There is a comment up here that the system provides PL/I support, and there is a library issue right below that, of supervisory entries and environment interfaces. Most ~~of these~~ things combine together, I think, in the following way. The issue here is that there are PL/I features which many systems have, but the thing that makes it worth putting up on the slide is that in the case of Multics the PL/I features couple into the environment. It isn't just a situation where there's this language feature called internal static which is nice and it allows you to do certain kinds of things, in Multics it actually couples into the environment, it means that you are able to harness some feature of the system in an important way. The best examples of these probably are ~~the~~ slightly technical, if your familiar with PL/I you'll probably recognize them, there are features such as pointers, and base variables, which in the case of our implementation, couple into the virtual memory, that is the way that one accesses the virtual memory. A pointer is a pointer into the virtual memory and that is the end of it. It means that from PL/I then, one can use these features we were talking about before. Another example is that external variables in PL/I, variables which have the class external, map into the objects which are in the file system. They map into ~~the~~ the directly addressed main storage of the environment around the program. In other words, files are the external variables of PL/I programs, and that again simplifies things. Another example is that PL/I conditions, map into the _____ structure that the system provides. That is a fairly standard intent, although some implementations I don't believe fully carry through on that idea. The point is that if you wish you can view it as a PL/I machine as I pointed out before, Multics can be viewed for several different levels. The significance of being able to view it as a PL/I machine is that one can stay in one language for everything. That is, you can construct a fairly sophisticated subsystem or you can start with a small

problem, which ever happens to be appropriate, and grow until you have a sophisticated subsystem. You don't have to leave the language to be able to get to the supervisor or to be able to exercise some of the more interesting features of the system. This, of course, simplifies maintenance of the systems if one is using it for operational environment, and it also means that a system programmer or a subsystem programmer who has taken the risk of building a system in PL/I can expect a reasonable amount of operational effectiveness when using PL/I. It isn't a matter of in order to wedge this language into the system we had to invent funny things, instead its much more a matter of the language harnesses the system and vice versa. So one can expect a reasonable amount of operational effectiveness from it. I guess, just to make sure we don't forget the idea, I've slipped administrative controls in at the bottom of the slide. The idea here is simply there are things such as accounting, access control, performance monitoring and usage reports which are considered part and parcel of the system, they were put in from the very bottom of the system, significance here is that operational control of subsystem users is part of the plan, it's not an add on idea. That, of course, will come out much more clearly in two of the ~~topics~~ talks that are coming out later, tomorrow probably, Bob Rolla and Tom VanVleck will ~~probably~~ be talking about two things which really do go into that some in significant detail. Let's look a little more carefully at the topics numbered 2 and 3 with flavors of 1. This business of virtual memory. I'd like to see if we could perhaps get to the point where its clear what some of the ideas that, some of the motivations that went into this thing are. One normally is dealing, this is kind of a simplified version of the picture we saw a few ~~minutes~~ minutes ago, the point being one has an actual hardware configuration, and if one is writing programs for a system that looks like that, generally he knows that there are two processors, you know that there are three memory boxes, you know that there is some collection of drum and bit.

and that's the reason for drawing the picture in this way. The point is that what you'd like to do is get away from the point where your looking at the real hardware, you'd like to be able to look at some pseudo-hardware from the point of view of the user or the subsystem writer. For that reason, frequently people talk about the idea of building a virtual machine for the user. Here, simply by using a variety of techniques, moving pieces of memory in and out, relocation, time slicing the processor and what not, the idea is that for each different user of the system, whether he be a user sitting at a console, or a job for the card reader, for each user there is effectively a virtual computer inside the system. Almost any time-sharing system can be viewed in this way, as a collection of virtual machines, each virtual processor has its own virtual memory, one of the important issues that begins to come out is that its without too much trouble possible to make the virtual memory have a flexible size and that's the reason for this mysterious comment about flexible sizes hiding in here. More than that, notice that there is an addressing notion that I've brought out, because we're going to have to come back to that. When a program, which is running in a virtual processor, wishes to get something from the memory it uses some kind of an instruction such as GET, and it gives an address, typically one actually writes those addresses as names, or actually writes in PL/I and PL/I constructs the addresses, but the key is that this is an index in one of the virtual memories and there is no real complaint about the fact that this virtual CPU has an address of the same number here because of the fact that ~~there~~ they're distinct. In fact it says that there's no sharing of virtual memory, and that's the place of departure, I think, that's the first point of departure we're interested in. Because the standard time-sharing system doesn't really offer, necessarily without a lot of thought, any ability for people to share information. Well, let's see if we can get to the next step then. This is a kind of half way step so I've numbered the slides in a sort of half way way. The point is that this isn't really the way its done, but it's a convenient quest to see what's going on. First of

