CSC4005 – Distributed and Parallel Computing

Prof. Yeh-Ching Chung

School of Data Science

Chinese University of Hong Kong, Shenzhen

Outline

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.

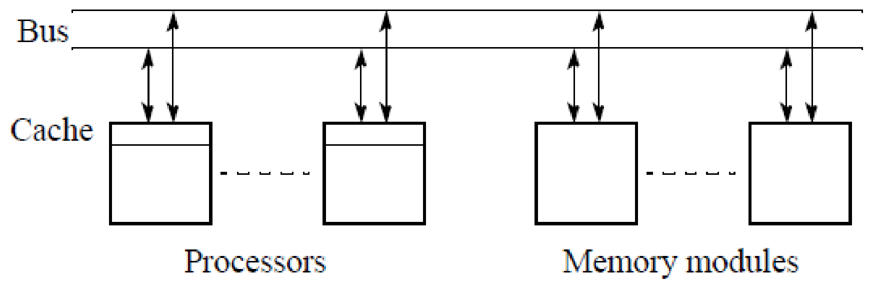


Figure 8.1 Shared memory multiprocessor using a single bus.

Several Alternatives for Programming (1)

- 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, ..)

Several Alternatives for Programming (2)

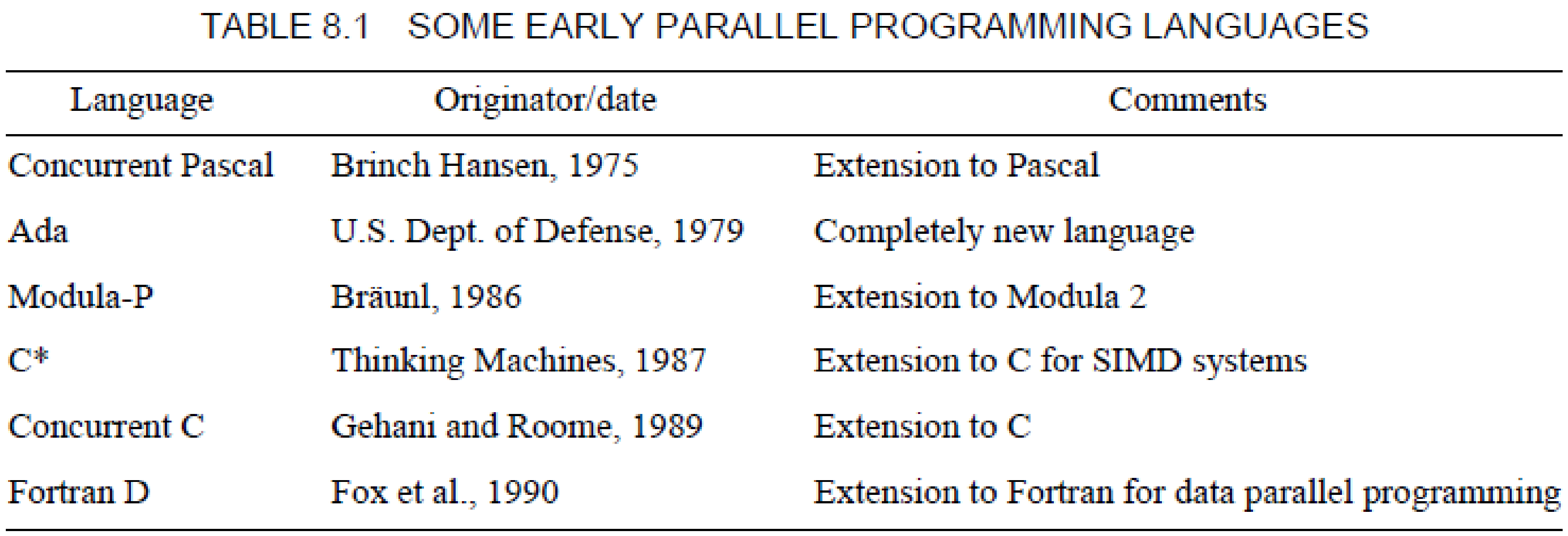


TABLE 8.1 SOME EARLY PARALLEL PROGRAMMING LANGUAGES

Language	Originator/date	Comments
Concurrent Pascal	Brinch Hansen, 1975	Extension to Pascal
Ada	U.S. Dept. of Defense, 1979	Completely new language
Modula-P	Bräunl, 1986	Extension to Modula 2
C*	Thinking Machines, 1987	Extension to C for SIMD systems
Concurrent C	Gehani and Roome, 1989	Extension to C
Fortran D	Fox et al., 1990	Extension to Fortran for data parallel programming

