Chapter 4, Problem 1PE

(5)

Step-by-step solution

Show all steps

100% (5 ratings) for this solution

Step 1/1

Multithreading in programming languages

To execute any application program, a number of threads (light weight processes) are used. This is done, because if any single thread is blocked, the application will not stop. Most of the programming languages uses the concept of multithreading. The examples to show the use of multithreading in programming languages are as given below:

• In a web server environment, a number of users request for the application. Thus, when any application is executed through the web server, a number of different threads are executed for different users.

• In graphic user interface, different threads are executed for different tasks. For example, first thread is used to fix the bugs, the second thread is responsible for the execution of the program and another one is responsible for the performance of the program.

If any program is executed using only one thread, then, if that thread is blocked, the total application will stop responding. So, it can be said that the performance of multi-threading is much better than system with single thread.

Chapter 4, Problem 2PE

(8)

Step-by-step solution

Show all steps

100% (18 ratings) for this solution

Step 1/3

Following are the differences between user-level threads and kernel-level threads:

User-level threads	**Kernel-level threads**
The existence of user-level threads is unknown to the kernel.	The existence of kernel-level threads is known to the kernel.

User-level threads are managed without kernel support.	Kernel-level threads are managed by the operating system.
User-level threads are faster to create than are kernel-level threads.	Kernel-level threads are slower to create than are user-level threads.
User-level threads are scheduled by the thread library.	Kernel-level threads are scheduled by the kernel.

Step 2/3

Circumstances where kernel-level threads are better than user-level threads:

• If the kernel is single-threaded, then kernel-level threads are better than user-level threads, because any user-level thread performing a blocking system call will cause the entire process to block, even if other threads are available to run within the application.

For example a process P1 has 2 kernel level threads and process P2 has 2 user-level threads. If one thread in P1 gets blocked, its second thread is not affected. But in case of P2 if one thread is blocked (say for I/O), the whole process P2 along with the 2nd thread gets blocked.

• In a multiprocessor environment, the kernel-level threads are better than user-level threads, because kernel-level threads can run on different processors simultaneously while user-level threads of a process will run on one processor only even if multiple processors are available.

Step 3/3

Circumstances where user-level threads are better than kernel-level threads:

• If the kernel is time shared, then user-level threads are better than kernel-level threads, because in time shared systems context switching takes place frequently. Context switching between kernel level threads has high overhead, almost the same as a process whereas context switching between user-level threads has almost no overhead as compared to kernel level threads.

Chapter 4, Problem 3PE

(4)

Step-by-step solution

Show all steps

100% (8 ratings) for this solution

Step 1/2

Kernel:

- Kernel is the head of any operating system.

- Several threads with different tasks are executed by the kernel.

- The interrupts generated by multi-threading are handled by kernel.

Step 2/2

The Role of kernel in threading are as follows:

When a context switch is occurred among the threads of kernel level, the kernel suspends all the threads and their respective values from the registers of the central processing unit (CPU). The newly scheduled threads restore the values in the registers.

Thus, due to this the current process state is also saved as well as the state og the incoming processes can also be restored.

For a kernel, the execution time is the main issue. If the new processes are created for the execution, rather than threads, it becomes more time consuming. So, the context stored in the registers of the CPU is switched.

Chapter 4, Problem 4PE

(2)

Step-by-step solution

Show all steps

100% (7 ratings) for this solution

Step 1/1

2481-4-10E SA: 8683

SR: 4578

A thread is a basic unit of CPU utilization, also known as light weight process. So, creating either a user or kernel thread involves allocating a small data structure to hold a thread ID, a program counter, a register set, stack and priority, whereas creating a process involves allocating the large data structure called as PCB(Process Control Block) which includes a memory map, list of open files, and environment variables. Allocating and managing the memory map is typically the most time-consuming activity.

Resources are used when a thread is created:

1. Code section

2. Data section and

3. Other operation system resources such as open files and signals with other threads belonging to the same process whereas each and every process has separate code section, data section and operating system resources.

Chapter 4, Problem 5PE

(4)

Step-by-step solution

Show all steps

100% (4 ratings) for this solution

Step 1/1

Threads with real time system

In an application which is based on real time processing, time plays an important role for execution. Threads are the parts of the process and also known as light weight process (LWP).

When a thread is noticeable as the thread of real time but a Light weight process is not bounded with this thread, then, the thread must have to be waiting for the attachment of LWP.

When a LWP is attached to any thread, the priority of that thread becomes higher. So, in a condition of dead lock, if a thread is having LWP, the thread has a minimum waiting time for the execution and time is saved. Thus, it can be said that the LWP attached thread is required in the real time systems.

Chapter 4, Problem 6E

(6)

Step-by-step solution

Show all steps

100% (5 ratings) for this solution

Step 1/1

The programming examples in which multithreading do not provide better performance than a single-threaded solution is as follows:

- In the linear programming, the multithreading does not provide better performance than a single-threaded solution. In the single-threaded solution, the linear programing has better performance. Programs like, computing the factorial of a number, sorting numbers, and minimizing the linear equations provide better performance using the single thread.

- The multithreading does not provide better performance in the “shell programs”. The shell programs use the single thread to execute the program. C-shell and Korn shell programs do not provide better performance in multithreading.

Chapter 4, Problem 7E

(4)

Step-by-step solution

Show all steps

Step 1/3

Solution:

In a single-processor system, the multithreaded solution uses the multiple kernels than a single-threaded solution for better performance because of the **following scenarios**:

- In a single-processor system, when the system **suffers a page fault** the kernel thread switches into another kernel thread while running the process.

- It leaves the time in a useful manner.

- Also, for the single-threaded solutions, when the page fault occurs, it is not capable to perform the process on the useful note in the system.

- It leads to the **waiting time and suffers the events in the process of the system**.

These are the scenarios does the multithread solutions perform better than the single-thread solutions on a single-processor system.

Step 2/3

For instance,

- If the user schedules the process in the kernel threads, if one kernel thread is blocked, it automatically switches to the other.

- It is **efficient for the system to schedule user level processes to the kernel threads**.

- For scheduling purposes, some OS assigns the user-level threads to the kernel threads.

- Multithreaded solution performs faster on the single-processor system by using kernel threads.

- It is **much more complicated for the single-threaded solution to achieve** the efficiency achieved by the multi-threaded solution.

- The advantage that deals are multi-tasking processes in the system.

Step 3/3

Two programming examples for the better performance is as follows:

1) Sequential programs are not good for candidates for multiprogramming. Calculating individual Tax returns is one of the examples for this type of program.

2) The second example can be any shell programs like C-shell or Korn shell. These programs monitor its own working places such as open files, current working directory, and environment variables.

✱ ortak veri alanı kullanan Programlar, Linear Programlar multithreading'e daha uygundur.

Chapter 4, Problem 8E

(5)

Step-by-step solution

Show all steps

100% (22 ratings) for this solution

Step 1/1

Heap memory and Global variables shared across in multi-thread process. In multi thread process, each thread has its own stack and register values.

Chapter 4, Problem 9E

(4)

Step-by-step solution

Show all steps

84% (6 ratings) for this solution

Step 1/2

Thus, a multithreaded system which has multiple user-level threads, cannot use different processors in multiprocessor system at the same time. Only a single process is visible to the operating system and the different threads of the process, present on separate processors are not scheduled.

Thus, on executing multiple user-level threads on multiprocessor system, no performance benefit occurs.

Step 2/2

Thus, user level threads cannot take advantage of multiple processors. They would perform the same.

Chapter 4, Problem 10E

(2)

Step-by-step solution

Show all steps

100% (10 ratings) for this solution

Step 1/2

Processes and **threads** both can typically be defined as sequences of execution, for any program. The main difference lies beneath the memory space they share on the RAM or main memory of the system while getting executed.

• Each process acquires a unique address space on the memory; whereas threads of a program usually share the same memory space for their execution.

• Processes act independently while threads always remain dependent on their consequent thread for their execution. Threads remain a chain of instructions where a small breakdown may terminate the entire execution.

Step 2/2

Google Chrome Browser maintains to open each new website as an independent process rather than opening them as threads to ensure that no website breakdown affects others' service.

One cannot maintain the efficiency of the browser while opening each new website as a thread, because threads are allocated with shared memory spaces. Hence they may affect each other if one of them crashes unexpectedly.

Shared memory

Chapter 4, Problem 11E

(7)

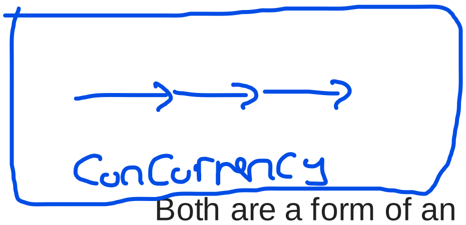
Step-by-step solution

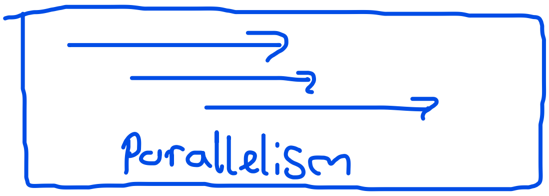
Show all steps

100% (8 ratings) for this solution

Step 1/2

Concurrency but not Parallelism





Both are a form of an operating system. Sometimes, to complete a task, both methods needed to finish the task and sometimes one. The priority to select, which form is better, depends upon the requirement of system and according to that operating system coding created.

Yes, it is possible to have concurrency but not parallelism.

Concurrency:

Concurrency means where two different tasks or threads start working together in an overlapped time period, however, it does not mean that they run at same instant.

Example:

Multi-task switch on a single-cored processor

In a Concurrency, minimum two threads are to be executed for processing.

A more generalized form of parallelism includes time-slicing which is a form of virtual parallelism.

Explanation:

Consider a Scenario, where Process 'A' and 'B' each have four different tasks P1, P2, P3, and P4. So in order to go for execution, first Process 'A' of P1 executes, then Process 'B' of P1 executes, Secondly, Process 'A' of P2 executes, then Process 'B' of P2 executes and goes on until all the threads are finished for their execution.

Step 2/2

Parallelism:

Parallelism is where two or more different tasks start their execution at the same time. It means that the two tasks or threads start working simultaneously.

Example:

Multi-cored Processor

Explanation:

Consider a Scenario, where Process 'A' and 'B' and each have four different tasks P1, P2, P3, and P4, so both process go for simultaneous execution and each works independently.

Therefore, concurrency can be occurring number of times which are same as parallelism if the process switching is quick and rapid. So, yes, it is possible to have concurrency but not parallelism.

Chapter 4, Problem 12E

(3)

Step-by-step solution

Show all steps

94% (16 ratings) for this solution

Step 1/3

As per Amdahl’s law, formula for the speedup gain of an application is $\text{speedup} \le 1/(S+(1-S)/N)$ where, S is the portion of the application that must be performed serially and N is the no of processing cores.

Step 2/3

(a)

For two processing cores and 60 percent parallel component, S is 40 percent that is 0.4 and N is 2.

$\text{speedup} \le 1/(0.4+(1-0.4)/2)$

$\text{speedup} \le 1.428$

Speedup gain is **1.428** times.

Step 3/3

(b)

For four processing cores and 60 percent parallel component

Here, S is 40 percent that is 0.4 and N is 4.

$\text{speedup} \le 1/(0.4+(1-0.4)/4)$

$\text{speedup} \le 1.81$

Speedup gain is **1.81** times.

Chapter 4, Problem 13E

(4)

Step-by-step solution

Show all steps

75% (4 ratings) for this solution

Step 1/6

Refer to the question provided in Exercise 4.21 for defining the answer for part (a).

Refer to the Project 1 provided in chapter 4 for defining the answer of part (b).

Refer to the Project 2 provided in chapter 4 for defining the answer of part (b).

Refer to the multithreaded web server provided in Section 4.1.

Step 2/6

Task Parallelism: The task or thread is allocated across the multiple computing cores and each thread operates different operation.

Data Parallelism: The subset of same data is allocated across the multiple computing cores and each core operates same operation on those data.

Step 3/6

a)

• Here, multiple threads are created and each thread will perform its dedicated task.

• For example, first thread is assigned to calculate average for all the numbers in the list.

• Whereas, second thread is performing the function to find minimum value and third thread is assigned to calculate maximum value in the program.

• **Hence, the problem exhibits Task Parallelism.**

Step 4/6

b)

• In Sudoku validator example, there are constraints that each row or column should contain the digits from 1 to 9. And each grid should have digits from 1 to 9 in its column, row and subgrids without repetitions.

• So, in this problem, different threads will perform the same task of checking digits from 1 to 9 but they will do this for different content.

• **Hence, the problem exhibits Data Parallelism.**

Step 5/6

c)

• Here, in sorting list, the list is divided in two halves by threads from process. Then, those two separate threads run concurrently and execute individually to generate two sorted sub lists from the global/original list. Thus, here it is using data parallelism.

• Then, third thread is also used which is combining the two sorted lists into single sorted list. Here, it is using task parallelism.

• Hence, this can be interpreted as the example of hybrid parallelism, that is, **it is using both data parallelism and task parallelism.**

Step 6/6

d)

• As described, the multithreaded web server creates different threads in such a way that each thread is dedicated to perform a specific task. This leads to better performance of the server.

• For example, threads for tasks like receiving the requests of clients and completing those requests exists separately.

• **Hence, the problem exhibits Task Parallelism.**

Chapter 4, Problem 14E

(4)

Step-by-step solution

Show all steps

78% (9 ratings) for this solution

Step 1/1

Threads count depends upon the priority and requirements of the application. So only thread is enough for this kind of application and this thread is going to handle both input and output operation.

• It is a concurrency approach. Here, it only make sense to create as many threads as there are blocking system calls, as the threads will be spent blocking.

• It doesn’t provides any benefits to create an additional threads.

Thus, **only a single thread creation makes sense for input and a single thread for output.**

Four threads are created to perform the CPU-intensive portion of the application. It is because, there should be as many threads as there are processing cores.

- It would be the waste of processing resources to use fewer threads.

- Also any number greater than four would be unable to run.

Chapter 4, Problem 15E

(16)

Step-by-step solution

Show all steps

100% (18 ratings) for this solution

Step 1/2

Unique processes created by the given code segment:

The fork() system call allows a parent process to create a new process call child process.

- The statement pid = fork();before the if statement creates one process. The parent process say p0 creates this process. Let it be p1.

- The statement fork();in the if statement creates one process. The parent process p1 creates this process. Let it be p2.

- After the if statement, parent process p0, process p1 and process p2 will execute fork(); creating three new processes.

o One process is created by parent process p0.

o One process is created by process p1.

o One process is created by process p2.

- Let the three new processes created by p0, p1, and p2 are p4, p5, and p6.

Therefore, 6 unique processes (p0, p1, p2, p3, p4, p1, p5) will be created.

Hence, the number of unique processes created are $\boxed{6}$.

Step 2/2

Unique threads created by the given code segment:

- Thread creation is done in if block by the function pthread_create(). Only child process p1 is executed in the if block. Therefore, process p1 will create one thread.

- In the if block one process p2 is created using fork(). Therefore, process p2 will also create a thread.

Hence, the number of unique threads created are $\boxed{2}$.

Chapter 4, Problem 16E

(2)

Step-by-step solution

Show all steps

75% (8 ratings) for this solution

Step 1/2

Threads and processes in Linux:

Linux operating systems consider both threads and processes as tasks; it cannot able to distinguish between them. In contrast, windows operating system threads and processes differently.

This approach has pros and cons while modelling threads and processes inside the kernel.

Step 2/2

Pros:

- Linux consider this as similar, so codes belong to operating system can be cut down easily.

- Scheduler present in the Linux operating systems do not need special code to test threads coupled with each processes.

- It considers different threads and processes as a single task during the time of scheduling.

Cons:

- This ability makes it harder for the Linux operating system to inflict process-wide resource limitations directly.

- Extra steps are needed to recognize the each processes belong to appropriate threads and complexity in performing relevant tasks.

Chapter 4, Problem 17E

(9)

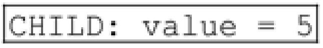
Step-by-step solution

Show all steps

95% (20 ratings) for this solution

Step 1/2

Output of LINE C in the program:

The output is  CHILD: value = 5 .

• The child process in the thread is forked by parent process; the parent process and child process each have its own memory space.

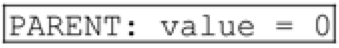
• After forking, the parent process waits for the completion of child process.

• New thread is created for child process and the runner() function is called which set the value of global variable to 5.

• Thus, after execution of this line, the value displayed will be $\boxed{5}$.

Step 2/2

Output of LINE P in the program:

The output is  PARENT: value = 0.

• After completing the child process, the value of the global variable present in parent process remains 0.

• Thus, after execution of this line, the value displayed will be $\boxed{0}$.

Chapter 4, Problem 18E

(9)

Step-by-step solution

Show all steps

100% (6 ratings) for this solution

Step 1/3

Performance implications multiprocessor system and a multithreaded program:

a) Kernel < Number of processors

The scheduler in the system can schedule only one kernel thread at a time to one user level process. Since, the system is many-to-many threading model, many user level processes and kernel threads will be idle.

The number of processors is greater than the kernel threads, processes can be mapped to kernel threads which is available and can be accessed quickly. Most of the processors will not be utilized and they will be idle.

Step 2/3

b) Kernel = Number of processors

The scheduler in the system can schedule one kernel thread at a time to one user level processor. Since, the system is many-to-many threading model, many user level processes and kernel threads will be idle.

The number of processors is equal to the kernel threads; hence all the user level processors run concurrently in the kernel threads assuming all kernel threads are free and none of them are blocked.

Step 3/3

c) Kernel > Number of processors

The scheduler in the system can schedule only one kernel thread at a time to one user level process. Since, the system is many-to-many threading model, many user level processes and kernel threads will be idle.

The number of processors is lesser than the kernel threads, processes can be mapped to kernel threads which is available and can be accessed quickly. If a kernel thread is busy, the user level processes will be swapped with kernel threads which are idle.

Chapter 4, Problem 19E

(2)

Step-by-step solution

Show all steps

Step 1/3

Thread cancellation

Thread cancellation is a process of cancelling thread before its completion. Thread cancellation leads to termination of the executing thread in the process.To cancel a thread, below given functions are used:

• Inorder to cancel thread, pthread_cancel() function is used.

• pthread_cancel() functions depends on pthread_setcancelstate() function.

• To cancel thread immediately PTHREAD_CANCEL_ASYNCHRONOUS type is set.

- The default cancellation type is PTHREAD_CANCEL_DEFERED which means thread is cancelled only when it reaches its termination point.

Step 2/3

Thread cancellation state and type

Thread cancellation state and type determine when the thread cancellation request is placed. There are two states in thread cancellation:

- PTHREAD_CANCEL_DISABLE in all cancellation state are held pending.

- PTHREAD_CANCEL_ENABLE in which cancellations requests are acted on according to thread cancellation types.

- The default thread cancellation type is PTHREAD_CANCEL_DISABLE.

Step 3/3

Thread cancellation points

The system test for pending cancellation requests in certain blocking functions, if cancellation function is ENABLED and its type is DEFERED. These points are known as cancellation points.

pthread_testcancel() function is used to create cancellation points.

Chapter 4, Problem 21PP

(11)

Step-by-step solution

Show all steps

100% (2 ratings) for this solution

Step 1/5

Program Plan:

- Declare the three function prototypes and declare global variables average, minimum and maximum.

- Define the structure to hold the length of the array and the array pointer.

- In main method Check whether input arguments are given or not by checking with argc variable.

- If argc is less than 1, then display the message.

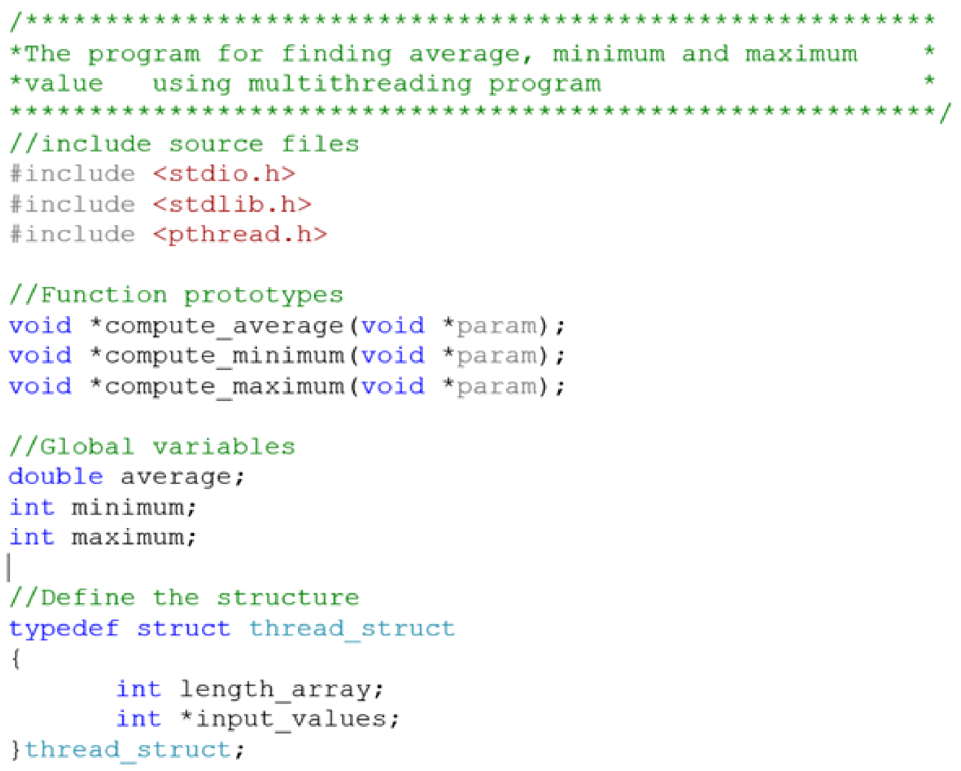
- Create three separate threads using pthread_create method to perform the average, minimum and maximum from the command line input numbers.

- Implement each individual method to compute three statistical functions and return the results to the main method.

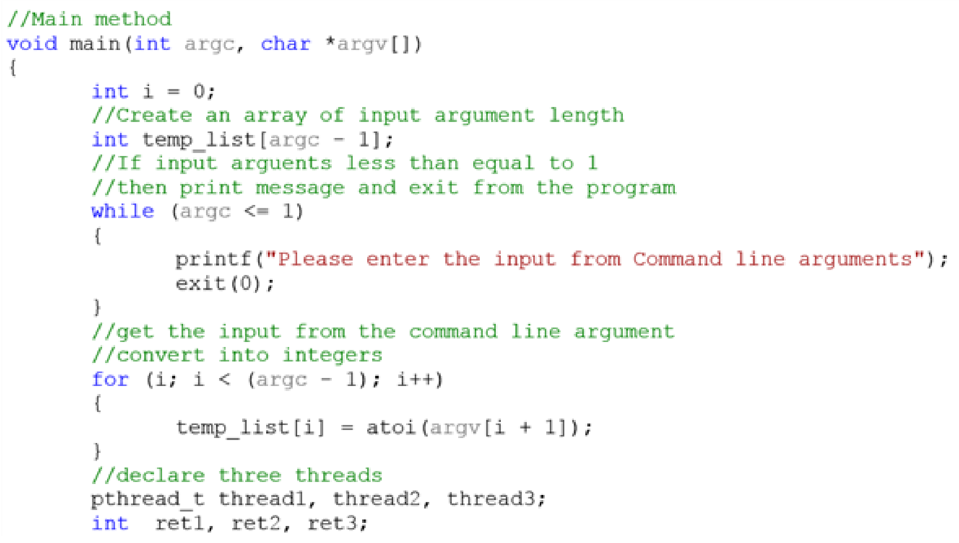
- Print the return results from the functions in main method.

