*

Programming with Shared Memory

*Material based on B. Wilkinson et al., "PARALLEL PROGRAMMING. Techniques and Applications Using Networked Workstations and Parallel Computers"

Introduction

Figure 1-1 shows a program designed to run as a single process.

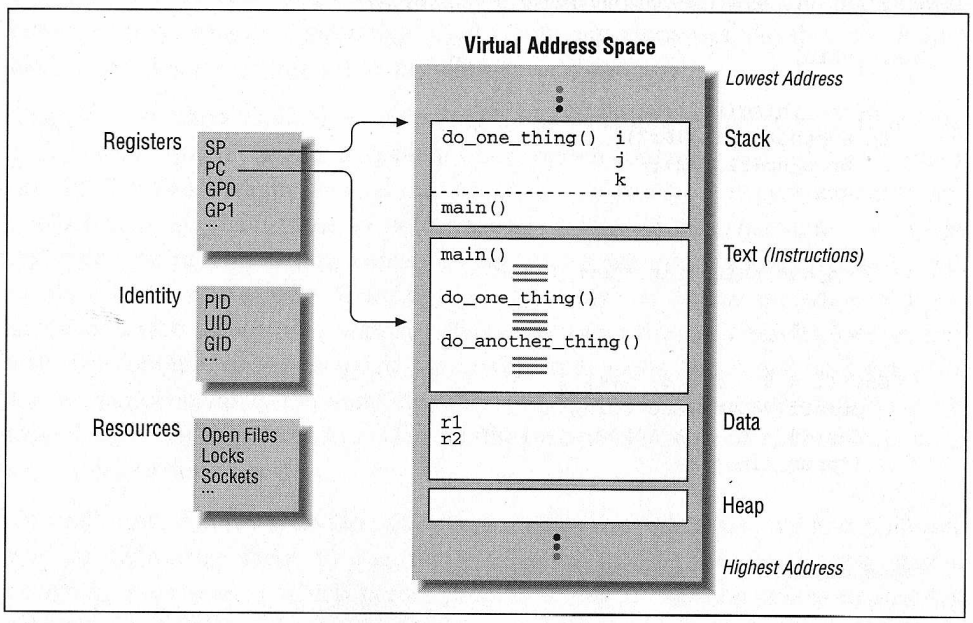

Figure 1–1: The simple program as a process *

*B. Nichols, D. Buttlar, and J. Proulx Farrell, *Pthreads Programming*, O'reilly, 1996.

Anatomy of a Process

A process contains several regions in its memory space.

- Read-only region containing program instructions.

- Read-write region containing global data.

- *Heap* area where from which memory is dynamically allocated using *malloc* systems calls.

- *Stack* containing *automatic variables* for the current function plus function arguments and return address of function that called the current function.

  Below that in the stack, are similar data for the calling function, and so on. Each function-specific area is called a *stack frame.*

Anatomy of a Process Cont.

Additional resources required by a process are:

- Machine registers, in particular the *program counter* (PC) and a pointer (SP) to the current stack frame.

- Tables for the process (handled by the operating system) that keeps track of resources supplied by the system. ie. open files (each with own file descriptor), sockets, system locks, etc.

Multiple Processes

Before threads, we could only divide a program into multiple tasks by creating multiple processes, as in Figure 1-5.

This is done using the *fork()* command that creates a child process identical to its parent except:

- Child is given separate process identifier (PID).

- The value returned by fork() call is different than that returned to parent.

Lot of overhead to create process as everything duplicated. Also, data space isn't shared, so harder to communicate.

