IN1011

Operating Systems

Lecture 04
(part 1): CPU Scheduling Animated
(part 2): Introduction to Threads
(part 3): Types of Threads
(part 4): Unix Thread API Demo

IN1011

Operating Systems

Lecture 04 (part 1): CPU Scheduling Animated

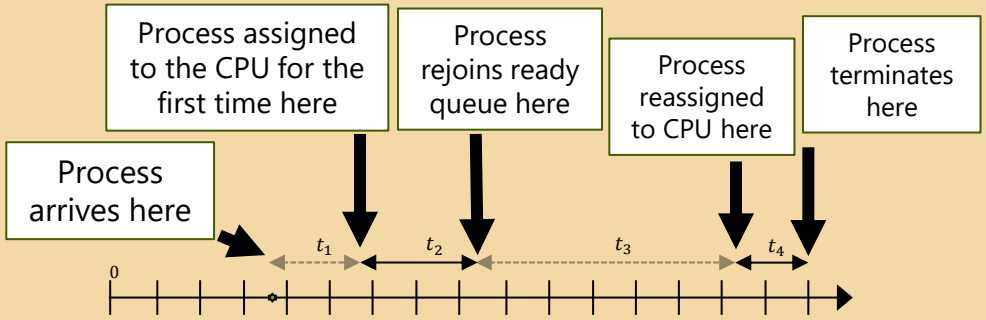
Quantitative Performance Measures

- **User oriented**
  - **Response Time**: time between when the process *arrives in the ready queue* and when it is assigned to the CPU for the first time**;
  - **Turnaround Time**: time between when the process *arrives in the ready queue* and when it finishes executing;
- **System oriented**
  - **Throughput**: is the number of processes that start and complete during the *duration of observing the system*.
    - The duration of observing the system is the time between when the first process arrives and when the last process terminates.
  - **Wait Time**: for each process, the total time the process spends waiting to be assigned to the CPU. Includes the *response time*;
- These definitions assume no overhead in the creation of a process, and negligible time taken to admit a new process to the ready queue;

**response time is often defined from when the process is created, rather than from when the process first enters the ready queue. All the calculations in IN1011 will be from when the process enters the ready queue

Quantitative performance Measures

Response time = t_1
Wait time = Response time + t_3
CPU burst time = $t_2 + t_4$
Turnaround time = CPU burst time + Wait time



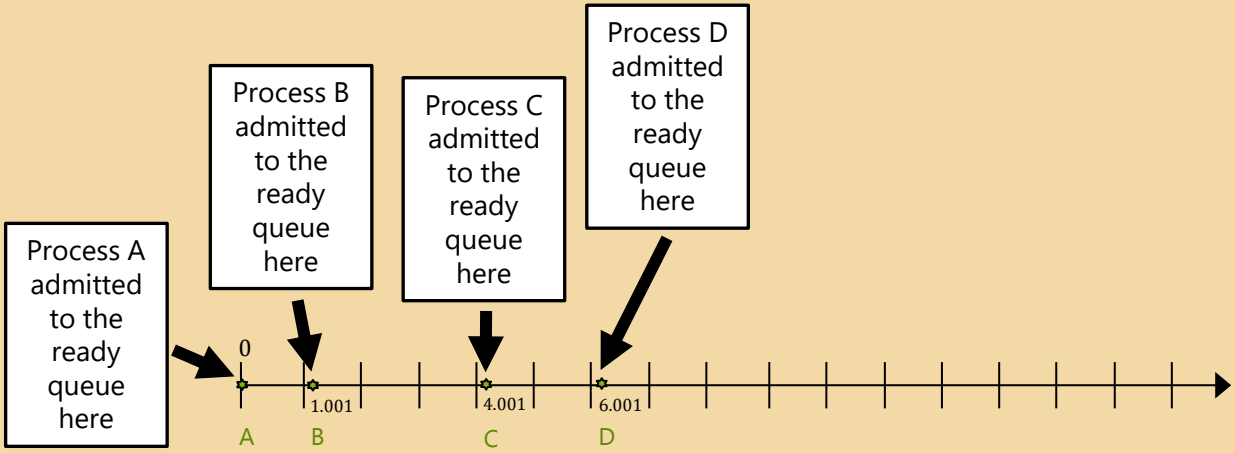
Arrival And Service Times

- Processes A, B, C and D are admitted into the ready queue (by the **long term scheduler**) at times 0, 1.001, 4.001 and 6.001;
- Their respective estimated service times – i.e. how long they need the CPU for – are 3, 6, 4, 2 units of time;
- Service times are another name for CPU burst times;
- Let's assume that:
  - these processes are all CPU bound, with no I/O requests;
  - these processes will successfully complete their executions;
  - no overhead in either assigning processes to the CPU, or freeing the CPU from processes;
  - There are no other processes;
  - processes join the ready queue as soon as they are created
- Using a **first-come first-served** (FCFS) scheduling policy, we now illustrate the order in which these processes are assigned to the CPU (by the **short term scheduler** and **dispatcher**). And how long they execute for, when they are assigned;
- We compute the average response time, average turnaround time, throughput and average wait time resulting from the FCFS policy;

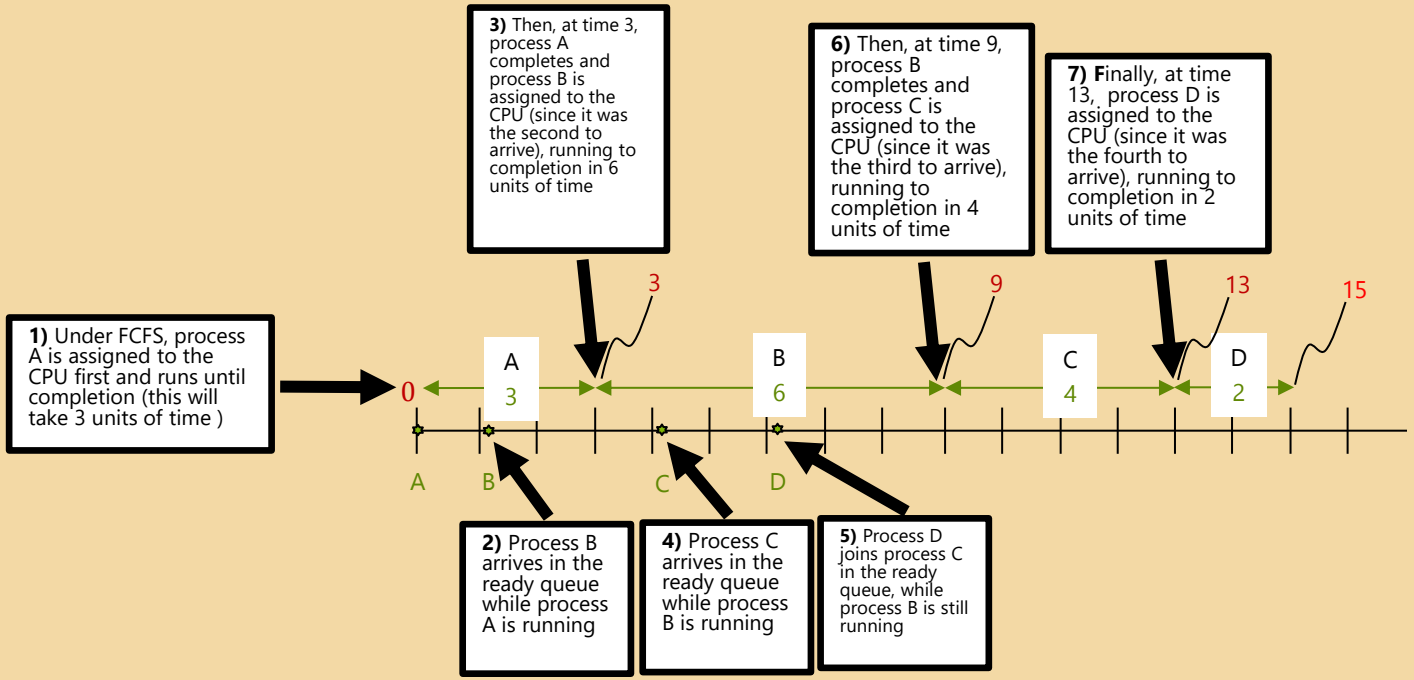
First-Come First-Served (FCFS)

- Selection function: under the FCFS policy and the assumptions about the processes, the steps are:
  - When a process is assigned to the CPU, it runs till it terminates;
  - When the currently running process terminates, the process that has been in the ready queue the longest is assigned to the CPU next;
- Decision mode: non-preemptive – so, a process will run on the CPU until it successfully completes, since we have assumed none of the processes will fail or make I/O requests;

Arrival Times

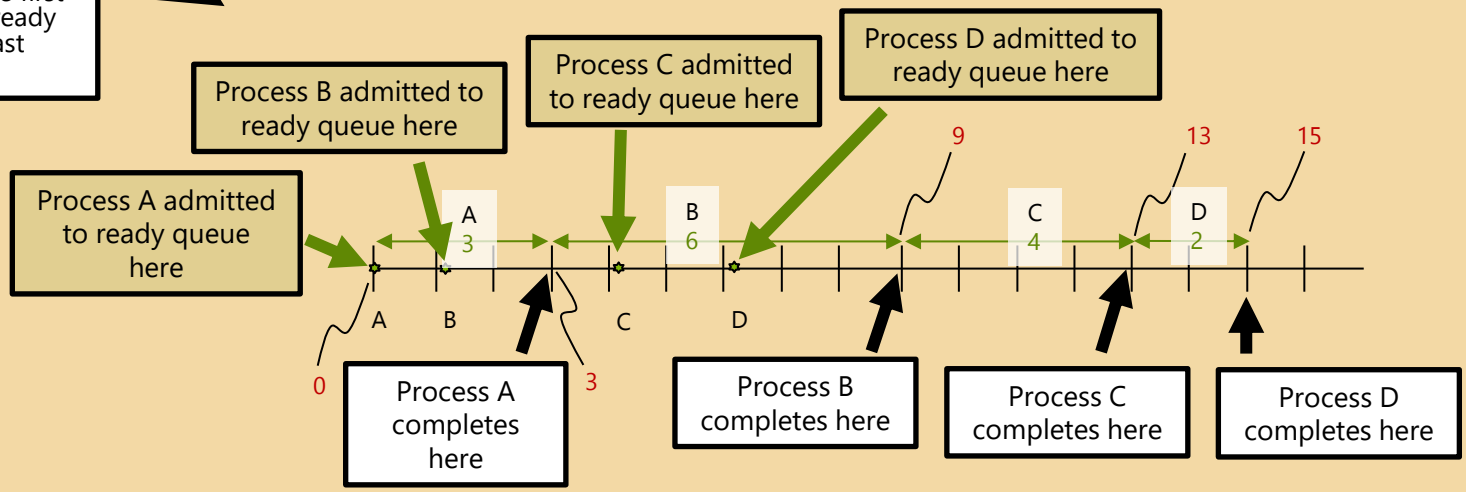


FCFS


1) Under FCFS, process A is assigned to the CPU first and runs until completion (this will take 3 units of time)
3) Then, at time 3, process A completes and process B is assigned to the CPU (since it was the second to arrive), running to completion in 6 units of time
6) Then, at time 9, process B completes and process C is assigned to the CPU (since it was the third to arrive), running to completion in 4 units of time
7) Finally, at time 13, process D is assigned to the CPU (since it was the fourth to arrive), running to completion in 2 units of time
0
A
3
B
6
C
4
D
2
3
9
13
15
A
B
C
D
2) Process B arrives in the ready queue while process A is running
4) Process C arrives in the ready queue while process B is running
5) Process D joins process C in the ready queue, while process B is still running

