

Operating System Principles

操作系统原理

Process&Thread Concepts

李旭东

leexudong@nankai.edu.cn

Nankai University

Objectives

- Execution of Program
- Process Concept
- Thread Concept

leexudong@nankai.edu.cn

2

Execution of Program

- Program
 - I(Input) , C(Compute) , P(Print)
- Type I
 - sequential processes

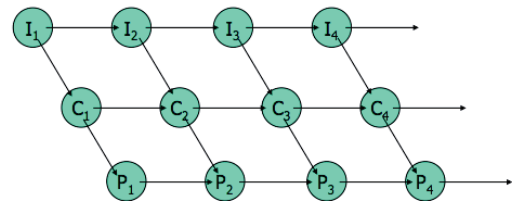


leexudong@nankai.edu.cn

3

Execution of Program

- Program
 - I(Input) , C(Compute) , P(Print)
- Type II
 - concurrent processes



leexudong@nankai.edu.cn

4

Multiprogramming

- Program
 - Program
 - Job
 - Task
 - ...
- Concurrent
 - Single CPU: Pseudoparallelism
 - Multiprocessor: Parallelism

leexudong@nankai.edu.cn

5

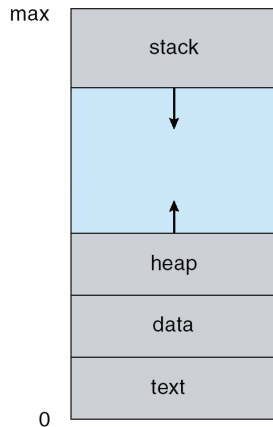
Process

- A process is just an instance of an executing program
- Including the current values of the program counter, registers, and variables
- Virtual CPU
- Main Memory
- Kernel Objects

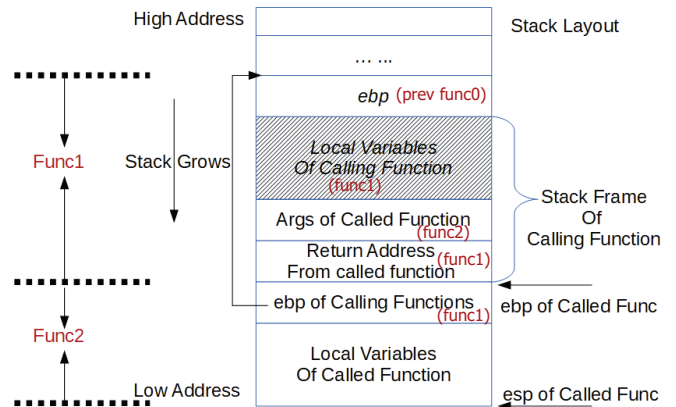
leexudong@nankai.edu.cn

6

Process in Memory



Stack Frame of Function



Multi-Processes

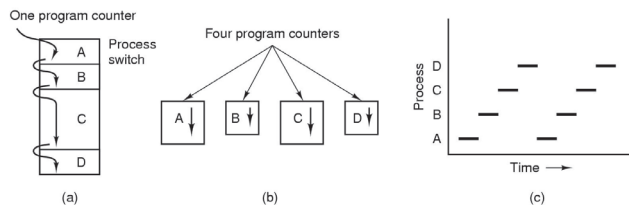


Figure 2-1. (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

Program v.s. Process

- Program v.s. Process
 - Recipe v.s. Cooking



Process Creation

- Four principal events
 - System initialization
 - Execution of a process creation system call by a running process
 - A user request to create a new process
 - Initiation of a batch job
- Functions
 - UNIX: fork, execve
 - Windows(Win32):CreateProcess

C Program Forking Separate Process

```
int main()
{
    pid_t pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-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");
    }
    printf("ok");
}
```

