Acknowledgements: I WOULD LIKE TO THANK Doug Wells, Dave Reed and Bob Mabee for some of the ideas below and for the criticisms of my earlier attempts at ddescription.

Disclaimer: This memo is an early attempt to get some of my ideas down on paper. It is not meant to be read by anyone who does not already agree with the ideas. It can, I hope serve as a basis for discussion amongst the converted.

A CENTRAL CONCEPT IN Multics is that of a process. Processes form the basis for communcation within Multics via the interprocess communications mechanisms. Processes are not, however, the only entities which must communicate. The mail (and send_message) commands provide for communications with more abstract entities. The mailbox (or con_msgs segment) serves as an intermediary between the sender and receiver and thus defines the details of the communications. Another example of extended communication is that between a user and an absentee job via its input parameters. The description of an absentee job contains a set of initial parameters. But it can't be said that these parameters serve as messages to a given process: they serve to communicate to the set of processes that may be created by a single absentee request.

The process is fundamental to Multics, but our current terminology is not sufficient for describing the relationships between processes. I take a process to consist of an address space, an execution point and an associated principle identifier. A process is uniquely identified by its lock id but is normally identified by its process id. This is both inconvenient and

insecure since it is easy to accidentally communicate with the wrong process since the lock id is not normally involved in interprocess communication. I will therefore assume that a process id incorporates the lock id and is therefore unique for all time. In practice a process my maintain (via software) its own set of control points in a given ring. At any time only one of these control points is active, i.e. is being maintained by the system as the process' control point. I will use the term "task" to describe such a control point. A group of these tasks share the address space of a process.

An absentee request is an example of a description of a process. A more common example of a description of a process is the user entry in a project definition table. When a process is created its initial state is defined by the intersection of its descriptions. For example, the parameters in the login line are constrained by the description in the pdt which is in turn constrained by the project's definition in the system definition table. Similarly absentee jobs are described by the request in the absentee queue subject to the constraints of the other tables.

While a process serves as the receives for ipc messages, the message segment serves as a receiver for its messages. One difficulty with message segments is that they are passive receives and must be polled to discover their contents. We can conceive of a message segment having associated with it a process

that is always waiting for a message to be received and waking up when the message does come. The problem here is that the life of a message segment is longer than the life of a process. This difficulty can be handled if we consider a message segment to contain a description of a process rather than being associated with a process. Whenever a message is received a process is created from this description to read the message. This forms a more general model for interprocess communication since we are not constrained to simply communicating between waiting processes. Instead we can conceive of the message being sent to the process generatd by the description in the message segment. For simplicity we can call the description itself a potential process since it has the same characteristics as a blocked process. (These potential processes have other, more significant implications in that they permit a user to leave a description of a process that can be activated by any user who has appropriate access. This provides an alternative to rings for creating protected subsystems and does not have the flaws in the ring mechanism that come from an attempt to share a common process and from the fact that rings impose a strict ordering on which subsystem is protected from which within the process).

Potential processes are not complete substitutes for kinetic processes they are not as rich. Kinetic processes contain much information that has been developed in the course of its operation and is represented in its bindings. The creation of a

process (even at 100 milliseconds) is a fairly expensive operation and one can economize by using an existing process rather than creating a new one. Thus one must have the option of using an existing process to handle messages received in addition to always creating a new one. In general, a message segment can be viewed as representing a queue for requests to a family of processes, some of which may be represented as a potential processes until they are needed. When a potential process is activated it may then go blocked to wait for a new request after processing its initial request, or it may destroy itself so that a new process must be created for the next request.

The use of branches in the hierarchy has a number of advantages over the current ipc mechanisms. It permits access control to be applied to messages, it generalizes that types of entities that may communicate and it is more flexible with respect to the messages that may be sent. It can be implemented by a combination of the current IPC mechanisms, message segments and a variation on the absentee manager.

