A "unique number" generator

by ARMEN NAHAPETIAN
"Arya-Mehr University of Technology
Tehran, Iran

ABSTRACT

In many computer systems and applications, the need arises for the software to generate unique numbers or names for the identification of dynamically generated entities.

Such a generator is described here which produces new numbers and reuses numbers released by the system.

An analysis is carried out to determine the storage requirements of the generator in relation to the number of requests that can be accommodated.

INTRODUCTION

In computer applications, situations often arise where the software dynamically generates entities that must be named for future reference.

Examples of such cases are compilers that generate names for temporary locations in assembly language, graphic systems that generate names for created subpictures, operating systems that generate internal names for jobs submitted or files created, etc.

The generation of unique names for such cases can, of course, be accomplished by incrementing a counter each time a new name is requested. The resulting integer can directly be used as a name or converted to an appropriate base if an alphabetic or alphanumeric name is needed.

For example, to generate five-character Fortran identifiers, the number can be converted to a mixed radix integer

\[ a_{25} b_{25} c_{10} d_{10} e_{10} \]

the first digit representing a letter and the rest alphanumeric characters.

The method just described has the following drawback. Since in most applications the names generated are used for a period of time and then released; eventually the counter will be incremented to its limit, even if the set of names currently in use has a much smaller size than this limit.

With additional restrictions imposed on the names, the limit of the counter might be reached even sooner.

If, for example, names were restricted to one letter followed by one digit, this limit would be 260.

To somewhat overcome this inefficiency, the number generator to be described can be used which not only generates new names, but also tries to reuse as many of the names released as possible.

THE GENERATOR

The generator works by simply storing the released numbers (names) in a fixed-size stack. If a new number is requested the stack is tried first and only in case of an empty stack is the counter incremented to generate a new number.

Since the size of the stack is finite, cases might arise where a returned number cannot be pushed onto a full stack. Thus, the reuse of such numbers is eliminated. The results obtained in section three can be used for choosing an appropriate stack size based on the counter limit and the total number of requests anticipated.

The algorithms that follow represent the “request” and “return” operations for the unique number n. The initial value of the variable “pointer” should be zero and “counter” must initially be set to the smallest integer that can be represented or is appropriate for a name. “Stacksize” and “countlimit” represent the maximum capacity of the stack and the maximum count of the counter, respectively. The array “stack” and integers “pointer,” “countlimit” and “stacksize” are assumed to be declared globally.

procedure request (n); integer n;
Begin
if pointer>0 then begin n\leftarrow stack (pointer); pointer \leftarrow pointer-1 end
else if counter=countlimit then overflow
else begin
counter\leftarrow counter+1;
next \leftarrow counter
end
end request;
procedure return (n); integer n;
Begin
if pointer < stacksize then begin
    pointer = pointer + 1;
    stack (pointer) = n
end
end return;

THE STACK SIZE

To determine the minimum stack size for a specific application, in addition to the number of names needed and the counter capacity, statistical information on the occurrence of the requests and returns must be provided. Of course in any case the stack size need not be larger than the number of distinct values the counter can produce.

The results of the following analysis can be used in cases where the occurrence of requests and returns (transitions) can assume to be a Markov process, i.e., the probability of a transition being a request or a return is independent of previous transitions. The results can even be used in applications where a large number of request lead into a steady state Markov process, and a large number of returns terminate the process. Since to determine the stack size in such a case one need only consider the steady state (at the beginning of the process the stack is almost empty, and at the end, most numbers lost due to a full stack will not be needed again for reuse).

Assuming a probability p for a request and q = 1 - p for a return at each transition, the state diagram and the transition probability matrix of the process are shown in Figure 1.

The number representing each state is the number of elements in the stack of size S and the state represented by G is the state where each time entered (due to a request) a new number must be generated by the counter due to an empty stack.

Using $T_i$ to represent the expected number of transitions needed to reach state G from state i, and assuming C to be the number of distinct values the counter can produce, the expected number of transitions for the generator to overflow can be shown as

$$T_{over} = T_0 + CT_G$$

i.e., the expected number of transitions for entering state G, C + 1 times starting from state zero.

And the expected number of transitions before overflow occurs, being one less than $T_{over}$ is

$$T = T_0 + CT_G - 1$$

$T_0$ and $T_G$ can be obtained from the following set of equations

- $T_G = p + q(1 + T_1)$
- $T_1 = p(1 + T_G) + q(1 + T_2)$
- $T_2 = p(1 + T_1) + p(1 + T_G)$
- $T_{S-1} = p(1 + T_{S-2}) + q(1 + T_S)$
- $T_S = p(1 + T_{S-1}) + q(1 + T_0)$

To be

- $T_0 = (1 + T_1) / (1 + T_2)$
- $T_G = (1 + T_1) / (1 + T_2)$

where $a = q/p$, thus the expected number of transitions before overflow occurs is

$$T = (C + 1) (a^{S+1} - 1) / (a - 1) - 1$$

And finally the expected number of requests that can be accommodated with a stack size of S and counter capacity of C,

$$R = pT = (C + 1) (a^{S+1} - 1) / (a - 1) - 1$$

which for $p = q = 1/2$ reduces to

$$R = (C + 1) (S + 1) - 1/2$$

CONCLUSION

A number generator was introduced that issues a unique number each time a request is made, trying to save and reissue returned number by using a stack.

The generator can be used for generating alphanumeric names by converting the generated numbers to an appropriate base.

Assuming the request and return transitions to be a Markov process, formulas were derived for the expected number of requests that can be accommodated in terms of the maximum capacities of the counter and the stack.

REFERENCE