Queensland University of Technology

CAB403
Systems Programming

Professor Timothy Chappell

Dinal Atapattu

October 26, 2023

Contents

Chapter 1

Introduction to Operating Systems

1.1 Operating System Structures

1.1.1 Operating System Services

- Operating systems provide an environment for execution of programs and services to programs and users
- Operating System services provides functions that are helpful to the user
    - User Interface - Almost all opeating systems have a user interface (UI)
        * Graphical (GUI)
        * Command Line (CLI)
        * Batch
    - Program Execution - The system must be able to load a program into memory and run that program, end execution, either normally or abnormally (indicating error)
    - I/O operations - A running program may require I/O, which involves either a file or I/O device

Chapter 2

Operating System Structures

Chapter 3

Processes

Chapter 4

Threads

A thread is a fundamental unit of CPU utilization, which forms the basis of multithreaded computer systems. Many modern applications are multithreaded, with threads running within an application. An application can divide tasks to seperate threads such as

- Updating the display
- Fetching data
- Spell checking
- Answering network requests

Thread creation, as opposed to process creation is lightweight, can simplify code, and increase efficiency. For this reason, kernels are often multithreaded.
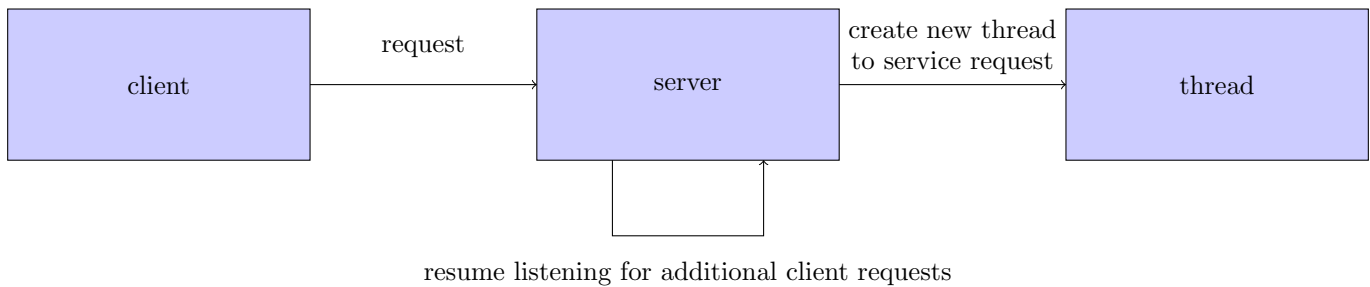
Figure 4.1: Multithreaded Server Architecture

Multithreading has the following benefits

- **Responsiveness**

  May allow continued execution if part of a process is blocked, important with user interfaces.

- **Resource Sharing**

  Threads share resources of process, easier to manage than shared memory or message parsing.

- **Economy**

  Cheaper than process creation, thread switching has lower overhead than context switching.

- **Scalability**

  Processes can take advantage of a multiprocessor architecture

4.1   Multicore Programming

In the early history of computer design, in order to combat the need for increased computing performance, single-CPU systems evoled into mult-CPU systems. Later, this evolved into including multiple compute cores on a single processing chip, where each core appears as a seperate CPU to the operating systems. These systems are defined as **multicore**.

4.1.1 Programming Challenges

The trend towards multicore systems continually places pressure on system designers and programmers to make better use of multiple compute cores. Designers of operating systems must write scheduling algorithms that use multiple processing cores to allow parallel execution
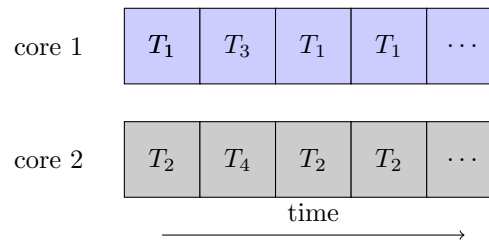
Figure 4.2: Parallel Execution on a Multicore System

In general, five areas present challenges in programming for multicore systems

- Identifying tasks

    This involves examining applications to find areas that can be divided into seperate, concurrent tasks. Ideally, tasks are independent of one another and thus can run in parallel on individual cores

