Bell Laboratories

subject: UNIX System Call Measurements

date: September 16, 1976

from: C. D. Perez
T. M. Raleigh
MF-76-8234-079

YY-8234-4

MEMORANDUM FOR FILE

Introduction

This memorandum is the first of a series which will deal with fundamental measurements of the UNIX operating system on the PDP-11 line of computers. A description is given of system calls and some of their basic measurements.

Overview

The purpose of the set of measurements described in this memorandum is to explore in detail the overhead incurred in making a system call under UNIX and to obtain some information on the raw processing speed of the current PDP 11 line of processors (11/40, 11/45 and 11/70).

Since the time to execute various system calls is on the order of tens or hundreds of microseconds, a facility for measurement that had a high resolution was needed to obtain the data we desired. A trace tool designed by T.M. Raleigh provided the ability to record time stamped events in a manner similar to the trace facilities available on larger time sharing systems. When used with the KW11-P programmable clock, time stamps accurate to 10 microseconds were obtained. In addition, a data reduction program called fsm was written to reduce the data and produce formatted listings and statistical information.

Using the trace as our basic measurement tool, an environment was created on each of the processors studied that attempted to minimize the number of perturbations of measurements. Unibus interference was the chief problem and this could only be minimized, not eliminated, because of the architecture of the PDP-11.

To conduct the experiment, several system calls were selected which do not require any scarce resources and hence do not need to roadblock until the resource is available. This eliminated any hardware configuration problems such as

disk rotational speed and seek time, number of processes sharing a resource, etc. System calls whose execution time is a function of these variables will be studied subsequently. The calls chosen were essentially CPU bound, and were expected to give similar results across the three machines studied. Prior to the experiment, the trace facility was calibrated by determining how long it took to record a time stamped event and this information was available to correct the data.

Besides information on relative processor speeds, we expected to obtain a measure of the minimum system call time, the amount of overhead for each argument that must be passed as part of a system call, the amount of overhead incurred in context switching as results of the measurements. It is the relative costs of time that constitute the important information gleaned from this experiment and not the absolute execution times for each system call. As new features are added to the system or existing ones are enhanced, these times may change slightly.

Measurement Tools

A trace facility has been designed and implemented for the UNIX operating system by T.M.Raleigh of the UNIX Support Group (USG). This tool was used for the measurements that are described in this memorandum. The steps involved in the insertion of the trace into the operating system are supplied as Attachment A. Documentation of the trace tool is provided as Attachment B; a brief overview is given here.

The trace software is capable of recording time stamped events that a user embeds either in his program or within the operating system. The programmable clock is used at a 100KHz rate so that time stamps with 10 usec. granularity can be recorded. Internal buffers (three were used in this experiment) collect data. When buffers are filled they are written onto an output device of the user's choice. Communication with trace both to activate it and to record data is done by means of the event subroutine call. By judiciously placing calls to event around specific activities, the time to accomplish them (i.e. the time between event calls) can be determined. Besides being a subroutine within the operating system, event is also a system call so that accurate measurements of user program activities can also be obtained.

Fsm is an interactive reduction program for digesting the raw data collected by the trace mechanism. Fsm can be used in an editor mode to examine and print the data or it can be used to build finite state machine models to perform statistical analysis on the raw data. Trace is a relatively new tool; this is the first use of the fsm program. The experiment afforded an opportunity to evaluate their behavior,

refining or debugging as the needs of each experiment directed it. Initial tests were made on each computer to arrive at a very accurate measurement of the time to execute an event system call both from a user program and a system module. System probes (calls to event) were applied at many points until every expenditure of time around the original calls could be explained.

Test Systems

Each of the tests in this experiment was run on three different PDP-11 computers. In summary, the test configurations were,

1. Dept. 8234's PDP-11/45 with 80K words of memory, a KW11-P clock, an RP03 moving head disk containing the root file system, an RF11 fixed head disk used for swapping and an RK05 moving head disk which was used to record the trace data.

