

CASE STUDY – OS/2 WARP REVIEW-2

COURSE CODE : 19CSE213

COURSE NAME: OPERATING SYSTEMS

TEAM MEMBERS :

S.No	Name of the Student	Roll No.
1.	RAVELLA ABHINAV	CB.EN.U4CSE19453
2.	PENUGONDA KOUSHIK	CB.EN.U4CSE19449
3.	SINGADI SHANTHAN REDDY	CB.EN.U4CSE19459
4.	NUSUM KARTHIK	CB.EN.U4CSE19444

THREADS

R. Abhinav
CB.EN.U4CSE19453

- What is a thread?

- A thread is a path of execution within a process.
A process can contain multiple threads.
- The thread consists of a program counter, stack and a set of registers

- Why threads:-

- The main and primary difference between a thread and a process is that thread run in a shared memory space and process run in separate memory space.
- Threads share lot of things among themselves
Eg:- Code section, data section, OS resources etc.,
- How do we use threads in our program?

The API's for using threads & managing threads are provided by a library called as "thread library".

- These thread libraries can be implemented by user space & by kernel space.
- Three most popular thread libraries that are used are as follows:
 - (i) Posix threads
 - (ii) Java threads
 - (iii) Win32 threads

Windows & threads:

- The Windows API uses Win32 thread library to create and manage threads.
- This library can be used by including the following header file in the program:
`#include <windows.h>`

Some system calls:

① createThread():

usage: This system call is to create a thread to execute within the virtual address space of the calling process.

Parameters that this system call has:

- ① Security attributes (default - NULL)
- ② Thread stack size (default - 0)
- ③ Thread start routine (function)
- ④ Variable (pointer to variable) that needs to be passed to the thread.
- ⑤ Creation Flags (default - 0)
- ⑥ This is a pointer to a variable that receives the thread-id.

• Security attributes:

This pointer to the structure determines whether the returned handle can be inherited by the child process or not. If 'NULL' the handle cannot be inherited.

2. Thread stack size :-

If this parameter is zero, the thread uses the size of executable.
This is basically the initial size of the stack.

3. Thread start routine :-

This is basically the function that has to be executed by the thread.

4. Parameters :-

A pointer to variable that is passed to the thread.

5. Creation Flags :-

This parameter controls the creation of threads. If '0' is passed gets executed immediately after creation.

6. Thread-id :-

A pointer to a variable that receives the thread-id.

Return-value :-

If thread execution succeed return value is a handle to a new thread & fails returns "NULL".

• CloseHandle() :

Usage: This function closes the open handle

Parameter:

Valid handle: to an object that is open.

This system call returns a non-zero value if the function succeeds

- Wait for Single Object () :

- * Usage:

Waits until the specified object is in the state & / time out interval elapses

- * Parameters:

Handle: The return value that thread creation gives i.e., Thread Handle in this case.

Time out in milliseconds. If a non-zero value is specified, the function waits until the object is signaled & / interval elapses.