- Balance

    While Identifying tasks that run in parallel, programmers msut also ensure that tasks perform equal work of equal value. In some instances, a certain task may not contribute as much value to the overall process as other tasks. Using seperate execution cores for that task might not be worth the cost

- Data Splitting

    Data accessed and manipulated by tasks must be divded to run on seperate cores, similar to how applications are divided to seperate tasks

- Data Dependency

    The data accessed by he tasks must be examined for dependencies between the two or more tasks. When data is dependent between cores, programmers must ensure that the execution of the tasks is synchronized to accomodate the dependency.
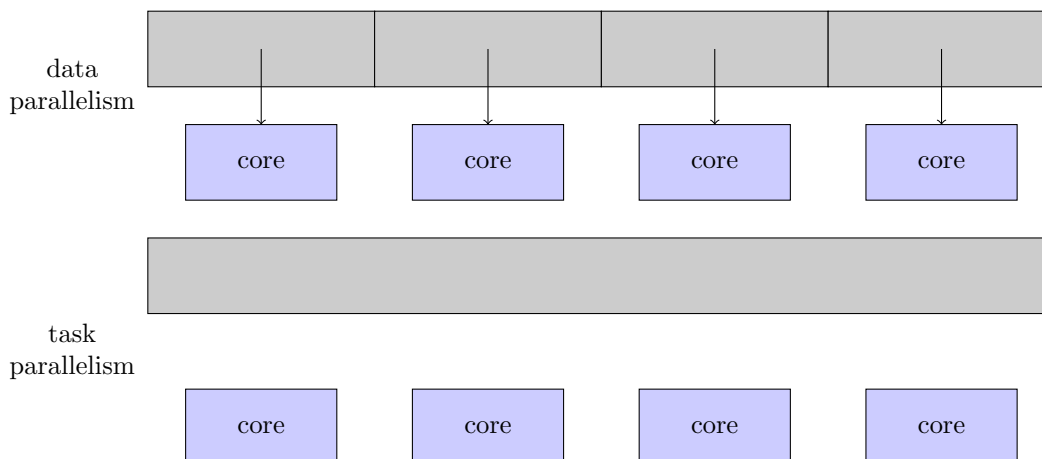
Figure 4.3: Data and Task Parallism

- Testing and Debugging

    When a program is running in parallel n multiple cores, many different execution paths are possible. Testing and debugging such concurrent programs is inherently more difficult than testing and debugging single threaded applications

4.1.2 Parallelism

Types of parallelism

- Data Parallelism

    Focuses on distributing subsets of the same data across multiple compute cores, performing the same operation on each core.

    Example: summing the contents of an array size N, on a dual core system, thread A sums the elements $[0]$ ... $[N/2-1]$, thread B sums the elements $[N/2]$ ... $[N-1]$. These threads will run in parallel on seperate cores.

- Task Parallelism

    Distributes tasks across multiple cores. Each thread peforms a unique operation. Different threads may operate on the same or different data.

    Example: Dual core system, applying two different arithmetic and/or other operation on the same block of data on seperate threads. These threads will will run parallel on seperate cores.

Data and Task parallelism may be done together, in a hybrid solution.
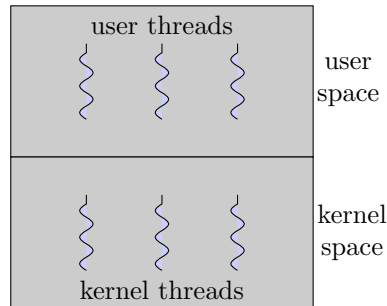
4.2   Multithreading Models

4.2.1   Many-To-One Model

Many user-level threads are mapped to a single kernel thread.
One thread block causes all to block.
Multiple threads cannot run in parallel on a multicore system because the kernel can only handle a single thread.
Rarely used.
Examples include

- Solaris Green threads

- GNU Portable threads
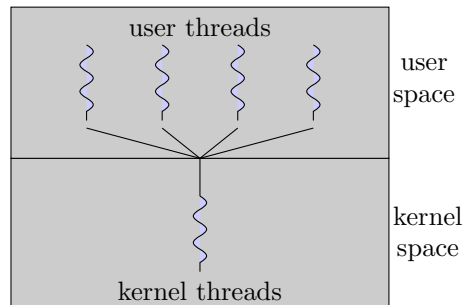
Figure 4.4: Many-to-one model

4.2.2   One-To-One Model

Each user-level thread maps to a kernel thread.
Creating a user level thread creates a kernel thread.
More concurrency than many-to-one.
Number of threads per process sometimes restricted due to overhead.
Examples include.