Creating Concurrent Process – FORK-JOIN Construct

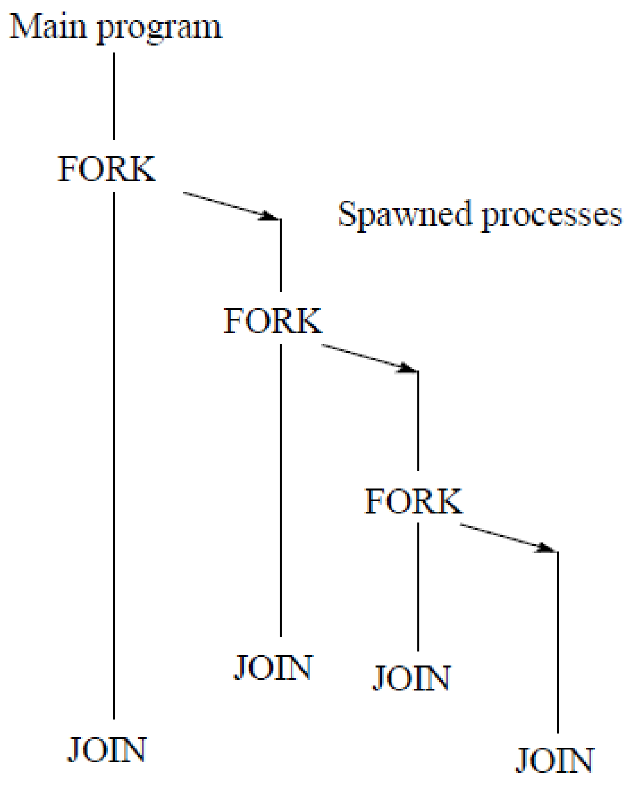


Figure 8.2 **FORK-JOIN** construct.

UNIX Heavyweight Processes (1)

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

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

UNIX Heavyweight Processes (2)

A single child process can be created by

```
                 .
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, could use

