Programming with Shared Memory

In a shared memory system, any memory location can be accessible by any of the processors.

A *single address space* exists, meaning that each memory location is given a unique address within a single range of addresses.

For a small number of processors, a common architecture is the single bus architecture, in

![Shared memory multiprocessor using a single bus.](image)

**Figure 8.1** Shared memory multiprocessor using a single bus.
Several Alternatives for Programming:

• Using a new programming language
• Modifying an existing sequential language
• Using library routines with an existing sequential language
• Using a sequential programming language and ask a parallelizing compiler to convert it into parallel executable code.
• UNIX Processes
• Threads (Pthreads, Java, ..)
<table>
<thead>
<tr>
<th>Language</th>
<th>Originator/date</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrent Pascal</td>
<td>Brinch Hansen, 1975&lt;sup&gt;a&lt;/sup&gt;</td>
<td>Extension to Pascal</td>
</tr>
<tr>
<td>Ada</td>
<td>U.S. Dept. of Defense, 1979&lt;sup&gt;b&lt;/sup&gt;</td>
<td>Completely new language</td>
</tr>
<tr>
<td>Modula-P</td>
<td>Bräunl, 1986&lt;sup&gt;c&lt;/sup&gt;</td>
<td>Extension to Modula 2</td>
</tr>
<tr>
<td>C*</td>
<td>Thinking Machines, 1987&lt;sup&gt;d&lt;/sup&gt;</td>
<td>Extension to C for SIMD systems</td>
</tr>
<tr>
<td>Concurrent C</td>
<td>Gehani and Roome, 1989&lt;sup&gt;e&lt;/sup&gt;</td>
<td>Extension to C</td>
</tr>
<tr>
<td>Fortran D</td>
<td>Fox et al., 1990&lt;sup&gt;f&lt;/sup&gt;</td>
<td>Extension to Fortran for data parallel programming</td>
</tr>
</tbody>
</table>


<sup>d</sup> Thinking Machines Corp. (1990), *C* Programming Guide, Version 6, Thinking Machines System Documentation.


Constructs for Specifying Parallelism

Creating Concurrent Processes – FORK-JOIN Construct

Have been applied as extensions to FORTRAN and to the UNIX operating system.

A `FORK` statement generates one new path for a concurrent process and the concurrent processes use `JOIN` statements at their ends.

When both `JOIN` statements have been reached, processing continues in a sequential fashion.

![Diagram of FORK-JOIN construct](image-url)

**Figure 8.2** FORK-JOIN construct.
UNIX Heavyweight Processes

The UNIX system call `fork()` creates a new process. The new process (child process) is an *exact copy* of the calling process except that it has a unique process ID. It has its own copy of the parent’s variables. They are assigned the same values as the original variables initially.

The forked process starts execution at the point of the fork.

On success, `fork()` returns 0 to the child process and returns the process ID of the child process to the parent process.

Processes are “joined” with the system calls `wait()` and `exit()` defined as

```c
wait(statusp); /*delays caller until signal received or one of its */
    /*child processes terminates or stops */
exit(status);  /*terminates a process. */
```

Hence, a single child process can be created by

```c
    pid = fork();  /* fork */
    Code to be executed by both child and parent
    if (pid == 0) exit(0); else wait(0);  /* join */
```

If the child is to execute different code, we could use

```c
    pid = fork();
    if (pid == 0) {
        code to be executed by slave
    } else {
        Code to be executed by parent
    }
    if (pid == 0) exit(0); else wait(0);
```
Threads

The process created with UNIX fork is a “heavyweight” process; it is a completely separate program with its own variables, stack, and memory allocation.

A much more efficient mechanism is a thread mechanism which shares the same memory space and global variables between routines.

![Figure 8.3 Differences between a process and threads.](image-url)
Pthreads

IEEE Portable Operating System Interface, POSIX, section 1003.1 standard

Executing a Pthread Thread

Main program

```c
pthread_create(&thread1, NULL, proc1, &arg);

pthread_join(thread1, *status);

proc1(&arg)
{
    return(*status);
}
```

Figure 8.4  `pthread_create()` and `pthread_join()`.
Pthread Barrier

The routine `pthread_join()` waits for one specific thread to terminate.