THREADS

NAME	RAVELLA ABHINAV
ROLL No.	CB.EN.U4CSE19453

```
#include <windows.h>
#include <stdio.h>

DWORD Sum_of;
DWORD Diff;
DWORD product;
DWORD divi;

typedef struct parameters{
    int operand1;
    int operand2;
}param, *para;

DWORD WINAPI Sum(LPVOID Param) {
    para Upper = (para)Param;
    Sum_of = Upper->operand1 + Upper->operand2;
    return 0;
}

DWORD WINAPI Difference(LPVOID Param) {
    para Upper = (para)Param;
    if (Upper->operand1 > Upper->operand2) {
        Diff = Upper->operand1 - Upper->operand2;
    }
    else{
        Diff = Upper->operand2 - Upper->operand1;
    }
    return 0;
}

DWORD WINAPI Product(LPVOID Param) {
    para Upper = (para)Param;
    product = Upper->operand1 * Upper->operand2;
    return 0;
}

DWORD WINAPI Divide(LPVOID Param) {
```

```

para Upper = (para)Param;

if (Upper->operand2 == 0) {
    divi = -1;
}
else{
    divi = Upper->operand1/Upper->operand2;
}
return 0;
}

int main() {
DWORD thread_id;
HANDLE ThreadHandle;
HANDLE ThreadHandle1;
HANDLE ThreadHandle2;
HANDLE ThreadHandle3;

struct parameters param;
struct parameters *params;

params = &param;

printf("please enter the first operand : ");
scanf("%d", &params->operand1);

printf("please enter the second operand : ");
scanf("%d", &params->operand2);

int operation;

printf("=====Menu=====\\n");
printf("1. ADD\\n");
printf("2. SUBTRACT\\n");
printf("3. PRODUCT\\n");
printf("4. DIVIDE\\n");
printf("=====END OF MENU=====\\n");

printf("please choose an operation : ");
scanf("%d", &operation);

if (operation == 1){

    ThreadHandle = CreateThread(
        NULL,          //default security attribute
        0,             //default stack size of the thread
        Sum,           //function that is called by the thread
        params,         //structure of arguments that are passed into
the threads
        0,             //default creation flags
        &thread_id      // the thread returns the thread id here
    );

    //this system call will wait for the thread to finish, for infinite
amount of time.

    WaitForSingleObject(ThreadHandle, INFINITE);

    //closing the thread here
}

```

```

        CloseHandle(ThreadHandle);

        printf("Sum: %lu\n", Sum_of);
    }

    else if(operation == 2) {

        ThreadHandle1 = CreateThread(
            NULL,           //default security attribute
            0,             //default stack size of the thread
            Difference,    //function that is called by the thread
            params,         //structure of arguments that are passed into
the threads
            0,             //default creation flags
            &thread_id      // the thread returns the thread id here
        );

        //this system call will wait for the thread to finish, for infinite
amount of time.

        WaitForSingleObject(ThreadHandle1, INFINITE);

        //closing the thread here

        CloseHandle(ThreadHandle1);

        printf("Difference: %lu\n", Diff);
    }

    else if(operation == 3) {
        ThreadHandle2 = CreateThread(
            NULL,           //default security attribute
            0,             //default stack size of the thread
            Product,       //function that is called by the thread
            params,         //structure of arguments that are passed into
the threads
            0,             //default creation flags
            &thread_id      // the thread returns the thread id here
        );

        //this system call will wait for the thread to finish, for infinite
amount of time.

        WaitForSingleObject(ThreadHandle2, INFINITE);

        //closing the thread here

        CloseHandle(ThreadHandle2);

        printf("product: %lu\n", product);
    }

    else{
        ThreadHandle3 = CreateThread(
            NULL,           //default security attribute
            0,             //default stack size of the thread
            Divide,        //function that is called by the thread
            params,         //structure of arguments that are passed into
the threads
            0,             //default creation flags
            &thread_id      // the thread returns the thread id here

```

```
) ;

//this system call will wait for the thread to finish, for infinite
amount of time.

WaitForSingleObject(ThreadHandle3, INFINITE);

//closing the thread here

CloseHandle(ThreadHandle3);

printf("Quotient: %lu\n", divi);
}

}
```

Output :

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19042.906]
(c) Microsoft Corporation. All rights reserved.

