Date: _____

Using History: LRU.

→ Replaces the least-recently-used page.

Access	Hit/Miss	Resulting.
0	Miss	0
1	Miss	0,1
2	Miss	0,1,2
0	Hit	1,2,0 → most ru
1	Hit	2,0,1 → most ru
3	Miss	0,1,3 → remove
0	Hit	1,3,0
3	Hit	1,0,3
1	Hit	0,3,1
2	Miss	3,1,2
1	Hit	3,2,1

→ ORDER SETS CHANGED

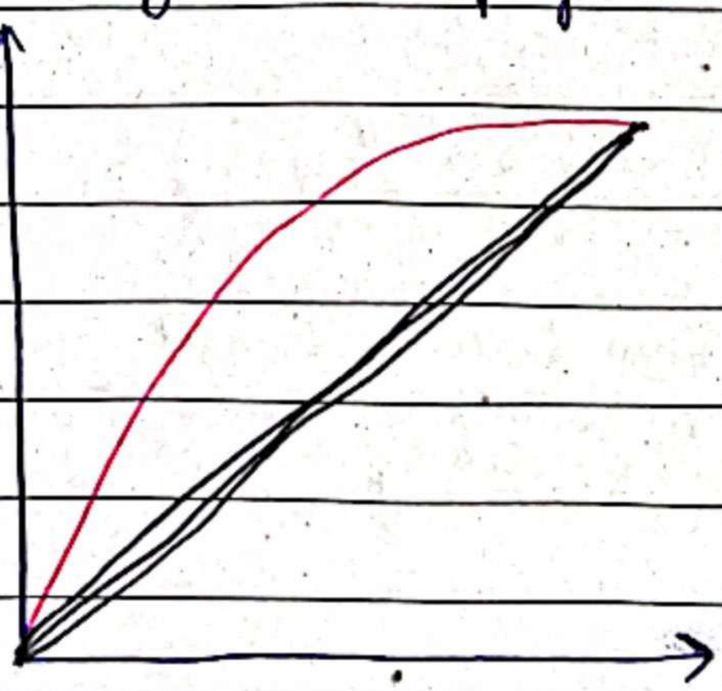
The Workload Example: The No-Locality Workload.

→ Each reference is to random page within the set of accessed pages.

→ workload accesses 100 unique pages over time.

→ choosing next page to refer to at random.

* optimal can predict future so better



— OPT

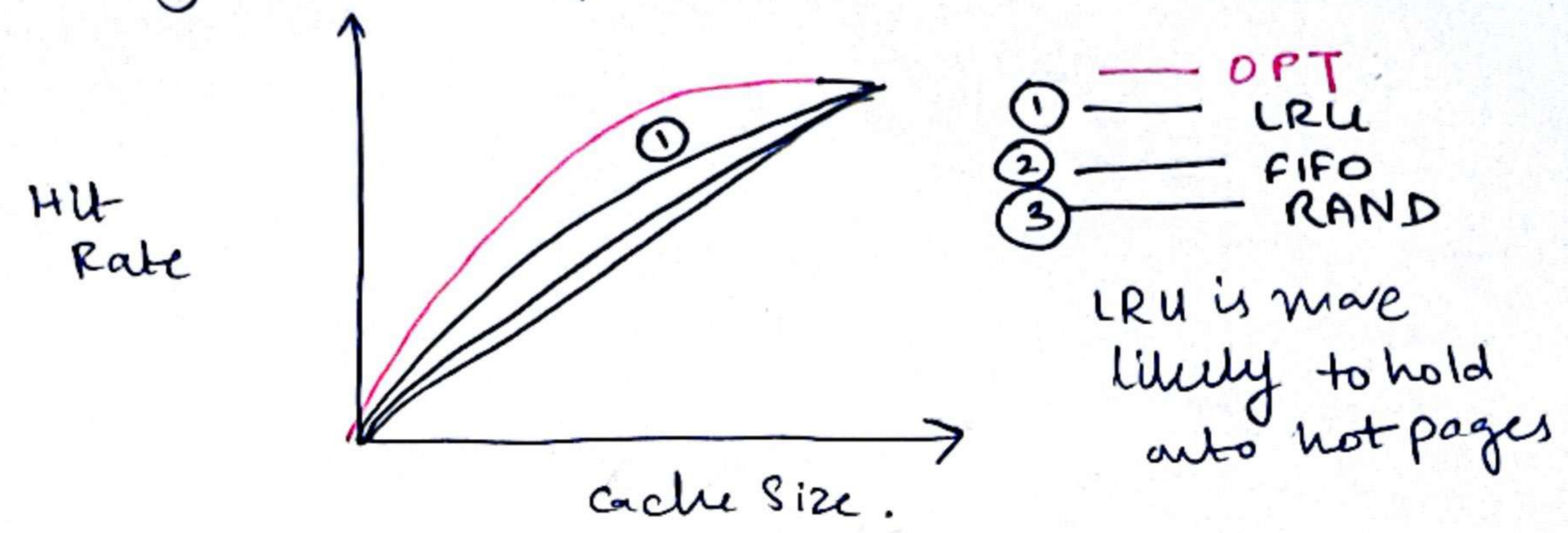
— LRU, FIFO RAND.