FCFS : Throughput

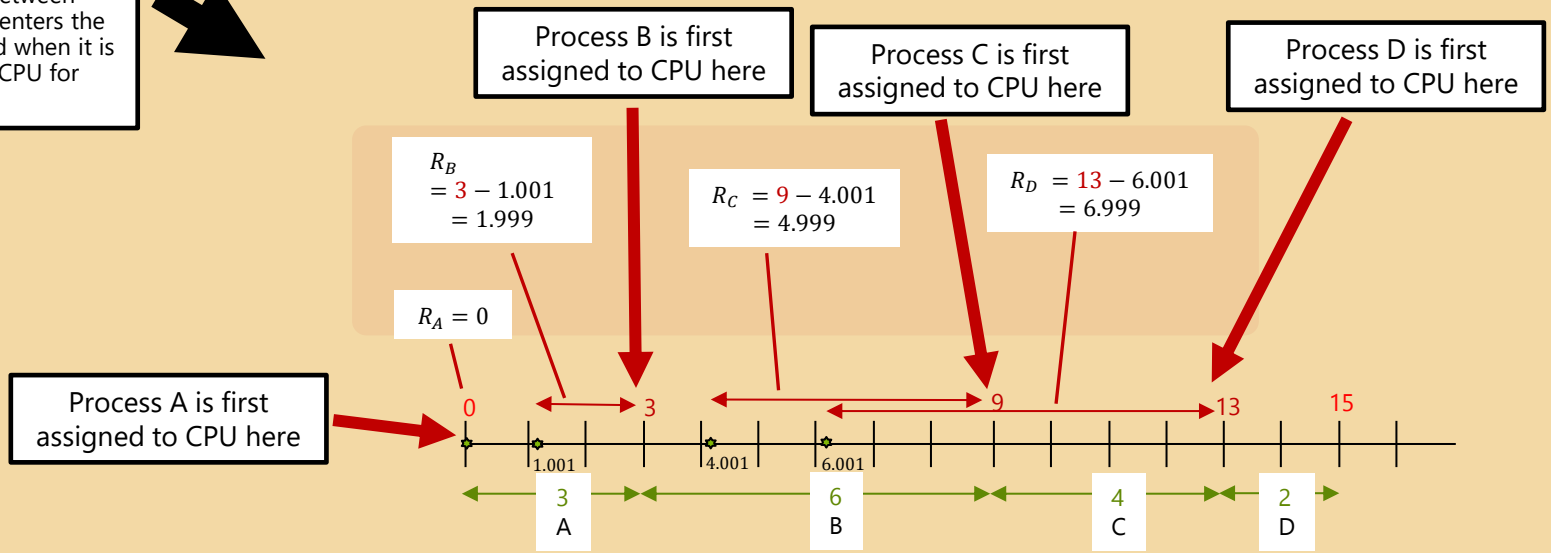
Throughput is the number of processes that start and complete during the *duration of observing the system*. The duration of observing the system is the time between when the first process arrives in the ready queue and when the last process terminates.



Throughput = $\frac{number\ of\ executed\ processes}{duration\ of\ observing\ system} = \frac{4}{15} =$ **0.267 processes per unit of time**

FCFS: Response Time

Response times for each of the processes (denoted R_i for process i). This is the length of time between when a process enters the ready queue and when it is assigned to the CPU for the first time

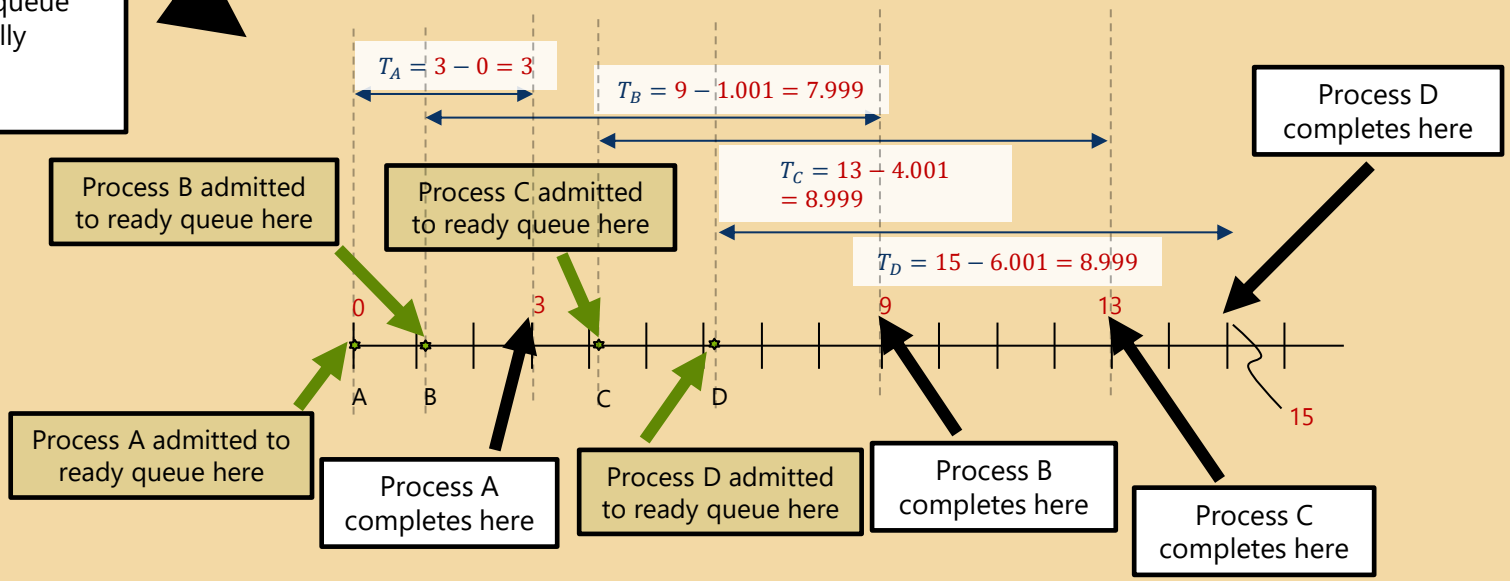


Average response time = $\frac{R_A+R_B+R_C+R_D}{4} = \frac{0+1.999+4.999+6.999}{4} = 3.499$ **time units**

FCFS : Turnaround Time

Turnaround times for each of the processes (denoted T_i for process i). This is the length of time between when a process first enters the ready queue and when it successfully finishes executing

Given the 3 measures we have now computed under FCFS, can you see why it is unnecessary for us to compute "wait times" separately?

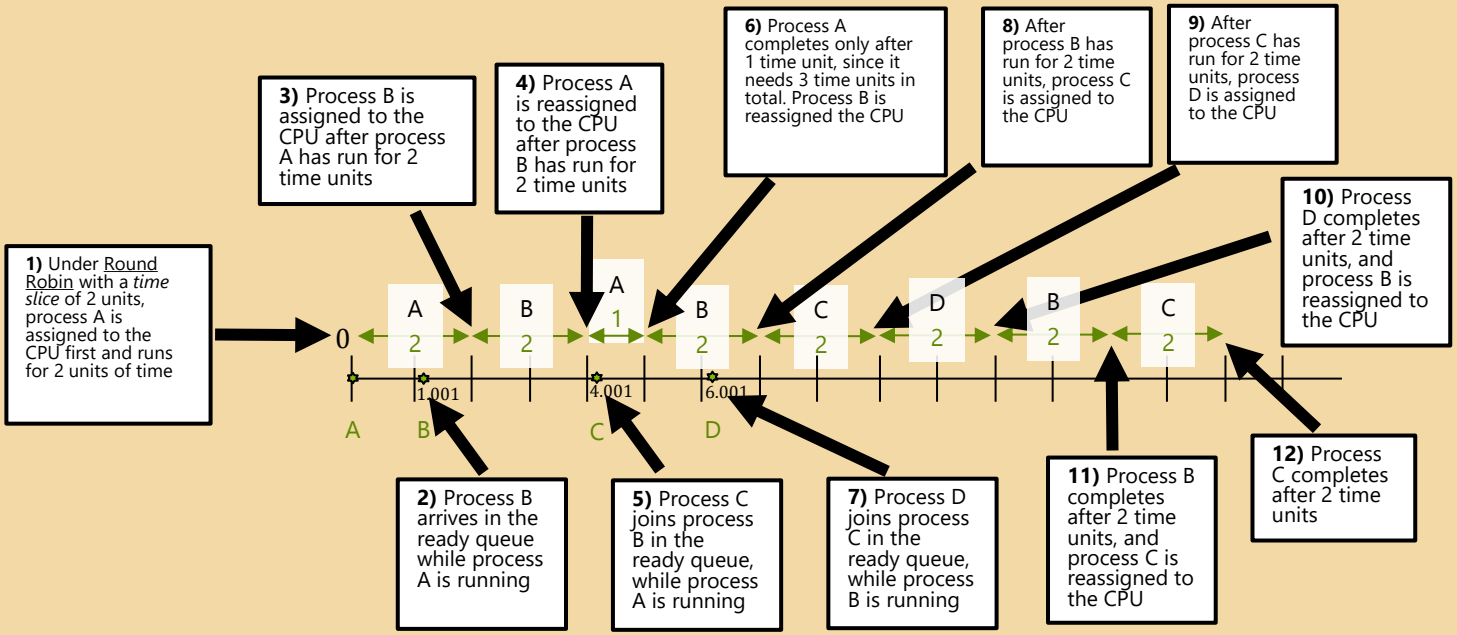


Average turnaround time = $\frac{T_A+T_B+T_C+T_D}{4} = \frac{3+7.999+8.999+8.999}{4} = 7.25$ **time units**

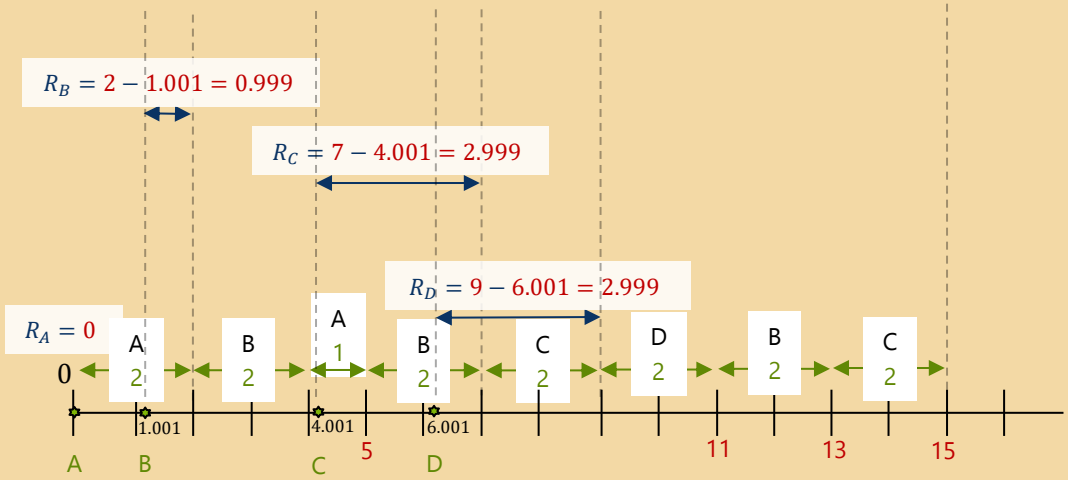
Round Robin

- Decision mode: preemptive, based on hardware clock/timer;
- Selection function: under the Round Robin policy and the assumptions about the processes, the steps are:
  - When a process is assigned to the CPU, it runs for a fixed amount of time (assume this is 2 units of time) called a *quantum* of time, or a *time slice*;
  - If the process does not terminate before the time quantum elapses, the process rejoins the ready queue. Otherwise, it terminates and yields the CPU;
  - the next process assigned to the CPU is the process that has been in the ready queue the longest;

