Concurrent Systems Exam Solutions

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

2017 Exam:

Q1 - Pthreads

a) The pthreads library is a toolkit for writing parallel programs. What tools does it provide?

- Thread Creation & Deletion: The pthreads library provides the functionality to allow you to create a series of threads that can work concurrently acting on various functions (processes) that are programmed by the user. These threads can also be exited and (cleanly*) deleted using the pthreads library.

- Mutex Locking & Unlocking: The pthreads library provides the functionality to make use of mutex locks which are used to protect shared resources between concurrent programs. By using mutex locks it allows programs to ensure they are the only ones using a shared resource at a given time. This ensures that concurrent programs run correctly and don't access shared resources at the same time.

- Condition Variables: The pthreads library provides the functionality to make use of condition variables which allow you to create a somewhat deterministic program execution in multi-threaded programs. It allows for processes to wait upon a given variable, and also allows process to broadcast condition variables to other concurrent processes.

- Semaphores: A semaphore is a variable or abstract data type used to control access to a common resource by multiple processes in a concurrent system such as a multitasking operating system.

b) What is the principal difference between a thread and a process.

Both processes and threads run independent sequences of execution. The typical difference is that threads (of the same process) run in a shared memory space, while processes run in separate memory spaces.

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.

A process is an executing instance of a program. A process is always stored in main memory. Several processes may be associated with the same program.

c) Explain the operation of the pthread library functions pthread_mutex_lock and pthread_mutex_join.

- *pthread_mutex_lock:* This is a method which grabs access to a mutex and attempts to lock it. If it is successfully locked the mutex is then inaccessible until it is finally unlocked again by calling the method *pthread_mutex_unlock*. The reason for this functionality means that a parallel system operating on a shared resources can unsure that the resource is not acted upon simultaneously, especially when processes are dependent on the shared resource.

- *pthread_mutex_join:* This is a method which allows the master thread (main program running multiple separate pthreads) to join the existing threads up once they have finished execution. This ensures that the programs resources aren't wasted running idle threads idefinitely.

d) Imagine you have inputted a large color picture, say a JPEG, and have expanded it into a two-dimensional array of pixels. Ten threads, each executing a copy of a thread function, cooperate to create a grey-level version of the picture in another array.

- *Write the thread function*
- *Write the function that creates the threads*

A) Main Function (Creating Threads)

```
48  int main(){
49
50    //Initialise mutex
51    mutex = (pthread_mutex_t)PTHREAD_MUTEX_INITIALIZER;
52    pthread_t greyscale_threads[NUM_THREADS];
53    int t;
54
55    //Create 10 threads and let them calculate greyscale
56    for(t=0;t<NUM_THREADS;t++){
57
58      returnCode = pthread_create(&greyscale_threads[t], NULL,
59                                  greyscaleThreadFunction, (void *)t);
60      if (returnCode) {
61        printf("ERROR return code from pthread_create() : %d\n",returnCode);
62        exit(-1);
63      }
64    }
65
66    //Wait for all threads to exit
67    for(t=0;t<=NUM_THREADS; t++)
68          pthread_join(greyscale_threads[t], NULL);
69
70    printf("Successfully exited all threads!\n");
71    return(0;)
72
73  }
74
```

B) Thread Function

```
#include <pthread.h>
#include <stdlib.h>
#include <string.h>

#define IMAGE_WIDTH 50
#define IMAGE_HEIGHT 50
#define NUM_THREADS 10

//image[ROW][COLUMN]
color colourImage[IMAGE_WIDTH][IMAGE_HEIGHT];
int greyscaleImage[IMAGE_WIDTH][IMAGE_HEIGHT];

int checkedIndices[IMAGE_WIDTH][IMAGE_HEIGHT];
pthread_mutex_t mutex;

typedef struct color {
    int R;
    int G;
    int B;
} color_t;

void *greyscaleThreadFunction(void *args){

  int i;
  int j;

  //Iterate down each row
  for(i=0;i<IMAGE_WIDTH;i++){
    //Iterate accross each column within current row
    for(j=0;j<IMAGE_HEIGHT;i++){
      //Get the pixel at image[i][j]
      pthread_mutex_lock(&print_state.mutex);

      if(!checkedIndices[i][j]){
        color_t pixel = colourImage[i][j];
        int greyscaleVal = (pixel.R + pixel.G + pixel.B)/3;
        greyscaleImage[i][j] = greyscaleVal;
        checkedIndices[i][j] = 1;
      }

      pthread_mutex_unlock(&print_state.mutex);
    }
  }
  pthread_exit(NULL);
}
```

Q2 – SPIN & Promela

a) SPIN and its associated modelling language Promela are radically different from most software development tools. Explain why this is so.