There is a special class of interprocess communications that falls outside normal interprocess communications. The interprocess signal mechanisms differs from IPC in that the messages serve to interrupt a running process. These interrupts are (unfortunately) converted into pl1 conditions by the fault handling mechanism. The message can be easily handled by the above scheme by creating a new process instead of interrupting an

existing one. The new process could then have access to stop or otherwise control the first process if necessary. There are, especially within the current Multics constraints, some reasons why it might be desirable to interrupt an existing process. This can be done, however, within the current interprocess communication mechanisms without the need to restrict the mechanism to four special signals. The purpose of the IPS mechanism is to force the process to read some messages without the necessecity of going blocked. This can be accomplished more simply by having one IPS signal that tells the process to interrogate its IPC channels for an important message. The user would then have a choice of designating which messages are to be considered important. The use of this "kick" signal and the subsequent interrogating of IPC channels would have the benifit of separating the processing of theseasynchronous signals from the pl1 condition mechanism and permitting them to be handled as per process faults. The fault handlers would then have the option of converting them into pl1 conditions or processing them in some other manner. One immediate effect of the new mechnism would be the ability to have a quit signal be associated with an I/O device instead of a process.

This "kick" signal and the associated mechism can be implemented easily within the current Multics signal by using IPC channels for the four current IPS signals and permitting users to send "kick" signals when they deem a message important. This

would, for example, find immediate application in the send_message program. A reason for hesitating on the implementation is the problem caused by a user maliciously flooding another's processes with these interrupt signals. This problem is not real in that it already exists with IPC CHANNELS AND WOULD NOT BE SIGNIFICANTLY AGGRAVATED by the additional capability to send interrupts. Even if the mechanism were restricted to special processes (as is the current IPS mechanisms) it would still permit improvements in such common signals as "quit". By using message segments as intermediaries for IPC messages one gains, as a byproduct, the ability to check access in the sender's process. This not only solves the problem of too many interrupts being sent, it also solves the problems of too many unauthorized IPC messages if all interprocess communication were required to go through message segments. (In the case of a null message, these is very inexpensive in that no message need be written in the message segment). Additional access modes pertinent to interprocess communication could be added to the message segment. These would include access to send a wakeup, access to create a process from its (protected) DESCRIPTION and access to send an interrupt to the process (or processes) associated with the message segment.

One consquence of the mechanisms described above is the ability of users to make effective use of multiple processes. The mechisms described are sufficient to coordinate cooperating

processes. Additional mechism is necessary to permit processes to control each other. Access must be associated with a process description and with a process to describe its relationship to other processes. The increased use of processes points out a dificiency in the current access control machanism. Presently one's ability to specify access is limited to a combination of a person's id, a project id and an instance (misnomer) tag. These provide resolution only to a family of processes authorized by a given user on a given project. Two additional capabilities are needed. The first is the ability to specify access for a given process. This can be done by permitted a process id to be component of a principle identifier. Secondly, a new family of process can be identified and should be easy to describe. This is the family of processes generated from a given process description. This component is formed from the unique id of a message segment of the process description and becomes part of the process' principle identifier.

The following are some of the types of access that processes may have with respect to one another:

- g The user may Generate a process. This access is associated with a description of a process. The generated process will have a principle identifier created from the principle identifier of the user who created the description, not from the name of the user who requested the generation of the process. (The requestor's name is available to the generated

process.)

- i The user may Interrupt the process. This access is associated with the description of the process and applies to any active process generated from the description.

- c The user may control this process by starting and stopping it.

- d The user may destroy this process.

- e The user may Edit this process by modifying SCU data or by modifying the KST information.

Other relationships between processes must also be defined. For example, it would be very useful to have two processes whose KST's are synchronized so that they can pass address pointers between each other. Also, a process may be definied as being dependent upon another such that if the superior process dies the inferior one also dies. This is important as a check accident runaway processes.

How would processes be used? This is a fairly open question. In the initial Multics design much use was to have been made of processes for such tasks as operating a user's console. Perhaps it is again time for us to consider such applications.

This is the second in a series of unedited ramblings about processes. All disclaimers that apply to the first rambling apply to this one.

The comments in this memo represent a more modest version of the proposals in the first memo. This memo concentrates more on what can be done with minimum change to Multics. I still claim that the more general ideas presented in the earlier memo should be (and can easily be) implemented in order to make processes truely useful. Since this memo is more concerned with immediate implementation for experimental, if not practical, purposes, it will go into more detail about the implementation.

