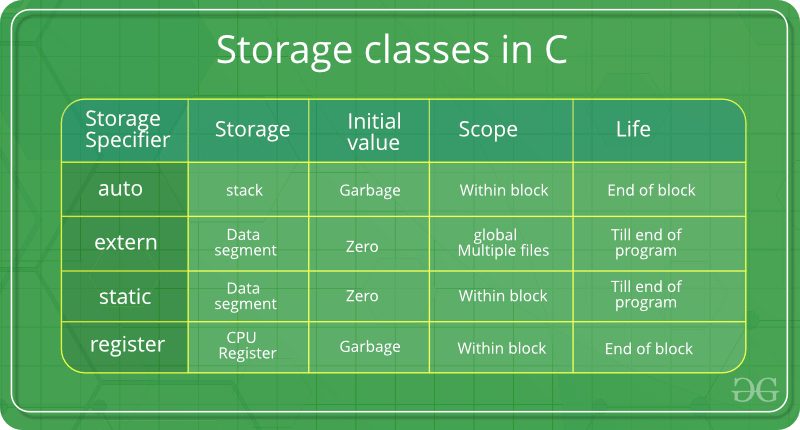
# Topics

## Storage Classes



## Constant qualifier

1. **Pointer to variable**

int \*ptr;

We can change the value of ptr and we can also change the value of object ptr pointing to.

1. **Pointer to constant**

const int \*ptr;

int const \*ptr;

We can change the pointer to point to any other integer variable, but cannot change the value of the object (entity) pointed using pointer ptr. The pointer is stored in the read-write area (stack in the present case).

1. **Constant pointer to variable**

int \*const ptr;

Above declaration is a constant pointer to an integer variable, means we can change the value of object pointed by pointer, but cannot change the pointer to point another variable.

1. **constant pointer to constant**

const int \*const ptr;

Above declaration is a constant pointer to a constant variable which means we cannot change value pointed by the pointer as well as we cannot point the pointer to other variables. Let us see with an example.

## System Calls

* It the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on.
* A system call is a way for programs to **interact with the operating system**.
* A computer program makes a system call when it makes a request to the operating system’s kernel.
* System call **provides** the services of the operating system to the user programs via Application Program Interface(API). It provides an interface between a process and operating system to allow user-level processes to request services of the operating system.
* System calls are the only entry points into the kernel system. All programs needing resources must use system calls.

|  |  |  |
| --- | --- | --- |
|  | **WINDOWS** | **UNIX** |
| Process Control | CreateProcess() ExitProcess() WaitForSingleObject() | fork() exit() wait() |
| File Manipulation | CreateFile() ReadFile() WriteFile() CloseHandle() | open() read() write() close() |
| Device Manipulation | SetConsoleMode() ReadConsole() WriteConsole() | ioctl() read() write() |
| Information Maintenance | GetCurrentProcessID() SetTimer() Sleep() | getpid() alarm() sleep() |
| Communication | CreatePipe() CreateFileMapping() MapViewOfFile() | pipe() shmget() mmap() |
| Protection | SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup() | chmod() umask() chown() |

## Structure padding

* In order to align the data in memory,  one or more empty bytes (addresses) are inserted (or left empty) between memory addresses which are allocated for other structure members while memory allocation. This concept is called structure padding.
* Architecture of a computer processor is such a way that it can read 1 word (4 byte in 32 bit processor) from memory at a time.
* To make use of this advantage of processor, data are always aligned as 4 bytes package which leads to insert empty addresses between other member’s address.
* Because of this structure padding concept in C, size of the structure is always not same as what we think.

       For example, please consider below structure that has 5 members.

struct student

{

       int id1;

       int id2;

       char a;

       char b;

       float percentage;

};..

* As per C concepts, int and float datatypes occupy 4 bytes each and char datatype occupies 1 byte for 32 bit processor. So, only 14 bytes (4+4+1+1+4) should be allocated for above structure.
* But, this is wrong.  Do you know why?
* Architecture of a computer processor is such a way that it can read 1 word from memory at a time.
* 1 word is equal to 4 bytes for 32 bit processor and 8 bytes for 64 bit processor. So, 32 bit processor always reads 4 bytes at a time and 64 bit processor always reads 8 bytes at a time.
* This concept is very useful to increase the processor speed.
* To make use of this advantage, memory is arranged as a group of 4 bytes in 32 bit processor and 8 bytes in 64 bit processor.
* Below C program is compiled and executed in 32 bit compiler. Please check memory allocated for structure1 and structure2 in below program.

