# THE PROCESSOR IDENTITY PROBLEM

Richard J. Lipton

Arvin Park

CS-TR-088-87

April 1987

# The Processor Identity Problem

*Richard J. Lipton*
*Arvin Park*

Department Computer Science
Princeton University
Princeton, New Jersey 08544

April 9, 1987

**Abstract**

In this paper we pose the problem of establishing unique identities for a group of $N$ identical processors which possess a common shared memory to which asynchronous read and write operations can be performed. We introduce a set of protocols which successfully assign unique integers from the set $\{0,1,2, \ldots ,N\text{-}1\}$ to each processor.

By applying these simple protocols to a shared memory multiprocessor system under development at Princeton [Park87], we have eliminated the need for hardwired addresses, or customized software for individual processing nodes. Individual processing nodes can now be installed or replaced without tedious configuration work. We have thereby greatly improved system modularity.

Keywords and Phrases - *Multiprocessor, Probabilistic Algorithms, Shared Memory, Synchronization.*

## 1. Introduction

Standard solutions to the dining philosophers problem claim that no philosophers starve. These solutions commendably synchronize communal utensil manipulation. However, they fail to address the more fundamental issue of identity.

Suppose we have $N$ philosophers named Bruce. Consider the dilemma the philosophers encounter while trying to seat themselves. "Bruce - , why don't you sit in chair number one. And you, Bruce, can sit in chair number two, ... , Bruce, you and Bruce can sit in the remaining two chairs." Our cadre of philosophers would scarcely be able to seat themselves round a table let alone ponder the cumbersome of the task of dining. In addition to burdening the intellectual faculties of a group of dining philosophers, the identity problem has implications in the seemingly unrelated field of multiprocessor system development.

We encountered the Processor Identity Problem while writing communication primitives for a shared memory multiprocessor system which is being built at Princeton [Park87]. This system allows processors to communicate by performing asynchronous read and write operations to a common shared memory. To implement these primitives it is necessary to assign each processor to a unique task and set of memory locations. Our protocols allow a set of identical processors running the same program to arrive at such a unique assignment.

Of course this problem may be trivially solved by loading a different program into each processor. But we have found that this entails a great amount of inconvenience. As updated versions of the software are created, the software must be custom tailored for each processor. The programs must subsequently be individually loaded into each processor. It is more efficient to maintain a single version of a program which can be broadcast in parallel to all processors.

Another option is to provide each processor with a special file, jumper configuration, or ROM location which contains a unique identification tag. (Ethernet socket addresses are maintained this way [Metc76].) This solution requires the overhead of maintaining global tables of unique addresses, as well as extra processor configuration work.

A third option is to provide the system with an atomic read-modify-write or test-and-set operation. This capability will allow processors to serially grab control of a single shared variable and thereby establish an ordering of processors. This type of architectural feature must be built into the hardware of the machine, and it can entail a great deal of design complexity. (Especially if arbitration circuits must be constructed for multiple data paths.)

Instead of accepting one of these standard alternatives, we decided to look for a simple algorithm that could be used by a group of identical processors running identical programs to agree on an assignment of unique identities for each processor.

If two truly identical processors begin running the same program at precisely the same time, it would clearly be impossible arrive at unique identities for each processor through shared memory communication. These two processors would proceed in lock step performing identical actions to the same memory locations. It would be impossible for one processor to detect the existence of the other processor let alone worry about distinguishing itself from the other processor.

An analogy can be made between the *Processor Identity Problem*, and the problem of differentiating between identical twins. Identical twins are structured around identically coded strands DNA in much the same way as our processors are structured around identically coded finite state machines. Twins differentiate themselves through interactions with random processes in their environments. One twin might receive a scar from a falling rock will the other one does not. One twin might win the New York state lottery and and spend the rest of his life tanning in Hawaii, while the other one remains an untanned elevator repairman in Trenton.

To differentiate between identical processors we similarly introduce an element of randomness. We assume that each processor possesses a random number generator. We argue that this is a "reasonable" assumption in section six of this paper.

In section two of this paper we present a model for, and precisely state the *Processor Identity Problem*. In section three we present our solution for the synchronous case. In section four we describe and analyze the *Random Wait Protocol* for solving the asynchronous case. We present the *Random Key Protocol* as an alternate solution to the asynchronous case in section five. In section six we discuss methods of producing different streams of random numbers from identical processors.

## 2. Model

Our model consists of a collection of $N$ processors which all access a common shared memory $SM$ which consists of $k$ words of size $b$ bits. $SM[i]$ refers to the $i$th word of shared memory $SM$. Each processor can perform asynchronous read and write operations to fixed sized words of shared memory without conflict from the other processors. When more than one processor simultaneously write to a single memory location two outcomes are possible. If the both processors write the same word, the memory location will contain that word. If both processors simultaneously write different bit patterns to the same memory location, the contents of the memory location is indeterminate. The only way processors can communicate to each other is through the shared memory $SM$ (see Figure 2.1).

Each processor possesses its own local memory which is not accessed by other processors. $LM[k]$ denotes the $k$th word of local memory $LM$. In addition, each processor possesses a random number generator (which we discuss further in section six).

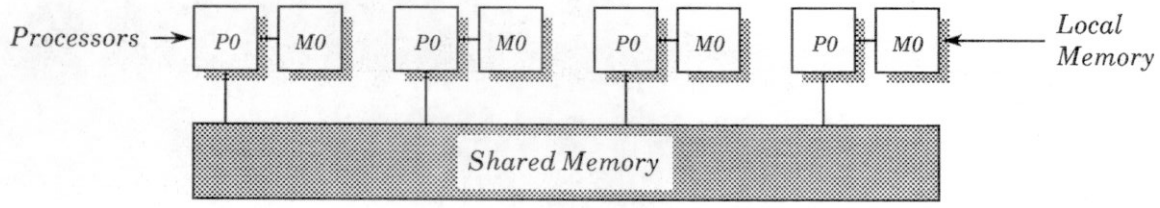


Figure 2.1: Shared Memory System Model

The goal of the identity problem is to produce a single protocol which is executed by every processor to produce a one-to-one assignment of the $N$ processors to the integers $\{0,1, 2, \ldots , N\text{-}1\}$ . If all processors are active, this protocol must terminate with probability one in a finite amount of time. Note that it is impossible to guarantee that a solution can be arrived at in any fixed amount of time. We state this fact more precisely in the following theorem.