2. Dept. 8234's PDP-11/70 with 128K words of memory, a KW11-P clock, an RS04 fixed head disk for a root file system, an RP04 moving head disk which contained the user file systems and a TU16 magnetic tape drive for recording trace data.

3. Dept. 1481's PDP-11/40 with 40K of memory, a KW11-P clock with the root file system on an RK03 moving head disk and the swap area on an RF11 fixed head disk. Data was collected on a TM11 magnetic tape.

Although both the 11/70 and the 11/45 are equipped with floating point hardware, tests were made on each system configured with and without that feature to define the additional overhead in context switching. The three computers will be used for subsequent tests with variations in configuration only as a particular experiment requires it.

Programmable Clock Problems

The trace mechanism uses the programmable clock to produce event entries with 10 usec. resolution. Before any data collection was done, the clocks (KW11-P) on each machine were checked for accuracy against external time sources (time of day) and against line frequency clocks (KW11-L) on nearby systems. For example, the KW11-P clock on the 11/45 system was checked against the line frequency clock of the 11/70 which in turn had been calibrated against the time of day. Our calibration procedure showed that the KW11-P clock sold by DEC had some serious design flaws which caused large time drifts when the programmable mode was used. These problems were caused by deficiencies in the oscillator circuitry and by a race condition that occurs in the interrupt request circuitry on the KW11-P board. This

was a known problem by DEC engineers and redesigned clocks were obtained for our measurements. The 11/40 system used for these experiments had no provision for a programmable clock, so it was outfitted to accept that device temporarily.

Environment

In order to minimize any perturbation in the data being recorded, a quiet system was constructed. We define a quiet system to be a single user system where care has been exercised to insure that there are no daemons or other processes running and that there is no unessential device in the system generating interrupts. The object was to minimize Unibus activity. Because of the architecture of the PDP-11 processor, speed can be influenced by activity on the Unibus. Figure 1 illustrates the problems which a Unibus architecture poses. Here it is shown that the entire system is organized around the Unibus with a bus arbitration unit determining when a device is allowed to use a cycle on the Unibus. The Unibus is allocated to a device for only one cycle (for simplicity assume a cycle to be 1 usec.) in order to complete a transfer. The result is that all of the devices on the bus (including the processor) compete for Unibus cycles; therefore the activity of one device may delay the operation of another. The chief source of contention is between the processor and devices to obtain cycles on the Unibus to access memory. As an example, consider the processor and a disk on the Unibus. The processor needs Unibus cycles to fetch instructions and operate from memory while the disk needs Unibus cycles to transfer data to or from memory. The processor and disk, however, request these cycles at different rates. The disk requires Unibus cycles only at the rate at which it transfers them to or from the disk surface. (This is a slight simplification since disks have a silo for buffering data in transit from the bus to the disk surface; when the silo is not full, it can make requests at the same speed as the processor.) If the device is, for example, an RK05 disk, the transfer speed is 11.1 usec. and if we assume that the processor makes memory requests at 1 usec. intervals, then it can be seen that on the average every 11 cycles the disk will steal a Unibus cycle. Thus, in terms of the amount of time it would take the processor to execute a given set of instructions, the time will be lengthened by 1/11.5 or about 8.7% whenever the disk is making a transfer. (During the time that any positioning operations are in progress, there are no disk transfers being made.)

Thus, any DMA I/O that occurs will lengthen the CPU time to execute a given set of instructions. This experiment included only system calls not involving any I/O; however, the trace facility outputs trace data onto a device when its internal buffers are full. Therefore, the very

fact that measurement data is being recorded results in the data being perturbed. Fortunately, at most one I/O operation is done for every 63 trace points that are recorded. A trace buffer containing 63 trace points is 512 bytes (256 words) long so that it is possible to estimate the effects of this interference. The effects differ with the speed of each device as will be illustrated by using the RK05 and RP04 disk drives as examples. The interval over which Unibus interference occurs for each device is the amount of time that it takes to transfer 256 words.

RK05 - 256*11.1 = 2.840 ms
RP04 - 256*2.5 = .6400 ms.