Round Robin

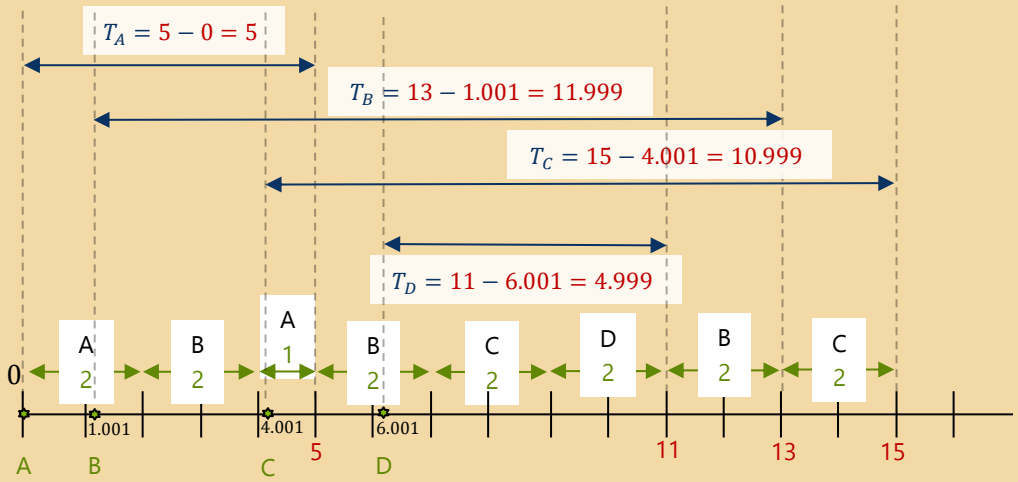

1) Under Round Robin with a time slice of 2 units, process A is assigned to the CPU first and runs for 2 units of time
0
A 2
B 2
A 1
B 2
C 2
D 2
B 2
C 2
1.001
4.001
6.001
A
B
C
D
2) Process B arrives in the ready queue while process A is running
3) Process B is assigned to the CPU after process A has run for 2 time units
4) Process A is reassigned to the CPU after process B has run for 2 time units
5) Process C joins process B in the ready queue, while process A is running
6) Process A completes only after 1 time unit, since it needs 3 time units in total. Process B is reassigned the CPU
7) Process D joins process C in the ready queue, while process B is running
8) After process B has run for 2 time units, process C is assigned to the CPU
9) After process C has run for 2 time units, process D is assigned to the CPU
10) Process D completes after 2 time units, and process B is reassigned to the CPU
11) Process B completes after 2 time units, and process C is reassigned to the CPU
12) Process C completes after 2 time units

Round Robin: Response Time



Average response time = $\frac{R_A+R_B+R_C+R_D}{4} = \frac{0+0.999+2.999+2.999}{4} = 1.75$ **time units**

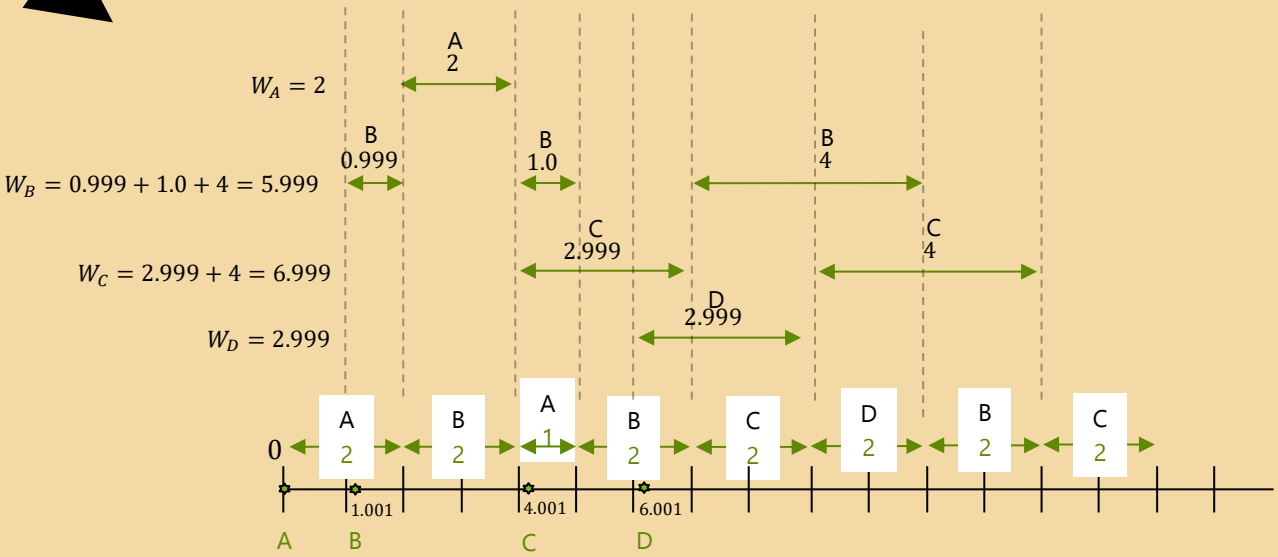
Round Robin: Turnaround Time



Average Turnaround time = $\frac{T_A+T_B+T_C+T_D}{4} = \frac{5+11.999+10.999+4.999}{4} = 8.25$ **time units**

Round Robin: Wait Time

Wait times for each process (denoted as W_i for process i). This is the total time the process spends waiting to be assigned to the CPU.

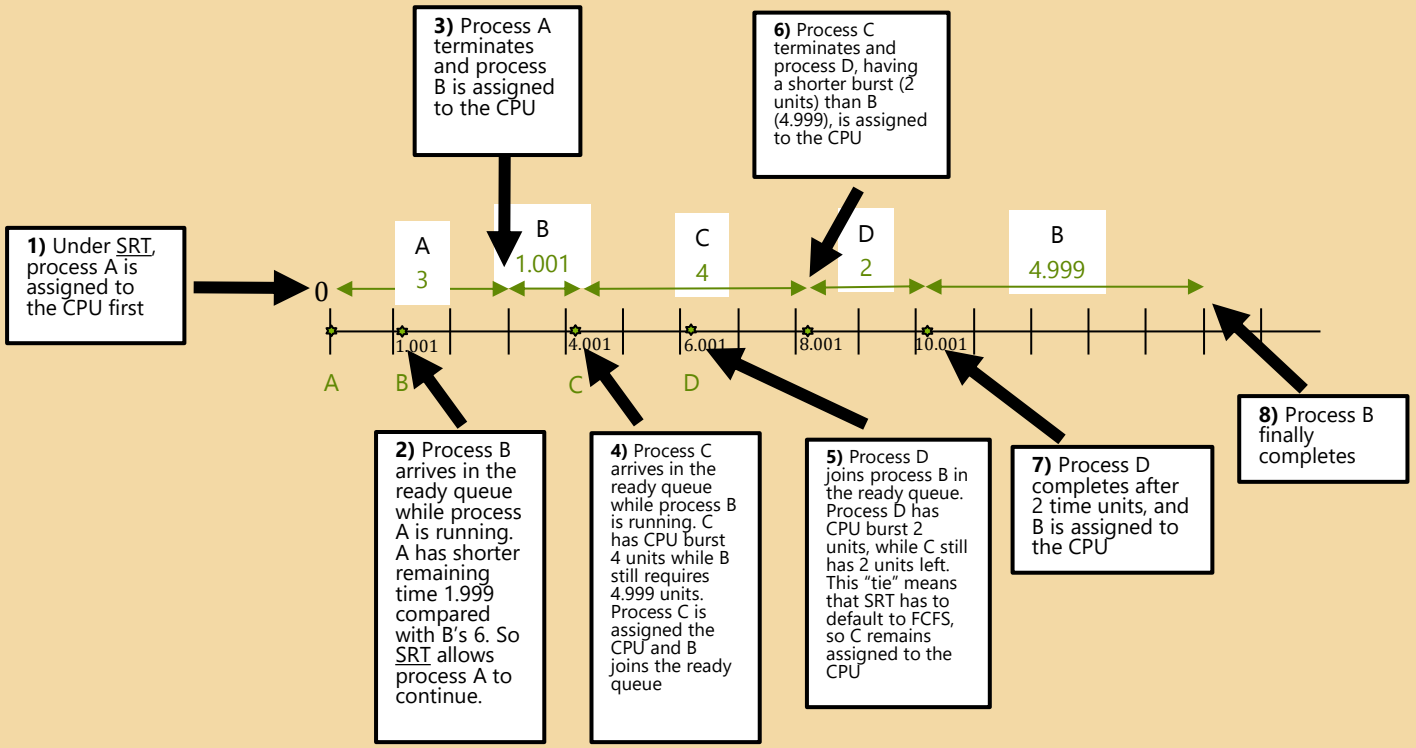


Average wait time = $\frac{W_A+W_B+W_C+W_D}{4} = \frac{2+5.999+6.999+2.999}{4} = 4.499$ **time units**

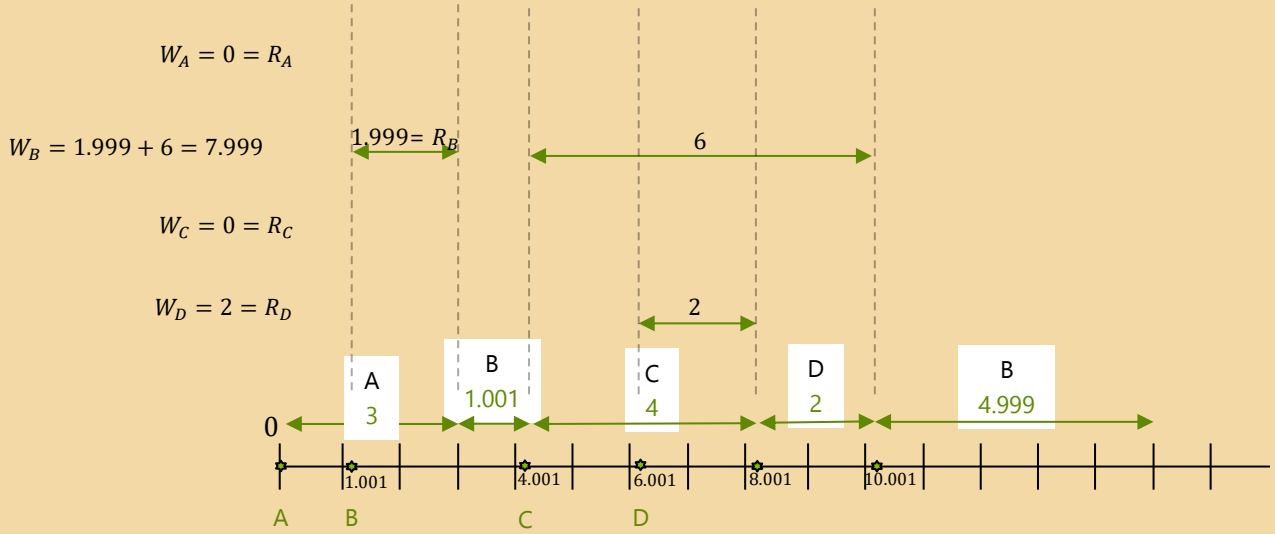
Shortest Remaining Time (SRT)