**Theorem 2.1:** *For any fixed time t, no protocol exists which always solves the Processor Identity Problem within time t.*

Proof: Let us examine the case of two processors. Assume such a protocol exists. After a fixed length of time $t$, one processor is assigned to the number 1 and the other processor is assigned to the number 2. This means the two processors must have performed different assignment operations within the time $t$. However, it is possible for two processors to remain exactly synchronized during the time $t$ performing identical sequences of operations. Even queries to different local random number generators may produce the same results for any fixed length of time $t$. (Of course this probability rapidly drops to zero as the number of queries to the local number generator increases.) Therefore, no protocol exist which always solves the *Processor Identity Problem* in any fixed length of time $t$. □

## 3. Synchronous Case

We first examine the synchronous case. This admits to the easiest solution, since one can guarantee that all processors execute a given instruction simultaneously. We first examine the two processor case.

### 3.1 Synchronous Protocol for Two Processors

For this protocol we only require two shared memory words of size 1 bit. We assume that both processors start at the same time and proceed in lock step. Both memory words are first initialized to

0 by each processor. The processors then pick one of the words at random, and then write a one to this word. There are two possible outcomes to this process: (1) Either both processors picked different words in which case the algorithm terminates. (The processor which picked the first word becomes number 0 and the other processor becomes number 1). (2) Or both processors pick the same word in which case the algorithm is repeated. The algorithm is stated more formally below:

```
SM[0] = 0;
SM[1] = 0;
while(SM[0] = 0 or SM[1] = 0) do begin
    SM[0] = 0;
    SM[1] = 0;
    i = random() modulo 2;
    SM[i] = 1;
}
processorid = i;
```

The program will halt with probability 1/2 after each iteration. The process can conceivably proceed for any fixed number of iterations but the expected number of trials can be calculated by considering each iteration to be a Bernoulli trial [Tuck80].

$$Expected\ number\ of\ trials = \sum_{j=1}^{\infty} jP(j) = \sum_{j=1}^{\infty} j\left(\frac{1}{2}\right)^{j} = 2$$

The expected number of trails is shown to be two. We next examine the synchronous case for $N \geq 2$.

### 3.2 Synchronous Protocol for $N \geq 2$

The solution to the Identity Protocol for $N$ processors is a generalization of the protocol for two processors. The protocol uses $N$ shared memory words of size 1 bit. Processors initially set each word to 0. Each processor then picks a random word and sets it to one. If less than $N$ bits are marked, then at least one bit was selected by more than one processor so the memory is re-initialized, and the process is repeated. If $N$ bits are set, each processor has chosen a unique word. The processors use their word addresses as a unique identification tags and exit the protocol. The algorithm is stated more formally below:

```
for all x between 0 and N-1 {                    /*set shared memory words to 0*/
```

```
        SM[x] = 0;
}
while(one or more shared memory bits are not set to 1) {
    for all x's from 0 to N-1 {                    /*set shared memory words to 0*/
        SM[x] = 0;
    }
    i = random() modulo N;                         /*pick a random bit and set it to 1*/
    SM[i] = 1;
}
processorid = i;
```

The algorithm will succeed with probability $N!/N^N$ in each iteration. The algorithm terminates with probability one as time goes to infinity but it may take a long time. The success probability can be increased by using $M$ bits of shared memory words where $M \geq N$. Each processor picks one of the $M$ bits at random and marks it. If $N$ bits are marked, each processor has a unique location. To produce an identification tag, each processor counts the number of marked locations with lower shared memory addresses, and uses this as an identification tag. This succeeds with probability $P' = (M!/(M-N)!)/M^N$, The expected number of trials is then.

$$Expected\ number\ of\ trials = \sum_{j=1}^{\infty} jP(j) = \sum_{j=1}^{\infty} jP'\left(1-P'\right)^j = \frac{1}{1-(1-P')} = \frac{1}{P'}$$

For example, if $M = 20$ and $N = 8$ the expected number of trials is five.

**4. Asynchronous Case: The Random Wait Protocol**

We now introduce the "*Random Wait Protocol*" which solves the asynchronous *Processor Identity Problem*. For the *Random Wait Protocol* to operate, each processor must be capable of producing a distribution of waiting times. It is not required that processors be synchronized, or that they produce the same distribution. It is only essential that each processor have some notion of time. This model of processing lies somewhere between the synchronous and the asynchronous extremes.

In practice, a waiting time can be generated by a timing loop which a processor executes for a given number of iterations. (Though clock skew between processors may make it impossible for these waiting times to be calibrated between processors.)

For some multi-tasking processors, however, it may be impossible to assume that a single process can execute a timing loop without interruption. They may simply be unable produce a distribution of

waiting times. For these systems we have developed the *Random Key Protocol* which is presented in section 4.

### 4.1 Random Wait Protocol for $N = 2$

In the *Random Wait Protocol* processors communicate by changing certain locations in memory at random intervals. If one processor periodically "toggles" a bit from 0 to 1 and 1 to 0, the other processor will eventually detect its presence even if both processors are toggling the same bit.

This protocol requires three bits of shared memory. One bit is used to signify that the operation is completed. We call this the "completion bit". The other two bits are "toggled" (periodically flipped from 0 to 1 and 1 to 0) by the processors. We call these the "*T* bits" (See Figure 4.1.1).

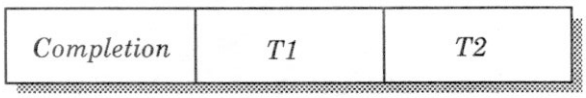


Figure 4.1.1: Shared Memory Organization for Two Processor Random Wait Protocol

The protocol operates in much the same manner as the synchronous protocol. Each processor first initializes the completion bit to a 0 value. The processors then pick one of the two *T* bits at random. Since the processors are not synchronized it is harder for the processors to detect the fact that they have picked the same *T* bit. This is what the toggle operation is used for. Each processor periodically toggles the *T* bit they have selected. After the toggle operation each processor reads both *T* bits, then waits a random amount of time before reading the *T* bits again. If either of the *T* bits was toggled an odd number of times during the waiting time, this will be detected.

If a processor detects a change in its own *T* bit, it knows that both processors are occupying the same location. It also knows it is the first processor to detect this fact So it selects the number of the other *T* bit as its processor identification tag and exits the program after setting the completion bit to 1.

