

Com-plete Resource Usage and Estimates

This chapter gives an overview of how Com-plete uses the various system resources and shows how you can estimate the amount used, based on Com-plete's needs and other factors at your installation. Depending on the type of resource, a shortage of that resource will under normal circumstances be handled by Com-plete. In some cases, resource shortage does not affect the operation of Com-plete; however, in others, it can mean that certain functions cannot be performed until sufficient storage becomes available. This is discussed in more detail at the start of each section.

Note:

The storage estimates given in this chapter refer to the base level of Com-plete 5.1. Future maintenance may cause these estimates to change. Please refer to the relevant documentation updates issued with each maintenance level.

This chapter covers the following topics:

- Virtual and Real Storage
- The Roll Subsystem
- The Com-plete Spool Data Set
- The Com-plete Sequential/Direct Data Set
- The UDEBUG Buffer Pool

Virtual and Real Storage

A number of mechanisms exist in Com-plete for managing storage. Each is described here, together with the various uses they are put to. This will give you an idea of how to utilize your available storage better and help you understand some of the performance issues and integrity problems associated with the storage management under Com-plete.

Real Storage

It is possible for Com-plete to be swappable. This is achieved by setting `applymod 86` at the startup of Com-plete. This can be useful to avoid test systems from having a detrimental effect on the system. Please note that as soon as ACCESS is started, Com-plete is made non-swappable and will remain non-swappable until the job is terminated. This is because Com-plete becomes a Cross Memory Server and to do this, it must be non-swappable. Stopping and starting ACCESS will not affect this, because the Cross Memory environment is never actually shut down when this is done.

Com-plete Savepool Areas

The lowest level of storage used under Com-plete is the Savepool element. These are elements used as saveareas for nucleus modules. For this reason, these must be allocated correctly from the start, because if they run out, depending on which part is affected by the failure, Com-plete is likely to go down immediately or at least in the following time. This is because it is not possible to expand this Com-plete resource as they are at such a basic level.

Savepool areas can be acquired by Com-plete routines with very little overhead. They are of a fixed length and use the same techniques as the Fixed Buffer Pool Manager to get and free the buffers. They are accountable in that usually, if an abend occurs to a task which has acquired a savepool area, the area can be returned to the available pool and will not be lost.

They can be allocated above and below the line based on the SAVEPOOL and SAVEPOOL-ANY sysparms. When it is possible, a savepool area from the pool which exists above the line will be used. When no such savepool is allocated or the allocated pool is empty, the system will allocate a savepool entry from the pool allocated below the line. When this is exhausted, unpredictable errors may occur depending on what point in Com-plete's processing the failure occurs. For this reason, great care must be taken in allocating the savepool areas.

Due to the transient nature of SAVEPOOL usage, it is impossible to estimate the exact amount of SAVEPOOL entries that will be required for a given installation. The safest recommendation that can be given would be that 5 savepool areas be allocated above and below the line for each active Com-plete task in the system (including system tasks).

It is possible to monitor the usage of the savepool areas using the UTIL MO subfunction SP. These should be monitored over the normal span of activity for the installation. For some, this will be every week, for others it may be every month. Once a full set of figures is available, the worst possible figures can be used as a guideline. Of course, an increase in the workload will again change this. It is therefore always wise to leave a little room for movement and monitor the performance of the savepool areas on a weekly basis.

Com-plete Fixed Buffer Pools

The current version of Com-plete uses a fixed length buffer pool mechanism to manage the most commonly used Com-plete buffers. This opens up new scope for 24-hour operation, as well as improving the availability of Com-plete, as fragmentation cannot exist as it can with variable type buffer pools.

This mechanism enables the allocation of buffer subpools below the 16M line, above the 16M line and in ESA Data Spaces.

When a requested size does not exist in a subpool because the size is greater than the largest subpool element size, the logic causes a new buffer subpool to be allocated with the required size to satisfy the request. This facility can be deactivated on a buffer-by-buffer basis: no new buffer subpool is then created and the request rejected.