- Decision mode: preemptive version of **shortest process next** (SPN) – i.e. the CPU could be reassigned whenever a new process enters the ready queue;
- Selection function: under the SRT policy and the assumptions on our 4 processes, the steps are:
  - When a process is assigned to the CPU, it runs until it either terminates or a new process joins the ready queue;
  - When a process terminates or a new process joins the ready queue, the process assigned to the CPU is the process with the shortest remaining CPU burst time;
  - In case of a tie, resort to FCFS

Shortest Remaining Time (SRT)



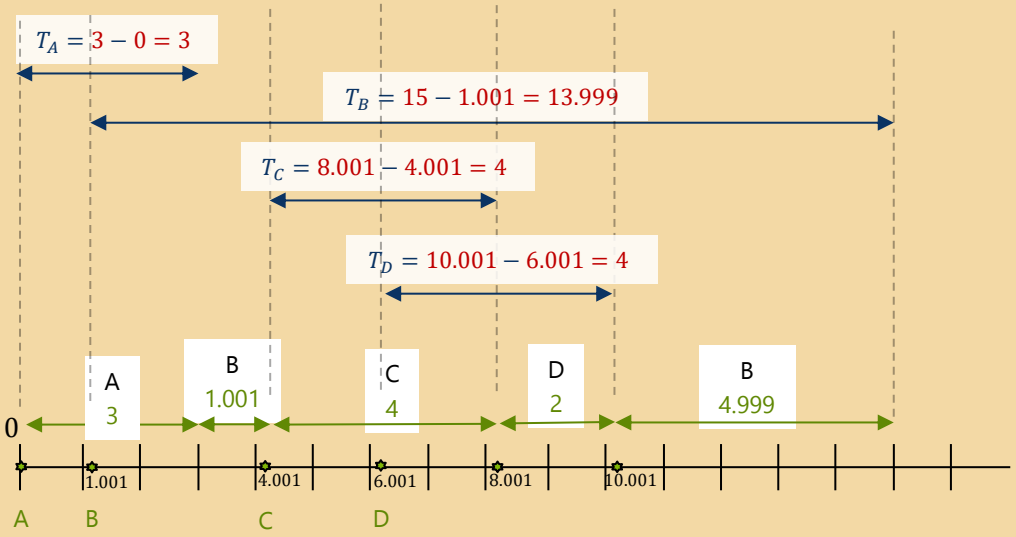
SRT : Wait Time and Response Time



Average wait time = $\frac{W_A+W_B+W_C+W_D}{4} = \frac{0+7.999+0+2}{4} = 2.5$ **time units**

Average response time = $\frac{R_A+R_B+R_C+R_D}{4} = \frac{0+1.999+0+2}{4} = 1$ **time unit**

SRT : Turnaround Time



Average turnaround time = $\frac{T_A+T_B+T_C+T_D}{4} = \frac{3+13.999+4+4}{4} = 6.25$ **time units**

IN1011

Operating Systems

Lecture 04 (part 2): Threads

Questions

- How does the OS help a program take advantage of multiple processors?
- How does the OS support applications that require several tasks to be performed “in parallel”?

Processes and Threads

- There are two aspects to processes we have learnt about:
  - Resource Ownership:
    - a process is assigned an address space containing program code, data, stack, heap and PCB;
    - a process is granted access to I/O devices and files;
    - in granting resources to processes, the OS protects processes from interfering with one another;
  - Scheduling/Execution:
    - A process is scheduled and dispatched by the OS to run on the CPU;
    - A process follows an execution path that may be interleaved with other processes;
    - A process can be in one of 7 (or more) states, having varying dispatching priority, over the course of its execution

Processes and Threads (contd.)

- We now formalise this distinction with yet another abstraction – **threads**.
- The unit of dispatching/execution is a ***thread*** or ***lightweight process***;
- The unit of resource ownership and protection is a ***process*** or ***task***;
- ***Multithreading*** – the ability of an OS to support multiple, concurrent paths of execution within a single process;

Processes and Threads (contd.)

- Threads within a process share resources, and so have more opportunity to "interfere" and "interact" with one another;
- This makes thread synchronisation necessary, and easier, than process synchronisation;

- Each thread has:
  - An execution state (e.g. Running, Ready, Blocked);
  - A saved thread context when not running (e.g. saved values of relevant CPU registers);
  - Associated variables such as program counter, stack pointer and frame pointer;
  - An execution stack;
  - A **thread control block** (TCB);
  - Some per-thread static storage for local variables;
  - Access to the memory and resources of its parent process, shared with all the other threads in the process

Part of a TCB

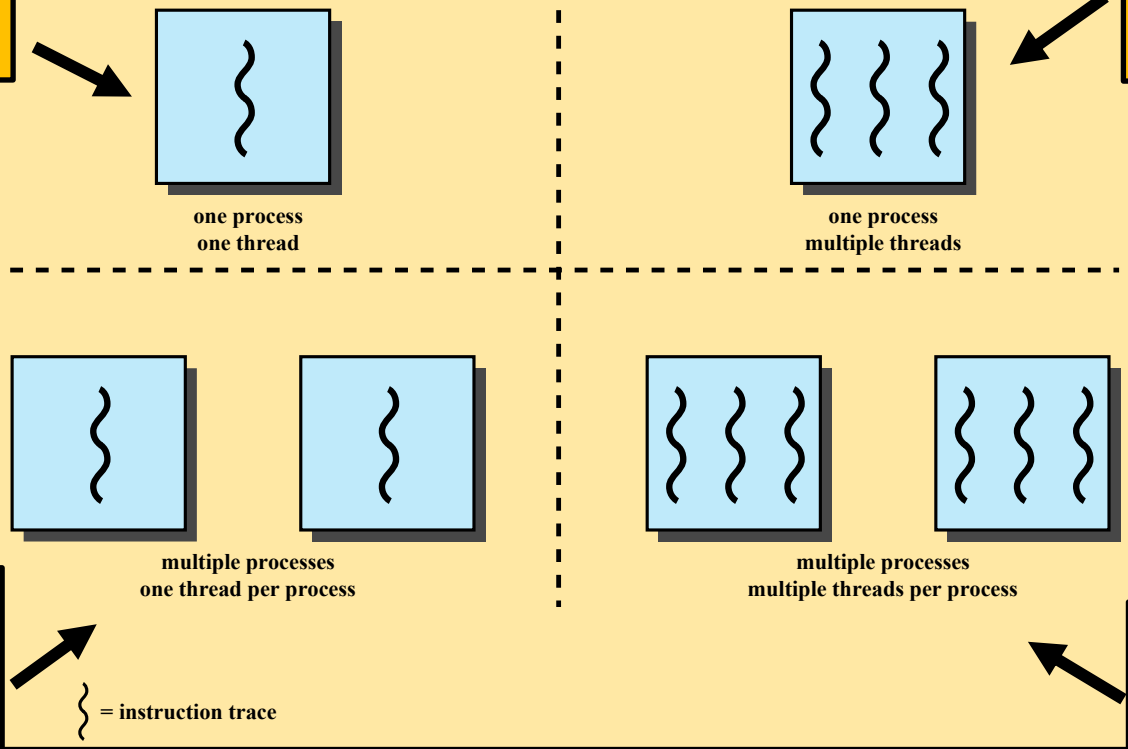
Part of a TCB

Part of a TCB

A TCB primarily contains information necessary for being assigned to, and executing on, the CPU (i.e. an execution or thread context). In contrast, the PCB contains this and more, including information about resource allocation and memory management (i.e. an environment context). Take a peek at part of Linux's PCB/TCB here (https://tinyurl.com/waaust3).

One or More Threads In a Process

A single thread in a single user process. The concept of a thread is not recognized. This is referred to as a single-threaded approach.
MS-DOS is an example



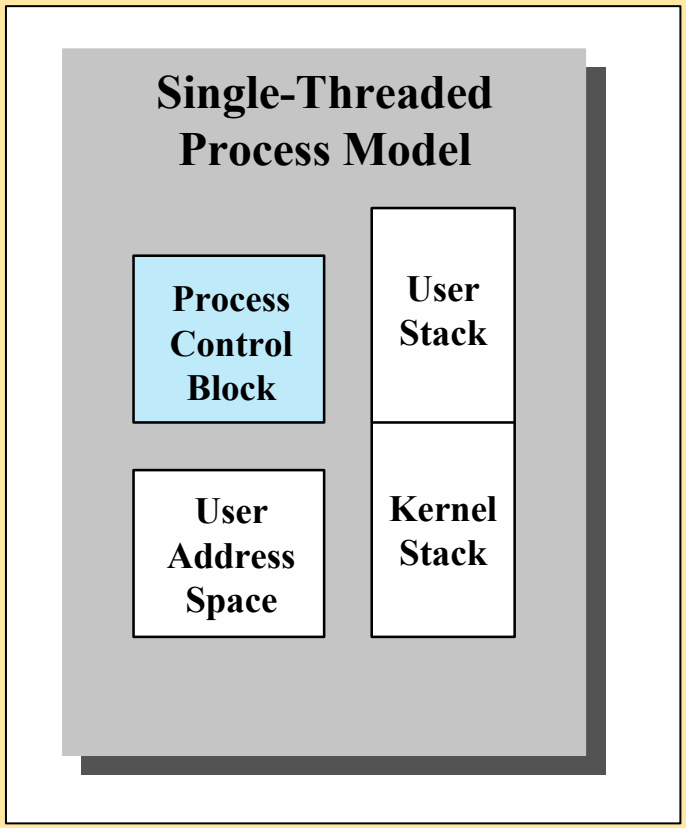
Multithreaded approaches e.g. Java runtime and POSIX compliant applications

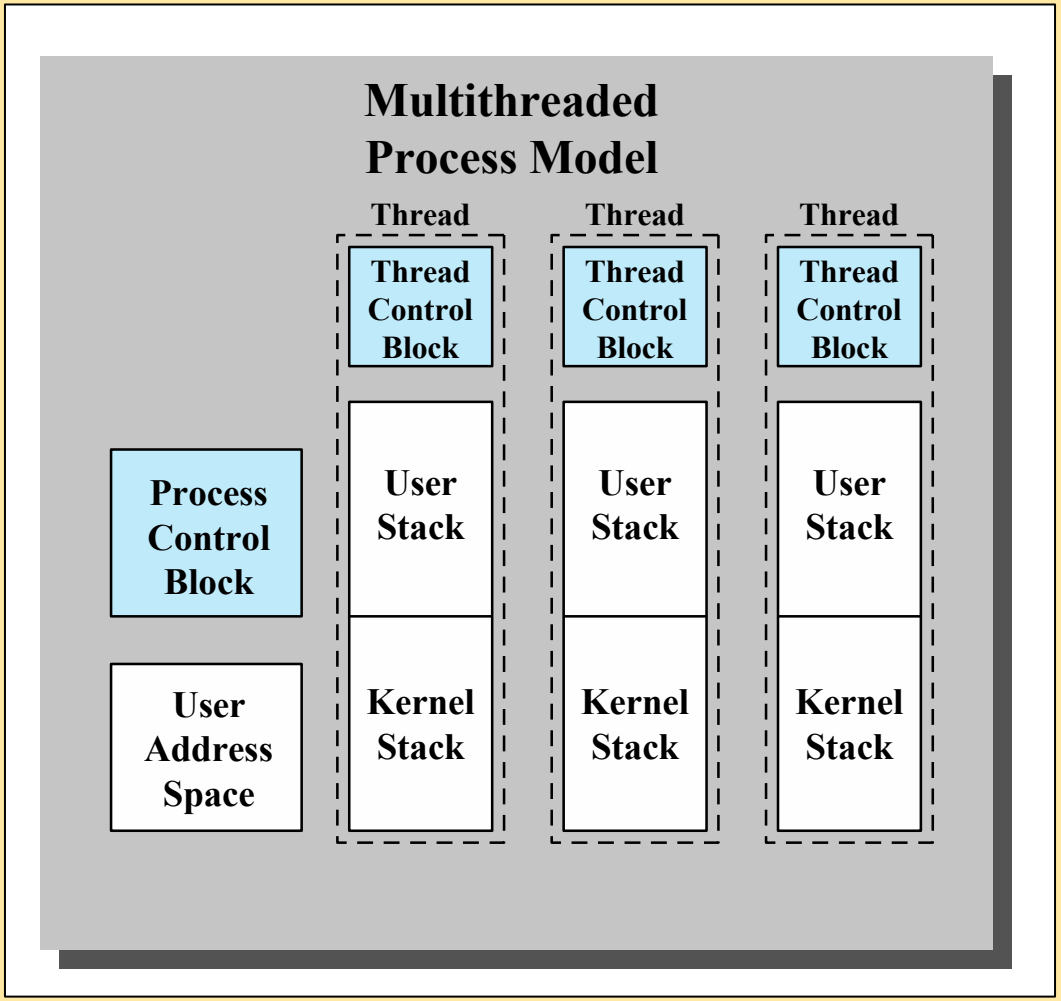
Many other OSes, including older versions of Unix, use multiple processes each with a single thread – so-called heavyweight processes.