If a processor detects a change in the other *T* bit, it knows that the other processor is "occupying" that bit so it sets the completion bit to one and then terminates. If a processor at any time detects that the completion bit has been set by the other processor it terminates. The protocol is stated more formally below and is graphically depicted in Figure 4.1.2:

```
completion = 0;                    /*set completion bits to zero*/
i = random() modulo 2;             /*select a random T bit*/
while(1) {
    toggle(T[i]);                  /*toggle your T bit*/
```

```
    myTbit = T[i];                    /*read your T bit*/
    otherTbit = T[(i+1) modulo 2];    /*read other T bit*/
    waitrandomtime();                 /*wait for a random amount of time*/
    if(completion==1) {               /*if completion bit already set*/
        processorid = i;              /*get processor identification number*/
        exit(0);                      /*and exit program*/
    }
    if(T[i] not = myTbit) {           /*compare your T bit with its previous value*/
        completion = 1;               /*if different set completion bit*/
        processorid = (i+1) modulo 2; /*get processor identification number*/
        exit(0);                      /*and exit program*/
    }
                                      /*compare other T bit with previous value*/
    else if(T[(i+1)modulo 2] not = otherTbit) {
        completion = 1;               /*if different set completion bit*/
        processorid = i;              /*get processor identification number*/
        exit(0);                      /* and exit program*/
    }
}
```

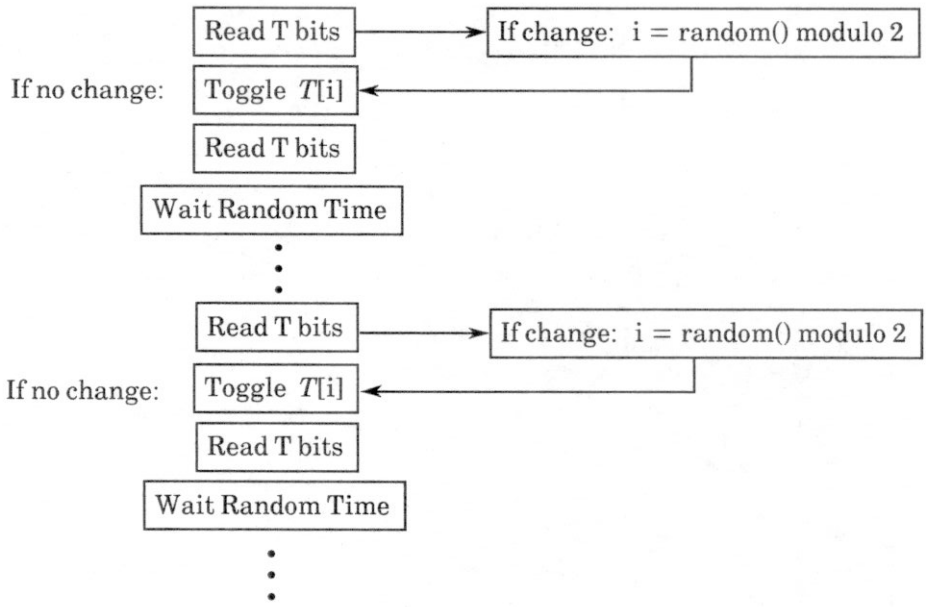


Figure 4.1.2: Random Wait Protocol For Two Processors

The random waiting time can be generated by a random number generator and a timing loop. We would like to use an exponentially distributed random waiting time. However, this cannot be exactly produced by a timing loop that executes only a discrete number of iterations. An exponential distribution can, however, be approximated to the nearest discrete interval.

To produce a waiting time from a random integer (which is assumed to be uniformly distributed from one and its maximum value) first produce a random number between 0 and 1 by dividing the random integer by its maximum possible value. them produce any exponentially distributed waiting using the following formula which we derived from the standard exponential probability distribution.

$$time = \frac{ln(1 - random)}{-\lambda}$$

Where $\lambda$ is the standard exponential parameter for the exponential distribution $P(t<T) = 1 - e^{-\lambda T}$.

We next show that the protocol is guaranteed to terminate.

**Lemma 4.1.1:** *If two active processors enter the two processor Random Random Wait Protocol, both processor will eventually exit the protocol.*

The protocol will terminate whenever one processor performs an odd number of toggle operations between two of the other processor's read operations. There is a constant probability of this occurring during each pair of read operations. Since read operations, and toggle operations continue to be performed while the protocol hasn't terminated, a detection event will occur with probability one as time goes to infinity. When one processor detects the presence of the other processor, it sets the completion flag and exits the protocol. The other processor will eventually read the completion flag and also terminate. The protocol will therefore always terminate. We next establish a lemma concerning detection.

**Lemma 4.1.2:** *Only one of the two processors can detect the presence of the other processor during a single execution of the Random Wait Protocol.*

Assume both processors were able to detect each others presence. Processors only detect toggles that occur between pairs of read operations. And processors perform no toggle operations between these read operations, or during the exit sequence if detection occurs. So if processor $A$ detects the presence of processor $B$, this means the processor $B$ has toggled between a pair of processor $A$'s read operations. This implies that processor $B$ has not yet detected anything. If both $A$ and $B$ detect each others presence this means processor $A$ must have toggled sometime after processor $B$'s toggle operations, however, processor $A$ never toggles after detecting processor $B$. Contradiction! □

We can now prove that our protocol correctly solves the *Processor Identity Problem.*

**Theorem 4.1:** *The Random Wait Protocol solves the Processor Identity Problem for two processors.*

Assume the Waiting Protocol doesn't solve the *Processor Identity Problem.* By lemma **4.1.1** the algorithm is guaranteed to terminate, this means the algorithm terminates with both processors assigned to the same identification number. However by Lemma **4.1.2** only one processor can detect the presence of the other processor. And the detecting processor always chooses an identification tag which is different from the present location tag of the other processor. This means that both processors cannot have selected the same identification tag. □

Let's now analyze the expected running time of the algorithm. The algorithm terminates whenever one processor detects the presence of the other processor. Assume that the read-toggle-read (*RTR*) sequence of operations occurs instantaneously. (This is a valid assumption if the waiting times are on average much longer than the duration of an *RTR* operation.) Each processor repeatedly performs *RTR* operations followed by exponentially distributed waiting times. Label one of the processors as *A*, and the other processor as *B*. Processor *A* can detect the presence of processor *B* if processor *B* performs an odd number of *RTR* operations during processor *A*'s waiting time. We can now represent any sequence of *RTR* operations as a series of *A*'s and *B*'s. The process stops when an odd number of consecutive *B*'s is terminated by an *A*, or an odd number of A's is terminated by a *B*. Assuming that both processor *A* and processor *B* have the same waiting time distributions, then the next number in the sequence is just as likely to be an *A* or a *B* (see Figure 4.1.3).

