THE CORE WAR *NEWSLETTER*

The Official Newsletter of the International Core War Society

Features

Volume 5, Number 4 **Fall 1992** **An ICWS Publication**

From Proteus to Smooth Noodle Map 6

by Matthew Hastings

[Editor's Note: This is Part One of Two. Part Two will appear next issue. Much of the terminology in this article may be unfamiliar to you. A glossary is supplied elsewhere in this newsletter. Mr. Hastings may be reached via email at hastings-matthew@YALE.EDU*]*

This warrior, in a long series of incarnations, is to my knowledge unique. It does not fit quite right into the usual categories of Scissors, Paper, and Stone.

The basic idea is: a small fast bomber runs for several thousand cycles, killing or rendering useless all Scanners, Scissors, and other complicated, single-process programs. Then the program SPLits or JuMPs to a replicator and thereby secures a number of ties against other replicators. It also gets several wins against Return of the Living Dead and other warriors which create a large number of small processes.

The idea behind this program is powerful. Until I came up with this one, I had no program higher than 16 or so on KotH. The first program of this new type, which I call the Proteus type, was named simply Proteus. It stayed at around 10–14 for several weeks on KotH. The code for Proteus and the other programs is at the end of this article *[Ed.: This issue and next issue]*.

To digress, for those who do not know, KotH stands for King of the Hill. It is an ongoing corewar contest run by William Shubert through Internet.

The rules for the regular hill are ICWS'88 with an 8000 process limit and an 8000 core size. KotH retains a hill of 20 programs, with new programs getting on the hill by pushing others off.

I will continue the version history and touch on each version before looking more closely at the latest version.

The next version (after Proteus) was called Hellicon and had two replicators instead of one to increase the probability that one would survive. Smooth Noodle Map was the next major step, although it fared worse.

I had learned about the Stone bombing routine by then, and so the beginning stage became much more deadly. After a few test battles against a simple Stone and a slaver, I decided to again use two replicators but to use an error checking scheme.

A quick checksum was done on the first replicator to determine which to SPLit. The bombing was reduced so that the core was bombed for only 3*8000/25 = 960 cycles.

This reduction in bomb density decreased the program's performance, but I thought it necessary to do well against Stone. Also added was a quick JMP back to the loop to continue bombing.

This was later replaced by a SPL 0, MOV 2 <-1, JMP -1 core-clear which was executed at the same time as the replicator. This program was around 18 on the hill.

The first truly succesful version was Smooth Noodle Map 4, in reality the next version. After Andy Pierce pointed out that DAT #0,#0 spacers attract CMP scanners, Smooth Noodle Map 4 dropped the spacers and copied the code to different sections instead (Proteus did this also, it was only the first Smooth Noodle Map that used spacers). The bombing time was increased to 3000 cycles to allow for mod 8 bombing with mutation interspersed evenly.

This gave Smooth Noodle Map the deadliest bombing routine on the hill due to its high speed and even spread. The core is affected mod 4 in a mere 3000 cycles. The program went back to using one replicator because I had realized that after 3000 cycles, only small segments of core will be unbombed.

If there were a checksum located, as before, directly in front of the first replicator, either the checksum would have been hit, a loss compared to no checksum, or the replicator following the checksum would have been hit.