To create a barrier waiting for all threads, `pthread_join()` could be repeated:

```c
for (i = 0; i < n; i++)
    pthread_create(&thread[i], NULL, (void *) slave, (void *) &arg);

for (i = 0; i < n; i++)
    pthread_join(thread[i], NULL);
```
Detached Threads

It may be that a thread may not be bothered when a thread it creates terminates and in that case a join would not be needed.

Threads that are not joined are called *detached threads*.

When detached threads terminate, they are destroyed and their resource released.

![Diagram of detached threads](image-url)

*Figure 8.5* Detached threads.
Statement Execution Order

Once processes or threads are created, their execution order will depend upon the system.

On a single processor system, the processor will be time shared between the processes/threads, in an order determined by the system if not specified, although typically a thread executes to completion if not blocked.

On a multiprocessor system, the opportunity exists for different processes/threads to execute on different processors.

The instructions of individual processes/threads might be interleaved in time.

Example

If there were two processes with the machine instructions

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction 1.1</td>
<td>Instruction 2.1</td>
</tr>
<tr>
<td>Instruction 1.2</td>
<td>Instruction 2.2</td>
</tr>
<tr>
<td>Instruction 1.3</td>
<td>Instruction 2.3</td>
</tr>
</tbody>
</table>

there are several possible orderings, including

- Instruction 1.1
- Instruction 1.2
- Instruction 2.1
- Instruction 1.3
- Instruction 2.2
- Instruction 2.3

assuming an instruction cannot be divided into smaller interruptible steps.

If two processes were to print messages, for example, the messages could appear in different orders depending upon the scheduling of processes calling the print routine.

Worse, the individual characters of each message could be interleaved if the machine instructions of instances of the print routine could be interleaved.
Compiler Optimizations

In addition to interleaved execution of machine instructions in processes/threads, a compiler (or the processor) might reorder instructions of your program for optimization purposes while preserving the logical correctness of the program.

Example

The statements

\[
\begin{align*}
a &= b + 5; \\
x &= y + 4;
\end{align*}
\]

could be compiled to execute in reverse order:

\[
\begin{align*}
x &= y + 4; \\
a &= b + 5;
\end{align*}
\]

and still be logically correct.

It may be advantageous to delay statement \( a = b + 5 \) because some previous instruction currently being executed in the processor needs more time to produce the value for \( b \).

It is very common for modern processors to execute machines instructions out of order for increased speed of execution.
Thread-Safe Routines

System calls or library routines are called thread safe if they can be called from multiple threads simultaneously and always produce correct results.

Example

Standard I/O is designed to be thread safe (and will print messages without interleaving the characters).

Routines that access shared data and static data may require special care to be made thread safe.

Example

System routines that return time may not be thread safe.

The thread-safety aspect of any routine can be avoided by forcing only one thread to execute the routine at a time. This could be achieved by simply enclosing the routine in a critical section but this is very inefficient.
Accessing Shared Data

Consider two processes each of which is to add one to a shared data item, $x$.

It will be necessary for the contents of the $x$ location to be read, $x + 1$ computed, and the result written back to the location.

With two processes doing this at approximately the same time, we have

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = x + 1;$</td>
<td>read $x$</td>
<td>read $x$</td>
</tr>
<tr>
<td></td>
<td>compute $x + 1$</td>
<td>compute $x + 1$</td>
</tr>
<tr>
<td></td>
<td>write to $x$</td>
<td>write to $x$</td>
</tr>
</tbody>
</table>

Figure 8.6 Conflict in accessing shared variable.
Critical Section

The problem of accessing shared data can be generalized by considering shared resources.

A mechanism for ensuring that only one process accesses a particular resource at a time is to establish sections of code involving the resource as so-called critical sections and arrange that only one such critical section is executed at a time.