C:\Users\RAVELLA ABHINAV\OneDrive\Desktop\Review2>gcc -o output thread.c

C:\Users\RAVELLA ABHINAV\OneDrive\Desktop\Review2>output.exe
please enter the first operand : 11
please enter the second operand : 13
=====Menu=====
1. ADD
2. SUBTRACT
3. PRODUCT
4. DIVIDE
=====END OF MENU=====
please choose an operation : 1
Sum: 24

C:\Users\RAVELLA ABHINAV\OneDrive\Desktop\Review2>
```

```
C:\Users\RAVELLA ABHINAV\OneDrive\Desktop\Review2>gcc -o output thread.c

C:\Users\RAVELLA ABHINAV\OneDrive\Desktop\Review2>output.exe
please enter the first operand : 11
please enter the second operand : 13
=====Menu=====
1. ADD
2. SUBTRACT
3. PRODUCT
4. DIVIDE
=====END OF MENU=====
please choose an operation : 2
Difference: 2

C:\Users\RAVELLA ABHINAV\OneDrive\Desktop\Review2>
```

```
C:\Users\RAVELLA ABHINAV\OneDrive\Desktop\Review2>output.exe
please enter the first operand : 11
please enter the second operand : 13
=====Menu=====
1. ADD
2. SUBTRACT
3. PRODUCT
4. DIVIDE
=====END OF MENU=====
please choose an operation : 3
product: 143

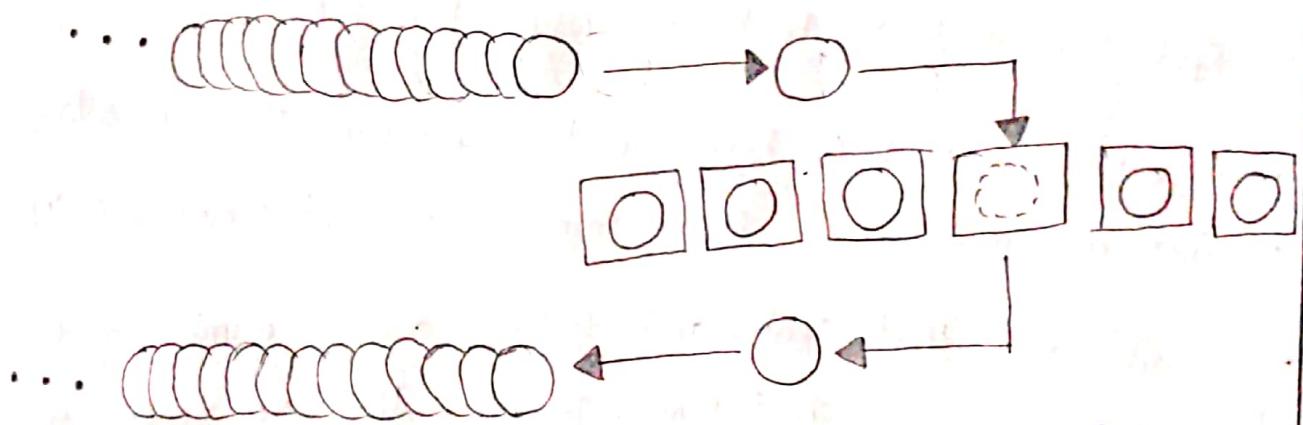
C:\Users\RAVELLA ABHINAV\OneDrive\Desktop\Review2>output.exe
please enter the first operand : 11
please enter the second operand : 13
=====Menu=====
1. ADD
2. SUBTRACT
3. PRODUCT
4. DIVIDE
=====END OF MENU=====
please choose an operation : 4
Quotient: 0

