



UNIVERSITY OF THE PUNJAB

B.S. in Computer Science / Fifth Semester – Fall 2023

Subject: Operating Systems

Paper: CC-311

Roll No. [REDACTED]

Time: 3 Hrs. Marks: [REDACTED]

THE ANSWERS MUST BE ATTEMPTED ON THE ANSWER SHEET PROVIDED

Q.1. Answer the following short questions:

(6x5=30)

- A) How we can avoid busy waiting in solution to critical section problem?
- B) What is resource allocation graph? Explain its usage with examples.
- C) Consider a system that has 40% hit ratio with 9 msec time to access TLB. If this system spends double time to access memory, then what will be the Effective Access Time?
- D) If a process has following page table with page size of 1KB. What will be the linear physical address of a memory location whose logical address is (2, 850).

Page #	Frame #
0	16
1	25
2	4
3	7
4	23
5	8

- E) How compare_and_swap() hardware instruction will be used to solve critical section problem?
- F) Explain the use of STBR and STLR while managing system memory using segmentation.

Q.2. Answer the following questions.

(3x10=30)

- A) Consider a system that has 32 megabytes of physical memory installed. There are maximum 8k pages of processes each of size 2k, then what will be the size of page-table? How size of page table will change if we change page size to 4K?
- B) Consider following code segment which uses fork() instruction for creation of a new process. Write how many new processes will be created and what will be the output of the code.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <iostream.h>
int main()
{
    int pid, pid1;
    pid = fork();
    cout << "$$$ - " << endl;
    pid1 = fork();
    if (pid1 == 0)
        cout << "### - " << endl;
    else{
        cout << "Pakistan Zindabad " << endl;
        wait(NULL);
        cout << "Child Terminate " << endl;
    }
    return 0;
}
```

- C) Write reader's code to solve readers-writers problem while assigning high priority to reader's process?

Q1 Short Questions

A. How we can avoid busy waiting in solution to critical section problem

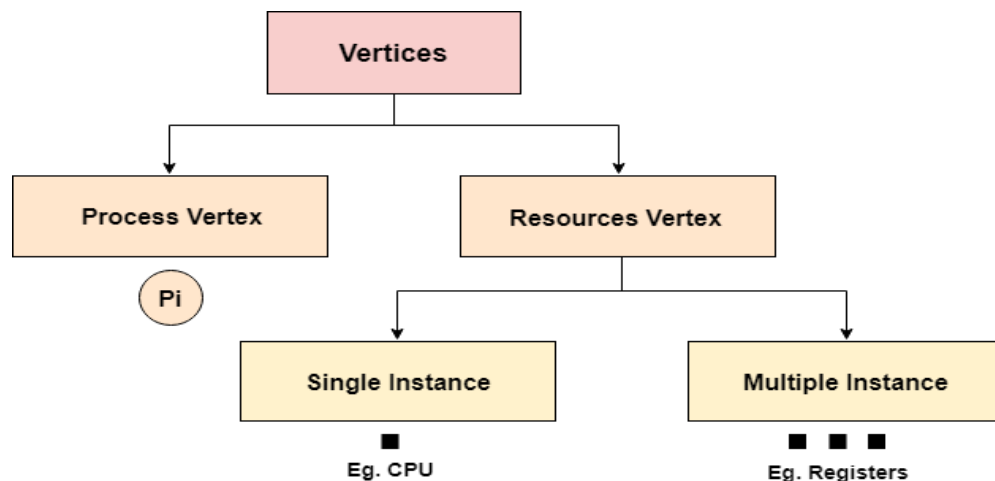
To avoid busy waiting in the solution to the critical section problem, you can use synchronization mechanisms like semaphores, mutexes, condition variables, sleep and wakeup mechanisms, or blocking queues. These allow threads to wait efficiently without actively spinning in a loop.

1. **Semaphore:** Semaphores are integer variables that can be accessed by two standard operations, **wait()** and **signal()**. The **wait()** operation decrements the semaphore value and waits if the value becomes negative. The **signal()** operation increments the semaphore value. By using semaphores, a thread can be made to wait efficiently without actively spinning in a loop.
2. **Mutex:** A mutex (short for mutual exclusion) is a synchronization primitive that grants exclusive access to a resource to only one thread at a time. When a thread acquires a mutex and finds it locked, it will be put to sleep until the mutex becomes available. This prevents busy waiting.

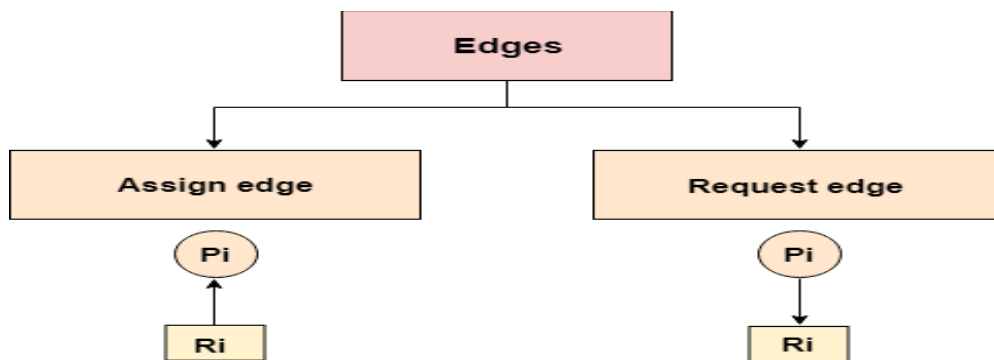
B. What is resource allocation graph? Explain its usage with examples.

Resource Allocation Graph