The first process to reach a critical section for a particular resource enters and executes the critical section.

The process prevents all other processes from their critical sections for the same resource.

Once the process has finished its critical section, another process is allowed to enter a critical section for the same resource.

This mechanism is known as mutual exclusion.
Locks

The simplest mechanism for ensuring mutual exclusion of critical sections.

A lock is a 1-bit variable that is a 1 to indicate that a process has entered the critical section and a 0 to indicate that no process is in the critical section.

The lock operates much like that of a door lock.

A process coming to the “door” of a critical section and finding it open may enter the critical section, locking the door behind it to prevent other processes from entering. Once the process has finished the critical section, it unlocks the door and leaves.

Spin Lock

Example

```c
while (lock == 1) do_nothing; /* no operation in while loop */
lock = 1; /* enter critical section */
    critical section
lock = 0; /* leave critical section */
```

![Diagram](Image)

**Figure 8.7** Control of critical sections through busy waiting.
Pthread Lock Routines

Locks are implemented in Pthreads with what are called *mutually exclusive lock* variables, or “mutex” variables.

To use a mutex, first it must be declared as of type `pthread_mutex_t` and initialized, usually in the “main” thread:

```c
    pthread_mutex_t mutex1;
    .
    .
    pthread_mutex_init(&mutex1, NULL);
```

NULL specifies a default attribute for the mutex.

A mutex can be destroyed with `pthread_mutex_destroy()`.

A critical section can then be protected using `pthread_mutex_lock()` and `pthread_mutex_unlock()`:

```c
    pthread_mutex_lock(&mutex1);
    .
    critical section
    .
    pthread_mutex_unlock(&mutex1);
```

If a thread reaches a mutex lock and finds it locked, it will wait for the lock to open.

If more than one thread is waiting for the lock to open when it opens, the system will select one thread to be allowed to proceed.

Only the thread that locks a mutex can unlock it.
Deadlock

Deadlock can occur with two processes when one requires a resource held by the other, and this process requires a resource held by the first process.

Deadlock can also occur in a circular fashion with several processes having a resource wanted by another.

These particular forms of deadlock are known as **deadly embrace**.

Given a set of processes having various resource requests, a circular path between any group indicates a potential deadlock situation.

Deadlock can be eliminated between two processes accessing more than one resource if both processes make requests first for one resource and then for the other.

**Pthreads**

Offers one routine that can test whether a lock is actually closed without blocking the thread — *pthread_mutex_trylock()* . This routine will lock an unlocked mutex and return 0 or will return with *EBUSY* if the mutex is already locked – might find a use in overcoming deadlock.
Semaphores

A semaphore, \( s \) (say), is a positive integer (including zero) operated upon by two operations named \( P \) and \( V \).

\[ P \text{ operation, } P(s) \]

Waits until \( s \) is greater than zero and then decrements \( s \) by one and allows the process to continue.

\[ V \text{ operation, } V(s) \]

Increments \( s \) by one to release one of the waiting processes (if any).

The \( P \) and \( V \) operations are performed indivisibly.

A mechanism for activating waiting processes is also implicit in the \( P \) and \( V \) operations. Though the exact algorithm is not specified, the algorithm is expected to be fair.

Processes delayed by \( P(s) \) are kept in abeyance until released by a \( V(s) \) on the same semaphore.
Mutual Exclusion of Critical Sections

Can be achieved with one semaphore having the value 0 or 1 (a *binary semaphore*), which acts as a lock variable, but the $P$ and $V$ operations include a process scheduling mechanism.

The semaphore is initialized to 1, indicating that no process is in its critical section associated with the semaphore.