```
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);
```

Thread

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

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

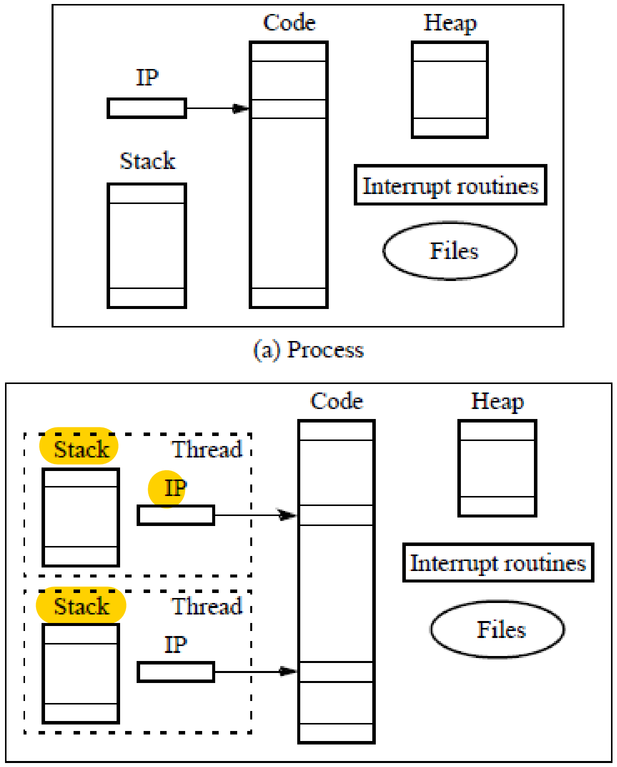


Figure 8.3 Differences between a process and threads.

Pthread

IEEE Portable Operating System Interface, POSIX, section 1003.1 standard

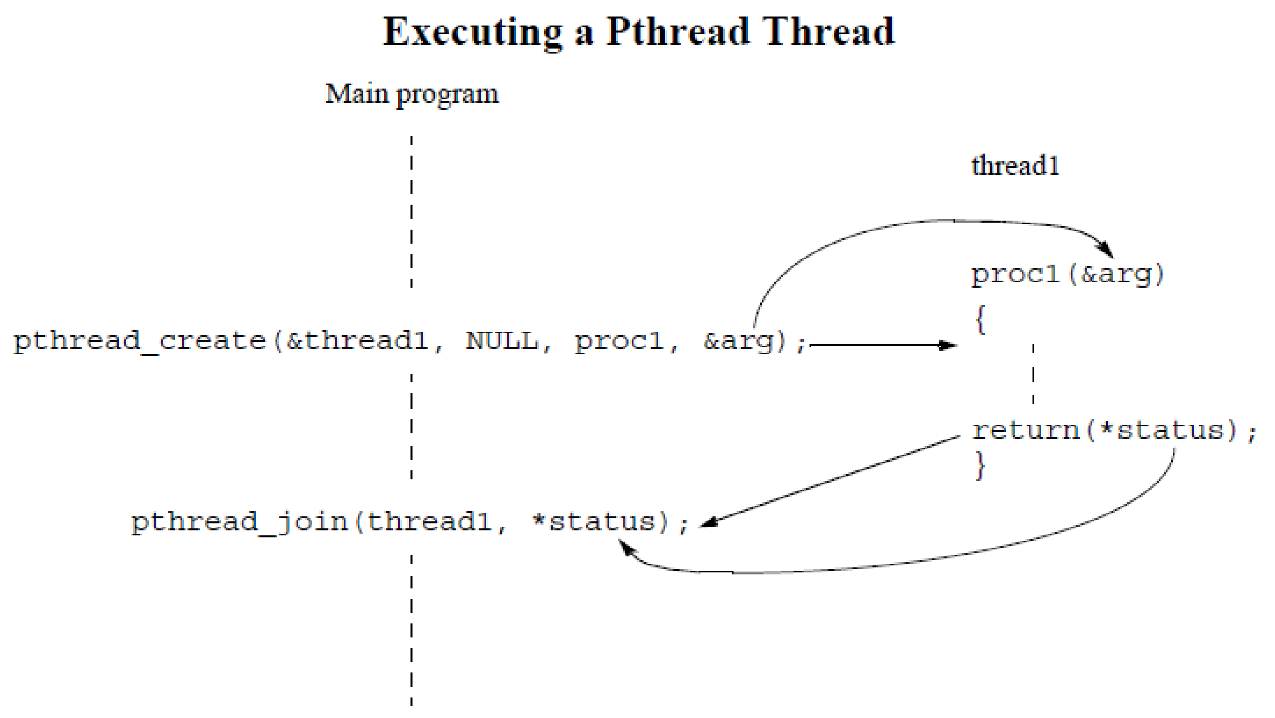

Executing a Pthread Thread
Main program
thread1
pthread_create(&thread1, NULL, proc1, &arg);
proc1(&arg)
{
return(*status);
}
pthread_join(thread1, *status);

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:

```
                  .
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 thread may not be bothered when a thread it creates terminates and in that case a join not be needed. Threads that are not joined are called *detached threads*.