Consider odd-even pairs of numbers in the sequence. Half of these pairs lead to a processor being detected (*AB* or *BA*). The other half lead to no detection (*AA* or *BB*). Each case is equally as likely, so after each pair of *RTR* operations, detection will occur with probability 1/2. This means the expected number of *RTR* operations to detection will be.

$$\textit{Expected number of RTR operations} = 2\left(\sum_{j=1}^{\infty} jP(j)\right) = 2\left(\sum_{j=1}^{\infty} j\left(\frac{1}{2}\right)^{j}\right) = 4$$

After one processor detects the other and exits, it the remaining processor will have to finish its waiting time before terminating. The expected value of this final waiting time is $1/\lambda$. The expected time between *RTR* events of two active processors is half the expected time for one processor or $1/2\lambda$. For exponential distributions, the expected time for the *k*th event, is just *k* times the expected time for one event [Klei75]. The expected time for the first four *RTR* operations is then $4*(1/2\lambda) = 2/\lambda$. The total expected waiting time is then becomes $2/\lambda + 1/\lambda = 3/\lambda$.

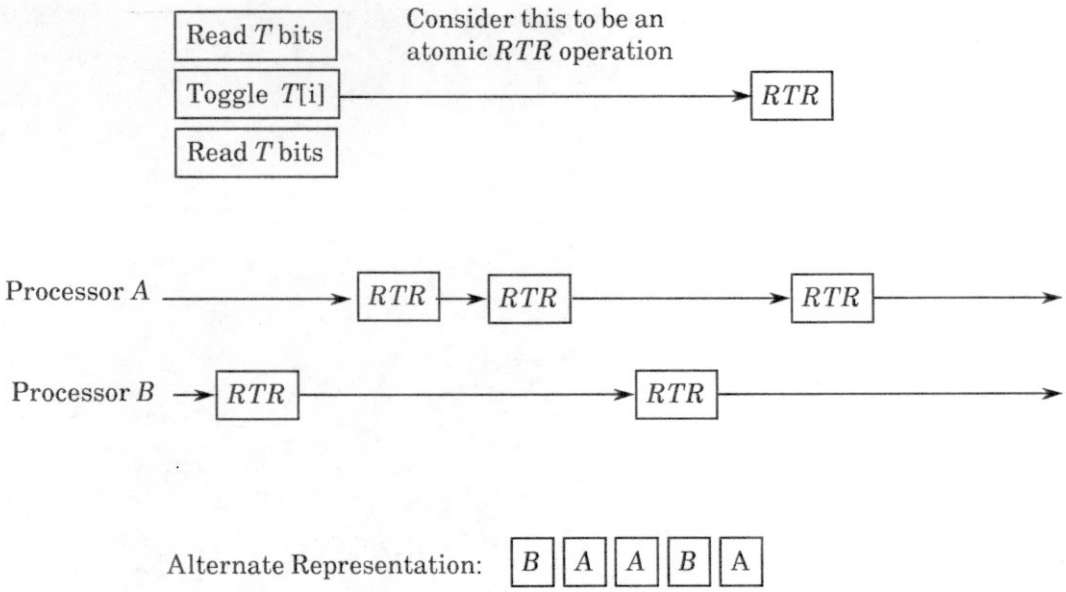


Figure 4.1.3: Random Wait Protocol For Two Processors

Note that the processors involved in the *Random Wait Protocol* need not generate the same exponentially distributed waiting time. It is only essential that each processor produces an exponential waiting time whose expected value is much larger than the time required for the slowest processor to perform an *RTR* operation.

### 4.2 Waiting Protocol for N ≥ 2

We generalize the *Random Wait Protocol* for the case $N \geq 2$. This requires $N+1$ bits. Like the previous protocol one, of the bits is reserved as a completion flag. The other $N$ bits are used as toggle bits (see Figure 4.2.1). The algorithm proceeds similarly to the $N = 2$ case. Each processor initially

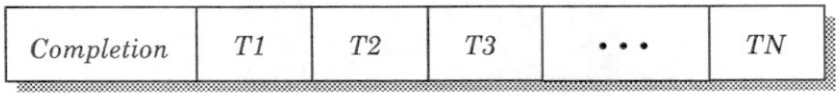


Figure 4.2.1: Shared Memory Organization for $N$ Processor Random Wait Protocol

sets the completion flag to 0. It then chooses a random $T$ bit and toggles it. Next, it reads all of the toggle bits, waits a random amount of time and reads all of the toggle bits again. If any of the $T$ bits have changed, it marks the changed $T$ bits as occupied in a table located in its local memory area. If a processor detects a change in its own $T$ bit, it moves to a random $T$ bit that it has not yet detected as occupied. If any processor detects toggling in all of the $T$ bits, it sets the completion flag to a 1 and

exits using its $T$ bit location as its processor identification tag. A processor which detects the completion bit is set exits the program using its toggle location for its processor identification tag. The algorithm is stated more formally below.

```
completion = 0;                          /*initialize completion bit to zero*/
i = random() modulo N;                   /*select a random T bit*/
LM[i] = 1;                               /*note this T bit as occupied*/
while(1) {
    toggle(T[i]);                        /*toggle your T bit*/
    readTbits();                         /*read your T bits*/
    waitrandomtime();                    /*wait for a random amount of time*/
    readTbits();                         /*read your T bits*/
    if(completion = = 1) {               /*if completion bit already set*/
        processorid = i;                 /*get processor identification number*/
        exit(0);                         /*and exit program*/
    }
    if(T[i] changed between reads) {     /*if your T bit changed*/
        while(LM[i] = = 1) {             /*get new T bit that has not been detected*/
            i= random() modulo N;        /*as occupied.*/
        }
        LM[i] = 1;                       /*mark this new T bit as occupied*/
    }
    for all x from 0 to N-1 {            /*if any T bit changed between reads*/
        if(T[x] changed between reads)
            LM[x] = 1;                   /*note this bit as occupied*/
    }
    if(all LM[x] = =1)                   /*if all T bits have been detected as occupied*/
        processorid = i;                 /*get processor identification number*/
        completion= 1;                   /*set completion bit*
        exit(0);                         /* and exit program*/
    }
}
```

This protocol works because any toggle bit that has one or more processors assigned to it will always have at least on processor assigned to it. This is Lemma 4.2.1.