Process Termination

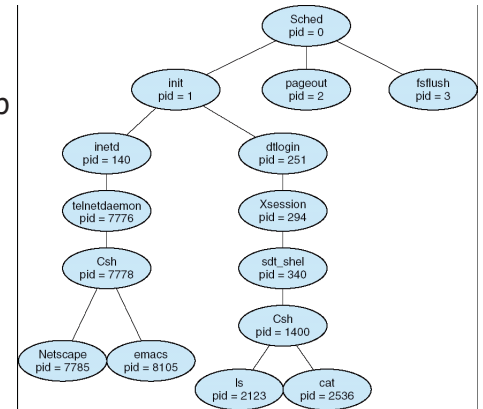
- Events which cause process termination
 - Normal exit (voluntary)
 - Error exit (voluntary)
 - Fatal error (involuntary)
 - Killed by another (involuntary)

leexudong@nankai.edu.cn

13

Process Hierarchies

- UNIX
 - Tree
 - Process group



leexudong@nankai.edu.cn

14

Process Hierarchies

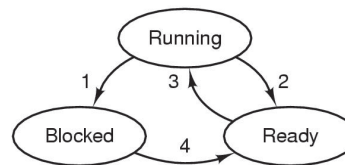
- Windows
 - No hierarchy
 - All processes are equal
 - Handle
 - Parent process use it to control child process

leexudong@nankai.edu.cn

15

Process States

- e.g.
 - `cat file1 file2 file3 | grep osp | wc -l`



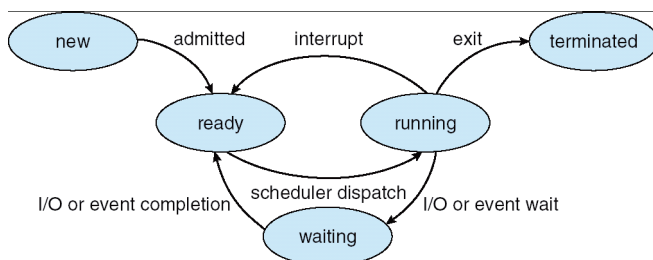
1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Figure 2-2. A process can be in running, blocked, or ready state. Transitions between these states are as shown.

leexudong@nankai.edu.cn

16

Process States



Implementation of Process

- Process control block (PCB)
- Process table: array of PCBs

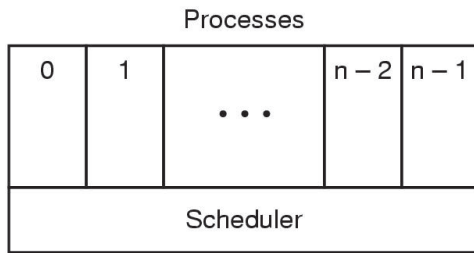
Process management	Memory management	File management
Registers Program counter Program status word Stack pointer Process state Priority Scheduling parameters Process ID Parent process Process group Signals Time when process started CPU time used Children's CPU time Time of next alarm	Pointer to text segment info Pointer to data segment info Pointer to stack segment info	Root directory Working directory File descriptors User ID Group ID

leexudong@nankai.edu.cn

18

Implementation of Process

- scheduler



leexudong@nankai.edu.cn

19

Implementation of Process

- Process switch by interrupt

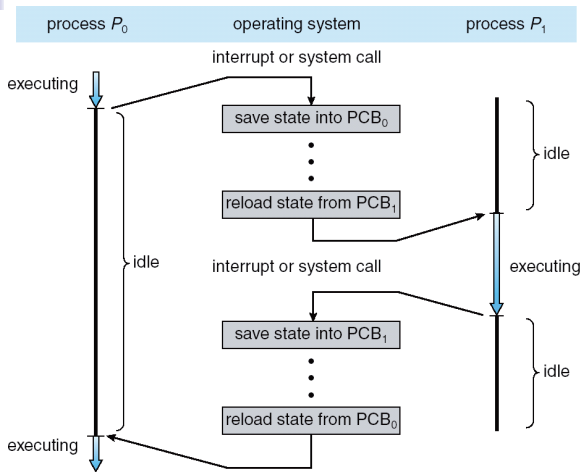
1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

Figure 2-5. Skeleton of what the lowest level of the operating system does when an interrupt occurs.

leexudong@nankai.edu.cn

20

CPU Switch From Process to Process



leexudong@nankai.edu.cn

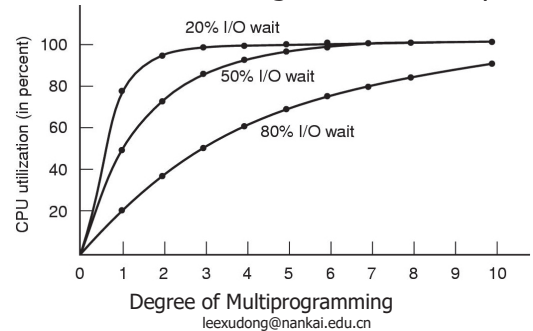
23

Modeling Multiprogramming

- CPU utilization

$$= 1 - p^n$$

- P : the time waiting for I/O to complete



22

Modeling Multiprogramming

- e.g.
- A computer
 - 512MB memory
 - 128MB for OS, 128MB/Process
 - 80% time to wait I/O
 - CPU utilization = $1 - 0.8^3 = 49\%$
- Add second memory: 512MB
 - CPU utilization = $1 - 0.8^7 = 79\%$
- Add third memory: 512MB
 - CPU utilization = ?

