TRINITY COLLEGE DUBLIN
THE UNIVERSITY OF DUBLIN

Faculty of Engineering, Mathematics and Science

School of Computer Science & Statistics

Integrated Computer Science Programme **Trinity Term 2015**
Year 2 Annual Examinations

Concurrent Systems and Operating Systems

Monday 27 April 2015 **Luce Upper** **09:30 – 11:30**

Dr Mike Brady

Instructions to Candidates:
Attempt **two** questions. All questions carry equal marks. Each question is scored out of a total of 20 marks.

You may not start this examination until you are instructed to do so by the Invigilator.

Materials permitted for this examination:
A two-page document, entitled *"Pthread Types and Function Prototypes"* accompanies this examination paper.

Non-programmable calculators are permitted for this examination — please indicate the make and model of your calculator on each answer book used.

1. (a) Suppose you were asked to "parallelise" an existing application; that is, to take an existing application, written for a single processor, and to make it run as quickly as possible on a computer with many cores.

   Imagine you have access to all the the programs, algorithms and original design documents and would be allowed do anything from altering the existing program to building a completely new application that was functionally identical.

   To maximise performance, there would be certain features you would look for, other features or circumstances you would try to avoid and then there would be certain things that might be unavoidable.

   How would you go about redesigning or rebuilding the application to make it run faster on a parallel architecture — what techniques would you employ, and what estimate of how much faster your redesigned application would run could you give?

   [8 marks]

   (b) Give a brief overview of the producer-consumer problem, and indicate what role condition variables play in it. [2 marks]

   (c) Given a buffer that uses a mutex for controlling access and two condition variables designed to signal when the buffer is non-empty and when it is non-full, write producer and consumer functions that run endlessly, producing or consuming items from the buffer, ensuring there is neither underflow nor overflow, using condition variables as appropriate in order to minimise processor usage without affecting speed of response.

   It is not necessary to implement a real buffer; instead, simulate a buffer with a single integer variable called buffer. Initialise the buffer's value to zero, indicating the buffer is empty. A producer adds one to the buffer if not full, a consumer subtracts 1 from the buffer if it's not empty. Assume that if the buffer is full when its value is 20. [10 marks]

2. (a) One of the big challenges of building computer systems is to ensure that they perform properly; that is, they work as intended, are reliable and efficient. Explain why this is particularly difficult with parallel systems and hence explain the significance of systems like Promela and SPIN. [8 marks]

(b) The diagram below is a simple parallel system consisting of two processes and intended to solve the so-called "critical section" problem such that only one process may be in its critical section at any time. That is, p4 and q4 can not both be eligible for execution at the same time. The design is faulty.

Critical Section Proposed Solution			
boolean wantp ← false, wantq ← false			
p		q	
loop forever		loop forever	
p1:	non-critical section	q1:	non-critical section
p2:	await wantq == false	q2:	await wantp == false
p3:	wantp ← true	q3:	wantq ← true
p4:	critical section	q4:	critical section
p5:	wantp ← false	q5:	wantq ← false

(i) Write the Promela code for this system. [4 marks]

(ii) Explain why the design is faulty, i.e. explain why it does not correctly implement the critical section. Use a scenario to show this if possible.

[4 marks]

(iii) Add the Promela core necessary to be in a position to prove that the system is faulty. With this extra code, how would you then use SPIN to find the proof? (Hint – when a process is in its critical section, the total number of processes in their critical sections must be exactly one.)

[4 marks]

3. (a) It is often said that operating systems have two roles — one is as a provider of a uniform environment for application programs and the other is as a manager of computer resources. Describe operating systems in these terms, and give an account of the main function of each component or section of the operating system.

[5 marks]

(b) Draw and explain the so-called "life-cycle" of a process. Compare and contrast pre-emptive and cooperative multitasking, highlighting the features, including the strengths and weaknesses of each. [5 marks]

(c) Draw diagrams and explain how the processes listed below would be scheduled on a processor where the scheduler is using (a) First-Come-First-Server (FCFS) without preemption and (b) Round Robin with preemption at 2 ms intervals.

Process Name	Start Time (ms)	Processor Time (ms)
a	0	10
b	2	6
c	3	5
d	6	8

[5 marks]

(d) Hence or otherwise, calculate the start delay (the time a process waits to start running), the overall run time (the time from when the process first gets onto the processor to when it is finished) and the turnaround time (the time from when the process is first ready to run until it is finished). Comment on the differences between the results for the two scheduling policies. [5 marks]

Pthread Types and Function Prototypes

Definitions

```
pthread_t;  //this is the type of a pthread;
pthread_mutex_t; //this is the type of a mutex;
pthread_cond_t; // this is the type of a condition variable
```

Create a thread

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                   void *(*thread_function)(void *), void *arg);
```

Static Initialisation

Mutexes and condition variables can be initialized to default values using the `INITIALIZER` macros. For example:

```
pthread_mutex_t count_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t count_cond = PTHREAD_COND_INITIALIZER;
```

Dynamic Initialisation

Mutexes, condition variables and semaphores can be initialized dynamically using the following calls:

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                   void *(*thread_function)(void *), void *arg);
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *mutexattr);
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *cond_attr);
int pthread_attr_init(pthread_attr_t *attr);
```

Deletion

```
int   pthread_mutex_destroy(pthread_mutex_t *);
int   pthread_cond_destroy(pthread_cond_t *);
```

Thread Function

The thread_function prototype would look like this:

```
void *thread_function(void *args);
```

Thread Exit & Join

```
void  pthread_exit(void *); // exit the thread i.e. terminate the thread
int   pthread_join(pthread_t, void **); // wait for the thread to exit.
```

Mutex locking and unlocking

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Pthread Condition Variables

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Semaphores

```
sem_t; // this is the type of a semaphore
int sem_init(sem_t *sem, int pshared, unsigned int value); // pshared = 0 for semaphores
int sem_wait(sem_t *sp); // wait
int sem_post(sem_t *sp); // post
int sem_destroy(sem_t * sem); // delete
```