Each mutually exclusive critical section is preceded by a $P (s)$ and terminated with a $V (s)$ on the same semaphore; i.e.,

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
<th>Process 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Noncritical section</td>
<td>Noncritical section</td>
<td>Noncritical section</td>
</tr>
<tr>
<td>$P (s)$</td>
<td>$P (s)$</td>
<td>$P (s)$</td>
</tr>
<tr>
<td>Critical section</td>
<td>Critical section</td>
<td>Critical section</td>
</tr>
<tr>
<td>$V (s)$</td>
<td>$V (s)$</td>
<td>$V (s)$</td>
</tr>
<tr>
<td>Noncritical section</td>
<td>Noncritical section</td>
<td>Noncritical section</td>
</tr>
</tbody>
</table>

Any process might reach its $P (s)$ operation first (or more than one process may reach it simultaneously).

The first process to reach its $P (s)$ operation, or to be accepted, will set the semaphore to 0, inhibiting the other processes from proceeding past their $P (s)$ operations, but any process reaching its $P (s)$ operation will be recorded so that one can be selected when the critical section is released.

When the process reaches its $V (s)$ operation, it sets the semaphore $s$ to 1 and one of the processes waiting is allowed to proceed into its critical section.
General Semaphore

Can take on positive values other than zero and one.

Such semaphores provide, for example, a means of recording the number of “resource units” available or used and can be used to solve producer/consumer problems.

Semaphore routines exist for UNIX processes. They do not exist in Pthreads as such, though they can be written and they do exist in the real-time extension to Pthreads.
Monitor

A suite of procedures that provides the only method to access a shared resource.

Essentially the data and the operations that can operate upon the data are encapsulated into one structure.

Reading and writing can only be done by using a monitor procedure, and only one process can use a monitor procedure at any instant.

A monitor procedure could be implemented using a semaphore to protect its entry; i.e.,

```java
monitor_procl()
{
    P(monitor_semaphore);
    .
    monitor body
    .
    V(monitor_semaphore);
    return;
}
```

Java Monitor

The concept of a monitor exists in Java.

The keyword `synchronized` in Java makes a block of code in a method thread safe, preventing more than one thread inside the method.
Condition Variables

Often, a critical section is to be executed if a specific global condition exists; for example, if a certain value of a variable has been reached.

With locks, the global variable would need to be examined at frequent intervals (“polled”) within a critical section. This is a very time-consuming and unproductive exercise.

The problem can be overcome by introducing so-called *condition variables*.

Operations

Three operations are defined for a condition variable:

- `Wait(cond_var)` — wait for a condition to occur
- `Signal(cond_var)` — signal that the condition has occurred
- `Status(cond_var)` — return the number of processes waiting for the condition to occur

The wait operation will also release a lock or semaphore and can be used to allow another process to alter the condition.

When the process calling `wait()` is finally allowed to proceed, the lock or semaphore is again set.

Example

Consider one or more processes (or threads) designed to take action when a counter, $x$, is zero. Another process or thread is responsible for decrementing the counter. The routines could be of the form

```c
action() counter()
{
    lock();
    while (x != 0)
        x--;
    wait(s);  // if (x == 0) signal(s);
    unlock();
    take_action();
}
```
Pthread Condition Variables

Condition variables associated with a specific mutex.

Given the declarations and initializations

```c
pthread_cond_t cond1;
 pthread_mutex_t mutex1;
.
 pthread_cond_init(&cond1, NULL);
 pthread_mutex_init(&mutex1, NULL);
```

the Pthreads arrangement for signal and wait is as follows:

```c
action() counter()
{
 .
 .
 pthread_mutex_lock(&mutex1); pthread_mutex_lock(&mutex1);
 while (c <> 0) c--;
 pthread_cond_wait(cond1, mutex1); if (c == 0) pthread_cond_signal(cond1);
 pthread_mutex_unlock(&mutex1); pthread_mutex_unlock(&mutex1);
 take_action(); .
 .
 .
}
```

Signals are not remembered, which means that threads must already be waiting for a signal to receive it.
Language Constructs for Parallelism

Shared Data

Shared memory variables might be declared as shared with, say,

    shared int x;

par Construct

For specifying concurrent statements:

    par {
        S1;
        S2;
        ...
        Sn;
    }

forall Construct

To start multiple similar processes together:

    forall (i = 0; i < n; i++) {
        S1;
        S2;
        ...
        Sn;
    }

which generates $n$ processes each consisting of the statements forming the body of the for loop, $S1, S2, \ldots, Sn$. Each process uses a different value of $i$.

