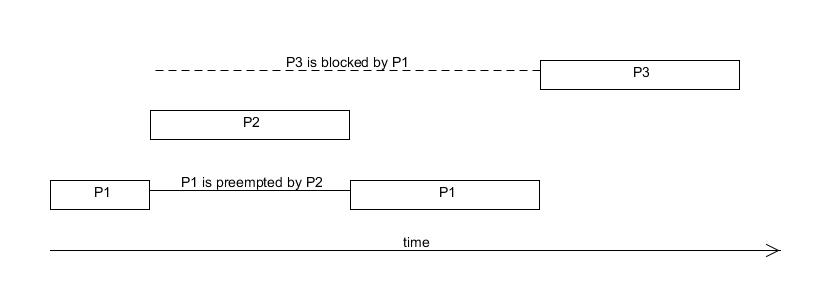
Question 1

Part a:

Priority inversion is situation that arises in preemptive, priority based scheduling system when a high priority process is forced to wait for a lower priority process because the lower priority process has locked some shared resource that is required by the higher priority process. Unbounded priority inversion occurs when the duration that the lower priority process blocks the higher priority process can be extended, possibly indefinitely, by processes with medium priorities preempting the lower priority process while in its critical section. For example consider processes P1, P2 and P3 with priorities 1, 2 and 3 respectively (higher number is higher priority). Say all processes have an absolute computation time of 3 seconds and processes P1 and P3 both need access to some shared resource for the durration of their run. Priority inversion would occur if P1 was released at T=0 and P2 and P3 were released simultaneously at T=1:



Priority inversion has occurred in this example because P3 is forced to wait for the lower priority process P1 due to its ownership of the shared resource. This priority inversion is unbounded because the duration that P3 must wait is extended with the medium priority process P2 preempts P1.

Part b:

Part c:  
  
A basic block is a set of statements that are straight line code, that is they do not contain any branches, loops etc.  
  
In the example program the basic blocks are:  
  
int byte = 0;

And:  
\*input\_buffer = 0;

And:  
\*input\_buffer ++= byte & (1 << n\_bits) - 1;  
input\_bits += 8;  
byte++;

Part d:

Multiprocessor systems can be classified as:

Homogeneous - all processors have identical properties

or

Heterogeneous – not all processors have the same properties

AND

loosely coupled – each processor has its own memory and I/O channels

or

tightly coupled – processors share memory and I/O channels

Part e:  
  
Global placement scheme - it is decided when a task becomes runnable which of the processors it will run on  
  
Partitioned placement scheme – each task is statically assigned to a single processor pre-runtime.

Part f:  
  
Because a task will always inherit the priority of a higher priority resource that it locks, even if it is not blocking any other tasks, blocking relationships do not have to be monitored to calculate the dynamic priorities of tasks in immediate ceiling inheritance, this makes it easier to implement than the original ceiling priority protocol. All blocking of tasks will occur before, a task starts to  
run, so there are less context switches in an immediate ceiling inheritance system than in an original ceiling inheritance system. One disadvantages of the immediate ceiling priority protocol are that dynamic priorities need to be reassigned more frequently.

Question 2

Part a:

|  |  |
| --- | --- |
| Task | Priority |
| T1 | 4 |
| T2 | 2 |
| T3 | 1 |
| T4 | 3 |
| T5 | 5 |

Part b:

In simple priority inheritance blocking time can be determined with the equation:

where usage(k,i) is a 1/0 function that is 1 if and only if some process with priority greater than or equal to Ti uses k and some process with priority less than Ti uses k.

T5 (highest priority so no interference):

T1 :

T4 :

T2 :

T3 :

Part c:

T5 (highest priority so no interference):

T1 :

T4 :

T2 :

T3 :

Question 3

Part a:

Tasks are independent, that is they do no block one another, deadlines are less than periods, the worst case execution times for processes are fixed and known, all tasks are periodic and released with an exact frequency. All priorities are distinct.

Part b:

For any given period of time Z, a process with period T and computation time C, will be released times, and will interfere with processes of lower priority for a total of

Now, let there be a set of tasks x1, x2, x3, …, xn with periods T1, T2, T3, …, Tn and worst case computation times C1, C2, C3, …, Cn respectively. Now consider a process xi:

Say the response time of xi is Ri, then any process with higher priority, say xj, will cause an interference of:

Thus the total interference can be calculated by summing these values for all the processes with higher priority, hp(i):

The response time of xi is the sum of its computation time and interference from other processes, thus:

Using the above formula for interference we arrive at:

Part c:

This equation can be used to determine Ri of a process by calling first assuming that Ri = Ci and then applying the equation recursively until a steady state is reached. For example define the function:

function calcResponseTime(task, respTime) {

newRespTime = task.executionTime;

for( x in task.getAllHigherPriorityTasks() ) {

newRespTime += ceiling( respTime / x.period ) \* x.executionTime;

}

if(newRespTime == respTime) {

return respTime;

}

else {

return calcResponseTime(task, newRespTime);

}

}

Then call:

calcResponseTime(task, task.executionTime);

Part d:

First, use the function to estimate the response time of all the tasks, then test that for each task the estimated response time is less than or equal to its deadline. If this test passes for all the tasks, then the system will meet all deadlines, if it fails for one or more tasks then the system will not meet all its deadlines (without modifications, see below).

Part e:

Fsafds

Question 4:

Priorities are: T1 > T2 > T3 > T5 > T4  (determined from period)

Consider T1:

R1 = 1 (Highest priority, thus no interference)

Consider T2:

Consider T3:

Consider T5:

Consider T4: