

Programming with Shared Memory

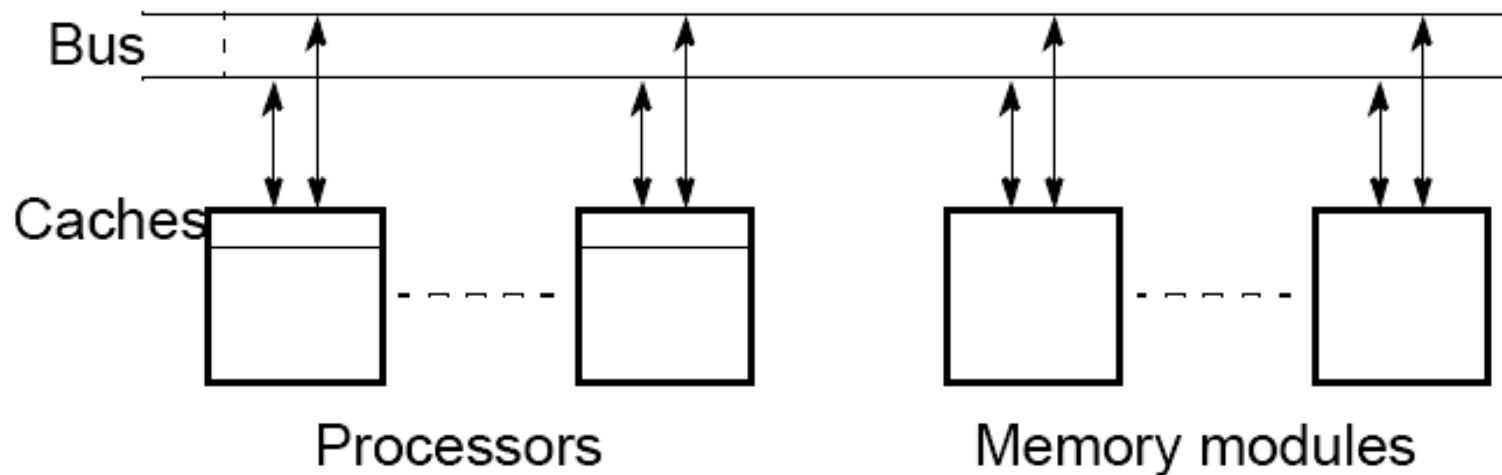
Shared memory multiprocessor system

Any memory location can be accessible by any of the processors.

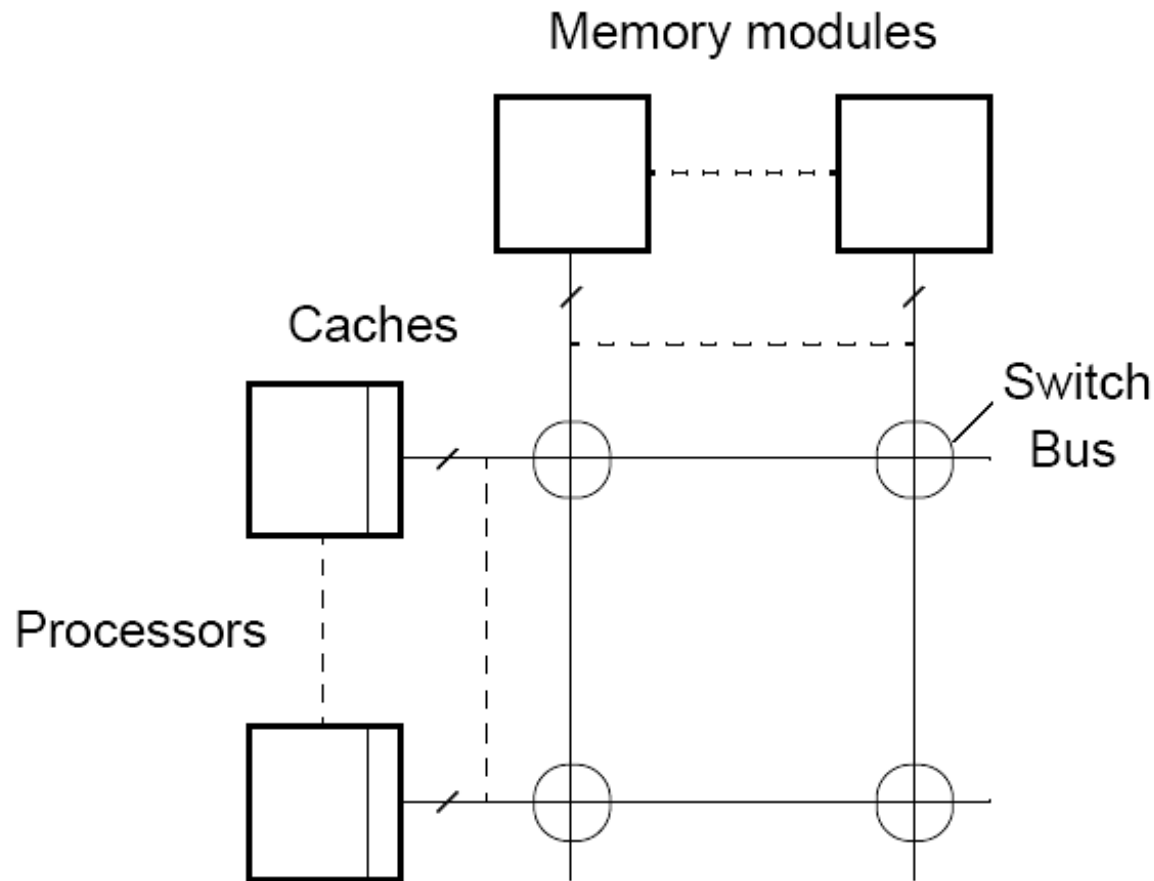
A *single address space* exists, meaning that each memory location is given a unique address within a single range of addresses.

Generally, shared memory programming more convenient although it does require access to shared data to be controlled by the programmer (using critical sections etc.)

Shared memory multiprocessor using a single bus



Shared memory multiprocessor using a crossbar switch



Alternatives for Programming Shared Memory Multiprocessors:

- Using heavy weight processes.
- Using threads. Example Pthreads
- Using a completely new programming language for parallel programming - not popular. Example Ada.
- Using library routines with an existing sequential programming language.
- Modifying the syntax of an existing sequential programming language to create a parallel programming language. Example UPC
- Using an existing sequential programming language supplemented with compiler directives for specifying parallelism. Example OpenMP

Using Heavyweight Processes

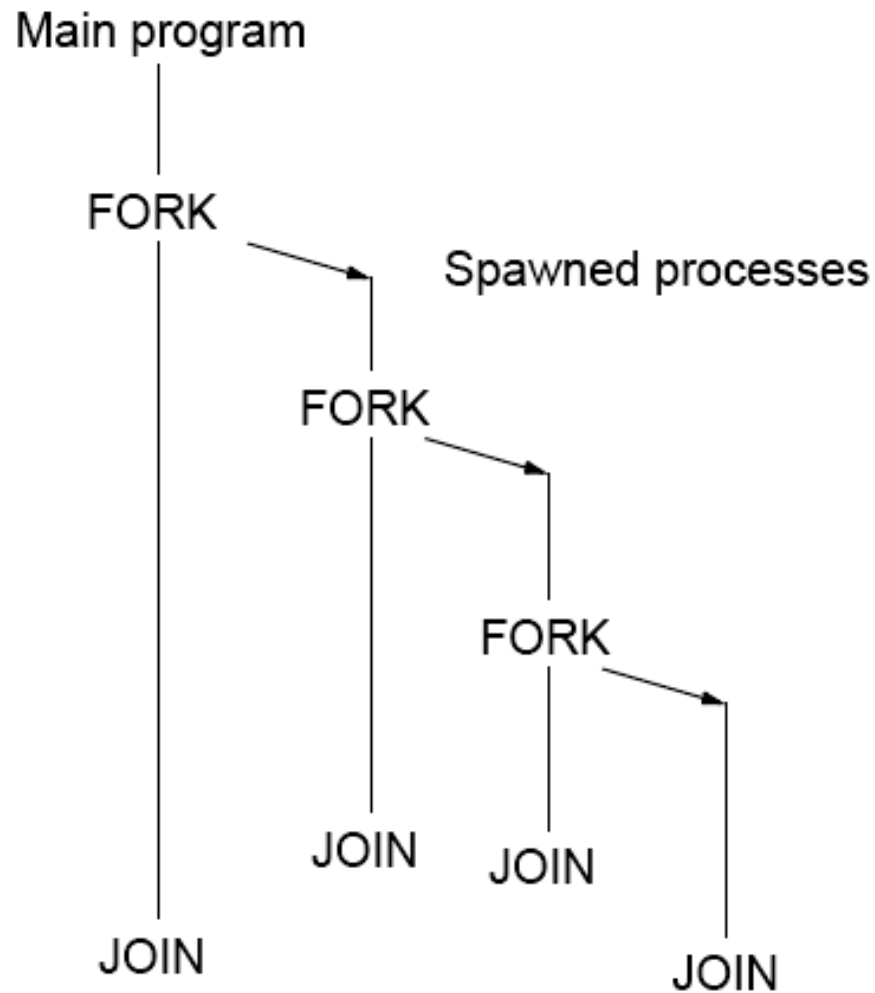
Operating systems often based upon notion of a process.