when the cache is large enough to fit the entire workload it doesn't matter which policy to use.

WORKLOAD EXAMPLE: 80-20 Workload.

→ Exhibits locality: 80% of reference are made to 20% of page

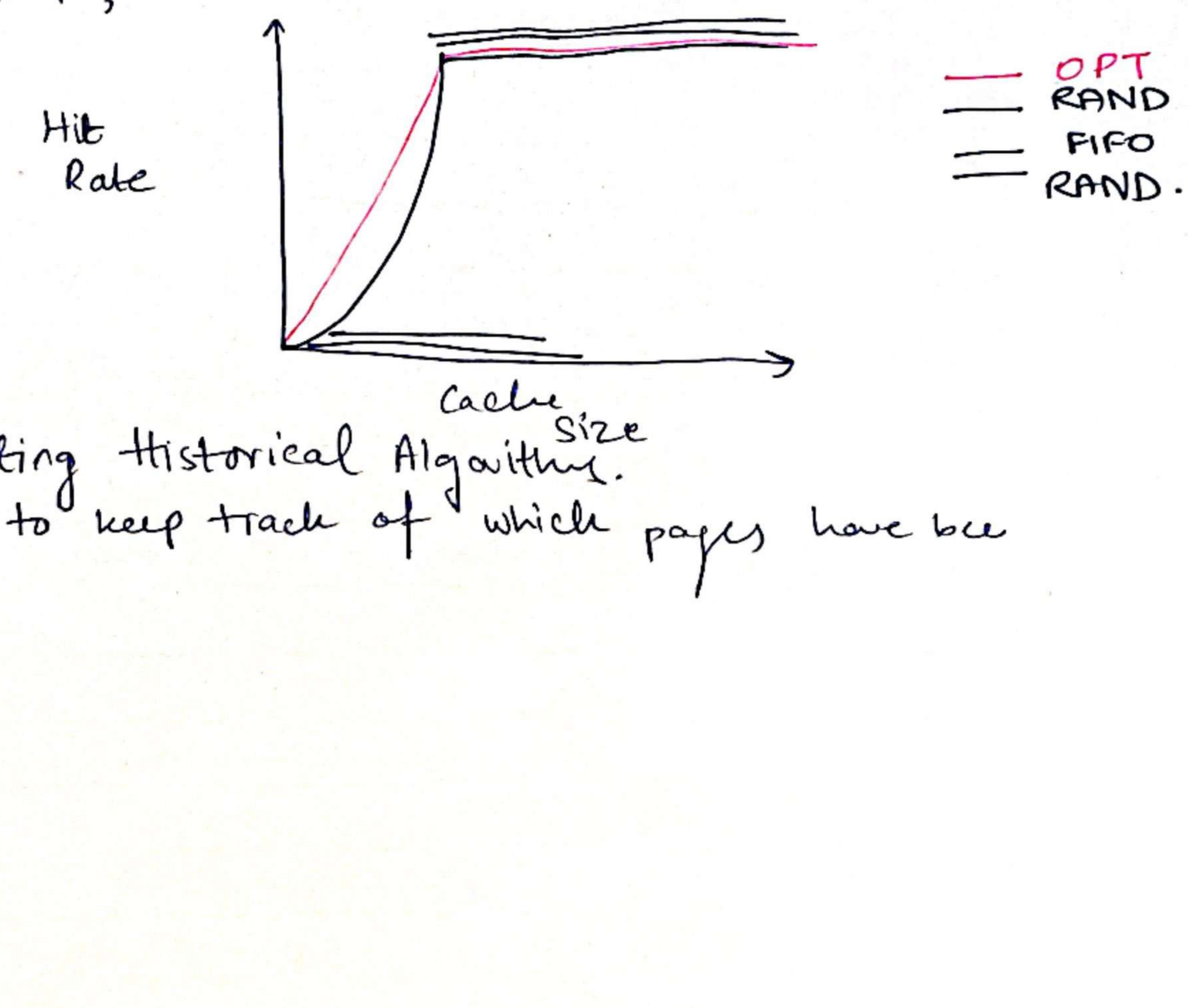
→ The remaing 20% of reference are made to remaing 80% of pages.