Step 2/5

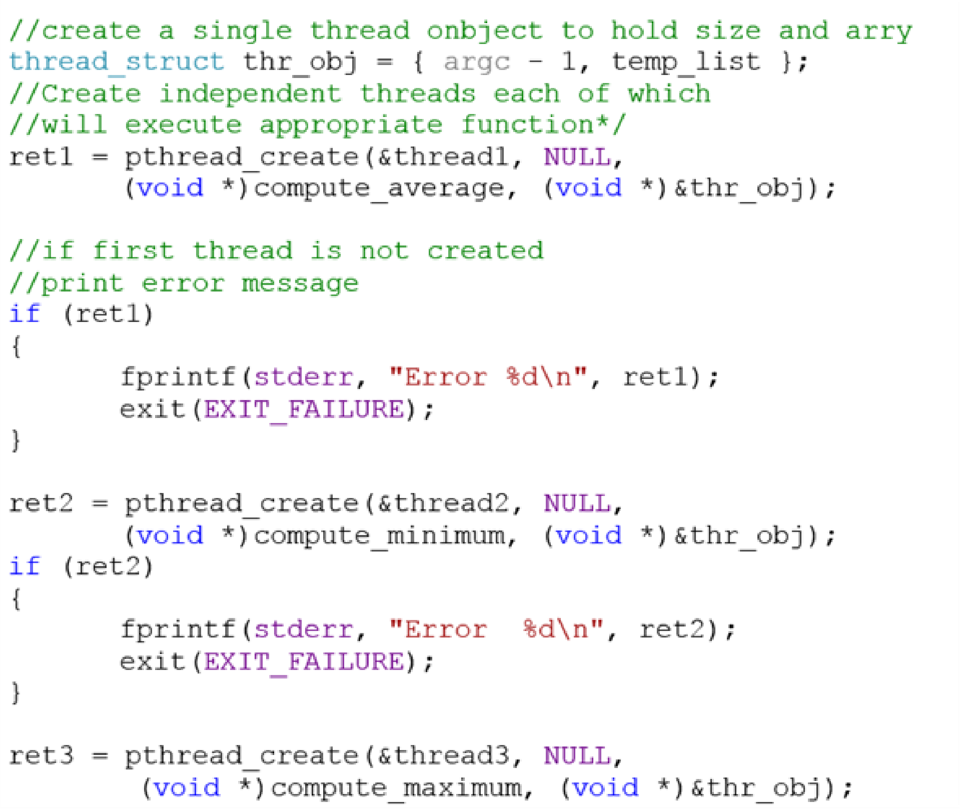
Program:



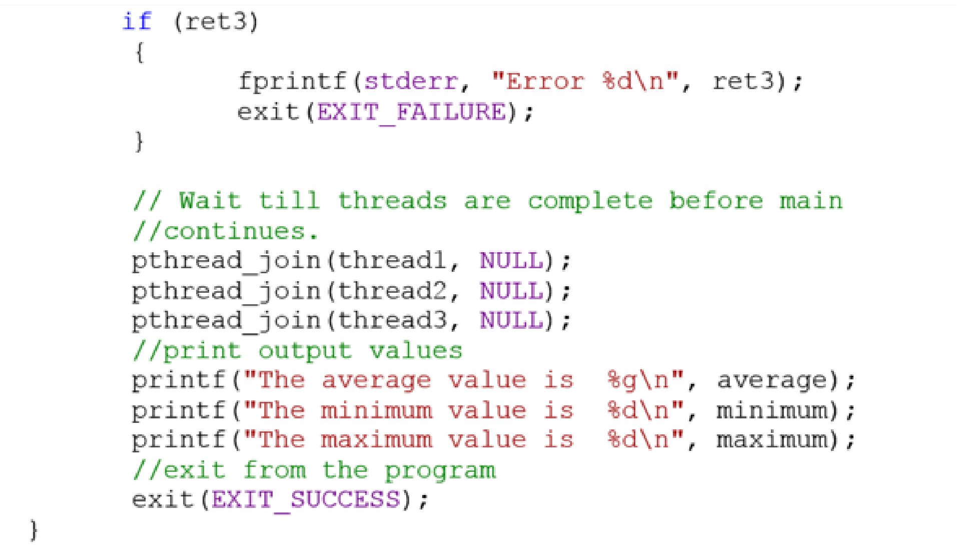
```
/**********************************************************
*The program for finding average, minimum and maximum    *
*value   using multithreading program                    *
**********************************************************/
//include source files
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

//Function prototypes
void *compute_average(void *param);
void *compute_minimum(void *param);
void *compute_maximum(void *param);

//Global variables
double average;
int minimum;
int maximum;

//Define the structure
typedef struct thread_struct
{
       int length_array;
       int *input_values;
}thread_struct;
```



```
//Main method
void main(int argc, char *argv[])
{
       int i = 0;
       //Create an array of input argument length
       int temp_list[argc - 1];
       //If input arguents less than equal to 1
       //then print message and exit from the program
       while (argc <= 1)
       {
              printf("Please enter the input from Command line arguments");
              exit(0);
       }
       //get the input from the command line argument
       //convert into integers
       for (i; i < (argc - 1); i++)
       {
              temp_list[i] = atoi(argv[i + 1]);
       }
       //declare three threads
       pthread_t thread1, thread2, thread3;
       int  ret1, ret2, ret3;
```

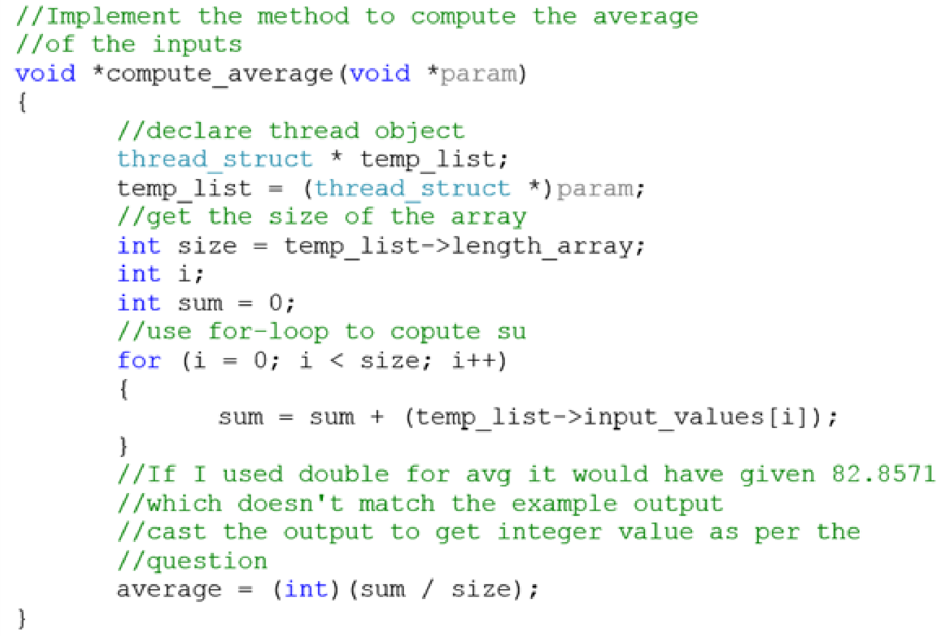


```
//create a single thread onbject to hold size and arry
thread_struct thr_obj = { argc - 1, temp_list };
//Create independent threads each of which
//will execute appropriate function*/
ret1 = pthread_create(&thread1, NULL,
       (void *)compute_average, (void *)&thr_obj);

//if first thread is not created
//print error message
if (ret1)
{
       fprintf(stderr, "Error %d\n", ret1);
       exit(EXIT_FAILURE);
}

ret2 = pthread_create(&thread2, NULL,
       (void *)compute_minimum, (void *)&thr_obj);
if (ret2)
{
       fprintf(stderr, "Error  %d\n", ret2);
       exit(EXIT_FAILURE);
}

ret3 = pthread_create(&thread3, NULL,
        (void *)compute_maximum, (void *)&thr_obj);
```

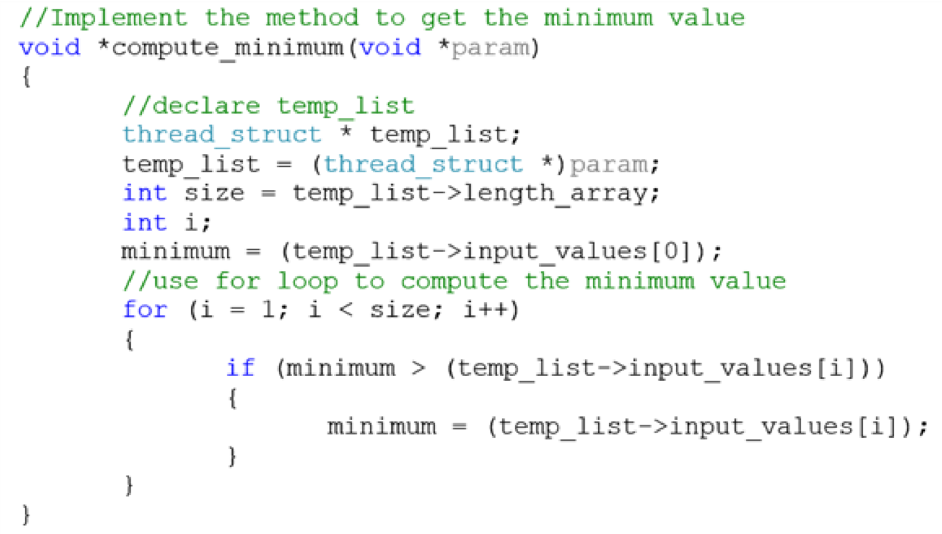


```
       if (ret3)
        {
               fprintf(stderr, "Error %d\n", ret3);
               exit(EXIT_FAILURE);
        }

        // Wait till threads are complete before main
        //continues.
        pthread_join(thread1, NULL);
        pthread_join(thread2, NULL);
        pthread_join(thread3, NULL);
        //print output values
        printf("The average value is  %g\n", average);
        printf("The minimum value is  %d\n", minimum);
        printf("The maximum value is  %d\n", maximum);
        //exit from the program
        exit(EXIT_SUCCESS);
 }
```

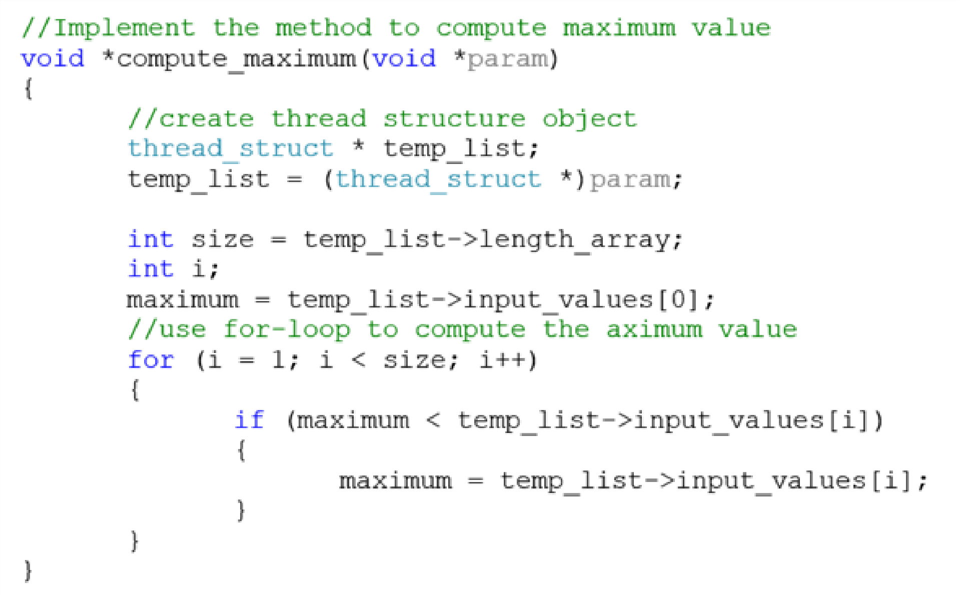
Step 3/5



```
//Implement the method to compute the average
//of the inputs
void *compute_average(void *param)
{
        //declare thread object
        thread_struct * temp_list;
        temp_list = (thread_struct *)param;
        //get the size of the array
        int size = temp_list->length_array;
        int i;
        int sum = 0;
        //use for-loop to copute su
        for (i = 0; i < size; i++)
        {
                sum = sum + (temp_list->input_values[i]);
        }
        //If I used double for avg it would have given 82.8571
        //which doesn't match the example output
        //cast the output to get integer value as per the
        //question
        average = (int)(sum / size);
}
```



```
//Implement the method to get the minimum value
void *compute_minimum(void *param)
{
        //declare temp_list
        thread_struct * temp_list;
        temp_list = (thread_struct *)param;
        int size = temp_list->length_array;
        int i;
        minimum = (temp_list->input_values[0]);
        //use for loop to compute the minimum value
        for (i = 1; i < size; i++)
        {
                if (minimum > (temp_list->input_values[i]))
                {
                        minimum = (temp_list->input_values[i]);
                }
        }
}
```



```
//Implement the method to compute maximum value
void *compute_maximum(void *param)
{
        //create thread structure object
        thread_struct * temp_list;
        temp_list = (thread_struct *)param;

        int size = temp_list->length_array;
        int i;
        maximum = temp_list->input_values[0];
        //use for-loop to compute the aximum value
        for (i = 1; i < size; i++)
        {
                if (maximum < temp_list->input_values[i])
                {
                        maximum = temp_list->input_values[i];
                }
        }
}
```

Step 4/5

Sample Input:

Command line Arguments:

90 81 78 95 79 72 85

Step 5/5

Sample Output:

The average value is 82

The minimum value is 72

The maximum value is 95

Chapter 4, Problem 22PP

(8)

Step-by-step solution

Show all steps

Step 1/4

Program Plan:

• Define a random floating point number randf for calculating the value in range[-1,1].

• Define the method calc_num which takes one value as an argument and calculate the number of randomly chosen points that lie inside a circle with radius equal to 1.0.

- Define the main method from where the execution of the program begins.

- Checking the condition for the initialization of the pthreads

- Create the thread that will calculate the number of randomly chosen points that lie in the circle.

- Calculate the value of PI and print it out.

Step 2/4

Program:

```
/******************************************************** Program for calculating the value of pie using the * * technique Monte Carlo which basically involves * * randomization. * ********************************************************/
```

Include a header file for performing the operation.

```
#include

#include

#include

#include

// The number of times the Monte Carlo technique

// will be used

const int COUNT = 10000;

// The number of times that the randomly chosen

// point lies inside the circle

static int num;
```

Declare the method randf which takes void as arguments and calculate a random floating point number in the range [-1, 1].

```
float randf(void);
```

Declare the method calc_num which takes one value as an arguments and the start function for the thread that determines the number of points that lie inside the circle.

```
void *calc_num(void *param);
```

Define the main method from where the execution of the program begins.

```
int main(void)
{
// The pthread attributes
pthread_attr_t attr;
// The id of the thread
pthread_t tid;
// Initialize the random number generator
// with the current time
srand(time(NULL));
```

Checking the condition for the initialization of the pthreads

```
// Initialize pthreads
if (pthread_attr_init(&attr) != 0) {
fputs("pthread_attr_init didn't work\n",
stdout);
return 1;
}
// Zero out the number of points that lie
// in the circle
num = 0;
```

Create the thread that will calculate the number of randomly chosen points that lie in the circle.

```
if (pthread_create(&tid, &attr, calc_num, NULL)
!= 0)
{
fputs("Couldn't create thread\n", stdout);
// De-initialize pthreads
```

```
pthread_attr_destroy(&attr);

return 2;

}

// Wait for the thread to exit

if (pthread_join(tid, NULL) != 0) {

fputs("pthread_join didn't work for "

"the thread", stdout);

// De-initialize pthreads

pthread_attr_destroy(&attr);

return 3;
```

Step 3/4

```
}

// Calculate the value of PI and print it out

printf("PI = %f\n", 4.0f * num / COUNT);

// De-initialize pthreads

pthread_attr_destroy(&attr);

return 0;

}
```

Define a random floating point number randf for calculating the value in range[-1,1].

```
float randf(void)

{

return ((float) rand()) / RAND_MAX * 2.0f - 1.0f;

}
```

Define the method calc_num which takes one value as an argument and calculate the number of randomly chosen points that lie inside a circle with radius equal to 1.0.

```
void *calc_num(void *param)

{
```

```
// Count how many times the points have been chosen
int index;
```

Iterate the loop for counting the number of times.

```
for (index = 0; index < COUNT; index++)
{
// Get a random x co-ordinate
float x = randf();
// Get a random x co-ordinate
float y = randf();
// Is the square of the length of line between
// (x, y) and the origin less than or equal to
// the square of the radius
if (x * x + y * y <= 1.0f)
num++;
}
// Exit from this thread
pthread_exit(0);
}
```

Step 4/4

Sample Output:

```
11:35 4$ ./13258-4-22PP
PI = 3.143200
11:35 4$ ./13258-4-22PP
PI = 3.120000
11:35 4$ ./13258-4-22PP
PI = 3.136800
```

```
11:35 4$ ./13258-4-22PP

PI = 3.136400
```

Chapter 4, Problem 23PP

(3)

Step-by-step solution

Show all steps

Step 1/4

Program Plan:

• Define a random floating point number randf for calculating the value in range[-1,1].

• Define the method calc_num which takes one value as an argument and calculate the number of randomly chosen points that lie inside a circle with radius equal to 1.0.

• Define the main method from where the execution of the program begins.

• Initialize the random number generator with the current time.

• Calculate the number of random points that are inside the circle.

• Calculate the value of PI and print it out.

Step 2/4

Program:

```
/**********************************************************  Program for calculating the value of pie using the * * technique OpenMP for parallelize the generation of * * points. * *********************************************************/
```

Include a header file for performing the operation.

```
#include

#include

#include

#include

// The number of times the Monte Carlo technique

// will be used

const int COUNT = 10000;
```

```
// The number of times that the randomly chosen

// point lies inside the circle

static int num;
```

Declare the method randf which takes void as arguments and calculate a random floating point number in the range [-1, 1].

```
float randf(void);
```

Declare the method calc_num which takes one value as an arguments and the start function for the thread that determines the number of points that lie inside the circle.

```
void calc_num(void);
```

Define the main method from where the execution of the program begins.

```
int main(void)

{

// Initialize the random number generator

// with the current time

srand(time(NULL));

// Zero out the number of points that lie

// in the circle

num = 0;

// Calculate the number of random points that

// are inside the circle

calc_num();

// Calculate the value of PI and print it out

printf("PI = %f\n", 4.0f * num / COUNT);

return 0;

}
```

Define a random floating point number randf for calculating the value in range[-1,1].

```
float randf(void)
```

```
{
return ((float) rand()) / RAND_MAX * 2.0f - 1.0f;
}
```

Define the method calc_num which takes one value as an argument and calculate the number of randomly chosen points that lie inside a circle with radius equal to 1.0.

```
void calc_num(void)
{
// Count how many times the points have been chosen
int index;
// Tell OpenMP that the for loop is to
// be executed in parallel pragma omp parallel for Loop
// for COUNT times
for (index = 0; index < COUNT; index++)
{
// Get a random x co-ordinate
float x = randf();
// Get a random x co-ordinate
float y = randf();
// Is the square of the length of line between
// (x, y) and the origin less than or equal to
```

Step 3/4

```
// the square of the radius
if (x * x + y * y <= 1.0f)
num++;
}
}
```

Step 4/4

Sample Output:

```
12:39 4$ ./13258-4-23PP

PI = 3.144800

12:39 4$ ./13258-4-23PP

PI = 3.134000

12:39 4$ ./13258-4-23PP

PI = 3.132400

12:39 4$ ./13258-4-23PP

PI = 3.134000
```

Chapter 4, Problem 24PP

(2)

Step-by-step solution

Show all steps

100% (2 ratings) for this solution

Step 1/3

Program plan:

- Include the required library files.
- Write a function to print the prime factorization.
- Write a function to run the prime computer.
- Write the loop to check the prime condition.
- Display the messages based on the factorization.
- Call the functions in the main.

Step 2/3

Program:

```
// include the required library files

#include
```

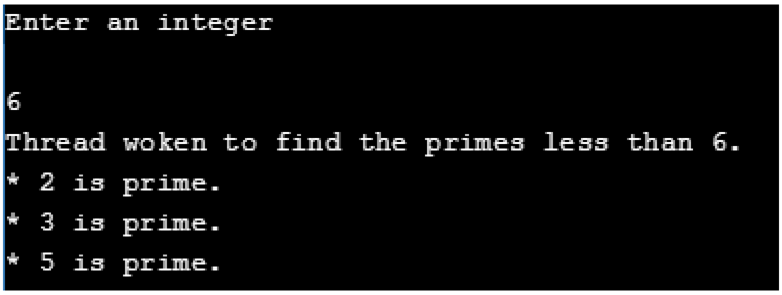
```
#include
#include
#include
#include
typedef struct {
pthread_cond_t start_working;
// mutex to protect the condition variable
pthread_mutex_t cond_mutex;
// number searched by the thread
long max_search;
} thread_parameters_t;
typedef struct {
long f[2];
} factor_t;
// function to print prime factorization.
void print_factorization(long n, factor_t *sieve) {
// declare the variable i
int i;
if (sieve[n].f[0]) {
for (i = 0; i < 2; ++i)
if (sieve[n].f[i])
print_factorization(sieve[n].f[i], sieve);
} else
printf(" %ld ", n);
}
// function to run the prime computer
```

```
void *primes_computer_runner(void *param) {
factor_t *sieve;
long i, prime, limit;
thread_parameters_t *thread_parameters = (thread_parameters_t*)param;
pthread_mutex_lock(&thread_parameters->cond_mutex);
pthread_cond_wait(&thread_parameters->start_working,
&thread_parameters->cond_mutex);
// thread goes to sleep state untill the condition is true
// it woken up when it reaches the next line and signaled
// from the main thread
printf("Thread woken to find the primes less than %ld.\n",
thread_parameters->max_search);
sieve = (factor_t*)calloc(thread_parameters->max_search, sizeof(factor_t));
assert(sieve != NULL);
// calloc allocates the memory and initialize it to zero
// use this zero to represent "is prime" according to the algorithm
limit = (long)sqrt(thread_parameters->max_search) + 1;
// checking for prime condition
for (prime = 2; prime <= limit; ++prime)
for (i=2; i*prime < thread_parameters->max_search; ++i) {
sieve[i*prime].f[0] = i;
sieve[i*prime].f[1]= prime;
}
// display the below message if the number is prime
for(i=2; i < thread_parameters->max_search; ++i)
if (!sieve[i].f[0])
```

```
printf("* %ld is prime.\n",i);

#ifdef SHOW_NONPRIME

//display the below message if the number is nonprime

else {

printf(" %ld is nonprime, factorization: (",i);

print_factorization(i, sieve);

puts(")");

}

#endif

free(sieve);

return NULL;

}

// main function begins

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

//pass a pointer to this struct to the computational thread

thread_parameters_t thread_parameters;

// Set the initial conditions of the thread to null

pthread_cond_init(&thread_parameters.start_working, NULL);

pthread_mutex_init(&thread_parameters.cond_mutex, NULL);

// This thread will do the computing of prime numbers.

pthread_t computational_thread;

// Create and start new thread.

pthread_create(&computational_thread, NULL, primes_computer_runner,

(void*)&thread_parameters);

puts("Enter an integer \n"

#ifdef SHOW_NONPRIME
"This program has been compiled to show prime factorizations of \n"
"the nonprimes it finds, with -DSHOW_NONPRIME in the Makefile.\n"
#endif
);
// Get the integer from the user
if (!scanf("%ld", &thread_parameters.max_search)) return 0;
// Wake up the thread to do the computation.
pthread_cond_broadcast(&thread_parameters.start_working);
//Wait for it to finish (makes this rather pointless, but I forgot
//about this assignment until the eleventh hour.)
pthread_join(computational_thread,NULL);
return 0;
}
```

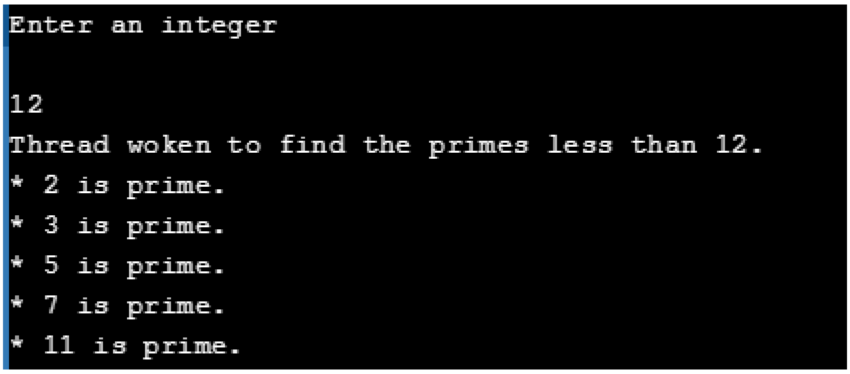
Step 3/3

Sample output 1:



```
Enter an integer

6
Thread woken to find the primes less than 6.
* 2 is prime.
* 3 is prime.
* 5 is prime.
```

Sample output 2:



```
Enter an integer

12
Thread woken to find the primes less than 12.
* 2 is prime.
* 3 is prime.
* 5 is prime.
* 7 is prime.
* 11 is prime.
```

Chapter 5, Problem 1PE

(7)

Step-by-step solution

Show all steps

100% (10 ratings) for this solution

Step 1/6

Critical section problems are often solved by preventing interrupts while shared variable is being modified.This process is simple to use in uniprocessor.But in case of multiprocessor, this is not feasible solution because disabling interrupts on a multiprocessor is most time consuming as the message is passed to all processors. Each time, a process needs to acquire a lock in the following way before entering into the critical section.

Step 2/6

```
do

{

acquire lcok

critical section

release lock

remainder section

}while(TRUE);
```

Step 3/6

This message passing technique delays getting entry into critical section. So, the system efficiency decreases. If the interrupts need to be updated by the system clock, then system clock is affected in updating the clock cycles by frequent interrupts being

occurred.

Step 4/6

Minimization of interrupts:

Special hardware instructions like TestAndSet() and swap() instructions can be used to test and modify the content of a word or to swap the contents of two words automatically. By using these instructions, the affect can be minimized.

Step 5/6

Following is the way to implement the TestAndSet() instruction

```
do

{

While(TestAndSet(&lock));

lock=FALSE;

}while(TRUE);
```

Step 6/6

Similarly ,the swap() instruction operates on the contents of two words where lock is initialized to FALSE and key is Boolean variable to each process.

```
do

{

key=TRUE;

while(key==TRUE)

Swap(&lock,&key);

//critical section

lock=FALSE;

}while(TRUE);
```

Chapter 5, Problem 2PE

(0)

Step-by-step solution

Show all steps

100% (4 ratings) for this solution

Step 1/4

Multiple Locking Mechanisms

Solaris, Linux and Windows XP all three operating systems implements multiple locking mechanism to protect the access of critical section code.

Consider a case given below:

• Processes perform many operations like read, write, update on the critical section. If OS doesn't use any locking mechanism, every process uses the critical section and change the code it will results inconsistency.

To avoid inconsistency, the above mentioned operating systems have multiple locking mechanisms. The multiple locking mechanisms implementation is shown below:

• A process p1 acquires a lock to critical section for its execution.

• At the same time, a process p2 wants to use the critical section but, it will be locked by some other process.

• P2 must wait until the lock releases by p1. Here, no chance to occurrence of inconsistency.