For each buffer pool allocated, a chain of buffer subpools will exist for that specific buffer pool. Each subpool represents one specific size / location combination. For example, a subpool can exist with size 1k below the line and above the line, satisfying two logically different types of request. Error handling is as described above with expansion of subpools, subject to storage availability provided for all buffer pools.

The COMSTOR Buffer Pool

When the Com-plete COMSTOR function is used, it generally means that many areas of the same size are allocated in COMSTOR storage depending on the application usage profile. As such, it made sense to use a fixed length buffer pool to handle these requests, as the buffer pool could then be tailored to meet the demands of the various COMSTOR area sizes required.

This provides the following advantages:

- COMSTOR no longer becomes fragmented, which previously called for an over allocation of COMSTOR to ensure that requests were not eventually failed.
- COMSTOR can take advantage of the expansion and contraction capability of the fixed buffer pool manager. This means that even where too little storage has been allocated, the storage can still be made available if there is enough space in the Com-plete region. Both of these factors ensure that COMSTOR need never be a consideration and will never make it necessary to stop and restart Com-plete.
- Storage can be located based on the usage of the applications. Some old applications require that the COMSTOR storage areas allocated for them be below the line due to ECBs being located there. This can be handled by allocating a COMSTOR subpool below the line for these applications. Newer applications can then take advantage of COMSTOR areas above the 16M line and even in ESA data spaces where the COMSTOR area is not being used to contain ECBs.

The COMSTOR buffer pool is built using the COMSTOR-BUFFERPOOL sysparm. The size of the element for each subpool defined determines the largest COMSTOR area size which can be provided from that subpool. Of course, should a smaller size be requested and no smaller subpool is available to handle that size, it too can be satisfied. The number of elements for the subpool determines how many COMSTOR areas are to be initially allocated. Should you wish to limit it to that, you must simply specify "0" for the number of elements by which it should expand if necessary. Otherwise, the subpool will be expanded as necessary to cope with the demands placed upon it. However, no new bufferpool is created if you request a larger COMSTOR area size than the biggest one defined in the sysparms

The location of the subpool storage will depend very much on the usage to which the allocated COMSTOR areas will be put. Where an application program has an ECB within the COMSTOR area and it runs in 24 bit mode, the area must be below the 16M line. When the application runs in 31 bit mode, the COMSTOR area can reside above the 16M line. When the application program does not have an ECB in its COMSTOR area, the subpool from where it will be gotten can be allocated in a data space on ESA capable systems.

Note that for the COMSTOR buffer pool, an element size can only exist for one subpool regardless of the variations in the location of the subpool. This is the case simply because the COMSTOR facility will always allocate the COMSTOR area in the most widely available storage subpool. This means that a subpool built in a data space will be used before one built above the line and so on. Therefore, if a second subpool of the same element size were built in a different location, it would probably never be used.

Apart from the COMSTOR subpools which are built for application program usage, COMSTOR also builds a subpool for the control of the COMSTOR facility in general. This is built based on the total number of elements allocated based on the provided COMSTOR-BUFFERPOOL parameters. Where a subpool with similar attributes to that required by COMSTOR is explicitly allocated by the user (i.e., LOC=ANY, ESIZE=72), this subpools element count will simply be increased by the number of entries which COMSTOR expects to use.

Storage Key of Buffer Pool Subpools

A facility has been introduced to enable the allocation of fixed buffer pool manager subpools in a storage protect key other than that used by Com-plete. Effectively this is another distinguishing factor for a subpool, therefore, a number of subpools can now exist within the same buffer pool with the same element size and location, however, each with a different storage protect key.

The default for the majority of Com-plete's fixed buffer pools is to allocate the subpool storage in Com-plete's key, however, the Adabas Interface requires storage which exists outside of the thread in the thread's key, in order to roll a user program out over an Adabas call. For this reason, the Adabas interface builds a buffer pool with a number of storage subpools in different storage keys. Refer to the discussion about the Adabas interface for more information. It is possible that other Com-plete subsystems may build storage subpools with different storage protect keys in the future.