(This is not strictly correct because, as mentioned earlier, when the silo is empty, a faster transfer rate occurs until the silo is full.) It can be seen that for a fixed number of probes recorded over an interval, there will be a large number of probes with small perturbation when recording on RK05's, while there will be a small number of probes with large perturbation when recording on RP04's. It should be emphasized that Unibus interference occurs only while I/O is occurring and that I/O only occurs for a small fraction of the time that the trace is active. By recording many iterations of the same measurement in one test, the average value recorded will be close to the true value.

In addition to the perturbations introduced by cycle stealing on the Unibus, there are also perturbations introduced by the system software. In particular, the basic timing of the operating system (supplied by the system clock) causes the system to look for other processes to run at the end of each time quantum. Any measurements must take into account the fact that the operating system may have intervened to perform some of its housekeeping functions in between the recording of two trace points. These interventions are however more easily extracted by requiring that trace points be added to system software to indicate where the system has intervened. In this way, data which has been perturbed because of the normal operation of the system can be removed from consideration. For the tests comprising this experiment (with the exception of the tests on the event call itself), two probes (events) were placed within the operating system. One was added to the clock interrrupt handler which supplies all of the system's timing; the other was inserted in the context switching function within the system. Other probes could have been added to account for I/O from the user terminal and I/O from the trace process itself, however, these were considered to occur at a low enough frequency for the idea to be discarded.

The measurement results discussed later in this memorandum have had the data for interference from the system software culled, however, no attempt has been made to

correct the data for Unibus interference. This was done because we were reluctant to transform the raw data and because each test recorded a large number of repetitions of the same data so that perturbations were smoothed out in the results. We have, however, presented two sets of numbers for the system call times that are summarized below. The first is the result of examining the data by hand and represents an effort to obtain median values. The second numbers are the mean value and standard deviation supplied by the reduction program after the data had been corrected.

System Calls

Before presenting the details of the experiment, some background information on the composition of system calls will be given. UNIX uses the TRAP instruction (SYS pseudo-operation in assembly language) to allow programs to make system calls. The basic form of a system call in assembly language is of the form shown for the read system call.

```
read = 24.
    mov  $arg1,r0
    sys  read
arg2:   ;
arg3:   ;
```

where the sys read is assembled as the TRAP instruction with a specific System Entry Point Table number (24.). The read system call passes three arguments to the system. It should be noted that one of the arguments is passed in register r0 while space for the other two arguments is assembled directly after the TRAP instruction. The operating system adjusts the program counter after the system call is satisfied so that the user program begins executing at the first instruction after the last argument. The format described above is used for making a direct system call. The direct system call, however, does not allow the user to write reentrant programs because the number of arguments passed to the system is usually variable and therefore must be set up just before the call. To solve this problem there is an indirect system call which has the following format,

```
indirect = 0.
.text
    mov  $arg1,r0
    sys  indirect
    rd;

.data
read = 24.
rd:
    sys  read
arg2:   ;
arg3:   ;
```

Note that the indirect form uses a copy of the direct form which is assembled in the data area of the program. This means that there will be more overhead in executing an indirect system call, however, this is overshadowed by the fact that one can write reentrant programs using the indirect system format. All of the C library interfaces to system calls are reentrant. C uses the indirect system call for system calls that require more than one argument (read, write, etc.) and the direct system call for system calls that require zero or one argument (getuid, getgid, etc.). From the above description, it can be seen that, aside from the amount of system software that must be executed to satisfy a system call, there will also be differences between system calls based on the number of arguments that must be passed. The way that the operating system handles system calls will be described in order to show what measurements are necessary.

