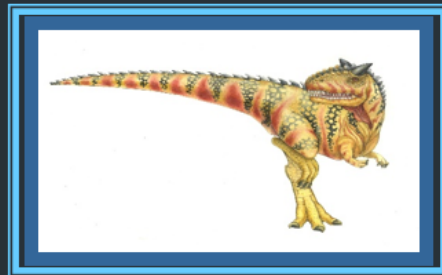


Module:2 - L2 - Processes



Dr. Rishikeshan C A
SCOPE, VIT Chennai

Outline

- ✂ Program, Process
- ✂ Process Concept
- ✂ PCB
- ✂ Threads
- ✂ Multithreading
- ✂ Process scheduling
- ✂ Loads program into memory, overwriting all but the kernel
- ✂ Program exit -> shell reloaded

Objectives

- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To explore interprocess communication using shared memory and message passing
- To describe communication in client-server systems

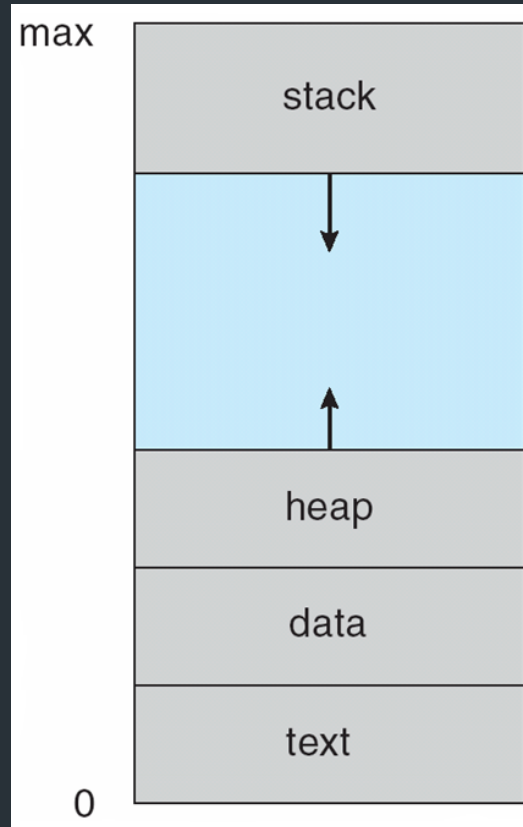
Process

- Program is termed as **passive** entity which stores on disk whereas process is **active** entity. Once the executable file gets loaded into memory then program will be termed as process
- Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- One program can be several processes
 - Consider multiple users executing the same program

Process

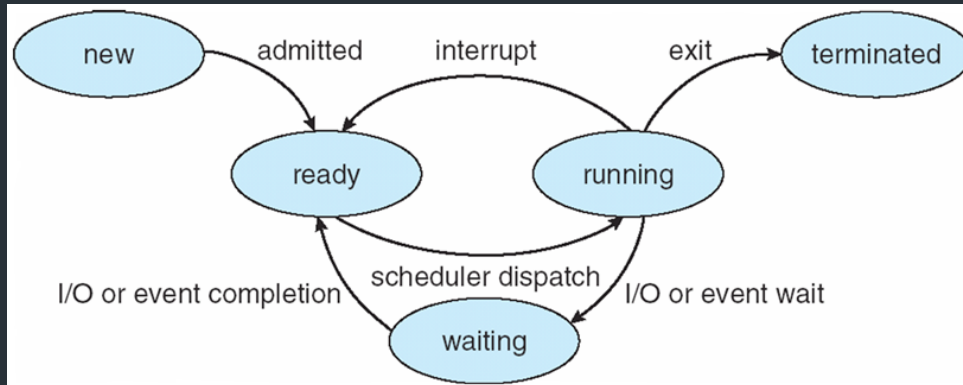
- Parts of Process
 - **Text Section:** Program code
 - Current activity including **program counter**, processor registers
 - **Stack** containing temporary data
 - Function parameters, return addresses, local variables
 - **Data section** containing global variables
 - **Heap** containing memory dynamically allocated during run time