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) SPIN has two principal modes of operation – interpretation and verification. Explain the difference between them, and explain their importance.

Interpretation mode is when SPIN takes a basic overview of the model and executes it. It finds any immediate flaws and notifies them for one given scenario.

However, verification allows us to verify claims that we ourselves make. For example we can claim the statements above - "Something bad never happens" and "Something good always happens eventually".

Verification allows us to make the following constructs:

- Basic Assertions
- End-State Labels
- Progress-State Labels
- Accept-State Labels
- Never Claims
- Trace Assertions

c) Explain the terms deadlock, livelock and starvation.

- *Deadlock:* In an operating system a deadlock occurs when a process or a thread enters a waiting state because a requested resource is held by another waiting process, which in turn in waiting for another waiting process etc. If a process is unable to change its state indefinitely then the system is said to be in deadlock.

- *Livelock:* Livelock is similar to deadlock however, the states of the processes involved in the livelock are constantly changing with regard to one another, none progressing.

- *Starvation:* Starvation is a problem encountered in concurrent computing where a process is perpetually denied necessary resources to process its work.

d) What is the Dining Philosophers problem and why is it of interest in the context of concurrent systems?

The Dining Philophers problem is when several philosophers are sitting around a table about to have something to eat. As philosophers do, they can only either think, become hungry and eat. However, the issue is that there are not enough forks for the philosophers so each philosopher will have to share a fork with his neighbour.

This is of interest in the context of concurrent systems as it provides quite a graphical and easy to understand representation of the issues that can occur within a concurrent system such as that of deadlock, livelock and concurrency.

For example, when a philosopher becomes hungry he will then attempt to eat. For a philosopher to be able to eat he needs to grab hold of two forks (both of which are shared with a neighbour). If he is unable to grab a fork he waits until one has become available. Also, if he manages to get hold of one fork he retains this fork and waits until the other fork has become available.

This leads to both deadlock, livelock and starvation within the system.

e) Write a Promela description of a reduced version of the Dining Philosophers Problem which has just two identical philophers at a table for two and with just two forks.

- First initialize data structures to hold information about your philosophers and also the forks.
- Then create an init() process to run each philosopher process.

```
#define NUM_PHIL 2

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;
  }
}
```

- Then create the actual process to model the philosophers themselves.

```
//Process to represent a philosopher
proctype P(int i){

  //Right fork at index i
  int right = i;
  //Left fork at (i+1)%NUM_PHIL
  int left = (i+1)%NUM_PHIL;

  //Think state
  think:
    atomic{
      peating[i] = false;
      pthinking[i] = true;
    };

  //Hungry state
  hungry:
    atomic{
      pthinking[i] = false;
      phungry[i] = false;
    };

    if
    :: skip;
      //Attempt to pickup left fork
      atomic{ forks[left] == -1 -> forks[left] = i};
      //Attempt to pickup right fork
      atomic{ forks[right] == -1 -> forks[right] = i};

    :: skip;
      //Attempt to pickup right fork
      atomic{ forks[right] == -1 -> forks[right] = i};
      //Attempt to pickup left fork
      atomic{ forks[left] == -1 -> forks[left] = i};
    fi;

  //Eating state
  eating:
    atomic{
      phungry[i] = false;
      peating[i] = true;
    };

  //Finished eating state
  done:
    forks[right] = -1;
    forks[left] = -1;
    goto think;
}
```

f) Show how the system you describe will suffer from some combination of deadlock, livelock and/or starvation.

1. Deadlock: Both of the philosophers seated at the table will be sharing the same resources i.e forks. Therefore there could exist a state where both philosophers are hungry and have instantaneously grabbed a fork, leaving no more forks on the table. This means that both are caught in deadlock waiting on a fork to be released from the other.

2. Livelock: Livelock could occur within this system if the philosophers decide that if they have not managed to get a second fork within 60 seconds they will place down the fork they currently hold. However, if it occurs that both philosophers 60 second intervals are in sync the system will then be in livelock as there could arise a situation whereas they will never be able to get a fork and will continue the loop indefinitely.

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.

Q3 – Virtual Memory

a) What is the difference between a virtual address and a physical address?

