Concurrent Systems Exam Solutions

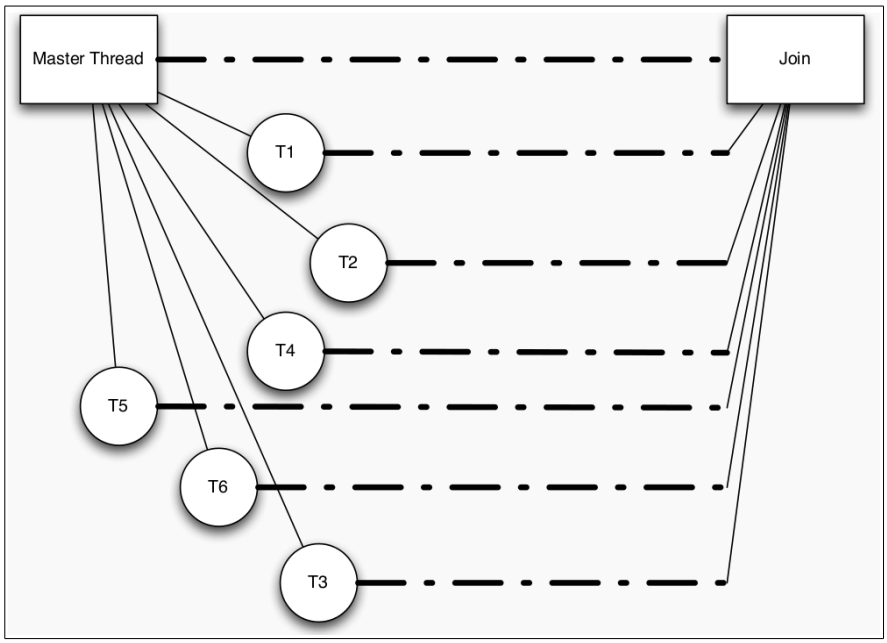
- Two hour Exam
- Answer 2 of 3 Questions
- 1 hour per question

2016 Exam:

Q1 - Threads

a) Write a brief note on threads – what they are, what they are for, how they differ from processes, how they interact and what advantages and shortcomings they have.

A thread is an entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources. Usually a process has only one thread of control. A process may also be made up of multiple threads of execution that execute instructions concurrently.



In multi-core systems threads allow a single process executing to branch off into several threads to perform separate functionalities concurrently and join up once finished. This allows program execution time to be sped up rapidly as it allows for one problem (process) to be broken into several smaller problems (threads) which can be solved simultaneously and combined to produce the one result as desired.

They differ from processes is that threads (of the same process) run in a shared memory space, while processes run in separate memory spaces. A process is an executing instance of a program. A process is always stored in main memory.

Threads can interact with shared resources say for example within the master thread (process). However, as they are executing concurrently there is no definitive way to determine when a thread may access/change the shared resource. This can lead to un-deterministic and unpredictable behavior within a program which is not desirable. To prevent this from happening thread libraries make use of mutexs and semaphores to place locks upon shared resources so that only one thread

can access the desired shared resource at a time. If another thread tries to access the shared resource whilst one thread has access to the resource, it must wait until the current thread has finished using the resource.

b) What does the term massively parallel mean in relation to a parallel program?

In parallel programming an embarrassingly parallel workload/problem is one where little or no effort is needed to separate the problem into a number of parallel tasks. This is often the case where there is little or no dependency or need for communication between those parallel tasks, or for results between them.

An example of this is 3D video rendering handled by a GPU where each frame/pixel can be handled with no interdependency.

c) Write a simple C function to check if a number is prime. Here is the functions signature:

```
int is_prime(int n);
```

```c
#include <stdio.h>
int is_prime(int n){

  int flag = 0;

  for(i=2; i<=n/2; ++i){
        // condition for nonprime number
        if(n%i==0){
              flag=1;
              break;
        }
   }

   if (flag==0)
       printf("%d is a prime number.",n);
   else
       printf("%d is not a prime number.",n);

   return 0;
}
```

Using this function, write a C program to count the number of prime numbers in the interval 1… n

```
8    void *primesThreadFunction(void *args){
9
10     int n = *((int *) args);
11
12     int flag = 0;
13     int i;
14
15     for(i=2; i<=n/2; ++i){
16       if(n%i==0){
17         flag=1;
18         break;
19       }
20     }
21
22     if (flag==0){
23       primes_count ++;
24       printf("%d is a prime number.",n);
25     }
26     else
27     printf("%d is not a prime number.",n);
28
29     pthread_exit(NULL);
30   }
```