LOOPING SEQUENTAL

→ Refers to 50 pages in sequential

→ Starting at 0, then ..1, .. upto page 49 and then we loop, repeating those accesses for total of 10,000 accesses to 50 unique pages.



Implementing Historical Algorithms.

→ to keep track of which pages have bee

Date:_____

CHAPTER 26: Concurrency

→ The real world contains actors that execute independently of but communicate with each other.

→ In modelling the world, many parallel executions have to be composed and coordinated and that's where the study of concurrency comes in.

LEVELS OF CONCURRENCY.

① → hardware level.
  → processors/cores
  → computers.

② → Operating System level.
  → process
  → threads

③ → software level.
  → dist/server.

① Hardware.
  → flynn's taxonomy
    → SISD
    → SIMD
      → Vector, Array Processors, GPU.
    → MIMD.
      → Shared memory
      → Distributed Memory

Date:_____

(2) Operating System.

→ Single - User vs Multi - User

→ Single - tasking vs Multi-tasking.

→ Multi - tasking vs Multi - threading

→ Multi - threading vs Multi - Processing

→ API

→ fork () vs pthread_create ()

(3) Software / Algorithm.

→ Types of Applications.

→ Client / server.

→ Peer to Peer

→ Types of Algorithms.

→ Sequential.

→ Parallel

→ Distributed.

What is a thread?

→ Thread is an independent path of execution.

→ In a multi-threaded. program, each thread has it's own PC, register (context), stack, but it shares the process's address space.

Date:_____

THREADS VS PROCESSES.

→ Threads are like processes:

→ they can execute independently

→ each thread has seperate PC and set of registers (context) while executing.

→ if two threads T1 & T2 are running on a single processer then:

→ only one can run at any given time.

→ switching from one thread to other requires a context switch.

→ each thread has it's own stack.

→ threads share same address space.

→ context switch b/w threads results in switching of stacks but not of page tables.

→ A single process can have one or more threads.

→ Thread states are saved in Thread Control Blocks (TCBs) instead of PCBs.

why use threads?

→ Parallelism

→ better exploit multi-core CPUs.

→ Blocking I/O

→ when doing a blocking I/O operation, only one thread gets blocked.

→ overlap I/O with other activities within a process.

→ Threads used in Web servers, DBMS etc.

Date:

(1) less time to
- create a new thread than a process.
- terminate a thread than a process.
- switch b/w two threads within same processes.

(2) less memory requirement than multiprogramming.

(3) since threads within the same processes share memory & files they can communicate with each other without involving the kernel.

CPU Scheduling output
→ output is dependent on which thread is created. and which is scheduled first on CPU.

THREAD
→ a new abstraction for single running process
→ multi threaded program
→ multiple PCs
→ share same address space.