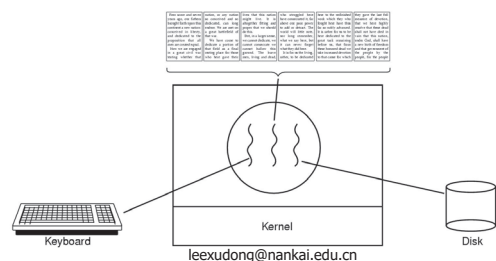
leexudong@nankai.edu.cn

23

Thread

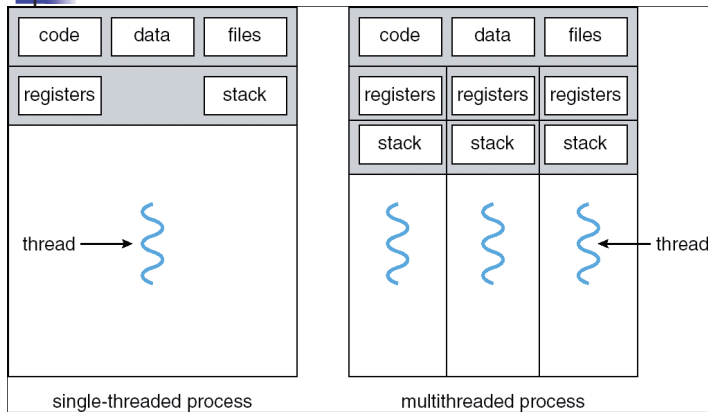
- Motivation

- In traditional os, each process has an address space and a single execution unit of control
- But ...: A word processor



24

Single and Multithreaded Processes



Benefits of MultiThreads

- Multithreaded programming
 - Responsiveness
 - Resource sharing
 - Economy
 - Utilization of multiprocessor architecture

leexudong@nankai.edu.cn

26

Cases of MultiThreads

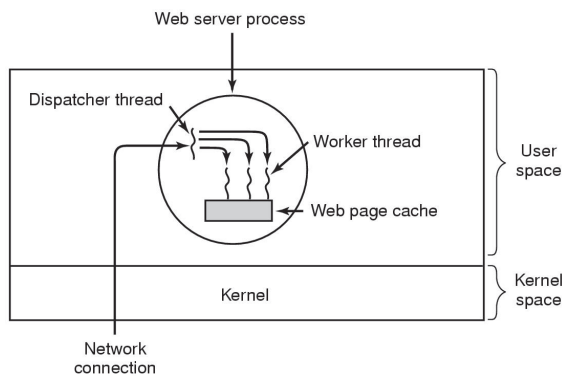


Figure 2-8. A multithreaded Web server.

leexudong@nankai.edu.cn

27

Cases of MultiThreads

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

Figure 2-9. A rough outline of the code for Fig. 2-8.

(a) Dispatcher thread. (b) Worker thread.

leexudong@nankai.edu.cn

28

Three ways to construct a server

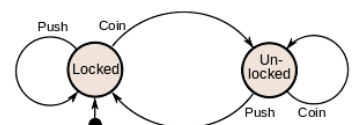
Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

leexudong@nankai.edu.cn

29

Finite-State Machine (FSM)

- e.g. a turnstile



leexudong@nankai.edu.cn

30

The Classical Thread Model

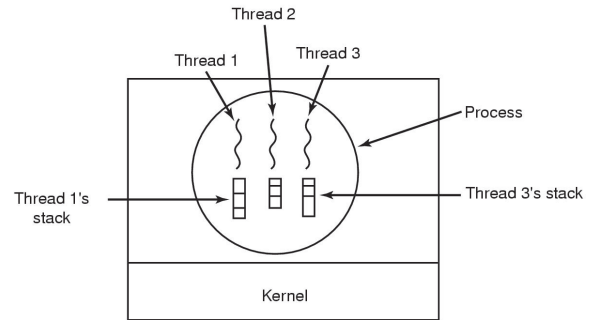
Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Figure 2-12. The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

leexudong@nankai.edu.cn

31

Each thread has its own stack



leexudong@nankai.edu.cn

32

POSIX Threads

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Example: POSIX Threads

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

#define NUMBER_OF_THREADS 10

void *print_hello_world(void *tid)
{
    /* This function prints the thread's identifier and then exits. */
    printf("Hello World. Greetings from thread %d\n", tid);
    pthread_exit(NULL);
}