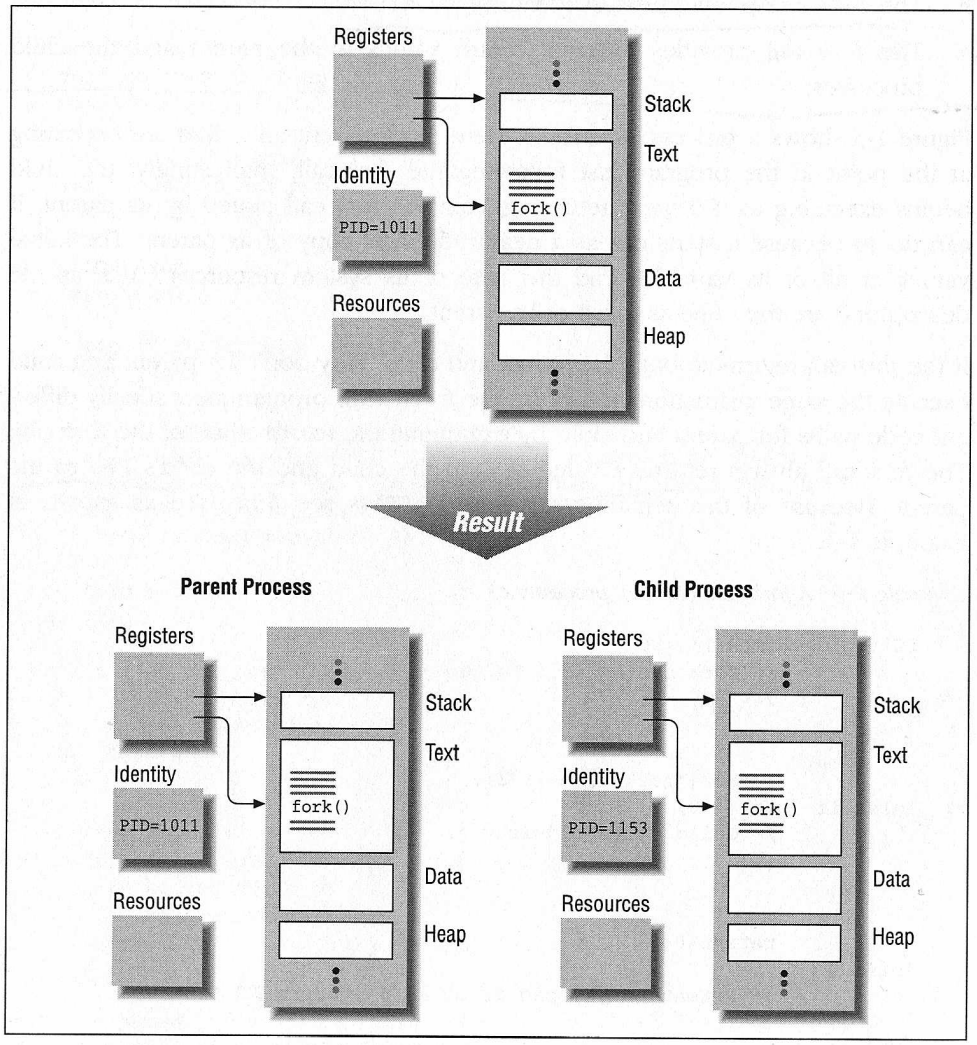

Figure 1–5: A program before and after a fork

*

*B. Nichols, D. Buttlar, and J. Proulx Farrell, *Pthreads Programming*, O'reilly, 1996.

Threads

Threads give us a more efficient way to divide our program.

With threads, multiple subtasks in a program are implemented as separate control streams in a single process.

In the threads model, we break a process into two parts:

- Program-wide resources such as global data and program instructions. This portion is referred to as the process.

- Information pertaining to the execution state of control stream, such as the PC and the stack. We refer to this part as the *thread.*

Figure 1-3 illustrates this. Threads are more efficient than multiple processes because more of the program is shared.