Context Switch
→ each thread has it's own PC and set of registers.
→ one or more TCB are needed to store state of each thread.
→ when switching from T1 to T2
→ register state of T1 is saved
→ register state of T2 is restored
→ the address space remains same.

Date:_____

Critical Section.

→ a piece of code that ~~shares~~ accesses a shared variable & must not be concurrently executed by more than one thread.

→ multiple threads can result in Race Condition

→ need to support atomicity for critical sections (mutual exclusion).

solution locks.

Locks.

→ Which of the following condition is necessary for deadlock to occur?

- ✓ mutual exclusion
- ✓ hold and wait
- ✓ no preemption

→ Prevent deadlock is to impose an ordering on acquisition of locks.

→ Deadlocks can occur in multithreaded environment even when a single process if two or more threads are waiting on each other to release locks.

Date:_____

CHAPTER 27: Interlude: Thread API

Thread Creation

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

→ thread: used to interact with this thread.

→ attr: used to specify any attributes this thread might have

→ stack size ...

→ start_routine: the function this thread start running in.

→ arg: the argument to be passed to function (start_routine).

→ void pointer allows us to pass in any type of argument.

Thread Completion:

```
int pthread_join (pthread_t thread, void ** value_ptr);
```

→ thread: specify which thread to wait for

→ value_ptr: a pointer to return value.

→ because pthread_join () routine changes the value, you need to pass in pointer to that value.

Date:_____

→ Stack is private to each thread.

→ Heap is shared among all threads.

Locks

→ Provide mutual exclusion to a critical section.

```
int pthread_mutex_lock(pthredd_mutex_t  *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

→ no other thread holds the lock → the thread will acquire the lock and enter the critical section.

→ if another thread hold the lock → the thread will not return from call until it has acquired the lock.

→ all locks must be propelly initialized.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

```
int pthread_mutex_trylock(Pthread_mutex_t *mutex)
int pthread_mutex_timelock(pthread_mutex_t *mutex,
                           struct timespec *abs_timeout);
```

→ trylock: return failure if lock is already held.

→ timelock: return after timeout.

Date: _____

CONDITION VARIABLES

→ Condition variables are useful when some kind of signalling must take place b/w threads

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

→ pthread_cond_wait
  → put the calling thread to sleep
  → wait for some other thread to signal it.

→ pthread_cond_signal
  → unlock at least one of the threads that are blocked on condition variable.

→ the wait call releases locks when putting said caller to sleep.
→ before returning after being woken the wait call re-acquire the lock.
  ① unlock mutex
  ② block calling thread until another thread signals condition variable.
  ③ relock the mutex (when thread is awoken).

→ the waiting thread re-checks the condition in a while-loop instead of if statement.
  → without rechecking, the waiting thread will continue thinking that the condition has changed even though it hasn't.

Date:_____

CHAPTER 28: Locks.

→ building a lock needs some help from the hardware & OS.

→ ensure that any critical section executes as if it were a single atomic instruction.

```
lock_t mutex;
...
lock (&mutex);
balance = balance + 1;
unlock (&mutex);
```

→ lock variable holds the state of the lock
  → available (or unlocked or free)
    → no threads holds the lock.
  → acquired (or locked or held).
    → exactly one thread holds the lock & presumably is in a critical section.

→ lock().
  → try to acquire the lock.
  → if no other threads holds the lock → the thread will acquire the lock.
  → enter critical section
    → this thread is said to be the owner of the lock.
  → other threads are prevented from entering entering critical section while first thread that holds it in there

*critical section is the owner of lock.

Date:_____

→ POSIX stands for Portable OS Interface.

→ The name that POSIX library uses for a lock.

→ Mutex - mutual exclusion. ie we make sure that only one thread can enter it's critical section at any given time.

→ if Thread 1 is in it's critical section and gets interrupted & Thread 2 is scheduled & tries to enter it's critical section, it should not be able to do so & shall block until Thread 1 has finished updating it's shared variables & it's out of it's critical section.