1. I will define three main levels of sharing between processes:

   i. Two processes are not assumed to share anything, or at least very little. Communication is via an I/O stream. The proceses themselves can be considered as I/O devices with special order calls suited to the needs of controlling processes.

   ii. The two proceses are assumed to be running on the same machine (same Multics?) and therefore can share segments. Communication via shared storage can supplement the communication via the I/O stream.

   iii. The two processes are assumed to share a name/address space. I have used the term tasking to describe this relationship. This can be implemented by multiple execution points associated with a name space or by sharing the execution point of a single process among multiple subtasks. In more restricted versions of tasking, the programmer may be permitted to assume that tasks of a given priority level will never be running simultaneously with a task at a higher priority level. Thus a high priority tasks can assume itself effectively masked against affects of lower priority tasks. I am not sure whether this is a very useful capability considering that it requires one to make major assumptions about one's environment: assumptions that can easily be wrong.

   There is also a form of cooperation between processes that falls between types i and ii. Two processes can share their address space so that segments always have the same segment number in the two proceses but the two processes have different name spaces. In practice, this sharing cannot apply to all segments since their must be some truely per processes segments. Some implementation of this latter model would simplify the sharing of data between two

processes. (It can be especially useful for list structures).

2. Many Multics programs make unnecessary assumptions about the stacked nature of their environment. For example, qedx cannot be called recursively. It detects recursive calls by somehow seeing if there is another invocation of itself active on the stack and types a message to the user informing him and asking him if he wants to abondon the earlier editing. If the other instance is running in a different task, qedx does not seem to detect the case and returning to the first invocation results in an error. The solution to this particular problem is to remove the restrictions against recursion. The only reason that I have been given for the lack of recursion is to permit programs other than qedx to reference string buffers by fixed names. Since the names of the buffers are considered internal information and any operation (such as sorting) can be performed by writing out, processing and rereading the data, this is a poor excuse.

   Other programs with similar problems include edm and archive. (teco is a notable exception). Another problem is one that I have come accross in program my own subsystem wherein there is a need for a per invocation data base that is referenced by a number of programs within the subsystem. If all the procedures were internal to one large procedure the structure can be referenced as global data. If, however, one wishes to achieve the effect of having a global data base for the invocation without explicitly passing a reference on all procedure calls and without the incorporatin of all procedures into one larger one, one can use a static pointer to reference the data. On recursive calls, one saves the static pointer to the previous invocation and sets it to point to the new data structure. Upon returning (or unwinding) the pointer is reset to the previous value. This solution works as long as one can assume that there is only one instance of the subsystem running at a given time and that one must return from a given invocation before resuming an earlier one. When this assumption is not applicable, it is more difficult to achieve the affect of a global data base.

   This is one example where a per-task static storage class would be desirable. One can provide such a storage class by extending the object segment format (and possibly the pl1 language) to permit the use of per task static storage that would be referenced analogously to the linkage section (or Webber's proposed static storage section) but which would use a special pointer in the base of the stack that is different for each stack/task. Thus the use of tasks would make it possible to make effective use of global

data bases that are associated with an invocation of a complete subsystem rather than all instances. This still does not solve the problem of a subsystem called recursively on a single static. The full solution would involve a combination of the method described above for stacked invocations and a per-task pointer to the data.

3. CURR\ 3. Currently there are only two types of processes on Multics: "interactive" and "absentee". Multics and user programs "know" too much about the characteristics of each process type for them to be well adapted for use of slave processes. An interactive job is assumed to have an intelligent human user at a console to read messages that are typed out and to answer questions in the start_up.ec. The initializer assumes that it can write directly on the console of an interactive job. On the other hand, the initializer requires that an absentee job write all its own messages on its output stream, including the messages saying that the process cannot be created. (I would assume that this means that a user is not charged for an unsuccessful interactive login but is charged for an absentee one that fails).

   The solution to this problem is to remove the arbitrary distinction between the two types of processes and permit the creater to specify particular attributes for a process. Also, the initializer should never write on a process' console directly, but should establish a communication segment in the process directory and wakeup would be snet (possibly accompanied by a "kick") to inform the recipient of the availability of the message. If a process cannot be created, the initializer can reply on the stream associated with the process or by returning a code to the creator.

4. If we are to be able to create processes, it is necessary to have the ability to destroy processes. This is currently a serious deficiency in the absentee job system. One must phone the operator to have an absentee job killed. More generally, the standard techique for killing a process is to phone the operator and ask him to do so. This doesn't seem very secure. (On the other hand, is there the possibility of a better scheme for those who do not have consoles handly?)