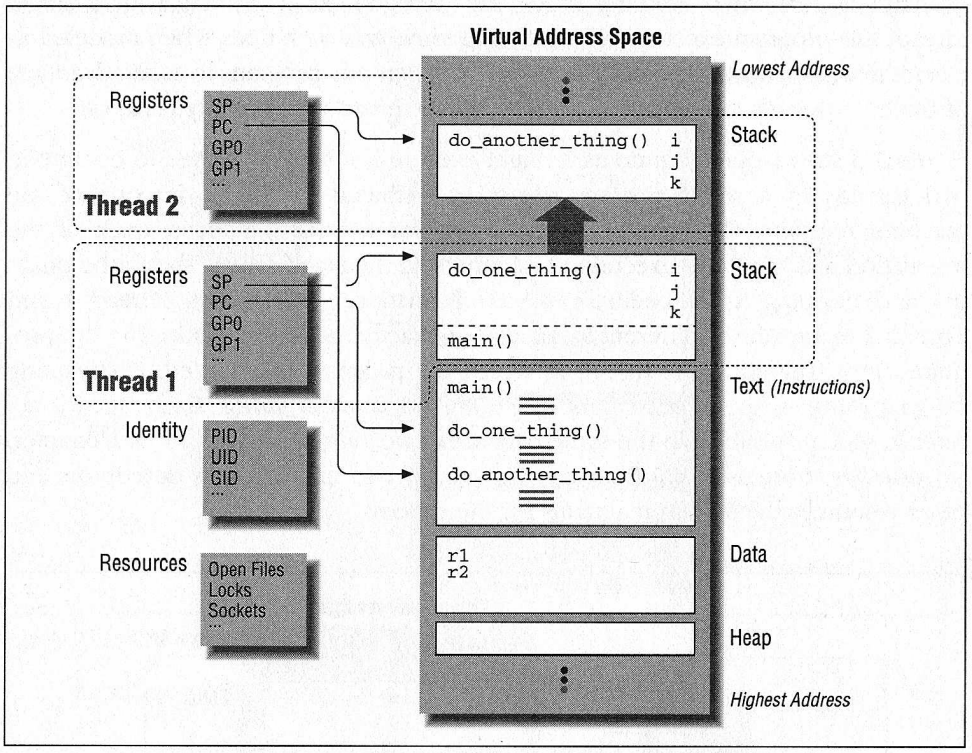

Figure 1–3: The simple program as a process with multiple threads *

*B. Nichols, D. Buttlar, and J. Proulx Farrell, *Pthreads Programming*, O'reilly, 1996.

Pthreads

Pthreads is the POSIX standard for multithreaded programming.

Its purpose to provide a portable application programming interface for threads as an alternative to the many platform-specific APIs.

Requires: #include <pthread.h>

To compile program "prog.c:"

```
gcc -o prog prog.c -lpthread  -D_REENTRANT
```

On Linux, the "-D_REENTRAN" activates a macro that replaces certain library functions (ie. readdir, localtime, ctime etc.) with full reentrant versions specifed by Pthreads (ie. readdir_r, localtime_r, ctime_r etc.).

On Solaris? don't know. Safer to be specific.

Creating Threads

To create a new thread, use:

```
pthread_create(threadID, thd_attr, routine, arg)

threadID - A pointer to a buffer in which an identifier
         for the thread is placed.
thd_attr - A pointer to a structure referred to as a
           thread attribute object.  If you pass NULL
           pointer, means you get default characteristics.
routine - A pointer to routine at which new thread will
          start executing.

          void *(*routine)  (void *)
arg - A pointer to the parameter to be passed to routine.

      void *arg
```

Synchronizing Threads Exit

Common practice is to start threads to perform certain tasks, and then have them exit.

Once these tasks are complete, we want to perform additional tasks.

This synchronization is achieved by using:

```
pthread_join(threadID, value)

threadID - An identifier for the thread.
value - Value is a pointer to the data returned by the thread
        represented by threadID. Normally pass NULL pointer as
        communication occurs via global variables.

        void **value
```

When *pthread_join()* is called, it doesn't return until the thread identified by threadID has exited.

A thread exits when its starting routine completes, or when it calls *pthread_exit().*

Protecting Data with Mutexes

As threads from the same process share the same global data space, they normally use global variables to communicate.

Variables used for communication are declared before *main()* (outside any function) so that all threads can easily access them by name.

Must protect data from race conditions such as:

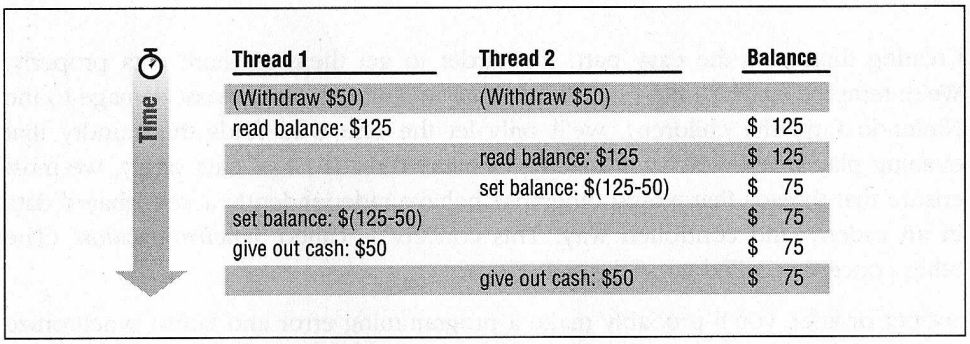

Figure 3–1: ATM race condition with two withdraw threads

*

*B. Nichols, D. Buttlar, and J. Proulx Farrell, *Pthreads Programming*, O'reilly, 1996.