all, the idea here is simply that, "Gee if you think about it, why not allow each of the virtual processes to look at each of the virtual memories", and you say to yourself that's straightforward if we had some ability for the processor to say not only which word it wants, but which one of the virtual memories the word is located in. So I begin talking about the idea of segments here because at exactly this point we introduce the notion that these separate little pieces of memory, these pieces of memory are figments of the imagination, if you think about it for a moment, because the real memory isn't constructed like that, but as far as this CPU is concerned they're very real. That's what he addresses, and in fact one may construct special hardware for the central processor and one constructs what is not exactly a two address computer, its a one address computer but each of the addresses has two components. And in general, there's a notion that programs know what segments they are dealing with, that's a number which is known to the programmer in the same way that this address is known to the program. Those two kinds of ideas begin to have segmented addresses, that's another way of putting it. What are the immediate results of this kind of view of the situation, if you can figure out how to implement it, I'll come back to that, if you no longer have to have one virtual memory per user, you can have one or more, and that's an immediate relaxation, of the situation which begins to simplify things, once you organize this way, now we can take advantage of the flexible length, and I've tried to indicate that by the way I've drawn these differently in size, the flexible length is very nice, because now I can put, for example, large arrays in these things and if this processor needs two of them he can line two of them next to each other and one can be longer than the other or it can grow. This one can grow without worrying whether or not this one is in the way. And of course there is something behind the scene making all that work, we'll come back to that. There is another funny implication here, that is, if I take the example of the Multics system today, if I were to go up and look at the system

the way it's operating right now I would find this one or more per user has been hit pretty hard, in fact there might as well be 10,000 segments in the system today. Another observation is that there are 40 users on the system at one instance, that would imply 10,000 of these, 40 of these, and 400,000 little criss cross lines. That adds some flutter to the diagram and as you well might expect there is a corresponding flutter in the implementation. As a result, as we will see in a moment, what we actually do, one has essentially a lot of issues to sort out in order to make that work properly, and what we do first of all is only install those lines which are of interest in any given program. That's an obvious simplification, more than that, in order to find out what's of interest to the program, those lines are installed dynamically, and there have been some references to dynamic linking which may come up again that's basically what one is talking about, the idea of dynamically growing the address space of this processor to include the things which it actually wants to use rather than include all the things in the system. Well that brings us then to what's basically the next example, the next step along the line. First of all, in order to make the implementation work out, a variety of things have to be done, and one of this variety of things is, remember that number 400,000 little criss cross lines, begins to get to be a nuisance, and one of the ways to begin to make that sort out smoothly, is to invent a map. A map between the segment numbers used by the processors and the real addresses in memory. This allows one not only to implement those 400,000 criss cross lines very smoothly, but it allows different CPU's to use segment numbers from a compact set, which turns out, ultimately, to be necessary for effectiveness. We haven't figured out how to allow the CPU's to use segment numbers from a non-compact set yet, there are some proposed implementations but so far, our implementation requires that we try to use a dense set of segment numbers, and therefore, if a segment such as S2 is going to be shared by two different CPU's, and those two CPU's are both using segment numbers from a