5. As part of the attempt to model the quit handling action of the listener as the creation of a new process (or task) it is recognized that some aspects of the environment are associated with a task (it seems more reasonable to speak of a task in this context as opposed to a process) rather than with the whole process. The most obvious one is the set of I/O attachments of user_input, user_output and error_output. One can generalize this to say that there should be a table

of attachments that are to be associated with a process and those that are to be associated with a task. This might not be feasible within the implementation restrictions of the current I/O switch (executive?). A compromise is to continue the explicity switching of streams when tasks are switched.

This is, in fact, how Doug Wells' current tasking system works. For example, one can write an exec_com that does a pl1 compilation in its own process:

```
&    Must use attach since we are writing to the
  stream for another process
&attach
&    get a process.
communicate
pl1 &1 -table
*The word quit overprinted in one column returns to
  listener.
*The word exit overprinted in one column destroys
  process.
```

In the above example, when the "quit" line is reached one is returned to the listener in the controlling process but the pl1 process continues to run. In the current implementation, its output messages will be typed on the user's console as they are generated. One may perform other work in one's controlling process. When the compilation is done, one can continue the task that controls the pl1 process and the next line in the exec_com is read. In this case it exits and destroys the slave process. (Note that even if there had been a line in the exec_com after the pl1 line, the quit would have taken effective as soon as it were read without waiting for the compilation to complete). Multiple ec's can be managing processes independently. When a process is resumed, the appropriate exec_com_stream is associated with user_input. (i.e. the one that was associated with it when the task was quitted).

6. Primitives for task coordination must be implemented. These include:

task_block_ — Like ipc_$block. It places the wait information on the queue of pending wakeups. It then scans for interesting wakeups that might have occured. An process that has received its wakeup is placed on a list of runnable tasks. The task with highest priority is then resumed. Resuming a task consists mainly of switching stacks and the I/O attachments mentioned above.

process_kick_ This is like task_block_ except that the current process is placed immediately on the list of runnable processes.

send_wakeup_ This is like the hcs_$wakeup interface except that a kick can be optionally sent. The kick can be specified explicitly or can be encoded in the channel name.

7. As was mentioned earlier, I/O streams can be an effective means of controlling and organizing tasks and processes. There are two aspects to this, the functions of control and the functions of communication. The communication function will be discussed in the next topic.

The control functions can be performed via order calls. For the controlling process the functiors available available include:

stop_process Stop an inferior process. An optional even channel can be specified for reactivating the process. Otherwise the event channel is supplied by the IOSIM. A call to task_block_ is effected in the associated task.

start_process Send a wakeup to the associated process to resume its execution after a stop.

execute_in_process Executes the specified procedure in the associated process. The procedure may be specified by an entry variable or a pathname. The IOSIM will convert the entry variable to or from a character string as required. (Query, within a given Multics installations, assuming the proper ACL's can a an internal procedure be executed in a different process than its parent's? A different task?)

Note that a process is destroyed when its controlling stream is detached. A "hold" disposition is available to allow the slave process to continue even when its parent disappears.

8. The interface between to tasks can be solely via an io_stream. For simplicity I will assume that there is a stream associated with each task. It the name of a task is "a", then its stream might be called "task_stream.a". The other end of the stream must also be attached to a task, we

may call this attachment "task_control.a". Then a task is running, its user_i/o is attached to its stream. For example, if we have five tasks: main, a, b, b1, b2 where main controls a and b and b controls b1 and b2 the following attachement table might exist while b is running:

```
user_i/o          syn        task_stream.b
user_input        syn        user_i/o
user_output       syn        user_i/o
error_output      syn        user_i/o
task_stream.b     task_io    (task_ctrl.b)
task_ctrl.b       task_io    (task_stream.b)
task_stream.b1    task_io    (task_ctrl.b1)
task_ctrl.b1      task_io    (task_stream.b1)
task_stream.b2    task_io    (task_ctrl.b2)
task_ctrl.b2      task_io    (task_stream.b2)
task_stream.a     task_io    (task_ctrl.a)
task_ctrl.a       task_io    (task_stream.a)
task_stream.main  tw_        tty032
```

In this example the console is associated with the main task, though it need not be. When a task attempts to read from its input stream, it automatically goes blocked if there is no input waiting. Thus for example, one can easily implement a system wherein one process simply manages the console and feeds data to another process by writing to its stream. If the console process is higher priority, it can respond immediately requests for echoing. The other process need not be aware of whether it is speaking directly to a console or is going through another process. Other models of console handling can be envisioned, such as requiring that the user explicitly state which process is to be given each line of input. (This is what is currently done for the Multics initializer console).