## way to search in sorted link list in better then O(n) time

SkipList 🡺 analogy of fast stations and slow stations

# Coding

## Max Sum Subarray

int kadane(vector<int> &A){

int n=A.size();

if(n<1) return 0;

int msum=A[0], csum=A[0];

for(int i=0; i<n; i++){

if(csum+A[i]<A[i])

csum=A[i];

else csum+=A[i];

msum=max(msum, csum);

}

return msum;

}

## Implement strtok

#include <stdio.h>

#include <string.h>

#include <malloc.h>

char\* my\_strtok(char\* s, char\* delm)

{

static int currIndex = 0;

if(!s || !delm || s[currIndex] == '\0')

return NULL;

char \*W = (char \*)malloc(sizeof(char)\*100);

int i = currIndex, k = 0, j = 0;

while (s[i] != '\0'){

j = 0;

while (delm[j] != '\0'){

if (s[i] != delm[j])

W[k] = s[i];

else goto It;

j++;

}

i++;

k++;

}

It:

W[i] = 0;

currIndex = i+1;

//Iterator = ++ptr;

return W;

}

int main(void)

{

char s[100] = "my name is khan";

char delm[] = " ";

//char newstr[100];

char \*str = my\_strtok(s, delm);

while(str){

printf("%s", str);

free(str);

str = my\_strtok(s, delm);

}

return 0;

}

## Binary Search

int binarySearch(int arr[], int l, int r, int x)

{

    if (r >= l) {

        int mid = l + (r - l) / 2;

        // If the element is present at the middle

        // itself

        if (arr[mid] == x)

            return mid;

        // If element is smaller than mid, then

        // it can only be present in left subarray

        if (arr[mid] > x)

            return binarySearch(arr, l, mid - 1, x);

        // Else the element can only be present

        // in right subarray

        return binarySearch(arr, mid + 1, r, x);

    }

    // We reach here when element is not

    // present in array

    return -1;

}

OR

int binarySearch(int arr[], int l, int r, int x)

{

    while (l <= r) {

        int m = l + (r - l) / 2;

        // Check if x is present at mid

        if (arr[m] == x)

            return m;

        // If x greater, ignore left half

        if (arr[m] < x)

            l = m + 1;

        // If x is smaller, ignore right half

        else

            r = m - 1;

    }

    // if we reach here, then element was

    // not present

    return -1;

}

## Create BST from sorted array

TreeNode\* createBST(const vector<int> &A, int l, int r){

TreeNode \*root=NULL;

if(l<=r){

int m=ceil(double(l+r)/2);

root=new TreeNode(A[m]);

root->left=createBST(A,l,m-1);

root->right=createBST(A,m+1,r);

}

return root;

}

## Palindrome

bool isPalindrome(string s){

int l=0, r=s.size()-1;

while(l<r){

if(s[l++]!=s[r--])

return false;

}

return true;

}

## Check bit of number is on or off

1) Let us take number 'NUM' and we want to check whether it's 0th bit is ON or OFF

bit = 2 ^ 0 (0th bit)

if NUM & bit == 1 means 0th bit is ON else 0th bit is OFF

2) Similarly if we want to check whether 5th bit is ON or OFF

bit = 2 ^ 5 (5th bit)

if NUM & bit == 1 means its 5th bit is ON else 5th bit is OFF.

## Integer to Binary

Recursive function:

void bin(unsigned n)

{

    /\* step 1 \*/

    if (n > 1)

        bin(n/2);

    /\* step 2 \*/

    cout << n % 2;

}

## Next Greater Element

vector<int> NGE(vector<int> &A){

int n=A.size();

vector<int> res;

if(n<1) {

res.push\_back(-1);

return res;

}

stack<int> st;

st.push(A[0]);

for(int i=1; i<n; i++){

while(!st.empty() && st.top()<A[i]){

res.push\_back(A[i]);

st.pop();

}

st.push(A[i]);

}

res.push\_back(-1);

return res;