Protecting Data with Mutexes Cont.

We use a form of synchronization called *mutual exclusion (mutex)* to prevent race conditions.

Mutexes are used to provide a thread with exclusive access to a shared resource for a time.

In the previous example, the first thread would have locked mutex for the balance variable before reading it.

This would have caused the other thread to wait until the lock was removed before trying to read the balance.

When the first thread had updated the balance, it would remove the lock and the second thread would then access the balance variable.

Using Pthreads Mutexes

For Pthreads, a specific mutex is defined as a variable of type *pthread_mutex_t*.

To use it, you would:

1. Initialize and create a mutex for all variables you want to protect.

2. Use function *pthreads_mutex_lock()* to lock the mutex of the resource that you want to access.

   Pthreads library guarantees that only one thread can lock a mutex at a given time.

   Other calls to *pthreads_mutex_lock()* for this mutex will block until mutex is unlocked.

3. Unlock mutex using function *pthreads_mutex_unlock()* when finished with resource.

Mutex Functions

Create and initialize mutex *TheMutex*:

```
pthread_mutex_t TheMutex=PTHREAD_MUTEX_INITIALIZER;
```

To lock mutex:

```
pthreads_mutex_lock(&TheMutex)
```

```
TheMutex - Pointer to mutex variable to be locked.
```

To unlock mutex:

```
pthreads_mutex_unlock(&TheMutex)
```

```
TheMutex - Pointer to mutex variable to be unlocked.
```

NOTE: Pthreads libraries don't enforce locks.
A thread could just not call *pthreads_mutex_lock()* and go ahead and use resource.

You have to make sure this doesn't happen.

pthreadseg.c

```
/*  This example base on code from:
 *  B. Nichols, D. Buttlar, and J. Proulx Farrell,
 *  "Pthreads Programming," O'reilly, 1996.  */

#include <stdio.h>
#include <pthread.h>

void do_another_thing(int *);
void do_one_thing(int *);
void do_wrap_up(int, int);

int r1 = 0, r2 = 0, r3 = 0;
pthread_mutex_t r3_mutex=PTHREAD_MUTEX_INITIALIZER;

int main (int argc, char **argv) {

  pthread_t thread1, thread2;
  pthread_attr_t custom_sched_attr;

  /* set scheduling policy to system so braindead solaris
     doesn't run threads sequentially */
  pthread_attr_init(&custom_sched_attr);
  pthread_attr_setscope(&custom_sched_attr,
                        PTHREAD_SCOPE_SYSTEM);
  r3 = 34;

  /* start two threads */
  pthread_create(&thread1, &custom_sched_attr,
                 (void *) do_one_thing, (void *) &r1);
```

```
  pthread_create(&thread2, &custom_sched_attr,
                 (void *) do_another_thing, (void *) &r2);

  /* wait till threads complete */
  pthread_join(thread1,NULL);
  pthread_join(thread2,NULL);

  /* print results */
  do_wrap_up(r1,r2);
  return 0;
}

void do_one_thing(int *pnum_times)
{
  int i,j,x;

  /* protect access to r3 */
  pthread_mutex_lock(&r3_mutex);
  if (r3 >0) {
    x = r3;
    r3 --;
  } else {
    x = 1;
  }
  pthread_mutex_unlock(&r3_mutex);
```

```
  for (i = 0; i < 4;i++) {
    printf("doing one thing\n");
    for (j = 0; j < 10000000; j++)
      x = x + i;
    (*pnum_times)++;
  }
}

void do_another_thing(int *pnum_times)
{
  int i,j,x;

  /* protect access to r3 */
  pthread_mutex_lock(&r3_mutex);
  if (r3 >0) {
    x = r3;
    r3 --;
  } else {
    x = 1;
  }
  pthread_mutex_unlock(&r3_mutex);

  for (i = 0; i < 4;i++) {
    printf("doing another thing\n");
    for (j = 0; j < 10000000; j++)
      x = x + i;
    (*pnum_times)++;
  }
}
```

```
void do_wrap_up(int first, int second) {

  int total;

  total = first+second;
  printf("Wrap up: one thing %d, another %d,
         total %d, r3 %d\n",first,second, total, r3);
}
```

Condition Variables

Condition variables allow threads to synchronize on the value of a variable.

If only had mutexes, would have to poll value. Inefficient and problematic.