The Com-plete Unit of Work (CUOW)

The primary descriptor for work in the Com-plete system is the Com-plete Unit of Work control block or CUOW. This control block is built in a buffer in the General buffer pool when a user program is started and exists until the user program terminates. The CUOW contains all information related to the user program.

Thread Groups and Sub-Groups

A user program can be catalogued to run in a specific thread group which must be defined at start up in the sysparms. If a program is not allocated or it has no thread group associated with it, it will run in the DEFAULT thread group *if* a thread sub-group exists within the group large enough to run the program.

For each thread group, one area is required for the Thread Group Control Block (TGCB) while one Thread Subgroup Control Block (TSCB) is built for each sub-group. Each thread within the sub-group is described by a Thread Control Block (THCB) as in previous releases of Com-plete.

It should be noted that the above control blocks (i.e. TGCB, TSCBs, and THCBs) are acquired from the General buffer pool. Previously THCBs were linked with the Com-plete nucleus and followed the system THCBs. The system THCBs remain where they always were but thread THCBs are no longer linked into the nucleus nor will they be allocated beside each other. It should also be noted that these changes include provision for the dynamic reconfiguration of the system which will give sites the ability to add or delete threads and/or thread sub-groups. This means that in the future, THCBs may disappear during the lifetime of a Com-plete run.

Task Groups

In some cases in the documentation, tasks and task groups will be referred to as processors or processor groups. You will also notice that UUTIL MO functions and operator commands related to tasks start with the letter 'P'. The reason that the task related control blocks are often referred to as processors in the documentation is a simple one of naming conventions. When function names and operator commands were being created, the T in 'thread' and the T in 'task' frequently caused the same name to be generated. For this reason, any task related function or control block is prefixed with 'P' which stands for processor. Generally speaking, where the term 'processor' is used, it can be substituted with 'task'.

Much like the thread groups, one or more task groups is allocated at Com-plete start-up. If a program has a task group associated with it in its catalogue entry, the program will run in that task group if it exists. If the program is not catalogued, or has no task group associated with it, it will run in the DEFAULT task group.

For each task group, a Task Group Control Block (TGCB) which for each task within the task group, a Task (Processor) Control Block (PRCB) is allocated. These control blocks are chained from the PRCB. Both PRCBs and the PGCB are allocated from the general buffer pool.

It is possible to add and delete tasks while the system is running through the TASKS operator command. It should be noted that for performance reasons, when a PRCB has been allocated for a task and the task is subsequently deleted, the PRCB is not actually freed. This can only occur when the task group itself is deleted which occurs at EOJ. These so called dormant control blocks can subsequently be reused if more tasks are added to a task group at some future point in the life of Com-plete.

Virtual Storage Usage

The following gives an overview of the major virtual storage areas in the Com-plete address space / region. These are the areas to consider, e.g., when planning for the number of threads to allocate. Experience shows, it is still hard to calculate the exact amount of storage that will be used. E.g., it may be difficult to tell how many I/O buffers are allocated for each of the datasets when they are opened, you don't know in advance which bufferpools will expand, etc. Software AG recommends that you do a rough estimate and start with a configuration that leaves 20-30% of your region below the 16M line free. Then, while Com-plete is running and the maximum number of users is active, use USTOR function ASU to determine the real address space utilization. If you then find that there is a good reserve, you can increase the number of threads in one or more sub-groups.

Thread Storage

Typically, the biggest part of all storage in the Com-plete address space is used by the threads. Each thread in a given sub-group occupies the same amount of storage below the line, as specified by sysparm THREAD-GROUP. Each thread, independently of thread group and sub-group, occupies the same amount of storage above the line if specified by sysparm THSIZEABOVE.

