**Processes**

is being recorded uh hello everyone uh we'll start our discussion now based on the third chapter that is on processes we have already covered a lot of topics from the first two chapters so i'll advise you to now start reading the chapter one and two from the textbook so let us revise some concepts that we have already learnt okay during the discussions that we had earlier we have said that process is a program in execution so when we say process is executing it is going to consume cpu so the instructions of that particular process they are going to be running on the cpu because the process is executing and in order to execute instructions on the cpu the instructions have to reside in the ram so obviously the process will occupy certain space in the ram the space will be occupied in the ram not only for code but the global variables that is the data of the process the local variables and parameters that is going to be occupied on the stack of the process there will be a heap for the process some memory will also be occupied by the shared libraries and we'll see the concept of shared leather is in detail when we see memory management what we have already seen also is that a process is most typically created using fork and exec we have seen what 4 does is 4 creates a duplicate process which is identical with the caller process and that is a fork which is a different type of a system call will return twice it will return once in parent and second time in the child because after the call to fork is over there are going to be two identical processes in execution and the four has to return in both of them so it returns the what is called as the process id of the child to the parent and it returns 0 to the child we have also seen how to write different code for the parent and for the child using the return value of 4. then we also discuss what exec does and we saw that exec will superimpose the specified executable on the currently running process so basically after exec is over the original process which called exec will vanish and a new process will get instituted in its place when i say a new process gets instituted i am not saying that the process id is going to change but basically the whole character of the process will change the code itself will change the stack will change the global data area will change and so on exec doesn't return if it succeeds because after exec is over the calling process has converted itself into the caller into another program the program which was specified as an argument to exec then we have also seen the concept of multi-programming and multitasking we said multi-programming is a system where multiple programs can reside in the computer memory at the same time while multitasking is necessarily multi-programming but it also involves scheduling of the processes one after another in a way that they share the cpu time now we have also discussed that multitasking is made possible with the help of a timer interrupt so as we know the operating system will set a particular value in the timer register and pass on the control to a process so the process will now start running and the process will execute on the cpu it will execute as long as the timer resistor is not decremented to zero with every cpu cycle the timer resistor will keep decrementing by 1 and when it becomes 0 the timer interrupt will be raised and as a result of timer interrupt which is the hardware interrupt we have seen that the control will jump to a predefined location at that predefined location the operating system has already copied its code and that code is called a scheduler in the case of a timer interrupt so scheduler which is a part of the operating system will be invoked on every timer interrupt the job of the scheduler in the operating system is to select the next process to execute and pass control over to that process obviously it will set the new value of the timer interrupt and then pass the control over to the next process we also saw inter process communication using pipe this was done during a lecture a demonstration lecture in the laboratory session so what we saw is that a pipe system call will create an operating system data structure which will act as a fifo or a queue which will have two ends a right end and a read end and both these ends are actually made made available to us as file descriptors so a process which calls pipe will basically get in return two numbers those two numbers are two file descriptors the zero eight one is for reading and the first one is for writing so now you can do the same system calls that you do on a normal file like the read and write and lc can close and all on the file descriptor what happens is if you create a pipe and then call a fork then the pipes buffer gets shared between the parent and child and the file descriptors get inherited so both the parent and child will have the same file descriptors but the particular file that is the pi file to which the point is shared so that is why it is possible now for one to write into the pipe and another to read from it so basically the pipe now acts as a inter process communication medium between the child process and the parent process and then we also saw demonstration of how the shell can connect two processes using a pipe we saw a demonstration of this as well then we have so far also covered the concept of a calling convention and calling convention is a convention which is documented for each processor the convention is needed to make the function calls happen because function calls are supposed to happen in lifo fashion and recursion is also supposed to work so the convention is basically a set of rules to ensure that the parameters are passed correctly correctly correctly it should have been and the return value is written correctly and this is often done using the stack and or either way the resistors so the parameters and written value get handled between two function calls using the stack and the resistors so basically the calling convention is a set of rules for the compiler to generate additional code in the caller function and in the called function then we also saw the concept of system programs that is the c compiler or in general a compiler which converts a language code to a machine code most typically then we saw the concept of a linker which will basically combine different object code files together machine code files together and connect the call of a function to the code of the function then we saw the concept of static linking or there is a mistake here it should have been a static linking and dynamic linking static clinking is one where the machine code from different object files is combined together in the final object code file so the size of the file is very big in dynamic linking instead of directly combining the machine code together a stub code a placeholder code is inserted which basically gives the location of the object code file that contains the the object code so obviously dynamic linking in this particular way will require the operating system to do um to do a more specific job now because when the particular program is to be executed then the particular machine code file will have to be dynamically picked up okay from that location and that is obviously linked to the concept of a loader which is basically the code of exec essentially inside the os and the job of this code is to load the executable file from the disk or whichever storage medium you have pick it up from there from there and load it into the operating system's memory properly and then we saw there can also be a static loading and dynamic loading in static loading the operating system will pick up all the object code and put it in memory in dynamic loading the operating system loader will basically pick up the needed machine code only when it is needed okay so when the program starts running all the code will not be in memory okay only when there is a need to run a particular piece of code it will be brought into memory that way it saves lot of memory in dynamic loading because you don't load the complete program entirely all right so these were some things we have already discussed i'll wait for questions any if you have any questions on whatever we have covered so far before we start understanding processes in more detail i'll wait for few more seconds so chaitas you can ask the question publicly you have asked the question privately the question asked is that there cannot be static linking and dynamic loading so why not check this so static linking will basically combine all the object code together and put it up put it in a single let us say executable file so the file will be very big now even then it is possible for the loader to let us say just pick up the the code of the main function put it in memory make it run when main calls let us say a particular function and let us say through some mechanism the os is able to detect that this particular code is not loaded in memory it can again go to the file and load the code of the function that is called by main and then pass on control over to it so it is possible okay i'll wait for half a minute more to check if there are more questions [Music] so rish has a comment that static loading dynamic linking is not possible so first comes dynamic linking isn't it because linking happens before loading so if there is a dynamic linking and we have seen you know the object code of a dynamically linked file uh we did a obj dump on a dynamically linked file object code file and saw the code and we saw there is a stub code so for example in the call to printf we saw it at the call of the printf is not the code of the printer but a stub code [Music] so if you do dynamic linking of that type then one thing is for sure that you now need a loader which understands dynamic linking so the loader should be able to locate the code of printf when the program is running okay so in that sense a if you think of a static loader now the static loader will have to look at all the function calls then locate the stub code in the execute in the object code file in the place of the stub code it will have to substitute the actual code okay and then you have the complete code in memory you know to do static loading i will say that okay this is like too much unnecessary work if you have dynamic linking because the whole purpose of dynamic linking was to ensure that the object code file will be picked up dynamically and that you know in a indirect sense is that pick it up only if it is needed so yes dynamic linking in itself is suggestive of dynamic loading so if you have a dynamic linking and static loading it makes of a very impractical and meaningless scheme in my opinion although it is practically possible although it is practically possible to do dynamic linking and static loading it doesn't make sense to do it because the the very fact that you did dynamic linking means you said that this particular object code file should be picked up dynamically this particular object code should be picked up from some other location so why it should be picked up at the time of execution of the program it can be picked up when the code needs to be executed so dynamic linking normally goes with dynamic loading it doesn't normally go with static loading any other questions thank you okay let's go ahead right so this is the diagram okay which is taken from your textbook and very often the memory layout of a c program is described like this now when we say