dense set, there is some problem, of overlapping segment numbers and the address map allows the CPU's to use different segment numbers for the same segment. That's perhaps getting a little bit deep without having all of the machinery behind it, there are a couple of papers that have been published, one ~~in particular~~, entitled the "Multics Virtual Memory", which goes into some depth as to what is actually going on in that mapping mechanism there. The point here is that the maps are introduced to provide an economical implementation and another nice thing is that they just happen to produce, very nicely as you go through the map, an opportunity to install access control. That is, to control the arrangements, so that segment 2 may be readable by this virtual CPU and readable by and writeable by that one. That's an important ability, to control the situation with some precision. Well let's put the segmented memory away for just a moment, and look at one more idea. Frequently, I think this is probably, as in slide number 1, there was a slide a little while ago which said, virtual machines, and it showed half a dozen or three or four completely independent virtual machines, that's a traditional way of viewing a time sharing system. In another observation, this is a traditional way of viewing a file system, there is some directory, stored in the directory, the directories contains names of files, the files are variable length, they have symbolic names of various kinds. The directories might be arranged in a hierarchy, in fact in Multics they are. But that isn't important to our picture. The thing that is important, is that the way you look at this file system, is that the user program calls ~~the~~ a supervisor program which has the name file system, and upon calling the file system, it issues something such as read and it gives an address of an area and the file system copies information from the appropriate file, if it says read from E, this is file E. It copies this information into the data area and that is essentially the kind of ~~comment~~ commerce that goes on. A certain kind of ability to share information

is frequently handled by putting links among the directories, that is this directory may contain the name of a file over there although you have to be a little careful that you don't get confused and allow the physical address of file C to appear two different places, or if you do let it appear two different places, you begin to get into problems by making sure that when you move the file everyone gets told about it. That therefore turns out to make links a non-trivial thing to implement, but it is a key way to get started in the sharing business. I bring that one up because the actual Multics file system in virtual memory are the merger of that slide and the previous slide, in this form. What we have done here, is that we have backed off just a little bit and said, "Well look, we have this ability to address a very large address space, once we have decided that you can look at things with segments, and two component addresses, why not pretend that the contents of the directory are actually in the address spaces of the central processor. You don't have to just pretend this, you can actually implement it so that the processor thinks that, and that is the key issue. In other words, the two slides are merely merged and this is the view that the subsystem writer has, as his program runs, his program has direct address to all the things that are currently attached, it doesn't have access to some things over here, which are still in the directory but it can map them in. In other words we can place another line across here dynamically. Of course, if you think about it a little bit, you'll notice that one thing didn't merge very well in our picture, I'm using names for our segments here and in the earlier picture we were using numbers, and that leads to an dual addressing scheme, that is, everything in the system is known by two names. That is, the symbolic name of the segment and the numerical name that the processor uses for it. Here is a place where there is a good opportunity for future research to understand how to reduce to a single naming scheme that is usable everywhere and still maintain an economical implementation on this size. On the other hand, the fact that we have two names, means then that there do have to be maps, there are indeed maps behind

17

the scene which allow the user to either, by a call to the supervisor once or perhaps behind his back when he tries to touch a piece of information which he has never touched before cause segments which have names to start with in the directory system to be mapped into the address space using segment numbers. Well the implication of that begins to show up, I'll show you a sample program. Here is a PL/I program, if you are a PL/I buff you probably already recognize that it's in an old version of the PL/I syntax rather than the current version, because there are a couple of missing declarations which didn't use to be required but now are. If your not a PL/I buff, if fact if you've never seen PL/I before you may still be able to decode what's going on. This is a procedure which, by my claim, is written to reach out into the environment to a segment named C, which a piece of the permanent storage of the system, a segment maybe, I created it yesterday and I put some things in it. It reaches out, segment C, considers that segment to be something which contains a thousand numbers and adds up those thousand numbers and returns them. That's all it does, the point is that there is no I/O in this program. There is no storage management in this program. The program is dealing with online storage, and yet it is doing so in PL/I. Let's see what's going on here. The key is that the variable C is declared to be external, and that is a tip off to the PL/I compiler that this variable is not one that is is supposed to make a storage for in this program it's one that will be discovered dynamically in the environment at the time the program is executed. So if we decide to run the program, we compile it the PL/I compiler leaves behind a program which has in it, at this point, and it tends to add from a location which hasn't yet actually been specified yet, but soon is filled in, and this has to do with the dynamic connection of those 400,000 lines, we begin to actually execute the program, we come into here, the first time we hit this, we will trigger an interrupt or a fault and go off to a little program which will ask the supervisor to map segment C into an address space,