Storage Occupied by Load Modules

- Com-plete nucleus modules;
- Com-plete server modules;
- RESIDENTPAGE modules;
- PGMLOOKASIDE modules.

The location of the modules is defined by their RMODE attribute, except for PGMLOOKASIDE programs, which are always loaded above the line where possible.

Terminal Table (TIBTAB) Storage

The TIBTAB can be assembled and linked, from which the size of the TIBTAB can easily be seen, or it can be dynamically generated for which the amount of storage used must be calculated. The size of each TIB is currently 192 bytes. This version of Com-plete allows the TIBTAB to reside above the 16M line, controlled by means of the load module's RMODE attribute or the sysparm TIBTAB=ANYnnnnn.

Savepool Storage

The size of one savepool element is currently 208 bytes.

Storage for Fixed Buffer Pools

The amount of storage used is the actual buffer storage itself, plus some storage for control blocks. Once successfully initialized, Com-plete wherever possible obtains storage already allocated from the buffer pools. Of course, if a buffer pool must be expanded, this storage is again requested from the operating system.

I/O Buffers and Control Blocks

- for Com-plete's own datasets;
- for VSAM and other datasets used by application programs.

Usually, control blocks and buffers for VSAM files are located above the 16M line, those for all other files below the line.

Natural Buffer Pool Manager

The Natural buffer pool manager simply allocates storage as specified in the sysparms. A message is issued indicating how much storage has been used and where it was obtained. Unless forced below the line via a sysparm, the Natural buffer pool storage is obtained above the line.

Control Blocks of the Operating System and Other Software products

- TCBs, RBs, etc.;
- RACF/ACF2 related control blocks, e.g. one ACEE per user;
- etc.

General Buffer Pool Usage

After initialization, apart from the storage requirements that are specifically allocated at startup, all Com-plete requests for storage are resolved from this buffer pool. This includes:

- Short term working storage requests;
- Medium term requests; this storage is held for the duration of a transaction;
- Long term requests.

There are many different types of working storage areas obtained. Size, number, and location of these areas heavily depend on various factors, making it almost impossible to calculate them exactly. Instead, you are recommended to start with a general buffer pool configuration with a basic number of buffers of 64 bytes, 128 bytes, 256 bytes, 512 bytes, 1 Kbyte, 2 Kbytes, 4 Kbytes, and 6 Kbytes, below and above the line, where applicable, for all of these sizes. For the basic numbers of buffers, reasonable values to start with can be either your values from the previous version of Com-plete (if you are upgrading to a higher version), or you can start with a configuration shown in the following example:

```

BUFFERPOOL=(64,100,100,ANY)    (*)
BUFFERPOOL=(64,100,100)
BUFFERPOOL=(128,100,100,ANY)
BUFFERPOOL=(128,100,100)
BUFFERPOOL=(256,100,100,ANY)

```

```

BUFFERPOOL=( 256 ,100 ,100 )
BUFFERPOOL=( 512 ,100 ,100 ,ANY )
BUFFERPOOL=( 512 ,100 ,100 )
BUFFERPOOL=( 1K ,100 ,100 ,ANY )    ( * )
BUFFERPOOL=( 1K ,100 ,100 )        ( * )
BUFFERPOOL=( 2K ,10 ,10 ,ANY )
BUFFERPOOL=( 2K ,10 ,10 )
BUFFERPOOL=( 4K ,10 ,10 ,ANY )
BUFFERPOOL=( 4K ,10 ,10 )
BUFFERPOOL=( 6K ,10 ,10 ,ANY )
BUFFERPOOL=( 6K ,10 ,10 )

```

Since they are related to other sysparms, the allocation parameters for the buffer pools marked with (*) may be changed internally during startup.

If you then monitor the buffer pool statistics, you can easily determine the buffers that should be increased or decreased in size or number.

The Roll Subsystem

While this version of COMPLETE removes the limit of a maximum of 16 threads, the number of threads is still limited by the amount of storage available in the address space. In general, the number of users will exceed the number of threads, so threads have to be shared between users.