the memory layout what kind of memory loud are we talking about we are talking about the memory layout that is most typically assumed by the compiler that is most typically assumed by the compiler because it is the job of the compiler to generate machine code and while generating machine code the compiler needs to assume certain things about the addresses that particular variables or the code will have if the compiler cannot have any idea about the location of variables in memory how is it going to generate the code so essentially it is a part of again agreed convention that the operating system will support a certain kind of memory layout for a program and that is why the compiler can assume that kind of a memory layout for a program so the memory layout goes something like this it is assumed that starting at lower addresses is the code and the code area is normally called text so the word text refers to the code of the program the machine code the instructions of the program now most typically a c program will have global variables but the global variables will be of two types uninitialized global variables like the in text here and initialize global variables like the into y here the initialize and the uninitialized variables they often go into two different regions in the memory layout of the c program now the question is why is it so it is so because what happens is when this code is compiled into an executable file obviously you know code like this will get compiled into machine instructions which will become part of the the code the machine code that it generated same is the case with code like this and so on we know obviously that variables like these and variables like these they are going to be occupied on the stack and the stack will actually get occupied only when the program is going to execute isn't it similarly we know that memory like this the malloc memory which is going to be allocated will happen only when the program is running but variables like this into y equal to 15 they are global variables and they need to be available throughout the life of the program right from when main starts executing till main returns that is why these variables they have to come into existence before mean starts executing i'll repeat the global variables they have to come into existence in the memory before main starts executing and that is obviously they cannot go on the stack and they have to be located in a different memory region now the variable y is assigned the value 15. now most typically one will say that this can be done using a memory instruction using instruction like mu 15 to some memory address but now the whole trouble is that in order to execute the instruction the program has to run in order to execute the instruction the program has to run all the instructions are part of only the code of the functions and main is the first function to run so how do you even initialize the global variable becomes a cyclic problem the simple solution to that is that in the executable file itself a particular area is created in which the variable value is written out so global variables which are initialized actually occupy a space in the executable file but the variables which are not initialized like x here they don't have to occupy space in the executable file all you need to mention is that you know space of 4 bytes is needed for x now imagine if there were 10 such variables x 1 x 2 x 3 x 10 then all you want to say in the executable file is that 40 bytes of memory is needed to to hold 10 such variables which are uninitialized but if you are let us say 10 variables global variables like y 1 y 2 y 3 up to y 10 then you can't only say that they need to occupy 40 bytes when the program starts running but you also need to mention the values that they need to hold when the program starts running that is why the uninitialized data occupies a different space in memory also and the initialized data occupies a different space in memory so what happens at the time of loading the program the initialized data is copied as it is from the executable file for the uninitialized data only some memory needs to be allocated without actually copying any content from the executable file so i hope this work connection between what the compiler writes into the executable file for the global data that is initialized and uninitialized what happens at the time of loading of the program and that is why in the memory there is a different section for initialized data and uninitialized data i hope this connection is clear so variables like x will basically occupy space in memory and in the uninitialized data section variables like y will occupy space in the initialized data section in the memory when the program is in execution obviously variables like values and i which are local variables and variables like rxc argue and for that matter all such local variables and parameters will occupy space on the stack so a particular region in the memory is the stack and you can see rcrv are there on the stack and the stack is growing downwards likely it has been now a long-standing convention that the heap will actually start you know where the initialized data ends and it will actually grow upwards so you can see that this memory space is actually logically shared between the stack and heap the heap grows upwards and the stack grows downwards so technically speaking there is even a possibility that the stack and hip start overlapping if if your code does something nasty so this possibility does exist so once again why is this memory layout used this memory layout is used by the compiler to generate the machine code compiler will assume that the code that is text is starting at a particular address the code because it is generating the code it will know that it will end at a particular address compiler will assume that the initialized data is at a particular address it will assume that the uninitialized data is at a particular address and that is why it can generate machine code for accessing those variables similarly compiler will assume that the stack is given by the esp and just generate code for variables local variables and parameters relative to the stack and so on there is a command called size if you run the size command on any object code file it will give you the sizes of the different sections that they are likely to occupy in the in the memory layout of the program so for example this is the output for the slash bin ls5 now so just to repeat what i said the compiler will assume that the program will be located like this in the ram when the program starts executed that is after exec is done and that is why the compiler is able to generate machine code assuming certain addresses for variables and code in the stack and heap and data and code areas any questions so the question is can i again explain why initialize and uninitialize data is in separate regions okay i'll do it again so see look at this particular code when compiler is done with this particular code okay what will happen machine code will be generated for you know loops like this and for returning value and so on but to do that the compiler needs to know the addresses of the variables now what does it know about address of variable like i it knows that it will be on the stack where on the stack at a particular offset from the stack pointer at which offset i can know that because i am going to calculate the offset i am the compiler i am going to actually calculate the offset for i n values uh what about x and y okay i only as a compiler know that there are global variables but they have to be located at a particular location in memory when the code is running so what do i do i assume a particular address for x and y and i can generate the code accordingly but now there is a problem here the problem is that y is initialized to the value 15 i am the compiler and i have to generate code for y equal to 15. the whole trouble is that i cannot generate a machine code that is an instruction for y equal to 15. why cannot i generate a machine instruction for setting y to 15 because y is a global variable it is not a part of any function initializing y to 15 is not a part of any function code here it is initialized as a global variable it is semantically a different thing if one was equal to 15 return inside main it is not written inside a main it is written outside main okay so what the programmer intends to say is that y should be initialized to 15 even before main starts executing now see the world trouble is the program is going to start in main and i want y to be initialized to 15 even before the program starts executing so how should that be done so the solution to that is that in the executable file itself the value of y will be stored in a different area as if it was 15 and at the time of loading the program the loader will load the value of 15 okay for the look in the location of y so that will be done by the compiler and the loader together and then your main will start running that is why in the executable file also there is a different section for initialized variables okay and the variable and the the executable file actually contains the value 15. now there are can be two types of global variable the initialize variables and the uninitialized variables initialize variables will contain the values in the executable file and those values will be picked up by the loader and put into memory as it is what about uninitialized variables now unusualized variables we say content random values now should they occupy space in the executable file there is no need absolutely no need because all you need to mention is that if you are let us say 100 uninitialized integers then the 100 initialize android initialized integers will occupy 400 bytes so what you need to mention in the executable file is loader please allocate 400 bytes when the program is to be loaded so the loader will look at that particular information and allocate 400 bytes but because of the very reason that they are to be handled by the loader separately the uninitialized data will occupy a different section in memory and initialize data will occupy a different section in memory does it answer your question rishikesh is the space for initialized and uninitialized data areas fixed do they grow no they don't they don't because if you look at the semantics of a c program then you have to declare all the variables the moment you declare all the variables and even if you declare your array you have to declare the size of the array the compiler can calculate the space that is needed and that is why they are fixed they do not grow it is the stack which will grow it is the heap which will grow but the data that is the initialized area is called the data area and the uninitialized area is called the bss area for whatever historical reason so the data section and the bss section they don't change that is the answer to your question september is saying global variables are initialized to zero automatically if you don't give value so the thing is why should you store those values in the ela file then you don't have to store it in the elf file at all all i need to mention is okay this much memory is needed for the for the uninitialized global variables there is no ar bracket n in c there is no arrow of a variable size in ncc so otherwise your question is unsubstantiated there is no array of a variable size in c arrays have to be of a fixed size in c if we are saying that okay i'm using