```
pthread_mutex_lock(&lock);
balance = balance + 1;
pthread_mutex_unlock(&lock);
```

→ we may be using different locks to protect different variables → increase concurrency.

APPROACH TO MAKE LOCK.

Controlling Interupts. 1st Solution to make lock.

① Disable Interupts for critical sections.

→ one of the earliest solutions used to provide mutual exclusion.

```
void lock() {
    DisableInterrupts();
}
```

```
void unlock() {
    EnableInterrupts();
}
```

when interupts are off
useful interupts can be
lost

Date:_____

→ Problem:

→ require too much trust in applications.

→ do not work on multiprocessors.

→ codes that mask/unmasks interupts be executed slowly by modern CPUs.

why hardware suppat needed?

(2nd Solution to make locks)
FLAGS.

```
void init (lock_t * mutex) {
    mutex → flag = 0;
}
void lock (lock_t * mutex) {
    while (mutex → flag == 1)
        ; // spin-wait (do nothing)
    mutex → flag = 1;
}
void unlock (lock_t * mutex) {
    mutex → flag = 0;
}
```

Problem (1)

no mutual exclusion.

Problem (2)

spin-waiting wastes time waiting for another thread.
(wastes CPU time)

→ So, we need an atomic instruction suppated by Hardware!

→ Atomic Exchange.

LOCK IMPLEMENTATION using HW. Date:_____

→ Hardware Support

① → Test & Set

② → Compare & Swap

③ → load linked & store conditional

④ → fetch & add.

Notice the & in each of them.

① TEST AND SET.

→ An instruction to support creation of simple locks.

```
int TestAndSet(int *ptr, int new){
    int old = *ptr;
    *ptr = new;
    return old;
}
```

ATOMIC INSTRUCTION

* cannot be interrupted

→ return (testing) old value pointed to by the ptr.

→ simultaneously updates (setting) said value to new.

→ This sequence of operations is performed atomically

Issues: A preemptive scheduler is required to work correctly on a single processor.

* this lock is called SPIN LOCK

* it resolves problem of mutual exclusion since no other thread can intervene

```
void lock(lock_t *lock){
while (testandset(flg, 1) == 1)
    ;// spin
}
```

Evaluating Spin locks.

→ Correctness: yes

→ The spin lock only allows single thread to enter critical section. * no mutual exclusion.

→ Fairness: No

→ Spin locks don't provide any fairness guarantees

→ A threads spining may spin forever.

```
int CompareAndSwap(int *ptr, int expected, int new
{   int actual = *ptr
    if (actual = expected)
        *ptr = new
    return actual
};
```

Date:

→ Performance.

→ in single CPU performance overheads can be quite painful

→ if number of threads roughly equals number of CPU, spin locks work reasonably well.

② Compare And Swap.

→ Test whether the value at address (ptr) is equal to expected

→ If so update the memory location pointed by ptr with new value

→ in either case, return actual value at that memory

```
void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag, 0, 1) == 1)
        ; // spin.
}
```

Load & linked & Store Conditional.

→ You look at a chair LL and see if it's empty

→ You decide to sit on it (SC) but only if nobody sat down while you were walking toward it. If someone did, you go back & check again

→ This mechanism avoids problems like race conditions where multiple threads try to sit on chair at same time.

(3) Date:_____

- How LL/SC work:

① Load Linked (LL).

→ It reads value from memory & remembers that memory address. marks linked.

→ Think of it like saying 'I'm watching this spot in memory for any changes.

② Store Conditional (SC).

→ It tries to write (store) a new value to same memory but only if no one else changed it since LL.

→ If someone else modified the memory, the SC fail -If nothing change it succeeds.

→ Avoid race conditions.

④ Fetch and Add.

Atomically increment a value while returning old value at a particular address.