Smooth Noodle Map 4 entered KotH at number 1 and retained this position or number 2 until it was removed for Smooth Noodle Map version 4.1.

```
; Proteus

ptr   DAT #0,     #0
mouse MOV #12,    ptr
loop  MOV @ptr,   <copy
      DJN loop,   ptr
      SPL @copy,  0
      ADD #653,   copy
      JMZ -5,      ptr
copy  DAT 0,       833
      MOV -50,     @-50
      ADD #636,    -51
      DJN -2,      <450
      SPL 1000,    0
      JMP -4,      1
      DAT 0,       0
start MOV #1,      1250
      MOV #1,      1150
      MOV -8,      1000
      MOV -8,      1000
      MOV -8,      1000
arb   MOV -8,      1000
      MOV -8,      1000
      MOV -8,      1000
      MOV mouse,   arb+2000
      MOV mouse+1, arb+2001
      MOV mouse+2, arb+2002
      MOV mouse+3, arb+2003
      MOV mouse+4, arb+2004
      MOV mouse+5, arb+2005
      MOV mouse+6, arb+2006
      JMP arb+997
      END start
```

```
; Smooth Noodle Map
      MOV <112,    -125
      ADD 2,       -1
      DJN -2,      -2
      SPL cloop,   -cloop+1
      JMP -2,      <-3
; . . .
; 32 DAT  #0, #0s here
; . . .
cloop ADD <start,  7
      DJN -1,      #8
      JMN strt2,   5
start MOV #16,     7
loop  MOV @-2,     <6
      MOV <-3,     <5
      DJN loop,    -4
      SPL @-669,   -1
      ADD #653,    2
      JMZ start,   -7
copy  DAT 0,       833
; . . .
; 28 DAT #0,#0s here
; . . .
ptr2  DAT 0,       0
strt2 MOV #16,     ptr2
loop2 MOV @ptr2,   <copy2
      MOV <ptr2,   <copy2
      DJN loop2,   ptr2
      SPL @copy2,  0
      ADD #653,    copy2
      JMZ strt2,   ptr2
      DAT 0,       0
copy2 DAT 0,       833
```

```
; Smooth Noodle Map, 4.1
;
start MOV s7,      <ptr1
      MOV s6,      <ptr1
      MOV s5,      <ptr1
arb   MOV s4,      <ptr1
      MOV s3,      <ptr1
      MOV s2,      <ptr1
      MOV s1,      <ptr1
      MOV r5,      <ptr2
      MOV r4,      <ptr2
      MOV r3,      <ptr2
      MOV r2,      <ptr2
      MOV r1,      <ptr2
      MOV ro,      <ptr2
      MOV rm,      <ptr2
      JMP start+300, <ptr1
      DAT #0,      #0
s1    ADD 3,       1
s2    MOV <2589,   -2583
s3    DJN -2,      #998
s4    SPL 2584,    -2584
s5    SPL 0,       -1
s6    MOV 2,       <-1
s7    JMP -1,      0
rm    MOV #12,     0
ro    MOV <-1,     <3
r1    MOV <-2,     <2
r2    JMN ro,      -3
r3    SPL @0,      -833
r4    JMZ rm,      rm
r5    MOV 0,       -1
ptr1  DAT #0,      #start+307
ptr2  DAT #0,   #arb+307+2584

;Lots of decoys here....
```

Pardon Me, But Your Teeth Are In My Neck

by Chris Lindensmith

In some of the past tournaments, one very effective type of program has used vampiric capture to defeat opponents. Cowboy won in 1988 with a fairly straight-forward vampire that used the captured cycles to try to bomb the remaining uncaptured cycles. In 1991 another vampiric capture program, Vlad, took third place. The generally good performance of vampires has prompted proposed variations on the standard such as the descendent count limit suggested by Jon Newman, but there do not seem to be many (if any) programs that are actually designed to counter-attack vampires. I will discuss here some of my attempts to make so called vampire killer programs along with their advantages and drawbacks.

Vampires work in general by preparing a section of code that I will call the "trap" and then attempt to hit their victims with a `JMP trap` bomb. Once a process has been captured, the victim process usually runs in a loop in which each time through the loop each captured process executes a `SPL trap` instruction in order to slow down the opponent. The trap usually contains a counter that tallies the number of captured processes until the maximum number of sibling processes has been reached, all of which then commit suicide leaving the vampire victorious. Some programs also subvert the capture processes and use them to throw `DAT` bombs at their own siblings. An exampe of a very simple vampire is shown below:

```
; Lost boy
; example vampire program
; ICWS'88

start  JMP CLOOP
; bombing location
btarg  DAT 0,      2
; the jump to the trap bomb
bomb   JMP @bomb,  1

; The Trap

; decrement the counter
trap   ADD #-1,    tcount
; create more victims
tloop  SPL trap
; still victims available?
       JMN tloop,  tcount
; no, so commit suicide
       MOV tcount, tloop
; the counter and suicide
tcount DAT 0,      64
       DAT 0,      0

; The Bomber

; decrement the bomb
cloop  ADD #-20,   bomb
```

```
; increment the target by
; same
      ADD #20,    btarg
; do not bomb self
      SLT #15,    btarg
; repeat as necessary
      JMP cloop
; throw the bomb
      MOV bomb,   @btarg
; repeat
      JMP cloop
```

It is obvious from looking at how the vampire captures its prey that it must provide some sort of pointer to its own location. This is what we will use to track and destroy the vampire, although there are limits and clever programmers can write vampires that are more difficult to track and kill.

