VENIX Kernel Response Time

Technical Memorandum
VenturCom, Inc.

June 1984

Kernel Performance Measurements

We have run several tests which we use to estimate several critical kernel performance parameters. The results are present below.

The tests were small 'C' language programs which exercised certain sequences of system calls. The source listings for the tests are included as an appendix to this memorandum.

System Call Overhead

The first test ("simsyscall") determines the intrinsic overhead in executing a system call, by measuring the rate at which a user process can enter and leave the kernel. It is essentially the same as the test run by David Fiedler (in the UNIX newsletter Unique) which is presented in the last page of the VenturCom document "A Series of Benchmark Tests".

The results we obtained for the "simsyscall" test are:

Processor	System Call Overhead
80186 (8 MHz)	0.36 ms
8088 (4.77 MHz)	0.82 ms
LSI 11/23	1.10 ms

Average Interrupt Latency

The system call overhead shown above is useful in calculating an average interrupt latency. We estimate that one half of the system call overhead on the 8088 and 80186 is the time taken inside the kernel to set up and dispatch the system call, while the other half is the time taken on the two stack switches and register save/restore sequences per call. The average interrupt latency is the time taken to switch from user mode to a kernel interrupt routine. Since this involves only a single stack switch and register save/restore sequence, it should represent approximately one fourth the total system call overhead for these two processors. This leads to:

Processor	Average Interrupt Latency
80186 (8 MHz)	0.1 ms
8088 (4.77 MHz)	0.2 ms

On the 11/23, the system call overhead must be interpreted somewhat differently, so the calculation to produce interrupt latency is not so straightforward. We estimate about 0.1 ms average interrupt latency for that processor.

Latency may be worse for interrupts which occur while the processor is executing certain critical code regions in the kernel where interrupts are locked out.

Signalling Time

The next test ("simsig") uses a program which sends a signal to itself, catches the signal, resets the signal catch mechanism, and returns. For each loop, two explicit system calls (kill() and signal()) and one effective call (the signal catch) take place. The times per loop of this test are:

Processor	Signal Time	Kernel Processing
80186 (8 MHz)	1.2 ms	.2 ms
8088 (4.77 MHz)	3.1 ms	.6 ms
LSI 11/23	4.6 ms	.7 ms

The "kernel processing" column indicates the time per loop actually spent by the kernel processing the system call. It is calculated by taking the "signal time" and deducting the system call overhead (as calculated above) for the three system calls per loop.

Context Switching Time

The last test measures the same sequence as the previous test, but the signal is sent between two processes. This forces context switching. The results of this test are:

Processor	Loop Speed	Context Switch
80186 (8 MHz)	5.4 ms	1.5 ms
8088 (4.77 MHz)	16.4 ms	5.1 ms
LSI 11/23	13.5 ms	2.2 ms

On every loop, each process runs once and signals the other: there are two signals, and two context switches. Thus the overhead of the two context switches may be estimated by taking the "loop speed" and deducting twice the previously calculated "signalling time". This number is divided by two to determine the time of a single context switch, which is shown in the table above.

These results can be used to estimate the interval between the end of the interrupt and the beginning of a new process. For example, on the 8 MHz 80186, the following values would be added:

.1 ms	(interrupt latency)
??	(device-dependent interrupt handler)
1.5 ms	(context switch time)
.1 ms	(return to new process)
??	(effects of 8087 floating point coprocessor)

Note: the 8088 and 80186 context switch times were measured on units without the floating point coprocessor, and so do not reflect the time needed to save/restore that chip's registers or the latency waiting for it to complete an operation. The presence of an 8087 produces a measurable but not major change in the times. The 11/23 benchmarks do include floating point coprocessor times.

```
/*
 * simsyscall   - Calculate the time to do a ''getuid'' system call.
 *
 *      This calculates the time to do a simple system call without
 *      any arguments and without any kernel processing.  It reflects
 *      the intrinsic overhead to do a system call; time to switch from
 *      'user' to 'kernel' context and dispatch the call.
 */
#define NLOOP   10000L          /* no. of iterations for good statistics */

main(){
        register unsigned int i = NLOOP;

        while( i-- )
                getuid();

        display("'getuid' system call",NLOOP);
}


/*
 * simsig       - Send and catch a signal to oneself.
 *
 *
 */
#define NLOOP   10000L          /* no. of iterations for good statistics */

long loop = NLOOP;

func(){

        signal(1,func);
}

main(){
        register int pid;

        pid = getpid();
        signal(1,func);
        while( loop-- )
                kill(pid,1);

        display("simple kill-catch-signal",NLOOP);
}
```

```
/*
 * twosig       - Send and catch a signal between two processes.
 *
 *
 */
#define NLOOP   2000L           /* no. of iterations for good statistics */

int pid;
int loop = NLOOP;

func(){

        signal(1,func);
        if( loop-- )
                kill(pid,1);
}

main(){
        register int i;

        pid = getpid();
        signal(1,func);
        if( (i = fork()) == 0 ){
                while( loop )
                        pause();
                exit(0);
        }
        pid = i;
        kill(pid,1);
        while( loop )
                pause();
        wait(0);

        display("two process catch-signal-kill loop",NLOOP);
}
```

```
/*
 * display      - printf the average time to do a 'loop'.
 */
#define HZ      60L

struct tbuffer {
        long    p_u_time;
        long    p_s_time;
        long    c_u_time;
        long    c_s_time;
} buf;

display(msg,loop)
char *msg;
long loop;
{
        long total, l;

        times(&buf);
        total = buf.p_u_time + buf.p_s_time + buf.c_u_time + buf.c_s_time;
        l = (total*100000L)/loop/HZ;
        printf("%s : %D.%02D ms per loop.\n", msg, l/100, l%100);
        printf("  System %D%%, user %D%%, child sys %D%%, child user %D%%\n",
                buf.p_s_time*100L/total, buf.p_u_time*100L/total,
                buf.c_s_time*100L/total, buf.c_u_time*100L/total );
}
```