1) V

```
int main () {
    pid_t pids [NO_OF_CHILDREN];
    for (i=0;  i< NO_OF_CHILDREN;  i++) {
        pids [i] = fork ();
        IF (pids [i]) < 0  → failed
        IF (pids [i]) == 0  → child
            i = 0  ⇒  cpu_task (7)
            i = 1  ⇒  mem_task (7)
            i = 2  ⇒  cpu_mem (7)
        ELSE  → parent
        (printf ("Proc with pid %d executing", getpid ()));
    }
    return 0;
}
```

```
void cpu_task (int n) {
    int result = 1;
    for (i from 1 to n)
        result *= i
    printf ("Factorial");
}
```

```
void mem_task (int n) {
    int *a = malloc (n * int_size);
    for (i from 0 to n)
        a[i] = i
    printf ("At %p", a);
    free (a);
}
```

2> Multi thread I/O

```
int main () {
    pthread_t p, q;
    char *r_filename = "input.txt";
    char *w_filename = "output.txt";
    pthread_create (&p, NULL, readFile, (void*) r_filename);
    pthread_create (&q, NULL, writeFile, (void*) w_filename);
    pthread_join (p, NULL);
    pthread_join (q, NULL);
    return 0;
}
```

```
void *readfile (void *arg) {
    char *filename = (char*) arg;
    FILE *file = fopen (filename, "r");
    char buffer[100];

    fgets (buffer, sot, file);
    printf ("Read: %s", buffer);

    flclose (file);
    pthread_exit (NULL);
```

```
void *writefile (void *arg) {
    char *filename = (char*) arg;
    FILE *file = fopen (filename, "a");
    char buffer[100];
    printf ("Enter TEX");
    fgets (buffer, sot, stdin);
    fprintf (file, "%s", buffer);

    fclose (file);
    pthread_exit (NULL);
```

__/__/____

UNIT 3

LOCKS

* The Basic Idea

```
lock_t mutex;
lock (&mutex);
balance += 1; // critical section
unlock (&mutex);
```

* lock is just a variable
* It holds the state of lock
  0 → unlocked / available
  1 → locked / acquired

* The basic idea behind is that only one thread should hold the lock and enter critical section

* lock() → tries to acquire the lock
  unlock() → releases the held lock

* Since threads are scheduled by OS, locks provide some amount of control to programmers

__/__/____

* Pthread Locks

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

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

* POSIX library lock

* Variable is passed
  Allows for different locks

* Fine-grained approach (↑ concurrency)

```
pHread_mutex_t  balance lock;
pthread_mutex_t  array lock;
```

* Building a Lock

* Need help from - hardware
  - OS

* Over years, different hardware primitives has been added to instruction sets of various computer architecture

* Evaluating Locks

i) Mutual Exclusion
- Whether lock does its basic task
- Does it prevent multiple threads to enter CS

ii) Fairness
- Does each thread contending to obtain lock has fair shot? Does any thread starve

iii) Performance
- What is the time overhead?
  - single thread
  - multi threads single CPU
  - multi threads multi CPU

__/__/__

* Controlling Interrupts

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

* For single processor system
* Disabling interrupts before entering cs
* So ~~not~~ no other threads will interfere

* - privileged operation
  (requires to trust the application)
  - trust can be abused
  - not for multiprocessor system
  - crucial interrupts can get lost

* First attempt : Simple flag

lock() : obtains the flag
unlock() : releases the flag
spin-wait : if flag is obtained

- Correctness : when unlock() is called, 2 threads can get flag
- Performance : spinning

[Test if flag == 0, set flag = 1]

__/__/____

* **Building Working Spin Lock (using Test-And-Set)**

- Hardware support for mutual exclusion:
  test-and-set instruction (xchg on x86)
  (ldstub on SPARC)

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

```
void lock (lock_t lock) {
    while (TestAndSet (&lock->flag, 1) == 1)
        ;
}
```

* test (of old lock value) } single
set (of new value) } atomic operation

* ensures mutual exclusion

* requires preemptive scheduler (sleep)

* Evaluating Spin Locks

i) Correctness : ✓ Provides mutual exclusion

ii) Fairness : X Can lead to starvation

iii) Performance : ✓ in multi processor
X in single processor

* Compare And Swap

```
int cAs (int *ptr, int exp, int new) {
    int actual = *ptr;
    if (actual == exp)
        *ptr = new;
    return actual;
}
```

- Another hardware primitive
- In lock() routine
  cAs (& lock → flag, 0, 1)

__/__/__

* Load Linked and Store Conditional

__/__/____

* Fetch and Add

```
int FetchAndAdd (int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

- It has struct lock_t { int turn;
  int ticket;
  }
- Thread fetches and adds to ticket
- That ticket value is its turn
- When unlocking, increment turn
- So thread whose turn it is, is executed

* Using Queues

```
struct lock_t { int flag;
                int guard;
                queue_t q;
              }

void lock (lock_t *m) {
    while (TaS (m->guard, 1) == 1)
    ;
    if (m -> flag == 0) {
        m -> flag = 1;
        m -> guard = 0;
    } else {
        queue_add (m->q, gettid());
        m -> guard = 0;
        park (m);
    }
}

void unlock (lock_t *m) {
    while (TaS (m->guard, 1) == 1) { ; }
    if (queue_empty ()) { m -> flag = 0 }
    else { unpark (queue_front (m->q)) }
    m -> guard = 0;
}
```

__/__/____

* Different OS, Different Support

```
void lock (int *mutex) {
  if (atomic_bit_test_set (31, mutex) == 0)
     return;
  atomic_increment (mutex);
  while (1) {
     if (a_b_t_s (31, mutex) == 0) {
        atomic_decrement (mutex);
        return;
     }
     futex_wait (mutex, 1);
}

void unlock (int *mutex) {
     if (atomic_add_zero (mutex, 0x8000)
        return;
     futex_wake (mutex);
}
```

futex_wait (address, expected) } Two
futex_wake (address) } Phase Lock