Since the vampire is spreading information all through memory as to how an unsuspecting program can find its trap, a vampire-killer simply needs a way to find these bombs and take advantage of the information that they contain before the bombs find the executing part of the program. Perhaps the simplest, although not the most effective means is to look through memory one location at a time and bomb the location to which any non-zero memory location points. The following program illustrates this technique:

```
; Simple Vampire Killer

; look through memory
lloop CMP empty,  <targ
; found something, bomb it
targ  JMP bomb,   #-2
; nothing, keep going
      JMP lloop
empty DAT 0,      0

; targ now points to trap
; entry
bomb  ADD @targ,  targ

; Do whatever type of
; bombing you want in here,
; using targ as starting
; point

      JMP out
; go to other work
```

This will not work if the bombs start coming from above instead of below, so you could easily extend this idea to check a few prepared “pickets”, one above and one below, in each of which the contents are known, and then bomb the vampire when it alters one of the pickets. I tried a program like this for my first vampire-killer when all I had were the 1988 tournament programs to compete against. It clobbered Cowboy nearly every time. As long as the vampiric bombs did not begin closer to the program than to the picket, Cowboy was easily beaten, as are most of the vampires that appear in the tournament rosters.

Authors of vampiric programs can also try techniques to avoid being so easily killed by anti-vampire programs. One simple idea is to insert one or more levels of indirection into the jump instructions that lead to the trap, placing intermediate locations safely away from the executing processes and the trap.

For low order indirection, where there are only a few intermediate locations, an indirect-vampire killer is easy to write; just insert a few `ADD @targ, targ` instructions. Since most traps are small and the executing instructions usually refer only to locations within the trap, this will still leave `targ` pointing somewhere within the trap and will also reduce the effectiveness of indirect vampires.

There are a few catches though. Since many traps include bombing instructions, you could end up pointing to the next location that the captured cycles are intending to bomb. This is not much of a problem if the trap has not been used yet and the bombs are set to begin nearby, but if the target pointer has been running, the indirect-vampire-killer could miss entirely. Adding a large number of `ADD` instructions could also make your program a larger target and slow it down so that the `JMP` bombs can get to it before it finishes off the vampire.

For higher order indirection, such as a long trail of `JMP` bombs that reference each other all the way back to the trap, the problem is more difficult but the vampire killer can expect to obliterate at least a part of the trail back and avoid capture. This still leaves the "captured" process jumping into oblivion, so it might be wise to split off a new process at a location away from the impending `JMP oblivion`.

Another simple technique is to separate the trap and the bomber section of the vampire by a large number of `DAT` statements at load time. A vampire killer would still be able to destroy the trap successfully preventing capture, although it would not get a quick and easy win.

Perhaps the most effective trick is used by the vampire program Net. It looks quickly through memory and only throws bombs when it finds a potential victim. This stealth can be devastating to nearly any program, and so far I have not been able to write a program that can consistently find it before being found.

So far, all of this discussion has assumed that opponents will use an offense of vampiric capture exclusively, but this is not at all true and leads to one of the major drawbacks of vampire-killing programs — speed (or lack thereof).

The vampire-killing programs are primarily defensive and need to have sibling processes that go through memory and attempt to kill off any other types of program that may be present. This leaves the vampire-killer running at half-speed or even slower against programs that do not try to capture their enemies.

It also greatly increases the size of the program, with perhaps a segment for offense, a segment to build and check pickets, and a segment to do the bombing when a vampire is found. The lack of speed and the large size can be serious disadvantages in tournaments where many small, fast bombers appear which quickly bomb memory and cripple or kill the vampire-killer. There is some comfort in that large programs can often survive corruption and be deadly even in some new, unexpected form.

In running one of my recent programs against the 1990 tournament roster, it performed better that everything except XTC, Net, and three or four fast bombers lik ZD. Hopefully some of the speed problems can be overcome and vampire killers can vie for the top spots in future tournaments.

Glossary — Continued From Page 9

Stealth - Lack of visibility to an opponent's program. Making B-fields zero to avoid B-scanners.

Stone - A Stone-like program designed to be a small bomber. Part of the Paper-Scissors-Stone analogy.