Processor time shares between processes, switching from one process to another. Might occur at regular intervals or when an active process becomes delayed.

Offers opportunity to deschedule processes blocked from proceeding for some reason, e.g. waiting for an I/O operation to complete.

Concept could be used for parallel programming. Not much used because of overhead but fork/join concepts used elsewhere.

FORK-JOIN construct



UNIX System Calls

No join routine - use `exit()` and `wait()`

SPMD model

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


UNIX System Calls

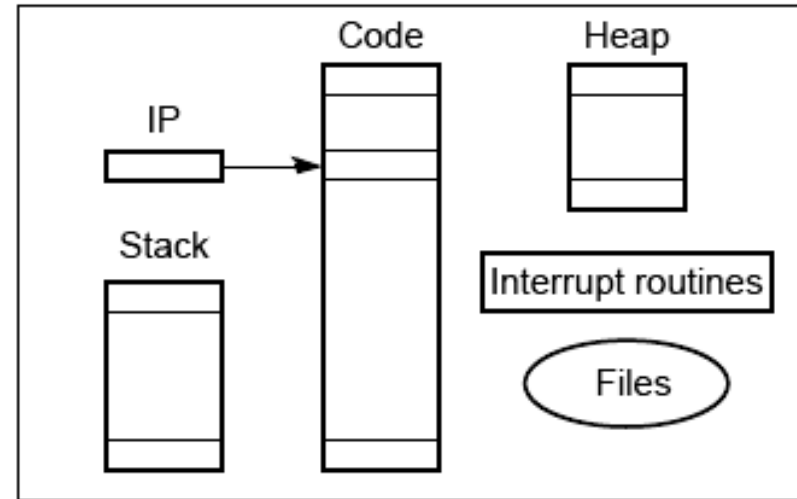
SPMD model with different code for master process and forked slave process.

```
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);  
:  
:
```

Differences between a process and threads

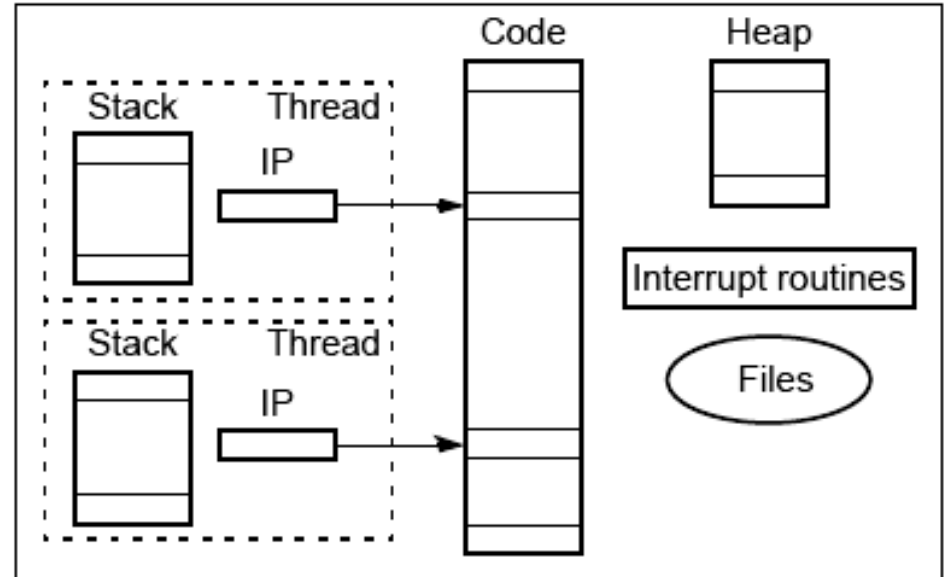
“heavyweight” process - completely separate program with its own variables, stack, and memory allocation.

(a) Process



Threads - shares the same memory space and global variables between routines.

(b) Threads



Pthreads

IEEE Portable Operating System Interface, POSIX, sec. 1003.1 standard

Executing a Pthread Thread

Main program

thread1

procl(&arg)

{

return(*status);

