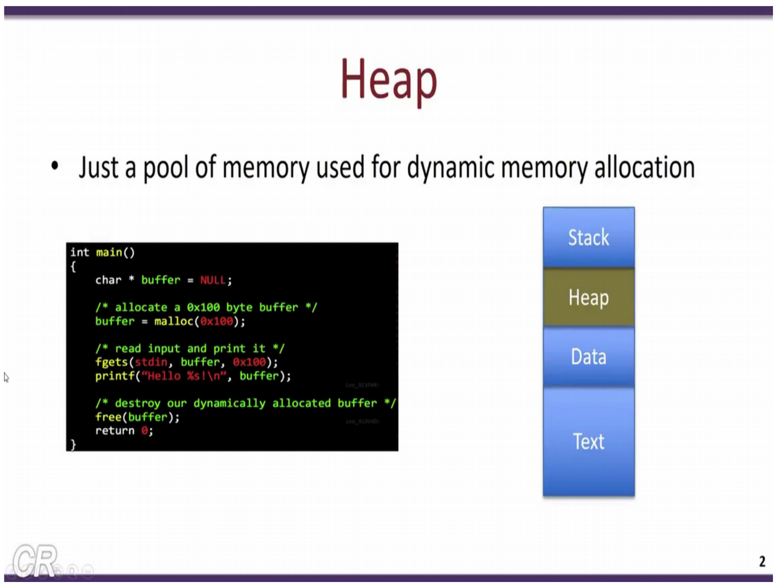
Information Security 5 Secure System Engineering
Professor Chester Rebeiro
Indian Institute of Technology Madras
Heap Exploits

Hello and welcome to this lecture in the course of secure system engineering. In today's lecture we will look at exploits that target the heap.

(Refer Slide Time: 0:26)



Heap

- Just a pool of memory used for dynamic memory allocation

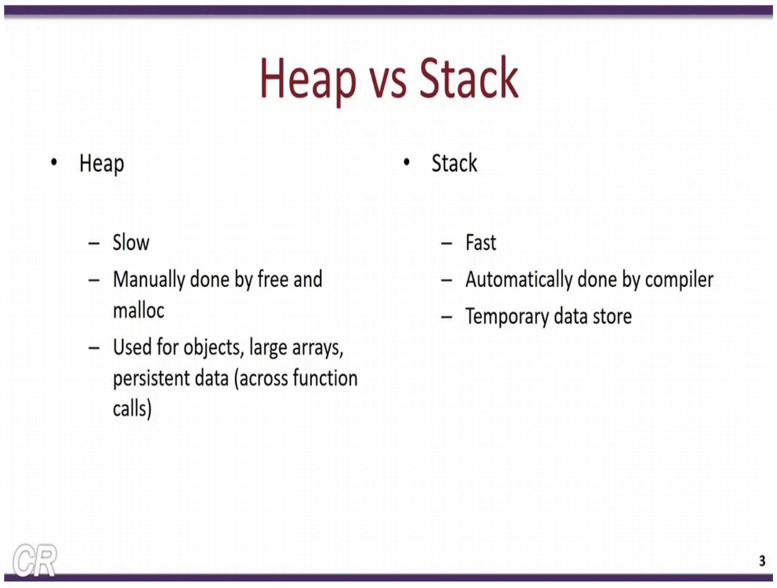
```c
int main()
{
    char * buffer = NULL;

    /* allocate a 0x100 byte buffer */
    buffer = malloc(0x100);

    /* read input and print it */
    fgets(stdin, buffer, 0x100);
    printf("Hello %s!\n", buffer);

    /* destroy our dynamically allocated buffer */
    free(buffer);
    return 0;
}
```

Stack
Heap
Data
Text

So as we know heap is essentially a pool of memory present in the processes address space where dynamically allocated memory resides, so every time for example that we use a malloc in a C program a chunk of memory gets allocated in the heap and when we free that memory then the chunk of memory gets freed, so in today's lecture we will actually be looking at how this dynamically allocated memory or in other words how malloc handles or manages the memory that is present in the heap, so let us just look at this motivating example.

So we have a small C program here and what we have here is an invocation to malloc which request 256 bytes, so when this malloc gets invoked as we know that in the process address space a chunk of memory of the size which is approximately 256 bytes would get allocated and the pointer to that memory is present in buffer. Now given this particular pointer to can read or write or manipulate data present in that heap chunk and at the end of usage we can then free to the buffer and after the buffer is freed that chunk of 256 bytes which we have just allocated in the heap can be used by other malloc which may be present in the program.

(Refer Slide Time: 2:03)



Heap vs Stack

- Heap
  - Slow
  - Manually done by free and malloc
  - Used for objects, large arrays, persistent data (across function calls)
- Stack
  - Fast
  - Automatically done by compiler
  - Temporary data store

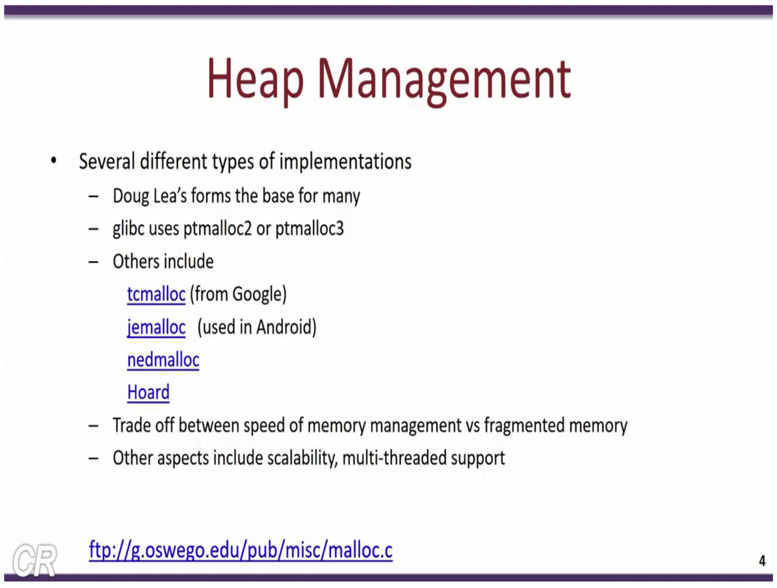
CR 3

So what is the difference between a heap and a stack? As you know stacks are used during function invocations and to pass parameters to functions as well as for local variables, so stacks essentially are fast they are fully managed by the compiler and what we mean by this is that we do not have to explicitly have statements which says create a particular area in the stack and free that area in the stack. On the other hand when you compile a program the compiler would insert instructions that would allocate stack that is a stack frame and free the stack frame every time a function is entered or returned respectively.

As we know stacks are used as temporary data storage, so whenever a function returns then the local variables which was allocated in the function is no more available. On the other hand heaps are extremely slow every one allocate some memory in the heap you have to explicitly invoke the library function malloc and similarly when you want to free that memory knew how to invoke the free call.

So this management is done by the program and quite often this could actually result in several different types of vulnerabilities as we will see during this lecture. Heaps are used for storage of objects large array and global data and typically all you require is to have a pointer to that particular malloc junk and you would be able to access that malloc data from any function.

(Refer Slide Time: 3:54)



Heap Management

- Several different types of implementations
  - Doug Lea's forms the base for many
  - glibc uses ptmalloc2 or ptmalloc3
  - Others include
    - tcmalloc (from Google)
    - jemalloc (used in Android)
    - nedmalloc
    - Hoard
  - Trade off between speed of memory management vs fragmented memory
  - Other aspects include scalability, multi-threaded support

ftp://g.oswego.edu/pub/misc/malloc.c

4

So what we will see in his lecture is how malloc manages the heap memory, how it allocates memory? What are the algorithms used for this allocation? Similarly what are the algorithms used for freeing the memory? What are the data structure used internally and so on? So there are a different variety of malloc implementations that are present, the tcmalloc for example is from Google, jemalloc is implementing in android operating systems and similarly you have nedmalloc and Hoard.

In all of these different malloc implementation there is a shuttle difference between the algorithm used to allocate memory and free memory as well, so essentially the algorithms will affect the speed at which memory is allocated or deallocated versus the fragmentation of the memory, so for instance you may have one implementation of malloc which very quickly can allocate and deallocate memory. In other words the time taken for malloc and the time taken for free is extremely small however it is generally what is seen. Such fast implementation of malloc would quite often result in insufficient use of the heap memory.

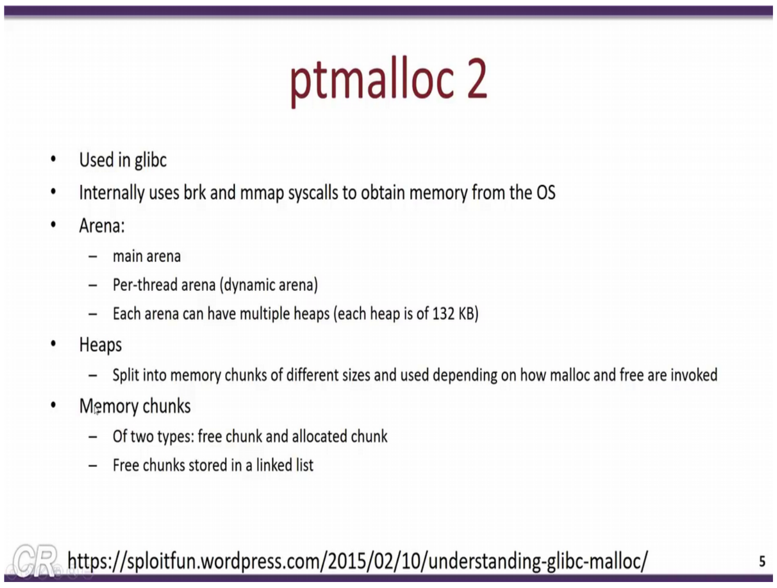
The concept of fragmentation of the memory sets in by which there will be tiny chunks of memory in the heap which is not going to be used, so all of these different types of malloc implementations. Trade-off between the speed of memory management versus the fragmented memory by speed of memory management imply the speed with which malloc can allocate memory as well as the speed with which free and deallocate memory quite often it is seen that implementations of malloc where malloc runs extremely fast would often result in fragmented memory.

So what we mean by fragmented memory is that they would be tiny chunk of memory may be of 2 or 4 or 8 bytes which are too small to be actually used by any program, so by these fragmented memory just reside in the heap and is of not much use, so essentially when malloc implementations are made they would have to trade-off between the memory management scheme so as to reduce the amount of fragmentation present.