- Physical Address: Actual address on RAM
- Virtual Address: Address generated by the processor

Physical addressing means that your program actually knows the real layout of RAM. When you access a variable address at 0x8746B3, thats where it's really stored in the physical RAM chips.

With virtual addressing, all application memory accesses go to a page table, which then maps from the virtual to the physical address. So every application has its own "private" address space, and no program can read or write to another programs memory.

Virtual addressing protects programs from crashing each other through poor pointer manipulation. Because each program has its own distinct virtual memory set, no program can read another's data.

b) Explain how the memory management part of a regular operating system implements virtual memory.

Each process within an operating system is given its own virtual address space which is divided into fixed-size pages (e.g 4KB or 212B). Programs then refer to addresses in their own virtual memory space (virtual addresses). The operating system carries out an "on the fly" conversion of these virtual addresses into physical addresses, which are applied to physical memory.

This above functionality is usually carried out by the Memory Management Unit (MMU) which is located within the CPU. In a multiprogramming environment, process (and OS) must share physical memory. As a result programs must be paged in and out of memory. Programmers have no way of knowing where in physical memory their process will be located, and processes won't be continuous. Virtual memory gives each process its own contiguous virtual memory space within

which the program works. Each process has access to its own virtual memory space, which should never map to the physical memory of another process.

Each process has its own page table which is used to map virtual pages to physical page frames. To translate a virtual page number to a physical page frame number :

1. Use the virtual page number as an index to the page table
2. Read the physical page frame number from the table

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

Virtual memory is not used in applications where assured real-time response is required as there is no definitive way of determining where in memory the process' respective data may be stored. This means that as a result there is no way of calculating the time of a real-time response for a given process.

The time required to read/write data within a process that uses virtual memory may vary greatly depending on the allocation of virtual memory by the Memory Management Unit (MMU).

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

Memory Access - $10 * 10^{-9}$ seconds
Backing Store Access - $10 * 10^{-3}$ seconds
Page Fault Probability - 0.0001

10,000 accesses will take:

$9{,}999*10*10^{-9} + 10*10^{-3}$
$= 99.9*10^{-6} + 10{,}000*10^{-6}$

$= 10{,}099.9\ \mu s$ / 10,000 accesses
$= 1.00999\ \mu s$ / access

Virtual Memory Performance critically dependent on extremely low page fault rates!
Looking for rates of 0.001% – 0.00001%

d) Explain, with diagrams, the operation of the scheduler in a conventional operating system. In your answer, explain the concepts of fairness and explain how it might be managed in a scheduler.

On a single processor system only one process can be executed at a time. A process is typically executed until it is required to wait for some event e.g I/O, timer, resource. When a process' execution is stalled we want to maximize CPU utilization by executing another process on the CPU.

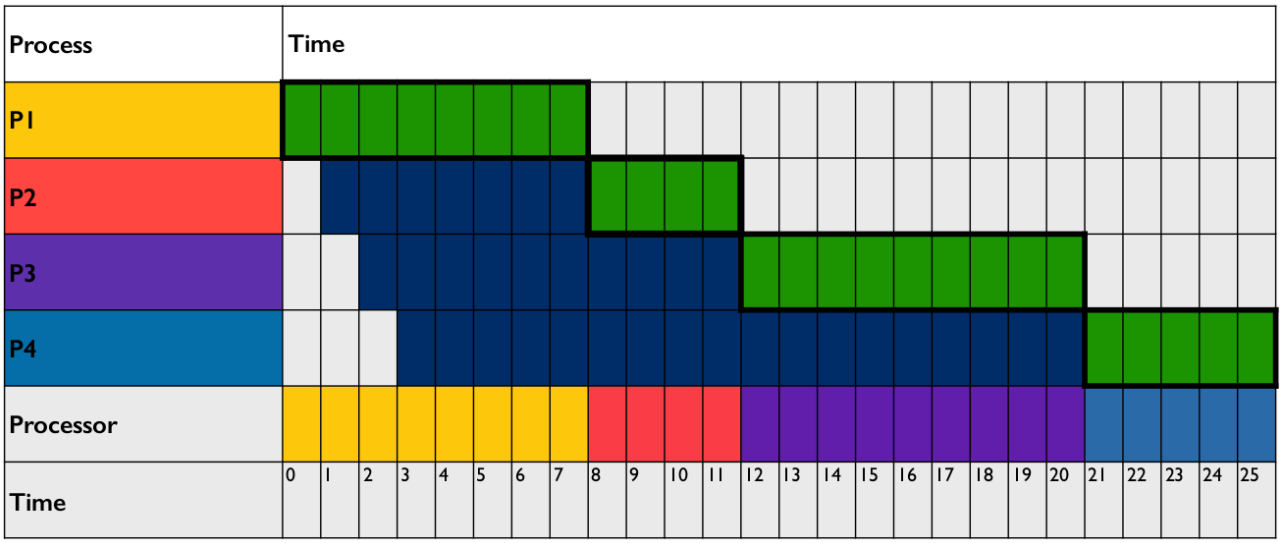
When the CPU becomes idle, the operating system must select another process to execute. This is the role of the short-term scheduler / CPU scheduler. Candidate processes for scheduling are those programs that are in memory and are ready for execution. The ready pool is not necessarily a FIFO queue, it depends on the implementation of the scheduler.