a non-standard c and i'm using ari bracket and that is a array of a variable size then i'll say yes it has to go on the heap it cannot go on the stack [Music] more questions i'm waiting so i have a homework for you and others try to write a code like that and try to compile it into an object code file do obj dump observe the machine code and try to verify what i said if you don't see what i said then you have to be able to explain what you see in the object code please do this okay don't trust my answers i want all of you to actually you know run lot of obj dumps and see the machine code to actually experience i know the things which we say in theory in practice okay there is a question what is happening before main starts running in memory when program starts running so the loader is at at work because exec has been called when exec is called essentially what happens is that the elf file is read elf is the executable file from it the code is read code is put into memory and that becomes the code section then the data region from the ela file is read that is initialized global variables they are copied into memory then a space is allocated in memory for uninitialized global variables and then the space is allocated for heap and stack and then the control is given to the main so does it answer your questions option okay good hello yes sir so if i use malloc then uh the value is stored in the heap right right yes the the variables are allocated on heap okay so if my size is too big so won't it won't the hip uh overlap the stack because hip is going into it it will yes so does it cause any problems uh it can it can so it can so how does the the program decide how much memory should be allocated just you know some heuristic some heuristic which can go wrong so for example see in practice you will very rarely see this happening okay wow and all okay we will see more when we study memory management because the most typical way today because there is something called as virtual memory and dynamic loading and page faults and so on and the memory model that compiler actually assumes is as if you know the program had all 4 gb ram available so if you have 4gb of ram available then you have a huge amount of stack and heap available to you so how is that magic done that we are not going to see right now but in in theory yes it is possible that the stack and heap start overlapping then there is no guarantee your program will work but this is all you know these are you know assuming that your program does not have unrealistic expectations of memory isn't it this will happen only when your program has unrealistic memory requirements but it is possible that you have some such programs and the stack and heaps actually overlap so sir i'm saying uh does the heap grow downwards so see when you call malloc where is malloc malloc is in a c library so who is managing the heap the c library is managing the epo program is not managing it isn't it so whether to decrease the heap on a free in the choice of the c library it may not it will just say i will keep that much hip it may saying that okay i'll make space for the stack so that is the implementation choice of the one who is writing the c library satyam what is your question i don't get when men calls another function to store local variables of another function is a new stack created below why there is only one stack why do you need another stack there is only one stack it just keeps going downward there is no stack for a function there is one stack for all functions septem i have not understood your question at all if you are confused that there is a different stack for different functions then you are totally mistaken in the concept of the stack it is because there is only one stack for all the function calls in a program that the function calls can work because the function calls are part of one program execution so they have to be on the same stack otherwise how will you connect function calls to each other so there has to be only one stack for all the functions what happens is that a new stack there is a new activation frame allocated with every function call and we have seen the concept of activation frame with the ebp and esp and they're pointing to a particular region and you know they're changing their locations with every function call so just let me know subterm if you are satisfied or not so rohit has a question what will happen if you declare a global integer pointer and then later may lock it so the global integer pointer because the pointer is let us say a 4 byte variable the memory for the pointer itself will be in the data region but the malloc memory will be from the heap region all that will happen is that the four byte pointer in the data region will basically store the address of the memory in the maloc region that's what will happen remember you don't malloc a pointer you malloc a memory area and make the pointer point to it so very often you know the wrong phrasing of english sentences creates a lot of confusion you don't mellock a pointer the pointer exists when you declare a pointer the pointer exists you malloc a region of memory and make the pointer point to it that's what you do varian has a question the pointer will be in uninitialized data region yes if you have not initialized the pointer to null then it will be in the uninitialized data region september the question how is it decided how much side is to be given to the stack to run the program i will say it is a it is a matter of a design choice between the operating system and the compiler so it is largely driven by the operating system memory management architecture how much size is to be given so for example when we see the code of xv6 you will know that the stack is of you know stack size has certain limitations in fact that could be a project you could do in xv6 to you know make the memory management more flexible so we have to wait for that okay before we see memory management siebert has a question isn't the heap like somewhat infinite memory source but here the heap size cannot be more than the frame allocated so nothing is infinite everything is finite you have a illusion of infiniteness nothing is infinite in this world in the computer world everything is finite i mean like uh infinite as in your available ram space but here this is even constrained than the available ram space if you read the man page of malloc it says that malloc can fail so heap doesn't mean that any available memory outside it just means what is any available memory no the available memory will be decided by the operating system if the os says no i don't have available memory then you don't have available memory so as you see if you are mistaken that malloc is guaranteed to work always then you are mistaken malloc can fail malloc does fail malloc does fail sometimes you know i'm saying like this region that is shown in the diagram suppose that is around 10 mb our ram is of 8 gb so once the 10 mb is totally occupied then the mark will fail but we have a lot of space still available yeah so it's a bad os then sir so but the size over here is pre-defined right of the heap yeah so it can be predefined it can be dynamically adjustable it can be dynamically adjustable also for example there is a brk system called the brk system call will give more memory okay so the your program can request the os to give more memory but then who uses brk you have never used a brk system call in your code most typically a code like brk will be used by malloc and free kind of library functions so i'm not sure if some of you have done a data structure project on implementing the malloc and free like your own melo can free library then you would be using the brk system called requesting the os for more memory and so on but depending on the os you know if you have a very simple primitive os you will not have a brk if you have advanced operating system you'll have a brk the heap can actually be adjusted and on modern systems like linux and some of the best libraries like the gnuc library you often have a high degree of flexibility in managing these memory areas see what i want to say essentially yeah it can fail there is no guarantee that a malloc will always succeed because it's possible that you know let us say you are running 1000 programs on your computer right now let's say you start running 1000 movies movie is a huge data because the images occupy a large amount of memory even as a data an image will take you know a huge array you will take you'll need a huge array to store the bitmap of the image and if you are let's say running 1000 videos at a time obviously you are going to run out of physical memory if not virtual memory and we have not yet seen what is virtual memory and it is possible that the the c library starts refusing your request because most probably because the os has started refusing the request so it is possible that malloc fails the question next question is uh how does real lock work shouldn't it overlap with other architecture so that i'll skip right now i'll skip that question that that is reserved for memory management we'll see we'll see how reallock works when we do memory management see it's like there are so many possibilities there okay it just all depends on how is the operating system memory management implemented and how is your malloc and free and real lock library and remember the library is a user level code it's its application code the c library so it all depends on how that that code is written so the the simple answer to your question is no there will be no overlap if there is overlap it will fail remember real lock can fail and one possible reason for real up to fail is that it is not possible to do real lock without overlap and overlap should not be allowed isn't it if your one program starts encroaching other programs memory it's a bad system nobody will use it so real lock will fail rather than doing a nasty thing it will say i don't do your job so that is the simple answer no the overlap will not happen but how does it all work what happens inside will skip it for reserve it for the memory management part i am introducing all these concerns because we have to understand the concept of a process first okay any more questions we are running out of time okay so fine let's go ahead let us just end up with the concept of what is a process control block now when a program starts running it's a process and the process have to be managed by the operating system because it has to schedule them one after another in a multitasking system so what will happen that there will be a record in the operating system data structure which will represent the process the record is called a process control block or pcb okay so os will maintain a list of process control blocks where there will be one pcb for each process the name of the the structure in linux is task struct and the name of the structure for pcb in x26 code is called proc so we'll actually see the code of both of them now what are the typical fields in the process control block now we have already discussed remember the arrow file descriptors we said you know that open will basically return the index into the area file descriptor and so on so the list of open files will is basically the area of file descriptors that will be maintained inside the pcb so it's a structure which will contain lot of fields and one of one of the fields is the array of file descriptors then it will also store the process id which is