Executing the TRAP instruction causes a trap into the operating system (Kernel space) which saves the user's registers. (Although there are two sets of general purpose registers on 11/45 and 11/70 processors, UNIX uses only one set to be compatible with 11/40's. As a consequence the registers must be saved on every system call.) If a processor has floating point hardware, then the floating point registers must be saved on every system call from a C program. The next step in making a sytem call is to determine whether a direct or indirect system call is being made. For indirect system calls, the address of the real system call template must be fetched from the user's address space. Once all of the arguments to the system call have been copied into the operating system, the function within the operating system that satisfies the system call can be invoked. Finally, the registers are restored and any return value or error indication is passed to the user. The floating point registers can also be restored. Note that in systems with floating point hardware, the floating point registers are saved and restored whether or not the user is executing program using that feature.

The Experiment

I. The first set of tests attempted to determine the amount of time to make a direct system call and how much additional time it takes to make an indirect system call. To measure these differences, dummy system functions were created and appropriate C library interfaces were written for them. These interfaces differed in the number of arguments to be passed to the system making it possible to test a direct system call as well as indirect calls of up to four arguments.

II. The second series of measurements sought to validate the data collected about basic system call times. The

following system calls were chosen because they require no other resources than the CPU to be executed.

dup	time
getgid	times
getpid	alarm
getuid	signal
csw	setuid
kill	setgid
nice	stime

The gtty system call was also tested even though it was dependent on the speed of the line interface to which the character device is attached. The execution time for each of one hundred iterations was recorded per system call with the exception of the dup call which was tested in groups of twelve. A system limitation allows a process to have a maximum of 15 files open at any one time. Since there were already three files open, the recording program was forced to close twelve files before continuing.

Several of the system calls required super-user permission in order to be executed; they were tested apart in a program that had super-user permissions. Each system call was tested on systems configured for floating point and non floating point processors.

III. A third measurement was made to determine how long it takes to make a subroutine call. Because the subroutine save/restore sequence represents a small enough amount of code a hand computation of the expected values could be made and compared with measured values. Since the granularity of the recording time is only 10 usec. which is of the same order as the length of time for the save/restore sequence, the measurement was done in two ways. A single call to an empty subroutine in a user program was enclosed between event calls; ten repetitions of this composed the first measurement. For a second sample, one hundred consecutive calls to the empty subroutine were enclosed in a single set of event calls. One hundred iterations of each of these methods composed a complete test.

Analysis

The first system call to be examined by trace was the event call itself. Its cost in time was vital as a correction factor for all other measurements. Because event may be inserted in user programs and system code, it was measured in each of these ways. The results of the tests appear in Figure 2, and are comparable to the data from subsequent measurements for a system call passing two arguments.

From the material available prior to test execution, it

was expected that the overall performance of the 11/70 would be approximately twice as fast as that of the 11/45. It was not known how far the 11/40 would lag behind the 11/45, although it was known that the Memory Management Unit of the 11/40 was slower than the same unit on the 11/45. The experiment produced evidence that the 11/70 performed 47% faster than the 11/45 (i.e. the 11/70 processor is about twice as fast as the 11/45) while the 11/40 is 29% slower than the 11/45.

By means of the tests repeated on both the floating point and non-floating point configurations of the same machines, it was determined that a saving of 140 usec. in context saving time results on the 11/70 (per system call), while a saving of 170 usec. is experienced on the 11/45, when these machines are configured without floating point hardware.

The results of experiment I appear as Figure 3 of this memorandum. As expected, the data displays a constant increase of time with the passing of each added argument. That increment may be approximated as 95 usec., 70usec., and 30 usec. for the 11/40, 11/45 and 11/70 respectively. Because the latter two figures represent systems with floating point, corrections to a non-floating point environment appear in parenthesis in Figure 3, so that comparisons can be readily made.

The result of the basic system call tests can be found in Figures 4 and 5 for floating point and non-floating point processors. It is possible to conclude that, as a rule of thumb, a direct system call takes an minimum of 520, 450, and 220 usec. on 11/40, 11/45, 11/70 processors respectively. For floating point hardware, that figure increases by 170 (11/45) or 140 (11/70) usec., and an additional 95 (11/40), 70 (11/45), and 30 (11/70) usec. for each additional argument passed in the system call. From Figure 3, it can be estimated that there is a noticeable difference between a direct and an indirect system call when only one argument (in r0) is passed.