}

`pthread_create(&thread1, NULL, procl, &arg);`

`pthread_join(thread1, *status);`

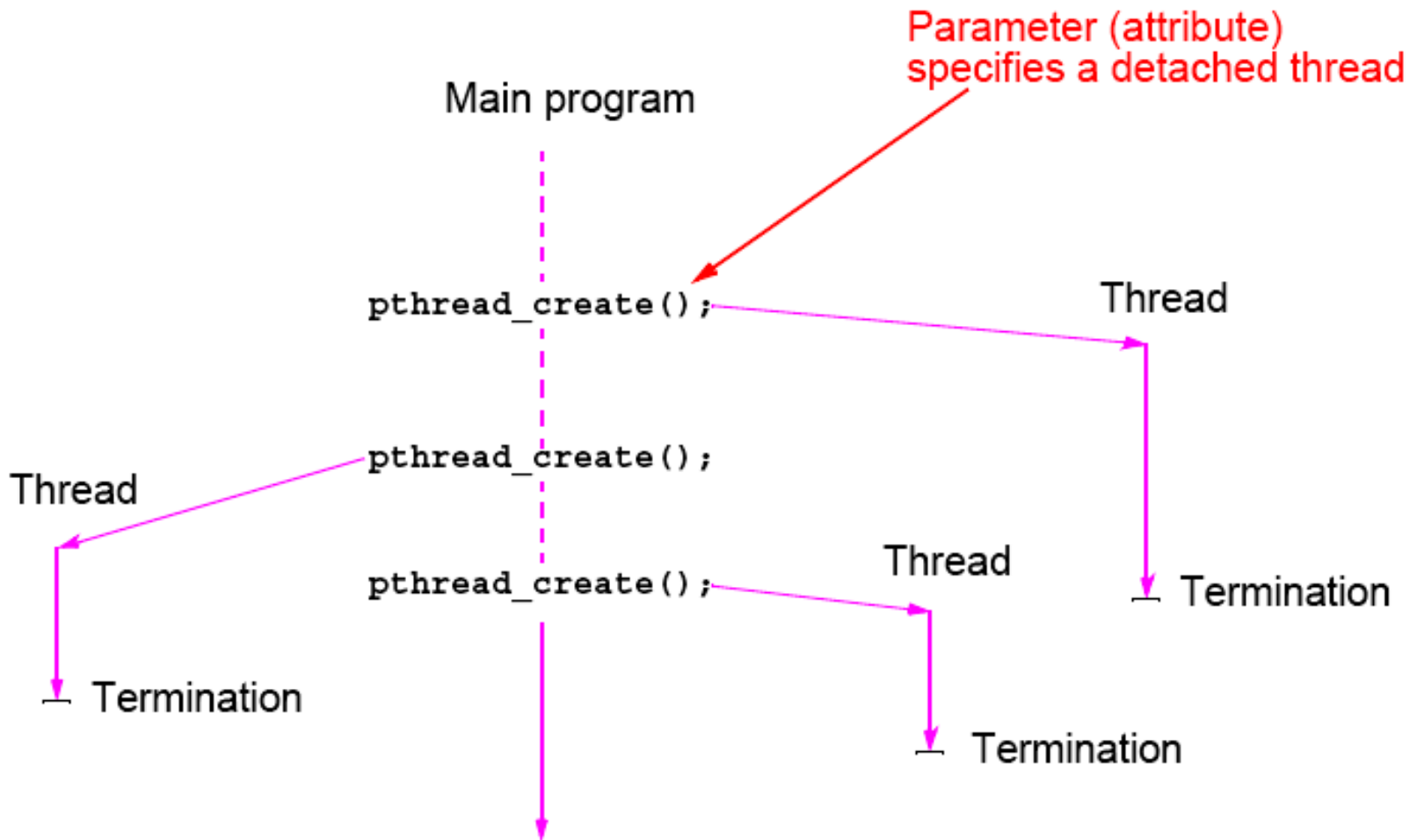
Detached Threads

It may be that thread are not bothered when a thread it creates terminates and then a join not needed.

Threads not joined are called *detached threads*.

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

Pthreads Detached Threads



Statement Execution Order

Single processor: Processes/threads typically executed until blocked.

Multiprocessor: Instructions of processes/threads interleaved in time.

Example

Process 1

Instruction 1.1

Instruction 1.2

Instruction 1.3

Process 2

Instruction 2.1

Instruction 2.2

Instruction 2.3

Several possible orderings, including

Instruction 1.1

Instruction 1.2

Instruction 2.1

Instruction 1.3

Instruction 2.2

Instruction 2.3

assuming instructions cannot be divided into smaller 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/Processor Optimizations

Compiler and processor reorder instructions for optimization.

Example

The statements

```
a = b + 5;  
x = y + 4;
```

could be compiled to execute in reverse order:

```
x = y + 4;  
a = b + 5;
```

and still be logically correct.

May be advantageous to delay statement `a = b + 5` because a previous instruction currently being executed in processor needs more time to produce the value for `b`. Very common for processors to execute machines instructions out of order for increased speed .

Thread-Safe Routines

Thread safe if they can be called from multiple threads simultaneously and always produce correct results.

Standard I/O thread safe (prints messages without interleaving the characters).

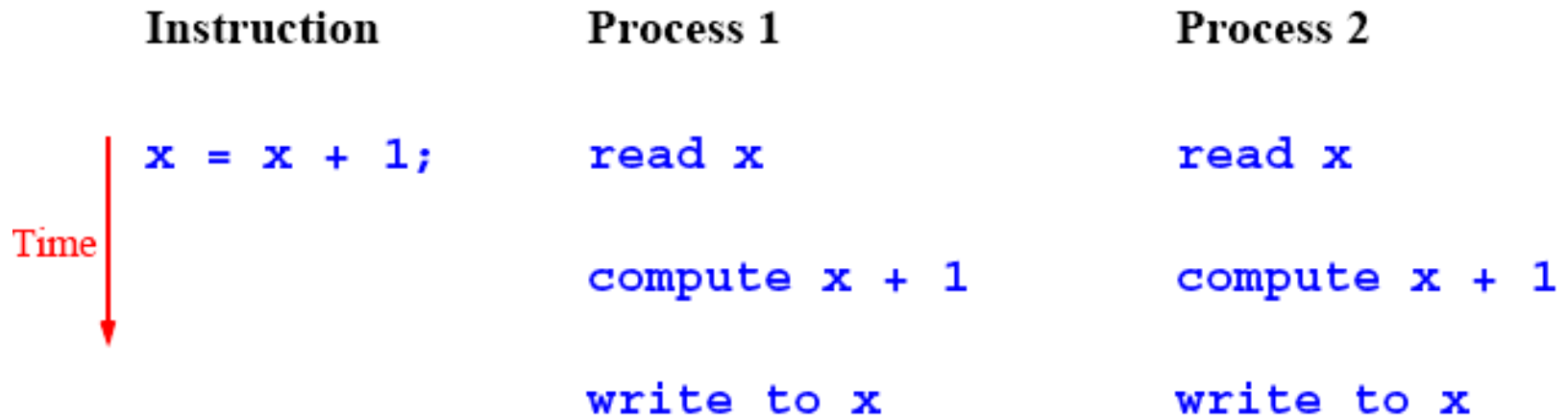
System routines that return time *may not be thread safe*.

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

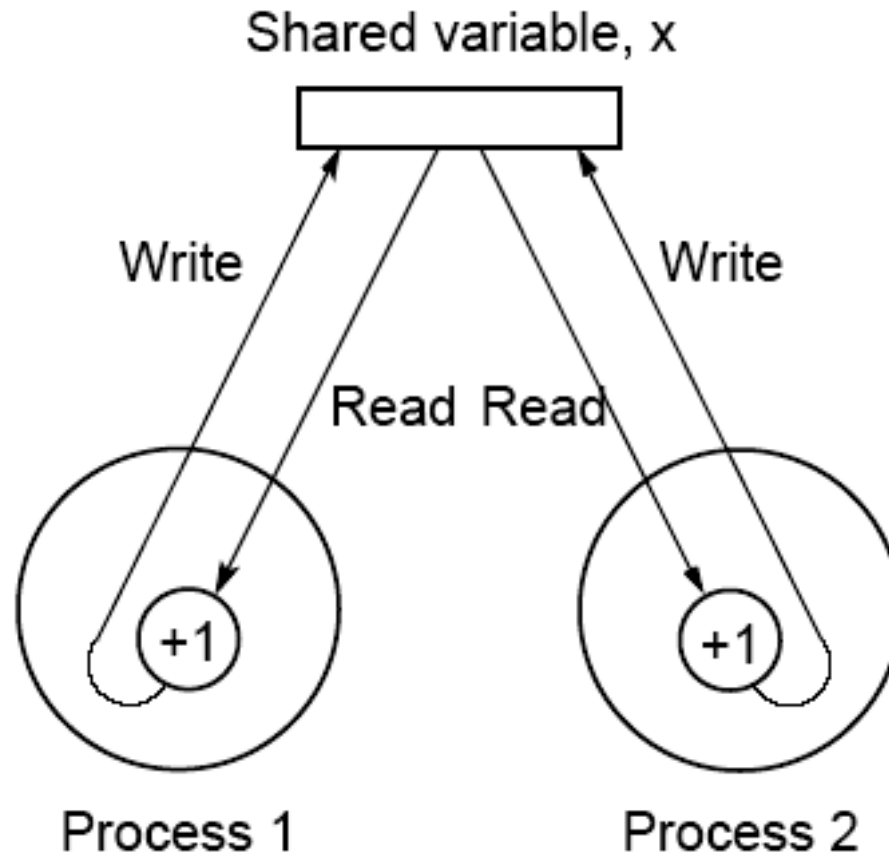
Accessing Shared Data

Accessing shared data needs careful control.

Consider two processes each of which is to add one to a shared data item, x . Necessary for the contents of the location x to be read, $x + 1$ computed, and the result written back to the location:



Conflict in accessing shared variable



Critical Section

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

This mechanism is known as *mutual exclusion*.

This concept also appears in an operating systems.