a unique number identifying the process and uh i'm sure you have seen the process ids in the output of the ps command so for every process the id of the process is also stored in its own pcb then it most typically also has a program counter in so remember this is the structure so it is to be hosted in memory okay the structure is stored in memory and this is a copy of the program counter register which is maintained in the pcb why is it maintained because remember in in multitasking the process will be scheduled and then taken out and scheduled again and then taken out and scheduled again and so on so every time you want to reschedule it you have to somehow remember where to reschedule it right and that is why the program counter there is a location in the pcb to store the copy of the program counter actually and then with the space for registers now why registers have to be saved in memory that is in the pcb field because again when a process is uh scheduled and it is removed from the cpu and you know it waits for some time and again it gets scheduled the registers it was using will be allocated to on the process when on the process schedule so they have to be saved and where will you save them okay they have to be saved in the pcb because the pcb is the necessary information about the process so that it can be scheduled again and again then again a process will occupy certain locations in the memory so which locations it is occupying in the memory will also be stored in the pcb some accounting information may be needed you know for example you want to see how long the process has been running and so on so that will also be stored if the process is doing some io like writing to the disk or reading from keyboard and so on then some status will be maintained and so on uh important field is the process state and we will see the concept of processed it now okay now in order to actually you know control all the processes what the operating system does is that it will maintain different queues or lists inside its data structures so there will be different type of queues or list or some such data structures uh it can be actually a different type of data structure also it could be a tree or a fancy tree or anything but basically all the processes have to be maintained in the data structure most typically these are queues so for example the processes which need to be scheduled one after another that they will be actually combined into a queue then there are possibly cues of you know processes which have requested io and normally let us say for example if your program has done a scanf then now it has done a scanf so now it is all going to do is wait for the user to press a key and now should that program be scheduled obviously it should not be scheduled if you schedule it it is it is going to do nothing because it is just going to do a wait right so such processes which are going to wait for some input or output to complete they will be put into different queues by the operating system again you know we have multiple cpus then multiple processes will run on multiple cpus and that information has also to be maintained so there is a concept of a process state you know from the time of process comes into existence till it is over in a multitasking system the the process will keep actually changing its state from one state to another state so what are the states that we are talking about we say when a process is created that it's a new process that is immediately after the fork exec is over you know the process is in a fresh new state okay now what is done uh is that the process is then moved the pcb is moved to a queue of processes which we say are ready to run and we say if the process is ready to run then it is in the ready state so basically ready to run process is one which can be picked up by the scheduler for executing and now if the process is in the ready state it can be picked up at appropriate time by the scheduler and it can be made to run so when we say the process is in the running state it is actually executing on the cpu it is actually executing on the cpu and the fetch decode execute and change the pc that kind of cycle is now happening with the process now this is a very interesting state because so many things are possible when a process is actually running what are the things that are possible when the process is running it is possible first of all that a hardware interrupt occurred some kind of interrupt any hardware interrupt and including a timer interrupt correct and we know what happens when a hardware interrupt occurs the process will no longer execute the control will jump into the operating system so so suppose there was an interrupt now because there was a interrupt the process cannot run actually so what will happen the process will again change its state to the ready state and the process control block will be moved to the queue of ready processes what is the other thing that is possible that the process it will cause exit or the main of the function main function says written now when that is done basically the process is going to be over and then we say the process has terminated okay when it is terminated all the resources that is the memory areas allocated to the process should ideally be freed by the operating system and returned to the pool of free memory and reused because the process is over so it should not consume any memory after that what is also possible that the process does i o okay that will let us see the process there's a scanf and because you don't know when the scanning will be over then the process can no longer run the process has to be you know kept into what you call as a weight queue the all the let us say there is a weight queue for the keyboard and all the processes which are waiting for the keyboard will be moved to a queue all the processes which are moved for mouse click will be moved to another weight queue let us say the process has said right to the hard disk let us say right system call it has called which is writing to a file and that is essentially involving the right to uh to the hard disk now why should that wait because most typically the the hard disk write is going to take milliseconds while the processor executes in nanoseconds so it's a huge order of magnitude difference so most typically the process will wait for the hard disk output to be complete and that will also be moved to a waiting queue whenever the weight is over and how is the weight is over you know that is all done through hardware interrupt when the key is pressed there is advantage of when the mouse is clicked there is hardware interrupt when the hard disk io is complete with the hardware interrupt so whenever there is a hardware interrupt again the os code will run the os code will determine which hardware interrupt occurred it will determine that hardware interrupt was for a particular process and let us say you know the input was given using keyboard for the process which was waiting in scanf that process will now be taken out of the waiting queue and just put it on to the ready queue so we'll say the process has moved to the ready state once again that is it is ready to be scheduled by the scheduler so you can see the process can go through different states you know during its lifetime the initial state is there and the final state is there from where it you know initial state it normally goes to ready and once it goes to the final terminator state it goes nowhere it is taken out but during its lifetime it keeps moving between ready to running and to waiting and to ready and so on and so on so it keeps moving between these states throughout its lifetime so this is the task struct this is a this is basically the pcb code from xv6 so in xp6 there is a struct proc now what are the things that make sense right now you can see there is a size okay an integer size which basically is the total size occupied by the process in memory so xc6 has a very very very very simple memory management mechanism and that is why all it has is a simple variable a number to indicate the total size of total size occupied by the process in memory the same thing in linux is actually much more complicated okay we will see that then there is a pointer to what is called as a page table then there is a pointer to something called as a kernel stack you will see that later this is the interesting thing the state of the process okay it's a new which is basically creating a prostate number and what are the possible values of a prostate you will see unused embryo sleeping runnable running and zombie so running means the process is running runnable means the process is ready to run sleeping means the process is waiting okay and embryo means the process had just been created that is it's a new process zombie is most typically a state after the process has done exit we will see more about zombie processes and there is unused which is you know just use if the processes that this particular structure is not being used okay then the news will be used then there is a pid as you can see there is a struct proc pointer you can see it's a self-referential pointer to another structure of the same type basically pointing to the parent process which created this then there are two more which are more related to memory management the traffic and the context and same with the channel this is actually the array of struct files as you can see there is a struct file pointer array and uh basically the file descriptor is an index into this array so we already discussed this and now we have seen this it also stores the name of the process in addition to the stuck proc this is just the declaration of the struct box what you will see is in the code of xv6 there is a global variable called p table and it contains an array of struct proc so this is the complete array of all the processes now again xv6 are the very simple data structure all it does is all process are managed in an array and we will see this code in more and more and deeper detail later on in the case of linux so this is a diagram of linux and some part of the linux transcript so this is the pcb in linux kernel the struct task stack it's a huge data structure actually with i don't know like i never counted but maybe up to 100 fields are there because linux is a complicated and very advanced operating system but some of the essential fields that the state okay just like xb6 or any other pc but the state to indicate in which state the process is right now there is a structured entity which tells scheduling information about the process again there is a pointer to the parent there is a list of children and the list of children is maintained using a structure called listed we will see more about that later again you will see there is a array of files so this is the file descriptor array it's a pointer but the pointer is normally you know pointing to an array so this is the add-on index in the array will be the file descriptor you will see that the memory management is not simply through a size but there is a struct mm underscore struct which actually contains lot of memory management information about the particular process so what happens is that the processes are actually linked together they are not in an array but they are linked together okay in the linux data structure in in the list kind of a thing and then there is a global pointer called current which always points to the process that is currently executing so this is roughly