```
int FetchAddress (int * ptr) {
    int old = * ptr
    * ptr = old + 1
    return old.
}
```

* assign unique ticket number to each thread.
* fist come, first serve
→ if thread gets ticket 0 will be served fist

Date:_____

Ticket lock.

→ can be built with fetch and add.

ensure progress for all threads → fairness

So much spinning

→ hardware-based spin locks are simple and they work.

→ in some cases, these solutions can be quite efficient.

→ any time a thread gets caught spinning, it wastes an entire time slice doing nothing.

How to avoid spinning?

We'll need OS support.

A simple approach: Just Yield.

→ when you are going to spin, give up the CPU to another Thread.

yield(): → OS system calls moves caller from running to ready state.

→ the cost of context switch can be substantial & starvation problem still exists.

* locks inside the OS will always use spinlocks.
  → OS cannot yield/sleep.
  → For user we ~~cannot~~ can use yield/sleep.

Date:_____

```
void lock () {
    while (TestAndSet (&flag, 1) == 1)
        yield (); // give up CPU
}
```

→ Sleep instead of spining, if lock is held.
→ Park() and unpark() support provided by Solaris sleep.
  → guard is a spin lock around flag and wait queues
  → ~~can~~ spin is done but only for the time we access flag & queue.
  → flag is not set to 0 on unpark().

Park() put a calling thread to sleep || unpark(threadID) wake particular thread

Futexes.

→ Linux provides futex
  → fast user space mutex
→ futex_wait (address, expected)
  → put calling thread to sleep
  → if value at address is not equal to expected, the call returns immediately.
→ futex_wake
  → wakes thread waiting in queue.

Mutex lets a thread acquire lock once
Recursive allows a thread to acquire lock again and again
→ number of time locked = Number of time unlocked.

TWO-PHASE LOCKS.

(1) first phase
→ the lock spins for a while, hoping that it can acquire lock
→ if lock not acquired during first spin, a second phase is entered

Phase (2)
→ The caller is put to sleep.
→ The caller is only woken up when lock becomes free later.

→ Coarse Grained vs Fine Grained locking, One Big Lock vs seperate locks
→ Fine grained allows more parallelism.
→ Multiple fine grained locks may be harder.

→ Deadlock
→ thread waits | indefinitely | one each other to release resources in loop.
→ lock in different orders then deadlock.
→

CHAPTER 29: LOCK BASED CONCURRENT DS.

→ Adding locks to data structure to make it usable by threads makes the structure thread safe.

How locks are added determine both correctness & performance of the data structure?

→ Many data structures exist, simple is the concurrent counter.

```
int get(counter_t *c) {
        pthread_mutex_lock(&c->lock)
        int rc = c->value;
        pthread_mutex_unlock(&c->lock)
        return rc;
}
```

→ each thread updates single shared counter
→ performance of concurrent counter scales poorly
→ locks add additional overhead which reduces performance.

Solution:

→ use approximate counters which update counter value after a certain threshold S so the lock is accessed S times.

APPROXIMATE COUNTING.

Date:_____

→ The sloppy counter works by representing
(1) a single ~~too~~ logical counter via numerous local physical counter or per CPU core.
(2) a single global counter
(3) there are locks are for local counter & global counter.

→ When thread running on a core wishes to increment counter
→ it increments local counter
→ each CPU has it's own local counter
(1) threads across CPUs can update local counter
(2) Thus counter updates are scalable.

→ local values are periodically transferred to global counter
→ acquire global lock
→ increment it by local counter's value
→ local counter is then set to zero

(after local count exceeds threshold then transfer to global)

* an sticky noted.

Date:_____

How often the local-to-global transfer occurs is determined by a threshold S (sloppiness).

→ The smaller S (Performance Poor, Global Count ACCURATE)
→ The more the counter behaves like non scalable counter.

→ The bigger S. (Performance Excellent, Global count Lags)
→ the more scalable the counter
→ the further off the global value might be from actual count.

LINKED LIST CONCURRENT

```
typedef struct _list_t {
        node_t *head;
        pthread_mutex_t lock
} list_t;
```

→ Insert Function, Malloc call could be moved out of lock to avoid branching.