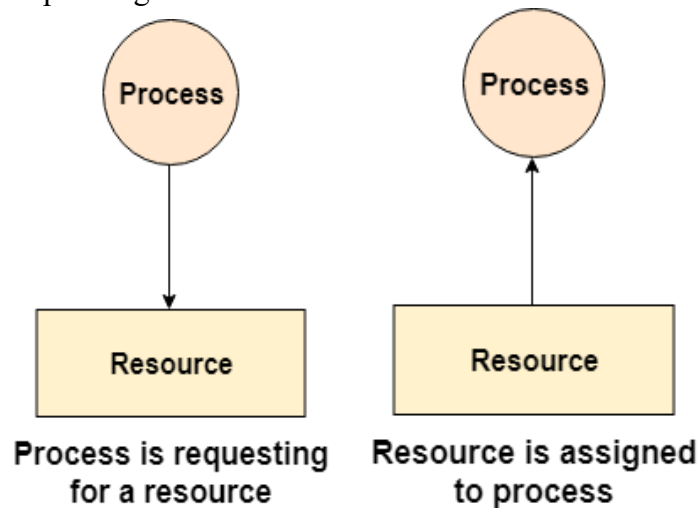
- The resource allocation graph is the pictorial representation of the state of a system. As its name suggests, the resource allocation graph is the complete information about all the processes, which are holding some resources or waiting for some resources.
- It also contains the information about all the instances of all the resources whether they are available or being used by the processes.
- In Resource allocation graph, a Circle represents the process while the Resource is represented by a rectangle. Let us see the types of vertices and edges in detail.



- Vertices are mainly of two types, Resource and process. A different shape will represent each of them. Circle represents process while rectangle represents resource.
- A resource can have more than one instance. Each instance will be represented by a dot inside the rectangle.

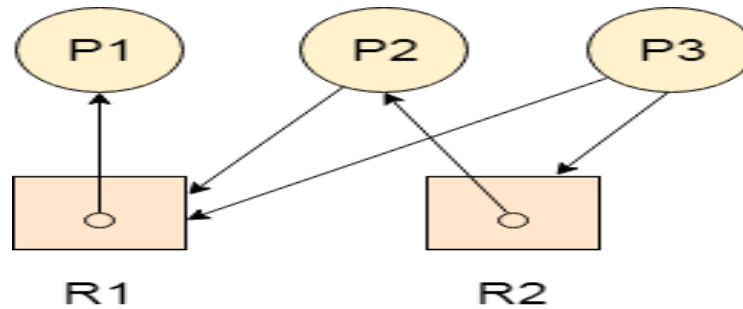


- Edges in RAG are also of two types, one represents assignment and other represents the wait of a process for a resource. The above image shows each of them.
- A resource is shown as assigned to a process if the tail of the arrow is attached to an instance to the resource and the head is attached to a process.
- A process is shown as waiting for a resource if the tail of an arrow is attached to the process while the head is pointing towards the resource.



Example

- Let us consider 3 processes P1, P2 and P3, and two types of resources R1 and R2. The resources are having 1 instance each.
- According to the graph, R1 is being used by P1, P2 is holding R2 and waiting for R1, P3 is waiting for R1 as well as R2.
- The graph is deadlock free since no cycle is being formed in the graph.



C. Consider a system that has 40% hit it with 9 msec time to access TLB. If this system spends double time to access memory, then what will be the Effective Access Time?

To calculate the effective access time (EAT) in this scenario, we need to consider the time it takes to access data when there's a TLB hit and when there's a TLB miss.

Given:

- TLB hit rate = 40% (0.40)
- TLB access time = 9 milliseconds (0.009 seconds)
- Time to access memory when there's a TLB miss = double the normal time, which means it will be 2 times the time taken for a TLB hit

Let's denote:

- T_{hit} = Time to access memory with TLB hit
- T_{miss} = Time to access memory with TLB miss

Then, the effective access time (EAT) can be calculated using the following formula:

$$EAT = (TLB_hit_rate \times T_{hit}) + ((1 - TLB_hit_rate) \times T_{miss})$$

Given that T_{miss} is double the time of T_{hit} , we have:

$$T_{miss} = 2 \times T_{hit}$$

So, plugging in the given values:

$$T_{hit} = 0.009 \text{ seconds}$$

$$T_{miss} = 2 \times 0.009 = 0.018 \text{ seconds}$$

$$EAT = (0.40 \times 0.009) + ((1 - 0.40) \times 0.018)$$

$$EAT = (0.40 \times 0.009) + (0.60 \times 0.018)$$

$$EAT = 0.0036 + 0.0108$$

$$EAT = 0.0144 \text{ seconds}$$

So, the effective access time (EAT) is 0.0144 seconds, or 14.4 milliseconds.

D. If a process has following page table with page size of 1KB. What will be the linear physical address of a memory location whose logical address is (2, 850).