At the same time ensure that the speed of memory management is good enough so that the performance of the application is not affected too much. Now these different malloc implementations also vary in the support that they provide for example the scalability which means what is the size of the heap that the particular malloc implementation can manage. The other aspect that comes into play is the multithreaded support, can the heap implementation actually provide support for programs which have multithreading present in it?

So in this particular lecture we will focus on one specific malloc implementation known as ptmalloc2, so ptmalloc2 is very common implementation which is used in gilibc, so most likely when you actually run your typical hello world program and you have malloc in it. It uses gilibc and hence the malloc would invoke a function which is present in ptmalloc2. Now all of these different types of malloc implementations originate from one specific implementation by Doug Lea you could actually search for Doug Lea and find the 1st implementation of malloc most other malloc implementations are derived from Doug Lea's implementation.

(Refer Slide Time: 8:08)



ptmalloc 2

- Used in glibc
- Internally uses brk and mmap syscalls to obtain memory from the OS
- Arena:
  - main arena
  - Per-thread arena (dynamic arena)
  - Each arena can have multiple heaps (each heap is of 132 KB)
- Heaps
  - Split into memory chunks of different sizes and used depending on how malloc and free are invoked
- Memory chunks
  - Of two types: free chunk and allocated chunk
  - Free chunks stored in a linked list

https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/ 5

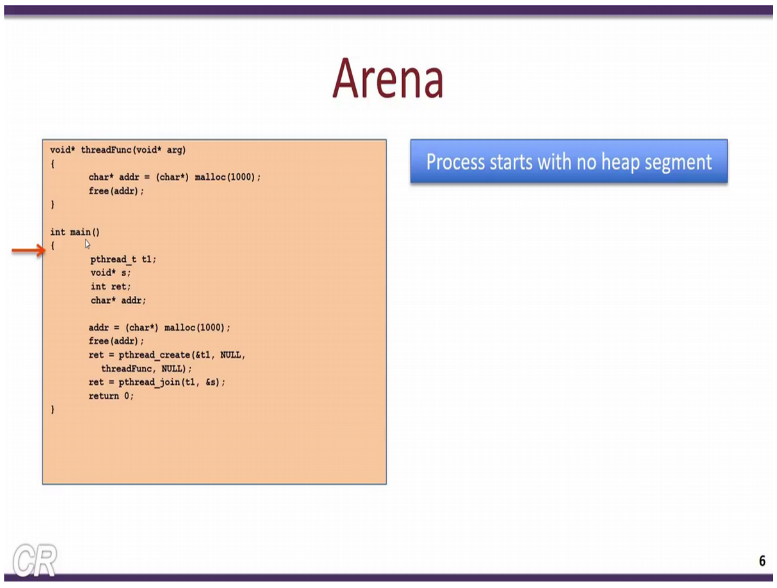
So let us look a little more in detail about ptmalloc2 note that while this is quite common in most recent daily Linux systems the 3rd version of that which is known as ptmalloc3 is also present, so there are subtle differences between ptmalloc2 and ptmalloc3 and we would not go into the details of these differences. In this lecture on the other hand we will be only focusing on the internals of ptmalloc2.

Now as we know ptmalloc2 is present in the gilibc library which is linked with all standard C or C plus plus programs that you write on a typical Linux system. Now internally ptmalloc2 uses those systems calls to obtain memory from the operating system, so the system calls are known as breaks and mmap, so whenever break or mmap is invoked so it leads to the operating system getting executed and the operating systems would allocate a chunk of memory for that particular process.

So for example if you write a C program and the 1st time you invoke the malloc call with that C program internally what malloc is going to do is that it is going to invoke the brake system call, so when the brake system call gets executed by the kernel the kernel would allocate a chunk of memory of size 132 kilobytes to the particular process, so that 132 kilobytes is used by the ptmalloc2 for its heap space, so within this particular area you have different components you have something known as Arena, then within the arena you have heaps and within the heaps you memory chunks.

So whenever your program invokes malloc for the first time internally ptmalloc2 could invoke the break system call of brk, so when break gets executed it causes really operating systems kernel to execute and the OS would allocate 132 kilobytes of memory for that particular process when break returns the ptmalloc code would then have algorithms to manage that 132 kilobytes of memory, so that memory is divided into multiple different components, so the largest component is known as the arena, the hyena is then split into heaps and then there are many chunks, right? So the memory chunks are present within the heap, now let us 1st look of an example of how an arena is used.

(Refer Slide Time: 11:09)



Arena

```
void* threadFunc(void* arg)
{
        char* addr = (char*) malloc(1000);
        free(addr);
}

int main()
{
        pthread_t t1;
        void* s;
        int ret;
        char* addr;

        addr = (char*) malloc(1000);
        free(addr);
        ret = pthread_create(&t1, NULL,
          threadFunc, NULL);
        ret = pthread_join(t1, &s);
        return 0;
}
```

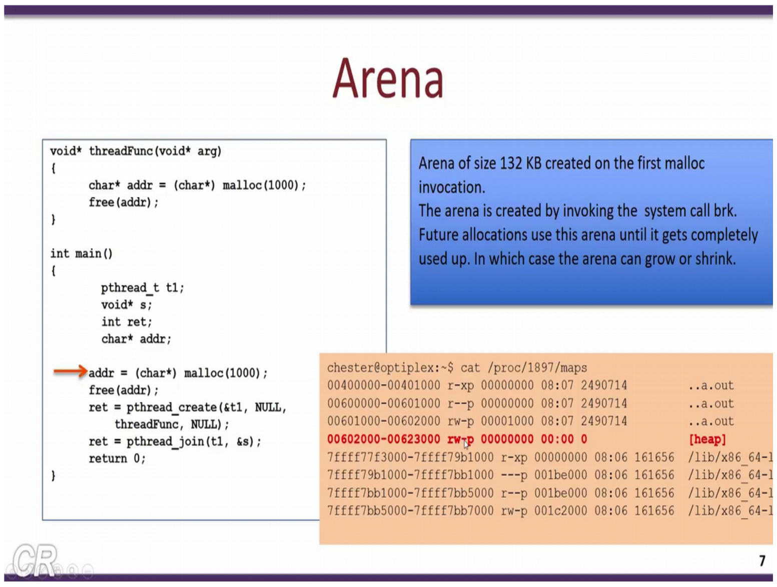
Process starts with no heap segment

6

So let us look at this particular program, so as you see this is a simple program which uses threads, so we have a statement over here where you allocate thousand bytes and then return an address then you free that address in this particular statement you create a thread using the pthread library and invoking the function pthread underscore create, so this function would then create a thread which would start executing from this thread form that is present over here, so at the end of this invocation of pthread create we have actually have the single process which has 2 threads the main thread and the threadfunc thread.

Now we invoke the pthread joint in order to ensure that the main thread waits for the thread function to complete before continuing its execution. In other words the pthread joint would block the main thread until the child thread or the threadfunc completes its execution and then the function would return. Now in the thread that we just created we allocate another thousand bytes and then free that particular thousand bytes, now we will see what happens to the heap and we actually start executing this program? As soon as the program starts it would be surprisingly for you to know that the size of the heap is initially 0, so essentially the program would start with absolutely no heap segment.

(Refer Slide Time: 12:45)



Arena

```
void* threadFunc(void* arg)
{
      char* addr = (char*) malloc(1000);
      free(addr);
}

int main()
{
        pthread_t t1;
        void* s;
        int ret;
        char* addr;

      addr = (char*) malloc(1000);
      free(addr);
      ret = pthread_create(&t1, NULL,
            threadFunc, NULL);
      ret = pthread_join(t1, &s);
      return 0;
}
```

Arena of size 132 KB created on the first malloc invocation.
The arena is created by invoking the system call brk.
Future allocations use this arena until it gets completely used up. In which case the arena can grow or shrink.

```
chester@optiplex:~$ cat /proc/1897/maps
00400000-00401000 r-xp 00000000 08:07 2490714          ..a.out
00600000-00601000 r--p 00000000 08:07 2490714          ..a.out
00601000-00602000 rw-p 00001000 08:07 2490714          ..a.out
00602000-00623000 rw-p 00000000 00:00 0                [heap]
7ffff77f3000-7ffff79b1000 r-xp 00000000 08:06 161656  /lib/x86_64-l
7ffff79b1000-7ffff7bb1000 ---p 001be000 08:06 161656  /lib/x86_64-l
7ffff7bb1000-7ffff7bb5000 r--p 001be000 08:06 161656  /lib/x86_64-l
7ffff7bb5000-7ffff7bb7000 rw-p 001c2000 08:06 161656  /lib/x86_64-l
```

7

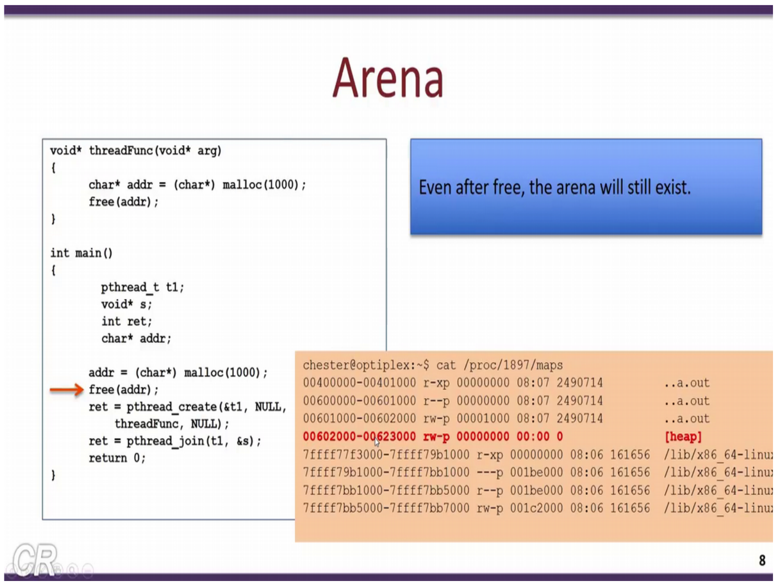
Now when malloc first gets executed in this particular statement over here it results in ptmalloc code that we have been talking about to get executed. Now when this particular statement comprising of the 1st malloc of this particular program gets executed it results in the malloc function present in their ptmalloc library to be invoked. Now the ptmalloc would determining that the heap segment is 0 and then it would invoke the break system call. As we have seen the brake system call would invoke the operating system and the operating system would allocate a chunk of 132 kilobytes for this particular process. Now ptmalloc would obtain that 132 kilobytes it would split it into 2 parts, now one part is roughly around thousand bytes and this…

Now one part is roughly around thousand bytes and a pointer to this part is what is written by a malloc and assigned to address, so the remaining part is the free-part. Now every subsequent malloc from the main thread would then utilise this particular free part of memory and therefore subsequent locations to malloc would not To actually request the operating system for the memory. So this large area of 132 kilobytes which the operating system have provided will then be used by all subsequent malloc in the main thread until that entire 132 kilobytes gets is completely utilised.