Multiple processes, with each of them capable of having multiple threads e.g. Windows, Solaris, Linux, Mac OSX

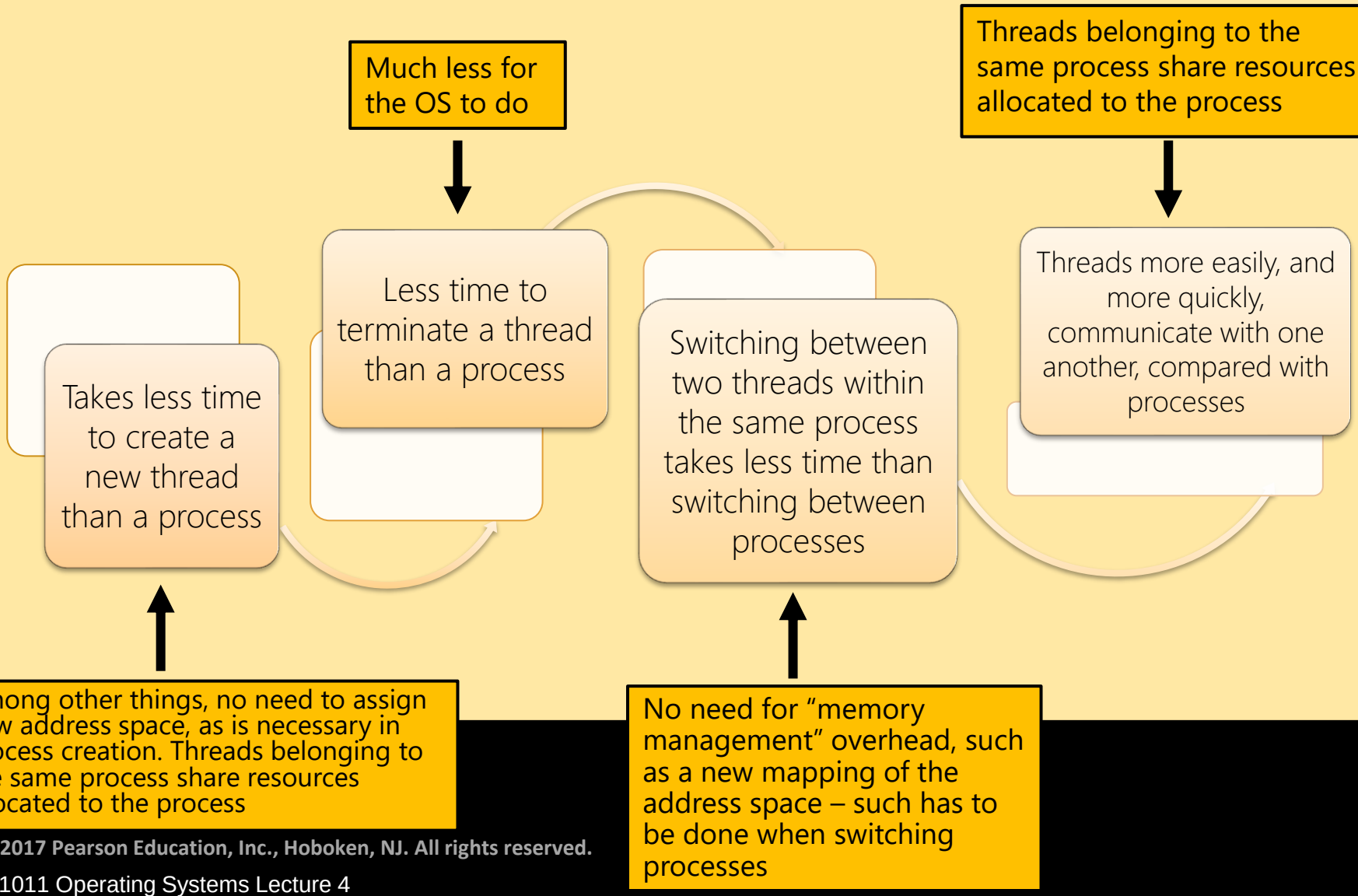
Some Benefits from Multithreading

- Multithreaded native applications;
  - a small number of highly threaded processes (e.g. PC and console games);
- Multiprocess applications
  - many single-threaded processes (e.g. Oracle database and SAP);
- Java applications
  - All applications that use a Java 2 platform, Enterprise Edition application server, can immediately benefit from multicore technology (e.g. Sun's Java application server, IBM's Websphere)
- Multi-instance applications
  - Multiple instances of the application in parallel





Key Benefits of Threads



Thread Use in a Single-User System

- *Foreground* and *background work*, e.g. for a spreadsheet program, one thread could manage the display of menus and read user inputs, while another executes user commands and updates the spreadsheet;
- *Asynchronous processing*, e.g. among the threads in the process for MS word, one thread could periodically save unsaved work;
- *Speed of execution*, e.g. computing with one batch of data (one thread) while reading the next batch (another thread);
- *Modular program structure*, e.g. programs that involve a variety of activities, sources and destinations of input/output, may be implemented as threads (e.g. online pc and video games).

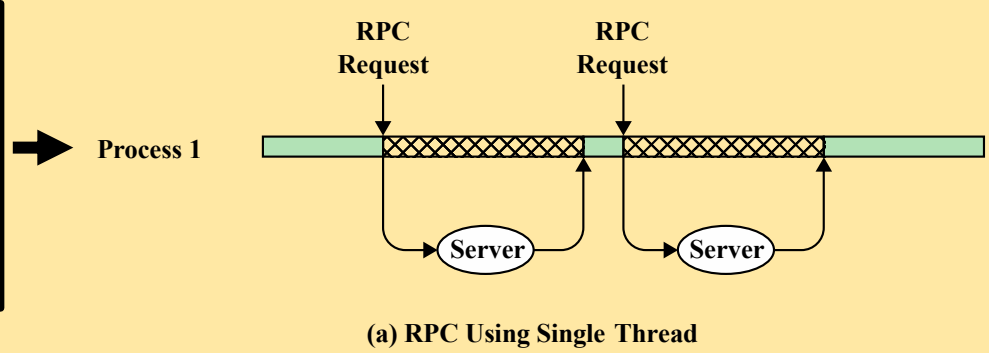
Thread Execution States

- The states of a thread are *running*, *ready* and *blocked*;
- Operations associated with thread state-changes include *spawn* (i.e. creating a thread), *block* (e.g. upon making an I/O request), *unblock* (e.g. upon completion of I/O request), *finish* (e.g. upon completing execution);

Scheduling and Thread States

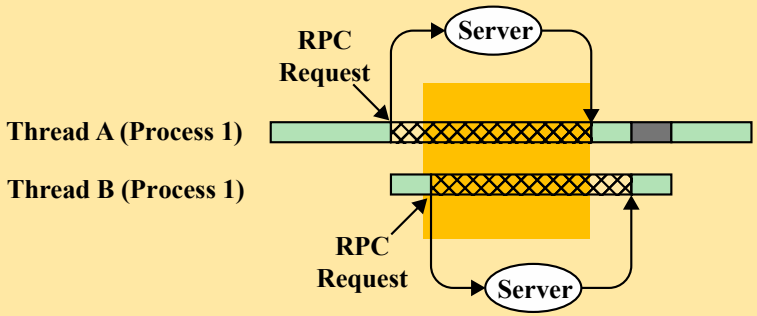
- In an OS that supports threads, scheduling and dispatching is done on a thread basis;
- Most of the thread's state information is maintained in thread-level data structures;
- The state of a process can impact the state of **all** of its threads:
  - Suspending a process involves suspending all threads of the process;
  - Termination of a process terminates all threads of the process;

Program makes independent **remote procedure calls** (RPC) to two different servers, serially. The intention is that the results of the calls will be combined. The time for both requests to complete is the sum of the times it takes for each request to complete.



(a) RPC Using Single Thread

Same as above, but now using application level multiprogramming. Calls are made by two threads in the process, overlapping a significant amount of the time it takes for the servers to service the calls.



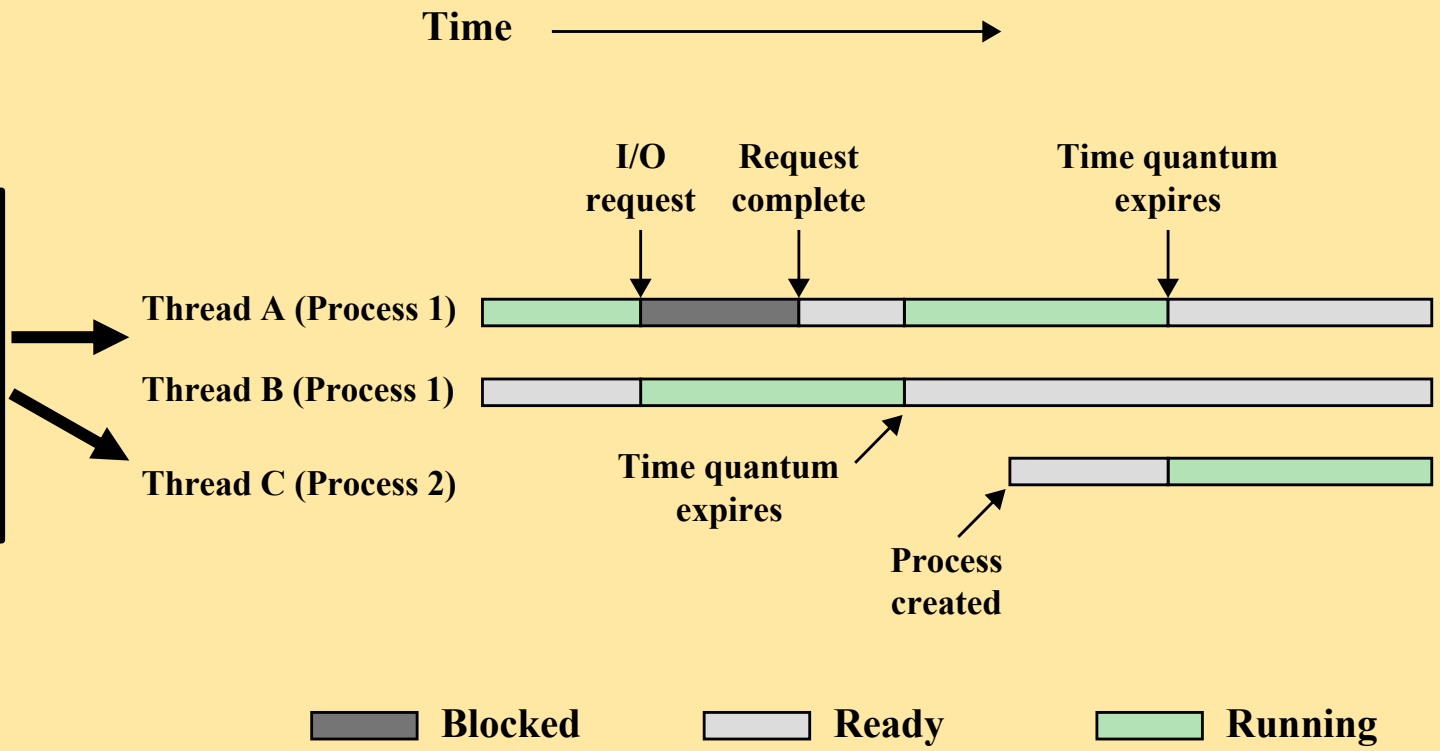
(b) RPC Using One Thread per Server (on a uniprocessor)