**Lemma 4.2.1:** *If a T bit is occupied by one or more processors during the Waiting Protocol, it will remain occupied by at least one processor until the protocol completes.*

This is similar to Lemma **4.1.2**. If a processor detects a toggle operation on its own $T$ bit it moves to a different $T$ bit. However, the processor which performed the toggle must still remain on the original $T$ bit.

Suppose that a $T$ bit which was occupied by one or more processors during the Waiting Protocol is occupied by no processors. One of the processors must have performed the last toggle operation that caused a processor to leave. Call this processor $A$. Processor $A$ must have been prompted to leave by a subsequent toggle operation, since $A$ was the last to perform a prompting toggle operation, it must have prompted itself to leave. This is impossible because $A$ performs no toggle operations while looking for a change in its own toggle bit. Our original assumption produces a contradiction. So at least one processor must remain on the toggle bit until the end of the protocol. □

This leads into the next lemma.

**Lemma 4.2.2:** *Every T bit must eventually be occupied by at least one processor during the Random Wait Protocol.*

Suppose one $T$ bit is never occupied. Call this bit $x$. Since there are $N$ bits and $N$ processors, at least one of the other $T$ bits must be occupied by more than one processor. As time goes to infinity, one of the processors occupying this bit will detect the presence of another processor occupying the same bit, and it will choose another bit at random to occupy. If it does not select bit $x$ it will select another bit, and the process will repeat itself. This process cannot repeat indefinitely. Every repetition has a $1/N$ probability of touching bit $x$. As time goes to infinity, bit $x$ will eventually be touched. □

Our next lemma guarantees that the protocol will terminate.

**Lemma 4.2.3:** *If all N processors actively execute the Random Wait Protocol, then all N processors will eventually exit the Random Wait Protocol.*

Suppose one processor never exits the *Random Wait Protocol.* Call this processor $x$. This means the completion flag is not set. Since otherwise the processor $x$ will exit the protocol. (Note that once the completion flag is set, it will never be erased, since is can only be set by a processor entering the protocol, and it can only be set after all processors have entered the protocol.) Since the processor never leaves the protocol, then the processor must never detect a toggle operation in one at least one of the bits. Call this bit $y$.

Lemmas **4.2.1** and **4.2.2** taken together say that all bits will eventually be occupied, and once occupied, they will remain occupied. The implies bit $y$ will be occupied, and remain occupied. If this is the case, processor $x$ will eventually detect a toggle operation on bit $y$. Our assumption that one processor never leaves the protocol leads to a contradiction, therefore all processors will eventually leave the protocol. □

We can now prove *Random Wait Protocol* solves the *Processor Identity Problem.*

**Theorem 4.2:** *If all processors remain active, the Random Waiting Protocol will eventually produce a one-to-one assignment of the N processors to the integers {0,1,2, ... ,N-1}.*

Proof: Suppose the *Random Wait Protocol* does not produce such an assignment. By lemma **4.2.3** all processors must have exited the protocol. Since processors only exit the protocol after selecting one of the integers {0,1,2, ... ,$N$-1}, each processor must have selected an integer from {0,1,2, ... ,$N$-1}. If the protocol did not produce a one-to-one assignment, at least two of the processors must have selected the same integer. This means at least one bit must not have been exited from. This bit cannot have been occupied. If it ever was occupied, by **lemma 4.2.1** it would have remained occupied and one of the processors would have exited from it. The fact that one bit was never occupied contradicts lemma **4.2.2**.

Our assumption that the *Random Wait Protocol* does not produce a one-to-one assignment of processors to the integers {0,1,2, ... ,$N$-1} leads to a contradiction. Therefore, the *Random Wait Protocol* must eventually produce a one-to-one assignment. □

To estimate the expected running time of the algorithm, note that each processor can change $T$ bits at most $N$ times during the protocol. This is because processors never move to a $T$ bit they have occupied previously.

As we have shown in section **4.1**, if a a processor moves to a bin that is already occupied by another processor, it will take on average $1/2\lambda$ time for one of the processors to detect the fact that the bin is doubly occupied. One of these processors will then jump to a new bin. We can produce a rough estimate of the expected running time by assuming that every move is made to a an unoccupied bin, and that the expected time for each of these moves is $1/2\lambda$. We also assume that $N$ moves have to be made to fill all the $T$ bits. Because of the memory-less property of the exponential distribution, we can assume that the expected value for the $N$ moves equals the sum of the expected values of each of the individual moves. Our approximate estimate of the running time then becomes $N/2\lambda$.

## 5. Random Key Protocol

The Waiting Protocol may not prove satisfactory for some applications. Processors may simply be incapable of producing a random distribution of waiting times because of multi-tasking or real-time constraints. Even if all of the processors can produce random waiting time distributions, the waiting process itself is quite time consuming. The *Random Wait Protocol* is guaranteed to produce a correct assignment of processors to integers as time goes to infinity. However, it may be more practical to use faster probabilistic methods.

The *Random Key Protocol* correctly solves the *Processor Identity Problem* with probability greater than 1-ε, where ε is an arbitrarily small real number less than 1. It does not use random waiting times so in general it operates more quickly than the *Random Wait Protocol*.

We first describe the *Random Key Protocol* for the case where the number of processors $N = 2$.

### 5.1 Random Key Protocol: Case $N = 2$

In the *Random Key Protocol*, shared memory is divided into two "bins". Each bin contains space for a large random number. (This random number may span one or more memory words.) Included in each bin is a "valid" bit which indicates if the contents of the bin is a valid random number (see Figure 5.1.1).