we will be able to fill in that instruction that was set up to do the add, and now this thing will run like the wind in ~~that~~ a sense. That is, it will run at full speed, an ordinary addition loop, exactly as you expected, behind the ~~xxxxx~~ scenes the actual information belonging, representing those 1000 integers will be paged in dynamically, that is probably in this case one page of information will be snapped into memory and will remain there as long as this loop is running and when the loop is finished and returned that page will ~~xxx~~ drift out sometime later under control of the paging algorithms. This is kind of a very simple example, its intended to exhibit what the key issue is in the smallest form I've been able see how, it becomes much more significant if you begin writing a program which does something more sophisticated, instead of this it is trying to invert a matrix. There are lots of programs around to invert matrices, in fact, there are two kinds of programs to invert matrices, there are those that invert matrices that fit in core memory, and there are programs that invert matrices that don't fit in core memory, and the differences between those kinds of programs may be a factor of two or maybe three in size and the difference is in input/output statements, moving things in and out of core memory. In the case of Multics one can write a matrix inversion program that will work over a wide range of sizes. In fact if it will fit in memory, that's great, it will run very fast. if it doesn't fit in memory that's too bad, your going to have to be slowed down while things move, but that's an inevidable consequence ~~if you going~~ of trying to invert a large matrix. The programmer instead of doing explicit movement of information instead can consentrate, first of all on his matrix inversion technique, and secondly he can concentrate, if it bedomes an issue, on the question of what is the order ~~of~~ in which he is exploring his matrix, because now the key issue turns out to be, how many different things ~~xxxxxxxxx~~ am I trying to touch in a short time. Rather than, do I have this piece here right now. And the issue of how many different things I'm touching seems to be a more fundamental issue for the programmer to be grappling with ~~xxx~~

in terms of algorithm and technique, than the mere question of which piece is here right now. There are several other ideas, I think that at this point I'll stop pushing on the virtual memory notion and mention a variety of other ideas that we have that may help fill out the picture. The point here being that I can talk about segments and the way that they are implemented all day. It's not obvious though, how they are used. How did we intend it to be sorted out. What's the real meaning of this on the implementation. First of all there are catalogues, the catalogues or directories or how ever you prefer to talk about them, are in a hierarchy. And this means that some simple organization problems have already been solved. In fact is doesn't really solve and very sophisticated organizing problems, but it does mean that the casual programmer who's got a couple of dozen files can keep them organized by the various thing he is working on. In fact even a medium sized project, or a fairly large project, such as the developement of Multics itself, which represents a thousand some source modules, can be handled by placing these modules in a kind of a subsection of this hierarchy and organized in such a way that people can find there way around without having to go ask questions, without having to wonder if something is in a given place at a given time. To give a feel for how all these pieces are fit together, we assign one process for each user. I suddenly begin using the technical term process here, and it's probably appropriate to define it, because you'll hear it again later, the virtual CPU that we saw on the earlier page plus its address map is the closest I can come to the definition of a process. Its the thing inside the machine which is the users agent, the thing which is actually making progress and doing work. We create one of these per user normally, and by user that's a person at a typewriter console, or a person who has perhaps submitted a card deck and wishes to have it run later. In fact, in principal there can be more than one process per user. The machinery is all there, the system uses that machinery to create it, that is to get the users themselves going, we have not opened the gates to a lot of users to do