C:\Users\RAVELLA ABHINAV\OneDrive\Desktop\Review2>
```

THREAD POOLING

→ what is a thread pool?

- A thread pool is a collection of worker threads that efficiently execute asynchronous callbacks on behalf of the application.
- Application can queue work items, associate work with waitable handler, automatically queue based on a timer, and bind with I/O.



- A thread pool maintains multiple threads waiting for tasks to be allocated for concurrent execution by the supressing program.

→ Advantages:

- Thread management costs from thread creation are minimized. This leads to better response times for processing workloads and allows for multi-threading of timer - gained workloads.

- Creating a thread pool over creating a new thread for each task is that thread creation & destruction overhead is restricted to the initial creation of the pool. which results in better performance & better system stability.

- Possible runtime features midway through application execution due to inability to create threads can be avoided with simple control to give.

→ Applications that can benefit using thread pools:

- An application that is highly parallel and can dispatch a large number of small work items asynchronously [ex: network I/O]
- An application that creates and destroys a large number of threads that each run for a short time. Thread Pooling can reduce the complexity of thread management.
- An application that often performs an exclusive unit on kernel objects as blocks on incoming events on an object. Thread Pooling can increase performance by reducing the number of context.

→ Usage Scenario:

Server applications, which after launch a thread for every new request better strategy → queue service requests from the queue, process it, and returns to the queue to get more work.

→ Thread Pool architecture contains of :

1. worker threads that execute the callback functions.
2. waiter threads that waits on multiple wait handlers.
3. A work queue.
4. A default thread pool for each process.
5. A worker factory that manages the worker threads.

→ Using thread pool functions.

Create threadpool function

Definition	Allocates a new pool of threads to execute callbacks.
Syntax	PTP_POOL CreateThreadpool { PVOID reserved };
Parameters	reserved [This Parameter is Shared & invert be null]
Return value	Succeeds → returns a pointer to a TPTPOOL structure (The newly allocated thread pool) Fails → returns NULL

Create Thread Pool Wait Function

Definition

Create a new wait object

Syntax

```
PTP_WAIT CreateThreadpoolWait [  
    PTP_WAIT_CALLBACK Pfnwa,  
    PVOID Pv,  
    PTP_CALLBACK_ENVIRON Pebe  
];
```

Parameters

Pfnwa → callback function to call when the wait complete or timer out.

Pv → optional application defined later to pass the callback function

Pebe → A PTP_CALLBACK_ENVIRON structure that defines the environment to execute in.

Succeeds → returns a pointer to a TPT_WAIT structure that defines wait object

Fails → returns null.

Close Thread Pool Function

definition

Closes the specified thread pool.

Syntax

```
void CloseThreadpool [  
    PTP_POOL Ptp  
];
```

Parameters

Ptp → a pointer to a TP_POOL structure that defines the thread pool that CreateThreadpool function return this pointer.

return value

None

THREAD SYNCHRONIZATION:-

If is concurrent execution of two (or) more threads that share critical resources. Synchronization of threads help in avoiding conflicts regarding critical resources. otherwise, conflicts may arise, when parallel running threads attempts to modify a common variable at the same time.

* Critical section: The region of a program that try to access shared resources and cause race condition.

* Race condition: It typically occurs when two (or) more threads try to read, write and possibly make the decisions based on the memory they are accessing concurrently.

Pseudocode:-

do

{

 || entry section \rightarrow wait(); here request are processed for entry into critical section.

 * critical section*

 || exit section \rightarrow signal(); here removes locks on critical section.

 - remaining section

}

while (True)

Solution for critical section:-

solution for a critical section Problem must satisfy following cond:-

- i) Mutual exclusion: out of group of threads, only one thread can be in its critical section at a given point of time.
- ii) Progress: If no thread is in the critical section, and if one or more thread wants to execute their critical section then only one of these threads must be allowed to get in.
- iii) Bounded-wait: After a thread makes request for getting into critical section, there is limit for how many process may get into the critical section, before threads request is granted.

widely used methods for critical section problem:-

* Peterson's solution.

* Mutex lock.

* Semaphores.

SEMAPHORE:- It is a signaling mechanism, and a thread that is waiting on Semaphore, can be signaled by another thread. There

are two types of semaphores:

i) Counting semaphore.

ii) Binary semaphore.

All the semaphores make use of two atomic operations i) wait and ii) signal.

System calls in windows :-

(i) Create semaphore :-

LPSecurityAttribute,

lInitialCount,

lMaximumCount,

lpName

);

LPSecurityAttribute :- it is a security attribute, if NULL handle can't be inherited by its child.

lInitialCount :- must be greater than zero and less than or equal to Max count. Signaled state \rightarrow greater than zero, Non-signaled state = 0.

lMaximumCount :- max count of semaphore object.

lpName :- name of semaphore object.

(ii) WaitForSingleObject :-

hHandle;

dwMilliseconds

);

hHandle :- A handle to object.

dwMilliseconds :- time-out interval in milliseconds

(iii) WaitForMultipleObjects (

nCount,
lpHandle,
bWaitAll
dwMilliseconds,

);

nCount: max Number of objects handles in array pointed to lpThreads

lpThread: array of object handles.

bWaitAll: If TRUE, returns when all objects are signaled, If FALSE, returns when any one object is signaled.

dwMillisecond: time-out interval in Millisecond.

(iv) ReleaseSemaphore (

hSemaphore,
nReleaseCount,
lpPreviousCount.

);

SYNCHRONIZATION

NAME	S.SHAN THAN REDDY
ROLL No.	CB.EN.U4CSE19459