Process in Memory



Process State

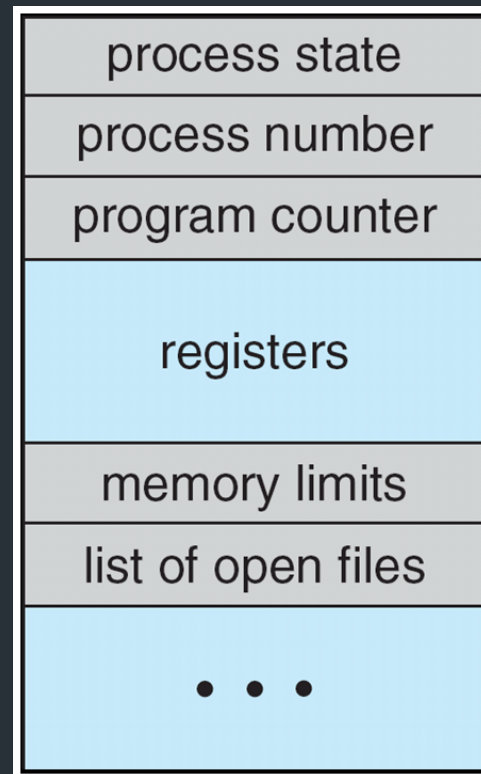
- As a process executes, it changes **state**
 - **new**: The process is being created
 - **running**: Instructions are being executed
 - **waiting**: The process is waiting for some event to occur
 - **ready**: The process is waiting to be assigned to a processor
 - **terminated**: The process has finished execution



Process Control Block (PCB)

Information associated with each process
(also called **task control block**)

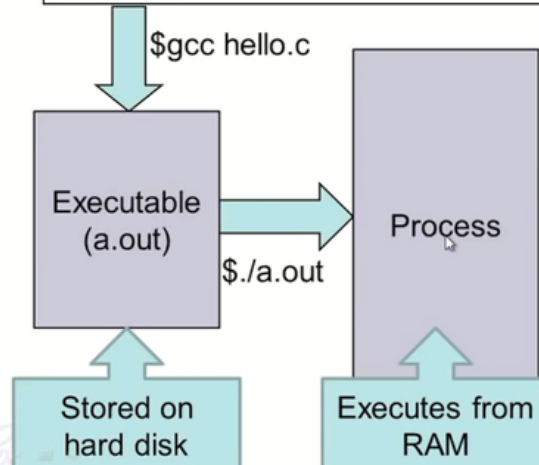
- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



Executing Programs (Process)

```
#include <stdio.h>

int main(){
    char str[] = "Hello World\n";
    printf("%s", str);
}
```



Executing Programs (Process)

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

```
int main(){  
    char str[] = "Hello World\n";  
    printf("%s", str);  
}
```

\$gcc hello.c

Executable
(a.out)