multiprocess computation simply because we don't feel we've got all the ~~data~~ inter- faces tuned quite right, and we'd rather not get people too deeply into that without understanding it a little bit better. The idea of how many segments are you talking about when you say segments do you mean that my address space is divided up into three pieces? or what? The idea here is we are geared up present so that the numbers on the order of 1000, which is possible, and let's indicate what's going on here, I say that the present design would allow a process to go to 1000 segments,, and we've had a few situations in which ~~it~~ that ~~exactly~~ is possible, in which ~~x x x x x x x x~~ processes have grown to that number of different segments in their address space at once. Typically, the numbers around 150. One would expect typically to find 150 segments in any given process. Another key issue, having to do with those criss cross lines, how much sharing are you talking about, typically of those 150 which you find in a process, 120 of those might be shared ~~spaces~~ segments, and that's an important observation. That means that sharing is being used in the system in an important way. Now of course that isn't because ~~x x x x~~ users have gotten so friendly that they're sharing all of their programs, what's really happened, is that the supervisor falls out. In other words, if you start out with the view your trying to implement shared procedures, all of a sudden you realize that the supervisor consists of nothing but one great big collection of ~~the~~ procedures which are being shared by all the users, and that's what a majority of those 120 shared segments are. They are ~~x x x x x~~ pieces of the supervisor which appear in every address space. That is, every user has them. But again you have an interesting point of view that because it's organized that way its now possible, for testing purposes, for one user to have a slightly different piece. One of the pieces of the supervisor can be different for one user, if you want to do that, and that way, it turns out to be a key way of making changes to the system and testing and adding improvements to the system. Another observation which is perhaps closely related down at the bottom here, is that segments are used for everything.

Everything which is stored in the system is in a segment. Procedures, and data are stored in segments, the supervisor is stored in segments, in fact, the supervisor itself, uses the virtual memory. When the system is first bootstrapped, the system contains, as Corbató was mentioning a few minutes ago something on the order of a million words of code, and the amount of that code which is written in the raw absolute hardware mode of the computer is about 100 instructions, and those 100 instructions appear at the very front end of the bootload tape. When one pushes a button those map in and the first thing those 100 instructions so is to build a primitive virtual memory environment in which to begin reading the remainder of the supervisor into, and the supervisor from then on is using the virtual memory, although obviously some of the features aren't there, dynamic paging isn't working at it, you only have 100 instructions in memory, but segments are, that is the supervisor is a collection of segments and that feature is working immediately. This means that the supervisor itself can be written in PL/I, all the way back to except for those first 100 instructions, which is a very nice feature. So segments store everything, and I think that's It's interesting to observe that there are a variety of problems, a whole bunch of different things that you run into, in an operating system which turn out to be satisfied by this one mechanism and that's one of the important ideas that we have tried to propagate everywhere throughout the system. Economy of mechanism, if we can find a way of making one mechanism solve three problems we will do so and see if we can't find, looking to see what's common about those problems and see if we can't find a way of making this thing work very smoothly. It's probably worth the mention of some of the machinery that's behind this, the techniques that are used. I'd love to go into detail about each of these, each one of those topics probably requires a full hour. In fact, the first two together one could easily spend a day on, going into length to explain how it is that one harnesses those two ideas without getting snowed under with the details, and keeping a system that runs fairly smoothly. But that does seem to work giving that one gets geared up

properly, they are a fairly sophisticated paging algorithms. I mention here demand paging, in fact we have now recently gone one step further and we are playing around with and actually have managed to get some additional performance out of pre-paging. I mean by that some predictive guessing as to which pages will be necessary based on previous running history of the program. The algorithms to try to do this are fairly complex, they have a - we have it fairly highly instrumented, and we keep score on how well it's doing, the idea here is that if you can predict which pages are of interest you can bring them in at a time when your not doing anything else. You can overlap that completely with other things, it's very easy - the extent to which you can predict what's going to happen, you can take advantage of that fact, it's very similar to that stock market. On the other hand, to the extent that you make predictions and you foul up is also very similar to the stock market. The questions is whether it overall pays off. We find that if we bring in 100 pages by prediction, we usually find later that 60 to 70 percent of them got used and it turns out that the net result is a slight gain, some what of an improvement. That's really going off into a much deeper talk than I, that's really the subject of a technical conference where something is intended to be much more technical. It might be useful to mention a couple of numbers to see what's going on. When we have forty users on the system, we're moving 50 to a hundred pages per second, back and forth out of our memory. That of course is a very hard number to interpret all by itself, it's an absolute number just sort of sitting there, but it gives you a feel, a little bit, for the kind of traffic that is going on, it means that every 10 to 20 milleseconds some piece of software is pushing some piece of data around and it gives you a little bit of an idea of the kind of headway between interesting events inside the system.