A condition variable provides a means of notification between threads that a certain condition has occurred (ie. a counter has reached 12).

A condition variable is used in conjunction with the mutex that protects the data.

Using Condition Variables

Once you initialize a condition variable, a thread can do two things:

- Wait on the condition variable. This is done by calling functions *pthread_cond_wait()* or *pthread_cond_timedwait()*.
  Both suspend the calling thread until another thread *signals* (see next item) using the condition variable.

- Signal other threads that are waiting on the condition variable.

  This is done using functions *pthread_cond_signal()* (wake one) or *pthread_cond_broadcast()* (wake all).

A signal isn't saved. If no thread is waiting, it's lost!

Must check condition not met before waiting on variable, or may block.

Condition Variable Functions

To initialize condition variable TheCV:

```
pthread_cond_t TheCV = PTHREAD_COND_INITIALIZER;
```

Assume we are watching a counter variable which has mutex *countMX* and conditional variable *countCV*. To wait on *countCV*, we would first lock *countMX* and then call:

```
pthread_cond_wait(&countCV, &countMX)

countCV - The conditional variable to wait on.
countMX - The mutex for the associated resource\
          for which we are waiting to reach some state.

          The mutex MUST be locked before the function is call
```

The function will release the mutex for the thread so that other threads can access the resource.

When another thread signals on the condition variable and this thread wakes up, it is returned the lock on the mutex before it continues to execute.

Condition Variable Functions Cont.

To signal on condition variable *countCV* and wake up a waiting thread (if any)

```
pthread_cond_signal(&countCV)

countCV - Wake one thread (if any waiting) that is waiting on
condition variable.
```

The thread would have first locked mutex *countMX* before calling *pthread_cond_signal()*. It would then unlock the mutex.

Once the mutex was unlocked, the lock would be given to the thread waiting, and it would start executing.

condvareg.c

```
/*  This example base on code from:
 *  B. Nichols, D. Buttlar, and J. Proulx Farrell,
 *  "Pthreads Programming," O'reilly, 1996. */

#include <stdio.h>
#include <pthread.h>
#define TCOUNT 10
#define WATCH_COUNT 12

void watch_count(int *);
void inc_count(int *);

int count = 0;
pthread_mutex_t count_mutex=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t count_threshold_cv
                  = PTHREAD_COND_INITIALIZER;
int thread_ids[3] = {0,1,2};

int main (int argc, char **argv) {

  int i;
  pthread_t threads[3];
  pthread_attr_t custom_sched_attr;

  /* set scheduling policy to system so braindead
     solaris doesn't run threads sequentially */
  pthread_attr_init(&custom_sched_attr);
  pthread_attr_setscope(&custom_sched_attr,
                        PTHREAD_SCOPE_SYSTEM);
```

```
  /* start threads */
  pthread_create(&threads[0], &custom_sched_attr,
        (void *) inc_count, (void *) &thread_ids[0]);
  pthread_create(&threads[1], &custom_sched_attr,
        (void *) inc_count, (void *) &thread_ids[1]);
  pthread_create(&threads[2], &custom_sched_attr,
        (void *) watch_count, (void *) &thread_ids[2]);

  /* wait for threads to complete */
  for (i = 0; i < 3; i++) {
    pthread_join(threads[i],NULL);
  }

  return 0;
}

void watch_count(int *idp)
{
  pthread_mutex_lock(&count_mutex);
  printf("watch_count(): Starting thread %d,
         count is %d\n",*idp, count);

  while(count < WATCH_COUNT) {
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch_count(): Thread %d, count is
           %d\n",*idp, count);
  }
  pthread_mutex_unlock(&count_mutex);
}
```

```
void inc_count(int *idp)
{
  int i,j,x;

  x = 1;

  for (i = 0; i < TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;
    printf("inc_count(): Thread %d, old count %d,
            new count %d\n",*idp, count -1, count);
    if (count == WATCH_COUNT)
      pthread_cond_signal(&count_threshold_cv);
    pthread_mutex_unlock(&count_mutex);

    /* do some work to allow another thread to run */
    for (j = 0; j < 10000000; j++)
      x = x + i;
  }
}
```

Dependency Analysis

For parallel programming, it's important to be able to identify which processes can be executed concurrently.

Two processes can not be executed concurrently if there is a dependency between them requiring that they must run sequentially.

Dependency analysis is the process of identifying the dependencies in a program.