}

## Delete node in BST

// C program to demonstrate delete operation in binary search tree

#include<stdio.h>

#include<stdlib.h>

struct node

{

    int key;

    struct node \*left, \*right;

};

// A utility function to create a new BST node

struct node \*newNode(int item)

{

    struct node \*temp =  (struct node \*)malloc(sizeof(struct node));

    temp->key = item;

    temp->left = temp->right = NULL;

    return temp;

}

// A utility function to do inorder traversal of BST

void inorder(struct node \*root)

{

    if (root != NULL)

    {

        inorder(root->left);

        printf("%d ", root->key);

        inorder(root->right);

    }

}

/\* A utility function to insert a new node with given key in BST \*/

struct node\* insert(struct node\* node, int key)

{

    /\* If the tree is empty, return a new node \*/

    if (node == NULL) return newNode(key);

    /\* Otherwise, recur down the tree \*/

    if (key < node->key)

        node->left  = insert(node->left, key);

    else

        node->right = insert(node->right, key);

    /\* return the (unchanged) node pointer \*/

    return node;

}

/\* Given a non-empty binary search tree, return the node with minimum

   key value found in that tree. Note that the entire tree does not

   need to be searched. \*/

struct node \* minValueNode(struct node\* node)

{

    struct node\* current = node;

    /\* loop down to find the leftmost leaf \*/

    while (current && current->left != NULL)

        current = current->left;

    return current;

}

/\* Given a binary search tree and a key, this function deletes the key

   and returns the new root \*/

struct node\* deleteNode(struct node\* root, int key)

{

    // base case

    if (root == NULL) return root;

    // If the key to be deleted is smaller than the root's key,

    // then it lies in left subtree

    if (key < root->key)

        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the root's key,

    // then it lies in right subtree

    else if (key > root->key)

        root->right = deleteNode(root->right, key);

    // if key is same as root's key, then This is the node

    // to be deleted

    else

    {

        // node with only one child or no child

        if (root->left == NULL)

        {

            struct node \*temp = root->right;

            free(root);

            return temp;

        }

        else if (root->right == NULL)

        {

            struct node \*temp = root->left;

            free(root);

            return temp;

        }

        // node with two children: Get the inorder successor (smallest

        // in the right subtree)

        struct node\* temp = minValueNode(root->right);

        // Copy the inorder successor's content to this node

        root->key = temp->key;

        // Delete the inorder successor

        root->right = deleteNode(root->right, temp->key);

    }

    return root;

}

## File reading

// reading a text file

#include <iostream>

#include <fstream>

#include <string>

using namespace std;

int main () {

string line;

ifstream myfile ("example.txt");

if (myfile.is\_open())

{

while (! myfile.eof() )

{

getline (myfile,line);

if (line = something)

{

// read next line and print it... but how?

}

myfile.close();

}

else cout << "Unable to open file";

return 0;

## Sieve of Eratosthenes

Given a number n, print all primes smaller than or equal to n.

void SieveOfEratosthenes(int n)

{

    // Create a boolean array "prime[0..n]" and initialize

    // all entries it as true. A value in prime[i] will

    // finally be false if i is Not a prime, else true.

    bool prime[n+1];

    memset(prime, true, sizeof(prime));

    for (int p=2; p\*p<=n; p++)

    {

        // If prime[p] is not changed, then it is a prime

        if (prime[p] == true)

        {

            // Update all multiples of p greater than or

            // equal to the square of it

            // numbers which are multiple of p and are

            // less than p^2 are already been marked.

            for (int i=p\*p; i<=n; i += p)

                prime[i] = false;

        }

    }

    // Print all prime numbers

    for (int p=2; p<=n; p++)

       if (prime[p])

          cout << p << " ";

}

## Print all prime factors of a given number

Following are the steps to find all prime factors.  
**1)** While n is divisible by 2, print 2 and divide n by 2.  
**2)** After step 1, n must be odd. Now start a loop from i = 3 to square root of n. While i divides n, print i and divide n by i. After i fails to divide n, increment i by 2 and continue.  
**3)** If n is a prime number and is greater than 2, then n will not become 1 by above two steps. So print n if it is greater than 2.