The multiprogramming of course couples very tightly with the demand paging because, while we're waiting for a page we try to run another program and that leads to, as you might guess, no end of complications. The length of time it will be before that page comes in, I mentioned down here that there's a high performance drum, in the time that you demand a page until it shows up, is sometimes 10 milleseconds. It averages about 15 or 16. That means if your going to decide to switch to another process, you'd darn well better have them there, you better be prepared to switch to them in a great big hurry, and expect to get something out of those 10 or 15 milleseconds, which isn't very much. On the other hand, that's the thing that gives us, that's the difference, I think, if we turn it of, that's the difference between running 25 users and 40 users, that's a very significant piece of machinery. The multiprogramming is done in a variable oasis, that is we dynamically vary the number of tasks, which the multiprogramming algorithm is concerned with, the numbers that we observe, it typically runs from two to eight people simultaneously, in core memory, and that number changes about once a second. that is, about once a second is the headway which the multiprogramming system decides that we've got too many or we've got too few, or one of these guys just left, I'm trying to give a kind of a feel for the overall view of what's going on inside the system here. Time allotment is an old trick, we have a scheduling algorithm, which is really not too different from the scheduling algorithm devised for the CTSS system, (it certainly is) the view that the user has of the scheduling algorithm is about the same, the view that the system has is somewhat more sophisticated because it turns out it interacts with multiprogramming and demand paging very hard, but the idea here is that we limit the length of time a process can run to one or two seconds the first time we run it, we run it at high priority, but only for a couple of seconds, to see if we can get him out of the way, to see if get a fast response to a guy that's got a small thing to do, such as a simple request to

change one line in a program. The idea is that overall this gives when the load is full, the system is fully loaded this scheme means that if your trying to do something small you get responses ~~in~~ on the order of 5 to 8 seconds of waiting. If the system is lightly loaded response is about as fast as the typewriter will turn around. Of course, that response time, its is the thing that determines whether or not the system is fully loaded. That is, we record a number on the system, that number really is the number of users beyond which the ~~xxxxxxxxx~~ ~~xxxxx~~ response gets miserable. ~~XXXX~~ The time allotment is the technique, which is hiding underneath that scheme. The high performance drum I already mentioned under multiprogramming, and demand paging, the only observation to make there is that it's built around a hardware queueing facility which minimizes the amount of work the software has to do and it's very important because it gets hit so often, the software is coming in very ~~often~~ frequently asking for things on that drum, and it's important that it not have a lot of work to do in order to tell the drum what to do. This is ~~xxxx~~ sometimes called a sorting drum, or a multi-queue drum. Things like this have been described in literature in a variety of places, there aren't too many places where one of these things is hooked into the general purpose service system however. Finally, in order to make that virtual memory work, we had to modify that processor fairly hard. We had to actually go inside and make the processor use two component addresses, everywhere, for the instruction counter, for the registers, for the addresses and instructions, and so on, and this implies a fair collection of machinery inside the processor to make it go. But again it's one of the supporting techniques which help make the whole thing fit together. In summary then, the Multics environment, I'd say, provides a base for the ~~xxxxxxxxx~~ subsystem construction and operation, I think the easiest thing for me to do is repeat a couple of ideas I mentioned earlier. It does this by providing already solved machinery to handle configuration dependents, overlapped IO, memory and storage management, address allocation and in particular, sharing of information, ~~and~~ By removing this list of problems, the problems that

the subsystem writer expected to solve, means that he can pay attention to his own problem, the problem of making his own system better, more human engineered etc. At this point, I think that we have about 15 or 20 minutes available for questions and we may be able to pick up questions that were left over from the morning discussion also.