When a user program has reached a certain point in its processing, for example a conversational terminal write, it no longer needs to reside in the thread, as the conversational write will take a relatively long period of time to complete. In this case, another user can use the thread. However, the current user's data must be saved somewhere. This section gives a definition of the terms used to describe the Com-plete method of doing this.

Com-plete Rollout/Rollin Processing

Currently Com-plete can save the image in a thread buffer called the "roll buffer". This saving of data is known as a "rollout".

When the user responds to the conversational write, the program copy must be found and moved back to the thread to continue execution. Again the image is copied from the roll buffer back into the thread. This process is referred to as a "rollin".

Com-plete Roll Buffers

In many installations, the sizes of the thread images which are rolled out from thread tend to be very consistent in a production environment. This means that with the provision of a few subpools of the required sizes, a roll buffer pool can be provided which will satisfy the vast majority of rolouts from thread. When a roll buffer pool can be used in this way, there are a number of advantages.

- The allocation and freeing of the buffer is infinitely quicker than for the variable buffer pool which can have a major performance impact where a lot of rolling is taking place.
- No fragmentation takes place, thus providing 24*7 functionality.

- The subpools can reside in a data space, thus freeing up more of the Com-plete region for other storage.

The fixed roll buffer pool is allocated based on the ROLL-BUFFERPOOL sysparm. The element size is the amount of storage available in a given subpool into which the roll subsystem can copy a thread image. This can best be determined by the second roll activities screen (PF11) in UCTRL which provides a map of the sizes of the thread images which are being rolled out. If the load is consistent over a run of Com-plete, this could be used to estimate the most appropriate subpool sizes which should be used for the fixed Roll buffer pool.

The number of elements for a subpool indicate how many thread images the subpool can accept and again, once the number to expand by is not set to 0, it will expand to handle the load. If the subpool is allocated in a data space, this should not pose a major problem. However, if the subpool is allocated in the primary region, due to the element sizes involved, the Com-plete region may fill very quickly.

As the location of these subpools is of no concern to users of the system, they should be allocated in the most available area which is the default. This means that on ESA capable systems, they will be allocated in a data space while on non-ESA capable but XA capable systems, they will be allocated above the 16M line.

Gets and Frees from this buffer pool are extremely efficient using 370 instructions to serialize access to the storage. By definition, it will never be fragmented so that it will still have the same potential to be used after two weeks of operation, as it would have had after being brought up. Lastly, the storage where the image is held can be allocated in a data space on ESA capable machines freeing up a large part of the primary address space for other work.

The Maximum Number of Rolled Out Images.

The maximum number of rolled out images can simply be calculated as follows.

Maximum Logged on Users * (Maximum No. of Stack Levels +1) +

This is a theoretical maximum: some users will not use or will be disallowed from using the maximum number of Stack Levels.

The Com-plete Spool Data Set

The Com-plete spool data set is a VSAM data set and contains the data for all print out spools in the system and all messages that exist on disk.

Data Set Structure

The MAXPO sysparm indicates the maximum number of printouts and/or messages that can exist in the system. Com-plete reserves this number of blocks at the start of the message data set to hold the control information for a printout. The printout data set must contain at least this number of blocks or Com-plete initialization will fail. If more blocks are available than the MAXPO specification, these blocks will be used for printouts whose data cannot fit in the first record along with the control data.

Printout Structure

During initialization, Com-plete interrogates the directory information (contained in the first n blocks of the spool data set) and constructs an in-store Message Core Queue (MCQ) for each printout queued. This MCQ information is supplemented by the information contained in the control record on disk. The control record can point to more than one data block, Com-plete also maintains a list of index blocks which are used for free space management within the spool data set, and possible positioning of the printout via USPOOL commands.