Locks

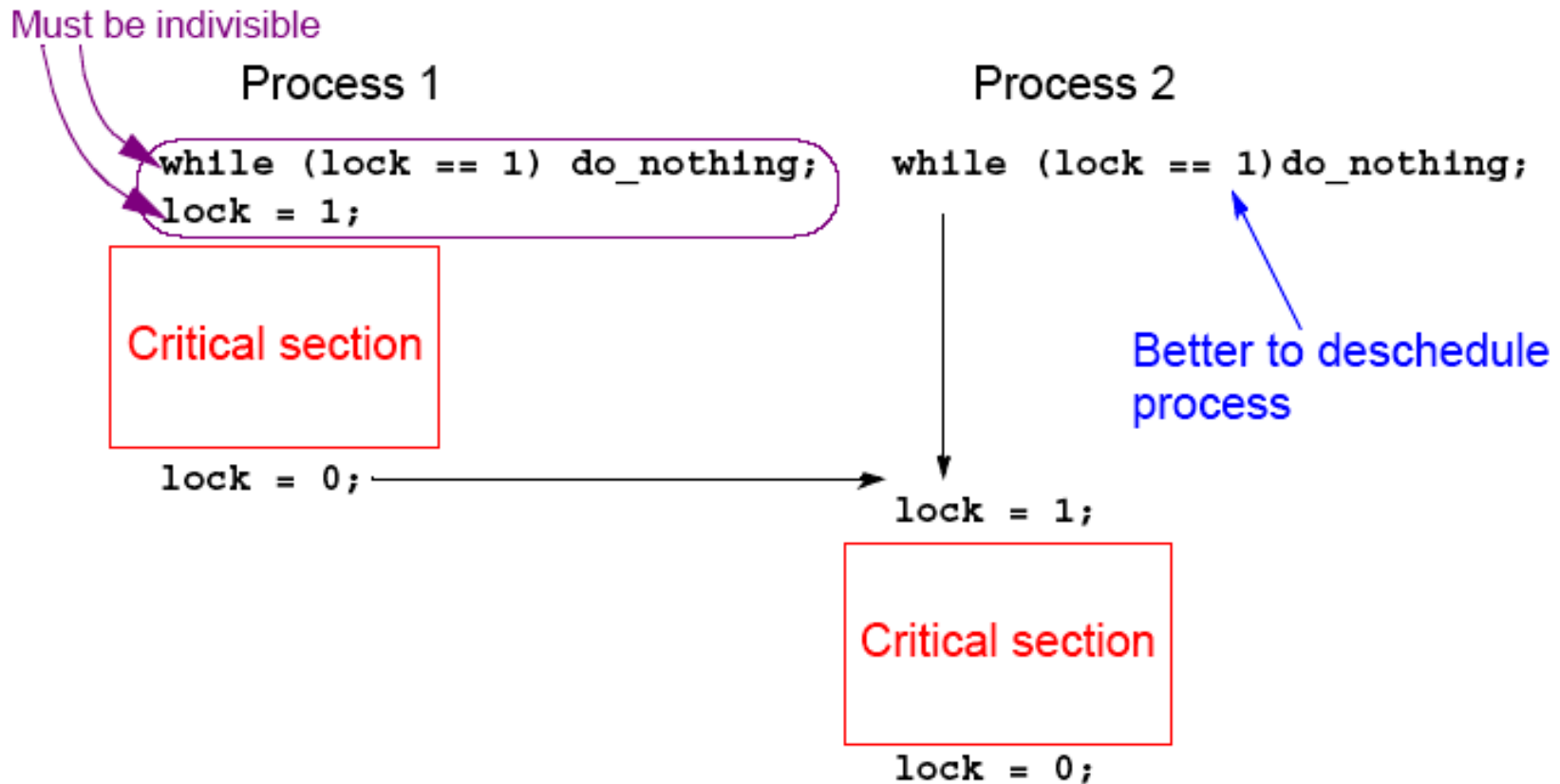
Simplest mechanism for ensuring mutual exclusion of critical sections.

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.

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.

Control of critical sections through busy waiting



Pthread Lock Routines

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

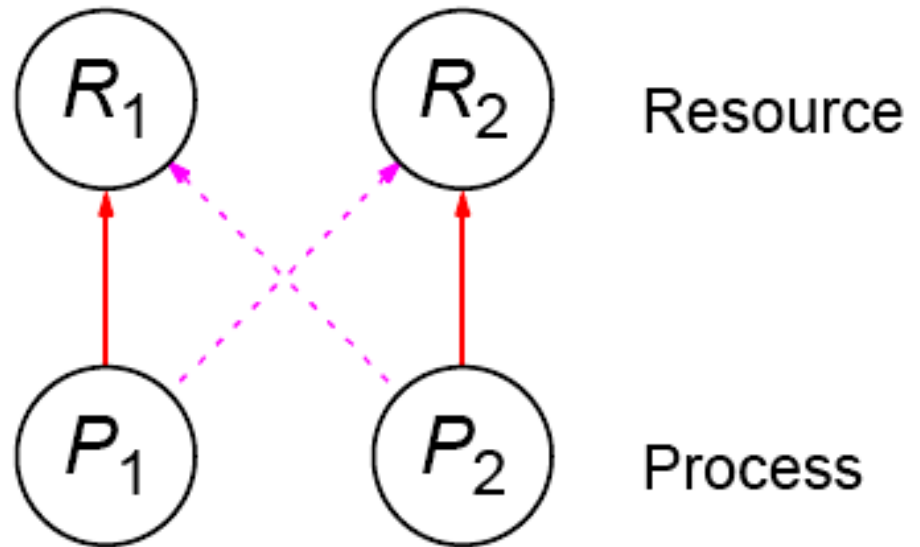
```
pthread_mutex_lock(&mutex1);  
critical section  
pthread_mutex_unlock(&mutex1);
```

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

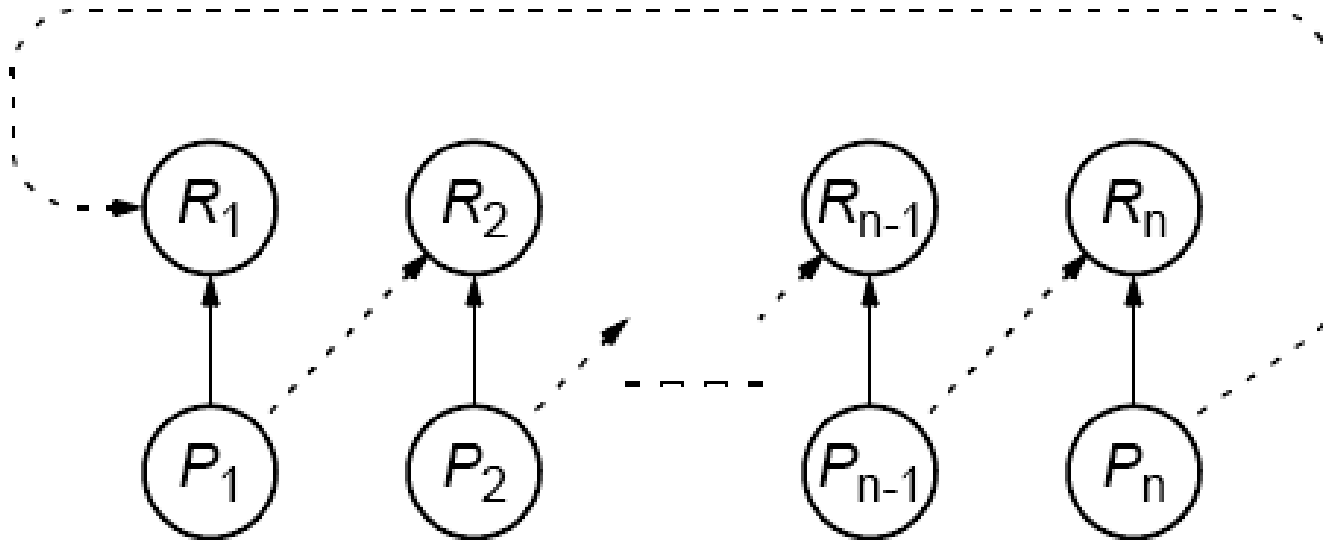
Can occur with two processes when one requires a resource held by the other, and this process requires a resource held by the first process.