Step 2/4

Windows:

• Windows uses *mutex lock*, when the number of threads waits in a queue. Thread acquires mutex lock on a nonsignaled dispatcher and releases the lock on a signed dispatcher.

• Critical-section uses *spin lock* when a thread waits for another thread to release the lock.

Step 3/4

Linux:

• Linux uses *mutex_lock ()* and *mutex_unlock ()* function to acquire and release the locks.

• Linux uses *Spin locks* only for short duration processes.

• When a process requires a lock for long duration, it must use *semaphores*.

Step 4/4

Solaris:

• Solaris uses *adaptive mutex lock* to protect the access of each data item in the critical section. If a data is locked, adaptive mutex lock follows two things:

o If lock is held by a thread currently running on the CPU, the thread spins until the lock becomes available.

o If the thread holding the lock is not in the running state, thread sleeps until the release of lock.

• Solaris uses *mutex lock* when a thread contains less than 500 instructions.

• Solaries uses *condition variables* and *semaphores* for longer code segments.

Chapter 5, Problem 3PE

(8)

Step-by-step solution

Show all steps

96% (23 ratings) for this solution

Step 1/1

Processes waiting to enter the critical section use the CPU to keep checking if they can enter their critical section. The act of using the CPU to repeatedly check for entry to the critical section is called *busy waiting.*

Busy waiting can be overcome by a wait ( ) and signal ( ) semaphore operation. When a process executes the wait ( ) operation and find that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, process can block itself.

To avoid busy waiting, we'll create resources that enforce mutual exclusion and make them accessible by through the operating system. Processes waiting to enter a critical section will be put in the blocked state, just like they were waiting for I/O, so that other processes can continue doing useful work.

Chapter 5, Problem 4PE

(4)

Step-by-step solution

Show all steps

100% (8 ratings) for this solution

Step 1/1

Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is called spin lock because the process "spins" while waiting for the lock. When locks are expected to be held for short times, spin locks are useful, they are employed on multiprocessor system where one thread can "spin" on one process while another thread performs its critical section on another processor.

Chapter 5, Problem 5PE

(5)

Step-by-step solution

Show all steps

100% (11 ratings) for this solution

Step 1/2

A semaphore S is an integer value indicates the number of resources available.

• The wait () operation decreases the value of semaphore S *until the value of* S *become 1. When a process wants a resource, then it performs the wait () operation to increment semaphore* S *value.*

• *The signal () operation increases the value of* semaphore S*. When the process does not need the resource anymore, then it performs the signal () operation to increment semaphore* S *value.*

Step 2/2

A process must do both the operations concurrently to achieve the mutual exclusion.

If the operations wait () and signal () do not execute automatically, then mutual exclusion may be violated.

The condition where the mutual exclusion may be violated when the wait () and signal () semaphore operation are not executed automatically is shown below:

If the number of available resources S is 1, and the two processes P_1 and P_2 executes only wait () operation concurrently, then the mutual exclusion may be violated.

When P_1 and P_2 determine that value of S is 1 and both performs wait () operation, then P_1 decrements S by 1 and enters critical section and P_2 decrements S by 1 and enters the critical section.

But, the processes are not performing the signal () operation after the wait () operation. This condition leads to the violation of the mutual exclusion.

The timeline of the processes P_1 and P_2 is as shown below:

T_0: P_1 determines that value of S = 1.

T_1: P_2 determines that value of S = 1.

T_2: P_1 decrements S by 1 and enters the critical section.

T_3: P_2 decrements S by 1 and enters the critical section.

Chapter 5, Problem 6PE

(5)

Step-by-step solution

Show all steps

80% (5 ratings) for this solution

Step 1/2

A variable that is used to control multiple processes under parallel programming is often known as a **semaphore**.

Operating system defines two types of semaphores; one is a counting semaphore and another one is binary semaphore.

A **binary semaphore** may only range between 0 and 1 value, and hence its functional operations are similar to that of a mutex lock.

Step 2/2

When it comes to represent the implementation of mutual exclusion with the use of a single binary semaphore, for '*n*' number of processes, user can execute it as given below:

Let the '*n*' number of processes be described as "*Pi*". Now for the mutual exclusion between these *n* processes we take a semaphore mutex which is initialized to '1'.

Hence, each *Pi* process would undergo the below condition:

```
do

{

wait (mutex);

signal (mutex);

}

while (true);
```

In this piece of code we can see a "do-while" condition. Here, the wait() function lies under critical section of mutual exclusion; whereas signal() function lies under the remainder section of same.

Chapter 5, Problem 7E

(13)

Step-by-step solution

Show all steps

100% (28 ratings) for this solution

Step 1/4

A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition.

Step 2/4

Let us consider a shared bank account existing between a husband and wife and they have shared balance amount sharedBalance = 1000

The two transactions will be started by both husband and wife simultaneously

Husband	Wife
withdraw()	deposit()
husbandT1: withdraw 500 from sharedAccount	wifeT1: deposit 500 into shared account
sharedBalance = sharedBalance – amountWithdrawn from sharedAccount	SharedBalance = sharedBalance+amountDeposit
sharedBalance = 1000 – 500 = 500	sharedBalance = 1000 + 500 = 1500

Step 3/4

Here sharedBalane is a resource data to be shared by both accounts simultaneously.

Due to the access of the shared data at the same time, inconsistency in the balance occurred. This is called updation anomoly.

Step 4/4

Prevent from race condition

Before acquiring the shared data we need to acquire a lock on shared data. After completion of withdrawal from shared balance, then release the lock. Then the next sequence of instruction like depositing into account will be started.

Husband	Wife

withdraw()
Lock():
husbandT1: withdraw 500
From shared Account
shared Balance =
sharedBalance – amountWithdrawn
sharedBalance =1000-500
=500
release():
	deposit()
	lock()
	wifeT1:
	deposit 500 into sharedAccount
	sharedBalance = sharedBalance + amountDeposit
	sharedBalance = 500+50 = 1000
	release():

Chapter 5, Problem 8E

(8)

Step-by-step solution

Show all steps

100% (8 ratings) for this solution

Step 1/4

Refer to the algorithm of the process P_i shown in Figure 5.21 in Chapter 5 from the textbook.

Consider the two processes P_0 and P_1 share the following variables of the algorithm:

boolean flag[2];

int turn;

The three requirements to solve the critical section problem are,

1. Mutual exclusion

2. Progress

3. Bounded waiting

Step 2/4

1. Mutual exclusion:

This requirement of the critical section states that only one process P_i can execute its critical section at a time. If one process is executing its critical section, then the remaining processes must not execute their critical sections at the same time.

Refer the structure of process P_i shown in Figure 5.21 in Chapter 5 from the textbook.

The other process is $P_j(\text{j} == 1 \text{ or } 0)$.

As per the algorithm, the process can enter the critical section only if its $flag$ variable is true. The code in the while loop makes the $flag$ variable of only one process as true at a time.

The process P_i enters its critical section only if $flag[j] = false$ or $turn = i$.

If $flag[j] = true$, the while loop repeats till the $flag[j]$ becomes $false$ and makes the variable $turn$ to j. Then process P_j executes its critical section. After completion of the execution of process P_j , it makes $flag[i] = true$ and allows process P_i to execute its critical section. That mean, the process can only execute its critical section when other process updates the $flag$ and $turn$ variables.

Hence, the algorithm allows only one process to executes its critical section at a time.

Therefore, the Dekker's algorithm satisfies the Mutual exclusion property.

Step 3/4

2. Progress:

When more than one process wants to enter the critical section at the same time, then a deadlock will occur. To avoid the deadlock only one process must execute its critical section at a time.

Therefore, when the processes want to enter the critical section at the same time, the processes which are not executing their remainder section can select which process may use its critical section.

If the processes P_i and P_j want to execute their critical section at same time, the process in the remainder section decides which process can enter into the critical section.

Suppose $flag[i] = true$, then P_i executes its critical section. After completion of the P_i execution, the code in the remainder section makes the variable $turn = j$ and $flag[i] = false$. Now the turn is for process P_j. Then, process P_j executes its critical section.

Therefore, the Dekker's algorithm satisfies the Progress property.

Step 4/4

2. Bounded waiting:

The amount of time a process waits after it made a request and before the request is accepted to enter its critical section. The process must not wait for a long time to get the resources. There is a limit on the amount of time the process waits to enter into its critical section.

In the algorithm, one process updates the value of the variable $turn$ of the other process. If the process P_i wants to enter its critical section, then $flag[i] = true$. After completion of the P_i execution, it sets the $turn$ variable to j and allows the process P_j to executes its critical section. Hence, both processes need not wait for indefinite time.

Therefore, the Dekker's algorithm satisfies the Bounding waiting property.

Chapter 5, Problem 9E

(8)

Step-by-step solution

Show all steps

100% (5 ratings) for this solution

Step 1/4

The three requirements for the critical-section problem are as follows:

- Mutual exclusion

- Progress requirement

- Bounded-waiting

The algorithm satisfies the three requirements/conditions for the critical-section problem.

Step 2/4

Mutual exclusion:

• A process enters into the critical section only when flag variable of any other process is not set to in_cs.

• The process sets its flag to in_cs before entering the critical section.

• Then the process checks if there is any other process whose flag variable is in_cs.

• Thus, it can be ensured that no two processes will be in the critical section.

Step 3/4

Progress requirement:

• Assume that more than one processes set their flag variable to in_cs at same time.

• After each sets its flag, they check if there is any other process whose flag variable is in_cs.

• As each realize that there is another competing processes whose flag variable is in_cs, each process move to the next iteration of the while() loop.

• Then they set their flag variable to in_want.

• Among all the processes, the process whose index value is near to turn variable will set its flag variable to in_cs.

• At each iteration, the index of the processes who set their flag variable to in_cs move closer to turn.

• Thus, it is ensured that only one process can sets its flag variable to in_cs.

Step 4/4

Bounded-waiting:

• Suppose a process k wants to enter the critical section. Then its flag variable to in_cs.

• If the index of any process that wants to enter the critical section is not in between turn and k, then it cannot enter the critical section.

• If the index of any process is between turn and k, then they can enter the critical section and the turn value will come closer to k.

- So, turn will become k at some point and can enter the critical section.

- Thus, the condition of bounded-waiting is satisfied.

Chapter 5, Problem 10E

(6)

Step-by-step solution

Show all steps

100% (11 ratings) for this solution

Step 1/1

If a user-level program is given the ability to disable interrupts, then it can disable the timer interrupt and prevent context switching from taking place ,there by allowing it to use the processor without letting other processes to execute.

Chapter 5, Problem 11E

(4)

Step-by-step solution

Show all steps

100% (6 ratings) for this solution

Step 1/1

Disabling interrupts on every processor can be a difficult task and further more can seriously diminish performance. Disabling interrupts on a multiprocessor can be time consuming, as the message is passed to all the processors. This message passing delays entry into each critical section and system efficiency decreases. Also, consider the effect on a system's clock, if the clock is kept updated by interrupts.

Chapter 5, Problem 12E

(4)

Step-by-step solution

Show all steps

100% (2 ratings) for this solution

Step 1/1

A lock that makes threads of a process enter into wait condition, while searching for availability of desired lock is known as a **spinlock**.

Process cannot hold a spinlock while attempting to acquire a semaphore because of the following reasons:

- Semaphores are sleeping lock in Linux. If a task wants to retrieve the semaphore which is already held then the task should be put on the waiting queue to sleep.

- Spinlocks are just another form of interrupt disabler and hence they cannot go into a sleeping state.

- As the wait duration is usually very small, no thread goes into sleep mode. Rather the threads remain active but still do not perform any useful work.

- As the wait time is very small, no process can sleep while holding a spinlock. The process will keep attempting until it get the lock.

- Trying to acquire a semaphore variable may take longer time than the wait duration of a spinlock. Hence, one cannot use a semaphore while being in a spinlock condition.

- Sleeping while holding spinlocks may force rest of the threads in the process to enter infinite loops until and unless the thread becomes active again.

Chapter 5, Problem 13E

(0)

Step-by-step solution

Show all steps

100% (2 ratings) for this solution

Step 1/2

Race condition:

The condition occurs inside critical section, critical section means when multiple threads access a shared variable at the same time.

Kernel Data Structure:

- Scheduling queue

- Kernel process table

- Process id management system (pidms)

Step 2/2

Scheduling Queue:

The scheduling queues in the kernel data structure, receive the process. One process in queue has been waiting for the I/O resources, and the resource is free now. The resource is allocated to the process. Second process waiting for the same I/O resource is still pending in the queue. These two processes in the queue are in a race with each other for the allocation of the I/O resources to move to the runnable queue for execution. Hence processes create a race condition in the runnable queue.

Kernel Process Table:

The kernel table also triggers a race for two processes created at the same time for allocation of space in the kernel table.

Process id management system: The pidms permits two processes to be created at the same time, accessing same resources, triggering a race between the two for getting the unique process id (pid).

Chapter 5, Problem 14E

(6)

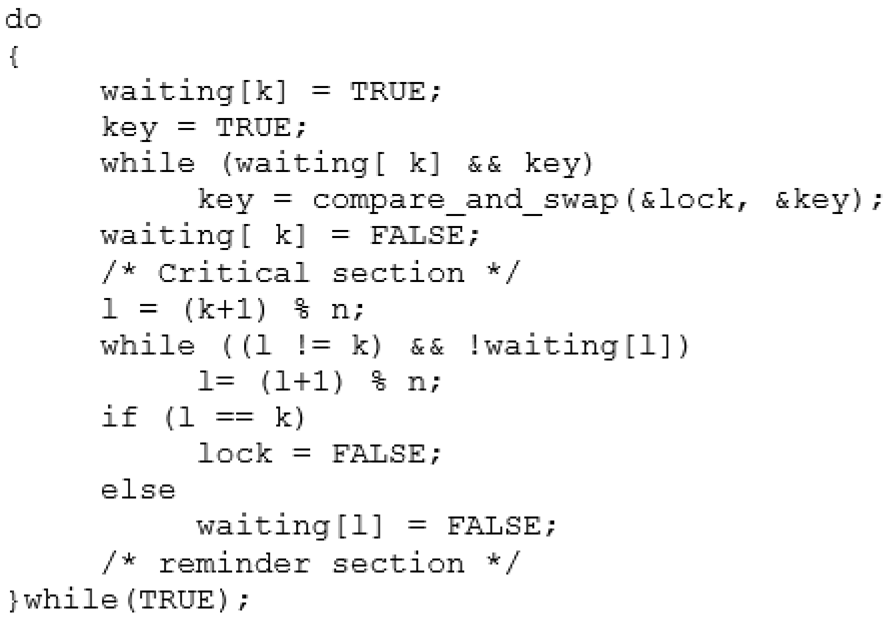
Step-by-step solution

Show all steps

Step 1/2

Consider the following piece of code to illustrate how a compare_and_swap () instruction helps in achieving mutual exclusion under bounded-waiting policies.

Bounded-waiting mutual exclusion with compare_and_swap():



```
do
{
     waiting[k] = TRUE;
     key = TRUE;
     while (waiting[ k] && key)
          key = compare_and_swap(&lock, &key);
     waiting[ k] = FALSE;
     /* Critical section */
     l = (k+1) % n;
     while ((l != k) && !waiting[l])
          l= (l+1) % n;
     if (l == k)
          lock = FALSE;
     else
          waiting[l] = FALSE;
     /* reminder section */
}while(TRUE);
```

Step 2/2

EXPLANATION:

Here, variable key is local to “P_k” process. The entire code goes on with two main shared variables, both taken as Booleans:

- lock

- waiting [n]

Both the Boolean variables are initialized to FALSE, for all ongoing processes that is "P_k"; where k = 0 to n-1 values.

According to the above code:

If lock = FALSE **,** no upcoming processes are in critical section. Whereas, if waiting[k] = FALSE is achieved P_k process not ready to enter into its critical section.

- According to Bounded Waiting, after a request is generated by a process to get into its critical section and before granting the request, there exist the limit on number of times other process are approved to get into its critical section.

- Bounded-waiting cannot be satisfied by the basic synchronization instructions available for hardware components.

- But the instructions available can be modified into a new program which satisfies the condition of bounded-waiting.

Chapter 5, Problem 15E

(8)

Step-by-step solution

Show all steps

100% (2 ratings) for this solution

Step 1/5

Some special hardware instructions are given by many machines that are used to test and swap the content of words automatically.

The functions test_and_set() and compare_and_swap() are the two hardware instruction used for the same.

Mutex structure is defined as:

typedef struct

{

int available**;**

}lock**;**

Step 2/5

Implementation using test_and_set() :

- In the lock struct, when the variable “available == 1”, it means that lock is not available. However when “available == 0” means that lock is released and available.

- The function test_and_set() provides mutual exclusion and is implemented automatically. Definition of the instruction is provided below as:

```
boolean test_and_set(boolean *target)

{

boolean rv =*target;

*target = true;

return rv;

}
```

- Initialization of the lock is done in init(lock *mutex) function in which the variable available is initialized as 0 which means it is available and can be locked by some thread.

- Then, in the function acquire(lock *mutex), atomic hardware instruction test_and_set() is called when it is locked by some thread to do thread’s processing.

- The function release(lock * mutex) clears the value of the variable available to 0.

Step 3/5

The definitions of functions are as follows:

```
void init(lock *mutex)

{

//lock is available at value 0, 1 otherwise.

mutex->available=0;

}

void acquire(lock * mutex)

{

// keep on testing the mutex until it is

// acquired

while(test_and_set(&mutex->available, 1) == 1);
}

void release(lock * mutex)

{

// release the mutex by clearing the

// available to 0

mutex->available=0;

}
```

Step 4/5

Implementation using compare_and_swap():

- The function compare_and_swap() has three parameters as opposed to the function test_and_set() which has only one parameter.

- This instruction returns the original value of the first parameter.

- The function compare_and_swap() is defined as below:

```
boolean compare_and_swap(int *value, int expected,

int new_value)

{

int temp = *value;

if(value==expected)

*value = new_value;

return temp;

}
```

- In the function compare_and_swap() also, when the variable “available == 1”, it means that lock is not available. However, when “available == 0”, it means that lock is released and available.

- The value is updated only if first parameter in compare_and_swap() function is pointing to a value which is same as second parameter, expected, in the compare_and_swap() function.

- Similar to test_and_set()function, release(lock * mutex) function clears the value of variable available to 0.

Step 5/5

The definitions of fun ctions are as follows:

```
void init(lock *mutex)
{
//lock is available at value 0, 1 otherwise.
mutex->available=0;
}
void acquire(lock * mutex)
{
// keep on comparing until the condition
// is met
while(compare_and_swap(
&mutex->available, 0, 1) == 1);
}
void release(lock * mutex)
{
// set the value of muted to 0
&mutex->available=0;
}
```

Chapter 5, Problem 16E

(2)

Step-by-step solution

Show all steps

Step 1/2

If a process is executing in its critical section, at same time process wants to get into its critical section, then it goes into the continuous loop in the entry section. This condition is known as busy waiting.

Step 2/2

The changes necessary to solve busy waiting:

• Connected with each mutex lock might a chance to be a queue of waiting processes.

• When A procedure determines those lock is unavailable, they need aid put under those queue.

• At A transform discharges the lock, it removes Also awakens the main procedure from the rundown for sitting tight procedures.

Chapter 5, Problem 17E

(2)

Step-by-step solution

Show all steps

100% (8 ratings) for this solution

Step 1/4

A lock that makes threads of a process enter into wait condition, while searching for availability of desired lock is known as a **spinlock**.

Mutual exclusion (mutex): When one process lies in its critical section, and the other process wanted to enter its critical section, then it cannot enter.

Step 2/4

In first condition, the lock is to be held for a short duration and hence “Spinlock” will be the best option for the processes. As the spinlock mechanism locks the threads for shortest possible time and do not go into long looped mechanisms.

Step 3/4

In second condition, the lock is to be held for a long duration and hence in this scenario “Mutex Lock” will be the best choice. As Mutex locks may sometimes undergo long time duration and are far apart from spinlock’s short duration holds.

Step 4/4

In third condition, the thread may be put to sleep while holding the lock; and hence once again a “Mutex Lock” would be the better one among two available lock choices. The main reason behind this is spinlocks do not allow threads to go in sleep mode, while holding the lock.

Chapter 5, Problem 18E

(0)

Step-by-step solution

Show all steps

100% (1 rating) for this solution

Step 1/5

A lock that makes threads of a process enter into wait condition, while searching for availability of desired lock is known as a **spinlock**.

Mutual exclusion (mutex): When any process is already present in its critical section and the other process wants to enter its critical section but cannot enter; then the mutual exclusion is achieved.

Step 2/5

- A lock can easily be replaced by another type of lock, just by changing the lock variable.

- Spinlocks can be used for short periods only; they typically avoid or do not allow the threads to sleep.

- Whereas, a mutex lock is totally opposite of spinlock. It allows long spinning loops, which means threads are allowed to sleep in the wait queue until the lock gets released by the current thread.

Step 3/5

Given below is a piece of code that illustrates the normal switching among spinlock and mutex lock while undergoing a program:

```
while(Lock_Obj.dest != Lock_Obj.compare)

{

if(HasThreasholdReached())

{

if(n_iterations + YIELD_ITERATION >=

MAX_SLEEP_ITERATION)

pthread_mutex_init(&mutex, NULL);

Sleep(0);

}
else

{

pthread_spin_init(&spinlock, 0);

SwitchToThread();

}

}
```

Step 4/5

• Assuming the time taken for context switching will be *T*. The code has few variables that work under “if-else” condition.

• If the total time taken to make a switch or locate available lock takes longer than the **“max_sleep_iteration”** time, then the thread is allocated to a mutex lock. And the reverse is allocated with the spinlock.

• The upper bound for holding a spinlock must be less than double of T = $(2\times T)$ because if a spinlock is waited for a longer period of time more than T, then it would be faster to put the thread to sleep which requires one context switch. Then, to awaken it will require a second context switch.

Step 5/5

Hence, the upper bound for holding a spin lock should be $(2\times T)$.

Chapter 5, Problem 19E

(5)

Step-by-step solution

Show all steps

100% (4 ratings) for this solution

Step 1/4

Mutex lock:

• Locks are basically used to destroy interrupts in multi-threaded systems. Locks prevent threads from entering a deadlock state. In the given piece of code, locks are used to overkill the interrupts.

• Before going beyond, how locks are created or occupied should be known. A lock usually requires a system call and make the process in sleep state if the lock is unavailable.

- The thread then raises a request to acquire a lock; in case the lock is unavailable, and the process is looped up with another context switch consequently.

Step 2/4

Atomic update:

- In the given piece of code, the atomic integer updates the variable allocated for ‘hits’ to ensure the system does not exceed the race condition on hits.

- The condition of achieving no race condition on hits can only be achieved if the process call is not interrupted by the Kernel.

Step 3/4

In the first approach, the race condition being applied on hits. The race condition is successfully shown by “hits++;” line. Hence, this is not the efficient approach among the two.

Step 4/4

Atomic integers are efficient than the Mutex locks, because no race condition is occurring, and the operations don’t require the overhead of locks.

Atomic variable is an integer variable like a counter variable, and it needs to be updated frequently. Since, atomic update doesn’t require the lock, it is more efficient.

Chapter 5, Problem 20E

(10)

Step-by-step solution

Show all steps

100% (5 ratings) for this solution

Step 1/5

R**ace condition** is the scenario where many processes retrieve and modify the same data simultaneously and outcome of the program is based on the sequence in which they are accessed.

Step 2/5

a.

- The given code illustrates the presence of one race condition; that is on the variable number_of_processes.

- The function fork() checks and increments the variable number_of_processes while; the exit()decrements the variable number_of_processes.

- Thus, both update the contents of the variable number_of_processes.

- The variable can be modified by the processes of fork() and exit() simultaneously and will result in inconsistencies due to the race condition occurring among various processes.

- Therefore, the final value of number_of_processes depends upon the process execution order.

Step 3/5

b.

- To prevent the race condition, the mutex lock with acquire() and release() operations can be placed at the beginning and end of a critical section in both the fork() and exit() functions.

- In the fork() function’s allocate_process(),the acquire()should be placed before the test condition while, release() should be placed after incrementing the variable number_of_processes.

- In the exit() function’s release_process()the acquire()should be placed before the decrement of the variable number_of_processes while; release() should be placed after the decrement of the variable number_of_processes.

- The updated code snippet is:

```
int allocate_process()
{
int new_pid;
acquire();
if(number_of_processes==MAX_PROCESSES)
return -1;
else{
++ number_of_processes;
release();
return new_pid;
}
void release_process(){
```

acquire();

-- number_of_processes;

release();

}

Step 4/5

c.

- All the operations on atomic_t variable are atomic i.e. they execute within one clock cycle.

- If the variable number_of_processes is made atomic_t, the increment and decrement operations on the variable will become atomic but the race condition can still occur in the allocate_process() function.

- This is because the value of number_of_processes is initially tested in the if-condition of the allocate_process() function.

- Consider the case where the value of the variable number_of_processes is 254 at the time of the test. But due to the race condition, another process makes it 255.

- Thus, when the process executes, the value of variable becomes 256 which is inconsistent.

Step 5/5

Therefore, replacing the integer variable with atomic_t will not prevent the race condition.

Chapter 5, Problem 21E

(2)

Step-by-step solution

Show all steps

75% (4 ratings) for this solution

Step 1/3

Solution:

Semaphores are defined as inter-pr ocess communication.

- It is used to control and monitor the open socket connections.

- It allows the process to create the open socket connections that communicate with the other processes.

- It describes the open connection’s existence and properties among the sockets.

- The server deals with each connection via semaphore communication, it instructs new connections are done until the limit is reached.