some data structure used in linux so we discuss the concept of radicus and weight queues basically in the linux kernel what they have is a list here and the listed is nothing but a structure containing two pointers okay the head pointer and teleporter because all the list in linux kernel are basically doubly linked circular list they are doubly linked circular list and as you know in a doubly linked circular list you will need a head and tail so the struct list here here is nothing but a structure containing the head and tail pointer and so you can see here this is the doubly linked list okay it is a null terminated list but it could be circular as well and then there can be another queue for you know the processes which are waiting for let us say keyboard input and they'll be linked up together in a doubly linked list so this diagram is not a doubly circular list but most typically there are doubly circular list which are used in the linux kernel so what happens during the life of a process the when the process is created using fork exec there is a pcb created pcb is created and the pcb is in on a queue called rediq when the scheduler selects the process and makes it run on the cpu the process structure is actually pcb is taken out of the ready queue and the pcb will be execute the process will be executing on the cpu now many things are possible when the process is executing suppose the process does i o request and then the pcb will be moved to the weight queue and the audio will happen and when the i o is over it will be again moved to the ready queue when the process is executing on cpu again it is possible that the timer interrupt happens and when the timer interrupt happens it will directly be moved to the ready queue because it is still ready to run only its time quantum is over or if it is possible that the process does the fork and creates a child and when the child is created it will and let us see the process does the weight system call then it will be moved to a queue of processes which are waiting for the child to terminate and when the child terminates they are again ready to run and taken to the ready queue or if you know there is a interrupt then the process will be moved to the processes which are waiting for the interrupt to occur and again when the interrupt is over it will be moved to the rediq and so on so that is how you know processes keep moving from one queue to another queue to another q another queue and so on and they keep getting shadowed whenever they are ready so this is essentially how the pcb keeps moving uh what's the time now um okay let me quickly show you um okay we will do this in the next class okay because the time is over and it's your lunch time in the next class i'll show you the task in linux and the listed and how the data structures are actually implemented in linux kernel and then we will move on to understanding the change of process states in more detail that will do in the next class

**XV6 Compilation**   
  
hello everyone today we will start the discussion and training of the x36 operating system code today we are not going to see any code of the operating system but we'll just get introduced to the overall structure of the code and certain files and the meaning of those files and how to compile it and how to run it and so on so before we begin with x36 i just want to introduce uh you to two tools that is c scope and c tags and this is particularly only for those people who use vi editor if you are using some other editor and i really don't know how many of you actually use vx but i'm still introducing it because i use vi so c scope like if you download code of x36 and you run this command spoke minus q with some file specified and if you also run the syntax command and then you download this particular file c scope maps dot whim in your home folder and you add a line source of the cisco maps.beam file in your dot lmrc file then basically what you do is you integrate the tools called c scope and syntax with create i told you this because when i demonstrate the code to you i'll be using vl and i'll be using vi combined with these tools now if you are using some other editors like atom or sublime text or whatever then then you figure it out find how to browse code using your own editor essentially what this commands to is this particular file the c scope maps dot whim it basically defines lot of shortcut keys that instruct vi to do particular things adding this particular line in vmrc will basically execute these commands when vr starts and those commands which have been specified by this they refer to some actions to be done using the c scope and the c tags commands so these two commands basically create something like a database that the c scope and c tags command will use so c scope basically is a core browser and it allows us to jump directly to a function call and get a list of all functions calling a particular function and all functions called by a particular function and go to the global definition of a variable and so on while c tags simply list all keywords and all their occurrences in a particular file so maybe i can show you something so here is my xb6 code i am sure all of you will figure out how to download xc6 code uh do i have a tags file i have a tags value as you can see so when i run c tags uh with uh i think this then the tax file is created the tax file is simply like this you know some keywords and file and line number where they occur so that is the file so the c tags file will be used by vi uh when you want to browse the code so uh yeah that's it these are the tools i am going to use so i just introduce those two to you now if you are reading the code then very often [Music] people want to see what we call as a call graph it so happens that codes like xp6 or linux kernel or let us say databases sorry for that matter any large code will have a sequence of function calls to the depth of 5 10 11 15 like that and then reading the code becomes a tiresome job if you are not able to visualize the call graph that is a calls b and b equals b and equals m and m equals z and z equals f and so on if you are not able to visualize those call graphs then understanding the code becomes a problem so there are tools like uh oxygen oxygen is basically a documentation generator and it can also be used to generate qualographs so this is a standard practice all over the world to write comments using the standard syntax in the code and then use tools like doxygen which automatically create an online readable documentation but then these tools can also be used to generate call graphs so basically what you have to do is uh when you download x6 code and installed oxygen on your ubuntu machine then you have to run this command oxygen minus j toxic config first what this does is it creates a configuration file for oxygen then you should modify that file the template file to actually suit your code for example you specify the project name and some output directory where the graphs will be stored and subdirectories and extract and which files are not to be included now why these files are to be excluded i'll tell you later why we don't include these files because these files are not part of kernel code and then whether to create call graph and color graph so you have to say all these things and then you run the oxygen doxyconfig so it will run for quite some time and it will generate html files uh which include the call graph so you can actually use firefox to see those files and the call graphs right i'm not going to do this you can just try it on your own and it will work so before we start seeing how to compile and execute the code and the details of some more files and some introduction to the exclusive opting system so it's a unix-like operating system unix type because the system calls that it provides are like unix okay they come very close to unix stickers it is a multitasking but a single user operating system so you can run multiple programs actually processes at the same time in a time shared fashion but there is only one user on the system multiple users are not allowed so now this becomes a interesting project to do that you actually make the xp6 operating system a multi-user system and this is not trained to make it multi-user but some of you can take up the task of making this operating system a multi-user operating system then the code has been written for x86 processor and it is in a sense good because most of you are familiar with certain x86 machine instructions so the reading of the code becomes easier for you it does not support all system calls though it supports only some system calls so another interesting project could be to add a significant number of system calls to the x36 operating system the most interesting thing is that the code is very small it is i'll say actually only 7 000 lines of code and that makes it a very good operating system for steady purpose so it was actually developed at mit us to teach the students operating system concepts and that is why we are using this operating system what it does not have and you may not understand these words right now we will get introduced to these concepts when we study different uh features of operating system so for example memory management and uh 4k exact system calls and how to manage files and so on so the con the features like demand paging copy and write for no shared memory fixed size stack you know they they they we don't have these features okay now as far as the design of the code is concerned when it comes to design of the operating system kernel itself that is how should the code be structured then the two most popular approaches are writing a monolithic kernel versus a microkernel approach now this diagram is trying to say something let us understand what it is saying both are indicating operating system code but here this diagram indicates a micro kernel code and this indicates a monolithic code as far as the interface given by the os 2 applications is concerned in both the cases the interface here and the interface here is nothing but system calls while both the kernels are basically going to use facilities provided by the hardware for their own execution and to manage the execution of the processes the difference lies in the way code is structured for a micro kernel and a monolithic kernel for a micro kernel what is done that out of the total part of the code of the operating system certain code like virtual memory code or scheduling code or basic ipc code is written as a small set of modules separate independent models which have no relation to each other other parts of the operating system like device drivers and file servers and so on they will basically interact with these smaller models and the code is quite properly layered quite properly modularized and so on as opposed to that in a monolithic operating system the separation of different components and modules is either unclear vague or you know they are all inter mixed together so yes there is a layering okay the layering of the functions one function calling another is there but very often you know you will find that there are exceptions also which break the layering so for example conceptually we say that the vfs will call the code of the file system but don't get surprised if it calls this code also or if it cause this code also directly so the layering often gets broken in a monolithic curve and that is why it all appears as a code which is all inter mixed together