Slide 0 - Introduction

- Only theory questions, e.g. definitions of OS, kernel, etc., the various types of Operating Systems

Slide 1 - Process

- Refer to the Quiz 1 practice sheet and solution for practice problems on forking
- Also look at theory questions, e.g. concepts/definitions of process, address space, process states, process control block, system calls, etc.

Slide 2 - Threads

- Theory questions, e.g definitions, types of threads, threading models, threading issues, etc.
- Questions on the output of threads based on code (must know the behavior of the pthread library and the main functions)
- No need to write an explanation, I am just showing it for clarity.

Examples

1. Find outputs of the following code.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void *t_func(void *arg);
int var=0;
int t_id[]={1,2};
int main(){
    pthread_t t1;
    pthread_t t2;
    int a1[]={t_id[0],5};
    int a2[]={t_id[1],3};
    pthread_create(&t1,NULL,t_func,(void *)a1);
    pthread_join(t1,NULL);
    pthread_create(&t2,NULL,t_func,(void *)a2);
    pthread_join(t2,NULL);
```

```
    printf("Value of var after operations of threads: %d\n",var);

    return 0;
}
void *t_func(void *arg){
    int *x=arg;
    if(x[0]==1){
        printf("Entered in Thread :%d\n",x[0]);
        var+=x[1];
        printf("Value of var after the operation of Thread %d: %d\n",x[0],var);
        printf("Operation Done by Thread %d...\n",x[0]);
    }
    else{
        printf("Entered in Thread :%d\n",x[0]);
        var-=x[1];
        printf("Value of var after the operation of Thread %d: %d\n",x[0],var);
        printf("Operation Done by Thread %d...\n",x[0]);
    }
}
```

Output	Explanation
`Entered in Thread :1`	Since pthread_join of t1 was called before thread t2 was created, t1 will always complete entirely before t2 is even started. So, we get the full output of t1, and then we get the full output of t2. Any other ordering for the output will be WRONG for this question.
`Value of var after the operation of Thread 1: 5`	
`Operation Done by Thread 1...`	
`Entered in Thread :2`	
`Value of var after the operation of Thread 2: 2`	
`Operation Done by Thread 2...`	
`Value of var after operations of threads: 2`	

2. Find outputs of the following code.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void *t_func(void *arg);
int var=0;
int t_id[]={1,2};
int main(){
    pthread_t t1;
    pthread_t t2;
    int a1[]={t_id[0],5};
    int a2[]={t_id[1],3};
    pthread_create(&t1,NULL,t_func,(void *)a1);
    pthread_create(&t2,NULL,t_func,(void *)a2);

    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    printf("Value of var after operations of threads: %d\n",var);

    return 0;
}
void *t_func(void *arg){
    int *x=arg;
    if(x[0]==1){
        printf("Entered in Thread :%d\n",x[0]);
        var+=x[1];
        printf("Value of var after the operation of Thread %d: %d\n",x[0],var);
        printf("Operation Done by Thread %d...\n",x[0]);
    }
    else{
        printf("Entered in Thread :%d\n",x[0]);
        var-=x[1];
        printf("Value of var after the operation of Thread %d: %d\n",x[0],var);
        printf("Operation Done by Thread %d...\n",x[0]);
    }
}
```

Output	Explanation

`Entered in Thread :1`	This is a valid output for this one, since both threads were joined after both were created, so there is no definite order for the threads. Sometimes thread 1 finishes first, sometimes thread 2 finishes first, other times both execute at the same time. In this case, any ordering will be valid.
`Value of var after the operation of Thread 1: 5`	
`Operation Done by Thread 1...`	
`Entered in Thread :2`	
`Value of var after the operation of Thread 2: 2`	
`Operation Done by Thread 2...`	
`Value of var after operations of threads: 2`	

Slide 3 - CPU Scheduling

- Given a table of processes, you will have to draw the Gantt chart for the processes. It might be any of the algorithms.
- Theory questions about the different algorithms.
- Refer to the practice problems for quiz for some problems on CPU Scheduling

Slide 4 - Synchronization

- Theory questions about the fundamentals of synchronization, and the basic structure of locking down a critical section, definitions etc.
- Questions on the output of code with synchronization (must know the behavior of the four methods used for mutex and semaphore, particularly the pthread_mutex library, and the semaphore library)
- No need to write an explanation, I am just showing it for clarity.