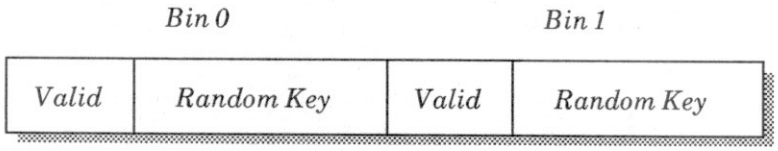


Figure 5.1.1: Shared Memory Organization for Two Processor *Random Key Protocol*

Each processor initially generates a large random number. These random numbers are assumed to be unique. If the random numbers are one hundred bits long, the probability that the two number will agree is $2^{-100}$. Each processor now possesses a unique random number. One might argue that such a number constitutes a unique identity, but for purposes of parallel processing it is more useful if these unique tags come from a predetermined set of tags $\{0,1,2, \ldots, N\text{-}1\}$. Multiprocessing tasks can be specified in terms of such a set.

Each processor first initializes the valid bits of both bins to 0. It then picks a random bin, and sets the valid bit of that bin before writing its large random number into the bin. The processor alternately reads the contents of both bins. If it detects a valid random number in the other bin, and verifies that its own random number remains in tact, it knows that both processors occupy different bins. It then takes the number of its bin as its processor ID, and exits. If the processor detects no change in either bin, it continues alternately reading each bin.

If the processor finds that its own random number has been disturbed, it knows that both processors are attempting to occupy the same bin. It sets its valid bit to 0 and selects a new random bin to write to. The algorithm is described more formally below:

```
validbit[0] = 0;                                  /*set valid bits to 0*/
validbit[1] = 0;
r = random();                                     /*get random number*/
while(1) {
    i = random() modulo 2;                        /*select random bin*/
    validbit[i] = 1;                              /*set valid bit of selected bin*/
    bin[i] = r;                                   /*write random number to bin*/
    while(validbit[i] =1 and bin[i] = r) {        /*repeat while your own bin is OK*/
        if (validbit[(i+1)modulo 2] = 1) {        /*read other valid bit*/
            if(validbit(i)=1 and bin[i]=r) {      /*if your own bin remains in tact*/
                processorid = i;                  /*get processor ID and exit*/
                exit(0);
            }
        }
    }
    valid[i] = 0;                                 /*reset valid bit*/
}
```

We first establish that at least one processor will always finish the protocol in Lemma 5.1.

**Lemma 5.1.1:** *At least one processor must eventually leave the two processor Random Key Protocol.*

Suppose neither processor leaves the protocol. If at least one processor is continually choosing a new random bin, the protocol will terminate as time goes to infinity because the processors will eventually occupy different bins in which case the at least one processor will leave. If neither processor is changing bins then both processors must always see a their own valid random number in their own bin and no valid random number in the other bin. If this is the case, both processors must occupy the same bin, therefore both processors must possess the same random number. Our assumption that no processor leaves the *Random Key Protocol* produces a contradiction. Therefore, at least one processor must eventually leave the two processor *Random Key Protocol.*□

We establish another useful lemma.

**Lemma 5.1.2:** *If both processors (A and B) initially choose distinct random numbers in the two processor Random Key Protocol, and processor A writes the valid bit of a bin, processor A will not touch the other bin unless its own bin is subsequently disturbed.*

Assume that processor $A$ sets the valid bit of bin 0, and that bin 0 is never subsequently written to by processor $B$. Also assume that processor $A$ subsequently writes to bin 1. Since processor $A$ only writes to its own bin, processor $A$ must have switched bins. Since processor $A$ only switches bins if its own bin is disturbed by processor $B$, processor $B$ must have disturbed bin 0 after the valid bit was set by $A$. This contradicts the assumption. □

**Lemma 5.1.3:** *If both processors initially choose distinct random numbers in the two processor Random Key Protocol, and one processor leaves the protocol, both processors must eventually leave the protocol.*

Assume that processor $A$ has left the protocol and processor $B$ remains executing the protocol. Processor $A$ left the protocol after first making certain that the other bin contained a valid bit. Processor $B$ must have set this valid bit because processor $A$ would have reset this bit before changing bins. Processor $A$ reads its own valid random number in its own bin before exiting the protocol. Both bins must now contain valid random numbers, because processor $A$ has not disturbed processor $B$'s bin before exiting. And by Lemma **5.2.2** processor $B$ will never disturb A's bin. However, if this were true than $B$ will eventually the protocol. This contradicts our original assumption, therefore if one processor leaves the protocol then both processors must leave the protocol. □

We next finally prove that the *Random Key Protocol* solves the *Processor Identity Problem* for the case of two processors.

**Theorem 5.1:** *If both processors initially choose distinct random numbers in the two processor Random Key Protocol, then the protocol will successfully solve the Processor Identity Problem.*

Proof: Suppose the *Random Key Protocol* does not solve the *Processor Identification Problem.* Lemma's **5.1.1** and **5.1.3** taken together say that both processors must eventually exit the protocol. If the protocol failed, they both must have exited with the same processor identification number. Which means they both must have exited while occupying the same bin. Label one processor $A$ and the other processor $B$. Processor $A$ must have seen a valid random number in the other bin before leaving. This can only have been written by $B$ since $A$ invalidates bins before leaving them. Since the other bin is

not disturbed in the exiting procedure, by lemma **5.1.2** processor $B$ cannot have changed bins. If processor $B$ has not changed bins, it exited the protocol with the other bin's number as a processor ID. This means both processors must have exited the protocol with unique identification tags. This contradicts our original assumption that the *Random Key Protocol* did not complete successfully. The *Random Key Protocol* therefore solves the *Processor Identification Problem* for the case of two processors. □

If both processors are actively performing the algorithm, then they will choose the same bin with probability 1/2 for each trial. The expected number of trails before successful completion of the algorithm is therefore exactly the same as the asynchronous case, 2. (Although each trial only completes after the slowest processor finishes.)

We now extend this protocol to the general case where $N \geq 2$.

### 5.2 Random Key Protocol: Case N ≥ 2

The *Random Key Protocol* for more than two processors is a generalization of the *Random Key Protocol* for two processors. The general protocol uses $M$ bins (where $M \geq N$). As in the two processor case, each processor initially chooses a large random number (tag) which is assumed to be unique. It then resets the valid bits in each of the $M$ bins to 0. The processor chooses one of the $M$ possible bins at random, and sets the valid bit in the bin before writing its tag to that bin. The processor remains on this bin unless the the bin is disturbed by another processor.

The termination condition is harder formulate in the general case. Even if a processor reads $N$ unique valid tags in the $M$ bins. This does not guarantee that all processors have settled into different bins. A valid tag may be overwritten after it is read, but before all of the other bins have been read. This means reading a processor's valid tag does not guarantee that it will have settled into a unique bin after the all $M$ bins are read.