Now when this entire 132 kilobytes of memory is completely utilised by the main thread a subsequent malloc would then invoke the operating system again and then get a new chunk of 132 kilobytes, so in this way you see that the operating system gets invoked only when there is no available space in the heap segment to satisfy a particular request, so what you see over here is the memory map of the particular process, so any Linux based system if you actually cat this particular file that is slash proc slash PID of that process slash maps it will give you the entire virtual address space for that particular process.

In this case the PID of the process this 1897 and therefore slash proc slash 1897 slash maps would give you the virtual address space for that particular process. Now at this particular point when the execution has just completed malloc, so what you see over here is that after this particular malloc a new segment gets allocated in the program, so this segment is the heap. Segment starts at 602000 to 623000 and if you can actually subtract these 2 you would see that the size of this particular segment is 132 kilobytes. Also notice that you have read write permission for this segment which means that you could read the data from that particular segment or write data to that particular segment. In other words it is a regular data segment. Code cannot be present in that segment because it is not an executable segment.

(Refer Slide Time: 16:29)



Arena

```
void* threadFunc(void* arg)
{
      char* addr = (char*) malloc(1000);
      free(addr);
}

int main()
{
        pthread_t t1;
        void* s;
        int ret;
        char* addr;

      addr = (char*) malloc(1000);
      free(addr);
      ret = pthread_create(&t1, NULL,
            threadFunc, NULL);
      ret = pthread_join(t1, &s);
      return 0;
}
```

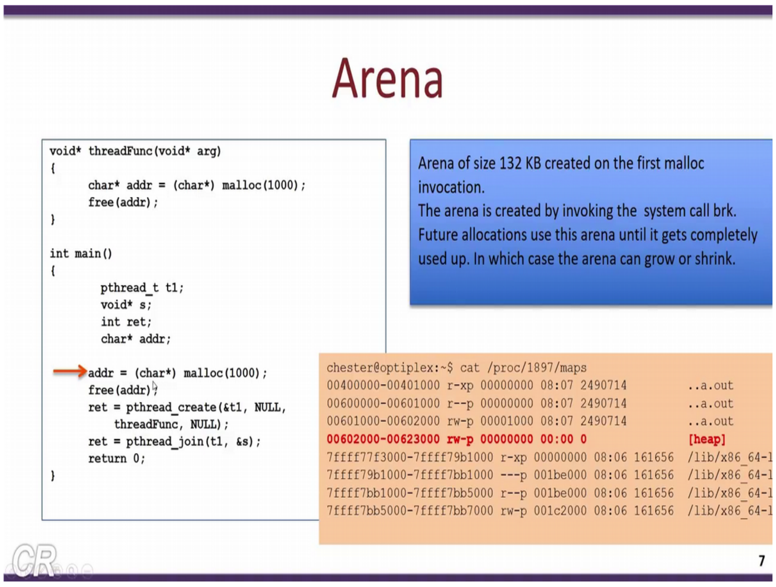
Even after free, the arena will still exist.

```
chester@optiplex:~$ cat /proc/1897/maps
00400000-00401000 r-xp 00000000 08:07 2490714          ..a.out
00600000-00601000 r--p 00000000 08:07 2490714          ..a.out
00601000-00602000 rw-p 00001000 08:07 2490714          ..a.out
00602000-00623000 rw-p 00000000 00:00 0                [heap]
7ffff77f3000-7ffff79b1000 r-xp 00000000 08:06 161656   /lib/x86_64-linu
7ffff79b1000-7ffff7bb1000 ---p 001be000 08:06 161656   /lib/x86_64-linu
7ffff7bb1000-7ffff7bb5000 r--p 001be000 08:06 161656   /lib/x86_64-linu
7ffff7bb5000-7ffff7bb7000 rw-p 001c2000 08:06 161656   /lib/x86_64-linu
```

8

Now when other program next invokes the free function with that particular address, what we notice from the maps is that the heap segment is still present. Next what we will do is we will free that particular address, in other words we are freeing this thousand bytes which has just got allocated and what you would see from the memory maps is that even after freeing that particular address the heap still remains the same that there is even though the heap is not being used by this program at this particular instant of time, so what this means is that even though the heap is not being used after this particular point in time by the program, nevertheless the heap segment is still present in the virtual address space of the particular program.

(Refer Slide Time: 17:22)



Arena

```
void* threadFunc(void* arg)
{
      char* addr = (char*) malloc(1000);
      free(addr);
}

int main()
{
        pthread_t t1;
        void* s;
        int ret;
        char* addr;

      addr = (char*) malloc(1000);
      free(addr);
      ret = pthread_create(&t1, NULL,
          threadFunc, NULL);
      ret = pthread_join(t1, &s);
      return 0;
}
```

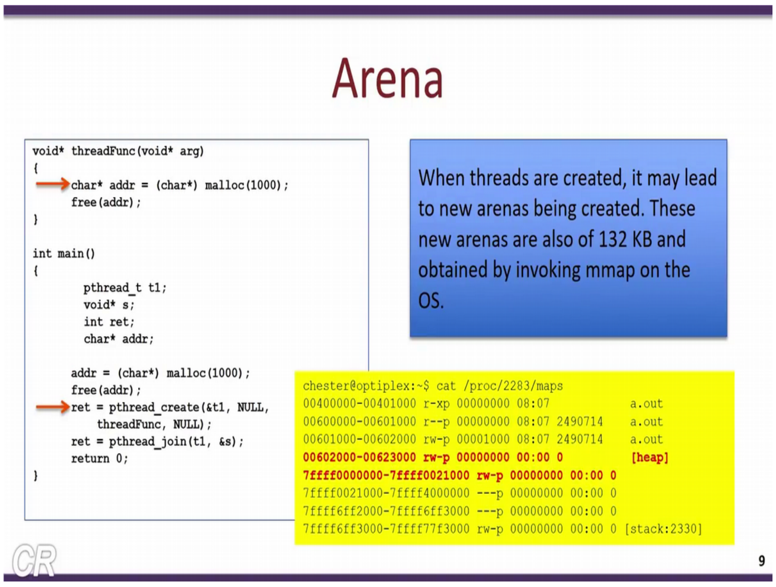
Arena of size 132 KB created on the first malloc invocation.
The arena is created by invoking the system call brk.
Future allocations use this arena until it gets completely used up. In which case the arena can grow or shrink.

```
chester@optiplex:~$ cat /proc/1897/maps
00400000-00401000 r-xp 00000000 08:07 2490714         ..a.out
00600000-00601000 r--p 00000000 08:07 2490714         ..a.out
00601000-00602000 rw-p 00001000 08:07 2490714         ..a.out
00602000-00623000 rw-p 00000000 00:00 0               [heap]
7ffff77f3000-7ffff79b1000 r-xp 00000000 08:06 161656  /lib/x86_64-l
7ffff79b1000-7ffff7bb1000 ---p 001be000 08:06 161656  /lib/x86_64-l
7ffff7bb1000-7ffff7bb5000 r--p 001be000 08:06 161656  /lib/x86_64-l
7ffff7bb5000-7ffff7bb7000 rw-p 001c2000 08:06 161656  /lib/x86_64-l
```

7

Next we will see what happens when you actually create a thread, so as we know and you invoke the pthread underscore create it results in a new thread getting created which is present here. Now in this child thread we malloc another thousand bytes. Now it will be surprising for you to know that when you malloc this thousand bytes what happens in ptmalloc is that it would result in a new chunk of memory getting allocated, so within this particular thread 1st invocation to malloc in that thread would again invoke the things system to the mmap system call and obtain another 132 kilobytes, so this entire area of 132 kilobytes which is allocated by the main thread of the program is known as the main arena. Now this main arena is split into various sub heaps and used to satisfy various malloc invocations from the main thread of the program.

(Refer Slide Time: 18:32)



Arena

```
void* threadFunc(void* arg)
{
    char* addr = (char*) malloc(1000);
    free(addr);
}

int main()
{
        pthread_t t1;
        void* s;
        int ret;
        char* addr;

    addr = (char*) malloc(1000);
    free(addr);
    ret = pthread_create(&t1, NULL,
        threadFunc, NULL);
    ret = pthread_join(t1, &s);
    return 0;
}
```

When threads are created, it may lead to new arenas being created. These new arenas are also of 132 KB and obtained by invoking mmap on the OS.

```
chester@optiplex:~$ cat /proc/2283/maps
00400000-00401000 r-xp 00000000 08:07              a.out
00600000-00601000 r--p 00000000 08:07 2490714      a.out
00601000-00602000 rw-p 00001000 08:07 2490714      a.out
00602000-00623000 rw-p 00000000 00:00 0            [heap]
7ffff0000000-7ffff0021000 rw-p 00000000 00:00 0
7ffff0021000-7ffff4000000 ---p 00000000 00:00 0
7ffff6ff2000-7ffff6ff3000 ---p 00000000 00:00 0
7ffff6ff3000-7ffff77f3000 rw-p 00000000 00:00 0 [stack:2330]
```

9

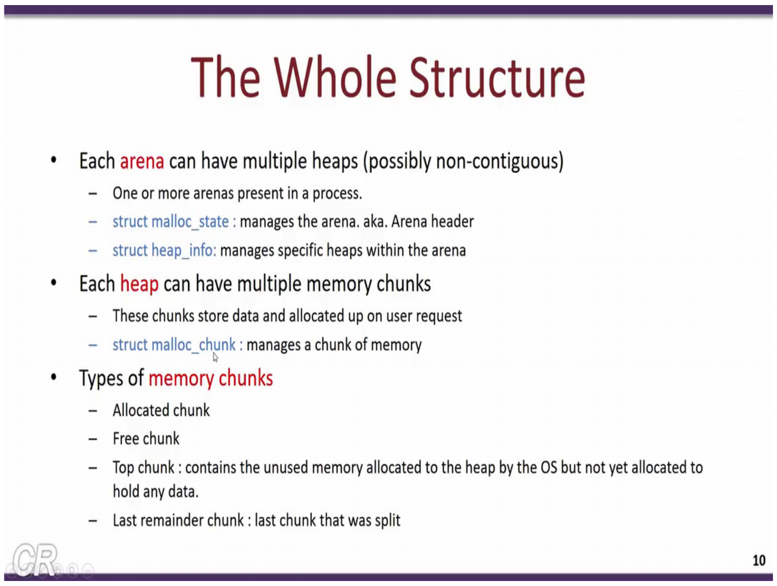
Now let us see what would happen when we invoke the pthread underscore create, so as we know pthread underscore create would create a thread function and the new thread could start shooting this particular function, so in this function we invoke malloc again and request another thousand bytes. The pointer to this thousand bytes is then stored in this local pointer address. When the malloc from this thread gets invoked for the 1st time what would happen is that the ptmalloc code would determining that this is a new thread and is the 1st invocation to malloc from that thread and therefore it will request the operating system to issue another 132 kilobytes of memory.