- Windows

- Linux
- Solaris (9 and later)
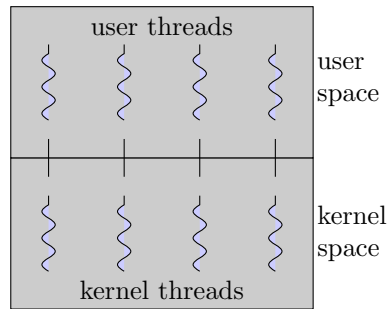
Figure 4.5: One-to-one model

4.2.3 Many-to-many model

Allows many user level threads to be mapped to many kernel threads.
Allows the operating system to create a sufficient number of kernel threads.
Examples include

- Solaris (pre version 9)
- Windows (*ThreadFiber* package)
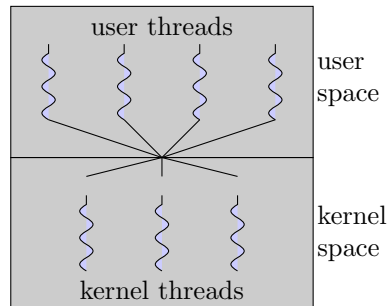
Figure 4.6: Many-to-many model

4.2.4 Two-level model

Similar to Many-to-many, except allows a user thread to be **bound** to a kernel thread Examples include

- IRIX
- HP-UX
- Tru64 UNIX
- Solaris (8 and earlier)
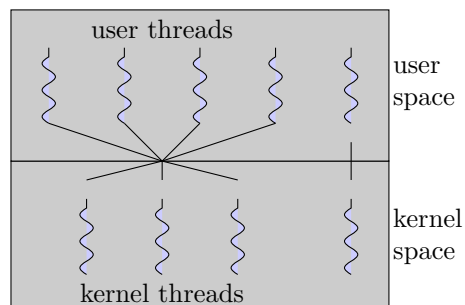
Figure 4.7: Two-level model

4.3 Thread Libraries

4.3.1 Pthreads

Refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronisation. This is a specification not an implementation

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

long double sum;          /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char **argv)
{
    pthread_t thread_id;       /* thread identifier */
    pthread_attr_t thread_attr; /* thread attributes */

    if (argc != 2)
    {
        fprintf(stderr, "usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0)
    {
        fprintf(stderr, "%d must be >= 0\n", atoi(argv[1]));
        return -1;
    }
    /* get default attributes */
    pthread_attr_init(&thread_attr);
    /* create the thread */
    pthread_create(&thread_id, &thread_attr, runner, argv[1]);
    /* wait for the thread to exit */
    pthread_join(thread_id, NULL);
    printf("sum = %Lf\n", sum);
}

/* thread executes this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;
    for (i = 1; i <= upper; i++)
    {
        sum += i;
    }
    pthread_exit(0);
}
```

Figure 4.8: Multithreading using the pthreads API

Although Windows does not natively support pthreads, some third-party implementations are available

```
#define NUM_THREADS 10
/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

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

Figure 4.9: Joining 10 threads using the pthreads API

4.3.2 Windows Threads

Similar to the technique used in pthreads. Uses a different library

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

DWORD Sum; /* data is shared by the threads */
/* The thread will execute this function */
DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 1; i <= Upper; i++)
    {
        Sum += i;
    }
    return 0;
}

int main(int argc, char **argv)
{
    DWORD ThreadID;
    HANDLE ThreadHandle;
    int Param;