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

  int n_threads = atoi(argv[0]);
  pthread_t prime_threads[n_threads];

  int t;
  int returnCode;

  //Create n threads and let them calculate primes
  for(t=1;t<n_threads;t++){

    returnCode = pthread_create(&prime_threads[t], NULL,
      primesThreadFunction, (void *)t);
      if (returnCode) {
        printf("ERROR return code from pthread_create() : %d\n",returnCode);
        exit(-1);
      }
    }

    //Wait for all threads to exit
    for(t=0;t<=n_threads; t++)
    pthread_join(prime_threads[t], NULL);

    printf("Successfully exited all threads!\n");
    printf("Number of primes between 1 and %d = %d\n", n_threads, primes_count);
    return(0);
  }
```

d) With an eye to efficiency and core utilization, what kinds of problems would you see with this program? How could you fix them?

As n gets larger as we iterate over our interval the probability of n becoming a prime number becomes significantly smaller, thus leading to an exponential increase in unnecessary threads being created. This is fine for small numbers of n however as n becomes significantly large we end up utilizing all of our CPU's resources running an extremely large number of threads.

To reduce this, we could first pre-process our value of n within the master thread (mainline) ensuring first that the number is not divisible by 2/3/4/5…. up to whatever number we see fit. This will as a result, significantly reduce the number of threads created and reduce the pressure on the CPU's resources.

Q2 – SPIN & Promela

a) SPIN is said to be a Model Checker. What exactly does that mean? Why might it be useful.

SPIN and Promela, when used together allow you to model a system or program quite simply whilst also allowing you to conduct in-depth performance testing of a given system.

By using Promela and SPIN you are able to fully ensure that a system is designed safely and is free of any deficiencies that might lead to the occurance of deadlock, livelock or starvation. Through SPINs's verification mode you can verify that even the most complicated of systems is free from any of the above problems.

“Something bad never happens” - SPIN tries to disprove this statement by finding a path to a case when something bad happens.

“Something good always happens eventually” - SPIN tries to find an infinite loop in which good doesn’t happen

b) In the context of Promela, SPIN, and concurrent processes, define scenario. Give a brief example.

With regards to the above, there can be multiple different routes that a concurrent process can take based on the decision variables and choices made throughout the execution of its life cycle. Each of these different routes produce what can be considered a different scenario. Variables will have unique values and states will be different with regards to each individual scenario.

An example of this is a deadlock scenario within the dining philosophers problem, this occurs when all philosophers have picked up 1 fork and are all waiting on each philosopher left/right to place down their fork. Thus causing the system to be stuck in limbo. This can be considered a given scenario as the states within the system contain unique values which are unique to this situation.

c) Develop a Promela model of the well-known Dining Philophers problem:

```
#define NUM_PHIL 5

int forks[NUM_PHIL] = -1;
bool pthinking[NUM_PHIL] = false;
bool phungry[NUM_PHIL] = false;
bool peating[NUM_PHIL] = false;

init{
  atomic{
    int i = 0;

    //Create each philosopher, run P
    do
    :: i < NUM_PHIL ->
      run P(i);
      i++;

    :: else ->
      break;
    od;
  }
}

```

```
24    //Process to represent a philosopher
25    proctype P(int i){
26
27      //Right fork at index i
28      int right = i;
29      //Left fork at (i+1)%NUM_PHIL
30      int left = (i+1)%NUM_PHIL;
31
32      //Think state
33      think:
34        atomic{
35          peating[i] = false;
36          pthinking[i] = true;
37        };
38
39      //Hungry state
40      hungry:
41        atomic{
42          pthinking[i] = false;
43          phungry[i] = false;
44        };
45
46        if
47        :: skip;
48          //Attempt to pickup left fork
49          atomic{ forks[left] == -1 -> forks[left] = i};
50          //Attempt to pickup right fork
51          atomic{ forks[right] == -1 -> forks[right] = i};
52
53        :: skip;
54          //Attempt to pickup right fork
55          atomic{ forks[right] == -1 -> forks[right] = i};
56          //Attempt to pickup left fork
57          atomic{ forks[left] == -1 -> forks[left] = i};
58        fi;
59
60      //Eating state
61      eating:
62        atomic{
63          phungry[i] = false;
64          peating[i] = true;
65        };
66
67      //Finished eating state
68      done:
69        forks[right] = -1;
70        forks[left] = -1;
71        goto think;
72    }
```

d) What do you add to the Promela model to detect deadlock, livelock and starvation?

1. Deadlock:

   To check for deadlock just pass the above model of the dining philosophers problem through SPIN's Model Checking tool. This runs the above system a defined number of times, checking to see if deadlock or any other issue ever occurs.