So letters see what happens when we actually created thread using the pthread underscore create function which starts a thread known as threadfunc, so the threadfunc starts to execute from this particular function. Now when we invoke malloc in this particular function that is in the thread function what would happen is that ptmalloc would determine that the 1st malloc request from the new thread and therefore it would request the operating system for other chunk of memory.

The operating system would then allocate another 132 kilobytes for that particular thread, so every malloc and free in this thread function will use this newly allocated region right, so this region is also managed as an arena. Now we will look at the virtual address map for this particular process at the point when this malloc has got invoked we will see that a new segment has just been created, the segment starts at 7ffff followed by 7 zeros to 7 ffff0021000, so you note that this particular region is also of 132 kilobytes, so every malloc that gets created from this particular thread use this Padilla segment for its allocation.

On the other hand every malloc that is invoked from the main thread would use this segment for its allocation, so this allocation is where you have the main arena and this newly created segment is where we would have arena for the thread of the thread arena, so in this way what we see is that it is likely that every thread that we create in a particular program gets a separate segment.

(Refer Slide Time: 21:29)



The Whole Structure

- Each arena can have multiple heaps (possibly non-contiguous)
  - One or more arenas present in a process.
  - struct malloc_state : manages the arena. aka. Arena header
  - struct heap_info: manages specific heaps within the arena
- Each heap can have multiple memory chunks
  - These chunks store data and allocated up on user request
  - struct malloc_chunk : manages a chunk of memory
- Types of memory chunks
  - Allocated chunk
  - Free chunk
  - Top chunk : contains the unused memory allocated to the heap by the OS but not yet allocated to hold any data.
  - Last remainder chunk : last chunk that was split

10

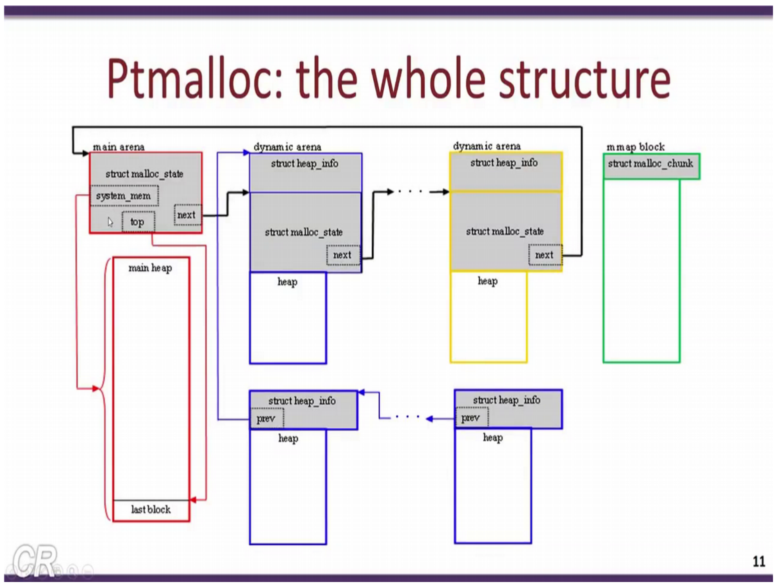
Now let us look at the entire structure of this heap memory, so as we know that one of the main components in the heap memory is the arena, so we have the main arena which comprises of the arena which is used by the main thread of the program and you could have various thread arenas corresponding to each thread that the process invokes. Now therefore in one process we could have multiple arena arc that are present, now if you look at the ptmalloc2 code you would see that there is a structure known as malloc underscore state which is used to manage the arenas, now each arena can have multiple heaps.

So if you look in the ptmalloc code and into the structure called malloc underscore state you would see that there is an instantiation of heap underscore info which would actually be a pointer to a particular heap. Now let us look at the whole structure of how the heap is managed as we know the main component in the heap is the arena. One program could have one or more arenas which are present, now each arena is of 132 kilobytes and these arenas are created by invocations to the operating system.

Now further what happens is that each arena can have multiple heaps, each of these heaps could be possibly non-contiguous and if you look at ptmalloc2 code you would see that there are 2 structures that are present the malloc underscore state structure is essentially used to manage arena, so this is also known as the end I had and it is present in memory and contains various meta data and other information to manage the arena.

Similarly if you look at the ptmalloc2 code you also have another structure known as heap underscore info, so this particular structure present as the meta data would be used to manage specific heaps with an arena. Now each heap can have multiple memory chunks, so these chunks could be allocated or unallocated which is also known as of free chunk and you could also have something known as a top chunk for a last remainder chunk. Now we will look at what this top chunk is and what last remainder chunk is later on but at this particular point of time it is interesting to note that the structure that actually manages this heap memory is known as the malloc underscore chunk, so you can look up the ptmalloc2 code and locate this malloc underscore chunk and see all the entries that are used to manage a particular memory chunk.

(Refer Slide Time: 24:47)


Ptmalloc: the whole structure

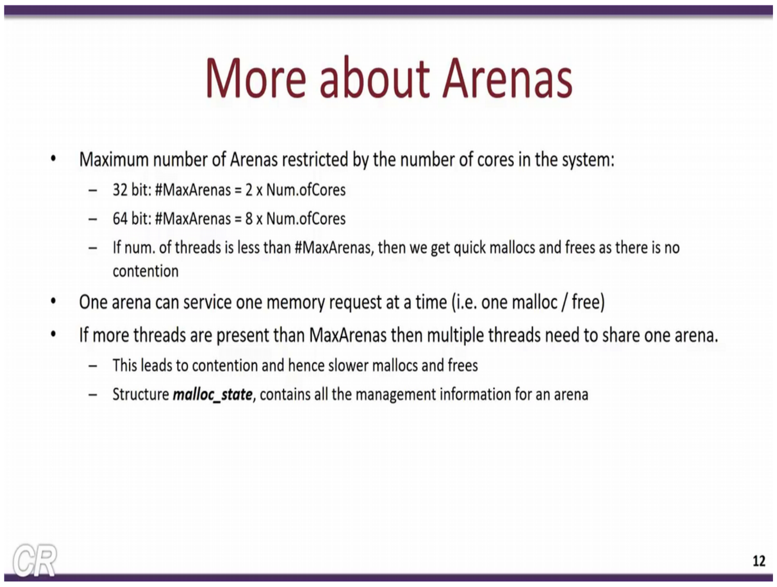
So this particular slide shows the entire structure of ptmalloc heap this particular entry over here is the main arena, so it is define as the struct malloc underscore state. Now within this particular structure there is an entry known as system map which is a pointer to the main heap, so now the main heap is actually allocated over here it is roughly slightly less than 132 kilobytes and we have top pointer which is pointing to the last block which is present in the heap, so this top pointer can be used determine when this heap is completely full.

Now every other arena that gets created would also have an arena structure, so we have another arena displayed over here so this is a thread arena also known as dynamic arena, so within this particular thread arena we again have a struck malloc underscore state which essentially contains the meta data for this arena that has just got created, so in addition to the main arena of particular process could also have many thread arenas, so over here we have actually shown there are 2 thread arenas one in blue color and the other one in yellow, so each thread arena is allocated to a particular thread in that process, so the thread arena is also known as the dynamic arena.

So as we have seen before each thread arena can contain multiple heaps, so these heaps are linked together by linked list. So in each of these thread arenas we again have the struck malloc underscore state which is the management block which contains the meta data for this particular arena corresponding to each heap that is present in this arena we have a struck heap underscore info which contains the meta data for that corresponding heap, so this particular example corresponding to the heap present in blue we have 3 heaps that are present therefore we have 3 heap info structures this is the 1st one, this is the 2nd one and this is the 3rd one.

All of these heaps are linked together in a link list and the head of the link list is this particular area. Now all these various arena that are present such as this arena this blue arena and this yellow arena are further linked together by a link list, so what you can conclude from all of this that the entire heap segment of a particular process, it is not necessarily one contiguoustic segment. On the other hand the heap segment in fact comprises of various smaller segments which are connected by link list. These smaller segments comprises of arenas and each arena comprises of heaps and all of these are then linked together by multiple link list, now the head of all of these list is the main arena.

(Refer Slide Time: 28:09)



More about Arenas

- Maximum number of Arenas restricted by the number of cores in the system:
  - 32 bit: #MaxArenas = 2 x Num.ofCores
  - 64 bit: #MaxArenas = 8 x Num.ofCores
  - If num. of threads is less than #MaxArenas, then we get quick mallocs and frees as there is no contention
- One arena can service one memory request at a time (i.e. one malloc / free)
- If more threads are present than MaxArenas then multiple threads need to share one arena.
  - This leads to contention and hence slower mallocs and frees
  - Structure ***malloc_state***, contains all the management information for an arena

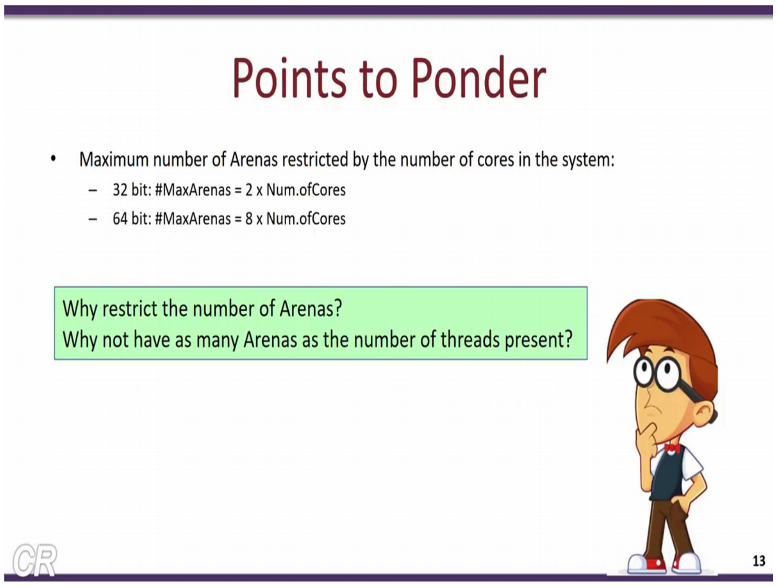
CR 12

Some more information about arenas in the heap there is a maximum number of arenas that can be present in a particular process typically in a 32-bit system the maximum number of arenas present is 2 times the number of cores present in that system, so these cores are the processor cores present in that system. In 64 bit system maximum number of arenas that are present in a particular process is 8 times the number of course, so for example if I am running a 32-bit program on a 32-bit system and the number of cores present is 4 it would imply that at most I could have 8 different arenas.