// C++ program to print all prime factors

#include <bits/stdc++.h>

using namespace std;

// A function to print all prime

// factors of a given number n

void primeFactors(int n)

{

    // Print the number of 2s that divide n

    while (n % 2 == 0)

    {

        cout << 2 << " ";

        n = n/2;

    }

    // n must be odd at this point. So we can skip

    // one element (Note i = i +2)

    for (int i = 3; i <= sqrt(n); i = i + 2)

    {

        // While i divides n, print i and divide n

        while (n % i == 0)

        {

            cout << i << " ";

            n = n/i;

        }

    }

    // This condition is to handle the case when n

    // is a prime number greater than 2

    if (n > 2)

        cout << n << " ";

}

/\* Driver code \*/

int main()

{

    int n = 315;

    primeFactors(n);

    return 0;

}

## Mergesort using multithreading

|  |  |  |
| --- | --- | --- |
| // CPP Program to implement merge sort using  // multi-threading  #include <iostream>  #include <pthread.h>  #include <time.h>    // number of elements in array  #define MAX 20    // number of threads  #define THREAD\_MAX 4    using namespace std;    // array of size MAX  int a[MAX];  int part = 0;    // merge function for merging two parts  void merge(int low, int mid, int high)  {      int\* left = new int[mid - low + 1];      int\* right = new int[high - mid];        // n1 is size of left part and n2 is size      // of right part      int n1 = mid - low + 1, n2 = high - mid, i, j;        // storing values in left part      for (i = 0; i < n1; i++)          left[i] = a[i + low];        // storing values in right part      for (i = 0; i < n2; i++)          right[i] = a[i + mid + 1];        int k = low;      i = j = 0;        // merge left and right in ascending order      while (i < n1 && j < n2) {          if (left[i] <= right[j])              a[k++] = left[i++];          else              a[k++] = right[j++];      }        // insert remaining values from left      while (i < n1) {          a[k++] = left[i++];      }        // insert remaining values from right      while (j < n2) {          a[k++] = right[j++];      }  }    // merge sort function  void merge\_sort(int low, int high)  {      // calculating mid point of array      int mid = low + (high - low) / 2;      if (low < high) {            // calling first half          merge\_sort(low, mid);            // calling second half          merge\_sort(mid + 1, high);            // merging the two halves          merge(low, mid, high);      }  }    // thread function for multi-threading  void\* merge\_sort(void\* arg)  {      // which part out of 4 parts      int thread\_part = part++;        // calculating low and high      int low = thread\_part \* (MAX / 4);      int high = (thread\_part + 1) \* (MAX / 4) - 1;        // evaluating mid point      int mid = low + (high - low) / 2;      if (low < high) {          merge\_sort(low, mid);          merge\_sort(mid + 1, high);          merge(low, mid, high);      }  }    // Driver Code  int main()  {      // generating random values in array      for (int i = 0; i < MAX; i++)          a[i] = rand() % 100;        // t1 and t2 for calculating time for      // merge sort      clock\_t t1, t2;        t1 = clock();      pthread\_t threads[THREAD\_MAX];        // creating 4 threads      for (int i = 0; i < THREAD\_MAX; i++)          pthread\_create(&threads[i], NULL, merge\_sort,                                          (void\*)NULL);        // joining all 4 threads      for (int i = 0; i < 4; i++)          pthread\_join(threads[i], NULL);        // merging the final 4 parts      merge(0, (MAX / 2 - 1) / 2, MAX / 2 - 1);      merge(MAX / 2, MAX/2 + (MAX-1-MAX/2)/2, MAX - 1);      merge(0, (MAX - 1)/2, MAX - 1);        t2 = clock();        // displaying sorted array      cout << "Sorted array: ";      for (int i = 0; i < MAX; i++)          cout << a[i] << " ";        // time taken by merge sort in seconds      cout << "Time taken: " << (t2 - t1) /                (double)CLOCKS\_PER\_SEC << endl;        return 0;  } Factorial using Multithreading #include <iostream>  #include <string>  #include <thread>  #define MAX\_THREAD 5  #include<bits/stdc++.h>  using namespace std;  int part=0, part\_size;  int fact\_nonthreaded(int n){  int res=1;  for(int i=2; i<=n; i++)  res\*=i;  return res;  }  void fact\_threaded(int n, vector<int> &data){  int thread\_part=part++;  int low=thread\_part\*part\_size+1;  int high=(thread\_part+1)\*part\_size;    high=(high>n)?n:high;  int res=1;    for(int i=low; i<=high; i++)  res\*=i;    data[thread\_part]=res;  }  int fact(int n){  int result=1;  vector<int> data(MAX\_THREAD,1);  if(n<=MAX\_THREAD)  return fact\_nonthreaded(n);  part\_size=ceil((double)n/MAX\_THREAD);  thread t[MAX\_THREAD];    for(int i=0; i<MAX\_THREAD; i++){  t[i]=thread(fact\_threaded, n, ref(data));  }    for(int i=0; i<MAX\_THREAD; i++){  t[i].join();  }    for(int i=0; i<MAX\_THREAD; i++){  result\*=data[i];  }  return result;  }  int main()  {  int n;  cin>>n;  cout<<fact(n)<<endl;  } Shared Memory SHARED MEMORY FOR WRITER PROCESS   |  | | --- | | #include <iostream>  #include <sys/ipc.h>  #include <sys/shm.h>  #include <stdio.h>  using namespace std;    int main()  {      // ftok to generate unique key      key\_t key = ftok("shmfile",65);        // shmget returns an identifier in shmid      int shmid = shmget(key,1024,0666|IPC\_CREAT);        // shmat to attach to shared memory      char \*str = (char\*) shmat(shmid,(void\*)0,0);        cout<<"Write Data : ";      gets(str);        printf("Data written in memory: %s\n",str);        //detach from shared memory      shmdt(str);        return 0;  } |   SHARED MEMORY FOR READER PROCESS   |  | | --- | | #include <iostream>  #include <sys/ipc.h>  #include <sys/shm.h>  #include <stdio.h>  using namespace std;    int main()  {      // ftok to generate unique key      key\_t key = ftok("shmfile",65);        // shmget returns an identifier in shmid      int shmid = shmget(key,1024,0666|IPC\_CREAT);        // shmat to attach to shared memory      char \*str = (char\*) shmat(shmid,(void\*)0,0);        printf("Data read from memory: %s\n",str);      //detach from shared memory      shmdt(str);        // destroy the shared memory      shmctl(shmid,IPC\_RMID,NULL);        return 0;  } | |