    Param = atoi(argv[1]);
    /* create the thread */
    ThreadHandle = CreateThread(
        NULL, /* default security attributes */
        0, /* default stack size */
        Summation, /* thread function */
        &Param, /* parameter to thread function */
        0, /* default creation flags */
        &ThreadID /* returns thread identifier */
    );

    /* close the thread handle */
    CloseHandle(ThreadHandle);

    printf("sum = %d\n", Sum);
}
```

Figure 4.10: Multithreading using the Windows API

4.4 Implicit Threading

The growing popularity of multicore processing means that applications now require hundreds, or thousands of threads. When designing such programs, correctness grows more and more difficult. Creating and management of threads done by compilers and run-time libraries are favoured in this case.

4.4.1 Thread Pools

Creates a number of threads in a pool where they await work Advantages

- Usually faster to service a request with an existing thread than create a new thread

- Alows the number of threads in an application(s) to be bound to the size of the pool

- Seperating tasks to be performed from mechanics of creating task allows different stratgeies for running task

    Tasks could be scheduled to be run periodically

```
DWORD WINAPI PoolFunction(PVOID Param)
{
    /*
     * this function runs as a seperate thread
     */
}
```

Figure 4.11: Thread pooling in the Windows API

OpenMP

Set of compiler directives and an API for C, C++, FORTRAN.
Provides support for parallel programming and shared-memory environments.
Identifies parallel regions - blocks of code that can run in parallel.

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

int main(int argc, char **argv)
{
    int thread_id;

    #pragma omp parallel
    {
        printf("Hello from process %d\n", omp_get_thread_num());
    }
    return 0;
}
```

Figure 4.12: OpenMP

4.4.2 Grand Central Dispatch

- Apple technology for Mac OS X and iOS operating systems

- Extensions to C, C++, API and run-time library

- Allows identification of parallel sections

- Manages most details of threading

- Blocks is in "^{}" - ^{ printf("I am a block"); }

- Blocks are placed in dispatch queue and then assigned to avaiable threads in thread pool when removed

- Two types of dispatch queues

    - **Serial** - blocks are removed FIFO, queue is per process, called **Main Queue**

        Programmers create additional serial queues within program

    - **Concurrent** - removed in FIFO, but many be removed at a time

        Three system wide queues with priorities low, default, high

```
dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_async(queue, ^{ printf("I am a block."); });
```

4.5 Threading Issues

- Semantics of **fork()** and **exec()** system calls.

- Signal handling (synchronous and asynchronous)

- Thread cancellation of target thread (asynchronous or deferred)

- Thread local storage

- Scheduler activations

4.5.1 Semantics of fork() and exec()

- Does fork() duplicate only the calling thread or all threads?

    Some UNIXes have two versions of fork

- exec() usually works as normal (replaces running process including all threads)

4.5.2 Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred

- A signal handler is used to process signals

    - Signal is generated by an event

    - Signal is delivered to a process

    - Signal is handled by one of two signal handlers

        default

        user-defined

- Every signal has a default handler that kernel runs when handling a signal

    - User-defined signal handler can override default

    - For single-threaded, signal delivered to process

- Where is the signal delivered in multi-threaded?

    - Deliver the signal to the thread to which the signal applies

    - Deliver the signal to every thread in the process

    - Deliver the signal to certain threads in process

4.5.3 Thread Cancellation

- Terminating a thread before it has finished

- Thread to be canceled is target thread

- Two general approaches

    - **Asynchronous cancellation** terminates the target thread immediately

    - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

```
pthread_t thread_id;
/* create the thread */
pthread_create(&thread_id, 0, worker, NULL);
...
/* cancel the thread */
pthread_cancel(thread_id);
```

Figure 4.13: pthread code to create and cancel a thread

- Invoking thread cancellation requests cancellation, but actual cancellation depends on the thread state

Mode	State	Type
Off	Disabled	-
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

Figure 4.14: Thread States

- If a thread has cancellation disabled, cancellation remains pending till enabled
- Default type is deferred

    Cancellation only occurs when thread reaches cancellation point

    - i.e., `pthread_testcancel();`
    - Then cleanup handler is invoked
- On Linux, thread cancellation is handled through signals