- Co-Operative Scheduling: Process will run until finished (no time limit)
- Pre-Emptive Scheduling: Process will run for a given time.

	First-Come-First-Served (FCFS)	Shortest Job First	Priority	Round Robin
Non-Preemptive	✔	Fragile	Fragile	This is FCFS
Pre-Emptive	*What could this be?*	✔	✔	✔

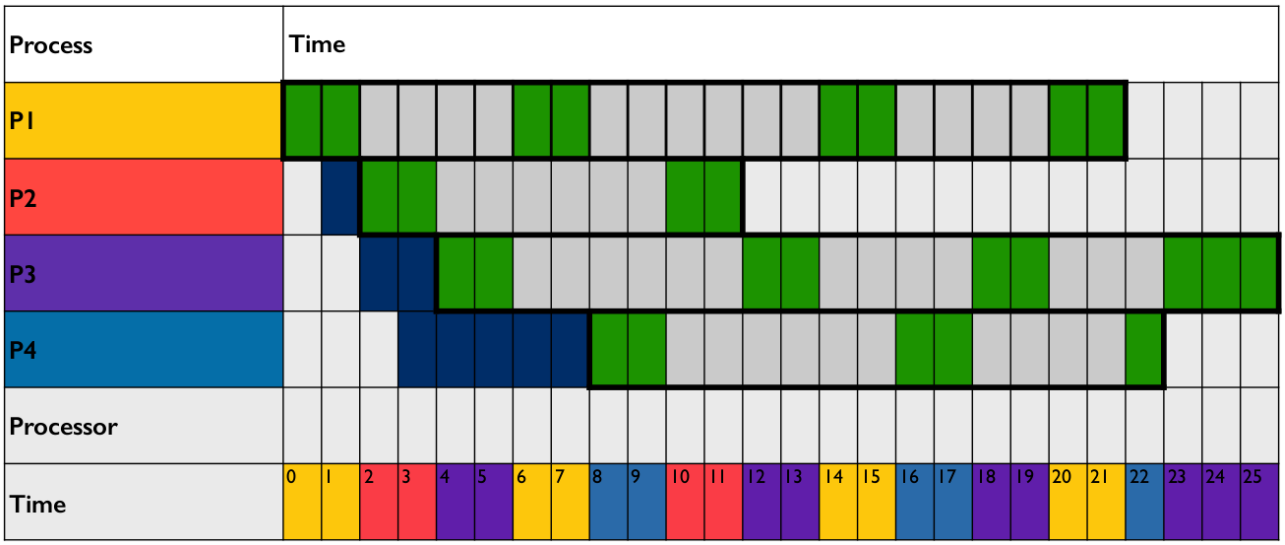
1. First-Come-First-Served (FCFS):

   This method of scheduling can be considered a FIFO queue. The process that is first allocated on the stack and marked as ready to be executed will be first executed and will run until the process has terminated. The queue will then select the process that came next onto the stack and allow this to execute until finished etc.



2. Round-Robin:

The process that requests the CPU first is the first to be executed. This method is implemented once again using an FIFO queue. However, when a process is executed it is placed at the tail of the FIFO queue. Each process is allowed to run for a specific length of time (either t = x or the process finishes in t < x).



3. Shortest-Job-First (Pre-Emptive):

This process allows the process with the least required processing time to have access to the CPU first. It then allows the process with the second least time to use the CPU and so on and so forth. This is difficult to implement as there is no real definitive way of determining how long a process will need access to the CPU for.