```
#include <windows.h>
#include <stdio.h>

#define MAX_SEM_COUNT 4
#define THREADCOUNT 4

HANDLE ghSemaphore;
int num1, num2, sum;

DWORD WINAPI ThreadProcSemaphore( LPVOID lpParam )
{

    // lpParam not used in this example
    UNREFERENCED_PARAMETER(lpParam);

    DWORD dwWaitResult;
    BOOL bContinue=TRUE;

    while(bContinue)
    {
        // Try to enter the semaphore gate.

        dwWaitResult = WaitForSingleObject(
            ghSemaphore,      // handle to semaphore
            INFINITE);       // zero-second time-out interval

        switch (dwWaitResult)
        {
            // The semaphore object was signaled.
            case WAIT_OBJECT_0:
                // TODO: Perform
                printf("\nThread with %d id is executing...", GetCurrentThreadId());
                printf("\nEnter 1st number: ");
                scanf("%d", &num1);

                printf("Enter 2nd number: ");
                scanf("%d", &num2);

                sum = num1 + num2;
                printf("Sum of %d and %d : %d\n", num1, num2, sum);

                bContinue=FALSE;
        }
    }

    // Release the semaphore when task is finished
}
```

```

        if (!ReleaseSemaphore(
            ghSemaphore, // handle to semaphore
            1,           // increase count by one
            NULL) )      // not interested in previous count
    {
        printf("ReleaseSemaphore error: %d\n", GetLastError());
    }
    break;

    // The semaphore was nonsignaled, so a time-out occurred.
    case WAIT_TIMEOUT:
        printf("Thread %d: wait timed out\n",
GetCurrentThreadId());
        break;
    }
}
return TRUE;
}

DWORD WINAPI ThreadProc( LPVOID lpParam )
{
    // lpParam not used in this example
    UNREFERENCED_PARAMETER(lpParam);
int i;

for( i=0; i < 5; i++)
{
    Sleep(2);
    printf("Thread with %d id is executing..... \n",
GetCurrentThreadId());
}

/*printf("\nEnter 1st number: ");
scanf("%d", &num1);

printf("\nEnter 2nd number: ");
scanf("%d", &num2);

sum = num1 + num2;

printf("Sum : %d", sum); */

return 0;
}

int main( void )
{
HANDLE aThread[THREADCOUNT];
DWORD ThreadID;
int i;

// Create a semaphore with initial and max counts of MAX_SEM_COUNT

ghSemaphore = CreateSemaphore(
    NULL,           // default security attributes
    1,              // initial value
    1,              // no.of resources
    NULL);          // unnamed semaphore
}

```

```

if (ghSemaphore == NULL)
{
    printf("CreateSemaphore error: %d\n", GetLastError());
    return 1;
}