```
void List_Insert(list_t *L, int key) {
//synchronisation not needed
node_t *new=malloc(sizeof(node_t));
if (new ==NULL){
        perror("malloc")
        return
}
new->key = key;
```

Date:_____

```
// just lock critical section.
pthread_mutex_lock (&L->lock);
new -> next = L -> head
L -> head = new
pthread_mutex_unlock (&L->lock).
}
```

Scaling linked list.

→ add a lock per node of list instead of having single lock.

→ when traversing the list

① grab lock of next node

② and then release current lock

PROBLEM:

→ high overhead because we need to access each node's lock when doing traversal.

CONCURRENT QUEUES.

→ There are Two locks

① head of queue lock.

② are for tail

→ to provide concurrency of enqueue and dequeue operations.

* locks in dequeue start to end

* enqueue locks where critical section

Date:_____

CONCURRENT HASH TABLE

→ built using concurrent list

→ uses lock per hash bucket each of which is represented by list.

CHAPTER 30 : CONDITION VARIABLES. Date:_____

→ There are many cases where a thread wishes to check whether a condition is true before continuing it's execution.

for eg.

→ A parent thread might wish to check whether a child tread has completed.

→ this is often called a join().

Q) Difference b/w parent-child process & parent-child thread.

→ Parent-Child Process

→ 2 seperate virtual memory

→ 2 seperate PID.

→ Though join parent process waits.

→ when child process is created the PC is also at 0.

→ Parent-Child Thread.

→ Only a pseudo name. use informally. Here they are peers because they're sharing the same VM.

→ They have their own stack but shared heap & code

→ only difference is are made the another.

Date:_____

volatile int done = 0;

→ volatile key word ensures that main thread always reads the actual value of done from memory

→ synchronisation b/w main thread and child thread.

```
void * child (void *arg) {
    printf("child \n");
    return NULL;          → done=1
}
int main (int argc, char * argv[]){
    printf ("parent: begin \n") ;
    pthread_t c;
    pthread_create (&c, NULL, child, NULL);
    while (done ==0)
        ; // spin.
    printf ("parent: end \n")
    return 0;
}
```

→ highly inefficient as parent spins and wastes CPU time → put parent to sleep until condition we are waiting becomes true.

HOW TO WAIT FOR CONDITION?

→ waiting on a condition

wait() — put a thread in queue of waiting threads.

→ signalling on the condition

signal() — same other thread when it changes said state, can wake one of those waiting threads & allow them to continue.

Date:_____

→ declaring a condition variable

pthread_cond_t c;

wait()

puts all threads on sleep

signal()

is executed when a thread has changed something.

pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m)

pthread_cond_signal(pthread_cond_t *c).

① unlock the mutex

② block the calling thread until another thread signals condition variable || sleep.

③ when thread wakes up it must re-acquire lock.

→ prevents race condition.

```
int done = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALISER.
pthread_cond_t c = PTHREAD_COND_INITIALISER.
void thr_exit() {
        pthread_mutex_lock(&m); (9)
        done = 1; (10)
        pthread_mutex_unlock(&m);
}
```

(11) → pthread_cond_signal(&c);

Date:_____

```
void *child (void *args){
    printf("child\n");   (7)
    thr_exit();          (8)
    return NULL;
}
```

```
void thr_join (){
    pthread_mutex_lock(&m);          (3)
    while (done == 0)                (4)
        pthread_cond_wait (&c, &m)   (5)
    pthread_mutex_unlock (&m)        (6)
}

int main (int argc, char *argv[]){
    printf("parent: begin\n");
    pthread_t p;
    pthread_create(&p, NULL, child, NULL)   (1)
    thr_join();                             (2)
    printf("parent: end\n");
    return 0;
}
```

→ Parent

(1) → create child thread & continues running itself

(2) → calls into thr_join() to wait for child thread to complete.

(3) → acquire lock

(4) → check if child is done.

(5) → put itself to sleep by calling wait()

(6) → release lock.

Date:_____

→ Child

(7) → prints message "child"

(8) → call thr_exit to WAKE parent thread

(9) → Grab lock

(10) → set variable done

(11) → signal parent to wake up.