Two-process deadlock



Deadlock (deadly embrace)

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



Pthreads

Offers one routine that can test whether a lock is actually closed without blocking the thread:

pthread_mutex_trylock()

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 positive integer (including zero) operated upon by two operations:

P operation on semaphore s

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

V operation on semaphore s

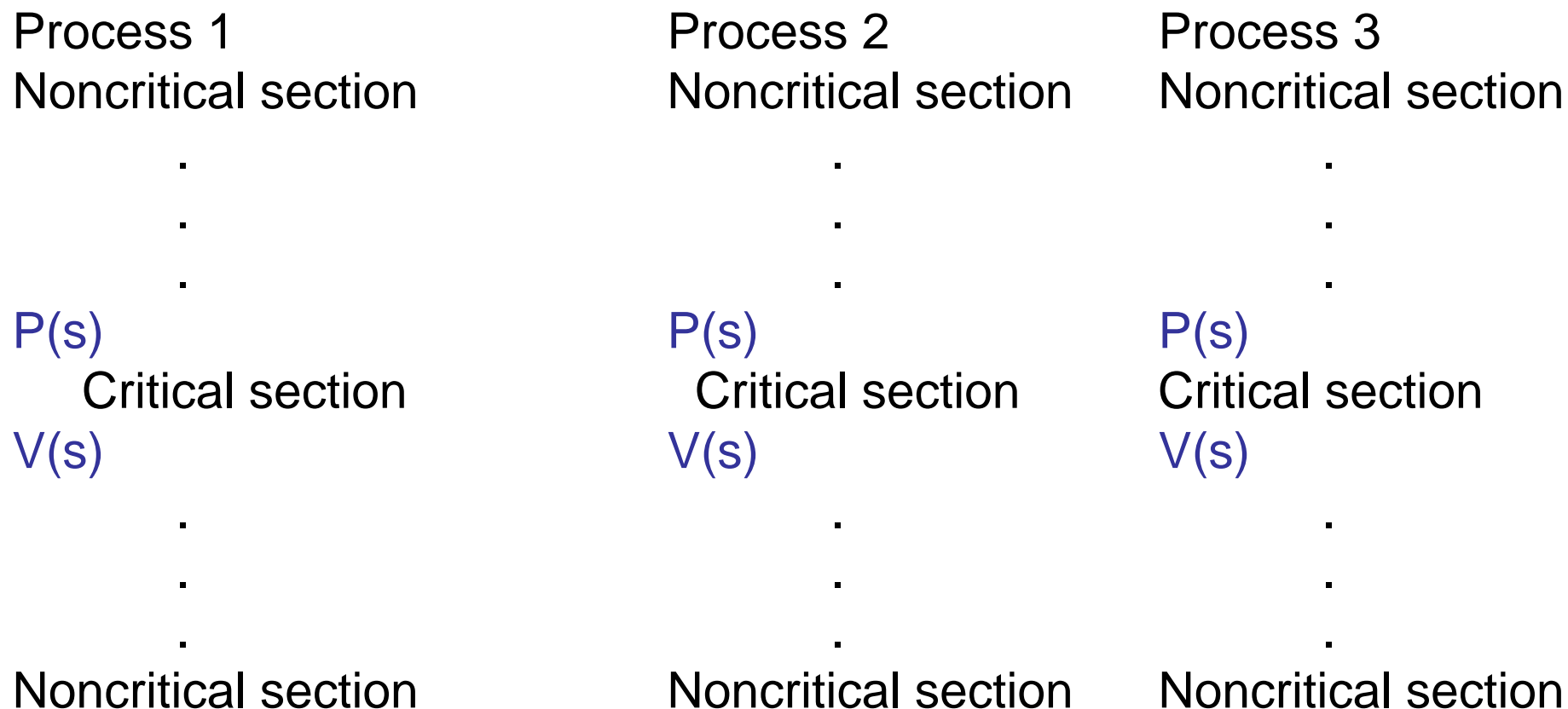
Increments s by one and releases one of the waiting processes (if any).

P and **V** operations are performed indivisibly.

Mechanism for activating waiting processes is also implicit in **P** and **V** operations. Though exact algorithm not specified, algorithm expected to be fair. Processes delayed by **P**(s) are kept in abeyance until released by a **V**(s) on the same semaphore.

Devised by Dijkstra in 1968. Letter **P** is from the Dutch word *passeren*, meaning “to pass,” and letter **V** is from the Dutch word *vrijgeven*, meaning “to release.”)

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:



General semaphore (or counting semaphore)

Can take on positive values other than zero and one.
Provide, for example, a means of recording the number of “resource units” available or used and can be used to solve producer/ consumer problems. - more on that in operating system courses.

Semaphore routines exist for UNIX processes.
Not exist in Pthreads as such, though they can be written
Do exist in real-time extension to Pthreads.

Monitor

Suite of procedures that provides only way to access shared resource. Only one process can use a monitor procedure at any instant.

Could be implemented using a semaphore or lock to protect entry, i.e.,

```
monitor_proc1()  
{  
  lock(x);  
  .  
  monitor body  
  .  
  unlock(x);  
  return;  
}
```

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.

Very time-consuming and unproductive exercise.

Can be overcome by introducing so-called *condition variables*.

Pthread Condition Variables

Pthreads arrangement for signal and wait:

```
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 *not* remembered - 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;  
    .  
    .  
    Sm;  
}
```

which generates n processes each consisting of the statements forming the body of the for loop, S1, S2, ..., Sm. 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]**, and **a[4]** to zero concurrently.

Dependency Analysis

To identify which processes could be executed together.

Example

Can see immediately in the code

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

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

However, it may not be that obvious. Need algorithmic way of recognizing dependencies, for a *parallelizing compiler*.

Bernstein's Conditions

Set of conditions sufficient to determine whether two processes can be executed simultaneously. Given:

I_i is the set of memory locations read (input) by process P_i .

O_j is the set of memory locations written (output) by process P_j .

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

$$I_1 \cap O_2 = \phi$$

$$I_2 \cap O_1 = \phi$$

where ϕ is an empty set. Set of outputs of each process must also be different; i.e.,

$$O_1 \cap O_2 = \phi$$

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

Example

Suppose the two statements are (in C)

`a = x + y;`

`b = x + z;`

We have

$I_1 = (x, y)$

$O_1 = (a)$

$I_2 = (x, z)$

$O_2 = (b)$

and the conditions

$I_1 \cap O_2 = \phi$

$I_2 \cap O_1 = \phi$

$O_1 \cap O_2 = \phi$

are satisfied. Hence, the statements `a = x + y` and `b = x + z` can be executed simultaneously.

OpenMP

An accepted standard developed in the late 1990s by a group of industry specialists.

Consists of a small set of compiler directives, augmented with a small set of library routines and environment variables using the base language Fortran and C/C++.

The compiler directives can specify such things as the par and forall operations described previously.

Several OpenMP compilers available.

For C/C++, the OpenMP directives are contained in `#pragma` statements. The OpenMP `#pragma` statements have the format:

`#pragma omp directive_name ...`

where `omp` is an OpenMP keyword.

May be additional parameters (clauses) after the directive name for different options.

Some directives require code to specified in a structured block (a statement or statements) that follows the directive and then the directive and structured block form a “construct”.

OpenMP uses “fork-join” model but thread-based.

Initially, a single thread is executed by a master thread. Parallel regions (sections of code) can be executed by multiple threads (a team of threads).

parallel directive creates a team of threads with a specified block of code executed by the multiple threads in parallel. The exact number of threads in the team determined by one of several ways.

Other directives used within a **parallel** construct to specify parallel for loops and different blocks of code for threads.

Parallel Directive

```
#pragma omp parallel  
structured_block
```

creates multiple threads, each one executing the specified structured_block, either a single statement or a compound statement created with { ... } with a single entry point and a single exit point.

There is an implicit barrier at the end of the construct.
The directive corresponds to forall construct.

Example

```
#pragma omp parallel private(x, num_threads)
{
    x = omp_get_thread_num();
    num_threads = omp_get_num_threads();
    a[x] = 10*num_threads;
}
```

Two library routines

`omp_get_num_threads()` returns number of threads that are currently being used in parallel directive

`omp_get_thread_num()` returns thread number (an integer from 0 to `omp_get_num_threads()` – 1 where thread 0 is the master thread).

Array `a[]` is a global array, and `x` and `num_threads` are declared as private to the threads.

Number of threads in a team

Established by either:

1. `num_threads` clause after the `parallel` directive, or
2. `omp_set_num_threads()` library routine being previously called,
or
3. the environment variable `OMP_NUM_THREADS` is defined in the order given or is system dependent if none of the above.