So what this means is that if I (())(29:02) 7 threads in this program then each thread including the main thread of the program would get a different arena. Now if I increase the number of threads in that particular program then it would mean that you would have multiple threads sharing the same arena, now therefore best efficiency and fastest malloc and frees are obtained when the number of threads are restrict to7 plus the main thread so therefore 8. Now once you increase the number of threads beyond 8 since there is a sharing of arenas it could result in a slight slowdown of the malloc and frees.

This slowdown is caused due to the contention for using a particular arena when a single arena is shared by 2 or more threads the ptmalloc code ensures that there is synchronisation between the threads in order to use the particular arena, so a various locking and unlocking mechanisms are implemented in ptmalloc2 to ensure that the state of the particular arena is always consistent, so it ensures that when one thread is trying to use a malloc and that malloc falls in a arena which is also shared by other thread then there is a locking mechanism to ensure synchronisation.

(Refer Slide Time: 30:39)


Points to Ponder

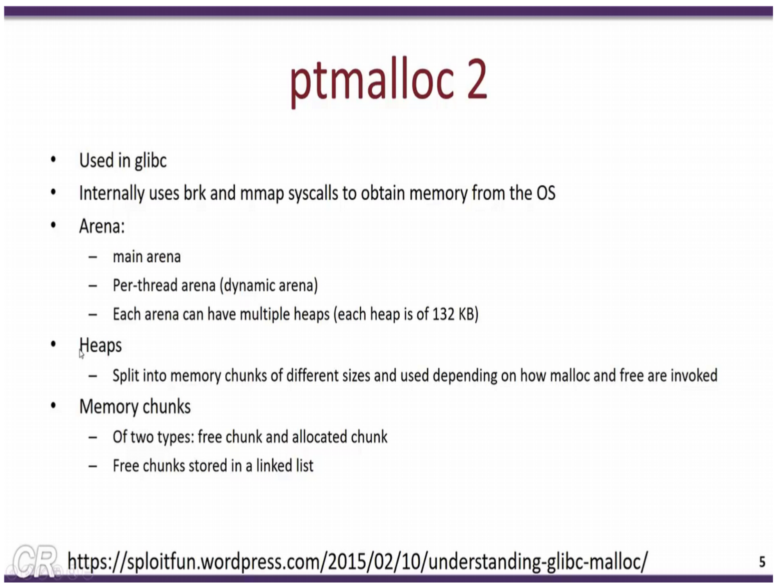
- Maximum number of Arenas restricted by the number of cores in the system:
  - 32 bit: #MaxArenas = 2 x Num.ofCores
  - 64 bit: #MaxArenas = 8 x Num.ofCores

Why restrict the number of Arenas?
Why not have as many Arenas as the number of threads present?

13

So something to think about we mentioned in the previous slide that each process has a maximum number of arenas that are present, now the question over here to think about is why is there such a restriction in the number of arenas? What would happen if we have unlimited number of arenas that is in other words what would happen if each thread that you create gets its own arena.

(Refer Slide Time: 31:09)

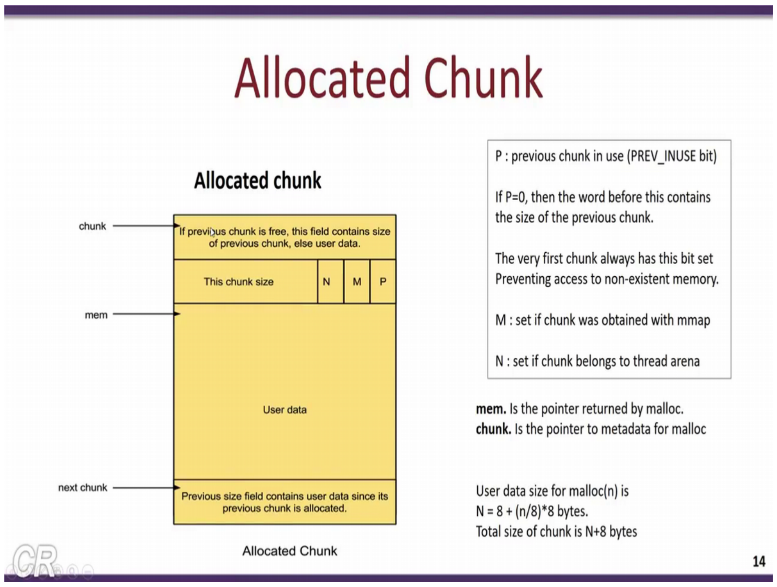

ptmalloc 2

- Used in glibc
- Internally uses brk and mmap syscalls to obtain memory from the OS
- Arena:
  - main arena
  - Per-thread arena (dynamic arena)
  - Each arena can have multiple heaps (each heap is of 132 KB)
- Heaps
  - Split into memory chunks of different sizes and used depending on how malloc and free are invoked
- Memory chunks
  - Of two types: free chunk and allocated chunk
  - Free chunks stored in a linked list

https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/ 5

Next we will talk about heaps which are present in an arena the memory with in a heap is split into memory chunks of different sizes and these chunks are actually used when we invoke malloc and free, so the chunks are of 2 types one is known as allocated chunks and the other one is known as the free chunks. Now the free chunks is actually stored in a linked list, now every time we do a malloc let us say we do a malloc of 1000 bytes what would happen is that the ptmalloc would execute it would find the corresponding arena and then it would determine the corresponding heap and then within the heap it would find the memory chunk that could satisfy the 1000 bytes request. When such a chunk is found then it would allocate that chunk to your process, so we will now look at how these allocated and free chunks are actually organized.

(Refer Slide Time: 32:16)



Allocated Chunk

Allocated chunk

chunk

If previous chunk is free, this field contains size of previous chunk, else user data.

This chunk size | N | M | P

mem

User data

next chunk

Previous size field contains user data since its previous chunk is allocated.

Allocated Chunk

P : previous chunk in use (PREV_INUSE bit)

If P=0, then the word before this contains the size of the previous chunk.

The very first chunk always has this bit set Preventing access to non-existent memory.

M : set if chunk was obtained with mmap

N : set if chunk belongs to thread arena

mem. Is the pointer returned by malloc.
chunk. Is the pointer to metadata for malloc

User data size for malloc(n) is
N = 8 + (n/8)*8 bytes.
Total size of chunk is N+8 bytes

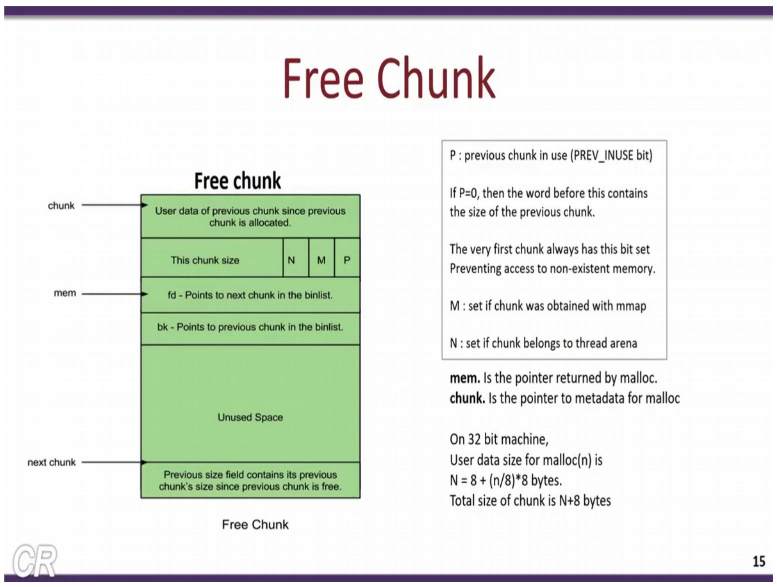
14

So an allocated chunk looks something like this when new malloc say for example 1000 bytes and this chunk gets allocated the point that gets return to the particular program is a pointer to this location over here. Now prior to this location there are some information known as meta data, this meta data could be either of 8 or 16 bytes depending on the system that you are running, so this meta data comprises of information about this allocated chunk. It has the chunk size which is present over here and then it has 3 flags N, M and P flag.

The P flag is used to determine whether the previous chunk over here which ends at this particular point is we used or an used, now if P equals to 0 then the contents of this would be the size of the previous chunk besides this we have 2 other flags the N flag and the M flag, so these 2 flags are not very important for us. The M flag set if the chunk was obtained using an mmap system call by the N flag is set if the chunk along to a thread arena.

From memory allocation point of view when you allocate 8 bytes it would mean that the 8 bytes starts from here and end over here and besides this there would be some more meta data bytes that gets allocated. So in a typical 32-bit system you would have 8 more bytes that gets allocated, so in total for a malloc of 8 bytes internally ptmalloc would actually allocate 16 bytes, 8 for the actual data for the user data and 8 more bytes for the meta data.

(Refer Slide Time: 34:10)



Free Chunk

Free chunk
chunk
User data of previous chunk since previous chunk is allocated.
This chunk size
N
M
P
mem
fd - Points to next chunk in the binlist.
bk - Points to previous chunk in the binlist.
Unused Space
next chunk
Previous size field contains its previous chunk's size since previous chunk is free.
Free Chunk

P : previous chunk in use (PREV_INUSE bit)

If P=0, then the word before this contains the size of the previous chunk.

The very first chunk always has this bit set Preventing access to non-existent memory.