In the example below, it's easy to see that each instance of the loop body is independent.

```
forall (i = 0; i < 5; i++)
   a[i] = 0;
```

It's not so easy to see that each instance of the loop body below is independent.

```
forall (i = 2; i < 6; i++) {
   x = i - 2*i + i*i;
   a[i] = a[x];
}
```

Bernstein's Conditions

Want to develop an algorithmic way to identify dependencies.

Bernstein's conditions are sufficient conditions to decide if two processes can be executed concurrently.

They refer to the memory locations used by the process for variables that the process reads/writes to.

We define two sets of memory locations, I and O, for input and output operations.

Let I_i be the set of memory locations that process P_i reads from. Let O_i be the set of memory locations that process P_i writes to.

Bernstein's Conditions Cont.

To be able to run processes P_1 and P_2 concurrently, they must satisfy:

$$I_1 \cap O_2 = \emptyset$$
$$I_2 \cap O_1 = \emptyset$$
$$O_1 \cap O_2 = \emptyset$$

These three conditions are the *Bernstein conditions.*

If all three satisfied, it's safe to run the two process simultaneously.

When applied to a single assignment statement, then variables to the right of the assignment operator correspond to I_i. Variables on the left correspond to O_i.

Bernstein's Conditions Example

Take each statement as a process:

```
1) a = x + y;
2) b = x + z;
```

This gives us:

$$I_1 = (x, y) \quad I_2 = (x, z)$$
$$O_1 = (a) \quad O_2 = (b)$$

Which satisfies:

$$I_1 \cap O_2 = \emptyset$$
$$I_2 \cap O_1 = \emptyset$$
$$O_1 \cap O_2 = \emptyset$$

However, the statements:

```
1) a = x + y;
2) b = a + b;
```

we have $I_2 \cap O_1 \neq \emptyset$, which violates condition 2.

Bernstein's Conditions and Compilers

Multiple statements can be grouped together as one process, and compared to another group of statements using this method.

Conditions are very general and can be automated in a compiler.

Can be used to determine instruction-level parallelism or function-level parallelism.

Assuming the functions do not access global variables, the inputs are the parameters passed to the functions, and the outputs are the return values.

Exploiting Natural Parallelism

We can exploit the natural parallelism in programming constructs such as loops.

```
for (i = 0; i <= 20; i++)
   a[i] = b[i];
```

Expands as below. Clearly, they satisfy Bernstein's conditions and thus could be executed concurrently by 20 processors.

```
   a[1] = b[1];
        .
        .
   a[20] = b[20];
```

The dependencies in the loop below can be handled by breaking the loop into two.

```
for (i = 3; i <= 20; i++)
   a[i] = a[i-2] + 4;
```

This evaluates as:

```
   a[3] = a[1] + 4;
   a[4] = a[2] + 4;
        .
        .
   a[19] = a[17] + 4;
   a[20] = a[18] + 4;
```

Exploiting Natural Parallelism Cont.

We can decompose the computation into two independent streams:

```
   a[3]  = a[1] + 4;        a[4]  = a[2] + 4;
   a[5]  = a[3] + 4;        a[6]  = a[4] + 4;
         .                        .
         .                        .
   a[17] = a[15] + 4;       a[18] = a[16] + 4;
   a[19] = a[17] + 4;       a[20] = a[18] + 4;
```

We can thus rewrite as the following two independent loops:

```
i = 3;
for (j = 1; j <= 9; j++) {
   a[i] = a[i-2] + 4;
   i = i + 2;
}
```

and:

```
i = 4;
for (j = 1; j <= 9; j++) {
   a[i] = a[i-2] + 4;
   i = i + 2;
}
```

Shared Data and Memory Caches

Typically in shared-memory multiprocessors, each processor has an attached high speed cache.

When a processor accesses a memory location, it fills a cache line with the data and its neighbours.

If another processor loads the same data, we have a potential coherency problem as each have a local copy of the data.

If they only read the data, no problem.

If one processor alters the copy in their cache, the copy in the other processors cache must either be updated or invalidated.

This is handled by the systems *cache coherence protocol.*

Shared Data and Memory Caches Cont.

Cache coherency protocols use one of two approaches:

Update Policy: This is when the copies of data in all caches are updated, when the data in one cache is modified.

Invalidate Policy: One a copy of data is modified in one cache, the copies in all other caches are marked invalid by setting a "valid" bit.

The data in a specific cache is only updated when the associated processor actually references it.