4.5.4 Thread-Local Storage

- Thread-local-storage (TLS) allows each thread to have its own copy of data
- Useful when you don't have control over thread creation (i.e., using a thread pool)
- Different from local variables
    - Local variables visible only during single function invocation
    - TLS visible across function invocations
- Similar to static data
    - TLS is unique to each thread

4.5.5 Scheduler Activations

- Both Many to Many and Two-Level require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads (lightweight process (LWP))
    - Appears to be a virtual processor on which process can schedule user thread to run
    - Each LWP is attached to a kernel thread
    - How many LWPs to create?
- Scheduler activations provide upcalls - a communication mechanism from the kernel to the upcall handler in the thread library
- This communication allows an application to maintain the correct number of kernel threads

user thread ⟵

LWP ⟵ lightweight process

kernel thread ⟵

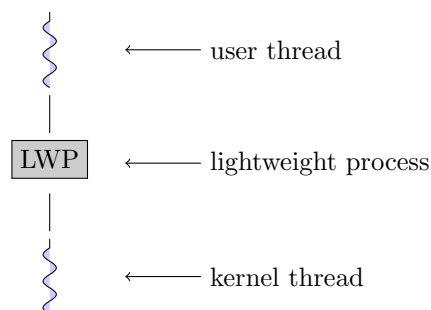Figure 4.15: Lightweight Process (LWP)

Chapter 5

Synchronisation

Processes can either execute concurrently or in parallel. These processes may be interrupted at any time, partially completing execution. Concurrent access to any shared memory may result in data inconsistency if not properly controlled.
Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

Example:

Suppose we wanted to provide a solution to the consumer-product problem that fills **all** the buffers. We can do so by having an integer `counter` that keeps track of the number of full buffers. Initially, `counter` is set to 0. It is incremented by the producer after producing a new buffer, and decremented by the consumer after it consumes a buffer.

```
while (true) {                              while (true) {
    /* produce an item in next produced */      while (counter == 0);
    while (counter == BUFFER_SIZE);                 /* do nothing */
        /* do nothing */                        next_consumed = buffer[out];
    buffer[in] = next_produced;                 out = (out + 1) % BUFFER_SIZE;
    in = (in + 1) % BUFFER_SIZE;                counter--;
    counter++;                                  /* consume the item in next_consum
}                                           }
```

Figure 5.1: Producer and Consumer

5.1 Race Conditions

While the producer and consumer routines shown are correct seperately, they may not function correctly when executed concurrently. For example: Suppose that the value of the variable `count` is currently 5 and the producer and consumer processes concurrently execute the statements "`count++`" and "`count--`". This will lead to the variable being 4, 5 or 6. With the only correct value being `count == 5`, which is executed if the producer and consumer execute seperately. We can show the value maybe incorrect by using the following implementation

$$register_1 = count$$
$$register_1 = register_1 + 1$$
$$count = register_1$$

where $register_1$ is one of the local CPU registers.
Similarly, the statement `count - 1` is implemented as follows

$$register_2 = count$$
$$register_2 = register_2 - 1;$$
$$count = register_2$$

where again, $register_2$ is a local CPU register.
Even though $register_1$ and $register_2$ are local CPU registers, the value of this register will be saved and stored by the interrupt handler
The concurrent execution `count++` and `count -- 1` is equivalent to the sequential execution where lower evel statements previously are interleaved in an arbitrary order, with the order of the high-level statement is preserved.

$$T_0 \quad producer \quad \text{execute} \quad register_1 = count \quad \{register_1 = 5\}$$

$$T_0 \quad producer \quad \text{execute} \quad register_1 = count \quad \{register_1 = 5\}$$

Chapter 6

Safety Critical Systems

Safety is the freedom from conditions that cause death, injury, illness, damage to or loss of equipment or property, or environmental harm

Software is inherently not safe or unsafe, however can contribute to unsafe conditions in a safety critical system. Such software is **Safety Critical**