# OS

## Process synchronization; Peterson’s solution (for 2 processes)

do{

flag[i]=true;

turn=j;

while(flag[j] && turn==j);

CS

flag[i]=false;

RM

}while(TRUE);

Test and Set

boolean TestAndSet(boolean \*target){

boolean \*rv=target;

\*target=TRUE;

Return \*rv;

}

do{

while(TestAndSet(&lock))

;//do nothing

CS

lock=false;

RS

}while(TRUE);

SWAP

do{

key=TRUE;

while(key==TRUE)

swap(&key, &lock);

CS

lock=false;

RS

}while(TRUE);

## SPINLOCK VS MUTEX

**The Theory**

In theory, when a thread tries to lock a mutex and it does not succeed, because the mutex is already locked, it will go to sleep, immediately allowing another thread to run. It will continue to sleep until being woken up, which will be the case once the mutex is being unlocked by whatever thread was holding the lock before. When a thread tries to lock a spinlock and it does not succeed, it will continuously re-try locking it, until it finally succeeds; thus it will not allow another thread to take its place (however, the operating system will forcefully switch to another thread, once the CPU runtime quantum of the current thread has been exceeded, of course).

**The Problem**

The problem with mutexes is that putting threads to sleep and waking them up again are both rather expensive operations, they'll need quite a lot of CPU instructions and thus also take some time. If now the mutex was only locked for a very short amount of time, the time spent in putting a thread to sleep and waking it up again might exceed the time the thread has actually slept by far and it might even exceed the time the thread would have wasted by constantly polling on a spinlock. On the other hand, polling on a spinlock will constantly waste CPU time and if the lock is held for a longer amount of time, this will waste a lot more CPU time and it would have been much better if the thread was sleeping instead.

**The Solution**

Using spinlocks on a single-core/single-CPU system makes usually no sense, since as long as the spinlock polling is blocking the only available CPU core, no other thread can run and since no other thread can run, the lock won't be unlocked either. IOW, a spinlock wastes only CPU time on those systems for no real benefit. If the thread was put to sleep instead, another thread could have run at once, possibly unlocking the lock and then allowing the first thread to continue processing, once it woke up again.