As a calibration test, the time to save and restore registers in a C subroutine call was determined by adding the instruction times of each of the lines of code found in the subroutine "csv" and "cret" and the amount of time to invoke them. From such a calculation one might expect the save-restore sequence to take 17, 35, and 37 usec. on the 11/70, 11/45, and 11/40 processors. The experiment performed to verify this expectation provided results that are remarkably close. That data is presented as Figure 6.

Conclusion

Basic system call measurements were taken of the UNIX Operating System in this experiment. The result of the effort was to lay a foundation. The absolute values of the measurements are not as useful as the relative value of the same measurement on the three processors, or as the measures of the floating point save and restore or argument fetching. We expect that measurements made of more complex system calls that have dependencies or other variables can be verified by having a prior knowledge of these basic timings. It is expected that this data would not vary greatly with subsequent software changes. The information will provide for useful comparisons with new DEC processors.

The trace tool is available for those interested in doing timing studies of their own. The UNIX user community will be kept informed of the results of measurements efforts, and are invited to submit ideas or requests for future tests. We are grateful to D. J. Sidor of Dept. 1481 for the permission to use the 11/40, and for his cooperation during the testing period.

C. D. Perez

C.D. Perez

T M Raleigh

T.M. Raleigh

MH-8234-CDP-nroff

Copy (with att.) to
Dept. 8234 Members
UNOS
G. L. Baldwin
J. W. Bowra
B. N. Dickman
A. S. Hall
H. S. London
D. M. Ritchie
D. J. Sidor
K.Thompson

Attachment A

System Modifications

The following source modules must be changed or added to a UNIX system to install the trace facility.

- trace.c - Its object module must be added in a load to build the UNIX.
- event.s - This is the library interface for the event system call and must be incorporated into 'libc.a' as the C library interface to event.
- sysent.c - An entry point for the event system call must be added to the System Entry Point Table; it has arbitrarily been chosen to be 63.
- main.c - requires a single line addition to allow the trace process to be spawned at the appropriate time.
- clock.c - requires minor additions of code to allow for selecting the programmable clock instead of the line frequency clock.
- trace.h - Similar to other system headers, it contains global definitions used by the trace modules. It is also a convenient place to define any software probes that are to be placed in system routines.

evproc

CALL
evproc()

RETURNS
None.

SYNOPSIS
During UNIX initialization the event process is created, readied, and placed in a dormant state until summoned for dumping data onto an output device.

DESCRIPTION

A call to trace.c/einit is made for buffer initialization. In addition to the version number and memory size normally printed when UNIX is booted, the message "hi" appears on the console indicating that a trace system is operative. Evproc then becomes a continuously running process locked in memory. Its function is to determine when a buffer is full, and to initiate I/O on that buffer. The following steps describe the operation of evproc. The process sleeps until the trace is turned on by the user. It then lies dormant until a buffer is filled and should be written onto the output device, at which time evproc is awakened and a call is made to trace.c/evwrite to start the output from the full buffer. After a buffer is written, the pointer "e_iop" used during writing is moved to the next unit in the ring of output buffers. If a write error occurs while dumping a buffer, then trace.c/evproc terminates the trace by calling trace.c/evsig. Evproc repeats these steps indefinitely until the trace is turned off, at which time it roadblocks until the trace is reactivated.

einit

CALL
einit()

RETURNS
None.

SYNOPSIS
Initialize the specified number of buffers for internal collection of event trace data.

DESCRIPTION

The number of buffers specified by NEBUF (usually 2 or more), are circularly linked with each buffer's status

marked as free (E_FREE). "Bufp" designates the buffer for depositing event data; "e_iop" designates which buffer will be the next one dumped. Both pointers are initially aimed at the first buffer. Record size is set at 512.

evwrite

CALL
evwrite(addr)
struct eb *addr;

RETURNS
None.

SYNOPSIS
A buffer of event trace data is written onto the output device.