Number of threads available can also be altered automatically to achieve best use of system resources by a “dynamic adjustment” mechanism.

Work-Sharing

Three constructs in this classification:

sections
for
single

In all cases, there is an implicit barrier at the end of the construct unless a **nowait** clause is included.

Note that these constructs do not start a new team of threads. That done by an enclosing **parallel** construct.

Sections

The construct

```
#pragma omp sections
{
    #pragma omp section
    structured_block
    #pragma omp section
    structured_block
    .
    .
    .
}
```

cause the structured blocks to be shared among threads in team.

`#pragma omp sections` precedes the set of structured blocks.

`#pragma omp section` prefixes each structured block.

The first `section` directive is optional.

For Loop

```
#pragma omp for  
for_loop
```

causes the for loop to be divided into parts and parts shared among threads in the team. The for loop must be of a simple form.

Way that for loop divided can be specified by an additional “schedule” clause. Example: the clause `schedule (static, chunk_size)` cause the for loop be divided into sizes specified by `chunk_size` and allocated to threads in a round robin fashion.

Single

The directive

```
#pragma omp single  
structured block
```

cause the structured block to be executed by one thread only.

Combined Parallel Work-sharing Constructs

If a `parallel` directive is followed by a single `for` directive, it can be combined into:

```
#pragma omp parallel for  
for_loop
```

with similar effects, i.e. it has the effect of each thread executing the same for loop.

If a `parallel` directive is followed by a single `sections` directive, it can be combined into:

```
#pragma omp parallel sections {  
    #pragma omp section  
        structured_block  
    #pragma omp section  
        structured_block  
        .  
        .  
        .  
}
```

with similar effect.

(In both cases, the `nowait` clause is not allowed.)

Master Directive

The **master** directive:

```
#pragma omp master  
structured_block
```

causes the master thread to execute the structured block.

Different to those in the work sharing group in that there is no implied barrier at the end of the construct (nor the beginning). Other threads encountering this directive will ignore it and the associated structured block, and will move on.

Synchronization Constructs

Critical

The **critical** directive will only allow one thread execute the associated structured block. When one or more threads reach the **critical** directive:

```
#pragma omp critical name  
    structured_block
```

they will wait until no other thread is executing the same critical section (one with the same name), and then one thread will proceed to execute the structured block. name is optional. All critical sections with no name map to one undefined name.

Barrier

When a thread reaches the barrier

`#pragma omp barrier`

it waits until all threads have reached the barrier and then they all proceed together.

There are restrictions on the placement of barrier directive in a program. In particular, all threads must be able to reach the barrier.

Atomic

The atomic directive

```
#pragma omp atomic  
expression_statement
```

implements a critical section efficiently when the critical section simply updates a variable (adds one, subtracts one, or does some other simple arithmetic operation as defined by `expression_statement`).

Flush

A synchronization point which causes thread to have a “consistent” view of certain or all shared variables in memory. All current read and write operations on the variables allowed to complete and values written back to memory but any memory operations in the code after flush are not started, thereby creating a “memory fence”. Format:

`#pragma omp flush (variable_list)`

Only applied to thread executing flush, not to all threads in the team.

Flush occurs automatically at the entry and exit of parallel and critical directives (and combined parallel for and parallel sections directives), and at the exit of for, sections, and single (if a no-wait clause is not present).

Ordered

Used in conjunction with for and parallel for directives to cause an iteration to be executed in the order that it would have occurred if written as a sequential loop.

See Appendix C of textbook for further details.

Shared Memory Programming Performance Issues

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.

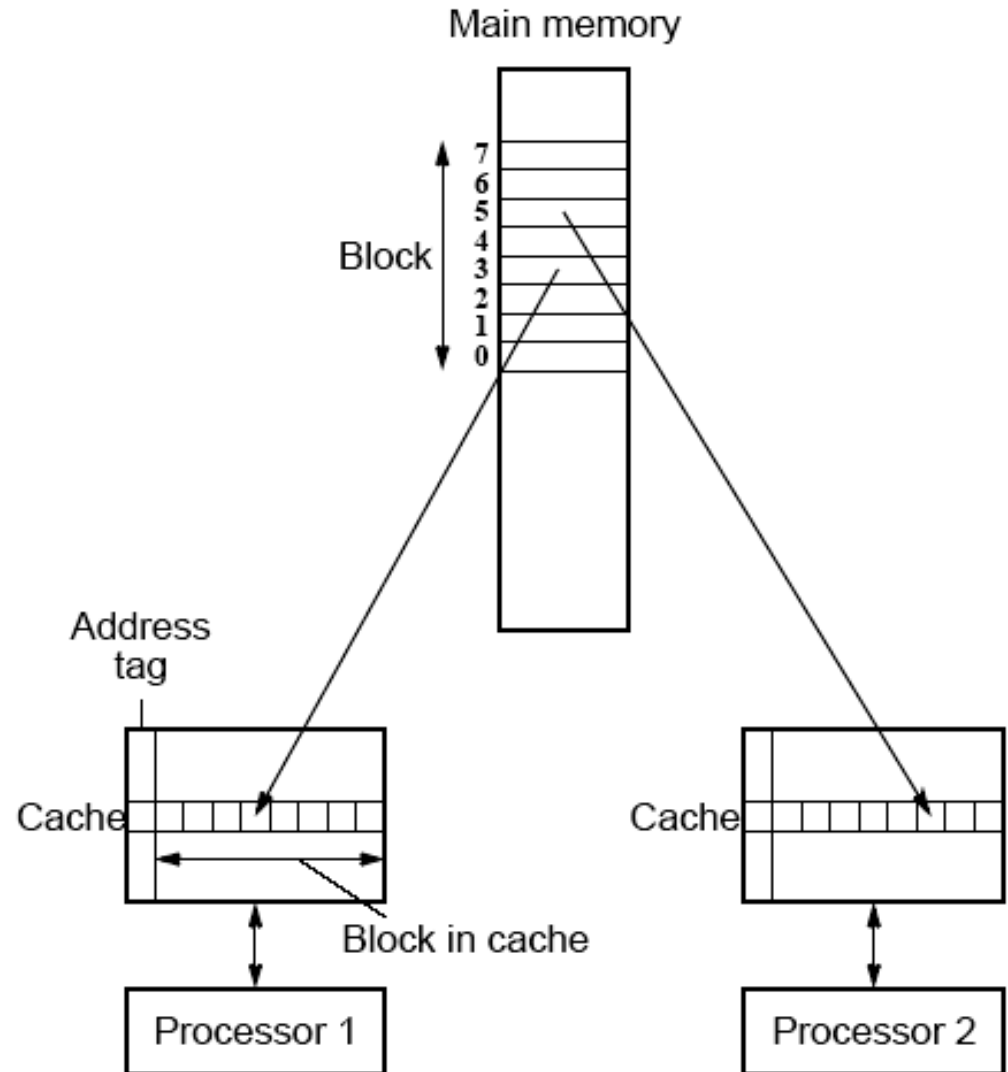
Cache coherence protocols

Update policy - copies of data in all caches are updated at the time one copy is altered.

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

Different parts of block required by different processors but not 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.



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.