Example

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

clears $a[0], a[1], a[2], a[3], \text{and } a[4]$ to zero concurrently.
Dependency Analysis

To identify which processes could be executed together.

Example

We can see immediately in the code

\[
\text{forall } (i = 0; i < 5; i++) \\
\quad a[i] = 0;
\]

that every instance of the body is independent of the other instances and all instances can be executed simultaneously.

However, it may not be obvious; e.g.,

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

Preferably, we need an algorithmic way of recognizing the dependencies, which might be used by a parallelizing compiler.
Bernstein's Conditions

Set of conditions that are sufficient to determine whether two processes can be executed simultaneously. Let us define two sets of memory locations, \( I \) (input) and \( O \) (output), such that

- \( I_i \) is the set of memory locations read by process \( P_i \).
- \( O_j \) is the set of memory locations altered by process \( P_j \).

For two processes \( P_1 \) and \( P_2 \) to be executed simultaneously, the inputs to process \( P_1 \) must not be part of the outputs of \( P_2 \), and the inputs of \( P_2 \) must not be part of the outputs of \( P_1 \); i.e.,

\[
I_1 \cap O_2 = \emptyset \\
I_2 \cap O_1 = \emptyset
\]

where \( \emptyset \) is an empty set. The set of outputs of each process must also be different; i.e.,

\[
O_1 \cap O_2 = \emptyset
\]

If the three conditions are all satisfied, the two processes can be executed concurrently.

Example

Suppose the two statements are (in C)

\[
\begin{align*}
a &= x + y; \\
b &= x + z;
\end{align*}
\]

We have

\[
\begin{align*}
I_1 &= (x, y) & O_1 &= (a) \\
I_2 &= (x, z) & O_2 &= (b)
\end{align*}
\]

and the conditions

\[
\begin{align*}
I_1 \cap O_2 &= \emptyset \\
I_2 \cap O_1 &= \emptyset \\
O_1 \cap O_2 &= \emptyset
\end{align*}
\]

are satisfied. Hence, the statements \( a = x + y \) and \( b = x + z \) can be executed simultaneously.
Shared Data in Systems with Caches

All modern computer systems have cache memory, high-speed memory closely attached to each processor for holding recently referenced data and code.

Cache coherence protocols

In the *update policy*, copies of data in all caches are updated at the time one copy is altered.

In the *invalidate policy*, when one copy of data is altered, the same data in any other cache is invalidated (by resetting a valid bit in the cache). These copies are only updated when the associated processor makes reference for it.

False Sharing

The key characteristic used is that caches are organized in blocks of contiguous locations. When a processor first references one or more bytes in the block, the whole block is transferred into the cache from the main memory.

Different parts of a block required by different processors but not the same bytes. If one processor writes to one part of the block, copies of the complete block in other caches must be updated or invalidated though the actual data is not shared.

![False Sharing in Caches](image-url)
Solution for False Sharing

Compiler to alter the layout of the data stored in the main memory, separating data only altered by one processor into different blocks.

This may be difficult to satisfy in all situations. For example, the code

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

is likely to create false sharing as the elements of `a`, `a[0]`, `a[1]`, `a[2]`, `a[3]`, and `a[4]`, are likely to be stored in consecutive locations in memory.

The only way to avoid false sharing would be to place each element in a different block, which would create significant wastage of storage for a large array.