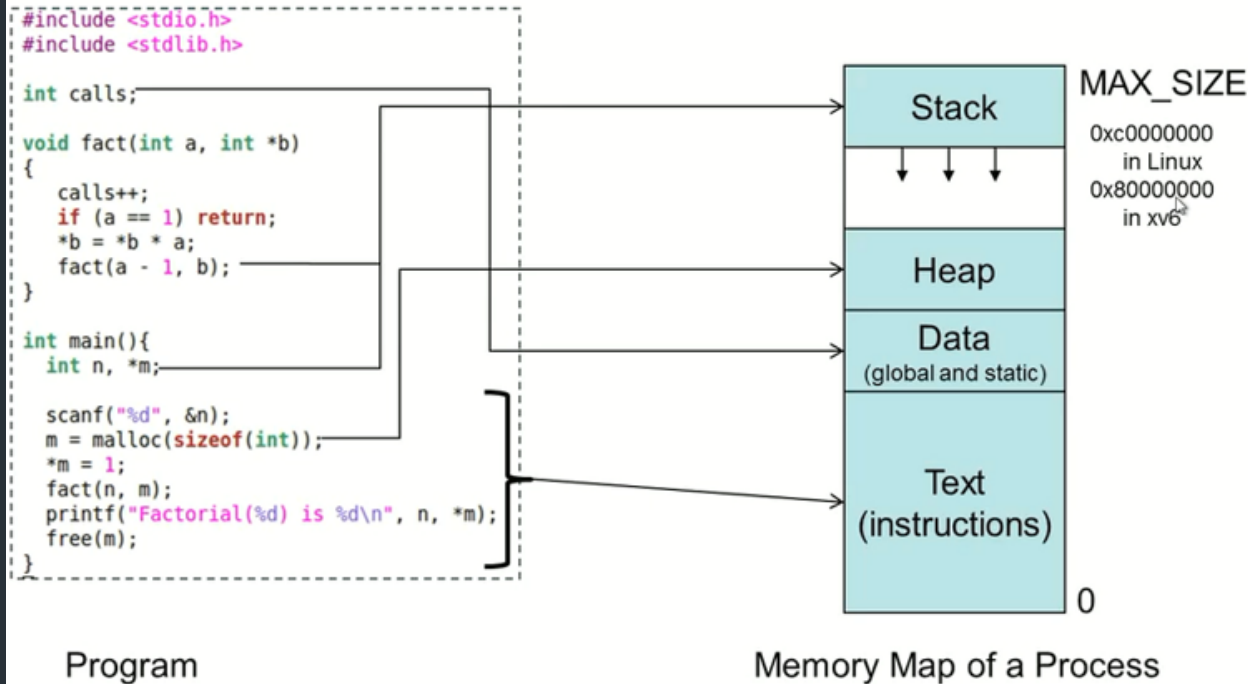
\$/a.out

Process

- Process

- A program in execution
- Present in the RAM
- Comprises of
 - Executable instructions
 - Stack
 - Heap
 - State in the OS (in kernel)
- State contains : registers, list of open files, related processes, etc.