Vampire - A slaver. A program designed to sap cycles away from an opponent and put them toward its own uses.

Glossary

Note: Many of these terms started out as specific programs and have moved on to become generic terms. If a term X *has as part of its definition,* "an X-like program", X *is just such a program.*

B-Scanners - Scanners which only recognize non-zero B-fields.

`CMP`-Scanner - A Scanner which uses a `CMP` instruction to look for opponents.

Color - Property of bombs which also slow down Scanners.

Decoys - Instructions meant to slow down Scanners. Typically, `DAT`s with non-zero B-fields.

Incendiary Bomb - An alternative to the `SPL 0/JMP -1` bomb. Looks like `SPL 0, -8/ MOV -1, <-1`. Creates a `SPL 0` carpet ahead of itself.

Leech - A Leech-like program. A program which enslaves another. Usually accomplished by bombing with `JMP`s to a `SPL 0` pit with an optional core-clear routine.

Off-axis - Scanners often search using a comparison between two locations of memory M/2 apart, where M is the memory size. Off-axis scanners use different offsets.

Paper - A Paper-like program. One which replicates a process many times. Part of the Paper-Scissors-Stone analogy.

Replicator - Generic for Paper. A program which makes many processes.

Scanner - A Scanner-like program which searches through core for an opponent rather than bombing blindly.

Scissors - A Scissors-like program designed to beat replicators. Part of the Paper-Scissors-Stone analogy.

Slaver - Generic term for Leech. A program which enslaves another.

Creating Modile Warriors Without Loops

by William Hamaker

Most mobile warriors have a program loop which copies the program to some new location. The obvious exception to this is IMP.

A trick involving multiple threads in a program can be used to create mobile warriors larger than IMP that "hop around" without loops. The basic idea is that a single MOV command can copy more than one statement if it is being executed by several threads at the same time. Consider the following examples:

```
; Hopper - a mobile
; warrior with 3
; commands and 3
; threads
;
; create and
; synchronize 3
; program threads
start   SPL next
        JMP hopper
next    SPL hopper
; start of actual
warrior
hopper  MOV #0, 3
        MOV <2, <2
        JMP @0, hopper
        END start
```

```
; Hopper2 - a mobile
; warrior with 4
; commands and 2
; threads
;
start   SPL hopper2
hopper2 MOV #0, 4
        MOV <3, <2
        MOV <2, <1
        JMP @0,hopper2
        END start
```

Since the memory is initialized to all zeros, you can skip the initial MOV #0 statement to create faster mobile warriors that die if they reach core locations that are filled in with non-zero values. While not useful as a lone warrior, a "flea generator" which keeps replacing any fleas that died might have some value.

```
; Flea - a mobile
; warrior that only
; lives if it lands
; next to a zero
;
; It has 2 commands
; and 2 threads
;
start   SPL flea
flea    MOV <2, <1
        JMP @0, flea
        END start
```

The MADgic Corner

Well, it seems that I am destined to never publish a newsletter without having to apologize for something. This time, fate and the U.S. Postal Service have conspired to delay this newsletter a full month. I am sorry and it shall not happen again.