although yes the modularity is still attempted and the layering is still attracted in a micro kernel this approach of modulization and layering of the code is taken to extreme and they just want to ensure that the model relation and larry is followed very strictly how do the different components of kernel code in a micro kernel talk to each other very often the operating system itself gives a service called messaging service and different components talk to the messaging service of the kernel and that is how processors will mostly talk to each other through the messaging service given by the kernel uh in a monolithic kernel the processes may have you know other type of functionalities for talking to each other and different parts of kernel they just don't use other parts of the kernel like messaging service to talk to each other they just call each other functions directly in a monolithic corner so the essential difference i also say that in a microkernel the kernels messaging service will be used by kernel components also but in a monolithic kernel the different parts of the kernel code will directly call each other's functions and that is basically the difference now x26 follows the monolithic kernel approach which is i think not very new to you because you have perhaps mostly written code this way only you know all your life for for applications let's get introduced to another tool called chemo so kmu is a virtual machine manager like virtualbox the virtualbox comes with a nice graphical interface click click click interface and so on kmu is more like a command line tool and just like virtualbox camo also creates a virtual hardware so it creates a virtual hardware with the bios with a virtual cpu ram this controller keyboard controller something called as a epic i o apic and l lipic about which we'll see more later this is for managing of the interrupts now we will see a further that you know in order to run xv6 there is a command to be run and that command is chemo and then there are so many options as you can see here right now this command is automatically involved when you run the command make chemo and we will see a demo of that very soon so we have to understand the chemo command but before that let us run xv6 first so all i have done here is i have downloaded xp6 code you will see that a certain c files files and certain input typical files and so on all you have to do is run make and chemo okay now you will see that the code is getting compiled all the commands that you see in front of you are basically gcc commands compiling the kernel it not only compiled the kernel it also started executing the curve let me just give you an overview so i i have come back to the terminal from where i started the command so see here i said make cameo you will see that it is compiling the mk face dot c then the ulip dot c then usage dot this is the s file so it's assembly code file and so on so you'll see that it is compiling it's running command ld which is based thinker and we will see more about linker later and so on so you see it is running lot of commands obj dumb and one more and a lot of gcc's and so on you will see lot of dcc is getting and files like id dot c are compiled into id auto interestingly here gcc is being run with so many options and yeah it is your job to learn all these options i will discuss some of these options to gcc you know during the course of discussion but please make a task to understand each of these options because it is our purpose in this course to understand everything about xb64 we will see a lot of dotto files are getting created and finally some ld command is being run once again and one more lt command interestingly here you will see that the ld command is combining lot of plotter files together something like obj dump and it is now apparently creating some file like kernel.asm eventually you'll see some command like dd we will see more about these commands finally you will see that a file called xv6 dot img is created here is a file called xb6 dot img this is the file which is basically a hard disk containing the code of xp6 and few more things so this xv6 artist it's a virtual artist called dot img file it's a virtual artist so how do you run this now you start camu camu as i told you create virtual hardware for us you will notice that eventually we are telling kmu to use this file which is the hard disk right and chemo will basically create a virtual hardware and use this file as the hard disk and as all of you know now the bios is there you know which game has created the bias will simply try to load the boot loaders that exist on this hard disk and then that will load the operating system and operating system will execute so this is xp6 running okay in this camo terminal x36 is running and it already comes with certain applications like ls so i did a ls and i see that there are very few files here like a readme file and uh a lot of us like cat eco grab init and ln ls so some commands are already there but not all many commands and that's it okay there is no no more files here very few files and i can actually do a cat from here of the readme file and the readme file could be seen by me here once again i'm going to run ls now you can try the echo command and the great command and init command not e and kill command and so on you can try all those commands later so this is how you start running xv6 let me go back to the presentation sir why was it running the same commands on the ubuntu operating system so the the arguments given to kmu are one is this minus serial moon colon stdio this is how uh x36 is multiplexing its own window into your normal terminal so it is available both ways so what happens is when it multiplexes in your normal terminal you can do the normal copy paste from the terminal okay and so it is with this option okay so answer to your question is with this particular option if you remove this option it will not do the multiplexing on the terminal so you can actually uh you know press ctrl a and see in the terminal and you can get the chemo prompt in the terminal itself so then you get all the advantages of the genome terminal okay because the genome terminal allows you to copy paste and configure and zoom in and zoom out and so on the chemo terminal doesn't allow you that then there is this argument to the chemo command which says minus drive file equal to fs.img index 1 and so on so what is being done that there are there is another hard disk okay fs.img is a hard disk and the virtual hard disk um controller which is created by camo is a ide controller and the index equal to one basically says that the hard disk fs dot img is available in the first slot of the ide and it is basically a disk in a something called as a raw format which is specific okay then minus smp2 basically says that this kernel should run as if there were two cores so multiprocessor it's a multiprocessor kernel very interesting so you can say minus smp 4 and then it will run as if there were four processors minus m512 says that the memory should be assumed to be 512 mb so you can even change these variables minus simply and minus m now let's move ahead and try to understand the structure of the code so let me just close this the moment i close this camu is terminated and i'm back to my prompt so you will see there are lot of files here dot h and dot c files and dot s files and dot asm files we will just get introduced to what these files mean okay and in the next class we will start reading some code so there are a lot of files okay cat dot c echo dot c for test dot c graph dot c is the whole listing here these are not part of the kernel code these are actually application programs and these application programs also get compiled when you compile xb6 they can be used for testing xv6 so for example now i can turn it right so because i am not running xv6 so ls is there and i'll say word count read me you will see that the word com command counted the number of characters and lines in the readme so make file is a is actually a automation tool to automate the compilation let me ask this quick question you can write in the chat how many of you know the make command and make file you can write yes in the chat just to convince me that you know about it and if you don't know the right to know i want to see the nose actually okay if you don't know about it write a no i want to particularly see the no okay one all of you who don't know about make please write no so there are two nodes i have seen now okay not comfortable okay no fine all right so some of you don't know right we'll cover this in the lab today the make command because actually some of you don't know so i'll cover this in the lab so make as i said is the automation tool to automate the compilation process you will notice that all i typed is make and you know lot of commands started running automatically and all those commands combined together they compiled my xb6 operating system code so it's a good tool then there is a file called dot hyphen bosch src which is uh basically we are not going to use this it's for running with the emulator or watches and we don't use it in this course then you will see that there are a lot of dot s files okay and this is all actually kernel code or rather part of the kernel code which is written in assembly together they are very small like hardly some 400 lines of code it is important to understand that some part of kernel code will have to be written in assemblies it cannot be written in c why we will come to this okay later when we understand how compilers work how linkers work how what are the calling conventions why does the kernel violate calling convention in certain places and that is why you cannot use a compiler to compile c code you have to use assembler to assemble assembly code and then link it you will understand all these things uh when we study about system programs and linkage conventions and so on then there is a file called kernel.ld so there is a program called linker and we will see more about linker in a separate lecture as i said when we study linking and loading and assembly and all these processes this file is basically instructions for the kernel occurred to the linker so that it can link the kernel properly and then there are miscellaneous files like readme notes and license readme is readme as you know then there are license which tells that this code is a free software code and so on we have already seen this command make kemu so chemo is an argument to make okay and there can be different arguments to make to make it do do do different things so basically this will compile the code and run during gaming emulator the nice thing is that you know the code is very nicely commented and that is why automatically you can generate a pdf of the xp6 code and which is nicely annotated line by line and the lines can match with the pdf textbook of the xp6 code so that you can you can always refer to the code of the line and the description in the media you run this and you get a pdf there's a command called make mk face which creates mkss program which is not part of the kernel obviously we will see more about this later and make clean will basically remove all the intermediaries uh like for example dot or files that auto files are not really needed once you have linked the kernel they are intermediary so you can remove them with the mclean now what happens during the process of running make file so let me let me close this fine we are