M : set if chunk was obtained with mmap

N : set if chunk belongs to thread arena

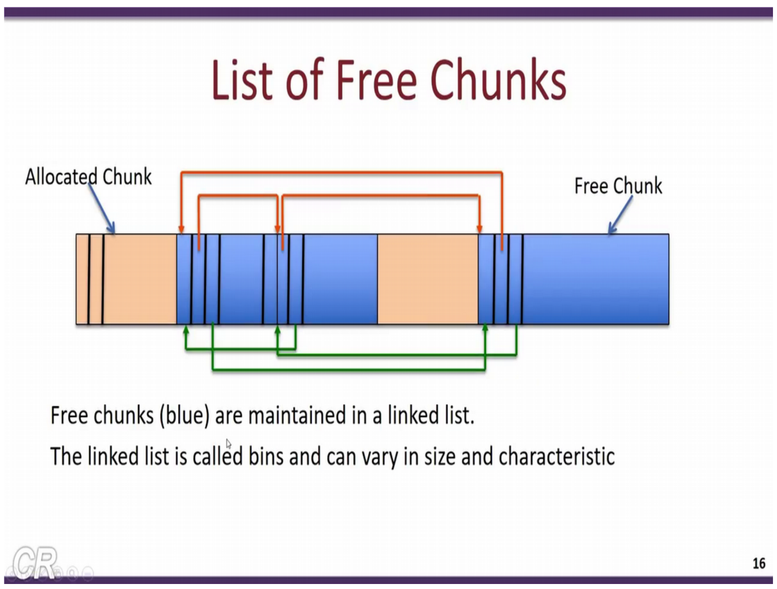
mem. Is the pointer returned by malloc.
chunk. Is the pointer to metadata for malloc

On 32 bit machine,
User data size for malloc(n) is
N = 8 + (n/8)*8 bytes.
Total size of chunk is N+8 bytes

15

Now when you free a chunk of memory using the call free and giving the address then malloc would actually convert that allocated chunk to a free chunk. Now the free chunk has a structure which looks like this, so what is important in this structure are these 2 entries the forward entry and the back entry, so essentially what free is doing is that it could add this particular free chunk into a link list, so this link list could be either a single link list or a double link list and this forward and back pointers are used to point to the previous and the next free chunk present in the heap.

(Refer Slide Time: 34:55)



List of Free Chunks

Free chunks (blue) are maintained in a linked list.

The linked list is called bins and can vary in size and characteristic

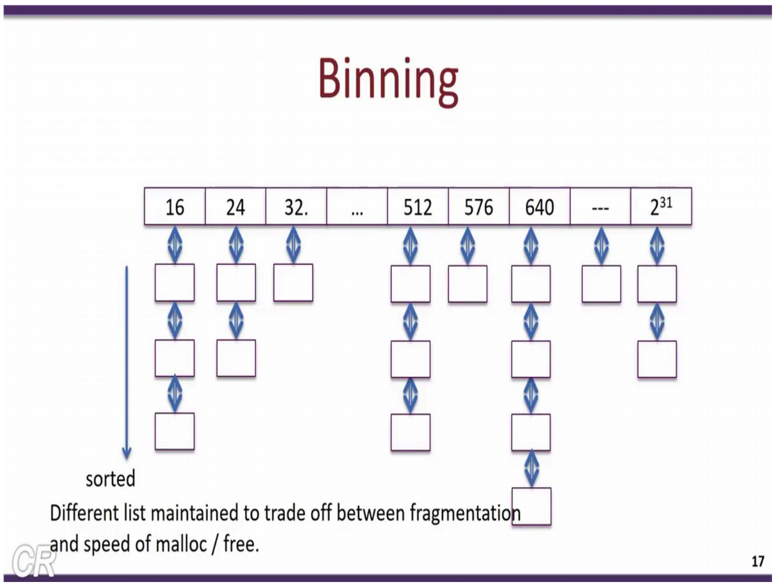
16

So this is how it will actually look like let us say this entire thing is the heap and within the heap you have various chunks the orange color shows the allocated chunks and the blue color shows the free chunks. Now the lines over here like this this and these lines are separation for the various meta data that are present, so important for us are the link list which are used to store the free chunks, so over here we are shown a w link list which actually is linked to all the free chunks that are present in this corresponding heap memory.

Now whenever there is a malloc that gets requested what happens first is that the ptmalloc code would pass through this particular link list and determining whether there is a free chunk of memory that it can satisfy the request for malloc. If such chunk is found then that particular data is removed from this link list and it is allocated for that memory, so thus you see whenever there is a malloc that is done it is likely that one chunk of free memory gets allocated on the other hand when a free occurs one of the allocated chunks gets freed and gets added on to the link list.

The problem with this approach is that the link list could be very large and therefore malloc would take a long time to find the appropriate chunk to be allocated, so for example let us say we have done a malloc of 1000 bytes then this link list has to be traversed until malloc finds one free chunk which is at least of 1000 bytes and this could take a long time and therefore what this actually done in ptmalloc is that we do not maintain just one link list but we maintain multiple different types of ink list.

(Refer Slide Time: 37:04)

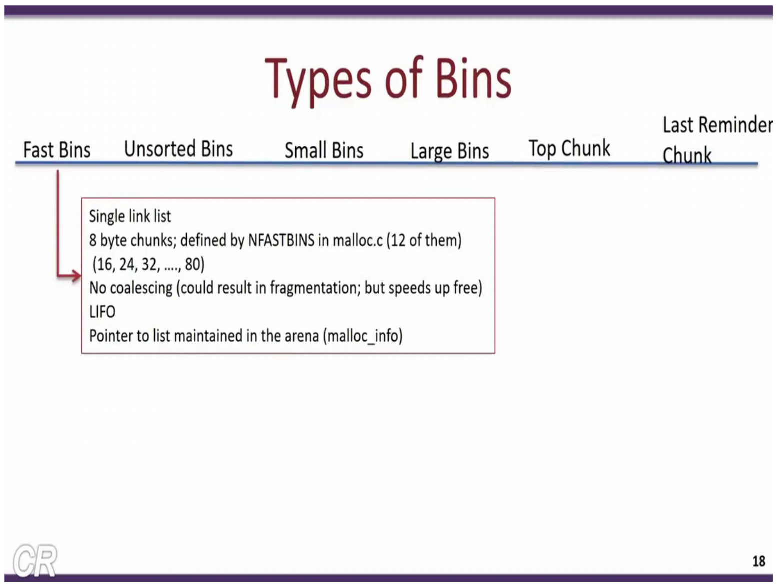


Binning

Different list maintained to trade off between fragmentation and speed of malloc / free.

So we call this as binning, so in the ptmalloc code in fact you have various different types of bins and each bin caters to different size of chunks that gets allocated, so for example here we have a linked list of chunks which are of 16 bytes on the other hand we have over here a link list of chunks comprising anything from 577 to 640 bytes, so now in this particular case when a malloc gets invoked it can directly go to that particular link list and select the appropriate free chunk and allocate that particular free chunk much more quickly.

(Refer Slide Time: 37:46)



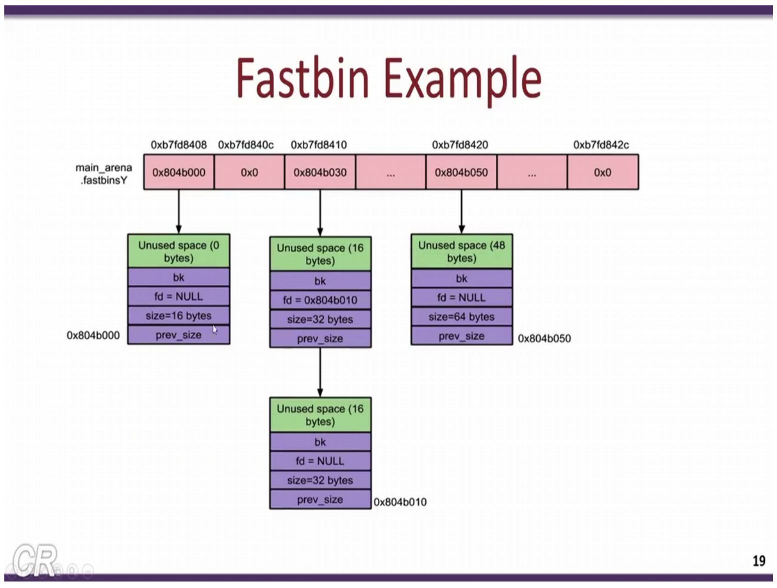
Types of Bins

Fast Bins Unsorted Bins Small Bins Large Bins Top Chunk Last Reminder Chunk

Single link list
8 byte chunks; defined by NFASTBINS in malloc.c (12 of them)
(16, 24, 32, ..., 80)
No coalescing (could result in fragmentation; but speeds up free)
LIFO
Pointer to list maintained in the arena (malloc_info)

In ptmalloc2 there are different types of bins that are used most notably are the fast bins, unsorted bins, small bins, large bins, so besides this we have something known as the top chunk and the last remainder chunk. Now each of these different bins are managed in a different way, so as to suite and best manage that particular memory. For example the fast bins are single link list they are not double link list, now each of these fast bins are in 8 byte chunks, so for example we have fast bins of 16 bytes, 24 bytes, 32 bytes and so on till 80 bytes and also in fast bins there is no coalescing. We will see what coalescing is at a later point, essentially coalescing is used to prevent fragmentation however if you bin happens to be the fast bin then there is no coalescing which is done. Now since it is a single link list so there is a last in 1st out kind of operation which goes on.

(Refer Slide Time: 38:59)

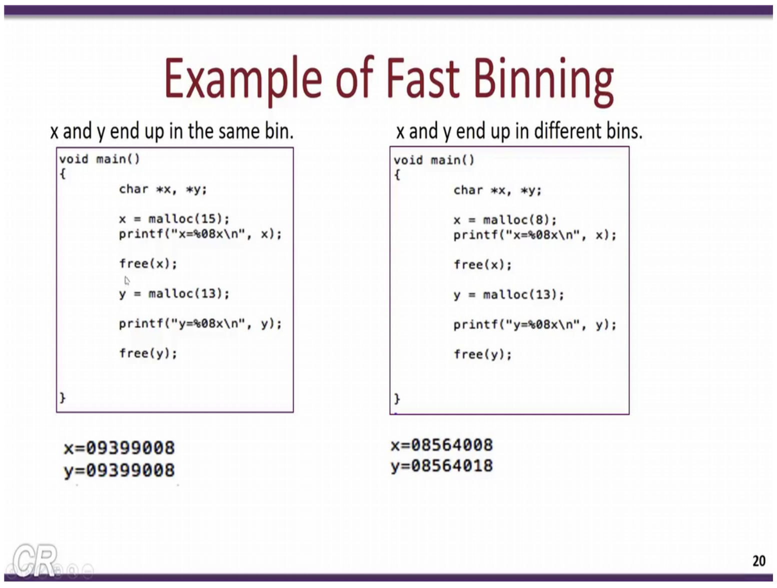


Fastbin Example

So this particular slide shows an example of fast bins, so in the main arena we would have an array like this and each array is a pointer to a link list for example this 1st element is a fast bin pointer and it is appointed to her link list of chunks which is of 16 bytes this entry is appointed to a link list of memory chunks of 32 bytes and so on, so whenever there is a malloc say of 32 bytes what PT malloc would do is that it would refer to this fast bin array it would immediately get that it is a 32 byte request it go into this location and it would pick up the 1st available chunk over here.