Receiver list

When a printout or message is sent, it is sent to a receiver or a list of receivers. The processing is different depending on whether a list of receivers or a single receiver is specified. When a single receiver is specified, the receiver is remembered by name and therefore the printout or message can be restarted in an environment with a dynamic TIBTAB defined. In the same environment, using lists may result in restart problems, as the list will be remembered using TIB numbers causing unpredictable results with the restart. When the TIBTAB is defined, the restart can normally proceed without problems, as unless the TIBTAB has changed, the various TIB numbers will still relate to the original terminal.

Spool Data Set "Data"

The data written to the spool data set is compressed using repeat to address (RTA) type commands, thus saving space. Lines are written into the data areas until the data area is exhausted, then a new data block is allocated and the next line is started in there. To avoid large searches when positioning printouts, index blocks are built. There will be one index block describing multiple data blocks. The index maintains the page and line number that each data block begins with, along with the data block number. Therefore for large printouts, no time is wasted reading from the start of the data blocks to the end of the data in searching for a specific line number.

Spool Data Set Space Calculations

You must first estimate the maximum number of printouts and messages that you expect to see in a system. Once this is done, estimate the average size of the printouts. For small printouts, the space available in the control record sometimes is sufficient so that no extra data blocks area available. However, for installations who have large "receive" lists and/or large printouts, additional index and data blocks must be allowed for over and above the blocks allocated for the message data MAXPO index blocks.

The Com-plete Sequential/Direct Data Set

The Com-plete sequential/direct (SD) dataset contains application SD files, terminal paging files, and Com-plete online dumps.

Data Set Structure

The space of this data set is split into two sections when it is initialized - one for application SD and paging files and the other one for Com-plete online dumps. The first record of the data set contains central information about the data set and its two sections.

Structure Of SD File Space

The SD file directory starts from the second record. The number of records used for this directory depends on the maximum number of SD files as indicated at initialization time and can be changed only by reinitialization of the data set. Each directory entry is 64 bytes long. Each directory record contains an integer number of entries. When an SD file is added, the entire directory is expanded and an entry is inserted at the appropriate place in ascending order by SD file name and TID number. For paging files, no directory entries exist; all information about paging files is held in-storage only.

The SD file directory records are followed by the space where SD and paging files are built. These records are used as a pool of unique-length blocks. 12 bytes of every block are required for internal use, so the resulting blocksize is VSAM RECORDSIZE - 12.

Every SD file consists of index blocks and data blocks. Blocks for both index and data are allocated dynamically from the pool when new records are written. When an SD file is deleted, all blocks assigned to it are freed and returned to the pool. A paging file is automatically deleted when the creating application program terminates; its blocks are returned to the pool.

The index of an SD file is partitioned into blocks on fullword boundaries. The structure of data blocks can be described as follows:

For Com-plete internal use, 1 byte is added to each SD file record. Depending on their length, records are blocked or split to the blocksize mentioned above.

In case of blocking, an integer number of records is written to each block. The last bytes of each block remain unused, if the blocksize is not an integer multiple of recordsize+1. If recordsize+1 > blocksize, records are split. In this case, each record starts on a block boundary. Thus, the last block of each record may remain partially unused.

Dump Space Size And Structure

The second part of the SD dataset is used for Com-plete online dumps. The first record(s) of it contain(s) a dump directory of 32 entries. When 32 dumps exist and another one has to be written, the oldest existing dump will be deleted and its directory entry and space will be reused.

The dump directory is followed by the dump space. Its size is defined when the SD dataset is initialized using TUSDUTIL. When deciding on the size of this space, you must consider the size of the largest thread as specified in the THREADS= startup option. It is recommended to assign space for 32 dumps of this size. If insufficient space is available to add another online dump, Com-plete will delete the oldest existing dump(s) and reuse this space to write the new one. This results in fewer than 32 dumps being held online.

The UDEBUG Buffer Pool

This buffer pool contains two subpools which in turn can contain three UDEBUG control block types.