Even code such as

```c
par {
  x = 0;
  y = 0;
}
```

where `x` and `y` are shared variables, could create false sharing as the variables `x` and `y` are likely to be stored together in a data segment.
Program Examples

To sum the elements of an array, \( a[1000] \):

```c
int sum, a[1000];
    sum = 0;
for (i = 0; i < 1000; i++)
    sum = sum + a[i];
```

UNIX Processes

For this example, the calculation will be divided into two parts, one doing the even \( i \) and one doing the odd \( i \); i.e.,

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{sum1} = 0; )</td>
<td>( \text{sum2} = 0; )</td>
</tr>
<tr>
<td>( \text{for} (i = 0; i &lt; 1000; i = i + 2) )</td>
<td>( \text{for} (i = 1; i &lt; 1000; i = i + 2) )</td>
</tr>
<tr>
<td>( \text{sum1} = \text{sum1} + a[i]; )</td>
<td>( \text{sum2} = \text{sum2} + a[i]; )</td>
</tr>
</tbody>
</table>

Each process will add its result (\( \text{sum1} \) or \( \text{sum2} \)) to an accumulating result, \( \text{sum} \) (after \( \text{sum} \) is initialized):

\[
\text{sum} = \text{sum} + \text{sum1}; \\
\text{sum} = \text{sum} + \text{sum2};
\]

producing the final answer. The result location, \( \text{sum} \), will need to be shared and access protected by a lock. For this program, a shared data structure is created:

![Figure 8.10 Shared memory locations for Section program example.](image-url)
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <stdio.h>
#include <errno.h>

#define array_size 1000 /* no of elements in shared memory */
extern char *shmat();
void P(int *s);
void V(int *s);

int main()
{
  int shmid, s, pid; /* shared memory, semaphore, proc id */
  char *shm; /* shared mem. addr returned by shmat() */
  int *a, *addr, *sum; /* shared data variables */
  int partial_sum; /* partial sum of each process */
  int i;

  int init_sem_value = 1;
  s = semget(IPC_PRIVATE, 1, (0600 | IPC_CREAT))
  if (s == -1) { /* if unsuccessful */
    perror("semget");
    exit(1);
  }
  if (semctl(s, 0, SETVAL, init_sem_value) < 0) {
    perror("semctl");
    exit(1);
  }

  /* initialize semaphore set */
  /* create segment */
  shmid = shmget(IPC_PRIVATE, (array_size*sizeof(int)+1),
      (IPC_CREAT|0600));
  if (shmid == -1) {
    perror("shmget");
    exit(1);
  }

  /* map segment to process data space */
  shm = shmat(shmid, NULL, 0); /* returns address as a character */
  if (shm == (char*)-1) {
    perror("shmat");
    exit(1);
  }
}
addr = (int*)shm; /* starting address */
sum = addr; /* accumulating sum */
addr++;
a = addr; /* array of numbers, a[] */

*sum = 0;
for (i = 0; i < array_size; i++) /* load array with numbers */
  *(a + i) = i+1;

pid = fork(); /* create child process */
if (pid == 0) { /* child does this */
partial_sum = 0;
  for (i = 0; i < array_size; i = i + 2)
    partial_sum += *(a + i);
else {
  /* parent does this */
  partial_sum = 0;
  for (i = 1; i < array_size; i = i + 2)
    partial_sum += *(a + i);
}
P(&s); /* for each process, add partial sum */
  *sum += partial_sum;
V(&s);

printf("\nprocess pid = %d, partial sum = %d\n", pid, partial_sum);
if (pid == 0) exit(0); else wait(0); /* terminate child proc */
printf("\nThe sum of 1 to %i is %d\n", array_size, *sum);

  /* remove semaphore */
if (semctl(s, 0, IPC_RMID, 1) == -1) {
  perror("semctl");
  exit(1);
}

  /* remove shared memory */
if (shmctl(shmid, IPC_RMID, NULL) == -1) {
  perror("shmctl");
  exit(1);
}
} /* end of main */
void P(int *s) /* P(s) routine*/
{
    struct sembuf sembuffer, *sops;
    sops = &sembuffer;
    sops->sem_num = 0;
    sops->sem_op = -1;
    sops->sem_flg = 0;
    if (semop(*s, sops, 1) < 0) {
        perror("semop");
        exit(1);
    }
    return;
}

void V(int *s) /* V(s) routine */
{
    struct sembuf sembuffer, *sops;
    sops = &sembuffer;
    sops->sem_num = 0;
    sops->sem_op = 1;
    sops->sem_flg = 0;
    if (semop(*s, sops, 1) < 0) {
        perror("semop");
        exit(1);
    }
    return;
}

SAMPLE OUTPUT

process pid = 0, partial sum = 250000
process pid = 26127, partial sum = 250500
The sum of 1 to 1000 is 500500
Pthreads Example

In this example, \( n \) threads are created, each taking numbers from the list to add to their sums. When all numbers have been taken, the threads can add their partial results to a shared location \( \text{sum} \).