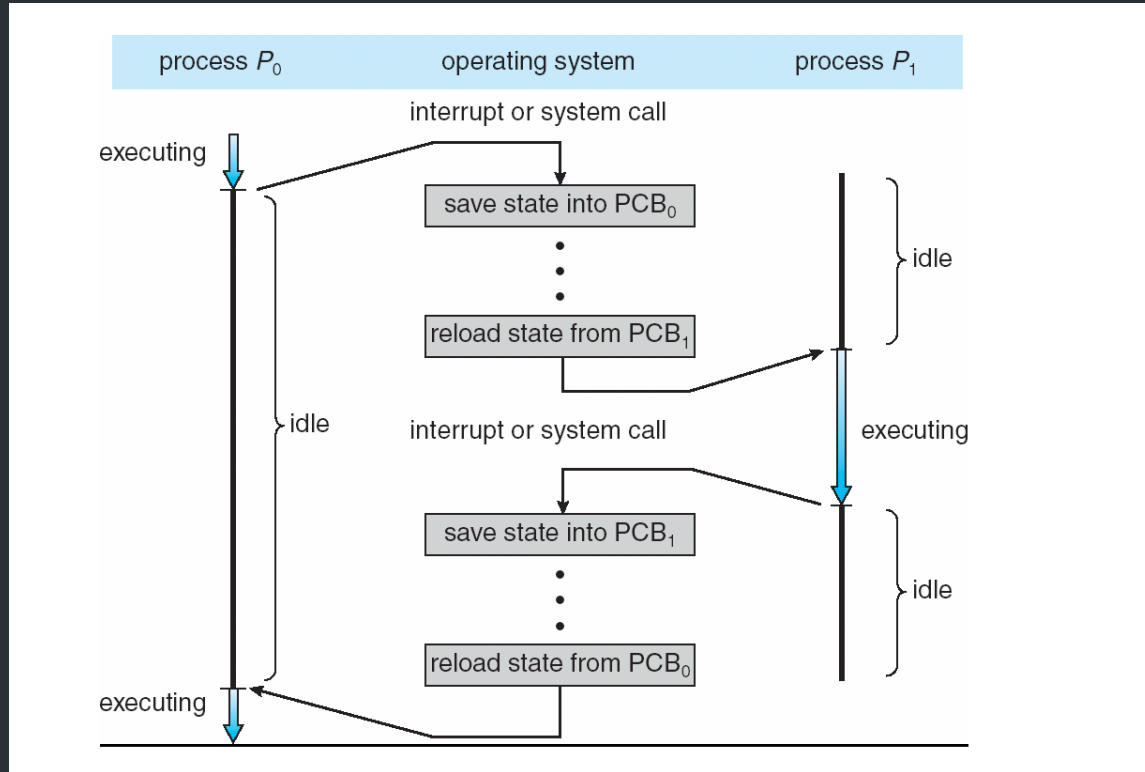
Process Memory Map



Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
 - The more complex the OS and the PCB → the longer the context switch
- Time dependent on hardware support
 - Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once

CPU Switch From Process to Process

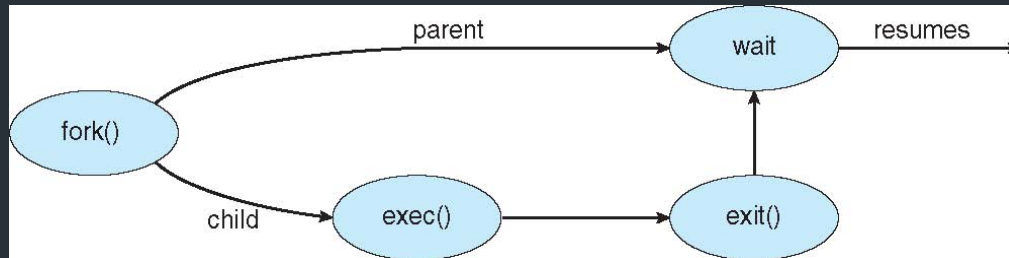


Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier (pid)**
- Resource sharing options
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
- Execution options
 - Parent and children execute concurrently
 - Parent waits until children terminate

Process Creation (Cont.)

- Address space
 - Child duplicate of parent
 - Child has a program loaded into it
- UNIX examples
 - **fork()** system call creates new process
 - **exec()** system call used after a **fork()** to replace the process' memory space with a new program



Process Creation



Process Creation

- Process creation in Unix is unique. Most operating systems implement a spawn mechanism to create a new process in a new address space, read in an executable, and begin executing it.
- Unix takes the unusual approach of separating these steps into two distinct functions: `fork()` and `exec()`. The first, `fork()`, creates a child process that is a copy of the current task. It differs from the parent only in its PID (which is unique), its PPID (parent's PID, which is set to the original process), and certain resources and statistics, such as pending signals, which are not inherited.
- The second function, `exec()`, loads a new executable into the address space and begins executing it. The combination of `fork()` followed by `exec()` is similar to the single function most operating systems provide.