Now there is the linked list operation wherein this location is now modified so as to point to the next available free chunk of 32 bytes, so you see that if your memory allocation is very small and it happens to be in the fast bin then malloc works very quickly. All that is required is just to find the offset in the fast bin pick out the 1st memory chunk that is present and do a small link list operation.

(Refer Slide Time: 40:22)



Example of Fast Binning

x and y end up in the same bin.

```
void main()
{
        char *x, *y;

        x = malloc(15);
        printf("x=%08x\n", x);

        free(x);

        y = malloc(13);

        printf("y=%08x\n", y);

        free(y);

}
```

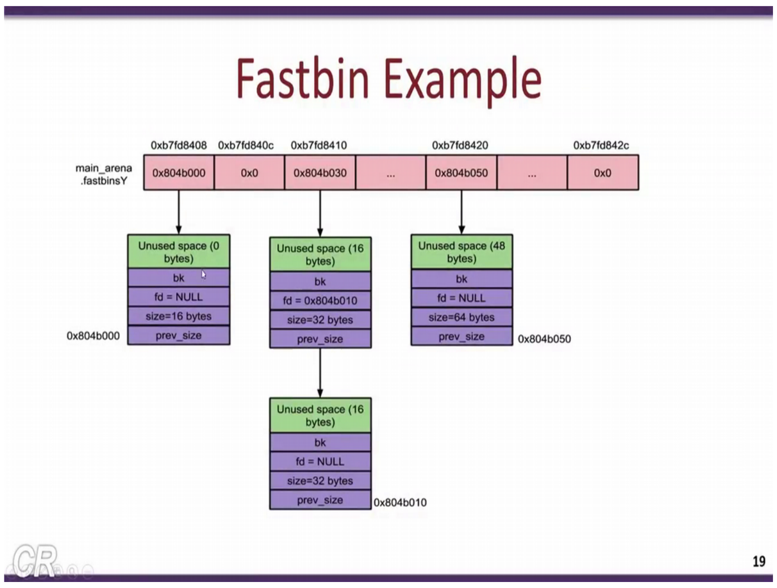
```
x=09399008
y=09399008
```

x and y end up in different bins.

```
void main()
{
        char *x, *y;

        x = malloc(8);
        printf("x=%08x\n", x);

        free(x);

        y = malloc(13);

        printf("y=%08x\n", y);

        free(y);

}
```

```
x=08564008
y=08564018
```

20

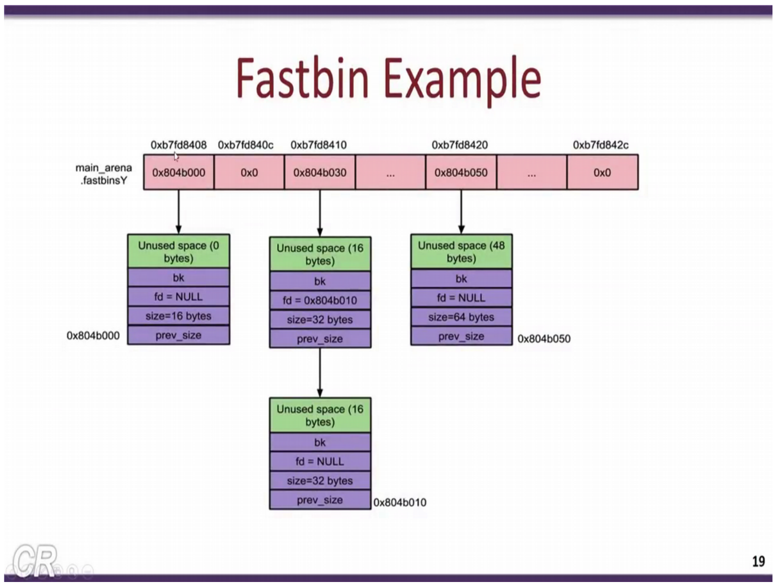


Fastbin Example

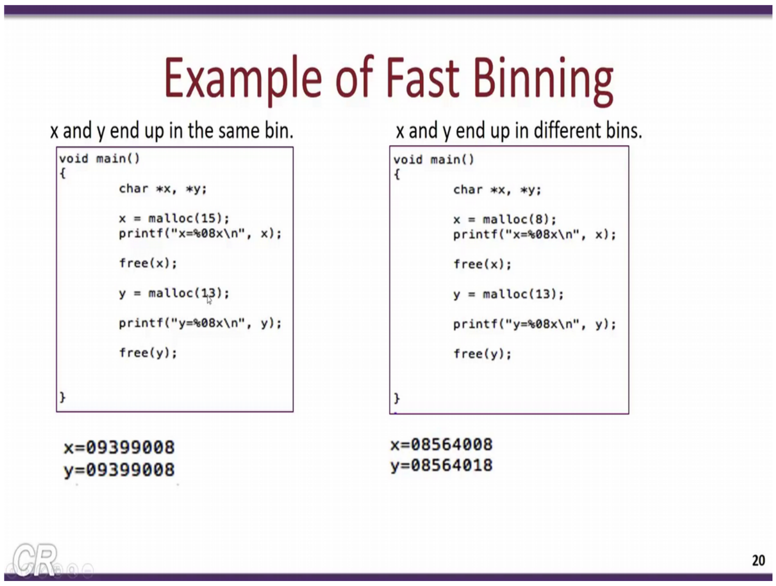
19

So in this example what we do is we malloc x which is of 15 bytes then print the value of x and then free x then v malloc y which is of 13 bytes print y and free y, so what happens when you execute this program is that both x as well as y, so there is a seeming function which gets applied and both of them would end up in the link list corresponding to this 16 bytes, so when this particular malloc gets executed totally a chunk of 16 bytes plus another 8 bytes gets allocated and the pointer to that memory is present in x, so when this free gets invoked the chunk gets added to the link list corresponding to the fast bin of 16 bytes.

(Refer Slide Time: 41:15)



Fastbin Example



Example of Fast Binning

x and y end up in the same bin.

```
void main()
{
        char *x, *y;

        x = malloc(15);
        printf("x=%08x\n", x);

        free(x);

        y = malloc(13);

        printf("y=%08x\n", y);

        free(y);

}
```

```
x=09399008
y=09399008
```

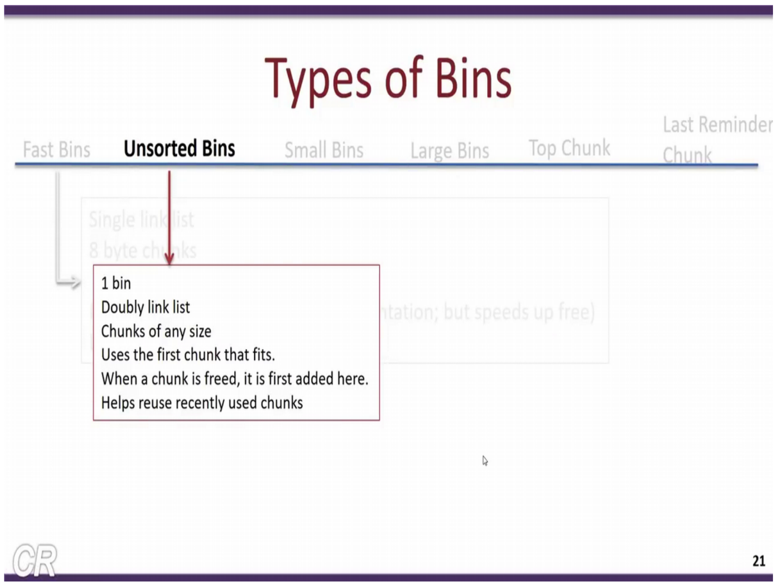
x and y end up in different bins.

```
void main()
{
        char *x, *y;

        x = malloc(8);
        printf("x=%08x\n", x);

        free(x);

        y = malloc(13);

        printf("y=%08x\n", y);

        free(y);

}
```

```
x=08564008
y=08564018
```

Therefore we would corresponding to the fast bin of 16 bytes we would have the chunk x that gets allocated. When we do a malloc of 13 bytes again after this free it would actually obtain the same list entry thus what would happen is that x and y would get the same address thus when we run this particular program we get x and y pointing to the same chunk of memory present in the process. However this is not true if the sizes of the malloc are different for example if you do a malloc 8 and a malloc 13 these happen to fall in different fast bin entries. Now on the other hand if the malloc sizes are different for example here we have 8 bytes and then 13 bytes then they end up in different fast bin thus over here x and y would get different addresses.

(Refer Slide Time: 42:16)



Types of Bins

Fast Bins Unsorted Bins Small Bins Large Bins Top Chunk Last Reminder Chunk

Single link list
8 byte chunks

1 bin
Doubly link list
Chunks of any size
Uses the first chunk that fits.
When a chunk is freed, it is first added here.
Helps reuse recently used chunks

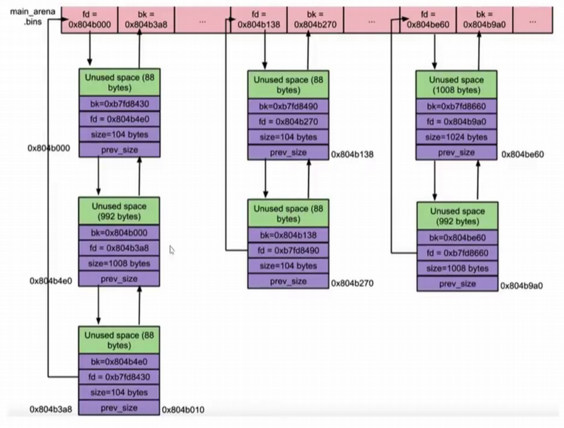
…ntation; but speeds up free)

21

Now besides this we have unsorted bins so this is one bin which is a double link list that it could have a chunks of any size the allocation policy used by ptmalloc is to use the first chunk that fits when a chunk is freed it gets first added here.