The shared location \( \text{global\_index} \) is used by each thread to select the next element of \( a[] \).

After \( \text{index} \) is read, it is incremented in preparation for the next element to be read.

The result location is \( \text{sum} \), as before, and will also need to be shared and access protected by a lock.

Figure 8.11  Shared memory locations for Section 8.4.2 program example.
#include <stdio.h>
#include <pthread.h>
#define array_size 1000
#define no_threads 10

/* shared data */
int a[array_size];  /* array of numbers to sum */
int global_index = 0;  /* global index */
int sum = 0;  /* final result, also used by slaves */
pthread_mutex_t mutex1;  /* mutually exclusive lock variable */

void *slave(void *ignored) /* Slave threads */
{
    int local_index, partial_sum = 0;
    do {
        pthread_mutex_lock(&mutex1); /* get next index into the array */
        local_index = global_index; /* read current index & save locally*/
        global_index++; /* increment global index */
        pthread_mutex_unlock(&mutex1);

        if (local_index < array_size) partial_sum += *(a + local_index);
    } while (local_index < array_size);

    pthread_mutex_lock(&mutex1); /* add partial sum to global sum */
    sum += partial_sum;
    pthread_mutex_unlock(&mutex1);

    return (); /* Thread exits */
}

main () {
    int i;
    pthread_t thread[10]; /* threads */
    pthread_mutex_init(&mutex1,NULL); /* initialize mutex */

    for (i = 0; i < array_size; i++) /* initialize a[] */
        a[i] = i+1;

    for (i = 0; i < no_threads; i++) /* create threads */
        if (pthread_create(&thread[i], NULL, slave, NULL) != 0)
            perror("Pthread_create fails");

    for (i = 0; i < no_threads; i++) /* join threads */
        if (pthread_join(thread[i], NULL) != 0)
            perror("Pthread_join fails");
    printf("The sum of 1 to %i is %d\n", array_size, sum);
    /* end of main */
}

SAMPLE OUTPUT

The sum of 1 to 1000 is 500500
Java Example

```java
public class Adder {
    private int[] array;
    private int sum = 0;
    private int index = 0;
    private int number_of_threads = 10;
    private int threads_quit;

    public Adder() {
        threads_quit = 0;
        array = new int[1000];
        initializeArray();
        startThreads();
    }

    public synchronized int getNextIndex() {
        if(index < 1000) return(index++); else return(-1);
    }

    public synchronized void addPartialSum(int partial_sum) {
        sum = sum + partial_sum;
        if(++threads_quit == number_of_threads)
            System.out.println("The sum of the numbers is " + sum);
    }

    private void initializeArray() {
        int i;
        for(i = 0; i < 1000; i++) array[i] = i;
    }

    public void startThreads() {
        int i = 0;
        for(i = 0; i < 10; i++)
            AdderThread at = new AdderThread(this, i);
            at.start();
    }
}
```
public static void main(String args[]) {
    Adder a = new Adder();
}

class AdderThread extends Thread {
    int partial_sum = 0;
    Adder parent;
    int number;
    public AdderThread(Adder parent, int number) {
        this.parent = parent;
        this.number = number;
    }
    public void run() {
        int index = 0;
        while(index != -1) {
            partial_sum = partial_sum + parent.array[index];
            index = parent.getNextIndex();
        }
        System.out.println("Partial sum from thread "+number+" is "+partial_sum);
        parent.addPartialSum(partial_sum);
    }
}