C Program Forking Separate Process

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

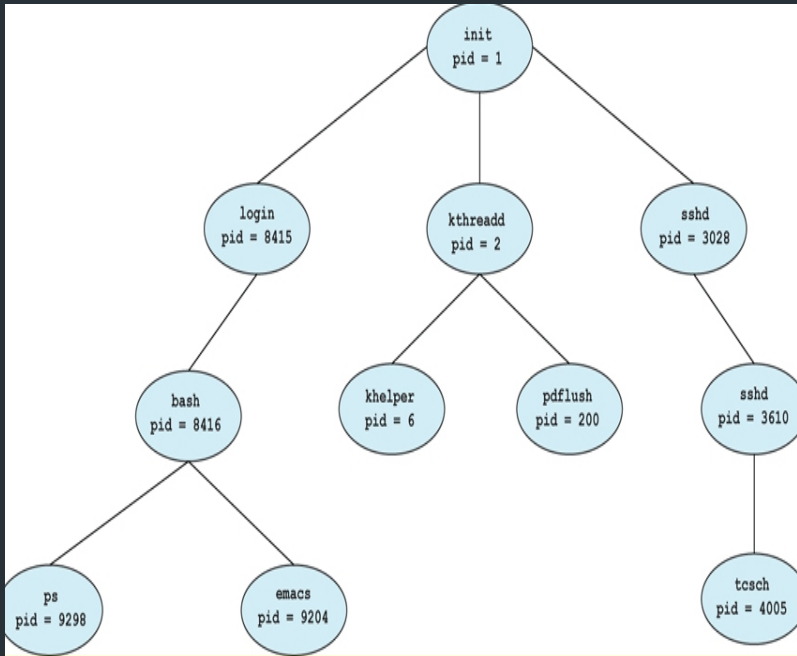
int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
```

Sample Tree of Processes in Linux



- On typical LINUX systems the process scheduler is termed **sched**, and is given PID 0. The first thing it does at system startup time is to launch **init**, which gives that process PID 1. Init then launches all system daemons and user logins, and becomes the ultimate parent of all other processes.

Process Termination

- Process executes last statement and then asks the operating system to delete it using the `exit()` system call.
- Parent may terminate the execution of children processes using the `abort()` system call. Some reasons for doing so:
 - Child has exceeded allocated resources
 - Task assigned to child is no longer required
 - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates
- If a process terminates, then all its children must also be terminated.
 - **cascading termination.** All children, grandchildren, etc. are terminated.
 - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the `wait()` system call. The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```
- If no parent waiting (did not invoke `wait()`) process is a **zombie**
- If parent terminated without invoking `wait()`, process is an **orphan**

Threads

- A basic unit of CPU utilization, consisting of a **program counter, a stack, and a set of registers**, (and a **thread ID**.) Traditional (heavyweight) processes have a single **thread** of control –
 - There is one program counter, and one sequence of instructions that can be carried out at any given time.
- Consider having multiple program counters per process
 - Multiple locations can execute at once
 - Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB

Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
 - Update display
 - Fetch data
 - Spell checking
 - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Threads

A thread is a basic unit of CPU utilization.

It comprises


A thread ID

A program counter

A register set and

A stack

It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.

A traditional / heavyweight process has a **single thread** of control. 
If a process has **multiple threads** of control, it can perform **more than one task at a time**.

User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
 - POSIX **Pthreads**
 - Windows threads
 - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
 - Windows
 - Solaris
 - Linux
 - Tru64 UNIX
 - Mac OS X

How to view Threads

ProcessThreadsView - C:\Users\JAISON\AppData\Local\Chromium\Application\chrome.exe

File Edit View Options Help

Thread ID	Context Switches	Last Context S...	Status	B...	D...	Created Time	User Time	Kernel Time	Visible ...	Hidden...	Window Title	Window CL...
9780	3	0	UserRequest	8	10	15-05-2019 11:16:53	00:00:00.000	00:00:00.000	0	0		
10908	3	0		8	9	15-05-2019 11:16:55	00:00:00.000	00:00:00.000	0	0		
13400	2	0	UserRequest	8	10	15-05-2019 11:16:53	00:00:00.000	00:00:00.000	0	0		
13624	2	0		6	7	15-05-2019 11:16:55	00:00:00.000	00:00:00.000	0	0		
13788	2	0		8	9	15-05-2019 11:16:55	00:00:00.000	00:00:00.000	0	0		