This policy is more common.

False Sharing of Data

When data Is loaded into a cache, it is loaded in a block.

Normally, this improves access time as data access usually has temporal and spatial locality.

This can cause a problem for multiprocessor systems as different processors might access data in the same block, but NOT the *same* data.

When one processor accesses data in one block, the copies in the other caches must be updated or invalidated even though data is not really being shared.

Called *false sharing*, and can reduce performance. See Figure 8.9.

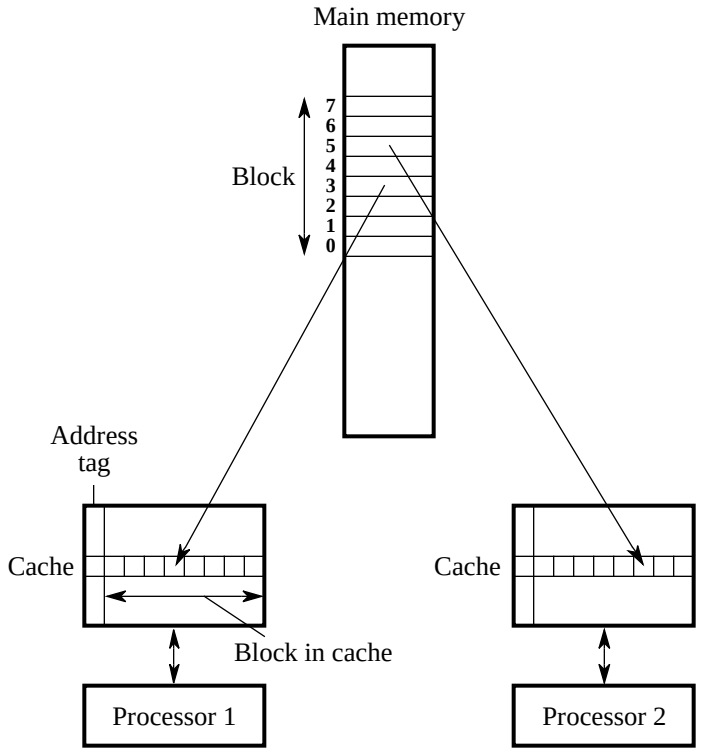

Figure 8.9 False sharing in caches.

False Sharing of Data Cont.

If system supports, can bypass the cache.

Another solution is to have compiler separate data accessed by different processors.

May not be easy to do. For example:

```
forall (i = 0; i < 5; i++)
   a[i] = 0;
```

This will probably cause false sharing as a[0], a[1], etc are going to be stored sequentially.

Would have to put each value in separate block.

Could be handled by SIMD computer.

Encryption*

To encrypt a message is to encipher the message using an algorithm such that you need a secret piece of information (called a key) to recover the original message.

A simple cipher for text messages would be to replace each letter of the alphabet by another. ie. to create a unique 1 to 1 mapping. This is an example of a substitution cypher.

say: $a \to c \quad b \to d \quad c \to e$

If you know the mapping, it is easy to reverse the process and recover the original message.

The original message is called the *plaintext* and the encrypted message is called the *ciphertext.*

*This discussion is based on information from: Simon Singh, *The Code Book*, Anchor Books, 1999.

Key Distribution

Typically, encryption methods are symmetric. This means the same key is used to encrypt the message and to decrypt the message.

Key distribution is the biggest problem. How do you securely get the key to the other person so they can decrypt the message?

Imagine if the key changed daily, and you had to swap keys with thousands of people?

How can this be done securely, easily, and cheaply?

It was taken as a given that two people must first exchange a key before they can send each other encrypted messages.

Key Exchange Method

In 1976 , this was challenged by Whitfield Diffey, Martin Hellman, and Ralph Merkle. They came up with the *Diffey-Hellman-Merkle key exchange Scheme.*

Allowed *Bob* and *Alice* to securely exchange a private key without meeting and in public, without *Eve* intercepting the key.

They agreed on a *one-way function* (a function that is very difficult to reverse) and each choose a private piece of information. They each applied it to the function, and sent the results to the other.

They then use the secret, the function, and the received info to compute a new private key that they then use to encrypt messages.

Key Exchange Method Cont.

The final key can only be computed if one knows one of the original two secrets that were never exchanged.

They have securely exchanged a private key without meeting, but they must first exchange several pieces of information. This requires that they be available to do this.

Requires coordination and contact.