To solve this problem, each processor keeps a count of the number of times it has changed bins. It writes this number into its bin along with its tag (see Figure 5.2.1). If a processor reads all $M$ bins

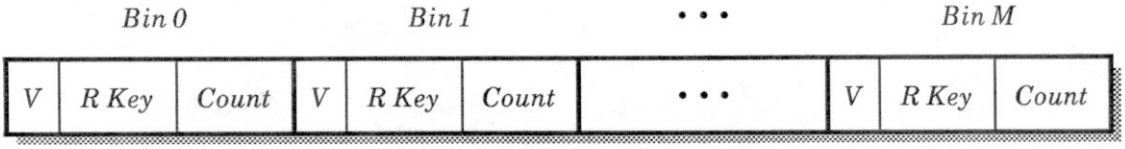


Figure 5.2.1: Shared Memory Organization for $N$ Processor *Random Key Protocol*

twice, and if each occupied bin contains the same valid random number and count value, then it can be assured that for at least some point in time all $M$ processors have settled into unique bins. Once

they have settled in to unique bins, no processor will move, because no processor will subsequently disturb another processor.

The general protocol is stated more formally below:

```
set all valid bits to 0;
r = random();                    /*get random number*/
count = 0;                       /*initialize count*/
do forever {
    i = random() modulo N;       /*select random bin*/
    bin[i] = (1,r,count);        /*write (valid bit, random number, count) to bin i*/
    while(bin[i] = (1,r,count)) {
        read all bins;
        if(N valid random numbers and bin[i] = (1,r,count) ) {
            read all bins;
            if(no change between reads) {
                processorid = i;
                exit(0);
            }
        }
    }
    bin[i] = (0,*,*);            /*reset only the valid bit of bin i*/
    count = count + 1;
}
```

The count value makes it possible for the protocol to successfully determine the termination condition. However, it imposes a limit on the number of times a processor may change bins. If the count value is stored in $L$ bits then the count value will cycle if it is incremented more than $2^L$ -1 times. If the $L$ is made sufficiently large, then all processors will successfully complete the protocol with probability greater than 1-$\alpha$ before changing bins $2^L$ times. (Where $\alpha$ is an arbitrarily small real number less than 1.)

If we assume that processors initially choose unique random keys, and that the *Random Key Protocol* will terminate before any processor's count value exceeds the maximum limit ($2^L$), then we can prove that the *Random Key Protocol* solves the Processor Identity Protocol. We first establish preliminary Lemmas.

**Lemma 5.2.1:** *A processor will only terminate the Random Key Protocol if each processor has a valid bit written to a bin, and no bin is occupied by more than one processor.*

Before exiting the protocol, a processor reads all of the bins in one pass, then reads the bins in a second pass. The processor only exits, if it sees that no bins have changed between reads, and that $N$ valid bits are set.

The count value ensures that the bin has not been altered between read operations, if the count value were not included, it would be possible for a valid key to be overwritten one processor and then rewritten by the original processor between read operations. This sequence of actions will change the count value, and this will be detected on the second read.

If the values remain the same between reads, then at some point in time between the two read passes $N$ bins are guaranteed to contain $N$ set valid bits (see Figure 5.2.2). Since processors erase valid bits before changing bins, there can be at most one valid bit for each processor. The existence of $N$ valid bits means the $N$ processors must have each written one of these valid bits, and that no bin is occupied by more than one processor. □

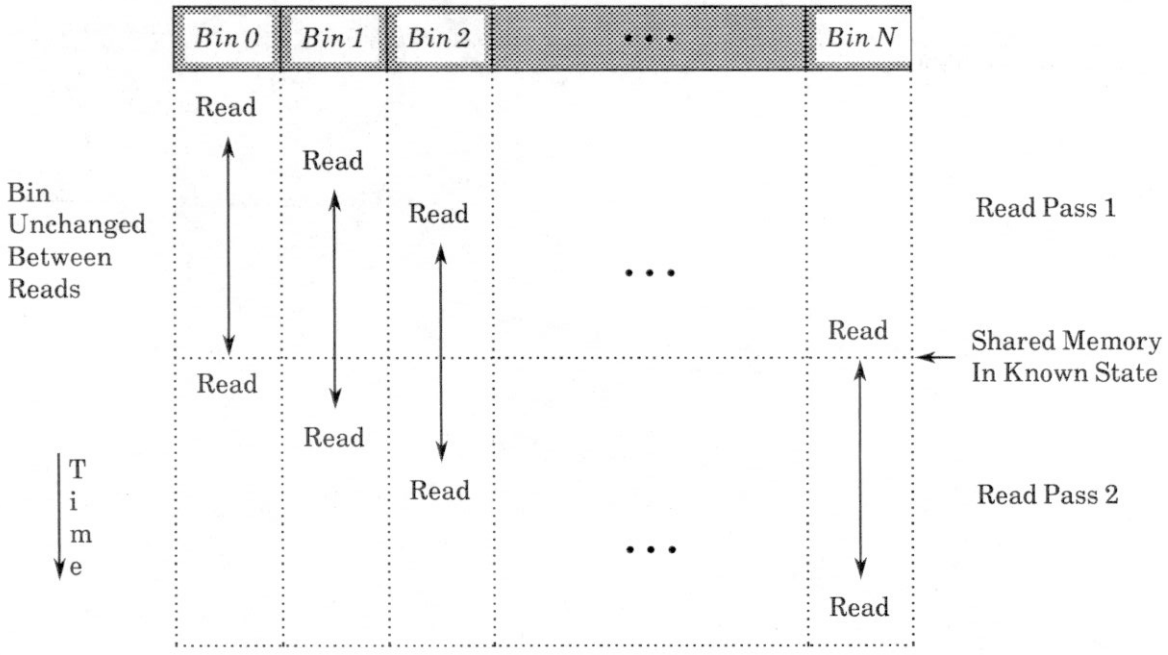


Figure 5.2.2: Termination Condition (Two Read Passes)

**Lemma 5.2.2:** *If each of the N processors has a valid bit written to a different bin in the Random Key Protocol, then no processor will ever subsequently change bins.*

Assume that each processor has a valid bit written to a unique bin, and that one or more processors subsequently move. One of these processors must be the first to move (processor $A$). (If more than one processor moves simultaneously, pick one of these at random and call it processor $A$.). Processor $A$ must have been prompted to move because its bin was written to by another processor (call this processor $B$). This writing must have occurred before all processors had valid bits written to different bins, because $A$ was the first to move afterwards. However, processor $B$ would have first erased $A$'s valid bit before moving to its own bin. This means $A$ must have subsequently rewritten its valid bit, and that processor $B$ could not have subsequently interfered with $A$'s bin. Our assumption that processors move produced a contradiction. Therefore, if each of the $N$ processors has a valid bit written to a different bin in the *Random Key Protocol*, then no processor will ever subsequently change bins. □

We next prove that the all processors will eventually have valid bits written to unique bins.

