Coláiste na Tríonóide, Baile Átha Cliath
Trinity College Dublin
Ollscoil Átha Cliath | The University of Dublin

Faculty of Engineering, Mathematics and Science

School of Computer Science & Statistics

Integrated Computer Science **Trinity Term 2018**
Year 2 Annual Examinations

Concurrent Systems and Operating Systems

Tuesday 15 May 2018 **Goldsmith Hall** **14:00 – 16:00**

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) The `pthreads` library is a toolkit for writing parallel programs. What tools does it provide? [3 marks]

   (b) What is the principal difference between a *thread* and a *process*? [2 marks]

   (c) What is the Dining Philosophers Problem, and why is it of interest in the context of concurrent systems? [2 marks]

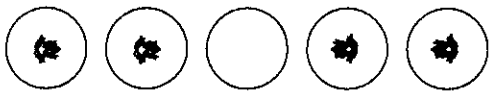
   (d) Write a simulation of the Dining Philosophers Problem, with five philosophers and five forks, implementing each philosopher as a `pthread`. Each thread should be able to print out its state and ID (an index number between 0 and 4). [10 marks]

   (e) Explain how you would test the operation of the program for deadlock, livelock and starvation. [3 marks]

2. (a) SPIN is used in the formal verification of parallel systems. What does that mean, and why are SPIN and Promela different from conventional programming languages and tools? [2 marks]

   (b) Explain the terms *deadlock*, *livelock* and *starvation*. [2 marks]

   (c) Consider the following "jumping frogs" example:



- Each circle represents a stone which may have a frog resting on it or which may be vacant. The initial situation is depicted in the diagram.
- The desired end state is where the two rightmost frogs are on the leftmost stones and the two leftmost frogs are on the rightmost stones.
- A frog can move to an adjacent stone if it is vacant.
- A frog can hop over an adjacent occupied stone to the next one if that one is vacant.
- A frog can move 'forwards' only. Thus, the two frogs on the left of the diagram can only move to the right; similarly the two frogs on the right of the diagram can only move to the left.
- No other moves are possible.

(i) Draw a diagram showing a sequence of moves that results in the desired end state. [4 marks]

(ii) Draw a diagram showing a sequence of moves that results in a deadlock state. [2 marks]

(iii) Write the Promela code to simulate the behaviour of a frog. The code should emulate either kind of frog – one that can move only to the right or one that can move only to the left. [4 marks]

(iv) Now, if the frogs could move 'backwards' as well as forwards, the system would suffer from the possibility of livelock. Show how you would use SPIN to verify if livelock existed. [6 marks]

3. (a) What is the difference between a *virtual address* and a *physical address*?

[2 marks]

(b) Explain how the memory management part of a regular operating system implements virtual memory. [6 marks]

(c) Virtual memory is normally not used in applications where assured real-time response is required. Why is that? [2 marks]

(d) With reasonable values for access times to main memory (10 nanoseconds, i.e. $10 * 10^{-9}$ seconds) and backing store (10 milliseconds, i.e. $10 * 10^{-3}$ seconds), and with a page fault ratio of 1 in a million, calculate the average virtual memory access time of a system. [2 marks]

(e) With the same main memory and disk access times as above, calculate the page fault ratio necessary to give an overall average virtual memory access time of 25 nanoseconds. [2 marks]

(f) Explain, with the aid of a diagram, the life cycle of a thread or a process.

[2 marks]

(g) Explain the operation of a (pre-emptive) round-robin scheduler. What are its advantages and disadvantage compared to other scheduling arrangements, such as priority-based or non-preemptive scheduling? [4 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
```