Public Key Encryption

For symmetric encryption, people must securely exchange a secret key.

What if the encryption was asymetric? if the message was encrypted with one key, but must be decrypted with another.

That knowing the encryption key (called *public key*), it's very difficult to determine the decryption key (*private key*)?

Alice could publish her public key in something like a telephone book. If Bob wants to send her an encrypted message, he looks up Alice's public key, and uses it to encrypt the message.

Now, only someone knowing Alice's private key can easily decrypt the message.

This was the flash of inspiration of Whitfield Diffie in 1975. He came up with the idea, but was unable to come up with an implementation.

RSA Public Key Encryption

In 1977, Ron Rivest, Adi Shamir, and Leonard Adleman (RSA) came up with the RSA encryption method.

Utilizes a one-way function based on modular arithmetic.

In modular arithmetic, we have n numbers, say for $n = 7$, 0 to 6. To add two numbers *mod 7*, it's as if we are going around a clock with those numbers.

For $2 + 3$*(mod 7)* we start at 2 and go around clock 3 positions to 5. However, $2 + 6$*(mod 7)* we get 1!

If we knew that one operand is 2 and the answer is 1, that doesn't tell us what the other operand is as many values generate 1.

RSA Public Key Encryption Cont

To know an answer *(mod x)*, we can calculate the normal answer, then divide by x. The remainder is the answer *(mod x)*.

If you were told that $3^x = 9$, it'd be easy to guess $x = 2$, but what about 3^x *(mod 7)* $= 1$?

To reverse a function in modular arithmetic, one usually has to generate a table by calculating the function for many values of x until you find one that works.

How RSA Works

Alice picks a private key consisting of two very large (say 256 bits) prime numbers, which we will label p and q.

Her public key is $N = p*q$ and a third number e chosen such that e, $(p-1)\times(q-1)$ are *relatively prime* (two numbers that share no factors in common except 1).

Alice publishes N and e but keeps p and q secret.

If bob wants to encrypt the number M to send to Alice, he calculates C (ciphertext) by:

$$C = M^e (mod\ N)$$

How RSA Works Cont.

To decrypt message, Alice must first calculate the number d which satisfies:

$$e \times d (mod\ (p-1) \times (q-1)) = 1$$

Knowing p, q, e this can be easily calculated using *Euclid's algorithm* (See for details : "Applied Cryptography second edition: protocols, algorithms, and source code in C" by Bruce Schneier, 1996. John Wiley & Sons, Inc.)

She can then decrypt the message using the following formula:

$$M = C^d (mod\ N)$$

Knowing N allows you to encrypt the message, but can only decrypt if know p and q.

Simple RSA Example

Alice chooses $p = 17$, $q = 11$ and $e = 7$.

This makes Alice's public key $N = 17 \times 11 = 187$ and $e = 7$.

Bob wants to send the letter X to Alice. ASCII for X is $(1011000)_2 = (88)_{10}$. Thus, $M = 88$.

Our Cypher text is:

$$C = M^e (mod\ N) = 88^7 (mod\ 187) = 11$$

ALice receives $C = 11$ and wants to decrypt it.

Simple RSA Example Cont.

She uses Euclid's algorithm to evaluate:

$$7 \times d(mod\ (16) \times (10)) = 1$$

and finds $d = 23$.

She then decrypts the message as follows:

$$M = C^d(mod\ N) = 11^{23}(mod\ 187) = 88$$

Knowing that 88 is ASCII for X, Alice has the original message.

Breaking RSA

Given a person's public key (N and e), how can we determine their private key (p and q) so we can decrypt the message?

We know that $N = p * q$ and that p and q are prime. That means that p and q are the only factors of N other than 1 and N itself.

If we can factor N, we get p and q.

If we start at $i = 2$ and divide N by i (increment by 1 after each try) until it divides evenly (no remainder), then i and our quotient are p and q.

An approach called *general number field sieve* is faster, but still takes a prohibitively long time for large N.

Speedup Anomalies

The simplest way to parallelize this is to simply divide up the search space.

It could be that the problem is divided such that one processor will find the key almost immediately and thus speedup will be greater than linear.

Referred to as an *acceleration anomaly.*

It is also possible that a single processor might stumble upon the key right away, but the search space for multiple processors could be divided such that no process will find the key quickly; thus speedup is less than linear.

Referred to as an *deceleration anomaly.*

If the speedup is actually less than one, it is called a *detrimental anomaly.*