```
void thr_exit(){
    pthread_mutex_lock(&m)
    pthread_cond_signal(&c)
    pthread_mutex_unlock(&m).
}
void thr_join(){
    pthread_mutex_lock(&m)
    pthread_cond_wait(&c, &m)
    pthread_mutex_unlock(&m).
}
```

→ Child runs immediately

→ child will signal, but there is no thread asleep on condition.

→ when parent runs it will call wait & be stuck.

→ no thread will ever wake it.

→ the main thread will wait indefinitely causing a deadlock

Date:_____

```
void thr_exit(){
        done=1
        pthread_cond_signal(&c).
}
void thr_join(){
    if (done==0)
        pthread_cond_wait(&c)
```

3

→ issue Race conditions because no locks used.

→ overwriting of variable done, parent was trying to go to sleep, but just before it calls wait() to go to sleep it gets interrupted by another child.

PRODUCER/CONSUMER (Bound Buffer Problem).

→ Producer

→ produce data items

→ wish to place data item in a buffer.

→ Consumer

→ Grab data items out of buffer consume them in same way

e.g (1) ~~HTTP~~ puts A producer puts HTTPS requests in work queue. Consumer threads take requests out of this queue & process them.

Date: _____

(2) When you pipe the output of one program into another.

`grep foo file.txt | wc -l`

(displays lines in file.txt with a string foo) ← grep foo file.txt = producer; wc -l = consumer. (displays # of lines)

→ runs two process concurrently

→ b/w them is an in-kernel BOUNDED BUFFER. is shared resource.

VERSION (1)

→ so synchronised access is required.

```
int buffer;
int count = 0;
void put(int value){
    assert (count == 0)
    count = 1
    buffer = value.
}
void int get () {
    assert (count == 1);
    count = 0;
    return buffer
}.
```

only put data in buffer when count = 0. // buffer empty

only get data from the buffer when count = 1 // buffer full

VERSION (1)

```
void buf* producer (void *args){
    int i
    int loops = (int) arg;
    for (i=0; i < loops; i++) {
        put(i)
    }
}
```

→ lock
→ condition to wait

→ signal
→ unlock

Date: 8

```
void *consumer(void *arg){
    int i
    while(1){
        int tmp = get()
        printf("%d\n", temp).
    }
}
```

SINGLE PRODUCER CONSUMER.

→ producer puts an integers into shared buffer loops number of times.

→ consumer gets data out of that buffer.

What could go wrong here...

→ Race Condition

Solution.

① Condition variable associated with lock mutex leading to synchronisation.

→ works well for single producer and single consumer.

If we have more than one of producer & consumer?

Refer to Table.

Date:_____

→ The problem arises for a simple reason
→ after producer woke T_{c1}, but before T_{c1} ever ran the state of bounded buffer changed by T_{c2}
→ There is no guarantee that when the woken thread runs, the state will still be as desired → Mesa semantics.
→ Virtually every system ever built employs Mesa semantics.

→ Hoare semantics: provides a stronger guarantee that the woken thread will run immediately upon being woken.

The above code was Mesa semantics, because it doesn't have control over ~~which consumer~~ signaled thread which is in ready state.

MESA Vs HOARE.

→ Real world Practically: In Mesa semantics, when a thread is signalled, it only gets notification to check condition again rather than being guaranteed that condition is immediately true. (as in Hoare semantics)

→ Context Switching Overhead: Signalling thread finishes it's critical section reducing context switches (Mesa).
Immediate context switch to waiting thread (Hoare).

Date:_____

→ Implementation : (Mesa) Easier to implement with existing scheduling mechanisms.
(Hoare) Requires tightly coupled guarantees complicating scheduler design.

→ Fairness & Scheduling:
(Mesa). Scheduler decides when to wake up the waiting thread, ensuring fairness
(Hoare). Signaled thread gets immediate priority potentially starving other threads

→ Suitability for Share Resource.
(Mesa). Threads reevaluate conditions, robust rapidly changing states
(Hoare). Signaled thread assume condition is true.

* A simple rule to remember with condition variable is always use while loop.
→ however you also require another condition variable because using one variable the consumer (1) is waking up consumer (2)
→ use 2 CV empty & fill.