- It makes the individual processes to exit to make a way to new connections.

A semaphore is initialized with open socket connections, the number of allowable connections is made.

Here,

- the acquire() function is called if the connection is accepted.

- the release() function is called if the connection is released.

- The repeated number of calls to acquire() will block the existing connection, where the number of allowable socket connections to the system is reached.

- The release() method is invoked where the existing connection is terminated.

Step 2/3

Process:

Initialize the semaphore so its value is N.

- When a connection comes in, wait on the semaphore.

- When a connection is released, signal the semaphore.

Semaphore operations are defined as

```
S=N;
acquire (s)
{
val--;
if(val<0)
{
//add the process to the list
block;
}
}

release(S)

{

val++;

if(val<=0)

{

//remove the process p from the list

wake_up(p);

}

}
```

Step 3/3

The main disadvantage of mutual exclusion is that they require all process to be busy waiting.

• Busy waiting degrades the CPU cycles, it leads other process does not use it.

• A semaphore that gives the result “spin” is called spin lock while the process will be waiting for a lock situation.

• The main advantage of the spin lock is, when the process must wait on a lock situation and the context switch takes much time.

• Therefore, no context switch is required

• Hence spin lock is useful for short locks.

• They are often used in multi-processor system where one thread spin while another thread performs its critical section.

The process instead of busy waiting, it will block itself, it places the process into the waiting queue.

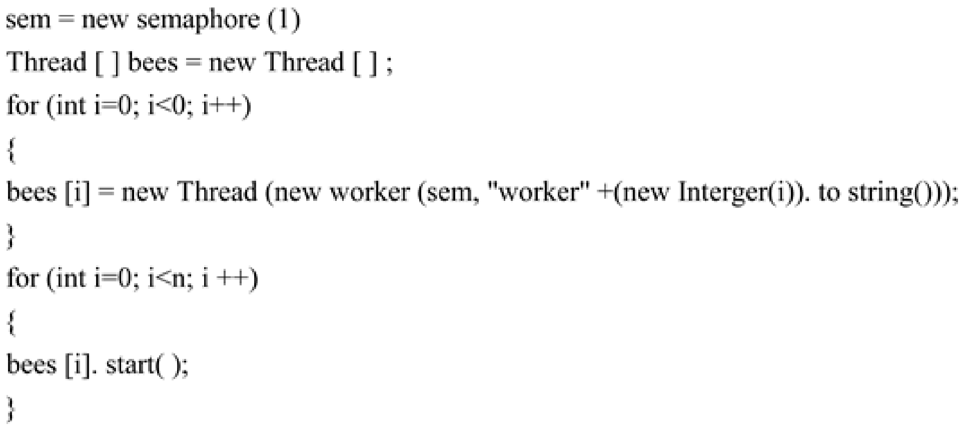
• Then, the process is switched to the waiting queue state with the association of semaphore.

• Then the CPU scheduler schedules another process to execute where the control is transferred to it.

• It restarts the process and transferred to the ready queue for the process execution.

• Depends on the CPU scheduling algorithm, the CPU switches between the running process to the ready process possibly.

Semaphore



```
sem = new semaphore (1)
Thread [ ] bees = new Thread [ ] ;
for (int i=0; i<0; i++)
{
bees [i] = new Thread (new worker (sem, "worker" +(new Interger(i)). to string()));
}
for (int i=0; i<n; i ++)
{
bees [i]. start( );
}
```

Chapter 5, Problem 22E

(1)

Step-by-step solution

Show all steps

Step 1/1

Benefits of slim- reader and writer (SRW) locks:

1.SRM locks activate the threads of single process to access a shared resource.SRM locks provide two modes exclusively for reading and writing the data .

2.Shared Mode which grants shared data read only to multiple threads and multiple threads can access the shared data concurrecntly.so,the performance can be increased.

3.Exclusive Mode, when a lock is acquired by a thread in write mode,then the lock will not be release until the total operation will complete.And no thread will access the same lock Until the writer releases the lock.

4.Once a SRM acquires a lock in shared mode then the there is no possibility to upgrade to exclusive lock state and similarly a lock acquired in exclusive mode donot down grade to shared mode.

5.SRM is light weigth and small i.e the size of the pointer (32 or 64 bits) and there are no associated kernel objects for waiting so that it requires minimal resources.

Chapter 5, Problem 23E

(5)

Step-by-step solution

Show all steps

100% (4 ratings) for this solution

Step 1/4

wait():

The wait() system call used to wait for another process to complete its execution in the operating system and it may wait until any of its child processes to exit in the operating system.

Step 2/4

signal():

It is a system call or a software interrupt generated by the operating system and sent to the process with some number. That number is called signal ID or signum. The signal is generated when the process is aborted, quit, trace, or hang while the process is executing.

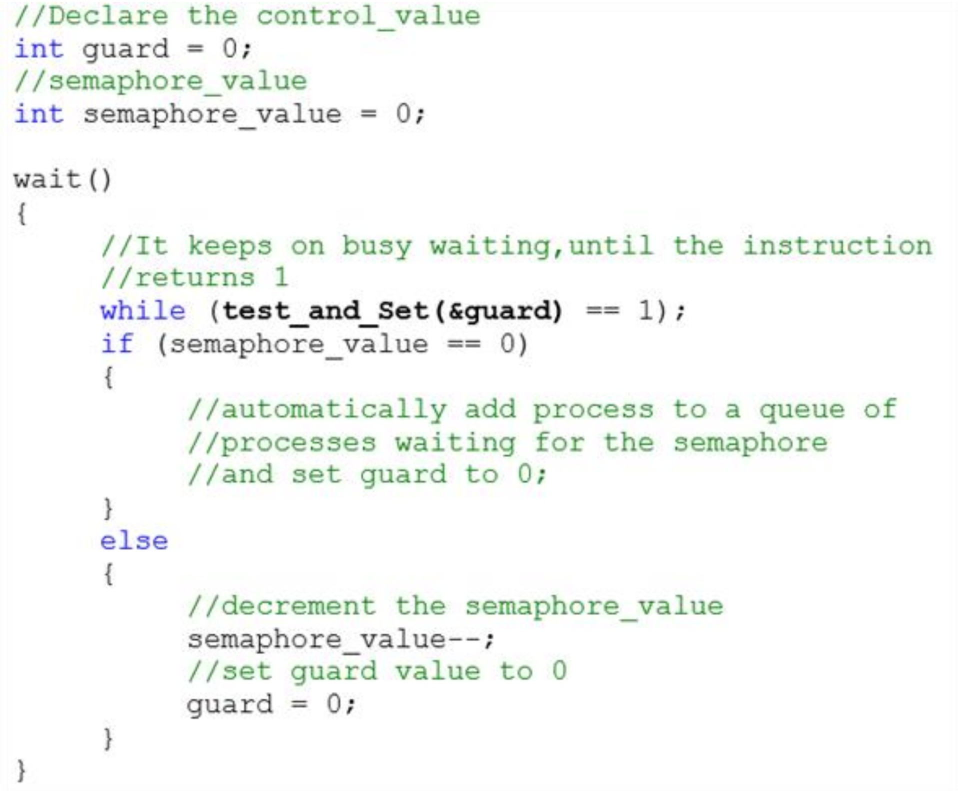
Step 3/4

test_and_set():

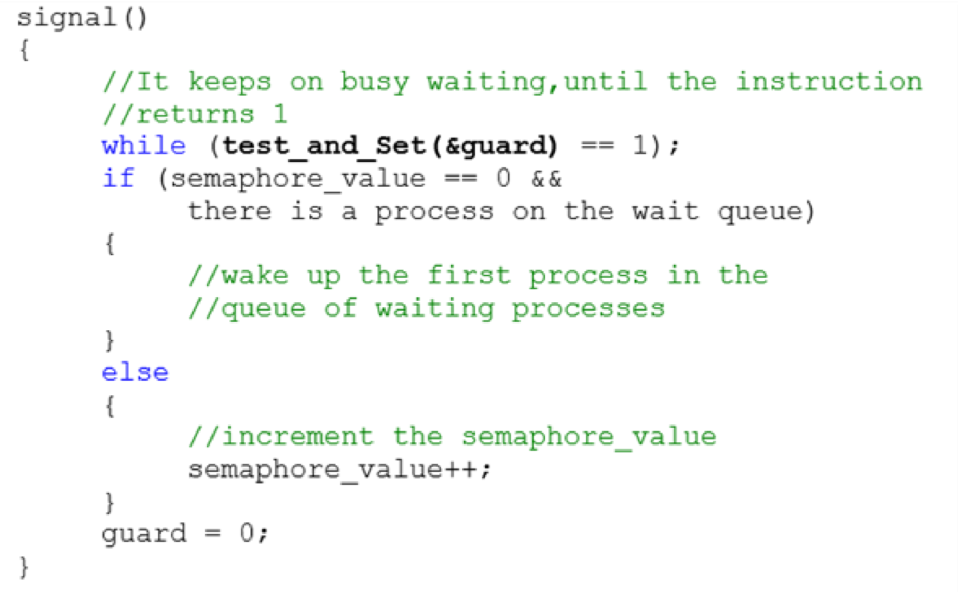
test_and_set() function performs two operations individually and cannot be interrupted in the middle of the process execution. This function used for handling the synchronization problem of two processes. It takes a shared lock variable as an input parameter which is either 0 (unlock) or 1(lock).

Step 4/4

Implementation of wait() and signal() are as follows:



```
//Declare the control_value
int guard = 0;
//semaphore_value
int semaphore_value = 0;

wait()
{
      //It keeps on busy waiting,until the instruction
      //returns 1
      while (test_and_Set(&guard) == 1);
      if (semaphore_value == 0)
      {
            //automatically add process to a queue of
            //processes waiting for the semaphore
            //and set guard to 0;
      }
      else
      {
            //decrement the semaphore_value
            semaphore_value--;
            //set guard value to 0
            guard = 0;
      }
}
```



```
signal()
{
      //It keeps on busy waiting,until the instruction
      //returns 1
      while (test_and_Set(&guard) == 1);
      if (semaphore_value == 0 &&
            there is a process on the wait queue)
      {
            //wake up the first process in the
            //queue of waiting processes
      }
      else
      {
            //increment the semaphore_value
            semaphore_value++;
      }
      guard = 0;
}
```

Chapter 5, Problem 25E

(4)

Step-by-step solution

Show all steps

100% (1 rating) for this solution

Step 1/2

Implementation of semaphore using monitor code:

```
// Semaphore of type monitor
monitor semaphore
{
// Variable declarations
int n = 0;
condition con;
// Function definition
semaphore increment()
{
// Increment the value
n++;
// Resume the suspended process
con.signal();
}
// Function definition
semaphore decrement()
{
// Condition to suspend the process
while (n == 0)
// Process is suspended
con.wait();
// Decrement the value
n--;
}
}
```

Step 2/2

Explanation:

- The variable “con” of condition is represented by threads waiting in a queue for variable “con”.

- A semaphore is coupled with each thread during its entry into the queue.

- In the semaphore increment operation, signal() operation is performed by a thread on a condition variable “con” to wake up the suspended process.

- In the semaphore decrement, wait() operation is performed by a thread on a condition variable “con”, which creates new semaphore.

o The value of new semaphore is initialized as 0.

o Then, the semaphore is decremented to block the new semaphore.

Chapter 5, Problem 27E

(3)

Step-by-step solution

Show all steps

Step 1/2

The explanation for the given statement is as follows:

- The answer to the bounded buffer question provided above copies the value obtained in the monitors’ local buffer and replicates it from the monitors’ local buffer to the consumer.

- The copy tasks can be costly if one were consuming huge amount of memory on behalf of every buffer.

- The augmented outlay of copy process signifies that the monitor is occupied for a elongated time interval while a process is in the produce or consumes task.

- This lessens the systems total output.

- The above setback can be relieved by storing pointers to buffer regions within the monitor as opposed to storing the buffer regions themselves.

- Therefore, one can revise the code provided to replicate the pointer to the buffer region in and out of the monitors’ state.

- The above task must be comparatively cheap and so the time interval that the monitor is being occupied is shorter, thereby improving the output of the monitor.

Step 2/2

```
monitor ResourceAllocator
{
boolean busy;
condition x;
void acquire(int time)
{if(busy)
x.wait(time);
busy=TRUE;
}
void release()
{
busy=FALSE;
x.signal();
}
initialization_code()
{
busy=FALSE;
}
}
```

Chapter 5, Problem 28E

(0)

Step-by-step solution

Show all steps

100% (4 ratings) for this solution

Step 1/1

Throughput in the readers-writers problem is increased by favoring multiple readers as opposed to allowing a single writer to exclusively access the shared values. On the other hand, favoring readers could result in starvation for writers.

The starvation in the readers writers problem could be avoided by keeping timestamps associated with waiting processes. When a writer is finished with its task, it would wake up the process that has been waiting for the longest duration. When a reader arrives and notices that another reader is accessing the database, then it would enter the critical section only if there are no waiting writers. These restrictions would guarantee fairness.

Chapter 5, Problem 29E

(1)

Step-by-step solution

Show all steps

100% (4 ratings) for this solution

Step 1/1

In monitor x. signal ( ) operation resumes exactly one suspended process. If no process is suspended, then the signal ( ) operation has no effect;that is the state of x is the same as if the operation had never been executed contrast this operation with the signal ( ) operation associated with semaphores, which always affects the state of the semaphore.

Chapter 5, Problem 30E

(2)

Step-by-step solution

Show all steps

Step 1/1

Wait (mutex);

------

body of F

--- - - -

if (next – count >0)

signal (next);

else

signal (mutex);

Mutual exclusion within a monitor is ensured.

Operation x. signal ( ) can be implemented as

```
if (x – count >0)

{ next – count ++;

Signal (x – sem);

Wait (next);

Next _ count -- -- ;

}
```

When a thread T1 calls signal(), the control switches to another thread T2 that is waiting for the condition immediately. In Hoare's implementation, when T2 leaves the monitor or calls another wait(), the control should be switched back to T1, not to the other threads that are waiting to get into the monitor. Since T1 is still inside the monitor, although it is inactive now, it would be better to let T1 finish its work first than admit the others into the monitor. The next semaphore and the next_count variable are used to keep track of this situation.

If signal() is the last statement of every monitor procedure, we are ensured that T1 is no longer in the monitor and therefore we do not need next semaphore and the next_count variable anymore.

Chapter 5, Problem 31E

(2)

Step-by-step solution

Show all steps

100% (1 rating) for this solution

Step 1/2

A monitor helps in achieving process synchronization. It allows threads to have both, mutual exclusion and the ability to wait for a certain condition to be true.

Consider a system with *n*-processes, having unique priority number. The goal is to write a monitor to allocate these processes to three identical printers:

- An array for the three printers in created.

- A procedure, *acquire()*, is created to check and allocate processes to the printer, using the priority numbers of the processes.

- Check if the printers are busy.

- If true, the process is suspended until one of the printer is free.

- Each process is assigned to one of the printers, by their priority number.

- Create a procedure, *release()*, to resume a process associated with the priority number.

- If a printer is free, the process is resumed and allocated to that printer.

Step 2/2

The implementation of the monitor is shown using the pseudocode:

```
type resource = monitor var P: array[0..2] of boolean; X: condition; procedure acquire
(id: integer, printer-id: integer);     begin         if P[0] and P[1] and P[2] then X.wait(id)
if not P[0] then printer-id := 0;                else if not P[1] then printer-id := 1;
else printer-id := 2;         P[printer-id]:=true;     end
```

```
procedure release (printer-id: integer)    begin        P[printer-id]:=false;        X.signal;
end
```

```
    begin        P[0] := P[1] := P[2] := false; end
```

Chapter 5, Problem 33E

(0)

Step-by-step solution

Show all steps

Step 1/3

The solution for the previous exercise will be correct under both conditions. It may suffer from a problem in which if a process wants to get a signal and if another process already in signal state, then the first process will prevail in waiting state.

Step 2/3

When the second process leaves the monitor from the signaling state, then the first process will progress forward to the signaling state.

The first process will wait to get into the signaling state else it will wait for another condition.

Step 3/3

A process will follow the while loop when it is in the wait state and it is replaced by ‘if’ condition if the process is ready to move in to the signaling state.

Therefore, a process will be in either of the following state.

- Signal and wait: First process either waits until second process leaves the monitor or waits for another condition.

- Signal and continue: Second process either waits until the first process leaves the monitor or waits for another condition.

Chapter 5, Problem 34E

(1)

Step-by-step solution

Show all steps

Step 1/1

We also require that writers have priority over readers: if a writer needs to write, current readers may complete reading, but new readers must be suspended until all writers are sleeping again.

We solve the problem by first introducing three shared integer variable: naw for the number f active writers (at most 1), nar for the number of active readers, and nww for the number of willing (nonsleeping) writers. The solution is now expressed by refining the regions CSR and CSW:

```
Monitor dp

{

Void write process ( )

{

do

{

wait (wrt)

- - - - -- - -

// writing is performed

- - - - - - - -

signal (wrt);

} while (true);

void read process ( )
```

```
{
do
{ wait (mutex);
read count ++;
of (read count = = 1
wait (wrt)
signal (mutex);
- - - - - -
// reading is performed
- - - - - - -
wait (mutex);
read count - - - -;
of (read count = = 0)
signal (wrt);
signal ( mutex);
} while (true);
}
```

Chapter 5, Problem 35E

(3)

Step-by-step solution

Show all steps

100% (4 ratings) for this solution

Step 1/2

Algorithm plan:

- Create a monitor to implement an alarm clock.
- Initialize a condition variable named as cond.
- Initialize an integer time variable for current time named as currentTime.

- Create a function with one integer type parameter that will accept the number of ticks.

- Declare an integer type variable named as alarms.

- Alarms = currentTime + ticks

- while (currentTime < alarms)

- Wait for the alarm signal.

- Create a function with name tick()

- Increment the value of current time

- currentTime = currentTime + 1;

- call the signal.

Step 2/2

Pseudocode:

```
monitor alarm

{

condition cond;

int currentTime = 0;

void timeDelay(int ticks)

{

int alarms;

alarms = currentTime + ticks

while (currentTime < alarms)

cond.wait(alarms);

cond.signal;

}

void tick()

{
```

```
currentTime = currentTime + 1;

cond.signal;

}

}
```

Chapter 5, Problem 37PP

(6)

Step-by-step solution

Show all steps

Step 1/3

a. Race condition:

The situation of numerous processes concurrently accessing and operating the same data and execution result is based on specific order in which access takes place is referred as race condition.

Here, the variable “available_resources” is data which is accessed and operated concurrently.

• The variable “available_resources” is data which is not synchronized between the threads. The value may not be up to date when one thread decreases this “available_resources” variable value.

Thus, the “available_resources” variable is act as data which involved in race condition.

Step 2/3

b. Locations in the code where the race condition occurs:

In the code, the race condition occurs in the two statements. There are,

• The code that decrement the available resources in decrease_count() method. The statement in this method is,

o available_resource -= count;

• The code that increment the available resources in decrease_count() method. The statement in this method is,

o available_resource += count;

Step 3/3

c. Fix the race condition using semaphore:

The semaphore is used to fix the race condition.

• Replace the increment and decrement operations by semaphore increment “signal()” method and semaphore decrement operations “wait()” method.

• Then, the code is given below:

```
#define MAX RESOURCES 5

int available_resources = MAX RESOURCES;
```

When a process wants to get a number of resources, it invokes the decrease_count() function using the wait() semaphore.

```
/* Decrease the available resources by count resources

and return 0, if sufficient resources available. */

int decrease_count(int count)

{

//Decrement the number of resource using wait()

wait(available_resources)

{

while (available_resources <= 0)

available_resources -= count;

}

return 0;

}
```

When a process wants to return a number of resources, it calls the increase_count() function using the signal() semaphore.

```
/* increase available_resources by count */

int increase_count(int count)

{
```

```
//Increment the number of resource using signal()
signal(available_resources)
{
// Increment the available resources value
available_resources += count;
}
// Return statement
return 0;
}
```

Chapter 5, Problem 38PP

(1)

Step-by-step solution

Show all steps

100% (1 rating) for this solution

Step 1/2

Monitor:

Semaphore provides the effective mechanism for process the synchronization but it is difficult to detect those errors. Instead use monitor to deal with such errors.

The syntax for monitor is given below:

```
monitor monitor_name
{
//Variable declarations
function function_name()
{
...
}
.
.
.

}
```

Step 2/2

Rewrite the code using monitor:

- The monitor is used to fix the race condition in the code (Note: Refer Question 5.37E in the book).

- Then, the code is given below:

```
monitor resources

{

// Variable declaration

int available_resources;

condition resource_available;
```

When a process wants to get a number of resources, it invokes the decrease_count() function using the wait() semaphore.

```
/* Decrease the available resources by count resources and return 0, if sufficient resources available. */

int decrease_count(int count)

{

while (available_resources < count)

resource_available.wait();

available_resources -= count;

}
```

When a process wants to return a number of resources, it calls the increase_count() function using the signal() semaphore.

```
/* increase available_resources by count */

int increase_count(int count)

{

// Increment the available resources count
available_resources += count;
resource_available.signal();
}
}
```

Chapter 6, Problem 1PE

(5)

Step-by-step solution

Show all steps

94% (15 ratings) for this solution

Step 1/1

2481-5-2E SA: 8683

SR: 4578

Given that there are *n* processes to be scheduled on one processor, and the first schedule can be done for any of the *n* processes, the total numbers of possible schedules in terms of ‘n’ are **factorial of n** => **n! (**n! = n×n –1×n–2×...×2×1).

For example, for 1 process, there is only one possible ordering: (1), for 2 processes, there are 2 possible orderings: (1, 2), (2, 1). For 3 processes, there are 6 possible orderings: (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1). For 4 processes, there are 24 possible orderings. So, for *n* processes, there are *n*! possible orderings.

Chapter 6, Problem 2PE

(5)

Step-by-step solution

Show all steps

100% (18 ratings) for this solution

Step 1/1

Scheduling

The difference between preemptive and non-preemptive scheduling is explained in the table below:

S. No	**Preemptive Scheduling**	**Non-Preemptive Scheduling**
1.	It interrupts a process in the middle of its execution.	Once a process enters the running state, it is **not** allowed

		to be **interrupted** till it has completed its service time.
2.	It **takes the CPU away** from the process being executed and allocates it to another process with a higher priority.	It makes sure that the process **keeps hold of the CPU** until it has completed its current CPU burst, thus completing its execution.
3.	It is **complex** to implement.	It is **easier** to implement.
4.	**High overhead** it produced due to constant switching.	**Lower overhead** due to simple implementation.
5.	The process switches from **running state** to **ready state** and then **waiting state** to **ready state**.	The process switches from **running state** to **waiting state** and then the process terminates.
6.	It is **costlier** to implement due to being complex.	It is **cheaper** to implement due to being simple.

Chapter 6, Problem 3PE

(12)

Step-by-step solution

Show all steps

100% (28 ratings) for this solution

Step 1/4

2481-5-17E SA: 8683

SR: 4578

a)

Given data:

Process Arrival Time Burst Time

P_1 0.0 8

P_2 0.4 4

P_3 1.0 1

Step 2/4

If the processes arrive in the order p_1, p2, p3, and are served in FCFS order, the result shown in Gantt chart, which is a bar chart that including start and finish times of each of the participating processes:

P_1	P_2	P_3

0 8 12 13

The Formula's are:

Turnaround time(P_i) = Completion Time(P_i) – Arrival Time(P_i)

n

Average Turnaround time = 1/n ∑ (C_i – A_i)

i=1

where C_i is Completion Time(P_i) and

A_i is Arrival Time (P_i)

Average Turnaround time = 1/3[(8-0) + (12-0.4) + (13-1)]

= 1/3[8 + 11.6 + 12]

= 1/3[31.6]

= 31.6/3

= 10.53

Therefore, the Average Turnaround time with FCFS algorithm =10.53

Step 3/4

b)

Using SJF scheduling, we would schedule the processes according to the following Gantt chart:

P_1	P_3	P_2

0 8 9 13

Average Turnaround time = 1/3[(8-0) + (13-0.4) + (9-1)]

= 1/3[8 + 12.6 + 8]

= 1/3[28.6]

= 28.6/3

= 9.53

Therefore, the Average Turnaround time with SJF algorithm=9.53

Step 4/4

c)

Using future –knowledge scheduling algorithm, we would schedule the processes according to the following Gantt chart:

	P_3	P_2	P_1

0 1 2 6 14

Average Turnaround time = 1/3[(14-0) + (6-0.4) + (2-1)]

= 1/3[14 + 5.6 + 1]

= 1/3[20.6]

= 20.6/3

= 6.87

Therefore, the Average Turnaround time with future-knowledge algorithm = 6.87

Chapter 6, Problem 4PE

(6)

Step-by-step solution

Show all steps

100% (14 ratings) for this solution

Step 1/1

2481-5-4E SA: 8683

SR: 4578

The advantage of having different time-quantum sizes at different levels of a multilevel queuing system is processes that need more frequent servicing for instance, interactive processes such as editors, can be in a queue with a small time quantum. Processes with no need for frequent servicing can be in a queue with a larger time quantum, requiring fewer context switches to complete the processing, making more efficient use of the computer.

Chapter 6, Problem 5PE

(5)

Step-by-step solution

Show all steps

100% (12 ratings) for this solution

Step 1/4

CPU Scheduling Algorithms

a. The relationship between priority and SJF or Shortest Job First is that the shortest job has the highest priority and is served first.

Step 2/4

b. The relationship shared between Multilevel Feedback Queues and FCFS or First Come First Serve is that the lowest level of Multilevel Feedback Queues is First Come First Serve.

Step 3/4

c. The relationship between priority and FCFS or First Come First Serve is that First Come First Serve assigns the highest priority to the job that has been in existence the longest time, thus serving the job that came first before the others.