// Create worker threads

printf("\nTHREAD SYNCHRONIZATION WITH SEMAPHORES\n");
for( i=0; i < 2; i++ )
{
    aThread[i] = CreateThread(
        NULL,           // default security attributes
        0,             // default stack size
        &ThreadProcSemaphore,          //starting address of the
thread
        NULL,           // no thread function arguments
        0,             // default creation flags
        &ThreadID); // receive thread identifier

    if( aThread[i] == NULL )
    {
        printf("CreateThread error: %d\n", GetLastError());
        return 1;
    }
}

// Wait for all threads to terminate

WaitForMultipleObjects(
    2,             // number of object handles in the array
    aThread,       // array of object handles
    TRUE,          // function returns when the state of all objects in
the handles array is signaled
    INFINITE);    // time-out interval, in milliseconds

// Close thread and semaphore handles

for( i=0; i < 2; i++ )
    CloseHandle(aThread[i]);

CloseHandle(ghSemaphore);

printf("\nTHREADS WITHOUT SEMAPHORES\n");
for( i=2; i < 4; i++ )
{
    aThread[i] = CreateThread(
        NULL,           // default security attributes
        0,             // default stack size
        (LPTHREAD_START_ROUTINE) ThreadProc,      //starting
address of the thread
        NULL,           // no thread function arguments
        0,             // default creation flags
        &ThreadID); // receive thread identifier

    if( aThread[i] == NULL )
    {
        printf("CreateThread error: %d\n", GetLastError());
        return 1;
    }
}

```

```
}

WaitForSingleObject(
    aThread[2],      // handle to semaphore
    INFINITE);       // zero-second time-out interval

WaitForSingleObject(
    aThread[3],      // handle to semaphore
    INFINITE);       // zero-second time-out interval

return 0;
}
```

Output :

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19042.906]
(c) Microsoft Corporation. All rights reserved.

C:\Users\RAVELLA ABHINAV\OneDrive\Desktop\Review2>gcc semaphore.cpp
C:\Users\RAVELLA ABHINAV\OneDrive\Desktop\Review2>a

THREAD SYNCHRONIZATION WITH SEMAPHORES

Thread with 17540 id is executing...
Enter 1st number: 59
Enter 2nd number: 53
Sum of 59 and 53 : 112

Thread with 18044 id is executing...
Enter 1st number: 44
Enter 2nd number: 49
Sum of 44 and 49 : 93

THREADS WITHOUT SEMAPHORES
Thread with 17400 id is executing.....
Thread with 9628 id is executing.....
Thread with 9628 id is executing.....
Thread with 17400 id is executing.....
Thread with 17400 id is executing.....
Thread with 9628 id is executing.....
Thread with 9628 id is executing.....
Thread with 17400 id is executing.....
Thread with 9628 id is executing.....
Thread with 17400 id is executing.....
C:\Users\RAVELLA ABHINAV\OneDrive\Desktop\Review2>
```

Synchronization

The Process synchronization is the task of coordinating the execution of Process in a way that no two processes can have access to the same shared data and resources.

They can also have variety of ways to coordinate multiple threads of executes.

using critical section objects: A critical section object provides synchronization similar to that provided by a mutex object, except that a critical section can be used only by the threads of a singular process. A synchronization object whose handle can be specified in one of the wait functions to coordinate the execution of multiple threads.

Enter critical section: waits for ownership of the specified critical section object.

Leave critical section: Releases ownership of the specified critical section object.

Initialize critical section: initialize a critical section object.

Delete critical section: Releases all resources used by the unowned critical section object.

- * In windows os, the code area requiring exclusive access to some shared data is called as critical section.
- * The structure type for working with critical section is CRITICAL-SECTION.
- * Critical section in Linux os is file variable mutex pthreads -mutex-t. Before using this variable needs to be initialized write the value of the constant PTHREAD-MUTEX-INITIALIZE or call pthread-mutex-init.

