock will become unlocked & will be granted the mutex.

```
int pthread_mutex_lock(          // Lock a
   pthread_mutex_t *mutex);      mutex
int pthread_unlock(              // Unlock a
   pthread_mutex_t *mutex);      mutex
```

SAFETY ISSUES (MUTEX)

- Double-locking: A problem which occurs when a thread attempts to acquire a lock that it already holds.
- Problems also arise if a thread accesses some shared variable wlout locking it first, or if a thread doesn't relinquish a lock.
- Important problem - DEADLOCK.

DEADLOCK:

The necessary conditions for deadlock:

* Mutual Exclusion - a resource can be assigned to at most 1 thread
* Hold & wait - threads hold some resources & request other resources.
* No preemption - a resource that is assigned to a thread can only be released by the thread that holds it.
* Circular wait - a cycle exists in which each thread waits for a resource that is assigned to another thread.

Approaches to dealing w/ deadlock:
(1) prevent deadlocks
(2) allow them to occur, but detect & break them.

POSIX threads don't provide a mechanism for breaking locks, ∴ there's only deadlock avoidance here.

- The bounded buffer problem can be solved using 2 condition variables — nonfull & nonempty.
- Since multiple threads will be updating these condition variables, we protect their access w/ a mutex.

① **LOCK HIERARCHIES**:
→ Preventing cycles in the resource allocation graph can prevent deadlocks.
→ We can prevent cycles by imposing an ordering on the locks & by requiring that all threads acquire their locks in the same order
→ A lock hierarchy requires programmers to know a priori what locks a thread needs to acquire.

② **Monitors**:
→ A monitor encapsulates code & data & ensures mutual exclusion.
→ A monitor has a set of well-defined entry points, its data can only be accessed by code that resides inside the monitor, & only one thread can execute the monitor's code at any time.
→ The limited no. of entry points facilitates the preservation of invariants.
They (↑) are properties that are assumed to be true upon entry & that must be restored upon exit.

③ **Re-Entrant Monitors**:
→ while monitors help enforce a locking discipline, they don't solve all concurrency problems.
→ Eg: if a procedure in a monitor attempts to re-enter the monitor by calling an entry procedure, deadlock will occur.

PERFORMANCE ISSUES

READERS & WRITERS EXAMPLE: GRANULARITY ISSUES

- There are different granularities for using condition variables.
- Consider a resource that can be shared by multiple readers or accessed exclusively by 1 writer.
  To coordinate access to such a resource, we can implement 4 routines that readers & writers can invoke — AcquireExclusive(), ReleaseExclusive(), AcquireShared(), & ReleaseShared(). These routines are each protected by a single mutex, & they collectively use 2 condition variables.
  Condition variable — wBusy & rBusy

- Only 1 condition variable could be used but the code would suffer from spurious wakeups in which writers can be awakened only to go back to sleep immediately.

THREAD SCHEDULING

- A kernel thread is a unit of scheduling w/in the kernel or OS.
  Threads can be mapped to kernel threads in 2 primary ways:

① We could map multiple threads to 1 kernel thread. Such threads can be created & destro-yed w/out OS intervention. Problem — if 1 thread blocks, the entire kernel thread blocks, & none of the other threads can execute. unbound threads

② Map each thread to its own kernel head. Here, the OS is aware of all threads, so when 1 thread blocks, another thread can be scheduled to execute in its place. They're called bound threads.

PRIORITY INVERSION

- Occurs when a low-priority thread holds a lock that a high-priority thread wishes to acquire.
- A more serious case occurs if a 3rd thread is created w/ a priority that is b/w the low- & high-priority threads. This medium-priority thread will preempt the low-priority thread. Thus, the high-priority thread must block as long as the medium-priority thread executes.

SOLUTIONS:

① Priority Ceiling - highest possible thread priority. Any thread that acquires a lock executes at this priority, which prevents some medium-level thread from preempting it.

② Priority Inheritance - when a thread acquires a mutex, it temporarily inherits the priority of the highest thread that is blocked waiting for that mutex.