Questions

Jean ____ If the subsystem is lighter,
The question is do you dare to let loose a virtual memory with a large memory address ppace, and a powerful sustem behind it to support it, if the neeve programmer is likely to not completely understand the implications of this.
Answer: The answer is 'yes'. You have to worry about this. You definitely have to worry about this. It's a tradeoff problem. As is well known there are two kinds of people who, I should say there are two technologies of sorting, if we use the example you provided. There is a technology of things that fit, and a technology of things that don't fit. People who talk about sorting things that don't fit talk about multi _____ tape mergers, and people who talk about things that do fit, talk about algorithmic searches and this sort of thing, and hash codes, etc. The key question here is whether the payoff from one of the key questions, there are several, one is whether the payoff from not having to worry about two technologies, which certainly has some effect, that means that you can stop talking about two different things, and get these two people talking to each other, and in between problems no longer lead to terrible troubles as they slowly grow. There is a payoff from that, you have to be very careful as to whether or not that is overweighted by the potential costs of some inefficiency I think even more important than that is that there seem to be some fairly straightforward tools you can provide the programmer with to allow him to understand what he's doing, for example, one of the things we did on Multics as an extension of CTSS, CTSS whenever you typed a command, the system typed back

to you the amount of computer time it used. On multics we type back at you not only the amount of computer time it used, we add the number of pages we moved as a result. What that means, the programmer, every time he runs his program, he sees that number right in front of him and when he makes a change, he sees the effect of that change printed out on his console. Not just on his response time, if its his response time he's not sure why he did it wrong. But if it comes out in that number he knows what he's done wrong and it means that, we have found that it allows the user to understand that he's made the mistake, the next thing is the tool to allow him find it. As a byproduct of our predictive paging scheme, it happens that we keep on a per user basis, a little ring buffer of the last 200 pages he moved, and it turns out to be very straightforward to allow him to print that out any time he likes. So we provide another tool, whenever he feels he's in trouble, he can print out that list, he looks at this number and it says that simple thing that you did caused us to move 85 pages. This guy says I didn't call for 85 pages I don't think, he can type the page trace command with the number 85 as the argument and it will list what the last 85 things brought in were and look through that and say Gee I didn't realize that I was doing it so badly. and that pair of tools has been used throughout the system itself, and we see the customers of the system also using that tool. I think that ultimately that kind of thing is the answer. The real answer, I hope, ultimately, is the kind of thinking which one does in the direction of what is the range of things I'm trying to touch all at once. That's a fundamental idea, which somehow seems like one of the crucial inputs to the question of what am I doing with the computer. The idea of what is the list of things I'm touching right now seems to be a much more useful thing for people to be thinking about than which overlay do I currently have scheduled to come in next. In other words it allows one to consentrate on the essence of the problem and I think that is important.

The problem of course doesn't go away.

Question: It seems to me that you can feel some of the arguments that are made as to whether the totally separate the worrying about storage, etc. from the title of the program, in essence what you are saying is if I now become _____ enforce my program into a mode where it operates efficiently on my storage, then I'll run much faster, I don't necessarily have to do that at a expense of time, but isn't that really