IEEE definition: "Software whose use in a system can result in unacceptable risk. Safety-critical software incliudes softwre whose operation or failure to operate can lead to a hazardous state. Software intended to reover from hazardous states, and software intended to mitigate the severity of an incident"

6.1 MISRA C

- Motor Industry Software Reliability Association

6.2 NASA Power of 10

- Avoid complex flow constructs (goto, recursion, jumps)

- All loops must have fixed bounds (prevents runaway code)

- Avoid heap memory allocation (no malloc, define everything in main)

- Restrict functions to a single page (max 50 lines)

- Use a minimum of two runtime assertions per function

- Restrict the scope of data to the smallest possible

- Check the return value of all non-void functions, or cast to void to indicate the return is useless

- Use the preprocessor sparingly. (DO NOT USE stdio.h, local.h, abort/exit/system from stdlib.h, time handling from time.h)

- Limit pointer use to a single dereference and DO NOT USE FUNCTION POINTERS

- Compile with all warnings active (Wall, Wextra, etc.) all warnings should then be addressed before release

Chapter 7

Distributed Systems

A distributed system is a collection of processors that do not share memory or a clock. Instead, each node has its own local memory. The nodes communicate over various networks, such as high-speed busses.

Applications of distributed systems vary widely, from providing transparent file access inside an organization, to large-scale cloud storage services, to business analysis of trends on large datasets, to parallel processing of scientific data. With the most ubiqiutous form of a distributed system being the Internet.

7.1 Basic Concepts

A distributed system is a collection of loosely coupled nodes interconnected by a communication network. From the point of view of a specific node in a distributed system, the rest of the nodes and their respective resources are remote, whereas its own resources are local.

Processors are variously called **nodes, computers, machines, hosts**

- **Site** is the location of the processor
- Generally a **server** has a resource a **client** node at a different site wants to use
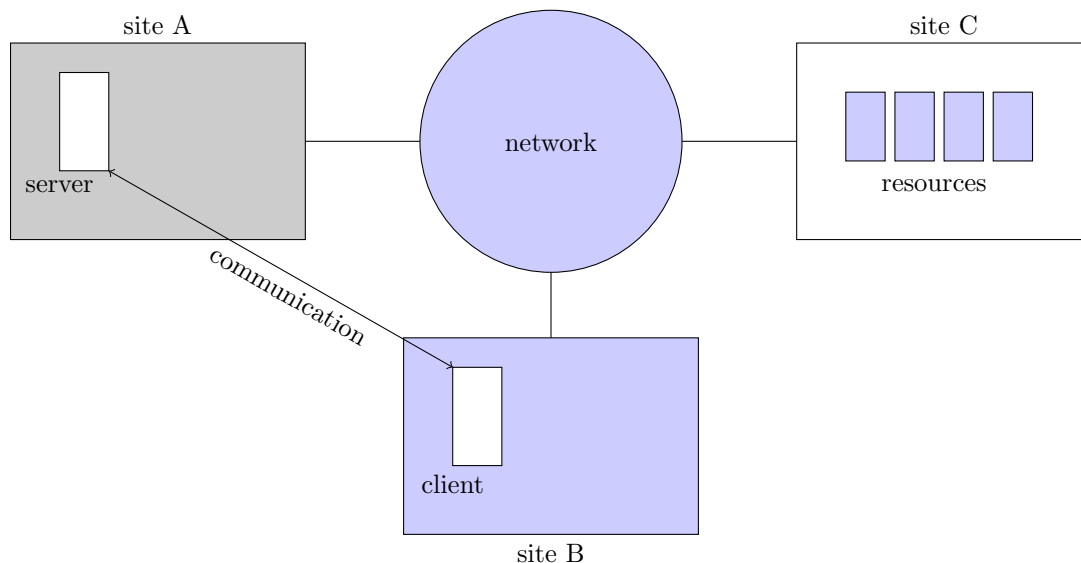
Figure 7.1: A client-server distributed system

7.2 Advantages of Distributed Systems

- **Resource Sharing**
  - Sharing and printing files at remote sites
  - Processing information in a distributed database

    – Using remote specialized hardware devices

- **Computation speedup**
  - **Load sharing**
  - **Job migration**

- **Reliability**

    Detect and recover from site failure, function transfer, reintegrate failed site.