Step 4/4

d. There is no relationship between the two scheduling algorithms RR or Round Robin and SJF or Shortest Job First. RR is a preemptive scheduling algorithm whereas SJF is a non-preemptive scheduling algorithm.

Chapter 6, Problem 6PE

(5)

Step-by-step solution

Show all steps

100% (7 ratings) for this solution

Step 1/1

2481-5-10E SA: 8683

SR: 4578

The I/O-bound processes would spend more time waiting for I/O than using the processor. So, they usually have used less CPU time in the recent past than other process. As soon as they are done with an I/O burst, they will get to execute on the processor. So this algorithm will favor I/O bound programs.

CPU-bound programs would spend more time on executing programs than I/O. However, since the algorithm takes into account only recent history, the CPU bound processes will not starve permanently. If a process has been prevented from using the processor for some time, it becomes one of the processes that have the least CPU time(=0) in the recent past and eventually it will get a chance to use CPU.

Chapter 6, Problem 7PE

(2)

Step-by-step solution

Show all steps

100% (4 ratings) for this solution

Step 1/1

Scheduling

The difference between **Process Contention Scope** or **PCS** scheduling and **System Contention Scope** or **SCS** scheduling is explained in the table given below:

S. No	**PCS Scheduling**	**SCS Scheduling**
1.	PCS scheduling is performed **within the process**. That is local to the process.	SCS scheduling is carried out **on the operating system** with kernel threads.

2.	PCS involves the thread library scheduling threads onto available Light Weight Processes.	In SCS scheduling, the system scheduler scheduling kernel threads to operate on single or multiple CPUs.
3.	On systems implementing **many-to-one** and **many-to-many** threads, PCS occurs, as competition occurs between threads that are part of the same process.	Systems employing **one-to-one** threads use only SCS.
4.	In Pthread scheduling, PTHREAD_SCOPE_PROCESS schedules threads by using PCS Scheduling, by scheduling the user threads onto attainable LWPs, using the many-to-many model.	In Pthread scheduling, PTHREAD_SCOPE_SYSTEM schedules threads by using SCS Scheduling, by connecting user threads to specific LWPs, effectively employing a one-to-one model.

Chapter 6, Problem 8PE

(0)

Step-by-step solution

Show all steps

100% (5 ratings) for this solution

Step 1/1

Threads

Y es it is **important** to **bind** a **real-time thread** to a light weight process **(LWP)**. By doing so, there will not be any latency while waiting for an accessible light weight process and the real-time user thread can be scheduled rapidly. If a real-time thread is not bound to an LWP, then the user thread may have to fight for an available LWP before being scheduled.

Chapter 6, Problem 9PE

(2)

Step-by-step solution

Show all steps

Step 1/3

Consider the data for the three processes and the recent CPU usage are as follows:

S. No	process	Recent CPU usage
1	P1	40
2	P2	18
3	P3	10

Formulae to compute priority is as follows:

$$\text{Priority} = \left(\frac{\text{recent CPU usage}}{2}\right) + \text{base}$$

Step 2/3

Compute the priorities for each process as follows:

- For process P1, substitute the recent CPU usage as 40 and base is 60:

$$\begin{aligned}\text{Pr iority} &= \frac{40}{2} + 60 \\ &= 20 + 60 \\ &= 80\end{aligned}$$

- For process P2, substitute the recent CPU usage as 18 and base is 60:

$$\begin{aligned}\text{Pr iority} &= \frac{18}{2} + 60 \\ &= 9 + 60 \\ &= 69\end{aligned}$$

- For process P3, substitute the recent CPU usage as 10 and base is 60:

$$\begin{aligned}\text{Pr iority} &= \frac{10}{2} + 60 \\ &= 5 + 60 \\ &= 65\end{aligned}$$

Step 3/3

From the above data the priorities for the three processes P1, P2, and P3 are 80, 69 and 65 respectively.

Here process P1 has the highest priority, P2 is the next priority and P3 has the lowest priority.

In traditional Unix scheduler the process priority is dynamic. The priorities are raised or lowered depends on the CPU interval time to execute a process by adjusting the priorities dynamically.

Hence, from the above information the first process is ready for execution which has highest priority, but the CPU time is 40(higher compared to other process). So, scheduler cannot wait longer time to wait for the first process execution, So, the priorities are lowered.

Chapter 6, Problem 10E

(3)

Step-by-step solution

Show all steps

94% (15 ratings) for this solution

Step 1/1

Importance of distinguish I/O-bound and CPU-bound programs

The scheduler need to distinguish I/O (Input/Output) – bound programs from CPU (Central Processing Unit) – bound programs for efficient use of computer resources. The scheduler efficiency depends on the optimum utilization of CPU. This can be achieved by observing the properties and distinguishing the programs.

I/O-bound programs utilize small amount of CPU resources. They have the property of performing small amount of computations before performing IO. This property makes I/O-bound programs not to use their entire CPU quantum.

CPU-bound programs utilize large amount of CPU resources. They have the property of performing computations without performing any blocking IO operations. This property makes CPU-bound programs to utilize their entire CPU quantum.

Based on these properties of I/O-bound and CPU-bound programs, scheduler should distinguish and give priorities. **Scheduler should give high priority to I/O-bound programs.** It should allow I/O-bound programs to execute before the CPU-bound programs. This scheduling will help in better utilization of computer resources.

Chapter 6, Problem 11E

(9)

Step-by-step solution

Show all steps

94% (15 ratings) for this solution

Step 1/3

CPU utilization and response time:

• Generally, CPU utilization should be maximized and response time should be minimized.

• CPU utilization can be maximized by minimizing the time spent in context switching, as the CPU is idle during context switching.

• The time taken from the submission of request to time the first response is produced is called the response time.

• When context switching is done rarely, then it would indeed result in decrease in the response time.

Step 2/3

b.

Average turnaround time and maximum waiting time:

• Generally, average turnaround time and average waiting time should be minimized.

• The interval from the time of submission of a process to the time of completion of a process is the turnaround time.

• Turnaround time is the time spent by a process waiting to get into memory, waiting in the ready queue, executing on CPU, and doing I/O.

• Waiting time is the time spent by a process waiting in the ready queue.

• The average turnaround time can be minimized by executing the jobs with shortest CPU bursts.

• If the jobs with shortest CPU bursts are executed first, then it will indeed increase the waiting time of the jobs with large CPU bursts and thus it will maximize the waiting time.

Step 3/3

c.

I/O device utilization and CPU utilization:

• Jobs can be categorized into CPU bound jobs and I/O bound jobs.

• I/O bound jobs spend most of the time using I/O devices and spend less time doing computation using CPU.

- CPU bound jobs spend most of the time doing computation using CPU and spend less time using I/O devices.

- CPU utilization can be maximized by executing CPU bound jobs that have large CPU bursts.

- I/O device utilization can be maximized by executing I/O bound jobs. As soon as the I/O device is free, it can be allocated to an I/O bound job.

Chapter 6, Problem 12E

(2)

Step-by-step solution

Show all steps

100% (6 ratings) for this solution

Step 1/2

Lottery ticket are based on the probabilistic scheduling algorithm in which each processor are provided with some number of lottery tickets and scheduler make decision of picking next process by drawing random ticket.

Step 2/2

Since every process are assigned a lottery ticket so if higher-priority processes are assigned more tickets, than the BTV scheduler can assure that threads which are higher priority gain more consideration from the CPU than threads that have lower priority. The reason behind is that the process which have more tickets has higher probability of getting the CPU.

Chapter 6, Problem 13E

(1)

Step-by-step solution

Show all steps

88% (8 ratings) for this solution

Step 1/3

There are two ways for maintaining a queue of processes for a processor:

- Processing core having its own run queue

- All processing core sharing single run queue

Step 2/3

Processing core having its own run queue: In this method each CPU works on its own local run queue. A processor can takes the process directly from queue.

Advantages:

• The processes present on the run queue are evenly distributed across the processor maintaining that run queue. This is known as load balancing.

• There is no interference or conflict even if two or more processors are making use of a scheduler simultaneously.

• At the time of making decision for a processing core, scheduler only requires to look at its local run queue.

Disadvantages:

• Extra required for maintaining different queue for different processor.

• Each processor has its own queue so when some processor's queue get completely empty then that processor get idle where as it might be possible that other processors are busy.

Step 3/3

All processing core sharing single run queue: In this method there is only one run queue which is shared by all CPU. Whenever any process assign a processor then process is taken from shared queue.

Advantages :

• This method leads to better utilization of processor.

• Any CPU that is idle can extract a runnable process from the common run queue and executes it.

• No extra work is required for maintaining different queue for different processor.

Disadvantages:

• There is load balancing problem in shared queue method.

• When there are different run queues, this can even lead to deadlock state if not protected with any locking scheme.

Chapter 6, Problem 14E

(5)

Step-by-step solution

Show all steps

100% (8 ratings) for this solution

Step 1/2

The formula to predict the length of the next CPU burst is as follows:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \ldots + (1-\alpha)^j \alpha t_{n-j} + \ldots + (1-\alpha)^{n+1} \tau_0 \qquad ......(1)$$

a.

Given $\alpha=0$ and $\mathrm{T}\,\tau_0 = 100 \text{ ms}$.

Substitute $\alpha=0$ in equation (1).

$$\begin{aligned} \tau_{n+1} &= 0t_n + (1-0)0t_{n-1} + \ldots + (1-0)^j 0t_{n-j} + \ldots + (1-0)^{n+1} \tau_0 \\ &= 0 + 0 + \ldots + 0 + \ldots + 1^{n+1} \tau_0 \\ &= \tau_0 \\ &= 100 \text{ ms} \end{aligned}$$

So, when $\alpha=0$ and $\mathrm{T}\,\tau_0 = 100 \text{ ms}$, the prediction for the next CPU burst will be 100 ms.

The recent history has no effect (current conditions are assumed to be transient).

Step 2/2

b.

Given $\alpha = 0.99$ and $\mathrm{T}\,\tau_0 = 10 \text{ ms}$.

Substitute $\alpha = 0.99$ in equation (1).

$$\tau_{n+1} = 0.99t_n + (1-0.99)0.99t_{n-1} + \ldots + (1-0.99)^j 0.99t_{n-j} + \ldots + (1-0.99)^{n+1} \tau_0$$

So, when $\alpha = 0.99$ and $\mathrm{T}\,\tau_0 = 10 \text{ ms}$, the prediction for the next CPU burst depends more on the most recent behavior of the process and less on the past history of the process.

Chapter 6, Problem 15E

(5)

Step-by-step solution

Show all steps

100% (5 ratings) for this solution

Step 1/2

A **CPU bound** process is one which has more CPU bound operations and an **I/O bound** process is one which has more of I/O bound operations.

• In the given scheduling algorithm, each process has initial 50 milliseconds of time quantum and a priority.

• So, if a process utilizes all of its time quantum, then 10 milliseconds more are assigned to it and its priority is also increased. This type of process is CPU bound process.

• On the other hand, when a process blocks for I/O before utilizing its full-time quantum, then its time quantum is reduced by 5 milliseconds but the priority remains same. This type of process is I/O bound process.

• It can be induced that after completion of each round, a CPU bound process gets more time quantum for execution and its priority is also increased.

Step 2/2

Hence, a regressive round-robin scheduler favors CPU-bound processes.

Chapter 6, Problem 16E

(40)

Step-by-step solution

Show all steps

100% (44 ratings) for this solution

Step 1/5

Turnaround time is the total time taken by process from submission to its completion.

Waiting time is the time spent by process in ready queue for getting the processor.

The burst time and priority of each processor is given below as:

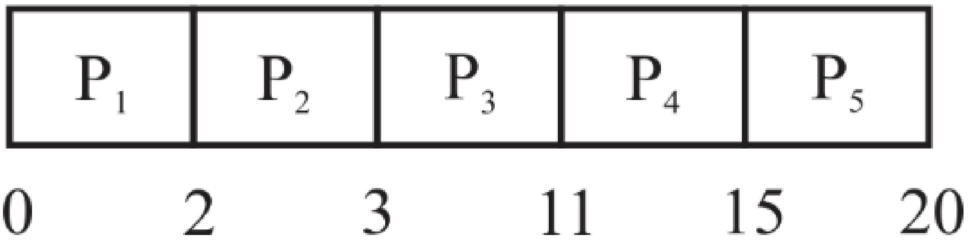
process	burst time	Priority
P_1	2	2

P_2	1	1
P_3	8	4
P_4	4	2
P_5	5	3

Step 2/5

FCFS algorithm: It Stand for First Come First Serve. In this, the process which comes first execute first, come second execute next.

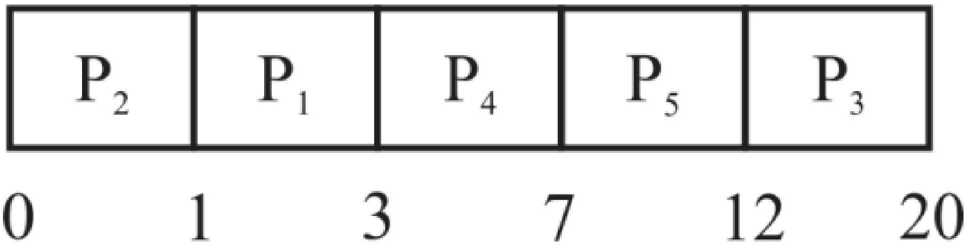
Gantt chart for FCFS algorithm:



In the above Gantt chart processes arrive in the order P_1, P_2, P_3, P_4 and P_5.

SJF algorithm: It stands for Shortest Job First. The process which has least burst time execute first the second least execute second and so on.

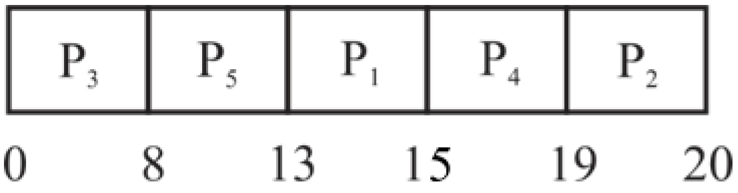
Gantt chart for SJF algorithm:



In the above Gantt chart processes when there is availability of CPU it is assigned to the processes.

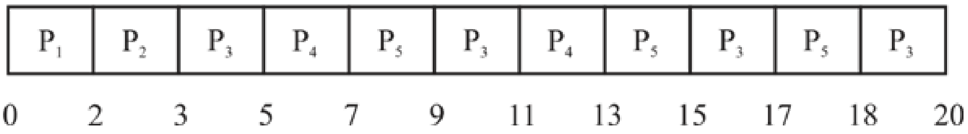
Non- preemptive priority algorithm: The process having highest priority showing with largest number execute first and showing second largest number execute second and so on.

Gantt chart for Non- preemptive priority algorithm:



Round robin algorithm: It works on time quantum.

Gantt chart for round robin algorithm with quantum=2



In the above Gantt chart each processes is interrupted after time slice of 2 quanta.

Step 3/5

$$\text{TurnaroundTime} = \text{BurstTime} + \text{WaitingTime}$$

Turnaround time of each processor in FCFS:

process	Turnaround time
P1	2
P2	3
P3	11
P4	15
P5	20

Turnaround time of each processor in SJF:

process	Turnaround time
P1	3
P2	1

P3	20
P4	7
P5	12

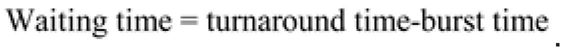
Turnaround time of each processor in Non-preemptive priority:

process	Turnaround time
P1	15
P2	20
P3	8
P4	19
P5	13

Turnaround time of each processor in Round Robin:

process	Turnaround time
P1	2
P2	3
P3	20
P4	13
P5	18

Step 4/5



Waiting time = turnaround time-burst time

Waiting time of each process in FCFS:

process	Waiting time
P1	0
P2	2
P3	3
P4	11
P5	15

Waiting time of each process in SJF:

process	Waiting time
P1	1
P2	0
P3	12
P4	3
P5	7

Waiting time of each process in Non-preemptive priority:

process	Waiting time
P1	13
P2	19

P3	0
P4	15
P5	8

Waiting time of each process in Round Robin:

process	Waiting time
P1	0
P2	2
P3	12
P4	9
P5	13

Step 5/5

$$\text{Average waiting time} = \frac{\text{Sum of waiting time of each process}}{\text{Total number of process}}$$

$$\text{Average waiting time using FCFS} = \frac{0+2+3+11+15}{5}$$
$$= 6.2$$

$$\text{Average waiting time using SJF} = \frac{1+0+12+3+7}{5}$$
$$= 4.6$$

$$\text{Average waiting time using non-preemptive priority} = \frac{13+19+0+15+8}{5}$$
$$= 11$$

$$\text{Average waiting time using RR} = \frac{0+2+12+9+13}{5}$$
$$= 7.2$$

SJF algorithm has the minimum average waiting time of 4.6.

Chapter 6, Problem 17E

(20)

Step-by-step solution

Show all steps

100% (15 ratings) for this solution

Step 1/5

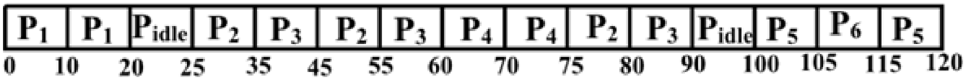
- It is given that Preemptive and round robin algorithm is used to schedule the process and each process has given a priority value. If there is no process to run then the task is taken as P_{idle}.

- According to **Preemptive algorithm**, if any process is running and any other higher priority comes into existence then the running process is preempted by higher priority process.

- **Round Robin scheduling algorithm** is based on the time quantum. The process will be provided to CPU by mechanism first come first serve. After processing each process at once with time quantum, the sequence will be repeated.

Step 2/5

Scheduling order of the process using Gantt chart is given below:


P1 | P1 | Pidle | P2 | P3 | P2 | P3 | P4 | P4 | P2 | P3 | Pidle | P5 | P6 | P5
0 10 20 25 35 45 55 60 70 75 80 90 100 105 115 120

Explanation of above Gantt chart:

- **In time interval 0 to 20:** P_1 will execute in this slot as arrival time of P_1 is 0 and no other process has arrived during the time slot.

- **In time interval 20 to 25:** No process will execute as P_1 already completed and no other process is available in ready queue.

- **In time interval 25 to 35:** P_2 will execute.

- **In time interval 35 to 45:** P_3 will execute.

- **In time interval 45 to 55:** P_2 will execute.

- **In time interval 55 to 60:** P_3 will execute. P_3 will be preempted as process P_4 with priority higher than P_3 has arrived.

- **In time interval 60 to 75:** P_4 will execute.

- **In time interval 75 to 80:** P_2 will execute.

- **In time interval 80 to 90:** P_3 will execute.

- **In time interval 90 to 100:** No process will execute as P_1 , P_2 , P_3 P_4 already completed and no other processes are available in ready queue.

- **In time interval 100 to 105:** P_5 will execute as no other process available at this moment.

- **In time interval 105 to 115:** P_6 will execute because it has more priority then P_5 .

- **In time interval 115 to 120:** P_5 will execute as P_6 has completed its burst time.

Step 3/5

Turnaround time is the total time taken by process from submission to its completion.

$$\text{Turn Around Time} = \text{Completion Time} - \text{Arrival Time}$$.

$$\begin{aligned}\text{Turnaround time for } P_1 &= 20-0\\ &= 20\end{aligned}$$

$$\begin{aligned}\text{Turnaround time for } P_2 &= 80-25\\ &= 55\end{aligned}$$

$$\begin{aligned}\text{Turnaround time for } P_3 &= 90-30\\ &= 60\end{aligned}$$

$$\begin{aligned}\text{Turnaround time for } P_4 &= 75-60\\ &= 15\end{aligned}$$

$$\begin{aligned}\text{Turnaround time for } P_5 &= 120-100\\ &= 20\end{aligned}$$

$$\text{Turnaround time for } P_6 = 115\text{-}105$$
$$= 10$$

Step 4/5

c.

Waiting time is the time spent by a process waiting in the ready queue.

Waiting time for $P_1 = 0\text{-}0 = 0$

$$\begin{aligned}\text{Waiting time for } P_2 &= (25\text{-}25)+(45\text{-}35)+(75\text{-}55)\\ &= 0+10+20\\ &= 30\end{aligned}$$

$$\begin{aligned}\text{Waiting time for } P_3 &= (35\text{-}30)+(55\text{-}45)+(80\text{-}60)\\ &= 5+10+20\\ &= 35\end{aligned}$$

$$\begin{aligned}\text{Waiting time for } P_4 &= (60\text{-}60)\\ &= 0\end{aligned}$$

$$\begin{aligned}\text{Waiting time for } P_5 &= (100\text{-}100)+(115\text{-}105)\\ &= 0+10\\ &= 10\end{aligned}$$

$$\begin{aligned}\text{Waiting time for } P_6 &= (105\text{-}105)\\ &= 0\end{aligned}$$

Step 5/5

d.

CPU utilization can be calculated as given below:

$$\begin{aligned}\text{CPU utilization rate} &= \frac{(\text{Highest turnaround time- CPU idle time})}{\text{highest turnaround time}}\times 100\\ &= \frac{120-15}{120}\times 100\\ &= 0.875\times 100\\ &= 87.5\%\end{aligned}$$

Chapter 6, Problem 18E

(0)

Step-by-step solution

Show all steps

100% (5 ratings) for this solution

Step 1/2

Nice value is used to provide a specific priority to the process at the time of invoking or calling.

- The default nice value is 0.

- The limits of nice value are -20 to +19.

- A lower nice value indicates a higher priority. It means -20 indicates a highest priority.

- A higher value indicates a lower priority. It means 20 or 19 shows the lower priority.

- A higher priority process is always assigned more CPU time than a low priority process.

Step 2/2

The non- root processes are allowed to assign nice values greater than or equal to zero.

- If the non- root processes are allowed to assign nice values less than zero, then all the non- root processes will provide themselves a higher priority.

- Then there will be an issue to which non-root processes the CPU is to allocated as they are all of high priority.

Hence, the root user can only be allowed to assign nice values less than zero.

Chapter 6, Problem 19E

(9)

Step-by-step solution

Show all steps

100% (18 ratings) for this solution

Step 1/2

Starvation due to scheduling algorithms

First – come, first – served (FCFS) algorithm: This scheduling algorithm allocates the CPU (Central Processing Unit) resources to the process that requests the resources first.

Shortest – Job – First (SJF): This scheduling algorithm allocates the CPU (Central Processing Unit) resources to the process that has the smallest CPU burst.

Round – robin: This scheduling algorithm allocates the CPU (Central Processing Unit) resources to each of the process in the circular queue for a time interval of up to 1 quantum (Time quantum is a small amount of generally from 10 to 100 milliseconds).

Priority Scheduling: This scheduling algorithm allocates the CPU (Central Processing Unit) resources to the process that has the highest priority (a priority is associated with each process).

Among all the four scheduling algorithms Shortest – Job – First and Priority scheduling could result in starvation.

Step 2/2

In the Shortest – Job – First scheduling algorithm, the longest job will be in the indefinite blocking or starvation. Starvation arises when shortest jobs keep on arriving before the current shortest job completes its execution. In this case, the shortest job is always available to execute and longest job will starve.

In the Priority scheduling algorithm, the lowest priority job will be in the indefinite blocking or starvation. Starvation arises when highest priority job keeps on arriving before the current highest priority job completes its execution. In this case, the highest priority job is always available to execute and low priority job will starve.

Therefore, Shortest – Job – First and Priority scheduling could result in starvation.

Chapter 6, Problem 20E

(3)

Step-by-step solution

Show all steps

100% (5 ratings) for this solution

Step 1/3

a)

• **If the two pointers referred to the same process in the ready queue using Round Robin scheduling Algorithm, the process priority get increased every time.**

• This phenomenon also called receiving preferential treatment.

• Preferential treatment means take the advantage over the other process in the ready queue which are having more priority.

Consider the two pointer p_1 and p_2 have the same references to the Process P_1 and these two pointers are put in the ready queue at time quantum t.

Then each process gets ½ = 50% of CPU time with at most 't' time units.

Each process waits for maximum time i.e (2-1)xt time unit until its next time quantum.

Step 2/3

b)

The two-major advantage of this preferential treatment scheme is as follows:

- In the Process control block, the CPU can give high priority to the Important processes. Hence, the important process will complete the job first.

- Higher Priority in the received treatment.

The Disadvantages preferential treatment are as follows:

- Shorter jobs will suffer to wait in the ready queue for its execution.

- Less Priority.

Step 3/3

c)

The best advice to modify the RR algorithm to avoid this issue without the duplicate pointers is allot a longer amount of time to processes deserving higher priority.

It means provide various time quantum to complete the jobs one after the other in the RR scheduling algorithm.

Chapter 6, Problem 21E

(6)

Step-by-step solution

Show all steps

92% (12 ratings) for this solution

Step 1/3

Given data:

- Number of I/O bound tasks=10

- Number of CPU bound tasks=1

- Time taken for context-switching=0.1 millisecond

Time quantum is 1 millisecond:

The CPU bound job will be preempted after the time quantum of 1 millisecond.

The I/O bound job will execute for 1 millisecond and after that context switching takes place.

When the time quantum is 1 millisecond, there will be 0.1 units of context switch for both CPU and I/O bound jobs.

Hence, the CPU utilization $= \frac{1.0}{(1.0+0.1)}$

$$= \frac{1}{1.1}$$
$$=0.909$$
$$=91\%$$

Step 2/3

Hence, the CPU utilization when the time quantum is 1 millisecond is **91%.**

Step 3/3

The time quantum is 10 milliseconds:

The CPU bound job will be executed for 10 milliseconds and then context switch takes place. The CPU bound job will not be preempted for 10 milliseconds.