Critical Sections Serializing Code

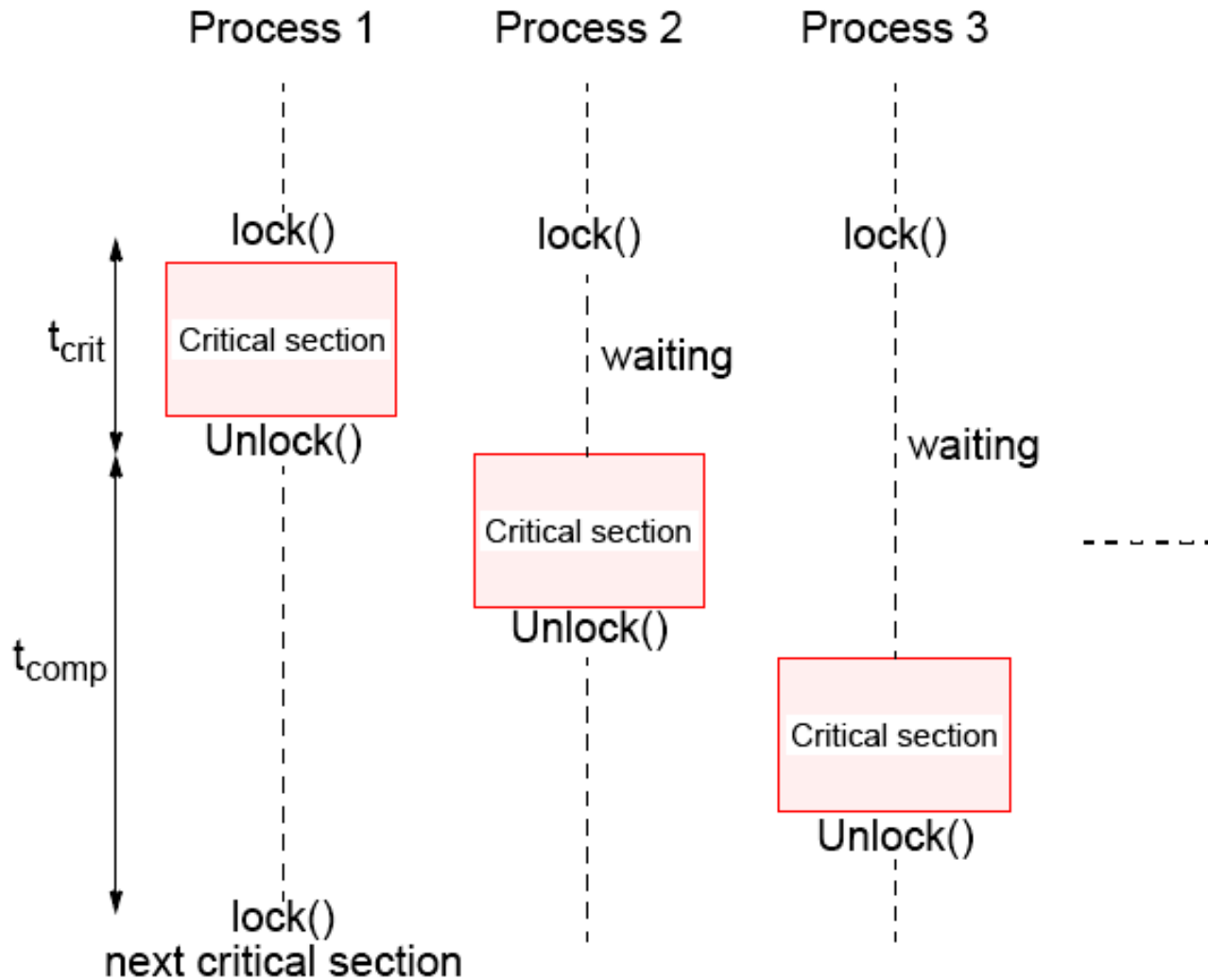
High performance programs should have as few as possible critical sections as their use can serialize the code.

Suppose, all processes happen to come to their critical section together.

They will execute their critical sections one after the other.

In that situation, the execution time becomes almost that of a single processor.

Illustration



When $t_{comp} < pt_{crit}$, less than p processor will be active

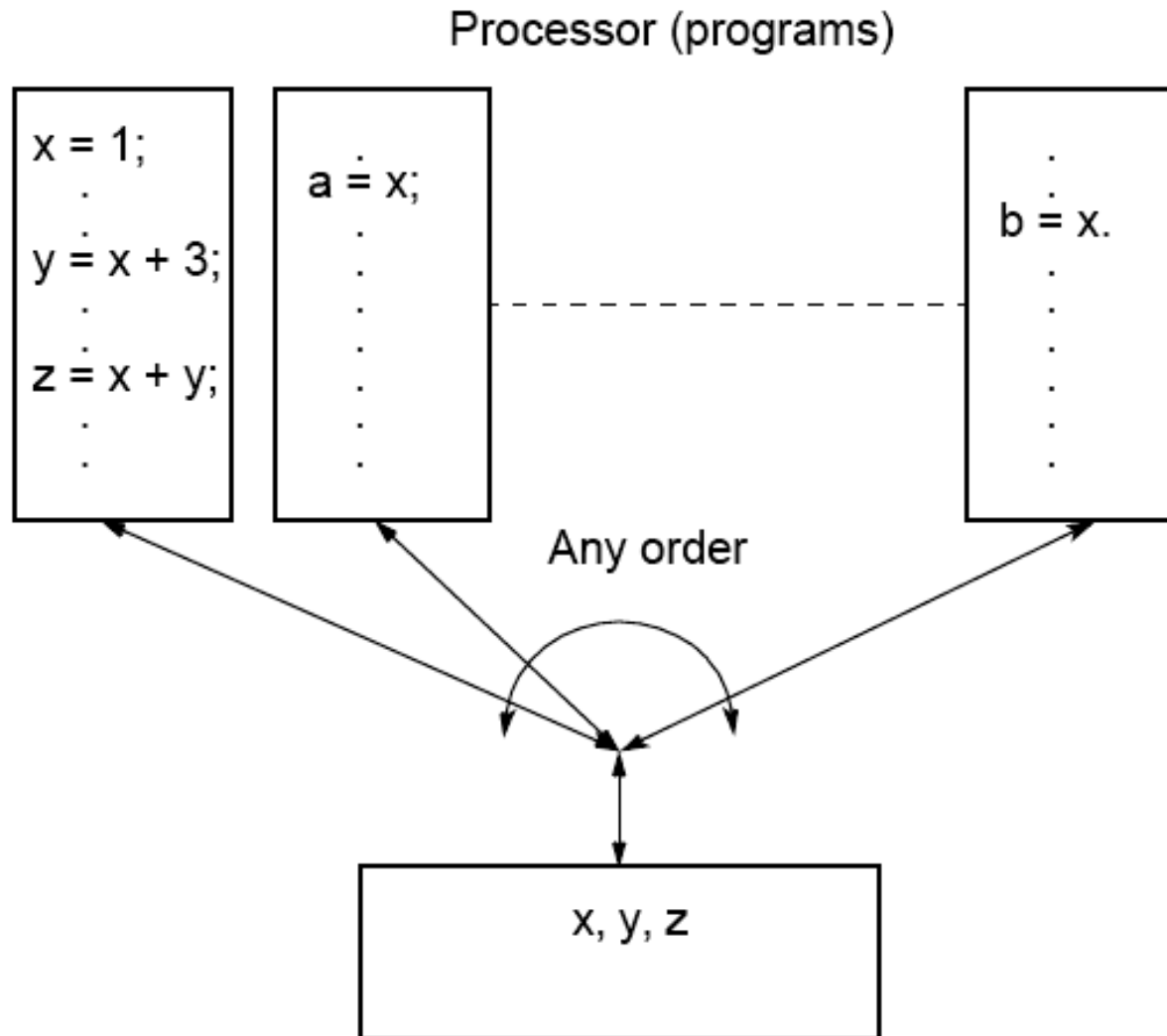
Sequential Consistency

Formally defined by Lamport (1979):

A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processors occur in this sequence in the order specified by its program.

i.e. the overall effect of a parallel program is not changed by any arbitrary interleaving of instruction execution in time.

Sequential Consistency



Writing a parallel program for a system which is known to be sequentially consistent enables us to reason about the result of the program.

Example

Process P1

```
.  
data = new;  
flag = TRUE;  
.br/>.br/>.br>.
```

Process 2

```
.  
.br/>.br/>.br/>while (flag != TRUE) { };  
data_copy = data;  
.br>.
```

Expect **data_copy** to be **set to new** because we expect the statement `data = new` to be executed before `flag = TRUE` and the statement `while (flag != TRUE) { }` to be executed before `data_copy = data`. Ensures that process 2 reads new data from another process 1. Process 2 will simply wait for the new data to be produced.

Program Order

Sequential consistency refers to “operations of each individual processor .. occur in the order specified in its program” or **program order**.