Blocked, waiting for response to RPC

Blocked, waiting for processor, which is in use by Thread B

Running

3 threads from 2 processes have their instructions interleaved, as the processes are scheduled to run on the CPU



IN1011

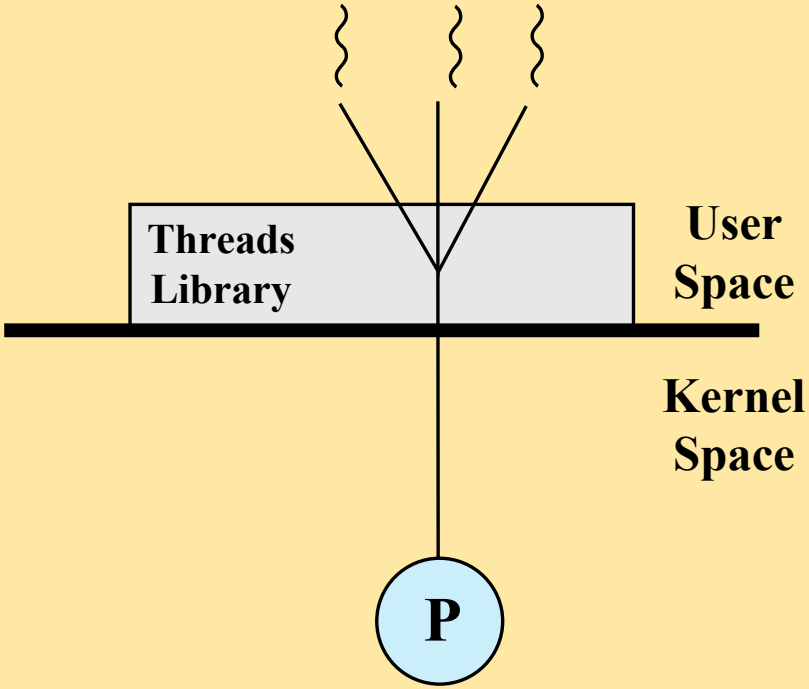
Operating Systems

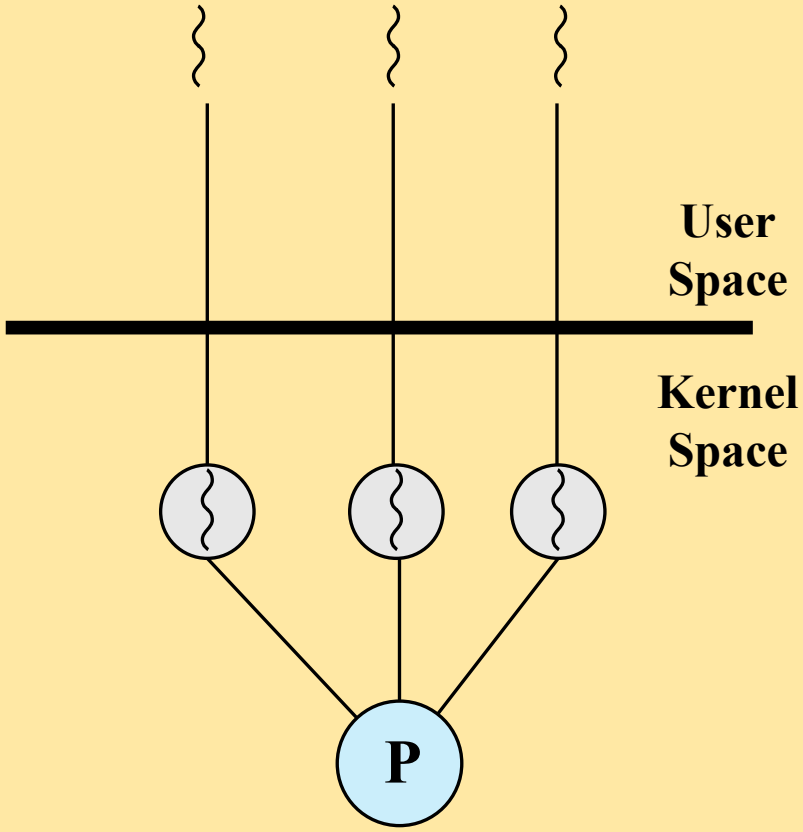
Lecture 04 (part 3): Types of Threads

Types of Threads

- **User-level thread** (ULT)
  - All thread management is done by the application;
  - The kernel is not aware of the existence of the threads;
- **Kernel-level thread** (KLT)
  - Thread management is done by the kernel
  - No thread management code at application level, simply an API to the OS kernel thread facility
  - The kernel maintains context information for the process as a whole, and for the threads thereof;

ULTs and KLTs

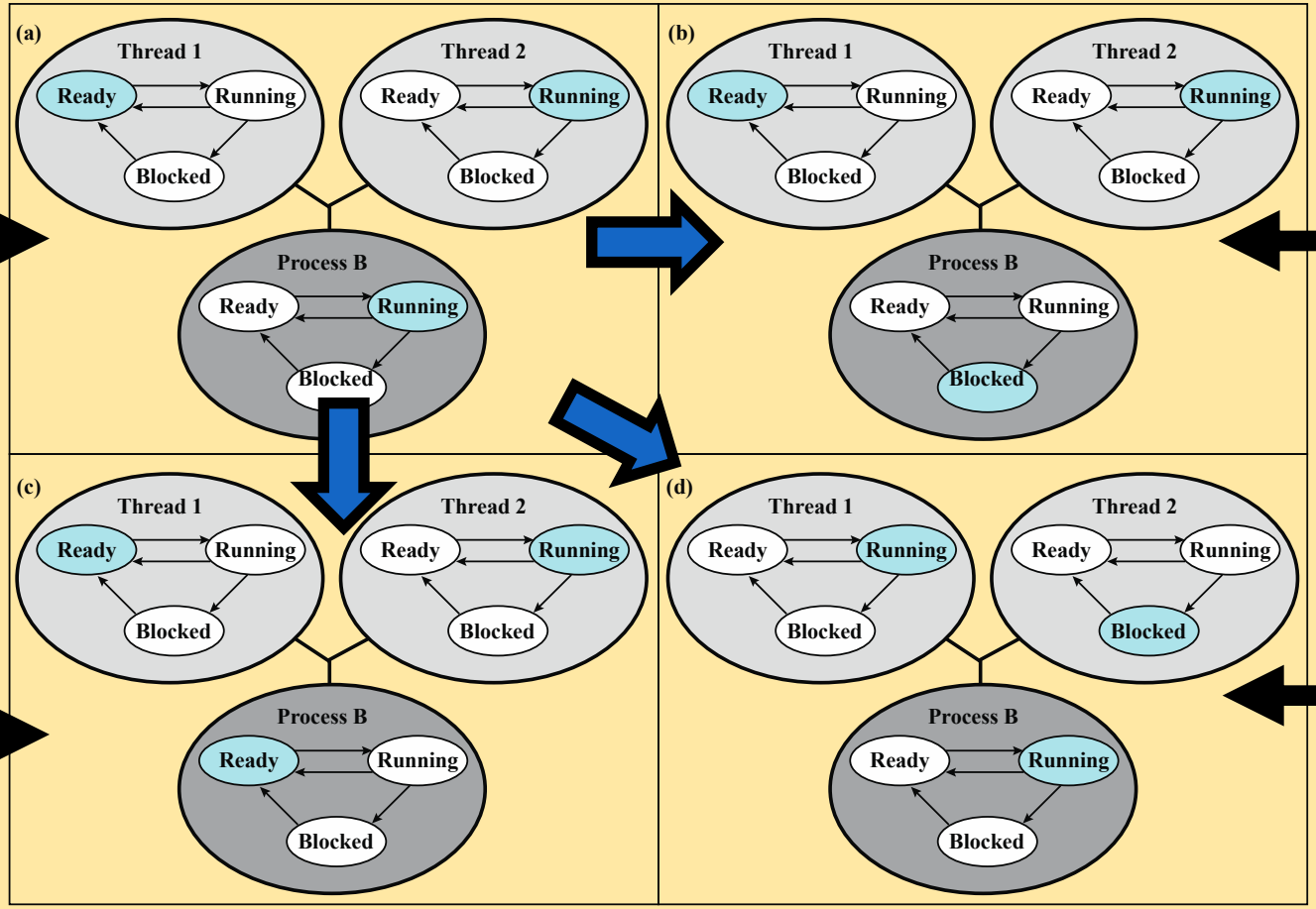




Thread and Process states for ULTs

A process B has been assigned to run on the CPU. It contains 2 ULTs managed by a **thread library**, with the CPU executing "Thread 2"s instructions.

Or, another process is re-assigned to the CPU (e.g. due to a Round Robin scheduling policy), causing process B to re-enter the ready queue. However, "Thread 2" is still in the running state.