13828	413	0	WrQueue	8	10	15-05-2019 11:16:53	00:00:00.000	00:00:00.000	0	0		
13848	9	0	UserRequest	8	9	15-05-2019 11:16:55	00:00:00.000	00:00:00.000	0	0		
13892	3	0	WrQueue	8	9	15-05-2019 11:16:53	00:00:00.000	00:00:00.000	0	0		
13960	4	0	UserRequest	8	9	15-05-2019 11:16:53	00:00:00.000	00:00:00.000	0	0		
14036	2,544	0	UserRequest	8	8	15-05-2019 11:16:49	00:00:01.326	00:00:00.171	0	0		
14304	2	0	UserRequest	8	9	15-05-2019 11:16:55	00:00:00.000	00:00:00.000	0	0		

Call Stack:

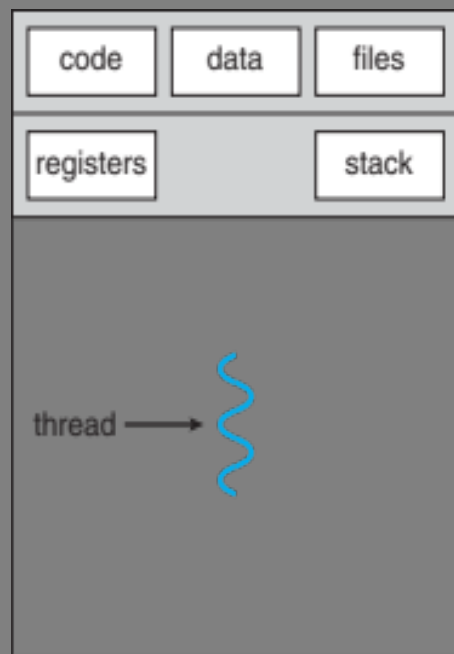
Processor Registers:

RAX = 00000000`00000000
RBX = 00000000`00000000
RCX = 00000000`00000010 chrome.exe+0x10
RDX = 00000000`032E0000
RSI = 00000000`FFFFFFFF
RDI = 00000000`0000023C chrome.exe+0x23c
RBP = 00000000`00000001 chrome.exe+0x1
RIP = 00000000`76DE135A ntdll.dll!NtWaitForSingleObject+0xa
RSP = 00000000`0340F878 -> DC 10 30 FD FE 07 00 0E 00 30 00 37 00 00 00 F0 9A 39 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00
R8 = 00000000`0333BE3F
R9 = 00000000`FFFFFFFF
R10 = 00000000`0333BE40

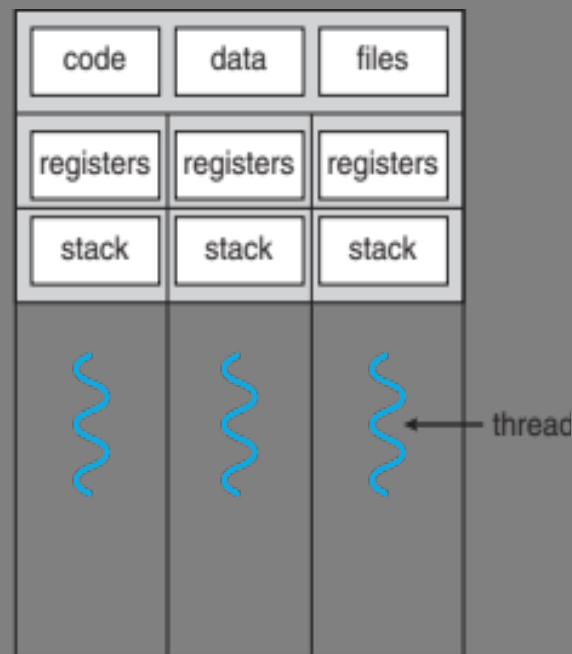
11 Threads, 1 Selected

NirSoft Freeware. <http://www.nirsoft.net>

Single and Multithreaded Processes



single-threaded process



multithreaded process

Multi-Threading - Benefits

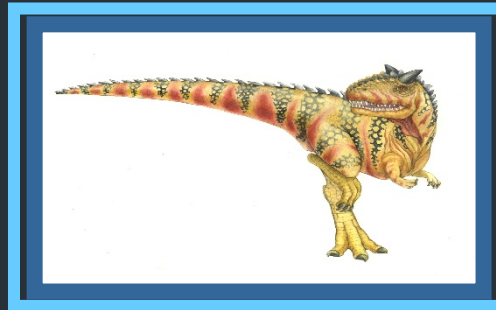
- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing** – threads share resources of process, easier than shared memory or message passing
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching
- **Scalability** – process can take advantage of multiprocessor architectures

References

- Operating System Concepts, 9th Edition” by Silberschatz Galvin Gagne
- Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. Posix abstractions in modern operating systems: The old, the new, and the missing. In Proceedings of the Eleventh European Conference on Computer Systems (EuroSys), pages 19:1–19:17. ACM, 2016
- <http://www.cs.nthu.edu.tw/~ychung/slides/CSC3150/Abraham-Silberschatz-Operating-System-Concepts---9th2012.12.pdf>

Thank You!

Contents for additional reference



Multicore Programming

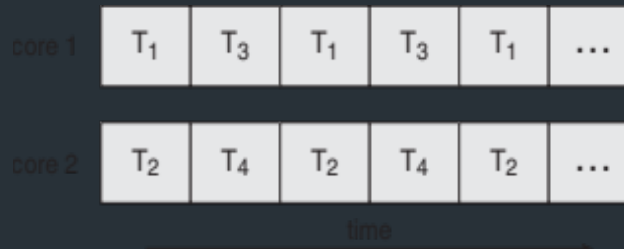
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
 - **Dividing activities**
 - **Balance**
 - **Data splitting**
 - **Data dependency**
 - **Testing and debugging**
- **Parallelism** implies a system can perform more than one task simultaneously
- **Concurrency** supports more than one task making progress
 - Single processor / core, scheduler providing concurrency

Concurrency vs. Parallelism

- Concurrent execution on single-core system:



- Parallelism on a multi-core system:



Multicore Programming (Cont.)

- Types of parallelism
 - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
 - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading
 - CPUs have cores as well as *hardware threads*
 - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- *Specification*, not *implementation*
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Pthreads Example

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

int 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 tid; /* the thread identifier */
    pthread_attr_t attr; /* set of 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;
    }
}
```