end of tape

Answer We're setting up a straw man here. And then trying to shoot it down, I claim. There have been people who claim that paging is going to somehow going to milk blood out of a turnip and that isn't the idea. Because there isn't any blood there, if you are using the system to its halt, there nothing you can do about that. If it's an inherent property of the program, that it needs a million words of memory, it's not going to run well with less than that, and the question is, what is the technique that I used to make it run well. Now, in other words, what I'm trying to say is that every program, if its a big one, has a performance problem. The thing which has been removed from the programmers immediate set of worries is, what is the list of addresses I'm dealing with, do I have to reuse these addresses and in what way do I reuse them. And now that I've reused it I have to be careful that I've told everyone who knew the old addresses about the new one. That kind of consideration is gone The kind of consideration for scheduling the I/O makes sure that that overlay arrives here just in time when I need it. That kind of consideration is gone, what is left, is as I mentioned, what I consider to be the essence of the problem, what is always going to be there, the issue of what is my program really doing, what are the different things it is really touching in order to solve its pattern, and is there another algorithm, that perhaps, in a more sophisticated way touches fewer things at the same time and therefore, in other words it seems to me it concentrates on a more significant issue. If that problem doesn't go away, and I'm not sure, perhaps the thing to do is to go back to the original statement of freeing the

programmer for worries of storage allocation and qualify that to indicate what aspects of the storage allocation we're trying to free him from.

Question: are you saying that the compiler should or should not have to worry about overflow of pages

Answer: In general he shouldn't.

Question: He shouldn't even provide facilities for what happens to his tables of a reasonable size overflow _____

Answer: It's a hard question. The issue is what should he do overall to make his compiler run well. Whatever environment he's dealing with. The answer is that we don't really understand too much yet, about the meaning of using a very large number of addresses and in completely uncontrolled way. So I'm not quite sure what the correct answer is to tell your compiler writer. In general, I would like to tell him, don't worry about reusing addresses, we'll move things around for you, worry about how many things you try to use at once. That's the closest we've come yet, it's not a doctrine in any sense, but it does help sort out things. Let me give you an example, I think it's an important example. We have a PL/I compiler today which is not our first PL/I compiler, our first PL/I compiler produced very poor code, for many of the various kinds of constructions of interest. We ve put together a project which was going to develop a new compiler. Fine, they sat down, they worked our they're new compiler, they decided they wanted to write it in PL/I, because it was the obvious tool, and allowed them everything they wanted to do. So they began to write it in PL/I, and it was very quickly evident that they're new compiler, if translated,by the old compiler, was not going to fit into memory. In fact, it didn't fit in memory by a long, long way. It was just huge. On the other hand, in a typical programming project, what that meant, what it could have meant, was that they had to stop work, and couldn't begin any debugging, really until they had figured how to carve it down and make it fit. Instead what they were able to do was to go right on ahead and write this compiler which was not going to fit, they ran it, they compiled it with the old compiler, it didn't fit,

fit, it ran very,very slowly xxx because of the fact that it paged so hard, but once it was working well enough to translate programs, they had it translate itself, and all of a sudden it fit. And the point is, that six months xxxxxxxx or a year of development time were cut out of that project because they didn't have to worry about the overlay. Now that's the kind of issue that I'm trying to get at.

Question: Measure of memory utilization this number of pages trying to access at once ____ which people trade off against_____ what's the tradeoff tradeoff xxxx relationship between programs

Answer: We haven't discovered that relationship by any means. XX I think we have an environment now, which one can explore that relationship in a very interesting way. Suppose I program it this way, suppose I program it that way, compare the two, then sit back and think. What was the abstract different between these and then sit xxxx back and think

Change tape

your still doing it by yourself.

Question: couldn't catch any of the question. Only the words some pomparisons.

Answer: That's awfully hard to do. I only know of two other time sharing systems that try to provide the virtual memory interface to input/output which allows one to directly talk to his environment this way, these two systems are the TSS360 and the 5020 time sharing system of Hitachi. and compared with those two systems the objectives are similar, very similar. The implications are quite different, the effectiveness of the Hitachi system is quite xlimited because they have been forced to work in a very small core memory environment although it does seem to work. The effectiveness of TSS are fairly reasonable, so in that sense those two are the systems which are most xxx comparable, one can go to a variety of other kinds of systems such as the Dartmouth time sharing system, the SPS 940 system, COL360, several others which provide, I would say

that Multics provides many of the same features that they do, but I will say none of them provide the xxx range of features that we're talking about here. I'm not sure if that's responsive to your question or not.