Syntax:

```
HANDLE CREATE THREAD (
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    SIZE_T dwStackSize,
    LPTHREAD-START-ROUTINE lpParameterAddress,
    -dRv-LPVOID lpParameter
    PWORD dwCreationFlags,
    LPWORD lpThreadId,
);
```

lpThread Attributes: A pointer to a security-Attributes structure that determines whether the required returned handle can be inherited by child process.

dw stack size) The initial size of stack in bytes.

lp start Address) A pointer to the application defined function to be executed by thread.

lp Parameter) A pointer to a variable to be passed to thread.

dw creation flags) The flags to a variable to that control the creation of thread if o the thread runs immediately

lp thread Id) It is a thread Identifier.

Initialize critical section) A pointer to the critical section object.

wait for single object (HANDLE handle,
 DWORD dwMilliseconds
);

* waits until the specified object is in the signaled state are timed out.

CPU SCHEDULING

• CPU scheduling is a process that allows one process to use the CPU while the execution of another process is on hold due to unavailability of any resources like I/O etc, thereby making full use of CPU. The aim of the CPU scheduling is to make the system efficient fast and fair.

Dragonfly BSD mainly supports 2 CPU scheduling algorithms

- * Light weight kernel threads (LWKT) and
- * Priority Based Round Robin (PBRR)

LIGHT WEIGHT KERNEL THREADS (LWKT)

The light weight kernel threads system decouples the traditional unix notions of an execution context from a VM address space. This is similar to what many other systems such as free BSD have done.

- * The LWKT has its own self-contained thread scheduler. Threads are tied to a process and can only move under special circumstances.
- * A thread in LWKT can only be preempted by an interrupt thread. Both fast interrupts, where the interrupt is handled in the current thread context, and threaded interrupts, where the LWKT scheduler switches to the interrupt thread and back when it's done, are supported by the LWKT system.

Cross Process Scheduling is implemented via asynchronous inter-processor interrupts. These messages can be batched for a given interrupt, the system executes graceful degradation under load.

A lot of work went into separating the LWKT scheduler and the user process scheduler. The LWKT thread scheduler is MP safe and uses a fast per-CPU fixed priority Round Robin scheme with a well defined API for communicating with other processors' schedulers. The traditional BSP multilevel scheduler is implemented on top of the threads scheduler.

Advantages of LWKT :-

- * Scheduler can decide to give more time for a process which has more number of threads than process having less number of threads.

Disadvantages of LWKT :-

- * LWKT can be really slow if the time difference between processes is very large, (i.e.) if the burst time range varies a lot then LWKT becomes slow as it gives priority to process with more number of threads which makes processes with less number of threads waiting for a long period of time.

Priority Based Round Robin

Priority based Round Robin (PBRR) focuses on the drawbacks of simple round robin architecture which gives equal priority to all the process (process are scheduled in FCFS manner). Because of this drawback round robin architecture is not efficient with smaller CPU burst. This result increased waiting time and response time of a process which results in the decrease in the system throughput.

Priority Based Round Robin (PBRR) eliminates the defects of implementing simple round architecture. PBRR algorithm will be executed in two steps which will help to minimize a number of performance parameters such as context switches, average waiting time, and Average turnaround time.

The algorithm performs following steps:

Step 1 : Allocate CPU to every process in RR fashion, according to the given priority, for given quantum (say k units) only for 1 time.

Step 2 : After step 1, the following are done.

- a) processes are managed in increasing order of their remaining CPU burst time in the ready queue. New priorities are

assigned according to the remaining CPU burst of process) the process with shortest CPU time is assigned with highest priority.

- b) The processes are executed according to the new priorities based on the remaining CPU bursts and each process gets the control of the CPU until they finished their execution.

Advantages of PBRR

- * Dramatically improves average response time.
- * Improves efficiency of context switching, average waiting and turnaround time.
- * Though every process gets a chance to run, the higher priority processes will complete first, and they need not wait for a very long time.