Pthreads Example (Cont.)

```
/* get the default attributes */
pthread_attr_init(&attr);
/* create the thread */
pthread_create(&tid,&attr,runner,argv[1]);
/* wait for the thread to exit */
pthread_join(tid,NULL);

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

/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```

Pthreads Code for Joining 10 Threads

```
#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);
```

Windows Multithreaded C Program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 0; i <= Upper; i++)
        Sum += i;
    return 0;
}

int main(int argc, char *argv[])
{
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;

    if (argc != 2) {
        fprintf(stderr, "An integer parameter is required\n");
        return -1;
    }
    Param = atoi(argv[1]);
    if (Param < 0) {
        fprintf(stderr, "An integer >= 0 is required\n");
        return -1;
    }
}
```

Windows Multithreaded C Program (Cont.)

```
/* 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 the thread identifier */

if (ThreadHandle != NULL) {
    /* now wait for the thread to finish */
    WaitForSingleObject(ThreadHandle, INFINITE);

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

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

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:

```
public interface Runnable
{
    public abstract void run();
}
```

- Extending Thread class
- Implementing the Runnable interface

Java Multithreaded Program

```
class Sum
{
    private int sum;

    public int getSum() {
        return sum;
    }

    public void setSum(int sum) {
        this.sum = sum;
    }
}

class Summation implements Runnable
{
    private int upper;
    private Sum sumValue;

    public Summation(int upper, Sum sumValue) {
        this.upper = upper;
        this.sumValue = sumValue;
    }

    public void run() {
        int sum = 0;
        for (int i = 0; i <= upper; i++)
            sum += i;
        sumValue.setSum(sum);
    }
}
```


Thank You!