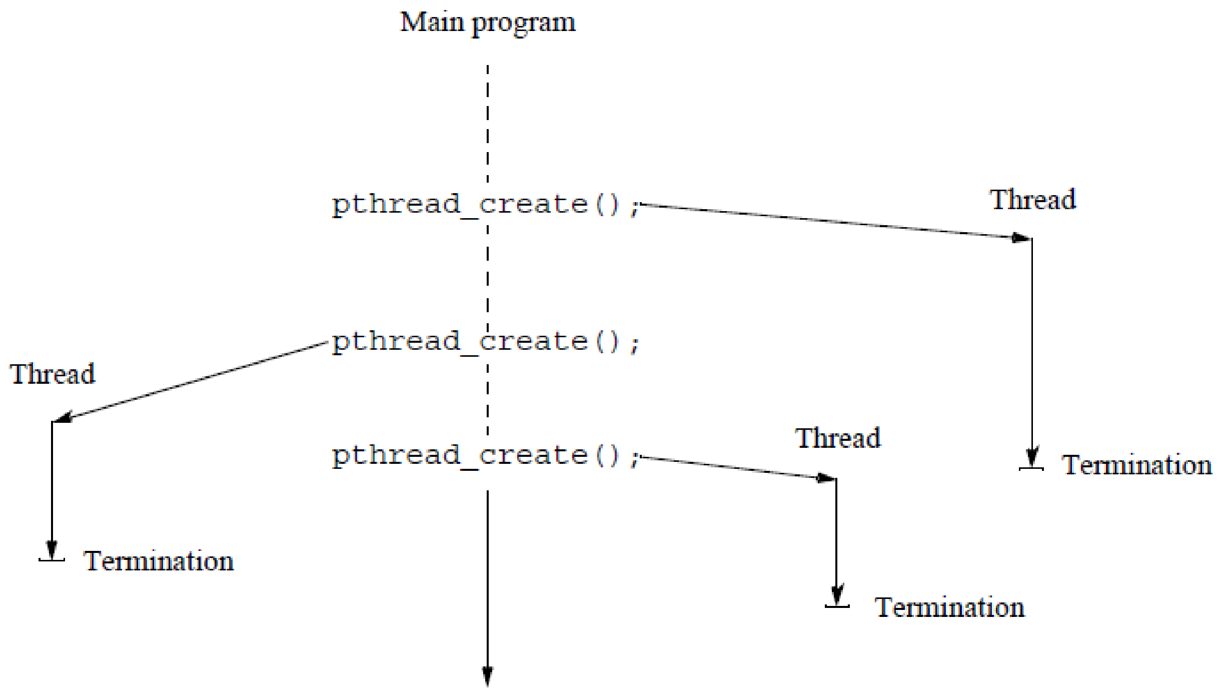


Figure 8.5 Detached threads.

Statement Execution Order (1)

On a multiprocessor system, instructions of individual processes/threads might be interleaved in time.

Example

Process 1	Process 2
Instruction 1.1	Instruction 2.1
Instruction 1.2	Instruction 2.2
Instruction 1.3	Instruction 2.3

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.

Statement Execution Order (2)

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 (or processor) might reorder instructions for optimization purposes.

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.

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`. 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 thread safe (prints messages without interleaving the characters).

Routines that access shared/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. 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

Time ↓	Instruction	Process 1	Process 2
	`x = x + 1;`	`read x`	`read x`
		`compute x + 1`	`compute x + 1`
		`write to x`	`write to x`

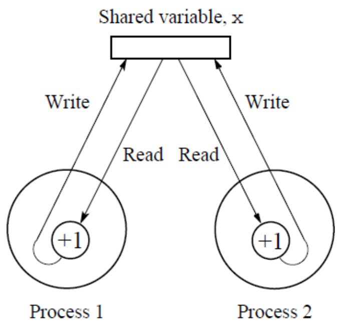


Figure 8.6 Conflict in accessing shared variable.

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.

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 (1)

Example

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

Spin Lock (2)