- **Communication**

    Message passing

All higher-level functions of a standalone system can be expanded to encompass a distributed system

Computers can be downsized, more flexibility, better user interfaces and easier maintenance by moving from a large system to a cluster of smaller systems performing distributed computing

7.3 Network-Operating Systems

7.4 Distributed Operating Systems

Chapter 8

CPU Scheduling

CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the computer more productive.

8.1   Basic Concepts

A single core system can only run a single process at a time. Others must wait till the CPU's core is free and can be rescheduled. Multiprogramming is the idea of having a process running at all times to maximise CPU utilization.

A process is executed until it must wait, typically for the completion of an I/O request. A simple computer system just idles during this period, waiting compute time, no work is accomplished. With multiprogramming, we try to use this time productively. By keeping several processes in memory, when one process has to wait, the operating takes the CPU away from the process and gives it to another process. On a multicore system this is extended to all processing cores on the system.

Such scheduling is fundamental to an operating system's functionality. Almost all computer resources are scheduled before use. The CPU, being one of the primary resources of a computer needs special attention during it's scheduling

.
.
.

load store	
add store	CPU Burst
read from file	

| *wait for I/O* | I/O Burst |

store increment	
index	CPU Burst
write to file	

| *wait for I/O* | I/O Burst |

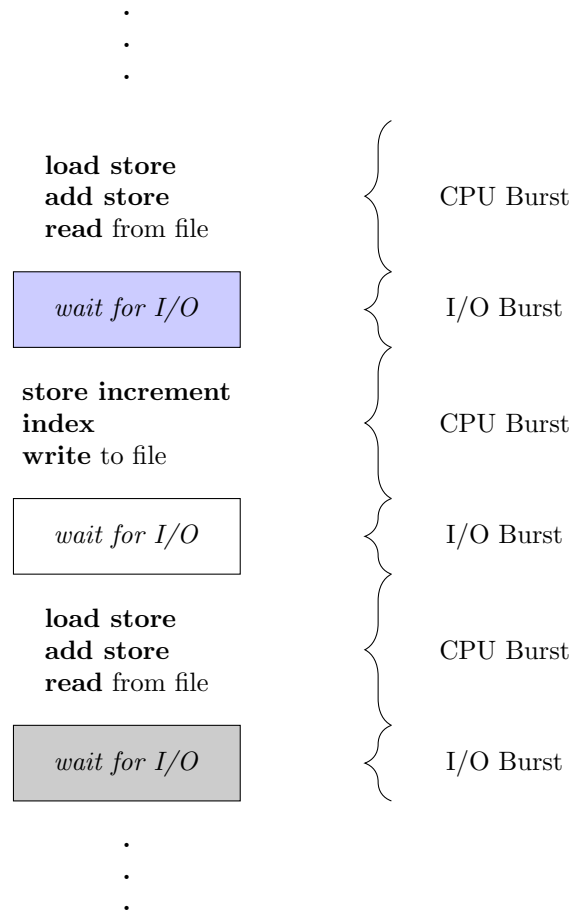| **load store** | |
| **add store** | CPU Burst |
| **read** from file | |

| *wait for I/O* | I/O Burst |

.
.
.

Figure 8.1: Alternating Sequence of CPU and I/O Bursts

CPU-I/O Burst Cycle

Chapter 9

Deadlocks

Chapter 10

Virtual Machines

10.1   Benefits and Features

- Templating - create an OS + application VM, provide it to customers, use it to create multiple instances

    Docker containers, AWS AMIs, etc.