On a multi-core/multi-CPU systems, with plenty of locks that are held for a very short amount of time only, the time wasted for constantly putting threads to sleep and waking them up again might decrease runtime performance noticeably. When using spinlocks instead, threads get the chance to take advantage of their full runtime quantum (always only blocking for a very short time period, but then immediately continue their work), leading to much higher processing throughput.

**The Practice**

Since very often programmers cannot know in advance if mutexes or spinlocks will be better (e.g. because the number of CPU cores of the target architecture is unknown), nor can operating systems know if a certain piece of code has been optimized for single-core or multi-core environments, most systems don't strictly distinguish between mutexes and spinlocks. In fact, most modern operating systems have hybrid mutexes and hybrid spinlocks. What does that actually mean?

A hybrid mutex behaves like a spinlock at first on a multi-core system. If a thread cannot lock the mutex, it won't be put to sleep immediately, since the mutex might get unlocked pretty soon, so instead the mutex will first behave exactly like a spinlock. Only if the lock has still not been obtained after a certain amount of time (or retries or any other measuring factor), the thread is really put to sleep. If the same code runs on a system with only a single core, the mutex will not spinlock, though, as, see above, that would not be beneficial.

A hybrid spinlock behaves like a normal spinlock at first, but to avoid wasting too much CPU time, it may have a back-off strategy. It will usually not put the thread to sleep (since you don't want that to happen when using a spinlock), but it may decide to stop the thread (either immediately or after a certain amount of time) and allow another thread to run, thus increasing chances that the spinlock is unlocked (a pure thread switch is usually less expensive than one that involves putting a thread to sleep and waking it up again later on, though not by far).

## Difference between Loading and Linking

[**Linking**](https://www.geeksforgeeks.org/compiler-design-linker/) and **Loading** are the utility programs that play a important role in the execution of a program. Linking intakes the object codes generated by the assembler and combines them to generate the executable module. On the other hand, the loading loads this executable module to the main memory for execution.

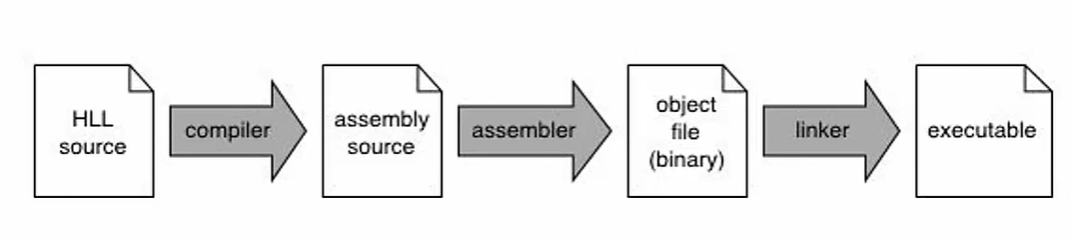
**Loading:**  
Bringing the program from secondary memory to main memory is called Loading.

**Linking:**  
Establishing the linking between all the modules or all the functions of the program in order to continue the program execution is called linking.

Dynamic loading refers to mapping (or less often copying) an executable or library into a process's memory after it has started. Dynamic linking refers to resolving symbols - associating their names with addresses or offsets - after compile time.

* **Dynamic loading, static linking**. The executable has an address/offset table generated at compile time, but the actual code/data aren't loaded into memory at process start. This is not the way things tend to work in most systems nowadays, but it would describe some old-fashioned overlay systems. I'd also be utterly unsurprised if some current embedded systems work this way too. In either case, the goal is to give the programmer control over memory use while also avoiding the overhead of linking at runtime.
* **Static loading, dynamic linking**. This is how dynamic libraries specified at compile time usually work. The executable contains a reference to the dynamic/shared library, but the symbol table is missing or incomplete. Both loading and linking occur at process start, which is considered "dynamic" for linking but not for loading.
* **Dynamic loading, dynamic linking**. This is what happens when you call *dlopen* or its equivalent on other systems. The object file is loaded dynamically under program control (i.e. after start), and symbols both in the calling program and in the library are resolved based on the process's possibly-unique memory layout at that time.
* **Static loading, static linking**. Everything is resolved at compile time. At process start everything is loaded into memory immediately and no extra resolution (linking) is necessary. In the abstract it's not necessary for the loading to occur from a single file, but I don't think the actual formats or implementations (at least those I'm familiar with) can do multi-file loading without dynamic linking.