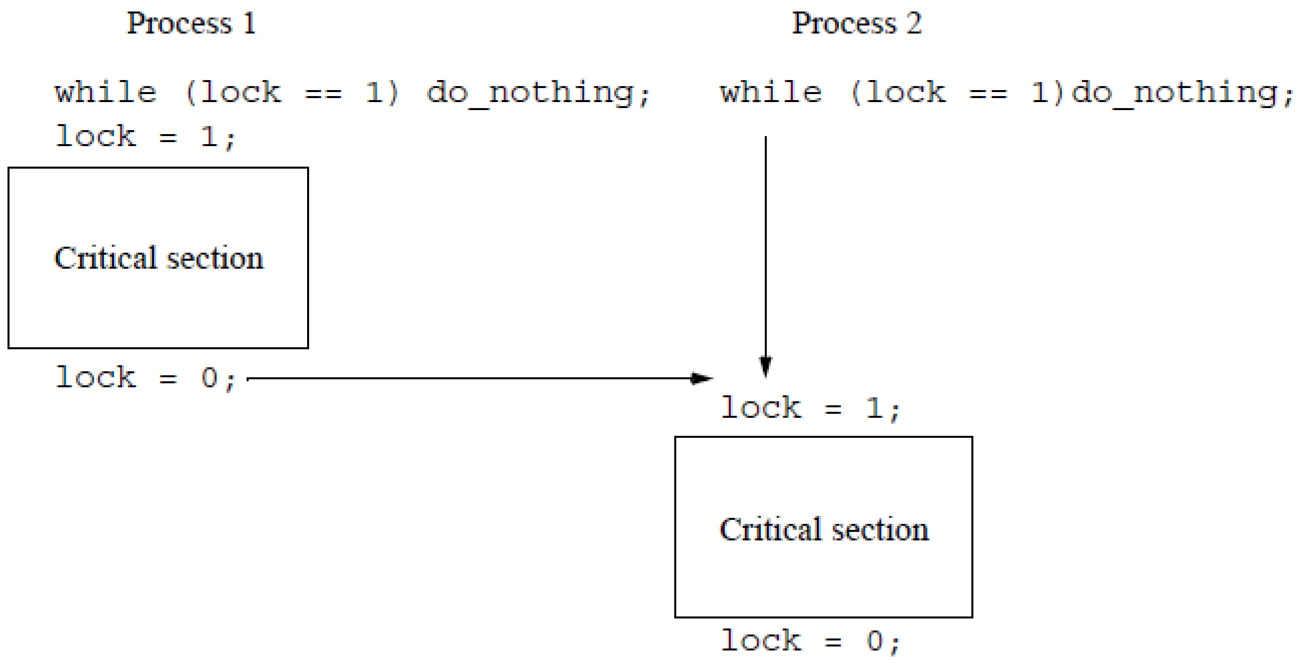


Figure 8.7 Control of critical sections through busy waiting.

Pthread Lock Routines (1)

Locks are implemented in Pthreads with *mutually exclusive lock* variables, or "mutex" variables.

To use mutex, it must be declared as of type `pthread_mutex_t` and initialized:

```
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()`.

Pthread Lock Routines (2)

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

```
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 (1)

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.

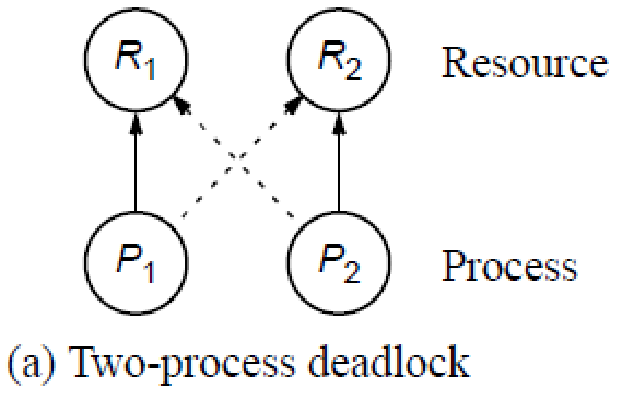
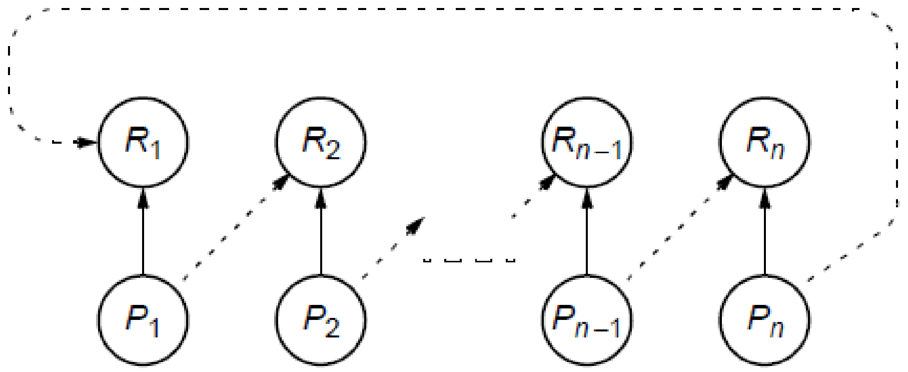


Figure 8.8 Deadlock (deadly embrace).