On a happier note, many of you talented authors out there have written articles and submitted them for publication right here in this very forum. Some of that work appears in this newsletter.

~~~~~~~~~~~~~~~~~~~~~~~~~

The 1992 annual ICWS is nearly here! This is your final call. Here once again is the vital information you need to submit a warrior:

Deadline: December 15, 1992.

Rules: ICWS '88, core size of 8192, maximum number of processes per warrior of 8000, and ties declared after 100000 cycles (turns per side). Wins 3 points, ties 1 point, and losses no points.

Submission: Send entries to JonNewman in one of these formats (in order of preference):

•Electronic mail to:

`jonn@microsoft.com`

• Text format, 3 1/2" Macintosh 800K disk, please check for viruses.

• Text format, 3 1/2" IBM 720K disk, please check for viruses.

• Text format, 5 1/4" IBM 720K disk, please check for viruses.

•Text format, 5 1/4" IBM 1.44MB disk, please check for viruses.

•Printed, maximum length 50 instructions.

Limit: Members may submit 1 entry each, 2 if in electronic form. Branch sections may submit 5 entries, 10 if in electronic form. Entries are limited to 50 instructions, 300 instructions if in electronic form. No scatter loading will be supported (instructions must be contiguous).

Legal: All entries become public domain upon submission. The ICWS will keep them confidential until after the Tournament.
~~~~~~~~~~~~~~~~~~~~~~~~~

~~~~~~~~~~~~~~~~~~~~~~~~

For those of you led astray by the previous issue of this newsletter, I have some corrections.

First, the USENET email server at `ucbvax.berkeley.edu` is no longer in service. I am trying to locate another server in the meantime. Send email to me if you want to be notified when I have found one.

If you somehow currently receive the newsgroup `rec.games.corewar` but just can not post, you can send your post via email to rec-games-corewar @ cs.utexas.edu and it will automatically be posted for you. I have verified this.

If you wanted to contact Tom Poindexter (of C-Robots fame) and could not contact him using last issue's email address for him, you might want to try tpoindex@nyx.cs.du.edu. Tom wrote that this address should work.

Quarterly Challenge #7

Your challenge for this issue, should you choose to accept it, is to submit a challenging question for your fellow readers. I will print the better challenges in subsequent issues.

Should you not accept the challenge above, the least you could do is submit a warrior or two to the annual ICWS tournament. There is no greater challenge than to take on fellow Core War enthusiasts in combat!

~~~~~~~~~~~~~~~~~~~~~~~~

The challenge last issue (QC #6) asked an interesting question about modulo numbers. For a core size of 8000, what step size produces the least sum (bomb) free space given 8000 "bombs"?

Well, thanks to the assistance of others with fast computers, I am happy to report that step sizes of 3039 and 3359 both produce least sum free spaces of 101410. Does this really matter, or is this just an academic question?

I put this question to the test. I compared Dewdney's Dwarf (which incidentally has the worst-performing step size of its class) against a Dwarf altered to use the best step size which would not overwrite itself. The comparison took place on KotH. BestDwarf and WorstDwarf battled the exact same opponents.

BestDwarf finished with 30% wins, 6% ties, and 64% losses. WorstDwarf, on the other hand, finished with only 21% wins, 5% ties, and 74% losses. The difference in performance is significant.

For a smarter program, the right bomb size could mean the difference between being one of the best programs and being *the best* program!

Here is a list of the best-performing bombing step sizes for the first ten separations (contiguous unbombed spaces. Example: Dwarf needs a separation of three to avoid overwriting itself).

I have left out the "negative" bombing step sizes. For example, -3039 = 4961 is just as good a bombing step size as 3039 and leads to the same sum free space. So all in all, there are really four "best" bombing step sizes. You need to choose which one is truly best for you.

Separation	Step Size	Free Space
0	3039	101410
0	3359	101410
1	3094	101856
1	2234	101856
2	none	
3	3044	111160
3	3364	111160
4	2365	114390
4	3315	114390
5	none	
6	none	
7	2376	133360
7	2936	133360
8	none	
9	2430	147380
9	2930	147380
10	none	

Given enough time, it would be interesting to explore the efficiencies of bombing with multiple step sizes. Perhaps bombing first with a step size of 3039 and then second with a step size of 3359, and then continuing to alternate between the two step sizes, would produce better results than just using a step size of 3039 or a step size of 3359 alone. And then again, perhaps it would be worse.

A. J. Pierce swears by the fibonacci series. I have not yet taken the time to explore these and other combinations of step sizes. Perhaps next issue.

The Core War Newsletter

Mark A. Durham
Editor

AMRAN
Founding Publisher

William R. Buckley
Founding Editor

The Core War Newsletter, the official newsletter of the International Core War Society since 1987, is published quarterly by the International Core War Society for Core War enthusiasts everywhere.

International Core War Society

Jon Newman
Director

Send all correspondence, including renewals and changes of address, to:

ICWS
c/o Jon Newman
13824 NE 87th Street
Redmond WA 98052-1959 USA
email: `jonn@microsoft.com`

Submissions of articles, letters, comments, suggestions, etc. to:

TCWN
c/o Mark A. Durham
18 Honeysuckle Terrace
Spartanburg SC 29307-3760 USA
phone: (803)579-0431
email: `durham@cup.portal.com`

Back issues of *TCWN* prior to 1992 are available from AMRAN, 5712 Kern Drive, Huntington Beach, CA 92649-4535, USA - email: xwbuckley@fullerton.edu.