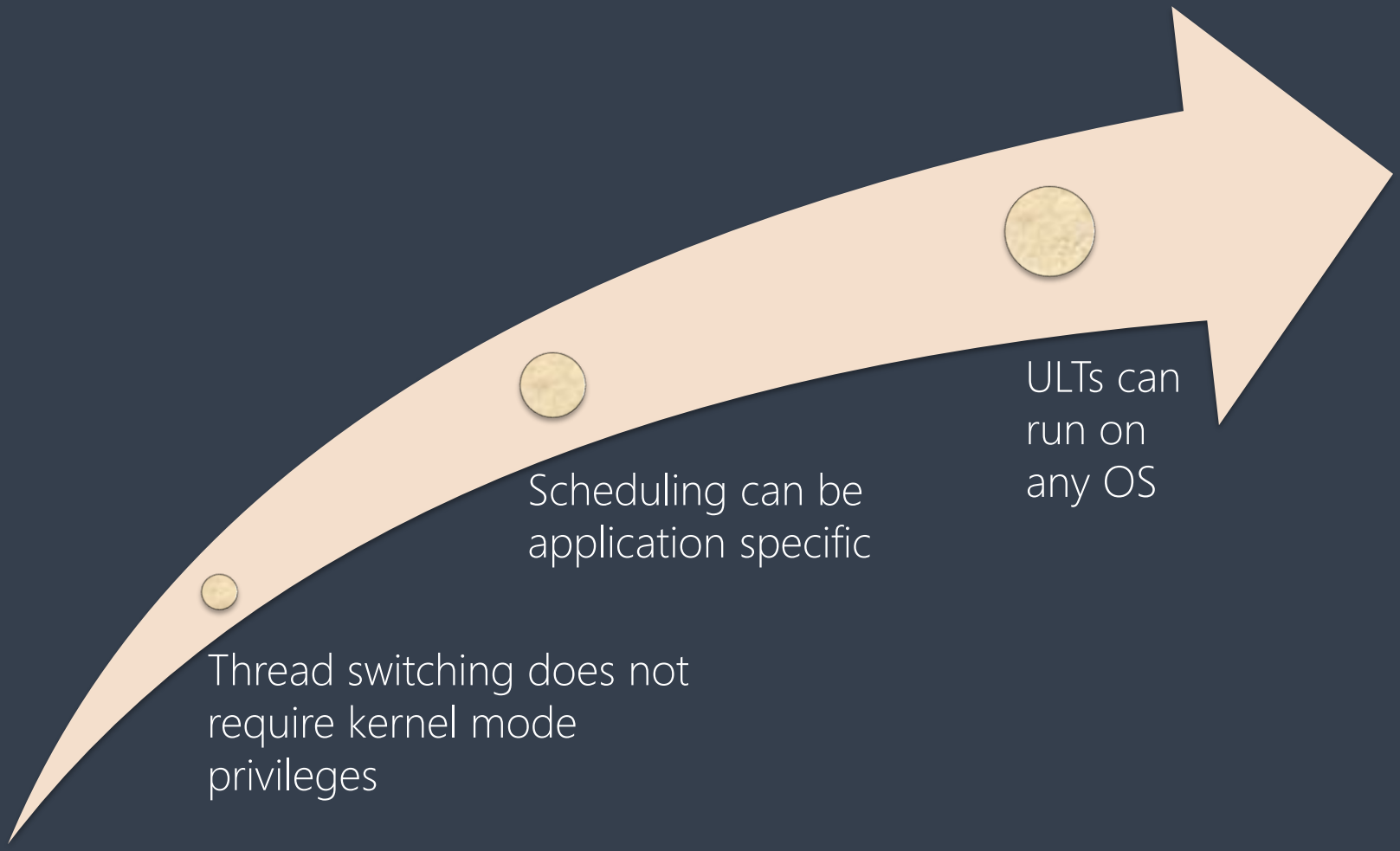
Colored state is current state

Figure 4.6 Examples of the Relationships Between User-Level Thread States and Process States

"Thread 2" makes an I/O request that the process forwards to the OS kernel, resulting in the entire process becoming blocked. The thread library still keeps "Thread 2" in the running state

Or, "Thread 2" becomes blocked, because it requires output from "Thread 1". The CPU executes "Thread 1"s instructions, since process B is still assigned to the CPU

Advantages of ULT



Disadvantages of ULT

- In a typical OS, many system calls result in blocking and incure overhead
  - So, when a ULT makes a system call, **all** threads within a process effectively become blocked;
- In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing (i.e. using multiple CPU cores) for speed improvements;
- Some strategies that attempt to overcome these disadvantages include:
  - "Jacketing" – a method for converting a blocking system call into a non-blocking system call;
  - Writing an application as multiple processes, rather than a single process
    - Drawback with this solution is the increased overhead and added programming complexity;

Advantages of KLT

- Multiprocessing: the kernel can simultaneously schedule multiple threads from the same process on multiple processors
- If one thread in a process is blocked, the kernel can schedule another thread of the same process
- Kernel routines, themselves, can be multithreaded

Disadvantages of KLT

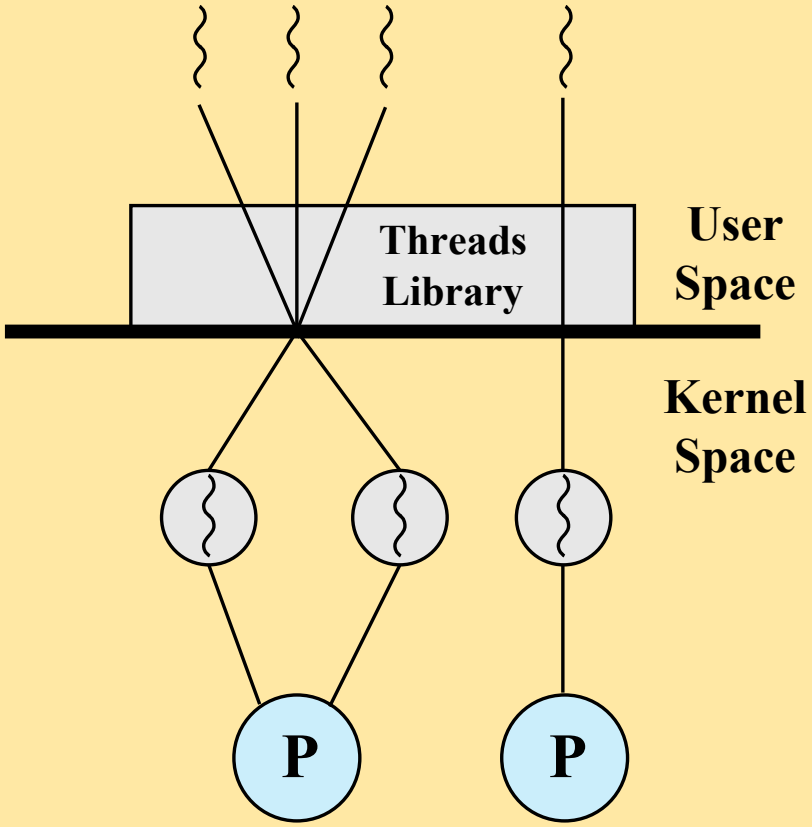
- Transfer of control from one thread to another, within the same process, requires mode switches

Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

Table 4.1
Thread and Process Operation Latencies (μs)

Combined Approaches

- Thread creation is done completely in the user space, as well as most of the scheduling and synchronization (e.g. Solaris OS);
- However, multiple ULTs from a single application are mapped into a smaller (or equal) number of KLTs;
- This allows for multiprocessing;



Thread-Process Relationships

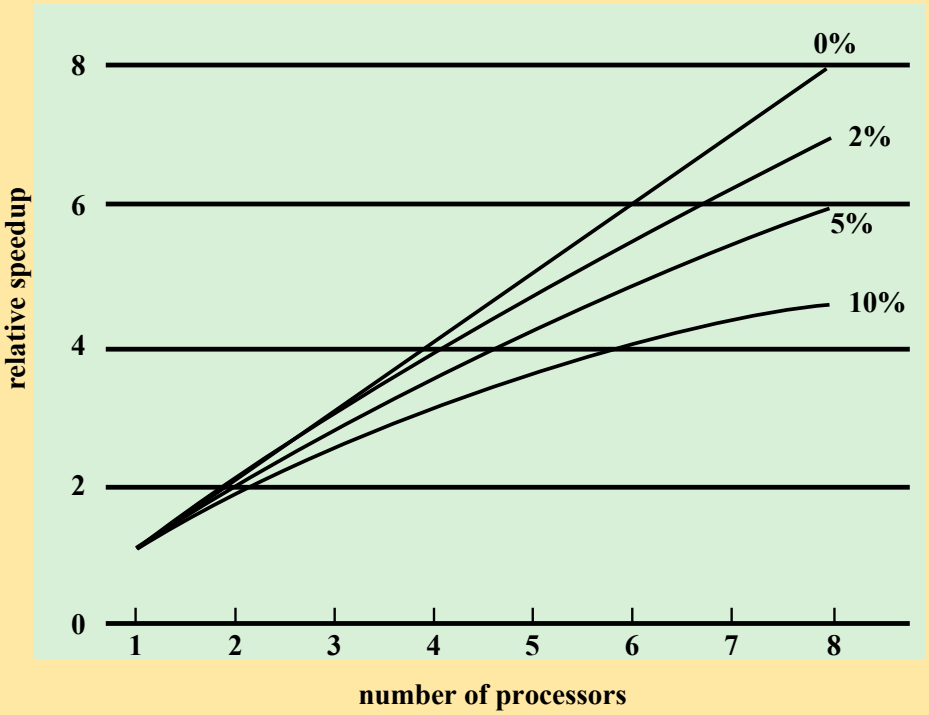
Threads:Processes	Description	Example Systems
1:1	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
M:1	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH
1:M	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra (Clouds), Emerald
M:N	Combines attributes of M:1 and 1:M cases.	TRIX

Why Aren't All Programs Multithreaded?

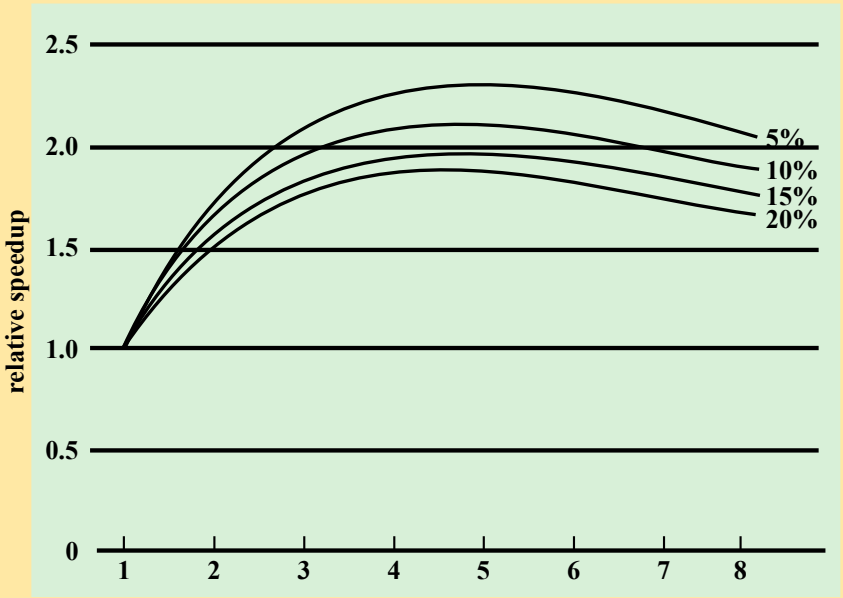
Amdahl's "law"

$$\frac{\text{(Time to execute serially)}}{\text{(Time to execute in parallel)}} = \frac{1}{(1-f) + f/N}$$

N is the number of CPUs/processors. And ***f*** is the proportion of time spent by the program in performing parallelisable tasks