Page#	Frame#
0	16

1	25
2	4
3	7
4	23
5	8

To find the linear physical address corresponding to a given logical address, we need to use the page table entries provided and the page size.

Given:

- Page size = 1KB
- Logical address = (2, 850)

To find the linear physical address:

1. Determine which page the logical address belongs to.
2. Find the frame number corresponding to that page in the page table.
3. Calculate the offset within the page.
4. Combine the frame number and offset to get the linear physical address.

Given the logical address (2, 850):

- Page number = 2
- Offset within the page = 850

Now, let's find the frame number corresponding to page 2 from the page table:

- For page 2, the frame number is 4.

Since the page size is 1KB, the logical address contains the page number and the offset within the page. To find the linear physical address, we use the formula:

$$\text{Linear Physical Address} = (\text{Frame Number} \times \text{Page Size}) + \text{Offset}$$

Substituting the values:

$$\text{Linear Physical Address} = (4 \times 1\text{KB}) + 850$$

$$\text{Linear Physical Address} = (4 \times 1024) + 850$$

$$\text{Linear Physical Address} = 4096 + 850$$

$$\text{Linear Physical Address} = 4946$$

Therefore, the linear physical address corresponding to the logical address (2, 850) is 4946.

E. How compare_and_swap() hardware instruction will be used to solve critical section problem?

The `'compare_and_swap()'` (CAS) hardware instruction can be used to implement lock-free algorithms for solving the critical section problem. In these algorithms, CAS is used to atomically check and modify a shared variable representing the lock. Threads attempt to acquire the lock by performing a CAS operation on this variable. If the CAS operation succeeds, meaning that no other thread has acquired the lock concurrently, the thread enters the critical section. If the CAS operation fails, indicating that another thread has already acquired the lock, the thread retries or performs some other action, such as waiting or yielding. CAS provides the necessary atomicity for implementing lock-free synchronization without the need for locks or mutexes, thus avoiding the overhead of context switching and potential deadlock situations.

F. Explain the use of STBR and STLR while managing system memory using segmentation.

In memory management using segmentation, the Segment Table Base Register (STBR) and the Segment Table Length Register (STLR) are key components of the hardware support for segmentation.

1. Segment Table Base Register (STBR):

- The STBR holds the base address of the segment table in memory.
- When a program executes, the CPU uses the STBR to locate the segment table in memory.
- By loading the STBR with the starting address of the segment table, the CPU knows where to find information about each segment of the program.

2. Segment Table Length Register (STLR):

- The STLR specifies the length or size of the segment table.
- It defines the range of memory addresses that the segment table covers.
- When a program accesses memory, the CPU uses the STLR to ensure that the segment table lookup stays within the bounds of the segment table.