Therefore, the time taken for a CPU bound job will be 10+0.1=10.1 ms.

The I/O bound job will take 10 milliseconds to complete. The I/O bound job will perform a context switch once for 1 millisecond. This is because it issues an I/O operation for every 1 millisecond of CPU utilization.

Therefore, the time taken for an I/O bound job will be 1ms + 0.1ms =1.1 ms.

As there are 10 jobs, the time taken for ten I/O bound jobs will be $10 \times 1.1 \text{ ms} = 11 \text{ ms}$.

Hence, the CPU utilization $= \frac{20}{(10.1+11)}$

$$= \frac{20}{21.1}$$

$$= 0.948$$

$$= 94.8\%$$

$$= 95\% \text{ (approx.)}$$

Hence, the CPU utilization when the time quantum is 10 millisecond is **95%.**

Chapter 6, Problem 22E

(0)

Step-by-step solution

Show all steps

Step 1/1

One possibility:

If the program uses only a large fraction of its time quantum and not the entire time quantum, thereby increase of priority associated with that process, The program could maximize the CPU time allocated to it by not fully utilizing its time quantums.

Another Possibility:

Round Robin Scheduling: A multilevel queue scheduling algorithm partitions the ready queue in several separate queues. The processes are permanently assigned to one queue, based on memory size, process priority, process type.

Breaks a process up into many smaller processes and dispatch each to be started and scheduled by the operating system. Separate queue might be used for foreground and background processes. Foreground queue might be Round Robin where as background may be FCFS algorithm.

Chapter 6, Problem 23E

(3)

Step-by-step solution

Show all steps

Step 1/2

(a)

$\beta > \alpha > 0$

β is the rate of change of priority when the process is running

α is the rate of change of priority when the process is waiting

0 is the rate when the rate when the process entered in ready queue.

• From the above question, all the processes having initial priority of 0 when it is entered into the ready queue.

• The priority is increased when the process is running from the ready state at the rate α which is greater than 0. Therefore, the highest priority of the process is $P(t) = \alpha(t - t_0)$.

• When the highest priority process is running (the first processes which entered into the ready queue), its priority is increasing at a rate ($\beta > \alpha > 0$). Therefore, no other processes will preempt until its completion.

• As the highest priority processes running is higher than the processes which are waiting, it seems to be a **First-Come-First-Serve (FCFS)** algorithm. The processors which are waiting for a long time will be run and its priority is increasing.

Hence, the algorithm results from the equation are **FCFS**.

Step 2/2

(b)

$\alpha < \beta < 0$

• As the highest priority processes are running, no other processes which are in the ready queue will not preempt the running process. Therefore, its priority is decreasing at a rate $\alpha < \beta < 0$.

• But, the process can be preempted by the new processes because when the new process enters into the ready queue, its initial priority is 0. The 0th priority is greater than the priority of the waiting process β .

Therefore, the last come process is served first. Therefore, the algorithm results from the explanation is **LIFO (Last In First Out).**

Chapter 6, Problem 24E

(5)

Step-by-step solution

Show all steps

100% (3 ratings) for this solution

Step 1/3

(a) FCFS—discriminates against short jobs since any short jobs arriving after long jobs

will have a longer waiting time.

Step 2/3

(b) RR—treats all jobs equally (giving them equal bursts of CPU time) so short jobs will

be able to leave the system faster since they will finish first.

Step 3/3

(c)Multilevel feedback queues—work similar to the RR algorithm—they discriminate

favorably toward short jobs.

Chapter 6, Problem 25E

(0)

Step-by-step solution

Show all steps

100% (1 rating) for this solution

Step 1/4

Windows scheduling algorithm uses 2 type of scheduling algorithm, one is priority based scheduling and preemptive scheduling algorithm.

There are 6 priority classes given below as:

- IDLE_PRIORITY_CLASS
- BELOW_NORMAL_PRIORITY_CLASS
- NORMAL_PRIORITY_CLASS
- ABOVE_NORMAL_PRIORITY_CLASS
- HIGH_PRIORITY_CLASS
- REALTIME_PRIORITY_CLASS

The values can be: IDLE, LOWEST, BELOW_NORMAL, NORMAL, ABOVE_NORMAL, HIGHEST, TIME_CRITICAL.

Step 2/4

A thread in the REALTIME_PRIORITY_CLASS with a relative priority of NORMAL is 24 (given from figure 6.22).

Step 3/4

A thread in the ABOVE_NORMAL_PRIORITY_CLASS with a relative priority of HIGHEST is 12 (given from figure 6.22).

Step 4/4

A thread in the BELOW_NORMAL_PRIORITY_CLASS with a relative priority of ABOVE_NORMAL is 7 (given from figure 6.22).

Chapter 6, Problem 26E

(1)

Step-by-step solution

Show all steps

Step 1/1

• The combination of REALTIME_PRIORITY_CLASS and TIME_CRTICAL priority provides the highest possible relative priority that is 31.

• Assume there is no threads belong to the REALTIME_PRIORITY_CLASS and that none may be assigned a TIME_CRTICAL priority.

• If above condition is true then a combination of HIGH_PRIORITY_CLASS and HIGHEST priority within that class with a numeric priority of 15 corresponds to the highest possible relative priority in Windows scheduling (According to figure 6.22).

Chapter 6, Problem 27E

(0)

Step-by-step solution

Show all steps

100% (1 rating) for this solution

Step 1/4

Time quantum is also known as time slice. It is small unit of time. Time quantum is used in time sharing system.

Time quantum and priority are inversely proportion to each other. Higher priority refers lesser the time quantum and vice versa.

Step 2/4

According to the Solaris operating system for time sharing needs the time quantum (in milliseconds) for a thread with priority 15 is 160 and for a thread with priority 40 is 40 (according to figure 6.23).

Step 3/4

A thread with priority 50 has used its entire time quantum without blocking, so the scheduler will assign a new priority to this thread that is 40 (according to figure 6.23).

Step 4/4

A thread with priority 20 blocks for I/O before its time quantum has expired, so the scheduler will assign a new priority to this thread of 52 (according to figure 6.22).

Chapter 6, Problem 28E

(1)

Step-by-step solution

Show all steps

100% (1 rating) for this solution

Step 1/4

CFS stands for Completely Fair Scheduler and it is default linux scheduling algorithm. Priority value is not provided directly by the CFS scheduler but it maintain the virtual run time of each task by pre-task variable vruntime which provide the runtime of each process.

Lower nice value shows higher priority and higher nice value shows the lower priority. A higher priority process is always assigned more CPU time than a low priority process.

Step 2/4

When both tasks A and B are CPU bound and nice value of A and B is given as -5 and +5 respectively then vruntime will be small for task A than task B as -5 for task A indicates a higher priority than task B.

Step 3/4

When task A is I/O bound and B is CPU bound, the vruntime will be smaller for task A than task B because task A has high priority and I/O bound process will run shortly before blocking for another I/O.

Step 4/4

When task A is CPU bound and task B is I/O bound the vruntime of task A will be less than value of vruntime of task B as CPU bound process will consume its time period whenever it will run on processor and it can be preempted by I/O bound.

Chapter 6, Problem 29E

(0)

Step-by-step solution

Show all steps

Step 1/3

In **Proportional share scheduler,** some CPU time is previously allocated to each processor and every job contains certain weight. And on the proportion of the weight, each job gets the part of available resources.

Step 2/3

A **priority inversion problem** is one in which a resource required by a high priority process is being acquired by a low priority process and that has preempted by medium priority process.

Due to which the highest priority process cannot be implemented before the medium priority process and lower priority process.

Example: A and B are 2 processes A has higher priority than B and B acquire an exclusive shared resource that is required by the A. let C be a medium priority that comes into existence. If B is running, A and C is blocked.

B can be preempted by C as it has higher priority. But A cannot run before B as A needs that resource which is acquired by B. So the priority of A and B is inverted.

Step 3/3

The solution to this problem is that a high priority process and a low priority process should interchange their priorities only till the high priority process completes its work with the resource, after that priorities can be same as initial.

This can be a solution which can be implemented within the context of a proportional share scheduler.

Chapter 6, Problem 30E

(0)

Step-by-step solution

Show all steps

Step 1/2

Rate-monotonic scheduling is the scheduling algorithm which is used today in real-time operating system. In this static-priority scheduling class is being used. In case of static-priority scheduling class if the duration of the cycle is short then priority of the job will be higher.

Earliest - deadline – first scheduling places a process in priority queue which is having a due deadline that means the process which is near its deadline given the highest priority, whereas **rate-monotonic scheduling** places a process in priority queue having shorter period which leads to higher priority.

Step 2/2

Circumstances where rate-monotonic scheduling algorithms are inferior in comparison to the earliest-deadline first scheduling algorithm in meeting the processes are as follows:

• In case of static priority assignment rate monotonic is preferred whereas in case of dynamic priority assignment earliest deadline first assignment is preferred.

• When tasks-sets with the higher processor utilization is required then in that situation Rate-monotonic scheduling algorithm is not preferred.

Chapter 6, Problem 31E

(9)

Step-by-step solution

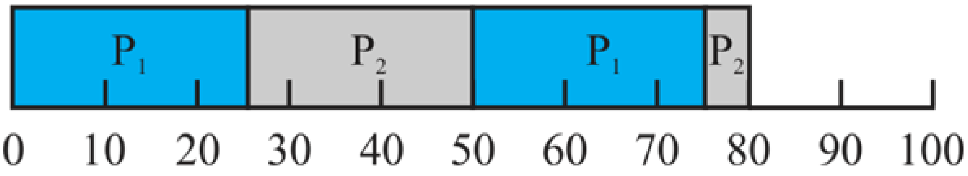
Show all steps

100% (6 ratings) for this solution

Step 1/2

Rate-monotonic scheduling places a process in priority queue having shorter period which leads to higher priority. It plays an important role in real time as it guarantee that the tasks are basically schedulable. The tasks which is meeting their deadline in a proper time is said to be schedulable.

Consider the following Gantt chart:



Explanation of the Gantt chart:

- Process P_1 has the shorter period so it will be scheduled first at time $t=0$. Its burst will finish at $t=25$, giving the CPU to process P_2 .

- This process will get to execute till the end of the deadline for process P_1 at $t=50$, upon which it will get preempted.

- Thus process P_2 got to execute for $t=50-25=25$. Process P_1 will execute till $t=50+25=75$.

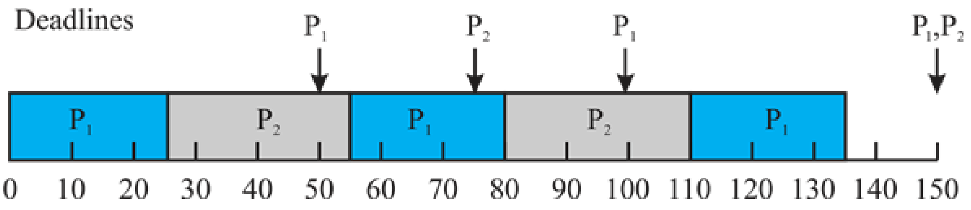
- This is the deadline for process P_2 but it still needs $t=30-25=5$ to complete its burst.

Hence, it can be said that the 2 processes cannot be scheduled by rate-monotonic scheduling.

Step 2/2

Earliest-deadline-first scheduling places a process in priority queue which is having a due deadline that means the process which is near its deadline given the highest priority.

Consider the following Gantt chart:



Explanation of the Gantt chart:

- These two processes can be scheduled with earliest-deadline-first scheduling. Scheduling will start with process P_1 since it has the earliest deadline $t=50$.

- Process P_2 will be scheduled in at $t=25$. It will not get preempted by process P_1 at $t=50$ as it has the earlier deadline, that is $t=75<t=100$.

- It will relinquish control of the CPU at the end of its burst, at $t=25+30=55$.

- At this point, process P_1 will be scheduled in. It too will execute till its burst's end, till $t=55+25=80$.

- At this time, process P_2 will be scheduled in as process P_1 's deadline has been met. It will not be preempted by process P_1 at $t=100$ as both the processes' deadlines are the same, that is $t=150$.

- Process P_1 will get the CPU at $t = 80 + 30 = 110$. Its burst will finish at $t = 110 + 25 = 135$, where the CPU will remain idle till $t = 150$.

Hence, it can be said that the 2 processes can be scheduled by earliest-deadline first scheduling.

Chapter 6, Problem 32E

(1)

Step-by-step solution

Show all steps

100% (2 ratings) for this solution

Step 1/4

A **hard real time** system is one in which a process is serviced within a given deadline or time.

The performance of the hard real time systems is affected by the interrupt latency and dispatch latency.

Step 2/4

Interrupt latency is the amount of time between the occurrence of the interrupt to the CPU and the first respond of the CPU to the interrupt.

For a hard real time system, Interrupt should be bounded as the CPU has to first complete the currently running instruction, save its state and then the interrupt will be serviced by ISR (Interrupt Service routine) which will check what type of interrupt h as been occurred.

It is very important in real operating system to reduce the interrupt latency because if interrupt latency is reduced, it will be ensured that the real time system get immediate attention.

Step 3/4

Dispatch latency is the amount of time required for the scheduler to stop currently running process and start another.

It must be bounded with hard real time system because preemption is needed for a current process to be stopped, save its state and then the CPU will be assigned to a high priority process immediately. To service it within in a given deadline, immediate retrieval to the CPU is necessary for real time system to reduce dispatch latency.

Step 4/4

So, minimization of the interrupt and dispatch latency should be done for ensuring that all the tasks in the real time need proper attention.

Therefore, both are necessary to be bounded in hard real time system.

Chapter 7, Problem 1PE

(2)

Step-by-step solution

Show all steps

100% (8 ratings) for this solution

Step 1/1

Dead lock in real life

When the resources are not shared among people or processes, the situation is known as dead lock. The three examples to show the deadlock condition in the real life are as given below:

1. When two cars are trying to cross the same one lane bridge from the opposite ways, the deadlock condition occurs.

2. When two trains are trying to cross the same one lane bridge on single track from the opposite ways, the deadlock condition occurs.

3. When two people are playing a game of baseball and one of them has the bat while the other holds the ball. In this situation, if the resources are not shared, the game cannot be played and a deadlock occurs.

Chapter 7, Problem 2PE

(0)

Step-by-step solution

Show all steps

100% (8 ratings) for this solution

Step 1/2

Unsafe state:

Unsafe state is the situation that happens when the system contains less available resources but guarantees the completion of at least one job running on the system. But this may lead to a deadlock problem.

Deadlock:

In a computer system, a set of blocked processes each holding a resource and it is waiting to acquire the resource held by another process in the set is called a deadlock problem.

Step 2/2

Example:

Consider that the system with 15 devices of same type and it is in an unsafe state.

Job Number	Device allocated	Maximum required	Remaining needs
1	5	9	4
2	3	7	4
3	4	8	4

From the table,

• The total number of devices allocated to the system is “12” and the total number of available devices is “15”.

• The remaining devices which are available idle in the system is “3”$(15-12 = 3)$.

• This proves that the system is in unsafe state. Because, the number of available devices is “3” and it is less than the number needed to satisfy the maximum requests “4”.

• So, none of the job completes its execution and it suffers from the problem of deadlock.

• But there is way to turn the system into a safe state,

o The way is that to put the Job number “2” into the execution and it will face the execution time problem (for example: “division by zero” problem) and it will be terminated after some period of time by releasing 3 allocated devices.

o Now, the system turns into safe state and it uses the released resources of “job2” to complete the execution of “job1” and “job3”.

Therefore, the system in an unsafe sate is not necessarily deadlocked.

Chapter 7, Problem 3PE

(15)

Step-by-step solution

Show all steps

100% (16 ratings) for this solution

Step 1/9

Allocation Max Available

A B C D A B C D A B C D

P_0 0 0 1 2 0 0 1 2 1 5 2 0

P_1 1 0 0 0 1 7 5 0

P_2 1 3 5 4 2 3 5 6

P_3 0 6 3 2 0 6 5 2

P_4 0 0 1 4 0 6 5 6

Step 2/9

a)

Need Matrix= Max - Allocataion

A B C D

P_0 0 0 0 0

P_1 0 7 5 0

P_2 1 0 0 2

P_3 0 0 2 0

P_4 0 6 4 2

Step 3/9

b)

The available matrix is [1 5 2 0]. So, the safety sequence occurs when the criteria Need<= Available is satisfied. From the Need matrix calculated above, the process P_0 does not require any resources. Also, the need matrix of P_3 is less than the available matrix.

Step 4/9

So, take the process P_0 and calculate [Available] = [Available]+[Allocation(P_0)], so Available = [1 5 2 0] + [0 0 1 2] = [1 5 3 2].

Next, take the process P_2 and update the available matrix as: [1 5 3 2] + [1 3 5 4] = [2 8 8 6].

Step 5/9

Now, the need matrix of process P_3 is less than the available matrix, so calculate the available matrix as: [2 8 8 6] + [0 6 3 2] = [2 14 11 8].

Step 6/9

Next, the need matrix of process P_4 is less than the available matrix, so calculate the available matrix as: [2 14 11 8] + [0 0 1 4] = [2 14 12 12].

Step 7/9

In the last step, the process P_1 is taken, and its need matrix is less than the available matrix, so calculate the available matrix as: [2 14 12 12] + [1 0 0 0] = [3 14 12 12].

Step 8/9

So, the system is in a safe state and the safe sequence is $< P_0, P_2, P_3, P_4, P_1 >$

Step 9/9

c)

When request (0, 4, 2, 0) arrives from P_1, first the Banker's algorithm checks whether request $\leq$ Available, since (0, 4, 2, 0) < (1, 5, 2, 0) algorithm pretends, that this request can been fulfilled and arrived at new state.

$$\begin{aligned}
&\boldsymbol{Avaliable = Avaliable - Request_i}\\
&\boldsymbol{Allocation_i = Allocation_i + Request_i}\\
&\boldsymbol{Need_i = Need_i - Request_i}
\end{aligned}$$

Allocation Max Need Available

Process A B C D A B C D A B C D A B C D

P_0 0 0 1 2 0 0 1 2 0 0 0 0 1 1 0 0

P_2 1 3 5 4 2 3 5 6 1 0 0 2 2 4 6 6

P_3 0 6 3 2 0 6 5 2 0 0 2 0 2 10 9 8

P_1 0 4 2 0 1 7 5 0 0 3 3 0 3 14 11 8

P_4 0 0 1 4 0 6 5 6 0 6 4 2

At this state after executing safety algorithm, one of the ordering (P_0, P_2, P_3, P_1, and P_4) satisfies the safety requirements.

Note: The other ordering of the process can be (P_0, P_2, P_1, $P_{3,}$ and P_4), (P_0, P_2, P_1, $P_{4,}$ and P_3) until and unless there is no possibility of deadlocks.

Hence, request can be granted immediately to process P $_{1.}$

Chapter 7, Problem 4PE

(0)

Step-by-step solution

Show all steps

Step 1/1

One way to ensure that circular wait should hold is to impose a total ordering of all resources types and to require that each process request resource in an increasing order of enumeration.

Let R= {R_1, R_2, . . . ,R_m} be the set of resource types. Each resource type is assigned a unique number which is of type integer. It allows us to compare two resources and to determine whether one precedes another in our ordering.

$F{:}R \rightarrow N$ is one-to-one function, N is a set of natural numbers.

To have no circular wait.

One must have $F(R_i)<F(R_{i+1})"i$

$F(R_0)<F(R_1)<..........<F(R_n)<F(R_0)$

By the transitivity rule $F(R_0)<F(R_0)$

is impossible. Hence no circular wait. But in containment. Whenever a thread acquire the lock for object A,B, C, D, E it must first acquire lock for another object F.

$A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow A$ is in deadlock.

$A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F$ not in deadlock.

i.e. if any two thread will be in dead lock than sixth thread will be in to avoid the dead lock. That is object F can occur and be acquiring a lock whenever a deadlock is to occur.

Chapter 7, Problem 5PE

(0)

Step-by-step solution

Show all steps

100% (1 rating) for this solution

Step 1/1

Banker Algorithm

The given code segment implements the banker’s algorithm. In the given code segment, two for loos are executed as the outer for() loops for n number of times which makes the run time to n^2 .

There are two if() conditions given in the code segment and a for loop which will be executed till m times. So, in this case the collective run time will be $O(mn^2)$.

In the worst case if both the conditions are satisfied, the for() loop will be executed till $O((m+m)n^2)$ times and $m+m$ represents m run time. Thus in any case the run time of the code segment will be $O(mn^2)$.

Chapter 7, Problem 6PE

(1)

Step-by-step solution

Show all steps

100% (4 ratings) for this solution

Step 1/2

2481-7-12E SA: 8683

SR: 6376

a)

The arguments in favour of installing the deadlock-avoidance algorithm are:

1. It avoids deadlocks and the costs incurred by rerunning the terminated jobs.

2. Reduces the wastage of resources by rerunning the jobs and also saves the time to run the aborted processes all over again.

3. Resource utilization will be maximized and the machine idle-time is reduced.

Step 2/2

b)

The arguments against installing the deadlock avoidance algorithm are:

1. The average execution time per job will be increased by about 10 percent.

2. The turnaround time would increase by about 20 percent. It introduces an overhead on the execution

3. Deadlocks occur about twice per month and they cost little than the overhead added by installing the deadlock avoidance algorithm.

Was this solution helpful?

4
0

Chapter 7, Problem 7PE

(1)

Step-by-step solution

Show all steps

84% (6 ratings) for this solution

Step 1/1

Detection of starving processes

No, the detection of the starve process is very difficult. A process is said to be starve process, if the process is never be executed. If a process is waiting for a long time for the requested resources, and the time is beyond the response time, than, the process is said to be starving. As the response time for any process is increased, the probability of starving is increased.

The system can calculate the response time T to identify the problem of starvation. The resources can be allocated to a fix time interval. The system must have to fix the time T for any process P for execution. In any condition, the response time should not be delayed. By doing so, the system can deal with the problem of starving.

Chapter 7, Problem 8PE

(3)

Step-by-step solution

Show all steps

100% (2 ratings) for this solution

Step 1/2

2481-7-7E SA: 8683

SR: 6376

a)

Deadlock will not occur. By the given resource allocation policy, the necessary condition that cannot occur is the **No Preemption** condition. No preemption states that resources cannot be preempted, that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task. But here, if blocked process has desired resource, then these resources are taken away from it and are given to the requesting process. So, it is not holding no preemption condition.

Step 2/2

b)

Indefinite blocking can occur. Because, processes might continuously come and take resources from the blocked process (as P2 gets resources from the blocked process P0); the blocked process might not get a chance to fulfill its needs.

Chapter 7, Problem 9PE

(3)

Step-by-step solution

Show all steps

100% (1 rating) for this solution

Step 1/1

Deadlock detection

Yes, the deadlock detection can be implemented with the help of safety algorithm. The value of Max is representing a process which is requested maximum time and Max is a vector. The value if i represent the resources for any process.

In the calculation of safety algorithm, Need matrix is required. The Need matrix represents

$$\text{Need} = \text{Max}_i - \text{Allocation}_i$$

Thus,

$$\text{Max} = \text{Need}_i + \text{Allocation}_i$$

It is given that *Waiting* matrix is playing the same role as played by the *Need* matrix. So, it can be said that

$$\text{Max} = \text{Waiting}_i + \text{Allocation}_i$$

So, the deadlock detection can be implemented with the help of safety algorithm.

Chapter 7, Problem 10PE

(2)

Step-by-step solution

Show all steps

100% (6 ratings) for this solution

Step 1/1

2481-7-16E SA: 8683

SR: 6376

There are four conditions that must be hold for a deadlock to occur.

1. Mutual exclusion

2. Hold and Wait

3. No pre-emption

4. Circular wait

Circular-wait occurs only when there are a set of processes where, one process is waiting for the resources of another process. There must be atleast more than one process. With a single process, the circular wait condition does not hold. So it is not possible to have a deadlock involving only one process.

Chapter 7, Problem 11E

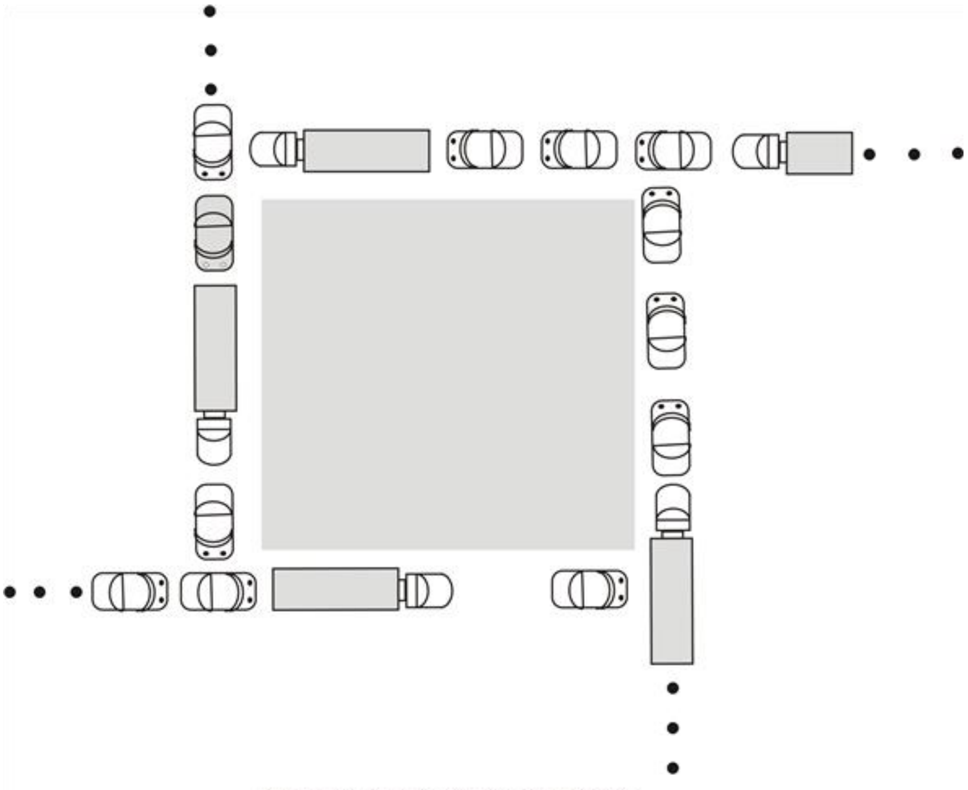
(1)

Step-by-step solution

Show all steps

91% (11 ratings) for this solution

Step 1/5



TRAFFIC DEADLOCK eXERCISE

a. The four necessary conditions for deadlock hold in this example for the following reasons: (i) Mutual Exclusion: Each of the vehicles present in the streets hold a non-sharable resource: the part of the road they occupy, which they cannot share with the other vehicles.