```
spin -a philo_nice_deadlock.prm
gcc -O2 -g -o pan pan.c -DSAFETY
./pan -m1000000
spin -p -t philo_nice_deadlock.prm
```

```
#processes: 6
                forks[0] = 4
                forks[1] = 0
                forks[2] = 1
                forks[3] = 2
                forks[4] = 3
                pthinking[0] = 0
                pthinking[1] = 0
                pthinking[2] = 0
                pthinking[3] = 0
                pthinking[4] = 0
                phungry[0] = 1
                phungry[1] = 1
                phungry[2] = 1
                phungry[3] = 1
                phungry[4] = 1
                peating[0] = 0
                peating[1] = 0
                peating[2] = 0
                peating[3] = 0
                peating[4] = 0
```

   Here SPIN shows us that each fork on the table is being held by an different philosopher and also that each philosopher is hungry and as a result is waiting on a fork to become free from one of the other philosophers. As a result the system can be considered to be stuck in deadlock.

2. Livelock:

   Livelock occurs when for example all Philosophers decide if you don't manage to pick up a second fork within 10 minutes you must put down your current fork and wait 10 minutes before trying again. However, if all Philosophers somehow get to the stage where the 10 minute intervals are the exact same, Livelock occurs.

   To check for this you would have to introduce a waiting period to your model and then run it through SPIN to ensure that livelock does not occur.

3. Starvation:

Starvation could occur within this system if say Philosopher A becomes hungry and tries to pick up for 1. At the same time Philosopher B becomes hungry and manages to get a hold of fork 1 and 2 first. Philosopher B continues eating, finishes and starts thinking again. However, Philosopher B immediately becomes hungry again and grabs the forks before Philosopher A. This happens indefinitely thus starving Philosopher B to death.

To check for this we introduce what is known as a never-claim and we use this with SPIN to ensure that this never-claim never becomes true at any stage.

```
2    never  {    /* !([](phungry -> <>peating)) */
3    T0_init:
4      do
5      :: (! ((peating[0])) && (phungry[0])) -> goto accept_S4
6      :: (1) -> goto T0_init
7      od;
8    accept_S4:
9      do
10     :: (! ((peating[0]))) -> goto accept_S4
11     od;
12   }
```

```
38     ** Check for starvation (use neverclaim)**
39     $ spin -a -N neverclaim philo_no_deadlock.prm
40     $ gcc -O2 -g -o pan pan.c
41     $ ./pan -a -n -m1000000
```

```
  <<<<<START OF CYCLE>>>>>
3220:   proc  5 (P:1) philo_no_deadlock.prm:49 (state 8)        [((forks[left]==-(1)))]
3220:   proc  5 (P:1) philo_no_deadlock.prm:49 (state 9)        [forks[left] = i]
3222:   proc  5 (P:1) philo_no_deadlock.prm:51 (state 11)       [((forks[right]==-(1)))]
3222:   proc  5 (P:1) philo_no_deadlock.prm:51 (state 12)       [forks[right] = i]
3224:   proc  5 (P:1) philo_no_deadlock.prm:62 (state 23)       [phungry[i] = 0]
3224:   proc  5 (P:1) philo_no_deadlock.prm:63 (state 24)       [peating[i] = 1]
3226:   proc  5 (P:1) philo_no_deadlock.prm:67 (state 26)       [forks[right] = -(1)]
3228:   proc  5 (P:1) philo_no_deadlock.prm:68 (state 27)       [forks[left] = -(1)]
3230:   proc  5 (P:1) philo_no_deadlock.prm:34 (state 1)        [peating[i] = 0]
3230:   proc  5 (P:1) philo_no_deadlock.prm:35 (state 2)        [pthinking[i] = 1]
3232:   proc  5 (P:1) philo_no_deadlock.prm:40 (state 4)        [pthinking[i] = 0]
3232:   proc  5 (P:1) philo_no_deadlock.prm:41 (state 5)        [phungry[i] = 1]
3234:   proc  5 (P:1) philo_no_deadlock.prm:47 (state 7)        [((left<right))]
spin: trail ends after 3234 steps
```

Philosopher 1 is hungry, and tries to pick up a fork. However at the same time Philosopher 5 tries to pick up the same shared fork which he manages to do. He continues eating, finishes, starts thinking again and immediately becomes hungry again and picks up the fork before Philosopher 1 can i.e starving him to death.