not going to run this so let me open the make file so here is a make file you will see there is a listing of files and that listing is assigned to the variable called objects and then there are variables being defined like tool prefix equal to something and there's a code like if else you know like shell code is there you will see the shape code there and then there is a if and if not devs let's ignore this part fine some variables are being defined like cc variable is defined to be gcc and s to be gas and so on now what are we looking for here we are looking for the lines which actually compile the dot c files so you will notice that there is this particular line and it's it has a kind of cryptic syntax underscore percent colon percent dot o and so on so basically it is just this instruction in the make file which is going to compile all the kernel dot c files so there is no separate instruction for each dot c file uh basically this particular line is sufficient to build a dot o for every dot c file it works okay how does it work we will learn when we see the make file we will learn about makefile in the lab session today then there are asm files and uh it so happens that for each asm file there is a equivalent object code file to be created or some of them also carry a counterpart dot c5 so you will have you'll see in make file you know instructions like this that boot block colon boot asm dot test boot main dot c and then certain instructions like these instructions so these instructions will tell how to compile or assemble the code and you know convert it into machine code and you know make it available for the kernel we will not go into details of this we will go into details of this throughout this course whenever we need to see a specific thing the makefile will also create files like underscore ln underscore ls these are nothing but the executable user programs okay how do they get compiled we will see you know in few slides now um i think it will be imperative if uh i think i have some for 35 minutes now um anyway i'll just cover the the outline of the mix file and we'll cover the makefile in the lab itself so x36 dot img is actually the final image of the xp6 okay now how is it created it is basically created using command like dd now what is a dd command dd command is basically a copy copy command but in a different way what it does is it copies a input file at some output file but in certain blocks and it can skip certain blocks it can uh it can copy only a few blocks of the file and so on so it's a powerful command so what is done is basically this command what it says is that read from the dev zero file so it will give you a lot of zeros and create a file called x6 or img which has 10 000 blocks then it says that read from the file called boot block so boot block is created separately and copy into xb6 dot img without truncating it so what will happen is that boot block will become the first block in nexus 6. img and then the next command says copy from the file kernel and copy it into the file exclusive dot img but skip the first block why skip the first block because the first block is already the boot block without truncating it fine so basically what is the xp6 dot img file it is a concatenation of the booth block and the kernel file okay so boot block contains boot code and the kernel contains control code and they are concatenated together and that is the xp6 dot imd which is basically the at disk image of the x36 that's how it is created how is the boot block created the boot block is from the boot asm.s and bootmain.c using certain compilation commands okay so we will not go into details of these compilation commands but the boot block is basically created from bootasm.list so we will see the code of these two lectures the boot asm putman.c how is the kernel itself created kernel is created uh from all the files mentioned in objects so i think now you videos are all these files okay objects are all these files in the make file it's listed so basically it is going to be created from all the obj files and few more files like intrigue auto entry other init code and so on so these commands will basically combine all the object files together and create the file called as karma so we have seen just now how the xp6 dot img is created first the boot block which is created from boot asm.s bootman.c and then the kernel which is created from all obj files right the fs dot img is another file it is another disk which contains basically the ls and all such programs and the readme file okay so how is it created using mkfs we will skip mkf is right now we'll visit it sometimes later then there are a lot of dot sim files okay these dot stream files are called symbol tables and why are the symbol tables useful again we will skip that particular part right now i'm just introducing you to the you know the names of the files and their larger purpose now you know we are done with the discussion on different type of files we will keep revisiting those files when we start studying the code interestingly the total size of the code if you run this command is hardly 10 000 lines out of which 700 lines are the dot watches src file so you can ignore that and the application code itself is some 2800 lines of code so so total number of kernel lines of code is hardly 7009 so it's not at all a big code if you look at codes like linux kernel which is millions of code this code is a very small code and that's why it makes a very interesting case study for understanding operating systems now i would like all of you to try the commands in xv6 that are listed here so for example now i have to close this and run make camo again um wrong spelling so xq6 will run so ls any we have already done there's a there's a file called user test and the user test is actually a set of uh testing code written to so now you can relate with this all of you have done the conformance testing just now the conformance testing was done basically by combining pixel code that was more of a choice you know design choice like whether you could write only c code and only shared code i should combine both of them because it gives a certain flexibility here the code is simply a c code so there is a user test dot c and what you see here is the binary file of that so if you run dot slash user test then it will basically run lot of lot of lot of tests and it will take few minutes to run and now that i have started it i will keep it running so you will notice now that it is basically going to test all the system calls right and few more things okay so i advise you to start reading that particular c code to get acquainted with how the test code is written and what are the different kind of test being run so uh let let this code keep running in the background okay we will switch back to our presentation so you can run ls you can just cat or you can also do like this okay ls semicolon ls and small shell program which is running there to interpret your commands will be able to separate on semicolon and run to ls command you can even use a pipe like this and pipe will work you can use a pipe with the eco and eco is a program which is provided in the code you can use pipe also and you can use multiple pipes for example here i have done multiple pipes and this is with multiple pipes and you can also use redirection like this ls redirected to out you can there's a command called mkdr so you can create a folder and uh interestingly you know like for example you create a folder and you send to the folder on your unless it will not work because there is no concept of a path in the shell here so if you run [Music] something like this it will work okay but if you if you run it from inside it will not work so just saying cd will not work the cd is not a cd like the bash you have to say cd slash and then it will work so i want you to get comfortable with running the commands inside xq6 because the commands are very useful for testing the kernel so yes sir okay uh oh the slide is not visible the terminal let me stop now wait wait so there was a lag also in between so we weren't able to see everything yeah i think i'll have to restart my um start my big session this way i am connecting again from my laptop okay so the screen is visible again and all i was saying is in the slides that i have i'll share the slides with you i have listed lot of comments like use of a pipe use of redirection creating a folder cd ls and so on you should try these commands okay and get comfortable running commands inside the explicits because uh you will be doing certain assignments and project based on xp6 and we will keep discussing the code again and again and again to test the code you will be running the commands so it won't take much time because most of you are already comfortable with unix commands and these are very close to unix commands so it should not take much time for you to try out the commands uh all right now two more files once again there's a file called ulip.c and it is basically the library code so because the user programs they you know the ls and cat and so on these these programs they need library functions like string copy string compare singling and so on so though all those functions are basically nulib.c and i'm sure you'll be excited to read the code of these functions which you might have called very often now there is a usage.oh which is basically a system assembly code file it will compile into the usage.oh so this file is basically to convert all calls like a function called to open because when you write c code you normally want to call open you don't want to write assembly code to do the samsung data so it will basically convert all the calls like call to open into assembly code okay using the int instructions in fact you can use the obj dump open it up if you run it on an object code file it will convert the object code to assembly so you can actually see instructions like this you know if you're an obj dump on usage.oh that the call to open is basically a call to int instruction you can actually see that okay and the same thing you can also see in uses dot s then there is a file called printf.c which is code of printer and i'm sure you should be you know interested to read the code of printer because it's a very heavily used function but this is not the same printf which you normally call from your c code the linux printf code is quite different this is a code of printf returns specifically for xb6 so it's a very small c code and it's not even a complete functionality of printf but it's a very interesting code so you can read this code and when the if time permits i will also discuss this code so the printf code basically here uses two more functions the print tint and put c both of which essentially called the write system call right so that is how the printf is written then there is a uml of dot c which is the implementation of malloc and because the c programs want to call malloc and free so there is a library which does malloc and free that is a umailoc.c this code is actually almost the same code as the code given in the kerning and hd textbook because in their textbook they have implemented a malloc and free and this is the almost the same code fine and this actually uses a system called called sbr to get memory from the exquisite kernel because to give memory to user program there has to be memory available who will make the memory available only the os how does the os give you more chunks of memory using