Step 2/5

(ii) Hold and Wait: Each of the vehicles hold the space resource they occupy and are waiting the space in front of them to be freed by other waiting vehicles.

Step 3/5

(iii) No Preemption: There is no possibility of preemption as none of the vehicles can give up their resource. In this situation preemption would have to take the form of a vehicle pulling into a parking lot, or a crane reaching down and lifting a vehicle off the road.

Step 4/5

(iv) Circular Wait: Circular wait is present because each vehicle is waiting for the space in front of it, and some vehicles occupy spaces where two vehicles wait on them. It is thus possible to trace a cycle of waiting cars. This is the weakest assertion in the set, though, and is clearly untrue out at the edges somewhere, since some car can clearly move someplace in the city. If you have ever experienced grid-lock, though you know that this is small comfort, and that a rule to avoid even "local" deadlock is extremely desirable.

Step 5/5

b. The simple rule that could be used to avoid traffic deadlocks in such a system is that intersections should always remain clear as lights change. In this way, the resource of space in the intersection is freed for use at periodic intervals (light changes).

Chapter 7, Problem 12E

(1)

Step-by-step solution

Show all steps

100% (3 ratings) for this solution

Step 1/1

Deadlock is a condition where a process enters a never ending wait situation and hence the execution is halted for uncertain time duration.

The major drawback of a process entering deadlock situation is, the resources the process was using at the time of deadlock arrival gets occupied until and unless the process wakes up from deadlock and finishes execution.

Discussing the four necessary conditions of deadlock for this multi-threaded system; the reader-writer locks may contribute in the way discussed below:-

• As there are several reader-writer locks, mutual exclusion is achieved successfully.

• Each thread can hold one reader and writer, while keep waiting for another lock. Hence, hold-and-wait condition is also possible.

• No reader-writer lock is available for taking away hence, preemption is not possible.

• As multiple threads may wait for a reader-writer lock already acquired by some other thread in the process; process may go into a circular wait condition.

If a multi-threaded application uses several reader-writer locks, the deadlock is more likely to arrive. Hence, the answer is YES a deadlock arrives in such situation.

Chapter 7, Problem 13E

(0)

Step-by-step solution

Show all steps

100% (1 rating) for this solution

Step 1/4

Refer to the code provided in Figure 7.4 of chapter 7 of the textbook for complete code.

Step 2/4

Occurrence of deadlock by the CPU scheduler: There are four necessary conditions that must hold for a deadlock to occur. They are as follows:

• Mutual exclusion – At least one resource must be in non-shareable mode.

• No-preemption – The resources can only be released voluntarily by the process.

• Hold and wait – A process must be holding at least 1 resource and should be waiting for another resource held by another process.

• Circular wait – In a set of processes {P1, P2…..Pn} there must exist waiting between them such that process P1 is waiting for the resource held by process P2 and so on.

Step 3/4

The function of CPU schedular is to select the process ready to execute and allocate them CPU.

The code given in Figure 7.4 uses two thread requesting for same process “first_mutex” and “second_mutex” but in different order. The work of CPU scheduler here is to provide CPU to the process which is ready for execution.

• Suppose, there is a situation that in do_work_one function, only one statement that is first_mutex, gets acquired but second_mutex is not acquired by do_work_one function.

• Now, after acquiring of the first_mutex CPU scheduler will give control to do_work_two function. In do_work_two function the first statement gets executed and it is seen that the second_mutex is not acquired at the moment. So, do_work_two function acquired the second_mutex.

• Now, when second statement of any function gets executed, then the statement needs to wait for the other mutex.

• This is because second statement of do_work_one function requires second_mutex which is currently held by do_work_two function.

• Similarly, second statement of do_work_two function requires first_mutex which is being held by do_work_one function.

• In this way, both of the functions enters the waiting state and deadlock occurs.

Step 4/4

Contribution of CPU scheduler for deadlock:

- The role of CPU scheduler in the given program is to provide CPU to the thread that is ready for execution.

- A CPU scheduler uses many algorithms to do so like giving fix time slot to a thread (pre-emption).

- Suppose, there is a situation that in do_work_one function, the first statement gets executed and the CPU scheduler gives the CPU time to do_work_two function.

- The do_work_two function will hold second_mutex while waiting for first_mutex while the do_work_one function is holding the first_mutex and waiting for second_mutex to be free.

- In this case, there will be a deadlock as both the threads are holding a mutex and requesting for another mutex.

Hence, this way CPU scheduler can contribute in deadlock process.

Chapter 7, Problem 14E

(0)

Step-by-step solution

Show all steps

100% (2 ratings) for this solution

Step 1/2

Function to fix deadlock:

The deadlock condition can be prevented by adding a new lock named as lock3. This is acquired before the two locks are acquired.

The transaction function to prevent the deadlock condition is as given below:

Step 2/2

```
void transaction(Account from, Account to, double amount)

{

Semaphore lock1, lock2, lock3;

//acquire the lock3

Wait(lock3);
// request to get the lock for lock1

lock1 = getLock(from);

// request to get the lock for lock2

lock2 = getLock(to);

// acquire the lock to perform the transaction

wait(lock1);

// acquire the lock to perform the transaction

wait(lock2);

// The first lock is used to withdraw the balance from any account and the second lock is used to deposit the balance in an account.

// withdraw the balance from the account

withdraw(from, amount);

// deposit the balance to the account

deposit(to, amount);

//After performing the transactions, both the locks is released by using signal parameter to prevent the deadlock condition.

// first release the lock 3

signal(lock3);

// release the lock2 and lock1 after performing transaction

signal (lock2);

signal(lock1);

}
```

Chapter 7, Problem 15E

(0)

Step-by-step solution

Show all steps

100% (1 rating) for this solution

Step 1/2

a.

Runtime overheads:

- The algorithm in which it is less efficient the resource allocation graph scheme is said to be known as banker's algorithm. It is one of the algorithms of deadlock-avoidance schemes.

- The runtime overheads get increased by the deadlock avoidance such that it is only because of the current resource allocation cost.

Step 2/2

b.

System throughput:

- The request should be given by the process to use the resource such that after completion of using it, the resource must be released by the process.

- For the prevention of deadlock, the deadlock avoidance scheme will allow a number of resources for concurrent use than schemes.

- Finally, the system throughput will get increased by the deadlock avoidance scheme.

Chapter 7, Problem 16E

(4)

Step-by-step solution

Show all steps

100% (3 ratings) for this solution

Step 1/7

In the banker's algorithm, a process must not request a maximum number of instances of each resource type than the number of resources available in the system. This algorithm allocates the requested number of resources to the process only if the system remains in a safe state. Otherwise, it does not allocate the requested number of resources to the process to avoid deadlock. Then the process must wait until the resource available.

- Available and Max are the data structures of the banker's algorithm.

- The Available is a vector contains *m* number of available resources of each resource type for *n* number of processes.

• The Max indicates the maximum number of demands of each process for each resource types.

Step 2/7

a. Increase Available:

• If the Available length increased, then the number of available resources of each resource type will increase.

• Then the instances of each resource type of a process will increases. The process can request for more instances of each available resource type.

• This change can be made safely without deadlock.

Step 3/7

b. Decrease Available:

• If the Available length decreased, then the number of available resources of each resource type will decrease. The resource will permanently remove from the system.

• Then the instances of each resource type of a process will decrease. The process cannot request more instances of each available resource type.

• If the process requests the number of instances of each resource type than the number of resources available in the system, then the deadlock will occur.

• This change may lead to deadlock.

Step 4/7

c. Increase Max for one process:

• If the Max increased for one process, the number of demands of the process will increase. The process can demand for more resources than allowed.

• If the process requests more instances of a resource than the available resources of each type, then the process must wait until another process release that resource.

• If another process does not release the resource, then this condition leads to deadlock and system will be in the unsafe state.

Step 5/7

d. Decrease Max for one process:

• If the Max decreased for one process, the number of demands of the process will decrease. The process cannot demand more resources than allowed.

- If the process requests fewer instances of a resource than the available resources of each type, then the process must not wait until another process release that resource.

- This condition does not lead to deadlock and system will be in the safe state.

Step 6/7

e. Increase the number of processes:

- If the number of processes increases in the system, the resource allocation will be difficult.

- If two or more processes requests for same instances of a resource at the same time, then some of the processes must wait until the resource is available.

- This condition has the possibility of deadlock. The system will be in an unsafe state.

Step 7/7

f. Decrease the number of processes:

- If the number of processes decreases in the system, the resource allocation will be easier.

- If the number of processes is less than the number of resources available in the system, then the system remains in a safe state.

Chapter 7, Problem 17E

(2)

Step-by-step solution

Show all steps

100% (6 ratings) for this solution

Step 1/2

When there are 4 resources to be shared among 3 processes, and one process can occupy at most 2 resources, the system will be deadlock free. It has been explained through following points:

- Two processes will hold 2 resources (1 each).

- Third process will occupy 2 resources.

- Third process can’t hold more than this, hence, it can release these resources after use.

- When any of the 2 or both process will be released by third process, other process will be able to hold the free resource.

Step 2/2

The situation can be better understood by the diagram given below:

In figure 1 all the resources R1, R2, R3, and R4 have been allocated among 3 processes P1, P2 and P3.

R1 has been allocated to P1.

R2 has been allocated to P2.

R3, R4 has been allocated to P3.

P3 has occupied the maximum number of resources it can hold; hence, it is not waiting any other process to release resource. It will first release the resource, then only will occupy any other resource.

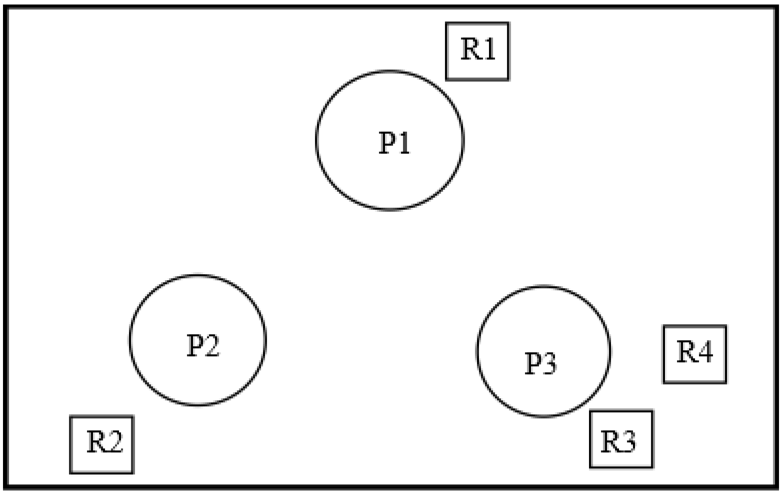


Figure 1

Now when P3 has released R3 as shown in Figure 2 but still holding R4. R3 now can be allocated to any of the process.

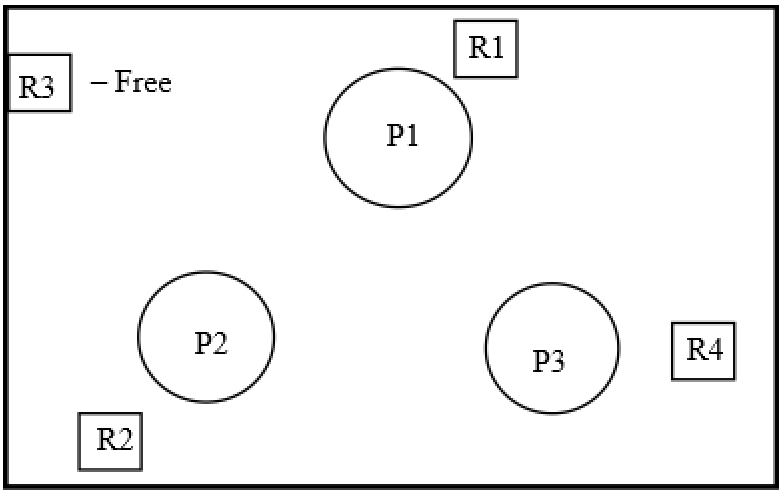


Figure 2

In figure 3, R3 has been allocated to P2. Hence, P2 has 2 resources R2 and R3.

R1 has been allocated to P1.

R4 has been allocated to P3.

This diagrammatic explanation proves that it is not a deadlock situation.

Chapter 7, Problem 18E

(3)

Step-by-step solution

Show all steps

Step 1/2

Proof using contradiction to show the system is deadlock free if the following two conditions hold:

Condition 1: The max need of each process is between 1 and m resources.

Condition 2: The sum of all max needs is less than $m+n$.

Let, the system is not deadlock free.

If there is deadlock in the system, then the sum of all allocations of processes is equal to the number of resources.

That is, $A = \sum_{i=1}^{n} Allocation(i) = m$.

This is because only one type of resource is present, and a process can request or release only one resource at a time.

From condition 2, The sum of all max needs is less than $m+n$. That is,

$$M < m+n$$

$$N + A < m+n$$

$$N + m < m+n$$

$$N < m$$

Therefore, $N < m$. That is, the sum of all needs of processes is less than the number of resources. Then, the need of at least one process is 1. That is $Need(i) \; = 0$. This condition shows that there is no deadlock.

Step 2/2

Form condition 1, the maximum need of each process is between 1 and m resources. That is, a process can release at least one resource. Then, the m resources of same type can be shared by $n-1$ processes.

Here, the two conditions satisfy the deadlock free condition.

Therefore, the system that holds the two conditions is deadlock free.

Chapter 7, Problem 19E

(3)

Step-by-step solution

Show all steps

Step 1/2

Use the following rule to perform the dining-philosophers problem without causing deadlock.

When a philosopher makes a request for the chopstick, there are three situations:

1. If there are no chopsticks, the request cannot be accepted and philosopher should wait until the availability of chopsticks.

2. If there is one chopstick, the request cannot be satisfied until there are two chopsticks available.

3. If there are two chopsticks available, the request can be accepted and philosopher can eat.

Step 2/2

Algorithm:

```
do
{
Think
if(chopstick[i]&& chopstick [i+1])
{
pickup(chopstick [i], chopstick [i+1 mod 5]);
Eat;
putdown(chopstick [i], chopstick [i+1 mod 5])
}
else
{
wait(chop stick [i]);
wait(chopstick [(i+1)%5])
}
}while(true);
```

Chapter 7, Problem 20E

(0)

Step-by-step solution

Show all steps

Step 1/3

Use the following rule to perform the dining-philosopher’s problem:

When a philosopher makes a request for the chopstick, there are three situations:

1. If there are no chopsticks, the request cannot be accepted and philosopher should wait until the availability of chopsticks.

2. If there is one chopstick, the request cannot be satisfied until there are three chopsticks available.

3. If there are two chopsticks, the request cannot be satisfied until there are three chopsticks available.

4. If there are three chopsticks available, the request can be accepted and philosopher can eat.

Step 2/3

Algorithm:

```
do
{
state[i]=Thinking;
state[i] = hungry;
if(chopstick[i]&& chopstick[i+1]&& chopstick[i+2])
{
pickup(chopstick[i],chopstick[i+1],chopstick[i+2mod 5]);
Eat;
pickdown(chopstick[i],chopstick[i+1],chopstick[i+2mod 5]);
}
else
{
wait(chopstick[i]);
wait(chopstick[i+1]);
wait(chopstick[(i+2)%5]);
}
}while(true)
```

Step 3/3

Explanation :

- At the start, start do-while loop means each state will be verified.

- At the first philosopher think, whether he is going to eat or not.

- Check whether three chopsticks are available or not. If yes, then,pickup the chopstick and start eating.

- Else wait for the chopsticks until Philosopher get all the three chopsticks.

Chapter 7, Problem 21E

(0)

Step-by-step solution

Show all steps

Step 1/3

Consider that a system is using two resources R_1 and R_2 and there are three processes P_1, P_2 and P_3. The resources will be allocated to the processes based upon the requirements.

The allocation status of the two resources R_1 and R_2 for the processes P_1, P_2 and P_3 are as follows:

$R_1R_2R_1R_2$

Consider the following:

$$\begin{aligned}&\mathit{Process}\ P_1 1\ 0\ 1\ 1\\&\mathit{Process}\ P_2 1\ 1\ 2\ 3\\&\mathit{Process}\ P_3 1\ 2\ 2\ 2\end{aligned}$$

Step 2/3

If the process P_2 were to make a request with (1, 0), then the system will allow it because the processes can complete in the sequence as follows:

$$P_1, P_3, P_2.$$

If the process P_2 were to make a request with (0, 1), then the system will allow it and the processes can complete in the sequence as follows:

$$P_3, P_1, P_2.$$

Step 3/3

But, if the process P2 were to make a request with (1, 1), then the request will get rejected as no process can satisfy the request. Hence no process will be able to complete.

Chapter 7, Problem 22E

(17)

Step-by-step solution

Show all steps

100% (23 ratings) for this solution

Step 1/7

Determination or safe or unsafe states:

In the Banker’s algorithm, four types of matrices are used. Those are,

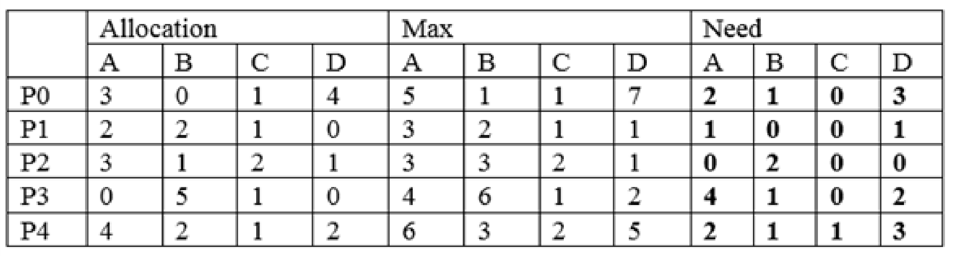
1. **Available:** An m length vector representing the number of available instances of any resource at a time.

2. **Max:** This is an n to m matrix (n specifying different processes). Each entry implies the maximum instances of a resource type that can be requested by a particular process.

3. **Allocation:** This is also an n to m matrix. Each entry specifies instances of a resource held by a particular process at a time.

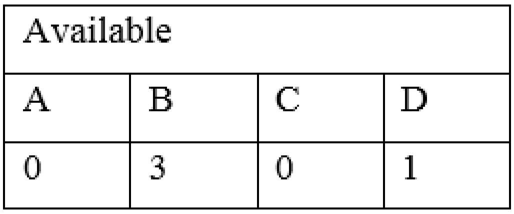
4. **Need:** An n to m matrix. Each entry in this matrix implies the number of instances of a resource needed by a particular process at a time. According to safety algorithm, **Need = Max – Allocation**

The snapshot of the system and **Need** matrix is,



	Allocation				Max				Need			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	3	0	1	4	5	1	1	7	**2**	**1**	**0**	**3**
P1	2	2	1	0	3	2	1	1	**1**	**0**	**0**	**1**
P2	3	1	2	1	3	3	2	1	**0**	**2**	**0**	**0**
P3	0	5	1	0	4	6	1	2	**4**	**1**	**0**	**2**
P4	4	2	1	2	6	3	2	5	**2**	**1**	**1**	**3**

The available matrix is,

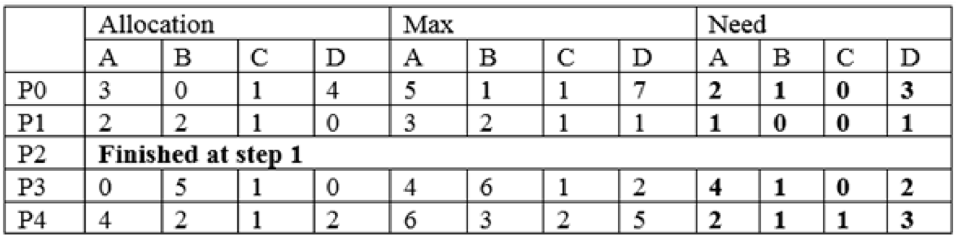


Available			
A	B	C	D
0	3	0	1

Step 1:

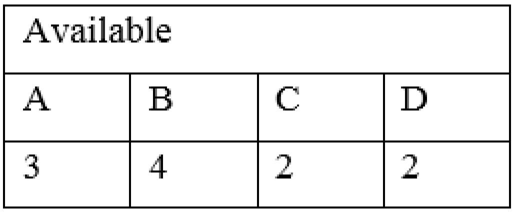
As, Need of P2 is less than Available, the requirements of P2 can be fulfilled with the available resources and P2 will be completed. After completion, it will release all resources to the Available pool.

So, the current snapshot of the system is,



	Allocation				Max				Need			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	3	0	1	4	5	1	1	7	**2**	**1**	**0**	**3**
P1	2	2	1	0	3	2	1	1	**1**	**0**	**0**	**1**
P2	**Finished at step 1**											
P3	0	5	1	0	4	6	1	2	**4**	**1**	**0**	**2**
P4	4	2	1	2	6	3	2	5	**2**	**1**	**1**	**3**

And,

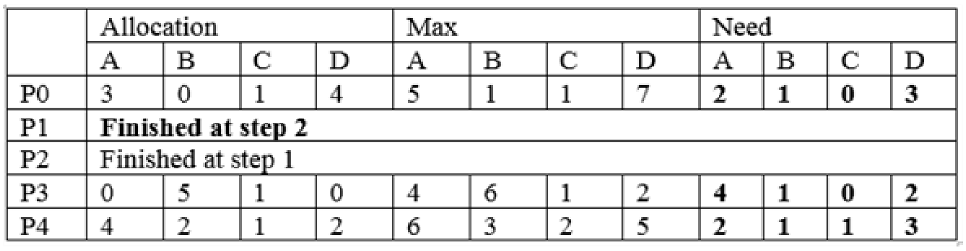


Available			
A	B	C	D
3	4	2	2

Step 2:

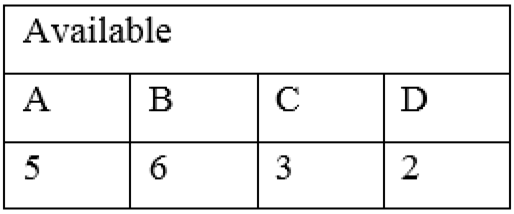
Now, Need of P1 is less than Available, the requirements of P1 can be fulfilled with the available resources and P1 will be completed. After completion, it will release all resources to the Available pool.

So, the current snapshot of the system is,



	Allocation				Max				Need			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	3	0	1	4	5	1	1	7	**2**	**1**	**0**	**3**
P1	**Finished at step 2**											
P2	Finished at step 1											
P3	0	5	1	0	4	6	1	2	**4**	**1**	**0**	**2**
P4	4	2	1	2	6	3	2	5	**2**	**1**	**1**	**3**

And,



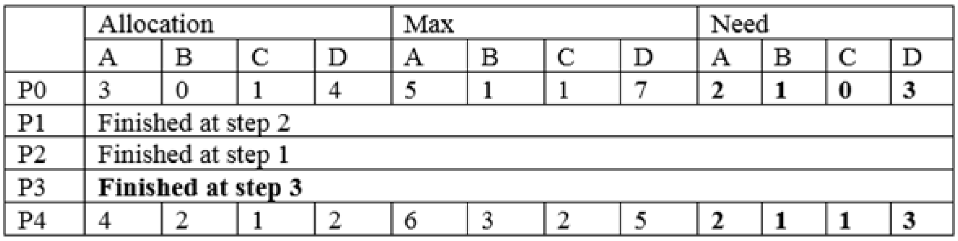
Available			
A	B	C	D
5	6	3	2

Step 3:

Now, Need of P3 is less than Available, the requirements of P3 can be fulfilled with the available resources and P3 will be completed. After completion, it will release all resources to the Available pool.

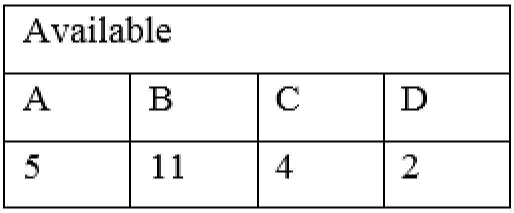
So, the current snapshot of the system is,

Step 2/7



	Allocation				Max				Need			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	3	0	1	4	5	1	1	7	**2**	**1**	**0**	**3**
P1	Finished at step 2											
P2	Finished at step 1											
P3	**Finished at step 3**											
P4	4	2	1	2	6	3	2	5	**2**	**1**	**1**	**3**

And,



Available			
A	B	C	D
5	11	4	2

Step 3/7

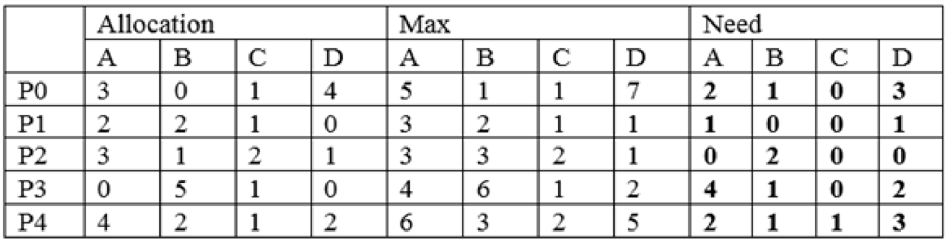
Step 4:

Now, Need of P0 and P4, both are greater than the Available matrix. So, neither of those will be catered with the current available resources.