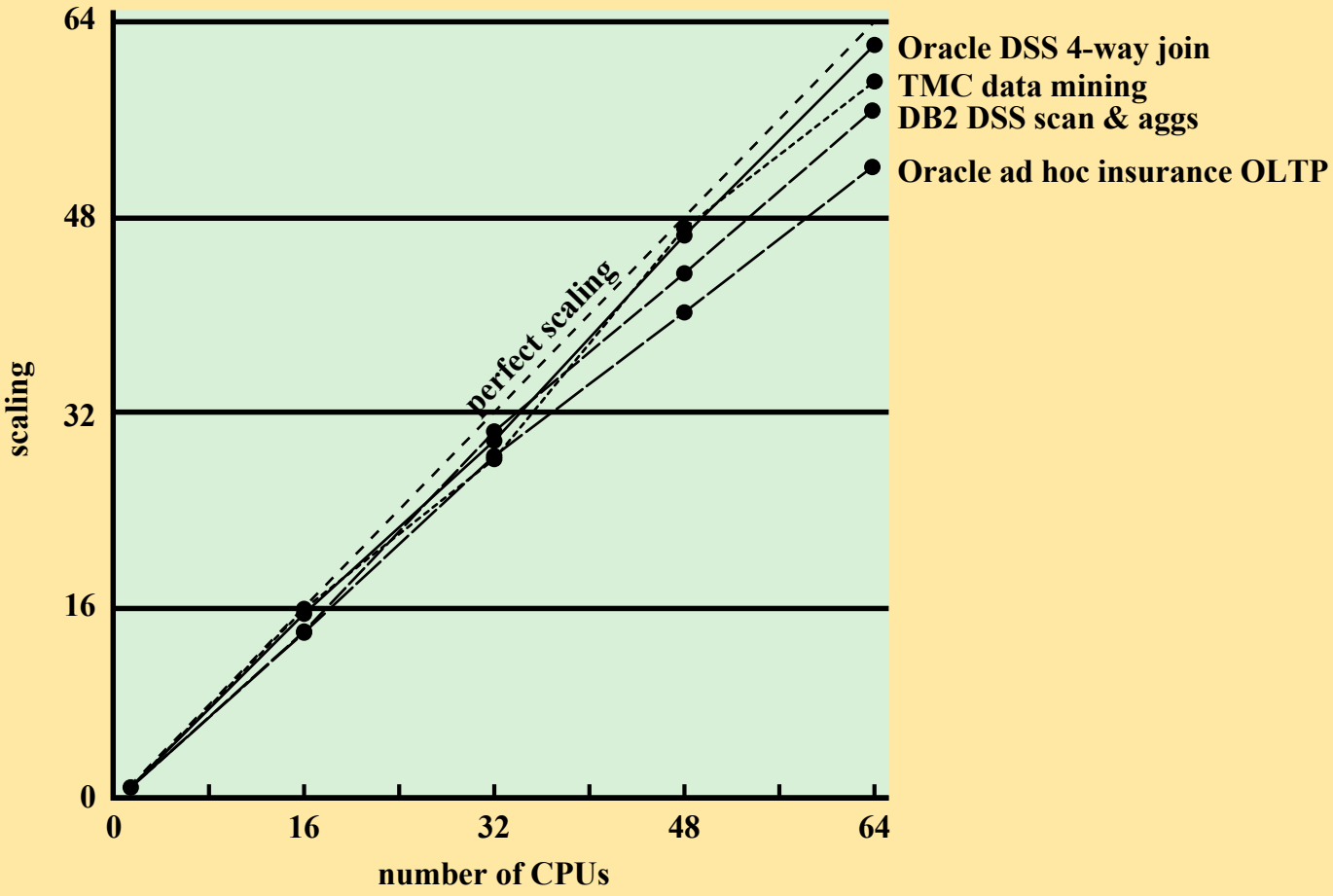


(a) Speedup with 0%, 2%, 5%, and 10% sequential portions



(b) Speedup with overheads

Servers do quite well, though!



IN1011

Operating Systems

Lecture 04 (part 4): Unix Thread API Demo

IN1011

Operating Systems

Lecture 04 (part 4): Unix Thread API Demo

Unix thread API demo

- Unix provides system calls – for spawning and controlling threads – to C/C++ application developers that use POSIX compliant API calls in their code;
- These include the methods:
  - pthread_create for spawning threads;
  - pthread_join for "parent" processes to wait for spawned threads to terminate before continuing execution;
  - The pair of pthread_mutex_lock and pthread_mutex_unlock for concurrency control;
- The demo code illustrating these methods is on Moodle;

Call to *pthread_exit* terminates the thread

Call to *pthread_create* within loop creates 10 threads, each of them running the do_some_work program to say "Hello World" and to perform 1e7 additions.

```
#include <iostream>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>

using namespace std;

#define NUM_OF_THREADS 10

int global_data = 0;

void* do_some_work(void* tpid)
{
    cout << "Hello world! Greetings from thread #" << (long)tpid << "\n";

    for (int i = 0; i < 1e7; ++i)
    {
        global_data = global_data + 1;
    }

    cout << "Thread #" << (long)tpid << " thinks the value of the global_data is now " << global_data << "\n";

    pthread_exit(NULL);
}



int main()
{
    /* 10 threads are created and print Hello World! */
    pthread_t Threads[NUM_OF_THREADS];

    int status;

    for (long i = 0; i < NUM_OF_THREADS; ++i)
    {
        status = pthread_create(&Threads[i], NULL, do_some_work, (void*)i);

        if (status) {
            cout << "ERROR; return code from pthread_create() is " << status << "\n";
            exit(-1);
        }
    }

    return 0;
}
```

Output of Example Code 1

```
Hello world! Greetings from thread #1
```

```cpp
#include <iostream>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>

using namespace std;

#define NUM_OF_THREADS 10


int global_data = 0;

void* do_some_work(void* tpid)
{
    cout << "Hello world! Greetings from thread #" << (long)tpid << "\n";

    for (int i = 0; i < 1e7; ++i)
    {
        global_data = global_data + 1;
    }

    cout << "Thread #" << (long)tpid << " thinks the value of the global_data is now " << global_data << "\n";

    pthread_exit(NULL);

}



int main()
{
    /* 10 threads are created and print Hello World! */
    pthread_t Threads[NUM_OF_THREADS];

    int status;

    for (long i = 0; i < NUM_OF_THREADS; ++i)
    {
        status = pthread_create(&Threads[i], NULL, do_some_work, (void*)i);

        if (status) {
            cout << "ERROR; return code from pthread_create() is " << status << "\n";
            exit(-1);
        }
    }

    for (long i = 0; i < NUM_OF_THREADS; ++i)
    {
        status = pthread_join(Threads[i], NULL);
        if (status) {
            cout << "ERROR; return code from pthread_join() is " << status << "\n";
            exit(-1);
        }
    }

    return 0;
}
```

Call to *pthread_join* causes the calling process to wait for the specified thread to terminate

Output of Example Code 2

```
Hello world! Greetings from thread #2
Hello world! Greetings from thread #5
Hello world! Greetings from thread #6
Hello world! Greetings from thread #7
Hello world! Greetings from thread #8
Hello world! Greetings from thread #9
Hello world! Greetings from thread #1
Hello world! Greetings from thread #4
Hello world! Greetings from thread #3
Hello world! Greetings from thread #0
Thread #2 thinks the value of the global_data is now 12065217
Thread #1 thinks the value of the global_data is now 16981617
Thread #6 thinks the value of the global_data is now 16963433
Thread #7 thinks the value of the global_data is now 16867248
Thread #5 thinks the value of the global_data is now 18378001
Thread #9 thinks the value of the global_data is now 18315323
Thread #8 thinks the value of the global_data is now 18969753
Thread #0 thinks the value of the global_data is now 19397106
Thread #4 thinks the value of the global_data is now 18708011
Thread #3 thinks the value of the global_data is now 20983642
```

Calls to *pthread_mutex_lock* and *pthread_mutex_unlock* causes threads to execute this code block on a first come first served basis, mutually exclusively.

```cpp
#include <iostream>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>

using namespace std;

#define NUM_OF_THREADS 10

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

int global_data = 0;

void* do_some_work(void* tpid)
{
    pthread_mutex_lock(&mutex1);
    cout << "Hello world! Greetings from thread #" << (long)tpid << "\n";

    for (int i = 0; i < 1e7; ++i)
    {
        global_data = global_data + 1;
    }

    cout << "Thread #" << (long)tpid << " thinks the value of the global_data is now " << global_data << "\n";

    pthread_mutex_unlock(&mutex1);

    pthread_exit(NULL);
}



int main()
{
    /* 10 threads are created and print Hello World! */
    pthread_t Threads[NUM_OF_THREADS];

    int status;

    for (long i = 0; i < NUM_OF_THREADS; ++i)
    {
        status = pthread_create(&Threads[i], NULL, do_some_work, (void*)i);

        if (status) {
            cout << "ERROR; return code from pthread_create() is " << status << "\n";
            exit(-1);
        }
    }

    for (long i = 0; i < NUM_OF_THREADS; ++i)
    {
        status = pthread_join(Threads[i], NULL);
        if (status) {
            cout << "ERROR; return code from pthread_join() is " << status << "\n";
            exit(-1);
        }
    }

    return 0;
}
```

Output of Example Code 3

```
Hello world! Greetings from thread #7
Thread #7 thinks the value of the global_data is now 10000000
Hello world! Greetings from thread #0
Thread #0 thinks the value of the global_data is now 20000000
Hello world! Greetings from thread #1
Thread #1 thinks the value of the global_data is now 30000000
Hello world! Greetings from thread #8
Thread #8 thinks the value of the global_data is now 40000000
Hello world! Greetings from thread #9
Thread #9 thinks the value of the global_data is now 50000000
Hello world! Greetings from thread #6
Thread #6 thinks the value of the global_data is now 60000000
Hello world! Greetings from thread #5
Thread #5 thinks the value of the global_data is now 70000000
Hello world! Greetings from thread #4
Thread #4 thinks the value of the global_data is now 80000000
Hello world! Greetings from thread #3
Thread #3 thinks the value of the global_data is now 90000000
Hello world! Greetings from thread #2
Thread #2 thinks the value of the global_data is now 100000000
```

```cpp
#include <iostream>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>

using namespace std;

#define NUM_OF_THREADS 10

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

int global_data = 0;

void* do_some_work(void* tpid)
{
    pthread_mutex_lock(&mutex1);
    cout << "Hello world! Greetings from thread #" << (long)tpid << "\n";

    for (int i = 0; i < 1e7; ++i)
    {
        global_data = global_data + 1;
    }

    cout << "Thread #" << (long)tpid << " thinks the value of the global_data is now " << global_data << "\n";

    pthread_mutex_unlock(&mutex1);

    pthread_exit(NULL);
}



int main()
{
    /* 10 threads are created and print Hello World! */
    pthread_t Threads[NUM_OF_THREADS];

    int status;

    for (long i = 0; i < NUM_OF_THREADS; ++i)
    {
        status = pthread_create(&Threads[i], NULL, do_some_work, (void*)i);

        if (status) {
            cout << "ERROR; return code from pthread_create() is " << status << "\n";
            exit(-1);
        }

        status = pthread_join(Threads[i],NULL);
        if (status) {
            cout << "ERROR; return code from pthread_join() is " << status << "\n";
            exit(-1);
        }
    }

    return 0;
}
```

Calls to *pthread_join* now performed within the loop, after each thread is created. Upon each call, the calling process does not proceed until the indicated thread has completed.

Output of Example Code 4

```
Hello world! Greetings from thread #0
Thread #0 thinks the value of the global_data is now 10000000
Hello world! Greetings from thread #1
Thread #1 thinks the value of the global_data is now 20000000
Hello world! Greetings from thread #2
Thread #2 thinks the value of the global_data is now 30000000
Hello world! Greetings from thread #3
Thread #3 thinks the value of the global_data is now 40000000
Hello world! Greetings from thread #4
Thread #4 thinks the value of the global_data is now 50000000
Hello world! Greetings from thread #5
Thread #5 thinks the value of the global_data is now 60000000
Hello world! Greetings from thread #6
Thread #6 thinks the value of the global_data is now 70000000
Hello world! Greetings from thread #7
Thread #7 thinks the value of the global_data is now 80000000
Hello world! Greetings from thread #8
Thread #8 thinks the value of the global_data is now 90000000
Hello world! Greetings from thread #9
Thread #9 thinks the value of the global_data is now 100000000
```