Deadlock (2)

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



(b) *n*-process deadlock

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.

Pthread Lock Checking Routine

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.

Semaphore

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

`P` operation, `P(s)`

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

`V` operation, `V(s)`

increments *s* by one to release one of the waiting processes (if any).

The `P` and `V` operations are performed indivisibly. Mechanism for activating waiting processes is implicit in `P` and `V` operations. Though exact algorithm is not specified, 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.

```
Process 1                Process 2                Process 3

Noncritical section      Noncritical section      Noncritical section
   .                        .                        .
P(s)                     P(s)                     P(s)
   .                        .                        .
Critical section         Critical section         Critical section
   .                        .                        .
V(s)                     V(s)                     V(s)
   .                        .                        .
Noncritical section      Noncritical section      Noncritical section
```

General Semaphore (1)

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.

General Semaphore (2)

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.,

```
monitor_proc1()
{
   P(monitor_semaphore);
          .
      monitor body
          .
   V(monitor_semaphore);
   return;
}
```

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.,

```
monitor_proc1()
{
  P(monitor_semaphore);
      .
    monitor body
      .
  V(monitor_semaphore);
  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.

This is a very time-consuming and unproductive exercise. The problem can be overcome by introducing so-called *condition variables*.

Condition Variable Operations (1)

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.

Condition Variable Operations (2)

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

```
action()                         counter()
{                                {
    .                                .
  lock();                          lock();
  while (x != 0)                   x--;
    wait(s);<--------------------  if (x == 0) signal(s);
  unlock();                        unlock();
  take_action();                     .
    .                                .
}                                }
```

Pthread Condition Variable

Associated with a specific mutex. Given the declarations and initializations:

```
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:

```
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 - threads must already be waiting for a signal to receive it.

Language Constructs for Parallelism (1)

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;
}
```

Language Constructs for Parallelism (2)

`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

We can see immediately in the code

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

hat every instance of the body is independent of the 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*.

Berstein'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_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, 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. The 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

$I_1 = (x, y) \quad O_1 = (a)$

$I_2 = (x, z) \quad O_2 = (b)$

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.

Shared Data in System with Caches (1)

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.

Shared Data in System with Caches (2)

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.

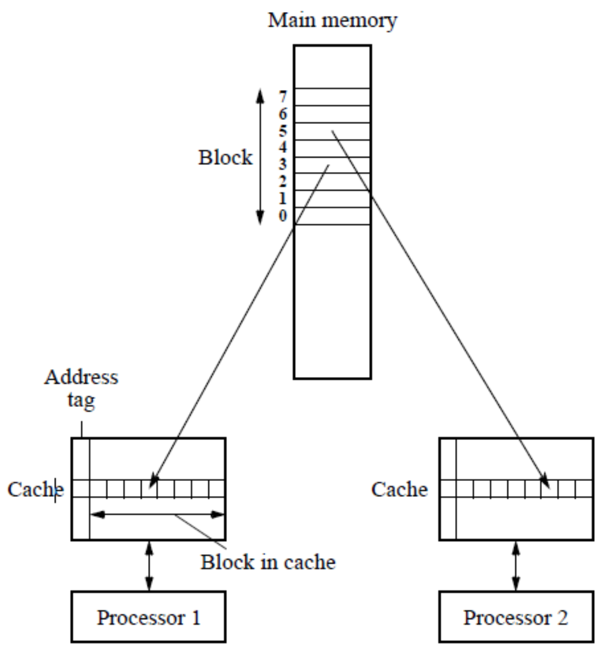


Figure 8.9 False sharing in caches.