So, the algorithm will stop and there is no safe sequence in this case. **So the state is unsafe.**

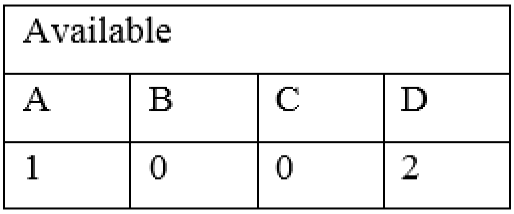
Step 4/7

The snapshot of the system and **Need** matrix is,



	Allocation				Max				Need			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	3	0	1	4	5	1	1	7	**2**	**1**	**0**	**3**
P1	2	2	1	0	3	2	1	1	**1**	**0**	**0**	**1**
P2	3	1	2	1	3	3	2	1	**0**	**2**	**0**	**0**
P3	0	5	1	0	4	6	1	2	**4**	**1**	**0**	**2**
P4	4	2	1	2	6	3	2	5	**2**	**1**	**1**	**3**

The available matrix is,



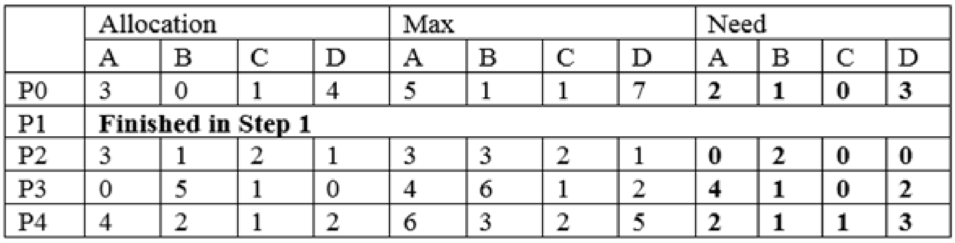
Available			
A	B	C	D
1	0	0	2

So, according to safety algorithm,

Step 1:

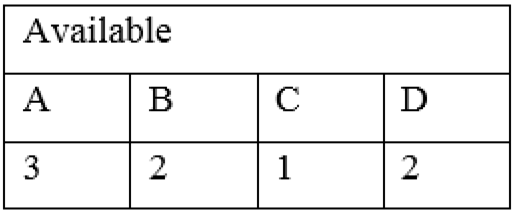
As, Need of P1 is less than Available, the requirements of P1 can be fulfilled with the available resources and P1 will be completed. After completion, it will release all resources to the Available pool.

So, the current snapshot of the system is,



	Allocation				Max				Need			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	3	0	1	4	5	1	1	7	**2**	**1**	**0**	**3**
P1	**Finished in Step 1**											
P2	3	1	2	1	3	3	2	1	**0**	**2**	**0**	**0**
P3	0	5	1	0	4	6	1	2	**4**	**1**	**0**	**2**
P4	4	2	1	2	6	3	2	5	**2**	**1**	**1**	**3**

And,



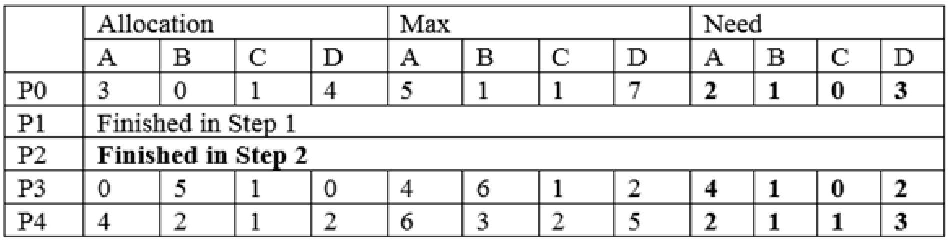
Available			
A	B	C	D
3	2	1	2

Step 5/7

Step 2:

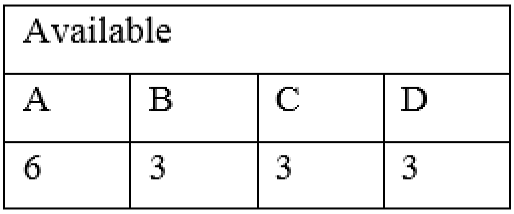
Now, Need of P2 is less than Available, the requirements of P2 can be fulfilled with the available resources and P2 will be completed. After completion, it will release all resources to the Available pool.

So, the current snapshot of the system is,



	Allocation				Max				Need			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	3	0	1	4	5	1	1	7	**2**	**1**	**0**	**3**
P1	Finished in Step 1											
P2	**Finished in Step 2**											
P3	0	5	1	0	4	6	1	2	**4**	**1**	**0**	**2**
P4	4	2	1	2	6	3	2	5	**2**	**1**	**1**	**3**

And,

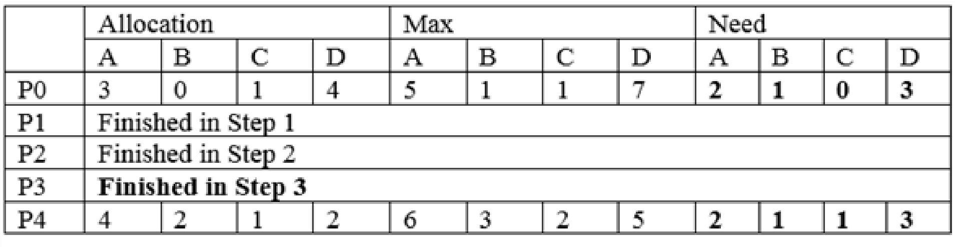


Available			
A	B	C	D
6	3	3	3

Step 3:

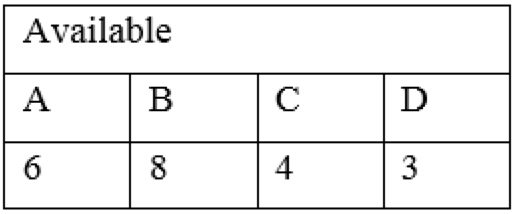
Now, Need of P3 is less than Available, the requirements of P3 can be fulfilled with the available resources and P3 will be completed. After completion, it will release all resources to the Available pool.

So, the current snapshot of the system is,



	Allocation				Max				Need			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	3	0	1	4	5	1	1	7	**2**	**1**	**0**	**3**
P1	Finished in Step 1											
P2	Finished in Step 2											
P3	**Finished in Step 3**											
P4	4	2	1	2	6	3	2	5	**2**	**1**	**1**	**3**

And,



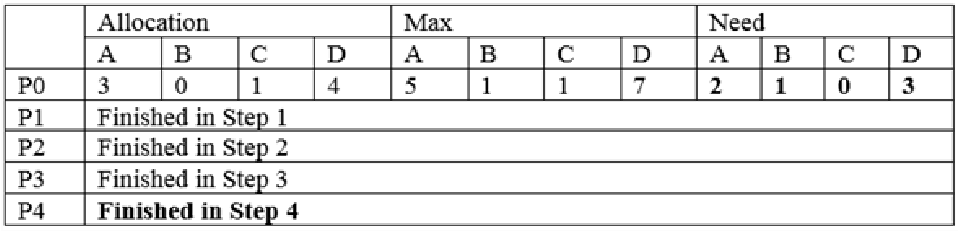
Available			
A	B	C	D
6	8	4	3

Step 6/7

Step 4:

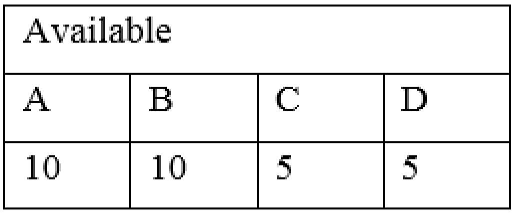
Now, Need of P4 is less than Available, the requirements of P4 can be fulfilled with the available resources and P4 will be completed. After completion, it will release all resources to the Available pool.

So, the current snapshot of the system is,



	Allocation				Max				Need			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	3	0	1	4	5	1	1	7	**2**	**1**	**0**	**3**
P1	Finished in Step 1											
P2	Finished in Step 2											
P3	Finished in Step 3											
P4	**Finished in Step 4**											

And,



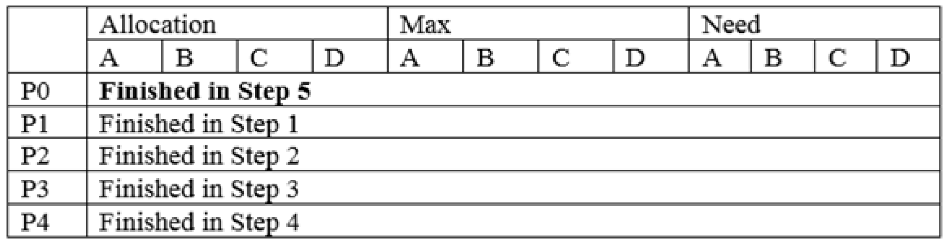
Available			
A	B	C	D
10	10	5	5

Step 7/7

Step 5:

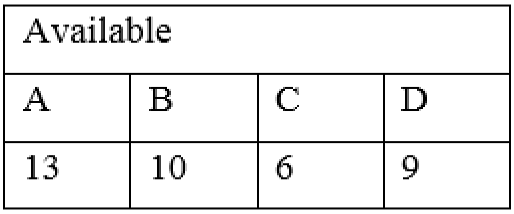
Now, Need of P0 is less than Available, the requirements of P0 can be fulfilled with the available resources and P0 will be completed. After completion, it will release all resources to the Available pool. And all processes will be completed without any deadlock. So the state is safe and the safe sequence will be the sequence of completion of the processes.

So, the current snapshot of the system is,



	Allocation				Max				Need			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	**Finished in Step 5**											
P1	Finished in Step 1											
P2	Finished in Step 2											
P3	Finished in Step 3											
P4	Finished in Step 4											

And,



Available			
A	B	C	D
13	10	6	9

Therefore, the safe sequence is P1 P2 P3 P4 P0.

Chapter 7, Problem 23E

(10)

Step-by-step solution

Show all steps

100% (13 ratings) for this solution

Step 1/16

Illustration of safe state:

In the Banker’s algorithm, four types of matrices are used. Those are:

1. **Available:** An m length vector representing the number of available instances of any resource at a time.

2. **Max:** This is an n to m matrix (n specifying different processes). Each entry implies the maximum instances of a resource type that can be requested by a particular process.

3. **Allocation:** This is also an n to m matrix. Each entry specifies instances of a resource held by a particular process at a time.

4. **Need:** An n to m matrix. Each entry in this matrix implies the number of instances of a resource needed by a particular process at a time. According to safety algorithm, **Need = Max – Allocation**

The snapshot of the system and **Need** matrix is,

	Allocation	Max	**Need**									
	A	B	C	D	A	B	C	D	A	B	C	D
P0	2	0	0	1	4	2	1	2	**2**	**2**	**1**	**1**
P1	3	1	2	1	5	2	5	2	**2**	**1**	**3**	**1**
P2	2	1	0	3	2	3	1	6	**0**	**2**	**1**	**3**
P3	1	3	1	2	1	4	2	4	**0**	**1**	**1**	**2**
P4	1	4	3	2	3	6	6	5	**2**	**2**	**3**	**3**

The Availability matrix is,

Available			
A	B	C	D
3	3	2	1

Step 1:

According to safety algorithm, Work = Available.

The Need for P0 is less than Work (2 <= 3, 2 <= 3, 1<= 2 and 1 <= 1), so, P0 will be allocated the needed resources and will be finished. After finishing of P0, all allocated resources of it will be added to the Work matrix. So, Work = Work + Allocated[i].

The updated matrices are:

	Allocation	Max	**Need**									

	A	B	C	D	A	B	C	D	A	B	C	D
P0	**Finished in step 1**											
P1	3	1	2	1	5	2	5	2	**2**	**1**	**3**	**1**
P2	2	1	0	3	2	3	1	6	**0**	**2**	**1**	**3**
P3	1	3	1	2	1	4	2	4	**0**	**1**	**1**	**2**
P4	1	4	3	2	3	6	6	5	**2**	**2**	**3**	**3**

The updated Work matrix is,

Work			
A	B	C	D
5	3	2	2

Step 2/16

Step 2:

Now, the Need for P3 is less than Work, so, P3 will be allocated the needed resources and it will be finished. After finishing of P3, all allocated resources of it, will be added to the Work matrix. So, the current scenario is,

	Allocation	Max	**Need**									
	A	B	C	D	A	B	C	D	A	B	C	D
P0	Finished in											

	step 1											
P1	3	1	2	1	5	2	5	2	**2**	**1**	**3**	**1**
P2	2	1	0	3	2	3	1	6	**0**	**2**	**1**	**3**
P3	**Finished in step 2**											
P4	1	4	3	2	3	6	6	5	**2**	**2**	**3**	**3**

The Work matrix is,

Work			
A	B	C	D
6	6	3	4

Step 3/16

Step 3:

After allocating P0 and P3, P1, P2 and P4 can be in any sequence as all of their need is less than available.

Now, the Need for P4 is less than Work, so, P4 will be allocated the needed resources and it will be finished. After finishing of P4, all allocated resources of it, will be added to the Work matrix.

Step 4/16

So, the current scenario is,

	Allocation	Max	Need									
	A	B	C	D	A	B	C	D	A	B	C	D

P0	Finished in step 1											
P1	3	1	2	1	5	2	5	2	**2**	**1**	**3**	**1**
P2	2	1	0	3	2	3	1	6	**0**	**2**	**1**	**3**
P3	Finished in step 2											
P4	Finished in step 3											

The Work matrix is,

Work			
A	B	C	D
7	10	6	6

Step 5/16

Step 4:

Now, the Need for P1 is less than Work, so, P1 will be allocated the needed resources and it will be finished. After finishing of P1, all allocated resources of it, will be added to the Work matrix.

So, the current scenario is,

	Allocation	Max	Need									

	A	B	C	D	A	B	C	D	A	B	C	D
P0	Finished in step 1											
P1	Finished in step 4											
P2	2	1	0	3	2	3	1	6	**0**	**2**	**1**	**3**
P3	Finished in step 2											
P4	Finished in step 3											

The Work matrix is,

Work			
A	B	C	D
10	11	8	7

Step 6/16

Step 5:

Now, the Need for P2 is less than Work, so, P2 will be allocated the needed resources and it will be finished. After finishing of P2, all allocated resources of it, will be added to the Work vector.

So, the current scenario is,

	Allocation	Max	Need									
	A	B	C	D	A	B	C	D	A	B	C	D
P0	Finished in step 1											
P1	Finished in step 4											
P2	Finished in step 5											
P3	Finished in step 2											
P4	Finished in step 3											

The Work matrix is,

Work			
A	B	C	D
12	12	8	10

So, all processes will be finished safely and there is no deadlock. The safe sequence will be,

P0 P3 P4 P1 P2.

Step 7/16

For a request from P_1:

The request from P1 for resources < 1, 1, 0, 0> refers to the Request vector for P1 as given below,

Request			
A	B	C	D
1	1	0	0

The snapshot of the system and **Need** matrix is,

	Allocation	Max	Need									
	A	B	C	D	A	B	C	D	A	B	C	D
P0	2	0	0	1	4	2	1	2	**2**	**2**	**1**	**1**
P1	3	1	2	1	5	2	5	2	**2**	**1**	**3**	**1**
P2	2	1	0	3	2	3	1	6	**0**	**2**	**1**	**3**
P3	1	3	1	2	1	4	2	4	**0**	**1**	**1**	**2**
P4	1	4	3	2	3	6	6	5	**2**	**2**	**3**	**3**

The availability matrix is,

Available

Step 8/16

A

B

C

D

3

3

2

1

• The Need of P1 is greater than the Request from P1 and Available matrix is also greater than the Request from P1. This means, P1 has not requested more resources than its maximum limit and currently system is able to satisfy the request.

• The system will pretend to make the allocation and will check whether this allocation leads to safe state or not. If it does not lead to sate state then the system will not confirm the allocation. The changes will be:

	Allocation	Max	Need									
	A	B	C	D	A	B	C	D	A	B	C	D
P0	2	0	0	1	4	2	1	2	**2**	**2**	**1**	**1**
P1	4	2	2	1	5	2	5	2	**1**	**0**	**3**	**1**
P2	2	1	0	3	2	3	1	6	**0**	**2**	**1**	**3**
P3	1	3	1	2	1	4	2	4	**0**	**1**	**1**	**2**
P4	1	4	3	2	3	6	6	5	**2**	**2**	**3**	**3**

The Availability matrix is,

Available			
A	B	C	D

2	2	2	1

Now, check that this allocation results in a safe or unsafe state. According to safety algorithm, Work = Availability.

Step 1:

Now, the Need for P0 is less than Work, so, P0 will be allocated the needed resources and it will be finished. After finishing of P0, all allocated resources of it, will be added to the Work matrix.

The changes will be,

	Allocation	Max	Need									
	A	B	C	D	A	B	C	D	A	B	C	D
P0	**Finished in step 1**											
P1	4	2	2	1	5	2	5	2	**1**	**0**	**3**	**1**
P2	2	1	0	3	2	3	1	6	**0**	**2**	**1**	**3**
P3	1	3	1	2	1	4	2	4	**0**	**1**	**1**	**2**
P4	1	4	3	2	3	6	6	5	**2**	**2**	**3**	**3**

The Work matrix is,

Work			
A	B	C	D
4	2	2	2

Step 9/16

Step 2:

Now, the Need for P3 is less than Work, so, P3 will be allocated the needed resources and it will be finished. After finishing of P3, all allocated resources of it, will be added to the Work matrix.

The changes will be,

	Allocation	Max	Need									
	A	B	C	D	A	B	C	D	A	B	C	D
P0	Finished in step 1											
P1	4	2	2	1	5	2	5	2	**1**	**0**	**3**	**1**
P2	2	1	0	3	2	3	1	6	**0**	**2**	**1**	**3**
P3	**Finished in step 2**											
P4	1	4	3	2	3	6	6	5	**2**	**2**	**3**	**3**

The Work matrix is,

Work			
A	B	C	D
5	5	3	4

Step 10/16

Step 3:

After allocating P0 and P3, P1, P2 and P4 can be in any sequence as all of their need is less than Work.

Now, the Need for P4 is less than Work, so, P4 will be allocated the needed resources and it will be finished. After finishing of P4, all allocated resources of it, will be added to the Work matrix.

The changes will be:

	Allocation	Max	Need									
	A	B	C	D	A	B	C	D	A	B	C	D
P0	Finished in step 1											
P1	4	2	2	1	5	2	5	2	**1**	**0**	**3**	**1**
P2	2	1	0	3	2	3	1	6	**0**	**2**	**1**	**3**
P3	Finished in step 2											

Step 11/16

P4

Finished in step 3

The Work matrix is,

Work			

A	B	C	D
6	9	6	6

Step 12/16

Step 4:

Now, the Need for P1 is less than Work, so, P1 will be allocated the needed resources and it will be finished. After finishing of P1, all allocated resources of it, will be added to the Work matrix. The changes will be:

	Allocation	Max	Need									
	A	B	C	D	A	B	C	D	A	B	C	D
P0	Finished in step 1											
P1	**Finished in step 4**											
P2	2	1	0	3	2	3	1	6	**0**	**2**	**1**	**3**
P3	Finished in step 2											
P4	Finished in step 3											

The Work matrix is,

Work			
A	B	C	D
10	11	8	7

Step 13/16

Step 5:

Now, the Need for P2 is less than Work, so, P2 will be allocated the needed resources and it will be finished. After finishing of P2, all allocated resources of it, will be added to the Work matrix.

The changes will be,

	Allocation	Max	Need									
	A	B	C	D	A	B	C	D	A	B	C	D
P0	Finished in step 1											
P1	Finished in step 4											
P2	**Finished in step 5**											
P3	Finished in											

	step 2											
P4	Finished in step 3											

The Work matrix is,

Work			
A	B	C	D
12	12	8	10

So, all processes are completed without deadlock.

Step 14/16

Hence, the system will be in the safe state if the request of P1 is granted. The system will grant the request.

Step 15/16

c.

For a request from P_4:

The request from P4 for resources <0, 0, 2, 0> refers to the Request vector for P4 as given below,

Request			
A	B	C	D
0	0	2	0

The snapshot of the system and **Need** matrix is,

	Allocation	Max	**Need**									
	A	B	C	D	A	B	C	D	A	B	C	D
P0	2	0	0	1	4	2	1	2	**2**	**2**	**1**	**1**
P1	3	1	2	1	5	2	5	2	**2**	**1**	**3**	**1**
P2	2	1	0	3	2	3	1	6	**0**	**2**	**1**	**3**
P3	1	3	1	2	1	4	2	4	**0**	**1**	**1**	**2**
P4	1	4	3	2	3	6	6	5	**2**	**2**	**3**	**3**

The Availability matrix is,

Available			
A	B	C	D
3	3	2	1

• Now, the Need of P4 is greater than the Request from P4 and Available matrix is also greater than the Request from P4. This means, P4 has not requested more resources than its maximum limit and currently system is able to satisfy the request.

• The system will pretend to make the allocation and will check whether this allocation leads to safe state or not. If it does not lead to sate state then the system will not confirm the allocation. The changes will be,

	Allocation	Max	Need									
	A	B	C	D	A	B	C	D	A	B	C	D
P0	2	0	0	1	4	2	1	2	**2**	**2**	**1**	**1**

P1	3	1	2	1	5	2	5	2	**2**	**1**	**3**	**1**
P2	2	1	0	3	2	3	1	6	**0**	**2**	**1**	**3**
P3	1	3	1	2	1	4	2	4	**0**	**1**	**1**	**2**
P4	1	4	5	2	3	6	6	5	**2**	**2**	**1**	**3**

The Availability matrix is,

Available			
A	B	C	D
3	3	0	1

No other process can be satisfied with the current Work resources.

Step 16/16

So, the system will be in unsafe state if the request of P4 is granted. Hence, the system will reject the request.

Chapter 7, Problem 24E

(0)

Step-by-step solution

Show all steps

Step 1/1

An algorithm that examines the state of the system to determine whether a dead lock has occurred. An algorithm to recover from dead lock

Detection algorithm should be invoked frequently. Resources allocated to deadlock processes will be idle until the deadlock can be broken. The number of processes involved in the dead lock cycle may grow.

If detection algorithm is involved for every resource request, it will incur considerable overhead in computation time. The algorithms whenever invoked, CPU utilization drops below 40%. Hence system throughput cripples due to dead lock and this causes CPU utilizations to drop.

Chapter 7, Problem 25E

(2)

Step-by-step solution

Show all steps

Step 1/2

Pseudocode using semaphores to prevent deadlock:

The following is the pseudocode to prevent deadlock using deadlock.

```
// Semaphore value to cross the bridge
semaphore allowed_to_cross_bridge = 1;
// Function definition
void go_into_bridge()
{
    // Suspend the process of crossing the birdge
    allowed_to_cross_bridge.wait();
}
// Function definition
void way_out_of_bridge()
{
    // Resume the suspended process
    allowed_to_cross_bridge.signal();
}
```

Step 2/2

Explanation:

- The variable “allowed_to_cross_bridge” is represented by process waiting in queue.

- During entry of the threads into the queue, a semaphore is coupled with each thread.

• In the semaphore “go_into_bridge()” function, the process (village) which entered first into processor (bridge) will get executed (crossing the bridge) and followed by that the process next in the queue will get suspended using wait() operation.

• In the semaphore “way_out_of_bridge()” function, the process which was suspended will be awakened and allowed to get executed (crossing the bridge).

Chapter 7, Problem 26E

(0)

Step-by-step solution

Show all steps

Step 1/2

Pseudocode using semaphores to prevent deadlock:

```
// semaphore value to cross the bridge
Semaphore allowed_to_cross_bridge=1;
// function definition
Void go_into_bridge()
{
    // suspended the process of crossing the bridge
    Allowed_to_cross_bridge.wait();
}
//function definition
Void way_out_of_bridge()
{
    // resume the suspended process
    Allow_to_cross_bridge.signal();
}
```

Step 2/2

To the above code add no_waiting_north, no_waiting_south and previous inorder to know which side has crossed most recently. If south side van is crossed recently then north side van is given chance to cross the bridge and south side van will be in waiting state until there is no van on the north side or vice versa.

If a car finishes crossing on the south side and no car waiting on the south side then the car on the north side will go or vice-versa.

```
monitor bridge
{
int no_waiting_north=0;
int no_waiting_south=0;
int on_bridge=0;
ok_to_crossbridge();
{}
int previous=0;
cross_bridge_south()
{
no_waiting_south++;
while( on_bridge || previous==0 && no_waiting_north>0)
{
allowed_to_cross_bridge.wait();
on_bridge=1;
no_waiting_south--;
previous=0;
}
leave_bridge_south()
{
on_bridge=0;
allowed_to_cross_bridge.announce();
}
}

cross_bridge_north()

{

no_waiting_north++;

while( on_bridge || previous==0 && no_waiting_south>0)

{

allowed_to_cross_bridge.wait();

on_bridge=1;

no_waiting_north--;

previous=0;

}

leave_bridge_north()

{

on_bridge=0;

allowed_to_cross_bridge.announce();

}

}
```