- This prevents segmentation faults or errors caused by accessing memory beyond the allocated segment table.

Together, the STBR and STLR provide the necessary hardware support for segmentation by enabling the CPU to efficiently access and manage segment information in memory. They allow the CPU to translate logical addresses generated by the program into physical addresses by

consulting the segment table, which contains information about the location and size of each segment in memory.

Q2 Long Questions

A. Consider a system that has 32 megabytes of physical memory Installed. There are maximum 8k pages of processes each of size 2k, then what will be the size of page-table? How size of page table will change if we change page size to 4K?

To calculate the size of the page table, we need to consider the number of pages and the size of each page entry in the page table.

Given:

- Physical memory installed: 32 megabytes
- Maximum 8k pages of processes, each of size 2k

Let's calculate the size of the page table for the initial page size of 2K:

1. Number of Pages:

- Maximum 8k pages means $8 * 1024$ pages.

2. Size of Page Entry:

- Since each page is 2K and assuming we need 1 entry per page, the size of each page entry in the page table is the size of a physical memory address, which depends on the total physical memory installed.

To calculate the size of the page table, we multiply the number of pages by the size of each page entry:

Size of Page Table=Number of Pages×Size of Page Entry

Now, let's calculate the size of the page table:

Number of Pages= $8*1024=8192$ pages

Assuming the physical memory installed is divided evenly among the pages:
Size of Page Entry=Total Physical Memory/Number of Pages

Size of Page Entry= $32/8192$ MB=4 KB

Size of Page Table= 8192 pages×4 KB=32 MB

So, the size of the page table for the initial page size of 2K is 32 megabytes, which is the same as the total physical memory installed.

Now, if we change the page size to 4K, let's recalculate the size of the page table:

1. Number of Pages:

- With a page size of 4K, there will be fewer pages required to cover the same amount of physical memory.

2. Size of Page Entry:

- Since each page is now 4K, the size of each page entry in the page table remains the same.

Let's calculate the new size of the page table:

Number of Pages=Total Physical Memory/Page Size

Number of Pages= 8192 pages * 4 KB= 32 MB

Size of Page Table=8192 pages×4 KB=32 MB
Size of Page Table=8192 pages×4 KB=32 MB

So, even if we change the page size to 4K, the size of the page table remains the same at 32 megabytes.

B. Consider following code segment which uses fork() Instruction for creation of a new process. Write how many new processes will be created and what will be the output of the code.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <iostream.h>
Int main()
{
    int pld, pid1;
    pld = fork();
    cout<< "$$$ - "<<endl;
    pld1 = fork();
    If (pid1==0)
        cout<< "#### - "<<endl;
    else{
        cout<< "Pakistan Zindabad "<<endl;
        wait(NULL);
        cout<< "Child Terminate"<<endl;
    }
    return 0;
}
```