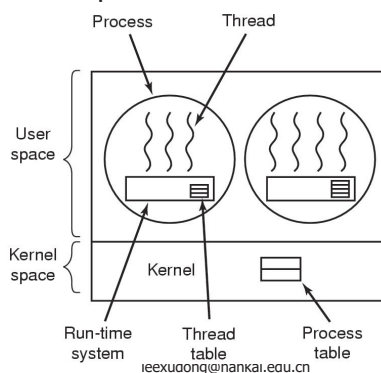
int main(int argc, char *argv[])
{
    /* The main program creates 10 threads and then exits. */
    pthread_t threads[NUMBER_OF_THREADS];
    int status, i;

    for(i=0; i < NUMBER_OF_THREADS; i++) {
        printf("Main here. Creating thread %d\n", i);
        status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

        if (status != 0) {
            printf("Oops. pthread_create returned error code %d\n", status);
            exit(-1);
        }
    }
    exit(NULL);
}
```

Different types of threads

- User-Level Thread
 - In user space

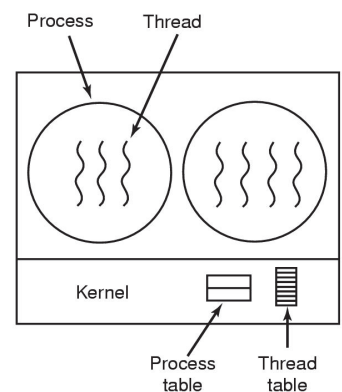


leexudong@nankai.edu.cn

35

Different types of threads

- Kernel-Level Thread
 - In os kernel



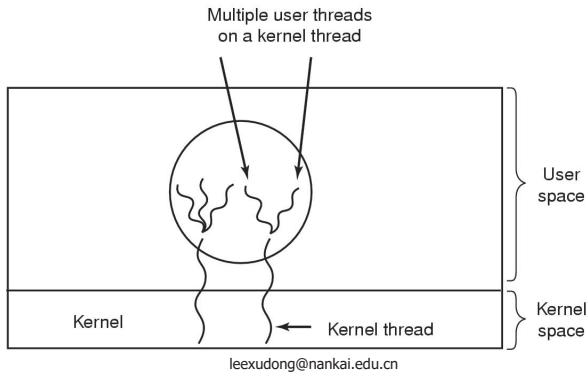
leexudong@nankai.edu.cn

36

Different types of threads

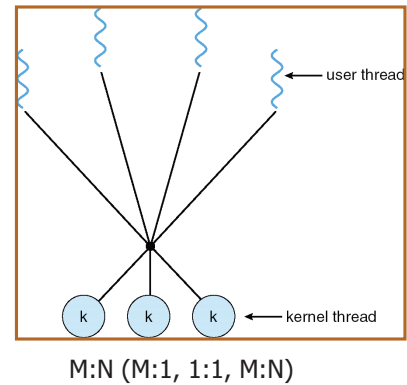
Hybrid Implementation Thread

Both kernel and User Thread



37

Hybrid Implementation Thread I



Hybrid Implementation Thread II

coroutine 协程 (M:N)

Golang

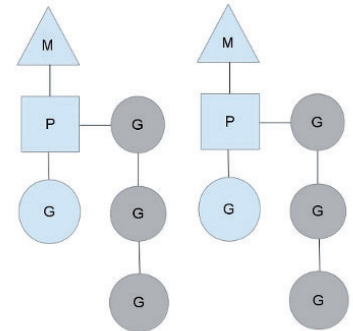


- GoLang 的调度器内部有三个重要的结构 :M, P, S
- M: 代表内核级 OS 线程 ;
- G: 代表一个 goroutine, 即用户级线程, 它有自己的栈、instruction pointer 和其他信息 (channel 等);
- P: 进程在用户级别的调度器, 使 go 代码在一个线程上跑, 它是实现从 M:1 到 M:N 映射.

Hybrid Implementation Thread II

coroutine 协程 (M:N)

Golang



Example: goroutine