- The first is the Com-plete Control Block (DBCCB) which is a control block required by a UDEBUG session which must always be available, that is, not rolled out with the UDEBUG session. It contains elements with a length of 56 bytes from which the Com-plete Control Block

areas and the Global Symbol entries are allocated. The allocation and usage of this pool can be managed using the FB function of UCTRL and the virtual storage required for can be determined using estimates for fixed length buffer pools described in the section **Storage for Fixed Buffer Pools**.

- The second control block is for UDEBUB breakpoints (DBBP). It contains elements with a length of 184 bytes from which the UDEBUB breakpoint areas are allocated. The allocation and usage of this pool can be managed using the FB function of UCTRL and the virtual storage required for can be determined using estimates for fixed length buffer pools described in the section **Storage for Fixed Buffer Pools**.
- The third is for UDEBUB global symbol entries which must be available to all users.

When a user starts a UDEBUB session, a DBCCB is allocated and when the session terminates, either normally or abnormally, the DBCCB is freed. This means that the number of DBCCBs allocated will be in direct proportion to the number of UDEBUB sessions.

Breakpoint buffers are gotten but never actually freed back to the buffer pool, they are logically freed by UDEBUB to make them available for other breakpoint requests. This was necessary to reduce the cost of serialization. It also means that once the breakpoint subpool has expanded, it will never be contracted. The following discussion therefore relates to the logical getting and freeing of breakpoint buffers.

Breakpoint buffers are obtained implicitly and explicitly. An explicit "get" occurs when the user sets a breakpoint successfully. The freeing of this buffer is a little more complex. Generally speaking, the breakpoint is deleted explicitly by the user or when the UDEBUB session terminates. However, this does not mean that the buffer can be freed. The buffer can only be freed when it can be determined that firstly, the breakpoint is inactive, and secondly, that the breakpoint is not activated while it is being deleted. This generally applies to a situation where the TIB being debugged is different to the TIB on which UDEBUB is running. To avoid costly serialization, these DBBP buffers can only be freed by the debugged terminal.

Even then, it is unlikely that the buffer can be freed. For example, if the breakpoint is for a program which resides in the thread, this cannot be deleted until this user is active in thread again. In the case, the buffer can therefore only be freed once this user has become active and the breakpoint actually reset.

The Roll Buffer

To enable the UDEBUB session to access storage from the user program which is being debugged, the debugged user will be forced to roll out over a BP, as opposed to the normal situation, in which the user is only rolled out when someone else needs the thread. Also, the user must be rolled out to the roll buffer, otherwise the storage will not be accessible by the UDEBUB session. Lastly, no compression of the free queue elements in a thread will be done to ensure that the thread layout is identifiable to the user. This means that the full catalog size of the program being debugged will always be rolled out.

Once the user is successfully rolled out to buffer, the system will ensure that this image is never staged out to ensure that it remains available to the debugging user. This means that the rollbuffer size must be chosen carefully to ensure firstly, that the thread of the program being debugged can always be rolled into the buffer, and that the impact of doing this does not impact the general performance of the Com-plete system.

CPU Usage

As UDEBUB is a debugging tool, the main purpose is to provide functionality with CPU usage a secondary consideration, as this functionality will only be used when testing programs. For this reason, the usage of UDEBUB in a system, particularly through the setting of breakpoints, can significantly affect the CPU usage within Com-plete, even for users who are not debugging or being debugged. Therefore the use of UDEBUB in a production environment must be strictly controlled.

This chapter gives an overview of how Com-plete uses the various system resources and shows how you can estimate the amount used, based on Com-plete's needs and other factors at your installation. Depending on the type of resource, a shortage of that resource will under normal circumstances be handled by Com-plete. In some cases, resource shortage does not affect the operation of Com-plete; however, in others, it can mean that certain functions cannot be performed until sufficient storage becomes available. This is discussed in more detail at the start of each section.

Note:

The storage estimates given in this section refer to the base level of Com-plete 6.1. Future maintenance may cause these estimates to change. Please refer to the relevant documentation updates issued with each maintenance level.