In previous figure, this order is that of the stored machine instructions to be executed.

Compiler Optimizations

The order is not necessarily the same as the order of the corresponding high level statements in the source program as a compiler may reorder statements for improved performance. In this case, the term **program order** will depend upon context, either the order in the source program or the order in the compiled machine instructions.

High Performance Processors

Modern processors usually reorder machine instructions internally during execution for increased performance.

This does not alter a multiprocessor being sequential consistency, **if** the processor only produces the final results in program order (that is, retires values to registers in program order which most processors do).

All multiprocessors will have the option of operating under the sequential consistency model. However, it can severely limit compiler optimizations and processor performance.

Example of Processor Re-ordering

Process P1

```
.  
new = a * b;  
data = new;  
flag = TRUE;  
.br/>.br/>.br/>.
```

Process 2

```
.  
.br/>.br/>.br/>.br/>while (flag != TRUE) { };  
data_copy = data;  
.
```

Multiply machine instruction corresponding to `new = a * b` is issued for execution. The next instruction corresponding to `data = new` cannot be issued until the multiply has produced its result. However the next statement, `flag = TRUE`, is completely independent and a clever processor could start this operation before the multiply has completed leading to the sequence:

Process P1

```
.  
new = a * b;  
flag = TRUE;  
data = new;  
  
. .  
. .  
. .  
. .
```

Process 2

```
.  
. .  
. .  
. .  
  
. .  
while (flag != TRUE) { };  
data_copy = data;  
  
. .
```

Now the while statement might occur before new is assigned to **data**, and the code would fail.

All multiprocessors have the option of operating under the sequential consistency model, i.e. not reorder the instructions and forcing the multiply instruction above to complete before starting subsequent instruction which depend upon its result.

Relaxing Read/Write Orders

Processors may be able to relax the consistency in terms of the order of reads and writes of one processor with respect to those of another processor to obtain higher performance, and instructions to enforce consistency when needed.

Examples

Alpha processors

Memory barrier (MB) instruction - waits for all previously issued memory accesses instructions to complete before issuing any new memory operations.

Write memory barrier (WMB) instruction - as MB but only on memory write operations, i.e. waits for all previously issued memory write accesses instructions to complete before issuing any new memory write operations - which means memory reads could be issued after a memory write operation but overtake it and complete before the write operation. (check)

SUN Sparc V9 processors

memory barrier (MEMBAR) instruction with four bits for variations Write-to-read bit prevent any reads that follow it being issued before all writes that precede it have completed. Other: Write-to-write, read-to-read, read-to-write.

IBM PowerPC processor

SYNC instruction - similar to Alpha MB instruction (check differences)

Shared Memory Program Examples

Program

To sum the elements of an array, **a[1000]**:

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

UNIX Processes

Calculation will be divided into two parts, one doing even i and one doing odd i ; i.e.,

Process 1

```
sum1 = 0;  
for (i = 0; i < 1000; i = i + 2)  
sum1 = sum1 + a[i];
```

Process 2

```
sum2 = 0;  
for (i = 1; i < 1000; i = i + 2)  
sum2 = sum2 + a[i];
```

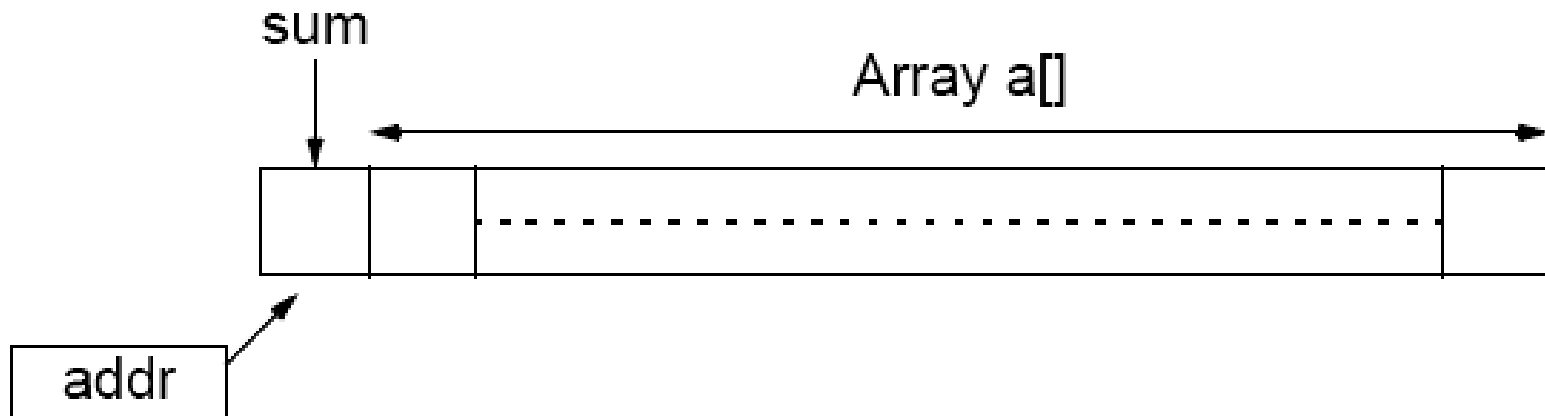
Each process will add its result (sum1 or sum2) to an accumulating result, sum :

```
sum = sum + sum1;
```

```
sum = sum + sum2;
```

Sum will need to be shared and protected by a lock. Shared data structure is created:

Shared memory locations for UNIX program example



```

#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;

    /* initialize semaphore set */

    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);
    }
}

```



```

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

/* map segment to process data space */
shm = shmat(shm, 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

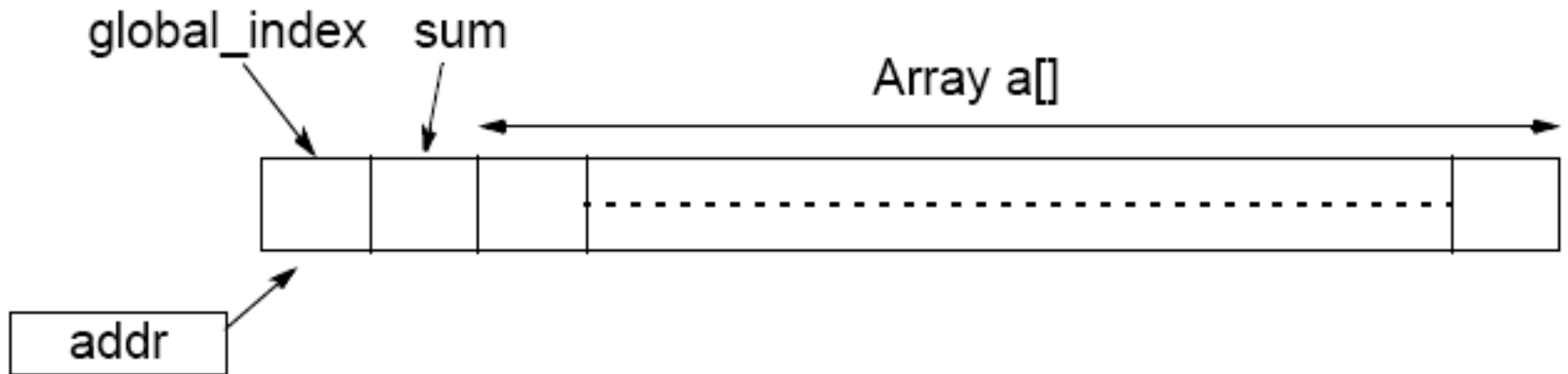
n threads created, each taking numbers from list to add to their sums. When all numbers taken, threads can add their partial results to a shared location sum.

The shared location `global_index` is used by each thread to select the next element of `a[]`.

After index is read, it is incremented in preparation for the next element to be read.

The result location is `sum`, as before, and will also need to be shared and access protected by a lock.

Shared memory locations for Pthreads 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

```
public class Adder
{
    public 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();
    }
}

{
    Adder a = new Adder();
}

}

```

```

    public static void main(String args[])

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);
    }
}

```