## Compilation process



## Process Schedulers

* Long term (Job schedular)
  + Brings new process to ready state
  + Degree of multiprogramming
* Short term
  + Selecting process from ready state to schedule it to running state

Dispatcher: brings process to running state

* Medium term schedular
  + Responsible for suspending and resuming the process.

## Segmentation

In Operating Systems, Segmentation is a memory management technique in which, the memory is divided into the variable size parts. Each part is known as segment which can be allocated to a process.

No external fragmentation.

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. No internal fragmentation.

## Threads

Threads are not independent of one another like processes are, and as a result threads share with other threads their code section, data section, and OS resources (like open files and signals). But, like process, a thread has its own program counter (PC), register set, and stack space.

When a process is forked, open file tables(list of file descriptors) will be copied to child thread.

## Multiprogramming, multitasking, multithreading and multiprocessing

1. **Multiprogramming –** A computer running more than one program at a time (like running Excel and Firefox simultaneously). Context switching.
2. **Multiprocessing –** A computer using more than one CPU at a time. parallel processing
3. **Multitasking –** Tasks sharing a common resource (like 1 CPU).
4. **Multithreading** is a logical extension of multitasking. (time sharing alongside the concept of context switching)

## Reader-Writer problem

Suppose that a database is to be shared among several concurrent processes.

Readers: want to read database

Writers: want to update database

**first readers-writers problem**, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. (writers may starve).

Solution:

semaphore mutex, wrt;

int readcount;

The structure of a writer process:

do {

wait(wrt);

II writing is performed

signal(wrt);

} while (TRUE);

The structure of a reader process:

do {

wait (mutex);

readcount++;

if (readcount == 1)

wait (wrt);

signal(mutex);

II reading is performed

wait(mutex);

readcount--;

if (readcount == 0)

signal(wrt);

signal(mutex);

} while (TRUE);

**The second readers-writers,** problem requires that, once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading. (readers may starve.)

## Multi-threading

[**<atomic>**](http://www.cplusplus.com/reference/atomic/)

[**<thread>**](http://www.cplusplus.com/reference/thread/)

[**<mutex>**](http://www.cplusplus.com/reference/mutex/)

[**<condition\_variable>**](http://www.cplusplus.com/reference/condition_variable/)

[**<future>**](http://www.cplusplus.com/reference/future/)

## Producer-Consumer

|  |
| --- |
| /\*\* |
|  | \* Producer-Comsumer example, written in C++ May 4, 2014 |
|  | \* Compiled on OSX 10.9, using: |
|  | \* g++ -std=c++11 conpro.cpp |
|  | \*\*/ |
|  |  |
|  | #include <iostream> |
|  | #include <thread> |
|  | #include <mutex> |
|  | #include <condition\_variable> |
|  |  |
|  | std::mutex mtx; |
|  | std::condition\_variable cv; |
|  |  |
|  | int meal = 0; |
|  |  |
|  | /\* consumer \*/ |
|  | void waiter(int ordernumber){ |
|  | std::unique\_lock<std::mutex> lck(mtx); |
|  | while(meal == 0) cv.wait(lck); |
|  | std::cout << "Order: "; |
|  | std::cout << ordernumber + 1 << " being taken care of with "; |
|  | std::cout << meal - 1 << " meals also ready." << std::endl; |
|  | meal--; |
|  | } |
|  |  |
|  | /\* Producer \*/ |
|  | void makeMeal(int ordernumber){ |
|  | std::unique\_lock<std::mutex> lck(mtx); |
|  | meal++; |
|  | cv.notify\_one(); |
|  | } |
|  |  |
|  | int main(){ |
|  |  |
|  | std::thread chefs[10]; |
|  | std::thread waiters[10]; |
|  |  |
|  | /\* Initialize customers and cheifs \*/ |
|  | for (int order = 0; order < 10; order++){ |
|  | chefs[order] = std::thread(makeMeal, order); |
|  | waiters[order] = std::thread(waiter, order); |
|  | } |
|  |  |
|  | /\* Join the threads to the main threads \*/ |
|  | for (int order = 0; order < 10; order++) { |
|  | waiters[order].join(); |
|  | chefs[order].join(); |
|  | } |
|  |  |
|  | return 0; |
|  | } |