- Live Migration - Move a running VM from host to another

    No interruption of user access

- All these features together $\implies$ cloud computing

    Using APIs, programs tell cloud infrastructure (servers, networking, storage) to create new guests, VMs, virtual desktops

10.2   Types of Virtual Machines

Many variations as well as hardware details, assiming VMMs take advantage of hardware features, they can simplify implementation and improve performance All Virtual Machines have a lifecycle

- Created by the VMM

- Resources assigned to it (cores, amount of memory, networking details, storage details)

- In type 0 hypervisor, resources are usually dedicated

- Other types dedicate or share resource (or mix)

- When no longer needed, VMs can be deleted, freeing resources

Steps are simpler and faster than a physical machine install

- Can lead to VM sprawl with lots of VMs, history and state are difficult to track

10.3   Types of VMs

10.3.1   Type 0 Hypervisor

Is an older idea under many names by hardware manufacturers

- Partitions, Domains

Hardware features must be implemented in firemware, the VMM is in firmware. Smaller feature set than other types

Chapter 11

Practicals

11.1 Introduction to Operating Systems

11.2 Operating System Structures

11.3 Processes

11.4 Threads

11.4.1 Thread Creation

Exercise: Modify sampleThread.c. Create a third thread and this time sum up the first 20 numbers 1,2,...20. Practice passing a struct to the thread:

```c
typedef struct num_thdata {
    int thread_no;
    int sum_to;
} thsum;
```

```c
#include <unistd.h>     /* Symbolic Constants */
#include <sys/types.h>  /* Primitive System Data Types */
#include <errno.h>      /* Errors */
#include <stdio.h>      /* Input/Output */
#include <stdlib.h>     /* General Utilities */
#include <pthread.h>    /* POSIX Threads */
#include <string.h>     /* String handling */

#define MESSAGE_REPEAT 2

/* struct to hold data to be passed to a thread
   this shows how multiple data items can be passed to a thread */
typedef struct str_thdata {
    int thread_no;
    char message[100];
} thdata;

typedef struct num_thdata {
    int thread_no;
    int sum_to;
} thsum;
/* declare sum as global */
int sum;
/* prototype for thread routine */
void *print_message_function(void *ptr);
void *sum_to_function(void *ptr);
```

```c
int main(int argc, char *argv[]) {
    pthread_t thread1, thread2, thread3; /* thread variables */
    thdata data1, data2;        /* structs to be passed to threads */
    thsum data3;
    /* initialize data to pass to thread 1 */
    data1.thread_no = 1;
    strcpy(data1.message, "Hello! Welcome to Practical 5 - Week 5 already!!!");

    /* initialize data to pass to thread 2 */
    data2.thread_no = 2;
    strcpy(data2.message, "Hi! Week 5 - Time flies by when programming in C");

    /* initialize data to pass to thread 3 */
    data3.thread_no = 3;
    data3.sum_to = 20;
    /* create threads 1, 2, and 3
     *  function must take a parameter of void *(the second void *)
     *  return a value of void * (first void)
     */
    pthread_create(&thread3, NULL, sum_to_function, &data3);
    pthread_create(&thread1, NULL, print_message_function, &data1);
    pthread_create(&thread2, NULL, print_message_function, &data2);
    /* Main block now waits for both threads to terminate, before it exits
       If main block exits, both threads exit, even if the threads have not
       finished their work */
    pthread_join(thread3, NULL);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    /* exit */
    return EXIT_SUCCESS;
} /* main() */

/**
 * print_message_function is used as the start routine for the threads used
 * it accepts a void pointer
 **/
void *print_message_function(void *ptr) {
    thdata *data;
    data = ptr; /* type cast to a pointer to thdata */

    /* do the work */
    for (int x = 0; x < MESSAGE_REPEAT; x++)  {
        printf("\n\nThread %d has the following message -->  %s \n", data->thread_no, data->message);
    }
    return NULL;
}

void *sum_to_function(void *ptr)
{
    thsum data = *((thsum*)ptr); // cast void to thsum
    for (int i = 0; i <= data.sum_to; i++)
    {
        sum += i;
    }
    printf("Thread number %d reports that the sum of the first %d numbers is = %d\n", data.thread_no, data.sum_t
    return NULL;
}
```

11.4.2   Producer and Consumer Threads

11.4.3   Searching Hash Table Values

11.4.4   OpenMP

11.5   Synchronisation

11.6   Safety Critical Systems

11.7   Distributed Systems

11.8   CPU Scheduling

11.9   Deadlocks