the sbr system call so that was it fine now if you really want to understand the build process in more detail which you'll obviously keep doing you can try this make camu with a pipe and t so what does the t do t works as a t okay t with one input and two output so normally a pipe will give the output of first command as the input of second command but if you use a t here then it will not only give the output uh on screen but it will also pass it on to the further commands so it will do both it will also show you show you the output on the screen but it will also save it in the this particular file so all the compilation commands will be available here yeah for you to observe later now one very interesting thing i want you to understand this okay and you will you will understand it more when we again discuss the compiler and linker encoder see when you normally compile a program on linux for example here i am and let us say i compile a file called mkhis dot c here okay and i compiled it and what was created is a a dot out fine let me remove the dot then what are you doing in this compilation process you are compiling it for a particular target machine you are saying that you are telling the compiler that i want to create a machine code file which should be able to run on a particular processor and os combination um what happens is by default the compiler will detect your processor it will detect the operating system you are running it on and by default it will compile a program for that os and that cpu so basically the adot out that was created just now was created for my ubuntu environment and the x86 processor bit so it all happened by default okay i did not have to tell gcc anything extra it automatically you know detected the target machine right but now while doing that what it did is it took the code from the c library it combined it with my code and now it created a final executable now interestingly the compiler was running on the same os no i'm running open to s right now and the compiler was running on the same os now if you want to compile a program for xs xv6 we really don't have a compiler inside xp6 we don't have a compiler program which was compiled for the exclusive operating system so the question is then how do you compile a program so that it runs on xp6 right so this is a very typical problem that that is called a bootstrap problem so to compile a user land program that is an application for xq6 we do not have a compiler on xb6 so what is done we compile the programs on linux but for xv6 okay and because we are compiling it for xb6 it is it will be running in the explicit operating system it cannot use the c libraries of my ubuntu linux i cannot combine the machine code of those programs with the machine code of the c library code on my linux it has to be combined with some other code and that is why now you can correlate that you know there is a usage.s and so on the the files are there because you can link with those files and they will execute inside explicits so in the make file there is a instruction for ulip and it basically compiles these dotto files what are these dot files this is called a printer this is code of malloc and this is the code of you know string copy programs and so on so all the commands like ls cat and so on they basically get combined with the code from these files but the instructions run on linux so what happens i am running cc on linux i am passing proper arguments to cc telling it don't link with the c library on linux link with these files instead and create a object code file which is suited for x36 so all the all the commands you know that are specified here the lde and so on they are basically telling explicit they are basically telling the gcc compiler and some a program called linker that you know you have to do work in such a way that the executable file will be created not for my ubuntu linux but for xp6 okay this is called cross compilation normally when you compile a program on your host for the host it is a straightforward compilation but now you are compiling on ubuntu but the target is the xp6 machine so this is called cross compilation the syntax here will again ignore and we will see more about it in the lab so this is how actually you know actually you know the cat will get compiled so gcc invoke with you know all these options you see so many options are there okay and it is our job to understand these options you will say then minus c minus o cat dot o so the cat dotter will get created but now you will see that there is a link command in work ld command which is combining the cad dot o with you leave dot o usage dot or printf dot dot and um log dot okay finally giving me the cat program so the cad dot o is not linked with the standard c library but with these files okay and what is the meaning of these options again we will skip right now we will see it later okay so that is how the user land programs are getting compiled we'll skip the mk phase right now then there are some header files and i don't have to tell you the purpose of header files by just looking at the slide i i i hope you'll understand the purpose of the header file so there's a type storage which basically defines certain types like u and q short you can and it's a very small file okay all all it does is it just defines these data types in c and then there is a stat dot h which is used by many programs like ls and so on it basically defines a structure called stack okay and some hash defines and then the fc entail.h which is also used by programs like ls and so on it basically defines these flags i'm sure you can relate to these flags they are used in open or read only or write only so these are defined as defined in fcnt and then there is a user.h which has basically the prototypes okay of all the system calls and the string copy like functions so that is user dotted a compilation of all the prototypes then i will share this slide with you just mug up some numbers because the numbers keep coming throughout the discussion numbers are names like current base ext name current link field stop and dave space etc these these names like current base and xt name are referring to particular numbers which are leads okay current base is 2 gb extreme is 1 mb current link is 2 gb plus 1 mb and so on these are basically referring to different memory locations and they keep coming throughout the discussion so all i'll say just mug up okay mug up these names and their values so that when you are reading you know your life is comfortable so with this i am just done with a very quick overview and let us say an outside introduction to the xq6 code further now we are going to jump into reading the code both the kernel code the the testing code the application code and the make file everything we are going to read and understand in more detail i just have few guidelines which i want to conclude about how to read the kernel code so the key to understanding kernel code is understanding understanding data structures this applies not only to kernel it actually applies to any other code in a large code you have to understand the data structures okay try to draw diagrams of the data structures try to understand how functions manipulate the data sectors okay so basically you know the global variables the type definition the list and arrays and etc you should have diagrams of them ready in your mind or on paper and you should know how different functions manipulate the data essentially what is a program you know it is a manipulation of data structures so if you are clear about how the data structures look like what is the purpose of every data structure which functions are going to manipulate which data structure what are the functions going to do then reading the code of functions is the easy job if you have understood the data structures basically you should know the purpose of every one of them now for example you are reading a code path now we will create a code path of a system called like exec so there is a forecast executing exclusive so when you are reading that code path try to we have to try to locate the key lines of code that do the major work now it so happens that you know in any well written code there is a lot of lot of code which is more of an error checking code now if this is true then don't run the code if that is true then don't run the code or if you know some weird condition is true then take some separate action and so on what we should be able to do is you know try to locate the key lines of the code that do the major one and the error checking code can be initially ignored because we want to focus on the you know more important code but it should not be ignored forever because the array checking code is very very important then a good practice is to keep summarizing so for example you read you started reading the code of exec and you have read 30 lines of code now summarize it okay what have you read summarize it and remember it okay remembering is important otherwise uh you know linking you know com you know linking the code together that is in your mind will be a very difficult job so keep summarizing and remember the summary of the code that you have read and summarize the summaries also so after you've read the 300 lines of code summarize the 300 lines of code also fine now i should tell you and i should warn you and i should seriously advise you that you if you want to read and understand this code you have to be good with c and assembly and os always i am going to teach you to certain extent assembly instructions i will teach you but please revise your concepts of c language okay take time out read the knr's book again read the c faq the c faq is very very important read it if you really want to understand this code because i will not explain you concepts of c you know when i read the kernel code i will assume that you understand c so if there are a lot among you who need an introduction to see concepts please talk to your cr and if your cr makes a recommendation i'll take a separate extra lecture on some concepts of c but otherwise i'll assume that all of you know c all right the relevant concepts of os and assembly i'll keep introducing right but as far as c is concerned please revise the cfaq and so on so that is your job okay next class we will start understanding the bootloader code right now i'm going to stop and in the lab i'll explain the make file so download the there is a question in the chat download the x86 version download the x86 version so i'll advise you to do a git clone actually just don't download a zip file do a git clone and that is a better way of actually approaching the code you do a git clone so how many of you don't know what i mean by git clone is there any like just say it doesn't know i don't know what is this git clone thing you can you can say in the chat that i don't know what is this git clone all right so it's good that most of you know what i mean by git clone so do a git clone of the code that will help and after you do a git clone create a branch if you know git i'll again demonstrate those things in the lab so let's not worry too much about that but better to work on a branch don't work on the master all right so i'm done with the you know very broad and outside introduction to xv6 uh all of you download it run it run instructions start doing that fine that will be your lab task for the week another laptop for the week apart from the other tasks that i have given to you so we'll stop now