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

Program Example (1)

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];
```

Program Example (2)

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

```
Process 1                           Process 2

sum1 = 0;                           sum2 = 0;
for (i = 0; i < 1000; i = i + 2)    for (i = 1; i < 1000; i = i + 2)
      sum1 = sum1 + a[i];                 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:

Program Example (3)

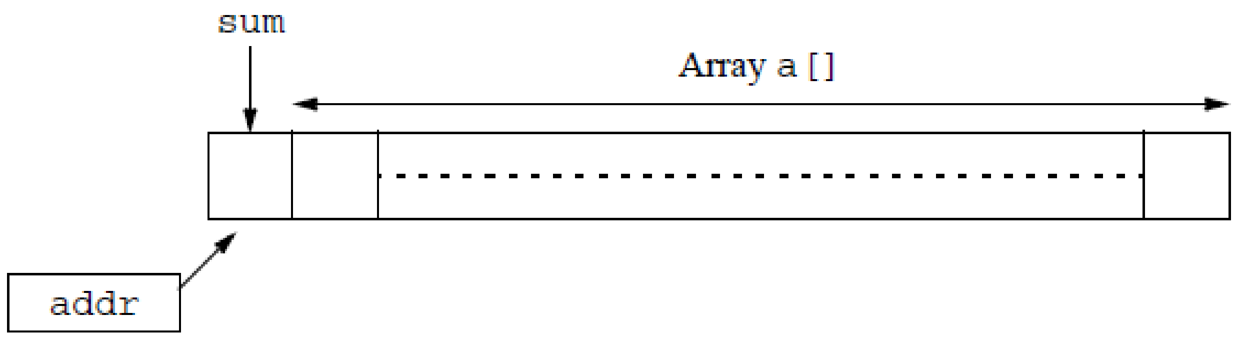


Figure 8.10 Shared memory locations for UNIX program example.

Program Example (4)

```
#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);
}
```

Program Example (5)

```
                                         /* 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);
}
```

Program Example (6)

```
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);
```

Program Example (7)

```
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 */
```

Program Example (8)

```c
void P(int *s)
{
   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;
}
```

```c
void V(int *s)
{
   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
```

Pthread Example (1)

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 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.

Pthread Example (2)

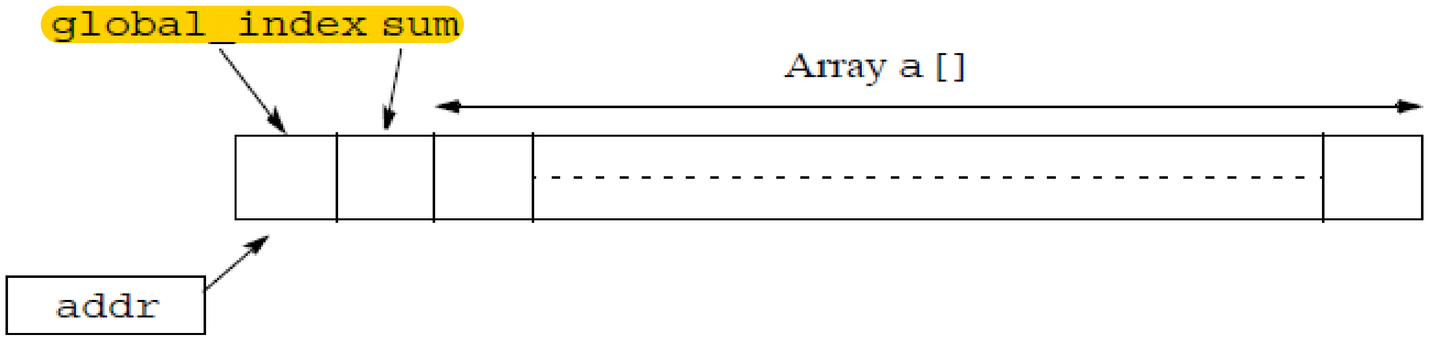


Figure 8.11 Shared memory locations for Pthreads program example.

Pthread Example (3)

```
#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 */
}
```

Pthread Example (4)

```
  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
```