**Lemma 5.2.3:** *If all N processors are actively performing the Random Key Protocol, then each processor will eventually have a valid bit written to a unique bin.*

If all processors are active and fewer than $N$ valid bits are written to the shared memory, then some processors do not have a valid tag in any bin. These processors will subsequently either write valid tags to empty bins (in which case $N$ valid bits will be set) or they will write to occupied bins, causing collisions and more displaced processors. Even if we assume that each collision causes all processors to change bins, the protocol has a probability $P' = (M!/(M\text{-}N)!)/M^N$ of assigning each processor to a unique bin after each collision. The collisions cannot go on indefinitely because the unique assignment of bins to processors will take place with probability one as the number of collision events goes to infinity. □

**Theorem 5.2.:** *If all processors remain active, the Random Key Protocol will produce a one-to-one assignment of the N processors to the integers {0,1,2, ... ,N-1}.*

Proof: Lemmas **5.2.2** and **5.2.3** taken together say that each processor will eventually settle into its own bin and remain there. The processors will then eventually read $N$ set valid bits on two separate scans of the bins without any of the bins changing. All processors will therefore terminate while occupying different bins. Each processor will therefore choose a unique number from {0,1,2, ... ,$N$-1}. The *Random Key Protocol* will therefore produce a one-to-one assignment of the $N$ processors to the integers {0,1,2, ... ,$N$-1}. □

An issue to be concerned about is the probability of two processors choosing the same random key at the beginning of the protocol. If the random numbers are $L$ bits long, the probability that $N$ processors choose unique random numbers is $(2^L)!/(2^L\text{-}N)!/(2^L)^N$. As in the two processor case, this number can be made arbitrarily close to one by increasing the size of $L$.

The expected number of trials that the protocol will have to complete will be similar to the asynchronous case. (The definition of a "trial" will have to be clarified though. Each processor moves a certain number of times during the protocol. The maximum of this number over all processors will be the number of trials for the protocol.) In the worst case all processors will be forced to move with each new trial. In this case the expected number of trials will be identical to the synchronous case. Expected number of trials = $1/P'$. Where $P' = (M!/(M\text{-}N)!)/M^N$.

**6. Generation of Random Numbers**

It is no trivial task to generate a stream of random numbers from a deterministic computing system. Random number generators are typically a deterministic program which takes as input a random seed. Finding unique seeds for all processors is not a trivial task. If each element of the computing system were truly deterministic, it would clearly be impossible. There are, however, channels through which random processes from the physical world can be be tapped from within a computing system.

On system power up, the contents of random access memory assumes a somewhat random state. Processors typically initialize the entire memory to a known state before proceeding. Before this initialization takes place, a random number can be generated as the result of a function performed one the contents of each local memory. (Remember that each processor possesses its own local memory.) One must make certain that correlations due to spatial locality do not affect the final random number. For instance, one might exclusive-or all bits whose addresses are multiples of the $i$th prime number to produce the $i$th bit of a random number.

A disk drive is a physical device that is affected by random processes. Variations in the power supply, vibrations, variable air friction, and rotational velocity all affect seek and latency times. The time for any given disk seek will be randomly distributed around an expected value. One can use deviations from the expected time of a disk operation as a source for random numbers.

Even if a processor has no disk drives, or uninitialized RAM locations, processors can reference their system clocks for unique random number generators seeds. If all processors are started at different times, and some amount of clock skew is present, then system clocks will suffice for seed generation.

Clock skew may also be used to generate unique random number seeds. The following program uses clock skew between processors to arrive at unique random number generator seeds. It requires one "bit" in shared memory, and one "count" value in local memory.

```
count = 0;
while (bit not = 1){
    if (count = N) bit = 1;
    count = count + 1;
}
seed = count;
```

The first processor to terminate will take on the seed value $N$. slower processors will take on lower values.

Even if all processors are tied to the same clock, and memory is assumed to be initialized, one can provide each processor with a large unique random number seed beforehand. This is not the same with assigning each processor a name from a small set of names that can be referred to by a program. Two processors will not have the same random number seed in practice. Processors can then be replaced without reconfiguration work, or modifications to global tables of processor to name assignments.

In practice there exist reasonable methods of providing each processor with a unique stream of random numbers. In theory some of them may not always work. However in theory, we can assume the existence of a perfect random number generator anyway.

## Conclusions

We have presented and analyzed a set of protocols which solve the Processors Identity Problem. These protocols can be used to greatly enhance system modularity by reducing configuration work associated with installing or replacing individual processor nodes.

## Acknowledgments

We'd like to thank Jonathan Sandberg, K. Balasubramanian, Bill Lin, and Rafael Alonso for their contributions in refining the ideas of this paper.

## References

[Klei75] L. Kleinrock, *Queuing Systems, Volume 1: Theory*, John Wiley & Sons, New York, 1975.

[Lamp86a] L. Lamport, "The Mutual Exclusion Problem: Part I - A Theory of Interprocess Communication", *JACM*, Volume 33, Number 2, April, 1986, pp. 313-326.

[Lamp86b] L. Lamport, "The Mutual Exclusion Problem: Part II - Statement and Solutions", *JACM*, Volume 33, Number 2, April, 1986, pp. 327-348.

[Metc76] R. M. Metcalfe, D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", *Communications of the ACM*, Volume 19, Number 7, July, 1976, pp. 395-404.

[Park87] A. Park, "Design Issues in Shared Memory Multiprocessor Systems", Ph.D. Dissertation, Department of Computer Science, Princeton University, Princeton, New Jersey, 08544, In preparation.

[Rabi82] M. O. Rabin, "The Choice Coordination Problem", *Acta Informatica*, 17, 1982, pp.121-134.

[Tuck80] A. Tucker, *Applied Combinatorics*, John Wiley & Sons, New York, 1980.