(Refer Slide Time: 42:34)

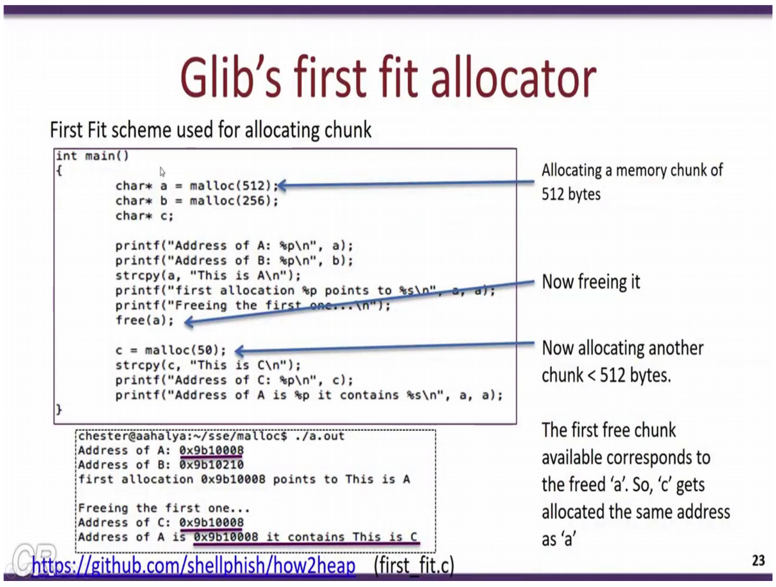
Unsorted Bin



22

So this is an example of an unsorted bins, so what we see over here is that there is a double link list so therefore there is a forward pointer and are back pointer. This is an array which is present in the main arena and the element is of type bins, so you also notice that each of these bins are of different size, so for example now if I do a malloc of 1000 bytes it would come over here it would find that 104 bytes is too small to satisfy the request it could come here and it could find out that 1008 bytes is exactly enough and therefore this particular chunk would get allocated to that request.

(Refer Slide Time: 43:17)



Glib's first fit allocator

First Fit scheme used for allocating chunk

```
int main()
{
        char* a = malloc(512);
        char* b = malloc(256);
        char* c;

        printf("Address of A: %p\n", a);
        printf("Address of B: %p\n", b);
        strcpy(a, "This is A\n");
        printf("first allocation %p points to %s\n", a, a);
        printf("Freeing the first one...\n");
        free(a);

        c = malloc(50);
        strcpy(c, "This is C\n");
        printf("Address of C: %p\n", c);
        printf("Address of A is %p it contains %s\n", a, a);
}
```

```
chester@aahalya:~/sse/malloc$ ./a.out
Address of A: 0x9b10008
Address of B: 0x9b10210
first allocation 0x9b10008 points to This is A

Freeing the first one...
Address of C: 0x9b10008
Address of A is 0x9b10008 it contains This is C
```

Allocating a memory chunk of 512 bytes

Now freeing it

Now allocating another chunk < 512 bytes.

The first free chunk available corresponds to the freed 'a'. So, 'c' gets allocated the same address as 'a'

https://github.com/shellphish/how2heap (first_fit.c)

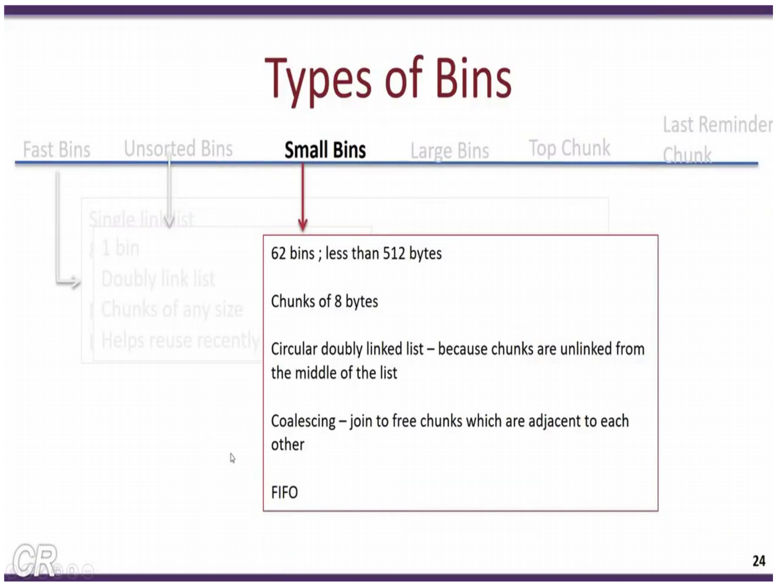
23

So let us see an example of the use of the first fit allocation which is present in ptmalloc, so what we do in this example is that we malloc 2 regions one of 512 bytes which is pointed to by a and the other of 256 bytes which is pointed to by b okay then what we do is we free a and we malloc c which is of 50 bytes note that c is much less than the 512 bytes and 256 bytes that has been allocated previously.

Now what happens when we free a is that a is too large to go into a fast bin and therefore it goes into this unsorted bin and therefore the unsorted bin would have an entry which corresponds to the chunk of approximately 512 bytes. Next what we do is we invoke malloc again with a request for 50 bytes and the pointer that malloc returns is stored in c. Now in the first fit allocation what would happen is the unsorted bin and traversed it could find that in this unsorted bin there is a chunk of 512 bytes that is present and therefore what it would do is it would split this chunk into 2 parts.

So one part is slightly more than 50 bytes and this part gets allocated to c. Now the other part which is not being used will still remain in the link list thus when we run this program this is what we would see, we would see that the initial allocation of a and b are at this addresses and when we actually allocate c after freeing a it could get exactly the same address that a has obtained, so you know that both address a and c are pointing to the same location that is 9b10008.

(Refer Slide Time: 45:19)



Types of Bins

Fast Bins | Unsorted Bins | **Small Bins** | Large Bins | Top Chunk | Last Reminder Chunk

62 bins ; less than 512 bytes

Chunks of 8 bytes

Circular doubly linked list – because chunks are unlinked from the middle of the list

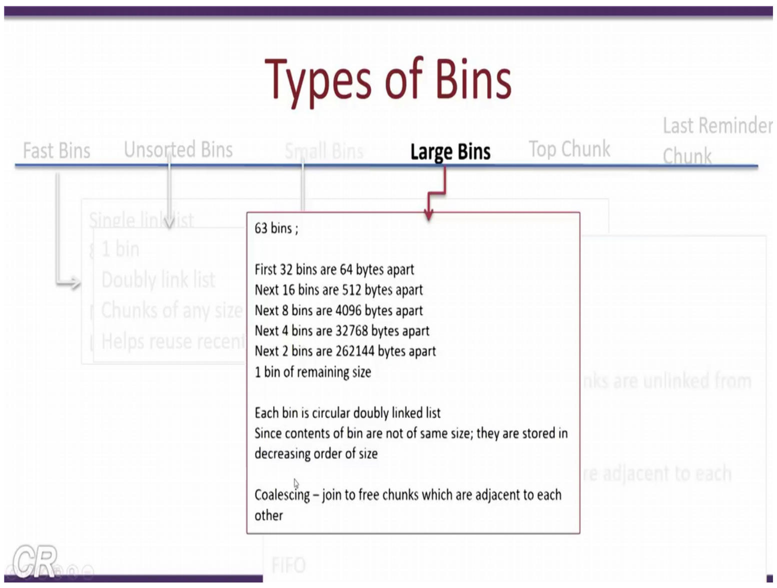
Coalescing – join to free chunks which are adjacent to each other

FIFO

24

Now the other bins that are present are the small bins there are 62 small bins which are less than 512 bytes and there are chunks of 8 bytes presented in them. These bins are also circular and doubly linked list and satisfy coalescing okay and it has 1st in 1st out.

(Refer Slide Time: 45:39)



Types of Bins

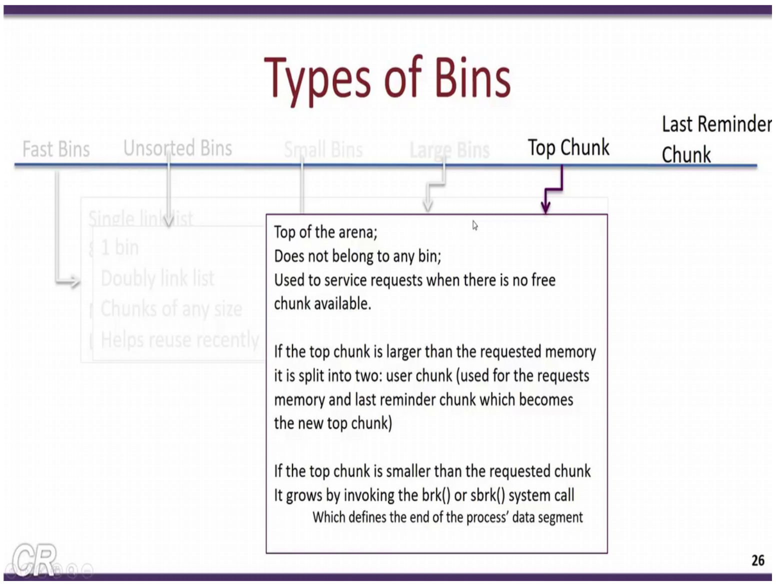
Fast Bins | Unsorted Bins | Small Bins | **Large Bins** | Top Chunk | Last Reminder Chunk

63 bins ;

First 32 bins are 64 bytes apart
Next 16 bins are 512 bytes apart
Next 8 bins are 4096 bytes apart
Next 4 bins are 32768 bytes apart
Next 2 bins are 262144 bytes apart
1 bin of remaining size

Each bin is circular doubly linked list
Since contents of bin are not of same size; they are stored in decreasing order of size

Coalescing – join to free chunks which are adjacent to each other



Types of Bins

Fast Bins
Unsorted Bins
Small Bins
Large Bins
Top Chunk
Last Reminder Chunk
Single link list
1 bin
Doubly link list
Chunks of any size
Helps reuse recently
26

Top of the arena;
Does not belong to any bin;
Used to service requests when there is no free chunk available.

If the top chunk is larger than the requested memory it is split into two: user chunk (used for the requests memory and last reminder chunk which becomes the new top chunk)

If the top chunk is smaller than the requested chunk
It grows by invoking the brk() or sbrk() system call
Which defines the end of the process' data segment

So there are also 63 large bins of various sizes. The top chunk is at the top of the arena and does not belong to any bin, so it is used to service requests when there is no free chunks available, so whenever you do a malloc what would happen is that the malloc would traversed through all the various link list and if it finds that there are no chance which are available in any of these link which can satisfy that particular malloc request then the part of the top chunk is taken and that part is allocated for the malloc request.

Now if the malloc request was large enough so that even the top chunk does not have sufficient memory to satisfy that particular malloc request then the OS gets invoked and I knew heap segment gets allocated, so this is done using the brk in maps or the sbrk system calls. So in this way we have seen how malloc is managed internally with various arenas, heaps, chunks and various types of bin. In the next lecture we will look at more into detail about the free function call essentially how free allocates and the allocates memory and how the link list are managed and in particular we will actually look at the ways to exploit this internal aspects of malloc and how to actually create and attack on this screen. Thank you.