## Booting

**Booting** is what happens when a [computer](https://simple.wikipedia.org/wiki/Computer) starts. This happens when the power is turned on. It is called "reboot" if it happens at other times. When you boot a computer, your [processor](https://simple.wikipedia.org/wiki/Processor) looks for instructions in system [ROM](https://simple.wikipedia.org/wiki/ROM) (the [BIOS](https://simple.wikipedia.org/wiki/BIOS)) and executes them. They normally 'wake up' [peripheral equipment](https://simple.wikipedia.org/wiki/Peripheral_equipment) and search for the [boot device](https://simple.wikipedia.org/wiki/Boot_device). The [boot device](https://simple.wikipedia.org/wiki/Boot_device) either loads the [operating system](https://simple.wikipedia.org/wiki/Operating_system) or gets it from someplace else.

POST (Power One Self Test) check for hardware.

Search for Bootstrap loader to load OS.

<https://www.howtogeek.com/398493/what-exactly-happens-when-you-turn-on-your-computer/>

## Thread Synchronization

<https://www.codeproject.com/Articles/598695/Cplusplus11-Threads-Locks-and-Condition-Variables>

## File system calls

<http://www2.cs.uregina.ca/~hamilton/courses/330/notes/unix/filesyscalls.html>

## Pipes

#### Pipe System Call

* pipe() is a system call that facilitates inter-process communication. It opens a **pipe**, which is an area of main memory that is treated as a "virtual file". The pipe can be used by the creating process, as well as all its child processes, for reading and writing.
* One process can write to this "virtual file" or pipe and another related process can read from it.
* If a process tries to read before something is written to the pipe, the process is suspended until something is written.
* The pipe system call finds the first two available positions in the process's open file table and allocates them for the read and write ends of the pipe. Recall that the open system call allocates only one position in the open file table.

Syntax in a C/C++ program:

#include <unistd.h>

int pip[2];

(void) pipe(pip);

With error checking:

#include <unistd.h>

int pip[2];

int result;

result = pipe(pip);

if (result == -1)

{

perror("pipe");

exit(1);

}

Programming notes:

* The pipe() system call is passed a pointer to the beginning of an array of two integers.
* It appears in C/C++ as if pipe() is passed the name of the array without any brackets.
* The system call places two integers into this array. These integers are the file descriptors of the first two available locations in the open file table.
* pip[0] - the read end of the pipe - is a file descriptor used to read from the pipe
* pip[1] - the write end of the pipe - is a file descriptor used to write to the pipe

The buffer size for the pipe is fixed. Once it is full all further writes are blocked. Pipes work on a first in first out basis.

#### Brief Example

* no error checking, no closing of pipe

#include <unistd.h>

int main()

{

int pid, pip[2];

char instring[20];

pipe(pip);

pid = fork();

if (pid == 0) /\* child : sends message to parent\*/

{

/\* send 7 characters in the string, including end-of-string \*/

write(pip[1], "Hi Mom!", 7);

}

else /\* parent : receives message from child \*/

{

/\* read from the pipe \*/

read(pip[0], instring, 7);

}

}

# Message Queue

|  |
| --- |
| We don’t have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.  Process must share a common key in order to gain access to the queue in the first place.  // C Program for Message Queue (Writer Process)  #include <stdio.h>  #include <sys/ipc.h>  #include <sys/msg.h>    // structure for message queue  struct mesg\_buffer {      long mesg\_type;      char mesg\_text[100];  } message;    int main()  {      key\_t key;      int msgid;        // ftok to generate unique key      key = ftok("progfile", 65);        // msgget creates a message queue      // and returns identifier      msgid = msgget(key, 0666 | IPC\_CREAT);      message.mesg\_type = 1;        printf("Write Data : ");      gets(message.mesg\_text);        // msgsnd to send message      msgsnd(msgid, &message, sizeof(message), 0);        // display the message      printf("Data send is : %s \n", message.mesg\_text);        return 0;  } |

MESSAGE QUEUE FOR READER PROCESS