DESCRIPTION

Whenever the trace is opened for writing onto the beginning of a file, the offset pointer to the output device is set to the head of the file. A non-zero block number is passed along with the buffer address to trace.c/evphys. If that call returned a non-zero value, the event signal is set to deactivate the trace, and sets the status to indicate an error. In every case the event count in the buffer being written is reset to 0, the output device block pointer is incremented, the number of missed events is reset to 0, and the buffer just written is marked free.

evphys

CALL

evphys(blkno,coreaddr)

RETURNS
Contents of "b_flags".

SYNOPSIS

Starts I/O on a buffer of trace data and forces the trace process to wait until the I/O is completed.

DESCRIPTION

After raising the processor's priority to 6, the value of b_flags is set to indicate that the buffer header is busy and is being used to write data. The first argument to evphys is the logical block number on the output device, and the second is the address of the buffer to be written. The major and minor device number of the output device are then

inserted in the buffer header. The device strategy routine is called to queue the request and the trace process roadblocks until I/O is completed. At that time the processor's priority is reset to zero and if any errors occurred during I/O, a nonzero value is returned by evphys.

evsig

CALL
evsig()

RETURNS
None.

SYNOPSIS
Transmits termination request to the trace process.

DESCRIPTION

Evsig dumps partially filled buffers and performs any other reinitialization that is necessary when the trace is terminated. Its first step is to insure that no more events can be recorded, by taking the trace process out of the running state (e_state is set to EREADY or ECLOSE). For any partially full buffers that have not been written onto the output device, a call is made to trace.c/evwrite to start I/O on the buffer. When all buffers are written out, the event signal, "e_sig" is reset to zero. If the e_req flag indicates that the output device should be closed (when the trace is stopped), the device close routine is called and the e_req flag is reset to zero.

evcom

CALL
evcom()

RETURNS
None.

SYNOPSIS
Implements the user interface to the event trace.

DESCRIPTION

The event system call within a user process controls the operation of the trace facility. If the first of two arguments is outside the valid range of 1-127, a system error is posted (EAGAIN) and the system call returns an error indication to the user. An argument of "1" is reserved for communicating contol information to the trace process. In such a case, the second argument may indicate the following actions:

open - activate the trace facility;
continue - reactivate the trace without seeking back to the beginning of the output file;
close - deactivate the trace without seeking to the beginning of the output file;
end - terminate the trace;
stat - provide the current status of the trace process.

Checks are made to determine that 1) the process making the event call is either the owner of the trace process or super-user; 2) the call is not an "open" or "continue" request if the trace is already active; 3) an attempt is not made to close the trace if it is already closed. For any of these situations a system error is posted (EAGAIN) and an error indication is returned to the user. The "open" request results in initializing the event status flag to 0, setting the request flag to cause a seek to the beginning of the output file, and providing that future output will overwrite any data already written on the output device. It also initializes the pointer to the first available position in the output buffer for receiving data. For the "continue" option, the request indicator allows for future output to be appended to the existing output and the same process information is copied. A "close" request sets the request flag to indicate that future output will be appended to existing data and the event signal is set to "quit". For an "end" request, the event request flag is set to indicate a "close" with terminate, the "kill" signal is placed in "e_sig", and the trace process is awakened in order to deactivate the trace facility. For a "stat" argument, the value found in e_stat is returned to the controlling process. For "open", "close", "continue" or "end" a call is made to trace.c/evrun to set the process status.

When the first argument is not "1", a user event is to be recorded. A call is made to trace.c/event, passing the two arguments supplied by the user. The first argument is modified by adding 128 to the event number, so that during subsequent analysis , the source of the event call can be determined as a user program or the operating system.

event

CALL
```
event(nos,arg)
int nos;
int arg;
```

RETURNS
0 if the trace is turned off, 1 if successful recording takes place, otherwise -1.

SYNOPSIS
Records an event in a buffer.