```
package main
import "fmt"

func loop(s_pre string, ch chan bool) {
    for i := 0; i < 20; i++ {
        fmt.Printf("%s_%d ", s_pre, i+1)
    }
    ch <- true
}

func main() {
    done := make(chan bool, 2)
    fmt.Println("Hello, 世界")
    go loop("A", done)
    loop("B", done)
    <-done
    <-done
    close(done)
    fmt.Println("\nEnd.")
}
```

leexudong@nankai.edu.cn

41

Threading Issues

- Semantics of fork() and exec() system calls
- Thread cancellation
- Signal handling
- Thread pools
- Thread specific data
- Scheduler activations

leexudong@nankai.edu.cn

42

Semantics of fork() and exec()

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

leexudong@nankai.edu.cn

43

Thread cancellation

- Terminating a thread before it has finished
- Two general approaches:
 - Asynchronous cancellation terminates the target thread immediately
 - Deferred cancellation allows the target thread to periodically check if it should be cancelled

leexudong@nankai.edu.cn

44

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 particular event
 - Signal is delivered to a process
 - Signal is handled
- Options:
 - 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 the process
 - Assign a specific thread to receive all signals for the process

leexudong@nankai.edu.cn

45

Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
 - Usually slightly faster to service a request with an existing thread than create a new thread
 - Allows the number of threads in the application(s) to be bound to the size of the pool

leexudong@nankai.edu.cn

46

Pop-Up Threads

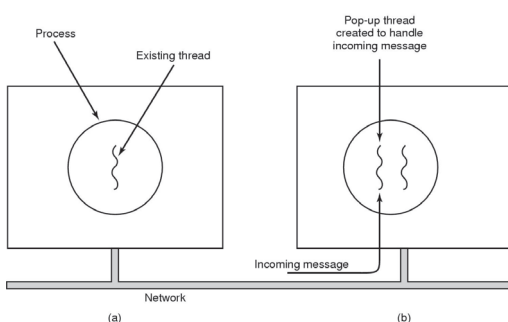


Figure 2-18. Creation of a new thread when a message arrives.
(a) Before the message arrives. (b) After the message arrives.

leexudong@nankai.edu.cn

47

Thread specific data

- Allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

leexudong@nankai.edu.cn

48

Making Single-Threaded Code Multithreaded

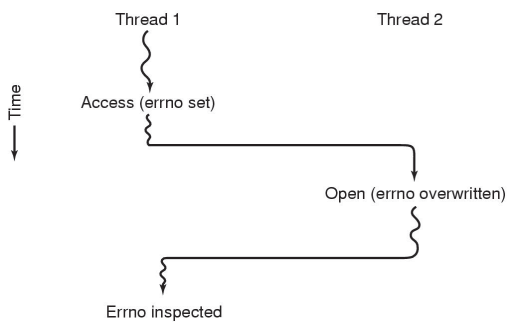


Figure 2-19. Conflicts between threads over the use of a global variable.

Making Single-Threaded Code Multithreaded

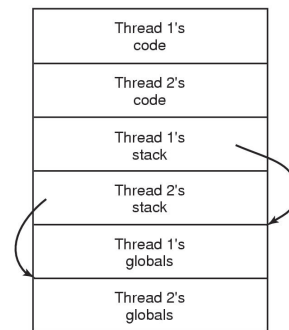


Figure 2-20. Threads can have private global variables.

Thread Scheduler Activations

- Goal
 - mimic the functionality of kernel threads,
 - but with the better performance and greater flexibility usually associated with threads packages implemented in user space
 - Avoiding unnecessary transitions between user and kernel space
- Upcall
 - Virtual processors
 - Run-time system
 - ? layer n may not call procedures in layer n + 1

leexudong@nankai.edu.cn

51

Summary

- Process
- Thread
- Process v.s. Thread
- Kernel Threads v.s. User Threads
- Heavyweight Process v.s. Lightweight Process v.s. Fiber

leexudong@nankai.edu.cn

52

Q&A?

Quiz

- Which of the following OSes do not support threads?
 - Macintosh
 - Windows NT
 - Windows95~Windows2000
 - Solaris
 - IRIX
 - AIX
 - OS/2
 - Digital UNIX
 - Linux

leexudong@nankai.edu.cn

54