This code segment creates new processes using the **fork()** system call. Let's analyze how many processes will be created and what will be the output.

1. The first **fork()** call (**pid = fork();**) creates a new process. Let's call this Process 1.
2. The **cout << "\$\$\$\$ - " << endl;** statement will be executed by both the parent process and the newly created child process. So, both Process 1 and its child will print "\$\$\$\$ - ".
3. Now, both Process 1 and its child will continue to execute the code below. The child process will execute the second **fork()** call (**pid1 = fork();**), creating another process. Let's call this Process 2.
4. The child process (Process 2) will print "#### - ".
5. The parent process (Process 1) will print "Pakistan Zindabad ", then wait for its child process (Process 2) to terminate. After the child process terminates, it will print "Child Terminate".

So, the total number of new processes created will be 2: one from the first **fork()** and another from the second **fork()** within the child process.

The output will be:

```
$$$$ -  
$$$$ -  
Pakistan Zindabad  
$$$$ -  
#### -  
Pakistan Zindabad  
Child Terminate  
Child Terminate
```

Explanation:

- The first two lines with "\$\$\$\$ - " are printed by both the parent process and its child created by the first **fork()**.
- Then, the parent process prints "Pakistan Zindabad " and waits for its child process (Process 2) to terminate.
- In the child process (Process 2), "#### - " is printed.
- Finally, both the parent and child processes print "Child Terminate".

C. Write reader's code to solve readers-writers problem while assigning high priority to reader's process?

To solve the readers-writers problem while assigning high priority to readers, we can use a solution that allows multiple readers to access the shared resource simultaneously while ensuring exclusive access for writers. One common solution is the Readers-Writer Lock.

Here's a simple implementation of a readers-writers solution with high priority given to readers in C++ using mutex and conditional variables:

```
#include <iostream>
```

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

```
using namespace std;
```

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
pthread_cond_t cond_reader = PTHREAD_COND_INITIALIZER;
```

```
pthread_cond_t cond_writer = PTHREAD_COND_INITIALIZER;
```

```
int readers_count = 0;
```

```
bool writing = false;
```

```
void *reader(void *arg) {
```

```
    while (true) {
```

```
        pthread_mutex_lock(&mutex);
```

```
        // Wait if there is a writer or if writers are waiting
```

```
        while (writing) {
```

```
            pthread_cond_wait(&cond_reader, &mutex);
```

```
        }
```

```
        // Increment readers count and signal other readers
```

```
        readers_count++;
```

```
        pthread_cond_broadcast(&cond_reader);
```

```
        pthread_mutex_unlock(&mutex);
```

```
        // Read data
```

```
pthread_mutex_lock(&mutex);

readers_count--;

// Signal writer if no readers are reading
if (readers_count == 0) {
    pthread_cond_signal(&cond_writer);
}

pthread_mutex_unlock(&mutex);

// Continue reading
}

return NULL;
}

void *writer(void *arg) {
    while (true) {
        pthread_mutex_lock(&mutex);

        // Wait if there are readers or a writer
        while (readers_count > 0 || writing) {
            pthread_cond_wait(&cond_writer, &mutex);
        }

        // Set writing flag to true
```

```
writing = true;

pthread_mutex_unlock(&mutex);

// Write data

pthread_mutex_lock(&mutex);

// Reset writing flag
writing = false;

// Signal readers
pthread_cond_broadcast(&cond_reader);
pthread_cond_signal(&cond_writer);

pthread_mutex_unlock(&mutex);

// Continue writing
}

return NULL;
}

int main() {
    pthread_t readers[5], writers[2];

    // Create reader threads
    for (int i = 0; i < 5; i++) {
```

```

        pthread_create(&readers[i], NULL, reader, NULL);
    }

    // Create writer threads
    for (int i = 0; i < 2; i++) {
        pthread_create(&writers[i], NULL, writer, NULL);
    }

    // Join threads
    for (int i = 0; i < 5; i++) {
        pthread_join(readers[i], NULL);
    }
    for (int i = 0; i < 2; i++) {
        pthread_join(writers[i], NULL);
    }

    return 0;
}

```

In this implementation, readers have high priority. They can access the shared resource as long as there are no writers currently writing or waiting to write. Writers have exclusive access to the resource; they can only write when there are no readers reading or waiting to read. The solution uses mutexes and conditional variables to synchronize access to the shared resource and coordinate between readers and writers.