DESCRIPTION

If the trace process is not ready to collect data, a return is made. The processor's priority is saved before raising it to level 6. This is done to lock out interrupts, thereby preventing an interrupt handler from attempting to record an event in the midst of another. Pointers are set to the correct buffer and offset for the next record to be written. If the buffer is found to be full,

a) a flag (E_SIO) is set to indicate that the buffer should be dumped;
b) the trace process is awakened by simulating the slp.c/wakeup function;
c) the buffer pointer is moved to the next buffer.

If that buffer is free, pointers for writing are repositioned and pre-writing initialization is repeated; otherwise this attempt to write a record is marked, the former priority is replaced and a return occurs.

When an unfilled buffer is located, a time stamp is constructed from the system clock, and identifying arguments are written into the record along with the process id of the process that requested the event to be recorded, or the process that was running when the event was recorded. A counter of recorded events is incremented, a pointer is positioned to the next available recording space in the buffer and the processor's priority is returned to its former value.

evrun

CALL
evrun (e_evprocp)

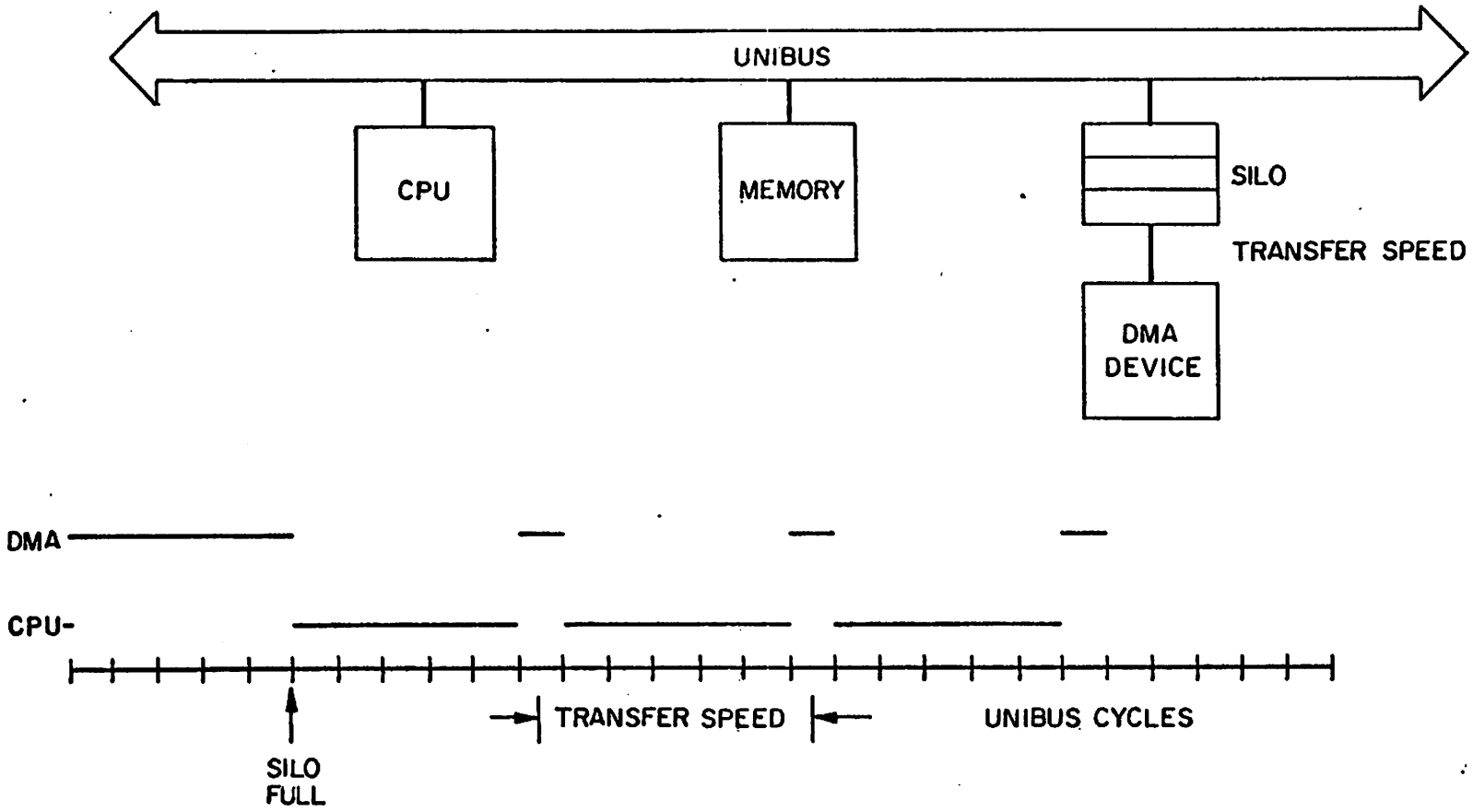
RETURNS
None.