Examples

1. Find outputs of the following code.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

int t_id[] = {1, 2};
void *t_func1(int *id);
void *t_func2(int *id);
int sum = 15;
pthread_mutex_t m;
sem_t s;

int main() {
    pthread_t t[2];

    sem_init(&s, 0, 0);
    pthread_mutex_init(&m, NULL);

    pthread_create(&t[0], NULL, (void *)t_func1, &t_id[0]);
    pthread_create(&t[1], NULL, (void *)t_func2, &t_id[1]);
```

```
    for (int i = 0; i < 2; i++) {
        pthread_join(t[i], NULL);
    }

    sem_destroy(&s);
    pthread_mutex_destroy(&m);

    printf("Total sum: %d\n", sum);
    return 0;
}

void *t_func1(int *id) {
    sem_wait(&s);

    pthread_mutex_lock(&m);
    for (int i = 0; i < 5; i++) {
        printf("Sum: %d\n", sum);
        sum -= 10;
    }
    pthread_mutex_unlock(&m);
    sem_post(&s);

    return NULL;
}

void *t_func2(int *id) {
    pthread_mutex_lock(&m);
    for (int i = 0; i < 5; i++) {
        printf("Sum: %d\n", sum);
        sum *= 3;
    }
    pthread_mutex_unlock(&m);

    sem_post(&s);
    return NULL;
}
```

Output	Explanation
Sum: 15 `Sum: 45`	In this case, we are using two synchronization tools: - Semaphore: to make sure t_func2 runs before t_func1.

Sum: 135	- Mutex: to make sure that the critical session can only be accessed by a single thread. The semaphore is initially set to 0. So if anyone waits for the semaphore then it cannot pass. That is, if t_func1 runs first it will hit the sem_wait and then block there. When t_func2 will run (whether first or second), it doesn't wait, but instead uses sem_post to unlock the semaphore. Then t_func1 can continue. So t_func2 will definitely run before t_func1.
Sum: 405	
Sum: 1215	
Sum: 3645	
Sum: 3635	
Sum: 3625	
Sum: 3615	
Sum: 3605	
Total sum: 3595	

2. Find outputs of the following code.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

int t_id[] = {1, 2};
void *t_func1(int *id);
void *t_func2(int *id);
int sum = 0;
sem_t s1, s2;

int main() {
   pthread_t t[2];

   sem_init(&s1, 0, 1);
   sem_init(&s2, 0, 0);

   pthread_create(&t[0], NULL, (void *)t_func1, &t_id[0]);
   pthread_create(&t[1], NULL, (void *)t_func2, &t_id[1]);

   for (int i = 0; i < 2; i++) {
      pthread_join(t[i], NULL);
   }
```

```
    sem_destroy(&s1);
    sem_destroy(&s2);

    printf("Total sum: %d\n", sum);
    return 0;
}

void *t_func1(int *id) {
    sem_wait(&s1);

    for (int i = 0; i < 10; i++) {
        printf("Sum: %d\n", sum);
        sum += 10;
    }

    sem_post(&s1);
    sem_post(&s2);
    return NULL;
}

void *t_func2(int *id) {
    sem_wait(&s2);

    for (int i = 0; i < 10; i++) {
        printf("Sum: %d\n", sum);
        sum -= 5;
    }

    sem_post(&s2);
    return NULL;
}
```

Output	Explanation

`Sum: 0`
`Sum: 10`
`Sum: 20`
`Sum: 30`
`Sum: 40`
`Sum: 50`
`Sum: 60`
`Sum: 70`
	Same as the last problem, but we are ensuring that t_func1 runs before t_func2, using the semaphore s2.
	
	t_func2 waits for s2, which is initially 0. Then t_func1 will use sem_post to unlock s2, and then t_func2 will run.

Sum: 80 Sum: 90 Sum: 100 Sum: 95 Sum: 90 Sum: 85 Sum: 80 Sum: 75 Sum: 70 Sum: 65 Sum: 60 Sum: 55 Total sum: 50	And s1 is used to just behave like a mutex lock.