SYNOPSIS

Makes the trace process ready to run.

DESCRIPTION

A wakeup is simulated by setting the event that the trace process is roadblocked on to zero and changing the state of the trace process (p_stat) to SRUN (ready).



CYCLE STEALING BY UNIBUS DMA DEVICES

Figure 1

	11/40		11/45		11/70	
	sys	user	sys	user	sys	user
nfpp	220	1340	150	870	77	420
fpp			150	1040	77	560

Figure 2

Event Call Timing (usec.)

	11/40			11/45			11/70		
subr	Med	Mean	SD	Med	Mean	SD	Med	Mean	SD
D(r0+0args)	520	549	153	615(445)	616	5	360(220)	367	43
I(r0+1arg)	790	812	52	800(630)	861	161	420(280)	436	70
I(r0+2args)	890	909	53	870(700)	904	84	470(330)	481	13
I(r0+3args)	980	1005	52	940(770)	1005	169	500(360)	523	21
I(r0+4args)	1080	1106	60	1005(835)	1069	169	520(380)	524	16

Figure 3

Direct vs. Indirect System Call Timing (usec.)

System	11/40			11/45			11/70		
subr	Med	Mean	SD	Med	Mean	SD	Med	Mean	SD
alarm	540	582	157	480	488	34	220	244	82
csw	510	521	29	460	477	115	220	216	8
dup	790	838	60	680	679	39	320	323	22
ggid	520	545	45	460	495	156	220	228	53
gpid	520	537	44	460	471	43	230	236	52
guid	530	564	155	460	465	25	220	241	77
gtty(s)	1310	1381	227	1010	1049	177	560	578	55
gtty(m)	1310	1366	196	1200	1209	58	600	594	32
kill	880	942	219	720	725	33	380	379	12
nice	550	567	44	480	500	112	240	249	61
time	560	584	49	490	506	118	230	242	53
tmes	1320	1346	66	1020	1028	47	520	527	15
sig	970	993	55	770	779	41	390	396	56
stme	800	849	97	530	554	74	290	305	54
sgid	810	918	246	540	580	157	270	291	91
suid	720	764	196	480	493	53	260	269	64

(s) - single user
(m) - multi-user

Figure 4

System Call Timing (usec.) - nofpp

System	11/40			11/45			11/70		
subr	Med	Mean	SD	Med	Mean	SD	Med	Mean	SD
alarm				630	640	31	350	363	17
getcsw				610	620	38	310	318	14
dup				830	826	37	460	455	26
ggid				620	653	171	350	357	53
gpid				620	636	121	330	345	56
guid				620	643	122	340	350	56
gtty(s)				1180	1177	32	730	732	17
gtty(m)				1390	1414	146	720	717	45
kill				870	896	110	490	516	34
nice				640	651	108	350	374	82
time				650	655	43	350	362	56
tmes				1180	1197	138	660	653	58
sig				930	972	207	500	507	18
stme				700	715	116	420	424	13
sgid				700	720	109	430	435	61
suid				650	660	118	380	386	16

(s) - single user
(m) - multi-user

Figure 5

System Call Timing - fpp

11/40			11/45			11/70		
Med	Mean	SD	Med	Mean	SD	Med	Mean	SD
64	64	2	37	37	1	18	18	5

Figure 6

Subroutine Overhead Timing (usec.)